实验6 进程通信（2）共享内存和消息队列通信

实验目的

1. 能够利用共享内存通信原理设计程序
2. 掌握消息传送的原理
3. 能够利用消息队列原理设计进程通信程序
4. 掌握进程通信的不同方式之间的区别、特点、适用情况

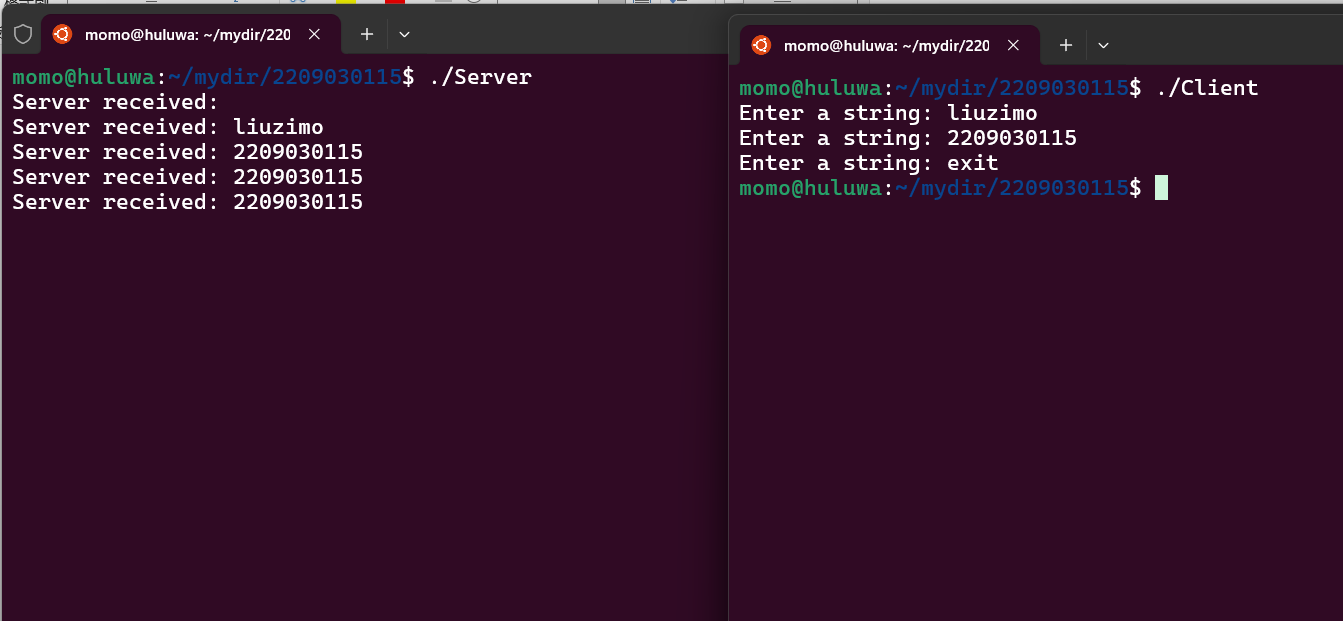
实验内容

1.设计两个程序，Client接受键盘输入的字符串，写入共享存储区；Server读出共享内存的内容输出。两个进程可以连续通信，设定结束字符串，两个进程的同步通过sleep或控制进程执行顺序实现（也可以考虑进程信号量）。

