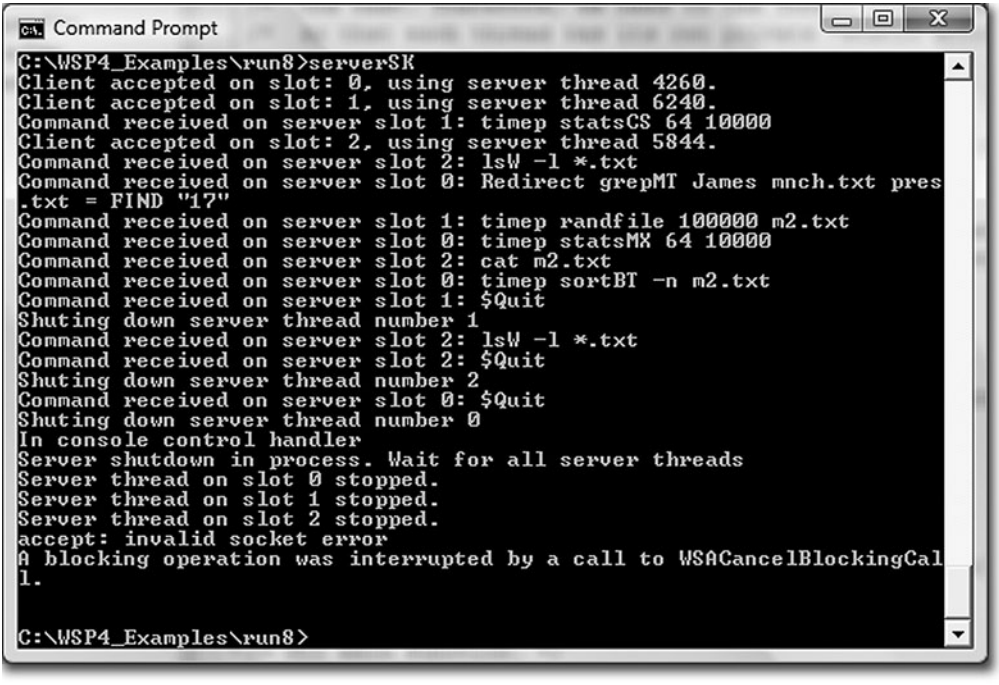
```
    EnterCriticalSection(&(pThArg->threadCs));
    __try {
        threadState = pThArg->thState = SERVER_THREAD_STOPPED;
    }
    __finally { LeaveCriticalSection(&(pThArg->threadCs)); }

    return threadState;
}
```

Running the Socket Server

Run 12–3 shows the server in operation, with several printed information messages that are not in the listings for Programs 12–2 and 12–3. The server has several clients, one of which is the client shown in Run 12–1 (slot 0).

The termination at the end occurs in the accept thread; the shutdown closes the socket, causing the `accept` call to fail. An exercise suggests ways to make this shutdown cleaner.



Run 12–3 `serverSK`**:** Requests from Several Clients

A Security Note

This client/server system, as presented, is *not* secure. If you are running the server on your computer and someone else knows the port and your computer name, your computer is at risk. The other user, running the client, can run commands on your computer that could, for example, delete or modify files.

A complete discussion of security solutions is well beyond this book's scope. Nonetheless, Chapter 15 shows how to secure Windows objects, and Exercise 12–15 suggests using Secure Sockets Layer (SSL).

In-Process Servers

As mentioned previously, in-process servers are a major enhancement in `serverSK`. Program 12–4 shows how to write a DLL to provide these services. Two familiar functions are shown, a word counting function and a `toupper` function.

By convention, the first parameter is the command line, while the second is the name of the output file. Beyond that, always remember that the function will execute in the same thread as the server thread, so there are strict requirements for thread safety, including but not limited to the following:

- The functions should not change the process environment in any way. For example, if one of the functions changes the working directory, that change will affect the entire process.
- Similarly, the functions should not redirect standard input or output.
- Programming errors, such as allowing a subscript or pointer to go out of bounds or the stack to overflow, could corrupt another thread or the server process itself. More generally, the function should not generate any unhandled exception because the server will not be able to do anything other than shut down.
- Resource leaks, such as failing to deallocate memory or to close handles, will ultimately affect the server application.

Processes do not have such stringent requirements because a process cannot normally corrupt other processes, and resources are freed when the process terminates. A typical development methodology, then, is to develop and debug a service as a process, and when it is judged to be reliable, convert it to a DLL.

