CHAPTER

6 Process Management

A process contains its own independent virtual address space with both code and data, protected from other processes. Each process, in turn, contains one or more independently executing *threads*. A thread running within a process can create new threads, create new independent processes, and manage communication and synchronization between the objects.

By creating and managing processes, applications can have multiple, concurrent tasks processing files, performing computations, or communicating with other networked systems. It is even possible to exploit multiple processors to speed processing.

This chapter explains the basics of process management and also introduces the basic synchronization operations that will be used throughout the rest of the book.

Windows Processes and Threads

Every process contains one or more threads, and the Windows thread is the basic executable unit. Threads are scheduled on the basis of the usual factors: availability of resources such as CPUs and physical memory, priority, fairness, and so on. Windows has supported symmetric multiprocessing (SMP) since NT4, so threads can be allocated to separate processors within a system.

From the programmer's perspective, each Windows process includes resources such as the following components:

- One or more threads.
- A virtual address space that is distinct from other processes' address spaces, except where memory is explicitly shared. Note that shared memory-mapped files share physical memory, but the sharing processes will use different virtual addresses to access the mapped file.
- One or more code segments, including code in DLLs.

- One or more data segments containing global variables.
- Environment strings with environment variable information, such as the current search path.
- The process heap.
- Resources such as open handles and other heaps.

Each thread in a process shares code, global variables, environment strings, and resources. Each thread is independently scheduled, and a thread has the following elements:

- A stack for procedure calls, interrupts, exception handlers, and automatic storage.
- Thread Local Storage (TLS)—arrays of pointers giving each thread the ability to allocate storage to create its own unique data environment.
- An argument on the stack, from the creating thread, which is usually unique for each thread.
- A context structure, maintained by the kernel, with machine register values.

Figure 6–1 shows a process with several threads. This figure is schematic and does not indicate actual memory addresses, nor is it drawn to scale.

This chapter shows how to work with processes consisting of a single thread. Chapter 7 shows how to use multiple threads.

Note: Figure 6–1 is a high-level overview from the programmer's perspective. There are numerous technical and implementation details, and interested readers can find out more in *Inside Windows 2000* (Solomon and Russinovich).

A UNIX process is comparable to a Windows process with a single thread.

Threads, in the form of POSIX Pthreads, are a recent addition to UNIX implementations and are now nearly universally used. Stevens (1992) does not discuss threads; everything is done with processes.

Needless to say, vendors and others have provided various thread implementations for many years; they are not a new concept. Pthreads is, however, the most widely used standard, and proprietary implementations are obsolete.

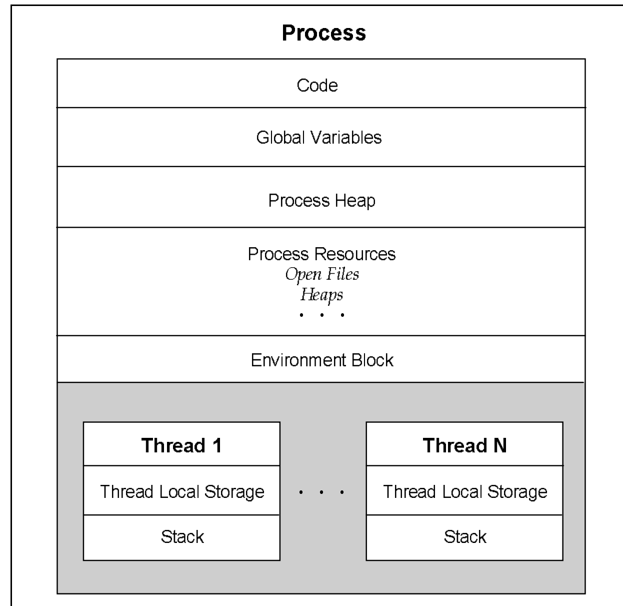


Figure 6–1 A Process and Its Threads

Process Creation

The fundamental Windows process management function is `CreateProcess`, which creates a process with a single thread. It is necessary to specify the name of an executable program file as part of the `CreateProcess` call.

It is common to speak of *parent* and *child* processes, but these relationships are not actually maintained by Windows. It is simply convenient to refer to the process that creates a child process as the parent.

`CreateProcess` has ten parameters to support its flexibility and power. Initially, it is simple to use default values. Just as with `CreateFile`, it is appropriate to explain all the `CreateProcess` parameters. Related functions then become easier to understand.

Note first that the function does not return a `HANDLE`; rather, two separate handles, one each for the process and the thread, are returned in a structure specified in the call. `CreateProcess` creates a new process with a *primary* thread. The example programs are always very careful to close both of these handles when they are no longer needed in order to avoid resource leaks; a common defect is to neglect to close the thread handle. Closing a thread handle, for instance, does not terminate the thread; the `CloseHandle` function only deletes the reference to the thread within the process that called `CreateProcess`.

Hart_ch06.fm Page 166 Tuesday, September 28, 2004 2:46 PM


```
BOOL CreateProcess (
   LPCTSTR lpApplicationName,
   LPTSTR lpCommandLine,
   LPSECURITY_ATTRIBUTES lpsaProcess,
   LPSECURITY_ATTRIBUTES lpsaThread,
   BOOL bInheritHandles,
   DWORD dwCreationFlags,
   LPVOID lpEnvironment,
   LPCTSTR lpCurDir,
   LPSTARTUPINFO lpStartupInfo,
   LPPROCESS_INFORMATION lpProcInfo)
```

Return: `TRUE` only if the process and thread are successfully created.

Parameters

Some parameters require extensive explanations in the following sections, and many are illustrated in the program examples.

`lpApplicationName` and `lpCommandLine` (this is an `LPTSTR` and not an `LPCTSTR`) together specify the executable program and the command line arguments, as explained in the next section.

`lpsaProcess` and `lpsaThread` point to the process and thread security attribute structures. `NULL` values imply default security and will be used until Chapter 15, which covers Windows security.

`bInheritHandles` indicates whether the new process inherits copies of the calling process's inheritable open handles (files, mappings, and so on). Inherited handles have the same attributes as the originals and are discussed in detail in a later section.

`dwCreationFlags` combines several flags, including the following.

- `CREATE_SUSPENDED` indicates that the primary thread is in a suspended state and will run only when `ResumeThread` is called.
- `DETACHED_PROCESS` and `CREATE_NEW_CONSOLE` are mutually exclusive; don't set both. The first flag creates a process without a console, and the second flag gives the new process a console of its own. If neither flag is set, the process inherits the parent's console.

Hart_ch06.fm Page 167 Tuesday, September 28, 2004 2:46 PM

- CREATE_NEW_PROCESS_GROUP specifies that the new process is the root of a new process group. All processes in a group receive a console control signal (Ctrl-c or Ctrl-break) if they all share the same console. Console control handlers were described in Chapter 4 and illustrated in Program 4–5. These process groups have similarities to UNIX process groups and are described later in this chapter.

Several of the flags control the priority of the new process's threads. The possible values are explained in more detail at the end of Chapter 7. For now, just use the parent's priority (specify nothing) or NORMAL_PRIORITY_CLASS.

lpEnvironment points to an environment block for the new process. If NULL, the process uses the parent's environment. The environment block contains name and value strings, such as the search path.

lpCurDir specifies the drive and directory for the new process. If NULL, the parent's working directory is used.

lpStartupInfo specifies the main window appearance and standard device handles for the new process. Use the parent's information, which is obtained from GetStartupInfo. Alternatively, zero out the associated STARTUPINFO structure before calling CreateProcess. To specify the standard input, output, and error handles, set the standard handler fields (hStdInput, hStdOutput, and hStdError) in the STARTUPINFO structure. For this to be effective, also set another STARTUPINFO member, dwFlags, to STARTF_USESTDHANDLES, and set all the handles that the child process will require. Be certain that the handles are inheritable and that the CreateProcess bInheritHandles flag is set. The Inheritable Handles subsection gives more information and an example.

lpProcInfo specifies the structure for containing the returned process, thread handles, and identification. The PROCESS_INFORMATION structure is as follows:

```
typedef struct PROCESS_INFORMATION {
   HANDLE hProcess;
   HANDLE hThread;
   DWORD dwProcessId;
   DWORD dwThreadId;
} PROCESS_INFORMATION;
```

Why do processes and threads need handles in addition to IDs? The ID is unique to the object for its entire lifetime and in all processes, whereas a given process may have several handles, each having distinct attributes, such as

Hart_ch06.fm Page 168 Tuesday, September 28, 2004 2:46 PM

security access. For this reason, some process management functions require IDs, and others require handles. Furthermore, process handles are required for the general-purpose, handle-based functions. Examples include the wait functions discussed later in this chapter, which allow waiting on handles for several different object types, including processes. Just as with file handles, process and thread handles should be closed when no longer required.

