OPERATING SYSTEMS

CS-370/Fall-2022

Operating Systems Assignment – 1

The UNIX Shell

This assignment consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. Your implementation will support input and output redirection, as well as pipes as a means of Inter process communication (IPC) between a pair of commands. Completing this project will involve using the UNIX fork ( ), exec (), wait ( ), dupe (), and pipe ( ) system calls and can be completed on any Linux, UNIX, or macOS system.

Overview

A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt osh> and the user's next command: cat prog.c. (This command displays the file prog.c in the terminal.)

osh›cat prog.c

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, cat prog . c) and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. However, UNIX shells typically also allow the child process to run in the background, or concurrently. To accomplish that, we add an ampersand (&) at the end of the command.

Thus, if we rewrite the above command as:

osh>cat prog.c &

the parent and child processes will run concurrently. The separate child process is created using the fork () system call, and the user's command is executed using one of the system calls in the exec ( ) family. For the purpose of this assignment, we will only be using execvp () system call.

The main ( ) function in the file called simple-shell.c presents the prompt osh> and outlines the steps to be taken after input from the user has been read. To recreate a shell, you must modify the contents of the main function such that it caters to different user inputs as specified in the scope of assignments tasks. Furthermore, like any normal shell, **you are to ensure that the program terminates if the user enters 'exit'.**

The assignment is divided into the following parts that need to be implemented:

1. Creating a child process and executing commands in the child
2. Executing multiple commands using &
3. Providing a history feature by remembering the past N commands
4. Inter-process communication using pipes and input-output redirection

Part – I

The first task is to modify the main() function so that a child process is forked and executes the command specified by the user.

This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings. A token represents a space separated group of characters.

For example, if the user enters the command

osh>ps -ael

at the osh> prompt, the values stored in the args array are:

1. args [0] = "ps"
2. args [1] = "-ael"
3. args [2] = NULL

This args array will be passed to the execvp() function, which has the following prototype:

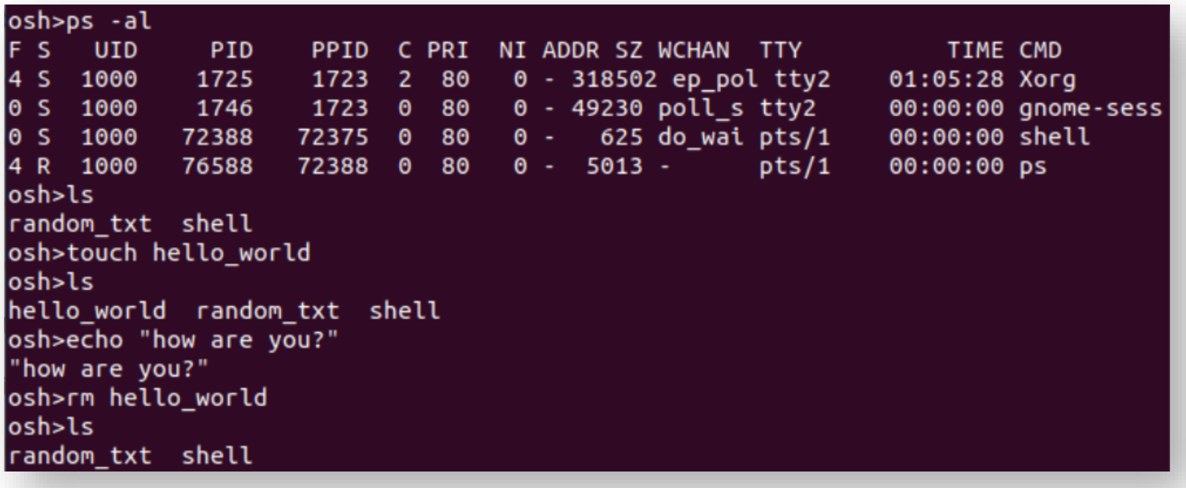
*execvp (char *command, char *params []).*

Here, command represents the command to be performed and params stores the parameters to this command.

