AY 2022-23

PROJECT REPORT

ON

OS-Lab file

Submitted for

ITC403: Operating System

By

Prabhat Kumar (21320)

IV SEMESTER

BTECH-IT

SCHOOL OF COMPUTING

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY UNA

HIMACHAL PRADESH

25th APRIL 2023

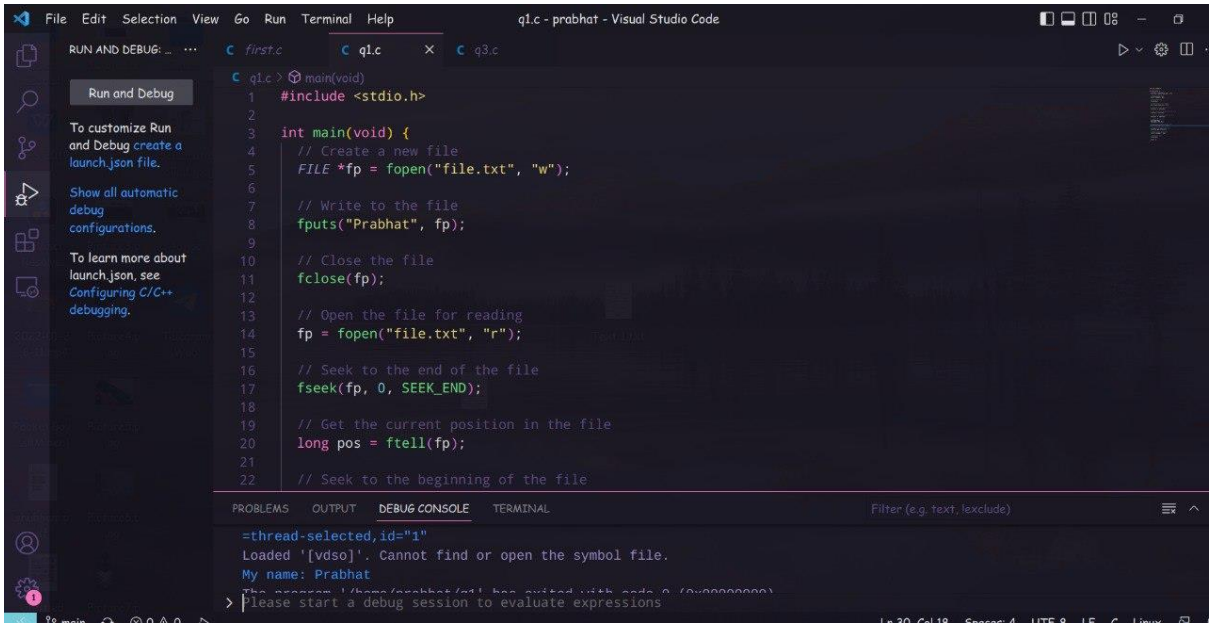
Lab 1

Aim: Create simple programs of file creation, opening, closing and seeking in C. Also demonstrate file read and write operations.

Theory: We employ a C application to demonstrate file creation, opening, closing, seeking, and read/write activities. The application should create file, open and close file.

```c
include <stdio.h>

int main(void) {
  // Create a new file
  FILE *fp = fopen("file.txt", "w");

  // Write to the file
  fputs("Prabhat", fp);

  // Close the file
  fclose(fp);

  // Open the file for reading
  fp = fopen("file.txt", "r");

  // Seek to the end of the file
  fseek(fp, 0, SEEK_END);

  // Get the current position in the file
  long pos = ftell(fp);

  // Seek to the beginning of the file
  fseek(fp, 0, SEEK_SET);

  // Read from the file
  char buf[100];
  fgets(buf, 100, fp);

  // Print the contents of the file
  printf("My name: %s\n", buf);

  // Seek to the end of the file
  fseek(fp, pos, SEEK_SET);
```

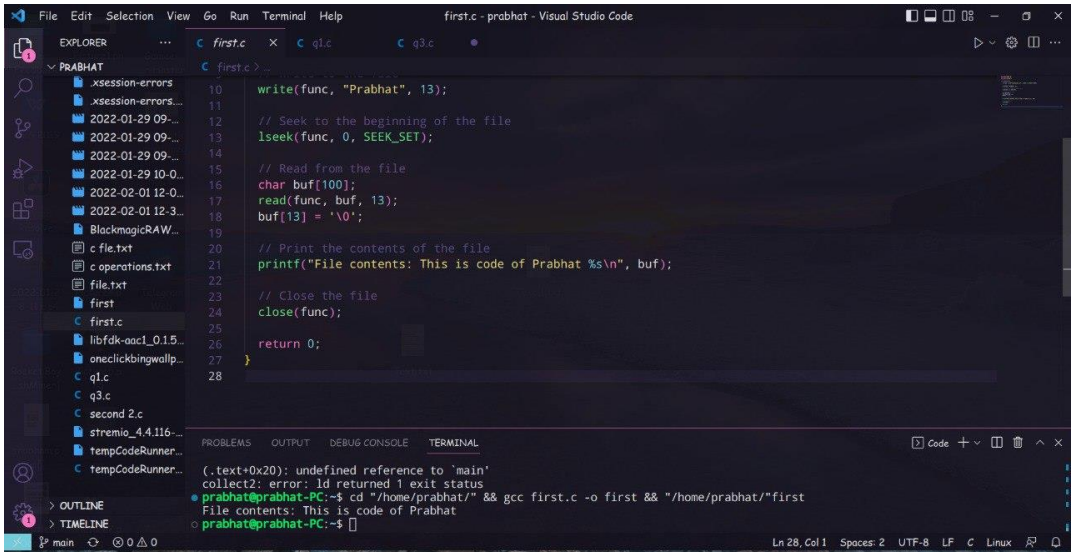
```c
  // Write to the file again
  fputs(" Goodbye!", fp);

  // Close the file
  fclose(fp);

  return 0;
}
```



Aim: Create, open and seek a file with different options/flags through above system calls.

Theory: To demonstrate how to create, open, and seek a file in the C programming language using system calls, in particular fopen() and fseek(). A file should be opened for writing, some data should be entered into it, the file should be closed, then it should be opened again for reading. This time, the programmer should seek to the file's beginning, read the contents, then print the contents to the screen.

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

int main(void) {
  // Create a new file
  int func = open("c operations.txt", O_CREAT | O_WRONLY, 0666);

  // Write to the file
  write(func, "Prabhat", 13);

  // Seek to the beginning of the file
  lseek(func, 0, SEEK_SET);

  // Read from the file
  char buf[100];
  read(func, buf, 13);
  buf[13] = '\0';

  // Print the contents of the file
  printf("File contents: This is code of Prabhat %s\n", buf);

  // Close the file
  close(func);

  return 0;
}
```

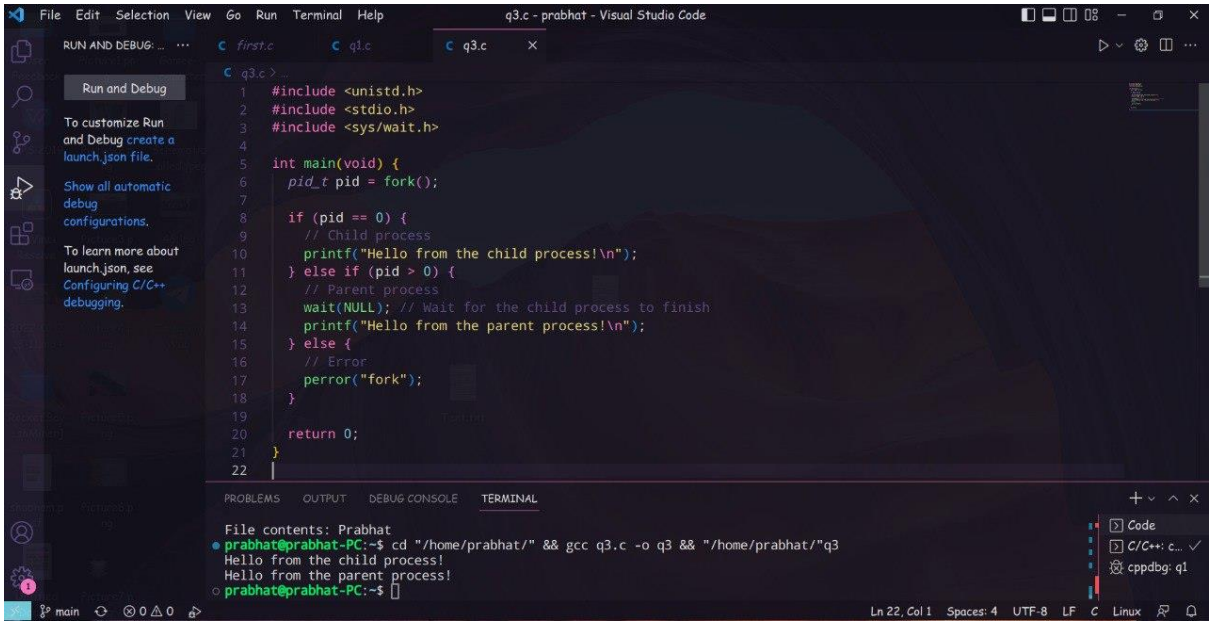


Aim: Try three more file related operations using system calls.

Theory: Three other file related operations using system calls are –

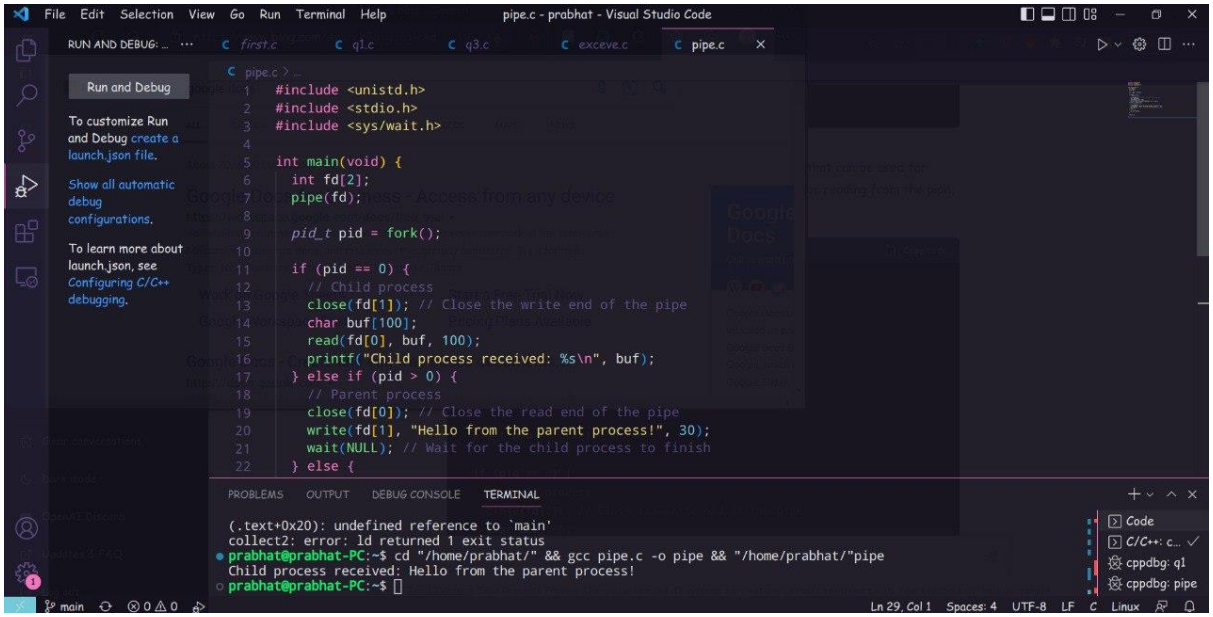
- Fork
- Pipe
- Exceve

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

int main(void) {
  pid_t pid = fork();

  if (pid == 0) {
    // Child process
    char *argv[] = {"ls", "-l", NULL};
    char *envp[] = {NULL};
    execve("/bin/ls", argv, envp);
  } else if (pid > 0) {
    // Parent process
    wait(NULL); // Wait for the child process to finish
    printf("Child process finished.\n");
  } else {
    // Error
    perror("fork");
  }

  return 0;
}
```



Execve: This system call replaces the current process image with a new process image. The new process image is specified by the pathname, and the arguments and environment.

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

int main(void) {
  pid_t pid = fork();

  if (pid == 0) {
    // Child process
    char *argv[] = {"ls", "-l", NULL};
    char *envp[] = {NULL};
    execve("/bin/ls", argv, envp);
  } else if (pid > 0) {
    // Parent process
    wait(NULL); // Wait for the child process to finish
    printf("Child process finished.\n");
  } else {
    // Error
    perror("fork");
  }

  return 0;
}
```



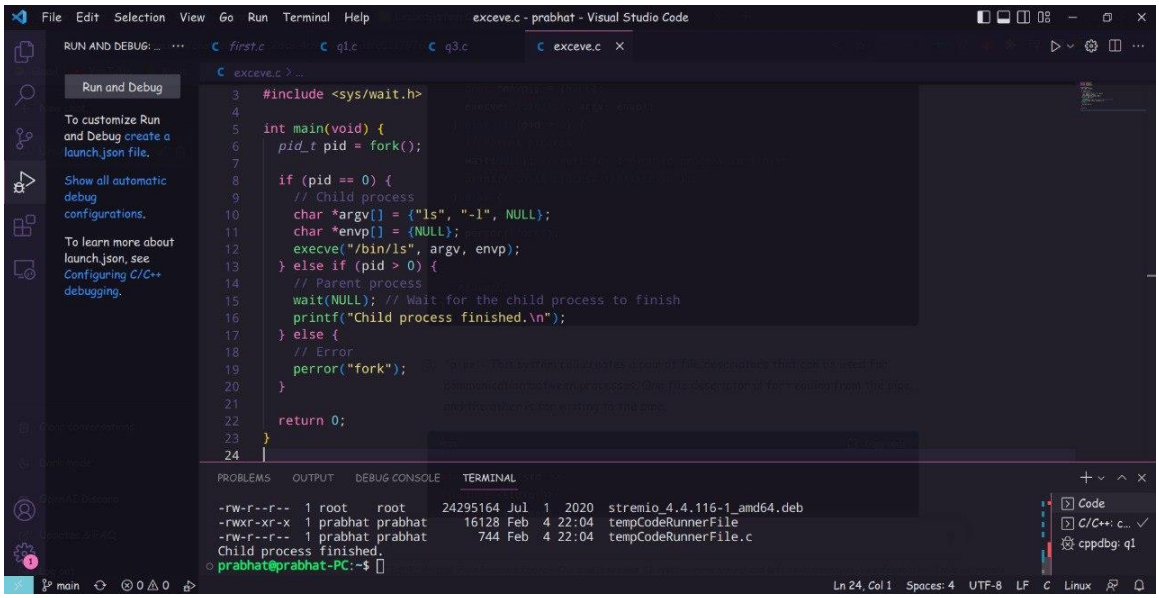
Pipe: This system call creates a pair of file descriptors that can be used for communication between processes. One file descriptor is for reading from the pipe, and the other is for writing to the pipe.

Code

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

int main(void) {
  int fd[2];
  pipe(fd);

  pid_t pid = fork();

  if (pid == 0) {
    // Child process
    close(fd[1]); // Close the write end of the pipe
    char buf[100];
    read(fd[0], buf, 100);
    printf("Child process received: %s\n", buf);
  } else if (pid > 0) {
    // Parent process
    close(fd[0]); // Close the read end of the pipe
    write(fd[1], "Hello from the parent process!", 30);
    wait(NULL); // Wait for the child process to finish
```

```
  } else {
    // Error
    perror("fork");
  }

  return 0;
}
```

Output



Lab 2

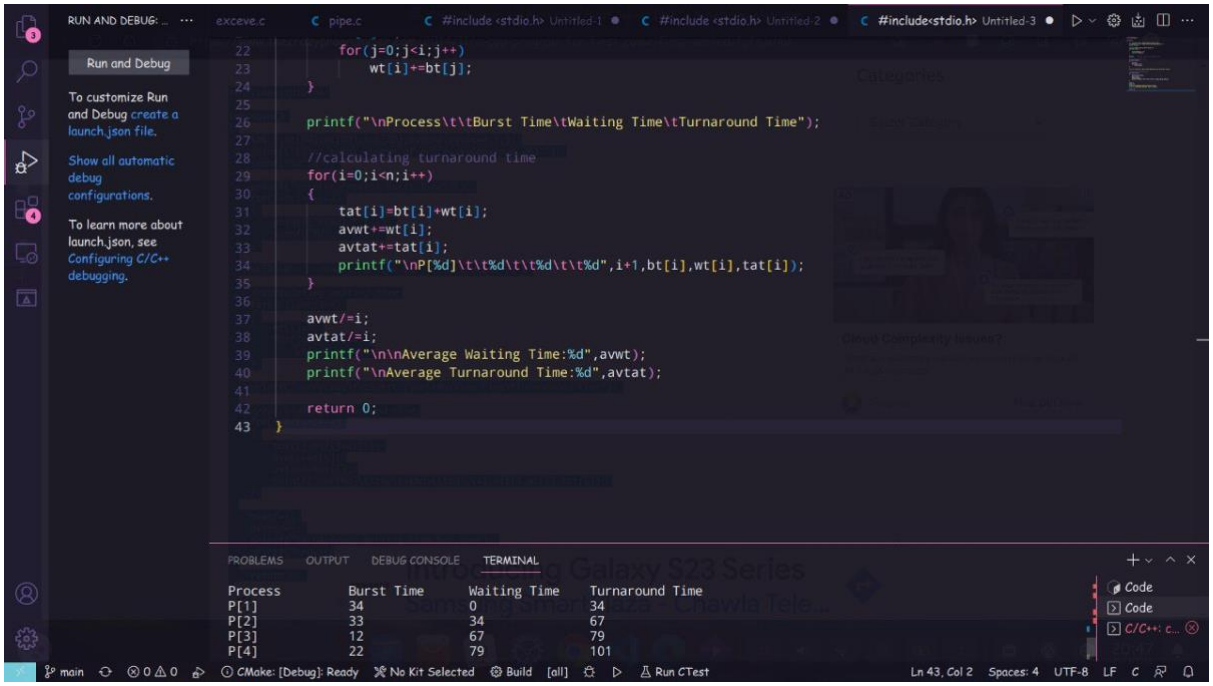
Aim: Create simple programs implementing FCFS and SJF using different ATs.

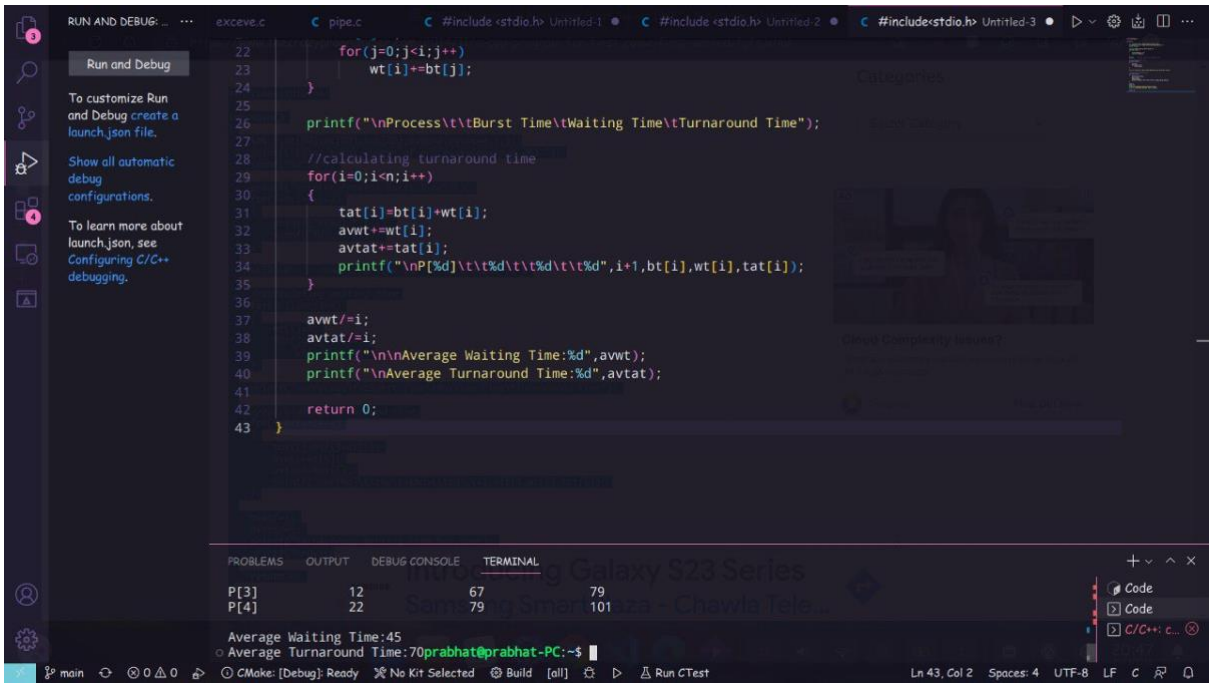
Theory: The FCFS scheduling algorithm is a non-preemptive scheduling algorithm in which the processes are executed in the order in which they arrive. The process that arrives first is executed first, and so on. In FCFS, the waiting time of a process is the time that elapses between its arrival and its execution, and the turnaround time is the time that elapses between its arrival and its completion.

