CS 3013 - Operating Systems

Project 1 (100 points)
Assigned: Friday, August 30, 2013
Checkpoint: Wednesday, September 4, 2013
Due: Tuesday, September 10, 2013

Project 1: Linux Shell

Many popular operating systems allow users to enter commands and control applications through a keyboard-driven *shell* interface. In this project, students will implement a shell in multiple phases, with each phase building upon the prior one. This will allow students to learn about process creation, termination, and resource usage in the Linux operating system. All coding is to be done in the C programming language. We highly recommend implementing or testing the project on the Ubuntu virtual machine you created in Project 0. Projects that do not compile or run correctly on the Ubuntu virtual machines may be penalized. This project is worth 100 points.

Phase 1: Single Command

The first phase program, called `runCommand`, reads in its own command line arguments, treating the first argument as a program to be executed. The runCommand program must execute the command and pass the additional arguments as parameters to the invoked command. For example, the command:

```
picard@enterprise:~> ./runCommand ls /home
```

will search the user's `$PATH` variable and invoke the proper `ls` command passing `/home` as the argument. This causes the listing of the home directory's contents. To do so, you must `fork()` off a child process and execute the command and its arguments using the `exec` system call. The parent process should wait for completion and then print statistics about the child process's execution. These statistics must include:

- the elapsed "wall-clock" time for the command to execute in milliseconds,
- the amount of CPU time used (both user and system time) in milliseconds,
- the number of times the process was preempted involuntarily (time quantum expired, preemption by higher priority process, etc.),
- the number of times the process gave up the CPU voluntarily (waiting for I/O or a resource),
- the number of page faults, and
- the number of page faults that could be satisfied using unreclaimed pages.

If the command is illegal, an error message must be generated. Satisfactory completion of this component is worth 20% of the project score.

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 "`man fork`" in 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, "`man wait`" will give you the manual page for the `wait` command typed to a shell, while "`man 2 wait`" will give you the manual page for the `wait()` system call. Executing "`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 may be 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.

Phase 2: Basic Shell

Make a copy of your `runCommand` 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 `runCommand` did.

However, two "built-in" shell commands 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:

```
picard@enterprise:~> ./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.

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.

```
picard@enterprise:~> ./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 */
==> ls /usr/src/linux
/* listing of files in the Linux source directory */
/* statistics about this ls command */
==> exit
picard@enterprise:~>      /* back to your original terminal window */
```

Helpful hints

You may need the following two functions:

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

Phase 3: Command Shell with Background Execution

Make a copy of the `shell` program from Phase 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. In this case, `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 built-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.

```
picard@enterprise:~> shell2
==> make -j4 O=~/kernelDst &
[1] 12345  /* indicate background task #1 and process id */
==>jobs
[1] 12345 make
/* print process id and command name for each task */
==> cat /etc/motd
/* display message of the day */
/* statistics about this cat command */
==> ls /usr/src/linux
[1] 12345 make completed.
/* print statistics about background make command */
/* output of ls command */
/* statistics about this ls command */
==> exit
picard@enterprise:~>  /* back to regular 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. 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 maintain information about outstanding commands that have not been completed.

Helpful hints

The following two functions may be 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 (e.g., "nobody is ready to be waited on yet").

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.

Checkpoint Contributions

Students must submit work that demonstrates substantial progress towards completing the project on the checkpoint date. Substantial progress is judged at the discretion of the grader to allow students flexibility in prioritizing their efforts. However, as an example, any assignment in which the first phase is completed and has a partial implementation of interactive shell will be considered as making substantial progress. **Projects that fail to submit a checkpoint demonstrating significant progress will incur a 10% penalty during final project grading.**

Deliverables and Grading

When submitting your project, please include the following:

- All of the files containing the code for all parts of the assignment.
- 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.
- The test files or input that you use to convince yourself (and others) that your programs actually work.
- Output from your tests.
- A document called `README.txt` 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. Only plaintext write-ups are accepted.

Please compress all the files together as a single .zip archive for submission. As with all projects, please **only standard zip files** for compression; **.rar, .7z, and other custom file formats will not be accepted**.

The project programming is only a portion of the project. Students should use the following checklist in turning in their projects to avoid forgetting any deliverables:

1. Sign up for a project partner or have one assigned (URL: `https://cerebro.cs.wpi.edu/cs3013/request_teammate.php`),
2. Submit the project code and documentation via InstructAssist (URL: `https://cerebro.cs.wpi.edu/cs3013/files.php`),
3. Complete your Partner Evaluation (URL: `https://cerebro.cs.wpi.edu/cs3013/evals.php`), and
4. Schedule your Project Demonstration (URL: `https://cerebro.cs.wpi.edu/cs3013/demos.php`), which may be posted slightly after the submission deadline.

A grading rubric has been provided at the end of this specification to give you a guide for how the project will be graded. No points can be earned for a task that has a prerequisite unless that prerequisite is working well enough to support the dependent task. Students will receive a scanned markup of this rubric as part of their project grading feedback.

Groups **must** schedule an appointment to demonstrate their project to the teaching assistants. Groups that fail to demonstrate their project will not receive credit for the project. If a group member fails to attend his or her scheduled demonstration time slot, he or she will receive a 10 point reduction on his or her project grade.

During the demonstrations, the TAs will be evaluating the contributions of group members. We will use this evaluation, along with partner evaluations, to determine contributions. If contributions are not equal, under-contributing students may be penalized.

Project 1 – Linux Shell – Grading Sheet/Rubric

Evaluation?

Grader: _____ Student Name: _____ _____

Date/Time: _____ Student Name: _____ _____

Team ID: _____ Student Name: _____ _____

Late?: _____

Checkpoint?: _____ Project Score: _____

Earned	Weight	Task ID	Description
_____	5%	1	Phase 1 – Correct execution on student test cases.
_____	10%	2	Phase 1 – Correct execution on grader test cases. Prerequisite: Task 1.
_____	5%	3	Phase 2 – Correct execution on student test cases. Prerequisite: Task 2.
_____	10%	4	Phase 2 – Suitable student test cases. Prerequisite: Task 3.
_____	20%	5	Phase 2 – Correct execution on grader's test cases. Prerequisite: Task 3.
_____	15%	6	Phase 3 – Supports background commands and jobs. Prerequisite: Task 3.
_____	10%	7	Phase 3 – Handles out-of-order commands. Prerequisite: Task 6.
_____	5%	8	Phase 3 – Comprehensive student test cases for out-of-order execution. Prerequisite: Task 7.
_____	20%	9	Phase 3 – Correct execution on student and grader tests. Prerequisite: Task 8.

Grader Notes: