SIMATS ENGINEERING

Engineer to Excel

SIMATS

Saveetha Institute of Medical And Technical Sciences

(Declared as Deemed to be University under Section 3 of UGC Act 1956)

CSA0477-OPERATING SYSTEM FOR SUPERCOMPUTERS

BY

D GOPI

192324290

Question:

1.Write a C program to create a new process using the appropriate system call. Retrieve and display the process identifier (PID) of the currently running process and its parent process.

Aim:

To understand and implement process creation using system calls in C, and to retrieve and display the process IDs of the current process and its parent.

Algorithm:

1. Start.
2. Include the required header files (stdio.h, unistd.h).
3. Call fork() to create a new process:
    - If fork() returns 0, the child process is executing.
    - If fork() returns a positive PID, the parent process is executing.
4. For both parent and child processes:
    - Retrieve the PID of the current process using getpid().
    - Retrieve the PID of the parent process using getppid().
5. Display the PID and PPID for both processes.
6. End.

Program:

```
#include <stdio.h>

#include <unistd.h>


int main() {

   pid_t pid;


   // Create a new process

   pid = fork();


   if (pid < 0) {
```

```
        // Error in process creation
        perror("Fork failed");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("Child Process:\n");
        printf("PID: %d\n", getpid());
        printf("Parent PID: %d\n", getppid());
    } else {
        // Parent process
        printf("Parent Process:\n");
        printf("PID: %d\n", getpid());
        printf("Parent PID: %d\n", getppid());
    }

    return 0;
}
```

Output Example:

Run the program in a terminal:

```
Parent Process:
PID: 12345
Parent PID: 6789

Child Process:
PID: 12346
Parent PID: 12345
```

Question 2:

Identify the system calls to copy the content of one file to another and illustrate the same using a C program.

Aim:

To understand and implement file handling using system calls like open(), read(), write(), and close() to copy the content of one file to another.

Algorithm:

1. Start.
2. Include the necessary headers (stdio.h, fcntl.h, unistd.h).
3. Open the source file in read-only mode using the open() system call.
4. Create or open the destination file in write mode using open() with appropriate permissions.
5. Read the content of the source file using the read() system call.
6. Write the read content to the destination file using the write() system call.
7. Repeat steps 5 and 6 until the entire file is copied.
8. Close both files using the close() system call.
9. End.

Program:

```
#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>


#define BUFFER_SIZE 1024


int main() {

    int srcFile, destFile;

    char buffer[BUFFER_SIZE];

    ssize_t bytesRead, bytesWritten;


    // Open the source file in read-only mode

    srcFile = open("source.txt", O_RDONLY);

    if (srcFile < 0) {

        perror("Error opening source file");

        return 1;
```

```
}

// Open/Create the destination file in write mode
destFile = open("destination.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (destFile < 0) {
    perror("Error opening/creating destination file");
    close(srcFile);
    return 1;
}

// Read from source and write to destination
while ((bytesRead = read(srcFile, buffer, BUFFER_SIZE)) > 0) {
    bytesWritten = write(destFile, buffer, bytesRead);
    if (bytesWritten != bytesRead) {
        perror("Error writing to destination file");
        close(srcFile);
        close(destFile);
        return 1;
    }
}

if (bytesRead < 0) {
    perror("Error reading from source file");
}

// Close both files
close(srcFile);
close(destFile);

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

```
}
```

Question 3:

Design a CPU scheduling program with C using First Come First Served (FCFS) technique with the given considerations.

Aim:

To simulate CPU scheduling using the First Come First Served (FCFS) technique, where all processes are activated at time 0 and no process waits on I/O devices.

Algorithm:

1. Start.
2. Input the number of processes and their burst times.
3. Initialize the waiting time and turn-around time for all processes to 0.
4. Calculate waiting times:
   - For each process i, waiting_time[i] = waiting_time[i-1] + burst_time[i-1].
5. Calculate turn-around times:
   - For each process i, turn_around_time[i] = waiting_time[i] + burst_time[i].
6. Compute the average waiting time and turn-around time.
7. Display the scheduling results (process, burst time, waiting time, turn-around time).
8. End.

Program:

```
#include <stdio.h>


int main() {

    int n, i;

    float avgWaitingTime = 0, avgTurnAroundTime = 0;


    // Input number of processes

    printf("Enter the number of processes: ");

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

```c
int burstTime[n], waitingTime[n], turnAroundTime[n];

// Input burst times
printf("Enter burst times for each process:\n");
for (i = 0; i < n; i++) {
    printf("Process %d: ", i + 1);
    scanf("%d", &burstTime[i]);
}

// Calculate waiting times
waitingTime[0] = 0; // First process has no waiting time
for (i = 1; i < n; i++) {
    waitingTime[i] = waitingTime[i - 1] + burstTime[i - 1];
}

// Calculate turn-around times
for (i = 0; i < n; i++) {
    turnAroundTime[i] = waitingTime[i] + burstTime[i];
}

// Calculate average waiting and turn-around times
for (i = 0; i < n; i++) {
    avgWaitingTime += waitingTime[i];
    avgTurnAroundTime += turnAroundTime[i];
}
avgWaitingTime /= n;
avgTurnAroundTime /= n;

// Display results
printf("\nProcess\tBurst Time\tWaiting Time\tTurn-Around Time\n");
```

```
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", i + 1, burstTime[i], waitingTime[i], turnAroundTime[i]);
    }

    printf("\nAverage Waiting Time: %.2f\n", avgWaitingTime);
    printf("Average Turn-Around Time: %.2f\n", avgTurnAroundTime);

    return 0;
}
```

Example Input/Output:

Input:

Enter the number of processes: 3

Enter burst times for each process:

Process 1: 5

Process 2: 8

Process 3: 12

Output:

```
Process Burst Time   Waiting Time   Turn-Around Time
1      5          0            5
2      8          5            13
3      12         13           25

Average Waiting Time: 6.00
Average Turn-Around Time: 14.33
```

Question 4: Shortest Execution Time (Non-Preemptive SJF)

4: Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next.

Aim: To implement the Shortest Job First (Non-Preemptive SJF) scheduling algorithm in C and calculate the average waiting time and turnaround time.

Algorithm:

1. Input the number of processes and their burst times.

2. Assign process IDs for identification.
3. Sort the processes by burst time in ascending order.
4. Calculate the waiting time for each process:
    - Waiting time for the first process is 0.
    - Waiting time for the rest is the cumulative burst time of all previous processes.
5. Calculate turnaround time for each process as:
    - Turnaround Time = Waiting Time + Burst Time.
6. Compute the average waiting and turnaround times.
7. Display the results in a tabular format.

