Documentation

This is a modular program in C that accepts only command-line arguments to carry out the following tasks:

> Creating a file/named pipe, a user should give permissions for the file.
>
> Read data from a file, where a user can specify the amount of data to read (in bytes), and from where to read the data in the file.
>
> Write data to a file, where a user can specify the amount of data to write, and 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
>
> Display statistical information of a specified file including owner, permissions, inode and all time stamps.
>
> Create an unnamed pipe designed for copying a file's content to another file. And also copy the contents of a file to another file without using pipe
>
> 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.

For each of the tasks, we have different functions.

TO COMPILE AND RUN THE CODE-

- **To compile the code**: gcc <program_name>.c -o <program_name>
  this will generate an executable file with name <program_name>
- **To run the code**: ./program_name <task> <arguments>

The creation of file is done using the function

```
void createFile(char *filename, mode_t mode) {
    int result = mknod(filename, S_IFREG | mode, 0);
    if (result == -1) {
        perror("Error creating file");
        exit(1);
    }
    printf("File '%s' created successfully.\n", filename);
}
```

For this particular task of creating file, one needs to give command :

./program_name create-file filename.txt 0666

where create-file is the task, myfile.txt is the name of file that the user wants to create and 0666 denotes the permission code.

The **mknod** function is used to create the file, but the permissions are not explicitly set in the code. The default permissions of the file will be determined by the system or the umask value.

The **S_IFREG** flag is used to indicate that the file type is a regular file. The mode is the permission mode for the file, which specifies the file's access permissions (e.g., read, write, execute) for the owner, group, and others. The | operator is a bitwise OR operation that combines the file type flag with the permission mode.

'0': This argument represents the device number, which is not relevant for regular files. For special files or named pipes, you would specify the appropriate device number.

The task 2 of reading the data from a file is done with function:

```
void readFile(char *filename, off_t offset, size_t size) {
    int file = open(filename, O_RDONLY);
    if (file == -1) {
```

```c
        perror("Error opening file");
        exit(1);
    }
    char *buffer = malloc(size);
    if (buffer == NULL) {
        perror("Error allocating memory");
        exit(1);
    }
    ssize_t bytesRead = pread(file, buffer, size, offset);
    if (bytesRead == -1) {
        perror("Error reading file");
        exit(1);
    }
    printf("Read %zd bytes from file '%s':\n", bytesRead, filename);
    //printf("%.*s\n", (int)bytesRead, buffer);
    fwrite(buffer, 1, bytesRead, stdout);
    printf("\n");
    free(buffer);
    close(file);
}
```

Command line for performing this task:

```
./program_name read-file filename.txt <offset position/starting point to read> <bytes>
```

int file = open(filename, O_RDONLY); - This line opens the file specified by the filename parameter in read-only mode. The open function returns a file descriptor, which is stored in the file variable. If the file fails to open, the function prints an error message using perror and exits.

char *buffer = malloc(size); - This line dynamically allocates memory of size size to a buffer using the malloc function. If the allocation fails (i.e., buffer is NULL), an error message is printed using perror, and the program exits.

ssize_t bytesRead = pread(file, buffer, size, offset); - This line reads size bytes from the file descriptor file, starting at the specified offset, into the buffer using the pread function. The number of bytes actually read is stored in the bytesRead variable. If an error occurs during reading, an error message is printed using perror, and the program exits.

The ssize_t type is used to handle values related to file I/O operations, such as the number of bytes read or written. It is defined in the <sys/types.h> header file. ssize_t allows for the representation of error conditions or negative values. For example, if a read operation fails, the ssize_t value returned may be -1 to indicate an error. Using an int instead of ssize_t would require additional checks or conventions to handle such error conditions.

free(buffer); - This line frees the dynamically allocated memory for the buffer.

