Xi'an Jiaotong-Liverpool University 西交利物浦大學

CPT104 - Operating Systems Concepts

Lab 8

Inter Process Communication

Pipe()

Linux, provides a rich set of mechanisms for inter-process communication (IPC), including the following:

- **signals**, which are used to indicate that an event has occurred.
- **pipes**, which can be used to transfer data between processes.
- **sockets**, which can be used to transfer data from one process to another, either on the same host computer or on different hosts connected by a network.
- **file locking**, which allows a process to lock regions of a file in order to prevent other processes from reading or updating the file contents.
- **message queues**, which are used to exchange messages (packets of data) between processes.
- **semaphores**, which are used to synchronize the actions of processes.
- **shared memory**, which allows two or more processes to share a piece of memory.

Each process has a number of file descriptors associated with it. These are **small integers** that we can use to access open files or devices. How many of these are available will vary depending on how the system has been configured. When a program starts, it usually has three of these descriptors already opened. These are:

❑ **0: Standard input**

❑ **1: Standard output**

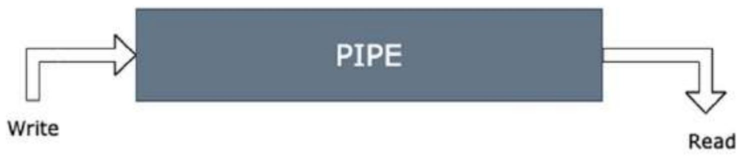
❑ **2: Standard error**

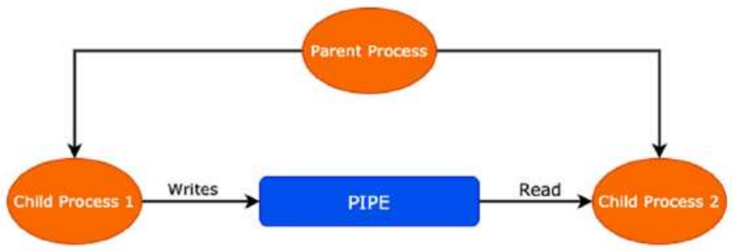
A pipe is a mechanism for interprocess communication; data written to the pipe by one process can be read by another process. The data is handled in a **first-in, first-out (FIFO)** order.

The pipe has no name; it is created for one use and both ends must be inherited from the single process which created the pipe.

A pipe has to be open at both ends simultaneously. If you read from a pipe file that doesn't have any processes writing to it, the read returns end-of-file.

Writing to a pipe that doesn't have a reading process is treated as an error condition.





Communication is achieved by one process **writing into the pipe** and other **reading from the pipe**. This system call would create a pipe for one-way communication.

When *Child Process1* writes data to the pipe, it is read by another process using a file descriptor. When data is written to the main memory, the pipe treats that data as a virtual file. Accessing that data is done with a file descriptor. The write and read operations in files are done with two standard file descriptors.

To achieve the pipe system call, **create two files**, **one to write into** the file and **another to read from** the file. The pipe function has the following prototype:

```
#include <unistd.h>

int pipe(int pipedes[2]);                    Returns: 0 if OK, −1 on error
```

pipedes takes an **integer descriptor array of size 2**, which performs the read and write operations.

pipedes[1] writes the content into the pipe, and

pipedes[0] reads the content from the pipe.

Any data written to pipedes[1]can be read back from pipedes[0].

Since pipes follow a unidirectional flow of data transfer, the read operation needs to be done after the write action is performed.

write() ssize_t write(int fd, const void *buf, size_t nbytes);

The **write()** system call writes the content to a specific file with certain arguments of that file descriptor. If successful, it returns the number of bytes written to the file; otherwise, it returns −1.

read() ssize_t read(int fd, void *buf, size_t nbytes);

The **read()** system call reads the content from the file descriptor or pipe. It returns the number of bytes read from the file descriptor and returns −1 if any failure occurs.

close() int close(int fd);

The **close()** system call closes the opened descriptors. The return type of this system call is an integer. It returns 0 if successful and −1 if any failure occurs.

Modes of Communication

Simplex

In a simplex mode of data transmission, data is transferred in a unidirectional way, which means that only one process, one person, or one device can send data that others receive.