The SJF scheduling algorithm is a non-preemptive scheduling algorithm in which the process with the shortest burst time is executed first. The idea behind SJF is that shorter jobs will complete faster, leading to better response time for the user. In SJF, the waiting time of a process is the time that elapses between its arrival and the start of its execution, and the turnaround time is the time that elapses between its arrival and its completion.

```c
#include<stdio.h>

void main()
{
    int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp;
    float avg_wt,avg_tat;
    printf("Enter number of process:");
    scanf("%d",&n);

    printf("\nEnter Burst Time:\n");
    for(i=0;i<n;i++)
    {
        printf("p%d:",i+1);
        scanf("%d",&bt[i]);
        p[i]=i+1;           //contains process number
    }

    //sorting burst time in ascending order using selection sort
    for(i=0;i<n;i++)
    {
```

```
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(bt[j]<bt[pos])
                pos=j;
        }

        temp=bt[i];
        bt[i]=bt[pos];
        bt[pos]=temp;

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

    wt[0]=0;            //waiting time for first process will be zero

    //calculate waiting time
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];

        total+=wt[i];
    }

    avg_wt=(float)total/n;      //average waiting time
    total=0;

    printf("\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];     //calculate turnaround time
        total+=tat[i];
        printf("\np%d\t\t  %d\t\t    %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }

    avg_tat=(float)total/n;     //average turnaround time
    printf("\n\nAverage Waiting Time=%f",avg_wt);
    printf("\nAverage Turnaround Time=%f\n",avg_tat);
}
```



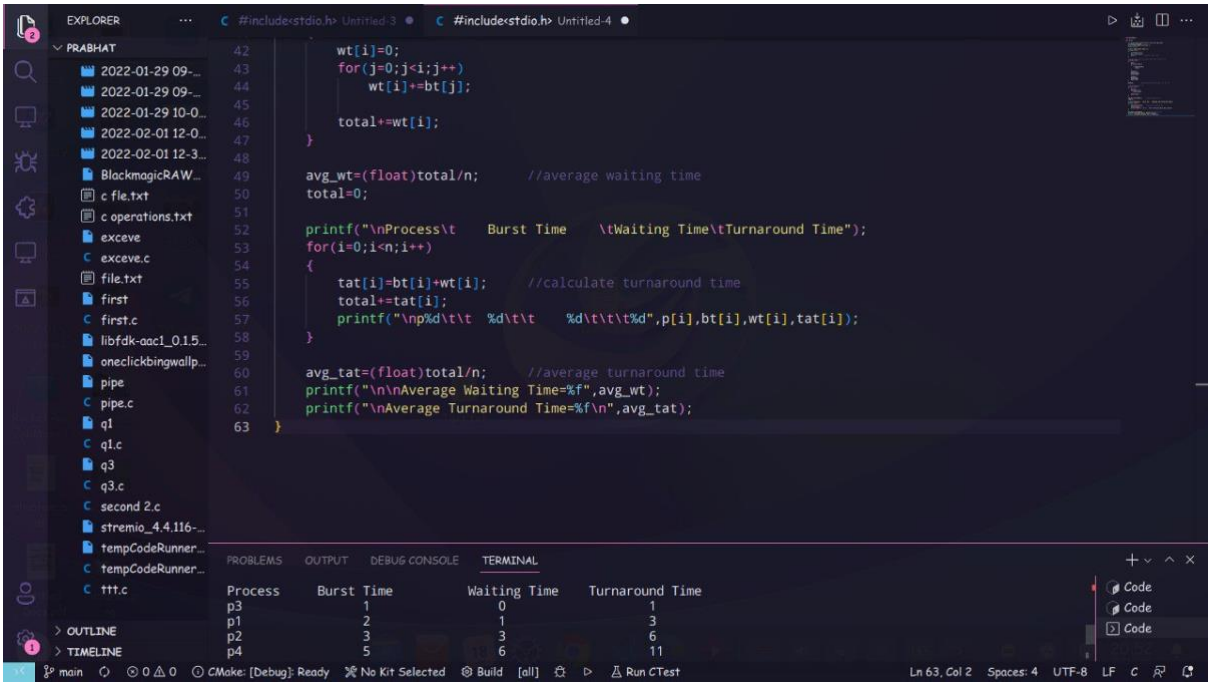


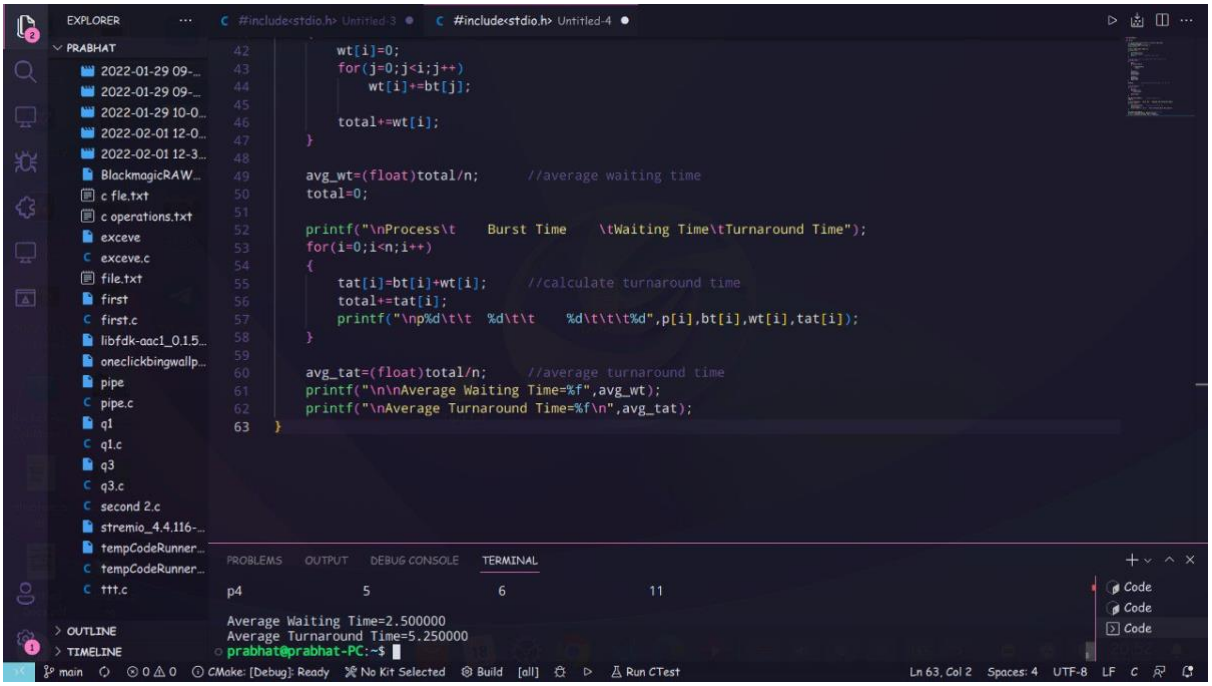
Aim:

Theory:

```c
#include<stdio.h>

int main()
{
    int n,bt[20],wt[20],tat[20],avwt=0,avtat=0,i,j;
    printf("Enter total number of processes(maximum 20):");
```

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

    printf("\nEnter Process Burst Time\n");
    for(i=0;i<n;i++)
    {
        printf("P[%d]:",i+1);
        scanf("%d",&bt[i]);
    }

    wt[0]=0;    //waiting time for first process is 0

    //calculating waiting time
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];
    }

    printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");

    //calculating turnaround time
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];
        avwt+=wt[i];
        avtat+=tat[i];
        printf("\nP[%d]\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);
    }

    avwt/=i;
    avtat/=i;
    printf("\n\nAverage Waiting Time:%d",avwt);
    printf("\nAverage Turnaround Time:%d",avtat);

    return 0;
}
```





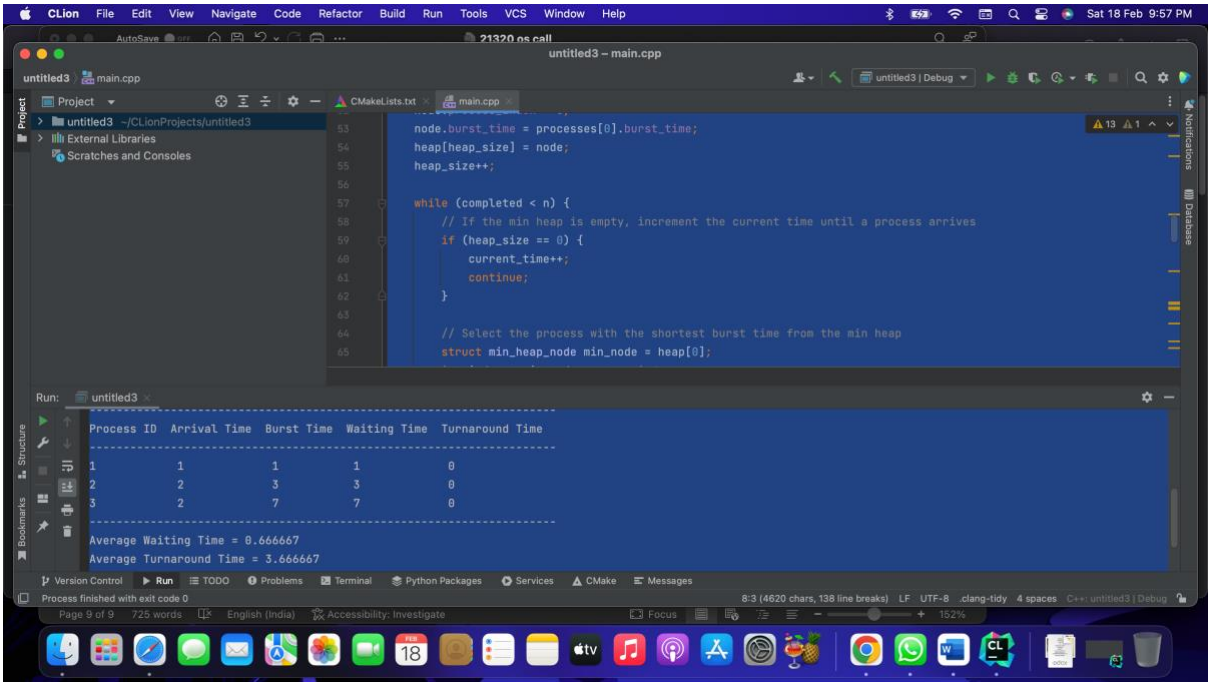
Q2 Min heap through Heap

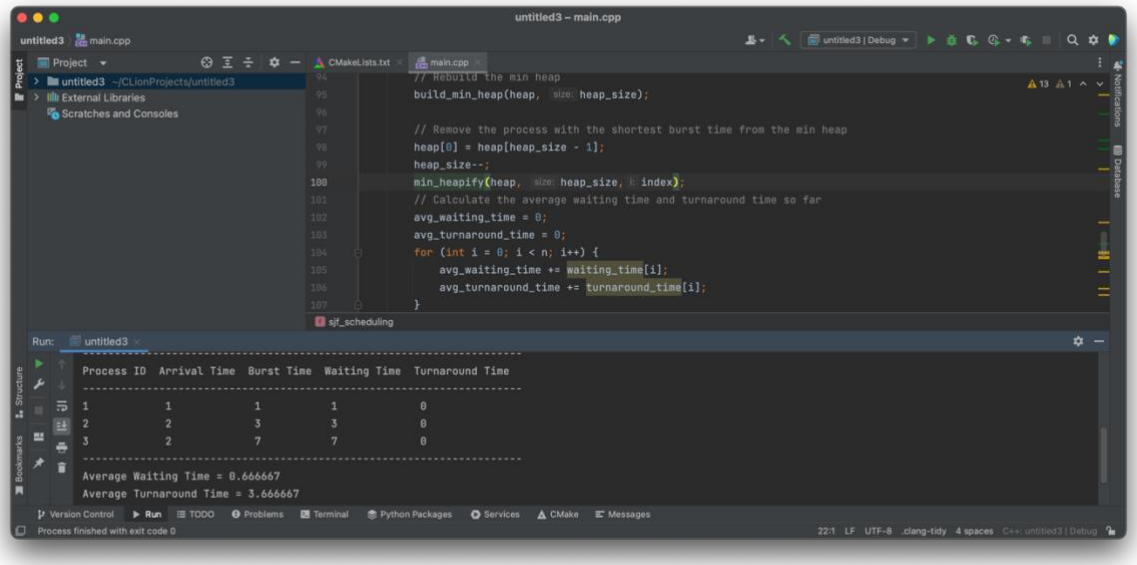
```c
#include <stdio.h>
#include <limits.h>

struct process {
```

```
    int process_id;
    int arrival_time;
    int burst_time;
};

struct min_heap_node {
    int process_index;
    int burst_time;
};

void min_heapify(struct min_heap_node* heap, int size, int i) {
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < size && heap[left].burst_time < heap[smallest].burst_time) {
        smallest = left;
    }

    if (right < size && heap[right].burst_time < heap[smallest].burst_time)
{
        smallest = right;
    }

    if (smallest != i) {
        struct min_heap_node temp = heap[i];
        heap[i] = heap[smallest];
        heap[smallest] = temp;
        min_heapify(heap, size, smallest);
    }
}

void build_min_heap(struct min_heap_node* heap, int size) {
    for (int i = (size / 2) - 1; i >= 0; i--) {
        min_heapify(heap, size, i);
    }
}

void sjf_scheduling(struct process* processes, int n) {
    int completed = 0;
    int current_time = 0;
    int heap_size = 0;
    struct min_heap_node heap[n];
    int completion_time[n], waiting_time[n], turnaround_time[n];
    float avg_waiting_time = 0, avg_turnaround_time = 0;

    // Add the first process to the min heap
    struct min_heap_node node;
    node.process_index = 0;
    node.burst_time = processes[0].burst_time;
    heap[heap_size] = node;
    heap_size++;

    while (completed < n) {
        // If the min heap is empty, increment the current time until a
process arrives
        if (heap_size == 0) {
            current_time++;
            continue;
        }
```

```
        // Select the process with the shortest burst time from the min
heap
        struct min_heap_node min_node = heap[0];
        int index = min_node.process_index;
        int burst = min_node.burst_time;

        // Update the completion time, waiting time, and turnaround time
for the selected process
        completion_time[index] = current_time + burst;
        waiting_time[index] = current_time - processes[index].arrival_time;
        turnaround_time[index] = completion_time[index] -
processes[index].arrival_time;

        // Update the current time and mark the process as completed
        current_time = completion_time[index];
        processes[index].burst_time = INT_MAX;
        completed++;

        // Add all processes that have arrived during the current burst
time to the min heap
        for (int i = 0; i < n; i++) {
            if (processes[i].arrival_time > current_time) {
                break;
            }

            if (processes[i].burst_time != INT_MAX) {
                struct min_heap_node node;
                node.process_index = i;
                node.burst_time = processes[i].burst_time;
                heap[heap_size] = node;
                heap_size++;
            }
        }

        // Rebuild the min heap
        build_min_heap(heap, heap_size);

        // Remove the process with the shortest burst time from the min
heap
        heap[0] = heap[heap_size - 1];
        heap_size--;
        min_heapify(heap, heap_size,index);
        // Calculate the average waiting time and turnaround time so far
        avg_waiting_time = 0;
        avg_turnaround_time = 0;
        for (int i = 0; i < n; i++) {
            avg_waiting_time += waiting_time[i];
            avg_turnaround_time += turnaround_time[i];
        }
        avg_waiting_time /= n;
        avg_turnaround_time /= n;
    }

    // Print the output
    printf("\nSJF Scheduling Algorithm\n");
    printf("-------------------------------------------------------------
------\n");
    printf("Process ID  Arrival Time  Burst Time  Waiting Time  Turnaround
Time\n");
    printf("-------------------------------------------------------------
------\n");
```

```
    for (int i = 0; i < n; i++) {
        printf("%-11d  %-12d  %-10d  %-12d  %-15d\n",
processes[i].process_id, processes[i].arrival_time,
               completion_time[i] - processes[i].arrival_time,
turnaround_time[i]);
    }
    printf("-----------------------------------------------------------------
------\n");
    printf("Average Waiting Time = %f\n", avg_waiting_time);
    printf("Average Turnaround Time = %f\n", avg_turnaround_time);
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct process processes[n];
    for (int i = 0; i < n; i++) {
        printf("Enter the arrival time and burst time of process %d: ", i +
1);
        scanf("%d %d", &processes[i].arrival_time,
&processes[i].burst_time);
        processes[i].process_id = i + 1;
    }
    sjf_scheduling(processes, n);
    return 0;
}
```


CLion File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
Sat 18 Feb 9:57 PM
untitled3 – main.cpp
node.burst_time = processes[0].burst_time;
heap[heap_size] = node;
heap_size++;
while (completed < n) {
// If the min heap is empty, increment the current time until a process arrives
if (heap_size == 0) {
current_time++;
continue;
}
// Select the process with the shortest burst time from the min heap
struct min_heap_node min_node = heap[0];
Run: untitled3
Process ID Arrival Time Burst Time Waiting Time Turnaround Time
1 1 1 1 0
2 2 3 3 0
3 2 7 7 0
Average Waiting Time = 0.666667
Average Turnaround Time = 3.666667
Process finished with exit code 0


untitled3 – main.cpp
// Rebuild the min heap
build_min_heap(heap, heap_size);
// Remove the process with the shortest burst time from the min heap
heap[0] = heap[heap_size - 1];
heap_size--;
min_heapify(heap, heap_size, index);
// Calculate the average waiting time and turnaround time so far
avg_waiting_time = 0;
avg_turnaround_time = 0;
for (int i = 0; i < n; i++) {
avg_waiting_time += waiting_time[i];
avg_turnaround_time += turnaround_time[i];
}
Run: untitled3
Process ID Arrival Time Burst Time Waiting Time Turnaround Time
1 1 1 1 0
2 2 3 3 0
3 2 7 7 0
Average Waiting Time = 0.666667
Average Turnaround Time = 3.666667
Process finished with exit code 0

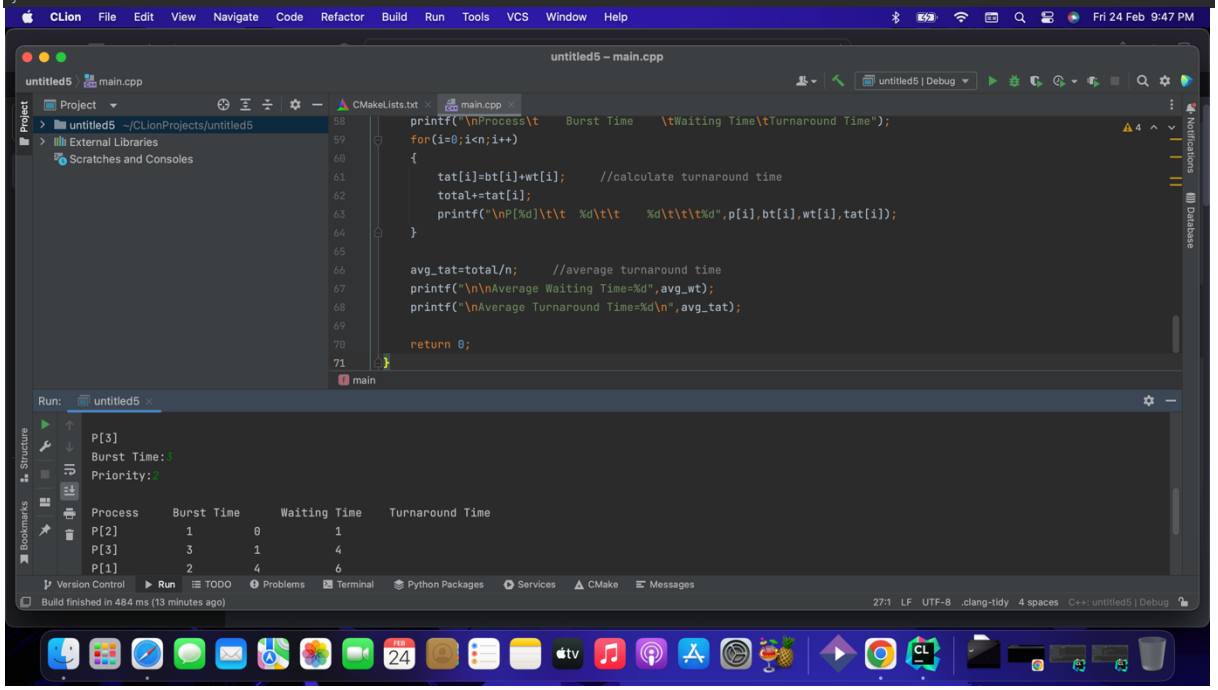
Lab 3

Aim: Create simple programs implementing RR and Priority scheduling using same and different ATs.

Theory: RR scheduling is a preemptive scheduling algorithm in which each process is given a fixed time slice or quantum to execute before it is preempted and moved to the end of the ready queue. The next process in the queue is then given a chance to run for the same quantum. This cycle of scheduling continues until all processes have completed their execution. RR scheduling ensures that each process gets a fair share of the CPU time, and it is commonly used in time-sharing systems.

```c
#include<stdio.h>

int main()
{
    int
bt[20],p[20],wt[20],tat[20],pr[20],i,j,n,total=0,pos,temp,avg_wt,avg_tat;
    printf("Enter Total Number of Process:");
    scanf("%d",&n);

    printf("\nEnter Burst Time and Priority\n");
    for(i=0;i<n;i++)
    {
        printf("\nP[%d]\n",i+1);
        printf("Burst Time:");
        scanf("%d",&bt[i]);
        printf("Priority:");
        scanf("%d",&pr[i]);
        p[i]=i+1;           //contains process number
    }

    //sorting burst time, priority and process number in ascending order
using selection sort
    for(i=0;i<n;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(pr[j]<pr[pos])
                pos=j;
        }

        temp=pr[i];
        pr[i]=pr[pos];
        pr[pos]=temp;

        temp=bt[i];
        bt[i]=bt[pos];
        bt[pos]=temp;
```

```
        temp=p[i];
        p[i]=p[pos];
        p[pos]=temp;
    }

    wt[0]=0; //waiting time for first process is zero

    //calculate waiting time
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];

        total+=wt[i];
    }

    avg_wt=total/n;       //average waiting time
    total=0;

    printf("\nProcess\t    Burst Time    \tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];     //calculate turnaround time
        total+=tat[i];
        printf("\nP[%d]\t\t  %d\t\t
%d\t\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }

    avg_tat=total/n;      //average turnaround time
    printf("\n\nAverage Waiting Time=%d",avg_wt);
    printf("\nAverage Turnaround Time=%d\n",avg_tat);

    return 0;
}
```

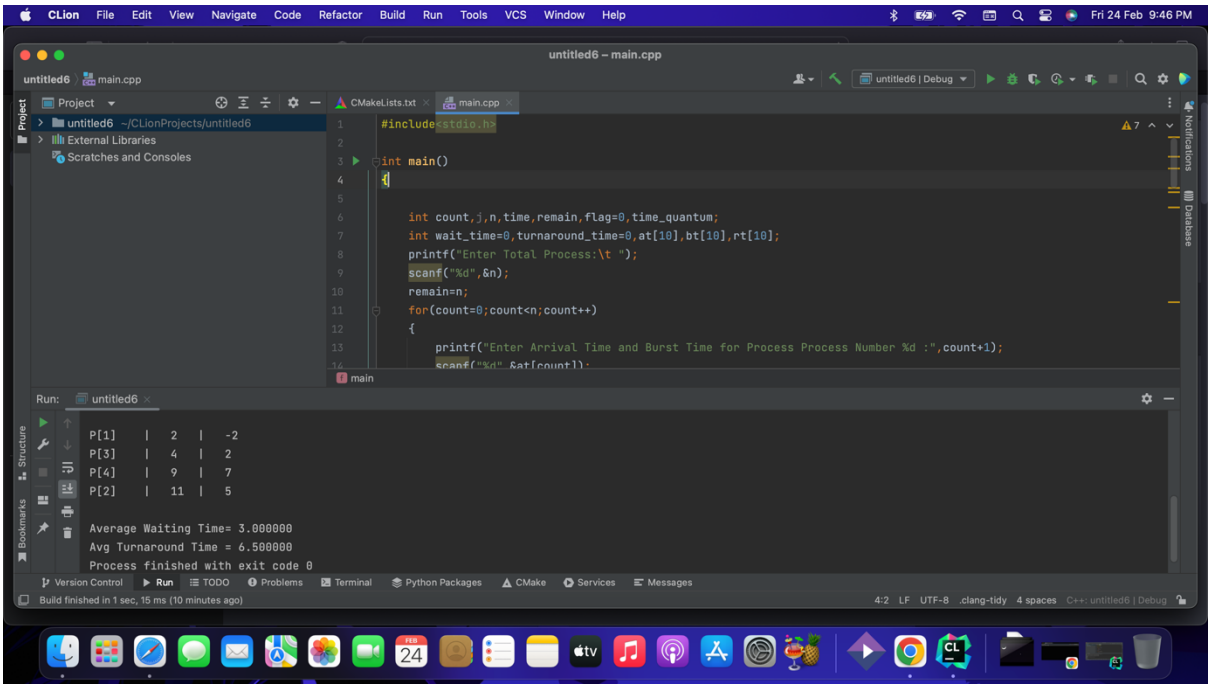


Q2. AIM :-Priority Scheduling

Theory:- Priority scheduling, on the other hand, is a non-preemptive scheduling algorithm in which each process is assigned a priority level based on factors such as its importance, its time sensitivity, and the resources it requires. The process with the highest priority is executed first, followed by the next highest priority process and so on. If two or more processes have the same priority, then they are scheduled in a round-robin manner. Priority scheduling ensures that high-priority processes get more CPU time and are completed in a timely manner.

```
#include<stdio.h>

int main()
{

    int count,j,n,time,remain,flag=0,time_quantum;
    int wait_time=0,turnaround_time=0,at[10],bt[10],rt[10];
    printf("Enter Total Process:\t ");
    scanf("%d",&n);
    remain=n;
    for(count=0;count<n;count++)
    {
        printf("Enter Arrival Time and Burst Time for Process Process
Number %d :",count+1);
        scanf("%d",&at[count]);
        scanf("%d",&bt[count]);
        rt[count]=bt[count];
    }
    printf("Enter Time Quantum:\t");
    scanf("%d",&time_quantum);
    printf("\n\nProcess\t|Turnaround Time|Waiting Time\n\n");
    for(time=0,count=0;remain!=0;)
    {
        if(rt[count]<=time_quantum && rt[count]>0)
        {
            time+=rt[count];
            rt[count]=0;
            flag=1;
        }
        else if(rt[count]>0)
        {
            rt[count]-=time_quantum;
            time+=time_quantum;
        }
        if(rt[count]==0 && flag==1)
        {
            remain--;
            printf("P[%d]\t|\t%d\t|\t%d\n",count+1,time-at[count],time-
at[count]-bt[count]);
            wait_time+=time-at[count]-bt[count];
```

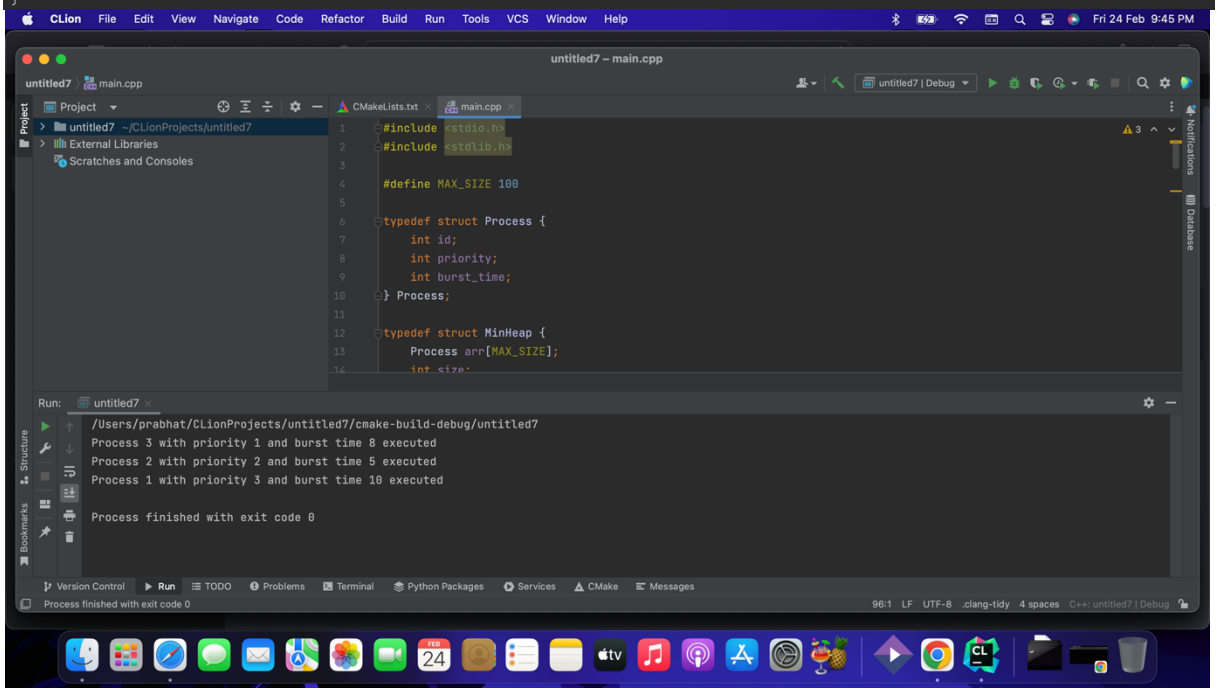
```
            turnaround_time+=time-at[count];
            flag=0;
        }
        if(count==n-1)
            count=0;
        else if(at[count+1]<=time)
            count++;
        else
            count=0;
    }
    printf("\nAverage Waiting Time= %f\n",wait_time*1.0/n);
    printf("Avg Turnaround Time = %f",turnaround_time*1.0/n);

    return 0;
}
```



Q3. Preemptive scheduling with MinHeaps

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

#define MAX_SIZE 100

typedef struct Process {
    int id;
    int priority;
    int burst_time;
} Process;

typedef struct MinHeap {
    Process arr[MAX_SIZE];
    int size;
} MinHeap;

void swap(Process *a, Process *b) {
    Process temp = *a;
    *a = *b;
    *b = temp;
}

void min_heapify(MinHeap *heap, int i) {
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < heap->size && heap->arr[left].priority < heap-
>arr[smallest].priority) {
        smallest = left;
    }

    if (right < heap->size && heap->arr[right].priority < heap-
>arr[smallest].priority) {
        smallest = right;
    }

    if (smallest != i) {
        swap(&heap->arr[i], &heap->arr[smallest]);
        min_heapify(heap, smallest);
    }
}

void build_min_heap(MinHeap *heap) {
    int i;
    for (i = heap->size / 2 - 1; i >= 0; i--) {
        min_heapify(heap, i);
    }
}

void insert_process(MinHeap *heap, int id, int priority, int burst_time) {
    if (heap->size == MAX_SIZE) {
        printf("Heap overflow!\n");
        return;
    }

    Process p = {id, priority, burst_time};
    heap->arr[heap->size++] = p;

    int i = heap->size - 1;
```

```
    while (i != 0 && heap->arr[(i - 1) / 2].priority >
heap>arr[i].priority) {
        swap(&heap->arr[(i - 1) / 2], &heap->arr[i]);
        i = (i - 1) / 2;
    }
}

Process pop_min_process(MinHeap *heap) {
    if (heap->size == 0) {
        printf("Heap underflow!\n");
        exit(1);
    }

    Process min = heap->arr[0];
    heap->arr[0] = heap->arr[heap->size - 1];
    heap->size--;

    min_heapify(heap, 0);

    return min;
}

int main() {
    MinHeap heap = {{0}, 0};

    insert_process(&heap, 1, 3, 10);
    insert_process(&heap, 2, 2, 5);
    insert_process(&heap, 3, 1, 8);

    build_min_heap(&heap);

    while (heap.size > 0) {
        Process p = pop_min_process(&heap);
        printf("Process %d with priority %d and burst time %d executed\n",
p.id, p.priority, p.burst_time);
    }

    return 0;
}
```



Lab 4

Aim: Perform MLQ scheduling with different Algorithms and with different ATs.

Theory: MLQ (Multi-Level Queue) scheduling is a process of dividing the ready queue into several separate queues with different priorities. Each queue has its own scheduling algorithm to decide which process should be executed next.Basic algorithms that drive the MLQ scheduling is that algorithm used in the process. Basic alogrithms are:-

1. First-Come, First-Served (FCFS): In this algorithm, the process that arrives first in the ready queue is executed first. It is a non-pre-emptive scheduling algorithm, which means that once a process starts executing, it will continue until it finishes or blocks.

2. Shortest Job First (SJF): In this algorithm, the process with the shortest CPU burst time is executed first. It is also a non-pre-emptive scheduling algorithm.

3. Priority Scheduling: In this algorithm, each process is assigned a priority based on its importance. The process with the highest priority is executed first. It can be either preemptive or non-pre-emptive.

4.Round Robin (RR): In this algorithm, each process is given a time slice or quantum to execute. Once the time slice is over, the process is preempted and added back to the end of the ready queue. It is a preemptive scheduling algorithm.

CODE

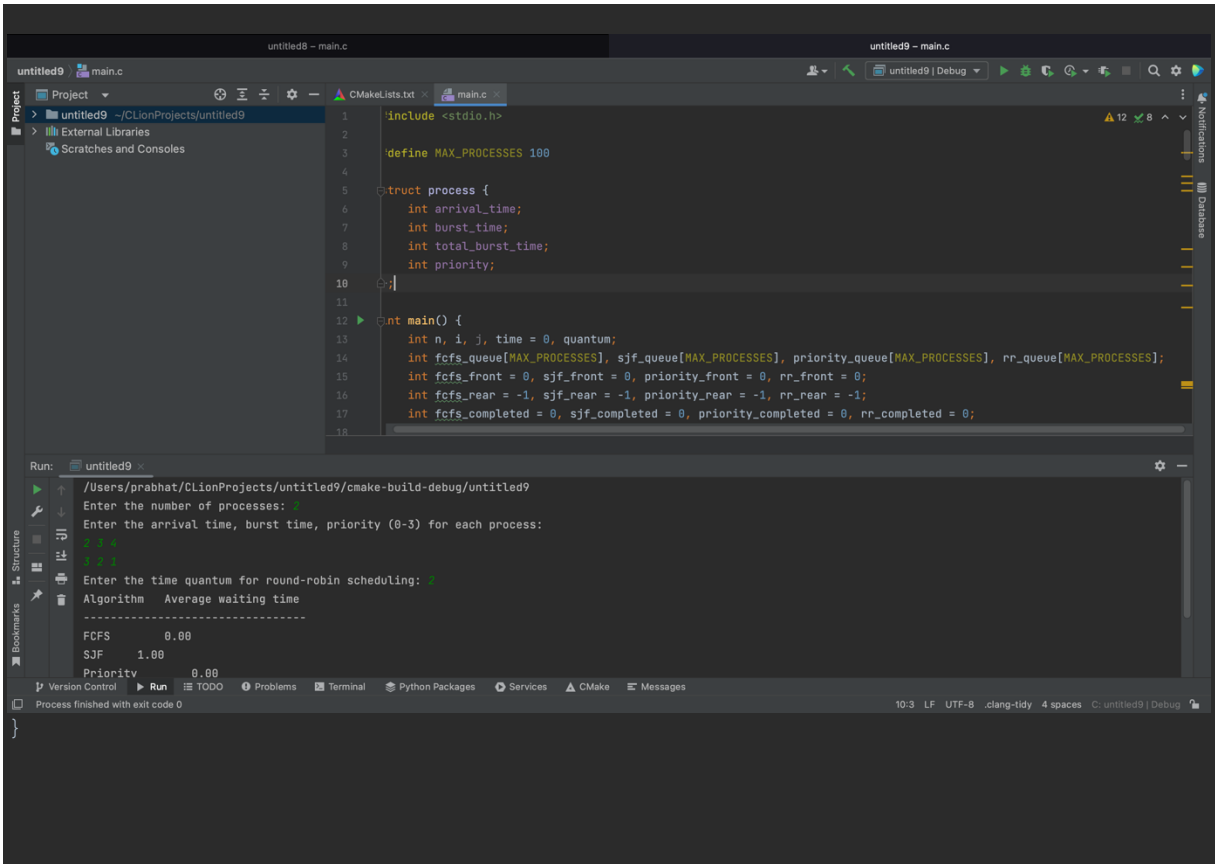
```
#include <stdio.h>
```

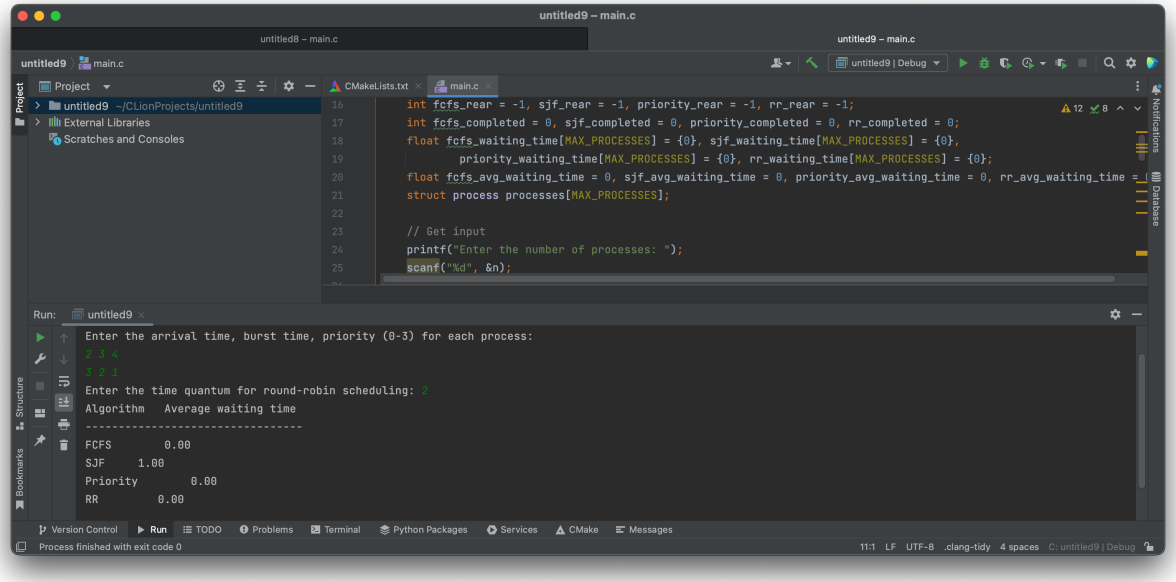
```c
#define MAX_PROCESSES 100

struct process {
    int arrival_time;
    int burst_time;
    int total_burst_time;
    int priority;
};

int main() {
    int n, i, j, time = 0, quantum;
    int fcfs_queue[MAX_PROCESSES], sjf_queue[MAX_PROCESSES],
priority_queue[MAX_PROCESSES], rr_queue[MAX_PROCESSES];
    int fcfs_front = 0, sjf_front = 0, priority_front = 0, rr_front = 0;
    int fcfs_rear = -1, sjf_rear = -1, priority_rear = -1, rr_rear = -1;
    int fcfs_completed = 0, sjf_completed = 0, priority_completed = 0,
rr_completed = 0;
    float fcfs_waiting_time[MAX_PROCESSES] = {0},
sjf_waiting_time[MAX_PROCESSES] = {0},
          priority_waiting_time[MAX_PROCESSES] = {0},
rr_waiting_time[MAX_PROCESSES] = {0};
    float fcfs_avg_waiting_time = 0, sjf_avg_waiting_time = 0,
priority_avg_waiting_time = 0, rr_avg_waiting_time = 0;
    struct process processes[MAX_PROCESSES];

    // Get input
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    printf("Enter the arrival time, burst time, priority (0-3) for each
process:\n");
    for (i = 0; i < n; i++) {
        scanf("%d %d %d", &processes[i].arrival_time,
&processes[i].burst_time, &processes[i].priority);
        processes[i].total_burst_time = processes[i].burst_time;
    }
    printf("Enter the time quantum for round-robin scheduling: ");
    scanf("%d", &quantum);

    // Add each process to the appropriate queue based on priority
    for (i = 0; i < n; i++) {
        if (processes[i].arrival_time <= time && processes[i].burst_time >
0) {
            if (processes[i].priority == 0) {
                fcfs_rear++;
                fcfs_queue[fcfs_rear] = i;
            } else if (processes[i].priority == 1) {
                sjf_rear++;
                sjf_queue[sjf_rear] = i;
            } else if (processes[i].priority == 2) {
                priority_rear++;
                priority_queue[priority_rear] = i;
            } else if (processes[i].priority == 3) {
                rr_rear++;
                rr_queue[rr_rear] = i;
            }
        }
    }

    // Run each queue until all processes are completed
    while (fcfs_completed + sjf_completed + priority_completed +
rr_completed < n) {
```

```
        // FCFS queue
        if (fcfs_rear >= fcfs_front) {
            int pid = fcfs_queue[fcfs_front];
            fcfs_front++;
            fcfs_completed++;
            time += processes[pid].burst_time;
            fcfs_waiting_time[pid] = time - processes[pid].arrival_time -
processes[pid].burst_time;
            fcfs_avg_waiting_time += fcfs_waiting_time[pid];
        }
            // SJF queue
        else if (sjf_rear >= sjf_front) {
            int pid = sjf_queue[sjf_front];
            sjf_front++;
            sjf_completed++;
            time += processes[pid].burst_time;
            sjf_waiting_time[pid] = time - processes[pid].arrival_time -
processes[pid].burst_time;
            sjf_avg_waiting_time += sjf_waiting_time[pid];
        }
            // Priority queue
        else if (priority_rear >= priority_front) {
            int pid = priority_queue[priority_front];
            priority_front++;
            priority_completed++;
            time += processes[pid].burst_time;
            priority_waiting_time[pid] = time - processes[pid].arrival_time
- processes[pid].burst_time;
            priority_avg_waiting_time += priority_waiting_time[pid];
        }
            // Round-robin queue
        else if (rr_rear >= rr_front) {
            int pid = rr_queue[rr_front];
            rr_front++;
            // If remaining burst time is less than or equal to time
quantum, complete the process
            if (processes[pid].burst_time <= quantum) {
                rr_completed++;
                time += processes[pid].burst_time;
                rr_waiting_time[pid] = time - processes[pid].arrival_time -
processes[pid].total_burst_time;
                rr_avg_waiting_time += rr_waiting_time[pid];
            }
                // If remaining burst time is greater than time quantum,
add process back to queue
            else {
                time += quantum;
                processes[pid].burst_time -= quantum;
                rr_rear++;
                rr_queue[rr_rear] = pid;
            }
        }
            // If all queues are empty, increment time
        else {
            time++;
        }

        // Add any new processes to the appropriate queue
        for (i = 0; i < n; i++) {
            if (processes[i].arrival_time <= time &&
processes[i].burst_time > 0) {
```

```
                if (processes[i].priority == 0) {
                    fcfs_rear++;
                    fcfs_queue[fcfs_rear] = i;
                } else if (processes[i].priority == 1) {
                    sjf_rear++;
                    sjf_queue[sjf_rear] = i;
                } else if (processes[i].priority == 2) {
                    priority_rear++;
                    priority_queue[priority_rear] = i;
                } else if (processes[i].priority == 3) {
                    rr_rear++;
                    rr_queue[rr_rear] = i;
                }
            }
        }
    }

// Calculate average waiting time for each algorithm
    fcfs_avg_waiting_time /= n;
    sjf_avg_waiting_time /= n;
    priority_avg_waiting_time /= n;
    rr_avg_waiting_time /= n;

// Print results
    printf("Algorithm\tAverage waiting time\n");
    printf("---------------------------------\n");
    printf("FCFS\t\t%.2f\n", fcfs_avg_waiting_time);
    printf("SJF\t\t%.2f\n", sjf_avg_waiting_time);
    printf("Priority\t\t%.2f\n", priority_avg_waiting_time);
    printf("RR\t\t   %.2f\n", rr_avg_waiting_time);

    return 0;
```

