some deviations from the MSDN documentation, which are described in Exercise 6–12 for investigation.

**Program 6–7** `JobObjectShell:` Monitoring Processes with a Job Object

```
/* Chapter 6 */
/* JobObjectShell.c JobShell extension
   Enhances JobShell with a time limit on each process.
   The process time limit (seconds) is argv[1] (if present)
      0 or omitted means no process time limit
*/

#include "Everything.h"
#include "JobManagement.h"

#define MILLION 1000000
HANDLE hJobObject = NULL;

JOBOBJECT_BASIC_LIMIT_INFORMATION basicLimits =
         {0, 0, JOB_OBJECT_LIMIT_PROCESS_TIME};

int _tmain (int argc, LPTSTR argv[])
{
   LARGE_INTEGER processTimeLimit;
   . . .
   hJobObject = NULL;
   processTimeLimit.QuadPart = 0;
   if (argc >= 2) processTimeLimit.QuadPart = atoi(argv[1]);
   basicLimits.PerProcessUserTimeLimit.QuadPart =
      processTimeLimit.QuadPart * 10 * MILLION;

   hJobObject = CreateJobObject(NULL, NULL);
   SetInformationJobObject(hJobObject,
      JobObjectBasicLimitInformation, &basicLimits,
      sizeof(JOBOBJECT_BASIC_LIMIT_INFORMATION));
   . . .
   /* Process commands. Call Jobbg, Jobs, etc. - listed below */
   CloseHandle (hJobObject);

   return 0;
}

/* Jobbg: Execute a command line in the background, put
   the job identity in the user's job file, and exit.
*/
int Jobbg (int argc, LPTSTR argv[], LPTSTR command)
{
   /* Execute the command line (targv) and store the job id,
      the process id, and the handle in the jobs file. */
```

```
    DWORD fCreate;
    LONG jobNumber;
    BOOL flags[2];

    STARTUPINFO startUp;
    PROCESS_INFORMATION processInfo;
    LPTSTR targv = SkipArg (command);

    GetStartupInfo (&startUp);

       /* Determine the options. */
    Options (argc, argv, _T ("cd"), &flags[0], &flags[1], NULL);

       /* Skip over the option field as well, if it exists. */
    if (argv[1][0] == '-')
       targv = SkipArg (targv);

    fCreate = flags[0] ? CREATE_NEW_CONSOLE : flags[1] ?
          DETACHED_PROCESS : 0;

    /* Create the job/thread suspended.
       Resume it once the job is entered properly. */
    CreateProcess (NULL, targv, NULL, NULL, TRUE,
          fCreate | CREATE_SUSPENDED | CREATE_NEW_PROCESS_GROUP,
          NULL, NULL, &startUp, &processInfo);

    AssignProcessToJobObject(hJobObject, processInfo.hProcess);

    jobNumber = GetJobNumber (&processInfo, targv);
    if (jobNumber >= 0)
       ResumeThread (processInfo.hThread);
    else {
       TerminateProcess (processInfo.hProcess, 3);
       CloseHandle (processInfo.hThread);
       CloseHandle (processInfo.hProcess);
       return 5;
    }

    CloseHandle (processInfo.hThread);
    CloseHandle (processInfo.hProcess);
    _tprintf (_T (" [%d] %d\n"), jobNumber, processInfo.dwProcessId);
    return 0;
}

/* Jobs: List all running or stopped jobs that have
   been created by this user under job management;
   that is, have been started with the jobbg command.
   List summary process count and user time information.
*/
```

```
int Jobs (int argc, LPTSTR argv[], LPTSTR command)
{
   JOBOBJECT_BASIC_ACCOUNTING_INFORMATION BasicInfo;

   DisplayJobs (); /* Not job objects, but jobbg created processes */

   /* Dispaly the job object information */
   QueryInformationJobObject(hJobObject,
      JobObjectBasicAccountingInformation, &BasicInfo,
      sizeof(JOBOBJECT_BASIC_ACCOUNTING_INFORMATION), NULL);
   _tprintf (_T("Total Processes: %d, Active: %d, Terminated: %d.\n"),
      BasicInfo.TotalProcesses, BasicInfo.ActiveProcesses,
      BasicInfo.TotalTerminatedProcesses);
   _tprintf (_T("User time all processes: %d.%03d\n"),
      BasicInfo.TotalUserTime.QuadPart / MILLION,
      (BasicInfo.TotalUserTime.QuadPart % MILLION) / 10000);

   return 0;
}
```

