46

SYSTEM V MESSAGE QUEUES

This chapter describes System V message queues. Message queues allow processes to exchange data in the form of messages. Although message queues are similar to pipes and FIFOs in some respects, they also differ in important ways:

- The handle used to refer to a message queue is the identifier returned by a call to *msgget()*. These identifiers are not the same as the file descriptors used for most other forms of I/O on UNIX systems.
- Communication via message queues is message-oriented; that is, the reader receives whole messages, as written by the writer. It is not possible to read part of a message, leaving the remainder in the queue, or to read multiple messages at a time. This contrasts with pipes, which provide an undifferentiated stream of bytes (i.e., with pipes, the reader can read an arbitrary number of bytes at a time, irrespective of the size of data blocks written by the writer).
- In addition to containing data, each message has an integer *type*. Messages can be retrieved from a queue in first-in, first-out order or retrieved by type.

At the end of this chapter (Section 46.9), we summarize a number of limitations of System V message queues. These limitations lead us to the conclusion that, where possible, new applications should avoid the use of System V message queues in favor of other IPC mechanisms such as FIFOs, POSIX message queues, and sockets. However, when message queues were initially devised, these alternative mechanisms were unavailable or were not widespread across UNIX implementations.

Consequently, there are various existing applications that employ message queues, and this fact forms one of the primary motivations for describing them.

46.1 Creating or Opening a Message Queue

The *msgget()* system call creates a new message queue or obtains the identifier of an existing queue.

```
#include <sys/types.h>        /* For portability */
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

Returns message queue identifier on success, or −1 on error

The *key* argument is a key generated using one of the methods described in Section 45.2 (i.e., usually the value `IPC_PRIVATE` or a key returned by *ftok()*). The *msgflg* argument is a bit mask that specifies the permissions (Table 15-4, on page 295) to be placed on a new message queue or checked against an existing queue. In addition, zero or more of the following flags can be ORed (|) in *msgflg* to control the operation of *msgget()*:

`IPC_CREAT`
: If no message queue with the specified *key* exists, create a new queue.

`IPC_EXCL`
: If `IPC_CREAT` was also specified, and a queue with the specified *key* already exists, fail with the error `EEXIST`.

These flags are described in more detail in Section 45.1.

The *msgget()* system call begins by searching the set of all existing message queues for one with the specified key. If a matching queue is found, the identifier of that queue is returned (unless both `IPC_CREAT` and `IPC_EXCL` were specified in *msgflg*, in which case an error is returned). If no matching queue was found and `IPC_CREAT` was specified in *msgflg*, a new queue is created and its identifier is returned.

The program in Listing 46-1 provides a command-line interface to the *msgget()* system call. The program permits the use of command-line options and arguments to specify all possibilities for the *key* and *msgflg* arguments to *msgget()*. Details of the command format accepted by this program are shown in the *usageError()* function. Upon successful queue creation, the program prints the queue identifier. We demonstrate the use of this program in Section 46.2.2.

Listing 46-1: Using *msgget()*

svmsg/svmsg_create.c

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

```
static void             /* Print usage info, then exit */
usageError(const char *progName, const char *msg)
{
    if (msg != NULL)
        fprintf(stderr, "%s", msg);
    fprintf(stderr, "Usage: %s [-cx] {-f pathname | -k key | -p} "
                        "[octal-perms]\n", progName);
    fprintf(stderr, "    -c           Use IPC_CREAT flag\n");
    fprintf(stderr, "    -x           Use IPC_EXCL flag\n");
    fprintf(stderr, "    -f pathname  Generate key using ftok()\n");
    fprintf(stderr, "    -k key       Use 'key' as key\n");
    fprintf(stderr, "    -p           Use IPC_PRIVATE key\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int numKeyFlags;            /* Counts -f, -k, and -p options */
    int flags, msqid, opt;
    unsigned int perms;
    long lkey;
    key_t key;

    /* Parse command-line options and arguments */

    numKeyFlags = 0;
    flags = 0;

    while ((opt = getopt(argc, argv, "cf:k:px")) != -1) {
        switch (opt) {
        case 'c':
            flags |= IPC_CREAT;
            break;

        case 'f':               /* -f pathname */
            key = ftok(optarg, 1);
            if (key == -1)
                errExit("ftok");
            numKeyFlags++;
            break;

        case 'k':               /* -k key (octal, decimal or hexadecimal) */
            if (sscanf(optarg, "%li", &lkey) != 1)
                cmdLineErr("-k option requires a numeric argument\n");
            key = lkey;
            numKeyFlags++;
            break;

        case 'p':
            key = IPC_PRIVATE;
            numKeyFlags++;
            break;
```

```
        case 'x':
            flags |= IPC_EXCL;
            break;

        default:
            usageError(argv[0], "Bad option\n");
        }
    }

    if (numKeyFlags != 1)
        usageError(argv[0], "Exactly one of the options -f, -k, "
                            "or -p must be supplied\n");

    perms = (optind == argc) ? (S_IRUSR | S_IWUSR) :
                getInt(argv[optind], GN_BASE_8, "octal-perms");

    msqid = msgget(key, flags | perms);
    if (msqid == -1)
        errExit("msgget");

    printf("%d\n", msqid);
    exit(EXIT_SUCCESS);
}
```

svmsg/svmsg_create.c

46.2 Exchanging Messages

The *msgsnd()* and *msgrcv()* system calls perform I/O on message queues. The first argument to both system calls (*msqid*) is a message queue identifier. The second argument, *msgp*, is a pointer to a programmer-defined structure used to hold the message being sent or received. This structure has the following general form:

```
struct mymsg {
    long mtype;                 /* Message type */
    char mtext[];               /* Message body */
}
```

This definition is really just shorthand for saying that the first part of a message contains the message type, specified as a long integer, while the remainder of the message is a programmer-defined structure of arbitrary length and content; it need not be an array of characters. Thus, the *mgsp* argument is typed as *void ** to allow it to be a pointer to any type of structure.

A zero-length *mtext* field is permitted, and is sometimes useful if the information to be conveyed can be encoded solely in the message type or if the existence of a message is in itself sufficient information for the receiving process.

46.2.1 Sending Messages

The *msgsnd()* system call writes a message to a message queue.

```
#include <sys/types.h>        /* For portability */
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Returns 0 on success, or –1 on error

To send a message with *msgsnd()*, we must set the *mtype* field of the message structure to a value greater than 0 (we see how this value is used when we discuss *msgrcv()* in the next section) and copy the desired information into the programmer-defined *mtext* field. The *msgsz* argument specifies the number of bytes contained in the *mtext* field.

> When sending messages with *msgsnd()*, there is no concept of a partial write as with *write()*. This is why a successful *msgsnd()* needs only to return 0, rather than the number of bytes sent.

