Istanbul Technical University
BLG 312E Operating Systems
Homework 1

Sadettin Fidan

April 10, 2025

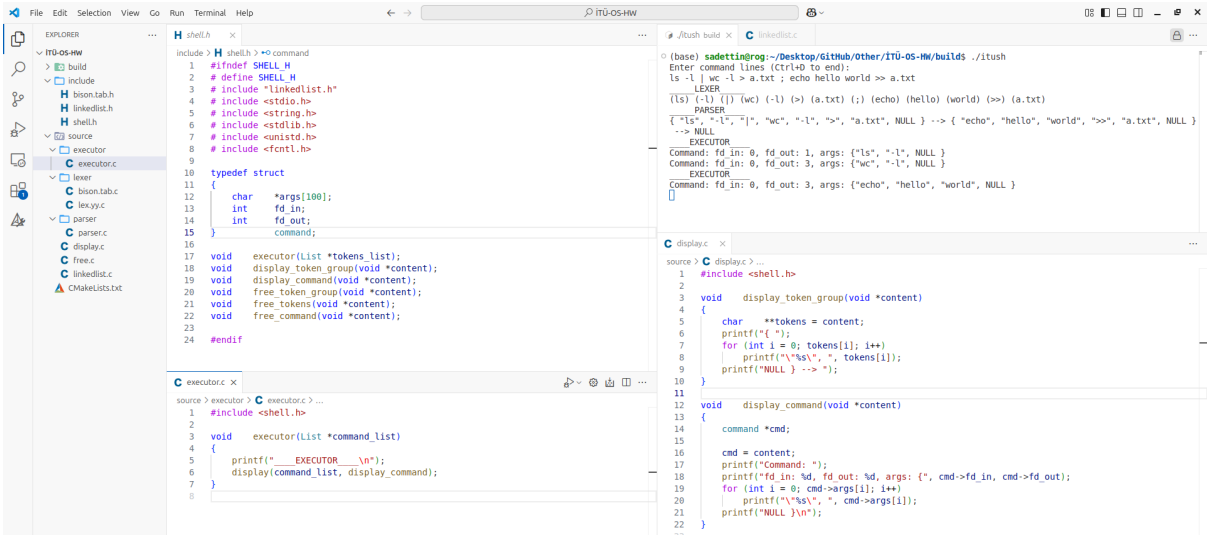
1 Preamble

2 Introduction

In this section, you will get familiarized with the homework structure.

2.1 Subject

In the screenshot below provided, you could see the output of the program, which is a shell program. We expect you to complete execution part of this custom linux shell which is currently only printing what it has.



In a shell, the pipeline is composed of lexer, parser, expander, executor respectively

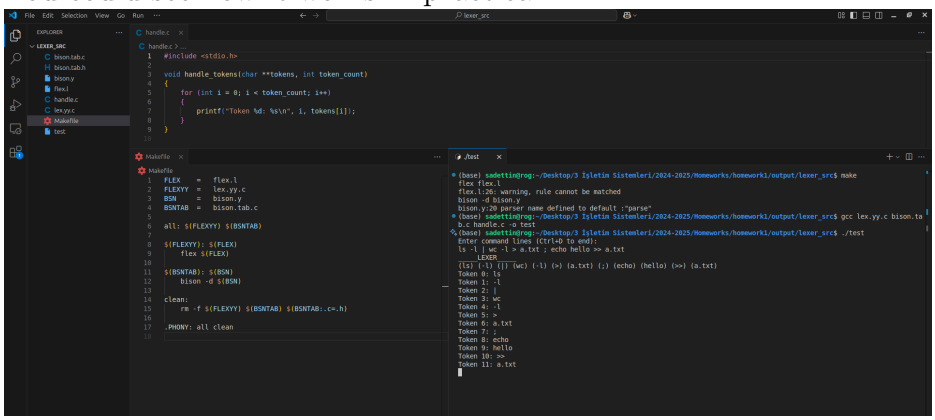
- Lexer is a string tokenizer component of a Linux shell that separates text into string tokens.
- Parser is about converting the string tokens into more suitable data structure that will be used during the execution. This structure includes input file descriptor, output file descriptor, arguments, etc.
- Expander is about wildcard expansion, removing of quotations, interpretation of dollar signs, etc. This part is not required for this homework.
- Executor is the final element in the pipeline. It creates a child process, sets the input and output of each child process, stops the parent process for child processes during their execution. Please see "executor.c" file provided as an executor example for "ls -l | wc -l > a.txt" command input.

