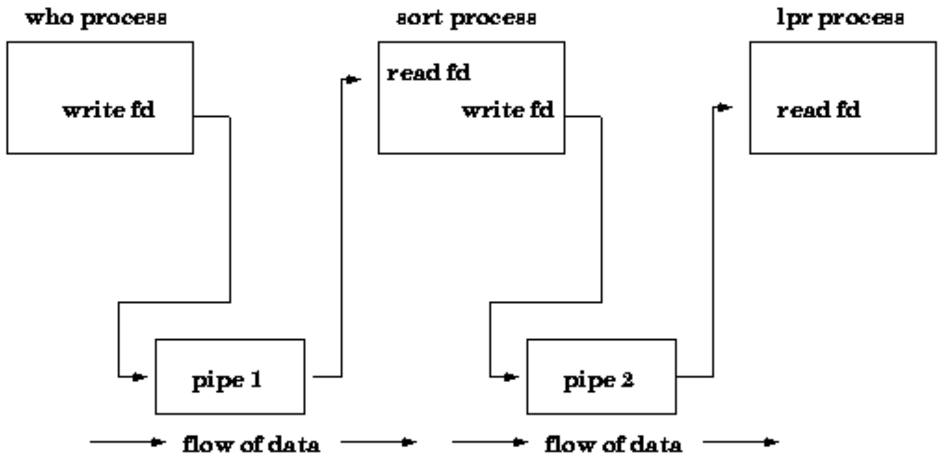
Unix Pipes

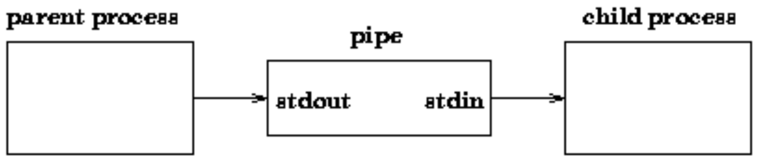
- A Unix pipe provides a one-way flow of data.
- For example, if a Unix users issues the command

  ```
  who | sort | lpr
  ```

  then the Unix shell would create three processes with two pipes between them:



- A pipe can be explicitly created in Unix using the *pipe* system call. Two file descriptors are returned--fildes[0] and fildes[1], and they are both open for reading and writing. A read from fildes[0] accesses the data written to fildes[1] on a first-in-first-out (FIFO) basis and a read from fildes[1] accesses the data written to fildes[0] also on a FIFO basis.
- When a pipe is used in a Unix command line, the first process is assumed to be writing to stdout and the second is assumed to be reading from stdin. So, it is common practice to assign the pipe write device descriptor to stdout in the first process and assign the pipe read device descriptor to stdin in the second process. This is elaborated below in the discussion of multiple command pipelines.

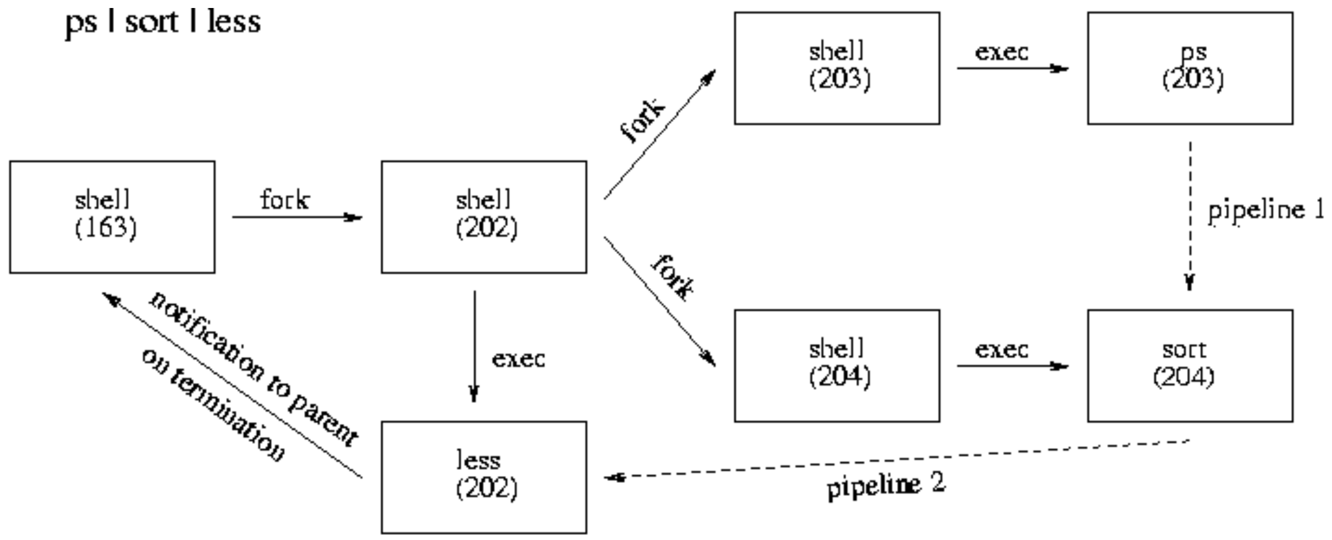


Multiple Command Pipelines: Architecture

Creating a pipeline between two processes is fairly simple, but building a multiple command pipeline is more complicated. The relationship between all of the processes in question is different than what one would expect when creating a simple pipeline between two processes. Normally a pipeline between two processes results in a fork() where child and parent are able to communicate.

