```
    return 0;
}
```

10.3.5 msgrcv

The function prototype of the *msgrcv* API is:

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

int msgrcv ( int msgfd, const void* msgPtr, int len, int mtype, int flag );
```

This API receives a message of type *mtype* from a message queue designated by the *msgfd*. The message received is stored in the object pointed to by the *msgPtr* argument. The *len* argument specifies the maximum number of message text bytes that can be received by this call.

The *msgfd* value is obtained from a *msgget* function call.

The actual value of the *msgPtr* argument is the pointer to an object that has the *struct msgbuf*-like data structure.

The *mtype* value is the message type of the message to be received. The possible values and meaning of this argument are:

***mtype* value**	**Meaning**
0	Receive the oldest message of any type in the queue
Positive integer	Receive the oldest message of the specified message type
Negative integer	Receive a message whose message type is less than or equal to the absolute value of the *mtype*. If there is more than one message in the queue meeting this criteria, receive the one that is the oldest and has the smallest message type value.

The *flag* value may be 0, which means the process may be blocked if no messages in the queue match the selection criteria specified by *mtype*. Furthermore, if there is a message in the queue that satisfies the *mtype* selection criteria but is larger than *len*, the function returns a failure status.

If a process specifies IPC_NOWAIT in the *flag* value, the call will be nonblocking. Also, if the MSG_NOERROR flag is set in the *flag* value, a message in the queue is selectable (if larger than *len*). The function returns the first *len* byte of message text to the calling process and discards the rest of the data.

The *msgrcv* function returns the number of bytes written to the *mtext* buffer of the object pointed to by the *msgPtr* argument or -1 if it fails.

The following *test_msgrcv.C* program is the same one depicted in the last section, but after a message has been sent to the message queue, the process invokes the *msgrcv* API to wait and retrieve a message of type 20 from the queue. If the call succeeds, the process prints the retrieved message to the standard output; otherwise, it calls *perror* to print a diagnostic message.

```
#include <iostream.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/msg.h>

/* data structure of one message */
struct mbuf
{
      long    mtype;
      char    mtext[MSGMAX];
} mobj = { 15, "Hello" };

int main()
{
     int perm = S_IRUSRIS_IWUSRIS_IRGRPIS_IWOTH;
     int fd = msgget (100, IPC_CREATIIPC_EXCLIperm);

     if (fd==-1 || msgsnd(fd,&mobj,strlen(mobj.mtext)+1,IPC_NOWAIT))
          perror("message");
     else if (msgrcv(fd,&mobj,MSGMAX,20,IMSG_NOERROR) > 0)
          cout << mobj.mtext << endl;
     else perror("msgrcv");
     return 0;
}
```

10.3.6 msgctl

The function prototype of the *msgctl* API is:

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

int msgctl ( int msgfd, int cmd, struct msqid_ds* mbufPtr );
```

This API can be used to query the control data of a message queue designated by the *msgfd* argument, to change the information within the control data of the queue, or to delete the queue from the system.

The *msgfd* value is obtained from a *msgget* function call.

The possible values of *cmd* and their meanings are:

***cmd* value**	**Meaning**
IPC_STAT	Copy control data of the queue to the object pointed to by *mbufPtr*
IPC_SET	Change the control data of the queue by those specified in the object pointed to by *mbufPtr*. The calling process must be the superuser, creator, or the assigned owner of the queue to be able to perform this task. Furthermore, this API can only set the queue's owner user ID and group ID, access permissions and/or lower the *msg_qbyte* limit of the queue
IPC_RMID	Remove the queue from the system. The calling process must be the superuser, creator, or the assigned owner of the queue to be able to perform this task

This API returns 0 if it succeeds or -1 if it fails.

The following *test_msgctl.C* program "opens" a message queue with the key ID of 100 and calls *msgctl* to retrieve the control data of the queue. If both the *msgget* and *msgctl* calls succeed, the process prints to the standard output the number of messages currently in the queue,. It sets the queue owner user ID to be its own process ID via another *msgctl* call. Finally, the process invokes *msgctl* again to remove the queue.

```
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int main()
{
    struct msqid_ds mbuf;
    int fd = msgget (100, 0);
    if (fd>0 && msgctl(fd,IPC_STAT,&mbuf))    {
        cout << "#msg in queue: " << mbuf.msg_qnum << endl;
        mbuf.msg_perm.uid = getuid();  // change owner user ID
        if (msgctl(fd,IPC_SET,&mbuf)==-1)
            perror("msgctl");
    } else
        perror("msgctl");
    if (msgctl(fd,IPC_RMID,0)) perror("msgctl - IPC_RMID");
    return 0;
}
```

10.3.7 Client/Server Example

This section depicts a client/server application using messages. The first program *server.C* creates a daemon server process that runs in the background continuously to provide services to its client processes. The client processes are created by running the *client.C* program. The client processes post service requests to the daemon by sending messages to a message queue owned and managed by the daemon. The daemon responds to the client by sending messages to the same message queue.

Specifically, each service request message sent by a client process to the daemon server consists of:

Message data field	Meaning
message type	Integer service request command
message text	Client process ID in character string format (note that data stored in this field can be any arbitrary byte stream, including nonprintable characters)

The service request commands supported by the server are:

Service req. command	Service provided
1	Sends local date and time to client
2	Sends the Coordinated Universal Time (UTC) to client
3	Removes the message queue and terminates the daemon process. No response to client
4-99	Sends an error message back to client

Each response message sent by the server to a client consists of:

Message data field	Meaning
message type	Client process ID
message text	Service response data in character string format

Because both the server and its client interact through a common message queue, a message class can be defined to encapsulate all the message API interfacing from application programs. The following *message.h* header defines such a message class and is used by both the *server.C* and *client.C* programs:

```
/* The message.h header used by both the client.C and server.C */
#ifndef MESSAGE_H
#define MESSAGE_H
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/wait.h>

/* common declarations for daemon/server process */
enum { MSGKEY= 176, MAX_LEN =256, ILLEGAL_CMD = 4 };
enum { LOCAL_TIME = 1, UTC_TIME = 2, QUIT_CMD = 3};

typedef struct mgbuf
{
     long       mtype;
     char       mtext[MAX_LEN];
} MSGBUF;

class message
{
   private:
```

```
    int    msgId;                // message queue descriptor
    struct mgbuf mObj;
public:
    /* constructor. Get hold of a message queue */
    message ( int key )
    {
        if (msgId=msgget(key,IPC_CREAT|0666)==-1)
            perror("msgget");
    };

    /* destructor function. Do nothing */
     ~message () {};

    /* Check message queue open status */
    int good () { return (msgId >= 0) ? 1 : 0; };


    /* remove a message queue */
    int rmQ ()
    {
        int rc=msgctl(msgId,IPC_RMID,0);
        if (rc==-1) perror("msgctl");
        return rc;
    };

    /* send a message */
    int send ( const void* buf, int size, int type)
    {
        mObj.mtype = type;
        memcpy(mObj.mtext,buf,size);
        if (msgsnd(msgId,&mObj,size,0))       {
            perror("msgsnd"); return -1;
        };
        return 0;
    };

    /* receive a message */
    int rcv ( void* buf, int size, int type, int* rtype)
    {
        int len = msgrcv(msgId,&mObj,MAX_LEN,type,MSG_NOERROR);
        if (len==-1)    {
            perror("msgrcv"); return -1;
        }
        // Copy command or return data to buf and rtype
        memcpy(buf,mObj.mtext,len);
        if (rtype) *rtype = mObj.mtype;
```

```
        return len;
    };
};
#endif
```

The advantage of using the message class is that application programs do not need to use the underlying message APIs. They interface with a message object via a character buffer and a message type, using almost the same interface as do the *read* and *write* APIs for files. Thus, it reduces user programming effort. Furthermore, as will be seen in a later section, the message class implementation can be changed to use different IPC methods, with no changes required in the application code.

The following *server.C* program illustrates use of the *message.h* header:

```
#include <strstream.h>
#include "message.h"
#include <string.h>
#include <signal.h>

int main()
{
   int    len, pid, cmdId, mypid = getpid();
   char   buf[256];
   time_t tim;

   /* setup this process as a daemon */
   for (int i=0; i < 20; i++) sigset (i,SIG_IGN);
   setsid();

   cout << "server: start executing...\n" << flush;

   message mqueue(MSGKEY);                    // open a message queue
   if (!mqueue.good()) exit(1);               // quit if queue open fails

   /* wait for each request from a client */
   while (len=mqueue.rcv(buf,sizeof(buf),-99,&cmdId) > 0)
   {
        /* extract a client's PID and check the PID is valid */
        istrstream(buf,sizeof(buf)) >> pid;   // same as pid = atoi(buf);
        if (pid < 100)            // 0-100 are reserved for command IDs
        {
           cerr << "Illegal PID: " << buf << '/' << pid << endl;
           continue;
        }
```

```
            /* Prepare response to a client */
            cerr<<"server: receive cmd #" <<cmdId
                <<", from client: "<<pid<<endl;
            switch (cmdId)
            {
               case LOCAL_TIME:
                  tim = time(0);
                  strcpy(buf,ctime(&tim));
                  break;
               case UTC_TIME:
                  tim = time(0);
                  strcpy(buf,asctime(gmtime(&tim)));
                  break;
               case QUIT_CMD:
                  cerr << "server: deleting msg queue...\n";
                  return mqueue.rmQ();
               default:                              /* send an error msg back */
                  ostrstream(buf,sizeof(buf)) << "Illegal cmd: " << cmdId << '\0';
            }
            /* send response to a client */
            if (mqueue.send(buf,strlen(buf)+1,pid)==-1)
               cerr << "Server: " << mypid << " send response fails\n";
         }; /* loop forever */
         return 0;
      }
```

The server process starts by setting itself up as a daemon process: It ignores all major signals and makes itself a session and process group leader. The process is now independent from its parent or sibling process.

The server process creates a message object with the key ID of 176 (this is chosen arbitrarily). The message constructor function creates a message queue if it does not preexist, with the assigned key ID and an access permission of read-write for all.

Once a message object is created, the server goes into a polling loop, where it waits for client processes to send service request messages to the message object. Specifically, since client service request commands are restricted to the range of 1-99, the server will poll messages whose message types are anything less than 100. Once a service request is read, the server checks that the client process ID is greater than 99 and sends a response message back to the client based on the service request command. However, if the service command is QUIT_CMD, the server deletes the message object and terminates itself.

One precaution in designing a program that uses messages is that a process should rarely need to read messages sent by itself. Thus, it is important that the process use a different set of message types for the outgoing and incoming messages. In the current example, the

server response message types for clients are the clients' process IDs. These process IDs are always assumed to be larger than 100 (this is enforced in the *clent.C* program). Furthermore, the clients' service request message types are the service request commands (these commands are in the range of 1-99). Therefore, the server reads only clients' service request messages and not their response messages, and the clients only read the server's response messages and not its service request messages.

The *client.C* program is:

```
#include <strstream.h>
#include <string,h>
#include "message.h"

int main()
{
   int     cmdId, pid, mypid = getpid();
   while (getpid() < 100)
         switch (pid=fork())     {                    // make sure client PID > 99
            case -1: perror("fork"), exit(1);
            case 0:  break;
            default: waitpid(pid,0,0); exit(0);
         }
   cout << "client: start executing...\n";

   message mqueue(MSGKEY);                            // create a message object
   if (!mqueue.good()) exit(1);                       // quit if queue open fails
   char procId[256], buf2[256];
   ostrstream(procId,sizeof(procId)) << getpid() << '\0';
   do {
         /* Get a cmd from the standard input */
         cout << "cmd> " << flush;                    // print an input prompt
         cin >> cmdId;                                // get a command from a user
         cout << endl;                                // force a <CR> at console
         if (cin.eof()) break;                        // exit if EOF

         /* check cmd is valid */
         if (!cin.good() || cmdId < 0 || cmdId > 99)        {
            cerr << "Invalid input: " << cmdId << endl; continue;
         }
         /* send request to daemon */
         if (mqueue.send(procId,strlen(procId),cmdId))
               cout << "client: " << getpid() << " msgsnd error\n";

         else if (cmdId==QUIT_CMD) break;       /* exit on QUIT_CMD */
```

```
            /* receive data from daemon */
            else if (mqueue.rcv(buf2,sizeof(buf2),mypid,0)==-1)
               cout << "client: " <<mypid << " msgrcv error\n";

            /* print server's response data */
            else cout << "client: " << mypid << "  " << buf2 << endl;

        } while (1); /* loop until EOF */

        cout << "client: " << mypid << " exiting...\n" << flush;
        return 0;
    }
```

The client program starts by first making sure its process ID is greater than or equal to 100. If this is not the case, it calls *fork* recursively until one of its child process IDs is greater than or equal to 100. In this process, all nonqualified "parent" processes simply wait for their child process to terminate before they terminate themselves. This is a simple way of guaranteeing that the client/server interaction condition (namely, that the client process ID be greater than 99) is met.

Once a client process is created, it "opens" a message object with the same key ID as the server. It then enters a loop, where it iteratively prompts users to enter a service request command from the standard input. For each command it receives, it checks whether the command is in the range of 1-99 then sends a service request message to the server process. After that, it reads the service response back from the server and prints the corresponding data to the standard output.

The client process terminates when EOF is encountered in the standard input (e.g., user presses <ctrl-D>) or an error is encountered in reading the standard input.

Some sample interaction of these client and server programs are:

```
chp13 % server &
server: start executing...
[1] 356
chp13 % client
client. start executing...
cmd> 1
server: receive cmd #1, from client: 357
client: 357  Tue Jan 24 22:23:17 1995
cmd> 2
server: receive cmd #2, from client: 357
```

```
client: 357  Wed Jan 25 06:23:19 1995
cmd> 4
server: receive cmd #4, from client: 357
client: 357  Illegal cmd: 4
cmd> 3
client: 357 exiting...
server: receive cmd #3, from client: 357
server: deleting msg queue...
[1]   Done                    mserver
chp13 %
```

Although the above example shows only one client interacting with a server, there can be multiple client processes running simultaneously and interacting with the same server.

10.4 POSIX.1b Messages

POSIX.1b messages are created and manipulated in a manner similar to UNIX System V messages. Specifically, the POSIX.1b defines the <mqueue.h> header and a set of messages APIs:

```
#include <mqueue.h>

mqd_t    mq_open( char* name, int flags, mode_t mode,
                  struct mq_attr* attrp );

int      mq_send ( mqd_t mqid, const char* msg, size_t len,
                  unsigned priority );

int      mq_receive (mqd_t mqid, char* buf, size  t len, unsigned* prio );

int      mq_close ( mqd_t mqid );

int      mq_notify ( mqd_t mqid, const struct mq_sigevent* sigvp );

int      mq_getattr ( mqd_t mqid, struct mq_attr* atttrp );

int      mq_setattr ( mqd_t mqid, struct mq_attr* atttrp,
                  struct mq_attr* oattrp );
```

The *mq_open* API is like the *msgget* function: It returns a handle of type *mqd_t*, which designates a message queue. Note that *mqd_t* is not an integer descriptor.

The message queue "opened" by the *mq_open* function is given a name as specified in the *name* argument. The *name* value should be a UNIX path name-like character string and should always begin with the "/" character. It is implementation-dependent of whether additional "/" characters are allowed in the *name* value. Furthermore, users should not expect that a file with the same name is created by this call.

The *flags* argument specifies the access manner of the queue by a calling process. Its values may be O_RDONLY (the calling process can only receive messages), O_WRONLY (the calling process may only send messages), or O_RDWR (the calling process may send and receive messages). Furthermore, the O_CREAT flag may also be specified to indicate that if the named queue does not exist, it should be created. Moreover, if the O_EXCL flag is specified with the O_CREAT flag, it forces the function to abort if the named queue already exists.

Finally, the O_NONBLOCK flag may also be specified in an *oflag* value to indicate that future access of the message queue (via *mq_send* and *mq_receive* APIs) should be nonblocking.

The *mode* and *attrp* arguments are needed only if the O_CREAT flag is specified in the *oflag* argument. They specify read-write access permission and special attributes for a message queue created by this call. The *struct mq_attr* data type is defined in the <mqeue.h> header.

The function returns a *mqd_t*-type message queue handle if it succeeds or a (*mqd_t*)-1 value if it fails.

For example, the following opens and creates, if necessary, a message queue */foo* for read-write access. The access permission assigned to a newly create queue are read-write for the owner only. Furthermore, the new message queue may hold up to 200 messages, each message not to exceed 1024 bytes:

```
struct mq_attr attrv;          // contains attributes for a new queue
attrv.mq_maxmsg = 200;         // at most 200 msg may be in a queue
attrv.mq_msgsize = 1024;       // at most 1024 bytes per message
attrv.mq_flags = 0;

mqd_t mqid = mq_open( "/foo", O_RDWRIO_CREAT, S_IRWXU, &attrv);
if (mqid == (mqd_t)-1) perror("mq_open");
```

The *mq_send* API sends a message to a message queue referenced by the *mqid* argument. The *msg* argument value is the address of a buffer that contains a message text, and the *len* argument specifies the message text size in number of characters. The *len* value should be less than the message queue limit (maximum size per message); otherwise, the call will fail.

The *priority* argument value is an integer between 0 and MQ_PRIO_MAX. It is used to sort messages in a queue-messages with higher *priority* values are accessed earlier than those with lower *priority* values. Furthermore, if two or more messages have the same *priority* value in a queue, they are sorted in decreasing order, based on their duration in the queue. The older messages are retrieved before the newer ones.

The function returns 0 if it succeeds and -1 if it fails. Note that the function may block a calling process if the message queue is already full. However, if the queue was opened with the O_NONBLOCK flag, the function aborts and returns immediately with a -1 failure status.

The following example sends a message *Hello POSIX.1b* to a message queue, designated by *mqid*, with a priority value of 5:

```
char* msg = "Hello POSIX.1b";
if (mq_send( mqid, msg, strlen(msg)+1, 5)) perror("mq_send");
```

The *mq_receive* API receives the oldest and highest priority message from a message queue referenced by the *mqid* argument. The *buf* argument value is the address of a buffer containing the message text, and the *len* argument specifies the maximum size of the *buf* argument. If a message to be received is larger than *len* bytes, the function will return a failure status.

The *priop* argument is the address of an unsigned integer variable holding the priority value of the receiving message. If the argument value is given as NULL, the receiving message's priority value is a don't-care.

The function returns the number of message text bytes that has been put in the *buf* argument if it succeeds or -1 if it fails. Note that the function may block a calling process if the message queue is empty. However, if the queue was opened with the O_NONBLOCK flag, the function aborts and returns immediately with a -1 failure status.

If multiple processes are blocked on the *mq_receieve* call, then when a message arrives in the queue, the process with the highest priority and the longest waiting time gets the message.

The following example receives a message from a message queue designated by *mqid*:

```
char buf[256];
unsigned prio;
if (mq_receieve( mqid, buf , sizeof buf, &prio)==-1) perror("mq_receieve");
cerr << "receive msg: '" << buf << "', priority=" << prio << endl;
```

The *mq_close* API deallocates resources used to associate a message queue with a message handle, *mqid*. This function returns a 0 if succeeds or -1 if it fails.

The *mq_notify* API is used if a process wishes to receive asynchronous notification of a message's arrival at an empty queue, instead of being blocked by a *mq_receieve* call to wait for such an event. The *mqid* argument designates a message queue to monitor message arrival; the *sigvp* argument value is the address of a *struct sigevent*-type variable. The *struct sigevent* data type is defined in the <signal.h> header and contains the signal number which should be generated by the calling process when a message arrives at the designated message queue.