Note: The new process obtains environment, working directory, and other information from the `CreateProcess` call. Once this call is complete, any changes in the parent will not be reflected in the child process. For example, the parent might change its working directory after the `CreateProcess` call, but the child process working directory will not be affected, unless the child changes its own working directory. The two processes are entirely independent.

The UNIX and Windows process models are considerably different. First, Windows has no equivalent to the UNIX `fork` function, which makes a copy of the parent, including the parent's data space, heap, and stack. `fork` is difficult to emulate exactly in Windows, and, while this may seem to be a limitation, `fork` is also difficult to use in a multithreaded UNIX system because there are numerous problems with creating an exact replica of a multithreaded system with exact copies of all threads and synchronization objects, especially on an SMP system. Therefore, `fork`, by itself, is not really appropriate in any multithreaded system.

`CreateProcess` is, however, similar to the common UNIX sequence of successive calls to `fork` and `execl` (or one of five other `exec` functions). In contrast to Windows, the search directories in UNIX are determined entirely by the `PATH` environment variable.

As previously mentioned, Windows does not maintain parent-child relationships among processes. Thus, a child process will continue to run after the creating parent process terminates. Furthermore, there are no process groups in Windows. There is, however, a limited form of process group that specifies all the processes to receive a console control event.

Windows processes are identified both by handles and by process IDs, whereas UNIX has no process handles.

Specifying the Executable Image and the Command Line

Either `lpApplicationName` or `lpCommandLine` specifies the executable image name. The rules are as follows.

- `lpApplicationName`, if not `NULL`, is the name of the executable. Quotation marks can be used if the image name contains spaces. More detailed rules are described below.
- Otherwise, the executable is the first token in `lpCommandLine`.

Hart_ch06.fm Page 169 Tuesday, September 28, 2004 2:46 PM

Usually, only `lpCommandLine` is specified, with `lpApplicationName` being `NULL`. Nonetheless, there are detailed rules for `lpApplicationName`.

- If `lpApplicationName` is not `NULL`, it specifies the executable module. Specify the full path and file name, or use a partial name and the current drive and directory will be used; there is no additional searching. Include the file extension, such as `.EXE` or `.BAT`, in the name.
- If the `lpApplicationName` string is `NULL`, the first white-space-delimited token in `lpCommandLine` is the program name. If the name does not contain a full directory path, the search sequence is as follows:
  1. The directory of the current process's image
  2. The current directory
  3. The Windows system directory, which can be retrieved with `GetSystemDirectory`
  4. The Windows directory, which is retrievable with `GetWindowsDirectory`
  5. The directories as specified in the environment variable `PATH`

The new process can obtain the command line using the usual `argv` mechanism, or it can invoke `GetCommandLine` to obtain the command line as a single string.

Notice that the command line is not a constant string. This is consistent with the fact that the `argv` parameters to the main program are not constant. A program could modify its arguments, although it is advisable to make any changes in a copy of the argument string.

The new process is not required to be built with the same `UNICODE` definition as that of the parent process. All combinations are possible. Using `_tmain` as discussed in Chapter 2 is helpful in developing code for either UNICODE or ASCII operation.

Inheritable Handles

Frequently, a child process requires access to an object referenced by a handle in the parent; if this handle is inheritable, the child can receive a copy of the parent's open handle. The standard input and output handles are frequently shared with the child in this way. To make a handle inheritable so that a child receives and can use a copy requires several steps.

The `bInheritHandles` flag on the `CreateProcess` call determines whether the child process will inherit copies of the inheritable handles of open files,

Hart_ch06.fm Page 170 Tuesday, September 28, 2004 2:46 PM

processes, and so on. The flag can be regarded as a master switch applying to all handles.

It is also necessary to make an individual handle inheritable; it is not done by default. To create an inheritable handle, use a SECURITY_ATTRIBUTES structure at creation time or duplicate an existing handle.

The SECURITY_ATTRIBUTES structure has a flag, bInheritHandle, that should be set to TRUE. Also, recall that nLength should be set to sizeof (SECURITY_ATTRIBUTES).

The following code segment shows how an inheritable file or other handle is typically created. In this example, the security descriptor within the security attributes structure is NULL; Chapter 15 shows how to include a security descriptor.

```
HANDLE h1, h2, h3;
SECURITY_ATTRIBUTES sa =
    {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE };
...
h1 = CreateFile (..., &sa, ... ); /* Inheritable. */
h2 = CreateFile (..., NULL, ... ); /* Not inheritable. */
h3 = CreateFile (..., &sa, ...);
    /* Inheritable. sa can be reused. */
```

A child process still needs to know the value of an inheritable handle, so the parent needs to communicate handle values to the child using an interprocess communication (IPC) mechanism or by assigning the handle to standard I/O in the STARTUPINFO structure, as is done in the first example of this chapter (Program 6–1) and in several additional examples throughout the book. This is generally the preferred technique because it allows I/O redirection in a standard way and no changes are needed in the child program.

Alternatively, nonfile handles and handles that are not used to redirect standard I/O can be converted to text and placed in a command line or in an environment variable. This approach is valid if the handle is inheritable because both parent and child processes identify the handle with the same handle value. Exercise 6–2 suggests how to demonstrate this, and a solution is presented on the book's Web site.

The inherited handles are distinct copies. Therefore, a parent and child might be accessing the same file using different file pointers. Furthermore, each of the two processes can and should close its own handle.

Figure 6–2 shows how two processes can have distinct handle tables with two distinct handles associated with the same file or other object. Process 1 is the parent, and Process 2 is the child. The handles will have identical values in both

Hart_ch06.fm Page 171 Tuesday, September 28, 2004 2:46 PM

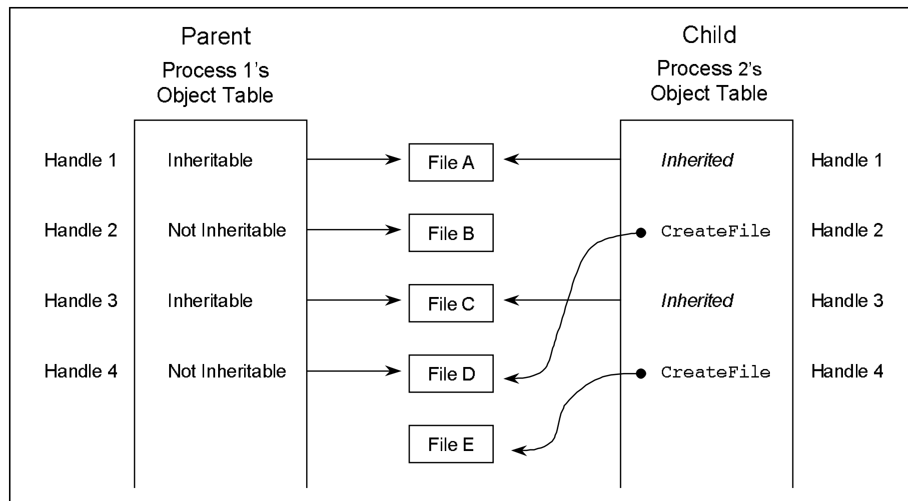


Figure 6–2 Process Handle Tables

processes if the child's handle has been inherited, as is the case with Handles 1 and 3.

