LAB NO.: 5 **Date:**

IPC-1 : PIPE, FIFO

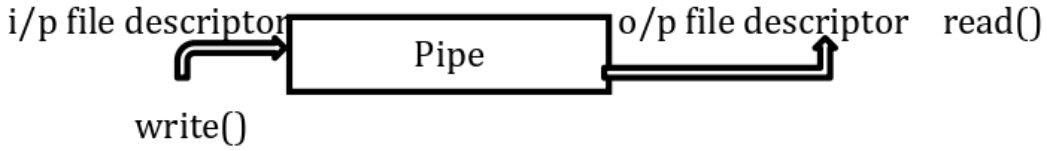
In this lab, the student will be able to:

1. Gain knowledge as to how Interprocess Communication (IPC) happens between two processes.
2. Execute programs for IPC using the different methods of pipe and fifo.

Inter-Process Communication (IPC), is the mechanism whereby one process can communicate with another process, i.e exchange of data. IPC in Linux can be implemented by using a pipe, shared memory and message queue.

Pipe

- Pipes are unidirectional byte streams that connect the standard output from one process into the standard input of another process. A pipe is created using the system call pipe that returns a pair of file descriptors.



- Call to the pipe () function which returns an array of file descriptors fd[0] and fd [1]. fd [1] connects to the write end of the pipe, and fd[0] connects to the read end of the pipe. Anything can be written to the pipe, and read from the other end in the order it came in.
- A pipe is one-directional providing one-way flow of data and it is created by the pipe() system call.

```
int pipe ( int *filedes ) ;
```

- An array of two file descriptors are returned- fd[0] which is open for reading, and fd[1] which is open for writing. It can be used only between parent and child processes.

PROTOTYPE: int pipe(int fd[2]);

RETURNS: 0 on success
-1 on error: errno = EMFILE (no free descriptors)
EMFILE (system file table is full)
EFAULT (fd array is not valid)

fd[0] is set up for reading, fd[1] is set up for writing. i.e., the first integer in the array (element 0) is set up and opened for reading, while the second integer (element 1) is set up and opened for writing.

```
#include <stdlib.h>
#include <stdio.h>    /* for printf */
#include <string.h>   /* for strlen */

int main(int argc, char **argv)
{
        int n;
        int fd[2];
        char buf[1025];
        char *data = "hello... this is sample data";
       pipe(fd);
        write(fd[1], data, strlen(data));
        if ((n = read(fd[0], buf, 1024)) >= 0) {
                buf[n] = 0;      /* terminate the string */
                printf("read %d bytes from the pipe: \"%s\"\n", n, buf);
        }
        else
                perror("read");
        exit(0);
}
```

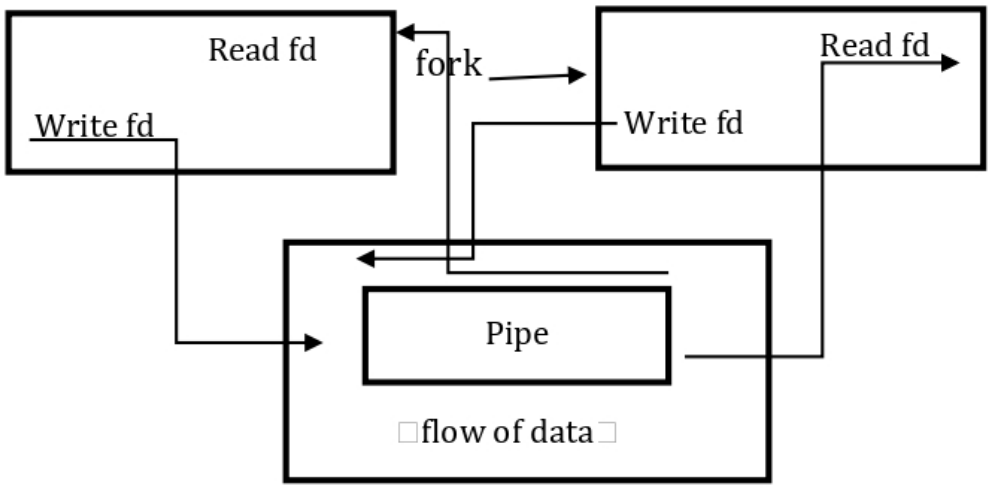


Fig. 7.1: Working of pipe in a single process which is immediately after fork()

- First, a process creates a pipe and then forks to create a copy of itself.
- The parent process closes the read end of the pipe.
- The child process closes the write end of the pipe.
- The fork system call creates a copy of the process that was executing.
- The process which executes the fork is called the parent process and the new process is which is created is called the child process.

```
#include <sys/wait.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(int argc, char *argv[])
{
   int pfd[2];
   pid_t cpid;
   char buf;
   assert(argc == 2);
   if (pipe(pfd) == -1) { perror("pipe");
```

```
    exit(EXIT_FAILURE); }
  cpid = fork();
  if (cpid == -1) { perror("fork");
  exit(EXIT_FAILURE); }

  if (cpid == 0) {    /* Child reads from pipe */
    close(pfd[1]);          /* Close unused write end */
    while (read(pfd[0], &buf, 1) > 0)
     write(STDOUT_FILENO, &buf, 1);
     write(STDOUT_FILENO, "\n", 1);
    close(pfd[0]);
    exit(EXIT_SUCCESS);

  } else {            /* Parent writes argv[1] to pipe */
    close(pfd[0]);          /* Close unused read end */
    write(pfd[1], argv[1], strlen(argv[1]));
    close(pfd[1]);          /* Reader will see EOF */
    wait(NULL);             /* Wait for child */
    exit(EXIT_SUCCESS);
  }
}
```

