LECCIÓN 21: MENSAJES

LECCIÓN 21: MENSAJES

21.1 Colas de mensajes

La manipulación de los IPC se efectúa mediante las llamadas de sistema. Las tres llamadas fundamentales de las colas de mensajes son:

- msgget → Creación
- msgctl → Control
- msgsnd, msgrcv → Comunicación

También se utilizan frecuentemente siete constantes:

- IPC_PRIVATE
- IPC_CREAT
- IPC_EXCL
- IPC_KERNELD
- IPC_NOWAIT
- IPC_RMID
- IPC_SET
- IPC_GET
- IPC_INFO

Las colas de mensajes se comparan generalmente con un sistema de buzones. El principio es relativamente simple: un proceso deposita uno o más mensajes en un "buzón". Otro proceso (o varios) puede leer cada uno de los mensajes, en el orden de su llegada, según el tipo de mensajes que desea. Es exactamente como cuando se recoge el correo: se puede preferir empezar por leer por la parte baja de la pila, por arriba, por las facturas o por las postales.

21.2 Las estructuras básicas

Para manipular una cola de mensajes, además de las llamadas al sistema, se necesitan tres estructuras.

*La estructura **msqid_ds** corresponde a un objeto de la cola de mensajes. Mediante esta estructura es posible manipular el objeto creado. Está definida en el archivo de cabecera <linux/msg.h>. He aquí una descripción de la estructura:

Tipo	Campo	Descripción
Struct ipc_perm	msg_perm	Derechos de acceso del objeto
Struct msg *	msg_first	Puntero al primer mensaje de la cola
Struct msg *	msg_last	Puntero al último mensaje de la cola
Time_t	msg_stime	Fecha a la última llamada a msgsnd
Time_t	msg_rtime	Fecha a la última llamada a msgrcv
Time_t	msg_ctime	Fecha a la última modificación del objeto

Struct wait_queue *	Wwait	Cola de procesos en espera de escritura
Struct wait_queue *	Rwait	Cola de procesos en espera de lectura
Unshort	msg_cbytes	Número de bytes actualmente en la cola
Unshort	msg_qnum	Número de mensajes en la cola
Unshort	msg_qbytes	Número máximo de bytes en la cola
Unshort	msg_lspid	Número del último proceso que ha efectuado un msgsnd
Unshort	msg_lrpid	Número del último proceso que ha efectuado un msgrcv

*Este archivo de cabecera contiene también la definición de la estructura **msginfo**, que se utiliza en una llamada a msgctl con IPC_INFO como argumento. Esta estructura se utiliza en particular en programas ipcs, y está reservada especialmente a programas del sistema de estadística o de observación de la máquina. La estructura está constituida por los campos siguientes:

Tipo	**Campo**	**Descripción**
Int	msgpool	Tamaño en kilobytes de los datos en la cola
Int	msgmap	Número de entradas en la tabla de mensajes
Int	msgmax	Tamaño máximo de los mensajes (en bytes)
Int	msgmnb	Tamaño máximo de la cola de mensajes
Int	msgmni	Número máximo de identificadores de colas de mensajes
Int	msgssz	Tamaño del segmento de mensaje
Int	msgtql	Número de cabeceras de mensajes del sistema
Unshort	msgseg	Número máximo de segmentos

*La estructura **msgbuf** almacena un mensaje y su tipo. Corresponde al modelo a utilizar para el envío y la recuperación de un mensaje en una cola. La estructura está constituida por los campos siguientes:

Tipo	**Campo**	**Descripción**
Long	Mtype	Tipo del mensaje
Char[1]	Mtext	Contenido del mensaje

Esta estructura no se usa en las aplicaciones. En realidad, toda estructura de datos depositada en una cola de mensajes debe tener necesariamente como primer campo el tipo del mensaje. Éste es un número estrictamente positivo, de tipo long, que permitirá la selección de un mensaje en la cola en función de su tipo.

