Example 1:

```
ls | sort
```

Shell's Descriptor Table

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	
4	
5	

User enters command:

```
ls | sort
```

Shell's Descriptor Table

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	Pipe *read* end
4	Pipe *write* end
5	

User enters command:

```
ls | sort
```

Shell creates a pipe(), which occupies the next two positions in the table. Pipe must be created **before** forking so that both children inherit the pipe.

(Parent) Shell's Descriptor Table

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	Pipe *read* end
4	Pipe *write* end
5	

User enters command:

```
ls | sort
```

Shell forks two child process. One to become `ls`, another to become `sort`. Both get copies of the parent's descriptor table.

Child 1 (to become `ls`)

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	Pipe *read* end
4	Pipe *write* end
5	

Child 2 (to become `sort`)

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	Pipe *read* end
4	Pipe *write* end
5	

(Parent) Shell's Descriptor Table

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	~~Pipe *read* end~~
4	~~Pipe *write* end~~
5	

User enters command:

```
ls | sort
```

At this point, the children have everything they need, so the parent shell can close both ends of the pipe.

Child 1 (to become `ls`)

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	Pipe *read* end
4	Pipe *write* end
5	

Child 2 (to become `sort`)

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	Pipe *read* end
4	Pipe *write* end
5	

(Parent) Shell's Descriptor Table

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	~~Pipe *read* end~~
4	~~Pipe *write* end~~
5	

User enters command:

```
ls | sort
```

The child on the left of the pipe should close the read end of the pipe, as it will only be writing.

The child on the right of the pipe should close the write end, as it will only be reading.

Child 1 (to become `ls`)

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	~~Pipe *read* end~~
4	Pipe *write* end
5	

Child 2 (to become `sort`)

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	Pipe *read* end
4	~~Pipe *write* end~~
5	

(Parent) Shell's Descriptor Table

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	~~Pipe *read* end~~
4	~~Pipe *write* end~~
5	

User enters command:

```
ls | sort
```

The child on the left of the pipe should close the read end of the pipe, as it will only be writing.

The child on the right of the pipe should close the write end, as it will only be reading.

Note: ALL PROCESSES must close the write end of the pipe for the pipe to eventually generate an "end of file" (EOF). Otherwise, the second child will wait forever for more data that might be coming through the pipe.

Child 1 (to become `ls`)

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	~~Pipe *read* end~~
4	Pipe *write* end
5	

Child 2 (to become `sort`)

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	Pipe *read* end
4	~~Pipe *write* end~~
5	

(Parent) Shell's Descriptor Table

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	~~Pipe *read* end~~
4	~~Pipe *write* end~~
5	

User enters command:

```
ls | sort
```

The child on the left uses dup2() to copy the write end of the pipe into position 1 of the table.

The child on the right uses dup2() to copy the read end of the pipe into position 0 of the table.

Child 1 (to become `ls`)

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	~~Standard out~~ Pipe write end
2	Standard error (terminal output)
3	~~Pipe *read* end~~
4	Pipe *write* end
5	

Child 2 (to become `sort`)

File Descriptor Number	Associated stream/file/socket/pipe
0	~~Standard in~~ Pipe read end
1	Standard out (terminal output)
2	Standard error (terminal output)
3	Pipe *read* end
4	~~Pipe *write* end~~
5	

(Parent) Shell's Descriptor Table

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	~~Pipe *read* end~~
4	~~Pipe *write* end~~
5	

User enters command:

```
ls | sort
```

Everything is now set up correctly, and the child processes are ready to exec(). The `ls` process will write into the pipe without even realizing it, and the `sort` process will read from it.

Child 1 (to become `ls`)

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	~~Standard out~~ Pipe write end
2	Standard error (terminal output)
3	~~Pipe *read* end~~
4	Pipe *write* end
5	

Child 2 (to become `sort`)

File Descriptor Number	Associated stream/file/socket/pipe
0	~~Standard in~~ Pipe read end
1	Standard out (terminal output)
2	Standard error (terminal output)
3	Pipe *read* end
4	~~Pipe *write* end~~
5	

Example 1:

```
./standard_out_error 1> out.txt 2> err.txt
```

Shell's Descriptor Table

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	
4	
5	

User enters command:

```
./standard_out_error 1> out.txt 2> err.txt
```

No need for the parent shell to manipulate any FDs if there's no pipe. The child process can open files and call dup2() without any need to inherit any changes.

Shell's Descriptor Table

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	
4	
5	

User enters command:

```
./standard_out_error 1> out.txt 2> err.txt
```

Shell forks a child process, which will eventually execute the `standard_out_error` program.

Child (to become `standard_out_error`)

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	
4	
5	

Shell's Descriptor Table

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	
4	
5	

User enters command:

```
./standard_out_error 1> out.txt 2> err.txt
```

Child opens files for the I/O redirects.

Both of these files need to open for writing:

```
open([filename], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
```

If you were opening for reading (to replace stdin), it would look like:

```
open([filename], O_RDONLY);
```

Child (to become `standard_out_error`)

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	File `out.txt` (open for writing)
4	File `err.txt` (open for writing)
5	

Shell's Descriptor Table

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	
4	
5	

User enters command:

```
./standard_out_error 1> out.txt 2> err.txt
```

Child uses dup2() to move the file designated by 1> into position 1 of the table (replacing standard out).

Child uses dup2() to move the file designated by 2> into position 2 of the table (replacing standard error).

Child (to become `standard_out_error`)

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	~~Standard out~~ File `out.txt` (open for writing)
2	~~Standard error~~ File `err.txt` (open for writing)
3	File `out.txt` (open for writing)
4	File `err.txt` (open for writing)
5	

Shell's Descriptor Table

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	Standard out (terminal output)
2	Standard error (terminal output)
3	
4	
5	

User enters command:

```
./standard_out_error 1> out.txt 2> err.txt
```

Everything is now set up correctly, and the child is ready to exec():

The child process will send its output into the file(s) without needing to change the program at all.

Child (to become `standard_out_error`)

File Descriptor Number	Associated stream/file/socket/pipe
0	Standard in (terminal input)
1	~~Standard out~~ File `out.txt` (open for writing)
2	~~Standard error~~ File `err.txt` (open for writing)
3	File `out.txt` (open for writing)
4	File `err.txt` (open for writing)
5	