重庆理工大学/电气学院
CHONGQING UNIVERSITY OF TECHNOLOGY

嵌入式Linux系统开发教程
—基于ARM处理器通用平台（arm9-arm11-contexA系列）

丛超

2025年5月

1 管道通信

2 消息队列

3 信号与信号量

4 共享内存

Linux系统中进程通信的机制继承自Unix，后经贝尔实验室与BSD对进程间通讯手段的改进与扩充，以及POSIX标准对Unix标准的统一，发展出如今Linux系统中使用的进程通信（IPC）机制，即包含管道通信、信号量、消息队列、共享内存以及socket通信等的诸多通讯机制。

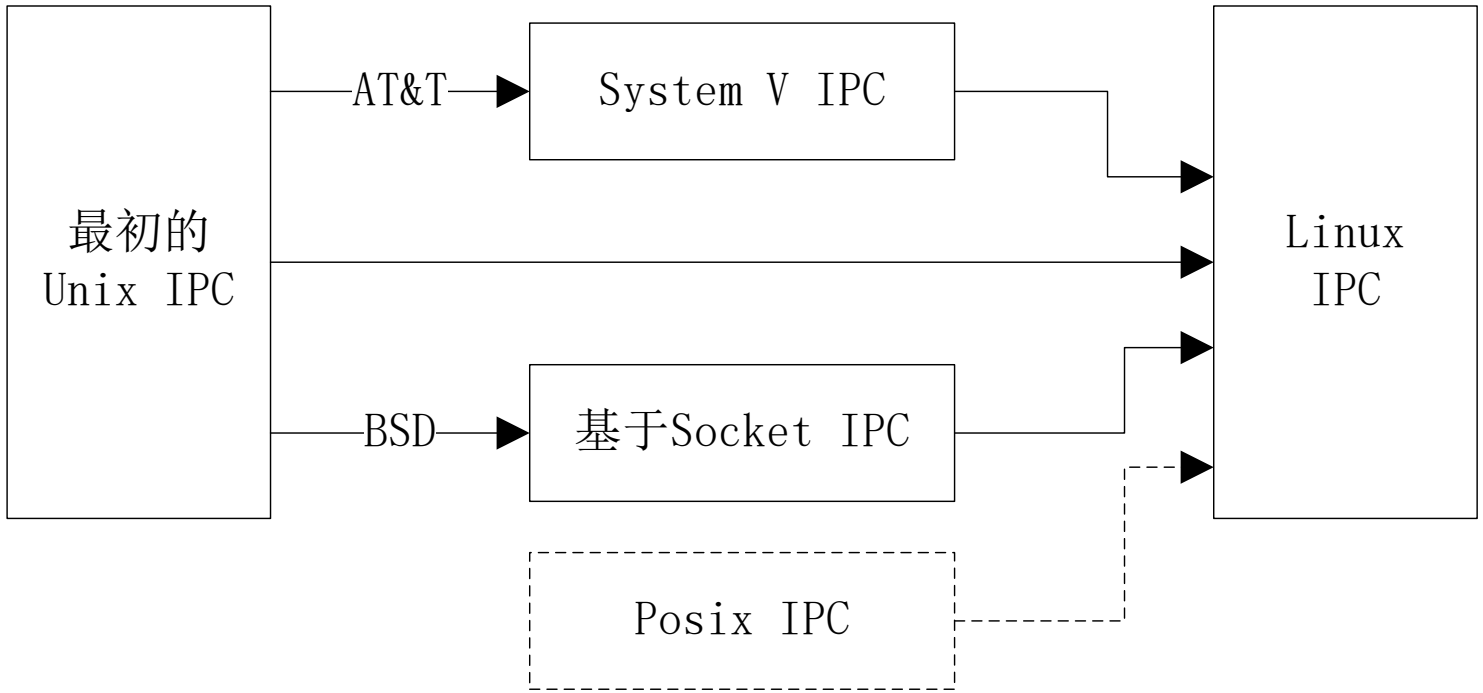

图7-1 Linux所继承的进程间通信

进程通信概述

Linux下进程间通信的几种主要手段简介：

1. 管道（pipe）及命名管道（named pipe）：管道可用于具有亲缘关系进程间的通信，有名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。
2. 信号（signal）：信号是比较复杂的通信方式，用于通知接收进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身；Linux除了支持Unix早期信号语义函数signal外，还支持语义符合Posix.1标准的信号函数sigaction。

