52

POSIX MESSAGE QUEUES

This chapter describes POSIX message queues, which allow processes to exchange data in the form of messages. POSIX message queues are similar to their System V counterparts, in that data is exchanged in units of whole messages. However, there are also some notable differences:

- POSIX message queues are reference counted. A queue that is marked for deletion is removed only after it is closed by all processes that are currently using it.
- Each System V message has an integer type, and messages can be selected in a variety of ways using *msgrcv()*. By contrast, POSIX messages have an associated priority, and messages are always strictly queued (and thus received) in priority order.
- POSIX message queues provide a feature that allows a process to be asynchronously notified when a message is available on a queue.

POSIX message queues are a relatively recent addition to Linux. The required implementation support was added in kernel 2.6.6 (in addition, *glibc* 2.3.4 or later is required).

> POSIX message queue support is an optional kernel component that is configured via the `CONFIG_POSIX_MQUEUE` option.

52.1 Overview

The main functions in the POSIX message queue API are the following:

- The *mq_open()* function creates a new message queue or opens an existing queue, returning a message queue descriptor for use in later calls.
- The *mq_send()* function writes a message to a queue.
- The *mq_receive()* function reads a message from a queue.
- The *mq_close()* function closes a message queue that the process previously opened.
- The *mq_unlink()* function removes a message queue name and marks the queue for deletion when all processes have closed it.

The above functions all serve fairly obvious purposes. In addition, a couple of features are peculiar to the POSIX message queue API:

- Each message queue has an associated set of attributes. Some of these attributes can be set when the queue is created or opened using *mq_open()*. Two functions are provided to retrieve and change queue attributes: *mq_getattr()* and *mq_setattr()*.
- The *mq_notify()* function allows a process to register for message notification from a queue. After registering, the process is notified of the availability of a message by delivery of a signal or by the invocation of a function in a separate thread.

52.2 Opening, Closing, and Unlinking a Message Queue

In this section, we look at the functions used to open, close, and remove message queues.

Opening a message queue

The *mq_open()* function creates a new message queue or opens an existing queue.

```
#include <fcntl.h>           /* Defines O_* constants */
#include <sys/stat.h>        /* Defines mode constants */
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag, ...
              /* mode_t mode, struct mq_attr *attr */);
```

Returns a message queue descriptor on success, or *(mqd_t)* −1 on error

The *name* argument identifies the message queue, and is specified according to the rules given in Section 51.1.

The *oflag* argument is a bit mask that controls various aspects of the operation of *mq_open()*. The values that can be included in this mask are summarized in Table 52-1.

Table 52-1: Bit values for the *mq_open() oflag* argument

Flag	Description
O_CREAT	Create queue if it doesn't already exist
O_EXCL	With O_CREAT, create queue exclusively
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only
O_RDWR	Open for reading and writing
O_NONBLOCK	Open in nonblocking mode

One of the purposes of the *oflag* argument is to determine whether we are opening an existing queue or creating and opening a new queue. If *oflag* doesn't include O_CREAT, we are opening an existing queue. If *oflag* includes O_CREAT, a new, empty queue is created if one with the given *name* doesn't already exist. If *oflag* specifies both O_CREAT and O_EXCL, and a queue with the given *name* already exists, then *mq_open()* fails.

The *oflag* argument also indicates the kind of access that the calling process will make to the message queue, by specifying exactly one of the values O_RDONLY, O_WRONLY, or O_RDWR.

The remaining flag value, O_NONBLOCK, causes the queue to be opened in nonblocking mode. If a subsequent call to *mq_receive()* or *mq_send()* can't be performed without blocking, the call will fail immediately with the error EAGAIN.

If *mq_open()* is being used to open an existing message queue, the call requires only two arguments. However, if O_CREAT is specified in *flags*, two further arguments are required: *mode* and *attr*. (If the queue specified by *name* already exists, these two arguments are ignored.) These arguments are used as follows:

- The *mode* argument is a bit mask that specifies the permissions to be placed on the new message queue. The bit values that may be specified are the same as for files (Table 15-4, on page 295), and, as with *open()*, the value in *mode* is masked against the process umask (Section 15.4.6). To read from a queue (*mq_receive()*), read permission must be granted to the corresponding class of user; to write to a queue (*mq_send()*), write permission is required.
- The *attr* argument is an *mq_attr* structure that specifies attributes for the new message queue. If *attr* is NULL, the queue is created with implementation-defined default attributes. We describe the *mq_attr* structure in Section 52.4.

Upon successful completion, *mq_open()* returns a *message queue descriptor*, a value of type *mqd_t*, which is used in subsequent calls to refer to this open message queue. The only stipulation that SUSv3 makes about this data type is that it may not be an array; that is, it is guaranteed to be a type that can be used in an assignment statement or passed by value as a function argument. (On Linux, *mqd_t* is an *int*, but, for example, on Solaris it is defined as *void **.)

An example of the use of *mq_open()* is provided in Listing 52-2.

Effect of *fork()*, *exec()*, and process termination on message queue descriptors

During a *fork()*, the child process receives copies of its parent's message queue descriptors, and these descriptors refer to the same open message queue descriptions.

(We explain message queue descriptions in Section 52.3.) The child doesn't inherit any of its parent's message notification registrations.

When a process performs an *exec()* or terminates, all of its open message queue descriptors are closed. As a consequence of closing its message queue descriptors, all of the process's message notification registrations on the corresponding queues are deregistered.

Closing a message queue

The *mq_close()* function closes the message queue descriptor *mqdes*.

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
                                        Returns 0 on success, or –1 on error
```

If the calling process has registered via *mqdes* for message notification from the queue (Section 52.6), then the notification registration is automatically removed, and another process can subsequently register for message notification from the queue.

A message queue descriptor is automatically closed when a process terminates or calls *exec()*. As with file descriptors, we should explicitly close message queue descriptors that are no longer required, in order to prevent the process from running out of message queue descriptors.

As *close()* for files, closing a message queue doesn't delete it. For that purpose, we need *mq_unlink()*, which is the message queue analog of *unlink()*.

Removing a message queue

The *mq_unlink()* function removes the message queue identified by *name*, and marks the queue to be destroyed once all processes cease using it (this may mean immediately, if all processes that had the queue open have already closed it).

