实验4 管道与消息队列

**一、实验目的**

1. 掌握管道、消息队列、共享内存的通信机制。

2. 使用管道机制、共享内存和消息队列实现进程之间通信。

**二、实验工具与设备**

装有Linux系统的计算机。

1. **实验预备知识**

**进程间通信(IPC)：**

**我们知道进程之间都是相互独立的，任何一个进程的全局变量在另一个进程中是看不到的，如果进程之间需要交换数据就要通过内核。进程间通信(InterProcess Communication)的本质就是让两个进程看到共同的资源。**

**进程间通信的目的**

**数据传输：一个进程需要将它的数据发送给另一个进程**

**资源共享：多个进程之间共享同样的资源**

**通知事件：一个进程需要向另一个进程发送消息，通知其发生了某种事情（比如进程终止父进程告诉子进程）**

**进程控制：有些进程希望完全控制另一个进程的执行，此时控制进程希望能够拦截另一个进程的所有陷入和异常，能够及时知道它的状态改变。**

**进程间通信的分类：**

**管道：匿名管道，命名管道**

**System V IPC:消息队列，共享内存，信号量**

**POSIX IPC：消息队列，共享内存，信号量，互斥量，读写锁，条件变量**

一、管道(pipe)

　　我们把一个进程连接到另一个进程的一个数据流称之为管道，是Unix中最古老的进程间通信形式。我们可以分为匿名管道和命名管道。

1. 匿名管道

特点：

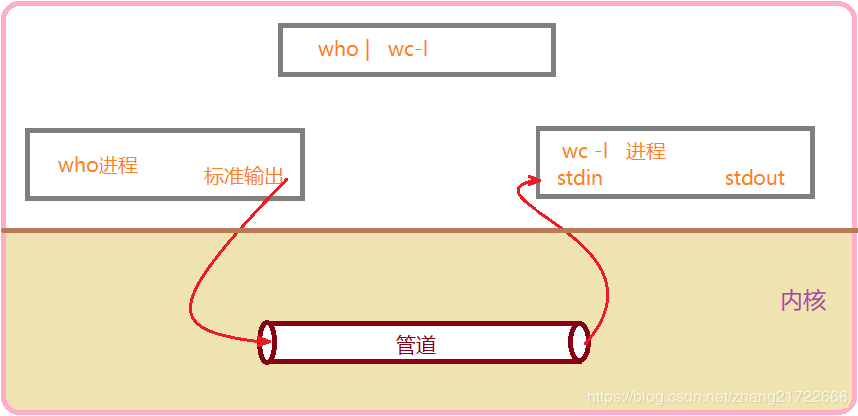
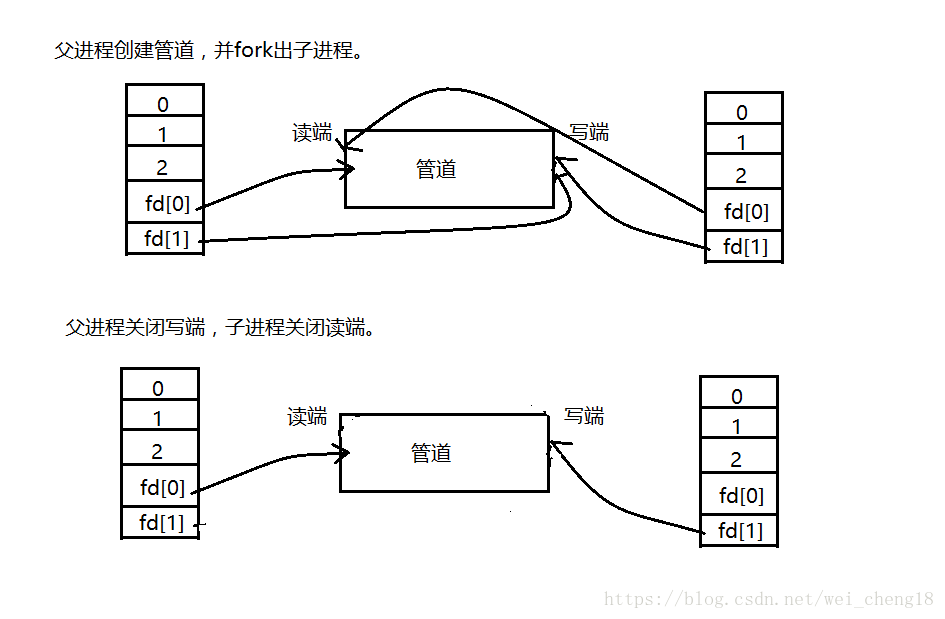
只能用于具有血缘关系的进程之间通信生命周期随进程，进程退出，管道释放管道是半双工的，数据只能从一个方向传输管道是基于字节流的管道是自带同步机制的，在保证数据安全的前提下，按照特定顺序访问临界资源