Linux下进程间通信的几种主要手段简介：

3. 消息队列（message）：消息队列是消息的链接表，包括Posix消息队列systemV消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。

4. 共享内存：使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。

Linux下进程间通信的几种主要手段简介：

5. 信号量（semaphore）：主要作为进程间以及同一进程不同线程之间的同步手段。
6. 套接口（socket）：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：Linux和System V的变种都支持套接字。

01

管道通信

管道是Linux中最早支持的IPC机制，是一个连接两个进程的连接器，它实际上是在进程间开辟一个固定大小的缓冲区，需要发布信息的进程运行写操作，需要接收信息的进程运行读操作。

管道是半双工的，输入输出原则是先入先出(FIFO)，写入数据在管道的尾端，读取数据在管道的头部。如果要实现双向交互，必须创建两个管道。

管道分为三种：

（1）无名管道（2）命名管道（3）标准流管道

无名管道

用于父子进程之间的通信，没有磁盘节点，位于内存中，它仅作为一个内存对象存在，用完后就销毁了。

命名管道

用于任意进程之间的通信，具有文件名和磁盘节点，位于文件系统，读写的内部实现和普通文件不同，而是和无名管道一样采用字节流的方式。

标准流管道

用于获取指令运行结果集。

管道具有以下特点：

（1）管道是半双工的，数据只能向一个方向流动。

（2）需要双方通信时，需要建立起两个管道。

（3）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）。

（4）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，

7.2.1无名管道

用于父子进程之间通信，管道以先进先出方式保存一定数量的数据。使用管道的时候一个进程从管道的一端写，另一个进程从管道的另一端读。

在主进程中利用fork创建一个子进程，让父子进程同时拥有对同一管道的读写句柄，然后在相应进程中关闭不需要的句柄。

操作流程：pipe->read(write)->close

1.创建管道

表头文件

```
#include<unistd.h>
```

定义函数

```
int pipe(int fd[2])
```

参数：fd文件描述符数组，由函数填写数据，fd[0]用于管道的read端，fd[1]用于管道的write端。

返回值：返回0表示成功，返回-1表示失败。

2.管道读写

管道读、写数据使用read和write函数，采用字节流的方式，具有流动性。读数据时，每读一段数据，则管道内会清除已读走的数据。

（1）读管道时，若管道为空，则被阻塞，直到管道另一端write将数据写入到管道为止。若写端已关闭，则返回0。

（2）写管道时，若管道已满，则被阻塞，直到管道另一端read将管道内数据取走为止。若读端已关闭，则写端返回21，errno被设为EPIPE，进程还会收到SIGPIPE信号(默认处理是终止进程，该信号可以被捕捉)。

3.管道关闭

用close函数，在创建管道时，写端需要关闭f[0]描述符，读端需要关闭f[1]描述符。当进程关闭前，每个进程需要把没有关闭的描述符都进行关闭。

无名管道需要注意的问题：

（1）管道是半双工方式，数据只能单向传输。如果在两个进程之间相互传送数据，要建立两条管道。

（2）pipe调用必须在调用fork以前进行，否则子进程将无法继承文件描述符。

（3）使用无名管道互相连接的任意进程，必须位于一个相关的进程组中。

Example7.2.1-1：使用pipe()实现父子进程通信，要求父进程作为写端，子进程作为读端。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
int main(int argc,char **argv){
  int fd[2];   //定义文件描述符数组，0表示读，1表示写
  int err=pipe(fd);    //创建管道
  if (err==-1){
                              printf("pipe err\n");
                              exit(0);
                  }
   pid_t pid=fork();
                  if (pid==-1) exit(0);
   else if (pid==0){   //子进程-读
     close(fd[1]);   //关闭写端
     char buf[256]={0};
     int size=read(fd[0],buf,256);   //读数据
     if (size >0)
        printf("son ---- %s\n",buf); //如果管道为空时，则关闭管道则read返回0
     else printf("son read err\n");
          close(fd[0]);
          exit(0);
          }

else if (pid >0){    //父进程-写
          close(fd[0]);   //关闭读端
          char *str="1234567890";
          int size=write(fd[1],str,strlen(str));  //写数据
          sleep(5);
          close(fd[1]);
          wait(NULL);
          exit(0);
          }
}
```

Example7.2.1-2:父进程用管道将字符串“hello!\n”传给子进程并显示

