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.

```
#include<stdio.h>

#include<unistd.h>

int main()

{

  printf("Process ID: %d\n", getpid() );

  printf("Parent Process ID: %d\n", getpid() );

  return 0;

}
```

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

```
#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>


int main() {

    FILE *fp1 = fopen("source.txt", "r");

    FILE *fp2 = fopen("destination.txt", "w");

    char buffer[1024];

    fread(buffer, 1, 1024, fp1);

    fwrite(buffer, 1, 1024, fp2);

    fclose(fp1);

    fclose(fp2);

    printf("file copied successfully\n");

    return 0;

}
```

3. Design a CPU scheduling program with C using First Come First Served technique with the following considerations.
a. All processes are activated at time 0.
b. Assume that no process waits on I/O devices

```
#include <stdio.h>

int main() {

   int n = 3, burst_time[] = {4, 3, 5}, start_time = 0, total_tat = 0, total_wt = 0;

   printf("\nPID\tBT\tST\tCT\tTAT\tWT\n");

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

      int completion_time = start_time + burst_time[i];

      int turnaround_time = completion_time;

      int waiting_time = turnaround_time - burst_time[i];

      printf("%d\t%d\t%d\t%d\t%d\t%d\n", i + 1, burst_time[i], start_time,

            completion_time, turnaround_time, waiting_time);

      total_tat += turnaround_time;

      total_wt += waiting_time;

      start_time = completion_time;

   }

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

}
```

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

```
#include <stdio.h>
#include <limits.h>
#define N 4
int main() {
   int bt[] = {6, 8, 7, 3};
   int ct[N], tat[N], wt[N], is_completed[N] = {0};
   int completed = 0, time = 0, total_tat = 0, total_wt = 0;
   while (completed < N) {
      int shortest = -1, min_bt = INT_MAX;
      for (int i = 0; i < N; i++) {
         if (!is_completed[i] && bt[i] < min_bt) {
            shortest = i;
            min_bt = bt[i];
         }
```

```
        }
        if (shortest == -1) {
            break;
        }
        time += bt[shortest];
        ct[shortest] = time;
        tat[shortest] = ct[shortest];
        wt[shortest] = tat[shortest] - bt[shortest];
        is_completed[shortest] = 1;
        total_tat += tat[shortest];
        total_wt += wt[shortest];
        completed++;
    }
    printf("\nPID\tBT\tCT\tTAT\tWT\n");
    for (int i = 0; i < N; i++) {
        printf("%d\t%d\t%d\t%d\t%d\n", i + 1, bt[i], ct[i], tat[i], wt[i]);
    }
    printf("\nAverage TAT: %.2f\n", (float)total_tat / N);
    printf("Average WT: %.2f\n", (float)total_wt / N);
    return 0;
}
```

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

```
#include <stdio.h>
#include <limits.h>
#define N 4
int main() {
    int bt[] = {6, 8, 7, 3}, pri[] = {2, 1, 3, 4}, ct[N], tat[N], wt[N], done[N] = {0};
    int completed = 0, time = 0, total_tat = 0, total_wt = 0;
    while (completed < N) {
        int highest = -1, min_pri = INT_MAX;
        for (int i = 0; i < N; i++) {
            if (!done[i] && pri[i] < min_pri) {
                highest = i;
                min_pri = pri[i];
            }
        }
        if (highest == -1) break;
        time += bt[highest];
        ct[highest] = time;
        tat[highest] = ct[highest];
        wt[highest] = tat[highest] - bt[highest];
        total_tat += tat[highest];
        total_wt += wt[highest];
        done[highest] = 1;
        completed++;
    }
    printf("\nPID\tBT\tPriority\tCT\tTAT\tWT\n");
    for (int i = 0; i < N; i++) {
```

```
        printf("%d\t%d\t%d\t\t%d\t%d\t%d\n", i + 1, bt[i], pri[i], ct[i], tat[i], wt[i]);
    }
    printf("\nAverage TAT: %.2f\nAverage WT: %.2f\n", (float)total_tat / N, (float)total_wt / N);
    return 0;
}
```

