1. To create ‘n’ children. When the children will terminate, display total cumulative time children spent in user and kernel mode

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/resource.h>
#include <time.h>

// Function for the child process to simulate work
void child_task() {
    // Simulate some work
    sleep(1);  // Replace with actual work
}

int main() {
    int n, i;
    pid_t pid;

    printf("Enter the number of children to create: ");
    scanf("%d", &n);

    struct rusage usage_start, usage_end;

    // Get resource usage before creating children
    getrusage(RUSAGE_CHILDREN, &usage_start);

    // Create n child processes
    for (i = 0; i < n; i++) {
        pid = fork();
        if (pid == 0) {
            // Child process
            child_task();
            exit(0);  // Terminate the child process
        } else if (pid < 0) {
            perror("fork");
            exit(EXIT_FAILURE);
        }
    }

    // Wait for all children to terminate
    for (i = 0; i < n; i++) {
        wait(NULL);
    }

    // Get resource usage after children terminate
    getrusage(RUSAGE_CHILDREN, &usage_end);

    // Calculate cumulative times
    double user_time = (usage_end.ru_utime.tv_sec - usage_start.ru_utime.tv_sec) +
                       (usage_end.ru_utime.tv_usec - usage_start.ru_utime.tv_usec) / 1e6;
    double system_time = (usage_end.ru_stime.tv_sec - usage_start.ru_stime.tv_sec) +
                         (usage_end.ru_stime.tv_usec - usage_start.ru_stime.tv_usec) / 1e6;

    // Display the results
```

```
    printf("\nTotal cumulative time spent in user mode: %.6f seconds\n", user_time);
    printf("Total cumulative time spent in kernel mode: %.6f seconds\n", system_time);

    return 0;
}
```

2.To generate parent process to write unnamed pipe and will read from it.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUFFER_SIZE 1024

int main() {
    int pipe_fd[2];  // File descriptors for the pipe
    pid_t pid;
    char write_msg[] = "Hello from the parent process!";
    char read_msg[BUFFER_SIZE];

    // Create the pipe
    if (pipe(pipe_fd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    // Fork to create a child process
    pid = fork();

    if (pid < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid > 0) {  // Parent process
        close(pipe_fd[0]);  // Close the read end of the pipe

        // Write to the pipe
        printf("Parent: Writing to pipe: %s\n", write_msg);
        write(pipe_fd[1], write_msg, strlen(write_msg) + 1);

        close(pipe_fd[1]);  // Close the write end of the pipe
        wait(NULL);         // Wait for the child to finish
    } else {  // Child process
        close(pipe_fd[1]);  // Close the write end of the pipe

        // Read from the pipe
        read(pipe_fd[0], read_msg, BUFFER_SIZE);
        printf("Child: Read from pipe: %s\n", read_msg);

        close(pipe_fd[0]);  // Close the read end of the pipe
        exit(0);
    }
    return 0;
```

```
}
```

3.To create a file with hole in it.

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

int main() {
    const char *filename = "sparse_file.txt";
    int fd;
    ssize_t bytes_written;

    // Open the file for writing (create if it doesn't exist)
    fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    // Write initial data
    const char *data1 = "Start of file.\n";
    bytes_written = write(fd, data1, 15);
    if (bytes_written < 0) {
        perror("write");
        close(fd);
        exit(EXIT_FAILURE);
    }

    // Create a hole using lseek
    off_t offset = lseek(fd, 1024 * 1024, SEEK_CUR);  // Move 1 MB ahead
    if (offset < 0) {
        perror("lseek");
        close(fd);
        exit(EXIT_FAILURE);
    }

    // Write more data
    const char *data2 = "End of file.\n";
    bytes_written = write(fd, data2, 13);
    if (bytes_written < 0) {
        perror("write");
        close(fd);
        exit(EXIT_FAILURE);
    }

    // Close the file
    close(fd);

    printf("Sparse file '%s' created successfully.\n", filename);
    return 0;
}
```

4.Takes multiple files as Command Line Arguments and print their inode number.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <file1> <file2> ...\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    for (int i = 1; i < argc; i++) {
        struct stat file_stat;

        // Get file information
        if (stat(argv[i], &file_stat) == -1) {
            perror(argv[i]);
            continue;  // Skip to the next file if stat fails
        }

        // Print the file name and its inode number
        printf("File: %s, Inode: %lu\n", argv[i], file_stat.st_ino);
    }

    return 0;
}
```

---

5. To handle the two-way communication between parent and child using pipe.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUFFER_SIZE 1024

int main() {
    int parent_to_child[2]; // Pipe for parent to child communication
    int child_to_parent[2]; // Pipe for child to parent communication
    pid_t pid;
    char parent_message[] = "Hello from Parent!";
    char child_message[] = "Hello from Child!";
    char buffer[BUFFER_SIZE];

    // Create the pipes
    if (pipe(parent_to_child) == -1 || pipe(child_to_parent) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    // Fork to create child process
    pid = fork();

    if (pid < 0) {
        perror("fork");
```

```
        exit(EXIT_FAILURE);
    }

    if (pid > 0) { // Parent process
        // Close unused ends of the pipes
        close(parent_to_child[0]); // Close read end of parent-to-child pipe
        close(child_to_parent[1]); // Close write end of child-to-parent pipe

