

Linux编程及应用

主讲：任继平

邮箱：rjp@mail.hzau.edu.cn

Linux

- ----- What?
- ----- Why?
- ----- How?

What?

**系统级软件---操作系统
Unix家族---Unix的小弟**

Why?

稳定
安全
可靠
开源

.....

How?

深入源代码
多实践

如何保证?

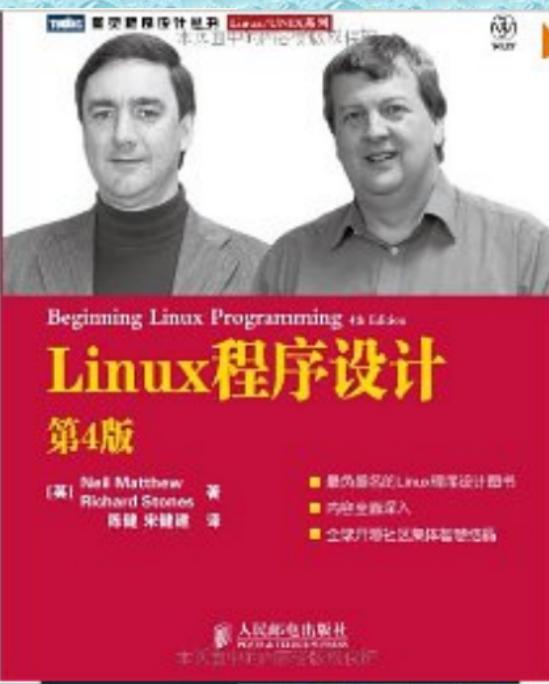
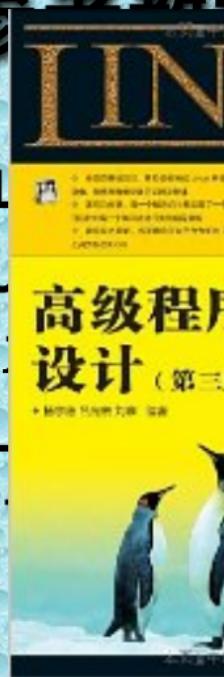
- * 上课考勤
- * 交电子版作业
- * 上课提问
- * 课后答疑

.....

如何考核？

Subject	Percentage
平时表现	30%
实验成绩	30%
期末考试	70%

参考教材



出版社 2011
拉戈 著,
著, 陈健

Linux编程及应用主要知识点

- (1) 补充：Linux下Shell编程
- (2) 磁盘文件（普通文件，链接文件，目录）管理。
 - 教材第4,5, 6章。
- (3) 进程及进程间通信。
 - 教材本书第7,8, 9章。
- (4) 线程及线程间同步。
 - 教材第10,11章。
- (5) 网络编程。
 - 教材第12,13,14,15章。
- (6) 其它：编程工具，编程环境。
 - 教材第1,2, 3章。

磁盘文件管理主要内容

- 普通文件IO操作。
 - ANSI C文件IO。文件描述符及相关操作。
 - POSIX 文件IO。文件流及相应操作。
- 目录文件管理。
 - 目录流及目录流操作。
- 符号链接文件管理。
 - 符号链接及操作。
- 磁盘文件属性获取与磁盘文件属性修改。

进程及进程间通信机制主要知识点

- 进程管理
 - 创建
 - 执行新代码
 - 退出
 - 等待
- 进程间通信
- 数据传递：
 - 管道（有名，无名管道）
 - IPC的消息队列
 - IPC共享内存
- 同步
 - 信号量
- 异步
 - 信号

线程与线程同步基本知识点

- 线程基本操作
 - 创建
 - 退出
 - 取消
 - 等待
- 线程同步机制
 - 互斥锁
 - 读写锁
 - 条件变量
 - 线程信号灯
- 线程与信号

网络编程知识点

- 网络基础及支撑函数，工具
 - TCP/IP协议栈，数据封包拆包过程，TCP，UDP，IP包头
 - BSD TCP，UDP网络编程流程及API函数
 - 地址处理函数，大小端问题，socket属性控制
 - 域名解析
- TCP高级
 - 阻塞与非阻塞处理
 - 多路复用
 - 信号驱动
- UDP高级
 - 广播
 - 组播

编程工具及编程环境知识点

- 编辑器，编译调试工具使用。
 - VIM
 - GCC
 - GDB
 - Makefile
- 头文件，库文件的使用，库文件的创建。
- 错误，帮助信息的获取，编译规范要求。
- 段域加载，内存管理基础知识。

第3章 Linux进程存储管理

1

Linux程序存储结构与进程结构

2

ANSI C内存管理API函数

3

常用Linux内存管理及调试工具

4

Linux进程环境及系统限制

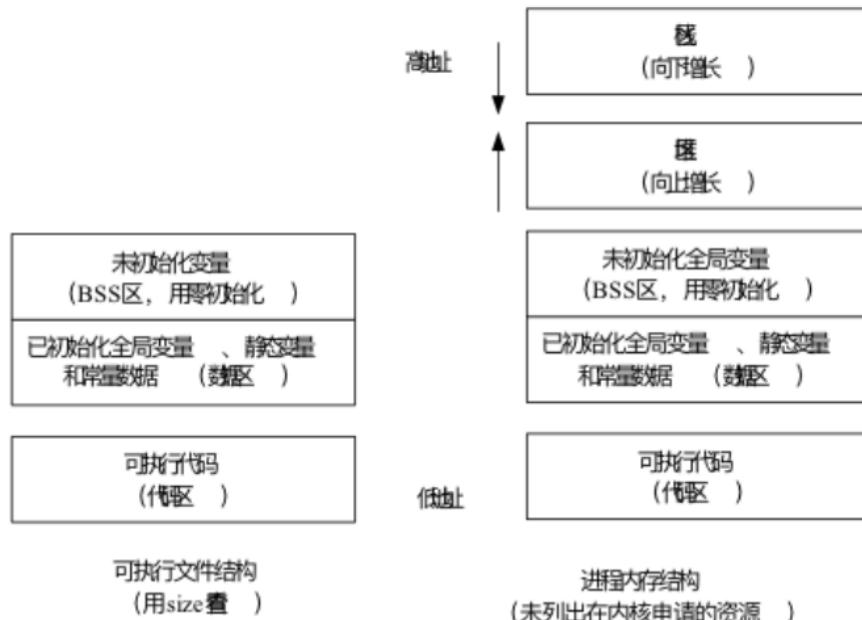
二进制可执行文件

```
[root@localhost Ctest]# ls test -l          //test为一个可执行程序
-rwxr-xr-x 1 root root 4868 Mar 26 08:10 test

[root@localhost Ctest]# file test           //此文件基本情况
test: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.5, dynamically linked
(uses shared libs), not stripped
```

```
[root@localhost Ctest]# size test           //此二进制可执行文件结构情况
//代码区    静态数据/全局初始化数据区  未初始化数据区  十进制总和  十六进制总和  文件名
text      data                      bss      dec      hex      filename
906       284                       4        1194    4aa      test
```

可执行文件与进程存储布局



各段说明

- (1) 代码区 (text segment)。加载的是可执行文件代码段，其加载到内存中的位置由加载器完成。
- (2) 全局初始化数据区/静态数据区 (Data Segment)。加载的是可执行文件数据段，存储于数据段（全局初始化，静态初始化数据）的数据的生存周期为整个程序运行过程。
- (3) 未初始化数据区 (BSS)。加载的是可执行文件BSS段，位置可以分开亦可以紧靠数据段，存储于数据段的数据（全局未初始化，静态未初始化数据）的生存周期为整个程序运行过程。
- (4) 栈区 (stack)。由编译器自动分配释放，存放函数的参数值、返回值、局部变量等。在程序运行过程中实时加载和释放，因此，局部变量的生存周期为申请到释放该段栈空间。
- (5) 堆区 (heap)。用于动态内存分配。堆在内存中位于BSS区和栈区之间。一般由程序员分配和释放，若程序员不释放，程序结束时有可能由OS回收。

C各存储类型比较

类型	作用域	生存域	存储位置
auto 变量	一对 { } 内	当前函数	变量默认存储类型，存储在栈区
extern 函数	整个程序	整个程序运行期	函数默认存储类型，代码段
extern 变量	整个程序	整个程序运行期	初始化在 data 段，未初始化在 BSS 段
static 函数	当前文件	整个程序运行期	代码段
static 全局变量	当前文件	整个程序运行期	初始化在 data 段，未初始化在 BSS 段
static 局部变量	一对 { } 内	整个程序运行期	初始化在 data 段，未初始化在 BSS 段
register 变量	一对 { } 内	当前函数	运行时存储在 CPU 寄存器中
字符串常量	当前文件	整个程序运行期	数据段

栈和堆的区别

- (1) 管理方式不同。
- (2) 空间大小不同。
- (3) 产生碎片不同。
- (4) 增长方向不同。
- (5) 分配方式不同。
- (6) 分配效率不同。

数据存储位置

```
#include <stdlib.h>

int a = 0; //a在全局已初始化数据区

char *p0; //p0在BSS区（未初始化全局变量）

int main(void)
{
    int b; //b在栈区

    char s[] = "abc"; //s在栈区，“abc”在已初始化数据区

    char *p1, *p2; //p1、p2在栈区

    char *p3 = "123456"; //123456\0字符串在已初始化数据区，p3在栈区

    static int c = 0; //C为全局（静态）数据，存在于已初始化数据区

    p1 = (char *)malloc(10); //分配得来的10字节的区域在堆区

    p2 = (char *)malloc(20); //分配得来的20字节的区域在堆区

    free(p1); //释放p1指向的堆区空间

    free(p2); //释放p2指向的堆区空间

    p1=NULL; //将p1置为NULL

    p2=NULL; //将p2置为NULL

}
```

常见内存错误

```
#include <stdio.h>
int* test(void)
{
    int i=10; //返回局部变量地址，这是不允许的
    return &i;
}

int main(void)
{
    int *p;
    p=test();
    printf("*p=%d\n",*p);
    return 0;
}
```

临时空间过大

```
int test()
{
    int a[60][250][1000],i,j,k;           //局部变量a过大， 占用了60MB空间
    for(k=0;k<1000;k++)
        for(j=0;j<250;j++)
            for(i=0;i<60;i++)
                a[i][j][k]=0;
}
```

申请堆空间后未释放

```
char *DoSomething(...)  
{  
    char *p, *q;  
    if ( (p = malloc(1024)) == NULL )  
        return NULL;  
    if ( (q = malloc(2048)) == NULL )  
        return NULL;  
    ...  
    return p;
```

第3章 Linux进程存储管理

- 1 Linux程序存储结构与进程结构
- 2 ANSI C内存管理API函数
- 3 常用Linux内存管理及调试工具
- 4 Linux进程环境及系统限制

malloc/free函数

malloc()函数用来在堆中申请内存空间，声明如下：

```
#include<stdlib.h>
extern void *malloc (size_t __size);
```

malloc()函数在内存动态存储区中分配一个长度为 size 字节的连续空间。返回一个指向所分配的连续存储域的起始地址的指针。当函数未能成功分配存储空间时（如内存不足）返回一个 NULL 指针。

使用完该段内存空间后，需要调用 free() 函数释放原先申请的内存空间。

```
extern void free (void * __ptr);
```

realloc更改已经配置的内存空间

```
extern void *realloc (void * __ptr, size_t __size);
```

- 第一个参数为试图更改大小的原堆空间位置，size为新的内存大小。
- 如果内存减少，malloc仅仅改变索引信息，但并不代表被减少的部分还可以访问，这一部分内存将交给系统内存分配子程序。
- 当需要扩大一块内存空间时，其返回情况如下：
 - 如果当前内存段后面拥有需要的内存空间，则直接扩展这段内存空间，realloc()将返回原指针；
 - 如果当前内存段后面的空闲字节不够，那么就使用堆中第一个能够满足这一要求的内存块，将目前的数据复制到新的位置，并将原来的数据块释放掉，返回新的内存块位置。
- 如果申请失败，将返回NULL，此时原来指针仍然有效。

```
ptr=realloc(ptr,new_amount) //如果分配失败，将使原来的空间位置不可获得
```

内存数据管理函数

memcpy()函数将 n 个字节从 src 所指向的位置拷贝到 dest 所指向位置。其函数声明如下：

```
//come from /usr/include/string.h
```

```
extern void *memcpy (void * __restrict __dest, __const void * __restrict __src, size_t __n);
```

memcpy()函数在实现内存单元拷贝时没有考虑源空间和目的空间有可能重叠的情况， memmove()函数在源码实现上考虑到了这一因素。此函数声明如下：

```
extern void *memmove (void * __dest, __const void * __src, size_t __n);
```

memset()函数将初始化指定内存单元，该函数声明如下：

```
extern void *memset (void * __s, int __c, size_t __n) __THROW __nonnull ((1));
```

此函数将设置自 s 开始后面 n 位的值为 c，如果执行成功，返回 s 的首地址。

memchr()函数将在一段内存空间中查找某个字符位置第一次出现的位置，该函数声明如下：

```
extern void *memchr (__const void * __s, int __c, size_t __n);
```

第3章 Linux进程存储管理

-  **Linux程序存储结构与进程结构**
-  **ANSI C内存管理API函数**
-  **常用Linux内存管理及调试工具**
-  **Linux进程环境及系统限制**

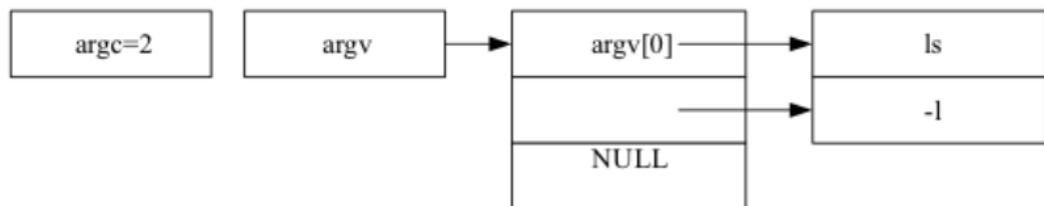
常用工具

- mcheck函数
- Valgrind内存检测工具

第3章 Linux进程存储管理

- 1 Linux程序存储结构与进程结构**
- 2 ANSI C内存管理API函数**
- 3 常用Linux内存管理及调试工具**
- 4 Linux进程环境及系统限制**

命令行参数管理



getopt 获取命令行参数

getopt() 函数用来解析命令行参数，该函数声明如下：

```
extern int getopt (int __argc, char *const *__argv, const char *__shortopts);
```

其第 1 个参数为命令参数的个数（`argc`）、第 2 个参数为指向这些参数的数组（`argv`），第 3 个参数为所有可能的参数字符串（`optstring`）。

环境变量

在 shell 终端下可以通过命令 env 或 set 查看当前系统环境信息。如下所示：

```
[root@localhost ~]# env
HOSTNAME=localhost.localdomain      //主机名
TERM=vt100                          //终端名
SHELL=/bin/bash                      //shell
HISTSIZE=1000                         I          //历史命令大小
SSH_CLIENT=:fffff:192.168.252.1 1297 22
SSH_TTY=/dev/pts/0                    //当前终端是一个网络伪终端
USER=root                            //用户名
```

Linux系统限制

char 数据类型

char 类型数据所占内存空间为 8 位。其中有符号字符型变量取值范围为-128~127，无符号型字符变量取值范围为 0~255。其限制如下：

```
/* Number of bits in a `char'. */  
# define CHAR_BIT 8                                //所占字节数  
/* Minimum and maximum values a `signed char' can hold. */      //有符号字符型范围  
# define SCHAR_MIN    (-128)  
# define SCHAR_MAX    127  
/* Maximum value an `unsigned char' can hold. (Minimum is 0.) */ //无符号字符型范围  
# define UCHAR_MAX   255  
/* Minimum and maximum values a `char' can hold. */
```

Linux系统限制

short int 类型数据所占内存空间为 16 位。其中有符号短整型变量取值范围为 -32768~32767，无符号短整型变量取值范围为 0~65535。其限制如下：

```
/* Minimum and maximum values a `signed short int' can hold. */           // 有符号短整型范围
#define SHRT_MIN (-32768)
#define SHRT_MAX 32767
/* Maximum value an `unsigned short int' can hold. (Minimum is 0.) */      // 无符号短整型范围
#define USHRT_MAX 65535
```

int 数据类型

int 类型数据所占内存空间为 32 位。其中有符号整型变量取值范围为 -2147483648~2147483647，无符号型整型变量取值范围为 0~4294967295U。其限制如下：

```
/* Minimum and maximum values a `signed int' can hold. */                  // 整形范围
#define INT_MIN (-INT_MAX - 1)
#define INT_MAX 2147483647
/* Maximum value an `unsigned int' can hold. (Minimum is 0.) */            // 无符号整形范围
#define UINT_MAX 4294967295U
```

获取/修改系统限制

#define RLIMIT_CPU	0	/* CPU time in ms */
#define RLIMIT_FSIZE	1	/* Maximum filesize */
#define RLIMIT_DATA	2	/* max data size */
#define RLIMIT_STACK	3	/* max stack size */
#define RLIMIT_CORE	4	/* max core file size */
#define RLIMIT_RSS	5	/* max resident set size */
#define RLIMIT_NPROC	6	/* max number of processes */
#define RLIMIT_NOFILE	7	/* max number of open files */
<hr/>		
#define RLIMIT_MEMLOCK	8	/* max locked-in-memory address space */
#define RLIMIT_AS	9	/* address space limit */
#define RLIMIT_LOCKS	10	/* maximum file locks held */
#define RLIM_NLIMITS	11	

Linux时间管理

- UTC时间和Local Time时间的区别：
 - UTC(Universal Time Coordinated)即GMT(Greenwich Mean Time)。
 - Local time为本地时间。
- 系统默认的时区配置文件位置为/etc/sysconfig/clock：

```
[root@localhost ~]# cat /etc/sysconfig/clock
ZONE="Asia/Chongqing"
UTC=true
ARC=false
```

时间管理函数

函数 time() 用来获取当前系统时间，函数声明如下：

```
extern time_t time (time_t *_timer);
```

其时间是自 1970-1-1 0:0:0 以来经历的秒数。如果其参数设置为空，将返回时间秒数，如果参数不为空，将存储于该参数中。

显然，直接使用秒数是不符合人们的习惯的，需要把秒数转换为人们熟悉的时间格式，函数 ctime 将返回当前时间字符串，该函数声明如下：

```
extern char *ctime (_const time_t *_timer);
```

其将时间转换为如下格式：

```
Day Mon dd hh:mm:ss yyyy
```

函数 gmtime 将返回当前时间，其时间基准为 UCT，该函数声明如下：

```
extern struct tm *gmtime (_const time_t *_timer);
```

localtime

```
extern struct tm *localtime (_const time_t * _timer);
```

以上两个函数将返回 struct tm 结构体存储时间，该结构体声明如下：

```
struct tm {  
    int tm_sec;           /* seconds after the minute: [0, 61] */  
    int tm_min;           /* minutes after the hour: [0, 59] */  
    int tm_hour;          /* hours after midnight: [0, 23] */  
    int tm_mday;          /* day of the month: [1, 31] */  
    int tm_mon;           /* month of the year: [0, 11] */  
    int tm_year;          /* years since 1900 */  
    int tm_wday;          /* days since Sunday: [0, 6] */  
    int tm_yday;          /* days since January 1: [0, 365] */  
    int tm_isdst;         /* daylight saving time flag: <0, 0, >0 */  
};
```

如果需要将此时间类型转换为人们习惯的时间字符串，可以调用 asctime 函数，该函数声明如下：

```
extern char *asctime (_const struct tm * _tp);
```

补充二 文件系统

- 一、数据结构
- 二、基本IO函数
- 三、文件与目录函数
- 四、文件锁定
- 五、管道
- 六、超级块与资源管理*
- 七、文件系统管理
- 八、虚拟文件系统*

一、数据结构

文件逻辑结构与读写指针

文件物理结构

分级目录

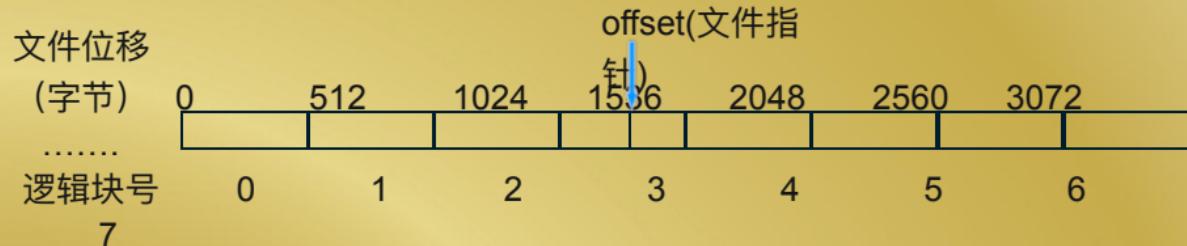
i 节点

磁盘分区和文件系统

文件表与内存i节点表

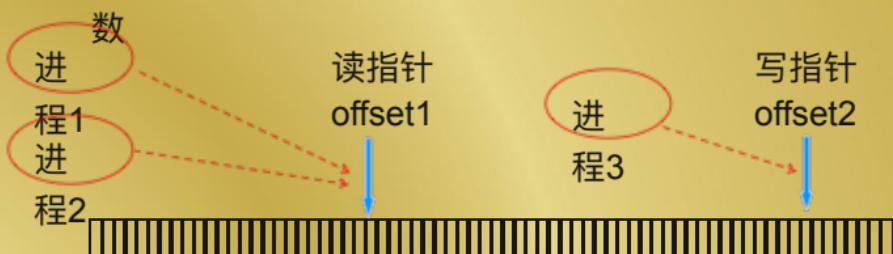
文件类型与访问权限

文件逻辑结构



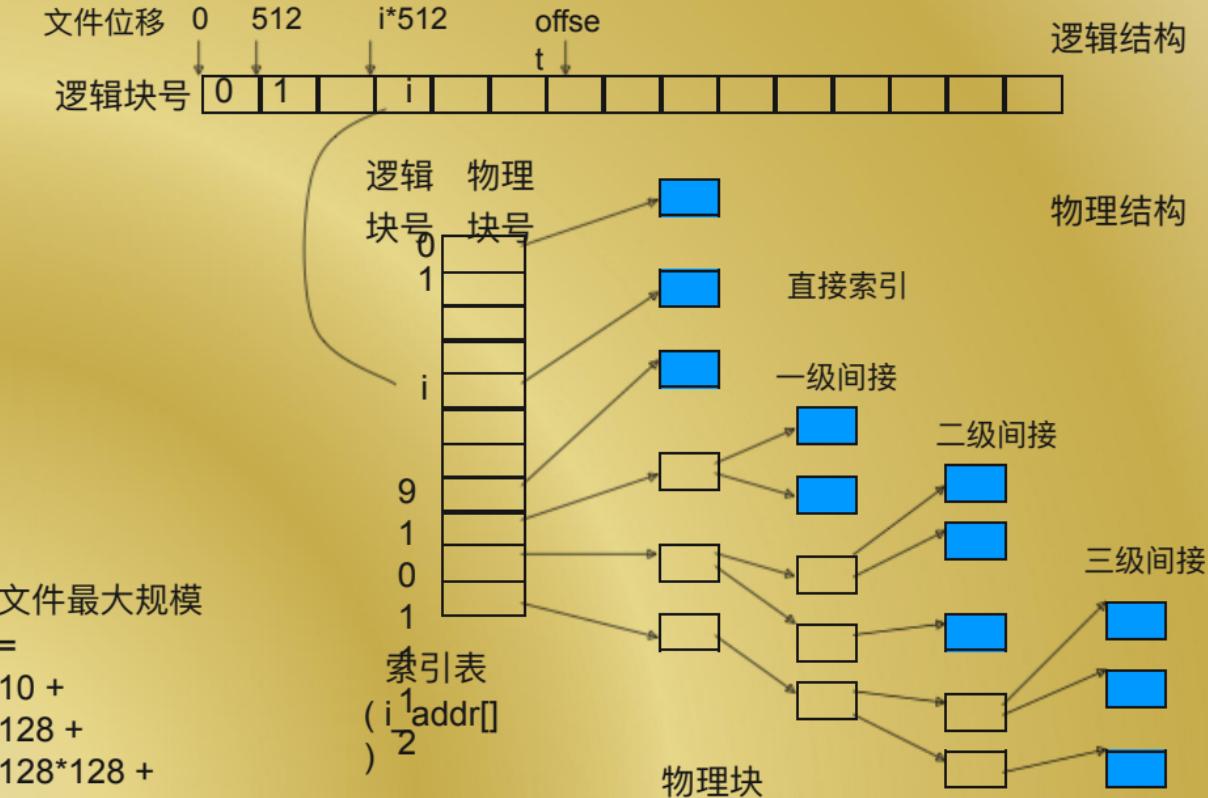
逻辑块号 = (文件指针/块大小) 整数

块内位移 = (文件指针/块大小) 余数



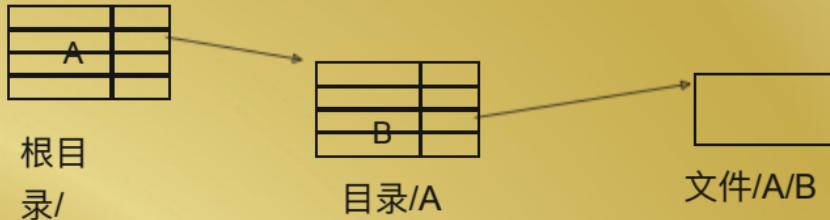
一个文件可以有多个读（或写）指针，
一个指针可由不同进程共享或由一个进程独占。

文件物理结构

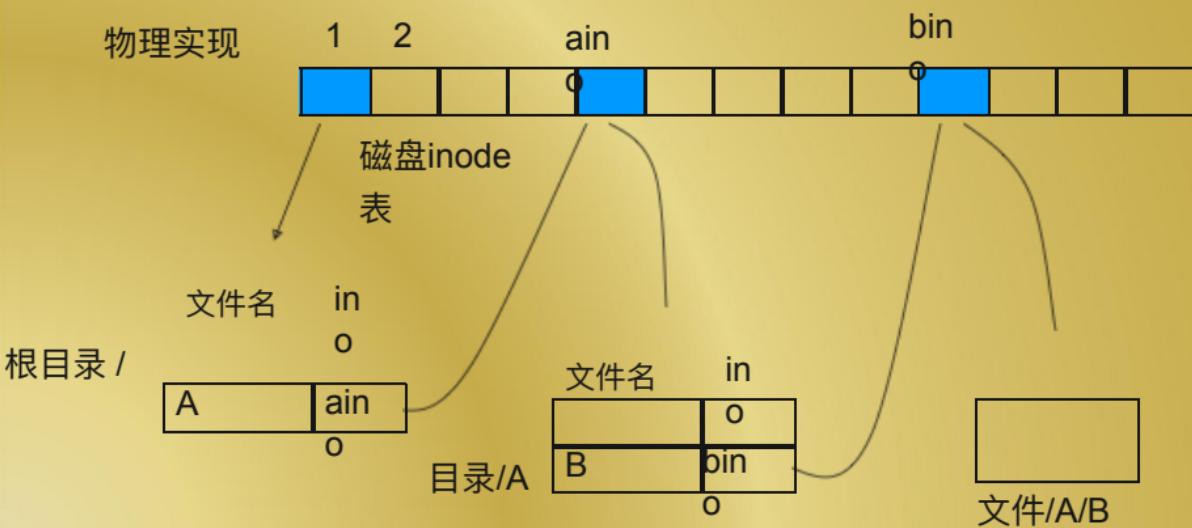


分级目录

逻辑结构



物理实现



i 节点

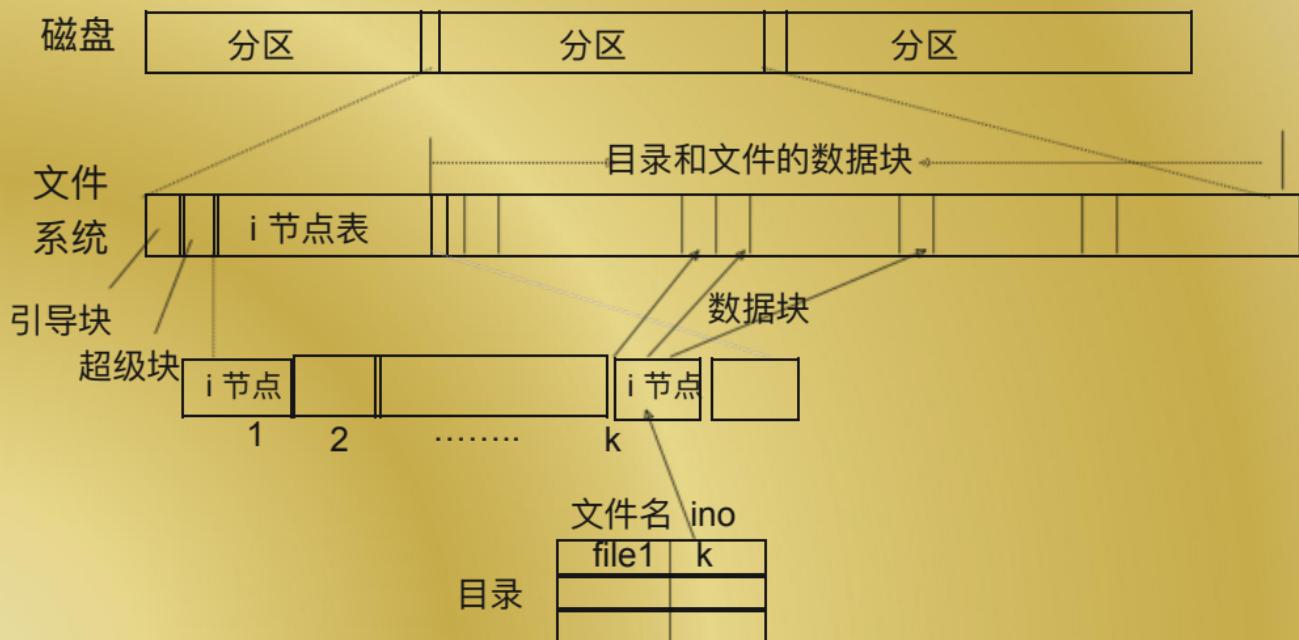
磁盘i节点(index node – inode)含有对应文件的所有说明信息。一个文件系统的所有磁盘i节点组成一个表，保存在超级块之后的确定地址。每个节点按其相对位置顺序编号(如1, 2, ...)，称为i节点号。

磁盘i节点的内容如下：

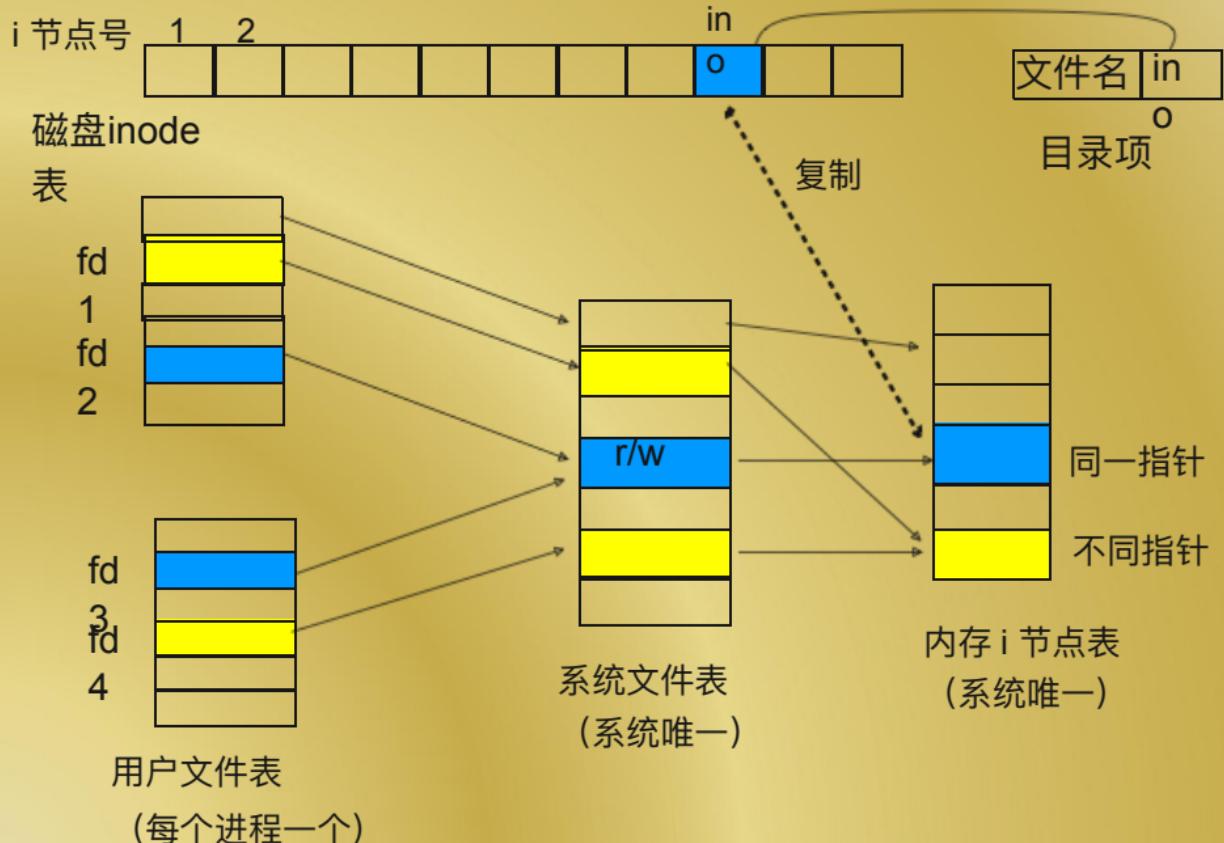
- 。文件主标识号: 用户id(i_uid), 组id(i_gid)
 - 。set_uid位和set_gid位 (i_mode)
 - 。文件类型 (i_mode)
 - 。文件访问权限 (i_mode)
 - 。连接数 (i_nlink)
 - 。文件所在物理块号表 (索引表 i_addr[])
 - 。文件长度 (i_size)
 - 。文件本身设备号 (特殊文件i_rdev)
 - 。文件创建修改访问时间(i_ctime, i_mtime, i_atime)
-

磁盘分区和文件系统

物理磁盘通常划分为若干分区，也称虚拟磁盘或逻辑磁盘。其中可存放操作系统，文件系统或交换区。



文件表与内存 i 节点表



内存 i 节点表

为加快文件名在目录树中的搜索速度设立了内存 i 节点表，其中每一项对应一个打开文件。打开文件时将 i 节点由磁盘复制到内存 i 节点表项中，关闭时再复制回磁盘原处。

文件打开时，磁盘 i 节点复制到内存 i 节点（除时间信息外），但增加如下内容：

- 。设备号（所在设备设备号 i_dev）
- 。i 节点号（所在设备的 i 节点号 i_ino）
- 。访问计数（访问此内存 i 节点的进程数）
- 。使用状态：互斥锁，安装点标志，文件或此节点是否改写
- 。指向 flock 结构的指针

文件关闭时，内存 i 节点除上述内容外的大部分信息复制到磁盘 i 节点上，但增加时间等信息：

整个系统只有一个内存 i 节点表，用以存放系统内所有内存 i 节点。
i 节点的大部分内容可由 stat(2) 和 ls 读取。

系统打开文件表

系统打开文件表简称系统文件表或文件表，此表整个系统只有一个，每项保存有对应打开文件的当前使用特征。共享此打开文件的各进程在自己的打开文件表中各有一表项，其中的指针指向该文件的文件表项。各表项内容如下：

文件状态标志：

读(O_RDONLY), 写(O_WRONLY), 读写(O_RDWR),

附加写(O_APPEND), 非阻塞(O_NONBLOCK),

同步(O_SYNC), 异步(O_ASYNC)。

它们可由open(2) 或 fcntl(2) 设置修改。

当前文件位移量（读写指针），read/write 由此位移开始向后读写文件的数据。读写打开时位移量缺省值为0，附加打开时其值为文件尾，打开后可由lseek(2) 重新设置。

i 节点表项指针。

用户打开文件表

每个进程有一个用户打开文件表（或简称用户文件表，描述字表），其大小由内核配置时确定。每一项表示该进程的一个打开文件，表的索引号称为文件描述字（符）。但描述字0, 1, 2有专门含义：

0：标准输入，缺省为键盘输入，可重定向为任一文件；

1：标准输出，缺省为屏幕输出，可重定向为任一文件；

2：出错输出，指定为屏幕输出，不可重定向。

文件描述字（符）可由系统调用creat,open返回，由dup复制；

表项内容如下：

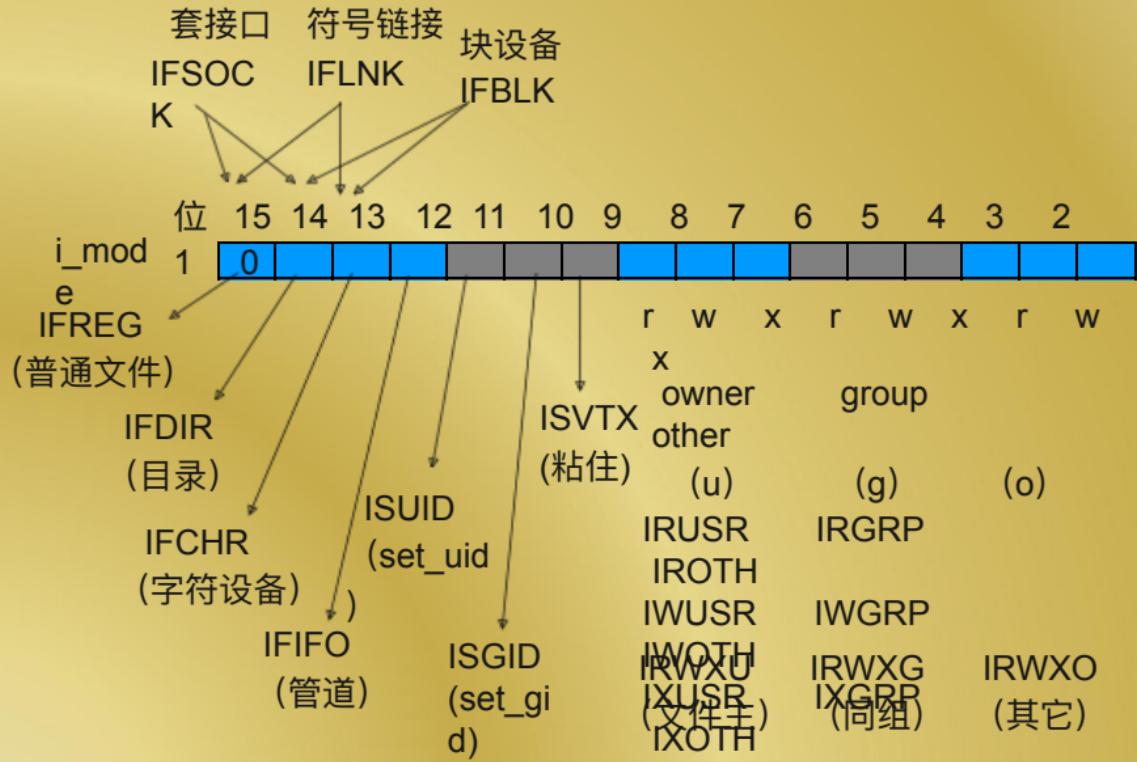
。文件描述字标志：目前只有执行关闭标志(close-on-exec)。

其值为0（缺省）表示当前进程执行一目标文件时不关闭这个打开文件；

1表示执行某文件时关闭此文件。

此标志可由fcntl(2)设置修改。

文件类型与访问权限



文件类型 (12-15位) 及建立的系统调用:

普通文件	(IFREG -1000000)	creat(), mknod()
目录文件	(IFDIR - 0400000)	mkdir()
块设备	(IFBLK - 060000)	mknod()
字符设备	(IFCHR - 020000)	mknod()
套接口	(IFSOCK-140000)	socket()
符号链接	(IFLNK - 120000)	symlink()
管道/FIFO	(IFIFO - 010000)	mknod()

访问许可 (0-8位) :

IRUSR (-000400)	(文件主可读)
IWUSR(-000200)	(文件主可写)
IXUSR (-000100)	(文件主可执行)
IRGRP (-000040)	(同组用户可读)
IWGRP(-000020)	(同组用户可写)
IXGRP (-000010)	(同组用户可执行)
IROTH (-000004)	(其它用户可读)
IWOTH(-000002)	(其它用户可写)
IXOTH (-000001)	(其它用户可执行)

其它 (9-11位) :

ISUID (set_uid位-004000) 设置用户id位。如此位为1则执行此文件后当前进程有效用户号置为文件主用户号。

ISGID (set_gid位-002000) 设置用户组id位。如此位为1则执行此文件后当前进程有效组号置为文件主的组号。

ISVTX (粘住位 -001000) 如此位为1则要求正文段不换出内存，只用于对换系统。

二、基本IO函数

- 打开文件open
- 关闭文件close
- 建立文件creat
- 复制文件描述字dup
- 读写文件read 和 write
- 移动文件指针lseek
- 文件控制fcntl

打开文件open

打开文件：

```
int open (const char *pathname, int oflag,.../* , mode_t mode */);
```