Program 12–4 shows a small DLL library with two simple functions, `wcip` and `toupperip`, which have functionality from programs in previous chapters. The code is in C, avoiding C++ decorated names. These examples do not support Unicode as currently written.

Program 12–4 `commandIP`: Sample In-Process Servers

```
/* Chapter 12. commands.c                    */
/*                                           */
/* "In Process Server" commands to use with serverSK, etc.*/
/*                                           */
/* There are several commands implemented as DLLs*/
/* Each command function must be a thread-safe function */
/* and take two parameters. The first is a string:*/
/* command arg1 arg2 ... argn (i.e.; a RESTRICTED command*/
/* line with no spaces or quotes in the command or args)*/
/* and the second is the file name for the output*/
/* The code is C, without decorated names*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

static void ExtractToken (int, char *, char *);

__declspec (dllexport)
int __cdecl wcip (char * command, char * output_file)
/* word count; in process (ONE FILE ONLY)*/
/* Count the number of characters, works, and lines in*/
/* the file specified as the second token in "command"*/
/* NOTE: Simple version; results may differ from wc utility*/
{
   FILE * fIn, *fOut;
   int ch, c, nl, nw, nc;
   char inputFile[256];

   ExtractToken (1, command, inputFile);

   fIn = fopen (inputFile, "r");
   if (fIn == NULL) return 1;

   ch = nw = nc = nl = 0;
   while ((c = fgetc (fIn)) != EOF) {
      if (c == '\0') break;
      if (isspace(c) && isalpha(ch))
         nw++;
      ch = c;
      nc++;
      if (c == '\n')
         nl++;
   }
   fclose (fIn);
```

```
    /* Write the results */
    fOut = fopen (output_file, "w");
    if (fOut == NULL) return 2;
    fprintf (fOut, " %9d %9d %9d %s\n", nl, nw, nc, inputFile);
    fclose (fOut);
    return 0;
}

__declspec (dllexport)
int __cdecl toupperip (char * command, char * output_file)
/* convert input to upper case; in process*/
/* Input file is the second token ("toupperip" is the first)*/
{
    FILE * fIn, *fOut;
    int c;
    char inputFile[256];

    ExtractToken (1, command, inputFile);

    fIn = fopen (inputFile, "r");
    if (fIn == NULL) return 1;
    fOut = fopen (output_file, "w");
    if (fOut == NULL) return 2;

    while ((c = fgetc (fIn)) != EOF) {
        if (c == '\0') break;
        if (isalpha(c)) c = toupper(c);
        fputc  (c, fOut);
    }
    fclose (fIn); fclose (fOut);
    return 0;
}

static void ExtractToken (int it, char * command, char * token)
{
    /* Extract token number "it" (first token is number 0)*/
    /* from "command". Result goes in "token"*/
    /* tokens are white space delimited*/
    . . . (see the Examples file
    return;
}
```

Line-Oriented Messages, DLL Entry Points, and TLS

`serverSK` and `clientSK` communicate using messages, where each message is composed of a 4-byte length header followed by the message content. A common alternative to this approach is to have the messages delimited by null characters.

The difficulty with delimited messages is that there is no way to know the message length in advance, and each incoming character must be examined. Receiving a single character at a time would be inefficient, however, so incoming characters are stored in a buffer, and the buffer contents might include one or more null characters and parts of one or more messages. *Buffer contents and state must be retained between calls to the message receive function.* In a single-threaded environment, static storage can be used, but multiple threads cannot share the same static storage.

In more general terms, we have a *multithreaded persistent state problem.* This problem occurs any time a thread-safe function must maintain information from one call to the next. The Standard C library `strtok` function, which scans a string for successive instances of a token, is a common alternative example of this problem.

Solving the Multithreaded Persistent State Problem

The first of this chapter's two solutions to the persistent state problem uses a combination of the following components.

- A DLL for the message send and receive functions.
- An entry point function in the DLL.
- Thread Local Storage (TLS, Chapter 7). The DLL index is created when the process attaches, and it is destroyed when the process detaches. The index number is stored in static storage to be accessed by all the threads.
- A structure containing a buffer and its current state. A structure is allocated every time a thread attaches, and the address is stored in the TLS entry for that thread. A thread's structure is deallocated when the thread detaches.
- This solution does have a significant limitation; you can only use one socket with this library per thread. The second solution, later in the chapter, overcomes this limitation.

The TLS, then, plays the role of static storage, and each thread has its own unique copy of the static storage.

