```
    /* create a server handle to wait for RPC calls to the set_alarm func*/
    svcp = new RPC_svc( ACLKPRCG, ACLKVER, "netpath");

    /* register the RPC function and wait for RPC calls */
    if (svcp->run_func( ACLKFUNC, set_alarm )) ;

    return 1; /* the server process should never get here */
}
```

The server program starts by creating an *RPC_svc* object to initiate the *set_alarm* RPC function. The program number, version number, and procedure number of this *set_alarm* RPC function are ACLKPROG, ACLKVER, and ACLKFUNC, respectively. The server calls the *RPC_svc::run* function to wait for client RPC requests to arrive.

When a client RPC request arrives, the *set_alarm* RPC function is called. The *set_alarm* function, in turn, calls the *RPC_svc::getargs* function to extract the RPC call-back information. This information is stored in the *argRec* variable. After the *RPC_svc::getargs* call succeeds, the server calls the *RPC_svc::reply* to send a dummy reply to the client. This finishes the RPC call, and the client can now go on to do something else.

After the *RPC_svc::reply* call, the server forks a child process to deal with the client, and the parent (the parent process) returns to the polling loop to wait for other client RPC requests.

The child process calls the *alarm* API to set up a SIGALRM signal to be sent to it after the client-specified alarm clock period elapses. It also calls the *signal* API to catch the SIGALRM signal when it is delivered to the child process. Finally, the child process calls the *pause* API to suspend its execution until the SIGALRM signal arrives.

When the SIGALRM signal is delivered to the child process, the *call_client* function is called. This function sets up an *RPC_cls* object to connect with the client RPC call-back function and sends the remaining alarm clock time (which should be zero) as argument to the client RPC function. After the RPC call completes, the function calls the *exit* function to terminate the child process.

The client program for this example is *aclk_cls.C*:

```
#include <netconfig.h>
#include "aclock.h"
#include "RPC.h"

#define CLNTPROGNUM 0x20000105
RPC_svc *svcp = 0;
```

```
/* client's RPC call-back function */
int callback( SVCXPRT* xtrp )
{
   u_long timv;
   /* get server's alarm remaining time */
   if (svcp->getargs( xtrp, (xdrproc_t)xdr_u_long, (caddr_t)&timv)
            !=RPC_SUCCESS)
   {
      cerr << "client: get alarm time fails\n";
      return -1;
   }
   cerr << "client: alarm time left is: " << timv << endl;

   /* send a dummy reply to server */
   if (svcp->reply(xtrp, (xdrproc_t)xdr_void, 0)!=RPC_SUCCESS) {
      cerr << "client: send reply failed\n";
      return -2;
   }
   /* do other work, then terminates the client process */
   exit(0);
}

/* register a call back with an RPC server */
int register_callback( char* local_host, char* svc_host, u_long alarm_time)
{
   /* tell remote server the process's host name, prog no, vers. no,
      func. no, and the alarm time
   */
   struct arg_rec argRec;
   argRec.hostname = local_host;
   argRec.prognum  = svcp->progno();
   argRec.versnum  = CLNTVERNUM;
   argRec.funcnum  = CLNTFUNCNUM;
   argRec.atime    = alarm_time;

   /* setup a client object to connect to the RPC server */
   RPC_cls clnt( svc_host, ACLKPROG, ACLKVER, "netpath");
   if (!clnt.good()) return 1;

   /* call the server's RPC function (set_alarm) */
   if (clnt.call( ACLKFUNC, (xdrproc_t)xdr_arg_rec, (caddr_t)&argRec,
            (xdrproc_t)xdr_void, (caddr_t)0 ) !=RPC_SUCCESS)
      return 2;
   cerr << "client: " << getpid() <<": RPC call done\n";
   return 0;
}
```

```
/* client main function */
int main (int argc, char* argv[])
{
   if (argc!=4)  {
      cerr << "usage: " << argv[0] << " <local-host> <svc-host> "
           << "<transport>\n";
      return 1;
   }

   /* create a server object to receieve call back from a remote server */
   if (!(svcp= new RPC_svc( CLNTPROGNUM, CLNTVERNUM, argv[3])))
      return 2;

   /* define the callback function */
   svcp->add_func( CLNTFUNCNUM, callback );

   /* register the callback with a remote server */
   if (register_callback( argv[1], argv[2], 10)) return 3;

   /* do other work here .... */

   svcp->run(); /* wait for alarm to expire */
   return 0;
}
```

The client process begins by creating an *RPC_svc* object to register its call-back RPC function, *callback*, with the *rpcbind* (via the *svc_create* API). The call-back function program, version, and procedure numbers are CLNTPROGNUM, CLNTVERNUM, and CLNTFUNCNUM, respectively. After the *RPC_svc* object is created, the client calls the *register_callback* function to inform the alarm server of the alarm clock period and the call-back information. Upon return of the *register_callback* function, the client proceeds to do other work. It then calls the *RPC_svc::run* function at the end to wait for the server call-back to arrive.

The *register_callback* function creates an *RPC_cls* object to connect to the alarm server RPC function. It calls the server RPC function and passes a record of data containing: (1) the client's host machine name; (2) the call-back function program, version, and procedure numbers; and (3) the alarm clock period. This information is needed by the server to call the client RPC function when the alarm period expires.

The client's *callback* RPC function is called by the alarm server when the alarm period expires. The *callback* function calls the *RPC_svc::getargs* to extract the server argument (the remaining alarm clock time). It then sends a dummy reply to the server via the *RPC_svc::reply* function. Finally, the function calls *exit* to terminate the client process.

The XDR conversion functions of the above program are contained in the *ack_xdr.c*:

```
#include "aclock.h"
bool_t xdr_name_t(register XDR *xdrs, name_t *objp)
{
   register long *buf;
   return (!xdr_string(xdrs, objp, MAXNLEN)) ? FALSE : TRUE;
}

bool_t xdr_arg_rec(register XDR *xdrs, arg_rec *objp)
{
   register long *buf;
   if  (!xdr_name_t( xdrs, &objp->hostname ))    return (FALSE);
   if  (!xdr_u_long( xdrs, &objp->prognum ))     return (FALSE);
   if  (!xdr_u_long( xdrs, &objp->versnum ))     return (FALSE);
   if  (!xdr_u_long( xdrs, &objp->funcnum ))     return (FALSE);
   if  (!xdr_u_long( xdrs, &objp->atime ))       return (FALSE);
   return (TRUE);
}
```

The XDR functions translate the client's call-back information, as specified in a *struct arg_rec* record, which is sent to the alarm server in the *set_time* RPC call.

The above programs are compiled and run as shown below. It is assumed that the server is running on a machine called *saturn*, while the client process is running on a machine called *fruit*:

On machine *saturn*:

```
saturn %  CC -DSYSV4 -o aclk_svc aclk_svc.C RPC.C aclk_xdr.c \
          -lsocket -lnsl
saturn %  aclk_svc &
```

On machine *fruit*:

```
fruit %     CC -DSYSV4 -o aclk_cls aclk_cls.C RPC.C aclk_xdr.c \
            -lsocket -lnsl
fruit %     aclk_cls fruit saturn netpath
client: 1567: RPC call done
client: alarm time left is: 0
```

12.10 Transient RPC Program Number

In the above example, the client RPC function has a predefined program number, version number, and procedure number. This restricts the client process in running more than one process at any one time on the network. One solution to this restriction is to create different versions of the client program, each with a different assigned RPC program number. However, this makes it hard to maintain the programs. A better solution is for each client process to generate a transient RPC program number at run time, so that multiple instances of the client processes may be active concurrently on a LAN (the server can differentiate them by their unique RPC program numbers). Note that the following discussion is based on the UNIX System V release 4 version of RPC. Not all UNIX systems support the transient port number generation method.

The RPC program numbers 0x40000000 -- 0x5fffffff are reserved for transient use. This allows any process to dynamically contact the *rpcbind* daemon to reserve one or more of these values as its RPC program number(s); as long as the number is not being used by other processes,. When the process terminates, the transient RPC program numbers that it claimed are made available again for use by other processes.

The *RPC_svc::gen_progNum* static function can be used to allocate a transient RPC program number. This function calls the *rpcb_set* API, for each number in the 0x40000000 and 0x5fffffff range, to query the *rpcbind* daemon whether a number is currently assigned to any process. The function stops at the first lowest available transient program number, and the *rpcb_set* registers that program number and the specified version number with the *rpcbind*.

The function prototype of the *rpcb_set* API is:

```
#include <rpc/rpc.h>

bool_t rpcb_set (const u_long prognum, const u_long versnum,
                 const struct netconfig* netconf, const struct netbuf* addr);
```

The *prognum* and *versnum* arguments are the requested RPC program and version numbers to be assigned to the calling process. The *netconf* argument contains the transport information of the calling process. Finally, the *addr* argument is the network address of the calling process.

The function returns TRUE if it succeeds, and the requested program and version numbers are registered with *rpcbind* for the process with the specified address and transport. This function returns FALSE if it fails.

The *netconf* and *addr* arguments of the *RPC_svc::gen_progNum* are a bit tricky to get,

particularly if a server handle is created via the *svc_create* API. However, if one uses the *svc_tli_create* API instead to create a server handle, the *netconf* and *addr* values of the server are readily available.

To accommodate the use of transient program numbers, the following overloaded *RPC_svc::RPC_svc* constructor function can be added to the *RPC_svc* class:

```
RPC_svc( int fd, char* transport, u_long progno, u_long versno )
{
   rc = 0;                                        /* assume failure status */

   struct netconfig *nconf = getnetconfigent(transport);
   if (!nconf) {
      cerr << "invalid transport: " << transport << endl;
      return;
   }

   /* create a server handle */
   SVCXPRT *xprt =svc_tli_create( fd, nconf, 0, 0, 0);
   if (!xprt) {
      cerr << "create server handle fails\n";
      return;
   }

   if (!progno)   {              /* generate a transient one */
      progno = gen_progNum( versno, nconf, &xprt->xp_ltaddr);
      nconf = 0;                 /* tell svc_reg don't talk to rpcbind */
   }

   if (svc_reg(xprt, progno, versno, dispatch, nconf)==FALSE)
      cerr << "register prognum failed\n";
   else {
      prgnum = progno, vernum = versno;
      rc = 1;
   }
   freenetconfigent( nconf );
};
```

In the above overloaded constructor function, the *getnetconfigent* function, is called to return a pointer to a *struct netconf* data record that contains the network transport information for the given function argument. Once the transport handle contained in the *nconf* variable is obtained, the *svc_tli_create* API is called to create a server handle and a default address for the given transport. The *netconf* variable is freed via the *freenetconfigent* function.

The function prototype of the *svc_tli_create* API is:

```
#include <rpc/rpc.h>
SVCXPRT*  svc_tli_create (const int fd, const struct netconfig* netconf,
        const struct t_bind* baddr, const u_int sendsz, const u_int recvsz);
```

The *fd* argument is a file descriptor referencing a transport device file. If its value is specified as RPC_ANYFD, a transport device file determined by the *netconf* argument is used. The address assigned to the server is determined by the *baddr* argument if it is not NULL; otherwise, a default address chosen by a given transport is used. The *sendsz* and *recvsz* arguments specify the desired sending and receiving buffer size for the server handle. If their values are zero, the default buffer sizes determined by the transport will be used.

The *svc_tli_create* API returns NULL if it fails; otherwise, it returns a server handle.

If the *svc_tli_create* succeeds and a given *progno* argument value is 0, the *RPC_svc::gen_progNum* function is called to generate a transient program number. After that, the *svc_reg* API is called to associate an RPC program number and version number with their dispatch function.

The function prototype of the *svc_reg* API is:

```
#include <rpc/rpc.h>
int     svc_reg (const SVCXPRT* xprt, const u_long prognum,
                 const u_long versnum, const void (*diaptch)(...),
                 const struct netconfig* netconf);
```

The *xprt* argument is a server handle. The *prognum* and *versnum* arguments are the RPC program and version numbers associated with a dispatch function, as given in the *dispatch* argument. The *netconf* argument specifies a network transport that can be used to register the RPC function and dispatch function with the *rpcbind* daemon. If the *netconf* argument is NULL, no such registration is needed.

In the *RPC_svc* constructor function, the *svc_reg* is called with the *netconf* argument set to zero (if the *RPC_svc::gen_progNum* has been called). This is because the *RPC_svc::gen_progNum* function automatically registers the RPC function with the *rpcbind* daemon.

The *svc_reg* function returns FALSE if it fails; otherwise, it returns TRUE, and an RPC server is set up successfully.

The *aclk_cls.C* program can be rewritten to use the overloaded *RPC_svc::RPC_svc* constructor function, which creates a transient program number. The new *aclk_cls2.C* program is:

```
#include <netconfig.h>
#include "aclock.h"
#include "RPC.h"
#define CLNTPROGNUM 0x20000105
RPC_svc *svcp = 0;

/* client's RPC call-back function */
int callback( SVCXPRT* xtrp )
{
   u_long timv;
   /* get server's alarm remaining time */
   if (svcp->getargs( xtrp, (xdrproc_t)xdr_u_long, (caddr_t)&timv)
            !=RPC_SUCCESS) {
      cerr << "client: get alarm time fails\n";
      return -1;
   }
   cerr << "client: alarm time left is: " << timv << endl;

   /* send a dummy reply to server */
   if (svcp->reply(xtrp, (xdrproc_t)xdr_void, 0)!=RPC_SUCCESS) {
      cerr << "client: send reply failed\n";
      return -2;
   }
   /* do other work, then terminates the client process */
   exit(0);
}

/* register a call back with an RPC server */
int register_callback( char* local_host, char* svc_host, u_long alarm_time)
{
   /* tell remote server the process's hostname, prog no, vers. no, func. no,
     . and the alarm time
   */
   struct arg_rec argRec;
   argRec.hostname = local_host;
   argRec.prognum  = svcp->progno();
   argRec.versnum  = CLNTVERNUM;
   argRec.funcnum  = CLNTFUNCNUM;
```

```
    argRec.atime    = alarm_time;

    /* setup a client object to connect to the RPC server */
    RPC_cls clnt( svc_host, ACLKPROG, ACLKVER, "netpath");
    if (!clnt.good()) return 1;

    /* call the server's RPC function (set_alarm) */
    if (clnt.call( ACLKFUNC, (xdrproc_t)xdr_arg_rec, (caddr_t)&argRec,
            (xdrproc_t)xdr_void, (caddr_t)0 ) !=RPC_SUCCESS)
       return 2;

    cerr << "client: " << getpid() <<": RPC call done\n";
    return 0;
}

/* client main function */
int main (int argc, char* argv[])
{
    if (argc!=4) {
       cerr << "usage: " << argv[0] << " <local-host> <svc-host> "
            << "<transport>\n";
       return 1;
    }

    /* create a server object to receieve call back from a remote server */
    if (!(svcp= new RPC_svc( RPC_ANYFD, argv[3], 0, CLNTVERNUM )))
       return 2;

    /* define the callback function */
    svcp->add_func( CLNTFUNCNUM, callback );

    /* register the callback with a remote server */
    if (register_callback( argv[1], argv[2], 10)) return 3;

    /* do other work here .... */

    svcp->run(); /* wait for alarm to expire */
    return 0;
}
```

Notice that the only difference between the new *aclk_cls.C* and the one depicted in Section 12.9 is on one line: the creation of the *RPC_svc* handle via the *new* operator in the main function.

The program can be compiled and run as in Section 12.9. The output of the old and new

client/server programs is the same.

12.11 RPC Services Using Inetd

RPC servers are commonly daemon processes that run continuously, waiting for RPC calls from their clients. This has the disadvantage that system resources allocated for these processes (e.g., the Process Table slots) cannot be used by other processes, even when the daemons are idle. To improve system resource utilization, port monitors such as *inetd* may be used to monitor network addresses for RPC services, while the RPC servers are not run at all. However, when an RPC request arrives, the port monitor spawns an RPC server to respond to that request, and the server terminates itself after the service is performed. Thus, system resources are allocated to an RPC server only for the duration when it is responding to a client request.

Most commercial UNIX systems use *inetd* as the port monitor. *inetd* is started at system boot, and it consults the */etc/inetd.conf* file for network addresses to monitor. Specifically, each entry of the */etc/inetd.conf* file has the following syntax:

<service> <transport> <protocol> <wait> <uid> <program> <arg>

The various fields in the entry state that if a request for <service> arrives, *inetd* should execute <program> and supply <arg> as its argument. The effective user ID of the executed process should be <uid>, and it uses <transport>/<protocol> to communicate with its client process. The commonly used <transport> and corresponding <protocol> values are:

Transport	Protocol
stream	tcp
dgram	udp

For socket-based services, the <wait> field should be specified as *nowait* for connection-based (tcp) transport and *wait* for connectionless (udp) transport. For TLI-based services, the <wait> field is commonly set as *wait*.