21.3 Creación y búsqueda de colas de mensajes

* La llamada al sistema **msgget** cumple dos funciones:

1. Creación de una nueva cola de mensajes
2. Búsqueda de una cola de mensajes existentes (creada por la otra aplicación, por ejemplo) mediante su clave

En ambos casos, sólo se puede usar si se posee una clave. El prototipo de esta llamada al sistema es el siguiente:

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

int msgget (key_t clave, int opcion);
```

El primer argumento corresponde a la clave de la cola de mensajes que ya existe, o de la que se desea crear. Si se pasa como clave el valor IPC_PRIVATE, se crea una cola. Si se pasa una clave diferente, se presenta dos posibilidades:

1. La clave no es utilizada por otra cola de mensajes: en este caso, es necesario indicar IPC_CREAT como opción. En este caso se creará la cola, con la clave pasada como parámetro. Las opciones pueden crear ciertos derechos de acceso.
2. La clave se utiliza en una cola de mensajes. Es necesario que se pase IPC_CREAT o bien IPC_EXCL como parámetro. En este caso, a continuación se puede leer o escribir en la cola de mensajes, si los derechos lo permiten.

Si todo ocurre correctamente, **msgget** devuelve el identificador de la cola de mensajes. En caso contrario, errno posee el siguiente valor:

Error	Significado
EACCES	Existe una cola de mensajes para la clave, pero el proceso que llama no tiene derechos de acceso sobre la cola de mensajes
EEXIST	Existe ya una cola de mensajes para la clave, y se han fijado las opciones IPC_CREAT e IPC_EXCL
EIDRM	La cola de mensajes está marcada como destinada a destruirse
ENOENT	No existe ninguna cola de mensajes para la clave, y la opción IPC_CREAT no se ha indicado
ENOMEM	Se habría podido crear una cola de mensajes, pero el sistema no tiene suficiente memoria para la nueva estructura
ENOSPC	Se ha alcanzado el número máximo de colas de mensajes

21.4 Control de las colas de mensajes

Tras haber creado una cola de mensajes, es posible manipularla modificando por ejemplo los permisos de acceso a la cola. Ante todo, hay que saber que los IPC se gestionan en el núcleo por la tabla de colas de mensajes. La llamada al sistema **msgctl** permite acceder y modificar ciertos campos de esta tabla para las colas a las que se tiene acceso.

* El prototipo de la llamada es:

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

int msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

El primer argumento **msquid** corresponde al identificador de la cola de mensajes dado por la llamada **msgget**. El segundo argumento **cmd** indica el tipo de la operación a efectuar sobre la cola de mensajes. Esta es la lista de las operaciones que se pueden realizar sobre la cola de mensajes:

- MSG_STAT (o IPC_STAT): copia la tabla asociada a la cola de mensajes en la dirección apuntada por **buf** de tipo estructura **msqid_ds**.
- IPC_SET: permite fijar ciertos miembros de la estructura **msqid_ds**. Esta operación actualiza automáticamente el campo **msg_ctime** que conserva la fecha de la última modificación de la entrada en la tabla de procesos. Es posible acceder a tres campos de la estructura: **msg_perm.uid**, **msg_perm.gid** y **msg_perm.mode**. Puede modificarse un campo suplementario, pero únicamente por parte del superusuario: **msg_qbytes**.
- IPC_RMID: permite destruir la cola de mensajes y los datos que contiene. Sólo puede destruir una cola de mensajes un proceso cuyo identificador de usuario efectivo corresponda al superusuario, al creador o al propietario de la cola de mensajes.
- MSG_INFO (o IPC_INFO): llena la estructura **struct msginfo** pasada como parámetro. Esto se usa por ejemplo en el programa ipcs.

21.5 Emisión de mensajes

La llamada al sistema **msgsnd** permite enviar un mensaje a una cola de mensajes. Su prototipo es el siguiente:

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

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

int msgsnd (int msqid, struct msgbuf *msgp, int msgsize, int msgopt);
```

Para que esta operación se desarrolle normalmente, es necesario haber obtenido el identificador de la cola de mensajes, pero también poseer los derechos necesarios para escribir en la cola de mensajes. El segundo parámetro corresponde a los datos que se quiere enviar a la cola de mensajes. El parámetro **msgsize** corresponde al tamaño del objeto que se envía a la cola.

