# 进程通信

## 概述

数据传输：一个进程要向另一个进程传输数据，数据量在一到几兆字节之间。

共享数据：多个进程想要操作共享数据，一个进程对共享数据的修改，其他进程应实时看到。

通知事件：一个进程需要向另一个（组）进程发送消息，通知它（它们）发生了某种事件（如进程终止时需要通知父进程）。

资源共享：多个进程之间共享同样的资源。为了做到这一点，需要内核提供锁和同步机制。

进程控制：有些进程希望完全控制另一个进程的执行（如Debug进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。

UNIX系统上通信和同步工具可以分为三类：

通信：这些工具关注进程之间的数据交换，如管道、FIFO、流socket、消息、共享内存

同步：这些工具关注进程和线程操作之间的同步，比如信号量、文件锁、互斥锁、条件变量

信号：

### IPC发展

Linux进程间通信(IPC)由以下几部分发展而来：

早期UNIX进程间通信、基于System V进程间通信、基于Socket进程间通信和POSIX进程间通信。

UNIX进程间通信方式包括：管道、FIFO、信号。

System V进程间通信方式包括：System V消息队列、System V信号灯、System V共享内存。

POSIX进程间通信包括：posix消息队列、posix信号灯、posix共享内存。

### SystemV IPC

System V IPC概述：

Unix系统存在信号，管道和命名管道等基本进程间通讯机制。

System V引入了三种高级进程间通信机制。

IPC存在于内核中而不是文件系统中。

IPC对象通过其标识符来引用和访问。

为方便进程间识别同一个IPC对象，引入了Key的概念。

System V IPC缺点：

编程接口过于复杂

IPC结构不为文件系统所知

IPC是一种受限资源，但是IPC结构没有保留其引用计数

IPC无回收被丢弃IPC结构的自动机制

系统重启或者使用ipcsrm命令删除

System V IPC对象：

System V IPC对象的访问

每个IPC结构都有get函数创建

服务器/客户机协商Key的方法

struct ipc\_perm{

uid\_t uid; /\*owner’s effective user id\*/

gid\_t gid; /\*owner’s effective group id\*/

uid\_t cuid; /\*creator’s effective user id\*/

gid\_t cgid; /\*creator’s effective group id\*/

mode\_t mode; /\*access mode\*/

......

}

### 进程通信

同一主机上进程的通信方式：

1. Unix进程间通信方式：管道PIPE、有名管道FIFO和信号signal
2. SystemV进程间通信方式：信号量（Semaphore）、消息队列（Message Queue）和共享内存（shared Memory）

### 网络通信

网络主机间的通信方式：

1. RPC：Remote Protocol Call远程过程调用（thrift、grpc、qpid）
2. Socket：基于TCP/IP协议

## 管道

### 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(单工)的（有的系统可能支持全双工），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）。

管道传输的是无格式字节流，并且管道缓冲区的大小是有限的（管道缓冲区存在于内存中，在管道创建时，为缓冲区分配一个页面大小）。

管道对于其两端的进程而言，就是一个文件，但它不是普通的文件，管道文件。

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。

### 管道通信

普通的Linux shell 都允许重定向，而重定向使用的就是管道。例如：ps | grep vsftpd

最常见的IPC机制

通过pipe系统调用

### 管道创建/关闭

管道创建/关闭

#include <unistd.h>

int pipe(int fd[2]);

返回：成功返回0，出错返回-1.

error:

EMFILE(没有空闲的文件描述符)

EMFILE(系统文件表已满)

EFAULT(fd 数组无效)

协同进程

协同进程中的作用就是协同另外一个进程一起工作，体现了对进程职责进行明确细化的编程思想。比如说进程A向同进程B输入数据，进程B对数据进行处理后又输出给A输出，进程A负责数据的输入输出，进程B则专门负责数据的处理，这样两个进程相互合作就构成了协同的关系。

### 代码

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#define MAX\_LEN 128

int main(void)

{

/\*0为读，1为写\*/

int fd[2] = {0}; //描述符

pid\_t pid = 0;

char line[MAX\_LEN] = {0};

int n = 0;

/\*创建管道，需要传入两个文件描述符\*/

if(pipe(fd) < 0)

{

perror("create pipe failed\n");

return -1;

}

/\*fork子进程\*/

if((pid = fork()) < 0)

{

perror("fork failed\n");

return -1;

}

/\*父进程\*/

else if(pid > 0)

{

/\*关闭管道的写描述符\*/

close(fd[1]);

