1.System V IPC进程间通信
1.1IPC进程间通信基础知识
1.1.1IPC进程间通信的种类
1.1.2IPC进程间通信相关命令
1.1.3IPC进程间通信键值获取及组成
1.2消息队列
1.2.1消息队列通信原理
1.2.2消息队列的API（msgget|msgsnd|msgrcv|msgctl）
1.2.3消息队列通信实例
01snd.c
02rcv.c
1.2.4msgctl函数详解
1.3共享内存
1.3.1共享内存的工作原理
1.3.2共享内存的API（shmget|shmat|shmdt|shmctl）
1.3.3共享内存的实例
01write.c
02read.c
1.3.4shmctl函数详解
1.4信号灯集
1.4.1信号灯集工作原理
1.4.2信号灯集的API（semget|semctl|semop）
1.4.3信号灯集的实例
01write.c
02read.c
2.总结

1.System V IPC进程间通信

1.1IPC进程间通信基础知识

1.1.1IPC进程间通信的种类

1. 消息队列
2. 共享内存
3. 信号灯集（信号量）

1.1.2IPC进程间通信相关命令

（1）查看IPC进程间通信ipcs

```
1    <img src="https://hqyj-note-picture.oss-cn-beijing.aliyuncs.com/picture_bak/DC21121-driver/202305171023967.png" alt="image-20230517102357888" style="zoom:80%;" />
```

ipcs -q *//查看消息队列*

ipcs -m *//查看共享内存*

ipcs -s *//查看信号灯集*

（2）删除IPC进程间通信ipcrm

ipcrm -q msqid *//删除消息队列*

ipcrm -m shmid *//删除共享内存*

ipcrm -s semid *//删除信号灯集*

1.1.3IPC进程间通信键值获取及组成

在使用IPC进程间通信的时候，首先需要获取一个key，

只有当两个进程拿到相同的键的之后才能找到同一个IPC。

```
1  #include <sys/types.h>
2  #include <sys/ipc.h>
3
4  key_t ftok(const char *pathname, int proj_id);
5  功能：获取一个键值（键不是唯一的）
6  参数：
7      @pathname:路径及名字
8      @proj_id:只有低8bit有效
9  返回值：成功返回键值，失败返回-1置位错误码
```

```
1  #include <head.h>
2
3  int main(int argc, const char* argv[])
4  {
5      key_t key;
6      struct stat st;
7
```

```c
    if ((key = ftok("/home/linux", 't')) == -1)
        PRINT_ERR("ftok error");

    printf("key = %#x\n", key);

    if (stat("/home/linux", &st))
        PRINT_ERR("stat error");

    printf("inode = %#lx,devno = %#lx,proj_id = %#x\n",st.st_ino,st.st_dev,'t');
    return 0;
}
```

```
linux@ubuntu:~/work/day8$ ./a.out
key = 0x74012316
inode = 0x1c2316,devno = 0x801,proj_id = 0x74
linux@ubuntu:~/work/day8$
```

key(32bit) = proj_id(8) + devno(8) +inode(16)