Output


untitled8 – main.c
untitled9 – main.c
untitled9 | Debug
Project
untitled9 ~/CLionProjects/untitled9
External Libraries
Scratches and Consoles
CMakeLists.txt
main.c
#include <stdio.h>
#define MAX_PROCESSES 100
struct process {
int arrival_time;
int burst_time;
int total_burst_time;
int priority;
};
int main() {
int n, i, j, time = 0, quantum;
int fcfs_queue[MAX_PROCESSES], sjf_queue[MAX_PROCESSES], priority_queue[MAX_PROCESSES], rr_queue[MAX_PROCESSES];
int fcfs_front = 0, sjf_front = 0, priority_front = 0, rr_front = 0;
int fcfs_rear = -1, sjf_rear = -1, priority_rear = -1, rr_rear = -1;
int fcfs_completed = 0, sjf_completed = 0, priority_completed = 0, rr_completed = 0;
Run: untitled9
/Users/prabhat/CLionProjects/untitled9/cmake-build-debug/untitled9
Enter the number of processes: 2
Enter the arrival time, burst time, priority (0-3) for each process:
2 3 4
3 2 1
Enter the time quantum for round-robin scheduling: 2
Algorithm Average waiting time
FCFS 0.00
SJF 1.00
Priority 0.00
Version Control
Run
TODO
Problems
Terminal
Python Packages
Services
CMake
Messages
Process finished with exit code 0
10:3 LF UTF-8 .clang-tidy 4 spaces C: untitled9 | Debug
}


untitled9 – main.c
untitled8 – main.c
untitled9 – main.c
untitled9 | Debug
Project
untitled9 ~/CLionProjects/untitled9
External Libraries
Scratches and Consoles
CMakeLists.txt
main.c
int fcfs_rear = -1, sjf_rear = -1, priority_rear = -1, rr_rear = -1;
int fcfs_completed = 0, sjf_completed = 0, priority_completed = 0, rr_completed = 0;
float fcfs_waiting_time[MAX_PROCESSES] = {0}, sjf_waiting_time[MAX_PROCESSES] = {0},
priority_waiting_time[MAX_PROCESSES] = {0}, rr_waiting_time[MAX_PROCESSES] = {0};
float fcfs_avg_waiting_time = 0, sjf_avg_waiting_time = 0, priority_avg_waiting_time = 0, rr_avg_waiting_time =
struct process processes[MAX_PROCESSES];
// Get input
printf("Enter the number of processes: ");
scanf("%d", &n);
Run: untitled9
Enter the arrival time, burst time, priority (0-3) for each process:
2 3 4
3 2 1
Enter the time quantum for round-robin scheduling: 2
Algorithm Average waiting time
FCFS 0.00
SJF 1.00
Priority 0.00
RR 0.00
Version Control
Run
TODO
Problems
Terminal
Python Packages
Services
CMake
Messages
Process finished with exit code 0
11:1 LF UTF-8 .clang-tidy 4 spaces C: untitled9 | Debug

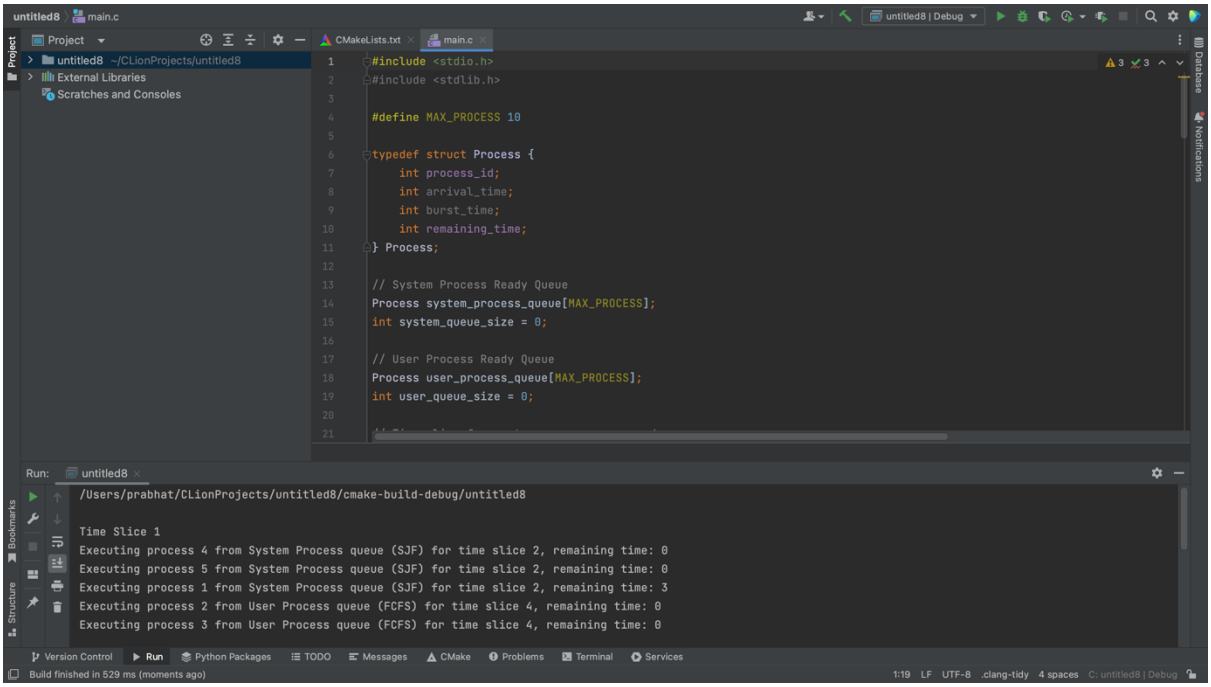
Q2 Develop MLQ, where a Time slice is given to each queue i.e., each queue gets a certain amount of CPU time(assume any) that it can schedule amongst its processes.

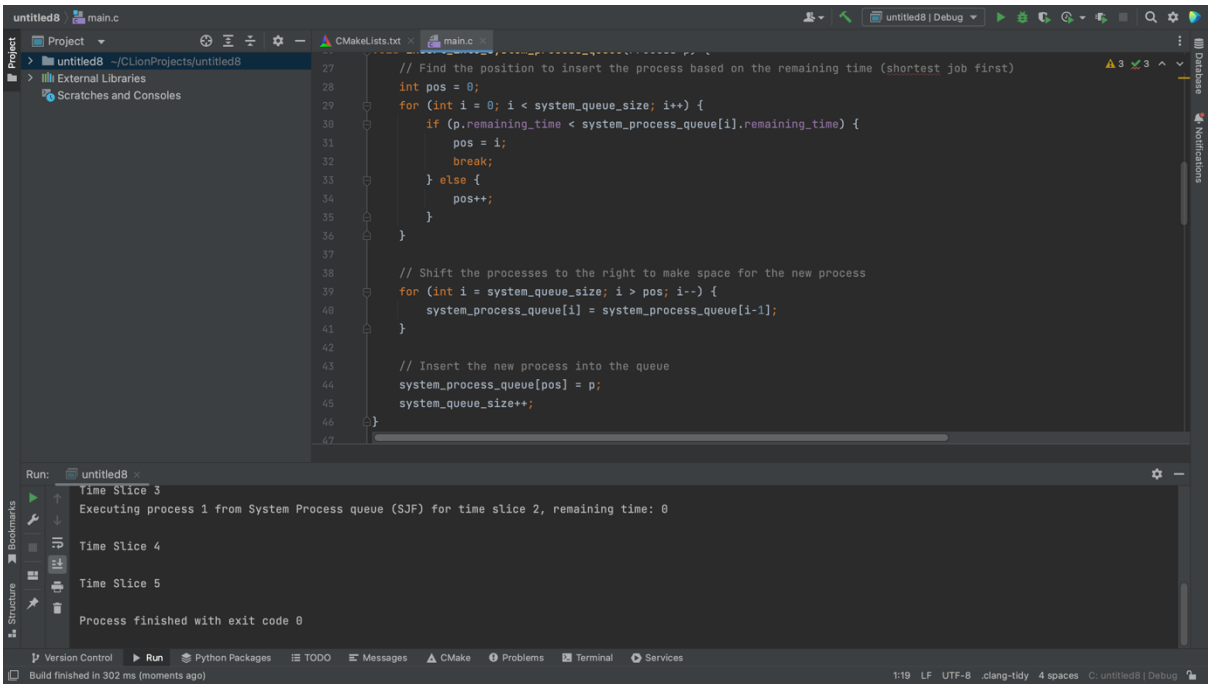
CODE

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

#define MAX_PROCESS 10

typedef struct Process {
    int process_id;
    int arrival_time;
    int burst_time;
    int remaining_time;
} Process;

// System Process Ready Queue
Process system_process_queue[MAX_PROCESS];
int system_queue_size = 0;

// User Process Ready Queue
Process user_process_queue[MAX_PROCESS];
int user_queue_size = 0;

// Time slice for system process queue and user process queue
int system_time_slice = 2;
int user_time_slice = 4;

// Insert a process into the system process queue
void insert_into_system_process_queue(Process p) {
    // Find the position to insert the process based on the remaining time
(shortest job first)
    int pos = 0;
    for (int i = 0; i < system_queue_size; i++) {
        if (p.remaining_time < system_process_queue[i].remaining_time) {
            pos = i;
            break;
        } else {
            pos++;
        }
    }

    // Shift the processes to the right to make space for the new process
    for (int i = system_queue_size; i > pos; i--) {
        system_process_queue[i] = system_process_queue[i-1];
    }

    // Insert the new process into the queue
    system_process_queue[pos] = p;
    system_queue_size++;
}

// Insert a process into the user process queue
```

```
void insert_into_user_process_queue(Process p) {
    // Insert the new process at the end of the queue
    user_process_queue[user_queue_size] = p;
    user_queue_size++;
}

// Execute all the processes in the system process queue and user process
queue for their respective time slice
void execute_all_queues() {
    // Execute the System Process queue using SJF
    for (int i = 0; i < system_queue_size; i++) {
        Process *p = &system_process_queue[i];
        if (p->remaining_time > 0) {
            p->remaining_time -= system_time_slice;
            if (p->remaining_time < 0) {
                p->remaining_time = 0;
            }
            printf("Executing process %d from System Process queue (SJF)
for time slice %d, remaining time: %d\n", p->process_id, system_time_slice,
p->remaining_time);
        }
    }

    // Execute the User Process queue using FCFS
    for (int i = 0; i < user_queue_size; i++) {
        Process *p = &user_process_queue[i];
        if (p->remaining_time > 0) {
            p->remaining_time -= user_time_slice;
            if (p->remaining_time < 0) {
                p->remaining_time = 0;
            }
            printf("Executing process %d from User Process queue (FCFS) for
time slice %d, remaining time: %d\n", p->process_id, user_time_slice, p-
>remaining_time);
        }
    }
}

int main() {
    // Example processes
    Process p1 = {1, 0, 5, 5};
    Process p2 = {2, 1, 3, 3};
    Process p3 = {3, 2, 4, 4};
    Process p4 = {4, 3, 2, 2};
    Process p5 = {5, 4, 1, 2};
    // Insert processes into their respective queues
insert_into_system_process_queue(p1);
insert_into_user_process_queue(p2);
insert_into_user_process_queue(p3);
insert_into_system_process_queue(p4);
insert_into_system_process_queue(p5);

// Execute the queues for a few time slices
for (int i = 0; i < 5; i++) {
    printf("\nTime Slice %d\n", i+1);
    execute_all_queues();
}

return 0;
}
```

Output


#include <stdio.h>
#include <stdlib.h>
#define MAX_PROCESS 10
typedef struct Process {
int process_id;
int arrival_time;
int burst_time;
int remaining_time;
} Process;
// System Process Ready Queue
Process system_process_queue[MAX_PROCESS];
int system_queue_size = 0;
// User Process Ready Queue
Process user_process_queue[MAX_PROCESS];
int user_queue_size = 0;
/Users/prabhat/CLionProjects/untitled8/cmake-build-debug/untitled8
Time Slice 1
Executing process 4 from System Process queue (SJF) for time slice 2, remaining time: 0
Executing process 5 from System Process queue (SJF) for time slice 2, remaining time: 0
Executing process 1 from System Process queue (SJF) for time slice 2, remaining time: 3
Executing process 2 from User Process queue (FCFS) for time slice 4, remaining time: 0
Executing process 3 from User Process queue (FCFS) for time slice 4, remaining time: 0


// Find the position to insert the process based on the remaining time (shortest job first)
int pos = 0;
for (int i = 0; i < system_queue_size; i++) {
if (p.remaining_time < system_process_queue[i].remaining_time) {
pos = i;
break;
} else {
pos++;
}
}
// Shift the processes to the right to make space for the new process
for (int i = system_queue_size; i > pos; i--) {
system_process_queue[i] = system_process_queue[i-1];
}
// Insert the new process into the queue
system_process_queue[pos] = p;
system_queue_size++;
}
Time Slice 3
Executing process 1 from System Process queue (SJF) for time slice 2, remaining time: 0
Time Slice 4
Time Slice 5
Process finished with exit code 0

Lab 5

Aim: Develop Producer-Consumer Synchronization with sleep() and wakeup() system calls.

Theory:-

Producer-Consumer synchronization with sleep() and wakeup() system calls can be implemented using the following steps:-

- Initialize a shared buffer that can hold a certain number of items.
- Create two semaphores, one for tracking the number of empty slots in the buffer and another for tracking the number of filled slots.
- Create two threads, one for the producer and one for the consumer.
- In the producer thread, use a while loop to generate data items and add them to the shared buffer. Before adding an item to the buffer, acquire the empty semaphore to make sure there is at least one empty slot in the buffer. Once the item is added, release the full semaphore to indicate that a slot has been filled.
- In the consumer thread, use a while loop to retrieve data items from the shared buffer. Before retrieving an item, acquire the full semaphore to make sure there is at least one filled slot in the buffer. Once the item is retrieved, release the empty semaphore to indicate that a slot has been emptied.
- Use the sleep() and wakeup() system calls to prevent busy waiting when the buffer is empty or full.

- However, using sleep() and wakeup() system calls for synchronization is not a recommended option because it can cause performance issues and lead to unpredictable behaviour in some cases. The sleep() system call can cause the calling thread to block for a certain amount of time, which can lead to delays in the execution of the program. Additionally, the wakeup() system call may not always work as expected, leading to potential deadlocks or other synchronization issues.

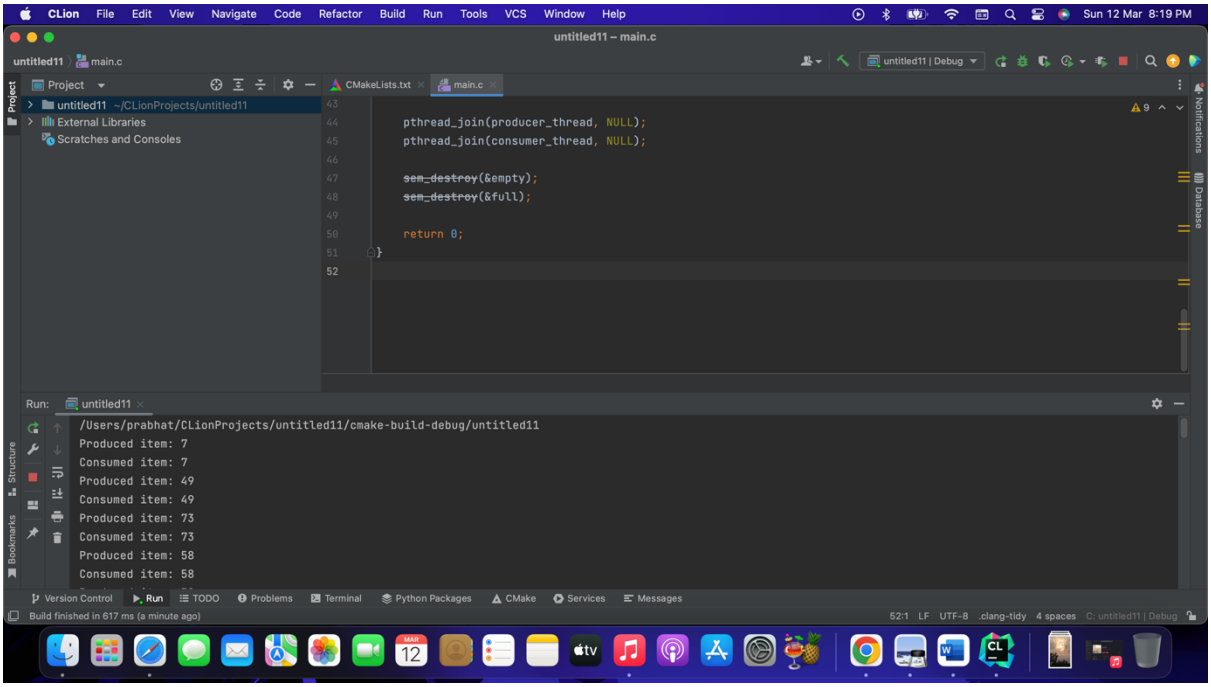
- Instead of using sleep() and wakeup() system calls, it is generally recommended to use more robust synchronization mechanisms, such as semaphores, mutexes, or condition variables. These mechanisms provide better performance and more reliable synchronization between threads, which can help to avoid potential issues with blocking or deadlocks.

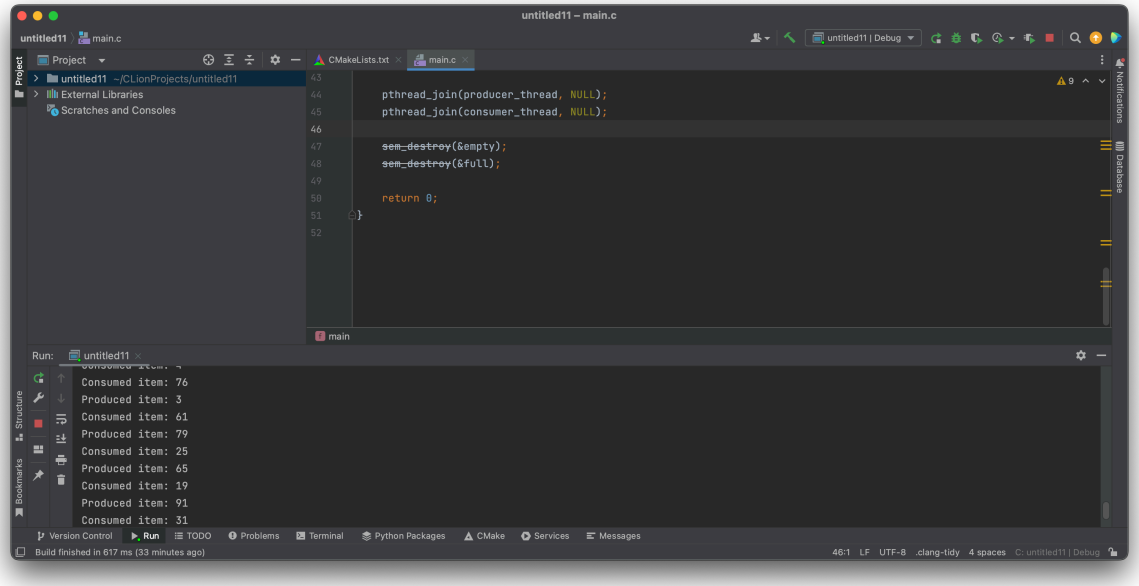
CODE:-

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

#define BUFFER_SIZE 10

sem_t empty, full;
int buffer[BUFFER_SIZE];
int in = 0, out = 0;

void *producer(void *arg) {
    while (1) {
        int item = rand() % 100;
        sem_wait(&empty);
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
        sem_post(&full);
        printf("Produced item: %d\n", item);
        sleep(1);
    }
}

void *consumer(void *arg) {
    while (1) {
        sem_wait(&full);
        int item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        sem_post(&empty);
        printf("Consumed item: %d\n", item);
        sleep(1);
    }
}

int main() {
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    pthread_t producer_thread, consumer_thread;
    pthread_create(&producer_thread, NULL, producer, NULL);
```

```
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

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

    return 0;
}
```

Output:-




untitled11 – main.c
untitled11 | Debug
Project
untitled11 ~/CLionProjects/untitled11
External Libraries
Scratches and Consoles
CMakeLists.txt
main.c
pthread_join(producer_thread, NULL);
pthread_join(consumer_thread, NULL);
sem_destroy(&empty);
sem_destroy(&full);
return 0;
}
main
Run:
untitled11
Consumed item: 76
Produced item: 3
Consumed item: 61
Produced item: 79
Consumed item: 25
Produced item: 65
Consumed item: 19
Produced item: 91
Consumed item: 31
Version Control
Run
TODO
Problems
Terminal
Python Packages
CMake
Services
Messages
Build finished in 617 ms (33 minutes ago)
46:1 LF UTF-8 .clang-tidy 4 spaces C: untitled11 | Debug

Lab 6

Aim: - Create n number of processes with different execution codes.

