Writing an Incomplete Shell

2015-16, CSCI 3150 - Programming Assignment 1

Specification version 1.1, updated on 2015 October 1

Abstract

The purpose of this assignment is to introduce students to the existing useful system calls in the Linux environment and the control of the creation as well as the termination of the processes.

Contents

1 Introduction

A significant portion of the interactive actions of Unix/Linux system is performed through an user-level command interpreter known as the **shell**, e.g., *bash* and *tcsh*. The shell implements many important aspects of the Linux operating system in terms of your daily use, including *process management* and *I/O redirection*.

In this assignment, you are required to implement a primitive version of the shell which makes heavy uses of the system calls provided by Linux.

What are the things that you are supposed to learn from this assignment?

- Hard skills:
  - Understand the internal workings of a shell;
  - Write a medium-sized program of hundreds locs;
  - Design and implement a structured program (and hope that you still remember what a structured program is);
  - Write and compile a program with more than one source file (if you want to);
  - Read man pages and dig out every little detail before you use a system/function call;
  - Using useful, existing function/system calls;
- Soft skills:
  - Manage your time under a *tight schedule*;
  - Cooperate with your partner *in harmony*;
  - Teach your partner with *patience*;
  - Learn from your partner *with an open mind*;

2 Program Flow

The shell program that you are going to implement is a simpler version than the ones that you use in Unix or Linux system. For simplicity, we name the shell that you are going to implement as the *OS shell*. Figure 1 shows the execution flow of the OS shell.

