Process Control

2015-16 Term 2, CSCI 3150 - Programming Assignment 1

Specification version 1.0, 2016 Feb 15

Abstract

The purpose of this assignment is to introduce students to the existing useful system calls in the Linux environment, controlling the creation as well as the termination of processes. Last but not least, you will learn how to limit the running time of running processes through the use of POSIX signal.

Contents

1 Introduction

In this assignment, there are two deliverables:

1. a simple shell which accepts command lines with multiple command line arguments, and
2. a simple scheduler which accepts command lines and launch processes in parallel or in a FIFO manner.

You know, 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 most important component of a shell is to management created processes.

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. In terms of process management:

1. First deliverable controls one parent and one child process only.
2. Second deliverable controls a group of processes.

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

- Understand the internal workings of a shell;
- Control multiple number of processes;
- Write a medium-sized program ($>$ 300 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 call or library function;

2 Deliverable 1 - Simple Shell

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.

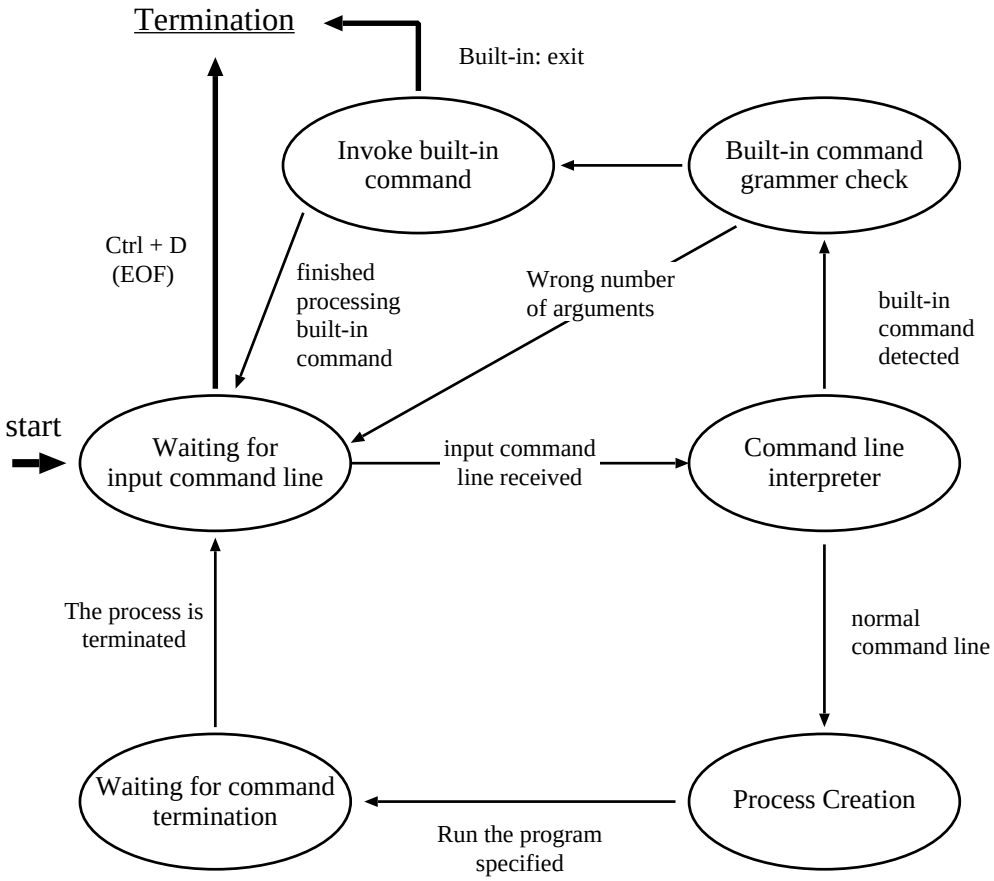


Figure 1: The shell's execution flow.

