National University of Computer and Emerging Sciences

Lab Manual 5
(Operating Systems)

Department of Computer Science
FAST-NU, Lahore

Pipes (Anonymous Pipes)

Ordinary/anonymous 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). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction.

A pipe has a read end and a write end. Data written to the write end of a pipe can be read from the read end of the pipe.

Creating a Pipe

On UNIX and Linux systems, ordinary pipes are constructed using the function

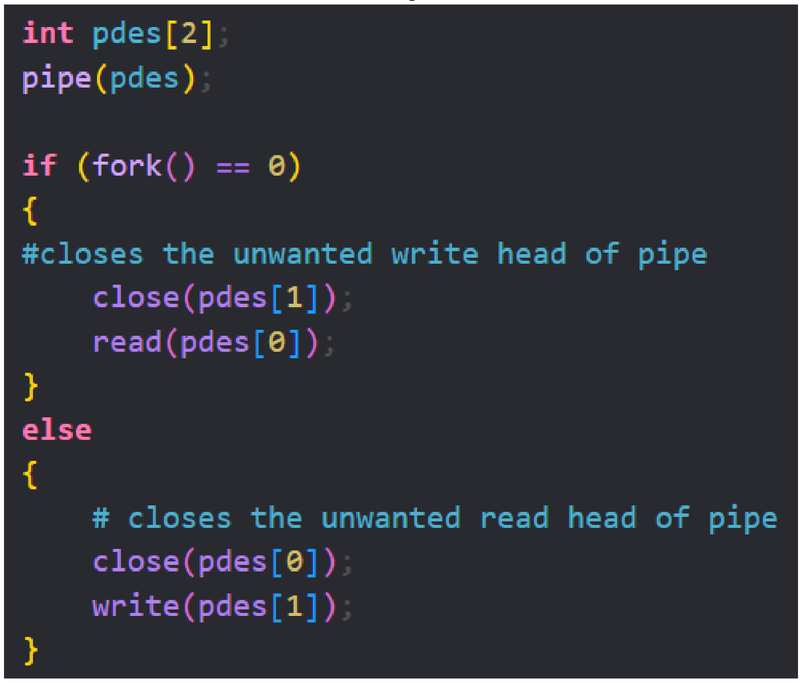
- int pipe(int fd[2]) -- creates a pipe
- returns two file descriptors, fd[0], fd[1].
- fd[0] is the read-end of the pipe
- fd[1] is the write-end.
- fd[0] is opened for reading,
- fd[1] for writing.
- pipe() returns 0 on success, -1 on failure.

• The standard programming model is that after the pipe has been set up, two (or more) cooperative processes will be created by a fork and data will be passed using read() and write().
• Pipes opened with pipe() should be closed with close(int fd).

When pipe() System Call Fails:

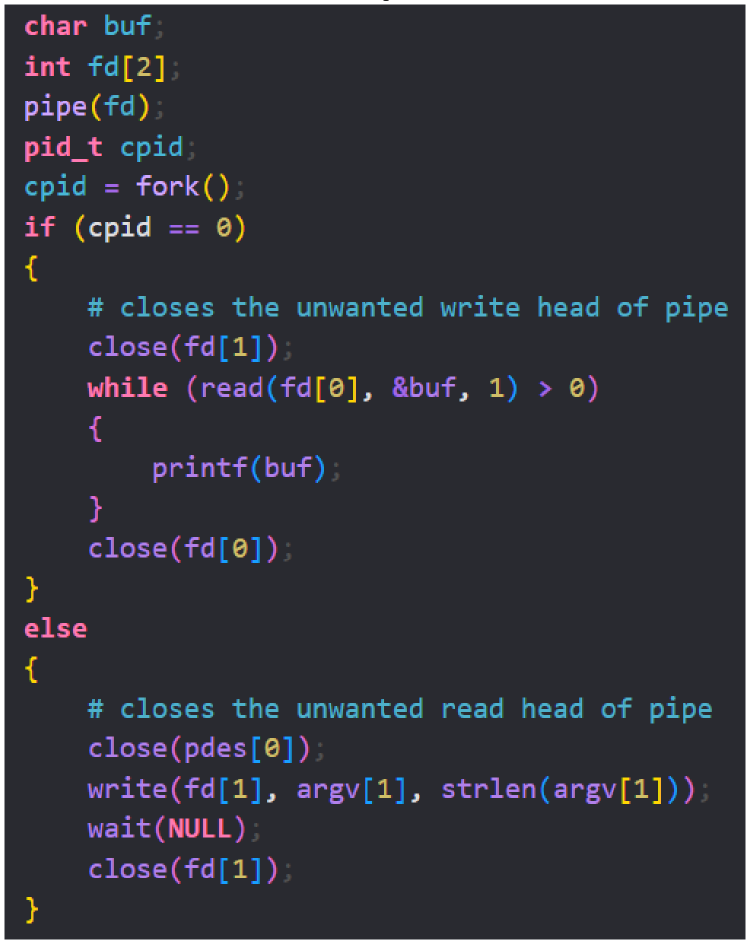
The pipe() system call fails for many reasons, including the following:
1 At least two slots are not empty in the FDT—too many files or pipes are open in the process.
2 Buffer space not available in the kernel

