Programming Assignment 4
- Extra Credit

Due by 11:59pm, Friday, May 15 2015

This assignment has three goals:

1. To observe the operating systems' (OS') activities at the user-level;
2. To implement an OS command interpreter (shell);
3. To use some system calls (such as `fork()` and `exec()`).

This assignment contains the following two parts: 1) observing the OS through the `/proc` file system; 2) building a shell. Everything you do in this assignment is at the user-level (outside of the OS kernel). You may work in groups of 2.

Part I: Observing the OS through the `/proc` file system

The OS is a program that uses various data structures. Like all programs in execution, you can determine the performance and other behavior of the OS by inspecting its state - the values stored in its data structures. In this part of the assignment, we study some aspects of the organization and behavior of a Linux system by observing values of kernel data structures exposed through the `/proc` virtual file system.

The `/proc` virtual file system:
Linux uses the `/proc` file system to collect information from kernel data structures. The `/proc` implementation provided with Linux can read many different kernel data structures. If you `cd` to `/proc` on a Linux machine, you will see a number of files and directories at that location. Files in this directory subtree each corresponds to some kernel data structure. The subdirectories with numeric names contain virtual files with information about the process whose process ID is the same as the directory name. Files in `/proc` can be read like ordinary ASCII files. You can open each file and read it using library routines such as `fgets()` or `fscanf()`. The `proc (5)` manual page explains the virtual files and their content available through the `/proc` file system.

Requirements in detail:
In this part, you are asked to write a program to report the behavior of the Linux kernel. Your program should run in two different versions. The default version should print the following values on `stdout`:

- Processor type
- Kernel version
- The amount of memory configured into this computer
- Amount of time since the system was last booted

A second version of the program should run continuously and print lists of the following dynamic values (each value in the lists is the average over a specified interval):

- The percentage of time the processor(s) spend in user mode, system mode, and the percentage of time the processor(s) are idle
- The amount and percentage of available (or free) memory
- The rate (number of sectors per second) of disk read/write in the system
- The rate (number per second) of context switches in the kernel
- The rate (number per second) of process creations in the system

If your program (compiled executable) is called `proc_parse`, running it without any parameter should print out information required for the first version. Running it with two parameters "`proc_parse <read_rate> <printout_rate>`" should print out information required for the second version. `read_rate` represents the time interval between two consecutive reads on the `/proc` file system. `printout_rate` indicates the time interval over which the average values should be calculated. Both `read_rate` and `printout_rate` are in seconds. For instance, `proc_parse 2 60` should read kernel data structures once every two seconds. It should then print out averaged kernel statistics once a minute (average of 30 samples). The second version of your program doesn't need to terminate.

Part II: Building a shell

UNIX shells:
The OS command interpreter is the program that people interact with in order to launch and control programs. On UNIX systems, the command interpreter is often called **shell**: a user-level program that gives people a command-line interface to launching, suspending, and killing other programs. `sh`, `ksh`, `csh`, `tcsh`, `bash`, ... are all examples of UNIX shells. You use a shell like this every time you log into a Linux machine at a URCS computer lab and bring up a terminal. It might be useful to look at the manual pages of these shells, for example, type "`man csh`".

The most rudimentary shell is structured as the following loop:

1. Print out a prompt (e.g., "`CPSC351Shell$ `");
2. Read a line from the user;
3. Parse the line into the program name and an array of parameters;
4. Use the `fork()` system call to spawn a new child process;
   - The child process then uses the `exec()` system call (or one of its variants) to launch the specified program;
   - The parent process (the shell) uses the `wait()` system call (or one of its variants) to wait for the child to terminate;
5. Once the child (the launched program) finishes, the shell repeats the loop by jumping to 1.

Although most commands people type on the shell prompt are the names of other UNIX programs (such as `ps` or `cat`), shells also recognize some special commands (called internal commands) that are not program names. For example, the `exit` command terminates the shell,

and the `cd` command changes the current working directory. Shells directly make system calls to execute these commands, instead of forking a child process to handle them.

Requirements in detail:
Your job is to implement a very primitive shell that knows how to launch new programs in the foreground and the background. It should also recognize a few internal commands. More specifically, it should support the following features.

- It should recognize the internal commands: `exit`, `jobs`, and `cd`. `exit` should use the `exit()` system call to terminate the shell. `cd` uses the `chdir()` system call to change to a new directory.
- If the command line does not indicate any internal commands, it should be in the following form:
    <program name> <arg1> <arg2> .... <argN> [&]
  Your shell should invoke the program, passing it the list of arguments in the command line. The shell must wait until the started program completes unless the user runs it in the background (with &).

To allow users to pass arguments you need to parse the input line into words separated by whitespace (spaces and '\t' tab characters). You might try to use `strtok_r()` for parsing (check the manual page of `strtok_r()` and Google it for examples of using it). In case you wonder, `strtok_r()` is a user-level utility, not a system call. This means this function is fulfilled without the help of the operating system kernel. To make the parsing easy for you, you can assume the '&' token (when used) is separated from the last argument with one or more spaces or '\t' tab characters.