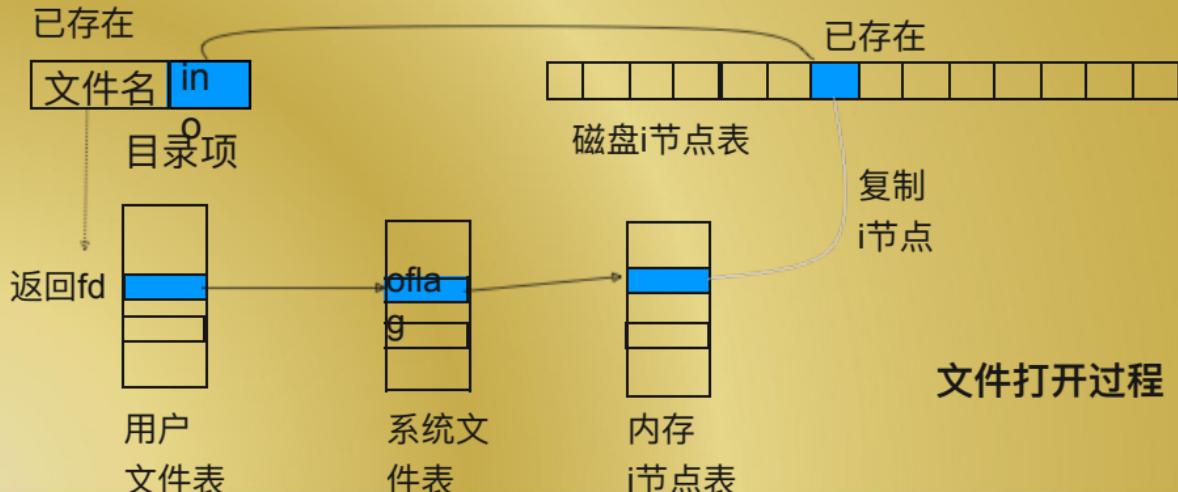
按oflag指定的方式打开 pathname 指定的文件，返回文件描述字；

oflag:	O_RDONLY	/* 只读 */
	O_WRONLY	/* 只写 */
	O_RDWR	/* 读写 */
	O_APPEND	/* 附加写 */
	O_CREAT	/* 如不存在则创建，配合mode参数 */
	O_EXCL	/* 用于原子操作，配合O_CREAT如不存在则创建 */
	O_TRUNC	/* 如只读或只写则长度截为0 */
	O_NOCTTY	/* 不作为控制终端 */
	O_NONBLOCK	/* 非阻塞 */
	O_SYNC	/* 等待物理IO完成 */

open实现算法*

open实现算法：

1. 按路径名搜索分级目录，找到节点读入内存节点表中；
2. 检查访问权限 (i_mode)；
3. 分配填写系统打开文件表项：读写标志(oflag)，文件指针，引用数(+1)，i节点指针；
4. 分配填写用户文件表项：系统文件表项指针；
5. 返回文件描述字（用户打开文件表项索引号）



关闭文件close*

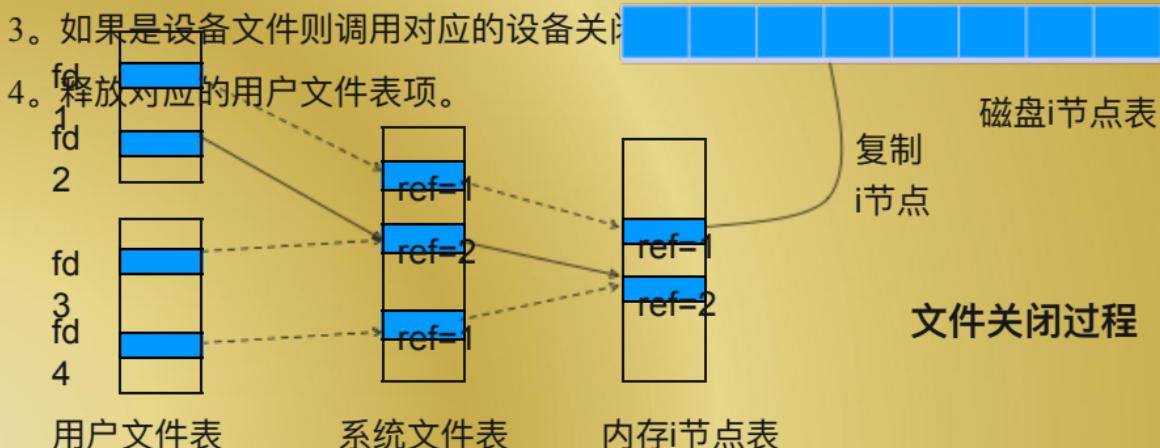
关闭文件：

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

关闭由文件描述符 fd 指向的文件。

实现算法：

- 1。由 fd 找到对应的用户文件表项，系统文件表项和内存 i 节点；
- 2。系统文件表项中的引用数减 1，如结果为 0 则释放此项并将内存 i 节点中的引用数减 1，如结果也为 0 则将此 i 节点写回磁盘 i 节点，释放内存 i 节点；
(如连接数为 0 则释放所有文件块，释放内存 i 节点，置磁盘 i 节点为空闲。)
- 3。如果是设备文件则调用对应的设备关闭函数。



建立文件creat

建立文件：

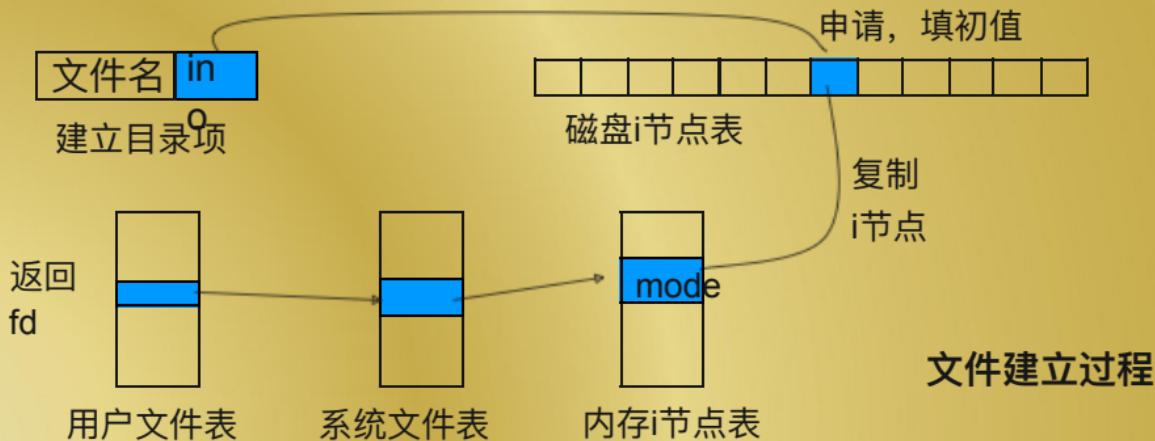
```
int creat (const char *pathname , mode_t mode );
```

creat 建立名字为 pathname 属性为mode 的文件，然后按只写方式打开 pathname 指定的文件，返回文件描述字；其效果等同于：

```
open( pathname, O_WRONLY|O_CREAT|O_TRUNC, mode );
```

若以读写打开方式创建文件则要用open()：

```
open( pathname, O_RDWR|O_CREAT|O_TRUNC, mode );
```



creat实现算法*

实现算法：

1. 按路径名搜索分级目录；
 如文件已存在则检查访问权限 (i_mode) , 如不允许则出错返回;
3. 如文不存在则分配磁盘空闲 i 节点，填初值(mode等), 在父目录中分配新目录项，填入文件名和节点号；
4. 磁盘 i 节点 读入内存 i 节点 (增减部分内容) ; 填写系统打开文件表项 (读写标志=只写, 文件指针=0, 引用数+1, i节点指针) ;
5. 分配填写用户打开文件表项 (文件表项指针) ;
6. 如文件已存在则释放存储块；
7. 返回文件描述字 (用户文件表项索引号)

复制文件描述字dup

复制文件描述字:

```
int dup (int fd);
```

fd 为原fd, 返回用户打开文件表中未用的编号最小的fd。

```
int dup2 (int fd, int fd2);
```

fd 为原fd, fd2指定复制的新的文件描述字。如fd2已打开则关闭后再复制。

例：把标准输入文件(fd = 0) 重定向到 fd 文件：

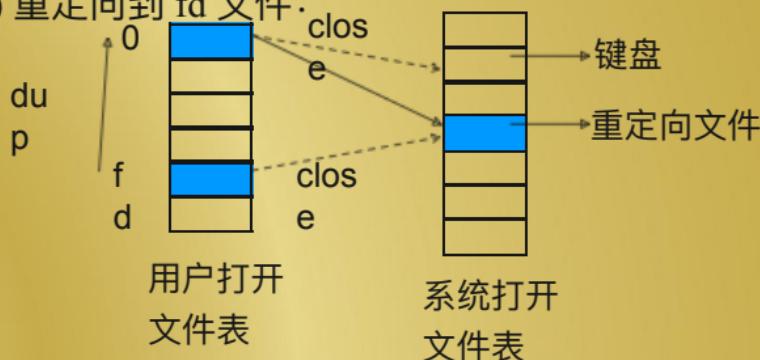
```
dup(fd);
```

```
close(fd);
```

或

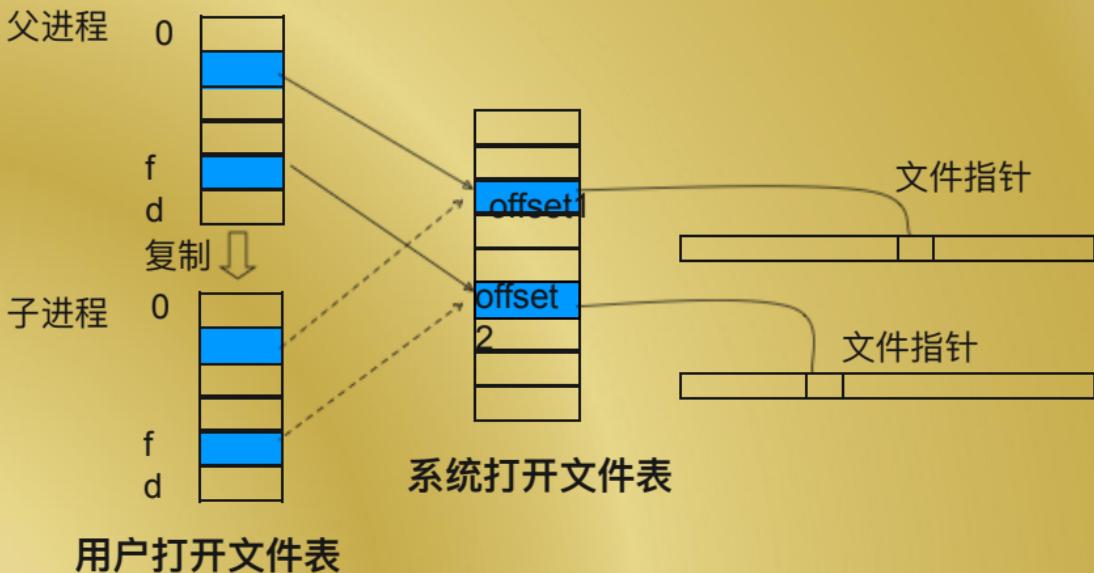
```
dup2(fd,0);
```

```
close(fd);
```



fork与文件描述字

与复制文件描述字有关的另一函数是fork()，它将父进程的全部文件描述字都复制给子进程，因而子进程不仅共享父进程的全部打开文件，也共享它们的文件指针。



读写文件read write

读文件：

```
ssize_t read (int fd, void *buff, size_t nbytes);
```

从文件中文件指针开始读nbytes字节到buff指定的内存区，文件指针后移实际读的字节数，返回实际读的字节数。

fd 文件描述字；

buff 内存地址（通常为数组）；

nbytes 要读的字节数。

写文件：

```
ssize_t write (int fd, void *buff, size_t nbytes);
```

从buff指定的内存区写nbytes到从文件指针开始的文件中，文件指针后移实际写的字节数，返回实际写的字节数。

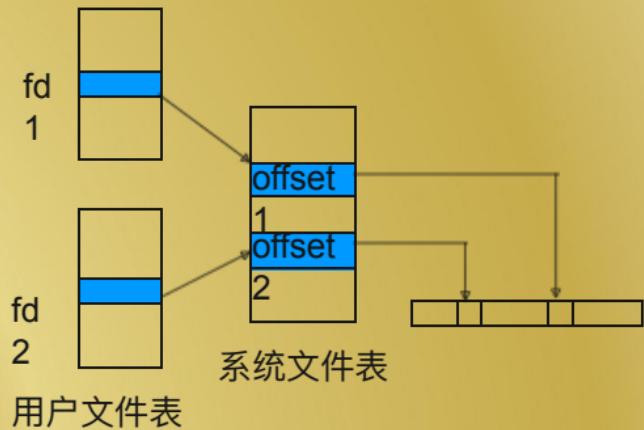
fd 文件描述字；

buff 内存地址（通常为数组）；

nbytes 要写的字节数。

例 1 读写共享

```
/* 进程A和进程B以各自的指针分别读写同  
一个文件 */  
  
#include< fcntl.h >  
  
main()                                /* 进程A */  
{  
    int fd;  
    char buf[512];  
    fd = open( "myfile", O_RDONLY );  
    read ( fd, buf, sizeof(buf) );  
}  
  
main()                                /* 进程B */  
{  
    int fd;  
    char buf[512];  
    for ( i = 0; i < sizeof(buf); i++ )  
        buf[i] = 'a';  
    fd = open ( "myfile", O_WRONLY );  
    write ( fd, buf, sizeof(buf) );  
}
```

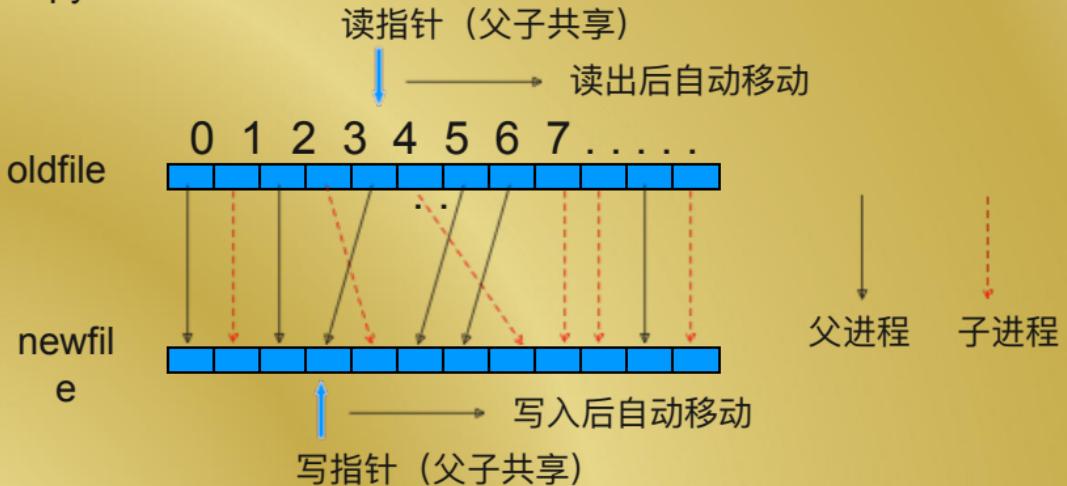


例 2 父子复制

父子两个进程并发地复制同一个文件，它们共享同一个读指针和同一个写指针，由于两者不一定严格同步，因此复制结果可能不确定

。

```
$ copy oldfile newfile
```



```
# include <fcntl.h> /* copy */
int fdrv, dwt;
char c;
int main( int argc, char *argv[ ] )
{
    if( argc !=3 ) exit(1);
    if( fdrv = open ( argv[1], O_RDONLY ) == -1) exit(1);
    if( fdwt = creat ( argv[2], O_0666 ) == -1) exit(1);

    fork();
    rdwrt(); /* 父子进程共享正文段 */
    exit(0);
}
rdwrt()
{
    for( ; ; )
    {
        if( read ( fdrv, &c, 1) != 1) return;
        write ( fdwt, &c, 1);
    }
}
```

read 实现算法*

1. 由fd 找到文件表项，作权限检查，取文件指针；

2. 由文件指针计算文件逻辑块号和块内位移；

 逻辑块号 = [文件指针/块大小]取整数；

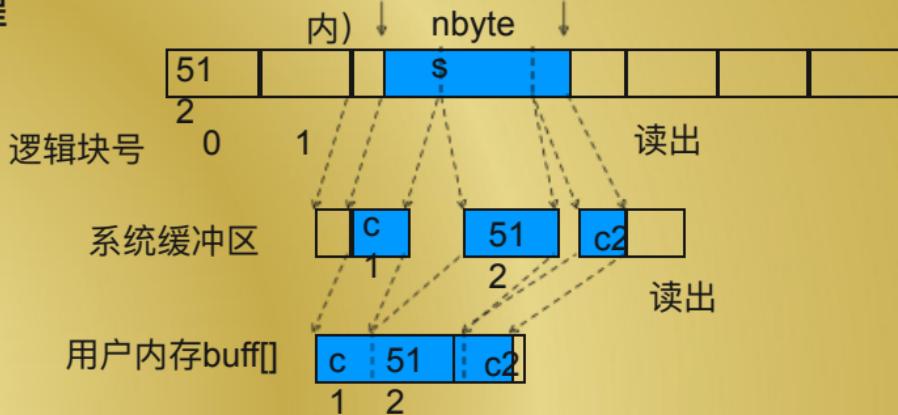
 块内位移 = [文件指针/块大小]取余数；

3. 查i节点表项的索引表，将逻辑块号转换成物理块号；

4. 读出对应物理块到系统缓冲区(bread, breada)，由块内位移开始取出适当的字节数到指定用户内存(buff)；

5. 如果未达到指定字节数 (nbytes) ， 则继续读下一块 (转4。) ， 直至读完；

6. 返回实际读出的字节数。 f_offset (文件指针，在系统文件表项
文件读出过程

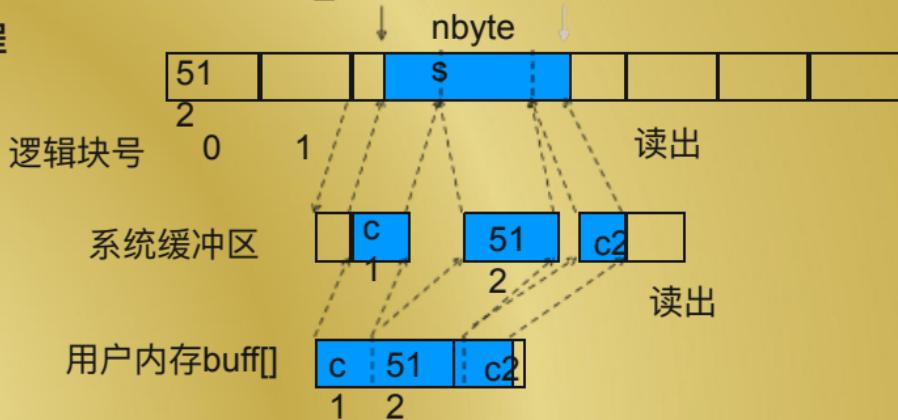


write实现算法*

1. 由fd找到文件表项，作权限检查，取文件指针；
2. 由文件指针计算文件逻辑块号和块内位移；
 逻辑块号 = [文件指针/块大小]取整数；
 块内位移 = [文件指针/块大小]取余数；
3. 查i节点表项的索引表，将逻辑块号转换成物理块号；
4. 由指定用户内存(buff)复制适当的字节数到系统缓冲区中（上述块内位移），可能
 第一块要先读出到缓冲区；然后同步(bwrite)或延迟写(bdwrite)入对应物理块；
5. 如果未达到指定字节数 (nbytes)，则继续写下一块 (转4。)，直至写完；
6. 返回实际写入的字节数。

f_offset (文件指针，在系统文件表项内)

文件写入过程



分散读集中写

Int writev(int fd, const struct iovec *iov, int iovcount);

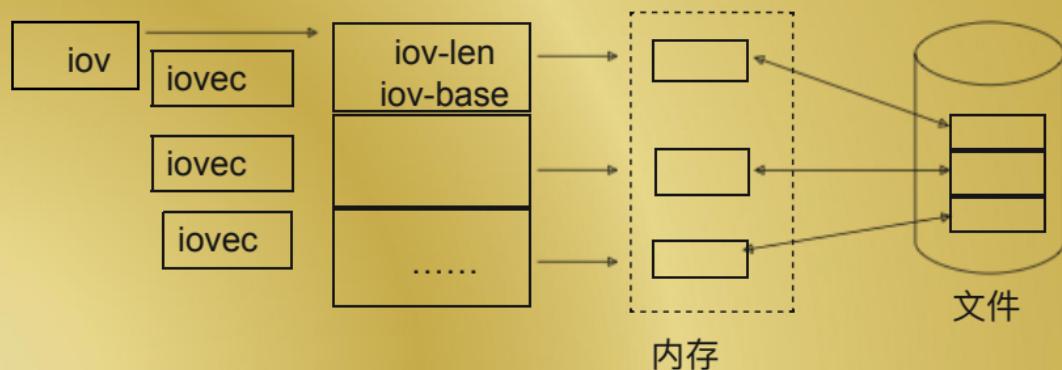
Int readv(int fd, const struct iovec *iov, int iovcount);

```
struct iovec{
```

```
    char *iov_base; 缓冲区起始地址
```

```
    Int     iov_len; 缓冲区长度
```

```
};
```



移动文件指针lseek

移动文件指针：

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

文件读写指针设置为(whence,offset)指定的位置(可以超过文件长度，基延长文件长度，空洞部分值为0)，返回新指针值。

whence: 起点

SEEK_SET(0) /* 文件头 */

SEEK_CUR(1) /* 当前文件读写指针所在位移 */

SEEK_END(2) /* 文件尾 */

offset: 位移量

文件读写指针的缺省值：读写打开时为0，附加写打开时为文件尾。

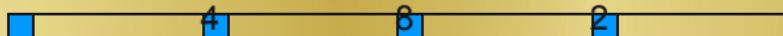


例 3 间隔读字符

```
#include <fcntl.h>
main(int argc, char *argv[])
{
    int fd, skval;
    char c;

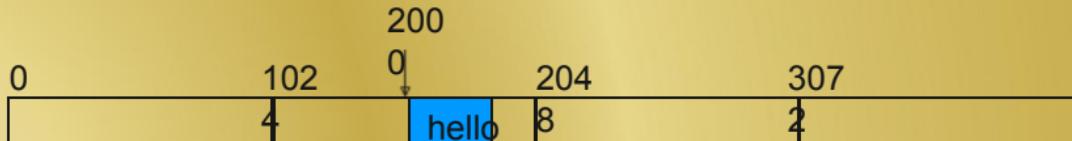
    if( argc != 2) exit();
    fd = open(argv[1], O_RDONLY);
    if( fd == -1) exit();
    while ( ( skval = read(fd, &c, 1));
    {
        printf("char %c \n" , c);
        skval = lseek(fd, 1023L,1);
        printf("new seek val %d \n" ,skval);
    }
}
```

0 102 204 307



例 4 读一段字符

```
# include <fcntl.h >
main()
{
    int fd;
    char buf [ 1024 ];
    fd = creat ( “myfile”, 0666 );
    lseek( fd, 2000L, 0 );
    write( fd, “hello”,5 );
    close(fd);
    fd = open( “myfile”, O_RDONLY );
    read ( fd, buf, 1024 );
    read ( fd, buf, 1024 );
    read ( fd, buf, 1024 );
}
```



文件控制fcntl

文件控制：

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

改变文件的性质。cmd表示五种功能：

F_DUPFD: 复制文件描述字；

F_GETFD / F_SETFD: 获得/设置文件描述字标志FD_CLOEXEC
(默认为0, 表示在进程exec后不关闭该文件)；

F_GETFL : 读文件状态: O_RDONLY (只读), O_WRONLY (只写),
O_RDWR (读写), O_APPEND (附加), O_NONBLOCK (非阻塞IO), O_SYNC (同步), O_ASYNC (异步)。

F_SETFL: 设置文件状态: O_APPEND, O_NONBLOCK, O_SYNC,
O_ASYNC

F_GETOWN / F_SETOWN: 获得/设置接受SIGIO信号 (异步IO) 或SIGURG
信号 (网络紧急数据) 的进程ID或进程组ID；

三。文件与目录函数

- 读取文件信息
- 访问权限
- 文件连接
- 特殊文件

读取文件信息stat

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

```
int lstat ( const char *pathname, struct stat *buf);  返回符号连接信息
```

```
int fstat ( int fd, struct stat *buf);
```

将文件的信息(来自inode)读入 buf, ls 命令需要这个函数。

```
struct stat {  
    mode_t st_mode; /* 文件类型与访问许可 */  
    ino_t st_ino; /* i节点号 */  
    dev_t st_dev; /* 所在设备号 */  
    dev_t st_rdev; /* 特殊文件设备号 */  
    nlink_t st_nlink; /* 连接数 */  
    uid_t st_uid; /* 文件主用户id */  
    gid_t st_gid; /* 文件主组id */  
    off_t st_size; /* 普通文件规模 (字节数) */  
    time_t st_atime; /* 最近访问时间 */  
    time_t st_mtime; /* 最近修改时间 */  
    time_t st_ctime; /* 最近文件状态改变时间 */  
    long st_blksize; /* 最好的IO块规模 */  
    long st_blocks; /* 512字节块数 */
```

在<sys/stat.h>中的文件类型宏（读 i_mode）

宏 文件类型

S_ISREG() 普通文件

S_ISDIR() 目录文件

S_ISCHR() 字符特殊文件

S_ISBLK() 块特殊文件

S_ISFIFO() FIFO

S_ISLNK() 符号连接

S_ISSOCK() 套接字

IFSOK
K

i_mod

位

15 14 13 12 11 10 9 8 7 6 5 4 3 2

1 d

IFREG 普通文件

IFDIR 目录

IFCHR 字符设备

IFBLK 块设备

IFLNK 符号连接

IFIFO 管道

例5. 打印文件类型

```
#include<sys/types.h>
#include<sys/stat.h>
#include<ourhdr.h>
int main(int argc, char *arg[])
{
    int i;
    struct stat buf;
    char *ptr;

    for(i=1; i < argc; i++)  {
        printf( "%s: ", argv[i] );
        if (lstat(argv[i], buf) < 0 ) { err_ret("lstat error"); continue }
        if      (S_ISREG(buf.st_mod))  ptr = " regular ";
        else if (S_ISDIR(buf.st_mod))   ptr = " directory ";
        else if (S_ISCHR(buf.st_mod))   ptr = " character special";
        else if (S_ISBLK(buf.st_mod))   ptr = " block special ";
    }
}
```

```
else if (S_ISFIFO(buf.st_mod))  ptr = " fifo ";
else if (S_ISLNK(buf.st_mod))   ptr = " symbolic link ";
else if (S_ISSOCK(buf.st_mod))  ptr = " socket ";
else                           ptr = " ** unknown mode ";
printf( "%s \n", ptr );
}
exit(0);
}
```

程序打印内容如下：

```
$ a.out /vmLinux /etc /dev/ttya /dev/sd0a /var/spool/cron/FIFO /bin \
/dev/printer
/vmLinux: regular
/etc: directory
/dev/ttya: character special
/dev/sd0a block special
/var/spool/cron/FIFO: fifo
/bin symbolic: link
/dev/printer: socket
```

修改访问权限

Linux的文件保护机制采用访问控制表(access control list)的简化：

访问用户分为三类：文件主(owner User - u)，同组用户(Group - g)，其它用户(Other - o)；

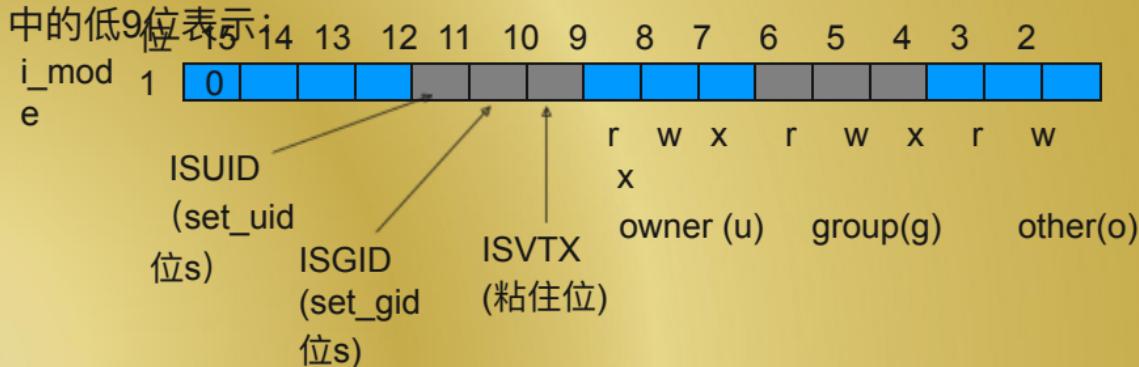
文件访问类型分为三种：读(read - r)，写(write - w)，执行(execute - x)；

目录解释：读(read)：对目录中文件名的列表；

写(write)：

执行(execute)：搜索目录；

对于指定文件3类用户的3种访问类型形成9种组合，用i节点中的i_mode



chmod

```
Int chmod ( const char *pathname, mode_t mode );
```

```
Int fchmod ( int fd, mode_t mode );
```

这两个函数用于更改指定文件的访问权限。chmod针对路径名 pathname 指定的文件，fchmod针对文件描述字 fd 指定的打开文件。

如果更改访问权限则进程的有效用户号必须是文件主的用户号。

mode 取值为如下常数（<sys/stat.h>中定义）表示的逐位或运算：

S_IRWXU (文件主可读, 写, 执行)

S_IRUSR (文件主可读) S_IWUSR (文件主可写) S_IXUSR (文件主可执行)

S_IRWXG (同组用户可读, 写, 执行)

S_IRGRP (同组可读) S_IWGRP (同组可写) S_IXGRP (同组可执行)

S_IRWXO (其它用户可读, 写, 执行)

S_IROTH (其它可读) S_IWOTH (其它可写) S_IXOTH (其它可执行)

S_ISUID (setuid) 设置用户位

例6. 修改文件权限

```
int main(void)
{
    struct stat      statbuf;

    if (stat("foo", &statbuf) < 0)
        err_sys ("stat error for foo");
    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        err_sys ("chmod error for bar");
    if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISUID) < 0)
        err_sys ("chmod error for foo");
    exit(0);
}
```

此程序执行前 ls 产生如下结果：

```
$ ls -l bar foo
-rw----- 1 stevens 1230 Nov 16 16:23 bar
-rwxrwxrwx 1 stevens 3210 Nov 16 16:23 foo
```

此程序执行后 ls 产生如下结果：

```
$ ls -l bar foo
-rw-r--r-- 1 stevens 1230 Nov 16 16:23 bar
-rwxrwxrwx 1 stevens 3210 Nov 16 16:23 foo
```

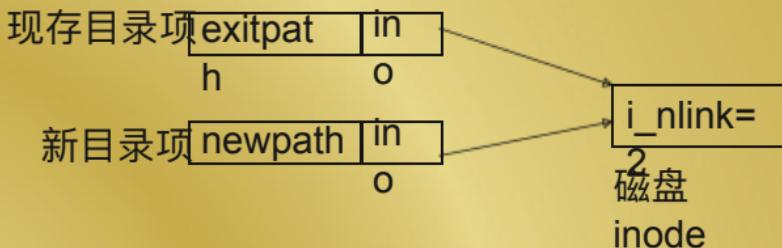
文件连接与删除

```
int link ( const char *existpath, const char *newpath);
```

创建一个新目录项newpath，其中指向现存文件existpath，相当于另起一个文件名。link也称硬连接。

shell 命令： \$ ln existpath newpath

- 限制：
 1. 只有超级用户才能连接目录文件。
 2. 两个文件必须在同一文件系统中（大多数实现）。



- 缺点：
1. 原有文件建立者要想删除文件，难以找到相应的连接者；
 2. 不能跨越不同文件系统。

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

删除目录项 pathname，连接数(nlink)减1。

shell 命令： \$ rm exitpath

 \$ rm newpath

实现 算法：

- 1。搜索文件目录树，找到要删除文件的父目录(namei);
- 2。如果是共享正文文件且连接数为1，则从分区表中清除；
- 3。修改父目录：清除文件名，i 节点置0；
- 4。连接数减1；
- 5。释放i 节点(put)；

(如连接数为0并且 i 节点引用数为1则释放所有文件块，释放内存 i 节点，置磁盘 i 节点为空闲，最终删除此文件)

注意：由于这个原因，删除一个正在打开的文件只能删除它的目录项。仅当以后再关闭该文件时（再次执行put）才会最终释放该节点。下面的例子说明了unlink的这个性质。

例7. 删除打开的文件

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

main(int argc, char **argv)
{
    int fd;
    char buf[1024];
    struct stat statbuf;

    if (argc != 2) exit();
    if((fd = open(argv[1], O_RDONLY ) == -1) exit(); /* 打开文件*/
    if(( unlink(argv[1]) == -1) exit(); /* 删除打开文件 */
    if(stat( argv[1], $statbuf) == -1) /* 用文件名查文件状态 */
        printf("stat %s fails as it should \n", argv[1] );
    else
        printf("stat %s succeeded!!! \n", argv[1] );
```

```
if(fstat( argv[1], $statbuf) == -1) /* 用 fd 查文件状态 */
    printf("stat %s fails!!! \n", argv[1]);
else
    printf("stat %s succeeded as it should \n", argv[1]);
        /* 应该成功 */
while (read( fd, buf, sizeof(buf) ) > 0)
    printf ("%s1024s", buf);      /* 读写应该成功 /
}
}
```

此程序退出(exit)时会关闭所有文件，由于这里的文件已删除，所以会最后清除这个文件。即使此程序异常退出也不会留下未清除的文件。利用unlink的这一特点，用户可以打开文件后立即删除它。以后用户可以正常读写这个文件，不用担心忘记删除它。

符号连接

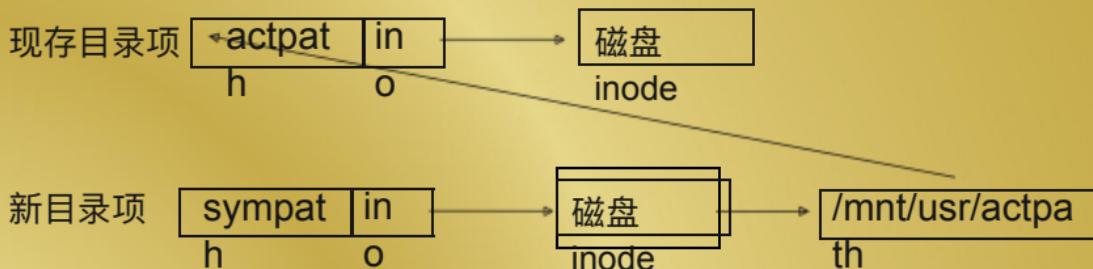
符号连接克服了硬连接的两个缺点。它为新的连接另外申请了一个I节点，并在它指向的文件中保存了原来的文件名。

```
int symlink ( const char *actpath, const char *sympath);
```

actpath 为现存文件名；sympath 为新的符号连接名。

shell 命令： \$ ln -s actpath sympath

执行此操作时不要求actpath文件已经存在，两个文件名可以不在一个文件系统上。



符号连接缺点：

1. 额外的扫描路径；
2. 额外的I节点；
3. 额外的存储（存文件名）。

特殊文件

系统调用

```
Int mknod ( const char *pathname, mode_t mode, dev_t dev);
```

建立设备特殊文件，fifo文件或普通文件，但不打开。文件主为有效用户号。

pathname: 文件路径名；

mode: 文件类型与访问权限；其中类型如下：

S_IFCHR: 字符特殊文件；

S_IFBLK: 块特殊文件；

S_IFIFO: 命名管道 (fifo) 文件；

S_IFREG: 普通文件；

dev: 设备号（包括主设备号和次设备号）；

命令

```
mknod name type major minor
```

name: 文件名

type: 文件类型：c, u: 字符特殊文件；b: 块特殊文件；p: fifo文件；

major: 主设备号；

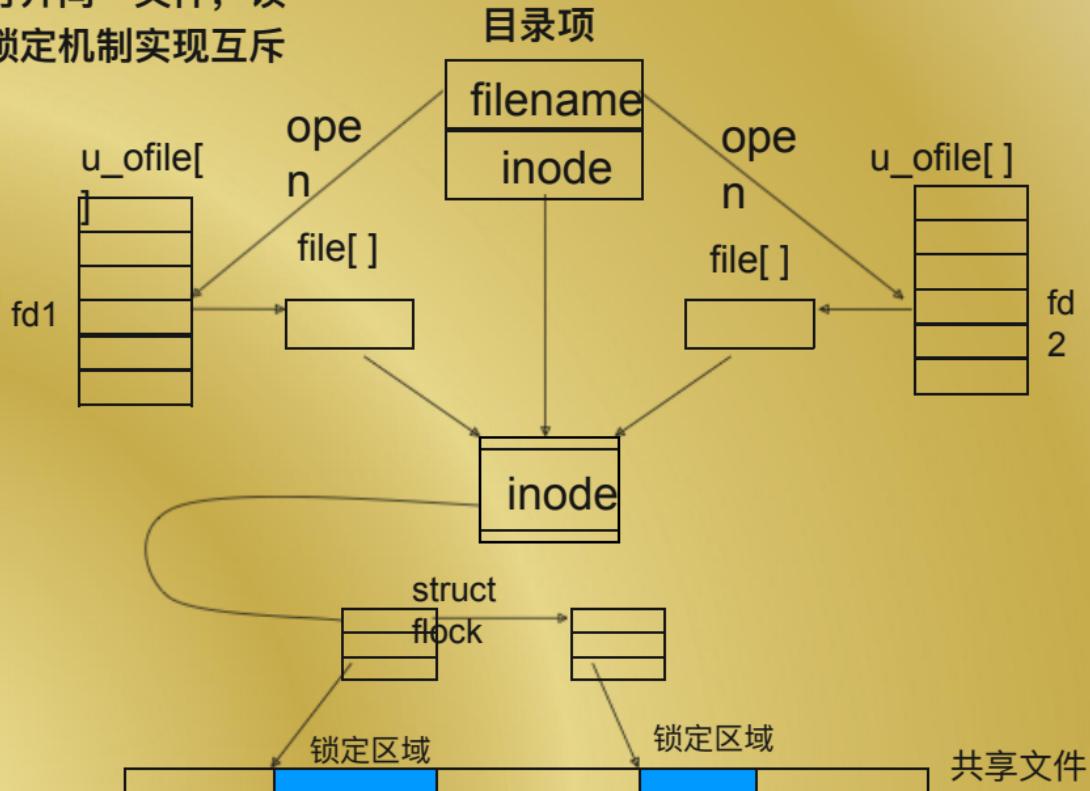
minor: 次设备号；

四。文件锁定

- 概述*
- **fcntl(2)**
- **lockf(3)**
- **flock(2)**

概 述*

通信进程打开同一文件，读
写时文件锁定机制实现互斥



fcntl(2)

fcntl 实现记录锁

```
int fcntl (int fd, int cmd, struct flock *flockptr) ;
```

cmd:

F_SETLK 设置锁，如不允许则返回错；

F_SETLKW 设置锁，如不允许则等待；

F_GETLK 测试能否建立锁；

```
struct flock { /* 指定锁类型及锁住的区域 */
```

```
    short l_type; /* 锁的类型 */
```

```
    /* F_RDLCK(共享读) , F_WRLCK(独占写) , F_UNLCK(解锁) */
```

```
    off_t l_start; /* 同lseek */
```

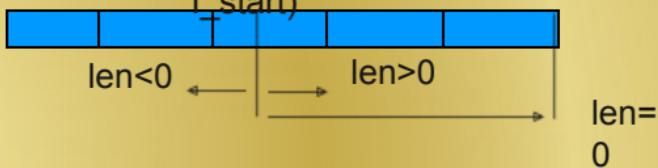
```
    short l_whence; /*同lseek */
```

```
    off_t l_len; /* 锁定区域的范围，区域起址由前两项决定 */
```

```
    pid_t l_pid; /*由 F_GETLK返回 */
```

```
}
```

指定位移 (l_whence,
l_start)



例8. 加锁与解锁

```
void my_lock(int fd) /* 加锁 */
{
    struct flock lock;
    lock.l_type = F_WRLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 0;
    lock.l_len = 0;
    fcntl(fd, F_SETLKW, &lock);
}

void my_unlock(int fd) /* 解锁 */
{
    struct flock lock;
    lock.l_type = F_UNLCK;
    lock.l_whence = SEEK_SET;
    lock.l_start = 0;
    lock.l_len = 0;
    fcntl(fd, F_SETLK, &lock);
}
```

lockf(3)

system V 库函数，调用fcntl(2)实现。

```
int lockf(int fd, int cmd, off_t len);
```

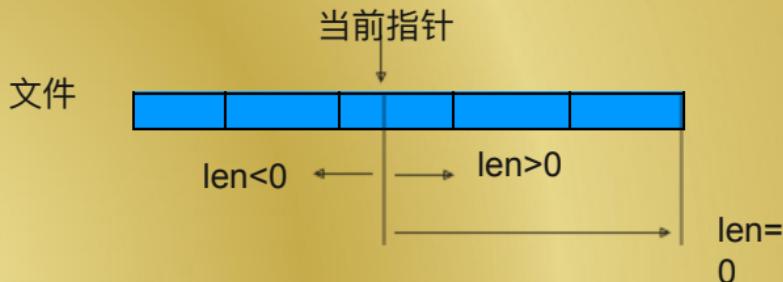
cmd: F_UNLOCK 解锁

F_LOCK 加锁 (如上锁则等待)

F_TEST 测试 (如上锁则返回-1)

F_TLOCK 相当于 F_TEST + FLOCK 原子操作 ($len > 0$)

len: 加锁范围：当前指针开始（向前/向后）的一段区域。



例9. 加锁与解锁

