1. Create a new process by invoking the appropriate system call. Get the process identifier of the currently running process and its respective parent using system calls and display the same using a C program.

   Program:

```
#include <stdio.h>
#include <unistd.h>
int main() {
  if (fork() == 0) {
    // Child process
    printf("Child Process: PID = %d, PPID = %d\n", getpid(), getppid());
  } else {
    printf("Parent Process: PID = %d\n", getpid());
  }
  return 0;
}
```

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

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
FILE *fptr1, *fptr2;
char filename[100], c;
printf("Enter the filename to open for reading \n");
scanf("%s", filename);
fptr1 = fopen(filename, "r");
 if (fptr1 == NULL)
{
printf("Cannot open file %s \n", filename);
exit(0);
}
printf("Enter the filename to open for writing \n");
scanf("%s", filename);
fptr2 = fopen(filename, "w");
 if (fptr2 == NULL)
{
printf("Cannot open file %s \n", filename);
exit(0);
}
c = fgetc(fptr1);
while (c != EOF)
{
```

```
fputc(c, fptr2);
c = fgetc(fptr1);
}
printf("\nContents copied to %s", filename);
fclose(fptr1);
fclose(fptr2);
return 0;
}
```

3.Design a CPU scheduling program with C using First ComeFirst Served

Program:

```
#include <stdio.h>
int main() {
  int n;
  printf("Enter number of processes: ");
  scanf("%d", &n);

  int bt[n], wt[n], tat[n], total_wt = 0, total_tat = 0;
  printf("Enter burst times:\n");
  for (int i = 0; i < n; i++) {
    printf("P%d: ", i + 1);
    scanf("%d", &bt[i]);
    wt[i] = (i == 0) ? 0 : wt[i - 1] + bt[i - 1];
    tat[i] = wt[i] + bt[i];
    total_wt += wt[i];
    total_tat += tat[i];
  }

  printf("\nP\tBT\tWT\tTAT\n");
  for (int i = 0; i < n; i++) {
    printf("P%d\t%d\t%d\t%d\n", i + 1, bt[i], wt[i], tat[i]);
  }
  printf("\nAvg WT: %.2f\nAvg TAT: %.2f\n", (float)total_wt / n, (float)total_tat / n);
```

```
    return 0;
}
```

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

Program:

```
#include <stdio.h>
int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);

    int bt[n], wt[n], tat[n], total_wt = 0, total_tat = 0;
    printf("Enter burst times:\n");
    for (int i = 0; i < n; i++) {
        printf("P%d: ", i + 1);
        scanf("%d", &bt[i]);
        wt[i] = (i == 0) ? 0 : wt[i - 1] + bt[i - 1];
        tat[i] = wt[i] + bt[i];
        total_wt += wt[i];
        total_tat += tat[i];
    }

    printf("\nP\tBT\tWT\tTAT\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\n", i + 1, bt[i], wt[i], tat[i]);
    }
    printf("\nAvg WT: %.2f\nAvg TAT: %.2f\n", (float)total_wt / n, (float)total_tat / n);
```

```
    return 0;
}
```

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

Program:

```
#include <stdio.h>
#define MAX 10
typedef struct { int id, priority; } Process;
void schedule(Process p[], int n) {
    printf("\nExecution Order:\n");
    for (int i = 0; i < n; i++) {
        int min = i;
            for (int j = i + 1; j < n; j++)
            if (p[j].priority < p[min].priority) min = j;
        Process temp = p[i];
        p[i] = p[min];
        p[min] = temp;
        printf("Process %d (Priority %d)\n", p[i].id, p[i].priority);
    }
}
int main() {
    Process p[MAX];
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    printf("Enter process details (ID Priority):\n");
    for (int i = 0; i < n; i++)
scanf("%d %d", &p[i].id, &p[i].priority);
    schedule(p, n);
```

```
    return 0;
}
```

6.Construct a C program to simulate Round Robin scheduling algorithmwith C.

Program:

```
#include <stdio.h>
#define MAX 10

typedef struct { int id, burst, remaining, tat, wt; } Process;

void round_robin(Process p[], int n, int quantum) {
    int time = 0, done = 0;
    float total_tat = 0, total_wt = 0;

    printf("\nExecution Order:\n");
    while (done < n) {
        for (int i = 0; i < n; i++) {
            if (p[i].remaining > 0){
                int exec = (p[i].remaining < quantum) ? p[i].remaining : quantum;
                printf("P%d [%d-%d] ", p[i].id, time, time + exec);
                time += exec;
                p[i].remaining -= exec;
                if (p[i].remaining == 0) {
                    done++;
                    p[i].tat = time;
                    p[i].wt = p[i].tat - p[i].burst;
                    total_tat += p[i].tat;
                    total_wt += p[i].wt;
```

```
            }
          }
        }
    }

    printf("\n\nID\tBurst\tTAT\tWT\n");
    for (int i = 0; i < n; i++)
        printf("%d\t%d\t%d\t%d\n", p[i].id, p[i].burst, p[i].tat, p[i].wt);

    printf("\nAverage TAT: %.2f, Average WT: %.2f\n", total_tat / n, total_wt / n);
}

int main() {
    Process p[MAX];
    int n, quantum;

    printf("Enter number of processes and time quantum: ");
    scanf("%d %d", &n, &quantum);
    printf("Enter process details (ID BurstTime):\n");
    for (int i = 0; i < n; i++)
        scanf("%d %d", &p[i].id, &p[i].burst), p[i].remaining = p[i].burst;

    round_robin(p, n, quantum);
    return 0;
}
```

7.Construct a C program to implement non-preemptive SJFalgorithm

Program:

```
#include <stdio.h>
#define MAX 10
typedef struct { int id, burst, tat, wt; } Process;
void sjf(Process p[], int n) {
```

```
    float total_tat = 0, total_wt = 0;
    for (int i = 0; i < n - 1; i++) // Sort by Burst Time
        for (int j = i + 1; j < n; j++)
            if (p[i].burst > p[j].burst) {
                Process temp = p[i]; p[i] = p[j]; p[j] = temp;
            }
    int time = 0;
    printf("\nID\tBurst\tTAT\tWT\n");
    for (int i = 0; i < n; i++) {
        p[i].tat = (time += p[i].burst);
        p[i].wt = p[i].tat - p[i].burst;
        total_tat += p[i].tat;
        total_wt += p[i].wt;
        printf("%d\t%d\t%d\t%d\n", p[i].id, p[i].burst, p[i].tat, p[i].wt);
    }

    printf("\nAvg TAT: %.2f, Avg WT: %.2f\n", total_tat / n, total_wt / n);
}
int main() {
    Process p[MAX];
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        scanf("%d %d", &p[i].id, &p[i].burst);
    sjf(p, n);
    return 0;
}
```

8. Construct a C program to simulate Round Robin scheduling algorithm with C.

Program:

```c
#include <stdio.h>
#define MAX 10

typedef struct { int id, burst, remaining, tat, wt; } Process;

void roundRobin(Process p[], int n, int quantum) {
    int time = 0, completed = 0;
    float total_tat = 0, total_wt = 0;

    while (completed < n) {
        for (int i = 0; i < n; i++) {
            if (p[i].remaining > 0) {
                int exec_time = (p[i].remaining > quantum) ? quantum : p[i].remaining;
                p[i].remaining -= exec_time;
                time += exec_time;
                if (p[i].remaining == 0) {
                    p[i].tat = time;
                    p[i].wt = p[i].tat - p[i].burst;
                    total_tat += p[i].tat;
                    total_wt += p[i].wt;
                    completed++;
                }
            }
        }
    }

    printf("\nID\tBurst\tTAT\tWT\n");
    for (int i = 0; i < n; i++)
        printf("%d\t%d\t%d\t%d\n", p[i].id, p[i].burst, p[i].tat, p[i].wt);
```

```
    printf("\nAvg TAT: %.2f, Avg WT: %.2f", total_tat / n, total_wt / n);
}

int main() {
    Process p[MAX];
    int n, quantum;

    printf("Enter number of processes: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d %d", &p[i].id, &p[i].burst);
        p[i].remaining = p[i].burst;
    }

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

    roundRobin(p, n, quantum);
    return 0;
}
```

9 Illustrate the concept of inter-process communication using sharedmemory with a C program

Program:

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

int main() {
```

```
    key_t key = 1234; // Unique key for shared memory
    int shmid = shmget(key, 1024, 0666 | IPC_CREAT); // Create shared memory segment
    char *shared_mem = (char *)shmat(shmid, NULL, 0); // Attach shared memory

    if (fork() == 0) { // Child process
        sleep(1); // Wait for parent to write
        printf("Child reads: %s\n", shared_mem);
        shmdt(shared_mem); // Detach shared memory
    } else { // Parent process
        strcpy(shared_mem, "Hello from shared memory!"); // Write to shared memory
        printf("Parent writes: %s\n", shared_mem);
        wait(NULL); // Wait for child process to finish
        shmdt(shared_mem); // Detach shared memory
        shmctl(shmid, IPC_RMID, NULL); // Destroy shared memory
    }
    return 0;
}
```

10. Illustrate the concept of inter-process communication usingmessage queue with a c program

Program:

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

struct msg_buffer {
    long msg_type;
    char msg_text[100];
};
```

```
int main() {
    key_t key = 1234; // Unique key for message queue
    int msgid = msgget(key, 0666 | IPC_CREAT); // Create message queue

    if (fork() == 0) { // Child process
        struct msg_buffer msg;
        msgrcv(msgid, &msg, sizeof(msg.msg_text), 1, 0); // Receive message
        printf("Child received: %s\n", msg.msg_text);
    } else { // Parent process
        struct msg_buffer msg;
        msg.msg_type = 1;
        strcpy(msg.msg_text, "Hello from message queue!"); // Prepare message
        msgsnd(msgid, &msg, sizeof(msg.msg_text), 0); // Send message
        printf("Parent sent: %s\n", msg.msg_text);
        wait(NULL); // Wait for child process to finish
        msgctl(msgid, IPC_RMID, NULL); // Remove message queue
    }
    return 0;
}
```

11. Illustrate the concept of multithreading using a C program

Program:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
sem_t forks[5];
void* dine(void* arg) {
    int id = *(int*)arg;
    sem_wait(&forks[id]); sem_wait(&forks[(id + 1) % 5]);
```

```
    printf("Philosopher %d eating\n", id);
    sem_post(&forks[(id + 1) % 5]); sem_post(&forks[id]);
    return NULL;
}
int main() {
    pthread_t philosophers[5]; int ids[5];
    for (int i = 0; i < 5; i++) sem_init(&forks[i], 0, 1);
    for (int i = 0; i < 5; i++) { ids[i] = i; pthread_create(&philosophers[i], NULL, dine, &ids[i]); }
    for (int i = 0; i < 5; i++) pthread_join(philosophers[i], NULL);
    return 0;
}
```