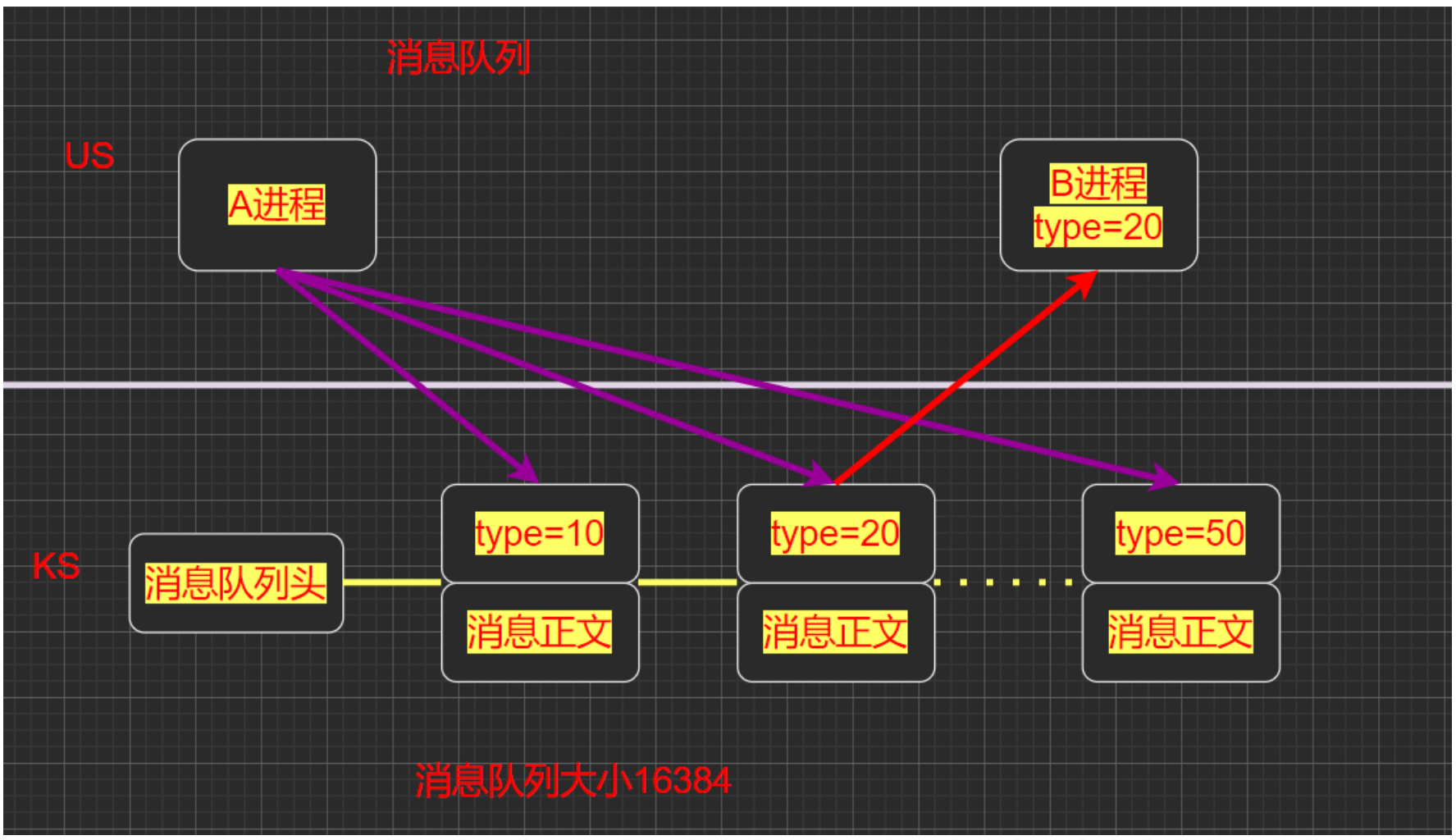
1.2消息队列

1.2.1消息队列通信原理

如果想要使用消息队列实现进程间通信，就必须在内核空间创建出来消息队列，消息队列默认大小是16384（16K），当创建好消息队列之后A进程可以向消息队列中发消息，消息的格式是类型+正文。当消息队列满的时候A进程如果还想往消息队列中发消息A进程休眠。B进程可以通过消息的类型从消息队列中取消息，取出的消息从队列中移除。如果B进程想要获取的消息类型在队列中不存在B进程休眠等。

A进程向消息队列中发消息的时候可以采用：阻塞，非阻塞方式

B进程从消息队列中收消息的时候可以采用：阻塞，非阻塞方式



1.2.2消息队列的API（msgget|msgsnd|msgrcv|msgctl）

```c
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
功能：创建消息队列
参数：
    @key：键值
```

```
        key:通过ftok获取
        IPC_PRIVATE：只能用于亲缘间进程的通信
    @msgflag：消息队列的标志位
        IPC_CREAT|0666  或   IPC_CREAT|IPC_EXCL|0666
返回值：成功返回消息队列号，失败返回-1置位错误码

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
功能：向消息队列中发消息
参数：
    @msqid:消息队列号
    @msgp:消息的首地址
          struct msgbuf {
               long mtype;     //消息的类型，必须大于0
               char mtext[255];  //消息的正文
           };
    @msgsz:消息正文的大小
    @msgflg:消息的标志
        0：阻塞发送
        IPC_NOWAIT：非阻塞发送
返回值：成功返回0，失败返回-1置位错误码

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,int msgflg);
功能：从消息队列获取消息
参数：
    @msqid：消息队列号
    @msgp：消息的首地址
    @msgsz：消息正文的大小
    @msgtyp：消息的类型
             如果=0，接收消息队列中的第一个消息
             如果>0 ,接收msgtyp指定的消息类型
             如果<0，那么将读取队列中第一个最小类型小于或等于msgtyp绝对值的消息。
             2-3-100-500-30-2000
                 -100===>100
             2-3-100-30
    @msgflg：消息的标志
        0：阻塞接收
        IPC_NOWAIT：非阻塞接收
返回值：失败返回-1置位错误码。成功返回接收接收的字节的个数
```

1.2.3消息队列通信实例

01snd.c

```c
#include <head.h>
typedef struct {
    long mtype;
    char name[19];
    char sex;
    int age;
} msg_t;

#define MSGSIZE  (sizeof(msg_t)-sizeof(long))

int main(int argc, const char* argv[])
{
    key_t key;
    int msgqid;
    // 1.获取键值
    if ((key = ftok("/home/linux/", 'p')) == -1)
        PRINT_ERR("ftok error");

    // 2.创建消息队列
    if ((msgqid = msgget(key, IPC_CREAT | 0666)) == -1)
        PRINT_ERR("msgget error");

    // 3.向消息队列中发消息
    int ret;
    msg_t msg;
    while (1) {
    retry:
        printf("input (type name sex age) > ");
        ret = scanf("%ld %s %c %d", &msg.mtype, msg.name, &msg.sex, &msg.age);
        if (ret != 4) {
            printf("input error,try again\n");
            while (getchar() != '\n');
            goto retry;
        }

        if(msgsnd(msgqid, &msg,MSGSIZE, 0))
            PRINT_ERR("msgsnd error");

