



<b>Course</b>	<b>Inter Process Communication -</b>	
<b>Module Day 4</b>	<b>Pipes and FIFO and Signals Handling</b>	<b>3 Hours</b>
	<p>At the end of this module you will be able to know:</p> <ul style="list-style-type: none"> <li>▪ What are the different IPC mechanisms available in Linux?</li> <li>▪ What is a Signal? And Different Signals in Linux,</li> <li>▪ What are the different Medias to generate Signals?</li> <li>▪ How to handle the Signals?</li> <li>▪ System Calls related to Signals,</li> <li>▪ What is a Pipe and Working with Pipes</li> <li>▪ What are the advantages and disadvantages of a Pipe?</li> <li>▪ What is a FIFO and working with FIFO</li> <li>▪ What are the advantages and disadvantages of FIFO?</li> </ul>	

Session Plan	
1	Introduction to IPC's
2	Different medias to generate Signals, Signal Processing of Handling
3	Important Signals in Linux, Examples and assignments.
4	Working with Pipes
5	Working with FIFOs.
6	System calls related to PIPE and FIFOs with code examples

	<b>Module Overview</b>
---	------------------------

- ✓ Introduction to IPC mechanisms
- ✓ Introduction to Signals,
- ✓ Different medias to generate signals,
- ✓ Signal Handling,

- ✓ Important Signals in Linux,
- ✓ System Calls related to Signals.
- ✓ Introduction to Pipes
- ✓ System calls related to Pipes
- ✓ Introduction to FIFOs
- ✓ Rules for reading/writing from/to FIFO
- ✓ System calls related to FIFOs



## Introduction to IPC mechanisms

Before starting to know Inter Process Communication (IPC) mechanisms in Linux, Let's understand what is communication?

**Communication:** Communication is the exchange of thoughts, messages, or information, as by speech, visuals, signals, writing, or behaviour with certain set of predefined rules applied.

The word communication is derived from the Latin word "communis", meaning to share. Communication requires a sender, a message, and a recipient.

### Inter Process Communication:

Interprocess communication (IPC) refers to the coordination of activities among cooperating processes. A common example of this need is managing access to a given system resource.

The increasingly important role that distributed systems play in modern computing environments exacerbates the need for IPC. Systems for managing communication and synchronization between cooperating processes are essential to many modern software systems. IPC has always played a prominent role in UNIX-variant operating systems.

As in the name, IPC means Communication (exchanging the data) between the different processes. To communicate different processes, The Linux IPC (Inter-process communication) facilities provide a method for multiple processes to communicate with one another. Those are:

- Pipes:
  - Pipes permit sequential communication from one process to a related process.
- FIFOs:

- FIFOs are similar to Pipes, except that unrelated processes can communicate because the Pipe is given a name in the file system.
- Shared Memory:
  - Shared memory permits processes to communicate by simply reading and writing to a specified memory location.
- Message Queues:
  - Information to be communicated is placed in a predefined message structure. The process generating the message specifies its type and places the message in a system-maintained message queue. Processes accessing the message queue can use the message type to selectively read messages of specific types in a first in first out (FIFO) manner. Message queues provide the user with a means of asynchronously multiplexing data from multiple processes.
- Sockets:
  - Sockets support communication between unrelated processes even on different computers.

### **IPC Properties:**

Persistence: How long and under what conditions an IPC object remains in existence.

- Process-persistent: exists until last process with IPC object open closes the object. Ex: Pipe.
- Kernel-persistent: exists until kernel reboots or IPC object is explicitly deleted. Ex: semaphore, shared memory.
- File system-persistent: exists in the file system or until IPC object is explicitly deleted. Ex. memory mapped shared memory.

### **Name-spaces**

IPC objects can be referred by names or identifiers.

Ex: sockets, Pipes and files are identified with file descriptors, which is a system-wide unique non-negative integer.

Ex: semaphore, message queue and shared memory are named with keys and identified by system IDs in their own name spaces.

All these IPC's are different by the following criteria:

- Whether they restrict communication, to related processes (processes with a common ancestor), to unrelated processes sharing the same file system, or to any computer connected to a network.
- Whether a communicating process is limited to only write data or only read data
- The number of processes permitted to communicate

- Whether the communicating processes are synchronized by the IPC — for example, a reading process halts until data is available to read.



## Introduction to Signals

- Signal is a notification, a message sent by either operating system or some application to your program (or one of its threads).
- Signals are a mechanism for one-way asynchronous notifications.
- A signal may be sent from the kernel to a process, from a process to another process, or from a process to itself.
- Processes can also communicate with signals, So we can call signals are also one of the IPC mechanism,
- **Signal** typically alerts a process to some event, such as a segmentation fault, or the user pressing Ctrl-C.
- Linux kernel implements about 32 signals. Each signal identified by a number, from **0 to 31**. Signals don't carry any argument and their names are mostly self explanatory.
- For instance SIGKILL or signal number 9 tells the program that someone tries to kill it.
- A **signal** is nothing but an **integer value** or an interrupt,
- All the signals are available in signal.h,
- All the signals are designed with specific meaning,
- All the signals are processing through the KERNEL,
- KERNEL will be maintaining one common signal handling table.

### Signal as Interrupt:

- In addition to informative nature of signals, they also interrupt your program. I.e to handle a signal, one of the threads in your program, stops its execution and temporarily switches to signal handler.

- Note that as in version 2.6 of Linux kernel, most of the signals interrupt are only one thread and not the entire application as it used to be once.
- Moreover, signal handler itself can be interrupted by some other signal

### Signals that report exceptions:

- Another way of using signals is to indicate that something bad have happened.
- For instance when your program causes a segmentation fault, operating system sends SIGSEGV signal to your application.
- And also when you are doing floating point operations without co-processor in your system, then SIGFPE signal will be sent to Application program.



### Different medias to generate signals

There are 5 different medias to generate signals:

- Commands
- Functions
- Keys
- Hardware exceptions
- Software conditions

### Commands:

- Use kill command to send a signal to a process.
- For example:

```
< > $ ./a.out&
```

```
< > ps -C a.out
```

```
PID TTY TIME CMD
```

```
3699 pts/1 00:00:00 a.out
```

```
< > $ kill 3699
```

- We can also use killall, pkill, xkill, etc.

## Functions:

- You can use the LINUX system call **kill** (from a C program) to send signal from one process to another.
- **Kill** system call takes two arguments:
  - the PID (process id) of the process that needs to be signaled
  - the signal that needs to be send to the process.
- Kill function returns 0 when it is successful.
- Kill function returns -1 when it is fails.
- Example:

- The following C code snippet shows how to use the kill function.

```
int send_signal (int pid)
{
    int ret;
    ret = kill(pid,SIGHUP);
    printf("ret : %d",ret);
}
```

## Keys:

- We can also generate signals by using keys:
- Examples:  
ctrl+c // (SIGINT) interrupt signal to kill the process  
ctrl+\ // (SIGQUIT) kill the process with core dump message  
ctrl+d // (SIGUP) kill the background running process

## Hardware exceptions:

- Performing floating point arithmetic operations without co-processor in system will cause to generate a signal called SIGFPE, because normal CPU can't perform this operation.

- Whenever you are doing an illegal operation regarding with memory, Then SIGSEGV signal will be generates.

### Software Conditions:

- In socket programming, When client is sending the data with high baud rate then TCP/IP layer in client will generate one signal **SIGURG** to server TCP/IP to say that to receive more data.
- If a process writes to a fifo, but there is no processes existence that having open for reading, **SIGPIPE** signal is generated. And writes returns with an error called EPIPE.
- Default action of EPIPE is to terminate the process.
- More ex: **SIGTRAP**, **SIGALRM**, etc



## Signal Handling

Each one of signals can be in one of three states:

- We may have our own signal handler for the signal.
- Signal may be handled by the default handler. Every signal has its default handler function. For instance, SIGINT default handler will terminate your application.
- Signal may be ignored. Ignoring signal sometimes referred to as blocking signal.
- Signal handling can be done by using **signal()** function.