/\*从管道读取数据\*/

n = read(fd[0],line,MAX\_LEN);

printf("read %d bytes from pipe :%s\n",n,line);

}

/\*子进程\*/

else

{

/\*关闭管道的读描述符\*/

close(fd[0]);

/\*向管道写入数据\*/

write(fd[1],"www.yanbinghu.com",sizeof("www.yanbinghu.com"));

}

return 0;

}

在程序中，我们创建了一个管道，父进程关闭了写通道，子进程关闭读通道；子进程向管道内写入字符串，而父进程从管道中读取字符串并输出。

运行结果：

read 18 bytes from pipe :www.yanbinghu.com

## 有名管道

### 概念

命名管道(FIFO)

在文件系统中以一个特殊的设备文件存在。

只要对FIFO有适当访问权限，FIFO可用在任何两个进程间的通信。

对FIFO的操作与操作普通文件一样。

由于管道是一种无形、无名的文件，它就只能通过fork的过程创建在“近亲”的进程之间，而不可能成为可以在任意两个进程之间建立通信的机制，更不可能称为一般的、通用的进程间通信模型。

同时，管道机制的这种缺点本身就强烈暗示人们，只有用有名、有形的文件来实现管道，就可以克服这种缺点。这里所谓的有名是指这样一个文件应该有文件名，使得任何进程都可以通过文件名或者路径与这个文件挂上钩。在这里，“有形”是指文件的inode应该存在于磁盘或者其他文件系统上，使得任何进程在任意时间（不仅仅是在fork时）都可以建立或断开与这个文件的联系，所以命名管道的出现是必然的。

为了实现“命名管道”，在普通文件，块设备文件，字符设备文件之外又建立了一个文件类型，称为FIFO文件。由于这种文件的访问，严格遵守先进先出的原则，而不允许在文件内移动读写指针位置的lseek操作。

有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

FIFO也常常有以下两个用途：

无需创建中间临时文件，复制输出流

多客户-服务进程应用中，通过FIFO作为汇聚点，传输客户进程和服务进程之间的数据

### 命名管道创建

#include <sys/types.h>

#include <sys/stat.h>

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

返回：若成功返回0，出错返回-1。

FIFO相关出错信息：

EACCES 无存取权限

EEXIST 指定文件不存在

ENAMETOOLONG 路径名太长

ENOENT 包含的目录不存在

ENOSPC 文件系统剩余空间不足

ENOTDIR 文件路径无效

EROFS 指定的文件存在于只读文件系统中

其他操作：

mkfifo

open

fopen

fork

### 代码

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <errno.h>

#include <fcntl.h>

#define FIFO "/tmp/fifo"

#define MAX\_LEN 128

int main(void)

{

int writeFd;

char line[MAX\_LEN] = {0};

if(mkfifo(FIFO,S\_IRUSR|S\_IWUSR) < 0 && (errno != EEXIST))

{

perror("make fifo failed:");

return -1;

}

/\*关闭管道的读描述符\*/

writeFd = open(FIFO,O\_WRONLY,0);

/\*向管道写入数据\*/

write(writeFd,"www.yanbinghu.com",sizeof("www.yanbinghu.com"));

close(writeFd);

return 0;

}

它首先创建了一个FIFO，并且打开后，往里面写入字符串，然后关闭退出。

读进程代码如下：

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <errno.h>

#include<fcntl.h>

#define FIFO "/tmp/fifo"

#define MAX\_LEN 128

int main(void)

{

int readFd,n;

char line[MAX\_LEN] = {0};

/\*打开FIFO，这里打开可能失败，应该要对返回值处理\*/

readFd = open(FIFO,O\_RDONLY,0);

/\*从FIFO读取数据\*/

n = read(readFd,line,MAX\_LEN);

printf("read %d bytes from pipe :%s\n",n,line);

close(readFd);

/\*删除FIFO\*/

unlink(FIFO);

return 0;

}

它先打开一个已知的FIFO，然后从FIFO中读取数据。

在一个终端先运行写进程，然后运行读进程，结果如下：

read 18 bytes from pipe :www.yanbinghu.com

我们可以看到，两个没有亲缘关系的进程可以通过FIFO进行通信。

## 信号

### 概述

信号是与一定的进程相联系的，也就是说，一个进程可以决定在进程中对哪些信息进行怎么样的处理。

信号（signal）是Linux进程间通信的一种机制，全程为软中断信号，也被称为软中断。信号本质上是在软件层次上对硬件中断机制的一种模拟。

信号表示一种事件，也可能异步发生，也就是并不与程序执行过程的任何事件同步。如果程序并未安排怎么处理一个特定的信号，那么该信号出现时程序就作出一个缺省的反应。标准未定义这个缺省反应是什么，但绝大多数编译器选择终止程序。另外，程序可以调用signal函数，或者忽略这个信号，或者设置一个信号处理函数，当信号发生时程序就调用这个函数。