On the other hand, the handle values may be distinct. For example, there are two handles for File D, where Process 2 obtained a handle by calling `CreateFile` rather than by inheritance. Finally, as is the case with Files B and E, one process may have a handle to an object while the other does not. This would be the case when the child process creates the handle or when a handle is duplicated from one process to another, as described in the upcoming Duplicating Handles section.

Process Handle Counts

A common programming error is to neglect to close handles when they are no longer needed; this can result in resource leakage, which in turn can degrade performance, cause program failures, and even impact other processes. NT 5.1 added a new function that allows you to determine how many handles any process has open. In this way, you can monitor your own process or other processes.

Here is the function definition, which is self-explanatory:

```
BOOL GetProcessHandleCount (
   HANDLE hProcess,
   PDWORD pdwHandleCount)
```

Hart_ch06.fm Page 172 Tuesday, September 28, 2004 2:46 PM

Process Identities

A process can obtain the identity and handle of a new child process from the PROCESS_INFORMATION structure. Closing the child handle does not, of course, destroy the child process; it destroys only the parent's access to the child. A pair of functions is used to obtain current process identification.

```
HANDLE GetCurrentProcess (VOID)

DWORD GetCurrentProcessId (VOID)
```

GetCurrentProcess actually returns a *pseudohandle* and is not inheritable. This value can be used whenever a process needs its own handle. You create a real process handle from a process ID, including the one returned by GetCurrentProcessId, by using the OpenProcess function. As is the case with all sharable objects, the open call will fail if you do not have sufficient security rights.

```
HANDLE OpenProcess (
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   DWORD dwProcessId)
```

Return: A process handle, or NULL on failure.

Parameters

dwDesiredAccess determines the handle's access to the process. Some of the values are as follows.

- SYNCHRONIZE—This flag enables processes to wait for the process to terminate using the wait functions described later in this chapter.
- PROCESS_ALL_ACCESS—All the access flags are set.
- PROCESS_TERMINATE—It is possible to terminate the process with the TerminateProcess function.
- PROCESS_QUERY_INFORMATION—The handle can be used by GetExitCodeProcess and GetPriorityClass to obtain process information.

Hart_ch06.fm Page 173 Tuesday, September 28, 2004 2:46 PM

bInheritHandle specifies whether the new handle is inheritable. dwProcessId is the identifier of the process requiring a handle.

Finally, a running process can determine the full pathname of the executable used to run it with GetModuleFileName or GetModuleFileNameEx, using a NULL value for the hModule parameter. A call from within a DLL will return the DLL's file name, not that of the .EXE file that uses the DLL.

Duplicating Handles

The parent and child processes may require different access to an object identified by a handle that the child inherits. A process may also need a real, inheritable process handle—rather than the pseudohandle produced by GetCurrent-Process—for use by a child process. To address this issue, the parent process can create a duplicate handle with the desired access and inheritability. Here is the function to duplicate handles:

```
BOOL DuplicateHandle (
   HANDLE hSourceProcessHandle,
   HANDLE hSourceHandle,
   HANDLE hTargetProcessHandle,
   LPHANDLE lphTargetHandle,
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   DWORD dwOptions)
```

Upon completion, lphTargetHandle points to a copy of the original handle, hSourceHandle. hSourceHandle is a handle in the process indicated by hSourceProcessHandle and must have PROCESS_DUP_HANDLE access; DuplicateHandle will fail if the source handle does not exist in the source process. The new handle, which is pointed to by lphTargetHandle, is valid in the target process, hTargetProcessHandle. Note that three processes are involved, including the calling process. Frequently, these target and source processes are the calling process, and the handle is obtained from GetCurrentProcess. Also notice that it is possible to create a handle in another process; if you do this, you then need a mechanism for informing the other process of the new handle's identity.

DuplicateHandle can be used for any handle type.

If dwDesiredAccess is not overridden by DUPLICATE_SAME_ACCESS in dwOptions, it has many possible values (see the MSDN library on-line help).

Hart_ch06.fm Page 174 Tuesday, September 28, 2004 2:46 PM

`dwOptions` is any combination of two flags.

- `DUPLICATE_CLOSE_SOURCE` causes the source handle to be closed.
- `DUPLICATE_SAME_ACCESS` causes `dwDesiredAccess` to be ignored.

Reminder: The Windows kernel maintains a reference count for all objects; this count represents the number of distinct handles referring to the object. This count is not available to the application program, however. An object cannot be destroyed until the last handle is closed and the reference count becomes zero. Inherited and duplicate handles are both distinct from the original handles and are represented in the reference count. Program 6–1, later in the chapter, uses inheritable handles. On the other hand, a handle passed from one process to another through some form of IPC is not a distinct handle, so if one process closes the handle, the handle cannot be used by any other process. This technique is rarely used, but Exercise 6–2 uses IPC to send the value of the inherited handle to another process.

Next, it is necessary to learn how to determine whether a process has terminated.

Exiting and Terminating a Process

After a process is complete, the process, or more accurately, a thread running in the process, can call `ExitProcess` with an exit code.

```
VOID ExitProcess (UINT uExitCode)
```

This function does not return. Rather, the calling process and all its threads terminate. Termination handlers are ignored, but there will be detach calls to `DllMain` (see Chapter 5). The exit code is associated with the process. A `return` from the main program, with a return value, will have the same effect as calling `ExitProcess` with the return value as the exit code.

Another process can use `GetExitCodeProcess` to determine the exit code.

```
BOOL GetExitCodeProcess (
   HANDLE hProcess,
   LPDWORD lpExitCode)
```

The process identified by hProcess must have PROCESS_QUERY_INFORMATION access (see OpenProcess, discussed earlier). lpExitCode points to the DWORD that receives the value. One possible value is STILL_ACTIVE, meaning that the process has not terminated.

Finally, one process can terminate another process if the handle has PROCESS_TERMINATE access. The terminating function also specifies the exit code.

```
BOOL TerminateProcess (
   HANDLE hProcess,
   UINT uExitCode)
```

Caution: Before exiting from a process, be certain to free all resources that might be shared with other processes. In particular, the synchronization resources of Chapter 8 (mutexes, semaphores, and events) must be handled carefully. SEH (Chapter 4) can be helpful in this regard, and the ExitProcess call can be in the handler. However, __finally and __except handlers are *not* executed when ExitProcess is called, so it is not a good idea to exit from inside a program. TerminateProcess is especially risky because the terminated process will not have an opportunity to execute its SEH or DLL DllMain functions. Console control handlers (Chapter 4 and later in this chapter) are a limited alternative, allowing one process to send a signal to another process, which can then shut itself down cleanly.

Program 6–3 shows a technique whereby processes cooperate. One process sends a shutdown request to a second process, which proceeds to perform an orderly shutdown.

UNIX processes have a process ID, or pid, comparable to the Windows process ID. getpid is similar to GetCurrentProcessId, but there are no Windows equivalents to getppid and getgpid because Windows has no process parents or groups.

Conversely, UNIX does not have process handles, so it has no functions comparable to GetCurrentProcess or OpenProcess.

UNIX allows open file descriptors to be used after an exec if the file descriptor does not have the close-on-exec flag set. This applies only to file descriptors, which are then comparable to inheritable file handles.

UNIX exit, actually in the C library, is similar to ExitProcess; to terminate another process, signal it with SIGKILL.

Hart_ch06.fm Page 176 Tuesday, September 28, 2004 2:46 PM

Waiting for a Process to Terminate

The simplest, and most limited, method of synchronizing with another process is to wait for that process to complete. The general-purpose Windows wait functions introduced here have several interesting features.

