

苏州大学实验报告

院、系	计算机学院	年级专业	20 计科	姓名	柯骅	学号	2027405033
课程名称	操作系统课程实践					成绩	
指导教师	李培峰	同组实验者	无	实验日期	2023.03.09		

实验名称 实验 2 进程通信与进程同步

一、实验目的

1. 理解进程间通信的概念和方法。
2. 掌握常用的 Linux 进程间通信的方法。
3. 加强对进程同步和互斥的理解，学会使用信号量解决资源共享问题。
4. 熟悉 Linux 进程同步原语。
5. 掌握信号量 wait/signal 原语的使用方法，理解信号量的定义、赋初值及 wait/signal 操作。

二、实验内容

1. （实验 4.1：两个进程相互通信）

- (1) 编写 C 程序，使用 Linux 中的 IPC 机制完成“石头、剪刀、布”游戏。
- (2) 修改上述 C 程序，使之能够在网络上运行。

2. （实验 4.2：进程同步实验）

编写 C 程序，使用 Linux 操作系统中的信号量机制模拟解决经典的进程同步问题：生产者-消费者问题。假设有一个生产者和一个消费者，缓冲区可以存放产品，生产者不断生产产品并存入缓冲区，消费者不断从缓冲区中取出产品并消费

三、实验步骤

1. （实验 4.1：两个进程相互通信）

（1）使用消息队列实现 IPC 的猜拳游戏

可以创建三个进程：一个进程为裁判进程，另外两个进程为选手进程。可将“石头、剪子、布”这三招定义为三个整型值，胜负关系为：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（如 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。

具体的实验步骤包括：

- ❖ 设计表示“石头、剪子、布”的数据结构 Game，其中存储了随机产生的选手的出拳情况，分别用 012 表示，以及它们之间的大小规则 judge() 函数，规则如下：0>1>2>0
- ❖ 设计比赛结果的存放方式：将每次的猜拳结果存放在数列 result_list 中，第 i 次结果存放在 result_list[i-1] 中
- ❖ 选择 IPC 的方法：消息机制
- ❖ 根据你所选择的 IPC 方法，创建对应的 IPC 资源，本次选择的是消息队列来实现，具体操作说明如下：

1. `int msgget(key_t key, int msgflg)`

作用：得到消息队列标识符或创建一个消息队列对象并返回消息队列标识符

函数传 入值	key	0(IPC_PRIVATE): 会建立新的消息队列
		大于0的32位整数: 视参数msgflg来确定操作。通常要求此值来源于ftok返回的IPC键值
	msgflg	0: 取消息队列标识符, 若不存在则函数会报错
		IPC_CREAT: 当msgflg&IPC_CREAT为真时, 如果内核中不存在键值与key相等的消息队列, 则新建一个消息队列; 如果存在这样的消息队列, 返回此消息队列的标识符
		IPC_CREAT IPC_EXCL: 如果内核中不存在键值与key相等的消息队列, 则新建一个消息队列; 如果存在这样的消息队列则报错
函数返 回值	成功: 返回消息队列的标识符	
	出错: -1, 错误原因存于error中	
附加说 明	上述msgflg参数为模式标志参数, 使用时需要与IPC对象存取权限 (如0600) 进行 运算来确定消息队列的存取权限	
错误代 码	EACCES: 指定的消息队列已存在, 但调用进程没有权限访问它 EEXIST: key指定的消息队列已存在, 而msgflg中同时指定IPC_CREAT和IPC_EXCL标志 ENOENT: key指定的消息队列不存在同时msgflg中没有指定IPC_CREAT标志 ENOMEM: 需要建立消息队列, 但内存不足 ENOSPC: 需要建立消息队列, 但已达到系统的限制	

