CS236 Spring 2025

Lab Quiz1

4 questions, 20 marks

Instructions

Before you begin, please check that you can see the following files on the Desktop:

- This question paper `labquiz1.pdf`, with 4 questions
- A tarball `labquiz1_code.tgz`

Untar the code file as shown below:
`tar -zxvf labquiz1_code.tgz`

You will now find a folder `labquiz1_code` in the same directory, which has the following template code:

- The shell template code to use in Q2 and Q3: `my-shell-template.c`
- Template code for Q4: `bitstring-send.c` and `bitstring-send-2.c`

Now, create your submission folder titled `submission_rollnumber` in the Desktop directory, where you must replace the string `rollnumber` above with your own roll number (e.g., your directory name should look like `submission_12345678`). Write your solutions in this folder. When you finish the quiz, ensure that the submission folder has *only* the following files for evaluation:

- `q1.txt` for Q1
- `my-shell-q2.c` for Q2
- `my-shell-q3.c` for Q3
- `bitstring-send.c` and `bitstring-send-2.c` for Q4

Submission Instructions

1. Once you are ready to submit, please run the command `check` from your terminal. This command will create a tarball of your submission folder. If the folder is not in the proper place or is empty, this script will throw an error. In that case, please fix the errors and try again.

2. Next, call one of the TAs and ask them to run the `submit` command from your terminal. The TA will enter the password to upload your submission tarball to our remote submission server. Please note that we will only be grading the files that are submitted in this manner. So please ensure that you submit the correct files.

3. **IMPORTANT NOTE: Please ensure that you are submitting the correct files with the correct filenames. Test your code properly before submission. Strictly NO code changes will be allowed after the exam.**

Questions

1. **[5 marks]** In each of the following sub-parts, write the terminal command to print the process-related information asked in the question. You can use `ps`, `top`, `pstree` or any such commands in the answer. You are encouraged to use man pages to explore the correct options to give to the commands. Note that the command in your answer should be a single line, which gives the desired output when copy-pasted into a bash terminal. Write your answers in the file `q1.txt`.

   Hint: press 'h' when inside a man page to get useful keyboard shortcuts to help you effectively search within a man page (particularly the searching section).

   To give you a better idea of how your answers should be written, here are a few sample questions and answers.

   Sample question 1: Write the command to list every process on the system using BSD syntax

   Expected answer: `ps ax`

   Sample question 2: Write the command to list the pid, ppid of process with pid 123

   Expected answer: `ps p 123 -o pid,ppid`

   Now, write the commands to get the following information in a Linux system.

   (a) Write the command to list the pid, state, rss occupied by all the process running on the system.

   (b) Explore the man page of the pstree command using `man pstree`. Find out the flag to use ascii characters to draw the tree. Now, write the command to print process tree of a process with pid 123 using ascii characters.

   (c) Write the command to list every process on the system using standard (ps) syntax.

   (d) Write the command to print a process tree for all processes using ps.

   (e) List the pid, command, time of all the processes that are not init (i.e., command name should not be init).

2. **[6 marks]** Write a program `my-shell-q2.c` that implements a simple bash-like shell. Use the `my-shell-template.c` given to you as your starting point. Your shell must support the following features.

   (a) Your shell must execute any simple Linux command given as input, e.g., `ls, cat, echo` and `sleep`, much like what you did in the part A of the "Shell" lab.

   You may assume that the input command has no more than 1024 characters, no more than 64 tokens, and each token is no longer than 64 characters. You may assume that the commands are simple Linux commands with no pipes or redirections.

   (b) Your shell must support the cd command using the `chdir` system call.

   (c) Your shell must show current working directory (cwd) in the command prompt, e.g., if the current directory of the shell is `/home/labuser`, then the command prompt should be `/home/labuser $`

   You may assume that the current working directory is no more than 256 characters in size. You can use the function `getcwd(char *buf, size_t size)`, where cwd will be stored and returned in the `buf` array. Read `man getcwd` for more information.

   (d) The child process spawned to run a command must return a suitable exit code in case of errors. Specifically, for any incorrect command given where `exec` fails, the child should exit with exit code 1. Modify your shell such that it prints the exit status of child process after the command completes, e.g., if the child exited with an exit code 1, the shell should print `EXITSTATUS: 1`.

   Note that if you use `wait(&ws)` to reap a child, where `ws` is an integer, then `WEXITSTATUS(ws)` gives the exit code of the child process. Read `man wait` for more information.

   (e) Your shell must not execute any command for more 10 seconds. For example, if you run `sleep 100`, then after 10 seconds of execution, the shell has to print `KILLED CHILD AFTER 10 SECONDS`, then print the exit code, and go back to the command prompt.

   You can use the function `alarm(unsigned sec)` to send the `SIGALRM` signal to the calling process after `sec` number of seconds. Read `man alarm` and `man signal` for more information.

   (f) An empty command (hitting enter key) should simply cause the shell to display a prompt again without any error messages. For all commands, you must take care to terminate and carefully reap any child process the shell may have spawned. You must run `ps` in a separate terminal to check that child processes are being reaped correctly, and no zombie is being left behind.

