OS practical:

1. Write a C program to implement Indexed file allocation Strategy .

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

#define MAX_BLOCKS 100

// Structure to represent a block
typedef struct {
    int data; // Data stored in the block
} Block;

// Structure to represent a file
typedef struct {
    int index_block; // Index block pointer
    int size;        // Size of the file in blocks
} File;

// Function to initialize blocks
void initializeBlocks(Block blocks[], int num_blocks) {
    for (int i = 0; i < num_blocks; i++) {
        blocks[i].data = 0; // Initialize data of each block
    }
}

// Function to allocate blocks for a file using indexed allocation
void allocateBlocks(File *file, Block blocks[], int num_blocks) {
    // Allocate an index block
```

```c
    file->index_block = rand() % num_blocks;

    // Simulate allocation of data blocks and update index block pointers
    for (int i = 0; i < file->size; i++) {
        int data_block = rand() % num_blocks;
        blocks[file->index_block].data = data_block; // Update index block pointer
    }
}

// Function to display allocated blocks for a file
void displayAllocatedBlocks(File file, Block blocks[]) {
    printf("Index Block: %d\n", file.index_block);
    printf("Allocated Blocks:\n");
    int current_block = file.index_block;
    while (current_block != -1) {
        printf("%d ", current_block);
        current_block = blocks[current_block].data;
    }
    printf("\n");
}

int main() {
    Block blocks[MAX_BLOCKS];
    int num_blocks = sizeof(blocks) / sizeof(blocks[0]);

    // Initialize blocks
    initializeBlocks(blocks, num_blocks);

    // Example files
    File file1 = {0, 5}; // File with size 5 blocks
```

```
    File file2 = {0, 3}; // File with size 3 blocks

    // Allocate blocks for files using indexed allocation
    allocateBlocks(&file1, blocks, num_blocks);
    allocateBlocks(&file2, blocks, num_blocks);

    // Display allocated blocks for files
    printf("File 1:\n");
    displayAllocatedBlocks(file1, blocks);
    printf("\nFile 2:\n");
    displayAllocatedBlocks(file2, blocks);

    return 0;
}
```

2. Implement Least Recently Used page replacement algorithm using C program .

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

#define NUM_FRAMES 3
#define NUM_PAGES 20

// Function to find the index of the least recently used page
int findLRU(int page_frames[], int page_frame_times[], int num_frames) {
    int lru_index = 0;
    int min_time = page_frame_times[0];

    for (int i = 1; i < num_frames; i++) {
        if (page_frame_times[i] < min_time) {
            min_time = page_frame_times[i];
```

```
            lru_index = i;
        }
    }

    return lru_index;
}

// Function to check if a page is present in memory
bool pageExists(int page, int page_frames[], int num_frames) {
    for (int i = 0; i < num_frames; i++) {
        if (page_frames[i] == page) {
            return true;
        }
    }
    return false;
}

int main() {
    int reference_string[NUM_PAGES] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5};
    int num_pages = NUM_PAGES;

    int page_frames[NUM_FRAMES] = {-1}; // Initialize all page frames to -1
    int page_frame_times[NUM_FRAMES] = {0}; // Initialize time of page frames to 0

    int page_faults = 0;

    // Iterate through the reference string
    for (int i = 0; i < num_pages; i++) {
        int page = reference_string[i];
```

```c
        // Check if the page is in memory
        if (!pageExists(page, page_frames, NUM_FRAMES)) {
            // Page fault occurred
            page_faults++;

            // Find the least recently used page
            int lru_index = findLRU(page_frames, page_frame_times, NUM_FRAMES);

            // Replace the least recently used page with the current page
            page_frames[lru_index] = page;

            // Update the time of the replaced page frame
            page_frame_times[lru_index] = i + 1;
        } else {
            // Update the time of the page frame
            for (int j = 0; j < NUM_FRAMES; j++) {
                if (page_frames[j] == page) {
                    page_frame_times[j] = i + 1;
                    break;
                }
            }
        }
    }

    // Print the number of page faults
    printf("Number of Page Faults: %d\n", page_faults);

    return 0;
}
```

3. Write a C program to implement Best Fit Memory Allocation Method .

```
#include <stdio.h>

#define MAX_BLOCKS 100

// Structure to represent a memory block
typedef struct {
    int start_address; // Starting address of the block
    int size;          // Size of the block
    int process_id;    // ID of the process occupying the block (-1 if block is free)
} MemoryBlock;

// Function to initialize memory blocks
void initializeMemoryBlocks(MemoryBlock memory_blocks[], int num_blocks) {
    for (int i = 0; i < num_blocks; i++) {
        memory_blocks[i].start_address = i * 100; // Start address of each block
        memory_blocks[i].size = 100;            // Each block has size 100
        memory_blocks[i].process_id = -1;       // Initialize all blocks as free
    }
}

// Function to display memory blocks
void displayMemoryBlocks(MemoryBlock memory_blocks[], int num_blocks) {
    printf("Memory Blocks:\n");
    printf("Start Address\tSize\tProcess ID\n");
    for (int i = 0; i < num_blocks; i++) {
        printf("%d\t\t%d\t%d\n", memory_blocks[i].start_address,
memory_blocks[i].size, memory_blocks[i].process_id);
    }
}
```

```
// Function to allocate memory to a process using Best Fit method
void allocateMemoryBestFit(MemoryBlock memory_blocks[], int num_blocks, int process_id, int size) {
    int best_fit_index = -1;
    int min_size = __INT_MAX__;

    // Find the best fit block (smallest free block that can accommodate the process)
    for (int i = 0; i < num_blocks; i++) {
        if (memory_blocks[i].process_id == -1 && memory_blocks[i].size >= size && memory_blocks[i].size < min_size) {
            min_size = memory_blocks[i].size;
            best_fit_index = i;
        }
    }

    if (best_fit_index != -1) {
        // Allocate memory to the process in the best fit block
        memory_blocks[best_fit_index].process_id = process_id;
    } else {
        // If no suitable block is found, print error message
        printf("Error: No suitable block found for process %d with size %d\n", process_id, size);
    }
}

int main() {
    MemoryBlock memory_blocks[MAX_BLOCKS];
    int num_blocks = sizeof(memory_blocks) / sizeof(memory_blocks[0]);

    // Initialize memory blocks
```

```
    initializeMemoryBlocks(memory_blocks, num_blocks);

    // Display initial memory blocks
    displayMemoryBlocks(memory_blocks, num_blocks);

    // Allocate memory to processes using Best Fit method
    allocateMemoryBestFit(memory_blocks, num_blocks, 1, 50);
    allocateMemoryBestFit(memory_blocks, num_blocks, 2, 70);
    allocateMemoryBestFit(memory_blocks, num_blocks, 3, 30);

    // Display memory blocks after allocation
    printf("\nMemory blocks after allocation:\n");
    displayMemoryBlocks(memory_blocks, num_blocks);

    return 0;
}
```

4. Write a C program to implement Worst Fit Memory Allocation Method

```
#include <stdio.h>

#define MAX_BLOCKS 100

// Structure to represent a memory block
typedef struct {
    int start_address; // Starting address of the block
    int size;          // Size of the block
    int process_id;    // ID of the process occupying the block (-1 if block is free)
} MemoryBlock;

// Function to initialize memory blocks
```

```
void initializeMemoryBlocks(MemoryBlock memory_blocks[], int num_blocks) {
    for (int i = 0; i < num_blocks; i++) {
        memory_blocks[i].start_address = i * 100; // Start address of each block
        memory_blocks[i].size = 100;            // Each block has size 100
        memory_blocks[i].process_id = -1;       // Initialize all blocks as free
    }
}

// Function to display memory blocks
void displayMemoryBlocks(MemoryBlock memory_blocks[], int num_blocks) {
    printf("Memory Blocks:\n");
    printf("Start Address\tSize\tProcess ID\n");
    for (int i = 0; i < num_blocks; i++) {
        printf("%d\t\t%d\t%d\n", memory_blocks[i].start_address,
memory_blocks[i].size, memory_blocks[i].process_id);
    }
}

// Function to allocate memory to a process using Worst Fit method
void allocateMemoryWorstFit(MemoryBlock memory_blocks[], int num_blocks, int
process_id, int size) {
    int worst_fit_index = -1;
    int max_size = -1;

    // Find the worst fit block (largest free block)
    for (int i = 0; i < num_blocks; i++) {
        if (memory_blocks[i].process_id == -1 && memory_blocks[i].size >= size &&
memory_blocks[i].size > max_size) {
            max_size = memory_blocks[i].size;
            worst_fit_index = i;
        }
```

```
    }

    if (worst_fit_index != -1) {
        // Allocate memory to the process in the worst fit block
        memory_blocks[worst_fit_index].process_id = process_id;
    } else {
        // If no suitable block is found, print error message
        printf("Error: No suitable block found for process %d with size %d\n",
process_id, size);
    }
}

int main() {
    MemoryBlock memory_blocks[MAX_BLOCKS];
    int num_blocks = sizeof(memory_blocks) / sizeof(memory_blocks[0]);

    // Initialize memory blocks
    initializeMemoryBlocks(memory_blocks, num_blocks);

    // Display initial memory blocks
    displayMemoryBlocks(memory_blocks, num_blocks);

    // Allocate memory to processes using Worst Fit method
    allocateMemoryWorstFit(memory_blocks, num_blocks, 1, 50);
    allocateMemoryWorstFit(memory_blocks, num_blocks, 2, 70);
    allocateMemoryWorstFit(memory_blocks, num_blocks, 3, 30);

    // Display memory blocks after allocation
    printf("\nMemory blocks after allocation:\n");
    displayMemoryBlocks(memory_blocks, num_blocks);
```

```
    return 0;
}
```

5. Write a C program to implement First Fit Memory Allocation Method

```
#include <stdio.h>

#define MAX_BLOCKS 100

// Structure to represent a memory block
typedef struct {
    int start_address; // Starting address of the block
    int size;          // Size of the block
    int process_id;    // ID of the process occupying the block (-1 if block is free)
} MemoryBlock;

// Function to initialize memory blocks
void initializeMemoryBlocks(MemoryBlock memory_blocks[], int num_blocks) {
    for (int i = 0; i < num_blocks; i++) {
        memory_blocks[i].start_address = i * 100; // Start address of each block
        memory_blocks[i].size = 100;              // Each block has size 100
        memory_blocks[i].process_id = -1;         // Initialize all blocks as free
    }
}

// Function to display memory blocks
void displayMemoryBlocks(MemoryBlock memory_blocks[], int num_blocks) {
    printf("Memory Blocks:\n");
    printf("Start Address\tSize\tProcess ID\n");
    for (int i = 0; i < num_blocks; i++) {
        printf("%d\t\t%d\t%d\n", memory_blocks[i].start_address,
memory_blocks[i].size, memory_blocks[i].process_id);
```

```
    }
}

// Function to allocate memory to a process using First Fit method
void allocateMemoryFirstFit(MemoryBlock memory_blocks[], int num_blocks, int
process_id, int size) {
    for (int i = 0; i < num_blocks; i++) {
        if (memory_blocks[i].process_id == -1 && memory_blocks[i].size >= size) {
            // If the block is free and has sufficient size, allocate memory to the process
            memory_blocks[i].process_id = process_id;
            return;
        }
    }
    // If no suitable block is found, print error message
    printf("Error: No suitable block found for process %d with size %d\n", process_id,
size);
}

int main() {
    MemoryBlock memory_blocks[MAX_BLOCKS];
    int num_blocks = sizeof(memory_blocks) / sizeof(memory_blocks[0]);

    // Initialize memory blocks
    initializeMemoryBlocks(memory_blocks, num_blocks);

    // Display initial memory blocks
    displayMemoryBlocks(memory_blocks, num_blocks);

    // Allocate memory to processes using First Fit method
    allocateMemoryFirstFit(memory_blocks, num_blocks, 1, 50);
    allocateMemoryFirstFit(memory_blocks, num_blocks, 2, 70);
```

```c
    allocateMemoryFirstFit(memory_blocks, num_blocks, 3, 30);

    // Display memory blocks after allocation
    printf("\nMemory blocks after allocation:\n");
    displayMemoryBlocks(memory_blocks, num_blocks);

    return 0;
}
```

