Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact:
International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data
Hart, Johnson M.
Windows system programming / Johnson M. Hart.
p. cm.
Includes bibliographical references and index.

QA76.76.A65H373 2010
005.3—dc22

2009046939

Copyright © 2010 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, February 2010
This book describes application development using the Microsoft Windows Application Programming Interface (API), concentrating on the core system services, including the file system, process and thread management, interprocess communication, network programming, and synchronization. The examples concentrate on realistic scenarios, and in many cases they’re based on real applications I’ve encountered in practice.

The Win32/Win64 API, or the Windows API, is supported by Microsoft’s family of 32-bit and 64-bit operating systems; versions currently supported and widely used include Windows 7, XP, Vista, Server 2003, Server 2008, and CE. Older Windows family members include Windows 2000, NT, Me, 98, and 95; these systems are obsolete, but many topics in this book still apply to these older systems.

The Windows API is an important factor for application development, frequently replacing the POSIX API (supported by UNIX and Linux) as the preferred API for applications targeted at desktop, server, and embedded systems now and for the indefinite future. Many programmers, regardless of experience level, need to learn the Windows API quickly, and this book is designed for them to do so.

Objectives and Approach

The objectives I’ve set for the book are to explain what Windows is, show how to use it in realistic situations, and do so as quickly as possible without burdening you with unnecessary detail. This book is not a reference guide, but it explains the central features of the most important functions and shows how to use them together in practical programming situations. Equipped with this knowledge, you will be able to use the comprehensive Microsoft reference documentation to explore details, advanced options, and the more obscure functions as requirements or interests dictate. I have found the Windows API easy to learn using this approach and have greatly enjoyed developing Windows programs, despite occasional frustration. This enthusiasm will show through at times, as it should. This does not mean that I feel that Windows is necessarily better than other operating system (OS) APIs, but it certainly has many attractive features and improves significantly with each major new release.

Many Windows books spend a great deal of time explaining how processes, virtual memory, interprocess communication, and preemptive scheduling work without showing how to use them in realistic situations. A programmer experienced in UNIX, Linux, IBM MVS, or another OS will be familiar with these concepts and will be
impatient to find out how they are implemented in Windows. Most Windows books also spend a great deal of space on the important topic of user interface programming. This book intentionally avoids the user interface, beyond discussing simple character-based console I/O, in the interest of concentrating on the important core features.

I’ve taken the point of view that Windows is just an OS API, providing a well-understood set of features. Many programmers, regardless of experience level, need to learn Windows quickly. Furthermore, understanding the Windows API is invaluable background for programmers developing for the Microsoft .NET Framework.

The Windows systems, when compared with other systems, have good, bad, and average features and quality. Recent releases (Windows 7, Vista, Server 2008) provide new features, such as condition variables, that both improve performance and simplify programming. The purpose of this book is to show how to use those features efficiently and in realistic situations to develop practical, high-quality, and high-performance applications.

**Audience**

I’ve enjoyed receiving valuable input, ideas, and feedback from readers in all areas of the target audience, which includes:

- Anyone who wants to learn about Windows application development quickly, regardless of previous experience.

- Programmers and software engineers who want to port existing Linux or UNIX (the POSIX API) applications to Windows. Frequently, the source code must continue to support POSIX; that is, source code portability is a requirement. The book frequently compares Windows, POSIX, and standard C library functions and programming models.

- Developers starting new projects who are not constrained by the need to port existing code. Many aspects of program design and implementation are covered, and Windows functions are used to create useful applications and to solve common programming problems.

- Application architects and designers who need to understand Windows capabilities and principles.

- Programmers using COM and the .NET Framework, who will find much of the information here helpful in understanding topics such as dynamic link libraries (DLLs), thread usage and models, interfaces, and synchronization.

- Computer science students at the upper-class undergraduate or beginning graduate level in courses covering systems programming or application devel-
opment. This book will also be useful to those who are learning multithreaded programming or need to build networked applications. This book would be a useful complementary text to a classic book such as *Advanced Programming in the UNIX Environment* (by W. Richard Stevens and Stephen A. Rago) so that students could compare Windows and UNIX. Students in OS courses will find this book to be a useful supplement because it illustrates how a commercially important OS provides essential functionality.

The only other assumption, implicit in all the others, is a knowledge of C or C++ programming.

**Windows Progress Since the Previous Editions**

The first edition of this book, titled *Win32 System Programming*, was published in 1997 and was updated with the second edition (2000) and the third edition (2004). Much has changed, and much has stayed the same since these previous editions, and Windows has been part of ongoing, rapid progress in computing technology. The outstanding factors to me that explain the fourth edition changes are the following:

- The Windows API is extremely stable. Programs written in 1997 continue to run on the latest Windows releases, and Windows skills learned now or even years ago will be valuable for decades to come.

- Nonetheless, the API has expanded, and there are new features and functions that are useful and sometimes mandatory. Three examples of many that come to mind and have been important in my work are (1) the ability to work easily with large files and large, 64-bit address spaces, (2) thread pools, and (3) the new condition variables that efficiently solve an important synchronization problem.

- Windows scales from phones to handheld and embedded devices to laptops and desktop systems and up to the largest servers.

- Windows has grown and scaled from the modest resources required in 1997 (16MB of RAM and 250MB of free disk space!) to operate efficiently on systems orders of magnitude larger and faster but often cheaper.

- 64-bit systems, multicore processors, and large file systems are common, and our application programs must be able to exploit these systems. Frequently, the programs must also continue to run on 32-bit systems.
Changes in the Fourth Edition

This fourth edition presents extensive new material along with updates and reorganization to keep up with recent progress and:

- Demonstrates example program operation and performance with screenshots.
- Describes and illustrates techniques to assure that relevant applications scale to run on 64-bit systems and can use large files. Enhancements throughout the book address this issue.
- Eliminates discussion of Windows 95, 98, and Me (the “Windows 9x” family), as well as NT and other obsolete systems. Program examples freely exploit features supported only in current Windows versions.
- Provides enhanced coverage of threads, synchronization, and parallelism, including performance, scalability, and reliability considerations.
- Emphasizes the important role and new features of Windows servers running high-performance, scalable, multithreaded applications.
- Studies performance implications of different program designs, especially in file access and multithreaded applications with synchronization and parallel programs running on multicore systems.
- Addresses source code portability to assure operation on Windows, Linux, and UNIX systems. Appendix B is enhanced from the previous versions to help those who need to build code, usually for server applications, that will run on multiple target platforms.
- Incorporates large quantities of excellent reader and reviewer feedback to fix defects, improve explanations, improve the organization, and address numerous details, large and small.

Organization

Chapters are organized topically so that the features required in even a single-threaded application are covered first, followed by process and thread management features, and finally network programming in a multithreaded environment. This organization allows you to advance logically from file systems to memory management and file mapping, and then to processes, threads, and synchronization, followed by interprocess and network communication and security. This organization also allows the examples to evolve in a natural way, much as a developer might cre-
ate a simple prototype and then add additional capability. The advanced features, such as asynchronous I/O and security, appear last.

Within each chapter, after introducing the functionality area, such as process management or memory-mapped files, we discuss important Windows functions and their relationships in detail. Illustrative examples follow. Within the text, only essential program segments are listed; complete projects, programs, include files, utility functions, and documentation are on the book’s Web site (www.jmhartsoftware.com). Throughout, we identify those features supported only by current Windows versions. Each chapter suggests related additional reading and gives some exercises. Many exercises address interesting and important issues that did not fit within the normal text, and others suggest ways for you to explore advanced or specialized topics.

Chapter 1 is a high-level introduction to the Windows OS family and Windows. A simple example program shows the basic elements of Windows programming style and lays the foundation for more advanced Windows features. Win64 compatibility issues are introduced in Chapter 1 and are included throughout the book.