Example: A Thread-Safe DLL for Socket Messages

Program 12–5 is the DLL containing two character string ("`CS`" in names in this example) or socket streaming functions: `SendCSMessage` and `ReceiveCS-Message`, along with a `DllMain` entry point (see Chapter 5). These two functions are similar to and essentially replace `ReceiveMessage`, listed earlier in this chapter, and the functions used in Programs 12–1 and 12–2.

The `DllMain` function is a representative solution of a multithreaded persistent state problem, and it combines TLS and DLLs. The resource deallocation in the `DLL_THREAD_DETACH` case is especially important in a server environment; without it, the server would eventually exhaust resources, typically resulting in either failure or performance degradation or both. *Note:* This example illustrates concepts that are not directly related to sockets, but it is included here, rather than in earlier chapters, because this is a convenient place to illustrate thread-safe DLL techniques in a realistic example.

If this DLL is to be loaded dynamically, you must load it before starting any threads that use the DLL; otherwise, there will not be a `DLL_THREAD_ATTACH` call to `DllMain`.

The *Examples* file contains client and server code, slightly modified from Programs 12–1 and 12–2, that uses this DLL.

Program 12–5 `SendReceiveSKST:` Thread-Safe DLL

```
/* SendReceiveSKST.c -- Multithreaded streaming socket DLL. */
/* Messages are delimited by null characters ('\0') */
/* so the message length is not known ahead of time. Incoming */
/* data is buffered and preserved from one function call to */
/* the next. Therefore, use Thread Local Storage (TLS) */
/* so that each thread has its own private "static storage." */

#include "Everything.h"
#include "ClientServer.h"/* Defines MESSAGE records. */

typedef struct STATIC_BUF_T {
/* "staticBuf" contains "staticBufLen" characters of residual data */
/* There may or may not be end-of-string (null) characters */
   char staticBuf[MAX_RQRS_LEN];
   LONG32 staticBufLen;
} STATIC_BUF;

static DWORD tlsIndex = TLS_OUT_OF_INDEXES; /* Initialize TLS index */
/* A single threaded library would use the following:
static char staticBuf[MAX_RQRS_LEN];
static LONG32 staticBufLen;
*/

/* number of attached, detached threads and processes. */
static volatile long nPA = 0, nPD = 0, nTA = 0, nTD = 0;

/* DLL main function. */
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason,
                    LPVOID lpvReserved)
```

```
{
    STATIC_BUF * pBuf;

    switch (fdwReason) {
        case DLL_PROCESS_ATTACH:
            tlsIndex = TlsAlloc();
            InterlockedIncrement (&nPA);
            /* There is no thread attach call for other threads created
               BEFORE this DLL was loaded.
               Perform thread attach operations during process attach
               Load this DLL before creating threads that use DLL. */

        case DLL_THREAD_ATTACH:
            /* Indicate that memory has not been allocated  */
            InterlockedIncrement (&nTA);
            /* Slots are initialized to 0 */
            return TRUE; /* This value is ignored */

        case DLL_PROCESS_DETACH:
            /* Free remaining resources this DLL uses. Some thread
               DLLs may not have been called. */
            InterlockedIncrement (&nPD);
            /* Count this as detaching the primary thread as well */
            InterlockedIncrement (&nTD);
            pBuf = TlsGetValue (tlsIndex);
            if (pBuf != NULL) {
                free (pBuf);
                pBuf = NULL;
            }
            TlsFree(tlsIndex);
            return TRUE;

        case DLL_THREAD_DETACH:
            /* May not be called for every thread using the DLL */
            InterlockedIncrement (&nTD);
            pBuf = TlsGetValue (tlsIndex);
            if (pBuf != NULL) {
                free (pBuf);
                pBuf = NULL;
            }
            return TRUE;

        default: return TRUE;
    }
}

__declspec(dllexport)
BOOL ReceiveCSMessage (MESSAGE *pMsg, SOCKET sd)
{
    /* FALSE return indicates an error or disconnect */
```

```
BOOL disconnect = FALSE;
LONG32 nRemainRecv, nXfer, k; /* Must be signed integers */
LPBYTE pBuffer, message;
CHAR tempBuff[MAX_MESSAGE_LEN+1];
STATIC_BUF *pBuff;

if (pMsg == NULL) return FALSE;
pBuff = (STATIC_BUF *) TlsGetValue (tlsIndex);
if (pBuff == NULL) { /* First time initialization. */
   /* Only threads that need this storage will allocate it */
   pBuff = malloc (sizeof (STATIC_BUF));
   if (pBuff == NULL) return FALSE; /* Error */
   TlsSetValue (tlsIndex, pBuff);
   pBuff->staticBufLen = 0; /* Intialize state */
}

message = pMsg->record;
/* Read up to the null character, leaving residual data
 * in the static buffer */

for (k = 0; k < pBuff->staticBufLen && pBuff->staticBuf[k] != '\0';
         k++) {
   message[k] = pBuff->staticBuf[k];
}  /* k is the number of characters transferred */
if (k < pBuff->staticBufLen) { /* a null was found in staticBuf */
   message[k] = '\0';
   pBuff->staticBufLen -= (k+1); /* Adjust the buffer state */
   memcpy (pBuff->staticBuf, &(pBuff->staticBuf[k+1]),
         pBuff->staticBufLen);
   return TRUE; /* No socket input required */
}

/* the entire static buffer was transferred. No null found */
nRemainRecv = sizeof(tempBuff) -
         sizeof(CHAR) - pBuff->staticBufLen;
pBuffer = message + pBuff->staticBufLen;
pBuff->staticBufLen = 0;

while (nRemainRecv > 0 && !disconnect) {
   nXfer = recv (sd, tempBuff, nRemainRecv, 0);
   if (nXfer <= 0) {
      disconnect = TRUE;
      continue;
   }

   /* Transfer to target message up to null, if any */
   for (k = 0; k < nXfer && tempBuff[k] != '\0'; k++) {
      *pBuffer = tempBuff[k];
      nRemainRecv -= nXfer; pBuffer++;
   }
```

```
        if (k < nXfer) { /* null has been found */
            *pBuffer = '\0';
            nRemainRecv = 0;
            /* Adjust the static buffer state for the next
             * ReceiveCSMessage call */
            memcpy (pBuff->staticBuf, &tempBuff[k+1], nXfer - k - 1);
            pBuff->staticBufLen = nXfer -k - 1;
        }
    }
    return !disconnect;
}

__declspec(dllexport)
BOOL SendCSMessage (MESSAGE *pMsg, SOCKET sd)
{
    /* Send the the request to the server on socket sd */
    BOOL disconnect = FALSE;
    LONG32 nRemainSend, nXfer;
    LPBYTE pBuffer;

    if (pMsg == NULL) return FALSE;
    pBuffer = pMsg->record;
    if (pBuffer == NULL) return FALSE;
    nRemainSend = min(strlen (pBuffer) + 1, MAX_MESSAGE_LEN);

    while (nRemainSend > 0 && !disconnect)  {
        /* send does not guarantee that the entire message is sent */
        nXfer = send (sd, pBuffer, nRemainSend, 0);
        if (nXfer <= 0) {
            disconnect = TRUE;
        }
        nRemainSend -=nXfer; pBuffer += nXfer;
    }
    return !disconnect;
}
```