```
#include <unistd.h>
#include <stdio.h>
int main()
{
   int filedes[2];   //定义文件描述符
   char buffer[80];  //定义数据存储区
   pipe(filedes);
   if(fork()>0)
   {/*父进程写hello!\n*/
      char s[ ] = "hello!\n";
      write(filedes[1],s,sizeof(s));
}
else
   {/*子进程读hello!\n*/
      read(filedes[0],buffer,80);
      printf("%s",buffer);
}
}
```

7.2.2有名管道

命名管道又称FIFO（First In First Out），它与无名管道不同之处在于，命名管道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，在文件系统中产生一个物理文件，其它进程只要访问该文件路径，就能彼此通过管道通信。在读数据端以只读方式打开管道文件，在写数据端以只写方式打开管道文件。

1.创建有名管道

表头文件

```
#include<sys/type.h>
#include<sys/stat.h>
```

定义函数

```
int mkfifo(char *pathname,mode_t mode);
```

参数：pathname 管道建立的临时文件，文件名在创建管道之前不能存在；

mode:管道的访问权限，如0666。

返回值：成功返回0，失败返回 -1。如果文件已存在则失败。

有名管道只需在写端创建，不需要在读端创建。

2.打开管道

使用open函数，默认设置为阻塞模式，不需要使用创建的方式，不需要在此处再设置访问权限。在读端以只读方式打开，在写端以只写方式打开。

3.管道读写

使用read和write函数。

（1）阻塞模式

读取数据时，以只读方式打开，若管道空，则被阻塞，直到写数据端写入数据为止。写入数据时，以只写方式打开，若管道已满，则被阻塞，直到读进程将数据读走。

（2）非阻塞模式

读取数据时，立即返回，管道没有数据时，返回0；有数据时，返回实际读取字节数。

写入数据时，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。

4.管道关闭

用close函数。进程关闭前，只需关闭各自的描述符即可。

5.文件移除

用unlink函数，将临时文件及时清除。在写端移除，不需要在读端移除。

Example7.2.2-1:使用FIFO实现没有亲缘关系进程间的通信

- 服务端代码**Example7.2.2-1-server.c:**

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
int main(){
    int err=mkfifo("/tmp/myfifo",0666);  //判断fifo文件是否存在
    if (err==-1){    //若fifo文件不存在就创建fifo，若存在则提示
            printf(" file name is exist\n");
            return 0;
    }
    int fd=open("/tmp/myfifo",O_WRONLY);  //以读写方式打开文件
    if (fd==-1){
            printf(" open err\n");
            unlink("/tmp/myfifo");
            return 0;
    }
    char buf[256];
    while(1){       //循环写入数据
        scanf("%s",buf);
        int size=write(fd,buf,strlen(buf));
        printf("write  size =%d,buf=%s\n",size,buf);
        if (strcmp(buf,"q")==0) break;
    }
    close(fd);
    unlink("/tmp/myfifo");  //移除文件
}
```

- 客户端代码**Example7.2.2-1-client.c**：

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
int main(){
    int  fd=open("/tmp/myfifo",O_RDONLY);    //以只读方式打开文件
    if (fd==-1){
        printf(" open err\n");
        return 0;
    }
char buf[256];
    while(1){   //不断读取fifo中的数据并打印
        bzero(buf,256);
        int size=read(fd,buf,256);
        printf("read  size =%d,buf=%s\n",size,buf);
        if (strcmp(buf,"q")==0) break;
    }
    close(fd);      //关闭文件
}
```

程序分析：由于是没有亲缘关系的进程间通信，因此需要在两段程序中实现。Example7.2.2-1-server实现FIFO写操作，Example7.2.2-1-client实现FIFO读操作。

打开两个终端，先运行服务端，结果为：

```
root@ubuntu:/home/linux/chapter7# ./Example7.2.2-1-server
hello
write  size =5,buf=hello
how are you!
write  size =3,buf=how
write  size =3,buf=are
write  size =4,buf=you!
```

再运行客户端，结果为：