**sighandler\_t signal(int signum, sighandler\_t handler);**

- signal() sets the disposition of the signal signum to handler, which is either SIG\_IGN, SIG\_DFL, or the address of a programmer-defined function (a "signal handler").
- The behavior of signal() varied historically across different versions of Linux. Avoid its use: use sigaction() instead.
- If the signal signum is delivered to the process, then one of the following happens:

- If the disposition is set to **SIG\_IGN**, then the signal is ignored.
- If the disposition is set to **SIG\_DFL**, then the default action associated with the signal.
- If the disposition is set to a function, then handler is called with argument signum.

### Examples:

- Signal ignoring:

**signal ( SIGINT, SIG\_IGN);**

- Signal Handler calling:

**signal ( SIGINT, function\_name);**

- Default action:

**signal ( SIGINT, SIG\_DFL);**

Ex:- /\*\*\*\*\* WAP to Show how SIGINT signal will kill your process \*\*\*\*\*/

Header files

```
int main() {
    while(1) {
        printf("press cntl+c \n");
        sleep(1);          // To generate 1 sec delay
    }
    return 0;
}
```

**Explanation:-** This program will print the string “press cntl+c” till user presses cntl+c. When user presses cntl+c then SIGINT signal will be sent to the process and default action is to terminate the process.

Ex:- /\*\*\*\*\* WAP to execute user defined signal handler for SIGINT \*\*\*\*\*/

Header files

```
void my_signal_handler() // user handler
{
    printf("you can't kill this process by pressing cntl+c\n");
}
```



```

}
int main() {
    signal(SIGINT, my_signal_handler);    // registering user handler to signal
    while(1) {
        printf("press cntl+c \n");
        sleep(1);           // To generate 1 sec delay
    }
    return 0;
}

```

**Explanation:-** This program will print the string "press cntl+c" till user presses cntl+c. When user presses cntl+c then SIGINT signal will be sent to the process and user defined signal handler will be executed. This program will terminate when you press cntl+z.

Ex:- /\*\*\*\*\* WAP to Show how ignore SIGINT signal \*\*\*\*\*/

Header files

```

int main() {
    signal(SIGINT, SIG_IGN); // ignoring SIGINT signal.
    while(1) {
        printf("press cntl+c \n");
        sleep(1);           // To generate 1 sec delay
    }
    return 0;
}

```



## Important Signals in Linux

### SIGHUP:

This signal indicates that someone has killed the controlling terminal. For instance, let's say our program runs in x-terminal. When someone kills the terminal program, without

killing applications running inside of terminal window, operating system sends SIGHUP to the program. Default handler for this signal will terminate your program.

### **SIGINT:**

This is the signal that being sent to your application when it is running in a foreground in a terminal and someone presses CTRL-C. Default handler of this signal will quietly terminate your program.

### **SIGQUIT:**

Again, according to documentation, this signal means “Quit from keyboard”. When you press CTRL-\ this signal will sent to process and terminates with core dump message.

### **SIGILL:**

Illegal instruction signal. This is a exception signal, sent to your application by the operating system when it encounters an illegal instruction inside of your program. Something like this may happen when executable file of your program has been corrupted.

### **SIGABRT:**

Abort signal means you used abort() API inside of your program. It is yet another method to terminate your program. abort() issues SIGABRT signal which in its term terminates your program (unless handled by your custom handler).

### **SIGFPE:**

Floating point exception. This is another exception signal, issued by operating system when your application caused an exception.

### **SIGSEGV:**

This is an exception signal as well. Operating system sends a program this signal when it tries to access memory that does not belong to it.

**SIGPIPE:**

Broken named pipe. As documentation states, this signal sent to your program when you try to write into fifo (another IPC) with no readers on the other side.

**SIGALRM:**

Alarm signal. Sent to your program using alarm() system call. The alarm() system call is basically a timer that allows you to receive SIGALRM in preconfigured number of seconds.

