

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 system resources the command used. In particular, *doit* should print:

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 `<>`.

```
% 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) >
==>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 putting 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 given below with comments given in `<>`.

```

% 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

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*.