Chapters 2 and 3 deal with file systems, console I/O, file locking, and directory management. Unicode, the extended character set used by Windows, is also introduced in Chapter 2. Examples include sequential and direct file processing, directory traversal, and management. Chapter 3 ends with a discussion of registry management programming, which is analogous in many ways to file and directory management.

Chapter 4 introduces Windows exception handling, including Structured Exception Handling (SEH), which is used extensively throughout the book. By introducing it early, we can use SEH throughout and simplify some programming tasks and improve quality. Vectored exception handling is also described.

Chapter 5 treats Windows memory management and shows how to use memory-mapped files both to simplify programming and to improve performance. This chapter also covers DLLs. An example compares memory-mapped file access performance and scalability to normal file I/O on both 32-bit and 64-bit systems.

Chapter 6 introduces Windows processes, process management, and simple process synchronization. Chapter 7 then describes thread management in similar terms and introduces parallelism to exploit multiprocessor systems. Examples in each chapter show the many benefits of using threads and processes, including program simplicity and performance.

Chapters 8, 9, and 10 give an extended, in-depth treatment of Windows thread synchronization, thread pools, and performance considerations. These topics are complex, and the chapters use extended examples and well-understood models to help you obtain the programming and performance benefits of threads while avoiding the numerous pitfalls. New material covers new functionality along with
performance and scalability issues, which are important when building server-based applications, including those that will run on multiprocessor systems.

Chapters 11 and 12 are concerned with interprocess and interthread communication and networking. Chapter 11 concentrates on the features that are properly part of Windows—namely, anonymous pipes, named pipes, and mailslots. Chapter 12 discusses Windows Sockets, which allow interoperability with non-Windows systems using industry-standard protocols, primarily TCP/IP. Windows Sockets, while not strictly part of the Windows API, provide for network and Internet communication and interoperability, and the subject matter is consistent with the rest of the book. A multithreaded client/server system illustrates how to use interprocess communication along with threads.

Chapter 13 describes how Windows allows server applications, such as the ones created in Chapters 11 and 12, to be converted to Windows Services that can be managed as background servers. Some small programming changes will turn the servers into services.

Chapter 14 shows how to perform asynchronous I/O using overlapped I/O with events and completion routines. You can achieve much the same thing with threads, so examples compare the different solutions for simplicity and performance. In particular, as of Windows Vista, completion routines provide very good performance. The closely related I/O completion ports are useful for some scalable multithreaded servers, so this feature is illustrated with the server programs from earlier chapters. The final topic is waitable timers, which require concepts introduced earlier in the chapter.

Chapter 15 briefly explains Windows object security, showing, in an example, how to emulate UNIX-style file permissions. Additional examples shows how to secure processes, threads, and named pipes. Security upgrades can then be applied to the earlier examples as appropriate.

There are three appendixes. Appendix A describes the example code that you can download from the book’s Web site (www.jmhartsoftware.com). Appendix B shows how to create source code that can also be built to run on POSIX (Linux and UNIX) systems; this requirement is common with server applications and organizations that need to support systems other than just Windows. Appendix C compares the performance of alternative implementations of some of the text examples so that you can gauge the trade-offs between Windows features, both basic and advanced.

**UNIX and C Library Notes and Tables**

Within the text at appropriate points, we contrast Windows style and functionality with the comparable POSIX (UNIX and Linux) and ANSI Standard C library features. Appendix B reviews source code portability and also contains a table list-
ing these comparable functions. This information is included for two principal rea-
sons:

- Many people are familiar with UNIX or Linux and are interested in the com-
  parisons between the two systems. If you don’t have a UNIX/Linux back-
  ground, feel free to skip those paragraphs in the text, which are indented and
  set in a smaller font.

- Source code portability is important to many developers and organizations.

Examples

The examples are designed to:

- Illustrate common, representative, and useful applications of the Windows
  functions.

- Correspond to real programming situations encountered in program develop-
  ment, consulting, and training. Some of my clients and course participants have
  used the code examples as the bases for their own systems. During consulting
  activities, I frequently encounter code that is similar to that used in the
  examples, and on several occasions I have seen code taken directly or modified
  from previous editions. (Feel free to do so yourself; an acknowledgment in your
  documentation would be greatly appreciated.) Frequently, this code occurs as
  part of COM, .NET, or C++ objects. The examples, subject to time and space con-
  straints, are “real-world” examples and solve “real-world” problems.

- Emphasize how the functions actually behave and interact, which is not
  always as you might first expect after reading the documentation. Throughout
  this book, the text and the examples concentrate on interactions between
  functions rather than on the functions themselves.

- Grow and expand, both adding new capability to a previous solution in a
  natural manner and exploring alternative implementation techniques.

- Implement UNIX/Linux commands, such as ls, touch, chmod, and sort,
  showing the Windows functions in a familiar context while creating a useful
  set of utilities.¹ Different implementations of the same command also give us

¹ Several commercial and open source products provide complete sets of UNIX/Linux utilities; there is
no intent to supplement them. These examples, although useful, are primarily intended to illustrate
Windows usage. Anyone unfamiliar with UNIX or Linux should not, however, have any difficulty un-
derstanding the programs or their functionality.
an easy way to compare performance benefits available with advanced Windows features. Appendix C contains the performance test results.

Examples in the early chapters are usually short, but the later chapters present longer examples when appropriate.

Exercises at the end of each chapter suggest alternative designs, subjects for investigation, and additional functionality that is important but beyond the book’s scope. Some exercises are easy, and a few are very challenging. Frequently, clearly labeled defective solutions are provided, because fixing the bugs is an excellent way to sharpen skills.

All examples have been debugged and tested under Windows 7, Vista, Server 2008, XP, and earlier systems. Testing included 32-bit and 64-bit versions. All programs were also tested on both single-processor and multiprocessor systems using as many as 16 processors. The client/server applications have been tested using multiple clients simultaneously interacting with a server. Nonetheless, there is no guarantee or assurance of program correctness, completeness, or fitness for any purpose. Undoubtedly, even the simplest examples contain defects or will fail under some conditions; such is the fate of nearly all software. I will, however, gratefully appreciate any messages regarding program defects—and, better still, fixes, and I’ll post this information on the book’s Web site so that everyone will benefit.

The Web Site

The book’s Web site (www.jmhartsoftware.com) contains a downloadable Examples file with complete code and projects for all the book’s examples, a number of exercise solutions, alternative implementations, instructions, and performance evaluation tests. This material will be updated periodically to include new material and corrections.

The Web site also contains book errata, along with additional examples, reader contributions, additional explanations, and much more. The site also contains PowerPoint slides that can be used for noncommercial instructional purposes. I’ve used these slides numerous times in professional training courses, and they are also suitable for college courses.

The material will be updated as required when defects are fixed and as new input is received. If you encounter any difficulties with the programs or any material in the book, check these locations first because there may already be a fix or explanation. If that does not answer your question, feel free to send e-mail to jmh_assoc@hotmail.com or jmhart62@gmail.com.
Acknowledgments

Numerous people have provided assistance, advice, and encouragement during the fourth edition’s preparation, and readers have provided many important ideas and corrections. The Web site acknowledges the significant contributions that have found their way into the fourth edition, and the first three editions acknowledge earlier valuable contributions. See the Web site for a complete list.

Three reviewers deserve the highest possible praise and thanks for their incisive comments, patience, excellent suggestions, and deep expertise. Chris Sells, Jason Beres, and especially Raymond Chen made contributions that improved the book immeasurably. To the best of my ability, I’ve revised the text to address their points and invaluable input.

Numerous friends and colleagues also deserve a note of special thanks; I’ve learned a lot from them over the years, and many of their ideas have found their way into the book in one way or another. They’ve also been generous in providing access to test systems. In particular, I’d like to thank my friends at Sierra Atlantic, Cilk Arts (now part of Intel), Vault USA, and Rimes Technologies.

