CS 744: Autumn 2019 - Programming Project 1 - Writing a UNIX Shell.

Project 1: Assigned 30th July. Due Aug 16th (11.59 pm)

Notes: The shell must be written entirely in C. The use of C++ is NOT permitted.

Part 1:

Write a small program that loops reading a line from standard input and checks the first word of the input line. If the first word is one of the following internal commands perform the designated task using only POSIX compliant UNIX system calls. Otherwise use the standard ANSI C **`system`** function to execute the line through the default system shell.

Internal Commands/Aliases:

`clr:` clear the screen.
`dir <directory>:` list the target directory contents (**`ls -al <directory>`**) - you will need to provide some command line parsing capability to extract the target directory for listing . Once you have built the replacement command line, use the **`system`** function to execute it.
`env:` list all the environment strings - the environment strings can be accessed from within a program by specifying the POSIX compliant environment list:

```
extern char **environ;
```

as a global variable. **`environ`** is an array of pointers to the environment strings terminated with a NULL pointer.
`quit:` quit from the program with a zero return value. Use the standard **`exit`** function.

For each of the above commands (except **`quit`**) print out the time it took to run the command when it is used in your shell.

External Commands:

For all other command line inputs, relay the command line to the parent shell for execution using the **`system`** function. When parsing the command line you may have to explicitly or implicitly **`malloc`** (**`strdup`**) storage for a copy of the command line. Ensure that you free any **`malloc`**ed memory after it is no longer needed. You may find **`strtok`** useful for parsing.
The C Standard Library has a number of other string related functions that you may find useful (**`string.h`**) contains links to descriptions of the other main "string" functions). As before print how long it took to run the command after the command's output.

Code should be in 'straight' C using the compiler of your choice (**`cc`** or **`gcc`**).

Part 2: Environment & Command Line Arguments

Basic information can be passed to a C process via either of two following methods:

- **Command Line Parameters**
  - `argv` - an array of character pointers (strings) 2nd argument to `main` the command line parameters include the entire parsed command line (including the program name).
  - `argc` - number of command line parameters
- **Environment**
  - `environ` - an array of character pointers (strings) in the format `"name=value"`
    the list is terminated with a `NULL` pointer.

Command Line Parameters

```
int main(int argc, char * argv[])
{
   int i;
   printf("argc = %d\n");                    // print arg count
   for (i = 0; i < argc; i++)                // print args
      printf("argv[%d]: %s\n",i, argv[i]);
   exit(0);
}
```

invoked through the command line:

```
> echoarg arg1 TEST fooo
```

will result in the following output:

```
argc = 4
argv[0]: echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: fooo
```

Accessing The Environment

While some implementations of C provide a 3rd argument to main (`char * envp[]`) with which to pass the environment, ANSI C only defines two arguments to main so the preferred way of passing the environment is by a built-in pointer (POSIX):

```
extern char **environ;  // NULL terminated array of char *
```

```
main(int argc, char *argv[])
{
   int i;
   for (i = 0; environ[i] != NULL; i++)
      printf("%s\n",environ[i]);
   exit(0);
}
```

example output:

```
GROUP=faculty
HOME=/usr/user
LOGNAME=user
PATH=/bin:/usr/bin:usr/user/bin
PWD=/usr/user/work
SHELL=/bin/tcsh
USER=user
```

Updating The Environment

You can use the functions **`getenv`**, **`putenv`**, **`setenv`** to access and/or change individual environment variables.

Internal storage is the exclusive property of the process and may be modified freely, but there is not necessarily room for new variables or larger values.

If the pointer **`environ`** is made to point to a new environment space it will be passed on to any programs subsequently invoked.

If copying the environment, you should find out how many entries are in the **`environ`** table and **`malloc`** enough space to hold that table and any extra string pointers you may want to put in it.

Remember, the environment is an array of pointers. The strings pointed to by those pointers need to exist in their own **`malloc`**ed memory space.

Set the following new environment variables using **putenv()/setenv()**:

COURSE=CS_744
ASSIGNMENT=ASSIGNMENT_1

Execute “env” to show COURSE, ASSIGNMENT and PWD (you can use **getenv()/environ**)

The Part2 Assignment: Extending Your Shell

Add the extra functionality listed below.