The final argument, *msgflg*, is a bit mask of flags controlling the operation of *msgsnd()*. Only one such flag is defined:

`IPC_NOWAIT`
: Perform a nonblocking send. Normally, if a message queue is full, *msgsnd()* blocks until enough space has become available to allow the message to be placed on the queue. However, if this flag is specified, then *msgsnd()* returns immediately with the error `EAGAIN`.

A *msgsnd()* call that is blocked because the queue is full may be interrupted by a signal handler. In this case, *msgsnd()* always fails with the error `EINTR`. (As noted in Section 21.5, *msgsnd()* is among those system calls that are never automatically restarted, regardless of the setting of the `SA_RESTART` flag when the signal handler is established.)

Writing a message to a message queue requires write permission on the queue.

The program in Listing 46-2 provides a command-line interface to the *msgsnd()* system call. The command-line format accepted by this program is shown in the *usageError()* function. Note that this program doesn't use the *msgget()* system call. (We noted that a process doesn't need to use a *get* call in order to access an IPC object in Section 45.1.) Instead, we specify the message queue by providing its identifier as a command-line argument. We demonstrate the use of this program in Section 46.2.2.

Listing 46-2: Using *msgsnd()* to send a message

svmsg/svmsg_send.c

```
#include <sys/types.h>
#include <sys/msg.h>
#include "tlpi_hdr.h"

#define MAX_MTEXT 1024

struct mbuf {
    long mtype;                     /* Message type */
    char mtext[MAX_MTEXT];          /* Message body */
};
```

```
static void             /* Print (optional) message, then usage description */
usageError(const char *progName, const char *msg)
{
    if (msg != NULL)
        fprintf(stderr, "%s", msg);
    fprintf(stderr, "Usage: %s [-n] msqid msg-type [msg-text]\n", progName);
    fprintf(stderr, "    -n       Use IPC_NOWAIT flag\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int msqid, flags, msgLen;
    struct mbuf msg;                    /* Message buffer for msgsnd() */
    int opt;                            /* Option character from getopt() */

    /* Parse command-line options and arguments */

    flags = 0;
    while ((opt = getopt(argc, argv, "n")) != -1) {
        if (opt == 'n')
            flags |= IPC_NOWAIT;
        else
            usageError(argv[0], NULL);
    }

    if (argc < optind + 2 || argc > optind + 3)
        usageError(argv[0], "Wrong number of arguments\n");

    msqid = getInt(argv[optind], 0, "msqid");
    msg.mtype = getInt(argv[optind + 1], 0, "msg-type");

    if (argc > optind + 2) {            /* 'msg-text' was supplied */
        msgLen = strlen(argv[optind + 2]) + 1;
        if (msgLen > MAX_MTEXT)
            cmdLineErr("msg-text too long (max: %d characters)\n", MAX_MTEXT);

        memcpy(msg.mtext, argv[optind + 2], msgLen);

    } else {                            /* No 'msg-text' ==> zero-length msg */
        msgLen = 0;
    }

    /* Send message */

    if (msgsnd(msqid, &msg, msgLen, flags) == -1)
        errExit("msgsnd");

    exit(EXIT_SUCCESS);
}
```

svmsg/svmsg_send.c

46.2.2 Receiving Messages

The *msgrcv()* system call reads (and removes) a message from a message queue, and copies its contents into the buffer pointed to by *msgp*.

```
#include <sys/types.h>        /* For portability */
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void *msgp, size_t maxmsgsz, long msgtyp, int msgflg);
                                Returns number of bytes copied into mtext field, or –1 on error
```

The maximum space available in the *mtext* field of the *msgp* buffer is specified by the argument *maxmsgsz*. If the body of the message to be removed from the queue exceeds *maxmsgsz* bytes, then no message is removed from the queue, and *msgrcv()* fails with the error `E2BIG`. (This default behavior can be changed using the `MSG_NOERROR` flag described shortly.)

Messages need not be read in the order in which they were sent. Instead, we can select messages according to the value in the *mtype* field. This selection is controlled by the *msgtyp* argument, as follows:

- If *msgtyp* equals 0, the first message from the queue is removed and returned to the calling process.
- If *msgtyp* is greater than 0, the first message in the queue whose *mtype* equals *msgtyp* is removed and returned to the calling process. By specifying different values for *msgtyp*, multiple processes can read from a message queue without racing to read the same messages. One useful technique is to have each process select messages matching its process ID.
- If *msgtyp* is less than 0, treat the waiting messages as a *priority queue*. The first message of the lowest *mtype* less than or equal to the absolute value of *msgtyp* is removed and returned to the calling process.

An example helps clarify the behavior when *msgtyp* is less than 0. Suppose that we have a message queue containing the sequence of messages shown in Figure 46-1 and we performed a series of *msgrcv()* calls of the following form:

```
msgrcv(id, &msg, maxmsgsz, -300, 0);
```

These *msgrcv()* calls would retrieve messages in the order 2 (type 100), 5 (type 100), 3 (type 200), and 1 (type 300). A further call would block, since the type of the remaining message (400) exceeds 300.

The *msgflg* argument is a bit mask formed by ORing together zero or more of the following flags:

`IPC_NOWAIT`
: Perform a nonblocking receive. Normally, if no message matching *msgtyp* is in the queue, *msgrcv()* blocks until such a message becomes available. Specifying the `IPC_NOWAIT` flag causes *msgrcv()* to instead return immediately

with the error ENOMSG. (The error EAGAIN would be more consistent, as occurs on a nonblocking *msgsnd()* or a nonblocking read from a FIFO. However, failing with ENOMSG is historical behavior, and required by SUSv3.)

MSG_EXCEPT

This flag has an effect only if *msgtyp* is greater than 0, in which case it forces the complement of the usual operation; that is, the first message from the queue whose *mtype* is *not* equal to *msgtyp* is removed from the queue and returned to the caller. This flag is Linux-specific, and is made available from <sys/msg.h> only if _GNU_SOURCE is defined. Performing a series of calls of the form *msgrcv(id, &msg, maxmsgsz, 100, MSG_EXCEPT)* on the message queue shown in Figure 46-1 would retrieve messages in the order 1, 3, 4, and then block.

MSG_NOERROR

By default, if the size of the *mtext* field of the message exceeds the space available (as defined by the *maxmsgsz* argument), *msgrcv()* fails. If the MSG_NOERROR flag is specified, then *msgrcv()* instead removes the message from the queue, truncates its *mtext* field to *maxmsgsz* bytes, and returns it to the caller. The truncated data is lost.

Upon successful completion, *msgrcv()* returns the size of the *mtext* field of the received message; on error, −1 is returned.

queue position	Message type (*mtype*)	Message body (*mtext*)
1	300	...
2	100	...
3	200	...
4	400	...
5	100	...