The port address for a <service> is defined in the */etc/services* file as:

<service> <port>/<protocol>

For example, given the following entry in a */etc/inetd.conf* file:

login stream tcp nowait root /etc/in.rlogind in.rlogind

when a remote user attempts to login to the local host, *inetd* should execute the */etc/in.rlogind* program as *root*. The *rlogin* process will use TCP/IP transport. The port address that the *rlogin* process uses is 513, as stated in the */etc/services* file:

```
logir                513/tcp
```

To instruct *inetd* to monitor a particular RPC request, the */etc/inetd.conf* file should contain an entry for the RPC server as;

```
<prog_num>/<vers_num>    <transport>  <protocol>   <wait> \
        <uid>          <program>          <arg>
```

Here, <prog_num> and <vers_num> are the RPC server's program and version numbers. The other fields can be the same as before for ONC-based RPC. However, in UNIX System V.4, the <transport> may be *tli* if the RPC server handle is created based on TLI, and the <protocol> values may be *rpc/tcp*, *rpc/udp*, or *rpc/**. The protocol value of *rpc/** means that the server may use any TLI-supported transport.

For example, the directory listing program as shown in Section 12.7.4 uses 0x200100 and 1 as the RPC program and version numbers. To make *inetd* support the service, the following entry should be added to the */etc/inetd.conf* file (here the executable file of the RPC server is assumed to be */proj/scan_svc3*):

For ONC:

```
# 536871168 is same as 0x20000100
536871168/1   stream   tcp   wait    root   /proj/scan_svc3   scan_svc3
```

For System V.4:

```
536871168/1   tli   rpc/*   wait    root   /proj/scan_svc3   scan_svc3
```

In addition to configuring *inetd*, the RPC server should create its *RPC_svc* (the *RPC_svc* class is described in Section 12.5) handle using the constructor:

```
RPC_svc::RPC_svc( int fd, char* transport, unsigned long progno,
                  unsigned long versnum);
```

Furthermore, the *fd* argument value for the *RPC_svc* constructor should be zero, as this is assigned by *inetd* to correspond to the incoming RPC request. In the above example, the directory listing server should create its *RPC_svc* handle as:

```
RPC_svc *svcp = new RPC_svc( 0, "tcp", 0x20000100, 1 );
```

When the *RPC_svc::RPC_svc* constructor is called, it creates the *RPC_svc* handle using the following RPC APIs:

For ONC:

- Call *svctcp_create* (for stream transport) or *svcudp_create* (for datagram transport) to create a server handle
- Call *svc_reg* to register a dispatch function to be invoked when an RPC call arrives

For System V.4:

- Call *getnetconfigent* to obtain a *struct netconfig* object for the transport (*tcp* or *udp*) desired
- Call *svc_tli_create* to create a server handle
- Call *svc_reg* to register a dispatch function to be invoked when an RPC call arrives

Readers who are interested in the detail calling sequence of the above APIs should consult the *RPC_svc* constructor code as shown in Section 12.5.

The above works are all the necessary changes for using *inetd* to monitoring RPC service requests. The rest of the server code is the same as if it were not using *inetd*. Furthermore, there are no change at all in the XDR functions and the client programs that make RPC calls.

As the final example, the remote directory listing program shown in Section 12.7.4 is rewritten below so that the server uses *inetd* to monitor RPC requests on its behalf. The client program (*scan_cls2.C*) and the XDR functions (*scan_xdr.c*) are the same as in Section 12.7.4 and, thus, are not depicted again. The modified server program is *scan_svc3*.C and is shown below:

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <string.h>
#include <malloc.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/resource.h>
#include "scan2.h"
```

```
#include "RPC.h"
static   RPC_svc *svcp = 0;
static   int    work_in_progrss = 0;
static   int    ttl = 60;                    /* time-to-live: 60 seconds */
/* The RPC function */
int scandir( SVCXPRT* xtrp )
{
   DIR           *dirp;
   struct dirent  *d;
   infolist       nl, *nlp;
   struct stat    statv;
   static res     res;
   argPtr         darg = 0;
   work_in_progress = 1;              /* process not killed by alarm */

   /* Get function argument from a client */
   if (svcp->getargs( xtrp, (xdrproc_t)xdr_argPtr,
                  (caddr_t)&darg)!=RPC_SUCCESS)
      return -1;

   /* start scaning the requested directory */
   if (!(dirp = opendir(darg->dir_name)))     {
      res.errno = errno;
      (void)svcp->reply(xtrp, (xdrproc_t)xdr_res, (caddr_t)&res);
      return -2;
   }

   /* free memoryallocated from a previous RPC call */
   xdr_free((xdrproc_t)xdr_res, (char*)&res);

   /* store files' informaton to res as the return values */
   nlp = &res.res_u.list;
   while (d=readdir(dirp))   {
      nl = *nlp = (infolist)malloc(sizeof(struct dirinfo));
      nl->name = strdup(d->d_name);
      nlp = &nl->next;
      if (darg->lflag)   {
         char pathnm[256];
         sprintf(pathnm,"%s/%s",darg->dir_name,d->d_name);
         if (!stat(pathnm,&statv))     {
            nl->uid = statv.st_uid;
            nl->modtime = statv.st_mtime;
         }
      }
   }
   *nlp = 0;
```

```
    res.errno = 0;
    closedir(dirp);

    /* Send directory listing to client */
    int    rc = svcp->reply(xtrp, (xdrproc_t)xdr_res,(caddr_t)&res);
    work_in_progess = 0;              /* process can be killed by alarm */
    return rc;
}

/* signal handling routine */
static void done ( int signo )
{
    if (!work_in_progress) exit(0);
    signal( SIGALRM, (void(*)(int)) done );
    alarm( ttl );
}

int main(int argc, char* argv[])
{
    struct rlimit rls;
    switch (fork()) {
        case 0:    break;
        case -1:   perror( "fork");
        default:   return errno;
    }

    /* close all I/O streams, except descriptor 0 */
    rls.rlim_max = 0;
    getrlimit(RLIMIT_NOFILE, &rls);
    if (rls.rlim_max == 0)  {
          fprintf( fp, "getrlimit failed\n");
          return 1;
    }

    for (int i = 1; i < rls.rlim_max; i++)    (void) close(i);

    /* all output messages redirected to the system console */
    int fd = open("/dev/console", 2);
    (void) dup2(fd, 1);
    (void) dup2(fd, 2);
    setsid();

    /* Now create the RPC server handle */
    svcp = new RPC_svc( 0, "tcp", SCANPROG, SCANVER );
    if (!svcp || !svcp->good()) {
          fprintf( stderr, "create RPC_svc object failed\n" );
```

```
        exit( 1 );
    }

    svcp->add_proc( SCANDIR, scandir );

    /* terminate daemon after alive for 60 seconds */
    signal( SIGALRM, done );
    alarm( ttl );

    svcp->run();

    return 0;
}
```

In the above program, the server creates an *RPC_svc* handle for the file descriptor zero and the transport used is *tcp*. It registers the *scandir* function to be callable by clients, then sets up the SIGALRM signal to be sent to itself. The latter is done because *inetd* does not spawn a new process for a service request if there is already a server process running (this is done to avoid creating too many redundant processes). Thus, it is common practice for a server spawned by *inetd* to remain blocked for a set period of time after it has serviced a request, so that it may catch the next service call that comes shortly.

After registering the *done* function as the signal handler for SIGALRM, the server program calls *RPC_svc::run*. This causes the *scandir* function to be called immediate, as there is already a pending request. When the *scandir* function returns, the server is blocked in the *RPC_svc::run* function waiting for the next RPC call. The server lives, at most, 60 seconds, unless a new RPC call arrives before the time is up. If that happens the *done* function will reset the alarm clock so that the process can run for another 60 seconds. The *work_in_progress* global variable is set whenever the *scandir* function is called, and it is reset when the *scandir* function returns. This variable is used by the *done* function to decide whether to terminate the process or to restart the alarm clock.

The client and server programs are compiled on a UNIX System V.4 system, as follows (on ONC systems, compile them without the -DSYSV4 option):

```
%   CC -c scan2_xdr.c RPC.C
%   CC -DSYSV4 -o scan_svc3 scan_svc3.C scan2_xdr.o \
            RPC.o -lsocket -lnsl
%   CC -DSYSV4 -o scan_cls2 scan_cls2.C scan2_xdr.o RPC.o \
            -lsocket -lnsl
```

The */etc/inetd.conf* file entry for the *scan_svc3* server is (using System V.4 format):

```
536871168/1tli    rpc/* wait    root    /proj/scan_svc3 scan_svc3
```

Finally, the sample run of the client and server programs is shown below. The server program is resided on a machine called *fruit*, while the client program may be run on any machine that is connected to *fruit*. Only the client program needs to be started manually, while the server program is executed by *inetd*:

```
%    scan_cls2 fruit .
....
....
...scan_cls2.C
...scan_svc2.C
...RPC.C
...RPC.h
...scan2_xdr.c
...scan2.h
...scan_svc2
...scan_cls2
Prog 536871168 (version 1) is alive
```

12.12 Summary

This chapter describes three methods of creating RPC programs: (1) using the system-supplied RPC library functions; (2) using the *rpcgen* compiler to create custom RPC function and client main programs; and (3) using the RPC classes to create full custom client and server RPC programs.

Of the three approaches, the last one, which uses RPC classes, is most flexible, in that users have complete control over the content of the client and server programs, and they can also control the transport properties used by their applications. Furthermore, the RPC classes encapsulate most of the low-level RPC API interface. Thus, programming effort is not much more time-consuming than when using the *rpcgen* compiler.

Finally, numerous examples are depicted in the chapter to illustrate various RPC programming techniques. These include RPC broadcast, asynchronous call-backs (from clients to servers), authentication, transient RPC program number generation, and using *inetd* to monitor RPC requests. Users may use these example programs as starting points and may modify them to create their own RPC-based applications.

CHAPTER 13

Multithreaded Programming

A thread is a piece of program code executed in a serial fashion. Most UNIX applications are single-threaded programs, as each of them executes only one piece of program code at any one time. For example, a single-threaded process may get a command from a user, execute the command, display the results to the user, then wait for a next command. While the process is executing a command, the user must wait for it to finish before entering subsequent commands.

A multithreaded program, on the other hand, can have several pieces of its code executed "concurrently" at any one time. Each piece of the code is executed by one thread of control. Thus, in the previous example, if the process were multithreaded, the user could enter commands immediately, one after the other, and the process executed all commands concurrently.

Multithreaded programming can be used to develop concurrent applications. These applications can be run on any multiprocessor systems and make good use of hardware resources. Specifically, if a multithreaded application runs on a system with M processors, each of its threads may be run on a separate processor simultaneously. Thus, the performance of the application may be improved by N times, where N is the maximum number of processors available at any one time, and N is less than or equal to M.

If a multithreaded application is run on a uniprocessor system, its performance may still be improved. For example, if one of its threads is blocked in a system call (e.g., waiting for data to be transferred to a tape device), another thread can be run on the processor right away. Thus, the overall execution time of the application is reduced.

In addition to the above benefits, multithreaded programming is also a good compliment to object-oriented programming. This is because each object-oriented application consists of a collection of objects interacting with each other to perform tasks. Each of these objects is an independent entity and can be executed by a thread and run in parallel with other objects. This results in significantly improvement in performance for these applications. For example, in a multithreaded, object-oriented, window system, each menu, button, text field, and scrolled window may be executed by a thread. Thus, any of these window objects may be activated, one right after the other, without waiting for other objects to finish their execution.This makes the entire GUI application more responsive ("interactive") to users than its single-threaded counterpart.

Threads differ from child processes created by the *fork* API in the following ways:

- Threads may be managed by either user-level library functions or the operating system kernel. Child processes as created by the *fork* system call are managed by the operating system kernel. In general, threads is more efficient, and require much less kernel attention, to create and manage than do child processes
- All threads in a process share the same data and code segments. A child process has its own copy of virtual address space that is separate from its parent process. Thus, threads use much less system resources than do child processes
- If a thread calls the *exit* or *exec* function, it terminates all the threads in the same process. If a child process calls the *exit* or *exec* function, its parent process is not affected
- If a thread modifies a global variable in a process, the changes are visible to other threads in the same process. Thus, some synchronization methods are needed for threads accessing shared data. This problem does not exist between child and parent processes

All in all, the benefits of using multithreaded programming are:

- It improves process throughput and responsiveness to users
- It allows a process to make use of any available multiprocessor hardware on any system it is run on
- It allows programmers to structure their code into independently executable units and maximize concurrency
- Threads reduce the need to use *fork* to create child processes, thus improving each process performance (less context switching). Also, less kernel involvement is needed in managing their execution
- It is the natural choice for multiprocessing, object-oriented, applications to enhance their performance

The drawback of multithreaded programming is users must be careful in designing thread synchronization in each program. This is to ensure that threads do not accidentally mis-read/write shared data or destroy their process via the *exit* or *exec* system call.

Multithreaded programming has been in development since the mid-1980s. Different versions of multithreaded programming interfaces were offered by different UNIX vendors. The POSIX committee has developed a set of multithreaded APIs, which is now part of the POSIX.1c standard. This chapter describes both the Sun Microsystems Solaris 2.x and POSIX1.c multithreaded APIs, with emphasis on the Sun APIs. The Sun multithreaded APIs are described because the POSIX.1c standard is new and not many UNIX vendors are supporting it yet. On the other hand, the Sun multithreaded APIs have been available for application programmers for quite some time. Furthermore, the Sun multithreaded APIs closely resemblance those of the POSIX.1c APIs, and there is almost a one-to-one correspondence of the Sun multithreaded APIs to the POSIX.1c APIs. Thus applications which are based on the Sun multithreaded APIs can be easily converted to the POSIX.1c standard.

13.1 Thread Structure and Uses

A thread consists of the following data structures:

- A thread ID
- A run-time stack
- A set of registers (e.g., program counter and stack pointer)
- A signal mask
- A schedule priority
- A thread-specific storage

A thread is created by the *thr_create* function (or the *pthread_create* in POSIX.1c). Each thread is assigned a thread ID that is unique among all threads in a process. A newly created thread inherits the process signal mask and is assigned a run-time stack, a schedule priority, and a set of registers. The run-time stack and registers (program counter and stack pointer) enable the thread to run independently of other threads. The schedule priority is used to schedule the execution of threads. A thread may change its inherited signal mask and allocate dynamic storage to store its own private data.

When a thread is created, it is assigned a function to execute. The thread terminates when that assigned function returns or when the thread calls the *thr_exit* (*pthread_exit* in POSIX.1c) function. When the first thread is created in a process, two threads are actually created: One to execute a specified function and the other to carry on the execution of the process. The latter thread terminates when the *main* function returns or when it calls the *thr_exit* function.

All threads in a process share the same data and code segments. If a thread writes data to global variables in the process, those changes are seen by other threads immediately. Furthermore, if a thread calls the *exit* or *exec* API, the containing process and all its threads are terminated. Thus, a terminating thread that does not want to destroy its containing process should call the *thr_exit* function instead.

A thread can change its signal mask via the *thr_sigsetmask* (*pthread_sigmask* in POSIX.1c) function. If a signal is delivered to a process, then any thread which has not masked the signal will receive it. A thread can send signals to other threads in the same process via the *thr_kill* (*pthread_kill* in POSIX.1c) function, but it cannot send signals to specific threads in a different process, as thread IDs are not unique among different processes. A thread may use the *signal* or *sigaction* API to set up per-thread signal handling.

A thread is assigned an integer thread schedule priority number. The larger the priority number the more frequently a thread is scheduled to run. A thread schedule priority number can be inquired and changed by the *thr_getprio* and *thr_setprio* (*pthread_attr_getschedparam* and *pthread_attr_setschedparam* in POSIX.1c) functions, respectively. In addition to these, a thread can deliberately yield its execution to other threads of the same priority via the *thr_yield* (*sched_yield* in POSIX.1c) function. Moreover, a thread can wait for the termination of another thread and get its exit status code with the *thr_join* (*pthread_join* in POSIX.1c) function.

Note that in Sun a thread can suspend and resume execution of another thread via the *thr_suspend* and *thr_continue* functions. Furthermore, if a function is executed by multiple threads and uses static or global variables to which data is assigned and used on a per-thread basis, it needs to create thread-specific storage to store these actual data for each thread. A thread-specific storage is allocated via the *thr_keycreate, thr_setspecific and thr_getspecific* functions.

13.2 Threads and Lightweight Processes

The Sun thread library functions create objects called lightweight processes (LWPs), which are scheduled by the kernel for execution. LWPs are like virtual processors in that the thread library functions schedule threads in a process to be bound to LWPs and be executed. If a thread bound to an LWP gets suspended (e.g., via the *thr_yield* or *thr_suspend* function), the LWP is assigned to bind to another thread and executes that thread's function. If an LWP makes a system call on behalf of a thread, it remains bound to that thread until the system call returns. If all LWPs are bound to threads and are blocked at system calls, the thread library functions create new LWPs to bind unbound threads that are waiting for execution. This ensures that a process is constantly executing. Finally, if there are more LWPs than threads existing in a process, the thread library functions remove idle LWPs to conserve system resources.

Most threads are unbound and can be scheduled to bind to any available LWP. However, a process may create one or more threads that are permanently bound to LWPs. These are called *bound threads*. They are used primarily if they need:

- To be scheduled by the kernel for real-time processing
- To have their own alternate signal stacks
- To have their own alarms and timer

The relationships of threads, LWPs, and hardware processors are depicted in Figure 13.1.

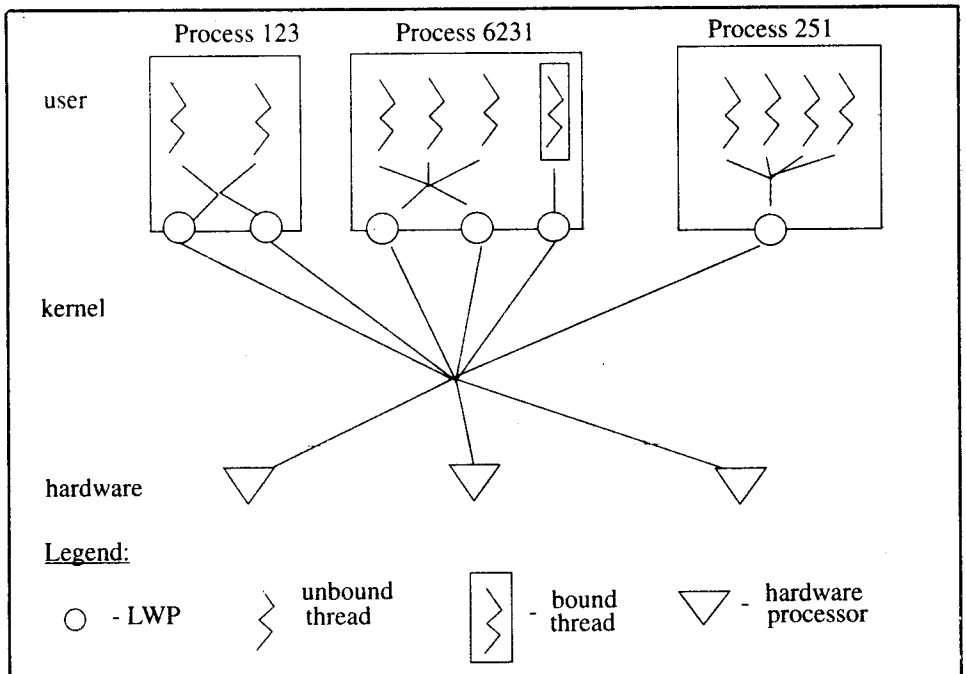

Figure 13.1 Threads, LWPs, and hardware processor scheduling relationship

In Figure 13-1, the process 123 has two unbound threads that are scheduled on two LWPs. The process 6231 has three unbound threads that are scheduled on two LWPs and one bound thread that is executed by another LWP. The process 251 has four unbound threads that are scheduled on one LWP. The unbound threads on each process are scheduled by the thread library functions to be bound and run on LWPs in that process. The LWPs of all processes are, in turn, scheduled by the kernel to run on the three existing hardware processors.

In POSIX.1c, threads have an attribute known as *scheduling contention scope*. If a thread contention scope attribute is set to PTHREAD_SCOPE_PROCESS, the thread is man-

aged by user-level library functions and is an "unbound" thread. All threads with this same attribute share processor resources that are available to their containing process. On the other hand, if a thread contention scope attribute is set to PTHREAD_SCOPE_SYSTEM, then the thread is managed by the operating system kernel and is considered "bound". POSIX.1c does not specify how a "bound" thread should be handled by the kernel.

13.3 Sun Thread APIs

This section describes only the Sun thread APIs. The POSIX.1c thread APIs are described in the Section 13.4. This is done so as to avoid confusing readers of the two different sets of APIs. The various sub-sections in Section 13.4 list the corresponding APIs between Sun to the POSIX.1c standard. These can be used for converting multithreaded applications from Sun to the POSIX.1c standard.

To use the Sun thread APIs, users should do the following:

- Include the <thread.h> header in their programs
- Compile and link their programs with the *-lthread* option. If the *-lC* option is specified also, then the *-lthread* option should be specified before the *-lC* switch. For example, the following compiles a multithreaded C++ program call *x.C*:

```
%    CC x.C -o x -lthread -lC
```

Unless otherwise stated, most of the thread APIs depicted below return a 0 value if they succeed or a -1 value if they fail. In case they fail, *perror* may be called to print error diagnostic messages.

13.3.1 thr_create

The *thr_create* function prototype is:

```
#include <thread.h>
int         thr_create (void* stackp, size_t stack_size, void* (*funcp)(void*),
                        void* argp, long flags, thread_t* tid_p);
```

The function creates a new thread to execute a function whose address is given in the *funcp* argument. The function specified in *funcp* should accept one *void**-typed input argument and return a *void** data. The actual argument to be passed to the *funcp* function, when the new thread starts executing, is specified in the *argp* argument.

The *stackp* and *stack_size* arguments contain the address of a user-defined memory region and its size in number of bytes, respectively. This memory is used as the new thread run-time stack. If the *stackp* value is NULL, the function allocates a stack region of *stack_size* bytes. If the *stack_size* value is 0, the function uses a system default value that is one megabyte of virtual memory. Users rarely need to supply their own memory region for a thread stack. Thus, the normal values to *stackp* and *stack_size* arguments are NULL and zero, respectively.

In addition to the above, the *flags* argument value may be zero, meaning that no special attributes are assigned to the new thread. On the other hand, the *flags* value may consist of one or more of the following bit-flags:

***flags* value**	**Meaning**
THR_DETACHED	Creates a detached thread. This means that when the thread terminates, all its resources and assigned thread ID can be reused for another thread. No thread should wait for it (via the *thr_join* function)
THR_SUSPENDED	Suspends the execution of the new thread until another thread calls the *thr_continue* function to enable it to execute
THR_BOUND	Creates a permanently bound thread
THR_NEW_LWP	Creates a new LWP along with the new thread
THR_DAEMON	Makes the new thread a daemon thread. Normally a multithreaded process terminates when all its threads are terminated. However, if the process contains one or more daemon threads, the process terminates immediately when all nondaemon threads are terminated.

The new thread ID is returned via the *tid_p* argument. If the actual value of the *tid_p* argument is assigned NULL, no thread ID is returned. The thread ID data type is *thread_t*.

The *thr_create* function may fail if there is not enough system memory to create a new thread, the *stackp* argument contains an invalid address, or the *stack_size* argument value is nonzero and less than the system-imposed minimum limit. The system-imposed minimum stack size limit for a thread is obtained from the *thr_min_stack* function:

```
size_t    thr_min_stack ( void );
```

A thread can find out its thread ID via the *thr_self* function:

```
thread_t    thr_self    ( void );
```

The following sample code creates a new detached and bound thread to execute a function called *do_it*. The argument passed to *do_it* d is the address of the *pInt* variable. The new thread's ID is assigned to the *tid* variable, and its stack is allocated by the function with the system default size:

```
extern void* do_it (void* ptr);
int         *pInt;
thread_t    tid;
if (thr_create( 0, 0, do_it, (void*)&pInt,
                        THR_DETACHED|THR_BOUND, &tid) < 0)
    perror("thr_create");
```

13.3.2 thr_suspend, thr_continue

The function prototypes of the *thr_suspend* and *thr_continue* functions are:

```
#include <thread.h>

int         thr_suspend ( thread_t tid );
int         thr_continue ( thread_t tid );
```

The *thr_suspend* function suspends the execution of a thread whose ID is designated by the *tid* argument value.

The *thr_continue* function resumes the execution of a thread whose ID is designated by the *tid* argument value.

These functions may fail if the *tid* value is invalid.

13.3.3 thr_exit, thr_join

The function prototypes of the *thr_exit* and *thr_join* functions are:

```
#include <thread.h>

int        thr_exit ( void* statusp);
int        thr_join ( thread_t tid, thread_t* dead_tidp, void** statusp);
```

The *thr_exit* function terminates a thread. The actual argument value to the *statusp* argument is the address of a static variable that contains the exit status code of the terminating thread. If no other thread is expected to retrieve the terminating thread exit status code (e.g., the thread is detached), the *statusp* argument value may be specified as NULL.

The *thr_join* function is called to wait for the termination of a nondetached thread. If the *tid* argument value is zero, the function waits for any thread to terminate. The *dead_tidp* and *statusp* argument values are addresses of variables that hold the terminated thread's ID and exit status value, respectively. The actual values to these arguments may be NULL, which means those data are unwanted.

The following example waits for all nondetached threads in a process to terminate, then terminates the current thread:

```
status int  *rc, rval=0;
thread_t   tid;
while (!thr_join(0, &tid, &rc))
   cout << "thread: " << (int)tid << ", exits, rc=" << (*rc) << endl;
thr_exit( (void*)&rval );
```

13.3.4 thr_sigsetmask, thr_kill

Each thread has its own signal mask which is inherited from its creating thread. A thread may modify its signal mask via the *thr_sigsetmask* API. When a signal is delivered to a multithreaded process, a thread in that process which has the signal unblocked will receive the signal. If there are multiple threads in the process with the signal unblocked, the system will arbitrarily pick any one of them as the target for the signal. Thus to simplify the implementation of multithreaded programs, it is recommended that a process elects a dedicated thread to handle one or more signals for the entire process, while other threads in the same process block those signals.

In addition to the above, a thread may also send a signal to another thread in the same process via the *thr_kill* API. The *thr_sigsetmask* and *thr_kill* function prototypes are:

```
#include <thread.h>
#include <signal.h>

int     thr_sigsetmask ( int mode, sigset_t *sigsetp, sigset_t *oldsetp);
int     thr_kill ( thread_t tid, int signum);
```

The *thr_sigsetmask* function sets the signal mask of a calling thread. The *sigsetp* argument contains one or more of the signal numbers applied to the calling thread. The *mode* argument specifies how the signal(s) specified in the *sigsetp* argument is to be used. The possible values of the *mode* argument are declared in the <signal.h> header. These values and their meanings are:

***mode* value**	**Meaning**
SIG_BLOCK	Adds signals contained in the *sigsetp* argument to the thread signal mask
SIG_UNBLOCK	Removes signals contained in the *sigsetp* argument from the thread signal mask
SIG_SETMASK	Replaces the thread signal mask with the signal(s) specified in the *sigsetp* argument

If the *sigsetp* argument value is NULL, the *mode* argument value is ignored.

The *oldsetp* argument value should be the address of a *sigset_t**-typed variable returning the old signal mask. If the *oldsetp* argument value is NULL, the old signal mask is ignored.

The *thr_kill* function sends the signal as given in the *signum* argument to a thread whose ID is given by the *tid* argument. The sending and receiving threads must be in the same process.

The following example adds the SIGINT signal to a thread signal mask, then sends the SIGTERM signal to a thread whose ID is 15:

```
sigset_t    set, oldset;
sigemptyset( &set );
sigaddset( &set, SIGINT );
if (thr_setsigmask( SIG_BLOCK, &set, &oldset)) perror("thr_sigsetmask");
if (thr_kill((thread_t)15, SIGTERM)) perror("thr_kill");
```

13.3.5 thr_setprio, thr_getprio, thr_yield

The prototypes of the *thr_setprio, thr_getprio and thr_yield* functions are:

```
#include <thread.h>

int     thr_setprio ( thread_t tid, int prio);
int     thr_getprio ( thread_t tid, int* priop);
void    thr_yield ( void );
```

The *thr_setprio* function sets the scheduling priority of a thread, as designated by the *tid* argument, to the *prio* value. Threads with higher priority values are scheduled more often than are those with lower values.

The *thr_getprio* function returns a thread's current priority value via the *priop* argument. The thread is designated by the *tid* argument.

The *thr_yield* function is called by a thread to yield its execution to other threads with the same priority. This function always succeeds and does not return any value.

Thread scheduling is done via thread library functions, not by the kernel. Threads are scheduled to bind to LWPs, which, in turn, are scheduled by the kernel to be executed on a hardware processor.

13.3.6 thr_setconcurrency, thr_getconcurrency

The function prototypes of the *thr_setconcurrency* and *thr_getconcurrency* functions are:

```
#include <thread.h>
int     thr_setconcurrency ( int amount );
int     thr_getconcurrency ( void );
```

The *thr_setconcurrency* function specifies the minimum number of LWPs that should be kept in a process. This ensures that a minimum number of threads are executing concurrently at any time. Note that the system takes the *amount* argument value as a hint, but it honors this concurrency request based on the availability of system resources.

The *thr_getconcurency* function returns the current concurrency level of a process.

13.3.7 Multithreaded Program Example

The following program is a rewrite of the RPC-based client and server programs shown in Section 12.5 of Chapter 12. The client program gets a message string from a user, issues an RPC call to the server *printmsg* function, which prints the message on the server system console.

The changes made in this version are in the client program (*msg_cls.C*) only: The client repeatedly prompts a user for a server host name and a message. For each host name and message data received from the user, the client process calls a function to allocate a dynamic memory region to hold the data. It then creates a thread to make an RPC call to a server running on the specified host, and requests that the user message be printed on the server system console.

The new *msg_cls2.C* client program is:

```
#include <thread.h>
#include <signal.h>
#include "msg2.h"
#include "RPC.h"

/* Record to hold one host name and message data for one thread */
typedef struct
{
   char     *host;
   char     *msg;
} MSGREC;

#define    MAX_THREAD        200
thread_t thread_list[ MAX_THREAD ];          // stores all threads' IDs

/* Function executed by a thread to send a message */
void* send_msg( void* ptr )
{
    int              res;
   MSGREC        *pRec = (MSGREC*)ptr;

   /* Set thread's signal mask to everything except SIGHUP */
   sigset_t  setv;
   sigfillset(&setv);
   sigdelset(&setv, SIGHUP);
   if (thr_sigsetmask(SIG_SETMASK,&setv,0)) perror("thr_setsigmask");

   /* Create a client handle to communicate with a host */
   RPC_cls cl( pRec->host, MSGPROG, MSGVER, "netpath");
```

```
    if (!cl.good()) thr_exit( &res );

    /* Call the remote host to print the message to its system console */
    (void)cl.call( PRINTMSG, (xdrproc_t)xdr_string, (caddr_t)&(pRec->msg),
                        (xdrproc_t)xdr_int, (caddr_t)&res);

    /* Delete dynamic memory */
    delete pRec->msg;
    delete pRec->host;
    delete pRec;

    /* Check RPC function execution status */
    if (res!=0) cerr << "clnt: call printmsg fails\n";
    int *rcp = new int(res);
    thr_exit( rcp );
    return 0;
}


/* Function to create a thread for a user message */
int add_thread( int& num_thread )
{
    char host[60], msg[256];
    thread_t tid;
    int res;

    /* Get remote host name and message from a user */
    cin >> host >> msg;
    if (cin.eof()) return RPC_FAILED;   /* normal return */
    if (!cin.good())  {                 /* I/O error detected */
        perror("cin");
        return RPC_FAILED;
    }

    /* Create dynamic memory for message text and host name */
    MSGREC *pRec = new MSGREC;
    pRec->host = new char[strlen(host)+1];
    pRec->msg = new char[strlen(msg)+1];
    strcpy(pRec->host,host);
    strcpy(pRec->msg,msg);