        // Write to the child
        printf("Parent: Sending message to child...\n");
        write(parent_to_child[1], parent_message, strlen(parent_message) + 1);

        // Read from the child
        read(child_to_parent[0], buffer, BUFFER_SIZE);
        printf("Parent: Received message from child: %s\n", buffer);

        // Close the used ends
        close(parent_to_child[1]);
        close(child_to_parent[0]);
    } else { // Child process
        // Close unused ends of the pipes
        close(parent_to_child[1]); // Close write end of parent-to-child pipe
        close(child_to_parent[0]); // Close read end of child-to-parent pipe

        // Read from the parent
        read(parent_to_child[0], buffer, BUFFER_SIZE);
        printf("Child: Received message from parent: %s\n", buffer);

        // Write to the parent
        printf("Child: Sending message to parent...\n");
        write(child_to_parent[1], child_message, strlen(child_message) + 1);

        // Close the used ends
        close(parent_to_child[0]);
        close(child_to_parent[1]);
    }

    return 0;
}
```

---

6. Print the type of file where file name accepted through Command Line.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

void print_file_type(const char *filename) {
    struct stat file_stat;

    // Get file information
    if (stat(filename, &file_stat) == -1) {
        perror("stat");
        return;
    }
```

```
    // Determine and print the file type
    printf("File: %s\n", filename);
    if (S_ISREG(file_stat.st_mode)) {
        printf("Type: Regular file\n");
    } else if (S_ISDIR(file_stat.st_mode)) {
        printf("Type: Directory\n");
    } else if (S_ISCHR(file_stat.st_mode)) {
        printf("Type: Character device\n");
    } else if (S_ISBLK(file_stat.st_mode)) {
        printf("Type: Block device\n");
    } else if (S_ISFIFO(file_stat.st_mode)) {
        printf("Type: FIFO/pipe\n");
    } else if (S_ISLNK(file_stat.st_mode)) {
        printf("Type: Symbolic link\n");
    } else if (S_ISSOCK(file_stat.st_mode)) {
        printf("Type: Socket\n");
    } else {
        printf("Type: Unknown\n");
    }
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <file1> [file2 ...]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    for (int i = 1; i < argc; i++) {
        print_file_type(argv[i]);
    }

    return 0;
}
```

---

7. To demonstrate the use of atexit() function.

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

void cleanup_function1() {
    printf("Cleanup Function 1: Executed at program termination.\n");
}

void cleanup_function2() {
    printf("Cleanup Function 2: Executed at program termination.\n");
}

int main() {
    // Register functions with atexit()
    if (atexit(cleanup_function1) != 0) {
        fprintf(stderr, "Failed to register cleanup_function1.\n");
        exit(EXIT_FAILURE);
    }

    if (atexit(cleanup_function2) != 0) {
```

```
        fprintf(stderr, "Failed to register cleanup_function2.\n");
        exit(EXIT_FAILURE);
    }

    printf("Main program: Registered cleanup functions.\n");

    // Normal program execution
    printf("Main program: Performing tasks...\n");

    // Program exits normally here, triggering the registered functions
    return 0;
}
```

---

8. Open a file goes to sleep for 15 seconds before terminating.

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

int main() {
    const char *filename = "example.txt";

    // Open the file
    int fd = open(filename, O_RDONLY);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    printf("File ’%s’ opened successfully. File descriptor: %d\n", filename, fd);

    // Sleep for 15 seconds
    printf("Going to sleep for 15 seconds. File remains open...\n");
    sleep(15);

    // Close the file
    if (close(fd) == -1) {
        perror("close");
        exit(EXIT_FAILURE);
    }

    printf("File closed successfully. Exiting program.\n");

    return 0;
}
```

---

9. To print the size of the file.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

void print_file_size(const char *filename) {
    struct stat file_stat;
```

```
    // Get file information
    if (stat(filename, &file_stat) == -1) {
        perror("stat");
        exit(EXIT_FAILURE);
    }

    // Print the size of the file
    printf("File: %s\n", filename);
    printf("Size: %ld bytes\n", file_stat.st_size);
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <file_name>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    print_file_size(argv[1]);

    return 0;
}
```

---

10. Read the current directory and display the name of the files, no of files in current directory

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

int main() {
    DIR *dir;
    struct dirent *entry;
    int file_count = 0;

    // Open the current directory
    dir = opendir(".");
    if (dir == NULL) {
        perror("opendir");
        exit(EXIT_FAILURE);
    }

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

    // Read the directory and count files
    while ((entry = readdir(dir)) != NULL) {
        // Skip the "." and ".." directories
        if (entry->d_name[0] != '.') {
            printf("%s\n", entry->d_name);
            file_count++;
        }
    }

    // Close the directory
    closedir(dir);
```

```
    printf("\nTotal number of files: %d\n", file_count);

    return 0;
}
```

---

11. Write a C program to implement the following unix/linux command (use fork, pipe and exec system call)
ls –l | wc –l

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int pipefd[2];  // Pipe file descriptors
    pid_t pid1, pid2;

    // Create the pipe
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    // Fork the first child to run ‘ls -1‘
    if ((pid1 = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid1 == 0) {
        // In the first child process:
        // Close unused write end of the pipe
        close(pipefd[0]);

