# ***INTERPROCESS COMMUNICATION USING FIFO (NAMED PIPES)***


Submitted By:

| | |
|---|---|
| L Chinmay | 1BM18IS050 |
| Kunala Sri Manaswini | 1BM18IS049 |

Submitted To:

Dr. Shubha Rao V

# *ABSTRACT*

Inter-process communication is a mechanism to allow the two processes to communicate with each other and synchronize their action. On modern systems, IPCs form the web that bind together each process within a large scale software architecture. There are mainly four types of inter-process communication :

1. PIPES
2. FIFO
3. Message Queuing
4. Semaphores

- **Pipe**s are temporary in the sense that they cease to exist when no process has them open.

- **FIFO**s or named pipes are special files that persist even after all processes have closed them. A FIFO has a name and permission just like an ordinary file and appears in directory listing.
- **Message Queuing** allows messages to be passed between processes using either a single queue or several message queue. This is managed by system kernel these messages are coordinated using an API.
- **Semaphore** is used in solving problems associated with synchronization and to avoid race condition. These are integer values which are greater than or equal to 0.

**Need of Inter process communication:**

There are several reasons for providing an environment that allows process cooperation:

- Information sharing
- Speedup
- Modularity
- Convenience

# *INTRODUCTION*

- Inter-process communication is a mechanism to allow the two processes to communicate with each other and synchronize their action. There are mainly four types of inter-process communication i.e PIPES, FIFO,Message Queuing and Semaphores..
- Pipes are temporary in the sense that they cease to exist when no process has them open
- FIFOs or named pipes, are special files that persist even after all processes have closed them
- A FIFO has a name and permissions just like an ordinary file and appears in a directory listing
- Any process with the appropriate permissions can access a FIFO
- A user creates a FIFO by executing the mkfifo() command from a command shell or by calling the mkfifo() function from within a program

# ***API USED:***

- *int mkfifo(const char *pathname, mode_t mode);*
- *int open(const char *pathname, int flags);*
- *int open(const char *pathname, int flags, mode_t mode);*
- *ssize_t read(int fd, void *buf, size_t count);*
- *ssize_t write(int fd, const void *buf, size_t count);*
-

# *CREATING FIFO FILE USING mkfifo()*

- **int mkfifo (const char *pathname, mode_t mode);**
- mkfifo() makes a FIFO special file with name pathname. mode specifies the FIFO's permissions.
- A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an anonymous communications channel, a FIFO special file is entered into the filesystem by calling mkfifo().
- Once you 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. Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa.

# *OPENING A FIFO USING open()*

- **int open(const char *path, int oflag, ... );**
- The open() function shall establish the connection between a file and a file descriptor. It shall create an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file.
- The path argument points to a pathname naming the file.
- The open() function shall return a file descriptor for the named file that is the lowest file descriptor not currently open for that process.
- The open file description is new, and therefore the file descriptor shall not share it with any other process in the system

# *READING A FIFO USING read()*

- **ssize_t read(int fd, void *buf, size_t count);**
- read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf. On files that support seeking, the read operation commences at the file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and read() returns zero.
- If count is zero, read() may detect the errors.On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested

# *WRITING INTO A FIFO USING write()*

- **ssize_t write(int fd, const void *buf, size_t count);**
- The number of bytes written may be less than count. For a seekable file may be applied, writing takes place at the current file offset, and the file offset is incremented by the number of bytes actually written.
- If the file was opened with O_APPEND, the file offset is first set to the end of the file before writing. The adjustment of the file offset and the write operation are performed as an atomic step.
- On success, the number of bytes written is returned (zero indicates nothing was written).
- On error, -1 is returned, and errno is set appropriately.

# *REFERENCES*


- **Linux Programmer's Manual**
- **www.geeksforgeeks.org**
- **www.w3schools.com**
- **Introduction to Named Pipes by Andy Vaught**