Let's Write a Shell!

2012-2013 CSCI 3150 - Programming Assignment 1

Specification version 1.0.1, Sep 16, 2012.

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 (500 - 1,000 lines of codes);

    - 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.
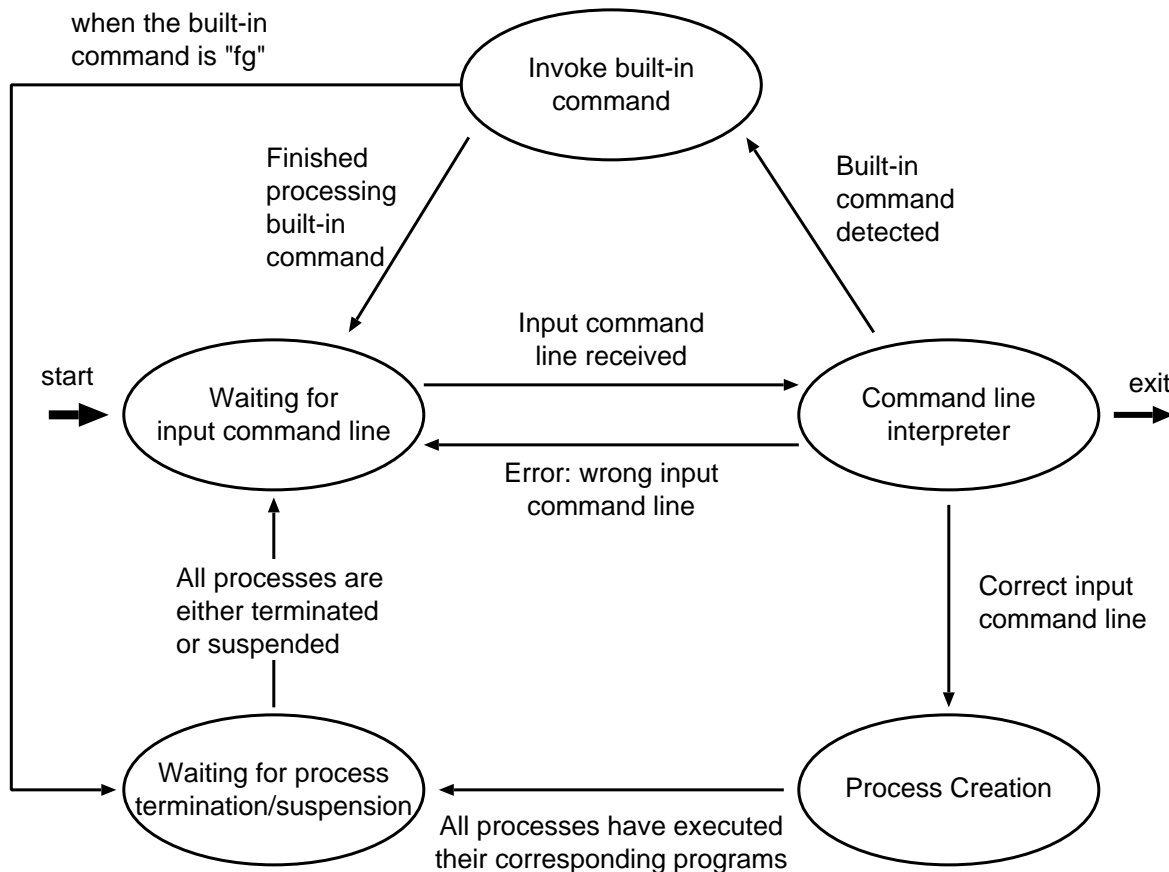

Figure 1: The shell's execution flow.

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]$ _
```

where "/home/tywong" is the 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/more
```

Then, the received input will be processed by the "**Command line interpreter**" of the OS shell, and we called the input, "`/bin/ls | /bin/more`", the **input command line**.

- If the command line interpreter finds that the input command line **agrees with the pre-defined syntax**, which will be defined in Section 2.2, then the OS shell should invoke the commands specified by the input command line.

- Else, the input command line will not be executed and the OS shell waits for another input command line, i.e., going back to the state "*Waiting for input command line*".

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.
2. An input command line ends with a carriage return character '$\backslash n$'.
3. There is no leading nor trailing space characters in the input command line.
4. A **token** is a series of characters without any space characters. Each **token** (or word) in the input command line is separated by at least one space character.

2.2 Command line interpreter

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.

