COMP3230 Principles of Operating Systems

Programming Assignment One

Due date: Oct. 18, 2022 at 23:59

Total 12+4 points

(version: 1.0)

Programming Exercise – Implement a simple Shell program

Objectives

1. An assessment task related to ILO4 [Practicability] – "demonstrate knowledge in applying system software and tools available in a 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; and
        - to get the process's running statistics.

Task

Shell program or commonly known as a 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 ***3230shell***. This program supports the following features (full details will be covered in the Specification section):

1. The program (3230shell) accepts a command from the user and executes the corresponding program with the given argument list.
2. It can 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 two built-in commands:
    - ***exit*** command that terminates the 3230shell program; once the program starts, it continuously accepts commands from the user until receiving the ***exit*** command.
    - ***timeX*** command that prints out the process statistics of terminated child process(es).
4. It supports two operators: **&** ([Bonus] background job) and **|** (pipe).
5. The 3230shell process could not be terminated by Cltr-c (or **SIGINT** signal).

We suggest you divide the implementation of the assignment into four stages:

- Stage 1 – Create the first version of the 3230shell program, which (i) accepts an input line (contains a command and its arguments) from the user, (ii) creates a child process to execute the command, (iii) waits for the child process to terminate, (iv) if the process is terminated by a signal, print the information, and (v) prints the running statistics of the terminated child process if the user includes the built-in *timeX* command. After the child process is terminated, the shell program prints the shell prompt and waits for the next command from the user.

- Stage 2 - Modify the previous version of the 3230shell program to allow it to (i) handle the SIGINT signal correctly, (ii) use the SIGUSR1 signal to control the child process, and (iii) terminate the shell program when the user enters the *exit* command.
- Stage 3 - Modify the previous version of the 3230shell program to allow it to (i) accept two commands (that each command may have its 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 running statistics of **each** terminated foreground child process.
- Stage 4 - Complete the final 3230shell program with the remaining features, i.e. accepts at most 5 commands with their arguments separate by the '|' sign.
  And the bonus features: (i) create background processes, (ii) handle the SIGCHLD signal correctly, (iii) detect the termination of the background processes, and (iv) allows the creation of the sequence of piped commands to be executed as background processes and detects their termination.

In summary, you are going to develop the 3230shell program incrementally.

Specification

The behavior of the 3230shell program:

- Like a traditional shell program, when the 3230shell process is ready to accept input, it displays a prompt and waits for input from the user:

  ```
  $$ 3230shell ##
  ```

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

  If the child process is running in the foreground, 3230shell should wait for the child process to terminate before displaying the prompt.

- The 3230shell 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.

  ```
  $$ 3230shell ##   /home/tmchan/a.out
  ```

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

  ```
  $$ 3230shell ##   ./a.out
  ```

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

  ```
  $$ 3230shell ##   gcc
  ```

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

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

- If the program cannot be located or executed, 3230shell displays an error message:

  ```
  $$ 3230shell ##   simp
  ```

```
3230shell: 'simp': No such file or directory
$$ 3230shell ##   /bin/simp
3230shell: '/bin/simp': No such file or directory
$$ 3230shell ##   ./start.sh
3230shell: './start.sh': Permission denied
```

- **|** sign - if the **|** (pipe) signs appear in between commands in each input line, 3230shell tries to create multiple child processes and "connects" 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:

```
$$ 3230shell ##   cat TP-2022.html | grep table | wc -l
6
$$ 3230shell ##
```

We can assume that it is incorrect to have the | sign as the first or last character on the input line. Also, it is incorrect to allow two | signs without a command in between. For example,

```
$$ 3230shell ## cat PASS2.txt | | grep shell
3230shell: should not have two consecutive | without in-between command
$$ 3230shell ##
```

Please refer to Lab 2 to learn how to use *pipe()* system functions to set up a pipe between two processes.

- built-in command: ***exit*** – If the user enters the ***exit*** command, 3230shell should release all its resources and then terminate, and the standard shell prompt reappears. For example, if the user types

```
$$ 3230shell ## exit
3230shell: Terminated
root@ff860ab794d9:/homec3230# ps
  PID TTY      STAT   TIME COMMAND
    1 pts/0    Ss     0:00 bash
   79 pts/0    R+     0:00 ps
root@ff860ab794d9:/home/c3230#
```

If the ***exit*** command has other arguments, 3230shell would not treat it as a valid request and would not terminate. In addition, if the ***exit*** command does not appear as the first word in the command line, 3230shell would not treat it as the ***exit*** command.

```
$$ 3230shell ##  not exit
3230shell: 'not': No such file or directory
$$ 3230shell ##  exit now
3230shell: "exit" with other arguments!!!
$$ 3230shell ##
```

