

## Contents

<b>About these reading notes</b>	<b>2</b>
<b>About this week's content</b>	<b>2</b>
<b>Chapter 15: Interprocess Communication</b>	<b>2</b>
15.1 Introduction . . . . .	2
15.2 Pipes . . . . .	3
Example: creating a pipe between a parent and its child . . . . .	4
Example: pager . . . . .	5
Implementation of the TELL and WAIT functions using pipes . . . . .	5
15.3 popen and pclose Functions . . . . .	5
Example: pager using popen . . . . .	6
Example: popen and pclose Functions . . . . .	6
Example: Transforming input using popen . . . . .	7
15.5 FIFOs . . . . .	7
Using FIFOs to Duplicate Output Streams . . . . .	8
Client-Server Communication Using a FIFO . . . . .	8
<b>signal(7)</b>	<b>9</b>
Signal dispositions . . . . .	9
Sending a signal . . . . .	9
Waiting for a signal to be caught . . . . .	9
Standard signals . . . . .	9
Queueing and delivery semantics for standard signals . . . . .	9
Signal numbering for standard signals . . . . .	10
Real-time signals . . . . .	10
Interruption of system calls and library functions by signal handlers . . . . .	10
Interruption of system calls and library functions by stop signals . . . . .	10
<b>signal(3)</b>	<b>10</b>
<b>kill(3)</b>	<b>11</b>
<b>raise(3)</b>	<b>11</b>

## About these reading notes

These are my own personal reading notes that I took (me, Franklin) as I read the textbook. I'm providing these to you as an additional resource for you to use while you're reading chapters from the textbook. These notes do not stand alone on their own — you might be able to get the *idea* of a chapter while reading these, but you're definitely not going to **get** the chapter by reading these alone.

These notes are inconsistently all of the following:

- Me summarizing parts of the text.
- Me commenting on parts of the text.
- Me asking questions to myself about the text.
  - ... and *sometimes* answering those questions.

The way that I would expect you to read or use these notes is to effectively permit me to be your inner monologue while you're reading the textbook. As you're reading chapters and sections within chapters, you can take a look at what I've written here to get an idea of how I'm thinking about this content.

## About this week's content

Our textbook surprisingly does not discuss interprocess communication (IPC), so we need to use some external sources.

An excellent source is [Advanced Programming in the UNIX Environment](#) (APUE) by W Richard Stevens and Stephen A Rago. I've requested that this chapter be scanned by the library, and it's available for you to view on the course web page through the course schedule.

You're also going to want to take a look at some manual pages for the system calls that you're going to be using for IPC (e.g., `signal`, `kill`, and `pipe`).

## Chapter 15: Interprocess Communication

### 15.1 Introduction

- Threads can talk to each other by reading and writing memory, processes need to be able to talk to each other, too!

But the only way for these processes to exchange information is by passing open files across a fork or an exec or through the file system.

- Why is that? Given what we know so far about processes, how else *could* you communicate between them without using any extra infrastructure?
- Why is “through the file system” a reasonable way to communicate between processes? Think about the abstraction of memory vs the abstraction of files for processes. Can processes see each other’s memory? Can you open the same file in two different processes?
- Figure 15.1 shows many different kinds of mechanisms for IPC, and OS support for these different mechanisms. The ones we’re mainly interested in are pipes, FIFOs (first-in-first-out), and signals.
  - You can generally safely ignore the acronyms UDS, SUS, XSI, etc. We care almost exclusively about Linux in this course.
  - Semaphores and shared memory are also good ways for processes to communicate with one another; semaphores are more of a signalling/locking mechanism. Shared memory is something you *might* see in class, but it’s a way for processes to share memory similar to threads, but it’s not anywhere near as easy to use.
  - If you’re deeply interested in the idea of semaphores, you should check out “[The Little Book of Semaphores](#)” by Allen B Downey. It’s free! And it’s ~300 pages about semaphores! Fun!
  - If you want to spend more time with sockets and streams, take COMP 3010 Distributed Computing or COMP 4300 Computer Networks.