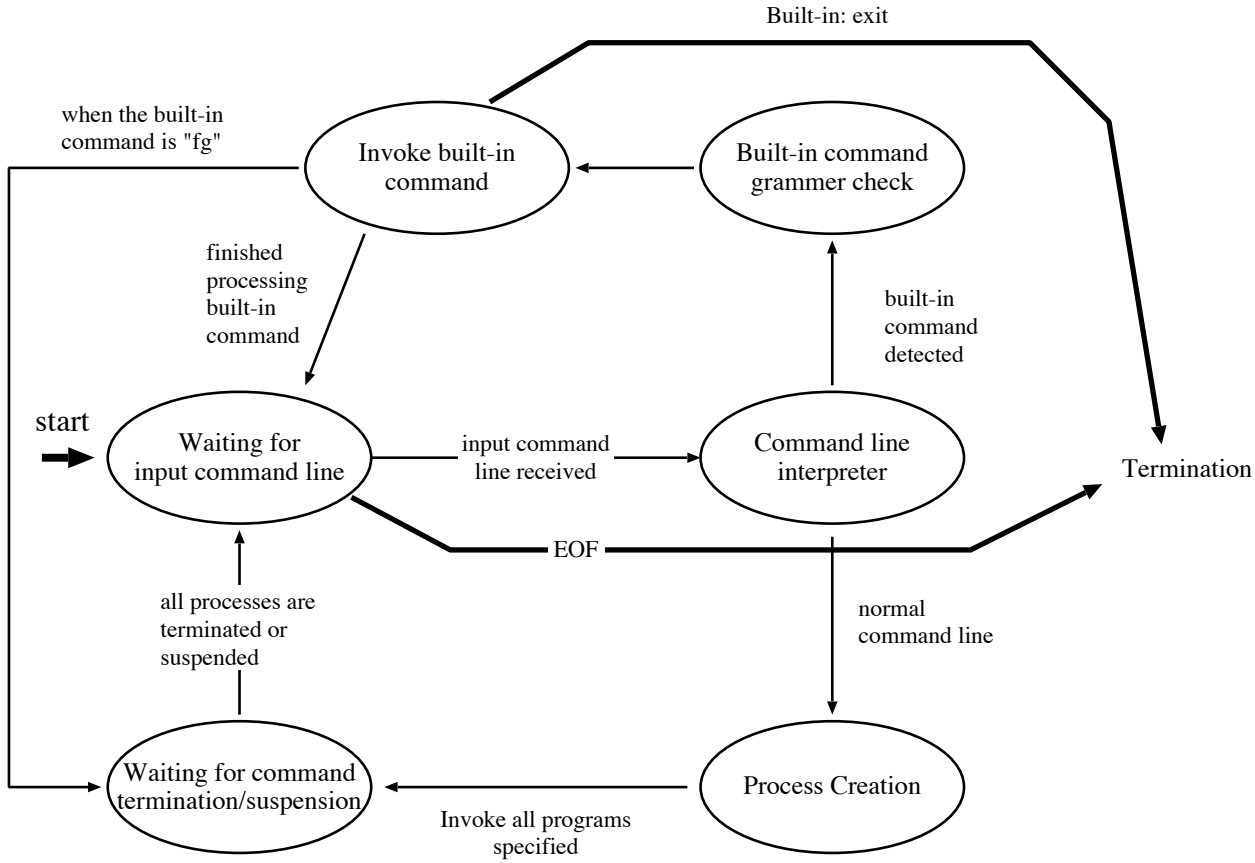

Figure 1: The shell's execution flow.

1. When the OS shell first starts, the program should be in the state "**Waiting for input command line**". The OS shell should show the following:

```
[3150 shell:/home/tywong]$ _
```

   We usually call it **the prompt**. Note that "`/home/tywong`" is an example directory at which the shell is invoked.

2. When the user types in a command followed by a **carriage return** (or the "enter" key), e.g.,

```
[3150 shell:/home/tywong]$ /bin/ls | /bin/less
```

then, the received input will be processed by the "**Command line interpreter**" of the OS shell, and we called the input, "`/bin/ls | /bin/less`", the **input command line**. In this assignment, we assume that an input command line always **agrees with the pre-defined syntax**, which will be defined in Section 2.2.

3. Although the syntax of the input command line is correct, the **semantic** of the input command line may be wrong. E.g., the following input command line:

```
[3150 shell:/home/tywong]$ /bin/sl | /bin/mole
```

contains invalid commands "`/bin/sl`" and "`/bin/mole`", but is correct in terms of its syntax. Therefore, the OS shell should report if the commands can be found.

2.1 Input command line

The **input command line** is a character string. The OS shell should read in the input command line from the **standard input stream** (`stdin` in C) of the OS shell. To ease your implementation, there are assumptions imposed on the input command line:

Assumptions

1. An input command line has a maximum length of 255 characters, not including the trailing newline character.
2. An input command line ends with a carriage return character '$\backslash n$'.
3. There would be no leading or trailing space characters in the input command line.
4. A **token** is a series of characters without any space characters. Each **token** is separated by at least one space character.

2.2 Command line syntax

Assumption: the syntax of the input command line is always correct.

The command line interpreter checks whether the input command line matches a pre-defined language or not.

- If yes, the OS shell checks and invokes the program(s) specified in the input command line.
- Else, the OS shell waits for the next input command line.

The language that the shell takes is similar to the *bash* shell and is specified as follows:

```
      [start]  :=  [empty string] or
               :=  [built-in command] [arg]* or
               :=  [command] [recursive] or
               :=  [command] [terminate]

  [recursive]  :=  '|' [command] [recursive] or
               :=  '|' [command] [terminate]

  [terminate]  :=  [empty string]
```

[command]	:=	[command name] [args]*
[command name]	:=	A string without any space, tab, > (ASCII 62), < (ASCII 60), | (ASCII 124), * (ASCII 42), ! (ASCII 33), ‘ (ASCII 96), ’ (ASCII 39), nor ” (ASCII 34) characters. Note that the command name is assumed to be different from a built-in command.
[arg]	:=	A string without any space, tab, > (ASCII 62), < (ASCII 60), | (ASCII 124), ! (ASCII 33), ‘ (ASCII 96), ’ (ASCII 39), nor ” (ASCII 34) characters.

