KURUKULASOORIYA KVR **200332X**

CS 3480 PROGRAMMING LAB REPORT

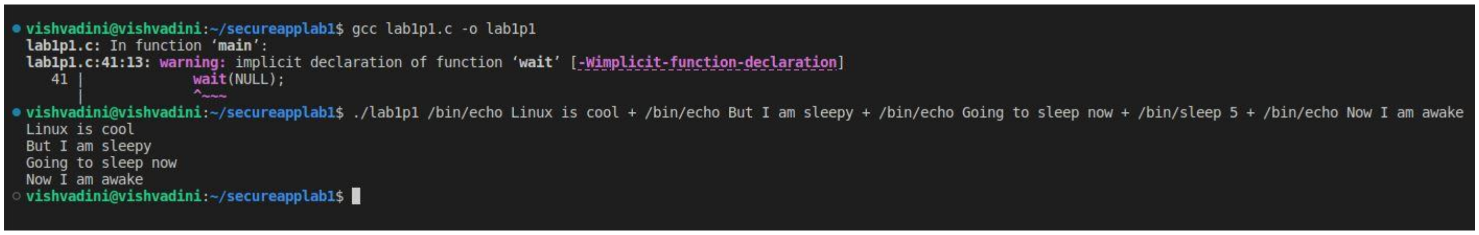
This programming lab assignment was targeted on getting a first-hand experience using the fork and exec family of system calls. The fork() system call is used to create a copy of the running process itself resulting in 2 independent processes running the same code but with separate memory spaces. The exec() system is used to load and run a new program, replacing the current program's code & memory with that.

This program was supposed to be written using C language and was tested in a Linux-based environment installed in a VMWare. Please refer to the README.txt for the instructions on building & execution of the code & lab1p1.c and lab1p2.c for the source code.

How the Task was Approached

Part I

In Part I of the lab, the requirement was to **make 5 processes run that are given in one command line input.** It was instructed to first take the split input string of separate 5 processes and spawn child processes one at a time using fork() and then execute it using exec() and continue the loop for the rest of the processes.

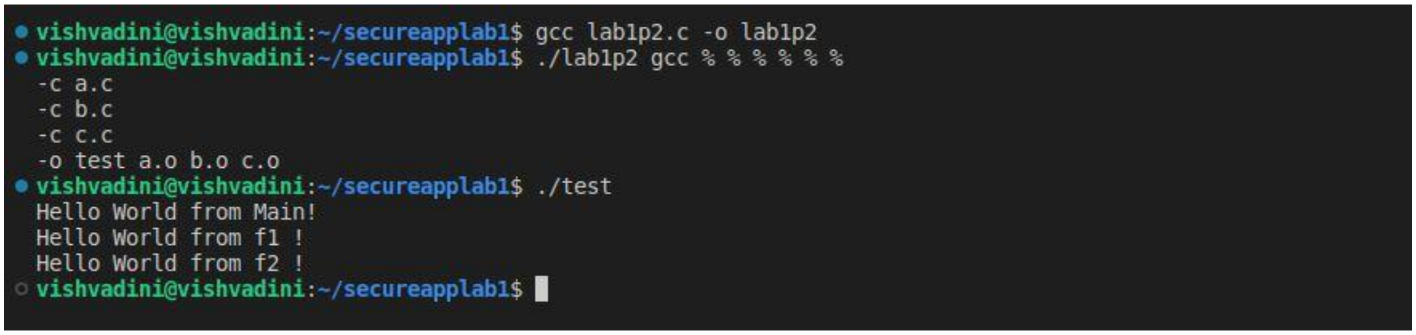


```
● vishvadini@vishvadini:~/secureapplab1$ gcc lab1p1.c -o lab1p1
  lab1p1.c: In function ‘main’:
  lab1p1.c:41:13: warning: implicit declaration of function ‘wait’ [-Wimplicit-function-declaration]
     41 |             wait(NULL);
        |             ^~~~
● vishvadini@vishvadini:~/secureapplab1$ ./lab1p1 /bin/echo Linux is cool + /bin/echo But I am sleepy + /bin/echo Going to sleep now + /bin/sleep 5 + /bin/echo Now I am awake
  Linux is cool
  But I am sleepy
  Going to sleep now
  Now I am awake
○ vishvadini@vishvadini:~/secureapplab1$ ▌
```

Please refer to Figure I given in this document for the flow chart representation of the Part I implementation.

