**一、实验目的及要求**

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（P/V）原语的使用方法，**理解**信号量的定义、赋初值及wait/signal（P/V）操作。

**二、实验仪器设备与软件环境**

1．一台装有Windows的微型计算机（Win7或Win10）；

2．在微型计算机上（通过虚拟机）安装了Linux环境；

3．Linux环境下的编辑器（vi/vim、gedit或vscode），编译器（gcc），调试器（gdb）。

**三、实验基础**

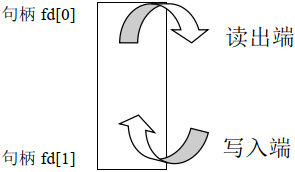
**（一）进程通信部分**

由于不同的进程运行在各自不同的内存空间中，一方对于变量的修改另一方是无法感知的。因此，进程之间的信息传递不可能通过变量或其它数据结构直接进行，只能通过进程间通信来完成。并发进程之间的相互通信是实现多进程间协作和同步的常用工具，具有很强的实用性，进程通信是操作系统内核层极为重要的部分。

进程间通信（IPC）就是在不同进程之间传播或交换信息。Linux下的进程通信手段基本上是从UNIX平台上的进程通信手段演变而来的。AT&T的贝尔实验室对UNIX早期的进程间通信手段进行了系统的改进和扩充，形成了“System V IPC”，但通信进程局限于单个计算机内部。加州大学伯克利分校的软件研发中心则形成了基于套接字（socket）的进程间通信机制。Linux把两者都继承了下来，因此Linux下的进程通信手段主要包括管道通信、消息队列、共享内存、信号量和套接字等。

**1、管道**

UNIX系统在 OS的发展上，最重要的贡献之一便是该系统首创了管道（pipe）。这也是 UNIX系统的一大特色。管道是指能够连接一个写进程和一个读进程的、并允许它们以生产者—消费者方式进行通信的一个共享文件，又称为pipe文件。由写进程从管道的写入端（句柄1）将数据写入管道，而读进程则从管道的读出端（句柄 0）读出数据。



管道包括无名管道和有名管道两种。

1）无名管道（用于父、子进程间）：一个临时文件。利用pipe( )建立起来的无名文件（无路径名）。只用该系统调用所返回的文件描述符来标识该文件，故只有调用pipe( )的进程及其子孙进程才能识别此文件描述符，才能利用该文件（管道）进行通信。

2）有名管道（用于任意两个进程间）：一个可以在文件系统中长期存在的、具有路径名的一种特殊类型的文件，也被称为FIFO文件。它克服无名管道使用上的局限性，可让更多的进程也能利用管道进行通信。因而其它进程可以知道它的存在，并能利用路径名来访问该文件。对有名管道的访问方式与访问其他文件一样，需先用open( )打开。

3）无名管道和有名管道的差异：使用无名管道则通信的进程之间需要一个父子关系，通信的两个进程一定是由一个共同的祖先进程启动，但无名管道没有数据交叉的问题。FIFO不同于无名管道之处在于它提供了一个路径名与之关联，以FIFO的文件形式存在于文件系统中，这样，即使与 FIFO的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过FIFO相互通信，因此，通过FIFO不相关的进程也能交换数据。且FIFO严格遵循先进先出（first in first out），对管道及FIFO的读总是从开始处返回数据，对它们的写则把数据添加到末尾。有名管道的名字存在于文件系统中，内容存放在内存中。

无名管道的优点是简单方便，缺点：1）局限于单向通信；2）只能创建在它的进程以及其有亲缘关系的进程之间；3）缓冲区有限。有名管道的优点是可以实现任意关系的进程间的通信，缺点：1）长期存于系统中，使用不当容易出错；2）缓冲区有限。

pipe文件的建立过程包括分配磁盘和内存索引结点、为读进程分配文件表项、为写进程分配文件表项、分配用户文件描述符。所涉及的系统调用有：

1）pipe( )

建立一无名管道格式： pipe(filedes)

参数定义

int pipe(filedes);

int filedes[2];

其中，filedes[1]是写入端，filedes[0]是读出端。

2）read( )

系统调用格式： read(fd,buf,nbyte)

功能：从fd所指示的文件中读出nbyte个字节的数据，并将它们送至由指针buf所指示的缓冲区中。如该文件被加锁、等待，直到锁打开为止。

参数定义：

int read(fd,buf,nbyte);

int fd;

char \*buf;

unsigned nbyte;

3）write( )

系统调用格式： read(fd,buf,nbyte)

功能：把nbyte个字节的数据，从buf所指向的缓冲区写到由fd所指向的文件中。如文件加锁，暂停写入，直至开锁。

参数定义同 read( )。进程间通过管道用write和read来传递数据，但write和read不能同时进行，在管道中只能有 4096 字节数据被缓冲。

4）sleep( )

系统调用格式： sleep(second);

功能：使现行进程暂停执行由自变量规定的秒数，用于进程的同步与互斥。

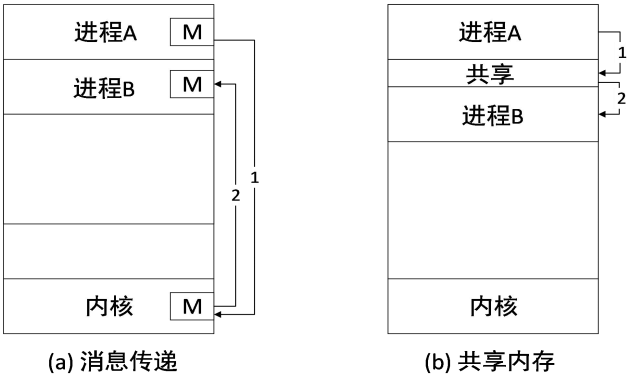
5）lockf( )

系统调用格式： lockf(fd,mode,size);

功能：对指定文件的指定区域（由size指示）进行枷锁或解锁，以实现进程的同步与互斥。其中fd是文件描述字，mode是锁定方式，=1表示加锁，=0表示解锁，size是指定文件fd的指定区域，用0表示从当前位置到文件尾。

**2、消息队列**

进程间通信有两种基本模式：共享内存和消息传递。在共享内存模式中，建立一块供协作进程共享的内存区域，进程通过向此共享区域读或写入数据来交换信息。在消息传递模式中，通过在协作进程间交换消息来实现通信。下图给出了这两种模式的对比。



上述两种模式在操作系统中都很常用，一般情况下消息传递对于交换少数量的数据很有用，因为不需要避免冲突且也容易实现。共享内存允许以最快的速度进行方便的通信，在计算机中它可以达到内存的速度。共享内存比消息传递快，因为其仅在建立共享内存区域时需要系统调用，一旦建立了共享内存，所有的访问都被处理为常规的内存访问，不需要来自内核的帮助，而消息传递系统通常用系统调用来实现，需要更多的内核介入的时间消耗。

消息队列（message queue，也被称为报文队列）是一个由消息链接而成的链表，它是消息的链式队列，被保存在内核中，可通过消息队列的引用标识符来访问，用于运行在同一台机器上的进程间通信。在Linux中，消息被放置在一个预定义的消息结构中，进程生成的消息则指明了该消息的类型，可以把它放入一个由系统负责维护的消息队列中。访问消息队列的进程可以根据消息的类型，有选择地从消息队列中遵照FIFO原则添加或读取特定类型的消息。消息队列具有一定的先进先出特性，但它可以实现消息的随机查询，并且克服了管道只能承载无格式字节流及缓冲区大小受限等缺点。

消息队列一旦被创建，就可由多个进程共享。发送消息的进程可以在任意时刻发送任意数量的消息到指定的消息队列中，并检查是否有接收进程在等待它所发送的消息，若有则将其唤醒。接收消息的进程则可以在需要消息时从指定的消息队列中获取消息，如果消息还没有到来，则转入阻塞状态并等待。

目前，Linux操作系统中有两种类型的消息队列：POSIX消息队列和System V消息队列。其中，System V消息队列已被大量使用。有关消息队列的操作通常有打开/创建消息队列、读写消息队列和获得/设置消息队列属性等。

**3、共享内存**

管道和消息队列的共同特点是通过内核来进行进程间通信。向管道、FIFO和消息队列写人数据时，需要把数据从进程复制到内核；读出数据时又需要从内核复制数据到进程。也就是说，进程间的通信必须借助内核，通过多次数据拷贝才能实现。

共享内存是指将同一块内存区映射到共享它的不同进程的地址空间中，共享内存是在进程之间共享和传递数据的一种简单但非常有效的方式。进程间的通信只需要对共享的内存区域进行操作，数据不再需要通过内核就可以在不同的进程间复制。

共享内存也是最高效的一种进程间通信方式，因为进程可以直接读写内存，这避免了对数据的各种不必要的复制。另外，进程之间在使用共享内存时，数据将一直保存在共享内存中，直到解除映射、通信完毕才会写回文件，从而达到高效通信的目的。但主要问题在于，当两个或多个进程使用共享内存进行通信时，系统内核并未对共享内存的访问提供同步机制，这容易造成不同进程在同时读写同一共享内存时数据不一致问题，因此程序员需要依靠某种同步机制(如互斥锁、信号量等)来同步进程对共享内存的访问。

在Linux中，每个进程的虚拟内存被分为多个页面，并且每个进程都会维护一个从内存地址到虚拟页面的映射关系(即页表)。尽管每个进程都有自己的内存地址，但不同的地址可以同时将同一内存页面映射到自己的地址空间，从而达到共享内存的目的。

Linux有两种共享内存机制——POSIX共享内存和System V共享内存，它们都是通过tmpfs（一种基于内存的文件系统，该文件系统的目录为/dev/shm）实现的。但POSIX共享内存是通过用户空间挂载的tmpfs文件系统实现的， 而System V共享内存则是通过内核本身的tmpfs文件系统实现的。两者的区别在于：System V共享内存是持久化的，只要机器不重启或不显式销毁，该共享内存就一直存在；而POSIX共享内存不是持久化的，如果进程被关闭，映射也将随之失效(事先映射到文件上的情况除外)。

需要注意的是，无论使用哪种共享内存机制，都必须注意数据存取的同步。通常，信号量被用于实现共享数据存取的同步，此外也可以通过shmctl()函数(如SHM\_LOCK、SHM\_UNLOCK等) 设置共享内存的某些标志位来实现共享数据存取的同步。

**4、信号量**

共享内存的缺点是没有同步机制。如何做到同一个共享的资源在同一时间只能被一个进程访问？这时信号量就出现了。信号量和共享内存往往要配合使用。

信号量本质上是一个计数器，也称为信号锁，可以用来控制多个进程对共享资源的访问。它和管道有所不同，它不以传送数据为主要目的，它主要是用来保护共享资源（信号量也属于临界资源），防止某进程正在访问共享资源时，其他进程也访问该资源。因此，信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据。本次实验中，安排在第二部分。

信号量通常通过两个原子操作P和V来访问：

P(s)：如果s的值大于零，就给它减1；如果它的值为零，就挂起该进程的执行；

V(s)：如果有其他进程因等待s而被挂起，就让它恢复运行，如果没有进程因等待s而挂起，就给它加1。

Linux提供了一组精心设计的信号量接口来对信号进行操作，这些函数都是用来对成组的信号量值进行操作的。它们声明在头文件sys/sem.h中。

**5、套接字**

套接字编程是实现Linux系统和其他大多数OS中进程间通信的主要方式之一。我们熟知的WWW服务、FTP服务、Telnet服务等都是基于套接字编程来实现的。除了适用于异地计算机进程间通信之外，套接字同样适用于本地同一台计算机内部的进程间通信。

套接字是一个通信标识类型的数据结构，包含通信目的地址、通信使用的端口号、网络通信的传输协议、进程所在的网络地址和针对客户或服务程序提供的不同系统调用（或 API 函数），是进程通信和网络通信的基本构件。套接字为客户-服务器模型而设计，通常包括两类：

