```
BOOL GetExitCodeThread (
   HANDLE hThread,
   LPDWORD lpExitCode)
```

`lpExitCode` will contain the thread's exit code. If the thread is still running, the value is `STILL_ACTIVE`.

Thread Identity

You can obtain thread IDs and handles using functions that are similar to those used with processes.

- `GetCurrentThread` returns a noninheritable pseudohandle to the calling thread.
- `GetCurrentThreadId` obtains the thread ID rather than the handle.
- `GetThreadId` obtains a thread's ID from its handle.
- `OpenThread` creates a thread handle from a thread ID. `OpenProcess` was very useful in `JobShell` (Program 6–3), and you can use `OpenThread` in a similar fashion.

Additional Thread Management Functions

While the thread management functions just discussed are sufficient in most cases, including the examples in this book, two additional functions were introduced in XP and Windows 2003 (that is, NT5). Brief descriptions follow.

1. `GetProcessIdOfThread`, which requires Windows 2003 or later (XP does not support it), finds the process ID of a thread from the thread's handle. You could use this function in a program that manages or interacts with threads in other processes. If necessary, use `OpenProcess` to obtain a process handle.
2. `GetThreadIOPendingFlag` determines whether the thread, identified by its handle, has any outstanding I/O requests. For example, the thread might be blocked on a `ReadFile` operation. The result is the status at the time that the function is executed; the actual status could change at any time if the target thread completes or initiates an operation.

Suspending and Resuming Threads

Every thread has a *suspend count*, and a thread can execute only if this count is `0`. One thread can increment or decrement the suspend count of another thread using `SuspendThread` and `ResumeThread`. Recall that a thread can be created in the suspended state with a count of `1`.

```
DWORD ResumeThread (HANDLE hThread)

DWORD SuspendThread (HANDLE hThread)
```

Both functions, if successful, return the previous suspend count. `0xFFFFFFFF` indicates failure.

Waiting for Threads to Terminate

One thread can wait for another thread to terminate in the same way that threads wait for process termination (Chapter 6). Use a wait function (`WaitForSingle-Object` or `WaitForMultipleObjects)` using thread handles instead of process handles. Note that the handles in the array passed to `WaitForMultipleObjects` do not all need to be of the same type; for example, thread, process, and other handle types can be mixed in a single call.

`WaitForMultipleObjects` can wait for only `MAXIMUM_WAIT_OBJECTS` (64) handles at one time, but you can perform a series of waits if you have a large number of threads. Program 6–1 already illustrated this technique; most programs in this book will perform single waits.

The wait function waits for the object, indicated by the handle, to become *signaled*. In the case of threads, `ExitThread` and `TerminateThread` set the object to the signaled state, releasing all other threads waiting on the object, including threads that might wait in the future after the thread terminates. Once a thread handle is signaled, it never becomes nonsignaled. The same is true of process handles but not of handles to some other objects, such as mutexes and events (see the next chapter).

Note that multiple threads can wait on the same object. Similarly, the `ExitProcess` function sets the process state and the states of all its threads to signaled.

Threads are a well-established concept in many OSs, and historically, many UNIX vendors and users have provided their own proprietary implementations. Some thread libraries have been implemented outside the kernel. POSIX Pthreads are now the standard. Pthreads are part of all commercial UNIX and Linux implementations. The system calls are distinguished from normal UNIX system calls by the `pthread_` prefix name. Pthreads are also supported on some proprietary non-UNIX systems.

`pthread_create` is the equivalent of `CreateThread`, and `pthread_exit` is the equivalent of `ExitThread`. One thread waits for another to exit with `pthread-_join`. Pthreads provide the very useful `pthread_cancel` function, which, unlike `TerminateThread`, ensures that completion handlers and cancellation handlers are executed. Thread cancellation would be a welcome addition to Windows, but Chapter 10 will show a method to achieve the same effect.

