Threading in VB4

Click & Retrieve Source CODE!

Use objects as threads in Visual Basic 4.0, thanks to the Win32 API and C/C++.

by Stephen R. Husak

Although VB4 is 32-bit, a disappointment with the language is the inability to create multithreading applications that take advantage of enhanced operating systems such as Windows 95 and Windows NT. Because native Visual Basic 4.0 does not have the built-in ability to generate and use threads using pure VB code, the VB4 runtime is not multithreaded. Visual Basic 4.0 cannot pass a Visual Basic function address to an API function that the Win32 API CreateThread function requires.

CreateThread needs an address of a function for which the thread should start executing (you can think of this as the Sub Main of the thread). One means for utilizing these true multitasking operating systems exists in using objects as threads. Previously, developers needed a relatively deep knowledge of C/C++ to implement threads, but now with some minor knowledge of C/C++ and the Win32 API, you can make a Visual Basic program that rivals most C/C++ programs.

Threads are a specific path of execution within a process. A thread typically has its own stack space in the process' address space, and its own set of CPU registers. Assuming a process is an instance of a running program, each executing process has at least one thread called the primary thread. All threads that are not primary threads are known as worker threads, and they do the small specific tasks within the program.

An application normally uses threads for small, quickly executed, specific tasks, such as Excel's background recalculation of a spreadsheet, and Word 95's backup spell-checking. Even Visual Basic 4.0's development environment uses threads to compile in the background. Threads give the user the appearance of responsiveness from an application. Excel users can watch the bottom line change as they enter year-end data. In Word, the user can see spelling mistakes as he or she types. In VB4, the user can run the program almost instantaneously after writing a function because most of the program is already compiled. By accomplishing these specific tasks, applications appear much more functional to the end user.

The programmer creates this responsiveness by having threads that are controlled or scheduled by the operating system based on priorities assigned to the threads. You can schedule threads by four main priorities: real-time, high, normal, and idle. You will rarely use real-time priority. Usually, programmers use it for applications that communicate directly to hardware that cannot be interrupted. High-priority threads—such as the one that watches for the Control-Alt-Delete key combination to display the Task Manager in Windows 95—are extremely responsive when running.

No matter what is happening in the system, the task manager is promptly displayed when the user presses Control-Alt-Delete. The majority of applications run with normal-priority threads. The foreground application, however, always has a slightly higher priority than the others. By default, all threads are initially assigned normal priority. Programmers typically use idle-priority threads for applications that monitor the system, such as the thread that zeros pages in RAM. Idle-priority threads only run when no other threads are running. Threads are run and scheduled by the operating system. The OS interrupts the CPU, executes the thread with the highest priority at that time, and waits until it is time to run the next thread. This is ideal on machines with multiple CPUs where multiple threads can actu-

```
Private Sub cmdThread_Click()
Dim lAddress As Long
' address of the thread's function
Dim lID As Long
' thread id
Dim dd As DIRDATA
' DirData structure for thread function
Screen.MousePointer = vbArrowHourglass
' clear form and lock the controls
Call ClearFrmMain
Call LockFrmControls(True)
' get starting time and start the polling timer
TimeStarted = CDbl(Now)
tmrPollThread.Enabled = True
' get address of DirBrowseThread function in our dll
lAddress = GetProcAddress(gh_Library, _
  "DirBrowseThread")
' set up the structure to pass to the thread
dd.hWndList = lstDirectories.hWnd
dd.hWndDirs = txtDirCount.hWnd
dd.hWndFiles = txtFileCount.hWnd
dd.sFileSpec = "*.*"
dd.lFileSpecLen = 3
dd.sPath = Left$(drvDrives.Drive, 1) & ":"
dd.lPathLen = 2
' create the thread in the context of VB's main thread
gh_Thread = 0
gh_Thread = CreateThread(ByVal _
  vbNullString, 0, lAddress, dd, 0, lID)
' user still has control ->
' timer will watch thread from here
' NOTE: if you have no need to watch
' the thread, the function
' would simply end here -> nothing else to do
End Sub
```

LISTING 1 ***Creating a Thread.*** *This code demonstrates how the thread is set up, as well as how the Win32 API call CreateThread is called after the DIRDATA structure is set up.*

Stephen R. Husak is a software engineer at Zenith Data Systems Corporation, a world-wide computer manufacturer of Intel and RISC-based machines for large corporations, education, the federal government, and resellers. Reach Stephen by e-mail at husak@netcom.com or s.husak@zds.com, or by CompuServe at 102467,3223.

ally be running at the same time on different CPUs.