● 基于文件型：通信进程都运行在同一台机器环境中，套接字基于本地文件系统支持，一个套接字关联到一个特殊文件，通信双方通过对该文件读写实现通信。

● 基于网络型：通常采用非对称方式通信，即发送者需要提供接收者命名。发送进程发出连接请求，随机申请一个套接字，主机为之分配一个端口与套接字绑定；接收进程拥有全局公认的套接字和指定端口（例如 http 服务器监听端口：80），并通过监听端口等待用户请求。

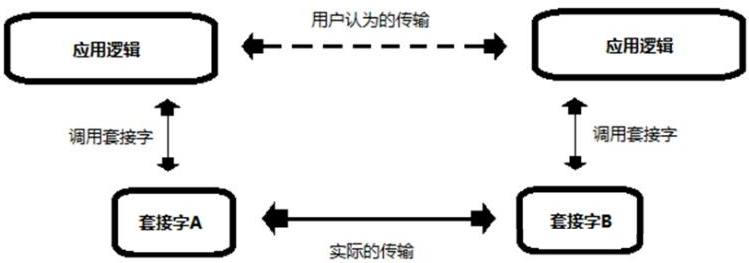
Linux支持多种套接字类型，主要包括以下3种：

（1）流式套接字，即TCP套接字，用SOCK\_STREAM表示；

（2）数据报套接字，即UDP套接字（或称无连接套接字），用SOCK\_DGRAM表示；

（3）原始套接字，用SOCK\_RAM表示。

Socket是面向“客户/服务器”（C/S）模型而设计的，针对客户和服务器程序提供不同的Socket系统调用。这种模式巧妙地解决了进程之间建立通信连接的问题。服务器Socket会公告给需要通信的一方。



假如通信双方都不知道对方的socket编号，就好比打电话的双方彼此不知道对方的电话号码，要通话是不可能的。采用了C/S模式后，由client端发起请求，server端接收请求，并应答，从而建立好链接。

1）Socket通信过程大致分为以下几个步骤：

服务器端的步骤如下：

（1）建立服务器端的 Socket，开始侦听整个网络中的连接请求。

（2）当检测到来自客户端的连接请求时，向客户端发送收到连接请求的信息，并建立与客户端之间的连接。

（3）当完成通信后，服务器关闭与客户端的 Socket 连接。

2）客户端的步骤如下：

（1）建立客户端的 Socket，确定要连接的服务器的主机名和端口。

（2）发送连接请求到服务器，并等待服务器的回馈信息。

（3）连接成功后，与服务器进行数据的交互。

（4）数据处理完毕后，关闭自身的 Socket 连接。

**（二）进程同步部分**

所谓同步，就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回，同时其它进程/线程也不能调用这个方法。按照这个定义，其实绝大多数函数都是同步调用（例如sin、isdigit等）。但是一般而言，我们在说同步、异步的时候，特指那些需要其他部件协作或者需要一定时间完成的任务。例如Window API函数SendMessage。该函数发送一个消息给某个窗口，在对方处理完消息之前，这个函数不返回。当对方处理完毕以后，该函数才把消息处理函数所返回的 LRESULT 值返回给调用者。

进程/线程间同步机制常有临界区、互斥区、事件、信号量四种方式：

（1）临界区：通过对多进程/线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。在任意时刻只允许一个线程对共享资源进行访问，如果有多个进程/线程试图访问公共资源，那么在有一个进程/线程进入后，其他试图访问公共资源的进程/线程将被挂起，并一直等到进入临界区的进程/线程离开，临界区在被释放后，其他进程/线程才可以抢占。

（2）互斥量：采用互斥对象机制。只有拥有互斥对象的进程/线程才有访问公共资源的权限，因为互斥对象只有一个，所以能保证公共资源不会同时被多个进程/线程访问。互斥不仅能实现同一应用程序的公共资源安全共享，还能实现不同应用程序的公共资源安全共享。

（3）信号量：它允许多个进程/线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大进程/线程数目。信号量对象对进程/线程的同步方式与前面几种方法不同，信号允许多个进程/线程同时使用共享资源，这与操作系统中的 PV 操作相同。它指出了同时访问共享资源的进程/线程最大数目。它允许多个进程/线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大进程/线程数目。

（4）事件：通过通知操作的方式来保持线程的同步，还可以方便实现对多个进程/线程的优先级比较的操作。

在以上四种方式中，通过互斥量可以指定资源被独占的方式使用。互斥量与临界区的作用非常相似，但互斥量是可以命名的，也就是说它可以跨越进程使用。所以创建互斥量需要的资源更多，所以如果只为了在进程内部是用的话使用临界区会带来速度上的优势并能够减少资源占用量。因为互斥量是跨进程的互斥量一旦被创建，就可以通过名字打开它。

互斥量、信号量、事件都可以被跨越进程使用来进行同步数据操作，而其他的对象与数据同步操作无关，但对于进程和线程来讲，如果进程和线程在运行状态则为无信号状态，在退出后为有信号状态。所以可以使用 WaitForSingleObject 来等待进程和线程退出。

四、实验内容

（一）**进程通信**

1．编写程序分别使用无名管道和有名管道实现进程的管道通信。用系统调用pipe()建立一管道，子进程向管道写一句话，父进程从管道中读出来自子进程的信息并显示；使用有名管道实现两个不相关进程间的信息传递。

2．编写C程序，使用Linux中的IPC机制完成“石头、剪刀、布”游戏。

3．共享内存：

1）mmap()系统调用下两个进程通过映射普通文件实现共享内存通信。开启两个终端，创建两个子程序，编译两个程序，两个程序通过命令行参数指定同一个文件来实现共享内存方式的进程间通，其中一个执行写操作，一个执行读操作。

2）mmap()系统调用下父子进程通过匿名映射实现共享内存。

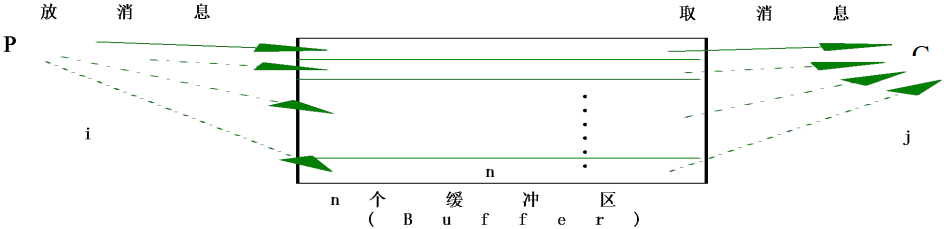
4．修改上述第2个C程序，使之能够在网络上运行。

（二）**进程同步**

1．生产者-消费者问题：

一组生产者向一组消费者提供消息，它们共享一个有界缓冲池，生产者向其中投放消息，消费者从中取得消息。假定这些生产者和消费者互相等效，只要缓冲池未满，生产者可将消息送入缓冲池,只要缓冲池未空，消费者可从缓冲池取走一个消息。

功能要求：根据进程同步机制，编写一个解决上述问题的程序，可显示缓冲池状态、放数据、取数据等过程。



2．读者-写者问题：

有一个写者、很多读者，多个读者可以同时读文件，但写者在写文件时不允许有读者在读文件，同样有读者在读文件时写者也不去能写文件。完成以下功能：

1）读者优先：增加读者优先的规定，当有读者在读文件时，对随后到达的读者和写者，要首先满足读者，阻塞写者。

2）写者优先：增加写者优先的规定，即当有读者和写者同时等待时，首先满足写者。当一个写者声明想写文件时，不允许新的读者再访问文件

3）无优先：不再规定读写的优先权，谁先等待谁就先使用文件。

3．哲学家就餐问题：假定有5位“哲学家”围着一个圆桌。每两位哲学家之间有一把餐叉（一共5把）。哲学家有时要思考一会，不需要餐叉；有时又要就餐。而一位哲学家只有同时拿到了左手边和右手边的两把餐叉，才能吃到东西。

4．理发师问题：有一个理发师，一把理发椅和 N 把供等候理发的顾客坐的椅子。如果没有顾客，则理发师便在理发师椅子上睡觉；当一个顾客到来时，必须唤醒理发师进行理发；如果理发师正在理发时又有顾客来到，则如果有空椅子可坐，就坐下来等待，否则就离开。

五、实验指导

**1、管道**

1)无名管道

#include <unistd.h>

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

int pid; /\*定义进程变量\*/

main(){

int fd[2];

char outpipe[100],inpipe[100]; /\*定义两个字符数组\*/

pipe(fd); /\*创建一个管道\*/

while ((pid=fork())==-1); /\*如果进程创建不成功,则空循环\*/

if(pid==0){ /\*如果子进程创建成功,pid 为进程号\*/

lockf(fd[1],1,0); /\*锁定管道\*/

sprintf(outpipe,"child process is sending message!"); /\*把串放入数组 outpipe 中\*/

write(fd[1],outpipe,50); /\*向管道写长为 50 字节的串\*/

sleep(5); /\*自我阻塞 5 秒\*/

lockf(fd[1],0,0); /\*解除管道的锁定\*/

exit(0);

}

else{

wait(0); /\*等待子进程结束\*/

read(fd[0],inpipe,50); /\*从管道中读长为 50 字节的串\*/

printf("%s\n",inpipe); /\*显示读出的数据\*/

exit(0); /\*父进程结束\*/

}

}

2)有名管道

fifow.c

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

#include <limits.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <stdio.h>

#include <string.h>

int main(){

const char \*fifo\_name = "/home/jenny/myfifo";

int pipe\_fd = -1;

int data\_fd = -1;

int res = 0;

const int open\_mode = O\_WRONLY;

int bytes\_sent = 0;

char buffer[PIPE\_BUF + 1];

int bytes\_read = 0;

if(access(fifo\_name, F\_OK) == -1){

//管道文件不存在

//创建有名管道

printf ("Create the fifo pipe.\n");

res = mkfifo(fifo\_name, 0777);

if(res != 0)

{

fprintf(stderr, "Could not create fifo %s\n", fifo\_name);

exit(EXIT\_FAILURE);

}

}

printf("Process %d opening FIFO O\_WRONLY\n", getpid());

//以只写阻塞方式打开 FIFO 文件，以只读方式打开数据文件

pipe\_fd = open(fifo\_name, open\_mode);

data\_fd = open("Data.txt", O\_RDONLY);

printf("Process %d result %d\n", getpid(), pipe\_fd);

if(pipe\_fd != -1) {

int bytes\_read = 0;

data\_fd = open("Data.txt", O\_RDONLY);

if (data\_fd == -1)

{

close(pipe\_fd);

fprintf (stderr, "Open file[Data.txt] failed\n");

return -1;

}

//向数据文件读取数据

bytes\_read = read(data\_fd, buffer, PIPE\_BUF);

buffer[bytes\_read] = '\0';

while(bytes\_read > 0)

{

//向 FIFO 文件写数据

res = write(pipe\_fd, buffer, bytes\_read);

if(res == -1)

{

fprintf(stderr, "Write error on pipe\n");

exit(EXIT\_FAILURE);

}

//累加写的字节数，并继续读取数据

bytes\_sent += res;

bytes\_read = read(data\_fd, buffer, PIPE\_BUF);

buffer[bytes\_read] = '\0';

}

close(pipe\_fd);

close(data\_fd);

}

else

exit(EXIT\_FAILURE);

printf("Process %d finished\n", getpid());

exit(EXIT\_SUCCESS);

}

fifior.c

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <fcntl.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <limits.h>

#include <string.h>

int main()

{

const char \*fifo\_name = "/home/jenny/myfifo";

int pipe\_fd = -1;

int data\_fd = -1;

int res = 0;

int open\_mode = O\_RDONLY;

char buffer[PIPE\_BUF + 1];

int bytes\_read = 0;

int bytes\_write = 0;

//清空缓冲数组

memset(buffer, '\0', sizeof(buffer));

printf("Process %d opening FIFO O\_RDONLY\n", getpid());