This function fails if there is a process that has already registered signal notification for the same message queue or is blocked by a *mq_receieve* call. Furthermore, even if a process succeeds with the *mq_notify* call and a notification signal is delivered (because a message has arrived at the message queue), it may still be unable to receive the message if another process issues a *mq_receieve* call before it does.

Finally, if the *sigvp* argument value is specified as NULL, the call unregisters signal notification for a specified message queue. The signal notification is also unregistered if the process already exists or calls the *mq_close* API on the message queue handle.

This function returns 0 if it succeeds or -1 if it fails.

The following example registers a SIGUSR1 signal to be delivered to the calling process if any message arrives at the message queue designated by the *mqid* variable:

```
struct sigevent sigv;
sigv.sigev_notify = SIGEV_SIGNAL;      // signal notification is requested
sigv.sigev_signo = SIGUSR1;            // send SIGUSR1 for notification
if (mq_notify(mqid, &sigv)==-1) perror("mq_notify");
```

The *mq_getattr* API queries attributes of a message queue designated by the *mqid* argument. The *attrp* is the address of a *struct mq_attr* type variable. The *struct mq_attr* data type is defined in the <mqeue.h> header. Some of its useful member fields and their meanings are:

Member Field	Meaning
mq_flags	Specifies whether the queue operation is blocking. Possible value is either 0 or O_NONBLOCK
mq_maxmsg	Maximum number of messages allowed in the queue at any one time
mq_msgsize	Maximum size, in bytes, allowed per message
mq_curmsgs	Number of messages currently in the queue

This function returns 0 if it succeeds or -1 if it fails.

The following example obtains attribute information for a message queue designated by the *mqid* variable:

```
struct mq_attr attrv;
if (mq_getattr(mqid, &attrv)==-1)
      perror("mq_getattr");
else  cout << "flags = " << attrv.mq_flags
            << ", cur no. msg: " << attrv.mq_curmsgs << endl;
```

The *mq_setattr* API sets the *mq_attr::mq_flags* attribute of a message queue designated by the *mqid* argument. The *attrp* is the address of a *struct mq_attr* type variable. This is the input argument to the function, and only the *attrp->mq_flags* value is used by the function. The legal value of this member field is either 0 (use blocking operation on the queue) or O_NONBLOCK (use nonblocking operation on the queue). The *oattrp* argument value, if specified as the address of a *struct mq_attr*-typed variable, returns the same information as the *mq_getattr* call prior to the *mq_setattr* call.

This function returns 0 if it succeeds or -1 if it fails.

The following example sets a message queue referenced by the *mqid* variable to use nonblocking operations. The old message queue attributes are ignored.

```
struct mq_attr attrv;
attrv.mq_flags = O_NONBLOCK;
if (mq_setattr(mqid, &attrv,0)==-1) perror("mq_setattr");
```

10.4.1 POSIX.1b Message Class

The message class defined in Section 10.3.7 is for UNIX System V messages only. The following *message2.h* header defines a new message class that uses POSIX.1b message APIs. Note that the new message class interface is the same as that in Section 10.3.7:

```
#ifndef MESSAGE2_H
#define MESSAGE2_H
#include <stdio.h>
#include <memory.h>
#include <sys/ipc.h>
#include <mqueue.h>                    // use POSIX.1b messages APIs

/* common declarations for daemon/server process
enum { MSGKEY=186, MAX_LEN=256, ILLEGAL_CMD = 4};
enum { LOCAL_TIME = 1, UTC_TIME = 2, QUIT_CMD = 3 };
struct mgbuf
{
        long        mtype;
        char        mtext[MAX_LEN];
};

/* POSIX.1b message class */
class message
{
   private:
        mqd_t   msgId;                          // message queue handle
        struct mgbuf mObj;
   public:
        /* System V compatible constructor function */
        message( int key )
        {
           char name[80];
           sprintf(name,"/MQUEUE%d",key);
           if ((msgId=mq_open(name,O_RDWRIO_CREAT,0666,0))
                                == (mqd_r)-1)
              perror("mq_open");
        };

        /* POSIX.1b style constructor function */
        message( const char* name )
        {
           if (msgId=mq_open(name,O_RDWRIO_CREAT,0666,0))
                                == (mqd_r)-1)
              perror("mq_open");
        };

        /* destructor function */
        ~message() {  (void)mq_close( msgId); };

        /* check queue open status */
        int good() { return (msgId >= 0) ? 1 : 0; };
```

```
        /* remove message queue */
        int rmQ()
        {
           return mq_close( msgId );
        };

        /* send a message */
        int send( const void* buf, int size, int type)
        {
           mObj.mtype = type;
           memcpy(mObj.mtext,buf,size);
           if (mq_send(msgId,(char*)&mObj,size,type)) {
              perror("mq_send");
              return -1;
           };
           return 0;
        };

        /* receive a message */
        int rcv( void* buf, int size, int type, unsigned* rtype)
        {
           struct mq_attr attrv;
           if (mq_getattr(msgId,&attrv)==-1)  {
              perror("mq_getattr");
              return -1;
           }
           if (!attrv.mq_curmsgs) return -1;     // no messages

           int len = mq_receive(msgId,(char*)&mObj,MAX_LEN,rtype);
           if (len < 0) {
              perror("mq_receieve");
              return -2;
           }

           if (type && ((type > 0 && type!=*rtype) ||
                        (type < 0 && -type < *rtype)) )
              mq_send(msgId, (char*)&mObj, len, *rtype );
              return -3;                          // not the requested type
           }
           memcpy(buf,mObj.mtext,len);
           return len;
        };
};
#endif                  /* MESSAGE2_H */
```

The POSIX.1b version of the message class, as shown above, can be used in the same manner as the System V version. Specifically, there are two *message::message* constructor functions, one is invoked with an integer key and the other with a message name. In either way, the *mq_open* function is called to open and create, if necessary, a message queue.

The *message::send, message::rcv* and *message::~message* functions have exactly the same interface as that of the System V message class. The only difference is that the POSIX.1b *message::rcv* function cannot really select messages based on a user's specified message type. Thus, the function gets the highest priority message from a queue. If the message type does not satisfy the user-defined message type, the function pushes the message back to the queue and returns a -1 failure status. A message type satisfies a user-defined type if: (1) the user-defined type is 0 (message type is don't-care); (2) the user-specified type matches the received message type exactly; or (3) the user-specified type is negative and the absolute value of that type is greater than or equal to the receiving message type.

The new *message2.h* header can be used in the same client and server programs as depicted in Section 13.3.7. The output of the newly compiled client and server programs should be identical to that of the System V version.

10.5 UNIX System V Semaphores

Semaphores provide a method to synchronize the execution of multiple processes. Semaphores are allocated in sets of one or more. A process can also use multiple semaphore sets. Semaphores are frequently used along with shared memory to establish an elaborate method for interprocess communication.

The UNIX System V semaphore APIs provide the following functions:

- Create a semaphore set
- "Open" a semaphore set and get a descriptor to reference the set
- Increase or decrease the integer values of one more semaphores in a set
- Query the values of one or more semaphores in a set
- Query or set control data of a semaphore set

Each semaphore has an unsigned short value. A process that has read permission for semaphores may query their values. A process that has write permission for semaphores can increase or decrease their values. If a process attempts to decrease a semaphore value such that the resultant value becomes negative, the operation, as well as the process, will be blocked until another process increases the semaphore's value to a number large enough that the blocked process's operation can succeed (i.e., the resultant semaphore value is 0 or a positive number). This forms the basis for multiprocess synchronization using semaphores:

- A process *X* that wishes to wait for another process *Y* decreases the value of one or more semaphores by some value such that it is blocked by the kernel
- When process *Y* is ready to let *X* resume execution, it increases the semaphores' values enough for *X*'s semaphore operation to succeed. The kernel unblocks *X*

If a semaphore's value is a positive integer but a process explicitly queries its value as to whether it is zero, the process will be blocked until the semaphore's value is decreased to zero by another process.

If a process has access to a set of semaphores, it can perform operations on individual semaphores in the set or operate on two or more semaphores in the set simultaneously. In the latter case, if an operation cannot be performed on any of the selected semaphores, the entire operation fails, and the values of the semaphore set are unchanged. Thus, the semaphore operations are atomic at the set level. This is to ensure that when two or more processes attempt to read and write values on the same semaphore set, only one can perform operations at a given time.

10.5.1 UNIX Kernel Support for Semaphores

In UNIX System V.3 and V.4, there is a semaphore table in the kernel address space that keeps track of all semaphore sets created in the system. Each entry of the semaphore table stores the following data for one semaphore set:

- A name that is an integer ID key assigned by the process which created the set. Other processes may specify this key to "open" the set and get a descriptor for future access of the set
- The creator user ID and group ID. A process whose effective user ID matches a semaphore set creator user ID may delete the set and also change control data of the set
- The assigned owner user ID and group ID. These are normally the same as the creator user and group IDs, but a creator process can set these values to assign a different owner and group membership for the set
- Read-write access permission of the set for owner, group members, and others. A process that has read permission to the set may query values of the semaphores and queries the assigned users and group IDs of the set. A process that has write permission to a set may change the values of semaphores
- The number of semaphores in the set
- The time when the last process changed one or more semaphore values
- The time when the last process changed the control data of the set
- A pointer to an array of semaphores

Semaphores in a set are referenced by array indices, such that the first semaphore in the set has an index of zero; the second semaphore has an index of 1; and so on. Furthermore, each semaphore stores the following data:

- The semaphore's value
- The process ID of the last process that operated on the semaphore
- The number of processes that are currently blocked pending the increase of semaphore value
- The number of processes which are currently blocked pending the semaphor's value becoming zero

Figure 10.3 depicts the UNIX kernel data structure for semaphores.

Like messages, semaphores are stored in a kernel address space and are persistent, despite their creator process's termination. Furthermore, each semaphore set has an assigned owner, and only processes that have superuser, set creator, or assigned owner privileges may delete the set or change its control data. If a semaphore set is deleted, any processes that are blocked at that time due to the semaphores are awakened by the kernel - the system calls they invoked are aborted and return a -1 failure status.

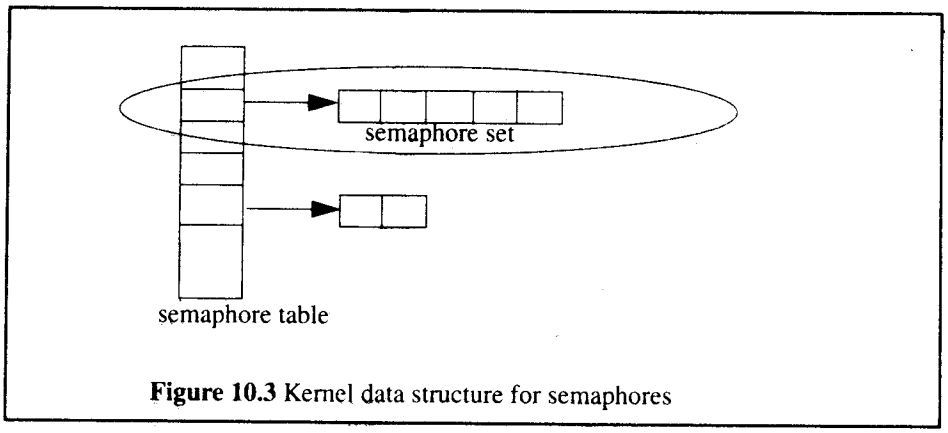

Figure 10.3 Kernel data structure for semaphores

Finally, there are several system-imposed limits on the manipulation of semaphores. These limits are defined in the <sys/sem.h> header:

System limit	Meaning
SEMMNI	The maximum number of semaphore sets that may exist at any given time in a system
SEMMNS	The maximum number of semaphores in all sets that may exist in a system at any one time

System limit	Meaning
SEMMSL	The maximum number of semaphores allowed per set
SEMOPM	The maximum number of semaphores in a set that may be operated on at any one time

The effects of these system-imposed limits on processes are:

- If a process attempts to create a new semaphore set that causes either the SEMMNI or the SEMMNS limit to be exceeded, the process will be blocked until one or more existing sets are deleted by a process.
- If a process attempts to create a semaphore set with more than SEMMSL semaphores, the system call fails.
- If a process attempts to operate on more than SEMOPM semaphores in a set in one operation, the system call fails.

10.5.2 The UNIX APIs for Semaphores

The <sys/ipc.h> header defines a *struct ipc_perm* data type, which stores the user ID, group ID, creator user ID and group ID, assigned name key, and the read-write permission of a semaphore set.

The semaphore table entry data type is *struct semid_ds*, which is defined in the <sys/sem.h> header. The data fields of the structure and the data stored are:

Data field	Data Stored
sem_perm	Data stored in a *struct ipc_perm* record
sem_nsems	Number of semaphores in the set
sem_base	Pointer to an array of semaphores
sem_otime	Time when last process operated on semaphores
sem_ctime	Time when last process changed control data of the set

In addition to the above, the *struct sem* data type, as defined in the <sys/sem.h> header, defines the data stored in a semaphore:

Data field	Data Stored
semval	Current semaphore's integer value
sempid	Process ID of the last process that operated on the semaphore
semncnt	Number of processes that are blocked waiting for

	the semaphore's value to be increased
semzcnt	Number of processes that are blocked waiting for the semaphore's value to become zero

Figure 10-4 illustrates uses of the aforementioned structures in the semaphore table and semaphore records.

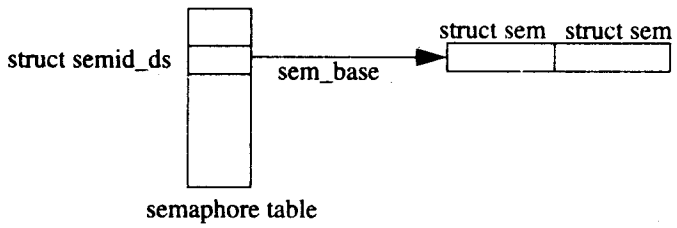

Figure 10-4. Data types of the semaphore table and semaphore record

The UNIX System V semaphore APIs are:

Semaphores API	Usages
semget	Open and create, if needed, a semaphore set
semop	Change or query semaphore value
semctl	Query or change control data of a semaphore set or delete a set

The header files needed for the semaphore APIs are:

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

10.5.3 semget

The function prototype of the *semget* API is:

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

int semget ( key_t key, int num_sem, int flag );
```

This function "opens" a semaphore set whose key ID is given in the *key* argument. The function returns a nonnegative integer descriptor that can be used in the other semaphore APIs to change or query semaphore value and to query and/or set control data for the semaphore set.

If the value of the *key* argument is a positive integer, the API attempts to open a semaphore set whose key ID matches that value. However, if the *key* value is the manifested constant IPC_PRIVATE, the API allocates a new semaphore set to be used exclusively by the calling process. Specifically, the "private" semaphores are usually allocated by a parent process which then forks one or more child processes. The parent and child processes then use the semaphores to synchronize their operations.

If the *flag* argument is 0; the API aborts when there is no semaphore set whose key ID matches the given *key* value. Otherwise, it returns a descriptor for that set. If a process wishes to create a new set with the given *key* ID (if none preexists), then the *flag* value should be the bitwise-OR of the manifested constant IPC_CREAT and the read-write access permissions for the new set.

The *num_sem* value may be 0 if the IPC_CREAT flag is not specified in the *flag* argument, or it is the number of semaphores to be allocated when a new set is to be created.

For example, the following system call creates a two-element semaphore set with the key ID of 15 and access permission of read-write for owner and read-only for group members and others (if such a set does not preexist). The call returns an integer descriptor for future references of the queue:

```
int perms = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
int semfdesc = semget ( 15, 2, IPC_CREAT | perms);
```

If a process wishes to guarantee creation of a new semaphore set, it can specify the IPC_EXCL flag with the IPC_CREAT flag, and the API will succeed only if it creates a new set with the given *key*.

The API returns a -1 value it if fails.

10.5.4 semop

The function prototype of the *semop* API is:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop ( int semfd, struct sembuf* opPtr, int len );
```

This API may be used to change the value of one or more semaphores in a set (as designated by *semfd)* and/or to test whether their values are 0. The *opPtr* is the pointer to any array of *struct sembuf* objects, each of which specifies one operation (query or change value) for a semaphore. The *len* argument specifies how many entries are in the array pointed to by *opPtr.*

The *struct sembuf* data type is defined in the <sys/sem.h> header as:

```
struct sembuf
{
    short   sem_num;        // semaphore index
    short   sem_op;         // semaphore operation
    short   sem_flg;        // operation flag(s)
};
```

The possible values of *sem_op* and their meanings are:

sem_op value	Meaning
a positive number	Increase the indexed semaphore value by this amount
a negative number	Decrease the indexed semaphore value by this amount
a zero	Test whether the semaphore value is 0

If a *semop* call attempts to decrease a semaphore's value to a negative number, or if it tests a semaphore's value as 0 but it is not, then the calling process will be blocked by the kernel. This will occur unless the IPC_NOWAIT flag is specified in the *sem_flg* fields of the array entries where *sem_op* is a negative number or zero.

Another flag that may be specified in the *sem_flg* fields of the *struct sembuf* objects is SEM_UNDO. This instructs the kernel to keep track of the net semaphore value change on the indexed semaphore (due to the *semop* call). When the calling process terminates, the kernel will reverse these changes so that any other processes awaiting such changes will not be locked out indefinitely. This would occur because the exiting process forgot to undo the changes it made to the semaphore set.

The API returns a 0 if it succeeds or -1 if it fails.

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

/* decrease 1st semaphore value by 1, test 2nd semaphore value is zero */
struct semid_ds sbuf[2] = {{0, -1, SEM_UNDO|IPC_NOWAIT}, {1, 0, 0} };

int main()
{
   int perms = S_IRWXU | S_IRWXG | S_IRWXO;
   int fd = semget (100, 2, IPC_CREAT | perms);
   if (fd==-1) perror("semget"), exit(1);
   if (semop(fd,sbuf,2)==-1) perror("semop");
   return 0;
}
```

This example opens a two-element semaphore set with the key ID of 100, and it creates the set with read-write permission for all (if it does not preexist).

If the *semget* call succeeds, the process calls *semop* to decrease the first semaphore value by 1 and tests the second semaphore value as zero. Furthermore, it specifies the IPC_NOWAIT and SEM_UNDO flags when it operates on the first semaphore.

10.5.5 semctl

The function prototype of the *semctl* API is:

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

int semctl ( int semfd, int num, int cmd, union semun arg );
```

This API can be used to query or change the control data of a semaphore set designated by the *semfd* argument or to delete the set altogether.

The *semfd* value is the semaphore set descriptor, as obtained from a *semget* function call.

The *num* value is a semaphore index where the next argument, *cmd*, specifies an operation to be performed on a specific semaphore within a set.