Si se pasa la opción IPC_NOWAIT, sólo tiene efecto cuando la cola está llena. En este caso, la llamada no es bloqueadora (a diferencia de una llamada sin este parámetro) y se devuelve el error EAGAIN.

Esta es la lista de errores que puede devolver esta llamada:

Error	Significado
EAGAIN	No puede enviarse el mensaje porque la cola de espera está llena y la opción IPC_NOWAIT ha sido activada
EACCES	No hay derechos de escritura
EFAULT	La dirección apuntada por msgp no es accesible
EIDRM	La cola ya no existe: ha sido destruida
EINTR	El proceso ha recibido una señal y por tanto la llamada al sistema ha fallado
EINVAL	Identificador erróneo de la cola, o el tipo del dato a enviar a la cola de mensajes no es positivo
ENOMEM	No hay suficiente memoria para realizar una copia del objeto

Esta llamada devuelve 0 en caso de éxito.

21.6 Recepción de mensajes

* La llamada **msgrcv** permite leer de la cola de mensajes, y posee el prototipo siguiente:

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

int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg);
```

Esta llamada al sistema permite, si los derechos del proceso son suficientes, recuperar un mensaje que se copiará en **msgp**. La zona de memoria apuntada por **msgp** tiene como tamaño máximo **msgsz**.

Pueden darse dos tipos de opciones en el campo **msgflg**:

- MSG_NOERROR: si el tamaño del mensaje es superior al tamaño especificado en el campo **msgsz**, y si la opción MSG_NOERROR

está posicionada, el mensaje se truncará. La parte sobrante se pierde. En el caso en que no esté esta opción, el mensaje no se retira de la cola y la llamada fracasa devolviendo como error E2BIG.

- IPC_NOWAIT: esta opción permite evitar la espera activa. Si la cola no está nunca vacía, se devuelve el error ENOMSG. Si esta opción no está activa, la llamada se suspende hasta que un dato del tipo solicitado entre en la cola de mensajes.

El tipo de mensaje a leer debe especificarse en el campo **msgtyp**:

- Si **msgtyp** es igual a 0, se lee el primer mensaje de la cola, es decir, el mensaje más antiguo, sea cual sea su tipo.
- Si **msgtyp** es negativo, entonces se devuelve el primer mensaje de la cola con el tipo menor, inferior o igual al valor absoluto de **msgtyp**.
- Si **msgtyp** es positivo, se devuelve el primer mensaje de la cola con un tipo estrictamente igual a **msgtyp**. En el caso de que esté presente la opción MSG_EXCPT, se devolverá el primer mensaje con un tipo diferente.

Estos son los errores que pueden resultar del uso de esta llamada al sistema:

Error	**Significado**
EINVAL	El identificador de la cola de mensajes no es válido, mtype es inferior a cero, o bien msgsz es inferior a cero o mayor que el tamaño máximo de un mensaje MSGMAX
EFAULT	La zona de memoria apuntada por msgp no es accesible
EIDRM	El proceso esperaba un mensaje y la cola ya no existe: ha sido destruida
EACCES	El proceso no tiene los derechos necesarios para acceder a la cola de mensajes
E2BIG	El tamaño del mensaje es mayor que msgsz y no se ha activado la opción MSG_NOERROR
ENOMSG	Se ha activado la opción IPC_NOWAIT y no se ha encontrado ningún mensaje en la cola
EINTR	El proceso esperaba un mensaje y ha recibido una señal

21.7 Código fuente

```
/*
 * linux/ipc/msg.c
 * Copyright (C) 1992 Krishna Balasubramanian
 *
 * Removed all the remaining kerneld mess
 * Catch the -EFAULT stuff properly
 * Use GFP_KERNEL for messages as in 1.2
 * Fixed up the unchecked user space derefs
 * Copyright (C) 1998 Alan Cox & Andi Kleen
 *
 */

#include <linux/malloc.h>
#include <linux/msg.h>
#include <linux/interrupt.h>
#include <linux/smp_lock.h>
#include <linux/init.h>