- The functions can wait for many different types of objects; process handles are just the first use of the wait functions.
- The functions can wait for a single process, the first of several specified processes, or all processes in a group to complete.
- There is an optional time-out period.

The two general-purpose wait functions wait for synchronization objects to become *signaled*. The system sets a process handle, for example, to the signaled state when the process terminates or is terminated. The wait functions, which will get lots of future use, are as follows:

```
DWORD WaitForSingleObject (
   HANDLE hObject,
   DWORD dwMilliseconds)
```

```
DWORD WaitForMultipleObjects (
   DWORD nCount,
   CONST HANDLE *lpHandles,
   BOOL fWaitAll,
   DWORD dwMilliseconds)
```

Return: The cause of the wait completion, or `0XFFFFFFFF` for an error (use `GetLastError` for more information).

Specify either a single process handle (`hObject`) or an array of distinct object handles in the array referenced by `lpHandles`. `nCount`, the size of the array, should not exceed `MAXIMUM_WAIT_OBJECTS` (defined as 64 in `WINNT.H`).

`dwMilliseconds` is the time-out period in milliseconds. A value of 0 means that the function returns immediately after testing the state of the specified

Hart_ch06.fm Page 177 Tuesday, September 28, 2004 2:46 PM

objects, thus allowing a program to poll for process termination. Use INFINITE for no time-out to wait until a process terminates.

fWaitAll, a parameter of the second function, specifies (if TRUE) that it is necessary to wait for all processes, rather than only one, to terminate.

The possible successful return values for this function are as follows.

- WAIT_OBJECT_0 means that the handle is signaled in the case of WaitForSingleObject or all nCount objects are simultaneously signaled in the special case of WaitForMultipleObjects with fWaitAll set to TRUE.
- WAIT_OBJECT_0+n, where 0 ≤ n < nCount. Subtract WAIT_OBJECT_0 from the return value to determine which process terminated when waiting for any of a group of processes to terminate. If several handles are signaled, the returned value is the smallest possible value. WAIT_ABANDONED_0 is a possible base value when using mutex handles; see Chapter 8.
- WAIT_TIMEOUT indicates that the time-out period elapsed before the wait could be satisfied by signaled handle(s).
- WAIT_FAILED indicates that the call failed; for example, the handle may not have SYNCHRONIZE access.
- WAIT_ABANDONED_0 is not possible with processes. This value is discussed in Chapter 8 along with mutex handles.

Determine the exit code of a process using GetExitCodeProcess, as described in the preceding section.

Environment Blocks and Strings

Figure 6–1 includes the process environment block. The environment block contains a sequence of strings of the form

```
Name = Value
```

Each environment string, being a string, is NULL-terminated, and the entire block of strings is itself NULL-terminated. PATH is one example of a commonly used environment variable.

To pass the parent's environment to a child process, set lpEnvironment to NULL in the CreateProcess call. Any process, in turn, can interrogate or modify its environment variables or add new environment variables to the block.

The two functions used to get and set variables are as follows:

```
DWORD GetEnvironmentVariable (
   LPCTSTR lpName,
   LPTSTR lpValue,
   DWORD cchValue)

BOOL SetEnvironmentVariable (
   LPCTSTR lpName,
   LPCTSTR lpValue)
```

lpName is the variable name. On setting a value, the variable is added to the block if it does not exist and if the value is not NULL. If, on the other hand, the value is NULL, the variable is removed from the block. The "=" character cannot appear in a value string.

GetEnvironmentVariable returns the length of the value string, or 0 on failure. If the lpValue buffer is not long enough, as indicated by cchValue, then the return value is the number of characters actually required to hold the complete string. Recall that GetCurrentDirectory (Chapter 2) uses a similar mechanism.

Process Security

Normally, CreateProcess gives PROCESS_ALL_ACCESS rights. There are, however, several specific rights, including PROCESS_QUERY_INFORMATION, CREATE_PROCESS, PROCESS_TERMINATE, PROCESS_SET_INFORMATION, DUPLICATE_HANDLE, and CREATE_THREAD. In particular, it can be useful to limit PROCESS_TERMINATE rights to the parent process given the frequently mentioned dangers of terminating a running process. Chapter 15 describes security attributes for processes and other objects.

UNIX waits for process termination using wait and waitpid, but there are no time-outs even though waitpid can poll (there is a nonblocking option). These functions wait only for child processes, and there is no equivalent to the multiple wait on a collection of processes, although it is possible to wait for all processes in a process group. One slight difference is that the exit code is returned with wait and waitpid, so there is no need for a separate function equivalent to GetExit-CodeProcess.

UNIX also supports environment strings similar to those in Windows. getenv (in the C library) has the same functionality as GetEnvironmentVariable except

that the programmer must be sure to have a sufficiently large buffer. `putenv`, `setenv`, and `unsetenv` (not in the C library) are different ways to add, change, and remove variables and their values, with functionality equivalent to `SetEnvironmentVariable`.

Example: Parallel Pattern Searching

Now is the time to put Windows processes to the test. This example, `grepMP`, creates processes to search for patterns in files, one process per search file. The pattern search program is modeled after the UNIX `grep` utility, although the technique would apply to any program that uses standard output. The search program should be regarded as a black box and is simply an executable program to be controlled by a parent process.

The command line to the program is of the form

```
grepMP pattern F1 F2 ... FN
```

The program, Program 6–1, performs the following processing:

- Each input file, `F1` to `FN`, is searched using a separate process running the same executable. The program creates a command line of the form `grep` *`pattern`* `FK`.
- The handle of the temporary file, specified to be inheritable, is assigned to the `hStdOutput` field of the new process's start-up information structure.
- Using `WaitForMultipleObjects`, the program waits for all search processes to complete.
- As soon as all searches are complete, the results (temporary files) are displayed in order, one at a time. A process to execute the `cat` utility (Program 2–3) outputs the temporary file.
- `WaitForMultipleObjects` is limited to `MAXIMUM_WAIT_OBJECTS` (64) handles, so it is called multiple times.
- The program uses the `grep` process exit code to determine whether a specific process detected the pattern.

Figure 6–3 shows the processing performed by Program 6–1.