The *arg* argument is a union-typed object that may be used to specify or retrieve the control data of one or more semaphores in the set, as determined by the *cmd* argument. The *union semun* data type is defined in the <sys/sem.h> header as:

```
union semun
{
    int             val;    // a semaphore value
    struct semid_ds *buf;   // control data of a semaphore set
    ushort          *array; // an array of semaphore values
};
```

The possible values of *cmd* and their meanings are:

***cmd* value**	**Meaning**
IPC_STAT	Copy control data of the semaphore to the object pointed to by *arg.buf*. The calling process must have read permission to the set
IPC_SET	Change the control data of the semaphore set by those data specified in the object pointed to by *arg.buf*. The calling process must be the supervisor, creator, or the assigned owner of the set to be able to perform this task. Furthermore, this API can establish only the set owner user and group IDs and access permission of the queue
IPC_RMID	Remove the semaphore from the system. The calling process must be the superuser, creator, or the assigned owner of the queue to be able to perform this task
GETALL	Copy all the semaphore values to the array pointed to by *arg.array*
SETALL	Set all the semaphore values by the corresponding values contained in an array pointed to by *arg.array*
GETVAL	Return the *num*-indexed semaphore value. *arg* is unused

cmd value	Meaning
SETVAL	Set the *num*-indexed semaphore value by the value specified in *arg.val*
GETPID	Return the process ID of the last process that operated on the *num*-indexed semaphore. *arg* is unused
GETNCNT	Return the number of processes that are currently blocked waiting for the *num*-indexed semaphore value to increase. *arg* is unused
GETZCNT	Return the number of processes that are currently blocked waiting for the *num*-indexed semaphore value to become zero. *arg* is unused

This API returns a *cmd*-specific value if it succeeds or -1 if it fails.

The following *test_sem.C* program "opens" a semaphore set with the key ID of 100 and calls *semctl* to retrieve the control data of the set. If both the *semget* and *semctl* calls are successful, the process prints to the standard output the number of semaphores in the set. It then sets the set owner user ID as its own process ID, via another *semctl* call. Finally, the process invokes *semctl* again to remove the set.

```
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/sem.h>

union semun {
        int    val;
        struct semid_ds *mbuf;
        ushort *array;
} arg;

int main()
{
   struct semid_ds mbuf;
   arg.mbuf = &mbuf;
   int fd = semget (100, 0, 0);
   if (fd>0 && semctl(fd,0, IPC_STAT,arg))    {
      cout << "#semaphores in the set" << arg.mbuf->sem_nsems<< endl;
      arg.mbuf->sem_perm.uid = getuid(); // change owner user ID
```

```
        if (semctl(fd,0,IPC_SET,arg)==-1) perror("semctl");
    }
    else perror("semctl");
    if (semctl(fd,0,IPC_RMID,0)) perror("semctl - IPC_RMID");
    return 0;
}
```

10.6 POSIX.1b Semaphores

The POSIX.1b semaphores are created and manipulated in a manner similar to those in UNIX System V. Specifically, the <semaphore.h> header and the following APIs are defined by the POSIX.1b:

```
#include <semaphore.h>
sem_t*   sem_open( char* name, int flags, mode_t mode, unsigned init_value );
int      sem_init ( sem_t* addr, int pshared, unsigned init_value );
int      sem_getvalue ( sem_t* idp, int* valuep );
int      sem_close ( sem_t* idp );
int      sem_destroy ( sem_t* id );
int      sem_unlink ( char* name );
int      sem_wait ( sem_t* idp );
int      sem_trywait ( sem_t* idp );
int      sem_post ( sem_t* idp );
```

The POSIX.1b semaphores differ from those of UNIX System V in the following ways:

- POSIX.1b semaphores are either identified by a UNIX path name (as created via *sem_open*), or remain unnamed (but given a starting virtual address as created via *sem_init*). System V semaphores are identified by an integer key
- POSIX.1b creates one semaphore for each *sem_open* or *sem_init* call, whereas multiple System V semaphores can be created for each *semget* call
- A POSIX.1b semaphore value is increased or decreased by a value of 1 for each *sem_post* and *sem_wait* call, respectively. With System V semaphores, users can increase or decrease semaphore value by any integer value for each *semop* call

The *sem_open* function creates a semaphore whose name is given by the *name* argument. The syntax of the *name* argument value is the same as that for POSIX.1b messages. The *flags* argument value may be 0 if it knows that the named semaphore already exists. Otherwise, the O_CREAT flag specifies that a semaphore of the given name should be created. In addition, the O_EXCL flag may be specified with the O_CREAT flag to force the function to return a failure status if a semaphore of the given name already exists. The *mode* and *init_value* arguments are used for a newly create semaphore. Specifically, the *mode* argument value is the read-write permission for user, group, and others to be assigned to the new semaphore. The *init_value* argument value is an unsigned integer value to be assigned to the semaphore.

The function returns a *sem_t* pointer if it succeeds or -1 if it fails.

The *sem_init* function is an alternative to the *sem_open* function. A process that uses *sem_init* first allocates a memory region for the semaphore to be created. This memory region may be a shared memory if the semaphore is to be accessed by other processes. The memory region address is passed as value to the *addr* argument of *sem_init*. The *pshared* argument value is 1 if the semaphore is to be shared with other processes, 0 otherwise. The *init_value* argument specifies the initial integer value to be assigned to the semaphore. This value should not be a negative number.

The function returns a 0 value if it succeeds or -1 if it fails.

The *sem_getvalue* function returns the current value of a semaphore designated by the *idp* argument. The return value is passed via the *valuep* argument. The function returns 0 if it succeeds or -1 if it fails.

The *sem_post* function increases a semaphore value by 1, whether the *sem_wait* function decreases its value by 1. A semaphore is designated by the *idp* argument in both functions. If the value is already zero, the *sem_wait* function will block the calling process until it can succeed in its operation. The *sem_trywait* is similar to *sem_wait*, except that it is nonblocking and returns a -1 failure status (if it cannot decrease a specified semaphore value).

The *sem_close* and *sem_unlink* functions are used with semaphores created by the *sem_open* function. The *sem_close* function disassociates a semaphore from a process, and the *sem_unlink* function removes a semaphore from the system.

The *sem_destroy* function is used with semaphores that are created by the *sem_init* function. It deletes a semaphore from the system.

All the *sem_post*, *sem_wait*, *sem_trywait*, *sem_close*, *sem_unlink*, and *sem_destroy* functions return 0 if they succeed or -1 if they fail.

The following *test_semp.C* example creates a semaphore of the name */sem.0* and initializes it with a value of 1. If the semaphore is created successfully, the process does a *sem_wait* on it, bringing the semaphore value to 0, then performs a *sem_post* to increase the value back to 1. Finally, the process closes the semaphore handle via *sem_close* and removes the semaphore from the system via the *sem_unlink* API.

```
#include <stdio.h>
#include <sys/stat.h>
#include <semaphore.h>

int main()
{
   sem_t *semp = sem_open("/sem.0", O_CREAT, S_IRWXU, 1);
   if (semp==(sem_t*)-1)          { perror("sem_open"); return -1; }
   if (sem_wait(semp)==-1)        perror("sem_wait");
   if (sem_post(semp)==-1)        perror("sem_post");
   if (sem_close(semp)==-1)       perror("sem_close");
   if (sem_unlink("/sem.0") == -1) perror("sem_unlink");
   return 0;
}
```

The second example below, *test_semp2.C*, uses *sem_init* to create a semaphore at a dynamic address created by the process via *malloc*. The semaphore is specified as unshared (*pshare* value is zero), and its initial value is set to 1. If the semaphore is created, the *sem_getvalue* API is called to obtain the current value, and the process depicts the value to the standard output. Finally, the semaphore is destroyed from the system via the *sem_destroy* API.

```
#include <iostream.h>
#include <stdio.h>
#include <malloc.h>
#include <semaphore.h>

int main()
{
   int      val;
   sem_t    semval;
   if (sem_init ( &semval, 0, 1 )==-1) {
            perror("sem_init"); return 2;
   }
   if (sem_getvalue( &emval, &val)==0)
```

```
        cout << "semaphore value: " << val << endl;
    if (sem_destroy(&semval) == -1)
        perror("sem_destroy");
    return 0;
}
```

10.7 UNIX System V Shared Memory

Shared memory allows multiple processes to map a portion of their virtual addresses to a common memory region. Thus, any process can write data to a shared memory region and the data are readily available to be read and modified by other processes.

Shared memory was invented to improve on the performance problem of messages: when a message is sent from a process to a message queue, the data are copied from the process virtual address space to a kernel data region. Then when another process receives this message, the kernel copies the message data from the region to the receiving process's virtual address space. Thus, message data are copied twice: From process to kernel and then to another process. Shared memory, on the other hand, does not have this data transfer overhead: Shared memory is allocated in the kernel virtual address when a process reads or writes data via a shared memory. The data is manipulated directly in the kernel memory region. However, shared memory does not provide any access control method for processes that use it. Therefore, it is a common practice to use semaphores, along with shared memory, to implement an interprocess communication media.

After a process attaches to a shared memory region, it gets a pointer to reference the shared memory. It can be used as if it was obtained via a dynamic memory allocator (i.e., *new*). The only difference is that data in a shared memory are persistent and do not go away, even if the process creating the shared memory region terminates.

There can be multiple shared memory regions existing in a given system at any one time.

10.7.1 UNIX Kernel Support for Shared Memory

In UNIX System V.3 and V.4, there is a shared memory table in the kernel address space that keeps track of all shared memory regions created in the system. Each entry of the table stores the following data for one shared memory region:

- A name that is an integer ID key assigned by the a process that created the shared memory. Other processes may specify this key to "open" the region and get a descriptor for future attachment to or detachment from the region

- The creator user and group IDs. A process whose effective user ID matches a shared memory region creator user ID may delete the region and may change control data of the region
- The assigned owner user and group IDs. These are normally the same as those of the creator user and group IDs, but a creator process can set these values to assign different owner and group membership for the region
- Read-write access permission of the region for owner, group members, and others. A process that has read permission to the region may read data from it and query the assigned user and group IDs of the region. A process that has write permission to a region may write data to it
- The size, in number of bytes, of the shared memory region
- The time when the last process attached to the region
- The time when the last process detached from the region
- The time when the last process changed control data of the region

Figure 10.5 depicts the kernel data structure for shared memory.

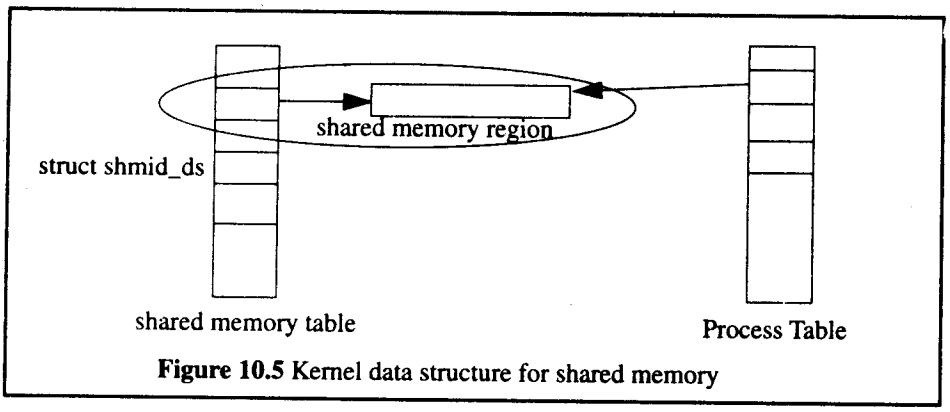

Figure 10.5 Kernel data structure for shared memory

Like messages and semaphores, shared memory is stored in kernel address space, and they are persistent, even if their creator processes no longer exist. Furthermore, each shared memory has an assigned owner, and only processes that have superuser, creator, or assigned owner privileges may delete the shared memory or change its control data.

Finally, there are several system-imposed limits on the manipulation of shared memory. These limits are defined in the <sys/shm.h> header:

System limit	Meaning
SHMMNI	The maximum number of shared memory regions that may exist at any given time in a system
SHMMIN	The minimum size, in number of bytes, of a shared memory region

System limit	Meaning
SHMMAX	The maximum size, in number of bytes, of a shared memory region

The effects of these system-imposed limits on processes are:

- If a process attempts to create a new shared memory, causing the SHMMNI limit to be exceeded, the process will be blocked until an existing region is deleted by another process
- If a process attempts to create a region whose size is less than SHMMIN or larger than SHMMAX, the system call will fail

10.7.2 The UNIX APIs for Shared Memory

The <sys/ipc.h> header defines a *struct ipc_perm* data type, which stores the owner user and group ID, creator user and group ID, assigned name key, and read-write permission of shared memory.

Each entry in the shared memory table is of type *struct shmid_ds*, which is defined in the <sys/shm.h> header. The data fields of the structure and the corresponding data it stores are:

Data field	Data Stored
shm_perm	Data stored in a *struct ipc_perm* record
shm_segsz	The shared memory region size, in number of bytes
shm_lpid	Process ID of the last process that attaches to the region
shm_cpid	Creator process ID
shm_nattch	Number of processes currently attached to the region
shm_atime	Time when the last process attached to the region
shm_dtime	Time when the last process detached from the region
shm_ctime	Time when the last process changed control data of the region

The UNIX System V shared memory APIs are:

Shared memory API	Uses
shmget	Open and create a shared memory
shmat	Attach a shared memory to a process virtual

	address space, so that the process can read and/or write data in the shared memory
shmdt	Detach a shared memory from the process virtual address space
shmctl	Query or change control data of a shared memory or delete the memory

The header files needed for the shared memory APIs are:

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

10.7.3 shmget

The function prototype of the *shmget* API is:

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

int shmget ( key_t key, int size, int flag );
```

This function "opens" a shared memory whose key ID is given in the *key* argument. The function returns a nonnegative integer descriptor that can be used in other shared memory APIs.

If the value of the *key* argument is a positive integer, the API attempts to open a shared memory whose key ID matches that value. However, if the *key* value is the manifested constant IPC_PRIVATE, the API allocates a new shared memory to be used exclusively by the calling process. Specifically, the "private" shared memory is usually allocated by a parent process, which then forks one or more child processes. The parent and child processes then use the shared memory to exchange data.

The *size* argument defines the size of the shared memory region that may be attached to the calling process via the *shmat* API. If this function call creates a new shared memory, its size will be defined by the *size* argument. However, if this call "opens" a preexisting shared memory, the *size* argument may be less than or equal to the allocated size of the shared memory. In the latter case, if *size* is less than the actual size of the shared memory, the calling process can access only the first *size* bytes of the shared memory.

If the *flag* argument is zero, the API fails if there is no shared memory whose key ID matches the given *key* value. Otherwise, it returns the descriptor for that set. If a process wishes to create a new shared memory with the given *key* ID (if none preexists), then the *flag* value should be the bitwise-OR of the manifested constant IPC_CREAT and the read-write access permission for the new memory.

This function returns a positive descriptor if it succeeds or -1 if it fails.

10.7.4 shmat

The function prototype of the *shmat* API is:

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

void * shmat ( int shmid, void* addr, int flag );
```

This function attaches a shared memory referenced by *shmid* to the calling process virtual address space. The process can then read/write data in that shared memory. Note that if this is a newly created shared memory, the kernel does not actually allocate the memory region until the first process calls this function to attach to it.

The *addr* argument specifies the desired starting virtual address in the calling process to which location the shared memory should be mapped. If this value is 0, the kernel is free to find an appropriate virtual address in the calling process to map to the shared memory. Most applications should set the *addr* value to zero, unless they explicitly store pointer or address references in the shared memory (e.g., keeping a linked list in the region). It becomes important for every process attached to the shared memory to specify the same virtual address (mapped to the shared memory).

The *flag* argument may contain the flag SHM_RND if the *addr* value is nonzero. The SHM_RND flag instructs the kernel that the virtual address specified in the *addr* argument may be rounded off to align with the page boundary. If the SHM_RND flag is not specified and the *addr* argument is not zero, the API fails (if the kernel cannot map the shared memory to the specified virtual address).

Another possible value of the *flag* argument is SHM_RDONLY, which means the calling process attaches to the shared memory for read-only. If this flag is not set, then by default the process may read and write data in the shared memory -- subject to permissions established by the creator of the shared memory region

The return value of this API is the mapped virtual address of the shared memory or -1 if it fails. Note that a process may call *shmat* multiple times to attach shared memory to multiple virtual addresses.

10.7.5 shmdt

The function prototype of the *shmdt* API is:

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

int shmdt ( void* addr );
```

This function detaches (or unmaps) shared memory from the specified *addr* virtual address of the calling process.

The *addr* value should be obtained from a *shmat* call prior to this function call.

The return value of the function is 0 if it succeeds or -1 if it fails.

The following *test_shm.C* program "opens" a shared memory with a size of 1024 bytes and the key ID value of 100. If the shared memory does not preexist, it is created by the *shmget* call, and its access permission is read-write for everyone.

After the shared memory is "opened," it is attached to the process virtual address via the *shmat* call. It then writes the message *Hello* to the beginning region of the memory and detaches from the memory. Any other process on the same system can now attach to that shared memory and read the message accordingly.

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
   int perms = S_IRWXU | S_IRWXG | S_IRWXO;
```

```
    int fd = shmget (100, 1024, IPC_CREAT | perms);
    if (fd==-1) perror("shmget"), exit(1);
    char* addr = (char*)shmat(fd, 0, 0);
    if (addr==(char*)-1) perror("shmat"), exit(1);
    strcpy( addr, "Hello");
    if (shmdt(addr)==-1) perror("shmdt");
    return 0;
}
```

10.7.6 shmctl

The function prototype of the *shmctl* API is:

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

int shmctl ( int shmid, int cmd, struct shmid_ds* buf );
```

This API can either query or change the control data of a shared memory designated by *shmid,* or delete the memory altogether.

The *shmid* value is the shared memory descriptor obtained from a *shmget* function call.

The *buf* argument is the address of a *struct shmid_ds*-type object that may be used to specify or retrieve the control data of a shared memory, as determined by the *cmd* argument. The possible values of *cmd* and their meanings are:

***cmd* value**	**Meaning**
IPC_STAT	Copy control data of the shared memory to the object pointed to by *buf.* The calling process must have read permission to the set
IPC_SET	Change the control data of the shared memory by the data specified in the object pointed to by *buf.* The calling process must be the superuser, creator, or assigned owner of the shared memory to be able to perform this task. Furthermore, this API can set only the region's owner user and group IDs and access permission
IPC_RMID	Remove the shared memory from the system. The calling process must be the superuser, creator, or

	assigned owner of the region to be able to perform this task. Note that if a shared memory to be removed has one or more processes attached to it, the removal operation will be delayed until these processes detach from it
SHM_LOCK	Lock the shared memory in memory. The calling process must have superuser privileges to perform this task
SHM_UNLOCK	Unlock the shared memory in memory. The calling process must have superuser privileges to perform this task

The function returns 0 if it succeeds or -1 if it fails.

The following *test_shm2.C* program "opens" a shared memory with the key ID of 100 and calls *shmctl* to retrieve the control data of the region. If both the *shmget* and *shmctl* calls are successful, the process prints to the standard output the size of the shared memory. It then sets the shared memory owner user ID as its owner user ID via another *shmctl* call. Finally, the process invokes *shmctl* again to remove the shared memory.