Part II

In Part II of the lab, the requirement was **to first build the 3 .c files given as the command line input and then link them to a single object file and verify the result.** Thus, the input was stored in an array and separately processed to build them and then link all 3 object files to create a single object file using fork & exec system calls. Then the newly created file's correctness was verified.



```
● vishvadini@vishvadini:~/secureapplab1$ gcc lab1p2.c -o lab1p2
● vishvadini@vishvadini:~/secureapplab1$ ./lab1p2 gcc % % % % % %
  -c a.c
  -c b.c
  -c c.c
  -o test a.o b.o c.o
● vishvadini@vishvadini:~/secureapplab1$ ./test
  Hello World from Main!
  Hello World from f1 !
  Hello World from f2 !
○ vishvadini@vishvadini:~/secureapplab1$ █
```

Please refer to Figure II given in this document for the flow chart representation of the Part II implementation.

Challenges Faced & How they were Resolved

1. **Handling strings in C language was difficult.**

It was very challenging to handle C strings that was taken from the inputs when trying to map them with pointers and append them to other chr arrays. It could be solved by referring to documents and video tutorials that explain how to use them in a correct manner.

2. **Keep getting linker error when trying to link a.o, b.o & c.o files to a single test file**

It was totally due to incorrectly referencing the a.o, b.o & c.o files, thus the linker was unable to find such files for the linking process. This was solved by rearranging the array structures that were used to store input arguments and the gcc command.

3. **Implementation of the Part II tasks was a bit difficult to understand.**

It was a bit unclear to figure out how the output of Part II task could be checked and verified for its correctness. This issue was discussed among some of the colleagues and arrived at the conclusion on give a name to the linking file as "**-o test a.o b.o c.o**" and then running it separately using "**./test**".

4. **Unawareness of C library functions on selecting the most suitable function for the requirement.**

Since the awareness of how to use C library functions to concatenate strings/ memory mapping/ array handling was at a low level, it was challenging to implement some of the tasks on processing the inputs. This issue could be tackled to some extent by referring to online resources.

