

浙大城市学院实验报告

- 课程名称：操作系统原理实验
- 实验项目名称：实验九 进程通信——消息队列
- 学生姓名：徐彬涵
- 专业班级：软件工程2003
- 学号：32001272
- 实验成绩：
- 指导老师：胡隽
- 日期：2022/05/04

实验目的

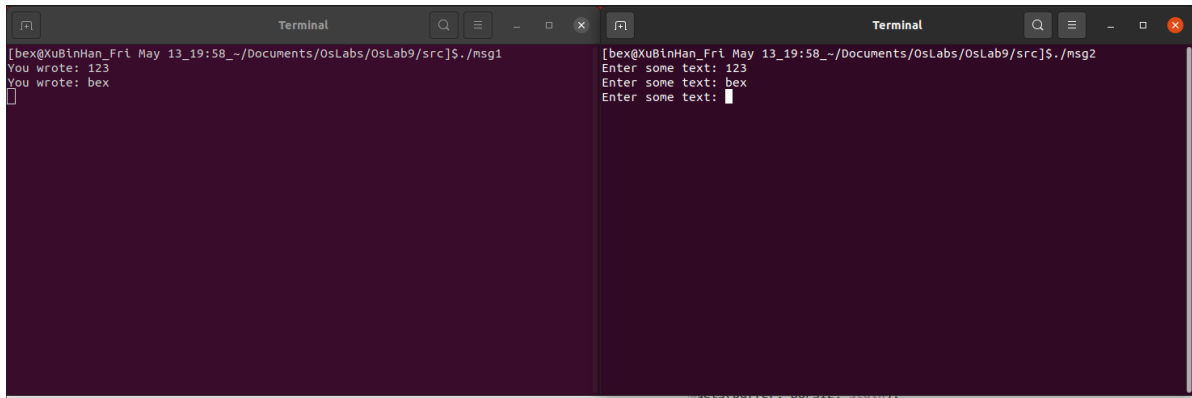
1. 了解Linux系统的进程间通信机构(IPC)
2. 理解Linux关于消息队列的概念
3. 掌握Linux支持消息队列的系统调用
4. 巩固进程同步概念

实验步骤

消息队列保存在内核中，是一个由消息结构组成的链表，每个消息队列（链表）有消息队列标识符，消息队列允许一个或多个进程写入或读取消息，因此消息队列提供了一种从一个进程向另一个进程发送一个数据块的方法，并可实现双向通信，消息队列解决了读写进程的同步和阻塞问题

msg1.c \ msg2.c

编译链接通过后，在两个终端分别运行发送进程和接收进程，观察进程并发执行结果，并思考下述问题：



1. 熟悉和消息队列相关的系统调用

```
1 //创建消息队列
2 int msgget(
3     key_t key, //消息对列的键值
4     int msgflag //消息队列的建立标志和存取权限
5 );
6 //例
7 msgget((key_t)1234, 0666 | IPC_CREAT);
8 /*会返回键值为1234的消息队列的消息队列标识符，如果消息队列已存在则返回
9 已存在消息队列的标识符
10 0666指的是消息队列权限
11 从左向右:
12 第一位:表示这是个八进制数 000
13 第二位:当前用户的经权限:6=110(二进制),每一位分别对就 可读,可写,可执行,,6说明
14 当前用户可读可写不可执行
15 第三位:group组用户,6的意义同上
16 第四位:其它用户,每一位的意义同上
17 这里表示所有用户都对这个消息队列有读写执行权限*/
```

```
1 //向消息队列中发送消息
2 int msgsnd(
3     int msqid, //消息队列的标识符
4     const void *msgp,
5     /*指向消息缓冲区的指针,此位置用来暂时存储发送和接收的消息，是一个用户
6 可定义的通用结果，形态如下
7     struct msgbuf{
8         long mtype; //消息类型，必须>0
9         char mtext[size]; //消息文本
10     }*/
11     size_t msgsz,
12     int msgflg
13 );
14 //返回值: 成功: 0、识别: -1
```

```

1 //从消息队列中接收消息
2 ssize_t msgrcv(
3     int msqid, //消息队列的标识符
4     void *msgp, //指向消息缓冲区的指针
5     size_t msgsz, //消息正文的字节数, 不包括消息类型指针变量
6     long msgtyp,
7     /*值为0表示接收消息队列的第一个消息
8     值大于0表示接收消息队列中第一个类型为msgtyp的消息
9     值小于0表示接收消息队列中第一个类型值不小于msgtyp绝对值且最小的*/
10    int msgflg
11    /*值为MSG_NOERROR表示是返回字节比msgsz多, 消息截短
12    值为IPC_NOWAIT, 在队列空时, msgsnd()不会阻塞, 立即返回-1
13    值为0, 在队列空时, 采取阻塞等待的处理方式*/
14 );
15 //返回值: 成功: 0、识别: -1