//以只读阻塞方式打开管道文件，注意与 fifowrite.c 文件中的 FIFO 同名

pipe\_fd = open(fifo\_name, open\_mode);

//以只写方式创建保存数据的文件

data\_fd = open("DataFormFIFO.txt", O\_WRONLY|O\_CREAT, 0644);

//printf("Process %d result %d\n",getpid(), pipe\_fd);

if (data\_fd == -1)

{

fprintf(stderr, "Open file[DataFormFIFO.txt] failed\n");

close(pipe\_fd);

return -1;

}

printf("Process %d result %d\n",getpid(), pipe\_fd);

if(pipe\_fd != -1)

{

do

{

//读取 FIFO 中的数据，并把它保存在文件 DataFormFIFO.txt 文件中

res = read(pipe\_fd, buffer, PIPE\_BUF);

bytes\_write = write(data\_fd, buffer, res);

bytes\_read += res;

}while(res > 0);

close(pipe\_fd);

close(data\_fd);

}

else { exit(EXIT\_FAILURE); }

printf("Process %d finished, %d bytes read\n", getpid(), bytes\_read);

exit(EXIT\_SUCCESS);

}

**2、消息队列**

针对实验内容，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。可将“石头”“剪刀”“布”这三招定义为三个整型值，胜负关系为：石头>剪刀>布>石头。

选手进程按照某种策略(如随机产生)出招，然后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。比赛采取多轮(如100轮)定胜负机制，并由裁判宣布最后结果。每次出招由裁判限定时间，超时则判负。

每一轮的胜负结果可以存放在文件或其他数据结构中。比赛结束后，打印每一轮的胜负情况和总的结果。

具体的实验步骤如下。

(1) 设计表示“石头”“剪刀”“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方法。

(4) 根据所选择的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。

下面给出使用System V消息队列IPC完成的示例程序。该程序使用fork() 函数创建了两个选手进程，当前进程为裁判进程。裁判进程创建了两个消息队列，且两个选手进程会发送出拳信息至不同的消息队列，最后由裁判进程从消息队列取得出拳信息并判断结果。

#include <time.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <sys/ipc.h>

#include <sys/msg.h>

struct Game //游戏信息

{

int Round;

long Type;

};

void result\_send(int num) //发送出拳信息

{

struct Game game;

game.Type=1;

game.Round=rand() % 3;

msgsnd(num,&game,sizeof(int),0);

}

int result\_announce(int a,int b) //出拳结果的判断

{

if ((a+1==b)||(a-3==b))

return -1; //a胜b

else if (a==b)

return 0; //ab平局

else

return 1; //a负b

}

void writeFile(int \*result\_list,int len) //将每盘的胜负结果存入文件

{

int count\_A=0;

int count\_B=0;

int pingju=0;

FILE \*fin;

if ((fin=fopen("result.txt", "w") ) ==NULL) //保存结果的文件为result.txt

printf("This file was n't opened");

int i;

for (i=0;i<len; i++)

{

switch(result\_list[i] )

{

case -1: {

count\_A++;

fprintf(fin,"NO.%d:A win\n", i+1) ;

printf("NO.%d:A win\n",i+1);

break;

}

case 0:{

pingju++;

fprintf(fin,"NO. %d:end in a draw\n", i+1) ;

printf("NO. %d:end in a draw\n",i+1);

break;

}

case 1:{

count\_B++;

fprintf(fin,"NO. %d:B win\n", i+1) ;

printf("NO. %d:B win\n", i+1) ;

break;

}

}

}

printf("\nThe final result is A win:%ds \nB win:%ds \nend in a draw %ds\n", count\_A, count\_B, pingju);

fprintf(fin,"\nThe final result is A win:%ds \nB win:%ds \nend in a draw %ds\n", count\_A, count\_B, pingju) ;

fclose(fin);

}

int main()

{

int times;

int key1 = 1234;

int key2 = 5678;

int \*result\_list;

pid\_t pid1, pid2;

int msgid1,msgid2;

msgid1=msgget(key1,IPC\_CREAT | 0666); //创建消息队列1

if (msgid1==-1) //消息队列1创建失败

{

fprintf (stderr,"failed with error") ;

exit (EXIT\_FAILURE) ;

}

msgid2=msgget(key2,IPC\_CREAT | 0666); //创建消息队列2

if(msgid2==-1) //消息队列2创建失败

{

fprintf(stderr,"failed with error") ;

exit(EXIT\_FAILURE) ;

}

printf ("Game start, please input rounds:") ; //输人比赛的轮数

scanf ("%d",&times) ;

result\_list=(int\*) malloc(times\*sizeof(int) ) ;

int i;

for (i=0;i<times;i++)

{

pid1 = fork() ; //创建选手1的进程

if (pid1==0)

{

srand((unsigned) time(0)\*3000); //以时间为种子

result\_send(msgid1) ; //生成选手1的出拳信息并发送到消息队列

exit(-1) ;

}

pid2 = fork() ; //创建选手2的进程

if (pid2==0)

{

srand((unsigned) time(NULL) \*i) ; //以时间为种子

result\_send(msgid2) ; //生成选手2的出拳信息并发送到消息队列

exit(-1) ;

}

if (pid1<0 || pid2<0)

{

fprintf(stderr,"Fork Failed") ;

exit(-1) ;

}

else

{

wait(NULL) ;

wait(NULL) ;

struct Game game1;

struct Game game2; //从消息队列1中取得选手1的出拳信息

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

//从消息队列2中取得选手2的出拳信息

msgrcv(msgid2, &game2, sizeof(game2) -sizeof(long) , 0, 0) ;

//评判出拳结果

int j=result\_announce(game1.Round,game2.Round) ;

//result\_list[i] =result\_announce(game1.Round，game2.Round) ;

result\_list[i] =j;

}

}

//将比赛结果写入文件

writeFile(result\_list, times) ;

//删除消息队列

if (msgctl(msgid1, IPC\_RMID,0)==-1)

{

fprintf(stderr,"msgctl(IPC\_RMID)failed\n") ;

}

if (msgctl(msgid2, IPC\_RMID,0)==-1)

{

fprintf(stderr,"msgctl(IPC\_RMID)failed\n") ;

}

exit(EXIT\_SUCCESS);

}

**3、共享内存：mmap()系统调用**

mmap()系统调用使得进程之间通过映射同一个普通文件实现共享内存。普通文件被映射到进程地址空间后，进程可以向访问普通内存一样对文件进行访问，不必再调用read()、write()等操作。

void\* mmap ( void \* addr , size\_t len , int prot , int flags , int fd , off\_t offset )

参数fd为即将映射到进程空间的文件描述字，一般由open()返回，同时，fd可以指定为-1，此时须指定flags参数中的MAP\_ANON，表明进行的是匿名映射。len是映射到调用进程地址空间的字节数，它从被映射文件开头offset个字节开始算起。prot参数指定共享内存的访问权限。可取如下几个值的或：PROT\_READ（可读），PROT\_WRITE （可写），PROT\_EXEC （可执行），PROT\_NONE（不可访问）。flags 由以下几个常值指定：MAP\_SHARED ，MAP\_PRIVATE，MAP\_FIXED。其中，MAP\_SHARED, MAP\_PRIVATE必选其一，而 MAP\_FIXED则不推荐使用。offset参数一般设为0，表示从文件头开始映射。参数addr指定文件应被映射到进程空间的起始地址，一般被指定一个空指针，此时选择起始地址的任务留给内核来完成。函数的返回值为最后文件映射到进程空间的地址，进程可直接操作起始地址为该值的有效地址。

1）两个进程通过映射普通文件实现共享内存通信

①开启两个终端，分别创建两个子程序：map\_normalfile1.c 及 map\_normalfile2.c。map\_normalfile1 进行写操作及 map\_normalfile2 执行读操作。

其中map\_normalfile1.c首先定义了一个people数据结构，（在这里采用数据结构的方式是因为，共享内存区的数据往往是有固定格式的，这由通信的各个进程决定，采用结构的方式有普遍代表性）。map\_normfile1首先打开或创建一个文件，并把文件的长度设置为5个 people结构大小。然后从mmap()的返回地址开始，设置了10个people结构。然后，进程睡眠 10秒钟，等待其他进程映射同一个文件，最后解除映射。map\_normfile2.c 只是简单的映射一个文件，并以people 数据结构的格式从 mmap()返回的地址处读取 10个 people 结构，并输出读取的值，然后解除映射。

②分别在两个终端上创建一个空文件：we.txt。

③编译两个程序，可执行文件分别为map\_normalfile1及map\_normalfile2。两个程序通过命令行参指定同一个文件来实现共享内存方式的进程间通信。在一个终端上先运行./map\_normalfile1 we.txt，程序输出结果如下：

name: b age 20;

name: c age 21;

name: d age 22;

name: e age 23;

name: f age 24;

name: g age 25;

name: h age 26;

name: i age 27;

name: j age 28;

name: k age 29;

④在 map\_normalfile1 输出 initialize over 之后，输出 umap ok 之前，在另一个终端上运行./map\_normalfile2 ，将会产生另一种输出结果如下：

name: b age 20;

name: c age 21;

name: d age 22;

name: e age 23;

name: f age 24;

name: age 0;

name: age 0;

name: age 0;

name: age 0;

name: age 0;

参考代码：

map\_normalfile1.c:

#include <sys/mman.h>

#include <sys/types.h>

#include <fcntl.h>

#include <unistd.h>

typedef struct{

char name[4];

int age;

}people; //定义一个people结构体

void main(int argc, char\*\* argv) // map a normal file as shared mem:

{

int fd,i;

people \*p\_map;

char temp;

fd=open(argv[1],O\_CREAT|O\_RDWR|O\_TRUNC,00777);

//打开或创建一个文件，大小为5个people结构体的大小

lseek(fd,sizeof(people)\*5-1,SEEK\_SET);

//是一个用于改变读写一个文件时读写指针位置的一个系统调用。

write(fd,"",1);

p\_map = (people\*) mmap( NULL,sizeof(people)\*10,PROT\_READ|PROT\_WRITE,

MAP\_SHARED,fd,0 );

//将文件或者其它对象映射进内存。初始化为10个people数据结构大小

close( fd );

temp = 'a'; // 赋初值

for(i=0; i<10; i++)

{

temp += 1;

memcpy( ( \*(p\_map+i) ).name, &temp,2 ); //内存拷贝函数

( \*(p\_map+i) ).age = 20+i;

}

printf(" initialize over \n ");

sleep(10); //进程睡眠10s，等待其它进程映射同一个文件，最后解除映射。

munmap( p\_map, sizeof(people)\*10 ); //在进程地址空间中解除一个映射关系。

printf( "umap ok \n" );

}

map\_normalfile2.c

typedef struct{

char name[4];

int age;

}people;

void main(int argc, char\*\* argv) // map a normal file as shared mem:

{

int fd,i;

people \*p\_map;

fd=open( argv[1],O\_CREAT|O\_RDWR,00777 );

p\_map = (people\*)mmap(NULL,sizeof(people)\*10,PROT\_READ|PROT\_WRITE,

MAP\_SHARED,fd,0);

for(i = 0;i<10;i++)

{

printf( "name: %s age %d;\n",(\*(p\_map+i)).name, (\*(p\_map+i)).age );

}

munmap( p\_map,sizeof(people)\*10 );

}

2）父子进程通过匿名映射实现共享内存

①开启一个终端，创建一个程序：main.c。在父进程中先调用mmap()，然后调用fork()。那么在调用fork()之后，子进程继承父进程匿名映射后的地址空间，同样也继承mmap()返回的地址，这样，父子进程就可以通过映射区域进行通信了。注意，这里不是一般的继承关系。一般来说，子进程单独维护从父进程继承下来的一些变量。而mmap()返回的地址，却由父子进程共同维护。