Example 1



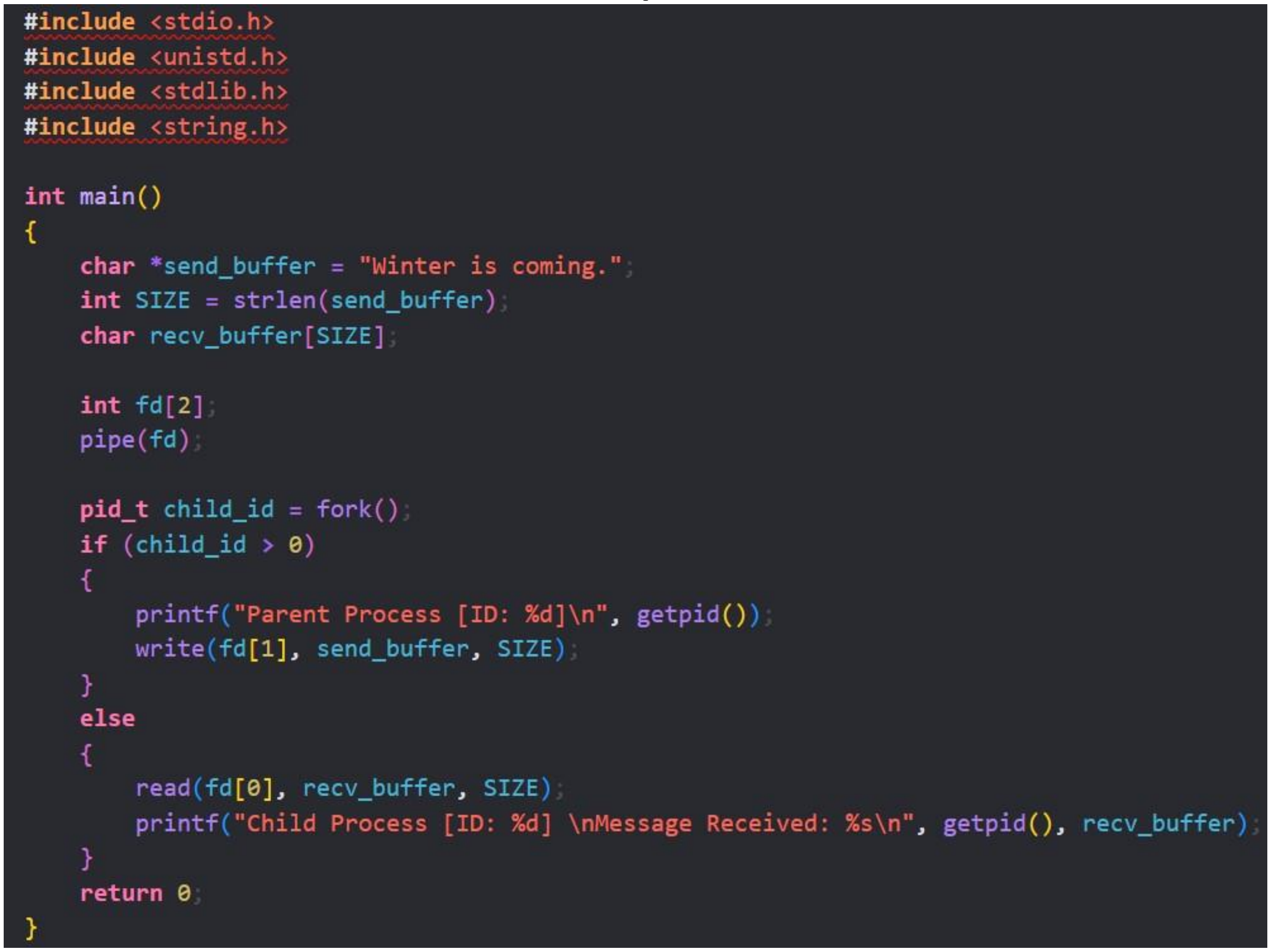
```
int pdes[2];
pipe(pdes);

if (fork() == 0)
{
#closes the unwanted write head of pipe
    close(pdes[1]);
    read(pdes[0]);
}
else
{
    # closes the unwanted read head of pipe
    close(pdes[0]);
    write(pdes[1]);
}
```

