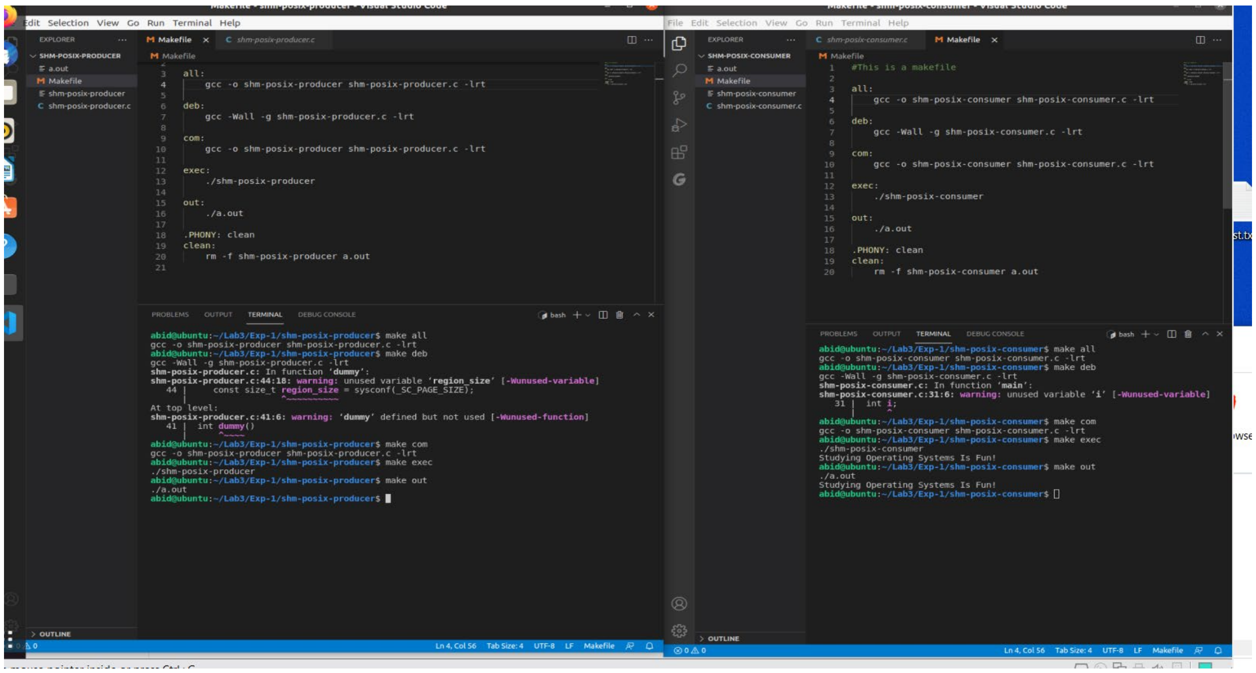
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Lab | Report

Report Subject: OS Experiment - Lab 3

Student ID	: 2019380141
Student Name	: ABID ALI
Experiment Name	: **Inter-Process Communication**
Email	: **abiduu354@gmail.com**

