P1

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

int main() {

    int fd;

    off\_t hole\_size = 1024 \* 1024; // 1 MB hole size

    // Open a file for writing (or create it if it doesn't exist)

    fd = open("file\_with\_hole.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, S\_IRUSR | S\_IWUSR);

    if (fd == -1) {

        perror("open");

        exit(EXIT\_FAILURE);

    }

    // Seek to the desired position to create a hole

    if (lseek(fd, hole\_size - 1, SEEK\_SET) == -1) {

        perror("lseek");

        close(fd);

        exit(EXIT\_FAILURE);

    }

    // Write a single byte to create the hole

    char hole\_data = 'A';

    if (write(fd, &hole\_data, 1) != 1) {

        perror("write");

        close(fd);

        exit(EXIT\_FAILURE);

    }

    // Close the file

    close(fd);

    printf("File with hole created successfully.\n");

    return 0;

}

P2

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <sys/types.h>

int main(int argc, char \*argv[]) {

    if (argc < 2) {

        fprintf(stderr, "Usage: %s <file1> <file2> ... <fileN>\n", argv[0]);

        return 1;

    }

    for (int i = 1; i < argc; i++) {

        const char \*filename = argv[i];

        struct stat file\_stat;

        if (stat(filename, &file\_stat) == -1) {

            perror("stat");

            continue; // Skip to the next file

        }

        printf("File: %s\n", filename);

        printf("Inode Number: %lu\n", (unsigned long)file\_stat.st\_ino);

        printf("\n");

    }

    return 0;

}

P3

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

#include <time.h>

#include <pwd.h>

#include <grp.h>

void print\_file\_properties(const char \*filename) {

    struct stat file\_stat;

    if (stat(filename, &file\_stat) == -1) {

        perror("stat");

        return;

    }

    printf("File: %s\n", filename);

    printf("Inode Number: %lu\n", (unsigned long)file\_stat.st\_ino);

    printf("Number of Hard Links: %lu\n", (unsigned long)file\_stat.st\_nlink);

    printf("File Size: %ld bytes\n", (long)file\_stat.st\_size);

    printf("File Permissions: ");

    printf((S\_ISDIR(file\_stat.st\_mode)) ? "d" : "-");

    printf((file\_stat.st\_mode & S\_IRUSR) ? "r" : "-");

    printf((file\_stat.st\_mode & S\_IWUSR) ? "w" : "-");

    printf((file\_stat.st\_mode & S\_IXUSR) ? "x" : "-");

    printf((file\_stat.st\_mode & S\_IRGRP) ? "r" : "-");

    printf((file\_stat.st\_mode & S\_IWGRP) ? "w" : "-");

    printf((file\_stat.st\_mode & S\_IXGRP) ? "x" : "-");

    printf((file\_stat.st\_mode & S\_IROTH) ? "r" : "-");

    printf((file\_stat.st\_mode & S\_IWOTH) ? "w" : "-");

    printf((file\_stat.st\_mode & S\_IXOTH) ? "x" : "-");

    printf("\n");

    struct passwd \*owner\_info = getpwuid(file\_stat.st\_uid);

    struct group \*group\_info = getgrgid(file\_stat.st\_gid);

    if (owner\_info != NULL)

        printf("Owner: %s\n", owner\_info->pw\_name);

    if (group\_info != NULL)

        printf("Group: %s\n", group\_info->gr\_name);

    printf("Last Access Time: %s", ctime(&file\_stat.st\_atime));

    printf("Last Modification Time: %s", ctime(&file\_stat.st\_mtime));

}

int main(int argc, char \*argv[]) {

    if (argc != 2) {

        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);

        return 1;

    }

    const char \*filename = argv[1];

    print\_file\_properties(filename);

    return 0;

}

P4

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <sys/types.h>

int main(int argc, char \*argv[]) {

    if (argc != 2) {

        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);

        return 1;

    }

    const char \*filename = argv[1];

    struct stat file\_stat;

    if (stat(filename, &file\_stat) == -1) {

        perror("stat");

        return 1;

    }

    if (S\_ISREG(file\_stat.st\_mode)) {

        printf("%s is a regular file.\n", filename);

    } else if (S\_ISDIR(file\_stat.st\_mode)) {

        printf("%s is a directory.\n", filename);

