# 进程间通信

在Linux系统中，有时候需要多个进程协作，完成某项任务。这就需要进程之间或者进程与线程之间需要传递消息，或者需要同步来协调彼此的工作。本章重点讲述进程间通信（interprocess communication 或者IPC）的机制。

前面信号一章曾经提到，信号也是一种进程间通讯的机制，尽管信号的主要作用不是这个。一个进程向另外一个进程发送信号，传递的信息是信号编号，除此以外，采用sigqueue函数发送信号，可以在信号上绑定数据(整型数字或者 指针)。尽管如此，不建议将信号作为进程间通信的常规手段。

在线程一章提到，线程，在Linux中称为轻量级进程，线程的同步手段（互斥量和条件等待）本质也是进程间通信。

进程间通信的手段，从作用上说，大体可以分成以下几类

通信：这些手段的目的是为了在进程传递消息，交换数据。管道，命名管道，消息队列（包括System V消息队列和POSIX消息队列），共享内存（包括System V共享内存和POSIX共享内存）都属于这种类型。细分下来，通信也可以分成两种，一种是用来传递消息的，另外一种是通过共享一片内存区域来完成信息的交换。

信号：作为进程间通信的一种，通知目标进程某件事情发生了，请执行事先约定好的动作

同步：这些手段的目的是协调进程的操作。某些内存的数据或者文件的数据块不能够由多个进程更新，否则可能会产生错误的结果。为了解决这种情况，Linux提供了下面几种：信号量（包括System V信号量和POSIX信号量），文件锁（包括文件锁和记录锁），以及用于线程的互斥量和条件等待。

从历史的角度来说，Linux下进程间通信的手段基本上是从UNIX平台上的进程间通信手段继承来的。

管道是第一个广泛应用的最古老的进程间通信形式，可以在程序中使用，更可以在shell中使用。日常在终端执行shell命令，会大量用到管道。管道的缺陷在于只能在有亲缘关系（有共同的祖先）的进程之间使用。为了解决这个问题，引入命名管道。

接下来AT&T的贝尔实验室和加州大学伯克利分校的伯克利软件发布中心(BSD)分别开发出了风格迥异的进程间通信手段。前者通过对早期的进程通信手段的改进和扩充，开发出System V IPC，包括消息队列，信号量和共享内存，但是这些手段，进程间通信始终局限在单个计算机之内。BSD则走了一条完全不同的道路，开发了基于套接字（socket）的进程间通信的手段，跳过了单机的限制，可以实现不同计算机之间的进程间通信。Linux作为后期之秀，将System V IPC和BSD socket都继承了下来，丰富了进程间通信的手段。

System V IPC工具的设计与传统的UNIX 文件模型不太一致。System V IPC工具没有类似文件描述符的IPC的句柄的概念。内核不会记录进程已经打开了某个IPC的对象，换句话说，内核没有维护IPC对象的引用计数，当多个进程打开IPC对象时，到底由那个进程销毁该IPC对象，完全要由应用程序来决定，这就增加了复杂性。除此以外，System V IPC工具由于和传统的文件模型不一致，接口比较复杂，System V的信号量尤为复杂。由于这些缺点，后面引入了POSIX IPC。

POSIX IPC工具是明显晚于System V IPC的，System V IPC广泛应用一段时间后，才开始设计POSIX IPC，因此，设计者可以借鉴System V IPC的长处，避免其缺点。从设计角度上将，POSIX IPC是优于System V IPC的，接口简单，易于使用。但是可移植性并不如System V IPC。

下面分别介绍提到了进程间通讯的工具。其中套接字后面会有专门章节介绍，就不在本章提及了。

12.1 管道

12.1.1管道概述

管道是出现最早的进程间通信的手段。在日常shell中我们执行命令，经常将上一个命令的输出，作为下一个命令的输入。用多个命令配合完成一件事情。这是用管道实现的。



如上图所示，进程who的输出，通过管道，传递个下游的wc进程作为输入，从而相互配合完成了一件事情。上图很形象地描述了管道的一种使用方法。

我们虽然从shell命令行可以初步的理解pipe的用法，但是还是需要细细分析管道如何做到多个进程之间的通信。

管道的作用是在有亲缘关系的进程之间传递消息。所谓有亲缘关系，是指有一个共同的祖先。所以管道并非只能在父子进程之间传递消息，也可以在兄弟进程之间传递，如shell命令中的管道，也可以在祖孙进程，自然也可以在叔侄进程之间传递消息，只要共同的祖先曾经调用了pipe函数，打开的管道文件就会在fork之后，被各个后代进程所共享。打开的管道文件，就像是一个家族秘密场所，远祖进程创建，家族所有成员都知晓。家族成员A可以将消息扔到该秘密场所，等待另外一个接头的家族成员B来取走消息，阅后即焚。

从严格意义上说，家族里面多个进程可以都往同一个秘密场所里面扔消息，也可以由多个进程从秘密场所里面取消息，但是有一个问题。管道实质是一个字节流，并非前面提到的消息，没有消息的边界，如果多个进程发送的字节流混在一起，无法辨认各个消息。所以一个管道，一般是2个有亲缘关系的进程之间用它来通信。从程序设计的角度来讲，远祖进程调用pipe函数的时候，就是为某两个有亲缘关系通信设计的，其他有亲缘关系的进程不许搅局。其他进程之间想通信怎么办？为它们创建它们之间需要用的另外的管道。

前面提到了管道是字节流，但是读取管道是消耗型的行为，就是前面提到的阅后即焚的行为，一个进程读取了管道内的一些内容之后，这些内容就不会继续在管道之中了。所以一般来讲管道是单向的。一个进程负责往管道里面写内容，另外一个进程读取管道的内容。两个有亲缘关系的进程发扬二杆子精神，都要往管道里面写，都要往管道里面读，自然也是可以的，但是字节流可能会变得混乱，从而无法完成通信的任务。如果两个进程之间想双向通信怎么办？可以建立两个管道，一个管道A进程写，B进程读，另一个管道，B进程写，A进程读，从而实现双向通信。

管道文件是一种文件，我们可以调用read，write，close等操作，管道文件又不是一种普通的文件，它属于一种独特的文件系统：pipefs。管道文件实质上就是作为缓冲区的内存页面，只不过采用了文件系统的机制。

12.1.2管道接口

在Linux下，可以使用如下接口创建管道：

#include <unistd.h>

int pipe(int pipefd[2]);

如果成功，返回值是0，如果失败，返回值是-1，并且设置errno。需要处理的errno有：

|  |  |
| --- | --- |
| errno | 原因 |
| EMFILE | 该进程使用的文件描述符已经多于MAX\_OPEN-2 |
| ENFILE | 系统中同时打开的文件已经超过系统的限制 |

成功调用pipe函数之后，会返回两个打开的文件描述符，一个是管道的读取端pipefd[0]，另一个是管道的写入端pipefd[1]。管道没有外部文件或者名字与之关联，程序没有选择，只能通过文件描述符来访问管道，因此，只有能看到这两个文件描述符的进程才能够使用管道。谁能看到进程打开的文件描述符呢？只有该进程，以及该进程的子孙进程。这就限制了管道的使用范围。

成功调用pipe函数之后，可以对pipefd[1]使用write，往管道里面写数据，如下所示：

write(pipefd[1],wbuf,count);

一旦向管道的写入端写入数据后，就可以对pipefd[0]调用read，读出管道里面的数据。如下所示，管道上的read调用返回的字节数等于请求字节数和管道中当前存在的字节数的最小值。如果当前管道为空，是read调用会阻塞。

read(pipefd[0],rbuf,count);

前面提到过管道一端是写入端pipefd[1]，一端是读入端pipefd[0]。不能对读入端文件描述符调用写操作，也不能对写入端文件描述符调用读操作。如果我二杆子精神再次爆发，非要写读入端，或者读写入端，会怎么样？

在调用pipe函数是，打开的两个文件，读入端pipefd[0]支持的文件操作定义在read\_pipefifo\_fops，写入端pipefd[1]支持的文件操作定义在write\_pipefifo\_fops。

const struct file\_operations read\_pipefifo\_fops = {

.llseek = no\_llseek,

.read = do\_sync\_read,

.aio\_read = pipe\_read,

.write = bad\_pipe\_w,

.poll = pipe\_poll,

.unlocked\_ioctl = pipe\_ioctl,

.open = pipe\_read\_open,

.release = pipe\_read\_release,

.fasync = pipe\_read\_fasync,

};

const struct file\_operations write\_pipefifo\_fops = {

.llseek = no\_llseek,

.read = bad\_pipe\_r,

.write = do\_sync\_write,

.aio\_write = pipe\_write,

.poll = pipe\_poll,

.unlocked\_ioctl = pipe\_ioctl,

.open = pipe\_write\_open,

.release = pipe\_write\_release,

.fasync = pipe\_write\_fasync,

};

我们可以看到，对读入端执行write操作，就会执行bad\_pipe\_w函数，对写入端执行read操作，就会执行bad\_pipe\_r函数。这两个函数比较简单直接返回-EBADF。

static ssize\_t

bad\_pipe\_r(struct file \*filp, char \_\_user \*buf, size\_t count, loff\_t \*ppos)

{

return -EBADF;

}

static ssize\_t

bad\_pipe\_w(struct file \*filp, const char \_\_user \*buf, size\_t count,loff\_t \*ppos)

{

return -EBADF;

}

我们只是介绍了pipe函数的用法，至今看不出来如何进行进程间通信。调用pipe之后，进程发生了什么呢？



可以看到，调用pipe函数之后，消耗了两个文件描述符，就是pipe返回的两个描述符。该进程可以往写入端文件描述符写入信息，进程就可以从读入端读出信息。可是一个进程玩管道，起不到任何通信的目的，这不是通信，这是自言自语。

如果调用pipe的函数调用了fork，创建了子进程，情况就不一样了，fork以后，子进程继承了父进程打开的文件。如下图所示，两条通信的通道就建立起来了。父进程可以往管道里写，子进程从管道里面读，也可以子进程往管道里写，父进程从管道里面读。这两条通路都是可选的，但是，不能都选。前文介绍过，管道里面是字节流，父子进程都写，都读，就会导致内容混在一起，对于读管道的一方，解析起来比较困难。常规的使用方法是父子进程一方只能写入，另一方只能读出，管道变成一个单工的通道，方便使用。如下图所示，父进程放弃读，子进程放弃写，变成父进程写入，子进程读出，成为一个通信的通道。

 

父进程如何放弃读，子进程如何放弃写？其实很简单，父进程把读端口pipefd[0]这个文件描述符关闭掉，子进程把写端口pipefd[1]这个文件描述符关闭掉就可以了，如下示例代码：

int pipefd[2];

pipe(pipefd);

switch(fork())

{

case -1:

/\*fork failed, error handler here\*/

case 0: /\*子进程\*/

close(pipefd[1]) ; /\*关闭掉写入端对应的文件描述符\*/

/\*子进程可以调用对pipefd[0]调用read\*/

break；

default: /\*父进程\*/

close(pipefd[0]); /\*父进程关闭掉读入端对应的文件描述符\*/

/\*父进程可以对pipefd[1]调用write，写入想告知子进程的内容\*/

break

}

从内核角度看，调用pipe之后，消耗了2个文件描述符，调用fork之后，子进程也就有了管道对应的两个文件描述符。和普通文件不同，两个文件描述符对应的是一块内存缓冲区域。如下图所示。

如果和示例代码一样，父进程关闭管道读出端，子进程关闭写入端，文件描述符的情况如图父子进程通信一样。

前面讲述了父子进程之间通过管道通信，如何在兄弟之间通过管道通信？父进程持有可以写入管道的文件描述符，父进程再次fork，新的子进程也可以写入管道。







如上图所示，父进程再次创建一个子进程B，子进程B就持有管道写入端，这时候两个子进程之间就可以通过管道通信了。父进程为了不干扰两个子进程通信，很自觉地关闭了自己的写入端。从此管道成为了两个子进程之间的单向的通信通道。在shell中执行管道命令就是这种情景，只不过管道描述符占用了标准输入和标准输出。后面会细细分析。

从上面的几个图上可以看出，任何两个有亲缘关系的进程，只要共同的祖先打开了一个管道，总能够通过关闭不相关的进程的某些管道文件描述符，建立起两者之间通信的单工的管道。

12.1.3 关闭未使用的管道文件描述符

前面提到过，用管道通信的两个进程，各持有一个管道文件描述符，不相干的进程自觉关闭掉这些文件描述符，原因不仅仅是使数据的流向更加清晰，也不仅仅是节省文件描述符，确保进程不会耗尽其文件描述符，更重要的原因是，关闭未使用的管道文件描述符对管道的正确使用十分重要。

管道有如下三条性质：

* 当所有的写入端关闭时，并且管道里的数据都被读出，读取端才会收到EOF标志。
* 当所有读取端都关闭时，当进程尝试再次往管道里面写入数据，内核会向写入进程发送一个SIGPIPE的信号。
* 当所有的读取端和写入端都关闭后，管道才能被销毁。

这三条性质的存在，决定了我们要关闭没用的管道文件描述符。

从管道读数据的进程，需要关闭其持有的管道写入文件描述符，这样当负责写入管道的进程写完所有要写的内容之后并且关闭写入文件描述符后，读取端读完管道中所有内容之后，就可以看到文件结束EOF（read函数返回0字节）。如果负责读的一端，不关闭写入文件描述符，就会有一个管道写入描述符泄露，当其他进程关闭了写入端的文件描述符，那么负责读的一端调用read时，将会阻塞于此，永不返回。这是因为内核维护的引用计数发现还有进程可以写入管道，但是偏偏是读管道所在的进程，这实际上形成了死锁：read返回不了，进程就没有几乎关闭管道的写入文件描述符，关闭不了管道的写入文件描述符，read就不会收到EOF，也就不会返回。

写入进程，需要关闭其持有的管道的读取文件描述符。原因是只有这样，当其他的读取端都关闭以后，该进程才能收到SIGPIPE信号，意识到不会有进程从管道里面读取数据了。SIGPIPE默认情况下会杀死一个进程，当然我们也可以捕获该信号，这样的话，write系统调用就会失败，errno是EPIPE。我们就可以及时知晓所有的读取端都关闭了。如果写入端的进程不关闭管道的读取文件描述符，哪怕其他进程都已经关闭了读取端，该进程仍可以向管道写入数据，但是只有生产者，没有消费者，最终就会写满管道，管道写满后，后续的写入请求就会被阻塞。从而再次形成死锁。

12.1.4 管道对应的内存区大小

终于提到了管道可能被写满。管道本质是一片内存区域，有大小，Linux自从2.6.11起，管道的默认大小是65536字节。可以调用fcntl来获取这个值得大小：

pipe\_capacity = fcntl(fd, F\_GETPIPE\_SZ);

这个值的大小可以修改，可以调用fcntl函数来修改管道的内存区域的大小：

ret = fcntl(fd, F\_SETPIPE\_SZ, size);

但是这个管道内存区域的大小，必须在页面大小和最大值之间，最大值记录在/proc/sys/fs/pipe-max-size里，对于特权用户，可以修改这个上限值。

➜ ~ cat /proc/sys/fs/pipe-max-size

1048576

管道容量有大小这个事实对于编程有什么影响呢？编程中，要意识到，管道有大小，写入需谨慎，不能连续地写入大量的内容，但管道满了，写入就会被阻塞；对于读取端，要及时的读取，防止管道内存耗尽，造成写入阻塞。

