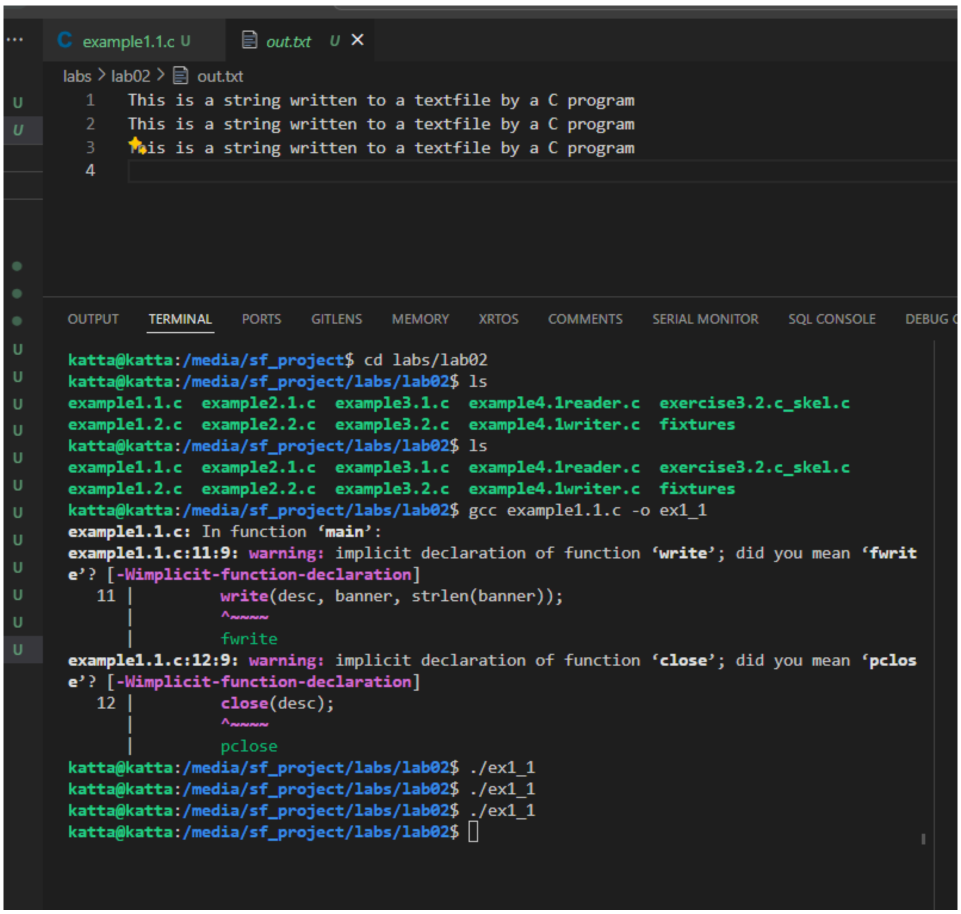
E/19/129
K.H. Gunawardana

<u>Lab 02 - Interprocess Communication</u>

<u>Exercise1.1</u>

Output:



a. Explain what the flags O_WRONLY, O_APPEND and O_CREAT do.

These flags, used with the `open()` function, control how a file is opened. `O_WRONLY` specifies that the file should be opened for writing only, meaning any attempt to read from it will fail. `O_APPEND` ensures that any data written to the file will be appended to the end of existing content, positioning the file pointer automatically at the end of the file before each write operation. `O_CREAT` instructs `open()` to create a new file if it doesn't exist, with its permission bits set according to the mode argument (explained below). If the file already exists, `O_CREAT` has no effect unless used with `O_EXCL`.

b. Explain what the modes S_IRUSR, S_IWUSR do.

The modes `S_IRUSR` and `S_IWUSR` specify the file's permission bits using symbolic constants defined in `<sys/stat.h>`. In the code, these modes are combined using the bitwise OR operator (`|`). `S_IRUSR` grants read permission to the file's owner (user), while `S_IWUSR` grants write permission to the file's owner (user). Combining these flags and modes, the code opens a file named "out.txt" for writing only (`O_WRONLY`), creates the file if it doesn't exist (`O_CREAT`), appends data to the end of the file (`O_APPEND`), and sets the file permissions so that only the owner (user) can read and write to it (`S_IRUSR | S_IWUSR`).

