COMP3230A Principles of Operating Systems

Programming Assignment Two – A small Shell program

Total 13 points　　　　　　　　　Due date: October 21, 2016 5:00pm

(version: 1.0)

Objectives

1. An assessment task related to ILO4 [Practicability] – "demonstrate knowledge in applying system software and tools available in modern operating system for software development".
2. A learning activity related to ILO 2a.
3. The goals of this programming assignment are:
   - to have hands-on practice in designing and developing a shell program, which involves the creation, management, and coordination of multiple processes;
   - to learn how to use various important Unix system functions
     - to perform process creation and program execution;
     - to support interaction between processes by using signals and pipes;
     - to get process's running statistics; and
     - to show the interrelationship of processes.

Task

Shell program or commonly known as command interpreter is a program that acts as the user interface to the Operating System and allows the system to understand your commands. For example, when you input the "**ls -la**" command, the shell executes the system utility program called **ls** for you. Shell program can be used interactively or in batch processing.

You are going to implement a C program that acts as an ***interactive shell program,*** named as ***myshell***. This program supports the following features (full details will be covered in the Specification section):

1. The program (myshell) accepts command from user and executes the corresponding program with the given argument list.
2. It is able to locate and execute any valid program (i.e. compiled programs) by giving an absolute path (starting with /) or a relative path (starting with ./) or by searching directories under the $PATH environment variable.
3. It has three built-in commands:
   - ***exit*** command that terminates the myshell program; once the program starts, it continuously accepts commands from user until receiving the ***exit*** command.
   - ***timeX*** command that prints out the process statistics of a terminated child process.
   - ***viewtree*** command that prints out the family tree of the myshell process.
4. It supports two operators: **&** (run as background job) and **|** (pipe).
5. The myshell process should not be terminated by Cltr-c (**SIGINT** signal).

The assignment consists of four stages:

- Lab 1 – Create the first version of the myshell program to (i) accept an input line (command and arguments) from user, (ii) create a child process to execute the command, (iii) wait for child process to terminate, and (iv) print the process statistics of the terminated child process before terminating the myshell program.

- Lab 2 - Modify the first version of the myshell program to allow it to (i) handle the SIGINT signal correctly, (ii) allow the creation of background processes, (iii) handle the SIGCHLD signal from a terminated background child process, (iv) continue to accept new command from user, and (v) terminate when the user enters the *exit* command.
- Lab 3 - Modify the second version of the myshell program to allow it to (i) accept two commands (with arguments) and a '|' sign between two commands in one input command line, (ii) join the output of the 1st command to the input of the 2nd command, (iii) wait for child processes to terminate, and (iv) accept the timeX built-in command for printing the process statistics of each terminated foreground child process.
- Part 4 - Complete the final myshell program with the remaining features.

In summary, you are going to develop the myshell program incrementally.

Specification

Behavior of the myshell program:

- Like traditional shell program, when the myshell process is ready to accept input, it displays a prompt and waits for input from the user:

  ```
  ## myshell $
  ```

  After accepting a line of input from user, it parses the input line to construct the command name(s) and associated argument list(s), and then creates child process(es) to execute the command(s). We can assume that the command line is upper-bounded by 1024 characters with 30 arguments at maximum (including the | and & signs).

  If the child process is running in the foreground, myshell should wait for the child process to terminate before display the prompt. If the child process is running in the background, myshell should continue to display the prompt and wait for more input from the user.

- The myshell process should be able to locate and execute any program that can be found by

  - the absolute path specified in the command line, e.g.

    ```
    ## myshell $  /home/tmchan/a.out
    ```

  - the relative path specified in the command line, e.g.

    ```
    ## myshell $  ./a.out
    ```

  - searching the directories listed in the environment variable $PATH, e.g.

    ```
    ## myshell $  gedit
    ```

    where *gedit* is in the directory /usr/bin, and $PATH=/bin:/usr/bin:/usr/local/bin

  Please refer to Lab 1 to learn how to locate and execute a valid command or program.

- If the program cannot be found or executed, myshell displays an error message:

  ```
  ## myshell $  simp
  myshell: 'simp': not such file or directory
  ## myshell $  /bin/simp
  myshell: '/bin/simp': not such file or directory
  ## myshell $  ./start.sh
  myshell: './start.sh': Permission denied
  ```

- **&** sign - if the last character on the input line is the **&** character, the target program will be executed in the background, rather than having the myshell process waits for the program to complete (i.e., in foreground execution). For example, when the user types:

  ```
  ## myshell $  emacs &                    (or emacs&)
  ## myshell $
  ```

  The prompt will be displayed immediately and myshell is ready to accept another input from the user while emacs is running in the background.