- 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,

```
$$ 3230shell ## timeX ps f
  PID TTY      STAT   TIME COMMAND
    1 pts/0    Ss     0:00 bash
```

```
   81 pts/0    S+     0:00 ./3230shell
   84 pts/0    R+     0:00  \_ ps f

(PID)84  (CMD)ps    (user)0.002 s  (sys)0.000 s
$$ 3230shell ##
$$ 3230shell ## timeX cat PASS2.txt | grep shell | wc -l
81
(PID)86  (CMD)cat    (user)0.001 s  (sys)0.000 s
(PID)87  (CMD)grep    (user)0.001 s  (sys)0.000 s
(PID)88  (CMD)wc    (user)0.001 s  (sys)0.000 s
$$ 3230shell ##
```

For each terminated child process, the shell displays the terminated process's ID, program name (without the path), user cpu time (in seconds) and system cpu time (in seconds). The output format is:
`(PID)127··(CMD)add····(user)17.056·s··(sys)0.000·s`
And '·' stands for a space character and the timings are formatted to 3 decimal places.

If the user just entered the ***timeX*** command without another command, the 3230shell process should report the error to the user:

```
$$ 3230shell ##  timeX
3230shell: "timeX" cannot be a standalone command
$$ 3230shell ##
```

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 [Bonus] and is considered an incorrect usage of the command:

```
$$ 3230shell ##  timeX ls &
3230shell: "timeX" cannot be run in background mode
$$ 3230shell ##
```

Please refer to Lab 1 to learn how to retrieve those processes' statistics from the wait4() system call.

- The 3230shell process is required to handle the SIGINT and SIGUSR1 signals. The corresponding signal handlers should be implemented.

  – **SIGINT** signal: The 3230shell process and its child processes are required to respond to the SIGINT signal (generated by pressing Ctrl-c or kill command) according to the following guideline:

    The foreground job should respond to the signal according to the predefined behavior of the program in response to the SIGINT signal. Thus, some foreground jobs may terminate while some others may not.

```
$$ 3230shell ##  ./forever
^CReceives SIGINT!! IGNORE IT :)
^CReceives SIGINT!! IGNORE IT :)
^CReceives SIGINT!! IGNORE IT :)

$$ 3230shell ##  ./loopf 10
^CInterrupt
^C$$ 3230shell ##
```

If a process is terminated by a signal, 3230shell should display the type of signal that caused the program to terminate, e.g., if detects SIGINT, prints "Interrupt"; if detects SIGKILL, prints "Killed". Please refer to Lab 2 to learn how to handle signals and display appropriate messages.

The 3230shell process should not be terminated by SIGINT. When the user presses Ctrl-c while the 3230shell process is waiting for input, 3230shell should react with a new prompt:

```
$$ 3230shell ##  ^C
$$ 3230shell ##  ^C
$$ 3230shell ##  ^C
$$ 3230shell ##
```

– **SIGUSR1** signal: This signal is available to users to define their own activity or event. In our case, child processes would not run the target command immediately after creation. We want them to wait for the signal (SIGUSR1) sent by the parent shell before executing the command. This mechanism guarantees all control structures used by the shell for managing processes have been updated before executing the child processes.

- [Bonus] **&** 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 3230shell process waits for the program to complete (i.e., in foreground execution). For example, when the user types:

```
$$ 3230shell ## vi &                          (or vi&)
$$ 3230shell ## ps f
  PID TTY      STAT   TIME COMMAND
    1 pts/0    Ss     0:00 bash
   11 pts/0    S+     0:00 ./3230shell
   14 pts/0    T      0:00  \_ vi
   15 pts/0    R+     0:00  \_ ps f
```

The prompt will be displayed immediately and 3230shell is ready to accept another input from the user while vi is running in the background, e.g., ps f.

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

```
$$ 3230shell ##  ps & ls
3230shell: '&' should not appear in the middle of the command line
```

- [Bonus] SIGCHILD 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 3230shell process reacts to the SIGCHLD signal according to whether the terminated child processes are foreground or background processes. In the case of the termination of a foreground child process(es), 3230shell should wait for the child process(es) to terminate. In the case of the termination of a background child process, as this is an asynchronous event, 3230shell could detect that through the SIGCHLD signal. Therefore, the 3230shell process needs a SIGCHLD handler to handle the termination of background processes.

Upon receiving the SIGCHLD signal from a terminated background child process, 3230shell should handle and release the zombie process. 3230shell should also output a statement to the terminal to indicate the termination of a background child process. However, the statement will not be displayed immediately; it will be displayed whenever the shell prompt reappears. We follow a way similar to how the traditional Linux bash shell works.