```
#include <mqueue.h>

int mq_unlink(const char *name);
                                        Returns 0 on success, or –1 on error
```

Listing 52-1 demonstrates the use of *mq_unlink()*.

Listing 52-1: Using *mq_unlink()* to unlink a POSIX message queue

pmsg/pmsg_unlink.c

```
#include <mqueue.h>
#include "tlpi_hdr.h"

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

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

    if (mq_unlink(argv[1]) == -1)
        errExit("mq_unlink");
    exit(EXIT_SUCCESS);
}
```

pmsg/pmsg_unlink.c

52.3 Relationship Between Descriptors and Message Queues

The relationship between a message queue descriptor and an open message queue is analogous to the relationship between a file descriptor and an open file (Figure 5-2, on page 95). A message queue descriptor is a per-process handle that refers to an entry in the system-wide table of open message queue descriptions, and this entry in turn refers to a message queue object. This relationship is illustrated in Figure 52-1.

> On Linux, POSIX message queues are implemented as i-nodes in a virtual file system, and message queue descriptors and open message queue descriptions are implemented as file descriptors and open file descriptions, respectively. However, these are implementation details that are not required by SUSv3 and don't hold true on some other UNIX implementations. Nevertheless, we return to this point in Section 52.7, because Linux provides some nonstandard features that are made possible by this implementation.

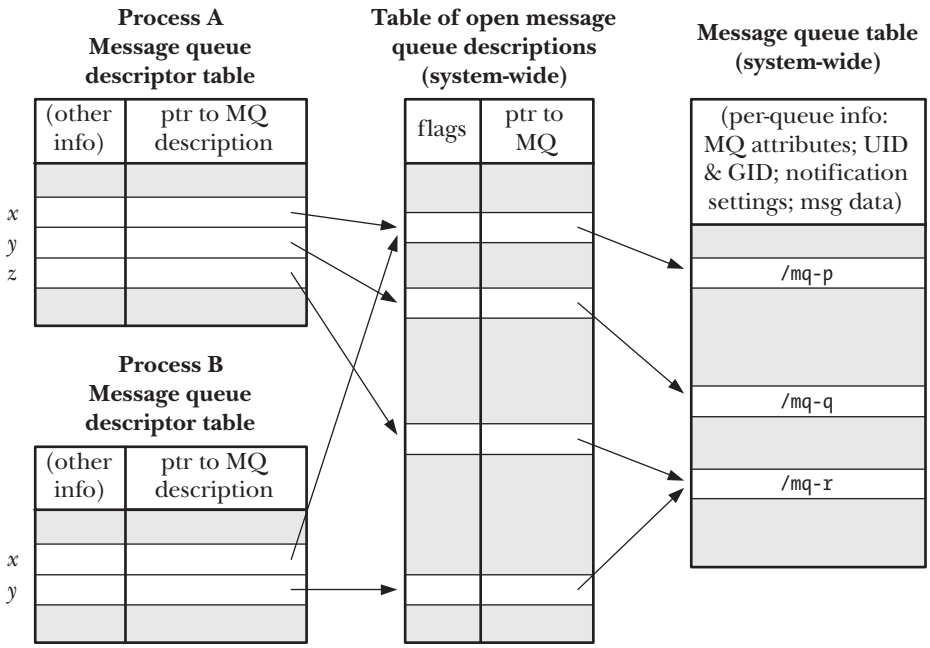

Figure 52-1: Relationship between kernel data structures for POSIX message queues

Figure 52-1 helps clarify a number of details of the use of message queue descriptors (all of which are analogous to the use to file descriptors):

- An open message queue description has an associated set of flags. SUSv3 specifies only one such flag, `O_NONBLOCK`, which determines whether I/O is nonblocking.
- Two processes can hold message queue descriptors (descriptor *x* in the diagram) that refer to the same open message queue description. This can occur because a process opens a message queue and then calls *fork()*. These descriptors share the state of the `O_NONBLOCK` flag.
- Two processes can hold open message queue descriptors that refer to different message queue descriptions that refer to the same message queue (e.g., descriptor *z* in process A and descriptor *y* in process B both refer to `/mq-r`). This occurs because the two processes each used *mq_open()* to open the same queue.

52.4 Message Queue Attributes

The *mq_open()*, *mq_getattr()*, and *mq_setattr()* functions all permit an argument that is a pointer to an *mq_attr* structure. This structure is defined in `<mqueue.h>` as follows:

```
struct mq_attr {
    long mq_flags;       /* Message queue description flags: 0 or
                            O_NONBLOCK [mq_getattr(), mq_setattr()] */
    long mq_maxmsg;      /* Maximum number of messages on queue
                            [mq_open(), mq_getattr()] */
    long mq_msgsize;     /* Maximum message size (in bytes)
                            [mq_open(), mq_getattr()] */
    long mq_curmsgs;     /* Number of messages currently in queue
                            [mq_getattr()] */
};
```

Before we look at the *mq_attr* structure in detail, it is worth noting the following points:

- Only some of the fields are used by each of the three functions. The fields used by each function are indicated in the comments accompanying the structure definition above.
- The structure contains information about the open message queue description (*mq_flags*) associated with a message descriptor and information about the queue referred to by that descriptor (*mq_maxmsg*, *mq_msgsize*, *mq_curmsgs*).
- Some of the fields contain information that is fixed at the time the queue is created with *mq_open()* (*mq_maxmsg* and *mq_msgsize*); the others return information about the current state of the message queue description (*mq_flags*) or message queue (*mq_curmsgs*).

Setting message queue attributes during queue creation

When we create a message queue with *mq_open()*, the following *mq_attr* fields determine the attributes of the queue:

- The *mq_maxmsg* field defines the limit on the number of messages that can be placed on the queue using *mq_send()*. This value must be greater than 0.

- The *mq_msgsize* field defines the upper limit on the size of each message that may be placed on the queue. This value must be greater than 0.

Together, these two values allow the kernel to determine the maximum amount of memory that this message queue may require.

The *mq_maxmsg* and *mq_msgsize* attributes are fixed when a message queue is created; they can't subsequently be changed. In Section 52.8, we describe two /proc files that place system-wide limits on the values that can be specified for the *mq_maxmsg* and *mq_msgsize* attributes.

The program in Listing 52-2 provides a command-line interface to the *mq_open()* function and shows how the *mq_attr* structure is used with *mq_open()*.

Two command-line options allow message queue attributes to be specified: *-m* for *mq_maxmsg* and *-s* for *mq_msgsize*. If either of these options is supplied, a non-NULL *attrp* argument is passed to *mq_open()*. Some default values are assigned to the fields of the *mq_attr* structure to which *attrp* points, in case only one of the *-m* and *-s* options is specified on the command line. If neither of these options is supplied, *attrp* is specified as NULL when calling *mq_open()*, which causes the queue to be created with the implementation-defined defaults for the queue attributes.

Listing 52-2: Creating a POSIX message queue

pmsg/pmsg_create.c

