第五天消息队列与共享内存

一、回顾知识点

1.1 管道PIPE

> 道(pipe)又称无名管道，由`int fd[2]`来应用，`fd[0]`读， `fd[1]`写

```
#include <unistd.h>
int pipe(int filedes[2]);  创建pipe
```

特点：

1、半双工，数据在同一时刻只能在一个方向上流动。
2、数据只能从管道的一端写入，从另一端读出。
3、写入管道中的数据遵循先入先出的规则。
4、管道所传送的数据是无格式的，这要求管道的读出方与写入方必须事先约定好数据的格式，如多少字节算一个消息等。
5、管道不是普通的文件，不属于某个文件系统，其只存在于内存中。
6、管道在内存中对应一个缓冲区。不同的系统其大小不一定相同。
7、从管道读数据是一次性操作，数据一旦被读走，它就从管道中被抛弃，释放空间以便写更多的数据。
8、管道没有名字，只能在具有公共祖先的进程之间使用

读写数据的特点

```
1、默认用 read 函数从管道中读数据是阻塞的。
2、调用 write 函数向管道里写数据，当缓冲区已满时 write 也会阻塞。
3、通信过程中，读端口全部关闭后，写进程向管道内写数据时，写进程会
（收到 SIGPIPE 信号）退出。
```

管道的缓冲区的大小：`64K`

【注意】当父进程结束时，子进程无法从标准输入终端获取数据了。

1.2 文件描述符重定向

每个进程都有一张文件描述符的表，进程刚被创建时，标准输入、标准输出、标准错误输出设备文件被打开，对应的文件描述符 0、1、2 记录在表中。

在进程中打开其他文件时，系统会返回文件描述符表中最小可用的文件描述符，并将此文件描述符记录在表中

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

dup 和 dup2 经常用来重定向进程的 stdin、stdout 和 stderr。

1.2.1 dup

> `int dup(int oldfd);`
>
> 复制 oldfd 文件描述符，并分配一个新的文件描述符，新的文件描述符是调用进程文件描述符表中最小可用的文件描述符。
>
> 成功：新文件描述符。
> 失败：返回 - 1，错误代码存于 errno 中

1.2.2 dup2

> `int dup2(int oldfd, int newfd)`
>
> 复制一份打开的文件描述符 oldfd，并分配新的文件描述符 newfd，newfd 也标识 oldfd 所标识的文件。
>
> oldfd 要复制的文件描述符，newfd 分配的新的文件描述符。
>
> 成功：返回 newfd
> 失败：返回-1，错误代码存于 errno 中

【注意】newfd 是小于文件描述符最大允许值的非负整数，如果 newfd 是一个已经打开的文件描述符，则首先关闭该文件，然后再复制。

1.3 复制的新旧文件描述符的特点

复制文件描述符后新旧文件描述符的特点:

```
1）使用 dup 或 dup2 复制文件描述符后，新文件描述符和旧文件描述符指向同一个文件，共享文件锁定、读写位置和各项权限。
2）当关闭新的文件描述符时，通过旧文件描述符仍可操作文件。
3）当关闭旧的文件描述时，通过新的文件描述符仍可可操作文件。
```

exec 前后文件描述符的特点:

```
1) close_on_exec 标志决定了文件描述符在执行 exec 后文件描述符是否可用。

2) 文件描述符的 close_on_exec 标志默认是关闭的， 即文件描述符在执行 exec后文件描述符是可用的。

3) 若没有设置 close_on_exec 标志位，进程中打开的文件描述符，及其相关的设置在 exec 后不变，可供新启动的程序使用
```

设置 close_on_exec 标志位的方法:

```
int flags;
flags = fcntl(fd, F_GETFD); //获得文件描述符标志
flags |= FD_CLOEXEC;        //打开标志位
flags &= ~FD_CLOEXEC;       //关闭标志位
fcntl(fd, F_SETFD, flags);  //设置标志
```

1.4 命名管道(FIFO)

> 命名管道(FIFO)和管道(pipe)基本相同，但也有一些显著的不同。

特点：

1、半双工，数据在同一时刻只能在一个方向上流动。
2、写入 FIFO 中的数据遵循先入先出的规则。
3、FIFO 所传送的数据是无格式的，这要求 FIFO 的读出方与写入方必须事先约定好数据的格式，如多少字节算一个消息等。
4、FIFO 在文件系统中作为一个特殊的文件而存在，但 FIFO 中的内容却存放在内存中。
5、管道在内存中对应一个缓冲区。不同的系统其大小不一定相同。
6、从 FIFO 读数据是一次性操作，数据一旦被读，它就从 FIFO 中被抛弃，释放空间以便写更多的数据。
7、当使用 FIFO 的进程退出后，FIFO 文件将继续保存在文件系统中以便以后使用。
8、FIFO 有名字，不相关的进程可以通过打开命名管道进行通信

