# OPERATING SYSTEMS – ASSIGNMENT 2

## Writing Your Own Shell

**Submission Date: 21.4.25**

- יש לכתוב תכנית המכילה ארבע סוגי קבצים (קובץ הגדרות - קובץ עם סיומת h., קובץ המכיל כל גופי פונקציות, קובץ ראשי – main, קובץ readme – הסבר על הפעלת תכנית). העבודה תמומש בשפת תכנות C בסביבת לינוקס.
- שם הקובץ שיוגש למערכת ההגשה יהיה מורכב מת"ז של המגיש/ים. לדוגמה:

עבור הגשה ביחיד - 111111111.zip

עבור הגשה בזוג - 111111111_22222222.zip

- שאלות לגבי העבודה יש לשאול בפורום באתר הקורס ("מודל") או בשעות קבלה של **גנאדי קוגן**.
- אין לשלוח שאלות הקשורות לעבודה במייל.
- **עליכם לקרוא היטב את הנהלים להגשת עבודות בית שפרסמו במודל.**

## ***Introduction:***

The object of this assignment is to gain experience with some advanced programming techniques such as process creation, termination, and overriding an existing process (**fork**, **wait**, **exec**, and additional system calls). To do this, you will be writing your own command shell - much like bash.

## ***What is Shell?***

A shell program is an application that allows interacting with the OS. In a shell, the user can run programs and also redirect the input and output. Shell additionally offers features such as line editing, history, file completion, etc.

***"A shell is an interface that allows you to interact with the kernel of an operating system."***

## **Shell Workflow:**

A shell parses commands entered by the user and executed them. The workflow of the shell will look like this:

1. Startup the shell
2. Wait for user input
3. Parse user input
4. Execute the command and return the result
5. Go back to 2

The shell is the parent process. This is the main process in our program which is waiting for user input. However, we cannot execute the command in the parent process itself, because of the following reasons:

1. A command with error will cause the shell to stop working. We want to avoid this.
2. Independent commands should have their own process blocks. This is known as the isolation principle.

## ***Shell pseudo-code*** **(22 points)**

```
int main (int argc, char **argv)
{
 int childPid;
 char * cmdLine;
 parseInfo *info;
 while(1){
      cmdLine= readline(">");
      info = parse(cmdLine);

      childPid = fork();

      if (childPid == 0)
      {
         /* child code */
         executeCommand(info); //calls  execvp
      }
      else
      {
         /* parent code */
         waitpid(childPid);
      }
  }
 }
```

**Remarks**:

1. You must follow the structure of the pseudo-code.
2. Variable 'info' - is the result of parsing 'cmdLine' into tokens, i.e., command to run and its parameters

## ***Required Features:***

Here are the features you should support in your shell:

1) **pwd:**

   The shell should indicate the current working directory using **pwd**.

2) **cd:**

   **cd** <path> should change the working directory.

3) **nano:**

   Create a file by the command **cat >** <filename> or **nano** <filename>.

4) **cat:**

   **cat** <filename>. View the contents of the file.

5) **chmod:**

   The chmod command is used to change the access permissions or modes of files and directories. The permission modes represent who can read, write, or execute the file.

   For example:

   chmod 755 file.txt

   There are three sets of permissions—owner, group, and public. Permissions are set using numeric modes from 0 to 7:

   - 7 – read, write, and execute.
   - 6 – read and write.
   - 4 – read only.
   - 0 – no permission.

   This sets the owner permissions to 7 (rwx), group to 5 (r-x), and public to 5 (r-x). You can also reference users and groups symbolically:

chmod g+w file.txt

The g+w syntax adds group write permission to the file.

Setting proper file and directory permissions is crucial for Linux security and controlling access. chmod gives you flexible control to configure permissions precisely as needed.

6) **cp:**

**cp** <file1> <file2>. Copy the file <file1> to file <file2>.

7) **pipe:**

The shell has to support **pipe** operator "|" between processes (the pipe operator needs to be executed only for two commands).

8) **clear:**

**clear** should clear the terminal.

9) **grep and grep -c:**

grep Return lines with a specific word/string/pattern in a file. grep [options/flags] [Pattern] <filename>. Should support in the following options(flags): grep -c [pattern] <filename>.

10) **ls, ls –l > output_file:**

Command lists directory contents of files and directories. **ls -l** show long listing information about the file/directory.

The command "ls -l > output file" will redirect the output of the "ls -l" command to a file named "output file" in the current directory.

The ">" symbol is used for output redirection, which means that the output of the command on the left side of

the symbol is sent to the file on the right side of the symbol, overwriting any existing content in the file.

Therefore, after running the command "ls -l > output file", a new file named "output file" will be created in the current directory (if it does not exist already) containing the output of the "ls -l" command. If the file already exists, its previous content will be replaced by the new output.

11) **tree:**

**tree** print all files. It is used to display the contents of any desired directory of your computer system in the form of a tree structure. By using this tree structure, you can easily find out the paths of your files and directories.

Note that, unlike the previous sections, you are required to build the command yourself. Write C program that prints out the contents of a directory and its subdirectories in a tree-like format. The **print_tree** function takes a path and a level as input (**the level parameter in the print_tree function is an integer that represents the depth or level of recursion in the directory tree. It is used to determine the amount of indentation to apply when printing the directory and file names**). It first opens the directory using the opendir function and checks if it was successfully opened. If not, it returns.
It then iterates over each entry in the directory using the readdir function provided by 'dirent.h'. For each entry, it checks if it is a directory or a file. If it is a directory, it creates a new subpath by concatenating the original path with the directory name. It then recursively calls print_tree with the subpath and an incremented level.
If the entry is a file, it simply prints out the file name with the appropriate indentation based on the level.