Example 2



```
char buf;
int fd[2];
pipe(fd);
pid_t cpid;
cpid = fork();
if (cpid == 0)
{
    # closes the unwanted write head of pipe
    close(fd[1]);
    while (read(fd[0], &buf, 1) > 0)
    {
        printf(buf);
    }
    close(fd[0]);
}
else
{
    # closes the unwanted read head of pipe
    close(pdes[0]);
    write(fd[1], argv[1], strlen(argv[1]));
    wait(NULL);
    close(fd[1]);
}
```

Example 3



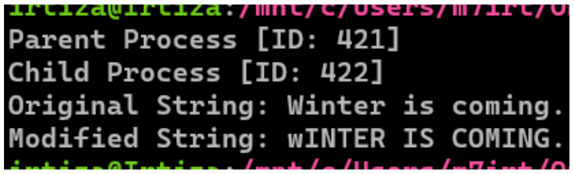
```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char *send_buffer = "Winter is coming.";
    int SIZE = strlen(send_buffer);
    char recv_buffer[SIZE];

    int fd[2];
    pipe(fd);

    pid_t child_id = fork();
    if (child_id > 0)
    {
        printf("Parent Process [ID: %d]\n", getpid());
        write(fd[1], send_buffer, SIZE);
    }
    else
    {
        read(fd[0], recv_buffer, SIZE);
        printf("Child Process [ID: %d] \nMessage Received: %s\n", getpid(), recv_buffer);
    }
    return 0;
}
```

Task 1 (2 marks)

Design a program using ordinary pipes in which one process (parent) sends a string message to a second process (child), and the second process reverses the case of each character in the message and prints it on the screen.

For example, if the first process sends the message **Hi There**, the second process will print **hi tHERE**.

Output:



```
Parent Process [ID: 421]
Child Process [ID: 422]
Original String: Winter is coming.
Modified String: wINTER IS COMING.
```

Task 2 (3 marks)

Design a program using ordinary pipes in which one process send an integer array to its child and child calculates its sum and send it back to the parent and parent displays the result.

This will require using two pipes, one for sending the original message from the first to the second process, and the other for sending the modified message from the second process back to the first process.

