CS 4420/5420

`Assignment 2
PARTS I & II

The New BASH Shell: Shell Skeleton

Max. Points: 50

Due: (announced on blackboard)

PART I

The Shell Skeleton:

In the two parts of the second assignment, you will eventually design and implement a simple shell command interpreter, called mybash, which is a subset of a typical UNIX shell. The basic function of a shell is to accept lines of text as input and execute programs in response. The command input must be in the following format (explained in more detail below):

```
cmd (arg)*  (<infile)? (| cmd (arg)* )* (>(>)? outfile)? (&)?
```

The shell will eventually have these basic features[1]:

- Execution of *simple commands*: a simple command consists of a command name followed by an optional sequence of arguments separated by spaces or tabs, each of which is a single word. A command may have up to 10 arguments. While a shell can have built-in commands (see below), it must be able to execute simple commands existing as independent files in the file system.
- *I/O Redirection*: A simple commands standard input may be redirected to come from a file by preceding the file name with "<". Similarly, the standard output may be redirected with ">", which truncates the output file. If the output redirection symbol is ">>", the output is appended to the file. An output file is created if it doesn't exist.
- A *pipeline* consists of a sequence of one or more simple commands separated with bars "|". Each except the last simple command in a pipeline has its standard output connected, via a pipe, to the standard input of its right neighbor.
- A pipeline (and a simple command) is terminated with a newline ("\n", [RETURN]), or an ampersand "&". In the first case the shell waits for the rightmost simple command to terminate before continuing. In the ampersand case, the shell does not wait.

[1] These are the basic features that you will need to implement in Part II. Some additional features will be added in assignment 3.

- Built-in commands: Built-in commands are assignment, *set, cd, exit, jobs, kill, fg, bg, pwd*[2].

Here are some examples of valid command lines:

```
ls
ls -l > file1
cat < infile | grep 14877 | wc
cat < infile | grep file1.txt > outfile &
prog1 arg1 arg2 arg3 arg4 < file2.txt | prog2 arg1 | prog3 > outfile
```

I will provide you with three headstart files:

- parser.c: include functions to parse command lines entered by the user,
- a skeleton mybash.c file, and
- a makefile.

The headstart files will be provided to you as an archive file (“`headstart1.tar`”). In order to unpack the archive files, copy the `headstart1.tar` file to your project subdirectory and enter the following command:

```
bash-3.00$ tar –xvf headstart1.tar
```

The `parser.c` file contains functions to scan and parse user input. The function

```
int ParseCommandLine(char line[], struct CommandData *data)
```

takes a command line as an argument and populates a `CommandData` structure. The function returns 1 if it was able to successfully parse the line, and zero if there was some kind of error in the line. The structure of `CommandData` is as follows:

```
struct Command {
  char *command;             /* simple command name */
  char *args[11];            /* Array of up to 11 arguments */
  int numargs;               /* number of arguments */
};

struct CommandData {
  struct Command TheCommands[20];  /* the commands to be
                                      executed.  TheCommands[0] is the first command
                                      to be executed.  Its output is piped to
                                      TheCommands[1], etc. */
  int numcommands; /* the number of commands in the above array */
  char *infile;    /* the file for input redirection, NULL if none */
  char *outfile;   /* the file for output redirection, NULL if none */
  int  background; /* 0 if process is to run in foreground, 1 if in background */
};
```

In this first part you won’t need to implement the actual functionality of the shell. Starting with the headstart files, you will complete a skeleton of the shell. This shell skeleton will execute the following loop:

[2] Some of these built-in commands will be implemented in assignment 3.

1. Print a shell prompt that displays the current working directory (eg, “/home/drews/>”)
2. Read the next command line (terminates with '\n')
3. Parse the command line using the `ParseCommandLine` function which will populate the `CommandData` structure.
4. Print information about each command line: For each simple command print the command name and all the arguments. Print the filenames for input redirection or output redirection, NULL if none. Indicate whether the command line is executed with background option turned on or off. Indicate whether the command is a built-in command or not.
5. Repeat, until the user enters the *exit* command

Requirements:

1. Read the Programming Project Guidelines carefully!
2. Read the following article by D. Ritchie and K. Thompson:

   D. Ritchie and K. Thompson, "The UNIX Time-Sharing System," Communications of ACM 7, 7, July 1974. http://citeseer.ist.psu.edu/ritchie74unix.html

3. Implement the Shell Skeleton as described above.

Organization:

You may work individually or in groups with up to one other person (two total). Your program must be written in either C or C++ and compile and run correctly on the prime machines in the lab.

