- PIPE

Pipe is a byte stream, i.e. there is no concept of message boundaries when using a pipe. Each read operation may read an arbitrary no. of bytes regardless of the size of bytes written by the writer. Furthermore, the data passes through the pipe sequentially, bytes are read from a pipe in exactly the order they were written. It is not possible to randomly access the data in pipe a pipe.

→ pipe are unidirectional, i.e. data can travel only in one direction, one end of the pipe is used for writing & other end is used for reading.

Reading from a pipe

~~When a process reads bytes from a pipe~~

→ By default, when a process attempts to read from a pipe that is currently empty, the read call blocks until some bytes are written into the pipe.

→ If the write end of a pipe is closed and a process tries to read, it will receive EOF Character, i.e. read() returns 0.

→ If two processes try to read from the same pipe, one process will get some of the bytes from the pipe and the other process will get the other bytes. Unless the two processes use some method to coordinate their access to the pipe, the data they read are likely to be incomplete.

writing to a pipe

-) by default, when a process attempts to write to a pipe that is currently full, the write() call blocks until there is enough space in the pipe.

-) If a process tries to write, say, 1000 bytes, and there is room for 500 bytes only, the call waits until 1000 bytes of space are available.

-) If the read end of a pipe is closed & a process tries to write, the kernel sends SIGPIPE to the writer process.

int pipe (int fd[2]);

→ A pipe is created using a pipe system call.

→ Creating a pipe is similar to ~~creating~~ opening 2 files. A successful call to pipe() returns two open file descriptors in the array 'fd'. One contains the read descriptor of the pipe, fd[0], and the other contains the write descriptor of the pipe fd[1].

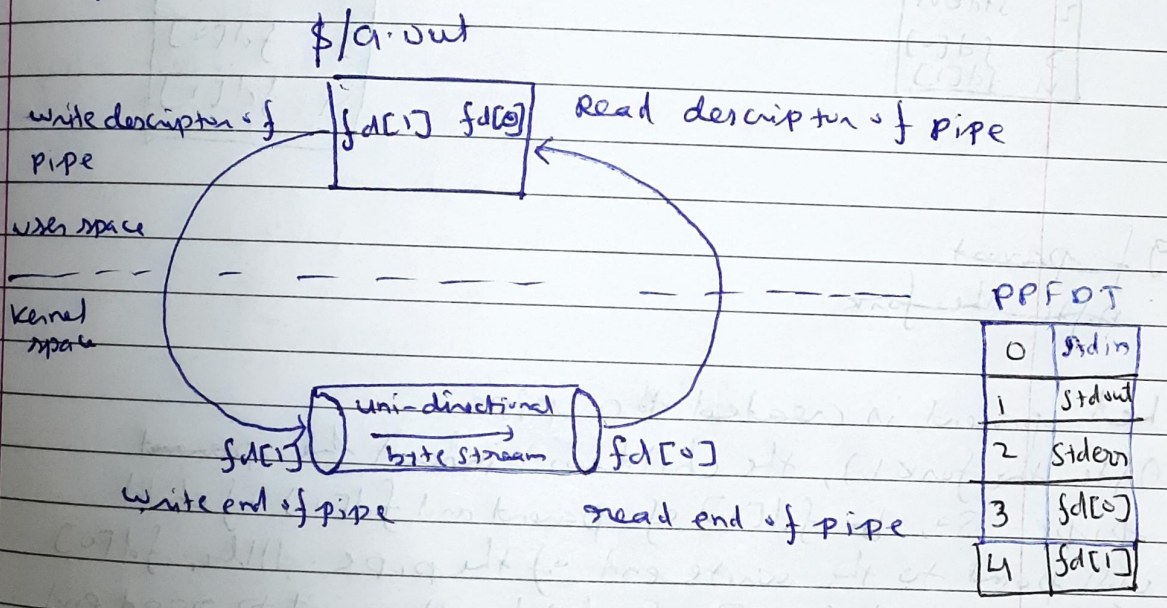
→ As with any file descriptor, we can use the read() and write() system calls to perform I/O on the pipe. Once written to the write end of a pipe, data is immediately available to be read from the read end. A read() from a pipe blocks if the pipe is empty.