6. Construct a C program to implement preemptive priority scheduling algorithm

```
#include <stdio.h>
struct Process { int id, burstTime, arrivalTime, priority, remainingTime, waitingTime,
turnaroundTime; };
int findNext(struct Process p[], int n, int t) {
    int idx = -1, pr = 1e9;
    for (int i = 0; i < n; i++) if (p[i].remainingTime > 0 && p[i].arrivalTime <= t && p[i].priority
< pr) idx = i, pr = p[i].priority;
    return idx;
}
int main() {
    struct Process p[] = {{1, 5, 0, 2, 5, 0, 0}, {2, 3, 1, 1, 3, 0, 0}, {3, 8, 2, 3, 8, 0, 0}};
    int n = 3, t = 0, c = 0; float wt = 0, tat = 0;
    while (c < n) {
        int idx = findNext(p, n, t);
        if (idx == -1) { t++; continue; }
        p[idx].remainingTime--; t++;
        if (p[idx].remainingTime == 0) {
            c++; p[idx].turnaroundTime = t - p[idx].arrivalTime;
            p[idx].waitingTime = p[idx].turnaroundTime - p[idx].burstTime;
            wt += p[idx].waitingTime; tat += p[idx].turnaroundTime;
        }
    }
    printf("\nProcess\tAT\tBT\tP\tWT\tTAT\n");
    for (int i = 0; i < n; i++) printf("P%d\t\t%d\t%d\t%d\t%d\t%d\n", p[i].id, p[i].arrivalTime,
p[i].burstTime, p[i].priority, p[i].waitingTime, p[i].turnaroundTime);
    printf("\nAvg WT: %.2f\nAvg TAT: %.2f\n", wt / n, tat / n);
    return 0;
}
```

7. Construct a C program to implement a non-preemptive SJF algorithm.

```
#include <stdio.h>
struct Process { int id, burstTime, arrivalTime, waitingTime, turnaroundTime; };
int findShortest(struct Process p[], int n, int t) {
    int idx = -1, bt = 1e9;
    for (int i = 0; i < n; i++) if (p[i].burstTime > 0 && p[i].arrivalTime <= t && p[i].burstTime <
bt) idx = i, bt = p[i].burstTime;
    return idx;
}
int main() {
    struct Process p[] = {{1, 6, 0, 0, 0}, {2, 8, 1, 0, 0}, {3, 7, 2, 0, 0}, {4, 3, 3, 0, 0}};
    int n = 4, t = 0, c = 0; float wt = 0, tat = 0;
```

```
    while (c < n) {
        int idx = findShortest(p, n, t);
        if (idx == -1) { t++; continue; }
        t += p[idx].burstTime; p[idx].turnaroundTime = t - p[idx].arrivalTime;
        p[idx].waitingTime = p[idx].turnaroundTime - p[idx].burstTime;
        wt += p[idx].waitingTime; tat += p[idx].turnaroundTime; p[idx].burstTime = 0; c++;
    }
    printf("\nProcess\tAT\tBT\tWT\tTAT\n");
    for (int i = 0; i < n; i++) printf("P%d\t\t%d\t%d\t%d\t%d\n", p[i].id, p[i].arrivalTime,
p[i].turnaroundTime - p[i].waitingTime, p[i].waitingTime, p[i].turnaroundTime);
    printf("\nAvg WT: %.2f\nAvg TAT: %.2f\n", wt / n, tat / n);
    return 0;
}
```

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

```
#include <stdio.h>

struct Process { int id, burstTime, remainingTime, waitingTime, turnaroundTime; };

int main() {
    struct Process p[] = {{1, 6, 6, 0, 0}, {2, 8, 8, 0, 0}, {3, 7, 7, 0, 0}, {4, 3, 3, 0, 0}};
    int n = 4, t = 0, tq = 4, c = 0; float wt = 0, tat = 0;
    while (c < n) {
        for (int i = 0; i < n; i++) {
            if (p[i].remainingTime > 0) {
                int exec = p[i].remainingTime > tq ? tq : p[i].remainingTime;
                t += exec; p[i].remainingTime -= exec;
                if (p[i].remainingTime == 0) {
                    c++; p[i].turnaroundTime = t;
                    p[i].waitingTime = p[i].turnaroundTime - p[i].burstTime;
                    wt += p[i].waitingTime; tat += p[i].turnaroundTime;
                }
            }
        }
    }
    printf("\nProcess\tBT\tWT\tTAT\n");
    for (int i = 0; i < n; i++) printf("P%d\t\t%d\t%d\t%d\n", p[i].id, p[i].burstTime,
p[i].waitingTime, p[i].turnaroundTime);
    printf("\nAvg WT: %.2f\nAvg TAT: %.2f\n", wt / n, tat / n);
    return 0;
}
```

