Subsections

IPC:Message Queues:<sys/msg.h>

The basic idea of a ***message queue*** is a simple one.

Two (or more) processes can exchange information via access to a common system message queue. The ***sending*** process places via some (OS) message-passing module a message onto a queue which can be read by another process (Figure 24.1). Each message is given an identification or `type` so that processes can select the appropriate message. Process must share a common `key` in order to gain access to the queue in the first place (subject to other permissions -- see below).

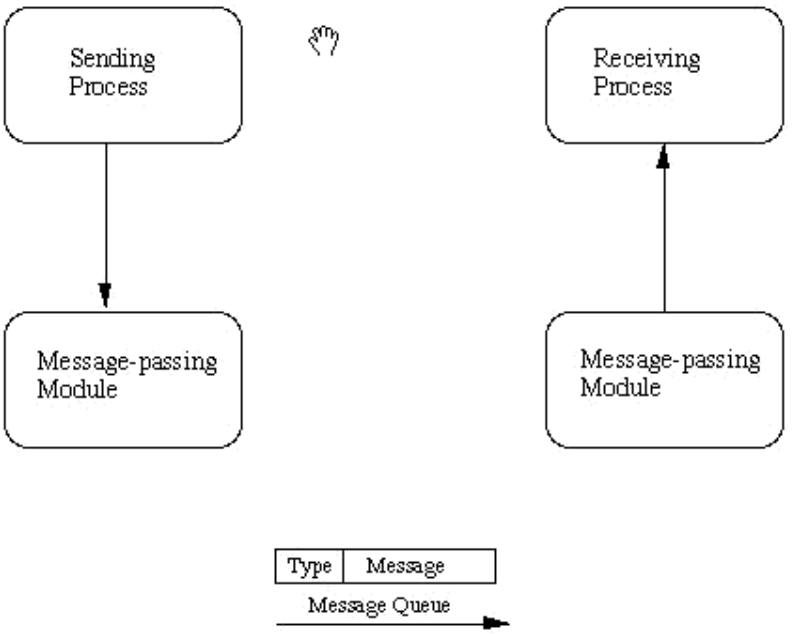

Fig. 24.1 Basic Message Passing IPC messaging lets processes send and receive messages, and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length. Messages can be assigned a specific type. Because of this, a server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Before a process can send or receive a message, the queue must be initialized (through the `msgget` function see below) Operations to send and receive messages are performed by the `msgsnd()` and `msgrcv()` functions, respectively.

When a message is sent, its text is copied to the message queue. The `msgsnd()` and `msgrcv()` functions can be performed as either blocking or non-blocking operations. Non-blocking operations allow for asynchronous message transfer -- the process is not suspended as a result of sending or receiving a message. In blocking or synchronous message passing the sending process cannot continue until the message has been transferred or has even been acknowledged by a receiver. IPC signal and other mechanisms can be employed to implement such transfer. A blocked message operation remains suspended until one of the following three conditions occurs:

- The call succeeds.
- The process receives a signal.
- The queue is removed.

Initialising the Message Queue

The `msgget()` function initializes a new message queue:

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

It can also return the message queue ID (`msqid`) of the queue corresponding to the key argument. The value passed as the `msgflg` argument must be an octal integer with settings for the queue's permissions and control flags.

The following code illustrates the `msgget()` function.

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

...


key_t key; /* key to be passed to msgget() */
int msgflg /* msgflg to be passed to msgget() */
int msqid; /* return value from msgget() */

...
key = ...
msgflg = ...