## Summary

Windows provides a straightforward mechanism for managing processes and synchronizing their execution. Examples have shown how to manage the parallel execution of multiple processes and how to obtain information about execution times. Windows does not maintain a parent-child relationship among processes, so the programmer must manage this information if it is required, although job objects provide a convenient way to group processes.

### Looking Ahead

Threads, which are independent units of execution within a process, are described in the next chapter. Thread management is similar in some ways to process management, and there are exit codes, termination, and waiting on thread handles. To illustrate this similarity, `grepMP` (Program 6–1) is reimplemented with threads in Chapter 7's first example program.

Chapter 8 then introduces synchronization, which coordinates operation between threads in the same or different processes.

## Exercises

6–1. Extend Program 6–1 (`grepMP`) so that it accepts command line options and not just the pattern.

6–2. Rather than pass the temporary file name to the child process in Program 6–1, convert the inheritable file handle to a `DWORD` (a `HANDLE` requires 4 bytes in Win32; investigate the Win64 `HANDLE` size) and then to a character string. Pass this string to the child process on the command line. The child process, in turn, must convert the character string back to a handle value to use for output. The `catHA.c` and `grepHA.c` programs in the *Examples* file illustrate this technique. Is this technique advisable, or is it poor practice, in your opinion?

6–3. Program 6–1 waits for all processes to complete before listing the results. It is impossible to determine the order in which the processes actually complete within the current program. Modify the program so that it can also determine the termination order. *Hint:* Modify the call to `WaitForMultipleObjects` so that it returns after each individual process terminates. An alternative would be to sort by the process termination times.

6–4. The temporary files in Program 6–1 must be deleted explicitly. Can you use `FILE_FLAG_DELETE_ON_CLOSE` when creating the temporary files so that deletion is not required?

6–5. Determine any `grepMP` performance advantages (compared with sequential execution) on different multiprocessor systems or when the files are on separate or network drives. Appendix C presents some partial results, as does Run 6–1.

6–6. Can you find a way to collect the user and kernel time required by `grepMP`? It may be necessary to modify `grepMP` to use job objects.

6–7. Enhance the `DisplayJobs` function (Program 6–5) so that it reports the exit code of any completed job. Also, give the times (elapsed, kernel, and user) used so far by all jobs.

6–8. The job management functions have a defect that is difficult to fix. Suppose that a job is killed and the executive reuses its process ID before the process ID is removed from the job management file. There could be an `OpenProcess` on the process ID that now refers to a totally different process. The fix requires creating a helper process that holds duplicated handles for every created process so that the ID will not be reused. Another technique would be to include the process start time in the job management file. This time

should be the same as the process start time of the process obtained from the process ID. *Note:* Process IDs will be reused quickly. UNIX, however, increments a counter to get a new process ID, and IDs will repeat only after the 32-bit counter wraps around. Therefore, Windows programs cannot assume that IDs will not, for all practical purposes, be reused.

6–9. Modify `JobShell` so that job information is maintained in the registry rather than in a temporary file.

6–10. Enhance `JobShell` so that the `jobs` command will include a count of the number of handles that each job is using. *Hint:* Use `GetProcessHandleCount` (see MSDN).

6–11. `Jobbg` (in the `JobShell` listing) currently terminates a process if there is no room in the table for a new entry. Enhance the program to reserve a table location before creating the process, so as to avoid `TerminateProcess`.

