01 实验概述

- 实验目的：了解和熟悉linux支持的消息通信机制。
- 实验任务：使用linux系统提供的系统调用 `msgget()`, `msgrev()`, `msgctl()` 编制一个长度为1K的消息发送和接受的程序。

02 实验过程

2.1 设计消息结构体

根据题意，消息长度为1KB，其中必要信息需包含发送者PID、消息编号、消息类型；
设计如下结构体，其中前3个变量为上述3个必要信息，最后用char填充空余空间。

```
struct msgform
{
    long mtype;       // 消息类型
    int mfrom;        // 消息发送者PID
    int mindex;       // 消息编号
    char mtext[1016]; // 消息文本
    // 其中 mtype 为消息队列中消息结构的固定内容，不计入消息大小
    // mform、mindex、mtext 大小共1024B，符合实验1K大小的消息要求
} msg;
```

消息大小的计算：
参考消息发送函数 `int msgsnd ( int msqid, struct msgbuf *msgp, int msgsz, int msgflg )` 中的消息长度参数 `msgsz`，其计算方式为

$$msgsz = sizeof(struct\ msgform) - sizeof(long);$$

即 `long mtype` 不计入消息长度，因而填充文本大小为

$$1024 - 4 - 4 = 1016B$$

2.2 消息队列约定

服务器和客户端程序做出如下约定：

- 使用 `KEY=75` 的消息队列进行通信；
- 消息队列由Server创建，Client仅是获得此消息队列id（而不是创建）；
- Server接收消息类型 `mtype=1` 的消息（因此两个Client发送消息时应将消息类型设置为1）；
- Client接收消息类型 `mtype=pid` 的消息（即接收消息类型与Client进程 `pid` 相同的消息，因而Server发送消息时应设置 `mtype` 为目标Client进程的 `pid` ）；

综上，在客户端程序 `msg_client.c` 与服务器程序 `msg_server.c` 中定义如下常量：

```
#define MSGKEY 75  // 用于消息队列KEY=75，双方使用此消息队列通信
#define SERVER 1   // server接收类型为1的消息
```

2.3 实验程序

a. 服务器程序 `msg_server.c`

要求：*由 `SERVER` 端创建一个 `Key` 为75的消息队列，等待 `CLIENT1` 端进程和 `CLIENT2` 端进程发来的消息。当收到 `CLIENT1` 端与 `CLIENT2` 端消息编号为1的消息后，作为结束信号，删除该队列，并结束SERVER。*

分析：

- 为使得消息队列由Server创建，这里使用设置 `msgget` 参数为 `0777|IPC_CREAT|IPC_EXCL` ；
- 对于Client端，将设置 `msgget` 参数为 `0777` ，即仅可请求已经存在的消息队列，而不能创建。

`msg_server.c` 主要代码如下：

```
int main()
```

```
{
    int i, pid;
    // 软中断处理
    for (i = 0; i < 20; i++)
        signal(i, cleanup);
    // 获得当前server进程pid
    pid = getpid();
    // 重复尝试创建消息队列（若成功，必定是有server创建此消息队列）
    while ((msgqid=msgget(MSGKEY,0777|IPC_CREAT|IPC_EXCL))==-1)
        ;
    printf("(SERVER  %d) created message queue with key %d\n", pid, MSGKEY);

    // cnt1 为server收到的消息编号为1的数量，若累计到2个，则结束服务器进程
    int cnt1 = 0;
    while (cnt1 < 2)
    {
        // 服务器接收类型为SERVER=1的消息
        msgrcv(msgqid, &msg, sizeof(msg) - sizeof(long), 1, 0);
        printf("(SERVER  %d) received message %d from CLIENT %d\n",
                                                pid, msg.mindex, msg.mfrom);
        // 检查收到的消息编号是否为1，如果是，则累加到cnt1
        if (msg.mindex == 1)
            cnt1++;
        // 写入返回消息信息，消息类型应为此消息的发送者client的pid（使得对应client能够收到）
        msg.mtype = msg.mfrom;
        msg.mfrom = pid;
        // server发送返回消息给对应的client
        msgsnd(msgqid, &msg, sizeof(msg) - sizeof(long), 0);
        printf("(SERVER  %d) sent return message %d of CLIENT %ld\n",
                                                pid, msg.mindex, msg.mtype);
    }

    return 0;
}
```