```
#include <mqueue.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

static void
usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s [-cx] [-m maxmsg] [-s msgsize] mq-name "
            "[octal-perms]\n", progName);
    fprintf(stderr, "    -c          Create queue (O_CREAT)\n");
    fprintf(stderr, "    -m maxmsg   Set maximum # of messages\n");
    fprintf(stderr, "    -s msgsize  Set maximum message size\n");
    fprintf(stderr, "    -x          Create exclusively (O_EXCL)\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int flags, opt;
    mode_t perms;
    mqd_t mqd;
    struct mq_attr attr, *attrp;

    attrp = NULL;
    attr.mq_maxmsg = 50;
    attr.mq_msgsize = 2048;
    flags = O_RDWR;
```

```
    /* Parse command-line options */

    while ((opt = getopt(argc, argv, "cm:s:x")) != -1) {
        switch (opt) {
        case 'c':
            flags |= O_CREAT;
            break;

        case 'm':
            attr.mq_maxmsg = atoi(optarg);
            attrp = &attr;
            break;

        case 's':
            attr.mq_msgsize = atoi(optarg);
            attrp = &attr;
            break;

        case 'x':
            flags |= O_EXCL;
            break;

        default:
            usageError(argv[0]);
        }
    }

    if (optind >= argc)
        usageError(argv[0]);

    perms = (argc <= optind + 1) ? (S_IRUSR | S_IWUSR) :
                getInt(argv[optind + 1], GN_BASE_8, "octal-perms");

    mqd = mq_open(argv[optind], flags, perms, attrp);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

    exit(EXIT_SUCCESS);
}
```

pmsg/pmsg_create.c

Retrieving message queue attributes

The *mq_getattr()* function returns an *mq_attr* structure containing information about the message queue description and the message queue associated with the descriptor *mqdes*.

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
```

Returns 0 on success, or −1 on error

In addition to the *mq_maxmsg* and *mq_msgsize* fields, which we have already described, the following fields are returned in the structure pointed to by *attr*:

mq_flags
: These are flags for the open message queue description associated with the descriptor *mqdes*. Only one such flag is specified: `O_NONBLOCK`. This flag is initialized from the *oflag* argument of *mq_open()*, and can be changed using *mq_setattr()*.

mq_curmsgs
: This is the number of messages that are currently in the queue. This information may already have changed by the time *mq_getattr()* returns, if other processes are reading messages from the queue or writing messages to it.

The program in Listing 52-3 employs *mq_getattr()* to retrieve the attributes for the message queue specified in its command-line argument, and then displays those attributes on standard output.

Listing 52-3: Retrieving POSIX message queue attributes

pmsg/pmsg_getattr.c

```
#include <mqueue.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    mqd_t mqd;
    struct mq_attr attr;

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

    mqd = mq_open(argv[1], O_RDONLY);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

    if (mq_getattr(mqd, &attr) == -1)
        errExit("mq_getattr");

    printf("Maximum # of messages on queue:   %ld\n", attr.mq_maxmsg);
    printf("Maximum message size:             %ld\n", attr.mq_msgsize);
    printf("# of messages currently on queue: %ld\n", attr.mq_curmsgs);
    exit(EXIT_SUCCESS);
}
```

pmsg/pmsg_getattr.c

In the following shell session, we use the program in Listing 52-2 to create a message queue with implementation-defined default attributes (i.e., the final argument to *mq_open()* is `NULL`), and then use the program in Listing 52-3 to display the queue attributes so that we can see the default settings on Linux.

```
$ ./pmsg_create -cx /mq
$ ./pmsg_getattr /mq
Maximum # of messages on queue:   10
Maximum message size:             8192
# of messages currently on queue: 0
$ ./pmsg_unlink /mq                                  Remove message queue
```

From the above output, we see that the Linux default values for *mq_maxmsg* and *mq_msgsize* are 10 and 8192, respectively.

There is a wide variation in the implementation-defined defaults for *mq_maxmsg* and *mq_msgsize*. Portable applications generally need to choose explicit values for these attributes, rather than relying on the defaults.

Modifying message queue attributes

The *mq_setattr()* function sets attributes of the message queue description associated with the message queue descriptor *mqdes*, and optionally returns information about the message queue.

```
#include <mqueue.h>

int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr,
               struct mq_attr *oldattr);
```

Returns 0 on success, or −1 on error

The *mq_setattr()* function performs the following tasks:

- It uses the *mq_flags* field in the *mq_attr* structure pointed to by *newattr* to change the flags of the message queue description associated with the descriptor *mqdes*.
- If *oldattr* is non-NULL, it returns an *mq_attr* structure containing the previous message queue description flags and message queue attributes (i.e., the same task as is performed by *mq_getattr()*).

The only attribute that SUSv3 specifies that can be changed using *mq_setattr()* is the state of the O_NONBLOCK flag.