#include <asm/uaccess.h>

extern int ipcperms (struct ipc_perm *ipcp, short msgflg);

static void freeque (int id);
static int newque (key_t key, int msgflg);
static int findkey (key_t key);

static struct msqid_ds *msgque[MSGMNI];
static int msgbytes = 0;
static int msghdrs = 0;
static unsigned short msg_seq = 0;
static int used_queues = 0;
static int max_msqid = 0;
static struct wait_queue *msg_lock = NULL;

void __init msg_init (void)
{
        int id;

        for (id = 0; id < MSGMNI; id++)
                msgque[id] = (struct msqid_ds *) IPC_UNUSED;
        msgbytes = msghdrs = msg_seq = max_msqid = used_queues = 0;
        msg_lock = NULL;
        return;
}

static int real_msgsnd (int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg)
{
```

```
        int id;
        struct msqid_ds *msq;
        struct ipc_perm *ipcp;
        struct msg *msgh;
        long mtype;

        if (msgsz > MSGMAX || (long) msgsz < 0 || msqid < 0)
                return -EINVAL;
        if (get_user(mtype, &msgp->mtype))
                return -EFAULT;
        if (mtype < 1)
                return -EINVAL;
        id = (unsigned int) msqid % MSGMNI;
        msq = msgque [id];
        if (msq == IPC_UNUSED || msq == IPC_NOID)
                return -EINVAL;
        ipcp = &msq->msg_perm;

 slept:
        if (msq->msg_perm.seq != (unsigned int) msqid / MSGMNI)
                return -EIDRM;

        if (ipcperms(ipcp, S_IWUGO))
                return -EACCES;

        if (msgsz + msq->msg_cbytes > msq->msg_qbytes) {
                if (msgsz + msq->msg_cbytes > msq->msg_qbytes) {
                        /* still no space in queue */
                        if (msgflg & IPC_NOWAIT)
                                return -EAGAIN;
                        if (signal_pending(current))
                                return -EINTR;
                        interruptible_sleep_on (&msq->wwait);
                        goto slept;
                }
        }

        /* allocate message header and text space*/
        msgh = (struct msg *) kmalloc (sizeof(*msgh) + msgsz, GFP_KERNEL);
        if (!msgh)
                return -ENOMEM;
        msgh->msg_spot = (char *) (msgh + 1);

        if (copy_from_user(msgh->msg_spot, msgp->mtext, msgsz))
        {
                kfree(msgh);
                return -EFAULT;
        }

        if (msgque[id] == IPC_UNUSED || msgque[id] == IPC_NOID
```

```
                || msq->msg_perm.seq != (unsigned int) msqid / MSGMNI) {
                kfree(msgh);
                return -EIDRM;
        }

        msgh->msg_next = NULL;
        msgh->msg_ts = msgsz;
        msgh->msg_type = mtype;
        msgh->msg_stime = CURRENT_TIME;

        if (!msq->msg_first)
                msq->msg_first = msq->msg_last = msgh;
        else {
                msq->msg_last->msg_next = msgh;
                msq->msg_last = msgh;
        }
        msq->msg_cbytes += msgsz;
        msgbytes  += msgsz;
        msghdrs++;
        msq->msg_qnum++;
        msq->msg_lspid = current->pid;
        msq->msg_stime = CURRENT_TIME;
        wake_up (&msq->rwait);
        return 0;
}

static int real_msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int
msgflg)
{
        struct msqid_ds *msq;
        struct ipc_perm *ipcp;
        struct msg *tmsg, *leastp = NULL;
        struct msg *nmsg = NULL;
        int id;

        if (msqid < 0 || (long) msgsz < 0)
                return -EINVAL;

        id = (unsigned int) msqid % MSGMNI;
        msq = msgque [id];
        if (msq == IPC_NOID || msq == IPC_UNUSED)
                return -EINVAL;
        ipcp = &msq->msg_perm;

        /*
         *  find message of correct type.
         *  msgtyp = 0 => get first.
         *  msgtyp > 0 => get first message of matching type.
         *  msgtyp < 0 => get message with least type must be < abs(msgtype).
         */
```