12.1.5 shell管道的实现

Shell中，管道大量应用，我们看到前一个命令的标准输出作为了后面一个命令的标准输入。管道是如何做到的呢。



兄弟进程可以通过管道传递消息，这不是什么意外的事情，前面已经图示了做法。关键是如何将标准输出作为管道的文件描述符，将标准输入作为管道的输入文件描述符。

答案就是复制文件描述符。

对于第一个子进程，执行dup2之后，标准输出对应的文件描述符1，也成为了管道的写入端。这时候，管道就有了两个写入端，按照前面的建议，需要关闭不相干的写入端，是读取端可以顺利的读到EOF，所以讲刚开始分配的管道写入端的文件描述符pipefd[1]关闭。

if(pipefd[1] != STDOUT\_FILENO)

{

dup2(pipefd[1],STDOUT\_FILENO);

close(pipefd[1]);

}

同样道理，对于第二个子进程，需要将标准输入复制到管道的读取端，读取管道的输出，作为标准输入。

if(pipefd[0] != STDIN\_FILENO)

{

dup2(pipefd[0],STDIN\_FILENO);

close(pipefd[0]);

}

简单来说，就是第一个子进程的标准输出绑定到了管道的写入端，从而第一个命令的输出，写入了管道，第二个子进程的管道的读取端，复制到了标准输入，从而一旦管道里面有了内容，管道的内容的内容也就成了标准输入。从而完成了shell终端的行为。

两个示例代码，为什么要判断管道的文件描述符是否等于标准输入和标准输出呢？原因是，很可能调用pipe的时候，进程已经关闭了标准输入和标准输出，pipe会分配最小的文件描述符，所以pipe的文件描述符可能等于0或者1。如果是这样的话，如果没有if判断加以保护，很可能变成了：

dup2(1,1);

close(1);

这样的话，第一行代码什么也没做，但是第二行代码就把管道的写入端给关闭了。从而没办法传递信息了。

12.1.6 与shell命令进行通信popen

管道一个重要作用是和shell命令进行通信，我们可以给shell命令提供输入，也可以获取到shell命令执行的输出。popen函数和pclose函数就是提供来做这件事情的。

就像system函数，没有system函数，我们通过fork，exec及wait家族函数也可以实现system的功能。但是system函数为我们提供了一些便利。同样道理，只用pipe函数及dup2等函数，也能实现popen函数的作用，但是popen对这种很常见的应用场景提供了便利，下面我们接口。

#include <stdio.h>

FILE \*popen(const char \*command, const char \*type);

int pclose(FILE \*stream);

popen函数会创建一个管道，并且创建一个子进程来执行shell，shell会创建一个子进程来执行command。根据type的不同，分成两种情况：

* + 如果type是r：command执行的标准输出，就会写入管道，从而被调用popen的进程读到。通过对popen返回的FILE类型指针执行read或者fgets等操作，可以读取到command的标准输出。



* + 如果type是w：调用popen的进程，可以通过对FILE类型的指针fp执行write，fputs等操作，负责往管道里面写入，写入的内容经过管道传给执行command的进程，作为命令的输入



popen函数成功时，会返回stdio库封装的FILE类型的指针，失败的时候，会返回NULL，并且设置errono。常见的失败有fork失败，pipe失败，或者分配内存失败。

IO结束了以后，可以调用pclose函数来关闭管道，并且等待子进程退出。尽管popen函数返回的是FILE类型的指针，也不应调用fclose函数来关闭popen函数打开的文件流指针，因为fclose不会等待子进程的退出。pclose函数成功时会返回子进程中shell的终止状态。popen函数和system函数很类似，如果无法command对应的命令无法执行，就如同执行了exit(127)一样。如果发生其他错误，pclose函数返回-1。可以从errno中获取到失败的原因。

下面我们给出一个简单的例子，来示范下popen的用法：

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

#include<errno.h>

#include<sys/wait.h>

#include<signal.h>

#define MAX\_LINE\_SIZE 8192

void print\_wait\_exit(int status)

{

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

if(WIFEXITED(status))

{

printf("normal termination,exit status = %d\n",WEXITSTATUS(status));

}

else if(WIFSIGNALED(status))

{

printf("abnormal termination,signal number =%d%s\n",

WTERMSIG(status),

#ifdef WCOREDUMP

WCOREDUMP(status)?"core file generated" : "");

#else

"");

#endif

}

}

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

{

FILE \*fp = NULL ;

char command[MAX\_LINE\_SIZE],buffer[MAX\_LINE\_SIZE];

if(argc != 2 )

{

fprintf(stderr,"Usage: %s filename \n",argv[0]);

exit(1);

}

snprintf(command,sizeof(command),"cat %s",argv[1]);

fp = popen(command,"r");

if(fp == NULL)

{

fprintf(stderr,"popen failed (%s)",strerror(errno));

exit(2);

}

while(fgets(buffer,MAX\_LINE\_SIZE,fp) != NULL)

{

fprintf(stdout,"%s",buffer);

}

int ret = pclose(fp);

if(ret == 127 )

{

fprintf(stderr,"bad command : %s\n",command);

exit(3);

}

else if(ret == -1)

{

fprintf(stderr,"failed to get child status (%s)\n",strerror(errno));

exit(4);

}

else

{

print\_wait\_exit(ret);

}

exit(0);

}

将文件名作为参数传递给程序，执行cat filename命令。popen调起子进程负责执行cat filename的命令，子进程的标准输出通过管道传递给父进程，父进程可以通过fgets来读取command的标准输出。

popen函数及pclose函数和system有很多相似的地方，但是也有显著的不同。调用system函数时，shell命令的执行被封装在了函数内部，所以system不返回，调用system的进程是不会继续执行的。但是popen函数不同，一旦调用popen函数，调用进程和执行command的进程处于并行的状态。pclose函数才会关闭管道，等待执行command的进程退出。换句话说，在popen之后，pclose之前，调用popen的进程和执行command的进程是并行的，这种差异带来了两种显著的不同：

* + 在这并行期间，调用popen的进程可能创建其他子进程，所以标准规定popen不能阻塞SIGCHLD信号。这也意味着，popen创建的子进程可能被提前执行的等待操作所捕获，调用pclose函数时，已经无法等待command子进程的退出，这种情况下，将返回-1，并且errno为ECHILD。
  + 调用进程和command子进程是并行的，所以标准要求，popen并不忽略SIGINT和SIGQUIT信号。如果是从键盘产生的上述信号，那么，调用进程和command子进程都会收到信号。

12.2 命名管道FIFO

上一节介绍的管道也被称为未命名管道。这种管道，因为没有实体文件与之关联，只能应用在有共同祖先的各个进程之间。在没有亲缘关系的任意两个进程之间，未命名管道就爱莫能助了（当然存在一种例外的情况，UNIX domain socket提供一种机制可以传递文件描述符，可以将管道的一个文件描述符传递给其他无亲缘关系的进程，从而使没有亲缘关系的进程也能用管道通信。这个方法我们并不细细讲述）。

命名管道就是为了解决未命名管道的这个缺点而引入的。FIFO和管道类似，最大的差别就是有实体文件与之关联。由于存在实体文件，这样不相关的没有亲缘关系的进程可以使用FIFO来实现进程之间的通信。

12.2.1 创建FIFO文件

#include <sys/types.h>

#include <sys/stat.h>

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

可以采用上面的接口创建FIFO文件，第二个参数的含义是FIFO文件的读写执行权利，和open函数类似。当然真实的读写执行权限，还需要按照当前进程的umask来取掩码，即：

real\_mode = (mode & ~umask)

除了用C接口，还可以使用命令来创建一个命名管道：

mkfifo [-m mode] pathname

pathname是创建管道文件的文件名，-m mode的使用方法和chmod的方法一样。

除此外，mknod命令也可以用来创建FIFO文件，使用方法如下：

mknod [-m mode] pathname p

末尾的p表示要创建命名管道（named pipe）。

创建出来的FIFO文件，ls –l查看，第一个字母是p，表示这是命名管道文件。

prw-rw-r-- 1 manu manu 0 2月 19 23:03 myfifo2

在shell编程中可以使用 –p file来判断是否是FIFO文件。在C语言中如何判断是否是FIFO文件呢。S\_ISFIFO宏可以判断，不过要首先通过stat或者fstat函数获取到文件的属性信息。在属性信息struct中，mode\_t类型的st\_mode可以用来判断是否是FIFO。

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

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

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

S\_ISFIFO(buf->st\_mode)

12.2.2 打开FIFO文件

一旦FIFO文件创建好了，就可以用它来实现进程间通信了。一般的文件操作函数如open，read，write，close，unlink等都可以用在FIFO文件上。推荐的使用方法两个进程一个读打开FIFO文件，另一个写打开FIFO文件。这样写进程写入FIFO的内容可以被读进程读取的到。打开一个FIFO文件，和打开普通文件相比，有不同语义。以O\_RDONLY的标志位打开一个FIFO文件时，调用进程会阻塞，直到另一进程以O\_WRONY的标志位打开该FIFO文件。同样道理，如果一个进程以O\_WRONLY的标志位打开一个FIFO文件，调用进程会阻塞，直到另一个进程以O\_RDONLY的标志位打开该FIFO文件。也就是说，打开FIFO文件会同步读取进程和写入进程。

乍看之下，读打开不能返回，在等写打开，同样写打开不能返回，在等读打开，这是死锁啊，谁都返回不了啊。事实上不是这样的。当读打开和写打开请求都到达FIFO文件时，读打开和写打开就都能返回了。

如果open操作不想阻塞呢？调用open时，使用O\_NONBLOCK标志位，如下所示：

fd = open(“fifo\_file”,O\_RDONLY | O\_NONBLOCK);

带O\_NONBLOCK标志位打开FIFO文件时：

* 如果当前是带O\_NONBLOCK标志位的读打开，
* 如果写入端已经就绪，成功返回
* 如果尚没有进程写打开FIFO文件，则仍然成功返回。
* 如果当前带O\_NONBLOCK标志位的写打开
* 如果FIFO已经被读打开，成功返回
* 如果尚没有进程读打开FIFO文件，返回-1，并设置errno为ENXIO

同样都是带O\_NONBLOCK的打开，为何没有写入端的读打开可以成功返回，而没有读取端写打开却失败返回-1？原因是，FIFO只有读取端，没有写入端，并无什么先显著的危害，所有尝试从FIFO中读取数据的操作都不会返回任何数据。反过来则不然。如果允许只存在写入端，不存在读取端，那么任何尝试向FIFO的写入，都会导致SIGPIPE信号的产生以及write调用返回EPIPE错误，所以在源头上堵住即open失败是合理的。

O\_NONBLOCK标志位不但影响打开的行为，也会影响读写FIFO文件的行为。在下一小节会介绍。如果open时忘记带上O\_NONBLOCK标志位，后面添加上该标志位，怎么办？答案是fcntl这个文件操作的瑞士军刀。

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

flags |= O\_NONBLOCK;

fcntl(fd,F\_SETFL,flags);

反过来，如果open的时候带有O\_NONBLOCK标志位，后面读写的时候，不希望带有该标志位，又当如何办？

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

flags &= ~O\_NONBLOCK;

fcntl(fd,F\_SETFL,flags);

值得一提的是，并非只有命名管道才有NON\_BLOCK标志位。Linux自2.6.27版本之后，提供了pipe2系统调用，glibc自2.9版本依赖提供了同名的API：

int pipe2(int pipefd[2], int flags);

该flags可以O\_NONBLOCK，来控制对无名管道的读写行为。

现在可以开始讲述，读写FIFO文件或者无名管道，在不同场景下的行为：

12.2.3 读写FIFO文件的行为

无名管道pipe和命名管道FIFO，在内核实现部分有很大的重叠，都属于管道文件系统（pipefs）。无名管道，分裂成了读取文件描述符，和写入文件描述符。而命名管道将两个描述符合二为一，如果读打开，就如同获取到了无名管道的读取文件描述符，如果是写打开，就如同获取到了无名管道的写入文件描述符。这种本质的一致，造成FIFO的读写控制，和无名管道的读写控制，是一模一样的。

影响管道或者FIFO文件读写行为的因素有

1 当前管道的存在的字节数p

2 是否有O\_NONBLOCK标志位

3 管道的最大容量PIPE\_BUF和要读写的字节数n的关系

4 读写端是否都存在。

根据读写，花开两朵，各表一枝，先说从FIFO或者管道读取端读

表从一个包含p字节的管道或者FIFO读取n字节的含义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 是否启用O\_NONBLOCK | p = 0 且存在写入端 | p = 0且所有写入端均已关闭 | p < n | p >= n |
| 否 | 阻塞 | 返回0 (EOF) | 读取p字节 | 读取n字节 |
| 是 | 失败(EAGAIN) | 返回0 (EOF) | 读取p字节 | 读取n 字节 |

从上表可以看出，

* O\_NONBLOCK标志位影响的仅仅是当管道为空时，读取操作的行为是阻塞，还是当即返回失败。
* 当read返回0时，表示已经遇到了EOF，并且所有的写入端都已经关闭了。这一般出现管道的使命结束时，此时，读取端也可以关闭了。

说完读，开始说写入。对于管道而言，如果一次写入的数据量不超过PIPE\_BUF个字节，那么内核会保证纵然有多个进程同时写入管道，写入的数据也不会混合。对于Linux而言，这个值是4096，一个页面的大小。这个值需要和管道的容量区分开。

* 写入字节数小于PIPE\_BUF（4096字节），确保原子写入，多进程一起往管道里面，数据也不会混合。带不带O\_NONBLOCK标志位的区别在于：
  + 不带O\_NONBLOCK，如果管道的空闲区域不足以写入n个字节，那么睡眠，等待读取端读取管道内容，直到管道能容纳n字节，开始一次性写入。
  + 带O\_NONBLOCK，发现管道空闲区域不足以写入n字节，脆败，当即返回失败，errno设为EAGAIN
* 当写入字节大于PIPE\_BUF（4096字节），不再保证原子写入。多进程往管道写入时，数据可能混合在一起。带不带O\_NONBLOCK标志位的区别在于：
* 不带O\_NONBLOCK标志位，写入n字节要做到使命必达，可能阻塞，返回时，成功返回时，所有n个字节都会写入管道或者FIFO
* 带O\_NONBLOCK标志位，尽力而为的策略，写满FIFO空闲区域后，就返回，实际写入字节数可能在1~n之间。调用者需要判断返回值，再次调用write以写入剩余的字节。

12.3 System V IPC概述

下面三种类型的进程间通信方法称为System V IPC：

* System V 消息队列
* System V信号量
* System V 共享内存

这三种IPC机制差别很大，之所以将它们放在一起讨论，一个重要的原因是这三种机制是一同被开发出来。最早出现在20世纪70年代末，1983年三者出现在主流的System V UNIX系统上，因此这三种机制被统称为System V IPC。

