Inter process communication (IPC)

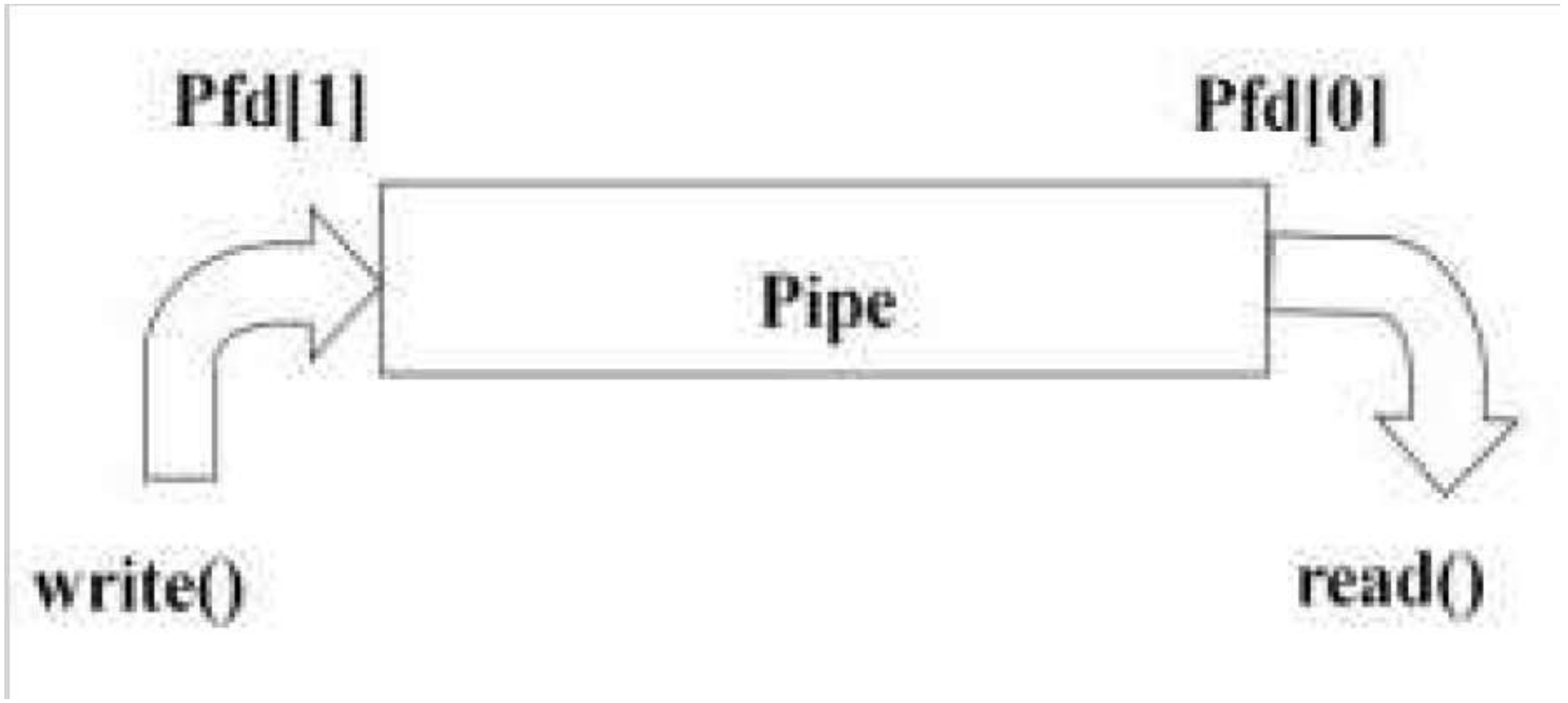
- Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions.
- The communication between these processes can be seen as a method of co-operation between them.
- Processes can communicate with each other using these two ways:

1. Shared Memory

2. Message passing

pipe() System call

- a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process.
- In UNIX Operating System, Pipes are useful for communication between related processes(inter-process communication).


Pfd[1]
Pfd[0]
Pipe
write()
read()

- It is not very useful for a single process to use a pipe to talk to itself.
- In typical use, a process creates a pipe just before it forks one or more child processes.
- The pipe is then used for communication either between the parent or child processes, or between two sibling processes.

