ENCE360 Lab 2: Processes and Pipes

Objectives

The overall goal of this lab is to introduce you to creating processes using the `fork()` API, and configuring these processes in ways common in user interaction.

The aims of this lab are for you to understand and be able to:

1. Create processes
2. Configure a process's file descriptor tables
3. Create pipes

Preparation

Download and extract your Lab 2 files from `lab2Processes.zip` on Learn. This contains the files `fork.c`, `pipe.c`, `dup2.c`, and `dup2Pipe.c`.

Once you've completed the programs below, make sure you also complete the quiz on the quiz server. These may need you to have finished code ready, so do the programming first!

Program `fork.c`

The system API to create a new process is called `fork()`. When executed by the parent process, `fork()` will create a child process that will have its own address space, that will look identical to that of the parent process. If the call fails, -1 is returned - the code indicating why it fails is placed in `errno`.

First, note the header files that are included; these are needed when creating processes:

```
#include <sys/types.h>
#include <unistd.h>
```

Compile and run `fork.c`, and observe the output generated.

- Can you explain these values? Draw a timeline to help!
- How do they relate to each process's data space?

Now uncomment the line

```
/* waitpid(childPid, NULL, 0); */
```

which will wait for the process with the ID `childPid` (i.e., the child) to exit. Recompile, and execute the modified version `fork.c`. Be sure you understand how and why the behaviour of the program has changed.

Program `pipe.c`

Processes, as well as having their own address spaces, also have their own list of open files, the *file descriptor table*. Any time a process opens a file, this is added to its table.

The program `pipe.c` creates a pipe, then forks, and the parent process sends the first command line argument via the created pipe to the child.

First, compile and execute `pipe.c`. Have a read of the contents and make sure you understand the output. Then, make a copy of `pipe.c` named `pipeToUpper.c`. Modify this new program so that the child receives the message through the pipe, converts it to upper-case using the `toupper()` function, and sends it back through a second pipe. You will need to create a second pipe and plumb everything correctly for this to work.

Program `dup2.c`

File descriptors can be duplicated. For example, when you type `ls > my.file` in the shell, the shell:

1. Opens `my.file`
2. Forks a child
3. The child process then calls the `dup2()` system API to close the `stdout` file descriptor (which usually points to a (pseudo)terminal, such as one found in a terminal window), and replace it with the `my.file` file descriptor.
4. `dup2()` then closes the old `my.file` file descriptor. Now, any writes to `stdout` will go to `my.file` instead of the terminal.
5. Finally, the shell execs the `ls` command. The `ls` command just writes to `stdout` as usual, and the output goes to `my.file`.

`dup2.c` is a file which illustrates the example described above, without the fork. Once you've got your head around this process, compile and execute `dup2.c`. Copy `dup2.c` to `dup2Sort.c`. Modify `dup2Sort.c` to execute the `sort` program instead of `ls`.
Give the sort program the arguments `-k +7` so that the listing is sorted by time, and set up `sort`'s `stdin` to come from a file called `my.file`. This should mean your program does the same task as `sort -k +7 < my.file`.

Make sure you understand the arguments given to the `open()` call!

Program `dup2Pipe.c`

Because each end of a pipe is a file descriptor, `dup2()` works with pipes also. For example, when you type `ls -l | sort -k +8`, the shell does something like this:

1. The shell creates a pipe using the pipe system call.

2. The shell forks a child. The child gets a copy of the parent's file descriptor table.

3. The child uses `dup2()` to close its existing `stdout`, and replace it with the write-end of the pipe. The file descriptors for the read and write end of

the pipe are then closed. Note that the pipe is still in existence, just the pointers to the ends are closed. `stdout` is pointing to the write-end.

4. The shell forks another child. This child does a similar thing to the first child, except with `stdin` and the read-end of the pipe.

5. The first child execs the `ls` command, and the second child execs the sort command. There is an example of this in the file `dup2Pipe.c`, without the extra fork that the shell does for the second child.

We will do something similar with `dup2Pipe.c`. First, compile and execute `dup2Pipe.c` and observe what it does. Then, copy `dup2Pipe.c` to, say, `myDup2Pipe.c`. Modify `myDup2Pipe.c` so that the output of the `sort` command is piped to the command `head -5`. This will require an additional process to be created.

Running the program should now be the same as running the shell command `ls -l |sort -k +9 | head -5`. Note that you ought to include the full path name to the `head`, `sort` and `ls` commands. To find this, use the `which` command. For example, `which ls` should yield `/bin/ls`.