12. dining philosphers

Program:

```
#include <pthread.h>
#include <stdio.h>
#define N 5

pthread_mutex_t forks[N];  // Correctly declared forks array

void *philosopher(void *id) {
    int i = *(int *)id;
    while (1) {
        pthread_mutex_lock(&forks[i]);           // Corrected variable name
        pthread_mutex_lock(&forks[(i + 1) % N]);  // Corrected variable name
        printf("Philosopher %d is eating\n", i);
        pthread_mutex_unlock(&forks[i]);
        pthread_mutex_unlock(&forks[(i + 1) % N]);
    }
}
```

```c
int main() {
    pthread_t threads[N];
    int ids[N];
    for (int i = 0; i < N; i++) {
        pthread_mutex_init(&forks[i], NULL);  // Corrected variable name
        ids[i] = i;
        pthread_create(&threads[i], NULL, philosopher, &ids[i]);
    }
    for (int i = 0; i < N; i++) pthread_join(threads[i], NULL);
    return 0;
}
```

13. Construct a C program to implement various memory allocationstrategies.

Program:

```c
#include <stdio.h>
#define N 4
#define M 5

void firstFit(int p[], int b[]) {
    printf("First Fit:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            if (b[j] >= p[i]) {
                printf("Process %d -> Block %d\n", i + 1, j + 1);
                b[j] -= p[i];
                break;
            }
        }
    }
}
```

```
void bestFit(int p[], int b[]) {
    printf("Best Fit:\n");
    for (int i = 0; i < N; i++) {
        int bestIdx = -1;
        for (int j = 0; j < M; j++) {
            if (b[j] >= p[i] && (bestIdx == -1 || b[j] < b[bestIdx]))
                bestIdx = j;
        }
        if (bestIdx != -1) {
            printf("Process %d -> Block %d\n", i + 1, bestIdx + 1);
            b[bestIdx] -= p[i];
        }
    }
}

void worstFit(int p[], int b[]) {
    printf("Worst Fit:\n");
    for (int i = 0; i < N; i++) {
        int worstIdx = -1;
        for (int j = 0; j < M; j++) {
            if (b[j] >= p[i] && (worstIdx == -1 || b[j] > b[worstIdx]))
                worstIdx = j;
        }
        if (worstIdx != -1) {
            printf("Process %d -> Block %d\n", i + 1, worstIdx + 1);
            b[worstIdx] -= p[i];
        }
    }
}
```

```
int main() {
    int block[] = {100, 500, 200, 300, 600};
    int process[] = {212, 417, 112, 426};

    int b1[M], b2[M], b3[M];
    for (int i = 0; i < M; i++) b1[i] = b2[i] = b3[i] = block[i];

    firstFit(process, b1);
    bestFit(process, b2);
    worstFit(process, b3);

    return 0;
}
```

14. .Construct a C program to organize the file using single leveldirectory

Program:

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

#define MAX_FILES 10

typedef struct {
    char name[20];
} File;

int main() {
    File dir[MAX_FILES];
    int count = 0, choice;

    while (1) {
        printf("\n1. Create 2. List 3. Delete 4. Exit\n");
        printf("Choice: ");
```

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

        if (choice == 1 && count < MAX_FILES) { // Create File
            printf("Enter file name: ");
            scanf("%s", dir[count++].name);
        } else if (choice == 2) { // List Files
            if (count) for (int i = 0; i < count; i++) printf("%d. %s\n", i+1, dir[i].name);
            else printf("No files.\n");
        } else if (choice == 3) { // Delete File
            char name[20];
            printf("Enter file name to delete: ");
            scanf("%s", name);
            int i, found = 0;
            for (i = 0; i < count; i++) {
                if (strcmp(dir[i].name, name) == 0) {
                    found = 1;
                    for (; i < count - 1; i++) dir[i] = dir[i + 1];
                    count--;
                    break;
                }
            }
            if (!found) printf("File not found.\n");
            else printf("File deleted.\n");
        } else if (choice == 4) break; // Exit
        else printf("Invalid choice.\n");
    }
    return 0;
}
```

15.Design a C program to organize the file using two level directorystructure.

Program:

```
#include <stdio.h>

#define MAX_DIRS 5
#define MAX_FILES 5

typedef struct {
   char dir_name[20], files[MAX_FILES][20];
   int file_count;
} Directory;

int main() {
   Directory dirs[MAX_DIRS];
   int dir_count = 0, choice, dir_index;

   while (1) {
      printf("\n1. Create Dir 2. List Dir 3. Add File 4. List Files 5. Exit\nChoice: ");
      scanf("%d", &choice);

      if (choice == 1 && dir_count < MAX_DIRS) {
         printf("Enter dir name: ");
         scanf("%s", dirs[dir_count].dir_name);
         dirs[dir_count++].file_count = 0;
      } else if (choice == 2) {
         for (int i = 0; i < dir_count; i++) printf("%d. %s\n", i + 1, dirs[i].dir_name);
      } else if (choice == 3) {
         printf("Enter dir index to add file: ");
         scanf("%d", &dir_index);
         dir_index--;
         if (dir_index >= 0 && dir_index < dir_count && dirs[dir_index].file_count < MAX_FILES) {
            printf("Enter file name: ");
```

```
            scanf("%s", dirs[dir_index].files[dirs[dir_index].file_count++]);
          } else {
            printf("Invalid dir or file limit.\n");
          }
        } else if (choice == 4) {
          printf("Enter dir index to list files: ");
          scanf("%d", &dir_index);
          dir_index--;
          if (dir_index >= 0 && dir_index < dir_count) {
            for (int i = 0; i < dirs[dir_index].file_count; i++) printf("%d. %s\n", i + 1,
dirs[dir_index].files[i]);
          } else printf("Invalid dir index.\n");
        } else if (choice == 5) break;
        else printf("Invalid choice.\n");
    }

    return 0;
}
```