Using the C Library in Threads

Most code requires the C library, even if it is just to manipulate strings. Historically, the C library was written to operate in single-threaded processes, so some functions use global storage to store intermediate results. Such libraries are not *thread-safe* because two separate threads might, for example, be simultaneously accessing the library and modifying the library's global storage. Proper design of threaded code is discussed again in Chapter 8, which describes Windows synchronization.

The `strtok` function is an example of a C library function that is not thread-safe. `strtok`, which scans a string to find the next occurrence of a token, maintains *persistent state* between successive function calls, and this state is in static storage, shared by all the threads calling the function.

Microsoft C solves such problems by supplying a thread-safe C library implementation named `LIBCMT.LIB`. There is more. Do not use `CreateThread`; if you do, there is a risk of different threads accessing and modifying the same data that the library requires for correct operation. Instead, use a special C function, `_beginthreadex`, to start a thread and create thread-specific working storage for `LIBCMT.LIB`. Use `_endthreadex` in place of `ExitThread` to terminate a thread.

Note: There is a `_beginthread` function, intended to be simpler to use, *but you should never use it*. First, `_beginthread` does not have security attributes or flags and does not return a thread ID. More importantly, it actually closes the thread handle it creates, and the returned thread handle may be invalid by the time the parent thread stores it. Also avoid `_endthread`; it does not allow for a return value.

The `_beginthreadex` arguments are exactly the same as for the Windows functions but without the Windows type definitions; therefore, be sure to cast the `_beginthreadex` return value to a `HANDLE` to avoid warning messages. Be certain to define `_MT` before any include files; this definition is in `Environment.h` for the sample programs. That is all there is to it. When you're using the Visual Studio development environment, be sure to do the following:

- Open the `Project Properties` pages.
- Under `Configuration Properties`, expand `C/C++`.
- Select `Code Generation`.
- For the `Runtime Library`, specify `Multi-threaded DLL (/MD)`.
- Terminate threads with `_endthreadex` or simply use a `return` statement at the end of the thread routine.

All examples will operate this way, and the programs will never use `CreateThread` directly, even if the thread functions do not use the C library.

Thread-Safe Libraries

User-developed libraries must be carefully designed to avoid thread safety issues, especially when persistent state is involved. A Chapter 12 example (Program 12–5), where a DLL maintains state in a parameter, shows one strategy.

Another Chapter 12 example (Program 12–6) demonstrates an alternative approach that exploits the `DllMain` function and TLS, which is described later in this chapter.

Example: Multithreaded Pattern Searching

Program 6–1, `grepMP`, used processes to search multiple files simultaneously. Program 7–1, `grepMT`, includes the `grep` pattern searching source code so that threads can perform the searching within a single process. The pattern searching code relies on the C library for file I/O. The main control program is similar to the process implementation.

This example also shows that a form of asynchronous I/O is possible with threads without using the explicit methods of Chapter 14. In this example, the program is managing concurrent I/O to multiple files, and the main thread, or any other thread, can perform additional processing before waiting for I/O completion. In the author's opinion, threads are a much simpler method of achieving asynchronous I/O, and Chapter 14 compares the methods, allowing readers to form their own opinions. We will see, however, that asynchronous I/O, combined with I/O completion ports, is useful and often necessary when the number of threads is large. Furthermore, as of NT6, extended asynchronous I/O often performs very well.

`grepMT`, for the purposes of illustration, differs in another way from `grepMP`. Here, `WaitForMultipleObjects` waits for a *single* thread to terminate rather than waiting for all the threads. The appropriate output is displayed before waiting for another thread to complete. The completion order will, in most cases, vary from one run to the next. It is easy to modify the program to display the results in the order of the command line arguments; just imitate grepMP.

Finally, notice that there is a limit of 64 threads due to the value of MAXIMUM_WAIT_OBJECTS, which limits the number of handles in the WaitForMultipleObjects call. If more threads are required, create the appropriate logic to loop on either WaitForSingleObject or WaitForMultipleObjects.