Figure 46-1: Example of a message queue containing messages of different types

As with *msgsnd()*, if a blocked *msgrcv()* call is interrupted by a signal handler, then the call fails with the error EINTR, regardless of the setting of the SA_RESTART flag when the signal handler was established.

Reading a message from a message queue requires read permission on the queue.

Example program

The program in Listing 46-3 provides a command-line interface to the *msgrcv()* system call. The command-line format accepted by this program is shown in the *usageError()* function. Like the program in Listing 46-2, which demonstrated the use of *msgsnd()*, this program doesn't use the *msgget()* system call, but instead expects a message queue identifier as its command-line argument.

The following shell session demonstrates the use of the programs in Listing 46-1, Listing 46-2, and Listing 46-3. We begin by creating a message queue using the IPC_PRIVATE key, and then write three messages with different types to the queue:

```
$ ./svmsg_create -p
32769                                                    ID of message queue
$ ./svmsg_send 32769 20 "I hear and I forget."
$ ./svmsg_send 32769 10 "I see and I remember."
$ ./svmsg_send 32769 30 "I do and I understand."
```

We then use the program in Listing 46-3 to read messages with a type less than or equal to 20 from the queue:

```
$ ./svmsg_receive -t -20 32769
Received: type=10; length=22; body=I see and I remember.
$ ./svmsg_receive -t -20 32769
Received: type=20; length=21; body=I hear and I forget.
$ ./svmsg_receive -t -20 32769
```

The last of the above commands blocked, because there was no message in the queue whose type was less than or equal to 20. So, we continue by typing *Control-C* to terminate the command, and then execute a command that reads a message of any type from the queue:

```
Type Control-C to terminate program
$ ./svmsg_receive 32769
Received: type=30; length=23; body=I do and I understand.
```

Listing 46-3: Using *msgrcv()* to read a message

svmsg/svmsg_receive.c

```
#define _GNU_SOURCE     /* Get definition of MSG_EXCEPT */
#include <sys/types.h>
#include <sys/msg.h>
#include "tlpi_hdr.h"

#define MAX_MTEXT 1024

struct mbuf {
    long mtype;                 /* Message type */
    char mtext[MAX_MTEXT];      /* Message body */
};

static void
usageError(const char *progName, const char *msg)
{
    if (msg != NULL)
        fprintf(stderr, "%s", msg);
    fprintf(stderr, "Usage: %s [options] msqid [max-bytes]\n", progName);
    fprintf(stderr, "Permitted options are:\n");
    fprintf(stderr, "    -e       Use MSG_NOERROR flag\n");
    fprintf(stderr, "    -t type  Select message of given type\n");
    fprintf(stderr, "    -n       Use IPC_NOWAIT flag\n");
```

```
#ifdef MSG_EXCEPT
    fprintf(stderr, "    -x       Use MSG_EXCEPT flag\n");
#endif
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int msqid, flags, type;
    ssize_t msgLen;
    size_t maxBytes;
    struct mbuf msg;            /* Message buffer for msgrcv() */
    int opt;                    /* Option character from getopt() */

    /* Parse command-line options and arguments */

    flags = 0;
    type = 0;
    while ((opt = getopt(argc, argv, "ent:x")) != -1) {
        switch (opt) {
        case 'e':       flags |= MSG_NOERROR;   break;
        case 'n':       flags |= IPC_NOWAIT;    break;
        case 't':       type = atoi(optarg);    break;
#ifdef MSG_EXCEPT
        case 'x':       flags |= MSG_EXCEPT;    break;
#endif
        default:        usageError(argv[0], NULL);
        }
    }

    if (argc < optind + 1 || argc > optind + 2)
        usageError(argv[0], "Wrong number of arguments\n");

    msqid = getInt(argv[optind], 0, "msqid");
    maxBytes = (argc > optind + 1) ?
                getInt(argv[optind + 1], 0, "max-bytes") : MAX_MTEXT;

    /* Get message and display on stdout */

    msgLen = msgrcv(msqid, &msg, maxBytes, type, flags);
    if (msgLen == -1)
        errExit("msgrcv");

    printf("Received: type=%ld; length=%ld", msg.mtype, (long) msgLen);
    if (msgLen > 0)
        printf("; body=%s", msg.mtext);
    printf("\n");

    exit(EXIT_SUCCESS);
}
```

svmsg/svmsg_receive.c

46.3 Message Queue Control Operations

The *msgctl()* system call performs control operations on the message queue identified by *msqid*.

```
#include <sys/types.h>        /* For portability */
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
                                                    Returns 0 on success, or –1 on error
```

The *cmd* argument specifies the operation to be performed on the queue. It can be one of the following:

`IPC_RMID`
: Immediately remove the message queue object and its associated *msqid_ds* data structure. All messages remaining in the queue are lost, and any blocked reader or writer processes are immediately awakened, with *msgsnd()* or *msgrcv()* failing with the error `EIDRM`. The third argument to *msgctl()* is ignored for this operation.

`IPC_STAT`
: Place a copy of the *msqid_ds* data structure associated with this message queue in the buffer pointed to by *buf*. We describe the *msqid_ds* structure in Section 46.4.

`IPC_SET`
: Update selected fields of the *msqid_ds* data structure associated with this message queue using values provided in the buffer pointed to by *buf*.

Further details about these operations, including the privileges and permissions required by the calling process, are described in Section 45.3. We describe some other values for *cmd* in Section 46.6.

The program in Listing 46-4 demonstrates the use of *msgctl()* to delete a message queue.

Listing 46-4: Deleting System V message queues

svmsg/svmsg_rm.c

```
#include <sys/types.h>
#include <sys/msg.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int j;
```

```
    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [msqid...]\n", argv[0]);

    for (j = 1; j < argc; j++)
        if (msgctl(getInt(argv[j], 0, "msqid"), IPC_RMID, NULL) == -1)
            errExit("msgctl %s", argv[j]);

    exit(EXIT_SUCCESS);
}
```

svmsg/svmsg_rm.c

46.4 Message Queue Associated Data Structure

Each message queue has an associated *msqid_ds* data structure of the following form:

```
struct msqid_ds {
    struct ipc_perm msg_perm;           /* Ownership and permissions */
    time_t          msg_stime;          /* Time of last msgsnd() */
    time_t          msg_rtime;          /* Time of last msgrcv() */
    time_t          msg_ctime;          /* Time of last change */
    unsigned long   __msg_cbytes;       /* Number of bytes in queue */
    msgqnum_t       msg_qnum;           /* Number of messages in queue */
    msglen_t        msg_qbytes;         /* Maximum bytes in queue */
    pid_t           msg_lspid;          /* PID of last msgsnd() */
    pid_t           msg_lrpid;          /* PID of last msgrcv() */
};
```

> The purpose of the abbreviated spelling *msq* in the name *msqid_ds* is to confuse the programmer. This is the only message queue interface employing this spelling.

