Inter-Process Communication

First, remember what a process is. A "process" is a running instance of a program (where a "program" is an executable file stored in your file system). We make a distinction between a "program" and a "process" because a program, like Notepad, can have several instances of itself running at any given time. Each such instance is a different process (but each instance can be said to be the same program). Notice that a process is to a program much like an object is to a class.

In Operating Systems you learn that each process runs in its own virtual memory space. Virtual memory spaces were invented so that each process is purposely walled off from every other process. You do not want a bug in one process (say an instance of Word) to cause the crash of another process (say an instance of Firefox). You also do not want one process (say an instance of Firefox) to be able to look at, and read, the memory of another process (say Word) and report back over the Internet what you are typing in your Word documents. So for stability and security reasons, processes are strictly isolated from each other (each in its own virtual memory space).

But if processes are strictly isolated from each other, they cannot communicate, or cooperate, with each other. That is too severe a restriction (think of how useful copy and paste is, among many other ways in which programs communicate with each other). So operating systems provide very tightly controlled mechanisms for processes to communicate (and therefore cooperate) with each other. These mechanisms are referred to as Inter-Process Communication.

"Inter-Process Communication" (IPC) is when one process passes a piece of data (a message) to another process. ALL forms of IPC involve the operating system kernel acting as an intermediary in the passing of the message (in order to guarantee the stability and security of the whole system). In fact, every form of IPC involves the kernel copying a segment of data (a buffer) from the virtual memory space of the sending process into the virtual memory space of the receiving process.

There are six simple mechanisms for IPC (and many complicated ones). The simple IPC mechanisms are,

1. files
2. command-line arguments
3. environment variables
4. program exit code (return value)
5. I/O redirection
6. pipes

Files can be used by any two processes to communicate with each other (more specifically, any two processes that have access to a shared file system). In this sense, files are a very general form of IPC (the messages can be as large as any file, the messages can be passed in both directions, and the communicating processes need not even be running on the same computer or at the same time), but it is a very slow form of IPC.

Command-line arguments can only be used by a parent process to pass (in a one-way direction) a message to a child process (a child process is a process that is started by the parent process). Also, this message must be fairly small and it can only be sent at the time the child process is created. So command-line arguments are a fairly restrictive form of IPC (but they get used a lot).

Environment variables are similar to command-line arguments. They are fairly small messages, they can only be passed in one direction (from parent to descendant processes) and the messages must be sent at the time a child process is created. The most significant difference between environment variables and command-line arguments is that an environment variable is also inherited by every descendant of the original child process. So an environment variable can be used to send a message to every descendant of the parent process. There are times when this feature can be useful.

A program's exit code is a single integer value that can be used by a child process to send a (very small) message back to its parent process. This form of communication can only take place when the child terminates. Exit codes are used mostly as error codes that let the parent know something about the success or failure of a child process.

I/O redirection can be used by a parent to communicate with a child while the child is running (not just when the child starts up). When the parent creates the child, the parent can have the operating system redirect one of the parent's output streams to the child's standard input stream, and then redirect the child's standard output stream to one of the parent's input streams. The amount of data that can be communicated this way is unlimited and the data can be communicated in both directions, from parent to child and from child to parent. This is a very common and useful form of IPC.

Pipes are a way for two sibling processes to communicate. Two processes are siblings if they were created by the same parent processes. A pipe must be created by the parent process and it must be created at the time the two child processes are created. But otherwise, pipes are a very general form of communication. The most common example of a parent process creating a pipe between two child processes is when we use a command-line shell to run two programs and we connect the two programs on the command-line with the pipe ('|') symbol. The shell process is the parent process and the two programs on the command-line become the child processes. The shell process has the operating system create a pipe object, then redirect the standard output of the first child process (the child that can send messages) to the input of the pipe, then redirect the standard input of the second child process (the child that will receive messages) to the output from the pipe. We say that pipes are a form of "stream" communication, in the sense that once the pipe is created, the two child processes can pass an unlimited stream of bytes through the (one-directional) pipe. Pipes are very fast (almost as fast as writing and reading from physical memory). The biggest restriction on pipes is that the two processes must be running on the same computer.

Two important advanced forms of IPC are sockets and shared memory.

Sockets are a way of providing stream communication between any two processes running on the same or different computers (but connected together by a network). Sockets are straightforward to use, they are very versatile, fairly fast, and they are now ubiquitous (these days, every language running on every operating system provides sockets).

Shared memory is when the operating system arranges for two processes (running on the same computer) to share a physical page frame in their virtual memory page tables. This means that the two processes can both read and write to the physical page frame. This is a fast, bidirectional, symmetrical, form of IPC but it causes many of the same problems that shared memory causes with threads (race conditions). Shared memory is the basis for many other forms of IPC.

NOTE: If you want to see the parent/child relationship of all the processes running on your computer, run either the ProcessExplorer.exe or ProcessHacker.exe programs that are in the folder with this file (for ProcessExplorer.exe, be sure to select the menu item "View -> Show Process Tree").

1.) Creating Processes

Before talking about "inter-process communication", we should first see how one process (i.e., a running program) can create another process. The first process will be called the parent process, and the second process, the one created by the first process, will be called the child process. Our goal will be for the two processes, the parent and its child, to communicate with each other.

In the sub-folders there are Java, Windows, and Linux programs that create processes. Compile and run these programs. They show how a Java or C program can start up other programs. Notice that some of these child processes are given command-line parameters when they are started. For example, on Windows the Notepad.exe editor is started and it is told to open a file. Also, the Windows Explorer.exe program is told to open the folder "C:\Windows\System32". This demonstrates a simple example of a parent process starting up a child processes and communicating to the child processes a bit of (inter-process) information.

To see the parent/child relationships created by the examples on Windows, run either the ProcessExplorer.exe or ProcessHacker.exe programs (for ProcessExplorer.exe, be sure to select the menu item "View -> Show Process Tree").

2.) Simple Inter-process Communication

Two of the simplest forms of Inter-process Communication are "command-line arguments" and "environment variables".

Command-line arguments are passed to a process when it is created (i.e., started up). It is important to realize that ALL running programs have command-line arguments (not just programs run from a "command-line"). Here is one way to prove that every running program has a "command-line" and also "command-line arguments".

Run Window's "Task Manager" program. There are several ways to run this program.
1.) Hold down the Ctrl and Shift keys and then strike the Esc key.
2.) Click on the Windows Start menu, click on the "Run..." item, type in the program name "taskmgr.exe", and click on OK.
3.) Go to the folder "C:\Windows\System32", find the file taskmgr.exe and double click on it.

When you have Task Manager running, click on the "View->Select Columns..." menu item. Scroll down to the bottom of the pop-up window. You should see a small box next to an item called "Command Line". Click on the box to put a check-mark in the box. Click on the OK button. In the Task Manager window you should now see a list of all the processes you are running and there should be a "Command Line" column for each process. This lets you see the actual command-line used to start each of those processes.

If you start a GUI program by double-clicking on a "shortcut icon" (for example, an item from Window's Start Menu), the command-line for the process is stored in the icon. Right click on a shortcut icon and click on the bottom item from the pop-up menu (the "Properties" item). You should see a tabbed pop-up window with a tab called "Shortcut". In that tab is a textbox labeled "Target" and in that textbox is the process's command line along with any command-line arguments. Since this is a textbox, you can modify the command-line and its arguments to change what the shortcut does when you double-click on it.

Environment variables are similar to command-line arguments in that they are fairly small messages, they can only be passed in one direction (from parent to descendant processes) and the messages must be sent at the time the child process is created. Just as ALL processes have command-line arguments, so do ALL processes have environment variables.