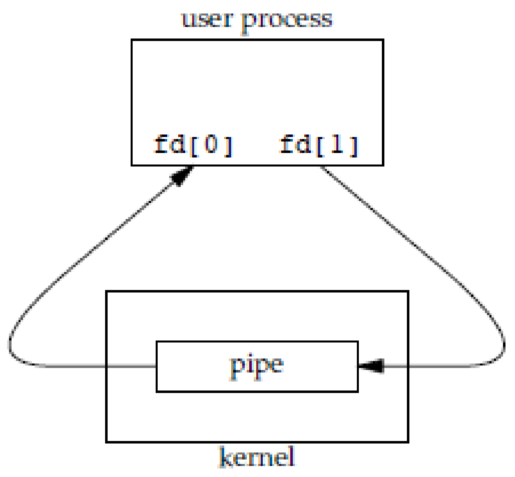
Half Duplex

In a half-duplex mode of data transmission, data is transmitted bidirectionally but not at the same time. In this mechanism, a process, or a person, or a device has access to send and receive data but not at the same time. When one process is sending, the other processes must listen or receive.

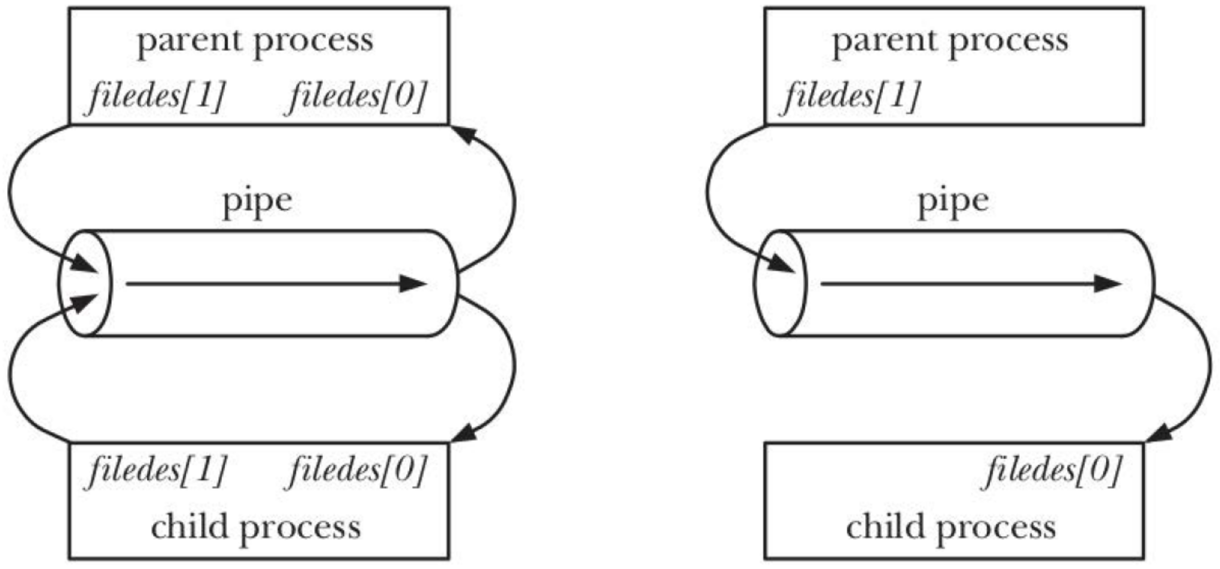
Full Duplex

In a full-duplex mode of data transmission, data is transferred in a bidirectional way between two processes, or two persons, or two devices.