16. Develop a C program for implementing random access file for processing the employee details.

Program:

```
#include <stdio.h>
#include <string.h>
#define MAX 5
struct Employee {
    int id;
    char name[30];
    float salary;
```

```
};

int main() {
    FILE *file = fopen("employees.dat", "r+b");
    if (!file) file = fopen("employees.dat", "w+b");
    struct Employee emp;
    int choice, pos;

    while (1) {
        printf("\n1. Add 2. Display 3. Exit: ");
        scanf("%d", &choice);
        if (choice == 3) break;

        printf("Position (0-%d): ", MAX-1);
        scanf("%d", &pos);

        if (pos < 0 || pos >= MAX) {
            printf("Invalid position.\n");
            continue;
        }

        fseek(file, pos * sizeof(emp), SEEK_SET);

        if (choice == 1) {
            printf("ID: "); scanf("%d", &emp.id);
            getchar(); // clear buffer
            printf("Name: "); fgets(emp.name, sizeof(emp.name), stdin);
            emp.name[strcspn(emp.name, "\n")] = 0;
            printf("Salary: "); scanf("%f", &emp.salary);
            fwrite(&emp, sizeof(emp), 1, file);
        } else if (choice == 2) {
```

```
        if (fread(&emp, sizeof(emp), 1, file))
            printf("ID: %d\nName: %s\nSalary: %.2f\n", emp.id, emp.name, emp.salary);
        else
            printf("No record found.\n");
    } else {
        printf("Invalid choice.\n");
    }
  }

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

17. Illustrate the deadlock avoidance concept by simulating Banker's algorithm with C.

Program:

```
#include <stdio.h>
#define P 5  // Number of processes
#define R 3  // Number of resources
int allocation[P][R], max[P][R], need[P][R], available[R];
int is_safe() {
  int work[R], finish[P] = {0}, safe_seq[P], count = 0;
  for (int i = 0; i < R; i++) work[i] = available[i];
  while (count < P) {
    int found = 0;
    for (int i = 0; i < P; i++) {
      if (!finish[i]) {
        int can_allocate = 1;
        for (int j = 0; j < R; j++) {
          if (need[i][j] > work[j]) {
            can_allocate = 0;
            break;
```

```
                }
            }
            if (can_allocate) {
                for (int j = 0; j < R; j++) work[j] += allocation[i][j];
                finish[i] = 1;
                safe_seq[count++] = i;
                found = 1;
            }
        }
    }
    if (!found) return 0;  // No safe sequence
  }

  printf("Safe Sequence: ");
  for (int i = 0; i < P; i++) printf("P%d ", safe_seq[i]);
  printf("\n");
  return 1;
}

int main() {
  int i, j;

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

  printf("Enter allocation matrix:\n");
  for (i = 0; i < P; i++)
    for (j = 0; j < R; j++) scanf("%d", &allocation[i][j]);

  printf("Enter max matrix:\n");
  for (i = 0; i < P; i++)
```

```c
        for (j = 0; j < R; j++) scanf("%d", &max[i][j]);

    // Calculate need matrix
    for (i = 0; i < P; i++)
        for (j = 0; j < R; j++) need[i][j] = max[i][j] - allocation[i][j];

    if (!is_safe()) {
        printf("System is not in a safe state.\n");
    }

    return 0;
}
```

18. Construct a C program to simulate producer-consumer problem using semaphores.

Program:

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <stdlib.h>  // Include for rand() and srand()
#include <time.h>    // Include for time()

#define MAX 5
#define MAX_PRODUCTIONS 10  // Limit the number of items to produce and consume

sem_t empty, full;
pthread_mutex_t mutex;
int buffer[MAX], in = 0, out = 0;

void* producer(void* arg) {
    int count = 0;
```

```
    while (count < MAX_PRODUCTIONS) {
        int item = rand() % 100;  // Generate random item
        sem_wait(&empty);        // Wait for an empty slot
        pthread_mutex_lock(&mutex);  // Enter critical section

        buffer[in] = item;  // Produce item
        printf("Produced: %d at %d\n", item, in);
        in = (in + 1) % MAX;  // Move to next slot
        count++;

        pthread_mutex_unlock(&mutex);  // Exit critical section
        sem_post(&full);        // Signal that there's a full slot
        sleep(1);  // Simulate production delay
    }
    return NULL;
}

void* consumer(void* arg) {
    int count = 0;
    while (count < MAX_PRODUCTIONS) {
        sem_wait(&full);        // Wait for a full slot
        pthread_mutex_lock(&mutex);  // Enter critical section

        int item = buffer[out];  // Consume item
        printf("Consumed: %d from %d\n", item, out);
        out = (out + 1) % MAX;  // Move to next slot
        count++;

        pthread_mutex_unlock(&mutex);  // Exit critical section
        sem_post(&empty);        // Signal that there's an empty slot
        sleep(1);  // Simulate consumption delay
```

```
    }
    return NULL;
}

int main() {
    pthread_t prod, cons;

    // Initialize semaphores
    sem_init(&empty, 0, MAX);  // All slots are empty initially
    sem_init(&full, 0, 0);     // No slots are full initially
    pthread_mutex_init(&mutex, NULL);  // Initialize mutex

    // Seed random number generator
    srand(time(NULL));

    // Create producer and consumer threads
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    // Wait for threads to finish
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    // Destroy semaphores and mutex after use
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

19. Design a C program to implement process synchronization using mutex locks.

Program:

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 5
pthread_mutex_t lock;  // Mutex lock
int counter = 0;       // Shared resource

void* increment(void* arg) {
   pthread_mutex_lock(&lock);
   counter++;
   printf("Thread %ld incremented counter to: %d\n", (long*)arg, counter);
   pthread_mutex_unlock(&lock);  // Unlock the mutex
   return NULL;
}
int main() {
   pthread_t threads[NUM_THREADS];

   // Initialize mutex
   pthread_mutex_init(&lock, NULL);

   // Create threads
   for (long i = 0; i < NUM_THREADS; i++) {
      pthread_create(&threads[i], NULL, increment, (void*)i);
   }
   for (int i = 0; i < NUM_THREADS; i++) {
      pthread_join(threads[i], NULL);
   }
```

```
    // Destroy mutex
    pthread_mutex_destroy(&lock);

    printf("Final counter value: %d\n", counter);
    return 0;
}
```