Allowing for the possibility that a particular implementation may define other modifiable flags, or that SUSv3 may add new flags in the future, a portable application should change the state of the O_NONBLOCK flag by using *mq_getattr()* to retrieve the *mq_flags* value, modifying the O_NONBLOCK bit, and calling *mq_setattr()* to change the *mq_flags* settings. For example, to enable O_NONBLOCK, we would do the following:

```
if (mq_getattr(mqd, &attr) == -1)
    errExit("mq_getattr");
attr.mq_flags |= O_NONBLOCK;
if (mq_setattr(mqd, &attr, NULL) == -1)
    errExit("mq_getattr");
```

52.5 Exchanging Messages

In this section, we look at the functions that are used to send messages to and receive messages from a queue.

52.5.1 Sending Messages

The *mq_send()* function adds the message in the buffer pointed to by *msg_ptr* to the message queue referred to by the descriptor *mqdes*.

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
            unsigned int msg_prio);
                                                Returns 0 on success, or –1 on error
```

The *msg_len* argument specifies the length of the message pointed to by *msg_ptr*. This value must be less than or equal to the *mq_msgsize* attribute of the queue; otherwise, *mq_send()* fails with the error EMSGSIZE. Zero-length messages are permitted.

Each message has a nonnegative integer priority, specified by the *msg_prio* argument. Messages are ordered within the queue in descending order of priority (i.e., 0 is the lowest priority). When a new message is added to the queue, it is placed after any other messages of the same priority. If an application doesn't need to use message priorities, it is sufficient to always specify *msg_prio* as 0.

> As noted at the beginning of this chapter, the type attribute of System V messages provides different functionality. System V messages are always queued in FIFO order, but *msgrcv()* allows us to select messages in various ways: in FIFO order, by exact type, or by highest type less than or equal to some value.

SUSv3 allows an implementation to advertise its upper limit for message priorities, either by defining the constant MQ_PRIO_MAX or via the return from *sysconf(_SC_MQ_PRIO_MAX)*. SUSv3 requires this limit to be at least 32 (_POSIX_MQ_PRIO_MAX); that is, priorities at least in the range 0 to 31 are available. However, the actual range on implementations is highly variable. For example, on Linux, this constant has the value 32,768; on Solaris, it is 32; and on Tru64, it is 256.

If the message queue is already full (i.e., the *mq_maxmsg* limit for the queue has been reached), then a further *mq_send()* either blocks until space becomes available in the queue, or, if the O_NONBLOCK flag is in effect, fails immediately with the error EAGAIN.

The program in Listing 52-4 provides a command-line interface to the *mq_send()* function. We demonstrate the use of this program in the next section.

Listing 52-4: Writing a message to a POSIX message queue

pmsg/pmsg_send.c

```
#include <mqueue.h>
#include <fcntl.h>              /* For definition of O_NONBLOCK */
#include "tlpi_hdr.h"

static void
usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s [-n] name msg [prio]\n", progName);
    fprintf(stderr, "    -n           Use O_NONBLOCK flag\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int flags, opt;
    mqd_t mqd;
    unsigned int prio;

    flags = O_WRONLY;
    while ((opt = getopt(argc, argv, "n")) != -1) {
        switch (opt) {
        case 'n':   flags |= O_NONBLOCK;        break;
        default:    usageError(argv[0]);
        }
    }

    if (optind + 1 >= argc)
        usageError(argv[0]);

    mqd = mq_open(argv[optind], flags);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

    prio = (argc > optind + 2) ? atoi(argv[optind + 2]) : 0;

    if (mq_send(mqd, argv[optind + 1], strlen(argv[optind + 1]), prio) == -1)
        errExit("mq_send");
    exit(EXIT_SUCCESS);
}
```

pmsg/pmsg_send.c

52.5.2 Receiving Messages

The *mq_receive()* function removes the oldest message with the highest priority from the message queue referred to by *mqdes* and returns that message in the buffer pointed to by *msg_ptr*.

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
                   unsigned int *msg_prio);
```

Returns number of bytes in received message on success, or –1 on error

The *msg_len* argument is used by the caller to specify the number of bytes of space available in the buffer pointed to by *msg_ptr*.

Regardless of the actual size of the message, *msg_len* (and thus the size of the buffer pointed to by *msg_ptr*) must be greater than or equal to the *mq_msgsize* attribute of the queue; otherwise, *mq_receive()* fails with the error EMSGSIZE. If we don't know the value of the *mq_msgsize* attribute of a queue, we can obtain it using *mq_getattr()*. (In an application consisting of cooperating processes, the use of *mq_getattr()* can usually be dispensed with, because the application can typically decide on a queue's *mq_msgsize* setting in advance.)

If *msg_prio* is not NULL, then the priority of the received message is copied into the location pointed to by *msg_prio*.

If the message queue is currently empty, then *mq_receive()* either blocks until a message becomes available, or, if the O_NONBLOCK flag is in effect, fails immediately with the error EAGAIN. (There is no equivalent of the pipe behavior where a reader sees end-of-file if there are no writers.)

The program in Listing 52-5 provides a command-line interface to the *mq_receive()* function. The command format for this program is shown in the *usageError()* function.

The following shell session demonstrates the use of the programs in Listing 52-4 and Listing 52-5. We begin by creating a message queue and sending a few messages with different priorities:

```
$ ./pmsg_create -cx /mq
$ ./pmsg_send /mq msg-a 5
$ ./pmsg_send /mq msg-b 0
$ ./pmsg_send /mq msg-c 10
```

We then execute a series of commands to retrieve messages from the queue:

```
$ ./pmsg_receive /mq
Read 5 bytes; priority = 10
msg-c
$ ./pmsg_receive /mq
Read 5 bytes; priority = 5
msg-a
$ ./pmsg_receive /mq
Read 5 bytes; priority = 0
msg-b
```

As we can see from the above output, the messages were retrieved in order of priority.

