**Lab\_Tutorial**

You need to have 7 files in this project:

1. Makefile
2. Header.h
3. myshell.c
4. main\_cat.c (show the content of file)
5. main\_echo.c (write the input string into the file)
6. main\_echo0.c (write the input string on the screen)
7. main\_printf.c (write an arbitrary string on the screen)

There are eleven steps/tasks in this lab and follow them below to finish the lab.

**Step1: Create lab directory**

Create a folder named “lab” and create 7 empty files with the following names (the names mentioned above):

Makefile, header.h, myshell.c, main\_cat.c, main\_echo.c, main\_echo0.c, main\_printf.c

**Step2: Makefile**

Copy and paste the following content to file “Makefile” that you just created it:

SRCS = main\_printf.c main\_echo0.c main\_echo.c main\_cat.c myshell.c   
OBJS = $(SRCS:.c=.o)  
CFLAGS = -g -I.  
  
printf: $(OBJS)  
 $(CC) main\_$@.o myshell.o   
 ./a.out  
  
echo0: $(OBJS)  
 $(CC) main\_$@.o myshell.o -o my\_echo  
 ./my\_echo Alice  
  
echo: $(OBJS)  
 $(CC) main\_$@.o myshell.o -o my\_echo  
 -rm file1 file2  
 ./my\_echo Alice @ file1  
 ./my\_echo Bob @@ file1  
 cat file1  
 ./my\_echo Charlie @ file1  
 cat file1  
 ./my\_echo David @@ file2  
 cat file2  
  
cat: $(OBJS)  
 $(CC) main\_$@.o myshell.o -o my\_cat  
 ./my\_cat file1  
  
clean:  
 rm \*.o \*.out

You may need to understand this “Makefile” which is explained below. This is required for finishing step 7. But you can skip to the step 3 now.

Each line of “Makefile” is described here.

* SRCS = main\_printf.c main\_echo0.c main\_echo.c main\_cat.c myshell.c

The variable SRCS defines the C source files (It is mentioned in step 1 that we have 5 C files in the folder “lab”.).

* OBJS = $(SRCS:.c=.o)

The variable OBJS defines the C object files (It replaces all files in SRCS with extension “.c” to extension “.o”).

So OBJS = main\_printf.o main\_echo0.o main\_echo.o main\_cat.o myshell.o.

* CFLAGS = -g –I

This variable includes two GCC flags. (“-g” for debugging and “-I” for header file).

* printf: $(OBJS)  
   $(CC) main\_$@.o myshell.o   
   ./a.out

First line: It introduces the target “printf” and puts “:” after it. Pay attention that after “:”, we see the list of files which target “printf” is dependent on. So here “printf” is dependent on all object files in OBJS. If target “printf” is executed, it will compile all the dependent object files.

Second line: The “CC” stands for “GCC”. Symbol “@” means the name of target which is “printf” here. So “[main\_$@.o](mailto:main_$@.o)” equals to “main\_printf.o”. As you remember, for compiling object file, we use “gcc filename.o”, and this line means

“gcc main\_printf.o myshell.o”. (These two files are the object files of main\_printf.c and myshell.c)

Third line: As you know, “./a.out” is the command to execute the files. (Note that there are tabs in the beginning of lines two and three)

* echo0: $(OBJS)  
   $(CC) main\_$@.o myshell.o -o my\_echo  
   ./my\_echo Alice

First line: It introduces the target “echo0” and puts “:” after it. Pay attention that after “:”, we see the list of files which command “echo0” is dependent on them. So here “echo0” is dependent on all object files in OBJS. If any of the object files changes, the “echo0” should be executed again.

Second line: The “CC” stands for “GCC”. Symbol “@” means the name of tag which is “echo0” here. So “[main\_$@.o](mailto:main_$@.o)” equals to “main\_echo0.o”. As you remember, for compiling object file, we use “gcc filename.o”, and this line means

“gcc main\_echo0.o myshell.o –o my\_echo”. “-o” stands for generating object file and it generates an executable file named by the parameter that comes after “-o”, so it creates the executable file “my\_echo”.

Third line: Since the executable file “my\_echo” is created in the second line, we can run this file by command “./my\_echo Alice”. “Alice” is the string we expect terminal to show us.

echo: $(OBJS)  
 $(CC) main\_$@.o myshell.o -o my\_echo  
 -rm file1 file2  
 ./my\_echo Alice @ file1  
 ./my\_echo Bob @@ file1  
 cat file1  
 ./my\_echo Charlie @ file1  
 cat file1  
 ./my\_echo David @@ file2  
 cat file2

The first and second lines are described above.

Third line: It removes file1 and file2. The “-“ symbol before “rm” means to run the following commands in the “Makefile” even when “rm” fails (for instance removing a non-exist file).