Finally, the function closes the directory using the closedir function.
The program recursively prints out the contents of all subdirectories in the same tree-like format, with increasing indentation for each level of subdirectory.

Finally, from your main program (shell) you need to execute the **tree** command using execvp. The execvp function compile and run a separate tree.c file using gcc.
The size of the character array is chosen to be 1024, which is a common size for character arrays used to store file paths in systems. This size is sufficient to store most file paths, but it may not be large enough to store very long file paths. In this case, the size of the character array is set to 1024, which should be sufficient for most paths encountered in practice.

12) **exit:**

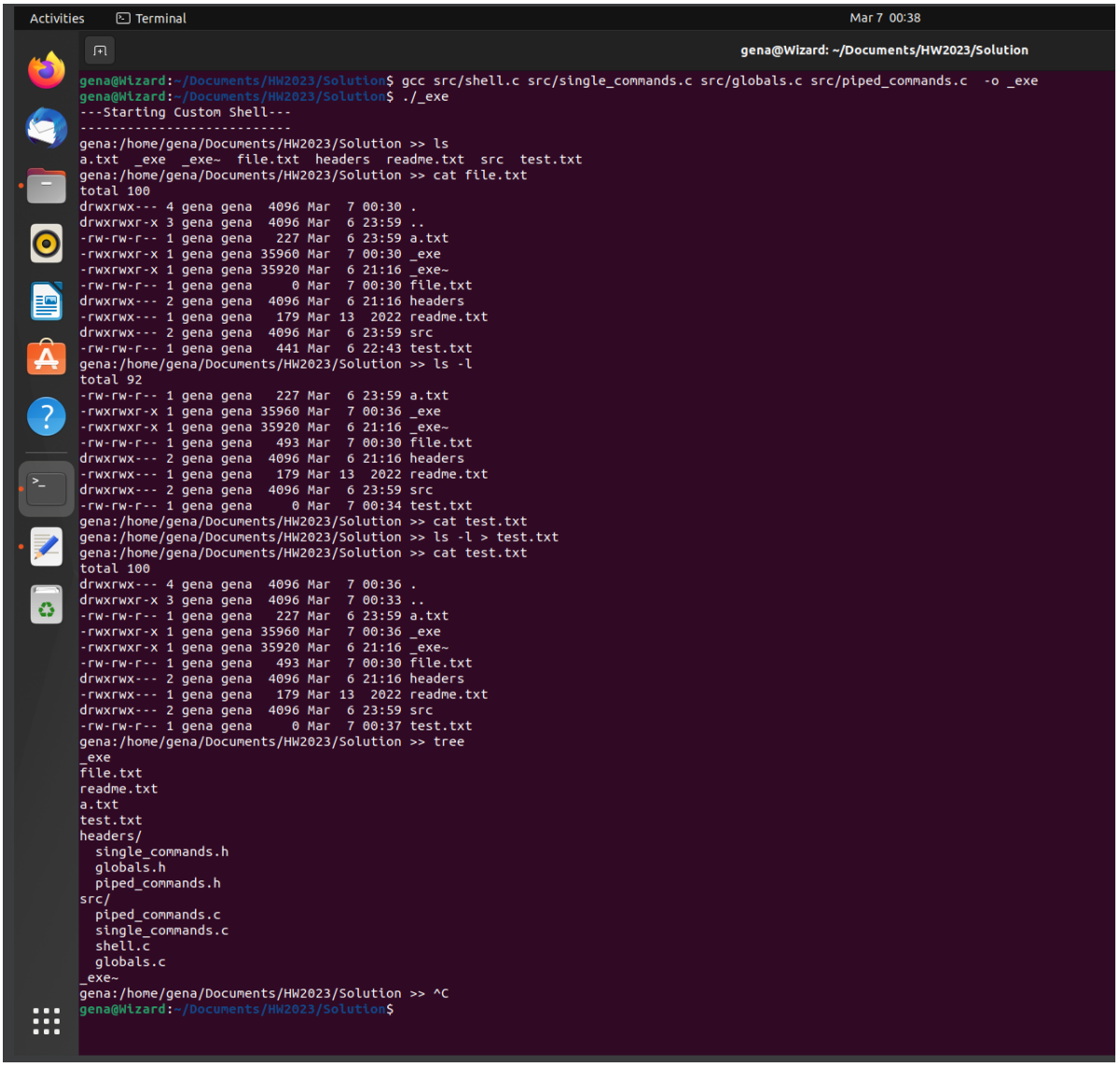
exit should terminate your shell process.

13) **rmdir** – Remove Directory:

To delete an empty directory, use the rmdir command. Note that rmdir can only remove empty directories – we'll need the rm command to delete non-empty ones.

- Each command must check whether the parameters are correct, and print an appropriate error message (much like a real shell does).

**Example:**




## ***Helpful Resources:***

Here is a list of resources you may find helpful.

**Linux/UNIX**

The following functions are some helpful functions (consult the man pages for details):

fork, exec, execvp, wait, waitpid, dup, pipe, strncmp, strlen, malloc, free, getcwd, chdir, dup2, dup, open, close, readline, gets, fgets, getchar, 'dirent.h', opendir, readdir, closedir.

# Enjoy!