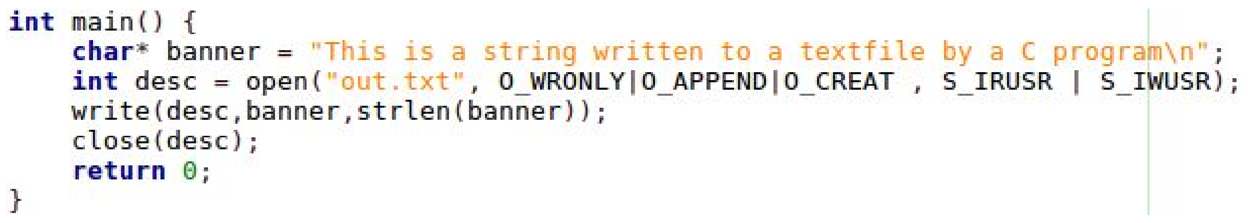
CO327 LAB 02 (Interprocess communication)

E/13/058
M.D.R.A.M DE SILVA
lab02_E13058.tar.bzz

Example 1.1



```
int main() {
    char* banner = "This is a string written to a textfile by a C program\n";
    int desc = open("out.txt", O_WRONLY|O_APPEND|O_CREAT , S_IRUSR | S_IWUSR);
    write(desc,banner,strlen(banner));
    close(desc);
    return 0;
}
```

Exercise 1.1

a.Explain what the flags O _WRONLY , O_APPEND and O_CREAT do.

O _WRONLY : Means it is open for writing only.

O_APPEND : Before each write the file offset is set to the end of the file if O_APPEND is set.

O_CREAT : If no file already exists a new file will be created

b. Explain what the modes S_IRUSR , S_IWUSR do.

S_IRUSR : 00400 user has read permission

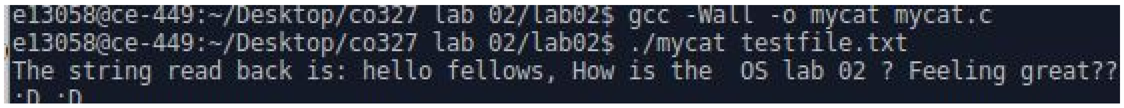
S_IWUSR : 00200 user has write permission

Exercise 1.2:

a. Write a program called mycat which reads a text file and writes the output to the standard Output.

-->Code is Included in the submission.

Mycat.c output :



```
e13058@ce-449:~/Desktop/co327 lab 02/lab02$ gcc -Wall -o mycat mycat.c
e13058@ce-449:~/Desktop/co327 lab 02/lab02$ ./mycat testfile.txt
The string read back is: hello fellows, How is the  OS lab 02 ? Feeling great??
:D :D
```

