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.

Aim:

To create a new process using fork() and display the process ID (PID) and parent process ID (PPID).

Algorithm:

1. Start the program.
2. Use fork() to create a new process.
3. Use getpid() to get PID.
4. Use getppid() to get PPID.
5. Display PID and PPID for both parent and child processes.
6. End the program.

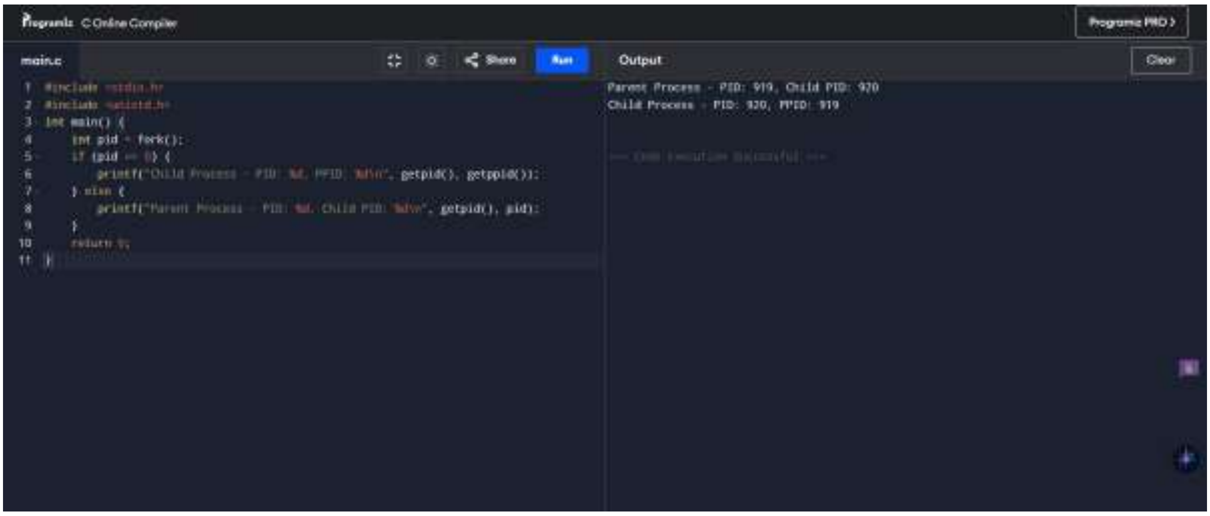
Code:

```
#include <stdio.h>

#include <unistd.h>

int main() {

    int pid = fork();

    if (pid == 0) {

        printf("Child Process - PID: %d, PPID: %d\n", getpid(), getppid());

    } else {

        printf("Parent Process - PID: %d, Child PID: %d\n", getpid(), pid);

    }

    return 0;

}
```

Sample Output:

Parent Process - PID: 1234, Child PID: 1235

Child Process - PID: 1235, PPID: 1234



RESULT:

Process creation and PID display program executed successfully.

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

AIM:

To illustrate file copying using system calls in Linux.

ALGORITHM:

1. Open source file in read-only mode
2. Open/create destination file in write mode
3. Read from source and write to destination until EOF
4. Close both files

CODE:

```
#include <stdio.h>

#include <string.h>

int main() {

    char source[1000], dest[1000];

    printf("Enter source content: ");

    fgets(source, sizeof(source), stdin);

    strcpy(dest, source); // Simulate file copy

    printf("Copied content: %s", dest);

    return 0;
```

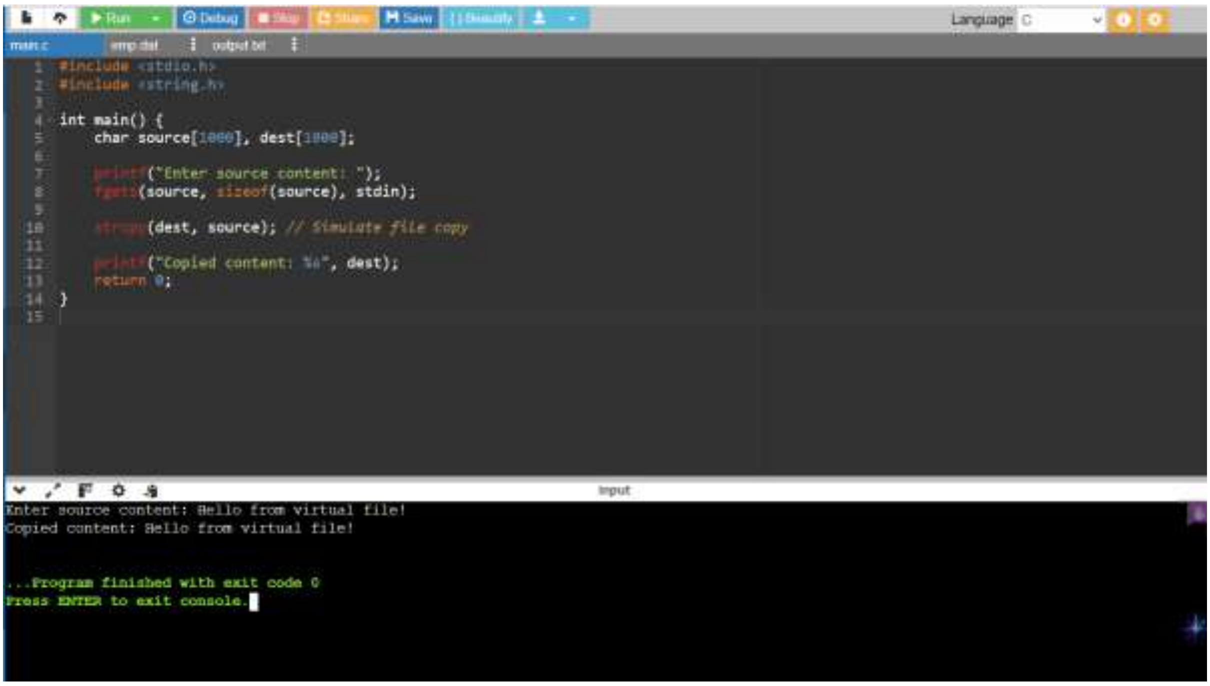
}

SAMPLE INPUT:

Enter source content: Hello from virtual file!

SAMPLE OUTPUT:

Copied content: Hello from virtual file!



RESULT:

File copy using system calls executed successfully.

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.

Aim:

To implement First Come First Served (FCFS) CPU scheduling algorithm in C.

Algorithm:

1. Input number of processes and burst times.
2. Calculate waiting time and turnaround time for each process.
3. Display average waiting time and turnaround time.

Code:

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

```c
int main() {
    int n, i;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    int bt[n], wt[n], tat[n];
    printf("Enter burst times:\n");
    for(i = 0; i < n; i++)
        scanf("%d", &bt[i]);
    wt[0] = 0;
    for(i = 1; i < n; i++)
        wt[i] = wt[i-1] + bt[i-1];
    for(i = 0; i < n; i++)
        tat[i] = wt[i] + bt[i];
    printf("P\tBT\tWT\tTAT\n");
    for(i = 0; i < n; i++)
        printf("%d\t%d\t%d\t%d\n", i+1, bt[i], wt[i], tat[i]);
    return 0;
}
```

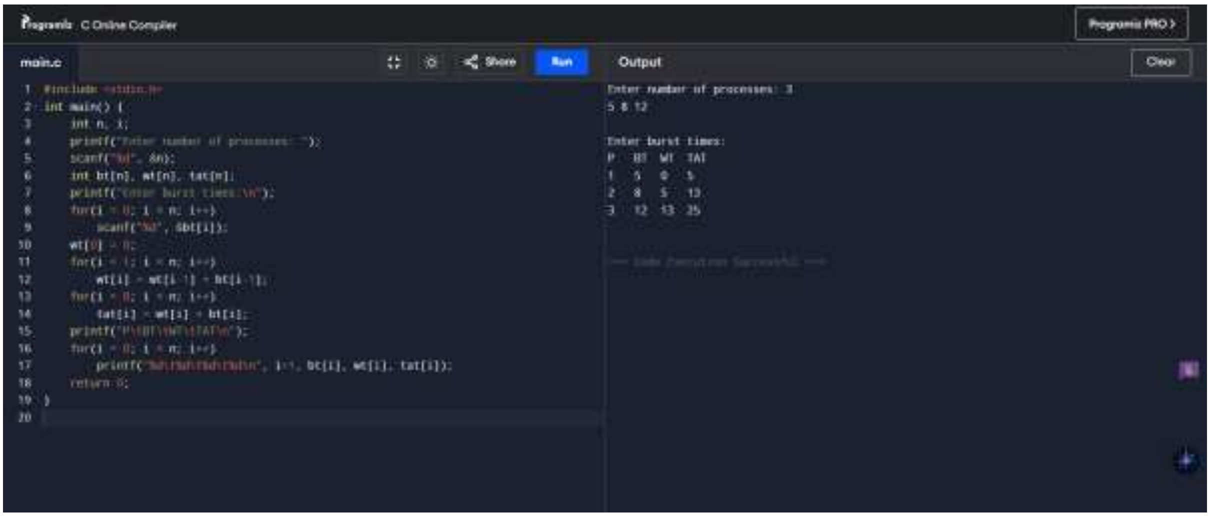
Sample Input:

3

5 8 12

Sample Output:

P	BT	WT	TAT
1	5	0	5
2	8	5	13
3	12	13	25



RESULT:

FCFS CPU scheduling program executed successfully.

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

Aim:

To implement non-preemptive Shortest Job First (SJF) scheduling algorithm.

Algorithm:

1. Input number of processes and burst times
2. Sort processes based on burst times
3. Calculate waiting time and turnaround time.
4. Display results.

Code:

```c
#include <stdio.h>

int main() {

    int n, i, j, temp;

    printf("Enter number of processes: ");

    scanf("%d", &n);

    int bt[n], p[n];

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

        printf("Enter burst time for P%d: ", i+1);

        scanf("%d", &bt[i]);
```

```
        p[i] = i+1;
    }
    for(i = 0; i < n-1; i++)
        for(j = i+1; j < n; j++)
            if(bt[i] > bt[j]) {
                temp = bt[i]; bt[i] = bt[j]; bt[j] = temp;
                temp = p[i]; p[i] = p[j]; p[j] = temp;
            }
    int wt = 0, tat = 0, total_wt = 0, total_tat = 0;
    printf("P\tBT\tWT\tTAT\n");
    for(i = 0; i < n; i++) {
        tat = wt + bt[i];
        printf("P%d\t%d\t%d\t%d\n", p[i], bt[i], wt, tat);
        total_wt += wt;
        total_tat += tat;
        wt = tat;
    }
    return 0;
}
```

SAMPLE INPUT:

Enter number of processes:4

Enter burst time for P1:6

Enter burst time for P2:8

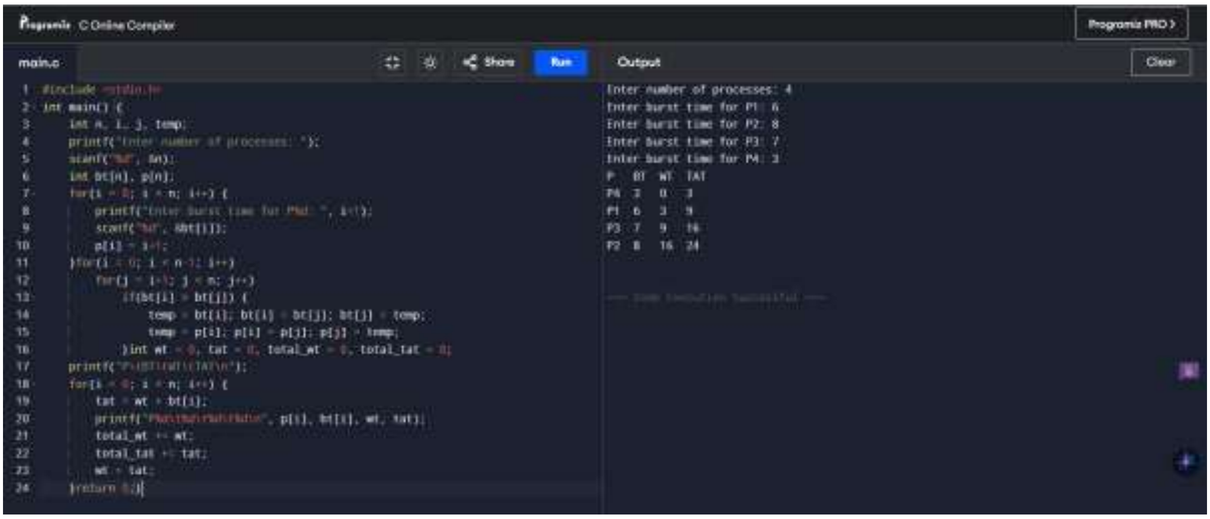
Enter burst time for P3:7

Enter burst time for P4:3

SAMPLE OUTPUT:

P	BT	WT	TAT
P4	3	0	3
P1	6	3	9

P3	7	9	16
P2	8	16	24



RESULT:

SJF scheduling program executed successfully.

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

Aim:

To implement non-preemptive Priority Scheduling in C.

Algorithm:

1. Input processes with burst time and priority.
2. Sort processes based on priority.
3. Calculate waiting time, turnaround time.
4. Display results.

Code:

```
#include <stdio.h>
struct Process {
    int pid, bt, pr;
};
int main() {
    struct Process p[10];
    int n, i, j;
```

```
    printf("No. of processes: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("P%d Burst Priority: ", i + 1);
        p[i].pid = i + 1;
        scanf("%d %d", &p[i].bt, &p[i].pr);
    }   for (i = 0; i < n-1; i++)
        for (j = i+1; j < n; j++)
            if (p[i].pr > p[j].pr) {
                struct Process temp = p[i]; p[i] = p[j]; p[j] = temp;
            }
    int wt = 0, tat;
    printf("\nProcess\tBT\tPriority\tWT\tTAT\n");
    for (i = 0; i < n; i++) {
        tat = wt + p[i].bt;
        printf("P%d\t%d\t%d\t\t%d\t%d\n", p[i].pid, p[i].bt, p[i].pr, wt, tat);
        wt = tat;
    }
}
```

Sample Input:

No. of processes: 3

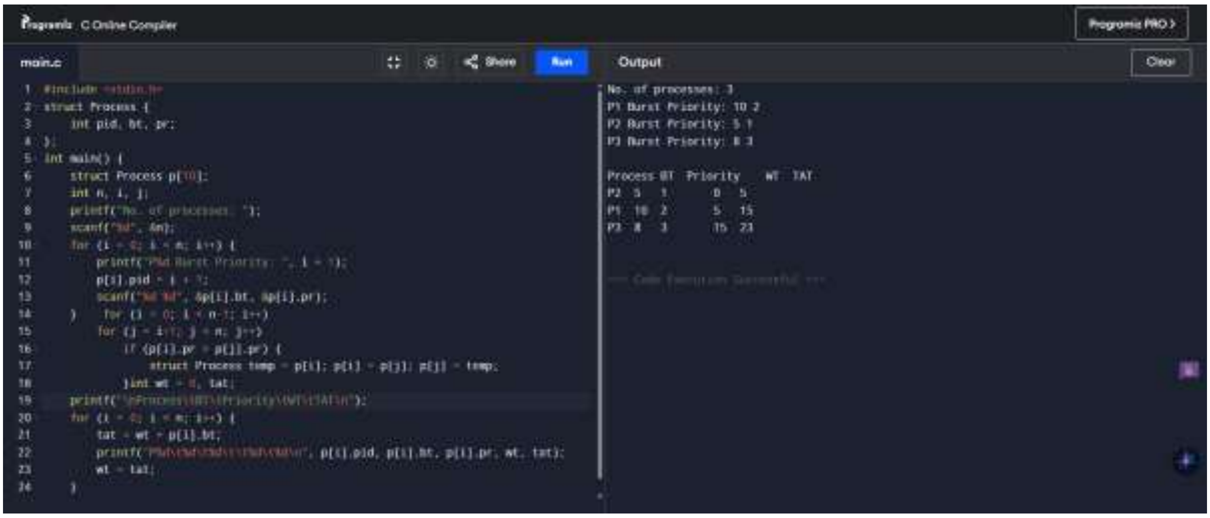
P1 Burst Priority: 10 2

P2 Burst Priority: 5 1

P3 Burst Priority: 8 3

Sample Output:

Process		BT	Priority	WT	TAT
P2	5	1	0	5	
P1	10	2	5	15	
P3	8	3	15	23\	



RESULT:

Priority scheduling (non-preemptive) executed successfully.