```

```
        if(msg.mtype == 1000) break;
    }

    return 0;
}
```

02rcv.c

```
#include <head.h>
typedef struct {
    long mtype;
    char name[19];
    char sex;
    int age;
} msg_t;

#define MSGSIZE  (sizeof(msg_t)-sizeof(long))

int main(int argc, const char* argv[])
{
    key_t key;
    int msgqid;
    // 1.获取键值
    if ((key = ftok("/home/linux/", 'p')) == -1)
        PRINT_ERR("ftok error");

    // 2.创建消息队列
    if((msgqid = msgget(key,IPC_CREAT|0666))==-1)
        PRINT_ERR("msgget error");

    // 3.接收消息
    long type;
    msg_t msg;
    while(1){
        printf("input (type) > ");
        scanf("%ld",&type);

        if(msgrcv(msgqid,&msg,MSGSIZE,type,0)==-1)
            PRINT_ERR("msgrcv error");

        printf("type=%ld,name=%s,sex=%c,age=%d\n",msg.mtype,msg.name,msg.sex,msg.age);
        if(msg.mtype == 1000) break;
    }
    if(msgctl(msgqid,IPC_RMID,NULL)) //删除消息队列
        PRINT_ERR("msgctl error");
    return 0;
}
```

1.2.4msgctl函数详解

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
功能：消息队列的控制
参数：
    @msqid：消息队列号
    @cmd：命令码
        IPC_STAT：获取消息队列的属性(ipcs -q msqid)
        IPC_SET：设置消息队列的属性(如设置消息队列大小)
        IPC_RMID:立即删除消息队列，唤醒所有等待的读取器和写入（ipcrm -q msqid）
            器进程(返回一个错误并将errno设置为EIDRM)。调用进程
            必须具有适当的特权，或者它的有效用户ID必须是消息队
            列的创建者或所有者的ID。在这种情况下，msgctl()的/*第
            三个参数将被忽略*/。
    @buf：msqid_ds消息队列属性结构体
返回值：成功返回0，失败返回-1，置位错误码

eg1:使用msgctl删除消息队列
    msgctl(msqid,IPC_RMID,NULL);
eg2:获取消息队列属性
    struct msqid_ds msqds;
    msgctl(msqid, IPC_STAT, &msqds); //获取到的属性在msqds结构体中存放

    //以下是对msqid_ds结构体的详解
    struct msqid_ds
    {
        struct ipc_perm msg_perm;   //权限结构体
        __time_t msg_stime;     //最后一次发送消息的时间
        __time_t msg_rtime;     //最后一次接收消息的时间
        __syscall_ulong_t __msg_cbytes;//当前消息队列中字节数
        msgqnum_t msg_qnum;     //当前消息队列中消息的个数
        msglen_t msg_qbytes;    //消息队列中能够容纳的字节数（16384）
        __pid_t msg_lspid;      //最后一次发送消息的进程号
        __pid_t msg_lrpid;      //最后一次接收消息的进程号
    };
```

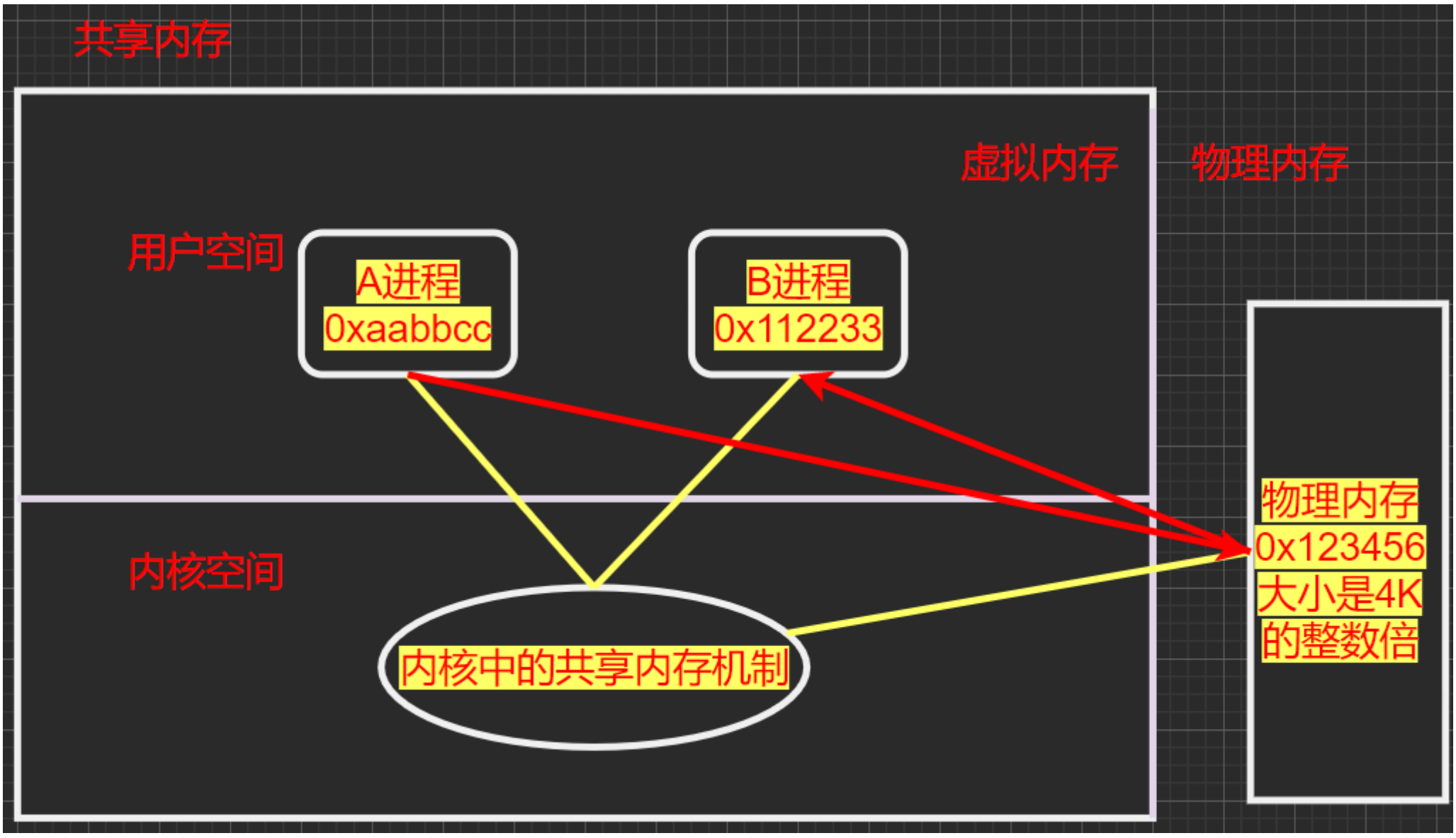
```
    struct ipc_perm
    {
        __key_t __key;          //键值
        __uid_t uid;            //消息队列所属的uid
        __gid_t gid;            //消息队列所属的gid
        __uid_t cuid;           //创建消息队列的uid
        __gid_t cgid;           //创建消息队列的gid
        unsigned short int mode;    //消息队列的读写权限
    };
```