Exercise1.2

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

Code:

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

#define SIZE 100

int main(int argc, char *argv[])
{
    int file;           // File descriptor
    char buffer[SIZE]; // Buffer for reading data
    int bytes_read;     // Number of bytes read

    // Check if the correct number of arguments is provided
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    // Open the file in read-only mode
    file = open(argv[1], O_RDONLY);
    if (file == -1)
    {
        fprintf(stderr, "Couldn't open the file for reading\n");
```

```
        return 1;
    }

    // Read from the file and write to stdout
    while ((bytes_read = read(file, buffer, SIZE)) > 0)
    {
        if (write(STDOUT_FILENO, buffer, bytes_read) != bytes_read)
        {
            fprintf(stderr, "Reading Error\n");
            close(file);
            return 1;
        }
    }

    // Check for read error
    if (bytes_read == -1)
    {
        fprintf(stderr, "Reading Error\n");
        close(file);
        return 1;
    }

    // Close the file
    if (close(file) == -1)
    {
        fprintf(stderr, "Closing Error\n");
        return 1;
    }

    return 0;
}
```

Output:

```
katta@katta:/media/sf_project/labs/lab02$ ./mycat out.txt
This is a string written to a textfile by a C program
This is a string written to a textfile by a C program
This is a string written to a textfile by a C program
String to write and read
katta@katta:/media/sf_project/labs/lab02$
```

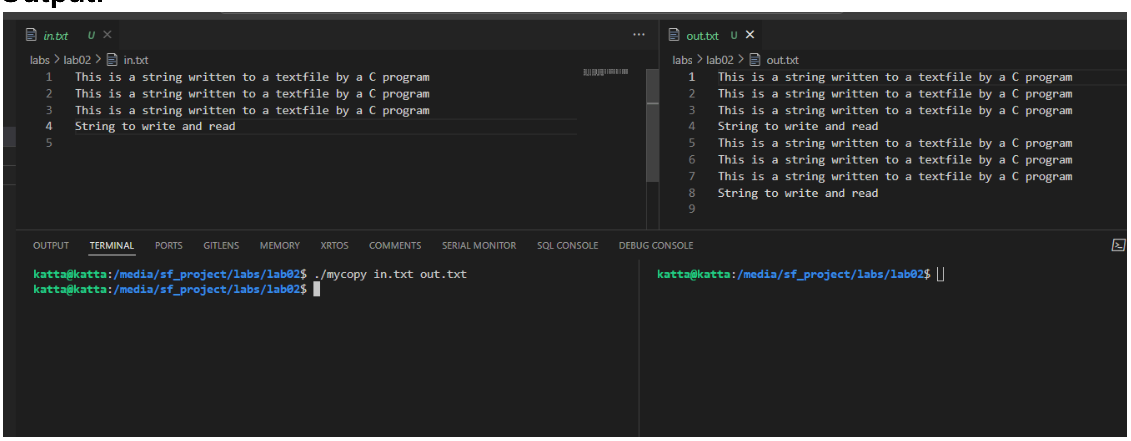
b. Write a program called mycopy using open(), read(), write() and close() which takes two arguments, viz. source and target file names, and copy the content of the source file into the target file. If the target file exists, just overwrite the file.

Code:

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

#define SIZE 100

int main(int argc, char *argv[])
{
    int fileFrom, fileTO; // File descriptor
    char buffer[SIZE];    // Buffer for reading data
    int bytes_read;       // Number of bytes read

    // Check if the correct number of arguments is provided
    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    // Open the file in read-only mode
    fileFrom = open(argv[1], O_RDONLY);
    fileTO = open(argv[2], O_WRONLY | O_APPEND | O_CREAT, S_IRUSR | S_IWUSR);

    if (fileFrom == -1 || fileTO == -1)
    {
        fprintf(stderr, "Couldn't open the file for reading\n");
        return 1;
    }

    // Read from the file and write to stdout
    while ((bytes_read = read(fileFrom, buffer, SIZE)) > 0)
    {
        if (write(fileTO, buffer, bytes_read) != bytes_read)
        {
            fprintf(stderr, "Reading Error\n");
            close(fileFrom);
            close(fileTO);
```

```c
                return 1;
            }
    }

    // Check for read error
    if (bytes_read == -1)
    {
        fprintf(stderr, "Reading Error\n");
        close(fileFrom);
        close(fileTO);
        return 1;
    }

    // Close the file
    close(fileFrom);
    close(fileTO);

    return 0;
}
```