Hart_ch06.fm Page 180 Tuesday, September 28, 2004 2:46 PM

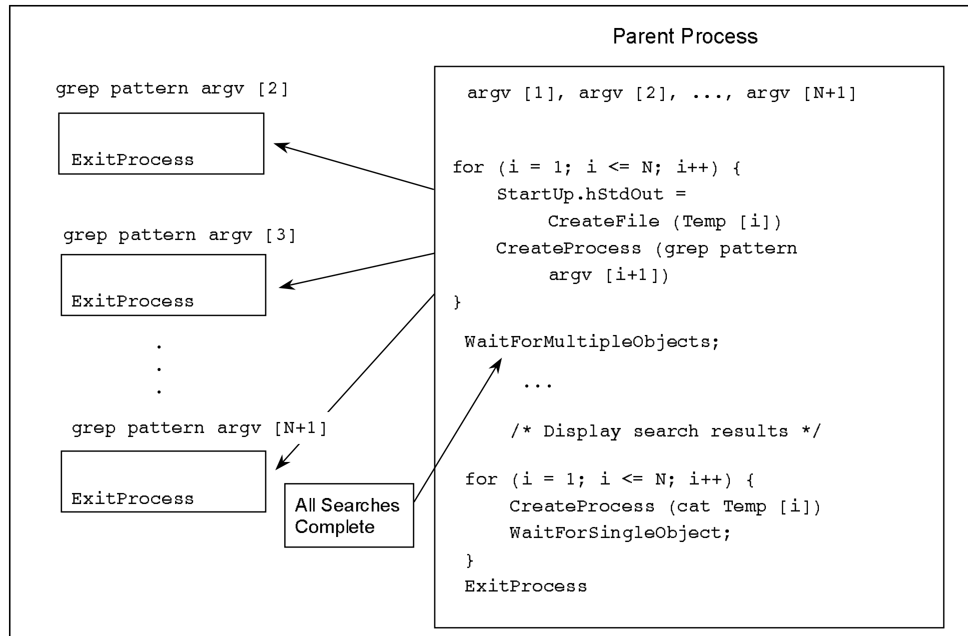


Figure 6–3 File Searching Using Multiple Processes

Program 6–1 `grepMP`: Parallel Searching

```
/* Chapter 6. grepMP. */
/* Multiple process version of grep command. */

#include "EvryThng.h"
int _tmain (DWORD argc, LPTSTR argv [])
/* Create a separate process to search each file on the
   command line. Each process is given a temporary file,
   in the current directory, to receive the results. */
{
   HANDLE hTempFile;
   SECURITY_ATTRIBUTES StdOutSA = /* SA for inheritable handle. */
         {sizeof (SECURITY_ATTRIBUTES), NULL, TRUE};
   TCHAR CommandLine [MAX_PATH + 100];
   STARTUPINFO StartUpSearch, StartUp;
   PROCESS_INFORMATION ProcessInfo;
   DWORD iProc, ExCode;
   HANDLE *hProc; /* Pointer to an array of proc handles. */
   typedef struct {TCHAR TempFile [MAX_PATH];} PROCFILE;
   PROCFILE *ProcFile; /* Pointer to array of temp file names. */
```

Hart_ch06.fm Page 181 Tuesday, September 28, 2004 2:46 PM


```
GetStartupInfo (&StartUpSearch);
GetStartupInfo (&StartUp);
ProcFile = malloc ((argc - 2) * sizeof (PROCFILE));
hProc = malloc ((argc - 2) * sizeof (HANDLE));

/* Create a separate "grep" process for each file. */
for (iProc = 0; iProc < argc - 2; iProc++) {
   _stprintf (CommandLine, _T ("%s%s %s"),
      _T ("grep "), argv [1], argv [iProc + 2]);
   GetTempFileName (_T ("."), _T ("gtm"), 0,
         ProcFile [iProc].TempFile); /* For search results. */
   hTempFile = /* This handle is inheritable */
      CreateFile (ProcFile [iProc].TempFile,
         GENERIC_WRITE,
         FILE_SHARE_READ | FILE_SHARE_WRITE, &StdOutSA,
         CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
   StartUpSearch.dwFlags = STARTF_USESTDHANDLES;
   StartUpSearch.hStdOutput = hTempFile;
   StartUpSearch.hStdError = hTempFile;
   StartUpSearch.hStdInput = GetStdHandle (STD_INPUT_HANDLE);

   /* Create a process to execute the command line. */
   CreateProcess (NULL, CommandLine, NULL, NULL,
         TRUE, 0, NULL, NULL, &StartUpSearch, &ProcessInfo);
   /* Close unwanted handles. */
   CloseHandle (hTempFile); CloseHandle (ProcessInfo.hThread);
   hProc [iProc] = ProcessInfo.hProcess;
}

/* Processes are all running. Wait for them to complete. */
for (iProc = 0; iProc < argc - 2; iProc += MAXIMUM_WAIT_OBJECTS)
   WaitForMultipleObjects ( /* Allows a large # of processes */
         min (MAXIMUM_WAIT_OBJECTS, argc - 2 - iProc),
         &hProc [iProc], TRUE, INFINITE);
/* Result files sent to std output using "cat." */
for (iProc = 0; iProc < argc - 2; iProc++) {
   if (GetExitCodeProcess(hProc [iProc], &ExCode) && ExCode==0) {
      /* Pattern was detected -- List results. */
      if (argc > 3) _tprintf (_T ("%s:\n"), argv [iProc + 2]);
      fflush (stdout); /* Multiple processes use stdout. */
      _stprintf (CommandLine, _T ("%s%s"),
            _T ("cat "), ProcFile [iProc].TempFile);
      CreateProcess (NULL, CommandLine, NULL, NULL,
            TRUE, 0, NULL, NULL, &StartUp, &ProcessInfo);
      WaitForSingleObject (ProcessInfo.hProcess, INFINITE);
      CloseHandle (ProcessInfo.hProcess);
      CloseHandle (ProcessInfo.hThread);
   }
```

Hart_ch06.fm Page 182 Tuesday, September 28, 2004 2:46 PM


```
        CloseHandle (hProc [iProc]);
        DeleteFile (ProcFile [iProc].TempFile);
    }
    free (ProcFile);
    free (hProc);
    return 0;
}
```

Processes in a Multiprocessor Environment

In Program 6–1, the processes and their primary (and only) threads run almost totally independently of one another. The only dependence is created at the end of the parent process as it waits for all the processes to complete so that the output files can be processed sequentially. Therefore, the Windows scheduler can and will run the process threads concurrently on the separate processors of an SMP system. This can result in substantial performance improvement when performance is measured as elapsed time to execute the program, and no explicit actions are required to get the performance improvement.

Appendix C shows some typical results. The performance improvement is not linear in terms of the number of processors due to overhead costs and the need to output the results sequentially. Nonetheless, the improvements are worthwhile and result automatically as a consequence of the program design, which delegates independent computational tasks to independent processes.

It is possible, however, to constrain the processes to specific processors if you wish to be sure that other processors are free to be allocated to other critical tasks. This can be accomplished using the processor affinity mask (see Chapter 9) in a job object. Job objects are described in a later section.

Finally, it is possible to create independent threads within a process, and these threads will also be scheduled on separate SMP processors. Chapter 7 describes threads and performance issues related to their use.

Process Execution Times

You can determine the amount of time that a process requires (elapsed, kernel, and user times) using the `GetProcessTimes` function, which is not available on Windows 9x.

Hart_ch06.fm Page 183 Tuesday, September 28, 2004 2:46 PM


```
BOOL GetProcessTimes (
   HANDLE hProcess,
   LPFILETIME lpCreationTime,
   LPFILETIME lpExitTime,
   LPFILETIME lpKernelTime,
   LPFILETIME lpUserTime)
```

The process handle can refer to a process that is still running or to one that has terminated. Elapsed time can be computed by subtracting the creation time from the exit time, as shown in the next example. The `FILETIME` type is a 64-bit item; create a union with a `LARGE_INTEGER` to perform the subtraction. The `lsW` example in Chapter 3 showed how to convert and display file times.

`GetThreadTimes` is similar and requires a thread handle for a parameter. Chapter 7 covers thread management.

Example: Process Execution Times

The next example (Program 6–2) is a command called `timep` (time print) that is similar to the UNIX `time` command (`time` is supported by the command prompt, so a different name is required). Elapsed, kernel, and system times can be printed, although only elapsed time is available on Windows 9x.

One use for this command is to compare the execution times and efficiencies of the various file copy and ASCII to Unicode functions implemented in previous chapters.

This program uses `GetCommandLine`, a Windows function that returns the complete command line as a single string rather than individual `argv` strings.

The program also uses a utility function, `SkipArg`, to scan the command line and skip past the executable name. The `SkipArg` listing is in Appendix A.

Program 6–2 uses the `GetVersionEx` function to determine the OS version. With Windows 9x and CE, only the elapsed time is available. The code for these systems is shown to illustrate that a program can, in some cases, be made to operate, at least partially, on a range of Windows versions.