6. Write a C program to implement Shortest Job First Scheduling algorithm

```c
#include <stdio.h>

// Structure to represent a process
typedef struct {
    int process_id; // Process ID
    int arrival_time; // Arrival time
    int burst_time; // Burst time
} Process;

// Function to swap two processes
void swap(Process *a, Process *b) {
    Process temp = *a;
    *a = *b;
    *b = temp;
}

// Function to sort processes by burst time (SJF)
void sortProcessesByBurstTime(Process processes[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
```

```
            if (processes[j].burst_time > processes[j + 1].burst_time) {
                swap(&processes[j], &processes[j + 1]);
            }
        }
    }
}

// Function to calculate waiting time and turnaround time for each process
void calculateTimes(Process processes[], int n, int waiting_time[], int
turnaround_time[]) {
    waiting_time[0] = 0; // Waiting time for the first process is 0

    // Calculate waiting time for each process
    for (int i = 1; i < n; i++) {
        waiting_time[i] = waiting_time[i - 1] + processes[i - 1].burst_time;
    }

    // Calculate turnaround time for each process
    for (int i = 0; i < n; i++) {
        turnaround_time[i] = waiting_time[i] + processes[i].burst_time;
    }
}

// Function to calculate average waiting time and average turnaround time
void calculateAverageTimes(Process processes[], int n) {
    int waiting_time[n], turnaround_time[n];

    // Sort processes by burst time (SJF)
    sortProcessesByBurstTime(processes, n);

    // Calculate waiting time and turnaround time for each process
```

```
    calculateTimes(processes, n, waiting_time, turnaround_time);

    // Calculate total waiting time and total turnaround time
    int total_waiting_time = 0, total_turnaround_time = 0;
    for (int i = 0; i < n; i++) {
        total_waiting_time += waiting_time[i];
        total_turnaround_time += turnaround_time[i];
    }

    // Calculate average waiting time and average turnaround time
    float avg_waiting_time = (float)total_waiting_time / n;
    float avg_turnaround_time = (float)total_turnaround_time / n;

    // Print average waiting time and average turnaround time
    printf("Average Waiting Time: %.2f\n", avg_waiting_time);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);
}

int main() {
    // Example processes (process_id, arrival_time, burst_time)
    Process processes[] = {
        {1, 0, 6},
        {2, 3, 8},
        {3, 4, 7},
        {4, 7, 3},
        {5, 10, 4}
    };
    int n = sizeof(processes) / sizeof(processes[0]);

    // Calculate and print average waiting time and average turnaround time
```

```
    calculateAverageTimes(processes, n);

    return 0;
}
```

7. Implement the paging concept using C program

```
#include <stdio.h>

#include <stdbool.h>

#define PAGE_SIZE 4096 // Page size in bytes

#define NUM_PAGES 8    // Number of pages in the virtual address space

#define NUM_FRAMES 4   // Number of frames in physical memory

// Page table entry structure

typedef struct {

    bool valid;      // Valid bit to indicate if the page is present in memory

    int frame_number; // Physical frame number where the page is stored

} PageTableEntry;

int main() {

    // Initialize page table

    PageTableEntry page_table[NUM_PAGES];

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

        page_table[i].valid = false;

        page_table[i].frame_number = -1;

    }

    // Virtual memory access simulation

    int virtual_address;

    printf("Enter virtual address (-1 to exit): ");

    scanf("%d", &virtual_address);
```

```
    while (virtual_address != -1) {
        // Calculate page number and offset from the virtual address
        int page_number = virtual_address / PAGE_SIZE;
        int offset = virtual_address % PAGE_SIZE;


        // Check if the page is in memory
        if (page_table[page_number].valid) {
            // Calculate the physical address using the frame number and offset
            int physical_address = page_table[page_number].frame_number *
PAGE_SIZE + offset;
            printf("Physical address: %d\n", physical_address);
        } else {
            printf("Page fault: Page %d is not in memory.\n", page_number);
            // Handle page fault (e.g., load the page into memory)
            // For simplicity, we just set the valid bit and frame number in the page table
            page_table[page_number].valid = true;
            page_table[page_number].frame_number = page_number % NUM_FRAMES;
// Simple page replacement policy
            printf("Page %d loaded into frame %d.\n", page_number,
page_table[page_number].frame_number);
        }


        // Prompt for the next virtual address
        printf("Enter virtual address (-1 to exit): ");
        scanf("%d", &virtual_address);
    }


    return 0;
}
```

8. . Implement LFU page replacement algorithm using C program

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

```
#define MAX_FRAMES 3
#define MAX_PAGES 20

// Function to check if a page exists in memory
bool pageExists(int page, int frames[], int num_frames) {
    for (int i = 0; i < num_frames; i++) {
        if (frames[i] == page) {
            return true;
        }
    }
    return false;
}

// Function to find the index of the least frequently used page
int findLFU(int pages[], int num_frames, int page_frequency[], int num_pages) {
    int min_index = 0;
    int min_frequency = page_frequency[0];

    for (int i = 1; i < num_frames; i++) {
        if (page_frequency[i] < min_frequency) {
            min_index = i;
            min_frequency = page_frequency[i];
        }
    }

    return min_index;
}

int main() {
```

```c
    int reference_string[MAX_PAGES] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5};
    int num_pages = 12;

    int frames[MAX_FRAMES];
    int num_frames = 0;

    // Initialize frames with -1 indicating empty frames
    for (int i = 0; i < MAX_FRAMES; i++) {
        frames[i] = -1;
    }

    // Frequency array to keep track of the usage frequency of each page
    int page_frequency[MAX_FRAMES] = {0};

    // Iterate through the reference string
    for (int i = 0; i < num_pages; i++) {
        // If the page is not in memory, replace the least frequently used page
        if (!pageExists(reference_string[i], frames, num_frames)) {
            if (num_frames < MAX_FRAMES) {
                // If there is empty space in memory, add the page
                frames[num_frames] = reference_string[i];
                num_frames++;
            } else {
                // Find the least frequently used page
                int lfu_index = findLFU(frames, num_frames, page_frequency,
num_pages);
                // Replace the least frequently used page
                frames[lfu_index] = reference_string[i];
            }
        }
        // Update the usage frequency of each page
```

```
        for (int j = 0; j < num_frames; j++) {
            if (frames[j] == reference_string[i]) {
                page_frequency[j]++;
            }
        }
    }

    return 0;
}
```

9. Implement First come first serve page replacement algorithm using C program

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

#define MAX_FRAMES 3
#define MAX_PAGES 20

// Function to check if a page exists in memory
bool pageExists(int page, int frames[], int num_frames) {
    for (int i = 0; i < num_frames; i++) {
        if (frames[i] == page) {
            return true;
        }
    }
    return false;
}

// Function to display the current state of memory
void displayMemory(int frames[], int num_frames) {
    printf("Memory: ");
    for (int i = 0; i < num_frames; i++) {
        if (frames[i] == -1) {
```