要求：程序源代码和运行截图。

|  |
| --- |
| Client |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <sys/ipc.h>  #include <sys/shm.h>  #include <sys/sem.h>  #include <sys/types.h>  #define SHM\_KEY 1234  #define SEM\_KEY 5678  #define SHM\_SIZE 1024  #define END\_STRING "exit"  void sem\_lock(int semid) {  struct sembuf sb = {0, -1, 0};  semop(semid, &sb, 1);  }  void sem\_unlock(int semid) {  struct sembuf sb = {0, 1, 0};  semop(semid, &sb, 1);  }  int main() {  int shmid, semid;  char \*shm\_addr;  shmid = shmget(SHM\_KEY, SHM\_SIZE, 0666 | IPC\_CREAT);  if (shmid < 0) {  perror("shmget error");  exit(1);  }  shm\_addr = (char \*)shmat(shmid, NULL, 0);  if (shm\_addr == (char \*)-1) {  perror("shmat error");  exit(1);  }  semid = semget(SEM\_KEY, 1, 0666 | IPC\_CREAT);  if (semid < 0) {  perror("semget error");  exit(1);  }  semctl(semid, 0, SETVAL, 1);  char input[SHM\_SIZE];  while (1) {  printf("Enter a string: ");  fgets(input, SHM\_SIZE, stdin);  input[strcspn(input, "\n")] = 0;  sem\_lock(semid);  strncpy(shm\_addr, input, SHM\_SIZE);  sem\_unlock(semid);  if (strcmp(input, END\_STRING) == 0) {  break;  }  sleep(1);  }  shmdt(shm\_addr);  shmctl(shmid, IPC\_RMID, NULL);  semctl(semid, 0, IPC\_RMID, 0);  return 0;  } |

|  |
| --- |
| Server |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <unistd.h>  #include <sys/ipc.h>  #include <sys/shm.h>  #include <sys/sem.h>  #include <sys/types.h>  #define SHM\_KEY 1234  #define SEM\_KEY 5678  #define SHM\_SIZE 1024  #define END\_STRING "exit"  void sem\_lock(int semid) {  struct sembuf sb = {0, -1, 0};  semop(semid, &sb, 1);  }  void sem\_unlock(int semid) {  struct sembuf sb = {0, 1, 0};  semop(semid, &sb, 1);  }  int main() {  int shmid, semid;  char \*shm\_addr;  shmid = shmget(SHM\_KEY, SHM\_SIZE, 0666 | IPC\_CREAT);  if (shmid < 0) {  perror("shmget error");  exit(1);  }  shm\_addr = (char \*)shmat(shmid, NULL, 0);  if (shm\_addr == (char \*)-1) {  perror("shmat error");  exit(1);  }  semid = semget(SEM\_KEY, 1, 0666 | IPC\_CREAT);  if (semid < 0) {  perror("semget error");  exit(1);  }  while (1) {  sem\_lock(semid);  printf("Server received: %s\n", shm\_addr);  sem\_unlock(semid);  if (strcmp(shm\_addr, END\_STRING) == 0) {  break;  }  sleep(10);  }  shmdt(shm\_addr);  return 0;  } |



2.运行以下程序，分析程序的运行结果。

（1）send.c

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

struct msgbuf

{

long mtype;

char mtext[1000];

};

int main()

{

int qid;

key\_t key=75;

int len;

struct msgbuf msg;

qid=msgget(key,IPC\_CREAT|0666);

printf("Input some text to send:\n");

fgets((&msg)->mtext,512,stdin);

msg.mtype=getpid();

len=strlen(msg.mtext);

msgsnd(qid,&msg,len,0);

printf("message has been sent to message queue\n");

return 0;

}

（2） receive.c

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

struct msgbuf

{

long mtype;

char mtext[1000];

};

int main()

{

int qid;

key\_t key=75;

int len;

struct msgbuf msg;

qid=msgget(key,0666);

msgrcv(qid,&msg,512,0,0);

printf("read from queue：%s\n",msg.mtext);

printf("The sending process id is:%ld\n",msg.mtype);

//msgctl(qid,IPC\_RMID,NULL)；

printf("message queue has been deleted\n");

return 0;