6. Construct a C program to implement pre-emptive priority scheduling algorithm.

Aim:

To implement Preemptive Priority Scheduling algorithm in C.

Algorithm:

1. Input processes with arrival time, burst time, and priority.
2. At each unit of time, pick the process with highest priority.
3. If a new process arrives with higher priority, preempt the current process.
4. Calculate Waiting Time (WT) & Turn Around Time (TAT).
5. Display results.

Code:

```
#include <stdio.h>

struct P { int id, bt, pr, rt; } p[10];

int main() {

    int n, done = 0;

    printf("No. of processes: "); scanf("%d", &n);

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

        printf("P%d BT Priority: ", i+1);

        p[i].id = i+1; scanf("%d %d", &p[i].bt, &p[i].pr);

        p[i].rt = p[i].bt;
```

```
    }
    printf("\nExecution: ");
    while (done < n) {
        int pos = -1, min = 999;
        for (int i=0;i<n;i++)
            if (p[i].rt > 0 && p[i].pr < min) min = p[i].pr, pos = i;
        if (pos == -1) break;
        printf("P%d ", p[pos].id);
        p[pos].rt--;
        if (p[pos].rt == 0) done++;
    }
}
```

Sample Input:

No. of processes: 3

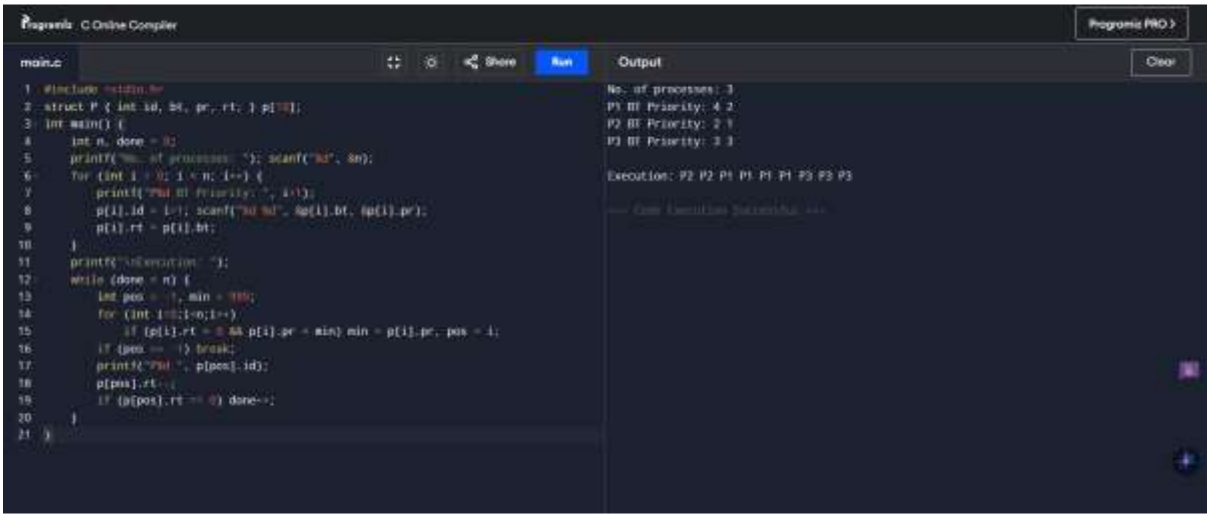
P1 BT Priority: 4 2

P2 BT Priority: 2 1

P3 BT Priority: 3 3

Sample Output:

Execution: P2 P2 P1 P1 P1 P1 P3 P3 P3



RESULT:

Non-preemptive SJF scheduling executed successfully.

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

AIM:

To write a C program to implement Non-Preemptive SJF CPU Scheduling.

Procedure

1. Enter number of processes and their burst times.
2. Sort processes by burst time (smallest first).
3. Calculate Waiting Time (WT):
4. → First WT = 0, next = previous WT + previous BT.
5. Calculate Turnaround Time (TAT) = WT + BT.
6. Display Process | BT | WT | TAT.
7. End.

Code

```
#include <stdio.h>

int main() {

   int n, bt[10], wt[10]={0}, tat[10], i, j, temp;

   printf("No. of processes: "); scanf("%d", &n);

   for (i=0; i<n; i++) { printf("BT of P%d: ", i+1); scanf("%d", &bt[i]); }

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

   for (i=1; i<n; i++) wt[i]=wt[i-1]+bt[i-1];

   printf("\nP\tBT\tWT\tTAT\n");

   for (i=0; i<n; i++) { tat[i]=wt[i]+bt[i]; printf("P%d\t%d\t%d\t%d\n", i+1, bt[i], wt[i],
tat[i]); }

}
```

Sample Input:

No. of processes: 3

BT of P1: 5

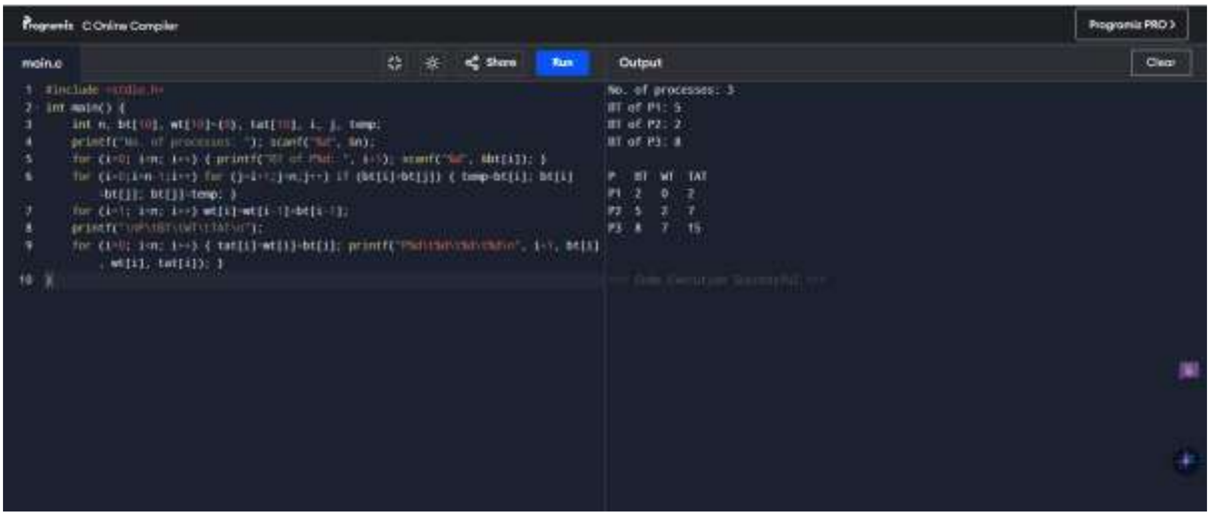
BT of P2: 2

BT of P3: 8

Sample Output:

P BT WT TAT

P1	2	0	2
P2	5	2	7
P3	8	7	15



RESULT:

Non-preemptive SJF scheduling executed successfully.

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

Aim:

To implement Round Robin Scheduling algorithm in C.

Algorithm:

1. Input processes and burst times.
2. Input time quantum.
3. Execute processes for time quantum repeatedly in a circular manner until completion.
4. Calculate WT & TAT.

Code

```
#include <stdio.h>
int main() {
    int n, bt[10], rem[10], tq, time = 0, done;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) scanf("%d", &bt[i]), rem[i] = bt[i];
    scanf("%d", &tq);
    while (1) {
```

```
        done = 1;
        for (int i = 0; i < n; i++)
            if (rem[i] > 0) {
                done = 0;
                if (rem[i] > tq) rem[i] -= tq, time += tq, printf("P%d ", i+1);
                else time += rem[i], rem[i] = 0, printf("P%d ", i+1);
            }
        if (done) break;
    }
    printf("\nTotal Time: %d\n", time);
}
```

Sample Input:

3

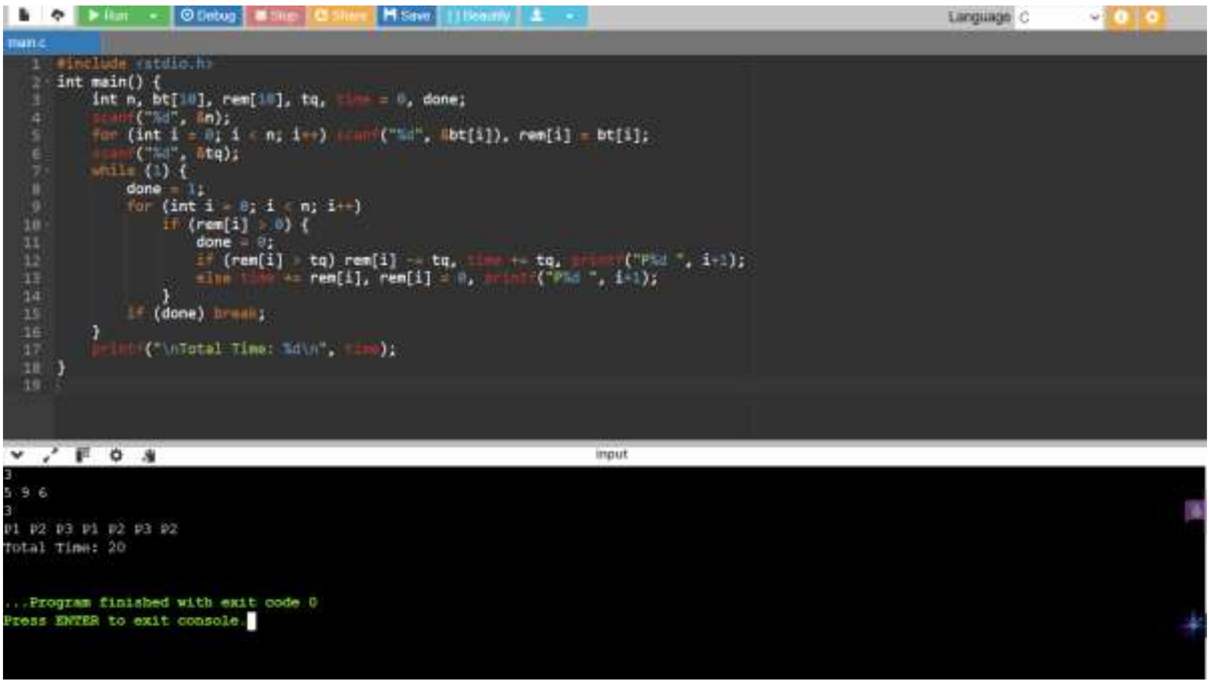
5 9 6

3

Sample Output:

P1 P2 P3 P1 P2 P3 P2

Total Time: 20



RESULT:

Round Robin scheduling program executed successfully.

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

Aim:

To demonstrate IPC using shared memory with producer & consumer processes.

Algorithm:

1. Create a shared memory segment.
2. Attach shared memory to both parent and child processes.
3. Parent writes data; child reads it.
4. Detach & delete shared memory after communication.

Code

```
#include <stdio.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <string.h>

#include <unistd.h>

int main() {

   int shmid = shmget(IPC_PRIVATE, 100, IPC_CREAT | 0666);

   char *str = (char *)shmat(shmid, NULL, 0)

   if (fork() == 0) {

      sleep(3);

      printf("Child reads: %s\n", str);

   } else {

      printf("Parent writes: ");

      fgets(str, 100, stdin);

      wait(NULL);

      shmdt(str);

      shmctl(shmid, IPC_RMID, NULL);
```

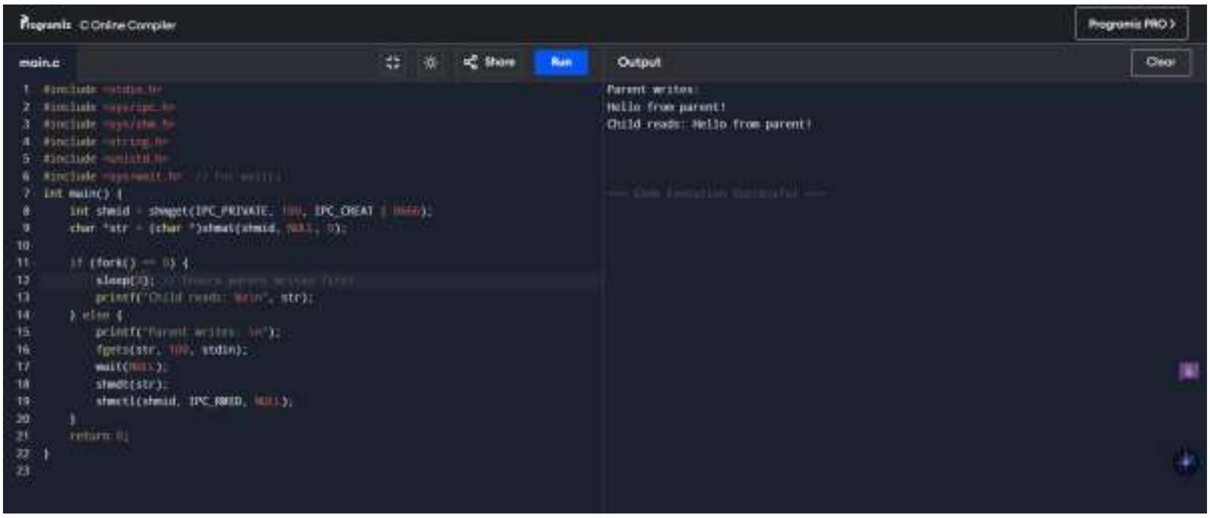
```
    }
    return 0;
}
```

Sample Input

Parent writes: Hello from parent

Sample Output

Child reads: Hello from parent



RESULT:

Shared memory IPC program executed successfully.

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

Aim:

To demonstrate IPC using message queue in C.

Algorithm:

1. Create a message queue.
2. Parent sends message; child receives it.
3. Display received message.
4. Remove message queue after communication.

Code

```
#include <stdio.h>

#include <sys/ipc.h>
```

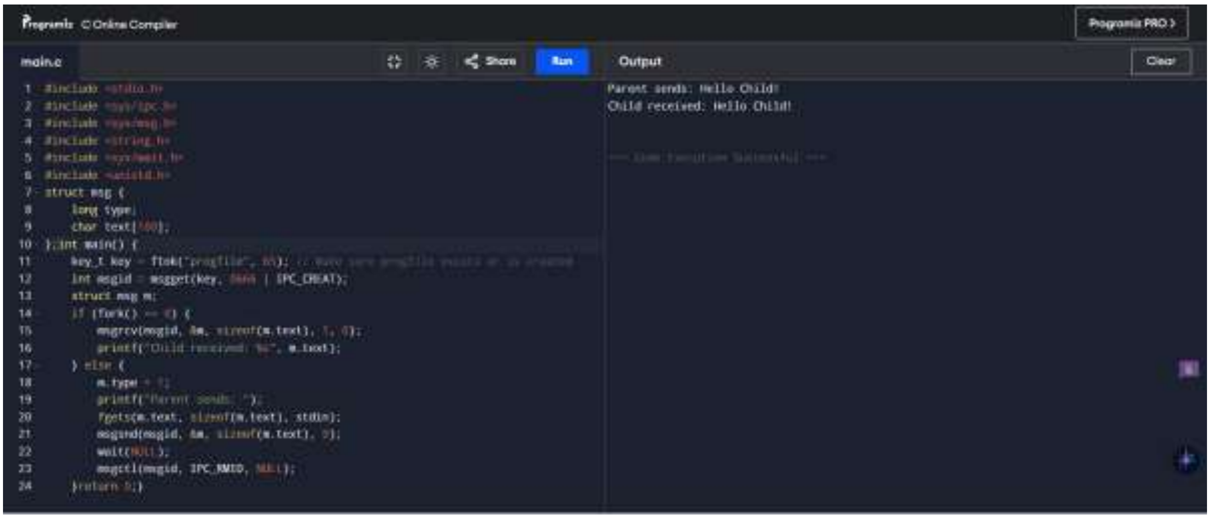
```
#include <sys/msg.h>
#include <string.h>
struct msg {
    long type;
    char text[100];
};
int main() {
    key_t key = ftok("progfile", 65);
    int msgid = msgget(key, 0666 | IPC_CREAT);
    struct msg m;
    if (fork() == 0) {
        msgrcv(msgid, &m, sizeof(m.text), 1, 0);
        printf("Child received: %s\n", m.text);
    } else {
        m.type = 1;
        printf("Parent sends: ");
        fgets(m.text, sizeof(m.text), stdin);
        msgsnd(msgid, &m, sizeof(m.text), 0);
        wait(NULL);
        msgctl(msgid, IPC_RMID, NULL);
    }
    return 0;
}
```

Sample Input

Parent sends: Hello Child!

