**LINUX 环境进程间通信（全）**

目录

[一．Linux环境进程间通信（一）：管道及有名管道 3](#_Toc288041323)

[1、 管道概述及相关API应用 3](#_Toc288041324)

[1.1 管道相关的关键概念 3](#_Toc288041325)

[1.2管道的创建： 3](#_Toc288041326)

[1.3管道的读写规则： 3](#_Toc288041327)

[1.4管道应用实例： 8](#_Toc288041328)

[1.5管道的局限性 10](#_Toc288041329)

[2、 有名管道概述及相关API应用 10](#_Toc288041330)

[2.1 有名管道相关的关键概念 10](#_Toc288041331)

[2.2有名管道的创建 10](#_Toc288041332)

[2.3有名管道的打开规则 10](#_Toc288041333)

[2.4有名管道的读写规则 11](#_Toc288041334)

[2.5有名管道应用实例 14](#_Toc288041335)

[小结： 14](#_Toc288041336)

[二．Linux环境进程间通信（二）：信号（上） 17](#_Toc288041337)

[1、信号及信号来源 17](#_Toc288041338)

[1.1信号本质 17](#_Toc288041339)

[1.2信号来源 17](#_Toc288041340)

[2、信号的种类 17](#_Toc288041341)

[2.1可靠信号与不可靠信号 17](#_Toc288041342)

[2.2实时信号与非实时信号 18](#_Toc288041343)

[3、进程对信号的响应 18](#_Toc288041344)

[4、信号的发送 19](#_Toc288041345)

[5、信号的安装（设置信号关联动作） 20](#_Toc288041346)

[6、信号集及信号集操作函数： 23](#_Toc288041347)

[7、信号阻塞与信号未决 24](#_Toc288041348)

[三．Linux环境进程间通信（二）：信号（下） 26](#_Toc288041349)

[1、信号生命周期 26](#_Toc288041350)

[2、信号编程注意事项 27](#_Toc288041351)

[2.1 防止不该丢失的信号丢失 27](#_Toc288041352)

[2.2 程序的可移植性 27](#_Toc288041353)

[2.3 程序的稳定性 28](#_Toc288041354)

[3、深入浅出：信号应用实例 29](#_Toc288041355)

[实例一：信号发送及处理 29](#_Toc288041356)

[实例二：信号传递附加信息 30](#_Toc288041357)

[实例三：信号阻塞及信号集操作 32](#_Toc288041358)

[结束语： 33](#_Toc288041359)

[四．Linux环境进程间通信（三）：消息队列 34](#_Toc288041360)

[1、消息队列基本概念 34](#_Toc288041361)

[2、操作消息队列 35](#_Toc288041362)

[消息队列API 36](#_Toc288041363)

[3、消息队列的限制 38](#_Toc288041364)

[4、消息队列应用实例 39](#_Toc288041365)

[小结： 41](#_Toc288041366)

[五．Linux环境进程间通信（四）：信号灯 44](#_Toc288041367)

[1、信号灯概述 44](#_Toc288041368)

[2、Linux信号灯 44](#_Toc288041369)

[3、信号灯与内核 44](#_Toc288041370)

[4、操作信号灯 45](#_Toc288041371)

[5、信号灯的限制 47](#_Toc288041372)

[6、竞争问题 48](#_Toc288041373)

[7、信号灯应用实例 48](#_Toc288041374)

[六．Linux环境进程间通信（五）：共享内存（上） 53](#_Toc288041375)

[1、内核怎样保证各个进程寻址到同一个共享内存区域的内存页面 53](#_Toc288041376)

[2、mmap()及其相关系统调用 54](#_Toc288041377)

[2.1 mmap()系统调用形式 54](#_Toc288041378)

[2.2 系统调用mmap()用于共享内存的两种方式 54](#_Toc288041379)

[2.3 系统调用munmap() 55](#_Toc288041380)

[2.4 系统调用msync() 55](#_Toc288041381)

[3、mmap()范例 55](#_Toc288041382)

[范例1：两个进程通过映射普通文件实现共享内存通信 55](#_Toc288041383)

[范例2：父子进程通过匿名映射实现共享内存 58](#_Toc288041384)

[4、对mmap()返回地址的访问 59](#_Toc288041385)

[结论 61](#_Toc288041386)

[七．Linux环境进程间通信（五）：共享内存（下） 61](#_Toc288041387)

[1、系统V共享内存原理 61](#_Toc288041388)

[2、系统V共享内存API 62](#_Toc288041389)

[3、系统V共享内存限制 63](#_Toc288041390)

[4、系统V共享内存范例 63](#_Toc288041391)

[结论 65](#_Toc288041392)

# 一．Linux环境进程间通信（一）：管道及有名管道

## 1、 管道概述及相关API应用

### 1.1 管道相关的关键概念

管道是Linux支持的最初Unix IPC形式之一，具有以下特点：

* 管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道；
* 只能用于父子进程或者兄弟进程之间（具有亲缘关系的进程）；
* 单独构成一种独立的文件系统：管道对于管道两端的进程而言，就是一个文件，但它不是普通的文件，它不属于某种文件系统，而是自立门户，单独构成一种文件系统，并且只存在与内存中。
* 数据的读出和写入：一个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。

### 1.2管道的创建：

|  |
| --- |
| #include <unistd.h>  int pipe(int fd[2]) |

该函数创建的管道的两端处于一个进程中间，在实际应用中没有太大意义，因此，一个进程在由pipe()创建管道后，一般再fork一个子进程，然后通过管道实现父子进程间的通信（因此也不难推出，只要两个进程中存在亲缘关系，这里的亲缘关系指的是具有共同的祖先，都可以采用管道方式来进行通信）。

### 1.3管道的读写规则：

管道两端可分别用描述字fd[0]以及fd[1]来描述，需要注意的是，管道的两端是固定了任务的。即一端只能用于读，由描述字fd[0]表示，称其为管道读端；另一端则只能用于写，由描述字fd[1]来表示，称其为管道写端。如果试图从管道写端读取数据，或者向管道读端写入数据都将导致错误发生。一般文件的I/O函数都可以用于管道，如close、read、write等等。

从管道中读取数据：

* 如果管道的写端不存在，则认为已经读到了数据的末尾，读函数返回的读出字节数为0；
* 当管道的写端存在时，如果请求的字节数目大于PIPE\_BUF，则返回管道中现有的数据字节数，如果请求的字节数目不大于PIPE\_BUF，则返回管道中现有数据字节数（此时，管道中数据量小于请求的数据量）；或者返回请求的字节数（此时，管道中数据量不小于请求的数据量）。注：（PIPE\_BUF在include/linux/limits.h中定义，不同的内核版本可能会有所不同。Posix.1要求PIPE\_BUF至少为512字节，red hat 7.2中为4096）。

关于管道的读规则验证：

|  |
| --- |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* readtest.c \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #include <unistd.h>  #include <sys/types.h>  #include <errno.h>  main()  {  int pipe\_fd[2];  pid\_t pid;  char r\_buf[100];  char w\_buf[4];  char\* p\_wbuf;  int r\_num;  int cmd;    memset(r\_buf,0,sizeof(r\_buf));  memset(w\_buf,0,sizeof(r\_buf));  p\_wbuf=w\_buf;  if(pipe(pipe\_fd)<0)  {  printf("pipe create error\n");  return -1;  }    if((pid=fork())==0)  {  printf("\n");  close(pipe\_fd[1]);  sleep(3);//确保父进程关闭写端  r\_num=read(pipe\_fd[0],r\_buf,100);  printf( "read num is %d the data read from the pipe is %d\n",r\_num,atoi(r\_buf));    close(pipe\_fd[0]);  exit();  }  else if(pid>0)  {  close(pipe\_fd[0]);//read  strcpy(w\_buf,"111");  if(write(pipe\_fd[1],w\_buf,4)!=-1)  printf("parent write over\n");  close(pipe\_fd[1]);//write  printf("parent close fd[1] over\n");  sleep(10);  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 程序输出结果：  \* parent write over  \* parent close fd[1] over  \* read num is 4 the data read from the pipe is 111  \* 附加结论：  \* 管道写端关闭后，写入的数据将一直存在，直到读出为止.  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ |

向管道中写入数据：

* 向管道中写入数据时，linux将不保证写入的原子性，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读走管道缓冲区中的数据，那么写操作将一直阻塞。   
  注：只有在管道的读端存在时，向管道中写入数据才有意义。否则，向管道中写入数据的进程将收到内核传来的SIFPIPE信号，应用程序可以处理该信号，也可以忽略（默认动作则是应用程序终止）。

对管道的写规则的验证1：写端对读端存在的依赖性

|  |
| --- |
| #include <unistd.h>  #include <sys/types.h>  main()  {  int pipe\_fd[2];  pid\_t pid;  char r\_buf[4];  char\* w\_buf;  int writenum;  int cmd;    memset(r\_buf,0,sizeof(r\_buf));  if(pipe(pipe\_fd)<0)  {  printf("pipe create error\n");  return -1;  }    if((pid=fork())==0)  {  close(pipe\_fd[0]);  close(pipe\_fd[1]);  sleep(10);  exit();  }  else if(pid>0)  {  sleep(1); //等待子进程完成关闭读端的操作  close(pipe\_fd[0]);//write  w\_buf="111";  if((writenum=write(pipe\_fd[1],w\_buf,4))==-1)  printf("write to pipe error\n");  else  printf("the bytes write to pipe is %d \n", writenum);    close(pipe\_fd[1]);  }  } |

则输出结果为： Broken pipe,原因就是该管道以及它的所有fork()产物的读端都已经被关闭。如果在父进程中保留读端，即在写完pipe后，再关闭父进程的读端，也会正常写入pipe，读者可自己验证一下该结论。因此，在向管道写入数据时，至少应该存在某一个进程，其中管道读端没有被关闭，否则就会出现上述错误（管道断裂,进程收到了SIGPIPE信号，默认动作是进程终止）

对管道的写规则的验证2：linux不保证写管道的原子性验证

|  |
| --- |
| #include <unistd.h>  #include <sys/types.h>  #include <errno.h>  main(int argc,char\*\*argv)  {  int pipe\_fd[2];  pid\_t pid;  char r\_buf[4096];  char w\_buf[4096\*2];  int writenum;  int rnum;  memset(r\_buf,0,sizeof(r\_buf));  if(pipe(pipe\_fd)<0)  {  printf("pipe create error\n");  return -1;  }    if((pid=fork())==0)  {  close(pipe\_fd[1]);  while(1)  {  sleep(1);  rnum=read(pipe\_fd[0],r\_buf,1000);  printf("child: readnum is %d\n",rnum);  }  close(pipe\_fd[0]);    exit();  }  else if(pid>0)  {  close(pipe\_fd[0]);//write  memset(r\_buf,0,sizeof(r\_buf));  if((writenum=write(pipe\_fd[1],w\_buf,1024))==-1)  printf("write to pipe error\n");  else  printf("the bytes write to pipe is %d \n", writenum);  writenum=write(pipe\_fd[1],w\_buf,4096);  close(pipe\_fd[1]);  }  }  输出结果：  the bytes write to pipe 1000  the bytes write to pipe 1000 //注意，此行输出说明了写入的非原子性  the bytes write to pipe 1000  the bytes write to pipe 1000  the bytes write to pipe 1000  the bytes write to pipe 120 //注意，此行输出说明了写入的非原子性  the bytes write to pipe 0  the bytes write to pipe 0  ...... |

结论：

写入数目小于4096时写入是非原子的！   
如果把父进程中的两次写入字节数都改为5000，则很容易得出下面结论：   
写入管道的数据量大于4096字节时，缓冲区的空闲空间将被写入数据（补齐），直到写完所有数据为止，如果没有进程读数据，则一直阻塞。

### 1.4管道应用实例：

#### 实例一：用于shell

管道可用于输入输出重定向，它将一个命令的输出直接定向到另一个命令的输入。比如，当在某个shell程序（Bourne shell或C shell等）键入who│wc -l后，相应shell程序将创建who以及wc两个进程和这两个进程间的管道。考虑下面的命令行：

$kill -l 运行结果见 [附一](http://www.ibm.com/developerworks/cn/linux/l-ipc/part1/index.html#a)。

$kill -l | grep SIGRTMIN 运行结果如下：

|  |
| --- |
| 30) SIGPWR 31) SIGSYS 32) SIGRTMIN 33) SIGRTMIN+1  34) SIGRTMIN+2 35) SIGRTMIN+3 36) SIGRTMIN+4 37) SIGRTMIN+5  38) SIGRTMIN+6 39) SIGRTMIN+7 40) SIGRTMIN+8 41) SIGRTMIN+9  42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12 45) SIGRTMIN+13  46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMAX-15 49) SIGRTMAX-14 |

#### 实例二：用于具有亲缘关系的进程间通信

下面例子给出了管道的具体应用，父进程通过管道发送一些命令给子进程，子进程解析命令，并根据命令作相应处理。

|  |
| --- |
| #include <unistd.h>  #include <sys/types.h>  main()  {  int pipe\_fd[2];  pid\_t pid;  char r\_buf[4];  char\*\* w\_buf[256];  int childexit=0;  int i;  int cmd;    memset(r\_buf,0,sizeof(r\_buf));  if(pipe(pipe\_fd)<0)  {  printf("pipe create error\n");  return -1;  }  if((pid=fork())==0)  //子进程：解析从管道中获取的命令，并作相应的处理  {  printf("\n");  close(pipe\_fd[1]);  sleep(2);    while(!childexit)  {  read(pipe\_fd[0],r\_buf,4);  cmd=atoi(r\_buf);  if(cmd==0)  {  printf("child: receive command from parent over\n now child process exit\n");  childexit=1;  }    else if(handle\_cmd(cmd)!=0)  return;  sleep(1);  }  close(pipe\_fd[0]);  exit();  }  else if(pid>0)  //parent: send commands to child  {  close(pipe\_fd[0]);  w\_buf[0]="003";  w\_buf[1]="005";  w\_buf[2]="777";  w\_buf[3]="000";  for(i=0;i<4;i++)  write(pipe\_fd[1],w\_buf[i],4);  close(pipe\_fd[1]);  }  }  //下面是子进程的命令处理函数（特定于应用）：  int handle\_cmd(int cmd)  {  if((cmd<0)||(cmd>256))  //suppose child only support 256 commands  {  printf("child: invalid command \n");  return -1;  }  printf("child: the cmd from parent is %d\n", cmd);  return 0;  } |

### 1.5管道的局限性

管道的主要局限性正体现在它的特点上：

* 只支持单向数据流；
* 只能用于具有亲缘关系的进程之间；
* 没有名字；
* 管道的缓冲区是有限的（管道制存在于内存中，在管道创建时，为缓冲区分配一个页面大小）；
* 管道所传送的是无格式字节流，这就要求管道的读出方和写入方必须事先约定好数据的格式，比如多少字节算作一个消息（或命令、或记录）等等；

## 2、 有名管道概述及相关API应用

### 2.1 有名管道相关的关键概念

管道应用的一个重大限制是它没有名字，因此，只能用于具有亲缘关系的进程间通信，在有名管道（named pipe或FIFO）提出后，该限制得到了克服。FIFO不同于管道之处在于它提供一个路径名与之关联，以FIFO的文件形式存在于文件系统中。这样，即使与FIFO的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过FIFO相互通信（能够访问该路径的进程以及FIFO的创建进程之间），因此，通过FIFO不相关的进程也能交换数据。值得注意的是，FIFO严格遵循先进先出（first in first out），对管道及FIFO的读总是从开始处返回数据，对它们的写则把数据添加到末尾。它们不支持诸如lseek()等文件定位操作。

### 2.2有名管道的创建

|  |
| --- |
| #include <sys/types.h>  #include <sys/stat.h>  int mkfifo(const char \* pathname, mode\_t mode) |

该函数的第一个参数是一个普通的路径名，也就是创建后FIFO的名字。第二个参数与打开普通文件的open()函数中的mode 参数相同。如果mkfifo的第一个参数是一个已经存在的路径名时，会返回EEXIST错误，所以一般典型的调用代码首先会检查是否返回该错误，如果确实返回该错误，那么只要调用打开FIFO的函数就可以了。一般文件的I/O函数都可以用于FIFO，如close、read、write等等。

### 2.3有名管道的打开规则

有名管道比管道多了一个打开操作：open。

FIFO的打开规则：

如果当前打开操作是为读而打开FIFO时，若已经有相应进程为写而打开该FIFO，则当前打开操作将成功返回；否则，可能阻塞直到有相应进程为写而打开该FIFO（当前打开操作设置了阻塞标志）；或者，成功返回（当前打开操作没有设置阻塞标志）。