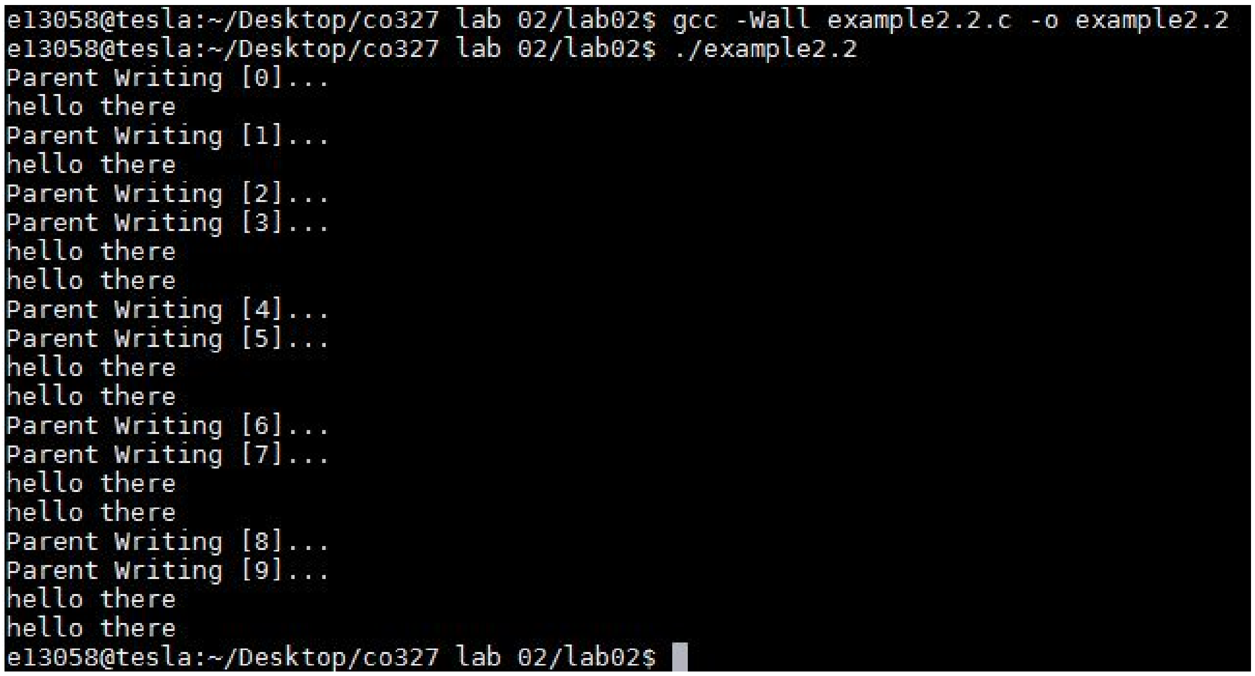
b. Write a program called mycopy using open() , read() , write() and close() which takes two arguments, viz. source and target file names, and copy the content of the source file into the target file. If the target file exists, just overwrite the file.

-->Code is Included in the submission.

<u>Pipes</u>

Exercise 2.1(Refer code in example 2.2.c)

Example 2.2.c output:



```
e13058@tesla:~/Desktop/co327 lab 02/lab02$ gcc -Wall example2.2.c -o example2.2
e13058@tesla:~/Desktop/co327 lab 02/lab02$ ./example2.2
Parent Writing [0]...
hello there
Parent Writing [1]...
hello there
Parent Writing [2]...
Parent Writing [3]...
hello there
hello there
Parent Writing [4]...
Parent Writing [5]...
hello there
hello there
Parent Writing [6]...
Parent Writing [7]...
hello there
hello there
Parent Writing [8]...
Parent Writing [9]...
hello there
hello there
e13058@tesla:~/Desktop/co327 lab 02/lab02$
```

a. What does write(STDOUT_FILENO, &buff, count); do?

Writes up to “count” number of bytes from the buffer pointed buff to the file referred to by STDOUT_FILENO.
STDOUT_FILENO is nothing but standard output.

b. Can you use a pipe for bidirectional communication? Why (not)?

- Can’t use for bidirectional communication If you use Ordinary pipes (Unnamed Pipes) . Because, it only allows to write from one process(parent) and read from the other process(child) through the pipe. For example, the producer writes to one end (write end of the pipe) and the consumer reads that from the other end(the read end of the pipe). If two way communication is required , two pipes can be used with each pipe sending data in a differennt direction.

- Can use for bidirectional communication if you use Named pipes. Here, no parent -child relationship is needed between communicating processes and several processes can use named pipes for communication.

c. Why cannot unnamed pipes be used to communicate between unrelated processes?

- Unnamed pipes can only be used for communications between processes on the same machine and they exists only when the processors are communicating with each other. Hence, cannot be used for unrelated processes.

- The main reason is that, an ordinary pipe cannot be accessed outside the process which created it. Ordinary pipes(unnamed pipes) can be used for parent child communication processes as a parent can create a child process and a pipe within its process.

d. Now write a program where the parent reads a string from the user and send it to the child and the child capitalizes each letter and sends back the string to parent and parent displays it. You'll need two pipes to communicate both ways.