②编译这个程序，可执行文件为 main。可得到父子间通信后的程序结果如下：

Child read: the 1 people’s age is 20

Child read: the 1 people’s age is 21

Child read: the 1 people’s age is 22

Child read: the 1 people’s age is 23

Child read: the 1 people’s age is 24

Parent read: the first people’s age is 100

Umap

Umap ok

参考代码：

main.c

#include <sys/mman.h>

#include <sys/types.h>

#include <fcntl.h>

#include <unistd.h>

typedef struct{

char name[4];

int age;

}people;

main(int argc, char\*\* argv)

{

int i;

people \*p\_map;

char temp;

p\_map=(people\*)mmap(NULL,sizeof(people)\*10,PROT\_READ|PROT\_WRITE,

MAP\_SHARED|MAP\_ANONYMOUS,-1,0);

//在父进程中先调用mmap()，然后调用fork()，再调用fork()函数之后，子进程继承父进程匿名映射后的地址空间，同样也继承了mmap()返回的地址。

if(fork() == 0) //创建一个进程

{

sleep(2); //睡眠2s，让出CPU，让其他进程使用。

for(i = 0;i<5;i++)

printf("child read: the %d people's age is %d\n",i+1,(\*(p\_map+i)).age);

(\*p\_map).age = 100;

munmap(p\_map,sizeof(people)\*10);

exit();

}

temp = 'a';

for(i = 0;i<5;i++)

{

temp += 1;

memcpy((\*(p\_map+i)).name, &temp,2);

(\*(p\_map+i)).age=20+i;

}

sleep(5); //父进程睡眠10s，等待子进程继承父进程匿名映射后的地址空间。

printf( "parent read: the first people,s age is %d\n",(\*p\_map).age );

printf("umap\n");

munmap( p\_map,sizeof(people)\*10 );

printf( "umap ok\n" );

}

**4、信号量**

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。由于生产者-消费者问题是经典的进程同步问题，因此对于如何设置信号量在此不再讨论，请参阅主教材中的相关内容加以学习。

下面给出的示例代码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。

#include <semaphore.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#define MAX 256

char \*buffer;

sem\_t empty; //定义同步信号量

sem\_t full; //定义同步信号量

sem\_t mutex; //定义互斥信号量

void \*producer() //生产者

{

sem\_wait(&empty) ; //empty的P操作

sem\_wait(&mutex) ; //mutex的P操作

printf("input something to buffer：") ;

buffer=(char\*) malloc(MAX) ; //给缓冲区分配内存空间

fgets(buffer, MAX, stdin) ; //输入产品至缓冲区

sem\_post(&mutex) ; //mutex的v操作

sem\_post(&full) ; //full的v操作

}

void \* consumer() //消费者

{

sem\_wait (&full) ; //full的P操作

sem\_wait (&mutex) ; //mutex的P操作

printf ("read product from buffer:%s", buffer) ;//从缓冲区中取出产品

memset (buffer,0, MAX) ; //清空缓冲区

sem\_post (&mutex) ; //mutex的V操作

sem\_post (&empty) ; //empty的V操作

}

int main()

{

pthread\_t id\_producer;

pthread\_t id\_consumer;

int ret;

sem\_init(&empty, 0, 10) ; //设置empty的初值为10

sem\_init(&full, 0, 0) ; //设置full的初值为0

sem\_init(&mutex, 0, 1) ; //设置mutex的初值为1

ret=pthread\_create(&id\_producer, NULL, producer, NULL) ;//创建生产者进程

ret=pthread\_create(&id\_consumer, NULL, consumer, NULL) ;//创建消费者进程

pthread\_join(id\_producer, NULL) ; //等待生产者进程结束

pthread\_join(id\_consumer, NULL) ; //等待消费者进程结束

sem\_destroy(&empty) ; //删除信号量

sem\_destroy(&full) ;

sem\_destroy(&mutex) ;

printf("The End...\n") ;

}

**5、socket套接字**

在上述代码框架下，将进程通信机制改为套接字通信机制加以实现。

下面给出另外一个套接字编程示例（java代码）。

1）客户端发送

public class SocketClient {

public static void main(String[] args) {

Socket socket = null;

BufferedReader br = null;

PrintWriter pw = null;

try {

//客户端 socket 指定服务器的地址和端口号

socket = new Socket("127.0.0.1", SocketServer.PORT);

System.out.println("Socket=" + socket);

//同服务器原理一样

br=new BufferedReader(new InputStreamReader(socket.getInputStream()));

pw = new PrintWriter(new BufferedWriter(new OutputStreamWriter(

socket.getOutputStream())));

for (int i = 0; i < 10; i++)

{

pw.println("howdy " + i);

pw.flush(); String str = br.readLine();

System.out.println(str);

}

pw.println("END"); pw.flush();

} catch (Exception e) { e.printStackTrace();

} finally {

try {

System.out.println("close......");

br.close(); pw.close(); socket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

2）服务端接收

public class SocketServer {

public static int PORT = 4331;

public static void main(String[] agrs) {

ServerSocket s = null; Socket socket = null; BufferedReader br = null;

PrintWriter pw = null;

try {

//设定服务端的端口号

s = new ServerSocket(PORT);

System.out.println("ServerSocket Start:"+s);

//等待请求，此方法会一直阻塞,直到获得请求才往下走

socket = s.accept();

System.out.println("Connection accept socket:"+socket);

//用于接收客户端发来的请求

br=new BufferedReader(new InputStreamReader(socket.getInputStream()));

//用于发送返回信息,可以不需要装饰这么多io流使用缓冲流时发送数据要注意调用.flush()方法

pw = new PrintWriter(new BufferedWriter(new OutputStreamWriter(socket.

getOutputStream())), true);

while(true){

String str = br.readLine();

if(str.equals("END")){ break; }

System.out.println("Client Socket Message:"+str);

Thread.sleep(1000);

pw.println("Message Received");

pw.flush();

}

} catch (Exception e) {

e.printStackTrace();

}finally{

System.out.println("Close.....");

try { br.close(); pw.close(); socket.close(); s.close();

} catch (Exception e2) {

}

}

}

}

六、实验过程（85分）

本实验要求学生按照以下操作步骤（含代码、数据、图表等）完成实验，并提交实验结果截图。

（一）**进程通信**（45分）

**1、管道**

1）无名管道

解题思路：

创建一个管道 pipe(fd)，其中 fd 是一个包含两个文件描述符的数组，fd[0] 用于读取数据，fd[1] 用于写入数据。

进行进程的 fork() 操作，创建一个子进程。子进程中会执行写入数据到管道的操作。

在子进程中，使用 lockf() 函数对管道进行锁定，防止其他进程同时写入数据。

在子进程中，使用 sprintf() 函数将要发送的数据写入字符数组 outpipe。

在子进程中，使用 write() 函数将数据写入管道，长度为 50 字节。

在子进程中，使用 sleep() 函数进行自我阻塞，持续 5 秒钟。

在子进程中，使用 lockf() 函数解除管道的锁定。

子进程执行完毕，退出。

在父进程中，使用 wait() 函数等待子进程结束。

在父进程中，使用 read() 函数从管道中读取数据，长度为 50 字节，存入字符数组 inpipe。

在父进程中，使用 printf() 函数显示读取到的数据。

父进程执行完毕，退出。

实验代码：

#include <unistd.h>

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/wait.h>

int pid;                           /\*定义进程变量\*/

int main(){

int fd[2];

char outpipe[100],inpipe[100];   /\*定义两个字符数组\*/

pipe(fd);                     /\*创建一个管道\*/

while ((pid=fork())==-1);       /\*如果进程创建不成功,则空循环\*/

if(pid==0){                  /\*如果子进程创建成功,pid 为进程号\*/

lockf(fd[1],1,0);          /\*锁定管道\*/

sprintf(outpipe,"child process is sending message!"); /\*把串放入数组 outpipe 中\*/

write(fd[1],outpipe,50);    /\*向管道写长为 50 字节的串\*/

sleep(5);                /\*自我阻塞 5 秒\*/

lockf(fd[1],0,0);          /\*解除管道的锁定\*/

exit(0);

}

else{

wait(0);                /\*等待子进程结束\*/

read(fd[0],inpipe,50);     /\*从管道中读长为 50 字节的串\*/

printf("%s\n",inpipe);     /\*显示读出的数据\*/

exit(0);                /\*父进程结束\*/

}

}

实验结果：

]U]R6691_`3YKL@JZPJRJMD

2）有名管道

解题思路：

首先，通过调用 access() 函数检查有名管道文件是否存在，如果不存在，则通过调用 mkfifo() 函数创建一个权限为 0777 的有名管道文件。

然后，通过调用 open() 函数以只写阻塞方式打开有名管道文件，获取返回的文件描述符 pipe\_fd。

接着，通过调用 open() 函数以只读方式打开数据文件 "Data.txt"，获取返回的文件描述符 data\_fd。

使用 read() 函数从数据文件中读取数据，将读取的数据存储在缓冲区 buffer 中，并将读取的字节数存储在变量 bytes\_read 中。

使用 write() 函数将缓冲区中的数据写入到有名管道文件中，将写入的字节数存储在变量 res 中，重复这一步骤直到数据文件中的数据全部写入到有名管道中。

最后，关闭有名管道文件和数据文件的文件描述符，并输出进程的完成信息，然后通过调用 exit() 函数退出进程。

实验代码：

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

#include <limits.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <stdio.h>

#include <string.h>

int main(){

const char \*fifo\_name = "/home/jenny/myfifo";

int pipe\_fd = -1;

int data\_fd = -1;

int res = 0;

const int open\_mode = O\_WRONLY;

int bytes\_sent = 0;

char buffer[PIPE\_BUF + 1];

int bytes\_read = 0;

if(access(fifo\_name, F\_OK) == -1){

//管道文件不存在

//创建有名管道

printf ("Create the fifo pipe.\n");

res = mkfifo(fifo\_name, 0777);

if(res != 0)

{

fprintf(stderr, "Could not create fifo %s\n", fifo\_name);

exit(EXIT\_FAILURE);

}

}

printf("Process %d opening FIFO O\_WRONLY\n", getpid());

//以只写阻塞方式打开 FIFO 文件，以只读方式打开数据文件

pipe\_fd = open(fifo\_name, open\_mode);

data\_fd = open("Data.txt", O\_RDONLY);

printf("Process %d result %d\n", getpid(), pipe\_fd);

if(pipe\_fd != -1) {

int bytes\_read = 0;

data\_fd = open("Data.txt", O\_RDONLY);

if (data\_fd == -1)

{

close(pipe\_fd);

fprintf (stderr, "Open file[Data.txt] failed\n");

return -1;

}

//向数据文件读取数据

bytes\_read = read(data\_fd, buffer, PIPE\_BUF);

buffer[bytes\_read] = '\0';

while(bytes\_read > 0)

{

//向 FIFO 文件写数据

res = write(pipe\_fd, buffer, bytes\_read);

if(res == -1)

{

fprintf(stderr, "Write error on pipe\n");

exit(EXIT\_FAILURE);

}

//累加写的字节数，并继续读取数据

bytes\_sent += res;

bytes\_read = read(data\_fd, buffer, PIPE\_BUF);

buffer[bytes\_read] = '\0';

}

close(pipe\_fd);

close(data\_fd);

}

else

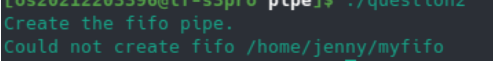
exit(EXIT\_FAILURE);

printf("Process %d finished\n", getpid());

exit(EXIT\_SUCCESS);

}

实验结果：



**2、消息队列**

解题思路：

程序通过 fork() 函数创建两个子进程，分别代表两名选手。每个子进程使用 srand() 函数根据时间生成随机数种子，然后调用 result\_send() 函数生成选手的出拳信息，并发送到消息队列中。父进程使用 msgrcv() 函数从消息队列中接收选手的出拳信息，并调用 result\_announce() 函数评判出拳结果，将结果保存到 result\_list 数组中。循环进行指定的轮数，最后调用 writeFile() 函数将比赛结果写入文件，并通过 msgctl() 函数删除消息队列。