函数原型：

#include <unisted>int pipe(int fd[2]);

功能：创建一个匿名管道   
参数：fd文件描述符数组，其中fd[0]表示读，fd[1]表示写   
返回值：成功返回0，失败返回错误代码

例子：



test.c

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<string.h>

int main()

{

int fd[2]={0};

if(pipe(fd)==-1)

{

perror("pipe");

return 1;

}

pid\_t pid;

pid=fork();

if(pid==-1)

{

perror("fork");

return 2;

}

if(pid==0)

{

//child

close(fd[0]);

int a=5;

char \*buf="Hello, I am your child";

while(a--)

{

write(fd[1], buf, strlen(buf));

sleep(2);

}

}

else

{

//Parent

close(fd[1]);

while(1)

{

char buf[1024]={0};

ssize\_t s = read(fd[0], buf, sizeof(buf)-1);

buf[s]=0;

if(s>0)

{

printf("Parent: %s\n", buf);

}

else if(s==0)

{

printf("quit\n");

break;

}else

{

perror("read");

return 3;

}

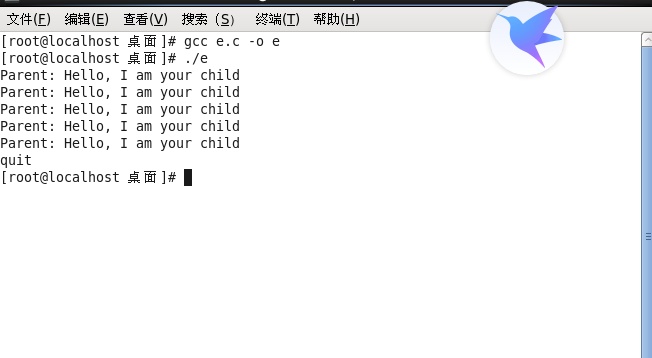
}

}

return 0;

}

结果



2.命名管道

特点：

命名管道是一种特殊类型的文件

命名管道可以用于不具有血缘关系的进程

除此之外与匿名管道基本相似

函数原型

#include <sys/types.h>#include <sys/stat.h>

int mkfifo(const char \*pathname, mode\_t mode);

功能：创建命名管道   
参数：pathname表示管道文件路径，mode表示文件权限   
返回值：成功返回0，失败返回-1，错误原因存于errno中

例子

client.c

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<fcntl.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<string.h>

int main()

{

int fd = open("mypipe", O\_WRONLY);

if(fd < 0)

{

perror("open");

return 1;

}

char buf[1024] = {0};

while(1)

{

printf("Please Enter # ");

fflush(stdout);

ssize\_t s = read(0, buf, sizeof(buf)-1);

if(s > 0)

{

buf[s] = 0;

write(fd, buf, strlen(buf));

}

else if(s == 0)

{

printf("read finish");

return 0;

}

else

{

perror("read");

return 2;

}

}

close(fd);

return 0;

}

server.c

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<fcntl.h>

#include<sys/types.h>

#include<sys/stat.h>

int main()

{

umask(0);

if(mkfifo("mypipe", 0644) < 0)

{

perror("mkfifo");

return 1;

}

int fd = open("mypipe", O\_RDONLY);

if(fd < 0)

{

perror("open");

return 2;

}

char buf[1024] = {0};

while(1)

