Lab3 文档

16302010015　　曹家俊

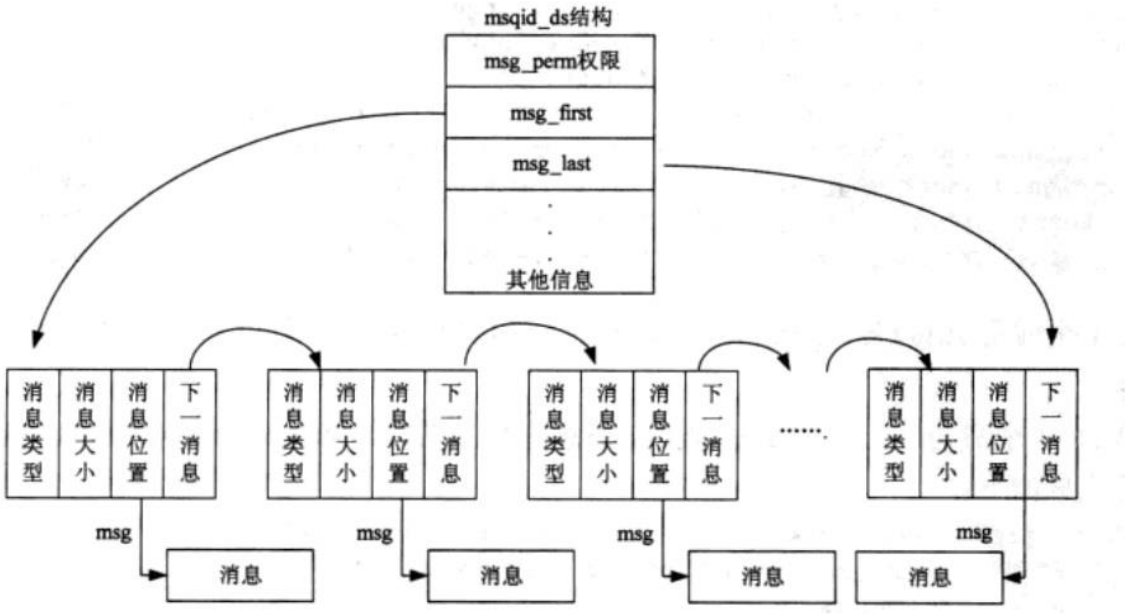
1. 点对点聊天

1.1 messagequeue 中信息的数据结构

①msqid_ds：该结构被系统内核用来保存消息队列对象的有关数据。每个消息队列都有一个 msqid_ds 结构与其相关联。

```
struct msqid_ds {
    struct ipc_perm msg_perm;    /*  见③ipc_perm */
    struct msg *msg_first;          /* first message on queue,unused */
    struct msg *msg_last;           /* last message in queue,unused */
    __kernel_time_t msg_stime;    /* last msgsnd time */
    __kernel_time_t msg_rtime;    /* last msgrcv time */
    __kernel_time_t msg_ctime;    /* last change time */
    unsigned long    msg_lcbytes;    /* Reuse junk fields for 32 bit */
    unsigned long    msg_lqbytes;    /* ditto */
    unsigned short msg_cbytes;    /* current number of bytes on queue */
    unsigned short msg_qnum;       /* number of messages in queue */
    unsigned short msg_qbytes;    /* max number of bytes on queue */
    __kernel_ipc_pid_t msg_lspid;      /* pid of last msgsnd */
    __kernel_ipc_pid_t msg_lrpid;      /* last receive pid */
};
```

消息队列组织图：



②msgbuf：该结构用来存储消息队列中消息的数据类型和内容

```
struct msgbuf {
        long type;      /* POSITIVE message type */
        char text[MAX_LEN];    /* message data */
};
```

③ipc_perm：该结构用来保存每个 IPC 对象权限信息

```
struct ipc_perm{
        key_t key;
        ushort uid;
        ushort gid;
        ushort cuid;    /* IPC 对象创建者的 uid */
        ushort cgid;
        ushort mode;    /* IPC 对象的存取权限*/
        ushort seq;
};
```

1.2 进程间通信的 API 函数

①msgget 函数：int msgget(key_t key,int flag);

这个函数可以创建一个新的消息队列，也可以打开一个已经存在的消息队列，这取决于 key 和 flag 的值，函数执行成功时会返回消息队列的引用标识符，否则返回-1。当一个新的消息队列创建时，与之对应的 msqid_ds 结构也会被初始化。

②msgsnd 函数：int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);

返回值：若成功则返回 0，若出错则返回-1。

参数 ptr 指向一个长整型数，它包含了正的整型消息类型，在其后紧跟着消息数据。（若 nbytes 是 0，则无消息数据。）参数 flag 的值可以指定为 IPC_NOWAIT：若消息队列已满（或者是队列中的消息总数等于系统限制值，或队列中的字节总数等于系统限制值），指定 IPC_NOWAIT 可使 msgsnd 立即出错返回 EAGAIN。如果没有指定 IPC_NOWAIT，则进程阻塞直到下述情况出现为止：有空间可以容纳要发送的消息；从系统中删除了此队列；或捕捉到一个信号，并从信号处理程序返回。当 msgsnd 函数成功返回时，与消息队列相关的 msqid_ds 结构得到更新，以表明发出该调用的进程 ID（msg_lspid）、进行该调用的时间（msg_stime），并指示队列中增加了一条消息（msg_qnum）。