Code:

```
#include <stdio.h>

int main() {
   int n, i, j, pos, temp;
   float avgWait = 0, avgTurn = 0;

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

   int burst[n], process[n], wait[n], turn[n];

   printf("Enter burst times:\n");
   for (i = 0; i < n; i++) {
      process[i] = i + 1;
      printf("Process %d: ", i + 1);
      scanf("%d", &burst[i]);
   }

   for (i = 0; i < n - 1; i++) {
      pos = i;
      for (j = i + 1; j < n; j++) {
```

```
        if (burst[j] < burst[pos])
            pos = j;
    }
    temp = burst[i];
    burst[i] = burst[pos];
    burst[pos] = temp;

    temp = process[i];
    process[i] = process[pos];
}

wait[0] = 0;
for (i = 1; i < n; i++) {
    wait[i] = wait[i - 1] + burst[i - 1];
}

for (i = 0; i < n; i++) {
    turn[i] = wait[i] + burst[i];
}

for (i = 0; i < n; i++) {
    avgWait += wait[i];
    avgTurn += turn[i];
}

avgWait /= n;
avgTurn /= n;

printf("\nProcess\tBurst\tWait\tTurnaround\n");
for (i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t%d\n", process[i], burst[i], wait[i], turn[i]);
```

```
    }

    printf("\nAverage Wait Time: %.2f\n", avgWait);
    printf("Average Turnaround Time: %.2f\n", avgTurn);

    return 0;
}
```

5: Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.

Aim: To implement the Highest Priority Scheduling algorithm in C and compute the average waiting time and turnaround time.

Algorithm:

1. Input the number of processes, burst times, and their priorities.
2. Assign process IDs for identification.
3. Sort processes by priority in ascending order (higher priority = lower value).
4. Calculate waiting times:
    - Waiting time for the first process is 0.
    - Waiting time for the rest is the cumulative burst time of all previous processes.
5. Calculate turnaround times:
    - Turnaround Time = Waiting Time + Burst Time.
6. Compute the average waiting and turnaround times.
7. Display results in tabular format.

Code:

```
#include <stdio.h>

int main() {
    int n, i, j, pos, temp;
    float avgWait = 0, avgTurn = 0;

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

```
int burst[n], priority[n], process[n], wait[n], turn[n];

printf("Enter burst times and priorities:\n");
for (i = 0; i < n; i++) {
    process[i] = i + 1;
    printf("Process %d - Burst Time: ", i + 1);
    scanf("%d", &burst[i]);
    printf("Process %d - Priority: ", i + 1);
    scanf("%d", &priority[i]);
}

for (i = 0; i < n - 1; i++) {
    pos = i;
    for (j = i + 1; j < n; j++) {
        if (priority[j] < priority[pos])
            pos = j;
    }
    temp = burst[i];
    burst[i] = burst[pos];
    burst[pos] = temp;

    temp = priority[i];
    priority[i] = priority[pos];
    priority[pos] = temp;

    temp = process[i];
    process[i] = process[pos];
}

wait[0] = 0;
```

```c
    for (i = 1; i < n; i++) {
        wait[i] = wait[i - 1] + burst[i - 1];
    }

    for (i = 0; i < n; i++) {
        turn[i] = wait[i] + burst[i];
    }

    for (i = 0; i < n; i++) {
        avgWait += wait[i];
        avgTurn += turn[i];
    }

    avgWait /= n;
    avgTurn /= n;

    printf("\nProcess\tBurst\tPriority\tWait\tTurnaround\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t\t%d\t%d\n", process[i], burst[i], priority[i], wait[i], turn[i]);
    }

    printf("\nAverage Wait Time: %.2f\n", avgWait);
    printf("Average Turnaround Time: %.2f\n", avgTurn);

    return 0;
}
```

Question 6: Pre-Emptive Priority Scheduling

Aim: To design a C program to implement the preemptive priority scheduling algorithm.

Algorithm:

1. Input the number of processes, burst times, and priorities.
2. Initialize waiting and turnaround times to 0 for all processes.

3. At each unit of time, select the process with the highest priority (smallest priority value) from the ready queue.
4. Decrease the remaining burst time of the selected process.
5. If a process completes, record its turnaround and waiting times.
6. Calculate and display average waiting and turnaround times.

Code:

```
#include <stdio.h>

int main() {
    int n, i, j, t = 0, completed = 0, smallest;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    int burst[n], remain[n], priority[n], process[n], wait[n], turn[n], isCompleted[n];
    float avgWait = 0, avgTurn = 0;

    for (i = 0; i < n; i++) {
        printf("Process %d - Burst Time: ", i + 1);
        scanf("%d", &burst[i]);
        printf("Process %d - Priority: ", i + 1);
        scanf("%d", &priority[i]);
        process[i] = i + 1;
        remain[i] = burst[i];
        isCompleted[i] = 0;
    }

    while (completed != n) {
        smallest = -1;
        for (i = 0; i < n; i++) {
            if (remain[i] > 0 && (smallest == -1 || priority[i] < priority[smallest])) {
                smallest = i;
```

```c
        }
    }
    if (smallest != -1) {
        remain[smallest]--;
        if (remain[smallest] == 0) {
            completed++;
            turn[smallest] = t + 1;
            wait[smallest] = turn[smallest] - burst[smallest];
            isCompleted[smallest] = 1;
        }
    }
    t++;
}

for (i = 0; i < n; i++) {
    avgWait += wait[i];
    avgTurn += turn[i];
}

avgWait /= n;
avgTurn /= n;

printf("\nProcess\tBurst\tPriority\tWait\tTurnaround\n");
for (i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t\t%d\t%d\n", process[i], burst[i], priority[i], wait[i], turn[i]);
}

printf("\nAverage Wait Time: %.2f\n", avgWait);
printf("Average Turnaround Time: %.2f\n", avgTurn);

return 0;
```

}

Question 7: Non-Preemptive Shortest Job First (SJF) Scheduling

Aim: To design a C program to implement the Non-Preemptive Shortest Job First (SJF) scheduling algorithm and calculate the average waiting time and turnaround time.

Algorithm:

1. Input the number of processes and their burst times.
2. Assign process IDs for identification.
3. Sort the processes based on burst time in ascending order.
4. Calculate the waiting time for each process:
   - Waiting time for the first process is 0.
   - For subsequent processes, waiting time is the cumulative burst time of all previous processes.
5. Calculate turnaround time for each process:
   - Turnaround Time = Waiting Time + Burst Time.
6. Compute the average waiting time and turnaround time.
7. Display the process details in a tabular format along with average values.

Code:

```
#include <stdio.h>


int main() {
  int n, i, j, temp, pos;
  float avgWait = 0, avgTurn = 0;


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


  int burst[n], process[n], wait[n], turn[n];


  printf("Enter burst times:\n");
  for (i = 0; i < n; i++) {
    process[i] = i + 1;
```