表System V IPC编程接口

|  |  |  |  |
| --- | --- | --- | --- |
|  | 消息队列 | 信号量 | 共享内存 |
| 头文件 | <sys/msg.h> | <sys/sem.h> | <sys/shm.h> |
| 关联数据结构 | msqid\_ds | semid\_ds | shmid\_ds |
| 创建或打开对象 | msgget() | semget() | shmget() + shmat() |
| 关闭对象 | 无 | 无 | 无 |
| 控制操作 | msgctl() | semctl() | shmctl() |
| 执行IPC | msgsnd()  msgrcv() | semop | 访问共享内存区的内存数据 |

从作用上看，三种通信机制各不相同，但是从设计和实现的角度看，还是有很多风格一致的地方。

System V IPC没有遵循一切皆文件的UNIX哲学，用标识符（id）和键值来标识一个System V IPC对象。这是System V IPC和后起之秀POSIX IPC最显著的区别。每种System V IPC都有一个相关的get调用（上表中的创建或打开对象一行），该函数返回一个整型的标识符ID，如同打开文件返回的文件描述符（整型数字），System V IPC后续函数操作都作用在标识符ID上，如同后续的文件读写函数都作用在open返回的文件描述符上。到此为止，看起来一切都好，和UNIX文件模型并无太大差别。实则不然。

**System V IPC对象作用范围是整个操作系统，内核没有维护引用计数**

调用各种get函数返回的ID是操作系统范围内的标识符，任何进程，无论是否存在亲缘关系，只要有相应的读写权限，都可以来操作System V IPC对象来达到通信的目的。System V IPC具有内核持久性。哪怕创建System V IPC对象的进程已经退出，哪怕有一段时间没有任何进程打开该对象，只要不执行删除操作（ipcrm命令或者调用控制操作函数）或者系统重启，后面启动的进程依然可以使用前人创建的System V IPC对象用来通信。对比一下无名管道，当最后一个持有管道描述符的进程退出之后，管道也就不复存在了。

**无法像操作文件一样操作System V IPC对象**

System V IPC对象在文件系统中没有实体文件与之关联。我们不能用文件相关的操作函数来访问它或者修改它的属性。所以不得不提供相关的系统调用（如msgctl，semop，shmget等）来操作这些对象。在shell中无法用ls查看存在的IPC对象，无法用rm来删除之，无法用chmod来修改它们的访问权限。好在提供了ipcs和ipcrm命令来操作该对象。由于System V IPC对象不是文件描述符，所以无法使用基于文件描述符的多路转接I/O技术：select()、poll()、epoll()。

12.3.1 标识符与IPC Key

前文介绍了System IPC靠标识符来识别和进行操作，同时标识符又是操作系统范围内的变量，这说明标识符十分关键。首先是该标识符要具有唯一性。和文件描述符不同，文件描述符是进程内有效的，一个进程的文件描述符4和另一个进程的文件描述符4可以没有任何关系。但是IPC的标识符是操作系统全局的变量，只要知道该值（哪怕是猜测获得），有相应的权限，任何进程都可以通过标识符进行进程间通信。

三种System V操作的起点都是调用get函数来获取标识符，如消息队列的get函数

int msgget(key\_t key, int msgflg);

其中第一个参数是key\_t类型，其实是一个整型的变量。IPC的get函数将key转换成相应的IPC标识符。根据IPC get函数的第二个参数oflag，有以下行为：

|  |  |  |
| --- | --- | --- |
| oflag参数 | Key不存在 | Key已经存在 |
| 无特殊标志 | 出错返回-1(ENOENT) | 成功返回0，获取到已有标识符 |
| IPC\_CREAT | 成功返回0，创建新标识符 | 成功返回0，获取到已有标识符 |
| IPC\_CREAT | IPC\_EXCL | 成功返回0，创建新标识符 | 出错返回-1(EEXIST) |

由key可以产生出IPC标识符，容易产生出一种误解，就是同一个key调用IPC的get函数总是返回同一个整型值。实际上，不是这样的。一个IPC对象在其生命周期以内，key到标识符的映射是稳定不变的，即同一个key调用get函数，总是返回相同的标识符，但是一旦key对应的IPC对象被显式删除或者系统重启，重新使用key创建的新的IPC对象被分配的标识符几乎肯定是不同的。

IPC对象在其生命周期之内，key到标识符的映射是稳定的，不同进程通过同一个key，获取到标识符，从而操作同一个System V IPC对象。那么现在问题就变成了如何选择key。

* 随机选在一个整数值作为key值，一般来说，整数值放在一个头文件中。所有使用该IPC对象的程序都要包含该头文件。需要注意的是，防止选择的key值重复，导致不须通信的进程之间意外通信，导致程序混乱。
* IPC的get函数提供了一种方法，当第一个参数值传递IPC\_PRIVATE常量作为key值，这样每次调用都会创建一个全新的IPC对象。就像手册里面提到，IPC\_PRIVATE的名字不太恰当，叫IPC\_NEW可能会更符合应用场景，因为只要传递这个参数，总是会创建新的IPC对象。这种方法的问题在于没有一个key能够让多个进程同时获取同一个IPC对象。要想多个进程同时操作IPC对象，要么父进程执行fork之前创建全新IPC对象，子进程继承IPC对象的标识符，要么IPC对象的创建进程想办法将标识符存储到一个其他进程可以获取的地方，如将标识符写入文件，其他进程通过文件直接获取到标识符。
* 使用ftok函数生成一个key。

ftok()是file to key的意思，通过pathname获取到一些信息和传入的第二个参数的低8位组合成一个整型的IPC key值，需要注意的是，pathname对应的文件必须是存在的。

#include <sys/types.h>

#include <sys/ipc.h>

key\_t ftok(const char \*pathname, int proj\_id);

这个函数在Linux上的实现是：按照给定的路径名，获取到文件的stat信息，从stat信息中取出st\_dev和st\_ino，然后结合给出的proj\_id，按照下图获取到32位的key值。



我们可以比较轻易的验证

#include <stdio.h>

#include <unistd.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <sys/ipc.h>

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

{

struct stat stat\_buf ;

if(argc != 2)

{

fprintf(stderr,"Usage : ftok <pathname>");

return 1;

}

stat(argv[1],&stat\_buf);

key\_t key = ftok(argv[1],0x12**34**);

printf("st\_dev : %lx, st\_inode : %lx , key : %x \n",

stat\_buf.st\_dev,stat\_buf.st\_ino,key);

return 0;

}

执行情况如下：

➜ ftok ./ftok ./ftok\_test.c

st\_dev : 8**01**, st\_inode : 24**0cb4** , key : **34010cb4**

在第二个参数相同的情况下，很难有两个文件映射出同一个key值。这里说很难，而不是绝对不，是因为这种情况是有可能发生的。这种冲突的出现需要满足下面三个条件：

* 两个文件所属的文件系统所在的磁盘的次设备号的低8位相同，
* 两个文件在各自的文件系统上inode的最低16位又相同
* 存在两个进程分别选择一个文件来调用ftok()来获取key值

虽然理论上是存在key冲突的可能，但是实际上，不同文件通过ftok函数产生出冲突的key值的可能性太低，除非刻意构造这种冲突，否则很难出现。因此使用ftok函数来获取key值是编程中常用的方法。

12.3.2 IPC的公共数据结构

对于System V IPC的三种机制，有很多的共性，所以从代码层面上说，有很多公共的部分。权限结构就是其中一个。IPC的权限结构至少包括如下成员：

struct ipc\_perm{

key\_t key;

uid\_t uid; /\*owner’s user ID\*/

gid\_t gid; /\*owner’s group ID\*/

uid\_t cuid; /\*creator’s user ID\*/

gid\_t cgid; /\*creator’s group ID\*/

mode\_t mode; /\*read/write permission\*/

ulong\_t seq ;

};

大部分字段都很明显。调用IPC get函数创建IPC对象时，所有的字段都会被赋值。后面可以调用msgctl，semctl或shmctl修改uid、gid和mode的值。只有IPC结构的创建者或者超级用户才能更改这些字段。

mode的值是控制读写权限的。所有的System V IPC对象都不具备执行权限，只有读写权限。其中对于信号量而言，写权限意味着修改权限。

|  |  |  |
| --- | --- | --- |
| 权限 | 标志位 | 位 |
| 用户读 | S\_IRUSR | 0400 |
| 用户写 | S\_IWUSR | 0200 |
| 用户组读 | S\_IRGRP | 0040 |
| 用户组写 | S\_IWGRP | 0020 |
| 其他读 | S\_IROTH | 0004 |
| 其他写 | S\_IWOTH | 0002 |

所以创建一个IPC对象时，一般会将设置访问权限，如下所示：

msg\_id = msgget(key,IPC\_CREATE | S\_IRUSR | S\_IWUSR |S\_IRGRP);

msg\_id = msgget(key,IPC\_CREATE | 0640);

除去这些，数据结构中的key和seq很有意思。key是比较简单，就是调用get函数创建IPC对象时传递进去的key值。如果key的值是宏IPC\_PRIVATE，实际的key值是0。

成员变量key好解释，seq就不那么好理解了。我们知道进程分配文件描述符，都是采用最小可用的算法。比如文件描述符5曾经被分配给文件A，但是很快进程关闭了文件A。如果进程尝试打开另外一个文件，此时如果5是最小可用的数字，那么新打开文件的文件描述符就是5。但是IPC对象的标识符不能采用这个算法。因为多个进程要用标识符来通信。如果采用最小可用的算法，一般来讲，IPC对象的个数不会太多，那么这个数字太容易被猜测到了。如果存在一个恶意程序，要攻击消息队列，它只需尝试0~100这样标识符，就可以偷偷取走消息队列里面的信息。

因此，Linux内核对三种System V IPC各自维护了一个seq值。每分配一个IPC对象，该对象对应的seq自加1。注意seq不是引用计数，因为删除一个对象，对应的seq并不自减。当这个seq值用来计算标识符的值。采用的算法即：

#define IPCMNI 32768

#define SEQ\_MULTIPLIER (IPCMIN)

static inline int ipc\_buildid(int id, int seq)

{

return SEQ\_MULTIPLIER \* seq + id;

}

上面公式中的id就是最小可用的值。因此，返回的ID是一个比较大的值。仍然以消息队列为例。如果开机后，消息队列为空，创建的第一个消息队列的标识符必然为0，创建的第二个消息队列和第三个消息队列的值为

32768 \* 1 +　1 = 32769

32768 \* 2 + 2 = 65538

下面我们写一个测试程序验证之：

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/msg.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <errno.h>

#include <string.h>

#define KEY 0x1234

#define PERMS (S\_IRUSR | S\_IWUSR)

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

{

if(argc != 2)

{

fprintf(stderr,"Usage: %s num\n",argv[0]);

return 1 ;

}

int num = atoi(argv[1]);

int i ;

int \* msgid\_array = malloc(num\*sizeof(int));

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

{

msgid\_array[i] = msgget(IPC\_PRIVATE, PERMS | IPC\_CREAT) ;

if(msgid\_array[i] == -1)

{

fprintf(stderr,"create msg queue failed (%s)\n",

strerror(errno));

break;

}

else

fprintf(stdout,"%-10d%-10d\n",

i,msgid\_array[i]);

}

int msgid = msgget(KEY,PERMS | IPC\_CREAT);

if(msgid == -1)

{

fprintf(stderr,"failed to get msg id (%s)\n",

strerror(errno));

return 2 ;

}

fprintf(stdout,"0x%-8x%-10d\n", KEY,msgid);

return 0;

}

该程序接受一个数字n，先生成n个IPC\_PRIVATE为键的消息队列对象。然后尝试创建或者获得一个以0x1234为键值的消息队列对象。

执行情况如下：

➜ ipc ipcs -q

--------- 消息队列 -----------

键 msqid 拥有者 权限 已用字节数 消息

➜ ipc ./id 2

0 0

1 32769

0x1234 65538

➜ ipc ipcs -q

--------- 消息队列 -----------

键 msqid 拥有者 权限 已用字节数 消息

0x00000000 0 manu 600 0 0

0x00000000 32769 manu 600 0 0

0x00001234 65538 manu 600 0 0

前三个的值的确是同我们计算得到的是一样的。现在我们来验证，一旦IPC对象被删除，用同一个key产生出来的标识符总是不相同。

➜ ipc ./id 1

0 98307

0x1234 65538

➜ ipc ipcs -q

--------- 消息队列 -----------

键 msqid 拥有者 权限 已用字节数 消息

0x00000000 0 manu 600 0 0

0x00000000 32769 manu 600 0 0

0x00001234 65538 manu 600 0 0

0x00000000 98307 manu 600 0 0

➜ ipc ipcrm -Q 0x00001234

➜ ipc ipcs -q

--------- 消息队列 -----------

键 msqid 拥有者 权限 已用字节数 消息

0x00000000 0 manu 600 0 0

0x00000000 32769 manu 600 0 0

0x00000000 98307 manu 600 0 0

➜ ipc ./id 0

0x1234 131074

➜ ipc ipcs -q

--------- 消息队列 -----------

键 msqid 拥有者 权限 已用字节数 消息

0x00000000 0 manu 600 0 0

0x00000000 32769 manu 600 0 0

0x00001234 131074 manu 600 0 0

0x00000000 98307 manu 600 0 0

我们看到用同一个key，新创建出来的IPC标识符是131074，而不是从前的那个值。那么这个值是怎么算出来的呢？最小可用的槽位是2 ，seq目前是4：

32768 \* 4 + 2 = 131074

从上面的讨论可以看出：

* IPC对象的标识符的获得和key值并没有什么直接的关系，并不存在某个公式

ID = func(key)

IPC对象的标识符取决于开机以来创建IPC对象的历史（seq）和当前可用的最小槽位的值（index），只是内核将标识符和key值关联起来。

* seq也是有上限的，当到达上限seq\_max时，将seq重新赋值为0。上限是:

(INT\_MAX/IPCMNI) = 2147483647 / 32768 = 65535

12.3.3 IPC相关的工具

在上一节，已经接触到了IPC的两个最重要的工具ipcs和ipcrm。

由于IPC不是文件，所以无法用ls，stat等操作文件的命令来获取信息。Linux提供了ipcs，来获取IPC的相关信息。

* q表示的消息队列，会显示出消息队列的键，标识符，所有者，权限，占据的字节总数和消息个数

--------- 消息队列 -----------

键 msqid 拥有者 权限 已用字节数 消息

* s表示信号量，会显示出信号量的键，标识符，拥有者，权限，及信号量集的大小。

--------- 信号量数组 -----------

键 semid 拥有者 权限 nsems

* m表示共享内存，会显示出共享内存的键值，标识符，拥有者，权限，字节数，将共享内存挂到自己虚拟地址空间的进程的个数，及状态信息。状态信息的值有一下几种可能：
* locked ：就是共享内存的页面已经锁定在RAM之中，不会被swap出去。shmctl控制函数可以调用SHM\_LOCK参数可以将共享内存锁定在RAM。
* dest ：共享内存的删除和消息队列及信号量不同。消息队列和信号量没有维护引用计数，说删就删，哪怕消息队列中还有消息，哪怕其他进程还持有标识符。共享内存不太一样，共享内存维护有持有共享内存的进程个数计数。当持有共享内存的进程个数不是0，又收到了IPC\_RMID的指令，操作系统采取的响应并不是粗暴的说删就删。内核将共享内存的状态置为SHM\_DEST，将key的值设置为IPC\_PRIVATE，即0x00000000。当持有共享内存进程个数为0时，真正销毁共享内存。是否允许标记为SHM\_DEST状态的共享内存被attach到新的进程呢？网上很多资料说IPC\_RMID之后，不允许进程attach处于dest状态的共享内存。事实上这种说法是不对的。Linux并没有做这种限制。这是Linux和大多数UNIX实现不同的一个地方。