The shell runs programs using two core system calls: `fork()` and `execvp()`. Read the manual pages to see how to use them. In short, `fork()` creates an exact copy of the currently running process, and is used by the shell to spawn a new process. The `execvp()` call is used to overload the currently running program with a new program, which is how the shell turns a forked process into the program it wants to run. In addition, the shell must wait until the previously started program completes unless the user runs it in the background (with &). This is done with the `wait()` system call or one of its variants (such as `waitpid()`). All these system calls can fail due to unforeseen reasons (see their manual pages for details). You should check their return status and report errors if they occur.

You can get a working shell from <u>here</u> (executable on Linux x86 platforms) and play with it. **Note that this is not intended to be a complete solution. It does not support some features that you are asked to implement.**

No input the user gives should cause the shell to exit (except when the user types `exit` or `Ctrl+D`). This means your shell should handle errors gracefully, no matter where they occur. Even if an error occurs in the middle of a long pipeline, it should be reported accurately and your shell should recover gracefully. In addition, your shell should not generate leaking open file descriptors. **Hint:** you can monitor the current open file descriptors of the shell process through the `/proc` file system.

Pipes:
Your shell needs to support pipes. Pipes allow the `stdin`s and `stdout`s of a list of programs to be concatenated in a chain. More specifically, the first program's `stdout` is directed to the `stdin` of the second program; the second program's `stdout` is directed to the `stdin` of the third program; and so on so forth. Multiple piped programs in a command line are separated with the token "|". A command line will therefore have the following form:

    <program1> <arglist1> | <program2> <arglist2> | ... | <programN> <arglistN> [&]

Try an example like this: pick a text file with more than 10 lines (assume it is called `textfile`) and then type

```
cat textfile | gzip -c | gunzip -c | tail -n 10
```

in a regular shell or in <u>the working shell</u> we provide. Pause a bit to think what it really does. Note that multiple processes need to be launched for piped commands and all of them should be waited on in a foreground execution. The `pipe()` and `dup2()` system calls will be useful.

Administrative policies

A note on the programming language:
C/C++ is the only choice for this assignment. We are not alone in this. Most existing operating system kernels (Linux and other UNIX variants) themselves are written in C; the remaining parts are written in assembly language. Higher-level languages (Java, Perl, ...), while possible, are less desirable because C allows more flexible and direct control of system resources.

Turn-in:

- You may work in groups of 2.

- You must firstly complete the design of your program and document your program design well (Resource for Design Part:
  http://www.tutorialspoint.com/software_engineering/software_design_basics.htm

  http://www.tutorialspoint.com/software_engineering/software_analysis_design_tools.htm
).

- Write a READ ME file (text file, do not submit a .doc file) which contains
  - − Your name(s) and email address.
  - − The programming language you used (i.e. C or C++).
  - − How to execute your program.
  - − Describe the names of the two executables, special compiling instructions, or anything else special you want to let us know.
  - − How did your team collaborate on your projects? Illustrate each member's contribution.
  - − Anything special about your submission that we should take note of.
- You are asked to electronically turn in your source files, a makefile and the design document of your program electronically (do not submit .o or executable code) through **TITANIUM** before deadline. After compiling, we should see two executables (one for Part I and one for Part II).

Grading guideline: will be penalized heavily for failure to pass tests. You should indicate clearly in your README what tests you conducted.

- 10%: design of your program (write in a document called "Design of Assignment 4").
- 10%: version 1 of the `/proc` parser.
- 20% version 2 of the `/proc` parser.
- 20% the basic shell supporting foreground and background execution, including `bg` and `fg` manipulation of individual processes, with reassignment of controlling terminal to the foreground job.
- 20%: your shell supports pipes.
- 15%: your shell handles errors gracefully.
- 5 %: README file
    - List your team members' Names and Emails
    - How to run your program (platform, commands in each steps and a screenshot of one testing)
    - Describe the names of the two executables, special compiling instructions, or anything else special you want to let us know.
    - How did your team collaborate on your projects? Illustrate each member's contribution.

Late turn-in policy:
Late turn-ins will be accepted for up to one day, with 10% penalty for each late day.

No turn-ins more than one day late will be accepted.

Academic Honesty:

Academic Honesty: All forms of cheating shall be treated with utmost seriousness. You may discuss the problems with other students, however, you must write your **OWN codes and solutions**. Discussing solutions to the problem is **NOT** acceptable (unless specified otherwise). Copying an assignment from another student or allowing another student to copy your work **may lead to an automatic F for this course**. Moss shall be used to detect plagiarism in programming assignments. If you have any questions about whether an act of collaboration may be treated as academic dishonesty, please consult the instructor before you   collaborate.   Details   posted   at http://www.fullerton.edu/senate/documents/PDF/300/UPS300-021.pdf.