实验一：进程间通信

实验时间：

6 小时

实验目的：

初步了解 Linux 系统中，进程间通信的方法。

实验目标：

编写一个程序，用 Linux 中的 IPC 机制，完成两个进程“石头、剪子、布”的游戏。

背景知识：

1．Linux 中的 IPC 机制简介

进程间通信（Interprocess Communication，IPC）实现了进程之间同步和交换数据的功能。本实验要求完成的是一个或几个用户态的进程，依靠内核提供的进程间通信的机制，完成几个用户进程之间的通信。通常，在 Linux 中允许以下几种进程间通信的机制：

管道和命名管道（FIFO）

最适合在进程之间实现生产者/消费者的交互。有些进程往管道中写入数据，而另外一些进程则从管道中读出数据。

信号量

这就是我们在原理课程中讲述的内核信号量的用户态版本。

消息

允许进程异步的交换消息（小块数据）。可以认为消息是传递附加信息的信号。

共享内存区

当进程之间在高效的共享大量数据的时候，这是一种最适合实现的交互方式。

另外，Linux 也允许相同主机上的进程之间，利用套接字（socket）进行通信。但套接字引入的最初目的是为了应用程序和网络接口之间实现数据通信，因此在这里就不再介绍。

2．FIFO

POSIX 引入了一个名为 mkfifo()的系统调用专门用来创建 FIFO，mkfifo()的函数原型为：

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

mkfifo()会根据参数 pathname 建立特殊的 FIFO 文件，该文件必须不存在，而参数 mode 为该文件的权限。FIFO 一旦被创建，就可以使用普通的 open()、read()、write()和 close()系统调用进行访问。

mkfifo()函数若成功，则返回 0，否则返回-1，错误原因存于 errno 中。

下面是一段示例代码。

```
//判断 FIFO 是否存在，如果不存在则建立
if (access(FIFO_NAME, F_OK) == -1) {
    res = mkfifo(FIFO_NAME, 0777);
    if (res != 0) {
        fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
        exit(1);
    }
}
```

3．System V IPC

System V IPC 包含一组系统调用，上文提及的利用信号量、消息和共享内存区进行进程间通信，都属于其中。IPC 数据结构是在进程请求 IPC 资源（信号量、消息队列或共享内存区）时动态创建的。每个 IPC 资源都是持久的，除非被进程显式的释放，否则永远驻留在内存中。由于一个进程可能需要相同类型的多个 IPC 资源，因此每个新资源都使用一个 32 位的 IPC 关键字来标识。这里重点介绍消息队列通信。

(1) 系统调用 msgget()

为了创建一个新的消息队列，或者访问一个现有的队列，可以使用系统调用 msgget()，其函数原型为：

```
int msgget(ket_t key, int msgflg);
```

msgget()的第一个参数是 IPC 关键字的值。这个关键字的值将被拿来与内核中其他消息队列的现有值相比较。比较之后，打开或访问操作依赖于第二个参数的内容。

- IPC_CREAT——如果内核中不存在该队列，则创建它。
- IPC_EXCL——与 IPC_CREAT 一起使用时，若队列早已存在则将出错。

如果只使用了 IPC_CREAT，msgget()或者返回新创建消息队列的标识符，或者返回现有的具有同一个关键字值的队列的标识符。如果同时使用了 IPC_CREAT 和 IPC_EXCL，那么将可能会有两个结果。或者创建一个新的队列，或者如果该队列存在，则调用将出错，并返回-1。IPC_EXCL 本身是没有什么用处的，但在与 IPC_CREAT 组合使用时，它可以用于保证没有一个现存的队列为

了访问而被打开。

有个可选的八进制许可模式，它是与掩码进行 OR 操作以后得到的。这是因为从功能上讲，每个 IPC 对象的访问权限与 Linux 文件系统的文件许可权是相似的。

下面是两段示例代码，说明如何创建一个新的消息队列，以及如何使用一个已有的消息队列。

```
//建立消息队列，只有相同用户可以使用
queue_id = msgget(QUEUE_ID, IPC_CREAT | IPC_EXCL | 0600);
if (queue_id == -1) {
    perror("main: msgget");
    exit(1);
}

//取得消息队列 ID
queue_id = msgget(QUEUE_ID, 0);
if (queue_id == -1) {
    perror("main: msgget");
    exit(1);
}
```

(2) 消息缓冲区

这个特殊的数据结构可以认为是消息数据的模板。虽然定义这种类型的数据结构是程序员的职责，但是读者绝对有必要知道实际上存在 msgbuf 类型的结构。在 linux/msg.h 中，它的定义如下：

```
/* message buffer for msgsnd and msgrcv calls */
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

msgbuf 结构中有两个成员：

- mtype——它是消息类型，以正数表示。
- mtext——它就是消息数据。

不要被消息数据元素（mtext）的名称所误导，应用程序的编程人员是可以重新定义 msgbuf 这个结构的。对于给定消息的最大的大小，存在一个内部的限制。在 Linux 中，它在 linux/msg.h 中定义。

(3) 系统调用 msgsnd()

一旦获得了消息队列的标识符，就可以开始在该队列上执行相关操作了。为了向队列传递消息，可以使用 msgsnd()系统调用：

