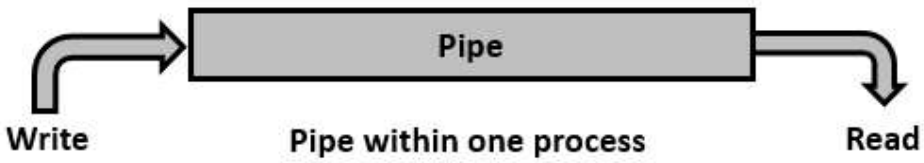
Inter Process C

Pipe is a communication medium between two or more related or interrelated processes. It can be either within one process or a communication between the child and the parent processes. Communication can also be multi-level such as communication between the parent, the child and the grand-child, etc. Communication is achieved by one process writing into the pipe and other reading from the pipe. To achieve the pipe system call, create two files, one to write into the file and another to read from the file.

Pipe mechanism can be viewed with a real-time scenario such as filling water with the pipe into some container, say a bucket, and someone retrieving it, say with a mug. The filling process is nothing but writing into the pipe and the reading process is nothing but retrieving from the pipe. This implies that one output (water) is input for the other (bucket).



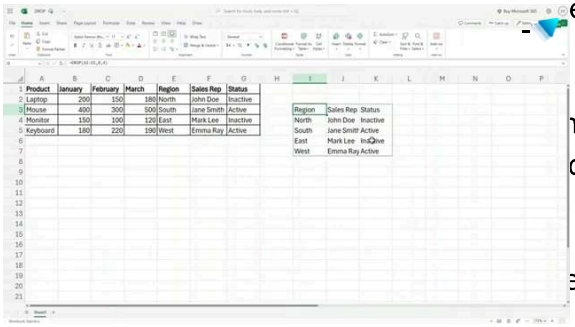

```
#include<unistd.h>

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

This system call would create a pipe for one-way communication i.e., it creates two ead from the pipe and other one is connected to

nd pipedes[1] is for writing. Whatever is written des[0].

and -1 in case of failure. To know the cause of rror() function.



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

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Even though the basic operations for file are read and write, it is essential to open the file before performing the operations and closing the file after completion of the required operations. Usually, by default, 3 descriptors opened for every process, which are used for input (standard input stdin), output (standard output stdout) and error (standard error stderr) having file descriptors 0, 1 and 2 respectively.

This system call would return a file descriptor used for further file operations of read/write/seek (lseek). Usually file descriptors start from 3 and increase by one number as the number of files open.

The arguments passed to open system call are pathname (relative or absolute path), flags mentioning the purpose of opening file (say, opening for read, O_RDONLY, to write, O_WRONLY, to read and write, O_RDWR, to append to the existing file O_APPEND, to create file, if not exists with O_CREAT and so on) and the required mode providing permissions of read/write/execute for user or owner/group/others. Mode can be mentioned with symbols.

Read 4, Write 2 and Execute 1.

For example: Octal value (starts with 0), 0764 implies owner has read, write and execute permissions, group has read and write permissions, other has read permissions. This can also be represented as S_IRWXU | S_IRGRP | S_IWGRP | S_IROTH, which implies or operation of 0700|0040|0020|0004 → 0764.

This system call, on success, returns the new file descriptor id and -1 in case of error. The cause of error can be identified with errno variable or perror() function.

```c
#include<unistd.h>

int close(int fd)
```


opened file descriptor. This implies the file is no
d can be reused by any other process. This system
case of error. The cause of error can be identified
n.

```
#include<unistd.h>

ssize_t read(int fd, void *buf, size_t count)
```

The above system call is to read from the specified file with arguments of file descriptor fd, proper buffer with allocated memory (either static or dynamic) and the size of buffer.

The file descriptor id is to identify the respective file, which is returned after calling open() or pipe() system call. The file needs to be opened before reading from the file. It automatically opens in case of calling pipe() system call.

This call would return the number of bytes read (or zero in case of encountering the end of the file) on success and -1 in case of failure. The return bytes can be smaller than the number of bytes requested, just in case no data is available or file is closed. Proper error number is set in case of failure.

To know the cause of failure, check with errno variable or perror() function.