Named Pipes: FIFOs

Pipes can share data between related processes, i.e. processes that have been started from a common ancestor process. We can use named pipe or FIFOs to overcome this. A named pipe is a special type of file that exists as a name in the file system but behaves like the unnamed pipes we have discussed already. We can create named pipes from the command line using

```
$ mkfifo filename
```

From inside a program, we can use

```
#include <sys/types.h>
```

```
#include <sys/stat.h>

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

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
int main()
    {
        int res = mkfifo("/tmp/my_fifo", 0777);
        if (res == 0) printf("FIFO created\n");
        exit(EXIT_SUCCESS);
    }
```

We can look for the pipe with

$ **ls -lF /tmp/my_fifo**

prwxr-xr-x 1 rick users 0 July 10 14:55 /tmp/my_fifo|

Notice that the first character of output is a p, indicating a pipe. The | symbol at the end is added by the ls command's -F option and also indicates a pipe. We can remove the FIFO just like a conventional file by using the rm command, or from within a program by using the unlink system call.

Producer-Consumer Problem (PCP):

- The producer process produces information that is consumed by a consumer process. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. Two types of buffers can be used.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is fixed buffer size.
- For bounded-buffer PCP basic synchronization requirement is:
  - Producer should not write into a full buffer (i.e. producer must wait if the buffer is full)
  - Consumer should not read from an empty buffer (i.e. consumer must wait if the buffer is empty)
  - All data written by the producer must be read exactly once by the consumer

Following is a program for Producer-Consumer problem using named pipes.
//producer.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
#define TEN_MEG (1024 * 1024 * 10)

int main()
{
        int pipe_fd;
        int res;
        int open_mode = O_WRONLY;
        int bytes_sent = 0;
        char buffer[BUFFER_SIZE + 1];
        if (access(FIFO_NAME, F_OK) == -1) {
                res = mkfifo(FIFO_NAME, 0777);
                if (res != 0) {
                        fprintf(stderr, "Could not create fifo %s\n", FIFO_NAME);
                        exit(EXIT_FAILURE);
                }
        }
        printf("Process %d opening FIFO O_WRONLY\n", getpid());
        pipe_fd = open(FIFO_NAME, open_mode);
        printf("Process %d result %d\n", getpid(), pipe_fd);
        if (pipe_fd != -1) {
                while(bytes_sent < TEN_MEG) {
                        res = write(pipe_fd, buffer, BUFFER_SIZE);
                        if (res == -1) {
```

```
                        fprintf(stderr, "Write error on pipe\n");
                        exit(EXIT_FAILURE);
                    }
                    bytes_sent += res;
                }
                (void)close(pipe_fd);
        }
        else {
                exit(EXIT_FAILURE);
        }
        printf("Process %d finished\n", getpid());
        exit(EXIT_SUCCESS);
}

//consumer.c

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO_NAME "/tmp/my_fifo"
#define BUFFER_SIZE PIPE_BUF
int main()
{
        int pipe_fd;
        int res;
        int open_mode = O_RDONLY;
        char buffer[BUFFER_SIZE + 1];
        int bytes_read = 0;
        memset(buffer, '\0', sizeof(buffer));
        printf("Process %d opening FIFO O_RDONLY\n", getpid());
        pipe_fd = open(FIFO_NAME, open_mode);
        printf("Process %d result %d\n", getpid(), pipe_fd);
```

```
        if (pipe_fd != -1) {
                do {
                        res = read(pipe_fd, buffer, BUFFER_SIZE);
                        bytes_read += res;
                } while (res > 0);
                (void)close(pipe_fd);
        }
        else {
                exit(EXIT_FAILURE);
        }
        printf("Process %d finished, %d bytes read\n", getpid(), bytes_read);
        exit(EXIT_SUCCESS);
}
```

The Readers-Writers Problem:

- Concurrent processes share a file, record, or other resources
- Some may read-only (readers), some may both read and write (writers)
- Two concurrent reads have no adverse effects
- Problems if
  - concurrent reads and writes
  - multiple writes

Two Variations

- First Readers-Writers problem: No reader be kept waiting unless a writer has already obtained exclusive write permissions (Readers have high priority)
- Second Readers-Writers problem: If a writer is waiting/ready, no new readers may start reading (Writers have high priority)

Lab Exercises:

1. Write a producer and consumer program in C using the FIFO queue. The producer should write a set of 4 integers into the FIFO queue and the consumer should display the 4 integers.
2. Demonstrate creation, writing to, and reading from a pipe.
3. Write a C program to implement one side of FIFO

4. Write a C program reading and writing a binary files in C

Additional Exercises:

1. Demonstrate creation of a process that writes through a pipe while the parent process reads from it.
2. Demonstrate second readers-writers problem using FIFO