{

printf("Please wait...\n");

ssize\_t s = read(fd, buf, sizeof(buf)-1);

if(s < 0)

{

perror("read");

return 3;

}

else if(s == 0)

{

printf("client is quit\n");

return 0;

}

else

{

buf[s-1] = 0;//为了去掉输入时的换行符，所以才s-1

printf("client says # %s\n", buf);

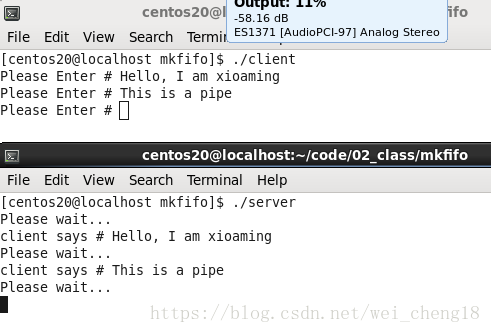
}

}

close(fd);

return 0;

}



二、消息队列(message queue)

消息队列也是System V IPC机制之一。

消息队列与命名管道类似，

但少了打开和关闭管道方面的复杂性。

但使用消息队列并未解决我们在使用命名管道时遇到的一些问题，

如管道满时的阻塞问题。

消息队列提供了一种在两个不相关进程间传递数据的简单有效的方法。

与命名管道相比，

消息队列的优势在于，它独立于发送和接收进程而存在，

这消除了在同步命名管道的打开和关闭时可能产生的一些困难。

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。

而且，每个数据块被认为含有一个类型，

接收进程可以独立地接收含有不同类型值的数据块。

好消息是，

A. 我们可以通过发送消息来几乎完全避免命名管道的同步和阻塞问题。

而且，

B. 我们可以用一些方法来提前查看紧急消息。

坏消息是，

A. 与管道一样，每个数据块有一个最大长度的限制，

而且，

B. 系统中所有队列所包含的全部数据块的总长度也有一个上限。

Linux系统中有两个宏定义:

 MSGMAX, 以字节为单位，定义了一条消息的最大长度。

 MSGMNB, 以字节为单位，定义了一个队列的最大长度。

函数定义

消息队列的函数定义如下:

#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);

int msgget(key\_t key, int msgflg);

int msgrcv(int msqid, void \*msg\_ptr, size\_t msg\_sz, long int msgtype, int msgflg);

int msgsnd(int msqid, const void \*msg\_ptr, size\_t msg\_sz, int msgflg);

1. msgget函数

我们用msgget函数来创建和访问一个消息队列:

int msgget(key\_t key, int msgflg);

参数:

key : 键值，

      和其他IPC机制一样，

      程序必须提供一个键值来命名某个特定的消息队列。

      特殊键值IPC\_PRIVATE用于创建私有队列，

      从理论上说，它应该只能被当前进程访问。

msgflag: 由9个权限标志组成。

         由IPC\_CREAT定义的一个特殊位必须和权限标志按位或才能创建一个新的消息队列。

返回值:

成功时，返回一个正整数，即队列标识;

失败时，返回-1.

2. msgsnd函数

该函数用来把消息添加到消息队列中:

int msgsnd(int msqid, const void \*msg\_ptr, size\_t msg\_sz, int msgflg);

消息的结构受两方面约束:

A. 它的长度必须小于系统规定的上限;

B. 它必须以一个长整形成员变量开始,

   接收函数将用这个成员变量来确定消息的类型。

当使用消息时，最好把消息结构定义如下:

struct my\_message {

  long int message\_type;

  /\* The data you wish to transfer \*/

}

由于消息的接收要用到message\_type,

所以不能忽略它，

且必须在声明自己的数据结构时包含它，

并且将它初始化为一个已知值。

参数:

msqid  : 是由msgget函数返回的消息队列标识符。

msg\_ptr: 一个指向准备发达消息的指针，

         这个消息必须以前面说的以一个长整形成员变量开始。

msg\_sz : 是msg\_ptr指向的消息的长度，

         这个长度不能包括长整形消息类型成员变量的长度。

msgflg : 控制在当前消息队列满或队列消息到达系统范围的限制时，

         将要发生的事情。

         IPC\_NOWAIT, 函数将立即返回，不发送消息并且返回值为-1.