```
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
   struct shmid_ds sbuf;
   int fd = shmget (100, 1024, 0);
   if (fd>0 && shmctl(fd, IPC_STAT,&sbuf))    {
      cout << "shared memory size if: " << sbuf.shm_segsz << endl;
      sbuf.shm_perm.uid = getuid();          // change owner user ID
      if (shmctl(fd,IPC_SET,&sbuf)==-1) perror("shmctl");
   }
   else perror("shmctl");
   if (shmctl(fd,IPC_RMID,0)) perror("shmctl - IPC_RMID");
   return 0;
}
```

10.7.7 Semaphore and Shared Memory Example

This section depicts another version of the client/server application shown in Section 10.3.7. This new version uses semaphores and shared memory to implement the message queue instead of using messages. Thus, the *message.h* header needs to be changed significantly. However, the message class interface to the client and server programs, as well as the *client.C* and *server.C* modules, are exactly the same as those in Section 10.3.7. This is the benefit of C++ classes -- as long as the external interfaces of a class are unchanged, no applications that make use of that class need be modified, even if the internal implementation of the class has changed dramatically.

To implement a message queue using semaphores and shared memory, the shared memory provides a kernel memory region to store any messages sent to the queue. The semaphores control which process can access the shared memory (to read or write a message) at any one time. Specifically, there may be multiple client processes sending requests to the server process simultaneously, using the same *semop* system calls to manipulate the semaphores. The design of these *semop* calls must ensure that either they are all blocked while the server is actively accessing the shared memory or only one of the client processes is actively accessing the shared memory (all the other client processes and the server process are blocked). To meet these objective, two semaphores will be used. The assignment of these semaphore values, for various purposes, are:

Semaphore 0	Semaphore 1	Usage
0	1	Server is waiting for a client to send message
1	0	Client's message is ready for server to read
1	1	Server's response data are ready for a client

The interaction of client processes and the server process to the semaphore set is:

- The server initially creates the semaphore set and a shared memory. It initializes the semaphore set value to be 0, 1 (i.e., the first semaphore value is zero and the second semaphore's value is one)
- The server waits for a client to send a request to the shared memory by performing a *semop* call on the semaphore set with the supplied values of -1, 0. This blocks the server, as the current value of the set is 0,1 and none of the semaphores in the set can be changed by the *semop* call
- When one or more clients attempt to send a message to the shared memory, they all perform a *semop* call with the supplied values 0,-1. One of them will succeed, as the set value is 0,1 at that moment. However, as soon as a client process succeeds in its *semop* call, it immediately changes values to 0,0. This blocks all other clients that are performing the semop call of 0, -1 and continues blocking the server performing the *semop* call of -1,0
- The client process that succeeds in changing the semaphore set value can now write

a service request command and its PID to the shared memory. After it is done, it will perform a *semop* call with the value of 1,0. This unblocks the server process from its *semop* call, but the new value continues to block other client processes that are doing the *semop* call of 0,-1. If the service request command is not QUIT_CMD, the client process will perform *semop* call with the values of -1,-1 and block the client

- Once the server is unblocked it will read the client service request from the shared memory. If the command is QUIT_CMD, it will deallocate the shared memory and semaphore set, then terminate itself. However, if the command is not QUIT_CMD, the server writes the response data to the shared memory, then performs the *semop* call with the values of 1,1. This will unblock the client process that is performing the *semop* call of -1,-1. Other client processes which are performing the *semop* call of 0,-1 are still blocked by the new semaphore values. After the *semop* call, the server will go back to the state of (b) (to wait for a service request from a new client)
- The client that is unblocked by the server sets the semaphore set value to 0,0 and reads the server's response data. It prints the data to the standard output and then sets the semaphore values to 0,1 before it terminates itself. The last *semop* call sets the system back to the state of (c) above, and one of the clients will be unblocked and start interacting with the server via the shared memory and semaphores

The semaphore value transition, in the different stages of a client/server interaction, is shown in Figure 10.6 (the semaphore values are shown inside the ovals):

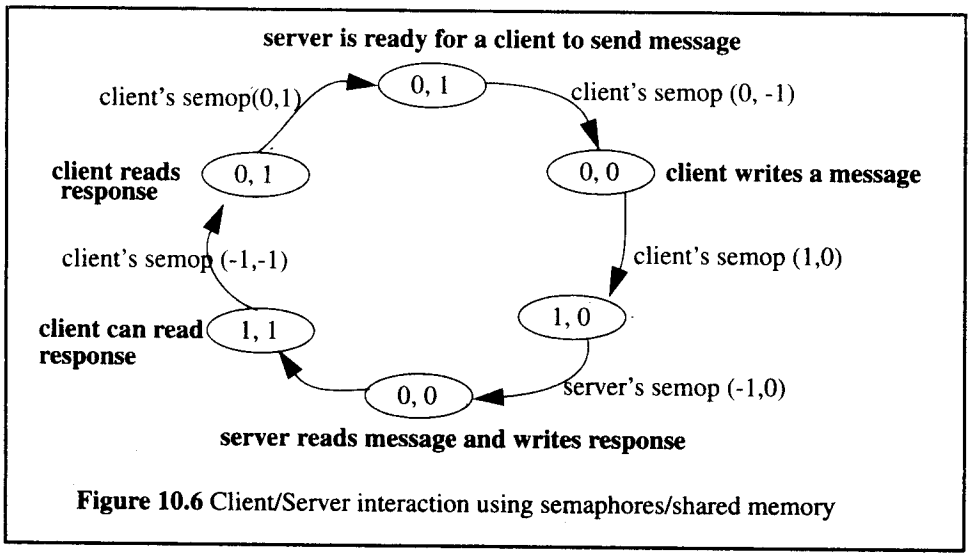

Figure 10.6 Client/Server interaction using semaphores/shared memory

The *message.h* header described in Section 10.3.7 is modified to use shared memory and semaphores instead of the message queue. The new message class is declared in the following *message3.h* header:

```
#ifndef MESSAGE3_H
#define MESSAGE3_H
#include <strstream.h>
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <sys/errno.h>

/* common declarations for daemon/server process */
enum { MSGKEY=186, MAX_LEN=256, SHMSIZE=1024, SEMSIZE=2 };
enum { LOCAL_TIME = 1, UTC_TIME = 2, QUIT_CMD = 3,
       ILLEGAL_CMD = 4, SEM_RD = 0, SEM_WR=1 };

struct mgbuf
{
   long     mtype;
   char     mtext[MAX_LEN];
};

class message
{
   private:
      int   shmId, semId;
      struct mgbuf  *msgPtr;
      enum ipc_op {  RESET_SEM, CLIENT_GET_MEM,
                     CLIENT_SND_REQ, SERVER_RCV_REQ,
                     SERVER_GET_MEM, SERVER_SND_RPY,
                     CLIENT_RCV_RPY
                  };
   public:

      /* try to change semaphores' values */
      void getsem( enum ipc_op  opType )
      {
         static struct sembuf  args[2] = { {SEM_RD}, {SEM_WR} };
         switch (opType)    {
            case SERVER_GET_MEM:
                  return;
            case CLIENT_GET_MEM:
                  args[SEM_RD].sem_op = 0,
                  args[SEM_WR].sem_op = -1;
```

```
                break;
        case CLIENT_SND_REQ:
                args[SEM_RD].sem_op = 1,
                args[SEM_WR].sem_op = 0;
                break;
        case SERVER_RCV_REQ:
                args[SEM_RD].sem_op = -1,
                args[SEM_WR].sem_op = 0;
                break;
        case SERVER_SND_RPY:
                args[SEM_RD].sem_op = 1,
                args[SEM_WR].sem_op = 1;
                break;
        case CLIENT_RCV_RPY:
                args[SEM_RD].sem_op = -1,
                args[SEM_WR].sem_op = -1;
                break;
        case RESET_SEM:
                args[SEM_RD].sem_op = 0,
                args[SEM_WR].sem_op = 1;
    }
    if (semop(semId,args,SEMSIZE)==-1) perror("semop");
};

/* constructor function */
message( int key )
{
    if ((shmId=shmget(key, SHMSIZE, 0))==-1) {
        if (errno==ENOENT) {    // create a brand new message object
            if ((shmId= shmget(key, SHMSIZE, IPC_CREAT|0666)) ==-1)
                perror("shmget");
            else if ((semId=semget(key, SEMSIZE, IPC_CREAT|0666))
                        ==-1)
                perror("semget");
            else getsem(RESET_SEM); // initialize a new semaphore set
        }
        else perror("shmget");
    }
    else if ((semId=semget(key,0,0))==-1)  /* get existing semaphores */
        perror("semget");

    if (shmId>=0 && !(msgPtr=(struct mgbuf*)shmat(shmId,0,0)))
        perror("shmat");
};
```

```
    /* destructor function */
    ~message() {};

    /* check message queue open status */
    int good() { return (shmId >= 0 && semId>=0) ? 1 : 0; };

    /* remove message queue */
    int rmQ()
    {
       if (shmdt((char*)msgPtr)<0) perror("semdt");
       if (!semctl(semId,0,IPC_RMID,0) && !shmctl(shmId,IPC_RMID,0))
          return 0;
       perror("shmctl or semctl");
       return -1;
    };

    /* send a message */
    int send( const void* buf, int size, int type)
    {
       int server = (type > 99);
       getsem(server ? SERVER_GET_MEM : CLIENT_GET_MEM);
       memcpy(msgPtr->mtext,buf,size);
       msgPtr->mtext[size] = '\0';
       msgPtr->mtype = type;
       getsem(server ? SERVER_SND_RPY : CLIENT_SND_REQ);
       return 0;
    };

    /* receive a message */
    int rcv( void* buf, int size, int type, int* rtype)
    {
       int server = (type < 0);
       getsem(server ? SERVER_RCV_REQ : CLIENT_RCV_RPY);
       memcpy(buf,msgPtr->mtext,strlen(msgPtr->mtext)+1);
       if (rtype) *rtype = msgPtr->mtype;
       if (!server) getsem(RESET_SEM);
       return strlen(msgPtr->mtext);
    };
};  /* message */

#endif
```

In the new *message.h* header, *getsem* is a utility function that performs *semop* on a semaphore set based on the actual *opType* argument values (as assigned by either a server or a client process).

The *message* constructor function "opens" a shared memory and semaphore set with two elements. These two objects have the same key ID. If the semaphore set is a brand new object, it will be initialized to the initial values of 0, 1. This signifies that the first semaphore value is zero and the second semaphore value is 1.

The *send* function "sends" a message to the shared memory. Because the *semop* operations is different between a server and a client process, the function uses the *type* argument value to identify whether the calling process is a server or a client. If the *type* argument value is greater than 99, the function is called by a client process. Otherwise, it is called by a server process. The function blocks the *getsem* call until the process can complete its *semop* operation. After that it writes a message to the shared memory and calls *getsem* again to set semaphore values that unblock its counterpart process.

The *read* function works similarly: The *semop* operations which read a message from the shared memory are different between a server and a client process. Thus, it uses the *type* argument value to identify the calling process: If the *type* argument value is less than zero (actual is -99), the caller is a server process (otherwise, it is a client process). The function blocks the *getsem* call until the process can complete its *semop* operation. After that it reads a message from the shared memory and resets the semaphores to the initial values 0,1 (if the process is a client).

The *rmQ* function is called when a server process receives the QUIT_CMD from a client process. The function invokes the *semctl* and *shmctl* APIs to delete the semaphore set and the shared memory, respectively then to terminate the server process. This is necessary as semaphores and shared memory are persistent objects in kernel space, even after those processes that created them are terminated.

The output of the new server and client processes using the new *message.h* is similar to the one using messages, as seen in Section 10.3.7:

```
ch13 % mserver &
[1] 337
ch13 % server: start executing...
ch13 % mclient
client: start executing...
cmd> 1
server: receive cmd #1, from client: 338
client: 338 Thu Jan 26 21:50:59 1995
cmd> 2
server: receive cmd #2, from client: 338
client: 338 Fri Jan 27 05:51:01 1995
cmd> 4
```

server: receive cmd #4, from client: 338
client: 338 Illegal cmd: 4
cmd> 3
client: 338 exiting...
server: receive cmd #3, from client: 338
server: deleting msg queue...
[1] Done mserver

10.8 Memory Mapped I/O

Mmap is a creation of BSD UNIX. It allows a process to map its virtual address space directly to a file object memory page in a kernel space. The process can read and write data with the file object directly via the mapped memory. Furthermore, if more than one process maps to the same file object simultaneously, they share a mapped memory region. They can communicate with each other in a manner similar to that of using a shared memory.

Mmap differs from the regular UNIX file APIs in that after a file is opened for access (a process calls *read* to read data from the file), the kernel fetches one or more pages of the requested data from the file's hard disk storage. The data is then put into a kernel memory region and then copied into a buffer in the calling process's virtual address. The reverse situation applies to the *write* API: when a process calls *write* to write data to a file, the kernel copies the data from the process buffer into a kernel memory region. When the memory region is filled or the process requests to flush the buffer, the data are copied to the file's hard disk.

However, if the same process uses *mmap* instead, the kernel still fetches one or more pages of file data from its hard disk storage and puts them into a kernel memory region. In this case the process can directly access data in the memory region by referencing the virtual addresses mapped to the region. Thus, *mmap* is more efficient in manipulating file data. Any data written to a mapped region are stored in its corresponding file object automatically.

One application of *mmap* is to develop programs that can resume execution after being previously terminated. For example, a database management program can use *mmap* to map its virtual address to a database file, and all the data it manipulates are stored in the mapped region. When the process is terminated, the data are stored in the database file automatically. When its program is executed again, the new process maps to the database file and all previously written data are readily available for further use.

Another use of *mmap* is to emulate the shared memory function. Specifically, two or more processes that wish to perform interprocess communication can use *mmap* to map to the same file object. They can then read and write data to each other via their mapped virtual addresses. A later section will show how to use *mmap* to reimplement the client/server application, as depicted in the previous section.

10.8.1 Memory mapped I/O APIs

The <sys/mman.h> header declares all the *mmap* APIs:

API	Meaning
mmap	Maps a process virtual address space to a file object
munmap	Disassociates a process virtual address from a file object
msync	Synchronizes mapped memory region data with its corresponding file object data on a hard disk.

10.8.2 mmap

The function prototype of the *mmap* API is:

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

caddr_t   mmap ( caddr_t  addr, int size, int prot, int flags, int fd, off_t pos );
```

This function maps a file object designated by *fd* to the virtual address of a process starting at *addr*. If the *addr* value is zero, the kernel assigns a virtual address to map. The *pos* argument specifies the starting location in the file object that is mapped to *addr*. Its value should be either zero or a multiple of the memory page size (use the *getpagesize* or *sysconf* API to get the system memory page size value). The *prot* argument specifies the access permission of the mapped memory. Its possible values and meanings are:

prot value	Meaning
PROT_READ	The mapped region can be read
PROT_WRITE	The mapped region can be written
PROT_EXEC	The mapped region can be executed

The *flags* argument specifies mapping options. Its possible values and meanings are:

flags value	Meaning
MAP_SHARED	Data written to the mapped region are visible to other processes that are mapped to the same file object

MAP_PRIVATE	Data written to the mapped region are not visible to other processes that are mapped to the same file object
MAP_FIXED	The *addr* value must be the starting virtual address of the mapped region. The function fails if this cannot be accomplished. If this flag is not specified or if the MAP_VARIABLE flag is defined and specified by a system, the kernel can select a different virtual address than *addr* for the mapped region

The return value of the function is the actual virtual address where the mapped region starts or MAP_FAILED if the function fails.

The PROT_EXEC flag is used when the file object to be mapped is an executable file and the calling process has the superuser privilege. When a user invokes a command in a UNIX system, it is common for the kernel to perform a *mmap* of the command's executable file. This executes the instruction code of the program directly from the file's mapped memory.

The MAP_PRIVATE flag specifies that any data written to a mapped memory are not visible to other processes mapped to the same file object. However, this also disables written data in the mapped memory from being stored back into the object's disk file. Furthermore, suppose processes *A* and *B* both map to a file called *FOO*; process *A* specifies MAP_PRIVATE and *B* specifies MAP_SHARED. If process B writes data to the mapped memory before *A* does, the new data are seen by both processes. However, once *A* writes data to the shared memory, the kernel creates a private copy of the memory pages that *A* has modified, while *B* is still using the old page. From that point onward, any data written by *A* or *B* to their respective memory pages are not visible to each other.

MAP_PRIVATE is used by a debugger process to *mmap* a program for execution. The debugger often writes user-defined breakpoints to the instruction code of the debugged process. Those breakpoints should not be reflected in the executable file of the debugged program, nor should they be seen by other processes that are also mapping the same program.

Before a process can call *mmap* to map a file object, it must call the *open* API and assign the file descriptor to the *fd* argument. Furthermore, if the file object is newly created by the *open* call, the process should write at least *size* bytes of data to initialize the file. This is done because *mmap* does not allocate any memory region. It simply marks that the process's virtual address (from *addr* to *addr+size)* is legal to use. If the file size is less than that, there is no memory allocated by the kernel beyond the memory page that contains the virtual address *addr+<file_size>*. If the process attempts to access data anywhere between *addr+<file_size>* and *addr+size*, it may receive a SIGBUS signal.

The following *test_mmap.C* program "opens" a new file called FOO and initializes its size to SHMSIZE byte (with the data of "\0"). It then closes the file descriptor *fd*, as it is no longer needed. Finally, the process writes the string *Hello* to the mapped region and then terminates itself. If a user looks at the content of the file via the *cat* command, he or she would notice that the file contains the string *Hello*.

```
#include <strstream.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
const int SHMSIZE = 1024;

int main()
{
    int ch='\0', fd = open ("FOO", O_CREAT | O_EXCL,0666);
    if (fd==-1) {
            perror("file exists"), exit(1);
    };
    for (int i=0; i < SHMSIZE; i++)          /* Initialize the file */
            write(fd, &ch, 1);
    caddr_t memP=mmap(0,SHMSIZE,PROT_READ|PROT_WRITE,
                        MAP_SHARED, fd, 0);
    if (memP==MAP_FAILED)   {
            perror("mmap");
            exit(2);
    }
    close(fd);                               /* don't need this anymore */
    ostrstream(memP, SHMSIZE) << "Hello UNIX\n";
}
```

10.8.3 munmap

The function prototype of the *munmap* API is:

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

int     munmap ( caddr_t addr, int size );
```

This function disassociates a mapped region from the process virtual address. The unmapped region starts at the *addr* virtual address and extends up to the memory page that contains the *addr+size* virtual address.

The function returns 0 if it succeeds or -1 if it fails.

10.8.4 msync

The function prototype of the *msync* API is:

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

int     msync ( caddr_t addr, int size, int flags );
```

This function synchronizes data in a mapped region with its corresponding file object data on the hard disk. If *size* is 0, all modified pages in the region that contain *addr* are synchronized. If *size* is greater than 0, only the pages containing *addr* to *addr+size* are synchronized.

The *flags* argument specifies the synchronization method. Its possible values and meanings are:

flags value	Meaning
MS_SYNC	Flush data from mapped region to hard disk. Wait for data transfer to complete
MS_ASYNC	Flush data from mapped region to hard disk. Do not wait for the data transfer to complete
MS_INVALIDATE	Invalidate the data in the mapped region. Next reference to the region will cause new pages to be fetched from the hard disk.

The function returns 0 if it succeeds or -1 if it fails.

10.8.5 Client/Server Program Using Mmap

The client/server example, as depicted in Section 10.7.7, can be changed easily to use *mmap* instead of shared memory. Once again, the only changes required are in the *message.h* header. The *client.C* and *server.C* modules are unchanged, as shown in Section 10.3.7.

