**Experiment No 4**

Title: Implementation of pipe() system call.

Aim: To study and implement pipe() system call.

Objective: To study and implement following

1. PIPE system call

Relevance: To get knowledge about mechanism that allows two or more processes to communicate and share data

Theory:

The pipe() system call in operating systems facilitates interprocess communication by creating a unidirectional communication channel between two processes. It allows one process to write data into the pipe, while another process can read from it. This mechanism is particularly useful for achieving coordination and data transfer between processes, such as in pipelines or filters. **Pipes** are a fundamental building block for implementing more complex communication and synchronization mechanisms in Unix-like operating systems. They provide a seamless way for processes to exchange data without the need for shared memory or explicit file operations, enhancing the modularity and efficiency of process communication in a multitasking environment.

**Pipes in OS** are a mechanism that allows two or more processes to communicate and share data. It enables the flow of data from the output (stdout) of one process directly into the input (stdin) of another process without the need for intermediate files or temporary storage.

Pipes in OS are typically used for **interprocess communication (IPC)** and are a fundamental feature in Unix-like operating systems, including **Linux** and **macOS**. They are represented by the | symbol in the command line.

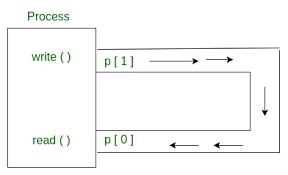
How a **pipe in OS** works:

* **Process A** generates some output data and sends it to the standard output (stdout).
* **Process B**, which is running concurrently or as a separate process, reads from its standard input (stdin).
* By connecting the stdout of **Process A** to the stdin of **Process B** using the | symbol in the command line, the data flows directly from **Process A** to **Process B** without being written to a file or stored in memory.

Pipes are useful for a wide range of tasks, such as:

* Combining the output of multiple commands or processes.
* Feeding the output of one command as input to another (e.g., command1 | command2).
* Implementing **data processing pipelines**, where each process performs a specific operation on the data.

Pipes in OS are a simple yet powerful way to build complex data processing workflows in a Unix-like environment, enabling efficient communication and coordination between processes.



**Syntax-**

**int pipe(int pipefd[2]);**

Where:

* pipefd is an integer array of size 2, which will hold two file descriptors after the pipe is created:
  1. pipefd[0] - This file descriptor is for reading from the pipe (the read end).
  2. pipefd[1] - This file descriptor is for writing to the pipe (the write end).

### Implementation-

The pipe() system call in the C programming language is used to create an **inter-process communication (IPC)** channel between two processes. It allows one process to send data to another process through this channel, which is essentially a unidirectional data stream. Here's a brief implementation of pipe() in an operating system in C:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main()

{

int fd[2]; *// File descriptors for the pipe*

if (pipe(fd) == -1)

{

perror("Pipe creation failed");

exit(EXIT\_FAILURE);

}

pid\_t pid = fork(); *// Fork a child process*

if (pid == -1)

{

perror("Fork failed");

exit(EXIT\_FAILURE);

}

if (pid == 0)

{

*// Child process*

close(fd[0]); *// Close the read end of the pipe*

char message[] = "Hello from the child process!\n";

write(fd[1], message, sizeof(message));

close(fd[1]); *// Close the write end of the pipe in the child*

}

else

{

*// Parent process*

close(fd[1]); *// Close the write end of the pipe*

char buffer[100];

read(fd[0], buffer, sizeof(buffer));

printf("Parent received: %s", buffer);

close(fd[0]); *// Close the read end of the pipe in the parent*

}

return 0;

}

**Output:…………**