        // Redirect stdout to the write end of the pipe
        dup2(pipefd[1], STDOUT_FILENO);

        // Close the write end after redirection
        close(pipefd[1]);

        // Execute the ‘ls -1‘ command
        execlp("ls", "ls", "-1", (char *)NULL);
        perror("execlp");  // If execlp fails
        exit(EXIT_FAILURE);
    }

    // Fork the second child to run ‘wc -l‘
    if ((pid2 = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid2 == 0) {
        // In the second child process:
        // Close unused read end of the pipe
```

```
        close(pipefd[1]);

        // Redirect stdin to the read end of the pipe
        dup2(pipefd[0], STDIN_FILENO);

        // Close the read end after redirection
        close(pipefd[0]);

        // Execute the ‘wc -l‘ command
        execlp("wc", "wc", "-l", (char *)NULL);
        perror("execlp");  // If execlp fails
        exit(EXIT_FAILURE);
    }

    // Parent process: close both ends of the pipe
    close(pipefd[0]);
    close(pipefd[1]);

    // Wait for both child processes to finish
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);

    return 0;
}
```

---

12. Write a C program to display all the files from current directory which are created in particular month

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/stat.h>
#include <time.h>
#include <string.h>

void display_files_in_month(const char *target_month) {
    DIR *dir;
    struct dirent *entry;
    struct stat file_stat;
    struct tm *time_info;
    char month[20];

    // Open the current directory
    dir = opendir(".");
    if (dir == NULL) {
        perror("opendir");
        exit(EXIT_FAILURE);
    }

    printf("Files created in the month of %s:\n", target_month);

    // Read the directory and check the modification time of each file
    while ((entry = readdir(dir)) != NULL) {
        // Skip "." and ".."
        if (entry->d_name[0] == '.') {
            continue;
```

```
        }

        // Get file metadata
        if (stat(entry->d_name, &file_stat) == -1) {
            perror("stat");
            continue;
        }

        // Convert the file's creation time to tm struct
        time_info = localtime(&file_stat.st_mtime);  // Use st_mtime for last modified time

        // Get the month name
        strftime(month, sizeof(month), "%B", time_info);  // %B gives full month name

        // Compare with target month
        if (strcmp(month, target_month) == 0) {
            printf("%s\n", entry->d_name);  // Print file name if the month matches
        }
    }

    closedir(dir);
}

int main() {
    char target_month[20];

    // Ask user for the month to filter by
    printf("Enter the month (e.g., January, February, etc.): ");
    fgets(target_month, sizeof(target_month), stdin);
    target_month[strcspn(target_month, "\n")] = 0;  // Remove newline character from input

    // Display files from the specified month
    display_files_in_month(target_month);

    return 0;
}
```

---

13. Write a C program to display all the files from current directory whose size is greater that n Bytes Where n is accept from user.

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <sys/stat.h>

void display_files_greater_than_n(int size_threshold) {
    DIR *dir;
    struct dirent *entry;
    struct stat file_stat;

    // Open the current directory
    dir = opendir(".");
    if (dir == NULL) {
        perror("opendir");
        exit(EXIT_FAILURE);
    }
```

```
    printf("Files larger than %d bytes:\n", size_threshold);

    // Read the directory and check the size of each file
    while ((entry = readdir(dir)) != NULL) {
        // Skip "." and ".."
        if (entry->d_name[0] == '.') {
            continue;
        }

        // Get file metadata
        if (stat(entry->d_name, &file_stat) == -1) {
            perror("stat");
            continue;
        }

        // Check if file size is greater than the threshold
        if (file_stat.st_size > size_threshold) {
            printf("%s (Size: %ld bytes)\n", entry->d_name, file_stat.st_size);
        }
    }

    closedir(dir);
}

int main() {
    int size_threshold;

    // Ask user for the file size threshold
    printf("Enter the size threshold (in bytes): ");
    if (scanf("%d", &size_threshold) != 1) {
        fprintf(stderr, "Invalid input.\n");
        exit(EXIT_FAILURE);
    }

    // Display files with size greater than the given threshold
    display_files_greater_than_n(size_threshold);

    return 0;
}
```

---

14. Write a C program to implement the following unix/linux command
i. ls –l > output.txt

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <fcntl.h>

int main() {
    pid_t pid;
    int file_fd;

    // Create a pipe for redirecting output to a file
    file_fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

```
    if (file_fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    // Fork a child process
    if ((pid = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        // In the child process:
        // Redirect standard output to the file
        if (dup2(file_fd, STDOUT_FILENO) == -1) {
            perror("dup2");
            exit(EXIT_FAILURE);
        }

        // Close the file descriptor after redirecting
        close(file_fd);

        // Execute the ‘ls -l‘ command
        execlp("ls", "ls", "-l", (char *)NULL);
        perror("execlp");  // If execlp fails
        exit(EXIT_FAILURE);
    }