```
    printf("Process %d: ", i + 1);
    scanf("%d", &burst[i]);
}

for (i = 0; i < n - 1; i++) {
    pos = i;
    for (j = i + 1; j < n; j++) {
        if (burst[j] < burst[pos])
            pos = j;
    }
    temp = burst[i];
    burst[i] = burst[pos];
    burst[pos] = temp;

    temp = process[i];
    process[i] = process[pos];
    process[pos] = temp;
}

wait[0] = 0;
for (i = 1; i < n; i++) {
    wait[i] = wait[i - 1] + burst[i - 1];
}

for (i = 0; i < n; i++) {
    turn[i] = wait[i] + burst[i];
}

for (i = 0; i < n; i++) {
    avgWait += wait[i];
    avgTurn += turn[i];
```

```
    }

    avgWait /= n;
    avgTurn /= n;

    printf("\nProcess\tBurst\tWait\tTurnaround\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\n", process[i], burst[i], wait[i], turn[i]);
    }

    printf("\nAverage Wait Time: %.2f\n", avgWait);
    printf("Average Turnaround Time: %.2f\n", avgTurn);

    return 0;
}
```

Question 8: Round Robin Scheduling

Aim: To design a C program to simulate the Round Robin scheduling algorithm.

Algorithm:

1. Input the number of processes, burst times, and the time quantum.
2. Initialize remaining times for all processes to their burst times.
3. Iterate over processes in a circular manner:
    - Execute each process for a time quantum or until completion.
    - Reduce the remaining time for the process.
    - Record waiting and turnaround times upon completion.
4. Repeat until all processes are completed.
5. Calculate and display average waiting and turnaround times.

Code:

```
#include <stdio.h>

int main() {
    int n, quantum, i, completed = 0, t = 0;
```

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

int burst[n], remain[n], wait[n], turn[n], process[n];
float avgWait = 0, avgTurn = 0;

printf("Enter the time quantum: ");
scanf("%d", &quantum);

for (i = 0; i < n; i++) {
    printf("Process %d - Burst Time: ", i + 1);
    scanf("%d", &burst[i]);
    remain[i] = burst[i];
    process[i] = i + 1;
}

while (completed != n) {
    for (i = 0; i < n; i++) {
        if (remain[i] > 0) {
            if (remain[i] <= quantum) {
                t += remain[i];
                remain[i] = 0;
                completed++;
                turn[i] = t;
                wait[i] = turn[i] - burst[i];
            } else {
                t += quantum;
                remain[i] -= quantum;
            }
        }
    }
```

```c
    }

    for (i = 0; i < n; i++) {
        avgWait += wait[i];
        avgTurn += turn[i];
    }

    avgWait /= n;
    avgTurn /= n;

    printf("\nProcess\tBurst\tWait\tTurnaround\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\n", process[i], burst[i], wait[i], turn[i]);
    }

    printf("\nAverage Wait Time: %.2f\n", avgWait);
    printf("Average Turnaround Time: %.2f\n", avgTurn);

    return 0;
}
```

Question 9: Inter-Process Communication Using Shared Memory

Aim: To illustrate inter-process communication (IPC) using shared memory in C.

Algorithm:

1. Create a shared memory segment using the shmget system call.
2. Attach the shared memory segment to the process's address space using shmat.
3. For the writer process:
    - Write data to the shared memory.
4. For the reader process:
    - Read data from the shared memory.
5. Detach the shared memory using shmdt.
6. Remove the shared memory segment using shmctl after communication is completed.

Code:

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>

int main() {
   key_t key = 1234;
   int shmid;
   char *shared_memory;
   shmid = shmget(key, 1024, 0666 | IPC_CREAT);
   shared_memory = (char *)shmat(shmid, NULL, 0);

   if (fork() == 0) {  // Child process (Writer)
      printf("Writer: Enter a message: ");
      fgets(shared_memory, 1024, stdin);
      shmdt(shared_memory);  // Detaching shared memory
   } else {  // Parent process (Reader)
      sleep(1);  // Ensure child writes first
      printf("Reader: Message from shared memory: %s", shared_memory);
      shmdt(shared_memory);  // Detaching shared memory
      shmctl(shmid, IPC_RMID, NULL);  // Removing shared memory
   }

   return 0;
}
```

Question 10: Inter-Process Communication Using Message Queue

Aim: To illustrate inter-process communication (IPC) using message queues in C.

Algorithm:

1. Create a message queue using msgget.
2. Define a message structure with fields for type and data.
3. Use msgsnd to send messages to the queue.
4. Use msgrcv to receive messages from the queue.
5. Remove the message queue using msgctl after communication is completed.

Code:

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <unistd.h>

struct message {
   long msg_type;
   char msg_text[100];
};

int main() {
   key_t key = 1234;
   int msgid;

   // Creating message queue
   msgid = msgget(key, 0666 | IPC_CREAT);

   if (fork() == 0) {  // Child process (Sender)
      struct message msg;
      msg.msg_type = 1;
      printf("Sender: Enter a message: ");
      fgets(msg.msg_text, sizeof(msg.msg_text), stdin);
```

```
        msgsnd(msgid, &msg, sizeof(msg), 0);
    } else {  // Parent process (Receiver)
        sleep(1);  // Ensure child sends first
        struct message msg;
        msgrcv(msgid, &msg, sizeof(msg), 1, 0);
        printf("Receiver: Message received: %s", msg.msg_text);
        msgctl(msgid, IPC_RMID, NULL);  // Removing message queue
    }

    return 0;
}
```

Here are the detailed solutions for the requested tasks:

Question 11: Illustrate the Concept of Multithreading

Aim: To implement a multithreading program in C.

Algorithm:

1. Include the necessary libraries for thread handling.
2. Define a function to be executed by threads.
3. Create multiple threads using pthread_create.
4. Execute the defined function for each thread.
5. Wait for all threads to complete using pthread_join.