如果当前打开操作是为写而打开FIFO时，如果已经有相应进程为读而打开该FIFO，则当前打开操作将成功返回；否则，可能阻塞直到有相应进程为读而打开该FIFO（当前打开操作设置了阻塞标志）；或者，返回ENXIO错误（当前打开操作没有设置阻塞标志）。

对打开规则的验证参见 [附2](http://www.ibm.com/developerworks/cn/linux/l-ipc/part1/index.html#b)。

### 2.4有名管道的读写规则

从FIFO中读取数据：

约定：如果一个进程为了从FIFO中读取数据而阻塞打开FIFO，那么称该进程内的读操作为设置了阻塞标志的读操作。

* 如果有进程写打开FIFO，且当前FIFO内没有数据，则对于设置了阻塞标志的读操作来说，将一直阻塞。对于没有设置阻塞标志读操作来说则返回-1，当前errno值为EAGAIN，提醒以后再试。
* 对于设置了阻塞标志的读操作说，造成阻塞的原因有两种：当前FIFO内有数据，但有其它进程在读这些数据；另外就是FIFO内没有数据。解阻塞的原因则是FIFO中有新的数据写入，不论信写入数据量的大小，也不论读操作请求多少数据量。
* 读打开的阻塞标志只对本进程第一个读操作施加作用，如果本进程内有多个读操作序列，则在第一个读操作被唤醒并完成读操作后，其它将要执行的读操作将不再阻塞，即使在执行读操作时，FIFO中没有数据也一样（此时，读操作返回0）。
* 如果没有进程写打开FIFO，则设置了阻塞标志的读操作会阻塞。

注：如果FIFO中有数据，则设置了阻塞标志的读操作不会因为FIFO中的字节数小于请求读的字节数而阻塞，此时，读操作会返回FIFO中现有的数据量。

向FIFO中写入数据：

约定：如果一个进程为了向FIFO中写入数据而阻塞打开FIFO，那么称该进程内的写操作为设置了阻塞标志的写操作。

对于设置了阻塞标志的写操作：

* 当要写入的数据量不大于PIPE\_BUF时，linux将保证写入的原子性。如果此时管道空闲缓冲区不足以容纳要写入的字节数，则进入睡眠，直到当缓冲区中能够容纳要写入的字节数时，才开始进行一次性写操作。
* 当要写入的数据量大于PIPE\_BUF时，linux将不再保证写入的原子性。FIFO缓冲区一有空闲区域，写进程就会试图向管道写入数据，写操作在写完所有请求写的数据后返回。

对于没有设置阻塞标志的写操作：

* 当要写入的数据量大于PIPE\_BUF时，linux将不再保证写入的原子性。在写满所有FIFO空闲缓冲区后，写操作返回。
* 当要写入的数据量不大于PIPE\_BUF时，linux将保证写入的原子性。如果当前FIFO空闲缓冲区能够容纳请求写入的字节数，写完后成功返回；如果当前FIFO空闲缓冲区不能够容纳请求写入的字节数，则返回EAGAIN错误，提醒以后再写；

对FIFO读写规则的验证：

下面提供了两个对FIFO的读写程序，适当调节程序中的很少地方或者程序的命令行参数就可以对各种FIFO读写规则进行验证。

#### 程序1：写FIFO的程序

|  |
| --- |
| #include <sys/types.h>  #include <sys/stat.h>  #include <errno.h>  #include <fcntl.h>  #define FIFO\_SERVER "/tmp/fifoserver"  main(int argc,char\*\* argv)  //参数为即将写入的字节数  {  int fd;  char w\_buf[4096\*2];  int real\_wnum;  memset(w\_buf,0,4096\*2);  if((mkfifo(FIFO\_SERVER,O\_CREAT|O\_EXCL)<0)&&(errno!=EEXIST))  printf("cannot create fifoserver\n");  if(fd==-1)  if(errno==ENXIO)  printf("open error; no reading process\n");    fd=open(FIFO\_SERVER,O\_WRONLY|O\_NONBLOCK,0);  //设置非阻塞标志  //fd=open(FIFO\_SERVER,O\_WRONLY,0);  //设置阻塞标志  real\_wnum=write(fd,w\_buf,2048);  if(real\_wnum==-1)  {  if(errno==EAGAIN)  printf("write to fifo error; try later\n");  }  else  printf("real write num is %d\n",real\_wnum);  real\_wnum=write(fd,w\_buf,5000);  //5000用于测试写入字节大于4096时的非原子性  //real\_wnum=write(fd,w\_buf,4096);  //4096用于测试写入字节不大于4096时的原子性    if(real\_wnum==-1)  if(errno==EAGAIN)  printf("try later\n");  } |

#### 程序2：与程序1一起测试写FIFO的规则，第一个命令行参数是请求从FIFO读出的字节数

|  |
| --- |
| #include <sys/types.h>  #include <sys/stat.h>  #include <errno.h>  #include <fcntl.h>  #define FIFO\_SERVER "/tmp/fifoserver"  main(int argc,char\*\* argv)  {  char r\_buf[4096\*2];  int fd;  int r\_size;  int ret\_size;  r\_size=atoi(argv[1]);  printf("requred real read bytes %d\n",r\_size);  memset(r\_buf,0,sizeof(r\_buf));  fd=open(FIFO\_SERVER,O\_RDONLY|O\_NONBLOCK,0);  //fd=open(FIFO\_SERVER,O\_RDONLY,0);  //在此处可以把读程序编译成两个不同版本：阻塞版本及非阻塞版本  if(fd==-1)  {  printf("open %s for read error\n");  exit();  }  while(1)  {    memset(r\_buf,0,sizeof(r\_buf));  ret\_size=read(fd,r\_buf,r\_size);  if(ret\_size==-1)  if(errno==EAGAIN)  printf("no data avlaible\n");  printf("real read bytes %d\n",ret\_size);  sleep(1);  }  pause();  unlink(FIFO\_SERVER);  } |

程序应用说明：

把读程序编译成两个不同版本：

* 阻塞读版本:br
* 以及非阻塞读版本nbr

把写程序编译成两个四个版本：

* 非阻塞且请求写的字节数大于PIPE\_BUF版本：nbwg
* 非阻塞且请求写的字节数不大于PIPE\_BUF版本：版本nbw
* 阻塞且请求写的字节数大于PIPE\_BUF版本：bwg
* 阻塞且请求写的字节数不大于PIPE\_BUF版本：版本bw

下面将使用br、nbr、w代替相应程序中的阻塞读、非阻塞读

验证阻塞写操作：

1. 当请求写入的数据量大于PIPE\_BUF时的非原子性：
   * nbr 1000
   * bwg
2. 当请求写入的数据量不大于PIPE\_BUF时的原子性：
   * nbr 1000
   * bw

验证非阻塞写操作：

1. 当请求写入的数据量大于PIPE\_BUF时的非原子性：
   * nbr 1000
   * nbwg
2. 请求写入的数据量不大于PIPE\_BUF时的原子性：
   * nbr 1000
   * nbw

不管写打开的阻塞标志是否设置，在请求写入的字节数大于4096时，都不保证写入的原子性。但二者有本质区别：

对于阻塞写来说，写操作在写满FIFO的空闲区域后，会一直等待，直到写完所有数据为止，请求写入的数据最终都会写入FIFO；

而非阻塞写则在写满FIFO的空闲区域后，就返回(实际写入的字节数)，所以有些数据最终不能够写入。

对于读操作的验证则比较简单，不再讨论。

### 2.5有名管道应用实例

在验证了相应的读写规则后，应用实例似乎就没有必要了。

## 小结：

管道常用于两个方面：（1）在shell中时常会用到管道（作为输入输入的重定向），在这种应用方式下，管道的创建对于用户来说是透明的；（2）用于具有亲缘关系的进程间通信，用户自己创建管道，并完成读写操作。

FIFO可以说是管道的推广，克服了管道无名字的限制，使得无亲缘关系的进程同样可以采用先进先出的通信机制进行通信。

管道和FIFO的数据是字节流，应用程序之间必须事先确定特定的传输"协议"，采用传播具有特定意义的消息。

要灵活应用管道及FIFO，理解它们的读写规则是关键。

#### 附1

kill -l 的运行结果，显示了当前系统支持的所有信号：

|  |
| --- |
| 1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL  5) SIGTRAP 6) SIGABRT 7) SIGBUS 8) SIGFPE  9) SIGKILL 10) SIGUSR1 11) SIGSEGV 12) SIGUSR2  13) SIGPIPE 14) SIGALRM 15) SIGTERM 17) SIGCHLD  18) SIGCONT 19) SIGSTOP 20) SIGTSTP 21) SIGTTIN  22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ  26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO  30) SIGPWR 31) SIGSYS 32) SIGRTMIN 33) SIGRTMIN+1  34) SIGRTMIN+2 35) SIGRTMIN+3 36) SIGRTMIN+4 37) SIGRTMIN+5  38) SIGRTMIN+6 39) SIGRTMIN+7 40) SIGRTMIN+8 41) SIGRTMIN+9  42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12 45) SIGRTMIN+13  46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMAX-15 49) SIGRTMAX-14  50) SIGRTMAX-13 51) SIGRTMAX-12 52) SIGRTMAX-11 53) SIGRTMAX-10  54) SIGRTMAX-9 55) SIGRTMAX-8 56) SIGRTMAX-7 57) SIGRTMAX-6  58) SIGRTMAX-5 59) SIGRTMAX-4 60) SIGRTMAX-3 61) SIGRTMAX-2  62) SIGRTMAX-1 63) SIGRTMAX |

除了在此处用来说明管道应用外，接下来的专题还要对这些信号分类讨论。

#### 附2

对FIFO打开规则的验证（主要验证写打开对读打开的依赖性）

|  |
| --- |
| #include <sys/types.h>  #include <sys/stat.h>  #include <errno.h>  #include <fcntl.h>  #define FIFO\_SERVER "/tmp/fifoserver"  int handle\_client(char\*);  main(int argc,char\*\* argv)  {  int r\_rd;  int w\_fd;  pid\_t pid;  if((mkfifo(FIFO\_SERVER,O\_CREAT|O\_EXCL)<0)&&(errno!=EEXIST))  printf("cannot create fifoserver\n");  handle\_client(FIFO\_SERVER);    }  int handle\_client(char\* arg)  {  int ret;  ret=w\_open(arg);  switch(ret)  {  case 0:  {  printf("open %s error\n",arg);  printf("no process has the fifo open for reading\n");  return -1;  }  case -1:  {  printf("something wrong with open the fifo except for ENXIO");  return -1;  }  case 1:  {  printf("open server ok\n");  return 1;  }  default:  {  printf("w\_no\_r return ----\n");  return 0;  }  }  unlink(FIFO\_SERVER);  }  int w\_open(char\*arg)  //0 open error for no reading  //-1 open error for other reasons  //1 open ok  {  if(open(arg,O\_WRONLY|O\_NONBLOCK,0)==-1)  { if(errno==ENXIO)  {  return 0;  }  else  return -1;  }  return 1;    } |

# 二．Linux环境进程间通信（二）：信号（上）

## 1、信号及信号来源

### 1.1信号本质

信号是在软件层次上对中断机制的一种模拟，在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是异步的，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达。

信号是进程间通信机制中唯一的异步通信机制，可以看作是异步通知，通知接收信号的进程有哪些事情发生了。信号机制经过POSIX实时扩展后，功能更加强大，除了基本通知功能外，还可以传递附加信息。

### 1.2信号来源

信号事件的发生有两个来源：硬件来源(比如我们按下了键盘或者其它硬件故障)；软件来源，最常用发送信号的系统函数是kill, raise, alarm和setitimer以及sigqueue函数，软件来源还包括一些非法运算等操作。

## 2、信号的种类

可以从两个不同的分类角度对信号进行分类：（1）可靠性方面：可靠信号与不可靠信号；（2）与时间的关系上：实时信号与非实时信号。在《Linux环境进程间通信（一）：管道及有名管道》的附1中列出了系统所支持的所有信号。

### 2.1可靠信号与不可靠信号

**"不可靠信号"**

Linux信号机制基本上是从Unix系统中继承过来的。早期Unix系统中的信号机制比较简单和原始，后来在实践中暴露出一些问题，因此，把那些建立在早期机制上的信号叫做"不可靠信号"，信号值小于SIGRTMIN(Red hat 7.2中，SIGRTMIN=32，SIGRTMAX=63)的信号都是不可靠信号。这就是"不可靠信号"的来源。它的主要问题是：

* 进程每次处理信号后，就将对信号的响应设置为默认动作。在某些情况下，将导致对信号的错误处理；因此，用户如果不希望这样的操作，那么就要在信号处理函数结尾再一次调用signal()，重新安装该信号。
* 信号可能丢失，后面将对此详细阐述。   
  因此，早期unix下的不可靠信号主要指的是进程可能对信号做出错误的反应以及信号可能丢失。

Linux支持不可靠信号，但是对不可靠信号机制做了改进：在调用完信号处理函数后，不必重新调用该信号的安装函数（信号安装函数是在可靠机制上的实现）。因此，Linux下的不可靠信号问题主要指的是信号可能丢失。

**"可靠信号"**

随着时间的发展，实践证明了有必要对信号的原始机制加以改进和扩充。所以，后来出现的各种Unix版本分别在这方面进行了研究，力图实现"可靠信号"。由于原来定义的信号已有许多应用，不好再做改动，最终只好又新增加了一些信号，并在一开始就把它们定义为可靠信号，这些信号支持排队，不会丢失。同时，信号的发送和安装也出现了新版本：信号发送函数sigqueue()及信号安装函数sigaction()。POSIX.4对可靠信号机制做了标准化。但是，POSIX只对可靠信号机制应具有的功能以及信号机制的对外接口做了标准化，对信号机制的实现没有作具体的规定。

信号值位于SIGRTMIN和SIGRTMAX之间的信号都是可靠信号，可靠信号克服了信号可能丢失的问题。Linux在支持新版本的信号安装函数sigation（）以及信号发送函数sigqueue()的同时，仍然支持早期的signal（）信号安装函数，支持信号发送函数kill()。

注：不要有这样的误解：由sigqueue()发送、sigaction安装的信号就是可靠的。事实上，可靠信号是指后来添加的新信号（信号值位于SIGRTMIN及SIGRTMAX之间）；不可靠信号是信号值小于SIGRTMIN的信号。信号的可靠与不可靠只与信号值有关，与信号的发送及安装函数无关。目前linux中的signal()是通过sigation()函数实现的，因此，即使通过signal（）安装的信号，在信号处理函数的结尾也不必再调用一次信号安装函数。同时，由signal()安装的实时信号支持排队，同样不会丢失。

对于目前linux的两个信号安装函数:signal()及sigaction()来说，它们都不能把SIGRTMIN以前的信号变成可靠信号（都不支持排队，仍有可能丢失，仍然是不可靠信号），而且对SIGRTMIN以后的信号都支持排队。这两个函数的最大区别在于，经过sigaction安装的信号都能传递信息给信号处理函数（对所有信号这一点都成立），而经过signal安装的信号却不能向信号处理函数传递信息。对于信号发送函数来说也是一样的。

### 2.2实时信号与非实时信号

早期Unix系统只定义了32种信号，Ret hat7.2支持64种信号，编号0-63(SIGRTMIN=31，SIGRTMAX=63)，将来可能进一步增加，这需要得到内核的支持。前32种信号已经有了预定义值，每个信号有了确定的用途及含义，并且每种信号都有各自的缺省动作。如按键盘的CTRL ^C时，会产生SIGINT信号，对该信号的默认反应就是进程终止。后32个信号表示实时信号，等同于前面阐述的可靠信号。这保证了发送的多个实时信号都被接收。实时信号是POSIX标准的一部分，可用于应用进程。

非实时信号都不支持排队，都是不可靠信号；实时信号都支持排队，都是可靠信号。

## 3、进程对信号的响应

进程可以通过三种方式来响应一个信号：（1）忽略信号，即对信号不做任何处理，其中，有两个信号不能忽略：SIGKILL及SIGSTOP；（2）捕捉信号。定义信号处理函数，当信号发生时，执行相应的处理函数；（3）执行缺省操作，Linux对每种信号都规定了默认操作，详细情况请参考[2]以及其它资料。注意，进程对实时信号的缺省反应是进程终止。

Linux究竟采用上述三种方式的哪一个来响应信号，取决于传递给相应API函数的参数。

## 4、信号的发送

发送信号的主要函数有：kill()、raise()、 sigqueue()、alarm()、setitimer()以及abort()。

1、kill()   
#include <sys/types.h>   
#include <signal.h>   
int kill(pid\_t pid,int signo)

|  |  |
| --- | --- |
| 参数pid的值 | 信号的接收进程 |
| pid>0 | 进程ID为pid的进程 |
| pid=0 | 同一个进程组的进程 |
| pid<0 pid!=-1 | 进程组ID为 -pid的所有进程 |
| pid=-1 | 除发送进程自身外，所有进程ID大于1的进程 |