```
while (!nmsg) {
        if (msq->msg_perm.seq != (unsigned int) msqid / MSGMNI) {
                return -EIDRM;
        }
        if (ipcperms (ipcp, S_IRUGO)) {
                return -EACCES;
        }

        if (msgtyp == 0)
                nmsg = msq->msg_first;
        else if (msgtyp > 0) {
                if (msgflg & MSG_EXCEPT) {
                        for (tmsg = msq->msg_first; tmsg;
                            tmsg = tmsg->msg_next)
                                if (tmsg->msg_type != msgtyp)
                                        break;
                        nmsg = tmsg;
                } else {
                        for (tmsg = msq->msg_first; tmsg;
                            tmsg = tmsg->msg_next)
                                if (tmsg->msg_type == msgtyp)
                                        break;
                        nmsg = tmsg;
                }
        } else {
                for (leastp = tmsg = msq->msg_first; tmsg;
                    tmsg = tmsg->msg_next)
                        if (tmsg->msg_type < leastp->msg_type)
                                leastp = tmsg;
                if (leastp && leastp->msg_type <= - msgtyp)
                        nmsg = leastp;
        }

        if (nmsg) { /* done finding a message */
                if ((msgsz < nmsg->msg_ts) && !(msgflg & MSG_NOERROR)) {
                        return -E2BIG;
                }
                msgsz = (msgsz > nmsg->msg_ts)? nmsg->msg_ts : msgsz;
                if (nmsg ==  msq->msg_first)
                        msq->msg_first = nmsg->msg_next;
                else {
                        for (tmsg = msq->msg_first; tmsg;
                            tmsg = tmsg->msg_next)
                                if (tmsg->msg_next == nmsg)
                                        break;
                        tmsg->msg_next = nmsg->msg_next;
                        if (nmsg == msq->msg_last)
                                msq->msg_last = tmsg;
                }
                if (!(--msq->msg_qnum))
```

```
                        msq->msg_last = msq->msg_first = NULL;

                    msq->msg_rtime = CURRENT_TIME;
                    msq->msg_lrpid = current->pid;
                    msgbytes -= nmsg->msg_ts;
                    msghdrs--;
                    msq->msg_cbytes -= nmsg->msg_ts;
                    wake_up (&msq->wwait);
                    if (put_user (nmsg->msg_type, &msgp->mtype) ||
                       copy_to_user (msgp->mtext, nmsg->msg_spot, msgsz))
                        msgsz = -EFAULT;
                    kfree(nmsg);
                    return msgsz;
                } else {  /* did not find a message */
                    if (msgflg & IPC_NOWAIT) {
                        return -ENOMSG;
                    }
                    if (signal_pending(current)) {
                        return -EINTR;
                    }
                    interruptible_sleep_on (&msq->rwait);
                }
        } /* end while */
        return -1;
}

asmlinkage int sys_msgsnd (int msqid, struct msgbuf *msgp, size_t msgsz, int
msgflg)
{
        int ret;

        lock_kernel();
        ret = real_msgsnd(msqid, msgp, msgsz, msgflg);
        unlock_kernel();
        return ret;
}

asmlinkage int sys_msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz,
        long msgtyp, int msgflg)
{
        int ret;

        lock_kernel();
        ret = real_msgrcv (msqid, msgp, msgsz, msgtyp, msgflg);
        unlock_kernel();
        return ret;
}

static int findkey (key_t key)
{
```

```
        int id;
        struct msqid_ds *msq;

        for (id = 0; id <= max_msqid; id++) {
                while ((msq = msgque[id]) == IPC_NOID)
                        interruptible_sleep_on (&msg_lock);
                if (msq == IPC_UNUSED)
                        continue;
                if (key == msq->msg_perm.key)
                        return id;
        }
        return -1;
}

static int newque (key_t key, int msgflg)
{
        int id;
        struct msqid_ds *msq;
        struct ipc_perm *ipcp;

        for (id = 0; id < MSGMNI; id++)
                if (msgque[id] == IPC_UNUSED) {
                        msgque[id] = (struct msqid_ds *) IPC_NOID;
                        goto found;
                }
        return -ENOSPC;