Output:



Exercise2.1

Output:

```
hello!
katta@katta:/media/sf_project/labs/lab02$ gcc example2.2.c -o ex2_2
katta@katta:/media/sf_project/labs/lab02$ ./ex2_2
Parent Writing [0]...
hello there
Parent Writing [1]...
Parent Writing [2]...
hello there
hello there
Parent Writing [3]...
Parent Writing [4]...
hello there
hello there
Parent Writing [5]...
Parent Writing [6]...
hello there
hello there
Parent Writing [7]...
Parent Writing [8]...
hello there
hello there
Parent Writing [9]...
katta@katta:/media/sf_project/labs/lab02$ hello there
```

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

The line `write(STDOUT_FILENO, buff, count);` in the code writes data from the `buff` array to the standard output. `STDOUT_FILENO` is the file descriptor for standard output. The second argument, `buff`, is the pointer to the data to be written. The third argument, `count`, specifies the number of bytes to write from `buff`. This ensures only the data read from the pipe is written to the standard output.

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

Unnamed pipes are unidirectional, limiting them to one-way communication with separate read and write ends. This restriction makes bidirectional communication impossible. However, alternatives like named pipes (FIFOs), sockets, and shared memory support bidirectional communication. Named pipes enable multiple processes to read and write from the same pipe. Sockets offer flexible IPC across machines, supporting both connection-oriented and connectionless communication. Shared memory allows processes to share memory segments, enabling them to read and write data to the same memory location.

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

Unnamed pipes are restricted to communication between related processes, like a parent and its child processes created with a fork. This limitation stems from the scope of file descriptors associated with unnamed pipes. When created with the pipe system call, two file descriptors are returned: one for reading and one for writing. These descriptors are only accessible to the creating process and its child processes. Unrelated processes cannot access or utilize the file descriptors of an unnamed pipe created by another process. Thus, unnamed pipes cannot facilitate communication between unrelated processes. Alternatives, like named pipes, sockets, and message queues, should be explored for inter-process communication between unrelated processes.

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

Code:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <sys/wait.h>

#define READ_END 0
#define WRITE_END 1
#define BUFFER_SIZE 256

void capitalize(char *str)
{
    for (int i = 0; str[i] != '\0'; i++)
    {
        if (isalpha(str[i]))
        {                              // Check if character is alphabetic
            str[i] = toupper(str[i]); // Convert to uppercase
        }
    }
}

int main()
{
    int pipe_ends_parent2child[2], pipe_ends_child2parent[2];
    pid_t pid;
    char buffer[BUFFER_SIZE];
```

```c
    int count;

    // Create pipes
    if (pipe(pipe_ends_parent2child) || pipe(pipe_ends_child2parent))
    {
        perror("Pipe creation");
        return -1;
    }

    // Fork process
    pid = fork();
    if (pid < 0)
    {
        perror("Fork");
        return -1;
    }

    if (pid > 0)
    {                                              // Parent process
        close(pipe_ends_parent2child[READ_END]);   // Close unused read end
        close(pipe_ends_child2parent[WRITE_END]); // Close unused write end

        printf("Enter a line of text: ");

        // Read input from user
        if (fgets(buffer, BUFFER_SIZE, stdin) == NULL)
        {
            printf("Error reading input.\n");
        }

        // Write input to child process
        write(pipe_ends_parent2child[WRITE_END], buffer, strlen(buffer));
        close(pipe_ends_parent2child[WRITE_END]);

        // Wait for child process to finish
        wait(NULL);

        // Read output from child process
        count = read(pipe_ends_child2parent[READ_END], buffer, BUFFER_SIZE);
        close(pipe_ends_child2parent[READ_END]);

        printf("The Output from child: %s", buffer);

        exit(EXIT_SUCCESS); // Exit parent process
    }
```

```c
    if (pid == 0)
    {                                                   // Child process
        close(pipe_ends_parent2child[WRITE_END]); // Close unused write end
        close(pipe_ends_child2parent[READ_END]);  // Close unused read end