**SIGTERM:**

This signal tells your program to terminate itself. Consider this as a signal to cleanly shut down while SIGKILL is an abnormal termination signal.

**SIGCHLD:**

Tells you that a child process of your program has stopped or terminated. This is handy when you wish to synchronize your process with a process with its child.

**SIGUSR1 and SIGUSR2:**

Finally, SIGUSR1 and SIGUSR2 are two signals that have no predefined meaning and are left for your consideration. You may use these signals to synchronize your program with some other program or to communicate with it.

**System Calls related to Signals**

## summary of signal system call

Include Files	<signals.h>	Manual section	2
Summary	<b>Void (*signal (int signum,void(*sighandler)(int)))(int);</b>		
Return	Success	Failure	
	Signals previous disposition	SIG_ERR(defined as -1)	

As you know that to work with signals we have signal() function. But historically, the behavior of this function will change from kernel version to version. So developers will not prefer to work with signal(). There is another function called sigaction() as development of signal() function.

This function declaration is given below:

**int sigaction (int signum, const struct sigaction \*act, struct sigaction \*oldact);**

## summary of sigaction() system call

Include Files	<signals.h>	Manual section	2
Summary	<b>int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);</b>		
Return	Success	Failure	
	0	-1	

- The sigaction() system call is used to change the action taken by a process on receipt of a specific signal. It is much more advanced comparing to good old **signal()**.
- signum specifies the signal and can be any valid signal except SIGKILL and SIGSTOP.

- If act is non-NULL, the new action for signal signum is installed from act. If oldact is non-NULL, the previous action is saved in oldact.
- The sigaction structure is defined as something like:

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction) (int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

Ex:- /\*\*\*\*\* WAP to execute user signal handler for SIGINT using sigaction() \*\*\*\*\*/

Header files

```
static volatile sig_atomic_t doneflag = 0;
```

```
static void handler( int sig, siginfo_t *siginfo, void *ptr) {
```

```
    printf("signal : %d\n", siginfo->si_signo);
```

```
    printf("sigcode: %d\n", siginfo->si_code);
```

```
    doneflag = 1;
```

```
}
```

```
int main() {
```

```
    struct sigaction act;
```

```
    memset( &act, '\0', sizeof(act));           // placing '\0' char in address &act by
                                                    sizeof(act) times
```

```
    act.sa_sigaction = &handler;
```

```
    act.sa_flags = SA_SIGINFO;
```

```
    if( sigaction( SIGINT, &act, NULL) < 0) {
```

```
        perror("error\n");
```

```
        return 1;
```

```
    }
```

```
    while( !doneflag) {
```

```
        printf("press cntl+c\n");
```

```
        sleep(1);
```

```

    }
    printf("program terminating\n");
    return 0;
}

```

### **Other System calls related to Signals:**

- **sigemptyset():** sigemptyset() initializes the signal set given by set to empty, with all signals excluded from the set.
- **sigfillset():** sigfillset initializes set to full, including all signals.
- **sigaddset():** To add the signal to the signal set.
- **sigdelset():** To delete the signal from signal set.
- **For more information on these signal functions, read manual pages.**



## **Introduction to PIPEs**

Pipes are the eldest IPC tools.

Pipes are a simple synchronized way of passing information between two processes.

A Pipe is a way to connect the output of one program to the input of another program without any temporary file.

Pipes can be viewed as a special file that can store only a limited amount of data and uses a first-in-first-out access scheme to retrieve data.

A Pipe has two ends associated with a pair of file descriptors; a read descriptor use to read data from Pipes and a write descriptor to write data to the Pipes.

Pipes are also called as unnamed / anonymous Pipes

Pipes can only be used between related processes (i.e. a process and one of its child processes, or two of its children).

Pipes cease to exist after the processes are done using them.

Data is passed in-order.

Pipe is a Half duplex synchronous communication channel.

Process reading from empty Pipes gets blocked.