Sinno是信号值，当为0时（即空信号），实际不发送任何信号，但照常进行错误检查，因此，可用于检查目标进程是否存在，以及当前进程是否具有向目标发送信号的权限（root权限的进程可以向任何进程发送信号，非root权限的进程只能向属于同一个session或者同一个用户的进程发送信号）。

Kill()最常用于pid>0时的信号发送，调用成功返回 0； 否则，返回 -1。注：对于pid<0时的情况，对于哪些进程将接受信号，各种版本说法不一，其实很简单，参阅内核源码kernal/signal.c即可，上表中的规则是参考red hat 7.2。

2、raise（）   
#include <signal.h>   
int raise(int signo)   
向进程本身发送信号，参数为即将发送的信号值。调用成功返回 0；否则，返回 -1。

3、sigqueue（）   
#include <sys/types.h>   
#include <signal.h>   
int sigqueue(pid\_t pid, int sig, const union sigval val)   
调用成功返回 0；否则，返回 -1。

sigqueue()是比较新的发送信号系统调用，主要是针对实时信号提出的（当然也支持前32种），支持信号带有参数，与函数sigaction()配合使用。

sigqueue的第一个参数是指定接收信号的进程ID，第二个参数确定即将发送的信号，第三个参数是一个联合数据结构union sigval，指定了信号传递的参数，即通常所说的4字节值。

|  |
| --- |
| typedef union sigval {  int sival\_int;  void \*sival\_ptr;  }sigval\_t; |

sigqueue()比kill()传递了更多的附加信息，但sigqueue()只能向一个进程发送信号，而不能发送信号给一个进程组。如果signo=0，将会执行错误检查，但实际上不发送任何信号，0值信号可用于检查pid的有效性以及当前进程是否有权限向目标进程发送信号。

在调用sigqueue时，sigval\_t指定的信息会拷贝到3参数信号处理函数（3参数信号处理函数指的是信号处理函数由sigaction安装，并设定了sa\_sigaction指针，稍后将阐述）的siginfo\_t结构中，这样信号处理函数就可以处理这些信息了。由于sigqueue系统调用支持发送带参数信号，所以比kill()系统调用的功能要灵活和强大得多。

注：sigqueue（）发送非实时信号时，第三个参数包含的信息仍然能够传递给信号处理函数； sigqueue（）发送非实时信号时，仍然不支持排队，即在信号处理函数执行过程中到来的所有相同信号，都被合并为一个信号。

4、alarm（）   
#include <unistd.h>   
unsigned int alarm(unsigned int seconds)   
专门为SIGALRM信号而设，在指定的时间seconds秒后，将向进程本身发送SIGALRM信号，又称为闹钟时间。进程调用alarm后，任何以前的alarm()调用都将无效。如果参数seconds为零，那么进程内将不再包含任何闹钟时间。   
返回值，如果调用alarm（）前，进程中已经设置了闹钟时间，则返回上一个闹钟时间的剩余时间，否则返回0。

5、setitimer（）   
#include <sys/time.h>   
int setitimer(int which, const struct itimerval \*value, struct itimerval \*ovalue));   
setitimer()比alarm功能强大，支持3种类型的定时器：

* ITIMER\_REAL： 设定绝对时间；经过指定的时间后，内核将发送SIGALRM信号给本进程；
* ITIMER\_VIRTUAL 设定程序执行时间；经过指定的时间后，内核将发送SIGVTALRM信号给本进程；
* ITIMER\_PROF 设定进程执行以及内核因本进程而消耗的时间和，经过指定的时间后，内核将发送ITIMER\_VIRTUAL信号给本进程；

Setitimer()第一个参数which指定定时器类型（上面三种之一）；第二个参数是结构itimerval的一个实例，结构itimerval形式见附录1。第三个参数可不做处理。

Setitimer()调用成功返回0，否则返回-1。

6、abort()   
#include <stdlib.h>   
void abort(void);

向进程发送SIGABORT信号，默认情况下进程会异常退出，当然可定义自己的信号处理函数。即使SIGABORT被进程设置为阻塞信号，调用abort()后，SIGABORT仍然能被进程接收。该函数无返回值。

## 5、信号的安装（设置信号关联动作）

如果进程要处理某一信号，那么就要在进程中安装该信号。安装信号主要用来确定信号值及进程针对该信号值的动作之间的映射关系，即进程将要处理哪个信号；该信号被传递给进程时，将执行何种操作。

linux主要有两个函数实现信号的安装：signal()、sigaction()。其中signal()在可靠信号系统调用的基础上实现, 是库函数。它只有两个参数，不支持信号传递信息，主要是用于前32种非实时信号的安装；而sigaction()是较新的函数（由两个系统调用实现：sys\_signal以及sys\_rt\_sigaction），有三个参数，支持信号传递信息，主要用来与 sigqueue() 系统调用配合使用，当然，sigaction()同样支持非实时信号的安装。sigaction()优于signal()主要体现在支持信号带有参数。

1、signal()   
#include <signal.h>   
void (\*signal(int signum, void (\*handler))(int)))(int);   
如果该函数原型不容易理解的话，可以参考下面的分解方式来理解：   
typedef void (\*sighandler\_t)(int)；   
sighandler\_t signal(int signum, sighandler\_t handler));   
第一个参数指定信号的值，第二个参数指定针对前面信号值的处理，可以忽略该信号（参数设为SIG\_IGN）；可以采用系统默认方式处理信号(参数设为SIG\_DFL)；也可以自己实现处理方式(参数指定一个函数地址)。   
如果signal()调用成功，返回最后一次为安装信号signum而调用signal()时的handler值；失败则返回SIG\_ERR。

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

sigaction函数用于改变进程接收到特定信号后的行为。该函数的第一个参数为信号的值，可以为除SIGKILL及SIGSTOP外的任何一个特定有效的信号（为这两个信号定义自己的处理函数，将导致信号安装错误）。第二个参数是指向结构sigaction的一个实例的指针，在结构sigaction的实例中，指定了对特定信号的处理，可以为空，进程会以缺省方式对信号处理；第三个参数oldact指向的对象用来保存原来对相应信号的处理，可指定oldact为NULL。如果把第二、第三个参数都设为NULL，那么该函数可用于检查信号的有效性。

第二个参数最为重要，其中包含了对指定信号的处理、信号所传递的信息、信号处理函数执行过程中应屏蔽掉哪些函数等等。

sigaction结构定义如下：

|  |
| --- |
| struct sigaction {  union{  \_\_sighandler\_t \_sa\_handler;  void (\*\_sa\_sigaction)(int,struct siginfo \*, void \*)；  }\_u  sigset\_t sa\_mask；  unsigned long sa\_flags；  void (\*sa\_restorer)(void)；  } |

其中，sa\_restorer，已过时，POSIX不支持它，不应再被使用。

1、联合数据结构中的两个元素\_sa\_handler以及\*\_sa\_sigaction指定信号关联函数，即用户指定的信号处理函数。除了可以是用户自定义的处理函数外，还可以为SIG\_DFL(采用缺省的处理方式)，也可以为SIG\_IGN（忽略信号）。

2、由\_sa\_handler指定的处理函数只有一个参数，即信号值，所以信号不能传递除信号值之外的任何信息；由\_sa\_sigaction是指定的信号处理函数带有三个参数，是为实时信号而设的（当然同样支持非实时信号），它指定一个3参数信号处理函数。第一个参数为信号值，第三个参数没有使用（posix没有规范使用该参数的标准），第二个参数是指向siginfo\_t结构的指针，结构中包含信号携带的数据值，参数所指向的结构如下：

|  |
| --- |
| siginfo\_t {  int si\_signo; /\* 信号值，对所有信号有意义\*/  int si\_errno; /\* errno值，对所有信号有意义\*/  int si\_code; /\* 信号产生的原因，对所有信号有意义\*/  union{ /\* 联合数据结构，不同成员适应不同信号 \*/  //确保分配足够大的存储空间  int \_pad[SI\_PAD\_SIZE];  //对SIGKILL有意义的结构  struct{  ...  }...  ... ...  ... ...  //对SIGILL, SIGFPE, SIGSEGV, SIGBUS有意义的结构  struct{  ...  }...  ... ...  }  } |

注：为了更便于阅读，在说明问题时常把该结构表示为附录2所表示的形式。

siginfo\_t结构中的联合数据成员确保该结构适应所有的信号，比如对于实时信号来说，则实际采用下面的结构形式：

|  |
| --- |
| typedef struct {  int si\_signo;  int si\_errno;  int si\_code;  union sigval si\_value;  } siginfo\_t; |

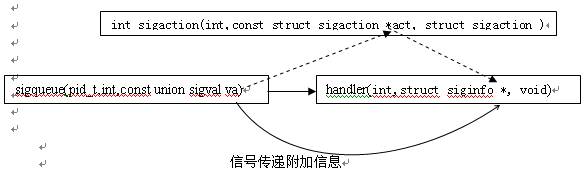
结构的第四个域同样为一个联合数据结构：

|  |
| --- |
| union sigval {  int sival\_int;  void \*sival\_ptr;  } |

采用联合数据结构，说明siginfo\_t结构中的si\_value要么持有一个4字节的整数值，要么持有一个指针，这就构成了与信号相关的数据。在信号的处理函数中，包含这样的信号相关数据指针，但没有规定具体如何对这些数据进行操作，操作方法应该由程序开发人员根据具体任务事先约定。

前面在讨论系统调用sigqueue发送信号时，sigqueue的第三个参数就是sigval联合数据结构，当调用sigqueue时，该数据结构中的数据就将拷贝到信号处理函数的第二个参数中。这样，在发送信号同时，就可以让信号传递一些附加信息。信号可以传递信息对程序开发是非常有意义的。

信号参数的传递过程可图示如下：



3、sa\_mask指定在信号处理程序执行过程中，哪些信号应当被阻塞。缺省情况下当前信号本身被阻塞，防止信号的嵌套发送，除非指定SA\_NODEFER或者SA\_NOMASK标志位。

注：请注意sa\_mask指定的信号阻塞的前提条件，是在由sigaction（）安装信号的处理函数执行过程中由sa\_mask指定的信号才被阻塞。

4、sa\_flags中包含了许多标志位，包括刚刚提到的SA\_NODEFER及SA\_NOMASK标志位。另一个比较重要的标志位是SA\_SIGINFO，当设定了该标志位时，表示信号附带的参数可以被传递到信号处理函数中，因此，应该为sigaction结构中的sa\_sigaction指定处理函数，而不应该为sa\_handler指定信号处理函数，否则，设置该标志变得毫无意义。即使为sa\_sigaction指定了信号处理函数，如果不设置SA\_SIGINFO，信号处理函数同样不能得到信号传递过来的数据，在信号处理函数中对这些信息的访问都将导致段错误（Segmentation fault）。

注：很多文献在阐述该标志位时都认为，如果设置了该标志位，就必须定义三参数信号处理函数。实际不是这样的，验证方法很简单：自己实现一个单一参数信号处理函数，并在程序中设置该标志位，可以察看程序的运行结果。实际上，可以把该标志位看成信号是否传递参数的开关，如果设置该位，则传递参数；否则，不传递参数。

## 6、信号集及信号集操作函数：

信号集被定义为一种数据类型：

|  |
| --- |
| typedef struct {  unsigned long sig[\_NSIG\_WORDS]；  } sigset\_t |

信号集用来描述信号的集合，linux所支持的所有信号可以全部或部分的出现在信号集中，主要与信号阻塞相关函数配合使用。下面是为信号集操作定义的相关函数：

|  |
| --- |
| #include <signal.h>  int sigemptyset(sigset\_t \*set)；  int sigfillset(sigset\_t \*set)；  int sigaddset(sigset\_t \*set, int signum)  int sigdelset(sigset\_t \*set, int signum)；  int sigismember(const sigset\_t \*set, int signum)；  sigemptyset(sigset\_t \*set)初始化由set指定的信号集，信号集里面的所有信号被清空；  sigfillset(sigset\_t \*set)调用该函数后，set指向的信号集中将包含linux支持的64种信号；  sigaddset(sigset\_t \*set, int signum)在set指向的信号集中加入signum信号；  sigdelset(sigset\_t \*set, int signum)在set指向的信号集中删除signum信号；  sigismember(const sigset\_t \*set, int signum)判定信号signum是否在set指向的信号集中。 |

## 7、信号阻塞与信号未决

每个进程都有一个用来描述哪些信号递送到进程时将被阻塞的信号集，该信号集中的所有信号在递送到进程后都将被阻塞。下面是与信号阻塞相关的几个函数：

|  |
| --- |
| #include <signal.h>  int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset))；  int sigpending(sigset\_t \*set));  int sigsuspend(const sigset\_t \*mask))； |

sigprocmask()函数能够根据参数how来实现对信号集的操作，操作主要有三种：

|  |  |
| --- | --- |
| 参数how | 进程当前信号集 |
| SIG\_BLOCK | 在进程当前阻塞信号集中添加set指向信号集中的信号 |
| SIG\_UNBLOCK | 如果进程阻塞信号集中包含set指向信号集中的信号，则解除对该信号的阻塞 |
| SIG\_SETMASK | 更新进程阻塞信号集为set指向的信号集 |

sigpending(sigset\_t \*set))获得当前已递送到进程，却被阻塞的所有信号，在set指向的信号集中返回结果。

sigsuspend(const sigset\_t \*mask))用于在接收到某个信号之前, 临时用mask替换进程的信号掩码, 并暂停进程执行，直到收到信号为止。sigsuspend 返回后将恢复调用之前的信号掩码。信号处理函数完成后，进程将继续执行。该系统调用始终返回-1，并将errno设置为EINTR。

#### 附录1

结构itimerval：

|  |
| --- |
| struct itimerval {  struct timeval it\_interval; /\* next value \*/  struct timeval it\_value; /\* current value \*/  };  struct timeval {  long tv\_sec; /\* seconds \*/  long tv\_usec; /\* microseconds \*/  }; |

#### 附录2

三参数信号处理函数中第二个参数的说明性描述：

|  |
| --- |
| siginfo\_t {  int si\_signo; /\* 信号值，对所有信号有意义\*/  int si\_errno; /\* errno值，对所有信号有意义\*/  int si\_code; /\* 信号产生的原因，对所有信号有意义\*/  pid\_t si\_pid; /\* 发送信号的进程ID,对kill(2),实时信号以及SIGCHLD有意义 \*/  uid\_t si\_uid; /\* 发送信号进程的真实用户ID，对kill(2),实时信号以及SIGCHLD有意义 \*/  int si\_status; /\* 退出状态，对SIGCHLD有意义\*/  clock\_t si\_utime; /\* 用户消耗的时间，对SIGCHLD有意义 \*/  clock\_t si\_stime; /\* 内核消耗的时间，对SIGCHLD有意义 \*/  sigval\_t si\_value; /\* 信号值，对所有实时有意义，是一个联合数据结构，  /\*可以为一个整数（由si\_int标示，也可以为一个指针，由si\_ptr标示）\*/    void \* si\_addr; /\* 触发fault的内存地址，对SIGILL,SIGFPE,SIGSEGV,SIGBUS 信号有意义\*/  int si\_band; /\* 对SIGPOLL信号有意义 \*/  int si\_fd; /\* 对SIGPOLL信号有意义 \*/  } |

实际上，除了前三个元素外，其他元素组织在一个联合结构中，在联合数据结构中，又根据不同的信号组织成不同的结构。注释中提到的对某种信号有意义指的是，在该信号的处理函数中可以访问这些域来获得与信号相关的有意义的信息，只不过特定信号只对特定信息感兴趣而已。

# 三．Linux环境进程间通信（二）：信号（下）

## 1、信号生命周期

从信号发送到信号处理函数的执行完毕

对于一个完整的信号生命周期(从信号发送到相应的处理函数执行完毕)来说，可以分为三个重要的阶段，这三个阶段由四个重要事件来刻画：信号诞生；信号在进程中注册完毕；信号在进程中的注销完毕；信号处理函数执行完毕。相邻两个事件的时间间隔构成信号生命周期的一个阶段。

http://www.ibm.com/developerworks/cn/linux/l-ipc/part2/3.gif

下面阐述四个事件的实际意义：

1. 信号"诞生"。信号的诞生指的是触发信号的事件发生（如检测到硬件异常、定时器超时以及调用信号发送函数kill()或sigqueue()等）。
2. 信号在目标进程中"注册"；进程的task\_struct结构中有关于本进程中未决信号的数据成员：

|  |
| --- |
| struct sigpending pending：  struct sigpending{  struct sigqueue \*head, \*\*tail;  sigset\_t signal;  }; |