Program 6–2 `timep`: Process Times

```
/* Chapter 6. timep. */

#include "EvryThng.h"
int _tmain (int argc, LPTSTR argv [])
{
```

Hart_ch06.fm Page 184 Tuesday, September 28, 2004 2:46 PM


```
STARTUPINFO StartUp;
PROCESS_INFORMATION ProcInfo;
union { /* Structure required for file time arithmetic. */
   LONGLONG li;
   FILETIME ft;
} CreateTime, ExitTime, ElapsedTime;
FILETIME KernelTime, UserTime;
SYSTEMTIME ElTiSys, KeTiSys, UsTiSys, StartTimeSys, ExitTimeSys;
LPTSTR targv = SkipArg (GetCommandLine ());
OSVERSIONINFO OSVer;
BOOL IsNT;
HANDLE hProc;

OSVer.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
GetVersionEx (&OSVer);
IsNT = (OSVer.dwPlatformId == VER_PLATFORM_WIN32_NT);
/* NT (all versions) returns VER_PLATFORM_WIN32_NT. */
GetStartupInfo (&StartUp);
GetSystemTime (&StartTimeSys);

/* Execute the command line; wait for process to complete. */
CreateProcess (NULL, targv, NULL, NULL, TRUE,
      NORMAL_PRIORITY_CLASS, NULL, NULL, &StartUp, &ProcInfo);

/* Assure that we have all REQUIRED access to the process. */
DuplicateHandle (GetCurrentProcess (), ProcInfo.hProcess,
      GetCurrentProcess (), &hProc,
      PROCESS_QUERY_INFORMATION | SYNCHRONIZE, FALSE, 0);
WaitForSingleObject (hProc, INFINITE);
GetSystemTime (&ExitTimeSys);

if (IsNT) { /* W NT. Elapsed, Kernel, & User times. */
   GetProcessTimes (hProc, &CreateTime.ft,
         &ExitTime.ft, &KernelTime, &UserTime);
   ElapsedTime.li = ExitTime.li - CreateTime.li;
   FileTimeToSystemTime (&ElapsedTime.ft, &ElTiSys);
   FileTimeToSystemTime (&KernelTime, &KeTiSys);
   FileTimeToSystemTime (&UserTime, &UsTiSys);
   _tprintf (_T ("Real Time: %02d:%02d:%02d:%03d\n"),
         ElTiSys.wHour, ElTiSys.wMinute, ElTiSys.wSecond,
         ElTiSys.wMilliseconds);
   _tprintf (_T ("User Time: %02d:%02d:%02d:%03d\n"),
         UsTiSys.wHour, UsTiSys.wMinute, UsTiSys.wSecond,
         UsTiSys.wMilliseconds);
   _tprintf (_T ("Sys Time: %02d:%02d:%02d:%03d\n"),
         KeTiSys.wHour, KeTiSys.wMinute, KeTiSys.wSecond,
         KeTiSys.wMilliseconds);
} else {
   /* Windows 9x and CE. Elapsed time only. */
   ...
```

Hart_ch06.fm Page 185 Tuesday, September 28, 2004 2:46 PM


```
    }
    CloseHandle (ProcInfo.hThread); CloseHandle (ProcInfo.hProcess);
    CloseHandle (hProc);
    return 0;
}
```

Using the `timep` Command

`timep` can now be used to compare the various ASCII to Unicode file copy and sorting utilities such as `atou` (Program 2-4) and `sortMM` (Program 5–5). Appendix C summarizes and briefly analyzes some results.

Notice that measuring a program such as `grepMP` (Program 6–1) gives kernel and user times only for the parent process. Job objects, described near the end of this chapter, allow you to collect information on a group of processes. Appendix C shows that, on an SMP system, performance can improve as the separate processes, or more accurately, threads, run on different processors. There can also be performance gains if the files are on different physical drives.

Generating Console Control Events

Terminating a process can cause problems because the terminated process cannot clean up. SEH does not help because there is no general method for one process to cause an exception in another.[1] Console control events, however, allow one process to send a console control signal, or event, to another process in certain limited circumstances. Program 4–5 illustrated how a process can set up a handler to catch such a signal, and the handler could generate an exception. In that example, the user generated a signal from the user interface.

It is possible, then, for a process to generate a signal event in another specified process or set of processes. Recall the `CreateProcess` creation flag value, `CREATE_NEW_PROCESS_GROUP`. If this flag is set, the new process ID identifies a group of processes, and the new process is the *root* of the group. All new processes created by the parent are in this new group until another `CreateProcess` call uses the `CREATE_NEW_PROCESS_GROUP` flag. The grouped processes are similar to UNIX process groups.

One process can generate a `CTRL_C_EVENT` or `CTRL_BREAK_EVENT` in a specified process group, identifying the group with the root process ID. The target processes must have the same console as that of the process generating the event. In

[1] Chapter 10 shows an indirect way for one thread to cause an exception in another thread, and the same technique can be used between threads in different processes.

Hart_ch06.fm Page 186 Tuesday, September 28, 2004 2:46 PM

particular, the calling process cannot be created with its own console (using the CREATE_NEW_CONSOLE or DETACHED_PROCESS flag).

```
BOOL GenerateConsoleCtrlEvent (
   DWORD dwCtrlEvent,
   DWORD dwProcessGroup)
```

The first parameter, then, must be one of either CTRL_C_EVENT or CTRL_BREAK_EVENT. The second parameter identifies the process group.

Example: Simple Job Management

UNIX shells provide commands to execute processes in the background and to obtain their current status. This section develops a simple "job shell" with a similar set of commands. The commands are as follows.

- jobbg uses the remaining part of the command line as the command line for a new process, or *job*, but the jobbg command returns immediately rather than waiting for the new process to complete. The new process is optionally given its own console, or is *detached*, so that it has no console at all. This approach is similar to running a UNIX command with the & option at the end.
- jobs lists the current active jobs, giving the job numbers and process IDs. This is similar to the UNIX command of the same name.
- kill terminates a job. This implementation uses the TerminateProcess function, which, as previously stated, does not provide a clean shutdown. There is also an option to send a console control signal.

It is straightforward to create additional commands for suspending existing jobs or moving them to the foreground.

Because the shell, which maintains the job list, may terminate, the shell employs a user-specific shared file to contain the process IDs, the command, and related information. In this way, the shell can restart and the job list will still be intact. Furthermore, several shells can run concurrently. An exercise places this information in the registry, rather than in a temporary file.

Concurrency issues will arise. Several processes, running from separate command prompts, might perform job control simultaneously. The job management

Hart_ch06.fm Page 187 Tuesday, September 28, 2004 2:46 PM

functions use file locking (Chapter 3) on the job list file so that a user can invoke job management from separate shells or processes.

The full program on the book's Web site has a number of additional features, not shown in the listings, such as the ability to take command input from a file. `JobShell` will be the basis for a more general "service shell" in Chapter 13 (Program 13–3). Windows NT services are background processes, usually servers, that can be controlled with start, stop, pause, and other commands.

Creating a Background Job

Program 6–3 is the job shell that prompts the user for one of three commands and then carries out the command. This program uses a collection of job management functions, which are shown in Programs 6–4, 6–5, and 6–6.

Program 6–3 `JobShell`: Create, List, and Kill Background Jobs

```
/* Chapter 6. */
/* JobShell.c -- job management commands:
   jobbg -- Run a job in the background.
   jobs -- List all background jobs.
   kill -- Terminate a specified job of job family.
         There is an option to generate a console control signal. */

#include "EvryThng.h"
#include "JobMgt.h"

int _tmain (int argc, LPTSTR argv [])
{
   BOOL Exit = FALSE;
   TCHAR Command [MAX_COMMAND_LINE + 10], *pc;
   DWORD i, LocArgc; /* Local argc. */
   TCHAR argstr [MAX_ARG] [MAX_COMMAND_LINE];
   LPTSTR pArgs [MAX_ARG];

   for (i = 0; i < MAX_ARG; i++) pArgs [i] = argstr [i];
   /* Prompt user, read command, and execute it. */
   _tprintf (_T ("Windows Job Management\n"));
   while (!Exit) {
      _tprintf (_T ("%s"), _T ("JM$"));
      _fgetts (Command, MAX_COMMAND_LINE, stdin);
      pc = strchr (Command, '\n');
      *pc = '\0';
      /* Parse the input to obtain command line for new job. */
      GetArgs (Command, &LocArgc, pArgs); /* See Appendix A. */
      CharLower (argstr [0]);
```