Internal Commands:

Add the capability to change the current directory (using **`cd <directory>`**) and set and change environment strings appropriately. Print the environment (only PWD and OLDPWD) before AND after the command.

Change the current default directory to ***`<directory>`***. If the ***`<directory>`*** argument is not present, report the **current** directory. This command should also change the ***`PWD and OLDPWD`*** environment string. For this you will need to study the **`chdir`**, **`getcwd`** and **`putenv`** functions.

While you are at it you might as well put the name of the current working directory in the shell prompt!

Be careful using **`putenv`** - the string you provide becomes part of the environment and should not be changed (or **`free`**d!). i.e. it should be made static - global or **`malloc`**ed (or **`strdup`**ed).

Part 3: Adding fork and exec and supporting background, serial and parallel execution

So far, our shell has used the system call to pass on command lines to the default system shell for execution. Since we need to control what open files and file descriptors are passed to these processes (i/o redirection), we need more control over their execution.

To do this we need to use the **`fork`** and **`exec`** system calls. **`fork`** creates a new process that is a clone of the existing one by just copying the existing one. The only thing that is different is that the new process has a new process ID and the return from the fork call is different in the two processes.

The **`exec`** system call re-initializes that process from a designated program; the program changes while the process remains!

Make sure you read the notes on **`fork`** and **`exec`** before continuing. Here is an example use of these calls:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

extern int errno;        // system error number

void syserr(char* );     // error report and abort routine

int main(int argc, char *argv[])
{
   pid_t pid;            // process ID
   int rc;               // return code

   pid = getpid();       // get our own pid
   printf("Process ID before fork: %d\n", (int)pid);

   switch (fork()) {
      case -1:
         syserr("fork");
      case 0:             // execution in child process
         pid = getpid();  // get child pid
         printf("Process ID in child after fork: %d\n", pid);
         execlp("sleepy", "sleepy", "10",NULL); // sleepy is your own
program to sleep.
         syserr("execl"); // error if return from exec
   }

// continued execution in parent process

   pid = getpid();        // re get our pid
   printf("Process ID in parent after fork: %d\n", pid);

   exit(0);
}

void syserr(char * msg)   // report error code and abort
{
   fprintf(stderr,"%s: %s", strerror(errno), msg);
   abort(errno);
}
```

In your Shell program, replace the use of **`system`** with **`fork`** and **`exec`**.

In addition, your shell must support background execution and serial and parallel execution of a set of commands specified as follows:

- If a command is followed by **`&`**, the command must be executed in the background. That is, the shell must start the execution of the command, and return to prompt the user for the next input, without waiting for the previous command to complete. The output of the command can get printed to the shell as and when it appears.
- Multiple user commands separated by **`&&`** should be executed one after the other in sequence in the foreground. The shell must move on to the next command in the sequence only after the previous one has completed (successfully, or with errors). An error in parsing one command should cause the shell to print the error message Shell: Incorrect command and move on to the next command. The shell should return to the command prompt after all the commands in the sequence have finished execution.
- Multiple commands separated by **`&&&`** should be executed in parallel in the foreground. That is, the shell should start execution of all commands simultaneously, and return to command prompt after all commands have finished execution.

You will now need to more fully parse the incoming command line so that you can set up the argument array (**`char *argv[]`** in the above examples). Note: remember to malloc/strdup and to free memory you no longer need!

Across all cases, carefully ensure that the shell reaps all its children that have terminated. For commands that must run in the foreground, the shell must wait for and reap its terminated foreground child processes before it prompts the user for the next input. For the command that creates background child processes, the shell must periodically check and reap any terminated background processes while running other commands. When the shell reaps a terminated background process at a future time, it must print a message **`Shell: Background process finished`** to let the user know that a background process has finished.

You will find that while a system function call only returns after the program has finished, the use of fork means that two processes are now running in foreground. In most cases you will not want your shell to ask for the next command until the child process has finished. This can be accomplished using the wait or waitpid functions. e.g.