Some examples of valid command line inputs are as follows.

- blank line
- Simple command: "`cat`", "`cat *`", and "`cat dog`"
- Pipeline: "`cat | cat`", and "`cat | cat | cat`"

Although you are not required to detect whether the input command line is correct or not, it is always good to know some examples of invalid command line inputs:

- "`cat | cat | `"
- "`>_<`" (using invalid characters.)

> **Important**: the OS shell is not required to ensure that a command works perfectly. It only needs to check the input command line against the grammar only.

If an input command line agrees with the above language, the OS shell will enter the "**Process creation**" state.

Useful Functions for this stage.
`fgets()`, `strtok()`

2.3 Process creation

The OS shell should **create the processes** according to the names specified in the input command line.

2.3.1 Locating the programs

There are two kinds of program name:

- If a program's name begins with the character '`/`', then that the program name is a **path name**, and is an absolute path.
- Else, that the program name is a **file name**.
- Names starting with "`./`" and "`../`" are considered as path names because they are specifying the program should be found in "`./`" (current directory) and "`../`" (parent directory), respectively.

If a program name is specified by a **path name**, e.g., `/bin/ls` or `./a.out`, then the creation of the process is straight-forward since you are given the location of the program already. Else, if a command name is specified by a **file name**, then the OS shell has to search for the program from the following locations in the following order:

Required path-searching order.
`/bin` → `/usr/bin` → `.` (current directory)

Remember that the **first matched entry** stops the search.

Example.
There exists two files: "`/bin/ls`" and "`./ls`". Then, the command "`ls`" will cause the OS shell to invoke the program "`/bin/ls`".

If the specified program cannot be found in the above three locations, the OS shell should report "`[command name]: command not found`".

Example.

`[3150 shell:/home/tywong]$ /bin/sl`
`/bin/sl: command not found`
`[3150 shell:/home/tywong]$ _`

Note that:

- The `execve` system call family is able to report whether you have failed to invoked a command or not by looking at its **return value** and **error number**, `errno`.

  In this assignment, you only need to report the cases that a program cannot be found (with both path name and file name). If the `errno` specifies other errors, then the shell should report:

  ```
  [command name]:  unknown error
  ```

- **Hint**: You are not required to perform an actual search for the command's location. You can totally depend on the `execve()` call family.

- Searching for commands outside the above three paths is prohibited. You will risk a **mark deduction** if your OS shell can invoke programs that are outside the above three paths.

Useful functions for this stage.
`fork()` system call, `execve` system call family, `setenv()`

2.3.2 Wildcard expansion

The wildcard in our shell is denoted by the character '`*`' only. In this assignment:

- A wildcard character will only appear in an argument to a program.
- In a command line, there can be more than one argument carrying a wildcard.
- An argument can contain more than one wildcard character.
- A wildcard can appear in the following ways:
  - "`/bin/ls *.txt`": in the beginning of a token;
  - "`/bin/ls hell*txt`": in the middle of a token;
  - "`/bin/ls hello*`": in the end of a token.

The meaning of wildcard expansion is that you are replacing the wildcard expression by looking up the files that matches that wildcard expression.

Example. Say there are four files in the directory: "a.txt", "b.txt", "c.txt", and "abc.mp3"
`ls *` ⇒ `ls a.txt abc.mp3 b.txt c.txt`
`ls *.txt` ⇒ `ls a.txt b.txt c.txt`
`ls a*` ⇒ `ls a.txt abc.mp3`
`ls b* a*` ⇒ `ls b.txt a.txt abc.mp3`
`ls *.iso` ⇒ `ls *.iso`

The expansion of wildcard expressions is actually **a task of the shell**. Therefore, in your shell implementation, when you find a token with wildcard characters, you have to:

1. Treat the token as one wildcard expression.
2. Then, expand that expression into a list of tokens, and this list of tokens should be sorted lexicographically.
3. When you cannot expand a particular wildcard expression, just **keep the expression unexpanded** (as shown in the last example).