    // Parent process: wait for the child to finish
    close(file_fd);  // Parent doesn’t need the file descriptor
    waitpid(pid, NULL, 0);

    printf("Output of ’ls -l’ has been written to output.txt\n");

    return 0;
}
```

---

15. Write a C program which display the information of a given file similar to given by the unix / linux command ls –l <file name>

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
#include <string.h>

void print_permissions(mode_t mode) {
    char perms[11] = "----------";

    // File type
    if (S_ISDIR(mode)) perms[0] = ’d’;
    else if (S_ISLNK(mode)) perms[0] = ’l’;

    // User permissions
```

```
    if (mode & S_IRUSR) perms[1] = 'r';
    if (mode & S_IWUSR) perms[2] = 'w';
    if (mode & S_IXUSR) perms[3] = 'x';

    // Group permissions
    if (mode & S_IRGRP) perms[4] = 'r';
    if (mode & S_IWGRP) perms[5] = 'w';
    if (mode & S_IXGRP) perms[6] = 'x';

    // Other permissions
    if (mode & S_IROTH) perms[7] = 'r';
    if (mode & S_IWOTH) perms[8] = 'w';
    if (mode & S_IXOTH) perms[9] = 'x';

    printf("%s ", perms);
}

void display_file_info(const char *filename) {
    struct stat file_stat;
    struct passwd *pwd;
    struct group *grp;
    char time_buf[80];

    // Get file statistics
    if (stat(filename, &file_stat) == -1) {
        perror("stat");
        exit(EXIT_FAILURE);
    }

    // Print file permissions
    print_permissions(file_stat.st_mode);

    // Number of links
    printf("%ld ", file_stat.st_nlink);

    // Owner name
    pwd = getpwuid(file_stat.st_uid);
    if (pwd) {
        printf("%s ", pwd->pw_name);
    } else {
        printf("%d ", file_stat.st_uid);
    }

    // Group name
    grp = getgrgid(file_stat.st_gid);
    if (grp) {
        printf("%s ", grp->gr_name);
    } else {
        printf("%d ", file_stat.st_gid);
    }

    // File size
    printf("%ld ", file_stat.st_size);

    // Last modification time
```

```
    strftime(time_buf, sizeof(time_buf), "%b %d %H:%M", localtime(&file_stat.st_mtime));
    printf("%s ", time_buf);

    // File name
    printf("%s\n", filename);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    display_file_info(argv[1]);

    return 0;
}
```

---

16. Write a C program that behaves like a shell (command interpreter). It has its own prompt say “NewShell$”. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command.

i) count c <filename> - print number of characters in file
ii) count w <filename> - print number of words in file
iii) count l <filename> - print number of lines in file

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>

// Function to count characters in a file
int count_characters(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("fopen");
        return -1;
    }

    int count = 0;
    char ch;
    while ((ch = fgetc(file)) != EOF) {
        count++;
    }
    fclose(file);
    return count;
}

// Function to count words in a file
int count_words(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("fopen");
```

```
        return -1;
    }

    int count = 0;
    char ch;
    int in_word = 0;
    while ((ch = fgetc(file)) != EOF) {
        if (ch == ' ' || ch == '\n' || ch == '\t') {
            in_word = 0;
        } else if (!in_word) {
            in_word = 1;
            count++;
        }
    }
    fclose(file);
    return count;
}

// Function to count lines in a file
int count_lines(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("fopen");
        return -1;
    }

    int count = 0;
    char ch;
    while ((ch = fgetc(file)) != EOF) {
        if (ch == '\n') {
            count++;
        }
    }
    fclose(file);
    return count;
}

// Function to execute shell commands
void execute_command(char *input) {
    char *args[100];
    char *token = strtok(input, " \n");
    int i = 0;

    // Parse the input into arguments
    while (token != NULL) {
        args[i++] = token;
        token = strtok(NULL, " \n");
    }
    args[i] = NULL;

    // Check for custom "count" commands
    if (i >= 3 && strcmp(args[0], "count") == 0) {
        const char *filename = args[2];
        if (strcmp(args[1], "c") == 0) {
            int count = count_characters(filename);
```

```
            if (count != -1) {
                printf("Number of characters: %d\n", count);
            }
        } else if (strcmp(args[1], "w") == 0) {
            int count = count_words(filename);
            if (count != -1) {
                printf("Number of words: %d\n", count);
            }
        } else if (strcmp(args[1], "l") == 0) {
            int count = count_lines(filename);
            if (count != -1) {
                printf("Number of lines: %d\n", count);
            }
        } else {
            printf("Invalid count command\n");
        }
    } else {
        // Handle external commands by forking a new process
        pid_t pid = fork();
        if (pid == -1) {
            perror("fork");
            return;
        }
        if (pid == 0) {
            // Child process: Execute command
            if (execvp(args[0], args) == -1) {
                perror("execvp");
            }
            exit(EXIT_FAILURE);
        } else {
            // Parent process: Wait for child to finish
            waitpid(pid, NULL, 0);
        }
    }
}

int main() {
    char input[256];

    while (1) {
        // Display custom prompt
        printf("NewShell$ ");

        // Read input from the user
        if (fgets(input, sizeof(input), stdin) == NULL) {
            break; // Exit on error or EOF
        }

        // Exit the shell if the user types "exit"
        if (strncmp(input, "exit", 4) == 0) {
            break;
        }