Threads are best used when they can make the most of the CPU's time and optimize the user's time as well. Recalculating a spreadsheet or printing in the background are typical examples of how to use threads. Notice that these examples are small tasks that do not disturb the user as he or she works on the system. Typically the programmer uses threads in these situations when the user is not affected by the performance of the thread.

GENERATING AND USING THREADS

Where data integrity is essential, pay close attention to how you use multiple threads. For example if two threads are writing to the same global variables or structures, and if one thread depends on the results of another, there is a danger that one thread will access the same area that the other thread is accessing. This can disrupt the integrity of the data. You can use thread synchronization techniques to manage this.

Thread synchronization works by creating objects that "watch" and wait for certain events to happen. Three such objects are events, mutexes, and semaphores. Events are basically Boolean states that either are signaled or nonsignaled. Mutexes are used to synchronize multiple threads and are owned by a thread. Semaphores are used for resource counting and are not owned by a thread. The Win32 API call WaitForSingleObject suspends a thread until the object becomes signaled. Likewise the call WaitForMultipleObjects suspends a thread until all objects in the list become signaled. Data integrity is safe-guarded with these objects.

You can use threads in VB4 by using an external C/C++ DLL that contains the thread's function. The Visual Basic code uses only the Win32 API to achieve the multithreading along with the DLL function. I will present sample code that shows you how to use a timer to communicate when the thread has terminated (you may download the DLL and all the code described in this column from a file titled VP0496.ZIP on *VBPJ's* CompuServe Forum, and World Wide Web and Microsoft Network sites. For details, see "How to Reach Us" in Letters to the Editor.) This minimizes VB program performance degradation and keeps the user abreast of the status of the thread.

Visual Basic isn't good at synchronizing threads. You can overcome this limitation by suspending Visual Basic's main thread by using the Win32 API calls for WaitForSingleObject or WaitForMultipleObjects.

You could synchronize threads with a "watcher" thread, but doing so will cause two threads to be created for each thread that needs to be executed. The implementation here is not generally recommended for watching threads because it continually polls the thread waiting for the termination of it. This is, however, the only way for VB's main thread to know when the worker thread has completed without suspending Visual Basic's main thread. If Visual Basic's main thread does not need to know

```
DLLEXPORT DWORD WINAPI DirBrowseThread(DIRDATA * dd)
{
char * sPath;
// buffer for path
  char * sFileSpec;
// buffer for file specification
  // allocate buffers and adjust
  // character widths passed from VB
  sPath = (char *) malloc(dd->lPathLen + 1);
  strncpy(sPath, dd->path, dd->lPathLen);
  sPath[dd->lPathLen] = 0;
  sFileSpec = (char *) malloc(dd->lFileSpecLen + 1);
  strncpy(sFileSpec, dd->filespec, dd->lFileSpecLen);
  sFileSpec[dd->lFileSpecLen] = 0;
  // set global variables
  gl_DirCount = 0;
  gl_FileCount = 0;
  // start the browse
  DirBrowse(sPath, sFileSpec, _
     dd->hWndList, dd->hWndDirs, dd->hWndFiles);
  // deallocate buffers & reset global variables
  free(sPath); free(sFileSpec);
  gl_DirCount = 0;
  gl_FileCount = 0;
  // exit the thread
  ExitThread(ERROR_SUCCESS);
  return (ERROR_SUCCESS);
}

DLLEXPORT void WINAPI _
  DirBrowse(LPSTR path, LPSTR filespec, HWND _
  hWndList,
             HWND hWndDirs, HWND hWndFiles) {
WIN32_FIND_DATA finddata;
// find data structure filled by calls
  HANDLE fSearch;
// handle to the file find instance
  BOOL rc = TRUE;
  // boolean return code fails when FindNextFile
  //fails
  char work[MAX_PATH];
  // work buffer to build paths and file specs
  // build the string required
  // by FindFirstFile (path\filespec
  // eg. c:\*.*)
  sprintf(work, "%s\\%s", path, filespec);
  // find the first file matching the spec
  fSearch = FindFirstFile(work, &finddata);
  while (rc && (fSearch != INVALID_HANDLE_VALUE)) {
     // weed out directories return as . and ..
     if (finddata.cFileName[0] != '.') {
        // send the name to the listbox
        sprintf(work, "%s\\%s", path, _
           finddata.cFileName);
        ListBox_AddString(hWndList, work);
        if (finddata.dwFileAttributes & _
           FILE_ATTRIBUTE_DIRECTORY) {
           // this is a directory
           // so count it and
           // recurse its structure
           sprintf(work, "%s\\%s", path, _
              finddata.cFileName);
           gl_DirCount += 1;
           DirBrowse(work, filespec,
              hWndList, hWndDirs, hWndFiles);
        }
        else  // it is a file
           gl_FileCount += 1;
     }
     // report stats to textboxes
     sprintf(work, "%i", gl_DirCount);
     Edit_SetText(hWndDirs, work);
     sprintf(work, "%i", gl_FileCount);
     Edit_SetText(hWndFiles, work);
     // get next matching file
     rc = FindNextFile(fSearch, &finddata);
  }
  // we've gotten all files so
  // close search and exit function
  FindClose(fSearch);
  return;
}
```

