LAB 6
File Handling in Linux Using lseek, fstat, stat, lstat, dup, dup2, and stdio Functions

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

Objectives:

The objective of this lab is to gain proficiency in advanced file handling in Linux using system calls such as `lseek`, `fstat`, `stat`, `lstat`, `dup`, and `dup2`, alongside standard I/O functions like `fopen`, `fprintf`, `sprintf`, and `sscanf`. Students will learn to manipulate file offsets, retrieve file metadata, duplicate file descriptors, and format and parse data from files, while ensuring robust error handling with `perror`.

Instructions:

- Use lab time efficiently; avoid distractions such as cell phones.
- Work on personal systems running Ubuntu; macOS users should adapt commands as needed.
- Create a `Lab6` 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`).
- Avoid copying solutions; experiment with system calls and stdio functions to deepen understanding.
- Close file descriptors properly to prevent resource leaks.

Tasks

Task 1: Using lseek to Move File Offset

Write a C program that opens a file named `sample.txt` in read-only mode using `open`. Use `lseek` to move the file offset to the 5th byte and read the next 10 bytes into a buffer. Print the contents and handle errors using `perror`. Observe how `lseek` affects the file offset.

Task 2: Retrieving File Metadata with stat and lstat

Write a C program that uses `stat` and `lstat` to retrieve metadata for `sample.txt` and a symbolic link to it (create one using `ln -s sample.txt samplelink`). Print the file size, inode number, and whether it's a regular file or symbolic link. Compare the results of `stat` and `lstat` and note differences in your observations.

Task 3: Duplicating File Descriptors with dup and dup2

Write a C program that opens `sample.txt` in read-only mode and duplicates its file descriptor using `dup` using this new descriptor read the file contents. Then, open a new file `output1.txt` in write-only mode and use `dup2` to redirect standard output (file descriptor 1) to `output1.txt`. Print a message using `printf` and verify it's written to `output.txt`. Include error handling.

Task 4: Formatting File Metadata with sprintf

Write a C program that uses `stat` to retrieve metadata for `sample.txt`. Use `sprintf` to format a string containing the file's size (in bytes), last modification time raw, and inode number. Write this formatted string to a file named `metadata.txt` using `fopen` and `fprintf`. Include error handling for file operations and note the advantage of using `sprintf` over direct string concatenation in your observations.

Task 5: Parsing File Data with sscanf

Write a C program that reads a line from a file named `data.txt` (create it with content like `"Item: 42 3.14 Widget"` using `echo`). Use `fopen` and `fgets` to read the line into a buffer, then use `sscanf` to parse the line into an integer (e.g., 42), a float (e.g., 3.14), and a string (e.g., "Widget"). Print the parsed values to standard output and handle errors if the file doesn't exist or the parsing fails.

System Calls Reference

This section provides details on the system calls used in this lab: `lseek`, `fstat`, `stat`, `lstat`, `dup`, and `dup2`.

`lseek`

Prototype: `off_t lseek(int fd, off_t offset, int whence);`

- **Description:** Repositions the file offset of the file descriptor.
- **Parameters:**
  - `fd`: File descriptor.
  - `offset`: Number of bytes to move.
  - `whence`: `SEEK_SET` (absolute), `SEEK_CUR` (relative to current), `SEEK_END` (relative to end).
- **Return Value:** New offset on success, -1 on error.

`fstat`

Prototype: `int fstat(int fd, struct stat *buf);`

- **Description:** Retrieves file metadata using a file descriptor.
- **Parameters:**
  - `fd`: File descriptor.
  - `buf`: Pointer to `struct stat` for storing metadata.
- **Return Value:** 0 on success, -1 on error.

`stat`

Prototype: `int stat(const char *pathname, struct stat *buf);`

- **Description:** Retrieves file metadata by pathname, following symbolic links.
- **Parameters:**
  - `pathname`: Path to the file.
  - `buf`: Pointer to `struct stat`.
- **Return Value:** 0 on success, -1 on error.

