Lecture 14: Interprocess communication and pipes

CS 3281
Daniel Balasubramanian,
Shervin Hajiamini, Sandeep Neema, and Bryan Ward

Overview of interprocess communication

- Interprocess communication is about ways to make processes “talk” to one another or “synchronize” with one another
  - Recall that processes have separate virtual address spaces, so they can’t just share variables
- Three big categories of IPC
  - Communication: how do processes exchange data
    - Example: send a list of files from one process to another
  - Synchronization: synchronize the actions of processes or threads
    - Think of synchronization as how to coordinate actions
      - Example: allow processes to avoid updating the same part of a file simultaneously
  - Signals: can be used for synchronization (but are primarily for other purposes)

Taxonomy of IPC

- The figure on the right shows a taxonomy of IPC mechanisms
    - We’ve looked at signals, mutexes, condition variables, and semaphores
    - We’ll be looking at pipes, message queues, shared memory, and memory mapping

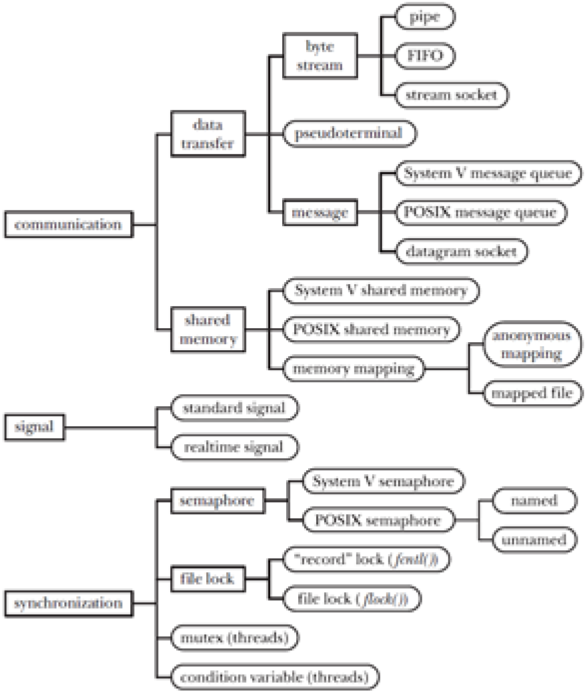


Figure 43-1: A taxonomy of UNIX IPC facilities

*Figure from *The Linux Programming Interface* by Michael Kerrisk

Fundamental concept: file descriptors

- File descriptor: normally small, non-negative integers that the kernel uses to identify the files accessed by a process
  - Example: when a process opens an existing file or creates a new file, the kernel returns a file descriptor that can be used to read or write the file
- All shells open three descriptors when a new program is run:
  - 0: standard input
  - 1: standard output
  - 2: standard error
- If nothing special done: all of them are connected to the terminal
  - In other words, input comes from the terminal, and output (including errors) are written to the terminal
- How does the kernel view and use file descriptors?

Kernel data structures for I/O

- (a) Per-process file descriptor table contains a pointer to a file table entry
- (b) Kernel maintains file table for all open files
  - Flags are read, write, append, sync, etc
  - V-node pointer is a pointer to v-node table entry
- (c) Kernel maintains V-node table
  - Each entry contains information about the type of file and pointers to functions that operate on the file
    - Usually also contains the file's *i-node*, which is its metadata
    - Note: Linux uses two i-nodes instead of a v-node: one generic and one specific

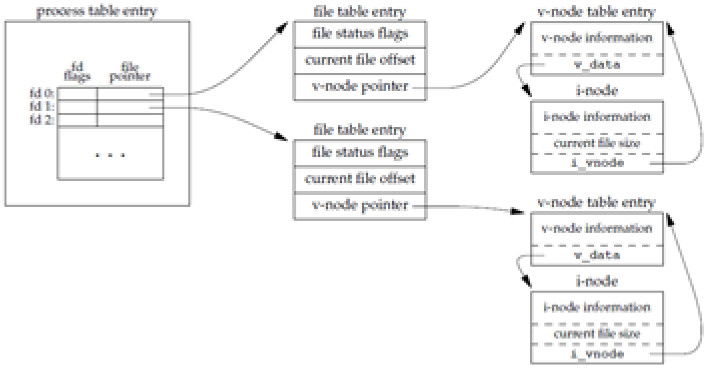


Figure 3.7 Kernel data structures for open files

*Figure from *Advanced Programming in the Unix Environment 3rd Edition* by Richard Stevens and Stephen Rago