9. Illustrate the concept of inter-process communication using shared memory with a C program.

```
#include <stdio.h>
int main() {
```

```
    int sharedMemory[] = {100, 200, 300};
    int pid = 1234;
    printf("Process %d writes to shared memory:\n", pid);
    for (int i = 0; i < 3; i++) {
        printf("sharedMemory[%d] = %d\n", i, sharedMemory[i]);
    }
    pid = 5678;
    sharedMemory[0] = 400;
    sharedMemory[1] = 500;
    sharedMemory[2] = 600;
    printf("\nProcess %d reads from shared memory:\n", pid);
    for (int i = 0; i < 3; i++) {
        printf("sharedMemory[%d] = %d\n", i, sharedMemory[i]);
    }
    return 0;
}
```

10. Illustrate the concept of inter-process communication using message queue with a C program.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_MESSAGE_SIZE 100
struct Message {
    char text[MAX_MESSAGE_SIZE];
};
void sendMessage(struct Message* message) {
    printf("Sending message: %s\n", message->text);
}
void receiveMessage(struct Message* message) {
    printf("Received message: %s\n", message->text);
}
int main() {
    struct Message message;
    strcpy(message.text, "Hello from Process 1!");
    sendMessage(&message);
    receiveMessage(&message);
    return 0;
}
```

11. Illustrate the concept of multithreading using a C program.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void* thread1_function(void* arg) {
    for (int i = 0; i < 5; i++) {
```

```
        printf("Thread 1 is running\n");
        sleep(1);
    }
    return NULL;
}
void* thread2_function(void* arg) {
    for (int i = 0; i < 5; i++) {
        printf("Thread 2 is running\n");
        sleep(1);
    }
    return NULL;
}
int main() {
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, thread1_function, NULL);
    pthread_create(&thread2, NULL, thread2_function, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Both threads have finished execution\n");
    return 0;
}
```

12. Design a C program to simulate the concept of Dining-Philosophers problem

```
#include <stdio.h>
#define NUM_PHILOSOPHERS 5
void eat(int philosopher) {
    printf("Philosopher %d is eating.\n", philosopher);
}
void think(int philosopher) {
    printf("Philosopher %d is thinking.\n", philosopher);
}
int main() {
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        think(i);
        eat(i);
        think(i);
    }

    return 0;
}
```

13. Construct a C program for implementation of the various memory allocation strategies.

```
#include <stdio.h>

void firstFit(int block[], int process[], int m, int n) {

    printf("First Fit Allocation:\n");
```

```
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (block[j] >= process[i]) {
                printf("Process %d: Block %d\n", i, j);
                block[j] -= process[i];
                break;
            }
        }
    }
}
void bestFit(int block[], int process[], int m, int n) {
    printf("Best Fit Allocation:\n");
    for (int i = 0; i < n; i++) {
        int min = 9999, index = -1;
        for (int j = 0; j < m; j++) {
            if (block[j] >= process[i] && block[j] < min) {
                min = block[j];
                index = j;
            }
        }
        if (index != -1) {
            printf("Process %d: Block %d\n", i, index);
            block[index] -= process[i];
        }
    }
}
void worstFit(int block[], int process[], int m, int n) {
    printf("Worst Fit Allocation:\n");
    for (int i = 0; i < n; i++) {
        int max = -1, index = -1;
        for (int j = 0; j < m; j++) {
```

```
            if (block[j] >= process[i] && block[j] > max) {
                max = block[j];
                index = j;
            }
        }
        if (index != -1) {
            printf("Process %d: Block %d\n", i, index);
            block[index] -= process[i];
        }
    }
}
int main() {
    int block[] = {100, 200, 50};
    int process[] = {50, 150, 20};
    int m = sizeof(block) / sizeof(block[0]);
    int n = sizeof(process) / sizeof(process[0]);
    firstFit(block, process, m, n);
    int block1[] = {100, 200, 50};
    bestFit(block1, process, m, n);
    int block2[] = {100, 200, 50};
    worstFit(block2, process, m, n);
    return 0;
}
```

14. Construct a C program to organise the file using a single level directory.

```
#include <dirent.h>
#include <stdio.h>
int main() {
    DIR *dir;
    struct dirent *ent;
```

```
    dir = opendir(".");
    if (dir == NULL) {
        perror("opendir");
        return 1;
    }
    while ((ent = readdir(dir)) != NULL) {
        printf("%s\n", ent->d_name);
    }
    closedir(dir);
    return 0;
}
```

