3.Implement **pipe()** system calls.

pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication.

The array pipefd is used to return two file descriptors referring to the ends of

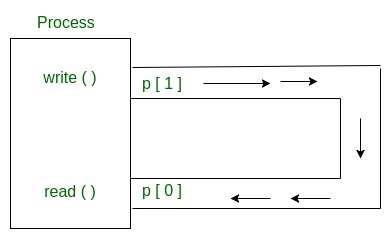
the pipe.

* pipefd[0] refers to the read end of the pipe.
* pipefd[1] refers to the write end of the pipe.

 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 linux Operating System, Pipes are useful for communication between related processes(inter-process communication).

* Pipe is one-way communication only i.e we can use a pipe such that One process write to the pipe, and the other process reads from the pipe.
* It opens a pipe, which is an area of main memory that is treated as a ***“virtual file”***.
* The pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to this “virtual file” or pipe and another related process can read from it.
* If a process tries to read before something is written to the pipe, the process is suspended until something is written.
* The pipe system call finds the first two available positions in the process’s open file table and allocates them for the read and write ends of the pipe.



The write() system call is used to write content to the pipe using the file descriptor pipefd[1].

The syntax of the write() method is as follows:

**#include<unistd.h>**

**ssize\_t write(int pipefd, void\* buff, size\_t s);**

The system call has three arguments:

* The file descriptor of the write end of the pipe, pipefd, is returned by the pipe() system call.
* A buffer, buff, contains data that needs to be written to the pipe end.
* The buffer can be dynamic or static in size, and the size is sent as a parameter, s.

On successful execution of the write() system call, the number of bytes written is returned. In case of error, -1 is returned.

The read() system call is used to write content to the pipe using the file descriptor pipefd[0].

The syntax of the read() method is as follows:

**#include<unistd.h>**

**ssize\_t returnsize = read(int pipefd, void\* buff, size\_t s);**

The system call has three arguments:

* The file descriptor of the read end of the pipe, pipefd, is returned by the pipe() system call.
* A buffer, buff, is used to store the data read from the pipe.
* The number of bytes that needs to be read from the pipe is sent as a parameter, s.

On successful execution of the read() system call, the number of bytes written is returned. In case of error, -1 is returned.

The pipe() system call automatically opens two file descriptors for reading and writing. Both the open file descriptors must be closed at the end of the process to execute the program successfully.

The following command is used to close the file descriptors:

**#include<unistd.h>**

**int close(int pipefd);**

Zero is returned on success by the close() system call. In case of failure, -1 is returned.

### ****Syntax in C language:****

**int pipe(int fds[2]);Parameters :**

**fd[0]** will be the fd(file descriptor) for the

read end of pipe.

**fd[1]** will be the fd for the write end of pipe.

**Returns :** 0 on Success.

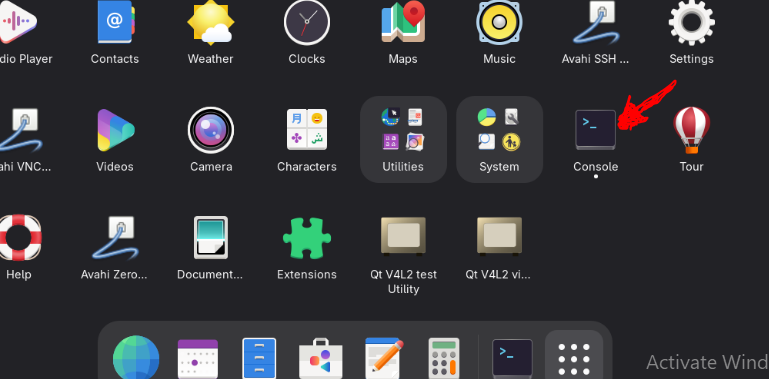
**-1** on error.

Pipes behave **FIFO**(First in First out), Pipe behave like a **queue** data structure. Size of read and write don’t have to match here. We can write **512** bytes at a time, but we can read only 1 byte at a time in a pipe.

When we use fork in any process, file descriptors remain open across child process and also parent process. If we call fork after creating a pipe, then the parent and child can communicate via the pipe.

**There are steps to implement pipe() system call as follows;**

1.Fistly I open console or terminal on installed arch linux opertating system



### 2.then I write nano pipe\_example.c on console to convert the C source code written in pipe\_example.c into machine code that the computer can execute. The GCC compiler translates high-level code into low-level instructions that the CPU can understand.

### C:\Users\Tshiba\Pictures\gcc 1.PNG

### 3. then I write a code on console arch linux

### C:\Users\Tshiba\Pictures\gcc 2.PNG

### 4.compile the the written code by writing gcc –o pipe\_example pipe\_example.c.

### C:\Users\Tshiba\Pictures\w.PNG

### If there is any error during compilation the compiler show the error.like

### C:\Users\Tshiba\Pictures\systm call4.PNG

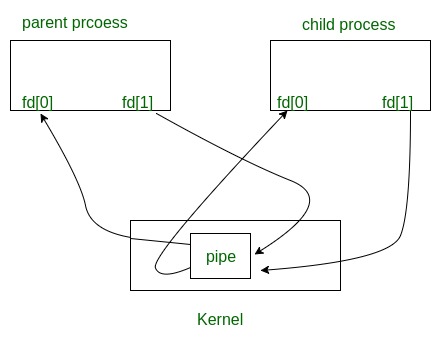
### 6.Run the program by writing the ./pipe\_example

### C:\Users\Tshiba\Pictures\w.PNG

### 7.finally the output of the code could be;

### 

### Parent and child sharing a pipe



* Since Linux general notification mechanism is built on the top of the pipe where kernel splices notification messages into pipes opened by user space.
* The owner of the pipe has to tell the kernel which sources of events to watch and filters can also be applied to select which subevents should be placed into the pipe.
* On success, zero is returned. On error, -1 is returned, [*errno*](https://man7.org/linux/man-pages/man3/errno.3.html) is set to indicate the error, and *pipefd* is left unchanged.
* On Linux (and other systems), **pipe**() does not modify *pipefd* on failure.
* A requirement standardizing this behavior was added in POSIX.1-2008 TC2.
* The Linux-specific **pipe2**() system call likewise does not modify *pipefd* on failure.

END!!!