```
my_lock(int fd)
{
    lseek(fd,0l,0);
    If (lockf (fd, F_LOCK, 0l) == -1) /*封锁整个文件*/
        err_sys(".....");
}
my_unlock(int fd)
{
    lseek(fd,0l,0);
    if(lockf(fd,F_UNLOCK,0L) == -1)
        err_sys(".....");
}
```

flock(2)

4.3 BSD 系统调用

int flock(int fd,int cmd);(2) 封锁整个文件

cmd:

LOCK_SH; 共享锁

LOCK_EX; 互斥锁

LOCK_UN; 解锁

LOCK_NB; 不阻塞锁定

例：

```
mylock(int fd)
{
    if ( flock(fd, LOCK_EX) == -1)
        err_sys(".....");
}
```

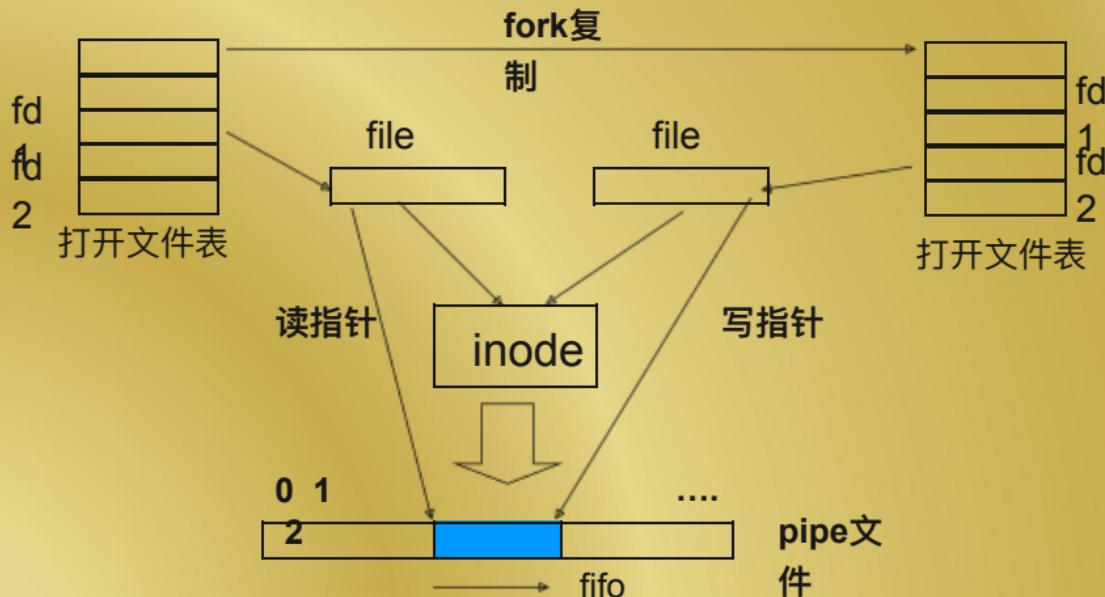
```
myunlock(int fd)
{
    if ( flock(fd,LOCK_UN) == -1)
        err_sys(".....");
}
```

五。管道

- 无名管道
- 有名管道

无名管道

父进程打开文件， fd用fork传递 给子进程, 读写进程（同一家族）自动同步



pipe特点

特点

1. 无文件名
2. 先进先出（自动同步检查实现）单向
3. 临时文件
4. 同一家族进程

系统调用：

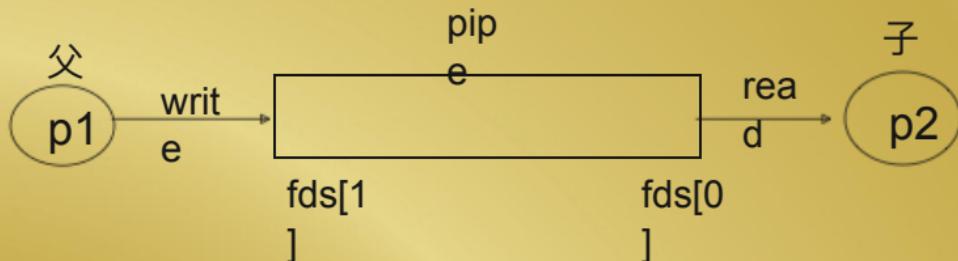
```
int pipe (int fds[2]);
```

返回fds[0]为读描述字， fds[1]为写描述字

shell应用：

```
$ ls|wc
```

编程方法



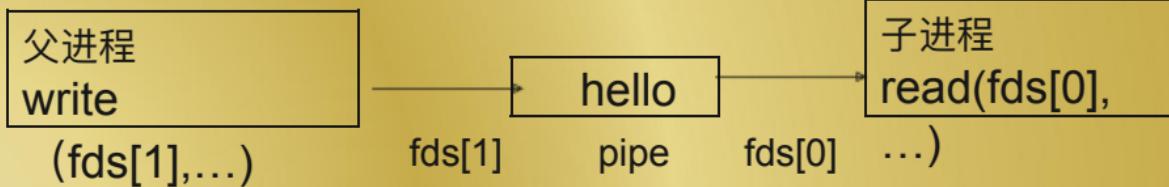
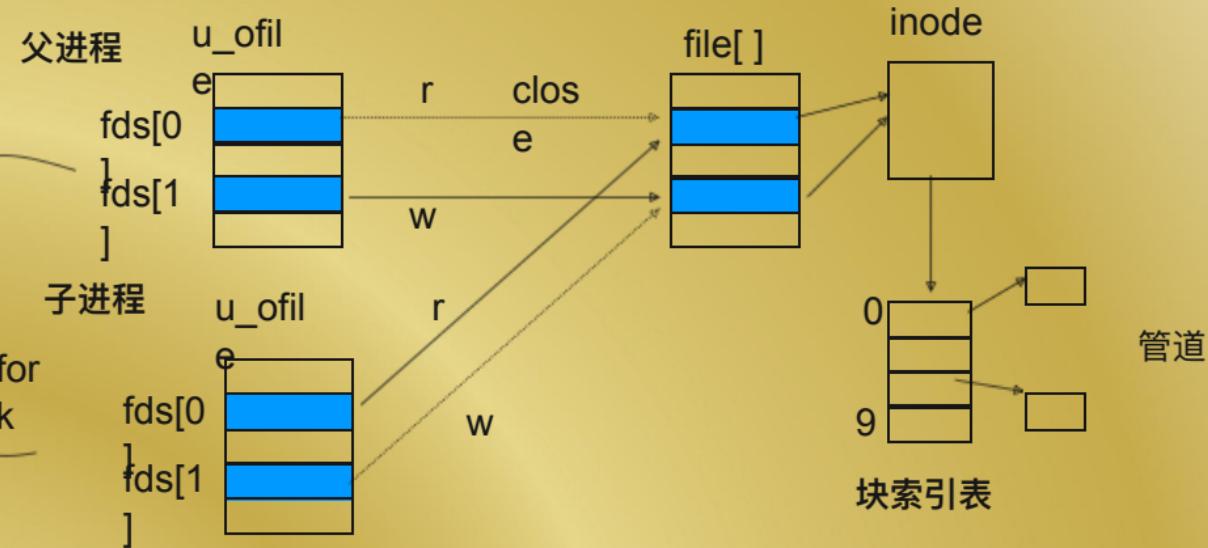
步骤：

1. 父进程 `pipe (fds[])` ;
2. `fork` 建立子进程；
3. 父子分别关闭`fds[0]`, `fds[1]`; 或`fds[1]`, `fds[0]` (相反)
4. 父子分别`write(fds[1])` , `read(fds[0])` ; 或相反

一端关闭时对另一端的影响：

如写关闭，则读`read`返回0；

如读关闭，则写`write`产生信号`SIGPIPE`。



例10. pipe

```
char string = "hello";
main()
{
    char buf[10];
    int fds[2], count;
    pipe[fds];

    if(fork() == 0)
    {
        /* 子进程从管道读出“hello” */
        if((count = read(fds[0],buf,sizeof(buf[]))) == 0)
            exit(0);
        printf("read%dbytes:%s\n", count, buf);
    }

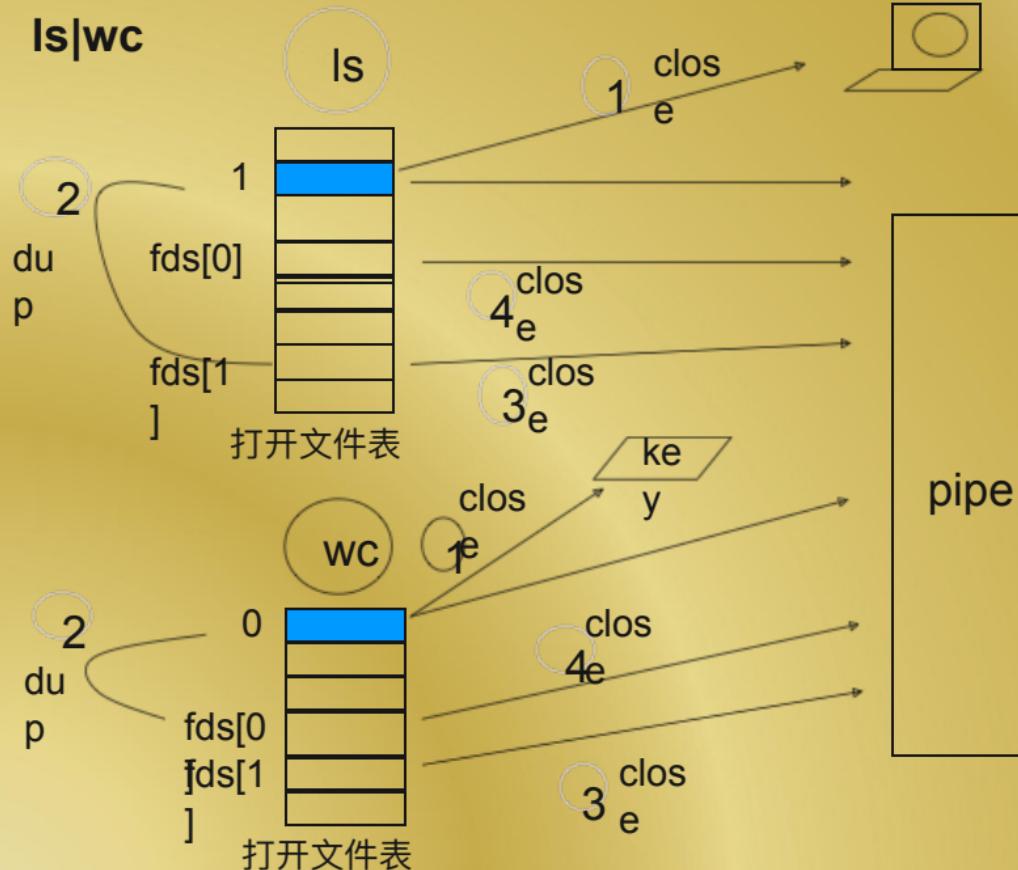
        /* 父进程向管道写入“hello” */
    if((count = write(fds[1], string, strlen(string))) == 1)
        exit(1);
}
```

shell管道线的实现

例： \$ ls|wc

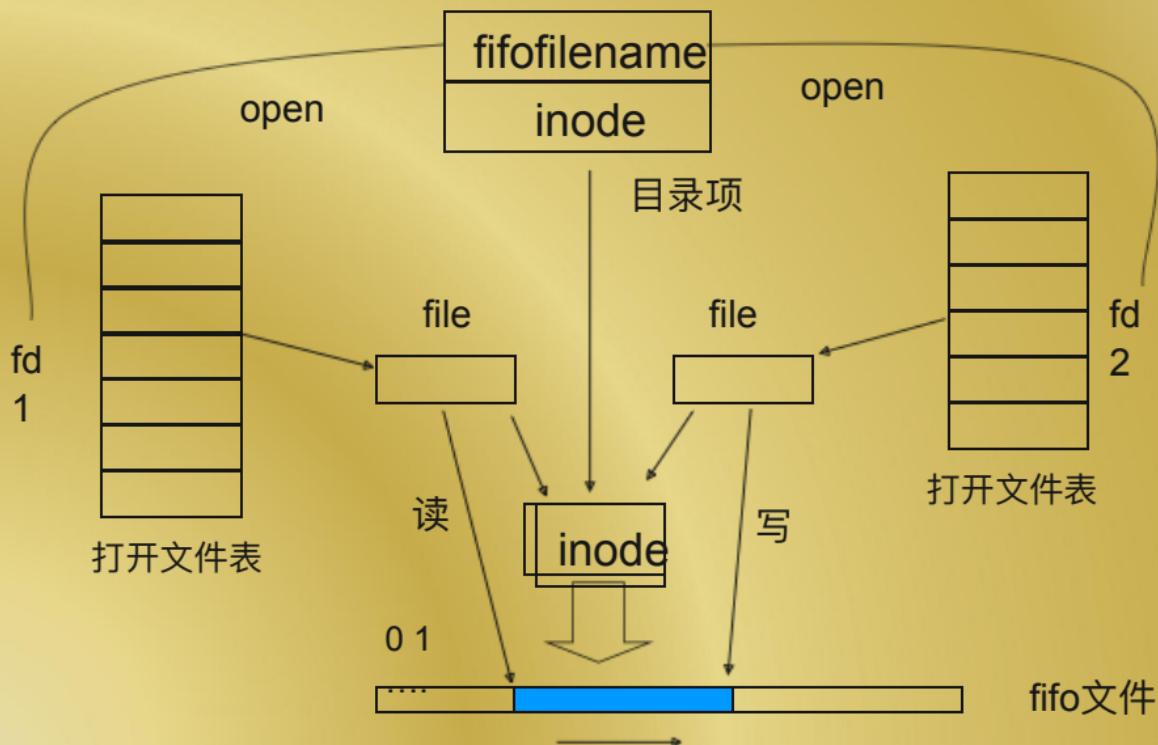
```
if(fork( ) == 0)
{
    pipe(fds[2]);           /* 子进程 */
    if (fork ( ) == 0){      /* 孙进程 */
        close(1);
        dup(fds[1]);         /* 复制fds[1] -> 1 */
        close(fds[1]);
        close(fds[0]);
        execvp("ls", "ls", 0); /* ls 执行write(1,...) */
    }
    close(0);
    dup(fds[0]);           /* 复制fds[0] -> 0 */
    close(fds[0]);
    close(fds[1]);
    execvp("wc", "wc", 0); /* wc执行read(0, ...) */
}
```

例：
ls|wc



有名管道

父子进程的fd分别由（文件名）得到，读写进程自动同步。



fifo特点

pipe:

- 无文件名
- 同一族系
- 临时存储文件（自动删除）

fifo:

- FIFO作为特殊文件而存在（有文件名）；
- 任何进程（不限于同一家族）均可共享；
- 可长期存在（永久存储,手工删除）。

创建fifo文件

shell:

mkfifo mode 文件名

例： \$ mkfifo a=rw myfifo

mknod 文件名 p

例： \$ mknod myfifo p

系统调用：

Int mknod (char *pathname, mode_t mode, dev_t dev);

例： mknod(“/tmp/myfifo”, S_IFIFO|0666, 0)

Int mkfifo(char *pathname, mode_t, mode);

例： mkfifo(‘/temp /fifo.1’, 0644);

例11. fifo

```
#include <fcntl.h>           /* fifo.c */
main( int argc, char *argv[ ] )
{
    int fd;
    char buf[256];
    mknod("myfifo", 0666,); /* 或mknod("myfifo", 010666, 0) */
    if(argc == 2)           /* a.out aaaaa */
        fd = open("myfifo", O_WRONLY);
    else                   /* a.out& */
        fd = open("myfifo", O_RDONLY);
    if(argc == 2)           /* a.out aaaaa */
        write(fd, argv[1], 6);
    else {                  /* a.out& */
        read(fd, buf, 6);
        printf("output = %s\n", buf);
    }
}
```

```
$ cc fifo.c /*编译， 目标代码: a.out */  
$ a.out& /* 后台进程 :a.out ,pid = 1357 */  
1357  
$ a.out aaaaa /* 前台进程 : a.out aaaaa */  
output = aaaaa /* a.out& 输出*/  
$
```



六。超级块与资源管理*

超级块

空闲节点管理

空闲块管理

超级块

每个分区只能含有一个文件系统。文件系统是指磁盘分区内的目录与其相应文件的自包含集合(有时也称文件卷)。如果是根文件系统则还含有一个自举块。

每个文件系统含有一个超级块，超级块含有本文件系统内的资源信息。其主要内容如下：

- 文件系统的规模；

- 文件系统中空闲块的数目；

- 在文件系统上可用的空闲块表(free[])；

- 空闲块表中下一个空闲块的（数组）下标(nfree)；

- 空闲块表的锁字段

- i 节点表的大小；

- 文件系统中空闲i 节点数目；

- 文件系统中空闲i 节点表(inode[])；

- 空闲i 节点表中下一个空闲i 节点的（数组）下标(ninode)；

- 空闲i 节点的锁字段；

- 超级块被修改标志；

- 文件系统只读标志

空闲i节点管理

概述

超级块内包含有关于本文件系统中空闲i 节点的资源信息，主要是一个空闲i 节点表。这是一个数组，其中每个表项纪录一个空闲i 节点的编号。

这些i节点号在数组中从下标0 开始顺序存放，不能有空白。所以最后一个有效项的下标加1也就是表内含有的空闲i 节点号的数目。这个数目保存在超级块中(ninode)。

申请/释放i节点按照后进先出的方式进行：申请i节点取inode[ninode-1]，然后ninode-1；释放i节点则将该节点号放入inode[ninode]，然后ninode+1；

当表满或表空时要做一些特殊处理：当表满时如再释放i节点，则无需填表而返回；如表空时再申请i节点，则需要从磁盘i节点表中找出空闲的i节点，将其编号由小到大顺序填入i节点表中，如此直至填满。其中最大（后）节点号称为**铭记 (remembered) i 节点号**。

在寻找磁盘中空闲i节点时，为了防止搜索范围太大，系统每次扫描只需从铭记i 节点号向后（大）寻找空闲i节点即可。因为小于此号的i节点已经被申请了。当释放一个i节点时，如果该节点号小于铭记i节点号，就应将此号取代这个的铭记i节点号。

由于文件系统安装时含有空闲i 节点（号）表的超级块已读入内存，所以申请/释放i节点的操作是很快的。仅当表空时才访问磁盘。

申请i节点

超级块空闲i节点表

空闲i节点号

空表项

申请i节点(号)过程

ninode=2 0	47		8	4															
	0	0	3	18	8	19	50												

- a. 现存20个i节点号
数组下标: 0---19

ninode=1 9	47		8																
	0	0	3	18	19	50													

- b. 申请 (取走)
48号空闲i节点

ninode=0 0	47																		
	0	8	50																

- c. 空闲i节点
号用完

ninode=5 0	53				47	47	47												
	5	0	6	48	49	50	50												

- d. 读入470 以后的
(471) 空闲i节点



释放i节点

磁盘i节点表



铭记i节点号

空闲i节点号	53			47	47	47
5				6	48	49

5

50

释放i节点(号)过程

- 最初空闲
i节点表 (已满)
铭记i节点号为535

铭记i节点号

空闲i节点号	49			47	47	47
9				6	48	49

9

50

铭记i节点号

空闲i节点号	49			47	47	47
9				6	48	49

9

50

- 释放499之后

$499 < 535$

499取代535成为

铭记i节点号

- 释放499之后

$601 > 499$

铭记i节点号

不变，仍为499

空闲块管理

概述

超级块内包含有关于本文件系统中空闲块的资源信息，主要是一个空闲块表。这是一个数组 (free[])，每个表项记录一个空闲块的编号。其中free[0]指向的空闲块用作索引块，其中的索引表free[]含有下一组空闲块号，而这个free[0]又指向下一个索引块，以此类推。

这些块号在数组中从下标0 开始向下顺序存放，不能有空白。所以最后一个有效项的下标加1也就是表内含有的空闲块号的数目。这个数目也存在超级块中(nfree)。

申请/释放块号按照后进先出的方式进行：申请块取free[nfree-1]，然后nfree-1;释放块则将该块号放入free[nfree],然后nfree+1;

申请/释放空闲块，当表满或表空时要做一些特殊处理：当表满时释放空闲块，则此释放块用作索引块。将空闲块表读入其中，然后将释放块的块号填入空闲块表第0项 (free[0]) ,nfree=1.

当表中只有1项时申请空闲块，则应将此项 (free[0]) 中块号取出返回给调用者，但此前这一块作为索引块应将其中的索引表内容写入超级块内的空闲块表 (free[]) 中，nfree = 50(假定)。

由于文件系统安装时含有空闲块（号）表的超级块已读入内存，所以申请/释放空闲块的操作是很快的。仅当表满或空时才访问磁盘。

应用实例

超级块中的空闲块表 (nfree[100]假定100项) (文件系统安装时随超级块读入内存)

nfree=

1

10

9

109块用作索引表的空闲块 (100)

109

211	208	205
-----	-----	-----

11

块

202

2

211

344	341	338
-----	-----	-----

24

块

335

3

a) 初始状

态

超级块中的空闲块

nfree表 free 109

2

949

949

109块用作索引表的空闲

109

211	208	205
-----	-----	-----

11

块

202

2

b) 释放949块之

超级块中的空闲块

nfree= 表

109	1
-----	---

1 9

109块用作索引表的空闲

211	208	205		11
202				2

211	211	338		24
211	335			3

211	344	341	338		24
211	335				3

211	335				3
211	335				3

c) 分配949块之后

超级块中的空闲块

0	109	211	208	205		11
0	202					2

211块用作索引表的空闲

211	344	341	338		24
211	335				3

211	335				3
211	335				3

d) 分配109块之后

109块内的索引表写入超级块空闲块

nfree[]

超级块中的空闲块表

nfree=	14	
1	8	
148	148块用作索引表的空闲	
块	211 208 205	
202		
211	211块用作索引表的空闲	
块	344 341 338	24
335		3

e) 释放148块之后

超级块空闲块表写入148块

内

148记入空闲块表free[0]中

七。文件系统管理

- 概述
- 文件系统的建立
- 文件系统的安装拆卸

概 述

文件系统位于一个磁盘分区或一个磁盘内。通过mkfs命令或系统调用可在指定分区或磁盘上建立一个文件系统。物理上，一个文件系统就是由超级块，i节点表，目录和文件数据块的集合；逻辑上，文件系统在用户面前表现为树形分级结构(除连接外)，最高级为一个根目录。

分区或磁盘作为逻辑盘在系统中作为普通设备看待，它们在/dev目录下也有文件名，如/dev/dsk1,/dev/fd0等，也有对应的i节点，但其中没有文件索引表，只有设备号。设备号包括两部分：主设备号和次设备号，它们分别标识设备的类别和具体的驱动器号。故设备号唯一地标识一个文件系统所在的具体设备。

系统中有一个磁盘分区含有根文件系统，所以叫做根设备。系统初启时计算机首先在这里找到操作系统核心，将其读入内存运行。

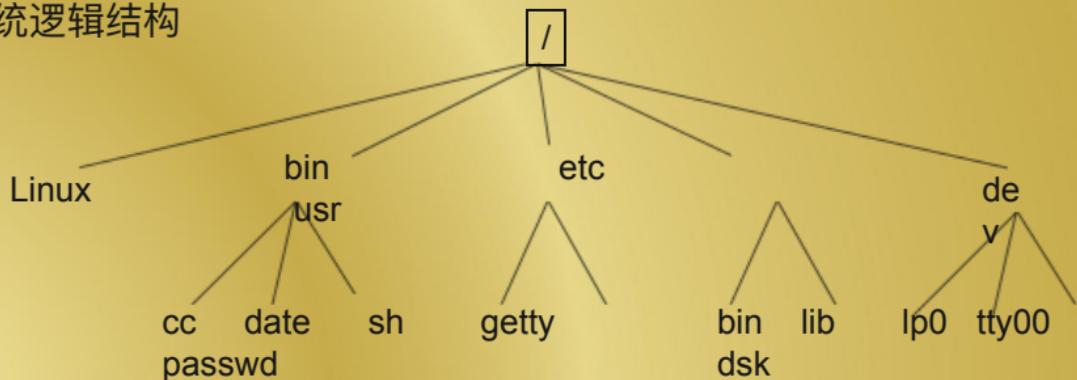
其它分区上的文件系统可以“安装”到根文件系统中的一个目录上（也叫安装点）。其效果是该目录逻辑上成为被安装文件系统的根目录。于是，这两个文件系统便合成为统一的树形分级结构。其后用户还可以将该文件系统“拆卸”下来。由于系统安全性的需要，安装拆卸操作必须由超级用户执行。

文件系统的建立

文件系统物理结构



文件系统逻辑结构



命 令

超级用户

`mkfs [-t fstype] filesystem [blocks] (8)`

在指定分区或磁盘上建立指定类型，指定块数的文件系统。对Linux类的文件系统要建立超级块和i节点表等。

`fstype`: 文件系统类型，如 ext2(Linux 缺省), minix, msdos, sysv, nfs, vfat.....

`filesystem`: 设备名, /dev/hda1, /dev/sdb2,....

`blocks`: 块数 (1K字节/块)

例：建立软盘文件系统，1.4M字节。

```
# mkfs /dev/fd0 1440
```

安装文件系统：

```
# mount /dev/fd0 /mnt/floppy
```

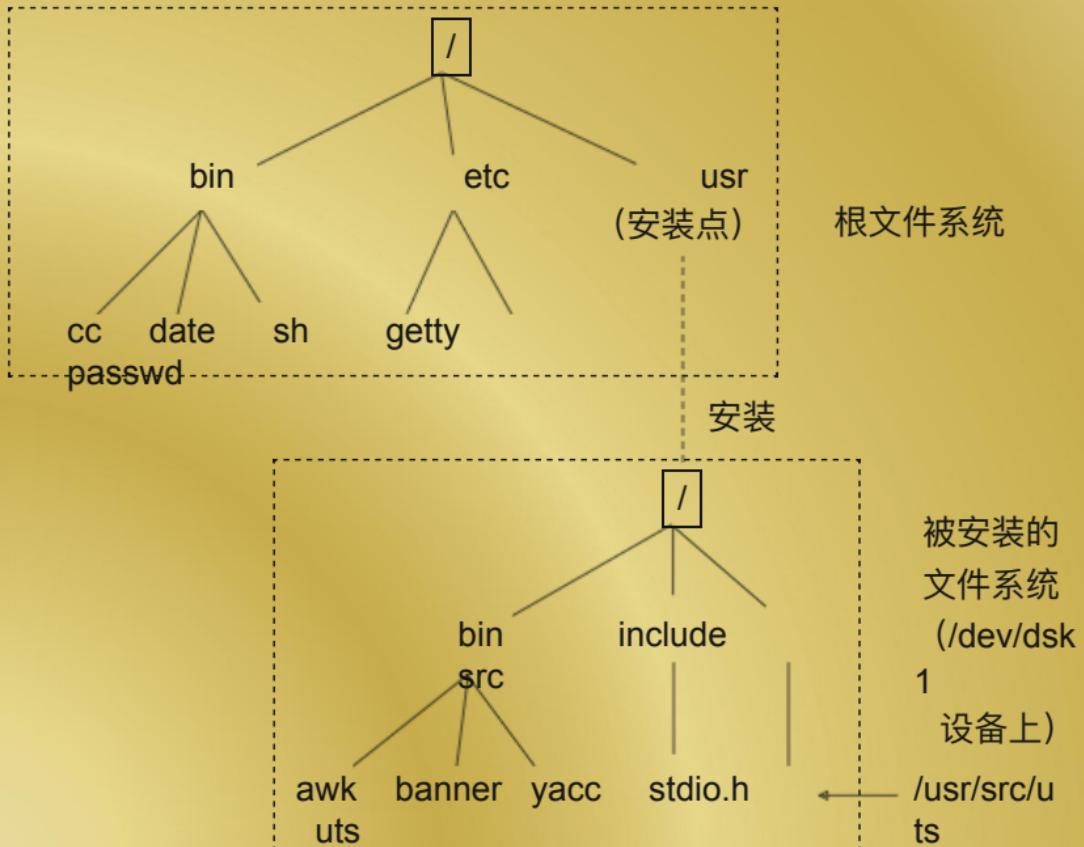
访问软盘上的文件

```
$ ls > /mnt/floppy/list
```

```
$ cat /mnt/floppy/list
```

.....

文件系统的安装拆卸



系统调用

```
int mount(const char *special, const char *dir, const char *fstype, int  
rwflag, const void *data );
```

安装文件系统。

*special : 被安装文件系统所在的设备的文件名, 如“/dev/dsk1”;

dir : 根文件系统上的安装点目录;

*fstype: 文件系统类型名, 如minix, sysv;

rwflag : 只读标志。0: 可写; 1: 只读;

*data: 由文件系统解释;

例:

```
mount( “/dev/dsk1”, ”/usr”, 0);
```

```
int umount ( const char *special );
```

```
int umount ( const char *dir );
```

拆卸special(dir)上的文件系统。

例:

```
umount( “/dev/dsk1”);
```

```
umount( “/usr”);
```

命 令

只超级用户使用：

安装文件系统：

mount [-t 文件系统类型] [-r] 设备名 安装目录名 (8)

-r : 被安装文件系统只读标志

例：

mount -r /dev/dsk1 /usr 安装硬盘文件系统

mount /dev/fd0 /mnt/floppy 安装软盘文件系统

mount -t vfat /dev/hda4 /mnt/win

 安装硬盘分区上windows的vfat文件系统

访问软盘上的文件

\$ ls > /mnt/floppy/list

\$ cat /mnt/floppy/list

.....

拆卸文件系统：

umount 设备名|安装目录名 (8)

例：

umount /dev/dsk1

umount /var

实现原理*



目录搜索过程：

1. 在内存节点表中查到的i 节点发现是安装点 (IMOUNT=1);
2. 查安装表找到有关表项，再找到被安装文件系统的根i节点；
3. 从这个根i节点开始搜索下一级目录/文件。

mount实现*

mount 实现算法：

参数：设备文件名，安装点目录名，只读标志。

- 1。检查超级用户权限；
- 2。取设备文件的索引节点，读入内存节点表项,得到设备号；
- 3。作合法性检查；
- 4。取安装点目录的i节点，读入内存节点表项；
- 5。设备的超级块读入内存缓冲区；
- 6。内存超级块内填初值（只读标志，空闲节点数置0等）
- 7。取设备的根i节点，读入内存节点表项；
- 8。找安装表一空项，填写如下内容：
 - 。设备号；
 - 。内存超级块地址；
 - 。设备的根i节点内存地址；
 - 。安装点目录的i节点内存地址；
- 9。标记安装点目录的内存i节点为已安装 (IMOUNT=1)

umount实现*

umount 实现算法：

参数：设备文件名

- 1。检查超级用户权限；
- 2。取设备文件的索引节点，读入内存节点表项,得到设备号；
- 3。由设备号找到安装表项；
- 4。清除系统内属于该文件系统的共享正文段表项；
- 5。将属于该文件系统的内存超级块，内存i节点和延迟写磁盘块写回磁盘；
- 6。释放该文件系统内存根i节点；
- 7。关闭设备，释放它的所有内存缓冲区；
- 8。清除目标目录i节点内的安装标志（IMOUNT=0）,释放该节点；
- 9。释放内存超级块；
- 10。释放安装表项。

八。虚拟文件系统*

- 概述
- file与dentry对象
- 虚拟i节点
- VFS与LFS

概 述

vnode/vfs 最早出现于sun公司的sunos中（1986），后来被AT&T公司集成到SVR4中。它们采用了面向对象的设计思想。

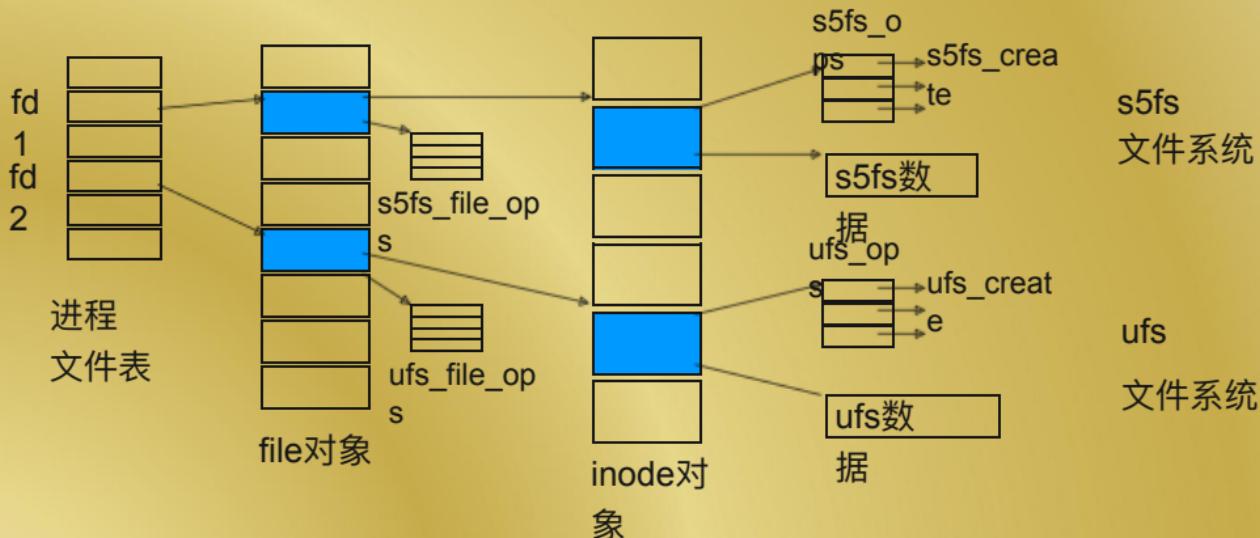
vnode (virtual node, 虚拟节点) 作为一种虚拟节点有一组统一的接口函数调用格式，但对于不同类的i 节点有着不同的实现函数体和私有数据。vnode 相当于面向对象中的抽象基类，它提供抽象接口，用于派生出各种文件节点的实现函数。这些文件节点既包括不同文件系统的inode 又包括socket等。

vfs (virtual file system ,虚拟文件系统) 作为一种虚拟的文件系统有一组统一的接口函数调用格式，但对于不同类的文件系统有着不同的实现函数和私有数据。vfs 相当于面向对象中的抽象基类，它提供抽象接口，用于派生出多种文件系统实现，如s5fs,ufs,NFS,FAT32等。

vnode/vfs 体系结构的目标：

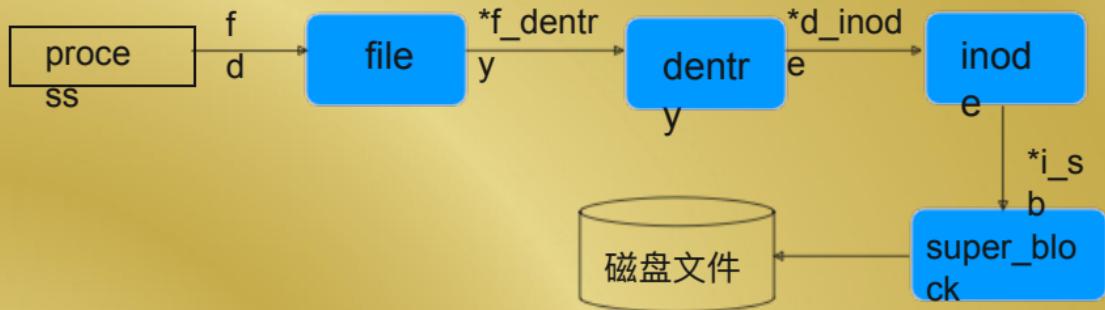
- 同时支持多种文件系统，如Linux类：s5fs, ufs 。非Linux类：FAT32；
- 用户对所有（在不同分区）安装的文件系统具有统一的视图（接口）；
- 支持网络文件系统(NFS)的共享和透明的访问；
- 允许厂家将定制的文件系统以模块方式加入内核。

面向对象的实现



Linux VFS对象关系

Linux虚拟文件系统中的对象及其关系：



file 对象与dentry对象

```
struct file {      /* file对象*/
    struct dentry *f_dentry;    /* dentry (目录项) 对象指针
        其中dentry.d_inode 为inode对象指针*/
    struct file_operations *f_op; /* 操作函数指针, 打开此文件时其初值
        由inode.i_fop指定 */
    loff_t f_pos;      /* 文件指针 */
    unsigned int f_count; /* 访问计数 */
    unsigned int f_flags; /* 打开方式 */
    .....
};
```

```
struct dentry {      /* dentry (目录项) 对象*/
    struct inode *d_inode; /* inode对象指针 */
    struct dentry *d_parent; /* 父目录 */
    struct list_head d_vfsmnt; /* 安装链表结构 */
    struct dentry_operations *d_op; /* 操作函数表 */
    struct super_block *sb; /* 超级块对象 */
    unsigned char d_iname[16]; /* 短成员名 */
    .....
```

file operations

lseek(file,offset,whence) /* 修改文件指针 */
read(file,buf,count,offset) /* 读文件offset后加count */
write(file,buf,count,offset) /* 写文件offset后加count */
readdir(dir,dirent,filldir) /* 读目录dir下一项到dirent,filldir 为读目录
项函数地址 */
poll(file,poll_table) /* 检查等待某文件操作事件 */
ioctl(inode,file,cmd,arg) /* 设备文件操作 */
mmap(file,vma) /* 内存映射 */
open(inode,file) /* 打开文件，链接 inode对象 */
flush(file) /* 关闭文件时减少 f-count计数 */
release(inode,file) /* 释放file对象 (f_count = 0) */
fsync(file,on) /* 文件在缓冲数据写回磁盘 */
check_media_change(dev) /* 检查可移动设备介质是否改变 */
revalidate(dev) /* 恢复设备一致性 (NFS用) */

虚拟i节点

抽象
基类

```
struct  
inode  
{  
    i_ino;  
    i_count;  
    i_dev;  
    i_mode;  
    *i_sb;  
    *i_op;  
    ....  
};  
  
union {  
    minix_i;  
    ext2_i;  
    sysv_i;  
    ntfs_i;  
    nfs_i;  
    socket_i;  
    ....  
};  
};
```

各种文件系统i
节点的通用信息

super
block

包含

```
struct  
inode_operations  
{  
    create();  
    lookup();  
    link();  
    unlink();  
    symlink();  
    ....  
};
```

linux i节点向量表

```
struct  
ext2_inode_info  
{  
    ....  
    _u32  
    i_data[15];  
    ....  
};
```

linux i节点私有数据

linux
文件系统
i节点
子类对象
(内存布
局)

实现思想

虚拟i节点的实现

类 struct inode 可看成是抽象基类，内存i 节点可看成是它的派生类对象的实现，每个派生类对应一个特定的i节点；

内存i 节点是在打开文件时调用文件系统超级块中的read_inode()函数读入内存的；

派生类的私有数据由联合(union)变量中的一员表示（如ext2文件系统对应成员类型为 struct ext2_inode_info）；

派生类的函数指针表由inode_operations[]表示，其中含有具体文件i 节点的函数入口地址；

inode对象

```
struct inode {  
    unsigned long      i_ino; /* i节点号 */  
    atomic_t          i_count; /* 访问计数 */  
    kdev_t            i_dev; /* 设备号 */  
    umode_t           i_mode; /* 属性 */  
    nlink_t           i_nlink; /* 链接数 */  
    uid_t             i_uid; /* 文件主用户号 */  
    gid_t             i_gid; /* 文件主组号 */  
    kdev_t            i_rdev; /* 实设备号 */  
    loff_t            i_size; /* 字节数 */  
    time_t            i_atime; /* 上次访问文件时间 */  
    time_t            i_mtime; /* 上次写文件时间 */  
    time_t            i_ctime; /* 上次修改节点时间 */  
    struct inode_operations *i_op; /* i节点操作函数表 */  
    struct file_operations *i_fop; /* file缺省操作函数表 */  
    struct super_block *i_sb; /* 超级块对象指针 */  
    struct file_lock   *i_flock; /* 文件锁链表指针 */
```

```
union {
    struct minix_inode_info  minix_i; /* minix文件系统i节点 */
    struct ext2_inode_info   ext2_i;   /* ext2文件系统i节点 */
    struct hpfs_inode_info  hpfs_i;   /* HP文件系统i节点 */
    struct ntfs_inode_info  ntfs_i;   /* NT文件系统i节点 */
    struct msdos_inode_info msdos_i;  /* MSDOS文件系统i节点 */
    struct nfs_inode_info   nfs_i;    /* NFS文件系统i节点 */
    struct sysv_inode_info  sysv_i;   /* SYSV文件系统i节点 */
    struct ufs_inode_info   ufs_i;    /* ufs文件系统i节点 */
    struct proc_inode_info  proc_i;   /* proc文件系统i节点 */
    struct socket           socket_i; /* 套接口 */
    .....
} u;
};
```

inode_operations

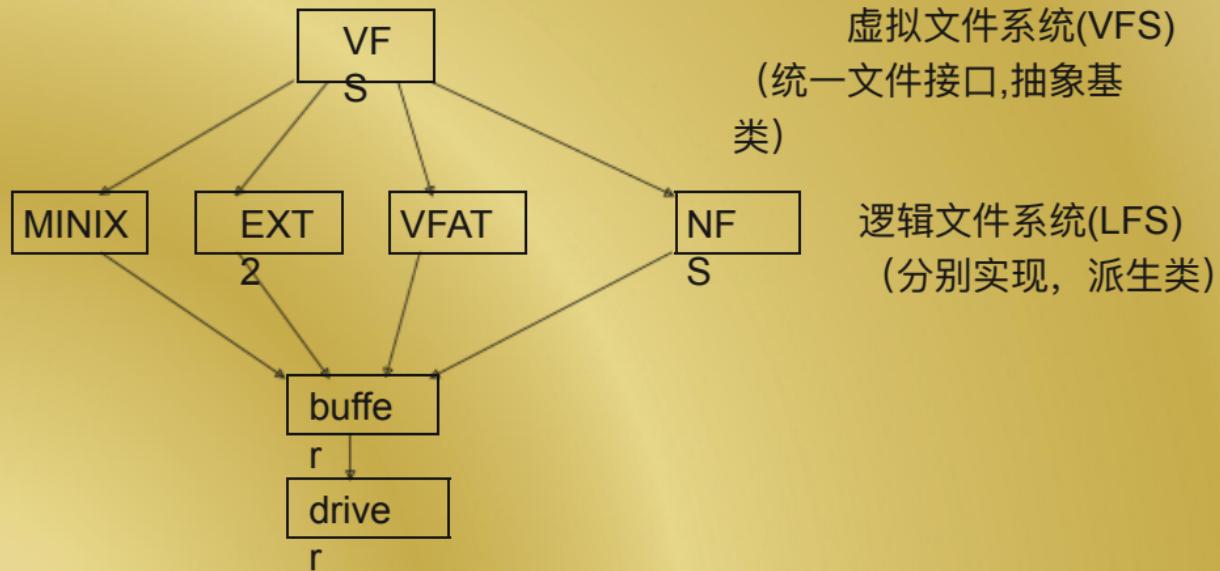
struct inode_operations 成员：

