

# Re: Process Synchronization using Pipes

---

*Source:* <http://unix.derkeiler.com/Newsgroups/comp.unix.programmer/2007-03/msg00621.html>

---

- *From:* "Arturo" <[arturo.rotondo@xxxxxxxxxx](mailto:arturo.rotondo@xxxxxxxxxx)>
  - *Date:* 27 Mar 2007 16:32:34 -0700
- 

On Mar 26, 7:41 pm, j...@xxxxxxxxxxxxx (Jens Thoms Toerring) wrote:

Arturo <[arturo.roto...@xxxxxxxxxx](mailto:arturo.roto...@xxxxxxxxxx)> wrote:

I should have mentioned I'm using C/C++

Please decide if you use C or C++, they are different languages.

not shell script. OK so I'll post the code so you can see how I'm approaching this. I'm a beginner at this, the code may look a little ugly, but it's just a simple chain where Producer forks to Filter1 whom forks to Filter2 whom forks to Consumer. The problem is ... The while loop seems to only execute once. I don't understand why, I think it may have to do with close() cause I've reading about how it deallocates the file descriptor. If this is the case then how can I solve this?

There are some many things broken with your program that I don't see any way to correct them. So instead here's a probably incomplete list:

- 0) Your main() isn't 'int main(void)' even though you never use argc and argv.
- 1) You continue even if you can't open the input file (note that perror() doesn't calll exit()).
- 2) You continue even if you are unable to create a pipe.
- 3) You don't seem to understand what feof() is good for (and that you don't need it at all if you use fgets(), which returns NULL when EOF is reached).
- 4) You use vfork().
- 5) You create a set new processes plus new pipes in each child for each line you read in while a single set would do perfectly well (or probaly better!) for all the lines if each child would loop until it doesn't get any more input.
- 6) You're writing spaghetti code instead of a set of functions that deal with the sub-tasks.

## Re: Process Synchronization using Pipes

- 7) You don't check and retain the return value of a `read()` call but instead seem to think that `read()` would treat a `\0` character similarly to the way `fgets()` treats a `\n`. But that isn't the case. To `read()` all characters are just the same. If you use `write()` and `read()` you usually need some kind of "protocol" that by which the sender informs the receiver of how many bytes to expect, e.g. by first sending the number of bytes and only then the data.
- 8) You seem to believe that `toupper()` only would work with alphabetical, lower-case characters.
- 9) You don't test if `fopen()` succeeds for the output file
- 10) You're closing the read end of the first pipe each time you read a new line and then you start a new process for the next line which is going to try to read from this already closed pipe descriptor. That never will work, once it's closed it's closed and won't suddenly be opened again just because you try to write to it. So already when you start dealing with the second line the top-most parent process will die on a `SIGPIPE` signal the moment it's going to try to write to the pipe since its child only has a closed pipe handle and so there's actually no receiver.

Do yourself a favor and start with a simpler task, e.g. reading from a file in the parent process, passing the data to the child process via the pipe and there just write the data to another file.

The basic problem is the same but you at least will have a fighting chance to get it right after some time (and you can simply test if the output is ok by doing a 'diff' on the input and output file).

To get to that point read the man pages of the functions you use very carefully, especially those for the low level I/O functions like `read()`.

Especially note that the number of bytes to read you pass to `read()` is an upper limit only and that you never can assume that you will get that many – `read` may return anything between zero bytes and the upper limit you called it with (or even `-1` if an error occurred). For this reason `read()` is normally always embedded in a loop to be able to restart `read()` if less bytes arrived than you asked for. For that reason it probably will be simpler if you restrict yourself to the use of the higher level I/O function like `fgets` and `fputs()` for a start (there's that nice function called `fdopen()` that gives you a `FILE` pointer when you pass it a file or pipe handle as its argument).

And, BTW, if you post code it would be really helpful if you would keep the line length down to say 72 characters and replace tabs by spaces.

Regards, Jens

--

\ Jens Thoms Toerring \_\_\_ j...@xxxxxxxxxxxxx  
|\_\_\_\_\_ <http://toerring.de>

## Re: Process Synchronization using Pipes

Thanks to all, I got it to work finally.

Jens that was pretty harsh, but I took your advice about starting with simply reading from a file and then writing the content to another file,  
and after getting that to work, I built on it and finally got it all to work together.