```
#include <head.h>
typedef struct {
    long mtype;
    char name[19];
    char sex;
    int age;
} msg_t;

#define MSGSIZE (sizeof(msg_t) - sizeof(long))

int main(int argc, const char* argv[])
{
    key_t key;
    int msgqid;
    // 1.获取键值
    if ((key = ftok("/home/linux/", 'p')) == -1)
        PRINT_ERR("ftok error");

    // 2.创建消息队列
    if ((msgqid = msgget(key, IPC_CREAT | 0666)) == -1)
        PRINT_ERR("msgget error");

    // 3.向消息队列中发消息
    int ret;
    msg_t msg;
    while (1) {
    retry:
        printf("input (type name sex age) > ");
        ret = scanf("%ld %s %c %d", &msg.mtype, msg.name, &msg.sex, &msg.age);
        if (ret != 4) {
            printf("input error,try again\n");
            while (getchar() != '\n')
                ;
            goto retry;
        }

        if (msgsnd(msgqid, &msg, MSGSIZE, 0))
            PRINT_ERR("msgsnd error");

        if (msg.mtype == 1000)
            break;
    }

    // 4.获取消息队列属性
    struct msqid_ds msqds;
    if (msgctl(msgqid, IPC_STAT, &msqds))
        PRINT_ERR("msgctl error");

    printf("key=%#x,uid=%d,gid=%d,mode=%#o,nq=%ld,nb=%ld,mb=%ld\n",
        msqds.msg_perm.__key, msqds.msg_perm.uid, msqds.msg_perm.gid,
        msqds.msg_perm.mode, msqds.msg_qnum, msqds.__msg_cbytes, msqds.msg_qbytes);

    // 5.设置消息队列属性
    msqds.msg_qbytes = 32768;
    if (msgctl(msgqid, IPC_SET, &msqds))
        PRINT_ERR("msgctl error");

    // 6.再次获取消息队列属性
    memset(&msqds, 0, sizeof(msqds));
    if (msgctl(msgqid, IPC_STAT, &msqds))
        PRINT_ERR("msgctl error");

    printf("key=%#x,uid=%d,gid=%d,mode=%#o,nq=%ld,nb=%ld,mb=%ld\n",
        msqds.msg_perm.__key, msqds.msg_perm.uid, msqds.msg_perm.gid,
        msqds.msg_perm.mode, msqds.msg_qnum, msqds.__msg_cbytes, msqds.msg_qbytes);