6–12. `JobObjectShell` exhibits several anomalies and defects. Investigate and fix or explain them, if possible.

- Run 6–7 shows seven total processes, all active, after the first two jobs are started. This value is correct (do you agree?). After the jobs terminate, there are now 10 processes, none of which are active. Is this a bug (if so, is the bug in the program or in Windows?), or is the number correct?
- Program 6–7 shows plausible user time results in seconds (do you agree?). It obtains these results by dividing the total user time by 1,000,000, implying that the time is returned in microseconds. MSDN, however, says that the time is in 100 ns units, so the division should be by 10,000,000. Investigate. Is MSDN wrong?
- Does the limit on process time actually work, and is the program implemented correctly? `sortBT` (Program 5–1) is a time-consuming program for experimentation.

*This page intentionally left blank*

CHAPTER

# 7 Threads and Scheduling

The thread is the basic unit of execution in Windows, and a process can contain multiple, independent threads sharing the process's address space and other resources. Chapter 6 limited processes to a single thread, but there are many situations in which multiple threads are desirable. This chapter describes and illustrates Windows thread management and introduces program parallelism. The example programs use threads to simplify program design and to enhance performance. Chapter 8 continues with a description of synchronization objects and the performance impact, positive and negative. Chapter 9 examines performance tuning and trade-off issues and describes new NT6 locking and thread pool features. Chapter 10 describes advanced synchronization programming methods and models that greatly simplify the design and development of reliable multithreaded programs. The remaining chapters and example programs use threads and synchronization as a basic tool.

This chapter ends with a very brief discussion of fibers, which allow you to create separate tasks within a thread, followed by an introduction to parallelism. *Fibers are rarely used, and many readers might wish to skip the topic.*

## Thread Overview

A *thread* is an independent unit of execution within a process. The multithreaded programming challenge requires organization and coordination of thread execution to simplify programs and to take advantage of the inherent parallelism of the program and the host computer.

Traditionally, programs execute as a single thread of execution. While several processes can execute concurrently, as in the Chapter 6 examples, and even interact through mechanisms such as shared memory or pipes (Chapter 11), concurrent single-threaded processes have several disadvantages.

- It is expensive and time consuming for the OS to switch running processes, and, in cases such as the multiprocess search (`grepMP`, Program 6–1), the

processes are all executing the same program. Threads allow concurrent file or other processing within a single process, reducing overall system overhead.

- Except in the case of shared memory, processes are not tightly coupled, and it is difficult to share resources, such as open files.
- It is difficult and inefficient for single-threaded processes to manage several concurrent and interacting tasks, such as waiting for and processing user input, waiting for file or network input, and performing computation.
- I/O-bound programs, such as the Caesar cipher conversion program in Chapter 2 (`cci`, Program 2–3) are confined to a simple read-modify-write model. When you're processing sequential files, it can be more efficient to initiate as many read and write operations as possible. Windows also allows asynchronous overlapped I/O (Chapter 14), but threads can frequently achieve the same effect with less programming effort.
- The Windows executive will schedule independent threads on separate processors of a multiprocessor[1] computer, frequently improving performance by exploiting the multiple processors to execute application components concurrently.

This chapter discusses Windows threads and how to manage them. The examples illustrate thread usage with parallel file searching and a multithreaded sort. These two examples contrast I/O- and compute-intensive concurrent activities performed with threads. The chapter also presents an overview of Windows process and thread scheduling and concludes with a brief introduction to parallelism.

## Perspectives and Issues

This chapter and those that follow take the point of view that not only do threads make certain programs simpler to design and implement but, with attention to a few basic rules and programming models, threaded programs also can improve performance and be reliable, easy to understand, and maintainable. Thread management functions are very similar to the process management functions so that, as just one example, there is a `GetThreadExitCode` function that is comparable to `GetProcessExitCode`.