```

```

1 //在消息队列上执行指定的操作
2 int msgctl(
3     int msqid, //消息队列的标识符
4     int cmd,
5     /*值为IPC_STAT, 读取消息队列的数据结构msqid_ds, 并将其存储在buf指定
6     的地址中
7     值为IPC_SET, 设置消息队列的数据结构msqid_ds中的ipc_perm域值, 值取自
8     buf参数
9     值为IPC_RMID, 从系统内核中删除消息队列*/
10    struct msqid_ds *buf //消息队列的msqid_ds结构类型变量
11 );
12 //返回值: 成功: 0、识别: -1

```

2. 尝试运行多个发送进程和多个接收进程，观察进程的并发执行，并解释原因

The image shows four terminal windows arranged in a 2x2 grid, illustrating the interleaved output of two concurrent processes, `msg1` and `msg2`. The top-left window shows `msg2` repeatedly printing "Enter some text: s". The top-right window shows `msg1` printing "You wrote: 123", "You wrote: bex", "You wrote: 123", "You wrote: 222", and then several "You wrote: s" messages. The bottom-left window shows `msg2` printing "Enter some text: 123", "Enter some text: bex", and "Enter some text: end". The bottom-right window shows `msg1` printing "You wrote: 345", "You wrote: bex", "You wrote: ss", and then several "You wrote: s" messages, followed by "You wrote: end". The interleaving of outputs demonstrates that the processes are running concurrently and sharing resources.