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 a command followed by a **carriage return** (or the "enter" key), e.g.,

```
[3150 shell:/home/tywong]$ /bin/ls -l
```

Then, the received input will be processed by the "**Command line interpreter**" of the OS shell, and we called the input, "`/bin/ls -l`", 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 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 exactly one space character.

2.2 Command line syntax

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

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]`
`[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), or ” (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), or ” (ASCII 34) characters.

Some examples of valid command line inputs are as follows.

- blank line
- Simple command: “`cat`”, “`cat *`”, and “`cat dog`”

Next, 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 child process** according to the name 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 programs 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.

2.4 Detecting process termination

Once the process is created, the shell has to wait until **all the processes have stopped executing**. In this assignment, we only consider the case that the child process stops its execution:

- due to the execution of the program codes, such as running `exit()` system call, or
- due to the incoming signals and the default signal handling routine is termination.

After the processes have been terminated, the OS shell should wait for the next input command line.

Useful system call for this stage.
`waitpid()` or `wait()`.

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 **two** 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`".

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.

3 Deliverable 2 - Simple Scheduler

We extend Deliverable 1 to order to implement a simple scheduler:

- Unlike an interactive shell, the scheduler reads a list of command lines. It then executes the command lines.
- The scheduler has two modes: the FIFO mode and the parallel mode.

3.1 System Design

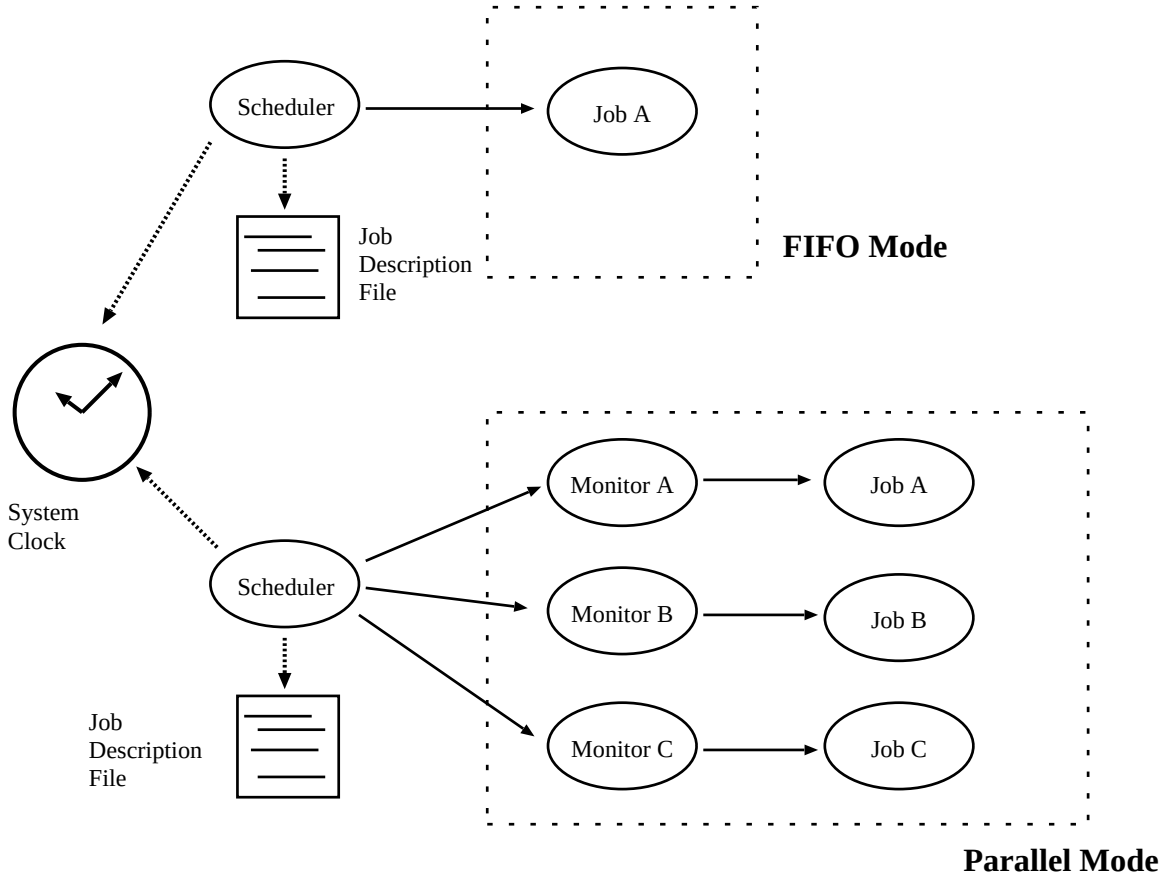
Our design involves several entities and are depicted in Figure 2. In the following subsections, we introduce them one by one.