Caution: grepMT performs asynchronous I/O in the sense that separate threads are concurrently, and synchronously, reading different files with read operations that block until the read is complete. You can also concurrently read from the same file if you have distinct handles on the file (typically, one per thread). These handles should be generated by CreateFile rather than DuplicateHandle. Chapter 14 describes asynchronous I/O, with and without user threads, and an example in the *Examples* file (cciMT; see Chapter 14) has several threads performing I/O to the same file.

Note: You can perform all sorts of parallel file processing using this design. All that is required is to change the "ThGrep" function at the end of Program 7–1. An exercise suggests that you implement a parallel word count (wc) program this way, but you could also edit files or compile source code files in parallel.

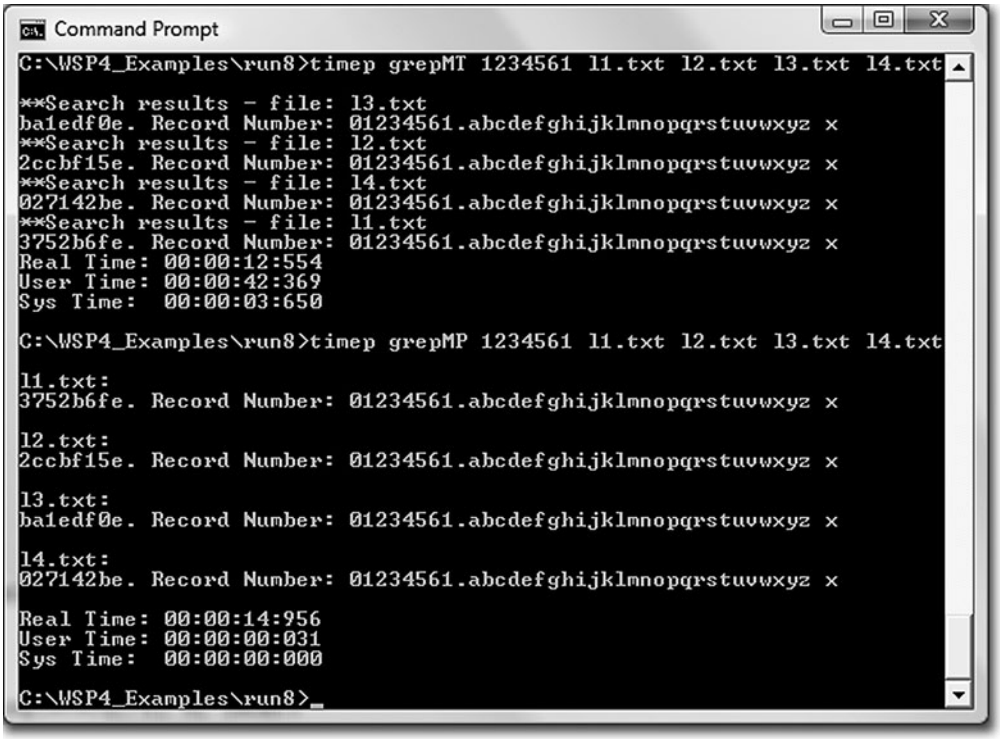
Program 7–1 grepMT: Multithreaded Pattern Searching

```
/* Chapter 7. grepMT. */
/* Parallel grep -- multiple thread version. */

#include "Everything.h"
typedef struct { /* grep thread's data structure. */
   int argc;
   TCHAR targv[4][MAX_PATH];
} GREP_THREAD_ARG;
typedef GREP_THREAD_ARG *PGR_ARGS;
static DWORD WINAPI ThGrep (PGR_ARGS pArgs);

int _tmain (int argc, LPTSTR argv[])
{
   GREP_THREAD_ARG * gArg;
   HANDLE * tHandle;
   DWORD threadIndex, exitCode;
   TCHAR commandLine[MAX_COMMAND_LINE];
   int iThrd, threadCount;
   STARTUPINFO startUp;
   PROCESS_INFORMATION processInfo;

   GetStartupInfo (&startUp);
   /* Boss thread: create separate "grep" thread for each file. */
   tHandle = malloc ((argc - 2) * sizeof (HANDLE));
   gArg = malloc ((argc - 2) * sizeof (GREP_THREAD_ARG));
```

```
    for (iThrd = 0; iThrd < argc - 2; iThrd++) {
       _tcscpy (gArg[iThrd].targv[1], argv[1]); /* Pattern. */
       _tcscpy (gArg[iThrd].targv[2], argv[iThrd + 2]);
       GetTempFileName /* Temp file name. */