You can use the ProcessExplorer.exe and ProcessHacker.exe programs to observe the environment variables for each running process. Start up either ProcessExplorer.exe or ProcessHacker.exe. Right click on any process and choose the "Properties" item from the pop-up menu. You should see a tabbed pop-up window with one tab labeled "Environment". Click on that tab, and you will see a list of all of the environment variables that the process inherited from its parent process. (On a Windows computer, most of the processes that you look at will have pretty much the exact same environment variables. Windows programs do not usually make much use of environment variables (other than the standard ones set by the operating system). On Unix/Linux computers, lots of programs make use of lots of environment variables.)

Notice that when RunProgram.java starts up the Java program
```
   CommandLineArguments.class
```
(which you need to compile before it can be run) there is no output on the console window. But if you run the program CommandLineArguments from a command line, it always produces output in the console window. This shows that there is a problem with using Java to run programs that use a console window (this

problem does not exist when C or C++ programs run console window programs). In the following folders, we will explore this problem with Java and see how to solve it (Java does not handle "standard input" and "standard output" in the correct way).

4.) I/O Redirection, Filters, and Pipelines

When a process is created by the operating system, the
process is always supplied with three open streams.
These three streams are called the "standard streams".
They are

```
   standard input  (stdin)
   standard output (stdout)
   standard error  (stderr)
```

We can visualize a process as an object with three "connections"
where data (bytes) can either flow into the process or flow out
from the process.

```
                    process
              +-----------------+
              |                 |
          ---->>stdin    stdout>>------
              |                 |
              |          stderr>>------
              |                 |
              +-----------------+
```

A console application will usually have its stdin stream connected
to the computer's keyboard and its stdout and stderr streams connected
to the console window.

```
                    process
              +-----------------+
              |                 |
  keyboard --->>stdin    stdout>>------+---> console window
              |                 |      |
              |          stderr>>------+
              |                 |
              +-----------------+
```

It is important to realize that the above picture is independent of
the programming language used to write the program which is running
in the process. Every process looks like this. It is up to each
programming language to allow programs, written in that language, to
make use of this setup provided by the operating system.

Every operating system has its own way of giving a process access to
the internal data structures the operating system uses to keep track
of what each standard stream is "connected" to.

The Linux operating system gives every process three "file descriptors",

```
    #define  STDIN_FILENO 0,  STDOUT_FILENO 1,  STDERR_FILENO 2
```

Linux provides the read() and write() system calls to let a process
read from and write to these file descriptors.

The Windows operating system gives every process three "handles",
You retrieve the handles using the GetStdHandle() function with
one of these input parameters.

```
     STD_INPUT_HANDLE, STD_OUTPUT_HANDLE, STD_ERROR_HANDLE
```

Windows provides the ReadFile() and WriteFile() system calls to
let a process read from and write to these handles.

Every programming language must have a way of representing the three
standard streams and every language must provide a way to read from
the standard input stream and a way to write to the standard output
and standard error streams.

For example, here is how the three standard streams are represented by some common languages.

```
  Standard C uses file objects
    FILE* stdin, stdout, stderr;

  C++ uses stream objects
    ios  std::cin, std::cout, std::cerr;

  Java uses stream objects
    InputStream System.in; PrintStream System.out, System.err;

  .net uses stream objects
    System.Console.In, System.Console.Out, System.Console.Error
```

The C language provides functions like getchar(), scanf(), and fscanf() to read from stdin and it provides printf() and fprintf() to write to stdout and stderr. On a Windows computer, C's printf() function will be implemented using the WriteFile() system call with the STD_OUTPUT_HANDLE handle. On a Linux computer, C's printf() function will be implemented using the write() system call with the STDOUT_FILENO file descriptor.

Every process is created by the operating system at the request of some other process (the parent process). When the parent process asks the operating system to create a child process, the parent tells the operating system how to "connect" the child's three standard streams.

At a shell command prompt, if you type a command like this,

```
C:\> foo > result.txt
```

the shell program (cmd.exe on Windows, or bash on Linux) is the parent process. The above command tells the shell process to ask the operating system to create a child process from the foo program. But in addition to asking the operating system to create the child process, the shell process also instructs the operating system to redirect the child process's standard output to the file result.txt. So when foo runs, it looks like this.

```
                 foo process
              +------------------+
              |                  |
  keyboard --->>stdin    stdout>>----> result.txt
              |                  |
              |          stderr>>----> console window
              |                  |
              +------------------+
```

Stdin and stderr have their default connections, and stdout is redirected to the file result.txt.

If you type a command like this,

```
C:\> foo > result.txt < data.txt
```

the shell process will ask the operating system to create a child process from the foo program and also ask the operating system to redirect the child process's standard output to the file result.txt and redirect the child process's standard input to the file data.txt. So when foo runs, it looks like this.

```
                 foo process
              +------------------+
              |                  |
  data.txt --->>stdin    stdout>>----> result.txt
              |                  |
              |          stderr>>----> console window
              |                  |
              +------------------+
```

If you type a command like this,

```
C:\> foo < data.txt | bar > result.txt
```

the shell process will ask the operating system to create two child processes, one from the foo program and the other from the bar program. In addition, the shell process will ask the operating system to create a "pipe" object and have the stdout of the foo process redirected to the input of the pipe, and have the stdin of the bar process redirected to the output of the pipe. Finally, the shell process will ask the operating system to redirect the bar process's standard output to the file result.txt and redirect the foo process's standard input to the file data.txt. So while this command is executing, it looks like this.

```
              foo process                  bar process
           +---------------+            +---------------+
           |               |    pipe    |               |
 data.txt-->>stdin  stdout>>--========-->>stdin  stdout>>----> result.txt
           |               |            |               |
           |        stderr>>---+        |        stderr>>--+-> console window
           |               |   |        |               | |
           +---------------+   |        +---------------+ |
                               |                          |
                               +--------------------------+
```

5.) A Process Redirecting Its Own Standard Streams

The examples in this folder show how a process can redirect its
own standard streams to files. By this we mean that a process might
be running with the following configuration of its standard streams

```
                   process
             +------------------+
             |                  |
  keyboard--->>stdin     stdout>>-----+--> console window
             |                  |     |
             |           stderr>>-----+
             |                  |
             +------------------+
```

We want to see how the process might, while it is running, change its
standard streams so that they look like the following picture, with
stdin and stdout connected to specific files chosen by the process.

```
                   process
             +------------------+
             |                  |
  data.txt--->>stdin     stdout>>------> result.txt
             |                  |
             |           stderr>>-----> console window
             |                  |
             +------------------+
```

This does not, by itself, seem like a very useful thing to be able
to do. It is almost exactly the same idea as just opening new file
streams for reading or writing (as in the picture below). The main
difference between the above picture and the picture below is that
the process above can use the scanf() and printf() functions to read
and write the files data.txt and result.txt while the process below
would have to use the fscanf() and fprintf() functions to read and
write to data.txt and result.txt.

```
                   process
             +-------------------+
             |                   |
  keyboard--->>stdin      stdout>>-----+---> console window
             |                   |     |
             |            stderr>>-----+
             |                   |
  data.txt--->>stream3   stream4>>-----> result.txt
             |                   |
             +-------------------+
```

But seeing how a process can redirect its standard streams is a good
starting point for understanding the techniques that are used for IPC.

An important reason for a process to redirect its own stdin and stdout
streams is so that it can have a child process "inherit" those redirected
streams. We will discuss a child process "inheriting" its parent's standard
streams in the next two folders.

5.) A Windows Process Redirecting Its Own Standard Streams

As mentioned in the previous folder, Windows uses "handles" as the way for processes to identify resources given to them by the operating system. In particular, since every process is created with three standard I/O streams, every Windows process will need to be given three "standard handles" to those standard streams. The Windows function
      SetStdHandle()
lets a Windows process change the stream that one of those "standard handles" refers to. So if a Windows process should open a new I/O stream (by calling the CreateFile() function), then the process can use SetStdHandle() to substitute the new stream for one of its original standard streams.