```
            printf("[ ] ");
        } else {
            printf("[%d] ", frames[i]);
        }
    }
    printf("\n");
}

int main() {
    int reference_string[MAX_PAGES] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5};
    int num_pages = 12;

    int frames[MAX_FRAMES];
    int num_frames = 0;

    // Initialize frames with -1 indicating empty frames
    for (int i = 0; i < MAX_FRAMES; i++) {
        frames[i] = -1;
    }

    // Iterate through the reference string
    for (int i = 0; i < num_pages; i++) {
        // If the page is not in memory, replace the oldest page (FCFS)
        if (!pageExists(reference_string[i], frames, num_frames)) {
            if (num_frames < MAX_FRAMES) {
                // If there is empty space in memory, add the page
                frames[num_frames] = reference_string[i];
                num_frames++;
            } else {
                // Replace the oldest page
```

```
            for (int j = 0; j < MAX_FRAMES - 1; j++) {
                frames[j] = frames[j + 1];
            }
            frames[MAX_FRAMES - 1] = reference_string[i];
        }
        // Display memory after page replacement
        displayMemory(frames, num_frames);
    }
  }

  return 0;
}
```

10. Implement bankers algorithm for deadlock detection using C program

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

// Maximum number of processes and resources
#define MAX_PROCESSES 5
#define MAX_RESOURCES 3

// Available resources
int available[MAX_RESOURCES];

// Maximum demand of each process
int maximum[MAX_PROCESSES][MAX_RESOURCES];

// Currently allocated resources to each process
int allocation[MAX_PROCESSES][MAX_RESOURCES];

// Remaining need of each process
```

```
int need[MAX_PROCESSES][MAX_RESOURCES];

// Number of processes and resources
int num_processes, num_resources;

// Function to initialize the resources and allocation matrices
void initialize() {
    // Available resources
    printf("Enter the available resources:\n");
    for (int i = 0; i < num_resources; i++) {
        scanf("%d", &available[i]);
    }

    // Maximum demand of each process
    printf("Enter the maximum demand of each process:\n");
    for (int i = 0; i < num_processes; i++) {
        printf("For process %d: ", i);
        for (int j = 0; j < num_resources; j++) {
            scanf("%d", &maximum[i][j]);
        }
    }

    // Currently allocated resources to each process
    printf("Enter the currently allocated resources to each process:\n");
    for (int i = 0; i < num_processes; i++) {
        printf("For process %d: ", i);
        for (int j = 0; j < num_resources; j++) {
            scanf("%d", &allocation[i][j]);
            // Calculate remaining need
            need[i][j] = maximum[i][j] - allocation[i][j];
```

```
        }
    }
}

// Function to check if the current state is safe
bool isSafe(int process, int request[]) {
    // Temporary arrays to store the current state
    int temp_available[MAX_RESOURCES];
    int temp_allocation[MAX_PROCESSES][MAX_RESOURCES];
    int temp_need[MAX_PROCESSES][MAX_RESOURCES];

    // Copy current state to temporary arrays
    for (int i = 0; i < num_resources; i++) {
        temp_available[i] = available[i];
    }
    for (int i = 0; i < num_processes; i++) {
        for (int j = 0; j < num_resources; j++) {
            temp_allocation[i][j] = allocation[i][j];
            temp_need[i][j] = need[i][j];
        }
    }

    // Try to allocate requested resources to the process
    for (int i = 0; i < num_resources; i++) {
        if (request[i] > temp_available[i] || request[i] > temp_need[process][i]) {
            return false;
        }
        temp_available[i] -= request[i];
        temp_allocation[process][i] += request[i];
        temp_need[process][i] -= request[i];
```

```
}

// Check if the new state is safe
bool finish[num_processes];
for (int i = 0; i < num_processes; i++) {
    finish[i] = false;
}

int work[MAX_RESOURCES];
for (int i = 0; i < num_resources; i++) {
    work[i] = temp_available[i];
}

while (true) {
    bool found = false;
    for (int i = 0; i < num_processes; i++) {
        if (!finish[i]) {
            int j;
            for (j = 0; j < num_resources; j++) {
                if (temp_need[i][j] > work[j]) {
                    break;
                }
            }
            if (j == num_resources) {
                for (int k = 0; k < num_resources; k++) {
                    work[k] += temp_allocation[i][k];
                }
                finish[i] = true;
                found = true;
            }
```

```
            }
        }
        if (!found) {
            break;
        }
    }

    // Check if all processes are finished
    for (int i = 0; i < num_processes; i++) {
        if (!finish[i]) {
            return false;
        }
    }

    return true;
}

// Function to perform resource request and check for deadlock
void resourceRequest(int process) {
    int request[MAX_RESOURCES];

    // Get resource request from user
    printf("Enter the resource request for process %d: ", process);
    for (int i = 0; i < num_resources; i++) {
        scanf("%d", &request[i]);
    }

    // Check if the request is within the maximum demand
    for (int i = 0; i < num_resources; i++) {
        if (request[i] > need[process][i]) {
```

```c
            printf("Error: Request exceeds maximum demand.\n");
            return;
        }
    }

    // Check if the request can be granted
    if (isSafe(process, request)) {
        // Grant the request
        for (int i = 0; i < num_resources; i++) {
            available[i] -= request[i];
            allocation[process][i] += request[i];
            need[process][i] -= request[i];
        }
        printf("Resource request granted.\n");
    } else {
        printf("Resource request denied (unsafe state).\n");
    }
}

int main() {
    // Input number of processes and resources
    printf("Enter the number of processes: ");
    scanf("%d", &num_processes);
    printf("Enter the number of resources: ");
    scanf("%d", &num_resources);

    // Initialize resources and allocation matrices
    initialize();

    // Perform resource request and check for deadlock
```

```
    while (true) {
        int process;
        printf("Enter the process requesting resources (0-%d), -1 to exit: ",
num_processes - 1);
        scanf("%d", &process);
        if (process == -1) {
            break;
        }
        if (process < 0 || process >= num_processes) {
            printf("Error: Invalid process ID.\n");
            continue;
        }
        resourceRequest(process);
    }

    return 0;
}
```