Submission:

You must submit your program by way of electronic submission as will be shown in class (and described on blackboard). The project number for the electronic submission is 1. Include the names of all the group members in all the source code files!

Grading:

The maximum number of points for this project is 10**.**

Hints and Help:

- A Unix Tutorial for Beginners: http://www.ee.surrey.ac.uk/Teaching/Unix/
- Unix Help for Users: http://unixhelp.ed.ac.uk/
- Useful system calls and library functions (see man pages for more info):
  `getcwd(), strncmp(), gets(), strdup()`

Example:

```
/home/drews/> ls

Number of simple commands : 1
command1    : ls
Input file  : (Null)
Output file : (Null)
Background option : OFF
Built-in command  : NO

/home/drews/> ls –l > file1

Number of simple commands : 1
command1    : ls
arg[0]      : -l
Input file  : (Null)
Output file : (file1)
Background option : OFF
Built-in command  : NO

/home/drews/> cat < infile | grep 14877 | wc

Number of simple commands : 3
command1    : cat
command2    : grep
arg[0]      : 14877
command3    : wc
Input file  :  infile
Output file :  (Null)
Background option : OFF
Built-in command  : NO

/home/drews/> cat < infile | grep file1.txt > outfile &

Number of simple commands : 2
command1    : cat
command2    : grep
arg[0]      : file1.txt
Input file  :  infile
Output file :  outfile
Background option : ON
Built-in command  : NO

/home/drews/> exit

command1    : exit
Input file  :  (Null)
Output file :  (Null)
Background option : OFF
Built-in command  : YES
```

<u>PART II:</u>

Basic Shell Functionality:

This is the second part of your shell project. You will start with the with PART I and complete a simple, but useful shell. Input to bash consists of lines using the same grammar as the first part of the assignment, with additions as listed below.

- In this part of the project you will implement the execution of *simple commands.* A simple command will be given either as an absolute path name, a relative path name, or just a command name, as in:

  ```
  /bin/ls   ./a.out   ls
  ```

  A simple command given as absolute or relative path name will be executed without further processing. If the command name is not an absolute or relative path, bash must search the directories in the `PATH` *environment* variable to find a directory containing the command and run the appropriate command, if found. For example, if the `PATH` environment variable contains:

  ```
  /bin:/usr/bin:/etc
  ```

  then bash should interpret the command "`ls`" as referring to the program "`/bin/ls`" and the command "`mount`" as referring to the program "`/etc/mount`". Note that the system call `access`() may be helpful for determining if an executable file exists with a given name.

- Your Bash shell will *redirect* the *input* and *output* of the command to the files specified on the command line, if included. By default, of course, bash should leave the `stdin`, `stdout`, and `stderr` attached to the terminal. Error checking and reporting is particularly important here.

- Your Bash shell will allow simple command pipelines consisting of two commands, such as

  ```
  ls –al | grep myfile.txt
  ```

Shell Variables and Process Environment:

- Bash assignments are of the form

  ```
  varname=value
  ```

  There can be no spaces either before or after the equals sign. ***Note that in order to support Bash assignments, it will be necessary to modify the parser code that I provided you with!***

Three of the variables, `PATH`, `HOME` and `DEBUG`, have meaning to your shell. The initial values of `PATH`, `HOME and DEBUG` should be read from the environment of the shell and any changes should immediately go back into the *shell's process environment*. If `DEBUG` is not available in your shell environment, add it with a default setting of `DEBUG=no`. Changing `PATH` MUST change the shell's search `PATH` immediately.

Finally, the built-in command

```
export varname
```

should cause the variable `varname` to become part of the environment of your shell. Any further changes to `varname` must also be propagated into the shell's environment, which is inherited by its children.