Anne H. Smith, the compositor, used her skill, persistence, and patience to prepare this new edition for publication; the book simply would not have been possible without her assistance. Anne and her husband, Kerry, also have generously tested the sample programs on their equipment.

The staff at Addison-Wesley exhibited the professionalism and expertise that make an author’s work a pleasure. Joan Murray, the editor, and Karen Gettman, the editor-in-chief, worked with the project from the beginning making sure that no barriers got in the way and assuring that hardly any schedules slipped. Olivia Basegio, the editorial assistant, managed the process throughout, and John Fuller and Elizabeth Ryan from production made the production process seem almost simple. Anna Popick, the project editor, guided the final editing steps and schedule. Carol Lallier and Lori Newhouse, the copy editor and proofreader, made valuable contributions to the book’s readability and consistency.

Johnson (John) M. Hart
jmhart62@gmail.com
December, 2009
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.
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.

```c
BOOL CreateProcess ( 
    LPCTSTR lpApplicationName, 
    LPCTSTR lpCommandLine, 
    LPSECURITY_ATTRIBUTES lpSecurityProcess, 
    LPSECURITY_ATTRIBUTES lpSecurityThread, 
    BOOL bInheritHandles, 
    DWORD dwCreationFlags, 
    LPVOID lpEnvironment, 
    LPCTSTR lpCurrentDir, 
    LPTSTR 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.

lpSecurityProcess and lpSecurityThread 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:

```c
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.
**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.

```c
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
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:

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

Upon completion, lpTargetHandle 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 lpTargetHandle, 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.

```c
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 `D11Main` (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.
EXITING AND TERMINATING A PROCESS

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 treated 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 getpid because Windows has no process parents or UNIX-like 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`.

### Waiting for a Process to Terminate

The simplest, and most limited, method to synchronize 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 collection 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:

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

```c
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 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 collection of processes to terminate. If several handles are signaled, the returned value is the minimum of the signaled handle indices. 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

\[
\text{Name} = \text{Value}
\]
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:

- **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 simple 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; however, the project and executable (grep.exe) are in the Examples file.

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 temporary file handle, specified to be inheritable, is assigned to the
  hStdOutput field in 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) han-
  dles, so the program calls it 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, and Run 6–1 shows program execution and timing results.

Program 6–1  grepMP: Parallel Searching

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

#include "Everything.h"
int main (DWORD argc, LPTSTR argv[]) {
    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 iPrc, exitCode, dwCreationFlags = 0;
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. */

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 (iPrc = 0; iPrc < argc - 2; iPrc++) {
    _stprintf (commandLine, _T ("grep "$s" "$s""),
               argv[1], argv[iPrc + 2]);
    GetTempFileName (_T ("."), _T ("gtm"), 0,
                    procFile[iPrc].tempFile); /* For search results. */
    hTempFile = /* This handle is inheritable */
                    CreateFile (procFile[iPrc].tempFile,
                                    GENERIC_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,
                   dwCreationFlags, NULL, NULL, &startUpSearch, &processInfo);
    /* Close unwanted handles. */
    CloseHandle (hTempFile); CloseHandle (processInfo.hThread);
    hProc[iPrc] = processInfo.hProcess;
}

/* Processes are all running. Wait for them to complete. */
for (iPrc = 0; iPrc < argc - 2; iPrc += MAXIMUM_WAIT_OBJECTS)
    WaitForMultipleObjects (/* Allows a large # of processes */
                            min (MAXIMUM_WAIT_OBJECTS, argc - 2 - iPrc),
                            &hProc[iPrc], TRUE, INFINITE);
/* Result files sent to std output using "cat." */
for (iPrc = 0; iPrc < argc - 2; iPrc++) {
    if (GetExitCodeProcess(hProc[iPrc], &exitCode) && exitCode==0) {
        /* Pattern was detected -- List results. */
        if (argc > 3) _stprintf (_T("%s
"), argv[iPrc + 2]);
        _stprintf (commandLine, _T ("cat "$s""),
                    procFile[iPrc].tempFile);
        CreateProcess (NULL, commandLine, NULL, NULL, TRUE,
                       dwCreationFlags, NULL, NULL, &startUp, &processInfo);
        WaitForSingleObject (processInfo.hProcess, INFINITE);
```c
    closeHandle (processInfo.hProcess);
    closeHandle (processInfo.hThread);
}

    closeHandle (hProc[iProc]);
    DeleteFile (procFile[iProc].tempFile);
}

free (procFile);
free (hProc);
return 0;

```

Run 6-1  grepMP: Parallel Searching
Run 6–1 shows `grepMP` execution for large and small files, and the run contrasts sequential `grep` execution with parallel `grepMP` execution to perform the same task. The test computer has four processors; a single or dual processor computer will give different timing results. Notes after the run explain the test operation and results.

Run 6–1 uses files and obtains results as follows:

- The small file test searches two `Examples` files, `Presidents.txt` and `Monarchs.txt`, which contain names of U.S. presidents and English monarchs, along with their dates of birth, death, and term in office. The “i” at the right end of each line is a visual cue and has no other meaning. The same is true of the “x” at the end of the `randfile`-generated files.
- The large file test searches four `randfile`-generated files, each with 10 million 64-byte records. The search is for a specific record number (1234562), and each file has a different random key (the first 8 bytes).
- `grepMP` is more than four times faster than four sequential `grep` executions (Real Time is 15 seconds compared to 77 seconds), so the multiple processes gain even more performance than expected, despite the process creation overhead.
- `timep` is Program 6–2, the next example. Notice, however, that the `grepMP` system time is zero, as the time applies to `grepMP` itself, not the `grep` processes that it creates.

**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 a multiprocessor computer. As Run 6–1 shows, this can result in substantial performance improvement when performance is measured as elapsed time to execute the program, and no explicit program actions are required to get the performance improvement.

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) for a process or thread.

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

**Process Execution Times**

You can determine the amount of time that a process has consumed (elapsed, kernel, and user times) using the `GetProcessTimes` function.

```c
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.

Chapter 3’s `lsW` example showed how to convert and display file times, although the kernel and user times are elapsed times rather than calendar times. `GetThreadTimes` is similar and requires a thread handle for a parameter.

**Example: Process Execution Times**

The next example (Program 6–2) implements the familiar `time` (time print) utility that is similar to the UNIX `time` command (`time` is supported by the Windows command prompt, so a different name is appropriate). `time` prints elapsed (or real), user, and system times.

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. `SkipArg` is in the `Examples` file.
Program 6-2  \texttt{timep}: Process Times

/* Chapter 6. \texttt{timep}. */

#include "Everything.h"

int \_main (int argc, LPTSTR argv[])
{
    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;
    LPTSTR tarqv = SkipArg (GetCommandLine ());
    HANDLE hProc;

    GetStartupInfo (&startUp);
    GetSystemTime (&startTimeSys);

    /* Execute the command line; wait for process to complete. */
    CreateProcess (NULL, tarqv, NULL, NULL, TRUE,
        NORMAL_PRIORITY_CLASS, NULL, NULL, &startUp, &procInfo);
    hProc = procInfo.hProcess;
    WaitForSingleObject (hProc, INFINITE);

    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\n"),
        elTiSys.wHour, elTiSys.wMinute, elTiSys.wSecond,
        elTiSys.wMilliseconds);
    _tprintf (_T("User Time: %02d:%02d:%02d\n"),
        usTiSys.wHour, usTiSys.wMinute, usTiSys.wSecond,
        usTiSys.wMilliseconds);
    _tprintf (_T("Sys Time: %02d:%02d:%02d\n"),
        keTiSys.wHour, keTiSys.wMinute, keTiSys.wSecond,
        keTiSys.wMilliseconds);

    CloseHandle (procInfo.hThread); CloseHandle (procInfo.hProcess);
    CloseHandle (hProc);
    return 0;
}
Using the `timep` Command

`timep` was useful to compare different programming solutions, such as the various Caesar cipher (`cci`) and sorting utilities, including `cci` (Program 2–3) and `sortmm` (Program 5–5). Appendix C summarizes and briefly analyzes some additional results, and there are other examples throughout the book.