两个输出进程交替输出内容，原因在于进程并发，相互争夺资源，当一个接收进程接收后另一个接收进程就不会再接收，导致了两进程的交替输出

msgthread.c

编译链接通过后，运行程序，通过系统提示，观察多线程的并发执行，并思考下述问题：

1. 熟悉和消息队列相关的系统调用

```
1 //创建消息队列
2 int msgget(
3     key_t key, //消息对列的键值
4     int msgflag //消息队列的建立标志和存取权限
5 );
6 //例
7 msgget((key_t)1234, 0666 | IPC_CREAT);
8 /*会返回键值为1234的消息队列的消息队列标识符，如果消息队列已存在则返回
9 已存在消息队列的标识符
10 0666指的是消息队列权限
11 从左向右:
12 第一位:表示这是个八进制数 000
13 第二位:当前用户的经权限:6=110(二进制),每一位分别对就 可读,可写,可执行,,6说
14 明当前用户可读可写不可执行
15 第三位:group组用户,6的意义同上
16 第四位:其它用户,每一位的意义同上
17 这里表示所有用户都对这个消息队列有读写执行权限*/
```

```
1 //向消息队列中发送消息
2 int msgsnd(
3     int msqid, //消息队列的标识符
4     const void *msgp,
5     /*指向消息缓冲区的指针,此位置用来暂时存储发送和接收的消息, 是一个用户
    可定义的通用结果, 形态如下
6     struct msgbyf{
7         long mtype; //消息类型, 必须>0
8         char mtext[size]; //消息文本
9     }*/
10    size_t msgsz,
11    int msgflg
12 );
13 //返回值: 成功: 0、识别: -1
```

```
1 //从消息队列中接收消息
2 ssize_t msgrcv(
3     int msqid, //消息队列的标识符
4     void *msgp, //指向消息缓冲区的指针
5     size_t msgsz, //消息正文的字节数, 不包括消息类型指针变量
6     long msgtyp,
7     /*值为0表示接收消息队列的第一个消息
8     值大于0表示接收消息队列中第一个类型为msgtyp的消息
9     值小于0表示接收消息队列中第一个类型值不小于msgtyp绝对值且最小的*/
10    int msgflg
11    /*值为MSG_NOERROR表示是返回字节比msgsz多, 消息截短
12    值为IPC_NOWAIT, 在队列空时, msgsnd()不会阻塞, 立即返回-1
13    值为0, 在队列空时, 采取阻塞等待的处理方式*/
14 );
15 //返回值: 成功: 0、识别: -1
```

```

1 //在消息队列上执行指定的操作
2 int msgctl(
3     int msgid, //消息队列的标识符
4     int cmd,
5     /*值为IPC_STAT, 读取消息队列的数据结构msgqid_ds, 并将其存储在buf指定
    的地址中
6     值为IPC_SET, 设置消息队列的数据结构msgqid_ds中的ipc_perm域值, 值取自
    buf参数
7     值为IPC_RMID, 从系统内核中删除消息队列*/
8     struct msgqid_ds *buf //消息队列的msgqid_ds结构类型变量
9 );
10 //返回值: 成功: 0、识别: -1

```

2. 回顾于POSIX线程控制的信号量相关的函数

信号量初始化: `sem_init(sem_t *sem, int pshared, unsigned int value);`

参数说明:

`sem_t *sem`: 信号量变量

`int pshared`: 指明信号量的类型。不为0时此信号量在相关进程间共享, 否则只 能为当前进程的所以线程共享

`unsigned int value`: 该参数指定信号量的初始值

返回值: 成功时返回0; 错误时, 返回-1, 并将`errno`设置为合适的值

P操作 (信号量的-1操作): `int sem_wait(sem_t *sem);`

参数说明:

`sem_t *sem`: 信号量变量

函数说明:

等待信号量, 如果信号量的值大于0, 将信号量的值减1, 立即返回。如果信号量的值为0, 则线程阻塞。

返回值: 成功返回0, 失败返回-1。

V操作 (信号量的+1操作): `int sem_post(sem_t *sem);`

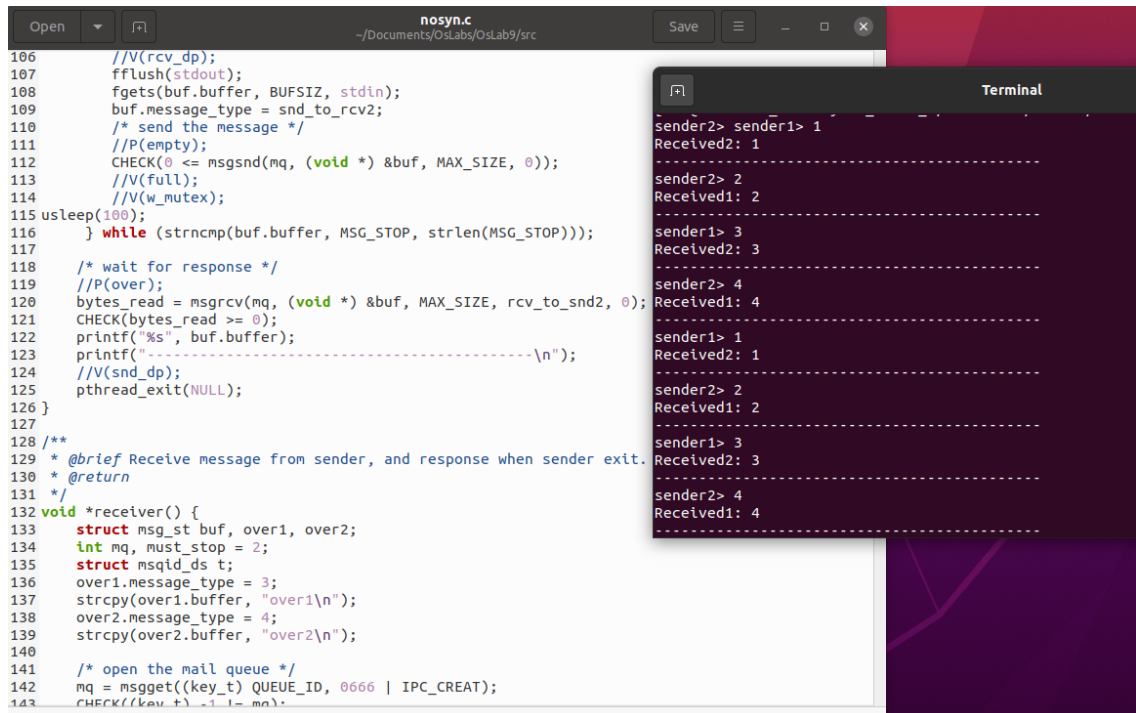
参数说明:

`sem_t *sem`: 信号量变量

函数说明:

释放信号量, 让信号量的值加1。若此时有`sem_wait`正在阻塞则唤醒。

3. 尝试删除信号量的同步控制, 观察并发线程运行的混乱状况



```
106 //V(rcv_dp);
107 fflush(stdout);
108 fgets(buf.buffer, BUFSIZ, stdin);
109 buf.message_type = snd_to_rcv2;
110 /* send the message */
111 //P(empty);
112 CHECK(0 <= msgsnd(mq, (void *) &buf, MAX_SIZE, 0));
113 //V(full);
114 //V(w_mutex);
115 usleep(100);
116 } while (strcmp(buf.buffer, MSG_STOP, strlen(MSG_STOP)));
117
118 /* wait for response */
119 //P(over);
120 bytes_read = msgrcv(mq, (void *) &buf, MAX_SIZE, rcv_to_snd2, 0);
121 CHECK(bytes_read >= 0);
122 printf("%s", buf.buffer);
123 printf("-----\n");
124 //V(snd_dp);
125 pthread_exit(NULL);
126 }
127
128 /**
129  * @brief Receive message from sender, and response when sender exit.
130  * @return
131  */
132 void *receiver() {
133     struct msg_st buf, over1, over2;
134     int mq, must_stop = 2;
135     struct msqid_ds t;
136     over1.message_type = 3;
137     strcpy(over1.buffer, "over1\n");
138     over2.message_type = 4;
139     strcpy(over2.buffer, "over2\n");
140
141     /* open the mail queue */
142     mq = msgget((key_t) QUEUE_ID, 0666 | IPC_CREAT);
143     CHECK((key_t) -1 != mq);
```

```
sender2> sender1> 1
Received2: 1
-----
sender2> 2
Received1: 2
-----
sender1> 3
Received2: 3
-----
sender2> 4
Received1: 4
-----
sender1> 1
Received2: 1
-----
sender2> 2
Received1: 2
-----
sender1> 3
Received2: 3
-----
sender2> 4
Received1: 4
-----
```

4. 理清例程中并发线程同步和互斥关系

`w_mutex` 信号用于实现互斥, 控制了两个sender线程不同时进行, `snd_dp` 信号量用于实现同步, 保证在一个sender线程向消息队列输入内容后reader线程必须输出内容之后下一个sender线程才可以再次输入内容, `full` 信号量表示消息队列中的信息数量, `empty` 信号量表示我们自己设定的消息队列中还可放入的信息数量, `over` 信号量用于控制sender线程的关闭