1. `int msgctl(int msqid, int cmd, struct msqid_ds *buf)`

作用：获取和设置消息队列的属性

函数传 入值	msqid	消息队列标识符
	cmd	IPC_STAT: 获得msqid的消息队列头数据到buf中
		IPC_SET: 设置消息队列的属性, 要设置的属性需先存储在buf中, 可设置的属性包括: msg_perm.uid、msg_perm.gid、msg_perm.model以及msg_qbytes
		buf: 消息队列管理结构体, 请参见消息队列内核结构说明部分
函数返 回值	成功: 0	
	出错: -1, 错误原因存于error中	
错误代 码	EACCESS: 参数cmd为IPC_STAT, 确无权限读取该消息队列 EFAULT: 参数buf指向无效的内存地址 EIDRM: 标识符为msqid的消息队列已被删除 EINVAL: 无效的参数cmd或msqid EPERM: 参数cmd为IPC_SET或IPC_RMID, 却无足够的权限执行	

1. `int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)`

说明：将 msgp 消息写入到标识符为 msqid 的消息队列

函数传 入值	msqid	消息队列标识符
	msgp	发送给队列的消息。msgp可以是任何类型的结构体, 但第一个字段必须为long类型, 即表明此发送消息的类型, msgrcv根据此接收消息。msgp定义的参照格式如下: struct s_msg{ /*msgp定义的参照格式*/ long type; /* 必须大于0,消息类型 */ char mtext[256]; /*消息正文, 可以是其他任何类型*/ } msgp;
	msgsz	要发送消息的大小, 不含消息类型占用的4个字节,即mtext的长度
	msgflg	0: 当消息队列满时, msgsnd将会阻塞, 直到消息能写进消息队列
		IPC_NOWAIT: 当消息队列已满的时候, msgsnd函数不等待立即返回
		IPC_NOERROR: 若发送的消息大于size字节, 则把该消息截断, 截断部分将被丢弃, 且不通知发送进程。

函数返回值	成功: 0
	出错: -1, 错误原因存于error中
错误代码	EAGAIN: 参数msgflg设为IPC_NOWAIT, 而消息队列已满 EIDRM: 标识符为msqid的消息队列已被删除 EACCESS: 无权限写入消息队列 EFAULT: 参数msgp指向无效的内存地址 EINTR: 队列已满而处于等待情况下被信号中断 EINVAL: 无效的参数msqid、msgsz或参数消息类型type小于0

msgsnd()为阻塞函数,当消息队列容量满或消息个数满会阻塞。消息队列已被删除,则返回 EIDRM 错误;被信号中断返回 E_INTR 错误。

如果设置 IPC_NOWAIT 消息队列满或个数满时会返回-1, 并且置 EAGAIN 错误。

msgsnd()解除阻塞的条件有以下三个条件:

- ①不满足消息队列满或个数满两个条件,即消息队列中有容纳该消息的空间。
- ②msqid 代表的消息队列被删除。
- ③调用 msgsnd 函数的进程被信号中断。