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 collection of processes. Run 6–1 and Appendix C show that, on a multiprocessor computer, 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. On the other hand, you cannot always count on such performance gains; for example, there might be resource contention or disk thrashing that could impact performance negatively.

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.

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

---

\(^1\) Chapter 10 shows an indirect way for one thread to cause an exception in another thread, and the same technique is applicable between threads in different processes.
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 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. Using a new console avoids console contention with `jobbg` and other jobs. 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 operations such as suspending and resuming existing jobs.

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. You could place this information in the registry rather than in a temporary file (see Exercise 6–9).

Concurrency issues will arise. Several processes, running from separate command prompts, might perform job control simultaneously. The job management functions use file locking (Chapter 3) on the job list file so that a user can invoke

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

Do not confuse these “jobs” with the Windows job objects described later. The jobs here are managed entirely from the programs developed in this section.
job management from separate shells or processes. Also, Exercise 6–8 identifies a
defect caused by job id reuse and suggests a fix.

The full program in the Examples file has a number of additional features, not
shown in the listings, such as the ability to take command input from a file. Job-
Shell will be the basis for a more general "service shell" in Chapter 13 (Program
13–3). Windows 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. Run 6–6 then demonstra-

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 "Everything.h"
#include "JobMgt.h"

int _tmain (int argc, LPTSTR argv[]) {
    BOOL exitFlag = FALSE;
    TCHAR command[MAX_COMMAND_LINE], *pc;
    DWORD i, localArgc; /* 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 (!exitFlag) {
        _tprintf (_T("%s"), _T("JM$"));
        fgets (command, MAX_COMMAND_LINE, stdin);
        pc = strchr (command, '\n');
        *pc = '\0';
        /* Parse the input to obtain command line for new job. */
        GetArgs (command, &localArgc, pArgs); /* See Appendix A. */
        charToLower (argstr[0]);
if (_tscmp (argstr[0], _T("jobbg")) == 0) {
    Jobbg (localArgc, pArgs, command);
} else if (_tscmp (argstr[0], _T("jobs")) == 0) {
    Jobs (localArgc, pArgs, command);
} else if (_tscmp (argstr[0], _T("kill")) == 0) {
    Kill (localArgc, pArgs, command);
} else if (_tscmp (argstr[0], _T("quit")) == 0) {
    exitFlag = 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 jobNumber;
    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 (argc[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." */

    jobNumber = GetJobNumber (&processInfo, targv); /* See JobMgt.h */
    if (jobNumber >= 0)
        ResumeThread (processInfo.hThread);
    else {
        TerminateProcess (processInfo.hProcess, 3);
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"), jobNumber, 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 ctrlC, ctrlB;

    iJobNo =
        Options (argc, argv, _T("bc"), &ctrlB, &ctrlC, NULL);

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

There's also a subtle defect in this function; a code comment identifies it, and Exercise 6–8 suggests a fix.

**Program 6-4  JobMgt:** Creating New Job Information

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

#include "Everything.h"
#include "JobMgt.h" /* Listed in Appendix A. */
void GetJobMgmtFileName (LPTSTR);
LONG GetJobNumber (PROCESS_INFORMATION *pProcessInfo,
    LPTSTR 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, fileSizeLow, fileSizeHigh;
    TCHAR jobMgmtFileName[MAX_PATH];
    OVERLAPPED regionStart;

    if (!GetJobMgmtFileName (jobMgmtFileName)) return -1; /* Produces "\tmp\UserName\JobMgt" */
    hJobData = CreateFile (jobMgmtFileName,
        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;

/* Find file size: GetFileSizeEx is an alternative */
errorCode = GetFileSizeEx (hJobData, &fileSizeHigh);
LockFileEx (hJobData, LOCKFILE_EXCLUSIVE_LOCK,
0, fileSizeLow + SJM_JOB, 0, &regionStart);

__try {
    /* Read records to find empty slot. */
    /* See text comments and Exercise 6-8 regarding a potential defect (and fix) caused by process ID reuse. */
    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);
} __endtry {
    UnlockFileEx (hJobData, 0, fileSizeLow + 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

```c
BOOL DisplayJobs (void)
{
    HANDLE hJobData, hProcess;
    JM_JOB jobRecord;
    DWORD jobNumber = 0, nXfer, exitCode, fileSizeLow, fileSizeHigh;
    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;

    /* Demonstration: GetFileSize instead of GetFileSizeEx */
    fileSizeLow = GetFileSize (hJobData, &fileSizeHigh);
    LockFileEx (hJobData, LOCKFILE_EXCLUSIVE_LOCK,
        0, fileSizeLow, fileSizeHigh, &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);
        }
    } _catch {
        _tprintf (_T(" Error %d\n"), GetLastError());
    }
}
```

**Example: Simple Job Management**

1. Display jobs in the job management function.
2. Use `GetJobMgtFileName` to get the job management file name.
3. Create a `HANDLE` for the job data file using `CreateFile`.
4. Set initial offset and overlapping event.
5. Get file size using `GetFileSize`.
6. Lock the file for exclusive access.
7. Loop through the jobs using `ReadFile`.
8. Display job information using `_tprintf`.
9. Handle exceptions using a try-catch block.
10. Display error message if an exception occurs.

This code demonstrates how to display active jobs in a job management function.
/* 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, fileSizeLow, fileSizeHigh,
        &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. */
SetFilePointer (hJobData, SJM_JOB * (jobNumber - 1),
NULL, FILE_BEGIN);

/* Lock and read the record. */
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;

EXAMPLE: SIMPLE JOB MANAGEMENT

Run 6-6 JobShell: Managing Multiple Processes
Run 6–6 shows the job shell managing several jobs using grep, grepMP, and sortBT (Chapter 5). Notes on Run 6–6 include:

- This run uses the same four 640MB files (11.txt, etc.) as Run 6–1.
- You can quit and reenter JobShell and see the same jobs.
- A “Done” job is listed only once.
- The grep job uses the -c option, so the results appear in a separate console (not shown in the screenshot).
- JobShell and the grepMP job contend for the main console, so some output can overlap, although the problem does not occur in this example.

**Job Objects**

You can collect processes together into job objects where the processes can be controlled together, and you can specify resource limits for all the job object member processes and maintain accounting information.

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 by a process in the job will also belong to the job unless the CREATE_BREAKAWAY_FROM_JOB flag is specified in the dwCreationFlags argument to CreateProcess.

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

```c
BOOL SetInformationJobObject ( 
    HANDLE hJob, 
    JOBOBJECTINFOCLASS JobObjectInformationClass, 
    LVOID lpJobObjectInformation, 
    DWORD chJobObjectInformationLength)
```
EXAMPLE: USING JOB OBJECTS

- 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, limits on the number of active processes, priority, and processor affinity (the processors of a multiprocessor computer 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.
- JOB_OBJECT_LIMIT_KILL_ON_JOB_CLOSE will terminate all processes in the job object when you close the last handle referring to the object.
- 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.

Example: Using Job Objects

Program 6–7, JobObjectShell, illustrates using job objects to limit process execution time and to obtain user time statistics. JobObjectShell is a simple extension of JobShell that adds a command line time limit argument, in seconds. This limit applies to every process that JobObjectShell manages.

When you list the running processes, you will also see the total number of processes and the total user time on a four-processor computer.

Caution: The term “job” is used two ways here, which is confusing. First, the program uses Windows job objects to monitor all the individual processes. Then, borrowing some UNIX terminology, the program also regards each managed process to be a “job.”

