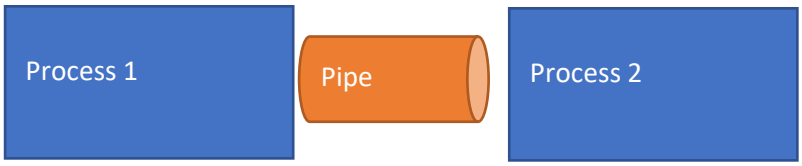
Lecture 12: Pipes

Pipes and Redirection

Pipes are a fancy version of redirection. So, to understand pipes, you should first familiarise yourself with redirection. As a quick refresher, many programs consist of a stream of information being converted into another, different stream of information. Redirection is the process of changing where that information comes from (whether standard in or a file) and where that information goes (whether standard out or another file).

Pipes serve as a form of redirection that exists between programs. Instead of reading from a file (which is essentially a finite process once the file runs out) or writing to a file, pipes allow one process to dynamically read the output of a second process and use it as input.



Pipes in Bash

We've already covered this, but as a quick refresher, you use the '|' symbol between two function calls. The output of the first function is given as input to the second function such as:

```
ls | wc -l
```

Here the output of "ls" is given as input to "wc -l" resulting in (presumably) the number of files in the current directory being printed out.

Pipes in C

While my previous description is probably sufficient to be an able user of pipes in Bash, there is something missing. While being able to use pipes is really useful, it doesn't necessarily imply that one understands pipes and how they work. In Bash the usage of pipes is actually somewhat limited in so far as there is an explicit way of calling them: "f1 | f2".

Under the Hood

So, what is a pipe, in terms of actual codey stuff? Well, if you have one program which is spitting out characters, and another that wants to read it, you need to have two programs that are happy to talk to one another. Next, you to have some way of transferring the characters between the programs. For this to work, there are two real options. One is that the second program is somehow looking directly into the first program and using memory references within that program… somehow. This seems a bit dangerous and difficult to implement. For example, how does the first program know to wait for the second program if it is producing output too quickly. What about the other way around, what if the second program reads faster than the first program is producing things. This is just not going to work.

Another way would be to have a commonly referenced bit of memory, that both programs are aware of where one program can dump data into and the other can read that same data from. Therefore, for a pipe to work, you need a buffer (a bit queue) where the output of one program can sit while the

other program reads it. That way the first program can wait if the buffer is full and then dump new characters when the buffer is empty.

This method also requires there to be some protocol around who reads and who writes and how that process works. In C, this is handled by file-descriptors (recall from previous lectures). In this case there will be one file-descriptor for reading, and one for writing. In C pipes, these values are stored in a two-element **int** array (i.e. int[2]). Note, the first element is the reading file descriptor and the second element is the writing file descriptor.

The Operation of the Reader and the Writer

Just to reiterate, the way it will work is that the writer will write until the buffer is full. The reader will read until the buffer is empty. Thus, the two processes can work together in a sort of handover of characters.

To do this, what we want to do is trick the reader into thinking that its standard-in is the file-descriptor defined by the pipe (i.e. pipe[0]) and also trick the writer into thinking its standard-out is the file-descriptor defined by the pipe (i.e. pipe[1]).

In code terms, this usually looks like the following:

For the writer:

```
close(pipe[0]);        // A writer does not read
dup2(pipe[1], 1);      // Standard out is now the pipe write end
close(pipe[1]);        // Close the pipe file-descriptor (it's is
                       // already stdout)
exec
```

For the reader:

```
close(pipe[1]);        // A reader does not write
dup2(pipe[0], 0);      // Standard in is now the pipe read end
close(pipe[0]);        // Close the pipe file-descriptor (it's is
                       // already stdin)
exec
```