The RedirectStandardStreams.c file demonstrates using the SetStdhandle() function. That program calls CreateFile() two times to open a new input and a new output stream and then is calls SetStdHandle() twice to redirect stdin and stdout to those two new streams.

Here are a sequence of picture illustrating the sequence of streams as the RedirectStandardStreams.exe process runs.

Suppose that initially stdin is the keyboard and stdin and stderr are the console window.

```
                    process
              +-----------------+
              |                 |
   keyboard--->>stdin    stdout>>---------> console window
              |                 |
              |          stderr>>------> console window
              |                 |
              +-----------------+
```

The process first calls CloseHandle() to close the stdin and stdout file handles, since those two files will no longer be used.

```
                    process
              +-----------------+
              |                 |
           --->>stdin    stdout>>-->
              |                 |
              |          stderr>>------> console window
              |                 |
              +-----------------+
```

The process then calls CreateFile() to open an input stream (using a temporary input handle).

```
                    process
              +------------------+
              |                  |
           --->>stdin     stdout>>-->
              |                  |
              |           stderr>>------> console window
              |                  |
 Readme.txt--->>hIn              |
              |                  |
              +------------------+
```

Then the process calls CreateFile() again to open an output stream (using a temporary output handle).

```
                    process
              +------------------+
              |                  |
           --->>stdin     stdout>>-->
              |                  |
              |           stderr>>------> console window
              |                  |
 Readme.txt--->>hIn         hOut>>-----> result.txt
              |                   |
              +-------------------+
```

Now the process calls SetStdHandle() two times to redirect stdin and
stdout to the new file streams.

```
                    process
              +-------------------+
              |                   |
 Readme.txt-+->>stdin     stdout>>--+-----> result.txt
            | |                   | |
            | |           stderr>>------> console window
            | |                   | |
            +->>hIn         hOut>>--+
              |                   |
              +-------------------+
```

At this point, the two temporary handles are no longer needed and can be
set to INVALID_HANDLE_VALUE.

```
                    process
              +-------------------+
              |                   |
 Readme.txt--->>stdin     stdout>>--------> result.txt
              |                   |
              |           stderr>>------> console window
              |                   |
              |                   |
              +-------------------+
```

The I/O redirection is complete and the process can treat the two new files
streams as its stdin and stdout.

At this point the RedirectStandardStreams.c program acts as a simple echo
filter between stdin and stdout. But notice that the program uses the low
level Windows I/O functions ReadFile() and WriteFile(). Why not use the
higher level C functions getchar() and printf()? The answer is that they
no longer work the way you would expect them to. Remember that Standard C
functions do not use Window's handles. Instead, Standard C functions use
pointers to FILE objects. The I/O redirection done by the SetStdHandle()
function does NOT redirect the C Library's standard FILE objects (as you
would expect it to). The file
   RedirectStandardStreams_broken.c
demonstrates this by using the C functions getchar() and printf() after the
calls to SetStdHandle(). These two functions still work on the original
stdin and stdout streams! They have not been redirected.

The file
   RedirectStandardStreams_kind_of_broken.c
demonstrates a kind of fix for this situation. It shows how to re-sync the
C Library's standard FILE objects with the Windows standard handles. The
code used in this program "works" but it is not correct since it works by
breaking the FILE object interface (the program writes to a FILE object
pointer, which is something that should never be done).

In this folder there are also the two files
   EchoFromFileToFile.c
   Echo_weird.c
These two programs are there mostly to make you think about the difference
between these three command lines (which, more or less, do the same thing).

```
   C:\> RedirectStandardStreams.exe
   C:\> EchoFromFileToFile.exe
   C:\> Echo_weird.exe < Readme.txt > result.txt
```

6.) A Child Inheriting (or Sharing) Streams With Its Parent

The examples in this folder (and the next) show how a parent process can control the standard streams (stdin, stdout, stderr) of a child process.

There are two ways that the parent can control what a child's standard streams are connected to when the child is created. The parent can either have the child "inherit" the parent's standard streams or the parent can redirect the child's streams to specific files.

The details of how a parent can redirect the child's standard streams to specific files will be discussed in the next folder. The details of a child "inheriting" its parent's standard streams will be discussed in this folder.

Let us explain what we mean by a child inheriting its parent's standard streams.

When a child inherits a standard stream from its parent, the parent and child processes "share" that stream. For example, if a child inherits its parent's standard output stream, then the parent and child share that output stream. On the other hand, if a child inherits its parent's standard input stream, then both processes share that one input stream. We now need to explain what it means for processes to "share" a stream.

Here is an illustration of a parent and a child process sharing their stdin and stdout streams.

```
                     parent
                +--------------+
                |              |
           +--->>stdin  stdout>>-------+
           |    |              |       |
           |    |      stderr>>--      |
           |    |              |       |
           |    |              |       |
           |    +--------------+       |
     ------+                           +----->
           |                           |
           |            child          |
           |       +--------------+    |
           |       |              |    |
           +------>>stdin  stdout>>----+
                   |              |
                   |      stderr>>--
                   |              |
                   |              |
                   +--------------+
```

Input and output streams are not shared in the same way. So we first look at shared output streams, and then consider shared input streams.

When two processes share an output stream, what each process writes to its output stream gets mixed together in the final destination of the shared stream. If the shared output stream is connected to a console window, then the lines of output from each of the two processes will be combined in the console window, usually in an unpredictable way. The same will happen if the shared output stream is connected to a file.

On the other hand, if two processes share an input stream, then both processes read data from that one stream. But two processes cannot "simultaneously" read bytes from a stream. Instead, the operating system must "serializes" their use of the input stream.

What this means is that whichever process calls the "read()" function first, that process gets whatever bytes are in the current input buffer for the shared stream. But the process that called "read()" second is then in line to be the next process to get the next batch of bytes that arrive in the stream's buffer. If both processes are continuously reading from the stream, then they will probably end up alternately reading input lines from whatever is the source of the input stream. Notice that any specific byte in the input stream can only be "read()" by one of the two processes. In the above picture for shared input streams, incoming bytes do not get copied into each of the two branches leading to the two processes. Any given input byte flows down only one of the two branches (not both).

In Linux or Windows, a child process always inherits its parent's standard streams (unless the parent redirects them; see the next folder). In Linux or Windows, if the parent's stdin is the keyboard, then so is the child's stdin. If the parent's stdout is the console screen, then so is the child's stdout (and if the parent's stdout is the file result.txt, then so is the child's stdout). This is very convenient. In Linux or Windows, every child starts with inheriting its parent's standard streams and if the parent chooses to, it can redirect the child's standard streams to files.

But in Java, when a child process is created, its three standard streams do not connect to anything! This is unlike Linux and Windows. The fact that Java does not work this way is a real problem. See the Java folder.

Here is the illustration of a parent and a child process sharing stdin and stdout. In this picture stdin is connected to the (shared) keyboard.

```
                     parent
                +--------------+
                |              |
           +--->>stdin  stdout>>-------+
           |    |              |       |
           |    |       stderr>>--     |
           |    |              |       |
           |    |              |       |
           |    +--------------+       |
keyboard---+                           +----->
           |                           |
           |            child          |
           |       +--------------+    |
           |       |              |    |
           +------>>stdin  stdout>>----+
                   |              |
                   |       stderr>>--
                   |              |
                   |              |
                   +--------------+
```

In this situation, there is a special case that we should consider. The user at the keyboard denotes that they have no more input to type by using the key combination ^Z on Windows (or ^D on Linux). Since only one of the two processes can be the recipient of that character, only one of the two processes sees an EOF condition (and probably exits). What about the other process? The user would need to know that it is necessary to enter a second ^Z (or ^D) from the keyboard in order to send an EOF condition to the second process. You can actually see this happen on Windows by using the program ParentChildShareStdin_parent.c and Window's Task Manager. (What would happen if the two processes had their shared stdin redirected to an input file? Would they both see the EOF condition or just one of them?)