First, we’ll modify the usual order and show Run 6–7, which runs the command:

```
JobObjectShell 60
```

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.
to limit each process to a minute. The example then runs to shell commands:

```
    timep grep 1234561 12.txt 13.txt 14.txt
    timep grepMP 1234561 12.txt 13.txt 14.txt
```

as in Run 6–6. Note how the `jobs` command counts the processes that `timep` creates as well as those that `grepMP` creates to search the files, resulting in seven processes total. There is also a lot of contention for the console, mixing output from several processes, so you might want to run this example with the `-c` option.

There are also a few unexpected results, which are described for further investigation in Exercise 6–12.

Program 6–7 gives the `JobObjectShell` listing; it’s an extension of `JobShell` (Program 6–3), so the listing is shortened to show the new code. There are
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**

```c
/* 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 (argv[0]) and store the job id,
      the process id, and the handle in the jobs file. */
```
DWORD fCreate;
LONG jobNumber;
BOOL flags[2];

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

GetStartupInfo (&startTime);

/* 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] == 'r')
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, &startTime, &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 */

    /* Display 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.
Abandoned mutexes 281
AbnormalTermination function 114–115
ABOVE_NORMAL_PRIORITY_CLASS flag 247
accept function 417
Access
  rights 521
tokens 520, 543
Access control entries (ACEs) 521, 525–527, 535–537, 542
Access control lists (ACLs) 521, 525–527, 535–537, 542, 543
Access control lists, discretionary
  (DACLs) 520
ACCESS_ALLOWED_ACE flag 537
ACCESS_DENIED_ACE flag 537
ACE see Access control entries
ACL see Access control lists
ACL_REVISION word 526
ACL_SECURITY_INFORMATION value 536
ACL_SIZE_INFORMATION flag 536
AcquireSRWLockExclusive function 311
AcquireSRWLockShared function 310
AddAccessAllowedAce function 526
AddAccessDeniedAce function 526
AddAuditACCESSACE function 543
Address space 132
AddVectorExceptionHandler function 128
AF_INET 414
Alertable
  I/O 492
    wait functions 494–495
AllocateAndInitializesid function 524, 543
AllocConsole function 53
Anonymous pipes 380
APC see Asynchronous Procedure Calls
Application portability 372, 549
Asynchronous I/O 482
  with threads 500–501
Asynchronous Procedure Calls
  (APCs) 366–371
Asynchronous thread cancellation 371
Attributes
  directory 72–74
  file 70–74

B
  _based keyword 162
  Based pointers 161, 162
  _beginthreadex Microsoft C
    function 231–232
  BELOW_NORMAL_PRIORITY_CLASS 247
  Berkeley Sockets 411, 412, 447
  Binary search tree 143–144
  bind function 415
  Boss/worker model 236–237
  Broadcast mechanisms 401

C
  C library 10–11
    in threads 231–232
  cache 263–265
  Callback function 319–324
  CallNamedPipe function 390
  calloc C library function 143
  cancelIOEx 486
  cancelWaitableTimer function 502
cat
  file concatenation program 41, 197
  program run 43
  UNIX command 41
ccai
  Caesar Cipher program run 45
  file encryption program 44
  performance 581
ccai_f program 45
ccai_fMM program 157
ccaiEL
  program 173
  program run 174
cciEX
  performance 581
  program 497
  program run 499
cciLBSS
  performance 581
cciMM
  performance 581
  program run 159
cciMT
  performance 581
cciMTMM
  performance 581
cciOV
  performance 581
  program 488
  program run 491
CDFS see CD_ROM File System
CD-ROM File System (CDFS) 26
ChangeFilePermission program 539
ChangeServiceConfig2 parameter 469
CHAR type 34
Character types 34–36
chmod
  program 528
  UNIX command 528
chmodw, lsfp program run 531
clclearfp function 109
Client connections to named pipes 387
Client/server
  command line processor 393–400
  model 236, 384
  named pipe connection 389
clientNP
  program 393
  program run 401
clientserver.h 393
clientsK
  program 424
  program run 425
closedir UNIX function 74
CloseHandle function 18, 31, 71, 151
CloseServiceHandle function 469
closesocket function 418
CloseThresholdBarrier function 344
Closing files 31–32
COM 167
command program 435
CompareFileTime function 73
CompareSid function 543
Completion routines 492–495
Condition variable (CV) model 337–342
Condition variable predicates 337
condition variables 362
CONTIN$ path name 40
connect function 419
ConnectNamedPipe
  function 388, 483
CONOUT$ path name 40
Console
  control events 204–205
  control handlers 124–126, 185
  I/O 40–53
ConsolePrompt function 53
__controlfp function 108
ControlService function 470
CopyFile function 19, 47, 48
Copying files 46–48
copySid function 543
Co-routines 254
cp UNIX command 13
cpc
  C library program 13
  performance 578
CPCC
  performance 579
  program run 20
  Windows program 19
cpuc performance 579
cpw
  performance 579
  Windows file copying program 17
cpwPA performance 579
CREATE ALWAYS flag 30
CREATE NEW flag 30
CREATE NEW_CONSOLE flag 185
CREATE_NEW_PROCESS_GROUP flag 185, 204
CREATE_SUSPENDED flag 185, 228
CreateDirectory function 49
CreateEvent function 287, 485
CreateFile function 28–31, 71, 148
CreateFileMapping function 150–151
CreateHardLink function 47, 71
CreateIndexFile function 165
Create IoCompletionPort function 506, 507
CreateMailslot function 404
CreateMutex function 279–280
CreateNamedPipe function 385, 483
createPipe function 380
createPrivateObjectSecurity function 541
createProcess function 184–185, 204, 247
CreateRemoteThread function 228
CreateSemaphore function 284
CreateThread function 226–228
CreateThresholdBarrier function 344
CreateWaitableTimer function 501
Creating
directories 49
files 28–31
CRITICAL_SECTION (CS) 302, 336, 343
guidelines 294–295
locking and unlocking 307
objects 269, 281–284
Spin Counts 308–309
CS see CRITICAL_SECTION
CTRL_flags 126, 204
Ctrl_c
program 126
program run 128
CV see Condition variable

D
DACL see Discretionary access control list
DACL_SECURITY_INFORMATION value 536
DaclDefaultedFlag 527
Datagrams 445–447
Deadlocks 281–284
__declspec C++ storage modifier 169
DeleteAce function 542
DeleteFile function 46
DeleteService function 469
Deleting
directories 49
files 46–48
DETACHED_PROCESS flag 185, 204
DeviceIoControl function 65
Directories
attributes 72–74
creating 49
deleting 49
managing and setting 50–51
moving 46–49
naming 27–28
setting 187
DisconnectNamedPipe function 388
DLL see Dynamic link libraries
dllexport storage modifier 169
dlimport storage modifier 169
DllMain function 175
Drive names 27
DUPLICATE_CLOSE_SOURCE flag 192
DUPLICATESAMEACCESS flag 192
DuplicateHandle function 191
Duplicating handles 191
dw_prefix 9, 29
DWORD type 29
Dynamic
data structures 131
link libraries (DLLs) 149, 167–175
memory management 131–134

E
EM_floating point masks 109
ENABLE_flags 52
_endthreadex Microsoft C function 231
EnterCriticalSection function 270
Environment block 185, 195–196
Environment strings 195–196
ERROR_HANDLE_EOF return value 493
ERROR_TO_INCOMPLETE return value 486
ERROR_PIPE_CONNECTED return value 388
ERROR_SUCCESS flag 89
Errors 110–112
/etc UNIX directory 87
Event handle 485, 496
eventPC
program 290
program run 292
Events 287–289, 336
Everything.h 38
__except 102
Exception handlers 101–111
Exception program 121
EXCEPTION_exception codes 106, 110, 111, 113
EXCEPTION_return values 104, 129
EXCEPTION_MAXIMUM_PARAMETERS 111
EXCEPTION_POINTERS 107, 129
EXCEPTION_RECORD structure 107
exec UNIX functions 187
eexec UNIX function 187
Executable image 187
ExitProcess function 192, 228, 230
ExitThread function 228
Explicit linking 170–172
Exporting and importing interfaces 169–170
Extended I/O 492–495