At this point, the queue is now empty. When we perform another blocking receive, the operation blocks:

```
$ ./pmsg_receive /mq
Blocks; we type Control-C to terminate the program
```

On the other hand, if we perform a nonblocking receive, the call returns immediately with a failure status:

```
$ ./pmsg_receive -n /mq
ERROR [EAGAIN/EWOULDBLOCK Resource temporarily unavailable] mq_receive
```

Listing 52-5: Reading a message from a POSIX message queue

pmsg/pmsg_receive.c

```
#include <mqueue.h>
#include <fcntl.h>              /* For definition of O_NONBLOCK */
#include "tlpi_hdr.h"

static void
usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s [-n] name\n", progName);
    fprintf(stderr, "    -n           Use O_NONBLOCK flag\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int flags, opt;
    mqd_t mqd;
    unsigned int prio;
    void *buffer;
    struct mq_attr attr;
    ssize_t numRead;

    flags = O_RDONLY;
    while ((opt = getopt(argc, argv, "n")) != -1) {
        switch (opt) {
        case 'n':   flags |= O_NONBLOCK;        break;
        default:    usageError(argv[0]);
        }
    }

    if (optind >= argc)
        usageError(argv[0]);

    mqd = mq_open(argv[optind], flags);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

    if (mq_getattr(mqd, &attr) == -1)
        errExit("mq_getattr");

    buffer = malloc(attr.mq_msgsize);
    if (buffer == NULL)
        errExit("malloc");
```

```
    numRead = mq_receive(mqd, buffer, attr.mq_msgsize, &prio);
    if (numRead == -1)
        errExit("mq_receive");

    printf("Read %ld bytes; priority = %u\n", (long) numRead, prio);
    if (write(STDOUT_FILENO, buffer, numRead) == -1)
        errExit("write");
    write(STDOUT_FILENO, "\n", 1);

    exit(EXIT_SUCCESS);
}
```

pmsg/pmsg_receive.c

52.5.3 Sending and Receiving Messages with a Timeout

The *mq_timedsend()* and *mq_timedreceive()* functions are exactly like *mq_send()* and *mq_receive()*, except that if the operation can't be performed immediately, and the O_NONBLOCK flag is not in effect for the message queue description, then the *abs_timeout* argument specifies a limit on the time for which the call will block.

```
#define _XOPEN_SOURCE 600
#include <mqueue.h>
#include <time.h>

int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
                 unsigned int msg_prio, const struct timespec *abs_timeout);
                                        Returns 0 on success, or –1 on error

ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
                 unsigned int *msg_prio, const struct timespec *abs_timeout);
        Returns number of bytes in received message on success, or –1 on error
```

The *abs_timeout* argument is a *timespec* structure (Section 23.4.2) that specifies the timeout as an absolute value in seconds and nanoseconds since the Epoch. To perform a relative timeout, we can fetch the current value of the CLOCK_REALTIME clock using *clock_gettime()* and add the required amount to that value to produce a suitably initialized *timespec* structure.

If a call to *mq_timedsend()* or *mq_timedreceive()* times out without being able to complete its operation, then the call fails with the error ETIMEDOUT.

On Linux, specifying *abs_timeout* as NULL means an infinite timeout. However, this behavior is not specified in SUSv3, and portable applications can't rely on it.

The *mq_timedsend()* and *mq_timedreceive()* functions originally derive from POSIX.1d (1999) and are not available on all UNIX implementations.

52.6 Message Notification

A feature that distinguishes POSIX message queues from their System V counterparts is the ability to receive asynchronous notification of the availability of a message on a previously empty queue (i.e., when the queue transitions from being empty to nonempty). This feature means that instead of making a blocking *mq_receive()* call

or marking the message queue descriptor nonblocking and performing periodic *mq_receive()* calls ("polls") on the queue, a process can request a notification of message arrival and then perform other tasks until it is notified. A process can choose to be notified either via a signal or via invocation of a function in a separate thread.

> The notification feature of POSIX message queues is similar to the notification facility that we described for POSIX timers in Section 23.6. (Both of these APIs originated in POSIX.1b.)

The *mq_notify()* function registers the calling process to receive a notification when a message arrives on the empty queue referred to by the descriptor *mqdes*.

```
#include <mqueue.h>

int mq_notify(mqd_t mqdes, const struct sigevent *notification);
                                                    Returns 0 on success, or –1 on error
```

The *notification* argument specifies the mechanism by which the process is to be notified. Before going into the details of the *notification* argument, we note a few points about message notification:

- At any time, only one process ("the registered process") can be registered to receive a notification from a particular message queue. If there is already a process registered for a message queue, further attempts to register for that queue fail (*mq_notify()* fails with the error EBUSY).
- The registered process is notified only when a new message arrives on a queue that was previously empty. If a queue already contains messages at the time of the registration, a notification will occur only after the queue is emptied and a new message arrives.
- After one notification is sent to the registered process, the registration is removed, and any process can then register itself for notification. In other words, as long as a process wishes to keep receiving notifications, it must reregister itself after each notification by once again calling *mq_notify()*.
- The registered process is notified only if some other process is not currently blocked in a call to *mq_receive()* for the queue. If some other process is blocked in *mq_receive()*, that process will read the message, and the registered process will remain registered.
- A process can explicitly deregister itself as the target for message notification by calling *mq_notify()* with a *notification* argument of NULL.

We already showed the *sigevent* structure that is used to type the *notification* argument in Section 23.6.1. Here, we present the structure in simplified form, showing just those fields relevant to the discussion of *mq_notify()*:

```
union sigval {
    int    sival_int;             /* Integer value for accompanying data */
    void  *sival_ptr;             /* Pointer value for accompanying data */
};
```

```
struct sigevent {
    int    sigev_notify;          /* Notification method */
    int    sigev_signo;           /* Notification signal for SIGEV_SIGNAL */
    union sigval sigev_value;     /* Value passed to signal handler or
                                     thread function */
    void (*sigev_notify_function) (union sigval);
                                  /* Thread notification function */
    void  *sigev_notify_attributes;   /* Really 'pthread_attr_t' */
};
```

The *sigev_notify* field of this structure is set to one of the following values:

SIGEV_NONE
: Register this process for notification, but when a message arrives on the previously empty queue, don't actually notify the process. As usual, the registration is removed when a new messages arrives on an empty queue.