Figure I : Flow Chart for Part I implementation

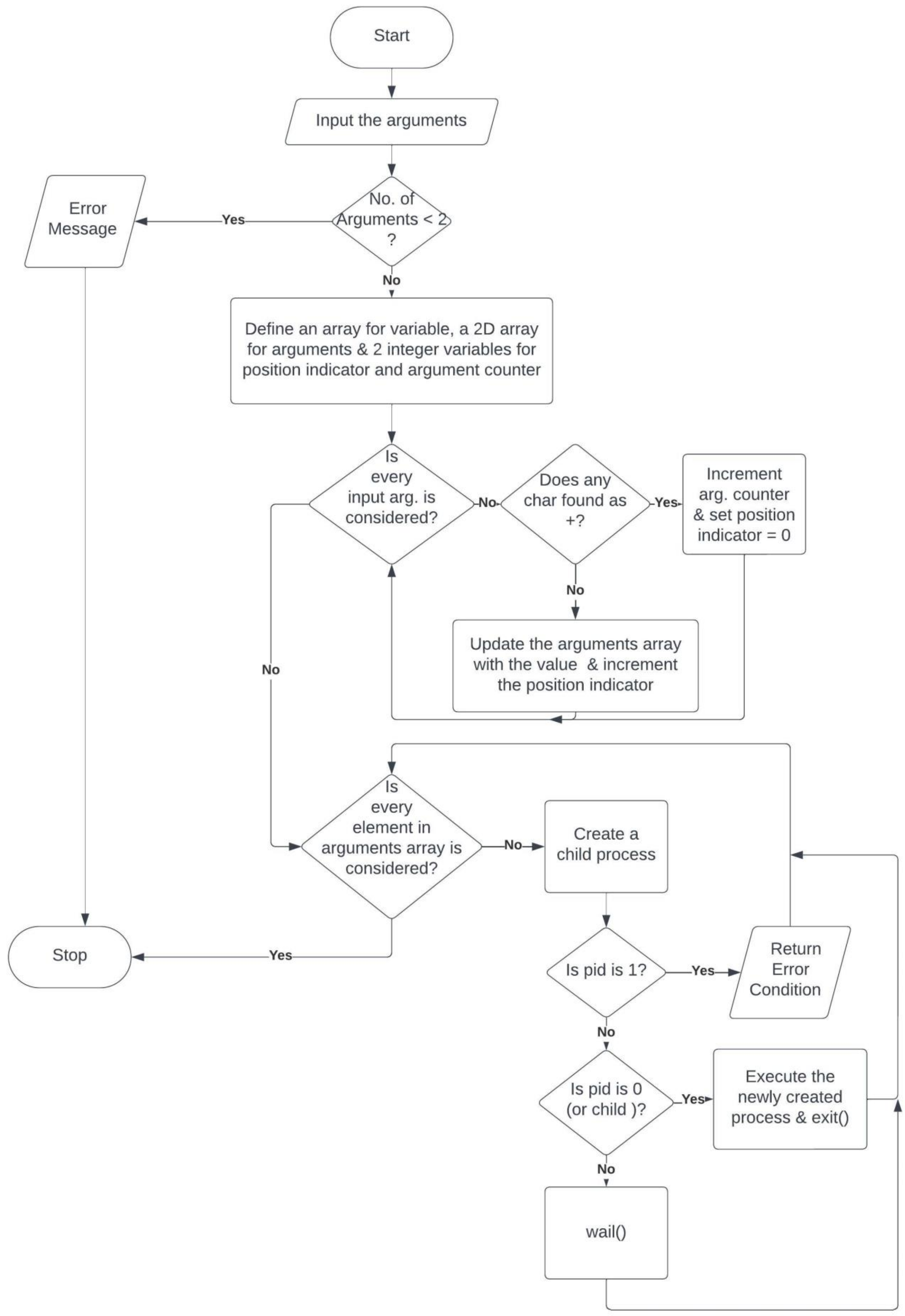
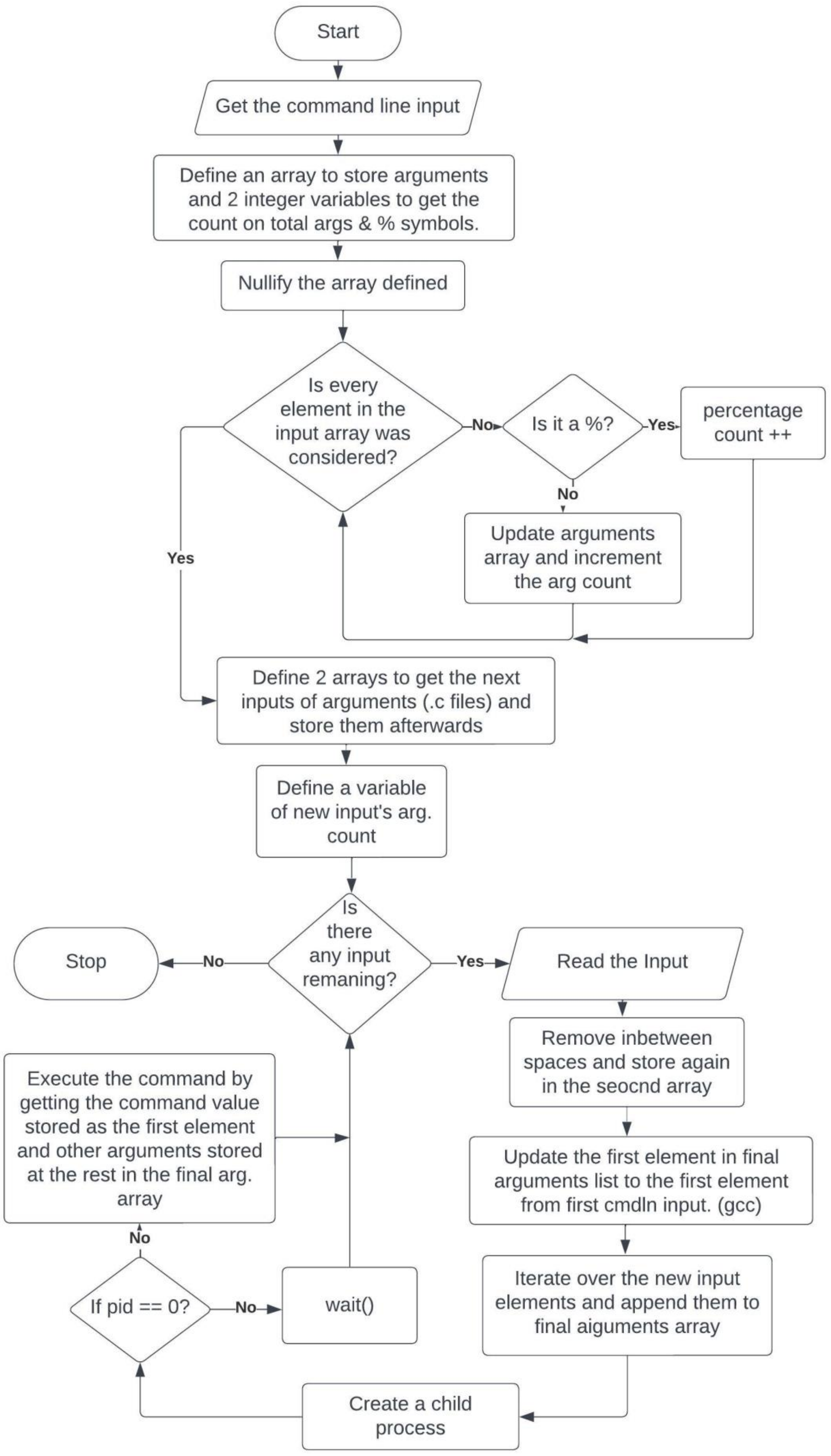