F
FAT see File Allocation Table
fclose C library function 32
fcntl UNIX function 85
ferror C library function 16
Fibers 253–255
FIFO UNIX named pipe 392
File Allocation Table (FAT) file system 26
FILE C library objects 32
File handle 33, 61, 82, 150
File mapping objects 150–154
File permissions
  changing 538
  reading 537
FILE_ATTRIBUTE_flags 30–??, 71, 73
FILE_BEGIN position flag 61
FILE_CURRENT position flag 61
FILE_END position flag 61
FILE_FLAG_flags 30, ??–31, 63, 386
FILE_FLAG_OVERLAPPED flag 483, 485, 492
FILE_MAP_ALL_ACCESS flag 152
FILE_MAP_READ flag 152
FILE_MAP_WRITE flag 152
FILE_SHARE_READ flag 29, 404
FILE_SHARE_WRITE flag 29
Files
  attributes 70–74
  closing 31–32
  copying 46–48
  creating 28–31
  deleting 46–48
  handles 31
  locking 81–86
  memory-mapped 131
  moving 46–49
  naming 27–28, 74
  opening 28–31
  paging 135
  pointers 60–62
  reading 32–33
  resizing 64
  searching for 70–71
  systems 25–26
  writing 33
FILETIME 72, 202, 456, 460

FileTimeToLocalFileTime function 73
FileTimeToSystemTime function 72
FillTree program 147
Filter
  exception filtering program run 124
  function program 123
Filter expressions 103–104
  finally 113
FIND_DATA structure 70
FindFirstFile function 64, 70–71
FindNextFile function 71
Floating-point exceptions 108–110
FlushViewOfFile function 153
fopen C library function 32
fork UNIX function 186
errorMessage function 38
fread C library function 34
free C library function 143
FreeConsole function 53
FreeLibrary function 172
freopen C library function 32
FSCTL_SET_SPARSE flag 65
fwrite C library function 34

G
GenerateConsoleCtrlEvent
  function 124, 205
Generic characters 34–36
GENERIC_READ 29
  for named pipes 540
GENERIC_WRITE 29
  for named pipes 540
GetACE function 537
GetAclInformation function 536
GetCommandLine function 202
GetCompressedFileSize function 64
GetCurrentDirectory function 50
GetCurrentProcess function 190
GetCurrentProcessId function 190
GetCurrentThreadId function 229
GetDiskFreeSpace function 63
GetEnvironmentVariable function 196
GetExceptionCode function 105–106
GetExceptionInformation
  function 106
GetExitCodeProcess function 192–193
GetExitCodeThread function 229
GetFileAttributes function 73
GetFileInformationByHandle function 71
GetFileSecurity function 535
GetFileSize function 64
GetFileSizeEx function 64
GetFileTime function 72
GetFileType function 73
GetFullPathName function 72
GetKernelObjectSecurity function 541
GetLastError function 19.38
GetMailslotInfo function 404
GetModuleFileName function 172.191
GetModuleFileNameEx function 191
GetModuleHandle function 172
GetNamedPipeHandleState function 387
GetNamedPipeInfo function 388
GetOverlappedResult function 486
GetPriorityClass function 247
GetPrivateObjectSecurity function 541
GetProcAddress function 172
GetProcessAffinityMask function 318.330
GetProcessHeap function 134.142
GetProcessIdOfThread function 229
GetProcessorAffinityMask function 250
GetProcessTimes function 202
GetQueuedCompletionStatus function 507
GetSecurityDescriptorControl function 523
GetSecurityDescriptorDacl function 536
GetSecurityDescriptorGroup function 525.536
GetSecurityDescriptorOwner function 525.536
GetSecurityDescriptorSacl function 543
GetShortPathName function 72
GetStartupInfo function 185
GetStdHandle function 40
GetSystemDirectory function 187
GetSystemInfo function 134
GetTempFileName function 74
GetTempPath function 74
GetThreadIOPendingFlag function 229
GetThreadPriority function 248
GetThreadTimes function 202
GetTokenInformation function 543
GetUserName function 524
GetWindowsDirectory function 187
Global storage 266
Granularity, locking 295
grep UNIX command 197
grepMP
  performance 583
  program run 200
  search program 198
grepMT
  performance 583
  program run 235
  search program 233
grepSQL performance 583
GROUP_SECURITY_INFORMATION value 536
Growable and nongrowable heaps 137
Guarded code blocks 102–104

H
HAL see Hardware Abstraction Layer
HANDLE variable type 18
Handlers
  exception 101–111
  termination 113–117
Handles 7, 39
duplicating 191
inheritable 188–189
pseudo 190
hard link 47
Hardware Abstraction Layer (HAL) 5
Heap handle 137, 138
HEAP_GENERATE_EXCEPTIONS flag 106, 136, 138, 140, 141
HEAP_NO_SERIALIZE flag 136, 138, 140, 141
HEAP_REALLOC_IN_PLACE_ONLY flag 140
HEAP_ZERO_MEMORY flag 138, 140
HeapAlloc function 106, 138
HeapCompact function 142
HeapCreate function 106, 136
HeapDestroy function 137
HeapFree function 139, 171
HeapLock function 141, 142, 284
HeapReAlloc function 139
Heaps 134–143
   growable and nongrowable 137
   synchronizing 284
HeapsSize function 140
HeapUnlock function 142, 284
HeapValidate function 142
HeapWalk function 142
HIGH_PRIORITY_CLASS 247
HighPart data item 62
HINSTANCE handle 171
KEY__registry keys 88
htonl function 418
htons function 418
huge files 60

I
I/O
   alertable 492
   asynchronous 482
   completion ports 316, 505–509
   console 40–53
   extended 492–495
   overlapped 447, 483–486
   standard 40, 51, 188
IDLE_PRIORITY_CLASS 247
Implicit linking 168–170
INADDR_ANY flag 416
Inheritance, handles 191
InitializeAcl function 521, 526
InitializeConditionVariable
   function 362
InitializeCriticalSection
   function 265, 269, 309
InitializeCriticalSectionAndSpinCount function 271
InitializeSecurityDescriptor
   function 523
InitializeSid function 524
InitializeSRWLock function 310, 318, 319
InitializeUnixSA function 531
InitUnPp function 532
In-process servers 434
Interfaces, exporting and importing 169–170
Interlocked functions 265, 296–297
InterlockedCompareExchange
   function 297
InterlockedDecrement function 265
InterlockedExchange function 296
InterlockedExchangeAdd function 296
InterlockedIncrement function 265
Internet protocol 414
Interprocess communication (IPC)
   one-way 188
   two-way 384–392
INVALID_HANDLE_VALUE 134, 387, 506
INVALID_SOCKET 414
IP address 416
IPC see Interprocess communication
IsValidAcl function 535
IsValidSecurityDescriptor
   function 535
IsValidSid function 535

J
Job
   management 205
   objects 214–215
JobMgt
   displaying active jobs program 211
   new job information function 209
   process ID program 212
JobObjectShell program run 216
JobShell
   background job program 206
   program run 213

K
Kernel objects 8, 541
Key handle 89
KEY_ALL_ACCESS flag 89
KEY_ENUMERATE_SUBKEYS flag 89
KEY_QUERY_VALUE flag 89
KEY_WRITE flag 89