        // Execute the command
        execute_command(input);
```

```
    }

    return 0;
}
```
---
17. Write a C program that behaves like a shell (command interpreter). It has its own prompt say “NewShell$”.
Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command.
i) list f <dirname> - print name of all files in directory
ii) list n <dirname> - print number of all entries
iii) list i<dirname> - print name and inode of all files

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>
#include <sys/wait.h>

// Function to list the names of all files in a directory
void list_files(const char *dirname) {
    DIR *dir = opendir(dirname);
    if (!dir) {
        perror("opendir");
        return;
    }

    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) {
        if (entry->d_name[0] != '.') { // Skip hidden files
            printf("%s\n", entry->d_name);
        }
    }
    closedir(dir);
}

// Function to count the number of entries in a directory
void count_entries(const char *dirname) {
    DIR *dir = opendir(dirname);
    if (!dir) {
        perror("opendir");
        return;
    }

    int count = 0;
    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) {
        count++;
    }
    printf("Number of entries: %d\n", count);
```

```
    closedir(dir);
}

// Function to list the name and inode of all files in a directory
void list_inodes(const char *dirname) {
    DIR *dir = opendir(dirname);
    if (!dir) {
        perror("opendir");
        return;
    }

    struct dirent *entry;
    struct stat file_stat;
    while ((entry = readdir(dir)) != NULL) {
        if (entry->d_name[0] != '.') { // Skip hidden files
            char file_path[1024];
            snprintf(file_path, sizeof(file_path), "%s/%s", dirname, entry->d_name);
            if (stat(file_path, &file_stat) == -1) {
                perror("stat");
                continue;
            }
            printf("%s - Inode: %ld\n", entry->d_name, file_stat.st_ino);
        }
    }
    closedir(dir);
}

// Function to execute shell commands
void execute_command(char *input) {
    char *args[100];
    char *token = strtok(input, " \n");
    int i = 0;

    // Parse the input into arguments
    while (token != NULL) {
        args[i++] = token;
        token = strtok(NULL, " \n");
    }
    args[i] = NULL;

    // Check for custom "list" commands
    if (i >= 3 && strcmp(args[0], "list") == 0) {
        const char *dirname = args[2];
        if (strcmp(args[1], "f") == 0) {
            list_files(dirname);
        } else if (strcmp(args[1], "n") == 0) {
            count_entries(dirname);
        } else if (strcmp(args[1], "i") == 0) {
            list_inodes(dirname);
        } else {
            printf("Invalid list command\n");
        }
    } else {
        // Handle external commands by forking a new process
        pid_t pid = fork();
```

```
        if (pid == -1) {
            perror("fork");
            return;
        }
        if (pid == 0) {
            // Child process: Execute command
            if (execvp(args[0], args) == -1) {
                perror("execvp");
            }
            exit(EXIT_FAILURE);
        } else {
            // Parent process: Wait for child to finish
            waitpid(pid, NULL, 0);
        }
    }
}

int main() {
    char input[256];

    while (1) {
        // Display custom prompt
        printf("NewShell$ ");

        // Read input from the user
        if (fgets(input, sizeof(input), stdin) == NULL) {
            break; // Exit on error or EOF
        }

        // Exit the shell if the user types "exit"
        if (strncmp(input, "exit", 4) == 0) {
            break;
        }

        // Execute the command
        execute_command(input);
    }

    return 0;
}
```

---

18. Write a C program that behaves like a shell (command interpreter). It has its own prompt say “NewShell$”.
Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command.
i) typeline +10 <filename> - print first 10 lines of file
ii) typeline -20 <filename> - print last 20 lines of file
iii) typeline a <filename> - print all lines of file

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

// Function to print the first n lines of a file
```

```
void print_first_n_lines(const char *filename, int n) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("fopen");
        return;
    }

    char line[1024];
    for (int i = 0; i < n && fgets(line, sizeof(line), file) != NULL; i++) {
        printf("%s", line);
    }
    fclose(file);
}

// Function to print the last n lines of a file
void print_last_n_lines(const char *filename, int n) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("fopen");
        return;
    }

    // Count total lines in the file
    char line[1024];
    int total_lines = 0;
    while (fgets(line, sizeof(line), file) != NULL) {
        total_lines++;
    }

    // Seek to the appropriate position to read the last n lines
    fseek(file, 0, SEEK_SET);
    int start_line = total_lines - n;
    if (start_line < 0) start_line = 0;

    // Print the last n lines
    int current_line = 0;
    while (fgets(line, sizeof(line), file) != NULL) {
        if (current_line >= start_line) {
            printf("%s", line);
        }
        current_line++;
    }

    fclose(file);
}

// Function to print all lines of a file
void print_all_lines(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("fopen");
        return;
    }

    char line[1024];
```

```
    while (fgets(line, sizeof(line), file) != NULL) {
        printf("%s", line);
    }
    fclose(file);
}

// Function to execute shell commands
void execute_command(char *input) {
    char *args[100];
    char *token = strtok(input, " \n");
    int i = 0;

    // Parse the input into arguments
    while (token != NULL) {
        args[i++] = token;
        token = strtok(NULL, " \n");
    }
    args[i] = NULL;