        // Read input from parent process
        count = read(pipe_ends_parent2child[READ_END], buffer, BUFFER_SIZE);
        close(pipe_ends_parent2child[READ_END]); // Close unused read end

        // Capitalize alphabetic characters
        capitalize(buffer);

        // Write modified input back to parent process
        write(pipe_ends_child2parent[WRITE_END], buffer, strlen(buffer));
        close(pipe_ends_child2parent[WRITE_END]); // Close unused write end
    }

    return 0;
}
```

Output:

```
katta@katta:/media/sf_project/labs/lab02$ gcc ex2_d.c -o ex2_d
katta@katta:/media/sf_project/labs/lab02$ ./ex2_d
Enter a line of text: co327 - Os
The Output from child: CO327 - OS
katta@katta:/media/sf_project/labs/lab02$ ./ex2_d
Enter a line of text: abcdefg12345&^^poZZZZ
The Output from child: ABCDEFG12345&^^POZZZZ
katta@katta:/media/sf_project/labs/lab02$
```

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.

Code:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>

#define READ_END 0
#define WRITE_END 1
#define BUFFER_SIZE 256

int main()
{
    int pipe_ends_child2parent[2];
    pid_t pid;
    char buffer[BUFFER_SIZE];
    int count;

    // Create pipe
    if (pipe(pipe_ends_child2parent))
    {
        perror("Pipe creation");
        return -1;
    }

    // Fork process
    pid = fork();
    if (pid < 0)
    {
        perror("Fork");
        return -1;
    }

    if (pid > 0)
    {                                               // Parent process
        close(pipe_ends_child2parent[WRITE_END]); // Close unused write
```

```
end

        // Wait for child process to finish
        wait(NULL);

        // Read output from child process
        printf("The Output from child:\n");

        count = read(pipe_ends_child2parent[READ_END], buffer,
BUFFER_SIZE);

        // Write output to stdout
        write(STDOUT_FILENO, buffer, count);

        close(pipe_ends_child2parent[READ_END]); // Close read end
        exit(EXIT_SUCCESS);                      // Exit parent process
    }

    if (pid == 0)
    { // Child process
        // Redirect stdout to the write end of the pipe
        dup2(pipe_ends_child2parent[WRITE_END], STDOUT_FILENO);

        // Close unused ends of the pipe
        close(pipe_ends_child2parent[READ_END]);
        close(pipe_ends_child2parent[WRITE_END]);

        // Execute ls command
        if (execlp("ls", "ls", NULL) == -1)
        {
            perror("execlp failed");
            exit(EXIT_FAILURE);
        }
    }

    return 0;
}
```

Output:

```
katta@katta:/media/sf_project/labs/lab02$ gcc ex3_1.c -o ex3_1
katta@katta:/media/sf_project/labs/lab02$ ./ex3_1
The Output from child:
ex1_1
ex1_2
ex2_1
ex2_2
ex2_d
ex2_d.c
ex3_1
ex3_1.c
example1.1.c
example1.2.c
example2.1.c
example2.2.c
example3.1.c
example3.2.c
example4.1reader.c
example4.1writer.c
exercise3.2.c_skel.c
fixtures
in.txt
mycat
mycat.c
mycopy
mycopy.c
out
out.txt
katta@katta:/media/sf_project/labs/lab02$
```

Exercise3.2

a. What does 1 in the line dup2(out,1); in the above program stand for?

   In the line dup2(out, 1); from the above program, the 1 stands for the file descriptor number of standard output (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()` are both crucial functions in Unix-like systems for duplicating file descriptors.

      `dup()` is straightforward, duplicating a file descriptor and assigning the new one to the lowest-numbered unused descriptor. It doesn't allow specifying a particular descriptor number, making it suitable for general use cases where precise control over the descriptor number isn't needed.
      `dup2()` offers more flexibility by letting you specify the desired new descriptor number. This allows precise control over descriptor numbers, useful in scenarios where managing specific descriptors or closing existing ones is necessary.

      Therefore, `dup()` is simpler and suitable for general duplication needs, while `dup2()` provides more control over descriptor numbers, depending on specific requirements.

   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)?

      There is no error is given code!!!

c. Write a program that executes "cat fixtures | grep | cut -b 1-9" command. A skeleton code for this is provided as exercise3.2.c_skel.c. You can use this as your starting point, if necessary.