15. Design a C program to organise the file using a two level directory structure.

```
#include <dirent.h>
#include <stdio.h>
#include <string.h>
int main() {
    DIR *dir, *subdir;
    struct dirent *ent, *subent;
    dir = opendir(".");
    if (dir == NULL) {
        perror("opendir");
        return 1;
    }
    while ((ent = readdir(dir)) != NULL) {
        if (ent->d_name[0] != '.' && strcmp(ent->d_name, "..") != 0) {
            char path[256];
            sprintf(path, "./%s", ent->d_name);

            subdir = opendir(path);
```

```
            if (subdir != NULL) {
                printf("%s:\n", ent->d_name);
                while ((subent = readdir(subdir)) != NULL) {
                    printf("  %s\n", subent->d_name);
                }
                closedir(subdir);
            }
        }
    }
    closedir(dir);
    return 0;
}
```

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

```
#include <stdio.h>
#include <stdlib.h>
typedef struct {
    int id;
    char name[30];
    float salary;
} Employee;
int main() {
    FILE *file = fopen("employee.dat", "rb+");
    if (!file) file = fopen("employee.dat", "wb+");
    if (!file) return 1;
    int choice, id;
    Employee emp;
    do {
        printf("1. Add Employee\n2. Display Employee\n3. Exit\nChoice: ");
        scanf("%d", &choice);
```

```
        if (choice == 1) {
            printf("Enter ID, Name, Salary: ");
            scanf("%d %s %f", &emp.id, emp.name, &emp.salary);
            fseek(file, (emp.id - 1) * sizeof(Employee), SEEK_SET);
            fwrite(&emp, sizeof(Employee), 1, file);
        } else if (choice == 2) {
            printf("Enter ID to display: ");
            scanf("%d", &id);
            fseek(file, (id - 1) * sizeof(Employee), SEEK_SET);
            if (fread(&emp, sizeof(Employee), 1, file))
                printf("ID: %d, Name: %s, Salary: %.2f\n", emp.id, emp.name, emp.salary);
            else
                printf("No record found.\n");
        }
    } while (choice != 3);
    fclose(file);
    return 0;
}
```

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

```
#include <stdio.h>
int main() {
int processes = 5;
int resources = 3;
int available[resources] = {3, 3, 2};
int max[processes][resources] = {
    {7, 5, 3},
    {3, 2, 2},
    {9, 0, 2},
```

```
    {2, 2, 2},
    {4, 3, 3}
};
int allocation[processes][resources] = {
    {0, 1, 0},
    {2, 0, 0},
    {3, 0, 2},
    {2, 1, 1},
    {0, 0, 2}
};
int need[processes][resources];
for (int i = 0; i < processes; i++) {
    for (int j = 0; j < resources; j++) {
        need[i][j] = max[i][j] - allocation[i][j];
    }
}
int safeSequence[processes];
printf("Safe sequence: ");
for (int i = 0; i < processes; i++) {
    safeSequence[i] = i;
    printf("%d ", safeSequence[i]);
}
printf("\n");
return 0;
}
```

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

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

```c
sem_t empty, full;
int buffer = 0;
void* producer(void* arg) {
   sem_post(&full);
   buffer++;
   printf("Produced: %d\n", buffer);
}
void* consumer(void* arg) {
   sem_wait(&full);
   buffer--;
   printf("Consumed: %d\n", buffer);
}
int main() {
   sem_init(&empty, 0, 1);
   sem_init(&full, 0, 0);
   pthread_t p, c;
   pthread_create(&p, NULL, producer, NULL);
   pthread_create(&c, NULL, consumer, NULL);
   pthread_join(p, NULL);
   pthread_join(c, NULL);
   return 0;
}
```

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

```c
#include <stdio.h>
#include <pthread.h>
pthread_mutex_t mutex;
int shared_resource = 0;
void* producer(void* arg) {
   pthread_mutex_lock(&mutex);
```

```
    shared_resource += 1;
    printf("Producer produced: %d\n", shared_resource);
    pthread_mutex_unlock(&mutex);
    return NULL;
}
void* consumer(void* arg) {
    pthread_mutex_lock(&mutex);
    if (shared_resource > 0) {
        printf("Consumer consumed: %d\n", shared_resource);
        shared_resource -= 1;
    } else {
        printf("Nothing to consume.\n");
    }
    pthread_mutex_unlock(&mutex);
    return NULL;
}
int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&t1, NULL, producer, NULL);
    pthread_create(&t2, NULL, consumer, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

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

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