3.2 Scheduler Process

In this assignment, our scheduler only implements limited functionalities, comparing to the real scheduler in the modern operating systems.

- The scheduler is *an ordinary process*, i.e., no special privilege should be given to the scheduler.
- The scheduler creates a set of jobs that are described in the *job description file*. The job description file contains a list of jobs that the scheduler should execute.
- While scheduling, the scheduler reads from the **system clock** to determine the timing in scheduling.
- The scheduler should not be terminated until all the jobs are scheduled and are terminated.

In a nutshell, the scheduler is an extension of Deliverable 1. The major difference is that the command lines are no longer provided by the user in an interactive way.



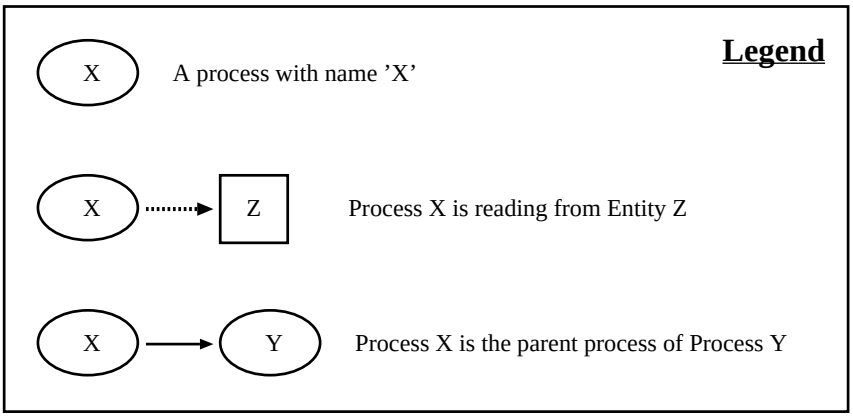


Figure 2: The system design involves multiple processes working together.

3.3 System clock

It is the current time value that is maintained by hardware. For the sake of convenience, we restrict the granularity of the time to be **in terms of seconds**.

Although it is easy for us to read the clock in the scale of microseconds (10^{-6}), this is not our educational goal since this only complicates the story; we planned to avoid that. The only one disadvantage of this design is sacrificing the precision of the scheduling. We will discuss this later in this writing.

3.4 Job description file

This is a plain-text file that stores a list of jobs in the format shown in Figure 3. We describe the format shown as follows.

Job #1	`Command`	`\t`	`Duration`	`\t`
Job #2	`Command`	`\t`	`Duration`	`\t`
Job #3	.....			

Figure 3: The format of the job description file.

- There are at most **ten jobs** stated in a job description file.
- One line represents one job, i.e., every job description is separated by one newline character.
- A line of job description must contain two non-empty fields: "`Command`" and "`Duration`". They are separated by exactly one tab character.
- The "`Command`" is a command line with the following constraints. Such a command line follows all rules as well as all assumptions in Deliverable 1.