## 15.2 Pipes

- Pipes are *old*.
- A pipe is an IPC mechanism for communication between processes that share a common ancestor (i.e., they have the same parent/grandparent/great grand parent/... process).

Every time you type a sequence of commands in a pipeline for the shell to execute...

- This is talking about the pipe symbol on the command line (i.e., `|`). You can use this command to connect the standard output of the command on the left side of the pipe to the standard input of the command on the right side of the pipe. For example, try running this:

```
yes | head -10
```

This command says “run the `yes` command (which prints out `y` repeatedly to standard output) and connect this standard output to the standard input for the command `head -10` (which prints out the first 10 lines of input on standard input)”.

- Uses **file descriptors** to facilitate communication (`int fd[2]`).
  - Why do you think that we’re using file descriptors? This is what we’re using with system calls like `open`, `read`, `write`, and `lseek`, why are we using this for communicating between processes?
- Figure 15.2 shows using a pipe in “half-duplex” mode. For the purposes of this course (and as stated above: for maximum compatibility), we’re limiting ourselves to “half-duplex pipes”. “Half-duplex pipes” is a really fancy way of saying “one way pipes”.

A pipe in a single process is next to useless.

- Why would that be? Why would a pipe in one process be useless? Think about it: Who would you even be communicating with if you created a pipe in one process?
- Figure 15.3 now shows us a more realistic use of `pipe` with `fork`.
  - Note that the `pipe` system call seems to be used in conjunction with `fork`! You **must** call `pipe` *before* `fork`, then close different ends of the pipe in the child and parent processes.
    - ★ If the `pipe` system call creates file descriptors, and the file descriptors are valid for both the child and parent process after a `fork`, what does this tell you about file descriptors in terms of ownership? Where does a file descriptor live? (e.g., the TCB? PCB?)
- Why do the two parent/child diagrams show the pipe existing in the kernel?

It’s possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing.

- Do you think this has any issues that might be related to critical sections and threads?
- The authors are writing here about how signals are sent to a process when a `write` system call is made, but the write-end of the pipe is closed. This pretty closely resembles a pattern you’ve seen before in programming! What is it?

### Example: creating a pipe between a parent and its child

- This shows us the order of operations in terms of calling `pipe`, then doing `fork`, and how each of the parent and child have to close different ends of the pipe.
  - The line `write(STDOUT_FILENO, line, n);` is *effectively* what `printf` or `puts` are ultimately calling. `STDOUT_FILENO` is the file handle number for ... standard output.

### Example: pager

- This is a slightly more complicated example.
- A “pager” is a program on the command line that’s job is to show you one “page” of output at a time. Two programs that are used to do this are `more` and `less`. `less` is a terrible pun (less is more).
  - When you’re using `man` (the manual pages), you’re using a pager!
- This program opens the filename passed on the command line, then reads its contents, writing them to the write end of the pipe, and the child reads from the read end of the pipe.
  - The child doesn’t *explicitly* read from the pipe, but instead, we’re using this new system call `dup2` to **overwrite** the child’s standard input with the read end of the pipe that was created. You should refer to `man dup2` for more information about this system call.
- This program basically does the following pipeline that might be run on a command line:

```
cat file.txt | /bin/more
```

This is an example of a [useless use of cat](#), but is an alternative view of what’s happening in this code.

### Implementation of the TELL and WAIT functions using pipes

You can safely ignore this section, it’s referring to a different chapter in APUE.

## 15.3 popen and pclose Functions

- OK, so we’re feeling comfortable with `fork` and `exec` (maybe). We’re also sort of feeling comfortable with `pipe`, and *maybe* `dup2` (`dup2` is in Figure 15.6 and is used to replace file handles).
  - The `popen` and `pclose` pair of functions kind of make it easy to run a process and *treat its standard output like a file that you can read or its standard input as a file you can write*. It’s a fairly common pattern to launch a process from within your own C program and want to read from it or write to its standard input.
  - You can learn more about `popen` and `pclose` by inspecting the manual pages (`man popen` and `man pclose`).
- Note that these functions are part of the **standard library**, they are explicitly not system calls.

