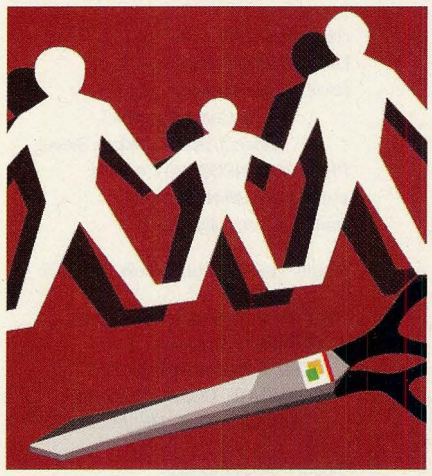
By Douglas A. Hamilton

All My Children

NT OFFERS RICH functionality for creating child processes; they can run as asynchronous activities, be connected together with pipes and can react to exception events such as interrupts. But with all this functionality comes the basic problem of trying to code any of it. In this column, we'll take a look at a couple of very simple examples of, first, just spawning a child and, second, setting up a pipe between the two processes.



SCOTT BALDWIN

Spawn.c

Creating a child process is probably simpler than you imagine. Here's a simple but complete program, spawn.c, to run an arbitrary command.

If you type "spawn notepad hello.c," it will start the notepad, ready to edit hello.c.

Spawn works by retrieving the command-line string, stripping off the first word (holding "spawn") and passing the rest to CreateProcess as the filename to be executed along with any arguments.

Figure 1:

```
#include <windows.h>
#include <stdio.h>
void cdecl main( void )
    {
    char *c = GetCommandLine();
    while (*c && *c != ' ')
        ;
    if (*c++)
        {
        PROCESS_INFORMATION child;
        static STARTUPINFO startup;
        startup.cb = sizeof(startup);
        CreateProcess(NULL, c, NULL,
                NULL,
                TRUE /* Inherit handles */,
                0, NULL, NULL, &startup,
                &child);
        WaitForSingleObject(child.hProcess,
                INFINITE);
        }
    else
        printf("usage: spawn command\n");
    ExitProcess(0);
    }
```

I've used the GetCommandLine call rather than the more traditional main function parameter argv, for two reasons. GetCommandLine is the underlying operating system call that the C start-up routines use to construct argc and argv. So this is a little faster just for that reason alone. But also, it was just a little more convenient in this situation, particularly since it had all the words already concatenated with spaces between them. (Why break it up into separate words just to paste them back together?)

Notice that there's no explicit provision here for tediously searching through the PATH directories nor is any consideration given to what kind of command is

being started. That's deliberate. CreateProcess is smart enough to know all about the search path and do the right thing no matter whether the child is a Win32, Win 3.*x*, DOS, POSIX or OS/2 application. If you don't specify an extension, it knows to try .EXE automatically. It even knows how to run .CMD files by starting up a copy of CMD.EXE. (But you do have to give the .CMD extension explicitly if that's what you mean.)

The first parameter to CreateProcess is used to specify the full pathname for the executable file. That can be useful if you don't want CreateProcess to automatically search the PATH directories. If you do know the full pathname, it is faster than letting CreateProcess exhaustively try every possibility.

The third and fourth parameters to CreateProcess let you attach security descriptors to the child process and its first thread. That's clearly an advanced topic and not something most of us need to worry about.

The fifth parameter specifies whether open inheritable handles like file handles, semaphores or other processes are to be inherited. Most child processes will need to inherit at least stdin; therefore, this parameter should be set to TRUE.

The sixth parameter determines whether the child is to be suspended, run under the control of a debugger or run with a different priority. The seventh can specify a unique set of environment variables. The eighth parameter sets the current directory for the start-up of the child process.

The STARTUPINFO structure passed as the ninth parameter lets you customize the size, shape or position of the window in which the child will run. Most of the time, CreateProcess will produce an acceptable window without the need to adjust all the fields in this structure. Only the size field need be specified.

Finally, the PROCESS_INFORMATION structure returns handles and ID numbers for the child process and its first thread. The process handle is especially important; we can wait on it indefinitely to find out when the child completes.

Pipe.c

Here's a more complex case, creating a pipe to our child's stdout. We'll read everything it writes into the pipe, copying it to our own stdout and then print the number of characters read.

In this example, we open a pipe, create an inheritable copy of the input handle and pass it to the child.

Figure 2:

```
#include <windows.h>
#include <stdio.h>
void cdecl main( void )
   {
   char *c = GetCommandLine();
   while (*c && *c != ' ')
      c++;
   if (*c++)
      {
      char buffer[4096];
      DWORD length, total = 0;
      HANDLE thisProcess =
                  GetCurrentProcess(),
               pipeout, pipein, childout, Stdout;
      PROCESS_INFORMATION child;
      static STARTUPINFO startup;
      startup.cb = sizeof(startup);

      CreatePipe(&pipeout, &pipein, NULL, 0);

      DuplicateHandle(thisProcess, pipein,
         thisProcess, &childout,
         NULL, TRUE /* Inheritable */,
         DUPLICATE_CLOSE_SOURCE |
            DUPLICATE_SAME_ACCESS);

      Stdout = GetStdHandle(
            STD_OUTPUT_HANDLE);
      SetStdHandle(
            STD_OUTUT_HANDLE,childout);
      CreateProcess(NULL, c, NULL, NULL,
            TRUE /* Inherit handles */,
            0, NULL, NULL, &startup, &child);
      CloseHandle(childout);

      while (ReadFile(pipeout, buffer,
            sizeof(buffer), &length, NULL))
         {
         total += length;
         WriteFile(Stdout, buffer, length,
            &length, NULL);
         }

      CloseHandle(pipeout);
      printf("total read = %d bytes\n", total);

      WaitForSingleObject(child.hProcess,
            INFINITE);
      }
   else
      printf("usage: pipe command\n");
   ExitProcess(0);
   }
```

CreatePipe is extremely simple: You pass it pointers to where it should store the read and write handles of the pipe along with an optional security descriptor and an optional advisory buffer size.

The handles that come back are not inheritable, however. So if we expect our child to be able to write anything into the pipe, we first have to create a duplicate handle that is inheritable. (Those who have worked with OS/2 will notice the difference here; under OS/2 the inheritance of a given handle could change on the fly. Not so under NT; it's set when the handle is created.)

DuplicateHandle is a general-purpose handle duplicator. It will duplicate any type of handle and can even duplicate handles between any two processes.

In this simple case, we're just duplicating the handle within our own process, closing the original and making the copy inheritable.

The next step is to save off the initial value of stdout and set stdout to be that input end of the pipe.

After creating the child process, we close our own handle to that input end. This is important because end-of-file on a pipe isn't detected until the last handle to the input end is closed.

The remaining step is to loop and read whatever output the child has produced and copy it to stdout. But one interesting point to notice is that even though we had used SetStdHandle to reset stdout before creating the child, that did not affect the handle we'd previously retrieved. Even printf works, because as soon as you link in printf, you also get some C runtime library initialization routines that get called when your program starts. They do the same thing we just did: They retrieve the initial setting for stdout and save it for future use.

Try typing in these examples and compiling them. For me, it's one thing to be told something works and another to find out for myself. I tested these examples on the July SDK; they should still work on the upcoming beta. ■

Douglas Hamilton is president of Hamilton Laboratories (Wayland, Mass.) and author of the Hamilton C Shell, an advanced interactive command processor and tools package for OS/2 and Windows NT. Reach Douglas on WIX as hamilton or care of Editor at the address on page 12.