             (".", "Gre", 0, gArg[iThrd].targv[3]);
       gArg[iThrd].argc = 4;

       /* Create a worker thread to execute the command line. */
       tHandle[iThrd] = (HANDLE)_beginthreadex (
             NULL, 0, ThGrep, &gArg[iThrd], 0, NULL);
    }
    /* Redirect std output for file listing process. */
    startUp.dwFlags = STARTF_USESTDHANDLES;
    startUp.hStdOutput = GetStdHandle (STD_OUTPUT_HANDLE);

    /* Worker threads are all running. Wait for them to complete. */
    threadCount = argc - 2;
    while (threadCount > 0) {
       threadIndex = WaitForMultipleObjects (
             threadCount, tHandle, FALSE, INFINITE);
       iThrd = (int) threadIndex - (int) WAIT_OBJECT_0;
       GetExitCodeThread (tHandle[iThrd], &exitCode);
       CloseHandle (tHandle[iThrd]);
       if (exitCode == 0) { /* Pattern found. */
          if (argc > 3) {
          /* Print file name if more than one. */
             _tprintf (_T ("\n**Search results - file: %s\n"),
                   gArg[iThrd].targv[2]);
             fflush (stdout);
          }
          /* Use the "cat" program to list the result files. */
          _stprintf (commandLine, _T ("%s%s"), _T ("cat "),
                gArg[iThrd].targv[3]);
          CreateProcess (NULL, commandLine, NULL, NULL,
                TRUE, 0, NULL, NULL, &startUp, &processInfo);
          WaitForSingleObject (processInfo.hProcess, INFINITE);
          CloseHandle (processInfo.hProcess);
          CloseHandle (processInfo.hThread);
       }
       DeleteFile (gArg[iThrd].targv[3]);

       /* Adjust thread and file name arrays. */
       tHandle[iThrd] = tHandle[threadCount - 1];
       _tcscpy (gArg[iThrd].targv[3], gArg[threadCount - 1].targv[3]);
       _tcscpy (gArg[iThrd].targv[2], gArg[threadCount - 1].targv[2]);
       threadCount--;
    }
}
```

```
/* The form of the grep thread function code is:
static DWORD WINAPI ThGrep (PGR_ARGS pArgs)
{
   . . .
} */
```