The *msgqnum_t* and *msglen_t* data types—used to type the *msg_qnum* and *msg_qbytes* fields—are unsigned integer types specified in SUSv3.

The fields of the *msqid_ds* structure are implicitly updated by the various message queue system calls, and certain fields can be explicitly updated using the *msgctl()* `IPC_SET` operation. The details are as follows:

msg_perm
: When the message queue is created, the fields of this substructure are initialized as described in Section 45.3. The *uid*, *gid*, and *mode* subfields can be updated via `IPC_SET`.

msg_stime
: When the queue is created, this field is set to 0; each later successful *msgsnd()* sets this field to the current time. This field and the other timestamp fields in the *msqid_ds* structure are typed as *time_t*; they store time in seconds since the Epoch.

msg_rtime
: This field is set to 0 when the message queue is created, and then set to the current time on each successful *msgrcv()*.

msg_ctime
: This field is set to the current time when the message queue is created and whenever an IPC_SET operation is successfully performed.

__msg_cbytes
: This field is set to 0 when the message queue is created, and then adjusted during each successful *msgsnd()* and *msgrcv()* to reflect the total number of bytes contained in the *mtext* fields of all messages in the queue.

msg_qnum
: When the message queue is created, this field is set to 0. It is then incremented by each successful *msgsnd()* and decremented by each successful *msgrcv()* to reflect the total number of messages in the queue.

msg_qbytes
: The value in this field defines an upper limit on the number of bytes in the *mtext* fields of all messages in the message queue. This field is initialized to the value of the MSGMNB limit when the queue is created. A privileged (CAP_SYS_RESOURCE) process can use the IPC_SET operation to adjust *msg_qbytes* to any value in the range 0 to INT_MAX (2,147,483,647 on 32-bit platforms) bytes. An unprivileged process can adjust *msg_qbytes* to any value in the range 0 to MSGMNB. A privileged user can modify the value contained in the Linux-specific /proc/sys/kernel/msgmnb file in order to change the initial *msg_qbytes* setting for all subsequently created message queues, as well as the upper limit for subsequent changes to *msg_qbytes* by unprivileged processes. We say more about message queue limits in Section 46.5.

msg_lspid
: This field is set to 0 when the queue is created, and then set to the process ID of the calling process on each successful *msgsnd()*.

msg_lrpid
: This field is set to 0 when the message queue is created, and then set to the process ID of the calling process on each successful *msgrcv()*.

All of the above fields are specified by SUSv3, with the exception of *__msg_cbytes*. Nevertheless, most UNIX implementations provide an equivalent of the *__msg_cbytes* field.

The program in Listing 46-5 demonstrates the use of the IPC_STAT and IPC_SET operations to modify the *msg_qbytes* setting of a message queue.

Listing 46-5: Changing the *msg_qbytes* setting of a System V message queue

svmsg/svmsg_chqbytes.c

```
#include <sys/types.h>
#include <sys/msg.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    struct msqid_ds ds;
    int msqid;
```

```
    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s msqid max-bytes\n", argv[0]);

    /* Retrieve copy of associated data structure from kernel */

    msqid = getInt(argv[1], 0, "msqid");
    if (msgctl(msqid, IPC_STAT, &ds) == -1)
        errExit("msgctl");

    ds.msg_qbytes = getInt(argv[2], 0, "max-bytes");

    /* Update associated data structure in kernel */

    if (msgctl(msqid, IPC_SET, &ds) == -1)
        errExit("msgctl");

    exit(EXIT_SUCCESS);
}
```

svmsg/svmsg_chqbytes.c

46.5 Message Queue Limits

Most UNIX implementations impose various limits on the operation of System V message queues. Here, we describe the limits under Linux and note a few differences from other UNIX implementations.

The following limits are enforced on Linux. The system call affected by the limit and the error that results if the limit is reached are noted in parentheses.

MSGMNI
: This is a system-wide limit on the number of message queue identifiers (in other words, message queues) that can be created. (*msgget()*, ENOSPC)

MSGMAX
: This is a system-wide limit specifying the maximum number of (*mtext*) bytes that can be written in a single message. (*msgsnd()*, EINVAL)

MSGMNB
: This is the maximum number of (*mtext*) bytes that can be held in a message queue at one time. This limit is a system-wide parameter that is used to initialize the *msg_qbytes* field of the *msqid_ds* data structure associated with this message queue. Subsequently, the *msg_qbytes* value can be modified on a per-queue basis, as described in Section 46.4. If a queue's *msg_qbytes* limit is reached, then *msgsnd()* blocks, or fails with the error EAGAIN if IPC_NOWAIT was set.

Some UNIX implementations also define the following further limits:

MSGTQL
: This is a system-wide limit on the number of messages that can be placed on all message queues on the system.

MSGPOOL
: This is a system-wide limit on the size of the buffer pool that is used to hold data in all message queues on the system.

Although Linux doesn't impose either of the above limits, it does limit the number of messages on an individual queue to the value specified by the queue's *msg_qbytes* setting. This limitation is relevant only if we are writing zero-length messages to a queue. It has the effect that the limit on the number of zero-length messages is the same as the limit on the number of 1-byte messages that can be written to the queue. This is necessary to prevent an infinite number of zero-length messages being written to the queue. Although they contain no data, each zero-length message consumes a small amount of memory for system bookkeeping overhead.

At system startup, the message queue limits are set to default values. These defaults have varied somewhat across kernel versions. (Some distributors' kernels set different defaults from those provided by vanilla kernels.) On Linux, the limits can be viewed or changed via files in the `/proc` file system. Table 46-1 shows the `/proc` file corresponding to each limit. As an example, here are the default limits that we see for Linux 2.6.31 on one x86-32 system:

```
$ cd /proc/sys/kernel
$ cat msgmni
748
$ cat msgmax
8192
$ cat msgmnb
16384
```

Table 46-1: System V message queue limits

Limit	Ceiling value (x86-32)	Corresponding file in /proc/sys/kernel
MSGMNI	32768 (IPCMNI)	msgmni
MSGMAX	Depends on available memory	msgmax
MSGMNB	2147483647 (INT_MAX)	msgmnb

The ceiling value column of Table 46-1 shows the maximum value to which each limit can be raised on the x86-32 architecture. Note that although the `MSGMNB` limit can be raised to the value `INT_MAX`, some other limit (e.g., lack of memory) will be reached before a message queue can be loaded with so much data.