1. 11. Write a C program to implement First Come First Serve Scheduling algorithm

```
#include <stdio.h>

// Structure to represent a process
typedef struct {
    int process_id;
    int arrival_time;
    int burst_time;
    int completion_time;
    int turnaround_time;
    int waiting_time;
} Process;
```

```
// Function to calculate completion time, turnaround time, and waiting time for each process
void calculateTimes(Process processes[], int n) {
    int total_waiting_time = 0;
    int total_turnaround_time = 0;

    // Calculate completion time for the first process
    processes[0].completion_time = processes[0].arrival_time + processes[0].burst_time;
    processes[0].turnaround_time = processes[0].completion_time - processes[0].arrival_time;
    processes[0].waiting_time = 0;

    total_waiting_time += processes[0].waiting_time;
    total_turnaround_time += processes[0].turnaround_time;

    // Calculate completion time, turnaround time, and waiting time for subsequent processes
    for (int i = 1; i < n; i++) {
        processes[i].completion_time = processes[i - 1].completion_time + processes[i].burst_time;
        processes[i].turnaround_time = processes[i].completion_time - processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time - processes[i].burst_time;

        total_waiting_time += processes[i].waiting_time;
        total_turnaround_time += processes[i].turnaround_time;
    }

    // Calculate average waiting time and average turnaround time
    double average_waiting_time = (double)total_waiting_time / n;
    double average_turnaround_time = (double)total_turnaround_time / n;
```

```
    // Print the results
    printf("Process\t Arrival Time\t Burst Time\t Completion Time\t Turnaround Time\t
Waiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t %d\t\t %d\t\t %d\t\t\t %d\t\t\t %d\n", processes[i].process_id,
               processes[i].arrival_time, processes[i].burst_time,
processes[i].completion_time,
               processes[i].turnaround_time, processes[i].waiting_time);
    }
    printf("\nAverage Waiting Time: %.2f\n", average_waiting_time);
    printf("Average Turnaround Time: %.2f\n", average_turnaround_time);
}

int main() {
    // Number of processes
    int n;

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

    // Array to store information about processes
    Process processes[n];

    // Input process details
    for (int i = 0; i < n; i++) {
        printf("Enter arrival time and burst time for process %d: ", i + 1);
        scanf("%d %d", &processes[i].arrival_time, &processes[i].burst_time);
        processes[i].process_id = i + 1;
    }
```

```
    // Calculate completion time, turnaround time, and waiting time for each process
    calculateTimes(processes, n);

    return 0;
}
```

12. Write a C program to implement Linked list file allocation Strategy

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

// Structure to represent a disk block
typedef struct Block {
    int block_number;
    struct Block *next;
} Block;

// Structure to represent a file
typedef struct File {
    int file_id;
    Block *blocks;
    struct File *next;
} File;

// Structure to represent the disk
typedef struct {
    File *files;
} Disk;

// Initialize disk
void initDisk(Disk *disk) {
    disk->files = NULL;
```

```
}

// Create a new file
void createFile(Disk *disk, int file_id, int num_blocks) {
    File *new_file = (File *)malloc(sizeof(File));
    if (new_file == NULL) {
        printf("Error: Memory allocation failed.\n");
        return;
    }
    new_file->file_id = file_id;
    new_file->blocks = NULL;
    new_file->next = NULL;

    // Allocate disk blocks for the file
    for (int i = 1; i <= num_blocks; i++) {
        Block *new_block = (Block *)malloc(sizeof(Block));
        if (new_block == NULL) {
            printf("Error: Memory allocation failed.\n");
            return;
        }
        new_block->block_number = i;
        new_block->next = new_file->blocks;
        new_file->blocks = new_block;
    }

    // Add file to disk
    new_file->next = disk->files;
    disk->files = new_file;
}
```

```
// Print file allocation details
void printDisk(Disk *disk) {
    printf("Files on disk:\n");
    File *curr_file = disk->files;
    while (curr_file != NULL) {
        printf("File ID: %d, Blocks: ", curr_file->file_id);
        Block *curr_block = curr_file->blocks;
        while (curr_block != NULL) {
            printf("%d ", curr_block->block_number);
            curr_block = curr_block->next;
        }
        printf("\n");
        curr_file = curr_file->next;
    }
}

// Free memory allocated for files and blocks
void cleanupDisk(Disk *disk) {
    File *curr_file = disk->files;
    while (curr_file != NULL) {
        Block *curr_block = curr_file->blocks;
        while (curr_block != NULL) {
            Block *temp = curr_block;
            curr_block = curr_block->next;
            free(temp);
        }
        File *temp = curr_file;
        curr_file = curr_file->next;
        free(temp);
    }
```

```
    disk->files = NULL;
}

int main() {
    Disk disk;
    initDisk(&disk);

    // Create some files with disk blocks
    createFile(&disk, 1, 3);
    createFile(&disk, 2, 4);
    createFile(&disk, 3, 2);

    // Print file allocation details
    printDisk(&disk);

    // Cleanup disk to free memory
    cleanupDisk(&disk);

    return 0;
}
```

13. Write a C program to implement the UNIX commands ‘cp, ls, grep’

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

// Implementation of cp command: Copy file contents from source file to destination file
void cp(const char *source_file, const char *destination_file) {
    FILE *source, *destination;
    char ch;
```

```
    source = fopen(source_file, "r");
    if (source == NULL) {
        printf("Error: Unable to open source file.\n");
        return;
    }

    destination = fopen(destination_file, "w");
    if (destination == NULL) {
        printf("Error: Unable to create destination file.\n");
        fclose(source);
        return;
    }

    while ((ch = fgetc(source)) != EOF) {
        fputc(ch, destination);
    }

    fclose(source);
    fclose(destination);
    printf("File copied successfully.\n");
}

// Implementation of ls command: List files in the current directory
void ls() {
    FILE *pipe = popen("ls", "r");
    if (pipe == NULL) {
        printf("Error: Unable to execute ls command.\n");
        return;
    }
```

```
    char buffer[128];
    while (fgets(buffer, sizeof(buffer), pipe) != NULL) {
        printf("%s", buffer);
    }

    pclose(pipe);
}

// Implementation of grep command: Search for a pattern in a file
void grep(const char *pattern, const char *filename) {
    FILE *pipe;
    char command[128];
    snprintf(command, sizeof(command), "grep \"%s\" %s", pattern, filename);

    pipe = popen(command, "r");
    if (pipe == NULL) {
        printf("Error: Unable to execute grep command.\n");
        return;
    }

    char buffer[128];
    while (fgets(buffer, sizeof(buffer), pipe) != NULL) {
        printf("%s", buffer);
    }

    pclose(pipe);
}

int main() {
    // Example usage of cp command
```

```
    cp("source.txt", "destination.txt");

    // Example usage of ls command
    printf("List of files in the current directory:\n");
    ls();

    // Example usage of grep command
    printf("Lines containing 'example' in 'sample.txt':\n");
    grep("example", "sample.txt");

    return 0;
}
```