→ Code is included in the submission.

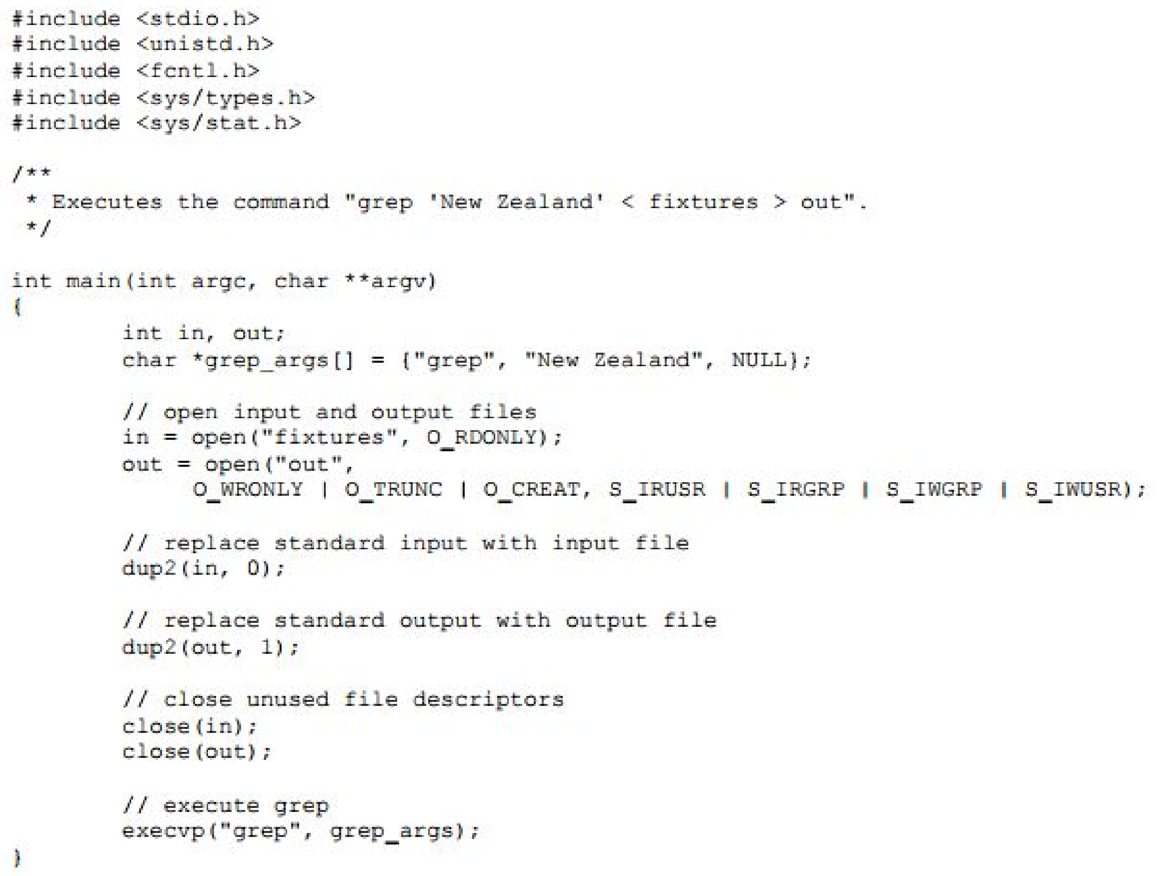
<u>dup() and dup2() System Calls</u>

Exercise3.1

Write a program that uses fork() and exec() to create a process of ls and get the result of ls back to the parent process and print it from the parent using pipes. If you cannot do this, explain why.