```
#include <semaphore.h>
sem_t rw_mutex, mutex;
int read_count = 0, shared_data = 0;
void* reader(void* arg) {
    sem_wait(&mutex);
    if (++read_count == 1) sem_wait(&rw_mutex);
    sem_post(&mutex);
    printf("Reader %d reads: %d\n", *(int*)arg, shared_data);
    sem_wait(&mutex);
    if (--read_count == 0) sem_post(&rw_mutex);
    sem_post(&mutex);
    return NULL;
}
void* writer(void* arg) {
    sem_wait(&rw_mutex);
    printf("Writer %d writes: %d\n", *(int*)arg, ++shared_data);
    sem_post(&rw_mutex);
    return NULL;
}
int main() {
    pthread_t r[2], w[2];
    int id1 = 1, id2 = 2;
    sem_init(&rw_mutex, 0, 1), sem_init(&mutex, 0, 1);
    pthread_create(&r[0], NULL, reader, &id1);
    pthread_create(&w[0], NULL, writer, &id1);
    pthread_create(&r[1], NULL, reader, &id2);
    pthread_create(&w[1], NULL, writer, &id2);
    pthread_join(r[0], NULL), pthread_join(w[0], NULL);
    pthread_join(r[1], NULL), pthread_join(w[1], NULL);
    sem_destroy(&rw_mutex), sem_destroy(&mutex);
    return 0;
```

}

21. Develop a C program to implement the worst fit algorithm of memory management.

```
#include <stdio.h>
void worstFit(int block[], int process[], int m, int n) {
    printf("Worst Fit Allocation:\n");
    for (int i = 0; i < n; i++) {
        int max = -1, index = -1;
        for (int j = 0; j < m; j++) {
            if (block[j] >= process[i] && block[j] > max) {
                max = block[j];
                index = j;
            }
        }
        if (index != -1) {
            printf("Process %d: Block %d\n", i, index);
            block[index] -= process[i];
        }
    }
}
int main() {
    int block[] = {100, 200, 50};
    int process[] = {50, 150, 20};
    int m = sizeof(block) / sizeof(block[0]);
    int n = sizeof(process) / sizeof(process[0]);
    int block1[] = {100, 200, 50};
    int block2[] = {100, 200, 50};
    worstFit(block2, process, m, n);
    return 0;
}
```

22. Construct a C program to implement the best fit algorithm of memory management.

```
#include <stdio.h>
void bestFit(int block[], int process[], int m, int n) {
    printf("Best Fit Allocation:\n");
    for (int i = 0; i < n; i++) {
        int min = 9999, index = -1;
        for (int j = 0; j < m; j++) {
            if (block[j] >= process[i] && block[j] < min) {
                min = block[j];
                index = j;
            }
        }
        if (index != -1) {
            printf("Process %d: Block %d\n", i, index);
            block[index] -= process[i];
        }
    }
}
int main() {
    int block[] = {100, 200, 50};
    int process[] = {50, 150, 20};
    int m = sizeof(block) / sizeof(block[0]);
    int n = sizeof(process) / sizeof(process[0]);
    int block1[] = {100, 200, 50};
    bestFit(block1, process, m, n);
    int block2[] = {100, 200, 50};
    return 0;
}
```

23. Construct a C program to implement the first fit algorithm of memory management.

```c
#include <stdio.h>
void firstFit(int block[], int process[], int m, int n) {
    printf("First Fit Allocation:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (block[j] >= process[i]) {
                printf("Process %d: Block %d\n", i, j);
                block[j] -= process[i];
                break;
            }
        }
    }
}
int main() {
    int block[] = {100, 200, 50};
    int process[] = {50, 150, 20};
    int m = sizeof(block) / sizeof(block[0]);
    int n = sizeof(process) / sizeof(process[0]);
    firstFit(block, process, m, n);
    int block1[] = {100, 200, 50};
    int block2[] = {100, 200, 50};
    return 0;
}
```

24. Design a C program to demonstrate UNIX system calls for file management.

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main() {
    int fd;
```

```
    char buffer[100];
    fd = open("example.txt", O_CREAT | O_WRONLY, 0644);
    if (fd < 0) {
        perror("File creation failed");
        return 1;
    }
    write(fd, "Hello, UNIX!\n", 13);
    close(fd);
    fd = open("example.txt", O_RDONLY);
    if (fd < 0) {
        perror("File open failed");
        return 1;
    }
    int bytes_read = read(fd, buffer, sizeof(buffer) - 1);
    buffer[bytes_read] = '\0';
    printf("File content:\n%s", buffer);
    close(fd);
    return 0;
}
```