Comments on the DLL and Thread Safety

- `DllMain`, with `DLL_THREAD_ATTACH`, is called whenever a new thread is created, but there is not a distinct `DLL_THREAD_ATTACH` call for the primary thread or any other threads that exist when the DLL is loaded. The DLL's `DLL_PROCESS_ATTACH` case must handle these cases.
- In general, and even in this case (consider the `accept` thread), some threads may not require the allocated memory, but `DllMain` cannot distinguish the different thread types. Therefore, the `DLL_THREAD_ATTACH` case does not

actually allocate any memory, and there is no need to call `TlsSetValue` because Windows initializes the value to `NULL`. The `ReceiveCSMessage` entry point allocates the memory the first time it is called. In this way, the thread-specific memory is allocated only by threads that require it, and different thread types can allocate exactly the resources they require.

- While this DLL is thread-safe, a given thread can use these routines with only one socket at a time because the persistent state is associated with the thread, not the socket. The next example addresses this issue.
- The DLL source code on the *Examples* file is instrumented to print the total number of `DllMain` calls by type.
- There is still a resource leak risk, even with this solution. Some threads, such as the accept thread, may never terminate and therefore will never be detached from the DLL. `ExitProcess` will call `DllMain` with `DLL_PROCESS_DETACH` but not with `DLL_THREAD_DETACH` for threads that are still active. This does not cause a problem in this case because the accept thread does not allocate any resources, and even memory is freed when the process terminates. There would, however, be an issue if threads allocated resources such as temporary files; the ultimate solution would be to create a globally accessible list of resources. The `DLL_PROCESS_DETACH` code would then have the task of scanning the list and deallocating the resources; this is left as an exercise.

Example: An Alternative Thread-Safe DLL Strategy

Program 12–5, while typical of the way in which TLS and `DllMain` are combined to create thread-safe libraries, has two major weaknesses noted in the comments in the previous section. First, the state is associated with the thread rather than with the socket, so a given thread can process only one socket at a time. Second, there is the resource leak risk mentioned in the last bullet above.