b. 客户端程序 msg_client.c

msg_client.c 主要代码如下：

```
void client(int clientid)
{
    int msgqid, pid;
    // 申请得到key=75的消息队列（如果不存在，则重复申请）
```

```
    while ((msgqid = msgget(MSGKEY, 0777)) == -1)
        ;
    // 获得当前client进程pid
    pid = getpid();
    // 依此发送编号10~1的消息
    for (int i = 10; i >= 1; i--)
    {
        // 设置消息信息
        msg.mfrom = pid;
        msg.mtype = SERVER;
        msg.mindex = i;
        // client向server发送消息
        msgsnd(msgqid, &msg, sizeof(msg) - sizeof(long), 0);
        printf("(CLIENT%d %d) sent message %d\n", clientid, pid, i);
        // client接收server的返回消息
        msgrcv(msgqid, &msg, sizeof(msg) - sizeof(long), pid, 0);
        printf("(CLIENT%d %d) received return message %d from SERVER %d\n",
                                    clientid, pid, msg.mindex, msg.mfrom);
    }
}

int main(int argc, char *argv[])
{
    int clientid = atoi(argv[1]);
    client(clientid);
    return 0;
}
```

c. 引子程序 lab3.c

作用：引子程序，先后 fork() 两个子进程，父进程执行服务端 SERVER 程序，两个子进程分别执行客户端 CLIENT1 程序和客户端 CLIENT2 程序，使得开始通信。

lab3.c 主要代码如下：

```
int main(int argc, char *argv[])
{
    int child_process_num = 2;
    int i;
    pid_t pid;
    // 创建2个子进程
    for (i = 0; i < child_process_num; i++)
    {
```

```
        while ((pid = fork()) < 0)
            ;
        if (pid == 0)
            break;
    }
    // 判断当前进程，选择对应程序执行
    if (pid > 0)
    {
        // 父进程，执行服务器程序
        char *argv_list[] = {"./msg_server", NULL};
        execv(argv_list[0], argv_list);
    }
    else
    {
        if (i == 0)
        {
            // 子进程1，执行客户端程序，传入参数1
            char *argv_list[] = {"./msg_client", "1", NULL};
            execv(argv_list[0], argv_list);
        }
        else if (i == 1)
        {
            // 子进程2，执行客户端程序，传入参数2
            char *argv_list[] = {"./msg_client", "2", NULL};
            execv(argv_list[0], argv_list);
        }
    }
    return 0;
}
```

2.4 实验结果

实验结果如下图：（输出较多，仅截取输出开始部分与结束部分）

```
o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab3$ ./run.sh
(SERVER  7542) created message queue with key 75
(CLIENT1 7543) sent message 10
(SERVER  7542) received message 10 from CLIENT 7543
(CLIENT2 7544) sent message 10
(SERVER  7542) sent return message 10 of CLIENT 7543
(SERVER  7542) received message 10 from CLIENT 7544
(CLIENT1 7543) received return message 10 from SERVER 7542
(CLIENT1 7543) sent message 9
(SERVER  7542) sent return message 10 of CLIENT 7544
(SERVER  7542) received message 9 from CLIENT 7543
(CLIENT2 7544) received return message 10 from SERVER 7542
(CLIENT2 7544) sent message 9
(SERVER  7542) sent return message 9 of CLIENT 7543
(SERVER  7542) received message 9 from CLIENT 7544
(CLIENT1 7543) received return message 9 from SERVER 7542
(SERVER  7542) sent return message 9 of CLIENT 7544
(CLIENT1 7543) sent message 8
(SERVER  7542) received message 8 from CLIENT 7543
(CLIENT2 7544) received return message 9 from SERVER 7542
(CLIENT2 7544) sent message 8
```

```
(CLIENT2 7544) sent message 4
(SERVER  7542) received message 2 from CLIENT 7543
(SERVER  7542) sent return message 2 of CLIENT 7543
(CLIENT1 7543) received return message 2 from SERVER 7542
(CLIENT1 7543) sent message 1
(SERVER  7542) received message 4 from CLIENT 7544
(SERVER  7542) sent return message 4 of CLIENT 7544
(CLIENT2 7544) received return message 4 from SERVER 7542
(SERVER  7542) received message 1 from CLIENT 7543
(CLIENT2 7544) sent message 3
(SERVER  7542) sent return message 1 of CLIENT 7543
(SERVER  7542) received message 3 from CLIENT 7544
(CLIENT1 7543) received return message 1 from SERVER 7542
(SERVER  7542) sent return message 3 of CLIENT 7544
(CLIENT2 7544) received return message 3 from SERVER 7542
(CLIENT2 7544) sent message 2
(SERVER  7542) received message 2 from CLIENT 7544
(SERVER  7542) sent return message 2 of CLIENT 7544
(CLIENT2 7544) received return message 2 from SERVER 7542
(CLIENT2 7544) sent message 1
(SERVER  7542) received message 1 from CLIENT 7544
(SERVER  7542) sent return message 1 of CLIENT 7544
(CLIENT2 7544) received return message 1 from SERVER 7542
o2igin@DESKTOP-60A5SFR:/mnt/d/files/Course/2023Fal-操作系统/lab/lab3$
```

