电子科技大学信息与软件工程学院

**实 验 报 告**

学 号 2018091602007

姓 名 王乐卿

（实验） 课程名称 操作系统基础

理论教师 刘瑶

实验教师 刘瑶

**电子科技大学教务处制表**

**电 子 科 技 大 学**

**实 验 报 告**

**学生姓名：王乐卿 学号：2018091602007 指导教师：刘瑶**

**实验地点：家 实验时间：2020.06.09**

**一、实验名称：进程间通信问题的实现**

**二、实验学时：4**

**三、实验目的：**

熟悉Linux下的应用程序开发

熟悉Linux的进程控制原语的使用

掌握Linux操作系统的进程间通信机制管道的使用。

掌握Linux操作系统中父进程与子进程的同步。

通过对进程间通信（消息队列）的设计，深入理解进程之间是如何通过消息队列进行通信的。

**四、实验原理：**

（一）管道

所谓“管道”，是指用于连接一个读进程和一个写进程以实现他们之间通信的一个共享文件，又名pipe文件。

向管道(共享文件)提供输入的发送进程(即写进程)， 以字符流形式将大量的数据送入管道；

接受管道输出的接收进程(即读进程)，则从管道中接收(读)数据。由于发送进程和接收进程是利用管道进行通信的，故又称为管道通信。

这种方式首创于UNIX系统，由于它能有效地传送大量数据，因而又被引入到许多其它操作系统中。

（二）消息队列

消息队列就是一个消息的链表。可以把消息看作一个记录，具有特定的格式以及特定的优先级。对消息队列有写权限的进程可以向消息队列按照一定的规则添加新消息；对消息队列有读权限的进程则可以从消息队列中读走消息。

**五、实验内容：**

（一）管道：

在Linux系统中使用系统调用fork()创建两个子进程，使用系统调用pipe()建立一个管道，两个子进程分别向管道各写一句话：

Child process 1 is sending a message!

Child process 2 is sending a message!

而父进程则从管道中读出来自于两个子进程的信息，显示在屏幕上。然后分别结束两个子进程的运行。

（二）消息队列：

编程实现两个进程通过消息队列进行通信，一个Server进程，一个client进程。

Server进程向client进程发送其进程ID，Client进程同时也向Server进程发送其进程ID，双方接收到消息后，将所接收到的进程ID输出到屏幕上。

**六、实验器材（设备、元器件）：**

1. 学生每人一台PC，安装Windows8操作系统。

2. 个人PC安装VMware虚拟机和Ubuntu系统。

**七、实验步骤：**

（一）实验3 利用管道实现两个进程的通信

1）算法思想

首先使用fork()函数创建两个子进程，

使用waitpid()函数实现父进程等待子进程运行完毕后从管道中读取数据并打印，只有子进程将数据写入管道后，父进程才能够执行打开管道操作，以此实现父进程和两个子进程同步，

由于fork函数让子进程完整地拷贝了父进程的整个地址空间，所以子进程都有管道的读端和写端。所以在相关进程中使用close(pipe\_fd[N])关掉不用的那一端。