操作FIFO文件时的特点:

> 系统调用的 I/O 函数都可以作用于 FIFO，如 open、close、read、write 等。
> 打开 FIFO 时，非阻塞标志(O_NONBLOCK)产生下列影响。

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

int mkfifo(const char *pathname, mode_t mode);
```

参数： pathname文件路径名，mode是文件的读写执行（r,w,x）相关的权限（可以使用0ddd数值表示，如0755）。

返回值： 0成功， 非0失败。

【注意】一个有名管道只能创建一次

1.4.1 特点1-不指定 O_NONBLOCK

> 不指定 O_NONBLOCK(即 open 没有位或 O_NONBLOCK)

```
1、open 以只读方式打开 FIFO 时，要阻塞到某个进程为写而打开此 FIFO
2、open 以只写方式打开 FIFO 时，要阻塞到某个进程为读而打开此 FIFO。
3、open 以只读、只写方式打开 FIFO 时会阻塞，调用 read 函数从 FIFO 里读数据时 read 也会阻塞。
4、通信过程中若写进程先退出了，则调用 read 函数从 FIFO 里读数据时不阻塞；若写进程又重新运行，则调用 read 函数从 FIFO 里读数据时又恢复阻塞。
5、通信过程中，读进程退出后，写进程向命名管道内写数据时，写进程也会（收到 SIGPIPE 信号）退出。
6、调用 write 函数向 FIFO 里写数据，当缓冲区已满时 write 也会阻塞。
```

1.4.2 特点2-指定 O_NONBLOCK

> 指定 O_NONBLOCK(即 open 位或 O_NONBLOCK)

```
1、先以只读方式打开：如果没有进程已经为写而打开一个 FIFO，只读open成功，并且open不阻塞。
2、先以只写方式打开：如果没有进程已经为读而打开一个 FIFO，只写 open 将出错返回-1。
3、read、write 读写命名管道中读数据时不阻塞。
4、通信过程中，读进程退出后，写进程向命名管道内写数据时，写进程也会（收到SIGPIPE 信号）退出
```

1.4.3 可读可写方式打开 FIFO 文件时的特点

```
1、open 不阻塞。
2、调用 read 函数从 FIFO 里读数据时 read 会阻塞。
3、调用 write 函数向 FIFO 里写数据，当缓冲区已满时 write 也会阻塞。
```

二、消息队列

2.1 消息队列的概述

消息队列是消息的链表，存放在内存中，由内核维护。

消息队列的特点：

1、消息队列中的消息是有类型的。

2、消息队列中的消息是有格式的。

3、消息队列可以实现消息的随机查询。消息不一定要以先进先出的次序读取，编程时可以按消息的类型读取。

4、消息队列允许一个或多个进程向它写入或者读取消息。

5、与无 名管道、命名管道一样，从消息队列中读出消息，消息队列中对应的数据都会被删除。

6、每个消息队列都有消息队列标识符，消息队列的标识符在整个系统中是唯一的。

7、只有内核重启或人工删除消息队列时，该消息队列才会被删除。若不人工删除 消息队列，消息队列会一直存在于系统中。

在 ubuntu 相关版本（12）的消息队列限制值如下:

- 每个消息内容最多为 `8K` 字节
- 每个消息队列容量最多为 `16K` 字节
- 系统中消息队列个数最多为 `1609` 个
- 系统中消息个数最多为 `16384` 个

2.2 ftok 函数

> System V 提供的 IPC 通信机制需要一个 key 值，通过 key 值就可在系统内获得一个唯一的消息队列标识符。key 值可以是人为指定的，也可以通过 `ftok` 函数获得。

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(const char *pathname, int proj_id);
```

功能： 获得项目相关的唯一的 IPC 键值。

参数：

- pathname：路径名
- `proj_id`：项目 ID，非 0 整数(只有低 8 位有效：0~255)

返回值：成功返回 key 值，失败返回 -1

如：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>

int main(int argc, char const *argv[])
{
    key_t key = ftok("/", 100);
    printf("key = %d\n", key);

    key_t key2 = ftok("/", 200);
    printf("key2 = %d\n", key2);

    return 0;
}
```

```
disen@qfxa:~/code3/day05$ gcc ftok1.c
disen@qfxa:~/code3/day05$ ./a.out
key = 1677787138
key2 = -939458558
disen@qfxa:~/code3/day05$ ./a.out
key = 1677787138
key2 = -939458558
disen@qfxa:~/code3/day05$
```

2.3 消息队列的操作

2.3.1 创建消息队列