Forth line: As we know, the second line creates the executable file “my\_echo”.

5th line: it should overwrite “Alice” to file1. “@” is the new redirection symbol defined in our “./my\_echo “ command.

6th line: it should append “Alice” to file1.

7th line: it shows the content of file1.

8th line: it should overwrite “Charlie” to file1.

10th line: it should append “David” to file2.

cat: $(OBJS)  
 $(CC) main\_$@.o myshell.o -o my\_cat  
 ./my\_cat file1

Second line: the executable file “my\_cat” is created.

Third line: This line should show the content of file1.

**Step3: Implement header.h**

Copy and paste this content to the file “header.h”:

#if !defined(HEADER\_H)

#define HEADER\_H

#include<stdio.h>

#include<unistd.h> //lseek, STDIN\_FILENO

#include<stdlib.h>

#include <fcntl.h>

#define FILE\_MODE (S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH)

int my\_strlen(char\* format\_string);

void my\_echo(int fd, char\* str);

void my\_printf(char\* format\_string);

void my\_cat(char\* pathname);

#endif

The last four lines are the functions that are called in the five C source files, so they are declared in “header.h”.

**Step4: Implement myshell.c**

Write 4 functions in the file “myshell.c”. Copy and paste this content to the file “myshell.c”:

#include "header.h"

int my\_strlen(char\* str){

…

}

void my\_cat(char\* pathname){

…

}

void my\_echo(int fd, char\* str){

…

}

void my\_printf(char\* str){

…

}

You should implement the body of these functions with the use of File I/O functions.

* Function “my\_strlen” should be able to find the length of an input array (char\* str).
* Function “my\_cat” should be able to show the content of the input pathname (char\* pathname).
* Function “my\_echo” should write the input string (char \* str) in to the specified file descriptor (int fd).
* Function “my\_printf” should be able to output the input string (char \*str) on the screen.

**Step5: Implement main\_printf.c**

The goal of writing this file is to show an arbitrary string on the screen. So, you should relate the implementation of this file and the implementation of function “my\_printf” in file “myshell.c” to show an arbitrary string on the screen.

Hint: As it is described in step 2, by running “Make printf” (run it in step9), it compiles the files “main\_printf.c” and “myshell.c”, and the executes “./a.out”. So you should write a simple main function (“int main”) now which only passes an arbitrary string into the function “my\_printf” in “myshell.c” (this function is described in step 4 and since you are calling it here, it is included in file “header.h” (step3)).

**Step6: Implement main\_echo0.c**

The goal of writing this file is to show the input string on the screen. So, you should relate the implementation of this file and the implementation of function “my\_echo” in file “myshell.c” to show the input string on the screen.

**Hint**: As it is described in step 2, by running “Make echo0” (run it in step9), it compiles the files “main\_echo0.c” and “myshell.c”, and the executes “./my\_echo Alice”. So you should write a simple main function (int main (int argc, char \*argv[]) ) now which only passes the input string into the function “my\_echo” in “myshell.c” (this function is described in step 4 and since you are calling it here, it is included in file “header.h” (step3)).

**Hint**: int main (int argc, char \*argv[])

The input variable “argc” is the number of parameters in the command. For example by running command “./my\_echo Alice”, we have two parameters (argc=2).

The input variable “argv” is a two dimensional array of parameters in the command. For example by running command “./my\_echo Alice”:

Argv[0] = myecho

Argv[1] = Alice

Argv[1][0] = A, Argv[1][1] = l, Argv[1][2] = I, Argv[1][3] = c, Argv[1][4] = e

**Step7: Implement main\_echo.c**

The goal of writing this file is to write the input string into the file. So, you should relate the implementation of this file and the implementation of function “my\_echo” in file “myshell.c” to show the input string on the screen.

**Hint**: As it is described in step 2, by running “Make echo” (run it in step9), it compiles the files “main\_echo.c” and “myshell.c”, and then executes “./my\_echo Alice @ file1” or “./my\_echo Alice @@ file1”. So you should write a simple main function (int main (int argc, char \*argv[]) ) which passes the input string and the input file into the function “my\_echo” in “myshell.c” (this function is described in step 4 and since you are calling it here, it is included in file “header.h” (step3)).

**Hint**: You should pay attention that the difference between “main\_echo.c” and “main\_echo0.c” is the number of input parameters. Here you have 4 input parameters while in “main\_echo0.c” you had 2 input parameters.

Hint: command “./my\_echo Alice @ file1” overwrites “Alice” in file “file1”, while command “./my\_echo Alice @@ file1” appends “Alice” to the end of file “file1”. You should consider it while implementing here.