Sample Output

Child received: Hello Child!



RESULT:

IPC using message queue program executed successfully.

11. Illustrate the concept of multithreading using a C program

Aim:

To illustrate the concept of multithreading using a C program.

Procedure:

1. Include necessary header files.
2. Create a thread function to print a message.
3. Use pthread_create() to create threads.
4. Use pthread_join() to wait for thread completion.

Code:

```
#include <stdio.h>

#include <pthread.h>

void *threadFunc(void *arg) {

   printf("Hello from thread!\n");

   return NULL;

}

int main() {

   pthread_t thread;

   pthread_create(&thread, NULL, threadFunc, NULL);

   pthread_join(thread, NULL);
```

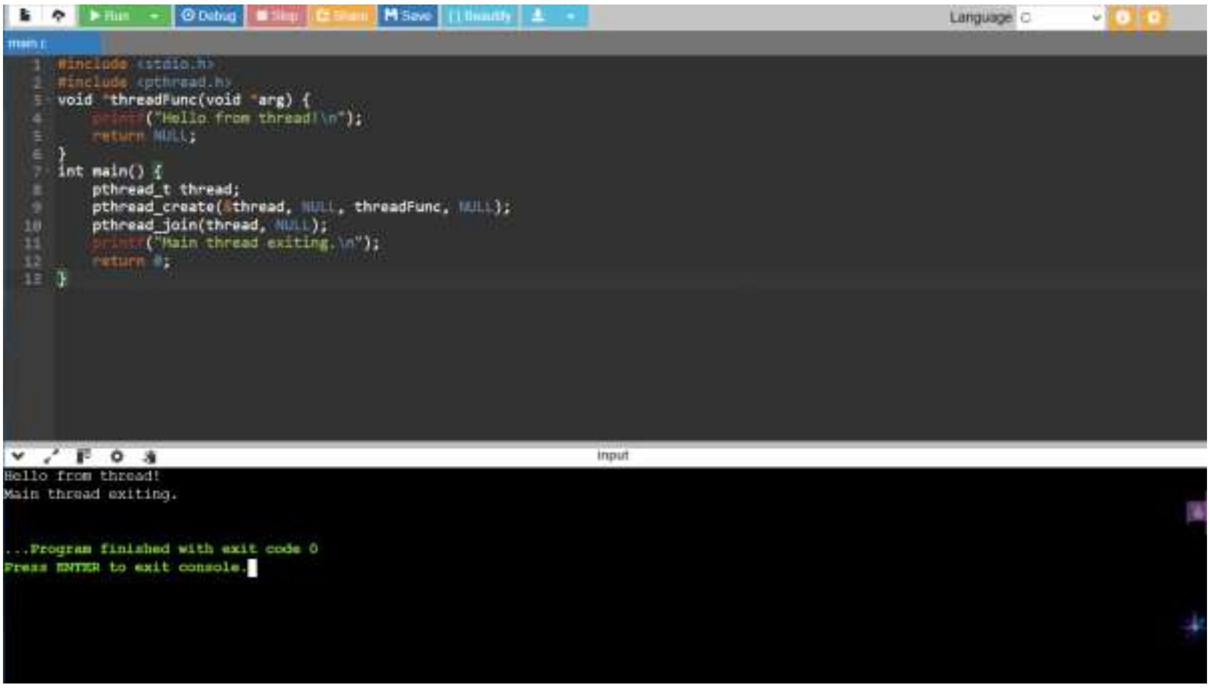
```
    printf("Main thread exiting.\n");
    return 0;
}
```

Sample Input: None

Sample Output:

Hello from thread!

Main thread exiting.



RESULT:

Multithreading program executed successfully.

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

Aim:

To simulate the Dining Philosophers Problem using C and semaphores.

Procedure:

1. Initialize semaphores for forks.
2. Each philosopher thinks, picks two forks (semaphores), eats, and then releases them.
3. Use threads to represent philosophers.

Code:

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

```c
#include <pthread.h>
#include <semaphore.h>
#define N 5
sem_t forks[N];
void *philosopher(void *num) {
    int id = *(int *)num;
    printf("Philosopher %d is thinking.\n", id);
    sem_wait(&forks[id]);
    sem_wait(&forks[(id + 1) % N]);
    printf("Philosopher %d is eating.\n", id);
    sem_post(&forks[id]);
    sem_post(&forks[(id + 1) % N]);
    printf("Philosopher %d finished eating.\n", id);
    return NULL;
}
int main() {
    pthread_t tid[N];
    int ids[N];
    for (int i = 0; i < N; i++)
        sem_init(&forks[i], 0, 1);
    for (int i = 0; i < N; i++) {
        ids[i] = i;
        pthread_create(&tid[i], NULL, philosopher, &ids[i]);
    }
    for (int i = 0; i < N; i++)
        pthread_join(tid[i], NULL);
    return 0;
}
```

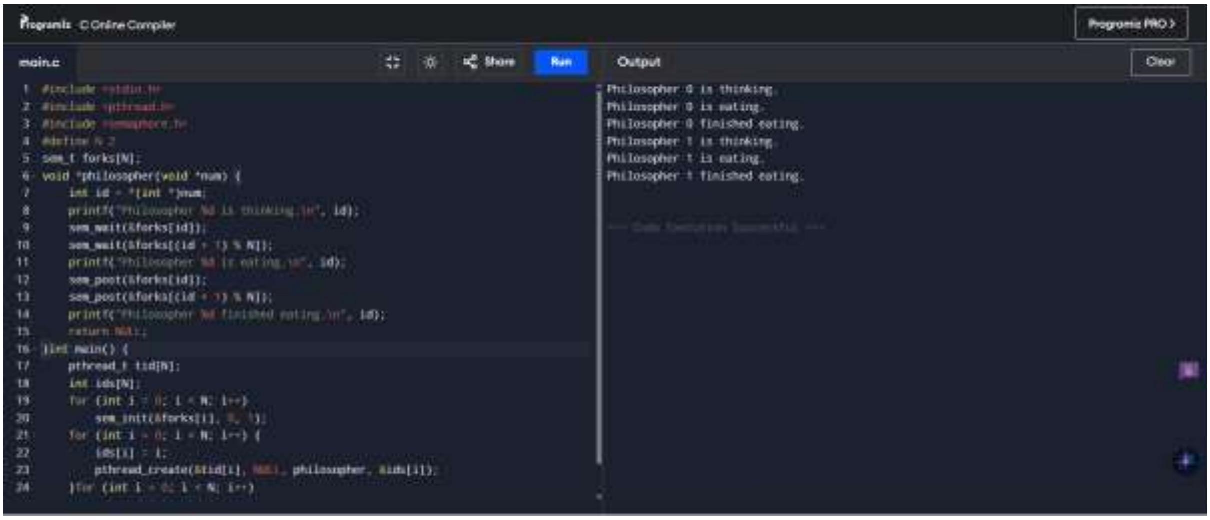
Sample Input: None

Sample Output

Philosopher 0 is thinking.

Philosopher 0 is eating.

Philosopher 0 finished eating.



RESULT:

Dining Philosophers problem program executed successfully.

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

Aim:

To implement various memory allocation strategies (First Fit, Best Fit, Worst Fit).

Procedure:

1. Define block and process arrays.
2. For each strategy, traverse blocks and allocate if suitable.
3. Print allocation status.

Code

```
#include <stdio.h>

#define SIZE 5

void firstFit(int blockSize[], int m, int processSize[], 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 (blockSize[j] >= processSize[i]) {
                allocation[i] = j;
                blockSize[j] -= processSize[i];
                break;
            }
    for (int i = 0; i < n; i++) {
        printf("Process %d -> ", i+1);
        if (allocation[i] != -1)
            printf("Block %d\n", allocation[i]+1);
        else
            printf("Not Allocated\n");
    }
}
int main() {
    int blockSize[SIZE] = {100, 500, 200, 300, 600};
    int processSize[4] = {212, 417, 112, 426};
    firstFit(blockSize, SIZE, processSize, 4);
    return 0;
}
```

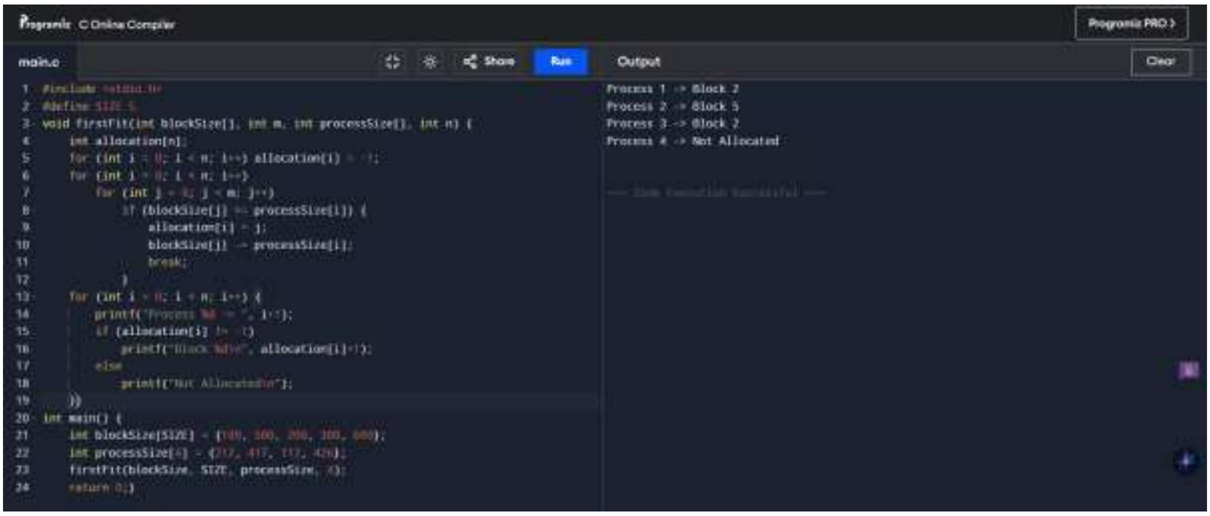
Sample Output:

Process 1 -> Block 2

Process 2 -> Block 5

Process 3 -> Block 2

Process 4 -> Not Allocated



RESULT:

Memory allocation strategies program executed successfully.

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

Aim:

To organize the file using a single-level directory.

Procedure:

1. Use an array of filenames.
2. Provide menu to create, delete, search files.

Code:

```
#include <stdio.h>

#include <string.h

struct { char fname[20]; } dir[10];

int main() {

   int n = 0, ch; char name[20];

   while (1) {

      printf("\n1.Create 2.Delete 3.Search 4.Exit: ");

      scanf("%d", &ch);

      if (ch == 1) scanf("%s", dir[n++].fname);

      else if (ch == 2) {

         scanf("%s", name);
```

```
            for (int i = 0; i < n; i++)
                if (strcmp(name, dir[i].fname) == 0) strcpy(dir[i].fname, "deleted");
        }
        else if (ch == 3) {
            scanf("%s", name);
            for (int i = 0; i < n; i++)
                if (strcmp(name, dir[i].fname) == 0) printf("Found\n");
        }
        else break;
    }
}
```

Sample Input:

1

file1

1

file2

3

file1

2

file2

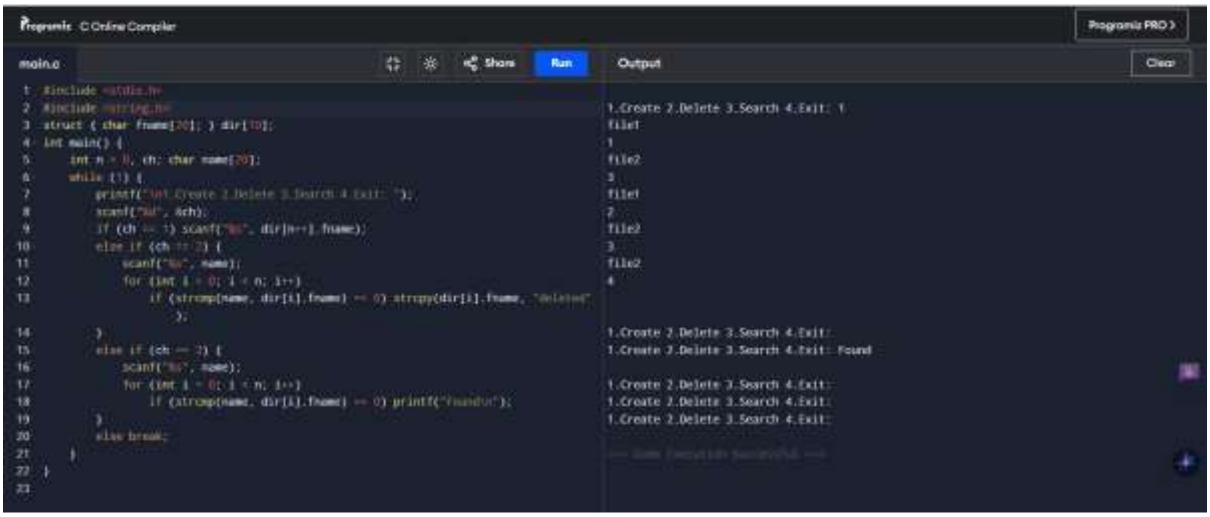
3

file2

4

Sample Output:

Found



RESULT:

Single-level directory structure program executed successfully.

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

Aim: To organize the file using a two-level directory structure.

Procedure:

1. Create multiple user directories.
2. Allow file creation inside user directories.

Code:

```
#include <stdio.h>

#include <string.h>

struct { char d[10], f[10][10]; int fc; } dir[10];

int main() {

    int dc = 0, ch; char d[10];

    while (1) {

        printf("\n1.Dir 2.File 3.List 4.Exit: ");

        scanf("%d", &ch);

        if (ch == 1) { printf("Dir: "); scanf("%s", dir[dc].d); dir[dc++].fc = 0; }

        else if (ch == 2) {
```

```
        printf("Dir: "); scanf("%s", d);
        for (int i=0;i<dc;i++)
            if (!strcmp(d, dir[i].d))
                scanf("%s", dir[i].f[dir[i].fc++]);
    }
    else if (ch == 3) {
        printf("Dir: "); scanf("%s", d);
        for (int i=0;i<dc;i++)
            if (!strcmp(d, dir[i].d))
                for (int j=0;j<dir[i].fc;j++) printf("%s\n", dir[i].f[j]);
    }
    else break;
  }
}
```

Sample Input:

1

Dir: proj

2

Dir: proj

file1

2

Dir: proj

file2

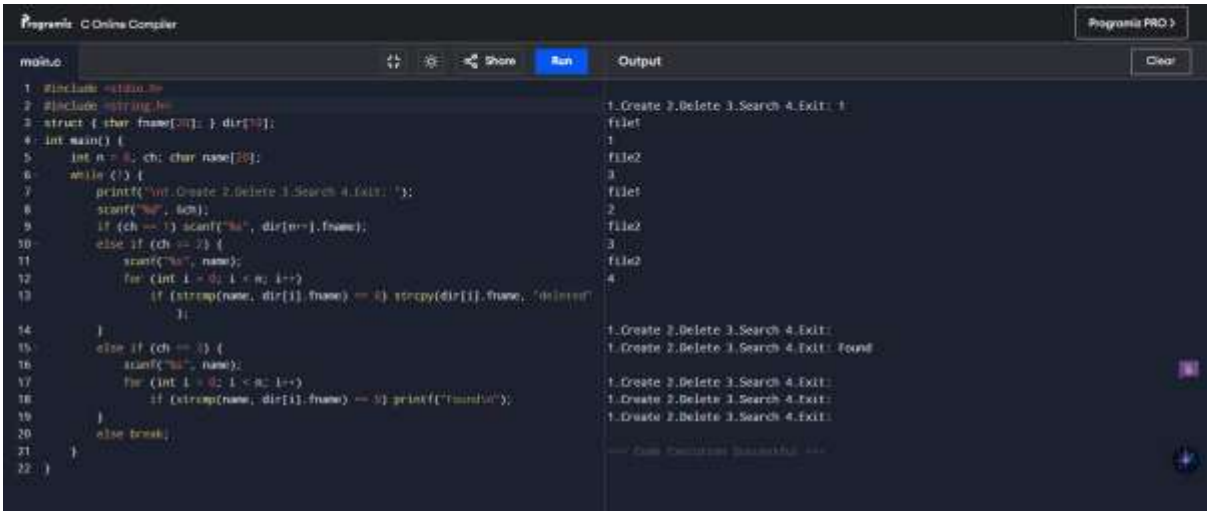
3

Dir: proj

Sample Output:

file1

file2



RESULT:

Two-level directory structure program executed successfully.