This may look a little confusing. Closing the wrong end of the pipe makes a lot of sense… if you realise that both programs need to know where the pipe is, so both programs get both ends of the pipe… open. This is problematic if we think about how a pipe works. For the pipe to function, it needs to orchestrate when to allow data in and out of the buffer. If a process that never reads has opened the reading end of the pipe, then the pipe might just wait indefinitely for that process to read from the pipe… something it will never do because it is the writer. So, without these initial **close** calls, the program will likely hang indefinitely. The **dup2** calls are pretty self-explanatory, it is simply the redirection of stdin/stdout to the file-descriptor defined by the pipe. Lies. This is not self explanatory. What happens here is that the thing that file descriptor 3 is looking at (we assume 0 stdin, 1 stdout, 2 stderr so fd=3 is probably the reading end of the pipe, at least for this example) is now also going to be pointed at by file descriptor 0 (i.e., stdin). The same applies for stdout being replaced with the thing file descriptor 4 (i.e., the writing end of the pipe) is pointing at. The final close call… always confuses me, even when I know what it is doing. When the dup2 occurs stdin/stdout are now connected to the buffer. The old pipe file descriptors are now irrelevant. Worse than that, they can potentially trick the buffer into waiting indefinitely similar to the reason we did the first **close** call.

Two Programs, One Pipe

So far, so good. The code makes sense, but seriously, how to two different programs access the same bit of memory? You could do it using a file, but surely that would be slow and awful. What you really want is two programs that can access the same variable. Except, that makes no sense… unless they are actually the same program. So, we need to use **fork**.

Code Time

```
int main()
{
	// Setup
	char * commands[2];   // Child Commands
	commands[0] = "ls";
	commands[1] = NULL;
	char * commands[2];   // Parent Commands
	commands2[0] = "wc";
	commands2[1] = NULL;
	char test_buffer[100]; // For printing

	// Fork and Pipes
	int my_pipe[2];
	if (pipe(my_pipe) == -1) {
		perror("Cannot create pipe\n");
	}
	pid_t my_pid; // Fork
	my_pid = fork();
	if (my_pid < 0) {
		printf("Failed Fork\n");
	}

	// Parent
	if (my_pid > 0) {
		close(my_pipe[1]);
		dup2(my_pipe[0], STDIN_FILENO);
		close(my_pipe[0]);
		wait(NULL);
		execvp("wc", commands2);
	}

	// Child
	else // I.e. my_pid == 0
	{
		close(my_pipe[0]);
		dup2(my_pipe[1], STDOUT_FILENO);
		close(my_pipe[1]);
		execvp("ls", commands);
	}
}
```

Code Explained

There are four parts to the above code: Setup, Fork and Pipe, Parent and Child. **Setup** is basically just initialising the commands we want to perform (in this case, ls and wc). **Fork and Pipe** initializes the pipe. Note that if the pipe returns '-1' it failed (simple error checking). Likewise, if the fork returns '-1' it failed.

The **Parent** closes the writing end, **dups** the reading end and closes the superfluous reading end (we did call a function that stands for duplicate). Then it waits for the **child** before calling "wc". The **Child** does the complementary task, closing the reading end, **dups** the writing end, closes the superfluous writing end and then calls "ls". Hence the program should basically replicate:

ls | wc

Some Final Notes

There are some limits on pipes (as you can imagine). Firstly, pipes have a limited capacity. Secondly there are some rules about how big things can be when sent between processes. In essence, you want to split big things into manageable chunks rather than send a giant contiguous bit of data (you can read up on POSIX standards for more information on this).

When a pipe fails (for example when there is no one to read from a pipe) then calling write will trigger an error "EPIPE". This way you can tell if something has gone wrong and handle it. Conversely, if there is no writer, the reader will hang indefinitely. If all the writers are closed, reading will return an **EOF** (this makes sense).

Finally, if there are multiple readers to a pipe… well you can imagine that this may cause problems. A pipe is a queue (first in, best dressed) so there is some logic to it, but you can imagine that processes running at different rates and at the whim of an operating system may give you unpredictable behaviour when they are all gorging at the same buffet.