Run 7–1 shows grepMT operation and compares the performance with grepMP, using the same four 640MB files.



```
C:\WSP4_Examples\run8>timep grepMT 1234561 l1.txt l2.txt l3.txt l4.txt

**Search results - file: l3.txt
ba1edf0e. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x
**Search results - file: l2.txt
2ccbf15e. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x
**Search results - file: l4.txt
027142be. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x
**Search results - file: l1.txt
3752b6fe. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x
Real Time: 00:00:12:554
User Time: 00:00:42:369
Sys Time:  00:00:03:650

C:\WSP4_Examples\run8>timep grepMP 1234561 l1.txt l2.txt l3.txt l4.txt

l1.txt:
3752b6fe. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x

l2.txt:
2ccbf15e. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x

l3.txt:
ba1edf0e. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x

l4.txt:
027142be. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x

Real Time: 00:00:14:956
User Time: 00:00:00:031
Sys Time:  00:00:00:000

C:\WSP4_Examples\run8>_
```

Run 7–1 `grepMT`: Multithreaded Pattern Searching

Performance Impact

`grepMP` and `grepMT` are comparable in terms of program structure and complexity, but `grepMT` has the expected advantage of better performance; it is more efficient for the kernel to switch between threads than between processes. Run 7–1 shows that the theoretical advantage is real, but not large (12.554 versus 14.956 seconds). Specifically, if you are processing multiple large files of about the same size, with one thread per file, performance improves nearly linearly with the number of files up to the number of processors on the computer. You may not see this improvement with smaller files, however, because of the thread creation and management overhead. Chapter 9 shows how to improve performance slightly more with NT6 thread pools.

Both implementations exploit multiprocessor systems, giving a considerable improvement in the elapsed time; threads, whether in the same process or in different processes, run in parallel on the different processors. The measured user time actually exceeds the elapsed time because the user time is the total for all the processors.

The *Examples* file contains a word counting example, `wcMT`, which has the same structure as `grepMT` and, on a multiprocessor computer, is faster than the Cygwin `wc` command (Cygwin, an open source set of UNIX/Linux commands, implements `wc`).

There is a common misconception, however, that this sort of parallelism using either `grepMP` or `grepMT` yields performance improvements only on multiprocessor systems. You can also gain performance when there are multiple disk drives or some other parallelism in the storage system. In such cases, multiple I/O operations to different files can run concurrently.

The Boss/Worker and Other Threading Models

`grepMT` illustrates the "boss/worker" threading model, and Figure 6–3 illustrates the relationship if "thread" is substituted for "process." The boss thread (the main thread in this case) assigns tasks for the worker threads to perform. Each worker thread is given a file to search, and the worker threads pass their results to the boss thread in a temporary file.

There are numerous variations, such as the *work crew model* in which the workers cooperate on a single task, each performing a small piece. The next example uses a work crew (see Figure 7–2). The workers might even divide up the work themselves without direction from the boss. Multithreaded programs can employ nearly every management arrangement that humans use to manage concurrent tasks.

The two other major models are the *client/server model* (illustrated in Figure 7–1 and developed in Chapter 11) and the *pipeline model*, where work moves from one thread to the next (see Chapter 10 and Figure 10–1 for an example of a multistage pipeline).

There are many advantages to using these models when designing a multithreaded program, including the following.

- Most multithreaded programming problems can be solved using one of the standard models, expediting design, development, and debugging.

- Not only does using a well-understood and tested model avoid many of the mistakes that are so easy to make in a multithreaded program, but the model also helps you obtain the best performance.
- The models correspond naturally to the structures of most programming problems.
- Programmers who maintain the program will be able to understand it much more easily if documentation describes the program in terms that everyone understands.
- Troubleshooting an unfamiliar program is much easier if you analyze it in terms of models. Frequently, an underlying problem is found when the program is seen to violate the basic principles of one of the models.
- Many common defects, such as race conditions and deadlocks, are also described by simple models, as are effective methods of using the synchronization objects described in Chapters 9 and 10.

These classical thread models are used in many OSs. The Component Object Model (COM), widely used in Windows systems, uses different terminology.

Example: Merge-Sort—Exploiting Multiple Processors

This example, diagrammed in Figure 7–2, shows how to use threads to get significant performance gains, especially on a multiprocessor computer. The basic idea is to divide the problem into component tasks, give each task to a separate thread, and then combine the results to get the complete solution. The Windows executive will automatically assign the threads to separate processors, so the threads will execute in parallel, reducing elapsed time.

This strategy, often called the *divide and conquer strategy* or the *work crew model,* is useful both for performance and as an algorithm design method. `grepMT`, Program 7–1, could be considered one example; it creates a thread for each file or pattern matching task.

Next, consider another example in which a single task, sorting a file, is divided into subtasks delegated to separate threads.