System Calls

- Read ()
- Write()
- Close()
- Pipe()
- Dup(), Dup2()

Pipe Creation (System Call)

```
#include <unistd.h>
int pipe(int filedes[2]);
```

Creates a pair of file descriptors pointing to a pipe node

Places them in the array pointed to by filedes

filedes[0] is for reading

filedes[1] is for writing.

On success, zero is returned.

On error, -1 is returned

Interprocess Communication using Pipes:

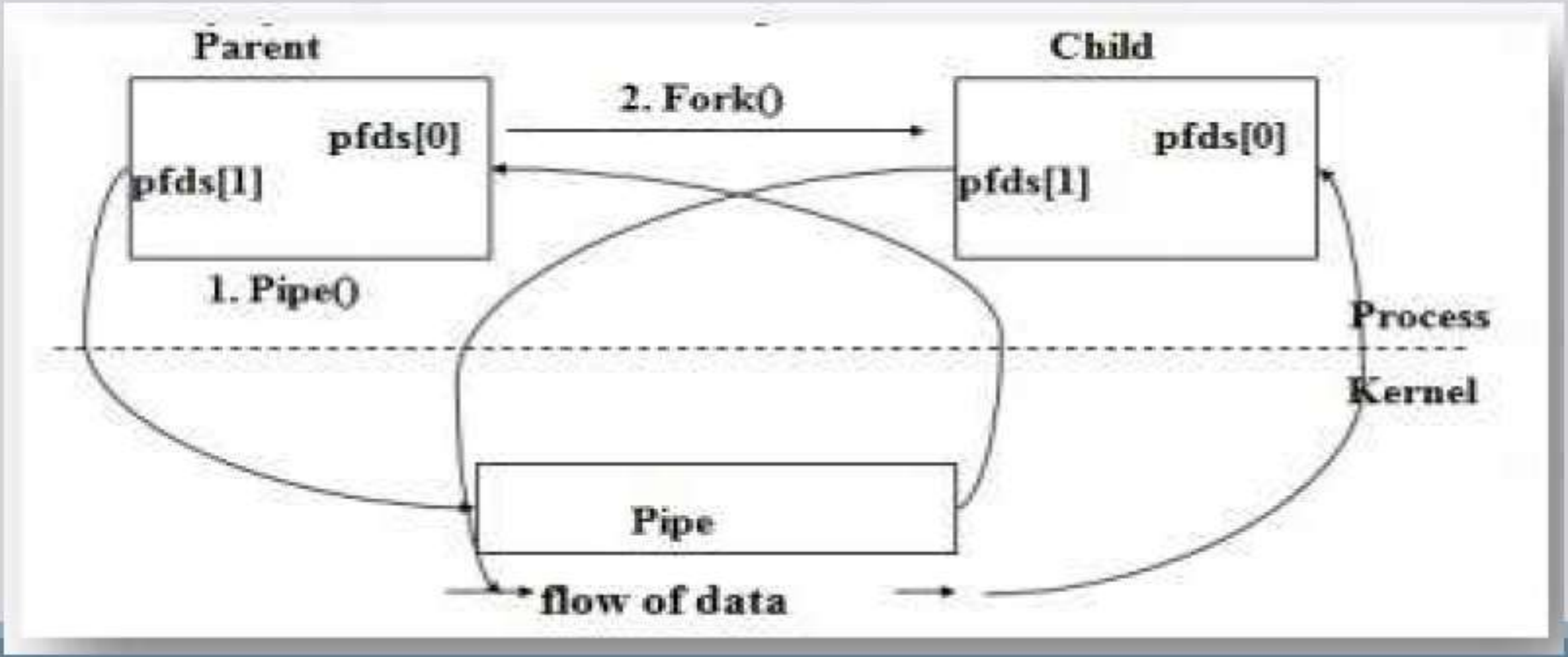
Remember: the two processes have a parent / child relationship.

The child was created by a fork() call that was executed by the parent.

The child process is an image of the parent process.

Thus, all the file descriptors that are opened by the parent are now available in the child

The file descriptors refer to the same I/O entity, in this case a pipe. The pipe is inherited by the child.



Read() & Write() (System Calls)

Read():

The read system call reads up to nbytes bytes of data from the file associated with the file descriptor fildes

Places them in the data area buf.

Returns the number of data bytes actually read, which may be less than the number requested.