```
create(dir,dentry,mode)      /* 在dir下建立dentry指定的磁盘i节点 */
lookup(dir,dentry)          /* 查目录dir 下dentry中指定的文件 */
link(old_dentry,dir,new_dentry) /* 建立硬链接 */
unlink(dir,dentry)          /* 删除硬链接 */
symlink(dir,dentry,symname) /* 建立符号链接 */
mkdir(dir,dentry,mode)       /* 建立目录 */
rmdir(dir,dentry)           /* 删除目录 */
mknod(dir,dentry,mode,rdev) /* 建立特殊文件 */
rename(old_dir,old_dentry,new_dir,new_dentry) /* 重命名 */
redlink(dentry,buffer,buflen) /* 读符号链 */
follow_link(inode,dir)      /* 解释符号链 */
readpage(file,pg)           /* 读数据页 */
writepage(file,pg)          /* 写数据页 */
bmap(inode,block)           /* 逻辑块号转换为物理块号 */
truncate(inode)              /* 修改文件长度 */
permission(inode,mask)       /* 检查访问许可 */
smap(inode,sector)          /* 逻辑块号转换为扇区号(FAT) */
updatepage(inode,pg,buf,offset,count,sync) /* 更新inode的数据页 */
revalidate(dentry)           /* 更新dentry指定文件的缓存属性 */
```

```
struct ext2_file_inode ext2_dir_inode ext2_symlink_inode  
inode_operations: _operations: _operations _operations
```

create	null()	ext2_create()	null()
lookup	null()	ext2_lookup()	null()
link	null()	ext2_link()	null()
unlink	null()	ext2_unlink()	null()
symlink	null()	ext2_symlink()	null()
mkdir	null()	ext2_mkdir()	null()
rmdir	null()	ext2_rmdir()	null()
mknod	null()	ext2_mknod()	null()
rename	null()	ext2_remlink()	null()
redlink	null()	null()	ext2_readlink()
follow_link	null()	null()	ext2_follow_link
readpage	generic_readpage()	null()	null()
writepage	null()	null()	null()
bmap	ext2_bmap()	null()	null()
truncate	ext2_truncate()	null()	null()
permission	ext2_permission()	ext2_permission()	null()
smap	null()	null()	null()
updatepage	null()	null()	null()

VFS与LFS



文件系统类型

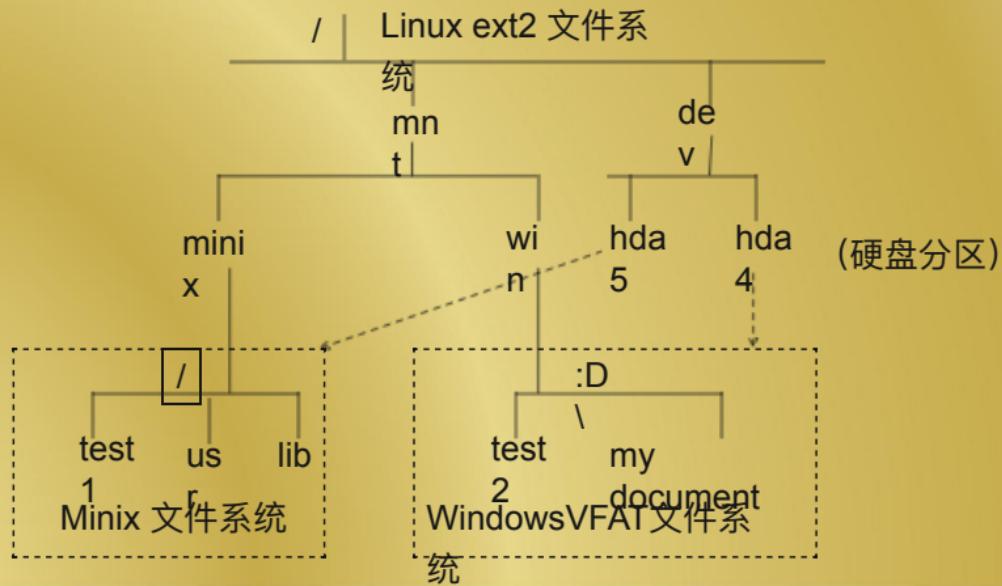
mount [-t 文件系统类型] [-r] 设备名 安装目录名 (8)

-t :文件系统类型

-r :被安装文件系统只读标志

例: # mount -t vfat /dev/hda4 /mnt/win

安装硬盘分区/dev/hda4上windows的 vfat 类新型文件系统。

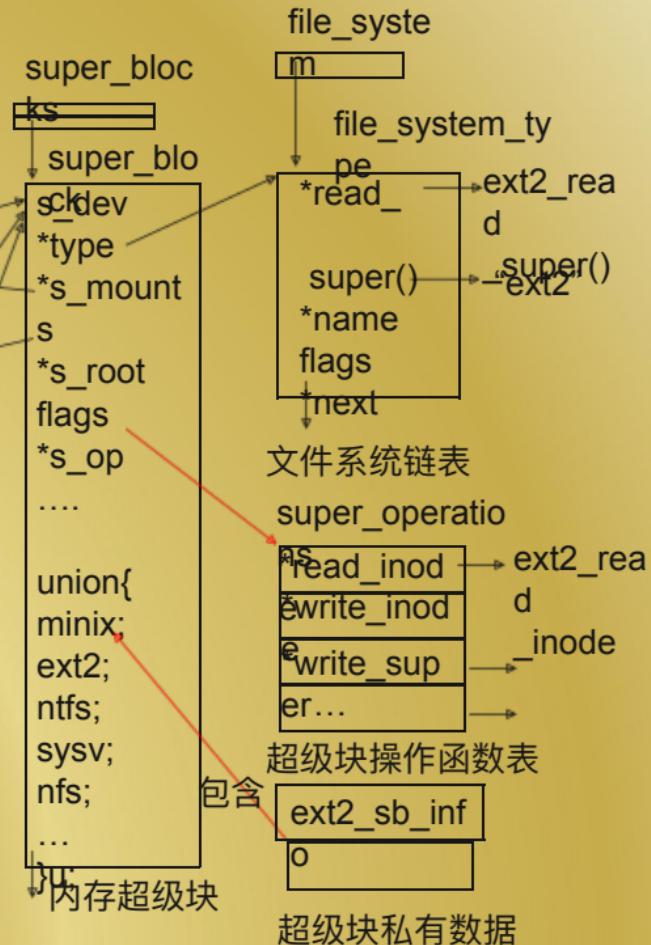
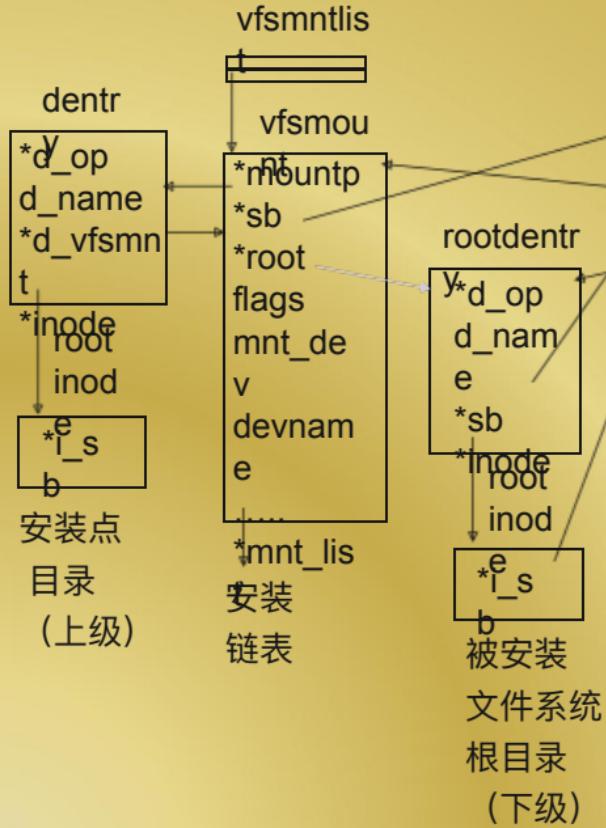


举 例

将minix文件拷贝到windows文件.

```
inf = open(“/mnt/minix/test1”, O_RDONLY, 0);
outf = open(“/mnt/win/test2”, O_WRONLY|O_CREATE|O_TRUNC, 0666);
    /* 相当于 creat(“/tmp/dir2/test2”, 0666); */
do {
    c = read( inf, buf, 4096);
    write(outf, buf, c);
} while (c);
close(outf);
close(inf);
```

VFS数据结构



实现思想

文件系统结构实现

类 struct file_system_type 可看成是抽象基类，代表抽象的文件系统。文件系统链表项是它的实例。文件系统初启时调用注册函数(register_filesystem(&ext2_fs_type,...))生成文件系统实例；派生类的 *read_super() 是读具体文件系统超级块的函数指针；*name 指向具体文件系统类型名。

超级块实现

类 struct super_block 可看成是抽象基类，代表抽象的超级块。内存超级块可看成是它的派生类对象的实现，每个派生类对应一个特定的文件系统；

超级块是在安装文件系统时调用文件系统链表项(file_system_type)中的 read_super() 函数读入内存的；

派生类的私有数据由联合(union)变量中的一员表示（如 ext2 文件系统对应成员类型为 struct ext2_sb_info）；

派生类的函数指针表由 super_operations[] 表示，其中含有具体文件系统超级块的函数入口地址。

file_system_type对象

```
struct file_system_type  
{  
    const char    *name;          /* 文件系统名 */  
    struct super_block *(*read_super)(struct super_block *, void *,int);  
                                /* 读超级块方法 */  
    int           fs_flags;      /* 安装标志 */  
    struct file_system_type * next;   /* 指向链表下个元素指针 */  
}
```

ext2_fs_type实例：

name: “ext2”
read_super: exte_read_super()
fs_flags: FSQUIRES_DEV

super_block对象

```
struct super_block {      /* 超级块 */
    struct list_head s_list;    /* 超级块链表指针 */
    kdev_t s_dev;            /* 设备号 */
    unsigned long s_blocksize; /* 块大小 (字节数) */
    unsigned char s_lock;     /* 锁 */
    struct file_system_type *s_type; /* 文件系统类型 */
    struct super_operations *s_op; /* 函数 (方法) 表 */
    unsigned long s_flags;    /* 安装标志 */
    struct dentry *s_root;    /* 所属根目录 */
    struct list_head s_mounts; /* 对应的vfsmount */

    .....

    union {                  /* 具体 (派生) 文件系统信息 */
        struct minix_sb_info minix_sb; /* minix 文件系统 */
        struct ext2_sb_info ext2_sb; /* ext2文件系统 */
        struct ntfs_sb_info ntfs_sb; /* windows NT 文件系统 */

        .....
    } u;
}
}
```

super_operations

```
struct super_operations 成员： struct super_operations ext2_sops = {}  
read_inode(inode)           ext2_read_inode  
write_inode(inode)          ext2_write_inode  
put_inode(inode)            ext2_put_inode  
delete_inode(inode)         ext2_delete_inode  
notify_change(dentry,iattr)  null  
put_super(super)            ext2_put_super  
write_super(super)          ext2_write_super  
statfs(super,bufsize)       ext2_statfs  
remount_fs(super,flags,date) ext2_remount_fs  
clear_inode(inode)          null  
umount_begin(super)         null
```

vfsmountct数据结构

```
struct vfsmountct
{
    struct dentry *mnt_mounpoint; /* 安装点, 上级目录 */
    struct super_block *mnt_sb;   /* 内存超级块指针 */
    struct dentry *mnt_root;      /* 下级根目录 */
    int mnt_flags;                /* 标志位 */
    char *mnt_devname;           /* 设备名, 例 /dev/dsk/hda1 */
    struct list_head mnt_instances; /* 属于同一fs 的vfsmount */
    struct list_head mnt_list;     /* 所有vfsmount 链表*/
    uid_t mnt_owner;              /* 安装用户id */
    .....
}
```

安装文件系统

mount 实现算法：

参数：设备文件名，安装点目录名，文件系统类型，只读标志。

1。根据安装点目录名找到它的内存i节点；

2。建立super_block对象；

a. 申请一内存缓冲区 (*sb) ；

b. 填写与文件系统类型无关的初始化部分；

c. 根据设备文件名找到它的设备号；

d. 根据文件系统类型找到它的read_super()方法

(例如：ext2_read_super()) ；

由磁盘读入超级块部分内容到内存缓冲区 (*sb) ；

e. 设备根节点读入内存；

f. 填写内存超级块：

 sb -> s_root =设备根节点内存地址；

 sb -> s_op = &ext2_sops;

3。申请vfsmount并填写初值：

 安装点目录i节点，下级根节点，内存超级块，标志，用户id及链表连接指针。

目录搜索过程

- 搜索到安装点目录（dentry对象，`d_vfsmnt!=0`），查找`d_vfsmnt`指向的vfsmount结构，即安装链表中的一项；
- 由root 找到下级被安装文件系统的根目录项；
- 进而找到对应的根i节点；由此向下搜索到所要求的文件的i节点。

打开文件

open实现算法：

输入参数：文件名，打开标志；输出参数：文件描述字

1. 由文件名检索目录分级结构：

- 。由根目录或当前目录的i节点开始搜索；
- 。如果遇到安装点则取下级文件系统的根节点；
- 。逐级调用各级目录的操作函数(*i_op -> lookup)() (如， ext2_lookup)，找到指定文件的节点号。

2. 建立inode对象。

- 。申请一inode对象(内存i节点)，填入一般初值；
- 。调用超级块对象的操作函数(*s_op ->read_inode)() (如， ext2_read_inode())
由磁盘读入有关具体文件系统(如ext2)待查文件的 i节点，填入内存i节点中；
- 。返回dentry对象(内含inode地址)。

3. 建立file对象。

申请file对象内存，填入如下信息：

f_dentry = dentry(其中d_inode = inode地址); f_flags = 打开标志;
f_pos = 0 (文件指针) ; f_op = i_fop (操作函数表) ;

4. 申请fd数据结构，填入file地址等信息；

5. 返回文件描述字 (fd数据结构索引) 。

Linux 编程及应用

主讲：任继平

邮

箱：rjp@mail.hzau.edu.cn

Linux

----- What?

----- Why?

----- How?

What?

理解Linux的五要素

- **UNIX 操作系统**
- **MINIX 操作系统**
- **GNU 计划**
- **POSIX 标准**
- **Internet 网络**

Why?

稳定
安全
可靠
开源

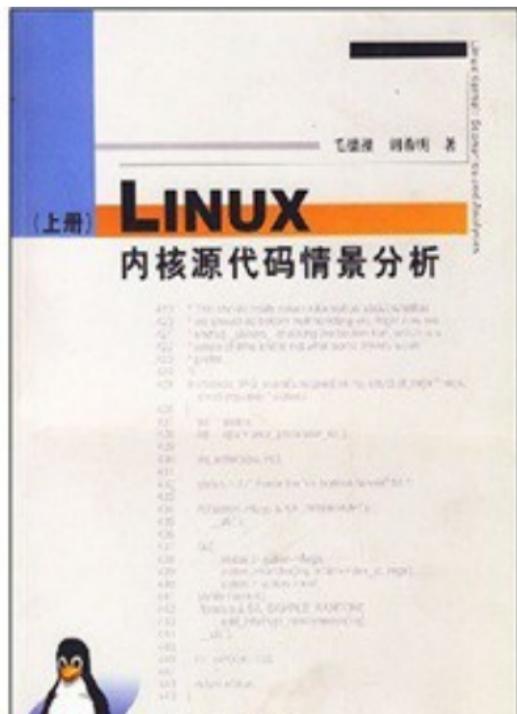
.....

How?

深入源代码 多实践

主要参考资料：

[1] 杨宗德.Linux高级程序设计（第三版）北京：人民邮电出版社2012



课程要求：

作业、实验和平时表现：30%

期末考试：70%



课程内容

网络编程

内存管理

文件系统

设备管理



进程管理

Shell编程

课程特点

Linux编程特点

涉及内核
原理

概念较
多，抽象

涉及面
广

Linux编程及应用主要知识点

- (1) 补充：Linux下Shell编程
 - (2) 磁盘文件（普通文件，链接文件，目录）管理
教材第4、5、6章
 - (3) 进程及进程间通信
教材本书第7、8、9章
 - (4) 线程及线程间同步
教材第10、11章
 - (5) 网络编程
教材第12、13、14、15章
 - (6) 其它：编程工具、编程环境
教材第1、2、3章
-

磁盘文件管理主要内容

- 普通文件IO操作
 - ANSI C文件IO, 文件描述符及相关操作
 - POSIX 文件IO, 文件流及相应操作
- 目录文件管理
 - 目录流及目录流操作
- 符号链接文件管理
 - 符号链接及操作
- 磁盘文件属性获取与磁盘文件属性修改
- 目标:
 - 能够实现cp命令, ls -l等基本命令的源代码编写

进程及进程间通信机制主要知识点

- 进程管理

- 创建

- 执行新代码

- 退出

- 等待

- 进程间通信

- 数据传递：

- 管道（有名，无名管道）

- IPC的消息队列

- IPC共享内存

- 同步

- 信号量

- 异步

- 信号

线程与线程同步基本知识点

- 线程基本操作

- 创建
 - 退出
 - 取消
 - 等待

- 线程同步机制

- 互斥锁
 - 读写锁
 - 条件变量
 - 线程信号灯

- 线程与信号

网络编程知识点

- 网络基础及支撑函数，工具
 - TCP/IP协议栈，数据封包拆包过程，TCP，UDP，IP包头
 - BSD TCP，UDP网络编程流程及API函数
 - 地址处理函数，大小端问题，socket属性控制
 - 域名解析
- TCP高级
 - 阻塞与非阻塞处理
 - 多路复用
 - 信号驱动
- UDP高级
 - 广播
 - 组播

编程工具及编程环境知识点

- 编辑器，编译调试工具使用
 - VIM
 - GCC
 - GDB
 - Makefile
 - 头文件，库文件的使用，库文件的创建
 - 错误，帮助信息的获取，编译规范要求
 - 段域加载，内存管理基础知识
-



补充 Shell 编程

本章学习目标

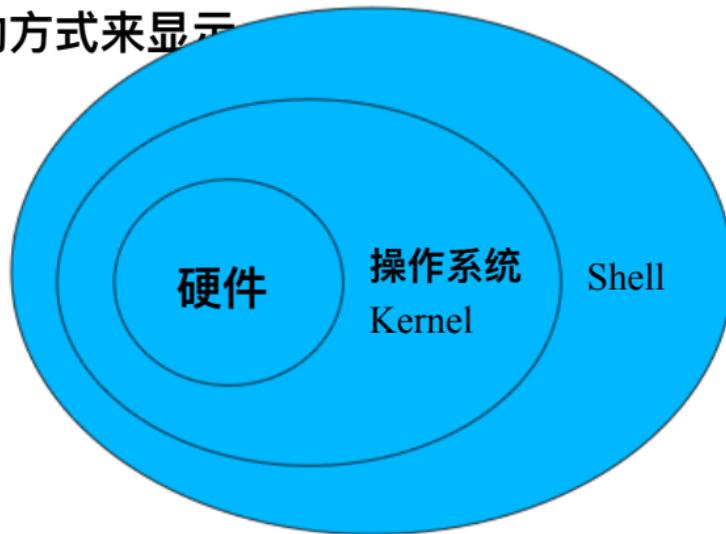
- 了解shell编程的基本概念及其特点
- 掌握shell各种**变量**的区别
- 掌握正则表达式的使用方法
- 熟悉shell的各种**流程控制**
- 了解shell的**函数**

目录

- 1.1 Shell概述
- 1.2 创建和执行shell程序
- 1.3 变量
- 1.4 位置参数 • 1.7 输入和输出
- 1.5 特殊字符 • 1.8 表达式的比较
- 1.6 运算符 • 1.9 流程控制语句
- 1.10 函数

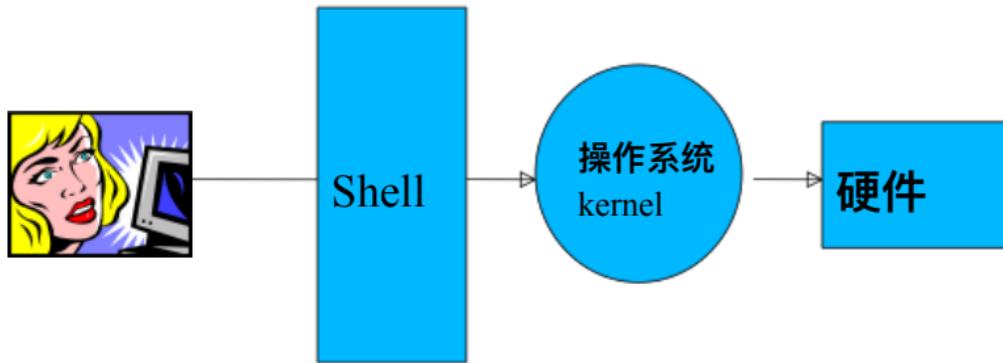
1.1 Shell概述

Shell就像一个壳层，这个壳层介于用户和操作系统之间，负责将用户的命令解释为操作系统可以接收的低级语言，并将操作系统响应的信息以用户可以了解的方式来显示。

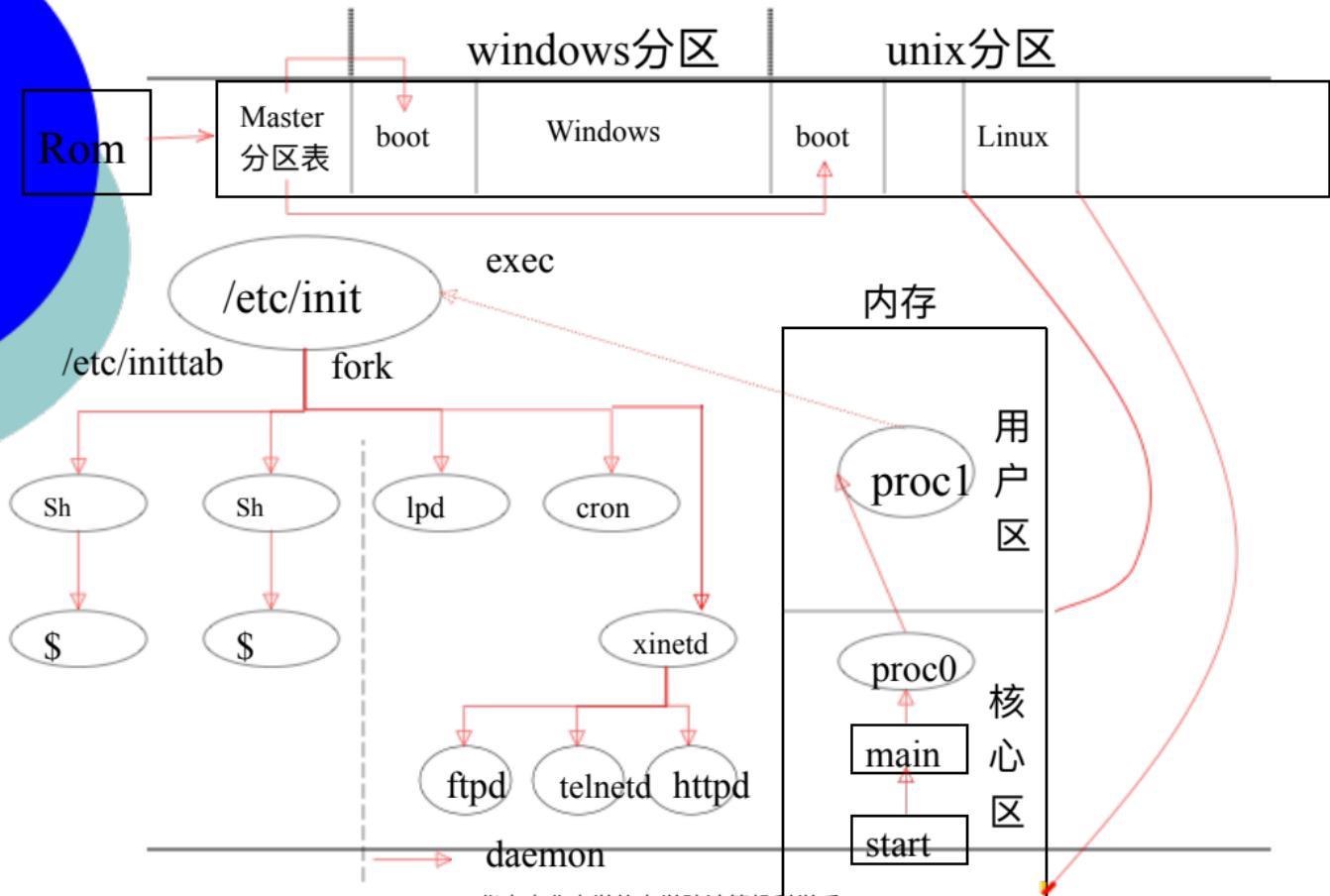


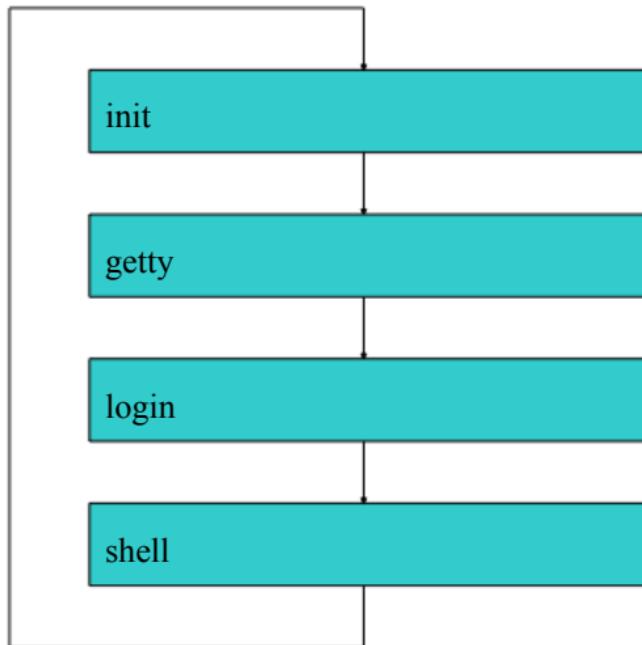
1.1 Shell概述

Shell角色图



系统初启和进程树





用户登录,退出循环过程

1.1 Shell概述

- 从用户登陆到注销期间，用户输入的每个命令都会经过解译及执行，其负责机制就是shell。
- shell是一个命令语言解释器，它拥有自己内建的shell命令集，shell也能被系统中其他应用程序所调用。用户在提示符下输入的命令都由shell先解释然后传给Linux核心。
- 有一些命令，如改变工作目录命令cd，是shell内置命令。还有一些命令，例如拷贝命令cp和移动命令mv，是存在于文件系统中某个目录下的单独的程序。对用户而言，不必关心一个命令是建立在shell内部还是一个单独的程序。

1.1 Shell概述

- shell的主要版本

在Linux系统中常见的shell版本有以下几种。

(1) Bourne shell(sh): 它是UNIX最初使用的shell，并且在每种UNIX都可以使用。它在shell编程方面相当优秀，但处理与用户的交互方面不如其他几种shell。

(2) C shell(csh): 它最初由Bill Joy编写，它更多地考虑了用户界面的友好性，支持如命令补齐等一些Bourne shell所不支持的特性，但其编程接口做得不如Bourne shell。C shell被很多C程序员使用，因为C shell的语法和C语言的很相似，C shell也由此得名。

(3) Korn shell(ksh): 它集合了C shell和Bourne shell的优点，并且和Bourne shell完全兼容。

1.1 Shell概述

- (4) Bourne Again shell(bash): bash是大多数Linux系统的默认shell。它是Bourne shell的扩展，并与Bourne shell完全向后兼容，而且在Bourne shell的基础上增加和增强了很多特性。。
- (5) tcsh: 它是C shell的一个扩展版本，与csh完全向后兼容，但它包含了更多使用户感觉方便的新特性，其最大的提高是在命令行编辑和历史浏览方面。它不仅和Bash shell提示符兼容，而且还提供比Bash shell更多的提示符参数。
- (6) pdksh: 它是一个专门为Linux系统编写的Korn shell(ksh)的扩展版本。Ksh是一个商用shell，不能免费提供，而pdksh是免费的。

1.1 Shell概述

表1 Linux中的各种shell

shell	名称描述	位置
ash	一个小shell (和sh类似)	/bin/ash
ash.static	一个不依靠软件库的ash版本	/bin/ash.static
bash	BourneAgainShell	/bin/bash
bash2	BourneAgainShell的新版本	/bin/bash2
bsh	ash的一个符号链接	/bin/bsh
csh	Cshell,tcsh的一个符号链接	/bin/csh
ksh	公共域受限制的shell(针对网络操作)	/usr/bin/ksh
sh	bash的一个符号链接	/bin/sh
tcsh	和csh兼容的shell	/bin/tcsh
zsh	一个和csh,ksh和sh兼容的shell	/bin/zsh

1.1 Shell概述

判断登入shell

使用echo命令来查询系统的“SHELL”环境变量，
命令如下：

> #echo \$SHELL

```
[root@localhost ~]# echo $SHELL  
/bin/bash  
[root@localhost ~]# █
```

1.1 Shell概述

- 暂时变更shell

除非是在受限的shell中,否则若要变更使用shell,只要执行该Shell程序名称(shell_name),即可切换到不同的Shell。如:

- # sh (或 # csh等等)

- 此处的shell_name是指shell的名称(例如, sh或csh)。暂时变更shell,可在其它的shell中进行试验。

- 键入exit或CTRL-D可以回到原始的shell中。

1.1 Shell概述

```
[root@localhost ~]# echo $SHELL  
/bin/bash  
[root@localhost ~]# sh  
sh-3.1# ps  
    PID  TTY          TIME CMD  
 2630  pts/0        00:00:00 bash  
 2658  pts/0        00:00:00 sh  
 2659  pts/0        00:00:00 ps  
sh-3.1# exit  
exit  
[root@localhost ~]# ps  
    PID  TTY          TIME CMD  
 2630  pts/0        00:00:00 bash  
 2660  pts/0        00:00:00 ps  
[root@localhost ~]# 
```

1.1 Shell概述

- Shell功能介绍

交互式处理 (Interactive Processing)

接收来自用户输入的命令后，shell 会根据命令类型的不同来执行，执行完毕后，shell 会将结果回传给用户，并等待用户下一次输入。用户执行exit 或是按 Ctrl+D 来注销 shell 才会结束。

1.1 Shell概述

- 命令补全功能

指用户输入命令后，有时**不需输入完整的命令**，而系统会自动找出最符合的命令名称，这种功能可以节省输入长串命令的时间。

不需要输入**完整的文件名**，只需输入开头几个字母，然后**按Tab键**时候，系统会补充完整，**连续按两次Tab (Esc) 键**系统会显示所有符合输入前缀的文件名称。

若忘了**命令的全名**，而只记得命令的开头字母，**按Tab键一次**会补充完整，**连续按两次Tab (Esc) 键**系统会显示所有符合输入前缀的命令名称。

1.1 Shell概述

- 查阅历史记录 – history命令
- 在Linux系统上输入命令并按下Enter后，这个命令就会存放在命令记录表（`~/.bash_history`）中，预定的记录为1000笔，这些都定义在环境变量中。
- 列出所有的历史记录：`#history`
- 只列出最近5笔记录犯例：`#history5`
- 使用命令记录号码执行命令：`#!561`
- 重复执行上一个命令：`#!!`
- 执行最后一次以ls开头的命令：`#!ls`

1.1 Shell概述

- 别名（Alias）功能
 - 查询目前系统所有别名：#alias
 - 设置别名：#alias dir='ls -l'
 - 使用别名：#dir /etc
 - 取消别名：#unalias dir
 - alias命令的效力仅限于该次登录，在注销系统后，这个别名的定义就会消失。如果希望每次登陆都使用这些别名，则应该将别名的设置加入“~/.bashrc”文件中，若是写入“/etc/bashrc”文件中，则系统上的所有用户都能使用这个别名。

1.2 创建和执行shell程序

- 不同的 shell 其编程(命令)语法有所不同
 - 较常见的 shell 脚本是 bash
 - 另一种较常见的 shell 脚本是 tcsh，其命令/语法类似 C 语言
- 学习脚本编程的原因
 - 在有些场合，希望一些常用的命令集能用一个命令实现；
 - 可以处理一些特定的问题，如计算每月上网的总时数。

1.2 创建和执行shell程序

- Shell 脚本编程前的准备

- 文本编辑器(vi 或 vim, gedit等)

- 脚本解释程序(bash, tcsh等)

- 其他工具 (用来扩充Shell 脚本的功能, 如: grep, wc)

1.2 创建和执行shell程序

例：显示当前的日期时间、执行路径、用户账号及所在的目录位置。

1. 建立 shell 脚本

如建立一个名为 ex1 的 shell 脚本，可提示符后输入命令：

➤ \$vi ex1.sh

1.2 创建和执行shell程序



在vi编辑器中输入下列内容：

```
#!/bin/bash
#This script is a test!
echo -n "Date and time is :"
date
echo -n "The executable path is :" $PATH
echo "Your name is :`whoami`"
echo -n "Your current directory is :"
pwd
#end
```

- 2. 用三种方法执行 shell 脚本

1.2 创建和执行shell程序

- >
 - >
 - >
 - >
 - >
 - >
 - >
 - >
 - >
- (1) 输入定向到shell脚本。
其一般形式是：\$bash < 脚本名
例如：\$bash < ex1
- (2) 以脚本名作为参数。其一般形式为：
\$bash 脚本名 [参数]
例如：\$bash ex1
如果以当前shell执行一个shell脚本，则可以使用如下简便形式：
\$. 脚本名 [参数]
- (3) 使用chmod命令将 shell 脚本的权限设置为可执行，然后在提示符下直接执行它。
例如：
\$chmod a+x ex1
\$./ex1

1.2 创建和执行shell程序

- 在编写shell时，第一行一定要指明系统需要那种shell解释你的shell程序，如:#!/bin/bash, #! /bin/csh, /bin/tcsh, 还是#!/bin/pdksh。
- 用上面执行 shell 脚本的三种方法分别体会这句话的作用。

1.3 变量

- 3种类型：
 - 环境变量：系统提供，不用定义，可以修改
 - 内部变量：系统提供，不用定义，不能修改
 - 用户变量：用户定义，可以修改
- 与其他语言的区别：非类型性质，也就是不必指定变量是数字或字符串等。

1.3 变量

环境变量

Linux环境（也称为shell环境）由许多变量及这些变量的值组成，由这些变量和变量的值决定环境外观。这些变量就是环境变量。

主要环境变量的有：

- (1) HOME：用户目录的全路径名。
- (2) UID 当前用户的识别字，取值是由数位构成的字串。
- (3) LOGNAME：即用户的注册名，由Linux自动设置。
- (4) MAIL：用户的系统信箱的路径。
- (5) PATH：shell从中查找命令的目录列表。

1.3 变量

- (6) PS1: shell的主提示符，在特权用户下，默认的主提示符是#，在普通用户下，默认的主提示符是\$。
- (7) PS2: 在Shell接收用户输入命令的过程中，如果用户在输入行的末尾输入“\”然后回车，或者当用户按回车键时Shell判断出用户输入的命令没有结束时，就显示这个辅助提示符，提示用户继续输入命令的其余部分，默认的辅助提示符是>。
- (8) PWD: 用户当前工作目录的绝对路径名，该变量的取值随cd命令的使用而变化。它指出用户目前在Linux文件系统中处在什么位置。它是由Linux自动设置的。
- (9) SHELL: 用户当前使用的shell。它也指出你的shell解释程序放在什么地方。
- (10) TERM: 用户终端类型。

1.3 变量

- \$HOME/.bash_profile (/etc/profile)
env

```
[fedora@localhost ~]$ env
HOSTNAME=localhost.localdomain
TERM=xterm
SHELL=/bin/bash
SSH_CLIENT=192.168.64.1 1057 22
SSH_TTY=/dev/pts/0
LC_ALL=zh_CN.GB18030
USER=fedora
PATH=/usr/kerberos/bin:/usr/local/bin:
      /bin:/usr/bin:/home/fedora/bin
PWD=/home/fedora
LANG=zh_CN.GB18030
HOME=/home/fedora
LOGNAME=fedora
CVS_RSH=ssh
SSH_CONNECTION=192.168.64.1 1057 192.168.64.3 22
```

1.3 变量

export

在任何时候，创建的变量都只是当前Shell的局部变量，不能被Shell运行的其他命令或Shell程序所用，**export命令**可以将一个局部变量提供给Shell执行的其他命令使用。

```
[fedora@localhost ~]$ export myname="ckj"
[fedora@localhost ~]$ export
declare -x HOME="/home/fedora"
declare -x HOSTNAME="localhost.localdomain"
declare -x INPUTRC="/etc/inputrc"
declare -x LANG="zh CN.GB18030"
declare -x LESSOPEN="| /usr/bin/lesspipe.sh %s"
declare -x LOGNAME="fedora"
declare -x PATH="/usr/kerberos/bin:/usr/local/bin:
/bin:/usr/bin:/home/fedora/bin"
declare -x PWD="/home/fedora"
declare -x SHELL="/bin/bash"
.
.
.
declare -x USER="fedora"
declare -x myname="ckj"
```

1.3 变量

内部变量（预定义变量）

内部变量是Linux所提供的一种特殊类型的变量，这类变量在程序中用来作出判断。在shell程序内这类变量的值是不能修改的。

部分内部变量是：

- \$#——传送给shell程序的位置参数的个数
- \$?——命令执行后返回的状态
- \$0——当前执行的进程的名称
- \$*——调用shell程序时所传送的全部参数成的单字符串

1.3 变量

ex1.sh的示范例子：

```
#ex9_3_1.sh
echo "Number of parameters is" $#
echo "Program name is" $0
echo "Parameters as a single string is" $*
```

在bash中，从命令行中执行ex9_3_1.sh如下：

```
#. ex9_3_1.sh zhang li
```

将得到如下的结果：

```
[root@localhost shell]# . ex9_3_1.sh zhang li
Number of parameters is 2
Program name is -bash
Parameters as a single string is zhang li
```

1.3 变量

用户变量

1. 变量名

用户定义的变量是最普通的shell变量。变量名是以字母或下线符开头的字母、数字和下线符序列，并且大小写字母意义不同。

在定义变量时，变量名前不应加符号\$，在引用变量的内容时则应在变量名前加\$；在给变量赋值时，等号两边一定不能留空格，若变量中本身就包含了空格，则整个字符串都要用双引号括起来。

在编写Shell程序时，为了使变量名和命令名相区别，建议所有的变量名都用大写字母来表示。

1.3 变量

2. 变量赋值

给变量赋值的过程也是声明一个变量的过程。

`set` 显示本地所有的变量

变量的赋值很简单，其一般形式是：

变量名=字符串/数字

例如：

`lcount=0`

`myname=fedora`

有时想在说明一个变量并对它设置为一个特定值后就不在改变它的值时，可以用下面的命令来保证一个变量的**只读性**：

`readonly 变量名`

1.3 变量

```
[fedora@localhost ~]$ LOCALTEST='test'  
[fedora@localhost ~]$ echo $LOCALTEST  
test  
[fedora@localhost ~]$ set  
BASH=/bin/bash  
BASH VERSION='3.1.17(1)-release'  
HISTFILE=/home/fedora/.bash_history  
HOME=/home/fedora  
HOSTNAME=localhost.localdomain  
HOSTTYPE=i686  
LANG=zh_CN.GB18030  
LC_ALL=zh_CN.GB18030  
LOGNAME=fedora
```

1.3 变量

```
[fedora@localhost ~]$ readonly LOCALTEST
[fedora@localhost ~]$ LOCALTEST='test1'
-bash: LOCALTEST: readonly variable
[fedora@localhost ~]$ readonly
declare -ir EUID="500"
declare -r LOCALTEST="test"
declare -ir PPID="2055"
declare -ir UID="500"
[fedora@localhost ~]$ set
```

1.3 变量

3. 访问变量值

可以通过给变量名加上前缀\$（美元符）来访问变量的值。

例如：

如果要把myname的值分配给变量
yourname，那么可以执行下面的命令：

yourname=\$myname

1.3 变量

4. 变量清除

unset variable-name

例：

```
[fedora@localhost ~]$ testvar="Just a test"
[fedora@localhost ~]$ echo ${testvar}
Just a test
[fedora@localhost ~]$ unset testvar
[fedora@localhost ~]$ echo $testvar