L
LARGE_INTEGER 202
   Microsoft C data type 62
   leave statement 114
LeaveCriticalSection function 270
Linking
   explicit 170–172
   implicit 168–170
   run-time 170–172
Linux xxvii
listen function 416
LoadLibrary function 171
LoadLibraryEx function 171
Local storage 266
INDEX  603

LocalFileTimeToFileTime function 73
LOCKFILE_EXCLUSIVE_LOCK flag 82
LOCKFILE_FAIL_IMMEDIATELY flag 82
LockFile function 81–82
LocServer locate the server function 407
LONG data type 62
LookupAccountName function 523–524
LowPart data item 62
lp prefix 29
lpsa prefix 30
lpsz prefix 29
LPTSTR type 35
lsFP program 530
lsReg
  listing Registry program 92
  program run 96
lsW
  file listing program 75
  program run 78

M
MAILSLOT_WAIT_FOREVER flag 404
Mailslots 401–405
main service entry program 455
MakeAbsoluteSD function 543
MakeSelfRelativeSD function 543
MAKEWORD macro 413
malloc C library function 143
Managing directories 50–51
Mapping, file 152–155
MapViewOfFile function 84, 152
MapViewOfFileEx function 152
Master-slave scheduling 255
MAX_PATH buffer length 51, 74
MAXIMUM_WAIT_OBJECTS 195
MCW_EM mask 109
Memory architecture 263
Memory barrier 263–268, 278
Memory block in heap 140
Memory management 131–134
  performance 297
Memory map size 152
Memory-mapped files 131, 149–155
MESSAGE type 422
Message waiting 294
Microsoft Visual C++ 547
mkfifo UNIX function 392
mmap UNIX function 154
mode UNIX argument 32
Mode word 51

Models
  boss/worker 236–237
  client/server 236, 384
  condition variable (CV) 337–342
  pipeline 236
  producer/consumer 331, 340
  threading 236–243
  work crew 236
MoveFile function 48–49
MoveFile_flags 49
MoveFileEx function 48–49
Moving
  directories 46–49
  files 46–49
MSG_PEEK flag 421
MsgWaitForMultipleObjects
  function 294
MsgWaitForMultipleObjectsEx 492
Multiple threads 340
Multiprocessor 5, 181, 201, 215
Multistage pipeline program 354
munmap UNIX function 154
Mutex 279–284, 336
  granularity 295
  guidelines 294
Mutual exclusion object 279–284

N
Named
  pipes 384–392
  sockets 416
Naming
  conventions 9
  directories 27–28
  drives 27
  files 27–28
NMPWAIT__named pipe flags 391
Nongrowable heap size 136
NORMAL_PRIORITY_CLASS flag 247
NT File System (NTFS) 26
NT services 453

O
Objects 195
  waiting for 294
Offset word 82, 484
offsetHigh word 82, 152, 484
Open systems 6–7
open UNIX function 32
OPEN_ALWAYS flag 30
Process
components 181–182
code 185
creation 183–186
evironment 195
handle inheritance 188–189
identities 190–191
priority 185
priority and scheduling 246–249
single 195
synchronization 194–195, 268–293
waiting for completion 194–195
PROCESS flags 190, 191, 247
PROCESS_INFORMATION structure 186, 190
ProcessItem function 529
Processor affinity 318, 329–331
Producer and consumer program 274
Producer/consumer model 331, 340
Program event logging 461
Program parallelism 244
pthread_Pthreads functions 231, 281, 288, 311, 339
Pthreads 362
application portability 372
condition variables 288, 339
in POSIX 230, 256, 280
open source implementation 376
PulseEvent function 288, 338, 340
pwd
program 55
program run 56
UNIX command 55

Q
qsort C library function 159
QuadPart data item 62
QueryJobInformationObject function 215
QueryServiceStatus function 471
QueueObj queue management functions 349, 363
Queues
definitions 348
in a multistage pipeline 352–354
management functions 349, 363
object 348–349
QueueUserAPC function 367
INDEX 605

R
Race conditions 267
RaiseException function 110–113
read UNIX function 33
ReadConsole function 52, 55
readdir UNIX function 74
ReadFile function 32–33, 380, 386, 483
ReadFileEx function 493
ReadFilePermissions program 537
Reading files 32–33
realloc C library function 143
REALTIME_PRIORITY_CLASS 247
ReceiveMessage function 423
RecordAccess
  program 66
    program run 69
recv function 420
recvfrom function 446
Redirect program run 383
ReferencedDomain 524
REG_flags 91
REG_BINARY registry data type 92
REG_DWORD registry data type 92
REG_EXPAND_SZ registry data type 92
REG_SZ registry data type 92
RegCloseKey function 89
RegCreateKeyEx function 90
RegDeleteKey function 91
RegDeleteValue function 92
REGEDIT32 command 86
RegEnumKeyEx function 90
RegEnumValue function 91
RegEnumValueEx function 92
Registry 86–88
  key management 89–91
RegOpenKeyEx function 89
RegQueryValueEx function 92
RegSetValueEx function 92
ReleaseMutex function 280
ReleaseSemaphore function 285, 342
ReleaseSRWLockExclusive
  function 311
ReleaseSRWLocksShared function 310
RemoveDirectory function 49
RemoveVectorExceptionHandler function 128
ReOpenFile function 31
ReportError
  program 39
ReportException function 112
ResetEvent function 288
ResumeThread function 185, 230
Run-time linking 170–172

S
SACL see System ACLs
SANs see Storage area networks
sbrk UNIX function 137
Scheduling 255
SCM see Service Control Manager
Searching for a file 70–71
SEC_IMAGE flag 150
Secure Socket Layer 434
Secure Sockets Layer 451
Security
  attributes 531
  attributes initialization program 532
  identifiers (SIDs) 523–525
  kernel object 541
  user object 541
  Windows objects 519
Security descriptors 520–527, 542–543
  reading and changing 535–537
SECURITY_ATTRIBUTES structure 188, 519–520
SECURITY_DESCRIPTOR structure 522
SEH see Structured Exception Handling
Semaphore 284–287, 342
Semaphore Throttle 313–315
send function 420
SendReceiveSKHA program 443
SendReceiveSKST program 438
sendto function 446
Sequential file processing 13
serverCP program 510
serverNP
  program 395
    program run 400
Servers, in-process 434
serverSK
  program 427
    program run 433
Service Control Manager (SCM) 454
SERVICE_AUTO_START flag 469
SERVICE_BOOT_START flag 469
SERVICE_DEMAND_START flag 469
SERVICE_STATUS structure 457–459
SERVICE_STATUS_HANDLE object 456
SERVICE_SYSTEM_START flag 469
SERVICE_TABLE_ENTRY array 455
ServiceMain functions 455–460

Services
  control handler 460–461
  control handler registration 456
  control manager 454
  control program 472
  controlling 470
  controls 460
  creating 468–469
  debugging 477
  deleting 468–469
  opening 467
  setting status 456
  starting 469
  state 459
  status query 471
  type 458
  wrapper program 462

ServiceShell
  program 472
  program run 476
  ServiceStartTable function 455
  ServiceStatus function 457
  ServiceType word ??–458
  SetConsoleCtrlHandler function 124
  SetConsoleMode function 51–52
  SetCriticalSectionSpinCount function 271, 309
  SetCurrentDirectory function 50
  SetEndOfFile function 64
  SetEnvironmentVariable function 196
  SetEvent function 288, 340
  SetFileAttributes function 73
  SetFilePointer function 60, 61, 62, 485
  SetFileSecurity function 535
  SetFileShortName function 72
  SetFileTime function 73
  SetInformationJobObject function 214
  SetKernelObjectSecurity function 541
  SetMailslotInfo function 404
  SetNamedPipeHandleState function 387
  SetPriorityClass function 247
  SetPrivateObjectSecurity function 541
  SetProcessAffinityMask function 250, 330
  SetSecurityDescriptorControl function 523
  SetSecurityDescriptorDacl function 527
  SetSecurityDescriptorGroup function 525
  SetSecurityDescriptorOwner function 521
  SetSecurityDescriptorSacl function 543
  SetServiceStatus function 457
  SetStdHandle function 40
  SetThreadAffinityMask function 330
  SetThreadIdealProcessor function 250
  SetThreadPriority function 248
  SetThreadPriorityBoost function 249
  Setting directories 50–51
  SetWaitableTimer function 502