Merge-sort, in which the array to be sorted is divided into smaller arrays, is a classic divide and conquer algorithm. Each small array is sorted individually, and the individual sorted arrays are merged in pairs to yield larger sorted arrays. The pairwise merging continues until completion. Generally, merge-sort starts with small arrays, which can be sorted efficiently with a simple algorithm. This example starts with larger arrays so that there can be one array for each processor. Figure 7–2 is a sketch of the algorithm.

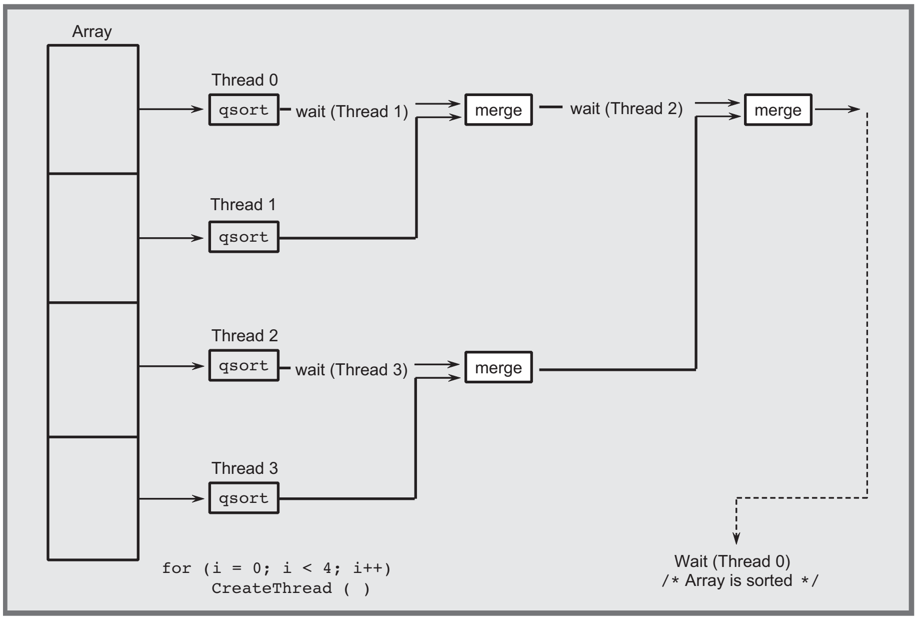


Figure 7–2 Merge-Sort with Multiple Threads

Program 7–2 shows the implementation details. The user specifies the number of tasks on the command line. Exercise 7–9 suggests that `sortMT` use `GetSystemInfo` to find the number of processors and then create one thread per processor.

Notice that the program runs efficiently on single-processor systems with sufficient memory and gains a significant performance improvement on multiprocessor systems. *Caution:* The algorithm as shown will work only if the number of records in the sort file is divisible by the number of threads and if the number of threads is a power of 2. Exercise 7–8 removes these limitations.

Note: In understanding this program, concentrate on the thread management logic separately from the logic that determines which portion of the array a thread should sort. Notice too that the C library `qsort` function is used, so there is no need to be concerned with developing an efficient basic sort function.

Additional Point to Notice: The thread creation loop (look for the comment "Create the sorting threads" on the second page of the listing) creates the worker threads in the suspended state. The threads are resumed only after all the worker threads are created. The reason for this can be seen from Figure 7–2; consider what would happen if Thread 0 waits for Thread 1 before Thread 1 is created. There would then be no handle to wait for. This is an example of a "race" condition where two or more threads make unsafe assumptions about the progress of the other threads. Exercises 7–10 and 7–13 investigate this further.