---


[fedora@localhost ~]$ [empty]
```

1.3 变量

给变量赋值

命令	环境
<code>locunt=0</code>	pdksh和bash
<code>set locunt=0</code>	tcs
<code>myname=Sanjiv</code>	pdksh和bash
<code>set name=Sanjiv</code>	tcs
<code>myname='Sanjiv Guha'</code>	pdksh和bash
<code>set name='Sanjiv Guha'</code>	tcs

1.3 变量

给变量赋值访问变量值

命令

环境

`lcount=$var` pdksh和bash

`set lcount=$var` tcsh

1.4 位置参数

1. 位置参数及引用

可以编写一个shell脚本，当从命令行或者其他shell脚本中调用它的时候，这个脚本接收若干参数。这些选项是通过Linux作为位置参数（positional parameter）提供给shell程序的。

在shell脚本中应有变量，接收实参，这类变量的名称很特别，分别是1, 2, 3, ..., 这类变量称为位置变量。位置参数1存放在位置变量\$1中，位置参数2存放在位置变量\$2中，.....，在程序中可以使用\$1, \$2,来访问。

1.4 位置参数

2. 用set命令为位置参数赋值

在shell程序中可以利用set命令为位置参数赋值或重新赋值。

(1) 一般格式: set [参数表]

(2) 说明: 该命令后面无参数时, 将显示系统中的系统变量的值; 如果有参数将分别给位置参数赋值。

1.4 位置参数

3. 位置参数移动

当位置变量个数超出9个时，就不能直接引用位置大于9的位置变量了，必须用shift命令移动位置参数。

(1) 一般形式：shift [n]

(2) 说明：每次执行时，把位置参数向左移动n位。如果没有参数，每次执行时，把位置参数向左移动1位。

1.4 位置参数

下述是一个shell程序的ex9_4_1.sh,只带了一个参数（名字），并在屏幕上显示这个名字：

```
#Name display program  
if [ $# -eq 0 ]  
then  
echo“Name not provided”  
else  
echo”Your name is $1”  
fi
```

1.4 位置参数

在bash中，如果执行ex9_4_1.sh如下：

```
#. ex9_4_1.sh
```

将得到输出：

```
Name not provided
```

但是，如果执行ex9_4_1.sh如下：

```
#. ex9_4_1.sh zhang
```

则得到如下的输出：

```
Your name is zhang
```

ex9_4_1.sh还说明了shell编程的另一个方面，即**内部变量**。在ex9_4_1.sh中的变量\$#是内部变量，并提供传给shell程序的位置参数的数目。

1.4 位置参数

位置变量小结

位置变量表示\$0,\$1.....\$9

\$0	\$1	\$2	\$3	\$4	\$5	\$6	\$7	\$8	\$9
脚本	A	B	C	D	E	F			

在脚本中使用位置参数

向系统命令传递参数

1.4 位置参数

```
#!/bin/bash
echo "This is the name of shell program: $0"
echo "This is the No.1 para: $1"
echo "This is the No.2 para: $2"
echo "This is the No.3 para: $3"
echo "This is the No.4 para: $4"
echo "This is the No.5 para: $5"
echo "This is the No.6 para: $6"
echo "This is the No.7 para: $7"
echo "This is the No.8 para: $8"
echo "This is the No.9 para: $9"
[root@localhost shell]# bash ex9_4_2.sh A B C D E F
This is the name of shell program: ex9_4_2.sh
This is the No.1 para: A
This is the No.2 para: B
This is the No.3 para: C
This is the No.4 para: D
This is the No.5 para: E
This is the No.6 para: F
This is the No.7 para:
This is the No.8 para:
This is the No.9 para:
[root@localhost shell]#
```

1.5 特殊字符

特殊字符中的某些字符

字符 说明

- \$ 指出shell变量名的开始
 - | 把标准输出通过管道传送到下个命令
 - # 标记注释开始
 - & 在后台执行进程
 - ? 匹配一个字符
 - * 匹配一个或几个字符
 - > 输出重定向操作符
 - < 输入重定向操作符
-

1.5 特殊字符

特殊字符中的某些字符(续)

字符	说明
----	----

>>	输出重定向操作符{添加到文件}
----	-----------------

<<	跟在输入结束字符串后 (HERE) 操作符
----	-----------------------

[]	列出字符的范围
----	---------

[a-z]	意a到z的全部字符
-------	-----------

[a,z]	意指a或z字符
-------	---------

.filename	执行(“源”)filename文件
-----------	-------------------

空格	在两个字之间的间隔符
----	------------

1.5 特殊字符

注释符‘#’

在 shell 编程中经常要对某些正文行进行注释，以增加程序的可读性。在 Shell 中以字符“#”开头的正文行表示注释行。

1.5 特殊字符

双引号

在字符串含有嵌入的空格时，用双引号括起来。

这是一个有关bash的例子：

```
var="teststring"
```

```
newvar="Value of var is $var"
```

```
echo $newvar
```

执行上面的三行shell程序,可得到如下的结果：

Value of var is teststring

1.5 特殊字符

单引号

利用单引号把字符括起来，以阻止shell解析变量。把前面的双引号改为单引号

```
var='teststring'
```

```
newvar='Value of var is $var'
```

```
echo $newvar
```

执行程序可得到如下的结果：

➤ Value of var is \$var

1.5 特殊字符

反斜杠

在某个字符前利用反斜杠可以阻止 shell 把后面的字符解释为特殊字符。例如，把 \$test 的值赋给变量 var。输入如下命令：

var=\$test

如下命令才把 \$test 存放在 var 中：

var=\\$test

1.5 特殊字符

```
#9.5.1
var1="teststring"
newvar1="value of var is $var1"
echo $newvar1
#9.5.2
newvar2='value of var is $var1'
echo $newvar2
#9.5.3
var2=$test
echo $var2
var3=\$test
echo $var3
[root@localhost shell]# bash ex9_5_0.sh
value of var is teststring
value of var is $var1
$test
[root@localhost shell]# █
```

1.5 特殊字符

反引号

通知 shell 执行由反引号定义的字符串。
当需要把执行命令的结果存放在变量中
时，就可以在 shell 程序中利用反引号。

例1：统计当前目录下一个文件中 test.txt
有几行并把结果存在叫做 var 的变量中：

➤ var=`wc -l test.txt`

例2：

➤ string="current directory is `pwd`"

➤ echo \$string

➤ current directour is /home/xyz

1.5 特殊字符

示例代码

```
#9.5.4
var=`wc -l /etc/passwd`
echo $var
string="current directory is `pwd`"
echo $string
```

运行结果

```
[root@localhost shell]# . ex9_5_4.sh
144 /etc/passwd
current directory is /root/shell
[root@localhost shell]# █
```

1.5 特殊字符

管道

可以通过管道把一个命令的输出传递给另一个命令作为输入。管道用竖杠 | 表示。

格式：命令1 | 命令2

举例：

- cat myfile |more
- ls -l |grep "myfile"

1.5 特殊字符

文件重定向

改变程序运行的输入来源和输出地点。

重定向标准输出

- cat file | sort > sort.out
- pwd >> path.out
- > newfile

重定向标准输入

- sort < file
- sort < name.txt > name.out

1.5 特殊字符

```
[fedora@localhost ~]$ cat readex.sh
echo -n "First Name: "
read firstname
echo -n "Last Name: "
read lastname
echo -e "Your First Name is: ${firstname}\n"
echo -e "Your Last Name is: ${lastname}\n"
[fedora@localhost ~]$ cat readex.sh | sort
echo -e "Your First Name is: ${firstname}\n"
echo -e "Your Last Name is: ${lastname}\n"
echo -n "First Name: "
echo -n "Last Name: "
read firstname
read lastname
[fedora@localhost ~]$ █
```

1.5 特殊字符

```
[fedora@localhost ~]$ sort < readex.sh
echo -e "Your First Name is: ${firstname}\n"
echo -e "Your Last Name is: ${lastname}\n"
echo -n "First Name: "
echo -n "Last Name: "
read firstname
read lastname
[fedora@localhost ~]$ █
```

1.6 运算符

- 运算符是对计算机发的指令
- 运算对象
 - 数字、字符
 - 变量
 - 表达式
- 表达式：运算符和运算对象的组合体

1.6 运算符

\$[]表示形式告诉 shell 对方括号中的表达式求值

```
[fedora@localhost ~]$ echo ${2+8}  
10
```

```
[fedora@localhost ~]$ echo $((2+8))  
10
```

```
[fedora@localhost ~]$ echo `expr 2 + 8`  
10
```

1.6 运算符

赋值运算符

=、 +=、 -

=、 *=、 /=、 %=、 &=、 ^=、 |=、 <<=、 >>=

```
[fedora@localhost ~]$ var=9
```

```
[fedora@localhost ~]$ let var+=1
```

```
[fedora@localhost ~]$ echo $var
```

```
10
```

```
[fedora@localhost ~]$ var+=1
```

```
[fedora@localhost ~]$ echo $var
```

101

```
[fedora@localhost ~]$ █
```

1.6 运算符

表达式替换

\$[] 和 \${()})

两种格式功能一样，所有的 shell 的求值都是用整数完成

\$[] 可以接受不同基数的数字

[base#n] n 表示基数从 2 到 36 的任何基数

```
[fedora@localhost ~]$ echo ${10#8+1}
```

9

```
[fedora@localhost ~]$ echo ${10#2+1}
```

3

1.7 输入和输出

echo

echo 命令可以显示文本行或变量，或者把字符串输入到文件。

- echo [option] string

- e 解析转义字符

- n 回车不换行，linux系统默认回车换行

- 转义符 (\c, \f, \t, \n , \a) c 不换行,f 进纸,t 跳格,n 换行, a 响铃。。。

1.7 输入和输出

```
echo -e "This echo's 3 new lines\n\n\n"
echo "OK"
echo
echo "This echo's 3 new lines\n\n\n"
echo "The log files have all been done">>mylogfile.txt
```

```
[root@localhost shell]# sh ex9_7_1.sh
This echo's 3 new lines
```

OK

```
This echo's 3 new lines\n\n\n
[root@localhost shell]# cat mylogfile.txt
The log files have all been done
[root@localhost shell]# █
```

1.7 输入和输出

• **read**

read 语句可以从键盘或文件的某一行文本中读入信息，并将其赋给一个变量。

read variable1 variable2 ...

➤ 如果只指定了一个变量，那么 read 将会把所有的输入赋给该变量，直到遇到第一个文件结束符或回车；

➤ 如果给出了多个变量，它们按顺序分别被赋予不同的变量。Shell 将用空格作为变量之间的分隔符。

1.7 输入和输出

```
echo -n "First Name: "
read firstname
echo -n "Last Name: "
read lastname
echo -e "Your First Name is: ${firstname}\n"
echo -e "Your Last Name is: ${lastname}\n"
```

```
[root@shuqin shell]# . ex9-7-2.sh
first name:wang
last name:shuqin
you're first name is wang
you're last name is shuqin
[root@shuqin shell]# _
```

1.8 表达式的比较

- shell程序中的test命令

在bash/pdksh中，命令test用于计算一个条件表达式的值。他们经常在条件语句和循环语句中被用来判断某些条件是否满足。

test命令的语法格式：

test expression

或者

[expression]

1.8 表达式的比较

1. 数字比较

- eq 比较两个数是否相等
- ne 比较两个数是否不等
- gt 比较一个数是否大于另一个数
- ge 比较一个数是否大于或是等于另一个数
- lt 比较一个数是否小于另一个数
- le 比较一个数是否小于或是等于另一个数

1.8 表达式的比较

例

```
if [ $1 -gt $2 ]
then
    echo "$1 > $2"
elif [ $1 -eq $2 ]
then
    echo "$1 = $2"
else
    echo "$1 < $2"
fi
```

```
if [ $1 -gt $2 ]
then
    echo "$1 > $2"
else
    if [ $1 -eq $2 ]
then
    echo "$1 = $2"
else
    echo "$1 < $2"
fi
fi
```

```
[root@localhost shell]# sh ex9_8_1.sh 10 20
10 < 20
[root@localhost shell]# sh ex9_8_1.sh 10 10
10 = 10
[root@localhost shell]# sh ex9_8_1.sh 20 10
20 > 10
```

1.8 表达式的比较

2. 字符串比较

- = 比较两个字符串是否相等，同则为“是”

- != 比较两个字符串是否不相等，不同则为“是”

- z 判断字符长度是否等于零，等于则为“是”

- n 判断字符长度是否大于零，大于零则为“是”

1.8 表达式的比较

示例代码

```
#=
if [ $1 = $2 ];then echo "$1 = $2"
else echo "$1 != $2"
fi
#!=
if [ $1 != $2 ];then echo "$1 != $2"
else echo "$1 = $2"
fi
#-n
if [ -n $1 ];then echo "$1 has a length greater than zero"
else echo "$1 has not a length greater than zero"
fi
#-z
if [ -z $2 ];then echo "$2 is empty"
else echo "$2 is not empty"
fi
```

1.8 表达式的比较

运行结果

```
[root@localhost shell]# sh ex9_8_2.sh linux unix
linux != unix
linux != unix
linux has a length greater than zero
unix is not empty
```

-

1.8 表达式的比较

3. 文件操作符

- e 如果文件存在则为真
 - r 确定是否对文件设置了读许可
 - w 确定文件是否设置了写许可
 - x 确定文件是否设置了执行许可
 - s 确定文件是否具有大于零的长度
 - d 确定文件是否为目录
 - f 确定文件是否为普通文件
 - L 确定文件是否为符号连接文件
-

1.8 表达式的比较

例

```
if [ -d $1 ]
then
    echo "$1 is a directory!"
else
    echo "$1 is not a directory!"
fi
```

```
[root@localhost shell]# sh ex9_8_3.sh /tmp
/tmp is a directory!
[root@localhost shell]# sh ex9_8_3.sh shell
shell is not a directory!
[root@localhost shell]# sh ex9_8_3.sh /root/shell
/root/shell is a directory!
```

1.8 表达式的比较

4. 逻辑操作符

逻辑操作符用来根据逻辑规则比较表达式。!, -a, -o字符表示NOT、AND和OR

! 求反（“非”）逻辑表达式

-a 逻辑AND（“与”）两个逻辑表达式

-o 逻辑OR（“或”）两个逻辑表达式

1.8 表达式的比较

例

```
if [ $1 -gt $2 -a $1 -gt $3 ]
then
    echo "max : $1"
fi
if [ $2 -gt $1 -a $2 -gt $3 ]
then
    echo "max : $2"
fi
if [ $3 -gt $1 -a $3 -gt $2 ]
then
    echo "max : $3"
fi
[root@localhost shell]# bash ex9_8_4.sh 10 108 111
max : 111
[root@localhost shell]# bash ex9_8_4.sh 410 108 111
max : 410
[root@localhost shell]# bash ex9_8_4.sh 410 1048 111
max : 1048
```

1.9 流程控制语句

- 二、条件语句
- 二、循环语句
- 三、杂项语句

1.9 流程控制语句——一、条件语句

一、条件语句

1. if 语句

2. case 语句

1.9 流程控制语句——一、条件语句

1. if 语句

if语句通过判断逻辑表达式来作出选择，在bash中的条件语句有如下的格式：

```
if [expression]; then  
    statements  
elif [expression]; then  
    statements  
else  
    statements  
fi
```

if 条件是可以嵌套的。

1.9 流程控制语句——一、条件语句

2. case 语句

case 语句是用来执行依赖于离散值或是匹配指定变量值的范围的语句。

bash 的 case 语句如下：

```
case str in  
str1|str2)  
    statements;;  
str3|str4)  
    statements;;  
*)  
    statements;;  
esac
```

每个条件下用
双分号 (;;) 来
终止语句！

1.9 流程控制语句——一、条件语句

```
case $1 in
 01|1) echo "Month is January";;
 02|2) echo "Month is February";;
 03|3) echo "Month is March";;
 04|4) echo "Month is April";;
 05|5) echo "Month is May";;
 06|6) echo "Month is June";;
 07|7) echo "Month is July";;
 08|8) echo "Month is Augest";;
 09|9) echo "Month is September";;
 10) echo "Month is October";;
 11) echo "Month is November";;
 12) echo "Month is December";;
 *) echo "Invalid parameter";;
esac
```

[root@localhost shell]# bash ex9_9_2.sh 1
Month is January
[root@localhost shell]# bash ex9_9_2.sh 01
Month is January
[root@localhost shell]# bash ex9_9_2.sh 13
Invalid parameter
[root@localhost shell]# bash ex9_9_2.sh lf
Invalid parameter

1.9 流程控制语句——二、循环语句

- 循环语句：用于重复执行一系列命令。
 - 一) for 语句
 - 二) until 语句
 - 三) while 语句
 - 四) select 语句

一) for语句

- 1. 第一种格式如下：

for 变量名 in 列表

do

命令1

命令2...

done

当变量值在列表里， for循环即执行一次所有命令， 使用变量名访问列表中取值。命令可为任何有效的shell命令和语句。变量名为任何单词。In列表用法是可选的， 如果不用它， for循环使用命令行的位置参数。in列表可以包含替换、字符串和文件名.例如：

一) for语句

```
#!/bin/bash
for loop in 1 2 3 4 5
do
    echo $loop
done
```

```
[root@shuqin shell]# bash exfor_1.sh
1
2
3
4
5
[root@shuqin shell]# _
```

一) for语句

例：假设需要把目录中的每个文件在一个叫做backup的子目录中建立备份
在bash中执行如下的程序：

```
#9.9.3
for filename in `ls`
do
    if ! [ -d $filename ] ;then
        cp $filename backup/$filename
    fi
    if [ $? -ne 0 ]; then
        echo "copy for $filename failed"
    fi
done
```

一) for语句

```
[root@localhost shell]# sh ex9_9_3.sh
[root@localhost shell]# cd backup
[root@localhost backup]# ll
total 168
-rwxr--r-- 1 root root 188 Apr  5 05:18 ex9_2_1.sh
-rw-r--r-- 1 root root 115 Apr  5 05:18 ex9_3_1.sh
-rw-r--r-- 1 root root  99 Apr  5 05:18 ex9_4_1.sh
[...]
[root@localhost backup]# rm -f *
[root@localhost backup]# ll
total 0
```

一) for语句

2.第二种格式如下：

```
for curvar >
do
statements
done
```

这种格式也可以写成如下：

```
for curvar in "$@"
do
statements
done
```

在这种格式中，对传给shell程序的每个位置参数执行一次statements。对每次循环，把位置参数的当前值赋给变量curvar。

记住\$@提供传给shell程序的一系列参数，全部参数排在一起。

一) for语句

```
#!/bin/sh
# forparam2
for params
do
    echo "You supplied $params as a command line option"
done
echo $params
done
```

```
$ forparam2 myfile1 myfile2 myfile3
You supplied myfile1 as a command line option
You supplied myfile2 as a command line option
You supplied myfile3 as a command line option
```

一) for语句

3. 第三种格式：

```
for((i=0; i<10; i++))
```

```
do
```

```
echo $i
```

```
done
```

二) until语句

用来执行一系列命令直到所指定的条件为真才能终止。

在bash中，利用如下的格式：

until expression

do

statements

done

1.9 流程控制语句——二、循环语句

3. while语句

在bash中，利用的如下的格式：

while expression

do

statements

done

1.9 流程控制语句——二、循环语句

例：求前五个偶数的和

bach的shell程序如下：

```
#9.9.4
#until
loopcount=0
result=0
until [ $loopcount -gt 4 ]
do
    loopcount=`expr $loopcount + 1`
    increment=`expr $loopcount \* 2`
    result=`expr $result \+ $increment`
done
echo $loopcount
echo "result is $result"
[root@localhost shell]# sh ex9_9_4.sh
5
result is 30
```

1.9 流程控制语句——二、循环语句

```
#while
loopcount=0
result=0
while [ $loopcount -lt 5 ]
do
    loopcount=`expr $loopcount + 1`
    increment=`expr $loopcount \* 2`
    result=`expr $result \+ $increment`
done
echo $loopcount
echo "result is $result"
```

```
[root@localhost backup]# sh ex9_9_5.sh
5
result is 30
```

1.9 流程控制语句——二、循环语句

4. select语句

select语句的格式如下：

select item in itemlist

do

statements

done

itemlist 是可选的，当未给出 itemlist 时，系统通过 item 中的项目一次重复一个，但当给出 itemlist 时，系统对 itemlist 中的每个项重复，对每次重复把 itemlist 的当前值赋给 item，而后 item 可作为执行语句的一部分。

1.9 流程控制语句——二、循环语句

如果编写一个提供用户挑选Continue或Finsh的选择菜单，则可编写如下的shell程序：

```
#!/bin/bash
select item in Continue Finsh
do
if [$item=="Finsh"]; then
    break
fi
done
```

执行select命令时，系统向用户显示一个选择数字的菜单——在这种情况下：1表示Continue，2表示Finsh。当用户选择1时，变量item包含值Continue；而当用户选择2时，变量item包含值Finsh.。当用户选择2时，即执行if语句，并终止循环。

1.9 流程控制语句——二、循环语句

```
select item in Continue Finsh
do
if [ $item = "Finsh" ];then
    break
fi
done
```

```
[root@localhost shell]# sh ex9_9_6.sh
1) Continue
2) Finish
#? 1
#? 1
#? 12
ex9_9_6.sh: line 5: [: := unary operator expected
#? 2
```

1.9 流程控制语句——三、杂项语句

- **杂项语句：**

1. shift 语句
2. break 语句
3. exit 语句

1.9 流程控制语句——三、杂项语句

1. shift语句

用来处理位置参数，并从左到右每次处理一个参数。应该记得，**位置参数是用\$1、\$2、\$3等来标识的**。shift命令的作用时，把每个位置参数向左移动一个位置，而当前的\$1丢失。

shift命令的格式如下：

➤ **shift number**

参数number是移动的数目，是可选的。当不特别指定number时，缺省值为1，即参数向左移动一个位置。当指定number时，则向左移动number个位置。

1.9 流程控制语句——三、杂项语句

```
echo $1  
shift  
echo $1  
shift  
echo $1
```

```
[root@localhost shell]# bash ex9_9_7.sh 3 4  
3  
4
```

```
[root@localhost shell]# bash ex9_9_7.sh 3 4 5  
3  
4  
5
```

1.9 流程控制语句——三、杂项语句

- ### 2. break 语句

用来终止重复执行的循环。break通常在进行一些处理后退出循环或case语句。如果是在一个嵌入循环里，可以指定跳出的循环个数。例如如果在两层循环内，用break 2刚好跳出整个循环。

- ### 3. continue语句

它不会跳出循环，只是跳过这个循环步。

1.10 函数

- 函数是shell程序中执行特殊过程的部件，并在shell程序中可以重复调用。

下面是在bash中函数定义格式。

```
func(){  
    statements  
}
```

可以调用函数如下：

```
func param1 param2 param3
```

参数 param1 param2 等是可以选择的。还能把参数作为单字符串来传送，例如 \$@。函数可以分析参数，就好像它们是传送给 shell 程序的位置参数。

1.10 函数

```
[root@shuqin shell]# cat hello_fun.sh
#!/bin/bash
function hello ()
{
    echo "this is the function hello"
}

echo "now, begin to the function hello"
hello
echo "back from the hello function"
[root@shuqin shell]# _
```

1.10 函数

```
[root@shuqin shell]# bash hello_fun.  
now, begin to the function hello  
this is the function hello  
back from the hello function  
[root@shuqin shell]# _
```

1.10 函数

```
[root@shuqin shell]# cat ex9_10_1.sh
sum()
{
    a=$1
    b=$2
    result=`expr $a + $b`
    echo $result
}

echo "enter two number"
read num1 num2
echo "result is:"
sum $num1 $num2
```

1.10 函数

```
[root@shuqin shell]# . ex9_10_1.sh
enter two number
23 56
result is:
79
```

1.10 函数

例：在传送月份数字后显示月份名或出错信息。

```
displaymonth(){  
case$1in  
    01|1)echo“MonthisJanuary”; ;  
    02|2)echo“MonthisFebruary”; ;  
    03|3)echo“MonthisMarch”; ;  
    04|4)echo“MonthisApril”; ;  
    *)echo“Invalidparameter”;  
    05|5)echo“MonthisMay”; ;  
esac  
}  
06|6)echo“MonthisJune”; ;  
displaymonth 8  
12)echo“MonthisDecember”; ;  
07|7)echo“MonthisJuly”; ;  
08|8)echo“MonthisAugust”; ;  
09|9)echo“MonthisSeptember”; ;  
10)echo“MonthisOctober”; ;  
11)echo“MonthisNovember”; ;
```

程序显示如下：
MonthisAugust

第7章 进程环境及进程属性

1

进程环境及进程属性

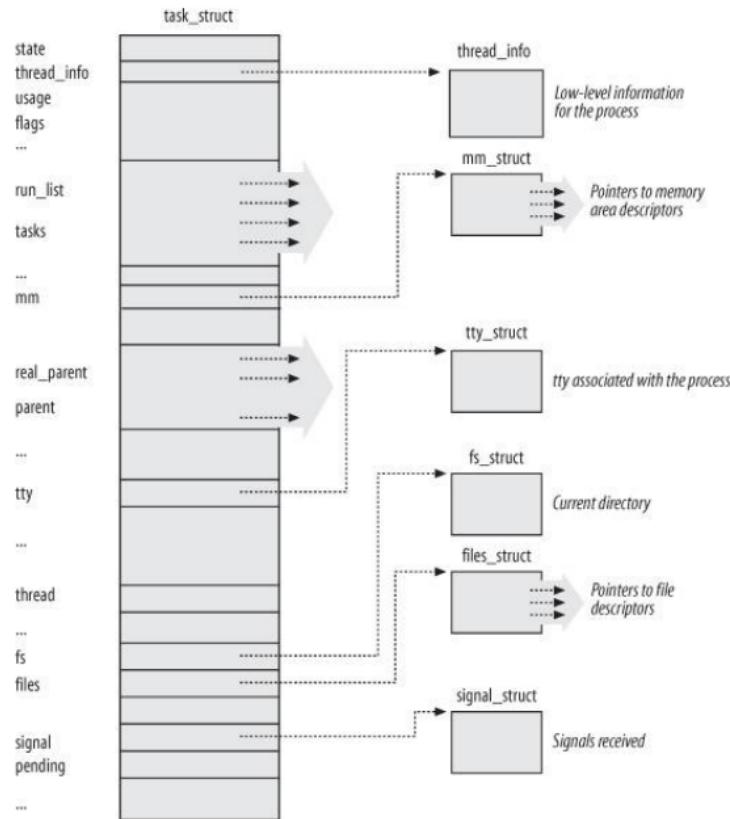
2

进程管理及控制

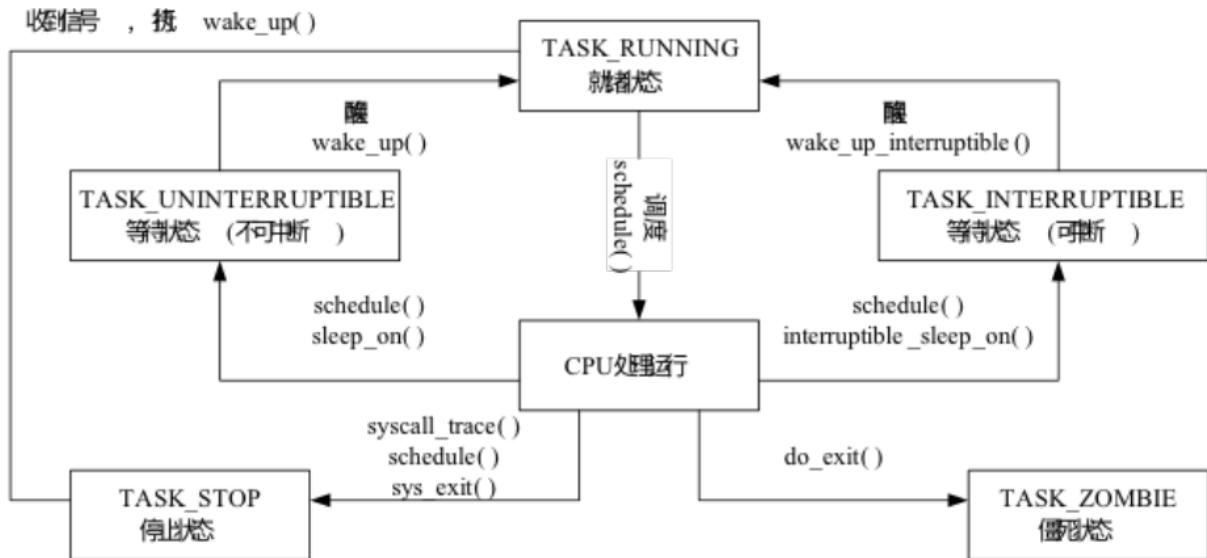
3

Linux特殊进程

进程资源



进程状态



进程基本属性 - 进程号 (PID)

```
[root@localhost ~]# ps aux //查看当前所有进程信息
USER  PID %CPU %MEM   VSZ   RSS TTY STAT START TIME COMMAND
root    1  0.0   0.3   1748   572 ?      S     04:19  0:02  init [3]
root    2  0.0   0.0     0     0 ?      SN    04:19  0:00  [ksoftirqd/0]
root    3  0.0   0.0     0     0 ?      S     04:19  0:00  [watchdog/0]
.....
```

在应用编程中，调用 `getpid()` 函数可以获得当前进程的 PID，该函数在 `/usr/include/unistd.h` 文件中声明：

```
#include </usr/include/unistd.h>
extern __pid_t getpid(void);
```

此函数没有参数。如果执行成功将返回当前进程的 PID，类型为 `pid_t`，如果执行失败则返回-1，错误原因储存于 `errno` 中。`pid_t` 类型其实就是 `int` 类型

父进程号 (PPID)

```
//come from /usr/include/unistd.h  
extern __pid_t getppid (void);
```

此函数没有参数。如果执行成功将返回当前进程的父进程 PID，类型为 pid_t，如果执行失败则返回-1，错误原因存储在 errno 中。

进程组号 (PGID)

- 进程组是一个或多个进程的集合。它们与同一作业相关联，可以接受来自同一终端的各种信号（关于信号的概念参阅第8章）。每个进程组都有唯一的进程组号，进程组号是可以在用户层修改的。

```
extern __pid_t getpgid(__pid_t pid);
```

此函数参数 pid 为要获得进程组号 (PGID) 的进程号 (PID)，如果为 0 表示获取当前进程组号 (PGID)，否则为指定进程的 PGID。如果执行成功将返回当前进程的进程组号 (PGID)，如果执行失败则返回 -1，错误原因存储在 errno 中。

加入一个现有的组或者一个新进程组的系统调用函数 setpgid() 声明如下：

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

其第 1 个参数为欲修改进程 PGID 的进程 PID，第 2 个参数为新的进程组号，如果这两个参数相等，则由 pid 指定的进程变成进程组组长；如果 pid 为 0，则修改当前进程的 PGID；如果 pgid 是 0，则由 pid 指定的进程的 PID 将用于进程组号 PGID。

会话

- 会话 (session) 是一个或多个进程组的集合。系统调用函数 getsid() 用来获取某个进程的会话号 SID。

```
extern __pid_t getsid (__pid_t __pid);
```

如果 pid 是 0，返回调用进程的会话号 SID，一般来说，该值等于进程组号 PGID。如果 pid 并不属于调用者所在的会话，那么调用者就不能得到。

- 某进程的会话 SID 是可以修改的，函数 setsid() 用来创建新会话，声明如下：

```
extern __pid_t setsid (void);
```
- 该进程变成新会话的进程 (session leader)，会话的进程是创建该会话的进程。
- 该进程成为一个新进程组的组长进程。新进程组 PGID 是该调用进程的 PID。
- 该进程没有控制终端。如果在调用 setsid 之前该进程就有一个控制终端，那么这种联系也会被中断。

控制终端

- 会话和进程组有以下一些特点：
 - (1) 一个会话可以有一个控制终端，建立与控制终端连接的会话首进程被称为控制进程。
 - (2) 一个会话中的几个进程组可被分成一个前台进程组和几个后台进程组，如果一个会话有一个控制终端，则它有一个前台进程组。
 - (3) 无论何时键入终端的中断键 (DELETE或Ctrl+C)，就会将中断信号发送给前台进程组的所有进程，无论何时键入终端的退出键 (Ctrl+\)，就会将退出信号发送给前台进程组的所有进程，如果终端检测到调制解调器 (或网络) 已经断开连接，则将挂断信号发送给控制进程 (会话首进程) 。

终端处理函数

函数 tcgetpgrp() 获取当前前台进程组的进程组号，该函数声明如下：

```
pid_t tcgetpgrp(int filedes);
```

函数 tcgetpgrp 返回与打开的终端（由 filedes 指定）相关联前台进程组的进程组号。

tcsetpgrp() 函数用来设置某个进程组是前台还是后台进程组，函数声明如下：

```
pid_t tcsetpgrp(int filedes, pid_t pgrp_id);
```

如果进程有一个控制终端，则将前台进程组 ID 设置为 pgrp_id，pgrp_id 的值应该是在同一会话中的一个进程组的 ID，filedes 为控制终端的文件描述符。

函数 tcgetsid() 可以获取控制终端的会话首进程的会话 ID，该函数声明如下：

```
pid_t tcgetsid(int filedes);
```

以下是使用以上函数的应用示例程序。

进程用户属性

- 进程真实用户号 (RUID)
- 真实用户组号 (RGID)
- 有效用户号 (EUID)
- 有效用户组号 (EGID)

进程真实用户号 (RUID)

对于进程而言，创建者该进程的用户 UID（执行此程序的用户）为此进程真实用户号 (RUID)。可以通过调用 `getuid()` 函数来获得当前进程的真实用户号 (RUID)。其函数定义在 `/usr/include/unistd.h` 文件中。函数声明为：

```
//come from /usr/include/unistd.h
extern __uid_t getuid (void)
```

此函数无参数，如果执行成功将返回当前进程的 UID；如果执行失败则返回 -1，错误原因存储在 `errno` 中。

进程有效用户号 (EUID)

- EUID主要用于权限检查。多数情况下，EUID和UID相同。如果可执行文件的setuid位有效，在除该文件的拥有者之外的用户运行该程序时，EUID和UID则不相同。即当某可执行文件设置了setgid位（见文件属性章节介绍）后，任何用户（包括root用户）运行此程序时，其有效用户组EUID为该文件的拥有者。

普通用户能够修改自己的密码的原因

```
[root@localhost ~]# ls /usr/bin/passwd -l
```

```
-r-s--x--x 1 root root 18852 Mar 7 2005 /usr/bin/passwd
```

用户密码存储于文件/etc/passwd，其访问权限如下：

```
[root@localhost ~]# ls -l /etc/passwd
```

```
-rw-r--r-- 1 root root 1582 Mar 19 06:37 /etc/passwd //文件的创建者为root, root有写权限
```

- /etc/passwd文件用来存储所有用户信息，任何用户都可以修改自己的密码，显然，其它用户在执行/usr/bin/passwd命令时修改了/etc/passwd文件（并不是说可以使用vi编辑器修改），但是，通过查看/etc/passwd文件的权限，发现普通用户对此文件仅有读的权限。是什么原因导致普通用户可以修改/etc/passwd文件呢？
- 这是因为用户执行“/usr/bin/passwd”命令时，/usr/bin/passwd文件设置了setuid位，在执行此程序（/usr/bin/passwd）时，该用户所拥有的权限等同于文件“/usr/bin/passwd”的拥有者root的权限，而root用户拥有对/etc/passwd文件写的权限，因此普通用户可以通过/usr/bin/passwd来修改/etc/passwd文件的内容。
- 如果清除掉“/usr/bin/passwd”文件的setuid权限位，普通用户就不能修改自己的密码了。

进程用户组号 (GID)

- 创建进程的用户所在的组号为该进程的进程用户组号 (GID)。可以通过调用getgid()函数来获得当前进程的真实用户组号 (GID)。

```
//come from /usr/include/unistd.h
extern __uid_t getgid (void)
```

此函数无参数，如果执行成功将返回当前进程的 GID。如果执行失败则返回-1，错误原因存储在 `errno` 中。

有效进程用户组号 (EGID)

一般情况下，EGID 和 GID 相同，但是，当某可执行文件设置了 setgid 位（见文件属性章节介绍）后，任何用户（包括 root 用户）运行此程序时，其有效用户组号 EGID 为该文件的拥有者所在的组。其原来与 EUID 类似。

可以通过调用 `getegid` 函数来获得当前进程的有效用户组号 (EGID)。其函数定义在 `/usr/include/unistd.h` 文件中。函数声明为：

```
//come from /usr/include/unistd.h
extern __uid_t getegid (void)
```

此函数无参数，如果执行成功将返回当前进程的 EGID；如果执行失败则返回-1，错误原因存储在 `errno` 中。

第7章 进程环境及进程属性

1

进程环境及进程属性

2

进程管理及控制

3

Linux特殊进程

创建进程

```
//come from /usr/include/unistd.h  
extern __pid_t fork (void);
```

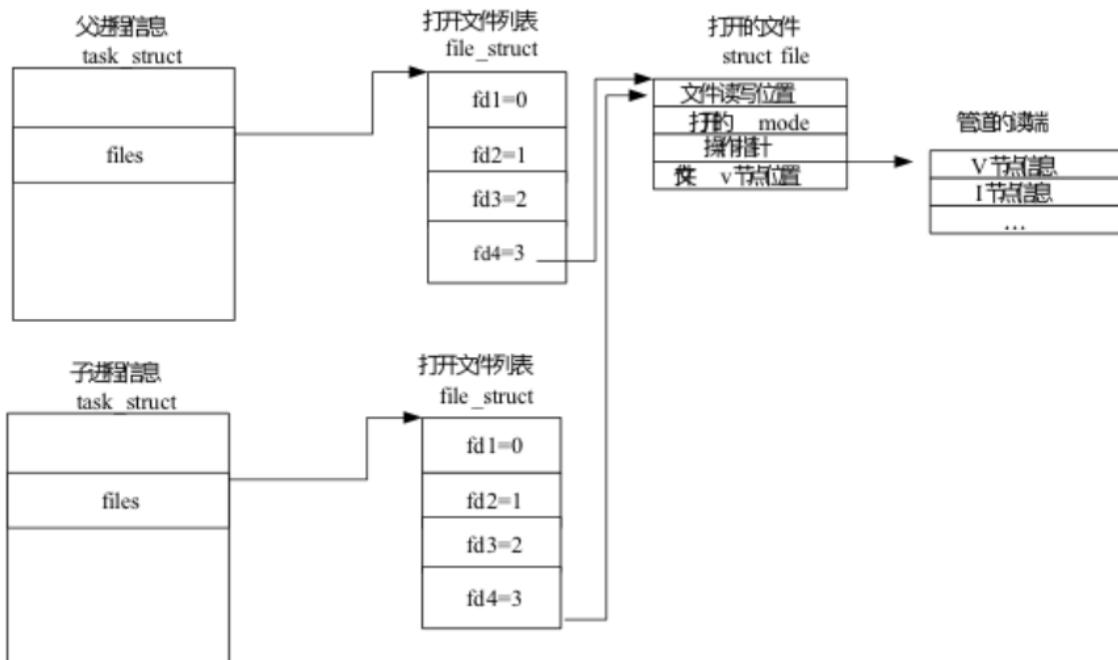
此函数没有参数。返回值如下：

- 如果执行成功，在父进程中将返回子进程（新创建的进程）的 PID，类型为 pid_t，在子进程将返回 0，以区别父子进程。
 - 如果执行失败，则在父进程中返回 -1，错误原因存储在 errno 中。
-
- fork 函数调用成功后，其子进程会复制父进程的几乎所有信息（除 PID 等信息），主要复制父亲进程的代码段、数据段、BSS、堆、栈（关于进程结构参阅本书第 3 章）、打开的文件描述符（但共用同一个文件表项）。
 - 另外，子进程从父进程继承下列属性：实际用户/组号、有效用户/组号以及保留的用户/组号、进程组号、环境变量、对文件的执行时关闭标志、信号处理方式设置、信号掩码、当前工作目录、根目录、文件模式创建掩码、文件大小限制等信息。

示例

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int main(int argc,char *argv[])
{
    pid_t pid;
    if((pid=fork())==1)           //创建子进程，在父进程执行
        printf("fork error");
    printf("bye!\n");             //父子进程都将执行这一代码
    return 0;
}
[root@localhost yangzongde]# gcc -o fork_example fork_example01.c //编译
[root@localhost yangzongde]# ./fork_example                         //执行
bye!                                                               //父子进程都将打印这条消息
bye!
```

对打开文件的处理



验证fork是否在子进程复制验证代码

- 验证数据段（使用全局、静态初始化变量），BSS段（未初始化全局，静态变量），栈（局部变量），堆（malloc返回空间）。
- 代码见教材。

vfork

- /* Clone the calling process, but without copying the whole address space. The calling process is suspended until the new process exits or is replaced by a call to `execve'. Return -1 for errors, 0 to the new process, and the process ID of the new process to the old process. */
- extern __pid_t vfork (void) ;
- #endif

Vfork与fork比较验证代码

- 见教材代码。

execX函数执行新代码

```
//come from /usr/include/unistd.h
/* Execute PATH with all arguments after PATH until a NULL pointer and environment from `environ'. */
extern int execl (_const char * _path, _const char * _arg, ...)
```

```
/* Execute PATH with all arguments after PATH until a NULL pointer,
and the argument after that for environment. */
extern int execle (_const char * _path, _const char * _arg, ...)
```

```
/* Execute FILE, searching in the `PATH' environment variable if it contains no slashes, with all
arguments after FILE until a NULL pointer and environment from `environ'. */
extern int execlp (_const char * _file, _const char * _arg, ...)
```

```
/* Execute PATH with arguments ARGV and environment from `environ'. */
extern int execv (_const char * _path, char * _const _argv[])
```

```
/* Execute FILE, searching in the `PATH' environment variable if it contains
no slashes, with arguments ARGV and environment from `environ'. */
extern int execvp (_const char * _file, char * _const _argv[])
```

execX函数比较

函数	使用文件名	使用路径名	使用参数表 (函数出现字母l)	使用argv (函数出现字母v)
execl		√	√	
execlp	√		√	
execle		√	√	
execv		√		√
execvp	√			√
execve		√		√