第三个成员是进程中所有未决信号集，第一、第二个成员分别指向一个sigqueue类型的结构链（称之为"未决信号信息链"）的首尾，信息链中的每个sigqueue结构刻画一个特定信号所携带的信息，并指向下一个sigqueue结构:

|  |
| --- |
| struct sigqueue{  struct sigqueue \*next;  siginfo\_t info;  } |

信号在进程中注册指的就是信号值加入到进程的未决信号集中（sigpending结构的第二个成员sigset\_t signal），并且信号所携带的信息被保留到未决信号信息链的某个sigqueue结构中。只要信号在进程的未决信号集中，表明进程已经知道这些信号的存在，但还没来得及处理，或者该信号被进程阻塞。

**注：**   
当一个实时信号发送给一个进程时，不管该信号是否已经在进程中注册，都会被再注册一次，因此，信号不会丢失，因此，实时信号又叫做"可靠信号"。这意味着同一个实时信号可以在同一个进程的未决信号信息链中占有多个sigqueue结构（进程每收到一个实时信号，都会为它分配一个结构来登记该信号信息，并把该结构添加在未决信号链尾，即所有诞生的实时信号都会在目标进程中注册）；   
当一个非实时信号发送给一个进程时，如果该信号已经在进程中注册，则该信号将被丢弃，造成信号丢失。因此，非实时信号又叫做"不可靠信号"。这意味着同一个非实时信号在进程的未决信号信息链中，至多占有一个sigqueue结构（一个非实时信号诞生后，（1）、如果发现相同的信号已经在目标结构中注册，则不再注册，对于进程来说，相当于不知道本次信号发生，信号丢失；（2）、如果进程的未决信号中没有相同信号，则在进程中注册自己）。

1. 信号在进程中的注销。在目标进程执行过程中，会检测是否有信号等待处理（每次从系统空间返回到用户空间时都做这样的检查）。如果存在未决信号等待处理且该信号没有被进程阻塞，则在运行相应的信号处理函数前，进程会把信号在未决信号链中占有的结构卸掉。是否将信号从进程未决信号集中删除对于实时与非实时信号是不同的。对于非实时信号来说，由于在未决信号信息链中最多只占用一个sigqueue结构，因此该结构被释放后，应该把信号在进程未决信号集中删除（信号注销完毕）；而对于实时信号来说，可能在未决信号信息链中占用多个sigqueue结构，因此应该针对占用sigqueue结构的数目区别对待：如果只占用一个sigqueue结构（进程只收到该信号一次），则应该把信号在进程的未决信号集中删除（信号注销完毕）。否则，不应该在进程的未决信号集中删除该信号（信号注销完毕）。   
   进程在执行信号相应处理函数之前，首先要把信号在进程中注销。
2. 信号生命终止。进程注销信号后，立即执行相应的信号处理函数，执行完毕后，信号的本次发送对进程的影响彻底结束。

**注：**   
1）信号注册与否，与发送信号的函数（如kill()或sigqueue()等）以及信号安装函数（signal()及sigaction()）无关，只与信号值有关（信号值小于SIGRTMIN的信号最多只注册一次，信号值在SIGRTMIN及SIGRTMAX之间的信号，只要被进程接收到就被注册）。   
2）在信号被注销到相应的信号处理函数执行完毕这段时间内，如果进程又收到同一信号多次，则对实时信号来说，每一次都会在进程中注册；而对于非实时信号来说，无论收到多少次信号，都会视为只收到一个信号，只在进程中注册一次。

## 2、信号编程注意事项

### 2.1 防止不该丢失的信号丢失

如果对八中所提到的信号生命周期理解深刻的话，很容易知道信号会不会丢失，以及在哪里丢失。

### 2.2 程序的可移植性

考虑到程序的可移植性，应该尽量采用POSIX信号函数，POSIX信号函数主要分为两类：

* + POSIX 1003.1信号函数： Kill()、sigaction()、sigaddset()、sigdelset()、sigemptyset()、sigfillset()、sigismember()、sigpending()、sigprocmask()、sigsuspend()。
  + POSIX 1003.1b信号函数。POSIX 1003.1b在信号的实时性方面对POSIX 1003.1做了扩展，包括以下三个函数： sigqueue()、sigtimedwait()、sigwaitinfo()。其中，sigqueue主要针对信号发送，而sigtimedwait及sigwaitinfo()主要用于取代sigsuspend()函数，后面有相应实例。

|  |
| --- |
| #include <signal.h>  int sigwaitinfo(sigset\_t \*set, siginfo\_t \*info). |

该函数与sigsuspend()类似，阻塞一个进程直到特定信号发生，但信号到来时不执行信号处理函数，而是返回信号值。因此为了避免执行相应的信号处理函数，必须在调用该函数前，使进程屏蔽掉set指向的信号，因此调用该函数的典型代码是：

|  |
| --- |
| sigset\_t newmask;  int rcvd\_sig;  siginfo\_t info;  sigemptyset(&newmask);  sigaddset(&newmask, SIGRTMIN);  sigprocmask(SIG\_BLOCK, &newmask, NULL);  rcvd\_sig = sigwaitinfo(&newmask, &info)  if (rcvd\_sig == -1) {  ..  } |

调用成功返回信号值，否则返回-1。sigtimedwait()功能相似，只不过增加了一个进程等待的时间。

### 2.3 程序的稳定性

为了增强程序的稳定性，在信号处理函数中应使用可重入函数。

信号处理程序中应当使用可再入（可重入）函数（注：所谓可重入函数是指一个可以被多个任务调用的过程，任务在调用时不必担心数据是否会出错）。因为进程在收到信号后，就将跳转到信号处理函数去接着执行。如果信号处理函数中使用了不可重入函数，那么信号处理函数可能会修改原来进程中不应该被修改的数据，这样进程从信号处理函数中返回接着执行时，可能会出现不可预料的后果。不可再入函数在信号处理函数中被视为不安全函数。

满足下列条件的函数多数是不可再入的：（1）使用静态的数据结构，如getlogin()，gmtime()，getgrgid()，getgrnam()，getpwuid()以及getpwnam()等等；（2）函数实现时，调用了malloc（）或者free()函数；（3）实现时使用了标准I/O函数的。The Open Group视下列函数为可再入的：

\_exit（）、access（）、alarm（）、cfgetispeed（）、cfgetospeed（）、cfsetispeed（）、cfsetospeed（）、chdir（）、chmod（）、chown（）、close（）、creat（）、dup（）、dup2（）、execle（）、execve（）、fcntl（）、fork（）、fpathconf（）、fstat（）、fsync（）、getegid（）、 geteuid（）、getgid（）、getgroups（）、getpgrp（）、getpid（）、getppid（）、getuid（）、kill（）、link（）、lseek（）、mkdir（）、mkfifo（）、 open（）、pathconf（）、pause（）、pipe（）、raise（）、read（）、rename（）、rmdir（）、setgid（）、setpgid（）、setsid（）、setuid（）、 sigaction（）、sigaddset（）、sigdelset（）、sigemptyset（）、sigfillset（）、sigismember（）、signal（）、sigpending（）、sigprocmask（）、sigsuspend（）、sleep（）、stat（）、sysconf（）、tcdrain（）、tcflow（）、tcflush（）、tcgetattr（）、tcgetpgrp（）、tcsendbreak（）、tcsetattr（）、tcsetpgrp（）、time（）、times（）、 umask（）、uname（）、unlink（）、utime（）、wait（）、waitpid（）、write（）。

即使信号处理函数使用的都是"安全函数"，同样要注意进入处理函数时，首先要保存errno的值，结束时，再恢复原值。因为，信号处理过程中，errno值随时可能被改变。另外，longjmp()以及siglongjmp()没有被列为可再入函数，因为不能保证紧接着两个函数的其它调用是安全的。

## 3、深入浅出：信号应用实例

linux下的信号应用并没有想象的那么恐怖，程序员所要做的最多只有三件事情：

1. 安装信号（推荐使用sigaction()）；
2. 实现三参数信号处理函数，handler(int signal,struct siginfo \*info, void \*)；
3. 发送信号，推荐使用sigqueue()。

实际上，对有些信号来说，只要安装信号就足够了（信号处理方式采用缺省或忽略）。其他可能要做的无非是与信号集相关的几种操作。

### 实例一：信号发送及处理

实现一个信号接收程序sigreceive（其中信号安装由sigaction（））。

|  |
| --- |
| #include <signal.h>  #include <sys/types.h>  #include <unistd.h>  void new\_op(int,siginfo\_t\*,void\*);  int main(int argc,char\*\*argv)  {  struct sigaction act;  int sig;  sig=atoi(argv[1]);    sigemptyset(&act.sa\_mask);  act.sa\_flags=SA\_SIGINFO;  act.sa\_sigaction=new\_op;    if(sigaction(sig,&act,NULL) < 0)  {  printf("install sigal error\n");  }    while(1)  {  sleep(2);  printf("wait for the signal\n");  }  }  void new\_op(int signum,siginfo\_t \*info,void \*myact)  {  printf("receive signal %d", signum);  sleep(5);  } |

说明，命令行参数为信号值，后台运行sigreceive signo &，可获得该进程的ID，假设为pid，然后再另一终端上运行kill -s signo pid验证信号的发送接收及处理。同时，可验证信号的排队问题。   
**注：**可以用sigqueue实现一个命令行信号发送程序sigqueuesend，见 [附录1](http://www.ibm.com/developerworks/cn/linux/l-ipc/part2/index2.html#5)。

### 实例二：信号传递附加信息

主要包括两个实例：

1. 向进程本身发送信号，并传递指针参数；

|  |
| --- |
| #include <signal.h>  #include <sys/types.h>  #include <unistd.h>  void new\_op(int,siginfo\_t\*,void\*);  int main(int argc,char\*\*argv)  {  struct sigaction act;  union sigval mysigval;  int i;  int sig;  pid\_t pid;  char data[10];  memset(data,0,sizeof(data));  for(i=0;i < 5;i++)  data[i]='2';  mysigval.sival\_ptr=data;    sig=atoi(argv[1]);  pid=getpid();    sigemptyset(&act.sa\_mask);  act.sa\_sigaction=new\_op;//三参数信号处理函数  act.sa\_flags=SA\_SIGINFO;//信息传递开关  if(sigaction(sig,&act,NULL) < 0)  {  printf("install sigal error\n");  }  while(1)  {  sleep(2);  printf("wait for the signal\n");  sigqueue(pid,sig,mysigval);//向本进程发送信号，并传递附加信息  }  }  void new\_op(int signum,siginfo\_t \*info,void \*myact)//三参数信号处理函数的实现  {  int i;  for(i=0;i<10;i++)  {  printf("%c\n ",(\*( (char\*)((\*info).si\_ptr)+i)));  }  printf("handle signal %d over;",signum);  } |

这个例子中，信号实现了附加信息的传递，信号究竟如何对这些信息进行处理则取决于具体的应用。

1. 2、 不同进程间传递整型参数：把1中的信号发送和接收放在两个程序中，并且在发送过程中传递整型参数。   
   信号接收程序：

|  |
| --- |
| #include <signal.h>  #include <sys/types.h>  #include <unistd.h>  void new\_op(int,siginfo\_t\*,void\*);  int main(int argc,char\*\*argv)  {  struct sigaction act;  int sig;  pid\_t pid;    pid=getpid();  sig=atoi(argv[1]);    sigemptyset(&act.sa\_mask);  act.sa\_sigaction=new\_op;  act.sa\_flags=SA\_SIGINFO;  if(sigaction(sig,&act,NULL)<0)  {  printf("install sigal error\n");  }  while(1)  {  sleep(2);  printf("wait for the signal\n");  }  }  void new\_op(int signum,siginfo\_t \*info,void \*myact)  {  printf("the int value is %d \n",info->si\_int);  } |

信号发送程序：命令行第二个参数为信号值，第三个参数为接收进程ID。

|  |
| --- |
| #include <signal.h>  #include <sys/time.h>  #include <unistd.h>  #include <sys/types.h>  main(int argc,char\*\*argv)  {  pid\_t pid;  int signum;  union sigval mysigval;  signum=atoi(argv[1]);  pid=(pid\_t)atoi(argv[2]);  mysigval.sival\_int=8;//不代表具体含义，只用于说明问题  if(sigqueue(pid,signum,mysigval)==-1)  printf("send error\n");  sleep(2);  } |

**注：**实例2的两个例子侧重点在于用信号来传递信息，目前关于在linux下通过信号传递信息的实例非常少，倒是Unix下有一些，但传递的基本上都是关于传递一个整数，传递指针的我还没看到。我一直没有实现不同进程间的指针传递（实际上更有意义），也许在实现方法上存在问题吧，请实现者email我。

### 实例三：信号阻塞及信号集操作

|  |
| --- |
| #include "signal.h"  #include "unistd.h"  static void my\_op(int);  main()  {  sigset\_t new\_mask,old\_mask,pending\_mask;  struct sigaction act;  sigemptyset(&act.sa\_mask);  act.sa\_flags=SA\_SIGINFO;  act.sa\_sigaction=(void\*)my\_op;  if(sigaction(SIGRTMIN+10,&act,NULL))  printf("install signal SIGRTMIN+10 error\n");  sigemptyset(&new\_mask);  sigaddset(&new\_mask,SIGRTMIN+10);  if(sigprocmask(SIG\_BLOCK, &new\_mask,&old\_mask))  printf("block signal SIGRTMIN+10 error\n");  sleep(10);  printf("now begin to get pending mask and unblock SIGRTMIN+10\n");  if(sigpending(&pending\_mask)<0)  printf("get pending mask error\n");  if(sigismember(&pending\_mask,SIGRTMIN+10))  printf("signal SIGRTMIN+10 is pending\n");  if(sigprocmask(SIG\_SETMASK,&old\_mask,NULL)<0)  printf("unblock signal error\n");  printf("signal unblocked\n");  sleep(10);  }  static void my\_op(int signum)  {  printf("receive signal %d \n",signum);  } |

编译该程序，并以后台方式运行。在另一终端向该进程发送信号(运行kill -s 42 pid，SIGRTMIN+10为42)，查看结果可以看出几个关键函数的运行机制，信号集相关操作比较简单。

**注：**在上面几个实例中，使用了printf()函数，只是作为诊断工具，pringf()函数是不可重入的，不应在信号处理函数中使用。

## 结束语：

系统地对linux信号机制进行分析、总结使我受益匪浅！感谢王小乐等网友的支持！   
Comments and suggestions are greatly welcome!