20. Construct a C program to simulate Reader-Writer problem using Semaphores.

Program:

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

sem_t rw_mutex, mutex;
int read_count = 0, data = 0;

void* reader(void* arg) {
    sem_wait(&mutex);
    read_count++;
    if (read_count == 1) sem_wait(&rw_mutex);
    sem_post(&mutex);

    printf("Reader %d: Read data = %d\n", (int)arg, data);

    sem_wait(&mutex);
    read_count--;
    if (read_count == 0) sem_post(&rw_mutex);
    sem_post(&mutex);
```

```c
    return NULL;
}

void* writer(void* arg) {
    sem_wait(&rw_mutex);
    data++;
    printf("Writer %d: Wrote data = %d\n", (int)arg, data);
    sem_post(&rw_mutex);
    return NULL;
}

int main() {
    pthread_t r[5], w[5];
    sem_init(&rw_mutex, 0, 1);
    sem_init(&mutex, 0, 1);

    int ids[5] = {1, 2, 3, 4, 5};
    for (int i = 0; i < 5; i++) {
        pthread_create(&r[i], NULL, reader, &ids[i]);
        pthread_create(&w[i], NULL, writer, &ids[i]);
    }
    for (int i = 0; i < 5; i++) {
        pthread_join(r[i], NULL);
        pthread_join(w[i], NULL);
    }

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

21. WORST FIT

```c
#include <stdio.h>

void worstFit(int blocks[], int bSize, int processes[], int pSize) {
    int allocation[pSize];
    for (int i = 0; i < pSize; i++) allocation[i] = -1;

    for (int i = 0; i < pSize; i++) {
        int worstIdx = -1;
        for (int j = 0; j < bSize; j++) {
            if (blocks[j] >= processes[i] &&
                (worstIdx == -1 || blocks[j] > blocks[worstIdx])) {
                worstIdx = j;
            }
        }
        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            blocks[worstIdx] -= processes[i];
        }
    }

    printf("Process\tSize\tBlock\n");
    for (int i = 0; i < pSize; i++) {
        printf("%d\t%d\t", i + 1, processes[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}
```

```c
int main() {
    int blocks[] = {100, 500, 200, 300, 600};
    int processes[] = {212, 417, 112, 426};
    int bSize = sizeof(blocks) / sizeof(blocks[0]);
    int pSize = sizeof(processes) / sizeof(processes[0]);

    worstFit(blocks, bSize, processes, pSize);
    return 0;
}
```

22.BEST FIT

```c
#include <stdio.h>

void bestFit(int blocks[], int bSize, int processes[], int pSize) {
    int allocation[pSize];
    for (int i = 0; i < pSize; i++) allocation[i] = -1;

    for (int i = 0; i < pSize; i++) {
        int bestIdx = -1;
        for (int j = 0; j < bSize; j++) {
            if (blocks[j] >= processes[i] &&
                (bestIdx == -1 || blocks[j] < blocks[bestIdx])) {
                bestIdx = j;
            }
        }
        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            blocks[bestIdx] -= processes[i];
        }
```

```
    }

    printf("Process\tSize\tBlock\n");
    for (int i = 0; i < pSize; i++) {
        printf("%d\t%d\t", i + 1, processes[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}

int main() {
    int blocks[] = {100, 500, 200, 300, 600};
    int processes[] = {212, 417, 112, 426};
    int bSize = sizeof(blocks) / sizeof(blocks[0]);
    int pSize = sizeof(processes) / sizeof(processes[0]);

    bestFit(blocks, bSize, processes, pSize);
    return 0;
}
```

23.FIRST-FIT

```
#include <stdio.h>

void firstFit(int blocks[], int bSize, int processes[], int pSize) {
    int allocation[pSize];
    for (int i = 0; i < pSize; i++) allocation[i] = -1;
```

```c
    for (int i = 0; i < pSize; i++) {
        for (int j = 0; j < bSize; j++) {
            if (blocks[j] >= processes[i]) {
                allocation[i] = j;
                blocks[j] -= processes[i];
                break;
            }
        }
    }

    printf("Process\tSize\tBlock\n");
    for (int i = 0; i < pSize; i++) {
        printf("%d\t%d\t", i + 1, processes[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}

int main() {
    int blocks[] = {100, 500, 200, 300, 600};
    int processes[] = {212, 417, 112, 426};
    int bSize = sizeof(blocks) / sizeof(blocks[0]);
    int pSize = sizeof(processes) / sizeof(processes[0]);

    firstFit(blocks, bSize, processes, pSize);
    return 0;
}
```