It is worth noting that usually, when two processes share a stream, one of them is not actually using the stream. More often than not, if a parent creates a child and has the child inherit its standard streams, the parent will not use those streams while the child is running. The parent will wait until the child is done (the child process terminates) before the parent goes back to using its standard streams.

The best, and most common, example of a parent and child sharing streams is when we type a command at the cmd.exe (or bash) shell prompt. For example, suppose we type the command

```
C:\>find "a"
```

The shell (cmd.exe) is the parent process and it tells the operating system to creates a child process from the file "find.exe" (that file is at C:\Windows\System32\find.exe) and the shell has that child process share stdin and stdout with the shell process. Once find.exe starts, cmd.exe stops using its (shared) copies of stdin and stdout and lets find.exe use its copies. As you type at the keyboard, find.exe looks for the letter "a" in stdin and echoes to stdout any input line that contains an "a" (find is really a version of grep). When you type ^Z at the beginning of a line, find.exe sees an EOF condition on its copy of stdin, so it exits. But cmd.exe never sees the EOF condition (the ^Z). When cmd.exe notices that find.exe has terminated, cmd.exe goes back to writing and reading from stdout and stdin. It issues a new prompt on stdout, and waits to read a new command on stdin. All in all, a very elegant way for a user to make one program run other programs in a shared window.

6.) Shared Streams in Windows

Windows is similar to Unix. When a parent process creates a child process, the child automatically inherits the parent's standard streams (see the file CreateChildInheritStdStreams.c).

There is one exception to this. If the parent asks for the child to have its own console window, then the child does not automatically inherit the standard streams (the new console's standard streams are the default ones). But it is pretty easy to redirect the standard streams of the new console back to the parent's standard streams.

This folder also contains two pairs of examples that demonstrate what happens when two processes share an input or an output stream.

```
   ParentChildShareStdout_parent.c
   ParentChildShareStdout_child.c
```

and

```
   ParentChildShareStdin_parent.c
   ParentChildShareStdin_child.c
```

There are a number of experiments that you should try with these files.

Run

```
   ParentChildShareStdin_parent.exe
   ParentChildShareStdin_child.exe
```

with a shared keyboard as stdin. Find the two processes in the Task Manager. Type ^Z at the shared keyboard. Use Task Manager to see which process gets the EOF condition. Type some more at the keyboard to see that the other process is still running. Type ^Z again to close the second process.

Introduce random pauses (using the Sleep() function) into both ParentChildShareStdin_parent.c and ParentChildShareStdin_child.c so that they do not just alternate reading from the shared input stream.

Change the keyboard from cooked input to raw input. Try changing just one of the two processes to raw keyboard input. Try changing both to raw input.

7.) A Parent Redirecting a Child's Standard Streams

The examples in this folder (and the previous folder) show how a parent process can control the standard streams (stdin, stdout, stderr) of a child process.

There are two ways that the parent can control what a child's standard streams are connected to when the child is created. The parent can either have the child "inherit" the parent's standard streams or the parent can redirect the child's streams to specific files.

The details of a child "inheriting" its parent's standard streams were discussed in the previous folder. This folder shows how the parent can redirect the child's standard streams to specific files.

(Notice that we are not yet talking about pipes. That is, we do not yet want to connect the stdout of the parent to the stdin of the child. That would be a pipe (so would connecting the stdout of the child to the stdin of the parent). We will discus that in the next folder.)

What we show in this folder is how a parent can create a child with the child's standard streams redirected to files, as in this picture.

```
                     parent
                +---------------+
                |               |
    keyboard -->>stdin   stdout>>-------> console
                |               |
                |        stderr>>--->
                |               |
                +---------------+

                        child
                   +---------------+
                   |               |
    data.txt ----->>stdin   stdout>>----> result.txt
                   |               |
                   |        stderr>>--->
                   |               |
                   +---------------+
```

In Windows, the operating system provides a way for the parent to create the child in exactly the above situation by using the STARTUPINFO data structure passed to the CreateProcess() function.

In Linux, the child process is first created by fork() with inherited streams, as in this picture.

```
                     parent
                +---------------+
                |               |
           +--->>stdin   stdout>>-------+
           |    |               |       |
           |    |        stderr>>--->   |
           |    |               |       |
           |    +---------------+       |
keyboard --+                            +-----> console
           |            child           |
           |       +---------------+    |
           |       |               |    |
           +------>>stdin   stdout>>----+
                   |               |
                   |        stderr>>--->
                   |               |
                   +---------------+
```

After the fork() is complete the child will redirect its own streams to the files (as in folder 5) before the child calls exec().

7.) A Parent Redirecting a Child's Standard Streams in Windows

The examples in this folder show how a parent Windows process
can redirect the standard streams (stdin, stdout, stderr) of a
child process.

In Windows it is fairly straightforward for a parent to create a
child process and redirect the child's stdin and stdout to files.
It takes quite a few Windows function calls, some of them not at
all obvious (like making the handles to the files "inheritable"),
but the individual steps are easy to understand.

Unlike in Linux, the redirection of the child's streams is done
before the child process is created (by using the STARTUPINFO
data structure passed to the operating system's CreatProcess()
function). So as soon as the child process starts running, it
is using its redirected streams.

To redirect the child's stdin and stdout streams the parent process
needs to do (at least) these six steps.

1.) Ask the OS to create an input file handle.
2.) Ask the OS to create an output file handle.
3.) Redirect the child's stdin to the input file.
4.) Redirect the child's stdout to the output file.
5.) Ask the OS to create the child process.
6.) Close the parent's handles to the input and output files.

Here are some pictures that explain the steps in redirecting the
child's stdin and stdout streams. Start with a parent process that
has the keyboard as its stdin and the console window as its stdout
and stderr.

```
                  parent
             +--------------+
             |              |
 keyboard -->>stdin  stdout>>--------> console window
             |              |
             |       stderr>>-----> console window
             |              |
             +--------------+
```

The parent needs to open the two files that will be used by the
child. So the parent needs to create two new streams, an input
stream and an output stream. In Windows the two new streams are
represented by "handles" and we can call these new handles hIn
and hOut.

```
                  parent
             +--------------+
             |              |
 keyboard -->>stdin  stdout>>--------> console window
             |              |
             |       stderr>>-----> console window
             |              |
 double.c -->>hIn      hOut>>-----> result.txt
             |              |
             +--------------+
```

Once these files have been opened, the parent can create the child
and have the child inherit these two new streams as its stdin and
stdout and inherit the parent's stderr (this is done using the
STARTUPINFO data structure). This will mean that, at least for a
short while, the parent and child will be sharing the streams to
the two files.

```
                  parent
              +--------------+
              |              |
 keyboard --->>stdin  stdout>>-------------------------> console window
              |              |
              |       stderr>>--------------------+--> console window
              |
              |              |                    |
              |              |                    |
double.c -+-->>hIn      hOut>>-----+-> result.txt |
          |   |              |     |              |
          |   +--------------+     |              |
          |                        |              |
          |           child        |              |
          |     +---------------+  |              |
          |     |               |  |              |
          +---->>stdin   stdout>>--+              |
                |               |                 |
                |        stderr>>-----------------+
                |               |
                +---------------+
```