Code:

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

void *print_message(void *thread_id) {
    int id = *(int *)thread_id;
    printf("Thread %d is executing.\n", id);
    pthread_exit(NULL);
}
```

```
int main() {
    pthread_t threads[3];
    int thread_ids[3];

    for (int i = 0; i < 3; i++) {
        thread_ids[i] = i + 1;
        pthread_create(&threads[i], NULL, print_message, (void *)&thread_ids[i]);
    }

    for (int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("All threads completed.\n");
    return 0;
}
```

Question 12: Simulate the Dining-Philosophers Problem

Aim: To simulate the Dining-Philosophers problem using multithreading and semaphores in C.

Algorithm:

1. Initialize semaphores for forks and a mutex for resource access.
2. Create threads representing philosophers.
3. Define actions: Thinking, Picking up forks, Eating, and Putting down forks.
4. Use semaphores to ensure no two neighboring philosophers eat simultaneously.

Code:

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

#define N 5
```

```c
sem_t forks[N];
pthread_t philosophers[N];

void *dine(void *arg) {
    int id = *(int *)arg;

    printf("Philosopher %d is thinking.\n", id);
    sleep(1);

    sem_wait(&forks[id]);
    sem_wait(&forks[(id + 1) % N]);

    printf("Philosopher %d is eating.\n", id);
    sleep(1);

    sem_post(&forks[id]);
    sem_post(&forks[(id + 1) % N]);

    printf("Philosopher %d has finished eating.\n", id);
    return NULL;
}

int main() {
    for (int i = 0; i < N; i++) {
        sem_init(&forks[i], 0, 1);
    }

    int ids[N];
    for (int i = 0; i < N; i++) {
        ids[i] = i;
```

```
        pthread_create(&philosophers[i], NULL, dine, &ids[i]);
    }

    for (int i = 0; i < N; i++) {
        pthread_join(philosophers[i], NULL);
    }

    printf("Dining completed.\n");
    return 0;
}
```

Question 13: Implement Memory Allocation Strategies

Aim: To demonstrate First Fit, Best Fit, and Worst Fit memory allocation strategies in C.

Code:

```
#include <stdio.h>

void first_fit(int block[], int m, int process[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) allocation[i] = -1;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (block[j] >= process[i]) {
                allocation[i] = j;
                block[j] -= process[i];
                break;
            }
        }
    }

    printf("Process\tSize\tBlock\n");
```

```
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%s\n", i + 1, process[i], allocation[i] != -1 ? "Allocated" : "Not Allocated");
    }
}

int main() {
    int block[] = {100, 500, 200, 300, 600};
    int process[] = {212, 417, 112, 426};
    int m = sizeof(block) / sizeof(block[0]);
    int n = sizeof(process) / sizeof(process[0]);

    printf("First Fit Memory Allocation:\n");
    first_fit(block, m, process, n);

    return 0;
}
```

Question 14: Organize Files Using Single-Level Directory

Aim: To simulate a single-level directory structure in C.

Code:

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

#define MAX_FILES 100

char directory[MAX_FILES][50];
int file_count = 0;

void create_file(char *filename) {
    strcpy(directory[file_count], filename);
    file_count++;
```

```
}

void list_files() {
    printf("Files in the directory:\n");
    for (int i = 0; i < file_count; i++) {
        printf("%s\n", directory[i]);
    }
}

int main() {
    create_file("file1.txt");
    create_file("file2.txt");
    create_file("file3.txt");

    list_files();
    return 0;
}
```

Question 15: Organize Files Using Two-Level Directory

Aim: To simulate a two-level directory structure in C.

Code:

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

#define MAX_DIRS 5
#define MAX_FILES 5

typedef struct {
    char name[50];
    char files[MAX_FILES][50];
    int file_count;
```

```
} Directory;

Directory directories[MAX_DIRS];
int dir_count = 0;

void create_directory(char *dirname) {
    strcpy(directories[dir_count].name, dirname);
    directories[dir_count].file_count = 0;
    dir_count++;
}

void create_file(char *dirname, char *filename) {
    for (int i = 0; i < dir_count; i++) {
        if (strcmp(directories[i].name, dirname) == 0) {
            strcpy(directories[i].files[directories[i].file_count], filename);
            directories[i].file_count++;
            return;
        }
    }
    printf("Directory not found.\n");
}

void list_files(char *dirname) {
    for (int i = 0; i < dir_count; i++) {
        if (strcmp(directories[i].name, dirname) == 0) {
            printf("Files in directory %s:\n", dirname);
            for (int j = 0; j < directories[i].file_count; j++) {
                printf("%s\n", directories[i].files[j]);
            }
            return;
        }
```

```
    }
    printf("Directory not found.\n");
}

int main() {
    create_directory("dir1");
    create_directory("dir2");

    create_file("dir1", "file1.txt");
    create_file("dir1", "file2.txt");
    create_file("dir2", "file3.txt");

    list_files("dir1");
    list_files("dir2");

    return 0;
}
```

Here are the solutions to the requested questions along with their respective algorithms and C program codes without comments:

Question 16: Random Access File for Employee Details

Aim: To implement random access file processing for employee details using C.

Algorithm:

1. Create a structure for employee details (e.g., name, ID, salary).
2. Open the file in read/write binary mode.
3. Use fseek to access a particular record based on the position.
4. Read and write employee data using fread and fwrite.
5. Close the file after all operations.

Code:

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

```
struct employee {
    int id;
    char name[50];
    float salary;
};

int main() {
    FILE *file;
    struct employee emp;

    file = fopen("employee.dat", "wb+");

    emp.id = 1;
    strcpy(emp.name, "John Doe");
    emp.salary = 50000;
    fwrite(&emp, sizeof(emp), 1, file);

    fseek(file, 0, SEEK_SET);
    fread(&emp, sizeof(emp), 1, file);
    printf("Employee ID: %d\nName: %s\nSalary: %.2f\n", emp.id, emp.name, emp.salary);

    fclose(file);
    return 0;
}
```

Question 17: Banker's Algorithm for Deadlock Avoidance

Aim: To implement the Banker's algorithm for deadlock avoidance in C.

Algorithm:

1. Initialize available resources and maximum demands.
2. Check if a process's request can be granted safely by checking available resources.

3. If yes, allocate resources and check for a safe sequence.
4. If no, reject the request and proceed with the next process.

Code:

```
#include <stdio.h>

#define P 5
#define R 3

int max[P][R], allot[P][R], need[P][R], available[R];

int isSafe() {
   int work[R], finish[P], safeSeq[P], count = 0;

   for (int i = 0; i < R; i++) {
      work[i] = available[i];
   }

   for (int i = 0; i < P; i++) {
      finish[i] = 0;
   }

   while (count < P) {
      int found = 0;
      for (int p = 0; p < P; p++) {
         if (!finish[p]) {
            int possible = 1;
            for (int r = 0; r < R; r++) {
               if (need[p][r] > work[r]) {
                  possible = 0;
                  break;
               }
```

```
            }
            if (possible) {
                for (int r = 0; r < R; r++) {
                    work[r] += allot[p][r];
                }
                finish[p] = 1;
                safeSeq[count++] = p;
                found = 1;
                break;
            }
        }
    }
    if (!found) {
        return 0;
    }
  }
  return 1;
}

int main() {
  int i, j;

  printf("Enter available resources: ");
  for (i = 0; i < R; i++) {
    scanf("%d", &available[i]);
  }

  printf("Enter the maximum demand matrix:\n");
  for (i = 0; i < P; i++) {
    for (j = 0; j < R; j++) {
        scanf("%d", &max[i][j]);
```

```
        }
    }

    printf("Enter the allocated resources matrix:\n");
    for (i = 0; i < P; i++) {
        for (j = 0; j < R; j++) {
            scanf("%d", &allot[i][j]);
        }
    }

    for (i = 0; i < P; i++) {
        for (j = 0; j < R; j++) {
            need[i][j] = max[i][j] - allot[i][j];
        }
    }

    if (isSafe()) {
        printf("System is in a safe state.\n");
    } else {
        printf("System is in an unsafe state.\n");
    }

    return 0;
}
```

Question 18: Producer-Consumer Problem Using Semaphores

Aim: To implement the producer-consumer problem using semaphores in C.

