Narendra Pratap Singh
Roll No. 35
Department of Computer Science
M.Sc. Computer Science (2nd Semester)

Advanced Operating Systems

Documentation for Coding Assignment on System Calls

OVERVIEW

Using a modular program in C Language for file handling, i.e., for creating, reading, writing to a file, creating named and unnamed pipes and performing file copying action through them, that too only using system calls and visualizing how the system calls work.

GOALS

1. **Create** a file/named pipe, a user should give permissions for the file.
2. **Read** data from a file, where a user can specify (i) the amount of data to read (in bytes), and (ii) from where to read the data in the file (e.g., read 100 bytes from a file f1.txt from the beginning)
3. **Write** data to a file, where a user can specify (i) the amount of data to write (in bytes), and (ii) from where to start writing the data to the file (e.g., write 100 bytes from a file f1.txt from offset 10 from current position)
4. **Display statistical information** of a specified file including owner, permissions, inode and all time stamps.
5. Create an **unnamed pipe** designed for **copying** a file's content (say 'a.txt') to another file ('b.txt').
6. Create a **named pipe** to help **communicate** between two processes, where a user can specify the purpose of the pipe i.e. reading or writing

Command Used to Execute -

Compilation - `gcc -o app file_handling.c library.c`

Execution - `./app`

Different Functions of the Program -

Create File

Take file name and permissions to be assigned to the file from user through command line, since the permissions entered by user in command line is in form of string, it is converted to octal using a function **strtol()** (present in **stdlib.h**) and datatype **mode_t** (present in **sys/stat.h**), generally used for representing file permissions in Unix-like systems, is used to store the permissions.

These parameters are passed to a user defined function, **createFile()** in which system call **creat(filename, permissions)** is used.

Command - `./app create nps1.txt 0777`

Read File

File name, amount of data to read, the offset from where to start reading and whence, which is used in the **lseek()** function to set the offset as per user's choice, are all taken as input by the user through the command line. Since the amount of data, offset and whence will be accepted as strings in the command line, we need to convert them into their particular data types. That is done via functions **atol()** (converting string to long) and **atoi()** (converting string to integer).

These parameters are passed to a user defined function, **readFile()** in which system calls **open(), lseek()** and **read()** are used to open file, set offset and read data respectively.
This function also provides the functionality to create a new file if the opened file does not exist in the system.

A dynamically allocated memory buffer is created to store the data read. And at the end of the buffer array, a null character, **\0** is entered to ensure that the buffer is properly terminated as a string.

Command - `./app read nps1.txt 10 0 0`

Write to a File

File name, the data to write to the file, the offset from where to start writing and whence, which is used in the **lseek()** function to set the offset as per user's choice, are all taken as input by the user through the command line. Since the offset and whence will be accepted as strings in the command line, we need to convert them into their particular data types. That is done via functions **atol()** (converting string to long) and **atoi()** (converting string to integer).

As the size of data is needed in system call **write()**, **strlen()** function is used to calculate size of the input for data.

All these parameters are then passed to a user defined function, **writeFile()**, in which system calls **open()**, **lseek()** and **write()** are used to perform the specified action. This function also provides the functionality to create a new file if the one being accessed does not exist in the system, for which, file name and permissions will be asked from the user to give as input.

Command - `./app write nps1.txt "My name is Narendra Pratap Singh" 0 0`

Display Statistical information of a File

For this, only file name is taken as input and passed to a user defined function **fileStats()** to show the required information in which system call **stat()** and a structure defined in **<sys/stat.h>** header, containing various file information fields like size, permissions, timestamps and more are stored.

Command - `./app file_info nps1.txt`

Create a named pipe

For this, we need a pipe name and the user needs to specify the purpose of the pipe (**read** or **write**). Both arguments are entered through the command line by the user and passed to a user defined function, **createNamedPipe()** in which first we check the purpose of the pipe and accordingly proceed.

When the user has the purpose to write, the pipe is created using **mkfifo()**, then we take data input from the user and then **open()** the pipe and **write()** data to it.

Now if we want to use this pipe to read, we will open another terminal, just like we previously opened pipe for write purposes, now we will write the purpose to read and input the same pipe name like before. This will now show the data that was written to the pipe and has now been read from it.

Command - `./app create_named_pipe pipe1 write`

Now this will ask user to enter data to write to pipe, input the data and press enter and now open another terminal, and write following command -

```
./app create_named_pipe pipe1 read
```

This will now display the data written to this named pipe.

Copy file using pipe

For this, we take the source file name and destination file name as an input from the user and pass them as a parameter to a user defined function **copyFilesUsingPipe()** in which system calls **pipe()**, **fork()**, **open()**, **read()**, **write()** and **waitpid()** are used.

We take an array with 2 elements, representing 2 descriptors i.e., 0 for read and 1 for write. Then the pipe is created using **pipe()** system call.

Using **fork()** command, a child process is being made which reads from pipe and writes to destination which closes the unused write end of the file using **close()** command. Then the destination file is opened, data is read using **read()** into a buffer array and the number of bytes read is stored. The data is then written to the destination file using **write()** and now read end of pipe is also closed.

Now if instead of the child process, the parent process reads and writes using pipe, close the read end of pipe, open the source file and read into the buffer array and then write to pipe using **write()**.

After reading and writing are completed, the parent process closes the write end of the pipe and the source file.

Command - `./app copy_with_pipe nps1.txt nps2.txt`