Answer No.1


Makefile - shm-posix-producer - Visual Studio Code
Edit Selection View Go Run Terminal Help
EXPLORER
SHM-POSIX-PRODUCER
a.out
Makefile
shm-posix-producer
shm-posix-producer.c
Makefile - shm-posix-consumer - Visual Studio Code
File Edit Selection View Go Run Terminal Help
SHM-POSIX-CONSUMER
shm-posix-consumer
shm-posix-consumer.c
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
OUTLINE
Ln 4, Col 56 Tab Size: 4 UTF-8 LF Makefile

```
all:
	gcc -o shm-posix-producer shm-posix-producer.c -lrt

deb:
	gcc -Wall -g shm-posix-producer.c -lrt

com:
	gcc -o shm-posix-producer shm-posix-producer.c -lrt

exec:
	./shm-posix-producer

out:
	./a.out

.PHONY: clean
clean:
	rm -f shm-posix-producer a.out
```

```
abid@ubuntu:~/Lab3/Exp-1/shm-posix-producer$ make all
gcc -o shm-posix-producer shm-posix-producer.c -lrt
abid@ubuntu:~/Lab3/Exp-1/shm-posix-producer$ make deb
gcc -Wall -g shm-posix-producer.c -lrt
shm-posix-producer.c: In function ‘dummy’:
shm-posix-producer.c:44:18: warning: unused variable ‘region_size’ [-Wunused-variable]
   44 |      const size_t region_size = sysconf(_SC_PAGE_SIZE);
      |                   ^~~~~~~~~~~
At top level:
shm-posix-producer.c:41:6: warning: ‘dummy’ defined but not used [-Wunused-function]
   41 |  int dummy()
      |      ^~~~~
abid@ubuntu:~/Lab3/Exp-1/shm-posix-producer$ make com
gcc -o shm-posix-producer shm-posix-producer.c -lrt
abid@ubuntu:~/Lab3/Exp-1/shm-posix-producer$ make exec
./shm-posix-producer
abid@ubuntu:~/Lab3/Exp-1/shm-posix-producer$ make out
./a.out
abid@ubuntu:~/Lab3/Exp-1/shm-posix-producer$ 
```

```
#This is a makefile

all:
	gcc -o shm-posix-consumer shm-posix-consumer.c -lrt

deb:
	gcc -Wall -g shm-posix-consumer.c -lrt

com:
	gcc -o shm-posix-consumer shm-posix-consumer.c -lrt

exec:
	./shm-posix-consumer

out:
	./a.out

.PHONY: clean
clean:
	rm -f shm-posix-consumer a.out
```

```
abid@ubuntu:~/Lab3/Exp-1/shm-posix-consumer$ make all
gcc -o shm-posix-consumer shm-posix-consumer.c -lrt
abid@ubuntu:~/Lab3/Exp-1/shm-posix-consumer$ make deb
gcc -Wall -g shm-posix-consumer.c -lrt
shm-posix-consumer.c: In function ‘main’:
shm-posix-consumer.c:31:6: warning: unused variable ‘i’ [-Wunused-variable]
   31 |  int i;
      |      ^
abid@ubuntu:~/Lab3/Exp-1/shm-posix-consumer$ make com
gcc -o shm-posix-consumer shm-posix-consumer.c -lrt
abid@ubuntu:~/Lab3/Exp-1/shm-posix-consumer$ make exec
./shm-posix-consumer
Studying Operating Systems Is Fun!
abid@ubuntu:~/Lab3/Exp-1/shm-posix-consumer$ make out
./a.out
Studying Operating Systems Is Fun!
abid@ubuntu:~/Lab3/Exp-1/shm-posix-consumer$ 
```

Fig-1

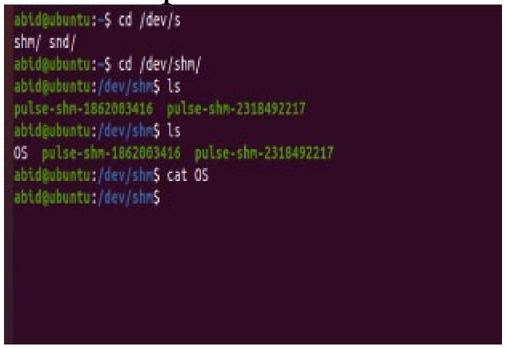
Questions:

1. On Linux, all shared memory objects can be found in /dev/shm. Can you find the shared memory objects that you created?

/dev/shm is a temporary file storage filesystem (see tmpfs) that uses RAM for the storage.
It can function as shared memory that facilitates IPC. It is a world-writeable directory.
Its use is completely optional within the kernel config file (i.e. it is possible not to have dev/shm at all)
Since RAM is significantly faster than disk storage, you can use /dev/shm instead of /tmp for the performance boost if your process is I/O intensive and extensively uses temporary files. (That said, /tmp sometimes use RAM storage too)
The size of /dev/shm is limited by excess RAM on the system, and hence you're more likely to run out of space on this filesystem.