Kernel data structures for I/O

- Figure on the right shows two processes with the same open file
- Two fds from the same process can also point to the same file table entry
  - The dup() system call
- Two fds from different processes can also point to the same file table entry
  - For instance, after a fork()
- Let’s use this knowledge to do something interesting

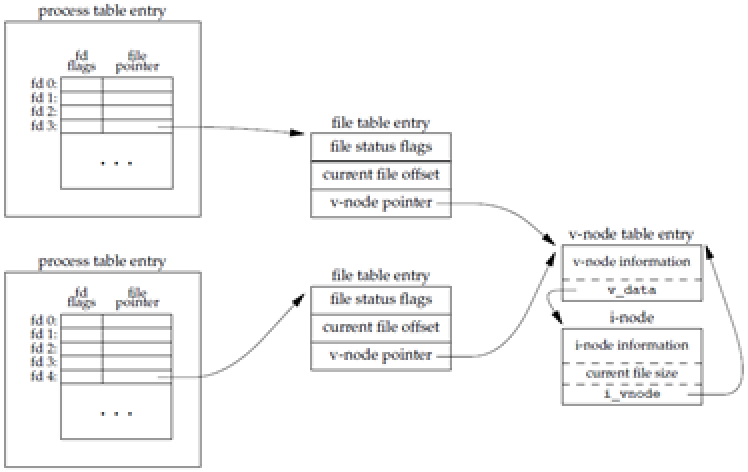


Figure 3.8 Two independent processes with the same file open

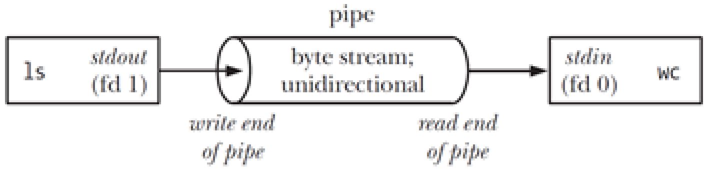
*Figure from *Advanced Programming in the Unix Environment 3rd Edition* by Richard Stevens and Stephen Rago

Pipes: motivation

- Big "real-world" use: connecting programs
  - How can the shell send the output of one program to the input of another program?
- Example
  - The ls program will show the contents of a directory
  - The wc program will count the number of lines in its input
  - How can we use these together to count the number of files in a directory?
- One (poor) solution:
  - Run ls and send its output to a temporary file (temp.txt)
  - Run wc using temp.txt as the input
  - Delete temp.txt

Pipes

- Better solution: use a pipe!
  - Think of it as a piece of “plumbing” that lets data flow from one process to another



- More formally: a pipe is a byte-stream IPC mechanism that provides a one-way flow of data between processes
  - All data written to the pipe is routed by the kernel to another process, which can then read it
  - Think of them as open files that have no corresponding image on your filesystem

Using pipes

- A process can create a pipe using the pipe system call:
  - `int pipe(int filedes[2]);`
  - That is, the pipe system call takes an integer array of size 2 (returns 0 on success)
    - filedes[0] can be used to read from the pipe
    - filedes[1] can be used to write to the pipe
- In-class demo
  - We will write one together!

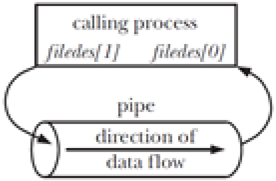


Figure 44-2: Process file descriptors after creating a pipe

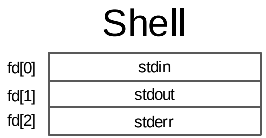
*Figure from *The Linux Programming Interface* by Michael Kerrisk

The shell and pipes

- Back to our motivation: how can pipes help the shell connect the output of one program to the input of another program?
  - Recall how the shell works:
    - Read a command
    - Do a fork() to create a new process
    - Do an exec() in the new process to run the program
    - Repeat
- We need the “standard output” of one process to go to the “standard input” of another process
  - Solution: have the shell “fix-up” the two processes’ file descriptors!