Note importantly that you do not need to implement the expansion by yourself. Please use the library call "`glob()`". Our tutor will cover this in the tutorials.

2.3.3 I/O redirection & pipeline

You are only required to implement one **I/O redirection feature** in this assignment: the pipe. We usually name a command line with pipes a pipeline and the following is an example input format of a pipeline.

Pipe
`[command 1] | [command 2]`

A pipe connects the standard output stream of "`[command 1]`" to the standard input stream of "`[command 2]`". In other words, the standard input stream of "`[command 2]`" is replaced by the standard output stream of "`[command 1]`".

Pipe, example

```
[3150 shell:/home/tywong]$ cat output.txt
first hello world
second hello world
[3150 shell:/home/tywong]$ cat output.txt | head -1
first hello world
[3150 shell:/home/tywong]$ _
```

where "`head -[n]`" works similar to `cat`, but it prints only the first n lines from its standard input stream to its standard output stream, unlike `cat` which prints all data.

Useful system calls for pipeline

`pipe(), dup2(), open(), close(), exit().`

Please note the following important points about the pipeline:

- First thing first, every process within a pipeline must be executing concurrently.
- The number of pipes is not limited in a pipeline.
- When one of the commands in the pipeline fails to execute, the shell should:

  1. report the corresponding error message(s), and then
  2. terminate that corresponding process only.

  In turns, the shell should let the remaining processes execute normally. As a consequence, the remaining processes should terminate gracefully if and only if the processes are connected through the pipes correctly.

2.4 Detecting process termination and suspension

Once the processes are launched, the shell has to wait until **all the processes have stopped executing**, either terminated or suspended. After the processes have stopped running, the OS shell should wait for the next input command line.

Useful system call for this stage.
`waitpid()` (`wait()` cannot help).

Note importantly that the OS shell should not leave any **zombies** in the system when the OS shell is ready to read a new input command line. Otherwise, **you will have 1% of the course mark deducted**.

2.5 Built-in shell commands

In a traditional shell, there are built-in shell commands. In this assignment, we implement **four** of them, and their requirements are listed as follows.

1. **cd** [**arg**]. This command is called "*change directory*" and it changes the current working directory of the shell. Some points to note:

   - The argument [**arg**] is the destination path to which the user wants to change. Note that [**arg**] can be either an absolute path or a relative path.
   - If the location specified by [**arg**] results in an error, the OS shell should report the error message: "`[arg]: cannot change directory`".
   - In addition, this built-in shell command takes only one argument. If the number of arguments is wrong, the OS shell should report the error message: "`cd: wrong number of arguments`".

Useful system calls. `chdir()`.

2. **exit**. This command terminates the shell. This command takes no arguments. If the number of arguments is wrong, the OS should report: "`exit: wrong number of arguments`".

If there are suspended jobs, the OS shell will not be terminated. Please note that this feature is required in Phase 2 of the assignment only.

"exit" example, with suspended jobs

```
[3150 shell:/home/tywong]$ exit - are you sure?
exit:  wrong number of arguments
[3150 shell:/home/tywong]$ cat
^Z
[3150 shell:/home/tywong]$ exit
There is at least one suspended job
[3150 shell:/home/tywong]$ _
```

3. **fg [job number]**. This command is called "*foreground*", and it wakes a suspended job, specified by the job number - **[job number]**. We will talk about this built-in command in detail in Section 2.7 on Page 15.

   Note that, this built-in shell command takes only one argument. If the number of arguments is wrong, the OS shell should report the error message: "`fg:  wrong number of arguments`".

4. **jobs**. This command prints a list of suspended jobs to the standard output stream of the OS shell.