```
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

功能:
　创建一个新的或打开一个已经存在的消息队列。不同的进程调用此函数，只要用相同的 key 值就能得到同一个消息队列的标识符。
参数:

- key: IPC 键值。
- `msgflg`：标识函数的行为及消息队列的权限。其取值：

- `IPC_CREAT`：创建消息队列。
- `IPC_EXCL`：检测消息队列是否存在。
- 位或权限位：消息队列位或权限位后可以设置消息队列的访问权限，格式和 open 函数的 mode_t 一样，但可执行权限未使用。

返回值：成功则返回消息队列的标识符，失败则返回-1。

如：

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int main(int argc, char const *argv[])
{
    key_t key = ftok("/", 200);
    int msgqid = msgget(key, IPC_CREAT | 0666);
    if (msgqid == -1)
    {
        perror("msgget");
        return 1;
    }
    printf("msgqid: %d\n", msgqid);
    return 0;
}
```

```
disen@qfxa:~/code3/day05$ ./a.out
msgqid: 0
```

使用 shell 命令操作消息队列

- `ipcs -q` 查看消息队列
- `ipcrm -q msqid` 删除消息队列

```
disen@qfxa:~/code3/day05$ ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0xc8010002 0          disen      666        0            0
```

```
disen@qfxa:~/code3/day05$ ipcrm -q 0
disen@qfxa:~/code3/day05$ ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

2.3.2 发送消息

消息队列的消息的格式:

```
typedef struct _msg
{
    long mtype;       /*消息类型*/
    char mtext[100]; /*消息正文*/
    ...               /*消息的正文可以有多个成员*/
} MSG;
```

消息类型必须是长整型的，而且必须是结构体类型的第一个成员，之后的是消息正文，正文可以有多个成员（正文成员可以是任意数据类型的）。

发送消息的函数:

> 将新消息添加到消息队列

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

参数:

- `msqid`: 消息队列的标识符
- `msgp`: 待发送消息结构体的地址
- `msgsz`: 消息正文的字节数(除了第一个成员long的大小之外的所有成员的大小总和)
- `msgflg`: 函数的控制属性
    - `0`: 阻塞（达到消息队列的最大容量时，则阻塞）直到条件满足为止
    - `IPC_NOWAIT`: 立即返回非阻塞

返回值： 成功 0；失败 -1。

如:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

typedef struct msg_
{
    long mType;         // 消息类型
    char content[100]; // 正文数据
    char title[32];    // 数据标题
} MSG;

int main(int argc, char const *argv[])
{
    key_t key = ftok("/", 200);
    int msgqid = msgget(key, IPC_CREAT | 0666);
    if (msgqid == -1)
    {
```

```
        perror("msgget");
        return 1;
    }
    printf("msgqid: %d\n", msgqid);

    // 发送数据（消息）
    MSG msg1;
    msg1.mType = 1;
    strcpy(msg1.content, argv[1]);
    strcpy(msg1.title, argv[2]);
    int ret = msgsnd(msgqid, &msg1, sizeof(MSG) - sizeof(long), 0);
    if (ret == 0)
    {
        printf("msg1 发送成功\n");
    }
    else
    {
        perror("msgsnd");
    }
    return 0;
}
```

```
disen@qfxa:~/code3/day05$ ./a.out hello test1
msgqid: 32768
msg1 发送成功
disen@qfxa:~/code3/day05$ ./a.out hello2 test2
msgqid: 32768
msg1 发送成功
disen@qfxa:~/code3/day05$ ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0xc8010002 32768      disen      666        408          3
```

如2： 优化上面的代码，消息类型也从命令行参数中获取

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdlib.h>

typedef struct msg_
{
    long mType;        // 消息类型
    char content[100]; // 正文数据
    char title[32];    // 数据标题
} MSG;

int main(int argc, char const *argv[])
{
    if (argc != 4)
    {
        printf("用法错误,正确格式: ./send 消息类型 正文 标题\n");
```

```
        return 1;
    }

    key_t key = ftok("/", 200);
    int msgqid = msgget(key, IPC_CREAT | 0666);
    if (msgqid == -1)
    {
        perror("msgget");
        return 1;
    }
    printf("msgqid: %d\n", msgqid);

    // 发送数据（消息）
    MSG msg1;
    msg1.mType = atol(argv[1]);
    strcpy(msg1.content, argv[2]);
    strcpy(msg1.title, argv[3]);
    int ret = msgsnd(msgqid, &msg1, sizeof(MSG) - sizeof(long), 0);
    if (ret == 0)
    {
        printf("msg1 发送成功\n");
    }
    else
    {
        perror("msgsnd");
    }
    return 0;
}
```

```
disen@qfxa:~/code3/day05$ ./send 2 no test3
msgqid: 32768
msg1 发送成功
disen@qfxa:~/code3/day05$ ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0xc8010002 32768      disen      666        680          5
```