```
int msgsnd(int msqid, struct msgbuf * msgp, size_t msgsz, int msgflg);
```

第一个参数是消息队列的标识符，它是前面调用 msgget()获得的返回值。

第二个参数是一个指针，指向我们重新定义和载入的消息缓冲区。msgsz 则包含了消息的大小，它是以字节为单位的，其中不包括消息类型的长度。

msgflg 可以设置为 0（忽略），也可以设置为 IPC_NOWAIT。如果消息队列已满，则消息将不会被写入到队列中，控制权将被还给调用进程。如果没有指定 IPC_NOWAIT，则调用进程将被阻塞，直到可以写消息为止。

下面是一段示例程序：

```
//申请消息缓冲区
msg = (struct msgbuf*)malloc(sizeof(struct msgbuf)+MAX_MSG_SIZE);
//发送数据
msg->mtype = 1;
sprintf(msg->mtext, "hello world");
rc = msgsnd(queue_id, msg, strlen(msg->mtext)+1, 0);
if (rc == -1) {
    perror("main: msgsnd");
    exit(1);
}
```

(4) 系统调用 msgrcv()

一旦消息队列中有消息了，进程就可以从队列中获得消息，可以使用系统调用 msgrcv()来完成这个功能：

```
ssize_t msgrcv(int msqid, struct msgbuf * msgq, size_t msgsz, long msgtype, int msgflg);
```

显然，第一个参数是用来指定在消息获取过程中所使用的队列的。第二个参数代表消息缓冲区变量的地址，获取的消息存放在这里。第三个参数代表消息缓冲区结构的大小，不包括 mtype 成员的长度，可以使用下面的公式来计算大小：

```
msgsz = sizeof(struct mymsgbuf) – sizeof(long);
```

第四个参数指定要从队列中获取的消息的类型。内核将查找队列中具有匹配类型的最老的消息，并把它的一个拷贝返回到由 msgp 变量指定的地址中。如果把 IPC_NOWAIT 作为一个参数传递给 msgflg，而队列中没有任何消息。则该次调用会向调用进程返回 ENOMSG。否则，调用进程将阻塞，直到满足 msgrcv()参数的消息到达队列为止。

下面是一段示例程序：

```
//申请消息缓冲区
msg = (struct msgbuf*)malloc(sizeof(struct msgbuf)+MAX_MSG_SIZE);
//读取消息队列中的数据
rc = msgrcv(queue_id, msg, MAX_MSG_SIZE+1, msg_type, 0);
if (rc == -1) {
    perror("main: msgrcv");
    exit(1);
}
```

实验步骤：

本实验可以创建三个进程，其中，一个进程为裁判进程，另外两个进程为选手进程。可以将“石头、剪子、布”这三招定义为三个整型值。胜负关系：石头〉剪子〉布〉石头。

选手进程按照某种策略（例如，随机产生）出招，交给裁判进程判断大小。裁判进程将对手的出招和胜负结果通知选手。比赛可以采取多盘（＞１００盘）定胜负，由裁判宣布最后结果。每次出招由裁判限定时间，超时判负。

每盘结果可以存放在文件或其他数据结构中。比赛结束，可以打印每盘的胜负情况和总的结果。

1．设计表示“石头、剪子、布”的数据结构，以及它们之间的大小规则。
2．设计比赛结果的存放方式。
3．选择 IPC 的方法。
4．根据你所选择的 IPC 方法，创建对应的 IPC 资源。
5．完成选手进程。
6．完成裁判进程。
－－以下要求选作：
７．决出班级的前三甲，与另外班级的前三甲比赛，决出年级冠军。
８．如果有兴趣，再把这个实验改造成网络版。即在设计时就要考虑ＩＰＣ层的封装。

实验结果：

实验步骤 1 的数据结构：

实验步骤 1 的大小规则：

实验步骤 2 的结构：

实验步骤 3 中，你所选择的 IPC 方法：

实验步骤 3，为何选择该方法？

实验步骤 3，如果选择消息队列机制，描述消息缓冲区结构：

实验步骤 4，如何创建 IPC 资源？

实验步骤 5，程序主要流程或关键算法：

实验步骤 6，程序主要流程或关键算法：

实验部分结果的展示：

思考题：

1．比较 Linux 系统中的几种 IPC 机制，并说明它们各适用于哪些场合？
2．试探索 Linux 内核中的 IPC 机制。