③msgrcv 函数：ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
返回值：若成功则返回消息的数据部分的长度，若出错则返回-1。
如同 msgsnd 中一样，ptr 参数指向一个长整型数（返回的消息类型存放在其中），跟随其后的是存放实际消息数据的缓冲区。nbytes 说明数据缓冲区的长度。若返回的消息大于 nbytes，而且在 flag 中设置了 MSG_NOERROR，则该消息被截断。如果没有设置这一标志，而消息又太长，则出错返回 E2BIG（消息仍留在队列中）。type 指定想要哪一种消息。

④msgctl 函数：int msgctl(int msqid, int cmd, struct msqid_ds *buf);
返回值：若成功则返回 0，若出错则返回-1。
cmd 参数说明对由 msqid 指定的队列要执行的命令：
IPC_STAT：取此队列的 msqid_ds 结构，并将它存放在 buf 指向的结构中。
IPC_SET：将 buf 指向的结构复制到与这个队列相关的 msqid_ds 结构中。
IPC_RMID：从系统中删除该消息队列以及仍在该队列中的所有数据，这种删除立即生效。

1.3 设计思路

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 和 CD 之间不能看到对方的聊天内容。

①使用一个 messagequeue 实现点对点通信

为了实现用一个 messagequeue 实现点对点通信，可以将 AB 和 CD 放入不同的组（group），在各自的组中（例如 AB），A 和 B 有不同的序列号（index）。通过 group 和 index 共同确定某一进程发送消息的 type，以及它应该接收的消息的 type（即 A 只能接收 B 发送的 type，以此类推）。具体地，进程 A 的 group 为 1，index 为 1；B 的 group 为 1，index 为 2；C 的 group 为 2，index 为 1；D 的 group 为 2，index 为 2。

定义某一进程发送消息的类型 **snd_type = group * 2 + index;**

应该接收的消息类型 **rcv_type = index == 1 ? (group * 2 + 2) : (group * 2 + 1);**

在 msgsnd 时发送 snd_type 类型的消息，在 msgrcv 时通过 rcv_type 鉴别当前进程需要接收的消息类型。这样的话，就可以只使用一个 messagequeue 实现 A 和 B 通信，C 和 D 通信且 AB 和 CD 互不干扰。