    1. The first token may be the name of the program or the path to the program.
    2. You can assume that the first token would never be a built-in command.
    3. You can assume that the scheduler can execute the command line successfully.

- The "`Duration`" is the number of seconds that the command is allowed to run. The possible values of the "`Duration`" can be `-1` plus all non-zero positive integers.

    1. When the value is `-1`, it means the command is allowed to run indefinitely until it terminates.
    2. For other values of the "`Duration`" field, that means the command is allowed to run for a limited time. After the said time has been passed, the scheduler would terminate that process.

    Note very important that the "`Duration`" is not the **accumulated CPU time of the process**, but the time elapsed after the job has been started. Although some may feel that it is a lousy implementation, this is so designed in order to lift your workload. Please stick with it.

3.5 Job Process

The job process executes the command stated in the job description file. It is created by the monitor process using the "`fork()`" system call and the "`exec`" system call family.

- A job process may print output to the console through both the standard output and the standard error streams.
- A job process may end before or after the time stated in the "`Duration`" field in the job description file.
- It is assumed that a job process would not read from the standard input stream.
- It is assumed that no signals would be sent from processes outside our scheduler system to a job process.
- It is assumed that a job process would not have any custom signal handling routine installed, meaning that every signal received should work with its default handler.

Requirement
After the time specified by "`Duration`" has been elapsed, the system should terminate the job process with the `SIGTERM` signal.

3.6 Scheduling Report

When one job finishes executing, your system should report the following:

```
<<Process 1234>>
time elapsed: 0.1900
user time   : 0.1000
system time : 0.0100
```

where,

- "`1234`" is the PID of the job process of this example.
- The "`time elapsed`" is the time period counting from the start of the job process to the termination of the job process, and it is expressed in terms of seconds.
- The "`user time`" is the user CPU time received by the job process, and it is expressed in terms of seconds.
- The "`system time`" is the system CPU time received by the job process, and it is expressed in terms of seconds.

As long as all reports from all jobs are printed before the scheduler process stops, the moment that the scheduling report is printed is not important.

Note that the "`time elapsed`", the "`user time`", and the "`system time`" can all be obtained by the correct use of the "`times(2)`" system call.

3.7 Execution Modes - FIFO and Parallel

Let us describe how the scheduler process works under the two execution modes with the following example job description file:

```
ls              10
./while1        3
ls -R /home     -1
```

3.7.1 FIFO Mode

- **At Time = 0**. The scheduler process starts and reads the job description file.

  Run the first task. The first job has one process only and is allowed to run for 10 seconds. While the first job is running, the scheduler should be blocked until:

  - The child process terminates before the 10-second timeout, or
  - 10 seconds is elapsed. For this case, the scheduler should send a `SIGTERM` signal to the child process in order to terminate it.

  In this example, the "`ls`" program should terminate after a very short amount of time. The termination of the job process should then wake up the scheduler.

- **After the first command line has terminated**. The scheduler process starts the second job. According to the example, it is allowed to run for 3 seconds only.

  Suppose that the second command is an infinite loop. The scheduler should wake up after 3 seconds, and then terminates the process using the `SIGTERM` signal.

- **After the second command line has terminated**. The scheduler process starts the third job. In this time, the job runs until it ends.

- **After the third command line has terminated**. The scheduler process terminates.

Useful system calls and library functions for this stage.
`alarm()`, `signal()`, `kill()`, `pause()`

3.7.2 Parallel Mode

Unlike the FIFO mode, in this mode, all command lines run in parallel. The scheduler process runs all of them at the same time.

The challenge of this implementation is that the scheduler has to keep track of time limits for all command lines! Since this is a challenge, we would not write too much about the implementation details, except the **monitor processes**.

3.8 Monitor Process - for Parallel Mode Only

You may be wondering why there is a monitor process. Of course, it exists not because we want to make things difficult. Rather, it is because of the "`times()`" system call.