    } else if (S\_ISLNK(file\_stat.st\_mode)) {

        printf("%s is a symbolic link.\n", filename);

    } else if (S\_ISFIFO(file\_stat.st\_mode)) {

        printf("%s is a named pipe (FIFO).\n", filename);

    } else if (S\_ISSOCK(file\_stat.st\_mode)) {

        printf("%s is a socket.\n", filename);

    } else if (S\_ISBLK(file\_stat.st\_mode)) {

        printf("%s is a block device.\n", filename);

    } else if (S\_ISCHR(file\_stat.st\_mode)) {

        printf("%s is a character device.\n", filename);

    } else {

        printf("%s is an unknown type of file.\n", filename);

    }

    return 0;

}

P5

#include <stdio.h>

#include <unistd.h>

int main(int argc, char \*argv[]) {

    if (argc != 2) {

        printf("Usage: %s <filename>\n", argv[0]);

        return 1;

    }

    const char \*filename = argv[1];

    // Use the access() function to check if the file exists

    if (access(filename, F\_OK) == 0) {

        printf("File '%s' is present in the current directory.\n", filename);

    } else {

        printf("File '%s' is not present in the current directory.\n", filename);

    }

    return 0;

}

P6

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h>

int main(int argc, char \*argv[]) {

    if (argc != 2) {

        printf("Usage: %s <prefix>\n", argv[0]);

        return 1;

    }

    const char \*prefix = argv[1];

    DIR \*dir;

    struct dirent \*entry;

    // Open the current directory

    dir = opendir(".");

    if (dir == NULL) {

        perror("opendir");

        return 1;

    }

    // Iterate through directory entries

    printf("Files beginning with '%s' in the current directory:\n", prefix);

    while ((entry = readdir(dir)) != NULL) {

        // Check if the entry is a regular file and starts with the specified prefix

        if (entry->d\_type == DT\_REG && strncmp(entry->d\_name, prefix, strlen(prefix)) == 0) {

            printf("%s\n", entry->d\_name);

        }

    }

    // Close the directory

    closedir(dir);

    return 0;

}

P7

#include <stdio.h>

#include <stdlib.h>

#include <dirent.h>

int main() {

    DIR \*dir;

    struct dirent \*entry;

    int fileCount = 0; // Variable to count files

    // Open the current directory

    dir = opendir(".");

    if (dir == NULL) {

        perror("opendir");

        return 1;

    }

    // Iterate through directory entries

    printf("Files in the current directory:\n");

    while ((entry = readdir(dir)) != NULL) {

        // Check if the entry is a regular file (not a directory or special file)

        if (entry->d\_type == DT\_REG) {

            printf("%s\n", entry->d\_name);

            fileCount++;

        }

    }

    // Close the directory

    closedir(dir);

    // Display the total number of files

    printf("Total number of files: %d\n", fileCount);

    return 0;

}

P8

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <string.h>

// Structure to store file information (name and size)

struct FileInfo {

    char \*name;

    off\_t size;

};

// Comparison function for sorting FileInfo structures by size in ascending order

int compareFileInfo(const void \*a, const void \*b) {

    return ((struct FileInfo \*)a)->size - ((struct FileInfo \*)b)->size;

}

int main(int argc, char \*argv[]) {

    if (argc < 2) {

        printf("Usage: %s <file1> <file2> ... <fileN>\n", argv[0]);

        return 1;

    }

    // Create an array of FileInfo structures

    int numFiles = argc - 1;

    struct FileInfo \*fileInfoArray = (struct FileInfo \*)malloc(numFiles \* sizeof(struct FileInfo));

    if (fileInfoArray == NULL) {

        perror("malloc");

        return 1;

    }

    // Retrieve file sizes and store them in the array

    for (int i = 1; i <= numFiles; i++) {

        struct stat fileStat;

        if (stat(argv[i], &fileStat) == -1) {

            perror("stat");

            return 1;

        }

        fileInfoArray[i - 1].name = argv[i];

        fileInfoArray[i - 1].size = fileStat.st\_size;

    }

    // Sort the FileInfo array based on file sizes

    qsort(fileInfoArray, numFiles, sizeof(struct FileInfo), compareFileInfo);

