CSCI 2500 — Computer Organization
Lab 05 (document version 1.0)

- This lab is due by the end of your lab session on Wednesday, September 27, 2023.
- This lab is to be completed **individually**. Do not share your code with anyone else.
- You **must** show your code and your solutions to a TA or mentor and answer their questions to receive credit for each checkpoint.
- Labs are available on Tuesdays before your lab session. Plan to start each lab early and ask questions during office hours, on the Discussion Forum on Submitty, and during your lab session.

1. **Checkpoint 1:** On your *nix system, make sure you have access to the following commands: `find`, `grep`, and `sed`.

   (a) Clone or download Submitty source code from the GitHub repository (https://github.com/Submitty/Submitty) and unpack it to a local directory.

   (b) Use a single `find` command with a regular expression to find all files (but not directories!) in the Submitty repository written in C++(`.c`, `.cc`, `.cpp`, `cxx`, `c++`, `h`, and `hxx` files), PHP (`.php` files), and Python (`.py` files) whose names are between 4 and 7 alphabet characters (i.e., a to z, case insensitive) optionally followed by one or two digits. Output the names of all matching files with paths.

   (c) Use a single `grep` command with a regular expression to find all Python files (`.py`) in the Submitty repository that use list comprehension with `if` (e.g., `[ line_ for line_ in line_list if len(line_) > 0 ]`). Output the names of all matching files with the path as well as each matching line, four lines preceding the matching line and four lines following it.

   (d) Use `find` and `sed` as a single command (i.e., use `-exec` flag of `find` to run `sed`) with a regular expression to change `str.format()` method of formatting strings in all Python files (`.py`) in the Submitty repository with newer f-Strings. For example,
   `"Missing the yaml file {}".format(file_path)`
   should be replaced with
   `f"Missing the yaml file {file_path}"`
   Remember that Python may use either single quotes or double quotes to delimit string literals.

   For simplicity, you may make the following assumptions:

   - Each Python statement is written on a single line
   - There is only one argument in `str.format()`
   - The expression inside `{}` in the original code is always empty
   - There will be no varargs in the argument of `str.format()` (e.g., `"PGPASSWORD='{}' host={}".format(*variables)`

   Remember that your replacements should only be made on applicable lines of code. Other code should not be affected. After performing replacement all code should remain syntactically valid and correct.

2. **Checkpoint 2:** Write a C program that acts as a simple shell. It prints a prompt, just like any standard shell, and then repeatedly takes one line of input from `stdin` and executes it as a command. Your shell must implement the features listed below:

   - Each command is the name of a built-in command or an executable (possibly with path) followed by command line arguments that depend on the specific command. Some commands might not have any arguments.
   - At least the following built-in commands must be supported:
     - `time` prints the current time.
     - `date` prints the current date.
     - `echo` prints the argument(s).
     - `history` prints all commands executed by your shell in the current session (i.e., in the current run of the shell). Note that history should be saved in heap, not a file.
     - `exit` terminates your shell.
   - Executable name is separated from the command line arguments by at least one space. Command line arguments are separated from each other by one or more spaces.
   - The executable may reside in any of the directories listed in the `PATH` environment variable.
   - You are not allowed to run or otherwise use any *nix shell (bash, zsh, etc.) Your implementation must be 100% pure C language with standard C and Unix libraries. Of course, you are allowed to use `fork()` and `exec()` family of functions.

   To simplify your implementation, you may make the following assumptions:

   - Spaces are only used to separate the executable name from the command line arguments and one argument from another argument. In other words, executable path/name or any command line argument may not contain spaces.
   - There will be no redirection (`>`, `<`) or pipes (—) in any of the commands.

   You also need to thoroughly test your shell to demonstrate to your lab TA/mentor that your code is correct. Given the time constraints of the lab period, you have to automate testing by writing some test input files and then running them with your shell by using a script. Here is some starting code idea but you need to develop it further:

```
#!/bin/bash
declare -a input_arr=("input01.txt" "input02.txt")
declare -a output_arr=("output01.txt" "output02.txt")

make clean
if [ $? -eq 0 ]
then
  echo "Successfully cleaned the project."
else
  echo "Cleaning the project failed. Terminating the script." >&2
  exit 1
fi

make all
if [ $? -eq 0 ]
then
  echo "Successfully built the project."
else
  echo "Building the project failed. Terminating the script." >&2
  exit 1
fi

echo "******************************************************************************"

## now loop through the array of inputs
```

```
for input_idx in "${!input_arr[@]}"
do

  if [ "$1" = "valgrind" ]
  then
    echo valgrind --leak-check=full --show-leak-kinds=all .\/kshell \< "${input_arr[$input_idx]}" 2\>\&1 \| tee "${output_arr[$input_idx]}"
    valgrind --leak-check=full --show-leak-kinds=all ./kshell < "${input_arr[$input_idx]}" 2>&1 | tee "${output_arr[$input_idx]}"
  else
    echo .\/kshell \< "${input_arr[$input_idx]}" 2\>\&1 \| tee "${output_arr[$input_idx]}"
    ./kshell < "${input_arr[$input_idx]}" 2>&1 | tee "${output_arr[$input_idx]}"
  fi
  if [ $? -eq 0 ];
  then
    echo "Successfully ran the project with input ${input_arr[$input_idx]} and captured output in ${output_arr[$input_idx]}."
  elif [ $? -eq 137 ];
  then
    echo "Running the project with input ${input_arr[$input_idx]} failed due to out of memory condition." >&2
  else
    echo "Running the project with input ${input_arr[$input_idx]} failed." >&2
  fi

  echo "******************************************************************************"
done
```

Note that only `stdout` and `stderr` are redirected to the output file, so it will not contain anything from `stdin`. If you run your shell in the interactive mode and manually enter all commands using your keyboard then they will show up on your screen and you would be able to copy the entire contents of your terminal which will include everything from all three standard streams.

To get the full checkpoint, you must also show your lab TA/mentor the output of `valgrind` or other similar tool proving that your code has no memory leaks. Try entering invalid commands in your shell and observe the output of `valgrind`. Explain the reported memory leaks. Why are they only shown when entering invalid commands?