Theory:- A process is an instance of a running program. A process is comprised of an executable code and a set of resources such as memory, file handles, and other system resources. In many cases, it can be useful to create multiple processes to perform different tasks concurrently. This can be done by creating child processes from a parent process.

In C, the fork() system call is used to create a new process. When fork() is called, the operating system creates a new process that is a copy of the parent process. The new process has its own address space and runs independently of the parent process.

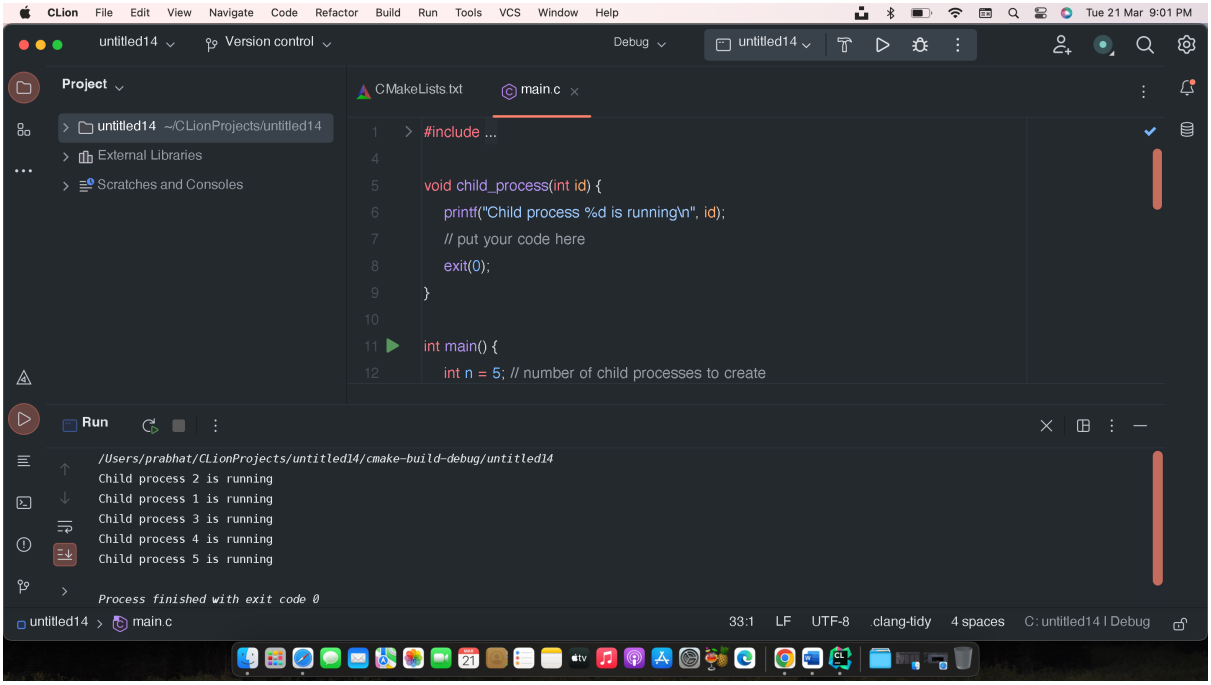
To create n number of processes with different execution codes, you can use a loop to call fork() multiple times. In each iteration of the loop, a new child process is created with a unique process ID (PID). You can then use the PID to identify each child process and execute a different code for each process.

CODE:-

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

void child_process(int id) {
    printf("Child process %d is running\n", id);
    // put your code here
    exit(0);
}

int main() {
    int n = 5; // number of child processes to create
    int i, pid;

    for (i = 1; i <= n; i++) {
        pid = fork();
        if (pid == 0) {
            child_process(i);
        } else if (pid < 0) {
            printf("Error: fork failed\n");
```

```
            exit(1);
        }
    }

    // wait for all child processes to complete
    int status;
    for (i = 1; i <= n; i++) {
        wait(&status);
    }

    return 0;
}
```

Output:-



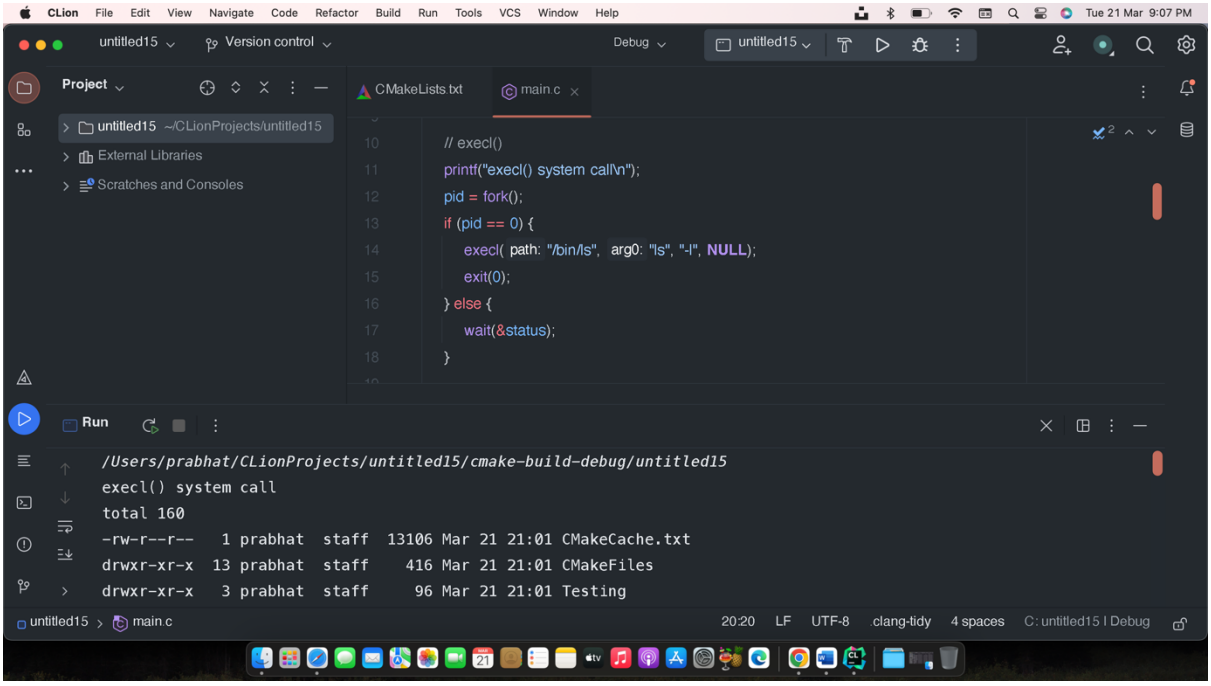
Q:- Try different system call of exec() family.

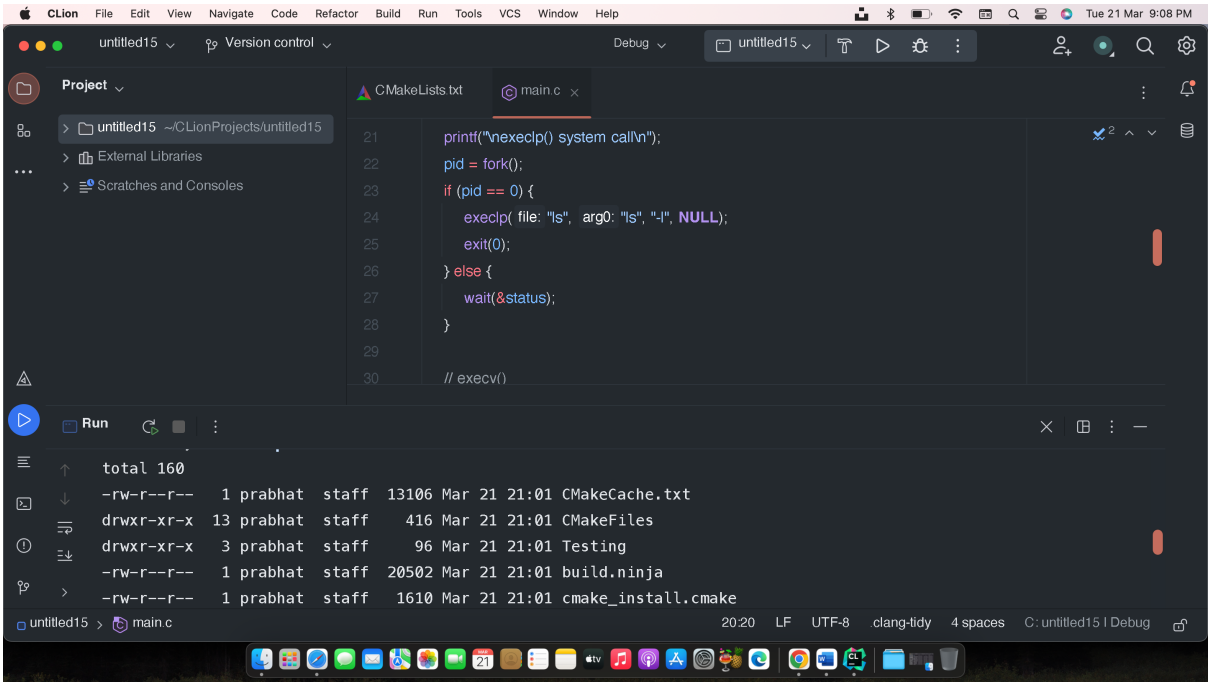
CODE:-

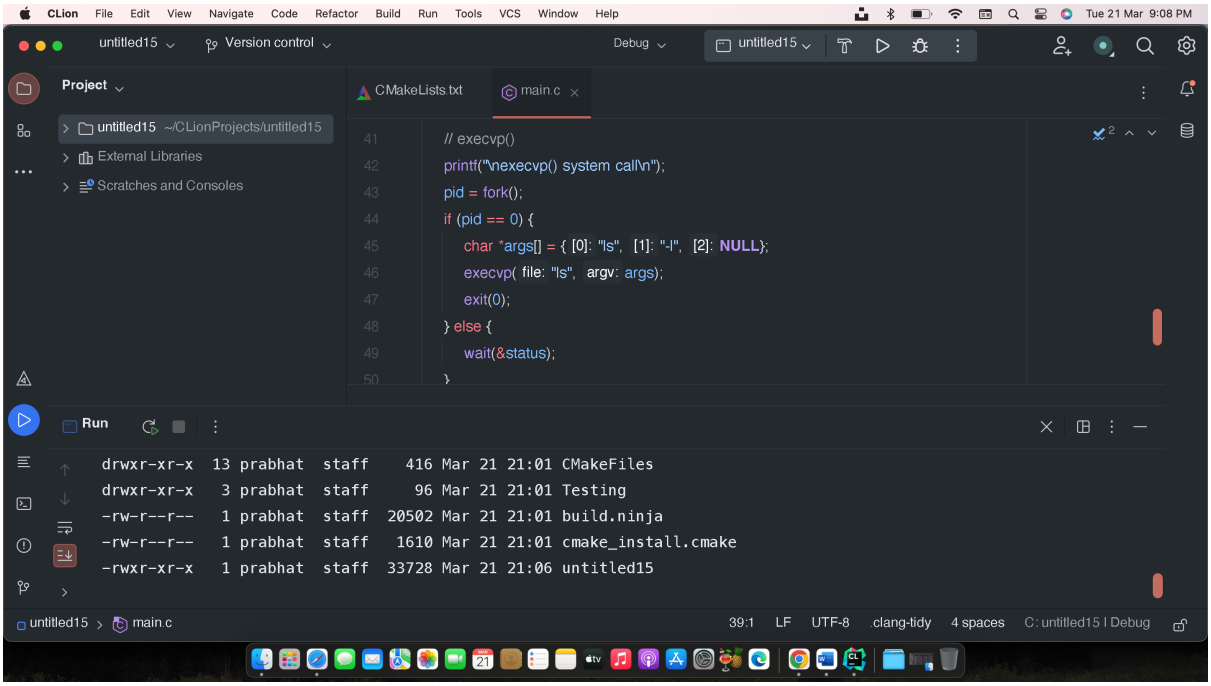
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid;
    int status;

    // execl()
    printf("execl() system call\n");
    pid = fork();
    if (pid == 0) {
        execl("/bin/ls", "ls", "-l", NULL);
        exit(0);
    } else {
        wait(&status);
    }

    // execlp()
    printf("\nexeclp() system call\n");
    pid = fork();
    if (pid == 0) {
        execlp("ls", "ls", "-l", NULL);
        exit(0);
    } else {
        wait(&status);
    }

    // execv()
    printf("\nexecv() system call\n");
    pid = fork();
    if (pid == 0) {
        char *args[] = {"ls", "-l", NULL};
        execv("/bin/ls", args);
        exit(0);
    } else {
        wait(&status);
    }

    // execvp()
    printf("\nexecvp() system call\n");
    pid = fork();
    if (pid == 0) {
        char *args[] = {"ls", "-l", NULL};
        execvp("ls", args);
        exit(0);
    } else {
        wait(&status);
    }
```

```
    return 0;
}
```

Output:-





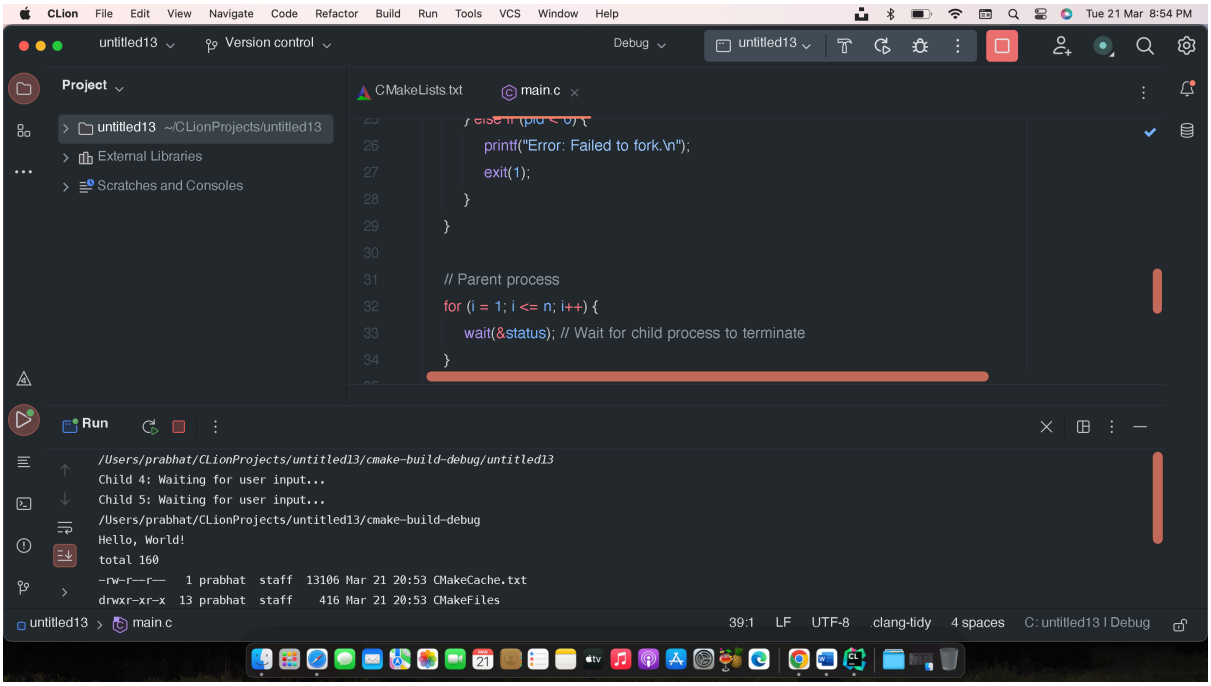


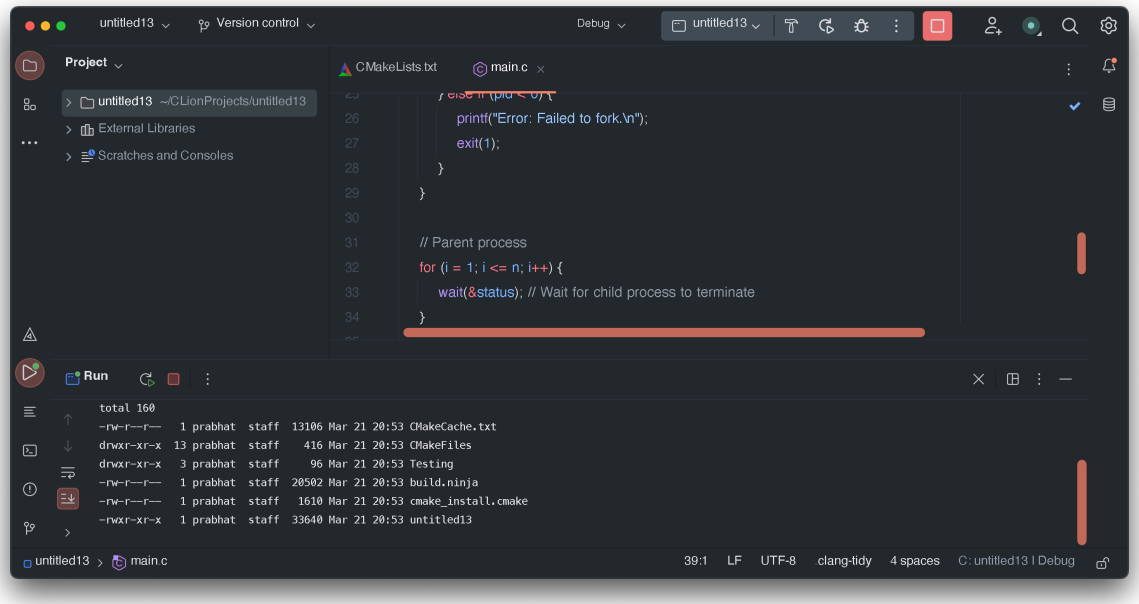
Q:- Write a code for tracing the execution of Parent and Child processes.