         如果IPC\_NOWAIT标志被消除，则发送进程将挂起，

         以等待队列中腾出可用空间。

返回值:

成功时，返回0，消息数据的一份副本将被放到消息队列中;

失败时，返回-1.

3. msgrcv函数

该函数从一个消息队列中获取消息:

int msgrcv(int msqid, void \*msg\_ptr, size\_t msg\_sz, long int msgtype, int msgflg);

参数:

msqid  : 由msgget函数返回的消息队列标识符;

msg\_ptr: 一个指向准备接收消息的指针，

         和前面介绍的一样，

         要以一个长整形成员变量开始。

msg\_sz : 是msg\_ptr指向的消息的长度，

         它不包括长整形消息成员变量的长度。

msgtype: 是一个长整形，

         它可以实现一个简单形式的接收优先级。

         == 0, 将获取队列中的第一个可用消息;

         >  0, 将获取具有相同消息类型的第一个消息;

         <  0, 将获取消息类型 <= msgtype的绝对值的第一个消息。

如果 只想按消息发送的顺序来接收它们，

     就把msgtype设置为0.

如果 只想获取某一特定类型的消息，

     就把msgtype设置为相应的类型值。

如果 只想接收类型 <= n的消息，

     就把msgtype设置为-n.

msgflg: 用于控制当队列中没有相应类型的消息可以接收时将发送的事情。

        如果 msgflg中的IPC\_NOWAIT标志被设置，

             函数立即返回-1.

        如果 msgflg中的IPC\_NOWAIT标志被消除，

             进程将会挂起以等待一条相应类型的消息到达。

返回值:

成功时， 函数返回放到接收缓存区中的字节数，

         消息被复制到由msg\_ptr指向的用户分配的缓存中。

         然后删除消息队列中对应的消息。

失败时， 返回-1.

4. msgctl函数

该函数用来控制消息队列，

int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);

msqid\_ds结构至少包含以下成员:

struct msqid\_ds {

  uid\_t msg\_perm.uid;

  uid\_t msg\_perm.gid

  mode\_t msg\_perm.mode;

}

参数:

msqid  : 由msgget返回的消息队列标识符。

command: 将要采取的动作，

         它可以取3个值，

IPC\_STAT  把msqid\_ds结构中的数据设置为消息队列的当前关联值

IPC\_SET   如果进程有足够的权限，

          就把消息队列的当着关联值设置为msqid\_ds结构中给出的值。

IPC\_RMID  删除消息队列

          如果删除消息队列时，

          某个进程正在msgsnd或msgrcv函数中等待，这两个函数将失败。

返回值:

成功时，返回0，

失败时，返回-1.

client.c

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<string.h>

#include<errno.h>

#include<sys/msg.h>

#include<sys/ipc.h>

#define IPC\_KEY 0x123123

//定义消息队列的key，好知道写道那个队列中，取哪个队列中拿

#define TYPE\_R 1

#define TYPE\_W 2

struct msgbuf

{

long mtype;

char mtext[1024];

};

int main()

{

int msqid = -1;

msqid = msgget(IPC\_KEY,IPC\_CREAT|0664);

if(msqid < 0)

{

perror("msgget error");

return -1;

}

while(1)

{

struct msgbuf buf;

memset(&buf,0,sizeof(struct msgbuf));

msgrcv(msqid,&buf,1024,TYPE\_W,0);

printf("w say:[%s]\n",buf.mtext);

memset(&buf,0,sizeof(struct msgbuf));

buf.mtype = TYPE\_R;

printf(">>>");

scanf("%s",buf.mtext);

msgsnd(msqid,&buf,1024,0);

}

msgctl(msqid,IPC\_RMID,NULL);

return 0;

}

server.c

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<string.h>

#include<errno.h>

#include<sys/msg.h>

#include<sys/ipc.h>

#define IPC\_KEY 0x123123

//定义消息队列的key，好知道写道那个队列中，取哪个队列中拿

#define TYPE\_R 1

#define TYPE\_W 2

struct msgbuf

{

long mtype;

char mtext[1024];