Algorithm:

1. Initialize two semaphores: one for full and one for empty.
2. The producer will produce items and signal the full semaphore.
3. The consumer will consume items and signal the empty semaphore.

4. Use a buffer to store items.
5. Ensure synchronization using semaphores to prevent race conditions.

Code:

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

#define MAX 5
int buffer[MAX];
int in = 0, out = 0;

sem_t empty, full, mutex;

void *producer(void *param) {
   int item;
   while (1) {
      item = rand() % 100;
      sem_wait(&empty);
      sem_wait(&mutex);

      buffer[in] = item;
      printf("Produced: %d\n", item);
      in = (in + 1) % MAX;

      sem_post(&mutex);
      sem_post(&full);
   }
}

void *consumer(void *param) {
   int item;
```

```c
    while (1) {
        sem_wait(&full);
        sem_wait(&mutex);

        item = buffer[out];
        printf("Consumed: %d\n", item);
        out = (out + 1) % MAX;

        sem_post(&mutex);
        sem_post(&empty);
    }
}

int main() {
    pthread_t prod, cons;

    sem_init(&empty, 0, MAX);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
```

```
}
```

Question 19: Process Synchronization Using Mutex Locks

Aim: To implement process synchronization using mutex locks in C.

Algorithm:

1. Initialize a mutex.
2. Use pthread_mutex_lock and pthread_mutex_unlock to control access to shared resources.
3. Ensure that only one thread accesses the shared resource at a time.

Code:

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

int counter = 0;
pthread_mutex_t lock;

void *increment(void *param) {
  pthread_mutex_lock(&lock);
  counter++;
  printf("Counter: %d\n", counter);
  pthread_mutex_unlock(&lock);
  return NULL;
}

int main() {
  pthread_t threads[5];
  pthread_mutex_init(&lock, NULL);

  for (int i = 0; i < 5; i++) {
    pthread_create(&threads[i], NULL, increment, NULL);
  }
```

```
    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&lock);
    return 0;
}
```

Question 20: Reader-Writer Problem Using Semaphores

Aim: To implement the Reader-Writer problem using semaphores in C.

Algorithm:

1. Use semaphores to control access to shared resource.
2. Readers can read concurrently, but writers need exclusive access.
3. Writers wait if a reader is reading, and readers wait if a writer is writing.

Code:

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

sem_t mutex, writeBlock;
int readCount = 0;

void *reader(void *param) {
    sem_wait(&mutex);
    readCount++;
    if (readCount == 1) {
        sem_wait(&writeBlock);
    }
    sem_post(&mutex);

    printf("Reading...\n");
```

```
    sem_wait(&mutex);
    readCount--;
    if (readCount == 0) {
        sem_post(&writeBlock);
    }
    sem_post(&mutex);

    return NULL;
}

void *writer(void *param) {
    sem_wait(&writeBlock);
    printf("Writing...\n");
    sem_post(&writeBlock);
    return NULL;
}

int main() {
    pthread_t r[5], w[5];

    sem_init(&mutex, 0, 1);
    sem_init(&writeBlock, 0, 1);

    for (int i = 0; i < 5; i++) {
        pthread_create(&r[i], NULL, reader, NULL);
        pthread_create(&w[i], NULL, writer, NULL);
    }

    for (int i = 0; i < 5; i++) {
        pthread_join(r[i], NULL);
```

```c
        pthread_join(w[i], NULL);
    }

    sem_destroy(&mutex);
    sem_destroy(&writeBlock);
    return 0;
}
```

Here are the solutions for the requested questions along with their respective algorithms and C code:

Question 21: Worst Fit Algorithm of Memory Management

Aim: To implement the worst-fit memory management algorithm in C.

Algorithm:

1. Given a list of memory blocks and processes, find the largest memory block.
2. Allocate the largest block that is capable of satisfying the process.
3. If no block is large enough, the process cannot be allocated.

Code:

```c
#include <stdio.h>

void worst_fit(int block[], int m, int process[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) allocation[i] = -1;

    for (int i = 0; i < n; i++) {
        int max_idx = -1;
        for (int j = 0; j < m; j++) {
            if (block[j] >= process[i]) {
                if (max_idx == -1 || block[j] > block[max_idx]) {
                    max_idx = j;
                }
```

```
            }
        }
        if (max_idx != -1) {
            allocation[i] = max_idx;
            block[max_idx] -= process[i];
        }
    }

    printf("Process\tSize\tBlock\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%s\n", i + 1, process[i], allocation[i] != -1 ? "Allocated" : "Not Allocated");
    }
}

int main() {
    int block[] = {100, 500, 200, 300, 600};
    int process[] = {212, 417, 112, 426};
    int m = sizeof(block) / sizeof(block[0]);
    int n = sizeof(process) / sizeof(process[0]);

    printf("Worst Fit Memory Allocation:\n");
    worst_fit(block, m, process, n);

    return 0;
}
```

Question 22: Best Fit Algorithm of Memory Management

Aim: To implement the best-fit memory management algorithm in C.

Algorithm:

1. Given a list of memory blocks and processes, find the smallest block that can fit the process.
2. Allocate the block to the process.

3. If no block is small enough, the process cannot be allocated.

Code:

```
#include <stdio.h>

void best_fit(int block[], int m, int process[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) allocation[i] = -1;

    for (int i = 0; i < n; i++) {
        int min_idx = -1;
        for (int j = 0; j < m; j++) {
            if (block[j] >= process[i]) {
                if (min_idx == -1 || block[j] < block[min_idx]) {
                    min_idx = j;
                }
            }
        }
        if (min_idx != -1) {
            allocation[i] = min_idx;
            block[min_idx] -= process[i];
        }
    }

    printf("Process\tSize\tBlock\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%s\n", i + 1, process[i], allocation[i] != -1 ? "Allocated" : "Not Allocated");
    }
}

int main() {
    int block[] = {100, 500, 200, 300, 600};
```

```
    int process[] = {212, 417, 112, 426};
    int m = sizeof(block) / sizeof(block[0]);
    int n = sizeof(process) / sizeof(process[0]);

    printf("Best Fit Memory Allocation:\n");
    best_fit(block, m, process, n);

    return 0;
}
```

Question 23: First Fit Algorithm of Memory Management

Aim: To implement the first-fit memory management algorithm in C.

Algorithm:

1. Given a list of memory blocks and processes, traverse the memory blocks in order.
2. Allocate the first block that is large enough to accommodate the process.
3. If no block is large enough, the process cannot be allocated.

Code:

```
#include <stdio.h>

void first_fit(int block[], int m, int process[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++) allocation[i] = -1;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (block[j] >= process[i]) {
                allocation[i] = j;
                block[j] -= process[i];
                break;
            }
        }
```

```
    }

    printf("Process\tSize\tBlock\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%s\n", i + 1, process[i], allocation[i] != -1 ? "Allocated" : "Not Allocated");
    }
}

int main() {
    int block[] = {100, 500, 200, 300, 600};
    int process[] = {212, 417, 112, 426};
    int m = sizeof(block) / sizeof(block[0]);
    int n = sizeof(process) / sizeof(process[0]);

    printf("First Fit Memory Allocation:\n");
    first_fit(block, m, process, n);

    return 0;
}
```

Question 24: UNIX System Calls for File Management

Aim: To demonstrate UNIX system calls for file management in C.

Algorithm:

1. Use open() to open a file.
2. Use read() and write() to manipulate the file contents.
3. Use close() to close the file.
4. Use lseek() to move the file pointer.

Code:

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

```
int main() {
    int fd = open("file.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    if (fd == -1) {
        perror("Error opening file");
        return 1;
    }

    char buf[] = "Hello, UNIX!";
    write(fd, buf, sizeof(buf));

    lseek(fd, 0, SEEK_SET);
    char read_buf[20];
    read(fd, read_buf, sizeof(buf));

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

    close(fd);
    return 0;
}
```

Question 25: I/O System Calls of UNIX

Aim: To implement the I/O system calls in UNIX (fcntl, seek, stat, opendir, readdir) using C.

Algorithm:

1. Use fcntl() to manipulate file descriptors.
2. Use lseek() for seeking specific byte positions in a file.
3. Use stat() to get file information.
4. Use opendir() and readdir() to read the contents of a directory.

Code:

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

```c
#include <unistd.h>
#include <sys/stat.h>
#include <dirent.h>

int main() {
    int fd = open("file.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    if (fd == -1) {
        perror("Error opening file");
        return 1;
    }

    struct stat statbuf;
    stat("file.txt", &statbuf);
    printf("File size: %ld bytes\n", statbuf.st_size);

    lseek(fd, 0, SEEK_SET);
    char buf[] = "Hello, UNIX!";
    write(fd, buf, sizeof(buf));

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

    close(fd);
    return 0;
}
```

Question 26: File Management Operations

Aim: To implement file management operations in C.

Algorithm:

1. Use fopen() to open a file.
2. Use fprintf() to write data.
3. Use fscanf() to read data.
4. Use fclose() to close the file.

Code:

```
#include <stdio.h>

int main() {
    FILE *file = fopen("file.txt", "w+");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    fprintf(file, "This is a file management program.\n");

    fclose(file);

    file = fopen("file.txt", "r");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    char line[100];
    while (fgets(line, sizeof(line), file)) {
        printf("%s", line);
```

```
    }

    fclose(file);
    return 0;
}
```

Question 27: Simulating ls UNIX Command

Aim: To simulate the functionality of the ls UNIX command.

Algorithm:

1. Use opendir() to open a directory.
2. Use readdir() to read directory entries.
3. Display each entry.

Code:

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

int main() {
    DIR *dir = opendir(".");
    if (dir == NULL) {
        perror("Unable to open directory");
        return 1;
    }

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

    closedir(dir);
    return 0;
}
```

Question 28: Simulating grep UNIX Command

Aim: To simulate the functionality of the grep UNIX command in C, which searches for a specific string within a file and displays the matching lines.

Algorithm:

1. Open the file using fopen().
2. Read the file line by line using fgets().
3. Use strstr() to search for the specified string in each line.
4. If the string is found, print the line.
5. Close the file after processing.

Code:

```
#include <stdio.h>

#include <string.h>


int main() {

   // Open the file in read mode

   FILE *file = fopen("file.txt", "r");

   if (file == NULL) {

      perror("Error opening file");

      return 1;

   }


   char line[100];

   char search_str[] = "search";  // The string to search for


   // Read the file line by line

   while (fgets(line, sizeof(line), file)) {

      // If the search string is found in the line, print it

      if (strstr(line, search_str) != NULL) {
```

```
            printf("%s", line);
        }
    }

    // Close the file after processing
    fclose(file);
    return 0;
}
```

Here are the solutions to the requested questions:

Question 29: Simulating the Solution of Classical Process Synchronization Problem (e.g., Producer-Consumer Problem)

Aim: To simulate the classical Producer-Consumer problem using semaphores.

Algorithm:

1. Create two semaphores: empty (for empty slots in the buffer) and full (for full slots in the buffer).
2. Producer will add an item to the buffer if there is space (i.e., empty is not 0).
3. Consumer will consume an item if the buffer has items (i.e., full is not 0).
4. Use mutual exclusion to avoid race conditions.

Code:

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

#define MAX 5
int buffer[MAX];
int in = 0, out = 0;

sem_t empty, full;
pthread_mutex_t mutex;
```

```c
void *producer(void *arg) {
    int item;
    while (1) {
        item = rand() % 100;  // Produce an item
        sem_wait(&empty);  // Wait if no empty slot
        pthread_mutex_lock(&mutex);

        buffer[in] = item;
        in = (in + 1) % MAX;
        printf("Produced: %d\n", item);

        pthread_mutex_unlock(&mutex);
        sem_post(&full);  // Signal that a full slot is available
    }
}

void *consumer(void *arg) {
    int item;
    while (1) {
        sem_wait(&full);  // Wait if no full slot
        pthread_mutex_lock(&mutex);

        item = buffer[out];
        out = (out + 1) % MAX;
        printf("Consumed: %d\n", item);

        pthread_mutex_unlock(&mutex);
        sem_post(&empty);  // Signal that an empty slot is available
    }
}
```

```c
int main() {
    pthread_t prod, cons;

    sem_init(&empty, 0, MAX);  // Initialize the empty semaphore
    sem_init(&full, 0, 0);     // Initialize the full semaphore
    pthread_mutex_init(&mutex, NULL);  // Initialize the mutex lock

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

Question 30: Thread-Related Concepts

Aim: To demonstrate thread operations like create, join, equal, and exit in C.

Algorithm:

1. Create a thread using pthread_create().
2. Use pthread_join() to wait for the thread to finish.
3. Use pthread_equal() to check if two threads are the same.
4. Use pthread_exit() to terminate a thread.