实验代码：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <time.h>

struct Game //游戏信息

{

int Round;

long Type;

};

void result\_send(int num) //发送出拳信息

{

struct Game game;

game.Type=1;

game.Round=rand() % 3;

msgsnd(num,&game,sizeof(struct Game)-sizeof(long),0); // 修正消息发送的大小

}

int result\_announce(int a,int b) //出拳结果的判断

{

if ((a+1==b)||(a-2==b)) // 修正判断条件

return -1; //a胜b

else if (a==b)

return 0; //ab平局

else

return 1; //a负b

}

void writeFile(int \*result\_list,int len) //将每盘的胜负结果存入文件

{

int count\_A=0;

int count\_B=0;

int pingju=0;

FILE \*fin;

if ((fin=fopen("result.txt", "w") ) ==NULL) //保存结果的文件为result.txt

printf("This file was n't opened");

int i;

for (i=0;i<len; i++)

{

switch(result\_list[i] )

{

case -1: {

count\_A++;

fprintf(fin,"NO.%d:A win\n", i+1) ;

printf("NO.%d:A win\n",i+1);

break;

}

case 0:{

pingju++;

fprintf(fin,"NO. %d:end in a draw\n", i+1) ;

printf("NO. %d:end in a draw\n",i+1);

break;

}

case 1:{

count\_B++;

fprintf(fin,"NO. %d:B win\n", i+1) ;

printf("NO. %d:B win\n", i+1) ;

break;

}

}

}

printf("\nThe final result is A win:%ds \nB win:%ds \nend in a draw %ds\n", count\_A, count\_B, pingju);

fprintf(fin,"\nThe final result is A win:%ds \nB win:%ds \nend in a draw %ds\n", count\_A, count\_B, pingju) ;

fclose(fin);

}

int main()

{

int times;

int key1 = 1234;

int key2 = 5678;

int \*result\_list;

pid\_t pid1, pid2;

int msgid1, msgid2;

msgid1 = msgget(key1, IPC\_CREAT | 0666); //创建消息队列1

if (msgid1 == -1) //消息队列1创建失败

{

fprintf(stderr, "Failed to create message queue 1");

exit(EXIT\_FAILURE);

}

msgid2 = msgget(key2, IPC\_CREAT | 0666); //创建消息队列2

if (msgid2 == -1) //消息队列2创建失败

{

fprintf(stderr, "Failed to create message queue 2");

exit(EXIT\_FAILURE);

}

printf("Game start, please input rounds:"); //输入比赛的轮数

scanf("%d", &times);

result\_list = (int \*)malloc(times \* sizeof(int));

int i;

for (i = 0; i < times; i++)

{

pid1 = fork(); //创建选手1的进程

if (pid1 == 0)

{

srand((unsigned)time(0) \* 3000); //以时间为种子

result\_send(msgid1); //生成选手1的出拳信息并发送到消息队列

exit(-1);

}

pid2 = fork(); //创建选手2的进程

if (pid2 == 0)

{

srand((unsigned)time(NULL) \* i); //以时间为种子

result\_send(msgid2); //生成选手2的出拳信息并发送到消息队列

exit(-1);

}

if (pid1 < 0 || pid2 < 0)

{

fprintf(stderr, "Fork Failed");

exit(-1);

}

else

{

wait(NULL);

wait(NULL);

struct Game game1;

struct Game game2; //从消息队列1中取得选手1的出拳信息

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

//从消息队列2中取得选手2的出拳信息

msgrcv(msgid2, &game2, sizeof(game2) - sizeof(long), 0, 0);

//评判出拳结果

int j = result\_announce(game1.Round, game2.Round);

result\_list[i] = j;

}

}

//将比赛结果写入文件

writeFile(result\_list, times);

//删除消息队列

if (msgctl(msgid1, IPC\_RMID, 0) == -1)

{

fprintf(stderr, "Failed to delete message queue 1");

}

if (msgctl(msgid2, IPC\_RMID, 0) == -1)

{

fprintf(stderr, "Failed to delete message queue 2");

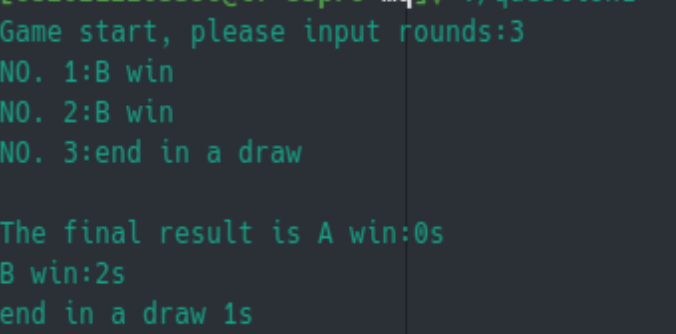
}

free(result\_list); //释放动态分配的内存

return 0;

}

实验结果：



**3、共享内存**

1）两个进程通过映射普通文件实现共享内存通信

解题思路：

使用 open 函数创建一个文件，设置文件的打开模式为创建、读写和截断，权限为 00777。

使用 lseek 函数将文件读写指针位置移动到文件末尾前一个 people 结构体大小的位置。

使用 write 函数向文件写入一个字节的数据，使文件大小达到 5 个 people 结构体的大小。

使用 mmap 函数将文件映射到内存中，映射的大小为 10 个 people 数据结构大小，设置映射的保护标志为可读可写。

关闭文件描述符。

使用循环为内存中映射的 people 结构体数组赋初值，包括设置 name 字段为递增的字母，并设置 age 字段为递增的整数。

使用 sleep 函数让进程睡眠 10 秒，等待其他进程映射同一个文件。

使用 munmap 函数在进程地址空间中解除对文件的映射关系。

输出提示信息表示解除映射完成。

返回 0 表示程序正常结束。

实验代码：

#include <sys/mman.h>

#include <sys/types.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

#include <string.h>

typedef struct {

char name[4];

int age;

} people; // 定义一个people结构体

int main(int argc, char\*\* argv) {

int fd, i;

people\* p\_map;

short temp; // 将temp变量的类型改为short，大小为2字节

fd = open(argv[1], O\_CREAT | O\_RDWR | O\_TRUNC, 00777); // 打开或创建一个文件，大小为5个people结构体的大小

lseek(fd, sizeof(people) \* 5 - 1, SEEK\_SET); // 改变文件读写指针位置

write(fd, "", 1); // 写入一个字节作为文件内容，使文件大小达到5个people结构体的大小

p\_map = (people\*)mmap(NULL, sizeof(people) \* 10, PROT\_READ | PROT\_WRITE,

MAP\_SHARED, fd, 0); // 将文件映射到内存，初始化为10个people数据结构大小

if (p\_map == MAP\_FAILED) {

perror("mmap");

return -1;

}

close(fd);

temp = 'a'; // 赋初值

for (i = 0; i < 10; i++) {

temp += 1;

memcpy((\*(p\_map + i)).name, &temp, 1); // 使用1作为拷贝大小

(\*(p\_map + i)).age = 20 + i;

}

printf("initialize over\n");

sleep(10); // 进程睡眠10s，等待其它进程映射同一个文件

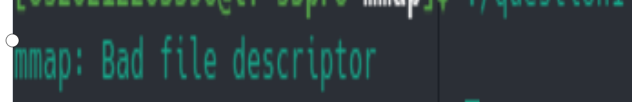
munmap(p\_map, sizeof(people) \* 10); // 在进程地址空间中解除一个映射关系

printf("umap ok\n");

return 0;

}

实验结果：



2）父子进程通过匿名映射实现共享内存

解题思路：

定义了一个结构体 people，包含一个名字（4个字符）和一个年龄（整数）。

在 main 函数中，首先调用 mmap 函数创建了一个共享内存区域，并将其映射到了 p\_map 指针指向的内存空间中，大小为 sizeof(people) \* 10 字节，权限为可读可写，标志为 MAP\_SHARED | MAP\_ANONYMOUS。

然后通过调用 fork 函数创建了一个子进程。在子进程中，先让子进程睡眠 2 秒，然后通过循环输出共享内存中的数据，并修改了第一个人的年龄为 100，最后调用 munmap 函数解除了共享内存的映射，并调用 exit 函数退出子进程。

在父进程中，通过循环给共享内存中的每个人的名字赋值，并设置对应的年龄，然后让父进程睡眠 5 秒。

最后，父进程输出共享内存中第一个人的年龄，并调用 munmap 函数解除了共享内存的映射，然后返回 0 退出程序。

实验代码：

#include <sys/mman.h>

#include <sys/types.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h> // 添加 stdlib.h 头文件以解决 exit() 隐式声明警告

typedef struct {

char name[4];

int age;

} people;

int main(int argc, char\*\* argv) {

int i;

people\* p\_map;

char temp;

p\_map = (people\*)mmap(NULL, sizeof(people) \* 10, PROT\_READ | PROT\_WRITE,

MAP\_SHARED | MAP\_ANONYMOUS, -1, 0);

if (fork() == 0) {

sleep(2);

for (i = 0; i < 5; i++)

printf("child read: the %d people's age is %d\n", i + 1, (\*(p\_map + i)).age);

(\*p\_map).age = 100;

munmap(p\_map, sizeof(people) \* 10);

exit(0);

}

temp = 'a';

for (i = 0; i < 5; i++) {

temp += 1;

memcpy((\*(p\_map + i)).name, &temp, 1); // 修复 memcpy() 函数读取 temp 变量时的内存越界访问警告

(\*(p\_map + i)).age = 20 + i;

}

sleep(5);

printf("parent read: the first people's age is %d\n", (\*p\_map).age);

printf("umap\n");

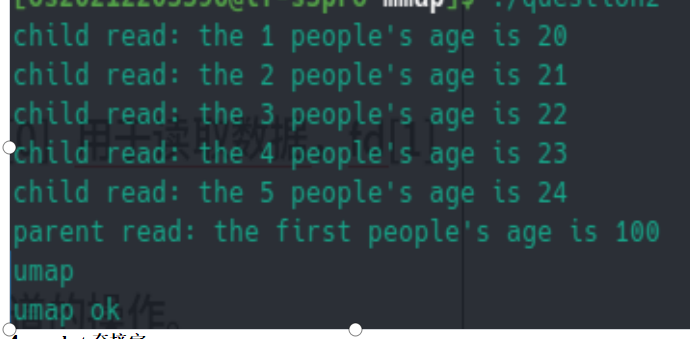
munmap(p\_map, sizeof(people) \* 10);

printf("umap ok\n");

return 0;

}

实验结果：



**4、socket套接字**

解题思路：

创建一个 Socket 对象，指定服务器的 IP 地址和端口号。在示例中，服务器的 IP 地址为 "127.0.0.1"，端口号使用了一个名为 SocketServer.PORT 的变量，表示服务器的端口号。

创建一个 BufferedReader 对象，用于从服务器接收数据。这里通过 socket.getInputStream() 获取到输入流，并将其包装成 BufferedReader 对象，以便按行读取数据。

创建一个 PrintWriter 对象，用于向服务器发送数据。这里通过 socket.getOutputStream() 获取到输出流，并将其包装成 PrintWriter 对象，以便写入数据。

使用 for 循环发送多次数据给服务器。在示例中，循环执行了10次，每次发送一个包含 "howdy" 和当前循环计数的字符串给服务器，并通过 pw.flush() 刷新输出流，确保数据被发送出去。

在每次发送数据后，通过 br.readLine() 从服务器接收一行数据，并将其打印到控制台上。

循环结束后，通过 pw.println("END") 发送一个结束标识给服务器，并通过 pw.flush() 刷新输出流。

最后，使用 finally 块关闭 BufferedReader、PrintWriter 和 Socket 对象，释放资源。

实验代码：

package com.lf;

import java.io.\*;

import java.io.BufferedWriter;

import java.io.PrintWriter;

import java.net.Socket;

public class SocketClient {

public static void main(String[] args) {

Socket socket = null;