#### 附录1：

用sigqueue实现的命令行信号发送程序sigqueuesend，命令行第二个参数是发送的信号值，第三个参数是接收该信号的进程ID，可以配合实例一使用：

|  |
| --- |
| #include <signal.h>  #include <sys/types.h>  #include <unistd.h>  int main(int argc,char\*\*argv)  {  pid\_t pid;  int sig;  sig=atoi(argv[1]);  pid=atoi(argv[2]);  sigqueue(pid,sig,NULL);  sleep(2);  } |

# 四．Linux环境进程间通信（三）：消息队列

消息队列（也叫做报文队列）能够克服早期unix通信机制的一些缺点。作为早期unix通信机制之一的信号能够传送的信息量有限，后来虽然POSIX 1003.1b在信号的实时性方面作了拓广，使得信号在传递信息量方面有了相当程度的改进，但是信号这种通信方式更像"即时"的通信方式，它要求接受信号的进程在某个时间范围内对信号做出反应，因此该信号最多在接受信号进程的生命周期内才有意义，信号所传递的信息是接近于随进程持续的概念（process-persistent），见 [附录 1](http://www.ibm.com/developerworks/cn/linux/l-ipc/part3/index.html#listing1)；管道及有名管道及有名管道则是典型的随进程持续IPC，并且，只能传送无格式的字节流无疑会给应用程序开发带来不便，另外，它的缓冲区大小也受到限制。

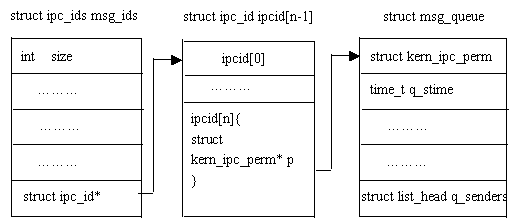
消息队列就是一个消息的链表。可以把消息看作一个记录，具有特定的格式以及特定的优先级。对消息队列有写权限的进程可以向中按照一定的规则添加新消息；对消息队列有读权限的进程则可以从消息队列中读走消息。消息队列是随内核持续的（参见 [附录 1](http://www.ibm.com/developerworks/cn/linux/l-ipc/part3/index.html#listing1)）。

目前主要有两种类型的消息队列：POSIX消息队列以及系统V消息队列，系统V消息队列目前被大量使用。考虑到程序的可移植性，新开发的应用程序应尽量使用POSIX消息队列。

在本系列专题的序（深刻理解Linux进程间通信（IPC））中，提到对于消息队列、信号灯、以及共享内存区来说，有两个实现版本：POSIX的以及系统V的。Linux内核（内核2.4.18）支持POSIX信号灯、POSIX共享内存区以及POSIX消息队列，但对于主流Linux发行版本之一redhad8.0（内核2.4.18），还没有提供对POSIX进程间通信API的支持，不过应该只是时间上的事。

因此，本文将主要介绍系统V消息队列及其相应API。 **在没有声明的情况下，以下讨论中指的都是系统V消息队列。**

## 1、消息队列基本概念

1. 系统V消息队列是随内核持续的，只有在内核重起或者显示删除一个消息队列时，该消息队列才会真正被删除。因此系统中记录消息队列的数据结构（struct ipc\_ids msg\_ids）位于内核中，系统中的所有消息队列都可以在结构msg\_ids中找到访问入口。
2. 消息队列就是一个消息的链表。每个消息队列都有一个队列头，用结构struct msg\_queue来描述（参见 [附录 2](http://www.ibm.com/developerworks/cn/linux/l-ipc/part3/index.html#listing2)）。队列头中包含了该消息队列的大量信息，包括消息队列键值、用户ID、组ID、消息队列中消息数目等等，甚至记录了最近对消息队列读写进程的ID。读者可以访问这些信息，也可以设置其中的某些信息。
3. 下图说明了内核与消息队列是怎样建立起联系的：   
   其中：struct ipc\_ids msg\_ids是内核中记录消息队列的全局数据结构；struct msg\_queue是每个消息队列的队列头。   
   

从上图可以看出，全局数据结构 struct ipc\_ids msg\_ids 可以访问到每个消息队列头的第一个成员：struct kern\_ipc\_perm；而每个struct kern\_ipc\_perm能够与具体的消息队列对应起来是因为在该结构中，有一个key\_t类型成员key，而key则唯一确定一个消息队列。kern\_ipc\_perm结构如下：

|  |
| --- |
| struct kern\_ipc\_perm{ //内核中记录消息队列的全局数据结构msg\_ids能够访问到该结构；  key\_t key; //该键值则唯一对应一个消息队列  uid\_t uid;  gid\_t gid;  uid\_t cuid;  gid\_t cgid;  mode\_t mode;  unsigned long seq;  } |

## 2、操作消息队列

对消息队列的操作无非有下面三种类型：

1、 打开或创建消息队列   
消息队列的内核持续性要求每个消息队列都在系统范围内对应唯一的键值，所以，要获得一个消息队列的描述字，只需提供该消息队列的键值即可；

注：消息队列描述字是由在系统范围内唯一的键值生成的，而键值可以看作对应系统内的一条路经。

2、 读写操作

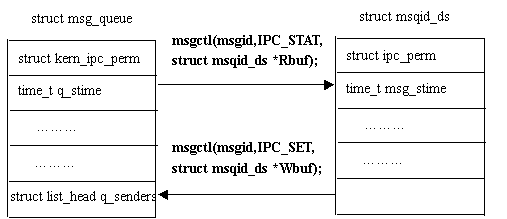
消息读写操作非常简单，对开发人员来说，每个消息都类似如下的数据结构：

|  |
| --- |
| struct msgbuf{  long mtype;  char mtext[1];  }; |

mtype成员代表消息类型，从消息队列中读取消息的一个重要依据就是消息的类型；mtext是消息内容，当然长度不一定为1。因此，对于发送消息来说，首先预置一个msgbuf缓冲区并写入消息类型和内容，调用相应的发送函数即可；对读取消息来说，首先分配这样一个msgbuf缓冲区，然后把消息读入该缓冲区即可。

3、 获得或设置消息队列属性：

消息队列的信息基本上都保存在消息队列头中，因此，可以分配一个类似于消息队列头的结构(struct msqid\_ds，见 [附录 2](http://www.ibm.com/developerworks/cn/linux/l-ipc/part3/index.html#listing2))，来返回消息队列的属性；同样可以设置该数据结构。



### 消息队列API

#### 1、文件名到键值

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  key\_t ftok (char\*pathname, char proj)； |

它返回与路径pathname相对应的一个键值。该函数不直接对消息队列操作，但在调用ipc(MSGGET,…)或msgget()来获得消息队列描述字前，往往要调用该函数。典型的调用代码是：

|  |
| --- |
| key=ftok(path\_ptr, 'a');  ipc\_id=ipc(MSGGET, (int)key, flags,0,NULL,0);  … |

#### 2、通信的三种方式

linux为操作系统V进程间通信的三种方式**（消息队列、信号灯、共享内存区）提供了一个统一的用户界面：**   
**int ipc**(unsigned int **call**, int **first**, int **second**, int **third**, void \* **ptr**, long **fifth**);

第一个参数指明对IPC对象的操作方式，对消息队列而言共有四种操作：MSGSND、MSGRCV、MSGGET以及MSGCTL，分别代表向消息队列发送消息、从消息队列读取消息、打开或创建消息队列、控制消息队列；first参数代表唯一的IPC对象；下面将介绍四种操作。

* **int ipc**( **MSGGET, int**first, **int**second, **int**third, **void**\*ptr, **long**fifth);   
  与该操作对应的系统V调用为：int msgget( (key\_t)first，second)。
* **int ipc**( **MSGCTL, int**first, **int**second, **int**third, **void**\*ptr, **long**fifth)   
  与该操作对应的系统V调用为：int msgctl( first，second, (struct msqid\_ds\*) ptr)。
* **int ipc**( **MSGSND, int**first, **int**second, **int**third, **void**\*ptr, **long**fifth);   
  与该操作对应的系统V调用为：int msgsnd( first, (struct msgbuf\*)ptr, second, third)。
* **int ipc**( **MSGRCV, int**first, **int**second, **int**third, **void**\*ptr, **long**fifth);   
  与该操作对应的系统V调用为：int msgrcv( first，(struct msgbuf\*)ptr, second, fifth,third)，

注：本人不主张采用系统调用ipc()，而更倾向于采用系统V或者POSIX进程间通信API。原因如下：

* 虽然该系统调用提供了统一的用户界面，但正是由于这个特性，它的参数几乎不能给出特定的实际意义（如以first、second来命名参数），在一定程度上造成开发不便。
* 正如ipc手册所说的：ipc()是linux所特有的，编写程序时应注意程序的移植性问题；
* 该系统调用的实现不过是把系统V IPC函数进行了封装，没有任何效率上的优势；
* 系统V在IPC方面的API数量不多，形式也较简洁。

#### 3.系统V消息队列API

系统V消息队列API共有四个，使用时需要包括几个头文件：

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h> |

**1）int msgget(key\_t key, int msgflg)**

参数key是一个键值，由ftok获得；msgflg参数是一些标志位。该调用返回与健值key相对应的消息队列描述字。

在以下两种情况下，该调用将创建一个新的消息队列：

* 如果没有消息队列与健值key相对应，并且msgflg中包含了IPC\_CREAT标志位；
* key参数为IPC\_PRIVATE；

参数msgflg可以为以下：IPC\_CREAT、IPC\_EXCL、IPC\_NOWAIT或三者的或结果。

**调用返回：**成功返回消息队列描述字，否则返回-1。

注：参数key设置成常数IPC\_PRIVATE并不意味着其他进程不能访问该消息队列，只意味着即将创建新的消息队列。

**2）int msgrcv(int msqid, struct msgbuf \*msgp, int msgsz, long msgtyp, int msgflg);**   
该系统调用从msgid代表的消息队列中读取一个消息，并把消息存储在msgp指向的msgbuf结构中。

msqid为消息队列描述字；消息返回后存储在msgp指向的地址，msgsz指定msgbuf的mtext成员的长度（即消息内容的长度），msgtyp为请求读取的消息类型；读消息标志msgflg可以为以下几个常值的或：

* IPC\_NOWAIT 如果没有满足条件的消息，调用立即返回，此时，errno=ENOMSG
* IPC\_EXCEPT 与msgtyp>0配合使用，返回队列中第一个类型不为msgtyp的消息
* IPC\_NOERROR 如果队列中满足条件的消息内容大于所请求的msgsz字节，则把该消息截断，截断部分将丢失。

msgrcv手册中详细给出了消息类型取不同值时(>0; <0; =0)，调用将返回消息队列中的哪个消息。

msgrcv()解除阻塞的条件有三个：

1. 消息队列中有了满足条件的消息；
2. msqid代表的消息队列被删除；
3. 调用msgrcv（）的进程被信号中断；

**调用返回：**成功返回读出消息的实际字节数，否则返回-1。

**3）int msgsnd(int msqid, struct msgbuf \*msgp, int msgsz, int msgflg);**   
向msgid代表的消息队列发送一个消息，即将发送的消息存储在msgp指向的msgbuf结构中，消息的大小由msgze指定。

对发送消息来说，有意义的msgflg标志为IPC\_NOWAIT，指明在消息队列没有足够空间容纳要发送的消息时，msgsnd是否等待。造成msgsnd()等待的条件有两种：

* 当前消息的大小与当前消息队列中的字节数之和超过了消息队列的总容量；
* 当前消息队列的消息数（单位"个"）不小于消息队列的总容量（单位"字节数"），此时，虽然消息队列中的消息数目很多，但基本上都只有一个字节。

msgsnd()解除阻塞的条件有三个：

1. 不满足上述两个条件，即消息队列中有容纳该消息的空间；
2. msqid代表的消息队列被删除；
3. 调用msgsnd（）的进程被信号中断；

**调用返回：**成功返回0，否则返回-1。

**4）int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);**   
该系统调用对由msqid标识的消息队列执行cmd操作，共有三种cmd操作：IPC\_STAT、IPC\_SET 、IPC\_RMID。

1. IPC\_STAT：该命令用来获取消息队列信息，返回的信息存贮在buf指向的msqid结构中；
2. IPC\_SET：该命令用来设置消息队列的属性，要设置的属性存储在buf指向的msqid结构中；可设置属性包括：msg\_perm.uid、msg\_perm.gid、msg\_perm.mode以及msg\_qbytes，同时，也影响msg\_ctime成员。
3. IPC\_RMID：删除msqid标识的消息队列；

**调用返回：**成功返回0，否则返回-1。

## 3、消息队列的限制

每个消息队列的容量（所能容纳的字节数）都有限制，该值因系统不同而不同。在后面的应用实例中，输出了redhat 8.0的限制，结果参见 [附录 3](http://www.ibm.com/developerworks/cn/linux/l-ipc/part3/index.html#listing3)。

另一个限制是每个消息队列所能容纳的最大消息数：在redhad 8.0中，该限制是受消息队列容量制约的：消息个数要小于消息队列的容量（字节数）。

注：上述两个限制是针对每个消息队列而言的，系统对消息队列的限制还有系统范围内的最大消息队列个数，以及整个系统范围内的最大消息数。一般来说，实际开发过程中不会超过这个限制。

## 4、消息队列应用实例

消息队列应用相对较简单，下面实例基本上覆盖了对消息队列的所有操作，同时，程序输出结果有助于加深对前面所讲的某些规则及消息队列限制的理解。

|  |
| --- |
| #include <sys/types.h>  #include <sys/msg.h>  #include <unistd.h>  void msg\_stat(int,struct msqid\_ds );  main()  {  int gflags,sflags,rflags;  key\_t key;  int msgid;  int reval;  struct msgsbuf{  int mtype;  char mtext[1];  }msg\_sbuf;  struct msgmbuf  {  int mtype;  char mtext[10];  }msg\_rbuf;  struct msqid\_ds msg\_ginfo,msg\_sinfo;  char\* msgpath="/unix/msgqueue";  key=ftok(msgpath,'a');  gflags=IPC\_CREAT|IPC\_EXCL;  msgid=msgget(key,gflags|00666);  if(msgid==-1)  {  printf("msg create error\n");  return;  }  //创建一个消息队列后，输出消息队列缺省属性  msg\_stat(msgid,msg\_ginfo);  sflags=IPC\_NOWAIT;  msg\_sbuf.mtype=10;  msg\_sbuf.mtext[0]='a';  reval=msgsnd(msgid,&msg\_sbuf,sizeof(msg\_sbuf.mtext),sflags);  if(reval==-1)  {  printf("message send error\n");  }  //发送一个消息后，输出消息队列属性  msg\_stat(msgid,msg\_ginfo);  rflags=IPC\_NOWAIT|MSG\_NOERROR;  reval=msgrcv(msgid,&msg\_rbuf,4,10,rflags);  if(reval==-1)  printf("read msg error\n");  else  printf("read from msg queue %d bytes\n",reval);  //从消息队列中读出消息后，输出消息队列属性  msg\_stat(msgid,msg\_ginfo);  msg\_sinfo.msg\_perm.uid=8;//just a try  msg\_sinfo.msg\_perm.gid=8;//  msg\_sinfo.msg\_qbytes=16388;  //此处验证超级用户可以更改消息队列的缺省msg\_qbytes  //注意这里设置的值大于缺省值  reval=msgctl(msgid,IPC\_SET,&msg\_sinfo);  if(reval==-1)  {  printf("msg set info error\n");  return;  }  msg\_stat(msgid,msg\_ginfo);  //验证设置消息队列属性  reval=msgctl(msgid,IPC\_RMID,NULL);//删除消息队列  if(reval==-1)  {  printf("unlink msg queue error\n");  return;  }  }  void msg\_stat(int msgid,struct msqid\_ds msg\_info)  {  int reval;  sleep(1);//只是为了后面输出时间的方便  reval=msgctl(msgid,IPC\_STAT,&msg\_info);  if(reval==-1)  {  printf("get msg info error\n");  return;  }  printf("\n");  printf("current number of bytes on queue is %d\n",msg\_info.msg\_cbytes);  printf("number of messages in queue is %d\n",msg\_info.msg\_qnum);  printf("max number of bytes on queue is %d\n",msg\_info.msg\_qbytes);  //每个消息队列的容量（字节数）都有限制MSGMNB，值的大小因系统而异。在创建新的消息队列时，//msg\_qbytes的缺省值就是MSGMNB  printf("pid of last msgsnd is %d\n",msg\_info.msg\_lspid);  printf("pid of last msgrcv is %d\n",msg\_info.msg\_lrpid);  printf("last msgsnd time is %s", ctime(&(msg\_info.msg\_stime)));  printf("last msgrcv time is %s", ctime(&(msg\_info.msg\_rtime)));  printf("last change time is %s", ctime(&(msg\_info.msg\_ctime)));  printf("msg uid is %d\n",msg\_info.msg\_perm.uid);  printf("msg gid is %d\n",msg\_info.msg\_perm.gid);  } |

程序输出结果见 [附录 3](http://www.ibm.com/developerworks/cn/linux/l-ipc/part3/index.html#listing3)。

## 小结：

消息队列与管道以及有名管道相比，具有更大的灵活性，首先，它提供有格式字节流，有利于减少开发人员的工作量；其次，消息具有类型，在实际应用中，可作为优先级使用。这两点是管道以及有名管道所不能比的。同样，消息队列可以在几个进程间复用，而不管这几个进程是否具有亲缘关系，这一点与有名管道很相似；但消息队列是随内核持续的，与有名管道（随进程持续）相比，生命力更强，应用空间更大。

#### 附录 1

在参考文献[1]中，给出了IPC随进程持续、随内核持续以及随文件系统持续的定义：

1. 随进程持续：IPC一直存在到打开IPC对象的最后一个进程关闭该对象为止。如管道和有名管道；
2. 随内核持续：IPC一直持续到内核重新自举或者显示删除该对象为止。如消息队列、信号灯以及共享内存等；
3. 随文件系统持续：IPC一直持续到显示删除该对象为止。

#### 附录 2 结构msg\_queue用来描述消息队列头，存在于系统空间：

|  |
| --- |
| struct msg\_queue {  struct kern\_ipc\_perm q\_perm;  time\_t q\_stime; /\* last msgsnd time \*/  time\_t q\_rtime; /\* last msgrcv time \*/  time\_t q\_ctime; /\* last change time \*/  unsigned long q\_cbytes; /\* current number of bytes on queue \*/  unsigned long q\_qnum; /\* number of messages in queue \*/  unsigned long q\_qbytes; /\* max number of bytes on queue \*/  pid\_t q\_lspid; /\* pid of last msgsnd \*/  pid\_t q\_lrpid; /\* last receive pid \*/  struct list\_head q\_messages;  struct list\_head q\_receivers;  struct list\_head q\_senders;  }; |