Basically, the "`times()`" system call can collect statistics of terminated child processes. Yet, what the system call has collected is a set of aggregated statistics about all the children of a process. That is to say, if our scheduler wants to collect the running times of all children and print them out separately, it is impossible without the introduction of the monitor process. As a middleman with only one child process, the `times()` system call invoked by the monitor process will correctly report the statistics of every child. By the way, the monitor process works similar to the program "`/usr/bin/time`".

- The scheduler process "*forks*" the monitor process.
- The monitor process "*forks and executes*" the command specified. That command becomes the job process.
- Then, when the job process terminates, the monitor process should know about it.
- The monitor process then prints the scheduling report of the execution of the job process. In case that you forget what the scheduling report is:

```
<<Process 1234>>
time elapsed: 0.1900
user time   : 0.1000
system time : 0.0100
```

How to terminate processes when "`Duration`" is up? This is the challenge that we set up for you. Let me give you a hint:

The monitor process is a FIFO scheduler with one job only!

4 Extra requirements to Both Deliverables

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

5 Milestones and Deliverables

This entire assignment is an individual assignment, contributing 20% to the course grade. You must use either C or C++ to implement your assignment. **Otherwise, you would receive zero marks**.

5.1 Grading Overview

Task	Marks	Deliverable 1	Deliverable 2
Reading & tokenizing input	2.0%	•	
Built-in command: `exit` & `cd`	1.0%	•	
Executing commands with arguments	3.0%	•	
Handling signals	1.0%	•	
Wildcard expansion	1.0%	•	
Continuous execution	2.0%	•	
FIFO Scheduler: 1 job only	2.0%		•
FIFO Scheduler: more than 1 job	3.0%		•
Parallel Scheduler: at most 2 jobs	2.0%		•
Parallel Scheduler: more than 2 jobs	3.0%		•

5.2 Deliverable 1: simple shell program (10%)

You have to implement a basic shell using `fork()`, `execve()` system call family, and `wait()` (or `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:
  - 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**. 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.
- **Continuous execution**. Your simple shell process should still be able to serve the user after the job process is terminated.
- **Deliverables**. Every student must submit a set of source files written by yourself. **Never submit any executables**. The files should be compiled into one executable only. For the ease of our grading, please name the compiled executable: `shell`.

Deadline: 23:59, March 4, 2016 (Friday).

5.3 Deliverable 2: the simple scheduler program (10%)

This deliverable is the simple scheduler described earlier. Every student must submit a set of source files written by yourself. **Never submit any executables**. The files should be compiled into one executable only. For the ease of our grading, please name the compiled executable: `scheduler`.

5.3.1 Usage of the program

The program should be executed in the following manner:

- FIFO Mode

```
./scheduler FIFO filename.txt
```

- Parallel Mode

```
./scheduler PARA filename.txt
```

For the arguments,

- **First Argument**. It is the selection of the modes. You can assume that we are always providing correct argument values.
- **Second Argument**. It is the pathname to the job description file, i.e., "`filename.txt`" in the example. Of course, the filename is not restricted to the name "`filename.txt`"; it can be any pathname.

  You can assume that the pathname always points to a valid, regular text file. The text file is always having the correct format of the job description file.

Deadline: 23:59, March 18, 2016 (Friday).

5.4 Marking

- The marking of the two program will be separated.
- We will go through days of **demonstrations** in order to grade your submissions in the computing lab.
- You 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 a common set of 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.

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

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

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

Submission Guideline

For the submission of the assignment, please refer to our course homepage:

CUHK network only: `http://course.cse.cuhk.edu.hk/~csci3150/`

Last reminder:

Deliverable 1 Deadline: 23:59, March 4, 2016 (Friday).

Deliverable 2 Deadline: 23:59, March 18, 2016 (Friday).

–END–

Change Log

Time	Details
2016 Feb 15	V1.0: first release.