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 execute application code, create new threads, create new independent processes, and manage communication and synchronization among the threads.

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 improve application performance by exploiting multiple CPU processors.

This chapter explains the basics of process management and also introduces the basic synchronization operations and wait functions that will be important 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; see the next chapter for a threads introduction. 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 long supported multiprocessor systems, so threads can be allocated to separate processors within a computer.

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. Note that shared memory-mapped files share physical memory, but the sharing processes will probably 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)—An arraylike collection 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 Russinovich, Solomon, and Ionescu, *Windows Internals: Including Windows Server 2008 and Windows Vista*.

A UNIX process is comparable to a Windows process.

Threads, in the form of POSIX Pthreads, are now nearly universally available and used in UNIX and Linux. Pthreads provides features similar to Windows threads, although Windows provides a broader collection of functions.

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 long obsolete. There is an open source Pthreads library for Windows.

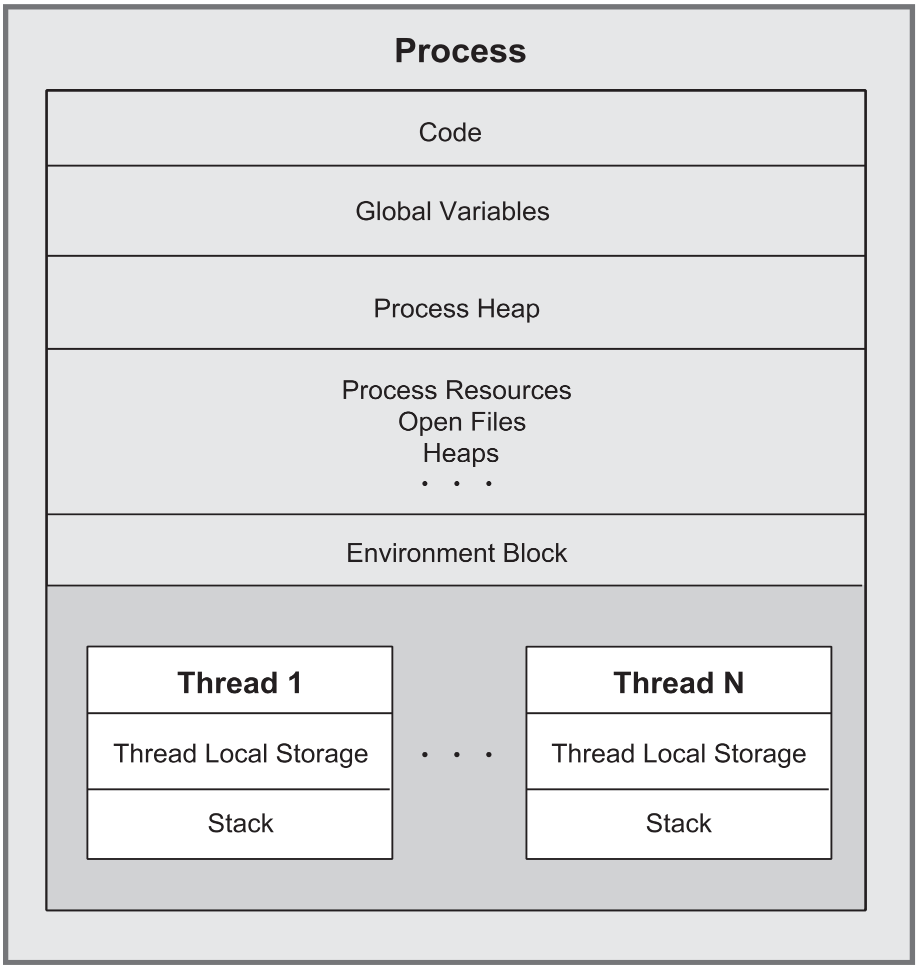

Figure 6–1 A Process and Its Threads

Process Creation

The fundamental Windows process management function is `CreateProcess`, which creates a process with a single thread. 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 Windows does not actually maintain these relationships. It is simply convenient to refer to the process that creates a child process as the parent.