if ((msqid = msgget(key, msgflg)) == &ndash;1)
  {
    perror("msgget: msgget failed");
    exit(1);
   } else
    (void) fprintf(stderr, &ldquo;msgget succeeded");
...
```

IPC Functions, Key Arguments, and Creation Flags: <sys/ipc.h>

Processes requesting access to an IPC facility must be able to identify it. To do this, functions that initialize or provide access to an IPC facility use a `key_t key` argument. (`key_t` is essentially an `int` type defined in <sys/types.h>

The `key` is an arbitrary value or one that can be derived from a common seed at run time. One way is with `ftok()` , which converts a filename to a key value that is unique within the system. Functions that initialize or get access to messages (also semaphores or shared memory see later) return an ID number of type int. IPC functions that perform read, write, and control operations use this ID. If the key argument is specified as `IPC_PRIVATE`, the call initializes a new instance of an IPC facility that is private to the creating process. When the `IPC_CREAT` flag is supplied in the flags argument appropriate to the call, the function tries to create the facility if it does not

exist already. When called with both the `IPC_CREAT` and `IPC_EXCL` flags, the function fails if the facility already exists. This can be useful when more than one process might attempt to initialize the facility. One such case might involve several server processes having access to the same facility. If they all attempt to create the facility with `IPC_EXCL` in effect, only the first attempt succeeds. If neither of these flags is given and the facility already exists, the functions to get access simply return the ID of the facility. If `IPC_CREAT` is omitted and the facility is not already initialized, the calls fail. These control flags are combined, using logical (bitwise) OR, with the octal permission modes to form the flags argument. For example, the statement below initializes a new message queue if the queue does not exist.

```
msqid = msgget(ftok("/tmp",
key), (IPC_CREAT | IPC_EXCL | 0400));
```

The first argument evaluates to a `key` based on the string ("/tmp"). The second argument evaluates to the combined permissions and control flags.

Controlling message queues

The `msgctl()` function alters the permissions and other characteristics of a message queue. The owner or creator of a queue can change its ownership or permissions using `msgctl()` Also, any process with permission to do so can use `msgctl()` for control operations.

The `msgctl()` function is prototypes as follows:

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

The `msqid` argument must be the ID of an existing message queue. The `cmd argument` is one of:

IPC_STAT
-- Place information about the status of the queue in the data structure pointed to by `buf`. The process must have read permission for this call to succeed.

IPC_SET
-- Set the owner's user and group ID, the permissions, and the size (in number of bytes) of the message queue. A process must have the effective user ID of the owner, creator, or superuser for this call to succeed.

IPC_RMID
-- Remove the message queue specified by the `msqid` argument.

The following code illustrates the `msgctl()` function with all its various flags:

```
#include<sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
...
if (msgctl(msqid, IPC_STAT, &buf) == -1) {
perror("msgctl: msgctl failed");
exit(1);
}
...
if (msgctl(msqid, IPC_SET, &buf) == -1) {
perror("msgctl: msgctl failed");
exit(1);
}
...
```

Sending and Receiving Messages

The `msgsnd()` and `msgrcv()` functions send and receive messages, respectively:

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

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
          int msgflg);
```

The `msqid` argument **must** be the ID of an existing message queue. The `msgp` argument is a pointer to a structure that contains the type of the message and its text. The structure below is an example of what this user-defined buffer might look like:

```
 struct mymsg {
      long      mtype;    /* message type */
      char mtext[MSGSZ]; /* message text of length MSGSZ */
}
```

The `msgsz` argument specifies the length of the message in bytes.

The structure member `msgtype` is the received message's type as specified by the sending process.

The argument `msgflg` specifies the action to be taken if one or more of the following are true:

- The number of bytes already on the queue is equal to `msg_qbytes`.
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- If (`msgflg & IPC_NOWAIT`) is non-zero, the message will not be sent and the calling process will return immediately.
- If (`msgflg & IPC_NOWAIT`) is 0, the calling process will suspend execution until one of the following occurs:
  - The condition responsible for the suspension no longer exists, in which case the message is sent.
  - The message queue identifier `msqid` is removed from the system; when this occurs, `errno` is set equal to `EIDRM` and -1 is returned.
  - The calling process receives a signal that is to be caught; in this case the message is not sent and the calling process resumes execution.

  Upon successful completion, the following actions are taken with respect to the data structure associated with `msqid`:

  - `msg_qnum` is incremented by 1.
  - `msg_lspid` is set equal to the process ID of the calling process.
  - `msg_stime` is set equal to the current time.