But the parent does not want to use the two file streams. Those two
streams are for the child process. So the parent process should close
its two handles to the files. That gives us the configuration that we
want.

```
                  parent
             +--------------+
             |              |
 keyboard -->>stdin  stdout>>----------------------> console window
             |              |
             |       stderr>>--------------------+--> console window
             |              |                    |
             +--------------+                    |
                                                 |
                     child                       |
               +---------------+                 |
               |               |                 |
  double.c --->>stdin   stdout>>----> result.txt |
               |               |                 |
               |        stderr>>-----------------+
               |               |
               +---------------+
```

Be sure to run CreateChildRedirectStdinStdout.exe with its
standard input and output streams redirected to files.

```
   C:\> CreateChildRedirectStdinStdout.exe < Readme.txt > something.txt
```

Notice that the redirection of the parent's streams should not
have any affect on the redirection of the child's streams.

```
             CreateChildRedirectStdinStdout.exe
                +--------------+
                |              |
  Readme.txt -->>stdin  stdout>>-------> something.txt
                |              |
                |       stderr>>--->
                |              |
                +--------------+

                      double.exe
                   +--------------+
                   |              |
    double.c ----->>stdin  stdout>>------> result.txt
                   |              |
                   |       stderr>>--->
                   |              |
                   +--------------+
```

Exercise: A somewhat convoluted way for a parent to redirect its child's standard streams is for the parent to first redirect its own standard streams (as in folder 5), then have the child inherit the parent's standard streams (as in folder 6), and then have the parent restore its standard streams to where they originally were.

Write a Windows program that redirects its stdin to a file (say double.c), redirects its stdout to another file (say result.txt), creates a child (say double.exe) that inherits the redirected streams from the parent, and then restores the parent's stdin and stdout to what they were when the program started up. Your program will need to keep copies of the stdin and stdout handles that the program began with and use the copies to restore the stdin and stdout handles back to what they were. Have your program wait for the child to end and then have your program read and write from its stdin and stdout to make sure that they have been properly restored.

Run your program with its stdin and stdout redirected like this,

```
C:\> myProgram < Readme.txt > soemthing.txt
```

to make doubly sure that your program is redirecting the child to the files double.c and result.txt and that your program then restores is own stdin and stdout to their original streams.

8.) Creating A Parent to Child Pipeline

In this folder we show how to create a simple pipeline between
a parent process and its child process.

By "simple pipeline" we mean that the stdout of the parent is
connected by a "pipe" to the stdin of the child. So whatever
the parent writes to its stdout shows up as input to the child's
stdin. We will also assume that the parent is reading data from
its stdin and that the child's stdout is inherited from the
parent. So we want to create something that looks like this.

```
              parent                           child
         +---------------+               +---------------+
         |               |     pipe      |               |
     --->>stdin   stdout>>--0========0-->>stdin   stdout>>--->
         |               |               |               |
         |       stderr>>--->            |       stderr>>--->
         |               |               |               |
         +---------------+               +---------------+
```

Notice in the above picture that there is something called a "pipe"
between the stdout of the parent and the stdin of the child. We DO
NOT draw the above picture the following way.

```
              parent                           child
         +---------------+               +---------------+
         |               |               |               |
     --->>stdin   stdout>>--------------->>stdin   stdout>>--->
         |               |               |               |
         |       stderr>>--->            |       stderr>>--->
         |               |               |               |
         +---------------+               +---------------+
```

This picture would imply that somehow the stdout of the parent is
connected directly to the stdin of the child in a way similar to
how a standard stream can be connected directly to a file. But
streams cannot be connected directly to other streams! To kind of
see why, imagine that the parent process called

```
   printf("hello");
```

but the child process was busy doing something and not reading from
its stdin. What would happen? Where would the "hello" string go if the
child process was not ready to immediately consume it? One possible
answer is that the parent process gets stuck on its call to printf
because the printf function cannot return until the child reads the
"hello" string from its stdin. But this is not a good solution. It
prevents the parent from getting anything done until the child reads
all of its input. Worse, it can lead to deadlock situations where a
chain of pipelines gets permanently stuck.

A better solution is to have the "pipeline" be a buffer owned by the
operating system.

```
              parent                             child
         +---------------+      pipe       +---------------+
         |               |   +--------+    |               |
     --->>stdin   stdout>>---|        |--->>stdin   stdout>>--->
         |               |   +--------+    |               |
         |               |     buffer      |               |
         |               |                 |               |
         |       stderr>>--->              |       stderr>>--->
         |               |                 |               |
         +---------------+                 +---------------+
```

When the parent process calls printf("hello") the operating system receives
the string "hello" and stores it in the pipe's buffer. (Remember that printf()
must lead to a system call to the operating system. When the operating system
gets the resulting system call, the OS notes that the destination of the
write system call is a pipe, not a file or an I/O device, and the OS stores

the string in the pipe's buffer which is part of the operating system.) When the child process calls getchar() the operating system gets a character out of the pipe's buffer and gives it to getchar. (Remember that getchar() must result in a read() system call, and when the OS sees that the system call is using a pipe, not a file or an I/O device, the OS reads data from the pipe's buffer to satisfy the read system call.) This way, the parent and child do not have to write to and read from the pipe in lockstep with each other.

If the parent process is a lot more energetic than the child process and the parent writes far more data into the pipe's buffer than the child reads from it, then it is possible that the pipe's buffer becomes full. If that should happen, then the operating system would need to block the parent process when the parent tries to write data into the full buffer. The parent process will stay blocked until the child process frees some space in the buffer by reading some data from it. When the operating system sees the free space in the buffer, it would un-block the parent and allow it to write some more data into the buffer. If the parent refills the buffer, the operating system would need to once again block the parent if it tried to write into the full buffer.

In a similar manner, if the child process tries to read from an empty pipe buffer, then the operating system would have to block the child process until the parent writes some data into the pipe. After the parent puts some data into the pipe's buffer, the operating system would un-block the child process and let it read the data from the pipe.

Notice that the pipe's buffer is the "inter-process communication channel" between the two virtual memory spaces that the parent and child processes run in (the buffer is in the kernel's address space).

It is important to know that the kernel space buffer within the pipe may not be the only buffer involved in the communication between the parent and child processes. When the parent and child communicate through the pipe the picture may actually look like the following. There might be a user space buffer in the parent process (used by printf() for example) and another user space buffer in the child process (used by scanf() for example).

```
          parent                                       child
     +---------------+   user space               +---------------+
     |               |    +-------+               |               |
 --->>stdin   stdout>>----|       |--+        +-->>stdin   stdout>>--->
     |               |    +-------+  |        |   |               |
     |               |     buffer    |        |   |               |
     |               |               |        |   |               |
     |        stderr>>-->   +--------+        |   |        stderr>>--->
     |               |      |                 |   |               |
     +---------------+      |    pipe         |   +---------------+
                            |  +-------+      |
                            +--|       |--+   |
                               +-------+  |   |
                                buffer    |   |
                                          |   |
                               +----------+   |
                               |              |
                               | user space   |
                               |  +-------+   |
                               +--|       |---+
                                  +-------+
                                   buffer
```

When the parent sees the end-of-file condition on its stdin stream, the parent closes its stdout stream (and the parent can then wait for the child to terminate). The operating system then flushes any data in the above three buffers and causes the end-of-file condition to become true for the child's stdin stream (after the child has read all of the data from all the buffers). The child process then knows that it can terminate. If the parent does not close its stdout stream, then the child will not know that there is no more data coming through the pipe and the child will block (forever) waiting for the parent to write something into the pipe's buffer. And since the child will be blocked but still running, the parent process will stay blocked (forever) waiting for the child to terminate. So the parent and child will wait forever on each other. A situation like this is called a "deadlock".

Creating a parent child pipeline is done in roughly four steps.
  Step 1) Ask the OS to create a pipe object.
  Step 2) Ask the OS to create the child process.
  Step 3) Redirect the parent's stdout to the input of the pipe.
  Step 4) Redirect the child's stdin to the output of the pipe.
However, the order of these four steps varies between Linux, Windows, and Java. See the individual sub folders.