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

Aim:

To implement random access file processing employee details.

Procedure:

1. Open file in binary mode.
2. Write employee details into the file.
3. Use fseek() for random access.

Code:

```
#include <stdio.h>

struct Emp { int id; char name[20]; float sal; };

int main() {

    struct Emp e;

    FILE *f = fopen("emp.dat", "wb+");

    for (int i=0; i<2; i++) {

        printf("ID Name Salary: ");

        scanf("%d %s %f", &e.id, e.name, &e.sal);

        fwrite(&e, sizeof(e), 1, f);

    }
```

```
    fseek(f, sizeof(e), SEEK_SET);
    fread(&e, sizeof(e), 1, f);
    printf("Record 2 -> ID:%d Name:%s Salary:%.2f\n", e.id, e.name, e.sal);
    fclose(f);
}
```

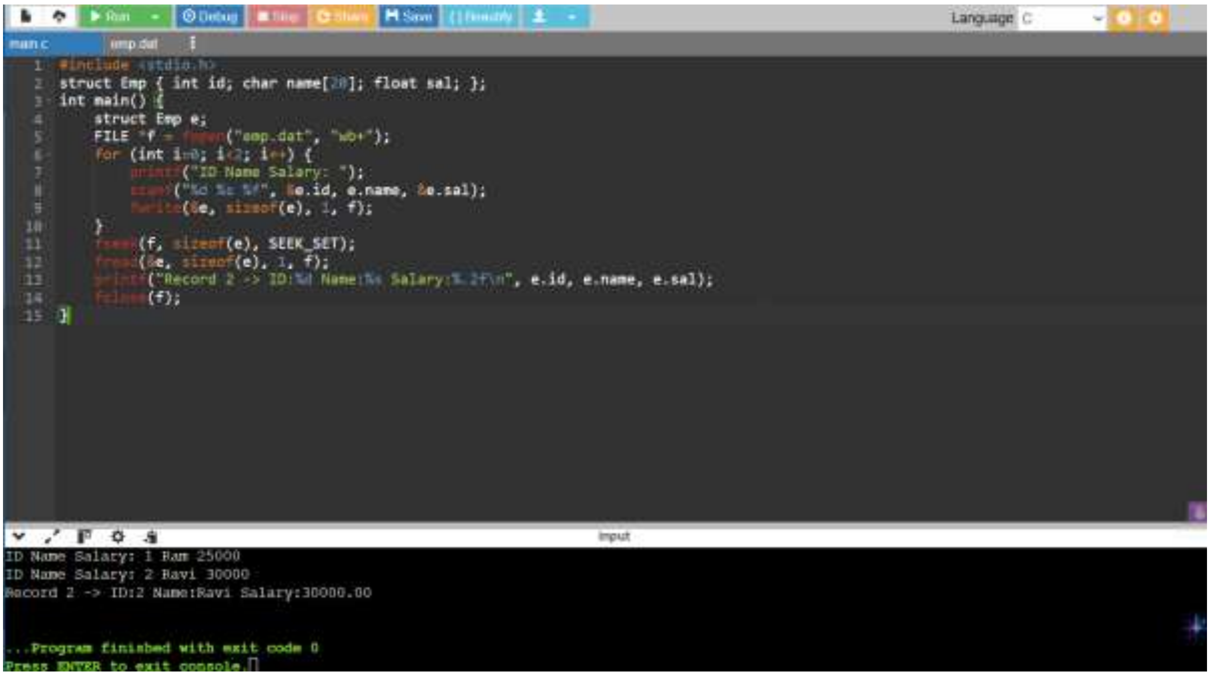
Sample Input:

ID Name Salary: 1 Ram 25000

ID Name Salary: 2 Ravi 30000

Sample Output:

Record 2 -> ID:2 Name:Ravi Salary:30000.00



RESULT:

Random access file program for employee records executed successfully.

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

Aim:

To illustrate the deadlock avoidance concept by simulating Banker's algorithm using C.

Procedure:

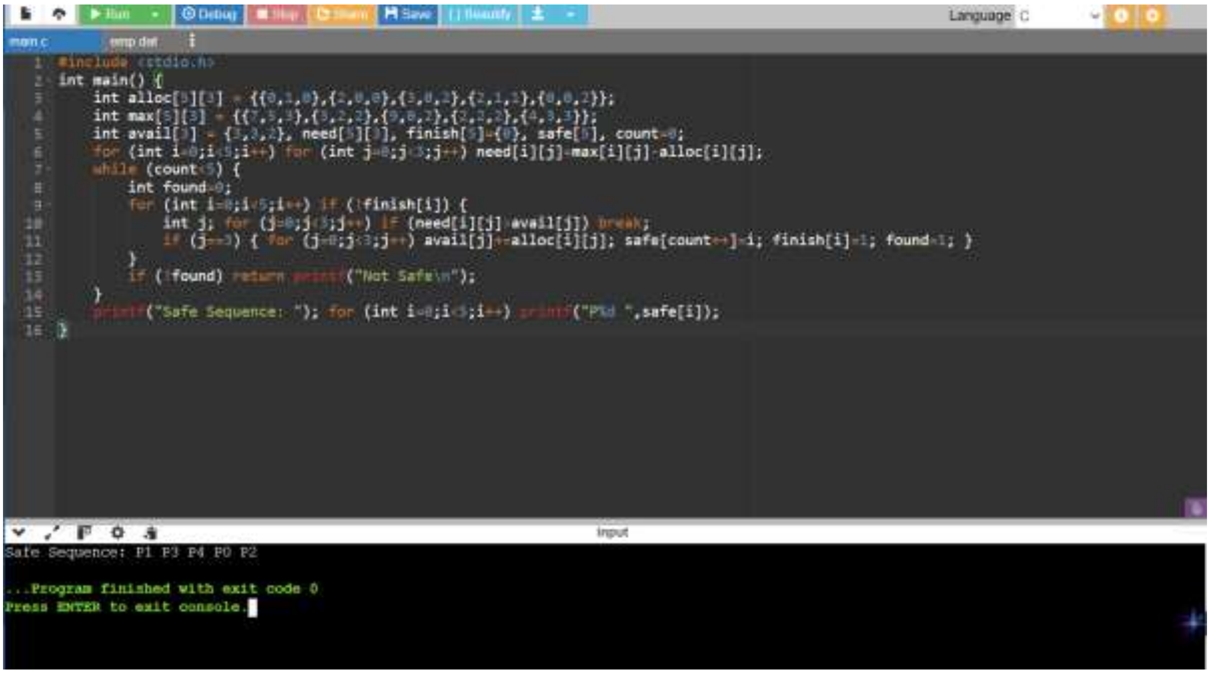
1. Input number of processes and resources.
2. Input Allocation, Maximum, and Available matrices.
3. Calculate Need matrix.
4. Find Safe Sequence using Banker's Algorithm.
5. Display result.

Code:

```
#include <stdio.h>

int main() {

    int alloc[5][3] = {{0,1,0},{2,0,0},{3,0,2},{2,1,1},{0,0,2}};

    int max[5][3] = {{7,5,3},{3,2,2},{9,0,2},{2,2,2},{4,3,3}};

    int avail[3] = {3,3,2}, need[5][3], finish[5]={0}, safe[5], count=0;

    for (int i=0;i<5;i++) for (int j=0;j<3;j++) need[i][j]=max[i][j]-alloc[i][j];

    while (count<5) {

        int found=0;

        for (int i=0;i<5;i++) if (!finish[i]) {

            int j; for (j=0;j<3;j++) if (need[i][j]>avail[j]) break;

            if (j==3) { for (j=0;j<3;j++) avail[j]+=alloc[i][j]; safe[count++]=i; finish[i]=1;
found=1; }

        }

        if (!found) return printf("Not Safe\n");

    }

    printf("Safe Sequence: "); for (int i=0;i<5;i++) printf("P%d ",safe[i]);

}
```

Sample Output:

Safe Sequence: P1 P3 P4 P0 P2



RESULT:

Banker's algorithm for deadlock avoidance executed successfully.

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

Aim:

To simulate producer-consumer problem using semaphores.

Procedure:

Initialize empty, full semaphores and mutex.

Producer inserts items into the buffer.

Consumer removes items from the buffer.

Use semaphores to ensure mutual exclusion and synchronization.

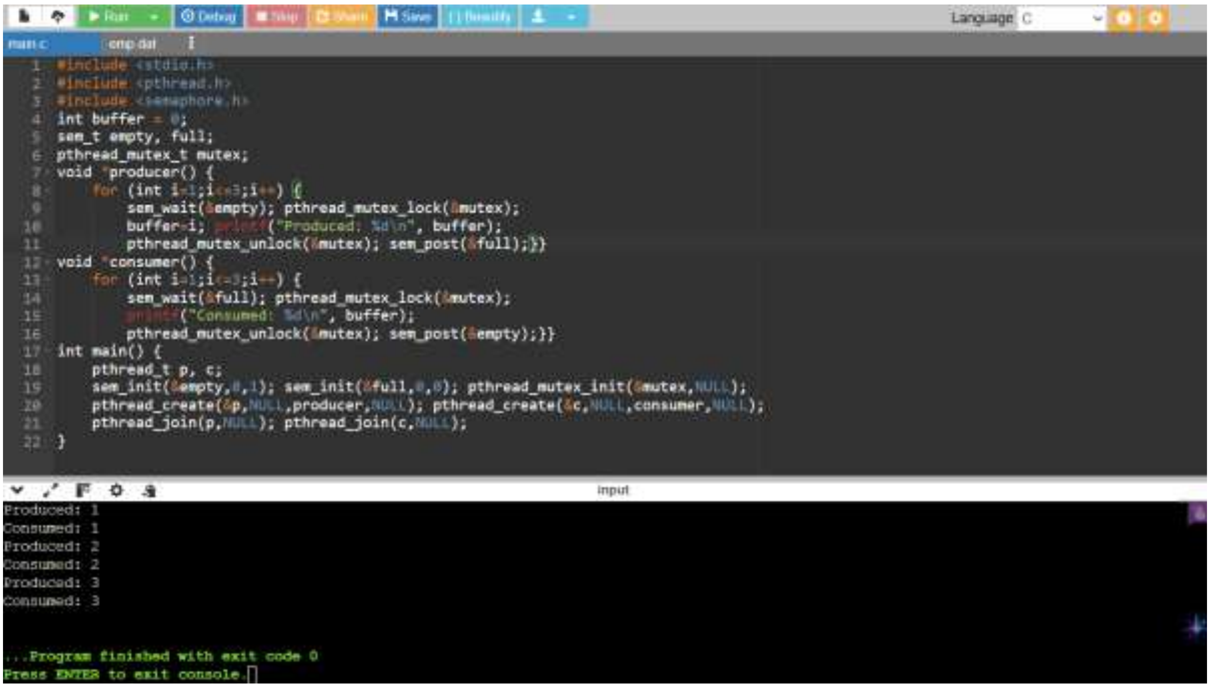
Code:

```
#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

int buffer = 0;

sem_t empty, full;

pthread_mutex_t mutex;
```

```c
void *producer() {
    for (int i=1;i<=3;i++) {
        sem_wait(&empty); pthread_mutex_lock(&mutex);
        buffer=i; printf("Produced: %d\n", buffer);
        pthread_mutex_unlock(&mutex); sem_post(&full);
    }
}
void *consumer() {
    for (int i=1;i<=3;i++) {
        sem_wait(&full); pthread_mutex_lock(&mutex);
        printf("Consumed: %d\n", buffer);
        pthread_mutex_unlock(&mutex); sem_post(&empty);
    }
}
int main() {
    pthread_t p, c;
    sem_init(&empty,0,1); sem_init(&full,0,0); pthread_mutex_init(&mutex,NULL);
    pthread_create(&p,NULL,producer,NULL); pthread_create(&c,NULL,consumer,NULL);
    pthread_join(p,NULL); pthread_join(c,NULL);
}
```

Sample Output:

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
```



RRESULT:

Producer-consumer problem using semaphores executed successfully.

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

Aim:

To implement process synchronization using mutex locks in C.

Procedure:

1. Create threads performing critical operations.
2. Protect critical section using pthread_mutex_lock() and pthread_mutex_unlock().
3. Execute critical section mutually exclusively.

Code:

```
#include <stdio.h>

#include <pthread.h>

int counter=0;

pthread_mutex_t lock;

void *inc() {

   for (int i=0;i<3;i++) {

      pthread_mutex_lock(&lock);

      counter++; printf("Counter: %d\n", counter);
```

```
        pthread_mutex_unlock(&lock);
    }
}
int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&lock,NULL);
    pthread_create(&t1,NULL,inc,NULL); pthread_create(&t2,NULL,inc,NULL);
    pthread_join(t1,NULL); pthread_join(t2,NULL);
}
```

Sample Output

Counter: 1

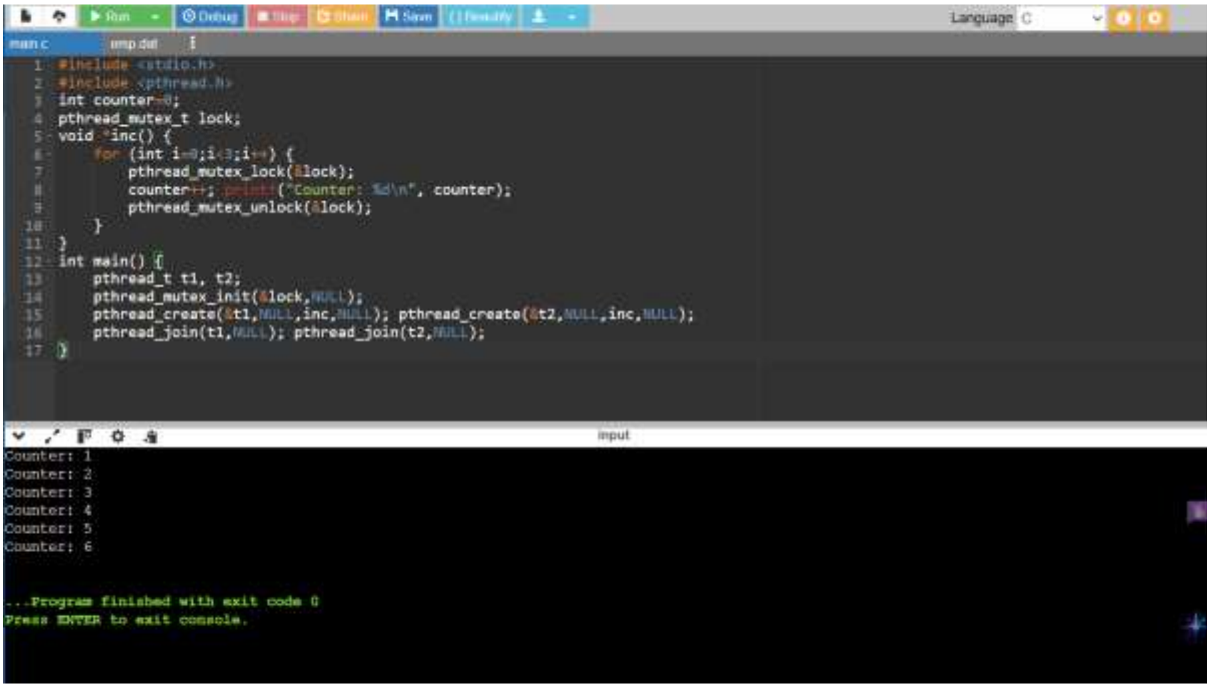
Counter: 2

Counter: 3

Counter: 4

Counter: 5

Counter: 6



RESULT:

Mutex-based synchronization program executed successfully.

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

Aim:

To simulate the Reader-Writer problem using semaphores.

Procedure:

1. Initialize semaphores for read/write synchronization.
2. Allow multiple readers but exclusive writers.
3. Ensure no simultaneous writer-reader conflict.

Code:

```
#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

sem_t wrt; int readcount=0, data=0; pthread_mutex_t mutex;

void *reader() {

   pthread_mutex_lock(&mutex); readcount++; if (readcount==1) sem_wait(&wrt);

   pthread_mutex_unlock(&mutex);

   printf("Reader read: %d\n", data);

   pthread_mutex_lock(&mutex); readcount--; if (readcount==0) sem_post(&wrt);

   pthread_mutex_unlock(&mutex);

}

void *writer() {

   sem_wait(&wrt); data++; printf("Writer wrote: %d\n", data); sem_post(&wrt);

}

int main() {

   pthread_t r1,r2,w1;

   sem_init(&wrt,0,1); pthread_mutex_init(&mutex,NULL);

   pthread_create(&r1,NULL,reader,NULL);

   pthread_create(&w1,NULL,writer,NULL);

   pthread_create(&r2,NULL,reader,NULL);

   pthread_join(r1,NULL); pthread_join(w1,NULL); pthread_join(r2,NULL);
```

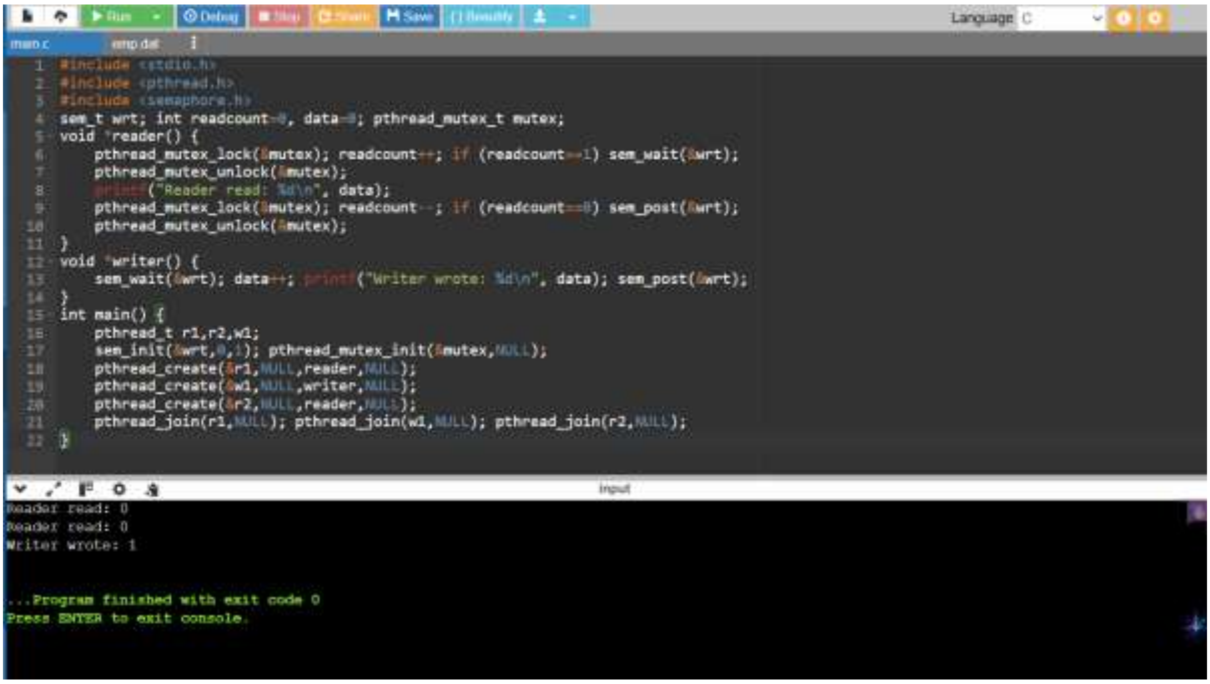
}

Sample Output:

Reader read: 0

Writer wrote: 1

Reader read: 1



RESULT:

Reader-Writer problem using semaphores executed successfully.

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

AIM:

To develop a C program to allocate memory to processes using the Worst Fit memory allocation strategy.

ALGORITHM:

1. Initialize memory blocks and process sizes.
2. For each process:
   a. Find the largest block that fits.
   b. Allocate memory and reduce block size.
3. Display allocation result.

CODE:

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

```c
int main() {
    int b[] = {100, 500, 200, 300, 600}, p[] = {212, 417, 112, 426}, a[4], i, j, k;
    for (i = 0; i < 4; i++) {
        a[i] = -1; k = -1;
        for (j = 0; j < 5; j++)
            if (b[j] >= p[i] && (k == -1 || b[j] > b[k])) k = j;
        if (k != -1) { a[i] = k; b[k] -= p[i]; }
    }
    for (i = 0; i < 4; i++)
        printf("P%d -> %s\n", i+1, a[i] != -1 ? "Block Found" : "Not Allocated");
    return 0;
}
```

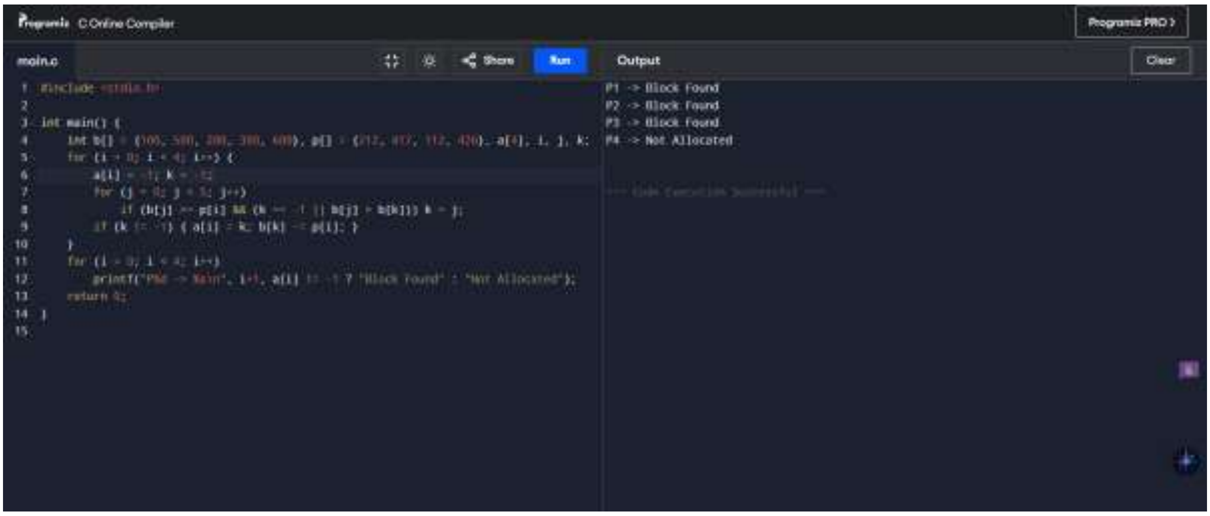
SAMPLE OUTPUT:

Process 1 -> Block 5

Process 2 -> Block 2

Process 3 -> Block 5

Process 4 -> Not Allocated



RESULT:

Worst fit memory allocation program executed successfully.

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

AIM:

To implement a C program that allocates memory to processes using the Best Fit memory allocation strategy.

ALGORITHM:

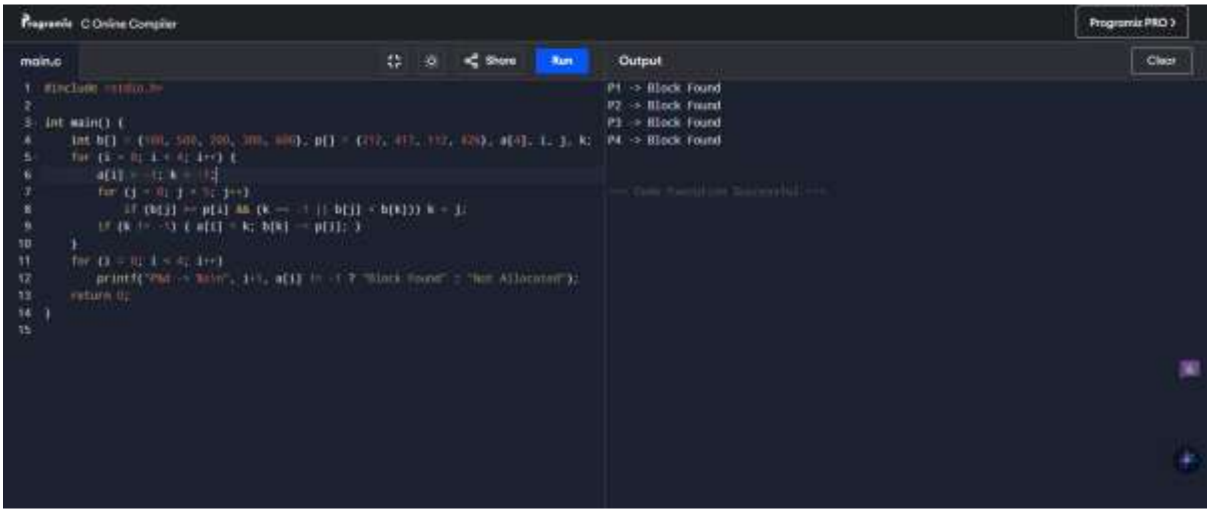
1. For each process:
   a. Find the smallest suitable block.
   b. Allocate memory and reduce that block.
2. Display allocation result.

CODE:

```
#include <stdio.h>

int main() {

    int b[] = {100, 500, 200, 300, 600}, p[] = {212, 417, 112, 426}, a[4], i, j, k;

    for (i = 0; i < 4; i++) {

        a[i] = -1; k = -1;

        for (j = 0; j < 5; j++)

            if (b[j] >= p[i] && (k == -1 || b[j] < b[k])) k = j;

        if (k != -1) { a[i] = k; b[k] -= p[i]; }

    }

    for (i = 0; i < 4; i++)

        printf("P%d -> %s\n", i+1, a[i] != -1 ? "Block Found" : "Not Allocated");

    return 0;

}
```

SAMPLE OUTPUT:

```
P1 -> Block Found

P2 -> Block Found

P3 -> Block Found

P4 -> Block Found
```



RESULT:

Best fit memory allocation program executed successfully.

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

AIM:

To write a C program that allocates memory to processes using the First Fit memory allocation strategy.

ALGORITHM:

1. For each process:
    a. Scan memory blocks from the beginning.
    b. Allocate the first block that fits.
2. Show result.

CODE:

```
#include <stdio.h>

int main() {

  int b[] = {100, 500, 200, 300, 600}, p[] = {212, 417, 112, 426}, a[4], i, j;

  for (i = 0; i < 4; i++) {

    a[i] = -1;

    for (j = 0; j < 5; j++) {

      if (b[j] >= p[i]) {

        a[i] = j;

        b[j] -= p[i];
```

```
                break;
            }
        }
    }
    for (i = 0; i < 4; i++)
        printf("P%d -> %s\n", i + 1, a[i] != -1 ? "Block Found" : "Not Allocated");
    return 0;
}
```

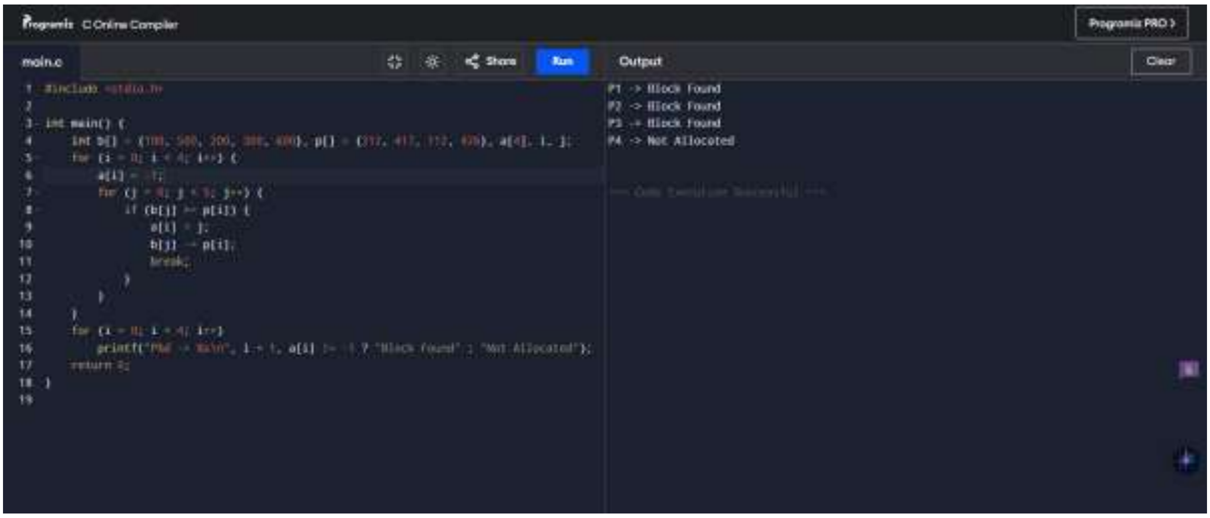
SAMPLE OUTPUT:

P1 -> Block Found

P2 -> Block Found

P3 -> Block Found

P4 -> Not Allocated



RESULT:

First fit memory allocation program executed successfully.

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

AIM:

To design a C program demonstrating UNIX file management system calls like open(), read(), write(), and close()

ALGORITHM:

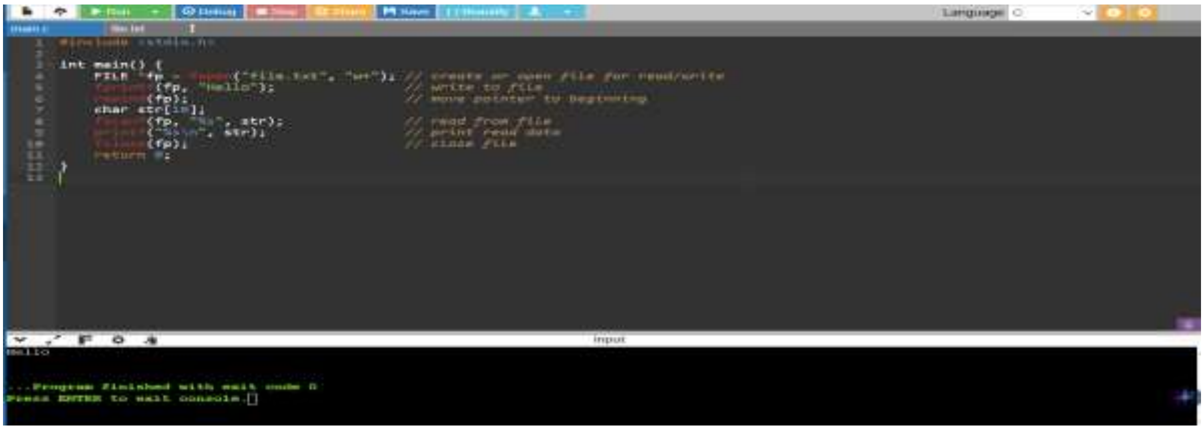
1. Open/create a file using open().
2. Write data using write().
3. Use lseek() to reset pointer.
4. Read data using read().
5. Close file using close().

CODE:

```
#include <stdio.h>

int main() {

   FILE *fp = fopen("file.txt", "w+"); // create or open file for read/write

   fprintf(fp, "Hello");             // write to file

   rewind(fp);                       // move pointer to beginning

   char str[10];

   fscanf(fp, "%s", str);            // read from file

   printf("%s\n", str);              // print read data

   fclose(fp);                       // close file

   return 0;

}
```

SAMPLE OUTPUT:

Hello



RESULT:

File management using UNIX system calls executed successfully.

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

AIM:

To implement a C program that demonstrates UNIX I/O system calls like fcntl, lseek, stat, opendir, and readdir.

ALGORITHM:

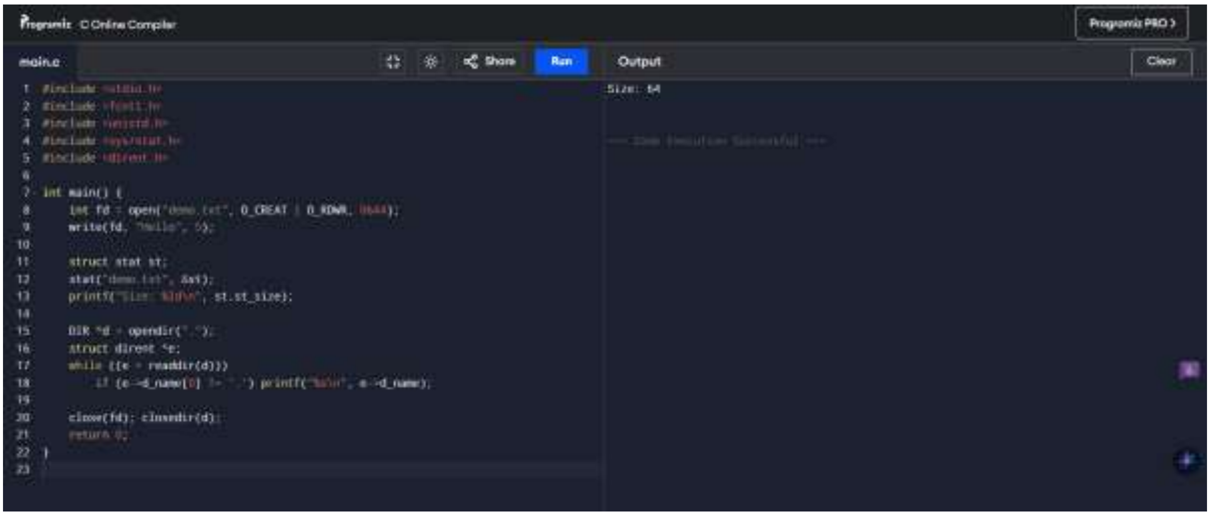
1. Create file using open()
2. Write data, seek with lseek()
3. Get file info using stat()
4. List directory using opendir() + readdir()

CODE:

```c
#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/stat.h>

#include <dirent.h>

int main() {

    int fd = open("demo.txt", O_CREAT | O_RDWR, 0644);

    write(fd, "Hello", 5);

    struct stat st;

    stat("demo.txt", &st);

    printf("Size: %ld\n", st.st_size);

    DIR *d = opendir(".");

    struct dirent *e;

    while ((e = readdir(d)))

        if (e->d_name[0] != '.') printf("%s\n", e->d_name);

    close(fd); closedir(d);

    return 0;

}
```

SAMPLE OUTPUT:

64



RESULT:

UNIX I/O system calls (fcntl, seek, stat, etc.) program executed successfully.

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

AIM:

To construct a C program that performs file management operations like create, write, read, and delete a file.

ALGORITHM:

1. Create and write to a file using fopen() and fprintf().
2. Read data using fgets().
3. Delete file using remove().
4. Print results.

CODE:

```
#include <stdio.h>

#include <stdlib.h>

int main() {

    FILE *fp;

    char data[100];

    // Create & write

    fp = fopen("file.txt", "w");

    if (fp == NULL) { printf("Error creating file\n"); return 1; }

    fprintf(fp, "Hello File!");
```

```
    fclose(fp);
    // Read file
    fp = fopen("file.txt", "r");
    if (fp == NULL) { printf("Error reading file\n"); return 1; }
    fgets(data, 100, fp);
    printf("File content: %s\n", data);
    fclose(fp);
    // Delete file
    if (remove("file.txt") == 0)
        printf("File deleted successfully.\n");
    else
        printf("Error deleting file.\n");
    return 0;
}
```

SAMPLE OUTPUT:

File content: Hello File!

File deleted successfully

RESULT:

File management operations program executed successfully.

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

AIM:

To develop a C program that simulates the basic function of the UNIX ls command, listing visible files in the current directory

ALGORITHM:

1. Open current directory with opendir(".")
2. Read entries using readdir()
3. Print non-hidden files (d_name[0] != '.')
4. Close directory

CODE:

```
#include <stdio.h>

#include <dirent.h>

int main() {

   DIR *d;

   struct dirent *dir;

   d = opendir(".");

   if (d) {

      while ((dir = readdir(d)) != NULL) {

         if (dir->d_name[0] != '.')  // Skip hidden files

            printf("%s  ", dir->d_name);

      }

      closedir(d);

      printf("\n");

   } else {

      printf("Unable to open directory.\n");

   }

   return 0;

}
```

SAMPLE OUTPUT:

main.c file1.txt demo.txt a.out

RESULT:

ls command simulation program executed successfully.

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

AIM:

To write a C program that simulates the grep UNIX command by searching for a word in a text file and printing matching lines

ALGORITHM:

1. Get filename and search word
2. Open file using fopen()
3. For each line:
   a. Check if it contains the word
   b. If yes, print it
4. Close file

CODE:

```
#include <stdio.h>

#include <string.h>

int main() {

    int n;

    char lines[10][100], word[50];

    printf("Enter number of lines: ");

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

```
    getchar(); // To consume newline
    printf("Enter %d lines:\n", n);
    for (int i = 0; i < n; i++)
        fgets(lines[i], sizeof(lines[i]), stdin);
    printf("Enter word to search: ");
    scanf("%s", word);
    printf("\nMatching lines:\n");
    for (int i = 0; i < n; i++)
        if (strstr(lines[i], word))
            printf("%s", lines[i]);
    return 0;
}
```

SAMPLE OUTPUT:

Enter number of lines: 2

Hello world

This is a test file

Enter word to search: test file

RESULT:

grep command simulation program executed successfully.

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

AIM:

To simulate a classical synchronization problem (Producer-Consumer) in C using semaphores and mutual exclusion.

ALGORITHM:

1. Initialize mutex, full, empty
2. In producer():
    - Wait on mutex and empty
    - Produce item
    - Signal full and mutex
3. In consumer():
    - Wait on mutex and full
    - Consume item
    - Signal empty and mutex

CODE:

```
#include <stdio.h>

int full = 0, empty = 3, item = 0;

int main() {

    int ch;

    while (1) {

        printf("\n1.Produce 2.Consume 3.Exit: ");

        scanf("%d", &ch);

        if (ch == 1) {

            if (empty > 0) {

                item++; full++; empty--;

                printf("Produced %d\n", item);

            } else printf("Buffer Full\n");

        }

        else if (ch == 2) {
```

```
            if (full > 0) {
                printf("Consumed %d\n", item);
                item--; full--; empty++;
            } else printf("Buffer Empty\n");
        }
        else break;
    }
    return 0;
}
```

SAMPLE INPUT:

1

1

2

2

3

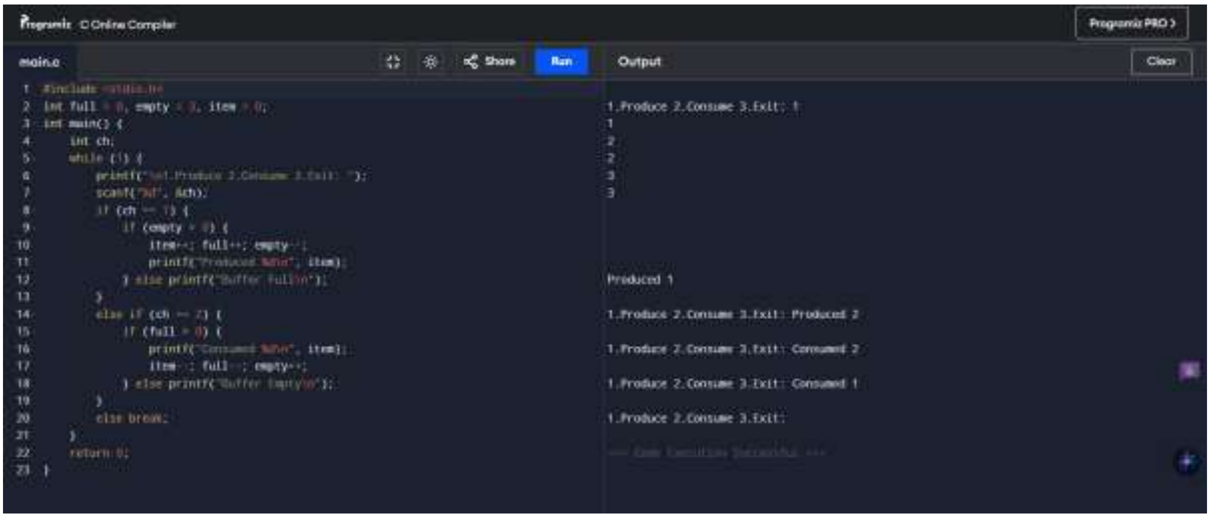
SAMPLE OUTPUT:

Produced 1

1.Produce 2.Consume 3.Exit: Produced 2

1.Produce 2.Consume 3.Exit: Consumed 2

1.Produce 2.Consume 3.Exit: Consumed 1



RESULT:

Classical process synchronization program executed successfully.

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

AIM:

To demonstrate creation, joining, equality checking, and exiting of threads using POSIX threads

ALGORITHM:

1. Create a thread using pthread_create()
2. Inside thread, compare thread IDs using pthread_equal()
3. Exit thread using pthread_exit()
4. In main, wait for thread using pthread_join()

CODE:

```
#include <stdio.h>

#include <pthread.h>

void* threadFunc(void* arg) {

   printf("Thread running with ID: %lu\n", pthread_self());

   // Check equality with itself (always true)

   if (pthread_equal(pthread_self(), pthread_self()))

      printf("Thread ID matches itself (equal)\n");

   pthread_exit("Thread exited");  // Exit with message

}

int main() {

   pthread_t t1;

   void* status;

   // Create

   pthread_create(&t1, NULL, threadFunc, NULL);

   printf("Main: Created thread %lu\n", t1);

   // Join

   pthread_join(t1, &status);
```

```c
    printf("Main: Joined thread, exit status: %s\n", (char*)status);
    return 0;
}
```

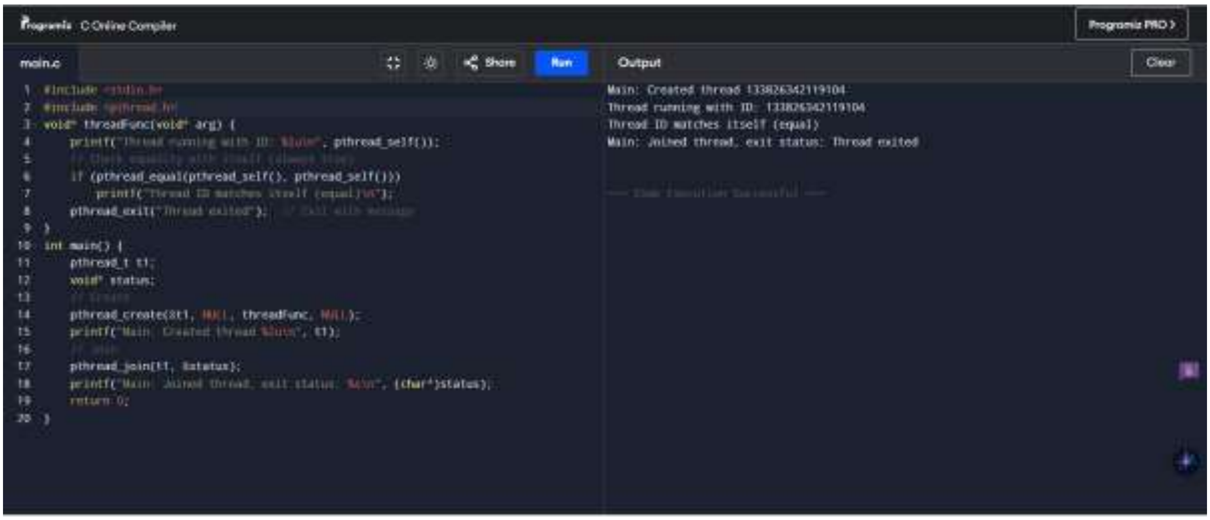
SAMPLE OUTPUT:

Main: Created thread 139797085206272

Thread running with ID: 139797085206272

Thread ID matches itself (equal)

Main: Joined thread, exit status: Thread exited



RESULT:

Thread handling operations (create, join, equal, exit) executed successfully.

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

AIM:

To simulate FIFO page replacement algorithm in memory management using C.

ALGORITHM:

1. Initialize empty frames.
2. For each page:
    - If already in frame → hit.
    - Else → replace oldest page (FIFO), and count page fault.
3. Print the current frame content.

CODE:

```c
#include <stdio.h>
int main() {
    int pages[20], frames[3] = {-1, -1, -1}, n, i, j, pos = 0, faults = 0, hit;
    printf("Enter number of pages: ");
    scanf("%d", &n);
    printf("Enter pages: ");
    for (i = 0; i < n; i++) scanf("%d", &pages[i]);
    for (i = 0; i < n; i++) {
        hit = 0;
        for (j = 0; j < 3; j++)
            if (frames[j] == pages[i]) hit = 1;
        if (!hit) {
            frames[pos] = pages[i];
            pos = (pos + 1) % 3;
            faults++;
        }
        printf("Frames: %d %d %d\n", frames[0], frames[1], frames[2]);
    }
    printf("Total Page Faults: %d\n", faults);
    return 0;
}
```

SAMPLE INPUT:

Enter number of pages: 4

Enter page numbers: 1 2 3 4

SAMPLE OUTPUT:

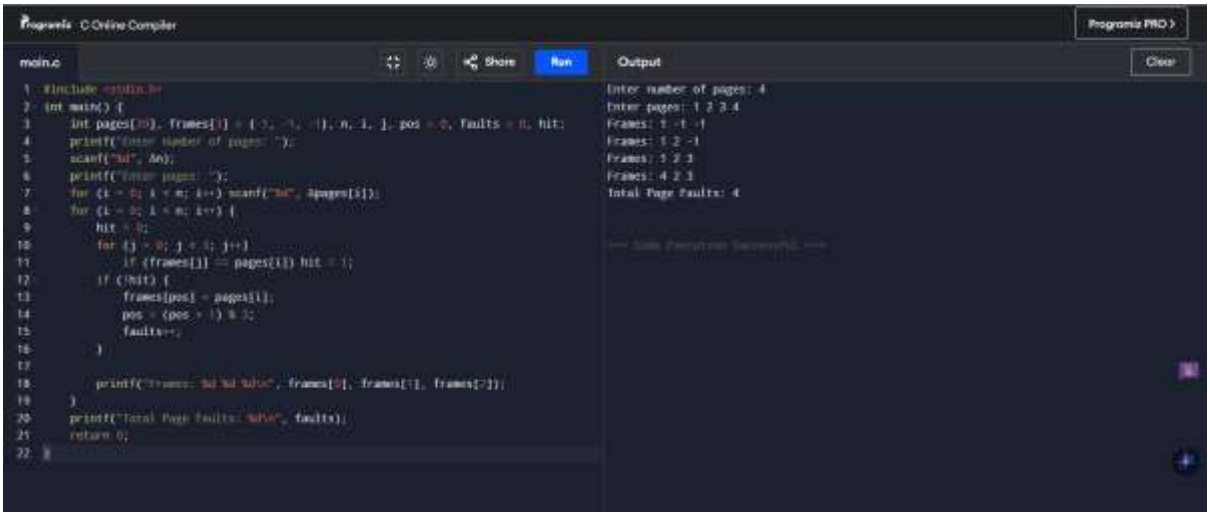
Frames: 1 -1 -1

Frames: 1 2 -1

Frames: 1 2 3

Frames: 4 2 3

Total Page Faults = 4



RESULT:

FIFO paging simulation program executed successfully.