A sample execution is shown below

```
/home/labuser/Desktop/q2 $ pwd
/home/labuser/Desktop/q2
EXITSTATUS: 0
/home/labuser/Desktop/q2 $ ls
a.out                   my-shell-q2.c           my-shell-template.c
EXITSTATUS: 0
/home/labuser/Desktop/q2 $ var
EXITSTATUS: 1
/home/labuser/Desktop/q2 $ cd ..
/home/labuser/Desktop $ sleep 5
EXITSTATUS: 0
/home/labuser/Desktop $ sleep 100
KILLED CHILD AFTER 10 SECONDS
EXITSTATUS: 0
/home/labuser/Desktop $
```

(last prompt should appear 10 seconds after running sleep 100)

3. **[4 marks]** In this question, you will write the program `my-shell-q3.c`, to add additional functionality for sequential foreground execution to your simple shell. Start with a simple shell program that supports the execution of basic commands, and add support to execute a sequence of commands, one after the other, in the foreground. The shell should execute each command in the sequence, reap the child process, and then proceed to the next command in the sequence. You do not need to add support for "cd" or any other shell feature that is not specified in this question. You can assume that all commands in the sequence are simple Linux commands.

   You must now support two new command delimiters which will be used to separate the commands in the sequence - `##` and `;;`, each with slightly different semantics as explained below.

   (a) If the sequential command is given as

   $$\texttt{cmd}_1 \texttt{ \#\# cmd}_2 \texttt{ \#\# cmd}_3 \texttt{ ...}$$

   then $\texttt{cmd}_i$ must execute only if $\texttt{cmd}_{i-1}$ completed without being interrupted by a Ctrl+C signal. That is, if this sequence of commands is interrupted by Ctrl+C, then the remaining sequence should abort and your shell should go back to the command prompt.

   (b) If the sequential command is given as

   $$\texttt{cmd}_1 \texttt{ ;; cmd}_2 \texttt{ ;; cmd}_3 \texttt{ ...}$$

   then $\texttt{cmd}_i$ is always executed, even if the previous command was interrupted by a signal. That is, if this sequence of commands is interrupted by Ctrl+C, then the shell must abort the interrupted command, but proceed to the next command in the sequence.

   (c) If both the delimiters are present in the sequential command, then `##` has higher precedence compared to `;;`. For example, if the sequential command is

   $$\texttt{cmd}_1 \texttt{ ;; cmd}_2 \texttt{ \#\# cmd}_3 \texttt{ \#\# cmd}_4 \texttt{ ;; cmd}_5 \texttt{ ;; cmd}_6 \texttt{ \#\# cmd}_7$$

   Then, $\texttt{cmd}_1$ must be executed first followed by $\texttt{cmd}_2$ `##` $\texttt{cmd}_3$ `##` $\texttt{cmd}_4$ followed by $\texttt{cmd}_5$ followed by $\texttt{cmd}_6$ `##` $\texttt{cmd}_7$. In the above example, if we give SIGINT signal when $\texttt{cmd}_2$ is running, then $\texttt{cmd}_5$ should run next. If we give SIGINT signal when $\texttt{cmd}_4$ is running, then $\texttt{cmd}_5$ should run next. If we do not give any signal, then $\texttt{cmd}_1$ should be executed followed by $\texttt{cmd}_2$, followed by $\texttt{cmd}_3$ and so on.

   Note that in case of invalid commands, the shell should execute the next command irrespective of the delimiter in between.

   A sample execution of the program is shown below:

```
$ pwd
/home/labuser/Desktop/q2
$ pwd ;; pwd ;; pwd
/home/labuser/Desktop/q2
/home/labuser/Desktop/q2
/home/labuser/Desktop/q2
$ pwd ## pwd ## pwd
/home/labuser/Desktop/q2
/home/labuser/Desktop/q2
/home/labuser/Desktop/q2
$ sleep 50 ;; sleep 6 ## sleep 7 ;; sleep 8
```

In the last command, we press Ctrl+C, first during the execution of `sleep 50`, so the shell moves to the next command `sleep 6`. We again press Ctrl+C during the execution of `sleep 6`, the shell moves directly to `sleep 8`.

4. In this question, we will use signals to communicate a string of 0s and 1s between a parent and child process. You are given template code where a parent process forks a child, and reads a bit string of 0s and 1s of length 8. You must now add code to send this string to the child process via signals, and the child must print out the string it has received. You can use any two different signals to denote 0 and 1. The parent must use the `kill` system call to send signals to the child, and when the child receives and handles the signals, it can infer the 0 or 1 being sent by the parent. The child must terminate after receiving the string and must be reaped by the parent, after which the parent terminates too. We have also provided a synchronization mechanism to ensure that the parent process only starts sending the bit after the child process is ready to receive it.

   We will implement this communication in two different ways in the two parts below.

   (a) **[3 marks]** Make changes to the template code `bitstring-send.c` to send signals in the parent and receive them suitably in the child, so that the bit string is communicated correctly. You can use any signals of your choice to send 0 and 1. You may want to use user-defined signals to avoid terminating the child accidentally. You can find out more by typing `man 7 signal` in the terminal.

   Note that we have given a mechanism where the parent sleeps between the transmissions (via `sleep 1`) to ensure the child receives all of the signals slowly, one after the other, without the signals getting jumbled up. This sleep is essential for correct communication in this part.

   (b) **[2 marks]** If you have completed the previous part, you may have noticed that there was a high delay in sending the string, due to the need to sleep between signals. To fix this problem, we will now add a way for the child to inform the parent that it has received the previous bit, so that the parent can proceed to the next bit faster. You will write this faster program by adding code to the template `bitstring-send-2.c`, which is similar to the previous template, but without the waiting and sleeping in the parent.

   You can solve this problem by sending a signal back from the child to the parent when the child receives the bit. We must also make appropriate modifications in the parent process to ensure that the parent does not send the next bit until the child has received the previous one. Hint: Look at the synchronization mechanism used in this code at the start of the communication, could we perhaps do something similar?

   A sample execution is shown below. Your output will look similar in both parts, except that the program executes much faster in the second part.

```
$ ./a.out
Please input a 8-bit bitstring: 10111100
[Parent] Input bitstring is  10111100
[Child] Received bitstring is 10111100
```