Here comes the language used by the shell: the shell takes similar language to the one used by the *bash* shell and is specified clearly as follows:

```
   [start]   :=   ""; or
             :=   [built-in command] [arg]*; or
             :=   [command] '<' [filename] [recursive]; or
             :=   [command] '<' [filename] [terminate]; or
             :=   [command] [recursive]; or
             :=   [command] [terminate] < [filename]; or
             :=   [command] [terminate].
 [recursive] :=   '|' [command] [recursive]; or
             :=   '|' [command] [terminate].
 [terminate] :=   ""; or
             :=   '>' [filename].
             :=   '>>' [filename].
```

```
      [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. Also, the command name is assumed to be different from every built-in command of this shell.

```
          [arg]   :=
```
A string without any space, tab, > (ASCII 62), < (ASCII 60), | (ASCII 124), * (ASCII 42), ! (ASCII 33), ' (ASCII 96), ' (ASCII 39), nor " (ASCII 34) characters.

```
[filename]  :=  A string without any space, tab, > (ASCII 62), < (ASCII
                60), | (ASCII 124), * (ASCII 42), ! (ASCII 33), ` (ASCII
                96), ' (ASCII 39), nor " (ASCII 34) characters.
```

> **Important**: in order to ease your pain in implementing this assignment,
> we restrict the maximum number of pipes, i.e., "|", in a command line to
> 2 only.

Some examples of valid command line inputs are as follows.

- blank line;

- "`cat | cat`", and "`cat | cat | cat`";

- "`cat > out.txt < in.txt`", and "`cat < in.txt > out.txt`";

- "`cat | cat > out.txt`"; etc.

Some examples of invalid input command lines are as follows.

- "`cat >`", "`cat <`", "`cat |`", or "`| cat`";

- "`cat > out.txt | cat`";

- "`cat | cat < in.txt`";

- "`cd / > out.txt`"; etc.

- "`cat | cat | cat | cat`" (due to the extra requirement).

> **Important**: you are not required to check whether a program has to work
> with the standard input and the standard output or not before involving
> an input redirection and an output redirection, respectively.

What does the above "*Important*" notice mean? For example, the command line input
"`/bin/ls < in.txt`" should have the same effect as the command line input "`/bin/ls`"

because the program "`/bin/ls`" does not read any data from the standard input stream. Nevertheless, the OS shell is not required to find out this fact. The OS shell still redirects the standard input/output stream as instructed.

Every valid input command line should begin with the [**start**] rule. Any input command line that does not agree with the above language is considered to be wrong. The following shows are the requirements of the command line interpreter:

1. If an input command line violates the above language, the shell has to return an message "**Error: invalid input command line**" before prompting for the next command.

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

2.3   Process creation

When the input command line is valid, the OS shell should **create the processes** according to the names specified in the input command line.

2.3.1   Finding the programs

There are two kinds of paths:

- If a name begins with the character '/', then that name specifies an **absolute path**.

- Else, that name specifies a **relative path**.

- Special cases are names starting with "./" and "../". Though the names are not starting with '/', they are considered as absolute paths.

If a program name is specified by an **absolute path**, e.g., /bin/ls or ./a.out, then the creation of the process is straight-forward. Else, if a command name is specified by a **relative path**, 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". For example:

Example.
[3150 shell:/home/tywong]$ /bin/sl	/bin/cat
/bin/sl: command not found.
[3150 shell:/home/tywong]$ _

9

Note that:

- The `exec` 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 absolute and relative paths). If the `errno` specifies other errors, then the shell should report:

  $$\texttt{[command name]: unknown error}$$

- 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 resource.
`fork()`, `exec` system call family, `setenv()` library call.

2.3.2 I/O Redirection

You are required to implement the **I/O redirection feature** in this assignment. You are only required to redirect the **standard input stream** and the **standard output stream**, but not the **standard error stream**. You have to implement the three following features: **input redirection**, **output redirection**, and **pipe**.

Input Redirection

Input redirection, basic rule
`[command name] [arg]* < [filename]`

What does this rule mean? The standard input stream of the program named `[command name]` is replaced by the file named `[filename]`. If the `[filename]` does not exist, then the shell program should not invoke the command and should report the following error message:

Input redirection, error message
`[filename]:  no such file or directory`

Input redirection, example
`[3150 shell:/home/tywong]$ cat input.txt`
`hello`
`[3150 shell:/home/tywong]$ cat < input.txt`
`hello`
`[3150 shell:/home/tywong]$ cat < input`
`input:  no such file or directory`
`[3150 shell:/home/tywong]$ _`

where the program "`cat`" (its path is "`/bin/cat`") behaves as follows.

- If the filename is specified as an argument, `cat` reads data from the specified file, and prints the read content to its standard output stream; or

- If no filename is specified as any argument, `cat` reads data from its standard input stream, and prints the read content to its standard output stream.

Output Redirection

Output redirection, basic rule 1
`[command name] [arg]* > [filename]`

What does this rule mean? The standard output stream of the program named `[command name]` is replaced by the file named `[filename]`. Concerning the `[filename]`, the handling is a little bit troublesome:

- If the `[filename]` does not exist, then the OS shell should **create** a new file named `[filename]`.

In case that the OS shell is not allowed to create the file `[filename]`, then the OS shell should report the error message "`[filename]: Permission denied`".

- If the `[filename]` exists, then the OS shell should **truncate** the file, meaning that the OS shell should erase all the existing content inside "`[filename]`".

In the case that the OS shell is not allowed to write to "`[filename]`", then the OS shell should not invoke the command and should report the error message "`[filename]: Permission denied`".

Output redirection, example 1