Task 3 of writing data to a file is done using this function:

```c
void writeFile(char *filename, off_t offset, char *data) {
    int file = open(filename, O_WRONLY);
    if (file == -1) {
        perror("Error opening file");
        exit(1);
    }
    ssize_t bytesWritten = pwrite(file, data, strlen(data), offset);
    if (bytesWritten == -1) {
        perror("Error writing to file");
        exit(1);
    }
     printf("Written %zd bytes to file '%s' at offset %ld.\n", bytesWritten, filename,
offset);
    close(file);
}
```

Command line to perform this task:

./program_name write-file filename.txt <offset> <string message>

int file = open(filename, O_WRONLY); - This line opens the file specified by the filename parameter in write-only mode. The open function returns a file descriptor, which is stored in the file variable. If the file fails to open, the function prints an error message using perror and exits.

ssize_t bytesWritten = pwrite(file, data, strlen(data), offset); - This line writes the contents of the data buffer to the file descriptor file using the pwrite function. It writes the string data with a length of strlen(data) bytes, starting from the specified offset. The number of bytes actually written is stored in the bytesWritten variable. If an error occurs during writing, an error message is printed using perror, and the program exits.

Task 4 of displaying the statistical information of a file is performed using the function:

```
void displayFileInfo(char *filename) {
    struct stat fileStat;
    int result = stat(filename, &fileStat);
    if (result == -1) {
        perror("Error getting file information");
        exit(1);
    }
    printf("File Information for '%s':\n", filename);
    printf("Owner: %d\n", fileStat.st_uid);
    printf("Permissions: %o\n", fileStat.st_mode & 0777);
    printf("Inode: %ld\n", fileStat.st_ino);
    printf("Last Access Time: %ld\n", fileStat.st_atime);
    printf("Last Modification Time: %ld\n", fileStat.st_mtime);
    printf("Last Status Change Time: %ld\n", fileStat.st_ctime);
}
```

Command line to perform this task:

./program_name display-file-info filename.txt

struct stat fileStat; - The struct stat structure is used to store information about a file, such as its size, ownership, permissions, timestamps, etc. It is defined in the <sys/stat.h> header file.

int result = stat(filename, &fileStat); - This line calls the stat function to retrieve information about the file specified by the filename parameter. The stat function fills the fileStat structure with the file's information. The return value of stat is stored in the result variable. If an error occurs during the stat call, an error message is printed using perror, and the program exits.

printf("Owner: %d\n", fileStat.st_uid); - This line prints the owner of the file, represented by the user ID (st_uid member of fileStat structure).

printf("Permissions: %o\n", fileStat.st_mode & 0777); - This line prints the file permissions in octal format. It uses a bitwise AND operation to extract the permission bits from the st_mode member of fileStat structure.

Task 5 of copying file content

```
void copyFileContent(char *sourceFilename, char *destinationFilename) {
    int sourceFile = open(sourceFilename, O_RDONLY);
    if (sourceFile == -1) {
        perror("Error opening source file");
        exit(1);
    }
    int destinationFile = open(destinationFilename, O_WRONLY | O_CREAT, 0644);
    if (destinationFile == -1) {
        perror("Error opening destination file");
        exit(1);
    }
    char buffer[4096];
    ssize_t bytesRead, bytesWritten;
    while ((bytesRead = read(sourceFile, buffer, sizeof(buffer))) > 0) {
        bytesWritten = write(destinationFile, buffer, bytesRead);
        if (bytesWritten == -1) {
            perror("Error writing to destination file");
            exit(1);
        }
    }
    if (bytesRead == -1) {
        perror("Error reading from source file");


 exit(1);
    }
   printf("File '%s' copied to '%s' successfully.\n", sourceFilename, destinationFilename);
    close(sourceFile);
    close(destinationFile);
}
```

Command line to use this:

./program_name copy-file <source_file.txt> <destination_file.txt>

while ((bytesRead = read(sourceFile, buffer, sizeof(buffer))) > 0) - This line reads data from the source file into the buffer using the read function. The read function returns the number of bytes read, which is stored in the bytesRead variable. The loop continues as long as data is read successfully (i.e., bytesRead is greater than 0). If we put the condition >= 0 , it goes into infinite loop as the function initial value of bytesRead will be equal to 0.

bytesWritten = write(destinationFile, buffer, bytesRead); - This line writes the data from the buffer to the destination file using the write function. The write function returns the number of bytes written, which is stored in the bytesWritten variable. If an error occurs during writing, an error message is printed using perror, and the program exits.