Start
Input the arguments
No. of Arguments < 2 ?
Yes
Error Message
No
Define an array for variable, a 2D array for arguments & 2 integer variables for position indicator and argument counter
Is every input arg. is considered?
No
Does any char found as +?
Yes
Increment arg. counter & set position indicator = 0
No
Update the arguments array with the value & increment the position indicator
No
Is every element in arguments array is considered?
No
Create a child process
Yes
Stop
Is pid is 1?
Yes
Return Error Condition
No
Is pid is 0 (or child)?
Yes
Execute the newly created process & exit()
No
wail()

Figure II : Flow Chart for Part II implementation



Start
Get the command line input
Define an array to store arguments and 2 integer variables to get the count on total args & % symbols.
Nullify the array defined
Is every element in the input array was considered?
No
Is it a %?
Yes
percentage count ++
No
Update arguments array and increment the arg count
Yes
Define 2 arrays to get the next inputs of arguments (.c files) and store them afterwards
Define a variable of new input's arg. count
Is there any input remaning?
No
Stop
Yes
Read the Input
Remove inbetween spaces and store again in the seocnd array
Update the first element in final arguments list to the first element from first cmdln input. (gcc)
Iterate over the new input elements and append them to final aiguments array
Create a child process
If pid == 0?
No
wait()
No
Execute the command by getting the command value stored as the first element and other arguments stored at the rest in the final arg. array