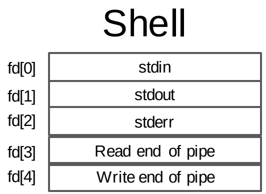
The shell and pipes (cont'd)

- Step 0: the shell has the three "standard" file descriptors open



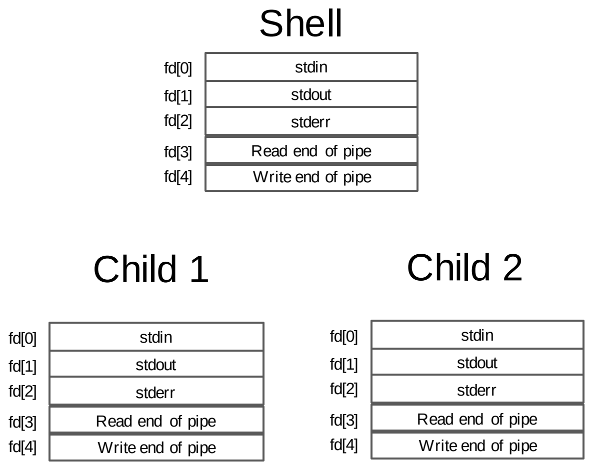
The shell and pipes (cont’d)

- Step 0: the shell has the three “standard” file descriptors open
- Step 1: the shell process calls pipe() to create the pipe



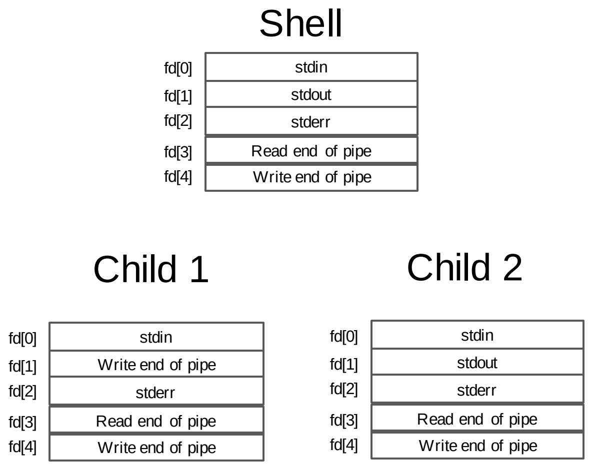
The shell and pipes (cont'd)

- Step 0: the shell has the three "standard" file descriptors open
- Step 1: the shell process calls pipe() to create the pipe
- Step 2: the shell process calls fork() twice to create the two child processes



The shell and pipes (cont’d)

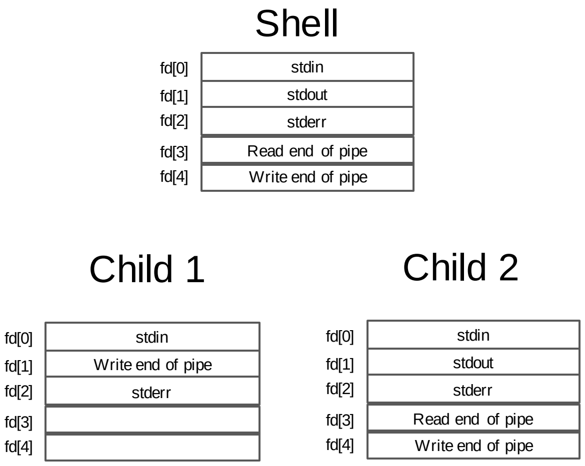
- Step 0: the shell has the three “standard” file descriptors open
- Step 1: the shell process calls pipe() to create the pipe
- Step 2: the shell process calls fork() twice to create the two child processes
- Step 3: in the first child process, the write end of the pipe is dup’ed onto the file descriptor for standard output