```
[3150 shell:/home/tywong]$ ls
[3150 shell:/home/tywong]$ cat > output.txt
Showing how to create "output.txt"
^D
[3150 shell:/home/tywong]$ ls
output.txt
[3150 shell:/home/tywong]$ cat output.txt
Showing how to create "output.txt"
[3150 shell:/home/tywong]$ cat > output.txt
Showing how to truncate "output.txt"
^D
[3150 shell:/home/tywong]$ cat output.txt
Showing how to truncate "output.txt"
[3150 shell:/home/tywong]$ _
```

Output redirection, basic rule 2

```
[command name] [arg]* >> [filename]
```

Rule 2 is again the redirection of the standard output stream. However, when the "`[filename]`" exists, the OS shell should **append** the new content to the existing content, instead of erasing the existing content. In the case that the OS shell is not allowed to append new data to the "`filename`", the OS shell should report the following error message:

Output redirection, error message
`[filename]:  Permission denied`

Output redirection, example 2
`[3150 shell:/home/tywong]$ cat output.txt`
`first message`
`[3150 shell:/home/tywong]$ cat >> output.txt`
`second message`
`^D`
`[3150 shell:/home/tywong]$ cat output.txt`
`first message`
`second message`
`[3150 shell:/home/tywong]$ _`

Pipe

Pipe, basic rule
`[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 I/O redirection
dup() / dup2(), open(), close(), and pipe().

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[1], the OS shell should wait for the next input command line.

Useful system calls.
wait() and waitpid().

Note important 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 ~~not~~ 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]A process can stop running because of normal or abnormal reasons.

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 <u>one argument</u>. 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`". Note that if there are suspended jobs, the OS shell will not be terminated.

   > **"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 17.

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

15

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	Ignore the signal.
SIGTERM	Ignore the signal.
SIGQUIT	Ignore the signal.
SIGTSTP	Ignore the signal.
SIGSTOP	Default signal handling routine.
SIGKILL	Default signal handling routine.

The signals that are not listed in the above table should adopt the default signal handling routines of the corresponding signals. Note important that only the OS shell changes the handling of the said signals. The child processes of 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 the an incomplete set 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 resumption 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 command termination/suspension**" 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
[3150 shell:/home/tywong]$ _
```

Note 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
[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.)

3 Milestones and Deliverables

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

3.1 Phase 1: command line interpreter + cd + exit (6%)

You have to implement a correct command line interpreter.

3.1.1 Input and Output

1. **Input**. The interpreter reads input command lines from the standard input stream (`stdin` in C). The program should stop when it encounters the "`exit`" built-in command.

2. **Output - for correct inputs**. If the syntax of the input command line is correct, the interpreter should break the input into **tokens** and print out the type of the tokens line by line.

   The output format is given as follows:

Phase 1 output format
Token 1:   "Content" (Type)
Token 2:   "Content" (Type)
...
Token n: "Content" (Type)

   where "`Content`" is the token itself and "`Type`" can be either one of the following:

Phase 1 output format, "Type"			
Redirect Input	Redirect Output	Built-in Command	Argument
Pipe	Command Name	Input Filename	Output Filename

```
┌────────────────────────────────────────────────────────────┐
│ Phase 1 input and output example                             │
├────────────────────────────────────────────────────────────┤
│                                                              │
│ [linux3:/home/tywong]$ ./milestone1                          │
│ [3150 shell:/home/tywong]$ ls -l                             │
│ Token  1: "ls" (Command Name)                                │
│ Token  2: "-l" (Argument)                                    │
│ [3150 shell:/home/tywong]$ cat > txt                         │
│ Token  1: "cat" (Command Name)                               │
│ Token  2: ">" (Redirect Output)                              │
│ Token  3: "txt" (Output Filename)                            │
│ [3150 shell:/home/tywong]$ cat >_<                           │
│ Error: invalid input command line                            │
│ [3150 shell:/home/tywong]$ cd /                              │
│ Token  1: "cd" (Built-in Command)                            │
│ Token  2: "/" (Argument)                                     │
│ [3150 shell:/]$ cd / /                                       │
│ cd: wrong number of arguments                                │
│ [3150 shell:/]$ exit                                         │
│ Token  1: "exit" (Built-in Command)                          │
│ [linux3:/home/tywong]$ _                                     │
│                                                              │
└────────────────────────────────────────────────────────────┘
```

3. **Output - for incorrect inputs**. If the syntax of the input command line is incorrect, the interpreter should print the message "**Error: invalid input command line**" to the standard output stream (`stdout` in C).

3.1.2 Requirements

- The interpreter recognizes built-in commands whenever a correctly-placed built-in command is encountered.

- The interpreter is not required to create processes, to execute the programs, nor to

recognize if the programs could be located.

- The interpreter should have both the "`cd`" and the "`exit`" built-in commands implemented correctly.

3.1.3 Deliverables

You are required to submit a set of source files. **Never submit any executables**. The files should be compiled into <u>one executable</u> only.

Deadline: 23:59, Oct 5, 2012 (Friday).

3.2 Phase 2: the shell (12%)

With a working interpreter, the shell can make use of the result from the interpreter to create processes.

In order to ease your pain in designing the program, we are going to break the programs into milestones (marks are attached, too). Notice that those milestones are **incremental**. E.g., Milestone 2 is building on top of Milestone 1. You are recommended to finish the milestones in the order they are presented.

3.2.1 Overview

Task	Marks	M1	M2	M3	M4	M5
Commands with arguments	2%	●	●	●	●	●
With input & output redirection	2%		●	●	●	●
With pipes	4%			●	●	●
Signals	1%				●	●
Job control + `jobs` + `fg`	3%					●

* Without loss of generality, "M1" means *Milestone 1.*

3.2.2 Phase 2 milestones

- **[Milestone 1].** This requires you to implement a simple shell. You should integrate the command line interpreter from Phase 1 into the shell.

- **[Milestone 2].** This requires you to implement both the input and the output redirections, i.e., ">" and "<". Of course, the two operators may appear in the same command line.

- **[Milestone 3].** This requires you to implement the support for the pipes. Remember, there are **at most two pipes** in a command line.

- **[Milestone 4].** This requires you to handle signals as instructed in Section 2.6 on Page 16.

- **[Milestone 5].** Last but not least, you have to implement the job control features described in Section 2.7 on Page 17 together with the associated built-in commands.

Note that 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.

For example, say you had passed the tests for Milestone 3, i.e., 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.

3.2.3 Deliverables

You are required to submit a set of source files. **Never submit any executables**. The files should be compiled into <u>one executable</u> only. Therefore, we should better plan ahead so that your phase 1 program can smoothly integrated into phase 2.

Deadline: 23:59, Oct 21, 2012 (Sunday).

3.3   Marking

The marking is separated based on the two phases mentioned.

- The marking of the interpreter program will be separated from the shell program.

- There can be cases that the program you submitted in Phase 1 is incorrect. Then, you may ask: *"how can I finish the assignment with a buggy interpreter?"*

  Don't worry. We allow the update of the interpreter's codes. Please just put the new code together with the shell program in the same submission package. As a matter of fact, we will mark your assignment as follows.

  - We will grade your interpreter from your phase 1 submission.
  - Then, we grade your shell from your phase 2 submission.
  - We will not re-grade your interpreter from phase 2.
  - If bugs appeared during the grading of the shell, we would never distinguish the source of the bugs (from phase 1 or not). We consider them as incorrect implementations of the involved features of the shell.

- We grade your assignment based on the input and the output. We seldom look into your code, locate your faults, nor give you partial marks.

Last but not least, we will go through days of **demonstrations** in grading your submissions. 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 newsgroups about the dates of the demonstration.

Submission Guideline

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

http://www.cse.cuhk.edu.hk/csci3150/

Last reminder:

Phase 1 Deadline: 23:59, Oct 5, 2012 (Friday).

Phase 2 Deadline: 23:59, Oct 21, 2012 (Sunday).

<div align="center">

–END–

</div>

Change Log

Affected Version	Time	Details
1.0	22:30, 2012 Sep 16	Page 14. Removing a word "*not*".