    // Display the sorted file names

    printf("File names in ascending order of sizes:\n");

    for (int i = 0; i < numFiles; i++) {

        printf("%s\n", fileInfoArray[i].name);

    }

    // Free dynamically allocated memory

    free(fileInfoArray);

    return 0;

}

P9

#include <stdio.h>

#include <stdlib.h>

#include <dirent.h>

#include <sys/stat.h>

#include <time.h>

// Function to check if a file was created in a specific month

int isCreatedInMonth(const char \*filename, int targetMonth) {

    struct stat fileStat;

    if (stat(filename, &fileStat) == -1) {

        perror("stat");

        return 0; // Error, assume not created in the target month

    }

    struct tm \*creationTime = localtime(&fileStat.st\_ctime);

    return creationTime->tm\_mon + 1 == targetMonth; // +1 because months are 0-based

}

int main(int argc, char \*argv[]) {

    if (argc != 2) {

        printf("Usage: %s <target\_month>\n", argv[0]);

        return 1;

    }

    int targetMonth = atoi(argv[1]);

    if (targetMonth < 1 || targetMonth > 12) {

        printf("Invalid month. Please enter a month between 1 and 12.\n");

        return 1;

    }

    DIR \*dir;

    struct dirent \*entry;

    // Open the current directory

    dir = opendir(".");

    if (dir == NULL) {

        perror("opendir");

        return 1;

    }

    // Iterate through directory entries

    printf("Files created in month %d:\n", targetMonth);

    while ((entry = readdir(dir)) != NULL) {

        if (entry->d\_type == DT\_REG && isCreatedInMonth(entry->d\_name, targetMonth)) {

            printf("%s\n", entry->d\_name);

        }

    }

    // Close the directory

    closedir(dir);

    return 0;

}

P10

#include <stdio.h>

#include <stdlib.h>

#include <dirent.h>

#include <sys/stat.h>

int main(int argc, char \*argv[]) {

    if (argc != 2) {

        printf("Usage: %s <size\_in\_bytes>\n", argv[0]);

        return 1;

    }

    long long n = atoll(argv[1]); // Convert the size argument to a long long integer

    if (n < 0) {

        printf("Invalid size. Please enter a non-negative number.\n");

        return 1;

    }

    DIR \*dir;

    struct dirent \*entry;

    // Open the current directory

    dir = opendir(".");

    if (dir == NULL) {

        perror("opendir");

        return 1;

    }

    // Iterate through directory entries

    printf("Files larger than %lld bytes:\n", n);

    while ((entry = readdir(dir)) != NULL) {

        if (entry->d\_type == DT\_REG) {

            struct stat fileStat;

            if (stat(entry->d\_name, &fileStat) == -1) {

                perror("stat");

                continue; // Skip this file and move to the next

            }

            if (fileStat.st\_size > n) {

                printf("%s\n", entry->d\_name);

            }

        }

    }

    // Close the directory

    closedir(dir);

    return 0;

}

P11

#include <stdio.h>

int main() {

    // Redirect standard output to a file

    FILE \*file = freopen("output.txt", "w", stdout);

    if (file == NULL) {

        perror("freopen");

        return 1;

    }

    // Print to the redirected stdout

    printf("This output is redirected to a file.\n");

    // Close the file (optional)

    fclose(file);

    return 0;

}

P12

#include <stdio.h>

#include <stdlib.h>

#include <dirent.h>

#include <string.h>

// Comparison function for sorting directory names alphabetically

int compareDirNames(const void \*a, const void \*b) {

    return strcmp(\*(const char \*\*)a, \*(const char \*\*)b);

}

int main() {

    DIR \*dir;

    struct dirent \*entry;

    int dirCount = 0; // Variable to count subdirectories

    // Open the current directory

    dir = opendir(".");

    if (dir == NULL) {

        perror("opendir");

        return 1;

    }

    // Create an array to store subdirectory names

    char \*\*dirNames = NULL;

    // Iterate through directory entries

    while ((entry = readdir(dir)) != NULL) {

        if (entry->d\_type == DT\_DIR && strcmp(entry->d\_name, ".") != 0 && strcmp(entry->d\_name, "..") != 0) {