If a read call returns 0, it had nothing to read; it reached the end of the file.

Again, an error on the call will cause it to return – 1.

- **#include <unistd.h>**
- **size_t read(int fildes, void *buf, size_t nbytes);**

Read() & Write() (System Calls)

Write()

The write system call arranges for the first nbytes bytes from buf to be written to the file associated with the file descriptor fildes.

Returns the number of bytes actually written.

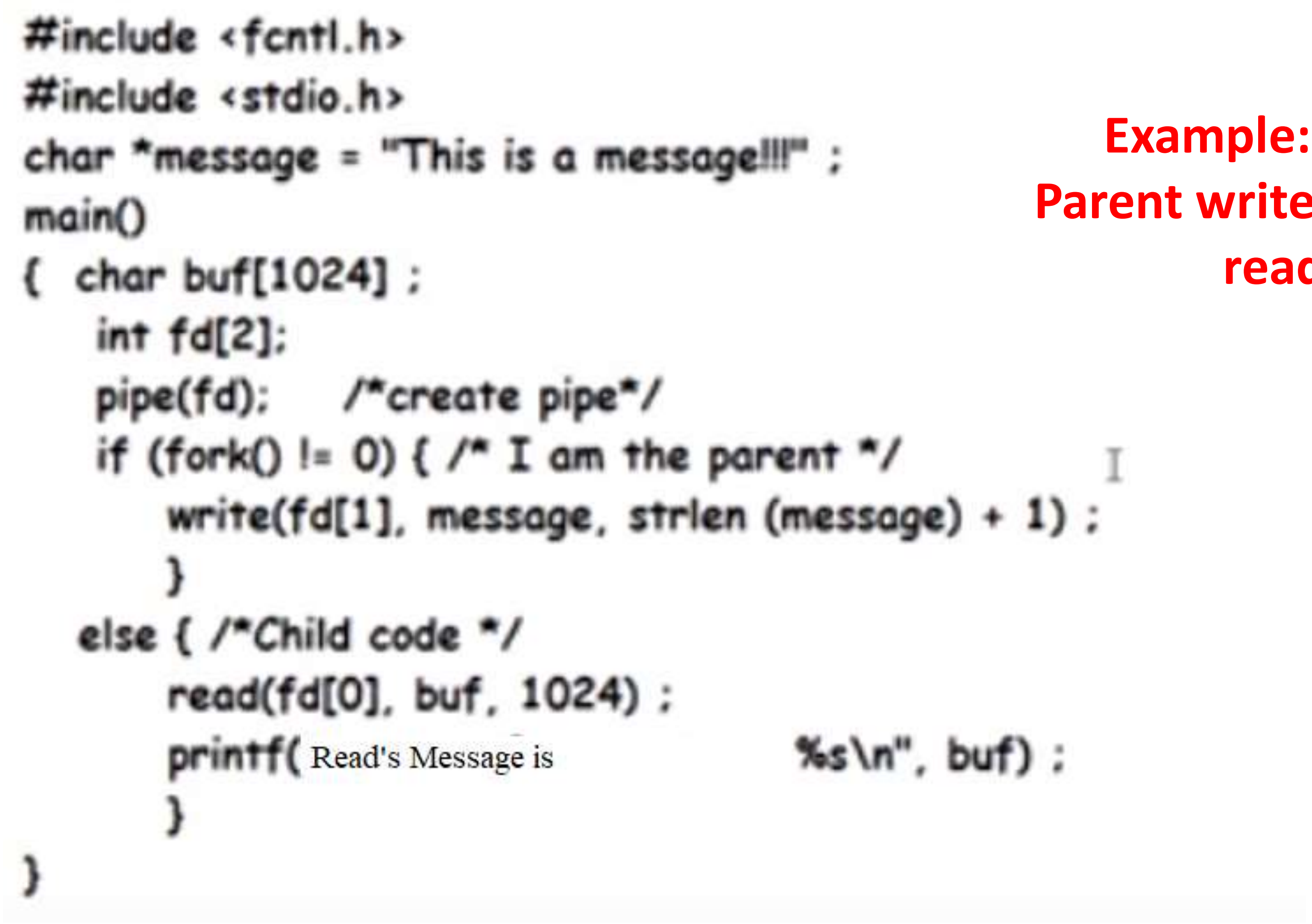
This may be less than nbytes if there has been an error in the file descriptor or if the underlying device driver is sensitive to block size.