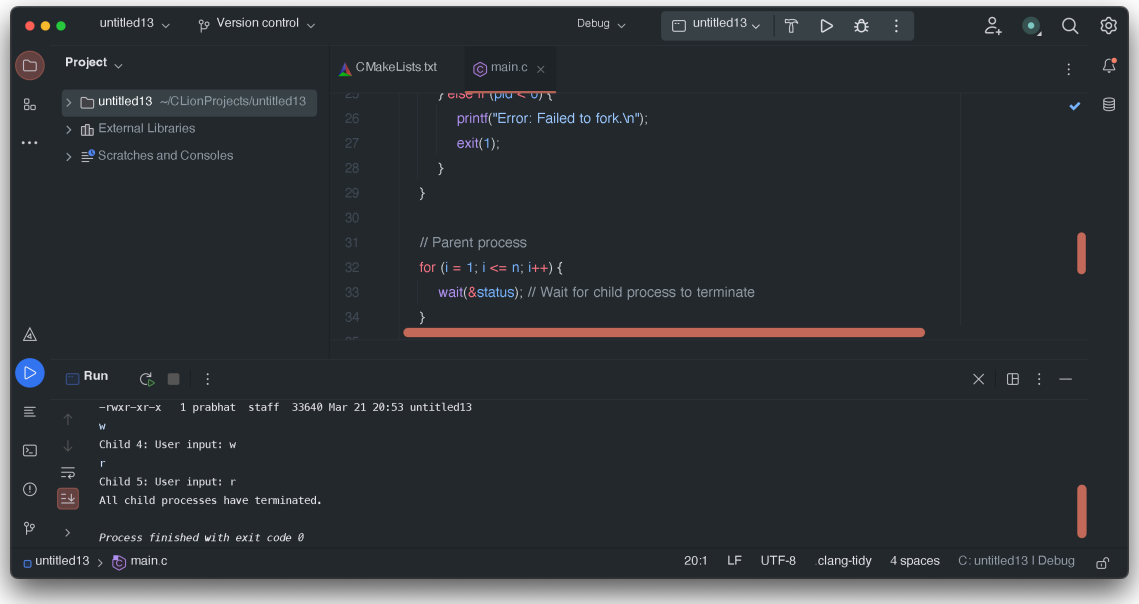
CODE:-

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

int main() {
    int n = 5; // Number of child processes to create
    int i, pid, status;

    for (i = 1; i <= n; i++) {
        pid = fork(); // Create a child process
        if (pid == 0) { // Child process
            if (i == 1) {
                execl("/bin/ls", "ls", "-l", NULL); // Execute "ls -l"
command
            } else if (i == 2) {
                execl("/bin/pwd", "pwd", NULL); // Execute "pwd" command
            } else if (i == 3) {
                execl("/bin/echo", "echo", "Hello, World!", NULL); //
Execute "echo" command
            } else {
                printf("Child %d: Waiting for user input...\n", i);
                char input[100];
                scanf("%s", input); // Wait for user input
                printf("Child %d: User input: %s\n", i, input);
            }
            exit(0); // Terminate child process
        } else if (pid < 0) {
            printf("Error: Failed to fork.\n");
            exit(1);
        }
    }

    // Parent process
    for (i = 1; i <= n; i++) {
        wait(&status); // Wait for child process to terminate
    }

    printf("All child processes have terminated.\n");
    return 0;
}
```

Output:-


CLion File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
Tue 21 Mar 8:54 PM
untitled13
Version control
Debug
untitled13
Project
untitled13 ~/CLionProjects/untitled13
External Libraries
Scratches and Consoles
CMakeLists.txt
main.c
printf("Error: Failed to fork.\n");
exit(1);
// Parent process
for (i = 1; i <= n; i++) {
wait(&status); // Wait for child process to terminate
Run
/Users/prabhat/CLionProjects/untitled13/cmake-build-debug/untitled13
Child 4: Waiting for user input...
Child 5: Waiting for user input...
/Users/prabhat/CLionProjects/untitled13/cmake-build-debug
Hello, World!
total 160
-rw-r--r-- 1 prabhat staff 13106 Mar 21 20:53 CMakeCache.txt
drwxr-xr-x 13 prabhat staff 416 Mar 21 20:53 CMakeFiles
untitled13 main.c
39:1 LF UTF-8 clang-tidy 4 spaces C: untitled13 | Debug


untitled13
Version control
Debug
untitled13
Project
untitled13 ~/CLionProjects/untitled13
External Libraries
Scratches and Consoles
CMakeLists.txt
main.c
printf("Error: Failed to fork.\n");
exit(1);
// Parent process
for (i = 1; i <= n; i++) {
wait(&status); // Wait for child process to terminate
Run
total 160
-rw-r--r-- 1 prabhat staff 13106 Mar 21 20:53 CMakeCache.txt
drwxr-xr-x 13 prabhat staff 416 Mar 21 20:53 CMakeFiles
drwxr-xr-x 3 prabhat staff 96 Mar 21 20:53 Testing
-rw-r--r-- 1 prabhat staff 20502 Mar 21 20:53 build.ninja
-rw-r--r-- 1 prabhat staff 1610 Mar 21 20:53 cmake_install.cmake
-rwxr-xr-x 1 prabhat staff 33640 Mar 21 20:53 untitled13
untitled13 main.c
39:1 LF UTF-8 clang-tidy 4 spaces C: untitled13 | Debug


untitled13
Version control
Debug
untitled13
Project
untitled13 ~/CLionProjects/untitled13
External Libraries
Scratches and Consoles
CMakeLists.txt
main.c
printf("Error: Failed to fork.\n");
exit(1);
}
}
// Parent process
for (i = 1; i <= n; i++) {
wait(&status); // Wait for child process to terminate
}
Run
-rwxr-xr-x 1 prabhat staff 33640 Mar 21 20:53 untitled13
w
Child 4: User input: w
r
Child 5: User input: r
All child processes have terminated.
Process finished with exit code 0
untitled13
main.c
20:1
LF
UTF-8
clang-tidy
4 spaces
C: untitled13 | Debug

Lab 7

Aim: - Implement the entire mechanism of Deadlock Avoidance by depicting the fulfilment of necessary condition and sufficient conditions of deadlock avoidance.

Theory:- Deadlock avoidance is a method used to prevent a deadlock from occurring in a computer system. Deadlock is a situation in which two or more processes are unable to continue executing because each is waiting for one of the others to do something. Deadlock avoidance is achieved by ensuring that the system never enters a state where a deadlock can occur. There are two conditions that must be satisfied for a system to avoid deadlock:-

Necessary Condition: - Mutual Exclusion

This condition states that at least one resource must be held in a non-sharable mode by each process. In other words, if a process is using a resource, no other process should be able to use that resource at the same time. This condition is necessary for a deadlock to occur, but it is not sufficient to prevent a deadlock.

These conditions must be fulfilled to prevent a deadlock:-

- Hold and Wait: A process must request all the resources it needs at once, rather than one at a time. This way, the process will not hold onto a resource while waiting for another resource to become available.
- No Pre-emption: Once a process is allocated a resource, it cannot be taken away until the process has completed its work on that resource. This

ensures that a process will not be forced to give up a resource that it needs to complete its work.

- Circular Wait: A circular chain of processes must not exist where each process is waiting for a resource held by the next process in the chain. This condition can be avoided by imposing a total ordering of all resource types and ensuring that each process requests resources in an increasing order of their numbering.

To implement the deadlock avoidance mechanism, the system must use a resource allocation algorithm that checks each time a process requests a resource whether granting the request would result in a deadlock.

CODE:-

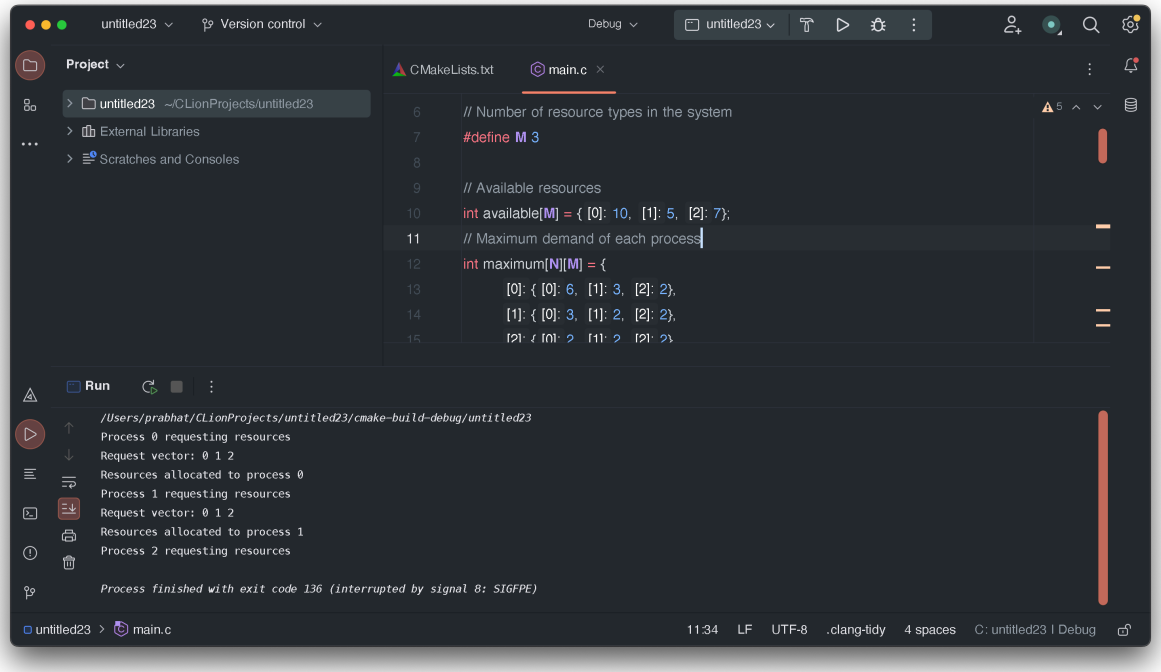
```
#include <stdio.h>
#include <stdlib.h>

// Number of processes in the system
#define N 5
// Number of resource types in the system
#define M 3

// Available resources
int available[M] = {10, 5, 7};
// Maximum demand of each process
int maximum[N][M] = {
        {6, 3, 2},
        {3, 2, 2},
        {2, 2, 2},
        {4, 3, 3},
        {1, 2, 2}
};
// Resources allocated to each process
int allocation[N][M] = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 2},
        {2, 1, 1},
        {0, 0, 2}
};
// Need of each process
int need[N][M];
```

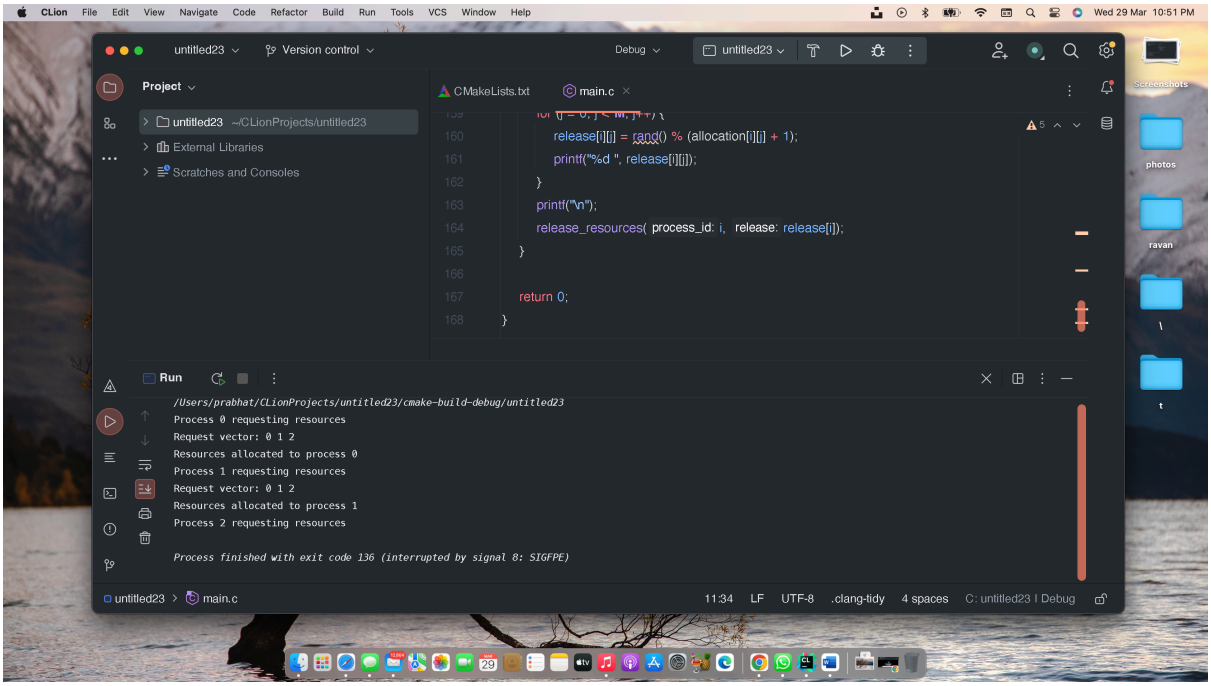
```c
// Function to calculate the need matrix
void calculate_need() {
    int i, j;
    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            need[i][j] = maximum[i][j] - allocation[i][j];
        }
    }
}

// Function to check if the system is in a safe state
int is_safe() {
    int i, j;
    int work[M];
    int finish[N] = {0};

    // Initialize work vector
    for (i = 0; i < M; i++) {
        work[i] = available[i];
    }

    // Find an unfinished process whose needs can be satisfied
    for (i = 0; i < N; i++) {
        if (finish[i] == 0) {
            int can_be_allocated = 1;
            for (j = 0; j < M; j++) {
                if (need[i][j] > work[j]) {
                    can_be_allocated = 0;
                    break;
                }
            }
            if (can_be_allocated) {
                // Allocate resources to this process
                finish[i] = 1;
                for (j = 0; j < M; j++) {
                    work[j] += allocation[i][j];
                }
                i = -1;  // Restart the loop from the beginning
            }
        }
    }

    // If all processes have been finished, the system is in a safe state
    for (i = 0; i < N; i++) {
        if (finish[i] == 0) {
            return 0;
        }
    }
    return 1;
}

// Function to request resources for a process
void request_resources(int process_id, int request[]) {
    int i, j;

    // Check if the request is valid
    for (i = 0; i < M; i++) {
        if (request[i] > need[process_id][i] || request[i] > available[i])
{
            printf("Error: Invalid request\n");
            return;
```

```
        }
    }

    // Try to allocate the resources to the process
    for (i = 0; i < M; i++) {
        available[i] -= request[i];
        allocation[process_id][i] += request[i];
        need[process_id][i] -= request[i];
    }

// Check if the system is still in a safe state
    if (is_safe()) {
        printf("Resources allocated to process %d\n", process_id);
    } else {
        // The system is not in a safe state, so undo the allocation
        for (i = 0; i < M; i++) {
            available[i] += request[i];
            allocation[process_id][i] -= request[i];
            need[process_id][i] += request[i];
        }
        printf("Resources cannot be allocated to process %d\n",
process_id);
    }
}

// Function to release resources held by a process
void release_resources(int process_id, int release[]) {
    int i;
    for (i = 0; i < M; i++) {
        available[i] += release[i];
        allocation[process_id][i] -= release[i];
        need[process_id][i] += release[i];
    }
}

int main() {
    int request[N][M] = {
            {0, 0, 0},
            {0, 0, 0},
            {0, 0, 0},
            {0, 0, 0},
            {0, 0, 0}
    };
    int release[N][M] = {
            {0, 0, 0},
            {0, 0, 0},
            {0, 0, 0},
            {0, 0, 0},
            {0, 0, 0}
    };
    int i, j;
    // Calculate the need matrix
    calculate_need();

// Request resources for each process in turn
    for (i = 0; i < N; i++) {
        printf("Process %d requesting resources\n", i);
        printf("Request vector: ");
        for (j = 0; j < M; j++) {
            request[i][j] = rand() % (need[i][j] + 1);
            printf("%d ", request[i][j]);
```

```
        }
        printf("\n");
        request_resources(i, request[i]);
    }

// Release resources held by each process in turn
    for (i = 0; i < N; i++) {
        printf("Process %d releasing resources\n", i);
        printf("Release vector: ");
        for (j = 0; j < M; j++) {
            release[i][j] = rand() % (allocation[i][j] + 1);
            printf("%d ", release[i][j]);
        }
        printf("\n");
        release_resources(i, release[i]);
    }

    return 0;
}
```

OUTPUT:-




CLion File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
Wed 29 Mar 10:51 PM
untitled23 Version control
Debug
untitled23
Project
untitled23 ~/CLionProjects/untitled23
External Libraries
Scratches and Consoles
CMakeLists.txt
main.c
release[i][j] = rand() % (allocation[i][j] + 1);
printf("%d ", release[i][j]);
printf("\n");
release_resources(process_id: i, release: release[i]);
return 0;
Run
/Users/prabhat/CLionProjects/untitled23/cmake-build-debug/untitled23
Process 0 requesting resources
Request vector: 0 1 2
Resources allocated to process 0
Process 1 requesting resources
Request vector: 0 1 2
Resources allocated to process 1
Process 2 requesting resources
Process finished with exit code 136 (interrupted by signal 8: SIGFPE)
untitled23 main.c
11:34 LF UTF-8 .clang-tidy 4 spaces C: untitled23 | Debug
Screenshots
photos
ravan
\
t

Lab 8

Aim: - Implement FF, BF and WF using different request sizes.

Theory:- FF, BF, and WF are allocation strategies used in memory management to allocate and deallocate memory blocks. These strategies differ in how they choose which memory block to allocate when a request for memory is made. The request size is the amount of memory requested by the process. Implementation of these strategies using different request sizes are implemented in below code.

First-Fit (FF):-

In the first-fit allocation strategy, the memory allocator searches for the first available block of memory that is large enough to fulfill the request. The search starts from the beginning of the memory space and stops at the first available block that is large enough to accommodate the request.

Best-Fit (BF):-

In the best-fit allocation strategy, the memory allocator searches for the smallest available block of memory that is large enough to fulfill the request. The search starts from the beginning of the memory space and stops at the first available block that is large enough to accommodate the request.

Worst-Fit (WF):-

In the worst-fit allocation strategy, the memory allocator searches for the largest available block of memory and allocates the request there. The search starts from the beginning of the memory space and stops at the largest available block of memory that is large enough to accommodate the request.

CODE:-

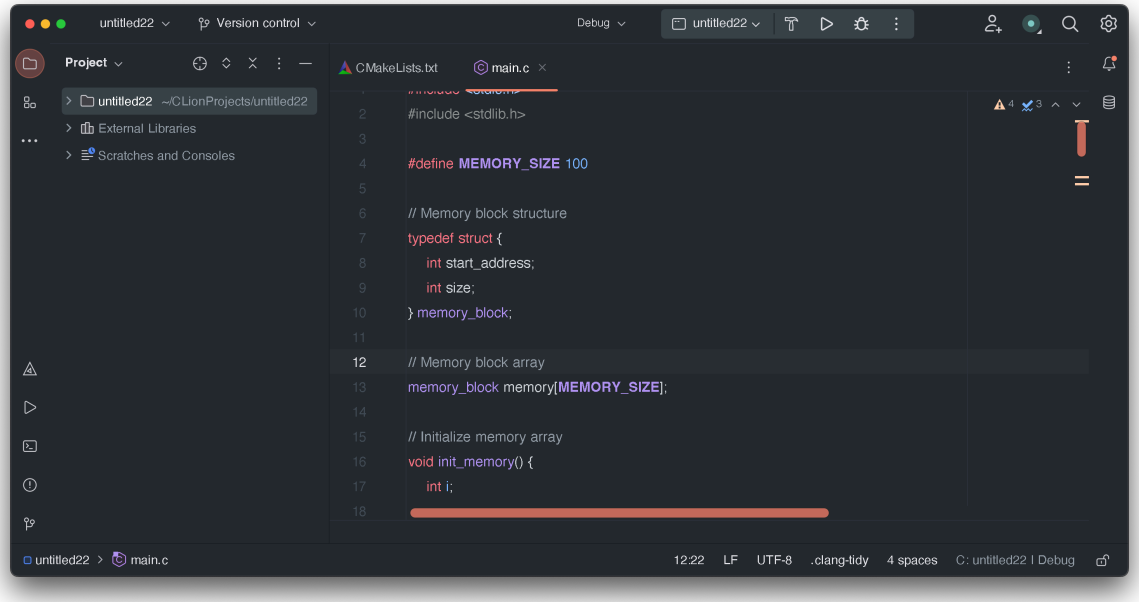
```c
#include <stdio.h>
#include <stdlib.h>

#define MEMORY_SIZE 100

// Memory block structure
typedef struct {
    int start_address;
    int size;
} memory_block;

// Memory block array
memory_block memory[MEMORY_SIZE];

// Initialize memory array
void init_memory() {
    int i;
    for (i = 0; i < MEMORY_SIZE; i++) {
        memory[i].start_address = -1;
        memory[i].size = -1;
    }
}