The following code illustrates `msgsnd()` and `msgrcv()`:

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

...

int msgflg; /* message flags for the operation */
struct msgbuf *msgp; /* pointer to the message buffer */
int msgsz; /* message size */
long msgtyp; /* desired message type */
int msqid /* message queue ID to be used */

...

msgp = (struct msgbuf *)malloc((unsigned)(sizeof(struct msgbuf)
- sizeof msgp->mtext + maxmsgsz));

if (msgp == NULL) {
(void) fprintf(stderr, "msgop: %s %d byte messages.\n",
"could not allocate message buffer for", maxmsgsz);
```

```
exit(1);

...

msgsz = ...
msgflg = ...

if (msgsnd(msqid, msgp, msgsz, msgflg) == -1)
perror("msgop: msgsnd failed");
...
msgsz = ...
msgtyp = first_on_queue;
msgflg = ...
if (rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg) == -1)
perror("msgop: msgrcv failed");
...
```

POSIX Messages: <mqueue.h>

The POSIX message queue functions are:

`mq_open()` -- Connects to, and optionally creates, a named message queue.

`mq_close()` -- Ends the connection to an open message queue.

`mq_unlink()` -- Ends the connection to an open message queue and causes the queue to be removed when the last process closes it.

`mq_send()` -- Places a message in the queue.

`mq_receive()` -- Receives (removes) the oldest, highest priority message from the queue.

`mq_notify()` -- Notifies a process or thread that a message is available in the queue.

`mq_setattr()` -- Set or get message queue attributes.

The basic operation of these functions is as described above. For full function prototypes and further information see the UNIX `man pages`

Example: Sending messages between two processes

The following two programs should be compiled and run ***at the same time*** to illustrate basic principle of message passing:

`message_send.c`
-- Creates a message queue and sends one message to the queue.

`message_rec.c`
-- Reads the message from the queue.

`message_send.c` -- creating and sending to a simple message queue

The full code listing for `message_send.c` is as follows:

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

```
#include <string.h>

#define MSGSZ     128


/*
 * Declare the message structure.
 */

typedef struct msgbuf {
         long    mtype;
         char    mtext[MSGSZ];
         } message_buf;

main()
{
    int msqid;
    int msgflg = IPC_CREAT | 0666;
    key_t key;
    message_buf sbuf;
    size_t buf_length;

    /*
     * Get the message queue id for the
     * "name" 1234, which was created by
     * the server.
     */
    key = 1234;

(void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx,\
%#o)\n",
key, msgflg);

    if ((msqid = msgget(key, msgflg )) < 0) {
        perror("msgget");
        exit(1);
    }
    else
     (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);


    /*
     * We'll send message type 1
     */

    sbuf.mtype = 1;

    (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);

    (void) strcpy(sbuf.mtext, "Did you get this?");

    (void) fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);

    buf_length = strlen(sbuf.mtext) + 1 ;



    /*
     * Send a message.
     */
    if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0) {
       printf ("%d, %d, %s, %d\n", msqid, sbuf.mtype, sbuf.mtext, buf_length);
        perror("msgsnd");
        exit(1);
    }

   else
      printf("Message: \"%s\" Sent\n", sbuf.mtext);

    exit(0);
}
```

The essential points to note here are:

- The Message queue is created with a basic `key` and message flag `msgflg = IPC_CREAT | 0666` -- create queue and make it read and appendable by all.
- A message of type (`sbuf.mtype`) 1 is sent to the queue with the message ``Did you get this?"

`message_rec.c` -- receiving the above message

The full code listing for `message_send.c`'s companion process, `message_rec.c` is as follows:

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

#define MSGSZ     128


/*
 * Declare the message structure.
 */

typedef struct msgbuf {
    long    mtype;
    char    mtext[MSGSZ];
} message_buf;


main()
{
    int msqid;
    key_t key;
    message_buf  rbuf;

    /*
     * Get the message queue id for the
     * "name" 1234, which was created by
     * the server.
     */
    key = 1234;

    if ((msqid = msgget(key, 0666)) < 0) {
        perror("msgget");
        exit(1);
    }


    /*
     * Receive an answer of message type 1.
     */
    if (msgrcv(msqid, &rbuf, MSGSZ, 1, 0) < 0) {
        perror("msgrcv");
        exit(1);
    }

    /*
     * Print the answer.
     */
    printf("%s\n", rbuf.mtext);
    exit(0);
}
```

The essential points to note here are:

- The Message queue is opened with `msgget` (message flag `0666`) and the ***same*** `key` as `message_send.c`.
- A message of the ***same*** type 1 is received from the queue with the message ``Did you get this?" stored in `rbuf.mtext`.