与其他进程间通信方式相比，信号所能传递的信息比较粗糙，只是一个整数。但正是由于传递的信息量少，信号也便于管理和使用，可以用于系统管理相关的任务，例如通知进程终结、中止或恢复等。

信号可能在任何时刻发生，所以由信号处理函数修改的变量的值可能会在任何时候发生改变，因此，不能指望这些变量在两条相邻的程序语句中肯定具有相同的值。Volatile关键字就是告诉编译器这个事实，防止它以一种可能修改程序含义的方式“优化”程序。

### 原理

信号由内核管理，产生方式多种多样：

1. 可以由内核自身产生，比如出现硬件错误、内存读取错误、分母为0除法等，内核需要通知相应进程；
2. 也可以由其他进程产生并发送给内核，再由内核传递给目标进程。

信号传递的过程：

1. 内核中针对每一个进程都有一个表来保存信号；
2. 当内核需要将信号传递给某个进程时，就在该进程对应的表中写入信号，这样就生成了信号；
3. 当该进程由用户态陷入内核态，再次切换到用户态之前，会查看表中的信号。如果有信号，进程就会首先执行信号对应的操作，此时叫做执行信号；
4. 从生成信号到将信号传递给对应进程这段时间，信号处于等待状态；
5. 我们可以编写代码，让进程阻塞（block）某些信号，也就是让这些信号始终处于等待的状态，直到进程取消阻塞（unblock）或者忽略信号。

### 信号名

SIGABRT 程序请求异常终止

SIGFPE 发生一个算术错误

SIGILL 检测到非法指令

SIGSEGV 检测到对内存的非法访问

SIGINT 收到一个交互性注意信号

SIGTERM 收到一个终止程序的请求

注：前几个都是同步的，SIGINT、SIGTERM是异步的，它们在程序外部发生，同城是由程序的用户触发。

### 函数

#### abort

SIGABRT是由函数abort引发的信号，用于终止程序。

#### raise

函数raise用于显式地引发一个信号（由参数指定），程序对这类信号的反应和哪些自主发生的信号都是相同的。可以调用这个函数，对信号处理函数进程测试。

#### signal

当一个信号发生时，程序可以使用三种方式对它作出反应。缺省的反应是由编译器定义的，通常是终止程序。程序也可以指定其他行为对函数作出反应，信号可以被忽略或者设置一个信号处理函数。

#### sigaction

原型：int sigaction(int sig, const struct sigaction \*act, struct sigaction \*oact);

说明：

1. 阻塞：sigaction函数具有阻塞功能，比如SIGINT信号来了，进入信号处理函数，默认情况下，在信号处理函数未完成之前，如果又来了一个SIGINT信号，其将被阻塞，只有信号处理函数处理完毕，才会对后来的SIGINT再进程处理，同时后续无论来多少个SIGINT，仅需要一个SIGINT，sigaction会对后续SIGINT进行排队合并处理；
2. sa\_mask，信号屏蔽集，可以通过函数sigemptyset/sigaddset等来清空和增加需要屏蔽的信号；
3. sa\_flag，如果取值为0，则表示默认行为。

sigaction与signal区别：

signal比sigaction简单，但是signal注册的信号在sa\_handler被调用之前会把信号的sa\_handler指针恢复，而sigaction注册的信号在处理信号时不会恢复sa\_handler指针。所以，用signal函数注册的信号处理函数只会被调用一次，之后收到这个信号将按照默认处理方式处理，如果想一直处理这个信号的话就得在信号处理函数中再用signal注册一次，一般都在信号处理函数开始处调用signal注册一次这个信号，虽然这样可以一直能处理这个信号，但是可以看出，在sa\_handler指针恢复到再次调用signal注册信号期间如果收到这个信号，那么这个信号就按照默认处理方式，如果是INT之类的信号，进程就有可能退出。虽然这种概率很小，但是还是存在风险的。因此更好的做法是，除了SIG\_IGN、SIG\_DFL之外，最好用sigaction来代替signal注册信号。