------------ 共享内存段 --------------

键 shmid 拥有者 权限 字节 nattch 状态

ipcrm是用来清除相应IPC对象的。可以根据键值来删除，也可以根据标识符来删除。

* 删除消息队列

ipcrm -q id 或者 ipcrm –Q key

* 删除信号量

ipcrm -s id 或者 ipcrm –S key

* 删除共享内存

ipcrm -m id 或者 ipcrm –M key

本节介绍了System V IPC的一些特点。下面三个小节，对消息队列，信号量和共享内存进行分别介绍。

12.4 System V 消息队列

前面介绍管道和FIFO都是字节流的模型，这种模型不存在记录边界。如果从管道里面读出100个字节，你无法确认这100个字节是单次100字节写入的还是10次10字节写入的，你也无法知晓这100字节是几个消息。管道或者FIFO里面数据如何解读完全取决于写入进程和读取进程的约定。

为了构建更结构化的消息，管道和FIFO的写入者和读取者必须约定一些手段：

* 分隔字符：多个消息之间以分隔字符来分隔
* 固定长度：无论消息长短，每条消息占用的字节数为约定好的固定值
* 显式长度：每条记录前，加上其长度作为消息的头。

从这个角度上讲，System V消息队列和POSIX消息队列都是优于管道和FIFO的。原因是消息队列机制中，读者和写者是通过消息来通信的，无需花费精力从字节流中解析出消息。

System V消息队列比管道或者FIFO优越的第二个地方在于每条消息都有type字段，消息的读取进程可以通过type字段选择自己感兴趣的消息，也可以根据type字段来实现按消息的优先级读取，而不需要死板地按照消息生成的顺序来读取消息。

内核会为每一个System V消息队列分配一个msg\_queue类型的结构体，很多书提到的msqid\_ds结构体已经废弃的过时的，在include/linux/msg.h依然可以找到其定义，仅仅是为了向后兼容。

struct msg\_queue {

struct kern\_ipc\_perm q\_perm;

time\_t q\_stime; /\* 上一次 msgsnd的时间\*/

time\_t q\_rtime; /\* 上一次 msgrcv的时间 \*/

time\_t q\_ctime; /\* 属性变化时间 \*/

unsigned long q\_cbytes; /\* 队列当前字节总数\*/

unsigned long q\_qnum; /\*队列当前消息总数\*/

unsigned long q\_qbytes; /\*一个消息队列允许的最大字节数\*/

pid\_t q\_lspid; /\*上一个调用msgsnd的进程ID\*/

pid\_t q\_lrpid; /\*上一个调用msgrcv的进程ID\*/

struct list\_head q\_messages;

struct list\_head q\_receivers;

struct list\_head q\_senders;

};

大部分都是字段的含义都是比较好理解的，稍微有点难理解的两个字段是q\_ctime和q\_qbytes：

* q\_ctime，字段含义是属性的变化时间。当消息队列创建时，将当前时间赋给q\_ctime，另外当调用msgctl函数的IPC\_SET来改变消息队列属性时，将当前时间赋值给q\_ctime
* q\_qbytes表示单个消息队列允许的字节总数，即所有消息队列中所有消息的消息体占用的字节总数不得超过该q\_qbytes。创建消息队列时，初始化为MSGMNB。可以查看/proc/sys/kernel/msgmnb来获取其默认值。随着消息队列里面消息越来越多，当前字节总数q\_cbytes可能越来越大，但是q\_cbytes不能超过q\_qbytes的值。

12.4.1 创建或打开一个消息队列

消息队列的创建和打开是由msgget函数完成的，成功后，获得消息队列的标识符。

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgget(key\_t key, int msgflg);

msgget函数的两个参数在12.3节已经详细讲述过了，就不再赘述。当调用成功时，返回消息队列的标识符，后续的msgsnd、msgrcv，msgctl都用该标识符来操作消息队列。当函数调用失败时，返回-1，并且设置相应的errno。常见的errno如下表所示：

|  |  |
| --- | --- |
| errno | 说明 |
| ENOENT | 对应key值得消息队列不存在，并且没有设定IPC\_CREAT标志位 |
| EACCES | 存在key值对应的消息队列，但是没有权限打开消息队列 |
| EEXIST | 存在key值对应的消息队列，但是同时设置了IPC\_CREAT和IPC\_EXCL标志位 |
| ENOSPC | 需要创建消息队列，但是超过了系统允许创建消息队列的上限。上限值为MSGMNI |
| ENOMEM | 需要创建消息队列，但是系统已经没有足够的内存。 |

创建消息队列，一个很明显的问题是，操作系统到底允许创建多少个消息队列？

上表当errno等于ENOSPC时，提到上限值是MSGMNI，这个上限记录在/proc/sys/kernel/msgmni中：

➜ ~ cat /proc/sys/kernel/msgmni

3970

操作系统会根据系统的硬件情况主要是内存大小，计算出一个合理的上限值。在我的系统上只能创建出3970个消息队列。可以用上一节的id程序验证之。

➜ ipc ipcs -q |grep 0x |wc -l

3970

➜ ipc ./id 1

create msg queue failed (No space left on device)

0x1234 121572990

我们可以通过修改/proc/sys/kernel/msgmni让系统支持更多，比如传递100000给msgmni，但是实际上，操作系统有硬上限32768，哪怕设置的值大于32768，操作系统最多也只能创建32768个消息队里。

echo 100000 > /proc/sys/kernel/msgmni

➜ ipc cat /proc/sys/kernel/msgmni

100000

➜ ipc ./id 100000

……

28797 1073741823

create msg queue failed (No space left on device)

0x1234 121572990

➜ ipc ipcs -q |grep 0x |wc -l

32768

然后是对消息队列进行初始化，其中q\_qbytes是一个系统的限制，表示本消息队列所有消息的总字节数的上限值。可以修改，默认值保存在/proc/sys/kernel/msgmnb。

msq->q\_stime = msq->q\_rtime = 0;

msq->q\_ctime = get\_seconds();

msq->q\_cbytes = msq->q\_qnum = 0;

msq->q\_qbytes = ns->msg\_ctlmnb;

msq->q\_lspid = msq->q\_lrpid = 0;

INIT\_LIST\_HEAD(&msq->q\_messages);

INIT\_LIST\_HEAD(&msq->q\_receivers);

INIT\_LIST\_HEAD(&msq->q\_senders);

之所以强调q\_qbytes，是因为这个值会影响msgsnd的行为。如果消息队列当前字节总数加上msgsnd要发送的字节会大于q\_qbytes，msgsnd函数就会阻塞，等待消息的消费者取走消息，或者当设置了IPC\_NOWAIT，msgsnd会返回EAGAIN。

12.4.2 发送消息

获取到消息队列的标识符之后，我们可以调用msgsnd函数向队列中插入消息。内核会负责将消息维护在消息队列中，等待另外进程来取走消息，完成通信的全过程。现在问题就来了，要想调用msgsnd函数，我们应该讲消息封装成什么样子？

struct msgbuf {

long mtype; /\* message type, must be > 0 \*/

char mtext[1]; /\* message data \*/

};

需要将消息体封装成msgbuf一样的结构体。

第一个字段是mtype，必须大于0。这个字段是非常有用的。有了这个值，我们可以实现：

* 消息的优先管理：比如当前实现采用的，mtype值越小，优先级越高
* 消息的各取所需：考虑生产者消费者模型，假如消息存在多个的消费者进程，我们可以通过设置不同的type，让不同消费者进程选择各自关心的消息体。

下面看下msgsnd函数的接口：

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg);

使用这个接口的时候，我们需要定义自己的msgbuf结构体：

struct msgbuf{

long mtype ;

char mtext[1]

}mbuf;

按照如下步骤完成消息的发送

1. 分配一个msgbuf类型的结构体变量。长度为sizeof(mbuf) + sizeof(mymessage);
2. 将消息mymessage拷贝到mbuf->mtext中去
3. 设置消息类型mbuf->mtype
4. 调用msgsnd函数，第二个参数msgp为mbuf的地址，第三个参数msgsz为mymessage的长度
5. 释放mbuf

其中最后一个参数为标志位，目前只提供了一种标志位IPC\_NOWAIT。这个标记位的作用是当消息队列满时，控制msgsnd函数的行为：

* 当没有设置IPC\_NOWAIT标志位时，msgsnd函数阻塞
* 当设置了IPC\_NOWAIT标志位时，msgsnd函数立刻失败返回-1，设置errno为EAGAIN。

那么问题就来了，什么叫做消息队列满？

* 本消息队列中所有消息体长度总和加上当前消息的长度大于允许的上限q\_qbytes

msgsz + msq->q\_cbytes > msq->q\_qbytes

* 本消息队列中所有消息的个数加上1，大于允许的上限q\_qbytes。

1 + msq->q\_qnum > msq->q\_qbytes

两个条件是或的关系，满足一个，表示消息队列满。其中第一个条件比较容易理解，即，所有消息的字节数太多，超过了上限。反倒是第二个不那么明显。如果有太多的空消息，虽然消息体的字节总数没有到达上限q\_qbytes，但是消息的个数达到了q\_qbytes，这种情况下，也会被判定为消息队列满。

下面分析下函数的返回值和常见错误。msgsnd函数不同于文件的write函数，write函数操作的是字节流，存在部分成功的概念，所以成功时，返回的是写入的字节个数，但是msgsnd函数操作的是封装好的消息，只有成功和失败，不存在部分成功的概念。所以当成功时，msgsnd函数返回0，失败时，msgsnd函数返回-1，并且设置errno。

|  |  |
| --- | --- |
| Errno | 说明 |
| EACCESS | 无相应权限 |
| EAGAIN | 消息队列已满，并且设置了IPC\_WAIT标志位 |
| EIDRM | 消息队里已经从系统中删除，不复存在 |
| EINTR | msgsnd被信号中断 |
| EINVAL | 标识符无效，mtype小于1，msgsz小于0或者大于上限MSGMAX |

出错的情形中有几个是非常有意思的。

首先说EINVAL，当msgsz不是无限制的，最大值是MSGMAX，记录在/proc/sys/kernel/msgmax里面，这个值可以调整。内核没有做进一步的限制，所以最终会依赖系统的内存大小。

➜ linux-3.2.44 cat /proc/sys/kernel/msgmax

8192

第二个是EIDRM。这是消息队列和信号量共同的缺陷。就是当年用着消息队列的时候，另外一个进程可能删除了消息队列，操作系统的行为是说删就删，这时候，msgsnd就会收到EIDRM的errno。删除消息队列是一个难点，多进程程序要由确定谁是最后一个访问消息队列的进程，由它负责删除消息队列。事实上是很难的。

第三个是EINTR。msgsnd函数可能会阻塞，因为有很大可能被信号中断，但是msgsnd函数不会自动重启，哪怕设置了SA\_RESTART标志。

如果一切正常，没有错误，内核的行为是：

msq->q\_lspid = task\_tgid\_vnr(current);

msq->q\_stime = get\_seconds();

if (!pipelined\_send(msq, msg)) {

/\* no one is waiting for this message, enqueue it \*/

list\_add\_tail(&msg->m\_list, &msq->q\_messages);

msq->q\_cbytes += msgsz;

msq->q\_qnum++;

atomic\_add(msgsz, &ns->msg\_bytes);

atomic\_inc(&ns->msg\_hdrs);

}

pipelined\_send函数是用来查找是否存在进程等待本消息，因为消息的接收进程会尝试接收某种mtype的消息，如果不存在，消息的接收进程可能会阻塞，如果本条消息的mtype恰好满足条件，那么消息可以不用进入消息队列，直接唤醒接收进程，把消息转给它，皆大欢喜。如果不存在等待本消息的进程，不得不进入消息队列，等待有缘人来接收。

12.4.3 接收消息

有发送就要有接收，没有接收者的消息是没有意义的。System V消息队列用msgrcv函数来接收消息。

ssize\_t msgrcv(int msqid, void \*msgp, size\_t maxmsgsz,

long msgtyp,int msgflg);

接收端也要定义msgbuf类型的结构体，和发送端一样。操作步骤如下：

1. 分配一个msgbuf类型的结构体，注意确保mtext的空间足够容纳消息体
2. 将msgbuf的地址作为第二个参数msgp，由于事先不知道消息体的大小，第三个参数maxmsgsz要确保足够大，可以是约定好的消息的最大长度。

第四个参数是消息队列的精华，由于这个参数的存在，读取消息的顺序无需和发送顺序一致，进而可以演化出很多用法。

|  |  |
| --- | --- |
| msgtyp | 动作 |
| 0 | 从消息队列中取出第一条消息 |
| > 0 | 有MSG\_EXCEPT标志位:从消息队列中取出mtype等于msgtyp的第一条消息  无MSG\_EXCEPT标志位:从消息队列中取出mtype不等于msgtyp的第一条消息 |
| < 0 | 从消息队列中取出mtype最小，并且值小于或等于msgtyp绝对值的第一条消息 |

当msgtyp小于0时，相当于将消息队列当成优先队列来处理，mtype的值越小，优先级越高。

当msgtyp大于0时，能够实现多个进程从消息队列中选择自己感兴趣的消息，并行不悖，互不干扰。比如让各个进程选择mtype和自己进程ID一致的消息。

第五个参数是可选标志位。msgrcv函数有3个标志位可选

* IPC\_NOWAIT：如果消息队列中不存在满足msgtyp要求的消息，默认情况是阻塞等待，但是一旦设置了IPC\_NOWAIT标志位，立即返回失败，并且设置errno为ENOMSG。
* MSG\_EXCEPT：这个标志位时Linux特有的，只有当msgtyp大于0时才有意义，含义是选择mtype != msgtyp的第一条消息
* MSG\_NOERROR：前面也提到，实现并不知道消息体的大小，尽管要求maxmsgsz要尽可能大，但是仍然存在maxmsgsz小于消息体大小的可能。如果发生这种情况，默认情况是返回错误E2BIG，但是如果设置了MSG\_NOERROR标志位，情况就不同了，将消息体截断返回。

函数调用成功时，返回消息体的大小，失败是返回-1，并且设置errno。大部分error情况和msgsnd函数类似，比较特别的是E2BIG和ENOMSG，都已经讨论过了，不再赘述。另外msgrcv函数如果被信号中断，也不会重启系统调用，哪怕设置了安装信号时设置了SA\_RESTART标志位。

12.4.4 控制消息队列

msgctl函数可以控制消息队列的属性。

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);

函数的功能取决cmd字段：

|  |  |
| --- | --- |
| cmd | 描述 |
| IPC\_STAT | 获取消息队列的属性信息 |
| IPC\_SET | 设置消息队列的属性 |
| IPC\_RMID | 删除消息队列 |

