实验二　Linux 进程间通信——1 管道通信和消息传递

Linux 系统的进程间通信机构(IPC)允许在任意进程间大批量地交换数据。本实验的目的是了解和熟悉 Linux 支持的信号量机制、管道机制、消息通信机制及共享存储区机制。

一、预备知识

Linux 中进程通信的主要方式包括三种。

1. 管道通信

最简单的 Linux 进程间的通信机制是管道,管道由特殊文件来表示。

在 Linux Shell 中,符号"|"可以创建管道,例如 ls | less。

管道具有的特点:半双工,数据只能在一个方向上流动。只能在具有家族关系的进程中使用。虽然称做文件,但不属于文件系统,只存在于内存中。只能从一端写入,从另一端读出。没有名字管道的缓冲区是有限的,在管道创建时,为其分配一个页面大小。写入管道的数据,读完之后就从管道中消失。

创建管道的函数:#include <unistd.h>

int pipe(int fildes[2]);

该函数创建了一个通信缓冲区,程序可以通过文件描述符 fildes[0]和 fildes[1]来访问这个缓冲区。写入管道的数据可以按先进先出的顺序从管道中读出。

返回值:0,成功;-1,失败并设置 errno。

例如:进程在运行中创建子进程,父进程利用管道向子进程发送信息,子进程从管道读出信息,并显示。

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main ()
{
  int fds[2],len;
  pid_t pid;
  static char ss[]="hello,world!";
  char output[255];
```

```
    /* Create a pipe. File descriptors for the two ends of the pipe are placed in
    fds. */
    pipe (fds);
    /* Fork a child process. */
    pid=fork ();
    if (pid == 0)
    {

        /* This is the child process. Close our copy of the write end of the file descriptor. */
        close (fds[1]);
        len=read (fds[0],output,255);
        close (fds[0]);
        fprintf(stderr,"%s\n",output);

}
else
        {

        /* This is the parent process. */
        /* Close our copy of the read end of the file descriptor. */
        close (fds[0]);
        write(fds[1],ss,sizeof(ss));
        close (fds[1]);
        wait(NULL);
        }
return 0;
}
```

2. 消息队列

消息队列(Message Queue)允许一个或多个进程向它写入或读取消息,从而实现进程之间消息传递。消息队列是消息的链表,存放在内核中并由消息队列标识符标识,该标识符又称为消息队列 ID。

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

系统 V 消息队列是随内核持续的,只有在内核重启或者显式删除一个消息队列时,该消息队列才会真正被删除。因此系统中记录消息队列的数据结构位于内

核中，系统中的所有消息队列都可以在该数据结构中找到访问入口。

消息队列的数据结构定义在 sys/msg.h 中。

```
struct msqid_ds {
  struct ipc_perm   msg_perm;  //读写权限
  struct msg   * msg_first;   //指向队列中第一个消息的指针
  struct msg   * msg_last;    //指向队列中最后一个消息的指针
  msglen_t     msg_cbytes;  //队列中当前的字节数
  msgqnum_t    msg_qnum;   //队列中当前的消息数
  msglen_t     msg_qbytes;   //队列中允许的最大字节数
  pid_t        msg_lspid;    //上一次发送消息的进程 ID
  pid_t        msg_lrpid;    //上一次接收消息的进程 ID
  time_t       msg_stime;   //上一次发送消息的时间
  time_t       msg_rstime;   //上一次接收消息的时间
  time_t       msg_ctime;   //上一次修改队列信息的时间
}
ipc_perm 的结构：
struct ipc_perm   {
  uid_t       uid;   //ower's user id
  gid_t         gid;   //ower's group id
  uid_t       cuid;   //creater's user id
  gid_t         cgid;   //creater's group id
  mode_t       mode;   //read write permissions
  ulong_t     seq;   //slot usage sequence number
  key_t         key;   //IPC   key
}
```

消息队列就是一个消息的链表。每个消息队列都有一个队列头，用结构 struct msqid_ds 来描述。队列头中包含了该消息队列的大量信息，包括消息队列键值、用户 ID、组 ID、消息队列中消息数目，等等，甚至记录了最近对消息队列读写进程的 ID。读者可以访问这些信息，也可以设置其中的某些信息。

对消息队列的操作有下面三种类型。

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

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

(2)读写操作。消息读写操作非常简单,对开发人员来说,每个消息都类似如下的数据结构。

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

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

(3)获得或设置消息队列属性。消息队列的信息基本上都保存在消息队列头中,因此,可以分配一个类似于消息队列头的结构(struct msqid_ds),来返回消息队列的属性;同样可以设置该数据结构。

结构 msg_queue 用来描述消息队列头,存在于系统空间:

```
struct msg_queue {
struct kern_ipc_perm q_perm;
time_t q_stime;             /* last msgsnd time */
time_t q_rtime;             /* last msgrcv time */
time_t q_ctime;             /* last change time */
unsigned long q_cbytes;     /* current number of bytes on queue */
unsigned long q_qnum;       /* number of messages in queue */
unsigned long q_qbytes;     /* max number of bytes on queue */
pid_t q_lspid;              /* pid of last msgsnd */
pid_t q_lrpid;              /* last receive pid */
struct list_head q_messages;
struct list_head q_receivers;
struct list_head q_senders;
}
```

结构 msqid_ds 用来设置或返回消息队列的信息,存在于用户空间;

```
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first;      /* first message on queue,unused  */
    struct msg *msg_last;       /* last message in queue,unused */
    _kernel_time_t msg_stime;  /* last msgsnd time */
```

```
    _kernel_time_t msg_rtime;   /* last msgrcv time */
    _kernel_time_t msg_ctime;   /* last change time */
    unsigned long   msg_lcbytes; /* Reuse junk fields for 32 bit */
    unsigned long   msg_lqbytes; /* ditto */
    unsigned short msg_cbytes;  /* current number of bytes on queue */
    unsigned short msg_qnum;    /* number of messages in queue */
    unsigned short msg_qbytes;  /* max number of bytes on queue */
    _kernel_ipc_pid_t msg_lspid;   /* pid of last msgsnd */
    _kernel_ipc_pid_t msg_lrpid;   /* last receive pid */
}
```

当创建一个消息队列时，需传递给 msgget 函数一个键值，类型是 key_t，通常是至少 32 位的整数，这个键值通常由 ftok 函数生成。或者是在创建消息队列时，传递给 msgget 函数一个特殊的键值 IPC_PRIVATE。

①文件名到键值。

头文件<sys/types.h>中定义了 key_t 数据类型

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok (const char * pathname, int id);
```

函数根据 pathname 和 id 的低 8 位值，生成一个键值。返回与路径 pathname 相对应的一个键值。该函数不直接对消息队列操作，但在 msgget()来获得消息队列描述字前，往往要调用该函数。

②msgget 函数。功能：新建或获取一个已经存在的消息队列的访问。

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int   msgget(key_t key, int oflag);
```

参数 key 是一个键值，由 ftok 获得或者是 IPC_PRIVATE；oflag 参数是一些标志位。该调用返回与健值 key 相对应的消息队列描述字。该描述字将被其它的消息队列函数使用。

参数 oflag 可以为以下：IPC_CREAT、IPC_EXCL。

将参数 oflag 设置为 IPC_CREAT，表示若没有与键值 key 相对应的消息队列存在，则创建队列，若队列已经存在，则返回队列的描述字。

若同时设置 IPC_CREAT 和 IPC_EXCL，表示若没有与键值 key 相对应的消息队列存在，则创建队列，若队列已经存在，则返回一个 EEXIST 错误。

在以下两种情况下，该调用将创建一个新的消息队列：

如果没有消息队列与健值 key 相对应，并且 oflag 中包含了 IPC_CREAT 标志位；

key 参数为 IPC_PRIVATE；调用返回：成功返回消息队列描述字，否则返回 −1。

参数 key 设置成常数 IPC_PRIVATE 并不意味着其他进程不能访问该消息队列，只意味着即将创建新的消息队列。当一个消息队列创建成功后，msqid_ds 结构被初始化。

③发送消息。

消息数据结构：

在<sys/msg.h>中有消息模板数据结构的定义