1.
- ```
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

作用: 从标识符为 msqid 的消息队列读取消息并存储于 msgp 中,读取后把此消息从消息队列中删除

|       |                      |                                                            |
|-------|----------------------|------------------------------------------------------------|
| 函数传入值 | msqid                | 消息队列标识符                                                    |
|       | msgp                 | 存放消息的结构体, 结构体类型要与msgsnd函数发送的类型相同                           |
|       | msgsz                | 要接收消息的大小, 不含消息类型占用的4个字节                                    |
|       | msgtyp               | 0: 接收第一个消息                                                 |
|       |                      | >0: 接收类型等于msgtyp的第一个消息                                     |
|       |                      | <0: 接收类型等于或者小于msgtyp绝对值的第一个消息                              |
|       | msgflg               | 0: 阻塞式接收消息, 没有该类型的消息msgrcv函数一直阻塞等待                         |
|       |                      | IPC_NOWAIT: 如果没有返回条件的消息调用立即返回, 此时错误码为ENOMSG                |
|       |                      | IPC_EXCEPT: 与msgtype配合使用返回队列中第一个类型不为msgtype的消息             |
|       |                      | IPC_NOERROR: 如果队列中满足条件的消息内容大于所请求的size字节, 则把该消息截断, 截断部分将被丢弃 |
| 函数返回值 | 成功: 实际读取到的消息数据长度     |                                                            |
|       | 出错: -1, 错误原因存于error中 |                                                            |

msgrcv()解除阻塞的条件有以下三个:

- ①消息队列中有了满足条件的消息。
- ②msqid 代表的消息队列被删除。
- ③调用 msgrcv() 的进程被信号中断。

在本次实验中, 首先使用 msgget() 函数创建消息队列:

1.
- ```
msgid1 = msgget(key1, IPC_CREAT | 0666);
```
2.
- ```
msgid2 = msgget(key2, IPC_CREAT | 0666);
```

msgflg (0666) 可以与 IPC\_CREAT 做或操作, 表示当 key 所命名的消息队列不存在时创建一个消息队列, 如果 key 所命名的消息队列存在时, IPC\_CREAT 标志会被忽略, 而只返回一个标识符。key 值初始化为大于 0 的 32 位整数, 建立新的消息队列。

随后, 使用 fork() 创建子进程, 子进程使用 msgsnd() 函数将两位选手的出拳情况传送进消息队列:

```
1. msgsnd(msgid, &game, sizeof(int), 0);
```

其中, msgid 是 msgget() 的返回值, 即消息队列识别符, game 为自定义类 Game 的实例, 用于存储选手出拳情况, sizeof(int) 由 game 中的有效数据大小决定, 0 表示当消息队列满时, msgsnd 将会阻塞, 直到消息能写进消息队列。

主进程等待两位选手的出拳信息传入消息队列后, 使用 msgrcv() 读取消息队列中的信息:

```
1. msgrcv(msgid1, &game1, sizeof(game1) - sizeof(long), 0, 0);
```

msgid1 是 msgget() 的返回值, 即消息队列识别符, game 为自定义类 Game 的实例, 用于存储选手出拳情况, sizeof() 由 game 中的有效数据大小决定, 第一个 0 表示接收消息队列中的第一个消息, 第二个 0 表示“阻塞式接收消息”, 没有该类型的消息 msgrcv 函数将一直阻塞等待。

最后, 使用 msgctl() 函数删除消息队列:

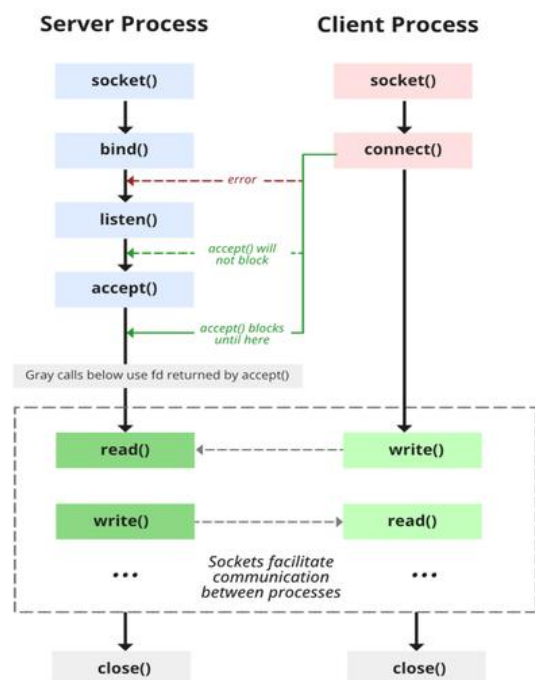
```
1. if (msgctl(msgid1, IPC_RMID, 0) == -1 || msgctl(msgid2, IPC_RMID, 0) == -1)
 msgid 是 msgget() 的返回值, 即消息队列识别符, IPC_RMID 表示删除消息队列。
```

❖ **完善整个流程:** 创建消息队列, 使用 fork 创建子进程, 用于随机产生选手出拳信息并将其放入消息队列, 主进程等待子进程结束后读取出拳信息, 比较, 输出结果并写入文件

## (2) 使用套接字实现, 使之能够在网络上运行

socket 原理如下:

服务端先创建一个套接字, 端口绑定, 对端口进行监听, 调用 accept 阻塞, 等待客户端连接。客户端创建一个套接字, 然后通过三次握手完成 tcp 连接后服务端 accept 返回重新建立一个套接字代表返回客户端的 tcp 连接, (在 accept 成功返回前有一个要注意的是 server 会有两个队列, 一个存放完成三次握手的, 一个存放未完成三次握手的, 每次 accept 会从完成三次握手的队列中取出一个并一直建立 TCP 连接, 此时才能算是真正的连接成功), 完成上面的步骤后即可开始数据的传输了, 数据传输结束后再调用 close 关闭连接



State diagram for server and client model of Socket



TCP 与 UDP 采用套接字的一般编写流程如下：

| TCP编程的服务器端一般步骤是：                                                                                                                                                                                                               | UDP编程的服务器端一般步骤是：                                                                                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1、创建一个socket，用函数socket();<br>2、设置socket属性，用函数setsockopt(); * 可选<br>3、绑定IP地址、端口等信息到socket上，用函数bind();<br>4、开启监听，用函数listen();<br>5、接收客户端上来的连接，用函数accept();<br>6、收发数据，用函数send()和recv(), 或者read()和write();<br>7、关闭网络连接;<br>8、关闭监听; | 1、创建一个socket，用函数socket();<br>2、设置socket属性，用函数setsockopt();* 可选<br>3、绑定IP地址、端口等信息到socket上，用函数bind();<br>4、循环接收数据，用函数recvfrom();<br>5、关闭网络连接;                             |
| TCP编程的客户端一般步骤是：                                                                                                                                                                                                                | UDP编程的客户端一般步骤是：                                                                                                                                                         |
| 1、创建一个socket，用函数socket();<br>2、设置socket属性，用函数setsockopt();* 可选<br>3、绑定IP地址、端口等信息到socket上，用函数bind();* 可选<br><br>4、设置要连接的对方的IP地址和端口等属性;<br>5、连接服务器，用函数connect();<br>6、收发数据，用函数send()和recv(), 或者read()和write();<br>7、关闭网络连接;      | 1、创建一个socket，用函数socket();<br>2、设置socket属性，用函数setsockopt();* 可选<br>3、绑定IP地址、端口等信息到socket上，用函数bind();* 可选<br><br>4、设置对方的IP地址和端口等属性;<br>5、发送数据，用函数sendto();      6、关闭网络连接; |

在本次实验中，采用了 TCP socket 来实现 IPC，

在服务端（caipan.c）中先使用 sockert() 函数创建一个 socket 实例，然后使用 build() 函数绑定 IP 地址和端口到已经创建的 socket 上，随后开启监听，等待客户端发送猜拳信息，当接收到客户端使用时间为种子生成的随机出拳信息后，服务端裁判判断输赢，并将结果告知两位选手。

在客户端（xuanshou1.c/xuanshou2.c）中，同样先创建一个 socket 实例，然后绑定 IP 地址和端口到已经创建的 socket 上，随后使用 connect() 连接到服务端，并发送随机产生的出拳信息，等待服务端返回结果并输出。

## 2. （实验 4.2：进程同步实验）

为了模拟解决生产者-消费者问题，需要创建两个进程，即生产者进程和消费者进程，并且要让这两个进程共享同一个缓冲区。生产者进程和消费者进程必须为并发执行的进程。

本次的实验解决了一个非常简单的生产者-消费者问题。代码中使用两个线程来模拟生产者和消费者，并且使用了 pthread 库提供的线程操作，因此需要包含头文件 pthread.h。至于信号量机制，代码中使用了 POSIX 信号量机制，该机制通常用于线程同步，因此需要包含头文件 semaphore.h

操作说明如下：

### 1. `int sem_init(sem_t *sem,int pshared,unsigned int value);`

sem\_init 函数用于创建信号量，该函数初始化由 sem 指向的信号对象，设置它的共享选项，并给它一个初始的整数值。pshared 控制信号量的类型，如果其值为 0，就表示这个信号量是当前进程的局部信号量，否则信号量就可以在多个进程之间共享，value 为 sem 的初始值。调用成功时返回 0，失败返回-1。

### 1. `int sem_wait(sem_t *sem);`

sem\_init 函数用于以原子操作的方式将信号量的值减 1。成功时返回 0，失败时返回-1。

### 1. `int sem_post(sem_t *sem);`

sem\_post 函数用于以原子操作的方式将信号量的值加 1。成功时返回 0，失败时返回-1。

### 1. `int sem_destroy(sem_t *sem);`

sem\_destroy 函数用于对用完的信号量的清理。成功时返回 0，失败时返回-1。

## 生产者消费者模式:

### 生产者

...  
生产一个产品  
...

1) 判断是否能获得一个空缓冲区, 如果不能则阻塞

**C1:** 把产品放入指定缓冲区

2) 满缓冲区数量加1, 如果有消费者由于等消费产品而被阻塞, 则唤醒该消费者

同步: 判断

临界区

同步: 通知

### 消费者

1) 判断是否能获得一个满缓冲区, 如果不能则阻塞

从满缓冲取出一个产品

2) 空缓冲区数量加1, 如果有生产者由于等空缓冲区而阻塞, 则唤醒该生产者

同步: 判断

临界区

同步: 通知

### 生产者:

```
{
 ...
 生产一个产品
 ...
 wait(empty);
 wait(m);
 ...
 C1: 把产品放入指定缓冲区
 ...
 signal(m);
 signal(full);
}
```

当empty大于0时, 表示有空缓冲区, 继续执行; 否则, 表示无空缓冲区, 当前生产者阻塞。

把full值加1, 如果有消费者等在full的队列上, 则唤醒该消费者。

### 消费者:

```
{
 ...
 wait(full);
 wait(m);
 ...
 C2: 从指定缓冲区取出产品
 ...
 signal(m);
 signal(empty);
 ...
 消费取出的产品
 ...
}
```

当full大于0时, 表示有满缓冲区, 继续执行; 否则, 表示无满缓冲区, 当前消费者阻塞。

把empty值加1, 如果有生产者等在empty的队列上, 则唤醒该生产者。

在本次实验中, 利用 linux 中的 `sem_wait()` 和 `sem_post()` 分别实现 PV 操作, 按照上述流程完成相应程序编写。

## 四、实验结果

### 1. (实验 4.1: 两个进程相互通信)

#### (1) 使用消息队列实现 IPC 的猜拳游戏

```
kh@ubuntu:~/test/test2/IPC$./game_msgq
Game start, please input rounds:10
ROUND.1: end in a draw
ROUND.2: end in a draw
ROUND.3: A win
ROUND.4: A win
ROUND.5: B win
ROUND.6: B win
ROUND.7: A win
ROUND.8: end in a draw
ROUND.9: B win
ROUND.10: end in a draw

The final result:
A win:3
B win:3
end in a draw:4
```

```
打开(O) result.txt 保存(S) 结果
~/test/test2/IPC
ROUND.1: end in a draw
ROUND.2: end in a draw
ROUND.3: A win
ROUND.4: A win
ROUND.5: B win
ROUND.6: B win
ROUND.7: A win
ROUND.8: end in a draw
ROUND.9: B win
ROUND.10: end in a draw