For this project, the execvp (function should be invoked as execvp(args [0], args).

For a single command, you can assume that the maximum number of arguments would be 40. Furthermore, it would be a helpful pointer to observe that each argument is space separated.

Some examples of sample outputs are attached below:



```
osh>ps -al
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000    1725    1723  2  80   0 - 318502 ep_pol tty2    01:05:28 Xorg
0 S  1000    1746    1723  0  80   0 - 49230 poll_s tty2    00:00:00 gnome-sess
0 S  1000   72388   72375  0  80   0 -   625 do_wai pts/1    00:00:00 shell
4 R  1000   76588   72388  0  80   0 -  5013 -      pts/1    00:00:00 ps
osh>ls
random_txt  shell
osh>touch hello_world
osh>ls
hello_world  random_txt  shell
osh>echo "how are you?"
"how are you?"
osh>rm hello_world
osh>ls
random_txt  shell
```

Part – II

The second task is to allow multiple commands to be executed in one go. This is generally an easier approach for commands to be executed so that they can be completed parallel to each other.

This is achieved by using the logical AND operator (&&).

```
osh> cat simple-shell.c && ls && cat test.c
```

In this case, instead of running cat simple-shell.c and then waiting for it to finish, your shell should run cat simple-shell.c, ls, and cat test.c(each with whatever arguments the user has passed to it) in parallel, *before* waiting for any of them to complete.

Please be mindful that there can be multiple AND operators, and hence, there can be multiple commands that can be executed one after the other.

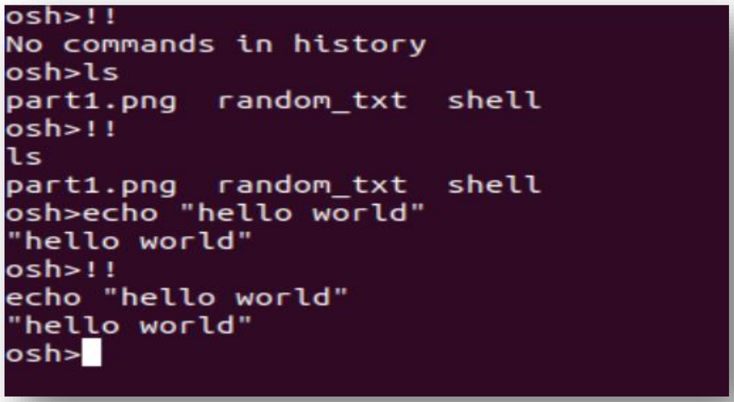
Part – III

The third task is intended to mimic the history feature of the UNIX shell and additionally optimize its operation.

When you generally open the shell and type the command '!!', it will show either of the two outputs:

1. It will **show** the command that was last executed and execute the same command.
2. If the user just opened the shell and did not execute any other command, then, the shell should just display to the user that there is no command found.

An example of this is shown below:



```
osh>!!
No commands in history
osh>ls
part1.png   random_txt   shell
osh>!!
ls
part1.png   random_txt   shell
osh>echo "hello world"
"hello world"
osh>!!
echo "hello world"
"hello world"
osh>
```

Furthermore, to build more functionality over this specification, we are to customize the commands that the user can go and access back from the history.

To enable this, we will provide the user with the keyword 'hist' that is followed by the number of previous commands that we wish to go.

There will be two clearly defined distinctions that you are to enable.

1. If the user types in 'hist' at the osh terminal without any arguments, then, the terminal should **display** the history of commands in chronological order with the last command executed being shown first.
2. Else if the user types in 'hist' in the osh terminal with one argument that specifies the order of the last executed command, then, the terminal should execute the appropriate command that befits the order mentioned. Furthermore, in case that there is no command found at the order specified, the terminal should display to the user that there is no command found.
   For instance, 'hist -3' would mean that we are to execute the third last command. However, in a case where the user only executed 2 commands in total, the terminal would display that there is no command found.