O_CREAT: This flag specifies that the file should be created if it doesn't exist. If the file already exists, this flag has no effect.

0644: This is an octal representation of file permissions. The leading 0 indicates an octal value. 0644 means that the owner has read and write permissions (rw-), while the group and others have read-only permissions (r--). These permissions are commonly used for regular files.

Task 6 of copying contents of a file using pipe:

```
void copyFileContentUsingPipe(char *sourceFilename, char *destinationFilename) {
    int pipefd[2];
    if (pipe(pipefd) == -1) {
        perror("Error creating pipe");
        exit(1);
    }

    pid_t pid = fork();
    if (pid == -1) {
        perror("Error forking process");
        exit(1);
    } else if (pid == 0) {
        // Child process - read from pipe and write to destination file
        close(pipefd[1]); // Close the write end of the pipe

        int destinationFile = open(destinationFilename, O_WRONLY | O_CREAT, 0644);
        if (destinationFile == -1) {
            perror("Error opening destination file");
            exit(1);
        }

        char buffer[4096];
        ssize_t bytesRead, bytesWritten;
        while ((bytesRead = read(pipefd[0], buffer, sizeof(buffer))) > 0) {
            bytesWritten = write(destinationFile, buffer, bytesRead);
            if (bytesWritten == -1) {
                perror("Error writing to destination file");
                exit(1);
            }
        }

        if (bytesRead == -1) {
            perror("Error reading from pipe");
            exit(1);
        }

        close(pipefd[0]); // Close the read end of the pipe
        close(destinationFile);
```

```
            printf("File   '%s'   copied   to   '%s'   successfully.\n",   sourceFilename,
destinationFilename);

        exit(0);
    } else {
        // Parent process - read from source file and write to pipe
        close(pipefd[0]); // Close the read end of the pipe

        int sourceFile = open(sourceFilename, O_RDONLY);
        if (sourceFile == -1) {
            perror("Error opening source file");
            exit(1);
        }

        char buffer[4096];
        ssize_t bytesRead, bytesWritten;
        while ((bytesRead = read(sourceFile, buffer, sizeof(buffer))) > 0) {
            bytesWritten = write(pipefd[1], buffer, bytesRead);
            if (bytesWritten == -1) {
                perror("Error writing to pipe");
                exit(1);
            }
        }

        if (bytesRead == -1) {
            perror("Error reading from source file");
            exit(1);
        }

        close(pipefd[1]); // Close the write end of the pipe
        close(sourceFile);
        wait(NULL); // Wait for the child process to finish

        exit(0);
    }
}
```

Command to perform this task:

```
./program_name copy-file-by-pipe <source.txt> <destination.txt>
```

int pipefd[2]; - This line declares an array pipefd of size 2 to hold the file descriptors of the pipe. pipefd[0] is used for reading from the pipe, and pipefd[1] is used for writing to the pipe.

if (pipe(pipefd) == -1) {...} - This line creates a pipe using the pipe function. If the pipe creation fails, the function prints an error message using perror and exits.

pid_t pid = fork(); - This line forks the process into a parent and a child process. The fork function creates a new process by duplicating the existing process. The return value pid indicates the result of the fork call.

if (pid == -1) {...} - This line checks if there was an error during the fork call. If pid is -1, it indicates an error, and the function prints an error message using perror and exits.

else if (pid == 0) {...} - This line represents the code block executed by the child process. The child process reads from the pipe and writes to the destination file.

close(pipefd[1]); - This line closes the write end of the pipe in the child process.

int destinationFile = open(destinationFilename, O_WRONLY | O_CREAT, 0644); - This line opens the destination file for writing in the child process. If the file doesn't exist, it is created with the specified permissions (0644). If the file opening fails, an error message is printed using perror, and the program exits.

