Lab 10 - Pipes and Select

Out: November 17, 2014

1 Introduction

You are a baby. Babies have short attention spans. It's difficult for you to be on top of your CS game when the terminal is always that boring black and white. Besides looking way too 80's for your style, you would like to be able to clearly see when errors crop up. In this lab you will make use of pipes and a `select()` loop (see below) to make things a little prettier.

2 Assignment

In this lab you will be writing a program that colors the output and error streams of an executable. The usage of your program will be as follows:

```
./colorizer <executable> [args]
```

More specifically, it should do the following:

- Setup the pipes to communicate with a child process.
- Fork the child process, then use `dup2()` to redirect the correct ends of the pipes to the child process's input, output and error streams. Close any uneeded file descriptors, then execute the given executable along with additional arguments passed to your program. To run the executable, you can use the function `execvp()`, which takes as parameters the name of the file being executed and the list of arguments to be passed to the executable. **NOTE:** be sure to remove the first argument from `argv` before passing it to `execvp()`; failure to do so can cause some nasty problems.
- Write a select loop to handle input/output to and from the child process. Any input to your program's STDIN should be written to the child process using pipes. Any output on the child's STDOUT should be read from the pipe and colorized green using the provided `print_output()` function. Similarly, output from STDERR should be read and colorized red using `print_error()`. If the input stream from STDIN or any of the output streams is closed (read returns 0), the loop should end and the program should exit cleanly.
- Close all the pipes when finished.

Install the stencil code for this lab by running

```
    cs033_install lab10
```

3 Using `pipe()`

Pipes are used to communicate between processes. They are created in C by using the `pipe` system call:

```
int pipe(int fds[2])
```

This will set up a pipe and set the file descriptors of the pipe in the given two-element array.

```
int my_pipe[2];
if (pipe(my_pipe)) { /* remember to check for system call failures! */
        perror("Pipe failed");
        /* exit or return with an error code */
}
```

If `pipe()` returns successfully, `my_pipe[0]` contains the file descriptor used to read the information from the pipe, and `my_pipe[1]` contains the file descriptor used to write information into the pipe. Pipes are unidirectional, meaning information can only be sent one way. You will need to have two pipes setup if you want bidirectional communication between two processes.

It is good practice to close the ends of the pipe you are not using, so if you are only writing to a pipe in a process, you should close the read file descriptor for that pipe and the process on the recieving end of the pipe should close the write file desciptor. Both ends should be closed when the pipes are no longer needed.

4 Solving The Concurrency Problem

The colorizer must continuously query the child process for changes and update accordingly. However, it also needs to constantly be ready to accept user input, which could come at any time, which presents a problem: normally, a call to `read()` or `fread()` blocks until it reads new data, at which point it returns and the program can continue. If `colorizer` is blocking while waiting for user input, it cannot simultaneously scan for updated process information[1]. You will be adding the code to get user input without blocking the program from updating its data.

4.1 Event-Driven Programming

Fortunately, there is a way around this problem: event-driven programming. Instead of simply reading from standard input and waiting for data to appear, we check whether there is data available at the top of each loop (referred to as an "event loop" here), and then read data if and only if there is data available to read. This allows the program to perform other tasks while waiting for input, interrupting those tasks to read input only when input arrives.

Many event-driven programs can be modeled by the following abstract pseudocode:

[1]It is possible to do this, but not in a single-threaded program. Multi-threaded programs can use multiple threads to block on input from multiple sources, but doing so is unlikely to be the best design choice.

```
while true
        do check for user input
        if input has arrived then
                do <process input>
        end if
        <program body>
end while
```

For a somewhat more concrete example, check out the lecture slide XXVIII-5. Note that this slide is from the networking lectures, and we're not actually doing this lab over a network, but the same concept of a select loop applies.

4.2 Using `select()`

The function you'll be using to check for user input at the top of your event loop is called `select`. This system call, its associated structs, and basic usage is fairly thoroughly described in the `man` pages of the departmental Linux machines if you want more information than is provided in this document.

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
     fd_set *exceptfds, struct timeval *timeout)
```

(Be sure to include `<sys/time.h>`, `<sys/types.h>`, and `<unistd.h>` to have access to the structs and functions necessary for this lab.)

The `select()` function checks on sets of file descriptors to see if they are ready for reading or writing. This means that we need to set up `readfds` to be a `fd_set` containing the `STDIN_FILENO` descriptor and the outputs from the pipes, and that `writefds` and `exceptfds` can both be `NULL`.

To set up `readfds`, declare a `fd_set`, clear it using the macro `FD_ZERO(fd_set *set)`, and fill it using the macro `FD_SET(int fd, fd_set *set)` (note that multiple file descriptors could be added to the set using `FD_SET()`). Be aware that calls to `select()` modify the contents of your `fd_sets`, so to produce correct behavior you'll have to reset your `fd_set` in each iteration of the loop before the call to `select()`.

Since we know what file descriptors we're interested in, we can easily determine what the first parameter, `nfds`. This parameter's value must be one more than the highest file descriptor in any of the sets `readfds`, `writefds`, `exceptfds`.

The final parameter to `select`, `timeout`, specifies how long `select()` should wait if no descriptors are ready when it is called. Since the point of using `select()` in our case is to avoid blocking, we can use a null pointer for `timeout`. Note that on Linux this struct would also modified by a call to `select()`, so you would have to re-fill it before every call to `select()` too.

Now you're finally ready to call `select`! Be sure to check if the return value is -1 to see if there was an error. If `select` completed successfully, it will return the number of ready file descriptors. When `select` indicates that `STDIN_FILENO` is ready, go ahead and call read - check for this using the macro `FD_ISSET(int fd, fd_set *set)` on `readfds` after `select()` has returned. Similarly, this can be done for the other file descriptors.

5 A `dup2()` Reminder

As a reminder from Shell 2, the type signature of dup2 is as follows:

```
dup2(int targetfd, int srcfd)
```

After `dup2()` is called, `srcfd` is closed and becomes a copy of `targetfd`. Thus, if we wanted to route all information sent to `STDOUT` to a file with file descriptor `fd`, the following call would be sufficient:

```
dup2(fd, STDOUT_FILENO)
```

6 Getting Checked Off

Once you're finished, please have a TA come over and show your work. To help verify your programs correctness, here's some commands you can try:

- `./colorize bash`
- `./colorize bc`
- `./colorize cs033_shell_2_demo`