```
root@ubuntu:/home/linux/chapter7# ./Example7.2.2-1-client
read  size =5,buf=hello
read  size =10,buf=howareyou!
```

02

信号与信号量

消息队列的实质就是一个存放消息的链表，该链表由内核维护。可以把消息看作一个记录，具有特定的格式。进程可以向其中按照一定的规则添加新消息；另一些进程则可以从消息队列中读走消息。

目前主要有两种类型的消息队列：POSIX消息队列以及System V消息队列，System V消息队列目前被大量使用。该消息队列是随内核持续的，只有在内核重起或者人工删除时，该消息队列才会被删除。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。

消息的发送不是同步机制，先发送到内核，只要消息没有被清除，则另一个程序无论何时打开都可以读取消息。消息可以用在同一进程之间，也可以用在不同进程之间。消息结构体必须自己定义，并按系统的要求定义。

```
struct msgbuf{ //结构体的名称自己定义
    long mtype; //必须是long，变量名必须是mtype
    char mdata[256]; //必须是char类型数组，数组名和数组长度由自己定义
};
```

mtype 是消息标识，大多用宏定义一组消息指令。

mdata 是对消息的文件说明，或信息。

7.3.1键值

表头文件

```
#include<sys/types.h>
#include<sys/ipc.h>
```

定义函数

```
key_t ftok(const char *pathname, int proj_id);
```

函数说明 系统建立IPC通讯时必须指定一个ID值。通常情况下，该ID值通过ftok函数得到。ftok（）函数使用由给定路径名命名的文件的身份以及proj_id的最低有效8位来生成key_t类型的系统V IPC密钥。

返回成功后，将返回生成的key_t值。失败时返回-1。

7.3.2打开/创建消息队列

表头文件

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

定义函数

```
int msgget(key_t key, int msgflg)
```

int msgget(key_t key, int msgflg)

函数说明

key：键值，由ftok获得，通常为一个整数，若键值为IPC_PRIVATE，将会创建一个只能被创建消息队列的进程读写的消息队列。

msgflg：标志位，用于设置消息队列的创建方式或权限，通常由一个9位的权限与以下值进行位操作后获得：

（1）IPC_CREAT：若内核中不存在指定消息队列，该函数会创建一个消息队列；若内核中已存在指定消息队列，则获取该消息队列；

（2）IPC_EXCL：与IPC_CREAT一同使用，表示如果要创建的消息队列已经存在，则返回错误。

（3）IPC_NOWAIT：读写消息队列要求无法得到满足时，不阻塞。

返回值 如果成功，则返回值将是消息队列标识符（非负整数），否则为-1。

在以下两种情况下，将创建一个新的消息队列：

（1）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。

（2）key参数为IPC_PRIVATE。

创建消息队列格式

```
int open_queue(key_t keyval)
{
intqid;
if((qid=msgget(keyval,IPC_CREAT))==-1)
{
  return(-1);
}
  return(qid);
}
```

7.3.4接收消息

表头文件

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

定义函数

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

定义函数

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

函数说明　从msqid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。在成功地读取了一条消息以后，队列中的这条消息将被删除。

返回值 操作成功返回0，失败返回-1。

7.3.5消息控制

表头文件

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

定义函数

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf)
```

函数说明　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：

（1）IPC_STAT：获取消息队列信息，返回信息存贮在buf指向msqid结构中。

（2）IPC_SET：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。

（3）IPC_RMID：删除msqid标识的消息队列。

参数buf是一个缓冲区，用于传递属性值给指定消息队列，或从指定消息队列获取属性值，其功能视参数cmd而定。其数据类型struct msqid为一个结构体，内核为每个消息队列维护了一个msqid_ds结构，用于消息队列的管理。

返回值 如果成功IPC_STAT、IPC_SET、IPC_RMID返回0，发生错误返回-1。

Example7.3.5-1：创建一个消息队列，实现向队列中存放数据和读取数据。

03

信号与信号量

信号是进程在运行过程中，由自身产生或由进程外部发过来，用来通知进程发生了异步事件。每个信号用一个整型常量宏表示，以SIG开头，比如SIGCHLD、SIGINT等，它们在系统头文件<signal.h>中定义。