```
switch (pid = fork ()) {
   case -1:
      syserr("fork");
   case 0:                  // child
      execvp (args[0], args);
```

```
      syserr("exec");
   default:                  // parent
      if (!dont_wait)
         waitpid(pid, &status, WUNTRACED);
}
```

Obviously, in the above example, if you wanted to run the child process 'in background', the flag **dont_wait** would be set and the shell would not wait for the child process to terminate.

Part 4: Adding IO Redirection

Your project shell must support i/o-redirection on both *stdin* and *stdout*. i.e. the command line:

MyShell> programname arg1 arg2 < inputfile > outputfile

will execute the program **programname** with arguments **arg1** and **arg2**, the *stdin* FILE stream replaced by **inputfile** and the *stdout* FILE stream replaced by **outputfile**.

With output redirection, if the redirection character is **>** then the **outputfile** is created if it does not exist and truncated if it does. If the redirection token is **>>** then **outputfile** is created if it does not exist and appended to if it does.

Note: you can assume that the redirection symbols, **<** , **>** & **>>** will be delimited from other command line arguments by white space - one or more spaces and/or tabs. This condition and the meanings for the redirection symbols outlined above and in the project differ from that for the standard shell.

I/O redirection is accomplished in the child process immediately after the **fork** and before the **exec** command. At this point, the child has inherited all the file handles of its parent and still has access to a copy of the parent memory. Thus, it will know if redirection is to be performed, and, if it does change the *stdin*and/or *stdout* file streams, this will only affect the child not the parent.

You can use **open** to create file descriptors for **inputfile** and/or **outputfile** and then use **dup** or **dup2** to replace either the *stdin* descriptor (**STDIN_FILENO** from **unistd.h**) or the *stdout* descriptor (**STDOUT_FILENO** from **unistd.h**).

However, the easiest way to do this is to use **freopen**. This function is one of the three functions you can use to open a standard I/O stream.

```
#include <stdio.h>
FILE *fopen(const char *pathname, const char * type);
FILE *freopen(const char * pathname, const char * type, FILE *fp);
FILE *fdopen(int filedes, const char * type);
```

All three return: file pointer if OK, NULL on error

The differences in these three functions are as follows:

1. **fopen** opens a specified file.

2. `freopen` opens a specified file on a specified stream, closing the stream first if it is already open. This function is typically used to open a specified file as one of the predefined streams, `stdin`, `stdout`, or `stderr`.
3. `fdopen` takes an existing file descriptor (obtained from `open`, etc) and associates a standard I/O stream with that descriptor - useful for associating pipes etc with an I/O stream.

The `type` string is the standard open argument:

`type`	Description
`r` or `rb`	open for reading
`w` or `wb`	truncate to 0 length or create for writing
`a` or `ab`	append; open for writing at end of file, or create for writing
`r+` or `r+b` or `rb+`	open for reading and writing
`w+` or `w+b` or `wb+`	truncate to 0 length or create for reading and writing
`a+` or `a+b` or `ab+`	open or create for reading and writing at end of file

where `b` as part of `type` allows the standard I/O system to differentiate between a text file and a binary file.
Thus:

```
freopen("inputfile", "r", stdin);
```

would open the file `inputfile` and use it to replace the standard input stream, `stdin`.
You may want to use the `access` function to check on existence or not of the files:

```
#include <unistd.h>
int access(const char *pathname, int mode);
                                        // Returns: 0 if OK, -1 on error
```

The `mode` is the bitwise OR of any of the constants below:

`mode`	Description
`R_OK`	test for read permission
`W_OK`	test for write permission
`X_OK`	test for execute permission
`F_OK`	test for existence of file

Looking at the project specification, `stdout` redirection should also be possible for the internal commands: `dir`, `env`.

Submission Instructions:

Submit a single tarball (<rollnumber>.tar.gz) consisting of a directory (named your rollnumber):

a. A Makefile - we will simply run this makefile and if it does not run or sources do not compile you will not get any credit at all.
b. A set of C sources (.c only) - you must have at least one file **`myshell.c`** which contains **`main.`**
c. A set of C headers (.h only)
d. A README that explains clearly how to build and run your shell program. This will also serve as user documentation for the shell.

Grading Rubric:

- Submission in proper format and with correct naming: 5
- Compiles with NO WARNINGS - 10
- Part 1: 25 (5x4 for clr, dir, env, quit + 5 for timing information)
- Part 2: 15 (7 for cd, 4 for updating the env, 4 for changing shell prompt)
- Part 3: 25 (10 for fork, exec, 5 each for background, sequential and parallel)
- Part 4: 20 for IO redirection (10 for output redirection and 10 for input redirection)

Test-cases:

NOTE: These are just sample testcases to help you understand better and may not be included in the test-cases used during grading.

PART 1: (Print time taken to execute cmd after showing output)

MyShell> dir /home/diptyaroop/Documents