The Linux-specific *msgctl()* `IPC_INFO` operation retrieves a structure of type *msginfo*, which contains the values of the various message queue limits:

```
struct msginfo buf;

msgctl(0, IPC_INFO, (struct msqid_ds *) &buf);
```

Details about `IPC_INFO` and the *msginfo* structure can be found in the *msgctl(2)* manual page.

46.6 Displaying All Message Queues on the System

In Section 45.7, we looked at one way to obtain a list of all of the IPC objects on the system: via a set of files in the `/proc` file system. We now look at a second method of obtaining the same information: via a set of Linux-specific IPC *ctl* (*msgctl()*, *semctl()*,

and *shmctl()*) operations. (The *ipcs* program employs these operations.) These operations are as follows:

- MSG_INFO, SEM_INFO, and SHM_INFO: The MSG_INFO operation serves two purposes. First, it returns a structure detailing resources consumed by all message queues on the system. Second, as the function result of the *ctl* call, it returns the index of the maximum item in the *entries* array pointing to data structures for the message queue objects (see Figure 45-1, on page 932). The SEM_INFO and SHM_INFO operations perform an analogous task for semaphore sets and shared memory segments, respectively. We must define the _GNU_SOURCE feature test macro to obtain the definitions of these three constants from the corresponding System V IPC header files.

  An example showing the use of MSG_INFO to retrieve a *msginfo* structure containing information about resources used by all message queue objects is provided in the file svmsg/svmsg_info.c in the source code distribution for this book.

- MSG_STAT, SEM_STAT, and SHM_STAT: Like the IPC_STAT operation, these operations retrieve the associated data structure for an IPC object. However, they differ in two respects. First, instead of expecting an IPC identifier as the first argument of the *ctl* call, these operations expect an index into the *entries* array. Second, if the operation is successful, then, as its function result, the *ctl* call returns the identifier of the IPC object corresponding to that index. We must define the _GNU_SOURCE feature test macro to obtain the definitions of these three constants from the corresponding System V IPC header files.

To list all message queues on the system, we can do the following:

1. Use a MSG_INFO operation to find out the maximum index (*maxind*) of the *entries* array for message queues.
2. Perform a loop for all values from 0 up to and including *maxind*, employing a MSG_STAT operation for each value. During this loop, we ignore the errors that may occur if an item of the *entries* array is empty (EINVAL) or if we don't have permissions on the object to which it refers (EACCES).

Listing 46-6 provides an implementation of the above steps for message queues. The following shell session log demonstrates the use of this program:

```
$ ./svmsg_ls
maxind: 4

index     ID       key      messages
   2    98306  0x00000000       0
   4   163844  0x000004d2       2
$ ipcs -q                                  Check above against output of ipcs

------ Message Queues --------
key        msqid      owner    perms    used-bytes   messages
0x00000000 98306      mtk      600      0            0
0x000004d2 163844     mtk      600      12           2
```

Listing 46-6: Displaying all System V message queues on the system

svmsg/svmsg_ls.c

```
#define _GNU_SOURCE
#include <sys/types.h>
#include <sys/msg.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int maxind, ind, msqid;
    struct msqid_ds ds;
    struct msginfo msginfo;

    /* Obtain size of kernel 'entries' array */

    maxind = msgctl(0, MSG_INFO, (struct msqid_ds *) &msginfo);
    if (maxind == -1)
        errExit("msgctl-MSG_INFO");

    printf("maxind: %d\n\n", maxind);
    printf("index     id       key      messages\n");

    /* Retrieve and display information from each element of 'entries' array */

    for (ind = 0; ind <= maxind; ind++) {
        msqid = msgctl(ind, MSG_STAT, &ds);
        if (msqid == -1) {
            if (errno != EINVAL && errno != EACCES)
                errMsg("msgctl-MSG_STAT");              /* Unexpected error */
            continue;                                   /* Ignore this item */
        }

        printf("%4d %8d  0x%08lx %7ld\n", ind, msqid,
                (unsigned long) ds.msg_perm.__key, (long) ds.msg_qnum);
    }

    exit(EXIT_SUCCESS);
}
```

svmsg/svmsg_ls.c

46.7 Client-Server Programming with Message Queues

In this section, we consider two of various possible designs for client-server applications using System V message queues:

- The use of a single message queue for exchanging messages in both directions between server and client.
- The use of separate message queues for the server and for each client. The server's queue is used to receive incoming client requests, and responses are sent to clients via the individual client queues.

Which approach we choose depends on the requirements of our application. We next consider some of the factors that may influence our choice.

Using a single message queue for server and clients

Using a single message queue may be suitable when the messages exchanged between servers and clients are small. However, note the following points:

- Since multiple processes may attempt to read messages at the same time, we must use the message type (*mtype*) field to allow each process to select only those messages intended for it. One way to accomplish this is to use the client's process ID as the message type for messages sent from the server to the client. The client can send its process ID as part of its message(s) to the server. Furthermore, messages to the server must also be distinguished by a unique message type. For this purpose, we can use the number 1, which, being the process ID of the permanently running *init* process, can never be the process ID of a client process. (An alternative would be to use the server's process ID as the message type; however, it is difficult for the clients to obtain this information.) This numbering scheme is shown in Figure 46-2.
- Message queues have a limited capacity. This has the potential to cause a couple of problems. One of these is that multiple simultaneous clients could fill the message queue, resulting in a deadlock situation, where no new client requests can be submitted and the server is blocked from writing any responses. The other problem is that a poorly behaved or intentionally malicious client may fail to read responses from the server. This can lead to the queue becoming clogged with unread messages, preventing any communication between clients and server. (Using two queues—one for messages from clients to the server, and the other for messages from the server to the clients—would solve the first of these problems, but not the second.)

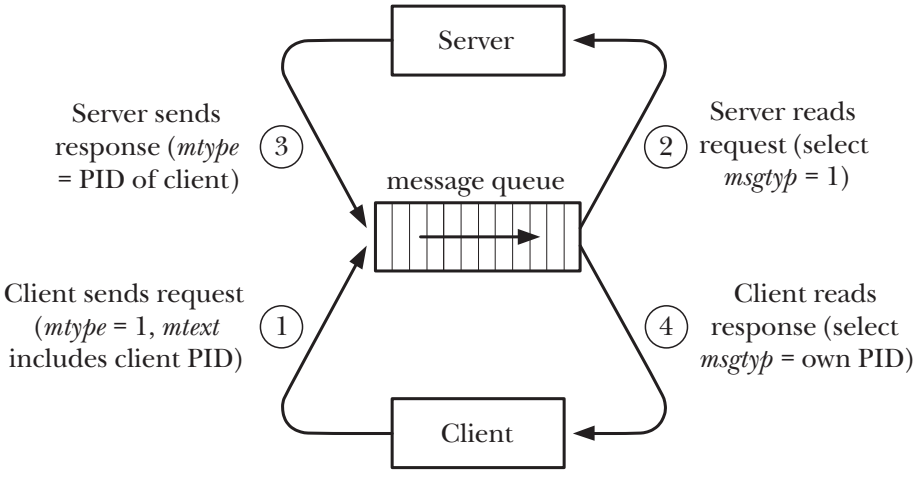

Figure 46-2: Using a single message queue for client-server IPC

Using one message queue per client

Using one message queue per client (as well as one for the server) is preferable where large messages need to be exchanged, or where there is potential for the

problems listed above when using a single message queue. Note the following points regarding this approach:

- Each client must create its own message queue (typically using the `IPC_PRIVATE` key) and inform the server of the queue's identifier, usually by transmitting the identifier as part of the client's message(s) to the server.
- There is a system-wide limit (`MSGMNI`) on the number of message queues, and the default value for this limit is quite low on some systems. If we expect to have a large number of simultaneous clients, we may need to raise this limit.
- The server should allow for the possibility that the client's message queue no longer exists (perhaps because the client prematurely deleted it).

We say more about using one message queue per client in the next section.

46.8 A File-Server Application Using Message Queues

In this section, we describe a client-server application that uses one message queue per client. The application is a simple file server. The client sends a request message to the server's message queue asking for the contents of a named file. The server responds by returning the file contents as a series of messages to the client's private message queue. Figure 46-3 provides an overview of the application.

Because the server performs no authentication of the client, any user that can run the client can obtain any of the files accessible to the server. A more sophisticated server would require some type of authentication from the client before serving the requested file.

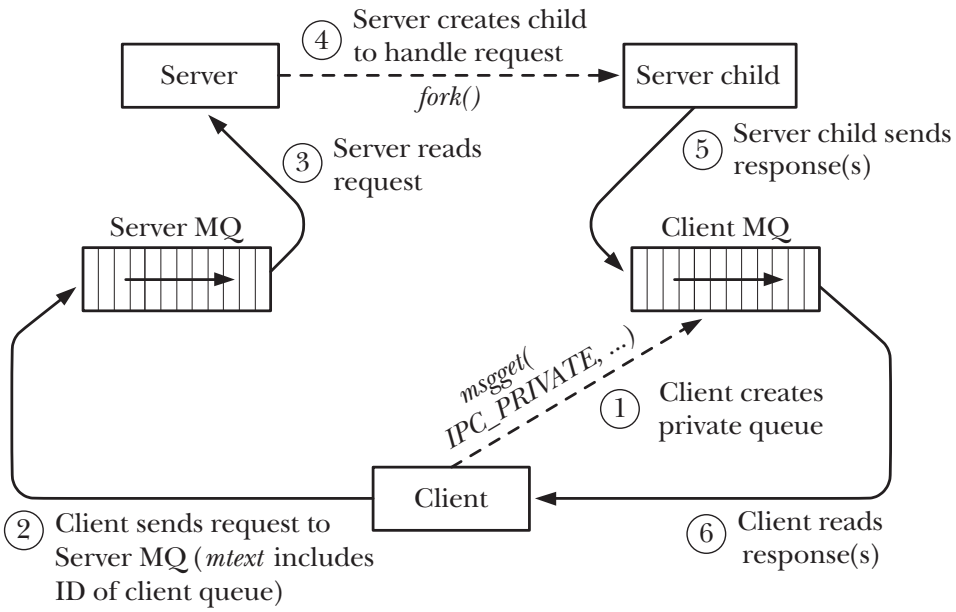

Figure 46-3: Client-server IPC using one message queue per client

Common header file

Listing 46-7 is the header file included by both the server and the client. This header defines the well-known key to be used for the server's message queue

(SERVER_KEY), and defines the formats of the messages to be passed between the client and the server.

The *requestMsg* structure defines the format of the request sent from the client to the server. In this structure, the *mtext* component consists of two fields: the identifier of the client's message queue and the pathname of the file requested by the client. The constant REQ_MSG_SIZE equates to the combined size of these two fields and is used as the *msgsz* argument in calls to *msgsnd()* using this structure.

The *responseMsg* structure defines the format of the response messages sent from the server back to the client. The *mtype* field is used in response messages to supply information about the message content, as defined by the RESP_MT_* constants.

Listing 46-7: Header file for svmsg_file_server.c and svmsg_file_client.c

svmsg/svmsg_file.h

```
#include <sys/types.h>
#include <sys/msg.h>
#include <sys/stat.h>
#include <stddef.h>                     /* For definition of offsetof() */
#include <limits.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/wait.h>
#include "tlpi_hdr.h"

#define SERVER_KEY 0x1aaaaaa1           /* Key for server's message queue */

struct requestMsg {                     /* Requests (client to server) */
    long mtype;                         /* Unused */
    int  clientId;                      /* ID of client's message queue */
    char pathname[PATH_MAX];            /* File to be returned */
};

/* REQ_MSG_SIZE computes size of 'mtext' part of 'requestMsg' structure.
   We use offsetof() to handle the possibility that there are padding
   bytes between the 'clientId' and 'pathname' fields. */

#define REQ_MSG_SIZE (offsetof(struct requestMsg, pathname) - \
                      offsetof(struct requestMsg, clientId) + PATH_MAX)

#define RESP_MSG_SIZE 8192

struct responseMsg {                    /* Responses (server to client) */
    long mtype;                         /* One of RESP_MT_* values below */
    char data[RESP_MSG_SIZE];           /* File content / response message */
};

/* Types for response messages sent from server to client */

#define RESP_MT_FAILURE 1               /* File couldn't be opened */
#define RESP_MT_DATA    2               /* Message contains file data */
#define RESP_MT_END     3               /* File data complete */
```

svmsg/svmsg_file.h

Server program

Listing 46-8 is the server program for the application. Note the following points about the server:

- The server is designed to handle requests concurrently. A concurrent server design is preferable to the iterative design employed in Listing 44-7 (page 912), since we want to avoid the possibility that a client request for a large file would cause all other client requests to wait.
- Each client request is handled by creating a child process that serves the requested file ⑧. In the meantime, the main server process waits upon further client requests. Note the following points about the server child:
  - Since the child produced via *fork()* inherits a copy of the parent's stack, it thus obtains a copy of the request message read by the main server process.
  - The server child terminates after handling its associated client request ⑨.
- In order to avoid the creation of zombie processes (Section 26.2), the server establishes a handler for `SIGCHLD` ⑥ and calls *waitpid()* within this handler ①.
- The *msgrcv()* call in the parent server process may block, and consequently be interrupted by the `SIGCHLD` handler. To handle this possibility, a loop is used to restart the call if it fails with the `EINTR` error ⑦.
- The server child executes the *serveRequest()* function ②, which sends three message types back to the client. A request with an *mtype* of `RESP_MT_FAILURE` indicates that the server could not open the requested file ③; `RESP_MT_DATA` is used for a series of messages containing file data ④; and `RESP_MT_END` (with a zero-length *data* field) is used to indicate that transmission of file data is complete ⑤.