信号的产生方式：

（1）程序执行错误，如除零内存越界，内核发送信号给程序；

（2）由另一个进程（信号函数）发送过来的信号；

（3）由用户控制中断产生信号，Ctrl+C终止程序信号；

（4）子进程结束时向父进程发送信号SIGCHLD信号；

（5）程序中设定的定时器产生信号SIGALRM信号；

7.4.1信号处理的方式


Linux系统中的信号
发送状态
当某种情况驱使内核发送信号时，信号会有一个短暂的发送状态；
阻塞状态
由于某种原因，发送的信号无法被传递，将处于阻塞状态；
未决状态
发送的信号被阻塞，无法到达进程，内核就会将该信号的状态设置为未决；
递达状态
若信号发送后没有阻塞，信号就会被成功传递并到达进程，此时为递达状态；
处理状态
信号被递达后会被立刻处理，此时信号处于处理状态。
信号递达进程后才可能被处理

信号的处理方式有三种：忽略、捕捉和执行默认动作。

1.缺省处理

接收默认处理的进程通常会导致进程本身消亡。例如连接到终端的进程，用户按下CTRL+C，将导致内核向进程发送一个SIGINT的信号，进程如果不对该信号做特殊的处理，系统将采用默认的方式处理该信号，即终止进程的执行。

2.忽略信号

进程可以通过代码，显示地忽略某个信号的处理，则进程遇到信号后，不会有任何动作。但有些信号不能被忽略。

3.捕捉信号

进程可以事先注册信号处理函数，当接收到信号时，由信号处理函数自动捕捉并且处理信号。

7.4.2 信号操作指令

1.信号指令

（1）kill [-s <信息名称或编号>][程序]

参数 -s 用于发送指定信号，程序是指进程的PID或工作编号。

（2）kill [-l <信息编号>]

参数 -l 用于显示所有信号。

1. kill 15523 //杀死pid是15523的进程，缺省-s的情况发送的是SIGKILL信号。
2. kill -s SIGHUP 12523 //向pid是12523的进程发送SIGHUP信号。

2.信号说明

不同事件发生时，信号会被发送给对应进程，每个进程对应的事件如下（部分）：

（1）SIGHUP：用户终端连接结束时发出，通常是在终端的控制进程结束时，默认终止进程。

（2）SIGINT：用户键入INTR字符（通常是Ctrl-C）时发出，用于通知前台进程组终止进程。

（3）SIGQUIT：用户键入QUIT字符（通常是Ctrl-/）来控制，类似于一个程序错误信号。

（4）……

3. 处理信号

信号的产生是一个异步事件，进程不知道信号何时会递达，也不会等待信号到来，因此进程需要能捕获到递达的信号；此外，用户可通过为信号自定义的处理动作，让进程在接收到信号后执行指定的操作。Linux系统中为用户提供了两个捕获信号——signal()和sigaction()，用于自定义信号处理方法。

（1）signal函数

表头文件

#include <signal.h>

定义函数

typedef void (*sighandler_t)(int);　//函数指针类型

sighandler_t signal(int signum, sighandler_t handler);

参数：signum 是要捕捉的信号，信号编号；

handler 是用于处理信号的函数，该参数还可以如下选择：

①SIG_DFN：采用缺省方式处理信号。

②SIG_IGN：忽略信号。

返回值 成功返回原来的信号处理函数指针，失败返回SIG_ERR。

Example7.4.2-1： 用signal捕捉信号

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
 void f1(int signum){   //自定义信号处理函数
        switch(signum){
        case SIGINT:
                printf("is SIGINT\n");
                break;
        case SIGQUIT:
                printf("is SIGQUIT\n");
                exit(0);
        case SIGKILL:
                printf("is SIGKILL\n");
        }
}
int main(int argc,char **argv){
    signal(SIGINT,f1);  //可捕捉信号  Ctrl+c
    signal(SIGQUIT,f1); //可捕捉信号  Ctrl+\
    signal(SIGKILL,f1); //无法捕捉的信号   kill
pid
    while(1){
        sleep(1);
    }
 }
```

（2）sigaction函数

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);

参数：signum是信号编号；act是传入参数，包含自定义信息处理函数和一些携带信息；oldact 是传出参数，包含旧的信息处理函数。act和oldact是自定义结构体类型的数据，其类型定义如下：