};

int main()

{

int msqid = -1;

msqid = msgget(IPC\_KEY,IPC\_CREAT|0664);

if(msqid < 0)

{

perror("msgget error");

return -1;

}

while(1)

{

struct msgbuf buf;

memset(&buf,0,sizeof(struct msgbuf));

buf.mtype = TYPE\_W;

printf(">>>");

scanf("%s",buf.mtext);

msgsnd(msqid,&buf,1024,0);

memset(&buf,0,sizeof(struct msgbuf));

msgrcv(msqid,&buf,1024,TYPE\_R,0);

printf("r say:[%s]\n",buf.mtext);

}

msgctl(msqid,IPC\_RMID,NULL);

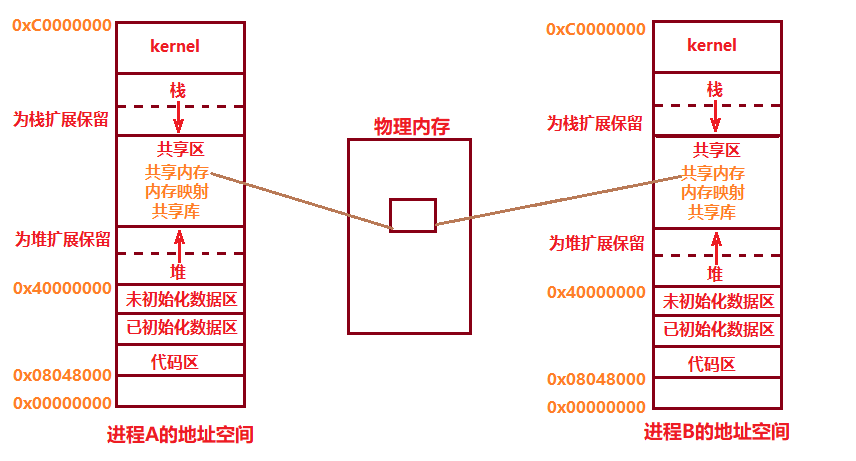
return 0;

}

1. 共享内存：

前边说过，进程间通信的实质就是让两个不相干的进程看到同一份公共的资源，而内存是资源的一种，那么，如果让两个进程可以使用同一块内存，两个进程都可以往这块内存里边写东西和取东西，这不就是实现了进程间的通信了么。通过共享内存实现进程间的通信，原理很简单，主要就是它的实现了。

每个进程都具有自己独立的地址空间，又怎么让两个进程共享同一块内存呢？我们都知道，给进程的地址空间其实都是虚拟地址，然后操作系统会根据进程的实际需要才给进程分配合适的物理内存（将虚拟地址映射到物理地址空间中去），那么问题就好解决了，只要两个进程映射到同一块物理内存空间不就行了？但是映射的过程是由操作系统完成与控制，怎么让属于不同进程的两块虚拟地址空间映射到同一块内存空间呢？



最简单的共享内存的使用流程

①ftok函数生成键值

②shmget函数创建共享内存空间

③shmat函数获取第一个可用共享内存空间的地址

④shmdt函数进行分离（对共享存储段操作结束时的步骤，并不是从系统中删除共享内存和结构）

⑤shmctl函数进行删除共享存储空间

1.ftok函数生成键值

每一个共享存储段都有一个对应的键值（key）相关联（消息队列、信号量也同样需要）。

所需头文件：#include<sys/ipc.h>

函数原型 ：key\_t ftok(const char \*path ,int id);

path为一个已存在的路径名

id为0~255之间的一个数值，代表项目ID，自己取

返回值：成功返回键值（相当于32位的int）。出错返回-1

例如：key\_t key = ftok( “/tmp”, 66);