SIGEV_SIGNAL
: Notify the process by generating the signal specified in the *sigev_signo* field. If *sigev_signo* is a realtime signal, then the *sigev_value* field specifies data to accompany the signal (Section 22.8.1). This data can be retrieved via the *si_value* field of the *siginfo_t* structure that is passed to the signal handler or returned by a call to *sigwaitinfo()* or *sigtimedwait()*. The following fields in the *siginfo_t* structure are also filled in: *si_code*, with the value SI_MESGQ; *si_signo*, with the signal number; *si_pid*, with the process ID of the process that sent the message; and *si_uid*, with the real user ID of the process that sent the message. (The *si_pid* and *si_uid* fields are not set on most other implementations.)

SIGEV_THREAD
: Notify the process by calling the function specified in *sigev_notify_function* as if it were the start function in a new thread. The *sigev_notify_attributes* field can be specified as NULL or as a pointer to a *pthread_attr_t* structure that defines attributes for the thread (Section 29.8). The union *sigval* value specified in *sigev_value* is passed as the argument of this function.

52.6.1 Receiving Notification via a Signal

Listing 52-6 provides an example of message notification using signals. This program performs the following steps:

1. Open the message queue named on the command line in nonblocking mode ①, determine the *mq_msgsize* attribute for the queue ②, and allocate a buffer of that size for receiving messages ③.
2. Block the notification signal (SIGUSR1) and establish a handler for it ④.
3. Make an initial call to *mq_notify()* to register the process to receive message notification ⑤.
4. Execute an infinite loop that performs the following steps:
   a) Call *sigsuspend()*, which unblocks the notification signal and waits until the signal is caught ⑥. Return from this system call indicates that a message

notification has occurred. At this point, the process will have been deregistered for message notification.

b) Call *mq_notify()* to reregister this process to receive message notification ⑦.

c) Execute a `while` loop that drains the queue by reading as many messages as possible ⑧.

Listing 52-6: Receiving message notification via a signal

pmsg/mq_notify_sig.c

```
#include <signal.h>
#include <mqueue.h>
#include <fcntl.h>              /* For definition of O_NONBLOCK */
#include "tlpi_hdr.h"

#define NOTIFY_SIG SIGUSR1

static void
handler(int sig)
{
    /* Just interrupt sigsuspend() */
}

int
main(int argc, char *argv[])
{
    struct sigevent sev;
    mqd_t mqd;
    struct mq_attr attr;
    void *buffer;
    ssize_t numRead;
    sigset_t blockMask, emptyMask;
    struct sigaction sa;

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

①   mqd = mq_open(argv[1], O_RDONLY | O_NONBLOCK);
    if (mqd == (mqd_t) -1)
        errExit("mq_open");

②   if (mq_getattr(mqd, &attr) == -1)
        errExit("mq_getattr");

③   buffer = malloc(attr.mq_msgsize);
    if (buffer == NULL)
        errExit("malloc");

④   sigemptyset(&blockMask);
    sigaddset(&blockMask, NOTIFY_SIG);
    if (sigprocmask(SIG_BLOCK, &blockMask, NULL) == -1)
        errExit("sigprocmask");
```

```
        sigemptyset(&sa.sa_mask);
        sa.sa_flags = 0;
        sa.sa_handler = handler;
        if (sigaction(NOTIFY_SIG, &sa, NULL) == -1)
            errExit("sigaction");

⑤       sev.sigev_notify = SIGEV_SIGNAL;
        sev.sigev_signo = NOTIFY_SIG;
        if (mq_notify(mqd, &sev) == -1)
            errExit("mq_notify");

        sigemptyset(&emptyMask);

        for (;;) {
⑥           sigsuspend(&emptyMask);         /* Wait for notification signal */

⑦           if (mq_notify(mqd, &sev) == -1)
                errExit("mq_notify");

⑧           while ((numRead = mq_receive(mqd, buffer, attr.mq_msgsize, NULL)) >= 0)
                printf("Read %ld bytes\n", (long) numRead);

            if (errno != EAGAIN)            /* Unexpected error */
                errExit("mq_receive");
        }
    }
```

pmsg/mq_notify_sig.c

Various aspects of the program in Listing 52-6 merit further comment:

- We block the notification signal and use *sigsuspend()* to wait for it, rather than *pause()*, to prevent the possibility of missing a signal that is delivered while the program is executing elsewhere (i.e., is not blocked waiting for signals) in the `for` loop. If this occurred, and we were using *pause()* to wait for signals, then the next call to *pause()* would block, even though a signal had already been delivered.
- We open the queue in nonblocking mode, and, whenever a notification occurs, we use a `while` loop to read all messages from the queue. Emptying the queue in this way ensures that a further notification is generated when a new message arrives. Employing nonblocking mode means that the `while` loop will terminate (*mq_receive()* will fail with the error `EAGAIN`) when we have emptied the queue. (This approach is analogous to the use of nonblocking I/O with edge-triggered I/O notification, which we describe in Section 63.1.1, and is employed for similar reasons.)
- Within the `for` loop, it is important that we reregister for message notification *before* reading all messages from the queue. If we reversed these steps, the following sequence could occur: all messages are read from the queue, and the `while` loop terminates; another message is placed on the queue; *mq_notify()* is called to reregister for message notification. At this point, no further notification signal would be generated, because the queue is already nonempty. Consequently, the program would remain permanently blocked in its next call to *sigsuspend()*.

52.6.2 Receiving Notification via a Thread

Listing 52-7 provides an example of message notification using threads. This program shares a number of design features with the program in Listing 52-6:

- When message notification occurs, the program reenables notification before draining the queue ②.
- Nonblocking mode is employed so that, after receiving a notification, we can completely drain the queue without blocking ⑤.