**IPC\_STAT**

为了获取属性信息或者设置属性，必须要有一个用户态的数据结构来描述消息队列的属性信息，这个数据结构就是msqid\_ds结构体，大部分字段和内核的msg\_queue结构体对应。注意，msgid\_ds结构体包含下面的成员变量，实际的定义并非严格相同。在编程中，只要包含了头文件，可以直接使用该结构体。

struct msqid\_ds {

struct ipc\_perm msg\_perm; /\* Ownership and permissions \*/

time\_t msg\_stime; /\* Time of last msgsnd(2) \*/

time\_t msg\_rtime; /\* Time of last msgrcv(2) \*/

time\_t msg\_ctime; /\* Time of last change \*/

unsigned long \_\_msg\_cbytes;

msgqnum\_t msg\_qnum;

msglen\_t msg\_qbytes;

pid\_t msg\_lspid; /\* PID of last msgsnd(2) \*/

pid\_t msg\_lrpid; /\* PID of last msgrcv(2) \*/

};

struct ipc\_perm {

key\_t \_\_key; /\* Key supplied to msgget(2) \*/

uid\_t uid; /\* Effective UID of owner \*/

gid\_t gid; /\* Effective GID of owner \*/

uid\_t cuid; /\* Effective UID of creator \*/

gid\_t cgid; /\* Effective GID of creator \*/

unsigned short mode; /\* Permissions \*/

unsigned short \_\_seq; /\* Sequence number \*/

};

在使用，我们可以通过下面简单的代码获取到消息队列的属性：

strutct msqid\_ds buf ; /\*注意包含头文件\*/

msgctl(mid,IPC\_STAT,&buf); /\*省略error handle\*/

printf(“current # of messages in queue is %d\n”,buf.msg\_qnum);

**IPC\_SET**

消息队列开放出来了4个属性可以设置。

* msg\_perm.uid
* msg\_perm.gid
* msg\_perm.mode
* msg\_qbytes

设置方法一般是首先调用IPC\_STAT获取到当前的设置，然后修改4个属性中的某个或者某几个属性，然后调用IPC\_SET，如下所示：

strutct msqid\_ds buf ; /\*注意包含头文件\*/

msgctl(mid,IPC\_STAT,&buf); /\*省略error handle\*/

buf.msg\_qbytes = NEW\_VALUE;

msgctl(mid,IPC\_SET,&buf);

**IPC\_RMID**

删除标识符对应的消息队列，只要有权限，脆删，说删就删，立刻就删。消息队列中的所有消息都会被清除，相关的数据结构被释放，所有阻塞的msgsnd函数和msgrcv函数会被唤醒，返回EIDRM错误。

12.5 System V信号量

信号量，又被称为信号量集，事实上，信号量集这个名称更符合，下面会有讨论。信号量的作用和消息队列不太一样，消息队列的作用是进程之间传递消息。信号量的作用是为了同步多个进程的操作。比如共享内存被多个进程挂载（attach），多个进程同时修改可能会导致数据的不一致。这就需要同步。

信号量的值为一个非负的值，表示当前可用的资源的数目，当使用信号量同步资源使用时，多个进程协同工作场景如下：

* 首先创建一个信号量来控制资源的使用，其初始值设置为可用资源的数目；
* 当一个进程申请资源使用时，检测控制该资源的信号量的值：
* 如果信号量值大于0，表示有资源可用，进程将信号量的值减1，表示它使用了1个单位的资源。须知检测和递减的操作必须是原子的。
* 如果信号量值等于0，申请资源的进程陷入阻塞。当有其他进程释放资源，使信号量的值大于0，进程被唤醒，再次申请资源。
* 当一个进程资源使用完毕，需要将信号量加1，表示归还资源。所有阻塞在资源上的进程都会收到通知，重新竞争资源。

最常见的信号量当属二元信号量，只有一个资源，当资源可用，信号量的值为1，当资源不可用，信号量的值为0。和线程的互斥量（mutex）十分类似。这种场景虽然是信号量最简单的场景，事实上，它也是使用最多的一种场景。

和二元信号量相比，System V信号量走的更远，在两个维度上做了扩展：

首先将某种资源的数目扩展开来。如果某种资源的个数不是1，而是更多。事实上，System V信号量对信号量管理的资源个数设置了上限SEMVMX，这个值是32768。

其次再将资源的种类扩展一下。System V信号量可以管理多种资源，比如第一种资源总数是5，第二种资源的总数是10。使用过程中选择申请哪种资源或者哪几种资源。正因为此，所以称信号量为信号量集更合适。就这点来讲，System V信号量有些过度设计。通常情况下，一个程序只会操作一个信号量，管理一种资源，同时管理多种资源，操作集合中的多个信号量的能力是多余的。这种扩展，增加了接口的复杂度。

除了接口复杂，难以使用外，信号量还有其他的缺陷：

创建信号量（semget）和对其赋初始值（semctl）分开进行，不能原子地创建并赋初始值。这种行为可能会导致竞争条件。可以设想如下场景：当信号量创建成功尚未赋初始值时，另外一个进程semget获取到信号量集，直接执行semop操作。如果多个进程绝对平等地操作信号量，这种场景就不得不考虑，并提供解决方案。如果能够确保，创建并初始化信号量的进程先执行，后面才有信号量的使用者启动，就不需要考虑这种场景了。

持有信号量的进程异常终止时，没有机会释放已经分配给它的资源。这种情况是最糟糕的。试想二元信号量，持有资源的进程异常退出了，其他进程再也没有机会申请到资源了。如果不人工介入，很难恢复。原因就在于内核没有维护引用计数，并没有详细的记录进程与信号量的之间的关联关系，因此进程退出时，无法释放相关资源。这应该算是System V信号量的阿克琉斯之踵了。

为了缓解这个致命缺陷，又设计出了信号量撤销值SEM\_UNDO。如果进程在调用semop申请或者释放资源时，设置了SEM\_UNDO标志位，内核会帮忙记录进程在信号量上的调整的总和semadj，一旦进程退出，无论正常退出还是异常退出，内核负责将信号量的当前值减去这个总和。这种补救看上去，很美好，实际上也没有想象的那么完美。

12.5.1 创建或打开信号量

信号量的get函数为semget函数。

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

int semget(key\_t key, int nsems, int semflg);

这个接口比较简单，第二参数nsem表示信号量集中信号量的个数。换句话说，就是要控制几种资源。大部分情况下，我们希望控制一种。如果我们仅仅是访问已经存在的信号量集，可以将nsem指定为0。

第三个参数semflag有IPC\_CREAT和IPC\_EXCL，行为在12.3节已经详细介绍过了，不赘述。

从接口上看，创建信号量时，并没有传递初始值。从可移植性的角度来说，程序员必须调用semctl函数的SET\_VAL命令或者SETALL命令来初始化信号量的值。由于要有两个系统调用完成创建和初始化，就会带来竞争条件：其他进程可能在信号量已经创建好，但是尚未初始化的时候，调用semop函数，申请资源或者释放资源。为了防止这种情况的出现，一个方法是判断sem\_otime，创建信号量时，sem\_otime确定被初始化成0，但是一旦调用了semctl的SET\_VAL或者SETALL命令，或者执行semop，都会更新sem\_otime为一个时间值。对于信号量的使用者（非创建者），首先判断sem\_otime，确定sem\_otime不等于0以后，方可调用semop操作信号量。

但是Linux的semget系统调用已经将信号量的值初始化为0。如果程序不牵扯可移植性，可以无需考虑调用semctl初始化为0。

创建信号量，从在系统的限制。比如最多可以创建多少个信号量集，每个信号量集最多可以管理多少个信号量，整个操作系统层面，所有的信号量集最多可以分配多少个信号量。

SEMMNI：系统最多可以分配多少个信号量集。

SEMMSL：每个信号量集最多可以分配多少个信号量

SEMMNS：所有信号量集的最多可以由多少个信号量。

SEMOPM：每次调用能够执行的操作数。

/proc/sys/kernel/sem中按顺序记录了SEMMSL、SEMMNS、SEMOPM和SEMMNI的值。可以看出，不做调整的情况下，最多可以创建128个信号量集，每个信号量集最多管理250个信号量，全系统最多可以由32000个信号量。

➜ ~ cat /proc/sys/kernel/sem

250 32000 32 128

限制可以进一步放大，但是全系统最多可以支持32768个信号量集，每个信号量集最多可以拥有65536个信号量（考虑到struct sembuf中sem\_num成员变量是unsigned short类型，决定了65536之后的信号量无法调用semop进行调整信号量的值）。

12.5.2 操作信号量

有了信号量集，对其中一个或多个信号量的操作有semop函数实现。

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

int semop(int semid, struct sembuf \*sops, unsigned nsops);

其中第二个数据结构sembuf结构体是生面孔，该结构体的定义在sys/sem.h中，一般来说，该结构体至少包含以下三个成员变量：

struct sembuf {

unsigned short int sem\_num ;

short sem\_op ;

short sem\_flg;

}

sem\_num：解决的是which的问题。因为信号量集中可能存在多个信号量，需要用这个参数来告知semop函数要操作那个信号量，0表示第一个信号量，1表示第二个信号量，依次类推，最大为nsems -1，不得大于或等于semget创建信号量时传递的第二个参数nsems。

一般来讲，不采用下面方法来初始化sembuf，因为考虑到可移植性，我们并没有十足的把握sembuf结构体的定义和上面给出定义，顺序一致。

struct sembuf myopbuf = {1,-1,0}

不过Linux的定义就是上面给出的定义，不考虑可移植性，可以放心采用上面的方法，比较合理的方法应该为：

struct sembuf myopsbuf[1] ；

myopsbuf[0].sem\_num = 1;

myopsbuf[0].sem\_op = -1;

myopsbuf[0].sem\_flg = 0 ;

semop函数每次操作一组信号量，每个信号量由一个sembuf表示，修改一个信号量最好也要定义成struct sembuf ops[1]这样的数组，semop函数的第三个入参表示的就是要操作的信号量的个数，即sembuf数组的长度。

如果调用semop函数同时操作多个信号量，会被原子地执行，要么内核完成所有操作，要么内核什么也不做。

下面介绍sembuf结构体中sem\_op的含义，首先介绍几个术语

semval：信号量的值，即struct sem里面的semval成员变量。表示当前可用的资源个数，永远非负。

semzcnt ：等待信号量值变成0的进程的个数。以前这个值是内核struct sem的成员变量，当前Linux实现，每次调用semctl获取这个值时，内核会计算一下。

semncnt： 等待信号量的值大于当前值的进程个数。白话说就是申请资源没申请到，陷入阻塞的进程的个数。

|  |  |
| --- | --- |
| sem\_op | 含义 |
| >0 | 释放资源，将对应信号量的semval的值加上sem\_op。如有其他进程等待资源，将进程唤醒 |
| = 0 | 调用进程等待信号量值semval变成0。  如果semval等于0，立即返回  如果semval不等于0：   * 指定了IPC\_NOWAIT立即返回失败，errno为EAGAIN * 未指定IPC\_NOWAIT，陷入阻塞。semzcnt的值加1   醒来一般有三种情况   * 信号量的值变成了0，成功返回，semzcnt个数减少1个 * 信号量被删除，返回失败，errno为ERMID * 进程捕捉到信号，系统调用被中断，errno为EINTR，semzcnt个数减少1个 |
| < 0 | 申请资源。  若信号量的值不小于sem\_op的绝对值，表示资源足够用，信号量的值semval减掉sem\_op的绝对值，成功返回。  如果信号量的值小于sem\_op的绝对值：   * 指定了IPC\_NOWAIT，立即返回失败，errno为EAGAIN * 未指定IPC\_NOWAIT，陷入阻塞，semncnt的值加1   一般有三种情况可以醒来：   * 信号量的值变的不小于sem\_op的绝对值，成功返回 * 信号量被删除，返回失败，errno为ERMID * 进程捕捉到信号，系统调用被信号中断，errno为EINTR。 |

到目前为止，semop已经足够复杂了，这还没有考虑SEM\_UNDO标志位。

由于内核之中并没记录进程操作信号量的信息，就会造成进程退出，信号量的值不会发生变化。对于持有资源，没来得及释放，就异常退出的话，该信号量可能会处于不能恢复的境地。为了弥补设计的这种缺陷，提供了SEM\_UNDO标志位。所有带有SEM\_UNDO标志位的semop操作，内核都会记录下信号量操作的效果，进程退出以后，撤销这个操作。如果在进程生命周期以内，频繁的调用semop函数，内核无需将每一笔变化都记录下来，只需要记录进程在一个信号量上使用SEM\_UNDO操作做出的调整总和即可，这个总和被称为信号量调整至semadj。进程退出时，将信号量的值加上semadj。

当sem\_flg设置了SEM\_UNDO标志位之后，semadj的调整如下：

|  |  |
| --- | --- |
| sem\_op | semadj的调整 |
| >0 | semadj = semadj - sem\_op |
| <0 | semadj = semadj + abs(sem\_op) |

客观来讲，SEM\_UNDO是很有用的，尤其是二元信号量，因为一个进程退出了，不应继续占有资源导致其他进程不能工作。

但是SEM\_UNDO也并不是包治百病的良药。信号量是管理资源的，本身并无实际含义，如果进程异常退出，资源并没有释放出来，处于一个不可用的状态，单单调整信号量的值，并不能使应用恢复到一个稳定一致的状态。

另外某些情况下，进程终止时，无法严格的按照semadj进行调整：

1. 信号量初始值时0
2. A进程将信号量增加2，并且设置了SEM\_UNDO标志位为
3. B进程将信号量减去1。信号量的值变成了1
4. A进程退出

按照逻辑，应该将当前信号量的值减去2。但是由于信号量的值是1，不可能减去2。对于此困境，Linux采用的办法是尽可能地减小信号量的值。对于本例，就是将信号量的值减少为0。

12.5.3 控制信号量

控制信号量的函数为semctl函数：

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd,/\* union semun arg\*/);

某些特定的操作需要第四个参数，第四个参数是联合体，很不幸的是这个联合体需要程序员自己定义。

union semun {

int val;

struct semid\_ds \*buf;

unsigned short \*array;

struct seminfo \*\_\_buf; /\*Linux特有的\*/

};

**IPC\_RMID**

第二个参数被忽略，立即删除信号量集。所有阻塞在semop函数上的进程被唤醒，返回错误，errno为ERMID。

**IPC\_STAT**

获取信号的信息，存放在union semun中的buf指向的结构体。

其中结构体semid\_ds无需自己定义，至少包含以下成员：

struct ipc\_perm sem\_perm;

time\_t sem\_otime;

time\_t sem\_ctime;

unsigned long sem\_nsems;

可以使用下面简单的代码获取到上述信息（省略错误处理）：

struct semid\_ds ds ;

union semun arg; /\*需确保semun联合体已经定义\*/

arg.buf = ds ;

semctl(semid,0,IPC\_STAT,arg);

printf(“last op time is %s\n”,ctime(&(ds.sem\_otime)));

**IPC\_SET**

通过union semun arg的成员变量buf，来设置sem\_perm.uid, sem\_perm.gid, sem\_perm.mode等。

**GETVAL**

返回信号量集第semnum个信号量的值，无需第四个参数。

**SETVAL**

将信号量的第semnum个信号的值设置为arg.val。

union semun arg；

arg.val = 0;

semctl(semid,0,SETVAL,arg);