2）流程图

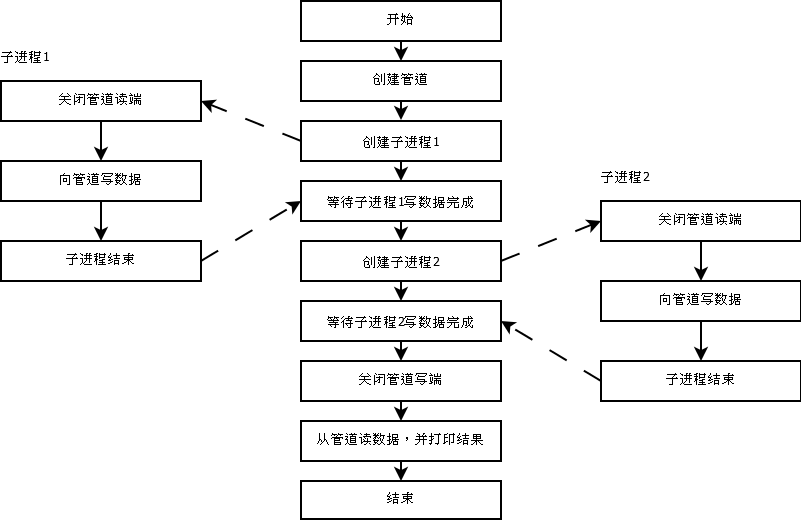


图 1管道实现通信流程图

3）实验代码

|  |
| --- |
| #include <unistd.h>  #include <sys/types.h>  #include <sys/wait.h>  #include <errno.h>  #include <string.h>  #include <stdio.h>  #include <stdlib.h>  int main( )  {  pid\_t pid1,pid2;  int pipe\_fd[2];  char \*msg1="child 1 process is sending message!";  char \*msg2="child 2 process is sending message!";  char buf[100];  /\*创建一个管道\*/  if(pipe(pipe\_fd)<0)  {  printf("pipe create error\n");  return -1;  }    /\*创建子进程\*/  if((pid1=fork( ))==0) // 子进程1执行序列  {  close(pipe\_fd[0]); // 子进程先关闭管道的读段  write(pipe\_fd[1],msg1,strlen(msg1)); //向管道写长为strlen(msg1)字节的串  exit(0);  }  else if(pid1>0) // 父进程执行序列  {  waitpid(pid1,NULL,0); // 同步1--等子进程1先写完  if((pid2=fork( ))==0) // 子进程2执行序列  {  close(pipe\_fd[0]); // 子进程先关闭管道的读段  write(pipe\_fd[1],msg2,strlen(msg2)); //向管道写长为strlen(msg2)字节的串  exit(0);  }  else if(pid2>0) // 父进程执行序列  {  waitpid(pid2,NULL,0); // 同步2--再等子进程2写完  close(pipe\_fd[1]); // 父进程先关闭管道的写段  if(read(pipe\_fd[0],buf,strlen(msg1))>0) //读出msg1  printf("%s\n",buf);  if(read(pipe\_fd[0],buf,strlen(msg2))>0) //读出msg2  printf("%s\n",buf);  exit(0);  }  }  return 0;  } |

代码 1管道实现通信代码

（二）实验4 利用消息队列实现进程间的通信

1）算法思想

首先打开键值所对应的消息队列，使用getpid()获取pid，然后使用msgsnd()将pid送入消息队列中，使用msgrcv()获取消息。

2）流程图

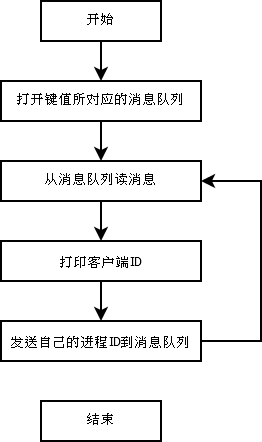


图 2server流程图

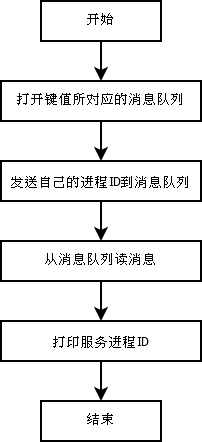


图 3Client流程图

3）实验代码

|  |
| --- |
| #include <unistd.h>  #include <stdlib.h>  #include <stdio.h>  #include <string.h>  #include <sys/msg.h>  #include <errno.h>  #define MAX\_TEXT 512  struct msg\_st  {  long int msg\_type;  char text[MAX\_TEXT];  };  int main()  {  struct msg\_st data\_accept;//创建一个消息data  struct msg\_st data\_send;//创建一个消息data  char buffer[BUFSIZ];//BUFSIZ代表了默认的缓冲大小  long int msgtype = 0; //返回消息队列中的第一个消息  int msgid = -1;  int pid;  char pid\_str[10]={0};  //建立消息队列  msgid = msgget((key\_t)1234, 0666 | IPC\_CREAT);//创建一个消息队列，读写权限是666（0代表是后面是8进制数）  if(msgid == -1)  {  fprintf(stderr, "msgget failed with error: %d\n", errno);  exit(EXIT\_FAILURE);  }  while(1)  {    //从队列中获取数据  if(msgrcv(msgid, (void\*)&data\_accept, BUFSIZ, msgtype, 0) == -1)  {  fprintf(stderr, "msgrcv failed with errno: %d\n", errno);  exit(EXIT\_FAILURE);  }  printf("从队列中获取的Client的pid是: %s\n",data\_accept.text);    //向消息队列中写消息，直到写入end  pid=getpid(); //获取pid  data\_send.msg\_type = 1; //将消息data的类型设为1  sprintf(pid\_str,"%d",pid); //将数值型的pid转换为char\*型的pid\_str  //strcat(buffer,pid\_str); //将pid\_str加入缓冲的末尾  strcpy(data\_send.text, pid\_str); //把缓冲池中的内容复制给data.text  printf("向队列中发送的Server的pid是：%s\n",data\_send.text);    //向队列发送数据  if(msgsnd(msgid, (void\*)&data\_send, MAX\_TEXT, 0) == -1)  {  fprintf(stderr, "msgsnd failed\n");  exit(EXIT\_FAILURE);  }  }  //删除消息队列  if(msgctl(msgid, IPC\_RMID, 0) == -1)  {  fprintf(stderr, "msgctl(IPC\_RMID) failed\n");  exit(EXIT\_FAILURE);  }    exit(EXIT\_SUCCESS);  } |

