```
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 `getgpid` 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:

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

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

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

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

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

```
Name = Value
```

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

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

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:

`lpName` is the variable name. On setting a value, the variable is added to the block if it does not exist and if the value is not `NULL`. If, on the other hand, the value is `NULL`, the variable is removed from the block. The "=" character cannot appear in an environment variable name, since it's used as a separator.

There are additional requirements. Most importantly, the environment block strings must be sorted alphabetically by name (case-insensitive, Unicode order). See MSDN for more details.

`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 `GetExitCodeProcess`.

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) handles, so the program calls it multiple times.
- The program uses the `grep` process exit code to determine whether a specific process detected the pattern.

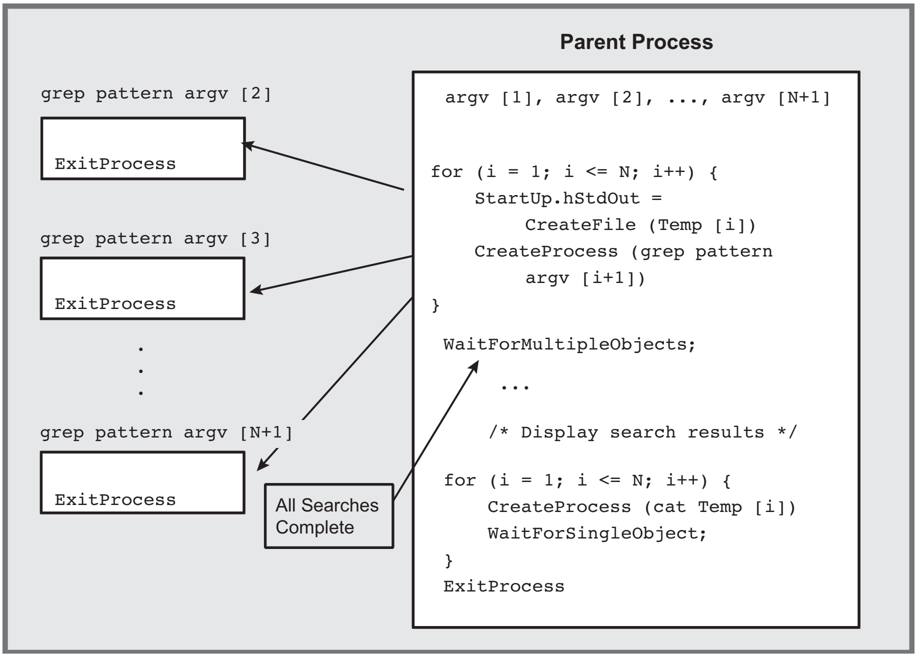


Figure 6–3 File Searching Using Multiple Processes

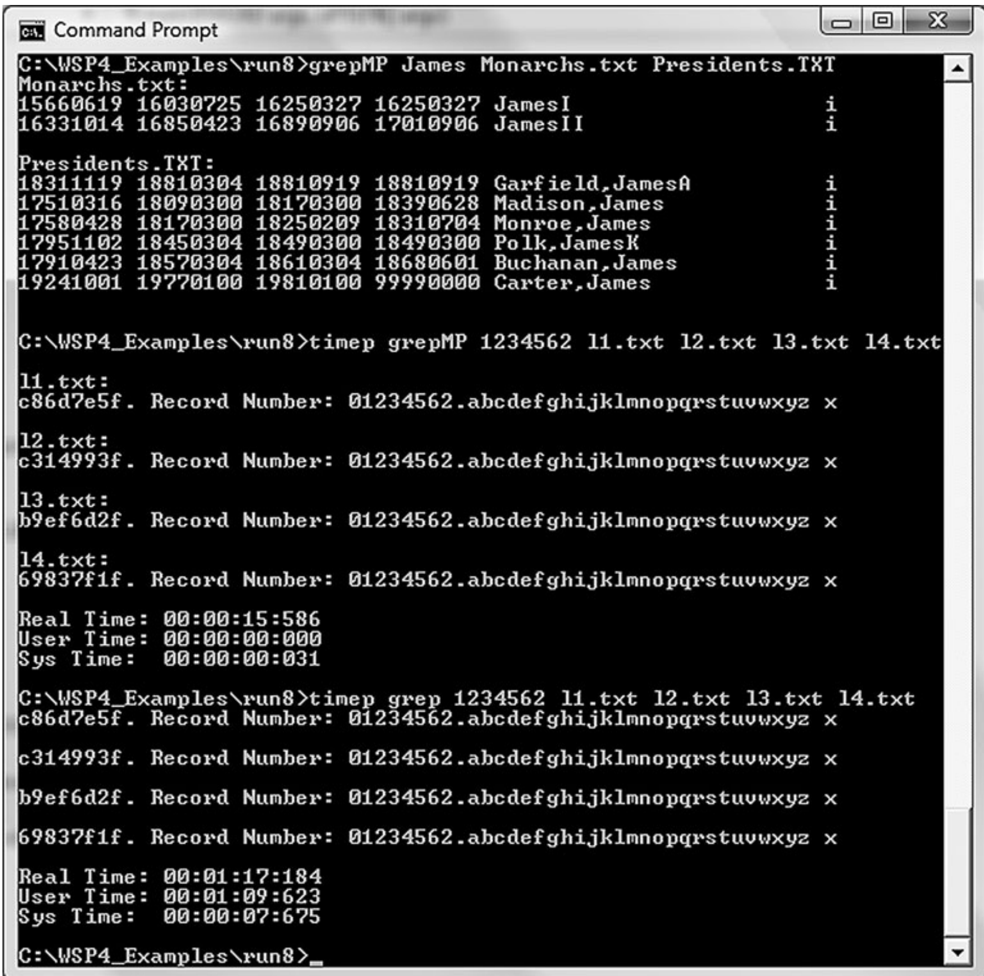
Figure 6–3 shows the processing performed by Program 6–1, and Run 6–1 shows program execution and timing results.

Program 6–1 `grepMP`: Parallel Searching

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

#include "Everything.h"
int _tmain (DWORD argc, LPTSTR argv[])
/* Create a separate process to search each file on the
   command line. Each process is given a temporary file,
   in the current directory, to receive the results. */
{
   HANDLE hTempFile;
   SECURITY_ATTRIBUTES stdOutSA = /* SA for inheritable handle. */
          {sizeof (SECURITY_ATTRIBUTES), NULL, TRUE};
   TCHAR commandLine[MAX_PATH + 100];
   STARTUPINFO startUpSearch, startUp;
   PROCESS_INFORMATION processInfo;
```

```
DWORD iProc, 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 (iProc = 0; iProc < argc - 2; iProc++) {
   _stprintf (commandLine, _T ("grep \"%s\" \"%s\""),
         argv[1], argv[iProc + 2]);
   GetTempFileName (_T ("."), _T ("gtm"), 0,
         procFile[iProc].tempFile); /* For search results. */
   hTempFile = /* This handle is inheritable */
      CreateFile (procFile[iProc].tempFile,
         GENERIC_WRITE,
         FILE_SHARE_READ | FILE_SHARE_WRITE, &stdOutSA,
         CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
   startUpSearch.dwFlags = STARTF_USESTDHANDLES;
   startUpSearch.hStdOutput = hTempFile;
   startUpSearch.hStdError = hTempFile;
   startUpSearch.hStdInput = GetStdHandle (STD_INPUT_HANDLE);

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

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