Below are some implementation specifications that you are expected to abide by in order to be provided credit for this entire part:

You are to initialize a dynamically allocated circular singly linked list using the structure of a node as defined in the *shell.c* file provided to you which will serve as a history buffer. This history buffer will store the last executed commands so that the user is able to jump back to any last executed command.

Hence, in order to simply execute the last command or execute any previous command using the 'hist' keyword, you are to make use of the dynamically allocated circular singly linked list to be able to gain credit for your implementation.

Part – IV

The fourth task can be divided into 2 phases. However, please note that the tasks should be logically devised in consideration of one another so that the implementation is easier. ☺

Phase – a

In the first phase, you will be introduced to inter-process communication using pipes. The final modification to your shell is to allow the output of one command to serve as input to another using a pipe. For example, the following command sequence

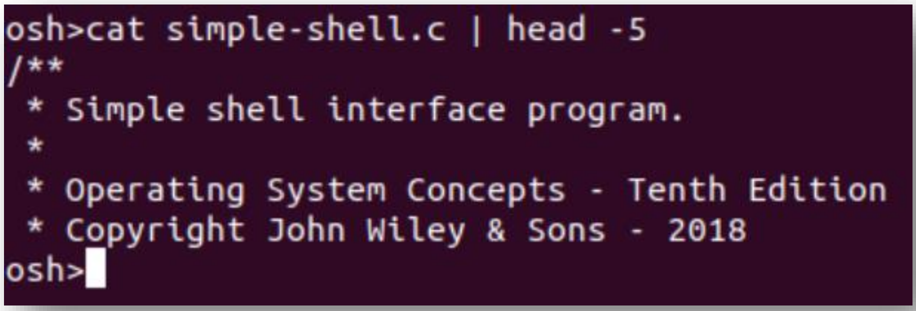
osh>ls -l | less

has the output of the command ls -l serve as the input to the less command. Both the ls and less commands will run as separate processes and will communicate using the UNIX pipe () function. Implementing the pipe functionality will also require usage of dup2() command which is used for the redirection of output to another process or a file. For instance, dup2() can be used to write the output of a command to a separate file instead of the terminal.

It duplicates an existing file descriptor to another file descriptor. For example, if fd is a file descriptor to the file out.txt, the call dup2(fd, STDOUT_FILENO); duplicates fd to standard output (the terminal). This means that any writes to standard output will in fact be sent to the out.txt file.

You can assume that the user will only be using one pipe command for this phase.

An example of this is shown below:



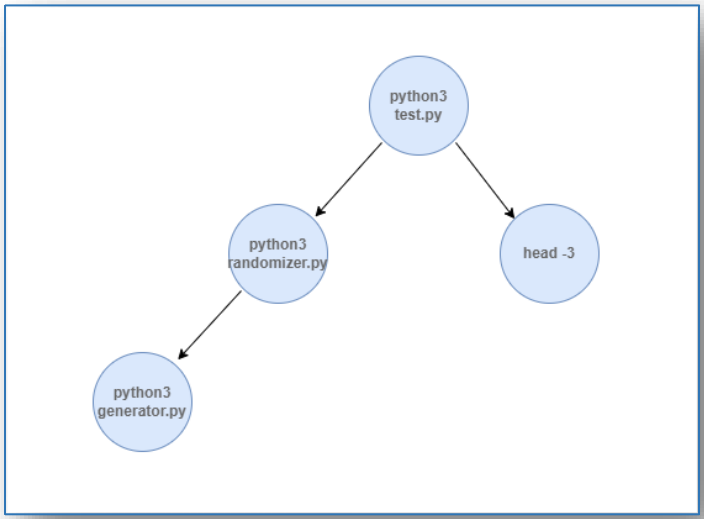
```
osh>cat simple-shell.c | head -5
/**
 * Simple shell interface program.
 *
 * Operating System Concepts - Tenth Edition
 * Copyright John Wiley & Sons - 2018
osh>
```

Phase – b

For this phase, you will be building upon the functionality derived from earlier phase. Primarily, you will be implementing a command tree that would model complex relationships between processes and execute them in a specific order.

