Be careful when redirecting both a process's stdin and stdout to pipes, for you can easily deadlock

devblogs.microsoft.com/oldnewthing/20110707-00

July 7, 2011

Raymond Chen

A common problem when people create a process and redirect both stdin and stdout to pipes is that they fail to keep the pipes flowing. Once a pipe clogs, the disturbance propagates backward until everything clogs up.

Here is a common error, in pseudocode:

```
// Create pipes for stdin and stdout
CreatePipe(&hReadStdin, &hWriteStdin, NULL, 0);
CreatePipe(&hReadStdout, &hWriteStdout, NULL, 0);

// hook them up to the process we're about to create
startup_info.hStdOutput = hWriteStdout;
startup_info.hStdInput = hReadStdin;

// create the process
CreateProcess(...);

// write something to the process's stdin
WriteFile(hWriteStdin, ...);

// read the result from the process's stdout
ReadFile(hReadStdout, ...);
```

You see code like this all over the place. I want to generate some input to a program and capture the output, so I pump the input as the process's stdin and read the output from the process's stdout. What could possibly go wrong?

This problem is well-known to unix programmers, but it seems that the knowledge hasn't migrated to Win32 programmers. (Or .NET programmers, who also encounter this problem.)

Recall how anonymous pipes work. (Actually, this description is oversimplified, but it gets the point across.) A pipe is a marketplace for a single commodity: Bytes in the pipe. If there is somebody selling bytes (`WriteFile`), the seller waits until there is a buyer (`ReadFile`). If there is somebody looking to buy bytes, then the buyer waits until there is a seller.

In other words, when somebody writes to a pipe, the call to `WriteFile` waits until somebody issues a `ReadFile`. Conversely, when somebody reads from a pipe, the call to `ReadFile` waits until somebody calls `WriteFile`. When there is a matching read and write, the bytes are transferred from the writer's buffer to the reader's buffer. If the reader asks for fewer bytes than the writer provided, then the writer continues waiting until all the bytes have been read. (On the other hand, if the writer provides fewer bytes than the reader requested, the reader is given a partial read. Yes, there's asymmetry there.)

Okay, so where's the deadlock in the above code fragment? We write some data into one pipe (connected to a process's stdin) and then read from another pipe (connected to a process's stdout). For example, the program might take some input, do some transformation on it, and print the result to stdout. Consider:

You	Helper
`WriteFile(stdin, "AB")`	
(waits for reader)	
	`ReadFile(stdin, ch)`
	reads `A`
(still waiting since not all data read)	
	encounters errors
	`WriteFile(stdout, "Error: Widget unavailable\r\n")`
	(waits for reader)

And now we're deadlocked. Your process is waiting for the helper process to finish reading all the data you wrote (specifically, waiting for it to read `B`), and the helper process is waiting for your process to finish reading the data it wrote to its stdout (specifically, waiting for you to read the error message).

There's a feature of pipes that can mask this problem for a long time: Buffering.

The pipe manager might decide that when somebody offers some bytes for sale, instead of making the writer wait for a reader to arrive, the pipe manager will be a market-maker and *buy the bytes himself*. The writer is then unblocked and permitted to continue execution. Meanwhile, when a reader finally arrives, the request is satisfied from the stash of bytes the pipe manager had previously bought. (But the pipe manager doesn't take a 10% cut.)

Therefore, the error case above happens to work, because the buffering has masked the problem:

You	Helper
`WriteFile(stdin, "AB")`	
pipe manager accepts the write	
`ReadFile(stdout, result)`	
(waits for read)	
	`ReadFile(stdin, ch)`
	reads `A`
	encounters errors
	`WriteFile(stdout, "Error: Widget unavailable\r\n")`
Read completes	

As long as the amount of unread data in the pipe is within the budget of the pipe manager, the deadlock is temporarily avoided. Of course, that just means it will show up later under harder-to-debug situations. (For example, if the program you are driving prints a prompt for each line of input, then the problem won't show up until you give the program a large input data set: For small data sets, all the prompts will fit in the pipe buffer, but once you hit the magic number, the program hangs because the pipe is waiting for you to drain all those prompts.)

To avoid this problem, your program needs to keep reading from stdout while it's writing to stdin, so that neither will block the other. The easiest way to do this is to perform the two operations on separate threads.

Next time, another common problem with pipes.

Exercise: A customer reported that this function would sometimes hang waiting for the process to exit. Discuss.

```
int RunCommand(string command, string commandParams)
{
 var info = new ProcessStartInfo(command, commandParams);
 info.UseShellExecute = false;
 info.RedirectStandardOutput = true;
 info.RedirectStandardError = true;
 var process = Process.Start(info);
 while (!process.HasExited) Thread.Sleep(1000);
 return process.ExitCode;
}
```

Exercise: Based on your answer to the previous exercise, the customer responds, "I added the following code, but the problem persists." Discuss.

```
int RunCommand(string command, string commandParams)
{
 var info = new ProcessStartInfo(command, commandParams);
 info.UseShellExecute = false;
 info.RedirectStandardOutput = true;
 info.RedirectStandardError = true;
 var process = Process.Start(info);
 var reader = Process.StandardOutput;
 var results = new StringBuilder();
 string lineOut;
 while ((lineOut = reader.ReadLine()) != null) {
  results.AppendLine("STDOUT: " + lineOut);
 }
 reader = Process.StandardError;
 while ((lineOut = reader.ReadLine()) != null) {
  results.AppendLine("STDERR: " + lineOut);
 }
 while (!process.HasExited) Thread.Sleep(1000);
 return process.ExitCode;
}
```

Raymond Chen

Follow