The new *message4.h* header which uses *semaphores* and *mmap* is as follows:

```
#ifndef MESSAGE4_H
#define MESSAGE4_H

#include <strstream.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <memory.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/mman.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <sys/errno.h>
/* common declarations for daemon/server process */
enum { MSGKEY=186, MAX_LEN=256, SHMSIZE=1024, SEMSIZE=2 };
enum { LOCAL_TIME = 1  UTC  TIME = 2, QUIT_CMD = 3,
        ILLEGAL_CMD = 4, SEM_RD = 0, SEM_WR=1 };
struct mgbuf
{
   long    mtype;
   char    mtext[MAX_LEN];
};

class message
{
   private:
      int            semId;
      struct mgbuf  *msgPtr;
      enum ipc_op {  RESET_SEM, CLIENT_GET_MEM,
                     CLIENT_SND_REQ, SERVER_RCV_REQ,
                     SERVER_GET_MEM, SERVER_SND_RPY,
                     CLIENT_RCV_RPY };
   public:
```

```
/* try to change semaphores' values */
void getsem( enum ipc_op  opType )
{
   static struct sembuf  args[2] = { {SEM_RD}, {SEM_WR} };
   switch (opType)    {
      case SERVER_GET_MEM:
            return;
      case CLIENT_GET_MEM:
            args[SEM_RD].sem_op = 0,
            args[SEM_WR].sem_op = -1;
            break;
      case CLIENT_SND_REQ:
            args[SEM_RD].sem_op = 1,
            args[SEM_WR].sem_op = 0;
            break;
      case SERVER_RCV_REQ:
            args[SEM_RD].sem_op = -1,
            args[SEM_WR].sem_op = 0;
            break;
      case SERVER_SND_RPY:
            args[SEM_RD].sem_op = 1,
            args[SEM_WR].sem_op = 1;
            break;
      case CLIENT_RCV_RPY:
            args[SEM_RD].sem_op = -1,
            args[SEM_WR].sem_op = -1;
            break;
      case RESET_SEM:
            args[SEM_RD].sem_op = 0,
            args[SEM_WR].sem_op = 1;
   }
   if (semop(semId,args,SEMSIZE)==-1) perror("semop");
};

/* constructor function */
message( int key )
{
   char mfile[256], fillchr='\0';
   ostrstream(mfile,sizeof mfile) << "FOO" << key << '\0';
   int fd =open(mfile,O_RDWR,0);
   if (fd==-1) {          /* a new file */
      if ((fd=open(mfile,O_RDWRIO_CREATIO_TRUNC,0777))==-1)
         perror("open");
      else     {
```

```
            /* zero fill the file for mmap to work.
               This is system dependent */
            for (int i=0; i < SHMSIZE; i++) write(fd, &fillchr, 1);
            if ((semId=semget(key, SEMSIZE, IPC_CREAT|0666))==-1)
               perror("semget");
            else getsem(RESET_SEM);  // initialize a new semaphore set
         }
      }
      else {                  /* connect to an existing entry */
         if ((semId=semget(key, 0, 0))==-1) perror("semget");
            if ((msgPtr=(struct mgbuf*)mmap(0, SHMSIZE, PROT_READ |
                  PROT_WRITE, MAP_SHARED, fd,0))== MAP_FAILED)
               perror("mmap");
            else close(fd);
      }
};

/* destructor function */
~message() {};

/* check message queue creation status */
int good() { return (semId>=0) ? 1 : 0; };

/* remove message queue */
int rmQ()
{
   if (!semctl(semId,0,IPC_RMID,0) &&
                        !munmap((caddr_t)msgPtr,SHMSIZE))
      return 0;
   perror("shmctl or semctl");
   return -1;
};

/* send a message */
int send( const void* buf, int size, int type)
{
   int server = (type > 99);
   getsem(server ? SERVER_GET_MEM : CLIENT_GET_MEM);
   memcpy(msgPtr->mtext,buf,size);
   msgPtr->mtext[size] = '\0';
   msgPtr->mtype = type;
   getsem(server ? SERVER_SND_RPY : CLIENT_SND_REQ);
```

```
        return 0;
    };

    /* receive a message */
    int rcv( void* buf, int size, int type, int* rtype)
    {
        int server = (type < 0);
        getsem(server ? SERVER_RCV_REQ : CLIENT_RCV_RPY);
        memcpy(buf,msgPtr->mtext,strlen(msgPtr->mtext)+1);
        if (rtype) *rtype = msgPtr->mtype;
        if (!server) getsem(RESET_SEM);
        return strlen(msgPtr->mtext);
    };
}; /* message */
#endif      /* MESSAGE4_H */
```

The changes in the new *message.h* header are in the *message* constructor, where the *mmap* call replaces the *shmget* call. The map file name is constructed with a file name prefix of *FOO* (this is chosen arbitrarily) and followed by the given key ID. Note that if the file is newly created, it is initialized with SHMSIZE bytes of NULL characters. This is to ensure that the entire mapped memory region is allocated by the kernel to hold valid data.

Another change in the *message.h* header is in the *rmQ* function. This function is called when a server process is terminating and needs to delete its semaphore set and detach from the mapped memory.

The rest of the *message.h* code is the same as that in Section 10.7.7. The output of the new client/server program is similar to that in Section 10.7.7 also.

10.9 POSIX.1b Shared Memory

The POSIX.1b shared memory APIs are:

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

int     shm_open( char* name, int flags, mode_t mode );
int     shm_unlink ( char* name );
```

The *shm_open* function creates a shared memory region whose name is given by the *name* argument. The syntax of the *name* argument value is the same as that for POSIX.1b messages. The *flags* argument contains the memory access flags (O_RDWR, O_RDONLY, or O_WRONLY) and any O_CREAT and O_EXCL flags. The *mode* argument is used if a new shared memory is created by this call. Its value is the read-write access permission (for user,

group, and others) assigned to the new shared memory.

The function returns a -1 if it fails, or a nonnegative handle if it succeeds.

Note that unlike the System V *shmget* API, the *shm_open* API does not specify the size of the shared memory region, which is defined in a subsequent *ftruncate* call. The *ftruncate* function prototype is:

```
int     ftruncate ( int fd, off_t shared_memory_size );
```

where the *fd* argument value is a shared memory handle, as returned by a *shm_open* call. The *shared_memory_size* argument contains the size of the shared memory region to be allocated. Once a shared memory is allocated and its size defined, the *mmap* function should be called. This maps the shared memory region to the virtual address space of the calling process.

After a process finishes using a shared memory, it calls the *munmap* function to unmap the shared memory from its virtual address space. Then the *shm_unlink* function may be called to remove the shared memory from the system. The argument to the *shm_unlink* function is a UNIX path name for a shared memory region.

The following *test_shmp.C* program "opens" a read-write-accessible shared memory with the name */shm.0,* via the *shm_open* call and sets its size with the *ftruncate* function call. The shared memory is mapped to the process's virtual address space via the *mmap* call. The *memp* variable holds the starting address of the shared memory.

Once the shared memory is set up in the process, the *sem_init* function is called to create a semaphore at the beginning address of the shared memory. The process works with the semaphore and shared memory. After the process is done, it calls the *sem_destroy* function to remove the semaphore from the system. It finally calls the *shm_unlink* and *shm_unmap* functions to remove the shared memory from the system and to unmap the shared memory from the virtual address.

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <sys/mman.h>
```

```
int main()
{
   long siz = sizeof(sem_t) + 1024;
   int shmfd = shm_open ("/shm.0", O_CREATIO_RDWR, S_IRWXU);

   if (shmfd==-1)     { perror("shm_open"); return 1; }
   if (ftruncate(shmfd, siz)==-1)
                      { perror("ftruncate"); return 2; }

   char* memp = (char*)mmap(0, siz, PROT_READ IPROT_WRITE,
                                          MAP_SHARED, shmfd, 0L);
   if (!memp)         { perror("mmap"); return 3; }

   (void)close(shmfd);
   if (sem_init((sem_t*)memp,1,1) < 0) { perror("sem_init"); return 4; }
   /* do work with the shared memory and semaphore */
   if (sem_destroy((sem_t*)memp) < 0) { perror("sem_destroy"); return 5; }
   if (shm_unlink("/shm.0") < 0) { perror("shm_unlink"); return 6; }
   return shm_unmap( memp, sizeof(sem_t) + 1024);
}
```

10.9.1 POSIX.1b Shared Memory and Semaphore Example

The client/server example depicted in Section 10.7.7 is rewritten using POSIX.1b shared memory and semaphore. Once again, the only changes required are in the *message.h* header. The *client.C* and *server.C* modules are unchanged, as shown in Section 10.3.7.

The new *messag5.h* header that contains a message class based on POSIX.1b shared memory and semaphore is:

```
#ifndef MESSAGE5_H
#define MESSAGE5_H

/* specify the following source code is POSIX.1b compliant */
#define   _POSIX_C_SOURCE          199309L
#include <strstream.h>
#include <stdio.h>
```

```
#include <memory.h>
#include <unistd.h>
#include <string.h>
#include <limits.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <sys/mman.h>

/* common declarations for daemon/server process */
enum { MSGKEY=186, MAX_LEN=256, MAX_MSG=20 };
enum { LOCAL_TIME=1, UTC_TIME=2, QUIT_CMD=3
                                        , ILLEGAL_CMD=4 };
/* data record for one message */
struct mgbuf
{
    long        mtype;                  // msg type
    char        mtext[MAX_LEN];         // msg text
};

/* data record for one shared memory region */
struct shm_header
{
    sem_t       semaphore;              // semaphore
    struct mgbuf  msgList[MAX_MSG];     // msg. list
};

/* message class */
class message
{
  private:
    struct shm_header   *memptr;
    sem_t               *sem_id;
    char                mfile[256];
    enum ipc_op { GET_MEM, SND_RPY, RCV_REQ, RESET_SEM };
  public:
    /* constructor function */
    message( int key )
    {
      /* create a shared memory region */
```

```
    ostrstream(mfile,sizeof mfile) << "FOO" << key << '\0';
    int fd = shm_open(mfile, O_CREATIO_RDWR,
                              S_IRWXUIS_IRWXGIS_IRWXO);
    if (fd==-1) { perror("shm_open"); return; }
    (void)ftruncate(fd,sizeof(struct shm_header));

    /* map shared memory to a process address space */
    if ((memptr=(struct shm_header*)mmap(0,
                              sizeof(struct shm_header),
            PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0))
            == MAP_FAILED)        {
        perror("mmap");
        return;
    }

    close(fd);

    /* create a semaphore in the shared memory region */
    sem_id = (sem_t*)&memptr->semaphore;
    if ((sem_init(sem_id, 1, 1))==-1) perror("sem_init");

    /* initialize msg list to be empty */
    for (int i=0; i < MAX_MSG; i++)
        memptr->msgList[i].mtype = INT_MIN;
};

/* destructor function: unmap shared memory from process */
~message() { munmap(memptr, sizeof(struct shm_header)); };

/* check share memory creation status */
int good() { return (memptr) ? 1 : 0; };

/* remove shared memory and semaphore */
int rmQ()
{
    if (sem_destroy(sem_id)==-1) perror("sem_destroy");
    if (shm_unlink(mfile)==-1) perror("shm_unlink");
    return munmap(memptr, sizeof(struct shm_header));
};
```

```
/* try to change semaphore's value */
void getsem( enum ipc_op  opType )
{
   switch (opType)    {
      case GET_MEM:
      case RCV_REQ:
            if (sem_wait(sem_id)==-1) perror("sem_wait");
            break;
      case SND_RPY:
      case RESET_SEM:
            if (sem_post(sem_id)==-1) perror("sem_post");
            break;
   }
};  /* getsem */

/* send a message to message queue*/
int send( const void* buf, int size, int type)
{
   getsem(GET_MEM);                        // acquire semaphore
   for (int i=0; i < MAX_MSG; i++)
      if (memptr->msgList[i].mtype==INT_MIN)   {
            /* find an empty slot in message queue to store msg */
            memcpy(memptr->msgList[i].mtext, buf, size);
            memptr->msgList[i].mtext[size] = '\0';
            memptr->msgList[i].mtype      = type;
            break;
      }
   if (i >= MAX_MSG)   {                   // msg queue is full
      cerr << "Too many messages in the queue!\n";
      return -1;                           // return failure
   }
   getsem(SND_RPY);                        // release semaphore
   return 0;                               // return OK
};  /* send */

/* receive a message */
int rcv( void* buf, int size, int type, int* rtype)
{
   do {
```

```
            getsem(RCV_REQ);                    // acquire semaphore
            int lowest_type = -1;
            for (int i=0; i < MAX_MSG; i++)     {
                if (memptr->msgList[i].mtype==INT_MIN) continue;

                /* done if type==0 or type matches msg. type */
                if (!type || type==memptr->msgList[i].mtype) break;

                /* if type < 0 find the lowest msg type < type */
                if (type < 0 && -type >= memptr->msgList[i].mtype)
                    if (lowest_type==-1 || (memptr->msgList[i].mtype <
                                memptr->msgList[lowest_type].mtype))
                        lowest_type = i;
            }
            if (i < MAX_MSG || lowest_type !=-1) {      // found one msg
                if (lowest_type!=-1) i = lowest_type;
                /* copy msg text and type to caller's variables */
                memcpy(buf,memptr->msgList[i].mtext,
                                strlen(memptr->msgList[i].mtext)+1);
                if (rtype) *rtype = memptr->msgList[i].mtype;
                /* mark queue slot as empty */
                memptr->msgList[i].mtype = INT_MIN;
                getsem(RESET_SEM);          // release semaphore
                return strlen((char*)buf);          // return msg. size
            }
            getsem(RESET_SEM);                      // release semaphore
            sleep(1);                               // sleep for 1 second
          } while(1);                               // check queue again
        }; /* rcv */
    }; /* message */
    #endif    /* MESSAGE5_H */
```

The new message class is different from that of Section 10.7.7. This is because the POSIX.1b semaphores are counting semaphores, and each semaphore value can be changed by 1 each time. Conversely, System V semaphore values can be altered by any integral value at a time. With this limitation, the new message class does not support a server and client engaging in private communication while other client processes are blocked by their *semop* call. This results in a new message class whose code is simpler: Each *send* or *receive* request is preceded by a *sem_wait* call to acquire a shared semaphore. The call is finished by a *sem_post* call which releases the acquired semaphore to unblock other processes (server or

client) seeking access to the message queue. Furthermore, the message class now more truly implements System V and POSIX.1b message behavior.

The constructor function of the message class gets an integer key as argument and composes a textual name for the shared memory to be allocated. The shared memory is allocated via the *shm_open* call. It is read-write by the process, and the access permission assigned to user, group, and others is read-write-execute (in case this memory does not exist before the call).

Once a shared memory is allocated, its size is set to the size of the *struct shm_header* via the *ftruncate* call. The *struct shm_header* defines all the data fields for one shared memory region --a shared semaphore variable and a list of message records to store server and client messages.

The shared memory region is mapped to the process virtual address space via *mmap*. The starting address of the mapped memory is determined by the kernel. After the *mmap* call, the file descriptor returned by the *shm_open* call is closed (it is no longer needed). Next, a semaphore is created via the *sem_open* call. The new semaphore is put at the starting address of the shared memory region designated as accessible by multiple processes, and its initial value is set to 1.

Finally, the message list is initialized by setting the message type of each message record in the list to INT_MIN (a large negative number) to indicate that they are unused.

When a message is sent to the message queue via the *message::send* function, the semaphore is first acquired via the *message::getsem* call (which, in turn, calls *sem_wait*). After this call, the process has acquired the semaphore and is allowed to access the message list in the queue. It scans every entry of the message list until it finds the first unused message record (whose message type is INT_MIN). It stores the message data (message text and type) in that record. When this is done, the process releases the semaphore via another *message::getsem* call (which, in turn, calls *sem_post*).

When a process tries to receive a message from the message queue via the *message::rcv* function, the semaphore is first acquired via the *message::getsem* call (which calls *sem_wait*). After this call, the process acquires the semaphore and is allowed to access the message list in the queue. It scans every entry of the message list until it finds a record whose type matches the message type specified by the calling process. It copies that record's message text and type to the input arguments of the function and releases the semaphore (via the *message::getsem* call). Finally, the function returns the message length to the caller. However, if no message in the queue matches the message type specified by a calling process, the function releases the semaphore, puts the process to sleep for 1 second, then repeats the entire message retrieval process. This is to block the calling process until a message arrives at the queue that satisfies process search criteria.

The *message::rmQ* function is called to destroy the shared memory and the semaphore. This is accomplished by calling *sem_destroy* to destroy the semaphore, *shm_unlink* to destroy the shared memory from the system, and finally, *munmap* to unmap the shared memory from the process virtual address space.

The new *message.h* header can be compiled with the client and server programs as shown in Section 10.3.7. The output of the new programs should be the same as that depicted in the Section 10.3.7.

10.10 Summary

This chapter examines UNIX System V.3, V.4, and POSIX.1b interprocess communication methods: messages, semaphores, shared memory, and *mmap*. The syntax of these APIs is explained in detail. Example programs that illustrate their use were also presented. A common drawback of these IPC methods is that there are no standards defined for their use be in intermachine communication. The next chapter will look at the BSD UNIX socket and UNIX System V.3 and V.4 Transport Level Interface (TLI) interprocess communication methods. Sockets and TLI can be used by processes running on different machines or on the same machine to communicate with each other.

CHAPTER 11

Sockets and TLI

The previous chapter examined UNIX System V IPC methods using messages, shared memory, and semaphores. These methods are useful for processes communicating on the same machine, but they do not support processes running on different machines to communicate. The primary reason for this drawback is that message queues, shared memory regions and semaphore sets are identified by integer keys. These are guaranteed to be unique only on individual machines, not across multiple machines. Thus it is impossible for a process running on computer *A* to reference a message queue on machine *B* merely by using the message queue key. POSIX.1b IPC methods eliminate this problem by using textual names for their messages, semaphores, and shared memory. This standard leaves it up to computer vendors to define and interpret these names for IPC to work across their machines.

To support IPC over a local area network (LAN), BSD UNIX 4.2 developed sockets which provide protocol-independent network interface services. Specifically, sockets can run on either TCP (Transport Connect Protocol) or UDP (User Datagram Protocol). A socket can be addressed by a host Internet address and a port number. The address is guaranteed to be unique on the entire Internet, as each machine has an unique address and port number. Thus, two processes running on separate machines may communicate via sockets

Since the introduction, sockets have been widely used in many network-based applications. Sockets are now available in BSD UNIX 4.3, 4.4, and even on UNIX System V.4. However, the implementation of sockets in UNIX System V.4 has some subtle differences from that of BSD UNIX. They are explained in later sections.

Transport Level Interface (TLI) was developed in UNIX System V.3. It was System V's answer to BSD UNIX sockets. Its use and APIs are similar to those of sockets. Furthermore, since TLI was developed based on STREAMS, it supports most transport protocols and is more flexible than sockets. TLI is available on both UNIX System V.3 and V4. Moreover, TLI is called XTI (X/Open Transport Interface) in the X/Open standard.

The following sections examine sockets and TLI APIs and show examples of socket-based and TLI-based applications. Note that sockets and TLI are not defined in POSIX.

11.1 Sockets

Sockets may be *connection-based* (i.e., sender and receiver socket addresses are pre-established before messages are passed between them) or *connectionless* (sender or receiver addresses must be passed along with each message sent from one process to another). There are different socket address formats, depending on the sockets' assigned domain. A domain defines the socket address format and the underlying transport protocol to be used. The common domains assigned to sockets are AF_UNIX (address format is a UNIX path name) and AF_INET (address format is the host name and port number).