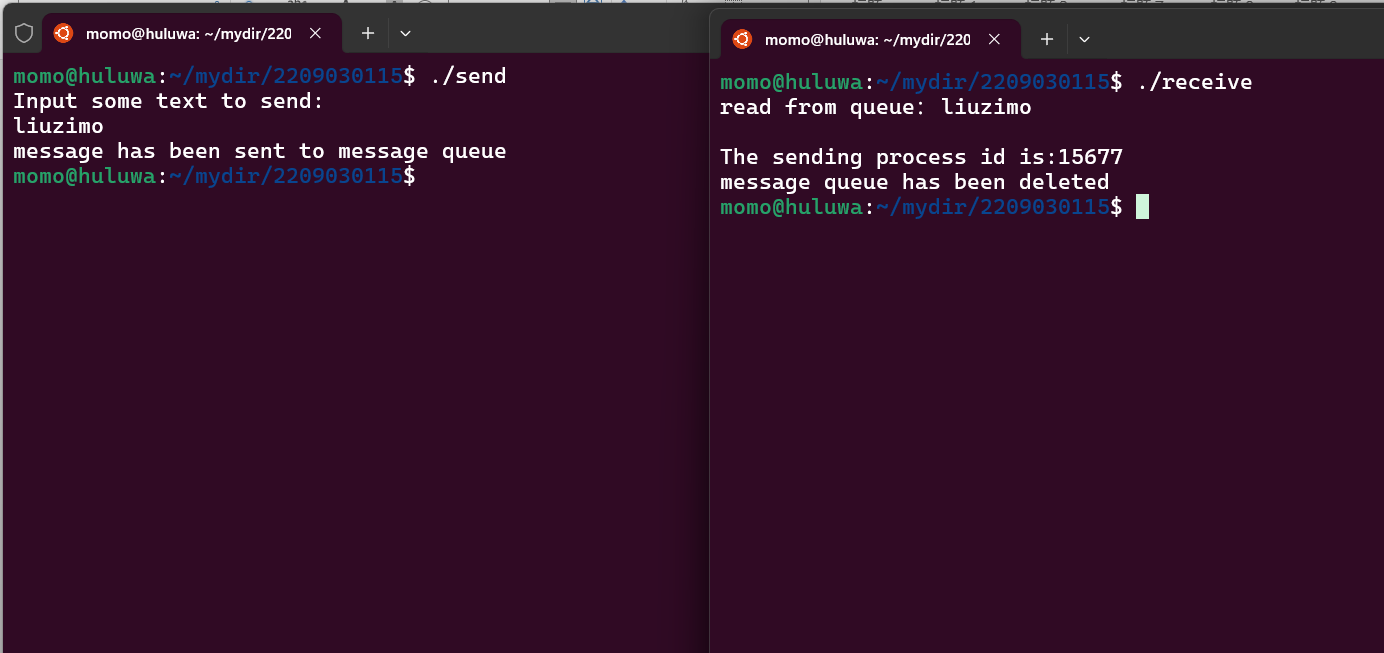
}

步骤：分别在前后台运行send和receive程序，观察两个进程的同步情况。

或者打开两个terminal，一个运行send，另一个运行receive，观察进程的同步情况。

要求：

1. 程序运行截图，分析程序实现的功能？



程序实现了两个进程间的通信，通过send.c获取一个消息队列并向队列中放信息，将输入的内容与进程ID一块发送给消息队列中。Receive.c从同一个消息队列中读取消息并进行输出。

