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

The first parameter, then, must be one of either `CTRL_C_EVENT` or `CTRL-_BREAK_EVENT`. The second parameter identifies the process group.

Example: Simple Job Management

UNIX shells provide commands to execute processes in the background and to obtain their current status. This section develops a simple "job shell"[2] 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

[2] 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. `JobShell` 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 demonstrates how to use the `JobShell` system.

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$"));
      _fgetts (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. */
      CharLower (argstr[0]);
      if (_tcscmp (argstr[0], _T ("jobbg")) == 0) {
         Jobbg (localArgc, pArgs, command);
      }
      else if (_tcscmp (argstr[0], _T ("jobs")) == 0) {
         Jobs (localArgc, pArgs, command);
      }
      else if (_tcscmp (argstr[0], _T ("kill")) == 0) {
         Kill (localArgc, pArgs, command);
      }
      else if (_tcscmp (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 (argv[1][0] == '-') targv = SkipArg (targv);

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

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

   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 cntrlC, cntrlB;

   iJobNo =
      Options (argc, argv, _T ("bc"), &cntrlB, &cntrlC, NULL);

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

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

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

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

/* Create a job number for the new process, and enter
   the new process information into the job database. */
{
   HANDLE hJobData, hProcess;
   JM_JOB jobRecord;
   DWORD jobNumber = 0, nXfer, exitCode, fileSizeLow, fileSizeHigh;
   TCHAR jobMgtFileName[MAX_PATH];
   OVERLAPPED regionStart;

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

```
    if (hJobData == INVALID_HANDLE_VALUE) return -1;

    /* Lock the entire file plus one possible new
       record for exclusive access. */
    regionStart.Offset = 0;
    regionStart.OffsetHigh = 0;
    regionStart.hEvent = (HANDLE)0;

    /* Find file size: GetFileSizeEx is an alternative */
    fileSizeLow = GetFileSize (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);
    } /* End try. */

    __finally {
       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

```
BOOL DisplayJobs (void)

/* Scan the job database file, reporting job status. */
{
   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);
      /* 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

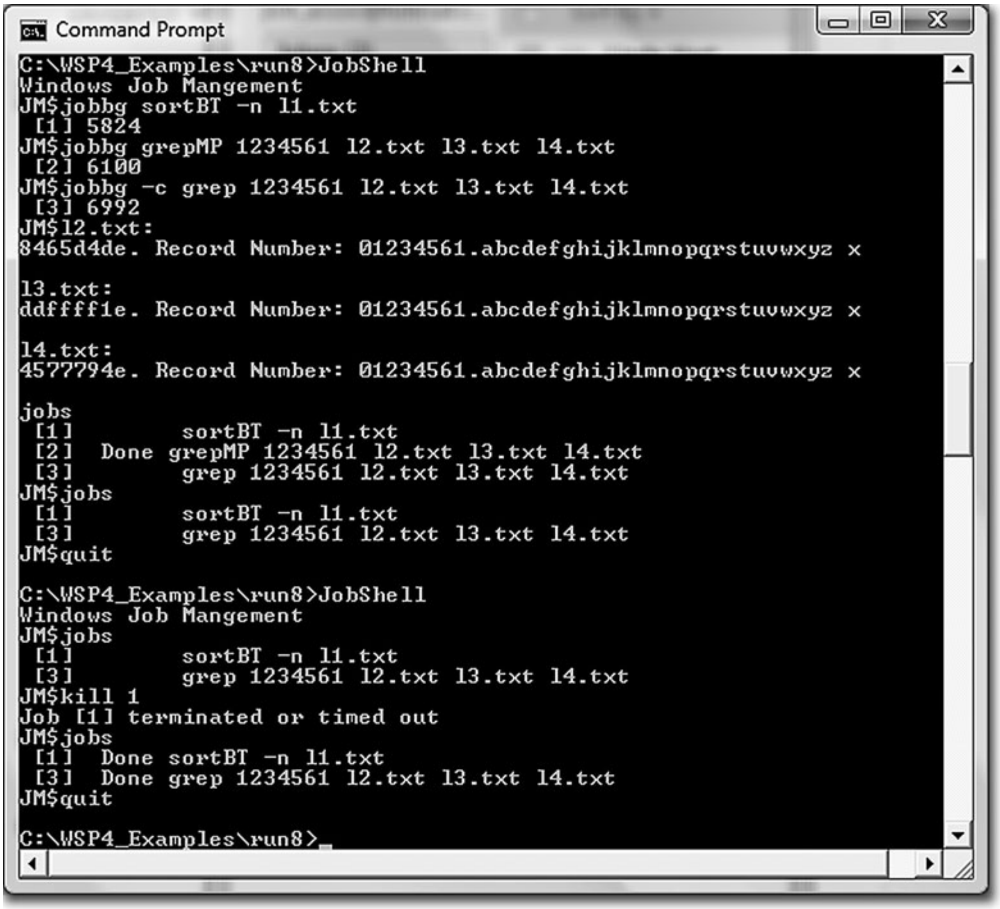
```
DWORD FindProcessId (DWORD jobNumber)

/* 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.
    * The full program assures that jobNumber is in range. */
   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;
}
```



```
C:\WSP4_Examples\run8>JobShell
Windows Job Mangement
JM$jobbg sortBT -n l1.txt
 [1] 5824
JM$jobbg grepMP 1234561 l2.txt l3.txt l4.txt
 [2] 6100
JM$jobbg -c grep 1234561 l2.txt l3.txt l4.txt
 [3] 6992
JM$l2.txt:
8465d4de. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x

l3.txt:
ddffff1e. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x

l4.txt:
4577794e. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x

jobs
 [1]         sortBT -n l1.txt
 [2]    Done grepMP 1234561 l2.txt l3.txt l4.txt
 [3]         grep 1234561 l2.txt l3.txt l4.txt
JM$jobs
 [1]         sortBT -n l1.txt
 [3]         grep 1234561 l2.txt l3.txt l4.txt
JM$quit

C:\WSP4_Examples\run8>JobShell
Windows Job Mangement
JM$jobs
 [1]         sortBT -n l1.txt
 [3]         grep 1234561 l2.txt l3.txt l4.txt
JM$kill 1
Job [1] terminated or timed out
JM$jobs
 [1]    Done sortBT -n l1.txt
 [3]    Done grep 1234561 l2.txt l3.txt l4.txt
JM$quit

C:\WSP4_Examples\run8>_
```