编程题

实现双向通话

dialog1.c

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/msg.h>
4 #include <sys/ipc.h>
5 #include <sys/wait.h>
6 #include <stdlib.h>
```

```

7  #include <string.h>
8  #include <unistd.h>
9  #include <errno.h>
10 #include<signal.h>
11
12 #define MSGKEY1 66
13 #define MSGKEY2 68
14
15
16 struct msgform
17 {
18     long mtype;
19     char mtext[1000];
20 }msg;
21
22 int msgqid;
23
24 void server( ) {
25     msgqid=msgget(MSGKEY1,0777|IPC_CREAT);
26     while (1){
27         sleep(1);
28         msgrcv(msgqid,&msg,1024,0,0);
29         if (msg.mtype == 2) {
30             printf("(server)received:%s\n", msg.mtext);
31             strcpy(msg.mtext, "");
32             msg.mtype = 1;
33             msgsnd(msgqid, &msg, 1024, 0);
34         }else
35             continue;
36     }
37 }
38 void client() {
39     /*打开消息队列*/
40     msgqid=msgget(MSGKEY2,0777|IPC_CREAT);
41     while (1) {
42         sleep(1);
43         scanf("%s", msg.mtext);
44         msg.mtype = 2;
45         msgsnd(msgqid, &msg, 1024, 0);
46         if (strncmp(msg.mtext, "bye", 3) == 0) break;
47     }
48 }

```



```

49  int main( )
50  {
51      int pid;
52      if(fork()!=0)
53          server();
54      else
55          client();
56      kill(pid,2);
57      wait(0);
58  }

```

dialog2.c

```

1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/msg.h>
4  #include <sys/ipc.h>
5  #include <sys/wait.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <unistd.h>
9  #include <errno.h>
10 #include<signal.h>
11
12 #define MSGKEY1 66
13 #define MSGKEY2 68
14
15
16 struct msgform
17 {
18     long mtype;
19     char mtext[1000];
20 }msg;
21
22 int msgqid;
23
24 void server( ) {
25     msgqid=msgget(MSGKEY2,0777|IPC_CREAT);
26     while (1){
27         sleep(1);
28         msgrcv(msgqid,&msg,1024,0,0);

```

```

29     if (msg.mtype == 2) {
30         printf("(server)received:%s\n", msg.mtext);
31         strcpy(msg.mtext, "");
32         msg.mtype = 1;
33         msgsnd(msgqid, &msg, 1024, 0);
34     }else
35         continue;
36 }
37 }
38 void client() {
39     /*打开消息队列*/
40     msgqid=msgget(MSGKEY1,0777|IPC_CREAT);
41     while (1) {
42         sleep(1);
43         scanf("%s", msg.mtext);
44         msg.mtype = 2;
45         msgsnd(msgqid, &msg, 1024, 0);
46         if (strncmp(msg.mtext, "bye", 3) == 0) break;
47     }
48 }
49 int main( )
50 {
51     int pid;
52     if(pid = fork() != 0)
53         server();
54     else
55         client();
56     kill(pid,2);
57     wait(0);
58 }

```

The image shows two terminal windows side-by-side, both titled 'Terminal'. The left terminal shows the server process output, and the right terminal shows the client process output.

Left Terminal (Server Process):

```

[bex@XuBinHan_Fri May 13_22:52_~/Documents/OsLabs/OsLab9/src]$ ./d1
hello
(server)received:hi
1
(server)received:2
(server)received:2
bex
(server)received:xbh
bye
[bex@XuBinHan_Fri May 13_22:53_~/Documents/OsLabs/OsLab9/src]$

```

Right Terminal (Client Process):

```

[bex@XuBinHan_Fri May 13_22:52_~/Documents/OsLabs/OsLab9/src]$ ./d2
(server)received:hello
hi
(server)received:1
(server)received:1
2
2
(server)received:bex
xbh
(server)received:bye
bye
[bex@XuBinHan_Fri May 13_22:53_~/Documents/OsLabs/OsLab9/src]$

```