found:
        msq = (struct msqid_ds *) kmalloc (sizeof (*msq), GFP_KERNEL);
        if (!msq) {
                msgque[id] = (struct msqid_ds *) IPC_UNUSED;
                wake_up (&msg_lock);
                return -ENOMEM;
        }
        ipcp = &msq->msg_perm;
        ipcp->mode = (msgflg & S_IRWXUGO);
        ipcp->key = key;
        ipcp->cuid = ipcp->uid = current->euid;
        ipcp->gid = ipcp->cgid = current->egid;
        msq->msg_perm.seq = msg_seq;
        msq->msg_first = msq->msg_last = NULL;
        msq->rwait = msq->wwait = NULL;
        msq->msg_cbytes = msq->msg_qnum = 0;
        msq->msg_lspid = msq->msg_lrpid = 0;
        msq->msg_stime = msq->msg_rtime = 0;
        msq->msg_qbytes = MSGMNB;
        msq->msg_ctime = CURRENT_TIME;
        if (id > max_msqid)
                max_msqid = id;
        msgque[id] = msq;
```

```
        used_queues++;
        wake_up (&msg_lock);
        return (unsigned int) msq->msg_perm.seq * MSGMNI + id;
}

asmlinkage int sys_msgget (key_t key, int msgflg)
{
        int id, ret = -EPERM;
        struct msqid_ds *msq;

        lock_kernel();
        if (key == IPC_PRIVATE)
                ret = newque(key, msgflg);
        else if ((id = findkey (key)) == -1) { /* key not used */
                if (!(msgflg & IPC_CREAT))
                        ret = -ENOENT;
                else
                        ret = newque(key, msgflg);
        } else if (msgflg & IPC_CREAT && msgflg & IPC_EXCL) {
                ret = -EEXIST;
        } else {
                msq = msgque[id];
                if (msq == IPC_UNUSED || msq == IPC_NOID)
                        ret = -EIDRM;
                else if (ipcperms(&msq->msg_perm, msgflg))
                        ret = -EACCES;
                else
                        ret = (unsigned int) msq->msg_perm.seq * MSGMNI + id;
        }
        unlock_kernel();
        return ret;
}

static void freeque (int id)
{
        struct msqid_ds *msq = msgque[id];
        struct msg *msgp, *msgh;

        msq->msg_perm.seq++;
        msg_seq = (msg_seq+1) % ((unsigned)(1<<31)/MSGMNI); /* increment, but
avoid overflow */
        msgbytes -= msq->msg_cbytes;
        if (id == max_msqid)
                while (max_msqid && (msgque[--max_msqid] == IPC_UNUSED));
        msgque[id] = (struct msqid_ds *) IPC_UNUSED;
        used_queues--;
        while (waitqueue_active(&msq->rwait) || waitqueue_active(&msq->wwait)) {
                wake_up (&msq->rwait);
                wake_up (&msq->wwait);
                schedule();
```

```
        }
        for (msgp = msq->msg_first; msgp; msgp = msgh ) {
                msgh = msgp->msg_next;
                msghdrs--;
                kfree(msgp);
        }
        kfree(msq);
}

asmlinkage int sys_msgctl (int msqid, int cmd, struct msqid_ds *buf)
{
        int id, err = -EINVAL;
        struct msqid_ds *msq;
        struct msqid_ds tbuf;
        struct ipc_perm *ipcp;

        lock_kernel();
        if (msqid < 0 || cmd < 0)
                goto out;
        err = -EFAULT;
        switch (cmd) {
        case IPC_INFO:
        case MSG_INFO:
                if (!buf)
                        goto out;
        {
                struct msginfo msginfo;
                msginfo.msgmni = MSGMNI;
                msginfo.msgmax = MSGMAX;
                msginfo.msgmnb = MSGMNB;
                msginfo.msgmap = MSGMAP;
                msginfo.msgpool = MSGPOOL;
                msginfo.msgtql = MSGTQL;
                msginfo.msgssz = MSGSSZ;
                msginfo.msgseg = MSGSEG;
                if (cmd == MSG_INFO) {
                        msginfo.msgpool = used_queues;
                        msginfo.msgmap = msghdrs;
                        msginfo.msgtql = msgbytes;
                }

                err = -EFAULT;
                if (copy_to_user (buf, &msginfo, sizeof(struct msginfo)))
                        goto out;
                err = max_msqid;
                goto out;
        }
        case MSG_STAT:
                if (!buf)
                        goto out;
```