Listing 52-7: Receiving message notification via a thread

pmsg/mq_notify_thread.c

```
#include <pthread.h>
#include <mqueue.h>
#include <fcntl.h>              /* For definition of O_NONBLOCK */
#include "tlpi_hdr.h"

static void notifySetup(mqd_t *mqdp);

static void                     /* Thread notification function */
① threadFunc(union sigval sv)
{
    ssize_t numRead;
    mqd_t *mqdp;
    void *buffer;
    struct mq_attr attr;

    mqdp = sv.sival_ptr;

    if (mq_getattr(*mqdp, &attr) == -1)
        errExit("mq_getattr");

    buffer = malloc(attr.mq_msgsize);
    if (buffer == NULL)
        errExit("malloc");

②   notifySetup(mqdp);

    while ((numRead = mq_receive(*mqdp, buffer, attr.mq_msgsize, NULL)) >= 0)
        printf("Read %ld bytes\n", (long) numRead);

    if (errno != EAGAIN)                        /* Unexpected error */
        errExit("mq_receive");

    free(buffer);
    pthread_exit(NULL);
}

static void
notifySetup(mqd_t *mqdp)
{
    struct sigevent sev;
```

```
③      sev.sigev_notify = SIGEV_THREAD;            /* Notify via thread */
       sev.sigev_notify_function = threadFunc;
       sev.sigev_notify_attributes = NULL;
               /* Could be pointer to pthread_attr_t structure */
④      sev.sigev_value.sival_ptr = mqdp;           /* Argument to threadFunc() */

       if (mq_notify(*mqdp, &sev) == -1)
           errExit("mq_notify");
   }

   int
   main(int argc, char *argv[])
   {
       mqd_t mqd;

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

⑤      mqd = mq_open(argv[1], O_RDONLY | O_NONBLOCK);
       if (mqd == (mqd_t) -1)
           errExit("mq_open");

⑥      notifySetup(&mqd);
       pause();                    /* Wait for notifications via thread function */
   }
```

pmsg/mq_notify_thread.c

Note the following further points regarding the design of the program in Listing 52-7:

- The program requests notification via a thread, by specifying `SIGEV_THREAD` in the *sigev_notify* field of the *sigevent* structure passed to *mq_notify()*. The thread's start function, *threadFunc()*, is specified in the *sigev_notify_function* field ③.
- After enabling message notification, the main program pauses indefinitely ⑥; timer notifications are delivered by invocations of *threadFunc()* in a separate thread ①.
- We could have made the message queue descriptor, *mqd*, visible in *threadFunc()* by making it a global variable. However, we adopted a different approach to illustrate the alternative: we place the address of the message queue descriptor in the *sigev_value.sival_ptr* field that is passed to *mq_notify()* ④. When *threadFunc()* is later invoked, this address is passed as its argument.

> We must assign a pointer to the message queue descriptor to *sigev_value.sival_ptr*, rather than (some cast version of) the descriptor itself because, other than the stipulation that it is not an array type, SUSv3 makes no guarantee about the nature or size of the type used to represent the *mqd_t* data type.

52.7 Linux-Specific Features

The Linux implementation of POSIX message queues provides a number of features that are unstandardized but nevertheless useful.

Displaying and deleting message queue objects via the command line

In Chapter 51, we mentioned that POSIX IPC objects are implemented as files in virtual file systems, and that these files can be listed and removed with *ls* and *rm*. In order to do this with POSIX message queues, we must mount the message queue file system using a command of the following form:

```
# mount -t mqueue source target
```

The *source* can be any name at all (specifying the string *none* is typical). Its only significance is that it appears in `/proc/mounts` and is displayed by the *mount* and *df* commands. The *target* is the mount point for the message queue file system.

The following shell session shows how to mount the message queue file system and display its contents. We begin by creating a mount point for the file system and mounting it:

```
$ su                                    Privilege is required for mount
Password:
# mkdir /dev/mqueue
# mount -t mqueue none /dev/mqueue
$ exit                                  Terminate root shell session
```

Next, we display the record in `/proc/mounts` for the new mount, and then display the permissions for the mount directory:

```
$ cat /proc/mounts | grep mqueue
none /dev/mqueue mqueue rw 0 0
$ ls -ld /dev/mqueue
drwxrwxrwt  2 root root 40 Jul 26 12:09 /dev/mqueue
```

One point to note from the output of the *ls* command is that the message queue file system is automatically mounted with the sticky bit set for the mount directory. (We see this from the fact that there is a *t* in the other-execute permission field displayed by *ls*.) This means that an unprivileged process can unlink only message queues that it owns.

Next, we create a message queue, use *ls* to show that it is visible in the file system, and then delete the message queue:

```
$ ./pmsg_create -c /newq
$ ls /dev/mqueue
newq
$ rm /dev/mqueue/newq
```

Obtaining information about a message queue

We can display the contents of the files in the message queue file system. Each of these virtual files contains information about the associated message queue:

```
$ ./pmsg_create -c /mq                  Create a queue
$ ./pmsg_send /mq abcdefg               Write 7 bytes to the queue
$ cat /dev/mqueue/mq
QSIZE:7       NOTIFY:0    SIGNO:0    NOTIFY_PID:0
```

The QSIZE field is a count of the total number of bytes of data in the queue. The remaining fields relate to message notification. If NOTIFY_PID is nonzero, then the process with the specified process ID has registered for message notification from this queue, and the remaining fields provide information about the kind of notification:

- NOTIFY is a value corresponding to one of the *sigev_notify* constants: 0 for SIGEV_SIGNAL, 1 for SIGEV_NONE, or 2 for SIGEV_THREAD.
- If the notification method is SIGEV_SIGNAL, the SIGNO field indicates which signal is delivered for message notification.

The following shell session illustrates the information that appears in these fields:

```
$ ./mq_notify_sig /mq &                 Notify using SIGUSR1 (signal 10 on x86)
[1] 18158
$ cat /dev/mqueue/mq
QSIZE:7       NOTIFY:0    SIGNO:10   NOTIFY_PID:18158
$ kill %1
[1]   Terminated    ./mq_notify_sig /mq
$ ./mq_notify_thread /mq &              Notify using a thread
[2] 18160
$ cat /dev/mqueue/mq
QSIZE:7       NOTIFY:2    SIGNO:0    NOTIFY_PID:18160
```

Using message queues with alternative I/O models

In the Linux implementation, a message queue descriptor is really a file descriptor. We can monitor this file descriptor using I/O multiplexing system calls (*select()* and *poll()*) or the *epoll* API. (See Chapter 63 for further details of these APIs.) This allows us to avoid the difficulty that we encounter with System V messages queues when trying to wait for input on both a message queue and a file descriptor (refer to Section 46.9). However, this feature is nonstandard; SUSv3 doesn't require that message queue descriptors are implemented as file descriptors.