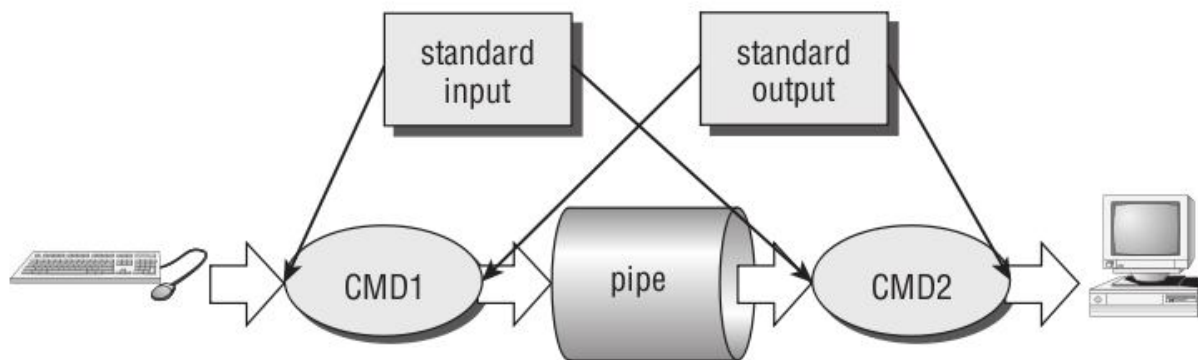
Process writing to filled pipe gets blocked.

Most Linux users will already be familiar with the idea of a pipe, linking shell commands together so that the output of one process is fed straight to the input of another. For shell commands, this is done using the Pipe character to join the commands, such as:

```
<shell>$ cmd1 | cmd2
```

Here the shell arranges the standard input and output of the two commands, so that

- The standard input to cmd1 comes from the terminal keyboard.
- The standard output from cmd1 is fed to cmd2 as its standard input.
- The standard output from cmd2 is connected to the terminal screen.



Pipes can be setup and used only between processes that share parent-child relationship. Generally the parent process creates a Pipe and then forks child processes. Each child process gets access to the Pipe created by the parent process via the file descriptors that get duplicated into their address space. This allows the parent to communicate with its children, or the children to communicate with each other using the shared file descriptor.

Try out: `<shell>$ ps -ef | grep $USER`



### System Calls related to PIPE's

An unnamed pipe is constructed with the pipe() system call.

### Summary of `pipe()` system call

Include Files	<unistd.h>	Manual section	2
Summary	<b><code>fd = pipe(int filedes[2]);</code></b>		
Return	Success	Failure	
	File Descriptors	-1	

0	stdin
1	stdout
2	stderr
.	.
.	.
....	.
MAX	.

Fig: FD Table before execution of pipe system call

If successful, the `pipe()` system call returns a pair of integer file descriptors, `fd[0]` (read end) and `fd[1]` (write end).

0	stdin
1	stdout
2	stderr
3	fd[0]
4	fd[1]
....	
MAX	

Fig: FD Table after execution of pipe system call



### Summary of write system call

Include Files	<unistd.h>	Manual section	2
Summary	<b>ssize_t write(int fd, count, void *buf, size_t</b>		
Return	Success	Failure	
	Number of bytes written	-1	

### Summary of read system call

Include Files	<unistd.h>	Manual section	2
Summary	<b>ssize_t read(int fd, count, void *buf, size_t</b>		
Return	Success	Failure	
	Number of bytes read	-1	

### Summary of dup system call

Include File(s)	<unistd.h>	Manual Section	2
Summary	<b>int dup( int oldfd );</b>		
Return	Success	Failure	Sets errno
	Next available nonnegative file descriptor	-1	Yes

### Summary of dup2 system call

Include File(s)	<unistd.h>	Manual Section	2
Summary	int dup2( int oldfd, int newfd );		
Return	Success	Failure	Sets errno
	newfd as a file descriptor for oldfd	-1	Yes