---

[1] Multiple CPUs are common, even on laptops. Several processors may be on a single "multicore" chip, and, in turn, a computer may have several multicore chips. In Edition 3, we used the term "symmetric multiprocessing" (SMP), but this does not accurately describe multiple multicore chips in a single computer. We use both "multiprocessor" and "multicore" from now on to describe multiple processors accessing common, shared memory.

This point of view is not universally accepted. Many writers and software developers mention thread risks and issues and prefer to use multiple processes when concurrency is required. Common issues and concerns include the following.

- Threads share storage and other resources within a process, so one thread can accidentally modify another thread's data, leading to defects such as race conditions and deadlocks.
- In certain circumstances, concurrency can drastically degrade, rather than improve, performance.
- Converting legacy single-threaded programs to exploit threads can be challenging, partly for the reasons above as well as lack of program understanding.

These concerns are real but are generally avoidable with careful design and programming, and many of the issues are inherent to concurrency, whether using threads within a process, multiple processes, or special-purpose techniques, such as Windows asynchronous I/O (Chapter 14).

## Thread Basics

Figure 6–1 in the previous chapter shows how threads exist in a process environment. Figure 7–1 illustrates threads by showing a multithreaded server that can process simultaneous requests from multiple networked clients; a distinct thread is dedicated to each client. A Chapter 11 example implements this model.

Threads within a process share data and code, but individual threads also have their own unique storage in addition to the shared data. Windows provides data for individual threads in several ways. Be aware, however, that the data is not totally protected from other threads within the process; the programmer must assure that threads do not access data assigned to other threads.

- Each thread has its own stack for function calls and other processing.
- The calling process can pass an argument (`Arg` in Figure 7–1), usually a pointer, to a thread at creation time. This argument is actually on the thread's stack.
- Each thread can allocate its own Thread Local Storage (TLS) indexes and can read and set TLS values. TLS, described later, provides small pointer arrays to threads, and a thread can access only its own TLS array. Among other advantages, TLS assures that threads will not modify one another's data.

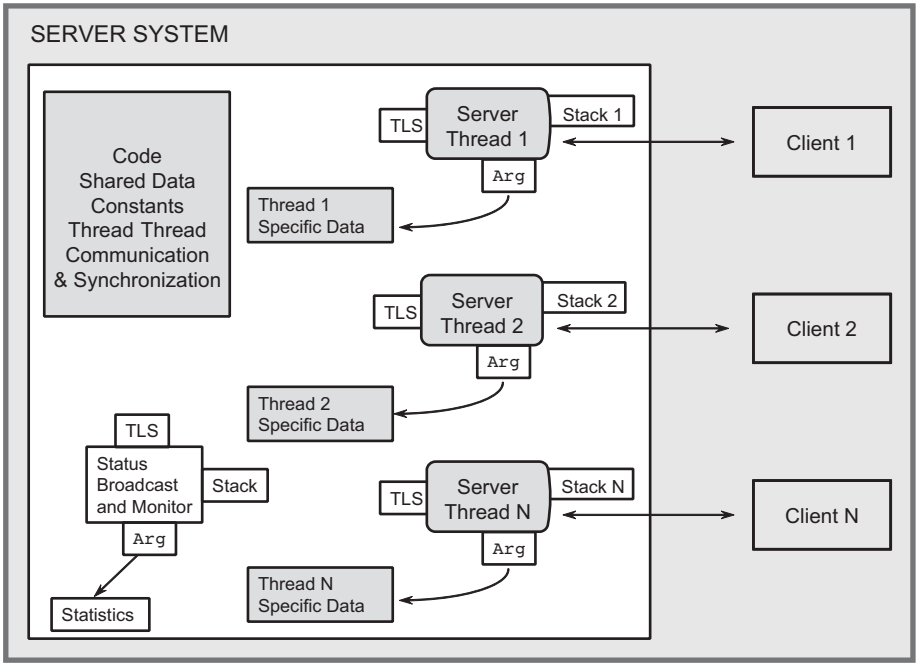


**Figure 7–1** Threads in a Server Environment

The thread argument can point to an arbitrary data structure. In Figure 7–1's server example, this structure might contain the current request and the thread's response to that request as well as other working storage.