14. How the data is allocated sequentially, Write a C program to implement

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

#define MAX_SIZE 100

// Global array to hold the data
int data[MAX_SIZE];
int next_index = 0;

// Function to allocate data sequentially
int allocate(int value) {
    if (next_index >= MAX_SIZE) {
        printf("Error: Maximum data size reached.\n");
        return -1; // Error code indicating failure
    }

    data[next_index] = value;
```

```
    next_index++;
    return next_index - 1; // Return the index where the data is allocated
}

// Function to print the allocated data
void printData() {
    printf("Allocated data: ");
    for (int i = 0; i < next_index; i++) {
        printf("%d ", data[i]);
    }
    printf("\n");
}

int main() {
    // Allocate some data sequentially
    int index1 = allocate(10);
    int index2 = allocate(20);
    int index3 = allocate(30);

    // Check if allocation was successful
    if (index1 != -1 && index2 != -1 && index3 != -1) {
        printf("Data allocated successfully.\n");
        // Print the allocated data
        printData();
    }

    return 0;
}
```

15. Write a C program to implement Producer-Consumer Problem using semaphore concept

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

```
#include <pthread.h>
#include <semaphore.h>

#define BUFFER_SIZE 5

// Shared buffer
int buffer[BUFFER_SIZE];
int in = 0, out = 0;

// Semaphore declarations
sem_t empty, full, mutex;

// Producer function
void *producer(void *arg) {
    int item = 1;

    while (1) {
        // Produce item
        sleep(1);

        // Wait for empty buffer slot
        sem_wait(&empty);
        // Acquire mutex lock
        sem_wait(&mutex);

        // Add item to buffer
        buffer[in] = item;
        printf("Produced item: %d\n", item);
        in = (in + 1) % BUFFER_SIZE;
        item++;
```

```
        }

        // Release mutex lock
        sem_post(&mutex);
        // Signal full buffer
        sem_post(&full);
    }
}

// Consumer function
void *consumer(void *arg) {
    while (1) {
        // Wait for full buffer
        sem_wait(&full);
        // Acquire mutex lock
        sem_wait(&mutex);

        // Remove item from buffer
        int item = buffer[out];
        printf("Consumed item: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;

        // Release mutex lock
        sem_post(&mutex);
        // Signal empty buffer
        sem_post(&empty);

        // Consume item
        sleep(2);
    }
}
```

```c
int main() {
    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);

    // Create producer and consumer threads
    pthread_t producer_thread, consumer_thread;
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    // Wait for threads to finish
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    // Destroy semaphores
    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
```

16. Write a C program to implement First Round Robin Scheduling algorithm

```c
#include <stdio.h>

// Structure to represent a process
struct Process {
    int id;
    int arrival_time;
```

```c
    int burst_time;
};

// Function to perform First Come First Serve (FCFS) Scheduling
void fcfsScheduling(struct Process processes[], int n) {
    int current_time = 0;
    int total_wait_time = 0;
    int total_turnaround_time = 0;

    printf("Process\tArrival Time\tBurst Time\tWait Time\tTurnaround Time\n");

    // Calculate wait time and turnaround time for each process
    for (int i = 0; i < n; i++) {
        // Wait time for current process
        int wait_time = current_time - processes[i].arrival_time;
        if (wait_time < 0) {
            wait_time = 0; // Process arrived after current time
        }
        total_wait_time += wait_time;

        // Turnaround time for current process
        int turnaround_time = wait_time + processes[i].burst_time;
        total_turnaround_time += turnaround_time;

        // Print process details
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrival_time,
processes[i].burst_time, wait_time, turnaround_time);

        // Update current time
        current_time += processes[i].burst_time;
    }
```

```
    // Calculate average wait time and average turnaround time
    float avg_wait_time = (float)total_wait_time / n;
    float avg_turnaround_time = (float)total_turnaround_time / n;

    // Print average wait time and average turnaround time
    printf("\nAverage Wait Time: %.2f\n", avg_wait_time);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);
}

int main() {
    // Number of processes
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    // Array to store processes
    struct Process processes[n];

    // Input process details
    for (int i = 0; i < n; i++) {
        printf("Enter details for Process %d:\n", i + 1);
        processes[i].id = i + 1;
        printf("Arrival Time: ");
        scanf("%d", &processes[i].arrival_time);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
    }

    // Perform First Come First Serve (FCFS) Scheduling
```

```
    fcfsScheduling(processes, n);


    return 0;
}
```

17. Write a C program to implement Priority Scheduling algorithm

```
#include <stdio.h>


// Structure to represent a process
struct Process {
    int id;
    int priority;
    int burst_time;
};


// Function to perform Priority Scheduling
void priorityScheduling(struct Process processes[], int n) {
    // Sort processes based on priority
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].priority > processes[j + 1].priority) {
                // Swap processes
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }


    int current_time = 0;
    int total_wait_time = 0;
    int total_turnaround_time = 0;
```

```
}

    printf("Process\tPriority\tBurst Time\tWait Time\tTurnaround Time\n");

    // Calculate wait time and turnaround time for each process
    for (int i = 0; i < n; i++) {
        // Wait time for current process
        int wait_time = current_time;
        total_wait_time += wait_time;

        // Turnaround time for current process
        int turnaround_time = wait_time + processes[i].burst_time;
        total_turnaround_time += turnaround_time;

        // Print process details
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].priority,
processes[i].burst_time, wait_time, turnaround_time);

        // Update current time
        current_time += processes[i].burst_time;
    }

    // Calculate average wait time and average turnaround time
    float avg_wait_time = (float)total_wait_time / n;
    float avg_turnaround_time = (float)total_turnaround_time / n;

    // Print average wait time and average turnaround time
    printf("\nAverage Wait Time: %.2f\n", avg_wait_time);
    printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);
}

int main() {
```

```
    // Number of processes
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    // Array to store processes
    struct Process processes[n];

    // Input process details
    for (int i = 0; i < n; i++) {
        printf("Enter details for Process %d:\n", i + 1);
        processes[i].id = i + 1;
        printf("Priority: ");
        scanf("%d", &processes[i].priority);
        printf("Burst Time: ");
        scanf("%d", &processes[i].burst_time);
    }