Each socket has an assigned type, which determines the manner in which data is transmitted between two sockets. If a socket type is *virtual circuit*, data are transmitted sequentially in a reliable fashion and are nonduplicated. If a socket type is *datagram*, data are transmitted in a nonsequenced and unreliable fashion. Connection-based socket type is usually virtual circuit, whereas connectionless socket type is usually datagram. Datagram sockets are generally faster than are virtual circuit sockets and are used in applications where speed is more important than reliability.

Each socket type supports one or more transport protocols, but there is always a default protocol specified for each socket type on a given UNIX system. The virtual circuit default protocol is TCP and the datagram default protocol is UDP.

Sockets that are used to communicate with each other must be of the same type and belong to the same domain. Furthermore, connection-based sockets communicate in a client/server manner: A server socket is assigned a "well-known" address and is constantly listening for client messages to arrive. A client process sends messages to the server via the server socket's advertised address. It is not necessary to assign an address to client sockets, as usually no process sends messages in this manner.

Connectionless sockets, on the other hand, communicate in a peer-to-peer manner: Each socket is assigned an address, and a process can send messages to other processes via their socket addresses.

The socket APIs are:

Socket APIS	Use
socket	Creates a socket of a given domain, type, and protocol
bind	Assigns a name to a socket
listen	Specifies the number of pending client messages that can be queued for a server socket.
accept	A server socket accepts a connection request from a client socket
connect	A client socket sends a connection request to a server socket
send, sendto	Sends a message to a remote socket
recv, recvfrom	Receives a message from a remote socket
shutdown	Shuts down a socket for read and/or write

The socket APIs' calling sequences that establish server and client virtual circuit connections are shown in Figure 11.1.

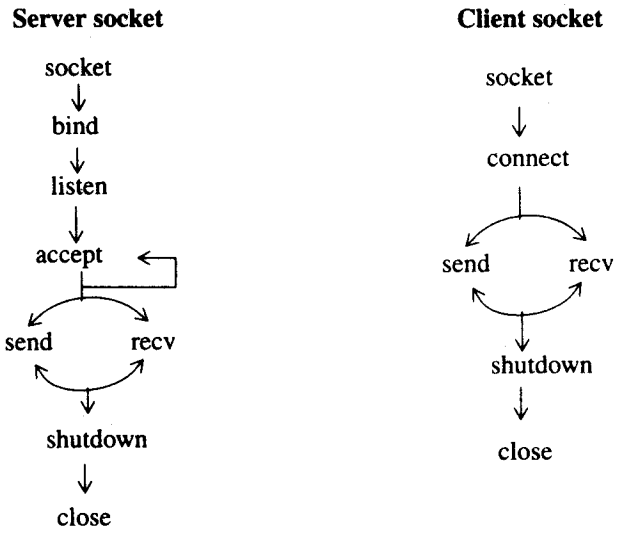

Figure 11.1 Socket API calling sequence for server and client sockets

To make sense of the use of these APIs, imagine that a socket is a telephone set. The *socket* API is to buy a phone from a shop. The *bind* API assigns a phone number to the phone. The *listen* API asks your phone company to set up a maximum allowance for Call Waiting on

your phone The *connect* API is to call someone, using your phone. The *accept* API answers a phone call. The *send* API speaks on the phone. The *recv* API listens to the caller through your phone. Finally, the *shutdown* API hangs up your phone after a call is finished. To discard the "phone," use the *close* API on the socket descriptor returned from a *socket* function call.

One the client's side, the process calls the *socket* function to set up a phone. It calls the *connect* function to dial a server's phone and uses the *send* and *recv* functions to communicate with the server. Once the conversation is over, it calls the *shutdown* function to hang up the phone and the *close* API to discard the "phone."

The calling sequences of socket APIs creating a datagram socket for interprocess communication are shown in Figure 11.2.

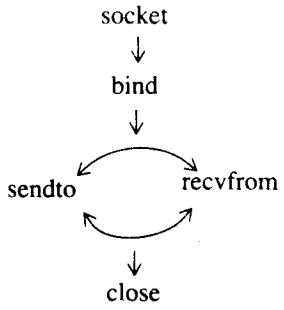

Figure 11.2 Socket API calling sequence to communicate via datagram sockets

The manipulation of datagram sockets is quite simple: A process calls *socket* to create a socket, then calls the *bind* function to assign a name to the socket. After that, the process calls the *sendto* function to send messages to other processes, and each message is tagged with the recipient's socket address. The process also receives messages from other processes via the *recvfrom* function. Each message received is tagged with the sender's socket address so that the process can reply via the same address.

Once the process finishes its IPC, it calls the *close* function to discard the socket. There is no need to call the *shutdown* function, as there is no virtual circuit established with other processes.

The following sections examine the socket APIs' syntax and use in more detail.

11.1.1 socket

The function prototype of the *socket* API is:

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

int     socket (int domain, int type, int protocol);
```

This function creates a socket of the given *domain, type, and protocol.*

The *domain* argument specifies the socket naming convention and the protocol address format. Some popular socket domains are AF_UNIX (UNIX domain) and AF_INET (Defense Advanced Research Project Agency Internet domain).

The *type* argument specifies a socket type. The possible values and their meanings are:

Socket type	Meaning
SOCK_STREAM	Establishes a virtual circuit for communication. Messages are sent in a sequenced, reliable, two-way, connection-based byte stream
SOCK_DGRAM	Establishes a datagram for communication. Messages are sent in a fast (usually connectionless) but unreliable fashion. Datagram messages are not guaranteed to be sent at all
SOCK_SEQPACKET	Provides a sequenced, reliable, two-way, connection-based message transmission with a fixed maximum message length

The *protocol* argument specifies a particular protocol to be used with the socket. Actual value is dependent on the *domain* argument. Usually, this is set to zero, and the kernel will choose an appropriate protocol for the specified domain.

The return value of the function is an integer socket descriptor if it succeeds or -1 if it fails. Note that a socket descriptor is the same as a file descriptor and uses up one file descriptor table slot in the calling process.

11.1.2 bind

The function prototype of the *bind* API is:

```
#include <sys/types.h>
#include <sys/socket.h>
int     bind ( int sid, struct sockaddr* addr_p, int len );
```

This function binds a name to a socket. The socket is referenced by *sid*, which is a socket descriptor, as returned by a *socket* function call. The *addr_p* argument points to a structure that contains the name to be assigned to the socket. The *len* argument specifies the size of the name structure pointed to by the *addr_p* argument.

The actual structure of the object pointed to by the *addr_p* argument is different for different domains. Specifically, for a UNIX domain socket, the name to be bound is a UNIX path name, and the structure of the object pointed to by the *addr_p* is:

```
struct sockaddr
{
   short    sun_family;
   char     sun_path[];
};
```

where the *sun_family* field should be assigned the value of AF_UNIX and the *sun_path* field should contain a UNIX path name. If the *bind* call succeeds, a file having a name specified in the *sun_path* field will be created in the file system. It should be deleted via the *unlink* API, once the socket is no longer needed.

For an Internet domain socket, the name to be bound consists of a machine host name and a port number. The structure of the object pointed to by the *addr_p* is:

```
struct sockaddr_in
{
   short          sin_family;
   u_short        sin_port;
   struct in_addr sin_addr;
};
```

where the *sin_family* field should be assigned the value of AF_INET. The *sin_port* field is a port number, and the *sin_addr* field is a host machine name where the socket resides. The *struct sockaddr_in* is defined in the <netinet/in.h> header.

This function returns a 0 if it succeeds or a -1 if it fails

11.1.3 listen

The function prototype of the *listen* API is:

```
#include <sys/types.h>
#include <sys/socket.h>
int        listen ( int  sid, int size );
```

This is called in a server process to establish a connection-based socket (of type SOCK_STREAM or SOCK_SEQPACKET) for communication.

The *sid* argument is a socket descriptor, as returned by a *socket* function call. This is the socket that the *listen* API acts upon.

The *size* argument specifies the maximum (backlog) number of connection requests that may be queued for the socket. In most UNIX systems, the maximum allowed value for the *size* argument is 5.

The return value of the function is 0 if it succeeds or -1 if it fails.

11.1.4 connect

The function prototype of the *connect* API is:

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

int        connect ( int  sid, struct sockaddr* addr_p, int len );
```

This is called in a client process in requesting a connection to a server socket.

The *sid* argument is a socket descriptor, as returned by a *socket* function call. In BSD 4.2 and 4.3, an unbound socket designated by *sid* is protocol-dependent as to whether it will be given a name. In System V.4, the socket is bound to the name assigned to it by the underlying transport provider.

The *addr_p* argument is a pointer to the address of a *sockaddr*-type object that holds the name of the server socket to be connected. The actual structure of the object is dependent on the domain of the server socket. The possible format is either *struct sockaddr* (for UNIX domain) or *struct sockaddr_in* (for Internet domain).

The *len* argument specifies the size, in number of bytes, of the object pointed to by the *addr_p* argument.

If *sid* designates a stream socket, a virtual circuit connection is established between the client and server sockets. The client's stream sockets may be connected only once. However, if *sid* designates a datagram socket, this establishes a default address for any subsequent *send* function calls via that socket. Datagram sockets may be "connected" multiple times to change their association with different default addresses. Datagram sockets may dissolve their association by connecting to a NULL address.

The function returns a 0 if it succeeds or a -1 if it fails.

11.1.5 accept

The function prototype of the *accept* API is:

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

int     accept ( int sid, struct sockaddr* addr_p, int* len_p );
```

This is called in a server process to establish a connection-based socket connection with a client socket (which calls *connect* to request connection establishment).

The *sid* argument is a socket descriptor, as returned by a *socket* function call. The *addr_p* argument is a pointer to the address of a *sockaddr*-typed object that holds the name of a client socket where the server socket is connected.

The *len_p* argument is initially set to the maximum size of the object pointed to by the *addr_p* argument. On return, it contains the size of the client socket name, as pointed to by the *addr_p* argument.

Note that if either the *addr_p* or the *len_p* argument is NULL, the function does not pass back the client's socket name to the calling process.

The function returns -1 if it fails; otherwise it returns a new socket descriptor that the server process can use to communicate with the client exclusively.

11.1.6 send

The function prototype of the *send* API is:

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

int        send ( int  sid, const char* buf, int len, int flag );
```

This function sends a message, contained in *buf*, of size *len* bytes, to a socket that is connected to the socket, as designated by *sid*.

The *flag* argument is normally assigned a 0 value, but it can also be set to MSG_OOB, which means that the message contained in *buf* should be sent as an out-of-band message.

There are two types of messages that can be transmitted by sockets: regular messages and out-of-band messages. By default, every message sent by a socket is a regular message, unless it is explicitly tagged as out of band. If there is more than one messages of a given type sent from a socket, these are received by another socket in a FIFO order. The recipient socket may select which type of message it wishes to receive. Out-of-band messages should be used as emergency messages only.

If a process uses a connection-based socket or a connectionless socket that has a default recipient address established (via a *connect* function call), it can use either the *send* or *write* APIs to send regular messages via that socket. However, whereas *send* and *sendto* can be used to send zero-length messages, *write* should not be used in such operations. Furthermore, in BSD 4.2 and 4.3, a *write* function call fails if it is used on an unconnected socket. In System V.4, the same function call appears to succeed, but no data is actually sent.

The function returns -1 if it fails; otherwise it returns the number of data bytes sent.

11.1.7 sendto

The function prototype of the *sendto* API is:

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

int   sendto ( int  sid, const char* buf, int len, int flag,
                         struct sockaddr* addr_p, int* len_p );
```

This function is the same as the *send* API, except that the calling process also specifies the address of the recipient socket name via the *addr_p* and *len_p* arguments.

The *sid, buf, len,* and *flag* arguments are the same as that of the *send* API. The *addr_p* is a pointer to the object that contains the name of a recipient socket. The *len_p* contains the number of bytes in the object pointed to by the *addr_p*.

The function returns -1 if it fails; otherwise, it returns the number of data bytes actually sent.

11.1.8 recv

The function prototype of the *recv* API is:

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

int         recv ( int  sid, char* buf, int len, int flag );
```

This function receives a message via a socket designated by *sid*. The message received is copied to *buf*, and the maximum size of *buf* is specified in the *len* argument.

If the MSG_OOB flag is specified in the *flag* argument, an out-of-band message is to be received; otherwise, a regular message is wanted. Furthermore, the MSG_OOB flag may be specified in *flag*, which means that the process wishes to "peek" at the incoming message but does not want to remove it from the socket stream. It can call *recv* again later to receive the message.

If a process uses a connection-based socket or a connectionless socket that has a default recipient address established (via the *bind* API), it can use either the *recv* or the *read* API to receive regular messages via that socket. However, in BSD 4.2 and 4.3 *read* fails if it is used on an unconnected socket. In System V.4, however, *read* returns a zero value on an unconnected socket in blocking mode or returns a -1 value if the socket is nonblocking.

The function returns -1 if it fails: otherwise, it returns the number of data bytes received in *buf*.

11.1.9 recvfrom

The function prototype of the *recvfrom* API is:

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

int  recvfrom (int sid, char* buf, int len, int flag,
                        struct sockaddr* addr_p, int* len_p);
```

This function is the same as the *recv* API, except that the calling process also specifies the *addr_p* and *len_p* arguments to receive the sender name.

The *sid, buf, len,* and *flag* arguments are the same as those of the *recv* API. The *addr_p* is a pointer to the object that contains the name of the sender socket. The *len_p* contains the number of bytes in the object pointed to by the *addr_p*.

The function returns -1 if it fails: otherwise, it returns the number of data bytes actually received.

11.1.10 shutdown

The function prototype of the *shutdown* API is:

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

int  shutdown (int sid, int mode );
```

This function closes the connection between a server and client socket.

The *sid* argument is a socket descriptor, as returned from a *socket* function call. This is the socket where the shutdown should occur.

The *mode* argument specifies the type of shutdown desired. Its possible values and meanings are:

Mode	Meaning
0	Closes the socket for reading. All further reading will return zero bytes (EOF)
1	Closes the socket for writing. Further attempts to send data to the socket will return a -1 failure code.
2	Closes the socket for reading and writing. Further attempts to send data to the socket will return a -1 failure code, and any attempt to read data from the socket will receive a zero value (EOF)

The function returns -1 if it fails, 0 if it succeeds.

11.2 a Stream Socket Example

This section depicts a pair of client/server programs that demonstrates how to set up stream sockets for IPC. The stream sockets used in the example may be UNIX domain sockets or Internet domain sockets. In the latter case, the client and server processes may run on the same machine or on two different machines.

To facilitate the implementation of socket-based applications, a *sock* class is defined to encapsulate the socket APIs from application programs. The advantages of this approach are: (1) it hides the low-level socket addresses set up from application programs. Users of this package manipulate socket addresses in terms of socket names or host names and port numbers only; and (2) the read and write member functions of the sock class are analogous to their equivalent UNIX file APIs. All these features reduce overhead in learning to use sockets, as well as the programming effort of users.

The *sock* class is defined in the *sock.h* header, as follows:

```
#ifndef SOCK_H
#define SOCK_H

#include <iostream.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <memory.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/systeminfo.h>
const int BACKLOG_NUM = 5;

class sock
{
   private:
     int sid;              // socket descriptor
     int domain;           // socket domain
     int socktype;         // socket type
     int rc;               // member function return status code

     /* Build a Internet socket name based on a hostname and a port no */
     int constr_name( struct sockaddr_in& addr, const char* hostnm,
                      int port )
     {
        addr.sin_family = domain;
        if (!hostnm)
          addr.sin_addr.s_addr = INADDR_ANY;
        else {
          struct hostent *hp = gethostbyname(hostnm);
          if (hp==0)    {
            perror("gethostbyname");
            return -1;
          }
          memcpy((char*)&addr.sin_addr,(char*)hp->h_addr,
                      hp->h_length);
        }
        addr.sin_port = htons(port);
        return sizeof(addr);
     };

     /* Build a UNIX domain socket name based on a pathname */
     int constr_name( struct sockaddr& addr, const char* Pathnm )
     {
        addr.sa_family = domain;
        strcpy(addr.sa_data, Pathnm );
```

```
        return sizeof(addr.sa_family) + strlen(Pathnm) + 1;
    };

    /* Convert an IP address to a character string host name */
    char* ip2name( const struct in_addr in )
    {
        u_long laddr;
        if ((int)(laddr = inet_addr(inet_ntoa(in))) == -1) return 0
        struct hostent *hp = gethostbyaddr((char*)&laddr,
                            sizeof (laddr), AF_INET);
        if (!hp) return 0;
        for (char **p = hp->h_addr_list; *p != 0; p++)      {
            if (hp->h_name) return hp->h_name;
        }
        return 0;
    };

public:
    /* sock object constructor function */
    sock( int dom, int type, int protocol=0 ) : domain(dom), socktype(type)
    {
        if ((sid=socket(dom, type,protocol))<0)  perror("socket");
    };

    /* sock object destructor function */
    ~sock()             { shutdown(); close(sid);  }; // discard a socket

    int fd()            { return sid; };              // return a socket's id

    int good()          { return sid >= 0;  };        // check sock obj status

    /* assign a UNIX or an Internet name to a socket */
    int bind( const char* name, int port=-1 )
    {
        if (port == -1)    {                          // UNIX domain socket
            struct sockaddr addr;
            int len = constr_name( addr, name);
            if ((rc= ::bind(sid,&addr,len))<0) perror("bind");
        }
        else  {                                       // Internet domain socket
            struct sockaddr_in addr;
            int len = constr_name( addr, name, port);
            if ((rc= ::bind(sid, (struct sockaddr *)&addr, len))<0  ||
                (rc=getsockname(sid, (struct sockaddr*)&addr, &len))<0)
                perror("bind or getsockname");
```

```
        else cout << "Socket port: " << ntohs(addr.sin_port) << endl;
    }
    /* setup connection backlog threshold for a STREAM socket */
    if (rc!=-1 && socktype!=SOCK_DGRAM &&
                        (rc=listen(sid,BACKLOG_NUM)) < 0)
        perror("listen");
    return rc;
};

/* A server socket accepts a client connection request */
int accept (char *name, int* port_p )
{
    if (!name) return ::accept(sid, 0, 0);
    if (!port_p || *port_p == -1)     {              // UNIX domain socket
        struct sockaddr addr;
        int size = sizeof(addr);
        if ((rc = ::accept(sid, &addr, &size)) >-1)
        strncpy(name,addr.sa_data,size), name[size]='\0';
    }
    else {                                           // Internet domain socket
        struct sockaddr_in addr;
        int size = sizeof (addr);
        if ((rc = ::accept( sid, (struct sockaddr*)&addr, &size)) >-1)  {
            if (name) strcpy(name,ip2name(addr.sin_addr));
            if (port_p) *port_p = ntohs(addr.sin_port);
        }
    }
    return rc;
};

/* A client socket initiates a connection request to a server socket */
int connect( const char* hostnm, int port=-1 )
{
    if (port==-1)  {                                 // UNIX domain socket
        struct sockaddr addr;
        int len = constr_name( addr, hostnm);
        if ((rc= ::connect(sid,&addr,len))<0) perror("bind"):
    }
    else {                                           // Internet domain socket
        struct sockaddr_in addr;
        int len = constr_name( addr, hostnm, port);
        if ((rc= ::connect(sid,(struct sockaddr *)&addr,len))<0)
            perror("bind");
    }
    return rc;
};
```

```
/* writes a message to a connected stream socket */
int write( const char* buf, int len, int flag=0, int nsid=-1 )
{
   return ::send(nsid==-1 ? sid : nsid, buf, len, flag );
};

/* reads a message from a connected stream socket */
int read( char* buf, int len, int flag=0, int nsid=-1 ) // read a msg
{
   return ::recv(nsid==-1 ? sid : nsid, buf, len, flag );
};

/* write to a socket of the given socket name */
int writeto( const char* buf, int len, int flag, const char* name,
                    const int port, int nsid=-1 )
{
   if (port==-1)  {                                  // UNIX domain socket
      struct sockaddr addr;
      int size = constr_name( addr, name);
      return ::sendto(nsid==-1 ? sid : nsid, buf, len, flag, &addr, size );
   }
   else  {                                           // Internet domain socket
      struct sockaddr_in addr;
      char buf1[80];
      if (!name)     {                               // use local host
         if (sysinfo(SI_HOSTNAME,buf1,sizeof buf1)==-1L)
            perror("sysinfo");
         name = buf1;
      }
      int size = constr_name( addr, name, port);
      return ::sendto(nsid==-1 ? sid : nsid, buf, len, flag,
                    (struct sockaddr*)&addr, size );
   }
};

/* Receive a message from a socket */
int readfrom( char* buf, int len, int flag, char* name, int *port_p,
                    int nsid =-1)
{
   if (!port_p || *port_p == -1)     {               // UNIX domain socket
      struct sockaddr addr;
      int size = sizeof(addr);
      if ((rc=::recvfrom(nsid==-1 ? sid : nsid, buf, len, flag, &addr,
                    &size)) >-1 && name)
         strncpy(name,addr.sa_data,rc), name[rc]='\0';
   }
```

```
        else {                                          // Internet domain socket
            struct sockaddr_in addr;
            int size = sizeof (addr);
             if ((rc = ::recvfrom(nsid==-1 ? sid : nsid, buf, len, flag,
                          (struct sockaddr*)&addr, &size)) >-1)     {
               if (name) strcpy(name,ip2name(addr.sin_addr));
               if (port_p) *port_p = ntohs(addr.sin_port);
            }
          }
          return rc;
        };

        /* shut down connection of a socket */
        int shutdown( int mode = 2 )
        {
           return ::shutdown (sid,mode);
        };
};      /* class sock */
#endif
```

The *sock* class is designed to hide the low-level socket API interface from application programs. Thus, an application that wishes to open a UNIX domain socket need only specify a UNIX path name to the *bind* or *connect* member functions. On the other hand, if an application wishes to open an Internet domain socket, it need only to specify the host name and port number. There is no need for an application to manipulate any *struct sockaddr*-typed objects. This saves programming time and reduces errors in setting up socket addresses.