2.3.3 接收消息

> 从标识符为 `msqid` 的消息队列中接收一个消息。
>
> 一旦接收消息成功，则消息在消息队列中被删除。

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

参数:

- `msqid`：消息队列的标识符，代表要从哪个消息列中获取消息。
- `msgp`： 存放消息结构体的地址。
- `msgsz`：消息正文的字节数。

- `msgtyp`：消息的类型、可以有以下几种类型
  - `msgtyp` = 0：返回队列中的第一个消息
  - `msgtyp` > 0：返回队列中消息类型为 `msgtyp` 的消息
  - `msgtyp` < 0：返回队列中消息类型值小于或等于 `msgtyp` 绝对值的消息，如果这种消息有若干个，则取类型值最小的消息
- `msgflg`：函数的控制属性
  - `0`： `msgrcv` 调用阻塞(空消息队列)直到接收消息成功为止。
  - `MSG_NOERROR`:若返回的消息字节数比 `msgsz` 字节数多,则消息就会截短到 `msgsz` 字节,且不通知消息发送进程。
  - `IPC_NOWAIT`:调用进程会立即返回。若没有收到消息则立即返回-1

返回值： 成功返回读取消息的长度，失败返回-1。

如：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdlib.h>

typedef struct msg_
{
    long mType;         // 消息类型
    char content[100]; // 正文数据
    char title[32];     // 数据标题
} MSG;

int main(int argc, char const *argv[])
{

    key_t key = ftok("/", 200);
    int msgqid = msgget(key, IPC_CREAT | 0666);
    if (msgqid == -1)
    {
        perror("msgget");
        return 1;
    }

    MSG msg; // 接收消息的结构体对象
    long msgtype = atol(argv[1]);
    ssize_t len = msgrcv(msgqid, &msg, sizeof(MSG) - sizeof(long), msgtype, 0);
    if (len == -1)
    {
        perror("msgrcv");
        return 1;
    }
    printf("接收的数据(%ld): %s(%s)\n", len, msg.title, msg.content);

    return 0;
}
```

2.3.4 消息队列的控制

> 对消息队列进行各种控制，如修改消息队列的属性，或删除消息消息队列。

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

参数:

- `msqid`：消息队列的标识符。
- `cmd`：函数功能的控制。
  - `IPC_RMID`：删除由 msqid 指示的消息队列，将它从系统中删除并破坏相关数据结构。
  - `IPC_STAT`：将 msqid 相关的数据结构中各个元素的当前值存入到由 buf 指向的结构中。
  - `IPC_SET`：将 msqid 相关的数据结构中的元素设置为由 buf 指向的结构中的对应值
- `buf`：`msqid_ds` 数据类型的地址，用来存放或更改消息队列的属性

```
struct msqid_ds
{
  struct ipc_perm msg_perm; /* structure describing operation permission */
  __time_t msg_stime;       /* time of last msgsnd command */
#ifndef __x86_64__
  unsigned long int __glibc_reserved1;
#endif
  __time_t msg_rtime;       /* time of last msgrcv command */
#ifndef __x86_64__
  unsigned long int __glibc_reserved2;
#endif
  __time_t msg_ctime;       /* time of last change */
#ifndef __x86_64__
  unsigned long int __glibc_reserved3;
#endif
  __syscall_ulong_t __msg_cbytes; /* current number of bytes on queue */
  msgqnum_t msg_qnum;       /* number of messages currently on queue */
  msglen_t msg_qbytes;      /* max number of bytes allowed on queue */
  __pid_t msg_lspid;        /* pid of last msgsnd() */
  __pid_t msg_lrpid;        /* pid of last msgrcv() */
  __syscall_ulong_t __glibc_reserved4;
  __syscall_ulong_t __glibc_reserved5;
};
```

返回值： 成功 0；失败 -1

如：

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

typedef struct msg_
{
    long mType;         // 消息类型
    char content[100]; // 正文数据
    char title[32];    // 数据标题
} MSG;

int main(int argc, char const *argv[])
{

    key_t key = ftok("/", 200);
    int msgqid = msgget(key, IPC_CREAT | 0666);
    if (msgqid == -1)
    {
        perror("msgget");
        return 1;
    }

    // 查看msgqid消息队列的最后发送时间和用户权限
    struct msqid_ds ds;
    int ret = msgctl(msgqid, IPC_STAT, &ds);
    if (ret == -1)
    {
        perror("msgctl");
        return 1;
    }

    time_t st = ds.msg_stime; // 时间戳
    struct tm *st_time = localtime(&st);
    printf("%d 最后发送消息的时间: %d-%02d-%02d %02d:%02d:%02d\n", msgqid,
           st_time->tm_year + 1900,
           st_time->tm_mon + 1,
           st_time->tm_mday,
           st_time->tm_hour,
           st_time->tm_min,
           st_time->tm_sec);

    printf("%d 消息队列的权限：%04o\n", msgqid, ds.msg_perm.mode);

    // 修改消息队列的权限为755
    ds.msg_perm.mode = 0755;
    if (msgctl(msgqid, IPC_SET, &ds) == 0)
    {
        printf("修改权限 755 成功\n");
```

```
    }
    execlp("ipcs", "ipcs", "-q", NULL);
    return 0;
}
```

```
disen@qfxa:~/code3/day05$ ./a.out
32768 最后发送消息的时间: 2023-08-18 10:48:06
32768 消息队列的权限: 0777
修改权限 755 成功

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0xc8010002 32768      disen      755        136          1
```

