CS 3013 Operating Systems

WPI, C Term 2007

Project 1 – LINUX Process Management

This project is to be done by each individual student. It is NOT a group project.

Introduction

This assignment is intended to introduce you to the process manipulation facilities in the Unix/Linux Operating System. You are to implement the program described below on the Fossil Lab machines or any CCC Linux machine.

Command Execution

You are to write a program doit that takes another command as an argument and executes that command. For instance, executing:

% doit cat /etc/motd

would invoke the cat command on the file /etc/motd, which will print the current "message of the day." After execution of the specified command has completed, *doit* should display statistics that show some of the system resources the command used. In particular, *doit* should print:

PROCESS STATISTICS
1. The amount of CPU time used (both user and system time) (in milliseconds),
2. The elapsed "wall-clock" time for the command to execute (in milliseconds),
3. The number of times the process was preempted involuntarily (e.g. time slice expired, preemption by higher priority process),
4. the number of times the process gave up the CPU voluntarily (e.g. waiting for a resource),
5. The number of page faults, and
6. The number of page faults that could be satisfied from the kernel's internal cache (e.g. did not require any input/output operations).

Basic Command Shell

Satisfactory completion of the command execution portion of this assignment is worth 10 of the 15 points. For three additional points, your program should be extended to behave like a shell program if no arguments are given at the command line (it should work as before if arguments are given on the command line). Your program should continually prompt for a command (which may have multiple arguments separated by white space) then execute the command and print the statistics. This work will involve breaking the line of text you read into an argument list. Your program should handle two "built-in" commands, which are handled internally by your shell.

- **exit**—causes your shell to terminate.
- **cd** dir—causes your shell to change the directory to dir.

Your program should also exit if an end-of-file is detected on input. You may assume that a line of input will contain no more than 128 characters or more than 32 distinct arguments. Note: you may not use the system call system available in Unix/Linux to execute the entered command. A sample session is given below with comments given in "<>". The doit prompt is given by "➔", the regular shell prompt by "%".

Output Example 1

```
% doit
➔cat /etc/motd
            < print the current message of the day >
            < statistics about this cat command >
➔cd dir
            < current directory is changed to dir (if successful) >
            <Note there are NO stats printed since this is an internal
            command>
➔ls
            < listing of files in the current directory >
            < statistics about this ls command >
➔exit
%           < back to the shell prompt >
```

Background Tasks

For the final two points of the project, you need to extend your basic command shell to handle background tasks. A background task is indicated by placing an ampersand ('&') character at the end of an input line. When a task is run in background, your shell should not wait for the task to complete, but immediately prompt the user for another command. Note that any output from the background command will be directed to the terminal display and will intermingle with output from your shell and other commands. With background tasks, you will need to modify your use of the wait() system call so that you check the process id that it returns. The returned process id might correspond to a background task rather than the currently invoked foreground task. In this case, your shell should print out the process id of the completed background task along with the command name. You also need to add an additional built-in command to your shell:

- jobs—lists all background tasks

A sample session with background tasks is shown with comments given in <>.

Output Example 2

```
% doit
➔numbercrunch &
[1] 12345               < Indicate background task and its process id >
➔jobs
[1] 12345 numbercrunch   < Print process id and command name for tasks
>
➔ls                     < Listing of files in the current directory >
                        < Statistics about this ls command >
➔cat /etc/motd
[1] 12345 Completed < indicate that the background jobs is complete >
                        < Statistics about this numbercrunch command >
                        < Print the current message of the day >
                        < Statistics about this cat command >
➔exit
%                       < Back to the shell prompt >
```

If the user tries to exit the shell before all background tasks have completed then your shell should refuse to exit and wait() until these tasks have completed. You should also observe how your mini-shell works in comparison to a regular Unix/Linux shell. Does it have all the same features? What limitations does it have? You should include your observations as a comment in your code that is turned in.

Helpful Hints For All Aspects Of The Project

The following system calls might be useful:
• fork() — create a new process.
• getrusage() — get information about resource utilization. Note: not all versions of Unix/Linux kernels fill in all fields of the rusage structure. Specifically it is known that Linux systems always return zero for voluntary and involuntary context switches. Your program should simply report what is returned.

• gettimeofday() — get current time for calculation of wall-clock time.

• execve() — execute a file. The library routine execvp() may be particularly useful.

• wait() — wait for process to terminate.

• chdir() — change the working directory of a process.

• strtok() — help in parsing strings.

To get help information about these routines, use the Unix/Linux "man" command. For instance, entering "man fork" will display the manual page entry for fork on the terminal. The manual pages are organized into sections. Section 1 is for UNIX commands, Section 2 is for system calls and Section 3 is for library routines. Some entries are contained in more than one section. For example to obtain information about the system call wait() (rather than the command wait) use "man 2 wait" where the section is explicitly given.

Submission of Assignment

Use the *turnin* command to submit your project with the project name of proj1. the executable should be producible from the source by typing "make" – this means that in this directory there will be a file named "Makefile" that tells the utility "make" how to build the executable.

CS3013 – Operating Systems
Project 1 Evaluation Sheet

Name:

Basic Commands: **Total of 50% of credit**

Handles any arbitrary command
Internally handles "cd"
Internally handles "exit"
Exits on "EOF" (^D)
Prints requested statistics as given in the table titled "Process Statistics".
Does not use Unix call "System"

Behaves like Shell, Prompting for commands: **Total of 20% of credit**

Behavior looks like that shown in Output Example 1

Background Tasks Capability: **Total of 20% of credit**

Starts jobs in the background when given "&".
Behaves as shown in Output Example 2. – Note especially that statistics on the background task are produced only after that task has finished; then it executes the foreground request.
Handles built-in command "jobs"
Brief write-up providing comparisons to regular Linux/Unix shell can be found in the code comments.
Program does not exit if a background task is running.

Auxiliary Items: **Total of 10% of credit**

The project is submitted with turning and is in a directory called "proj1".
When we cd into proj1, we can type "make" and the executable is produced from the source(s).
The code has comments and is structured – it's not spaghetti code.