Code:

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

#define INPUTFILE "fixtures"

/**
 * Executes the command "cat fixtures | grep <search_term> | cut -b 1-9".
 */

int main(int argc, char **argv)
{
    if (argc < 2)
    {
        fprintf(stderr, "Usage: %s <search_term>\n", argv[0]);
        return 1;
    }

    int status;
    int pipes[4];

    // arguments for commands
    char *cat_args[] = {"cat", INPUTFILE, NULL};
    char *grep_args[] = {"grep", "-i", argv[1], NULL};
    char *cut_args[] = {"cut", "-b", "1-9", NULL};

    // set up first pipe (cat to grep)
    if (pipe(pipes) == -1)
    {
        perror("pipe");
        return 1;
    }

    // set up second pipe (grep to cut)
    if (pipe(pipes + 2) == -1)
    {
        perror("pipe");
        return 1;
    }

    // fork the first child (to execute cat)
```

```c
if (fork() == 0)
{
    // replace cat's stdout with write part of 1st pipe
    dup2(pipes[1], 1);

    // close all pipes
    close(pipes[0]);
    close(pipes[1]);
    close(pipes[2]);
    close(pipes[3]);

    // execute cat command
    execvp("cat", cat_args);
    perror("execvp cat");
    return 1;
}
else
{
    // fork second child (to execute grep)
    if (fork() == 0)
    {
        // replace grep's stdin with read end of 1st pipe
        dup2(pipes[0], 0);

        // replace grep's stdout with write end of 2nd pipe
        dup2(pipes[3], 1);

        // close all pipes
        close(pipes[0]);
        close(pipes[1]);
        close(pipes[2]);
        close(pipes[3]);

        // execute grep command
        execvp("grep", grep_args);
        perror("execvp grep");
        return 1;
    }
    else
    {
        // fork third child (to execute cut)
        if (fork() == 0)
        {
            // replace cut's stdin with read end of 2nd pipe
            dup2(pipes[2], 0);
```

```
                // close all pipes
                close(pipes[0]);
                close(pipes[1]);
                close(pipes[2]);
                close(pipes[3]);

                // execute cut command
                execvp("cut", cut_args);
                perror("execvp cut");
                return 1;
            }
        }
    }

    // only the parent gets here and waits for 3 children to finish
    // close all pipes in the parent
    close(pipes[0]);
    close(pipes[1]);
    close(pipes[2]);
    close(pipes[3]);

    // wait for children
    for (int i = 0; i < 3; i++)
    {
        wait(&status);
    }

    return 0;
}
```

Output:

```
katta@katta:/media/sf_project/labs/lab02$ gcc ex3_2_c.c -o ex3
katta@katta:/media/sf_project/labs/lab02$ ./ex3 Sri
Feb 13: S
Feb 21: S
Feb 26: S
Feb 28: E
March 8:
March 11:
katta@katta:/media/sf_project/labs/lab02$
```

Exercise4.1

a. Comment out the line “mkfifo(fifo,0666);” in the reader and recompile the program. Test the programs by alternating which program is invoked first. Now, reset the reader to the original, comment the same line in the writer and repeat the test. What did you observe? Why do you think this happens? Explain how such an omission (i.e., leaving out mkfifo()function call in this case) can make debugging a nightmare.

   Commenting out `mkfifo(fifo, 0666);` in the reader and recompiling causes the reader to fail if run first, as it cannot find the named pipe. If the writer runs first, it creates and writes to the pipe, then wait for the reader, causing the reader to can read if run afterward.
   When `mkfifo(fifo, 0666);` is commented out in the writer instead, running the reader first creates the pipe and waits for input, allowing the writer to succeed when run next. However, if the writer runs first, it fails to find the pipe.
   This highlights the importance of `mkfifo`. Omitting it can cause unpredictable behavior based on execution order, making debugging difficult. Without creating the named pipe, processes may fail inconsistently, leading to confusion and harder-to-trace errors. Proper error handling and setup are essential to avoid these issues.

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.

   Parent Program
   Code:

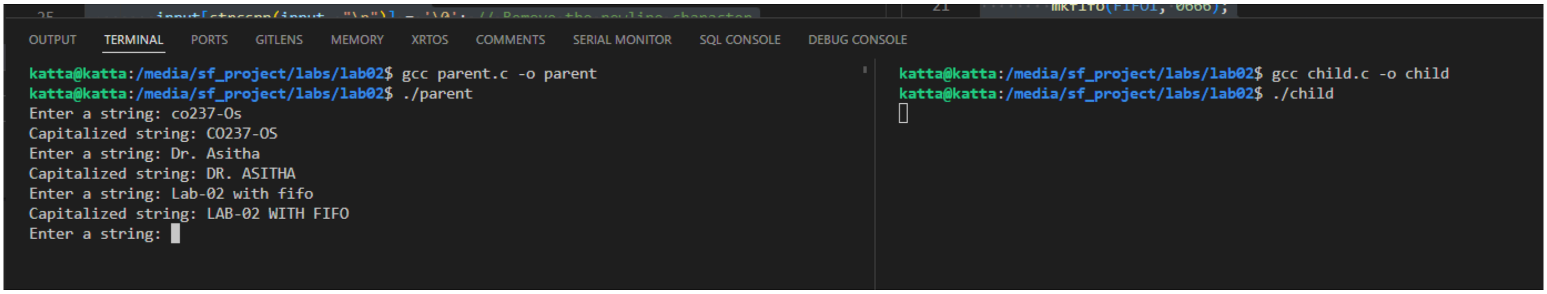
```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#define FIFO1 "/tmp/fifo1"
#define FIFO2 "/tmp/fifo2"
#define MAX_SIZE 1024