If the function returns 0, it means no data was written; if it returns -1, there has been an error in the write call, and the error will be specified in the errno global variable.

- #include <unistd.h>
- size_t write(int fildes, const void *buf, size_t nbytes);

3. Close()

close() closes the file indicated by the file descriptor fildes.



```
#include <fcntl.h>
#include <stdio.h>
char *message = "This is a message!!!" ;
main()
{  char buf[1024] ;
    int fd[2];
    pipe(fd);     /*create pipe*/
    if (fork() != 0) { /* I am the parent */
        write(fd[1], message, strlen (message) + 1) ;
        }
   else { /*Child code */
        read(fd[0], buf, 1024) ;
        printf( Read's Message is              %s\n", buf) ;
        }
}
```

Example:- Pipe concept …!

Parent writes into pipe and child reads from pipe

Named pipe

In computing, a named pipe (also known as a **FIFO**) is one of the methods for inter-process communication.

- It is an extension to the traditional pipe concept on Unix. A traditional pipe is “unnamed” and lasts only as long as the process.
- A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.

- Usually a named pipe appears as a file, and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.
- A FIFO special file is entered into the filesystem by calling mkfifo() in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

Creating a FIFO File

The easiest way to create a FIFO file is to use the *mkfifo* command.

This command is part of the standard Linux utilities and can simply be typed at the command prompt of your shell.

You may also use the *mknod* command to accomplish the same thing. This command requires an additional argument but otherwise it works pretty much the same.

The following example shows one way to use the mkfifo command:

```
prompt> mkfifo /tmp/myFIFO
```

There is also a system call that allows you to do this so you can put it in a program.

- `int mkfifo(const char *pathname, mode_t mode);`

mkfifo() makes a FIFO special file with name ***pathname***. Here ***mode*** specifies the FIFO’s permissions. It is modified by the process’s umask in the usual way: the permissions of the created file are (mode & ~umask).

Named Pipes (System Calls)

int open(const char *pathname, int flags);

Given a ***pathname*** for a file, returns a file descriptor, The argument ***flags*** must include one of the following access modes:

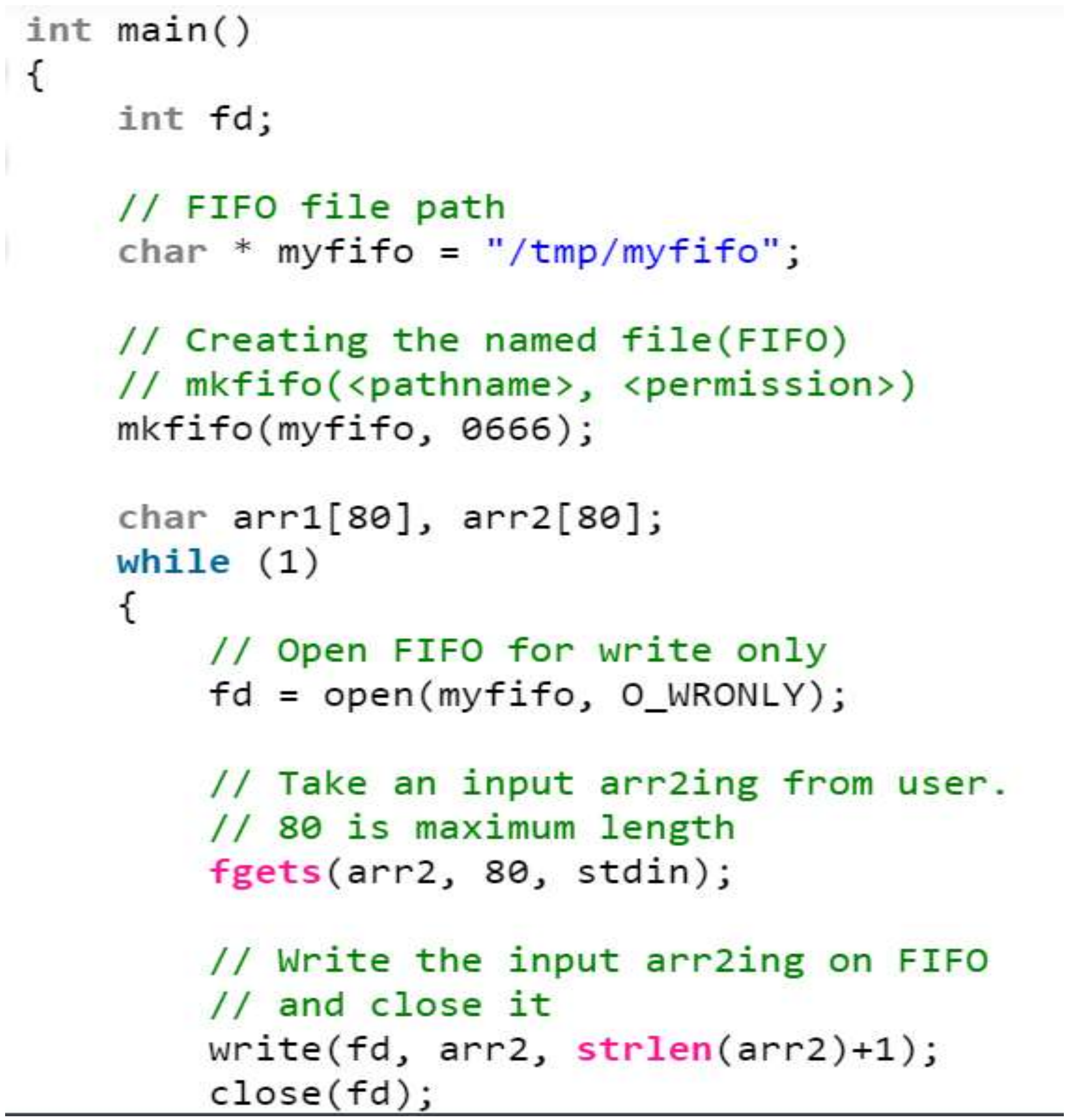
O_RDONLY, O_WRONLY, or O_RDWR.