2.4 综合练习

> 消息队列实现多人聊天程序

消息结构体类型

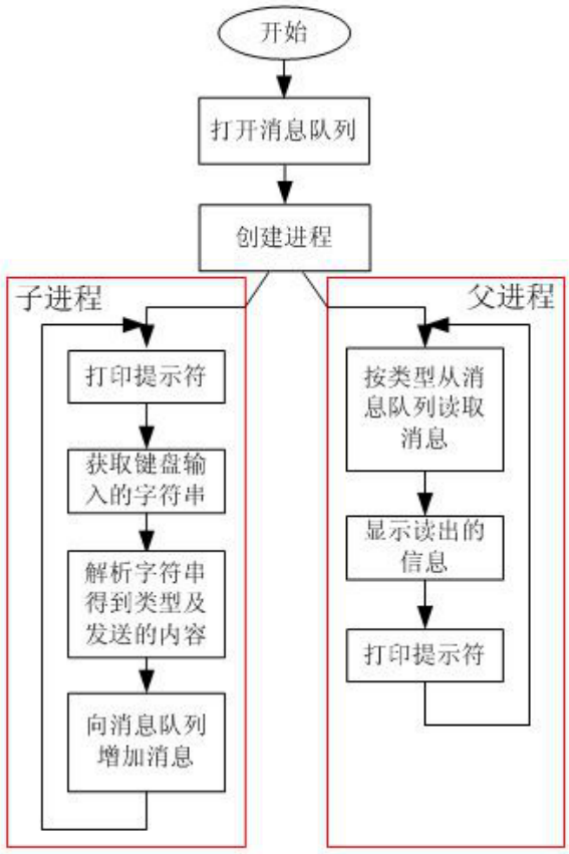
```
typedef struct msg
{
    long type; //接收者类型
    char text[100]; //发送内容
    char name[20]; //发送者姓名
}MSG;
```

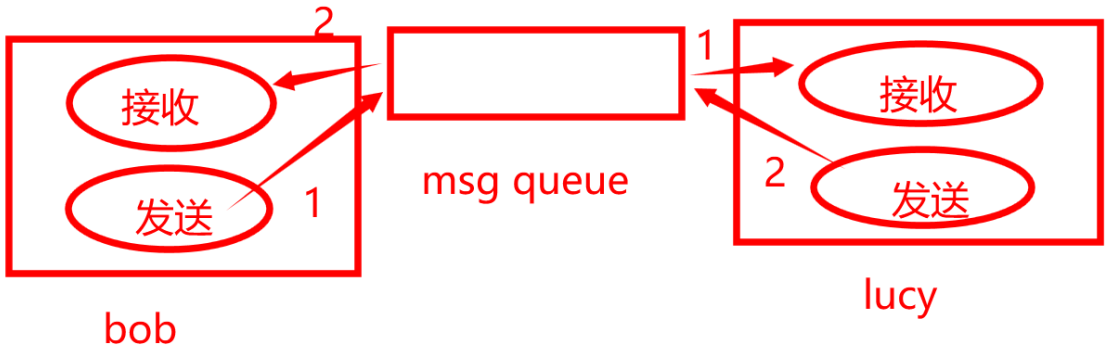
每个程序都有两个任务，一个任务是负责接收消息，一个任务是负责发送消息，通过 fork 创建子进程实现多任务。