```
                err = -EINVAL;
                if (msqid > max_msqid)
                        goto out;
                msq = msgque[msqid];
                if (msq == IPC_UNUSED || msq == IPC_NOID)
                        goto out;
                err = -EACCES;
                if (ipcperms (&msq->msg_perm, S_IRUGO))
                        goto out;
                id = (unsigned int) msq->msg_perm.seq * MSGMNI + msqid;
                tbuf.msg_perm   = msq->msg_perm;
                tbuf.msg_stime  = msq->msg_stime;
                tbuf.msg_rtime  = msq->msg_rtime;
                tbuf.msg_ctime  = msq->msg_ctime;
                tbuf.msg_cbytes = msq->msg_cbytes;
                tbuf.msg_qnum   = msq->msg_qnum;
                tbuf.msg_qbytes = msq->msg_qbytes;
                tbuf.msg_lspid  = msq->msg_lspid;
                tbuf.msg_lrpid  = msq->msg_lrpid;
                err = -EFAULT;
                if (copy_to_user (buf, &tbuf, sizeof(*buf)))
                        goto out;
                err = id;
                goto out;
        case IPC_SET:
                if (!buf)
                        goto out;
                err = -EFAULT;
                if (!copy_from_user (&tbuf, buf, sizeof (*buf)))
                        err = 0;
                break;
        case IPC_STAT:
                if (!buf)
                        goto out;
                break;
        }

        id = (unsigned int) msqid % MSGMNI;
        msq = msgque [id];
        err = -EINVAL;
        if (msq == IPC_UNUSED || msq == IPC_NOID)
                goto out;
        err = -EIDRM;
        if (msq->msg_perm.seq != (unsigned int) msqid / MSGMNI)
                goto out;
        ipcp = &msq->msg_perm;

        switch (cmd) {
        case IPC_STAT:
                err = -EACCES;
```

```
                if (ipcperms (ipcp, S_IRUGO))
                        goto out;
                tbuf.msg_perm   = msq->msg_perm;
                tbuf.msg_stime  = msq->msg_stime;
                tbuf.msg_rtime  = msq->msg_rtime;
                tbuf.msg_ctime  = msq->msg_ctime;
                tbuf.msg_cbytes = msq->msg_cbytes;
                tbuf.msg_qnum   = msq->msg_qnum;
                tbuf.msg_qbytes = msq->msg_qbytes;
                tbuf.msg_lspid  = msq->msg_lspid;
                tbuf.msg_lrpid  = msq->msg_lrpid;
                err = -EFAULT;
                if (!copy_to_user (buf, &tbuf, sizeof (*buf)))
                        err = 0;
                goto out;
        case IPC_SET:
                err = -EPERM;
                if (current->euid != ipcp->cuid &&
                    current->euid != ipcp->uid && !capable(CAP_SYS_ADMIN))
                    /* We _could_ check for CAP_CHOWN above, but we don't */
                        goto out;
                if (tbuf.msg_qbytes > MSGMNB && !capable(CAP_SYS_RESOURCE))
                        goto out;
                msq->msg_qbytes = tbuf.msg_qbytes;
                ipcp->uid = tbuf.msg_perm.uid;
                ipcp->gid =  tbuf.msg_perm.gid;
                ipcp->mode = (ipcp->mode & ~S_IRWXUGO) |
                        (S_IRWXUGO & tbuf.msg_perm.mode);
                msq->msg_ctime = CURRENT_TIME;
                err = 0;
                goto out;
        case IPC_RMID:
                err = -EPERM;
                if (current->euid != ipcp->cuid &&
                    current->euid != ipcp->uid && !capable(CAP_SYS_ADMIN))
                        goto out;

                freeque (id);
                err = 0;
                goto out;
        default:
                err = -EINVAL;
                goto out;
        }
out:
        unlock_kernel();
        return err;
}
```