A simple pipeline like the one above is most often used to allow two processes to work together (in parallel) to accomplish some task. Each process does only part of the task. After the first process has done its part of the task on a piece of input data from stdin, it passes its result to the second process to do its share of processing. The final result is written to stdout by the second process. Notice that as soon as the first process has passed a piece of data to the second process, the first process can read another piece of data from stdin and begin working on it while the second process is working on the data that the first process had just fed it (think of a small "bucket brigade" or a short assembly line). So the two processes are working at the same time (assuming that there is more than one CPU, which is almost always true these days). So a pipeline becomes a way to have two CPU's work together on a task, possibly speeding up the time to accomplish the whole task. This is one form of "parallel processing".

Here is a real example of a (non-parallel) parent-child pipeline. In Windows, when you execute the command

```
C:\> dir | find "z"
```

this is in fact a parent-child pipeline with cmd.exe as the parent and find.exe as the child.

```
              cmd.exe                        find.exe
          +---------------+               +--------------+
          |               |     pipe      |              |
keyboard-->>stdin  stdout>>---0========0-->>stdin  stdout>>-->console window
          |               |               |              |
          |        stderr>>--->           |       stderr>>--->
          |               |               |              |
          +---------------+               +--------------+
```

What happened to the dir program? Well, in Windows there is no such thing as the "dir.exe" program. The dir command is an 'internal' shell command (the Linux bash shell also has 'internal' shell commands). This means that the cmd.exe process itself does all the directory listing work and produces all of the directory listing output. Since cmd.exe itself does all the dir work, cmd.exe needs to pipe its own output into the find.exe process's standard input.

Question: cmd.exe does not really create a pipeline as in the above picture. Instead it would create the following parent-child pipeline.

```
                              +--------------------------------+
                              |                                |
              cmd.exe         |                  find.exe      |
          +---------------+   |              +--------------+  |
          |               |   |     pipe     |              |  |
keyboard-->>stdin  stdout>>---+  0=======0-->>stdin  stdout>>--+-->console
window
          |               |      |           |              |
          |        stderr>>-->   |           |       stderr>>--->
          |               |      |           |              |
          |       stream3>>------+           +--------------+
          |               |
          +---------------+
```

That is, cmd.exe will first create a new output stream for itself and then connect that stream to the input of the pipe. Why does cmd.exe create the pipe this way? (Hint: What should happen when find.exe finishes?)

For a list of all the 'internal' commands in cmd.exe, see this web page.
   https://ss64.com/nt/syntax-internal.html

Here is documentation on the bash shell's 'builtin' commands.
   http://man7.org/linux/man-pages/man1/bash.1.html#SHELL_BUILTIN_COMMANDS

Question: How would you do an experiment on Windows to verify that a command of the form

```
C:\> dir | some_program.exe
```

is in fact a parent-child pipeline between cmd.exe and some_program.exe? (Hint: It would help if you wrote some_program.c yourself.)

Exercise: Write a program that creates a parent-child pipeline but with the parent and child process in the opposite order from what it is above. That is, create the following pipeline with the stdout of the child feeding into the stdin of the parent (and with a shared stderr).

```
             child                           parent
        +---------------+               +---------------+
        |               |     pipe      |               |
    --->>stdin   stdout>>--0========0-->>stdin   stdout>>------->
        |               |               |               |
        |        stderr>>---+           |        stderr>>---+-->
        |               |   |           |               |   |
        +---------------+   |           +---------------+   |
                            |                               |
                            +-------------------------------+
```

Write another program with a pipeline that looks like this. Why is this a better configuration for the parent-child pipeline? (Hint: What should happen when the child terminates?)

```
    +-------------------------------------+
    |                                     |
    |        child                        |       parent
    |   +---------------+                 |  +---------------+
    |   |               |     pipe        |  |               |
 ---+-->>stdin   stdout>>--0========0-+   +->>stdin   stdout>>------->
        |               |             |      |               |
        |        stderr>>---+         |      |        stderr>>---+-->
        |               |   |         |      |               |   |
        +---------------+   |         +----->>stream3        |   |
                            |                |               |   |
                            |                +---------------+   |
                            |                                    |
                            +------------------------------------+
```

8.) Creating A Parent to Child Pipeline in Windows

We want to create the following pipeline using the Windows operating system.

```
            parent                          child
       +---------------+              +---------------+
       |               |     pipe     |               |
   --->>stdin   stdout>>--0========0-->>stdin   stdout>>------>
       |               |              |               |
       |       stderr>>-----+         |       stderr>>--+-->
       |               |     |        |               |  |
       +---------------+     |        +---------------+  |
                             |                           |
                             +---------------------------+
```

To set up this pipeline, the parent process needs to do (at least) these four steps.

1.) Ask the OS to create a pipe object.
2.) Redirect the child's stdin to the output of the pipe.
3.) Ask the OS to create the child process.
4.) Redirect the parent's stdout to the input of the pipe.

The parent also needs to do several other smaller steps that are described below.

When the parent creates the pipe object, the operating systems returns two handles, a handle to the input into the pipe and a handle to the output from the pipe. One important thing to realize is that the handle to the pipe's input is a handle to an output stream (because the parent will write to that stream) and the handle to the pipe's output is a handle to an input stream (because the child will read from that stream). This can be confusing. We will use terminology like "the pipe's input handle" and "the pipe's write handle" to mean the same thing (similarly, "the pipe's output handle" is the same thing as "the pipe's read handle").

Notice that the child will not need the pipe's input handle and, once the child has been created, the parent will no longer need the pipe's output handle.

When creating the pipe, the operating system defaults to making both of the pipe's handles not inheritable. But the child needs to inherit the pipe's output (read) handle. So immediately after creating the pipe object, the parent
instructs the operating system to make the pipe's output handle inheritable.

The parent then sets up the child's STARTUPINFO data structure and has the child inherit the parent's stdout and stderr streams, and has the child's stdin stream redirected to the (inheritable) pipe output handle.

The parent then creates the child process.

After the child has been created, the parent closes its copy of the pipe's output handle since the parent will not read anything from this handle. Notice that the parent's closing of this handle does not affect the child's ability to use it's (already inherited) copy of the handle.

After the child has been created the parent also redirects its own stdout to the pipe's input handle.

The pipeline is now in place, and the parent reads data from its stdin, does its share of the processing of the data, writes the data to its stdout (which is the pipe's input) where it can be read by the child, through its stdin, who finishes processing the data and writes it to its stdout.

When the parent sees the end-of-file condition on its stdin steam, the parent closes the pipe's input handle (and the parent can then wait for the child to

terminate). The operating system then causes the end-of-file condition to
become true for the child's stdin (after the child has read all the data from
the pipe's buffer), and the child process knows to terminate. If the parent
does not close the pipe's input handle, then the child will not know that
there is no more data coming through the pipe and the child will block
(forever) waiting for the parent to write something into the pipe's buffer.
And since the child will be blocked but still running, the parent process
will stay blocked (forever) waiting for the child to terminate. So the parent
and child will wait forever on each other. A situation like this is called a
"deadlock".