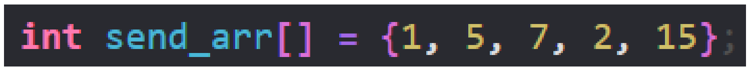
Help:

Use int size = sizeof(int) * num_of_elements_in_array;

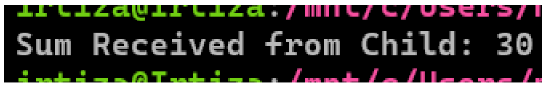
Syntax:

write(fd[1], send_array, size);
read(fd[0], recv_array, size);

Output:



```
int send_arr[] = {1, 5, 7, 2, 15};
```



```
Sum Received from Child: 30
```

Task 3 (5 marks)

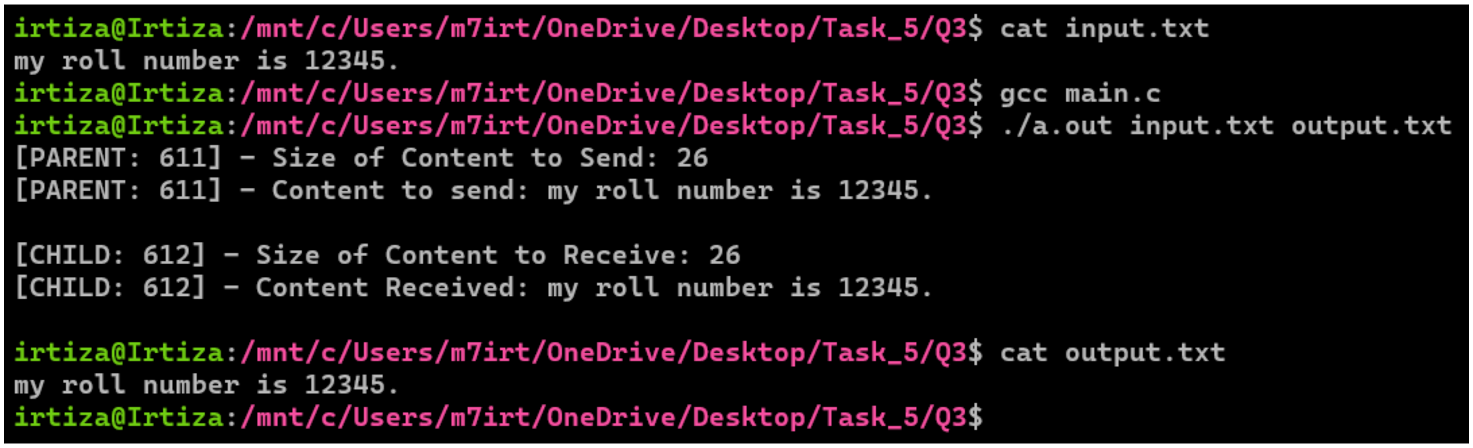
Design a file-copying program named FileCopy using ordinary pipes. This program will be passed two parameters: the first is the name of the file to be copied, and the second is the name of the copied file. The program will then create an ordinary pipe and write the contents of the file to be copied to the pipe. The child process will read this file from the pipe and write it to the destination file. For example, if we invoke the program as follows:

- ➔ gcc FileCopy.c –o FileCopy
- ➔ FileCopy input.txt output.txt

The file **input.txt** will be written to the pipe. The child process will read the contents of this file and write it to the destination file copy.txt.

Help:
You will have to send the size of the read content through pipe to the parent before sending the actual content.
To get the size of string use **strlen(content)**
To get the size of int use **sizeof(int)**

Output:



```
irtiza@Irtiza:/mnt/c/Users/m7irt/OneDrive/Desktop/Task_5/Q3$ cat input.txt
my roll number is 12345.
irtiza@Irtiza:/mnt/c/Users/m7irt/OneDrive/Desktop/Task_5/Q3$ gcc main.c
irtiza@Irtiza:/mnt/c/Users/m7irt/OneDrive/Desktop/Task_5/Q3$ ./a.out input.txt output.txt
[PARENT: 611] - Size of Content to Send: 26
[PARENT: 611] - Content to send: my roll number is 12345.

[CHILD: 612] - Size of Content to Receive: 26
[CHILD: 612] - Content Received: my roll number is 12345.

irtiza@Irtiza:/mnt/c/Users/m7irt/OneDrive/Desktop/Task_5/Q3$ cat output.txt
my roll number is 12345.
irtiza@Irtiza:/mnt/c/Users/m7irt/OneDrive/Desktop/Task_5/Q3$
```