→ From implementation point of view, a pipe is a fixed-size main memory circular buffer created & maintained by the kernel. The kernel handles the synchronization required for making the reader process wait when the pipe is empty & the writer process wait when the pipe is full.

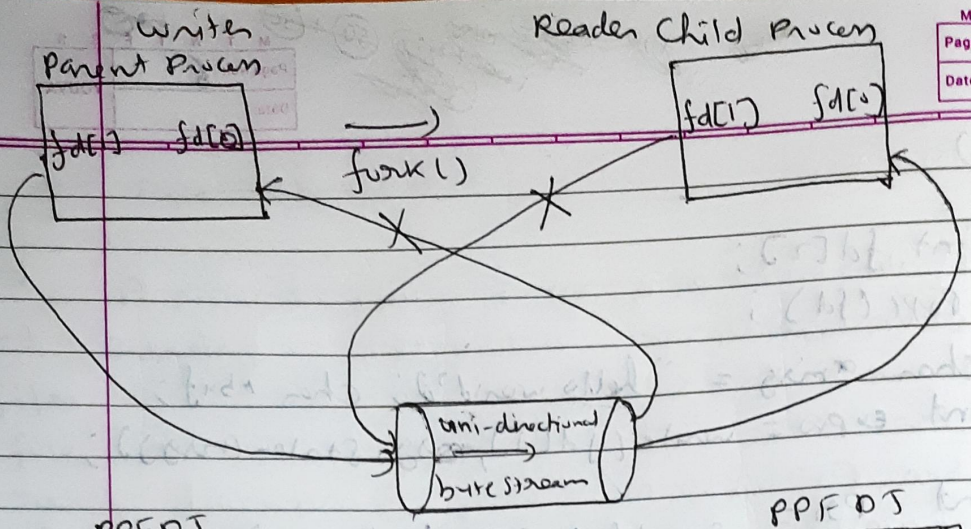
```
main()
{
    int fd[2];
    pipe(fd);
    char *msg = "hello world"; char *buf;
    int pw = write(fd[1], msg, strlen(msg));

    int pr = read(fd[0], buf, pw);
    write(1, buf, pr);

}
```



→ check the fd no. 3 & 4 in /proc



PPFDT

0	Stdin
1	Stdout
2	Stderr
3	fd[0]
4	fd[1]

PPFDT

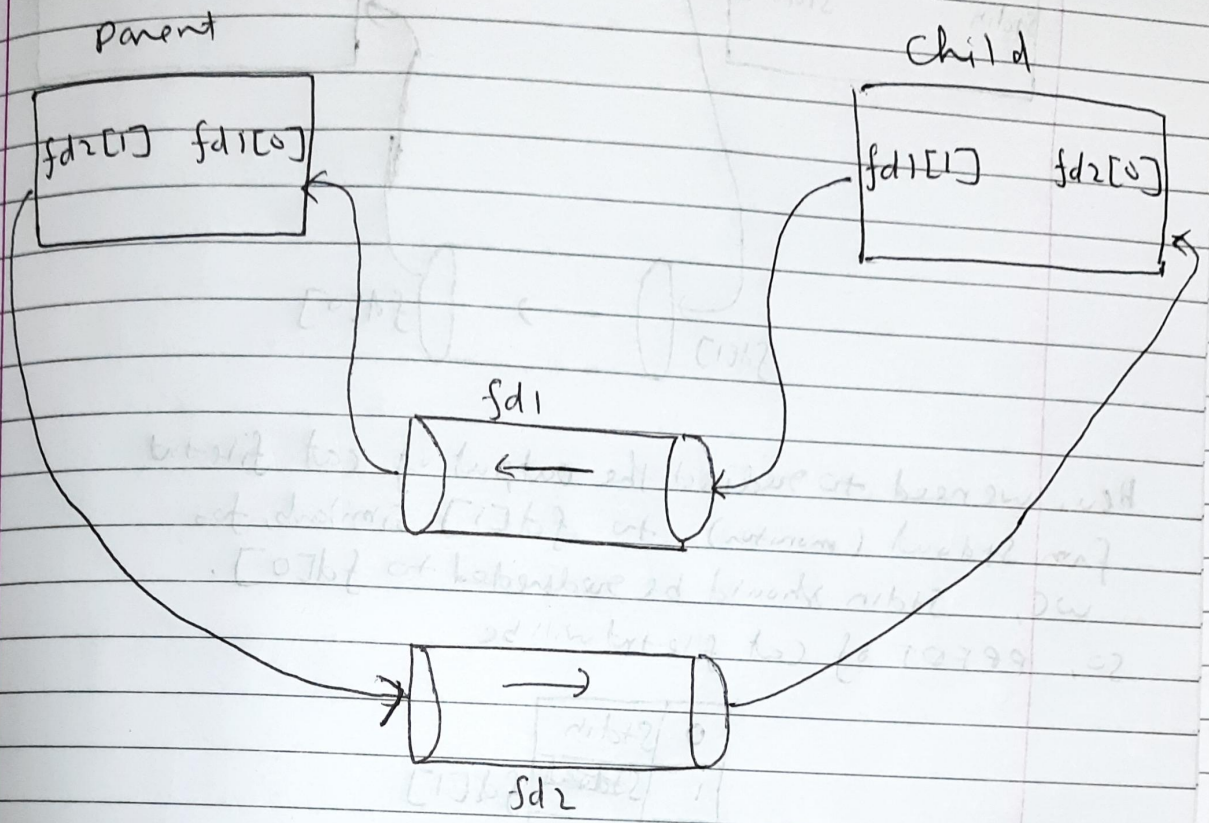
0	Stdin
1	Stdout
2	Stderr
3	fd[0]
4	fd[1]