Here are some pictures that explain the steps in creating the pipeline.
Start with a parent process that has the keyboard as its stdin and the
console window as its stdout and stderr.

```
                  parent
             +--------------+
             |              |
 keyboard--->>stdin  stdout>>--------> console window
             |              |
             |       stderr>>-----> console window
             |              |
             +--------------+
```

The parent first needs to create the pipe object. Creating a pipe creates
two new streams, an input stream and an output stream.

```
                  parent
             +----------------+
             |                |
 keyboard--->>stdin    stdout>>--------> console window
             |                |
             |         stderr>>-----> console window
             |                |
        +--->>hPipeOut_Rd     |
        |    |                |
        |    |     hPipeIn_Wr>>----+
        |    |                |    |
        |    +----------------+    |
        |                          |
        +---------0=======0--------+
                    pipe
```

The parent creates the child process and tells the operating
system to have the child inherit the pipe's output stream as
the child's standard input stream and also have the child
inherit (share) both of the parent's standard output and error
streams (this is all done using the STARTUPINFO data structure)..

```
                   parent
              +----------------+
              |                |
 keyboard---->>stdin    stdout>>------------+----> console window
              |                |            |
              |         stderr>>--------+---|--> console window
              |                |        |   |
          +-->>hPipeOut_Rd     |        |   |
          |   |                |        |   |
          |   |     hPipeIn_Wr>>----+   |   |
          |   |                |    |   |   |
          |   +----------------+    |   |   |
          |                         |   |   |
          |          pipe           |   |   |
          +--------0=======0--------+   |   |
          |                             |   |
          |                             |   |
          |          child              |   |
          |     +----------------+      |   |
          |     |                |      |   |
          +---->>stdin   stdout>>----------+
                |               |      |
                |        stderr>>------+
                |               |
                +---------------+
```

Notice that the parent and child are sharing the output stream from
the pipe. But the parent has no need to read anything from the pipe,
so the parent can use the CloseHandle() function to close its handle
to that stream.

```
                  parent
             +----------------+
             |                |
 keyboard--->>stdin    stdout>>-----------+----> console window
             |                |           |
             |         stderr>>-------+---|--> console window
             |                |       |   |
             |     hPipeIn_Wr>>----+  |   |
             |                |    |  |   |
             +----------------+    |  |   |
                                   |  |   |
                    pipe           |  |   |
         +--------0=======0--------+  |   |
         |                            |   |
         |                            |   |
         |          child             |   |
         |     +---------------+      |   |
         |     |               |      |   |
         +---->>stdin   stdout>>----------+
               |               |      |
               |        stderr>>------+
               |               |
               +---------------+
```

Let us reorganize this picture a bit.

```
              parent
        +----------------+
        |                |
kbd---->>stdin    stdout>>--------------------------------------+----->
        |                |                                      |
        |         stderr>>----------------------------------+---|--->
        |                |              +---------------+   |   |
        |                |    pipe      |               |   |   |
        |     hPipeIn_Wr>>--0=======0-->>stdin   stdout>>-------+
        |                |              |               |   |
        +----------------+              |        stderr>>---+
                                        |               |
                                        +---------------+
```

Now the parent can use the SetStdHandle() function to redirect its
stdout to the input of the pipe.

```
              parent
        +----------------+
        |                |
kbd---->>stdin    stdout>>---+
        |                |   |
        |         stderr>>---|----------------------------------+--->
        |                |   |              +---------------+   |
        |                |   |    pipe      |               |   |
        |     hPipeIn_Wr>>---+--0=======0-->>stdin   stdout>>--------->
        |                |                  |               |   |
        +----------------+                  |        stderr>>---+
                                            |               |
                                            +---------------+
```

The parent no longer needs the extra handle to the input of the pipe
so it can set the handle to the value INVALID_HANDLE_VALUE (but the

parent should not close that handle).

```
              parent
        +----------------+
        |                |
kbd---->>stdin    stdout>>----+
        |                |    |
        |         stderr>>--+ |
        |                |  | |              +---------------+
        |                |  | |    pipe      |               |
        |                |  | +--0=======0-->>stdin   stdout>>--------->
        |                |  |                |               |
        +----------------+  |                |       stderr>>---+--->
                            |                |               |   |
                            |                +---------------+   |
                            +------------------------------------+
```

When we clean up the picture a bit, we see that the pipeline is now complete.

```
              parent                       child
         +---------------+               +---------------+
         |               |     pipe      |               |
kbd----->>stdin   stdout>>--0========0-->>stdin   stdout>>-------->
         |               |               |               |
         |        stderr>>---+           |       stderr>>---+-->
         |               |   |           |               |   |
         +---------------+   |           +---------------+   |
                             |                               |
                             +-------------------------------+
```

```
In this folder there are two files,
   Microsoft_Weird_Parent_Child_Pipeline.c
   Microsoft_Weird_Echo.c
copied from this URL.

   https://msdn.microsoft.com/en-us/library/windows/desktop/ms682499(v=
   vs.85).aspx
```
These files create an unnecessarily complicated parent-child pipeline. The pipeline also has a bug in it, caused by an incorrect comment made in the main file. If the input file to the parent process is too large, the parent-child pipeline will deadlock when the amount of data sent to the child is sufficient to fill both of the pipeline buffers. When both of the buffers are full, the parent becomes blocked on a write to the (full) child input pipe buffer. Once the parent is blocked, it cannot read from the child's output pipe buffer, which would be needed to free up space in the full child input pipe buffer.

The moral of this is that overly complicated programs lead to subtle bugs because it becomes too difficult to think about the program. There was no reason for the author to make their pipeline example so complicated. But the over complication lead to an easily avoided bug.

Experiment: After the parent calls CreatePipe(), the streams look like this.

```
                      parent
                +-------------------+
                |                   |
    keyboard--->>stdin       stdout>>--------> console window
                |                   |
                |            stderr>>-----> console window
                |                   |
           +--->>hPipeOut_Rd        |
           |    |                   |
           |    |        hPipeIn_Wr>>----+
           |    |                   |    |
           |    +-------------------+    |
           |                             |
           +----------0========0---------+
                       pipe
```

At this point, the parent process can use the pipe's input and output streams to write and read data to and from the pipe's buffer. This gives us a chance to do an interesting experiment to find out how big the pipe's buffer is. With the streams configured as in the last picture, write a while-loop that writes one byte of data at a time into the pipe's buffer. Keep a running count of the number of bytes written into the pipe and print (to stderr or stdout) the count for each byte written to the pipe. When the pipe's buffer gets full, the loop will hang and the last number written to stdout (or stderr) will be a count of the pipe's buffer size.

If you use fprintf(hPipeIn_Wr,...) to write to the pipe, use
fflush(hPipeIn_Wr)
after every call to fprintf() to make sure that bytes do not accumulate in a local buffer kept by fprintf(). What happens if you don't flush the calls to fprintf()?

9.) Creating A Child to Child Pipeline

In this folder we have a parent process create a two stage pipeline between two child processes. This is exactly what a shell program (like bash on Linux or cmd.exe on Windows) does when you use a pipe symbol in a command line like this.

```
C:\  program_1 | program_2
```

Here are the steps that a shell program goes through to "execute" this command line.

1.) The shell program should create a pipe object.
2.) The shell program should create two child processes, one each from the files program_1.exe and program_2.exe,
3.) The stdin of program_1 should be inherited from the shell,
4.) The stdout of program)2 should be inherited from the shell,
5.) The stdout of program_1 should be redirected to the input of the pipe.
6.) The stdin of program_2 should be redirected to the output of the pipe.

The correct order of these steps depends on the operating system, but each step is needed by each operating system.

Here is a diagram of how the parent and its two child processes should have their streams connected together.

```
                       parent
                  +---------------+
                  |               |
       +--------->>stdin   stdout>>---------------------------+
       |          |               |                           |
       |          |        stderr>>---->                      |
       |          |               |                           |
   ----+          +---------------+                           +---->
       |                                                      |
       |                                                      |
       |        child_1                        child_2        |
       |   +---------------+              +---------------+   |
       |   |               |    pipe      |               |   |
       +-->>stdin   stdout>>--0=======0-->>stdin   stdout>>---+
           |               |              |               |
           |        stderr>>--->          |        stderr>>---->
           |               |              |               |
           +---------------+              +---------------+
```

Notice that the parent and child_1 processes share an input stream and the parent and child_2 processes share an output stream. It will usually be the case that the parent process will put itself to sleep until both of the child processes are done so the parent will not interfere with the children's I/O on the shared streams. It is important to realize that after both children close their copies of the shared streams and then terminate, the parent can go back to using its copies of the two shared streams. When the children close their copies of the shared streams, that has no effect on the parent's copies of those streams.

