# linux进程通信（IPC）

## 目的

1. 数据传输
2. 资源共享
3. 通知事件
4. 进程控制

## 发展

Linux进程间通信（IPC）由以下几部分发展而来

1. UNIX进程间通信
2. 基于System V进程间通信
3. POSIX进程间通信

## 分类

1. 管道与有名管道、
2. 信号量
3. 共享内存
4. 消息队列
5. 套接字
6. 信号

## 管道通信

管道通信是单向的、先进先出的，他把一个进程的输出和另一个进程的输入连接在一起。管道包括无名管道和有名管道两种，前者用于父进程和子进程之间的通信，后者可用于运行于同一系统中的任意两个进程间通信。

无名管道由pipe()函数创建

int pipe(int filedis[2])

当一个管道建立时，他会建立两个文件描述符：filedis[0]用于读管道，filedis[1]用于写管道。

管道关闭只需将这两个文件描述符关闭即可，可以使用普通的close函数关闭。

close(filedis[0])。

注意事项

必须在系统调用fork()前调用pipe(),否则子进程将不会继承文件描述符。

代码示例：

#include <unistd.h>

#include <sys/types.h>

#include <errno.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

int main()

{

int pipe\_fd[2];

pid\_t pid;

char buf\_r[100];

char\* p\_wbuf;

int r\_num;

memset(buf\_r,0,sizeof(buf\_r));

/\*创建管道\*/

if(pipe(pipe\_fd)<0)

{

printf("pipe create error\n");

return -1;

}

/\*创建子进程\*/

if((pid=fork())==0) //子进程执行序列

{

printf("\n");

close(pipe\_fd[1]);//子进程先关闭了管道的写端

sleep(2); /\*让父进程先运行，这样父进程先写子进程才有内容读\*/

if((r\_num=read(pipe\_fd[0],buf\_r,100))>0)

{

printf("%d numbers read from the pipe is %s\n",r\_num,buf\_r);

}

close(pipe\_fd[0]);

exit(0);

}

else if(pid>0) //父进程执行序列

{

close(pipe\_fd[0]); //父进程先关闭了管道的读端

if(write(pipe\_fd[1],"Hello",5)!=-1)

printf("parent write1 Hello!\n");

if(write(pipe\_fd[1]," Pipe",5)!=-1)

printf("parent write2 Pipe!\n");

close(pipe\_fd[1]);

waitpid(pid,NULL,0); /\*等待子进程结束\*/

exit(0);

}

return 0;

}

命名管道的创建

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

当打开FIFO时，非阻塞标志（O\_NONBLOCK）

将对以后的读写产生以下影响：

1. 没有使用，访问要求无法满足时进程将阻塞。
2. 使用时，访问要求无法满足时不阻塞，立刻出错返回。

## 信号通信

信号机制是UNIX系统中最为古老的进程通信机制，很多条件可以产生一个信号。

1. 当用户按某些按键时，产生信号。
2. 硬件异常产生信号：除数为零、无效的存储访问等等。
3. 进程用kill函数将信号发送给另一个进程。
4. 用户可用kill命令将信号发送给其他进程。

信号类型

SIGHUP:从终端发出的结束信号

SIGINT:来自键盘的中断信号（Ctrl + C）

SIGKILL:该信号结束接收信号的进程

SIGTERM:kill命令发出的信号

SIGCHLD:表示子进程停止或者结束的信号

SIGSTOP：来自键盘（Ctrl+Z）或者调试程序的停止执行

信号处理

1. 忽略此信号，但有两种信号却决不能被忽略。SIGKILL和SIGSTOP.这两种信号不能被忽略的原理是：他们向超级用户提供了一种终止或者停止进程的方法。
2. 执行用户希望的动作
3. 执行系统默认动作