```
struct sigaction {
    void(*sa_handler)(int);
    void(*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void(*sa_restorer)(void);
};
```

4. 发送信号

系统调用中发送信号常用的函数有kill()、raise()、abort()等，其中kill是最常用的函数，该函数的作用是给指定进程发送信号，但是否杀死进程，取决于所发送信号的默认动作。

（1）kill和raise函数

表头文件

#include<signal.h>

定义函数

int kill(pid_t pid,int signo);

功能：向指定的进程发送信号。

定义函数

int raise(int sig);

功能：把信号发送到当前的进程。

参数：sig为要发送信号的编号，使用kill()函数可以实现与该函数相同的功能，该函数与kill()之间的关系如下：

raise(sig==kill(getpid(),sig)

Example7.4.2-3：使用fork（）函数创建一个子进程，在父进程中使用kill（）发送信号，杀死子进程，并使用raise（）函数自杀。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <wait.h>
int main(int argc,char **argv){
        pid_t pid=fork();
        if (pid==-1) return 0;
        else if (pid==0) { //子进程
                printf("son ----- pid=%d\n",getpid());
                while(1)
    sleep(1);
        }
        else { //父进程
                int i;
printf("parent ----- pid=%d\n",getpid());
                sleep(10);
                kill(pid,SIGKILL); //杀死子进程
        wait(NULL);
                printf("kill son over\n");
sleep(3);
    raise(SIGKILL); //父进程自杀
    for(i=0;i<10;i++)
        printf("parent --kill waiting\n");
        sleep(1);
    }
}
```

（2）alarm和pause函数

表头文件

```
#include <unistd.h>
```

定义函数

```
int pause(void);
```

函数说明：造成进程主动挂起，等待信号唤醒。调用该函数后进程将主动放弃CPU，进入阻塞状态，直到有信号递达将其唤醒，才继续工作。

定义函数

unsigned int alarm(unsigned int seconds);

函数说明：计时器，驱使内核在指定秒数后发送信号到调用该函数的进程。

参数：seconds为时间(秒)，通过设置参数倒计时，当时间到达时发出SIGALRM信号。

Example7.4.2-4： 时钟计时器

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
int b=0;
   void f1(int a){
         b+=20;
         printf("%d:%d\n",b/60 ,b%60);
         alarm(20);
   }
 int main(int argc,char **argv){
         signal(SIGALRM,f1);  //可捕捉信号
         alarm(20);      //时钟计时器可以唤醒
sleep
         pause();        //任何信号都可以激活
暂停
         sleep(10);
         while(1) {
               sleep(1);
               printf("---- \n");
         }
    }
```

（3）abort函数

表头文件

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

定义函数

```
void abort(void);
```

函数说明：abort()是使异常程序终止，同时发送SIGABRT信号给调用进程。该函数在调用之时会先解除阻塞信号SIGABRT，然后才发送信号给自己。它不会返回任何值，可以视为百分百调用成功。

Example7.4.2-6：

```
#include <stdio.h>
#include <stdlib.h>
void main( void ){
    FILE *stream;
    if( (stream = fopen( "NOSUCHF.ILE", "r" )) == NULL )
     {
        perror( "Couldn't open file" );
        abort();
      }
     else
        fclose( stream );
     }
```

信号量

1. 基本概念

信号量是System V机制，System V机制包括信号量、消息队列、共享内存。该机制为保证多个进程之间通信，需要提供一个每个进程都必须一致的主键值，来识别另一个进程已创建IPC。

2.信号量原理

在多任务的操作系统中，如多个进程会为了完成一个任务会相互协作，这就是进程间的同步，有时为了争夺有限的系统资源会进入竞争状态，这就是进程的互斥关系。

信号量是用来解决进程间同步和互斥问题的一种进程之间的通信机制。

信号量的实现原理是PV原子操作。P操作使信号量减1，如果信号量为0，则被阻塞，直到信号量大于0；V操作使信号量加1。

信号量操作:

（1）初始化（initialize），也叫做建立（create）；

（2）等信号（wait），也叫做挂起（pend）；

（3）给信号（signal）或发信号（post）。

信号量分类：

（1）整型信号量（integer semaphore)：信号量是整数；

（2）记录型信号量（record semaphore)：每个信号量s除一个整数值s.value（计数）外，还有一个进程等待队列s.L，其中是阻塞在该信号量的各个进程的标识；

（3）二进制信号量(binary semaphore)：只允许信号量取0或1值。

每个信号量至少须记录两个信息：信号量的值和等待该信号量的进程队列。

7.5.1信号量创建

表头文件