We consider a number of ways to improve and extend the server program in Exercise 46-4.

Listing 46-8: A file server using System V message queues

svmsg/svmsg_file_server.c

```
#include "svmsg_file.h"

static void             /* SIGCHLD handler */
grimReaper(int sig)
{
    int savedErrno;

    savedErrno = errno;                 /* waitpid() might change 'errno' */
①   while (waitpid(-1, NULL, WNOHANG) > 0)
        continue;
    errno = savedErrno;
}
```

```
   static void             /* Executed in child process: serve a single client */
② serveRequest(const struct requestMsg *req)
   {
       int fd;
       ssize_t numRead;
       struct responseMsg resp;

       fd = open(req->pathname, O_RDONLY);
       if (fd == -1) {                     /* Open failed: send error text */
③          resp.mtype = RESP_MT_FAILURE;
           snprintf(resp.data, sizeof(resp.data), "%s", "Couldn't open");
           msgsnd(req->clientId, &resp, strlen(resp.data) + 1, 0);
           exit(EXIT_FAILURE);              /* and terminate */
       }

       /* Transmit file contents in messages with type RESP_MT_DATA. We don't
          diagnose read() and msgsnd() errors since we can't notify client. */

④      resp.mtype = RESP_MT_DATA;
       while ((numRead = read(fd, resp.data, RESP_MSG_SIZE)) > 0)
           if (msgsnd(req->clientId, &resp, numRead, 0) == -1)
               break;

       /* Send a message of type RESP_MT_END to signify end-of-file */

⑤      resp.mtype = RESP_MT_END;
       msgsnd(req->clientId, &resp, 0, 0);         /* Zero-length mtext */
   }

   int
   main(int argc, char *argv[])
   {
       struct requestMsg req;
       pid_t pid;
       ssize_t msgLen;
       int serverId;
       struct sigaction sa;

       /* Create server message queue */

       serverId = msgget(SERVER_KEY, IPC_CREAT | IPC_EXCL |
                                 S_IRUSR | S_IWUSR | S_IWGRP);
       if (serverId == -1)
           errExit("msgget");

       /* Establish SIGCHLD handler to reap terminated children */

       sigemptyset(&sa.sa_mask);
       sa.sa_flags = SA_RESTART;
       sa.sa_handler = grimReaper;
⑥      if (sigaction(SIGCHLD, &sa, NULL) == -1)
           errExit("sigaction");
```

```
    /* Read requests, handle each in a separate child process */

    for (;;) {
        msgLen = msgrcv(serverId, &req, REQ_MSG_SIZE, 0, 0);
        if (msgLen == -1) {
⑦           if (errno == EINTR)         /* Interrupted by SIGCHLD handler? */
                continue;               /* ... then restart msgrcv() */
            errMsg("msgrcv");           /* Some other error */
            break;                      /* ... so terminate loop */
        }

⑧       pid = fork();                   /* Create child process */
        if (pid == -1) {
            errMsg("fork");
            break;
        }

        if (pid == 0) {                 /* Child handles request */
            serveRequest(&req);
⑨           _exit(EXIT_SUCCESS);
        }

        /* Parent loops to receive next client request */
    }

    /* If msgrcv() or fork() fails, remove server MQ and exit */

    if (msgctl(serverId, IPC_RMID, NULL) == -1)
        errExit("msgctl");
    exit(EXIT_SUCCESS);
}
```

svmsg/svmsg_file_server.c

Client program

Listing 46-9 is the client program for the application. Note the following:

- The client creates a message queue with the `IPC_PRIVATE` key ② and uses *atexit()* ③ to establish an exit handler ① to ensure that the queue is deleted when the client exits.
- The client passes the identifier for its queue, as well as the pathname of the file to be served, in a request to the server ④.
- The client handles the possibility that the first response message sent by the server may be a failure notification (*mtype* equals `RESP_MT_FAILURE`) by printing the text of the error message returned by the server and exiting ⑤.
- If the file is successfully opened, then the client loops ⑥, receiving a series of messages containing the file contents (*mtype* equals `RESP_MT_DATA`). The loop is terminated by receipt of an end-of-file message (*mtype* equals `RESP_MT_END`).

This simple client doesn't handle various possibilities resulting from failures in the server. We consider some improvements in Exercise 46-5.

Listing 46-9: Client for file server using System V message queues

svmsg/svmsg_file_client.c

```
#include "svmsg_file.h"

static int clientId;

static void
removeQueue(void)
{
    if (msgctl(clientId, IPC_RMID, NULL) == -1)
①       errExit("msgctl");
}

int
main(int argc, char *argv[])
{
    struct requestMsg req;
    struct responseMsg resp;
    int serverId, numMsgs;
    ssize_t msgLen, totBytes;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pathname\n", argv[0]);

    if (strlen(argv[1]) > sizeof(req.pathname) - 1)
        cmdLineErr("pathname too long (max: %ld bytes)\n",
                (long) sizeof(req.pathname) - 1);

    /* Get server's queue identifier; create queue for response */

    serverId = msgget(SERVER_KEY, S_IWUSR);
    if (serverId == -1)
        errExit("msgget - server message queue");

②   clientId = msgget(IPC_PRIVATE, S_IRUSR | S_IWUSR | S_IWGRP);
    if (clientId == -1)
        errExit("msgget - client message queue");

③   if (atexit(removeQueue) != 0)
        errExit("atexit");

    /* Send message asking for file named in argv[1] */

    req.mtype = 1;                      /* Any type will do */
    req.clientId = clientId;
    strncpy(req.pathname, argv[1], sizeof(req.pathname) - 1);
    req.pathname[sizeof(req.pathname) - 1] = '\0';
                                        /* Ensure string is terminated */

④   if (msgsnd(serverId, &req, REQ_MSG_SIZE, 0) == -1)
        errExit("msgsnd");
```

```
    /* Get first response, which may be failure notification */

    msgLen = msgrcv(clientId, &resp, RESP_MSG_SIZE, 0, 0);
    if (msgLen == -1)
        errExit("msgrcv");

⑤   if (resp.mtype == RESP_MT_FAILURE) {
        printf("%s\n", resp.data);      /* Display msg from server */
        if (msgctl(clientId, IPC_RMID, NULL) == -1)
            errExit("msgctl");
        exit(EXIT_FAILURE);
    }

    /* File was opened successfully by server; process messages
       (including the one already received) containing file data */

    totBytes = msgLen;                  /* Count first message */
⑥   for (numMsgs = 1; resp.mtype == RESP_MT_DATA; numMsgs++) {
        msgLen = msgrcv(clientId, &resp, RESP_MSG_SIZE, 0, 0);
        if (msgLen == -1)
            errExit("msgrcv");

        totBytes += msgLen;
    }

    printf("Received %ld bytes (%d messages)\n", (long) totBytes, numMsgs);

    exit(EXIT_SUCCESS);
}
```

svmsg/svmsg_file_client.c

The following shell session demonstrates the use of the programs in Listing 46-8 and Listing 46-9:

```
$ ./svmsg_file_server &                 Run server in background
[1] 9149
$ wc -c /etc/services                   Show size of file that client will request
764360 /etc/services
$ ./svmsg_file_client /etc/services
Received 764360 bytes (95 messages)     Bytes received matches size above
$ kill %1                               Terminate server
[1]+  Terminated        ./svmsg_file_server
```