In an extension of this model to n pipes, it is natural to assume a chain of processes in which each is the child of the previous one, until the n'th child is forked. But this model does not work because the parent shell must wait for the *last* command in the pipeline to complete, not the first, as would be the case with a chained pipeline.

A multiple process pipeline can be represented graphically as:



In this example we see that the parent shell (163) forks one child process (202) and then waits for it to complete. The child process (202) is the parent of all the pipe command processes. The child creates two pipes and then calls fork() for each of its children (203 and 204). Each new child process redirects STDIN and STDOUT to a pipe appropriately and calls exec() to execute the proper command. A process that has been exec()ed will never return. When the child (202) of the parent shell (163) reaches the last command it simply redirects STDIN to the second pipe and exec()s the last command. The parent (163) waits for this last command to exit. This is very important. The parent shell must wait on the last command to finish before continuing. If it does not, interactive commands such as "less" will not work properly.

The processes in the above figure have the following relationships:

```
Parent PID          Child PID
----------          ---------
163                 202
202                 203
202                 204
```

One important thing to note here is that each process in the pipeline is a child of the original child of the shell (pid 202). They are not children of each other the further down the pipeline we go. Another thing to note is that only the shell (process 163) executes a wait. All the others simply die after they exec their respective command.

Multiple Command Pipelines: File Descriptor Considerations

- In this sample implementation of multiple pipes, the process that is the child of the shell, i.e., 202 above, is responsible for creating all the needed pipes before it forks off any of its children. Thus, each of children has a set of file descriptors for all pipes in the total pipeline. In this example, processes 203 and 204 each have descriptors for the two pipes that are created.

- The fact that all forked processes have all the file descriptors implies that a programmer must address two problems:
  - Specifying for each process exactly which pipe file descriptor among the many it has access to is *its* stdin and *its* stdout, and
  - Eventually closing all file descriptors that comprise its pipes so that the pipes don't hang.

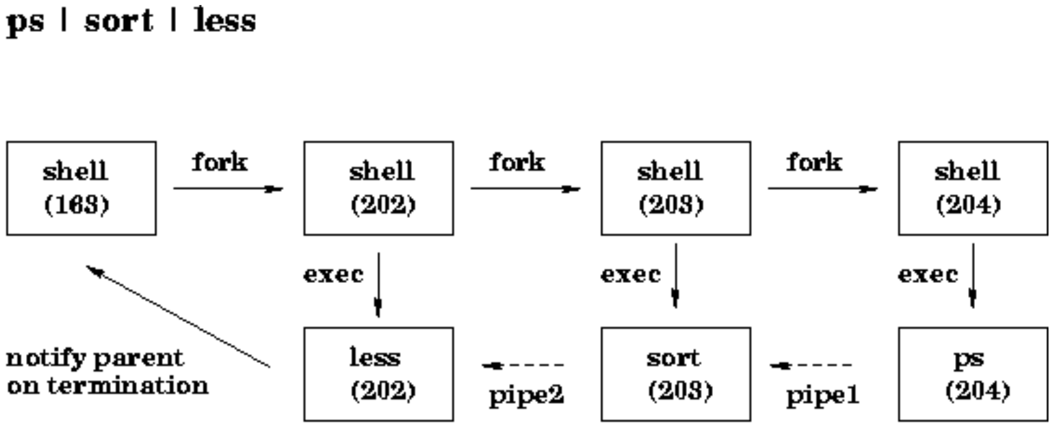
- To take care of the first problem, you must use the Unix command dup2(), to *duplicate* a pipe file descriptor to stdin or stdout (whichever is appropriate), e.g.,

  ```
  dup2(pipefd, stdin) or dup2(pipefd, stdout)
  ```

- To address the second problem, you must be sure that *each* forked process closes all of its pipe file descriptors before it execs its respective command. In the figure above, for example, when process 202 creates two pipes before it forks process 203 and 204, processes 202, 203 and 204 will all have all four descriptors for the two pipes. In this case, each of these processes, after dup2'ing its respective descriptor to standard in or standard out, must close all four descriptors.

Alternate Architecture for Multiple Pipeline Commands

A second process architecture to implement pipes is shown below. Duplication of files descriptors and closing all *inherited* descriptors must still be carefully addressed.



This architecture has an advantage over the previous one in that it can be implemented such that each forked shell has *knowledge* of two pipes and four file descriptors, maximum (shell 202 need create only one pipe itself). In the previous architecture, all the subsequent shells inherit all the pipes and file descriptors from shell 202.

Example Pipe Programs

In /usr/class/cis762/shell/examples there is an example program demonstrating how to use dup2, pipe and exec in the creation of pipes. Please note that in the example the piped commands are *hard-coded*, i.e., the pipes do not handle any command, or any number of piped commands, as is needed for a general shell piping mechanism. For this reason, the program uses execlp instead of execvp. execlp is easier to use when you know the command name ahead of time. execvp is easier to use if you are generating the command name in a character array and passing an array element to the exec command.

You may want to look at a second example of one, hard-coded pipe.

References

This is a good pipes tutorial.

Sandra Mamrak and Shaun Rowland
Last modified: April 27, 2004