~~If parent~~
~~After the fork~~

Let parent is created a pipe.
After the fork(), the child will inherit the ~~parent~~ pipe. So, fd[1] of parent and fd[1] of child will point to the write end of the pipe. Like, fd[0] of parent and fd[0] of child will point to read end of the pipe.
So, to make the parent process write in the pipe & parent to read in the pipe, fd[0] of parent & fd[1] of child should be ~~read~~ deleted.

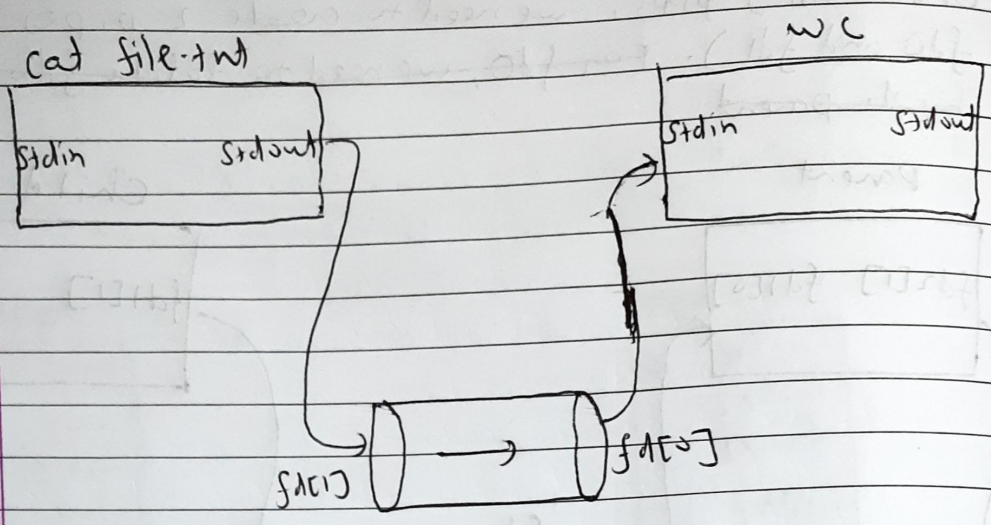
PPFDT of parent will also be inherited by the child process.

⇒ For bi-directional communication between parent & child using pipe, we need to create 2 pipes (say fd0 and fd1). ~~For fd0, we need to delete fds[0] by the parent~~



So, we need to delete fd1[1] and fd2[0] of parent and ~~fd0~~ fd1[0] and fd2[1] of the child process.

⇒ cat file.txt | wc



Here, we need to redirect the output of cat file.txt from stdout (monitor) to fd[1]. Similarly, for wc, stdin should be redirected to fd[0].