An effective alternative approach to thread-safe library functions is to create a handle-like structure that is passed to every function call. The state is then maintained in the structure. The application explicitly manages the state, so you can manage multiple sockets in a thread, and you can even use the sockets with fibers (there might be one socket, or more, per fiber). Many UNIX and Linux applications use this technique to create thread-safe C libraries; the main disadvantage is that the functions require an additional parameter for the state structure.

Program 12–6 modifies Program 12–5. Notice that `DllMain` is not necessary, but there are two new functions to initialize and free the state structure. The send and receive functions require only minimal changes. An associated server, `serverSKHA`, is included in the *Examples* file and requires only slight changes in order to create and close the socket handle (`HA` denotes "handle").

Program 12–6 SendReceiveSKHA: Thread-Safe DLL with a State Structure

```
/* SendReceiveSKHA.c -- multithreaded streaming socket. */
/* This is a modification of SendReceiveSKST.c to illustrate a */
/* different thread-safe library technique. */
/* State is preserved in a handle-like state structure rather than */
/* using TLS. This allows a thread to use several sockets at once. */
/* Messages are delimited by null characters ('\0'). */

#include "Everything.h"
#include "ClientServer.h"/* Defines MESSAGEa. */

typedef struct SOCKET_HANDLE_T {
/* Current socket state */
/* Contains "staticBuffLen" characters of residual data */
/* There may or may not be end-of-string (null) characters */
   SOCKET sk;
   char staticBuff[MAX_RQRS_LEN];
   LONG32 staticBuffLen;
} SOCKET_HANDLE, * PSOCKET_HANDLE;

/* Functions to create and close "streaming socket handles" */
__declspec (dllexport)
PVOID CreateCSSocketHandle (SOCKET s)
{
   PVOID p;
   PSOCKET_HANDLE ps;

   p = malloc (sizeof(SOCKET_HANDLE));
   if (p == NULL) return NULL;
   ps = (PSOCKET_HANDLE)p;
   ps->sk = s;
   ps->staticBuffLen = 0; /* Initialize buffer state */
   return p;
}

__declspec (dllexport)
BOOL CloseCSSocketHandle (PSOCKET_HANDLE psh)
{
   if (psh == NULL) return FALSE;
   free (psh);
   return TRUE;
}

__declspec(dllexport)
BOOL ReceiveCSMessage (MESSAGE *pMsg, PSOCKET_HANDLE psh)
/*  Use PVOID so that calling program does not need to include the */
/* SOCKET_HANDLE definition. */
```

```
{
    /* TRUE return indicates an error or disconnect */
    BOOL disconnect = FALSE;
    LONG32 nRemainRecv = 0, nXfer, k; /* Must be signed integers */
    LPSTR pBuffer, message;
    CHAR tempBuff[MAX_RQRS_LEN+1];
    SOCKET sd;

    if (psh == NULL) return FALSE;
    sd = psh->sk;

    /* This is all that's changed from SendReceiveSKST! */
    message = pMsg->record;
    /* Read up to the null character, leaving residual data
     * in the static buffer */

    for (k = 0;
           k < psh->staticBuffLen && psh->staticBuff[k] != '\0'; k++) {
        message[k] = psh->staticBuff[k];
    }  /* k is the number of characters transferred */
    if (k < psh->staticBuffLen) { /* null found in buffer */
        message[k] = '\0';
        psh->staticBuffLen -= (k+1); /* Adjust buffer state */
        memcpy (psh->staticBuff, &(psh->staticBuff[k+1]),
               psh->staticBuffLen);
        return TRUE; /* No socket input required */
    }

    /* The entire static buffer was transferred. No null found */
    nRemainRecv = sizeof(tempBuff) - 1 - psh->staticBuffLen;
    pBuffer = message + psh->staticBuffLen;
    psh->staticBuffLen = 0;

    while (nRemainRecv > 0 && !disconnect) {
        nXfer = recv (sd, tempBuff, nRemainRecv, 0);
        if (nXfer <= 0) {
            disconnect = TRUE;
            continue;
        }

        nRemainRecv -=nXfer;
        /* Transfer to target message up to null, if any */
        for (k = 0; k < nXfer && tempBuff[k] != '\0'; k++) {
            *pBuffer = tempBuff[k];
            pBuffer++;
        }
        if (k >= nXfer) { /* null not found, read more */
            nRemainRecv -= nXfer;
        } else { /* null has been found */
            *pBuffer = '\0';
```