- Without any other information (i.e., pretend that we can't see `#include <stdio.h>`, what about the function signatures of `popen` and `pclose` tells you that they are not system calls?
- If we get a `FILE *` back from `popen`, why do you think we can't just use `fclose`?
- Notice that the argument you're passing to `popen` describes how you're going to interact with the process that's forked (e.g., you're reading its `stdout` or writing to its `stdin`). Figures 15.9 and 15.10 show us the difference between passing "r" and "w".
  - Can you pass **both** "r" and "w"? Check the man page! Try doing it!
  - Does it even make sense to pass both? What would that mean? The physical thing that you're reading from the `FILE *` would be different from the physical thing that you're writing to the `FILE *`...
- The authors refer to the `system` function. `system` is *sort of* like `fork` and `exec` together, except instead of launching the program you specify directly, the entire string is passed to a *shell* to interpret and execute. That can be pretty convenient compared to doing `fork` and `exec` yourself, especially if you want to use special shell features like [file globbing](#) or redirection.

### Example: pager using popen

- This *simplifies* the pager program significantly. The functionality is the same, but we don't have to dance around `fork`, `exec`, `pipe`, and `dup2`, it's all just handled for us.
  - Plus, we get to use those nice `FILE *` functions instead of dealing with file handles.
- Check out how they define `PAGER`. Does that expression work in the shell on your local machine? Does it work for your aviary's default shell? (hint: it doesn't) What shell *does* it work on? (it's `bash`).

### Example: popen and pclose Functions

- Remember: these functions **are not** system calls. That means that you can implement this kind of a function in your own code using the basic `fork`, `exec`, and `dup2` system calls.
  - This snippet of code is actually doing some cool stuff around file descriptors and using them as array indices. This is effectively building an open file table! This is something that the OS needs to track for both a PCB (e.g., Linux's `struct task_struct`) or at the OS-scope level (all open files for all processes).
- For reference, you can also look at [the source code for glibc's popen](#).

- Reading through the source for APUE's `popen` gives us an answer to the above question about using "r" **and** "w" for `popen`. Or, at least it tells us that we *can't* do it, even if it doesn't explain why.
- Reading through the source for APUE's `pclose` might give you some hints about why we can't just use `fclose` on the `FILE *` returned by `popen`.
- There are some security concerns around `popen`! Yeah, it's convenient, but it's really easy to get it wrong.
  - There is **a lot** going on under this statement, but basically: it's straightforward for a malicious user to get your program to run any other program as your user. This is *really* bad if your code is running as root.

### Example: Transforming input using `popen`

- This is the last example for `popen`, showing a small program called `myuc1c`.
- Figure 15.3 is showing the visual relationship between the running programs in Figures 15.14 and 15.15 and the "terminal" where you (a user) is typing stuff in.
- This is another nice example, showing how different processes can interact with one another, but this time with `popen`.

## 15.5 FIFOs

- All the piping we saw up until this point are **anonymous pipes**.
  - They have no names, and the only way to even know about the existence of these pipes is to have had the pipe be created by your parent process (i.e., you were `fork`-ed off of a process that called `pipe` before `fork`).
- Sometimes you want different processes to talk to each other that are not related to each other this way!

We saw in Chapter 4 that a FIFO is a type of file.

- OK, so you didn't read chapter 4; that's OK. A FIFO is a type of file. That's a really short way of saying that it's a *thing* that's provided by the OS that you can interact with via system calls like `open`, `read` and `write`.
- We get two new system calls to do this, but `mkfifo` is really the one we care about right now.
  - `mkfifo` takes a **path to a file** because it **makes a file**.

- This means that working with the FIFO is *effectively the same* as working with a file (as above)!
- Again (and again and again and again): Once you’ve created the FIFO, you’re literally working with a file. OK, not literally. Figuratively? Abstractly? Yeah, abstractly. A FIFO is the same abstraction as a file, so you can use the same file interface of `open`, `read`, and `write`.
  - As far as you can tell, though, a FIFO looks and behaves like a file, even if you do `ls` in the directory where the FIFO lives.