执行新代码对打开文件的处理

- 在执行exec系列函数时，默认情况下，新代码对于可以使用在原来代码中打开的文件描述符，即执行exec系列函数时，并不关闭原来的文件描述符。
- 但如果调用fcntl函数
`fcntl(fd,F_SETFD,FD_CLOEXEC)`
- 即关闭FD_CLOEXEC项，则在执行execX系列函数后将关闭原来打开的文件描述符。

等待进程结束

```
//come from /usr/include/sys/Wait.h  
/* Wait for a child to die. When one does, put its status in *STAT_LOC  
and return its process ID. For errors, return (pid_t) -1. */  
extern __pid_t wait (__WAIT_STATUS __stat_loc); //wait() 函数
```

- 调用wait()函数的父亲进程将等待该进程的任意一个子进程结束后才继续执行（如果有多个子进程，只需要等待其中的一个进程）。

waitpid

```
//come from /usr/include/sys/Wait.h
extern __pid_t waitpid (__pid_t __pid, int *__stat_loc, int __options);
```

其第一个参数为进程 PID 值，该值可以设置为如下范围内的值：

PID>0，表示等待进程 PID 为该 PID 值的进程结束。

PID=-1，表示等待任意进程结束。

PID=0，表示等待与当前进程的进程组 PGID 一致的进程结束。

PID<-1，表示等待进程组 PGID 是此值的绝对值的进程结束。

第二个参数为调用它的函数中某个变量地址，如果执行成功，将用来存储结束进程的结束状态。

第三个参数为等待选项，可以设置为 0，亦可为 WNOHANG 和 WUNTRACED。

退出进程

- 可以通过以下方式结束进程。
 - 向exit或_exit发布一个调用。
 - 在main函数中执行return。
 - 隐含的离开main函数。

函数说明

注册一个函数在 exit 退出时调用。其函数声明如下：

```
/* Register a function to be called when `exit' is called. */  
extern int atexit (void (*__func) (void))
```

```
/* Register a function to be called with the status given to `exit' and the given argument. */  
extern int on_exit (void (*__func) (int __status, void * __arg), void * __arg) ;
```

示例代码

```
#include<stdlib.h>
void test_exit(int status,void *arg)
{
    printf("before exit()\n");
    printf("exit %d\n",status);
    printf("arg=%s\n",(char *)arg);
}

int main()
{
    char *str="test";
    on_exit(test_exit,(void *)str);           //在退出前执行test_exit()函数
    exit(4321);
}
```

exit 与 _exit

```
#include<stdlib.h>
int main(int argc,char *argv[])
{
    printf("output\n");
    printf("content in buffer"); //后面不能有回车
    _exit(0); //只输出output，没有清理缓冲区
    //exit(0); //改为此句将输出output \ncontent in buffer
}
```

```
[root@~]# gcc -o _exit_example _exit_example.c
```

```
[root@~]# ./_exit_example
```

```
output
```

exit与return的区别

- C语言关键字与函数exit()在main函数退出时有相似之处，但两者有本质的区别：
- return 退出当前函数主体，exit()函数退出当前进程，因此，在main函数里面return(0)和exit(0)完成一样的功能。
- return仅仅从子函数中返回，而子进程用exit()退出，调用exit()时要调用一段终止处理程序，然后关闭所有I/O流。

示例代码

```
int test(void)
{
    printf("a\n");
    sleep(1);
    //exit(0);
    return 0;
}

int main(int argc,char *argv[])
{
    int i;
    i++;
    printf("i=%d\n",i);
    while(1)
        test();
    return 0;
}
```

修改进程用户相关信息

- access核实用户权限

```
extern int access (const char *name, int type) ;
```

此函数的第一个参数为欲访问的文件（需包含路径），第二参数为相应的访问权限，文件权限定义如下：

```
//come from /usr/include/unistd.h
/* Values for the second argument to access. These may be OR'd together.*/
#define R_OK 4      /* Test for read permission.*/          //读权限
#define W_OK 2      /* Test for write permission.*/        //写权限
#define X_OK 1      /* Test for execute permission. */       //执行权限
#define F_OK 0      /* Test for existence.*/           //文件是否存在
```

设置进程真实用户RUID

```
extern int setuid (_uid_t _uid);
```

此函数有一个参数，即欲设置的进程真实用户号（RUID）：

- 如果当前用户超级用户，则将设置真实用户号（UID）、有效用户号 EUID 为指定 ID，并返回 0 以标识成功；
- 如果当前用户是普通用户，且欲设置的 UID 值为自己的 UID，则可以修改成功，否则则无权修改，此函数将返回 -1。

设置进程有效用户EUID

函数 `seteuid()` 用来设置有效用户号（EUID）。该函数声明如下：

```
/* Set the effective user ID of the calling process to UID. */  
extern int seteuid (_uid_t __uid);
```

- 如果是超级用户，将设置有效用户号（EUID）为指定 ID，如果调用成功，该函数将返回 0；如果调用失败，将返回 -1 并有以下错误代码设置。
- 如果是普通用户，可以设置 EUID 为自己的 ID，如果想设置为其它用户则不予更改，

将返回失败。此外，还可以调用 `setegid` 来设置有效 EGID。

`setegid()` 函数可以用来设置有效用户组 ID，原理与 `seteuid()` 类型，函数声明如下：

```
/* Set the effective group ID of the calling process to GID. */  
extern int setegid (_gid_t __gid);
```

第7章 进程环境及进程属性

1

进程环境及进程属性

2

进程管理及控制

3

Linux特殊进程

守护进程

- 守护进程（Daemon）是运行在后台的一种特殊进程，其脱离于终端，之所以脱离于终端是为了避免进程被任何终端所产生的信息所打断，其在执行过程中的信息也不在任何终端上显示。守护进程周期性地执行某种任务或等待处理某些发生的事情，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。
- 一般情况下，守护进程可以通过以下方式启动：
 - 在系统启动时由启动脚本启动，这些启动脚本通常放在/etc/rc.d目录下；
 - 利用inetd超级服务器启动，如telnet等；
 - 由cron定时启动以及在终端用nohup启动的进程也是守护进程。

守护进程编程要点

(1) 屏蔽一些有关控制终端操作的信号（关于信号内容请参阅第8章）。这是为了防止在守护进程没有正常运行起来前，控制终端受到干扰退出或挂起。基本示例代码如下：

```
signal(SIGTTOU,SIG_IGN);  
signal(SIGTTIN,SIG_IGN);  
signal(SIGTSTP,SIG_IGN);  
signal(SIGHUP ,SIG_IGN);
```

(2) 在后台运行。这是为了避免挂起控制终端将其放入后台执行。方法是在进程中调用 fork 使父进程终止，让其在子进程中后台执行。

```
if(pid=fork())  
    exit(0); //是父进程，结束父进程，子进程继续
```

(3) 脱离控制终端和进程组。因为进程属于一个进程组，进程组号（PGID）就是进程组长的进程号（PID）。同进程组中的进程共享一个控制终端，这个控制终端通常是创建进程的 shell 登录终端。而控制终端和进程组通常是从父进程继承下来的。需要摆脱它们，使之不受它们的影响。因此需要调用 setsid() 使子进程成为新的会话组长，如下示例代码所示：

```
setsid();
```

守护进程编程要点

(4) 禁止进程重新打开控制终端。现在，进程已经成为无终端的会话组长。但它可以重新申请打开一个控制终端。可以通过使进程不再成为会话组长来禁止进程重新打开控制终端，采用的办法是再次创建一个子进程，如下示例所示：

```
if(pid=fork()) +  
    exit(0);           //结束第一子进程，第二子进程继续（第二子进程不再是会话组长） +
```

(5)[关闭打开的文件描述符。进程从创建它的父进程那里继承了打开的文件描述符，]

一般情况下，不再需要，包括标准输入输出（因为守候进程是后台执行）。如不关闭，将会浪费系统资源，造成进程所在的文件系统无法卸下以及引起无法预料的错误。按如下方法关闭它们：

```
#define NOFILE      256    //不同的系统有不同的限制。  
for(i=0;i< NOFILE;i++)      //关闭打开的文件描述符。  
    close(i); +
```

(5) 改变当前工作目录。进程活动时，其工作目录所在的文件系统不能卸下。因此，一般需要将守候进程的工作目录改变到其根目录。对于需要转储核心，写运行日志的进程将工作目录改变到特定目录如/tmp。如以下示例所示：

```
chdir("/") +
```

守护进程编程要点

(6) 重设文件创建掩模。进程从创建它的父进程那里继承了文件创建掩模。它可能修改守护进程所创建的文件的存取权限。为防止这一点，将文件创建掩模清除：

```
umask(0);
```

(7) 处理 SIGCHLD 信号（子进程退出信号）。但对于某些进程，特别是服务器进程往往在请求到来时生成子进程处理请求。如果父进程不等待子进程结束，子进程将成为僵尸进程（zombie）从而占用系统资源。如果父进程等待子进程结束，将增加父进程的负担，影响服务器进程的并发性能。在 Linux 下可以简单地将 SIGCHLD 信号的操作设为 SIG_IGN 来解决这一问题。

```
signal(SIGCHLD,SIG_IGN); //signal函数的使用参阅第8章内容
```

这样，内核在子进程结束时不会产生僵尸进程。

日志信息及其管理

在 Linux 系统下，日志守护进程 `syslogd` 专门负责管理日志信息：

```
[root@localhost root]# ps -aux |grep syslogd
root      1592  0.0  0.1  1440  168 ?        S     10:22   0:00 syslogd -m 0
```

日志守护进程 `syslogd` 根据配置文件 `/etc/syslog.conf` 决定各进程发送的日志信息写入的文件内容，如下所示为配置文件内容：

```
[root@localhost root]# cat /etc/syslog.conf
# Log all kernel messages to the console.
# Logging much else clutters up the screen.
#kern.*                                     /dev/console
# Log anything (except mail) of level info or higher.
```

建立与日志守候进程联系

```
extern void openlog ( __const char * __ident, int __option, int __facility);
```

openlog()将打开当前程序与日志守候进程之间的联系。其共有三个参数：

第1个参数：要向每个消息加入的字符串，一般可设置为为当前进程名；

第2个参数：用来描述已打开选项。如下所示：

```
/*
 * Option flags for openlog.
 *
 * LOG_ODELAY no longer does anything.
 * LOG_NDELAY is the inverse of what it used to be.
 */

#define LOG_PID      0x01/* log the pid with each message */          //日志中包含进程ID
#define LOG_CONS     0x02/* log on the console if errors in sending */    //如果消息无法送到日志服务，将输出到终端
#define LOG_ODELAY   0x04/* delay open until first syslog() (default) */ //直到调用syslog才打开
#define LOG_NDELAY   0x08/* don't delay open */                         //立即打开
#define LOG_NOWAIT   0x10/* don't wait for console forks: DEPRECATED */ //废弃
#define LOG_PERROR   0x20/* log to stderr as well */                  //错误信息同时发送到stderr
```

openlog()

第3个参数：消息的类型，决定将消息写入到哪个日志文件中。

```
/* facility codes */
#define LOG_KERN      (0<<3) /* kernel messages */           //内核内容
#define LOG_USER      (1<<3) /* random user-level messages */ //随机用户级
#define LOG_MAIL      (2<<3) /* mail system */                //电子邮件
#define LOG_DAEMON    (3<<3) /* system daemons */             //系统守候进程
#define LOG_AUTH      (4<<3) /* security/authorization messages */ //安全认证消息
#define LOG_SYSLOG    (5<<3) /* messages generated internally by syslogd */ //syslogd产生的
#define LOG_LPR       (6<<3) /* line printer subsystem */        //行打印机系统
#define LOG_NEWS      (7<<3) /* network news subsystem */       //网络子系统
#define LOG_UUCP      (8<<3) /* UUCP subsystem */              //UUCP子系统
#define LOG_CRON      (9<<3) /* clock daemon */                 //时钟
#define LOG_AUTHPRIV  (10<<3) /* security/authorization messages (private) */ //私有的安全认证
#define LOG_FTP       (11<<3) /* ftp daemon */                  //ftp守候进程
```

写日志信息

```
extern void syslog (int __pri, __const char * __fmt, ...);
```

本函数的第 1 个参数决定日志级别，常用的级别如下示例所示：

```
#define LOG_EMERG    0 /* system is unusable */           //系统不可用
#define LOG_ALERT     1 /* action must be taken immediately */ //必须立即报告的
#define LOG_CRIT      2 /* critical conditions */          //冲突
#define LOG_ERR       3 /* error conditions */            //错误
#define LOG_WARNING   4 /* warning conditions */          //警告
#define LOG_NOTICE    5 /* normal but significant condition */ //普通但有特殊标识
#define LOG_INFO      6 /* informational */                //消息
#define LOG_DEBUG     7 /* debug-level messages */         //调度级
```

第 2 个参数为日志输出格式，类似于 printf 函数的第 2 个参数。

守候进程应用示例

- 见教材代码。

孤儿进程与僵死进程

- 因父亲进程先退出而导致一个子进程被init进程收养的进程为孤儿进程。
- 而已经退出但还没有回收资源的进程为僵死进程。
- 示例代码见教材。

要变成 daemon，一个程序需要完成下面的步骤。

1. 执行一个 fork()，之后父进程退出，子进程继续执行。（结果是 daemon 成为了 init 进程的子进程。）之所以要做这一步是因为下面两个原因。
 - 假设 daemon 是从命令行启动的，父进程的终止会被 shell 发现，shell 在发现之后会显示出另一个 shell 提示符并让子进程继续在后台运行。
 - 子进程被确保不会成为一个进程组首进程，因为它从其父进程那里继承了进程组 ID 并且拥有了自己的唯一的进程 ID，而这个进程 ID 与继承而来的进程组 ID 是不同的，这样才能够成功地执行下面一个步骤。
2. 子进程调用 setsid()（参见 34.3 节）开启一个新会话并释放它与控制终端之间的所有关联关系。
3. 如果 daemon 从来没有打开过终端设备，那么就无需担心 daemon 会重新请求一个控制终端了。如果 daemon 后面可能会打开一个终端设备，那么必须要采取措施来确保这个设备不会成为控制终端。这可以通过下面两种方式实现。
 - 在所有可能应用到一个终端设备上的 open() 调用中指定 O_NOCTTY 标记。
 - 或者更简单地说，在 setsid() 调用之后执行第二个 fork()，然后再次让父进程退出并让孙子进程继续执行。这样就确保了子进程不会成为会话组长，因此根据 System V 中获取终端的规则（Linux 也遵循了这个规则），进程永远不会重新请求一个控制终端（参见 34.4 节）。

4. 清除进程的 umask (参见 15.4.6 节) 以确保当 daemon 创建文件和目录时拥有所需的权限。
5. 修改进程的当前工作目录, 通常会改为根目录 (/)。这样做是有必要的, 因为 daemon 通常会一直运行直至系统关闭为止。如果 daemon 的当前工作目录为不包含/的文件系统, 那么就无法卸载该文件系统 (参见 14.8.2 节)。或者 daemon 可以将工作目录改为完成任务时所在的目录或在配置文件中定义的一个目录, 只要包含这个目录的文件系统永远不会被卸载即可。如 cron 会将自身放在/var/spool/cron 目录下。
6. 关闭 daemon 从其父进程继承而来的所有打开着的文件描述符。(daemon 可能需要保持继承而来的文件描述的打开状态, 因此这一步是可选的或者是可变更的。)之所以需要这样做的原因有很多。由于 daemon 失去了控制终端并且是在后台运行的, 因此让 daemon 保持文件描述符 0、1 和 2 的打开状态毫无意义, 因为它们指向的就是控制终端。此外, 无法卸载长时间运行的 daemon 打开的文件所在的文件系统。因此, 通常的做法是关闭所有无用的打开着的文件描述符, 因为文件描述符是一种有限的资源。

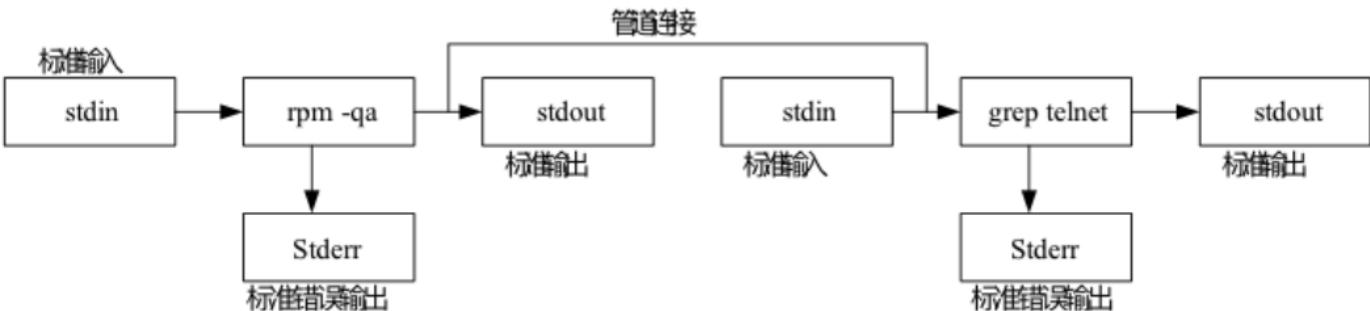
7. 在关闭了文件描述符 0、1 和 2 之后，daemon 通常会打开 /dev/null 并使用 dup2()（或类似的函数）使所有这些描述符指向这个设备。之所以要这样做是因为下面两个原因。

- 它确保了当 daemon 调用了在这些描述符上执行 I/O 的库函数时不会出乎意料地失败。
- 它防止了 daemon 后面使用描述符 1 或 2 打开一个文件的情况，因为库函数会将这些描述符当做标准输出和标准错误来写入数据（进而破坏了原有的数据）。

第8章 进程间通信－管道和信号

- 1** 进程间通信－PIPE
- 2** 进程间通信—FIFO
- 3** 信号中断处理

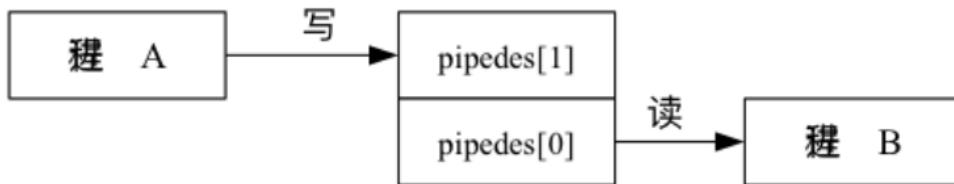
管道示例



创建无名管道

```
extern int pipe (int _pipedes[2])
```

此函数的参数是一个整型数组（下标为 2）。如果执行成功，pipe 将存储两个整型文件描述符于 `_pipedes[0]` 和 `_pipedes[1]` 中，它们分别指向管道的两端。如果系统调用失败，将返回 -1。



文件描述符重定向

- (1) cat<test01
- (2) cat>test02<test01
- (3) cat>test02 2>error <test01
- (4) cat>test02 1&2 <test01
- (5) cat 1&2 1>test02<test01

dup() / dup2()

```
extern int dup (int _fd)
```

dup()会复制一份原来已经打开的文件描述符，新的描述符指向系统文件表中下一个可用的最小非负文件描述符，它将与原来的文件描述符共享同一个文件指针（包括文件指针的偏移量值），并拥有相同的文件权限及模式。当调用 dup()时，总返回下一个最低的可用文件描述符。

例如下面语句即可以将输出重定向到管道：

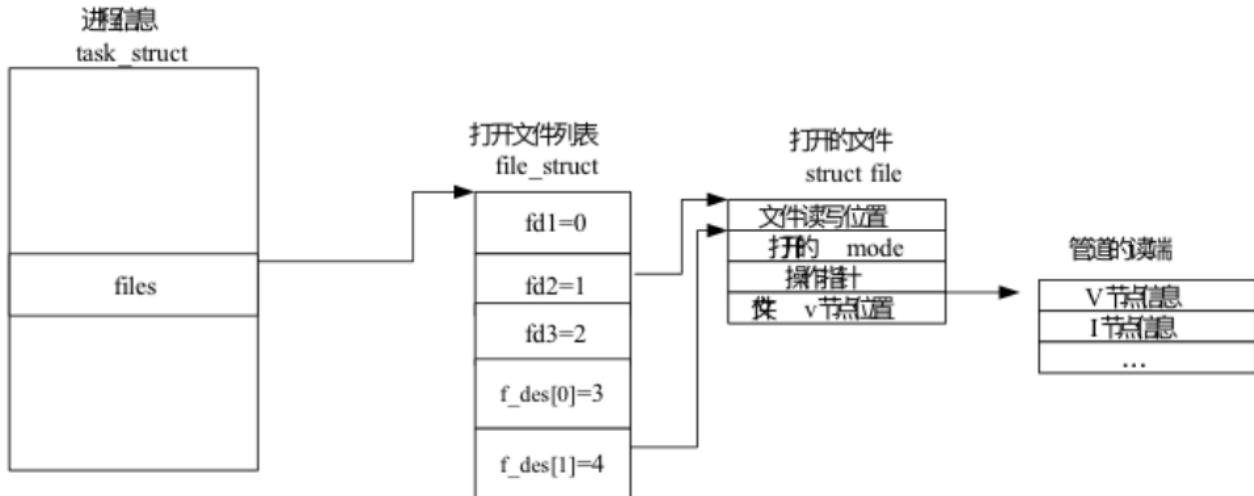
```
int f_des[2];
pipe(f_des);           //创建无名管道。
close(fileno(stdout)); //关闭标准输出设备。
dup(f_des[1]);         //返回一个最低的可用的文件描述符,即已经关闭的标准输出设备(文件
描述符为1)。
```

此后，所有写向标准输出的数据都将写入到管道中。因此，要复制标准输出输入设备，应先关闭这一设备（默认是打开的），然后再复制。

```
extern int dup2 (int _fd, int _fd2)
```

dup2()有两个参数，fd 和 fd2，fd2 是小于文件描述符的最大允许值的非负整数。如果 fd2 是一个已打开的文件描述符，则首先关闭该文件，然后再复制。

复制文件描述符



实现who|sort

- 即使使用无名管道将执行who命令的进程与执行sort命令的进程联系在一起，将当前系统用户信息按排序方法输出。
- 过程及示例代码见教材。

流重定向

```
extern FILE *popen (_const char *_command, _const char *_modes);
```

popen 函数创建 (fork) 一个子进程，并在子进程中执行第一个参数程序，同时返回一个文件指针，即第一个参数 “*_command” 指向要执行的命令（可执行程序）的指针。第二个参数表示 I/O 模式的类型。

- 如果此命令的输出将做为其它命令的输入，即输出重定向，则需要设置其第 2 个参数为 “r” 权限；
- 如果此向命令输入数据要从其它命令输出数据，即输入重定向，则需要其第 2 个参数为 “w” 权限。

在使用完后重定向后，需要使用 pclose() 关闭相应的流对象，该函数声明如下：

```
/* Close a stream opened by popen and return the status of its child. */
```

```
extern int pclose (FILE * _stream);
```

第8章 进程间通信－管道和信号

1

进程间通信－PIPE

2

进程间通信—FIFO

3

信号中断处理

FIFO应用示例

```
[root@localhost ~]# mknod PIPETEST p          //命令为mknod, 参数为p
[root@localhost ~]# ls PIPETEST -l
prw-r--r-- 1 root root 0 Apr 14 15:16 PIPETEST
[root@localhost ~]# cat test.c                  //查看test.c文件的基本内容
#include <stdio.h>
#include <stdlib.h>
static char buff [256];
static char* string;
int main ()
{
    string=	buff;
    printf ("Please input a string: ");
    gets (string);
    printf ("\nYour string is: %s\n", string);
}
```

管道示例

```
[root@localhost ~]# cat test.c >PIPETEST&
```

//将test.c内容输入到管道文件

```
[1] 2779
```

```
[root@localhost ~]# cat<PIPETEST
```

//从管道中读数据，内容为test.c内容

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

```
#include <stdlib.h>
```

```
static char buff [256];
```

```
static char* string;
```

```
int main ()
```

```
{
```

```
    string=	buff;
```

```
    printf ("Please input a string: ");
```

```
    gets (string);
```

```
    printf ("\nYour string is: %s\n", string);
```

```
}
```

```
[1]+ Done
```

```
cat test.c >PIPETEST
```

创建FIFO

```
/* Create a new FIFO named PATH, with permission bits MODE. */  
extern int mkfifo (const char * _path, mode_t _mode) ;
```

mkfifo()会根据参数建立特殊的有名管道文件，该文件必须不存在，而参数 mode 为该文件的权限，mkfifo()建立的 FIFO 文件其他进程都可以用读写一般文件的方式存取。当使用 open()函数打开 FIFO 文件时，O_NONBLOCK 会有影响。

如果执行成功将返回 0，否则返回-1，失败原因存储于 errno 中。

应用示例

- 亲缘关系进程使用有名管道通信应用实例
 - 见教材。
- 非亲缘关系进程使用有名管道通信应用实例
 - 见教材。

管道基本特点总结

- 两类型管道具有以下特点：
 - (1) 管道是特殊类型的文件，在满足先入先出的原则条件下可能进行读写，但不能定位读写位置。
 - (2) 管道是单向的，要实现双向，需要两个管道。无名管道只能实现亲缘关系进程间通信（即无名管道的两个文件描述符可以被两者都访问到），而有名管道以磁盘文件的方式存在，可以实现本机任意两进程间通信。
 - (3) 无名管道阻塞问题。无名管道无须显式打开，创建时直接返回文件描述符，而在读写时需要确实对方的存在，否则将退出。即如果当前进程向无名管道的写数据时，必须确定其别一端为某个进程（这个进程可以是当前进程）拥有，即有一个（或多个）进程的文件描述符表中至少有一个成员指向管道的另一端（显然，能够读写管道当前端，则本端在当前进程中是可以访问的）。如果写入无名管道的数据超过其最大值，写操作将阻塞，如果管道中没有数据，读操作将阻塞，如果管道发现另一端断开（另一端文件描述符关闭），将自动退出。
 - (4) 有名管道阻塞问题。有名管道在打开时需要确实对方的存在，否则将阻塞。即以读方式打开某管道，该操作得以继续执行的条件是：在此之前，已经有一个进程以写的方式打开此管道，否则阻塞，直到条件满足，因此有名管道将阻塞在打开位置。也可以以读写（O_RDWR）方式打开有名管道，进程能够继续执行（不阻塞），只是这样操作没有什么意思，即当前进程读，当前进程写。

第8章 进程间通信－管道和信号

1

进程间通信－PIPE

2

进程间通信—FIFO

3

信号中断处理

Linux常见信号与处理

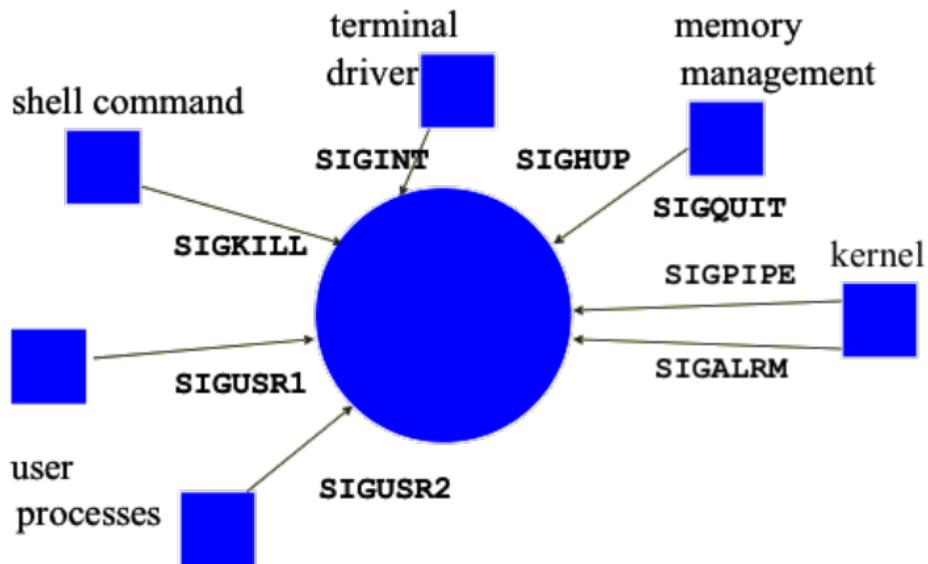
Linux 在/usr/include/asm/signum.h 中还详细定义了信号的信号值。内容如下：

```
//come from /usr/include/asm/signum.h
/* Signals. */
#define SIGHUP      1      /* Hangup (POSIX). */
#define SIGINT      2      /* Interrupt (ANSI). */
#define SIGQUIT     3      /* Quit (POSIX). */
#define SIGILL      4      /* Illegal instruction (ANSI). */
#define SIGTRAP     5      /* Trace trap (POSIX). */
#define SIGABRT     6      /* Abort (ANSI). */
#define SIGIOT      6      /* IOT trap (4.2 BSD). */
#define SIGBUS      7      /* BUS error (4.2 BSD). */
#define SIGFPE      8      /* Floating-point exception (ANSI). */
#define SIGKILL     9      /* Kill, unblockable (POSIX). */
```

信号的处理流程

- (1) 信号被某个进程产生，并设置此信号传递的对象（一般为对应进程的pid），然后传递给操作系统；
- (2) 操作系统根据接收进程的设置（是否阻塞）而选择性的发送给接收者，如果接收者阻塞该信号（且该信号是可以阻塞的），操作系统将暂时保留该信号，而不传递，直到该进程解除对此信号的阻塞（如果对应进程已经退出，则丢弃此信号）；如果对应进程没有阻塞，操作系统将传递此信号；
- (3) 目的进程接收到此信号后，将根据当前进程对此信号设置的预处理方式，暂时终止当前代码的执行，保护上下文（主要包括临时寄存器数据、当前程序位置以及当前CPU的状态）、转而执行中断服务程序，执行完成后再恢复到被中断的位置。当然，对于可抢占式内核，在中断返回时还将引发新的调度。

可能的信号来源



kill产生一个信号

kill()函数用来向指定进程发送一个信号。此函数声明如下：

```
//come from /usr/include/signal.h  
/* Send signal SIG to process number PID. If PID is zero, send SIG to all processes in the current process's  
process group.If PID is < -1, send SIG to all processes in process group - PID. */  
extern int kill (_pid_t _pid, int _sig)
```

此函数的第一个参数为要传递信号的进程号（PID），第 2 个参数即发送的信号值。pid 可以取以下几种值：

- pid>0：将信号发送给进程的 PID 值为 pid 的进程。
- pid=0：将信号发送给和当前进程在同一进程组的所有进程。
- pid=-1：将信号发送给系统内的所有进程。
- pid<0：将信号发送给进程组号 PGID 为 pid 绝对值的所有进程。

如果成功完成返回值 0，否则返回值-1，并设置 errno 以指示错误。

raise自举一个信号

传递信号给当前进程可以调用 raise() 函数。raise() 函数用来给当前进程发送一个信号，即唤醒一个进程。此函数声明如下：

```
//come from /usr/include/signal.h
/* Raise signal SIG, i.e., send SIG to yourself. */
extern int raise (int __sig)
```

此函数相当于采用以下方式执行 kill() 函数：

```
if(kill (getpid(), int __sig)==-1)
    perror("raise");
```

alarm()定时

alarm()函数用来传递定时信号，即在多少时间内产生 SIGALRM 信号，此函数每调用一次，产生一个信号，并不是循环产生 SIGALRM 信号。

```
//come from /usr/include/unistd.h
extern unsigned int alarm (unsigned int _seconds)
```

此函数只有一个参数，即在多少时间（秒为单位）内发送 SIGALRM 信号给当前进程，
默认情况下，当进程接受到 alarm 信号后将中止执行；如果 sec 为 0，则取消所有先前发出的
报警请求。

如果在调用 alarm() 函数前没有调用过 alarm() 函数，如果执行成功，将返回值 0，否则返
回 -1，并置 errno 标识错误。

ualarm定时

ualarm 将使当前进程在指定时间（第 1 个参数，以 us 为单位）内产生 SIGALRM 信号，然后每隔指定时间（第 2 个参数，以 us 为单位）重复产生 SIGALRM 信号。如果执行成功，将返回 0。该函数声明如下：

```
extern __useconds_t ualarm (__useconds_t __value, __useconds_t __interval);
```

信号处理与signal安装信号

- 信号处理办法
 - (1) 忽略此信号。大多数信号都可使用这种方式进行处理，但有两种信号不能被忽略，SIGKILL和SIGSTOP。这两种信号不能被忽略的原因是：它们向超级用户提供一种使进程终止或停止的可靠方法。
 - (2) 捕捉信号。通知内核在某种信号发生时调用一个用户函数。在用户函数中，可执行用户希望对这种事件进行的处理，这需要安装此信号。例如捕捉到SIGCHLD信号，则表示子进程已经终止，所以此信号的捕捉函数可以调用waitpid()以取得该子进程的进程PID以及它的终止状态和资源。
 - (3) 执行系统默认操作。Linux系统对任何一个信号都规定了一个默认的操作。

signal安装信号

```
typedef void (*__sighandler_t) (int);  
extern __sighandler_t signal (int __sig, __sighandler_t __handler)
```

此函数有两个参数，第一个参数 sig 为接收到的信号，第二个参数为接收到此信号后的处理代码入口（即处理子程序）或下面几个宏：

```
/* Fake signal functions. */  
  
#define SIG_ERR ((__sighandler_t)-1)      /* Error return. */      //返回错误  
#define SIG_DFL ((__sighandler_t)0)        /* Default action. */    //执行信号默认操作  
#define SIG_IGN ((__sighandler_t)1)        /* Ignore signal. */     //忽略信号
```

如果执行成功，此函数将返回针对此信号的上一次设置，如果设置多次，最终生效者为最近一次设置操作。如果执行失败，将返回 SIG_ERR 错误。

sigaction安装信号

```
extern int sigaction (int __sig, struct sigaction * __act, struct sigaction * __oact)
```

此函数的第一个参数为接收到的信号，第二、三个参数均为信号结构 `sigaction`（用于描述要采取的操作及相关信息，见后续说明）变量。第二个参数用来指定欲设置的信号处理信息，第 3 个参数将返回执行此程序前此信号处理信息。

如果第二个参数 `act` 不为空指针，则指定信号关联的操作为此参数指向的结构。如果参数 `oact` 不为空指针，则用来存储以前设置的与此信号关联的操作。

如果参数 `act` 为空指针，则信号处理保持不变；因此，该调用可用于询问对指定信号的当前处理。

struct sigaction

```
struct sigaction {  
    union {  
        __sighandler_t _sa_handler;           // SIG_DFL、SIG_IGN 信号，类似signal函数  
        void (*_sa_sigaction)(int, struct siginfo *, void *); //信号捕获函数，可以获取其它信息  
    } _u;  
  
    sigset_t sa_mask;                      //执行信号捕获函数期间要阻塞的其他信号集  
    unsigned long sa_flags;                //影响信号行为的特殊标志  
    void (*sa_restorer)(void);            //没有使用  
};  
  
#define sa_handler _u._sa_handler          //对两成员进行重定义  
#define sa_sigaction _u._sa_sigaction
```

信号集与屏蔽信号

- 中断是可以被屏蔽（阻塞）的（部分硬件中断是必须立即处理的，例如复位中断），因此，Linux的信号是可以屏蔽，即阻塞信号。但这与前面提到的忽略是有区别的。
- 信号忽略：系统仍然传递该信号，只是相应进程对该信号不作任何处理而已。
- 信号阻塞：系统不传递该信号，显示该进程无法接收到该信号直到进程的信号集发生改变。

sigprocmask设置进程阻塞的信号集

```
extern int sigprocmask (int __how, __const sigset_t *__restrict __set, sigset_t *__restrict __oset)
```

此函数第 1 个参数为更改该集的方式如下所示：

```
//come from /usr/include/asm/signal.h
```

```
#define SIG_BLOCK          0      /* for blocking signals */
```

```
#define SIG_UNBLOCK        1      /* for unblocking signals */
```

```
#define SIG_SETMASK        2      /* for setting the signal mask */
```

- SIG_BLOCK 将第 2 个参数所描述的集合添加到当前进程阻塞的信号集中。
- SIG_UNBLOCK 将第 2 个参数所描述的集合从当前进程阻塞的信号集中删除。
- SIG_SETMASK 无论之前的阻塞信号，设置当前进程阻塞的集合为第 2 个参数描述的对象。

如果 set 是空指针，则参数 how 的值没有意义，且不会更改进程的阻塞信号集，因此该调用可用于查询当前受阻塞的信号。

执行成功后，`sigprocmask()`返回 0；否则返回 -1，并设置 `errno` 以指明错误，同时进程的阻塞信号集将保持不变。

等待信号

pause()函数将使当前进程处于等待状态，直到当前进程阻塞信号外任意一个信号出现。其函数声明如下：

```
/* Suspend the process until a signal arrives. This always returns -1 and sets `errno' to EINTR. */
extern int pause (void);
```

此函数将挂起当前进程，直到接收到一个信号后才重新恢复执行。其始终返回-1 并设置 error 变量为 EINTR。

等待信号函数 sigsuspend()声明如下：

```
extern int sigsuspend (_const sigset_t * __set);
```

sigsuspend()函数将调用进程阻塞的信号集替换为其参数值，然后挂起该线程，直到传递一个非指定集合中信号为止。

信号应用示例 - 基本功能

- 创建了两个进程：
 - 父亲进程执行文件拷贝操作（为验证此程序，请选择大小在M级以上文件），如果接收到SIGUSR1信号，将打印出当前的拷贝进度，因此，父亲进程需要安装SIGUSR1信号；
 - 子进程每隔一个固定时间（其时间由ularm函数产生SIGALRM信号来决定）向父亲进程发送SIGUSR1信号。因此，子进程需要安装SIGALRM信号。
- 代码见教材。

第9章 System V进程间通信

- 1 System V IPC基础**
- 2 消息队列**
- 3 信号量通信机制**
- 4 共享内存**

ipcs

```
[root@localhost yangzongde]# ipcs
```

----- Shared Memory Segments ----- //共享内存

//key值 key	ID shmid	拥有者 owner	权限 perms	大小 bytes	挂接进程数 nattch	状态 status
---------------	-------------	--------------	-------------	-------------	-----------------	--------------

+

----- Semaphore Arrays ----- //信号量

//key值 key	ID semid	拥有者 owner	权限 perms	值 nsems
---------------	-------------	--------------	-------------	------------

+

----- Message Queues ----- //消息队列

//key值 key	ID msqid	拥有者 owner	权限 perms	大小 used-bytes	内容 messages
---------------	-------------	--------------	-------------	------------------	----------------

key值和ID值

- Linux系统为每个IPC机制都分配了唯一的ID，所有针对该IPC机制的操作都使用对应的ID。因此，通信的双方都需要通过某个办法来获取ID值。显然，创建者根据创建函数的返回值可获取该值，但另一个进程如何实现呢？显然，Linux两个进程不能随意访问对方的空间（一个特殊是，子进程可以继承父亲进程的数据，实现父亲进程向子进程的单向传递），也就不能够直接获取这一ID值。
- 为解决这一问题，IPC在实现时约定使用key值做为参数创建，如果在创建时使用相同的key值将得到同一个IPC对象的ID（即一方创建，另一方获取的是ID），这样就保证了双方可以获取用于传递数据的IPC机制ID值。

ftok

```
extern key_t ftok ( __const char * __pathname, int __proj_id);
```

此函数有两个参数， pathname 为文件路径名，可以是特殊文件（例如目录文件），也可以是当前目录“.”，而通常也是设置此参数为当前目录，因为当前目录一般都是存在的，且不会被立即删除。第二个参数为一个 int 型变量。

ftok()函数创建 key 值过程中使用了该文件属性的 st_dev 和 st_ino。具体构成如下：

- key 值的第 31~24（共 8bit）为 ftok()第 2 个参数的低 8 位；
- key 值的第 23~16（共 8bit）为该文件的 st_dev 属性的低 8 位；
- key 值的第 15~0（共 16bit）为该文件的 st_ino 属性的低 16 位；

因此，如果使用相同的文件路径及整数（第 2 个参数），得到的 key 值是唯一的，而唯一的 key 值创建某类 IPC 机制时将得到同一个 IPC 机制（但如果使用相同的 key 值分别创建一个消息队列和一个信号量，两者没有联系），而文件路径的访问对两个进程来说很容易统一，因此，便捷的实现了两进程通信机制的确定。

