Project 1: Command Shell

Introduction

This project is intended to introduce you to the process manipulation facilities in operating systems such as Linux, Unix, or Windows. You will implement a simple command shell capable of supporting concurrently executing commands and printing out statistics about each one. The project will be carried out in three parts, each part building upon the previous one.

- The first part, called `doit`, simply executes a single command and prints statistics about it.
- The second part, called `shell`, reads and parses input lines from a terminal or a file and executes each line as a separate command, much like `doit`.
- The third part, called `shell2`, is an extension of `shell` in which a command that is terminated with an `&` character is executed in the background while additional commands are read and executed.

Although this project could be carried out on any Linux system of recent vintage, it will be evaluated and graded on the guest system of the virtual machine distributed in class. Error messages and subtle details of behavior often vary among Linux versions. It is strongly suggested that you implement it (or at least test it) on the OpenSUSE 11.4 system of your virtual machine that you installed as part of Project 0.

Part 1: Command Execution (3 points)

Write a program in C called `doit` that takes another command as an argument, executes that command, and prints statistics about it. For instance, executing

```
% doit ls -l /usr/src/linux
```

invokes the `ls` command to list the directory `/usr/src/linux`.

To execute a command, you must follow the traditional Unix practice of forking a new process and causing that new process to execute the command with its arguments using one of the variants of `exec()`. The command executes in the child process, and the parent process, `doit`, waits for the child to terminate. It then collects and prints statistics about the child. The following statistics should be collected and printed:

1. the elapsed “wall-clock” time for the command to execute in milliseconds,
2. the amount of CPU time used (both user and system time) in milliseconds,
3. the number of times the process was preempted involuntarily (e.g., time quantum expired, preemption by higher priority process, etc.),
4. the number of times the process gave up the CPU voluntarily (e.g., waiting for an I/O or resource),

5. the number of page faults, and

6. the number of page faults that could be satisfied from the kernel’s internal.

See below for how to get the information for these statistics.

If the command is illegal, doit should complain. Note that this part of the assignment does not require you to read and parse the command line, because all of the arguments will have been previously parsed and passed to the main() function as part of argv.

Satisfactory completion of this part is worth three of the ten points of the project.

Helpful hints

One of the purposes of this assignment is for you to learn how to find information in the online documentation of Unix and Linux (called man pages) and, from that documentation, to learn how to invoke the various system facilities from your program.

For example, to learn about the fork() function, type

```bash
% man fork
```

to your favorite Unix or Linux shell. Manual pages are organized into sections. Section 1 is for commands to the shell, section 2 is for system calls, and section 3 is for library routines, etc. Some entries are contained in more than one section. For example,

```bash
% man wait
```

will give you the manual page for the wait command typed to a shell, while

```bash
% man 2 wait
```

will give you the manual page for the wait() system call.

```bash
% man man
```

tells you how to use the man command to view and/or print manual pages.

For this part of the assignment, the following systems calls and library functions are needed:

- fork() — create a new process by cloning an existing one
- execvp() or one of its variants – execute a file. This is a front-end for the system call execve(), which replaces the current process with a new program to execute.
- wait() – wait for a process to terminate.
- getrusage() – get information about resource utilization.
- gettimeofday() – get current time for calculation of wall-clock time.

Note: The getrusage() function returns a data structure with a lot of fields in it. However, the man pages say that only some of these fields are actually populated by the Linux kernel.
Part 2: Basic Command Shell (3 points)

Make a copy of your `doit` program and call it `shell`. Adapt this program so that it operates in a loop. During each iteration, `shell` prompts for and reads a new line of text from `stdin`. It then parses the line to divide it into tokens and places each token into an argument vector. Under normal conditions, `shell` should `fork()` a new child process to execute that command with the accumulated arguments. It should then `wait` for that child to terminate and print statistics about that child in the same way that `doit` did.

Two “built-in” shell commands, however, are treated differently. These are:-

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

These must not be `forked` and executed in child processes. Instead, they must be executed “in line” in the `shell` process itself, because their effects need to persist to subsequent commands.

A requirement of this assignment (to assist the grader) is that `shell` must be capable of accepting its input from the standard input stream `stdin`. Normally, `stdin` is connected to the keyboard input of the terminal window. However, it is also possible in Linux to pipe a file to `stdin` as follows:-

```
% shell < file.txt
```

This exposes the need to handle the end-of-file condition. When the end of file is detected in `stdin`, `shell` should act as if `exit` had been typed.

Note that the Linux `man` pages explicitly state that `getrusage()` returns the *cumulative* statistics for all children of a process, not just the statistics for the most recent child. Therefore, you must keep a record of the statistics of previous children. When you call `getrusage()` after a particular child has terminated, you must subtract the previous statistics from the most recent ones returned by `getrusage()` in order to find out how many resources that the particular child used.¹

To keep things simple, you may design your `shell` for lines of input containing not more than 128 characters and not more than 32 distinct arguments. You should print an error if this condition is violated.

