LAB 5
File Handling in Linux Using System Calls

Department of Computer Science, School of Engineering
Report Submission Date: Due in Next Lab

Objectives:

The objective of this lab is to develop proficiency in file handling in Linux using system calls such as `open`, `read`, and `write`. Students will explore opening files, reading and writing data, and copying file contents efficiently, while implementing robust error handling using `perror` for detailed diagnostics.

Instructions:

- Utilize the lab time effectively; avoid distractions such as cell phones.
- Use personal systems running Ubuntu for flexibility; macOS users should adapt commands accordingly.
- Create a `Lab5` directory to store all programs. Compile and test each program individually.
- Test programs with sample files (e.g., create `sample.txt` using `echo "Test data" > sample.txt`).
- Refrain from copying solutions directly; understand the system calls and experiment with variations.
- Ensure proper file descriptor closure to prevent resource leaks.

Tasks

Task 1: Opening a File in Read-Only Mode

Write a C program that uses the `open` system call to open a file named `sample.txt` in read-only mode. Ensure that the program prints an appropriate error message if the file doesn't exist or if there's an error opening the file. Use the `perror` function instead of `printf`, and note the advantage of using it in your observations.

Task 2: Reading File Contents

Extend the program from Task 1 to include the `read` system call. After successfully opening `sample.txt`, read its contents into a buffer of size 1024 bytes using the `read` system call. Print the contents of the file to standard output. Ensure error handling is included for the reading operation.

Task 3: Creating and Writing to a File

Write a C program that creates a new file named `output.txt` using the `open` system call in write-only mode. Then, using the `write` system call, write the following text to the file: "Hello, this is a test message." Ensure proper error handling for file creation and writing operations.

Task 4: Copying File Contents with Command-Line Arguments

Write a C program that takes two command-line arguments: the names of an input file and an output file. Use the `open`, `read`, and `write` system calls to copy the contents of the input file to the output file. Your program should handle the following scenarios:

- Verify that the correct number of command-line arguments are provided. If not, print a usage message indicating the correct usage of the program.
- If the input file cannot be opened for reading, print an error message and exit the program with a non-zero exit status.
- If the output file cannot be opened for writing, print an error message and exit the program with a non-zero exit status.
- Ensure that the program can handle copying large files efficiently by reading and writing data in chunks rather than all at once. Properly close any opened files and handle any errors that may occur during the file copying process.

System Calls Reference

This section provides details on the system calls used in this lab: `open`, `read`, `write`, and `close`.

`open`

Prototype: `int open(const char *pathname, int flags, mode_t mode);`

- **Description:** Opens a file and returns a file descriptor. Used to access files for reading, writing, or both.
- **Parameters:**
  - `pathname`: Path to the file (e.g., "sample.txt").
  - `flags`: Options like `O_RDONLY` (read-only), `O_WRONLY` (write-only), `O_CREAT` (create if not exists).
  - `mode`: Permissions (e.g., 0644 for rw-r–r–) if `O_CREAT` is used.
- **Return Value:** File descriptor on success, -1 on error (sets `errno`).

`read`

Prototype: `ssize_t read(int fd, void *buf, size_t count);`

- **Description:** Reads data from a file descriptor into a buffer.
- **Parameters:**
  - `fd`: File descriptor from `open`.
  - `buf`: Buffer to store read data.
  - `count`: Maximum bytes to read.
- **Return Value:** Number of bytes read, 0 at end of file, -1 on error.

`write`

Prototype: `ssize_t write(int fd, const void *buf, size_t count);`

- **Description:** Writes data from a buffer to a file descriptor.
- **Parameters:**
  - `fd`: File descriptor from `open`.
  - `buf`: Buffer containing data to write.
  - `count`: Number of bytes to write.
- **Return Value:** Number of bytes written, -1 on error.

`close`

Prototype: `int close(int fd);`

- **Description:** Closes a file descriptor, freeing system resources.
- **Parameters:**
  - `fd`: File descriptor to close.
- **Return Value:** 0 on success, -1 on error.

Example Commands

```
ls -li
```

Listing 1: Listing Files with Inodes

Description: Displays files with inode numbers, useful for identifying hard and soft links.

```
ln -s xyx softlink
```

Listing 2: Creating a Symbolic Link

Description: Creates a symbolic (soft) link named `softlink` pointing to `xyx`.

```
ln abc hardlink
```

Listing 3: Creating a Hard Link

Description: Creates a hard link named `hardlink` for the file `abc`.

```
ls /dev | less
```

Listing 4: Listing Device Files

Description: Lists device files in `/dev`, piped to `less` for scrolling.

```
sudo mount /dev/sdb1 /mnt
```

Listing 5: Mounting a Flash Device

Description: Mounts a flash device (e.g., `/dev/sdb1`) to `/mnt`. Replace `sdb1` with the actual device name (use `lsblk` to identify).

```
lsblk
```

Listing 6: Listing Block Devices

Description: Lists block devices (e.g., disks, USB drives) and their partitions, useful for identifying device names like `sdb1` for mounting.