Use-cases

1. To minimize disk I/O. For example, I need to download very large zip files from an FTP server, unzip them, and then import them into a database. I unzip to /dev/shm so that for both the unzip and the import operations the HD only needs to perform half the operation, rather than moving back and forth between source and destination. It speeds up the process immensely.
2. /dev/shm is a good place for separate programs to communicate with each other. For example, one can implement a simple resource lock as a file in shared memory. It is fast because there is no disk access. It also frees you up from having to implement shared memory blocks yourself. You get (almost) all of the advantages of a shared memory block using stdio functions. It also means that simple bash scripts can use shared memory to communicate with each other.

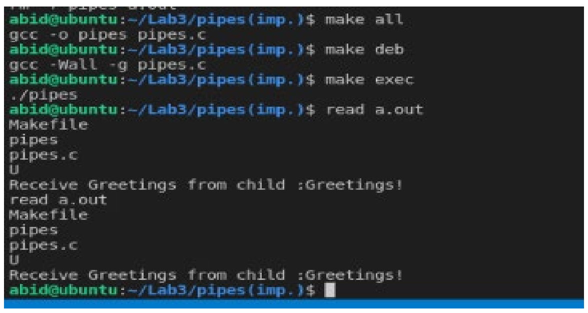


```
abid@ubuntu:~$ cd /dev/s
shm/ snd/
abid@ubuntu:~$ cd /dev/shm/
abid@ubuntu:/dev/shm$ ls
pulse-shm-1862003416  pulse-shm-2318492217
abid@ubuntu:/dev/shm$ ls
OS  pulse-shm-1862003416  pulse-shm-2318492217
abid@ubuntu:/dev/shm$ cat OS
abid@ubuntu:/dev/shm$
```

Yes,I can see the the shared memory objects that was created by me.We can see that,when we type /dev/shm then after listing the files we can see the 'OS" file in /dev/shm directory.

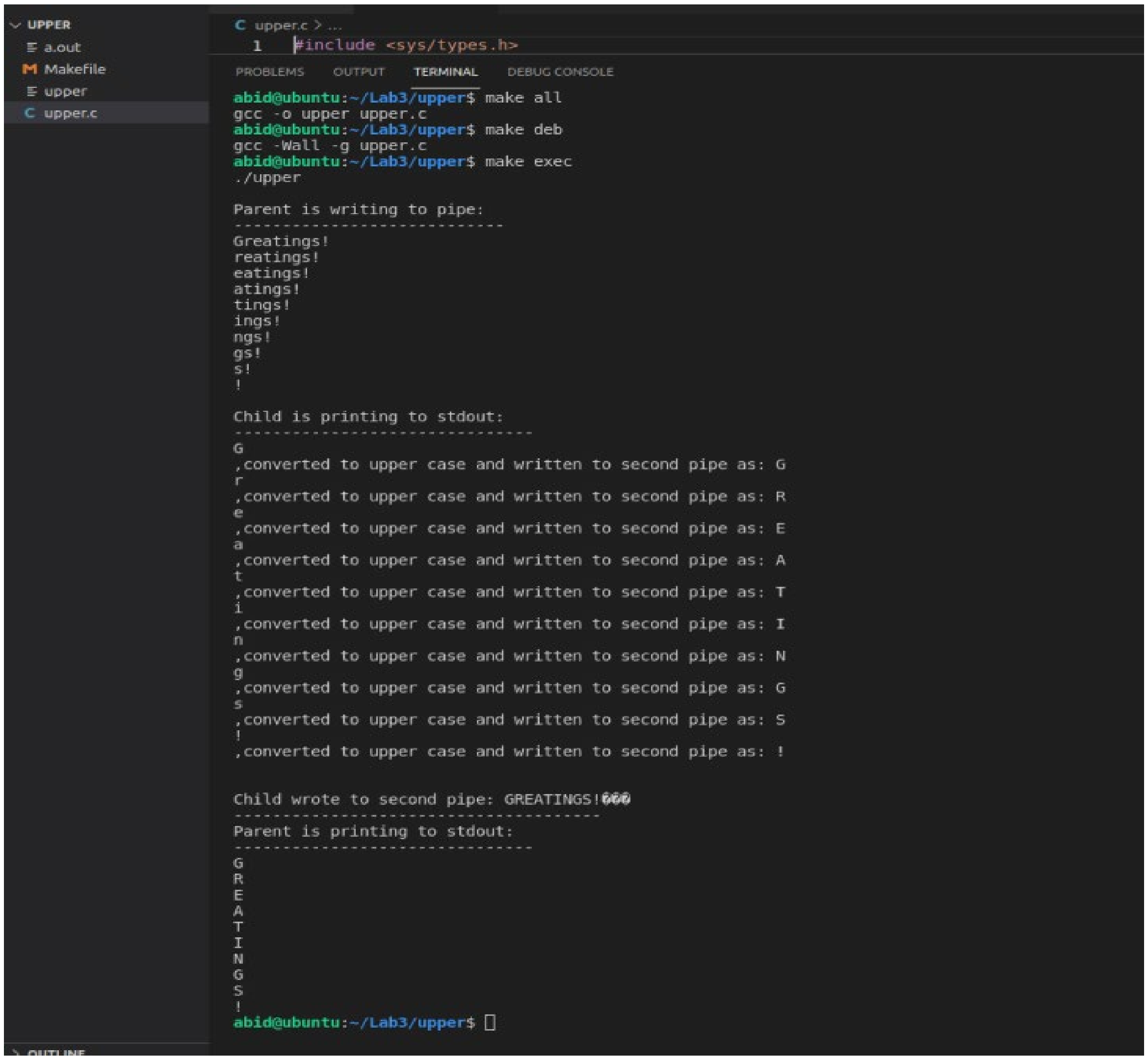
We can see also at Fig-1 that , when we write in producer part then we can see that in the consumer part that code has been passed.So,we can say that there is a occurrence of shared memory