Some further example message queue programs

The following suite of programs can be used to investigate interactively a variety of massage passing ideas (see exercises below).

The message queue **must** be initialised with the `msgget.c` program. The effects of controlling the queue and sending and receiving messages can be investigated with `msgctl.c` and `msgop.c` respectively.

`msgget.c`: Simple Program to illustrate `msget()`

```
/*
 * msgget.c: Illustrate the msgget() function.
 * This is a simple exerciser of the msgget() function. It prompts
 * for the arguments, makes the call, and reports the results.
 */

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

extern void  exit();
extern void  perror();

main()
{
 key_t  key;  /* key to be passed to msgget() */
 int  msgflg,  /* msgflg to be passed to msgget() */
   msqid;  /* return value from msgget() */

 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 (void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
 (void) fprintf(stderr, "Enter key: ");
 (void) scanf("%li", &key);
 (void) fprintf(stderr, "\nExpected flags for msgflg argument
are:\n");
 (void) fprintf(stderr, "\tIPC_EXCL =\t%#8.8o\n", IPC_EXCL);
 (void) fprintf(stderr, "\tIPC_CREAT =\t%#8.8o\n", IPC_CREAT);
 (void) fprintf(stderr, "\towner read =\t%#8.8o\n", 0400);
 (void) fprintf(stderr, "\towner write =\t%#8.8o\n", 0200);
 (void) fprintf(stderr, "\tgroup read =\t%#8.8o\n", 040);
 (void) fprintf(stderr, "\tgroup write =\t%#8.8o\n", 020);
 (void) fprintf(stderr, "\tother read =\t%#8.8o\n", 04);
 (void) fprintf(stderr, "\tother write =\t%#8.8o\n", 02);
 (void) fprintf(stderr, "Enter msgflg value: ");
 (void) scanf("%i", &msgflg);

 (void) fprintf(stderr, "\nmsgget: Calling msgget(%#lx,
%#o)\n",
 key, msgflg);
 if ((msqid = msgget(key, msgflg)) == -1)
 {
  perror("msgget: msgget failed");
  exit(1);
 } else {
  (void) fprintf(stderr,
   "msgget: msgget succeeded: msqid = %d\n", msqid);
  exit(0);
 }
}
```