#### sigemptyset

#### sigaddset

#### sigprocmask

## 消息队列

### 概述

消息队列就是消息的一个链表，它允许一个或多个进程向它写消息，一个或多个进程从中读消息。具有一定的FIFO的特性，但是可实现消息的随即查询。这些消息存在于内核中，由“队列ID”来标识。

消息队列的实现包括创建和打开队列、添加消息、读取消息和控制消息队列这四种操作。

消息队列是由消息的链表，存储在内核中并由消息队列标识符标识。消息队列克服了信号传递信号量少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

消息队列传递的信息还是比较大的，管道信息传递较少，信号更少。为了解决管道、有名管道的弊端，出现了消息队列，在某个进程往另一个进程写入消息之前，并不需要另外某个进程在该队列上等待消息的到达。

与管道和FIFO不同，进程可以在没有另外一个进程等待读的情况下进行写。另外一方面，管道和FIFO一旦相关进程都关闭并退出后，里面的数据也就没有了，但是对于消息队列，一个进程往消息队列中写入数据后退出，另外一个进程仍然可以打开并读取消息。消息队列与UNIX域套接字相比，在速度上没有多少优势。

### 消息队列属性

struct msqid\_ds{

struct ipc\_perm msg\_perm;

msgqnum\_t msg\_qnum; /\*of messages on queue\*/

sglen\_t msg\_qbytes; /\*max of bytes on queue\*/

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

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

time\_t msg\_stime; /\*last msgsnd time\*/

time\_t msg\_ctime; /\*last-change time\*/

}

### 函数

##### msgget

#include <sys/msg.h>

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

返回值：成功返回消息队列ID，出错返回-1。

flag：

ICP\_CREAT,IPC\_EXCL，权限组合

##### msgctl

消息队列操作

#include <sys/msg.h>

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

返回：成功返回0，出错返回-1。

cmd:

IPC\_STAT 获取消息队列的属性

IPC\_SET 设置属性

IPC\_RMID 删除队列

##### msgsnd

发送消息：

#include <sys/msg.h>

int msgsnd(int msgqid,const void \*ptr,size\_t nbytes,int flag);

返回：成功返回0，出错返回-1。

ptr:

struct mymesg{

long mtype; /\*positive message type\*/

char mtext[512]; /\*message data,of length nbytes\*/

}

flag:

IPC\_NOWWAIT

##### msgrcv

接受消息：

#include <sys/msg.h>

ssize\_t msgrcv(int msgqid,void \*ptr,size\_t nbytes,long type,int flag);

返回：成功返回消息的数据部分长度，出错返回-1。

type = 0:获得消息队列中第一个消息

type > 0:获得消息队列中类型为type的第一个消息

type < 0:获得消息队列中小于或等于type绝对值的消息(类型最小的)

## 共享内存

### 概述

共享内存区域是被多个进程共享的一部分物理内存。如果多个进程都把该内存区域映射到自己的虚拟地址空间，则这些进程就都可以直接访问该共享内存区域，从而可以通过该区域进行通信。

共享内存是进程间共享数据的一种最快的方法，一个进程向共享内存区域写入数据，共享这个内存区域的所有进程就可以立刻看到其中的内容。

共享内存就是映射一段能被其他进程所访问的内存（一旦这样的内存区域映射到共享它的进程的地址空间，这些进程间数据的传递就不再涉及内核，往共享内存区域存放信息或从中取走消息的进程间通常需要某种形式的同步），这段共享内存由一个进程创建，但多个进程都可以访问。

共享内存是最快的IPC方式，它是针对其他进程间通信方式效率低而专门设计的，它往往与其他通信机制如信号量配合使用，来实现进程间的同步和通信。

#### 方法

Posix提供了两种在无亲缘关系进程间共享内存的方法：

1. 内存映射文件：open函数打开，mmap函数把得到的描述符映射到当前进程地址空间中的一个文件；
2. 共享内存区对象：shm\_open打开一个IPC name，所返回的描述符由mmap映射到当前进程的地址空间。