    // Check for custom "typeline" commands
    if (i >= 3 && strcmp(args[0], "typeline") == 0) {
        const char *filename = args[2];
        if (strcmp(args[1], "+") == 0) {
            int n = atoi(args[2]);
            print_first_n_lines(filename, n);
        } else if (strcmp(args[1], "-") == 0) {
            int n = atoi(args[2]);
            print_last_n_lines(filename, n);
        } else if (strcmp(args[1], "a") == 0) {
            print_all_lines(filename);
        } else {
            printf("Invalid typeline command\n");
        }
    } else {
        // Handle external commands by forking a new process
        pid_t pid = fork();
        if (pid == -1) {
            perror("fork");
            return;
        }
        if (pid == 0) {
            // Child process: Execute command
            if (execvp(args[0], args) == -1) {
                perror("execvp");
            }
            exit(EXIT_FAILURE);
        } else {
            // Parent process: Wait for child to finish
            waitpid(pid, NULL, 0);
        }
    }
}

int main() {
    char input[256];
```

```
    while (1) {
        // Display custom prompt
        printf("NewShell$ ");

        // Read input from the user
        if (fgets(input, sizeof(input), stdin) == NULL) {
            break; // Exit on error or EOF
        }

        // Exit the shell if the user types "exit"
        if (strncmp(input, "exit", 4) == 0) {
            break;
        }

        // Execute the command
        execute_command(input);
    }

    return 0;
}
```

---

19. Write a C program that behaves like a shell (command interpreter). It has its own prompt say “NewShell$”.Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should
i) additionally interpret the following command.
ii) search f <pattern> <filename> - search first occurrence of pattern in filename
iii) search c <pattern> <filename> - count no. of occurrences of pattern in filename
iv) search a <pattern> <filename> - search all occurrences of pattern in filename

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

// Function to search for the first occurrence of a pattern in a file
void search_first_occurrence(const char *pattern, const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("fopen");
        return;
    }

    char line[1024];
    int line_num = 1;
    while (fgets(line, sizeof(line), file) != NULL) {
        if (strstr(line, pattern)) {
            printf("Pattern found in line %d: %s", line_num, line);
            fclose(file);
            return;
        }
        line_num++;
    }
```

```
    printf("Pattern not found in the file.\n");
    fclose(file);
}

// Function to count the number of occurrences of a pattern in a file
void count_occurrences(const char *pattern, const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("fopen");
        return;
    }

    char line[1024];
    int count = 0;
    while (fgets(line, sizeof(line), file) != NULL) {
        char *ptr = line;
        while ((ptr = strstr(ptr, pattern)) != NULL) {
            count++;
            ptr++; // Move pointer past the current match
        }
    }

    printf("Pattern found %d times.\n", count);
    fclose(file);
}

// Function to search and print all occurrences of a pattern in a file
void search_all_occurrences(const char *pattern, const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("fopen");
        return;
    }

    char line[1024];
    int line_num = 1;
    int found = 0;
    while (fgets(line, sizeof(line), file) != NULL) {
        char *ptr = line;
        while ((ptr = strstr(ptr, pattern)) != NULL) {
            printf("Pattern found in line %d: %s", line_num, line);
            found = 1;
            ptr++; // Move pointer past the current match
        }
        line_num++;
    }

    if (!found) {
        printf("Pattern not found in the file.\n");
    }

    fclose(file);
}

// Function to execute shell commands
```

```
void execute_command(char *input) {
    char *args[100];
    char *token = strtok(input, " \n");
    int i = 0;

    // Parse the input into arguments
    while (token != NULL) {
        args[i++] = token;
        token = strtok(NULL, " \n");
    }
    args[i] = NULL;

    // Check for custom "search" commands
    if (i >= 3 && strcmp(args[0], "search") == 0) {
        const char *pattern = args[1];
        const char *filename = args[2];

        if (strcmp(args[1], "f") == 0) {
            search_first_occurrence(args[2], filename);
        } else if (strcmp(args[1], "c") == 0) {
            count_occurrences(args[2], filename);
        } else if (strcmp(args[1], "a") == 0) {
            search_all_occurrences(args[2], filename);
        } else {
            printf("Invalid search command\n");
        }
    } else {
        // Handle external commands by forking a new process
        pid_t pid = fork();
        if (pid == -1) {
            perror("fork");
            return;
        }
        if (pid == 0) {
            // Child process: Execute command
            if (execvp(args[0], args) == -1) {
                perror("execvp");
            }
            exit(EXIT_FAILURE);
        } else {
            // Parent process: Wait for child to finish
            waitpid(pid, NULL, 0);
        }
    }
}

int main() {
    char input[256];

    while (1) {
        // Display custom prompt
        printf("NewShell$ ");

        // Read input from the user
        if (fgets(input, sizeof(input), stdin) == NULL) {
```

```
            break; // Exit on error or EOF
        }

        // Exit the shell if the user types "exit"
        if (strncmp(input, "exit", 4) == 0) {
            break;
        }

        // Execute the command
        execute_command(input);
    }

    return 0;
}
```

---

20. Write a C program which receives file names as command line arguments and display those filenames in ascending order according to their sizes.

i) (e.g $ a.out a.txt b.txt c.txt, …)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

// Structure to hold file name and its corresponding size
typedef struct {
    char *filename;
    off_t size;
} FileInfo;

// Comparison function for qsort
int compare(const void *a, const void *b) {
    FileInfo *fileA = (FileInfo *)a;
    FileInfo *fileB = (FileInfo *)b;

    // Compare by file size in ascending order
    if (fileA->size < fileB->size) return -1;
    if (fileA->size > fileB->size) return 1;
    return 0;
}