24.unix systemcalls

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

int main() {
    int fd;
    char buffer[100];

    // Create and open a file
    fd = open("example.txt", O_CREAT | O_RDWR, 0644);
    if (fd < 0) {
        perror("Failed to open file");
        return 1;
    }

    // Write to the file
    write(fd, "Hello, UNIX system calls!", 25);

    // Move file pointer to the beginning
    lseek(fd, 0, SEEK_SET);

    // Read from the file
    read(fd, buffer, 25);
    buffer[25] = '\0'; // Null-terminate the string

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

    // Close the file
    close(fd);
```

```
    return 0;
}
```

25.i/o system calls of unix

```
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>
extern int errno;
int main()
{

int fd = open("foo.txt", O_RDONLY | O_CREAT);
printf("fd = %d\n", fd);
if (fd ==-1)
{
printf("Error Number % d\n", errno);
perror("Program");
}
return 0;
}
```

26. Construct a C program to implement the file management operations.

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

int main() {
    char buffer[100];
```

```
    int fd = open("example.txt", O_CREAT | O_RDWR, 0644);
    write(fd, "Hello, File!", 12);
    lseek(fd, 0, SEEK_SET);
    read(fd, buffer, 12);
    buffer[12] = '\0';
    printf("File Content: %s\n", buffer);
    close(fd);

    unlink("example.txt");
    return 0;
}
```

27. Develop a C program for simulating the function of ls UNIX Command.

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

int main() {
    char fn[100], pat[100], temp[200];
    FILE *fp;

    printf("Enter file name: ");
    scanf("%s", fn);
    printf("Enter the pattern: ");
    scanf("%s", pat);

    fp = fopen(fn, "r");
    if (fp == NULL) {
        perror("Error opening file");
        return 1;
    }
```

```
    while (fgets(temp, sizeof(temp), fp) != NULL) {
        if (strstr(temp, pat)) { // Check if the pattern exists in the line
            printf("%s", temp);
        }
    }

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

28. Write a C program for simulation of GREP UNIX command

```
#include <stdio.h>
#include <stdlib.h>>
#include <string.h>
#define MAX_LINE_LENGTH 1024
void searchFile(const char *pattern, const char *filename)
{
FILE *file = fopen(filename, "r");
if (file == NULL) {
perror("Error opening file"); exit(1);
}
char line[MAX_LINE_LENGTH]; while
(fgets(line, sizeof(line), file)) {
if (strstr(line, pattern) != NULL) {
printf("%s", line);
}
}
fclose(file);
}
```

```
int main(int argc, char *argv[]) {
if (argc != 3) {
fprintf(stderr, "Usage: %s <pattern> <filename>\n", argv[0]);
return 1;
}
const char *pattern = argv[1];
const char *filename = argv[2];
searchFile(pattern, filename);
return 0;
}
```

29. Write a C program to simulate the solution of Classical Process Synchronization Problem

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

int buffer, count = 0;
sem_t empty, full;
pthread_mutex_t mutex;

void *producer(void *arg) {
  for (int i = 1; i <= 5; i++) {
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);
    buffer = i;
    printf("Produced: %d\n", buffer);
    pthread_mutex_unlock(&mutex);
    sem_post(&full);
  }
  return NULL;
```

```
}

void *consumer(void *arg) {
    for (int i = 1; i <= 5; i++) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        printf("Consumed: %d\n", buffer);
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
    return NULL;
}

int main() {
    pthread_t prod, cons;
    sem_init(&empty, 0, 1);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    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;
}
```

30. Write C programs to demonstrate the following thread related concepts. (i)create (ii) join (iii) equal (iv) exit

```
#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

void* func(void* arg)

{

pthread_detach(pthread_self());

 printf("Inside the thread\n");

 pthread_exit(NULL);

}

void fun()

{

pthread_t ptid;

pthread_create(&ptid, NULL, &func, NULL);

printf("This line may be printed" " before thread terminates\n");

if(pthread_equal(ptid, pthread_self()))

{

printf("Threads are equal\n");

}

else

printf("Threads are not equal\n");

pthread_join(ptid, NULL);

printf("This line will be printed" " after thread ends\n");

pthread_exit(NULL);

}

int main()

{

fun();

return 0;

}
```

31. Construct a C program to simulate the First in First Out paging technique of memory management

```c
#include <stdio.h>

int main() {
   int pages[100], frames[10], n, f, faults = 0, idx = 0;
   printf("Enter number of pages: ");
   scanf("%d", &n);
   printf("Enter the page sequence: ");
   for (int i = 0; i < n; i++) scanf("%d", &pages[i]);
   printf("Enter number of frames: ");
   scanf("%d", &f);

   for (int i = 0; i < f; i++) frames[i] = -1; // Initialize frames

   for (int i = 0; i < n; i++) {
      int found = 0;
      for (int j = 0; j < f; j++) { // Check if the page is already in a frame
         if (frames[j] == pages[i]) found = 1;
      }
      if (!found) { // Page fault
         frames[idx] = pages[i];
         idx = (idx + 1) % f; // Circular index for FIFO
         faults++;
      }
      printf("Frames: ");
      for (int j = 0; j < f; j++) printf("%d ", frames[j]);
      printf("\n");
   }

   printf("Total Page Faults: %d\n", faults);
   return 0;
}
```

32. Construct a C program to simulate the Least Recently Used paging technique of memory management.