52.8 Message Queue Limits

SUSv3 defines two limits for POSIX message queues:

MQ_PRIO_MAX
: We described this limit, which defines the maximum priority for a message, in Section 52.5.1.

MQ_OPEN_MAX
: An implementation can define this limit to indicate the maximum number of message queues that a process can hold open. SUSv3 requires this limit to be at least _POSIX_MQ_OPEN_MAX (8). Linux doesn't define this limit. Instead, because Linux implements message queue descriptors as file descriptors (Section 52.7), the applicable limits are those that apply to file descriptors. (In other words, on Linux, the per-process and system-wide limits on the number of file descriptors actually apply to the sum of file descriptors and message queue descriptors.) For further details on the applicable limits, see the discussion of the RLIMIT_NOFILE resource limit in Section 36.3.

As well as the above SUSv3-specified limits, Linux provides a number of /proc files for viewing and (with privilege) changing limits that control the use of POSIX message queues. The following three files reside in the directory /proc/sys/fs/mqueue:

msg_max
: This limit specifies a ceiling for the *mq_maxmsg* attribute of new message queues (i.e., a ceiling for *attr.mq_maxmsg* when creating a queue with *mq_open()*). The default value for this limit is 10. The minimum value is 1 (10 in kernels before Linux 2.6.28). The maximum value is defined by the kernel constant HARD_MSGMAX. The value for this constant is calculated as (131,072 / *sizeof(void *)*), which evaluates to 32,768 on Linux/x86-32. When a privileged process (CAP_SYS_RESOURCE) calls *mq_open()*, the msg_max limit is ignored, but HARD_MSGMAX still acts as a ceiling for *attr.mq_maxmsg*.

msgsize_max
: This limit specifies a ceiling for the *mq_msgsize* attribute of new message queues created by unprivileged processes (i.e., a ceiling for *attr.mq_msgsize* when creating a queue with *mq_open()*). The default value for this limit is 8192. The minimum value is 128 (8192 in kernels before Linux 2.6.28). The maximum value is 1,048,576 (INT_MAX in kernels before 2.6.28). This limit is ignored when a privileged process (CAP_SYS_RESOURCE) calls *mq_open()*.

queues_max
: This is a system-wide limit on the number of message queues that may be created. Once this limit is reached, only a privileged process (CAP_SYS_RESOURCE) can create new queues. The default value for this limit is 256. It can be changed to any value in the range 0 to INT_MAX.

Linux also provides the RLIMIT_MSGQUEUE resource limit, which can be used to place a ceiling on the amount of space that can be consumed by all of the message queues belonging to the real user ID of the calling process. See Section 36.3 for details.

52.9 Comparison of POSIX and System V Message Queues

Section 51.2 listed various advantages of the POSIX IPC interface over the System V IPC interface: the POSIX IPC interface is simpler and more consistent with the traditional UNIX file model, and POSIX IPC objects are reference counted, which simplifies the task of determining when to delete an object. These general advantages also apply to POSIX message queues.

POSIX message queues also have the following specific advantages over System V message queues:

- The message notification feature allows a (single) process to be asynchronously notified via a signal or the instantiation of a thread when a message arrives on a previously empty queue.
- On Linux (but not other UNIX implementations), POSIX message queues can be monitored using *poll()*, *select()*, and *epoll*. System V message queues don't provide this feature.

However, POSIX message queues also have some disadvantages compared to System V message queues:

- POSIX message queues are less portable. This problem applies even across Linux systems, since message queue support is available only since kernel 2.6.6.
- The facility to select System V messages by type provides slightly greater flexibility than the strict priority ordering of POSIX messages.

> There is a wide variation in the manner in which POSIX message queues are implemented on UNIX systems. Some systems provide implementations in user space, and on at least one such implementation (Solaris 10), the *mq_open()* manual page explicitly notes that the implementation can't be considered secure. On Linux, one of the motives for selecting a kernel implementation of message queues was that it was not deemed possible to provide a secure user-space implementation.

52.10 Summary

POSIX message queues allow processes to exchange data in the form of messages. Each message has an associated integer priority, and messages are queued (and thus received) in order of priority.

POSIX message queues have some advantages over System V message queues, notably that they are reference counted and that a process can be asynchronously notified of the arrival of a message on an empty queue. However, POSIX message queues are less portable than System V message queues.

Further information

[Stevens, 1999] provides an alternative presentation of POSIX message queues and shows a user-space implementation using memory-mapped files. POSIX message queues are also described in some detail in [Gallmeister, 1995].

52.11 Exercises

52-1. Modify the program in Listing 52-5 (`pmsg_receive.c`) to accept a timeout (a relative number of seconds) on the command line, and use *mq_timedreceive()* instead of *mq_receive()*.

52-2. Recode the sequence-number client-server application of Section 44.8 to use POSIX message queues.

52-3. Rewrite the file-server application of Section 46.8 to use POSIX message queues instead of System V message queues.

52-4. Write a simple chat program (similar to *talk(1)*, but without the *curses* interface) using POSIX messages queues.

52-5. Modify the program in Listing 52-6 (`mq_notify_sig.c`) to demonstrate that message notification established by *mq_notify()* occurs just once. This can be done by removing the *mq_notify()* call inside the `for` loop.

52-6. Replace the use of a signal handler in Listing 52-6 (`mq_notify_sig.c`) with the use of *sigwaitinfo()*. Upon return from *sigwaitinfo()*, display the values in the returned *siginfo_t* structure. How could the program obtain the message queue descriptor in the *siginfo_t* structure returned by *sigwaitinfo()*?

52-7. In Listing 52-7, could *buffer* be made a global variable and its memory allocated just once (in the main program)? Explain your answer.