2.shmget函数创建共享存储空间并返回一个共享存储标识符

所需头文件：#include<sys/shm.h>

函数原型： int shmget(key\_t key, size\_t size,int flag);

key为ftok生成的键值

size为共享内存的长度，以字节为单位

flag为所需要的操作和权限，可以用来创建一个共享存储空间并返回一个标识符或者获得一个共享标识符。

flag的值为IPC\_CREAT：如果不存在key值的共享存储空间，且权限不为0，则创建共享存储空间，并返回一个共享存储标识符。如果存在，则直接返回共享存储标识符。

flag的值为 IPC\_CREAT | IPC\_EXCL：如果不存在key值的共享存储空间，且权限不为0，则创建共享存储空间，并返回一个共享存储标识符。如果存在，则产生错误。

返回值：成功返回共享存储ID；出错返回-1

例如：int id = shmget(key,4096,IPC\_CREAT|IPC\_EXCL|0666);创建一个大小为4096个字节的权限为0666（所有用户可读可写，具体查询linux权限相关内容）的共享存储空间，并返回一个整形共享存储标识符，如果key值已经存在有共享存储空间了，则出错返回-1。

   int id = shmget(key,4096,IPC\_CREAT|0666);创建一个大小为4096个字节的权限为0666（所有用户可读可写，具体查询linux权限相关内容）的共享存储空间，并返回一个共享存储标识符，如果key值已经存在有共享存储空间了，则直接返回一个共享存储标识符。

3.shmat函数获取第一个可用共享内存空间的地址

所需头文件：#include<sys/shm.h>

函数原型： void \*shmat(int shmid, const void \*addr, int flag);

shmid为shmget生成的共享存储标识符

addr指定共享内存出现在进程内存地址的什么位置，直接指定为NULL让内核自己决定一个合适的地址位置

flag为对数据的操作，如果指定为SHM\_RDONLY则以只读方式连接此段，其他值为读写方式连接此段。

翻阅linux下shm.c文件得到#define SHM\_RDONLY      010000  /\* read-only access \*/

返回值：成功返回指向共享存储段的指针；错误返回-1（打印出指针的值为全F）

例如：char \*addr  = shmat(id, NULL, 0);就会返回第一个可用的共享内存地址的指针的值给addr

4.shmdt函数进行分离

当不需要对此共享内存进行操作时候，调用shmdt函数进行分离，不是删除此共享存储空间哟。

所需头文件：#include<sys/shm.h>

函数原型： int shmdt(const void \*addr);

addr为shmat函数返回的地址指针

返回值：成功返回0；错误返回-1

例如：int ret = shmdt(addr);

5.shmctl函数对共享内存进行控制

简单的操作就是删除共享存储空间了，也可以获取和改变共享内存的状态

所需头文件：#include<sys/shm.h>

函数原型：int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);

shmid就是shmget函数返回的共享存储标识符

cmd有三个，常用删除共享内存的为IPC\_RMID；IPC\_STAT：得到共享内存的状态，把共享内存的shmid\_ds结构复制到buf中；IPC\_SET：改变共享内存的状态，把buf所指的shmid\_ds结构中的uid、gid、mode复制到共享内存的shmid\_ds结构内。（内核为每个共享存储段维护着一个结构，结构名为shmid\_ds，这里就不讲啦，里面存放着共享内存的大小，pid，存放时间等一些参数）

buf就是结构体shmid\_ds

返回值：成功返回0；错误返回-1

例如：int ret = shmctl(id, IPC\_RMID,NULL);删除id号的共享存储空间

使用ipcs -m可以查看当前系统所有的共享内存空间信息

如果你的程序创建了一个共享内存段，但没有销毁，可以使用命令ipcrm -m shmid命令删除共享内存段，不然程序再运行有可能出错。

下面用一个代码例子来使用共享内存

我创建了一个结构体，想让结构体存入共享内存。写了两个程序，service和client，代码基本相同，不同就是service程序的开始创建共享内存。这两个程序是一个死循环，让你选择是存数据还是读数据还是销毁共享内存。

command.h文件。好让service和client调用嘛，方便。

#define NAME\_LEN 20

typedef struct {

char name[NAME\_LEN];

int age;

}ckx;

int sharememory(int ipc\_size,int flag);

int create\_ipc(int ipc\_size);

int get\_ipc(int ipc\_size);

command.c文件，构造想存入的结构体，和共享内存的操作函数

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<sys/types.h>

#include<sys/shm.h>

#include "Command.h"

int sharememory(int ipc\_size,int flag)