```
total 7404
drwxr-xr-x  2 diptyaroop diptyaroop    4096 Jul 29 15:43  .
drwxr-xr-x 57 diptyaroop diptyaroop    4096 Jul 31 07:56  ..
-rw-rw-r--  1 diptyaroop diptyaroop  104473 Jul 11 20:36  183050016-1.pdf
-rw-rw-r--  1 diptyaroop diptyaroop 1621389 Jul 29 15:43
183050016_Diptyaroop_Maji_Seminar_Report.pdf
-rw-rw-r--  1 diptyaroop diptyaroop  476847 Feb 14 11:08
60829a85d1c579a401be9642d12bb097.jpg
-rw-rw-r--  1 diptyaroop diptyaroop 4756761 Jul 17 11:24  DEPT_Handbook_19-20.pdf
-rw-r--r--  1 diptyaroop diptyaroop  282000 Apr 20 12:20  test.xcf
-rw-rw-r--  1 diptyaroop diptyaroop  277336 Apr  3 11:04 'tmem-based cache implementation
__and partitioning policies for cgroup end-points.odp'
```

-rw-rw-r-- 1 diptyaroop diptyaroop   35516 Jul 17 11:29 'Untitled spreadsheet - Dipty.pdf'
Time taken to execute command = …

(equivalent to ls -al)
(Can use opendir() etc. as Sir mentioned on Moodle, or system())
(<directory> will be absolute path. So, no need to worry about relative paths)

MyShell> env

CLUTTER_IM_MODULE=xim
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
LESSCLOSE=/usr/bin/lesspipe %s %s
XDG
_MENU_PREFIX=gnome-
LANG=en_IN
GDM_LANG=en_US
MANAGERPID=1923
DISPLAY=:0
INVOCATION_ID=8a3262e617a241dfa6a82a1dbd0679b0
GNOME_SHELL_SESSION_MODE=ubuntu
COLORTERM=truecolor
XDG_VTNR=7
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
S_COLORS=auto
XDG_SESSION_ID=c2
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/diptyaroop
USER=diptyaroop

DESKTOP_SESSION=ubuntu
QT4_IM_MODULE=xim
TEXTDOMAINDIR=/usr/share/locale/
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/472b0136_cc67_4dce_98ee_d22219a03a97
DEFAULTS_PATH=/usr/share/gconf/ubuntu.default.path
PWD=/home/diptyaroop/Documents
HOME=/home/diptyaroop
JOURNAL_STREAM=9:37286
TEXTDOMAIN=im-config
SSH_AGENT_PID=2077
QT_ACCESSIBILITY=1
LIBVIRT_DEFAULT_URI=qemu:///system
XDG_SESSION_TYPE=x11
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/local/share:/usr/share:/var/lib/snapd/desktop
XDG_SESSION_DESKTOP=ubuntu
DBUS_STARTER_ADDRESS=unix:path=/run/user/1000/bus,guid=beaf43242e2593920d43ad075d40fc61
GTK_MODULES=gail:atk-bridge
TERM=xterm-256color
SHELL=/bin/bash
VTE_VERSION=5202
XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
QT_IM_MODULE=ibus
XMODIFIERS=@im=ibus
IM_CONFIG_PHASE=2
DBUS_STARTER_BUS_TYPE=session
XDG_CURRENT_DESKTOP=ubuntu:GNOME
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
GNOME_TERMINAL_SERVICE=:1.85
XDG_SEAT=seat0
SHLVL=1
LANGUAGE=en_IN:en
GDMSESSION=ubuntu
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LOGNAME=diptyaroop
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus,guid=beaf43242e2593920d43ad075d40fc61
XDG_RUNTIME_DIR=/run/user/1000
XAUTHORITY=/home/diptyaroop/.Xauthority
XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg

PATH=/home/diptyaroop/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
SESSION_MANAGER=local/diptyaroop-GS65:@/tmp/.ICE-unix/1944,unix/diptyaroop-GS65:/tmp/.ICE-unix/1944
LESSOPEN=| /usr/bin/lesspipe %s
GTK_IM_MODULE=ibus
_=/usr/bin/env
OLDPWD=/home/diptyaroop