### Using FIFOs to Duplicate Output Streams

- Now we’re starting to build up fairly complex pipelines of information. In the first figure, the output from `prog1` is being fed into inputs for both `prog3` and `prog2`.
  - *Can you do this with anonymous pipes?*
- `tee` is a command from [the `moreutils` package](#) that effectively accomplishes what this diagram does. Its name is inspired from the `tee` command, which reads from standard input and writes to both standard output and a file (see `man tee`).

### Client-Server Communication Using a FIFO

- Oh hey look, it’s `tee`!
- Note this specific idea of a “well-known” path name. There’s no magic here (not here, anyway); your distributed client-server architecture program needs to have some known thing between clients and servers so that they can communicate with each other. That known thing is a path to a FIFO that clients and servers can read/write.
- This “client-server” architecture is a fairly common use of FIFOs.
  - It’s nice because it makes it easy to build a client-server architecture on a single machine, then spread it across multiple machines *fairly* easily later on if you need to scale up.
- The problem described here with multiple readers is that readers can’t know when something was written *for them*. The solution is to add **even more** FIFOs, basically one for each “client”.
- Ultimately, figures 15.22 and 15.23 are ways that we can get around some of the issues of FIFOs, like them being “half-duplex”.
  - One of the issues figure 15.23 is trying to address is that (as above), clients can’t know when a message is for them. Could you handle this situation without actually adding more FIFOs? What would you have to change about how you (the server) communicate with the clients in terms of the messages you’re sending?



## **signal(7)**

- This is an overview of signals!
- You only need to really care about the sections that are stated here.

### **Signal dispositions**

- All signals have a default action, including, uh, stop running the process when a specific signal is received.

### **Sending a signal**

- There are 6 different functions (they're listed as "system calls and library functions") for sending signals to processes.
  - Why are there so many???
  - Which ones do you think are system calls and which are library calls? Why are they different? What is the difference between a system call and a library call?

### **Waiting for a signal to be caught**

- There are also two system calls for waiting around for a signal to be sent.
  - In terms of what you saw with threads and synchronization primitives, does this match any of those concepts?

### **Standard signals**

- There's also a long list of standard signals. Definitely don't memorize this!
  - Be *aware* of this list, and be aware of signals that "cannot be caught, blocked, or ignored".
    - ★ Why wouldn't you want to, for example, catch or ignore SIGKILL/SIGSTOP???

### **Queueing and delivery semantics for standard signals**

- This is interesting: a process can receive multiple signals at the same time, but "Standard signals do not queue".
  - What do they do, then?

## Signal numbering for standard signals

- Again, definitely don't memorize this!
- How is this (numbering) related to system calls?
- Why do you think different architectures have different signal numbers?

## Real-time signals

- This is outside the scope of our course.

## Interruption of system calls and library functions by signal handlers

- OK, interesting. A signal can be received by a process (or, at least, sent to a process) while the OS is servicing a system call on its behalf.
  - That means that a signal can be received by a process while in the middle of something like `a read` or `a write`.
- Looking at the description for `A1` and the `hard_work` function, does this explain why we have to keep sleeping?

## Interruption of system calls and library functions by stop signals

- This is telling us that we can **pause** a process, then restart it again using signals. Neat!
  - Some system calls have the behaviour of indicating a failure when this happens. Why do you think they would do this? Look at the list of system calls that exhibit this behaviour, what do they have in common?

## `signal(3)`

- This manual page almost immediately tells you to avoid using this system call and to instead use `sigaction`. In terms of our course, we care about the `signal` system call because it's *simple*. **Look at** the manual page for `sigaction`, how is it different from the `signal` system call?

## **kill(3)**

- `kill` is a terrible name for what this does.
- The manual page here tells us that a process can have a group ID (in the section starting “If *pid* is 0...”). This extends our previous perception of what a process might have in its PCB. Why might a process have a group at all?
- Why might the `kill` system call fail with `EPERM`? Can you send signals to a process that’s not owned by you? Why not?

## **raise(3)**

- While this is a system call, honestly, it’s a convenience system call (doing `getpid` on your behalf).
  - Also, neat: a new system call! `getpid`
- Why on earth would you want to send a signal to **yourself**?