**GETALL**

将信号量集中所有信号的值存放在第四个参数arg的成员变量array中。确保array数组有足够的空间存放。这个操作将忽略第二个参数semnum。

**SETALL**

用第四个参数arg的成员变量array数组中的值初始化信号量集中的所有信号量。一般来说这个操作用在信号量的初始化上，正常使用期间很少调用SETALL。

需要注意的是如果调用了SETVAL或者SETALL，使用信号量的所有进程的semadj都会清零。

下面3个操作返回单个信号量的关系，并不需要第四个参数：

**GETPID**

返回上一个执行对第semnum信号量执行semop的进程的进程ID，如果不存在，则返回0。

**GETNCNT**

等待在第semnum信号量的值增长的进程个数。

**GETZCNT**

等待第semnum个信号量的值变成0的进程的个数。

12.6 System V 共享内存

共享内存允许多个进程共享一块存储空间。典型的应用过程如下

最初由一个进程大公无私，调用shmget函数创建出指定大小的共享内存区域，返回一个共享内存的标识符。

其次其他进程可以通过shmget获取到共享内存的标识符，再根据标识符，调用shmat函数可以将这块共享的内存区域附接（attach）到调用进程的虚拟地址空间。shmat函数的返回值是一个虚拟地址addr，该地址就是最初大公无私进程分配的共享内存的起点的指针。从此多个进程都可以修改这块内存或者读取这块内存。

当进程不需要再访问共享内存时，调用shmdt函数将共享内存与进程分离，从此之后，进程就无法继续使用这块内存了。这一步不是必须的，进程退出时，会自动分离。

当所有进程都不需要使用这块共享内存了，调用shmctl的IPC\_RMID命令将其销毁。共享内存的IPC\_RMID与消息队列及信号量不同，内核会检查是否还有进程使用该共享内存，如果有，仅仅是将共享内存标记为dest，并不真正删除。只有当所有使用共享内存的进程都执行过shmdt，共享内存的引用计数为0，才真正销毁。

12.6.1 创建或者打开共享内存

shmget函数负责创建或者打开共享内存段。

#include <sys/ipc.h>

#include <sys/shm.h>

int shmget(key\_t key, size\_t size, int shmflg);

size必须是正整数，表示要创建的共享内存的大小。内核以页面大小的整数倍来分配共享内存的，因此，实际size会被向上取整为页面大小的整数倍。

如果没有设置IPC\_CREAT标志位，第二个参数对段无实际意义，但是必须小于或者等于共享内存的大小，否则会有EINVAL错误。

创建共享内存，系统存在一些限制。

* + SHMMNI：系统所能创建的共享内存的最大个数。可以修改，但是最大不得超过32768。

➜ ~ cat /proc/sys/kernel/shmmni

4096

* SHMMIN：一个共享内存段的最小字节数。这个限制被定义成了1，无法修改。在内核实现中，纵然共享内存的大小为1字节，也是分配一个页面的大小
* SHMMAX：一个共享内存段的最大字节数。 可以修改，实际上限取决于可用的RAM和swap空间。

➜ ~ cat /proc/sys/kernel/shmmax

33554432

* SHMALL：系统中，共享内存占用页面的总数。这个值可以修改，取决于可用的RAM和swap空间。

➜ ~ cat /proc/sys/kernel/shmall

2097152

12.6.2 使用共享内存

创建共享内存是为了多个进程使用这块内存，达到信心交流，传递消息的目的。shmat函数可以将shmid标识的共享内存附接到调用进程的虚拟地址空间。

#include <sys/types.h>

#include <sys/shm.h>

void \*shmat(int shmid, const void \*shmaddr, int shmflg);

第二个参数shmaddr标识希望将共享内存附接到地址，一般来讲这个参数为NULL，表示内核来选择一个合适的地址。这是附接一个段的通用方法，增强了代码的可移植性。

如果第二个参数不是NULL，并且没有设置SHM\_RND标志位，表示用户要铁了心地将共享内存附接到shmaddr这个地址处。此时这个地址须是系统分页大小的整数倍，否则会出错返回，errno被设置为EINVAL。如果很多情况下，纵然shmaddr已经按照系统页对齐，也不会附接成功，比如该处地址已经被使用。

如果第二个参数不是NULL，并且设置了SHM\_RND标志位，那么这个shmaddr地址不是系统分页的整数倍也无妨，内核会负责将shmaddr向下取整到SHMLBA（shared memory low boundary address）的整数倍。SHMLBA，一般是系统页的整数倍，是个体系相关的常量，在x86系统上，该值等于系统页的大小。

为第二个参数指定非NULL的值，是不建议的行为：首先它降低了程序的可移植性，在一个系统上有效，移植到另一个系统上可能会失败；其次它增大了系统调用返回失败的概率，因为选择地址很可能已经被使用。

第三个参数为标志位：

SHM\_RDONLY：附接共享内存，但是只能只读访问，不能修改共享内存的内容。如果不指定SHM\_RDONLY，表示默认情况，即可读可写。

SHM\_RND：如果第二个参数shmaddr地址不是NULL，将地址向下取整（round down）为SHMLBA的整数倍。

SHM\_REMAP：这个标志位的含义是，如果shmaddr地址处于一个已经在用的地址范围内，也不返回错误，强制替换[shmaddr，shmaddr+size]处的内容。如果指定了这个标志位，第二个参数shmaddr不得为NULL，否则失败，置errno为EINVAL。这个标志位是Linux特有的标志位。普通应用，不建议使用。

当shmat函数执行成功时，返回一个地址值，表示将共享内存附接到该地址，从此可以像操作普通地址一样来对待这个值。当shmat函数执行失败时，返回-1，并设置errno。

一般来讲，共享内存段存储的是用户定义的结构体，该结构体作为多个进程之间通信的消息体。使用方式如下：

* 在调用shmget函数的时候，size的值为用户定义的结构体的大小。
* 调用shmat时将返回地址赋给用户定义结构体类型的指针。
* 操作用户定义结构体的指针，来修改或读取共享内存的内容，达到通信的目的。
* 对于写入方：修改指针指向结构体的成员变量，达到修改共享内存的目的
* 对于读取方：获取指针指向结构体的成员变量的值，达到获取共享内存值的目的。

共享内存作为进程间通信的手段，并不能独立完成通信的任务，大部场景下须和信号量或者线程锁配合使用。试想以下场景：多个进程没有同步，一同写入共享内存，可能会使共享内存的值不是期望的结果；退一步，纵然只有一个写入者，一个读取者，也可能会出现写入者正在修改，读取者读取的内容是中间结果，而不是修改成功后的完整结果。因此共享内存多是和信号量或者线程锁配合使用。

12.6.3 分离共享内存

当一个进程不在需要访问共享内存时，可以执行shmdt函数，将共享内存段分离出其虚拟地址空间。

#include <sys/types.h>

#include <sys/shm.h>

int shmdt(const void \*shmaddr);

shmdt函数同shmctl的IPC\_RMID是不同的。一个进程和共享内存分离，仅仅是将共享内存的引用计数减1，并不会删除共享内存。事实上，只有共享内存引用计数为0时，调用shmctl函数的IPC\_RMID命令才会真正的删除共享内存。从这个意义上说，当进程不再需要共享内存了，需要及时的调用shmdt。否则，shmctl函数的IPC\_RMID也无法真正将共享内存释放，造成内存泄露。

对于共享内存来说：

* fork创建子进程，子进程会继承父进程附接的所有共享内存段
* 进程调用exec函数，进程会分离所有附接的共享内存段
* 进程退出时，共享内存段也会自动分离

12.6.4 控制共享内存

shmctl函数用来控制共享内存。

#include <sys/ipc.h>

#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);

当cmd为IPC\_STAT和IPC\_SET时，需要用到第三个参数。其中shmid\_ds结构体定义如下：

struct shmid\_ds {

struct ipc\_perm shm\_perm;

size\_t shm\_segsz;

time\_t shm\_atime;

time\_t shm\_dtime;

time\_t shm\_ctime;

pid\_t shm\_cpid;

pid\_t shm\_lpid;

shmatt\_t shm\_nattch;

...

};

**IPC\_STAT**

获取shmid对应的共享内存的信息。所谓信息，就是上面结构体的内容。

* shm\_perm：除了常规的读写以外，还有两个标志位：
* SHM\_DEST：表示已经执行IPC\_RMID操作，只待附接进程数变成0，就可以真正销毁共享内存了。
* SHM\_LOCKED：该共享内存执行过shmctl的SHM\_LOCK，表示将共享内存锁定在了内存之中，不会被置换出去。
* shm\_segsz：共享内存的字节数。
* shm\_atime：创建共享内存时设置成0，当进程调用shmat附接共享内存时，将时间更新为当前时间。
* shm\_dtime：创建共享内存时设置成0，当进程调用shmdt分离共享内存时，将时间更新成当前时间。
* shm\_ctime：当共享内存创建时，设置成当前时间；当调用IPC\_SET操作，更新这个值为当前时间
* shm\_nattch：附接该共享内存到其虚拟地址空间的进程的个数。

**IPC\_SET**

IPC\_SET也只能修改shm\_perm中的uid，gid及mode。

**IPC\_RMID**

如果共享内存的引用计数shm\_nattch等于0，可以立即删除共享内存。但是如果仍然存在进程附接该共享内存，并不执行真正的删除操作，而是待所有进程都已分离之后，再执行删除操作。在shm\_perm的mode中会有一个只读的标志位SHM\_DEST来表征这种状态。纵然共享内存处于SHM\_DEST状态，依然允许新的进程调用shmat函数来附接。

**SHM\_LOCK**

将共享内存锁定在RAM中，而不会被置换出去。这种做法可以提升访问共享内存的性能。因为进程访问共享内存所在的分页时，不会发生缺页中断而导致性能降低。注意调用SHM\_LOCK并不保证shmctl函数结束时，所有共享内存页位于RAM中，当没有驻留在RAM中的页面因为访问需要，由缺页中断而被一个个引入RAM后，该页面就会被锁定，而不会被交换出去，除非调用了下面提到的SHM\_UNLOCK，否则分页一直驻留在内存中。SHM\_LOCK设置的是共享内存的属性，而不是进程的属性，所以哪怕所有附接共享内存的进程都已退出，被锁定的共享内存页仍然被锁定在RAM中。所以为了防止资源泄露，调用SHM\_LOCK时一定要谨慎。

**SHM\_UNLOCK**

和SHM\_LOCK相反，是解锁，允许共享内存的页面被交换出去。

12.7 POSIX IPC概述

POSIX IPC和System V IPC一样，也是包含三种：

* POSIX消息队列
* POSIX信号量
  + 命名信号量
  + 未命名信号量
* POSIX共享内存

POSIX IPC的出现晚于System V IPC，因此POSIX IPC的设计者充分地借鉴了System V IPC的长处和缺点。从使用的角度来说，POSIX IPC的接口明显更加简单，更加易于使用。坦率地讲，System V IPC的接口是比较难用的，System V信号量集尤为难用。

POSIX IPC的模型，使用open、close和unlink函数，和传统的UNIX文件模型一致。易于理解和操作。和System V IPC使用key获取IPC对象，操作风格虽然统一，但是不同于传统的UNIX文件模型。

最后，POSIX IPC在内核中维护有引用计数，删除比较简单。因为可以调用close简单地断开和POSIX IPC的连接，当所有进程都关闭对象时，POSIX IPC对象就会被销毁。System V IPC中的共享内存有类似的性质，但是System V消息队列和System V信号量集则完全不同，因为没有引用计数，多进程的程序可能很难确定哪个进程是最后一个需要访问对象的进程，从而很难确定何时可以安全地删除System V IPC对象。

下面看下POSIX IPC的接口：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 消息队列 | 信号量 | 共享内存 |
| 头文件 | <mquque.h> | <semaphore.h> | <sys/mman.h> |
| 创建或打开 | mq\_open | sem\_open | shm\_open+mmap |
| 关闭 | mq\_close | sem\_close | munmap |
| 删除 | mq\_unlink | sem\_unlink | shm\_unlink |
| 执行IPC | mq\_send  mq\_receive | sem\_post  sem\_wait  sem\_getvalue | 在共享内存区域内操作数据 |
| 其他操作 | mq\_getattr  mq\_setattr  mq\_notify | sem\_init（初始化未命名信号量）  sem\_destroy（销毁未命名信号量） | 无 |

**创建及打开对象**

要访问POSIX IPC对象，就必须通过某种方式来识别或者获取它。就像System V IPC采用key，由共同的key从而访问到同一个System V IPC对象。对于POSIX IPC而言，通过的是类似于路径名的POSIX IPC名字，mq\_open、sem\_open及shm\_open第一个参数都是POSIX IPC名字。为了可移植性，POSIX IPC的名字须是斜线打头，后面跟一个或者多个非斜线的字符，如下所示：

fd = shm\_open(“/myshm”,O\_CREAT | O\_RDWR, S\_IRUSR | S\_IWUSR);

注意，POSIX IPC名字可以是已经存在的文件名，也可以是不存在的文件名，为了增强可移植性，一般要遵循以下建议

1. POSIX IPC名字以字符斜杠“/”开头，不包含其他斜杠，尽量不要用“/”作为POSIX名字。
2. 将POSIX IPC名字define定义在头文件，万一不可移植，尚可以修改头文件解决。

有了POSIX IPC名字，接下来就是要创建POSIX IPC对象了。创建或者打开，都是有open系列函数完成。接下来的对IPC对象的操作就要操作open函数返回的句柄，和操作文件一样。

对于POSIX IPC的open系列函数而言，一般至少包含三个参数name，oflag，mode。

name已经说过，就是POSIX IPC名字。下面分析第二个参数打开标志位

|  |  |  |  |
| --- | --- | --- | --- |
| 说明 | mq\_open | sem\_open | shm\_open |
| 只读 | O\_RDONLY |  | O\_RDONLY |
| 只写 | O\_WRONLY |  |  |
| 读写 | O\_RDWR |  | O\_RDWR |
| 若不存在则创建 | O\_CREAT | O\_CREAT | O\_CREAT |
| 排他性创建 | O\_EXCL | O\_EXCL | O\_EXCL |
| 非阻塞模式 | O\_NONBLOCK |  |  |
| 若已存在则截断 |  |  | O\_TRUNC |

如果真的创建POSIX IPC对象，需要第三个mode，来指定权限，这个权限和文件的权限类似，不外乎S\_IRUSR、S\_IWUSR、S\_IRGRP、S\_IWGRP、S\_IROTH及S\_IWOTH这六种权限。

打开还是创建，取决于oflag是否设置O\_CREAT以及O\_EXCL标志位，内在的逻辑和System V IPC一致，如下表所示：

|  |  |  |
| --- | --- | --- |
| oflag标志位 | 对象不存在 | 对象存在 |
| 无特殊标志 | 出错，errno为ENOENT | 成功，引用已存在对象 |
| O\_CREAT | 成功，创建新对象 | 成功，引用已存在对象 |
| O\_CREAT | O\_EXCL | 成功，创建新对象 | 失败，errno为EEXIST |

**关闭对象**

对于POSIX消息队列和信号量而言，存在一个IPC close（mq\_close和sem\_close） 函数来关闭IPC对象，表示进程已经使用完毕。POSIX共享内存对象的关闭则需要调用munmap函数来解除映射。