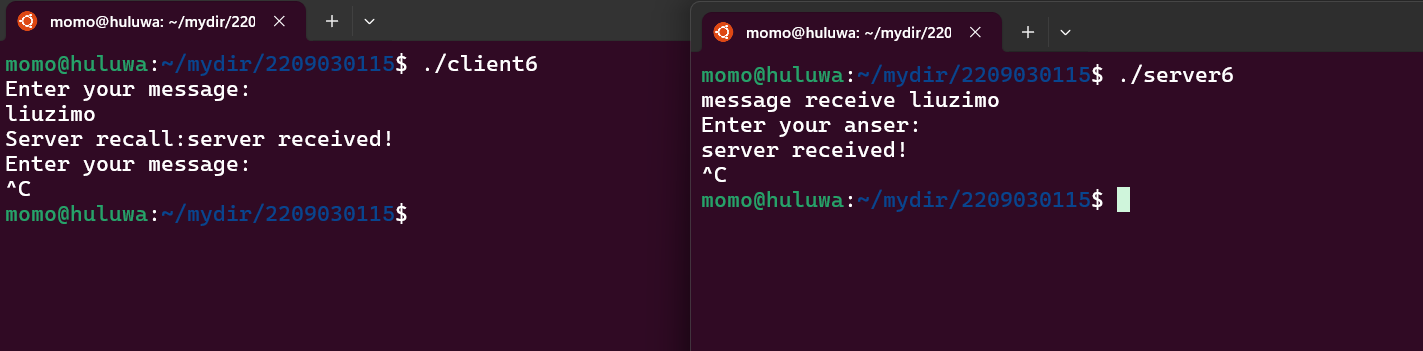
1. 分析消息队列的如何控制消息发送接收的同步

使用msgget来创建消息队列或者获取现有的消息队列，从而可以保证发送发与接受方使用同一个共享的消息队列。Send在发送中将mtype设置为当前进程的id，这样就可以区分消息是由哪个进程发送。通过msgsnd函数发送消息，它不会等待接收端的接收消息而是立即返回，send端在消息发送后不需要等待接收端的同步读取。而msgrcv函数用于接收消息，该函数会阻塞接收端直到消息队列中有符号条件的消息，从而实现了隐式同步。

3.编写两个程序：client实现将键盘接收的一行字符串发送给server，输出接收到的应答消息；server接收client消息输出，发送”server received!”给client后结束。使用系统调用msgget( )，msgsnd( )，msgrcv( )，及msgctl( )编制程序。

（1）至少实现client和server的一次双向通信。

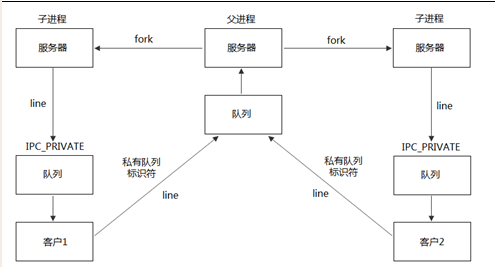
实现双向通信的截图：（server端可以自定义回发消息，client端接收）



|  |
| --- |
| Client:  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #include <unistd.h>  #include <stdlib.h>  #include <stdio.h>  #include <string.h>  struct msgbuf{  long mtype;  char message[1000];  };  int main(){  int msg\_id1;  int a = msgctl(1234, IPC\_RMID, NULL);  msg\_id1 = msgget((key\_t)10001,IPC\_CREAT|0666);    if(msg\_id1 == -1){  perror("msgget error:");  }  struct msgbuf msginfo;  struct msgbuf msginfo1;    msginfo.mtype = 1;  msginfo1.mtype = 2;    char input[1000];  while(1){  printf("Enter your message:\n");    fgets(input, 1000, stdin);  input[strcspn(input, "\n")] = 0;    strncpy(msginfo.message,input,1000);    if(msgsnd(msg\_id1,&msginfo,sizeof(msginfo.message),0) == -1){  perror("message send error:");  return -1;  }  //sleep(5);  if(msgrcv(msg\_id1,(void\*)&msginfo1,sizeof(msginfo1.message),2,0) == -1){  perror("message recv error:");  return -1;  }  printf("Server recall:%s\n",msginfo1.message);    }    return 0;  } |
| Server:  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #include <unistd.h>  #include <stdlib.h>  #include <stdio.h>  #include <string.h>  struct msgbuf{  long mtype;  char message[1000];  };  int main(){  int msg\_id1;  msg\_id1 = msgget((key\_t)10001,IPC\_CREAT|0666);    if(msg\_id1 == -1){  perror("msgget error:");  }  struct msgbuf msginfo;  struct msgbuf msginfo1;  msginfo.mtype = 1;  msginfo1.mtype = 2;    char input[1000];  while(1){  bzero(&msginfo,sizeof(msginfo));  if (msgrcv(msg\_id1, (void\*)&msginfo, sizeof(msginfo.message),1,0) == -1)  {  perror("message recv error:");  return -1;  }  printf("message receive %s\n",msginfo.message);    if(msginfo.message == "exit"){  break;  }  sleep(2);  printf("Enter your anser:\n");  fgets(input, 1000, stdin);  input[strcspn(input, "\n")] = 0;    strncpy(msginfo1.message,input,1000);    if(msgsnd(msg\_id1,&msginfo1,sizeof(msginfo1.message),0) == -1){  perror("message send error:");  return -1;  }    sleep(30);    }  return 0;    } |

（2）选做的功能：将server程序设置为能够并发处理多个客户的程序（参考下图的程序结构），服务器无限循环，CTRL+C结束，设计捕捉信号，信号处理中删除消息队列，服务器收到客户机的消息后发给不同客户机不同的应答消息（内容自定）。

要求：程序代码及运行结果截图.



