### 1 点对点聊天

在四个进程ABCD中，使用一个messagequeue实现A与B，C与D的点对点聊天。进程AB不能看到CD的聊天内容，反之亦然。

#### 1.1 messagequeue中信息的数据结构

1. //消息结构体，包含类型、发送时间、消息
2. **typedef** **struct** msg\_struct {
3. **long** type;//消息类型
4. **char** time[TIME\_SIZE];//发送消息的时间
5. **char** message[MSG\_SIZE];//消息内容
6. } my\_msg;

如上所示，消息队列中的信息的数据结构是自己定义的my\_msg

第一部分是long类型的type，是消息结构体必须有的，表示本次发送的消息类型

第二部分是本次消息发送的时间

第三部分是本身消息的主题内容

#### 1.2 进程间通信的调用的API函数

1. //ftok函数，生成ipc的key，用于msgget的确定一个唯一的队列
2. //pathname是一个文件名，proj\_id与pathname共同确定一个key
3. //成功返回key，失败返回小于0的数
4. key\_t ftok (**const** **char** \*pathname, **int** proj\_id);
5. //根据key获取消息队列
6. //msgflg是标识符，可用于设定权限等，可通过IPC\_CREAT、IPC\_EXCL、00777指定
7. //成功返回队列id，失败返回-1
8. **int** msgget(key\_t key, **int** msgflg)
9. //根据消息队列id管理队列
10. //msgid是消息队列id，cmd是控制标志，有IPC\_STAT、IPC\_SET等
11. //buf是结构体msqid\_ds，根据cmd类型来设置队列信息或获取信息
12. //成功返回0，失败返回-1
13. **int** msgctl(**int** msqid, **int** cmd, **struct** msqid\_ds \*buf)
14. //向消息队列发送消息函数
15. //msqid是队列id，msgp是发送的消息结构体的指针
16. //msgsz是消息的大小（不包含消息类型长度），msgflg是设置发送消息的方式
17. //成功返回0，失败返回-1
18. **int** msgsnd(**int** msqid, **const** **void** \*msgp, **size\_t** msgsz, **int** msgflg)
19. //接收消息的函数
20. //msqid是队列的id，msgp是即将存放接收消息的结构体指针
21. //msgsz是想要接收消息的大小，msgtyp是想要接受的消息类型
22. //msgflg指定接收方式
23. //成功返回实际接受的消息长度，失败返回-1
24. ssize\_t msgrcv(**int** msqid, **void** \*msgp, **size\_t** msgsz, **long** msgtyp,**int** msgflg)

#### 1.3 设计思路

1. 四个进程使用一个消息队列，需要分辨不同的消息

2. 给每个进程设置特定的发送的消息类型和特定的接收类型

1. //判断是哪个用户
2. //设定特定发送与接收消息类型
3. **if** (strcmp(argv[1], "A") == 0) {
4. send\_type = CLIENT\_A;
5. recv\_type = CLIENT\_B;
6. is\_client = IS\_CLIENT\_TRUE;
7. }

3. A给B发送的消息类型是B可以接收的消息类型，其他进程相似

4. 发送和接收消息需要同时在线，所以采用父子进程协同工作，父进程接收用户输入发送消息，子进程接收消息队列的消息并打印。

1. //创建子进程开始接收消息
2. **if** ((pid = fork()) == 0) {
3. **while** (1) {
4. //接收消息队列消息
5. ......
6. }
7. } **else** {
8. //设置SIGINT处理函数
9. signal(SIGINT, sig\_handle\_parent);
10. **while** (1) {
11. //清空消息结构
12. memset(msg\_send.message, 0, MSG\_SIZE);
13. memset(msg\_send.time, 0, TIME\_SIZE);
14. //阻塞等待消息
15. fgets(msg\_send.message, MSG\_SIZE, stdin);
16. fflush(stdin);
17. ......
18. //发送消息
19. msgsnd(msqid, &msg\_send, MSG\_SIZE + TIME\_SIZE, 0);
20. ......
21. }
22. }

5. 进程接收Ctrl+C退出时，父进程先杀死子进程，然后自己再退出

1. //SIGINT处理函数，先杀死子进程，再退出父进程
2. **void** sig\_handle\_parent(**int** signo) {
3. kill(pid, SIGKILL);
4. wait(NULL);
5. exit(0);
6. }

