We've traced the pipe, and it's coming from inside the process!

devblogs.microsoft.com/oldnewthing/20110708-00

July 8, 2011

Raymond Chen

We saw last time one of the deadlocks you can run into when playing with pipes. Today we'll look at another one:

> Our program runs a helper process with stdin/stdout/stderr redirected. The helper process takes input via stdin and prints the result to stdout. Sometimes we find that the `WriteFile` from the controlling process into the stdin pipe hangs. Closer examination reveals that the helper process no longer exists. Under these conditions, should the `WriteFile` fail, since the reader is no longer available?

If you attempt to write to a pipe when there is nobody around to call `ReadFile` to read the data out the other end, the call to `WriteFile` should fail with the error `ERROR_BROKEN_PIPE` (known in Unix-land as `EPIPE`). What does it mean when the write pends? It means that there is still somebody around who can read the data out of the pipe, but the internal pipe buffer is full, so the write call waits for the reader to drain the data.

But the helper process no longer exists. Maybe it crashed or exited prematurely. That means that there is nobody around to read the data out of the pipe. Why, then, does the call not return immediately with an error?

Because there is still somebody around to read the data out of the pipe.

> Did you remember to close the controlling process's copy of the read end of the pipe?

If the controlling process hasn't closed its copy of the read end of the pipe, then the pipe is correct in believing that there is still somebody around to read the data out of the pipe, namely *you*. You have a handle to the read end of the pipe, so the pipe manager cannot declare the pipe dead; for all it knows, you intended for the controlling process to call `ReadFile` to read the data out of the pipe. As far as the pipe is concerned, you simply haven't gotten around to it yet, so the pipe waits patiently.

> Yes, our code calls `CloseHandle` on the controlling process's copy of the pipe handles. I've highlighted it below. (Error checking has been elided for simplicity.)
>
> ```
> // create the pipe for stdout/stderr
> CreatePipe(&hReadPipeTmp, &hWritePipeTmp, NULL, 0);
> // duplicate the handles with bInheritHandle=FALSE to prevent
> // them from being inherited
> DuplicateHandle(GetCurrentProcess(), hWritePipeTmp,
>                 GetCurrentProcess(), &hWritePipe,
>                 0, FALSE, DUPLICATE_SAME_ACCESS);
> DuplicateHandle(GetCurrentProcess(), hReadPipeTmp,
>                 GetCurrentProcess(), &hReadPipe,
>                 0, FALSE, DUPLICATE_SAME_ACCESS);
> // create the pipe for stdin
> CreatePipe(&hHelperReadPipe, &hHelperWritePipe,
>            NULL, 0);
> // disable inheritance on on the write end of the stdin pipe
> SetHandleInformation(hHelperWritePipe, HANDLE_FLAG_INHERIT, 0);
> // prepare to create the process
> ... blah blah blah other stuff unrelated to handles ...
> startupInfo.hStdInput = hHelperReadPipe;
> startupInfo.hStdOutput = hWritePipeTmp;
> startupInfo.hStdError = hWritePipeTmp;
> CreateProcess(...);
> // Here is where we close the handles
> CloseHandle(hReadPipeTmp);
> CloseHandle(hWritePipeTmp);
> // Write the input to the helper process (hangs here sometimes)
> WriteFile(hHelperWritePipe, ...);
> ```

This is another case of getting so excited about doing something that you forget to do it. (Notice how the comments to that article very quickly descend into a discussion of command line quotation marks.)

Observe that the handles being closed are `hReadPipeTmp` and `hWritePipeTmp`, which is a good thing to do, but neither has any effect on the `WriteFile`. The `WriteFile` is writing to `hHelperWritePipe` and therefore the handle you need to close is `hHelperReadPipe`. Since that handle is still open in the controlling process, the pipe manager will not break the pipe, because it's waiting for you to read from it.

Raymond Chen

Follow