结构msqid\_ds用来设置或返回消息队列的信息，存在于用户空间；

|  |
| --- |
| struct msqid\_ds {  struct ipc\_perm msg\_perm;  struct msg \*msg\_first; /\* first message on queue,unused \*/  struct msg \*msg\_last; /\* last message in queue,unused \*/  \_\_kernel\_time\_t msg\_stime; /\* last msgsnd time \*/  \_\_kernel\_time\_t msg\_rtime; /\* last msgrcv time \*/  \_\_kernel\_time\_t msg\_ctime; /\* last change time \*/  unsigned long msg\_lcbytes; /\* Reuse junk fields for 32 bit \*/  unsigned long msg\_lqbytes; /\* ditto \*/  unsigned short msg\_cbytes; /\* current number of bytes on queue \*/  unsigned short msg\_qnum; /\* number of messages in queue \*/  unsigned short msg\_qbytes; /\* max number of bytes on queue \*/  \_\_kernel\_ipc\_pid\_t msg\_lspid; /\* pid of last msgsnd \*/  \_\_kernel\_ipc\_pid\_t msg\_lrpid; /\* last receive pid \*/  }; |

//可以看出上述两个结构很相似。

#### 附录 3

**消息队列实例输出结果：**

|  |
| --- |
| current number of bytes on queue is 0  number of messages in queue is 0  max number of bytes on queue is 16384  pid of last msgsnd is 0  pid of last msgrcv is 0  last msgsnd time is Thu Jan 1 08:00:00 1970  last msgrcv time is Thu Jan 1 08:00:00 1970  last change time is Sun Dec 29 18:28:20 2002  msg uid is 0  msg gid is 0  //上面刚刚创建一个新消息队列时的输出  current number of bytes on queue is 1  number of messages in queue is 1  max number of bytes on queue is 16384  pid of last msgsnd is 2510  pid of last msgrcv is 0  last msgsnd time is Sun Dec 29 18:28:21 2002  last msgrcv time is Thu Jan 1 08:00:00 1970  last change time is Sun Dec 29 18:28:20 2002  msg uid is 0  msg gid is 0  read from msg queue 1 bytes  //实际读出的字节数  current number of bytes on queue is 0  number of messages in queue is 0  max number of bytes on queue is 16384 //每个消息队列最大容量（字节数）  pid of last msgsnd is 2510  pid of last msgrcv is 2510  last msgsnd time is Sun Dec 29 18:28:21 2002  last msgrcv time is Sun Dec 29 18:28:22 2002  last change time is Sun Dec 29 18:28:20 2002  msg uid is 0  msg gid is 0  current number of bytes on queue is 0  number of messages in queue is 0  max number of bytes on queue is 16388 //可看出超级用户可修改消息队列最大容量  pid of last msgsnd is 2510  pid of last msgrcv is 2510 //对操作消息队列进程的跟踪  last msgsnd time is Sun Dec 29 18:28:21 2002  last msgrcv time is Sun Dec 29 18:28:22 2002  last change time is Sun Dec 29 18:28:23 2002 //msgctl()调用对msg\_ctime有影响  msg uid is 8  msg gid is 8 |

# 五．Linux环境进程间通信（四）：信号灯

## 1、信号灯概述

信号灯与其他进程间通信方式不大相同，它主要提供对进程间共享资源访问控制机制。相当于内存中的标志，进程可以根据它判定是否能够访问某些共享资源，同时，进程也可以修改该标志。除了用于访问控制外，还可用于进程同步。信号灯有以下两种类型：

* 二值信号灯：最简单的信号灯形式，信号灯的值只能取0或1，类似于互斥锁。   
  注：二值信号灯能够实现互斥锁的功能，但两者的关注内容不同。信号灯强调共享资源，只要共享资源可用，其他进程同样可以修改信号灯的值；互斥锁更强调进程，占用资源的进程使用完资源后，必须由进程本身来解锁。
* 计算信号灯：信号灯的值可以取任意非负值（当然受内核本身的约束）。

## 2、Linux信号灯

linux对信号灯的支持状况与消息队列一样，在red had 8.0发行版本中支持的是系统V的信号灯。因此，本文将主要介绍系统V信号灯及其相应API。在没有声明的情况下，以下讨论中指的都是系统V信号灯。

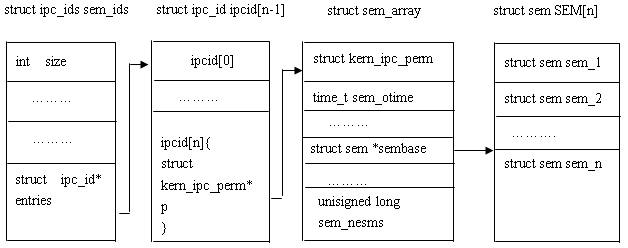
注意，通常所说的系统V信号灯指的是计数信号灯集。

## 3、信号灯与内核

1、系统V信号灯是随内核持续的，只有在内核重起或者显示删除一个信号灯集时，该信号灯集才会真正被删除。因此系统中记录信号灯的数据结构（struct ipc\_ids sem\_ids）位于内核中，系统中的所有信号灯都可以在结构sem\_ids中找到访问入口。

2、下图说明了内核与信号灯是怎样建立起联系的：

其中：struct ipc\_ids sem\_ids是内核中记录信号灯的全局数据结构；描述一个具体的信号灯及其相关信息。



其中，struct sem结构如下：

|  |
| --- |
| struct sem{  int semval; // current value  int sempid // pid of last operation  } |

从上图可以看出，全局数据结构struct ipc\_ids sem\_ids可以访问到struct kern\_ipc\_perm的第一个成员：struct kern\_ipc\_perm；而每个struct kern\_ipc\_perm能够与具体的信号灯对应起来是因为在该结构中，有一个key\_t类型成员key，而key则唯一确定一个信号灯集；同时，结构struct kern\_ipc\_perm的最后一个成员sem\_nsems确定了该信号灯在信号灯集中的顺序，这样内核就能够记录每个信号灯的信息了。kern\_ipc\_perm结构参见《Linux环境进程间通信（三）：消息队列》。struct sem\_array见附录1。

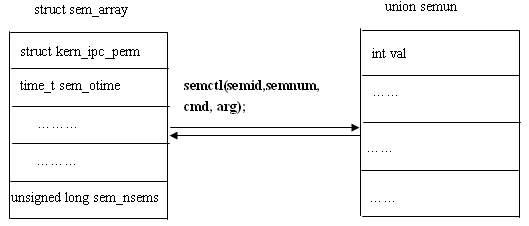
## 4、操作信号灯

对消息队列的操作无非有下面三种类型：

1、 打开或创建信号灯   
与消息队列的创建及打开基本相同，不再详述。

2、 信号灯值操作   
linux可以增加或减小信号灯的值，相应于对共享资源的释放和占有。具体参见后面的semop系统调用。

3、 获得或设置信号灯属性：   
系统中的每一个信号灯集都对应一个struct sem\_array结构，该结构记录了信号灯集的各种信息，存在于系统空间。为了设置、获得该信号灯集的各种信息及属性，在用户空间有一个重要的联合结构与之对应，即union semun。



联合semun数据结构各成员意义参见附录2

信号灯API

1、文件名到键值

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  key\_t ftok (char\*pathname, char proj)； |

它返回与路径pathname相对应的一个键值，具体用法请参考《Linux环境进程间通信（三）：消息队列》。

2、 linux特有的ipc()调用：

int ipc(unsigned int call, int first, int second, int third, void \*ptr, long fifth);

参数call取不同值时，对应信号灯的三个系统调用：   
当call为SEMOP时，对应int semop(int semid, struct sembuf \*sops, unsigned nsops)调用；   
当call为SEMGET时，对应int semget(key\_t key, int nsems, int semflg)调用；   
当call为SEMCTL时，对应int semctl(int semid，int semnum，int cmd，union semun arg)调用；   
这些调用将在后面阐述。

注：本人不主张采用系统调用ipc()，而更倾向于采用系统V或者POSIX进程间通信API。原因已在Linux环境进程间通信（三）：消息队列中给出。

3、系统V信号灯API

系统V消息队列API只有三个，使用时需要包括几个头文件：

|  |
| --- |
| #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h> |

1）int semget(key\_t key, int nsems, int semflg)   
参数key是一个键值，由ftok获得，唯一标识一个信号灯集，用法与msgget()中的key相同；参数nsems指定打开或者新创建的信号灯集中将包含信号灯的数目；semflg参数是一些标志位。参数key和semflg的取值，以及何时打开已有信号灯集或者创建一个新的信号灯集与msgget()中的对应部分相同，不再祥述。   
该调用返回与健值key相对应的信号灯集描述字。   
调用返回：成功返回信号灯集描述字，否则返回-1。   
注：如果key所代表的信号灯已经存在，且semget指定了IPC\_CREAT|IPC\_EXCL标志，那么即使参数nsems与原来信号灯的数目不等，返回的也是EEXIST错误；如果semget只指定了IPC\_CREAT标志，那么参数nsems必须与原来的值一致，在后面程序实例中还要进一步说明。

2）int semop(int semid, struct sembuf \*sops, unsigned nsops);   
semid是信号灯集ID，sops指向数组的每一个sembuf结构都刻画一个在特定信号灯上的操作。nsops为sops指向数组的大小。   
sembuf结构如下：

|  |
| --- |
| struct sembuf {  unsigned short sem\_num; /\* semaphore index in array \*/  short sem\_op; /\* semaphore operation \*/  short sem\_flg; /\* operation flags \*/  }; |

sem\_num对应信号集中的信号灯，0对应第一个信号灯。sem\_flg可取IPC\_NOWAIT以及SEM\_UNDO两个标志。如果设置了SEM\_UNDO标志，那么在进程结束时，相应的操作将被取消，这是比较重要的一个标志位。如果设置了该标志位，那么在进程没有释放共享资源就退出时，内核将代为释放。如果为一个信号灯设置了该标志，内核都要分配一个sem\_undo结构来记录它，为的是确保以后资源能够安全释放。事实上，如果进程退出了，那么它所占用就释放了，但信号灯值却没有改变，此时，信号灯值反映的已经不是资源占有的实际情况，在这种情况下，问题的解决就靠内核来完成。这有点像僵尸进程，进程虽然退出了，资源也都释放了，但内核进程表中仍然有它的记录，此时就需要父进程调用waitpid来解决问题了。   
sem\_op的值大于0，等于0以及小于0确定了对sem\_num指定的信号灯进行的三种操作。具体请参考linux相应手册页。   
这里需要强调的是semop同时操作多个信号灯，在实际应用中，对应多种资源的申请或释放。semop保证操作的原子性，这一点尤为重要。尤其对于多种资源的申请来说，要么一次性获得所有资源，要么放弃申请，要么在不占有任何资源情况下继续等待，这样，一方面避免了资源的浪费；另一方面，避免了进程之间由于申请共享资源造成死锁。   
也许从实际含义上更好理解这些操作：信号灯的当前值记录相应资源目前可用数目；sem\_op>0对应相应进程要释放sem\_op数目的共享资源；sem\_op=0可以用于对共享资源是否已用完的测试；sem\_op<0相当于进程要申请-sem\_op个共享资源。再联想操作的原子性，更不难理解该系统调用何时正常返回，何时睡眠等待。   
调用返回：成功返回0，否则返回-1。

3) int semctl(int semid，int semnum，int cmd，union semun arg)   
该系统调用实现对信号灯的各种控制操作，参数semid指定信号灯集，参数cmd指定具体的操作类型；参数semnum指定对哪个信号灯操作，只对几个特殊的cmd操作有意义；arg用于设置或返回信号灯信息。   
该系统调用详细信息请参见其手册页，这里只给出参数cmd所能指定的操作。

|  |  |
| --- | --- |
| IPC\_STAT | 获取信号灯信息，信息由arg.buf返回； |
| IPC\_SET | 设置信号灯信息，待设置信息保存在arg.buf中（在manpage中给出了可以设置哪些信息）； |
| GETALL | 返回所有信号灯的值，结果保存在arg.array中，参数sennum被忽略； |
| GETNCNT | 返回等待semnum所代表信号灯的值增加的进程数，相当于目前有多少进程在等待semnum代表的信号灯所代表的共享资源； |
| GETPID | 返回最后一个对semnum所代表信号灯执行semop操作的进程ID； |
| GETVAL | 返回semnum所代表信号灯的值； |
| GETZCNT | 返回等待semnum所代表信号灯的值变成0的进程数； |
| SETALL | 通过arg.array更新所有信号灯的值；同时，更新与本信号集相关的semid\_ds结构的sem\_ctime成员； |
| SETVAL | 设置semnum所代表信号灯的值为arg.val； |

调用返回：调用失败返回-1，成功返回与cmd相关：

|  |  |
| --- | --- |
| Cmd | return value |
| GETNCNT | Semncnt |
| GETPID | Sempid |
| GETVAL | Semval |
| GETZCNT | Semzcnt |

## 5、信号灯的限制

1、 一次系统调用semop可同时操作的信号灯数目SEMOPM，semop中的参数nsops如果超过了这个数目，将返回E2BIG错误。SEMOPM的大小特定与系统，redhat 8.0为32。

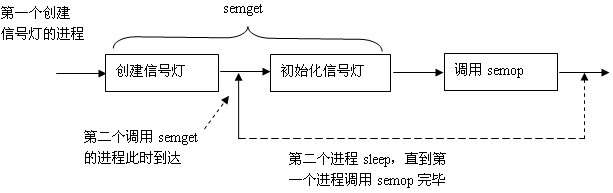
2、 信号灯的最大数目：SEMVMX，当设置信号灯值超过这个限制时，会返回ERANGE错误。在redhat 8.0中该值为32767。

3、 系统范围内信号灯集的最大数目SEMMNI以及系统范围内信号灯的最大数目SEMMNS。超过这两个限制将返回ENOSPC错误。redhat 8.0中该值为32000。

4、 每个信号灯集中的最大信号灯数目SEMMSL，redhat 8.0中为250。 SEMOPM以及SEMVMX是使用semop调用时应该注意的；SEMMNI以及SEMMNS是调用semget时应该注意的。SEMVMX同时也是semctl调用应该注意的。

## 6、竞争问题

第一个创建信号灯的进程同时也初始化信号灯，这样，系统调用semget包含了两个步骤：创建信号灯；初始化信号灯。由此可能导致一种竞争状态：第一个创建信号灯的进程在初始化信号灯时，第二个进程又调用semget，并且发现信号灯已经存在，此时，第二个进程必须具有判断是否有进程正在对信号灯进行初始化的能力。在参考文献[1]中，给出了绕过这种竞争状态的方法：当semget创建一个新的信号灯时，信号灯结构semid\_ds的sem\_otime成员初始化后的值为0。因此，第二个进程在成功调用semget后，可再次以IPC\_STAT命令调用semctl，等待sem\_otime变为非0值，此时可判断该信号灯已经初始化完毕。下图描述了竞争状态产生及解决方法：



实际上，这种解决方法也是基于这样一个假定：第一个创建信号灯的进程必须调用semop，这样sem\_otime才能变为非零值。另外，因为第一个进程可能不调用semop，或者semop操作需要很长时间，第二个进程可能无限期等待下去，或者等待很长时间。