    // 7.删除消息队列
    if (msgctl(msgqid, IPC_RMID, NULL))
        PRINT_ERR("msgctl error");
    return 0;
}
```

1.3共享内存

1.3.1共享内存的工作原理

共享内存是所有进程间通信方式中效率最高的一个，因为当创建共享内存之后，需要通信的A和B进程可以直接操作这块物理内存空间，省去了向内核拷贝数据的过程。共享内存的大小是4K的整数倍。



1.3.2共享内存的API（shmget|shmat|shmdt|shmctl)

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
功能：创建共享内存
参数：
    @key:键值
        key:通过ftok获取
        IPC_PRIVATE：只能用于亲缘间进程的通信
    @size:共享内存的大小 4k整数倍
    @msgflag:共享的标志位
        IPC_CREAT|0666  或   IPC_CREAT|IPC_EXCL|0666
返回值：成功返回共享内存编号，失败返回-1置位错误码

void *shmat(int shmid, const void *shmaddr, int shmflg);
功能：映射共享内存到当前的进程空间
参数：
    @shmid:共享内存的编号
    @shmaddr:NULL，让系统自动分配
    @shmflg:共享内存的操作方式
         0：读写
         SHM_RDONLY：只读
返回值：成功返回共享内存的首地址，失败返回（void *）-1,并置位错误码

int shmdt(const void *shmaddr);
功能：取消地址映射
参数：
    @shmaddr:指向共享内存的指针
返回值：成功返回0，失败返回-1置位错误码

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
功能：共享内存控制的函数
参数：
    @shmid:共享内存的编号
    @cmd:操作的命令码
         IPC_STAT ：获取
         IPC_SET：设置
         IPC_RMID：删除共享内存
            标记要销毁的段。实际上，只有在最后一个进程将其分离之后
            (也就是说，关联结构shmid_ds的shm_nattch成员为零时)，
            段才会被销毁。调用者必须是段的所有者或创建者，或具有特权。buf参数被忽略。
    @buf:共享内存属性结构体指针
返回值:成功返回0，失败返回-1置位错误码
```

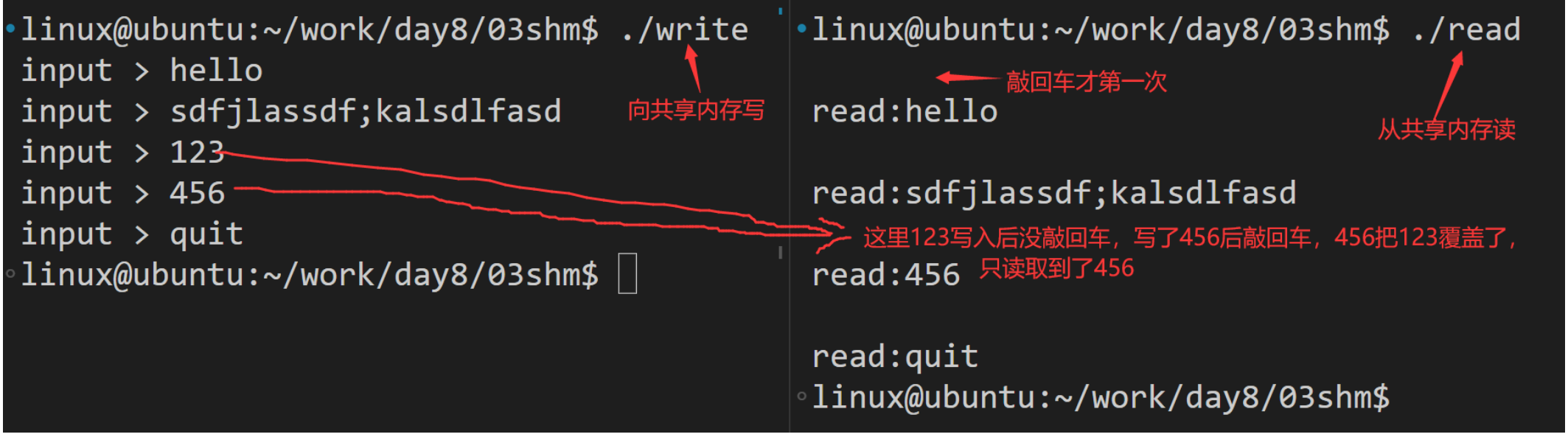
1.3.3共享内存的实例

01write.c

```c
#include <head.h>
#define SHMSIZE (4096)
int main(int argc, const char* argv[])
{
    key_t key;
    int shmid;
    char* waddr;
    // 1.获取键值 ftok
    if ((key = ftok("/home", 'g')) == -1)
        PRINT_ERR("ftok error");
    // 2.创建共享内存 shmget
    if ((shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666)) == -1)
        PRINT_ERR("shmget error");
    // 3.将共享内存映射到用户空间 shmat
    if ((waddr = shmat(shmid, NULL, 0)) == (void*)-1)
        PRINT_ERR("shmat error");
    // 4.共享内存操作（写）
    while (1) {
        printf("input > ");
        fgets(waddr, SHMSIZE, stdin);
        if (waddr[strlen(waddr) - 1] == '\n')
            waddr[strlen(waddr) - 1] = '\0';
        if (strcmp(waddr, "quit") == 0)
            break;
    }
    // 5.取消映射 shmdt
    if(shmdt(waddr))
        PRINT_ERR("shmdt error");
    return 0;
}
```

02read.c

```c
#include <head.h>
#define SHMSIZE (4096)
int main(int argc, const char* argv[])
{
    key_t key;
    int shmid;
    char* raddr;
    // 1.获取键值 ftok
    if ((key = ftok("/home", 'g')) == -1)
        PRINT_ERR("ftok error");
    // 2.创建共享内存 shmget
    if ((shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666)) == -1)
        PRINT_ERR("shmget error");
    // 3.将共享内存映射到用户空间 shmat
    if ((raddr = shmat(shmid, NULL, 0)) == (void*)-1)
        PRINT_ERR("shmat error");
    // 4.从共享内存中数数据
    while (1) {
        getchar(); // 敲回车读一次数据，如果不写这句话会疯狂刷屏
        printf("read:%s\n", raddr);
        if (strncmp(raddr, "quit", 4) == 0)
            break;
    }
    // 5.取消映射 shmdt
    if (shmdt(raddr))
        PRINT_ERR("shmdt error");
    // 6.删除共享内存 shmctl
    if (shmctl(shmid, IPC_RMID, NULL))
        PRINT_ERR("shmctl error");
    return 0;
}
```