The *sock* member functions are almost one-to-one correspondents with the socket APIs. This makes it easy for users who like to switch between using socket APIs and using the *sock* class objects. Furthermore, the *sock::read, sock::wirte, sock::readfrom,* and *sock::writeto* functions have a *nsid* argument that is assigned when a calling process is a server. It can communicate with a client process via the *nsid* socket descriptor, as obtained from an *sock::accept* function call.

A server program that makes use of the *sock* class to establish a stream socket connection with a client program is shown below:

```
/* sock_stream_srv.C */
#include "sock.h"

const char* MSG2 = "Hello MSG2";
const char* MSG4 = "Hello MSG4";

int main( int argc, char* argv[])
{
   char  buf[80], socknm[80];
```

```
    int  port=-1, nsid, rc;

    if (argc < 2)      {
       cerr << "usage: " << argv[0] << " <socknamelport> [<host>]\n";
       return 1;
    }

    /* check if port no. of a socket name is specified */
    (void)sscanf(argv[1],"%d",&port);

    /* create a stream socket */
    sock sp( port!=-1 ? AF_INET : AF_UNIX, SOCK_STREAM );
    if (!sp.good()) return 1;

    /* Bind a name to the server socket */
    if (sp.bind(port==-1 ? argv[1] : argv[2],port) < 0) return 2;

    /* accept a connection request from a client socket */
    if ((nsid = sp.accept(0, 0)) < 0) return 1;

    /* read MSG1 from a client socket */
    if ((rc=sp.read(buf, sizeof buf, 0, nsid)) < 0) return 5;
    cerr << "server: receive msg: '" << buf << "'\n";

    /* write MSG2 to a client socket */
    if (sp.write(MSG2,strlen(MSG2)+1,0,nsid)<0) return 6;

    /* read MSG3 from a client socket */
    if (sp.readfrom( buf, sizeof buf, 0, socknm, &port, nsid) > 0)
    cerr << "server: recvfrom '" << socknm << "' msg: '" << buf << "'\n";

     /* write MSG4 to a client socket */
    if (write(nsid,MSG4,strlen(MSG4)+1)==-1) return 7;
}
```

The command line argument for this server program may be a UNIX path name for creating a UNIX domain socket. Conversely, it may be a port number and a host name (optional) for creating an Internet domain socket. In the latter case, if a host name is not specified, the host name of the local machine is used instead. Note that in the *sock::bind* function call, if an Internet domain socket is created, the function prints the assigned port number to the standard error port. This is so that a client process can reference that port number when creating a socket to communicate with the server's socket.

After a socket is created and assigned a name, the server process waits for a client connection to be established with its socket. After that, it receives the MSG1 message via the

sock::read function from the client socket, prints the message, and sends the MSG2 message via the *sock::write* function to the client process. The server reads the MSG3 message from the client via the *sock::readfrom* function and replies to the client with the MSG4 message via the *write* API. Finally, the server process terminates, and the socket it allocated is discarded via the *sock::~sock* destructor function.

The client program that communicates with the above server is:

```
/* sock_stream_cls.C */
#include "sock.h"
const char* MSG1 = "Hello MSG1";
const char* MSG3 = "Hello MSG3";
int main( int argc, char* argv[])
{
   int  port=-1, rc;
   if (argc < 2)  {
      cerr << "usage: " << argv[0] << " <socknamelport> [<host>]\n";
      return 1;
   }

   /* check if port number of socket name is specified */
   (void)sscanf(argv[1],"%d",&port);

   /* 'host' may be a socket name or a host name */
   char buf[80], *host= (port==-1) ? argv[1] : argv[2], socknm[80];

   /* create a client socket */
   sock sp( port!=-1 ? AF_INET : AF_UNIX, SOCK_STREAM );
   if (!sp.good()) return 1;

   /* connect to a server socket */
   if (sp.connect(host,port) < 0) return 8;

   /* Send MGS1 to server */
   if (sp.write(MSG1, strlen(MSG1)+1) < 0) return 9;

   /* read MSG2 from server */
   if (sp.read(buf,sizeof buf) < 0) return 10;
   cerr << "client: recv '" << buf << "'\n";

   /* Send MGS3 to server */
   if ((rc=sp.writeto( MSG3, strlen(MSG3)+1, 0, host, port, -1)) < 0)
      return 11;
```

```
    /* read MSG4 from server */
    if ((rc=read(sp.fd(),buf,sizeof buf))==-1) return 12;
    cerr << "client: read msg: '" << buf << "'\n";

    /* shut down socket explicitly */
    sp.shutdown();
}
```

The command line arguments for this client program are the same as those of the server program: a UNIX pathname or a port name followed by an optional host name. The program creates a UNIX domain socket or an Internet domain socket based on the command line arguments.

The client program calls the *sock::connect* function in connecting to the server socket. It writes the MSG1 message to the server via the *sock::write* function, then reads the MSG2 message via the *sock::read* function. After that, the client writes the MSG3 message to the server via the *sock::writeto* function and reads the reply MSG4 message via the *read* API. When the client process terminates, it explicitly shuts down the socket via the *sock::shutdown()* function call.

The above client/server can be run with UNIX domain sockets or Internet domain sockets. The following screen log depicts a sample interaction between server and client processes using UNIX domain sockets. The server socket name is arbitrarily set to SOCK:

```
% CC -o sock_stream_srv sock_stream_srv.C -lsocket -lnsl
% CC -o sock_stream_cls sock_stream_cls.C -lsocket -lnsl
% sock_stream_srv SOCK &
[1] 373
% sock_stream_cls SOCK
server: receive msg: 'Hello MSG1'
client: recv 'Hello MSG2'
server: recvfrom '' msg: 'Hello MSG3'
client: read msg: 'Hello MSG4'
[1]  + Done       sock_stream_srv SOCK
```

Note that in the above, when the server receives MSG3 from the client via the *sock::readfrom* function call, the *sockNm* variable is assigned with a NULL string. This is because the client socket has not been bound with a name in the client process. In a client/server setup, it is generally allowed that only the server socket be named, so that client sockets can be connected to it (and not vice versa).

The following screen log depicts a sample interaction between server and client processes using Internet domain sockets. Here, the machine name that hosts both processes is *fruit*. It lets the system pick the available port number for the client and server sockets:

```
% sock_stream_srv 0 fruit &
[1] 374
Socket port: 32804
% sock_stream_cls 32804 fruit
server: receive msg: 'Hello MSG1'
client: recv 'Hello MSG2'
server: recvfrom 'fruit' msg: 'Hello MSG3'
client: read msg: 'Hello MSG4'
[1] + Done       sock_stream_srv 0 fruit
```

Note that the same client/server programs are run, but with different command line arguments. The client/server processes are run with Internet domain sockets. The output of the programs are the same as in the UNIX domain sockets example. Furthermore, although the above example shows that the server and client processes are executed on the same machine, the result would be the same if run on separate machines. The only difference in running the two programs are: (1) execute the server program on one machine (for example, *fruit*); (2) execute the client program on a remote machine and specify the server socket port number and host name as command line arguments-the output message (i.e., *receive MSG1 and MSG3*) is displayed on its host machine, while the clients' output messages (i.e., *receive MSG2 and MSG4*) are displayed on their host machine.

The datagram socket-based programs also use the *sock.h* header, as above. The first program, *sock_datagram_srv.C* is:

```
#include "sock.h"
const char* MSG2 = "Hello MSG2";
const char* MSG4 = "Hello MSG4";

int main( int argc, char* argv[])
{
   int port = -1, rc;
   char buf[80], socknm[80];

   if (argc < 2)  {
      cerr << "usage: " << argv[0]
           << " <socknamelport> [<remote-host>]\n";
      return 1;
   }
```

```
    /* Check if port number or socket name is specified */
    (void)sscanf( argv[1],"%d",&port);

    /* Create a datagram socket */
    sock sp( port==-1 ? AF_UNIX : AF_INET, SOCK_DGRAM );
    if (!sp.good()) return 1;

    /* assign a name to the socket */
    if (sp.bind(port==-1 ? argv[1] : argv[2],port) < 0) return 2;

    /* read MSG1 from peet */
    if ((rc=sp.readfrom( buf, sizeof buf, 0, socknm, &port, -1)) < 0) return 1;
    cerr << "server: recvfrom from '" << socknm << "' msg: " << buf << endl;

    /* write MSG2 to peer */
    if ((rc= sp.writeto( MSG2, strlen(MSG2)+1, 0, socknm, port, -1)) < 0)
       return 2;

    /* establish a default client address */
    if ((rc = sp.connect(socknm, port)) < 0) return 3;

    /* read MSG3 from peer*/
    if ((rc = sp.read(buf, sizeof buf, 0)) < 0) return 4;
    cerr << "server: receive msg: '" << buf << "'\n";

    /* write MSG4 to peer */
    if (write(sp.fd(),MSG4,strlen(MSG4)+1)<0) return 5;
}
```

The above program is almost the same as the *sock_stream_srv.C*, except that the socket created here is declared to be SOCK_DGRAM (via the *sock::sock* function). The program is given its assigned socket name (for creating UNIX domain socket) or port number and/or host name (for creating an Internet domain socket) at the command line. This socket name is known by the peer that wishes to communicate with it. After the socket is created, the program reads a message from its peer via the *sock::readfrom* function, returning the peer's socket name. The program responds to its peer with the MSG2 message via the *sock::writeto* function. After that, the program establishes a default connection address with the peer socket via the *sock::connect* call. It then uses the *sock::read* function to read the peer's MSG3 message and, finally, replies with the MSG4 message via the *write* API.

The peer program, *sock_datagram_cls.C*, which communicates with the above program is:

```
#include "sock.h"
const char* MSG1 = "Hello MSG1";
```

```
const char* MSG3 = "Hello MSG3";
int main( int argc, char* argv[])
{
   char buf[80], socknm[80];
   int  nlen, port = -1, rc;
   if (argc < 2)  {
         cerr << "usage: " << argv[0]
               << " <sockname | port> [ <remote-host>]\n";
      return 1;
   }
   /* check if port number or socket name is specified */
   (void)sscanf(argv[1],"%d",&port);

   /* create a datagram socket */
   sock sp( port==-1 ? AF_UNIX : AF_INET, SOCK_DGRAM );
   if (!sp.good()) return 1;

   if (port==-1)  {                              // UNIX domain socket
      sprintf(buf,"%s%d", argv[1], getpid());  // construct client socket name
      if (sp.bind(buf,port) < 0) return 2;      // assign name to socket
   }
   else if (sp.bind(0,0) < 0) return 2;         // assign name to socket

   /* write MSG1 to peer */
   if ((rc=sp.writeto( MSG1, strlen(MSG1)+1, 0, port==-1? argv[1] :
argv[2],
            port, -1)) < 0)
      return 6;

   /* read MSG2 from peer */
   if ((rc=sp.readfrom( buf, sizeof buf, 0, socknm, &port, -1)) < 0) return 7;
   cerr << "client: recvfrom '" << socknm << "' msg: " << buf << endl;

   /* establish a default peer socket address */
   if (sp.connect(socknm,port) < 0) return 8;

   /* write MSG3 to peer */
   if (sp.write(MSG3, strlen(MSG3)+1) < 0) return 9;

   /* read MSG4 from peer */
   if ((rc=read(sp.fd(),buf,sizeof buf))==-1) return 10;
   cerr << "client: read msg: " << buf << endl;

   sp.shutdown();
}
```

The command line arguments for the above example are its peer's socket name or port number and/or host name. The program constructs its socket name (for UNIX domain socket) by taking its peer socket name and appending it with its own process ID. It could also let the system assign it a port number (for Internet domain socket) from the host machine. Once the program establishes its own datagram socket (via the *sock::sock* and *sock::bind* function calls), it sends an MSG1 message to the peer socket via the *sock::writeto* function and waits for the reply message MSG2 via the *sock::readfrom* function call. Continuing, the process sets up a default peer socket address via the *sock::connect* function. It then uses the *sock::write* function and the *read* API to send an MSG3 message to its peer process. These functions also allow it to receive the MSG4 message from the peer. Before the program terminates, it uses the *sock::shutdown* function to shut down the socket.

The sample output of these two programs on UNIX domain sockets is:

```
% CC -o sock_datagram_srv sock_datagram_srv.C -I socket -lnsl
% CC -o sock_datagram_cls sock_datagram_cls.C -lsocket -lnsl
% sock_datagram_srv SOCK_DG &
% sock_datagram_cls SOCK_DG
server: recvfrom from 'SOCK_DG572' msg: Hello MSG1
client: recvfrom 'SOCK_DG' msg: Hello MSG2
server: receive msg: 'Hello MSG3'
client: read msg: Hello MSG4
[1] + Done              sock_datagram_srv SOCK_DG
```

Notice in the above output that the first peer socket name is SOCK_DG and the counterpart socket name is SOCK_DG572. The output of these two programs is similar to that of the stream socket client/server programs.

The sample outputs of these two program on Internet domain sockets are:

```
% sock_datagram_srv 0 fruit &
Socket port: 32838
% sock_datagram_cls 32838 fruit
Socket port: 32840
server: recvfrom from 'fruit' msg: Hello MSG1
client: recvfrom 'fruit' msg: Hello MSG2
server: receive msg: 'Hello MSG3'
client: read msg: Hello MSG4
[1] + Done              sock_datagram_srv 0 fruit
```

In the above example, the first peer socket port number is 32838. It runs on a machine called *fruit*. The second process socket port number is 32840. It also runs on *fruit* (although it can be run on a different machine). The two processes interact in exactly the same fashion as when they were using UNIX domain sockets. The output is also identical, except for the printout of socket names.

11.3 Client/Server Message-Handling Example

This section depicts a new version of the client/server example shown in Chapter 10, Section 10.3.7. The new version uses stream sockets to set up a communication channel between the message server and each of its client processes. Furthermore, because the server is connected directly to each client process, each message sent from a client consists of a service command (e.g., LOCAL_TIME, GMT_TIME or QWUIT_CMD, etc.) encoded in character string format. The server also sends the service response to its client in a character string format.

The message server program, *sock_msg_srv.C*, is shown below:

```
#include "sock.h"
#include <sys/times.h>
#include <sys/types.h>
#define MSG1 "Invalid cmd to message server"
typedef enum { LOCAL_TIME, GMT_TIME, QUIT_CMD,
                                        ILLEGAL_CMD } CMDS;
/* process a client's commands */
int process_cmd (int fd )
{
   char   buf[80];
   time_t  tim;
   char*   cptr;

   /* read commands from a client until EOF or QUIT_CMD */
   while (read(fd, buf, sizeof buf) > 0)     {
      int    cmd = ILLEGAL_CMD;
      (void)sscanf(buf,"%d",&cmd);
      switch (cmd)        {
         case LOCAL_TIME:
            tim = time(0);
            cptr = ctime(&tim);
            write(fd, cptr, strlen(cptr)+1);
            break;
         case GMT_TIME:
            tim = time(0);
```

```
            cptr = asctime(gmtime(&tim));
            write(fd, cptr, strlen(cptr)+1);
            break;
         case QUIT_CMD:
            return cmd;
         default:
            write(fd, MSG1, sizeof MSG1);
      }
   }
   return 0;
}

int main( int argc, char* argv[])
{
   char  buf[80], socknm[80];
   int  port=-1, nsid, rc;
   fd_set  select_set;
   struct timeval timeRec;

   if (argc < 2)      {
      cerr << "usage: " << argv[0] << " <socknamelport> [<host>]\n";
      return 1;
   }

   /* check if port no. of a socket name is specified */
   (void)sscanf(argv[1],"%d",&port);

   /* create a stream socket */
   sock sp( port!=-1 ? AF_INET : AF_UNIX, SOCK_STREAM );
   if (!sp.good()) return 1;

   /* Bind a name to the server socket */
   if (sp.bind(port==-1 ? argv[1] : argv[2],port) < 0) return 2;

   for (;;)      {                        // Poll for client connections
      timeRec.tv_sec = 1;                    // polling time-out after one second
      timeRec.tv_usec= 0;
      FD_ZERO( &select_set );
      FD_SET( sp.fd(), &select_set );

       /* wait for time-out or a read event occurs for socket */
      rc = select(FD_SETSIZE, &select_set, 0, 0, &timeRec );
      if (rc > 0 && FD_ISSET(sp.fd(), &select_set))      {
         /* accept a connection request from a client socket */
         if ((nsid = sp.accept(0, 0)) < 0) return 1;
```

```
            /* process commands */
            if (process_cmd(nsid)==QUIT_CMD) break;

            close(nsid); /* recycle file descriptor */
          }
          /* else do something else */
        }
        sp.shutdown();
        return 0;
    }
```

The invocation syntax of a message server program is the same as that of the *sock_stream_srv.C* example. It allocates a stream socket and binds a name to it in the same manner. However, once the socket is set up, the server uses the *select* API to poll read events occurring on the socket. Each *select* polling time out after 1 second so that the server may be programmed to do something else, if needed.