int read(int fd, void *buf, size_t count);

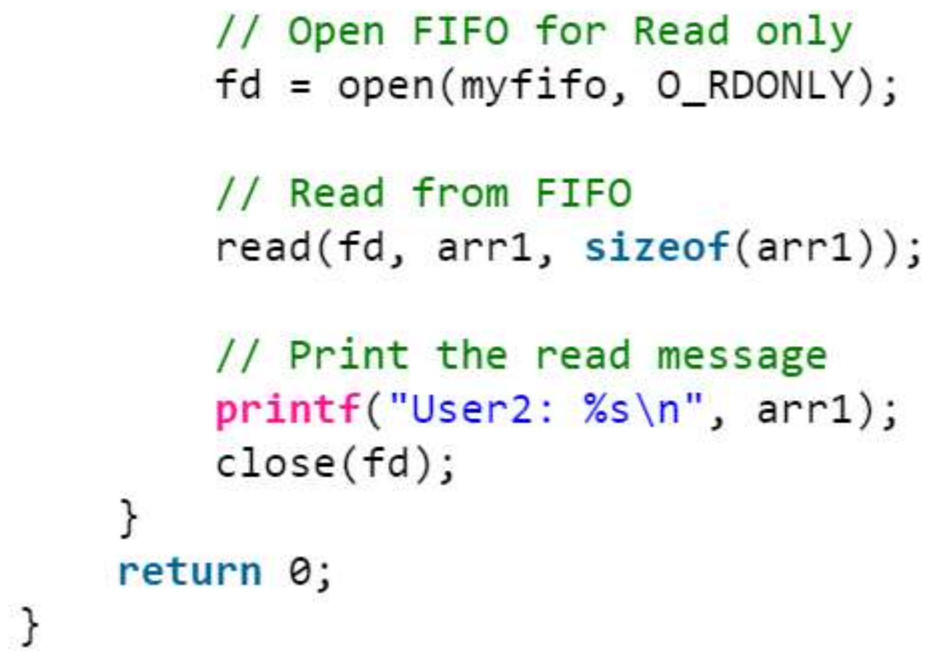
int write(int fd, const void *buf, size_t count);

int close(fd);

Process 1



```
int main()
{
    int fd;

    // FIFO file path
    char * myfifo = "/tmp/myfifo";

    // 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);
```