## 7、信号灯应用实例

本实例有两个目的：1、获取各种信号灯信息；2、利用信号灯实现共享资源的申请和释放。并在程序中给出了详细注释。

|  |
| --- |
| #include <linux/sem.h>  #include <stdio.h>  #include <errno.h>  #define SEM\_PATH "/unix/my\_sem"  #define max\_tries 3  int semid;  main()  {  int flag1,flag2,key,i,init\_ok,tmperrno;  struct semid\_ds sem\_info;  struct seminfo sem\_info2;  union semun arg; //union semun： 请参考附录2  struct sembuf askfor\_res, free\_res;  flag1=IPC\_CREAT|IPC\_EXCL|00666;  flag2=IPC\_CREAT|00666;  key=ftok(SEM\_PATH,'a');  //error handling for ftok here;  init\_ok=0;  semid=semget(key,1,flag1);  //create a semaphore set that only includes one semphore.  if(semid<0)  {  tmperrno=errno;  perror("semget");  if(tmperrno==EEXIST)  //errno is undefined after a successful library call( including perror call)  //so it is saved in tmperrno.  {  semid=semget(key,1,flag2);  //flag2 只包含了IPC\_CREAT标志, 参数nsems(这里为1)必须与原来的信号灯数目一致  arg.buf=&sem\_info;  for(i=0; i<max\_tries; i++)  {  if(semctl(semid, 0, IPC\_STAT, arg)==-1)  { perror("semctl error"); i=max\_tries;}  else  {  if(arg.buf->sem\_otime!=0){ i=max\_tries; init\_ok=1;}  else sleep(1);  }  }  if(!init\_ok)  // do some initializing, here we assume that the first process that creates the sem  // will finish initialize the sem and run semop in max\_tries\*1 seconds. else it will  // not run semop any more.  {  arg.val=1;  if(semctl(semid,0,SETVAL,arg)==-1) perror("semctl setval error");  }  }  else  {perror("semget error, process exit"); exit(); }  }  else //semid>=0; do some initializing  {  arg.val=1;  if(semctl(semid,0,SETVAL,arg)==-1)  perror("semctl setval error");  }  //get some information about the semaphore and the limit of semaphore in redhat8.0  arg.buf=&sem\_info;  if(semctl(semid, 0, IPC\_STAT, arg)==-1)  perror("semctl IPC STAT");  printf("owner's uid is %d\n", arg.buf->sem\_perm.uid);  printf("owner's gid is %d\n", arg.buf->sem\_perm.gid);  printf("creater's uid is %d\n", arg.buf->sem\_perm.cuid);  printf("creater's gid is %d\n", arg.buf->sem\_perm.cgid);  arg.\_\_buf=&sem\_info2;  if(semctl(semid,0,IPC\_INFO,arg)==-1)  perror("semctl IPC\_INFO");  printf("the number of entries in semaphore map is %d \n", arg.\_\_buf->semmap);  printf("max number of semaphore identifiers is %d \n", arg.\_\_buf->semmni);  printf("mas number of semaphores in system is %d \n", arg.\_\_buf->semmns);  printf("the number of undo structures system wide is %d \n", arg.\_\_buf->semmnu);  printf("max number of semaphores per semid is %d \n", arg.\_\_buf->semmsl);  printf("max number of ops per semop call is %d \n", arg.\_\_buf->semopm);  printf("max number of undo entries per process is %d \n", arg.\_\_buf->semume);  printf("the sizeof of struct sem\_undo is %d \n", arg.\_\_buf->semusz);  printf("the maximum semaphore value is %d \n", arg.\_\_buf->semvmx);    //now ask for available resource:  askfor\_res.sem\_num=0;  askfor\_res.sem\_op=-1;  askfor\_res.sem\_flg=SEM\_UNDO;    if(semop(semid,&askfor\_res,1)==-1)//ask for resource  perror("semop error");    sleep(3);  //do some handling on the sharing resource here, just sleep on it 3 seconds  printf("now free the resource\n");    //now free resource  free\_res.sem\_num=0;  free\_res.sem\_op=1;  free\_res.sem\_flg=SEM\_UNDO;  if(semop(semid,&free\_res,1)==-1)//free the resource.  if(errno==EIDRM)  printf("the semaphore set was removed\n");  //you can comment out the codes below to compile a different version:  if(semctl(semid, 0, IPC\_RMID)==-1)  perror("semctl IPC\_RMID");  else printf("remove sem ok\n");  } |

注：读者可以尝试一下注释掉初始化步骤，进程在运行时会出现何种情况（进程在申请资源时会睡眠），同时可以像程序结尾给出的注释那样，把该程序编译成两个不同版本。下面是本程序的运行结果（操作系统redhat8.0）：

|  |
| --- |
| owner's uid is 0  owner's gid is 0  creater's uid is 0  creater's gid is 0  the number of entries in semaphore map is 32000  max number of semaphore identifiers is 128  mas number of semaphores in system is 32000  the number of undo structures system wide is 32000  max number of semaphores per semid is 250  max number of ops per semop call is 32  max number of undo entries per process is 32  the sizeof of struct sem\_undo is 20  the maximum semaphore value is 32767  now free the resource  remove sem ok |

Summary：信号灯与其它进程间通信方式有所不同，它主要用于进程间同步。通常所说的系统V信号灯实际上是一个信号灯的集合，可用于多种共享资源的进程间同步。每个信号灯都有一个值，可以用来表示当前该信号灯代表的共享资源可用（available）数量，如果一个进程要申请共享资源，那么就从信号灯值中减去要申请的数目，如果当前没有足够的可用资源，进程可以睡眠等待，也可以立即返回。当进程要申请多种共享资源时，linux可以保证操作的原子性，即要么申请到所有的共享资源，要么放弃所有资源，这样能够保证多个进程不会造成互锁。Linux对信号灯有各种各样的限制，程序中给出了输出结果。另外，如果读者想对信号灯作进一步的理解，建议阅读sem.h源代码，该文件不长，但给出了信号灯相关的重要数据结构。

#### 附录1

struct sem\_array如下：

|  |
| --- |
| /\*系统中的每个信号灯集对应一个sem\_array 结构 \*/  struct sem\_array {  struct kern\_ipc\_perm sem\_perm; /\* permissions .. see ipc.h \*/  time\_t sem\_otime; /\* last semop time \*/  time\_t sem\_ctime; /\* last change time \*/  struct sem \*sem\_base; /\* ptr to first semaphore in array \*/  struct sem\_queue \*sem\_pending; /\* pending operations to be processed \*/  struct sem\_queue \*\*sem\_pending\_last; /\* last pending operation \*/  struct sem\_undo \*undo; /\* undo requests on this array \*/  unsigned long sem\_nsems; /\* no. of semaphores in array \*/  }; |

其中，sem\_queue结构如下：

|  |
| --- |
| /\* 系统中每个因为信号灯而睡眠的进程，都对应一个sem\_queue结构\*/  struct sem\_queue {  struct sem\_queue \* next; /\* next entry in the queue \*/  struct sem\_queue \*\* prev;  /\* previous entry in the queue, \*(q->prev) == q \*/  struct task\_struct\* sleeper; /\* this process \*/  struct sem\_undo \* undo; /\* undo structure \*/  int pid; /\* process id of requesting process \*/  int status; /\* completion status of operation \*/  struct sem\_array \* sma; /\* semaphore array for operations \*/  int id; /\* internal sem id \*/  struct sembuf \* sops; /\* array of pending operations \*/  int nsops; /\* number of operations \*/  int alter; /\* operation will alter semaphore \*/  }; |

#### 附录2

union semun是系统调用semctl中的重要参数：

|  |
| --- |
| union semun {  int val; /\* value for SETVAL \*/  struct semid\_ds \*buf; /\* buffer for IPC\_STAT & IPC\_SET \*/  unsigned short \*array; /\* array for GETALL & SETALL \*/  struct seminfo \*\_\_buf; /\* buffer for IPC\_INFO \*/ //test!!  void \*\_\_pad;  };  struct seminfo {  int semmap;  int semmni;  int semmns;  int semmnu;  int semmsl;  int semopm;  int semume;  int semusz;  int semvmx;  int semaem;  }; |

# 六．Linux环境进程间通信（五）：共享内存（上）

采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据[1]：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

Linux的2.2.x内核支持多种共享内存方式，如mmap()系统调用，Posix共享内存，以及系统V共享内存。linux发行版本如Redhat 8.0支持mmap()系统调用及系统V共享内存，但还没实现Posix共享内存，本文将主要介绍mmap()系统调用及系统V共享内存API的原理及应用。

## 1、内核怎样保证各个进程寻址到同一个共享内存区域的内存页面

1、page cache及swap cache中页面的区分：一个被访问文件的物理页面都驻留在page cache或swap cache中，一个页面的所有信息由struct page来描述。struct page中有一个域为指针mapping ，它指向一个struct address\_space类型结构。page cache或swap cache中的所有页面就是根据address\_space结构以及一个偏移量来区分的。

2、文件与address\_space结构的对应：一个具体的文件在打开后，内核会在内存中为之建立一个struct inode结构，其中的i\_mapping域指向一个address\_space结构。这样，一个文件就对应一个address\_space结构，一个address\_space与一个偏移量能够确定一个page cache 或swap cache中的一个页面。因此，当要寻址某个数据时，很容易根据给定的文件及数据在文件内的偏移量而找到相应的页面。

3、进程调用mmap()时，只是在进程空间内新增了一块相应大小的缓冲区，并设置了相应的访问标识，但并没有建立进程空间到物理页面的映射。因此，第一次访问该空间时，会引发一个缺页异常。

4、对于共享内存映射情况，缺页异常处理程序首先在swap cache中寻找目标页（符合address\_space以及偏移量的物理页），如果找到，则直接返回地址；如果没有找到，则判断该页是否在交换区(swap area)，如果在，则执行一个换入操作；如果上述两种情况都不满足，处理程序将分配新的物理页面，并把它插入到page cache中。进程最终将更新进程页表。   
注：对于映射普通文件情况（非共享映射），缺页异常处理程序首先会在page cache中根据address\_space以及数据偏移量寻找相应的页面。如果没有找到，则说明文件数据还没有读入内存，处理程序会从磁盘读入相应的页面，并返回相应地址，同时，进程页表也会更新。

5、所有进程在映射同一个共享内存区域时，情况都一样，在建立线性地址与物理地址之间的映射之后，不论进程各自的返回地址如何，实际访问的必然是同一个共享内存区域对应的物理页面。   
注：一个共享内存区域可以看作是特殊文件系统shm中的一个文件，shm的安装点在交换区上。

上面涉及到了一些数据结构，围绕数据结构理解问题会容易一些。

## 2、mmap()及其相关系统调用

mmap()系统调用使得进程之间通过映射同一个普通文件实现共享内存。普通文件被映射到进程地址空间后，进程可以向访问普通内存一样对文件进行访问，不必再调用read()，write（）等操作。

注：实际上，mmap()系统调用并不是完全为了用于共享内存而设计的。它本身提供了不同于一般对普通文件的访问方式，进程可以像读写内存一样对普通文件的操作。而Posix或系统V的共享内存IPC则纯粹用于共享目的，当然mmap()实现共享内存也是其主要应用之一。

### 2.1 mmap()系统调用形式

void\* mmap ( void \* addr , size\_t len , int prot , int flags , int fd , off\_t offset )   
参数fd为即将映射到进程空间的文件描述字，一般由open()返回，同时，fd可以指定为-1，此时须指定flags参数中的MAP\_ANON，表明进行的是匿名映射（不涉及具体的文件名，避免了文件的创建及打开，很显然只能用于具有亲缘关系的进程间通信）。len是映射到调用进程地址空间的字节数，它从被映射文件开头offset个字节开始算起。prot 参数指定共享内存的访问权限。可取如下几个值的或：PROT\_READ（可读） , PROT\_WRITE （可写）, PROT\_EXEC （可执行）, PROT\_NONE（不可访问）。flags由以下几个常值指定：MAP\_SHARED , MAP\_PRIVATE , MAP\_FIXED，其中，MAP\_SHARED , MAP\_PRIVATE必选其一，而MAP\_FIXED则不推荐使用。offset参数一般设为0，表示从文件头开始映射。参数addr指定文件应被映射到进程空间的起始地址，一般被指定一个空指针，此时选择起始地址的任务留给内核来完成。函数的返回值为最后文件映射到进程空间的地址，进程可直接操作起始地址为该值的有效地址。这里不再详细介绍mmap()的参数，读者可参考mmap()手册页获得进一步的信息。

### 2.2 系统调用mmap()用于共享内存的两种方式

（1）使用普通文件提供的内存映射：适用于任何进程之间；此时，需要打开或创建一个文件，然后再调用mmap()；典型调用代码如下：

|  |
| --- |
| fd=open(name, flag, mode);  if(fd<0)  ... |

ptr=mmap(NULL, len , PROT\_READ|PROT\_WRITE, MAP\_SHARED , fd , 0); 通过mmap()实现共享内存的通信方式有许多特点和要注意的地方，我们将在范例中进行具体说明。

（2）使用特殊文件提供匿名内存映射：适用于具有亲缘关系的进程之间；由于父子进程特殊的亲缘关系，在父进程中先调用mmap()，然后调用fork()。那么在调用fork()之后，子进程继承父进程匿名映射后的地址空间，同样也继承mmap()返回的地址，这样，父子进程就可以通过映射区域进行通信了。注意，这里不是一般的继承关系。一般来说，子进程单独维护从父进程继承下来的一些变量。而mmap()返回的地址，却由父子进程共同维护。   
对于具有亲缘关系的进程实现共享内存最好的方式应该是采用匿名内存映射的方式。此时，不必指定具体的文件，只要设置相应的标志即可，参见范例2。

### 2.3 系统调用munmap()

int munmap( void \* addr, size\_t len )   
该调用在进程地址空间中解除一个映射关系，addr是调用mmap()时返回的地址，len是映射区的大小。当映射关系解除后，对原来映射地址的访问将导致段错误发生。

### 2.4 系统调用msync()

int msync ( void \* addr , size\_t len, int flags)   
一般说来，进程在映射空间的对共享内容的改变并不直接写回到磁盘文件中，往往在调用munmap（）后才执行该操作。可以通过调用msync()实现磁盘上文件内容与共享内存区的内容一致。

## 3、mmap()范例

下面将给出使用mmap()的两个范例：范例1给出两个进程通过映射普通文件实现共享内存通信；范例2给出父子进程通过匿名映射实现共享内存。系统调用mmap()有许多有趣的地方，下面是通过mmap（）映射普通文件实现进程间的通信的范例，我们通过该范例来说明mmap()实现共享内存的特点及注意事项。

### 范例1：两个进程通过映射普通文件实现共享内存通信

范例1包含两个子程序：map\_normalfile1.c及map\_normalfile2.c。编译两个程序，可执行文件分别为map\_normalfile1及map\_normalfile2。两个程序通过命令行参数指定同一个文件来实现共享内存方式的进程间通信。map\_normalfile2试图打开命令行参数指定的一个普通文件，把该文件映射到进程的地址空间，并对映射后的地址空间进行写操作。map\_normalfile1把命令行参数指定的文件映射到进程地址空间，然后对映射后的地址空间执行读操作。这样，两个进程通过命令行参数指定同一个文件来实现共享内存方式的进程间通信。

下面是两个程序代码：

|  |
| --- |
| /\*-------------map\_normalfile1.c-----------\*/  #include <sys/mman.h>  #include <sys/types.h>  #include <fcntl.h>  #include <unistd.h>  typedef struct{  char name[4];  int age;  }people;  main(int argc, char\*\* argv) // map a normal file as shared mem:  {  int fd,i;  people \*p\_map;  char temp;    fd=open(argv[1],O\_CREAT|O\_RDWR|O\_TRUNC,00777);  lseek(fd,sizeof(people)\*5-1,SEEK\_SET);  write(fd,"",1);    p\_map = (people\*) mmap( NULL,sizeof(people)\*10,PROT\_READ|PROT\_WRITE,  MAP\_SHARED,fd,0 );  close( fd );  temp = 'a';  for(i=0; i<10; i++)  {  temp += 1;  memcpy( ( \*(p\_map+i) ).name, &temp,2 );  ( \*(p\_map+i) ).age = 20+i;  }  printf(" initialize over \n ")；  sleep(10);  munmap( p\_map, sizeof(people)\*10 );  printf( "umap ok \n" );  }  /\*-------------map\_normalfile2.c-----------\*/  #include <sys/mman.h>  #include <sys/types.h>  #include <fcntl.h>  #include <unistd.h>  typedef struct{  char name[4];  int age;  }people;  main(int argc, char\*\* argv) // map a normal file as shared mem:  {  int fd,i;  people \*p\_map;  fd=open( argv[1],O\_CREAT|O\_RDWR,00777 );  p\_map = (people\*)mmap(NULL,sizeof(people)\*10,PROT\_READ|PROT\_WRITE,  MAP\_SHARED,fd,0);  for(i = 0;i<10;i++)  {  printf( "name: %s age %d;\n",(\*(p\_map+i)).name, (\*(p\_map+i)).age );  }  munmap( p\_map,sizeof(people)\*10 );  } |

map\_normalfile1.c首先定义了一个people数据结构，（在这里采用数据结构的方式是因为，共享内存区的数据往往是有固定格式的，这由通信的各个进程决定，采用结构的方式有普遍代表性）。map\_normfile1首先打开或创建一个文件，并把文件的长度设置为5个people结构大小。然后从mmap()的返回地址开始，设置了10个people结构。然后，进程睡眠10秒钟，等待其他进程映射同一个文件，最后解除映射。

map\_normfile2.c只是简单的映射一个文件，并以people数据结构的格式从mmap()返回的地址处读取10个people结构，并输出读取的值，然后解除映射。

分别把两个程序编译成可执行文件map\_normalfile1和map\_normalfile2后，在一个终端上先运行./map\_normalfile2 /tmp/test\_shm，程序输出结果如下：

|  |
| --- |
| initialize over  umap ok |

在map\_normalfile1输出initialize over 之后，输出umap ok之前，在另一个终端上运行map\_normalfile2 /tmp/test\_shm，将会产生如下输出(为了节省空间，输出结果为稍作整理后的结果)：

|  |
| --- |
| name: b age 20; name: c age 21; name: d age 22; name: e age 23; name: f age 24;  name: g age 25; name: h age 26; name: I age 27; name: j age 28; name: k age 29; |

在map\_normalfile1 输出umap ok后，运行map\_normalfile2则输出如下结果：

|  |
| --- |
| name: b age 20; name: c age 21; name: d age 22; name: e age 23; name: f age 24;  name: age 0; name: age 0; name: age 0; name: age 0; name: age 0; |