```
#include <sys/shm.h>
```

定义函数

```
int semget( key_t key, int nsems, int flag);
```

函数说明　使用函数semget可以创建或者获得一个信号量集ID，函数中参数key用来变换成一个标识符，每一个IPC对象与一个key相对应。

定义函数

int semget(key_t key, int nsems, int flag);

参数：key是主键，用于其它进程中进行识别的唯一标识；nsems 是信号量的个数；flag 是访问权限和创建表示，可选择：

① IPC_CREAT：创建；

②IPC_EXCL：防止重复创建。

返回值：成功返回信号量的ID，失败返回-1。

7.5.2信号量操作

表头文件

```
#include <sys/sem.h>
```

1.控制信号量

```
int semctl(int semid, int semnum, int cmd,union semun arg);
```

函数说明：对信号量或信号量集进行控制。

参数：semid为信号量ID，是semget返回值；semnum为信号量的编号；cmd为各种操作指令，如下：

（1）IPC_STAT 获取信号量信息；

（2）SETVAL 设置信号量的值；

（3）GETVAL 获取信号量的值；

（4）IPC_RMID 删除信号量。

arg是需要设置或获取信号量的结构，这个联合体的结构需要自己定义。

```
union semun{
    int val;
    struct semid_ds *buf;
unsigned short *array;
};
```

2.操作信号量

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

函数说明：改变信号量的值。

参数：semid 是信号量ID；nsops是操作数组sops中的操作个数，通常为1；sops是信号量结构指针，定义如下：

```
struct sembuf{
short sem_num; //信号量编号,通常设为0
short sem_op; //信号量操作，-1则表p操作，1则表示为V操作
short sem_flg; //信号量操作标志，通常设为SEM_UNDO
//在没释放信号量时，系统自动释放
};
```

Example7.5.2-1:semop函数使用

```
1.  #include <sys/types.h>
2.  #include <sys/ipc.h>
3.  #include <sys/sem.h>
4.  #include <stdio.h>
5.  #include <stdlib.h>
6.  int main( void )
7.  {
8.    int sem_id;
9.    int nsems = 1;
10.   int flags = 0666;
11.   struct sembuf buf;
12.   sem_id = semget(IPC_PRIVATE, nsems, flags);
13.   /*创建一个新的信号量集*/
14.   if ( sem_id < 0 )
15.   {
16.     perror("semget");
17.     exit(1);
18.   }
19.   /*输出相应的信号量集标识符*/
20.   printf ("successfully created a semaphore: %d\n",sem_id);
21.   buf.sem_num = 0;
22.   /*定义一个信号量操作*/
23.   buf.sem_op = 1;
24.   /*执行释放资源操作*/
25.   buf.sem_flg = IPC_NOWAIT;
26.   /*定义semop函数的行为*/
27.   if ((semop( sem_id, &buf, nsems))<0)
28.   {
29.     /*执行操作*/
30.     perror ("semop");
31.     exit (1);
32.   }
33.   system ( "ipcs -s ");
34.   /*查看系统IPC状态*/
35.   exit ( 0 );
36. }
```

04

内存共享

共享内存指在多处理器的计算机系统中，可以被不同中央处理器访问的大容量内存。

由于多个CPU需要快速访问存储器，这样就要对存储器进行缓存。

任何一个缓存的数据被更新后，由于其他处理器也可能要存取，共享内存就需要立即更新，否则不同的处理器可能用到不同的数据。

共享内存是Unix下的多进程之间的通信方法，这种方法通常用于一个程序的多进程间通信，实际上多个程序间也可以通过共享内存来传递信息。

7.6.1共享内存创建

表头文件

```
#include <sys/shm.h>
```

定义函数

```
int shmget( key_t key, size_t size, int flag );
```

函数说明　函数中参数key用来变换成一个标识符，而且每一个IPC对象与一个key相对应。当新建一个共享内存段时，size参数为要请求的内存长度（以字节为单位）

flag为标志，可选择的标志有：

①S_IRUSR：属主的读权限；

②S_IWUSR：属主的写权限；

③S_IROTH：其它用户读权限；

④S_IWOTH：其它用户写权限；

⑤S_IRGRP：属组的读权限；

⑥S_IWGRP：属组的写权限；

⑦IPC_CREAT ：创建；

⑧IPC_EXCL：如果已创建，则再次创建则失败。

返回值：失败是-1，成功返回的是内存ID。

Example7.6.1-1：shmget函数使用

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <stdio.h>
#define BUFSZ 4096
int main ( void )
{
  int shm_id; /*共享内存标识符*/
  shm_id=shmget(IPC_PRIVATE,BUFSZ,0666);
  if(shm_id<0)
  {/*创建共享内存*/
    perror("shmget");
    exit(1);
  }
  printf("successfully created segment : %d \n", shm_id ) ;
  system("ipcs -m");/*调用ipcs命令查看IPC*/
  exit(0);
}
```