When child_1 sees the end-of-file condition on its stdin stream, child_1 closes its stdout stream. The operating system then causes the end-of-file condition to become true for child_2's stdin stream (after child_2 has read all the data from the pipe's buffer). The child_2 process then knows that it can terminate.

If child_1 never closes its stdout stream, then child_2 (and the operating system) will not know that there is no more data coming through the pipe and child_2 will block (forever) waiting for child_1 to write something into the pipe's buffer. And since the child_2 will be blocked but still running, the parent process will stay blocked (forever) waiting for child_2 to terminate. So the parent and child_2 will wait forever on each other. A situation like this is called a "deadlock".

   9.) Creating A Child to Child Pipeline in Windows

We want to create the following pipeline using the Windows
operating system.

```
                       parent
                  +---------------+
                  |               |
       +--------->>stdin   stdout>>-----------------------------+---->
       |          |               |                             |
       |          |        stderr>>--------------------------+--|----->
       |          |               |                          |  |
   ----+          +---------------+                          |  |
       |                                                     |  |
       |                                                     |  |
       |         child_1                      child_2        |  |
       |   +---------------+              +---------------+  |  |
       |   |               |    pipe      |               |  |  |
       +-->>stdin   stdout>>--0=======0-->>stdin   stdout>>-----+
           |               |              |               |  |
           |        stderr>>---+          |        stderr>>--+
           |               |   |          |               |  |
           +---------------+   |          +---------------+  |
                               |                             |
                               +-----------------------------+
```

To set up this pipeline, the parent process needs to do (at least)
these seven steps.

  1.) Ask the OS to create a pipe object.
  2.) The stdin of the 1st child should be inherited from the parent.
  3.) Redirect the stdout of the 1st child to the input of the pipe.
  4.) Ask the OS to create the 1st child processes,
  5.) The stdout of the 2nd child should be inherited from the parent,
  6.) Redirect the stdin of the 2nd child to the output of the pipe.
  7.) Ask the OS to create the 2nd child processes,

The parent also needs to do several other smaller steps that are described
below.

Here are some pictures that explain the steps in creating the pipeline.
Start with a parent process that has the keyboard as its stdin and the
console window as its stdout and stderr.

```
                  parent
             +--------------+
             |              |
 keyboard--->>stdin  stdout>>--------> console window
             |              |
             |       stderr>>-------> console window
             |              |
             +--------------+
```

The parent first needs to create the pipe object. Creating a pipe creates
two new streams, an input stream and an output stream.

```
                  parent
             +----------------+
             |                |
 keyboard--->>stdin    stdout>>--------> console window
             |                |
             |         stderr>>-------> console window
             |                |
        +--->>hPipeOut_Rd     |
        |    |                |
        |    |     hPipeIn_Wr>>----+
        |    |                |    |
        |    +----------------+    |
        |                          |
        |                          |
        +---------0=======0--------+
                    pipe
```

The parent creates the child_1 process and tells the operating
system to have child_1 inherit the pipe's input stream as the
child's standard output stream and also have child_1 inherit
(share) both of the parent's standard input and error streams
(this is all done using the child's STARTUPINFO data structure)..

```
                  parent
             +----------------+
             |                |
kbd--+------>>stdin    stdout>>--------------> console window
     |       |                |
     |       |         stderr>>---------+--> console window
     |       |                |         |
     |   +-->>hPipeOut_Rd     |         |
     |   |   |                |         |
     |   |   |     hPipeIn_Wr>>----+    |
     |   |   |                |    |    |
     |   |   +----------------+    |    |
     |   |                         |    |
     |   |          pipe           |    |
     |   +--------0=======0--------+    |
     |                             |    |
     |                             |    |
     |              child_1        |    |
     |         +---------------+   |    |
     |         |               |   |    |
     +-------->>stdin   stdout>>---+    |
               |               |        |
               |        stderr>>--------+
               |               |
               +---------------+
```

Notice that the parent and child are sharing the input stream into
the pipe. But the parent has no need to write anything to the pipe,
so the parent can close its copy of that stream.

```
                  parent
             +----------------+
             |                |
kbd--+------>>stdin    stdout>>--------------> console window
     |       |                |
     |       |         stderr>>---------+--> console window
     |       |                |         |
     |   +-->>hPipeOut_Rd     |         |
     |   |   |                |         |
     |   |   +----------------+         |
     |   |                              |
     |   |          pipe                |
     |   +--------0=======0--------+    |
     |                             |    |
     |                             |    |
     |              child_1        |    |
     |         +---------------+   |    |
     |         |               |   |    |
     +-------->>stdin   stdout>>---+    |
               |               |        |
               |       stderr>>--------+
               |               |
               +---------------+
```

Let us reorganize this picture a bit.

```
                  parent
             +----------------+
             |                |
kbd--+----->>stdin     stdout>>---------------------> console window
     |       |                |
     |       |         stderr>>------------------+--> console window
     |       |                |                  |
     |       |                |                  |
     |       |     hPipeOut_Rd<<--------------+  |
     |       |                |               |  |
     |       +----------------+               |  |
     |                                        |  |
     |                                        |  |
     |              child_1                   |  |
     |         +---------------+              |  |
     |         |               |     pipe     |  |
     +-------->>stdin   stdout>>---0=======0--+  |
               |               |                 |
               |       stderr>>-----------------+
               |               |
               +---------------+
```

Now the parent creates the child_2 process and tells the operating
system to have child_2 inherit the pipe's output stream as the child's
standard input stream and also have child_2 inherit (share) both of
the parent's stdout and stderr streams (this is all done using the
child's STARTUPINFO data structure)..

```
                  parent
             +----------------+
             |                |
kbd--+----->>stdin     stdout>>------------------------------------------+--->
     |       |                |                                          |
     |       |         stderr>>---------------------------------------+--|--->
     |       |                |                                       |  |
     |       |                |                                       |  |
     |       |     hPipeOut_Rd<<--------------+                       |  |
     |       |                |               |                       |  |
     |       +----------------+               |                       |  |
     |                                        |                       |  |
     |                                        |                       |  |
     |              child_1                   |        child_2        |  |
     |         +---------------+              |   +---------------+   |  |
     |         |               |     pipe     |   |               |   |  |
     +-------->>stdin   stdout>>---0=======0--+-->>stdin   stdout>>------+
               |               |                  |               |   |
               |        stderr>>--------+         |        stderr>>---+
               |               |        |         |               |   |
               +---------------+        |         +---------------+   |
                                        |                             |
                                        +-----------------------------+
```

Notice that the parent and child are sharing the output stream from
the pipe. But the parent has no need to read anything from the pipe,
so the parent can close its copy of that stream.

```
                  parent
             +----------------+
             |                |
kbd--+----->>stdin     stdout>>------------------------------------------+-->
     |       |                |                                          |
     |       |         stderr>>--------------------------------------+---|--->
     |       |                |                                      |   |
     |       +----------------+                                      |   |
     |                                                               |   |
     |                                                               |   |
     |              child_1                            child_2       |   |
     |         +---------------+                 +---------------+   |   |
     |         |               |     pipe        |               |   |   |
     +-------->>stdin   stdout>>---0=======0---->>stdin   stdout>>-------+
               |               |                 |               |   |
               |        stderr>>--------+        |       stderr>>---+
               |               |        |        |               |   |
               +---------------+        |        +---------------+   |
                                        |                            |
                                        +----------------------------+
```
The pipeline is now complete.