  We can assume that it is incorrect to include the & sign at other locations of the command line except at the rightmost end:

  ```
  ## myshell $ ps & ls
  myshell: '&' should not appear in the middle of the command line
  ```

- **|** sign - if the **|** (pipe) signs appear in between commands in each input line, myshell tries to create multiple child processes and "connect" the standard output of the command before **|** (pipe) and links it to the standard input of the command after **|** (pipe). That is, each command (except the first one) reads the previous command's output. We can assume that the user will not enter more than 4 pipes (i.e., 5 commands) in an input command line. For example, when the user types:

  ```
  ## myshell $  cat TP-2016.html | grep table | wc -l
  6
  ## myshell $
  ```

  Please refer to Lab 3 to learn how to use *pipe()* and *dup()* system functions to set up a pipe between two processes.

- The myshell process should support correct use of both | and & signs.

- built-in command: ***exit*** – If the user enters the ***exit*** command, myshell should release all its resources and then terminate, and the standard shell prompt reappears. For example, if the user types

  ```
  ## myshell $ exit
  myshell: Terminated
  posnet@c3234-Ubuntu-1404:~/Desktop $
  ```

  If the ***exit*** command has other arguments, myshell would not treat it as a valid request and would not terminate. In addition, if the ***exit*** command is not appeared as the first word in the command line, myshell would not treat it as the ***exit*** command.

  ```
  ## myshell $ not exit
  myshell: 'not': No such file or directory
  ## myshell $ exit now
  myshell: "exit" with other arguments!!!
  ## myshell $
  ```

- built-in command: ***timeX*** – the user uses the ***timeX*** command to find out the process statistics of all terminated child process(es) under that input command line. For example,

  ```
  ## myshell $ timeX ps f
    PID TTY      STAT   TIME COMMAND
   3227 pts/7    Ss+    0:00 bash
   2245 pts/0    Ss     0:00 bash
  ```

```
 4355 pts/0    S+     0:00  \_ ./myshell
 4356 pts/0    R+     0:00     \_ ps f

PID          CMD              RTIME          UTIME          STIME
4356         ps               0.01 s         0.00 s         0.00 s
## myshell $
```

If the user just entered the ***timeX*** command without another command, the myshell process should report the error to the user:

```
## myshell $ timeX
myshell: "timeX" cannot be a standalone command
## myshell $
```

In addition, ***timeX*** can only be used to report the process statistics of the foreground process(es), it does not apply to the background processes and is considered as an incorrect usage of the command:

```
## myshell $ timeX ls &
myshell: "timeX" cannot be run in background mode
## myshell $
```

All information about the statistics of a process can be found in the stat file under /proc/[process id] directory. Please refer to Lab 1 to learn how to retrieve those process's statistics from the proc filesystem.

- The myshell process is required to handle three signals: SIGINT, SIGCHLD and SIGUSR1. The corresponding signal handlers should be implemented.

  – **SIGINT** signal: The myshell process and its child processes are required to response to the SIGINT signal (generated by pressing Ctrl-c) as according to the following guideline:
  Foreground job should response to SIGINT signal if the signal is detected; whether the foreground job will terminate or not should depend on the defined behavior of the program in response to the SIGINT signal. Thus, some foreground job may terminate while some others may not.

```
## myshell $ ./forever
^CReceives SIGINT!! IGNORE IT :)
^CReceives SIGINT!! IGNORE IT :)
^CReceives SIGINT!! IGNORE IT :)

## myshell $ ./loopf 10
^C## myshell $
```

  The myshell process and any background processes should not be terminated by SIGINT. When the user presses Ctrl-c while the myshell process is waiting for input, myshell should react with a new prompt:

```
## myshell $ ^C
## myshell $ ^C
## myshell $ ^C
## myshell $
```

  – **SIGCHLD** signal: When the child process completed its execution, the system always sends the SIGCHLD signal to its parent process; the default action of the parent process is just to ignore the signal.

In our case, the myshell process reacts to the SIGCHLD signal as according to whether the terminated child processes are foreground or background processes. In the case of the termination of a foreground child process, myshell does not need the SIGCHLD signal to inform it as it keeps on waiting for the child process to terminate. In the case of the termination of a background child process, as this is an asynchronous event, myshell could only detect that by means of the SIGCHLD signal. Therefore, the myshell process needs a SIGCHLD handler to handle the termination of background processes.

Upon receiving the SIGCHLD signal from a terminated child process, myshell prints out a statement to indicate that it has detected the termination of a child process. For example:

```
## myshell $ ./loopf 3               ← This is a foreground process
Time is up: 3 seconds                ← These two lines are from the ./loopf program
Program terminated.
## myshell $ ./loopf 3 &             ← This is a background process
## myshell $
## myshell $
## myshell $ Time is up: 3           ← Output of the ./loopf program
seconds
Program terminated.
[32334] loopf Done                   ← This indicates a background process has
                                     terminated
```

– **SIGUSR1** signal: This signal is available to users to define their own activity or event. In our case, the myshell process uses this signal to inform its child process when to start executing the target command. With this control, the child process has to wait for the signal before executing the command, and this allows the myshell process to have more control.