信号发送

发送信号的主要函数有kill和raise。

区别：

Kill既可以向自身发送信号，也可以向其他进程发送此信号，而raise是向进程自身发送信号。

int kull(pid\_t pid,int signo)

int raise(int signo)

kill的pid参数有四种不同的情况：

1. pid>0

将信号发送给进程ID为pid的进程

1. pid==0

将信号发送给同组的进程

1. pid<0

将信号发送给其他进程组id等于pid绝对值的进程

4、pid== -1

将信号发送给所有进程。

信号处理

#include <signal.h>

void (\*signal (int signo, void (\*func)(int)))(int)

如 何 理 解 ？

typedef void (\*sighandler\_t)(int)

sighandler\_t signal(int signum, sighandler\_t handler))

Func可能的值是：

1、SIG\_IGN：忽略此信号

2、SIG\_DFL: 按系统默认方式处理

3、信号处理函数名：使用该函数

## 共享内存

共享内存是被多个进程共享的一部分物理内存。是进程间共享数据一种最快的方法。

共享内存实现分为两个步骤：

1. 创建共享内存，使用shmget函数。
2. 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。

创建

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

映射

int shmat(int shmid,char \*shmaddr,int flag)

删除

int shmdt(char \*shmaddr)

示例代码

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include "shm\_com.h"

int main(void)

{

int running=1;

void \*shared\_memory=(void \*)0;

struct shared\_use\_st \*shared\_stuff;

int shmid;

/\*创建共享内存\*/

shmid=shmget((key\_t)1234,sizeof(struct shared\_use\_st),0666|IPC\_CREAT);

if(shmid==-1)

{

fprintf(stderr,"shmget failed\n");

exit(EXIT\_FAILURE);

}

/\*映射共享内存\*/

shared\_memory=shmat(shmid,(void \*)0,0);

if(shared\_memory==(void \*)-1)

{

fprintf(stderr,"shmat failed\n");

exit(EXIT\_FAILURE);

}

printf("Memory attached at %X\n",(int)shared\_memory);

/\*让结构体指针指向这块共享内存\*/

shared\_stuff=(struct shared\_use\_st \*)shared\_memory;

/\*控制读写顺序\*/

shared\_stuff->written\_by\_you=0;

/\*循环的从共享内存中读数据，直到读到“end”为止\*/

while(running)

{

if(shared\_stuff->written\_by\_you)

{

printf("You wrote:%s",shared\_stuff->some\_text);

sleep(1); //读进程睡一秒，同时会导致写进程睡一秒，这样做到读了之后再写

shared\_stuff->written\_by\_you=0;

if(strncmp(shared\_stuff->some\_text,"end",3)==0)

{

running=0; //结束循环

}

}

}

/\*删除共享内存\*/

if(shmdt(shared\_memory)==-1)

{

fprintf(stderr,"shmdt failed\n");

exit(EXIT\_FAILURE);

}

exit(EXIT\_SUCCESS);

}

## 消息队列

消息队列克服了早期unix管道通信容量不足的缺点。

消息队列是一个消息的链表。可以把消息看做一个记录，具有特定的格式。

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。

系统V消息队列是随内核持续的，只有在内核重启或者人工删除时，该消息队列才会被删除。

键值

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，必须提供该消息队列的键值。

key\_t ftok(char \*pathname,char proj)

返回文件名对应的键值。

打开

int msgget(ket\_t key,int msgflg)

返回与键值key相对应的消息队列描述字。

发送消息

int msgsnd(int msqid,struct msgbuf\*msgsz,int msgflg)

向消息队列中发送一条消息。

消息格式

struct msgbuf

{

long mtype; //消息类型

char mtext[1];//消息数据首地址

};

接收消息

int msgrcv(int msqid, struct msgbuf \*msgp, int msgsz, long msgtyp, int msgflg)

功 能：从msqid代表的消息队列中读取一个msgtyp类型的消息,并把消息存储在msgp指向的的msgbuf结构中。在成功地读取了一条消息以后，队列中的这条消将 被删除。

代码示例：

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

struct my\_msg\_st

{

long int my\_msg\_type;

char some\_text[BUFSIZ];

};

int main(void)

{

int running=1;

int msgid;

struct my\_msg\_st some\_data;

long int msg\_to\_receive=0;

/\*创建消息队列\*/

msgid=msgget((key\_t)1234,0666 | IPC\_CREAT);

if(msgid==-1)

{

fprintf(stderr,"msgget failed with error: %d\n",errno);

exit(EXIT\_FAILURE);

}

/\*循环从消息队列中接收消息\*/

while(running)

{

/\*读取消息\*/

if(msgrcv(msgid,(void \*)&some\_data,BUFSIZ,msg\_to\_receive,0)==-1)

{

fprintf(stderr,"msgrcv failed with error: %d\n",errno);

exit(EXIT\_FAILURE);

}

printf("You wrote: %s",some\_data.some\_text);

/\*接收到的消息为“end”时结束循环\*/

if(strncmp(some\_data.some\_text,"end",3)==0)

{

running=0;

}

}

/\*从系统内核中移走消息队列\*/

if(msgctl(msgid,IPC\_RMID,0)==-1)

{

fprintf(stderr,"msgctl(IPC\_RMID) failed\n");

exit(EXIT\_FAILURE);

}

exit(EXIT\_SUCCESS);

}

## 信号量

信号量(又名：信号灯)与其他进程间通信方式不大相同，主要用途是保护临界资源。进程可以根据它判定是否能够访问某些共享资源。除了用于访问控制外，还可用于进程同步。信号量(Semaphore)，有时被称为信号灯，是在多线程环境下使用的一种设施，是可以用来保证两个或多个关键代码段不被[并发](http://baike.baidu.com/item/%E5%B9%B6%E5%8F%91)调用。在进入一个关键代码段之前，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。其它想进入该关键代码段的线程必须等待直到第一个线程释放信号量。为了完成这个过程，需要创建一个信号量VI，然后将Acquire Semaphore VI以及Release Semaphore VI分别放置在每个关键代码段的首末端。确认这些信号量VI引用的是初始创建的信号量。

描述

以一个停车场的运作为例。简单起见，假设停车场只有三个车位，一开始三个车位都是空的。这时如果同时来了五辆车，看门人允许其中三辆直接进入，然后放下车拦，剩下的车则必须在入口等待，此后来的车也都不得不在入口处等待。这时，有一辆车离开停车场，看门人得知后，打开车拦，放入外面的一辆进去，如果又离开两辆，则又可以放入两辆，如此往复。在这个停车场系统中，车位是公共资源，每辆车好比一个[线程](http://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B)，看门人起的就是信号量的作用。抽象的来讲，信号量的特性如下：信号量是一个非负整数（车位数），所有通过它的[线程](http://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B)/进程（车辆）都会将该整数减一（通过它当然是为了使用资源），当该整数值为零时，所有试图通过它的线程都将处于等待状态。在信号量上我们定义两种操作： Wait（等待） 和 Release（释放）。当一个线程调用Wait操作时，它要么得到资源然后将信号量减一，要么一直等下去（指放入阻塞队列），直到信号量大于等于一时。Release（释放）实际上是在信号量上执行加操作，对应于车辆离开停车场，该操作之所以叫做“释放”是因为释放了由信号量守护的资源。

创建

同[共享内存](http://baike.baidu.com/item/%E5%85%B1%E4%BA%AB%E5%86%85%E5%AD%98)一样，系统中同样需要为[信号量集](http://baike.baidu.com/item/%E4%BF%A1%E5%8F%B7%E9%87%8F%E9%9B%86)定制一系列专有的操作函数（semget，semctl等）。系统命令ipcs可查看当前的系统IPC的状态，在命令后使用-s参数。使用函数semget可以创建或者获得一个信号量集ID，函数原型如下：

#include <sys/shm.h>

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

操作

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

对信号灯进行控制。