Assignment #2

CS 180 Fall 2020

Deadline:	Part 2: As specified on the moodle
Topics covered:	Process creation and management
Deliverables:	To submit all relevant files that will implement the uShell program and provide a Makefile. Please zip up the file according to the naming conventions laid out in the cs180 syllabus. Your program must be compile-able in the Cygwin environment using the `/usr/bin/gcc` or g++ compiler.
Objectives:	To learn and understand the basic of the internals of a shell program.

Programming Statement: A new shell program called uShell

In this assignment, you are tasked with the assignment to create a basic shell program called **uShell**. A shell program behaves typically in a similar fashion to a command prompt. Usually, it has the following capabilities:

- Ignore comments statement
- Perform internal commands (Part 1)
- Run programs and launch processes (Part 1)
- Setting of environment variables (Part 1)
- Perform background process (Part 2)
- Performing piping as a means of interprocess communications between processes (Part 2)

1 Introduction to part 2

In part 2, the student is expected to extend the same `uShell` program in part 1. In particular, the student is expected to implement the following:

- An internal command called `finish`, to force the shell program to wait for background processes.
- Run background processes by supporting a syntax when a word "& " is appended to the end of the command line.
- Support multiple pipes between commands.

In this specification, we shall concentrate on describing these additional features. Please consult part 1 for the other specifications.

2 Background commands and synchronization

The default behaviour of external commands is that they are executed before the next command is read. However, background jobs provide an opportunity for the user to launch jobs that the user does not have to wait for before running the next command. An example is `mspaint`, which is a program that will open up the Microsoft Paint program in a window. Usually, one would need to close the window before continuing, but background jobs give us a way to avoid that problem. The syntax for running a background job is by adding a "& " to the end of the command. When a background job is launched, the shell program responds by printing the process index (internal to the shell) and the process id (via the function `getpid()`). For example:

```
uShell> mspaint.exe & # A window for mspaint program should pop up after this
[0] 1496
uShell> # mspaint is the first background job to launch, hence 0.
uShell> # mspaint's process id from getpid is 1496.
uShell> mspaint.exe& # wrong use of the &
Error: mspaint.exe& cannot be not found.
```

2.1 finish - Internal Command

The internal command that can be used in connection with this is the internal command `finish`. `finish` forces the shell program to wait for particular background processes to complete before continuing. The format for using `finish` is `finish` <process index>. The given process ID must be a valid one. For example:

```
uShell>notepad.exe &
[0] 10064
uShell>finish 0
process 10064 exited with exit status 0.
uShell>notepad.exe &
[1] 12456
uShell>finish 0
Process Index 0 Process ID 10064 is no longer a child process.
uShell>finish 1
process 12456 exited with exit status 0.
```

2.2 Pipes

Finally, an additional feature of `uShell` is the support of pipes. This is also a composite external command, where multiple external commands are executed concurrently. The syntax for pipes is the following:

```
ext_cmd1 | ext_cmd2 | ext_cmd3 | ...
```

where each `ext_cmd` is a sequence of words that form any legitimate external command as described above. The only difference is that these external commands cannot be background jobs (but you are not expected to perform error handling of this). You do not have to support piping with internal commands, although that may be a useful extension to implement.

Semantically, what the syntax mean is that the output of the preceding external commands is fed into the succeeding external commands. So the output of `ext_cmd1` is feeding into the output of `ext_cmd2` and so on and so forth. It is important to note that these commands are running concurrently, so there is no need to wait for `ext_cmd1` to complete before `ext_cmd2` runs. Finally, the last external command in the chain of pipes outputs to the stdout. Here are some examples of pipes:

```
uShell>ls.exe
a.exe config.obj my_setenv.c test-regex.cpp
a.exe.stackdump debug_print.cpp my_setenv.h test-template.cpp
command.h debug_print.h my_setenv.o variable-store.cpp
command.h.orig errors.txt parse.cpp variable-store.h
config.cpp main.cpp parse.cpp.orig variable-store.obj
config.cpp.orig main.cpp.orig parse.h
config.h main.o parse.h.orig
config.h.orig Makefile parse.o
uShell>ls.exe | grep.exe setenv
my_setenv.c
my_setenv.h
my_setenv.o
uShell>ls.exe | grep.exe setenv | wc
wc: command not found
uShell>ls.exe | grep.exe setenv | wc.exe
3 3 36
```

3 Rubrics

The total marks for this assignment is 100. The rubrics are as the following: commands piped together.

- (50 marks) Your background jobs are launched properly and finish works as expected, including the error messages that are described above.
- (50 marks) Pipe commands should work as expected for any number of external.