`lstat`

Prototype: `int lstat(const char *pathname, struct stat *buf);`

- **Description:** Similar to `stat`, but does not follow symbolic links.
- **Parameters:** Same as `stat`.
- **Return Value:** 0 on success, -1 on error.

dup

Prototype: `int dup(int oldfd);`

- **Description:** Duplicates a file descriptor to the lowest available descriptor.
- **Parameters:**
  - `oldfd`: File descriptor to duplicate.
- **Return Value:** New file descriptor on success, -1 on error.

dup2

Prototype: `int dup2(int oldfd, int newfd);`

- **Description:** Duplicates `oldfd` to `newfd`, closing `newfd` if already open.
- **Parameters:**
  - `oldfd`: File descriptor to duplicate.
  - `newfd`: Target descriptor number.
- **Return Value:** `newfd` on success, -1 on error.

Example Commands

```
echo "This is a test file with some data" > sample.txt
```

Listing 1: Creating a Sample File

Description: Creates a sample file for testing.

```
echo "Item: 42 3.14 Widget" > data.txt
```

Listing 2: Creating a Data File for Parsing

Description: Creates a file with formatted data for parsing with `sscanf`.

```
ln -s sample.txt sample_link
```

Listing 3: Creating a Symbolic Link

Description: Creates a symbolic link to `sample.txt`.

```
ls -l sample.txt sample_link
```

Listing 4: Checking File Metadata

Description: Displays metadata, distinguishing between the file and its symbolic link.

```
ls -l /proc/self/fd
```

Listing 5: Viewing File Descriptors

Description: Lists open file descriptors for the current process.

Sample Example: File Copy Program from Previous Lab

This example implements a file copy program from the last task of the previous lab. The program takes two command-line arguments: the source file path and the destination file path. It uses the `open()` system call to open the source file in read-only mode and the destination file in write-only mode (creating it if it doesn't exist). The `read()` system call reads chunks of data from the source file, and the `write()` system call writes these chunks to the destination file. The program includes error handling for all system calls and buffer overflow situations, closing files with `close()` after completion.

```c
// Header files
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h> // Various data types
#include <sys/stat.h>  // Permissions for user, group, other
#define BUFFER_SIZE 4096

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <source_file> <destination_file>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // Open source file
    int source_fd = open(argv[1], O_RDONLY);
    if (source_fd == -1) {
        perror("Error opening source file");
        exit(EXIT_FAILURE);
    }

    // Open destination file (creating it if it doesn't exist)
    int dest_fd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (dest_fd == -1) {
        perror("Error opening destination file");
        close(source_fd);
        exit(EXIT_FAILURE);
    }

    // Copy data from source file to destination file
    char buffer[BUFFER_SIZE];
    ssize_t bytes_read, bytes_written;
    while ((bytes_read = read(source_fd, buffer, BUFFER_SIZE)) > 0) {
        bytes_written = write(dest_fd, buffer, bytes_read);
        if (bytes_written != bytes_read) {
            perror("Error writing to destination file");
            close(source_fd);
            close(dest_fd);
            exit(EXIT_FAILURE);
        }
    }
```

```c
    if (bytes_read == -1) {
        perror("Error reading from source file");
        close(source_fd);
        close(dest_fd);
        exit(EXIT_FAILURE);
    }

    // Close files
    if (close(source_fd) == -1) {
        perror("Error closing source file");
        exit(EXIT_FAILURE);
    }
    if (close(dest_fd) == -1) {
        perror("Error closing destination file");
        exit(EXIT_FAILURE);
    }

    printf("File copied successfully.\n");
    return 0;
}
```

Listing 6: File Copy Program in C

Notes:

- Exit status checking can be done using the following commands:

```
./your_program
echo $?
```

Listing 7: File Copy Program in C

`exit(1)` is equivalent to `exit(EXIT_FAILURE)`.

- To redirect standard error stream to a file:

```
./your_program 2> error.log
```

Listing 8: File Copy Program in C

This command runs `your_program` and redirects error messages (stderr) to `error.log`. Inspect `error.log` to view errors.

- To see both program output and standard error on the terminal:

```
./your_program 2>&1
```

Listing 9: File Copy Program in C