Code:

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

```c
void *thread_function(void *arg) {
    printf("Thread is running\n");
    pthread_exit(NULL);  // Exit the thread
}

int main() {
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, thread_function, NULL);
    pthread_create(&thread2, NULL, thread_function, NULL);

    pthread_join(thread1, NULL);  // Wait for thread1 to finish
    pthread_join(thread2, NULL);  // Wait for thread2 to finish

    if (pthread_equal(thread1, thread2)) {
        printf("Threads are equal\n");
    } else {
        printf("Threads are different\n");
    }

    return 0;
}
```

Question 31: FIFO Paging Technique of Memory Management

Aim: To simulate the FIFO (First In, First Out) paging technique in memory management.

Algorithm:

1. Keep track of pages in memory.
2. When a page is requested, if it is not in memory, replace the page that has been in memory the longest.

Code:

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

```
#define MAX_FRAMES 3
int frames[MAX_FRAMES];

void fifo_paging(int pages[], int n) {
    int page_faults = 0;
    int front = 0;

    for (int i = 0; i < n; i++) {
        int page = pages[i];
        int found = 0;

        // Check if page is already in memory
        for (int j = 0; j < MAX_FRAMES; j++) {
            if (frames[j] == page) {
                found = 1;
                break;
            }
        }

        if (!found) {
            frames[front] = page;
            front = (front + 1) % MAX_FRAMES;
            page_faults++;
            printf("Page %d loaded into memory\n", page);
        }
    }
    printf("Total Page Faults: %d\n", page_faults);
}

int main() {
```

```
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2};
    int n = sizeof(pages) / sizeof(pages[0]);

    printf("FIFO Paging Simulation:\n");
    fifo_paging(pages, n);

    return 0;
}
```

Question 32: Least Recently Used (LRU) Paging Technique

Aim: To simulate the Least Recently Used (LRU) paging technique in memory management.

Algorithm:

1. Keep track of pages in memory and their recent usage.
2. When a new page is referenced, if it is not in memory, replace the least recently used page.

Code:

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

#define MAX_FRAMES 3
int frames[MAX_FRAMES];

void lru_paging(int pages[], int n) {
    int page_faults = 0;

    for (int i = 0; i < n; i++) {
        int page = pages[i];
        int found = 0;

        // Check if page is already in memory
        for (int j = 0; j < MAX_FRAMES; j++) {
            if (frames[j] == page) {
```

```
            found = 1;
            break;
        }
    }

    if (!found) {
        // Replace the least recently used page
        frames[i % MAX_FRAMES] = page;
        page_faults++;
        printf("Page %d loaded into memory\n", page);
    }
  }

  printf("Total Page Faults: %d\n", page_faults);
}

int main() {
  int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2};
  int n = sizeof(pages) / sizeof(pages[0]);

  printf("LRU Paging Simulation:\n");
  lru_paging(pages, n);

  return 0;
}
```

Question 33: Optimal Paging Technique

Aim: To simulate the Optimal paging technique in memory management.

Algorithm:

1. Keep track of pages in memory.
2. Replace the page that will not be used for the longest period in the future.

Code:

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

#define MAX_FRAMES 3
int frames[MAX_FRAMES];

int optimal_paging(int pages[], int n) {
  int page_faults = 0;

  for (int i = 0; i < n; i++) {
    int page = pages[i];
    int found = 0;

    // Check if page is already in memory
    for (int j = 0; j < MAX_FRAMES; j++) {
      if (frames[j] == page) {
        found = 1;
        break;
      }
    }

    if (!found) {
      int farthest = -1, replace = -1;
      for (int j = 0; j < MAX_FRAMES; j++) {
        int k;
        for (k = i + 1; k < n; k++) {
          if (frames[j] == pages[k]) {
            if (k > farthest) {
              farthest = k;
              replace = j;
```

```
                }
                break;
            }
        }
        if (k == n) {
            replace = j;
            break;
        }
    }
    frames[replace] = page;
    page_faults++;
    printf("Page %d loaded into memory\n", page);
  }
 }
 return page_faults;
}

int main() {
 int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2};
 int n = sizeof(pages) / sizeof(pages[0]);

 printf("Optimal Paging Simulation:\n");
 int page_faults = optimal_paging(pages, n);
 printf("Total Page Faults: %d\n", page_faults);

 return 0;
}
```

Here are the solutions to questions 34 to 40:

Question 34: File Allocation Strategy - Sequential Allocation (Contiguous Allocation)

Aim: To simulate the file allocation strategy where records are stored contiguously in a file, and each record can only be accessed by reading all the previous records.

Algorithm:

1. Create a file with multiple records stored sequentially.
2. Read records one by one in the order they are stored.
3. Accessing a record requires reading all the previous records.

Code:

```
#include <stdio.h>


void sequential_allocation() {
   FILE *file = fopen("file.txt", "w");
   if (file == NULL) {
      printf("Error opening file!\n");
      return;
   }


   // Writing records to file
   for (int i = 1; i <= 5; i++) {
      fprintf(file, "Record %d\n", i);
   }
   fclose(file);


   // Reading records sequentially
   file = fopen("file.txt", "r");
   if (file == NULL) {
      printf("Error opening file!\n");
      return;
   }
   char buffer[100];
   while (fgets(buffer, sizeof(buffer), file)) {
      printf("%s", buffer);  // Accessing records sequentially
```

```
    }
    fclose(file);
}

int main() {
    sequential_allocation();
    return 0;
}
```

Question 35: File Allocation Strategy - Indexed Allocation

Aim: To simulate a file allocation strategy where all file pointers are brought together into an index block, and each entry points to the corresponding block of the file.

Algorithm:

1. Create an index block that holds pointers to actual data blocks.
2. Each file block is accessed through the index block.

Code:

```
#include <stdio.h>

#define NUM_BLOCKS 3
#define BLOCK_SIZE 10

void indexed_allocation() {
    FILE *file = fopen("file.txt", "w");
    if (file == NULL) {
        printf("Error opening file!\n");
        return;
    }

    // Writing records to blocks
    for (int i = 1; i <= NUM_BLOCKS; i++) {
        fprintf(file, "Block %d: Data stored in block %d\n", i, i);
```

```
    }
    fclose(file);

    // Simulating indexed allocation
    file = fopen("file.txt", "r");
    if (file == NULL) {
        printf("Error opening file!\n");
        return;
    }

    char buffer[100];
    int index = 0;
    while (fgets(buffer, sizeof(buffer), file)) {
        printf("Index %d -> %s", index++, buffer);  // Accessing via index
    }
    fclose(file);
}

int main() {
    indexed_allocation();
    return 0;
}
```

Question 36: File Allocation Strategy - Linked Allocation

Aim: To simulate the file allocation strategy where each file is a linked list of disk blocks, and each block points to the next.