    // Perform Priority Scheduling
    priorityScheduling(processes, n);

    return 0;
}
```

18. Implement pipe concept in Inter Process Communication using C program

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

int main() {
    int pipefd[2];
```

```c
pid_t pid;
char message[] = "Hello, child process!";

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

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

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

if (pid == 0) { // Child process
    close(pipefd[1]); // Close the write end of the pipe in the child process

    char buffer[100];
    // Read from the pipe
    read(pipefd[0], buffer, sizeof(buffer));
    printf("Child Process: Received message from parent: %s\n", buffer);

    // Close the read end of the pipe
    close(pipefd[0]);
    exit(EXIT_SUCCESS);
} else { // Parent process
    close(pipefd[0]); // Close the read end of the pipe in the parent process
```

```
        // Write to the pipe
        write(pipefd[1], message, sizeof(message));
        printf("Parent Process: Sent message to child: %s\n", message);

        // Close the write end of the pipe
        close(pipefd[1]);
    }

    return 0;
}
```

19. Implement the concept of threading and synchronization using C program

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

#define NUM_THREADS 2
#define MAX_COUNT 10000

int counter = 0;
pthread_mutex_t mutex;

void *increment(void *arg) {
    for (int i = 0; i < MAX_COUNT; i++) {
        pthread_mutex_lock(&mutex);
        counter++;
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}
```

```
void *decrement(void *arg) {
    for (int i = 0; i < MAX_COUNT; i++) {
        pthread_mutex_lock(&mutex);
        counter--;
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    pthread_mutex_init(&mutex, NULL);

    // Create threads
    if (pthread_create(&threads[0], NULL, increment, NULL) != 0) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
    }
    if (pthread_create(&threads[1], NULL, decrement, NULL) != 0) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
    }

    // Wait for threads to finish
    for (int i = 0; i < NUM_THREADS; i++) {
        if (pthread_join(threads[i], NULL) != 0) {
            perror("pthread_join");
            exit(EXIT_FAILURE);
        }
```

```
    }

    // Destroy mutex
    pthread_mutex_destroy(&mutex);

    printf("Final Counter Value: %d\n", counter);

    return 0;
}
```

20. Write a shell program to find the given number is odd or even

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    // Check if the number of command-line arguments is correct
    if (argc != 2) {
        printf("Usage: %s <number>\n", argv[0]);
        return 1;
    }

    // Convert the command-line argument to an integer
    int number = atoi(argv[1]);

    // Check if the number is odd or even
    if (number % 2 == 0) {
        printf("even");
    } else {
        printf("odd");
    }

    return 0;
```

}

21. Create a shell program to perform arithmetic operations

```
#!/bin/bash

# Function to perform addition
add() {
   result=$(($1 + $2))
   echo "Result: $result"
}

# Function to perform subtraction
subtract() {
   result=$(($1 - $2))
   echo "Result: $result"
}

# Function to perform multiplication
multiply() {
   result=$(($1 * $2))
   echo "Result: $result"
}

# Function to perform division
divide() {
   if [ $2 -eq 0 ]; then
      echo "Error: Division by zero is not allowed."
   else
      result=$(($1 / $2))
      echo "Result: $result"
```

```
    fi
}

# Main function
main() {
    echo "Enter first number:"
    read num1
    echo "Enter second number:"
    read num2
    echo "Enter operator (+, -, *, /):"
    read operator

    case $operator in
        +) add $num1 $num2 ;;
        -) subtract $num1 $num2 ;;
        \*) multiply $num1 $num2 ;;
        /) divide $num1 $num2 ;;
        *) echo "Error: Invalid operator." ;;
    esac
}

# Execute main function
main
```

22. Create a shell program to find the greatest among the given three numbers

```
#!/bin/bash

# Function to find the greatest among three numbers
find_greatest() {
    if [ $1 -gt $2 ] && [ $1 -gt $3 ]; then
```

```
    echo "The greatest number is $1"
  elif [ $2 -gt $1 ] && [ $2 -gt $3 ]; then
    echo "The greatest number is $2"
  else
    echo "The greatest number is $3"
  fi
}

# Main function
main() {
  echo "Enter three numbers:"
  read num1 num2 num3
  find_greatest $num1 $num2 $num3
}

# Execute main function
main
```

23. . Write a shell program to find the factorial of a given number (20)

```
#!/bin/bash

# Function to calculate factorial
factorial() {
  if [ $1 -eq 0 ]; then
    echo 1
  else
    local i=$1
    local result=1
    while [ $i -gt 0 ]; do
      result=$((result * i))
```

```
        i=$((i - 1))
      done
      echo $result
  fi
}

# Main script
echo "Enter a number:"
read num

# Check if input is a positive integer
if ! [[ $num =~ ^[0-9]+$ ]]; then
  echo "Error: Input is not a positive integer"
  exit 1
fi

# Calculate and display factorial
fact=$(factorial $num)
echo "Factorial of $num is: $fact"
```

24. Write a shell program to find the sum of n numbers (20)

```
#!/bin/bash

# Prompt the user to enter the value of n
echo "Enter the value of n: "
read n

sum=0
```

```
# Loop to read n numbers and calculate their sum
for ((i = 1; i <= n; i++)); do
   echo "Enter number $i: "
   read num
   sum=$((sum + num))
done

# Print the sum of the n numbers
echo "The sum of $n numbers is: $sum"
```

25. Execute the following commands in UNIX operating system using opendir

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

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

   // Open the current directory
   dir = opendir(".");
   if (dir == NULL) {
      perror("opendir");
      return 1;
   }

   // Read and print directory entries
   while ((entry = readdir(dir)) != NULL) {
      printf("%s\n", entry->d_name);
   }
```

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

    return 0;
}
```

26. . Execute the following commands in UNIX operating system using stat.

```
#include <unistd.h>
#include <fcntl.h> // for open()

int main() {
    // Open a file (example.txt) and get a file descriptor
    int fd = open("example.txt", O_RDONLY);
    if (fd == -1) {
        // Error handling for failed file opening
        perror("open");
        return 1;
    }

    // Use the file descriptor (read/write/etc.)

    // Close the file descriptor
    if (close(fd) == -1) {
        // Error handling for failed closing
        perror("close");
        return 1;
    }

    return 0;
}
```

27. Execute the following commands in UNIX operating system using close.

```c
#include <unistd.h>
#include <fcntl.h> // for open()

int main() {
    // Open a file (example.txt) and get a file descriptor
    int fd = open("example.txt", O_RDONLY);
    if (fd == -1) {
        // Error handling for failed file opening
        perror("open");
        return 1;
    }

    // Use the file descriptor (read/write/etc.)

    // Close the file descriptor
    if (close(fd) == -1) {
        // Error handling for failed closing
        perror("close");
        return 1;
    }

    return 0;
}
```

28. Execute the following commands in UNIX operating system using getpid

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid = getpid(); // Retrieve the PID of the current process
    printf("PID of the current process: %d\n", pid);
```

```
    return 0;
}
```

29. Execute the following commands in UNIX operating system using exit.

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

int main() {
    printf("Before exit()\n");
    exit(EXIT_SUCCESS); // Terminate the program
    printf("After exit()\n"); // This line won't be executed
}
```

30. . Execute the following commands in UNIX operating system using wait.