**Step8: Implement main\_cat.c**

The goal of writing this file is to show the content of file. So, you should relate the implementation in this file and the implementation of function “my\_cat” in file “myshell.c” to show the contents on the screen.

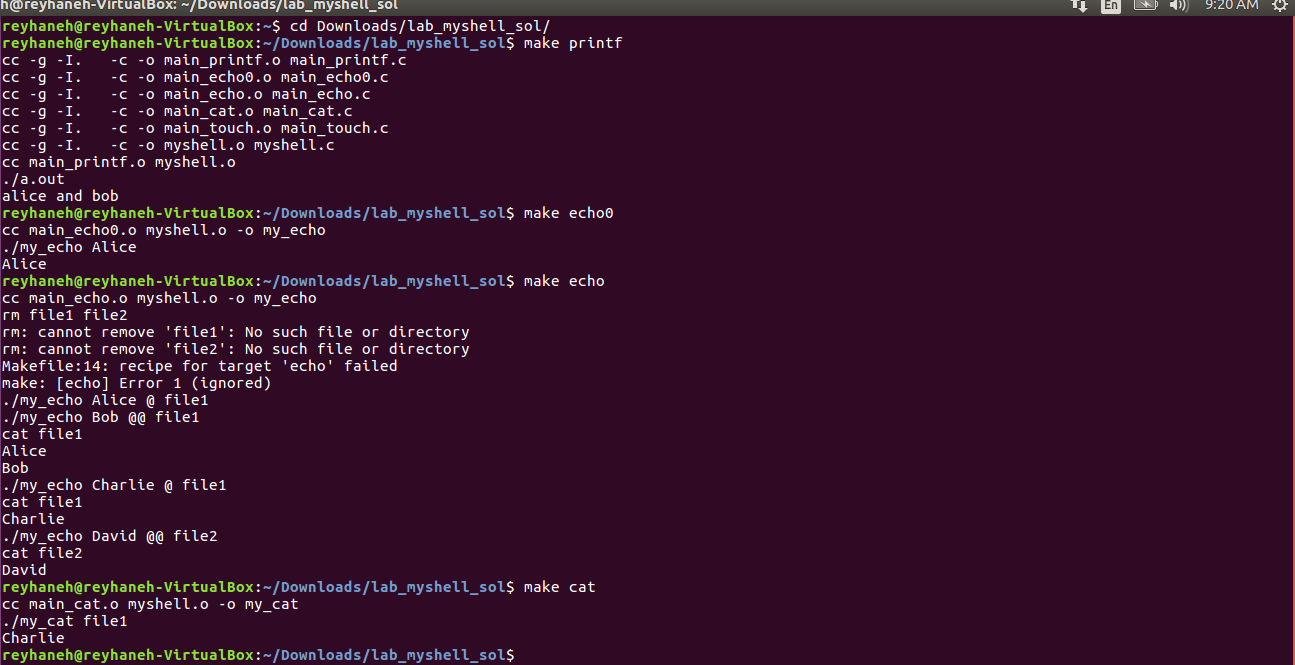
**Hint**: As it is described in step 2, by running “Make cat” (run it in step9), it compiles the files “main\_cat.c” and “myshell.c”, and the executes “./my\_cat file1”. So you should write a simple main function (int main (int argc, char \*argv[]) ) which only passes the input file into the function “my\_cat” in “myshell.c” (this function is described in step 4 and since you are calling it here, it is included in file “header.h” (step3)).

**Step9: Run Makefile**

Enter the “lab” directory and run the following commands in this step:

* Make printf
* Make echo
* Make echo0
* Make cat

If you run these commands, you should see the output the same as the following image (Note that my arbitrary string in function “my\_printf” was “alice and bob”):



As you see, the output of each command is based on the commands in “Makefile”.

**Step10: Execute your program and submit deliverables**

Run these commands in terminal:

* make cat; ./my\_cat file1
* make echo0; ./my\_echo Alice
* make echo; ./my\_echo Bob @ file1; cat file1
* ./my\_echo Alice @@ file1; cat file1
* make printf

Or

* Make printf
* Make echo
* Make echo0
* Make cat

*Submit to Blackboard all the files (including Makefile, header file and 5 c files) and screen shots of running the above commands.*

**Step11 (Bonus task 20%): Implement touch**

This bonus task requires you to design a module that implements my\_touch executable in this MyShell framework.

1. You should add the file “main\_touch.c” and add function “my\_touch” to the file “myshell.c”.
2. You may also want to update “header.c” to include the declaration of the new function.
3. You should add new target and dependency rule to compile main\_touch.c and myshell.c. The rule also includes the commands to run the command “./my\_touch file13; ls .”
4. Implement function “my\_touch” such that it gets the filename from commandline and creates the file.
5. After writing the code, compile the code and execute commands by ”make touch”.