Program 7–2 `sortMT`: Merge-Sort with Multiple Threads

```
/* Chapter 7. SortMT.
   File sorting with multiple threads (a work crew).
   sortMT [options] nt file */

#include "Everything.h"
#define DATALEN 56 /* Key: 8 bytes; Data: 56 bytes. */
#define KEYLEN 8
typedef struct _RECORD {
      CHAR key[KEYLEN];
      TCHAR data[DATALEN];
} RECORD;
#define RECSIZE sizeof (RECORD)
typedef RECORD * LPRECORD;

typedef struct _THREADARG {          /* Thread argument */
   DWORD iTh;                        /* Thread number: 0, 1, 2, ... */
   LPRECORD lowRecord;               /* Low record */
   LPRECORD highRecord;              /* High record */
} THREADARG, *PTHREADARG;

static int KeyCompare (LPCTSTR, LPCTSTR);
static DWORD WINAPI SortThread (PTHREADARG pThArg);
static DWORD nRec; /* Total number of records to be sorted. */
static HANDLE *pThreadHandle;

int _tmain (int argc, LPTSTR argv[])
{
   HANDLE hFile, mHandle;
   LPRECORD pRecords = NULL;
   DWORD lowRecordNum, nRecTh, numFiles, iTh;
   LARGE_INTEGER fileSize;
   BOOL noPrint;
   int iFF, iNP;
   PTHREADARG threadArg;
   LPTSTR stringEnd;

   iNP = Options (argc, argv, _T ("n"), &noPrint, NULL);
   iFF = iNP + 1;
   numFiles = _ttoi (argv[iNP]);

   /* Open the file and map it */
   hFile = CreateFile (argv[iFF], GENERIC_READ | GENERIC_WRITE,
         0, NULL, OPEN_EXISTING, 0, NULL);
   /* For technical reasons, we need to add bytes to the end. */
   SetFilePointer(hFile, 2, 0, FILE_END);
   SetEndOfFile(hFile);
```

```
    mHandle = CreateFileMapping(hFile, NULL, PAGE_READWRITE,
                                0, 0, NULL);

    /* Get the file size. */
    GetFileSizeEx (hFile, &fileSize);
    nRec = fileSize.QuadPart / RECSIZE;/* Total number of records. */
    nRecTh = nRec / numFiles;/* Records per thread. */
    threadArg = malloc (numFiles*sizeof (THREADARG)); /* thread args */
    pThreadHandle = malloc (numFiles * sizeof (HANDLE));

    /* Map the entire file */
    pRecords = MapViewOfFile(mHandle, FILE_MAP_ALL_ACCESS, 0, 0, 0);
    CloseHandle (mHandle);

    /* Create the sorting threads. */
    lowRecordNum = 0;
    for (iTh = 0; iTh < numFiles; iTh++) {
       threadArg[iTh].iTh = iTh;
       threadArg[iTh].lowRecord = pRecords + lowRecordNum;
       threadArg[iTh].highRecord = pRecords + (lowRecordNum + nRecTh);
       lowRecordNum += nRecTh;
       pThreadHandle[iTh] = (HANDLE)_beginthreadex (NULL,
          0, SortThread, &threadArg[iTh], CREATE_SUSPENDED, NULL);
    }

    /* Resume all the initially suspened threads. */
    for (iTh = 0; iTh < numFiles; iTh++)
       ResumeThread (pThreadHandle[iTh]);

    /* Wait for the sort-merge threads to complete. */
    WaitForSingleObject (pThreadHandle[0], INFINITE);
    for (iTh = 0; iTh < numFiles; iTh++)
       CloseHandle (pThreadHandle[iTh]);

    /*  Print out the entire sorted file as one single string. */
    stringEnd = (LPTSTR) pRecords + nRec*RECSIZE;
    *stringEnd =_T('\0');
    if (!noPrint) {
        _tprintf (_T("%s"), (LPCTSTR) pRecords);
    }
    UnmapViewOfFile(pRecords);
    /* Restore the file length */
    SetFilePointer(hFile, -2, 0, FILE_END);
    SetEndOfFile(hFile);

    CloseHandle(hFile);
    free (threadArg); free (pThreadHandle);
    return 0;
} /* End of _tmain. */
```