CS 333: Operating Systems Lab
Autumn 2018

Lab 3: morse code and turtles

Goal

In this lab you will learn about handling signals and to build a simple interactive shell of your own.

Before you begin

- *You must use C for this lab. You may use C++ for the string-handling parts of the assignment.*
- System calls of interest: `fork`, `exec`, `wait`, `kill`, `signal`, `chdir` and their variants.

morse code

In this part we will work with writing custom signal handlers of different types.

0. Read about *signals* and processes.
   `man signal`, `man 7 signal`,
   http://www.alexonlinux.com/signal-handling-in-linux
   http://www.cs.princeton.edu/courses/archive/spr06/cos217/lectures/23signals.pdf

   Refer to the sample output folder for the format of outputs for each of the tasks.

1.a Write a program (`p1a.c`) to handle the `SIGINT` and `SIGTERM` signals. The process should print a custom message asking a question about whether the program should really exit, and exit only on confirmation. Look up accompanying files for sample outputs.

1.b Write a program (`p1b.c`) that spawns a number of child processes, using the following command:
`./p1b <n>`
where n is the number of child processes. Each child process sleeps for a random duration and exits. Override the `SIGCHLD` signal handler in the parent process and print the sequence of exits of the child processes.

1.c *What would happen if a child process forked another (grand) child process, which eventually terminated.* Recreate this situation (`p1c.c`), check for output and reason about it. Two situations to test—when the grandchild process terminates before the child and when the child terminates before the grandchild. Write your observations and reasoning as part of the submission. (Note that this requires handling the SIGCHILD signal.)

turtles (and shells)

Write a program `turtle.c` that implements a simple command shell for Linux. Sample code `make_tokens.c` is provided to tokenize strings (to deal with user commands); you may reuse this as part of your shell. Use any creative message as a prompt of the shell waiting for user commands. Below are the the commands you need to implement in the shell, and the expected behavior of the shell for that command.

Note: You must use the `fork` and `exec` system calls to implement the shell. The idea is for the main program to act as a parent process that accepts and parses commands and then instantiates child processes to execute the desired commands.

You must **not** use the `system` function provided by the C-library. Also, you must execute Linux system programs wherever possible, instead of re-implementing functionality. For example: Given the following command
`echo "India has a gold medal in a track event at the World Athletics Championship"`
you should use the `echo` binary available rather than implementing `echo`.

The following functionality should be supported:

- `cd directory-name` must cause the shell process to change its working directory. This command should take one and only one argument; an incorrect number of arguments (e.g., commands such as `cd`, `cd dir1 dir2` etc.) should print an error in the shell. `cd` should also return an error if the change directory command cannot be executed (e.g., because the directory does not exist). For this, and all commands below, incorrect number of arguments or incorrect command format should print an error in the shell. After such errors are printed by the shell, the shell should not crash. It must simply move on and prompt the user for the next command.

- All simple standalone built-in commands of Linux e.g., (`ls`, `cat`, `echo`, `sleep`) should be executed, as they would be in a regular shell. All such commands should execute in the foreground, and the shell should wait for the command to finish before prompting the user for the next command. Any errors returned during the execution of these commands must be displayed in the shell.

- Any simple Linux command followed by the output redirector `'>'` (e.g., `echo hi > hi.txt`) should cause the output of the command to be redirected to the specified output file. The command should execute in the foreground. An incorrect command format must print an error and prompt for the next command.
  Note: This can achieved by manipulating the sequence of file open and close actions, and/or using the dup system call.

- Any list of simple Linux commands separated by "`;;`" (e.g., `sleep 100 ;; echo hi ;; ls -l`) should all be executed in the **foreground** and **sequentially** one after the other. The shell should start the execution of the first command, and proceed to the next one only after the previous command completes (successfully or unsuccessfully). The shell should prompt the user for the next input only after all the commands have finished execution. An error in one of the commands should simply cause the shell to move on to the next command in the sequence. An incorrect command format must print an error and prompt for the next command.

Important Guidelines:

- When a process completes its execution, all of the memory and resources associated with it are de-allocated so they can be used by other processes. This cleanup has to be done by the parent of the process and is called `reaping the child process`.The shell must also carefully reap all its children that have terminated. For commands that must run in the foreground, the shell must wait for and reap its terminated foreground child processes before it prompts the user for the next input.

- By carefully reaping all children (foreground and background), the shell must ensure that it does not leave behind any zombies or orphans when it exits.

- You must implement all the commands above in your shell. Test your shell using several test cases, and record observations and reasoning in your report.

Tips

- You are given a sample code `make-tokens.c` that takes a string of input, and "tokenizes" it (i.e., separates it into space-separated commands). You may find it useful to split the user's input string into individual commands.

- You may assume that the input command has no more than 1024 characters, and no more than 64 "tokens". Further, you may assume that each token is no longer than 64 characters.

- You may want to build a simple shell that runs simple Linux built-in commands first, before you go on to tackle the more complicated cases.

- You will find the `dup` system call and its variants useful in implementing I/O redirection and pipes. When you `dup` a file descriptor, be careful to close all unused, extra copies of the file descriptor. Also recall that child processes will inherit copies of the parent file descriptors, so be careful and close extra copies of inherited file descriptors also. Otherwise, your I/O redirection and pipe implementations will not work correctly.

- You must catch the SIGINT signal in your shell and handle it correctly, so that your shell does not terminate on a `Ctrl+C`, but only on receiving the exit command.
- You will find the `chdir` system call useful to implement the `cd` command.
- Carefully handle all error cases listed above for each command. For example, an incorrect command string should always print an error.

Submission Guidelines

- All submissions via moodle. Name your submission as: **<rollno_lab3>.tar.gz**
- The tar should contain the following files in the following directory structure:

```
<roll_number_lab3>/
|__morse-code/
|____p1a.c
|____p1b.c
|____p1c.c
|____p1c.txt
|__turtles/
|____turtle.c
|____report.txt
|____output.txt
```

- Your code (`turtle.c`) should be well commented, indented and easily readable.
- The `report.txt` should explain briefly how you built the shell
- The README contains test-cases (some with expected outputs and some not). You should try all the test-cases using your own shell. The file `output.txt` should show results for sample runs of your code using the test-cases mentioned as well as your own test-cases.
- We will evaluate your submission by reading through your code, executing it over several test cases, and by reading your report.
- **Deadline:**
  Monday 30th July 2017 5.00 PM via moodle.

self-study of pipes

- `pipe` is a system call to create an object for inter process communication via a pair of files descriptors. One of the descriptors (file descriptor #0) is the read end and the other descriptor is the write end (file descriptor to be used for writes).
- `dup` is a system call to create a copy of a file descriptor. It uses the lowest unused descriptor in the per process file descriptor table for the new process. `dup2` is a variant of `dup` system call.
- `man 2 pipe`, `man 2 dup`
- `pipes.c` and `dup.c` are sample programs as part of the lab for self study.
  Study it and look up other examples and usages of these two system calls.
- **No submission required**.