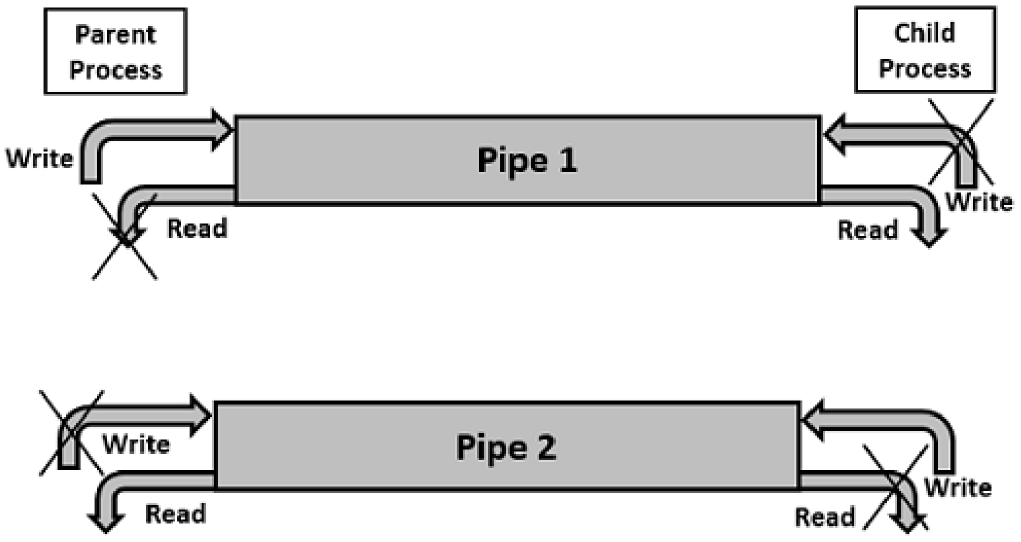
The unused ends of a pipe are usually closed before starting to use a pipe. There are also legitimate reasons for closing the used ends, e.g. when one process wants to shut down the communication.