**Ex-1:** /\*\*\*\*\* WAP to create a Pipe \*\*\*\*\*/

```
int main()
{
    int i, fd[2];

    i = pipe( fd );
    if( i == -1)
        printf("pipe is not created\n");
    else
        printf("pipe is created successfully\n");
    return 0;
}
```

**Using Pipe:** To use Pipe we have to use normal read and write file system calls.

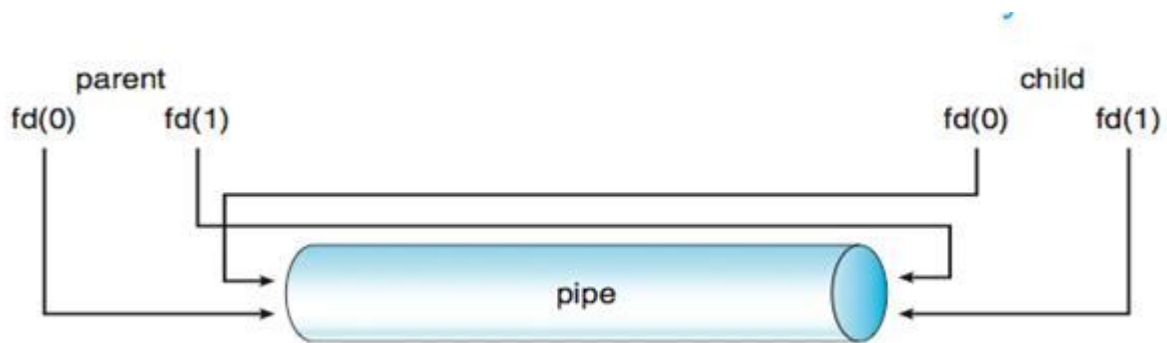
**EX-3:** /\*\*\*\*\* WAP to send a message from child to parent process \*\*\*\*\*/

```
Int main()
{
    Int fd[2], x;
    Char buf[10];
    pipe(fd);
    x = fork();
    if(x == -1)
    {
        printf("error in creating child\n");
        exit(0);
    }
}
```

```

else if(x == 0)
{
    close(fd[0]);
    write(fd[1], "hello", 5);
    printf("message sent\n");
}
else
{
    close(fd[1]);
    read(fd[0], buf, 5);
    printf("received msg: %s\n", buf);
}
return 0;
}

```



### Advantages:

- Built-in synchronization.
- Process persistent(saves resources)
- Can be used at shell command line

### Limitations:

- Cannot be use to communicate between unrelated processes.



## Introduction to FIFO's

The work around for the problems in Pipe is to create a named Pipe which is also called as a FIFO.

- FIFO's will be created on file system as device file
- First In First Out
- Unlike Pipes FIFOs are named hence called as Named Pipes
- Unlike Pipes, FIFOs can be used to communicate between unrelated processes also.
- They can be opened just like normal files using their names.

### Rules for reading a FIFO:

- A reader requesting less data than is in FIFO returns only the requested amount of data. The remaining is left for subsequent reads.
- If a process asks to read more data than is in FIFO, Returns only the available data.
- If a process asks to read, if there is no data in FIFO and if no process having open for writing, a read returns Zero immediately when O\_NDELAY flag is set.
- If O\_NDELAY flag is not set, then it will wait for data.

### Rules for writing to FIFO:

- If a process writes less than the capacity of the FIFO, the write is guaranteed to be atomic.
- If a process writes more data than the capacity of FIFO, there is no guarantee that the write operation is atomic.
- If a process writes to a FIFO, but there is no process existence that having open for reading, "**SIGPIPE**" signal is generated. And write returns with an error called "**EPIPE**".
- If that process is not handling the SIGPIPE signal, the default action is to terminate the process.