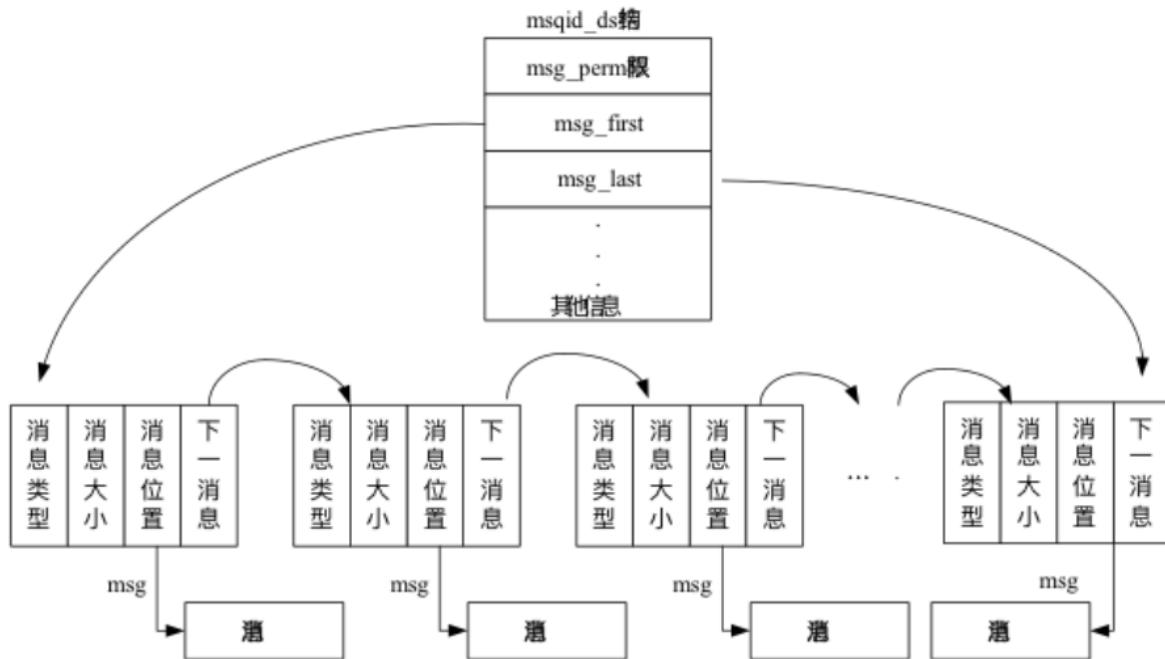
示例

```
1 #include<sys/ipc.h>
2 #include<stdio.h>
3 #include<sys/stat.h>
4 #include<stdlib.h>
5 int main(int argc,char *argv[]){
6     key_t key;          int i;          struct stat buf;
7     if(argc!=3)         {
8         printf("use: command path number\n");
9         return 1;
10    }
11    i=atoi(argv[2]);
12    if((stat(argv[1],&buf))==-1)        {
13        perror("stat");
14        exit(EXIT_FAILURE);
15    }
16    printf("file st_dev=%x\n",buf.st_dev);
17    printf("file st_ino=%x\n",buf.st_ino);
18    printf("number=%x\n",i);
19    key=ftok(argv[1],i);
20    printf("key=0x%x \tkey>>24=%x \tkey&0xffff=%x \t(key>>16)&0xff=%x\n",
21           key,key>>24,key&0xffff,(key>>16)&0xff);
22 }
```

第9章 System V进程间通信

- 1 System V IPC基础
- 2 消息队列
- 3 信号量通信机制
- 4 共享内存

消息队列



消息队列属性

```
struct msqid_ds {  
    struct ipc_perm msg_perm;           //权限  
    struct msg *msg_first;             /* first message on queue,unused*/ //指向消息头  
    struct msg *msg_last;              /* last message in queue,unused */ //指向消息尾  
    __kernel_time_t msg_stime;         /* last msgsnd time */ //最近发送消息时间  
    __kernel_time_t msg_rtime;         /* last msgrev time */ //最近接收消息时间  
    __kernel_time_t msg_ctime;         /* last change time */ //最近修改时间  
    unsigned long   msg_lbytes;        /* Reuse junk fields for 32 bit */  
    unsigned long   msg_lqbytes;        /* ditto */  
    unsigned short  msg_cbytes;        /* current number of bytes on queue */ //当前队列大小  
    unsigned short  msg_qnum;          /* number of messages in queue */ //当前队列消息个数  
    unsigned short  msg_qbytes;        /* max number of bytes on queue */ //队列最大值  
    __kernel_ipc_pid_t msg_lpid;       /* pid of last msgsnd */ //最近msgsnd的pid  
    __kernel_ipc_pid_t msg_rpid;       /* last receive pid */ //最近receive的pid  
};
```

消息struct msg结构体

```
//come from /usr/src/kernels/`uname -r`/include/linux/msg.h
67 /* one msg_msg structure for each message */
68 struct msg_msg {
69     struct list_head m_list;
70     long   m_type;           //消息类型
71     int m_ts;                /* message text size */ //消息大小
72     struct msg_msgseg* next; //下一个消息位置
73     void *security;          //真正消息位置
74     /* the actual message follows immediately */
75 };
```

消息队列属性控制

```
extern int msgctl(int __msqid, int __cmd, struct msqid_ds *__buf);
```

其包括三个参数：

第一个参数 `__msqid` 为消息队列标识符，该值为使用 `msgget` 函数创建消息队列的返回值。

第二个参数 `__cmd` 为执行的控制命令，即要执行的操作。包括以下选项：

```
//come from /usr/include/linux/ ipc.h  
/* Control commands used with semctl, msgctl and shmatl see also specific commands in sem.h, msg.h and  
shm.h */
```

#define IPC_RMID 0 /* remove resource */	//删除
#define IPC_SET 1 /* set ipc_perm options */	//设置ipc_perm参数
#define IPC_STAT 2 /* get ipc_perm options */	//获取ipc_perm参数
#define IPC_INFO 3 /* see ipcs */	//如ipcs

发送信息到消息队列

```
extern int msgsnd (int __msqid, __const void * __msgp, size_t __msgsz, int __msgflg);
```

此函数参数说明如下：

第一个参数 `msqid` 为指定的消息队列标识符（由 `msgget` 生成的消息队列标识符），即将消息添加到哪个消息队列中。

第二个参数 `msgp` 为指向的用户定义缓冲区，下面是用户定义缓冲区结构：

```
//come from /usr/include/linux/msg.h

/* message buffer for msgsnd and msgrev calls */

struct msghdr {
    long mtype;           /* type of message */          //消息类型
    char mtext[1];        /* message text */           //消息内容，在使用时自己重新定义此结构
};
```

- `mtype` 是一个正整数，由产生消息的进程生成，用于表示消息的类型，因此，接收进程可以用来进行消息选择（消息队列在存储信息时是按发送的先后顺序放置的）。
- `mtext` 是文本内容，即消息内容。此处大小为 1，显示不够用，在使用时自己重新定义此结构。

发送信息到消息队列 (2)

第三个参数为接收信息的大小，其数据类型为 `size_t`，即 `unsigned int` 类型。其大小为 0 到系统对消息队列的限制值。

第四个参数用来指定在达到系统为消息队列所定的界限（如达到字数限制）时应采取的操作。

- 如果设置为 `IPC_NOWAIT`，如果需要等待，则不发送消息并且调用进程立即返回错误信息 `EAGAIN`。
- 如果设置为 0，则调用进程挂起执行，直到达到系统所规定的最大值为止，并发送消息。

成功调用后，此函数将返回 0，否则返回 01，同时将对消息队列 `msqid` 数据结构的成员执行下列操作：

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

从消息队列接收信息

```
extern int msgrev(int __msqid, void * __msgp, size_t __msgsz, long int __msgtyp, int __msgflg);
```

此函数从与 `msqid` 指定的消息队列标识符相关联的队列中读取消息，并将其放置到由 `msgp` 指向的结构中。

第一个参数为读的对象，即从哪个消息队列获得消息。

第二个参数为一个临时消息数据结构，用来保存读取的信息。其定义如下：

```
//come from /usr/include/linux/msg.h  
/* message buffer for msgsnd and msgrev calls */  
  
struct msgbuf {  
    long mtype;           /* type of message */      //消息类型。  
    char mtext[1];        /* message text */       //存储消息位置，需要重新定义。  
};
```

`mtype` 是接收消息的类型（由发送进程指定）。`mtext` 为消息的文本。

第三个参数 `msgsz` 指定 `mtext` 的大小（以字节为单位）。如果收到的消息大于 `msgsz`，并且 `msgflg&MSG_NOERROR` 为真，则将该消息截至 `msgsz` 字节，消息的截断部分将丢失，并且不向调用进程提供截断的提示。

第四个参数 `msgtyp` 指定请求的消息类型：

- `msgtyp = 0` 收到队列中的第一条消息，任意类型。
- `msgtyp > 0` 收到第一条 `msgtyp` 类型的消息。
- `msgtyp < 0` 收到第一条最低类型（小于或等于 `msgtyp` 的绝对值）的消息。

第五个参数 `msgflg` 指定所需类型消息不在队列上时将要采取的操作。

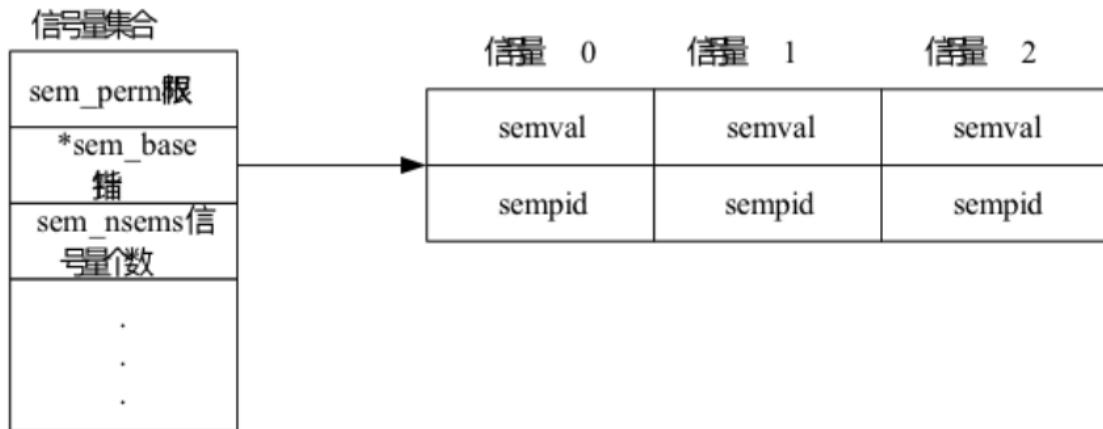
消息队列应用实例

- 见教材。

第9章 System V进程间通信

- 1 System V IPC基础
- 2 消息队列
- 3 信号量通信机制
- 4 共享内存

信号量通信机制概念图



信号量集合属性

```
struct semid_ds {  
    struct ipc_perm    sem_perm;          /* permissions .. see ipc.h */      //权限  
    __kernel_time_t   sem_otime;         /* last semop time */            //最近semop时间  
    __kernel_time_t   sem_ctime;         /* last change time */           //最近修改时间  
    struct sem     *sem_base;           /* ptr to first semaphore in array */ //队列第一个信号量  
    struct sem_queue *sem_pending;       /* pending operations to be processed */ //阻塞信号量  
    struct sem_queue **sem_pending_last; /* last pending operation */        //最后一个阻塞信号量  
    struct sem_undo  *undo;             /* undo requests on this array */    //undo队列  
    unsigned short    sem_nsems;         /* no. of semaphores in array */    //信号量数量  
};
```

信号量结构

```
come from /usr/src/kernels/^uname -r^/include/linux/sem.h^
/* One semaphore structure for each semaphore in the system. */^
struct sem {^
    int semval;      /* current value */                      //信号量的值^
    int sempid;     /* pid of last operation */                //最近一个操作的进程号PID^
};^
```

创建信号量集合

```
extern int semget (key_t __key, int __nsems, int __semflg);
```

第一个参数为 key_t 类型的 key 值，一般由 ftok 函数产生，此内容请参阅本章第 1 节。

第二个参数 __nsems 为创建的信号量个数，各信号量以数组的方式存储。这个数组用于初始化数组对象。

第三个参数 __semflg 用来标识信号量集合的权限。如 0770，为文件的访问权限类型。此外，还可以附加以下参数值。这些值可以与基本权限以或的方式一起使用。

```
//come from /usr/include/bit/ipc.h
/* resource get request flags */
#define IPC_CREAT    00001000      /* create if key is nonexistent */如果key不存在，创建
#define IPC_EXCL     00002000      /* fail if key exists */        如果key存在，返回失败
#define IPC_NOWAIT   00004000      /* return error on wait */       如果需要等待，直接返回错误
```

控制信号量集合、信号量

```
extern int semctl (int __semid, int __seminum, int __cmd, ...);
```

该函数最多可有四个参数（有可能只有三个参数）。第一个参数 `__semid` 为要操作的信号量集合标识符，该值一般由 `semget` 函数返回。

第二个参数为集合中信号量的编号。如果标识某个信号量，此值为该信号量的下标（从 0 到 `n-1`）；如果标识整个信号量集合，则设置为 0。

第三个参数为要执行的操作，如果是对整个信号量集合，这些操作在 `/usr/include/linux/ipc.h` 文件中定义。其操作包括 `IPC_RMID`、`IPC_SET`、`IPC_STAT` 和 `IPC_INFO`，具体含义同 `msgctl` 的相关操作。

```
//come from /usr/include/linux/ipc.h
/*
 * Control commands used with semctl, msgctl and shmctl see also specific commands in sem.h, msg.h and
shm.h */
#define IPC_RMID 0      /* remove resource */           //删除
#define IPC_SET  1      /* set ipc_perm options */        //设置ipc_perm参数
#define IPC_STAT 2      /* get ipc_perm options */       //获取ipc_perm参数
#define IPC_INFO 3      /* see ipcs */                  //获取系统信息
```

1 如果是对信号量集合中的某个或某些信号量操作，则包括：

2 #define GETPID 11 /* get sempid */ //获取信号量拥有者的pid值

3 如果使用此操作，第2个参数为0，第4个参数无效，如果执行成功，semctl将返回该进程pid值，否则返回-1。

4 #define GETVAL 12 /* get semval */ //获取信号量的值，函数返回信号量的值

5 如果使用此操作，第2个参数为信号量编号，如果执行成功，semctl将返回当前信号量的值，否则返回-1。

6 #define GETALL 13 /* get all semval's */ //获取所有信号量的值

7 如果使用此操作，第2个参数为0，第4个参数为存储所有信号量值内存空间首地址，如果执行成功，semctl将返回0，否则返回-1。

8 #define GETNCNT 14 /* get semncnt */ //获取等待信号量的值递增的进程数

9 如果使用此操作，第2个参数为0，如果执行成功，semctl将返回等待信号量的值递增的进程数，否则返回-1。

10 #define GETZCNT 15 /* get semzcnt */ //获取等待信号量的值递减的进程数

11 如果使用此操作，第2个参数为0，如果执行成功，semctl将返回等待信号量的值递减的进程数，否则返回-1。

12 #define SETVAL 16 /* set semval */ //设置信号量的值，设置的值在第4个参数中

13 如果使用此操作，第2个参数为为信号量编号，第4个参数为欲设置的值，如果执行成功，semctl将返回0，否则返回-1。

14 #define SETALL 17 /* set all semval's */ //设置所有信号量的值

15 如果使用此操作，第2个参数为0，第4个参数为欲设置的信号量值所在数组首地址，如果执行成功，semctl将返回0，否则返回-1。

信号量操作

```
extern int semop (int __semid, struct sembuf *__sops, size_t __nsops);
```

此函数第一个参数为要操作的信号量集合标识符，该值一般由 `semget` 函数返回。

第二个参数为 `struct sembuf` 结构的变量，其定义如下：

```
//come from /usr/include/linux/sem.h
/* semop system calls takes an array of these. */

struct sembuf {
    unsigned short  sem_num; /* semaphore index in array */      //信号量下标
    short          sem_op;   /* semaphore operation */        //信号量操作
    short          sem_flg; /* operation flags */           //操作标识
};
```

此结构体有三个成员变量。

- (1) `sem_num` 为操作的信号量编号。
- (2) `sem_op` 为作用于信号量的操作：该值如果为正整数表示增加信号量的值（如果为 1，表示在原来基础上加 1，如果为 3，表示在原来基础上加 3），如果为负整数表示减小信号量的值，如果为 0 表示对信号量的当前值进行是否为 0 的测试。
- (3) `sem_flg` 为操作标识

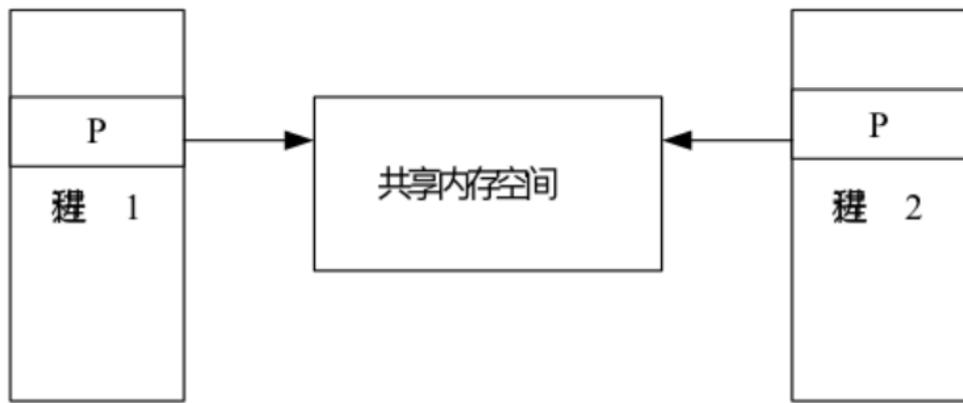
使用信号量实现生产消费问题

- 生产消费问题是一个经典的数学问题，要求生产者－消费者在固定的仓库空间条件下，生产者每生产一个产品将占用一个仓库空间，生产者生产的产品库存不能越过仓库的存储量，消费者每消费一个产品将增加一个仓库空间，消费者在仓库产品为0时不能再消费。
- 本例中采用信号量来解决这个问题，为了便于理解，本例中使用了两个信号量，一个用来管理消费者（以下为sem_produce），一个用来管理生产者（以下为sem_custom），即sem_produce表示当前仓库可用空间的数量，sem_custom用来表示当前仓库中产品的数量。
 - 对于生产者来说，其需要申请的资源为仓库中的剩余空间，因此，生产者在生产一个产品前，申请sem_produce信号量，当此信号量的值大于0，即有可用空间，将生产产品，并将sem_produce的值减去1（因为占用了一个空间）；同时，当其生产一个产品后，当前仓库的产品数量增加1，需要将sem_custom信号量自动加1。
 - 对于消费者来说，其需要申请的资源为仓库中的产品，因此，消费者在消费一个产品前，将申请sem_custom信号量，当此信号量的值大于0时，即有可用产品，将消费一个产品，并将sem_custom信号量的值减去（因为消费了一个产品），同时，当消费一个产品，当前仓库的剩余空间增加1，需要将sem_produce信号量自动加1。

第9章 System V进程间通信

- 1 System V IPC基础
- 2 消息队列
- 3 信号量通信机制
- 4 共享内存

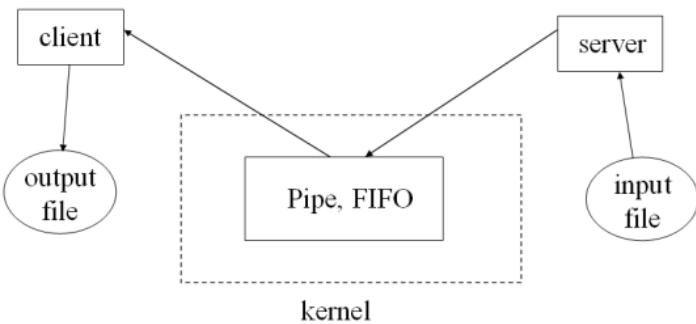
共享内存IPC原理



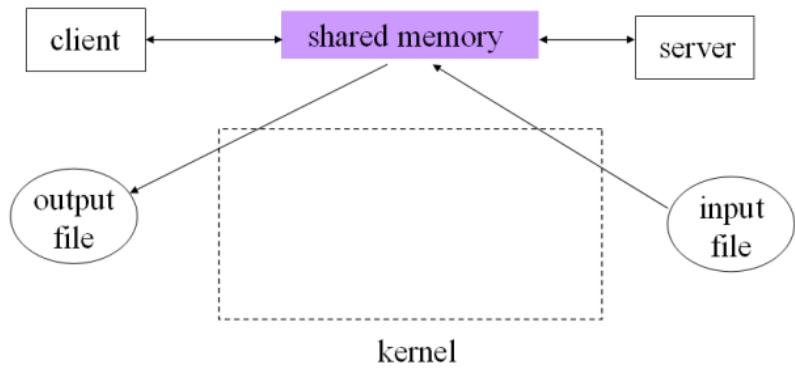
共享内存属性

```
struct shmid_ds {  
    struct ipc_perm     shm_perm;      /* operation perms */      //操作权限  
    int                shm_segsz;    /* size of segment (bytes) */ //段长度大小  
    __kernel_time_t    shm_atime;    /* last attach time */      //最近attach时间  
    __kernel_time_t    shm_dtime;    /* last detach time */      //最近detach时间  
    __kernel_time_t    shm_ctime;    /* last change time */      //最近change时间  
    __kernel_ipc_pid_t shm_cpid;     /* pid of creator */      //创建者pid  
    __kernel_ipc_pid_t shm_lpid;     /* pid of last operator */ //最近操作pid  
    unsigned short     shm_nattch;   /* no. of current attaches */ //当前连接数  
    unsigned short     shm_unused;   /* compatibility */  
    void               *shm_unused2;  /* ditto - used by DIPC */  
    void               *shm_unused3;  /* unused */  
};
```

共享内存与管道对比



kernel



kernel

创建共享内存

```
extern int shmget (key_t __key, size_t __size, int __shmflg);
```

其有 3 个参数。第一个参数为 `key_t` 类型的 `key` 值，一般由 `ftok` 函数产生，此内容请参阅本章第 1 节。

第二个参数 `size` 为欲创建的共享内存段大小（单位为字节）。

第三个参数 `shmflg` 用来标识共享内存段的创建标识。包括：

```
//come from /usr/include/linux/ ipc.h  
  
#define IPC_CREAT    01000      /* Create key if key does not exist. */      //如果不存在就创建。  
#define IPC_EXCL     02000      /* Fail if key exists. */                      //如果存在则返回失败。  
#define IPC_NOWAIT   04000      /* Return error on wait. */                     //不等待直接返回。
```

另外，在`/usr/include/linux/shm.h` 文件中还定义了另外两个选项：

```
//come from /usr/include/linux/shm.h  
  
/* permission flag for shmget */  
  
#define SHM_R        0400      /* or S_IRUGO from <linux/stat.h> */      //可读。  
#define SHM_W        0200      /* or S_IWUGO from <linux/stat.h> */      //可写。
```

共享内存控制

```
extern int shmatl (int __shmid, int __cmd, struct shmid_ds *__buf);
```

第一个参数为要操作的共享内存标识符，该值一般由 `shmget` 函数返回。

第二个参数为要执行的操作，这些操作在 `/usr/include/linux/ipc.h` 文件中定义。其操作包括 `IPC_RMID`、`IPC_SET`、`IPC_STAT` 和 `IPC_INFO`，具体含义同 `msgctl` 的相关操作。

```
//come from /usr/include/linux/ipc.h

/* Control commands used with semctl, msgctl and shmatl see also specific commands in sem.h, msg.h and
shm.h */

#define IPC_RMID 0      /* remove resource */           //删除
#define IPC_SET  1      /* set ipc_perm options */        //设置ipc_perm参数
#define IPC_STAT 2      /* get ipc_perm options */       //获取ipc_perm参数
#define IPC_INFO 3      /* see ipcs */                  //如ipcs
```

如果是超级用户，还可以执行以下两个命令：

```
// come from /usr/include/sys/shm.h

/* super user shmatl commands */

#define SHM_LOCK     11          //锁定共享内存段
#define SHM_UNLOCK   12          //解锁共享内存段
```

第三个参数为 `struct shmid_ds` 结构的临时共享内存变量信息，此内容根据第二个参数的不一样而改变。

共享内存应用示例

- 共享内存的权限管理示例
- 共享内存处理应用示例
- 代码见教材。

第10章 Linux多线程编程

1

线程基本概念与线程操作

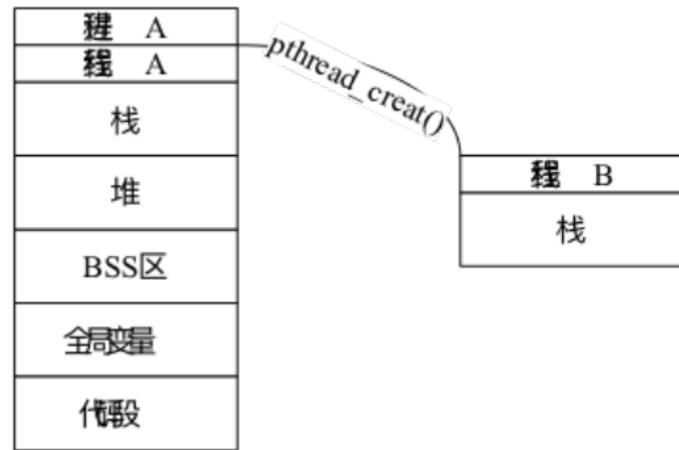
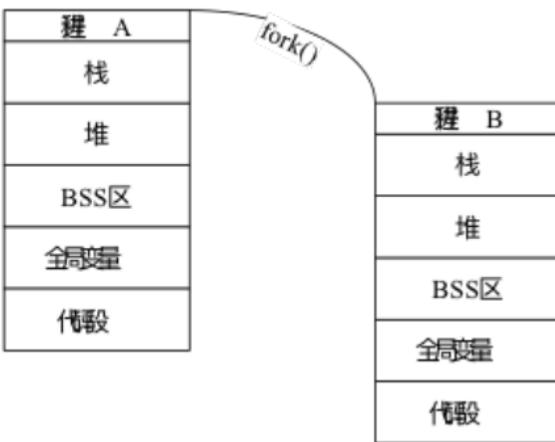
2

线程属性控制

3

线程调度策略

线程与进程的对比



线程资源

- 线程自己基本上不拥有系统资源，只拥有少量在运行中必不可少的资源（如程序计数器、一组寄存器、栈、线程信号掩码、局部线程变量和线程私有数据），但是它可与同属一个进程的其他线程共享进程所拥有的全部资源（同一地址空间、通用的信号处理机制、数据与I/O）。
- 进程在使用时占用了大量的内存空间，特别是进行进程间通信时一定要借助操作系统提供的通信机制，这使得进程有自身的弱点，而线程占用资源少，使用灵活，很多应用程序中都大量使用线程，而较少的使用多进程，但是，线程不能脱离进程而存在，另外，线程的层次关系，执行顺序并不明显，对于初学者大量使用线程会增加程序的复杂度。

进程/线程应用对比

应用功能	线程	进程
创建	pthread_create	fork,vfork
退出	pthread_exit	exit
等待	pthread_join	wait、 waitpid
取消/终止	pthread_cancel	abort
读取ID	pthread_self()	getpid()
调度策略	SCHED_OTHER、 SCHED_FIFO、 SCHE D_RR	SCHED_OTHER、 SCHED_FIFO、 SC HED_RR
通信机制	信号量、 信号、 互斥锁、 条件变量、 读写锁	无名管道、 有名管道、 信号、 消息队列、 信号量、 共享内存

创建线程

```
1 extern int pthread_create (pthread_t * __restrict __newthread,
2                           __const pthread_attr_t * __restrict __attr,
3                           void * (*__start_routine) (void *),
4                           void * __restrict __arg)
```

此函数第一个参数用来存储线程ID，参数为指向线程ID的指针，在目前版本的Linux下²，线程的ID在某个进程中唯一，也就是说，如果在父子进程中创建多个线程，线程ID值有可能相同。如果创建成功，在此参数中返回ID，如果设置为NULL，则不会返回生成的线程的标识值。pthread_t类型定义如下：

```
6 //come from /usr/include/bits/pthreadtypes.h
```

```
7 typedef unsigned long int pthread_t;
```



//无符号长整型变量，打印id时需要使用%u打印

8 第二个参数用来设置线程属性，主要设置与栈相关的属性，本章下一小节将详细介绍这

一内容。一般情况下，此参数设置为NULL，新的线程将使用系统默认的属性。

9 第三个参数是线程运行的代码起始地址，即在此线程中运行哪段代码。

10 第四个参数是运行函数的参数地址。如果需要给自定义的函数传入多个参数，则需要使用一个包含这些参数的结构体地址。

11 如果线程创建成功，它将拥有自己的线程属性和执行栈，并从调用程序那里继承信号掩码和调试优先级。

12 此函数如果执行成功，将返回0，如果失败，将返回非0值。

线程退出与等待

- 新创建的线程从执行用户定义的函数处开始执行，直到出现以下情况时退出：
- 调用pthread_exit函数退出。
- 调用pthread_cancel函数取消该线程。
- 创建线程的进程退出或者整个函数结束。
- 其中的一个线程执行了exec类函数执行新的进程。

Pthread 库线程退出函数声明如下：

```
//come from /usr/include/bits/pthreadtypes.h  
extern void pthread_exit (void * __retval)
```

使用 pthread_exit 库函数调用可以结束一个线程，其结束方式与进程调用 exit() 函数类似。此函数只有一个参数，它用来保存线程退出状态。

等待线程

```
extern int pthread_join(pthread_t __th, void **__thread_return);
```

此函数将阻塞调用当前线程的进程，直到此线程退出。即这个函数是一个线程阻塞的函数，调用它的进程将一直等待到被等待的线程结束为止，当函数返回时，处于被等待状态的线程资源被收回。如果执行成功，将返回 0，如果失败则返回一个非 0。

第一个参数为被等待的线程 ID，此线程必须同调用它的进程相联系，即创建该线程时不能指明此线程为独立的线程，默认情况下为关联线程。

第二个参数为一个用户定义的指针，指向一个保存等待线程的完整退出状态的静态区域，它可以用来存储被等待线程的返回值，如果在创建线程时 `pthread_create` 函数的第二个参数 `_attr` 设置为 `NULL`，则此退出的状态信息会丢失。

取消线程

- 取消线程是指取消一个正在执行线程的操作，当然，一个线程能够被取消并终止执行需要满足以下条件：
 - 该线程是否可以被其它取消，这是可以设置的，在Linux系统下，默认是可以被取消的，可用宏分配是PTHREAD_CANCEL_DISABLE和PTHREAD_CANCEL_ENABLE；
 - 该线程处于可取消点才能取消。也就是说，该线程被设置为可以取消状态，另一个线程发起取消操作，该线程并不是一定马上终止，只能在可取消点才中止执行。可以设置为立即取消和在取消点取消。可用宏为PTHREAD_CANCEL_DEFERRED和PTHREAD_CANCEL_ASYNCHRONOUS。

pthread_cancel()

```
extern int pthread_cancel (pthread_t __cancelthread);
```

pthread_cancel()函数请求取消执行线程。它允许线程以受控方式终止进程中执行的任何线程。目标线程的可取消性状态和类型将决定取消何时生效。仅当目标线程的可取消性状态为 PTHREAD_CANCEL_ENABLE 时，才可进行取消。

执行取消操作时，将调用线程的取消清理处理程序（pthread_cleanup_push 函数）。调用取消清理处理程序的顺序与安装这些处理程序的顺序相反。

pthread_cancel()的调用者将不会等待取消目标线程操作完成才返回，只是发送该信号，如果执行成功，将返回零。否则返回错误编号，但并不设置 errno 变量。

设置可取消状态

`pthread_setcancelstate()`和`pthread_setcanceltype()`可用来设置和查询当前线程的可取消性状态或类型。`pthread_setcancelstate()`函数声明如下：

```
extern int pthread_setcancelstate (int __state, int * __oldstate);
```

此函数有两个参数，`state`为调用线程的可取消性状态所要设置的值；`oldstate`为存储调用线程原来的可取消性状态的内存地址。

- 可设置的`state`的合法值：
 - 如果目标线程的可取消性状态为`PTHREAD_CANCEL_DISABLE`，则针对目标线程的取消请求将处于未决状态，启用取消后才执行取消请求。
 - 如果目标线程的可取消性状态为`PTHREAD_CANCEL_ENABLE`，则针对目标线程的取消请求将被传递。默认情况下，在创建某个线程时，其可取消性状态设置为`PTHREAD_CANCEL_ENABLE`。

设置取消类型

- `pthread_setcanceltype()`函数用来设置取消类型，即允许取消的线程在接收到取消操作后是立即中止还是在取消点中止，该函数声明如下：
`extern int pthread_setcanceltype (int __type, int * __oldtype)`
- 此函数有两个参数，`type`为调用线程的可取消性类型所要设置的值。`oldtype`为存储调用线程原来的可取消性类型的地址。`type`的合法值包括：
 - 如果目标线程的可取消性状态为`PTHREAD_CANCEL_ASYNCHRONOUS`，则可随时执行新的或未决的取消请求。
 - 如果目标线程的可取消性状态为`PTHREAD_CANCEL_DEFERRED`，则在目标线程到达一个取消点之前，取消请求将一直处于未决状态。
- 在创建某个线程时，其可取消性类型设置为`PTHREAD_CANCEL_DEFERRED`。

第10章 Linux多线程编程

1

线程基本概念与线程操作

2

线程属性控制

3

线程调度策略

线程资源

- 线程只拥有少量在运行中必不可少的资源，主要包括：
- 程序计数器：标识当前线程执行的位置；
- 一组寄存器：当前线程执行的上下文内容；
- 栈：用于实现函数调用、局部变量。因此，局部变量是私有的；
- 线程信号掩码：因此可以设置每线程阻塞的信号，见本书下一章内容；
- 局部线程变量：在栈中申请的数据；
- 线程私有数据。见前一小节介绍。

线程属性结构体

```
/* Attributes for threads. */  
typedef struct __pthread_attr_s {  
    int __detachstate;           //是否可以被等待  
    int __schedpolicy;          //调度策略  
  
    struct __sched_param __schedparam; //某调用策略的参数  
    int __inheritsched;          //是否继承创建线程的调度策略  
    int __scope;                 //争用范围  
    size_t __guardsize;          //栈保护区大小  
    int __stackaddr_set;         //  
    void *__stackaddr;           //栈起始地址  
    size_t __stacksize;          //栈大小  
} pthread_attr_t;
```

线程ID

线程最重要的属性为线程的 ID 值，函数 `pthread_self()` 将返回当前线程的 ID 值，该函数声明如下：

```
/* Obtain the identifier of the current thread. */  
extern pthread_t pthread_self(void);
```

在当前 Linux 下，线程 ID 是在某进程中惟一，如下所示，在不同的进程中创建的线程可能出现 ID 值相同的情况，这些决定了争用范围只能是线程内竞争。

初始化线程属性对象

- extern int pthread_attr_init (pthread_attr_t *__attr)

属性	缺省值	描述
scope	PTHREAD_SCOPE_PROCESS	新线程与进程中的其他线程发生竞争
detachstate	PTHREAD_CREATE_JOINABLE	线程可以被其它线程等待
stackaddr	NULL	新线程具有系统分配的栈地址
stacksize	0	新线程具有系统定义的栈大小
priority	0	新线程的优先级为0
inheritsched	PTHREAD_EXPLICIT_SCHED	新线程不继承父线程调度优先级
schedpolicy	SCHED_OTHER	新线程使用优先级调度策略

获取/设置线程detachstate属性

```
extern int pthread_attr_setdetachstate (pthread_attr_t * __attr, int __detachstate) ;
```

`pthread_attr_setdetachstate()`用于设置已初始化属性对象 `attr` 中的 `detachstate` 属性。
`detachstate` 属性的新值将传递给 `detachstate` 参数中的此函数，其合法值包括：

- `PTHREAD_CREATE_DETACHED` 此选项使得使用 `attr` 创建的所有线程处于分离状态。线程终止时，系统将自动回收与带有此状态的线程相关联的资源。这类线程不能被其它线程等待。
- `PTHREAD_CREATE_JOINABLE` 此选项使得使用 `attr` 创建的所有线程处于可连接状态。线程终止时，不会回收与带有此状态的线程相关联的资源。要回收系统资源，应用程序必须在其它线程调用 `pthread_detach()` 或 `pthread_join()` 函数。

`detachstate` 的缺省值是 `PTHREAD_CREATE_JOINABLE`。

```
extern int pthread_attr_getdetachstate ( __const pthread_attr_t * __attr, int * __detachstate) ;
```

`pthread_attr_getdetachstate()`可以从线程属性对象 `attr` 中检索 `detachstate` 属性值，并在 `detachstate` 参数中返回此值。

获取/设置线程栈相关属性

```
extern int pthread_attr_setstacksize (pthread_attr_t * __attr, size_t __stacksize)
```

此函数第 1 个参数为线程属性，第 2 个参数 stacksize 定义使用此属性对象创建的线程的用户堆栈大小（以字节为单位）。stacksize 属性的合法值包括：

- PTHREAD_STACK_MIN 此选项指定，使用此属性对象创建的线程的用户栈大小将使用缺省堆栈大小。此值为某个线程所需的最小堆栈大小。但对于所有线程来说，

这个最小值可能无法接受。

- 具体的大小值。定义使用线程的用户堆栈大小的数值。必须大于或等于最小堆栈大小 PTHREAD_STACK_MIN。

函数 `pthread_attr_getstacksize()` 可以从线程属性对象 `attr` 中获取 `stacksize` 属性并在 `stacksize` 参数中返回此值。其函数声明如下：

```
/*Return the currently used minimal stack size. */
```

```
extern int pthread_attr_getstacksize (_const pthread_attr_t * __restrict __attr, size_t * __restrict __stacksize)
```

获取/设置stack地址属性

pthread_attr_setstackaddr()用于设置已初始化属性对象 attr 中的栈基址属性。函数声明如下：

```
/* Set the starting address of the stack of the thread to be created. Depending on whether the stack grows up or down the value must either be higher or lower than all the address in the memory block. The minimal size of the block must be PTHREAD_STACK_MIN. */  
extern int pthread_attr_setstackaddr (pthread_attr_t * __attr, void * __stackaddr) ;
```

pthread_attr_getstackaddr()可以从线程属性对象 attr 中检索 stackaddr 属性并在 stackaddr 参数中返回此值。函数声明如下：

```
/* Return the previously set address for the stack. */  
extern int pthread_attr_getstackaddr ( __const pthread_attr_t * __restrict __attr, void ** __restrict __stackaddr ) ;
```

获取/设置栈保护区属性

```
extern int pthread_attr_setguardsize (pthread_attr_t * __attr, size_t __guardsize)
```

为应用程序设置栈保护区属性可以考虑以下因素：

- 溢出保护可能会导致系统资源浪费。如果应用程序创建大量线程，并且已知这些线程永远不会溢出其栈，则可以关闭溢出保护区。通过关闭溢出保护区，可以节省系统资源。
- 线程在栈上分配大型数据结构时，可能需要较大的溢出保护区来检测栈溢出。

guardsize 的缺省值为 PAGESIZE 字节。PAGESIZE 的实际值与实现相关，并且不可以在所有实现上使用相同值。如果用户堆栈的存储不是由 pthread 库分配的，将忽略 guardsize 属性。应用程序负责防止堆栈溢出。

pthread_attr_getguardsize() 可以从线程属性对象 attr 中读取 guardsize 属性值，并在 guardsize 参数中返回此值。如果守护区向上舍入为 PAGESIZE 的倍数，则此函数的调用必须

将以前的 pthread_attr_setguardsize() 函数调用指定的守护区大小存储在 guardsize 参数中。其函数声明如下：

```
/* Get the size of the guard area created for stack overflow protection. */
```

```
extern int pthread_attr_getguardsize (__const pthread_attr_t * __attr, size_t * __guardsize)
```

第10章 Linux多线程编程

- 1 线程基本概念与线程操作
- 2 线程属性控制
- 3 线程调度策略

调度策略

- 在操作系统中，调度策略主要有：
 - FIFO先入先出原则：首先请求服务的对象首先得到CPU的处理。
 - 最短作业优先原则：需要最小系统时间的服务首先得到处理。
 - 最高优先级优先原则：优先级最高的服务首先得到处理。
 - 时间轮片原则：每个任务分配一个系统时间片，轮流执行。
- POSIX为线程指定了三个调度策略：
 - SCHED_OTHER：系统默认策略。如时间轮片策略或基于优先级的调度策略。
 - SCHED_FIFO：先进先出原则。具有最高优先级的、等待时间最长的线程将成为下一个要执行的线程。
 - SCHED_RR：轮转调度。类似于FIFO，但加上了时间轮片策略。

获取/设置线程属性调度属性

```
extern int pthread_attr_setinheritsched (pthread_attr_t * __attr, int __inherit) ;
```

inheritsched 的合法值包括：

- PTHREAD_INHERIT_SCHED 此选项指定，从创建线程中继承调度策略及关联属性。如果使用 attr 创建了线程，将忽略 attr 参数中的调度策略和关联属性。
- PTHREAD_EXPLICIT_SCHED 此选项指定，从此属性对象中获得已创建线程的调度策略及关联属性。

pthread_attr_getinheritsched()可以从线程属性对象 attr 中获取调度策略继承属性并在 inherit 参数中返回此值。其函数声明如下：

```
/* Return in *INHERIT the scheduling inheritance mode of *ATTR. */
```

```
extern int pthread_attr_getinheritsched (__const pthread_attr_t * __restrict __attr, int * __restrict __inherit) ;
```

获取/设置调度策略属性

```
/* Set scheduling policy in *ATTR according to POLICY. */  
extern int pthread_attr_setschedpolicy(pthread_attr_t *attr, int __policy);
```

此函数第 1 个参数为线程属性。

第 2 个参数 policy 用来设置调度策略。包括以下 3 个选项：

```
//come from /usr/include/bits/sched.h  
/* Scheduling algorithms. */  
  
#define SCHED_OTHER    0          //默认策略  
#define SCHED_FIFO    1          //先入先出原则  
#define SCHED_RR     2          //时间轮转
```

pthread_attr_getschedpolicy()可以从线程属性对象 attr 中读取调度策略属性并在 policy 参数中返回此值。函数声明如下：

```
/* Return in *POLICY the scheduling policy of *ATTR. */  
extern int pthread_attr_getschedpolicy(__const pthread_attr_t *__restrict __attr, int *__restrict __policy);
```