Answer No.2(1)



```
abid@ubuntu:~/Lab3/pipes(imp.)$ make all
gcc -o pipes pipes.c
abid@ubuntu:~/Lab3/pipes(imp.)$ make deb
gcc -Wall -g pipes.c
abid@ubuntu:~/Lab3/pipes(imp.)$ make exec
./pipes
abid@ubuntu:~/Lab3/pipes(imp.)$ read a.out
Makefile
pipes
pipes.c
U
Receive Greetings from child :Greetings!
read a.out
Makefile
pipes
pipes.c
U
Receive Greetings from child :Greetings!
abid@ubuntu:~/Lab3/pipes(imp.)$
```

Answer No.2(2)



```
#include <sys/types.h>
```

```
abid@ubuntu:~/Lab3/upper$ make all
gcc -o upper upper.c
abid@ubuntu:~/Lab3/upper$ make deb
gcc -Wall -g upper.c
abid@ubuntu:~/Lab3/upper$ make exec
./upper

Parent is writing to pipe:
----------------------------
Greatings!
reatings!
eatings!
atings!
tings!
ings!
ngs!
gs!
s!
!

Child is printing to stdout:
-------------------------------
G
,converted to upper case and written to second pipe as: G
r
,converted to upper case and written to second pipe as: R
e
,converted to upper case and written to second pipe as: E
a
,converted to upper case and written to second pipe as: A
t
,converted to upper case and written to second pipe as: T
i
,converted to upper case and written to second pipe as: I
n
,converted to upper case and written to second pipe as: N
g
,converted to upper case and written to second pipe as: G
s
,converted to upper case and written to second pipe as: S
!
,converted to upper case and written to second pipe as: !


Child wrote to second pipe: GREATINGS!���
---------------------------------------
Parent is printing to stdout:
-------------------------------
G
R
E
A
T
I
N
G
S
!
abid@ubuntu:~/Lab3/upper$
```

Questions:

1 Does the pipe allow bidirectional communication, or is communication unidirectional in an ordinary pipe?

Solution:

We can see that, we used a two way ordinary pipe in the second question.That means, we used one unidirectional pipe and another unidirectional pipe. Ordinary pipes are unidirectional, allowing only one-way communication-Ordinary pipes allow two processes to communicate in standard producer– consumer fashion: the producer writes to one end of the pipe (the write-end) and the consumer reads from the other end (the read-end).

So,we can say that, communication is unidirectional in an ordinary pipe
But,when we use two ordinary pipe is connected it becomes two way communication.

2 If two-way communication is allowed, is it half duplex (data can travel only one way at a time) or full duplex (data can travel in both directions at the same time)?