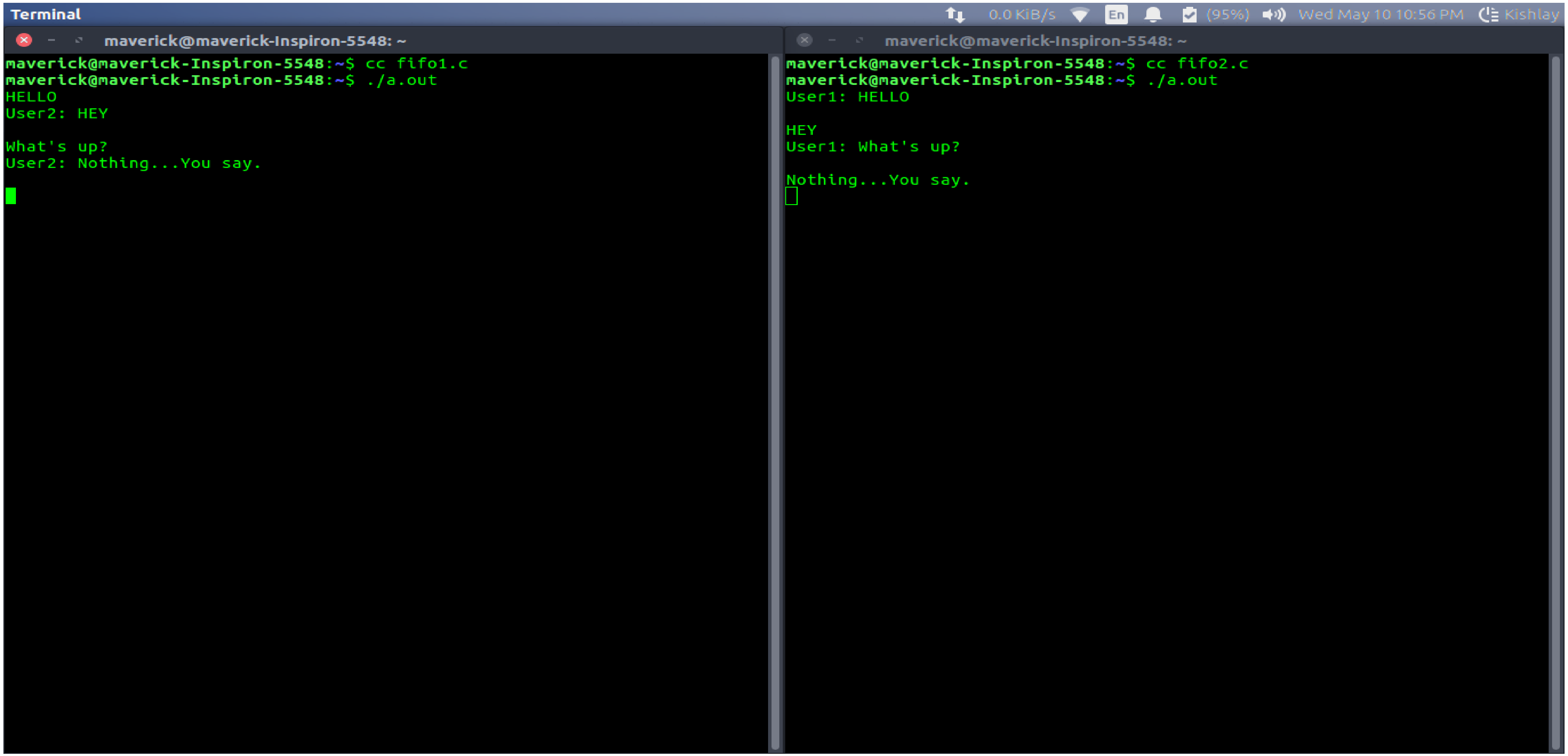


```
        // Open FIFO for Read only
        fd = open(myfifo, O_RDONLY);

        // Read from FIFO
        read(fd, arr1, sizeof(arr1));

        // Print the read message
        printf("User2: %s\n", arr1);
        close(fd);
    }
    return 0;
}
```

Process 2

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

int main()
{
    int fd1;

    // FIFO file path
    char * myfifo = "/tmp/myfifo";

    // 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);
    }
    return 0;
```

Output


Terminal
0.0 KiB/s
En
(95%)
Wed May 10 10:56 PM
Kishlay
maverick@maverick-Inspiron-5548: ~
maverick@maverick-Inspiron-5548: ~