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 (`l1.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`.

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

- `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,[3] 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
```

[3] 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.

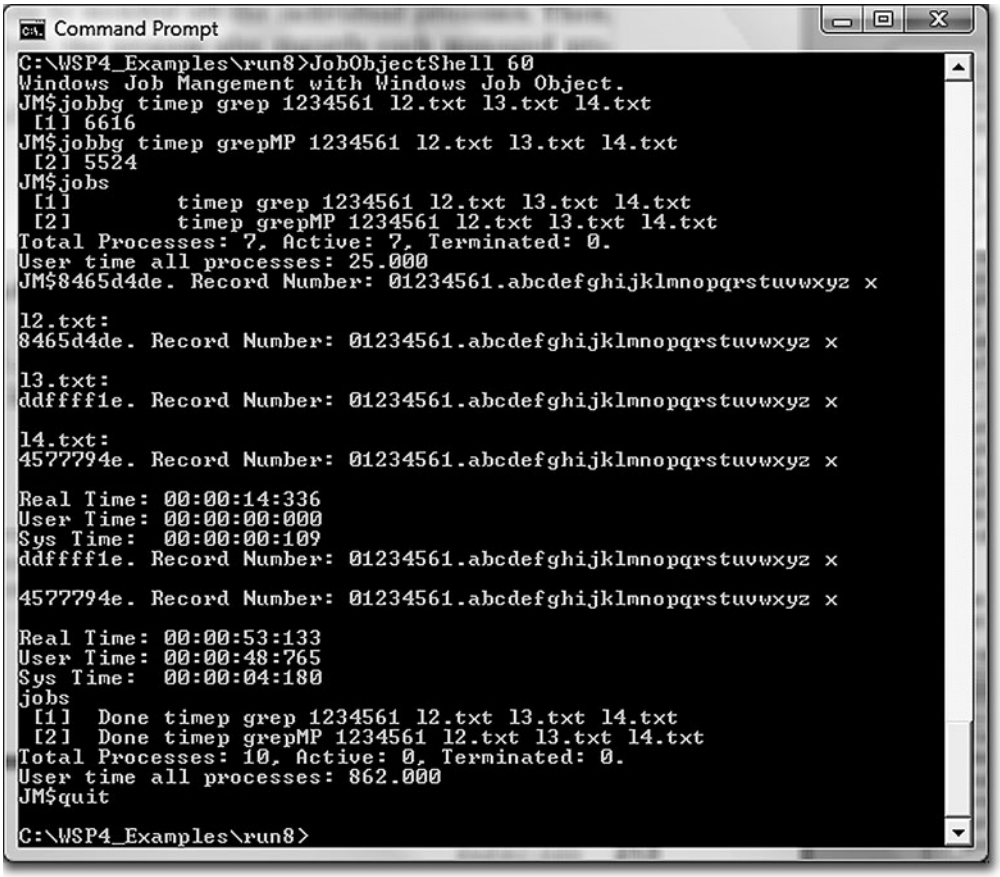
to limit each process to a minute. The example then runs to shell commands:

```
timep grep 1234561 l2.txt l3.txt l4.txt
timep grepMP 1234561 l2.txt l3.txt l4.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



```
C:\WSP4_Examples\run8>JobObjectShell 60
Windows Job Mangement with Windows Job Object.
JM$jobbg timep grep 1234561 l2.txt l3.txt l4.txt
 [1] 6616
JM$jobbg timep grepMP 1234561 l2.txt l3.txt l4.txt
 [2] 5524
JM$jobs
 [1]         timep grep 1234561 l2.txt l3.txt l4.txt
 [2]         timep grepMP 1234561 l2.txt l3.txt l4.txt
Total Processes: 7, Active: 7, Terminated: 0.
User time all processes: 25.000
JM$8465d4de. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x

l2.txt:
8465d4de. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x

l3.txt:
ddffff1e. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x

l4.txt:
4577794e. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x

Real Time: 00:00:14:336
User Time: 00:00:00:000
Sys Time:  00:00:00:109
ddffff1e. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x

4577794e. Record Number: 01234561.abcdefghijklmnopqrstuvwxyz x

Real Time: 00:00:53:133
User Time: 00:00:48:765
Sys Time:  00:00:04:180
jobs
 [1]  Done timep grep 1234561 l2.txt l3.txt l4.txt
 [2]  Done timep grepMP 1234561 l2.txt l3.txt l4.txt
Total Processes: 10, Active: 0, Terminated: 0.
User time all processes: 862.000
JM$quit

C:\WSP4_Examples\run8>
```

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