代码 2 Server代码

|  |
| --- |
| #include <unistd.h>  #include <stdlib.h>  #include <stdio.h>  #include <string.h>  #include <sys/msg.h>  #include <errno.h>  #define MAX\_TEXT 512  struct msg\_st  {  long int msg\_type;  char text[MAX\_TEXT];  };  int main()  {  struct msg\_st data\_accept;//创建一个消息data  struct msg\_st data\_send;//创建一个消息data  char buffer[BUFSIZ];//BUFSIZ代表了默认的缓冲大小  long int msgtype = 0; //返回消息队列中的第一个消息  int msgid = -1;  int pid;  char pid\_str[10]={0};  //建立消息队列  msgid = msgget((key\_t)1234, 0666 | IPC\_CREAT);//创建一个消息队列，读写权限是666（0代表是后面是8进制数）  if(msgid == -1)  {  fprintf(stderr, "msgget failed with error: %d\n", errno);  exit(EXIT\_FAILURE);  }  while(1)  {  //向消息队列中写消息，直到写入end  pid=getpid(); //获取pid  data\_send.msg\_type = 1; //将消息data的类型设为1  sprintf(pid\_str,"%d",pid); //将数值型的pid转换为char\*型的pid\_str  //strcat(buffer,pid\_str); //将pid\_str加入缓冲的末尾  strcpy(data\_send.text, pid\_str); //把缓冲池中的内容复制给data.text  printf("向队列中发送的Client的pid是：%s\n",data\_send.text);    //向队列发送数据  if(msgsnd(msgid, (void\*)&data\_send, MAX\_TEXT, 0) == -1)  {  fprintf(stderr, "msgsnd failed\n");  exit(EXIT\_FAILURE);  }    //从队列中获取数据  if(msgrcv(msgid, (void\*)&data\_accept, BUFSIZ, msgtype, 0) == -1)  {  fprintf(stderr, "msgrcv failed with errno: %d\n", errno);  exit(EXIT\_FAILURE);  }  printf("从队列中获取的Server的pid是: %s\n",data\_accept.text);  }  //删除消息队列  if(msgctl(msgid, IPC\_RMID, 0) == -1)  {  fprintf(stderr, "msgctl(IPC\_RMID) failed\n");  exit(EXIT\_FAILURE);  }    exit(EXIT\_SUCCESS);  } |