LISTING 2 ***The Worker Bees.*** *These two functions do all the work for the DLL. They are simple C/C++ functions that perform the functions needed to support the threading in Visual Basic. The DirBrowse While inner loop does most of the work of the directory tree browse by recursively calling DirBrowse. Message Crackers, for sending a message back to Visual Basic, are used to make the code more readable.*

when the worker thread has completed, then the timer and polling function is unnecessary.

Given a specified drive, the thread function DirWalk "walks" the directory tree of the drive recursively. It updates counters in the main application that provide the total count of files and directories searched. I've created a sample program that illustrates this function. It has three buttons: the first, labeled "No Thread" (cmdNoThread), executes the function without creating a thread. The second button executes the function using a thread, and the third button executes the function using a thread offering a button to cancel the thread as it executes. When the DirWalk function is executed with the thread, the user has complete control over the VB application and can do other tasks with the main program as it is executing (such as moving the window). When the dialog with the cancel button is shown, the user can cancel the thread as it is executing.

EVEN VB4'S DEVELOPMENT ENVIRONMENT USES THREADS TO COMPILE IN THE BACKGROUND.

The Visual Basic portion of the program is easy to understand. The main code for creating the thread resides in the "Thread" button's (cmdThread) event code (see Listing 1). Before this code can run, you must load the DLL through the LoadLibrary Win32 API call. LoadLibrary will return a handle to the instance of the loaded DLL (note that in Win32 an instance handle is the same as a module handle). I loaded the DLL with LoadLibrary in the Sub Main of the program and assigned its value to a global variable gh_Library. The constant APP_DLLPATHNAME contains the DOS name of the DLL:

```
Sub Main()
gh_Library = LoadLibrary(APP_DLLPATHNAME)
End Sub
```

After you have the instance handle to the DLL, you can pass that to the function GetProcAddress along with the name of the exported function that is defined in the C/C++ code for the DLL:

```
lAddress = GetProcAddress(gh_Library, _
   "DirBrowseThread")
```

This returns the address in the DLL that is loaded in memory to the start of the function DirBrowseThread. With this address you can now create the thread using CreateThread and give it some other parameters required for execution:

```
Declare Function CreateThread Lib "kernel32" _
   (lpThreadAttributes As Any, _
   ByVal dwStackSize As Long, _
   ByVal lpStartAddress As Long, _
   lpParameter As Any, _
   ByVal dwCreationFlags As Long, _
   lpThreadId As Long) As Long
```

You set the lpThreadAttributes variable that is passed to null, and you set dwStackSize to zero indicating you want simply the default stack size for the thread (1 MB). The lpStartAddress is the lAddress you derived from GetProcAddress. The variable, lpParameter, is a structure that contains the parameters you wish to pass to the thread.