"jobs" example

```
[3150 shell:/home/tywong]$ jobs
[1] cat
[2] cat | cat
[3] top
[3150 shell:/home/tywong]$ _
```

The jobs are sorted by the time that they are created for the first time. The smallest job number is always one, and the OS shell should not skip any numbers. We will talk about this built-in command in detail in Section 2.7.

Important
The priority of the built-in shell commands always comes first. In other words, suppose there exists a program "`/bin/cd`", then when the user types in "`cd`", only the built-in shell command **cd** will be invoked instead of `/bin/cd`.

2.6 Signal Handling

Though a shell plays a special role in Unix/Linux operating systems, as a matter of fact, it is just a normal user-space program. Therefore, the OS shell will handle every possible signal in its default manner. However, you will not expect that a shell will be stopped or terminated by signals such as **Ctrl + Z** and **Ctrl + C**. Therefore, you are required to change the signal handling routine of the OS shell. The signals that you should take care of are listed as follows:

Signal	**Signal Handling**
`SIGINT (Ctrl + C)`	Ignore the signal.
`SIGTERM` (default signal of command "kill")	Ignore the signal.
`SIGQUIT (Ctrl + \)`	Ignore the signal.
`SIGTSTP (Ctrl + Z)`	Ignore the signal.
`SIGSTOP`	Default signal handling routine.
`SIGKILL`	Default signal handling routine.

For signals that are not listed in the above table, the OS shell should adopt the default signal handling routines of the corresponding signals. Note importantly that only the OS shell changes the handling of the said signals. That means the child processes created by the OS shell should, however, adopt the default signal handling routines.

Signal handling example

`[3150 shell:/home/tywong]$ cat`
`^C`
`[3150 shell:/home/tywong]$ _`

In the above example, the signal `SIGINT` is generated by `Ctrl + C`, and that signal terminates the process "`cat`" only, but not the OS shell.

2.7 Incomplete job control

You are going to implement an subset of job control features, comparing to those of the shell programs you can find in Unix/Linux.

2.7.1 Definition of job and job control

What is a job? A job is created by a **valid input command line**. All the processes that are created by the OS shell, because of a valid input command line, are considered as one job. Job control means the way that we control all those processes together. In the OS shell, we implement the suspension and the continuation of the jobs.

2.7.2 Suspension handling

When a job is running, e.g., "`cat`", the user can suspend it through the signal "`SIGTSTP`", which can be generated by typing **Ctrl + Z**. In brief, there are three ways to suspend a process.

1. Type "`Ctrl + Z`" on the terminal in which the process is running;
2. Send the signal "`SIGTSTP`" by using */bin/kill*. E.g., `/bin/kill -TSTP [process name]`;
3. Use the system call `kill()`. Yet, you have to write a program to invoke the system call "`int kill(pid_t pid, int signal_number)`", with the signal `SIGTSTP`.

When a job is suspended, the OS shell should go back to the "**Waiting for input command line**" state (see Figure 1 on page 4). The following example illustrates this point.

Job suspension, example

```
[3150 shell:/home/tywong]$ cat | cat
^Z
[3150 shell:/home/tywong]$ _
```

2.7.3 List the suspended jobs

When jobs created by the OS shell become suspended, the OS shell should be able to list those jobs using the **built-in shell command "`jobs`"**.

List suspended job, example

```
[3150 shell:/home/tywong]$ jobs
No suspended jobs
[3150 shell:/home/tywong]$ cat | cat
^Z
[3150 shell:/home/tywong]$ jobs
[1] cat | cat
[3150 shell:/home/tywong]$ _
```

2.7.4 Resuming the suspended jobs

The OS shell should also provide another built-in shell command "`fg`" for the user to resume the suspended jobs.

Wake suspended job, example

```
[3150 shell:/home/tywong]$ jobs
[1] cat | cat
[3150 shell:/home/tywong]$ fg 1
Job wake up:  cat | cat
^C
[3150 shell:/home/tywong]$ jobs
No suspended jobs
[3150 shell:/home/tywong]$ _
```

Note that when the supplied job number does not exist, you should report the error message: "`fg: no such job`". Note also important that the OS shell is not required to handle the case that a suspended job is **killed** by other processes.

