CS 3113 Project 1B

Roll Your Own Shell (75 points)

TL;DR

For this project, you should extend part 1A by:

1. Adding changes to the environment though the `cd` command.
2. Replace the `system` call with `fork` - `exec`.
3. Add I/O redirection to the command line.

Grading Criteria

Task	Percent
Code compiles with `make all`	10%
Code compiles with `make clean`	10%
The `cd` commands works and alters environment	20%
`fork` - `exec` implemented for commands	20%
Input/Output and redirectors work	40%
Total	**100%**

If the instance is not reachable you will receive zero credit.

You will submit your external ip address for grading. And a compressed folder of your make file and source code with the name `project1b_<4x4>.tar.gz` where `<4x4>` should be replaced with your OU 4x4. The compressed folder should include your makefile and any source and header files.

How to get data from the instance

https://cloud.google.com/compute/docs/instances/transfer-files

Updating The Environment

You can use the functions `getenv`, `putenv`, `setenv` to access and/or change individual environment variables. (See below for more information). 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.

Extending your shell

Previously, you wrote a simple shell that looped reading a line from standard input and checked the first word of the input line. This week you should try adding the extra functionality listed below.

Internal Commands/Aliases:

Add the capability to change the current directory and set and change environment strings:

```
cd <directory>
```

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` environment string. For this you will need to study the `chdir` , `getcwd` , and `putenv` functions (discussed below).

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 `strduped`).

Adding `fork` and `exec` to the shell

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 there 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 reinitializes that process from a designated program; the program changes while the process remains! Make sure you read the notes on `fork` and `exec` below and the text.

Your TODOs

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

2. 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). N.B. remember to `malloc` / `strdup` and to `free` memory you no longer need!

3. 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 (see below for more detail). e.g.

```c
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.

4. The commenting in the above examples is minimal. In the projects you will be expected to provide more descriptive commentary!

I/O Redirection

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

`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` . For more cases on how redirection I/O redirection should be handled, please visit the textbook website: http://www.tldp.org/LDP/intro-linux/html/sect_05_01.html .

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 filehandles 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 effect 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.

```c
#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 the 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:

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

would open the file `inputfile` and use it to replace the standard input stream, `stdin`.

You should also use the `access` function to check on existence or not of the files:

```c
#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`, `environ`, `help`.

Extra Info

env processes

Each process has an environment associated with it. The environment strings are usually of the form: `name=value` (standard NULL terminated strings) and are referenced by an array of pointers to these strings. This array is made available to a process through the C Run Time library as:

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

While an application can access the environment directly through this array, some functions are available to access and manipulate the environment:

```c
#include <stdlib>

char *getenv(const char *name);

// Returns pointer to value associated with name, NULL if not found.
```

`getenv` returns a pointer to the `value` of a `name=value` string. You should use `getenv` to fetch a specific value from the environment rather than accessing `environ` directly. `getenv` is supported by both the ANSI C and POSIX standards. In addition to fetching the value of an environment variable, sometimes it is necessary to set a variable. You may want to change the value of an existing variable, or add a new variable to the environment. Unfortunately not all systems support this capability.

```c
#include <stdlib>

int putenv(const char *str);

int setenv(const char *name, const char *value, int rewrite);

//Both return: 0 if OK, non-zero on error

void unsetenv(const char *name);
```

- `putenv` takes a string of the form `name=value` and places it in the environment list. If the name already exists, its old definition is first removed.

- `setenv` sets `name` to `value`. If `name` already exists, its old definition is first removed if `rewrite` is non-zero. Otherwise the value is not overwritten.
- `unsetenv` removes any definition of `name`.

You may need to use `putenv` with the environment value left blank to unset an environment object. i.e. `putenv("myname=")`

chdir

```c
#include <unistd.h>

int chdir(const char * pathname);

// Returns: 0 if OK, -1 on error
```

Every process has a current working directory. This directory is where the search for all relative pathnames starts (all pathnames that do not begin with a slash). When a user logs on to a UNIX system, the current working directory normally starts at the directory specified in the `/etc/passwd` file - the user's "home" directory. The current working directory is an attribute of a process; the home directory is an attribute of a login name. You can change the current working directory of the calling process by calling the `chdir` function.

`pathname` can be absolute or relative and can start with (or be) the relative strings "`.`" and "`..`".

Enter the `man chdir` command on your system to get a more detailed description of this function.

Note: The `chdir` function does NOT change the `PWD` environment variable.

getcwd

```c
#include <unistd.h>

char *getcwd(char *buf, size_t size);

//Returns: buf if OK, NULL on error
```

Every process has a current working directory which can be set using `chdir`. While `chdir` can use a relative pathname argument, there is a need for a function to derive the absolute pathname of the directory. `getcwd` performs this function.

The function is passed the address of a buffer, `buf`, and its `size`. The buffer must be large enough to

accommodate the full absolute pathname plus a terminating `NULL` byte, or an error is returned.

Some implementations of `getcwd` allow the first argument `buf` to be `NULL`. In which case the function calls malloc to allocate size number of bytes dynamically. This is not part of the POSIX standard and should be avoided.

Enter the `man getcwd` command on your system to get a more detailed description of this function.

fork

Process creation in UNIX is achieved by means of the kernel system call, `fork()`. When a process issues a `fork` request, the operating system performs the following functions (in kernel mode):

1. It allocates a slot in the process table for the new process
2. It assigns a unique process ID to the child process
3. It makes a copy of the parent's process control block
4. It makes a copy of the process image of the parent (with the exception of any shared memory)
5. It increments counters for any files owned by the parent to reflect that an additional process now also owns those files
6. It assigns the child process to a Ready to Run state
7. It returns the ID number of the child to the parent process and a 0 value to the child process - this function is called once and returns twice!

```c
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);

//Returns: 0 in child, process ID of child in parent, -1 on error
```

The `fork` system call creates a new process that is essentially a clone of the existing one. The child is a complete copy of the parent. For example, the child gets a copy of the parent's data space, heap and stack. Note that this is a copy. The parent and child do not share these portions of memory. The child also inherits all the open file handles (and streams) of the parent with the same current file offsets.

The parent and child processes are essentially identical except that the new process has a new process ID and the return value from the `fork` call is different in the two processes:

- The "parent" process gets the new process ID of the "child" returned from the fork call. If, for some reason the process can not be cloned, then -1 is returned

- The "child" process is returned 0 (zero) from the fork call.

exec

To actually load and execute a different process, the `fork` request is used first to generate the new process. The kernel system call: `exec(char* programfilename)` is then used to load the new program image over the `fork`ed process:

- `exec` identifies the required memory allocation for the new program and alters the memory allocation of the process to accommodate it
- The program is loaded into memory and execution is commenced at the start of the `main()` routine.

The `exec` system call reinitializes a process from a designated program; the program changes while the process remains! The `exec` call does not change the process ID and process control block (apart from memory allocation and current execution point); the process inherits all the file handles etc. that were currently open before the call.

Without `fork`, `exec` is of limited use; without `exec`, `fork` is of limited use (A favorite exam questions is to ask in what circumstances you would/could use these functions on their own. Think about it and be prepared to discuss these scenarios).

`exec` variants:

System Call	Argument Format	Environment Passing	PATH search
ececl	list	auto	no
execv	array	auto	no
execle	list	manual	no
execve	array	manual	no
execlp	list	auto	yes
execvp	array	auto	yes