You set dwCreationFlags to zero, and lpThreadID is a Visual Basic Long variable that will return the ID number of the thread. The CreateThread function returns a handle to the thread created or zero in case of failure. Because dwCreationFlags is set to zero (the default), the thread immediately starts executing. If it were set to CREATE_SUSPENDED, the thread would be set up to run but would need to be started using the Win32 API call ResumeThread defined in Visual Basic as:

```
Declare Function ResumeThread Lib "kernel32" _
   (hThread as Long) as Long
```

The DIRDATA user-defined type that I use to pass parameters to the thread is set up before the call to CreateThread. I set this up by making hWndList equal to the handle of the list box and likewise for the edit boxes that contain the folder and file counts. You set sFileSpec to the file specification for the search (in this example, *.* returns all files and folders). Set lFileSpecLen to the number of characters contained in sFileSpec (three). Likewise, you set sPath to the drive selected in the drive selection box and you set lPathLen to two:

```
Public Type DIRDATA
hWndList As Long
hWndDirs As Long
hWndFiles As Long
sFileSpec As String * MAX_FILE_SPEC
lFileSpecLen As Long
sPath As String * MAX_FILE_PATH
lPathLen As Long
End Type
```

I have found that this is the best way to pass multiple parameters to the thread function and mimic other examples that use C/C++ exclusively. (I didn't use variable-length Visual Basic strings or Variants in this example, in an effort to make the C/C++ code clear and concise.)

THE DIRBROWSE FUNCTION GOES TO WORK

The DirBrowse function in the DLL is a simple recursive function that runs through the directory tree of a given drive. The prototype for DirBrowse takes a string for the path and file specification plus three window handles, the first for a list box to add the elements into and the last two for edit boxes to update the count of folders and files. I used edit boxes because Visual Basic does not make the handle to a static text box available:

```
DLLEXPORT void WINAPI DirBrowse(LPSTR path, _
   LPSTR filespec, HWND hWndList, HWND _
   hWndDirs, HWND hWndFiles);
```

The inner loop does most of the work of the DirBrowse functionality (see Listing 2). The Win32 API provides functions that make browsing the directory structure of a drive quite easy. FindFirstFile starts the search by matching the file specification, and returning a handle to the search instance and filling in the find data structure with information about the first file found. From this call, FindNextFile continues finding files with the same file specification continually putting the information in the find data structure. When FindNextFile returns false, there are no more

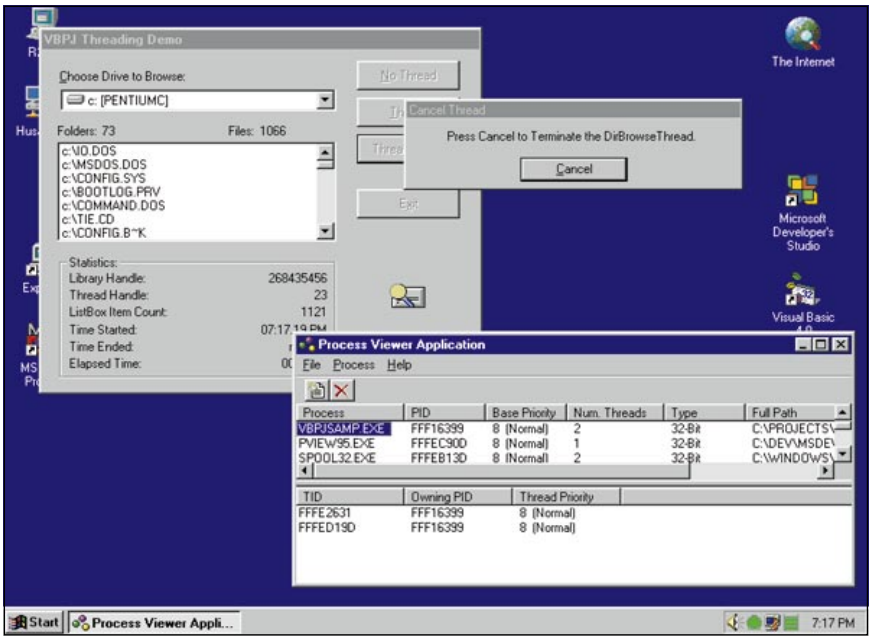

FIGURE 1 ***View Processes in Action.*** *Two threads are listed for VBPJSAMP.EXE when it is run using a thread; when the thread is finished executing the count goes back down to one. Using Pview, you can observe the number of threads executing within a process' space.*

files available and FindClose closes the searching instance.