Windows programs can exploit multiprocessor systems by allowing different threads, even from the same process, to run concurrently on separate processors. This capability, if used properly, can enhance performance, but without sufficient care and a good strategy to exploit multiple processors, execution can actually be slower than on a single-processor computer, as we'll see in Chapter 9.

## Thread Management

It should come as no surprise that threads, like any other Windows object, have handles and that there is a `CreateThread` system call to create an executable thread in the calling process's address space. As with processes, we will sometimes speak of "parent" and "child" threads, although the OS does not make any such distinction. `CreateThread` has several unique requirements.

## CreateThread

The `CreateThread` function allows you to:

- Specify the thread's start address within the process's code.
- Specify the stack size, and the stack space is allocated from the process's virtual address space. The default stack size is the parent's virtual memory stack size (normally 1MB). One page is initially committed to the stack. New stack pages are committed as required until the stack reaches its maximum size and cannot grow anymore.
- Specify a pointer to a thread argument. The argument can be nearly anything and is interpreted by the thread and its parent.
- `CreateThread` returns a thread's ID value and its handle. A `NULL` handle value indicates a failure.

```
HANDLE CreateThread (
   LPSECURITY_ATTRIBUTES lpsa,
   DWORD dwStackSize,
   LPTHREAD_START_ROUTINE lpStartAddr,
   LPVOID lpThreadParm,
   DWORD dwCreationFlags,
   LPDWORD lpThreadId)
```

### *Parameters*

`lpsa` is the familiar security attributes structure.

`dwStackSize` is the byte size of the new thread's stack. Use `0` to default to the primary thread's stack size.

`lpStartAddr` points to the thread function (within the calling process) to be executed. This function accepts a single pointer argument and returns a 32-bit `DWORD` exit code. The thread can interpret the argument as a `DWORD` or a pointer. The thread function signature, then, is as follows:

```
DWORD WINAPI ThreadFunc (LPVOID)
```

lpThreadParm is the pointer passed as the thread argument and is interpreted by the thread and its parent, normally as a pointer to an argument structure.

dwCreationFlags, if 0, means that the thread is ready to run immediately. If dwCreationFlags is CREATE_SUSPENDED, the new thread will be in the suspended state, requiring a ResumeThread function call to move the thread to the ready state.

lpThreadId points to a DWORD that receives the new thread's identifier. The pointer can also be NULL, indicating that no thread ID will be returned.

The CreateRemoteThread function allows a thread to be created in another process. Compared with CreateThread, there is an additional parameter for the process handle, and the function addresses must be in the target process's address space. CreateRemoteThread is one of several interesting, and potentially dangerous, ways for one process to affect another directly, and it might be useful in writing, for example, a debugger. There is no way to call this function usefully and safely in normal applications. Avoid it!

## ExitThread

All threads in a process can exit using the ExitThread function. A common alternative, however, is for a thread to return from the thread function using the exit code as the return value. The thread's stack is deallocated and all handles referring to the thread are signaled. If the thread is linked to one or more DLLs (either implicitly or explicitly), then the DllMain functions (Chapter 4) of each DLL will be called with DLL_THREAD_DETACH as the "reason."

```
VOID ExitThread (DWORD dwExitCode)
```

When the last thread in a process exits, the process itself terminates.

One thread can terminate another thread with the TerminateThread function, but the thread's resources will not be deallocated, completion handlers do not execute, and there is no notification to attached DLLs. It is best if the thread terminates itself with a return statement; TerminateThread usage is strongly discouraged, and it has the same disadvantages as TerminateProcess.

## GetExitCodeThread

A terminated thread (again, a thread normally should terminate itself) will continue to exist until the last handle to it is closed using CloseHandle. Any other thread, perhaps one waiting for some other thread to terminate, can retrieve the exit code.