2.5 结果分析

- 多次运行程序，发现 `server.c` 总是可成功创建消息队列，说明客户端与服务器的 `msgget` 的不同参数设置有效，确实使得消息队列总是由Server创建；

- 分析输出结果，可知Server(PID 7542)总是接收Client1(PID 7543)、Client2(PID 7544)的消息，并返回消息；而两个Client总是接收到Server的消息，说明实验程序逻辑无误。

03 思考

（1）如何保证先由 SERVER 端创建一个 Key 为75的消息队列后， CLIENT 端才能使用该消息队列;

通过设置Server和Client获得消息队列时 `msgget` 的参数实现，具体如下：

- Server： `while((msgqid=msgget(MSGKEY,0777|IPC_CREAT|IPC_EXCL))==-1)`
  - 此参数设置使得消息队列必须由当前进程（Server）创建，如果已经存在，则返回-1；
  - 使用 `while` 循环重复申请，直到创建成功；
- Client： `while ((msgqid = msgget(MSGKEY, 0777)) == -1) {}`
  - 此参数设置使得申请的消息队列必须已经存在，Client仅仅是获得这个消息队列（而不主动创建）；
  - 如果不存在此消息队列，则返回-1，循环重复申请（直到Server创建消息队列成功Client才可成功获得）；

（2）请设计消息结构 `struct msgform` ，每个消息至少包含消息类型、消息编号、进程 `pid`;

消息结构体设置如下，
分析见2.1 设计消息结构体。

```c
struct msgform
{
    long mtype;       // 消息类型
    int mfrom;        // 消息发送者PID
    int mindex;       // 消息编号
    char mtext[1016]; // 消息文本
    // 其中 mtype 为消息队列中消息结构的固定内容，不计入消息大小
    // mform、mindex、mtext 大小共1024B，符合实验1K大小的消息要求
} msg;
```

（3）请设计消息类型，保证：服务端可以接收所有客户端发给服务端的消息；每个客户端只能接收服务端发给自己的返回消息，而不会接收发给其他客户端的消息；服务端和客户端都不能接收自己发出去的消息。

1. 使得服务端可以接收所有客户端发给服务端的消息
    - 令客户端发送给服务器的消息的 `mtype=1` ；
    - 令服务器接收消息时仅接收 `mtype=1` 的消息；
2. 使得每个客户端只能接收服务端发给自己的返回消息，而不会接收发给其他客户端的消息
    - 令客户端发给服务器的消息中带有 `mfrom` 字段（即客户端PID）；
    - 令服务器返回给客户端的消息中 `mtype` 字段设置为发送客户端的PID；
    - 客户端仅接收 `mtype` 与自己的PID相同的消息；
3. 服务端和客户端都不能接收自己发出去的消息
    - 此要求在 `1,2` 中已经得到满足。