// Function to get file size using stat()
off_t get_file_size(const char *filename) {
    struct stat statbuf;
    if (stat(filename, &statbuf) == -1) {
        perror("stat");
        return -1;
    }
    return statbuf.st_size;
}

int main(int argc, char *argv[]) {
    // Check if there are file arguments passed
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <file1> <file2> ...\n", argv[0]);
        return 1;
```

```
    }

    // Create an array to hold file info
    FileInfo *files = malloc((argc - 1) * sizeof(FileInfo));
    if (files == NULL) {
        perror("malloc");
        return 1;
    }

    // Get file sizes and store the file names and sizes
    for (int i = 1; i < argc; i++) {
        files[i - 1].filename = argv[i];
        files[i - 1].size = get_file_size(argv[i]);
        if (files[i - 1].size == -1) {
            free(files);
            return 1;
        }
    }

    // Sort the files array by file size
    qsort(files, argc - 1, sizeof(FileInfo), compare);

    // Print the file names in ascending order of their sizes
    printf("Files sorted by size:\n");
    for (int i = 0; i < argc - 1; i++) {
        printf("%s (size: %ld bytes)\n", files[i].filename, files[i].size);
    }

    // Free allocated memory
    free(files);
    return 0;
}
```

---

21. Write a C program which create a child process which catch a signal sighup, sigint and sigquit. The Parent process send a sighup or sigint signal after every 3 seconds, at the end of 30 second parent send sigquit signal to child and child terminates my displaying message "My DADDY has Killed me!!!

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

// Signal handler for SIGHUP
void handle_sighup(int sig) {
    printf("Child received SIGHUP\n");
}

// Signal handler for SIGINT
void handle_sigint(int sig) {
    printf("Child received SIGINT\n");
}
```

```
// Signal handler for SIGQUIT
void handle_sigquit(int sig) {
    printf("Child received SIGQUIT\n");
    printf("My DADDY has Killed me!!!\n");
    exit(0); // Child process terminates
}

int main() {
    pid_t pid;

    // Fork a child process
    pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        exit(1);
    }

    if (pid == 0) {
        // Child process

        // Set up signal handlers
        signal(SIGHUP, handle_sighup);
        signal(SIGINT, handle_sigint);
        signal(SIGQUIT, handle_sigquit);

        // Child process sleeps indefinitely to wait for signals
        while (1) {
            pause(); // Wait for signals
        }

    } else {
        // Parent process
        // Send SIGHUP or SIGINT every 3 seconds, and SIGQUIT at the 30-second mark

        for (int i = 0; i < 9; i++) {
            // Send SIGHUP every 3 seconds for the first 9 seconds
            if (i % 2 == 0) {
                kill(pid, SIGHUP);
                printf("Parent sent SIGHUP to child\n");
            } else {
                kill(pid, SIGINT);
                printf("Parent sent SIGINT to child\n");
            }
            sleep(3); // Wait for 3 seconds
        }

        // Send SIGQUIT after 30 seconds
        kill(pid, SIGQUIT);
        printf("Parent sent SIGQUIT to child\n");

        // Wait for the child process to terminate
        wait(NULL);
    }
```

```
    return 0;
}
```
---

22. Write a C program to implement the following unix/linux command (use fork, pipe and exec system call). Your program should block the signal Ctrl-C and Ctrl-\ signal during the execution.

i. ls –l | wc –l

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

// Function to block signals
void block_signals() {
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGINT);  // Block Ctrl-C
    sigaddset(&set, SIGQUIT); // Block Ctrl-\
    sigprocmask(SIG_BLOCK, &set, NULL); // Apply signal blocking
}

int main() {
    int pipefd[2];  // Pipe file descriptors
    pid_t pid1, pid2;

    // Create a pipe
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(1);
    }

    // Block SIGINT and SIGQUIT signals
    block_signals();

    // Fork first child to run "ls -l"
    if ((pid1 = fork()) == -1) {
        perror("fork");
        exit(1);
    }

    if (pid1 == 0) {
        // Child 1 (runs "ls -l")
        close(pipefd[0]); // Close unused read end of pipe
        dup2(pipefd[1], STDOUT_FILENO); // Redirect stdout to pipe

        // Execute "ls -l"
        execlp("ls", "ls", "-l", NULL);
        perror("execlp ls");  // execlp() will return if there is an error
        exit(1);
    }

    // Fork second child to run "wc -l"
    if ((pid2 = fork()) == -1) {
        perror("fork");
```

```
        exit(1);
    }

    if (pid2 == 0) {
        // Child 2 (runs "wc -l")
        close(pipefd[1]); // Close unused write end of pipe
        dup2(pipefd[0], STDIN_FILENO); // Redirect stdin to pipe

        // Execute "wc -l"
        execlp("wc", "wc", "-l", NULL);
        perror("execlp wc");  // execlp() will return if there is an error
        exit(1);
    }

    // Parent process
    close(pipefd[0]); // Close both ends of the pipe in the parent
    close(pipefd[1]);

    // Wait for both child processes to finish
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);

    return 0;
}
```

---

23. Write a C Program that demonstrates redirection of standard output to a file

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

int main() {
    // Open a file for writing
    FILE *file = freopen("output.txt", "w", stdout);

    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    // Redirected output (it will go to "output.txt" instead of the console)
    printf("This will be written to the file output.txt\n");
    printf("Redirection of stdout works!\n");

    // Close the file (optional since fclose is automatically called on program exit)
    fclose(file);

    return 0;
}
```