BufferedReader br = null;

PrintWriter pw = null;

try {

// 客户端 socket 指定服务器的地址和端口号

socket = new Socket("127.0.0.1", SocketServer.PORT);

System.out.println("Socket=" + socket);

// 同服务器原理一样

br = new BufferedReader(new InputStreamReader(socket.getInputStream()));

pw = new PrintWriter(new BufferedWriter(new OutputStreamWriter(socket.getOutputStream())));

for (int i = 0; i < 10; i++) {

pw.println("howdy " + i);

pw.flush();

String str = br.readLine();

System.out.println(str);

}

pw.println("END");

pw.flush();

} catch (Exception e) {

e.printStackTrace();

} finally {

try {

System.out.println("close......");

if (br != null) br.close();

if (pw != null) pw.close();

if (socket != null) socket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

package com.lf;

import java.net.\*;

import java.io.\*;

public class SocketServer {

public static int PORT = 4331;

public static void main(String[] args) {

ServerSocket s = null; Socket socket = null; BufferedReader br = null;

PrintWriter pw = null;

try {

//设定服务端的端口号

s = new ServerSocket(PORT);

System.out.println("ServerSocket Start:"+s);

//等待请求，此方法会一直阻塞,直到获得请求才往下走

socket = s.accept();

System.out.println("Connection accept socket:"+socket);

//用于接收客户端发来的请求

br=new BufferedReader(new InputStreamReader(socket.getInputStream()));

//用于发送返回信息,可以不需要装饰这么多io流使用缓冲流时发送数据要注意调用.flush()方法

pw = new PrintWriter(new BufferedWriter(new OutputStreamWriter(socket.

getOutputStream())), true);

while(true){

String str = br.readLine();

if(str.equals("END")){ break; }

System.out.println("Client Socket Message:"+str);

Thread.sleep(1000);

pw.println("Message Received");

pw.flush();

}

} catch (Exception e) {

e.printStackTrace();

}finally{

System.out.println("Close.....");

try { br.close(); pw.close(); socket.close(); s.close();

} catch (Exception e2) {

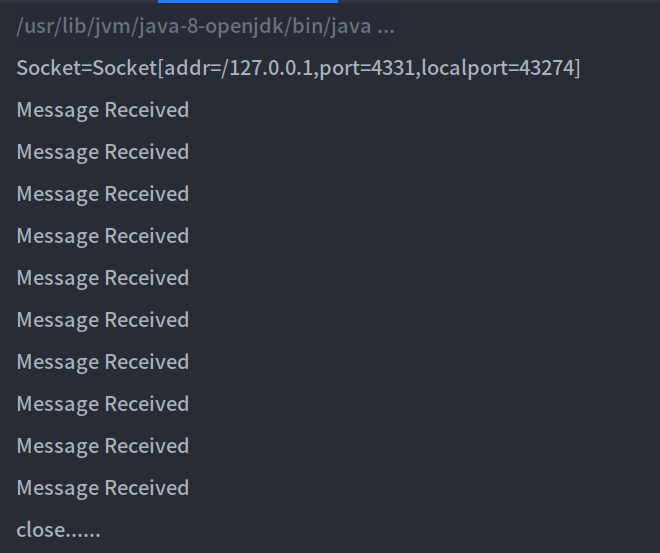
}

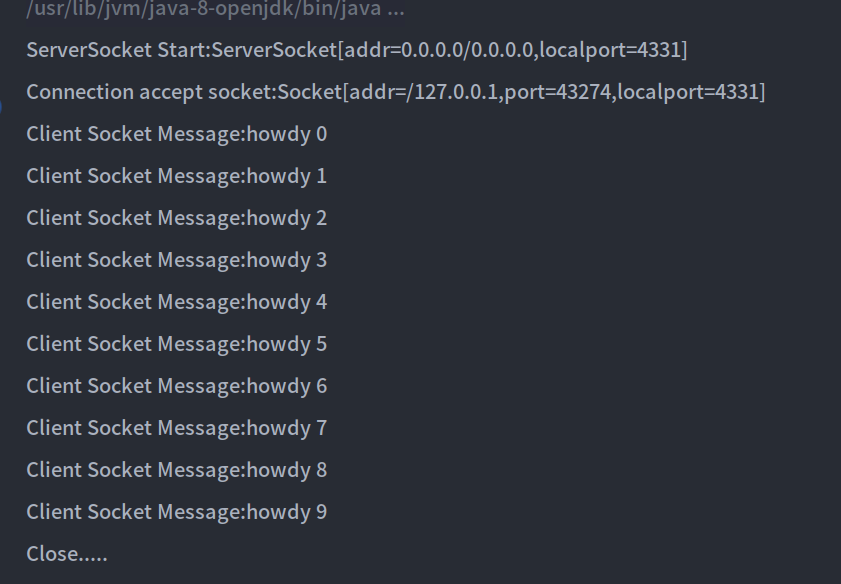
}

}

}

实验结果：





（二）**进程同步**（40分）

1．生产者-消费者问题

实验步骤如下：

1）初始化一个货架，并存货 100；最大容量 100；取货每次取 60，放货每次放 40；小于 60不能取，大于 100不能放。

2）创建两个进程，分别实现存、取操作。

3）两个进程并发执行各 5 次。

4）查看执行结果，并分析原因。

解题思路：

使用一个共享变量 shelf 来表示货架上的存货数量，初始化为 100，表示货架上有 100 个存货，最大容量为 100。

使用互斥锁 mutex 来保护对共享变量 shelf 的访问，以确保线程间的互斥访问。

使用条件变量 cond\_producer 和 cond\_consumer 分别来进行生产者和消费者线程之间的通信。生产者线程在货架满时等待 cond\_producer，消费者线程在货架空时等待 cond\_consumer。

生产者线程函数 producer 中，生产者线程通过获取互斥锁 mutex，检查货架是否已满。如果货架已满，则通过条件变量 cond\_producer 进行等待，直到有空位。一旦有空位，生产者线程生产 40 个商品并放入货架，然后通过条件变量 cond\_consumer 发送信号给消费者线程，最后释放互斥锁 mutex。

消费者线程函数 consumer 中，消费者线程通过获取互斥锁 mutex，检查货架是否为空。如果货架为空，则通过条件变量 cond\_consumer 进行等待，直到有商品可供消费。一旦有商品可供消费，消费者线程从货架上取走 60 个商品，然后通过条件变量 cond\_producer 发送信号给生产者线程，最后释放互斥锁 mutex。

在 main 函数中，创建一个生产者线程和一个消费者线程，并通过 pthread\_create 函数启动它们。然后使用 pthread\_join 函数等待线程结束。最后销毁互斥锁和条件变量。

实验代码：

#include <stdio.h>

#include <pthread.h>

#define MAX\_CAPACITY 100

#define MAX\_PRODUCER\_OPERATIONS 5

#define MAX\_CONSUMER\_OPERATIONS 5

int shelf = 100; // 货架上的存货数量

pthread\_mutex\_t mutex; // 互斥锁

pthread\_cond\_t cond\_producer, cond\_consumer; // 条件变量

int producer\_count = 0, consumer\_count = 0; // 生产者和消费者操作计数

// 生产者线程函数

void\* producer(void\* arg) {

int id = \*(int\*)arg;

int operations = 0;

while (operations < MAX\_PRODUCER\_OPERATIONS) {

pthread\_mutex\_lock(&mutex); // 加锁

// 如果货架已满，等待条件变量 cond\_producer

while (shelf >= MAX\_CAPACITY) {

printf("Producer %d: Shelf is full, waiting...\n", id);

pthread\_cond\_wait(&cond\_producer, &mutex);

}

// 生产并放入货架

shelf += 40;

printf("Producer %d: Produced 40 items, current shelf: %d\n", id, shelf);

pthread\_cond\_signal(&cond\_consumer); // 发送信号给消费者线程

pthread\_mutex\_unlock(&mutex); // 解锁

operations++;

}

return NULL;

}

// 消费者线程函数

void\* consumer(void\* arg) {

int id = \*(int\*)arg;

int operations = 0;

while (operations < MAX\_CONSUMER\_OPERATIONS) {

pthread\_mutex\_lock(&mutex); // 加锁

// 如果货架为空，等待条件变量 cond\_consumer

while (shelf <= 0) {

printf("Consumer %d: Shelf is empty, waiting...\n", id);

pthread\_cond\_wait(&cond\_consumer, &mutex);

}

// 取货并从货架上移除

shelf -= 60;

printf("Consumer %d: Consumed 60 items, current shelf: %d\n", id, shelf);

pthread\_cond\_signal(&cond\_producer); // 发送信号给生产者线程

pthread\_mutex\_unlock(&mutex); // 解锁

operations++;

}

return NULL;

}

int main() {

pthread\_t producer\_thread, consumer\_thread;

int producer\_id = 1, consumer\_id = 1;

pthread\_mutex\_init(&mutex, NULL);

pthread\_cond\_init(&cond\_producer, NULL);

pthread\_cond\_init(&cond\_consumer, NULL);

// 创建生产者和消费者线程

pthread\_create(&producer\_thread, NULL, producer, &producer\_id);

pthread\_create(&consumer\_thread, NULL, consumer, &consumer\_id);

// 等待线程结束

pthread\_join(producer\_thread, NULL);

pthread\_join(consumer\_thread, NULL);

pthread\_mutex\_destroy(&mutex);

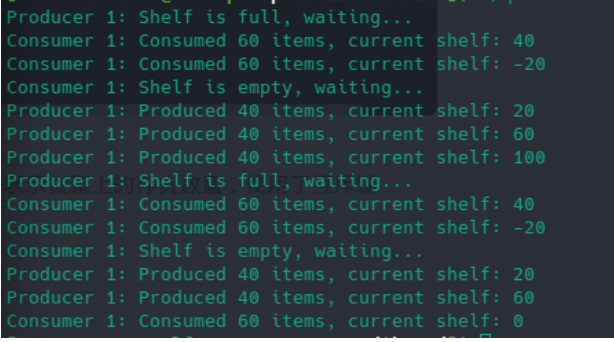
pthread\_cond\_destroy(&cond\_producer);

pthread\_cond\_destroy(&cond\_consumer);

return 0;

}

实验结果：



2．读者-写者问题

算法分析：

无限循环中运行，不断地读取或写入共享资源，并在完成后休眠一段时间。

读者线程的主要逻辑如下：

获取互斥锁 mutex。

检查是否有写者正在写入，如果有，则等待写者完成，即调用 pthread\_cond\_wait(&cond\_reader, &mutex)。

如果没有写者在写，则增加读者计数 readers。

释放互斥锁 mutex。

读取共享资源 data。

再次获取互斥锁 mutex。

减少读者计数 readers。

如果没有读者了，则唤醒所有写者，即调用 pthread\_cond\_broadcast(&cond\_writer)。

释放互斥锁 mutex。

休眠一段时间。

写者线程的主要逻辑如下：

获取互斥锁 mutex。

检查是否有其他读者在读或者其他写者在写，如果有，则等待，即调用 pthread\_cond\_wait(&cond\_writer, &mutex)。

如果没有读者在读或者写者在写，则增加写者计数 writers。

释放互斥锁 mutex。

写入共享资源 data。

再次获取互斥锁 mutex。

减少写者计数 writers。

唤醒所有读者，即调用 pthread\_cond\_broadcast(&cond\_reader)。

唤醒一个写者，即调用 pthread\_cond\_signal(&cond\_writer)。

释放互斥锁 mutex。

休眠一段时间。

实验代码：

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#define MAX\_READERS 5

#define MAX\_WRITERS 2

pthread\_mutex\_t mutex;          // 互斥锁，用于对共享资源的访问控制

pthread\_cond\_t cond\_reader;     // 读者条件变量，用于读者线程之间的同步

pthread\_cond\_t cond\_writer;     // 写者条件变量，用于写者线程之间的同步

int readers = 0;               // 当前读者数量

int writers = 0;               // 当前写者数量

int data = 0;                  // 共享资源

void \*reader(void \*arg) {

   int id = \*((int \*)arg);

   while (1) {

       // 进入读临界区前先获取互斥锁

       pthread\_mutex\_lock(&mutex);

       while (writers > 0) {

           // 如果有写者在写，则等待写者完成

           pthread\_cond\_wait(&cond\_reader, &mutex);