// Allocate memory using first-fit strategy
int first_fit(int request_size) {
    int i, j, start_address = -1;
    for (i = 0; i < MEMORY_SIZE; i++) {
        if (memory[i].size == -1) { // Check if block is free
            start_address = i;
            j = i + 1;
            while (j < MEMORY_SIZE && memory[j].size == -1) {
                j++;
            }
            if (j - i >= request_size) { // Check if block is large enough
                memory[i].start_address = i;
                memory[i].size = request_size;
                return i;
            }
            i = j - 1;
```

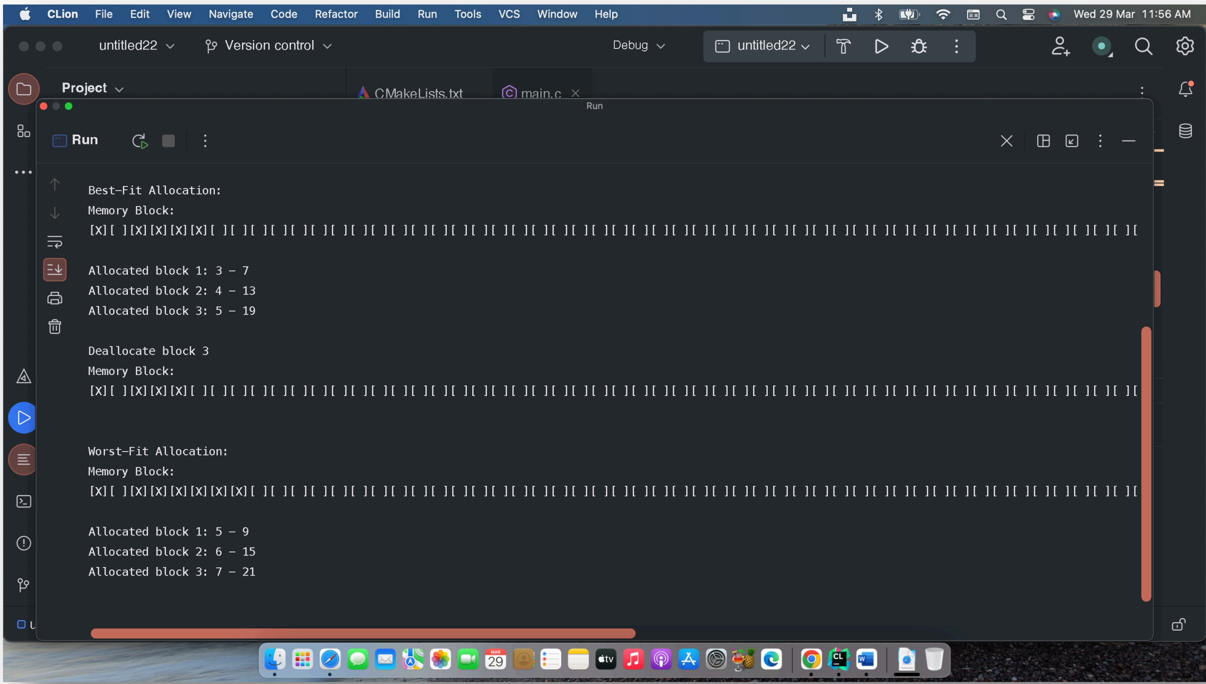
```
        }
    }
    return -1; // Not enough free memory
}

// Allocate memory using best-fit strategy
int best_fit(int request_size) {
    int i, j, start_address = -1, smallest_size = MEMORY_SIZE;
    for (i = 0; i < MEMORY_SIZE; i++) {
        if (memory[i].size == -1) { // Check if block is free
            j = i + 1;
            while (j < MEMORY_SIZE && memory[j].size == -1) {
                j++;
            }
            if (j - i >= request_size && j - i < smallest_size) { // Check
if block is large enough and smallest
                start_address = i;
                smallest_size = j - i;
            }
            i = j - 1;
        }
    }
    if (start_address != -1) {
        memory[start_address].start_address = start_address;
        memory[start_address].size = request_size;
    }
    return start_address;
}

// Allocate memory using worst-fit strategy
int worst_fit(int request_size) {
    int i, j, start_address = -1, largest_size = -1;
    for (i = 0; i < MEMORY_SIZE; i++) {
        if (memory[i].size == -1) { // Check if block is free
            j = i + 1;
            while (j < MEMORY_SIZE && memory[j].size == -1) {
                j++;
            }
            if (j - i >= request_size && j - i > largest_size) { // Check
if block is large enough and largest
                start_address = i;
                largest_size = j - i;
            }
            i = j - 1;
        }
    }
    if (start_address != -1) {
        memory[start_address].start_address = start_address;
        memory[start_address].size = request_size;
    }
    return start_address;
}

// Deallocate memory block
void deallocate(int start_address) {
    int i;
    for (i = start_address; i < start_address + memory[start_address].size;
i++) {
        memory[i].start_address = -1;
        memory[i].size = -1;
    }
```

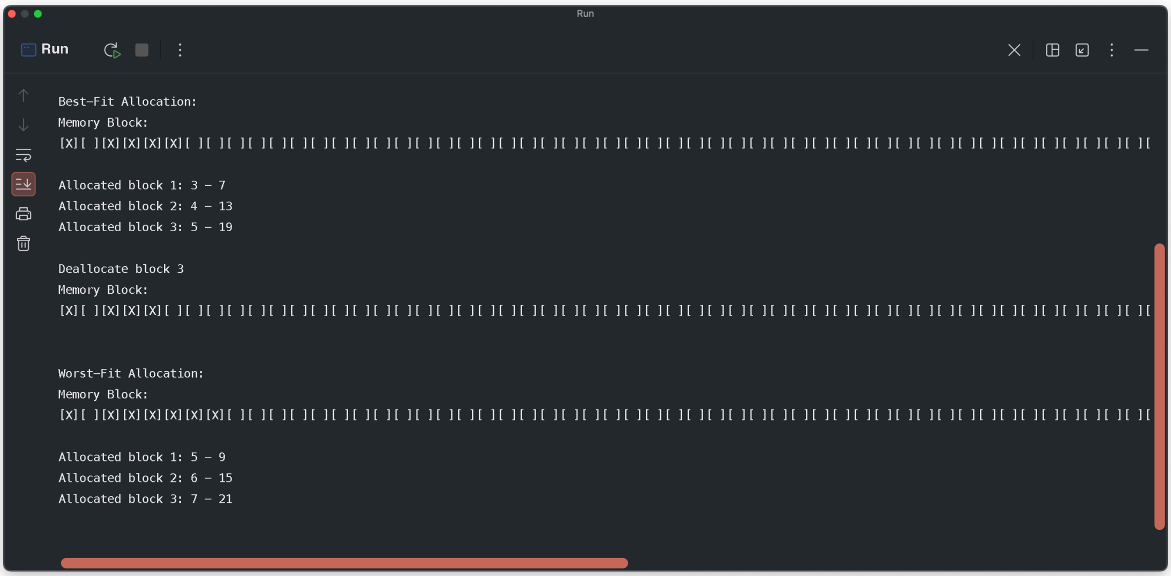
```
}

// Print memory block
void print_memory() {
    int i;
    printf("Memory Block:\n");
    for (i = 0;i < MEMORY_SIZE; i++) {
        if (memory[i].size == -1) {
            printf("[ ]"); // Free memory block
        } else {
            printf("[X]"); // Allocated memory block
        }
    }
    printf("\n\n");
}

int main() {
    init_memory();
    // Allocate memory using first-fit strategy
    printf("First-Fit Allocation:\n");
    int addr1 = first_fit(5);
    int addr2 = first_fit(10);
    int addr3 = first_fit(15);
    print_memory();
    printf("Allocated block 1: %d - %d\n", addr1, addr1 +
memory[addr1].size - 1);
    printf("Allocated block 2: %d - %d\n", addr2, addr2 +
memory[addr2].size - 1);
    printf("Allocated block 3: %d - %d\n", addr3, addr3 +
memory[addr3].size - 1);
    printf("\n");

// Deallocate memory blocks
    printf("Deallocate block 2\n");
    deallocate(addr2);
    print_memory();
    printf("\n");

// Allocate memory using best-fit strategy
    printf("Best-Fit Allocation:\n");
    addr1 = best_fit(5);
    addr2 = best_fit(10);
    addr3 = best_fit(15);
    print_memory();
    printf("Allocated block 1: %d - %d\n", addr1, addr1 +
memory[addr1].size - 1);
    printf("Allocated block 2: %d - %d\n", addr2, addr2 +
memory[addr2].size - 1);
    printf("Allocated block 3: %d - %d\n", addr3, addr3 +
memory[addr3].size - 1);
    printf("\n");

// Deallocate memory blocks
    printf("Deallocate block 3\n");
    deallocate(addr3);
    print_memory();
    printf("\n");

// Allocate memory using worst-fit strategy
    printf("Worst-Fit Allocation:\n");
    addr1 = worst_fit(5);
```

```
    addr2 = worst_fit(10);
    addr3 = worst_fit(15);
    print_memory();
    printf("Allocated block 1: %d - %d\n", addr1, addr1 +
memory[addr1].size - 1);
    printf("Allocated block 2: %d - %d\n", addr2, addr2 +
memory[addr2].size - 1);
    printf("Allocated block 3: %d - %d\n", addr3, addr3 +
memory[addr3].size - 1);
    printf("\n");

    return 0;
}
```

OUTPUT:-




CLion File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
Wed 29 Mar 11:56 AM
untitled22
Version control
Debug
Project
CMakeLists.txt
main.c
Run
Best-Fit Allocation:
Memory Block:
Allocated block 1: 3 – 7
Allocated block 2: 4 – 13
Allocated block 3: 5 – 19
Deallocate block 3
Memory Block:
Worst-Fit Allocation:
Memory Block:
Allocated block 1: 5 – 9
Allocated block 2: 6 – 15
Allocated block 3: 7 – 21


Run
Best-Fit Allocation:
Memory Block:
Allocated block 1: 3 – 7
Allocated block 2: 4 – 13
Allocated block 3: 5 – 19
Deallocate block 3
Memory Block:
Worst-Fit Allocation:
Memory Block:
Allocated block 1: 5 – 9
Allocated block 2: 6 – 15
Allocated block 3: 7 – 21

Lab 9

Aim: - Implement each of the above algorithms(separately) using a reference string : - 4,7,6,1,7,6,1,2,7,2 and number of frames for allocation: 3

Theory:- Page replacement algorithm is a crucial component of memory management in modern computer systems. The main function of a page replacement algorithm is to choose which pages in the memory should be removed to make space for incoming pages. When the memory becomes full, and a new page needs to be brought in, the operating system needs to remove an existing page from the memory to make space for the new page.

The choice of the page to remove is based on the specific page replacement algorithm being used. The algorithm's primary objective is to minimize the number of page faults, which occur when a program tries to access a page that is not in the memory. Page faults result in a significant overhead in terms of disk I/O and CPU usage as the operating system needs to bring the page back into memory from disk, which is a time-consuming process.

In general, the goal of all page replacement algorithms is to minimize the number of page faults that occur, as page faults result in a significant overhead in terms of disk I/O and CPU usage. The choice of the algorithm depends on various factors such as the size of the memory, the size of the program, and the access patterns of the program.

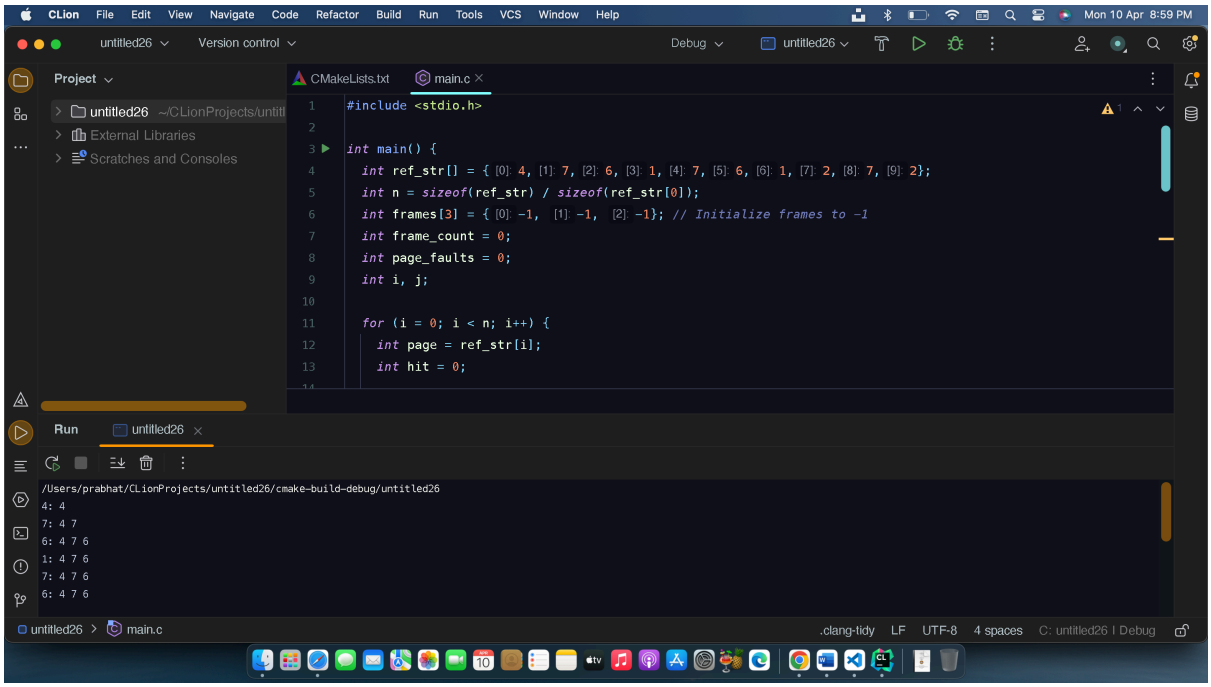
FIFO Algorithm

The First-In-First-Out (FIFO) algorithm is a page replacement algorithm that works on the principle of the first page that comes in is the first page that goes out. It uses a queue to keep track of the order in which pages are brought into the memory. When a page fault occurs, the oldest page in the queue (i.e., the page that was brought in first) is removed from the memory and the new page is brought in its place.

CODE:-

```c
#include <stdio.h>

int main() {
  int ref_str[] = {4,7,6,1,7,6,1,2,7,2};
  int n = sizeof(ref_str) / sizeof(ref_str[0]);
  int frames[3] = {-1, -1, -1}; // Initialize frames to -1
  int frame_count = 0;
  int page_faults = 0;
  int i, j;

  for (i = 0; i < n; i++) {
    int page = ref_str[i];
    int hit = 0;

    for (j = 0; j < frame_count; j++) {
      if (frames[j] == page) {
        hit = 1;
        break;
      }
    }

    if (!hit) {
      frames[frame_count] = page;
      frame_count++;

      if (frame_count > 3) {
        frame_count = 3;
      }

      page_faults++;
    }

    printf("%d: ", page);

    for (j = 0; j < frame_count; j++) {
      printf("%d ", frames[j]);
    }

    printf("\n");
  }

  printf("Page faults: %d\n", page_faults);

  return 0;
}
```

OUTPUT:-



LRU Algorithm:

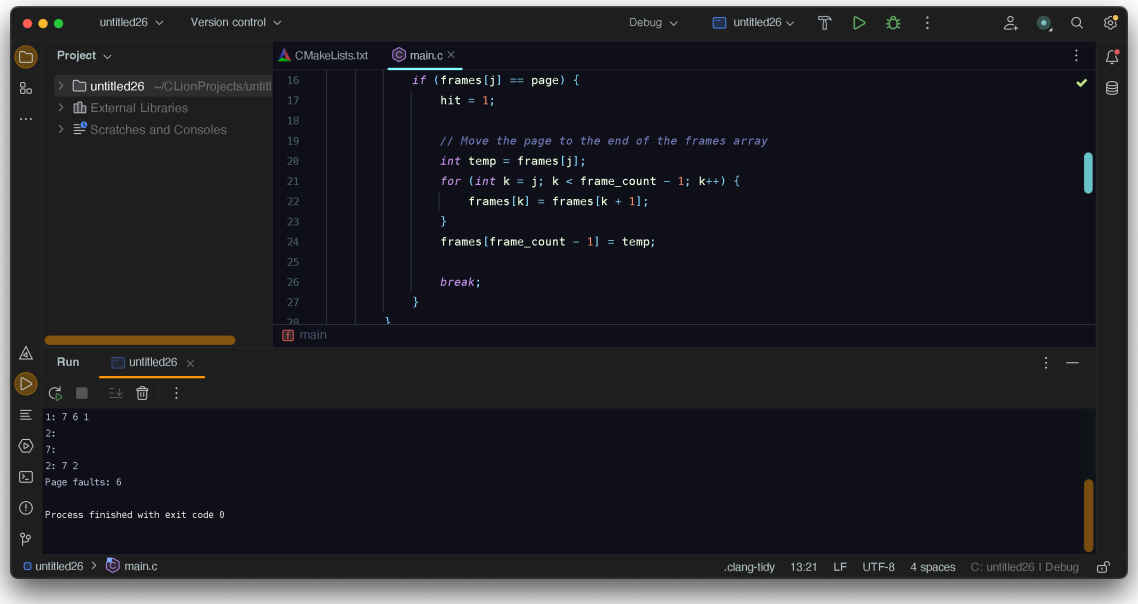
The Least Recently Used (LRU) algorithm is a page replacement algorithm that works on the principle of the page that has not been used for the longest time is the one that should be replaced. It maintains a record of the time when each page is last accessed, and when a page fault occurs, it selects the page that has the oldest access time for replacement.

CODE:-

```c
#include <stdio.h>

int main() {
    int ref_str[] = {4,7,6,1,7,6,1,2,7,2};
```

```c
    int n = sizeof(ref_str) / sizeof(ref_str[0]);
    int frames[3] = {-1, -1, -1}; // Initialize frames to -1
    int frame_count = 0;
    int page_faults = 0;
    int i, j;

    for (i = 0; i < n; i++) {
        int page = ref_str[i];
        int hit = 0;

        for (j = 0; j < frame_count; j++) {
            if (frames[j] == page) {
                hit = 1;

                // Move the page to the end of the frames array
                int temp = frames[j];
                for (int k = j; k < frame_count - 1; k++) {
                    frames[k] = frames[k + 1];
                }
                frames[frame_count - 1] = temp;

                break;
            }
        }

        if (!hit) {
            if (frame_count < 3) {
                frames[frame_count] = page;
                frame_count++;
            } else {
                // Replace the least recently used page
                frames[0] = frames[1];
                frames[1] = frames[2];
                frames[2] = page;
            }

            page_faults++;
        }

        printf("%d: ", page);


                while (j < frame_count) {
                    printf("%d ", frames[j]);
                    j++;
                }

        printf("\n");
    }

    printf("Page faults: %d\n", page_faults);

    return 0;
}
```

OUTPUT:-



Optimal Algorithm:

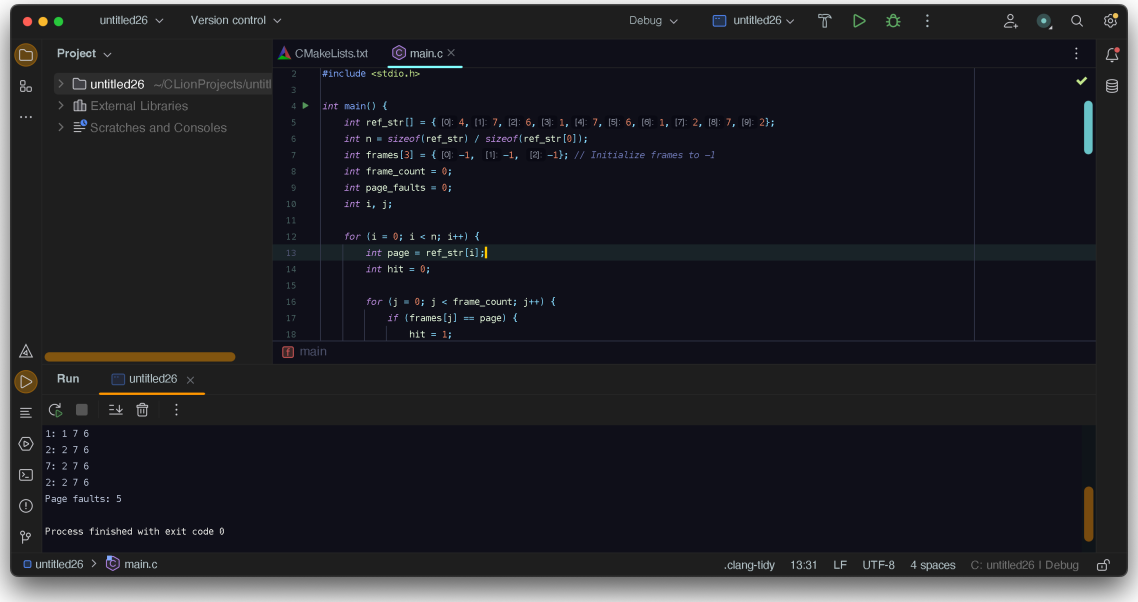
The Optimal algorithm is a page replacement algorithm that works on the principle of selecting the page that will not be used for the longest period of time in the future. This algorithm requires knowledge of the future reference string, which is not always possible to obtain. Therefore, this algorithm is usually used for benchmarking other algorithms rather than in practical implementations.

CODE:-