### FIFO Operations:

Condition	Normal	O_NDELAY set
Read end is open and write end is not open	Wait until Write end is open	Return Error immediately
Write end is open and read end is not open	Wait until read end is open	Returns Error immediately
Reading fifo, but no data in fifo	Wait until there is data in fifo. No process write end is open return zero	Return immediately zero
Writing to fifo when it is full	Wait until space is available, then write to fifo	Returns immediately zero



## System Calls related to FIFO's

Required Header files:

### Summary of `mknod ( )` system call

Include Files	<code>&lt;sys/types.h&gt;</code> <code>&lt;sys/stat.h&gt;</code> <code>&lt;fcntl.h&gt;</code> <code>&lt;unistd.h&gt;</code>	Manual section	2
Summary	<code>int mknod (const char *pathname, mode_t mode,</code>		
Return	Success	Failure	
	0	-1	

Here:

pathname: user defined name to the FIFO.  
 mode: Permissions to FIFO. Ex: 0777, 0744, etc.

The `dev` argument for `mknod` is used only when a character or block special file is specified. For character and block special files, the `dev` argument is used to assign the

major and minor number of the device. For nonprivileged users, the `mknod` system call can only be used to generate a FIFO. When generating a FIFO, the `dev` argument should be left as 0.

### **Opening FIFO:**

FIFO can be opened like normal files by using `open()` function. Using this function we can open FIFO in read and write modes separately as shown in below:

Summary of `mkfifo()` system call:

Include Files	<code>&lt;sys/types.h&gt;</code> <code>&lt;sys/stat.h&gt;</code>	Manual section	3
Summary	<b><code>int mkfifo (const char *pathname, mode_t mode);</code></b>		
Return	Success	Failure	
	0	-1	

Mode:

Write mode: 0

Read mode: 1

### **Reading/Writing from/to FIFO:**

Once you open FIFO you can read from and write to FIFO by using normal file system with calls like: `read()`, `write()` and `close()`.

### **Unlink FIFO:**

After using FIFO if you want to delete or unlink FIFO the following function has to be used.

### Summary of unlink() system call:

Include Files	<unistd.h>	Manual section	3
Summary	<b>int unlink ( FIFO_NAME);</b>		
Return	Success	Failure	
	0	-1	

EX: /\*\*\*\*\* WAP to send a message from child process to its parent process. \*\*\*\*\*/

```
int main() {
    int wfd, rfd, var; char buf[25];

    var = mkfifo("fifo", S_IFIFO | 0777); // FIFO will be
    created in current directory.
    if(var == -1) {
        printf("error in creating FIFO\n");
        exit(0);
    }

    switch(fork()) {

        case -1: printf("error in creating child process\n");
                exit(1);
        case 0:  printf("I am in child\n");
                wfd = open("fifo", 1);
                if(wfd == NULL) {
                    printf("error in opening fifo in write
mode:\n");
                    exit(2);
                }
                var = write(wfd, "hello to Linus Torvolds",
24);
                if(var < 0) {
                    printf("error in writing to FIFO:\n");
                    exit(3);
                }
                printf("write success to FIFO in child:\n");
                break;
        default: wait(0);
    }
}
```

```

        printf("I am in Parent\n");
        rfd = open("fifo", 0);
        if(rfd == NULL) {
            printf("error in opening FIFO in read
mode:\n");
            exit(2);
        }
        var = read(rfd, buf, 24);
        if(var <= 0) {
            printf("error in reading from
FIFO:\n");
            exit(3);
        }
        printf("read success in parent:\ndata is:
%s\n", buf);
    }
    return 0;
}

```

### **Limitations:**

- Data cannot be broadcast to multiple receivers.
- If there are multiple receivers, there is no way to direct to a specific reader or vice versa.
- Cannot be used across network
- Less secure than a Pipe, since any process with valid access permission can access data.
- Cannot store data
- No message boundaries. Data is treated as a stream of bytes.

### **Key learning:**

1. What are FIFO and Pipes?
2. How does Pipe work as an IPC?
3. What are limitations of FIFO?
4. List the system calls related to FIFOs and Pipe
5. What does Pipe() return on success.
6. How do you create a FIFO?
7. Differentiate between named and unnamed Pipes.
8. What are the advantages of FIFOs over Pipes?
9. What are the advantages of FIFO?