46.9 Disadvantages of System V Message Queues

UNIX systems provide a number of mechanisms for transmitting data from one process to another on the same system, either in the form of an undelimited byte stream (pipes, FIFOs, and UNIX domain stream sockets) or as delimited messages (System V message queues, POSIX message queues, and UNIX domain datagram sockets).

A distinctive feature of System V message queues is the ability to attach a numeric type to each message. This provides for two possibilities that may be useful to applications: reading processes may select messages by type, or they may employ

a priority-queue strategy so that higher-priority messages (i.e., those with lower message type values) are read first.

However, System V message queues have a number of disadvantages:

- Message queues are referred to by identifiers, rather than the file descriptors used by most other UNIX I/O mechanisms. This means that a variety of file descriptor–based I/O techniques described in Chapter 63 (e.g., *select()*, *poll()*, and *epoll*) can't be applied to message queues. Furthermore, writing programs that simultaneously handle inputs from both message queues and file descriptor–based I/O mechanisms requires code that is more complex than code that deals with file descriptors alone. (We look at one way of combining the two I/O models in Exercise 63-3.)
- The use of keys, rather than filenames, to identify message queues results in additional programming complexity and also requires the use of *ipcs* and *ipcrm* instead of *ls* and *rm*. The *ftok()* function usually generates a unique key, but it is not guaranteed to do so. Employing the `IPC_PRIVATE` key guarantees a unique queue identifier, but leaves us with the task of making that identifier visible to other processes that require it.
- Message queues are connectionless, and the kernel doesn't maintain a count of the number of processes referring to the queue as is done with pipes, FIFOs, and sockets. Consequently, it can be difficult to answer the following questions:
  - When is it safe for an application to delete a message queue? (Premature deletion of the queue results in immediate loss of data, regardless of whether any process might later be interested in reading from the queue.)
  - How can an application ensure that an unused queue is deleted?
- There are limits on the total number of message queues, the size of messages, and the capacity of individual queues. These limits are configurable, but if an application operates outside the range of the default limits, this requires extra work when installing the application.

In summary, System V message queues are often best avoided. In situations where we require the facility to select messages by type, we should consider alternatives. POSIX message queues (Chapter 52) are one such alternative. As a further alternative, solutions involving multiple file descriptor–based communication channels may provide functionality similar to selecting messages by type, while at the same time allowing the use of the alternative I/O models described in Chapter 63. For example, if we need to transmit "normal" and "priority" messages, we could use a pair of FIFOs or UNIX domain sockets for the two message types, and then employ *select()* or *poll()* to monitor file descriptors for both channels.

46.10 Summary

System V message queues allow processes to communicate by exchanging messages consisting of a numeric type plus a body containing arbitrary data. The distinguishing features of message queues are that message boundaries are preserved and that the receiver(s) can select messages by type, rather than reading messages in first-in, first-out order.

Various factors led us to conclude that other IPC mechanisms are usually preferable to System V message queues. One major difficulty is that message queues are not referred to using file descriptors. This means that we can't employ various alternative I/O models with message queues; in particular, it is complex to simultaneously monitor both message queues and file descriptors to see if I/O is possible. Furthermore, the fact that message queues are connectionless (i.e., not reference counted) makes it difficult for an application to know when a queue may be deleted safely.

46.11 Exercises

46-1. Experiment with the programs in Listing 46-1 (`svmsg_create.c`), Listing 46-2 (`svmsg_send.c`), and Listing 46-3 (`svmsg_receive.c`) to confirm your understanding of the *msgget()*, *msgsnd()*, and *msgrcv()* system calls.

46-2. Recode the sequence-number client-server application of Section 44.8 to use System V message queues. Use a single message queue to transmit messages from both client to server and server to client. Employ the conventions for message types described in Section 46.8.

46-3. In the client-server application of Section 46.8, why does the client pass the identifier of its message queue in the body of the message (in the *clientId* field), rather than in the message type (*mtype*)?

46-4. Make the following changes to the client-server application of Section 46.8:

a) Replace the use of a hard-coded message queue key with code in the server that uses `IPC_PRIVATE` to generate a unique identifier, and then writes this identifier to a well-known file. The client must read the identifier from this file. The server should remove this file if it terminates.

b) In the *serveRequest()* function of the server program, system call errors are not diagnosed. Add code that logs errors using *syslog()* (Section 37.5).

c) Add code to the server so that it becomes a daemon on startup (Section 37.2).

d) In the server, add a handler for `SIGTERM` and `SIGINT` that performs a tidy exit. The handler should remove the message queue and (if the earlier part of this exercise was implemented) the file created to hold the server's message queue identifier. Include code in the handler to terminate the server by disestablishing the handler, and then once more raising the same signal that invoked the handler (see Section 26.1.4 for the rationale and steps required for this task).

e) The server child doesn't handle the possibility that the client may terminate prematurely, in which case the server child would fill the client's message queue, and then block indefinitely. Modify the server to handle this possibility by establishing a timeout when calling *msgsnd()*, as described in Section 23.3. If the server child deems that the client has disappeared, it should attempt to delete the client's message queue, and then exit (after perhaps logging a message via *syslog()*).

46-5. The client shown in Listing 46-9 (`svmsg_file_client.c`) doesn't handle various possibilities for failure in the server. In particular, if the server message queue fills up (perhaps because the server terminated and the queue was filled by other clients), then the *msgsnd()* call will block indefinitely. Similarly, if the server fails to send a response to the client, then the *msgrcv()* call will block indefinitely. Add code to the client that sets timeouts (Section 23.3) on these calls. If either call times out, then the program should report an error to the user and terminate.

46-6. Write a simple chat application (similar to *talk(1)*, but without the *curses* interface) using System V messages queues. Use a single message queue for each client.