The final result:
A win:3
B win:3
end in a draw:4
```

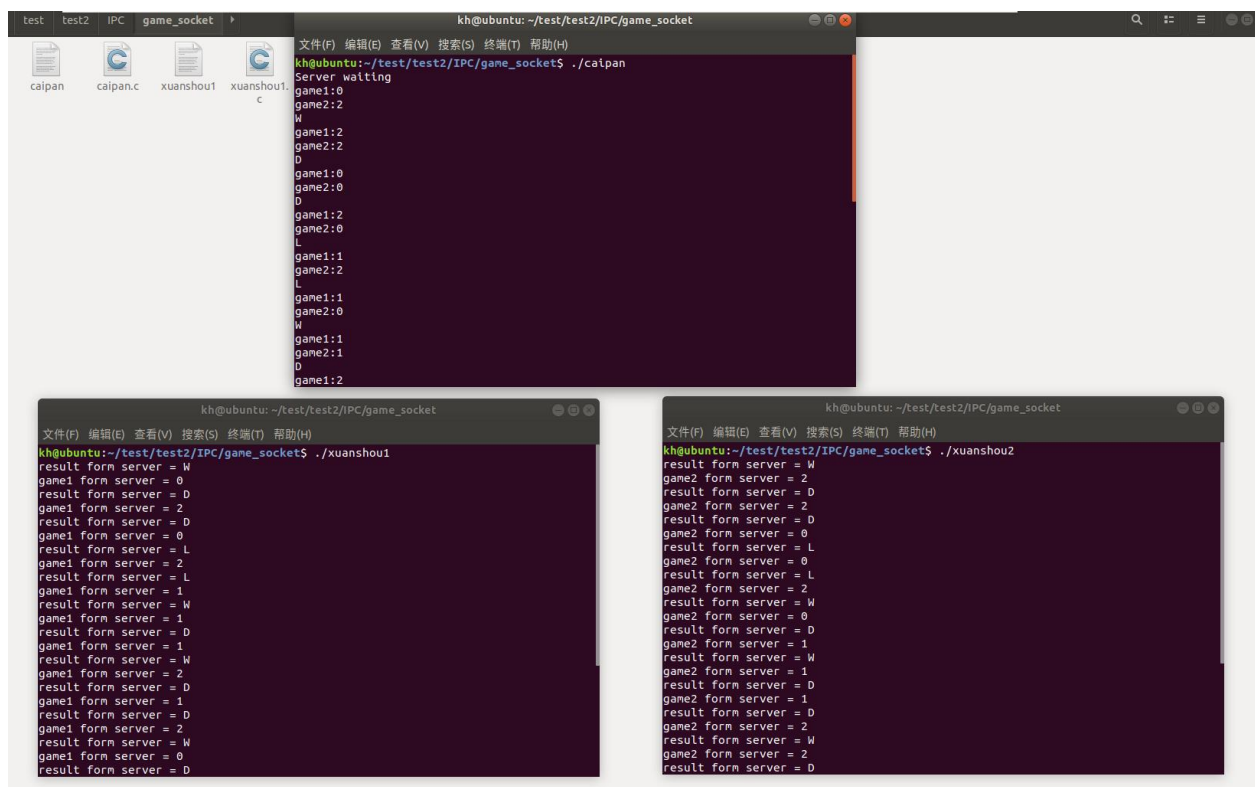
游戏模拟进行了 10 轮, AB 各赢了 3 轮, 平局 4 轮, 文本与命令行一致

## (2) 使用套接字实现，使之能够在网络上运行

在仅运行 caipan.c 时，等待客户端应答：

```
kh@ubuntu:~/test/test2/IPC/game_socket$./caipan
Server waiting
```

在 xuanshou1.c 与 xuanshou2.c 运行之后，进程之间实现了交互，不断地出拳并产生结果输出：