- All the process communication means are lost as exec() replaces all original code when it is used to replace the execution image of a forked child. And since, a successful exec() doesn’t return back to the process which it was initiated by.

Exercise3.2



```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

/**
 * Executes the command "grep 'New Zealand' < fixtures > out".
 */

int main(int argc, char **argv)
{
        int in, out;
        char *grep_args[] = {"grep", "New Zealand", NULL};

        // open input and output files
        in = open("fixtures", O_RDONLY);
        out = open("out",
             O_WRONLY | O_TRUNC | O_CREAT, S_IRUSR | S_IRGRP | S_IWGRP | S_IWUSR);

        // replace standard input with input file
        dup2(in, 0);

        // replace standard output with output file
        dup2(out, 1);

        // close unused file descriptors
        close(in);
        close(out);

        // execute grep
        execvp("grep", grep_args);
}
```

a. What does 1 in the line dup2(out,1); in the above program stands for? 1 is the default file id for **stdout.**

b. The following questions are based on the example3.2.c

i. Compare and contrast the usage of dup() and dup2(). Do you think both functions are necessary? If yes, identify use cases for each function. If not, explain why.

- dup() and dup2() both are system calls which are used to duplicate file descriptors.
- dup() : dup(int filedes) takes a file descriptor as argument and duplicates the next lowest available file descriptor with the same file table pointer. It basically uses the lowest-numbered unused descriptor for the new descriptor.
- dup2() makes the new file descriptor be the copy o the old file descriptor ,closing the old descriptor if necessary. If old descriptor is not valid then the call fails.

- Yes, both functions are necessary.
- Use cases : dup() → I/O redirection, dup2() → Redirecting Error messages.

ii. There's one glaring error in this code (if you find more than one, let me know!). Can you identify what that is (hint: look at the output)?
Error is that it doesn't return at the command line . This is dues to not handlind operations in order.
FIrst we have to cat and then do the grep . In here, if we set a wait() command in the parent process to wait until child does the cat part then the error would be corrected.

iii. Modify the code to rectify the error you have identified above.
Remove close(0), close(1) in code
→ Code is included in the submission.

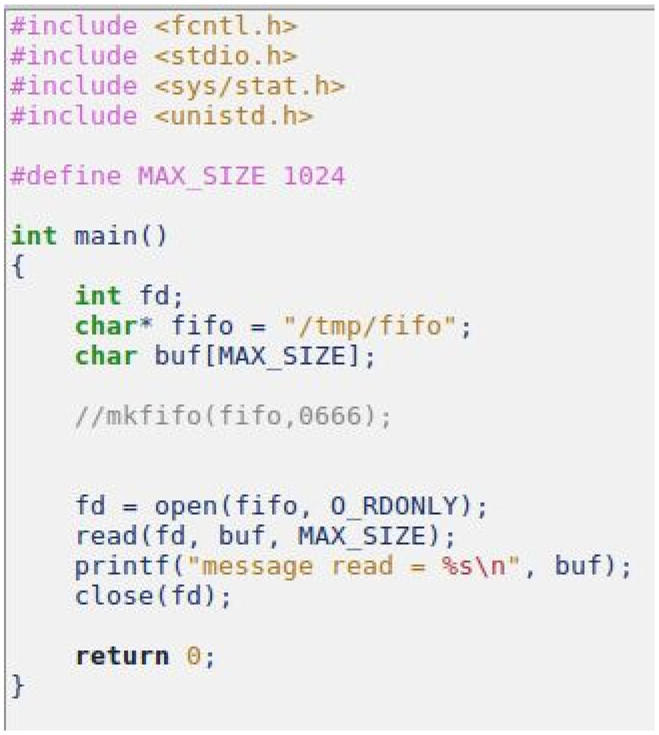
c. Write a program that executes "cat fixtures | grep | cut -b 1-9" command. A skeleton code for this is provided as exercise 3.2.c_skel.c. You can use this as your starting point, if necessary.