2.2 Materials

1. **Lexer Source Code:** The provided lexer source code demonstrates a basic implementation using Flex and Bison tools to generate C sources for shell parsing.

   - **Flex (Lexical Analyzer):** Generates a scanner that recognizes lexical patterns in text, including:
     - Commands
     - Arguments
     - Operators (|, >, <, &&, etc.)
     - Variables
   - **Bison (Parser Generator):** Creates a parser that analyzes text structure according to grammar rules, handling:
     - Command syntax definition
     - Operator precedence
     - Abstract syntax tree construction

   You could see how it works in practical:



2. **Template Codes:**

```
.
├── CMakeLists.txt
├── include
│   ├── bison.tab.h
│   ├── linkedlist.h
│   └── shell.h
└── source
    ├── check.c
    ├── display.c
    ├── executor
    │   └── executor.c
    ├── free.c
    ├── lexer
    │   ├── bison.tab.c
    │   └── lex.yy.c
    ├── linkedlist.c
    └── parser
        └── parser.c

6 directories, 12 files
```

- include: a folder that holds header files.
- source: a folder that holds C source files.
- lexer is handled by lex.yy.c, bison.tab.c, bison.tab.h.
- parser is handled by parser.c
- executor, the final step, is handled by executor.c

2.3 Tools

1. **Makefile:** It is a tool to automize compilation process of a project commonly coded in C/C++. Usage: "make <command>".
2. **cmake:** It creates Makefile without complex setup.
   To run the CMakeLists.txt:
   - Create a folder named "build" next to CMakeLists.txt using "mkdir" command.
   - Go into the folder using "cd" command.
   - Run "cmake ..", then you will see Makefile is created.
   - Run "make", then you will see the executable "itush".
3. **Dockerfile:** It creates a lightweight virtual environment allowing many different kind of operating systems. A Dockerfile will be provided for you.

3 Requirements

- **Exit status:** Your shell should store the exit status for each command in its "struct command". Exit status should be get from "waitpid" function. Generally 0 is a child program's exit status for proper exit, and the value is different according to whether it is the command itself, argument, etc. causing the inproper exit.
- **Leak check:** When your itush is run with "valgrind −−leak-check=full ./itush", no leaks should be seen.
- **Pipe implementation:** You should utilize pipes which are communication channels between the two consecutive child processes. Also you need to explain what is pipe doing as a comment.
- **Fork implementation:** When "cat | cat | ls" is executed under itush parent process, you should first see the output of ls, then the terminal is waiting for input. And when you hit the enter button twice, all the child processes should exit. As you succesfully handle this situation, you will need to explain how this happens (hint: SIGPIPE).
- **File descriptors:** You should close all file descriptors stored with fd_in and fd_out after each execution. If these file descriptors are somehow closed automatically, you should explain how.

4 Rules

- You should put comment for each line of code you have coded.
- The project should work with Dockerfile given.
- You should implement your executor in a single file.
- Memory leaks should be avoided, and the project should be compiled with new "executor.c".

5 Allowed Libraries

1. stdio.h
2. stdlib.h
3. unistd.h
4. fcntl.h
5. string.h
6. sys/wait.h

6 Functions

6.1 Must know

1. open, close: "man 2 open", "man 2 close".
2. pipe: "man 2 pipe".
3. fork: "man 2 fork".
4. perror: "man 3 perror".
5. printf: "man 3 printf"
6. exit: "man exit"
7. dup, dup2: "man 2 dup", "man 2 dup2".
8. execvp: "man 3 execvp".
9. waitpid: "man 2 waitpid".

6.2 Others

- sysconf: "man 3 sysconf"
- fcntl: "man 2 fcntl"
- malloc, free: "man 3 malloc", "man 3 free".
- strcmp, strdup: "man 3 strcmp", "man 3 strdup".