A command tree can be simply put as a binary tree with commands (along with their arguments) stored as an array of characters in the tree nodes.

An example of command tree is shown below:

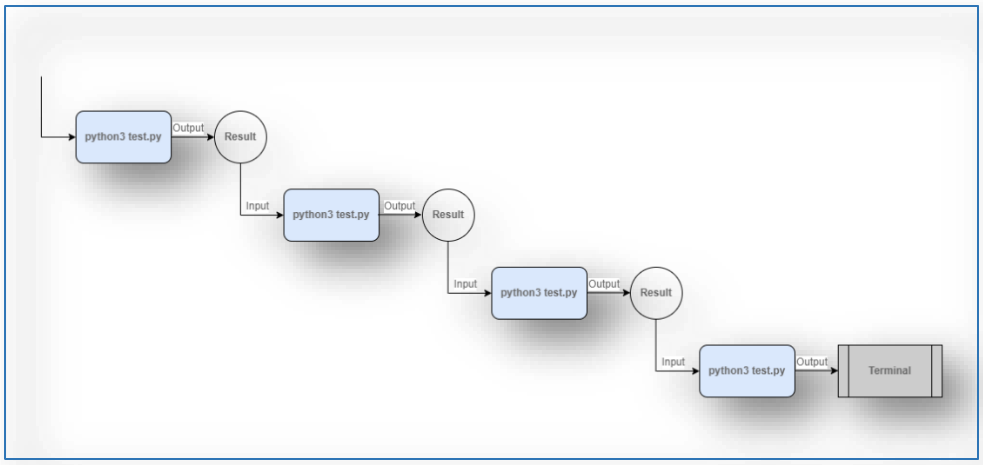


In-order traversal of the above shown tree would be as follows:

python3 generator.py→ python3 randomizer.py → python3 test.py → head -3

The primary distinction between a command tree and a normal sequential chaining using the && operator is that the output of the first executed command will be fed as the input to the next command in order using pipes in the former approach.

For instance, for the above shown tree, the in-order command tree chaining would follow a pattern of execution as shown below:



Tree definition

In order to define trees using strings that can be entered via the terminal, the format is as specified below:

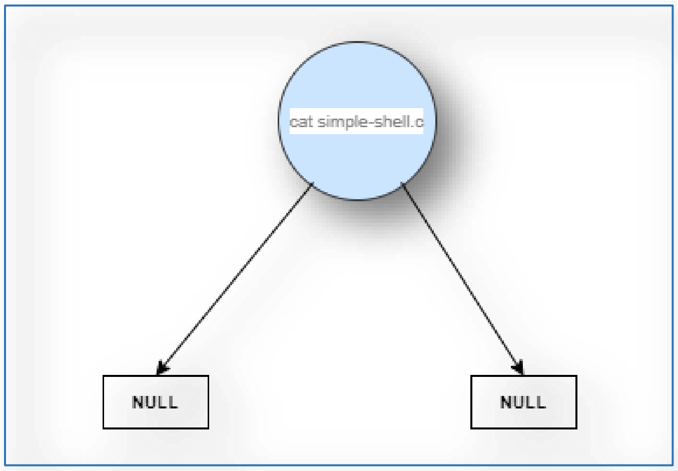
Tree (root_data,left_tree,right_tree)

Trees are recursively defined using this approach and a NULL child of a node is indicated using an empty parenthesis, ().The Tree keyword should be detected by your program as an indication that the user is attempting to make a command tree for execution.