```
struct msgbuf {
long mtype;    //消息的类型，必须>0
char mtext[1];//消息数据，消息的长度可根据需要定义，不一定是 1
}
```

大多数应用可以根据需要定义自己的消息数据结构，不必局限于模板的定义。

例如：

```
#defime MY_DATA 8
typedef struct my_msgbuf  {
  long mtype;
  int16_t mshort;
  char mchar[MY_DATA];
}Message;
```

发送消息的系统调用：

```
#include <sys/msg.h>
int msgsnd(int msqid, const void * ptr, size_t length, int flag);
```

向 msgid 代表的消息队列发送一个消息，即将发送的消息存储在 ptr 指向的 msgbuf 结构中，消息的大小由 length 指定。对发送消息来说，有意义的 flag 标志为 IPC_NOWAIT，指明在消息队列没有足够空间容纳要发送的消息时，msgsnd 是否等待。造成 msgsnd()等待的条件有两种：(1)当前消息的大小与当前消息队列中的字节数之和超过了消息队列的总容量；(2)系统中的消息数太多。若上述两个条件之一满足，且定义了 flag 标志为 IPC_NOWAIT，调用返回一个错误 EAGAIN，否则，阻塞进程，直到阻塞被解除。

④接收消息。

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *ptr, size_t length, long type, int flag);
```

该系统调用从 msgid 代表的消息队列中读取一个消息，并把消息存储在 ptr 指向的 msgbuf 结构中。

msqid 为消息队列描述字；消息返回后存储在 ptr 指向的地址，length 指定 msgbuf 的数据成员的长度(即消息内容的长度)，type 为请求读取的消息类型，若 type=0，返回队列中的第一个消息；若 > 0，则返回队列中第一个 type 值等于该值的消息，若 < 0，返回最小的 type 值小于或等于该值绝对值的第一个消息。读消息标志 flag，定义了当一个请求消息类型不在队列中时，应该怎样处理。可以为以下几个常量的域：

IPC_NOWAIT 如果没有满足条件的消息，调用立即返回，此时，errno=ENOMSG；否则阻塞进程，直到解除阻塞。

与 msgsnd 解除阻塞的条件类似，msgrcv()解除阻塞的条件也有三个：消息队列中有了满足条件的消息；msqid 代表的消息队列被删除；调用 msgrcv()的进程被信号中断；

MSG_NOERROR 如果队列中满足条件的消息内容大于所请求的 length 字节，则把该消息截断，截断部分将丢失。否则，将返回错误 E2BUG。

调用返回：成功返回读出消息的实际字节数，否则返回－1。

⑤msgctl 函数。该函数提供了对消息队列的各种控制操作。

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buff);
```

该系统调用对由 msqid 标识的消息队列执行 cmd 操作，共有三种 cmd 操作：IPC_STAT、IPC_SET 、IPC_RMID。

IPC_STAT：该命令用来获取消息队列信息，返回的信息存贮在 buff 指向的 msqid_ds 结构中；

IPC_SET：该命令用来设置消息队列的属性，要设置的属性存储在 buff 指向的 msqid 结构中；可设置属性包括：msg_perm. uid、msg_perm. gid、msg_perm. mode 以及 msg_qbytes，同时，也影响 msg_ctime 成员。

IPC_RMID：删除 msqid 标识的消息队列；对于这个命令选项，第三个参数忽略(NULL)。

调用返回：成功返回 0，否则返回－1。

二、实验目的

1. 熟悉有关 Linux 中进程通信的系统调用。
2. 学习有关 Linux 的进程创建，理解进程创建后两个并发进程的执行。
3. 掌握管道、消息缓冲等进程通信方法并了解其特点和使用限制。

三、实验内容

1. Linux 消息缓冲通信

编写一个程序，实现以下功能。给出源程序代码和运行结果。父进程创建一个消息队列和一个子进程，由子进程发送消息，父进程等待子进程结束后接收子进程发来的消息，并显示消息内容。以“end”作为结束消息。

程序代码：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/types.h>
#include<linux/msg.h>
#define   MAXMSG   512                    //定义消息长度
struct   my_msg                          //定义消息缓冲区数据结构
{
  long int my_msg_type;
  char some_text[MAXMSG];
}msg;
main()
{
  int p;
  int   msgid;                           //定义消息缓冲区内部标识
  char buffer[BUFSIZ];                   //定义用户缓冲区
  msgid=msgget(1234,0666|IPC_CREAT);     //创建消息队列,key 为 1234
  long   int    msg_to_receive=0;
  while((p=fork())==-1);
  if(p==0){
      while(1)
      {
          puts("Enter some text:");      //提示键入消息内容
```

```
            fgets(buffer,BUFSIZ,stdin);              //标准输入送 buffer
            msg.my_msg_type=1;                       //设置消息类型为 1
            strcpy(msg.some_text,buffer);            //buffer 送消息缓冲
            msgsnd (msgid,&msg,MAXMSG,0);            //发送消息到消息队列
            if(strncmp(msg.some_text,"end",3)==0)     //消息为"end"则结束
                break;
        }
        exit(0);
    }
    else{
        wait(0);
        while (1)
        {
            msgrcv (msgid, &msg, BUFSIZ, msg_to_receive, 0);      //接收消息
            printf ("You wrote:%s",msg.some_text);                //显示消息
            if (strncmp (msg.some_text,"end",3)==0)     //消息为"end"则结束
                break;
        }
        msgctl (msgid, IPC_RMID,0);                   //撤消消息队列

    }
}
```