`msgctl.c`Sample Program to Illustrate `msgctl()`

```
/*
 * msgctl.c:  Illustrate the msgctl() function.
 *
 * This is a simple exerciser of the msgctl() function.  It allows
 * you to perform one control operation on one message queue.  It
 * gives up immediately if any control operation fails, so be
careful
 * not to set permissions to preclude read permission; you won't
be
 * able to reset the permissions with this code if you do.
 */
#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/msg.h>
#include   <time.h>

static void do_msgctl();
extern void exit();
extern void perror();
static char warning_message[] = "If you remove read permission
for \
     yourself, this program will fail frequently!";

main()
{
 struct msqid_ds buf;     /* queue descriptor buffer for IPC_STAT
          and IP_SET commands */
 int    cmd,  /* command to be given to msgctl() */
     msqid;  /* queue ID to be given to msgctl() */

 (void fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");

 /* Get the msqid and cmd arguments for the msgctl() call. */
 (void) fprintf(stderr,
  "Please enter arguments for msgctls() as requested.");
 (void) fprintf(stderr, "\nEnter the msqid: ");
 (void) scanf("%i", &msqid);
 (void) fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
 (void) fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
 (void) fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
 (void) fprintf(stderr, "\nEnter the value for the command: ");
 (void) scanf("%i", &cmd);

 switch (cmd) {
  case IPC_SET:
   /* Modify settings in the message queue control structure.
*/
   (void) fprintf(stderr, "Before IPC_SET, get current
values:");
   /* fall through to IPC_STAT processing */
  case IPC_STAT:
   /* Get a copy of the current message queue control
    * structure and show it to the user. */
   do_msgctl(msqid, IPC_STAT, &buf);
   (void) fprintf(stderr, ]
   "msg_perm.uid = %d\n", buf.msg_perm.uid);
   (void) fprintf(stderr,
   "msg_perm.gid = %d\n", buf.msg_perm.gid);
   (void) fprintf(stderr,
   "msg_perm.cuid = %d\n", buf.msg_perm.cuid);
   (void) fprintf(stderr,
   "msg_perm.cgid = %d\n", buf.msg_perm.cgid);
   (void) fprintf(stderr, "msg_perm.mode = %#o, ",
   buf.msg_perm.mode);
   (void) fprintf(stderr, "access permissions = %#o\n",
```

```
   buf.msg_perm.mode & 0777);
   (void) fprintf(stderr, "msg_cbytes = %d\n",
       buf.msg_cbytes);
   (void) fprintf(stderr, "msg_qbytes = %d\n",
       buf.msg_qbytes);
   (void) fprintf(stderr, "msg_qnum = %d\n", buf.msg_qnum);
   (void) fprintf(stderr, "msg_lspid = %d\n",
       buf.msg_lspid);
   (void) fprintf(stderr, "msg_lrpid = %d\n",
       buf.msg_lrpid);
   (void) fprintf(stderr, "msg_stime = %s", buf.msg_stime ?
   ctime(&buf.msg_stime) : "Not Set\n");
   (void) fprintf(stderr, "msg_rtime = %s", buf.msg_rtime ?
   ctime(&buf.msg_rtime) : "Not Set\n");
   (void) fprintf(stderr, "msg_ctime = %s",
       ctime(&buf.msg_ctime));
   if (cmd == IPC_STAT)
    break;
   /*  Now continue with IPC_SET. */
   (void) fprintf(stderr, "Enter msg_perm.uid: ");
   (void) scanf ("%hi", &buf.msg_perm.uid);
   (void) fprintf(stderr, "Enter msg_perm.gid: ");
   (void) scanf("%hi", &buf.msg_perm.gid);
   (void) fprintf(stderr, "%s\n", warning_message);
   (void) fprintf(stderr, "Enter msg_perm.mode: ");
   (void) scanf("%hi", &buf.msg_perm.mode);
   (void) fprintf(stderr, "Enter msg_qbytes: ");
   (void) scanf("%hi", &buf.msg_qbytes);
   do_msgctl(msqid, IPC_SET, &buf);
   break;
  case IPC_RMID:
  default:
   /* Remove the message queue or try an unknown command. */
   do_msgctl(msqid, cmd, (struct msqid_ds *)NULL);
   break;
 }
 exit(0);
}

/*
 * Print indication of arguments being passed to msgctl(), call
 * msgctl(), and report the results. If msgctl() fails, do not
 * return; this example doesn't deal with errors, it just reports
 * them.
 */
static void
do_msgctl(msqid, cmd, buf)
struct msqid_ds    *buf;  /* pointer to queue descriptor buffer */
int    cmd,  /* command code */
    msqid;  /* queue ID */
{
 register int rtrn;  /* hold area for return value from msgctl()
*/

 (void) fprintf(stderr, "\nmsgctl: Calling msgctl(%d, %d,
%s)\n",
   msqid, cmd, buf ? "&buf" : "(struct msqid_ds *)NULL");
 rtrn = msgctl(msqid, cmd, buf);
 if (rtrn == -1) {
  perror("msgctl: msgctl failed");
  exit(1);
 } else {
  (void) fprintf(stderr, "msgctl: msgctl returned %d\n",
      rtrn);
 }
}
```