Data in the pipe flows through the kernel（single process）

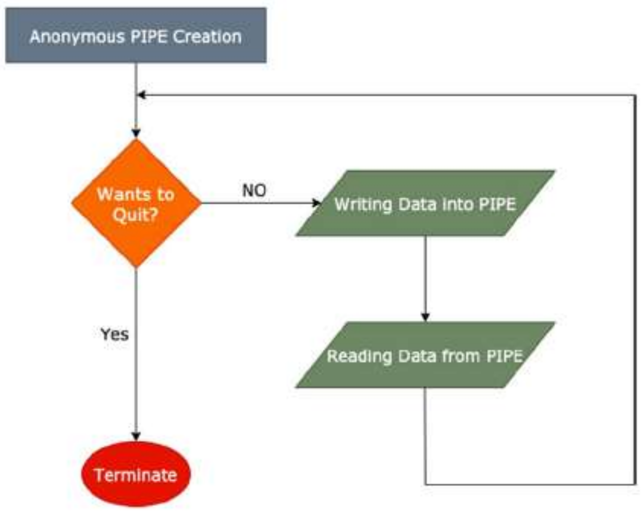


a) After *fork()*　　b) After closing unused descriptors

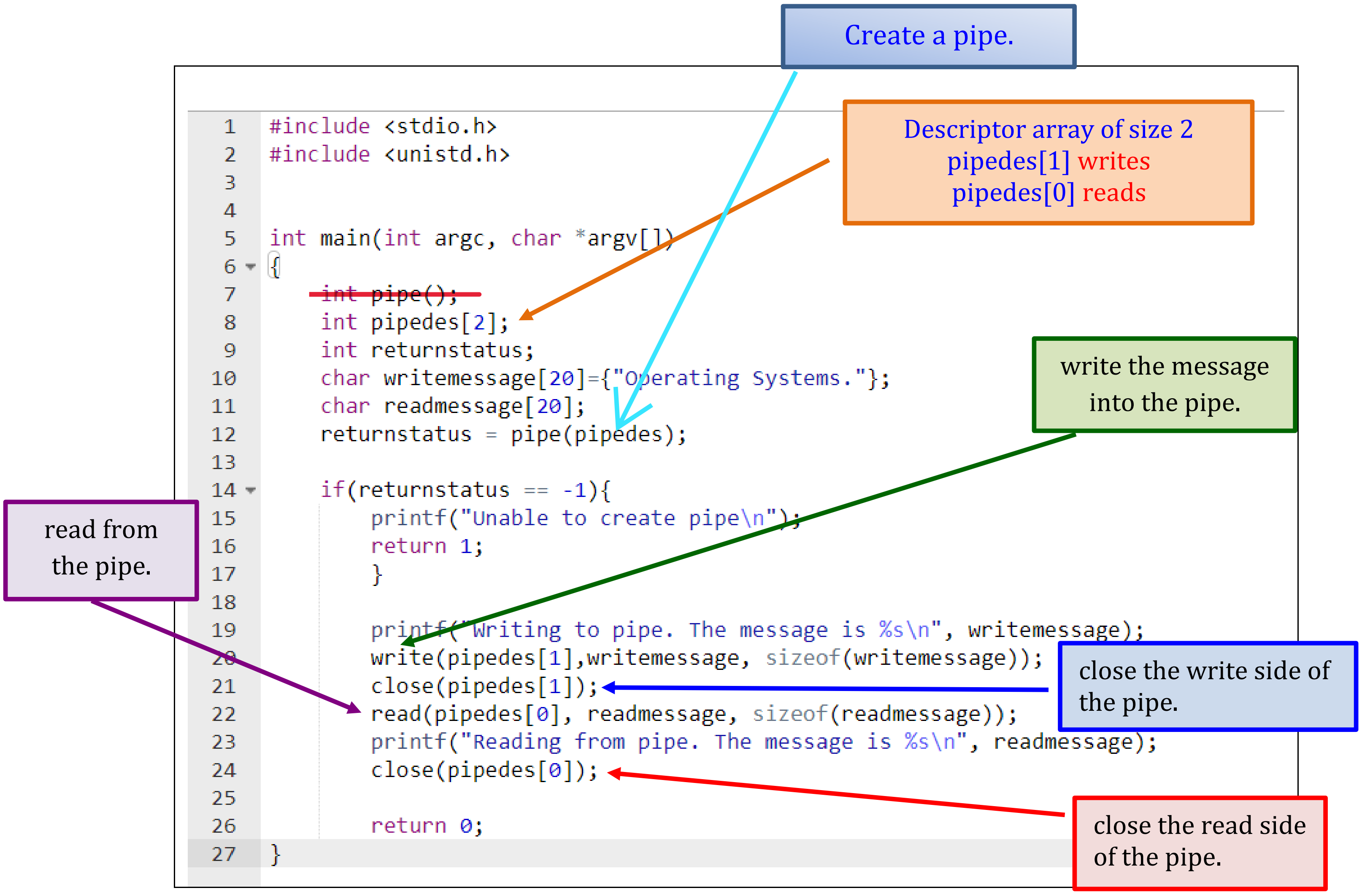


Two-way Communication Using Pipes

https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_pipes.htm

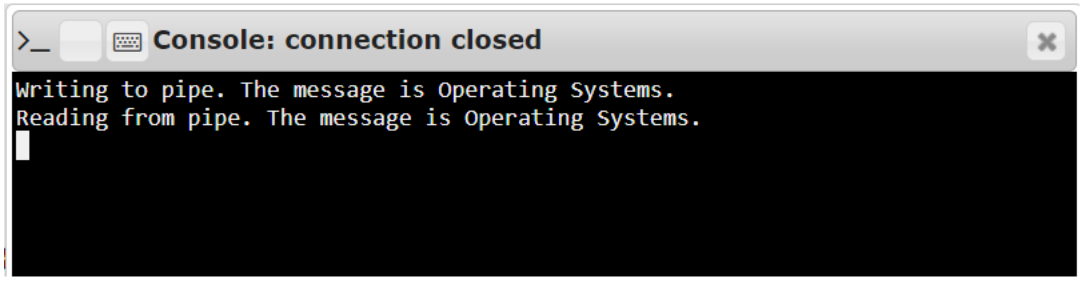
Algorithm	
Step 1 – Create a pipe using the *pipe()* system call. **Step 2**. If the user quits, the program terminates; otherwise, go to step 3. **Step 3**. Write the data into the pipe using a *write()* system call. **Step 4**. Once the write operation is done, the *read()* system call reads the data from the pipe. **Step 5**. Steps 3 and 4 repeat until the user quits the program.	 Anonymous PIPE Creation Wants to Quit? NO Writing Data into PIPE Reading Data from PIPE Yes Terminate

Example: Program to write and read a message using pipe.


Create a pipe.
Descriptor array of size 2
pipedes[1] writes
pipedes[0] reads
write the message into the pipe.
read from the pipe.
close the write side of the pipe.
close the read side of the pipe.