```
kh@ubuntu:~/test/test2/IPC/game_socket$./caipan
Server waiting
game1:0
game2:2
W
game1:2
game2:2
D
game1:0
game2:0
D
game1:2
game2:0
L
game1:1
game2:2
L
game1:1
game2:0
W
game1:1
game2:1
D
game1:2

kh@ubuntu:~/test/test2/IPC/game_socket$./xuanshou1
result form server = W
game1 form server = 0
result form server = D
game1 form server = 2
result form server = D
game1 form server = 0
result form server = L
game1 form server = 2
result form server = L
game1 form server = 1
result form server = W
game1 form server = 1
result form server = D
game1 form server = 1
result form server = W
game1 form server = 2
result form server = D
game1 form server = 1
result form server = D
game1 form server = 2
result form server = W
game1 form server = 0
result form server = D

kh@ubuntu:~/test/test2/IPC/game_socket$./xuanshou2
result form server = W
game2 form server = 2
result form server = D
game2 form server = 2
result form server = D
game2 form server = 0
result form server = L
game2 form server = 0
result form server = L
game2 form server = 2
result form server = W
game2 form server = 0
result form server = D
game2 form server = 1
result form server = W
game2 form server = 1
result form server = D
game2 form server = 2
result form server = W
game2 form server = 2
result form server = W
game2 form server = 2
result form server = D
```

## 2. （实验 4.2：进程同步实验）

输入产品（123456）至缓冲区之后，运行结果如下：

```
kh@ubuntu:~/test/test2/sync$./sync
input something to buffer:123456
read product from buffer:123456
The End...
```

## 五、实验思考与总结

### 1. （实验 4.1：两个进程相互通信）

（1）示例代码中随机数的取值对于模拟“石头、剪刀、布”游戏很重要，如果取值不当，就可能出现大量平局的情况，故请思考Linux随机数的合理取值方法。

有几种方法可以实现随机数的合理取值：①当以时间作为种子，取随机数时，需要将时间岔开，以保证随机数的合理。②以其他数值作为种子，如使用getpid()来获得进程PID。③用其他方法产生随机数。

（2）比较Linux 操作系统中的几种IPC机制，并说明它们各自适用于哪些场合。

管道：无名管道简单方便，但局限于单向通信的工作方式，并且只能在创建它的进程及其子孙进

程之间实现管道的共享：有名管道虽然可以提供给任意关系的进程使用，但是由于其长期存在于系统之中，使用不当容易出错。

消息队列：消息缓冲可以不再局限于父子进程，而允许任意进程通过共享消息队列来实现进程间通信，并由系统调用函数来实现消息发送和接收之间的同步，从而使得用户在使用消息缓冲进行通信时不再需要考虑同步问题，使用方便，但是信息的复制需要额外消耗 CPU 的时间，不适宜于信息量大或操作频繁的场所。

共享内存：共享内存针对消息缓冲的缺点改而利用内存缓冲区直接交换信息，无须复制，快捷、信息量大是其优点。但是共享内存的通信方式是通过将共享的内存缓冲区直接附加到进程的虚拟地址空间中来实现的，因此，这些进程之间的读写操作的同步问题操作系统无法实现。必须由各进程利用其他同步工具解决。另外，由于内存实体存在于计算机系统中，所以只能由处于同一个计算机系统内的诸进程共享。不方便网络通信。

## 2. （实验 4.2：进程同步实验）

（1）多线程并发与多进程并发有何不同与相同之处？

在单核环境下，多线程并发时，如果这些线程属于同一个进程，就属于同一进程的运行，整个进程的执行效率提高。如果这些线程分属于不同的进程，就意味着多进程之间的并发。在支持多线程的系统中，多进程并发就是多线程并发，而多线程并发不一定是多进程并发。在多核环境下，多线程还可能并行。

## 3. 实验总结

本次是进程通信与进程同步的实验。在进程通信中我们可能会用到很多种方法，他们各自特点不同，例如：套接字方法可以应用于网络，而管道机制编写简单，需要根据不同的场景来选择合适的方法，同时也运用并熟悉了 linux 中的消息机制和 TCP/UDP socket 的原语。

在进程同步的实验中，重新回顾了消费者生产者模式，同时运用了 linux 中的 sem 系列函数来切实地解决了进程同步问题，对于 linux 与 c 语言有了更深的理解。