So, PPFDT of cat file.txt will be

0	stdin
1	~~stdout~~ fd[1]
2	stderr

PPFDT of wc will be

0	fd[0]
1	stdout
2	stderr

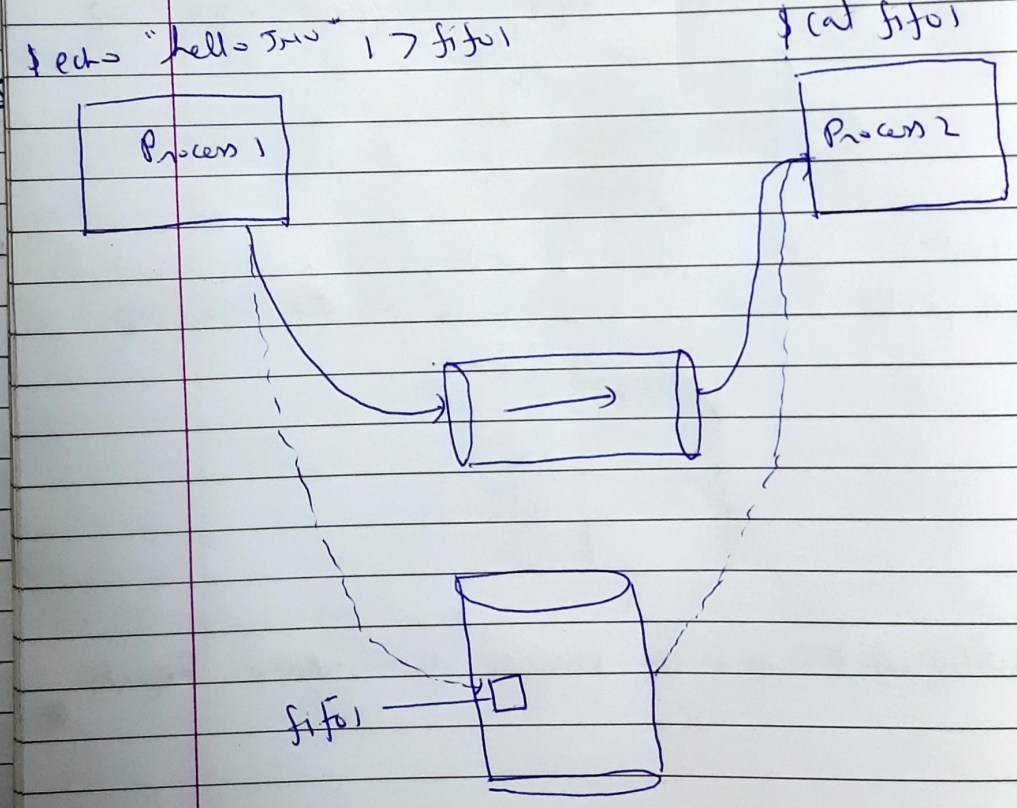
dup2 system call is used to redirect the file descriptor.

Named pipe or FIFO

→ Pipes can't be used between unrelated process

→ FIFOs are like pipe as they are stream oriented and half-duplex but they can communicate between unrelated processes.

→ A FIFO can be created by one process and can be opened by multiple processes for reading or writing. When processes are reading or writing via FIFO, kernel passes all data internally without writing it to the file system. Thus, a FIFO file has no contents on the file system; the file system entry merely serves as a reference point so that process can access the pipe using a name in the file system.



$ mkfifo fifo1

$ echo "hello Jnu" 1> fifo1

It will block the terminal because in pipe or fifo, until a process not reads the pipe, writer process will block. Like, until a process not writes, reader process will block. But, this is not the case with file reading or writing. Hence, we can say that storing in the fifo on harddisk is just for reference & not for actual storage. on ls -l, size of fifo will be shown 0.

mkfifo ()
int mkfifo (const char *pathname, mode_t mode);

→ makes a FIFO special file with name pathname. The 2nd argument mode specifies the FIFO's permissions.

mknode ()
int mknod (const char *nam, mode_t mode, dev_t device);

→ mknod() system call creates a FIFO file with name mentioned as the 1st argument.

→ 2nd argument 'mode' specifies both permissions to use & type of node to be created. It should be a combination (bitwise OR) of one of the file types (S_IFREG, S_IFIFO, S_IFCHR, S_IFBLK, S_IFSOCK) & permissions for the file.

→ for creating a named pipe, the 3rd argument is set to 0. However, to create a character or block special file we need to mention the major or minor numbers of the newly created device file.