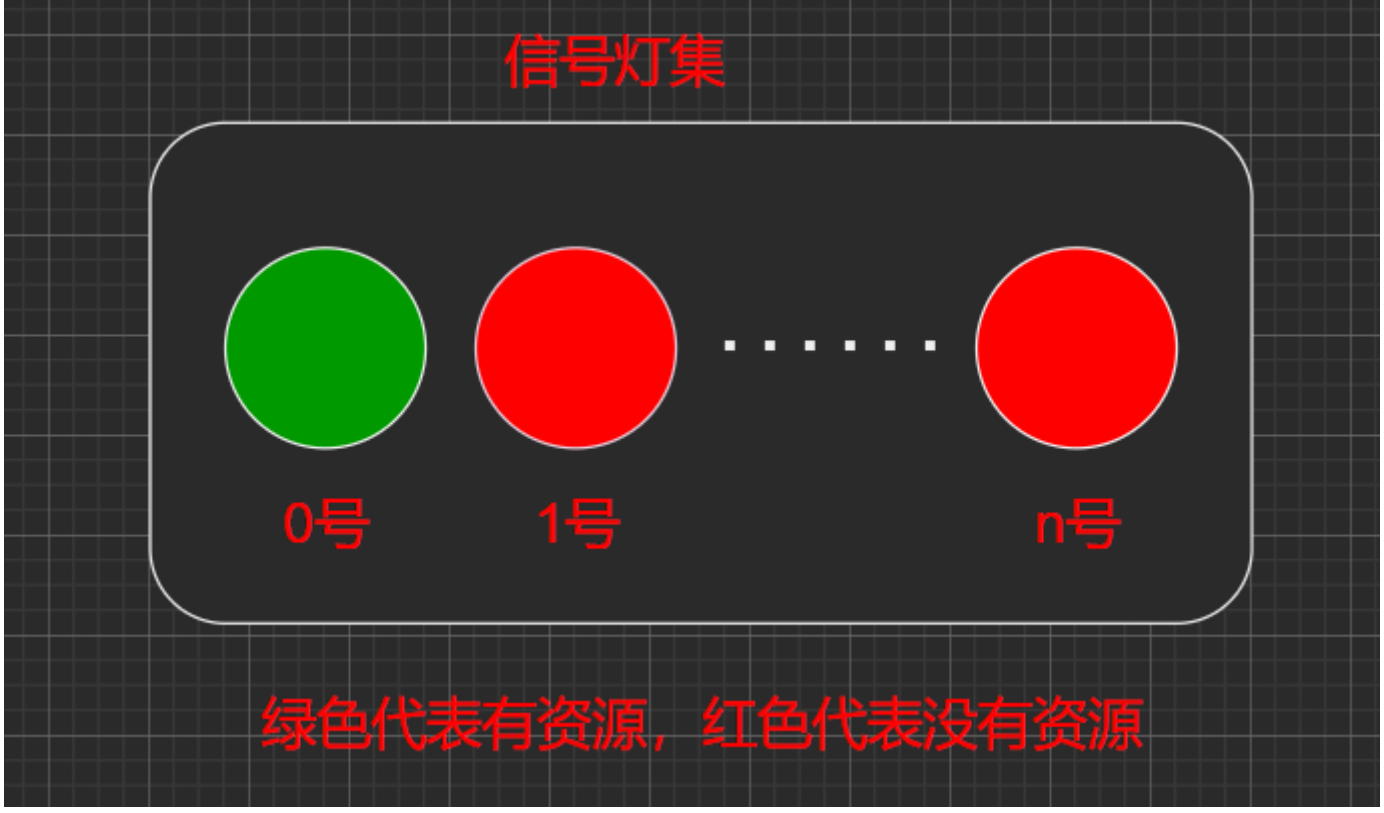
1.3.4shmctl函数详解

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
功能：共享内存控制的函数
参数：
    @shmid:共享内存的编号
    @cmd:操作的命令码
         IPC_STAT ： 获取  （ipcs -m）
         IPC_SET：设置
         IPC_RMID：删除共享内存 （ipcrm -m shmid）
            标记要销毁的段。实际上，只有在最后一个进程将其分离之后
            （也就是说，关联结构shmid_ds的shm_nattch成员为零时），
            段才会被销毁。调用者必须是段的所有者或创建者，或具有特权。buf参数被忽略。
    @buf:共享内存属性结构体指针
返回值:成功返回0，失败返回-1置位错误码