```
maverick@maverick-Inspiron-5548:~$ cc fifo1.c
maverick@maverick-Inspiron-5548:~$ ./a.out
HELLO
User2: HEY

What's up?
User2: Nothing...You say.
```

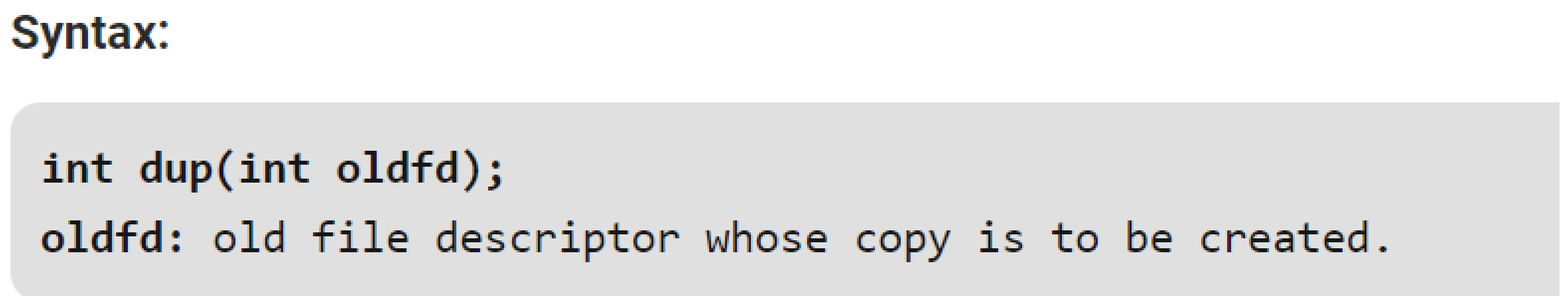
```
maverick@maverick-Inspiron-5548:~$ cc fifo2.c
maverick@maverick-Inspiron-5548:~$ ./a.out
User1: HELLO

HEY
User1: What's up?

Nothing...You say.
```

dup()

The dup() system call creates a copy of a file descriptor.

• It uses the lowest-numbered unused descriptor for the new descriptor.

• If the copy is successfully created, then the original and copy file descriptors may be used interchangeably.

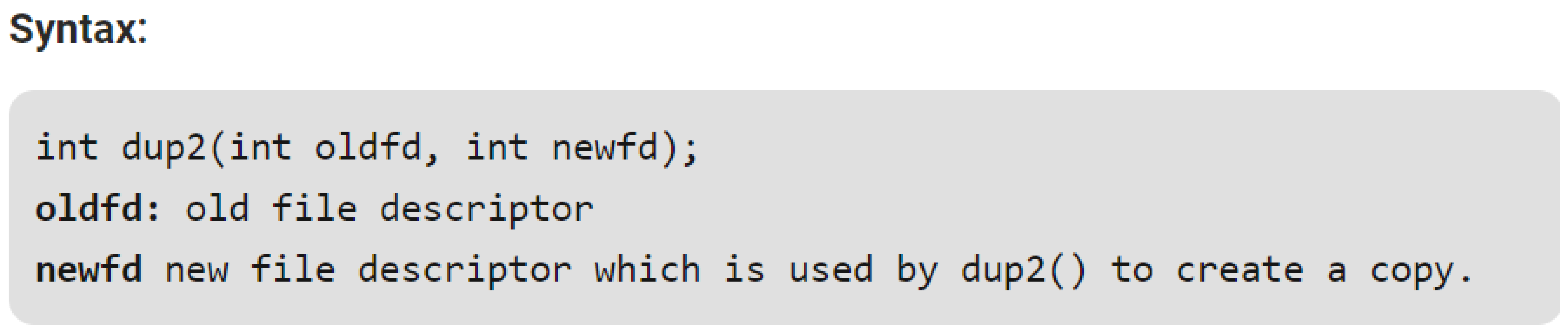


Syntax:

```
int dup(int oldfd);
oldfd: old file descriptor whose copy is to be created.
```

dup2()

The dup2() system call is similar to dup() but the basic difference between them is that instead of using the lowest-numbered unused file descriptor, it uses the descriptor number specified by the user.



Syntax:

```
int dup2(int oldfd, int newfd);
oldfd: old file descriptor
newfd new file descriptor which is used by dup2() to create a copy.
```