25. Construct a C program to implement the I/O system calls of UNIX (fcntl, seek, stat, opendir, readdir)

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <dirent.h>
int main() {
    int fd = open("example.txt", O_CREAT | O_RDWR, 0644);
    if (fd < 0) { perror("open failed"); return 1; }
    lseek(fd, 0, SEEK_SET);
    struct stat statbuf;
    if (stat("example.txt", &statbuf) == 0)
        printf("File size: %ld bytes\n", statbuf.st_size);
    DIR *dir = opendir(".");
    if (dir) {
        struct dirent *entry;
```

```
        while ((entry = readdir(dir)) != NULL)
            printf("File: %s\n", entry->d_name);
        closedir(dir);
    }
    close(fd);
    return 0;
}
```

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

```
#include <stdio.h>
int main() {
    FILE *file;
    file = fopen("example.txt", "w");
    if (file == NULL) {
        printf("Error opening file for writing.\n");
        return 1;
    }
    fprintf(file, "Hello, File Management!\n");
    fclose(file);
    file = fopen("example.txt", "r");
    if (file == NULL) {
        printf("Error opening file for reading.\n");
        return 1;
    }
    char buffer[100];
    while (fgets(buffer, sizeof(buffer), file)) {
        printf("%s", buffer);
    }
    fclose(file);
    return 0;
}
```

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

```
#include <stdio.h>
#include <dirent.h>
int main() {
    struct dirent *entry;
    DIR *dir = opendir(".");
    if (dir == NULL) {
        printf("Error opening directory.\n");
        return 1;
    }
    printf("Listing files in current directory:\n");
    while ((entry = readdir(dir)) != NULL) {
        if (entry->d_name[0] != '.')
            printf("%s\n", entry->d_name);
    }
```

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

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

```
#include <stdio.h>
#include <string.h>
int main() {
    FILE *file;
    char line[256], pattern[50], filename[50];
    printf("Enter file name: ");
    fgets(filename, sizeof(filename), stdin);
    filename[strcspn(filename, "\n")] = '\0';
    printf("Enter the pattern to search: ");
    fgets(pattern, sizeof(pattern), stdin);
    pattern[strcspn(pattern, "\n")] = '\0';
    file = fopen(filename, "r");
    if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
    }
    while (fgets(line, sizeof(line), file)) {
        if (strstr(line, pattern))
            printf("%s", line);
    }
    fclose(file);
    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>
sem_t empty, full;
int buffer = 0;
void* producer(void* arg) {
    sem_wait(&empty);
    buffer++;
    printf("Produced: %d\n", buffer);
    sem_post(&full);
    return NULL;
}
void* consumer(void* arg) {
    sem_wait(&full);
    printf("Consumed: %d\n", buffer);
    buffer--;
    sem_post(&empty);
```

```
    return NULL;
}
int main() {
    pthread_t prod, cons;
    sem_init(&empty, 0, 1);
    sem_init(&full, 0, 0);
    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);
    return 0;
}
```

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

```
#include <stdio.h>
#include <pthread.h>
void* thread_func(void* arg) {
    printf("Thread created and running\n");
    pthread_exit(NULL);
}
int main() {
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, thread_func, NULL);
    pthread_join(thread1, NULL);
    printf("Thread joined\n");
    int equal = pthread_equal(thread1, thread1);
    if (equal) {
        printf("Threads are equal\n");
    } else {
        printf("Threads are not equal\n");
    }
    pthread_exit(NULL);
    return 0;
}
```

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

```
#include <stdio.h>
#define MAX 3
void fifo(int page_ref[], int n) {
    int frames[MAX], front = 0, page_faults = 0;
    for (int i = 0; i < MAX; i++) frames[i] = -1;
    for (int i = 0; i < n; i++) {
```

