Lab

Pipes and File Descriptors

Recall that the `bash` shell can link together multiple commands which will feed the output of one command as the input to the next command. For example the command

```
cat /dev/simple_device | grep hello
```

will run `cat` on the file `/dev/simple_device` and pipe ('|') the output to the `grep` command. The `cat` program will copy the contents of the file to `stdout`, which is normally the terminal window. If the `grep` program is not given a filename in which to search, it will read `stdin` to search for the string given (i.e., `hello`). The character '|' instructs the command interpreter to take the output from the first command and pipe it as input to the second command.

The goal of this lab is to write `C` code that will run these commands concurrently and have output from first command be read as input by the second command. That means you must:

- create a parent to run one command
- create a child to run the second command
- create a generic pipe between a parent and child process
- redirect the output from the first command into the pipe
- redirect the input of the second command to read from the pipe.

You should be able to extend your solution for this lab to a solution for performing the piping part of the Shell Assignment. To complete this lab, you must first familiarize yourself with the following system calls.

`fork, execvp`

You should already be familiar with these two from our previous lab. If you skipped this lab, you should go back and complete it right now.

`pipe`

Read the pipe section of the Linux IPC handout, then go through the pipe example code. You should compile and run this code to get a hands on understanding. Note that the piping code in this example does not provide a generic way of piping data for arbitrary processes. For this, we will need the `dup2` system call.

dup2

Next, read the documentation for the `dup2` system call (e.g., you can check out its man page and/or read about it at the end of Chapter 3 in our textbook). Here's a brief description:

```
int dup2(int oldfd, int newfd);
oldfd: old file descriptor whose copy is to be created.
newfd: new file descriptor which is used by dup2() to make the copy.
result: makes newfd be the copy of oldfd, closing newfd first if necessary.
```

Note that if we pass a file descriptor that's already bound (e.g., `stdout`) as `newfd`, then `dup2` will close it then override it. Here is a quick demo:

```c
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>
#include<fcntl.h>

int main()
{
  // do not confuse open() with fopen(): open() is a syscall
  // that returns a file descriptor, not a file.
  int fd = open("foo.txt",O_WRONLY);

  // here the newfd is the file descriptor of stdout (i.e. 1)
  dup2(fd, 1) ;

  printf("I will be written to foo.txt");

  close(fd);
  return 0;
  // We aren't doing any error handling in this demo. Sad!
}
```

Piping `cat` into `grep`

Once you have played with the aforementioned system calls, complete the following steps.

1. Create a text file "bar.txt" that contains some text with the word "foo" in it.
2. In that same directory, run the following command:

   ```
   cat bar.txt | grep foo
   ```

3. Write `C` code using fork, execvp, pipe, and dup2 that carries out the command above.

Here's a primitive ASCII art schematic:

```
(child process) grep foo  <===  ()_____)  <=== cat bar.txt (parent process)
```

The pipe drawn above will be an integer array of size two that contains the file descriptors used for IPC between the parent and the child. Index 0 should be the read end, and index 1 should be the write end. Within your parent process and within your child process you will need to use `dup2`. Recall that the file descriptor associated with `stdin` is 0 and it is 1 for `stdout`. Finally, note the unidirectionality: your child won't be piping anything to the parent (*think about how you would implement bidirectional IPC using pipes*). Because neither the child or the parent will be using both ends of the pipe, you need to be sure to `close` the end of the pipe that you are not using (e.g., the parent should not be using the pipe's input fd).

Checkpoints/Hints

- Your first inclination may be to create a complete solution with all the concepts (fork, pipe, exec, dup) and then try to debug. This is a bad idea because when something goes wrong it will be difficult to know where the problem is within all those calls.
- Instead, break the problem into a number of steps where only one concept is implemented per step. First step for this lab would be to create two processes, one to handle running each of the given programs (`cat`, `grep`). How can you tell if your process creation code works? How will you know which process should run which command? Make sure you place lots of print statements in the code so you can tell the sequence and variable values along the way.
- Once you have the two processes, how can you share a single pipe between them? What is shared between parent and child? You should be able to create a pipe once and share it between processes. How can you know the pipe works? We need it to communicate from one process to the other, so we can write a message into the pipe from one process and read the message from the pipe in the other process. We can use the pipe file descriptors directly to test this functionality.
- Once the pipe is established and we can write and read from it, we need to cause the standard output from one process to use the pipe. Similarly, we need the other process to use the pipe as its standard input. We can use `dup2` to create a clone of the file descriptor of the pipe and place it as the standard input or output. We should implement these one at a time. Redirect one of the pipe ends to a standard in/out descriptor. Now change the code to read/write from standard descriptor. For example, use `fprintf` to write to the pipe directly and change it to use the `stdout` descriptor to write to standard output. Try the same thing with changing from reading the pipe to reading the standard input.

- Once you have both redirections working, you can try spawning another program from each process. But again, use simple programs to spawn. For example, use `cat` to write the message to the pipe. If your second process already is working for reading the pipe and echoing it to output, then it will still work when you change the source of the message. You could also try `cat` as the command for the second process. If you do not give cat a filename, it will try to read standard input and write it to standard output. Make sure the message is read from a file and output sent to the terminal.
- The last step would be to replace the command for the second process to be `grep` with appropriate parameters. Each step has only introduced a single new concept, even when we have two processes. Once we get the basics working, we can add the next more complex concept knowing the underlying concept is already working. Any new errors or bugs will most likely be from the new functionality being added.

A note on `flushing`

File I/O is buffered. That means that when you write a string to output, it may actually sit in a buffer until more data is written. This may pose a problem when needing to see exact timing between two processes. Read up on `flush`, or alternatively, setting the buffer size for a file to zero with `setbuf`. Another common workaround that sometimes works is to use `fprintf` and print to `stderr` instead of `stdout` (UNIX flushes `stderr` more often than `stdout` for obvious reasons).