```
#include<unistd.h>

ssize_t write(int fd, void *buf, size_t count)
```

The above system call is to write to the specified file with arguments of the file descriptor fd, a proper buffer with allocated memory (either static or dynamic) and the size of buffer.

The file descriptor id is to identify the respective file, which is returned after calling open() or pipe() system call.

The file needs to be opened before writing to the file. It automatically opens in case of calling pipe() system call.

This call would return the number of bytes written (or zero in case nothing is written) on success and -1 in case of failure. Proper error number is set in case of failure.

To know the cause of failure, check with errno variable or perror() function.

s.

rite and read two messages using pipe.

Step 1 – Create a pipe.

Step 2 – Send a message to the pipe.

Step 3 – Retrieve the message from the pipe and write it to the standard output.

Step 4 – Send another message to the pipe.

Step 5 – Retrieve the message from the pipe and write it to the standard output.

Note – Retrieving messages can also be done after sending all messages.

Source Code: simplepipe.c

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

int main() {
   int pipefds[2];
   int returnstatus;
   char writemessages[2][20]={"Hi", "Hello"};
   char readmessage[20];
   returnstatus = pipe(pipefds);

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

   printf("Writing to pipe - Message 1 is %s\n", writemessages[0]);
   write(pipefds[1], writemessages[0], sizeof(writemessages[0]));
   read(pipefds[0], readmessage, sizeof(readmessage));
   printf("Reading from pipe  Message 1 is %s\n", readmessage);
   printf("Writing to pipe - Message 2 is %s\n", writemessages[0]);
   write(pipefds[1], writemessages[1], sizeof(writemessages[0]));
   read(pipefds[0], readmessage, sizeof(readmessage));
   printf("Reading from pipe  Message 2 is %s\n", readmessage);
```


be checked for every system call. To simplify the
e calls.

Execution Steps

Compilation

```
gcc -o simplepipe simplepipe.c
```

Execution/Output

```
Writing to pipe - Message 1 is Hi
Reading from pipe  Message 1 is Hi
Writing to pipe - Message 2 is Hi
Reading from pipe  Message 2 is Hell
```

Example program 2 – Program to write and read two messages through the pipe using the parent and the child processes.

Algorithm

pipe.

message from the pipe and writes it to the standard

Step 5 – Repeat step 3 and step 4 once again.

Source Code: pipewithprocesses.c

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

int main() {
   int pipefds[2];
   int returnstatus;
   int pid;
   char writemessages[2][20]={"Hi", "Hello"};
   char readmessage[20];
   returnstatus = pipe(pipefds);
   if (returnstatus == -1) {
      printf("Unable to create pipe\n");
      return 1;
   }
   pid = fork();

   // Child process
   if (pid == 0) {
      read(pipefds[0], readmessage, sizeof(readmessage));
      printf("Child Process - Reading from pipe  Message 1 is %s\n",
readmessage);
      read(pipefds[0], readmessage, sizeof(readmessage));
      printf("Child Process - Reading from pipe  Message 2 is %s\n",
readmessage);
   } else { //Parent process
      printf("Parent Process - Writing to pipe - Message 1 is %s\n",
writemessages[0]);
      write(pipefds[1], writemessages[0], sizeof(writemessages[0]));
      printf("Parent Process - Writing to pipe - Message 2 is %s\n",
writemessages[1]);
      write(pipefds[1], writemessages[1], sizeof(writemessages[1]));
```