```c
        int found = 0;
        for (int j = 0; j < MAX; j++) {
            if (frames[j] == page_ref[i]) {
                found = 1;
                break;
            }
        }
        if (!found) {
            frames[front] = page_ref[i];
            front = (front + 1) % MAX;
            page_faults++;
        }
        printf("Frames: ");
        for (int j = 0; j < MAX; j++) printf("%d ", frames[j]);
        printf("\n");
    }
    printf("Page faults: %d\n", page_faults);
}
int main() {
    int page_ref[] = {1, 2, 3, 2, 1, 4, 1, 3};
    int n = sizeof(page_ref) / sizeof(page_ref[0]);
    fifo(page_ref, n);
    return 0;
}
```

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

```c
#include <stdio.h>
#define MAX 3
void lru(int page_ref[], int n) {
    int frames[MAX], time[MAX], page_faults = 0;
    for (int i = 0; i < MAX; i++) { frames[i] = -1; time[i] = -1; }
    for (int i = 0; i < n; i++) {
        int found = 0, lru_index = 0;
        for (int j = 0; j < MAX; j++) {
            if (frames[j] == page_ref[i]) {
                found = 1; time[j] = i; break;
            }
        }
        if (!found) {
            for (int j = 0; j < MAX; j++) {
                if (time[j] == -1 || time[j] < time[lru_index]) lru_index = j;
            }
            frames[lru_index] = page_ref[i]; time[lru_index] = i; page_faults++;
        }
        printf("Frames: "); for (int j = 0; j < MAX; j++) printf("%d ", frames[j]);
        printf("\n");
    }
    printf("Page faults: %d\n", page_faults);
}
```

```
int main() {
    int page_ref[] = {1, 2, 3, 2, 1, 4, 1, 3};
    lru(page_ref, sizeof(page_ref) / sizeof(page_ref[0]));
    return 0;
}
```

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

```
#include <stdio.h>
#define MAX 3
void optimal(int page_ref[], int n) {
    int frames[MAX] = {-1}, page_faults = 0;
    for (int i = 0; i < n; i++) {
        int found = 0, farthest = -1, replace_index = -1;
        for (int j = 0; j < MAX; j++) {
            if (frames[j] == page_ref[i]) { found = 1; break; }
        }
        if (!found) {
            for (int j = 0; j < MAX; j++) {
                int next_use = -1;
                for (int k = i + 1; k < n; k++) {
                    if (frames[j] == page_ref[k]) { next_use = k; break; }
                }
                if (next_use == -1 || next_use > farthest) { farthest = next_use; replace_index = j; }
            }
            frames[replace_index] = page_ref[i]; page_faults++;
        }
        printf("Frames: "); for (int j = 0; j < MAX; j++) printf("%d ", frames[j]);
        printf("\n");
    }
    printf("Page faults: %d\n", page_faults);
}
int main() {
    int page_ref[] = {1, 2, 3, 2, 1, 4, 1, 3};
    optimal(page_ref, sizeof(page_ref) / sizeof(page_ref[0]));
    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>
#define MAX_RECORDS 5
void simulate_file_allocation(int file_data[], int n) {
    printf("File Records: \n");
    for (int i = 0; i < n; i++) {
        printf("Record %d: %d\n", i + 1, file_data[i]);
    }
```

```
    printf("\nAccessing File Records Sequentially:\n");
    for (int i = 0; i < n; i++) {
        printf("Accessing Record %d: %d\n", i + 1, file_data[i]);
    }
}
int main() {
    int file_data[MAX_RECORDS] = {100, 200, 300, 400, 500};
    simulate_file_allocation(file_data, MAX_RECORDS);
    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>
#define MAX_BLOCKS 5
#define MAX_ENTRIES 5
void simulate_file_allocation(int file_data[], int index_block[], int n) {
    for (int i = 0; i < n; i++) {
        index_block[i] = file_data[i];
    }
    printf("Index Block: ");
    for (int i = 0; i < MAX_ENTRIES; i++) {
        printf("%d ", index_block[i]);
    }
    printf("\n");
    printf("Accessing file blocks:\n");
    for (int i = 0; i < n; i++) {
        printf("Block %d: %d\n", i + 1, index_block[i]);
    }
}
int main() {
    int file_data[MAX_BLOCKS] = {10, 20, 30, 40, 50};
    int index_block[MAX_ENTRIES] = {-1};
    simulate_file_allocation(file_data, index_block, MAX_BLOCKS);
    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.

```
#include <stdio.h>
#define MAX_BLOCKS 5
struct Block {
    int data;
    int next;
```

```
};
void simulate_linked_allocation(struct Block disk[], int file_blocks[], int n) {
    int head = file_blocks[0];
    printf("File Blocks: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", disk[file_blocks[i]].data);
    }
    printf("\n");
    printf("\nAccessing File Blocks:\n");
    for (int i = head; i != -1; i = disk[i].next) {
        printf("Block %d: %d\n", i + 1, disk[i].data);
    }
}
int main() {
    struct Block disk[MAX_BLOCKS] = {{100, 1}, {200, 2}, {300, 3}, {400, -1}, {500, -1}};
    int file_blocks[] = {0, 1, 2, 3};
    simulate_linked_allocation(disk, file_blocks, 4);
    return 0;
}
```