Example – 1

```
osh> Tree (cat simple-shell.c,(),())
```

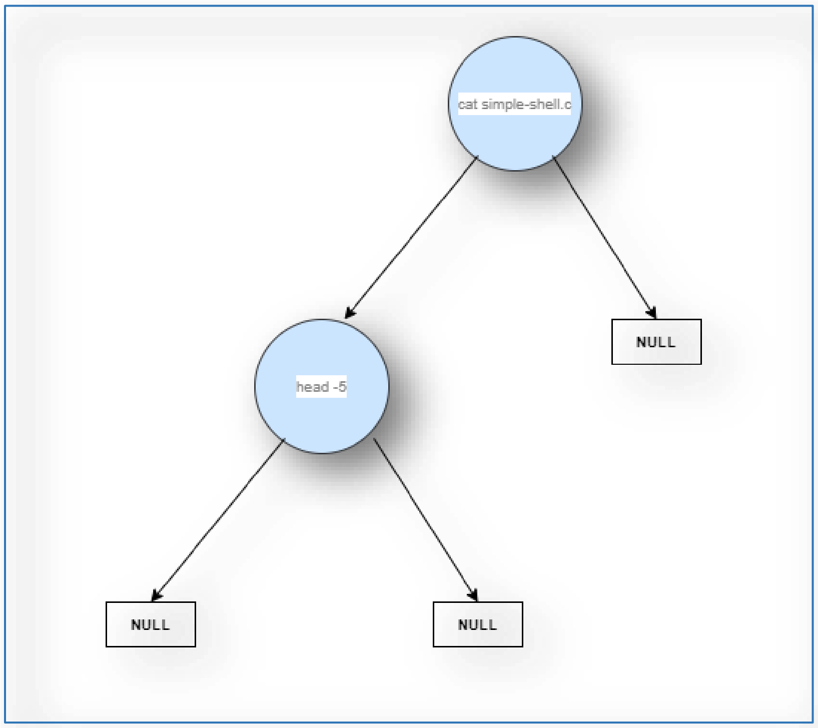
This would create a tree as following:



Example – 2

```
osh> Tree (cat simple-shell.c,(head -5,(),()),())
```

This would create a tree as following:



Implementing trees

To make a tree, you are provided with a function tree_maker(char* args) that has the specification as follows:

tree_maker (char* args)

- **Arguments** – Takes a pointer to character array that stores the complete tree.
- **Return Type** – Returns a pointer to the root of the tree.

Example – 1

- Args array - (cat simple-shell.c,(head -5,(),()),())
- Return – A pointer to a root of the tree as shown in Example- 2 of tree definition

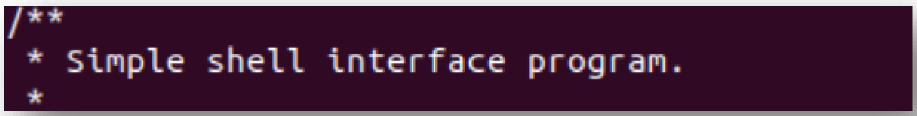
Task details

Your task is to make trees using the command line and the function tree_maker (char* args) provided. Furthermore, with the tree returned, you are to execute the commands in the sequence of the tree's in-order traversal such that the output of the commands are fed as input to the next command in sequence.

For instance, if the in-order traversal of a tree is as follows:

cat simple-shell.txt → head -6 → head -3

The output should be as follows:



```
/**
 * Simple shell interface program.
 *
```

Theoretically, this would mean that cat simple-shell.txt would be executed first by a child process and its output would be propagated to another child process that would execute head -6 over the output of cat simple-shell.txt. Following this, the output of head -6 would be propagated to another child process that would execute head -3 over the output of head -6.

This communication will occur via multiple pipes and file descriptors.

Instructions

- The skeleton code has been provided to you in the file simple-shell.c.
- While submitting, please zip the file and name the zip as your PairNumber_PA1.zip. For example, 001_PA1.zip. The pair number can be accessed from the spreadsheet containing pairing preferences that was shared earlier.
- Do not plagiarize. Any cases of plagiarism will be forwarded to the disciplinary committee
- The rigid deadline for this assignment is 2nd October, 2022. Make sure to start early to avoid any last-minute inconveniences