---

24. Write a program that illustrates how to execute two commands concurrently with a pipe.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
```

```c
#include <sys/wait.h>

int main() {
    int pipefd[2];
    pid_t pid1, pid2;

    // Create a pipe
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(1);
    }

    // Fork the first child process (for "ls -l")
    pid1 = fork();
    if (pid1 < 0) {
        perror("fork");
        exit(1);
    }

    if (pid1 == 0) {
        // First child (runs "ls -l")
        close(pipefd[0]);  // Close the unused read end of the pipe
        dup2(pipefd[1], STDOUT_FILENO);  // Redirect stdout to the pipe

        // Execute "ls -l"
        execlp("ls", "ls", "-l", NULL);
        perror("execlp ls");  // execlp() will return if there is an error
        exit(1);
    }

    // Fork the second child process (for "grep txt")
    pid2 = fork();
    if (pid2 < 0) {
        perror("fork");
        exit(1);
    }

    if (pid2 == 0) {
        // Second child (runs "grep txt")
        close(pipefd[1]);  // Close the unused write end of the pipe
        dup2(pipefd[0], STDIN_FILENO);  // Redirect stdin to the pipe

        // Execute "grep txt"
        execlp("grep", "grep", "txt", NULL);
        perror("execlp grep");  // execlp() will return if there is an error
        exit(1);
    }

    // Parent process closes both ends of the pipe
    close(pipefd[0]);
    close(pipefd[1]);

    // Wait for both child processes to finish
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);
```

```
    return 0;
}
```

---

25. Write a C program that illustrates suspending and resuming processes using signals.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>

// Signal handler to print a message when suspended and resumed
void signal_handler(int sig) {
    if (sig == SIGSTOP) {
        printf("Child process suspended.\n");
    } else if (sig == SIGCONT) {
        printf("Child process resumed.\n");
    }
}

int main() {
    pid_t pid;
    struct timespec ts = {1, 0};  // 1-second delay for demonstration

    // Register signal handlers
    signal(SIGSTOP, signal_handler);  // Not necessary as SIGSTOP is automatic
    signal(SIGCONT, signal_handler);

    pid = fork();
    if (pid == -1) {
        perror("fork failed");
        exit(1);
    }

    if (pid == 0) {
        // Child process
        printf("Child process started. PID: %d\n", getpid());

        while (1) {
            // Simulating some work in the child process
            printf("Child is working...\n");
            sleep(2);  // Sleep for 2 seconds to simulate work
        }
    } else {
        // Parent process
        printf("Parent process started. PID: %d\n", getpid());

        // Give some time for child to start
        sleep(3);

        // Suspend child process (send SIGSTOP)
        printf("Parent: Suspending child process.\n");
        kill(pid, SIGSTOP);  // Suspend the child process
```

```
        // Wait for some time before resuming the child
        nanosleep(&ts, NULL);

        // Resume child process (send SIGCONT)
        printf("Parent: Resuming child process.\n");
        kill(pid, SIGCONT);  // Resume the child process

        // Wait for child process to finish (terminate it)
        wait(NULL);
    }

    return 0;
}
```

---

26. Write a C program that illustrates inters process communication using shared memory.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <string.h>
#include <sys/wait.h>

#define SHM_SIZE 1024  // Size of shared memory
#define SHM_KEY 1234   // Key for shared memory segment

int main() {
    int shm_id;
    char *shm_ptr;

    // Create shared memory segment
    shm_id = shmget(SHM_KEY, SHM_SIZE, 0666 | IPC_CREAT);
    if (shm_id == -1) {
        perror("shmget failed");
        exit(1);
    }

    // Attach the shared memory segment to the address space of the process
    shm_ptr = (char *)shmat(shm_id, NULL, 0);
    if (shm_ptr == (char *)-1) {
        perror("shmat failed");
        exit(1);
    }

    pid_t pid = fork();  // Create a child process

    if (pid == -1) {
        perror("fork failed");
        exit(1);
    }

    if (pid == 0) {
        // Child process: Read from shared memory
```

```
        sleep(1);  // Wait for parent to write data
        printf("Child Process: Reading from shared memory: %s\n", shm_ptr);

        // Detach the shared memory
        if (shmdt(shm_ptr) == -1) {
            perror("shmdt failed in child");
            exit(1);
        }

        exit(0);
    } else {
        // Parent process: Write to shared memory
        printf("Parent Process: Writing to shared memory...\n");
        snprintf(shm_ptr, SHM_SIZE, "Hello from parent to child!");

        // Wait for child process to read the message
        wait(NULL);

        // Detach the shared memory in parent
        if (shmdt(shm_ptr) == -1) {
            perror("shmdt failed in parent");
            exit(1);
        }

        // Remove the shared memory segment
        if (shmctl(shm_id, IPC_RMID, NULL) == -1) {
            perror("shmctl failed");
            exit(1);
        }

        exit(0);
    }

    return 0;
}
```

---

chatgpt link to explain the codes
https://chatgpt.com/share/674988b1-24c4-800b-a50e-fcb388b8b05d

==========

==========