The child process then reads from the pipe using read(pipefd[0], buffer, sizeof(buffer)) and writes the data to the destination file using write(destinationFile, buffer, bytesRead) in a loop until no more data is read from the pipe.

close(pipefd[0]); - This line closes the read end of the pipe in the child process.

close(destinationFile); - This line closes the file descriptor of the destination file in the child process.

exit(0); - This line exits the child process.

The code block after the else statement is executed by the parent process. It reads from the source file and writes to the pipe.

close(pipefd[0]); - This line closes the read end of the pipe in the parent process.

int sourceFile = open(sourceFilename, O_RDONLY); - This line opens the source file for reading in the parent process. If the file opening fails, an error message is printed using perror, and the program exits.

The parent process reads from the source file using read(sourceFile, buffer, sizeof(buffer)) and writes the data to the pipe using write(pipefd[1], buffer, bytesRead) in a loop until no more data is read from the source file.

close(pipefd[1]); - This line closes the write end of the pipe in the parent process.

wait(NULL); - This line makes the parent process wait for the child process to finish.

Task 7 for creating a named pipe is done using:

```
void createNamedPipe(char *pipeName, int mode) {
    int result = mkfifo(pipeName, mode);
    if (result == -1) {
        perror("Error creating named pipe");
        exit(1);
    }
    printf("Named pipe '%s' created successfully.\n", pipeName);
}
```

Command to perform this task:

```
./program_name create-named-pipe <pipe_name> <permission_mode>
```

int result = mkfifo(pipeName, mode); - This line creates a named pipe with the specified name and mode using the mkfifo function. The pipeName parameter specifies the name of the pipe, and the mode parameter specifies the permissions of the pipe. The mkfifo function returns 0 on success or -1 on failure. The return value is stored in the result variable.

Task 8 for communicating through pipe is done using:

```
void communicateThroughPipe(char *pipeName, int mode) {
    int pipe;
    char buffer[4096];

    if (mode == O_RDONLY) {
```

```c
        pipe = open(pipeName, O_RDONLY);
        printf("Reading from named pipe '%s'.\n", pipeName);
        ssize_t bytesRead;
        while ((bytesRead = read(pipe, buffer, sizeof(buffer))) > 0) {
            printf("Received %zd bytes from named pipe:\n", bytesRead);
            printf("%.*s\n", (int)bytesRead, buffer);
        }
        if (bytesRead == -1) {
            perror("Error reading from named pipe");
            close(pipe);
            exit(1);
        }
        close(pipe);
    }
    else if (mode == O_WRONLY) {
        pipe = open(pipeName, O_WRONLY);
        printf("Writing to named pipe '%s'.\n", pipeName);
        fflush(stdout);  // Flush the output buffer

        printf("Enter data to write (max %d characters): ", 4096);
        fflush(stdout);  // Flush the output buffer
        scanf("%*c");
       ssize_t bytesWritten = read(0, buffer, sizeof(buffer));  // Read from standard input
(0)
        if (bytesWritten == -1) {
            perror("Error reading from standard input");
            exit(1);
        }
        ssize_t bytesActuallyWritten = write(pipe, buffer, bytesWritten);
        if (bytesActuallyWritten == -1) {
            perror("Error writing to named pipe");
            close(pipe);
            exit(1);
        }
    }
    else {
        printf("Invalid pipe mode.\n");
        exit(1);
    }
    if (pipe == -1) {
        perror("Error opening named pipe");
        exit(1);
    }
}
```

For this task, we are assuming we have already created a named pipe using create-named-pipe.

Commands to perform this task:

- ➔ Open terminal let say terminal 1, run the program in "read" mode after compiling it:

  ```
  ./program_name communicate-through-pipe <pipe_name> <operation/mode>
  ```

  In this case of read mode and having pipe named 'mypipe' command will be –

  ```
  ./program_name communicate-through-pipe mypipe read
  ```

- ➔ In another terminal window let say terminal 2, write the command

  ```
  ./program_name communicate-through-pipe mypipe write
  ```

  Now you will have a prompt to enter the data you want to write and as you will write the data to it, the same will be read by pipe in terminal 1.
  So this function will communicate between two processes using pipe.

  Note: Ensure that both processes are running simultaneously.