Hart_ch06.fm Page 188 Tuesday, September 28, 2004 2:46 PM


```
      if (_tcscmp (argstr [0], _T ("jobbg")) == 0) {
         Jobbg (LocArgc, pArgs, Command);
      }
      else if (_tcscmp (argstr [0], _T ("jobs")) == 0) {
         Jobs (LocArgc, pArgs, Command);
      }
      else if (_tcscmp (argstr [0], _T ("kill")) == 0) {
         Kill (LocArgc, pArgs, Command);
      }
      else if (_tcscmp (argstr [0], _T ("quit")) == 0) {
         Exit = TRUE;
      }
      else _tprintf (_T ("Illegal command. Try again\n"));
   }
   return 0;
}

/* jobbg [options] command-line [Options are mutually exclusive]
      -c: Give the new process a console.
      -d: The new process is detached, with no console.
      If neither is set, the process shares console with jobbg. */
int Jobbg (int argc, LPTSTR argv [], LPTSTR Command)
{
   DWORD fCreate;
   LONG JobNo;
   BOOL Flags [2];
   STARTUPINFO StartUp;
   PROCESS_INFORMATION ProcessInfo;
   LPTSTR targv = SkipArg (Command);

   GetStartupInfo (&StartUp);
   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 job/thread suspended. Resume once job entered. */
   CreateProcess (NULL, targv, NULL, NULL, TRUE,
         fCreate | CREATE_SUSPENDED | CREATE_NEW_PROCESS_GROUP,
         NULL, NULL, &StartUp, &ProcessInfo);
      /* Create a job number and enter the process ID and handle
         into the job "data base." */

   JobNo = GetJobNumber (&ProcessInfo, targv); /* See "JobMgt.h" */
   if (JobNo >= 0)
      ResumeThread (ProcessInfo.hThread);
   else {
      TerminateProcess (ProcessInfo.hProcess, 3);
```

Hart_ch06.fm Page 189 Tuesday, September 28, 2004 2:46 PM


```
      CloseHandle (ProcessInfo.hProcess);
      ReportError (_T ("Error: No room in job list."), 0, FALSE);
      return 5;
   }
   CloseHandle (ProcessInfo.hThread);
   CloseHandle (ProcessInfo.hProcess);
   _tprintf (_T (" [%d] %d\n"), JobNo, ProcessInfo.dwProcessId);
   return 0;
}

/* jobs: List all running or stopped jobs. */
int Jobs (int argc, LPTSTR argv [], LPTSTR Command)
{
   if (!DisplayJobs ()) return 1; /* See job mgmt functions. */
   return 0;
}

/* kill [options] JobNumber
   -b Generate a Ctrl-Break
   -c Generate a Ctrl-C
      Otherwise, terminate the process. */
int Kill (int argc, LPTSTR argv [], LPTSTR Command)
{
   DWORD ProcessId, JobNumber, iJobNo;
   HANDLE hProcess;
   BOOL CntrlC, CntrlB, Killed;

   iJobNo =
      Options (argc, argv, _T ("bc"), &CntrlB, &CntrlC, NULL);

   /* Find the process ID associated with this job. */
   JobNumber = _ttoi (argv [iJobNo]);
   ProcessId = FindProcessId (JobNumber); /* See job mgmt. */
   hProcess = OpenProcess (PROCESS_ALL_ACCESS, FALSE, ProcessId);
   if (hProcess == NULL) { /* Process ID may not be in use. */
      ReportError (_T ("Process already terminated.\n"), 0, FALSE);
      return 2;
   }
   if (CntrlB)
      GenerateConsoleCtrlEvent (CTRL_BREAK_EVENT, ProcessId);
   else if (CntrlC)
      GenerateConsoleCtrlEvent (CTRL_C_EVENT, ProcessId);
   else
      TerminateProcess (hProcess, JM_EXIT_CODE);
   WaitForSingleObject (hProcess, 5000);
   CloseHandle (hProcess);
   _tprintf (_T ("Job [%d] terminated or timed out\n"), JobNumber);
   return 0;
}
```

Hart_ch06.fm Page 190 Tuesday, September 28, 2004 2:46 PM

Notice how the `jobbg` command creates the process in the suspended state and then calls the job management function, `GetJobNumber` (Program 6–4), to get a new job number and to register the job and its associated process. If the job cannot be registered for any reason, the job's process is terminated immediately. Normally, the job number is generated correctly, and the primary thread is resumed and allowed to run.

Getting a Job Number

The next three programs show three individual job management functions. These functions are all included in a single source file, `JobMgt.c`.

The first, Program 6–4, shows the `GetJobNumber` function. Notice the use of file locking with a completion handler to unlock the file. This technique protects against exceptions and inadvertent transfers around the unlock call. Such a transfer might be inserted accidentally during code maintenance even if the original program is correct. Also notice how the record past the end of the file is locked in the event that the file needs to be expanded with a new record.

Program 6–4 `JobMgt`: Creating New Job Information

```
/* Job management utility function. */

#include "EvryThng.h"
#include "JobMgt.h" /* Listed in Appendix A. */
void GetJobMgtFileName (LPTSTR);
LONG GetJobNumber (PROCESS_INFORMATION *pProcessInfo,
       LPCTSTR Command)

/* Create a job number for the new process, and enter
   the new process information into the job database. */
{
   HANDLE hJobData, hProcess;
   JM_JOB JobRecord;
   DWORD JobNumber = 0, nXfer, ExitCode, FsLow, FsHigh;
   TCHAR JobMgtFileName [MAX_PATH];
   OVERLAPPED RegionStart;

   if (!GetJobMgtFileName (JobMgtFileName)) return -1;
                /* Produces "\tmp\UserName.JobMgt" */
   hJobData = CreateFile (JobMgtFileName,
          GENERIC_READ | GENERIC_WRITE,
          FILE_SHARE_READ | FILE_SHARE_WRITE,
          NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
   if (hJobData == INVALID_HANDLE_VALUE) return -1;
```

```
    /* Lock the entire file plus one possible new
       record for exclusive access. */

    RegionStart.Offset = 0;
    RegionStart.OffsetHigh = 0;
    RegionStart.hEvent = (HANDLE)0;
    FsLow = GetFileSize (hJobData, &FsHigh);
    LockFileEx (hJobData, LOCKFILE_EXCLUSIVE_LOCK,
          0, FsLow + SJM_JOB, 0, &RegionStart);

    __try {
       /* Read records to find empty slot. */
       while (ReadFile (hJobData, &JobRecord, SJM_JOB, &nXfer, NULL)
             && (nXfer > 0)) {
          if (JobRecord.ProcessId == 0) break;
          hProcess = OpenProcess(PROCESS_ALL_ACCESS,
                FALSE, JobRecord.ProcessId);
          if (hProcess == NULL) break;
          if (GetExitCodeProcess (hProcess, &ExitCode)
                && (ExitCode != STILL_ACTIVE)) break;
          JobNumber++;
       }

       /* Either an empty slot has been found, or we are at end
          of file and need to create a new one. */

       if (nXfer != 0) /* Not at end of file. Back up. */
          SetFilePointer (hJobData, -(LONG)SJM_JOB,
                NULL, FILE_CURRENT);
       JobRecord.ProcessId = pProcessInfo->dwProcessId;
       _tcsnccpy (JobRecord.CommandLine, Command, MAX_PATH);
       WriteFile (hJobData, &JobRecord, SJM_JOB, &nXfer, NULL);
    } /* End try. */

    __finally {
       UnlockFileEx (hJobData, 0, FsLow + SJM_JOB, 0,
             &RegionStart);
       CloseHandle (hJobData);
    }
    return JobNumber + 1;
}
```

Listing Background Jobs

Program 6–5 shows the `DisplayJobs` job management function.

Program 6–5 `JobMgt`: Displaying Active Jobs

```
BOOL DisplayJobs (void)

/* Scan the job database file, reporting job status. */
{
   HANDLE hJobData, hProcess;
   JM_JOB JobRecord;
   DWORD JobNumber = 0, nXfer, ExitCode, FsLow, FsHigh;
   TCHAR JobMgtFileName [MAX_PATH];
   OVERLAPPED RegionStart;

   GetJobMgtFileName (JobMgtFileName);
   hJobData = CreateFile (JobMgtFileName,
         GENERIC_READ | GENERIC_WRITE,
         FILE_SHARE_READ | FILE_SHARE_WRITE,
         NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

   RegionStart.Offset = 0;
   RegionStart.OffsetHigh = 0;
   RegionStart.hEvent = (HANDLE)0;
   FsLow = GetFileSize (hJobData, &FsHigh);
   LockFileEx (hJobData, LOCKFILE_EXCLUSIVE_LOCK,
         0, FsLow, FsHigh, &RegionStart);

   __try {
   while (ReadFile (hJobData, &JobRecord, SJM_JOB, &nXfer, NULL)
         && (nXfer > 0)){
      JobNumber++;
      if (JobRecord.ProcessId == 0)
         continue;
      hProcess = OpenProcess (PROCESS_ALL_ACCESS, FALSE,
            JobRecord.ProcessId);
      if (hProcess != NULL)
         GetExitCodeProcess (hProcess, &ExitCode);
      _tprintf (_T (" [%d] "), JobNumber);
      if (hProcess == NULL)
         _tprintf (_T (" Done"));
      else if (ExitCode != STILL_ACTIVE)
         _tprintf (_T ("+ Done"));
      else _tprintf (_T (" "));
      _tprintf (_T (" %s\n"), JobRecord.CommandLine);

      /* Remove processes that are no longer in system. */

      if (hProcess == NULL) { /* Back up one record. */
         SetFilePointer (hJobData, -(LONG)nXfer,
               NULL, FILE_CURRENT);
         JobRecord.ProcessId = 0;
```

```
            WriteFile (hJobData, &JobRecord, SJM_JOB, &nXfer, NULL);
         }
      } /* End of while. */
   } /* End of __try. */

   __finally {
      UnlockFileEx (hJobData, 0, FsLow, FsHigh, &RegionStart);
      CloseHandle (hJobData);
   }

   return TRUE;
}
```

Finding a Job in the Job List File

Program 6–6 shows the final job management function, `FindProcessId`, which obtains the process ID of a specified job number. The process ID, in turn, can be used by the calling program to obtain a handle and other process status information.

Program 6–6 `JobMgt`: Getting the Process ID from a Job Number

```
DWORD FindProcessId (DWORD JobNumber)

/* Obtain the process ID of the specified job number. */
{
   HANDLE hJobData;
   JM_JOB JobRecord;
   DWORD nXfer;
   TCHAR JobMgtFileName [MAX_PATH];
   OVERLAPPED RegionStart;

   /* Open the job management file. */
   GetJobMgtFileName (JobMgtFileName);

   hJobData = CreateFile (JobMgtFileName, GENERIC_READ,
         FILE_SHARE_READ | FILE_SHARE_WRITE,
         NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
   if (hJobData == INVALID_HANDLE_VALUE) return 0;

   /* Position to the entry for the specified job number.
    * The full program assures that JobNumber is in range. */

   SetFilePointer (hJobData, SJM_JOB * (JobNumber - 1),
         NULL, FILE_BEGIN);

   /* Lock and read the record. */
```

Hart_ch06.fm Page 194 Tuesday, September 28, 2004 2:46 PM


```
   RegionStart.Offset = SJM_JOB * (JobNumber - 1);
   RegionStart.OffsetHigh = 0; /* Assume a "short" file. */
   RegionStart.hEvent = (HANDLE)0;
   LockFileEx (hJobData, 0, 0, SJM_JOB, 0, &RegionStart);
   ReadFile (hJobData, &JobRecord, SJM_JOB, &nXfer, NULL);
   UnlockFileEx (hJobData, 0, SJM_JOB, 0, &RegionStart);
   CloseHandle (hJobData);
   return JobRecord.ProcessId;
}
```

Job Objects

Processes can be collected together into *job objects* where the processes can be controlled as a group, resource limits can be specified for all the job object member processes, and accounting information can be maintained. Job objects were introduced with Windows 2000 and are supported in all NT5 systems.

The first step is to create an empty job object with `CreateJobObject`, which takes two arguments, a name and security attributes, and returns a job object handle. There is also an `OpenJobObject` function to use with a named object. `CloseHandle` destroys the job object.

`AssignProcessToJobObject` simply adds a process specified by a process handle to a job object; there are just two parameters. A process cannot be a member of more than one job, so `AssignProcessToJobObject` fails if the process associated with the handle is already a member of some job. A process that is added to a job inherits all the limits associated with the job and adds its accounting information to the job, such as the processor time used.

By default, a new child process created with `CreateProcess` will also belong to the job unless the `CREATE_BREAKAWAY_FROM_JOB` flag is specified in the `dwCreationFlags` argument to `CreateProcess`. In the default case, `Assign-ProcessToJobObject` will fail if you attempt to assign the child process to a job.

Finally, you can specify control limits on the processes in a job using `SetInformationJobObject`.

```
BOOL SetInformationJobObject (
   HANDLE hJob,
   JOBOBJECTINFOCLASS JobObjectInformationClass,
   LPVOID lpJobObjectInformation,
   DWORD cbJobObjectInformationLength)
```

Hart_ch06.fm Page 195 Tuesday, September 28, 2004 2:46 PM

- hJob is a handle for an existing job object.
- JobObjectInformationClass specifies the information class for the limits you wish to set. There are five values; JobObjectBasicLimitInformation is one value and is used to specify information such as the total and per-process time limits, working set size limits,[2] limits on the number of active processes, priority, and processor affinity (the processors of an SMP system that can be used by threads in the job processes).
- lpJobObjectInformation points to the actual information required by the preceding parameter. There is a different structure for each class.
- JOBOBJECT_BASIC_ACCOUNTING_INFORMATION allows you to get the total time (user, kernel, and elapsed) of the processes in a job.
- The last parameter is the length of the preceding structure.

QueryJobInformationObject obtains the current limits. Other information classes impose limits on the user interface, I/O completion ports (see Chapter 14), security, and job termination.

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.

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) will be reimplemented with threads in the first example program of Chapter 7.

Chapter 8 will then introduce synchronization, which can be used to coordinate operation between threads in the same or different processes.

[2] The working set is the set of virtual address space pages that the OS determines must be loaded in memory before any thread in the process is ready to run. This subject is covered in most OS texts.

Hart_ch06.fm Page 196 Tuesday, September 28, 2004 2:46 PM

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) 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 on the book's Web site illustrate this technique.

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) when you have an SMP system or when the files are on separate or network drives. Appendix C presents some partial results.

6–6. Can you find a way, perhaps using job objects, 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.

Hart_ch06.fm Page 197 Tuesday, September 28, 2004 2:46 PM

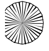
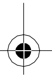
6–8. The job management functions have a defect that is difficult to fix. Suppose that a job is killed and the executive reuses its process ID before the process ID is removed from the job management file. There could be an `OpenProcess` on the process ID that now refers to a totally different process. 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. Extend `JobShell` so that the processes are associated with a job object. Impose time and other limits on the jobs, allowing the user to enter some of these limits.

6–11. Enhance `JobShell` so that the `jobs` command will include a count of the number of handles that each job is using. *Hint*: Use `GetProcessHandle-Count`, which requires NT 5.1.

6–12. Build project `Version` (on the Web site), which uses `version.c`. Run the program on as many different Windows versions as you can access, including Windows 9x and NT 4.0 systems if possible. What are the major and minor version numbers for those systems, and what other information is available?



Hart_ch06.fm Page 198 Tuesday, September 28, 2004 2:46 PM