 eg1:
    shmctl(shmid,IPC_RMID,NULL); //删除共享内存
eg2：
    struct shmid_ds buf;
    shmctl(shmid,IPC_STAT,&buf); //获取消息队列的属性
    //以下是shmid_ds结果的详解
    struct shmid_ds
    {
        struct ipc_perm shm_perm; //权限结构体
        size_t shm_segsz;         //共享内存的大小，单位是字节
        __time_t shm_atime;       //最后一次调用shmat的时间
        __time_t shm_dtime;       //最后一次调用shmdt的时间
        __time_t shm_ctime;       //最后一次调用shmctl改变属性的时间
        __pid_t shm_cpid;         //创建共享内存的PID
        __pid_t shm_lpid;         //最后一次操作共享内存的PID
        shmatt_t shm_nattch;      //当前多少个进程关联共享内存
    };
    struct ipc_perm
    {
        __key_t __key;          //键值
        __uid_t uid;            //消息队列所属的uid
        __gid_t gid;            //消息队列所属的gid
        __uid_t cuid;           //创建消息队列的uid
        __gid_t cgid;           //创建消息队列的gid
        unsigned short int mode;    //消息队列的读写权限
    };
```

1.4信号灯集

1.4.1信号灯集工作原理

信号量（信号灯集）：是实现进程同步的机制，在一个信号灯集中可以有很多个信号灯。在信号灯集内信号灯相互独立，每个灯的值的改变不会影响其他的信号灯，信号灯的值一般设置为二值量（1或者0，1代表有资源，0代表没有资源）。



1.4.2信号灯集的API（semget|semctl|semop）

```
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
功能：创建一个信号灯集
参数：
    @key:键值
         IPC_PRIVATE
         key
    @nsems:信号灯集合中信号灯的个数
    @semflag:创建的标志位
        IPC_CREAT|0666  或  IPC_CREAT|IPC_EXCL|0666
返回值：成功返回semid,失败返回-1置位错误码


int semctl(int semid, int semnum, int cmd, ...);
功能：信号灯集的控制函数
参数：
    @semid信号灯集的ID
    @senum:信号灯的编号
    @cmd:命令码
        SETVAL：设置信号灯的值 --->第四个参数val选项
        GETVAL：获取信号灯的值 --->不需要第四个参数
        IPC_STAT：获取信号灯集的属性--->第二个参数被忽略，第四个参数buf选项
        IPC_SET ：设置信号灯集的属性--->第四个参数buf选项
        IPC_RMID:第二参数被忽略，第4个参数不用填写
    @...:
          union semun {
               int              val;    /* Value for SETVAL */
               struct semid_ds *buf;    /* Buffer for IPC_STAT, IPC_SET */
           };
返回值：失败返回-1置位错误码
        成功：
            GETVAL:成功返回信号灯的值
            其余的命令码成功返回0
eg:设置灯的值
    union semun sem = {
        .val = 1,
    }
    semctl(semid,0,SETVAL,sem); //将0号灯初始值设置为1
    union semun sem = {
        .val = 0,
    }
    semctl(semid,1,SETVAL,sem); //将1号灯初始值设置为0
eg:获取信号灯的值
    val = semctl(semid,0,GETVAL); //获取0号灯的值

eg:获取信号灯集的属性
     struct semid_ds buf;
      union semun sem = {
        .buf = &buf,
      }
    semctl(semid,0, IPC_STAT,sem); //获取信号灯集的属性，第二参数被忽略
eg:删除信号等集
    semctl(semid,0,IPC_RMID); //删除信号等集

int semop(int semid, struct sembuf *sops, size_t nsops);
功能：信号灯集中信号灯的操作函数
参数：
    @semid:信号灯集的编号
```

```
    @sops:操作方式
        struct sembuf{
           unsigned short sem_num;  //信号灯的编号
           short          sem_op;   //操作方式（PV）
                                    -1:P操作，申请资源
                                     1:V操作，释放资源
           short          sem_flg;  //操作的标志位
                                    0：阻塞
                                    IPC_NOWAIT：非阻塞方式操作
        }
    @nsops:本次操作信号灯的个数
返回值：成功返回0，失败返回-1置位错误码
```

1.4.3信号灯集的实例

01write.c

```c
#include <head.h>
#define SHMSIZE (4096)
union semun {
    int val; /* Value for SETVAL */
    struct semid_ds* buf; /* Buffer for IPC_STAT, IPC_SET */
};
int mysem_init(int semid, int which, int value)
{
    union semun sem = {
        .val = value,
    };
    if (semctl(semid, which, SETVAL, sem) == -1)
        PRINT_ERR("semctl error");

    return 0;
}
int P(int semid, int which)
{
    struct sembuf op = {
        .sem_num = which, // 那个灯
        .sem_op = -1, // -1 P申请   1 V释放
        .sem_flg = 0, // 阻塞
    };
    if (semop(semid, &op, 1))
        PRINT_ERR("semop P error");
    return 0;
}
int V(int semid, int which)
{
    struct sembuf op = {
        .sem_num = which, // 那个灯
        .sem_op = 1, // -1 P申请   1 V释放
        .sem_flg = 0, // 阻塞
    };
    if (semop(semid, &op, 1))
        PRINT_ERR("semop V error");
    return 0;
}
int main(int argc, const char* argv[])
{
    key_t key;
    int shmid, semid;
    char* waddr;
    // 1.获取键值 ftok
    if ((key = ftok("/home", 'g')) == -1)
        PRINT_ERR("ftok error");
    // 2.创建共享内存 shmget
    if ((shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666)) == -1)
        PRINT_ERR("shmget error");
    // 3.将共享内存映射到用户空间 shmat
    if ((waddr = shmat(shmid, NULL, 0)) == (void*)-1)
        PRINT_ERR("shmat error");