```
gcc pipewithprocesses.c o pipewithprocesses
```

Execution

```
Parent Process - Writing to pipe - Message 1 is Hi
Parent Process - Writing to pipe - Message 2 is Hello
Child Process - Reading from pipe  Message 1 is Hi
Child Process - Reading from pipe  Message 2 is Hello
```

Two-way Communication Using Pipes

Pipe communication is viewed as only one-way communication i.e., either the parent process writes and the child process reads or vice-versa but not both. However, what if both the parent and the child needs to write and read from the pipes simultaneously, the solution is a two-way communication using pipes. Two pipes are required to establish two-way communication.

Following are the steps to achieve two-way communication −

Step 1 − Create two pipes. First one is for the parent to write and child to read, say as pipe1. Second one is for the child to write and parent to read, say as pipe2.

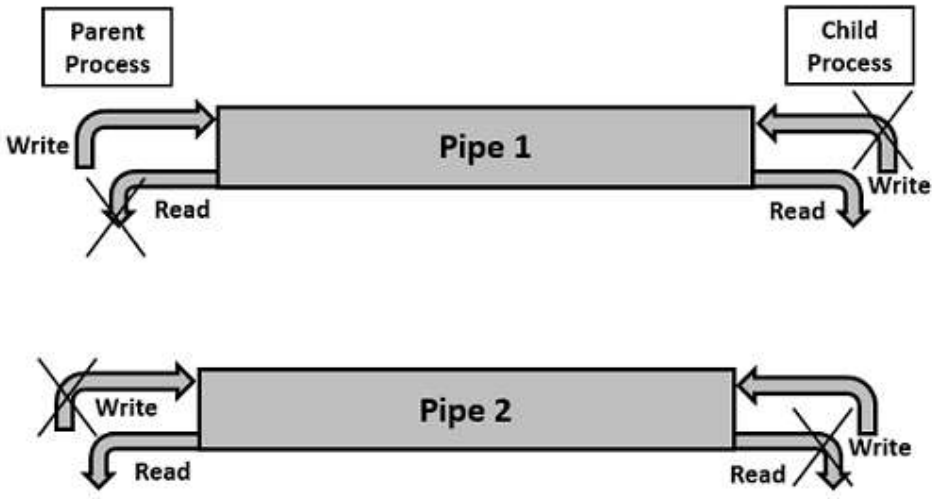
Step 2 − Create a child process.

Step 3 − Close unwanted ends as only one end is needed for each communication.

Step 4 − Close unwanted ends in the parent process, read end of pipe1 and write end of pipe2.

Step 5 − Close the unwanted ends in the child process, write end of pipe1 and read end of pipe2.

Step 6 − Perform the communication as required.



Sample Programs

Sample program 1 – Achieving two-way communication using pipes.

Algorithm

Step 1 – Create pipe1 for the parent process to write and the child process to read.

Step 2 – Create pipe2 for the child process to write and the parent process to read.

Step 3 – Close the unwanted ends of the pipe from the parent and child side.

Step 4 – Parent process to write a message and child process to read and display on the screen.

Step 5 – Child process to write a message and parent process to read and display on the screen.

Source Code: twowayspipe.c

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

int main() {
   int pipefds1[2], pipefds2[2];
```



```
s2;

Hi";
Hello";

;
```

```
    if (returnstatus1 == -1) {
       printf("Unable to create pipe 1 \n");
       return 1;
    }
    returnstatus2 = pipe(pipefds2);

    if (returnstatus2 == -1) {
       printf("Unable to create pipe 2 \n");
       return 1;
    }
    pid = fork();

    if (pid != 0) // Parent process {
       close(pipefds1[0]); // Close the unwanted pipe1 read side
       close(pipefds2[1]); // Close the unwanted pipe2 write side
       printf("In Parent: Writing to pipe 1  Message is %s\n",
pipe1writemessage);
       write(pipefds1[1], pipe1writemessage, sizeof(pipe1writemessage));
       read(pipefds2[0], readmessage, sizeof(readmessage));
       printf("In Parent: Reading from pipe 2  Message is %s\n", readmessage);
    } else { //child process
       close(pipefds1[1]); // Close the unwanted pipe1 write side
       close(pipefds2[0]); // Close the unwanted pipe2 read side
       read(pipefds1[0], readmessage, sizeof(readmessage));
       printf("In Child: Reading from pipe 1  Message is %s\n", readmessage);
       printf("In Child: Writing to pipe 2  Message is %s\n",
pipe2writemessage);
       write(pipefds2[1], pipe2writemessage, sizeof(pipe2writemessage));
    }
    return 0;
}
```

Execution Steps

```
In Parent: Writing to pipe 1  Message is Hi
In Child: Reading from pipe 1  Message is Hi
In Child: Writing to pipe 2  Message is Hello
In Parent: Reading from pipe 2  Message is Hello
```

Data Science Advanced Certification

Cloud Computing And DevOps

Advanced Certification In Business Analytics

Artificial Intelligence And Machine Learning

DevOps Certification

Game Development Certification

Front-End Developer Certification

AWS Certification Training

Python Programming Certification

COMPILERS & EDITORS

Online Java Compiler

Online Python Compiler

Online Go Compiler

Online C Compiler

Online C++ Compiler

Online C# Compiler

Online PHP Compiler

Online MATLAB Compiler

Online Bash Compiler

Online SQL Compiler

Online Html Editor

tutorialspoint

Tutorials Point is a leading Ed Tech company striving to provide the best learning material on technical and non-technical subjects.