int main()
{
    int fd1, fd2;
    int len;
```

```c
    char input[MAX_SIZE];
    char output[MAX_SIZE];

    while (1)
    {
        // Get input from the user
        printf("Enter a string: ");
        fgets(input, MAX_SIZE, stdin);
        input[strcspn(input, "\n")] = '\0'; // Remove the newline character
        len = strlen(input);

        // Create the FIFOs
        mkfifo(FIFO1, 0666);
        // Send the input to the capitalizer program
        fd1 = open(FIFO1, O_WRONLY);
        write(fd1, input, strlen(input) + 1);
        close(fd1);

        // Create the FIFOs
        mkfifo(FIFO2, 0666);
        // Receive the capitalized string from the capitalizer program
        fd2 = open(FIFO2, O_RDONLY);
        read(fd2, output, MAX_SIZE);
        close(fd2);

        output[len] = '\0'; // Add the null terminator

        // Print the capitalized string
        printf("Capitalized string: %s\n", output);

        // Remove the FIFOs
        unlink(FIFO1);
        unlink(FIFO2);
    }

    return 0;
}
```

Child Program Code:

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

#define FIFO1 "/tmp/fifo1"
#define FIFO2 "/tmp/fifo2"
#define MAX_SIZE 1024

int main()
{
    int fd1, fd2;
    char input[MAX_SIZE];
    char output[MAX_SIZE];

    while (1)
    { // Open the FIFO for reading the input
        mkfifo(FIFO1, 0666);
        fd1 = open(FIFO1, O_RDONLY);
        read(fd1, input, MAX_SIZE);
        close(fd1);

        // Open the FIFO for writing the output
        mkfifo(FIFO2, 0666);
        fd2 = open(FIFO2, O_WRONLY);

        // replace standard output with output file
        dup2(fd2, 1);

        // Capitalize the input string using the tr command
        // Send the capitalized string back to the writer program
        FILE *tr_pipe = popen("tr '[a-z]' '[A-Z]'", "w");
        fwrite(input, sizeof(char), strlen(input), tr_pipe);
        pclose(tr_pipe);

        close(fd2);
    }

    return 0;
}
```

Output:


OUTPUT
TERMINAL
PORTS
GITLENS
MEMORY
XRTOS
COMMENTS
SERIAL MONITOR
SQL CONSOLE
DEBUG CONSOLE
katta@katta:/media/sf_project/labs/lab02$ gcc parent.c -o parent
katta@katta:/media/sf_project/labs/lab02$ ./parent
Enter a string: co237-Os
Capitalized string: CO237-OS
Enter a string: Dr. Asitha
Capitalized string: DR. ASITHA
Enter a string: Lab-02 with fifo
Capitalized string: LAB-02 WITH FIFO
Enter a string:
katta@katta:/media/sf_project/labs/lab02$ gcc child.c -o child
katta@katta:/media/sf_project/labs/lab02$ ./child