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

AIM:

To simulate the LRU page replacement algorithm using arrays and timestamps in C.

ALGORITHM:

1. Initialize 3 frames with -1.
2. For each page:
    - If it's a hit → update its time.
    - If it's a miss → replace the least recently used page.
3. Count and print page faults and frames.

CODE:

```
#include <stdio.h>
int main() {
    int pages[20], n, frames[3] = {-1, -1, -1}, used[3] = {0}, time = 0, faults = 0;
    printf("Enter number of pages: ");
    scanf("%d", &n);
    printf("Enter page numbers: ");
    for (int i = 0; i < n; i++) scanf("%d", &pages[i]);
```

```c
    for (int i = 0; i < n; i++) {
        int hit = 0, min = 0;
        for (int j = 0; j < 3; j++) {
            if (frames[j] == pages[i]) {
                hit = 1;
                used[j] = ++time;
            }
        }
        if (!hit) {
            for (int j = 1; j < 3; j++)
                if (used[j] < used[min]) min = j;
            frames[min] = pages[i];
            used[min] = ++time;
            faults++;
        }
        printf("Frames: %d %d %d\n", frames[0], frames[1], frames[2]);
    }
    printf("Total Page Faults = %d\n", faults);
    return 0;
}
```

SAMPLE INPUT:

Enter number of pages: 4

Enter pages: 1 2 3 1

SAMPLE OUTPUT:

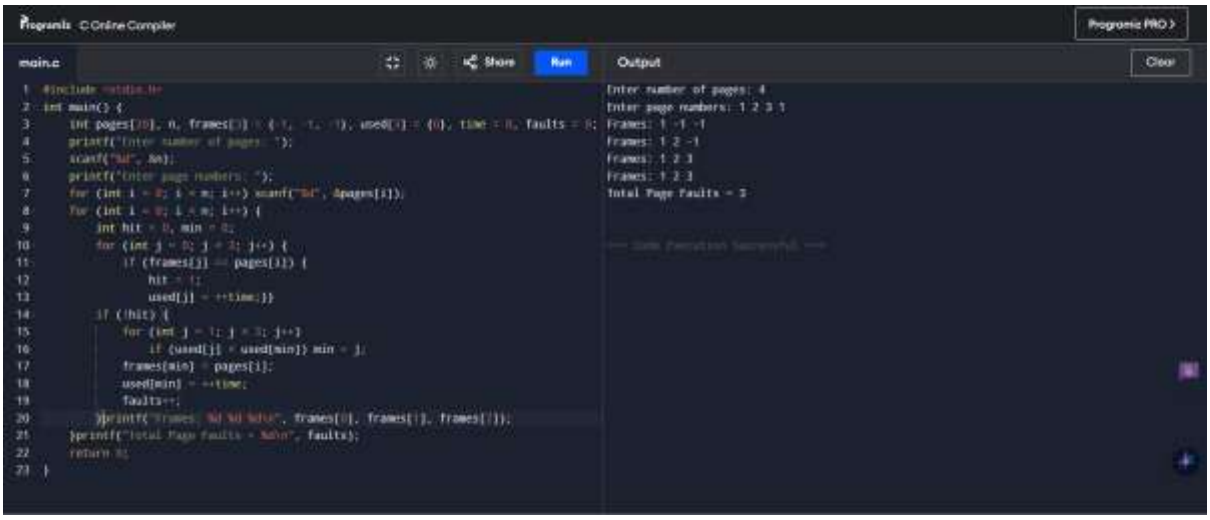
Frames: 1 -1 -1

Frames: 1 2 -1

Frames: 1 2 3

Frames: 1 2 3

Total Page Faults = 3



RESULT:

LRU paging simulation program executed successfully.

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

AIM:

To simulate Optimal page replacement using C by predicting which page won't be used for the longest time in the future and replacing that page.

ALGORITHM:

1. Initialize empty frames.
2. For each page:
    - If in frame → hit.
    - Else → Replace page not used soonest in future.
3. Count page faults.
4. Print total faults.

CODE:

```
#include <stdio.h>

int predict(int p[], int f[], int n, int idx) {

    int far = -1, pos = -1;

    for (int i = 0; i < 3; i++) {

        int j;

        for (j = idx; j < n; j++)

            if (f[i] == p[j]) break;

        if (j == n) return i;
```

```
        if (j > far) far = j, pos = i;
    }
    return pos;
}
int main() {
    int p[20], f[3] = {-1, -1, -1}, n, i, j, hit, pos, faults = 0;
    printf("Pages: "); scanf("%d", &n);
    for (i = 0; i < n; i++) scanf("%d", &p[i]);
    for (i = 0; i < n; i++) {
        hit = 0;
        for (j = 0; j < 3; j++)
            if (f[j] == p[i]) hit = 1;
        if (!hit) {
            pos = (f[0] == -1 || f[1] == -1 || f[2] == -1) ?
                  (f[0] == -1 ? 0 : f[1] == -1 ? 1 : 2) :
                  predict(p, f, n, i+1);
            f[pos] = p[i]; faults++;
        }
        printf("Frames: %d %d %d\n", f[0], f[1], f[2]);
    }
    printf("Faults: %d\n", faults);
}
```

SAMPLE INPUT:

Pages: 4

7 0 1 2

SAMPLE OUTPUT:

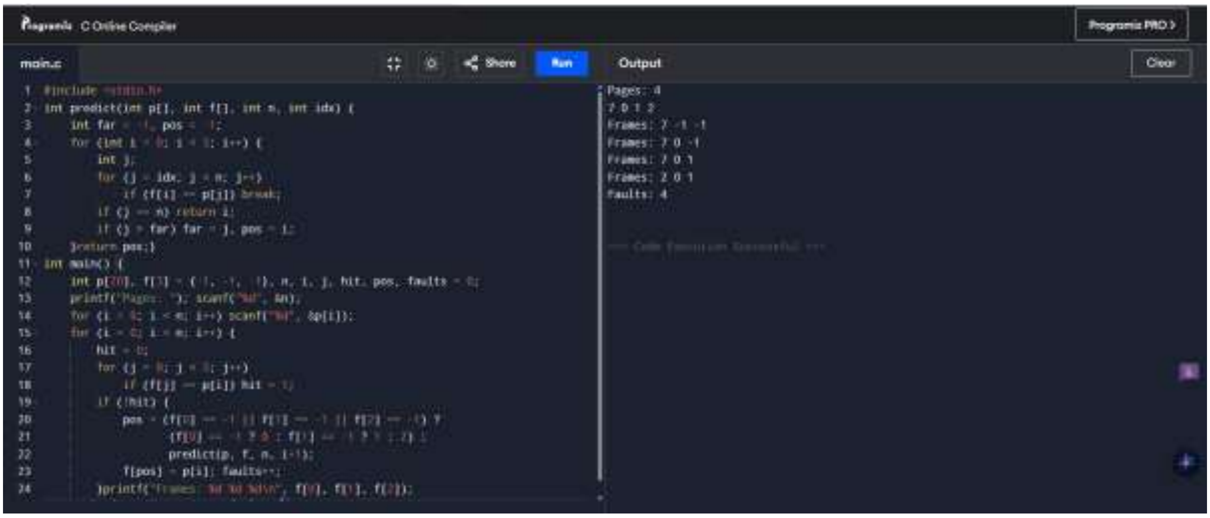
Frames: 7 -1 -1

Frames: 7 0 -1

Frames: 7 0 1

Frames: 2 0 1

Faults: 4



RESULT:

Optimal paging simulation program executed successfully.

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.

AIM:

To simulate sequential file allocation using a simple C program.

ALGORITHM:

1. Input number of files and their starting blocks + lengths.
2. Store blocks sequentially.
3. To access a record, read from the starting block up to the desired record.

CODE:

```
#include <stdio.h>

struct File {

    int start, length;

};

int main() {

    int n, i, j;

    struct File f[10];
```

```
    printf("Enter number of files: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("File %d start block & length: ", i + 1);
        scanf("%d %d", &f[i].start, &f[i].length);
    }
    printf("\nFile\tBlocks\n");
    for (i = 0; i < n; i++) {
        printf("%d\t", i + 1);
        for (j = 0; j < f[i].length; j++)
            printf("%d ", f[i].start + j);
        printf("\n");
    }

    return 0;
}
```

SAMPLE INPUT:

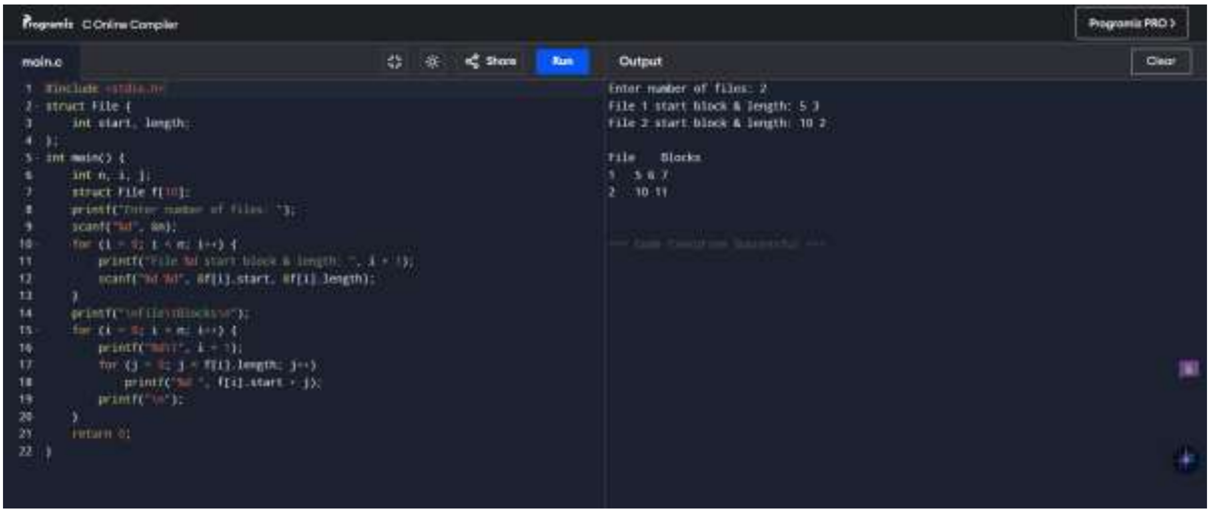
Enter number of files: 2

File 1 start block & length: 5 3

File 2 start block & length: 10 2

SAMPLE OUTPUT:

```
File    Blocks
1       5 6 7
2       10 11
```



RESULT:

Sequential file allocation program executed successfully.

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.

AIM:

To simulate Indexed File Allocation where the index block stores pointers to all the blocks of a file.

ALGORITHM:

1. For each file:
   - Input the index block and block numbers used by the file.
2. Store and display:
   - The index block and all its entries (pointers to file blocks).

CODE:

```
#include <stdio.h>

int main() {

  int files, i, j, blocks, indexBlock, dataBlock[10];

  printf("Enter number of files: ");

  scanf("%d", &files);

  for (i = 0; i < files; i++) {

    printf("Enter index block for file %d: ", i + 1);

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

```
        printf("Enter number of blocks for file %d: ", i + 1);
        scanf("%d", &blocks);
        printf("Enter block numbers: ");
        for (j = 0; j < blocks; j++)
            scanf("%d", &dataBlock[j]);
        printf("\nFile %d => Index Block: %d => Blocks: ", i + 1, indexBlock);
        for (j = 0; j < blocks; j++)
            printf("%d ", dataBlock[j]);
        printf("\n\n");
    }
    return 0;
}
```

SAMPLE INPUT:

Enter number of files: 2

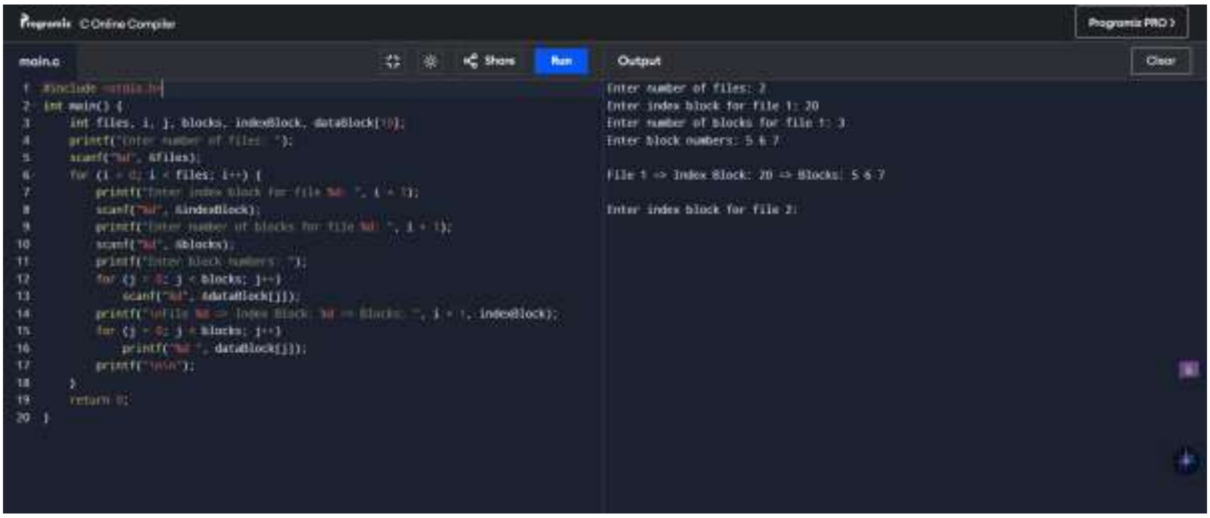
Enter index block for file 1: 20

Enter number of blocks for file 1: 3

Enter block numbers: 5 6 7

SAMPLE OUTPUT:

File 1 => Index Block: 20 => Blocks: 5 6 7



RESULT:

Indexed file allocation strategy program executed successfully.

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.

AIM:

To simulate Linked Allocation where files are scattered blocks connected by pointers.

ALGORITHM:

1. For each file, input:
    - Start and end block
    - Number of blocks
    - Sequence of blocks linked together
2. Display each file's linked block path.

CODE:

```
#include <stdio.h>

int main() {

  int files, i, j, blocks, start, end, chain[30];

  printf("Enter number of files: ");

  scanf("%d", &files);

  for (i = 0; i < files; i++) {

    printf("\nFile %d:\n", i + 1);

    printf("Enter start and end blocks: ");

    scanf("%d %d", &start, &end);

    printf("Enter number of blocks: ");

    scanf("%d", &blocks);

    printf("Enter block chain: ");

    for (j = 0; j < blocks; j++)

      scanf("%d", &chain[j]);

    printf("File %d => Start: %d, End: %d => Chain: ", i + 1, start, end);
```

```
        for (j = 0; j < blocks; j++)
            printf("%d -> ", chain[j]);
        printf("NULL\n");
    }
    return 0;
}
```

SAMPLE INPUT:

Enter number of files: 1

File 1:

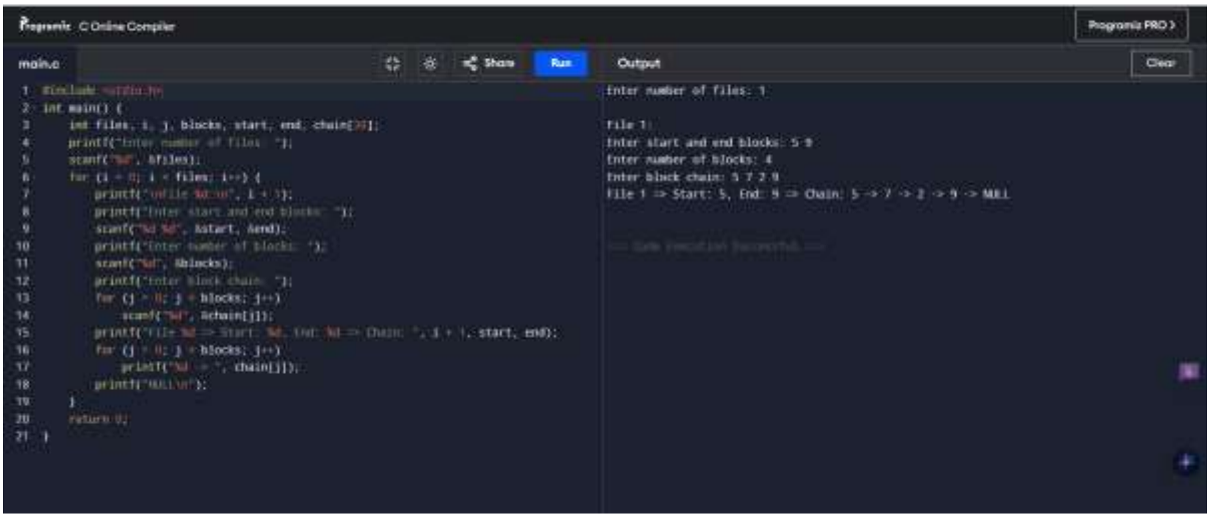
Enter start and end blocks: 5 9

Enter number of blocks: 4

Enter block chain: 5 7 2 9

SAMPLE OUTPUT:

File 1 => Start: 5, End: 9 => Chain: 5 -> 7 -> 2 -> 9 -> NULL



RESULT:

Linked file allocation strategy program executed successfully.

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

AIM:

To simulate FCFS Disk Scheduling, where disk requests are handled in the order they arrive.

ALGORITHM:

1. Input number of requests and their positions.
2. Input initial head position.
3. For each request (in order), calculate the seek time.
4. Print total and average seek time.

CODE:

```c
#include <stdio.h>

#include <stdlib.h>

int main() {

    int n, i, head, total = 0;

    int req[100];

    printf("Enter number of requests: ");

    scanf("%d", &n);

    printf("Enter request sequence: ");

    for (i = 0; i < n; i++) scanf("%d", &req[i]);

    printf("Enter initial head position: ");

    scanf("%d", &head);

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

        total += abs(req[i] - head);

        head = req[i];

    }

    printf("Total seek time: %d\n", total);

    printf("Average seek time: %.2f\n", (float)total / n);

    return 0;

}
```

SAMPLE INPUT:

Enter number of requests: 5

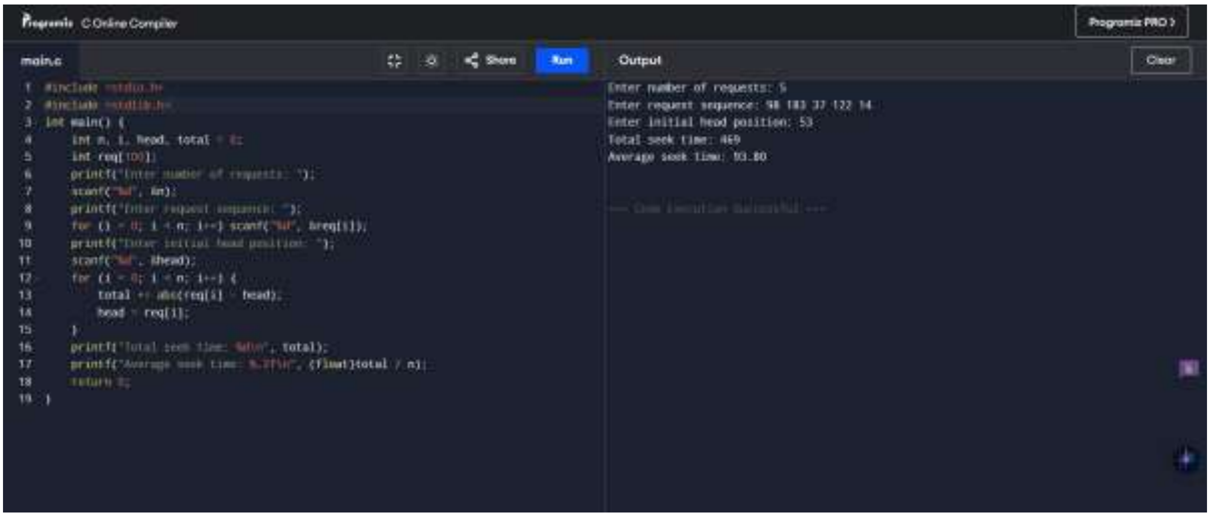
Enter request sequence: 98 183 37 122 14

Enter initial head position: 53

SAMPLE OUTPUT:

Total seek time: 640

Average seek time: 128.00



RESULT:

First Come First Served disk scheduling program executed successfully.

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

AIM:

To simulate the SCAN disk scheduling algorithm where the disk head moves in one direction (like an elevator), then reverses.

ALGORITHM:

1. Input request queue and initial head position.
2. Input direction (toward 0 or toward max disk size).
3. Sort requests.
4. Move in the given direction, servicing all requests.
5. Reverse direction and service remaining.

CODE:

```
#include <stdio.h>

#include <stdlib.h>

int cmp(const void *a, const void *b) { return (*(int*)a - *(int*)b); }

int main() {

    int r[100], n, h, i, d, t = 0;

    printf("Requests: "); scanf("%d", &n);

    for (i = 0; i < n; i++) scanf("%d", &r[i]);
```

```c
    printf("Head pos: "); scanf("%d", &h);
    printf("Dir (0=left,1=right): "); scanf("%d", &d);
    r[n++] = h;
    qsort(r, n, sizeof(int), cmp);
    int p; for (i = 0; i < n; i++) if (r[i] == h) { p = i; break; }
    printf("Seq: ");
    if (d) {
        for (i = p; i < n; i++) t += abs(h - (h = r[i])), printf("%d ", r[i]);
        for (i = p - 1; i >= 0; i--) t += abs(h - (h = r[i])), printf("%d ", r[i]);
    } else {
        for (i = p; i >= 0; i--) t += abs(h - (h = r[i])), printf("%d ", r[i]);
        for (i = p + 1; i < n; i++) t += abs(h - (h = r[i])), printf("%d ", r[i]);
    }
    printf("\nTotal: %d\nAvg: %.2f\n", t, (float)t / (n - 1));
    return 0;
}
```

SAMPLE INPUT:

Requests: 5

98 183 37 122 14

Head pos: 53

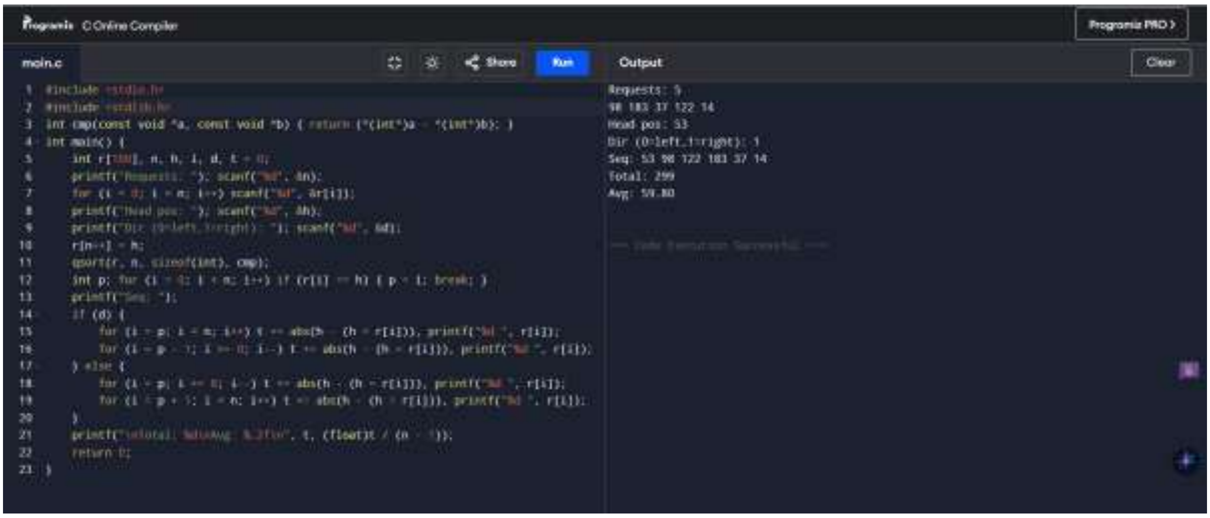
Dir (0=left,1=right): 1

SAMPLE OUTPUT:

Seq: 53 98 122 183 37 14

Total: 322

Avg: 64.40



RESULT:

SCAN disk scheduling algorithm program executed successfully.

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

AIM:

To simulate the C-SCAN (Circular SCAN) disk scheduling algorithm where the disk head moves in one direction only (like SCAN), but instead of reversing, it jumps to the beginning and continues.

ALGORITHM:

1. Input requests and initial head position.
2. Add head to the request queue.
3. Sort the request queue.
4. Move right from the head to the end, then jump to start and continue.
5. Calculate total and average seek time.

CODE:

```
#include <stdio.h>

#include <stdlib.h>

int cmp(const void *a, const void *b) { return (*(int*)a - *(int*)b); }

int main() {

    int r[100], n, i, head, size, pos, t = 0;

    printf("Requests: "); scanf("%d", &n);

    for (i = 0; i < n; i++) scanf("%d", &r[i]);

    printf("Disk size & Head: "); scanf("%d %d", &size, &head);
```

```c
    r[n++] = head;
    qsort(r, n, sizeof(int), cmp);
    for (i = 0; i < n; i++) if (r[i] == head) { pos = i; break; }
    printf("Sequence: ");
    for (i = pos; i < n; i++) { printf("%d ", r[i]); t += abs(head - r[i]); head = r[i]; }
    if (pos > 0) {
        t += abs(head - (size - 1)) + (size - 1); head = 0;
        for (i = 0; i < pos; i++) { printf("%d ", r[i]); t += abs(head - r[i]); head = r[i]; }
    }
    printf("\nTotal Seek: %d\nAvg Seek: %.2f\n", t, (float)t / (n - 1));
    return 0;
}
```

SAMPLE INPUT:

Requests: 5

95 180 34 119 11

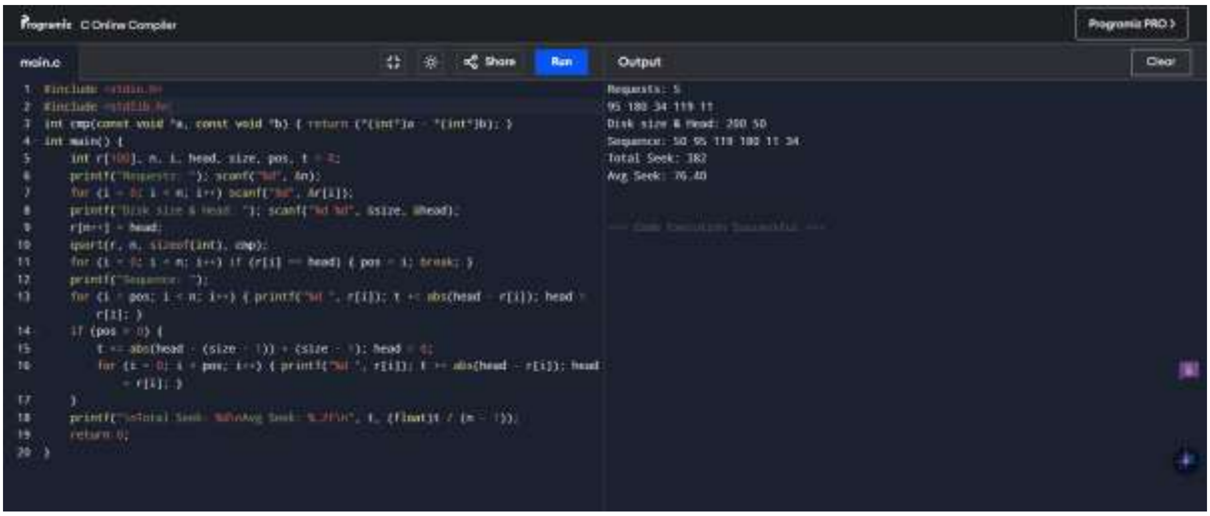
Disk size & Head: 200 50

SAMPLE OUTPUT:

Sequence: 50 95 119 180 11 34

Total Seek: 391

Avg Seek: 78.20



RESULT:

C-SCAN disk scheduling algorithm program executed successfully.

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

AIM:

To illustrate file access permissions and user types in Linux.

ALGORITHM:

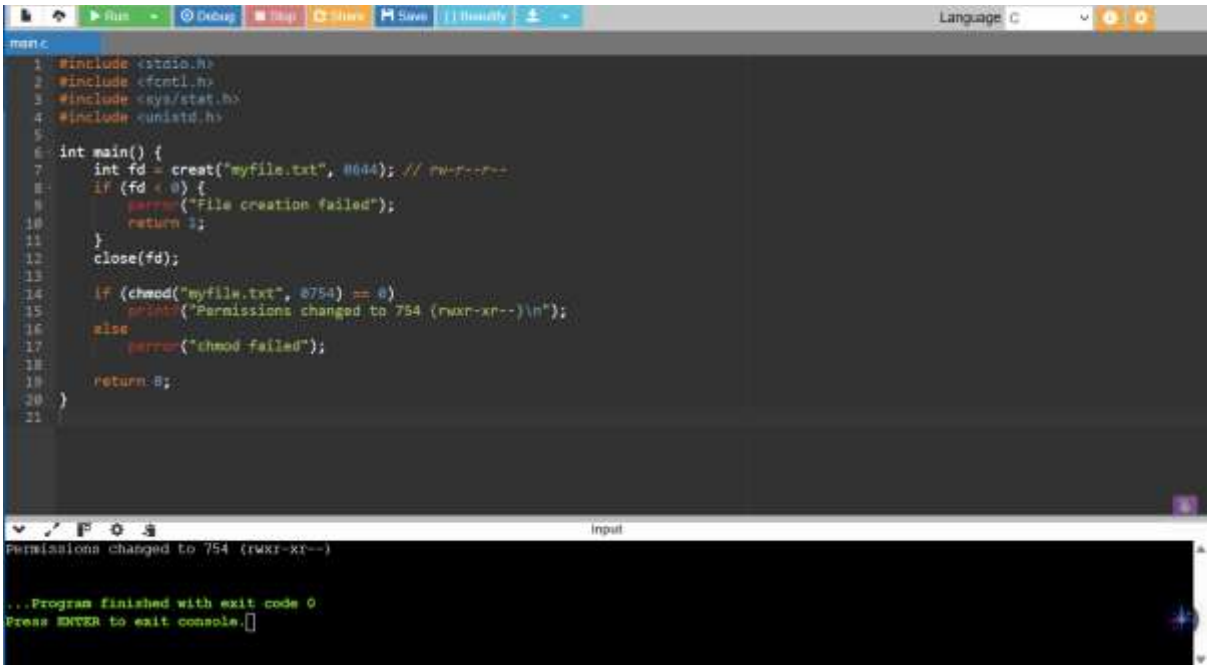
1. Create a file.
2. Use ls -l to view permissions.
3. Use chmod to modify permissions.
4. Check the result again using ls -l.

CODE:

```
#include <stdio.h>

#include <fcntl.h>

#include <sys/stat.h>

#include <unistd.h>

int main() {

   int fd = creat("myfile.txt", 0644); // rw-r--r--

   if (fd < 0) {

      perror("File creation failed");

      return 1;

   }

   close(fd);

   if (chmod("myfile.txt", 0754) == 0)

      printf("Permissions changed to 754 (rwxr-xr--)\n");

   else

      perror("chmod failed");

   return 0;

}
```

SAMPLE OUTPUT:

Permissions changed to 754 (rwxr-xr--)



RESULT:

File access permission and Linux user type program executed successfully.