一个进程负责接收信息，它只接收某种类型的消息，只要别的进程发送此类型的消息，此进程就能收到。收到后通过消息的 name 成员就可知道是谁发送的消息。

另一个进程负责发信息，可以通过输入来决定发送消息的类型。

设计程序的时候，接收消息的进程接收消息的类型不一样，这样就实现了发送的消息只被接收此类型消息的人收到，其它人收不到。这样就是实现了给特定的人发送消息。





如： mq_bbs.c

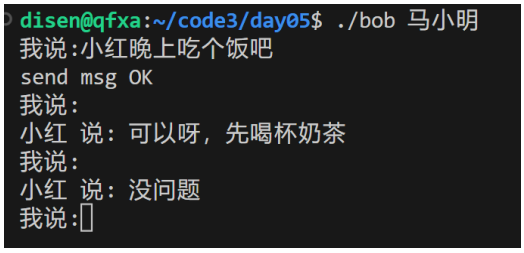
```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>

typedef struct msg_
{
    long mType;         // 消息类型
    char content[100]; // 正文数据
    char name[32];      // 姓名
} MSG;

int main(int argc, char const *argv[])
```

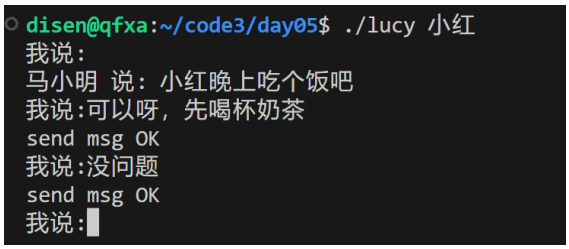
```
{
    if (argc != 2)
    {
        printf("invalid, format is : ./xx name\n");
        return 1;
    }

    key_t key = ftok("/", 160);
    int msgqid = msgget(key, IPC_CREAT | 0666);
    if (msgqid == -1)
    {
        perror("msgget");
        return 1;
    }
    int pid = fork();
    if (pid == 0)
    {
        // 子进程
        while (1)
        {
            write(STDOUT_FILENO, "我说:", 7);
            MSG msg;
            scanf("%s", msg.content);
            strcpy(msg.name, argv[1]);
#ifdef BOB
            msg.mType = 1;
#else
            msg.mType = 2;
#endif
            if (msgsnd(msgqid, &msg, sizeof(MSG) - sizeof(long), 0) != -1)
            {
                printf("send msg OK\n");
                if (strncmp(msg.content, "bye", 3) == 0)
                {
                    break;
                }
            }
        }
        // kill(getppid(), SIGSTOP);
        _exit(0);
    }
    else if (pid > 0)
    {
        // 父进程
        while (1)
        {
            MSG msg;
#ifdef BOB
            msg.mType = 2;
#else
            msg.mType = 1;
#endif
            if (msgrcv(msgqid, &msg, sizeof(MSG) - sizeof(long), msg.mType, 0)
!= -1)
            {
                printf("\n%s 说: %s\n", msg.name, msg.content);

                if (strncmp(msg.content, "bye", 3) == 0)
```

```
                {
                    break;
                }
                write(STDOUT_FILENO, "我说:", 7);
            }
        }
        // kill(pid, SIGSTOP);
        wait(NULL);
    }

    return 0;
}
```

两次编译：

```
gcc mq_bbs.c -o bob -D BOB
gcc mq_bbs.c -o lucy
```





三、共享内存

3.1 共享内存概述

> 共享内存允许两个或者多个进程共享给定的存储区域

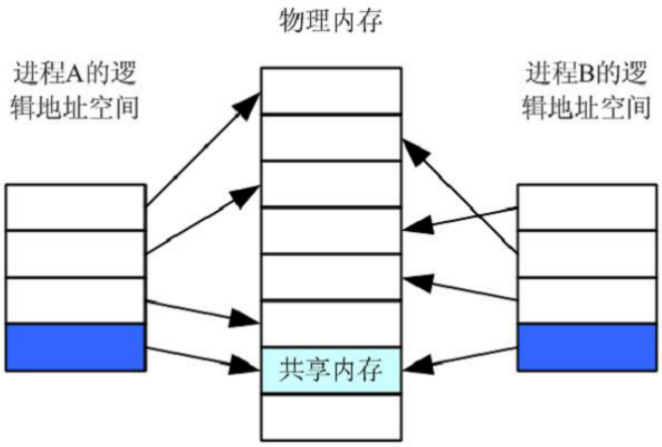
共享内存的特点:

1、 共享内存是进程间共享数据的一种最快的方法。

一个进程向共享的内存区域写入了数据，共享这个内存区域的所有进程就可以立刻看到其中的内容。

2、使用共享内存要注意的是多个进程之间对一个给定存储区访问的互斥

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。



在 ubuntu 相关版本中（12+）中共享内存限制值如下：

1、共享存储区的最小字节数：`1`
2、共享存储区的最大字节数：`32M`
3、共享存储区的最大个数：`4096`
4、每个进程最多能映射的共享存储区的个数：`4096`