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


int main(int argc, char *argv[])
{
    ~~int pipe();~~
    int pipedes[2];
    int returnstatus;
    char writemessage[20]={"Operating Systems."};
    char readmessage[20];
    returnstatus = pipe(pipedes);

    if(returnstatus == -1){
        printf("Unable to create pipe\n");
        return 1;
        }

        printf("Writing to pipe. The message is %s\n", writemessage);
        write(pipedes[1],writemessage, sizeof(writemessage));
        close(pipedes[1]);
        read(pipedes[0], readmessage, sizeof(readmessage));
        printf("Reading from pipe. The message is %s\n", readmessage);
        close(pipedes[0]);

        return 0;
}
```

Output



```
Writing to pipe. The message is Operating Systems.
Reading from pipe. The message is Operating Systems.
```

Implementation of Pipes Using Child and Parent Processes

simplex mode of data transmission

Algorithm

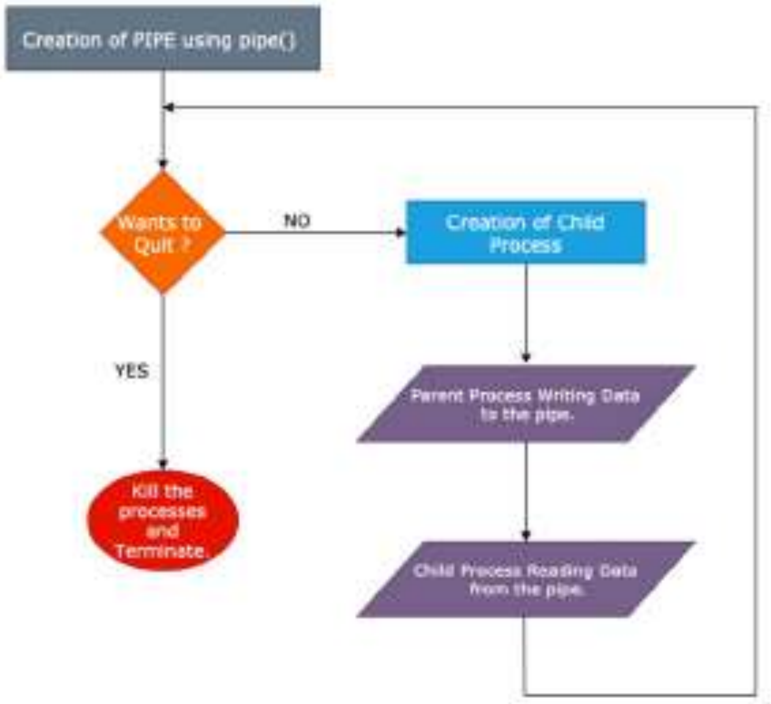
Step 1 – Create a pipe.

Step 2 – If the user quits, the program terminates; otherwise, create a child process.

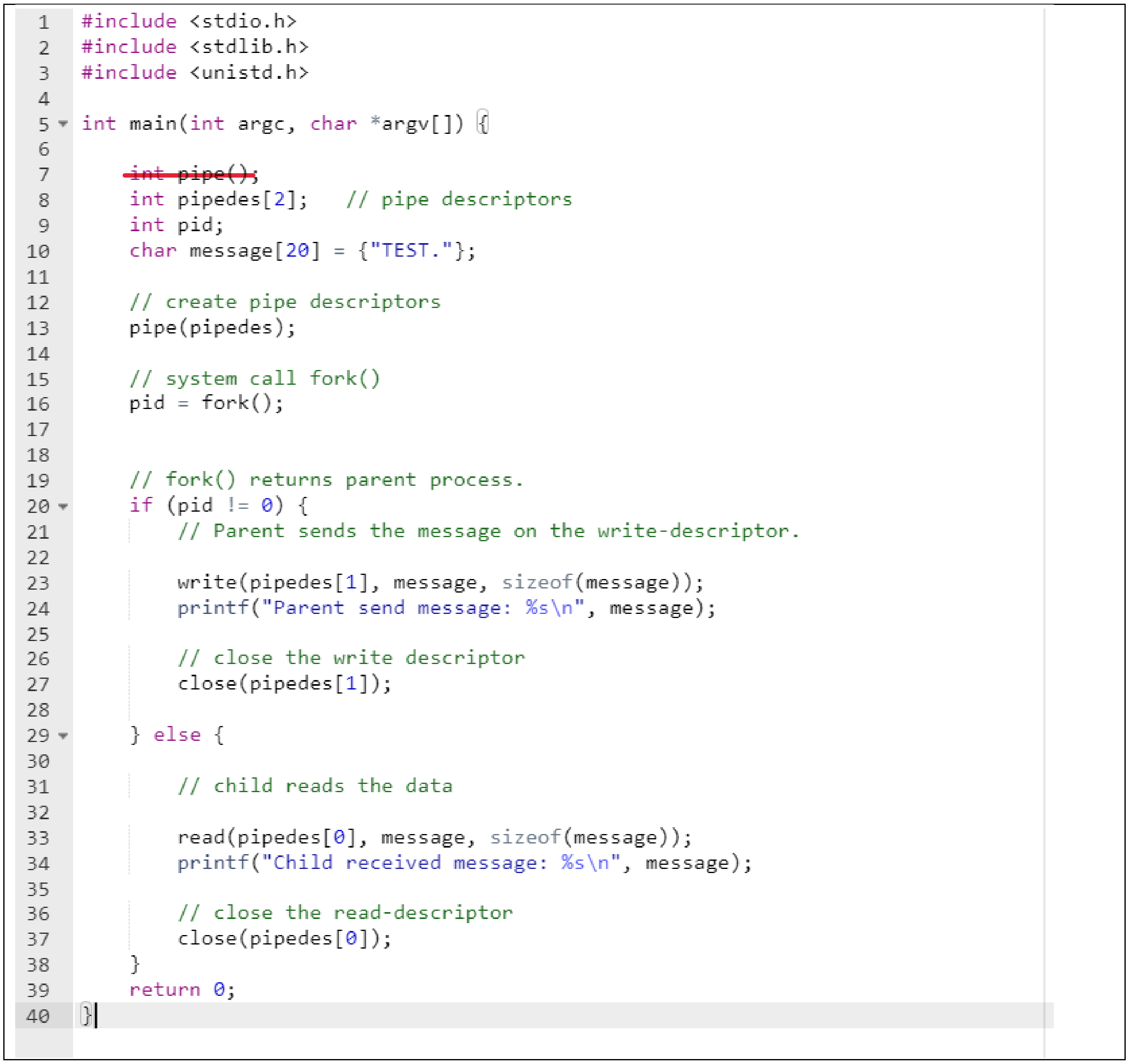
Step 3 – Write the data into the pipe using the parent process.

Step 4 – Read the data from the pipe using the child process.

Step 5 – Steps 3 and 4 repeat until the user quits.

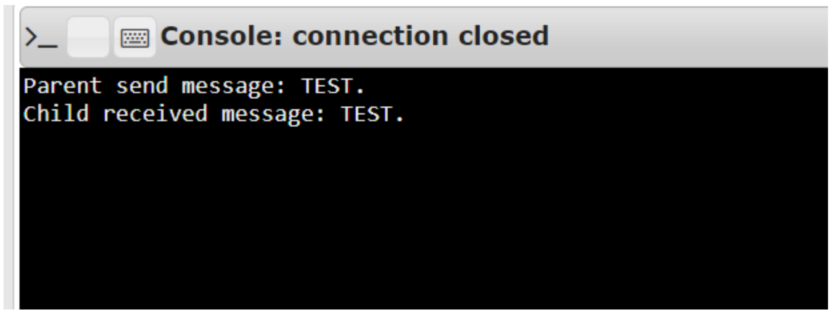

Creation of PIPE using pipe()
Wants to Quit ?
NO
Creation of Child Process
YES
Parent Process Writing Data to the pipe.
Kill the processes and Terminate.
Child Process Reading Data from the pipe.

Example 2: Program to write and read the message **TEST** through the pipe using *the parent and the child processes.* (one way communication: **Parent to Child**)



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

int main(int argc, char *argv[]) {

    ~~int pipe();~~
    int pipedes[2];    // pipe descriptors
    int pid;
    char message[20] = {"TEST."};

    // create pipe descriptors
    pipe(pipedes);

    // system call fork()
    pid = fork();


    // fork() returns parent process.
    if (pid != 0) {
        // Parent sends the message on the write-descriptor.

        write(pipedes[1], message, sizeof(message));
        printf("Parent send message: %s\n", message);

        // close the write descriptor
        close(pipedes[1]);

    } else {

        // child reads the data

        read(pipedes[0], message, sizeof(message));
        printf("Child received message: %s\n", message);

        // close the read-descriptor
        close(pipedes[0]);
    }
    return 0;
}
```

Output



```
Parent send message: TEST.
Child received message: TEST.
```

We now continue with the **Lab Exercise** (see LMO)

● Just like the Lab Example, you will write and test your code;

● You will also need to submit your code using the LMO VPL.

Reference:
For more information: refer to book 15.2