    // 4.创建信号灯集，并初始化信号灯
    if ((semid = semget(key, 2, IPC_CREAT | IPC_EXCL | 0666)) == -1) {
        if (errno = EEXIST) {
            semid = semget(key, 2, IPC_CREAT|0666);
        } else {
            PRINT_ERR("semget error");
        }
    } else {
        mysem_init(semid, 0, 1); // 将0号灯设置为1
        mysem_init(semid, 1, 0); // 将1号灯设置为0
    }

```

```
    // 5.共享内存操作（写）
    while (1) {
        P(semid, 0); // 申请0号灯资源
        printf("input > ");
        fgets(waddr, SHMSIZE, stdin);
        if (waddr[strlen(waddr) - 1] == '\n')
            waddr[strlen(waddr) - 1] = '\0';
        V(semid, 1); // 释放1号灯资源
        if (strcmp(waddr, "quit") == 0)
            break;
    }
    // 6.取消映射 shmdt
    if (shmdt(waddr))
        PRINT_ERR("shmdt error");
    return 0;
}
```

02read.c

```
#include <head.h>
#define SHMSIZE (4096)
union semun {
    int val; /* Value for SETVAL */
    struct semid_ds* buf; /* Buffer for IPC_STAT, IPC_SET */
};
int mysem_init(int semid, int which, int value)
{
    union semun sem = {
        .val = value,
    };
    if (semctl(semid, which, SETVAL, sem) == -1)
        PRINT_ERR("semctl error");

    return 0;
}
int P(int semid, int which)
{
    struct sembuf op = {
        .sem_num = which, // 那个灯
        .sem_op = -1, // -1 P申请   1 V释放
        .sem_flg = 0, // 阻塞
    };
    if (semop(semid, &op, 1))
        PRINT_ERR("semop P error");
    return 0;
}
int V(int semid, int which)
{
    struct sembuf op = {
        .sem_num = which, // 那个灯
        .sem_op = 1, // -1 P申请   1 V释放
        .sem_flg = 0, // 阻塞
    };
    if (semop(semid, &op, 1))
        PRINT_ERR("semop V error");
    return 0;
}
int main(int argc, const char* argv[])
{
    key_t key;
    int shmid, semid;
    char* raddr;
    // 1.获取键值 ftok
    if ((key = ftok("/home", 'g')) == -1)
        PRINT_ERR("ftok error");
    // 2.创建共享内存 shmget
    if ((shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666)) == -1)
        PRINT_ERR("shmget error");
    // 3.将共享内存映射到用户空间 shmat
    if ((raddr = shmat(shmid, NULL, 0)) == (void*)-1)
        PRINT_ERR("shmat error");

    // 4.创建信号灯集，并初始化信号灯
    if ((semid = semget(key, 2, IPC_CREAT | IPC_EXCL | 0666)) == -1) {
        if (errno = EEXIST) {
            semid = semget(key, 2, IPC_CREAT | 0666);
        } else {
            PRINT_ERR("semget error");
        }
    } else {
        mysem_init(semid, 0, 1); // 将0号灯设置为1
        mysem_init(semid, 1, 0); // 将1号灯设置为0
    }

```

```
    // 5.从共享内存中数数据
    while (1) {
        P(semid, 1); // 申请1号灯资源
        printf("read:%s\n", raddr);
        V(semid, 0); // 释放0号灯资源
        if (strncmp(raddr, "quit", 4) == 0)
            break;
    }
    // 6.取消映射 shmdt
    if (shmdt(raddr))
        PRINT_ERR("shmdt error");
    // 7.删除共享内存 shmctl
    if (shmctl(shmid, IPC_RMID, NULL))
        PRINT_ERR("shmctl error");
    // 8.删除信号灯集
    if (semctl(semid, 0, IPC_RMID))
        PRINT_ERR("semctl error");
    return 0;
}
```

2.总结

一、数据结构

1.顺序表

2.单链表

3.单向循环链表

4.双向链表

5.双向循环链表

6.栈（顺序栈，链式栈）

7.队列（循环队列，链式队列）

8.二叉树（创建，遍历）

9.快排

10.哈希

二、IO进程

fopen/fclose/fgetc/fputc/fgets/fputs/fread/fwrite/fseek/ftell/rewind

perror/strerror/snprintf/sprintf/fprintf/time/localtime

open/read/write/close/lseek/stat/lstat/getpwuid/getgrgid

opendir/readdir/closedir

fork/getpid/getppid/exit/_exit/wait/waitpid/dup/dup2/system

pthread_create/pthread_self/pthread_exit/pthread_join/pthread_detach/pthread_cancel

pthread_mutex_init/pthread_mutex_lock/pthread_mutex_unlock/pthread_mutex_destroy

sem_init/sem_wait/sem_post/sem_destroy

pthread_cond_init/pthread_cond_wait/pthread_cond_signal/

pthread_cond_broatcast/pthread_cond_destroy

pipe/mkfifo/signal/kill/raise/alarm

ftok/msgget/msgsnd/msgrcv/msgctl

shmget/shmat/shmdt/shmctl

semget/semctl/semop