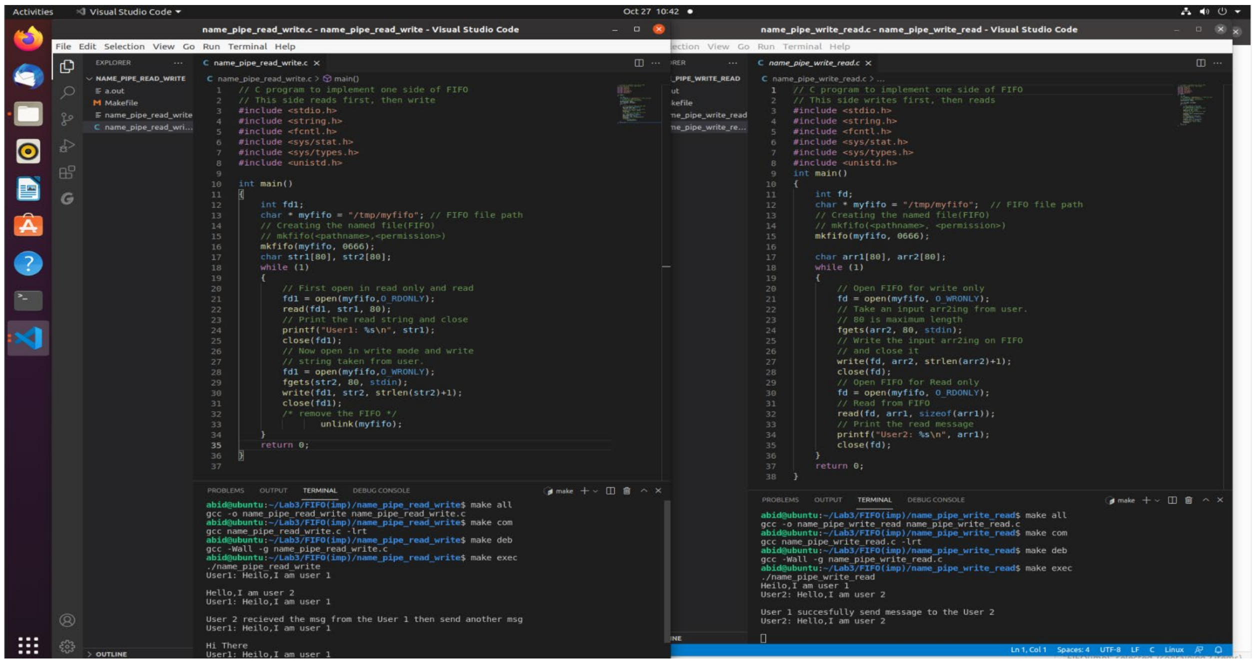
Solution:

If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction. In a half-duplex or semiduplex system, both parties can communicate with each other, but not simultaneously; the communication is one direction at a time. An example of a half-duplex device is a walkie-talkie, a two-way radio that has a push-to-talk button. ... In this experiment,we made a program almost like a walkie-talkie or a chat bot where we send message from one terminal and another person can send message but it doesn't act instantly.
As we can see it's half duplex.

3 Must a relationship (such as parent–child) exist between the communicating processes in an ordinary pipe?

Yes, (unnamed) pipes can only connect parent/child processes (unless you try *very* hard). Named pipes may be used to pass data between unrelated processes, while normal
So, there must be a relationship where parent–child exist between the communicating processes in an ordinary pipe.Otherwise,information can't be pass from one direction to another direction.