#### 步骤

共享内存实现分为两个步骤：

创建共享内存，使用shmget函数。

映射共享内存，将这段创建的共享内存映射到具体的进程空间去，使用shmat函数。

### 共享内存段属性

struct shmid\_ds{

struct ipc\_perm shm\_perm;

size\_t shm\_segsz; /\*size of segment in bytes\*/

pid\_t shm\_lpid; /\*pid of last shmop()\*/

pid\_t shm\_cpid; /\*pid of creator\*/

shmatt\_t shm\_nattch; /\*number of current attaches\*/

time\_t shm\_atime; /\*last-attach time\*/

time\_t shm\_dtime; /\*last-detach time\*/

time\_t shm\_ctime; /\*last-change time\*/

}

### 创建共享内存对象

#incldue <sys/shm.h>

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

返回值：如果成功，返回共享内存段标识符。如果失败，则返回-1。

原型:errno

EINVAL 无效的内存段大小

EEXIST 内存段已经存在，无法创建

EIDRM 内存段已经被删除

EACCES 权限不够

ENOMEM 没有足够的内存来创建内存段

#### open

#### shm\_open

#### shm\_unlink

#### ftruncate

### 服务端进程

服务进程调用shmget建立共享内存对象

同时制定大小及访问许可

服务进程将共享内存对象的标识符通知客户进程

服务进程附加共享内存对象

获得共享内存在该进程空间中的地址

服务进程通过该地址访问该共享内存

### 客户端进程

客户进程获得共享内存对象的标识符

客户进程附加共享内存对象

获得共享内存在该进程空间中的地址

客户进程通过改地址访问该共享内存

## 信号量

### 概述

信号量（semaphore），有时称为信号灯，是多线程环境下使用的一种设施，可以用来保证两个或多个关键代码段不被并发调用（就是具有原子性的计数器）。

信号量是一个计数器，可用来控制多个进程对共享资源的访问。它常常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因为，作为进程间以及同一进程内不同线程之间的同步手段。

信号量用于多进程或者同一进程的不同线程之间的通信。在进入一个关键代码之前，线程必须获取一个信号量，一旦该关键代码段完成了，那么该线程必须释放信号量，其他想进入该关键代码的线程必须等待直到第一个线程释放信号量。

进程利用信号量获得共享资源步骤：

测试控制该资源的信号量

若信号量为正，则进程可以使用该资源

使用资源时，进程将该信号量减1

不再使用资源时，进程将该信号量值加1

若信号量为0，则进程进入睡眠状态

### 流程

它的主要流程如下：

检查控制该资源的信号量

如果信号量值大于0，则资源可用，并且将其减1，表示当前已被使用

如果信号量值为0，则进程休眠直至信号量值大于0

也就是说，它实际上是提供了一个不同进程或者进程的不同线程之间访问同步的手段。

### 分类

Unix中涉及的信号量包括：

1. Posix有名信号量（又称有名信号量）：使用Posix IPC名字标识，可用于进程或线程间的同步；
2. Posix基于内存的信号量（又称无名信号量）：存放在共享内存区中，可用于进程或线程间的同步；
3. SystemV信号量：在内核中维护，可用于进程或线程间的同步。

**整型信号量：**信号量是整数

**记录型信号量：**每个信号量s处于一个整数值s.value（计数）外，还有一个进程等待队列sL，其中是在阻塞在该信号量的各个进程的标识

**二进制信号量：**只允许信号量取0或1的值

### 信号量属性

struct semid\_ds{

struct ipc\_perm sem\_perm;

unsigned short sem\_nsems; /\*of semaphores in set\*/

time\_t sem\_otime; /\*last semop time\*/

time\_t sem\_ctime; /\*last change time\*/

}

### 创建信号量集

#incldue <sys/sem.h>

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

返回值：成功返回信号量集ID，出错返回-1。

flag

IPC\_CREAT,IPC\_EXCL,权限组合

### 信号集控制

#include <sys/sem.h>

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

union semum{

int val;

struct semid\_ds buf;

usigned short \*array;

}

通过cmd参数设定对信号量要执行的操作:

IPC\_STAT 获取信号量集的属性

IPC\_SET 设置信号量集的属性

IPC\_RMID 删除信号量集

GETVAL 返回semnum信号量的值

SETVAL 设置semnum信号量的值

GETTALL 获取所有信号量的值

SETALL 设置所有信号量的初始值

信号量集的操作

#include <sys/sem.h>

int semop(int semid,struct sembuf \*semop,size\_t nops);

返回：成功返回0，出错返回-1。

struct sembuf{

unsigned short sem\_num; /\*member in set\*/

short sem\_op; /\*operation(negative,0,positive)\*/

short sem\_flg; /\*IPC\_NOWAIT,SEM\_UNDO\*/

}

基于基本信号量的up和down操作(PV操作)

可用于进程间的互斥和同步

### 初始化

初始化也叫建立（create），原型如下：

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

### 等信号

等信号也叫挂起（suspend），原型如下：

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

### 给信号

给信号（signal）或者发信号（psot），原型如下：

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

### 清理

## 套接字