```
#include <stdio.h>
int main() {
    int pages[100], frames[10], time[10], n, f, faults = 0, counter = 0;
    printf("Enter number of pages: ");
    scanf("%d", &n);
    printf("Enter page sequence: ");
    for (int i = 0; i < n; i++) scanf("%d", &pages[i]);
    printf("Enter number of frames: ");
    scanf("%d", &f);

    for (int i = 0; i < f; i++) frames[i] = -1;

    for (int i = 0; i < n; i++) {
        int found = 0, lru = 0;
        for (int j = 0; j < f; j++) {
            if (frames[j] == pages[i]) {
                found = 1;
                time[j] = ++counter;
                break;
            }
            if (time[j] < time[lru]) lru = j;
        }
        if (!found) {
            frames[lru] = pages[i];
            time[lru] = ++counter;
            faults++;
        }
        printf("Frames: ");
```

```
        for (int j = 0; j < f; j++) printf("%d ", frames[j]);
        printf("\n");
    }
    printf("Total Page Faults: %d\n", faults);
    return 0;
}
```

33. Construct a C program to simulate the optimal paging technique of memory management

```
#include <stdio.h>
int findOptimal(int frames[], int pages[], int f, int n, int idx) {
    for (int i = 0; i < f; i++) {
        int found = 1;
        for (int j = idx; j < n; j++) {
            if (frames[i] == pages[j]) {
                found = 0;
                break;
            }
        }
        if (found) return i;
    }
    return 0;
}

int main() {
    int pages[100], frames[10], n, f, faults = 0;

    printf("Enter number of pages: ");
    scanf("%d", &n);
    printf("Enter page sequence: ");
    for (int i = 0; i < n; i++) scanf("%d", &pages[i]);
    printf("Enter number of frames: ");
```

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

    for (int i = 0; i < f; i++) frames[i] = -1;

    for (int i = 0; i < n; i++) {
        int found = 0;
        for (int j = 0; j < f; j++) {
            if (frames[j] == pages[i]) found = 1;
        }

        if (!found) {
            int pos = (i < f) ? i : findOptimal(frames, pages, f, n, i + 1);
            frames[pos] = pages[i];
            faults++;
        }

        for (int j = 0; j < f; j++) printf("%d ", frames[j]);
        printf("\n");
    }
    printf("Total Page Faults: %d\n", faults);
    return 0;
}
```

34. Consider a file system where the records of the file are stored one after another both physically and logically. A record of the file can only be accessed by reading all the previous records. Design a C program to simulate the file allocation strategy.

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

int main() {
    char file[100][100], record[100];
    int n;
```

```
    printf("Enter the number of records: ");
    scanf("%d", &n);
    printf("Enter the records:\n");
    for (int i = 0; i < n; i++) scanf("%s", file[i]);

    printf("Enter the record to search: ");
    scanf("%s", record);

    for (int i = 0; i < n; i++) {
        printf("Reading record: %s\n", file[i]);
        if (strcmp(file[i], record) == 0) {
            printf("Record '%s' found at position %d.\n", record, i + 1);
            return 0;
        }
    }

    printf("Record '%s' not found.\n", record);
    return 0;
}
```

35. Consider a file system that brings all the file pointers together into an index block. The ith entry in the index block points to the ith block of the file. Design a C program to simulate the file allocation strategy.#include <stdio.h>

```
#include <string.h>

#define MAX_BLOCKS 10
#define MAX_RECORDS 10

int main() {
    char file[MAX_BLOCKS][100], index_block[MAX_BLOCKS];
    int n, block_size;
```

```c
    // Input for number of blocks and block size
    printf("Enter number of blocks: ");
    scanf("%d", &n);
    printf("Enter block size: ");
    scanf("%d", &block_size);

    // Simulating the file blocks with records
    printf("Enter the records in each block:\n");
    for (int i = 0; i < n; i++) {
        printf("Block %d: ", i + 1);
        scanf("%s", file[i]);
        index_block[i] = i; // Simulate the index block, pointing to each block
    }

    // Input for record to search
    char record[100];
    printf("Enter record to search: ");
    scanf("%s", record);

    // Searching for the record in the blocks using the index block
    for (int i = 0; i < n; i++) {
        if (strcmp(file[index_block[i]], record) == 0) {
            printf("Record '%s' found at block %d.\n", record, index_block[i] + 1);
            return 0;
        }
    }

    printf("Record '%s' not found.\n", record);
    return 0;
}
```

36. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block. Design a C program to simulate the file allocation strategy.

Program:

```
#include <stdio.h>

#include <stdlib.h>


struct Block {

   int data;

   struct Block* next;

};


void display(struct Block* head) {

   while (head) {

      printf("%d -> ", head->data);

      head = head->next;

   }

   printf("NULL\n");

}


struct Block* allocate(int data, struct Block* last) {

   struct Block* newBlock = (struct Block*)malloc(sizeof(struct Block));

   newBlock->data = data;

   newBlock->next = NULL;

   if (last) last->next = newBlock;

   return newBlock;