       }

       readers++;

       pthread\_mutex\_unlock(&mutex);

       // 读取共享资源

       printf("Reader %d reads data: %d\n", id, data);

       // 离开读临界区前获取互斥锁

       pthread\_mutex\_lock(&mutex);

       readers--;

       if (readers == 0) {

           // 如果没有读者了，则唤醒所有写者

           pthread\_cond\_broadcast(&cond\_writer);

       }

       pthread\_mutex\_unlock(&mutex);

       // 休眠一段时间

       sleep(1);

   }

   return NULL;

}

void \*writer(void \*arg) {

   int id = \*((int \*)arg);

   while (1) {

       // 进入写临界区前先获取互斥锁

       pthread\_mutex\_lock(&mutex);

       while (readers > 0 || writers > 0) {

           // 如果有读者在读或者有其他写者在写，则等待

           pthread\_cond\_wait(&cond\_writer, &mutex);

       }

       writers++;

       pthread\_mutex\_unlock(&mutex);

       // 写入共享资源

       data++;

       printf("Writer %d writes data: %d\n", id, data);

       // 离开写临界区前获取互斥锁

       pthread\_mutex\_lock(&mutex);

       writers--;

       pthread\_cond\_broadcast(&cond\_reader); // 唤醒所有读者

       pthread\_cond\_signal(&cond\_writer);     // 唤醒一个写者

       pthread\_mutex\_unlock(&mutex);

       // 休眠一段时间

       sleep(1);

   }

   return NULL;

}

int main() {

   pthread\_t readers[MAX\_READERS];

   pthread\_t writers[MAX\_WRITERS];

   int reader\_ids[MAX\_READERS];

   int writer\_ids[MAX\_WRITERS];

   // 初始化互斥锁和条件变量

   pthread\_mutex\_init(&mutex, NULL);

   pthread\_cond\_init(&cond\_reader, NULL);

   pthread\_cond\_init(&cond\_writer, NULL);

   // 创建读者线程

   for (int i = 0; i < MAX\_READERS; i++) {

       reader\_ids[i] = i + 1;

       pthread\_create(&readers[i], NULL, reader, &reader\_ids[i]);

   }

   // 创建写者线程

   for (int i = 0; i < MAX\_WRITERS; i++) {

       writer\_ids[i] = i + 1;

       pthread\_create(&writers[i], NULL, writer, &writer\_ids[i]);

   }

   // 等待所有读者线程和写者线程结束

   for (int i = 0; i < MAX\_READERS; i++) {

       pthread\_join(readers[i], NULL);

   }

   for (int i = 0; i < MAX\_WRITERS; i++) {

       pthread\_join(writers[i], NULL);

   }

   // 销毁互斥锁和条件变量

   pthread\_mutex\_destroy(&mutex);

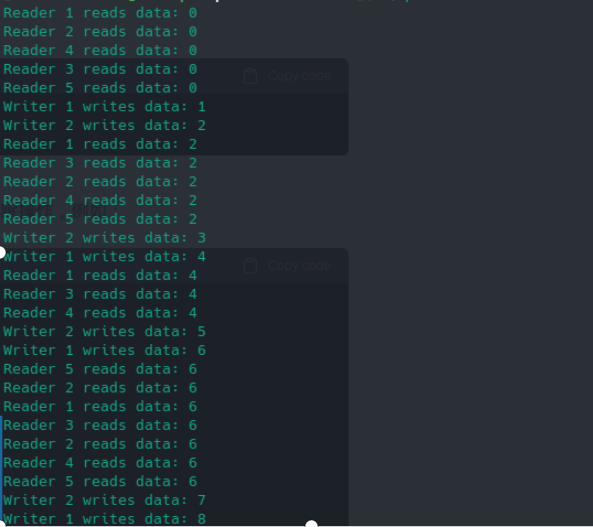
   pthread\_cond\_destroy(&cond\_reader);

   pthread\_cond\_destroy(&cond\_writer);

   return 0;

}

实验结果：



3．哲学家就餐问题

解题思路：

在主函数 main() 中，首先定义了一些全局变量和互斥锁。然后，通过一个循环初始化了 NUM\_PHILOSOPHERS 个叉子的互斥锁。接着，再次通过循环创建了 NUM\_PHILOSOPHERS 个哲学家的线程，并将每个哲学家的 ID 作为参数传递给哲学家线程函数 philosopher()。然后，通过另一个循环等待所有哲学家线程的结束。最后，再次通过循环销毁了所有叉子的互斥锁。

在哲学家线程函数 philosopher() 中，首先从参数中获取哲学家的 ID、左边叉子的编号和右边叉子的编号。然后，通过一个无限循环来表示哲学家的持续行为。在循环中，哲学家会先进行思考（输出相应的信息并休眠1秒），然后尝试拿起左边和右边的叉子（使用互斥锁保证同一时刻只有一个哲学家能够拿起叉子），接着进餐（输出相应的信息并休眠1秒），最后放下左边和右边的叉子（释放互斥锁）。这样，哲学家们可以交替地进行思考和进餐，避免了死锁的发生。

整体思路是通过互斥锁来保护共享资源（叉子），以避免竞态条件和死锁的发生，从而实现了哲学家的安全就餐行为

实验代码：

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#define NUM\_PHILOSOPHERS 5

pthread\_mutex\_t forks[NUM\_PHILOSOPHERS]; // 叉子的互斥锁

pthread\_t philosophers[NUM\_PHILOSOPHERS]; // 哲学家的线程

void \*philosopher(void \*arg) {

int id = \*((int \*)arg);

int left\_fork = id; // 左边叉子的编号

int right\_fork = (id + 1) % NUM\_PHILOSOPHERS; // 右边叉子的编号

while (1) {

// 思考

printf("Philosopher %d is thinking.\n", id);

sleep(1);

// 拿起左边的叉子

pthread\_mutex\_lock(&forks[left\_fork]);

printf("Philosopher %d picks up fork %d.\n", id, left\_fork);

// 拿起右边的叉子

pthread\_mutex\_lock(&forks[right\_fork]);

printf("Philosopher %d picks up fork %d.\n", id, right\_fork);

// 进餐

printf("Philosopher %d is eating.\n", id);

sleep(1);

// 放下左边的叉子

pthread\_mutex\_unlock(&forks[left\_fork]);

printf("Philosopher %d puts down fork %d.\n", id, left\_fork);

// 放下右边的叉子

pthread\_mutex\_unlock(&forks[right\_fork]);

printf("Philosopher %d puts down fork %d.\n", id, right\_fork);

}

return NULL;

}

int main() {

int i;

int ids[NUM\_PHILOSOPHERS];

// 初始化叉子的互斥锁

for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_mutex\_init(&forks[i], NULL);

}

// 创建哲学家的线程

for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

ids[i] = i;

pthread\_create(&philosophers[i], NULL, philosopher, &ids[i]);

}

// 等待线程结束

for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

pthread\_join(philosophers[i], NULL);

}

// 销毁互斥锁

for (i = 0; i < NUM\_PHILOSOPHERS; i++) {

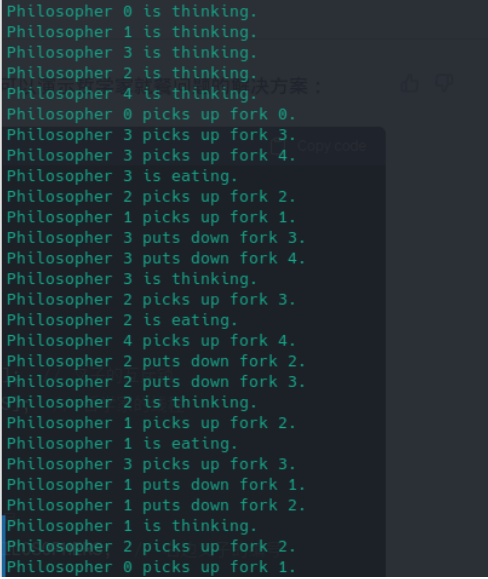
pthread\_mutex\_destroy(&forks[i]);

}

return 0;

}

实验结果：



4．理发师问题

解题思路：

定义了全局变量 num\_chairs 表示椅子上的顾客数量，num\_customers 表示已经服务的顾客数量，以及互斥锁 mutex 和条件变量 cond\_barber、cond\_customer 用于线程间同步。

理发师线程 barber 函数通过使用互斥锁和条件变量进行同步，实现了以下逻辑：

当没有顾客时，理发师线程会进入休眠状态，等待条件变量 cond\_barber 的信号。

当有顾客时，理发师会从椅子上取走一个顾客，进行理发（这里使用 sleep(1) 模拟理发过程）。

然后，理发师会增加已经服务的顾客数量，并通过条件变量 cond\_customer 通知等待的顾客线程。

顾客线程 customer 函数通过使用互斥锁和条件变量进行同步，实现了以下逻辑：

如果有空椅子，顾客线程会占用一个椅子，并通知理发师线程。

如果没有空椅子，顾客线程会离开理发店。

然后，顾客线程会等待已经服务的顾客数量达到自己的编号，表示理发完成，并离开理发店。

在 main 函数中，首先初始化互斥锁和条件变量，并创建理发师线程和足够多的顾客线程（这里使用 MAX\_CHAIRS \* 2 个顾客线程进行模拟）。然后，通过 pthread\_join 函数等待所有的顾客线程结束，最后释放互斥锁和条件变量，并退出程序。

实验代码：

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#define MAX\_CHAIRS 5 // 椅子的最大数量

pthread\_mutex\_t mutex; // 互斥锁，保护共享资源

pthread\_cond\_t cond\_barber; // 理发师的条件变量

pthread\_cond\_t cond\_customer; // 顾客的条件变量

int num\_chairs = 0; // 当前椅子上的顾客数量

int num\_customers = 0; // 已经服务的顾客数量

void \*barber(void \*arg) {

   while (1) {

       pthread\_mutex\_lock(&mutex);

       while (num\_chairs == 0) { // 如果没有顾客，理发师等待

           printf("Barber is sleeping...\n");

           pthread\_cond\_wait(&cond\_barber, &mutex);

       }

       num\_chairs--; // 取一个顾客的位置

       printf("Barber is cutting hair for a customer.\n");

       sleep(1); // 模拟理发

       num\_customers++; // 已经服务的顾客数量增加

       printf("Barber finishes cutting hair for a customer.\n");

       pthread\_cond\_signal(&cond\_customer); // 通知等待的顾客

       pthread\_mutex\_unlock(&mutex);

   }

   return NULL;

}

void \*customer(void \*arg) {

   int id = \*((int \*)arg);

   pthread\_mutex\_lock(&mutex);

   if (num\_chairs < MAX\_CHAIRS) { // 如果有空椅子，顾客进入店内

       num\_chairs++; // 顾客占用一个椅子

       printf("Customer %d sits on a chair.\n", id);

       if (num\_chairs == 1) { // 如果是第一个顾客，通知理发师

           pthread\_cond\_signal(&cond\_barber);

       }

       pthread\_mutex\_unlock(&mutex);

       pthread\_mutex\_lock(&mutex);

       while (num\_customers < id) { // 等待理发师完成理发

           pthread\_cond\_wait(&cond\_customer, &mutex);

       }

       pthread\_mutex\_unlock(&mutex);

       printf("Customer %d leaves the barbershop.\n", id);

   } else { // 如果没有空椅子，顾客离开

       pthread\_mutex\_unlock(&mutex);

       printf("Customer %d leaves the barbershop because no chairs are available.\n", id