→ Code is included in the submission.

Exercise4.1

a. Comment out the line "mkfifo(fifo,0666);" in the reader and recompile the program.

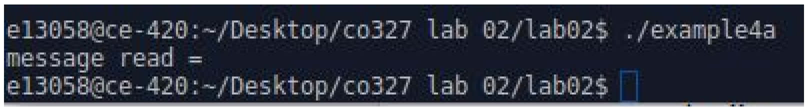
Scenario 1



```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>

#define MAX_SIZE 1024

int main()
{
    int fd;
    char* fifo = "/tmp/fifo";
    char buf[MAX_SIZE];

    //mkfifo(fifo,0666);


    fd = open(fifo, O_RDONLY);
    read(fd, buf, MAX_SIZE);
    printf("message read = %s\n", buf);
    close(fd);

    return 0;
}
```

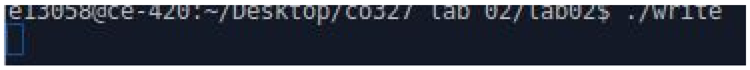
Test the programs by alternating which program is invoked first.

When reader is invoked first,
Reader looks for a value in fifo and since no value is written doesn't return a message.



```
e13058@ce-420:~/Desktop/co327 lab 02/lab02$ ./example4a
message read =
e13058@ce-420:~/Desktop/co327 lab 02/lab02$
```

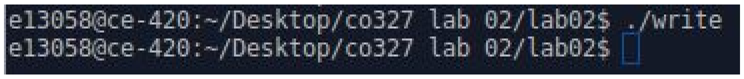
Writer writes and wait till reader read.



```
e13058@ce-420:~/Desktop/co327 lab 02/lab02$ ./write
```

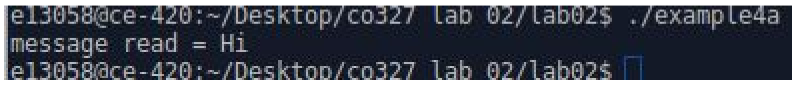
When writer invoked first then reader,

- Writer waits and returns when the reader reads the value written



```
e13058@ce-420:~/Desktop/co327 lab 02/lab02$ ./write
e13058@ce-420:~/Desktop/co327 lab 02/lab02$
```

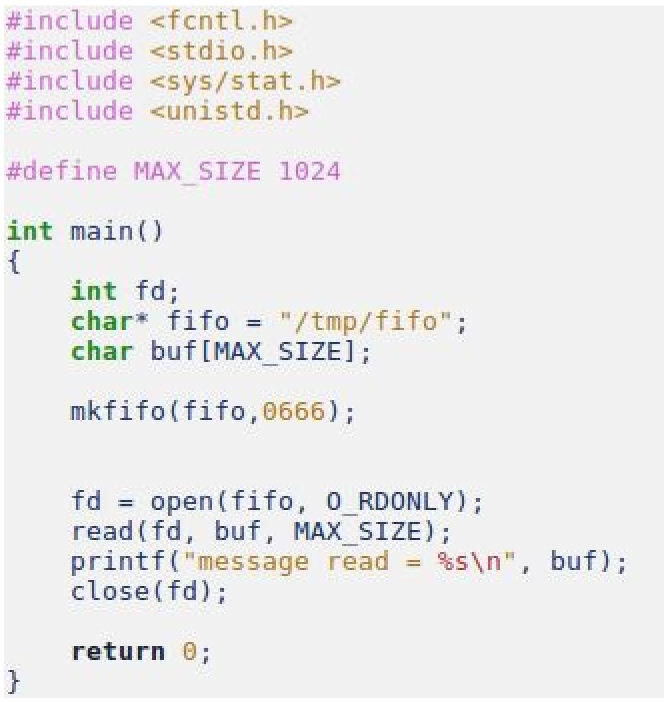
- The written value is read by reader.



```
e13058@ce-420:~/Desktop/co327 lab 02/lab02$ ./example4a
message read = Hi
e13058@ce-420:~/Desktop/co327 lab 02/lab02$
```