代码 3Client代码

**八、实验结果与分析（含重要数据结果分析或核心代码流程分析）**

（一）实验3运行结果

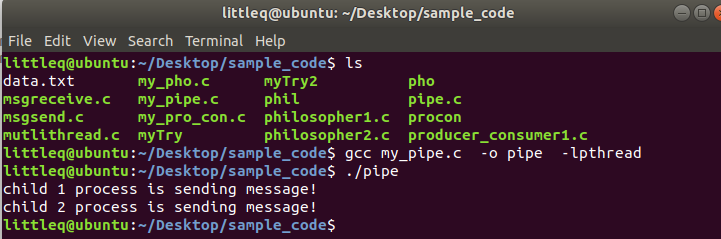


图 4管道实现通信运行结果

（二）实验4运行结果

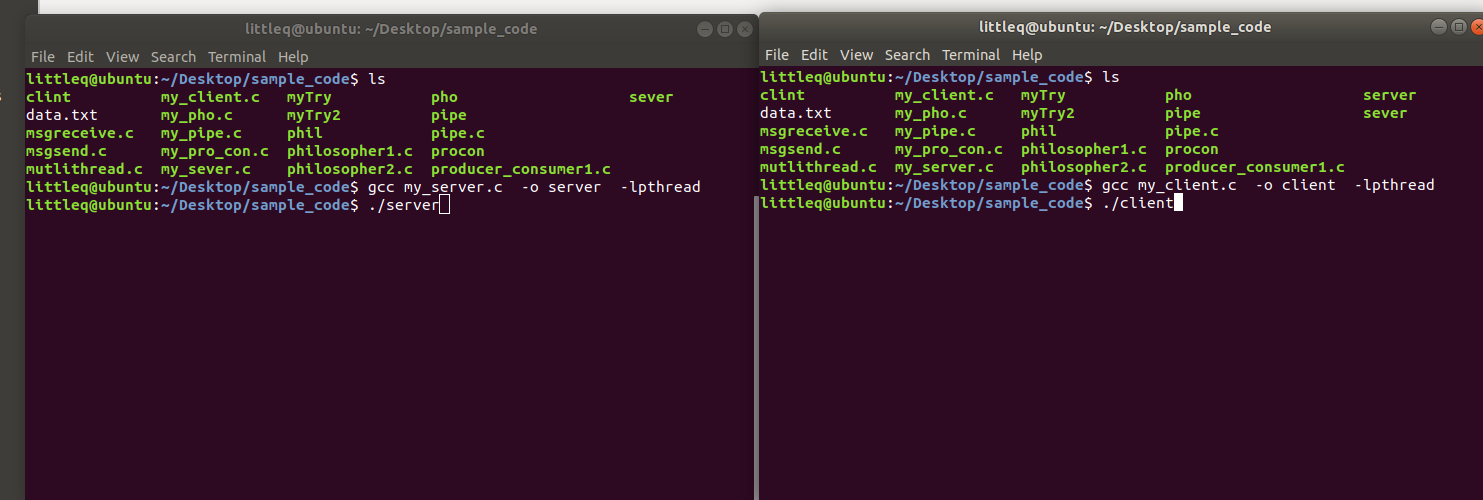


图 5运行命令

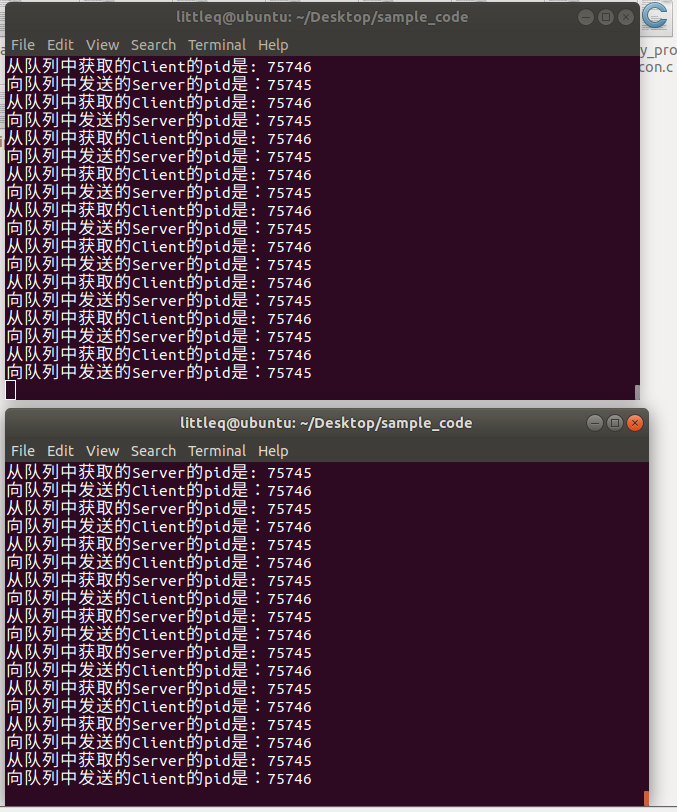


图 6 消息队列通信运行结果

**九、总结及心得体会：**

通过本次实验，我初步掌握了Linux操作系统中父进程与子进程的同步，同时通过对进程间通信（消息队列）的设计，更加深入的理解了进程之间是如何通过消息队列进行通信的。

同时经过本次实验，我更加深刻的认识到了学以致用的重要性，理论的学习记忆很容易忘却，而真正的实操之后，记忆会更加深刻，同时理解也能更加深入。

**十、对本实验过程及方法、手段的改进建议：**

建议加入一些linux的前导实验，以衔接更为平滑。

**报告评分：**

**指导教师签字：**