National University of Computer and Emerging Sciences



Laboratory Manual

for

Operating Systems Lab

(CL-220)

|  |  |
| --- | --- |
|  | Mr. Ibtahim Nadir |
| Lab Instructor(s) | Zumirrah Khalid |
| Section | 4E |
| Semester | Spring 2021 |

Department of Computer Science

FAST-NU, Lahore, Pakistan

## Objectives

In this lab, students will practice:

1. Inter-process communication using unnamed pipes
2. Reading and writing files using open, write, and read system calls

Important Note:

* Comment your code intelligently.
* Indent your code properly.
* Use meaningful variable names.
* Use meaningful prompt lines/labels for input/output.
* Use meaningful project and C/C++ file name

# Pipes

Ordinary pipes allow two processes to communicate in standard producer consumer fashion: the producer writes to one end of the pipe (the write-end) and the consumer reads from the other end (the read-end). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used with each pipe sending data in a different direction.

References: Operating System concepts Page no 142 section 3.6.3.1

A pipe has a read end and a write end.

Data written to the write end of a pipe can be read from the read end of the pipe.

# Creating a pipe

On UNIX and Linux systems, ordinary pipes are constructed using the function

• int pipe(int fd[2]) -- creates a pipe

• returns two file descriptors, fd[0], fd[1].

• fd[0] is the read-end of the pipe

• fd[1] is the write-end.

• fd[0] is opened for reading,

• fd[1] for writing. pipe() returns 0 on success, -1 on failure and sets errno accordingly.

• The standard programming model is that after the pipe has been set up, two (or more) cooperative processes will be created by a fork and data will be passed using read() and write().

• Pipes opened with pipe() should be closed with

close(int fd).

Reference: http://linux.die.net/man/2/pipe

# Example

Listing 1

#include <iostream>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

using namespace std;

int main()

{

char buf[5];

int fd = open ("file.txt" , O\_RDONLY, S\_IRUSR);

ssize\_t size = read(fd, buf, sizeof (buf));

write(2, buf, size);

while (size > 0)

{

size = read(fd, buf, sizeof (buf));

write(2, buf, size);

}

return 0;

}

Listing 2

#include <sys/types.h>

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#define BUFFER\_SIZE 25

#define READ\_END 0

#define WRITE\_END 1

int main(void)

{

char write\_msg[BUFFER\_SIZE] = "Greetings";

char read\_msg[BUFFER\_SIZE];

int fd[2];

pid\_t pid;

/\* create the pipe \*/

if (pipe(fd) == -1) {

fprintf (stderr,"Pipe failed");

return 1;

}

/\* fork a child process \*/

pid = fork();

if (pid < 0) { /\* error occurred \*/

fprintf(stderr, "Fork Failed");

return 1;

}

if (pid > 0) { /\* parent process \*/

/\* close the unused end of the pipe \*/

close(fd[READ\_END]);

/\* write to the pipe \*/

write(fd[WRITE\_END], write\_msg, strlen(write\_msg)+1);

/\* close the write end of the pipe \*/

close(fd[WRITE\_END]);

}

else { /\* child process \*/

/\* close the unused end of the pipe \*/

close(fd[WRITE\_END]);

/\* read from the pipe \*/

read( fd[READ\_END], read\_msg, BUFFER\_SIZE);

printf("read %s",read\_msg);

/\* close the write end of the pipe \*/

close(fd[READ\_END]);

}

return 0;

}

# Failure

When pipe() System Call Fails:

The pipe() system call fails for many reasons, including the following:

1 At least two slots are not empty in the FDT—too many files or pipes are open in the process.

# Open System call for Filing

Open system call is used for opening a file.

**int open(const char \****pathname***, int** *flags***, mode\_t** *mode***);**

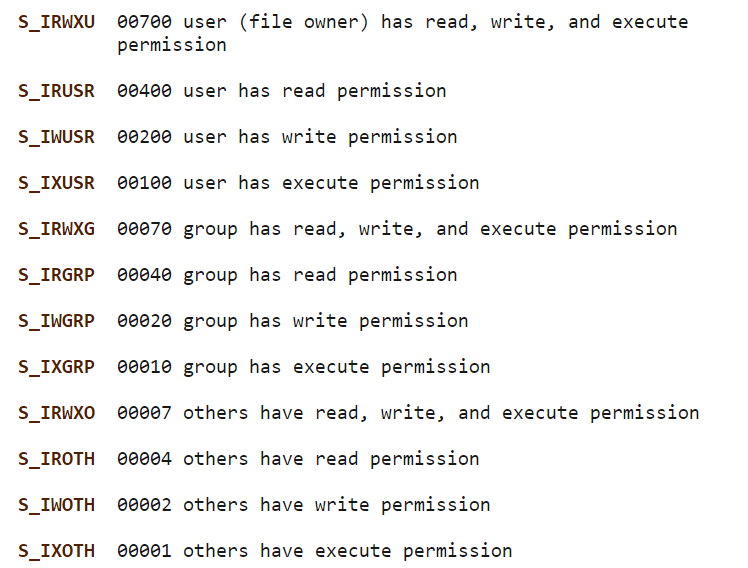
1. pathname is a file name
2. The argument *flags* must include one of the following *access modes*

**O\_RDONLY**, **O\_WRONLY**, or **O\_RDWR**. These request opening the file in read-only, write-only, or read/write modes, respectively. Apart from above, flags can also have any of the following:

1. **O\_APPEND (file is opened in append mode)**
2. **O\_CREAT** (If *pathname* does not exist, create it as a regular file.)
3. **O\_EXCL** Ensure that this call creates the file: if this flag is specified in conjunction with **O\_CREAT**, and *pathname* already exists, then **open**() fails.

**Note: to use two flags at once use bitwise OR operator, i.e., O\_WRONLY | O\_CREAT**

1. **Mode is only required when a new file is created and is used to set permissions on the new file**



# Inlab Question

Design a program using ordinary pipes in which parent process sends a message from a file named file.txt to a child process, and the child process remove the occurrences of all the special characters including &,@,#,%,\*,?,&,$,”,and ~. And send the modified version back to the parent process and the parent process writes the modified data to the file updated.txt . This will require using two pipes, one for sending the original message from the first to the second process, and the other for sending the modified message from the second back to the first process.

Note: Use only read, write and open system calls. Use of Cin, cout, prinf, ofstream, ifstream etc. will result in zero marks.