);

   }

   return NULL;

}

int main() {

   pthread\_t barber\_thread;

   pthread\_t customer\_threads[MAX\_CHAIRS \* 2]; // 创建足够多的顾客线程

   pthread\_mutex\_init(&mutex, NULL);

   pthread\_cond\_init(&cond\_barber, NULL);

   pthread\_cond\_init(&cond\_customer, NULL);

   pthread\_create(&barber\_thread, NULL, barber, NULL); // 创建理发师线程

   // 创建顾客线程

   int customer\_ids[MAX\_CHAIRS \* 2]; // 存储顾客线程的 ID

   for (int i = 0; i < MAX\_CHAIRS \* 2; i++) {

       customer\_ids[i] = i + 1; // 顾客编号从 1 开始

       pthread\_create(&customer\_threads[i], NULL, customer, &customer\_ids[i]);

       sleep(1); // 控制顾客线程的创建速度

   }

   // 等待所有顾客线程结束

   for (int i = 0; i < MAX\_CHAIRS \* 2; i++) {

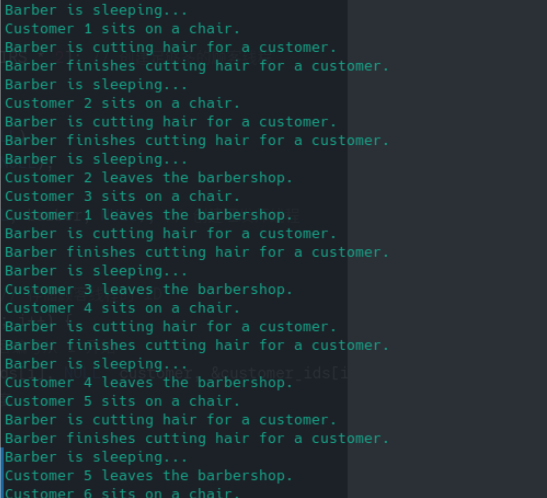
       pthread\_join(customer\_threads[i], NULL);

   }

   return 0;

}

实验结果：



六．实验结果分析（15分）

1．实验思考

（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。

在Linux中，可以使用/dev/urandom设备来生成高质量的伪随机数。/dev/urandom是一个特殊的文件，它会生成伪随机数，这些随机数可以用于各种应用，包括模拟游戏中的随机事件，如“石头、剪刀、布”游戏。

（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。

在Linux操作系统中，存在几种不同的进程间通信（IPC）机制，它们各自适用于不同的场合。以下是几种常见的IPC机制以及它们的特点和适用场合：

管道（Pipe）：管道是一种半双工的IPC机制，用于在具有亲缘关系的进程间传递数据。它只能在父进程和其直接子进程之间进行通信。管道有两种类型：匿名管道（Anonymous Pipe）和命名管道（Named Pipe）。匿名管道只能在具有父子关系的进程间通信，而命名管道可以在无关进程间通信。

信号（Signal）：信号是一种异步的IPC机制，用于在进程间传递简短的消息。它可以用于在进程间传递简单的通知、中断和异常等信息，用于处理进程间的异步事件。

共享内存（Shared Memory）：共享内存是一种高效的IPC机制，可以在多个进程间共享内存区域，从而实现进程间的数据共享。共享内存适用于需要高速数据传输和大量数据共享的场合，但需要进行显式的同步和互斥来保证多进程的数据一致性。

信号量（Semaphore）：信号量是一种计数器，用于在进程间控制共享资源的访问。它可以用于实现进程间的同步和互斥，从而避免竞争条件和死锁。信号量适用于需要控制多个进程对共享资源的并发访问的场合。

消息队列（Message Queue）：消息队列是一种通过消息传递进行进程间通信的机制，进程可以通过发送和接收消息来进行通信。消息队列适用于需要在进程间传递复杂数据结构和消息的场合，可以实现异步、多对多的通信方式。

套接字（Socket）：套接字是一种用于在网络中进行进程间通信的机制，它允许不同主机上的进程进行通信。套接字适用于需要在网络环境下进行进程间通信的场合，可以实现跨主机、跨网络的通信。

每种IPC机制都有其独特的优点和适用场合，开发者在选择IPC机制时应根据具体的应用需求、数据传递的性质、进程之间的关系等因素进行合理的选择

（3）在生产者-消费者问题中，实现一个产品的放入与取出跟一组多个产品的放入与取出，在实现方法上有什么不同？

在生产者-消费者问题中，涉及到单个产品的放入与取出以及一组多个产品的放入与取出，实现方法上存在一些不同。以下是两者之间可能存在的不同之处：

产品的放入与取出方式：在单个产品的放入与取出中，通常使用简单的变量或标志位来表示产品的状态，例如使用一个缓冲区或队列来存储产品。生产者将产品放入缓冲区或队列中，而消费者从缓冲区或队列中取出产品。而在一组多个产品的放入与取出中，可能需要使用更复杂的数据结构，如多维数组或多个缓冲区或队列来管理多个产品的状态。

同步与互斥机制：在单个产品的放入与取出中，通常可以使用简单的同步与互斥机制来管理生产者和消费者之间的并发访问，例如使用互斥锁（Mutex）或条件变量（Condition Variable）来实现生产者和消费者之间的同步与互斥。而在一组多个产品的放入与取出中，可能需要更复杂的同步与互斥机制，例如使用多个互斥锁或条件变量来管理多个产品的状态和访问。

缓冲区或队列的管理：在单个产品的放入与取出中，通常只需要一个缓冲区或队列来管理产品的状态和访问。而在一组多个产品的放入与取出中，可能需要管理多个缓冲区或队列，以确保多个产品之间的正确管理和访问。

生产者和消费者之间的通信：在单个产品的放入与取出中，生产者和消费者之间的通信通常较为简单，可以通过缓冲区或队列来传递产品。而在一组多个产品的放入与取出中，生产者和消费者之间的通信可能需要更复杂的协调和同步，以确保多个产品之间的顺序和关联性。

需要注意的是，实现生产者-消费者问题时，不同的应用场景和需求可能需要不同的设计和实现方法。在选择实现方法时，应考虑到具体的问题需求，包括产品的数量、访问模式、同步和互斥需求等，并根据具体情况选择合适的同步与互斥机制、缓冲区或队列管理方式以及生产者和消费者之间的通信方式。

（4）多线程并发与多进程并发有何不同与相同之处?

多线程并发和多进程并发是两种不同的并发模型，用于处理多个任务或进程同时执行的情况。它们在实现方式、资源管理、通信和同步等方面存在一些不同和相同之处，如下所示：

不同之处：

实现方式：多线程并发是在同一进程内创建多个线程，这些线程共享进程的内存空间和资源，通过线程间的函数调用和共享变量来实现通信和同步。而多进程并发是创建多个独立的进程，每个进程有自己独立的内存空间和资源，通过进程间的通信机制（如管道、消息队列、共享内存等）来实现通信和同步。

资源管理：在多线程并发中，多个线程共享进程的内存空间和资源，因此需要谨慎管理共享资源，如使用互斥锁、条件变量等来保护共享数据。而在多进程并发中，每个进程有自己独立的内存空间和资源，资源管理相对独立，不需要显式的同步机制。

通信和同步：在多线程并发中，线程间的通信和同步相对简单，可以通过共享变量、互斥锁、条件变量等进行实现。而在多进程并发中，由于进程间的内存空间相互独立，通信和同步的实现相对复杂，需要使用进程间通信（IPC）机制来进行数据传递和同步。

相同之处：

并发性：多线程并发和多进程并发都可以实现多个任务或进程的并发执行，从而提高系统的处理能力和效率。

需要考虑同步和竞态条件：无论是多线程并发还是多进程并发，都需要考虑到并发访问共享资源时可能出现的竞态条件（Race Condition）和需要进行同步的情况，以保证数据的一致性和正确性。

可以提高系统的并发性和响应性：无论是多线程并发还是多进程并发，都可以利用系统的多核资源，提高系统的并发性和响应性，从而更好地满足多任务处理的需求。

可能存在并发相关的问题：无论是多线程并发还是多进程并发，都可能存在一些并发相关的问题，如死锁、竞态条件、资源竞争等，需要合理的设计和实现来解决这些问题。

需要根据具体的应用场景和需求来选择合适的并发模型，考虑到系统的性能、复杂性、资源管理和通信同

2．分析、总结

在生产者-消费者问题中，实现单个产品的放入与取出和一组多个产品的放入与取出，在实现方法上有不同的挑战和需求。在单个产品的放入与取出的情况下，需要确保生产者和消费者之间的同步和互斥，以避免竞态条件和数据一致性的问题。可以使用互斥锁、条件变量等同步机制来实现。而在一组多个产品的放入与取出的情况下，需要考虑多个产品之间的关系和管理，可能需要使用队列、缓冲区等数据结构来管理多个产品，并设计合适的算法和逻辑来保证产品的放入和取出的正确性。

多线程并发和多进程并发有不同之处和相同之处。多线程并发是在同一进程内创建多个线程，共享进程的内存空间和资源，通过线程间的函数调用和共享变量来实现通信和同步。多进程并发是创建多个独立的进程，每个进程有自己独立的内存空间和资源，通过进程间的通信机制来实现通信和同步。不同之处包括实现方式、资源管理、通信和同步的机制等。相同之处包括都能实现并发执行、需要考虑同步和竞态条件、可以提高系统的并发性和响应性等。

并发编程是一个复杂的领域，涉及到多个线程或进程之间的交互和竞争，可能会引发一系列的并发相关的问题，如死锁、竞态条件、资源竞争等。合理的并发编程设计和实现可以提高系统的性能和效率，但也需要小心处理共享资源的同步和互斥，以保证数据的一致性和正确性。在选择多线程并发还是多进程并发时，需要考虑具体的应用场景和需求，权衡各自的优缺点，选择合适的并发模型。

总的来说，对于并发编程和操作系统中的不同概念和机制，深入理解其原理、特点和应用场景，合理选择并实现合适的并发模型，是保证并发程序正确性和性能的关键。同时，合理的并发编程设计和实现需要充分考虑资源管理、通信和同步等方面的问题，避免并发相关的问题和潜在的错误。

3．体会

设计合理的并发模型是关键：并发编程涉及到多个线程或进程之间的交互和竞争，因此设计合理的并发模型是至关重要的。选择合适的并发模型，如多线程、多进程、协程等，需要考虑具体的应用场景和需求，权衡各自的优缺点，以满足程序的性能、正确性和可维护性等要求。

同步和互斥是重要的问题：在并发编程中，多个线程或进程可能同时访问共享资源，因此同步和互斥是重要的问题。合理使用同步机制，如互斥锁、条件变量、信号量等，可以保证多个线程或进程之间的正确交互，避免竞态条件和数据一致性的问题。

错误处理和调试困难：并发编程中的错误可能会很难调试和定位，因为错误可能涉及到多个线程或进程之间的交互和竞争。因此，对错误的处理需要特别谨慎，包括错误的检测、处理和恢复。同时，合适的调试技术和工具也是必要的，以便在出现问题时能够迅速定位和解决。

性能和效率需要平衡：并发编程通常被用于提高系统的性能和效率，但不当的并发设计和实现可能会导致资源浪费、竞争和性能下降。因此，在并发编程中，需要仔细权衡性能和效率之间的平衡，合理地管理资源，避免不必要的竞争和开销。

学习和实践是不断改进的关键：并发编程是一个复杂的领域，需要不断学习和实践，不断改进和优化，并根据具体的应用场景和需求进行调整和优化。了解并发编程的最佳实践、常见的错误和解决方案，不断改进并发代码，是提高并发编程技能和保证并发程序质量的关键。

综上所述，合理的并发模型、同步和互斥的设计、错误处理和调试技巧、性能和效率的平衡以及持续的学习和实践，都是在并发编程中取得成功的关键要素。