Useful resource
`signal()` system call (read the man-page using the command "`man 7 signal`" on Linux and "`man -s 3HEAD signal`" on Unix) and `waitpid()` system call.

2.7.5 Defects in the incomplete job control

A frustrating defect would appear when you are playing with signals in a *wrong* way.

- [**The wrong signal**]. A strange scenario happens when "`Ctrl + C`" is pressed while there are suspended jobs.
- [**The wrong thing(s) appeared**]. The suspended jobs are killed!

Defect example:

`[3150 shell:/home/tywong]$ cat`
`^Z`
`[3150 shell:/home/tywong]$ cat`
`^C`
`[3150 shell:/home/tywong]$ fg 1`
`Job wake up:  cat`
`[3150 shell:/home/tywong]$ _ //woooo...my poor cat...`

Hence, you do not have to worry about the above strange behavior because this is the defect of the design of this assignment. (Of course, writing more codes can solve the problem. But, the lecturer would like to stop torturing you.)

If you are questing for perfection, please look up the following keywords from Google and Wikipedia and have a perfect job control implementation:

- Process group, foreground process group, background process group.
- Library call: `tcsetpgrp()`, system call: `setpgid(), getpgid()`.

2.8 Extra requirements

The following requirements are restrictions over the executions of programs. The **rule of thumb** of the following restrictions is to ensure that your shell is implemented

- through your own command line interpreter, and
- by using the fork-execute-wait(pid) system call combination.

In other words, you should experience how to build a shell using system calls. Most importantly, the restrictions ensure that credits are only given to those who have spent efforts.

1. [**Extra requirement #1**] You are not allowed to invoke the `system(3)` library call. Otherwise, you would score 0 marks for this assignment.
2. [**Extra requirement #2**] You are not allowed to invoke any existing shell programs in this assignment. Otherwise, you would score 0 marks for this assignment. One exception is that the input command lines is to invoke those existing shell programs.

   Note that the phrase *"existing shell programs"* implies the shell programs installed in the operating system including, but not restricted to, "`/bin/sh`" "`/bin/bash`", etc.

3 Milestones and Deliverables

This assignment contributes 18% of the course grade. We break this assignment into **2 phases**. You must use either C or C++ to implement your assignment. **Otherwise, you would receive zero marks**.

3.1 Overview

The assignment is divided into following phases of development.

Task	Marks	Phase 1	Phase 2
Reading & tokenizing input	2.0%	•	
Built-in command: `exit` & `cd`	0.5%	•	
Executing commands with arguments, without pipe	3.0%	•	
Handling signals	1.0%		•
Executing commands with pipes	6.0%		•
Wildcard expansion	1.0%		•
Job control without pipes	2.0%		•
Job control with pipes	2.0%		•
Built-in command: "`jobs`"	0.5%		•

3.2 Phase 1: simple shell program (5.5%)

You have to implement a basic shell using `fork()`, `execve()` system call family, and `waitpid()`. This is an individual task.

- **Reading & tokenizing input**. Your OS shell has to read and interpret the input. For Phase 1, no pipes would be supplied, i.e., only commands with one program name, the "`exit`", and the "`cd`" built-in commands will be supplied.
- **Built-in command:** `exit` **&** `cd`. Your implementation should perform the expected actions of the above two built-in commands.
- **Executing commands**. Here comes the use of the system calls `fork()`, `execve()` family, and `waitpid()`. Please note the following:
  - No pipes. Thus, only one new process is created for each command line.
  - No job control. That means we are not going to suspend the foreground job.
  - Detection of process termination. Note that we will kill the foreground job using ways including, but not limited to, Ctrl + C and the `/bin/kill` program.
  - Unlimited amount of program arguments. Of course, the length of the input command is limited to 255 bytes.