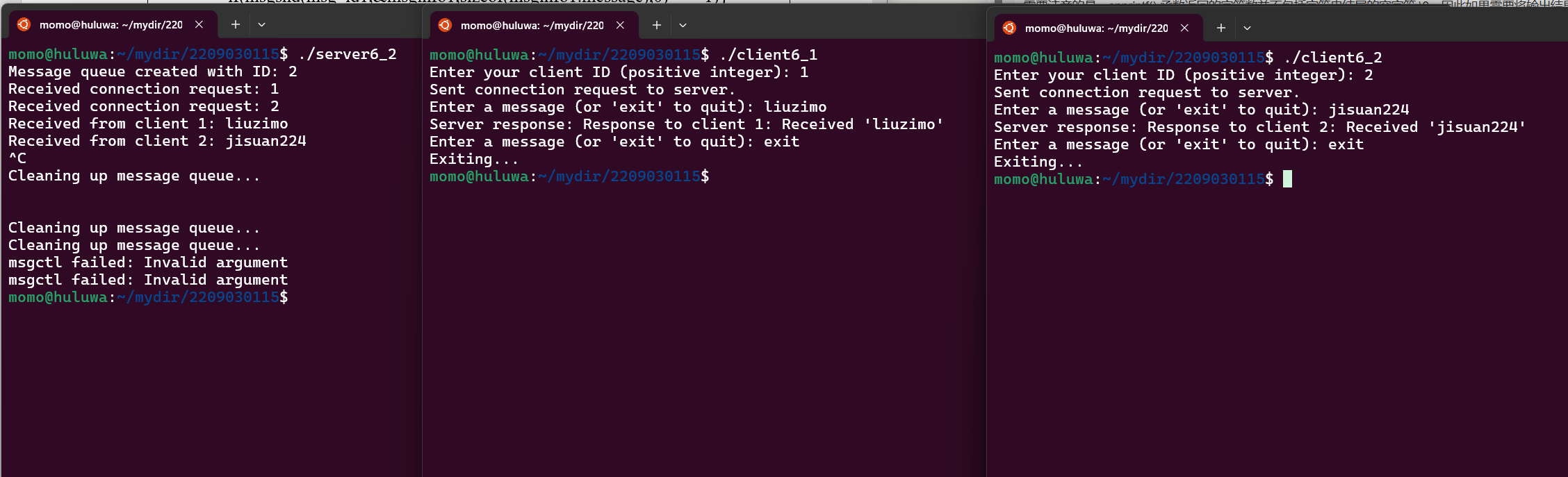
截图：

展示了服务端接收client6\_1与client6\_2的信息。

服务端首先接收客户端的client ID以此分别是哪个客户端并针对不同的客户端建立对应的子进程来处理客户端的消息，并可回发接收成功的消息给客户端即：Server response ：Response to client 1： Received’liuzimo’.

其次因为父进程在fork（）为子进程时，子进程会继承父进程的上下文，所以利用不同的子进程即可完成针对不同客户端的信息通信。其次父进程在创建子进程后，会继续返回监听状态，继续等待其他客户端的消息，由于子进程与父进程均独立运行，所以父进程不会因为某个客户端的处理过程而被阻塞掉。

在按下ctrl+c结束服务器时，服务器会选择删除消息队列，因为我只使用了一个id为12345的消息队列，因为父进程与子进程均会响应ctrl+c的信号，所以会执行三次cleanup函数，之所以在执行cleanup函数时执行了三次时第一次会成功其余两次会返回错误信息，是因为在 msgctl() 成功删除消息队列后，再次尝试删除同一个队列会失败。