#### 1.4 运行效果



如上图，A、B、C、D四个程序在互相通信，A和B可以相互通信，C和D可以相互通信，他们无法看见另外两个人的消息，实现了点对点聊天

### 2 群聊

利用share memory实现群聊功能。将进程接收的消息直接打印在屏幕上，当前进程输入的消息回车后发送给所有的进程。

#### 2.1 如何将信息写入share memory

1、先要使用shmget函数获取共享内存的指针

2、写入消息前，先要获取信号量，对共享内存进行唯一写者操作

3、将输入的信息使用memcpy函数将消息复制到共享内存中。如下：

1. memset(share\_point, 0, **sizeof**(my\_msg));
2. memcpy(share\_point, &msg\_send, **sizeof**(my\_msg));

4、写入完成之后，释放信号量，让其他进程可以写入

#### 2.2 share memory中的数据结构

1. //消息结构体，共享内存结构也和这个一样
2. **typedef** **struct** massage {
3. **int** read\_tag;//读取标志，只有在本地标志与共享标志相等时才可以读取
4. pid\_t pid;//pid，写入本次消息的pid
5. **char** time[TIME\_SIZE];//消息发送的时间
6. **char** name[NAME\_SIZE];//发送消息的用户名称
7. **char** message[MSG\_SIZE];//消息的主体内容
8. } my\_msg;

如上所示，共享内存中的数据结构为自定义的结构体my\_msg。

包含了消息的发送者、时间、消息主体等信息

#### 2.3 设计思路

1、所有人都可以收到消息，所以使用同一个共享内存

2、为了同时可以发送与接收消息，采用父子进程协同工作

3、发送消息后，每个人都可以收到消息，为了避免收到重复消息，使用参数read\_tag，只有当本地read\_tag和共享内存的read\_相等时才可以读取信息，每次读取之后，read\_tag加一，发送方发送消息时，会将共享内存的tag加一，如下所示

1. //判断read\_tag是否和本身read\_tag相同，是的话说明有新的消息
2. **if** (share\_point->read\_tag == read\_tag) {
3. read\_tag = read\_tag + 1;
4. //如果pid与父进程pid不同才打印消息
5. **if** (share\_point->pid != ppid) {
6. printf(" %s" share\_point->message);
7. }
8. }

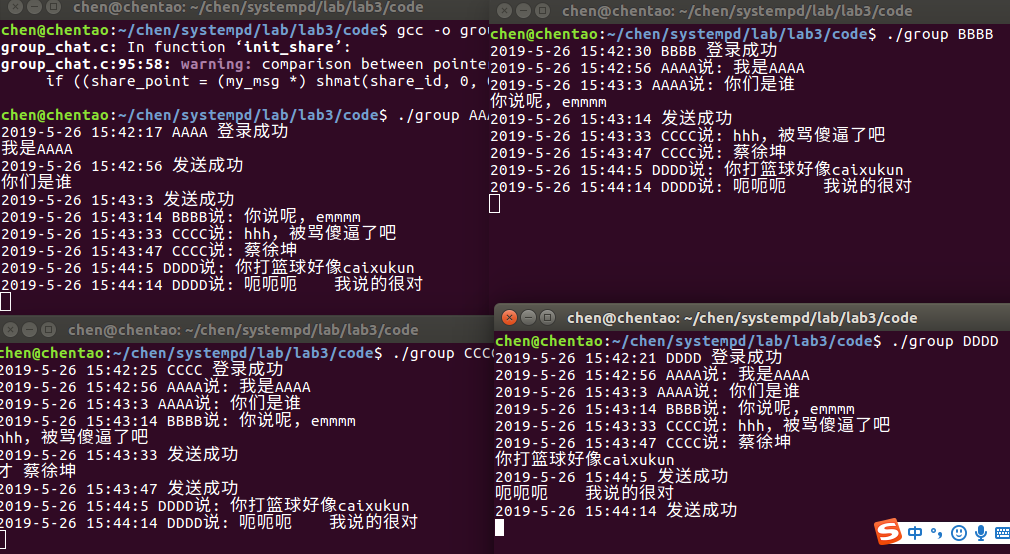
4、为了避免接收到自己发送的消息，在共享内存中存入发送消息的进程pid，接收消息时，子进程判断该pid是否与父进程的pid相同，如果相同，说明是自己发送的消息，则不接受；如果不同，则接收打印消息

5：为了实现唯一写者，采用信号量，在某个进程即将执行写操作之前，获取信号量，对写进行加锁，在写完之后，进行解锁。

6、避免读取的时候内容被修改，所以读取的时候也会获取信号量进行加锁，读取完毕之后进行解锁

7、程序接收CTRL+C退出时，会让子进程与共享内存解绑，杀死子进程，之后父进程与共享内存解绑，判断与共享内存相连的进程数，如果为0，则删除共享内存块，之后父进程退出。

#### 2.4 效果展示



如上所示，每个进程都可以看见其他进程发送的消息