            // Allocate memory for the directory name and store it in the array

            char \*dirName = strdup(entry->d\_name);

            if (dirName != NULL) {

                dirNames = (char \*\*)realloc(dirNames, (dirCount + 1) \* sizeof(char \*));

                if (dirNames != NULL) {

                    dirNames[dirCount++] = dirName;

                }

            }

        }

    }

    // Sort the subdirectory names alphabetically

    qsort(dirNames, dirCount, sizeof(char \*), compareDirNames);

    // Display the sorted subdirectory names

    printf("Subdirectories in alphabetical order:\n");

    for (int i = 0; i < dirCount; i++) {

        printf("%s\n", dirNames[i]);

        free(dirNames[i]); // Free allocated memory for each name

    }

    // Free the array of directory names

    free(dirNames);

    // Close the directory

    closedir(dir);

    return 0;

}

P13

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

int main() {

    // Save the current file descriptor for standard output (usually 1, stdout)

    int original\_stdout = dup(STDOUT\_FILENO);

    // Open the "output.txt" file for writing, creating it if it doesn't exist, and truncating it

    int fd = open("output.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);

    if (fd == -1) {

        perror("open");

        return 1;

    }

    // Use the dup2() system call to redirect standard output to the file descriptor of "output.txt"

    if (dup2(fd, STDOUT\_FILENO) == -1) {

        perror("dup2");

        return 1;

    }

    // Now, anything printed to stdout will be redirected to "output.txt"

    // Print to the redirected stdout

    printf("This output is redirected to a file.\n");

    // Close the file descriptor and restore the original stdout

    close(fd);

    dup2(original\_stdout, STDOUT\_FILENO);

    // Continue to print to the original stdout (console)

    printf("This output goes to the console.\n");

    return 0;

}

P14

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <string.h>

int main(int argc, char \*argv[]) {

    if (argc != 2) {

        printf("Usage: %s <filename>\n", argv[0]);

        return 1;

    }

    const char \*filename = argv[1];

    struct stat fileStat;

    // Use the stat() system call to retrieve file properties

    if (stat(filename, &fileStat) == -1) {

        perror("stat");

        return 1;

    }

    // Determine and print the file type based on the file mode

    if (S\_ISREG(fileStat.st\_mode)) {

        printf("File type: Regular file\n");

    } else if (S\_ISDIR(fileStat.st\_mode)) {

        printf("File type: Directory\n");

    } else if (S\_ISCHR(fileStat.st\_mode)) {

        printf("File type: Character device\n");

    } else if (S\_ISBLK(fileStat.st\_mode)) {

        printf("File type: Block device\n");

    } else if (S\_ISFIFO(fileStat.st\_mode)) {

        printf("File type: FIFO or pipe\n");

    } else if (S\_ISLNK(fileStat.st\_mode)) {

        printf("File type: Symbolic link\n");

    } else if (S\_ISSOCK(fileStat.st\_mode)) {

        printf("File type: Socket\n");

    } else {

        printf("File type: Unknown\n");

    }

    return 0;

}

P15

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

    int pipe\_fd[2]; // File descriptors for the pipe

    pid\_t child\_pid;

    // Create the pipe

    if (pipe(pipe\_fd) == -1) {

        perror("pipe");

        exit(1);

    }

    // Create a child process

    child\_pid = fork();

    if (child\_pid == -1) {

        perror("fork");

        exit(1);

    }

    if (child\_pid == 0) {

        // Child process (reader)

        close(pipe\_fd[1]); // Close the write end of the pipe

        char buffer[100];

        int bytes\_read;

        // Read from the pipe

        bytes\_read = read(pipe\_fd[0], buffer, sizeof(buffer));

        if (bytes\_read == -1) {

            perror("read");

            exit(1);

        }

        printf("Child Process (Reader) received: %.\*s\n", bytes\_read, buffer);

        close(pipe\_fd[0]); // Close the read end of the pipe in the child

    } else {

        // Parent process (writer)

        close(pipe\_fd[0]); // Close the read end of the pipe

        // Write to the pipe

        const char \*message = "Hello, Child!";

        if (write(pipe\_fd[1], message, strlen(message)) == -1) {

            perror("write");

            exit(1);

        }

        close(pipe\_fd[1]); // Close the write end of the pipe in the parent

    }

    return 0;

}