②一边等待接收消息，一边发送消息

fork 一个子进程专门等待 msgrcv（接收消息），父进程（当前进程）等待用户输入（发消息）。

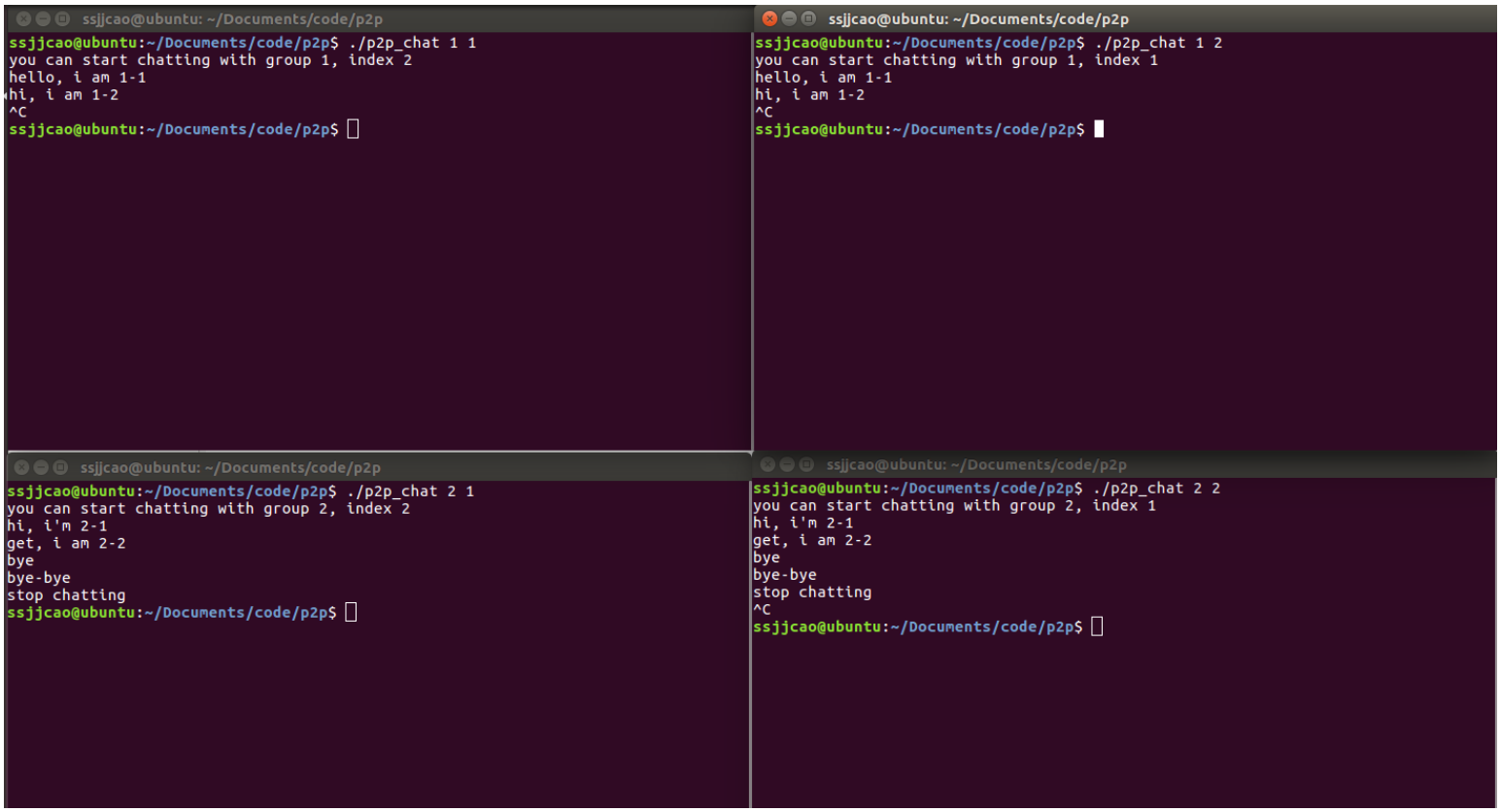
③运行说明

./p2p_chat [group] [index]

其中 group 为大于 1 的整数，index 只能为 1 或 2，只有同一个 group 的两个进程才能互相通信。

输入“stop chatting”可以结束聊天。

运行截图：



2. 群聊

2.1 如何将信息写入 share memory

使用 mempy 方法将信息写入 share memory

```
memcpy(shm_addr, message_to_send, SHM_SIZE);
```

其中，shm_addr 为通过 shmat 获取的共享存储地址，message_to_send 为需要写入的信息。

2.2 share memory 中的数据结构

```
struct shmid_ds {
        struct ipc_perm shm_perm;       /* ownership and permissions */
        size_t    shm_segsz;      /* size of segment (bytes) */
```

```
    time_t    shm_atime;     /* last attach time */
    time_t    shm_dtime;     /* last detach time */
    time_t    shm_ctime;     /*last change time */
    pid_t    shm_cpid;       /* pid of creator */
    pid_t    shm_lpid;       /* pid of last shmop() */
    shmatt_t    shm_nattch;    /* number of current attaches */
    …
};
```

内核为每个共享存储段维护一个结构，即上述的 shmid_ds 结构。

2.3 设计思路

用共享存储实现群聊功能，首先要创建各个进程群聊时使用的共享存储标识符。因为考虑到：若一个进程正在向共享内存区写数据，则在它做完这一步操作前，别的进程不应当去读、写这些数据。所以还要注意多个进程对共享存储区访问的互斥（本次 lab 中，我通过信号量实现，定义两个信号量值，序号为 0 的 EMPTY 和序号为 1 的 FULL）。那么，一开始除了创建共享存储标识符，还要创建信号量标识符。

整体的流程设计思路：group_master 创建共享存储标识符和信号量标识符，并初始化信号量（EMPTY 的值为 1，FULL 的值为 0），然后 group_master 等待 FULL 控制的读资源>0。在 group_master 做完初始化工作后，开启多个进程来互相通信（即 group_chat）。当某进程要发送消息时，它会等待 EMPTY，即企图获取 EMPTY 控制的写资源。发送成功后释放 FULL 控制的读资源。此时一直处于等待状态下的 group_master 可以读取共享存储中该进程新发送的消息（获得了 FULL 管理的读资源），读完之后释放 EMPTY 控制的写资源（即各个进程又可以发消息到共享存储中）。每次 group_master 将获取的新消息广播给所有正在通信的所有进程。广播的实现：通过 kill 发送 SIGUSR1 信号给所有在线的进程（通过一个文件管理在线进程，每次有一个新的进程加入，则将其 pid 加入文件中）。

运行说明：

lock.[h|c] 实现信号量的 lock -1 和 unlock +1

group_master.c 群聊管理，负责初始化共享存储标识符和信号量标识符，以及退出（通过 Ctrl+C 退出）时释放资源。

group_chat.c 进行群聊 通过./group_chat [name]运行，输入 send 后可以发送消息，发送完消息后，该消息会显示所有在线的聊天进程中，并显示谁发送的。

运行截图：

编译：gcc group_master.c lock.c -o group_master

gcc group_chat.c lock.c -o group_chat