In the inner loop, the test for a directory is based on the attributes of the found element. The directories "." and ".." are ignored. If the file is a directory, DirBrowse is called again with the same parameters but with a different starting path, the path of the next level of the directory structure. In this way, the function recursively searches the drive directory tree.

The DirBrowse function is called directly by the button labeled "No Thread" (cmdNoThread). When called this way it is not created on a thread and is called in the traditional sense in Visual Basic using a Declare statement:

```
Declare Sub DirBrowse Lib "thread.dll" _
   (ByVal path As String, ByVal filespec _
   As String, hWndList As Long, hWndDirs _
   As Long, hWndFiles As Long)
```

Notice that when the DirBrowse function is called directly through Visual Basic, the interface of the program is unresponsive. You cannot move the form, the list box is not refreshed as items are added into it, and the folder and directory counts are not updated as the function is being run.

When the DLL is called from Visual Basic using the GetProcAddress method and CreateThread, the function DirBrowseThread is used to set up the parameters from the DIRDATA structure to call the DirBrowse function. The DirBrowseThread function also calls the Win32 API call ExitThread to signal to the operating system (and Visual Basic) that the thread has completed its task and can be destroyed. Note that using ExitThread on a process' main thread effectively ends the application. The only parameter that ExitThread takes is the return code of the thread. For this function this return code will always be ERROR_SUCCESS.

There are a couple ways you can pass data between the DLL and Visual Basic. Passing handles through the function parameters is the easiest way in this case and for the purposes of updating the interface for the user. When using this method, Visual Basic can watch for change events and use them to synchronize interface elements as the thread is running.

For example, the counting of the list box items in the statistic frame (even though on a timer) is updating as items are being added to it from the DLL. Another way Visual Basic can get data from the DLL functions is somewhat similar to the Win32 API. Visual Basic can call another exported function in the DLL to simply retrieve the information that the thread has collected. Think of this as the typical MoveNext type method used when Visual Basic accesses data in a database using the data access objects. The database effectively retrieves (or has ready) all elements the query has requested, but the programmer must iterate through the items to retrieve all the results. This is also similar to some of the Win32 API functions such as RegEnumKeyEx, which returns one key after another until no more are available.

When the thread is running, a timer is running in Visual Basic that makes the Win32 API call GetExitCodeThread, passing the handle of the active thread to the function. It will return either STILL_ACTIVE or the actual return code of the thread (in this case, this is ERROR_SUCCESS). If the thread is not active, the code updates the statistics and stops the timer. Otherwise the statistics are updated and the timer continues. This way, Visual Basic will know when the thread has completed executing. To cancel a thread while it is executing, the Win32 API call TerminateThread cancels a thread given a handle to a thread and an exit code. With this call, you can have a cancel dialog as demonstrated by the button labeled Thread Dialog (cmdThreadDlg). Using PView, which comes with Visual C++, you can observe the VBPJSAMP.EXE thread process in action (see Figure 1).

This technique works well with Visual Basic 4.0 in Windows 95 and Windows NT. A good reference for learning about threads and how they work is *Advanced Windows: The Developer's Guide to the Win32 API for Windows NT 3.5 and Windows 95,* by Jeffrey Richter (Microsoft Press, ISBN: 1-55615-677-4). This book is an excellent source of information on threads, processes, thread synchronization techniques, and other advanced Win32 topics. Other sources of information are the Microsoft Win32 SDK and the Microsoft Developer Network. Experimenting with the techniques introduced here in your own code can help you find effective ways to use threads. ☒

WHERE DATA INTEGRITY IS ESSENTIAL,
PLAY CLOSE ATTENTION TO HOW YOU USE
MULTIPLE THREADS.