Linux命令查看共享内存限制：

```
ipcs -l
```

3.2 共享内存操作

3.2.1 获得共享存储标识符

> 创建或打开一块共享内存区

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

参数:

- key：IPC 键值
- size：该共享存储段的长度(字节)
- shmflg：标识函数的行为及共享内存的权限。
  - IPC_CREAT：如果不存在就创建
  - IPC_EXCL：如果已经存在则返回失败
  - 位或权限位：共享内存位或权限位后可以设置共享内存的访问权限，格式和 open 函数的 mode_t 一样，但可执行权限未使用。

返回值: 成功，返回共享内存标识符(shmid)；失败，返回 - 1。

使用 shell 命令操作共享内存:

- 查看共享内存 `ipcs -m`
- 删除共享内存 `ipcrm -m shmid`

如：

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

int main(int argc, char const *argv[])
{
    // IPC键
    key_t key = ftok("/", 28);

    // 获取共享内存的标识
    int shmid = shmget(key, 32, IPC_CREAT | 0666);
    if (shmid == -1)
    {
        perror("shmget");
        return 1;
    }
    printf("成功获取共享内存空间: %d\n", shmid);

    return 0;
}
```

```
disen@qfxa:~/code3/day05$ ./a.out
成功获取共享内存空间: 3112973
disen@qfxa:~/code3/day05$ ipcs -m
```

3.2.2 共享内存映射

> 共享内存映射(attach), 将一个共享内存段映射到调用进程的数据段中

```c
#include <sys/types.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr,int shmflg);
```

参数:

- shmid：共享内存标识符。
- shmaddr：共享内存映射地址(若为 NULL 则由系 统自动指 定)，推荐使用 NULL。
- shmflg：共享内存段的访问权限和映射条件
  - 0：共享内存具有可读可写权限。
  - `SHM_RDONLY`：只读。
  - `SHM_RND`： （`shmaddr` 非空时才有效）

    没有指定 SHM_RND 则此段连接到 shmaddr 所指定的地址上(shmaddr 必需 页对齐)。

    指定了 SHM_RND 则此段连接到 shmaddr- shmaddr%SHMLBA 所表示的地上。

返回值: 成功, 返回共享内存段映射地址; 失败，返回 -1

【注意】shmat 函数使用的时候第二个和第三个参数一般设为 NULL 和 0，即系统自动指定共享内存地址，并且共享内存可读可写

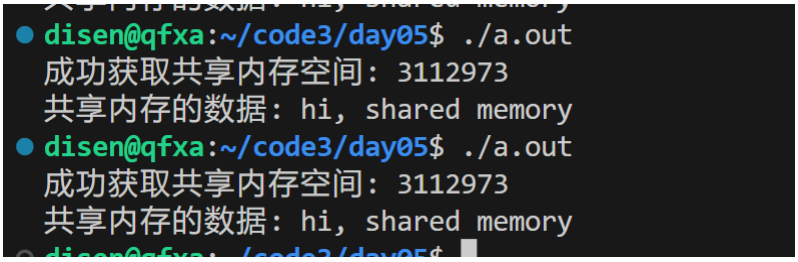
如： 获取共享内存地址，并写数据

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char const *argv[])
{
    // IPC键
    key_t key = ftok("/", 28);

    // 获取共享内存的标识
    int shmid = shmget(key, 32, IPC_CREAT | 0666);
    if (shmid == -1)
    {
        perror("shmget");
        return 1;
    }
    printf("成功获取共享内存空间: %d\n", shmid);

    // 获取共享内存的首地址
    char *buf = shmat(shmid, NULL, 0);
    if (buf == (char *)(-1))
    {
        perror("shsat");
        return 1;
    }
    strcpy(buf, "hi, shared memory");
    printf("write data ok\n");
    return 0;
}
```

如2： 获取共享内存的地址，并读取数据

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char const *argv[])
{
    // IPC键
    key_t key = ftok("/", 28);

    // 获取共享内存的标识
    int shmid = shmget(key, 32, IPC_CREAT | 0666);
    if (shmid == -1)
    {
        perror("shmget");
        return 1;
    }
    printf("成功获取共享内存空间: %d\n", shmid);
```

```
    // 获取共享内存的首地址
    char *buf = shmat(shmid, NULL, 0);
    if (buf == (char *)(-1))
    {
        perror("shsat");
        return 1;
    }
    printf("共享内存的数据: %s\n", buf);
    return 0;
}
```



3.2.3 解除共享内存映射

> 解除共享内存映射(detach)，将共享内存和当前进程分离。
>
> 【注意】仅仅是断开联系并不删除共享内存