当进程退出时或执行exec系列函数时，自动关闭POSIX IPC对象。

12.8 POSIX消息队列

12.8.1 POSIX消息队列概述

POSIX消息队列和System V消息队列有很多相似之处，比如数据交换的格式都是消息。但是两者之间也有不同：

* POSIX消息队列存在引用计数，所以删除操作要比System V消息队列方便
* POSIX消息队列本质是优先队列，调用receive操作总是返回优先级最高的最早达到的消息。相比之下System V消息队列更加灵活，receive操作可以返回任意指定优先级的消息
* POSIX消息队列提供了notify的机制，允许当队列中存在一条消息时，异步地通知进程，这是System V消息队列所没有的功能。

下面我们分别介绍消息队列的接口：

12.8.2 POSIX消息队列的创建、打开、关闭及删除

之所以在本节介绍三个接口，是因为POSIX消息队列的接口和操作文件的接口非常类似。

mq\_open函数如同操作文件的open函数，用来创建或者打开一个消息队列。

#include <fcntl.h>

#include <sys/stat.h>

#include <mqueue.h>

mqd\_t mq\_open(const char \*name, int oflag);

mqd\_t mq\_open(const char \*name, int oflag, mode\_t mode,

struct mq\_attr \*attr);

oflag标志位包括O\_RDONLY、O\_WRONLY、O\_RDWR、O\_CREAT、O\_EXCL及O\_NONBLOCK。除了O\_NONBLOCK标志位，其他都是老朋友了，不必赘述，单提一下O\_NONBLOCK。后续的mq\_send函数和mq\_receive函数可能会陷入阻塞，在这种情况下，如果设置了O\_NONBLOCK标志位，就立刻返回失败，设置errno为EAGAIN。

第三个参数mode和第四个参数attr只有在创建消息队列的时候才有意义。如果仅仅是打开消息队列，无需这两个参数。mode设置的是访问权限，attr设置的是消息队列的属性。在介绍mq\_getattr函数和mq\_setattr函数会介绍。默认情况下，第四个参数可以传递NULL，表示创建默认属性的消息队列。

当mq\_open调用成功是返回一个mqd\_t类型的消息队列描述符。对于Linux平台而言，这就是一个int型的数字，其实这个数字和open函数返回的文件描述本质是一样的，从内核的ipc/mqueue.c中mq\_open系统调用实现可以看出：

SYSCALL\_DEFINE4(mq\_open, const char \_\_user \*, u\_name, int, oflag, mode\_t, mode,struct mq\_attr \_\_user \*, u\_attr)

{

…

fd = get\_unused\_fd\_flags(O\_CLOEXEC);

…

return fd;

}

在Linux平台下，消息队列描述符，本质是文件描述符，这个事实十分重要。它牵扯到消息队列的很多属性。比如单个进程所允许打开的消息队列的个数限制，限制的本质是，进程所允许打开的文件描述符的个数；在比如由于消息队列描述符本质是文件描述符，所以可以使用select/poll/epoll等多路复用系统调用来监控这个描述符，这是System V消息队列无法做到的。

调用fork之后，子进程也获得了消息队列描述符的副本，这个副本会引用同样的打开的消息队列。

调用exec之后，由于消息队列描述符带有了O\_CLOEXEC标志位，所以其打开的消息队列描述符会被关闭。

当进程退出时，所有打开的消息队列都会被关闭。

mq\_close函数用来关闭消息队列描述符，这个函数和关闭文件的close函数十分类似。

#include <mqueue.h>

int mq\_close(mqd\_t mqdes);

如果进程已经注册了消息通知，那么消息通知也会被删除，因为任一时刻，只能有一个进程向特定消息队列注册，接收消息通知，因此消息通知删除后，其他进程就能注册消息通知了。

进程退出，或者进程执行exec，内核都会负责自动关闭消息队列，将消息队列的引用计数减1。

POSIX消息队列也具有内核持久性，哪怕所有进程相关进程都执行了mq\_close，消息队列的引用计数为0，只要不显式调用mq\_unlink，该队列及队列上的消息依然存在。因此要销毁消息队列，需要调用mq\_unlink函数。

#include <mqueue.h>

int mq\_unlink(const char \*name);

需要提及的是上面提到的三个函数，链接的时候需要和实时库librt链接起来，即指定

-lrt选项来完成。

下面我们通过两个测试程序来学习消息队列的创建：

第一个小程序是用来创建消息队列，如果传入了-e选项，表示创建时加上O\_EXCL标志位。

#include <mqueue.h>

#include <stdio.h>

#include <sys/stat.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

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

{

int c,flags;

mqd\_t mqd;

flags = O\_RDWR|O\_CREAT;

while((c=getopt(argc,argv,"e")!=-1))

{

switch(c)

{

case 'e':

flags |= O\_EXCL;

break;

}

}

if(optind!=argc-1)

{

fprintf(stderr,"usage:mqcreate [-e] <name>\n");

return -1;

}

mqd = mq\_open(argv[optind],flags,S\_IRUSR|S\_IWUSR,NULL);

if(mqd == -1)

{

fprintf(stderr,"mq\_open failed (%s)\n",

strerror(errno));

return -2;

}

mq\_close(mqd);

return 0;

}

第二个小程序是用来删除POSIX消息队列的：

#include <mqueue.h>

#include <stdio.h>

#include <sys/stat.h>

#include <sys/msg.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>

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

{

if(argc != 2)

{

fprintf(stderr,"usage mqunlink <name>\n");

return -1;

}

int ret = mq\_unlink(argv[1]);

if(ret != 0)

{

fprintf(stderr,"mq\_unlink failed (%s)\n",

strerror(errno));

return -2;

}

return 0;

}

Linux下POSIX提供了mqueue类型的虚拟文件系统，可以通过挂载，很方便的使用ls和rm来列出或者删除POSIX消息队列。

可以通过如下命令挂载消息队列到文件系统：

mount -t mqueue source target

其中source可以为none，target是挂载点。比如可以通过如下命令挂载消息队列：

➜ ~ mkdir /dev/mqueue

➜ ~ mount -t mqueue none /dev/queue

➜ ~ mount |grep mqueue

none on /dev/mqueue type mqueue (rw)

使用第一个程序编译出mqcreate二进制程序，使用第二个程序编译出mqunlink二进制程序，可以做如下实验：

➜ ./mqcreate /abcd

➜ ll /dev/mqueue

总用量 0

-rw------- 1 manu manu 80 3月 9 22:26 abcd

➜ cat /dev/mqueue/abcd

QSIZE:0 NOTIFY:0 SIGNO:0 NOTIFY\_PID:0

可以看出通过mqcreate创建出来的消息队列，可以通过ls /dev/mqueue查看，甚至可以通过cat /dev/mqueue/queue\_name来获取到消息队列的信息。至于内容的含义，后面会逐步提到。

12.8.3 POSIX消息队列的属性

前文介绍mq\_open函数时提到，第四个参数是mq\_attr类型的，表示消息队列的属性。不但mq\_open函数会用到，POSIX消息队列提供了mq\_getattr函数和mq\_setattr函数，这两个函数分别是用来获取消息队列的属性信息和修改消息队列的属性。那么我们先看下消息队列有哪些属性：

struct mq\_attr {

long mq\_flags;

long mq\_maxmsg;

long mq\_msgsize;

long mq\_curmsgs;

}

这个结构体定义在<mqueue.h>文件中：

* mq\_flags：0或者设置了O\_NONBLOCK
* mq\_maxmsg：通过mq\_send函数添加消息的上限。对于普通进程，默认值是10。

➜ ~ cat /proc/sys/fs/mqueue/msg\_max

10

这个值是可以调整的，最大值是HARD\_MSGMAX，对于32位系统，该值是32768，对于64位系统，该值是65535。这个上限值是内核版本相关的，在较老的版本里面，计算公式是131072/sizeof(void\*)，但是较新的内核版本，为下面计算公式）

#define HARD\_MSGMAX (32768\*sizeof(void \*)/4)

* mq\_msgsize：定义了加入消息队列的单条消息的最大size。这个值默认情况下是8192字节。特权用户可以修改，但是内核硬上限为1048576字节。

➜ ~ cat /proc/sys/fs/mqueue/msgsize\_default

8192

➜ ~ cat /proc/sys/fs/mqueue/msgsize\_max

8192

* mq\_curmsg：消息队列当前消息数。

对于mq\_maxmsg和mq\_msgsize这两个属性，在消息队列创建的时候，就已经确定下来了，虽然提供有mq\_setattr函数，但是该函数并不能修改这两个属性。因此如果应用确实需要超过8K字节的消息体，或者需要消息队列需要接收很多消息，需要在调用mq\_open函数时，设置好mq\_msgsize和mq\_maxmsg属性。

消息队列创建以后，我们可以调用mq\_getattr来获取这些属性信息，也可以调用mq\_setattr来修改属性。

#include <mqueue.h>

int mq\_getattr(mqd\_t mqdes, struct mq\_attr \*attr);

int mq\_setattr(mqd\_t mqdes, struct mq\_attr \*newattr,

struct mq\_attr \*oldattr);

可以改变的属性只有第一个mq\_flags，即可以改变O\_NONBLOCK标志位是否置位。其它的属性均不可以修改。改变O\_NONBLOCK属性的方法如下：

mq\_getattr(mqd,&attr);

attr.mq\_flags |= O\_NONBLOCK; /\*设置O\_NONBLOCK属性\*/

//attr.mq\_flags &=(~O\_NONBLOCK);/\*取消O\_NONBLOCK属性\*/

mq\_setattr(mqd,&attr,NULL)

12.8.4 消息的发送和接收

**发送消息**

POSIX消息队列发送消息和接收消息的接口都十分容易理解，从易用性的角度来讲，要优于System V消息队列，尽管System V消息队列的接口更加灵活。

#include <mqueue.h>

int mq\_send(mqd\_t mqdes, const char \*msg\_ptr,

size\_t msg\_len, unsigned msg\_prio);

接口如何使用一目了然：

第三个参数msg\_len表示消息体的长度，长度为0也是合法的，最大不得超过mq\_msgsize。如果消息体太大，会返回失败，并且置errno为EMSGSIZE。

第四个参数为消息的优先级，为一个非负的整数。问题就来了，最大容许的优先级为多少。在Linux上，这个上限为32768。

#define MQ\_PRIO\_MAX 32768

如果消息队列已满，mq\_send函数可能会阻塞。如果设置了O\_NONBLOCK标志位，这种情况下mq\_send函数返回失败，errno被置为EAGAIN。

**接收消息**

接收消息的接口如下

ssize\_t mq\_receive(mqd\_t mqdes, char \*msg\_ptr,

size\_t msg\_len, unsigned \*msg\_prio);

对于POSIX消息队列而言，总是取走优先级最高的，最先到达的消息。

第二个参数msg\_ptr是存放从消息队列取来的消息体的内存缓冲区的地址，第三个参数msg\_len是该内存缓冲区的大小。因为消息体长度的不确定，所以该缓冲区的大小不得小于最大消息体的长度（mq\_msgsize），否则消息体长度超过缓冲区的大小，就会失败，返回EMSGSIZE的错误。如果获得消息队列的最大消息长度？mq\_getattr函数!

如果第四个参数msg\_prio不是NULL，那么就会将取到的消息体的优先级复制到msg\_prio指向的整型变量。第四个参数可以为NULL，表示压根不在乎消息体的优先级。

如果调用mq\_receive函数时，消息队列中并没有消息，则函数陷入阻塞。如果设置了O\_NONBLOCK标志位，立即失败返回，设置errno为EAGIAN。

POSIX消息队列本质就是个优先级队列。优先级高的消息总是被先取出。从这个角度上看，System V消息队列更灵活，它可以让各个进程选取自己感兴趣的消息。因此从灵活性的角度讲，System V消息队列胜出一局。

12.8.5 消息的通知

对于System V消息队列，消息队列里面存在消息，无法通知其他进程来取。对于从消息队列取消息的进程而言，要么调用msgrev函数，阻塞于此，直到消息队列里面有消息，要么调用msgrev函数时设置IPC\_NOWAIT标志位，尽管不阻塞了，但是不得不轮询。如果空的消息队列一收到消息，给相应进程发出通知，则该进程就可以及时的处理消息，这种机制称为异步通知机制。POSIX消息队列就引入了这种机制。

POSIX消息队列提供了两种异步通知的方法可供选择

* 产生一个信号
* 创建一个线程来执行一个事先指定的函数

这种通知机制的注册，是通过mq\_notify函数实现的：

#include <mqueue.h>

int mq\_notify(mqd\_t mqdes, const struct sigevent \*sevp);

这个函数的关键在第二个入参上，结构体包含如下参数，但是并不保证顺序严格按照下面的定义，记不清楚成员变量，可以通过man sigevent来查看手册。

union sigval{

int sigval\_int;

void \*sigval\_ptr;

}

struct sigevent {

int sigev\_notify;

int sigev\_signo;

union sigval sigev\_value;

void (\*sigev\_notify\_function)(union sigval);

void \*sigev\_notify\_attributes;

}

结构体sigevent的第一个成员sigev\_notify用来表示采用哪种通知方式：

* SIGEV\_NONE：当消息到达空的消息队列时，不采取任何通知行动
* SIGEV\_SIGNAL：当消息到达空的消息队列时，采用发送信号的方式通知进程。如果进程用带SA\_SIGINFO标志位的sigaction来注册信号处理函数，那么可以获取到如下表信息。

|  |  |
| --- | --- |
| **sig\_info\_t的成员** | **值** |
| si\_signo | 信号的编号，即mq\_notify注册时sigevent结构体中的sigev\_signo |
| si\_code | SI\_MESGQ |
| si\_pid | 发送消息的进程的PID |
| si\_uid | 发送消息的进程的真实用户UID |

* SIGEV\_THREAD：当消息到达空的消息队列时，进程会创建一个线程，执行sigevent结构体中定义的sigev\_notify\_function。sigev\_notify\_attributes可以为NULL，也可以为指向一个pthread\_attr\_t结构的指针，这个参数来决定线程的属性。线程要执行的函数sigev\_notify\_function也是有入参的，入参即sigevent结构图中的union sigval sigev\_value。

从现在看来，我们有两种办法实现消息的异步处理。一种是信号处理，一种是线程处理消息。须知，只能有一个函数调用mq\_notify函数，第二个调用者会失败。

下面我们分别讲述两种机制：

**信号通知**

最容易想到的方法就是，在信号处理函数中，调用mq\_receive函数，并且进一步处理消息。这种方法很不幸是不安全的。信号一章提到，信号的本质是异步，很多函数都不是异步信号安全的。我们日常使用的绝大多数函数都不是异步信号安全的。因此，信号处理函数中执行比较丰富的操作，是一种不负责任的做法，势必会触雷。一种常规的做法是，信号处理函数，只会设置一个标志变量，由主函数流程负责检查该变量的值，从而达到安全的目的。

一种安全的做法是sigwait函数。在信号一章中提到，这个函数的引入，解决了信号的异步带来的很多问题。可以说这个函数提供了一种同步的方式来等待信号的降临。

#include <signal.h>

int sigwait(const sigset\_t \*set, int \*sig);

将要等待的信号放置到set中，sigwait函数调用就会阻塞，直到set集合中某个信号处于未决状态，sigwait函数返回，信号的值记录在sig指针指向的整型变量中。需要注意的一点是，调用sigwait函数之前，set中的所有信号都要被阻塞，否则结果是不可预知的。