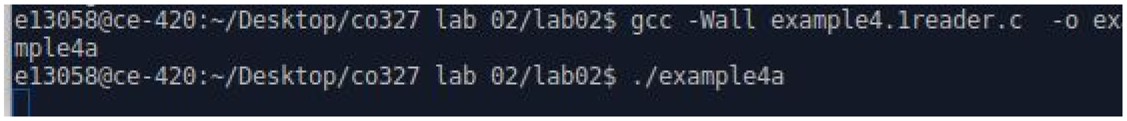
<u>Senario 2</u>

Now, reset the reader to the original, comment the same line in the writer and repeat the test.



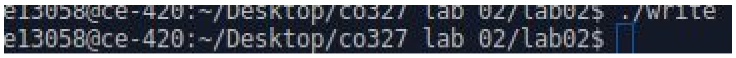
```c
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>

#define MAX_SIZE 1024

int main()
{
    int fd;
    char* fifo = "/tmp/fifo";
    char buf[MAX_SIZE];

    mkfifo(fifo,0666);


    fd = open(fifo, O_RDONLY);
    read(fd, buf, MAX_SIZE);
    printf("message read = %s\n", buf);
    close(fd);

    return 0;
}
```

Reader executed first,
 reader waits while a value is written by writer.

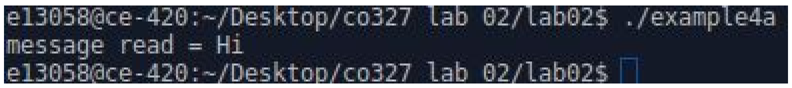


```
e13058@ce-420:~/Desktop/co327 lab 02/lab02$ gcc -Wall example4.1reader.c  -o ex
mple4a
e13058@ce-420:~/Desktop/co327 lab 02/lab02$ ./example4a
```

Writer writes a value, and then reader reads the value written by writer and exit.



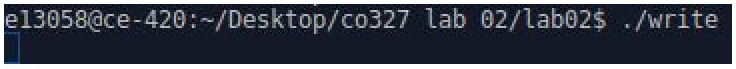
```
e13058@ce-420:~/Desktop/co327 lab 02/lab02$ ./write
e13058@ce-420:~/Desktop/co327 lab 02/lab02$
```



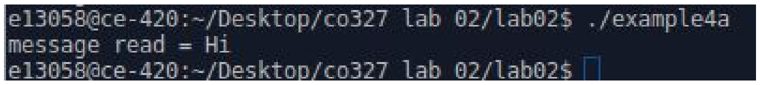
```
e13058@ce-420:~/Desktop/co327 lab 02/lab02$ ./example4a
message read = Hi
e13058@ce-420:~/Desktop/co327 lab 02/lab02$
```

When writer invoked first then reader,

Writer writes and wait till reader reads


```
e13058@ce-420:~/Desktop/co327 lab 02/lab02$ ./write
```

Reader reads the value written by writer and writer exits when this happen.


```
e13058@ce-420:~/Desktop/co327 lab 02/lab02$ ./example4a
message read = Hi
e13058@ce-420:~/Desktop/co327 lab 02/lab02$
```

Why do you think this happens?

The main reason for this behavior is not having the pipe connection between the two processes. We saw that when no pipe was used the value written could only be seen when the writer is invoked at first. But when we use a pipe in either ways (writer first or reader first)
The output message is seen. Hence, using a pipe would also save more time in debugging as it will reduce the debugging concerns.

b. Write two programs: one, which takes a string from the user and sends it to the other process, and the other, which takes a string from the first program, capitalizes the letters and send it back to the first process. The first process should then print the line out. Use the built in command tr() to convert the string to uppercase.

→ Code is included in the submission.