A sample session of a `shell` is given below, with comments in /*...*/. The prompt of this example is `===>`, but you may use any prompt character(s) you wish.

```
% shell
===>cat /etc/motd
    /* print the text of motd – i.e., the current message of the day */
    /* print statistics about the cat command */
===>cd dir
    /* current directory is changed to dir. No statistics because this is a built-in command of the shell */
```

¹ An alternative is for `shell` to invoke `doit` after parsing the command. `doit` prints the c statistics.
/* listing of files in the Linux source directory */
/* statistics about this ls command */
==>[4] exit
% /* back to your original terminal window */

Satisfactory completion of this part of the project is worth three of the ten points.

Helpful hints
You will need the following two functions:–

- chdir() – change working directory
- strtok() – search for tokens in the input string

Part 3: Command Shell with Background Execution (3 points)

Make a copy the shell program of part 2, and call this copy shell2. Modify shell2 to handle background tasks, which are indicated by ampersand ('&') characters at the ends of input lines. When a task is run in background, shell2 should not block until the child process completes. Instead, it prompts the user for another command. As a result, there may be multiple child processes active at once, even while shell2 is reading additional commands. Moreover, a background task can terminate at any time; when it does, shell2 should display a message that the particular task has terminated, and it should follow that message with the statistics about the command of that task.

In addition to the built-in commands of shell, shell2 must also handle the following build-in command:–

- jobs – lists all background tasks currently active

Note that output from background commands that is directed to the terminal window may intermingle with the output of other commands and with the output from your shell2.

A sample session with background tasks is given below.

% shell2
==>[1] 12345 make -j4 O=~/kernelDst &
[1] 12345 /* indicate background task #1 and process id */
==>[1] jobs
[1] 12345 make
/* print process id and command name for each task */
==>[1] cat /etc/motd
/* display message of the day */
/* statistics about this cat command */
/* print statistics about background make command */
/* output of ls command */
/ * statistics about this ls command */

==>/exit
% /* back to Unix prompt */

If the user tries to exit shell2 before all background tasks have completed, then shell2 should refuse to exit, print a message, and wait() for those tasks to be completed.

Satisfactory completion of this part of the project is worth three of the ten points.

As part of the write-up describing your program, you must explain how you keep track of outstanding processes in shell2 — i.e., the data structures and algorithms for maintaining information about outstanding commands that have not been completed.

**Helpful hints**

The following two functions will be very useful:

- **wait3()** – lets you wait for *any* child process; returns rusage statistics
- **wait4()** – lets you wait for a specific child process; returns rusage statistics

Either of these functions can be called with the **WNOHANG** option, which causes the **wait()** function to not block but rather return with an error code.

A suggested approach to handling background tasks is as follows. After forking a child process to invoke a background command (i.e., with a ' &' character at the end), go into a loop using **wait3(WNOHANG)** to wait for *any* child to finish. If **wait3()** returns information about a child process that has finished, print its statistics and repeat the loop. However, if **wait3()** indicates that no child process has finished lately, exit the loop and prompt for the next command.

In the case that a command is *not* a background process (i.e., does not end with a ' &' character), then you should use a **wait3()** loop *without the WNOHANG argument*. This will pick up any previous background commands that may have completed. Once the non-background task has been waited for, loop again using **wait3(WNOHANG)** to pick up any remaining tasks that have finished. When **wait3(WNOHANG)** returns with an error, then prompt for the next command.

**Submission of Assignment**

Submit your assignment for grading via Turnin as directed in class. The Turnin system can be accessed at

[https://turnin.cs.wpi.edu:8088/](https://turnin.cs.wpi.edu:8088/)

This assignment is called **Project 1**. Your submission should include

1. All of the files containing the code for all parts of the assignment.
2. One file called **Makefile** that can be used by the **make** command for building the three executable programs. It should support the “**make clean**” command, “**make all**” and **make** each of the three programs individually.
3. The test files or input that you use to convince yourself (and others) that your programs actually work.
4. Output from your tests.

5. A document called README.txt, README.doc, or README.pdf explaining your project and anything that you feel the instructor should know when grading the project. In particular, describe the data structure and algorithm you used to keep track of background jobs. Also, explain how you tested your programs. Write-ups in MS Word or PDF format are preferred.

It is helpful to zip all of the files together. Please use a zip program to create a file with the .zip extension; the professor cannot expand files with the .rar extension. Be sure to put your name at the top of every file you submit!

**Grading Rubric**

Each part of this assignment will be graded separately. Points will be assigned to each part as follows:

- One point for compiling the program of that part with no warnings.
- One point for correctly executing the grader’s test cases.
- One point for submitting your own test case(s) that comprehensively test your program

In addition, one point will be awarded for a satisfactory write-up explaining your submission.

If your `Makefile` does not generate a working program for a part of this assignment, that part will not be graded and no credit will be awarded for it. However, the graders will attempt to contact you to let you know of such a problem so that you can resubmit the part.

**Individual Assignment**

This is an individual project, not a team project. However, you may discuss the project with each other and work out one or more joint solutions. After you have worked out a joint solution, code up the project in your own style and write up the explanation in your own words.

Direct copying of each other or of solutions by others is a violation of the WPI Academic Honesty Policy.

Be sure to test your project on a virtual machine of this course, so that you are confident that it compiles correctly without warnings.