}


int main() {

   struct Block *head = NULL, *last = NULL;

   int data;
```

```
    for (int i = 0; i < 3; i++) { // Simulate 3 blocks
        printf("Enter data for block %d: ", i + 1);
        scanf("%d", &data);
        last = allocate(data, last);
        if (!head) head = last;
    }

    printf("File blocks: ");
    display(head);
    return 0;
}
```

37. Construct a C program to simulate the First Come First Served disk scheduling algorithm.

Program:

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

int main() {
    int n, initial_head, total_head_movement = 0;

    // Input the number of disk requests and the initial position of the disk head
    printf("Enter the number of disk requests: ");
    scanf("%d", &n);

    int requests[n];
    printf("Enter the disk requests: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }

    printf("Enter the initial position of the disk head: ");
```

```c
    scanf("%d", &initial_head);

    // FCFS Disk Scheduling
    int current_position = initial_head;

    printf("\nDisk Access Order:\n");
    for (int i = 0; i < n; i++) {
        printf("Move from %d to %d\n", current_position, requests[i]);
        total_head_movement += abs(requests[i] - current_position);
        current_position = requests[i];
    }

    // Output the total head movement
    printf("\nTotal Head Movement: %d\n", total_head_movement);

    return 0;
}
```

38. Design a C program to simulate SCAN disk scheduling algorithm

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

void scanDiskScheduling(int requests[], int n, int head, int direction, int total_tracks) {
    int left = 0, right = 0, total_head_movement = 0;
    int left_arr[n], right_arr[n];

    // Divide the requests into left and right of the head
    for (int i = 0; i < n; i++) {
        if (requests[i] < head) left_arr[left++] = requests[i];
        else right_arr[right++] = requests[i];
    }
```

```
    // Sort left and right arrays
    for (int i = 0; i < left - 1; i++) for (int j = i + 1; j < left; j++) if (left_arr[i] < left_arr[j]) { int temp =
left_arr[i]; left_arr[i] = left_arr[j]; left_arr[j] = temp; }
    for (int i = 0; i < right - 1; i++) for (int j = i + 1; j < right; j++) if (right_arr[i] > right_arr[j]) { int temp =
right_arr[i]; right_arr[i] = right_arr[j]; right_arr[j] = temp; }

    // Move in the given direction
    if (direction == 0) { // Left
        for (int i = left - 1; i >= 0; i--) { total_head_movement += abs(head - left_arr[i]); head = left_arr[i];
}
        total_head_movement += head; head = 0;
        for (int i = 0; i < right; i++) { total_head_movement += abs(head - right_arr[i]); head = right_arr[i];
}
    } else { // Right
        for (int i = 0; i < right; i++) { total_head_movement += abs(head - right_arr[i]); head = right_arr[i];
}
        total_head_movement += (total_tracks - 1 - head); head = total_tracks - 1;
        for (int i = left - 1; i >= 0; i--) { total_head_movement += abs(head - left_arr[i]); head = left_arr[i];
}
    }

    printf("Total head movement: %d\n", total_head_movement);
}

int main() {
    int n, head, direction, total_tracks;
    printf("Enter number of requests: ");
    scanf("%d", &n);
    int requests[n];
    printf("Enter requests: ");
    for (int i = 0; i < n; i++) scanf("%d", &requests[i]);
    printf("Enter initial head position: ");
```

```
    scanf("%d", &head);
    printf("Enter total tracks: ");
    scanf("%d", &total_tracks);
    printf("Enter direction (0 for left, 1 for right): ");
    scanf("%d", &direction);

    scanDiskScheduling(requests, n, head, direction, total_tracks);
    return 0;
}
```

39. Develop a C program to simulate C-SCAN disk scheduling algorithm.

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

void cScanDiskScheduling(int requests[], int n, int head, int direction, int total_tracks) {
    int total_head_movement = 0, left = 0, right = 0;
    int left_arr[n], right_arr[n];

    // Divide requests into left and right of the head
    for (int i = 0; i < n; i++) {
        if (requests[i] < head) left_arr[left++] = requests[i];
        else right_arr[right++] = requests[i];
    }

    // Sort the arrays
    for (int i = 0; i < left - 1; i++) for (int j = i + 1; j < left; j++) if (left_arr[i] < left_arr[j]) { int temp = left_arr[i]; left_arr[i] = left_arr[j]; left_arr[j] = temp; }
    for (int i = 0; i < right - 1; i++) for (int j = i + 1; j < right; j++) if (right_arr[i] > right_arr[j]) { int temp = right_arr[i]; right_arr[i] = right_arr[j]; right_arr[j] = temp; }

    // Move in the given direction
    if (direction) {
```

```
        for (int i = 0; i < right; i++) total_head_movement += abs(head - right_arr[i]), head = right_arr[i];
        total_head_movement += (total_tracks - 1 - head), head = total_tracks - 1;
        for (int i = 0; i < left; i++) total_head_movement += abs(head - left_arr[i]), head = left_arr[i];
    } else {
        for (int i = left - 1; i >= 0; i--) total_head_movement += abs(head - left_arr[i]), head = left_arr[i];
        total_head_movement += head, head = 0;
        for (int i = 0; i < right; i++) total_head_movement += abs(head - right_arr[i]), head = right_arr[i];
    }

    printf("Total head movement: %d\n", total_head_movement);
}

int main() {
    int n, head, direction, total_tracks;
    printf("Enter number of requests: ");
    scanf("%d", &n);
    int requests[n];
    printf("Enter requests: ");
    for (int i = 0; i < n; i++) scanf("%d", &requests[i]);
    printf("Enter initial head position: ");
    scanf("%d", &head);
    printf("Enter total tracks: ");
    scanf("%d", &total_tracks);
    printf("Enter direction (0 for left, 1 for right): ");
    scanf("%d", &direction);

    cScanDiskScheduling(requests, n, head, direction, total_tracks);
    return 0;
}
```