|  |
| --- |
| Server:  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #include <signal.h>  #include <unistd.h>  #define MSG\_KEY 12345  #define MSG\_SIZE 128  struct message {  long msg\_type;  char msg\_text[MSG\_SIZE];  };  int msg\_id;  void cleanup(int signo) {  printf("\nCleaning up message queue...\n");  if (msgctl(msg\_id, IPC\_RMID, NULL) == -1) {  perror("msgctl failed");  }  exit(0);  }  void handle\_client(int client\_id) {  struct message msg;  char response[200];  while (1) {  if (msgrcv(msg\_id, &msg, sizeof(msg.msg\_text), client\_id, 0) == -1) {  perror("msgrcv failed");  exit(1);  }  printf("Received from client %d: %s\n", client\_id, msg.msg\_text);  snprintf(response, 200, "Response to client %d: Received '%s'", client\_id, msg.msg\_text);    msg.msg\_type = client\_id + 100;  strncpy(msg.msg\_text, response, MSG\_SIZE);  if (msgsnd(msg\_id, &msg, sizeof(msg.msg\_text), 0) == -1) {  perror("msgsnd failed");  exit(1);  }  }  }  int main() {  struct message msg;  signal(SIGINT, cleanup);  msg\_id = msgget(MSG\_KEY, IPC\_CREAT | 0666);  if (msg\_id == -1) {  perror("msgget failed");  exit(1);  }  printf("Message queue created with ID: %d\n", msg\_id);  while (1) {  if (msgrcv(msg\_id, &msg, sizeof(msg.msg\_text), 1, 0) == -1) {  perror("msgrcv failed");  continue;  }  printf("Received connection request: %s\n", msg.msg\_text);  int client\_id = atoi(msg.msg\_text);  pid\_t pid = fork();  if (pid < 0) {  perror("fork failed");  continue;  }  if (pid == 0) {  handle\_client(client\_id);  exit(0);  }  }  return 0;  } |
| Client:  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <sys/ipc.h>  #include <sys/msg.h>  #include <unistd.h>  #define MSG\_KEY 12345  #define MSG\_SIZE 128  struct message {  long msg\_type;  char msg\_text[MSG\_SIZE];  };  int main() {  int msg\_id;  struct message msg;  int client\_id;  char input[MSG\_SIZE];  msg\_id = msgget(MSG\_KEY, 0666);  if (msg\_id == -1) {  perror("msgget failed");  exit(1);  }  printf("Enter your client ID (positive integer): ");  scanf("%d", &client\_id);  msg.msg\_type = 1;  snprintf(msg.msg\_text, MSG\_SIZE, "%d", client\_id);  if (msgsnd(msg\_id, &msg, sizeof(msg.msg\_text), 0) == -1) {  perror("msgsnd failed");  exit(1);  }  printf("Sent connection request to server.\n");  while (1) {  printf("Enter a message (or 'exit' to quit): ");  scanf(" %[^\n]", input);  if (strcmp(input, "exit") == 0) {  printf("Exiting...\n");  break;  }  msg.msg\_type = client\_id;  strncpy(msg.msg\_text, input, MSG\_SIZE);  if (msgsnd(msg\_id, &msg, sizeof(msg.msg\_text), 0) == -1) {  perror("msgsnd failed");  exit(1);  }  if (msgrcv(msg\_id, &msg, sizeof(msg.msg\_text), client\_id + 100, 0) == -1) {  perror("msgrcv failed");  exit(1);  }  printf("Server response: %s\n", msg.msg\_text);  }  return 0;  } |

1. 总结无名管道、消息队列、共享内存的通信特点和适用场合。

无名管道：

特点：无名管道是半双工的，数据只能沿一个方向流动，只能在有亲缘关系的进程（如父子进程）之间使用。容量有限，进行写操作时可能被阻塞。

适用场合：适用于同一进程派生的父子进程之间的简单数据传递。

消息队列：

特点：可以在任何相互独立的进程之间通信，并且以消息为单位传递数据，每条消息包含标识符和内容。通过标识符实现不同优先级的消息处理，同步性较强，避免了管道中必须按顺序读写的问题。

适用场合：任务调度、异步消息传递、事件驱动的系统，以及用于需要在多个进程间传递复杂结构化数据的场景。

共享内存：

特点：高效性，数据直接存储在共享内存区，读写速度快。共享内存允许不相关的进程共享数据，并且需要同步机制，因为多个进程可以同时访问，需要借助信号量或其他同步机制避免冲突。通常在进程结束时要删除相关的共享内存区域。

适用场合：适用于需要共享大量数据或高频率数据访问的场景，如实时系统、大规模数据处理、共享缓存。