Time taken to execute command = …

PART 2:

(Show the env variables PWD and OLDPWD before and after directory has changed. Print them in the format shown below, before your shell prompt comes)
(cd <directory> --> "directory" will be a relative path, so you have to get current path first)

(/home/diptyaroop) MyShell> cd Documents
BEFORE:
OLDPWD = <whatever the earlier directory was, or empty, if OLDPWD value was not set)
PWD = /home/diptyaroop/
AFTER:
OLDPWD = /home/diptyaroop
PWD = /home/diptyaroop/Documents
(/home/diptyaroop/Documents) MyShell>

(/home/diptyaroop/Documents) MyShell> cd ..
BEFORE: ...
AFTER: ...
(/home/diptyaroop/) MyShell>

(/home/diptyaroop/) MyShell> cd
BEFORE: ...
AFTER: ...
(/home/diptyaroop/) MyShell>
(reporting current directory)

(/home/diptyaroop/) MyShell> cd Random
ERROR: Directory does not exist
BEFORE: ...
AFTER: ...
(/home/diptyaroop/) MyShell>
(If <directory> does ont exist, it will show an error message starting with ERROR:)

(/home/diptyaroop/) MyShell> env
COURSE=CS_744
ASSIGNMENT=ASSIGNMENT_1
PWD=/home/diptyaroop

PART 3:

(You are expected to support at least these commands:
ls, pwd, echo, sleep, cd, wc, cat
Although, once you get the hang of it, supporting other commands are not so difficult.)

(NOTE: Final testcases won't have mix of serial/parallel/background processes. So, you don't have to worry about them.
Example:
echo a && echo b &&& echo c
echo a && sleep 5 &
Sleep 5 &&& echo ab && ls & etc.
These type of commands WON'T be given)

(/home/diptyaroop/) MyShell> echo abc
abc

(/home/diptyaroop/) MyShell> pwd
/home/diptyaroop/

(/home/diptyaroop/) MyShell> sleep 5 &
[1] 6613
(/home/diptyaroop/) MyShell>
(You can continue using your shell, but if you press enter or run any command after 5 secs, it should also show the following message:
MyShell: Background process [6613] finished.
Example:

(/home/diptyaroop/) MyShell> sleep 5 &
[1] 6756
(/home/diptyaroop/) MyShell> pwd
/home/diptyaroop/
(/home/diptyaroop/) MyShell> echo r
r
MyShell: Background process [6756] finished.
(/home/diptyaroop/) MyShell>
)

(/home/diptyaroop/) MyShell> echo a && pwd
a
/home/diptyaroop/Documents

(/home/diptyaroop/) MyShell> sleep 2 && echo ab
ab
(ab should be displayed after 2 secs)

(/home/diptyaroop/) MyShell> echo ab && sleep 2
ab
(Command prompt should come 2 secs after displaying ab)

(/home/ankush/) MyShell> sleep 5 &&& sleep 2 &&& echo ab
ab
(Display ab instantly and shell should return to command prompt after 5 secs.)

PART 4:

(Output redirection)
(/home/ankush/) MyShell> echo Hello > file1
(Create a file with a file name file1 if there is no such file)
(/home/ankush/) MyShell> cat file1
Hello
(/home/ankush/) MyShell> echo Hey > file1
(If file exists, '>' overwrites the file)
(/home/ankush/) MyShell> cat file1
Hey
(/home/ankush/) MyShell> echo Hello >> file1
(/home/ankush/) MyShell> cat file1
Hey
Hello
(If file exists, '>>' appends to the file, else, file is created)

(Input redirection)
(/home/ankush/) MyShell> wc -l < file1
2
(print number of lines in file file1 created above)