- The dup system call:
  - `int dup2(int oldfd, int newfd);`
  - Duplicates the descriptor in oldfd to the descriptor in newfd

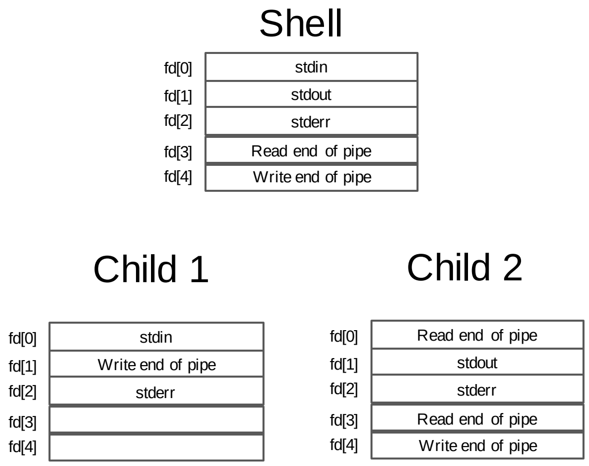
The shell and pipes (cont’d)

- Step 0: the shell has the three “standard” file descriptors open
- Step 1: the shell process calls pipe() to create the pipe
- Step 2: the shell process calls fork() twice to create the two child processes
- Step 3: in the first child process, the write end of the pipe is dup’ed onto the file descriptor for standard output
  - Child process closes both pipe fds and calls exec



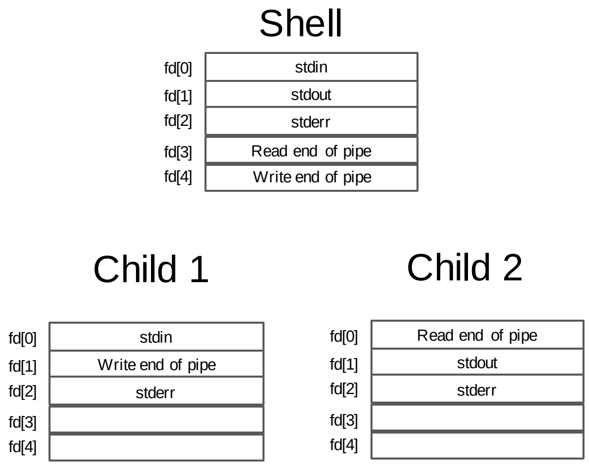
The shell and pipes (cont’d)

- Step 0: the shell has the three “standard” file descriptors open
- Step 1: the shell process calls pipe() to create the pipe
- Step 2: the shell process calls fork() twice to create the two child processes
- Step 3: in the first child process, the write end of the pipe is dup’ed onto the file descriptor for standard output
    - Child process closes both pipe fds and calls exec
- Step 4: in the second child process, the read end of the pipe is dup’ed onto the file descriptor for standard input



The shell and pipes (cont'd)

- Step 0: the shell has the three "standard" file descriptors open
- Step 1: the shell process calls pipe() to create the pipe
- Step 2: the shell process calls fork() twice to create the two child processes
- Step 3: in the first child process, the write end of the pipe is dup'ed onto the file descriptor for standard output
  - Child process closes both pipe fds and calls exec
- Step 4: in the second child process, the read end of the pipe is dup'ed onto the file descriptor for standard input
  - Child process closes both pipe fds and calls exec



The shell and pipes (cont'd)

- Step 0: the shell has the three "standard" file descriptors open
- Step 1: the shell process calls pipe() to create the pipe
- Step 2: the shell process calls fork() twice to create the two child processes
- Step 3: in the first child process, the write end of the pipe is dup'ed onto the file descriptor for standard output
  - Child process closes both pipe fds and calls exec
- Step 4: in the second child process, the read end of the pipe is dup'ed onto the file descriptor for standard input
  - Child process closes both pipe fds and calls exec
- Step 5: shell process closes both pipe fds