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

int main() {
    pid_t pid;

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

    if (pid < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Child process
        printf("Child process executing...\n");
        sleep(3); // Simulate some work
```

```
        printf("Child process completed.\n");
        exit(EXIT_SUCCESS);
    } else {
        // Parent process
        printf("Parent process waiting for child...\n");
        wait(NULL); // Wait for any child process to terminate
        printf("Parent process completed.\n");
    }

    return 0;
}
```

31. Execute the following commands in UNIX operating system using readdir

i) readdir

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

int main() {
    // Open the current directory
    DIR *dir = opendir(".");

    // Check if opendir() succeeded
    if (dir == NULL) {
        perror("opendir");
        return 1;
    }

    // Read directory entries
    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) {
        printf("%s\n", entry->d_name);
```

```
        }

        // Check if readdir() encountered an error
        if (errno != 0) {
            perror("readdir");
            closedir(dir);
            return 1;
        }

        // Close the directory
        closedir(dir);

        return 0;
    }
```

32. Execute the following commands in UNIX operating system using fork

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

int main() {
    pid_t pid;

    // Create a new process
    pid = fork();

    if (pid == -1) {
        // Error handling
        perror("fork");
        return 1;
    } else if (pid == 0) {
        // Child process
```

```c
        printf("Child process, PID: %d\n", getpid());
    } else {
        // Parent process
        printf("Parent process, PID: %d, Child PID: %d\n", getpid(), pid);
    }

    return 0;
}
```

33. Explain the following system calls in UNIX operating system using exec

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    char *args[] = {"ls", "-l", NULL};

    // Execute the "ls -l" command
    if (execvp("ls", args) == -1) {
        perror("execvp");
        return 1;
    }

    // This line will not be executed if execvp is successful
    printf("This line will not be printed.\n");

    return 0;
}
```

34. . Write the syntax and execute the following commands using PWD#include <stdio.h>

```c
#include <unistd.h>
#include <stdlib.h>
```

```c
#define MAX_PATH_LENGTH 4096

int main() {
    char cwd[MAX_PATH_LENGTH];

    // Get the current working directory
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        printf("Current working directory: %s\n", cwd);
    } else {
        perror("getcwd");
        return 1;
    }

    return 0;
}
```

35. CD

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

int main() {
    // Change the current working directory to "/home/user"
    if (chdir("/home/user") == 0) {
        printf("Changed directory to /home/user\n");
    } else {
        perror("chdir");
        return 1;
    }

    return 0;
```

```
}
```

36. RMDIR

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

int main() {
    // Attempt to remove the directory "/tmp/example"
    if (rmdir("/tmp/example") == 0) {
        printf("Directory removed successfully.\n");
    } else {
        perror("rmdir");
        return 1;
    }

    return 0;
}
```

37. LS

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

int main() {
    // Open the current directory
    DIR *dir = opendir(".");
    if (dir == NULL) {
        perror("opendir");
        return 1;
    }

    // Read directory entries
```

```
    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) {
        printf("%s\n", entry->d_name);
    }

    // Close the directory
    closedir(dir);

    return 0;
}
```

38. COPY

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

int main() {
    FILE *source_file, *destination_file;
    char source_file_name[100], destination_file_name[100];
    char ch;

    // Get source file name from the user
    printf("Enter source file name: ");
    scanf("%s", source_file_name);

    // Open the source file for reading
    source_file = fopen(source_file_name, "r");
    if (source_file == NULL) {
        perror("Error opening source file");
        return 1;
    }
```

```
    // Get destination file name from the user
    printf("Enter destination file name: ");
    scanf("%s", destination_file_name);

    // Open the destination file for writing
    destination_file = fopen(destination_file_name, "w");
    if (destination_file == NULL) {
        perror("Error opening destination file");
        fclose(source_file);
        return 1;
    }

    // Copy data from source file to destination file character by
character
    while ((ch = fgetc(source_file)) != EOF) {
        fputc(ch, destination_file);
    }

    // Close files
    fclose(source_file);
    fclose(destination_file);

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

39. MOVE

```
#include <stdio.h>

int main() {
    const char *old_filename = "old_file.txt";
```

```
    const char *new_filename = "new_file.txt";

    // Move file
    if (rename(old_filename, new_filename) != 0) {
        perror("Error moving file");
        return 1;
    }

    printf("File moved successfully.\n");

    return 0;
}
```

40. WHO

```
#include <stdlib.h>

int main() {
    // Execute the "who" command
    system("who");

    return 0;
}
```

41.WHO AM I

```
#include <stdlib.h>

int main() {
    // Execute the "whoami" command
    system("whoami");
```

```
    return 0;
}
```

42. MAN

```
#include <stdlib.h>

int main() {
    // Execute the "man" command
    system("man ls"); // Example: Display manual page for the "ls" command

    return 0;
}
```

43. CAT

```
#include <stdlib.h>

int main() {
    // Execute the "cat" command to display the content of files
    system("cat file1.txt file2.txt");

    return 0;
}
```

44. MKDIR

```
#include <stdlib.h>

int main() {
    // Execute the "mkdir" command to create a directory
    system("mkdir directory_name");
```

```c
        return 0;
    }
```

45.CLEAR

```c
#include <stdio.h>

int main() {
    // Print ANSI escape code to clear the screen
    printf("\033[2J\033[H");

    return 0;
}
```

46. explain the concept of looping using shell programming

```c
#include <stdio.h>

int main() {
    // For Loop
    printf("For Loop:\n");
    for (int i = 1; i <= 5; ++i) {
        printf("Number: %d\n", i);
    }

    // While Loop
    printf("\nWhile Loop:\n");
    int count = 1;
    while (count <= 5) {
```

```
        printf("Count: %d\n", count);
        count++;
    }

    // Do-While Loop
    printf("\nDo-While Loop:\n");
    count = 1;
    do {
        printf("Count: %d\n", count);
        count++;
    } while (count <= 5);

    return 0;
}
```