{

int id;

key\_t key=ftok("/tmp",66);

if(key < 0)

{

printf("get key error\n");

return -1;

}

id = shmget(key,ipc\_size,flag);

if(id < 0)

{

printf("get id error\n");

return -1;

}

return id;

}

int create\_ipc(int ipc\_size)

{

return sharememory(ipc\_size,IPC\_CREAT|IPC\_EXCL|0666);

}

int get\_ipc(int ipc\_size)

{

return sharememory(ipc\_size,IPC\_CREAT|0666);

}

int destroy\_sharememory(int id)

{

return shmctl(id,IPC\_RMID,NULL);

}

service.c文件。创建共享内存空间啦，读写等

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<sys/shm.h>

#include<sys/types.h>

#include<stdlib.h>

#include "Command.h"

int main()

{

int id=create\_ipc(sizeof(ckx));

int i=0;

ckx \*p;

if(id < 0)

{

printf("create sharememory error\n");

return 0;

}

id = 0;

while(1)

{

printf("\n\n1.input data to sharememory\n2.get sharememory data\n\

3.destroy sharememory\ninput select:");

scanf("%d",&i);

if(i > 3 |i< 1)

{

printf("input error\n");

continue;

}

id = get\_ipc(sizeof(ckx));

if(id < 0)

{

printf("get sharememory error\n");

break;

}

p = (ckx \*)shmat(id,NULL,0);

if(p < 0)

{

printf("get sharememory addr error\n");

p = NULL;

break;

}

if(i == 1)

{

char name[NAME\_LEN];

int age=0;

printf("input name:");

fflush(stdin);

getchar();

gets(name);

printf("input age:");

scanf("%d",&age);

strcpy(p->name,name);

p->age = age;

printf("write success\n");

if(shmdt(p) == -1)

{

printf("shmdt error\n");

}

id = 0;

}

if(i == 2)

{

printf("name:%s \t age:%d\n",p->name,p->age);

if(shmdt(p) == -1)

{

printf("shmdt error\n");

break;

}

id = 0;

}

if(i == 3)

{

if(shmdt(p) == -1)

{

printf("shmdt error\n");

break;

}

break;

}

}

if(id !=0)

{

if(destroy\_sharememory(id)<0)

{

printf("destroy error\n");

}

}

}

client.c基本上就和service.c代码差不多啦，只是想体现共享内存嘛，service读写和client读写，观察现象，体现共享内存

#include<stdio.h>

#include<sys/shm.h>

#include<stdlib.h>

#include<string.h>

#include<sys/types.h>

#include<stdlib.h>

#include "command.h"

int main()

{

int i=0;

ckx \*p;

int id = 0;

while(1)

{

printf("\n\n1.input data to sharememory\n2.get sharememory data\n\

3.destroy sharememory\ninput select:");

scanf("%d",&i);

if(i > 3 |i< 1)

{

printf("input error\n");

continue;

}

id = get\_ipc(sizeof(ckx));

if(id < 0)

{

printf("get sharememory error\n");

break;

}

p = (ckx \*)shmat(id,NULL,0);

if(p < 0)

{

printf("get sharememory addr error\n");

p = NULL;

break;

}

if(i == 1)

{

char name[NAME\_LEN];

int age=0;

fflush(stdin);

getchar();

printf("input name:");

gets(name);

printf("input age:");

scanf("%d",&age);

strcpy(p->name,name);

p->age = age;

printf("write success\n");

if(shmdt(p) == -1)

{

printf("shmdt error\n");

}

id = 0;

}

if(i == 2)

{

printf("name:%s \t age:%d\n",p->name,p->age);

if(shmdt(p) == -1)

{

printf("shmdt error\n");

break;

}

id = 0;

}

if(i == 3)

{

if(shmdt(p) == -1)

{

printf("shmdt error\n");

break;

}

break;

}

}

if(id !=0)

{

if(destroy\_sharememory(id)<0)

{

printf("destroy error\n");

}

}

}