获取/设置调度策略参数属性

pthread_attr_setschedparam()用于设置已初始化属性对象 attr 中的调度策略参数属性。 schedparam 属性的新值将传递给 param 参数中的此函数。函数声明如下：

```
/* Set scheduling parameters (priority, etc) in *ATTR according to PARAM. */  
extern int pthread_attr_setschedparam (pthread_attr_t * __restrict __attr,  
                                     __const struct sched_param * __restrict __param)
```

此函数第 1 个参数为线程属性。

第 2 个参数是对结构体 sched_param 的一个引用，即用来作为调度策略的补充参数。其声明如下：

```
//come from /usr/include/bits/sched.h  
/* The official definition. */  
struct sched_param {  
    int __sched_priority;           //优先级值  
};
```

函数 pthread_attr_getschedparam()可以从线程属性对象 attr 中获取调度策略参数属性并在 param 参数中返回此值。函数声明如下：

```
/* Return in *PARAM the scheduling parameters of *ATTR. */
```

```
extern int pthread_attr_getschedparam (__const pthread_attr_t * __restrict __attr,  
                                      __struct sched_param * __restrict __param)
```


第11章 线程间同步机制

- 1 互斥锁通信机制
- 2 条件变量通信机制
- 3 读写锁通信机制
- 4 线程与信号

互斥锁基本原理

- 互斥以排他方式防止共享数据被并发修改。互斥锁是一个二元变量，其状态为开锁（允许0）和上锁（禁止1），将某个共享资源与某个特定互斥锁绑定后，对该共享资源的访问如下操作：
 - (1) 在访问该资源前，首先申请该互斥锁，如果该互斥处于开锁状态，则申请到该锁对象，并立即占有该锁（使该锁处于锁定状态），以防止其它线程访问该资源；如果该互斥锁处于锁定状态，默认阻塞等待；
 - (2) 只有锁定该互斥锁的进程才能释放该互斥锁。其它线程的释放操作无效。

互斥锁基本操作函数

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

pthread_mutex_init

```
extern int pthread_mutex_init(pthread_mutex_t *__mutex, __const pthread_mutexattr_t *__mutexattr)
```

第一个参数 mutex 是指向要初始化的互斥锁的指针。

第二个参数 mutexattr 是指向属性对象的指针，该属性对象定义要初始化的互斥锁的属性。如果该指针为 NULL，则使用缺省的属性（关于属性操作见后）。

可以使用宏 PTHREAD_MUTEX_INITIALIZER 初始化静态分配的互斥锁。此宏定义如下：

```
//come from /usr/include/pthread.h
#define PTHREAD_MUTEX_INITIALIZER {{ 0, } }
```

对于静态初始化的互斥锁，不需要调用 pthread_mutex_init() 函数。

使用 PTHREAD_MUTEX_INITIALIZER 初始化互斥锁的代码段如下：

```
pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;
```

使用默认属性初始化互斥锁的代码段如下：

```
pthread_mutexattr_t mattr;
int ret;
ret = pthread_mutex_init(&mp, NULL);
```

使用自定义属性初始化互斥锁的代码段如下：

```
ret = pthread_mutex_init(&mp, &mattr);
```

申请互斥锁

如果一个线程要占用一共享资源，其必须先给互斥锁上锁。pthread_mutex_lock()函数以阻塞方式申请互斥锁。pthread_mutex_lock()函数声明如下：

```
extern int pthread_mutex_lock (pthread_mutex_t * __mutex)
```

pthread_mutex_trylock()函数以非阻塞方式申请互斥锁，函数声明如下：

```
/* Try locking a mutex. */
```

```
extern int pthread_mutex_trylock (pthread_mutex_t * __mutex)
```

pthread_mutex_lock()和pthread_mutex_trylock()如果成功完成，将返回0。否则，返回一个错误编号，以指明错误。

释放互斥锁

pthread_mutex_unlock()函数用来释放互斥锁。其函数声明如下：

```
/* Unlock a mutex. */  
extern int pthread_mutex_unlock (pthread_mutex_t * __mutex)
```

其参数 mutex 为指向要解锁的互斥锁的指针。释放操作只能由占有该互斥锁的线程完成。如果成功完成, pthread_mutex_unlock()返回 0。否则, 返回指明错误的错误编号(未设置 errno 变量)。

第11章 线程间同步机制

- 1 互斥锁通信机制
- 2 条件变量通信机制
- 3 读写锁通信机制
- 4 线程与信号

条件变量基本原理 互斥锁不能解决的问题

```
pthread_mutex_t work_mutex; +  
int i=3; +  
int j=7; +  
+  
thread_A  
pthread_lock()  
{  
    i++;  
    j++;  
}  
+  
pthread_unlock()  
  
thread_B  
pthread_lock()  
{  
    if(i==j)  
        do_something();  
}  
+  
pthread_unlock()
```

互斥锁不能解决的问题

- 如果只使用互斥锁，可能导致`do_something()`永远不会执行，这是程序员所不期望的，如下分析所示：
 - 线程A抢占到互斥锁，执行操作，完成后 $i==4, j=6$ ；然后释放互斥锁；
 - 线程A和线程B都有可能抢占到锁，如果B抢占到，条件不满足，退出；如果线程A抢占到，则执行操作，完成后 $i==5, j=5$ ；然后释放互斥锁；
 - 同理，线程A和线程B都有可能抢占到锁，如果B抢占到，则条件满足，`do_something()`得以执行，得到预期结果。但如果此时A没有抢占到，执行操作后 $i=6, j=4$ ，此后 i 等于 j 的情况永远不会发生。

条件变量解决的问题

```
pthread_mutex_t work_mutex; //  
pthread_cond_t condition; //增加条件变量  
int i=3;  
int j=7;  
  
thread_A  
pthread_lock()  
{  
    i++;  
    j--;  
  
    if(i==j)  
        释放锁，通知等待条件变量；  
  
    else  
        do_something();  
}  
pthread_unlock()  
  
thread_B  
pthread_lock()  
{  
    if(i==j)  
        释放锁，通知等待条件变量的线程；  
    else  
        do_something();  
}  
pthread_unlock()
```

条件变量基本操作

功能	函数
初始化条件变量	pthread_cond_init
阻塞等待条件变量	pthread_cond_wait
通知等待该条件变量的第1个线程	pthread_cond_signal
在指定的时间之内阻塞等待条件变量	pthread_cond_timedwait
通知等待该条件变量的所有线程	pthread_cond_broadcast
销毁条件变量状态	pthread_cond_destroy

条件变量应用

- 见书上例程。

第11章 线程间同步机制

- 1 互斥锁通信机制
- 2 条件变量通信机制
- 3 读写锁通信机制
- 4 线程与信号

读写锁通信机制

- 在对数据的读写应用中，更多的是读操作，而写操作较少，例如对数据库数据的读写应用。为了满足当前能够允许多个读出，但只允许一个写入的需求，线程提供了读写锁来实现。其基本原则如下：
- (1) 如果有其它线程读数据，则允许其它线程执行读操作，但不允许写操作；
- (2) 如果有其它线程写数据，则其它线程的读、写操作均允许。
- 因此，其将该锁分为了读锁和写锁。
- (1) 如果某线程申请了读锁，其它线程可以再申请读锁，但不能申请写锁；
- (2) 如果某线程申请了写锁，则其它线程不能申请读锁，也不能申请写锁。
- 定义读写锁对象的代码如下：

```
pthread_rwlock_t rwlock; //全局变量
```

读写锁基本操作

功能	函数
初始化读写锁	pthread_rwlock_init
阻塞申请读锁	pthread_rwlock_rdlock
非阻塞申请读锁	pthread_rwlock_tryrdlock
阻塞申请写锁	pthread_rwlock_wrlock
非阻塞申请写锁	pthread_rwlock_trywrlock
释放锁（无论是读锁还是写锁）	pthread_rwlock_unlock
销毁读写锁	pthread_rwlock_destroy

第11章 线程间同步机制

- 1 互斥锁通信机制**
- 2 条件变量通信机制**
- 3 读写锁通信机制**
- 4 线程与信号**

线程在信号操作时有以下特性

- (1) 每一个线程可以向别的线程发送信号。pthread_kill()函数用来完成这一操作。
- (2) 每一个线程可以设置自己的信号阻塞集合。pthread_sigmask()函数用来完成这一操作，其类似于进程的sigprocmask()函数。
- (3) 每个线程可以设置针对某信号处理的方式，但同一进程中对某信号的处理方式只能有一个有效，即最后一次设置的处理方式。
- (4) 如果别的进程向当前进程中发送一个信号，由哪个线程处理是未知的。

线程信号管理

pthread_kill 函数用来在线程间发送信号，其声明如下：

```
//come from /usr/include/bits/sigthread.h  
/* Send signal SIGNO to the given thread. */  
extern int pthread_kill(pthread_t __threadid, int __signo)
```

其有两个参数，`threadid` 是要向其传送信号的线程。`signo` 是要传送给线程的信号。
`pthread_kill()` 函数用于请求将信号传送给线程。调用进程中，信号将被异步定向到线程。

如果 `signo` 为 0，就会进行错误检查而不发送信号。成功完成后，`pthread_kill()` 将返回 0。
否则就会返回一个错误编号，用于指明错误（未设置 `errno` 变量）。

pthread_sigmask调用线程的信号掩码

```
//come from /usr/include/bits/sigthread.h
extern int pthread_sigmask (int __how, __const __sigset_t * __restrict __newmask,+
                           __sigset_t * __restrict __oldmask);
```

第一个参数 how 定义如何更改调用线程的信号掩码。合法值包括：

```
#define SIG_BLOCK          0 /* for blocking signals */
#define SIG_UNBLOCK         1 /* for unblocking signals */
#define SIG_SETMASK         2 /* for setting the signal mask */
```

SIG_BLOCK 将第 2 个参数所描述的集合添加到当前进程阻塞的信号集中。

SIG_UNBLOCK 将第 2 个参数所描述的集合从当前进程阻塞的信号集中删除。

SIG_SETMASK 无论之前的阻塞信号，设置当前进程阻塞的集合为第 2 个参数描述的对象。

如果 set 是空指针，则参数 how 的值没有意义，且不会更改线程的阻塞信号集，因此该调用可用于查询当前受阻塞的信号。

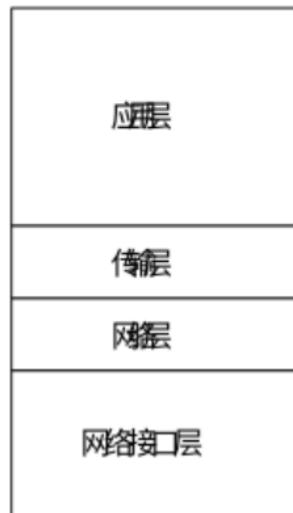
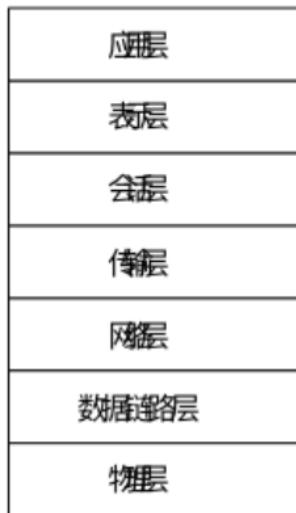
线程信号应用实例

- 见教材示例。

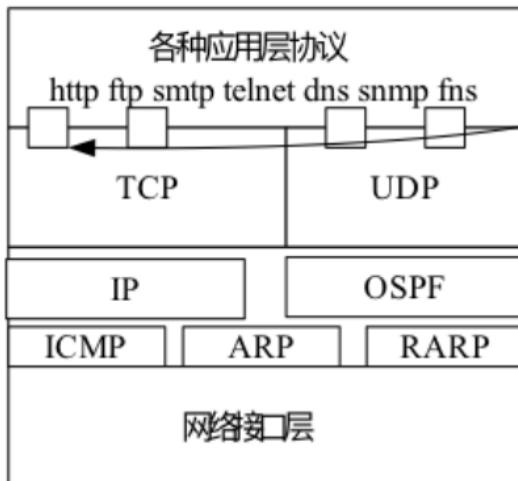
第12章 Linux socket网络编程基础

- 1 网络通信基础
- 2 BSD Socket TCP网络通信编程
- 3 BSD Socket UDP网络通信编程
- 4 使用TCP实现简单聊天程序

OSI模型及TCP/IP协议模式



TCP/IP体系结构及各层协议

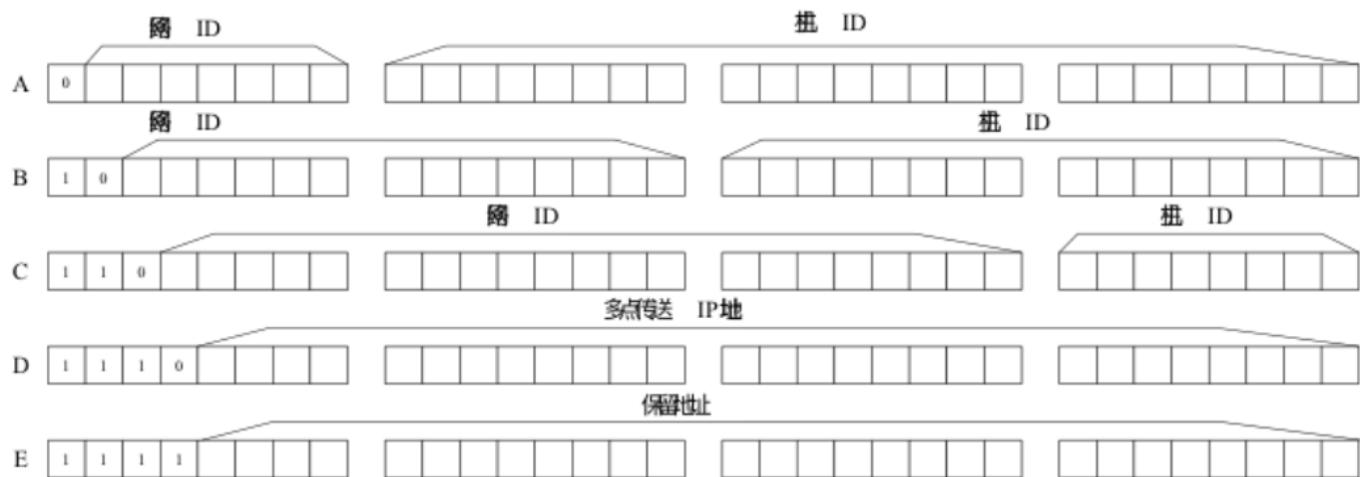


端：
在主机内唯一 标识应用程序

IP地址：
逻辑上唯一的 节点地址

MAC地址：
物理上唯一的标识一台主机

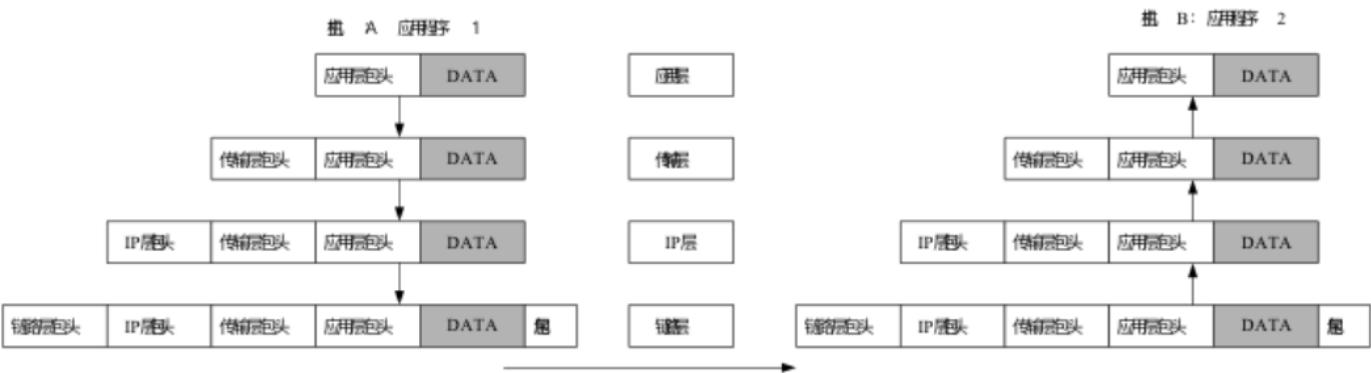
IP地址分类



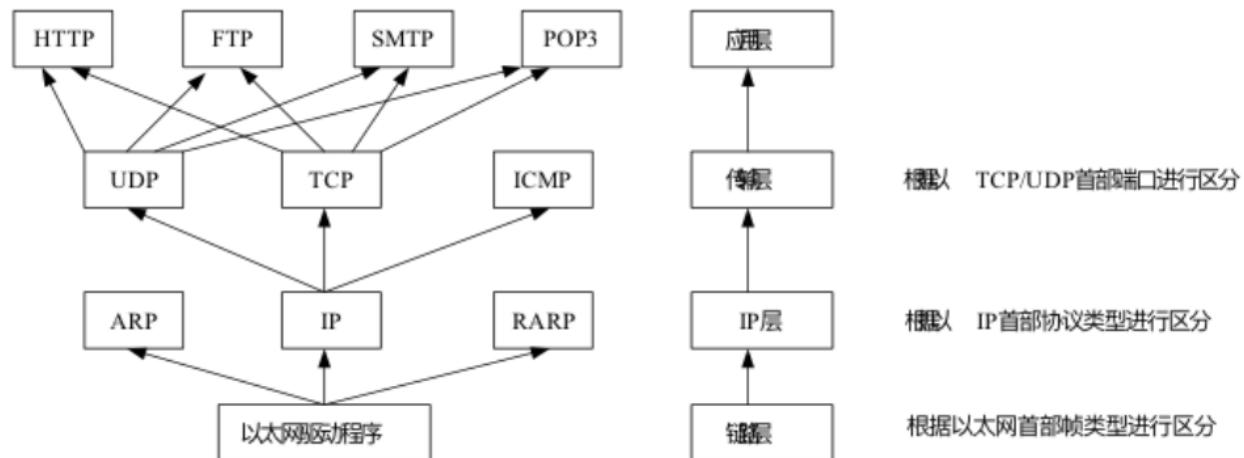
IP地址划分

类别	前8位（二进制）	点分十进制第一字节范围	默认子网掩码	广播地址	网络数
A	0XXXXXXX	1~126 (127为回环地址)	255.0.0.0	X.255.255.255	126
B	10XXXXXX	128~191	255.255.0.0	X.X.255.255	16384
C	110XXXXX	192~223	255.255.255.0	X.X.X.255	2097152
D	1110XXXX	224~239	N/A	N/A	N/A
E	1111XXXX	240~254	N/A	N/A	N/A

网络数据包封装与拆包过程



数据包接收拆包分类流程



以太网链路层数据帧格式

6字节	6字节	4字节	46~1510字节	2字节
目的MAC地址	源MAC地址	类型	数据	CRC

0800 IP

0806 ARP请求/应答

0835 RARP请求/应答

IP数据包头

```
1 //come from /usr/include/linux/ip.h
2 struct iphdr {
3 #if defined(__LITTLE_ENDIAN)           //小端定义方式
4     uint8_t      ihl:4,                //表头长度
5                 version:4;            //版本
6 #elif defined (__BIG_ENDIAN)          //大端时
7     uint8_t version:4,                //服务类型
8         ihl:4;                      //总长度 I
9 #endif
10    uint8_t tos;
11    uint16_t tot_len;
12    uint16_t id;
13    uint16_t frag_off;
14    uint8_t ttl;
15    uint8_t protocol;
16    uint16_t check;
17    uint32_t saddr;
18    uint32_t daddr;
19    /*The options start here. */
20};
```

TCP包头

```
1 //come from /usr/include/linux/tcp.h
2 struct tcphdr {
3     __u16    source;           //源端口号
4     __u16    dest;            //目的端口号
5     __u32    seq;             I //封装序号
6     __u32    ack_seq;         //ACK序号
7 #if defined(__LITTLE_ENDIAN_BITFIELD) //小端时
8     __u16    res1:4,          doff:4,
9                 fin:1,           //传送结束
10                syn:1,           //建立同步
11                rst:1,           psh:1,
12                ack:1,            //确认数据包
13                urg:1,            //紧急数据包
14                ece:1,            cwr:1;
15 #elif defined(__BIG_ENDIAN_BITFIELD) //大端时
16     .....
17 #else
18     #error "Adjust your <asm/byteorder.h> defines"
19 #endif
20     __u16    window;          //滑动窗口大小
21     __u16    check;           //检验码
22     __u16    urg_ptr;         //紧急信息
```

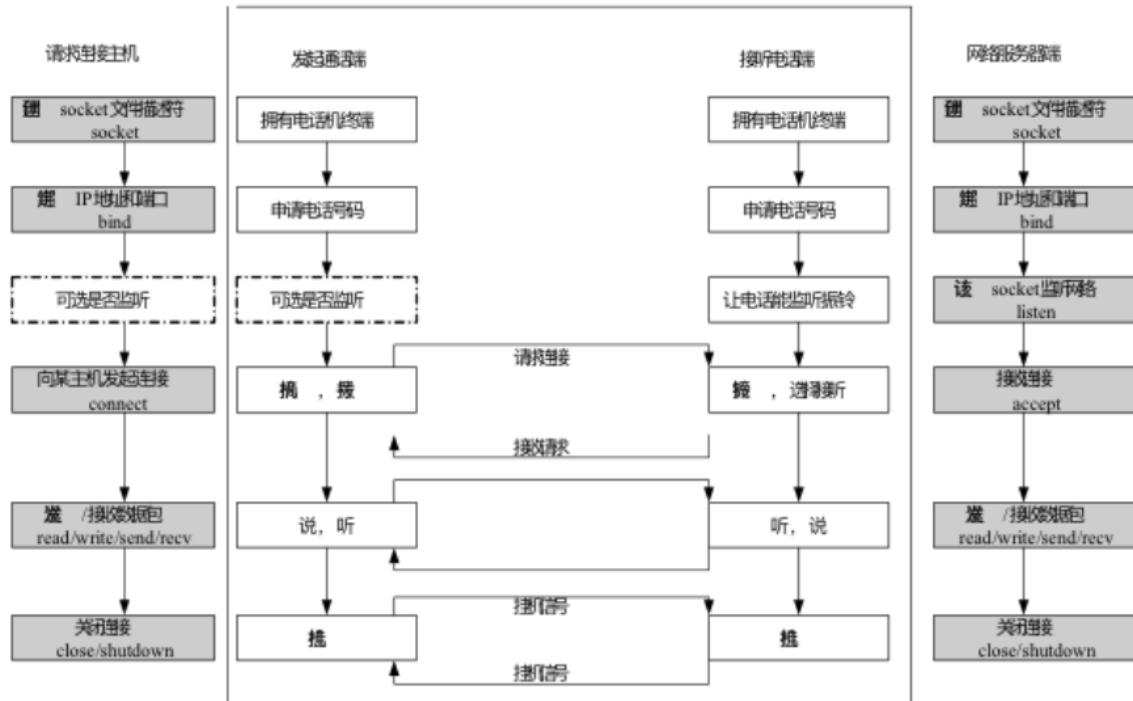
UDP数据包头

```
1 //come from /usr/include/linux/udp.h
2 struct udphdr {
3     __u16    source;      //源端口号
4     __u16    dest;        //目的端口号
5     __u16    len;         //信息长度
6     __u16    check;       //检验和
7 };
```

第12章 Linux socket网络编程基础

- 1 网络通信基础
- 2 BSD Socket TCP网络通信编程
- 3 BSD Socket UDP网络通信编程
- 4 使用TCP实现简单聊天程序

类比电话通信，面向连接的socket通信实现



通信过程

首先，服务器端需要做以下准备工作：

- (1) 调用socket()函数。建立socket对象，指定通信协议。
- (2) 调用bind()函数。将创建的socket对象与当前主机的某一个IP地址和端口绑定。
- (3) 调用listen()函数。使socket对象处于监听状态，并设置监听队列大小。

客户端需要做以下准备工作：

- (1) 调用socket()函数。建立socket()对象，指定相同通信协议。
- (2) 应用程序可以显式的调用bind()函数为其绑定IP地址和端口，当然，也可以将这工作交给TCP/IP协议栈。

接着建立通信连接：

- (1) 客户端调用connect()函数。向服务器端发出连接请求。
- (2) 服务端监听到该请求，调用accept()函数接受请求，从而建立连接，并返回一个新的socket文件描述符专门处理该连接。

然后通信双方发送/接收数据：

- (1) 服务器端调用write()或send()函数发送数据，客户端调用read()或者recv()函数接收数据。反之客户端发送数据，服务器端接收数据。
- (2) 通信完成后，通信双方都需要调用close()或者shutdown()函数关闭socket对象。

BSD Socket网络编程API **socket**

```
1 //come from /usr/include/sys/socket.h
2 extern int socket (int _domain, int _type, int _protocol)
3 此函数如果执行成功，将返回一个打开的socket文件描述符。
4 此时，该socket对象没有绑定任何IP信息，还不能进行通信。
5 如果执行失败，将返回-1。
```

Socket参数说明

- 第一个参数用来指明此socket对象所使用的地址簇或协议簇.

```
//come from /usr/include/bit/socket.h

/* Protocol families. */
#define PF_UNSPEC    0          /* Unspecified. */           //未定义
#define PF_LOCAL     1          /* Local to host (pipes and file-domain). */ //本地通信
#define PF_UNIX      PF_LOCAL   /* Old BSD name for PF_LOCAL. */ //旧的BSD名称
#define PF_FILE      PF_LOCAL   /* Another non-standard name for PF_LOCAL. */ //另一个非标准名称
#define PF_INET      2          /* IP protocol family. */        //IPV4协议簇
#define PF_AX25      3          /* Amateur Radio AX.25. */       //AX.25
#define PF_IPX       4          /* Novell Internet Protocol. */ //Novell网协议
#define PF_APPLETALK 5          /* Appletalk DDP. */           //苹果Talk
#define PF_NETROM    6          /* Amateur radio NetROM. */      //业余无线电NetROM
#define PF_BRIDGE    7          /* Multiprotocol bridge. */     //多协议桥接
#define PF_ATMPVC   8          /* ATM PVCs. */                //ATM
#define PF_X25      9          /* Reserved for X.25 project. */ //保留用于X.25项目
#define PF_INET6    10         /* IP version 6. */             //IPV6
```

Socket参数说明

- 第二个参数为socket的类型。

```
//come from /usr/include/bits/socket.h
/* Types of sockets. */
enum __socket_type{
    SOCK_STREAM = 1,           /* Sequenced, reliable, connection-based byte streams. */
#define SOCK_STREAM SOCK_STREAM      //可靠的，面向连接的流socket，即TCP
    SOCK_DGRAM = 2,            /* Connectionless, unreliable datagrams of fixed maximum length. */
#define SOCK_DGRAM SOCK_DGRAM      //不可靠的，面向无连接的数据报socket，即UDP
    SOCK_RAW = 3,              /* Raw protocol interface. */ //原始套接口
```

BSD Socket网络编程API bind

```
//come from /usr/include/sys/socket.h
/* Give the socket FD the local address ADDR (which is LEN bytes long). */
extern int bind (int __fd, __CONST_SOCKADDR_ARG __addr, socklen_t __len)
```

此函数将指定 socket 与对应网络地址（含有 IP 和端口信息）绑定，如果执行成功，将返回 0，如果执行失败，将返回 -1。

第一个参数是由 socket 函数返回的文件描述符。

- 完成此步，该socket拥有了本地IP地址，端口，通信协议，不能接收客户端的请求，但可以向服务器发起连接。

Bind参数说明

- 第二个参数是一个指向sockaddr结构的指针。 struct sockaddr 只是提供地址类型规范，根据不同的应用， sockaddr需要选用不同的类型。

```
//come from /usr/include/linux/socket.h
#define __CONST_SOCKADDR_ARG      __const struct sockaddr *
struct sockaddr {
    sa_family_t   sa_family;           /* address family, AF_xxx*/ //协议簇
    char        sa_data[14];          /* 14 bytes of protocol address*/ //协议地址
};
```

地址结构体定义-- UNIX域套接字

```
//come from /usr/include/sys/un.h
/* Structure describing the address of an AF_LOCAL (aka AF_UNIX) socket. */
#define __SOCKADDR_COMMON(sa_prefix)    sa_family_t sa_prefix##family
struct sockaddr_un
{
    __SOCKADDR_COMMON (sun_);
    char sun_path[108];           /* Path name. */           //文件路径名
};
```

地址结构体定义-- IPV4

```
//come from /usr/include/netinet/in.h
/* Structure describing an Internet socket address. */

struct sockaddr_in
{
    __SOCKADDR_COMMON (sin_);
                                //协议 AF_INET
    in_port_t sin_port;          /* Port number. */           //端口号
    struct in_addr sin_addr;     /* Internet address. */      //IP地址
    /* Pad to size of `struct sockaddr'. */
    unsigned char sin_zero[sizeof (struct sockaddr) -
                           __SOCKADDR_COMMON_SIZE -
                           sizeof (in_port_t) -
                           sizeof (struct in_addr)]; //预留位，以适应struct sockaddr位
};
```

端口与IP地址

```
[root@localhost ~]# head /etc/services
# /etc/services:
...
# service-name    port/protocol    [aliases ...]    [# comment]
...
tcpmux          1/tcp           # TCP port service multiplexer
tcpmux          1/udp           # TCP port service multiplexer
rje             5/tcp           # Remote Job Entry
rje             5/udp           # Remote Job Entry
```

struct in_addr 为 32 位 IP 地址（类型为 unsigned int 型），定义如下：

```
//come from /usr/include/linux/in.h
/* Internet address. */
struct in_addr {
    __u32 s_addr;
};
```

创建消息队列

```
extern int msgget (key_t __key, int __msgflg);
```

第一个参数 `key` 为由 `ftok` 创建的 `key` 值，关于 `ftok` 函数本章在前一小节已经介绍。

第二个参数 `_msgflg` 的低位用来确定消息队列的访问权限。如 `0770`，为文件的访问权限类型。此外，还可以附加以下参数值。这些值可以与基本权限以或的方式一起使用。

```
//come from /usr/include/bit/ipc.h
/* resource get request flags */

#define IPC_CREAT    00001000      /* create if key is nonexistent */如果key不存在，则创建
                                         //存在，返回ID
#define IPC_EXCL     00002000      /* fail if key exists */       如果key存在，返回失败
#define IPC_NOWAIT   00004000      /* return error on wait */    如果需要等待，直接返回错误
```

BSD Socket网络编程API **listen**

```
//come from /usr/include/sys/socket.h
/* Prepare to accept connections on socket FD. N connection requests will be queued before further requests
are refused. Returns 0 on success, -1 for errors. */
extern int listen (int __fd, int __n);
```

- 第一个参数是绑定了IP及端口信息的socket文件描述符。
- 第二个参数为请求排队的最大长度。当有多个客户端程序和服务器端相连时，此值表示可以使用的处于等待的队列长度。
- **listen** 函数将绑定的socket文件描述符变为监听套接字，完成此步：**服务器已经准备接收客户端连接请求了。**

BSD Socket网络编程API

客户端发起连接 connect

```
//come from /usr/include/sys/socket.h
```

```
/*Open a connection on socket FD to peer at ADDR (which LEN bytes long).For connectionless socket types,  
just set the default address to send to and the only address from which to accept transmissions. */
```

```
extern int connect (int __fd, __CONST_SOCKADDR_ARG __addr, socklen_t __len);
```

- 其第一个参数为socket返回的文件描述符。
- 第二个参数储存了服务器端的地址（包括服务器的IP地址和端口信息）。
- 第三个参数为该地址的长度。
- 如果执行成功，此函数将与地址为addr的服务器建立连接，并返回0，如果失败则返回-1。
- **正确完成此步：客户端socket拥有了目的IP，端口信息。**

```
//come from /usr/include/sys/socket.h
/*
 * wait a connection on socket FD. When a connection arrives, open a new socket to communicate with it,
 * set *ADDR (which is *ADDR_LEN bytes long) to the address of the connecting peer and *ADDR_LEN to the
 * address's actual length, and return the new socket's descriptor, or -1 for errors. */
extern int accept (int __fd, __SOCKADDR_ARG __addr, socklen_t * __restrict __addr_len);
```

- 第一个参数是监听网络后的socket文件描述符。
- 第二参数为struct sockaddr 类型的地址空间首地址，第三个参数为该段地址空间长度，因此第二个参数用来存储客户端的IP地址和端口信息，以便为客户端返回数据。
- 需要注意的是，如果执行成功，此函数将**返回一个新的文件描述符**以标识该连接，从而使原来的文件描述符可以继续等待新的连接，这样便可以实现多客户端。如果执行失败，将返回-1。
- 至此，两端的连接已经建立，而**服务器端又是如何区别多个连接的呢？**

如何区分多个客户端

- 对于任何一个TCP连接，最重要的信息包括源IP：源端口，目的IP：目的端口四个信息。例如，客户机192.168.0.10/24的3000、4000两端口同时向服务器192.168.0.100/24的80端口发起两个连接，在服务器端是如何区别两个连接的呢？

文件描述符	源IP	源端口	目的IP	目的端口	描述
fd=3	192.168.0.100/24	80	*.*.*.*	*	此fd用来继续监听网络
fd=4	192.168.0.100/24	80	192.168.0.10/24	3000	此fd用来处理第1个连接
fd=5	192.168.0.100/24	80	192.168.0.10/24	4000	此fd用来处理第2个连接

BSD Socket网络编程API 读/写socket

```
//come from /usr/include/unistd.h
/* Read NBYTES into BUF from FD.  Return the number read, -1 for errors or 0 for EOF. */
extern ssize_t read (int __fd, void * __buf, size_t __nbytes) __wur;           //读文件内容
/* Write N bytes of BUF to FD.  Return the number written, or -1. */
extern ssize_t write (int __fd, __const void * __buf, size_t __n) __wur;      //写文件内容
```

BSD Socket网络编程API send/recv

```
//come from /usr/include/sys/socket.h
/* Send N bytes of BUF to socket FD. Returns the number sent or -1. */
extern ssize_t send (int __fd, __const void * __buf, size_t __n, int __flags);

//come from /usr/include/sys/socket.h
/* Read N bytes into BUF from socket FD. Returns the number read or -1 for errors. */
extern ssize_t recv (int __fd, void * __buf, size_t __n, int __flags);
```

- 第一个参数为发送的目标socket对象；
- 第二个参数为欲发送的数据位置；
- 第三个参数为数据的大小；
- 第四个参数操作flags，支持的值为0或MSG_OOB（发送带外数据）等。对套接字调用write()的行为与将flags设置为0的send()的行为完全相同。
- 如果执行成功，此函数将返回发送数据的大小，如果失败，将返回-1。

BSD Socket网络编程API 关闭socket对象 close/shutdown

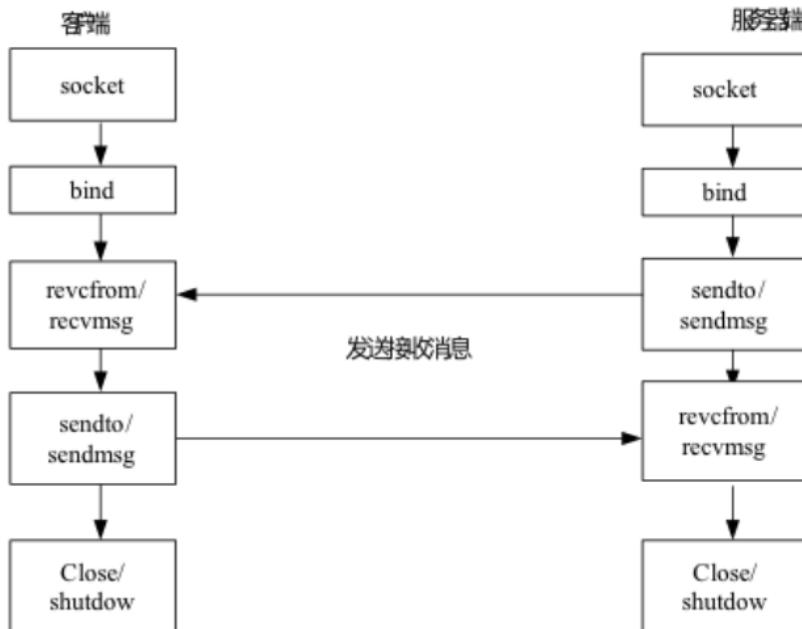
```
//come from /usr/include/sys/socket.h
/* Shut down all or part of the connection open on socket FD.
   HOW determines what to shut down:
   SHUT_RD    = No more receptions;          //无接收
   SHUT_WR    = No more transmissions;        //无发送
   SHUT_RDWR  = No more receptions or transmissions. //无接收和发送
   Returns 0 on success, -1 for errors. */
extern int shutdown (int _fd, int _how);
```

- howto=0 这个时候系统会关闭读通道，但是可以继续往socket描述符中写。
- howto=1 关闭写通道，和上面相反，此时只可以读。
- howto=2 关闭读写通道，和close一样，在多进程程序里，当几个子进程共享一个套接字时，如果使用shutdown，那么所有的子进程都将不能操作，这时只能使用close()函数来关闭子进程的套接字描述符。

第12章 Linux socket网络编程基础

- 1 网络通信基础
- 2 BSD Socket TCP网络通信编程
- 3 BSD Socket UDP网络通信编程
- 4 使用TCP实现简单聊天程序

面向无连接通信模型



BSD Socket网络编程API sendto/ recvfrom

```
//come from /usr/include/sys/socket.h
/* Send N bytes of BUF on socket FD to peer at address ADDR (which is _ADDR_LEN bytes long).
```

Returns the number sent, or -1 for errors.

```
extern ssize_t sendto (int __fd, __const void *__buf, size_t __n,int __flags, __CONST_SOCKADDR_ARG
__addr, socklen_t __addr_len);
```

- 第一个参数为发送的目标socket对象。
- 第二个参数为欲发送的数据信息。
- 第三个参数为发送数据的大小。
- 第四个参数为flags，如send函数所示。
- 第五个参数欲发送数据的目标地址，其结构体前面已经介绍。
- 第六个参数为此结构体的大小。

```
extern ssize_t recvfrom (int __fd, void * __restrict __buf, size_t __n, int __flags, __SOCKADDR_ARG
__addr, socklen_t * __restrict __addr_len);
```

BSD Socket网络编程API `getsockname / getpeername`

- 获得一个套接字（这个套接口至少完成了绑定本地IP地址）的本地地址。

```
extern int getsockname (int _fd, _SOCKADDR_ARG _addr, socklen_t * _restrict _len);
```

如果成功则返回0，如果发生错误则返回-1。

第1个参数为欲读取信息的socket文件描述符

第2, 3个参数分别为存储地址的内存空间地址和大小。

`getpeername()`函数将取得一个已经连接上的套接字的远程信息（比如IP地址和端口）。

```
extern int getpeername (int _fd, _SOCKADDR_ARG _addr, socklen_t * _restrict _len);
```

```
struct sockaddr_test;
getsockname(new_fd,(struct sockaddr *)&test, &size);           //已经完成的连接
printf("ip=%os,port=%od\n",inet_ntoa(test.sin_addr),ntohs(test.sin_port));
```

应用示例

- 使用 AF_UNIX 实现本机数据流通信示例 见代码
- 使用 AF_INET 实现 UDP 点对点通信示例 见代码

第12章 Linux socket网络编程基础

-  **1** 网络通信基础
-  **2** BSD Socket TCP网络通信编程
-  **3** BSD Socket UDP网络通信编程
-  **4** 使用TCP实现简单聊天程序

服务端运行结果 (IP地址为192.168.0.93)

```
[yangzongde@localhost sock]$ ./tcp_p_p_chat_server 192.168.0.93      7575      5  
绑定自己的IP地址  绑定端口  等待队列大小
```

运行过程中，提示信息如下：

```
server: got connection from 192.168.0.93, port 53095, socket 4  
newfd=4  
input the message to send:hello          //要求输入消息  
message:hello  
send successful,send 5byte!  
the other one close quit
```

在服务器另一终端运行 netstat 命令，其中一项记录信息如下：

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
协议	接收队列	发送队列	本地地址	对方地址	TCP状态	进程/程序名
tcp	0	0	192.168.0.93:7575	192.168.0.133:34438	ESTABLISHED	12260/tcp_p_p_chat_s

客户机运行结果(IP地址为192.168.0.133)

```
[yangzongde@localhost sock]$ ./tcp_p_p_chat_client    192.168.0.93      7575
                                                服务器的IP地址      服务端开放的端口

socket created.

server connected.

recv successful:'hello',5 byte recv                //收到的消息

pls send message to send:quit                      //发送退出消息

i will quit!
```

在客户端另一终端运行 netstat 命令，其中一项记录信息如下：

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
协议	接收队列	发送队列	本地地址	对方地址	TCP状态	进程/程序名
tcp	0	0	192.168.0.133:34438	192.168.0.93:7575	ESTABLISHED	901/tcp_p_p_chat_c

- 此程序只实现一端到端的数据传递，且只能一发一收的方式。
- 具体见代码分析。

习题

- (1) 七层模型与TCP/IP协议模型比较，各层完成的基本功能，对应的各层最主要功能是什么？
- (2) 写出你所知道的网络设备、网络协议其简单工作原理描述，并列出其工作在TCP/IP协议栈的哪一层。
- (3) 试写出网络数据的封包与拆包过程。并分析说明TCP、IP、UDP数据包头信息。
- (4) IP地址的如何区分，A、B、C类地址范围，哪些地址是私有地址，能够实现子网合并和拆分。子网掩码是什么，怎样计算一台主机的网络ID和主机ID。将192.168.0.1/24网段划分成8个子网，写出网络ID，主机ID范围，广播地址，子网掩码。
- (5) 为什么办公区的私有IP地址主机可以连接到internet，NAT的功能是什么？