    /* Create a suspended thread to process the message */
    if (thr_create( 0, 0, send_msg, pRec, THR_SUSPENDED, &tid ))  {
        perror("thr_create");
        return RPC_FAILED;
    }
```

```
    else if (num_thread>= MAX_THREAD)     {
       cerr << "Too many threads created!\n";
       return RPC_FAILED;
    }
    else {                    /* Create a thread successfully */
       thread_list[num_thread++] = tid;
       cout  << "Thread: " << (int)tid << " created for msg: '"
             << msg << " [" << host << "]'\n";
    }
    return RPC_SUCCESS;
}

/* Client main function */
int main(int argc, char* argv[])
{
    int             num_thread=0;
    thread_t        tid;
    int             *res;

    /* Set concurrency level */
    if (thr_setconcurrency(5)) perror("thr_setoncurrency");
    cout     << "No. LPWs: " << thr_getconcurrency() << endl;

    /* Create a thread to send each mesg input by a user */
    while (add_thread(num_thread)==RPC_SUCCESS) ;

    /* Set each thread's priority and launch it */
    for (int i=0; i < num_thread; i++)
    {
       thr_setprio(thread_list[i],i);
       thr_continue(thread_list[i]);
    }

    /* Wait for every thread to terminate */
    while (!thr_join(0,&tid,(void**)&res))
    {
       cerr << "thread: " << (int)tid << ", exited. rc=" << (*res) << endl;
       delete res;
    }

    /* Terminate the main tread */
    thr_exit(0);
    return 0;
}
```

In the above program, the client process starts up by calling the *thr_setconcurrency* function to set the number of available LWPs to five. This specifies that at least five threads can run concurrently at any one time. The process finds out the number of actual LWPs created via the *thr_getconcurrency* function and prints that information to the standard output.

The process next calls the *add_thread* function repeatedly until it returns a zero return value. Each time the *add_thread* function is called, it gets a host name and a message from a user via the standard input. If end-of-file or an input error occurs, the function returns a zero value to *main* to indicate the end of user input. If the input data is retrieved successfully, the *add_thread* function allocates a dynamic memory to store the user-specified host name and message into a MSGREC-typed record. The function then calls the *thr_create* function to create a thread to execute the *send_msg* function. The *send_msg* function input argument is the address of a MSGREC-typed variable just allocated. The thread is created with a system-allocated stack, and it is suspended immediately after being created. The newly created thread's ID is stored in a global array *thread_list*, and the *add_thread* function returns a 1 value to *main* to indicate a successful execution status. If, however, the thread is not created successfully or MAX_THREAD threads have already been created, the function returns a zero value to *main* to signal the error.

After the *add_thread* function has created all the threads needed to handle all user input data, the *main* function (the main thread) scans the *thread_list* array and sets the schedule priority of each thread. The thread ID is stored in the array with a value related to the thread's position (index) in the array. Thus, the first thread in the array has the lowest priority, the next one has the second-lowest priority, and so on. The *main* function launches each suspended thread to run via the *thr_continue* function. After all this, the *main* function waits for each thread to terminate and prints the thread ID and exit status code to the standard output.

Each thread launched executes the *send_msg* function. The thread first sets its signal mask to include everything except the SIGHUP signal. It then creates an RPC client handle to be connected to the *printmsg* server whose host name is given in the *send_msg* function input argument. If the RPC client handle is created successfully, the thread calls the *RPC_cls::call* function to send the user message to the *printmsg* server. This prints the message to the server's system console. Finally, the thread deletes the dynamic memory that stores the host name and message, then creates a dynamic memory to store the return status value, and returns this dynamic data to the *main* function via the *thr_exit* call. The reason that the exit code is stored in a dynamic memory is because the *send_msg* function is executed by multiple threads, and each thread must return its own status code. Thus, the return value cannot be stored in an automatic or a static variable, but must be put in a dynamic variable that is unique for each thread.

The *printmsg* server program (*msg_svc2.C*) and the *RPC_cls* class definition as specified in the *RPC.h* header are shown in Section 12.5 of Chapter 12. The new *msg_cls2.C* client program is compiled as follows:

```
% CC -DSYSV4 msg_cls2.C -o msg_cls2 -lthread -lnsl
```

The *-lthread* and *-lnsl* options tell the link editor to link the *msg_cls2.o* object file with the thread (*libthread.so*) and network (*libnsl.so*) libraries, respectively. The *libthread.so* library contains all the thread API object codes, and the *libnsl.so* library contains the RPC-related codes.

The sample run and output of the client program is shown below. The words in italic are input data entered by a user from the standard input.

```
% msg_cls2
No. LPWs: 5
fruit    happy_day
Thread: 4 created for msg: 'happy_day [fruit]'
fruit    easter_sunday
Thread: 5 created for msg: 'easter_sunday [fruit]'
fruit    Good_bye
Thread: 6 created for msg: 'Good_bye [fruit]'
thread: 5, exited. rc=139424
thread: 6, exited. rc=139424
thread: 4, exited. rc=139424
%
```

The system console of the machine *fruit* shows the following for the above run:

```
server: 'easter_sunday'
server: 'Good_bye'
server: 'happy_day'
```

13.4 POSIX.1c Thread APIs

This section describes the POSIX.1c thread APIs for basic thread manipulation. The corresponding POSIX.1c and Sun thread APIs are:

Sun thread API	**POSIX.1c thread API**
thr_create	pthread_create
thr_self	pthread_self
thr_exit	pthread_exit
thr_kill	pthread_kill
thr_join	pthread_join

To use these APIs, application program should include the <pthread.h> header which declares all the POSIX.1c multithreaded function prototypes. Furthermore, if a program manipulates threads scheduling priority, then the <sched.h> header should be included also.

Unless otherwise stated, most of the thread APIs depicted below return a 0 value if they succeed, or a -1 value if they fail. In case they fail *perror* may be called to print error diagnostic messages.

13.4.1 pthread_create

The *pthread_create* function prototype is:

```
#include <pthread.h>
int         pthread_create ( pthread_t* tid_p, const pthread_attr_t* attr,
                             void* (*funcp)(void*), void* argp );
```

The API creates a new thread to execute a function whose address is given in the *funcp* argument. The function specified in *funcp* should accept a *void**-typed input argument and return a *void** data. The actual argument to be passed to the *funcp* function, when the new thread starts execution, is specified in the *argp* argument.

The new thread ID is returned via the *tid_p* argument. If the actual value of the *tid_p* argument is assigned NULL, then no thread ID is returned. The thread ID data type is *pthread_t*.

The *attr* argument contains properties to be assigned to the newly created thread. The *attr* argument value may be NULL, if the new thread is to use the system default property values, or it is the address of an attribute object. POSIX.1c defines a set of APIs to create, destroy, inquire or set attribute objects. An attribute object may be associated with multiple threads, so that whenever an attribute object's properties are updated, all threads associated with that object will be affected by the changes. This is different from Sun threads where properties are specified for each thread individually.

The *pthread_attr_init* and *pthread_attr_destroy* APIs creates and destroys, respectively, an attribute object:

```
#include <pthread.h>
int         pthread_attr_init ( pthread_attr_ t* attr_p );
int         pthread_attr_destroy ( pthread_attr_ t* attr_p );
```

Once an attribute object is created via the *pthread_attr_init* function, its properties may be checked or set via the *pthread_attr_get* and *pthread_attr_set* APIs, respectively. The possible properties which may be contained in an attribute object and the associated APIs to check and set them are:

Property	**API to check**	**API to set**
Contention scope	pthread_attr_getscope	pthread_attr_setscope
Stack size	pthread_attr_getstacksize	pthread_attr_setstacksize
Stack address	pthread_attr_getstackaddr	pthread_attr_setstackaddr
Detach State	pthread_attr_getdetachstate	pthread_attr_setdetachstate
Schedule policy	pthread_attr_getschedpolicy	pthread_attr_setschedpolicy
Schedule Parameters	pthread_attr_getschedparam	pthread_attr_setschedparam

All the above *pthread_attr_get* APIs take two arguments: The first argument is the pointer to an attribute object, and the second argument is the address of a variable to hold the inquired property value. Similarly, all the above *pthread_attr_set* APIs also take two arguments:The first argument is the pointer to an attribute object, and the second argument is either a new property value or the pointer to a variable which holds the new property value.

The thread scheduling contention scope has been explained in Section 13.2. The possible values to set this property are PTHREAD_SCOPE_PROCESS or PTHREAD_SCOPE_SYSTEM.

A thread detach state specifies if a thread is created detached or joinable. The possible values for this property are PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE.

A thread scheduling policy specifies, among other things, the scheduling priority of a thread. The second argument to the *pthread_attr_getschedparam* and *pthread_attr_setschedparam* APIs is the address of a *struct sched_param*-typed variable. In this variable there is an integer-typed *shced_priority* field, which specifies the scheduling priority of any thread that owns this property.

Finally, a newly created thread's run-time stack size and address may be set in a similar fashion as the *stack_size* and *stackp* arguments to the *thr_create* call (see Section 13.3.1). The difference here is that *pthread_attr_setstacksize* and *pthread_attr_setstackaddr* APIs are used to set these properties for one or more threads.

The following sample code creates a new detached and bound thread with a scheduling priority of five. The thread executes a function called *do_it* with an argument specified by the *pInt* variable. The new thread's ID is assigned to the *tid* variable and its stack is allocated by the function with the system default size:

```
extern void* do_it (void* ptr);
int                     *pInt;
pthread_t               tid;
pthread_attr_t          attr, *attrPtr = &attr;
struct sched_param      sched;

if (pthread_attr_init( attrPtr ) == -1) {
    perror("pthread_attr_init");
    attrPtr = 0;
}
else {
    pthread_attr_setdetachstate( attrPtr,PTHREAD_CREATE_DETACH);
    pthread_attr_setscope( attrPtr, PTHREAD_SCOPE_SYSTEM);
    if (pthread_attr_getschedparam(attrPtr,&sched)==0) {
        sched.sched_priority = 5;
        pthread_attr_setschedparam(attrPtr, &sched );
    }
}
if (pthread_create( &tid, &attr, do_it, (void*)&pInt ) == -1)
     perror("pthread_create");
```

13.4.2 pthread_exit, pthread_detach, pthread_join

The function prototypes of the *pthread_exit, pthread_detach* and *pthread_join* functions are:

```
#include <pthread.h>

int     pthread_exit ( void* status );
int     pthread_detach ( void* status );
int     pthread_join ( pthread_t tid, void** statusp);
```

Both the *pthread_exit* and *pthread_detach* functions terminate a thread. The *pthread_exit* function may be used by a non-detached thread, while the *pthread_detach* function is used by a detached thread. The actual argument value to the *statusp* argument is the address of a static variable which contains the exit status code of the terminating thread. If no other thread is expected to retrieve the terminating thread exit status code (e.g., the thread is detached), then the *statusp* argument value may be specified as NULL.

The *pthread join* function is called to wait for the termination of a non-detached thread and returns the thread's exit status value as passed via a *pthread_exit* call.

The following example waits for a non-detached threads to terminate, then terminates the current thread:

```
status int      *rc, rval=0;
thread_t        tid;
if (!pthread_join(tid, &rc))
        cout << "thread: " << (int)tid << ", exits, rc=" << (*rc) << endl;
pthread_exit( (void*)&rval );
```

13.4.3 pthread_sigmask, pthread_kill

The prototypes of the *pthread_sigmask* and *pthread_kill* functions are:

```
#include <pthread.h>
#include <signal.h>

int     pthread_sigmask ( int mode, sigset_t *sigsetp, sigset_t *oldsetp);
int     pthread_kill ( pthread_t tid, int signum);
```

The *pthread_sigmask* function sets the signal mask of a calling thread. The *sigsetp* argument contains one or more of the signal numbers to be applied to the calling thread. The *mode* argument specifies how the signal(s) specified in the *sigsetp* argument is to be used. The possible values of the *mode* argument, as declared in the <signal.h> header, and their meanings are:

***mode* value**	**Meaning**
SIG_BLOCK	Adds signals contained in the *sigsetp* argument to the thread signal mask
SIG_UNBLOCK	Removes signals contained in the *sigsetp* argument from the thread signal mask
SIG_SETMASK	Replaces the thread signal mask with the signal(s) specified in the *sigsetp* argument

If the *sigsetp* argument value is NULL, the *mode* argument value is don't-care.

The *oldsetp* argument value should be the address of a *sigset_t**-typed variable returning the old signal mask. If the *oldsetp* argument value is NULL, then the old signal mask is ignored.

The *pthread_kill* function sends a signal, as specified in the *signum* argument, to a thread whose ID is given by the *tid* argument. The sending and receiving threads must be in the same process.

The following example adds the SIGINT signal to a thread signal mask, then sends the SIGTERM signal to a thread whose ID is 15:

```
sigset_t    set, oldset;
sigemptyset( &set );
sigaddset( &set, SIGINT );
if (pthread_setmask( SIG_BLOCK, &set ,&oldset ))
        perror("thr_sigsetmask");
if (pthread_kill((thread_t)15, SIGTERM)) perror("pthread_kill");
```

13.4.4 sched_yield

The function prototype of the *sched_yield* API is:

```
#include <pthread.h>
int         sched_yield ( void );
```

The *sched_yield* function is called by a thread to yield its execution to other threads with the same priority. This function returns 0 on success and -1 when fails. This API is the POSIX.1c counterpart to the Sun's *thr_yield* API.

13.5 Thread Synchronization Objects

Threads in a process share the same address space as the process. This means that global and static variables in the process are accessible by all its threads. To ensure the correct manipulation of these variables, threads must use some methods to synchronize their operations. Specifically, no thread should access a variable while its value is being changed by a thread. Furthermore, no thread should change a variable value while other threads are reading that variable value.

Thread synchronization is also needed when two or more threads are doing I/O operations on a common stream. For example, if two threads are writing to a stream simultaneously, it is unpredictable as to which data is output to that stream. Also, a similar problem occurs when two threads are reading data from a stream simultaneously.

To solve these thread synchronization problems, Sun and POSIX.1c provide the following objects to control thread operations on shared data and I/O streams in a process:

- Mutually exclusive locks (mutex locks)
- Condition variables
- Semaphores

Of the above objects, mutex locks are the most primitive and efficient to use. They are used to serialize the access of shared data or execution of code segments.

Condition variables are the second most efficient method after mutex locks. They are commonly used with mutex locks to control asynchronous access of shared data.

Semaphores are more complex than mutex locks and condition variables. They are used in the similar manner as the UNIX System V and POSIX.1b semaphores.

In addition to the above, Sun provides read-write locks as one more means for threads synchronization. Specifically, read-write locks allow multiple read-access and single write-access to any shared data. This capability is not provided by the aforementioned objects. Read-write locks are commonly used to guard data that are read frequently, but changed infrequently, by multiple threads.

Processes that use any one of these synchronization objects need to define storage for them in their virtual address space. If these objects are defined in shared memory regions that are accessible by multiple processes, they can be used to synchronize threads in these different processes.

The following sections describe these objects in more detail.

13.5.1 Mutually Exclusive Locks (mutex Locks)

Mutex locks serialize the execution of threads, such that when multiple threads try to acquire a mutex lock, only one of them can succeed and continue its execution. The other threads are blocked until the lock is released (unlocked) by its owner thread. When a mutex lock is released, it is unpredictable as to which pending thread is freed to acquire the lock.

The following table lists the corresponding Sun and POSIX.1c APIs for mutex locks manipulation:

Sun APIs	POSIX.1c API
mutex_init	pthread_mutex_init
mutex_destroy	pthread_mutex_destroy
mutex_lock	pthread_mutex_lock
mutex_trylock	pthread_mutex_trylock
mutex_unlock	pthread_mutex_unlock

The following two sub-sections describe these Sun and POSIX.1c mutex lock APIs in more detail.

13.5.1.1 Sun Mutex Locks

The Sun thread library functions for mutex lock operations are:

Function	Use
mutex_init	Initializes a mutex lock
mutex_lock	Sets a lock on a mutex lock
mutex_unlock	Unlocks a mutex lock
mutex_trylock	Like *mutex_lock*, except it is nonblocking
mutex_destroy	Discards a mutex lock

The prototypes of these functions are:

```
#include <thread.h>

int     mutex_init ( mutex_t *mutxp, int type, void* argp );
int     mutex_lock( mutex_t* mutxp );
int     mutex_trylock ( mutex_t* mutxp );
int     mutex_unlock ( mutex_t* mutxp );
int     mutex_destroy ( mutex_t* mutxp );
```

The *mutxp* argument value is the address of a *mutex_t*-typed variable. This variable is defined by the calling thread and is set to reference a mutex lock via the *mutex_init* function. The *type* argument of the *mutex_init* function specifies whether the mutex lock is accessible by threads in different processes. Its possible values and meanings are:

type value	Meaning
USYNC_PROCESS	The mutex lock can be used by threads in different processes
USYNC_THREAD	The mutex lock can be used by threads in the calling process only

The *argp* argument of the *mutex_init* function is currently unused. Its value should be specified as 0.

A mutex lock should be initialized only once in a process. It is discarded by the *mutex_destroy* function.

A mutex lock is acquired (or locked) by a thread via the *mutex_lock* function and released (unlocked) via the *mutex_unlock* function. The *mutex_lock* function blocks a calling thread if the lock has already been acquired by another thread. The thread is unblocked when the mutex lock is unlocked, and the thread is allowed to acquire it.

The *mutex_trylock* function is similar to the *mutex_lock* function except that if a requested lock is already acquired by another thread, the function returns an error status to the calling thread, rather than blocking it.

13.5.1.2 POSIX.1c Mutex Locks

The POSIX.1c APIs for mutex lock operations are similar to that of Sun:

Function	Use
pthread_mutex_init	Initializes a mutex lock
pthread_mutex_lock	Sets a lock on a mutex lock
pthread_mutex_unlock	Unlocks a mutex lock
pthread_mutex_trylock	Like *pthread_mutex_lock*, except it is nonblocking
pthread_mutex_destroy	Discards a mutex lock

The prototypes of these functions are:

```
#include <pthread.h>
int     pthread_mutex_init ( pthread_mutex_t *mutxp,
                             pthread_mutexattr_t* attrp);
int     pthread_mutex_lock( pthread_mutex_t* mutxp );
int     pthread_mutex_trylock ( pthread_mutex_t* mutxp );
```

```
#include <pthread.h>

int     pthread_mutex_unlock ( pthread_mutex_t* mutxp );
int     pthread_mutex_destroy ( pthread_mutex_t* mutxp );
```

The *mutxp* argument value is the address of a *pthread_mutex_t*-typed variable. This variable is defined by the calling thread and is set to reference a mutex lock via the *pthread_mutex_init* function. The *attrp* argument is a pointer to an attribute object for the new mutex lock. This value may be NULL, if a mutex lock is to use default property values; otherwise, it is a pointer to a *pthread_mutexattr_t*-typed object that contains property values for the new mutex lock.

As an alternative of calling the *pthread_mutex_init* API, a mutex lock may also be initialized via the PTHREAD_MUTEX_INITIALIZER static initializer. Thus, the following code:

```
pthread_mutex_t     lockx;
(void)pthread_mutex_init( &lockx, 0 );
```

is the same as this statement:

```
pthread_mutex_t     lockx = PTHREAD_MUTEX_INITIALIZER;
```

The *pthread_mutex_lock*, *pthread_mutex_trylock*, *pthread_mutex_unlock*, and *pthread_mutex_destroy* have the similar invocation syntax and the same corresponding function as the Sun *mutex_lock*, *mutex_trylock*, *_mutex_unlock*, and *_mutex_destroy* APIs, respectively.

13.5.1.3 Mutex Lock Examples

The following *printmsg* function may be called by any thread to print messages to the standard output. This function ensures that only one thread can print a message to the standard output at a time. The *main* function is a sample application to test the *printmsg* function:

```
/* printmsg.C */
static mutex_t lockx;                    // define the lock storage
void* printmsg (void* msg )
{
   /* acquire the lock */
   if (mutex_lock(&lockx))
      perror("mutex_lock");
```

```
    else {
      /* print the msg */
      cout << (char*)msg << endl << flush;

      /* release the lock */
      if (mutex_unlock(&lockx)) perror("mutex_unlock");
    }
}

int main(int argc, char* argv[])
{
   /* initialize the mutex lock */
    if (mutex_init(&lockx, USYNC_THREAD,NULL)) perror("mutex_lock");

   /* create threads which call printmsg */
   while (--argc > 0)
      if (thr_create(0,0,printmsg,argv[argc],0, 0)) perror("thr_create");

   /* wait for all threads to terminate */
   while (!thjr_join(0,0,0)) ;

   /* discard the mutex lock */
   if (mutex_destroy(&lockx)) perror("mutex_destroy");
   return 0;
}
```

In the above example, the *main* function initializes the mutex lock *lockx* to be used by all threads in the same process. It then creates multiple threads, one per command line argument, to call the *printmsg* function and display the command argument strings to the standard output. It does not matter how many threads call the *printmsg* function simultaneously, as the function serves only one thread at a time.

A sample run of the *printmsg.C* program and its output is:

```
% CC printmsg.C -lthread -o printmsg
% printmsg "1" "2" "3"
3
2
1
```

As another example, the following *glob_dat* class defines a data type for variables that can be accessed by multiple threads simultaneously. This class is defined in a *glob_dat.h* header:

```
#ifndef GLOB_DAT_H
#define GLOB_DAT_H

#include <iostream.h>
#include <thread.h>
#include <stdio.h>

class glob_dat
{
   private:
      int         val;
      mutex_t     lockx;
   public:
      // constructor function
      glob_dat( int a )
      {
         val = a;
         if (mutex_init(&lockx, USYNC_THREAD,0)) perror("mutex_init");
      };
      // destructor function
      ~glob_dat() { if (mutex_destroy(&lockx)) perror("mutex_destroy"); };

      // set new value
      glob_dat& operator=( int new_val )
      {
         if (!mutex_lock(&lockx)) {
            val = new_val;
            if (mutex_unlock(&lockx)) perror("mutex_unlock");
         }else perror("mutex_unlock");
         return *this;
      };
      // retrieve value
      int getval( int* valp )
      {
         if (!mutex_lock(&lockx)) {
            *valp= val;
            if (!mutex_unlock(&lockx)) return 0;
            perror("mutex_unlock");
         } else perror("mutex_lock");
         return -1;
      };

      // show value to an output stream
      friend ostream& operator<<( ostream& os glob_dat& gx)
      {
```

```
        if (!mutex_lock(&gx.lockx)) {
            os << gx.val;
            if (mutex_unlock(&gx.lockx)) perror("mutex_lock");
        } else perror("mutex_lock");
        return os;
    };
};      /* glob_dat */
#endif
```

The actual value of a *glob_dat*-type variable is stored in the *glob_dat::val* data field. The *glob_dat::lockx* mutex lock is used to serialize the access of the *glob_dat::val* by multiple threads. The *glob_dat::lockx* and *glob_dat::val* members are initialized when a variable of this type is defined. The *glob_dat::lockx* mutex lock is discarded when the *glob_dat::~glob_dat* destructor function is called.

The *glob_dat::getval, glob_dat::operator<<*, and *glob_dat::operator=* member functions allow threads to retrieve, show, and set any *glob_dat-typed* variable value. By using the mutex lock, these functions are MT (multithreaded) -safe, meaning they can be called by multiple threads simultaneously to operate on the same variable without causing problems.

The following example program, *glob_dat.C*, illustrates how a *glob_dat*-typed variable, *globx* is accessed by multiple threads:

```
#include <strstream.h>
#include "glob_dat.h"
glob_dat globx (0);                          // define globx

void* mod_val (void* np )
{
   int old_val, new_val;
   istrstream((char*)np) >> new_val;
   if (!globx.getval(&old_val))    {         // get current value
      globx = old_val + new_val;             // add new value
      cout  << (int)thr_self() << ": arg=" << (char*)np
            << "' -> " << globx << endl;                  // show result
   }
   return 0;
}

int main(int argc, char* argv[])
{
   thread_t tid;
   while (--argc > 0)                  // for each command line argument
```

```
    if (thr_create(0,0,mod_val,argv[argc],0,&tid)) perror("thr_create");
  while (!thr_join(0,0,0)) ;          // wait for threads to terminate
  return 0;
}
```

The *main* function creates the *globx* variable with an initial value of zero. It then creates a thread for each command line argument to execute the *mod_val* function. The actual argument value passed to each *mod_val* function call is a command line argument. It is made up of an integer value text string (e.g., *"51"*). The *mod_val* function converts its argument to an integer value, gets the current value of the *globx* value and adds the new integer value to the current value. The resultant value is assigned to the *globx* variable, which is then printed to the standard output.

A sample run of the above program and its output is:

```
% CC glob_dat.C -lthread -o glob_dat
% glob_dat "1" "2" "3"
4: arg='3' -> 3
5: arg='2' -> 5
6: arg='1' -> 6
```

One problem to avoid in using mutex locks is in creating a dead-lock condition within a process. This may occur when a process uses multiple mutex locks and two or more threads in that process attempt to acquire these locks in random order. For example, suppose two mutex locks (*A* and *B*) are initialized in a process and two threads (*X* and *Y*) in the process use these locks. Suppose thread *X* has acquired lock *A*, and thread *Y* has acquired lock *B*. When X attempts to acquire lock *B*, it is blocked because lock *B* is owned by thread *Y*. If thread *Y* then attempts to acquire lock *A*, it is also blocked. This creates a dead-lock condition: both *X* and *Y* are blocked as each of them tries to acquire a lock owned by the other. Neither of these threads can proceed any further, as the owner of the lock is blocked and unable to release the lock.

To prevent dead-lock conditions, threads that use mutex locks should always acquire them in the same order and/or use the *thr_trywait* function instead of *thr_wait* to acquire locks. In the first method, because all threads acquire locks in the same order, it is impossible for a thread to attempt acquiring a lock that is owned by a blocked thread. In the second method, if a thread uses the *thr_trywait* function to acquire a lock, the function returns a non-zero value immediately (if the lock is owned by another thread). Thus, the thread is not blocked and can perform some other work and try to acquire the lock again later.

13.5.2 Condition Variables

Condition variables are used to block threads until certain conditions become true. Condition variables are usually used with mutex locks so that multiple threads can wait on the same condition variable. This is done as follows: First, a thread acquires a mutex lock, but is blocked by a condition variable, pending the occurrence of a specific condition. While the thread is blocked, the mutex lock it acquires is released automatically. When another thread modifies the state of the specific condition, it signals the condition variable to unblock the thread. When the thread is unblocked, the mutex lock is reacquired automatically, and the thread tests the condition again. If the condition remains false, the thread is again blocked by the condition variable. On the other hand, if the condition is now true, the thread releases the mutex lock and proceeds with its execution

13.5.2.1 Sun Condition Variables

The Sun thread library functions for condition variable operation are:

Function	Use
cond_init	Initializes a condition variable
cond_wait	Blocks on a condition variable
cond_timedwait	Same as *cond_wait*, except that a time out period for the block duration is specified
cond_signal	Unblocks a thread that is waiting on a condition variable
cond_broadcast	Unblocks all threads that are waiting on a condition variable
cond_destroy	Discards a condition variable

The prototypes of these functions are:

```
#include <thread.h>

int     cond_init ( cond_t *condp, int type, int argp );
int     cond_wait ( cond_t* condp, mutex_t *mutxp);
int     cond_timedwait ( cond_t* condp, mutex_t* mutxp, timestruct_t * timp );
int     cond_signal ( cond_t* condp );
int     cond_broadcast ( cond_t* condp );
int     cond_destroy ( cond_t* condp );
```

The *condp* argument value is the address of a *cond_t*-typed variable. This variable is set to reference a condition variable via the *cond_init* function. The *type* argument of the *cond_init* function specifies whether the condition variable is accessible by different processes. Its possible values and meanings are:

***type* value**	**Meaning**
USYNC_PROCESS	The condition variable can be used by threads in different processes
USYNC_THREAD	The condition variable can be used by threads in the calling process only

The *arg* argument of the *cond_init* function is currently unused. Its value should be specified as zero.

The *cond_wait* function blocks the calling thread to wait for the condition variable to change state as specified by the *condp* argument. It also releases the mutex lock as specified by the *mutxp* argument.

When another thread calls the *cond_signal* function on the same condition variable, the mutex lock is reacquired for a pending thread. That thread is unblocked and resumes execution at the return of the *cond_wait* function call. If there are multiple threads blocked by the same condition variable, they should all use the same mutex lock. When the condition variable is signaled, only one of the blocked threads can succeed in acquiring the mutex lock and proceed with execution. The other threads continue to be blocked by the same mutex lock.

The *cond_timewait* function is similar to the *cond_wait* function, except that it has a third argument *timp* which specifies that the calling thread should not be blocked past the time-of-day as specified in that argument.

The *cond_signal* signals a condition variable, as specified by the *condp* argument and unblocks the thread waiting on that variable. The *cond_broadcast* signals a condition variable and unblocks all threads that are waiting. If there is no thread waiting on a signaled condition variable, the *cond_signal* or *cond_broadcast* call on that variable has no effect.

13.5.2.2 Condition Variable Example

The following example illustrates the uses of a mutex lock and a condition variable. The program, *pipe.C*, creates a reader thread and a writer thread. The reader thread reads data from a user and passes them to a writer thread via a global array *msgbuf*. The writer thread prints messages contained in the *msgbuf* to the standard output. The two threads terminate when end-of-file is encountered from the user. The reader and writer threads synchronize their access of *msgbuf* via a mutex lock and a condition variable.

The *pipe.C* program is:

```
#include <iostream.h>
#include <thread.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>

#define FINISH()  { cerr << (int)thr_self() << " exits\n"; \
      mutex_unlock(&mutx); thr_exit(0); return 0; }

mutex_t mutx;
cond_t  condx;
int     msglen, done;
char    msgbuf[256];


/* write messages sent from the reader thread to the standard output */
void* writer (void* argp )
{
   do   {
      mutex_lock(&mutx);                    // acquire the mutex lock
      while (!msglen)     {                 // loop if no message
            cond_wait(&condx,&mutx);        // wait on the cond. variable
         if (done) FINISH();                // kill thread if done
      }
      cout << "*> " << msgbuf << endl;      // print mesg to std output
      msglen = 0;                           // reset msg buffer size
      mutex_unlock(&mutx);                  // release the mutex lock
   } while (1);

    FINISH();                               // clean up and exit
}

/* read messages from user and send them to the writer thread */
void* reader (void* argp )
{
   do   {
      mutex_lock(&mutx);                    // acquire the mutex lock
      if (!msglen)  {                       // check buffer is empty
         if (!cin.getline(msgbuf,256)) break;  // get input from user
         msglen = strlen(msgbuf)+1;         // set msg length
         cond_signal(&condx);               // signal writer to read msg
      }
      mutex_unlock(&mutx);                  // release the mutex lock
```

```
    } while (1);

    FINISH();                                   // clean up and exit
}

/* main thread to control the reader and writer threads */
main()
{
    thread_t wtid, rtid, tid;

    /* initialize mutex lock and condition variable */
    mutex_init(&mutx, USYNC_PROCESS, 0);
    cond_init(&condx, USYNC_PROCESS, 0);

    /* create a writer thread */
    if (thr_create(0,0,writer,0,0,&wtid)) perror("thr_create");


    /* create a read thread */
    if (thr_create(0,0,reader,0,0,&rtid)) perror("thr_create");

    /* wait for the read thread to exit */
    if (!thr_join(rtid,&tid,0))  {
        done = 1;
        cond_signal(&condx);
    }

    /* clean up */
    mutex_destroy(&mutx);
    cond_destroy(&condx);
    thr_exit(0);
}
```

The *main* function initializes the *mutx* mutex lock and the *condx* condition variable. It then creates the writer thread and reader thread to execute the *writer* and *reader* functions. The main thread then waits for the reader thread to terminate via the *thr_join* function and signals the write thread to terminate via the *done* global variable. After the reader and writer threads are terminated, the main thread discards the mutex lock and the condition variable via the *mutex_destroy* and *cond_destroy* functions, respectively.

The reader thread reads one or more input lines from a user from the standard input. For each line it reads, it first acquires the mutex lock to make sure it can access the *msgbuf* and *msglen* global variables. When the mutex lock is acquired successfully, the thread puts the user message into the *msgbuf* array and sets the *msglen* variable to be the size of the message text. It then uses the *cond_signal* and *mutex_unlock* functions to signal the writer pro-

cess to check the *msglen* variable. The reader thread terminates when it cannot read an input text from the user. In that case, it releases the mutex lock and terminates itself via the *thr_exit* function.

The writer thread constantly polls the *msglen* global variable. If the *msglen* value is not zero, a message is available in the *msgbuf* array and can be printed to the standard output. To process each message, the thread first acquires the mutex lock to make sure it can access the *msglen* and *msgbuf* variables exclusively. If that succeeds, it blocks the *cond_wait* function call until the reader thread or main thread calls the *cond_signal* function to unblock it. When the writer thread is unblocked, it checks whether the *done* variable value is nonzero. If so, the main thread signals the writer thread to exit. However, if the *done* variable is zero and the *msglen* variable value is not zero, the thread prints the message contained in *msgbuf* to the standard output. Otherwise, it goes back to the *cond_wait* loop to wait for a message to arrive. Note that when the writer thread is blocked in the *cond_wait* call, the *mutx* mutex lock is released so that the reader or main thread can acquire it. The *cond_wait* function automatically reacquires the mutex lock before unblocking the writer thread when the reader or main thread calls the *cond_signal* function on the *condx* variable.

A sample run of the *pipe.C* program and its output is:

```
%    CC pipe.C -lthread -o pipe
%    pipe
Have a good day
*> Have a good day
Bye-Bye
*> Bye-Bye
5 exists
4 exists
```

13.5.2.3 POSIX.1c Condition Variables

The corresponding POSIX.1c APIs to the Sun APIs for condition variable manipulation are:

POSIX.1c API	Sun API
pthread_cond_init	cond_init
pthread_cond_wait	cond_wait
pthread_cond_timedwait	cond_timedwait
pthread_cond_signal	cond_signal
pthread_cond_broadcast	cond_broadcast
pthread_cond_destroy	cond_destroy

The function prototypes of these POSIX.1c APIs are:

```
#include <pthread.h>

int     pthread_cond_init ( pthread_cond_t *condp, pthread_condattr_t *attr );
int     pthread_cond_wait ( pthread_cond_t* condp, pthread_mutex_t *mutxp);
int     pthread_cond_timedwait (pthread_cond_t* condp,
                                pthread_mutex_t* mutxp, struct timespec* timp );
int     pthread_cond_signal ( pthread_cond_t* condp );
int     pthread_cond_broadcast ( pthread_cond_t* condp );
int     pthread_cond_destroy ( pthread_cond_t* condp );
```

The *condp* argument value is the address of a *pthread_cond_t*-typed variable which references an allocated condition variable. The *attr* argument of the *pthread_cond_init* function is a pointer to an attribute object which specifies properties for the condition variable. The actual argument for *attr* may be 0 if the condition variable is to use default property values.

Note that a POSIX.1c condition variable may be initialized via the static initializer PTHREAD_COND_INITIALIZER. Thus the following code:

```
pthread_cond_t    cond_var;
(void) pthread_cond_init( &cond_var, 0 );
```

is the same as this statement:

```
pthread_cond_t    cond_var = PTHREAD_COND_INITIALIZER;
```

The *pthread_cond_wait*, *pthread_cond_timedwait*, *pthread_cond_signal*, *pthread_cond_broadcast* and *pthread_cond_destroy* have the same function as the Sun *cond_wait*, *cond_timedwait*, *cond_signal*, *cond_broadcast* and *cond_destroy* APIs, respectively. Please refer to Section 13.5.2.1 for description of these APIs.

13.5.3 Sun Read-Write Locks

Read-write locks are like mutex locks, except that these locks can be acquired for read-only and write-only. One or more threads can hold a read lock on a read-write lock simultaneously. A thread that wishes to set a write lock on a read-write lock is blocked until all read locks are released. On the other hand, if a thread acquires a write lock on a read-write lock,

no other threads can set any read or write lock on it until the former thread releases its write lock. If two threads attempt to acquire a read-write lock at the same time, one for read and the other for write, the write lock will be granted. Read-write locks are not as efficient as mutex locks, but they can be used to permit simultaneous read access of data by multiple threads.

Read-write locks are not defined in POSIX.1c, so they are Sun-specific.The Sun thread library functions for read-write locks operations are:

Function	Use
rw_init	Initializes a read-write lock
rw_rdlock	Acquires a read lock
rw_tryrdlock	Acquires a read lock, in nonblocking mode
rw_wrlock	Acquires a write lock
rw_trywrlock	Acquires a write lock, in nonblocking mode
rw_unlock	Unlocks a read-write lock
rw_destroy	Discards a read-write lock

The prototypes of these functions are:

```
#include <thread.h>

int     rw_init ( rwlock_t *rwp, int type, void* argp );
int     rw_rdlock ( rwlock_t* rwp);
int     rw_tryrdlock ( rwlock_t* rwp );
int     rw_wrlock ( rwlock_t * rwp);
int     rw_trywrlock( rwlock_t* rwp );
int     rw_unlock ( rwlock_t* rwp );
int     rw_destroy ( rwlock_t* rwp );
```

The *rwp* argument value is the address of a *rwlock_t*-typed variable. This variable is set to reference a read-write lock via the *rw_init* function. The *type* argument of the *rw_init* function specifies whether or not the read-write lock is accessible by different processes. Its possible values and meanings are:

type value	Meaning
USYNC_PROCESS	The read-write lock can be used by threads in different processes
USYNC_THREAD	The read-write lock can be used by threads in the calling process only

The *argp* argument of the *rw_init* function is currently unused. Its value should be specified as zero.

The *rw_rdlock* function attempts to acquire a read lock on a read-write lock as specified by the *rwp* argument. This function blocks the calling thread until the operation succeeds.

The *rw_tryrdlock* function is like the *rw_rdlock* function, except that it is non-blocking. If the function aborts because the requested lock has been set as write-only by another thread, it returns a non-zero value and sets the *errno* to EBUSY.

The *rw_wrlock* function attempts to acquire a write lock on a read-write lock as specified by the *rwp* argument. This function blocks the calling thread until the operation succeeds.

The rw_*trywrlock* function is like the *rw_wrlock* function, except that it is nonblocking. If the function aborts because the requested lock is owned by another thread, it returns a non-zero value and sets the *errno* to EBUSY.

The *rw_unlock* and *rw_destroy* functions unlock and discard, respectively, a read-write lock as specified by the *rwp* argument.

The following *pipe2.C* program is a rewrite of the *pipe.C* program presented in the last section. The new program uses a read-write lock instead of a mutex lock and a condition variable to synchronize the reader and writer threads.

```
/* pipe2.C */
#include <iostream.h>
#include <thread.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>

rwlock_t rwlk;
int    msglen, done = 0;
char   msgbuf[256];

/* writer thread function */
void* writer (void* argp )
{
   while (!done)   {
      if (rw_rdlock(&rwlk)) perror("rw_rdlock");      // set a read lock
      if (msglen)   {                                 // check mesg. exits
         cout << "*> " << msgbuf << endl;             // print a message
```

```
            msglen = 0;                                    // reset msg. buffer
        }
        if (rw_unlock(&rwlk)) perror("rw_unlock(1)");      // unlock rw lock
        thr_yield();                                       // yield to other thread
    }
    /* clean up and exit thread */
    cerr << "write thread (" << (int)thr_self() << ") exits\n";
    thr_exit(0);
    return 0;
}

/* reader thread function */
void* reader (void* argp )
{
    do {
        if (rw_wrlock(&rwlk)) perror("rw_wrlock");         // set a write lock
        if (!msglen)  {                                    // empty buffer ?
            if (!cin.getline(msgbuf,256)) break;           // get a mesg from a user
            msglen = strlen(msgbuf)+1;                     // set mesg size
        }
        if (rw_unlock(&rwlk)) perror("rw_unlock(2)");      // unlock rw lock
        thr_yield();                                       // yield to other thread
    } while (1);

    /* clean up and exit thread */
    cerr << "read thread (" << (int)thr_self() << ") exits\n";
    if (rw_unlock(&rwlk)) perror("rw_unlock(3)");
    thr_exit(0);
    return 0;
}

/* main thread function */
main()
{
    thread_t wtid, rtid, tid;
    (void)rw_init(&rwlk, USYNC_PROCESS, 0);

    /* create a writer thread */
    if (thr_create(0,0,writer,0,0,&wtid)) perror("thr_create");

    /* create a read thread */
    if (thr_create(0,0,reader,0,0,&rtid)) perror("thr_create");

    /* wait for read process to exit */
    if (!thr_join(rtid,&tid,0)) done = 1;
```

```
    /* clean up */
    rwlock_destroy(&rwlk);
    thr_exit(0);
}
```

The *main* function initializes a read-write lock, then creates reader and writer threads to work together. The main thread waits for the reader thread to exit, then signals the writer thread to terminate via the *done* variable. After that, the main thread discards the read-write lock and exits.

The reader thread executes the *read* function. It gets one or more lines of text from a user. For each input message line, it acquires a read-write lock for write access. This ensures that it has exclusive use of the *msgbuf* and *msglen* variables. When the write lock is acquired, the thread writes the message text to the *msgbuf* array, sets the *msglen* to the size of the message text, and releases the read-write lock. The thread then yields its execution to the writer thread, so that the latter can access the lock and *msgbuf* variable. After that, it gets the next message from the user and repeats the above process. The thread exits when it cannot read any text from a user. When that happens, it unlocks the read-write lock and exits via the *thr_exit* function.

The writer thread executes the *write* function. It reads one message at a time from the reader thread and prints the message to the standard output. For each message it processes, it first acquires a read lock on the read-write lock and checks that the *msglen* variable value is greater than zero. If a message is present in the *msgbuf* variable, the thread prints the message to the standard output and resets the *msglen* to zero. After these procedures, the thread unlocks the read-write lock and yields its execution to the reader thread so that the latter can access the lock and put the next message into the *msgbuf* buffer. The thread terminates when the global variable *done* is set to nonzero by the main thread. This means that no more messages are available, and the thread exits via the *thr_exit* function.

A sample run of the *pipe2.C* program and its output is:

```
%    CC pipe2.C -lthread -o pipe2
%    pipe2
Have a good day
*> Have a good day
Bye-Bye
*> Bye-Bye
^D
read thread (5 )exists
write thread (4) exists
%
```

13.5.4 Semaphores

Both the Sun thread library and POSIX.1c provide APIs for semaphore manipulation of. These thread-based semaphores are similar to System V semaphores in that each semaphore has an integer value that must be either zero or a positive number. A semaphore value may be set by any thread that has access to it. If a thread attempts to decrement a semaphore value that will result in a negative number, the thread is blocked. It remains this way until another thread increases the semaphore value to a large enough number for the blocked thread's operation on the semaphore to result in a zero or positive value.

The blocking features of semaphores can be used to synchronize the execution of certain code segments or the access of shared data by multiple threads. Before accessing a shared resource, each thread attempts to decrement a semaphore value by 1. Only one of them succeeds, while the rest are blocked. Once the successful thread finishes using the shared resource, it increments the semaphore value to its previous value so that another thread can be unblocked and proceed with accessing the shared resource.

Semaphores can be used in place of mutex locks and condition variables. However, whereas mutex locks can be released only by the threads holding them, any thread can increment or decrement a semaphore to which they have access. To ensure program reliability, additional programming effort is needed by users to keep track of what each thread is doing in regard to a semaphore.

POSIX.1c uses the same semaphore APIs as in POSIX.1b for thread synchronization. Please refer to Chapter 10, Section 10.6 for a description of these APIs. Sun Microsystems, on the other hand, defines a different set of semaphore APIs for thread synchronization. These APIs and their correspondence to the POSIX.1b semaphore APIs are:

Sun API	POSIX.1b API	Use
sema_init	sem_init	Initializes a semaphore
sema_post	sem_post	Increases a semaphore value by 1
sema_wait	sem_wait	Decreases a semaphore value by 1
sema_trywait	sem_trywait	Decreases a semaphore value by 1, but nonblocking
sema_destroy	sem_destroy	Discards a semaphore

The prototypes of the Sun semaphore APIs are:

```
#include <synch.h>

int     sema_init ( sema_t *svp, int init_val, int type, void* argp );
int     sema_wait ( sema_t* rvp );
```

```
#include <synch.h>

int     sema_trywait ( sema_t* rvp );
int     sema_post ( sema_t * svp);
int     sema_destroy ( sema_t* svp );
```

The *svp* argument value is the address of a *sema_t*-typed variable. This variable is set to reference a semaphore via the *sema_init* function. The new semaphore is set to an initial value as specified by the *init_val* argument. The *type* argument of the *sema_init* function specifies whether or not the semaphore is accessible by different processes. Its possible values and meanings are:

***type* value**	**Meaning**
USYNC_PROCESS	The semaphore can be used by threads in different processes
USYNC_THREAD	The semaphore can be used by threads in the calling process only

The *argp* argument of the *sema_init* function is currently unused. Its value should be specified as zero.

The *sema_wait* function attempts to decrement the value of a semaphore as specified by the *svp* argument. This function blocks the calling thread until the operation succeeds.

The *sema_trywait* function is like the *sema_wait* function, except that it is non-blocking. If the function aborts because the requested semaphore value cannot be decreased, it returns a nonzero value and sets the *errno* to EBUSY.

The *sema_post* function increases the value of a semaphore as specified by the *svp* argument.

The *sema_destroy* function discards a semaphore as specified by the *svp* argument.

The following *pipe3.C* program is a rewrite of the *pipe.C* program presented in the last section. The new program uses a semaphore instead of a read-write lock to synchronize the reader and writer threads.

```
/* pipe3.C */
#include <iostream.h>
#include <thread.h>
```

```
#include <string.h>
#include <stdio.h>
#include <signal.h>
#define FINISH()  { cerr << (int)thr_self() << " exits\n";  thr_exit(0); return 0; }
sema_t  semx;
int     msglen, done;
char    msgbuf[256];

/* a write thread function */
void* writer (void* argp )
{
   do   {
      sema_wait(&semx);                    // acquire a semaphore
      if (msglen)   {                      // if a message is in buffer
         cut << "*> " << msgbuf << endl;   // print out the message
         msglen = 0;                       // reset message buffer size
      }
      sema_post(&semx);                    // release a semaphore
      thr_yield();                         // let other threads run
   } while (!done);                        // do until no more messages
   FINISH();                               // clean-up and terminate
}
/* a reader thread function */
void* reader (void* argp )
{
   do   {
      sema_wait(&semx);                    // acquire a semaphore
      if (!msglen)   {                     // check buffer is empty
         if (cin.getline(msgbuf,256))      // get a new message
            msglen = strlen(msgbuf)+1;     // set message size to msglen
         else done = 1;                    // nc more input messages
      }
      sema_post(&semx);                    // release the semaphore
      thr_yield();                         // let other threads run
   } while (!done);                        // do until no more messages
   FINISH();                               // clean-up and terminate
}

/* main thread function */
main()
{
   thread_t wtid, rtid;
   /* initialize a semaphore with an initial value of 1 */
   sema_init(&semx, 1, USYNC_PROCESS, 0);
```

```
    /* create a writer thread */
    if (thr_create(0,0,writer,0,0,&wtid)) perror("thr_create");
    /* create a reader thread */
    if (thr_create(0,0,reader,0,0,&rtid)) perror("thr_create");

    /* wait for all threads to exit */
    while (!thr_join(0,0,0)) ;

    /* clean up */
    sema_destroy(&semx);
    thr_exit(0);
}
```

The above program is similar to the last two examples. The difference here is that a semaphore is used instead of a mutex lock, condition variable, or read-write lock. Specifically, a *semx* semaphore is initialized in the main thread with an initial value of 1. When the reader and writer threads are run, both try to acquire the semaphore via the *sema_wait* function call. Only one of them can succeed.

If the writer thread acquires the semaphore before the reader thread does, it finds the message buffer empty and releases the semaphore. This yields its execution to the reader thread. When the reader thread acquires the semaphore, it reads a message from the user and puts it into the *msgbuf* array. This sets the *msglen* to be the size of the message text. It then releases the semaphore and yields its execution to the writer thread.

When the writer thread is unblocked by the semaphore, it finds that the *msgbuf* is not empty and prints the message contained in it to the standard output. It then resets the *msglen* variable and releases the semaphore. This starts the next round of message processing by the reader and writer threads.

When a reader thread cannot read a message from the standard input (may be due to end-of-file), it sets the *done* variable to 1. This signals both itself and the writer thread to terminate via the *thr_exit* function call.

A sample run of the *pipe3.C* program and its output is:

```
%    CC pipe3.C -lthread -o pipe3
%    pipe3
Have a good day
*> Have a good day
Bye-Bye
*> Bye-Bye
^D
```