**从程序的运行结果中可以得出的结论**

1、 最终被映射文件的内容的长度不会超过文件本身的初始大小，即映射不能改变文件的大小；

2、 可以用于进程通信的有效地址空间大小大体上受限于被映射文件的大小，但不完全受限于文件大小。打开文件被截短为5个people结构大小，而在map\_normalfile1中初始化了10个people数据结构，在恰当时候（map\_normalfile1输出initialize over 之后，输出umap ok之前）调用map\_normalfile2会发现map\_normalfile2将输出全部10个people结构的值，后面将给出详细讨论。   
注：在linux中，内存的保护是以页为基本单位的，即使被映射文件只有一个字节大小，内核也会为映射分配一个页面大小的内存。当被映射文件小于一个页面大小时，进程可以对从mmap()返回地址开始的一个页面大小进行访问，而不会出错；但是，如果对一个页面以外的地址空间进行访问，则导致错误发生，后面将进一步描述。因此，可用于进程间通信的有效地址空间大小不会超过文件大小及一个页面大小的和。

3、 文件一旦被映射后，调用mmap()的进程对返回地址的访问是对某一内存区域的访问，暂时脱离了磁盘上文件的影响。所有对mmap()返回地址空间的操作只在内存中有意义，只有在调用了munmap()后或者msync()时，才把内存中的相应内容写回磁盘文件，所写内容仍然不能超过文件的大小。

### 范例2：父子进程通过匿名映射实现共享内存

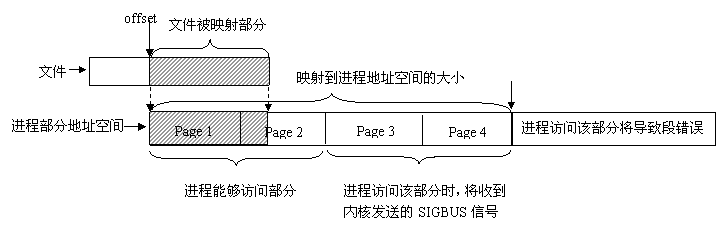
|  |
| --- |
| #include <sys/mman.h>  #include <sys/types.h>  #include <fcntl.h>  #include <unistd.h>  typedef struct{  char name[4];  int age;  }people;  main(int argc, char\*\* argv)  {  int i;  people \*p\_map;  char temp;  p\_map=(people\*)mmap(NULL,sizeof(people)\*10,PROT\_READ|PROT\_WRITE,  MAP\_SHARED|MAP\_ANONYMOUS,-1,0);  if(fork() == 0)  {  sleep(2);  for(i = 0;i<5;i++)  printf("child read: the %d people's age is %d\n",i+1,(\*(p\_map+i)).age);  (\*p\_map).age = 100;  munmap(p\_map,sizeof(people)\*10); //实际上，进程终止时，会自动解除映射。  exit();  }  temp = 'a';  for(i = 0;i<5;i++)  {  temp += 1;  memcpy((\*(p\_map+i)).name, &temp,2);  (\*(p\_map+i)).age=20+i;  }  sleep(5);  printf( "parent read: the first people,s age is %d\n",(\*p\_map).age );  printf("umap\n");  munmap( p\_map,sizeof(people)\*10 );  printf( "umap ok\n" );  } |

考察程序的输出结果，体会父子进程匿名共享内存：

|  |
| --- |
| child read: the 1 people's age is 20  child read: the 2 people's age is 21  child read: the 3 people's age is 22  child read: the 4 people's age is 23  child read: the 5 people's age is 24  parent read: the first people,s age is 100  umap  umap ok |

## 4、对mmap()返回地址的访问

前面对范例运行结构的讨论中已经提到，linux采用的是页式管理机制。对于用mmap()映射普通文件来说，进程会在自己的地址空间新增一块空间，空间大小由mmap()的len参数指定，注意，进程并不一定能够对全部新增空间都能进行有效访问。进程能够访问的有效地址大小取决于文件被映射部分的大小。简单的说，能够容纳文件被映射部分大小的最少页面个数决定了进程从mmap()返回的地址开始，能够有效访问的地址空间大小。超过这个空间大小，内核会根据超过的严重程度返回发送不同的信号给进程。可用如下图示说明：



注意：文件被映射部分而不是整个文件决定了进程能够访问的空间大小，另外，如果指定文件的偏移部分，一定要注意为页面大小的整数倍。下面是对进程映射地址空间的访问范例：

|  |
| --- |
| #include <sys/mman.h>  #include <sys/types.h>  #include <fcntl.h>  #include <unistd.h>  typedef struct{  char name[4];  int age;  }people;  main(int argc, char\*\* argv)  {  int fd,i;  int pagesize,offset;  people \*p\_map;    pagesize = sysconf(\_SC\_PAGESIZE);  printf("pagesize is %d\n",pagesize);  fd = open(argv[1],O\_CREAT|O\_RDWR|O\_TRUNC,00777);  lseek(fd,pagesize\*2-100,SEEK\_SET);  write(fd,"",1);  offset = 0; //此处offset = 0编译成版本1；offset = pagesize编译成版本2  p\_map = (people\*)mmap(NULL,pagesize\*3,PROT\_READ|PROT\_WRITE,MAP\_SHARED,fd,offset);  close(fd);    for(i = 1; i<10; i++)  {  (\*(p\_map+pagesize/sizeof(people)\*i-2)).age = 100;  printf("access page %d over\n",i);  (\*(p\_map+pagesize/sizeof(people)\*i-1)).age = 100;  printf("access page %d edge over, now begin to access page %d\n",i, i+1);  (\*(p\_map+pagesize/sizeof(people)\*i)).age = 100;  printf("access page %d over\n",i+1);  }  munmap(p\_map,sizeof(people)\*10);  } |

如程序中所注释的那样，把程序编译成两个版本，两个版本主要体现在文件被映射部分的大小不同。文件的大小介于一个页面与两个页面之间（大小为：pagesize\*2-99），版本1的被映射部分是整个文件，版本2的文件被映射部分是文件大小减去一个页面后的剩余部分，不到一个页面大小(大小为：pagesize-99)。程序中试图访问每一个页面边界，两个版本都试图在进程空间中映射pagesize\*3的字节数。

版本1的输出结果如下：

|  |
| --- |
| pagesize is 4096  access page 1 over  access page 1 edge over, now begin to access page 2  access page 2 over  access page 2 over  access page 2 edge over, now begin to access page 3  Bus error //被映射文件在进程空间中覆盖了两个页面，此时，进程试图访问第三个页面 |

版本2的输出结果如下：

|  |
| --- |
| pagesize is 4096  access page 1 over  access page 1 edge over, now begin to access page 2  Bus error //被映射文件在进程空间中覆盖了一个页面，此时，进程试图访问第二个页面 |

## 结论

采用系统调用mmap()实现进程间通信是很方便的，在应用层上接口非常简洁。内部实现机制区涉及到了linux存储管理以及文件系统等方面的内容，可以参考一下相关重要数据结构来加深理解。在本专题的后面部分，将介绍系统v共享内存的实现。

# 七．Linux环境进程间通信（五）：共享内存（下）

系统调用mmap()通过映射一个普通文件实现共享内存。系统V则是通过映射特殊文件系统shm中的文件实现进程间的共享内存通信。也就是说，每个共享内存区域对应特殊文件系统shm中的一个文件（这是通过shmid\_kernel结构联系起来的），后面还将阐述。

## 1、系统V共享内存原理

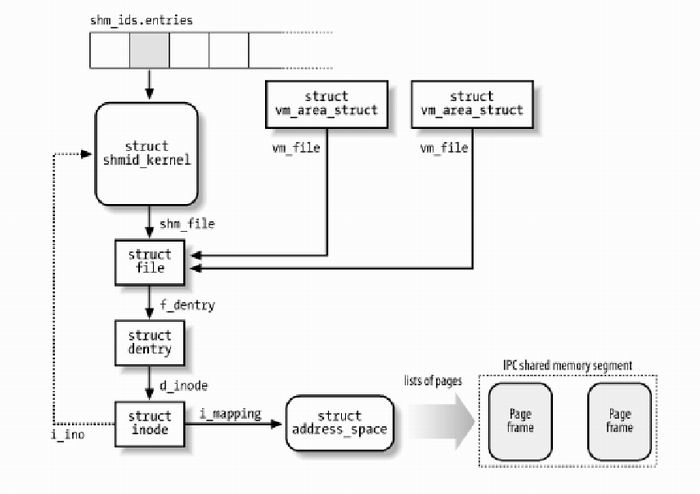
进程间需要共享的数据被放在一个叫做IPC共享内存区域的地方，所有需要访问该共享区域的进程都要把该共享区域映射到本进程的地址空间中去。系统V共享内存通过shmget获得或创建一个IPC共享内存区域，并返回相应的标识符。内核在保证shmget获得或创建一个共享内存区，初始化该共享内存区相应的shmid\_kernel结构注同时，还将在特殊文件系统shm中，创建并打开一个同名文件，并在内存中建立起该文件的相应dentry及inode结构，新打开的文件不属于任何一个进程（任何进程都可以访问该共享内存区）。所有这一切都是系统调用shmget完成的。

注：每一个共享内存区都有一个控制结构struct shmid\_kernel，shmid\_kernel是共享内存区域中非常重要的一个数据结构，它是存储管理和文件系统结合起来的桥梁，定义如下：

|  |
| --- |
| struct shmid\_kernel /\* private to the kernel \*/  {  struct kern\_ipc\_perm shm\_perm;  struct file \* shm\_file;  int id;  unsigned long shm\_nattch;  unsigned long shm\_segsz;  time\_t shm\_atim;  time\_t shm\_dtim;  time\_t shm\_ctim;  pid\_t shm\_cprid;  pid\_t shm\_lprid;  }; |

该结构中最重要的一个域应该是shm\_file，它存储了将被映射文件的地址。每个共享内存区对象都对应特殊文件系统shm中的一个文件，一般情况下，特殊文件系统shm中的文件是不能用read()、write()等方法访问的，当采取共享内存的方式把其中的文件映射到进程地址空间后，可直接采用访问内存的方式对其访问。

这里我们采用[1]中的图表给出与系统V共享内存相关数据结构：



正如消息队列和信号灯一样，内核通过数据结构struct ipc\_ids shm\_ids维护系统中的所有共享内存区域。上图中的shm\_ids.entries变量指向一个ipc\_id结构数组，而每个ipc\_id结构数组中有个指向kern\_ipc\_perm结构的指针。到这里读者应该很熟悉了，对于系统V共享内存区来说，kern\_ipc\_perm的宿主是shmid\_kernel结构，shmid\_kernel是用来描述一个共享内存区域的，这样内核就能够控制系统中所有的共享区域。同时，在shmid\_kernel结构的file类型指针shm\_file指向文件系统shm中相应的文件，这样，共享内存区域就与shm文件系统中的文件对应起来。

在创建了一个共享内存区域后，还要将它映射到进程地址空间，系统调用shmat()完成此项功能。由于在调用shmget()时，已经创建了文件系统shm中的一个同名文件与共享内存区域相对应，因此，调用shmat()的过程相当于映射文件系统shm中的同名文件过程，原理与mmap()大同小异。

## 2、系统V共享内存API

对于系统V共享内存，主要有以下几个API：shmget()、shmat()、shmdt()及shmctl()。

|  |
| --- |
| #include <sys/ipc.h>  #include <sys/shm.h> |

shmget（）用来获得共享内存区域的ID，如果不存在指定的共享区域就创建相应的区域。shmat()把共享内存区域映射到调用进程的地址空间中去，这样，进程就可以方便地对共享区域进行访问操作。shmdt()调用用来解除进程对共享内存区域的映射。shmctl实现对共享内存区域的控制操作。这里我们不对这些系统调用作具体的介绍，读者可参考相应的手册页面，后面的范例中将给出它们的调用方法。

注：shmget的内部实现包含了许多重要的系统V共享内存机制；shmat在把共享内存区域映射到进程空间时，并不真正改变进程的页表。当进程第一次访问内存映射区域访问时，会因为没有物理页表的分配而导致一个缺页异常，然后内核再根据相应的存储管理机制为共享内存映射区域分配相应的页表。

## 3、系统V共享内存限制

在/proc/sys/kernel/目录下，记录着系统V共享内存的一下限制，如一个共享内存区的最大字节数shmmax，系统范围内最大共享内存区标识符数shmmni等，可以手工对其调整，但不推荐这样做。

在[2]中，给出了这些限制的测试方法，不再赘述。

## 4、系统V共享内存范例

本部分将给出系统V共享内存API的使用方法，并对比分析系统V共享内存机制与mmap()映射普通文件实现共享内存之间的差异，首先给出两个进程通过系统V共享内存通信的范例：

|  |
| --- |
| /\*\*\*\*\* testwrite.c \*\*\*\*\*\*\*/  #include <sys/ipc.h>  #include <sys/shm.h>  #include <sys/types.h>  #include <unistd.h>  typedef struct{  char name[4];  int age;  } people;  main(int argc, char\*\* argv)  {  int shm\_id,i;  key\_t key;  char temp;  people \*p\_map;  char\* name = "/dev/shm/myshm2";  key = ftok(name,0);  if(key==-1)  perror("ftok error");  shm\_id=shmget(key,4096,IPC\_CREAT);  if(shm\_id==-1)  {  perror("shmget error");  return;  }  p\_map=(people\*)shmat(shm\_id,NULL,0);  temp='a';  for(i = 0;i<10;i++)  {  temp+=1;  memcpy((\*(p\_map+i)).name,&temp,1);  (\*(p\_map+i)).age=20+i;  }  if(shmdt(p\_map)==-1)  perror(" detach error ");  }  /\*\*\*\*\*\*\*\*\*\* testread.c \*\*\*\*\*\*\*\*\*\*\*\*/  #include <sys/ipc.h>  #include <sys/shm.h>  #include <sys/types.h>  #include <unistd.h>  typedef struct{  char name[4];  int age;  } people;  main(int argc, char\*\* argv)  {  int shm\_id,i;  key\_t key;  people \*p\_map;  char\* name = "/dev/shm/myshm2";  key = ftok(name,0);  if(key == -1)  perror("ftok error");  shm\_id = shmget(key,4096,IPC\_CREAT);  if(shm\_id == -1)  {  perror("shmget error");  return;  }  p\_map = (people\*)shmat(shm\_id,NULL,0);  for(i = 0;i<10;i++)  {  printf( "name:%s\n",(\*(p\_map+i)).name );  printf( "age %d\n",(\*(p\_map+i)).age );  }  if(shmdt(p\_map) == -1)  perror(" detach error ");  } |

testwrite.c创建一个系统V共享内存区，并在其中写入格式化数据；testread.c访问同一个系统V共享内存区，读出其中的格式化数据。分别把两个程序编译为testwrite及testread，先后执行./testwrite及./testread 则./testread输出结果如下：

|  |
| --- |
| name: b age 20; name: c age 21; name: d age 22; name: e age 23; name: f age 24;  name: g age 25; name: h age 26; name: I age 27; name: j age 28; name: k age 29; |

通过对试验结果分析，对比系统V与mmap()映射普通文件实现共享内存通信，可以得出如下结论：

1、 系统V共享内存中的数据，从来不写入到实际磁盘文件中去；而通过mmap()映射普通文件实现的共享内存通信可以指定何时将数据写入磁盘文件中。注：前面讲到，系统V共享内存机制实际是通过映射特殊文件系统shm中的文件实现的，文件系统shm的安装点在交换分区上，系统重新引导后，所有的内容都丢失。

2、 系统V共享内存是随内核持续的，即使所有访问共享内存的进程都已经正常终止，共享内存区仍然存在（除非显式删除共享内存），在内核重新引导之前，对该共享内存区域的任何改写操作都将一直保留。

3、 通过调用mmap()映射普通文件进行进程间通信时，一定要注意考虑进程何时终止对通信的影响。而通过系统V共享内存实现通信的进程则不然。注：这里没有给出shmctl的使用范例，原理与消息队列大同小异。

## 结论

共享内存允许两个或多个进程共享一给定的存储区，因为数据不需要来回复制，所以是最快的一种进程间通信机制。共享内存可以通过mmap()映射普通文件（特殊情况下还可以采用匿名映射）机制实现，也可以通过系统V共享内存机制实现。应用接口和原理很简单，内部机制复杂。为了实现更安全通信，往往还与信号灯等同步机制共同使用。

共享内存涉及到了存储管理以及文件系统等方面的知识，深入理解其内部机制有一定的难度，关键还要紧紧抓住内核使用的重要数据结构。系统V共享内存是以文件的形式组织在特殊文件系统shm中的。通过shmget可以创建或获得共享内存的标识符。取得共享内存标识符后，要通过shmat将这个内存区映射到本进程的虚拟地址空间。