- **Handling signals**. (This task is moved to Phase 2) The shell has to ignore some of the signals as described earlier. Nevertheless, the child processes should be handling the involved signals with the default handlers.
- **Deliverables**. Phase 1 is an individual work. Every student, both CSCI 3150 & ESTR 3102, must submit a set of source files written by yourself. **Never submit any executables**. The files should be compiled into one executable only.

Deadline: 23:59, October 5, 2015 (Monday).

3.3 Phase 2: the complete shell (12.5%)

Phase 1 is just a warmup and Phase 2 is the real thing. Therefore, it is a group work. Note that Phase 2 is NOT extending Phase 1. You and your partner can discuss which / whose codes would be adopted in Phase 2.

- **Executing commands with pipes**. This part will be quite time consuming because your OS shell has to create processes and run them in parallel. In this milestone, you may wonder how to detect the termination of a job. We define:

  > A job (or command line) is terminated if all processes of the corresponding job are terminated.

- **Wildcard expansion**. The wildcard expansion must be coupled with the execution of the command line. Henceforth, we will test your program with inputs with and without pipes.
- **Job control without pipes**. Here comes the suspension of a process and the built-in command '`fg`'. Your shell implementation must be able to detect the suspension of the foreground process and then display the shell prompt again.

  When the user inputs the built-in command "`fg`", your shell implementation should be able to bring the suspended process back to the foreground.

- **Job control with pipes**. We are going further to support a job with multiple processes. In addition, we define what the meaning of a suspended job:

  > A job (or command line) is suspended if all processes of the corresponding job are suspended.

  Your shell implementation must be able to detect the suspension of the foreground job and then display the shell prompt again.

  When the user inputs the built-in command "`fg`", your shell implementation should be able to bring the suspended job back to the foreground.

- **Built-in Command: "`jobs`"**. This is a simple account job to show the list of suspended processes.

- **Note**. Since the milestones are incremental by design, marks would be lost if a bug from an earlier milestone appears when you are showing us a latter milestone.

  Say you had passed the tests for running a simple command with pipes. However, a hidden bug in handling pipes manifested while you were demonstrating the job control features. Then, you would lose marks for the concerned testcase in the job control milestone, but not the marks for the pipe features.

- **Deliverables**. You are required to submit a set of source files. **Never submit any executables**. The files should be compiled into one executable only.

Deadline: 23:59, October 19, 2015 (Monday).

3.4 Marking

- The marking of the Phase 1 program will be separated from the Phase 2 program.

- Well, there would be unhappy scenarios that the program you submitted in Phase 1 was incorrect, and you discovered that while you were working on Phase 2. Then, you may ask:

"*can I re-submit Phase 1?*"

The answer is "**No, you cannot**".

- We will go through days of **demonstrations** in order to grade your submissions in the computing lab.
- You and your partner both have to attend the demonstration and instruct our tutors how to work with your submissions. This is to make sure that you can appeal for the results right at the spot. Please stay tune with the course homepage about the dates of the demonstration.
- We grade your assignment based on the test inputs with the expected outputs. We would never look into your code, locate your faults, or give you partial marks.
- We grade your work with human efforts, not by any automatic grading platforms. Nonetheless, please remove any debugging messages before you submit. You know, those debugging messages would risk your grade if your normal output was too hard to be located.

Submission Guideline

For the submission of the assignment, please refer to the following link:

http://course.cse.cuhk.edu.hk/~csci3150/

Last reminder:

Phase 1 Deadline: 23:59, October 5, 2015 (Monday).

Phase 2 Deadline: 23:59, October 19, 2015 (Monday).

–END–

Change Log

Time	Details
2015 September 17	V1.0: first release.
18:30, 2015 Oct 1	V1.1: we removed the signal handling task from Phase 1. The same task is now in Phase 2.