Shared
  memory in UNIX 154
  variables 271–273
  shutdown function 417
  SID management 543
  STD_NAME_USE flag 524
  SIDs see Security identifiers
  Signaled state 230
  Signaling producer and consumer program 290
  SignalObjectAndWait (SOAW) function 337, 339, 342–344, 492
  Signals 125, 185
    in UNIX 113
  simplePC program 274
  simplePC program run 277
  SimpleService
    operation 476
  SimpleService program 462
  SimpleService program run 466
  SimpleServiceLog.txt listing 467
  64-bit file addresses 59–60
  SkipArg function 202
  Sleep function 253
  SleepConditionVariableCS function 362
  SleepConditionVariableSRW function 363
  SleepEx function 494
Sockets
  Critical Section comparison 310
  SMP see Symmetric multiprocessing
  SOAW see signalAndWait
  SOCK_DGRAM flag 445
  sockaddr structure 415
  sockaddr_in structure 416
  socket function 414
  SOCKET_ERROR flag 415, 420
  Socket-based
    client program 424
    server program 427
  Sockets
    Berkeley 412, 447
    binding 415–416
    client functions 419–422, 423
    closing 417
    connecting to client 417
    connecting to server 419–420
    creating 414
    disconnecting 417
    initialization 413
    message receive 422–423
    server functions 414–419, 426
  sort UNIX command 143
  sortio
    binary search tree program 145
    program run 148
  sortiof
    program 159
    program run 161
  sortim
    based pointers program 163
    creating the index program 165
    program run 166
  sortimt
    merge-sort program 239
    program run 242, 243
  Sparse file 64
  Spin Counts 271, 297, 308–309
  SrvcBsc files
    mailslot client program 406
  SW see Slim Reader/Writer
  SSIZE_T data type 136
  SSL see Secure Sockets Layer
  Stack unwind 116
  Standard
    I/O 40, 51, 188
    input 188
  StartAddr function pointer 227
  STARTF_USESTDHANDLES flag 185
  StartService function 470
  StartServiceCtrlDispatcher function 454
  stat UNIX function 74
  STATE_TYPE data structure 337
  statsMX
    program run 315
    thread statistics program 303
  statsSRW program run 322
  statsXX program run 305, 312
  Status functions for named pipes 387
  STATUS_ACCESS_VIOLATION exception code 141
  STATUS_NO_MEMORY exception code 106, 141
  STDERROR_HANDLE flag 40
  STD_INPUT_HANDLE flag 40
  STD_OUTPUT_HANDLE flag 40
  STILL_ACTIVE process status 193, 229
  Storage area networks (SANs) 26
  Storage, local and global 266
  Strings, environment 195–196
  Structured Exception Handling (SEH) 101–102, 117
  Structures, overlapped 484–485
  SuspendThread function 230
  Symmetric multiprocessing (SMP) 181, 264
  SynchObj
    queue definitions 348
    threshold barrier definitions
      program 345
  Synchronization 246–249
    heap 284
    objects 492
    performance impact 302–303
    processes 194–195
    processes and threads 268–293
  SYNCHRONIZE flag 190
  synchronous cancellation 371
  System
    ACLs (SACLs) 520, 543
    error codes 19
    include files 9
    SystemTimeToFileTime function 73
T
  tchar.h 35
  TCHAR type 34
  TCP/IP 412, 414
Temporary file names 74
TerminateProcess function 193, 205
TerminateThread function 228, 230
Termination handlers 113–117
testTHB
  program run 347
testTHB test program 345
THREAD_HANDLE threshold barrier handle 344
ThreadObject threshold barrier implementation program 345
Thread Local Storage (TLS) 182, 225, 245–246
Thread pool 312–323
Thread stack 372
THREAD_MODE_BACKGROUND_BEGIN flag 248
THREAD_MODE_BACKGROUND_END flag 248
THREAD_PRIORITY flags 248
ThreadFarm thread argument 228
Threadpool timers 505
Threads
  common mistakes 251–252
  creating 226–228
  file locking 81–86
  identity 229
  local storage (TLS) 225
  models 236–243
  overview 223–224
  primary 184
  priority and scheduling 246–249
  resuming 229–230
  single 181–182
  states 249–251
  statistics program 303
  storage 225–226, 245–246
  suspending 229–230
  synchronization 246–249, 268–293
  terminating 228
  waiting for termination 230
  with asynchronous I/O 500–501
  with the C library 231–232
Thread-safe
  code 259–268
  DLL program 438
  DLL program with state structure 443
  libraries 232
ThreeStage.c multistage pipeline
  program 354
ThreeStage_[_Sig] program run 360
ThreeStageCS[_[Sig]] program run 361
ThreeStageCV program run 366
Threshold barrier object 344–348
time UNIX command 202
TimeBeep program 503
Timed waits 252
timep
  performance 575
  process times program 203
Timers
  waitable 501–503
TLS see Thread Local Storage
TLS_MINIMUM_AVAILABLE flag 245
TlsAlloc function 246
TlsFree function 246
TlsGetValue function 246
TlsSetValue function 246
_tm_fun function 36
touch
  program 79
  program run 79
toupper
  program 118
  program run 120
TransactNamedPipe function 390, 483
TRUNCATE_EXISTING flag 30
__try 102
TryEnterCriticalSection
  function 270
Try-except blocks 102–104, 113–116
Try-finally blocks 113–116
U
UCT see Universal Coordinated Time
UDF see Universal Disk Format
ULONGLONG data type 62
Unicode 34–36
Unicode UTF-16 34
Universal Coordinated Time (UCT) 72
Universal Disk Format (UDF) 26
unlink UNIX function 49
UnlockFileEx function 83
UnmapViewOfFile function 153
Unwinding stacks 116
utime UNIX function 74
V
va_arg C library function 53
va_end C library function 53
va_start C library function 53
Value management 91–92
INDEX

Variables, environment 195–196
vectorhandler 128
version exercise run 180
Virtual
  address space 132
  memory manager 133
  memory space allocation 152
Visual C++ 9.22
volatile storage modifier 262, 265, 374

W
Wait
  for messages and objects 294
    functions 494–495
WATT_ABANDONED_0 return value 195, 281
WATT_FAILED return value 195
WATT_OBJECT_0 return value 195
WATT_TIMEOUT return value 195
Waitable timers 501–503
WaitForMultipleObjects function 194–195, 230, 279, 288
WaitForMultipleObjectsEx
    function 494
WaitForSingleObject function 194–195, 230, 279
WaitForSingleObjectEx function 494
Waiting for a process 194–195
WaitNamedPipe function 388
WakeAllConditionVariable
    function 363
wakeConditionVariable function 363
wchar.h 36
WCHAR type 34
Win16 compatibility 9
winbase.h file 9, 541
Windows
  API 2
    condition variables 362–365
    principles 7–9
    sockets 412, 447, 448
    support 5, 181
    versions 3
Windows 2003 Server 31
windows.h file 18, 9
winnt.h file 9, 29, 107, 195
Winsock 411
  API 412
    Initialization 413
Work crew model 236
writeUNIX function 33
WRITE_DAC permission 536
WriteConsole function 52
WriteFile function 33, 483
WriteFileEx function 493
Writing files 33
WS_2.DLL 413
WSACleanup function 413
WSADATA structure 413
WSAGetLastError function 413
WSAStartup function 413