`msgop.c`: Sample Program to Illustrate `msgsnd()` and `msgrcv()`

```
/*
 * msgop.c: Illustrate the msgsnd() and msgrcv() functions.
```

```
 *
 * This is a simple exerciser of the message send and receive
 * routines. It allows the user to attempt to send and receive as
many
 * messages as wanted to or from one message queue.
 */

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

static int ask();
extern void exit();
extern char *malloc();
extern void perror();

char first_on_queue[] = "-> first message on queue",
 full_buf[] = "Message buffer overflow. Extra message text\
      discarded.";

main()
{
 register int    c;  /* message text input */
 int    choice; /* user's selected operation code */
 register int    i;  /* loop control for mtext */
 int    msgflg; /* message flags for the operation */
 struct msgbuf     *msgp;  /* pointer to the message buffer */
 int    msgsz;  /* message size */
 long    msgtyp;  /* desired message type */
 int    msqid,  /* message queue ID to be used */
     maxmsgsz, /* size of allocated message buffer */
     rtrn;  /* return value from msgrcv or msgsnd */
 (void) fprintf(stderr,
  "All numeric input is expected to follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 /* Get the message queue ID and set up the message buffer. */
 (void) fprintf(stderr, "Enter msqid: ");
 (void) scanf("%i", &msqid);
 /*
  * Note that <sys/msg.h> includes a definition of struct
msgbuf
  * with the mtext field defined as:
  *   char mtext[1];
  * therefore, this definition is only a template, not a
structure
  * definition that you can use directly, unless you want only
to
  * send and receive messages of 0 or 1 byte. To handle this,
  * malloc an area big enough to contain the template - the size
  * of the mtext template field + the size of the mtext field
  * wanted. Then you can use the pointer returned by malloc as a
  * struct msgbuf with an mtext field of the size you want. Note
  * also that sizeof msgp->mtext is valid even though msgp
isn't
  * pointing to anything yet. Sizeof doesn't dereference msgp,
but
  * uses its type to figure out what you are asking about.
  */
 (void) fprintf(stderr,
  "Enter the message buffer size you want:");
 (void) scanf("%i", &maxmsgsz);
 if (maxmsgsz < 0) {
  (void) fprintf(stderr, "msgop: %s\n",
    "The message buffer size must be >= 0.");
  exit(1);
 }
 msgp = (struct msgbuf *)malloc((unsigned)(sizeof(struct
msgbuf)
    - sizeof msgp->mtext + maxmsgsz));
 if (msgp == NULL) {
```

```
 (void) fprintf(stderr, "msgop: %s %d byte messages.\n",
   "could not allocate message buffer for", maxmsgsz);
 exit(1);
}
/* Loop through message operations until the user is ready to
 quit. */
while (choice = ask()) {
 switch (choice) {
 case 1: /* msgsnd() requested: Get the arguments, make the
   call, and report the results. */
  (void) fprintf(stderr, "Valid msgsnd message %s\n",
   "types are positive integers.");
  (void) fprintf(stderr, "Enter msgp->mtype: ");
  (void) scanf("%li", &msgp->mtype);
  if (maxmsgsz) {
   /* Since you've been using scanf, you need the loop
      below to throw away the rest of the input on the
      line after the entered mtype before you start
      reading the mtext. */
   while ((c = getchar()) != '\n' && c != EOF);
   (void) fprintf(stderr, "Enter a %s:\n",
       "one line message");
   for (i = 0; ((c = getchar()) != '\n'); i++) {
    if (i >= maxmsgsz) {
     (void) fprintf(stderr, "\n%s\n", full_buf);
     while ((c = getchar()) != '\n');
     break;
    }
    msgp->mtext[i] = c;
   }
   msgsz = i;
  } else
   msgsz = 0;
  (void) fprintf(stderr,"\nMeaningful msgsnd flag is:\n");
  (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#8.8o\n",
   IPC_NOWAIT);
  (void) fprintf(stderr, "Enter msgflg: ");
  (void) scanf("%i", &msgflg);
  (void) fprintf(stderr, "%s(%d, msgp, %d, %#o)\n",