```
            CloseHandle (processInfo.hProcess);
            CloseHandle (processInfo.hThread);
        }

        CloseHandle (hProc[iProc]);
        DeleteFile (procFile[iProc].tempFile);
    }
    free (procFile);
    free (hProc);
    return 0;
}
```



```
Command Prompt
C:\WSP4_Examples\run8>grepMP James Monarchs.txt Presidents.TXT
Monarchs.txt:
15660619 16030725 16250327 16250327 JamesI              i
16331014 16850423 16890906 17010906 JamesII             i

Presidents.TXT:
18311119 18810304 18810919 18810919 Garfield,JamesA     i
17510316 18090300 18170300 18390628 Madison,James       i
17580428 18170300 18250209 18310704 Monroe,James        i
17951102 18450304 18490300 18490300 Polk,JamesK         i
17910423 18570304 18610304 18680601 Buchanan,James      i
19241001 19770100 19810100 99990000 Carter,James        i


C:\WSP4_Examples\run8>timep grepMP 1234562 l1.txt l2.txt l3.txt l4.txt

l1.txt:
c86d7e5f. Record Number: 01234562.abcdefghijklmnopqrstuvwxyz x

l2.txt:
c314993f. Record Number: 01234562.abcdefghijklmnopqrstuvwxyz x

l3.txt:
b9ef6d2f. Record Number: 01234562.abcdefghijklmnopqrstuvwxyz x

l4.txt:
69837f1f. Record Number: 01234562.abcdefghijklmnopqrstuvwxyz x

Real Time: 00:00:15:586
User Time: 00:00:00:000
Sys Time:  00:00:00:031

C:\WSP4_Examples\run8>timep grep 1234562 l1.txt l2.txt l3.txt l4.txt
c86d7e5f. Record Number: 01234562.abcdefghijklmnopqrstuvwxyz x

c314993f. Record Number: 01234562.abcdefghijklmnopqrstuvwxyz x

b9ef6d2f. Record Number: 01234562.abcdefghijklmnopqrstuvwxyz x

69837f1f. Record Number: 01234562.abcdefghijklmnopqrstuvwxyz x

Real Time: 00:01:17:184
User Time: 00:01:09:623
Sys Time:  00:00:07:675

C:\WSP4_Examples\run8>_
```

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.

```
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 `timep` (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). `timep` 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 `timep`: Process Times

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

#include "Everything.h"
int _tmain (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 targv = SkipArg (GetCommandLine ());
   HANDLE hProc;

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

   /* Execute the command line; wait for process to complete. */
   CreateProcess (NULL, targv, NULL, NULL, TRUE,
         NORMAL_PRIORITY_CLASS, NULL, NULL, &startUp, &procInfo);
   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:%03d\n"),
         elTiSys.wHour, elTiSys.wMinute, elTiSys.wSecond,
         elTiSys.wMilliseconds);
   _tprintf (_T ("User Time: %02d:%02d:%02d:%03d\n"),
         usTiSys.wHour, usTiSys.wMinute, usTiSys.wSecond,
         usTiSys.wMilliseconds);
   _tprintf (_T ("Sys Time: %02d:%02d:%02d:%03d\n"),
         keTiSys.wHour, keTiSys.wMinute, keTiSys.wSecond,
         keTiSys.wMilliseconds);

   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.