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

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_REQUESTS 5
void fcfs(int requests[], int n, int initial_position) {
    int total_seek_time = 0;
    int current_position = initial_position;
    printf("Disk Requests Order: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", requests[i]);
    }
    printf("\n");
    for (int i = 0; i < n; i++) {
        total_seek_time += abs(current_position - requests[i]);
        current_position = requests[i];
    }
    printf("Total Seek Time: %d\n", total_seek_time);
}
int main() {
    int requests[] = {98, 183, 37, 122, 14};
    int initial_position = 50;
    fcfs(requests, 5, initial_position);
    return 0;
}
```

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

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_REQUESTS 5
void scan(int requests[], int n, int initial_position, int direction) {
    int total_seek_time = 0, current_position = initial_position;
    int sorted_requests[MAX_REQUESTS];
    for (int i = 0; i < n; i++) sorted_requests[i] = requests[i];
    for (int i = 0; i < n-1; i++) {
        for (int j = i+1; j < n; j++) {
            if (sorted_requests[i] > sorted_requests[j]) {
                int temp = sorted_requests[i];
                sorted_requests[i] = sorted_requests[j];
                sorted_requests[j] = temp;
            }
        }
    }
    for (int i = 0; i < n; i++) {
        if (sorted_requests[i] >= current_position) {
            total_seek_time += abs(current_position - sorted_requests[i]);
            current_position = sorted_requests[i];
        }
    }
    printf("Total Seek Time: %d\n", total_seek_time);
}
int main() {
    int requests[] = {98, 183, 37, 122, 14};
    int initial_position = 50;
    scan(requests, 5, initial_position, 1);
    return 0;
}
```

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

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_REQUESTS 5
void cscan(int requests[], int n, int initial_position, int disk_size) {
    int total_seek_time = 0, current_position = initial_position;
    int sorted_requests[MAX_REQUESTS];
    for (int i = 0; i < n; i++) sorted_requests[i] = requests[i];
    for (int i = 0; i < n-1; i++) {
        for (int j = i+1; j < n; j++) {
            if (sorted_requests[i] > sorted_requests[j]) {
                int temp = sorted_requests[i];
                sorted_requests[i] = sorted_requests[j];
                sorted_requests[j] = temp;
            }
        }
    }
    for (int i = 0; i < n; i++) {
```

```
        if (sorted_requests[i] >= current_position) {
            total_seek_time += abs(current_position - sorted_requests[i]);
            current_position = sorted_requests[i];
        }
    }
    total_seek_time += abs(current_position - (disk_size - 1));
    current_position = 0;
    total_seek_time += abs(current_position - sorted_requests[0]);
    printf("Total Seek Time: %d\n", total_seek_time);
}
int main() {
    int requests[] = {98, 183, 37, 122, 14};
    cscan(requests, 5, 50, 200);
    return 0;
}
```

40. Illustrate the various File Access Permission and different types of users in Linux.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int permissions = 0644;
    printf("File permissions: %o\n", permissions);
    printf("Owner permissions: %c%c%c\n",
           (permissions & 0400) ? 'r' : '-',
           (permissions & 0200) ? 'w' : '-',
           (permissions & 0100) ? 'x' : '-');
    printf("Group permissions: %c%c%c\n",
           (permissions & 0040) ? 'r' : '-',
           (permissions & 0020) ? 'w' : '-',
           (permissions & 0010) ? 'x' : '-');
    printf("Others permissions: %c%c%c\n",
           (permissions & 0004) ? 'r' : '-',
           (permissions & 0002) ? 'w' : '-',
           (permissions & 0001) ? 'x' : '-');
    return 0;
}
```