Please refer to Lab 2 to learn how to install a signal handler to handle a specific signal.

- [Extra feature for the teamwork] built-in command: ***viewtree*** – this command displays all child processes of the myshell process as well as their descendants in the form of tree diagram. As the system keeps all processes information in the /proc filesystem, we can search all processes to identify their relationship and construct the family tree. For example:

```
## myshell $ ./nested 4 &            ← This call creates 4 levels of processes
## myshell $ ./loopf 30 &            ← This process loops for 30 seconds
## myshell $ ./loopf 20 &            ← This process loops for 20 seconds
## myshell $ viewtree
myshell - nested - nested - nested - nested - nested
        - loopf
        - loopf
## myshell $ ps f                    ← Another view to visualize the hierarchy
  PID TTY      STAT   TIME COMMAND
28794 pts/0    Ss     0:00 bash
  823 pts/0    S+     0:00  \_ ./myshell
  824 pts/0    S      0:00      \_ ./nested 4
  825 pts/0    S      0:00      |   \_ ./nested 4
  826 pts/0    S      0:00      |       \_ ./nested 4
  827 pts/0    S      0:00      |           \_ ./nested 4
  828 pts/0    Z      0:00      |               \_ [nested] <defunct>
  831 pts/0    R      0:21      \_ ./loopf 30
```

```
832 pts/0    R      0:13      \_ ./loopf 20
833 pts/0    R+     0:00       \_ ps f
```

Documentation

1. At the head of the submitted source code, state clearly the
   - Student name and No.:
   - Student name and No.:
   - Development platform:
   - Last modified date:
   - Compilation – describe how to compile your program
2. Inline comments (try to be detailed so that your code could be understood by others easily)

Computer platform to use

For this assignment, you are expected to develop and test your program under Ubuntu 14.04. Your programs must be written in C and successfully compiled with gcc.

Group or Individual Work

You can **form a team of two to work on the assignment** (but the team has to do an additional feature - *viewtree*)**, or if you prefer, you can work individually**. Please keep an eye on the announcement on the course's Moodle site for the registration procedure.

Grading Criteria

Documentation (1.5 points)	High Quality (1.0 point)	• Include necessary documentation to clearly indicate the logic of the program
	Standard Quality (0.5 points)	• Include required student's info at the beginning of the program • Include minimal inline comments
Correctness of the program (11.5 points for individual work) (13.5 points for teamwork)#	Process creation and execution – foreground (2 points)	• Should be able to print "## myshell $ " and accept user's input • Should be able to execute the input command using a child process • Can locate and execute a command with full path, or with relative path, or under the standard PATH • Can execute a command with any number of arguments • Can handle error situation correctly, e.g., incorrect filename, incorrect path, not a binary file, etc. • Should wait for foreground process to complete before accepting next request • Allow user execute multiple commands (with arguments) one at a time • Should handle all zombie processes

	Process creation and execution – background (2 points)	• Correct use of the & sign and display appropriate message when encountering error • Should be able to execute the input command using a child process in the background • Allow user enter the next request without waiting for the child process to terminate • Support creation of multiple background jobs running in parallel • Support creation of multiple background jobs running in parallel with a foreground job with the defined behavior
	Process creation and execution – use of ‘|’ (2 points)	• Correct use of | sign; can report improper use of | sign • Can execute two commands which are connected by a pipe • Can execute two commands with any number of arguments and are connected by a pipe • Can execute at most 5 commands with any number of arguments and are connected by a sequence of pipes • Support creation of multiple pipeline processes running in the background
	Use of signals (3 points)	SIGINT signal (1.2 points) • Correct behavior of the myshell process, its foreground child processes, and its background child processes in handling the SIGINT signal SIGCHLD signal (1.2 points) • Should handle the SIGCHLD signals from all terminated background child processes and be able to print out the “Done” statement for each background child process SIGUSR1 signal (0.6 points) • All child processes should wait for the SIGUSR1 signal before executing the target command.
	Built-in command: timeX (2 points)	• Correct use of the *timeX* command; can report improper usage • For each terminated foreground child process, the system prints out the process statistics of the process in the correct format
	Built-in command: exit (0.5 points)	• Correct use of the *exit* command; can report improper usage
	Built-in command: viewtree (2 points)	[Not for individual work] • Correct use of the *viewtree* command; can report improper usage • Can identify all the child processes and their descendants of the myshell process • Should show the processes interrelationship in a tree-like format

# - The maximum score of the teamwork is 15 points. We shall apply a scaling factor - $^{13}/_{15}$ to get the final score for each member of the team.

Plagiarism

Plagiarism is a very serious offense. Students should understand what constitutes plagiarism, the consequences of committing an offense of plagiarism, and how to avoid it. **Please note that we may request you to explain to us how your program is functioning as well as we may also make use software tools to detect software plagiarism.**