```
5 exists
4 exists
```

13.6 Thread-Specific Data

Automatic variables and dynamic memory allocated within a function are owned by each thread that executes that function. Global and static variables can be shared by multiple threads, but they must use mutex locks, conditional variables, etc. to synchronize their access of shared data. Some other global variables, however, cannot be synchronized to maintain a minimum level of concurrency within multithreaded programs. For example, the *errno* variables defined in the C library are set by each system call. Thus, if multiple threads make system calls concurrently, the *errno* variable must be set to different values for different threads. This problem can be worked around by requiring that only one thread at a time make a system call. This renders multithreaded programs to run in single-threaded mode.

The *errno* problem is resolved by the thread library, which automatically creates a private copy of *errno* for each thread that makes system calls. Thus, multiple threads can make system calls and check their *errno* values at the same time.

Another similar problem is when functions contain static variables concurrently accessible by multiple threads. For example, the C library function *ctime* returns the character string of a local date and time. This string is stored in the internal static buffer of the *ctime* function:

```
const char* ctime ( const time_t *timval )
{
    static char timbuf[...];
    /* convert timval to local date/time and store result to timbuf */
    return timbuf;
}
```

Thus, if several threads call the above *ctime* function simultaneously, the function must somehow be able to return different results for different threads. One could resolve the above problem by allocating dynamic buffers to store the requested date/time in each call. However, this causes several problems:

- The function takes more time and memory to execute. This taxes the performance and memory requirement of programs that use this function
- Existing programs that use this function need to be changed to deallocate the dynamic memory returned by this function
- Unchanged single-threaded programs cannot be used in the multithreaded environment.

To solve the above problem, both POSIX.1c and Sun define "re-entrance" versions of popular library functions. These new functions can be called by multiple threads concurrently with no side effects, and their performance is the same as or better than their single-threaded counterparts. The names of these re-entrance functions are the same as their counterparts but with a *_r* suffix. Thus, the re-entrance version of the *ctime* function is called *ctime_r*. All these new functions take one additional argument, which is the address of a variable that holds the returned value. The variable is defined by the calling threads. Thus, multiple threads may call the same function simultaneously and each receives its answer via its supplied variable. Existing or single-threaded programs may continue to use the old library functions and are not affected by the multithreaded environment. New multithreaded programs should use the re-entrance versions of library functions.

The new *ctime_r* function definition is as follows:

```
char* ctime_r (const time_t* timval, char buf[])
{
    /* convert timval to local date/time and store the result to buf */
    ....
    return buf;
}
```

Note that the *ctime_r* function uses its input argument *buf* to store the caller's requested date and time stamp and returns the *buf* address to the caller.

Sun provides re-entrance versions of C, math, and socket library functions.

Even with re-entrance functions and thread library-supported global variables that are defined dynamically for each thread, there may still be a need to have user-defined thread-specific data. For example, users who develop a utility package (for example, a new GUI package) that can be used by other programmers may wish to define their own versions of *errno*. This would allow their users to check *errno* for any error code returned by the utility functions. However, their package functions may be called by multiple threads concurrently, creating a need for their functions to define a per-thread-specific *errno*. Before showing how this is done, the following functions are defined in the Sun thread library for manipulation of thread-specific data:

Function	**Use**
thr_keycreate	Defines a common key for all threads
thr_setspecific	Store a thread value to a key
thr_getspecific	Retrieves a thread value from a key

The prototypes of these functions are:

```
#include <thread.h>

int     thr_keycreate( thread_key_t * keyp, void (* destr)(void*) );
int     thr_setspecific ( thread_key_t key, void* valuep);
int     thr_setspecific( thread_key_t key, void** valuep);
```

The *keyp* argument value is the address of a *thread_key_t*-typed variable. This variable is initialized by the *thr_keycreate* function. The optional *destr* argument is the address of a user-defined function that may be called to discard a thread value. This occurs when a thread that has registered a value with the **keyp* variable terminates. The argument value passed to the *destr* function is the address of a terminating thread value that is registered with the **keyp* via the *thr_setepecific* function.

The *thr_setspecific* function is called by a thread to register a value with a key. The *key* argument specifies which key with which to register. The *valuep* argument contains the address of the thread value. A key may maintain multiple values at any one time, but only one per thread.

The *thr_getspecific* function is called to retrieve the value of a calling that which has been registered with a key designated by the *key* argument. The thread value is returned via the *valuep* argument.

The following *pkg.h* header defines a data class that can be used by multiple threads simultaneously. Specifically, the class defines an *errno* and an *ofstream* object for each thread so that there are no conflicts among threads calling the same class functions. The *pkg.h* file content is:

```
#ifndef PKG_H
#define PKG_H
#include <fstream.h>
#include <stdio.h>
#include <thread.h>

/* record to store a set of thread-specific data */
class thr_data
{
    int             errno;          // errno for a thread
    ofstream&       ofs;            // output stream for a thread
    /* other stuffs */
  public:
    /* constructor */
```

```
        thr_data( int errval, ofstream& os) : errno(errval), ofs(os) {};

        /* destructor */
        ~thr_data()         { ofs.close(); };

        /* return a thread's errno */
        int& errval()       { return errno; };

        /* return a thread's ostream handle */
        ofstream& os()      { return ofs;  };

        /* other member functions */
};

/* Utility package class */
class Xpackage
{
        thread_key_t        key;                    // key for all threads
        ofstream            ocerr;                  // default output stream
        /* other package data */

    public:

        /* called when a thread dies. Discard a thread-specific data */
        friend void destr( void* valueP )
        {
            thr_data  *pDat = (thr_data*)valueP;
            delete pDat;
        };

        /* constructor */
        Xpackage() : ocerr("err.log")
        {
            if (thr_keycreate(&key,destr)) perror("thr_create");
        };

        /* destructor */
        ~Xpackage()         { ocerr.close(); };
        /* called when each thread starts */
        void new_thread( int errval,  ofstream& os )
        {
            thr_data      *pDat;
            pDat = new thr_data(errval, os);        // alloc a thread-specific data
            if (thr_setspecific(key,pDat)) perror( "thr_setspecific");
        };
```

```
        /* set a thread's errno and return a value */
        int set_errno( int rc )
        {
           thr_data      *pDat;
           if (thr_getspecific(key,(void**)&pDat))
              perror("thr_getspecific");
           else pDat->errval() = rc;
           return rc==0 ? 0 : -1;
        };

        /* return current errno value for a thread */
        int errno()
        {
           thr_data      *pDat;
           if (!thr_getspecific(key,(void**)&pDat))
              return pDat->errval();
           else perror("thr_getspecific");
           return -1;
        };

        /* return a thread's outstream handle */
        ofstream& os()
        {
           thr_data      *pDat;
           if (!thr_getspecific(key,(void**)&pDat)) return pDat->os();
           perror("thr_getspecific");
           return ocerr;
        };

        /* a sample package function */
        int chgErrno(int new_val )
        {
              return set_errno( new_val +int(thr_self() );
        };
        /* other package functions */
};
#endif /* PKG_H */
```

All thread-specific data are stored in a *thr_data*-typed record. The record stores *errno* and *ofstream* objects that are private for each thread. Users may redefine the *thr_data* class to contain additional thread-specific data if desired.

There should be a *Xpackage*-typed variable defined globally in every user program. When the program starts up, the *Xpackage::Xpackage()* constructor is called, which initial-

izes the *Xpackage::key* variable shared by all threads in the process. The *Xpackage::ocerr* variable references a default output stream in case a thread-defined output stream cannot be opened. If a user wants to store different types of thread-specific data in a process, he or she may define multiple *Xpackage*-typed variables, one for each type of thread-specific data.

The *Xpackage::set_errno()* is called by the package functions to set the *errno* value for a calling thread. The *Xpackage::errno()* is called by a thread to retrieve its private *errno* value. The *Xpackage::os()* is called to return an output stream object for a thread.

The *Xpackage::chgErrno()* is a sample package function that performs useful tasks. In this example, however, the *Xpackage::chgErrno()* function does nothing but set the per-thread *errno* value to be the thread ID plus 100. Like all other defined package functions, it returns a 0 value if succeeds or a -1 value otherwise (the thread-specific *errno* is set with an error code accordingly).

The *destr* function is called whenever a thread terminates. The *destr* function discards thread-specific *thr_data*-typed data. If a thread has registered multiple *thr_data*-typed records with multiple *Xpackage*-typed variables, then the *destr* function is called multiple times, once for each *thr_data*-typed data belonging to the terminating thread.

The following *thr_errno.C* file depicts a sample user program that makes use of the *Xpackage* class. The file content is:

```
#include "pkg.h"
Xpackage pkgObj;                              /* a package object */

/* function executed by each thread */
void* func1( void* argp )
{
   int *rcp = new int(1);

   /* open a thread's outstream */
   ofstream ofs ((char*)argp);
   if (!ofs) thr_exit((void**)&rcp);

   /* initialize a thread-specific data */
   pkgObj.new_thread( 0, ofs );

   /* do work with package functions here */
   pkgObj.chgErrno( 100 ); /* change  a thread's errno */

   /* write some data to a thread's outstream */
   pkgObj.os() << (char*)argp << " [" << (int)thr_self() << "] finishes\n";
```

```
    /* thread terminates, set exit code */
    *rcp = pkgObj.errno();
    thr_exit(rcp);
    return 0;
}

/* main thread's function */
int main(int argc, char** argv)
{
    thread_t    tid;
    int         *rc;

    /* create a thread for each command line argument.*/
    while (--argc > 0)
        if (thr_create(0,0,func1,(void*)argv[argc],0,&tid)) perror("thr_create");

    /* wait for all threads to terminate */
    while (!thr_join(0,&tid,(void**)&rc))    {
        cerr << "thread: " << (int)tid << " exists. rc=" << *rc << endl;
        delete rc;                          // delete thread's dynamic mem
    }

    /* terminate the main thread */
    thr_exit(0);
    return 0;
}
```

The program is invoked with one or more file names as command line arguments. Each argument is the file name of a thread output stream. For each argument, the main thread creates a new thread to execute the *func1* function. The argument to the *func1* function is the new thread's output stream file name. After all the threads are created, the main thread waits for them to exit and then terminates.

When a thread starts executing the *func1* function, it defines an *ofstream* object to reference the given output file. It then calls the *pkgObj.new_thread()* to allocate its thread-specific data storage, which stores the *ofstream* object and initializes the *errno* value to zero. When these are done, the thread calls the package functions to perform actual work. At the end it calls the *pkgObj.chgErrno()* function to set its *errno* value to 100 plus its thread ID value. This causes the *errno* value of each thread that executes the *func1* function to be unique for each thread. The thread then calls the *pkgObj.os()* function to return its output stream object and also prints the stream object output file name and thread ID. The thread terminates by specifying its private *errno* value as the actual argument to the *thr_exit* function call.

The sample run of the *thr_errno* program and its output is:

```
%    CC thr_errno.C -o thr_errno -lthread
%    thr_errno    a    b
thread: 5 exists. rc=105
thread: 4 exists. rc=104
%    cat a
a [5] finishes
%    cat b
b [4] finishes
```

In the above sample run, the *thr_errno* program is invoked with two file names: *a* and *b*. Two threads are created to execute the *func1* functions. The first thread ID is 5 and it creates an output file called *a* with the content of *a [5] finishes*. The thread terminates with an exit value of 104. The second thread ID is 4. It creates a file called *b* with the content of *b [5] finishes* and its exit value is 105.

13.7 The Multithreaded Programming Environment

To support multithreaded programming, Sun Solaris provides a thread library for users to create and manipulate threads in their programs. There are also modifications in the standard libraries such that there are re-entrance versions of many popular library functions. Global variables (e.g., *errno*) that are exported from standard libraries are defined dynamically for each thread that uses them. All of these ensure that multiple threads make use of standard libraries concurrently with reliable results.

Besides all the above, Sun Solaris also modifies the kernel to support symmetrical multi-processing and LWP scheduling. There is also a multithreaded version of the *debugger* and *truss* commands which debug and trace individual thread activities in a process. All of these features are expected to be provided by other vendors that support multithreaded programming environments on their systems.

13.8 Distributed Multithreaded Application Example

This section describes a multithreaded distributed program. This is an interactive program that executes user shell commands on any machine on a LAN. The program uses RPC to communicate with dedicated RPC servers running on remote hosts. When a user issues a shell command, the program creates a thread to connect with an RPC server on a user-specified host. The server executes the user command on that host. and its return status code is checked by the thread to flag any error code to the user.

By using a thread to handle each user command, the program can accept a new command while it is executing one or more previous commands by other threads. Thus users do not have to wait for each command to finish execution before issuing another one. This makes the program more "interactive" than an equivalent program that is single-threaded. Furthermore, the program can readily make use of any multi-processor resources that may be available on its the host machine.

By using RPC, the program can distribute its work loads to other computers on the network. This greatly enhances the performance and flexibility of the program. Furthermore, by using the RPC broadcasting technique, the program can automatically determine which hosts have its RPC servers running for communication purposes. Thus, the only setup required for users is to load the RPC servers on host machines of their choice (which may be heterogeneous platforms such as UNIX workstations, VMS machines and Windows-NT machines) and run the program on a host machine that supports multithreaded programs (e.g., a Sun Solaris workstation).

The RPC server that communicates with the interactive program and executes user shell commands is contained in the *shell_svc.C* program. This program uses the RPC classes as defined in Chapter 12 to create a *RPC_svc* object for RPC operations. The *shell_svc.C* file is:

```
#include "mshell.h"
#include "RPC.h"
RPC_svc *svcp;                                  // RPC server object pointer

/* RPC function to execute one user's shell command */
int execshell( SVCXPRT* xtrp )
{
   static int  res=0, rc= RPC_SUCCESS;
   char *shell_cmd = 0;

   /* get user's shell command from a RPC client */
   if (svcp->getargs( xtrp, (xdrproc_t)xdr_string, (caddr_t)&shell_cmd)
         !=RPC_SUCCESS)
      return -1;

   /* execute the command via the system function */
   res = system(shell_cmd);

   /* send execution result to the RPC client */
   if (svcp->reply(xtrp, (xdrproc_t)xdr_int, (caddr_t)&res)!=RPC_SUCCESS)
        rc =  -2;

   return rc;
}
```

```
/* RPC server main function */
int main(int argc, char* argv[])
{
   /* create a RPC server object for the execshell function */
   RPC_svc *svcp = new RPC_svc( SHELLPROG, SHELLVER,
            argc==2 ? argv[1] : "netpath");
   if (!svcp || !svcp->good()) return 1;
   /* wait for RPC clients' requests */
   if (svcp->run_func( EXECSHELL, execshell )) return 3;
   return 0;                                   /* shouldn't get here */
}
```

The RPC server program creates an *RPC_svc* object to execute the *execshell* function when requested by a RPC client. The *execshell* function takes a NULL-terminated character string from a client that contains a UNIX shell command. It calls the *system* function to execute that shell command. The return value of the *system* function call is returned to the RPC client via the *RPC_svc::reply* function. The RPC server process is a daemon and is run in the background until the system is shut down or is explicitly killed by a user.

The RPC.h header and its companion RPC.C files are defined in Section 12.5 of Chapter 12. The following *mshell.h* header defines the RPC program number, version number, and function number for the *execshell* function:

```
#ifndef MSHELL_H
#define MSHELL_H
#include <rpc/rpc.h>

#defineSHELLPROG ((unsigned long)(0x20000001))
#defineSHELLVER  ((unsigned long)(1))
#defineEXECSHELL ((unsigned long)(1))

#endif /* !MSHELL_H */
```

The RPC server program is compiled and run as follows on any computer that supports UNIX System V Release 4-style RPC functions:

```
%   CC -DSYSV4 shell_svc.C RPC.C -o shell_svc -lnsl
%   shell_svc &
```

The interactive main program is *main_shell.C*. It is a menu-driven program that repeatedly displays a menu of selections to the standard output. A user enters a selection by the menu index, and the program executes that selection via a thread. The *main_shell.C* program is:

```
#include <strstream.h>
#include <thread.h>
#include <string.h>
#include <stdio.h>
#include "mshell.h"
#include "shellObj.h"

/* collect remote hosts that have server running on it */
extern void* collect_hosts( void* argp );
extern void* display_hosts( void* argp );
extern void* exec  shell( void* argp );

shellObj *funcList[4];                          // dispatch table
int   numFunc = sizeof(funcList)/sizeof(funcList[0]);
rwlock_t   rwlck;                               // read-write lock

/* Quit program */
void* quit_prog( void* argp )
{
   return (void*)0;
}

/* Execute one shell command on a host */
void* getcmd( void* argp )
{
   char host[20], cmd[256], cmd2[256];
   cout << "shell cmd> " << flush;
   cin >> cmd;
   cin.getline(cmd2,256);
   cout << "host: " << flush;
   cin >> host;
   if (!cin)
      cout << "Inavlid input\n" << flush;
   else
   {
      if (strlen(cmd2)) strcat(strcat(cmd," "),cmd2);
      strcat(strcat(cmd,"/"),host);
      char* ptr = new char[strlen(cmd)+1];
      ostrstream(ptr,strlen(cmd)+1) << cmd;
      if (thr_create(0,0,exec_shell,ptr,THR_DETACHED,0)) per-
ror("thr_create");
   }
   return (void*)1;
}
```

```
/* Get an user input selection and execute it */
int exec_cmd()
{
   char buf[256];
   cout << "Selection> " << flush;
   cin >> buf;
   cout << endl;
   if (cin)
   {
      int  idx = -1;
      istrstream(buf) >> idx;
      if (idx >=0 && idx < numFunc)
         return funcList[idx]->doit(0);
      else cerr << "Invalid input\n";
   }
   return 0;
}

/* display menu to user */
void display_menu(ostream& os)
{
   for (int i=0; i < numFunc; i++)
      (void)funcList[i]->usage(os,i);
}

/* initialize read-write lock and dispatch table */
int init_all()
{
   /* initialize the read-write lock */
   if (rw_init (&rwlck, USYNC_THREAD, 0))
   {
      perror("rwlock_init");
      return -1;
   }
   /* initialize the dispatch table */
   funcList[0] =  new shellObj( "Collect host names", collect_hosts);
   funcList[1] =  new shellObj( "Display host names", display_hosts);
   funcList[2] =  new shellObj( "Execute a shell command"  getcmd, 0);
   funcList[3] =  new shellObj( "Quit", quit_prog, 0);
   return 0;
}

/* main routine */
int main()
{
   if (init_all() == 0)
```

```
        do  {
            display_menu(cerr);
            if (!exec_cmd()) break;
        } while (1);
    thr_exit(0);
}
```

In this interactive program, there are four menu selection to users:

Index	Function	Usage
0	*collect_hosts*	Finds all hosts running the RPC server
1	*display_hosts*	Displays all host names running the RPC server
2	*getcmd*	Execute some shell command
3	*quit_prog*	Quits the interactive program

The above four functions' addresses are stored in a dispatch table *funcList*. When a user enters a selection, that integer value is used to index the dispatcher table and to invoke the function for that entry. For example, if a user enters a selection of zero, the *collect_hosts* function is invoked.

Each entry of the *funclist* table is a pointer to a *shellObj* object. The *shellObj* class is defined in the *shellObj.h* header as:

```
#ifndef SHELLOBJ_H
#define SHELLOBJ_H

#include <iostream.h>
#include <thread.h>
#include <string.h>
typedef void* (*FNPTR)( void *);

class shellObj
{
        char*       help_msg;
        FNPTR       action;
        void*       (*fnptr)( void* );
        int         create_thread;
    public:
        // constructor. Set help msg, action func and errno
        shellObj( const char* msg, FNPTR func, int thr_ok=1)
        {
            help_msg = new char[strlen(msg)+1];
            strcpy(help_msg,msg);
```

```
        action  = func;
        create_thread = thr_ok;
    };
    // destructor function
    ~shellObj() { delete help_msg; };

    // print object usage
    ostream& usage( ostream& os, int idx)
    {
        os << idx << ": " << help_msg << endl;
        return os;
    };

    // do action
    int doit( void* argp )
    {
        if (create_thread) {
            thread_t  tid;
            if (thr_create(0,0,action,argp,THR_DETACHED,&tid))
                perror("thr_create");
            return (int)tid;
        }
        else return (int)action(argp);
    };
};
#endif /* !SHELLOBJ_H */
```

Each *shellObj* object contains a help message that explains its use, a function pointer to a user-defined function (which may be called to do the actual work of the object), and a flag to specify whether or not the user-defined function should be invoked via a thread when it is called. The four *shellObj* objects for the four functions *collect_hosts, display_hosts, getcmd* and *quit_prog* are created and referenced via the *funcList* table entry 0, 1, 2, and 3, respectively.

The *main* function of the interactive program first calls the *init_all* function to initialize: (1) the read-write lock *rwlck*; and (2) the *funcList* dispatch table with four *shellObj* objects. After that, the main function goes into a loop, calling the *display_menu* function to display a selection menu to a user. It next calls the *exec_cmd* function to prompt a user for a menu selection and invokes a function for that selection. The loop exits when the *exec_cmd* returns a zero value and the *main* function terminates via the *thr_exit* function.

The *display_menu* function scans through the *funcList* dispatch table and invokes the *shellObj::usage()* function of each *shellObj* object addressed by table entry. This causes each *shellObj* object to print out its use and dispatch table index to the output stream *cerr*.

Each user selection to the menu is obtained via the *exec_cmd* function. The function checks that the user input index is in the range of zero to 3 and flags an error if this is not true. If user input is valid, the *exec_cmd* function uses that number as an index to the *funcList* table to obtain a pointer to a *shellObj* object. It then invokes the *shellObj::doit()* function of that object. The *exec_cmd* function returns either a zero value, if it cannot obtain data from a user, or the return value of the *shellObj::doit()* function that it invoked.

The *exec_cmd* function passes the 0 argument value to each *shellObj::doit()* function it invokes. For the menu selection of 0 and 1, the corresponding *collect_hosts* and *display_hosts* functions are executed by detached threads immediately. However, for the menu selections of 2 and 3, the corresponding *getcmd* and *quit_prog* functions are not executed by detached threads. Each of these four functions returns a nonzero value if its execution is successful, a zero value if it fails.

The threads created to execute the *collect_hosts* and *display_hosts* functions are detached from the main threads. This means that as soon as they terminate, their resource and thread IDs can be reused by other new threads. This is so because the interactive program may run for a long time and must maintain its responsiveness to user inputs. It cannot afford to suspend itself in any *thr_join* function call to wait for other threads to exit.

There is no argument value passed to any *collect_hosts* and *display_hosts* function calls. The *getcmd* function gets an input shell command and a host name from a user. It then creates a thread to execute the *exec_shell* function. The actual value passed to the *exec_shell* function is a NULL-terminated character string that contains a shell command, followed by a "/" and finally a host name where the command is to be executed. Thus, if an actual argument passed to an *exec_shell* function is:

```
"cal 1995 > foo/fruit"
```

it asks to execute the *cal 1995 > foo* shell command on the host *fruit*. The list of available hosts that have the *shell_svc* daemon installed are collected by the *collect_hosts* functions, and the list is displayed to the user via the *display_hosts* function.

The *collect_hosts, display_hosts,* and *exec_shell* function definitions are contained in the *shell_cls.C* file:

```
#include <fstream.h>
#include <strstream.h>
#include <stdio.h>
#include <netdir.h>
#include <string.h>
#include <thread.h>
#include "mshell.h"
#include "RPC.h"
```

```
#define TMPFILE   "/tmp/hosts"

/* read-write lock to synchronize the access to the hostlist table by threads */
extern rwlock_t rwlck;                  // defined and initialized in main_shell.C

/* maintain a list of available hosts */
#define MAXHOSTS  30
static char *hostlist[MAXHOSTS];        // list of host names
static int  numhosts;                   // actual no. of hosts

/* make a RPC call to a server to execute one shell command */
int exec_host( const char* cmd, char* host )
{
   int   res=0;
   /* create a client handle to connect to the RPC server */
   RPC_cls cl( host, SHELLPROG, SHELLVER, "netpath");
   if (!cl.good()) return 1;
   /* authenticate client tothe RPC server */
   cl.set_auth( AUTH_SYS );

   /* call the execshell function on the server */
   if (cl.call( EXECSHELL, (xdrproc_t)xdr_string, (caddr_t)&cmd,
                (xdrproc_t)xdr_int, (caddr_t)&res) != RPC_SUCCESS)
      return 2;

   // flag an error if execshell fails
   if (res!=0)  cerr << "clnt: exec cmd fails\n";
   return res;
}

/* check if a named host is in the hostlist table */
int check_host( const char* hostnm )
{
   if (rw_rdlock(&rwlck)) perror("rw_rdlock");      // acquire lock for read

   for (int i=0; i < numhosts; i++)
      if (!strcmp(hostlist[i],hostnm)) break;       // break if name found

   if (rw_unlock(&rwlck)) perror("rw_unlock");      // release read lock

   return  (i < numhosts) ? 1 : 0;                  // return 1 is name is OK
}

/* Executed by a thread created by func3: exec one command on a host */
void* exec_shell( void* argp )
{
```

```
    int rc = 0;
    char* cmd = (char*)argp;
    char* host = strrchr(cmd,'/');
    *host++ = '\0';
    if (!check_host(host))        {                    // check host name is OK
       cout << "Invalid host: '" << host << "'\n" << flush;
       rc = 1;
       thr_exit(&rc);                                  // exit with an error
    }
    rc = exec_host( cmd, host );
    thr_exit(&rc);                                     // exit thread
    return 0;
}

/* Executed by a thread called by func2: display all available hosts */
void* display_hosts( void* argp )
{
    int rc = 0;
    char buf[256], cmd[256];

    if (rw_rdlock(&rwlck)) perror("rw_rdlock");        // acquire a read lock

    if (!numhosts)                                     // NOP if no hosts
       rc = -1;
    else {
       /* store the hosts listing to a temporary file */
       ostrstream(buf,256) << TMPFILE << "." << thr_self();
       ofstream ofs (buf);
       if (!ofs)
          cerr << "Create temp file '" << buf << "' failed\n";
       else     {
          for (int i=0; i < numhosts; i++)
             ofs << i << ":  " <<  hostlist[i] << endl;
          ofs.close();

          /* pop-up a xterm to display the host listing */
          ostrstream(cmd,256) << "xterm -title Hosts -e view " << buf;
          if (system(cmd)) perror("system");

          /* remove the temporary file */
          if (unlink(buf)) perror("unlink");
       }
    }

    if (rw_unlock(&rwlck)) perror("rw_unlock");        // release the read lock
    thr_exit(&rc);                                     // terminate the thread
```

```
   if (rw_unlock(&rwlck)) perror("rw_unlock");        // release the read lock
   thr_exit(&rc);                                    // terminate the thread
   return 0;
}


/* record a remote host name to the hostlist table */
int add_host( const char* hostnm )
{
   int new_entry = 1;                                // success return code
   if (rw_wrlock(&rwlck)) perror("rw_wrlock");       // acquire write lock

   for (int i=0; i < numhosts; i++)                  // is name in table?
      if (!strcmp(hostlist[i],hostnm)) break;

   if (i >= numhosts)     {                          // name not in list
      if (numhosts >= MAXHOSTS)
         cerr << "Too many remote hosts detected\n";
      else     {                                     // add a new entry
         hostlist[numhosts] = new char[strlen(hostnm)+1];
         strcpy(hostlist[numhosts++],hostnm);
      }
   }
   else new_entry = 0;                               // failure return code
   if (rw_unlock(&rwlck)) perror("rw_unlock");       // release write lock
   return new_entry;
}


/* client's broadcast call back function */
bool_t callme (caddr_t res_p, struct netbuf* addr, struct netconfig *nconf)
{
   struct nd_hostservlist *servp;

   /* extract server's hostname(s) from the addr argument */
   if (netdir_getbyaddr(nconf,&servp,addr))
      perror("netdir_getbyaddr");

   else for (int i=0; i < servp->h_cnt; i++) // add host names to hostlist table
      if (!add_host( servp->h_hostservs[i].h_host ))
         return TRUE;              /* end broadcast if found a host twice */

   return FALSE;                             // get more servers' responses
}
```

```
    int rc = RPC_cls::broadcast( SHELLPROG, SHELLVER, 0,
                (resultproc_t)callme,          (xdrproc_t)xdr_void,
                (caddr_t)NULL, (xdrproc_t)xdr_void, (caddr_t)NULL);

    /* check broadcast results */
    switch (rc) {
      case RPC_SUCCESS:                          // successful
        break;
      case RPC_TIMEDOUT:                         // time-out
        if (numhosts) break;
      default:          /* flag an error if no hosts responded */
        cerr << "RPC broadcast failed\n";        // fail
        rc = 1;
    }
    thr_exit(&rc);                               // terminate the thread
    return 0;
}
```

When the *collect_hosts* function is invoked, it makes an RPC broadcast call to ping all *shell_svc* daemons on the network. Each daemon response to the RPC broadcast is registered by the *callme* function. This function extracts the daemon's host name via the *netdir_getbyaddr* function and adds that host name to the *hostlist* table via the *add_host* function. The *add_host* function returns a value of 1 if it successfully adds a new host name to the *hostlist* table, a zero value otherwise. The *callme* function terminates the RPC broadcast if it sees that the same host responds to the broadcast twice. This would indicate that the RPC broadcast has not received any new responses and is in the rebroadcast process.

Once the RPC broadcast is finished, the *collect_hosts* function checks the broadcast result and terminates its thread with a return value of RPC_SUCCESS for success and a non-zero value for failure.

Note that access to the *hostlist* table is guarded by the read-write lock *rwlck*. This is needed as the *hostlist* table is accessed by all threads created to execute the *collect_hosts*, *display_hosts*, and *exec_shell* functions. It is important to ensure that the writer threads (executing the *collect_hosts* function) do not access the *hostlist* table simultaneously with the reader threads (executing the *display_hosts* and *exec_shell* functions)

When the *display_hosts* function is invoked, it displays all the host names in the *hostlist* table. These are the hosts running the *shell_svc* daemon. The function acquires and releases the read-write lock before and after it accesses the *hostlist* table. This is to ensure that the *collect_host* threads do not modify the table while reading it.

To make the host listing output separate from the main thread menu display, the *display_hosts* function stores the host listing to a temporary file and invokes an *xterm* to exe-

cute the *view <temp_file>* command on a separate window. The window disappears when a user quits the *view* command and the temporary file is discarded. The function terminates the thread with a exit value of 0 if it succeeds, a nonzero value if it fails.

When the *exec_shell* command is invoked, its input argument is a NULL-terminated character string that contains a user-defined shell command and a host name. The function calls the *check_host* command to make sure that the user-specified host name is in the *hostlist* table. If it is not, the thread terminates with a 1 value (failure). On the other hand, if a user-specified host name is valid, the function calls the *exec_host* function, which, in turn, creates an *RPC_cls* object to connect to the *shell_svc* daemon on the given host. Furthermore, the *exec_host* function calls the *execshell* RPC function via the *RPC_cls* object to execute the user shell command on that host. The *exec_host* and the *exec_shell* thread terminate with a zero value if the remote shell execution is successful or a nonzero value otherwise.

The interactive program is made up of the *main_shell.C* and *shell_cls.C* files. They are compiled as follows and the sample output of its execution is shown below. For this example, the hosts that have the *shell_svc* daemon installed are *fruit* and *veggie*.

```
%    CC -DSYSV4 shell_cls.C main_shell.C -o shell_cls -lthread -lnsl
%    shell_cls
0: Collect hosts names
1: Display hosts names
2: Execute a command
3: Quit
Selection> 0

0: Collect hosts names
1: Display hosts names
2: Execute a command
3: Quit
Selection> 1
< output shown in a xterm window: 0: fruit 1: veggie >

0: Collect hosts names
1: Display hosts names
2: Execute a command
3: Quit
Selection> 2

shell cmd> cal 1995 > foo
host> fruit
```

```
0: Collect hosts names
1: Display hosts names
2: Execute a command
3: Quit
Selection> 3
```

13.9 Summary

This chapter described multithreaded programming techniques based on the Sun Microsystems Solaris thread library and the POSIX.1c standard. Specifically, both Sun and POSIX1.c provide a set of thread library functions for users to create and manipulate threads in their applications. Various synchronization objects like mutex locks, condition variables and semaphores are also provided for users to synchronize thread access of shared data in the same process.

Other system support of multithreaded programs include special APIs for modifying per-thread base signal mask and provide re-entrance versions of major library functions. All of these thread-specific environment supports are also expected to be available in other platforms that support multithreaded programs.

Multithreaded programming is particularly useful for multiprocessing and object-oriented applications. This is demonstrated in the last section, where a distribution and multithreaded interactive program is depicted. This example program provides a framework for users to create their own applications, making use of any multiprocessing or network computing resources available on the machines running their applications.

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z