7.6.2共享内存的操作

由于共享内存这一特殊的资源类型，使它不同于普通的文件，因此，系统需为其提供专有的操作函数。

表头文件

#include <sys/shm.h>

定义函数

int shmctl(int shm_id, int cmd, struct shmid_ds *buf);

函数说明 函数中参数shm_id为所要操作的共享内存段的标识符，struct shmid_ds型指针参数buf的作用与参数cmd的值相关，参数cmd指明了所要进行的操作。

cmd命令值含义

cmd的值	意 义
IPC_STAT	取shm_id所指向内存共享段的shmid_ds结构，对参数buf指向的结构赋值
IPC_SET	使用buf指向的结构对sh_mid段的相关结构赋值，只对以下几个域有作用，shm_perm. uid shm_perm.gid以及shm_perm.mode 注意此命令只有具备以下条件的进程才可以请求： 1．进程的用户ID等于shm_perm.cuid或者等于shm_perm.uid 2．超级用户特权进程
IPC_RMID	删除shm_id所指向的共享内存段，只有当shmid_ds结构的shm_nattch域为零时，才会真正执行删除命令，否则不会删除该段。注意此命令的请求规则与IPC_SET命令相同
SHM_LOCK	锁定共享内存段在内存，此命令只能由超级用户请求
SHM_UNLOCK	对共享内存段解锁，此命令只能由超级用户请求

需要注意的是，共享内存与消息队列以及信号量相同，在使用完毕后都应该进行释放。

另外，当调用fork()函数创建子进程时，子进程会继承父进程已绑定的共享内存；当调用exec()函数更改子进程功能以及调用exit()函数时，子进程中都会解除与共享内存的映射关系，因此在必要时对共享内存进行删除。

7.6.3共享内存段连接到本进程空间

表头文件

#include <sys/shm.h>

定义函数

void *shmat(int shm_id, const void *addr, int flag);

函数说明 函数中参数shm_id指定要引入的共享内存，参数addr与flag组合说明要引入的地址值。

返回值 函数成功执行返回值为实际引入的地址，失败返回–1。shmat函数成功执行会将shm_id段的shmid_ds结构的shm_nattch计数器的值加1。

7.6.4共享内存解除

表头文件

```
#include <sys/shm.h>
```

定义函数

```
int shmdt( void *addr);
```

函数说明　当对共享内存段操作结束时，应调用shmdt函数，作用是将指定的共享内存段从当前进程空间中脱离出去。

返回值　参数addr是调用shmat函数的返回值，函数执行成功返回0，并将该共享内存的shmid_ds结构的shm_nattch计数器减1，失败返回–1。

Example7.6.4-1： shmdt函数使用

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdlib.h>
#include <stdio.h>
int main ( int argc, char *argv[] )
{
 int shm_id ;
 char * shm_buf;
 if ( argc != 2 ){
  /*命令行参数错误*/
  printf ("USAGE: atshm <identifier>");
  /*打印帮助消息*/
  exit (1);
 }
 shm_id = atoi(argv[1]);/*得到要引入的共享内存段*/
 /*引入共享内存段，由内核选择要引入的位置*/
 if ((shm_buf = shmat( shm_id, 0, 0))<(char *) 0 ){
  perror ("shmat");
  exit(1);}
printf("segment attached at %p\n",shm_buf);/*输出导入的位置*/
 system("ipcs -m");
 sleep(3);/*休眠*/
 if((shmdt(shm_buf)) < 0 ){/*与导入的共享内存段分离*/
  perror("shmdt");
  exit(1);}
 printf("segment detached \n");
 system("ipcs -m " );/*再次查看系统IPC状态*/
 exit (0);
}
```

重庆理工大学/电气学院

CHONGQING UNIVERSITY OF TECHNOLOGY

嵌入式Linux系统开发教程

—基于ARM处理器通用平台（arm9-arm11-contexA系列）