When a client sends a service command to the server, it calls the *process_cmd* function to process all the service commands initiated by the client. The *process_cmd* function returns when the client sends either QUIT_CMD or a noninteger service command to the server. In either case, the server closes the *nsid* file descriptor that references the socket, as created by the *sock::accept* function call. After that, the server either continues to poll the stream socket for connection to another client or simply shuts down the stream socket and terminates itself.

The client program, *sock_msg_cls.C*, is:

```
#include "sock.h"
#define QUIT_CMD 2

int main( int argc, char* argv[])
{
   if (argc < 2)  {
      cerr << "usage: " << argv[0] << " <sockname | port> [<host>]\n";
       return 1;
   }
   int  port=-1, rc;

   /* check if port number of socket name is spceified */
   (void)sscanf(argv[1],"%d",&port);

   /* 'host' may be a socket name or a host name */
   char buf[80], *host= (port==-1) ? argv[1] : argv[2], socknm[80];
```

```
    /* create a client socket */
    sock sp( port!=-1 ? AF_INET : AF_UNIX, SOCK_STREAM );
    if (!sp.good()) return 1;

    /* connect to a server socket */
    if (sp.connect(host,port) < 0) return 8;

    /* Send cmds 0 -> 2 to server */
    for (int cmd=0; cmd < 3; cmd++)    {
       /* compose a command to server */
       sprintf(buf,"%d",cmd);
       if (sp.write(buf,strlen(buf)+1) < 0) return 9;

       /* exit the loop if QUIT_CMD */
       if (cmd==QUIT_CMD) break;

       /* read reply from server */
       if (sp.read(buf,sizeof buf) < 0) return 10;
        cerr << "client: recv '" << buf << "'\n";
    }
    sp.shutdown();
    return 0;
}
```

The client program's invocation syntax is the same as that of the *sock_stream_cls.C* example. It allocates a stream socket and connects it to the server socket in the same manner. However, once the socket is set up, the client sends the following service messages to the server: (1) tell local date/time; (2) tell GMT date/times; (3) quit command; and (4) solicit a warning message from the server.

For each service command sent by a client (except the QUIT_CMD), the client collects the server response and prints the results to the standard output. The client program terminates after it has sent the QUIT_CMD to the server.

The client and server programs are compiled as follows:

```
% CC -o sock_msg_srv sock_msg_srv.C -I socket -lnsl
% CC -o sock_msg_cls sock_msg_cls.C -lsocket -lnsl
```

A sample console log of the server/client program interaction is:

```
% sock_msg_srv 0 fruit &
[1] 441
```

```
Socket port: 32792
% sock_msg_cls 32792 fruit
client: recv 'Sun Feb 12 00:41:25 1995'
client: recv 'Sun Feb 12 08:41:25 1995'
[1] + Done              sock_msg_srv 0 fruit
```

In the above, the client and server processes communicate using Internet domain stream sockets. The client collects only the server responses found on the LOCAL_TIME and UTC_TIME commands. Again, by using the Internet domain sockets, the above server and client processes can be run on separate machines.

The following console log shows the same client/server processes interacting in the same manner as the above, but using only UNIX domain sockets:

```
% sock_msg_srv SOCK_MSG &
[1] 446
% sock_msg_cls SOCK_MSG
client: recv 'Sun Feb 12 00:42:38 1995'
client: recv 'Sun Feb 12 08:42:38 1995'
[1] + Done              sock_msg_srv SOCK_MSG
```

11.4 TLI

Transport Level Interface was developed in UNIX System V.3 as an alternative to sockets. TLI is more flexible than sockets and is based on STREAM, which supports most transport protocols. TLI creates transport end points whose behavior and function are similar to those of sockets. For example, TLI transport end points may communicate with each other in either a connection-based or a connectionless mode. Furthermore, processes running on different machines or on the same machine may communicate via their TLI transport end points. Both sockets and TLI transport end points are designated by file descriptors. Thus, a process may set the O_NONBLOCK flag on these file descriptors, either when they are assigned or via the *fcntl* function. This renders the execution of corresponding socket or TLI end point operations as nonblocking.

A TLI end point cannot communicate with a socket. When a TLI transport end point is created, a user must specify that a transport protocol be bound to that end point. In sockets, however, a user is not required to specify a transport protocol in creating a socket. The *socket* API will choose a default protocol based on the socket type. In addition to these differences, the address assigned to a socket for intramachine communication is different from that of a TLI end point. A TLI end point is assigned an integer port number for communicating with other transport end points on the same machine, whereas a socket is assigned a UNIX path

name for the same function. For communication over the Internet, a TLI transport end point is assigned a machine name and a service port number (similar to that of *socket*). Finally, for connection-based communication, client sockets are usually not assigned addresses unless the client processes explicitly assign them. On the other hand, TLI transport end points are always assigned addresses, either by users or by the underlying transport protocol.

There is an almost one-to-one correspondence between TLI APIs and those of sockets. This makes it easy for socket-based applications to be converted to TLI. The next section gives an overview of TLI APIs and their comparison to socket APIs. The subsequent sections describe the syntax and use of TLI APIs in more detail. The final two sections depict two examples of TLI application: One is a rewrite of the client/server examples shown in Section 11.2, using TLI transport end point instead of sockets. The other example shows how to use TLI transport end points to send datagram messages.

11.4.1 TLI APIs

TLI system functions and their use are:

TLI API	Function
t_open	Creates a transport end point and specifies an underlying transport protocol
t_bind	Assigns a name to a transport end point. For a connection-based end point, this also specifies the maximum number of backlog connection requests allowed
t_listen	Waits for a connection request from a client transport end point
t_accept	Accepts a connection request from a remote transport end point
t_connect	Sends a connection request to a server transport end point
t_snd	For a connection-based transport end point only. Sends a message to a connected end point
t_rcv	For a connection-based transport end point only. Receives a message from a connected end point
t_sndudata	Sends a datagram message to a transport end point with a given address
t_rcvudata	Receives a datagram message and a sender address from a transport end point
t_snddis	Aborts a connection
t_rcvdis	Returns an abort connection indication with a reason

TLI API	Function
t_sndrel	Sends a request to a transport end point for orderly release of a connection
t_rcvrel	Returns an orderly connection release indication from a connected transport end point
t_error	Similar to *perror*, but prints TLI-specific error diagnostics if a TLI function call fails
t_alloc	Allocates dynamic memory for a transport end point
t_free	Deallocates dynamic memory of a transport end point
t_close	Closes a transport end point descriptor

The calling sequences of the above TLI APIs in establishing a server and client virtual circuit connection are shown in Figure 11.3.

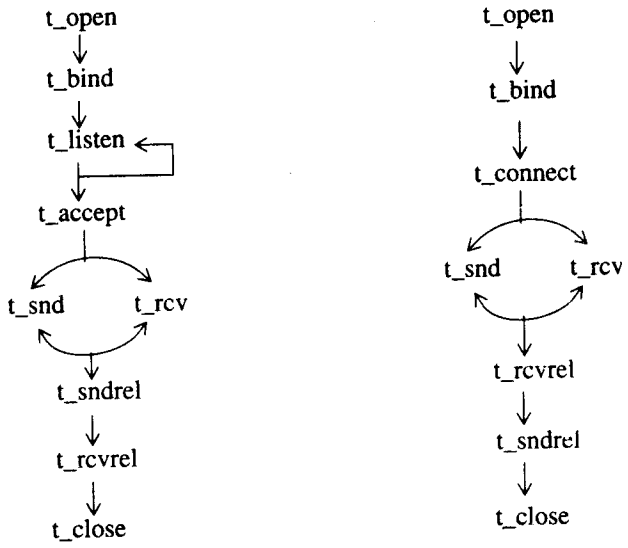

Figure 11.3 TLI API calling sequence for server and client transport end points

Notice the similarity of the calling sequences in the above to those depicted in Figure 11-1. Specifically, the *t_open* API is similar to the *socket* API, the *t_bind* is like the *bind* API, and the t_*snd* and *t_rcv* functions are similar to the *send* and *rcv* APIs, respectively. However, the *t_listen* API is not the same as the *listen* API. This is because the *listen* API sets the maximum number of pending connection requests allowed for a socket. For a transport end point,

this information is set in the *t_bind* API instead. The *t_listen* API is actually like the socket's *accept* API, in that it causes the calling process to wait for a client connection request. Once a connection request is received, the *t_listen* function returns with the client transport end point address. The server may call either *t_accept* to accept the connection or the *t_snddis* function to abort the connection.

Like the *accept* API, the *t_accept* API may generate a new descriptor that designates a private transport end point for the server to communicate with a client process. The server process can continue to monitor further connection requests from other client processes via the original descriptor, as obtained from the *t_open* call.

Once a server and client process finish their communication, either one of them can call the *t_sndrel* function to send a disconnect notification request to their counterpart. The other process calls the *t_rcvrel* function to receive that notification and responds with a *sndrel* function call. Once the first process receives the reply notification via the *t_rcvrel* function, the transport end points are disconnected, and both processes may call the *t_close* function to dispose their transport end points.

As an alternative to the *t_sndrel* and *t_rcvrel* functions, a server and client process may call the *t_snddis* and *t_rcvdis* functions to abort a transport connection. The *t_snddis* function is abortive, and any data remaining to be sent over the transport end points are discarded immediately. The *t_sndrel* function, on the other hand, is nonabortive, and any remaining data to be passed between the transport connection are delivered before the connection is destroyed. All transport protocols used with TLI must support the *t_snddis* and *t_rcvdis* functions, but they are not required to support the *t_sndrel* and *t_rcvrel* functions.

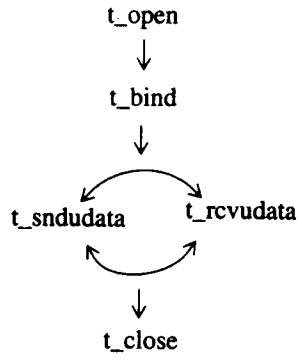

Figure 11.4 TLI API calling sequence to create a datagram transport end point

The TLI API calling sequence that creates a datagram transport end point is shown in Figure 11.4.

The manipulation of datagram transport end points is quite simple: A process calls *t_open* to create a transport end point, then calls the *t_bind* function to assign a name to the end point. After that, the process calls the *t_sndudata* function to send messages to other processes, and each message is tagged with a recipient transport end point address. The process also receives messages from other processes via the *t_rcvudata* function. Each message received is tagged with a sender transport end point address, so that the process can reply accordingly.

Once the process finishes its interprocess communication, it calls the *t_close* to discard the transport end point descriptor. There is no need to call the *t_snddis* or *t_sndrel* function here, as there is no virtual circuit established with other processes.

The following few sections examine TLI API syntax and use in more detail.

11.4.2 t_open

The function prototype of the *t_open* API is:

```
#include <tiuser.h>
#include <fcntl.h>

int     t_open (char* path, int aflag, struct t_info* info);
```

This function creates a transport end point that uses a transport provider, as specified by *path*. The actual value to *path* may be the path name of a device file that designates the transport provider. For example, the */dev/ticlts* file designates a UDP transport provider, and the */dev/ticotsord* file designates a virtual circuit transport provider.

The *aflag* argument specifies the access mode of the transport end point by the calling process. Its value is defined in the <fcntl.h> header and is usually O_RDWR, which means that the calling process may send and receive messages via the transport end point. In addition, the O_NONBLOCK flag may also be specified with the O_RDWR flag, which renders the transport end point to perform nonblocking operations.

The *info* argument returns default characteristics of the underlying transport provider. This information is usually ignored and the actual value for *info* may be 0. However, if a user is interested in examining these default characteristics, the user can define a *struct t_info* typed variable and pass the address of that variable as actual value to the *info* argument.

When the function returns, he can look at the content of that variable. One interesting piece of data is contained in the *info->servtype* field. The possible values and meanings are:

***servtype* value**	**Meaning**
T_COTS	The transport provider supports virtual circuit connection but not the orderly disconnection request
T_COTSORD	The transport provider supports virtual circuit connection and the orderly disconnection request
T_CLTS	The transport provider supports the passing of datagram messages

The above constants are declared in the <tiuser.h> header. The return value of the function is -1 if it fails. If it succeeds, the value is a descriptor that designates a transport end point created by the function.

The following statements create a connection-based transport end point that supports orderly release of connection. Furthermore, the transport end point operations are performed in nonblocking mode, and the transport default characteristic information is not wanted. The transport end point descriptor is assigned to the variable *fd*:

```
int fd = t_open( "/dev/ticotsord", O_RDWRIO_NONBLOCK,0);
if (fd == -1) t_error("t_open");
```

The following statements create a datagram transport end point. The transport end point operations are to be performed in the normal blocking mode, and the transport default characteristics are returned via the *info* variable. The transport end point descriptor is assigned to the variable *fd*:

```
struct t_info info;
int fd = t_open( "/dev/ticlts", O_RDWR, &info);
if (fd==-1) t_error("t_open");
```

Some transport end points have system-defined addresses, as specified in the */etc/services* file. For example, the following two lines in a */etc/services* file define two end points:

```
# /etc/services
test            4045/tcp
utst1           5001/udp
```

In the above, the first transport end point service name is *test* and it uses *TCP* as its transport provider. Thus, it is a connection-based end point. The second end point service name is *utst1* and it uses *UDP* as its transport provider. This makes it a datagram end point.

Given the above definitions in the */etc/services* file, one can determine the address of a transport end point address and its transport provider device file name as follows:

```
struct nd_hostserv hostserv;
struct netconfig   *nconf;
struct nd_addrlist *addr;
void               *hp;
int                type = NC_TPI_COTS_ORD;
if ((hp=setnetpath()) == 0)
{
      perror("Can't init network");
      exit(1);
}
hostserv.h_host = "fruit";          // assume the machine host name
hostserv.h_serv = "test";           // transport's service name
while ((nconf=getnetpath(hp)) != 0)
{
      if (nconf->nc_semantics == type
         && netdir_getbyname(nconf, &hostserv, &addr)==0)
         break;
}
endnetpath(hp);

if (nconf == 0 )
      cerr << "No transport found for service: 'test'\n";
else if ((tid=t_open(nconf->nc_device, O_RDWR, 0)) < 0)
      t_error("t_open fails");
else cerr << "transport end point's address is specified in addr\n";
```

In the above code segment, the machine name and the transport service name are assumed to be *fruit* and *test*, respectively. This information is stored in the *hostserv* variable. The *setnetpath*, *getnetpath*, and *endnetpath* functions are used to retrieve each entry in the */etc/netconfig* file, where each entry contains a transport provider type and a corresponding device file name. In the above example, a transport provider of type NC_TPI_COTS_ORD (connection-based and support for orderly connection release) is sought. For each transport provider whose type matches the criteria, the *nconf* and *hostserv* variables are passed to the *netdir_getbyname* function. This finds the address of: (1) a transport end point on the specified machine (*fruit* in this example); and (2) the given service name (*test*). This function uses the specified transport provider.

The transport end point address is returned via the *addr* variable. This *addr* variable is used later on in the *t_bind* (server process) or *t_connect* (client process) function calls.

The above code can be modified as follows to get a datagram transport end point address. It is also the device file path name of the transport provider for *utst1* service:

* Assign the value NC_TPI_CLTS to the *type* variable instead of NC_TPI_COTS_ORD
* Assign the value *utst1* to the *hostserv.h_serv* data field

11.4.3 t_bind

The function prototype of the *t_bind* API is:

```
#include <tiuser.h>
int     t_bind (int fd, struct t_bind* inaddr, struct t_bind* outaddr);
```

This function binds an address (or a name) to a transport end point, as designated by the *fd* argument. The actual value of *fd* is obtained from a *t_open* call.

The *inaddr* argument contains the address assigned to the transport end point. Its actual value may be NULL, which means that the transport end point underlying the transport provider is to assign the necessary address.

The *struct t_bind* is declared as:

```
struct t_bind
{
    struct netbuf   addr;
    unsigned        qlen;
};
```

where the *qlen* field specifies the maximum number of connection requests that may be pending for the transport end point. This is set to a nonzero value for a server transport end point and a zero value for all other uses. The *addr* field contains the address to be assigned to the transport end point.

The *struct netbuf* is declared as:

```
struct netbuf
{
        unsigned int    maxlen;
        unsigned int    len;
        char*           buf;
};
```

where the *len* value specifies the number of characters in *buf* that contains the transport end point address. The *maxlen* field is a *don't-care* for *inaddr*.

The *outaddr* argument returns the actual address assigned to the transport end point by the underlying transport provider. This may be different from that specified in *inaddr*. If the transport provider fails to bind the address specified in *inaddr* to the end point, it will bind a different address to it instead. The actual value to *outaddr* may be NULL, which means that the calling process does not care what address is assigned to the transport end point. This is usually the case for a connection-based client transport end point.

If the actual value for *outaddr* is the address of a *struct t_bind* type variable, then, as input, the *outaddr->buf* is an address of a buffer defined by a calling process, and the *outaddr->maxlen* field specifies the maximum size of the *outaddr->buf* buffer. On return, the *outaddr->len* field contains the number of characters in the *outaddr->buf*, which stores the transport end point address, as assigned by the transport provider.

The function returns a -1 if it fails, a 0 if it succeeds.

The following sample code binds an address returned by the *netdir_getbyname* function (see the example in the last section) to a transport end point designated by *fd*. Furthermore, the end point may accept up to five client connection requests at any one time. Notice that *t_alloc* is used to allocate dynamic storage for the *struct t_bind*-typed object (as pointed to by *bind*). This guarantees that all the fields in the object are initialized properly.

```
struct t_bind *bind = t_alloc (fd, T_BIND, T_ALL);
if (!bind)
   t_error("t_alloc fails for T_BIND");
else {
   bind->qlen = 5;
   bind->addr = *(addr->n_addrs);
   if (t_bind(fd, bind, bind) < 0) t_error("t_bind");
}
```

A transport end point may be bound to an integer-type address, which is useful if the end point communicates with other transport end points on the same machine only. The following code binds an address of 2 to a transport end point designated by *fd*:

```
struct t_bind *bind = t_alloc (fd, T_BIND, T_ALL);
if (bind)    {
   bind->qlen = 5;
   bind->addr.len = sizeof(int);
   *(int*)bind->addr.buf = 2;
   if (t_bind(fd, bind, bind) < 0) t_error("t_bind");
} else t_error("t_alloc fails for T_BIND");
```

11.4.4 t_listen

The function prototype of the *t_listen* API is:

```
#include <tiuser.h>

int      t_listen (int fd, struct t_call* call );
```

This function waits for a client connection request to arrive at a transport end point as designated by *fd*. The client's transport end point address is returned via the *call* argument.

By default this function blocks the calling process until a client connection request is received. However, if *fd* is specified to be nonblocking (using the O_NONBLOCK flag in the *t_open* call or set via the *fcntl* function), the *t_listen* function returns immediately if no client connection request is detected by the function. The *t_errno* global variable will be set to TNODATA.

The *struct t_call* is declared as:

```
struct t_call
{
      struct netbuf    addr;
      struct netbuf    opt;
      struct netbuf    udata;
      int              sequence;
};
```