```
#include <sys/types.h>
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

参数:

- shmaddr：共享内存映射地址

返回值: 成功返回 0，失败返回 -1。

如:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char const *argv[])
{
    // IPC键
    key_t key = ftok("/", 28);

    // 获取共享内存的标识
    int shmid = shmget(key, 32, IPC_CREAT | 0666);
    if (shmid == -1)
    {
        perror("shmget");
        return 1;
    }
```

```
    printf("成功获取共享内存空间: %d\n", shmid);

    // 获取共享内存的首地址
    char *buf = shmat(shmid, NULL, 0);
    if (buf == (char *)(-1))
    {
        perror("shsat");
        return 1;
    }
    strcpy(buf, "hi,disen");
    printf("共享内存的数据: %s\n", buf);

    // 解除映射
    if (shmdt(buf) == -1)
    {
        perror("shmdt");
        return 1;
    }
    printf("解决映射成功\n");

    return 0;
}
```

3.2.4 共享内存控制

> 共享内存空间的控制

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

参数:

- shmid：共享内存标识符。
- cmd：函数功能的控制
  - IPC_RMID：删除。
  - IPC_SET：设置 shmid_ds 参数。
  - IPC_STAT：保存 shmid_ds 参数。
  - SHM_LOCK：锁定共享内存段(超级用户)。
  - SHM_UNLOCK：解锁共享内存段。
- buf: shmid_ds 数据类型的地址，用来存放或修改共享内存的属性。

返回值: 成功返回 0，失败返回 -1。

如:

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char const *argv[])
{
    // IPC键
    key_t key = ftok("/", 28);

    // 获取共享内存的标识
    int shmid = shmget(key, 32, IPC_CREAT | 0666);
    if (shmid == -1)
    {
        perror("shmget");
        return 1;
    }
    printf("成功获取共享内存空间: %d\n", shmid);

    // 删除共享内存空间
    struct shmid_ds ds;
    int ret = shmctl(shmid, IPC_RMID, &ds);
    if (ret == 0)
    {
        printf("成功删除共享内存 %d\n", shmid);
    }
    return 0;
}
```

```
disen@qfxa:~/code3/day05$ ./a.out
成功获取共享内存空间: 3145741
成功删除共享内存 3145741
disen@qfxa:~/code3/day05$ ipcs -m
```

3.3 综合练习

> 生产者消费者模式

编写两个进程，一个是生产者进程，另一个是消费者进程。它们通过共享内存区域进行通信，生产者将数据写入共享内存，消费者从共享内存中读取数据。

数据结构：

```c
typedef struct {
    int data;
    int flag;
} Data;
```

如：

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
```

```c
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

typedef struct
{
    int data;
    int flag;
} Data;

int main(int argc, char const *argv[])
{

    key_t key = ftok("/", 28); // IPC键
    int shmid = shmget(key, sizeof(Data), IPC_CREAT | 0666);

    if (shmid == -1)
    {
        perror("shmget");
        return 1;
    }

    if (fork() == 0)
    {
        // 生产进程
        Data *data = shmat(shmid, NULL, 0);
        if (data == (Data *)(-1))
        {
            perror("shmat");
            _exit(1);
        }
        data->flag = 0;
        data->data = 0;
        for (int i = 0; i < 100; i++)
        {
            while (data->flag == 1)
                ;
            data->data++;
            data->flag = 1;
            printf("生产进程 生产了data(%d)\n", data->data);
        }
        if (shmdt(data) == -1)
        {
            perror("shmdt");
        }
        _exit(0);
    }

    if (fork() == 0)
    {
        // 消费者进程
        Data *data = shmat(shmid, NULL, 0);
        if (data == (Data *)(-1))
        {
            perror("shmat");
            _exit(1);
        }
```

```
        while (1)
        {
            while (data->flag == 0)
                ;

            printf("消费进程 消费了 data(%d)\n", data->data);
            if (data->data == 100)
                break;
            sleep(1);
            data->flag = 0;
        }
        if (shmdt(data) == -1)
        {
            perror("shmdt");
        }
        _exit(0);
    }
    while (1) // 主进程
    {
        int pid = waitpid(0, NULL, WUNTRACED);
        if (pid == -1)
        {
            break;
        }
        printf("%d 子进程结束\n", pid);
    }
    // 删除共享内存
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
```

```
disen@qfxa:~/code3/day05$ gcc shm6.c
disen@qfxa:~/code3/day05$ ./a.out
生产进程 生产了data(1)
消费进程 消费了 data(1)
消费进程 消费了 data(2)
生产进程 生产了data(2)
消费进程 消费了 data(3)
生产进程 生产了data(3)
```