Algorithm:

1. Use linked blocks to simulate file allocation, where each block contains a pointer to the next block.
2. The directory stores pointers to the first and last blocks.

Code:

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

```c
#include <stdlib.h>

struct Block {
    int data;
    struct Block* next;
};

void linked_allocation() {
    // Simulating linked allocation with linked blocks
    struct Block* head = (struct Block*)malloc(sizeof(struct Block));
    head->data = 1;
    head->next = (struct Block*)malloc(sizeof(struct Block));
    head->next->data = 2;
    head->next->next = (struct Block*)malloc(sizeof(struct Block));
    head->next->next->data = 3;
    head->next->next->next = NULL;

    // Traversing the linked blocks (files)
    struct Block* temp = head;
    while (temp != NULL) {
        printf("Block Data: %d\n", temp->data);
        temp = temp->next;
    }

    // Free memory
    temp = head;
    while (temp != NULL) {
        struct Block* to_free = temp;
        temp = temp->next;
        free(to_free);
    }
```

```
}

int main() {
    linked_allocation();
    return 0;
}
```

Question 37: First Come First Served (FCFS) Disk Scheduling Algorithm

Aim: To simulate the First Come First Served (FCFS) disk scheduling algorithm.

Algorithm:

1. FCFS processes disk requests in the order they arrive.
2. Calculate the total head movement based on the order of requests.

Code:

```
#include <stdio.h>

void fcfs_disk_scheduling(int requests[], int n, int start) {
    int total_head_movement = 0;
    int current_position = start;

    printf("Disk Scheduling Order: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", requests[i]);
        total_head_movement += abs(requests[i] - current_position);
        current_position = requests[i];
    }
    printf("\nTotal Head Movement: %d\n", total_head_movement);
}

int main() {
    int requests[] = {98, 183, 41, 122, 14, 124, 65, 67};
    int n = sizeof(requests) / sizeof(requests[0]);
```

```
    int start = 50;  // Start position of the disk head

    fcfs_disk_scheduling(requests, n, start);
    return 0;
}
```

Question 38: SCAN Disk Scheduling Algorithm

Aim: To simulate the SCAN disk scheduling algorithm.

Algorithm:

1. SCAN moves the disk arm in one direction, servicing requests until it reaches the end.
2. It then reverses direction and services requests on the way back.

Code:

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

void scan_disk_scheduling(int requests[], int n, int start, int disk_size) {
    int total_head_movement = 0;
    int direction = 1;  // 1 for right, -1 for left

    // Sort requests array
    for (int i = 0; i < n-1; i++) {
        for (int j = i+1; j < n; j++) {
            if (requests[i] > requests[j]) {
                int temp = requests[i];
                requests[i] = requests[j];
                requests[j] = temp;
            }
        }
    }

    // Move disk head in the chosen direction
```

```
    for (int i = 0; i < n; i++) {
        if (requests[i] >= start) {
            printf("Service Request: %d\n", requests[i]);
            total_head_movement += abs(requests[i] - start);
            start = requests[i];
        }
    }

    printf("Total Head Movement: %d\n", total_head_movement);
}

int main() {
    int requests[] = {98, 183, 41, 122, 14, 124, 65, 67};
    int n = sizeof(requests) / sizeof(requests[0]);
    int start = 50;  // Initial head position
    int disk_size = 200;

    scan_disk_scheduling(requests, n, start, disk_size);
    return 0;
}
```

Question 39: C-SCAN Disk Scheduling Algorithm

Aim: To simulate the C-SCAN disk scheduling algorithm.

Algorithm:

1. C-SCAN is similar to SCAN, but it moves the disk head in one direction, and when it reaches the end, it jumps to the beginning and starts servicing requests again.

Code:

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

void cscan_disk_scheduling(int requests[], int n, int start, int disk_size) {
```

```
    int total_head_movement = 0;
    int direction = 1;  // 1 for right

    // Sort requests
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (requests[i] > requests[j]) {
                int temp = requests[i];
                requests[i] = requests[j];
                requests[j] = temp;
            }
        }
    }

    // Move in one direction and jump back when reaching end
    for (int i = 0; i < n; i++) {
        if (requests[i] >= start) {
            printf("Service Request: %d\n", requests[i]);
            total_head_movement += abs(requests[i] - start);
            start = requests[i];
        }
    }

    printf("Total Head Movement: %d\n", total_head_movement);
}

int main() {
    int requests[] = {98, 183, 41, 122, 14, 124, 65, 67};
    int n = sizeof(requests) / sizeof(requests[0]);
    int start = 50;
    int disk_size = 200;
```

```
    cscan_disk_scheduling(requests, n, start, disk_size);
    return 0;
}
```

Question 40: File Access Permissions and Users in Linux

Aim: To illustrate file access permissions and different types of users in Linux.

Explanation:

- In Linux, file access permissions are divided into read, write, and execute permissions.
- Permissions are set for three categories of users: owner, group, and others.
- chmod command is used to set file permissions.

Code Example:

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

void display_permissions(mode_t mode) {
    printf("File Permissions: \n");

    // Owner permissions
    printf("Owner: ");
    if (mode & S_IRUSR) printf("r");
    else printf("-");
    if (mode & S_IWUSR) printf("w");
    else printf("-");
    if (mode & S_IXUSR) printf("x");
    else printf("-");

    // Group permissions
    printf(" Group: ");
```

```
    if (mode & S_IRGRP) printf("r");
    else printf("-");
    if (mode & S_IWGRP) printf("w");
    else printf("-");
    if (mode & S_IXGRP) printf("x");
    else printf("-");

    // Others permissions
    printf(" Others: ");
    if (mode & S_IROTH) printf("r");
    else printf("-");
    if (mode & S_IWOTH) printf("w");
    else printf("-");
    if (mode & S_IXOTH) printf("x");
    else printf("-");

    printf("\n");
}

int main() {
    // Create a new file (if not exists)
    FILE *file = fopen("myfile.txt", "w");
    if (file == NULL) {
        printf("Error creating file\n");
        return 1;
    }
    fprintf(file, "This is a test file.\n");
    fclose(file);

    // Set file permissions using chmod (Owner: rwx, Group: r-x, Others: r--)
    if (chmod("myfile.txt", S_IRUSR | S_IWUSR | S_IXUSR | S_IRGRP | S_IXGRP | S_IROTH) < 0) {
```

```
        perror("chmod failed");
        return 1;
    }

    // Get and display the file permissions using stat
    struct stat file_info;
    if (stat("myfile.txt", &file_info) < 0) {
        perror("stat failed");
        return 1;
    }

    // Display file permissions
    display_permissions(file_info.st_mode);

    return 0;
}
```