For example:

```
$$ 3230shell ## ./loopf 3     ← This is a foreground process
Time is up: 3 seconds         ← These two lines are from the ./loopf program
Program terminated.
$$ 3230shell ## ./loopf 8 &   ← This is a background process
$$ 3230shell ##
$$ 3230shell ## timeX ./add   ← Running a foreground process
Time is up: 8 seconds         ← Output of the ./loopf program
Program terminated.

(PID)102  (CMD)add    (user)22.699 s  (sys)0.000 s    ← Timing info of the foreground process
[101] loopf Done              ← This indicates a background process has terminated
$$ 3230shell ##
```

Second example – the zombie processes should be removed as soon as the process has terminated.

```
$$ 3230shell ## ./add &       ← We start 4 background processes
$$ 3230shell ## ./add &
$$ 3230shell ## ./add &
$$ 3230shell ## ./add &
$$ 3230shell ## ps f           ← Running a foreground ps process to view the status
  PID TTY      STAT   TIME COMMAND
   88 pts/1    Ss+    0:00 /bin/sh
    1 pts/0    Ss     0:00 bash
   81 pts/0    S+     0:00 ./3230shell
  104 pts/0    R      0:11  \_ ./add
  105 pts/0    R      0:08  \_ ./add
  106 pts/0    R      0:05  \_ ./add
  107 pts/0    R      0:01  \_ ./add
  108 pts/0    R+     0:00  \_ ps f
$$ 3230shell ## ps f           ← After 30 seconds, run ps f again
  PID TTY      STAT   TIME COMMAND
   88 pts/1    Ss+    0:00 /bin/sh
    1 pts/0    Ss     0:00 bash
   81 pts/0    S+     0:00 ./3230shell
  109 pts/0    R+     0:00  \_ ps f
[104] add Done                 ← Now displays the messages before the next prompt
[105] add Done
[106] add Done
[107] add Done
$$ 3230shell ##
```

Third example – the background process is terminated because of a signal.

```
$$ 3230shell ## ./forever &    ← We run a background process
$$ 3230shell ##
$$ 3230shell ## ps f           ← Show current status
  PID TTY      STAT   TIME COMMAND
   88 pts/1    Ss+    0:00 /bin/sh
    1 pts/0    Ss     0:00 bash
   81 pts/0    S+     0:00 ./3230shell
  125 pts/0    R      0:04  \_ ./forever
  126 pts/0    R+     0:00  \_ ps f
$$ 3230shell ##                ← The background process is still running
$$ 3230shell ##
$$ 3230shell ##    ← Via another terminal, send SIGTERM to proc 125; then press enter at prompt ##
[125] forever Terminated        ← Now displays the messages before the next prompt
$$ 3230shell ##
```

Documentation

1. At the head of the submitted source code, state clearly the
   - Student name and No.:
   - Development platform:
   - Remark – describe how much you have completed; whether you have implemented the bonus part.
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 programs on the workbench2 Linux platform. Your programs must be written in C and successfully compiled with gcc. It would be nice if you develop and test your program on your own machine (WSL2 or Ubuntu docker image) or academy servers. After fully tested locally, upload the program to the workbench2 for the final tests.

Submission

Submit your program to the Programming # One submission page at the course's moodle website. Name your program with this format: 3230shell_StudentNumber.c (replace StudentNumber with your HKU student number). As the Moodle site may not accept source code submission, please compress your program to the zip or tgz format before uploading.

Grading Criteria

1. Your submission will be primarily tested under workbench2. Make sure that your program can be compiled *without any errors*. Otherwise, we have no way to test your submission and you will get a zero mark.
2. As the tutor will check your source code, please write your program with good readability (i.e., with good code convention and sufficient comments) so that you will not lose marks due to possible confusion.

Documentation (1 point)	• Include necessary comments to clearly indicate the logic of the program • Include required student's info at the beginning of the program	
Correctness of the program (11 points)	Process creation and execution – foreground (3 points)	• Should be able to print "$$ 3230shell ## " and accept user's input • Should be able to execute the input command using a child process • Can locate and execute a command with a full path, with a relative path, or under the standard PATH • Can execute a command with any number of arguments • Can handle error situations 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 to execute multiple commands (with arguments) one after the other • Should remove all zombie processes

	Process creation and execution – use of '|' (3 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
	Use of signals (2.5 points)	SIGINT signal (1.5 points) • Correct behavior of the 3230shell process and the foreground child processes in handling the SIGINT signal SIGUSR1 signal (1 point) • All child processes should wait for the SIGUSR1 signal before executing the target commands.
	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
Bonus part (4 points)	Process creation and execution – background (2 points)	• Correct use of the & sign and display appropriate messages when encountering errors • Should be able to execute the input command using a child process in the background • Allow user to enter the next request without waiting for the background process to terminate • Support creation of multiple background jobs running in parallel with a foreground job with the defined behavior • Support creation of multiple pipeline processes running in the background
	SIGCHLD signal (2 points)	• Should handle the SIGCHLD signals from all terminated child processes and be able to print out the appropriate statement for each background child process upon termination.

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 of software tools to detect software plagiarism.**