2. 编写进程

分别编写发送进程和接收进程，由发送进程发送消息，接收进程接收消息。采用先执行发送进程后执行接收进程的方式同步。以"end"作为结束消息。

发送进程：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/types.h>
#include<linux/msg.h>
#define  MAXMSG  512                    //定义消息长度
struct  my_msg                          //定义消息缓冲区数据结构
{
    long int my_msg_type;
```

```
    char some_text[MAXMSG];
};msg;
main(     )
{
    int   msgid;                                    //定义消息缓冲区内部标识
    char buffer[BUFSIZ];                            //定义用户缓冲区
    msgid=msgget(1234,0666|IPC_CREAT);              //创建消息队列,key 为 1234
    while(1)
    {
        puts("Enter some text:");                   //提示键入消息内容
        fgets(buffer,BUFSIZ,stdin);                 //标准输入送 buffer
        msg.my_msg_type=1;                          //设置消息类型为 1
        strcpy(msg.some_text,buffer);               //buffer 送消息缓冲
        msgsnd (msgid,&msg,MAXMSG,0);               //发送消息到消息队列
        if(strncmp(msg.some_text,"end",3)==0)       //消息为"end"则结束
        break;
  }
  exit(0);
}
```

接收进程：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<sys/types.h>
#include<linux/msg.h>
#define   MAXMSG   512          //定义消息长度
struct   my_msg                 //定义消息 f 缓冲区数据结构
{
  long int my_msg_type;
  char some_text[MAXMSG];
};msg;
main()
{
  int   msgid;                  //定义消息缓冲区内部标识
  long   int    msg_to_receive=0;
  msgid=msgget(1234, 0666|IPC_CREAT);       //获取消息队列,key 为 1234
```

```
    while (1)
    {
        msgrcv (msgid, &msg, BUFSIZ, msg_to_receive, 0); //接收消息
        printf ("You wrote:%s",msg.some_text);          //显示消息
        if (strncmp (msg.some_text,"end",3)==0)         //消息为"end"则结束
            break;
    }
    msgctl (msgid, IPC_RMID,0);                         //撤消消息队列
    exit (0);
}
```

四、上机思考题

编写一段程序，使用系统调用 fork()创建两个子进程，再用系统调用 signal()让父进程捕捉键盘上的中断信号(即按 ctrl+c 键)，当捕捉到中断信号后，父进程用系统调用 kill()向两个子进程发出信号，子进程捕捉到信号后，分别输出下列信息后终止：

```
Child process 1 is killed by parent!
Child process 2 is killed by parent!
```

父进程等待两个子进程终止后，输出以下信息后终止：

```
Parent process is killed!
```

实验三　Linux 进程间通信——2 共享内存

一、预备知识

Linux 进程间通信的另一种方式是采用共享内存的方式，试调试运行如下关于共享内存的程序代码。程序执行内容依次如下：

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了数据，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(如图 2-4 所示)。这块共享虚拟内存的页面，出现在每一个共享