Answer No.2(3)



```c
// C program to implement one side of FIFO
// This side reads first, then write
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int fd1;
    char * myfifo = "/tmp/myfifo"; // FIFO file path
    // Creating the named file(FIFO)
    // mkfifo(<pathname>,<permission>)
    mkfifo(myfifo, 0666);
    char str1[80], str2[80];
    while (1)
    {
        // First open in read only and read
        fd1 = open(myfifo,O_RDONLY);
        read(fd1, str1, 80);
        // Print the read string and close
        printf("User1: %s\n", str1);
        close(fd1);
        // Now open in write mode and write
        // string taken from user.
        fd1 = open(myfifo,O_WRONLY);
        fgets(str2, 80, stdin);
        write(fd1, str2, strlen(str2)+1);
        close(fd1);
        /* remove the FIFO */
                unlink(myfifo);
    }
    return 0;
}
```

```
abid@ubuntu:~/Lab3/FIFO(imp)/name_pipe_read_write$ make all
gcc -o name_pipe_read_write name_pipe_read_write.c
abid@ubuntu:~/Lab3/FIFO(imp)/name_pipe_read_write$ make com
gcc name_pipe_read_write.c -lrt
abid@ubuntu:~/Lab3/FIFO(imp)/name_pipe_read_write$ make deb
gcc -Wall -g name_pipe_read_write.c
abid@ubuntu:~/Lab3/FIFO(imp)/name_pipe_read_write$ make exec
./name_pipe_read_write
User1: Heilo,I am user 1

Hello,I am user 2
User1: Heilo,I am user 1

User 2 recieved the msg from the User 1 then send another msg
User1: Heilo,I am user 1

Hi There
User1: Heilo,I am user 1
```

```c
// C program to implement one side of FIFO
// This side writes first, then reads
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int fd;
    char * myfifo = "/tmp/myfifo";  // FIFO file path
    // Creating the named file(FIFO)
    // mkfifo(<pathname>, <permission>)
    mkfifo(myfifo, 0666);

    char arr1[80], arr2[80];
    while (1)
    {
        // Open FIFO for write only
        fd = open(myfifo, O_WRONLY);
        // Take an input arr2ing from user.
        // 80 is maximum length
        fgets(arr2, 80, stdin);
        // Write the input arr2ing on FIFO
        // and close it
        write(fd, arr2, strlen(arr2)+1);
        close(fd);
        // Open FIFO for Read only
        fd = open(myfifo, O_RDONLY);
        // Read from FIFO
        read(fd, arr1, sizeof(arr1));
        // Print the read message
        printf("User2: %s\n", arr1);
        close(fd);
    }
    return 0;
}
```

```
abid@ubuntu:~/Lab3/FIFO(imp)/name_pipe_write_read$ make all
gcc -o name_pipe_write_read name_pipe_write_read.c
abid@ubuntu:~/Lab3/FIFO(imp)/name_pipe_write_read$ make com
gcc name_pipe_write_read.c -lrt
abid@ubuntu:~/Lab3/FIFO(imp)/name_pipe_write_read$ make deb
gcc -Wall -g name_pipe_write_read.c
abid@ubuntu:~/Lab3/FIFO(imp)/name_pipe_write_read$ make exec
./name_pipe_write_read
Heilo,I am user 1
User2: Hello,I am user 2

User 1 succesfully send message to the User 2
User2: Hello,I am user 2
```

Questions:

1 What are the advantages of using named pipe?

Answer:

A named pipe is a special file that is used to transfer data between unrelated processes. One (or more) processes write to it, while another process reads from it. Named pipes are visible in the file system and may be viewed with `ls' command like any other file. (Named pipes are also called fifos; this term stands for `First In, First Out'.)

Named pipes may be used to pass data between unrelated processes, while normal (unnamed) pipes can only connect parent/child processes (unless you try *very* hard).

Named pipes (fifo) have three advantages we know about:

- you don't have to start the reading/writing processes at the same time.
- you can have multiple readers/writers which do not need common ancestry.
- as a file you can control ownership and permissions.

Problems:

I was confused with the problem and how to use terminal and what to type. Couldn’t understand how to do the makefile.Not familiar with the concept of pipe.

Solution:

To solve those problems I looked for information in internet. In order to understand some questions and procedure I also asked the teacher to help me understand them. And provided instructions helped to solve some of my errors during the experiment.

Conclusion:

At the beginning, I was unfamiliar with pipe and how to implement it. Gradually, reading lot of article and reading teachers ppt then I solved those problem one by one. In this experiment ,small helps and suggestions from the teacher was very helpful that saved my time.I enjoyed the practical and learned lot of interesting things.

Attachments:

1) ABID ALI_2019380141_OS(Lab 3).docx
2) ABID ALI_2019380141_OS(Lab 3).pdf
3)Code_ABID ALI_2019380141(Lab-3).zip