`CreateProcess` has 10 parameters to support its flexibility and power. Initially, it is simplest to use default values. Just as with `CreateFile`, it is appropriate to explain all the `CreateProcess` parameters. Related functions are then 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 single *primary* thread (which might create additional threads). 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`.

```
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 the program calls `ResumeThread`.
- `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.
- `CREATE_UNICODE_ENVIRONMENT` should be set if `UNICODE` is defined.
- `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 limited similarities to UNIX process groups and are described later in the "Generating Console Control Events" section.

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` is complex and specifies the main window appearance and standard device handles for the new process. We'll use two principal techniques to set the start up information. Programs 6–1, 6–2, 6–3, and others show the proper sequence of operations, which can be confusing.

- Use the parent's information, which is obtained from `GetStartupInfo`.
- First, clear the associated `STARTUPINFO` structure before calling `CreateProcess`, and then specify the standard input, output, and error handles by setting the `STARTUPINFO` standard handler fields (`hStdInput`, `hStdOutput`, and `hStdError`). 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.

`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, although the ID is invalid when the process or thread is destroyed and the ID may be reused. On the other hand, a given process may have several handles, each having distinct attributes, such as 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/Linux 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 program because there are numerous problems with creating an exact replica of a multithreaded program with exact copies of all threads and synchronization objects, especially on a multiprocessor computer. Therefore, `fork`, by itself, is not really appropriate in any multithreaded application.

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 UNIX-style 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. 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. This is not a command line, and it should not be enclosed with quotation marks.
- 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. A program could modify its arguments, although it is advisable to make any changes in a copy of the argument string.

It is not necessary to build the new process with the same UNICODE definition as that of the parent process. All combinations are possible. Using _tmain as described 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, and Program 6-1 is the first of several examples. 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, 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, which is not the 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, set `nLength` to `sizeof (SECURITY_ATTRIBUTES)`.

The following code segment shows how to create an inheritable file or other handle. 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. You can reuse sa. */
```

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 in the next example (Program 6–1) and in several additional examples throughout the book. This is generally the preferred

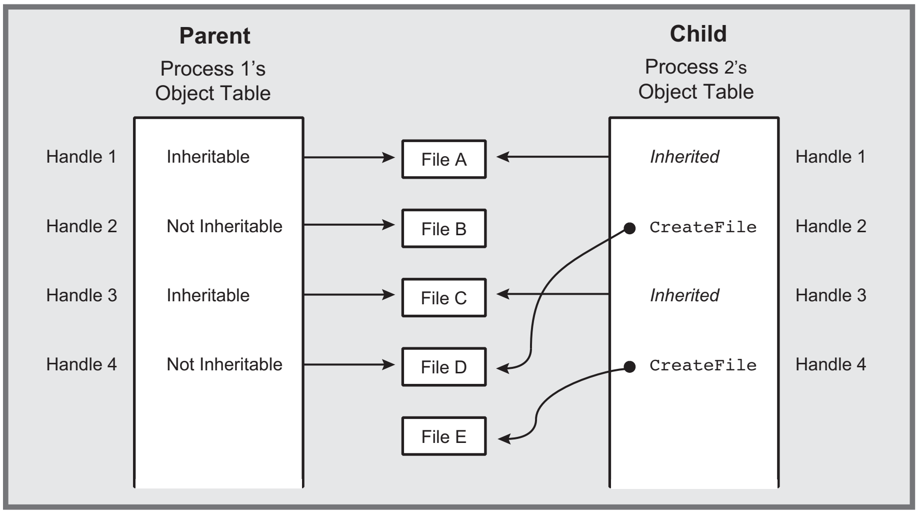

Figure 6–2 Process Handle Tables

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 in the *Examples* file.

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 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. Also, 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. Finally, while not shown in the figure, a process can have multiple handles to refer to the same object.

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 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.

bInheritHandle specifies whether the new process handle is inheritable. dwProcessId is the identifier of the process to be opened, and the returned process handle will reference this process.

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 with a non-null hModule value 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 GetCurrentProcess—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 receives 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, but generally not advisable, 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 MSDN).

`dwOptions` is any combination of two flags.

- `DUPLICATE_CLOSE_SOURCE` causes the source handle to be closed and can be specified if the source handle is no longer useful. This option also assures that the reference count to the underlying file (or other object) remains constant.
- `DUPLICATE_SAME_ACCESS` uses the access rights of the duplicated handle, and `dwDesiredAccess` is 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. An object cannot be destroyed (e.g., deleting a file) 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.

Next, we learn how to determine whether a process has terminated.

Exiting and Terminating a Process

After a process has finished its work, the process (actually, 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.