EXPERIMENT NO: 4

I/O SYSTEM CALLS OF LINUX

AIM

To perform file I/O using system calls in Linux such as creating, opening, reading, writing, and closing files using low-level system calls.

1) Creating a New File

Algorithm

1. **Start** the program.
2. **Include the necessary header files**:
   `<fcntl.h>` for file control operations.
   `<stdio.h>` for input and output functions.
   `<stdlib.h>` for program control functions like `exit()`.
3. **Define the file name** as `"new_file.txt"`.
4. **Call the `open()` system call** with the following parameters:
   (If successful, open() returns a file descriptor (an integer used to reference the file) and If unsuccessful, it returns -1)
   File name (`"new_file.txt"`).
   Flags (`O_CREAT | O_WRONLY`):
   a. `O_CREAT`: Create the file if it does not exist.
   b. `O_WRONLY`: Open the file in write-only mode.
   File permissions (`0644`):
   c. Owner can read and write.
   d. Group and others can only read.
5. **Check the return value of `open()`**:
   If the return value (`fd`) is `-1`, an error occurred:
   a. Print the error message using `perror()`.
   b. Exit the program with a failure status using `exit(EXIT_FAILURE)`.
6. If the file is created successfully:
   Print a success message with the file name using `printf()`.
7. **Close the file** using the `close()` system call to release system resources.
8. **Return 0** to indicate successful program execution.
9. **End** the program.

Program

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
```

```
                                // Define the name of the file to be created.
    const char *filename = "new_file.txt";

                                // Open (or create if it doesn't exist) the file in write-only mode.
                                // Set the file permissions to 0644 (owner: read-write, others: read-only).
    int fd = open(filename, O_CREAT | O_WRONLY, 0644);

                                // Check if the file descriptor indicates an error (-1).
    if (fd == -1)
    {
                                // Print an error message based on the system error (stored in `errno`).
        perror("Error creating file");

                                // Exit the program with a failure status.
        exit(EXIT_FAILURE);
    }

                                // If the file created successfully, print success message with the filename.
    printf("File '%s' created successfully.\n", filename);

                                // Close the file to free up resources.
    close(fd);

                                // Return 0 to indicate the program executed successfully.
    return 0;
}
```

2) Program to create, open, write and read a file.

Algorithm:

Step 1: Start
Begin the program execution.
Step 2: Include Header Files
Include the following header files:
- <stdio.h>: For standard input and output operations (e.g., printf() and perror()).
- <stdlib.h>: For functions like exit().
- <fcntl.h>: For file control system calls like open().
- <unistd.h>: For file I/O functions like read(), write(), and close().
- <string.h>: For string manipulation functions like strlen().

Step 3: Declare and Initialize Variables
Define the filename as a constant string filename = "my_file.txt".
Declare an integer variable fd to hold the file descriptor.

Step 4: Create and Open the File for Writing

Use the open() system call to:

- Create a new file named "my_file.txt" (if it doesn't exist).
- Open it in write-only mode (O_WRONLY) and create mode (O_CREAT).
- Set file permissions to 0644 (read-write for the owner, read-only for others).

Error Handling for File Creation:

- If open() returns -1, print an error message using perror().
- Terminate the program with a failure status using exit(EXIT_FAILURE).

Step 5: Write Data to the File

Define the string data = "Hello, world!" to be written to the file.

Use the write() system call to write the contents of data to the file:

- Pass the file descriptor fd, the data, and the length of the string (strlen(data)).

Handle Write Success/Failure:

- If write() returns a positive value, print the number of bytes written.
- Handle errors (if any) as needed.

Step 6: Close the File

- Use the close() system call to close the file after writing.

Step 7: Reopen the File for Reading

- Use the open() system call to open "my_file.txt" in read-only mode (O_RDONLY).

Error Handling for File Reopening:

- If open() returns -1, print an error message using perror().
- Terminate the program with a failure status using exit(EXIT_FAILURE).

Step 8: Read Data from the File

- Define a buffer array buffer[100] to store the data read from the file.
- Use the read() system call to read data into the buffer:
- Pass the file descriptor fd, the buffer, and the size of the buffer (sizeof(buffer)).

Handle Read Success/Failure:

- If read() returns a positive value, print the number of bytes read and the content of the buffer.
- If the end of the file is reached, handle it as needed.

Step 9: Close the File Again

- Use the close() system call to close the file after reading.

Step 10: End the Program

- Return 0 to indicate successful execution.
- End the program.

Program

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
int main() {
   const char *filename = "my_file.txt";
   int fd; // File descriptor

                    // Create a new file (if it doesn't exist) and open it for writing
```

```
    fd = open(filename, O_CREAT | O_WRONLY, 0644);
    if (fd == -1) {
        perror("Error creating/opening file");
        exit(EXIT_FAILURE);
    }

                        // Write data to the file
    const char *data = "Hello, world!";
    ssize_t bytes_written = write(fd, data, strlen(data));
    if (bytes_written > 0) {
        printf("Wrote %ld bytes to the file.\n", bytes_written);
    }

                        // Close the file
    close(fd);

                        // Reopen the file for reading
    fd = open(filename, O_RDONLY);
    if (fd == -1) {
        perror("Error opening file for reading");
        exit(EXIT_FAILURE);
    }

                        // Read data from the file
    char buffer[100];
    ssize_t bytes_read = read(fd, buffer, sizeof(buffer));
    if (bytes_read > 0) {
        printf("Read %ld bytes from the file: %s\n", bytes_read, buffer);
    }

                        // Close the file again
    close(fd);

    return 0;
}
```

RESULT

The program to implement file I/O using system calls in Linux has been implemented and output is verified.