Note that there is a difference between the set of shell variables and the set of variables that are part of the Shell's process environment (will be covered in the lecture)!

Built-in Commands:

- In this assignment your Bash will support the following built-in commands:

Builtin Command	**Behavior**
`cd`	Change directory to the value of the `HOME` variable
`cd <directory>`	Change the current working directory to the value of the variable `directory`
`export <variable>`	Add the variable `variable` to the shell's process environment
`set`	Display shell variables
`pwd`	Print the value of the `HOME` variable
`exit`	Exit shell

The built-in functions are not executed by forking and executing an executable. Instead, the shell process executes them itself. All other command must be executed in a child process.

Debugging:

Shell-level debugging is enabled when the value of the variable DEBUG is "`yes`", and disabled when it is "`no`"

Example:

```
/home/drews/> DEBUG=yes
Debugging is on
/home/drews/> ls -l > file1

DEBUG OUTPUT:
--------------------------
DEBUG: Number of simple commands: 1
DEBUG: command1   : ls
DEBUG: arg[0]           : -l
DEBUG: Input file       : (Null)
DEBUG: Output file      : (file1)
DEBUG: Background option : OFF
DEBUG: Built-in command : NO
--------------------------

/home/drews/> DEBUG=no
Debugging is off
/home/drews/>
```

Note that when debugging in enabled, *you must still execute the commands in addition to printing information about them*. You may add any other code to the debugging output that you with. I only require that there be some debugging output present. The more use you make of this feature, the easier time you will have in debugging your program.

Outline of a Simple Shell with Builtin Commands:

- The following (pseudo) C-code skeleton illustrates the basic shell operation:

```
int main(void){

/* Variable declarations and definitions */

  while(1) {
          DisplayPrompt();
          ParseCommandLine(inputBuffer, CommandStructure);

          if( BuiltInTest(CommandStructure)  == 1 )
                  ExecBuiltInCommand(CommandStructure); /* execute built-in command */
          else{
          /* The steps for executing the command are: */
          (1)  Redirect Input/Output for the command, if necessary
          (2)  Create a new process which is a copy of the calling process (-> fork()
               system call)
          (3)  Child process will run a new program (-> one of the exec() system calls)
          (4) if( BackgroundOption(CommandStructure) == 0 )
                  the parent shell will wait(), otherwise it will continue the while loop
           } /* end else */
     } /* end while */
} /* end main */
```

Grading:

1. [10 points] Implementation of simple commands given as absolute command names.
2. [10 points] Implementation of simple commands given as command name using the `PATH` variable to search for the executable file.
3. [5 points] Implementation of input redirection
4. [5 points] Implementation of output redirection with/without output append.
5. [10 points] Implementation of simple command pipelines
6. [5 points] ***Correct*** implementation of Shell assignments and the `export` built-in command (which will necessitate the modification of the parser code)
7. [5 points] Other built-in commands (cd, cd pathname, pwd, exit, set)

Your will be provided with a test script that you can use to test the correctness of your shell.

Organization:

- You may work individually or in groups with up to one other person (two total). Your program must be written in either C or C++ and compile and run correctly on the prime machines in the lab.

Submission:

- You must submit your program as will be shown in class (and on blackboard). Include the names of all the group members in all the source code files!

Hints and Help:

- The man pages are your friends! Manpages of particular interest would be:

  `fork(2)`

  `exit(2)`

  `exec(2), execl(2), execv(2), execle(2), execve(2), execlp(2), execvp(2)`

  `wait(2), waitpid(2)`

  `dup2(3C)`

  `signal(3C), sigaction(2), sigprocmask(2)`

```
getcwd(3C), chdir(2)
```

For example, to pull up the manpage for dup2 on the Solaris machines, you will have to type

```
bash-3.00$ man -s 3C dup2
```

- You may find it convenient to implement the features in the following order:

  1. Run commands with absolute path names

  2. Use PATH to find commands

  3. Implement variable assignments

  4. Implement other built-in commands

  5. Implement I/O redirection