   "msgop: Calling msgsnd", msqid, msgsz, msgflg);
  (void) fprintf(stderr, "msgp->mtype = %ld\n",
      msgp->mtype);
  (void) fprintf(stderr, "msgp->mtext = \"");
  for (i = 0; i < msgsz; i++)
   (void) fputc(msgp->mtext[i], stderr);
   (void) fprintf(stderr, "\"\n");
   rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
   if (rtrn == -1)
    perror("msgop: msgsnd failed");
   else
    (void) fprintf(stderr,
       "msgop: msgsnd returned %d\n", rtrn);
   break;
 case 2: /* msgrcv() requested: Get the arguments, make the
      call, and report the results. */
  for (msgsz = -1; msgsz < 0 || msgsz > maxmsgsz;
     (void) scanf("%i", &msgsz))
   (void) fprintf(stderr, "%s (0 <= msgsz <= %d): ",
       "Enter msgsz", maxmsgsz);
  (void) fprintf(stderr, "msgtyp meanings:\n");
  (void) fprintf(stderr, "\t 0 %s\n", first_on_queue);
  (void) fprintf(stderr, "\t>0 %s of given type\n",
   first_on_queue);
  (void) fprintf(stderr, "\t<0 %s with type <= |msgtyp|\n",
      first_on_queue);
  (void) fprintf(stderr, "Enter msgtyp: ");
  (void) scanf("%li", &msgtyp);
  (void) fprintf(stderr,
      "Meaningful msgrcv flags are:\n");
  (void) fprintf(stderr, "\tMSG_NOERROR =\t%#8.8o\n",
      MSG_NOERROR);
  (void) fprintf(stderr, "\tIPC_NOWAIT =\t%#8.8o\n",
      IPC_NOWAIT);
  (void) fprintf(stderr, "Enter msgflg: ");
```

```
   (void) scanf("%i", &msgflg);
   (void) fprintf(stderr, "%s(%d, msgp, %d, %ld, %#o);\n",
       "msgop: Calling msgrcv", msqid, msgsz,
       msgtyp, msgflg);
   rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);
   if (rtrn == -1)
    perror("msgop: msgrcv failed");
   else {
    (void) fprintf(stderr, "msgop: %s %d\n",
        "msgrcv returned", rtrn);
    (void) fprintf(stderr, "msgp->mtype = %ld\n",
        msgp->mtype);
    (void) fprintf(stderr, "msgp->mtext is: \"");
    for (i = 0; i < rtrn; i++)
     (void) fputc(msgp->mtext[i], stderr);
    (void) fprintf(stderr, "\"\n");
   }
   break;
  default:
   (void) fprintf(stderr, "msgop: operation unknown\n");
   break;
  }
 }
 exit(0);
}

/*
 * Ask the user what to do next. Return the user's choice code.
 * Don't return until the user selects a valid choice.
 */
static
ask()
{
 int response; /* User's response. */

 do {
  (void) fprintf(stderr, "Your options are:\n");
  (void) fprintf(stderr, "\tExit =\t0 or Control-D\n");
  (void) fprintf(stderr, "\tmsgsnd =\t1\n");
  (void) fprintf(stderr, "\tmsgrcv =\t2\n");
  (void) fprintf(stderr, "Enter your choice: ");

  /* Preset response so "^D" will be interpreted as exit. */
  response = 0;
  (void) scanf("%i", &response);
 } while (response < 0 || response > 2);

 return(response);
}
```

Exercises

Exercise 12755

Write a 2 programs that will both send and messages and construct the following dialog between them

- (Process 1) Sends the message "Are you hearing me?"
- (Process 2) Receives the message and replies "Loud and Clear".
- (Process 1) Receives the reply and then says "I can hear you too".

Exercise 12756

Compile the programs `msgget.c`, `msgctl.c` and `msgop.c` and then

- investigate and understand fully the operations of the flags (access, creation ***etc.*** permissions) you can set interactively in the programs.
- Use the programs to:

- Send and receive messages of two different message `type`s.
- Place several messages on the queue and inquire about the state of the queue with `msgctl.c`. Add/delete a few messages (using `msgop.c` and perform the inquiry once more.
- Use `msgctl.c` to alter a message on the queue.
- Use `msgctl.c` to delete a message from the queue.

Exercise 12757

Write a ***server*** program and two ***client*** programs so that the ***server*** can communicate privately to ***each client*** individually via a ***single*** message queue.

Exercise 12758

Implement a ***blocked*** or ***synchronous*** method of message passing using signal interrupts.

Dave Marshall
1/5/1999