以SIGUSR1为例，我们调用mq\_notify函数，使消息降临空队列时，发送信号SIGUSR1，主流程等待SIGUSR1，收到信号时，去消息队列取出消息，这个流程如下：

mqd\_t mqd;

struct mq\_attr attr ;

sigset\_t newmask ;

struct sigevent sigev;

mqd = mq\_open(mq\_filename,O\_RDONLY|O\_NONBLOCK);

mq\_getattr(mqd,&attr);

buffer = malloc(attr.mq\_msgsize)；/\*确保buffer足够大\*/

sigemptyset(&newmask);;

sigaddset(&newmask,SIGUSR1);

sigprocmask(SIG\_BLOCK,&newmask,NULL);/\*阻塞等待的信号\*/

sigev.sigev\_notify = SIGEV\_SIGNAL;

sigev.sigev\_signo = SIGUSR1;

mq\_notify(mqd,&sigev);

for( ; ; )

{

sigwait(&newmask,&signo);/\*等待SIGUSR1信号\*/

if(signo == SIGUSR1)

{

mq\_notify(mqd,&sigev); /\*先重新注册notify函数\*/

while( n = mq\_receive(mqd,buffer,attr.mq\_msgsize,NULL) >= 0)

{

/\*process the message in buffer\*/

}

if(errno != EAGAIN)

{

/\*some error happened\*/

}

}

}

需要一提的是，mq\_notify函数发出信号，完成使命之后，要想继续使用这种通知机制，需要重新调用mq\_notify函数注册。

**通过线程处理消息**

POSIX消息队列提供的另外一种方法就是创建线程，执行预先约定的函数。在使用中，需要将sigev.sigev\_notify设置成SIGEV\_THREAD，同时设置好线程应该执行的函数，即将sigev.sigev\_notify\_function设置成约定好的函数。如果线程函数需要入参，你可以将任何变量的地址填入sigev.sigev\_value.sival\_ptr中，到达传递参数的目的。

整体代码流程如下（示意代码，不完整）：

static void notify\_function(union sigval sv)

{

struct mqd\_t \*mqdp = sv.sival\_ptr；

mq\_getattr(\*mqdp,&attr);

buffer = malloc(attr.mq\_msgsize)；/\*确保buffer足够大\*/

notify\_setup(mqdp);

while(n = mq\_recevie(\*mqdp,buffer,attr.mq\_msgsize,NULL)>=0)

{

/\*处理buffer中的消息体\*/

}

if(errno != EAGAIN)

{

/\*发生错误\*/

}

free(buffer);

}

static void notify\_setup(mqd\_t\* mqdp)

{

struct sigevent sig\_ev ;

sigev.sigev\_notify = SIGEV\_THREAD;

sigev.sigev\_notify\_function = notify\_function;

sigev.sigev\_notify\_attributes = NULL;

sigev.sigev.value.sival\_ptr = mqdp;

mq\_notify(\*mqdp,\*sigev);

}

int main()

{

mqd = mq\_open(mqfilename,O\_RDONLY | O\_NONBLOCK);

notify\_setup(&mqd);

}

和信号通知机制一样，一旦创建线程执行，通知机制就结束了，需要重新调用mq\_notify函数来注册。

12.9 POSIX信号量

12.9.1 POSIX信号量概述

POSIX信号量和System V信号量类似，是一种进程间或者一个进程内部多个线程之间同步的手段。

对于信号量的典型使用如下：

1. 创建信号量，同时要指定信号量的初始值。对于最常见的二值信号量而言，初始值通常是1，当然也可能是0。
2. 等待（wait）信号量。当进程要执行临界区代码时，通常要首先等待信号量。如果信号量的值小于或者等于0，那么就等待。一旦其值变成大于0，就将信号量的值减1。
3. 发布（post）一个信号量。等进程已经执行完临界区代码，通常要发布信号量，将信号量的值加1，表示释放共享资源。如果此时信号量的值为0，并且有进程或线程阻塞在信号量上，那么，post操作会唤醒某个阻塞的进程或线程。但是如果存在多个进程或线程阻塞在信号量上，到底哪个进程或线程被唤醒是不确定的。

POSIX提供了两类信号量：有名信号量和无名信号量。这两种信号量本质都是一样的，从下图可以看出，最重要的sem\_wait接口和sem\_post接口，都是一样的。如此说来，两种信号量有何不同，各自应用在哪些场景？

无名信号量，又称为基于内存的信号量，由于其没有名字，没法通过open操作，直接找到对应的信号量，所以很难直接用于没有关联的两个进程之间。无名信号量多用于多线程程序线程之间的互斥和同步。

有名信号量由于有名字，多个不相干的进程可以通过名字打开同一个信号量，从而完成同步操作，所以有名信号量操作要方便一些，适用范围也要比无名信号量广。



和System V信号量相比，信号量的创建和赋初始值是一个接口完成，不存在System V信号量存在的初始化竞争的问题。另外，POSIX信号量从使用的角度来说，接口更加友好，更易于使用。System V信号量表面看很强大，但是有过度设计之嫌，接口异常难用。要想正常使用System V信号，不得不封装成更好用的接口。

下面我们介绍POSIX信号量。

12.9.2 创建、打开、关闭和删除有名信号量

创建或者打开有名信号量，需要调用sem\_open函数

#include <fcntl.h>

#include <sys/stat.h>

#include <semaphore.h>

sem\_t \*sem\_open(const char \*name, int oflag);

sem\_t \*sem\_open(const char \*name, int oflag,

mode\_t mode, unsigned int value);

oflag标志包括O\_CREAT和O\_EXCL标志位，带了O\_CREAT标志位，表示要创建信号量。

mode表示创建的新信号量的访问权限，标志位和open一样，mode参数的值也会根据进程的umask来取掩码。也不多提。

value是新建信号量的初始值。创建和赋初值有一个接口完成，就不会出现System V信号量可能出现的初始化竞争的问题了。value的值在最小值0，和最大值SEM\_VALUE\_MAX之间。SUSv3要求最大值至少等于32767，对于Linux而言，这个限制为INT\_MAX（在Linux/x86平台上，该值是2147483647）。

当sem\_open函数失败时，返回SEM\_FAILED，并且设置errno。

注意，不要尝试创建sem\_t结构体的副本，如下所示：

sem\_t \*sem\_p,sem\_dup;

sem\_p = sem\_open(…);

sem\_dup = \*sem\_p;/\*非法操作\*/

sem\_wait(&sem\_dup);

上面定义了sem\_p的副本sem\_dup，在副本上执行sem相关的操作，行为是不可预知的，不要这样使用。切记，后面所有的调用都要用通过sem\_open返回的sem\_t类型的指针操作，而不能使用结构体的副本。

当一个进程打开有名信号量时，系统会记录进程与信号的关联关系。调用sem\_close时，会终止这种关联关系，同时信号量的进程数的引用计数减1.

#include <semaphore.h>

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

进程终止时，进程打开的命名信号量会自动关闭。当进程执行exec系列函数时，进程打开的有名信号量会自动关闭。

但是关闭不等同于删除，如果要删除信号量需要调用sem\_unlink函数。

#include <semaphore.h>

int sem\_unlink(const char \*name);

将有名信号量的名字作为参数，传递给sem\_unlink，该函数会负责将有名信号量删除。由于系统为信号量维护了引用计数，所以只有当所有打开信号量的进程都关闭了之后，才会真正的删除。

12.9.2 有名信号量的使用

对于信号量的使用，就是在进入临界区之前，首先等待信号量（sem\_wait），退出临界区之后，发布信号量（sem\_post）。

**等待信号量**

#include <semaphore.h>

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

sem\_wait函数用来等待信号量，它会将信号量的值减1。如果调用sem\_wait函数时，信号量的当前值大于0，那么sem\_wait函数立刻返回。否则sem\_wait函数陷入阻塞，待信号的值大于0之后，再执行减1操作，成功返回。

如果陷入阻塞的sem\_wait函数被信号中断，那么会返回失败，并且置errno为EINTR。无论使用sigaction注册信号处理函数时，是否使用了SA\_RESTART标志位，都不会自动重启系统调用。

如果仅仅是尝试等待信号量，不想陷入阻塞，可以调用sem\_trywait函数。

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

sem\_trywait会尝试等待，如果信号量的值大于0，那么该函数将信号量的值减1之后，立刻返回。如果信号量的当前值为0，那么sem\_trywait不会陷入阻塞，而是立刻返回失败，并置errno为EAGAIN。

一种是可能无限期阻塞（sem\_wait），一种是绝不阻塞（sem\_trywait），除了这两种以外，系统还提供了第三种接口，有限期等待：sem\_timedwait。

int sem\_timedwait(sem\_t \*sem,

const struct timespec \*abs\_timeout);

第二个参数为一个绝对时间。可以使用gettimeofday函数获取到struct timeval 类型的当前时间，然后将timeval转换成timespec类型的结构体。然后在该值上加上想等待的时间。或者调用clock\_gettime函数，直接获得timespec结构体类型的变量表示当前时刻，然后在结构体上加上想等待的时间，作为第二个参数传给sem\_timedwait函数。

如果在指定的时间前，信号量的值仍然为0，那么返回失败，置errno为ETIMEOUT。

**发布信号量**

sem\_post函数用来发布信号量，表示资源已经使用完毕，或者临界区代码执行完毕，可以归还资源。该函数会使信号量的值加1。

#include <semaphore.h>

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

如果发布信号量之前，信号量的值是0，并且已经有进程或者线程在等待在信号量上，此时会有一个进程被唤醒，被唤醒的进程会继续sem\_wait函数的减1操作。如果多个进程等待在信号量上，无法确认那个进程会被唤醒。

**获取信号量的值**

sem\_getvalue函数会返回当前信号量的值，将值写入sval指向的变量。

#include <semaphore.h>

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

如果信号量的值大于0，含义自不必说，但是如果信号量的值等于0，同时又很多进程或者线程阻塞在信号上，那么应该返回0还是返回一个负值，其绝对值等于等待进程的个数？看起来后者更有意义，因为从该值可以获知到竞争的激烈程度。但是Linux返回0。

当sem\_getvalue返回时，其返回的值可能已经过时了。从这个意义上将，该接口的意义并不大。

12.9.3 无名信号量创建和销毁

无名信号量，由于其没有名字，所以适用范围要小于有名信号量。只有将无名信号量放在多个进程或线程都共同可见的内存区域内才有意义，否则协作的进程无法操作信号量，达不到同步或者互斥的目的。所以一般而言，无名信号量多用于线程之间。因为线程共享地址空间，访问共同的无名信号量时很容易办到的事情。或者将信号量创建在共享内存内，多个进程通过操作共享内存的信号量达到同步或互斥的目的。



**初始化无名信号**

无名信号量的初始化是通过sem\_init函数来完成的。

#include <semaphore.h>

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

第二个pshared参数用来声明信号量时在线程间共享还是在进程间共享。0表示在线程间共享，非零值表示信号量将在进程间共享。要想在进程间共享，信号量必须位于共享内存区域内。

无名信号量的生命周期是有限的，对于线程间共享的信号量，线程组退出了，无名信号量也就不复存在了。对于进程间共享的信号量，信号量的持久性与所在的共享内存的持久性一样。

信号量初始化以后，就可以像操作有名信号量一样操作了。

**销毁无名信号量**

#include <semaphore.h>

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

sem\_destroy用来销毁sem\_init函数初始化的信号量。只有所有进程都不会再等待一个信号量时，才能安全销毁。对Linux实现而言，省略sem\_destroy函数，也不会带来异常。但是为了安全和可移植，还是要合适时机，正常销毁信号量。

12.10 POSIX共享内存

POSIX共享内存可以在无关进程之间共享一个内存区域的方法。和System V信号相比，POSIX使用了文件系统来标识共享内存，并且调用操作文件的接口来操作共享内存。每创建一个POSIX共享内存，挂载在/dev/shm下的tmpfs文件系统中会新增一个文件。

和System V共享内存相比，POSIX共享内存的大小可以动态的调整，因为POSIX共享内存是基于文件的，所以可以很方面的通过ftruncate函数调整共享内存的大小。对于共享内存的使用者，可以通过munmap和mmap重建映射。System V共享内存的大小在创建时已经确定，无法调整。

总体来讲，POSIX共享内存要优于System V共享内存。如果两项比较，建议使用POSIX共享内存。

12.10.1 共享内存的创建、使用和删除

共享内存的创建本质是两个接口，首先是调用shm\_open返回文件描述符，然后通过mmap，将共享内存映射到进程的地址空间。两个函数的搭配很像System V的shmget函数和shmat函数。

#include <sys/mman.h>

#include <sys/stat.h>

#include <fcntl.h>

int shm\_open(const char \*name, int oflag, mode\_t mode);

oflag标志要包含O\_RDONLY或者O\_RDWR标志位，除此外，可以选择的标志位有O\_CREAT表示创建，O\_EXCL配合O\_CREAT表示排他创建。另外一个标志位时O\_TRUNC表示将共享内存的size截断成0。

第三个参数mode和sem\_open及mq\_open不太一样，shm\_open总是要有第三个参数。mode参数配合O\_CREAT标志位使用，用来设定共享内存的访问权限。如果仅仅是打开共享内存，可以传递0。

shm\_open函数调用成功时，会返回一个文件描述符。内核会自动设置FD\_CLOEXEC标志位，表示如果进程执行了exec函数，该文件描述符自动关闭。

因为共享内存时间，所以可以调用文件相关的函数，如fstat函数，fchmod函数，fchwon函数。其中最重要常用的函数要当属ftruncate函数。因为新创建的共享内存，默认大小总是0。所以在mmap之前，需要调用ftruncate函数，调整文件的大小。

#include <unistd.h>

int ftruncate(int fd, off\_t length);

调整了size之后，就可以调用mmap函数将共享内存映射到进程的地址空间了。

#include <sys/mman.h>

void \*mmap(void \*addr, size\_t length, int prot, int flags,

int fd, off\_t offset);

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

第一个参数指定映射的地址，一般来讲这个值传递为NULL，由内核负责选择一个合适的位置。

第二个参数指定映射的字节数。对于共享内存而言，就是共享内存的大小。

第三个参数表示对映射的保护信息，可能的取值如下：

* PROT\_NONE：表示区域无法访问
* PROT\_READ：区域内容
* PROT\_WRITE：区域内容可写
* PROT\_EXEC：区域内容可以执行

对于共享内存而言，一般为可读可写，或者可读。

第四个参数flags对于共享内存而言总是选择MAP\_SHARED，至于其他标志位，略过不提。感兴趣的可以man mmap查看。

第五个参数就是shm\_open打开的文件描述符。

第六个参数，对共享内存的使用，一般总是传递0。

函数的成功是，返回一个指针，透过这个指针，就可以正常操作共享内存了。如果失败，mmap函数返回MAP\_FAILED。

一旦mmap成功，shm\_open的fd就可以关闭了，直接调用close函数即可。后续操作可以操作在mmap返回的地址上。比如向共享内存写入，或者读取共享内存的内容。

不再需要共享内存，需要删除共享内存，接口为：

int shm\_unlink(const char \*name);

POSIX共享内存具有内核持久性，即系统重启内容会丢失，除此外，显示删除共享内存，也会使内容消失。不过删除操作不会立刻影响既有的映射。内容会一致保持，直到进程调用了munmap函数，解除映射。