```c
#include <stdio.h>

int main() {
    int ref_str[] = {4,7,6,1,7,6,1,2,7,2};
    int n = sizeof(ref_str) / sizeof(ref_str[0]);
    int frames[3] = {-1, -1, -1}; // Initialize frames to -1
    int frame_count = 0;
    int page_faults = 0;
    int i, j;

    for (i = 0; i < n; i++) {
        int page = ref_str[i];
        int hit = 0;

        for (j = 0; j < frame_count; j++) {
            if (frames[j] == page) {
                hit = 1;
                break;
            }
        }

        if (!hit) {
            if (frame_count < 3) {
                frames[frame_count] = page;
                frame_count++;
            } else {
                int max_distance = -1;
                int replace_index = -1;

                for (j = 0; j < frame_count; j++) {
                    int distance = 0;
                    int k;

                    for (k = i + 1; k < n; k++) {
                        if (frames[j] == ref_str[k]) {
                            break;
                        }
                        distance++;
                    }

                    if (distance > max_distance) {
                        max_distance = distance;
                        replace_index = j;
                    }
                }

                frames[replace_index] = page;
            }
```

```
            page_faults++;
        }

        printf("%d: ", page);

        for (j = 0; j < frame_count; j++) {
            printf("%d ", frames[j]);
        }

        printf("\n");
    }

    printf("Page faults: %d\n", page_faults);

    return 0;
}
```

OUTPUT:-



Lab 10

Aim: - Build and implement single level and 2-level directories.

Theory:- **Single Level Directory**

In a single level directory structure, all files are stored in a single directory. This directory acts as the root directory for the file system, and all files are located directly within it. Each file is given a unique name within the directory, and this name is used to identify the file. This type of directory structure is simple to implement, but it can become unwieldy if there are a large number of files.

Two-level Directory

A two-level directory structure is an improvement over the single level structure, as it allows for a hierarchical organization of files. In this structure, there is a root directory that contains a number of subdirectories. Each subdirectory can contain files and additional subdirectories, forming a tree-like structure. Each file or subdirectory is given a unique name within its parent directory. This name is used to identify the file or subdirectory within the directory tree. This structure allows for better organization and management of files, as related files can be grouped together in the same subdirectory.

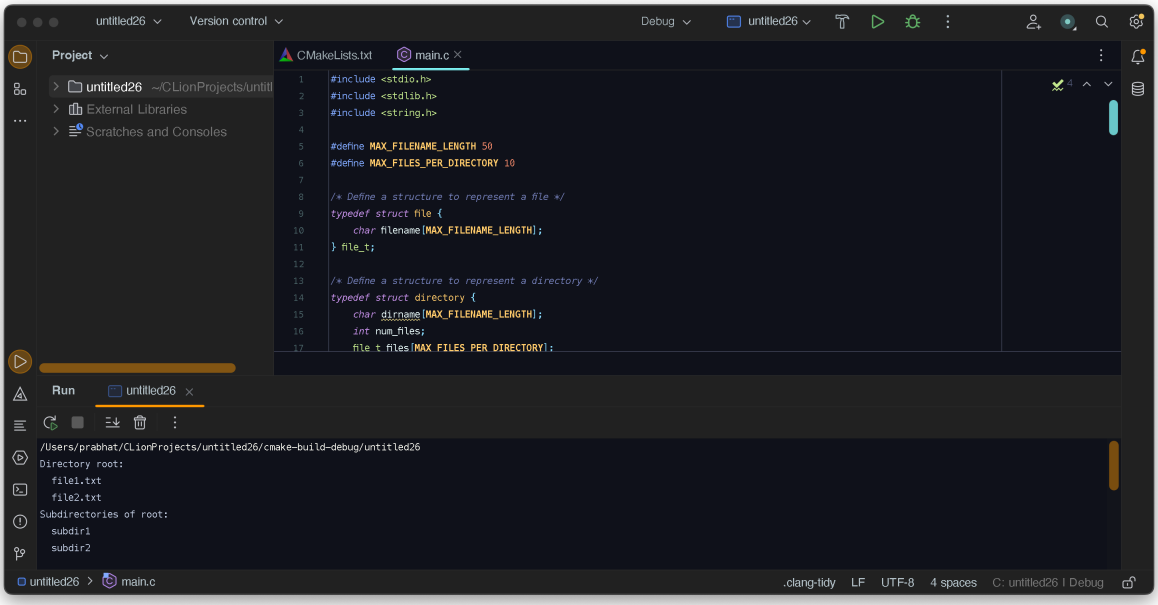
CODE:-

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

#define MAX_FILENAME_LENGTH 50
#define MAX_FILES_PER_DIRECTORY 10
```

```c
/* Define a structure to represent a file */
typedef struct file {
    char filename[MAX_FILENAME_LENGTH];
} file_t;

/* Define a structure to represent a directory */
typedef struct directory {
    char dirname[MAX_FILENAME_LENGTH];
    int num_files;
    file_t files[MAX_FILES_PER_DIRECTORY];
    struct directory* subdirectories;
} directory_t;

/* Create a new file with the given filename */
file_t* create_file(char* filename) {
    file_t* new_file = malloc(sizeof(file_t));
    if (new_file == NULL) {
        printf("Failed to allocate memory for file.\n");
        exit(1);
    }
    strcpy(new_file->filename, filename);
    return new_file;
}

/* Add a new file to the given directory */
void add_file_to_directory(directory_t* directory, char* filename) {
    if (directory->num_files >= MAX_FILES_PER_DIRECTORY) {
        printf("Directory %s is full.\n", directory->dirname);
        return;
    }
    file_t* new_file = create_file(filename);
    directory->files[directory->num_files] = *new_file;
    directory->num_files++;
}

/* Create a new directory with the given dirname */
directory_t* create_directory(char* dirname) {
    directory_t* new_directory = malloc(sizeof(directory_t));
    if (new_directory == NULL) {
        printf("Failed to allocate memory for directory.\n");
        exit(1);
    }
    strcpy(new_directory->dirname, dirname);
    new_directory->num_files = 0;
    new_directory->subdirectories = NULL;
    return new_directory;
}

/* Add a new subdirectory to the given directory */
void add_subdirectory_to_directory(directory_t* directory, char* dirname) {
    directory_t* new_directory = create_directory(dirname);
    directory_t* current_directory = directory->subdirectories;
    if (current_directory == NULL) {
        directory->subdirectories = new_directory;
    } else {
        while (current_directory->subdirectories != NULL) {
            current_directory = current_directory->subdirectories;
        }
        current_directory->subdirectories = new_directory;
    }
}

/* Print the files in the given directory */
void print_files_in_directory(directory_t* directory) {
    printf("Directory %s:\n", directory->dirname);
    if (directory->num_files == 0) {
```

```
        printf("  No files.\n");
    } else {
        for (int i = 0; i < directory->num_files; i++) {
            printf("  %s\n", directory->files[i].filename);
        }
    }
}

/* Print the subdirectories of the given directory */
void print_subdirectories(directory_t* directory) {
    if (directory->subdirectories == NULL) {
        return;
    }
    directory_t* current_directory = directory->subdirectories;
    printf("Subdirectories of %s:\n", directory->dirname);
    while (current_directory != NULL) {
        printf("  %s\n", current_directory->dirname);
        current_directory = current_directory->subdirectories;
    }
}

int main() {
    /* Create a root directory */
    directory_t* root_directory = create_directory("root");

    /* Add some files to the root directory */
    add_file_to_directory(root_directory, "file1.txt");
    add_file_to_directory(root_directory, "file2.txt");

    /* Add a subdirectory to the root directory */
    add_subdirectory_to_directory(root_directory, "subdir1");

    /* Add some files to the subdirectory */
    directory_t* subdir1 = root_directory->subdirectories;
    add_file_to_directory(subdir1, "subdir1file1.txt");
    add_file_to_directory(subdir1, "subdir1file2.txt");

    /* Add a subdirectory to the subdirectory */
    add_subdirectory_to_directory(subdir1, "subdir2");

    /* Add some files to the subdirectory of the subdirectory */
    directory_t* subdir2 = subdir1->subdirectories;
    add_file_to_directory(subdir2, "subdir2file1.txt");

    /* Print the files in the root directory */
    print_files_in_directory(root_directory);

    /* Print the subdirectories of the root directory */
    print_subdirectories(root_directory);

    /* Print the files in the subdirectory */
    print_files_in_directory(subdir1);

    /* Print the subdirectories of the subdirectory */
    print_subdirectories(subdir1);

    /* Print the files in the subdirectory of the subdirectory */
    print_files_in_directory(subdir2);

    return 0;
}
```

OUTPUT:-



Q2:- Build and implement tree-structured directories.

Tree-Structured Directory

A tree-structured directory is a type of hierarchical file system organization that allows for multiple levels of directories. In this structure, there is a root directory that contains a number of subdirectories. Each subdirectory can contain files and additional subdirectories, forming a tree-like structure.

In a tree-structured directory, each node in the tree represents a directory, and the links between the nodes represent the parent-child relationships between directories. The root node represents the root directory, and each child node represents a subdirectory of the parent node.

Files are typically stored in the leaf nodes of the tree, and each file is uniquely identified by its path, which consists of the names of all the directories that must be traversed to reach the file from the root directory.Tree-structured directories are highly flexible and can be used to organize large numbers of files in a logical and intuitive way. However, they can be more complex to implement and maintain than simpler directory structures like single-level or two-level directories.

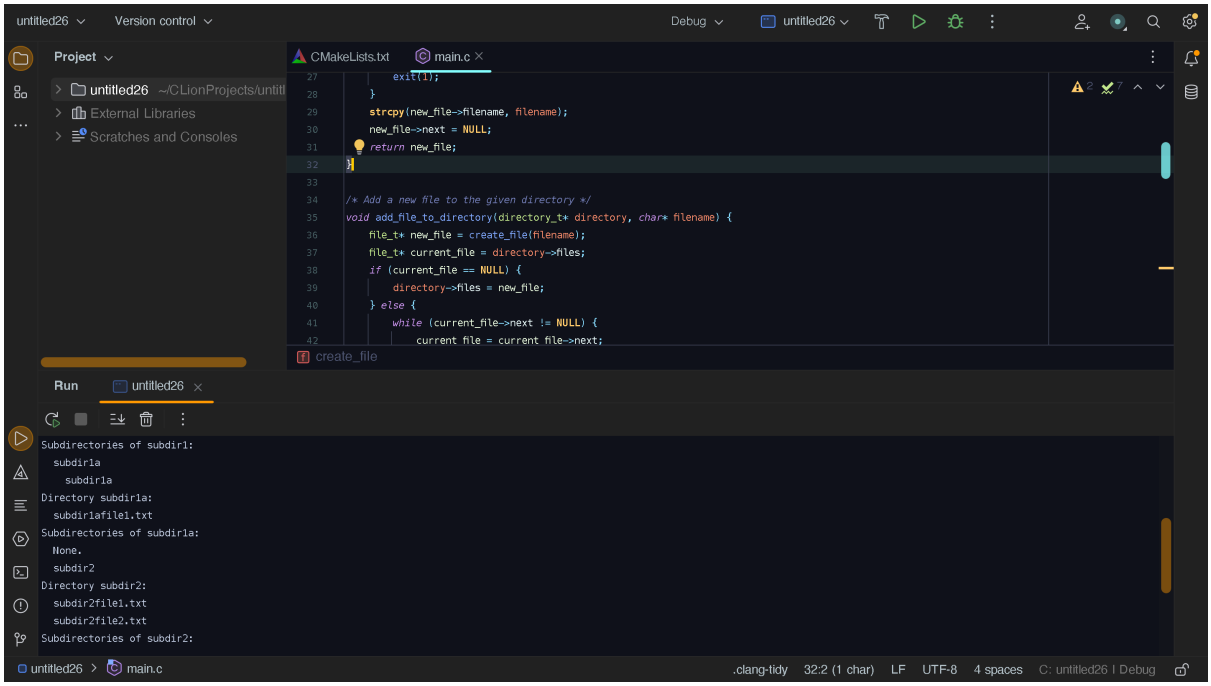
CODE:-

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

#define MAX_FILENAME_LENGTH 50

/* Define a structure to represent a file */
typedef struct file {
    char filename[MAX_FILENAME_LENGTH];
    struct file* next;
} file_t;

/* Define a structure to represent a directory */
typedef struct directory {
    char dirname[MAX_FILENAME_LENGTH];
    struct directory* parent;
    struct directory* child;
    struct directory* next;
    file_t* files;
} directory_t;

/* Create a new file with the given filename */
file_t* create_file(char* filename) {
    file_t* new_file = malloc(sizeof(file_t));
    if (new_file == NULL) {
        printf("Failed to allocate memory for file.\n");
        exit(1);
    }
    strcpy(new_file->filename, filename);
    new_file->next = NULL;
    return new_file;
}

/* Add a new file to the given directory */
void add_file_to_directory(directory_t* directory, char* filename) {
    file_t* new_file = create_file(filename);
    file_t* current_file = directory->files;
    if (current_file == NULL) {
        directory->files = new_file;
    } else {
        while (current_file->next != NULL) {
            current_file = current_file->next;
        }
        current_file->next = new_file;
    }
}
```

```c
/* Create a new directory with the given dirname */
directory_t* create_directory(char* dirname, directory_t* parent) {
    directory_t* new_directory = malloc(sizeof(directory_t));
    if (new_directory == NULL) {
        printf("Failed to allocate memory for directory.\n");
        exit(1);
    }
    strcpy(new_directory->dirname, dirname);
    new_directory->parent = parent;
    new_directory->child = NULL;
    new_directory->next = NULL;
    new_directory->files = NULL;
    return new_directory;
}

/* Add a new child directory to the given directory */
void add_child_directory(directory_t* directory, char* dirname) {
    directory_t* new_directory = create_directory(dirname, directory);
    directory_t* current_directory = directory->child;
    if (current_directory == NULL) {
        directory->child = new_directory;
    } else {
        while (current_directory->next != NULL) {
            current_directory = current_directory->next;
        }
        current_directory->next = new_directory;
    }
}

/* Print the files in the given directory */
void print_files_in_directory(directory_t* directory) {
    printf("Directory %s:\n", directory->dirname);
    file_t* current_file = directory->files;
    if (current_file == NULL) {
        printf("  No files.\n");
    } else {
        while (current_file != NULL) {
            printf("  %s\n", current_file->filename);
            current_file = current_file->next;
        }
    }
}

/* Print the subdirectories of the given directory */
void print_subdirectories(directory_t* directory) {
    printf("Subdirectories of %s:\n", directory->dirname);
    directory_t* current_directory = directory->child;
    if (current_directory == NULL) {
        printf("  None.\n");
    } else {
        while (current_directory != NULL) {
            printf("  %s\n", current_directory->dirname);
            current_directory = current_directory->next;
        }
    }
}

/* Print the directory tree rooted at the given directory */
void print_directory_tree(directory_t* directory, int depth) {
    printf("%*s%s\n", depth * 2, "", directory->dirname);
    print_files_in_directory(directory);
    print_subdirectories(directory);
    directory_t* current_directory = directory->child;
    while (current_directory != NULL) {
        print_directory_tree(current_directory, depth + 1);
```

```
        current_directory = current_directory->next;
    }
}

int main() {
    /* Create the root directory */
    directory_t* root_directory = create_directory("root", NULL);

    /* Add some files to the root directory */
    add_file_to_directory(root_directory, "rootfile1.txt");
    add_file_to_directory(root_directory, "rootfile2.txt");

    /* Add some subdirectories to the root directory */
    add_child_directory(root_directory, "subdir1");
    add_child_directory(root_directory, "subdir2");

    /* Add some files and subdirectories to the subdirectories */
    directory_t* subdir1 = root_directory->child;
    add_file_to_directory(subdir1, "subdir1file1.txt");
    add_file_to_directory(subdir1, "subdir1file2.txt");
    add_child_directory(subdir1, "subdir1a");

    directory_t* subdir1a = subdir1->child;
    add_file_to_directory(subdir1a, "subdir1afile1.txt");

    directory_t* subdir2 = subdir1->next;
    add_file_to_directory(subdir2, "subdir2file1.txt");
    add_file_to_directory(subdir2, "subdir2file2.txt");
    add_child_directory(subdir2, "subdir2a");

    /* Print the directory tree */
    print_directory_tree(root_directory, 0);

    return 0;
}
```

OUTPUT:-



Q3:- Build and implement acyclic-structured directories.

Acyclic-structured directories are similar to tree-structured directories in that they allow for hierarchical organization of files and directories. However, unlike tree-structured directories, acyclic-structured directories do not allow for cycles in the directory structure, which means that there can be no loops or circular references in the directory hierarchy. Acyclic-structured directories provide a flexible and scalable way to organize large numbers of files and directories, while also enforcing a strict hierarchy that prevents cycles and circular references.

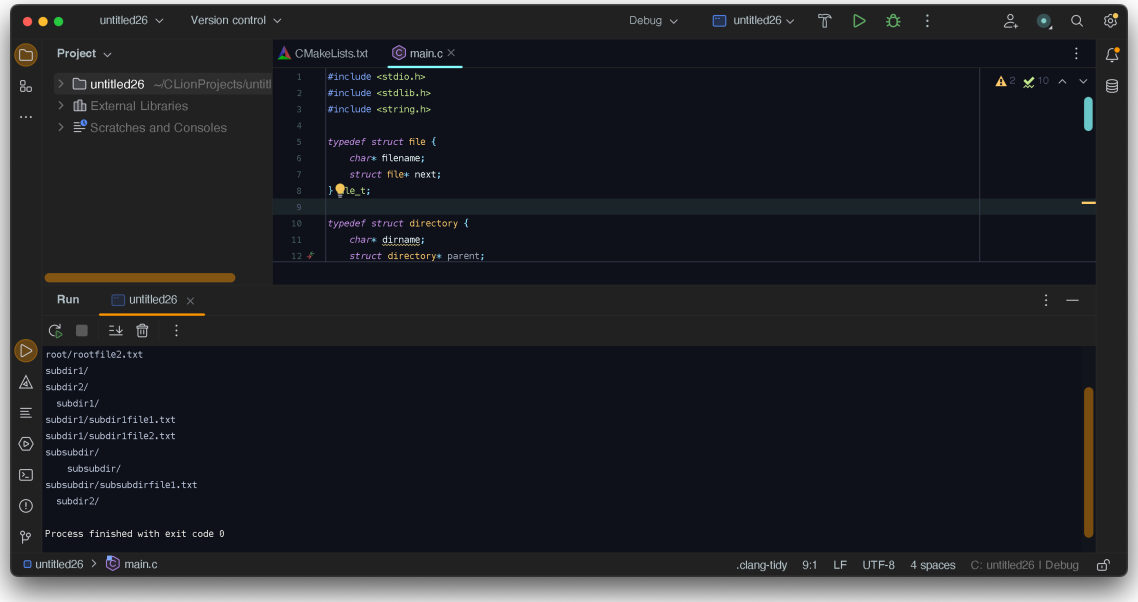
CODE:-

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

typedef struct file {
    char* filename;
    struct file* next;
} file_t;

typedef struct directory {
```

```
    char* dirname;
    struct directory* parent;
    struct directory* child;
    struct directory* next_sibling;
    file_t* files;
} directory_t;

/* Create a new file with the given filename */
file_t* create_file(char* filename) {
    file_t* new_file = (file_t*)malloc(sizeof(file_t));
    new_file->filename = strdup(filename);
    new_file->next = NULL;
    return new_file;
}

/* Add a file to the given directory */
void add_file_to_directory(directory_t* directory, char* filename) {
    file_t* new_file = create_file(filename);
    if (directory->files == NULL) {
        directory->files = new_file;
    } else {
        file_t* current_file = directory->files;
        while (current_file->next != NULL) {
            current_file = current_file->next;
        }
        current_file->next = new_file;
    }
}

/* Create a new directory with the given dirname and parent directory */
directory_t* create_directory(char* dirname, directory_t* parent) {
    directory_t* new_directory = (directory_t*)malloc(sizeof(directory_t));
    new_directory->dirname = strdup(dirname);
    new_directory->parent = parent;
    new_directory->child = NULL;
    new_directory->next_sibling = NULL;
    new_directory->files = NULL;
    return new_directory;
}

/* Add a child directory to the given directory */
void add_child_directory(directory_t* directory, char* dirname) {
    directory_t* new_directory = create_directory(dirname, directory);
    if (directory->child == NULL) {
        directory->child = new_directory;
    } else {
        directory_t* current_directory = directory->child;
        while (current_directory->next_sibling != NULL) {
            current_directory = current_directory->next_sibling;
        }
        current_directory->next_sibling = new_directory;
    }
}

/* Print the files in the given directory */
void print_files_in_directory(directory_t* directory) {
    file_t* current_file = directory->files;
    while (current_file != NULL) {
        printf("%s/%s\n", directory->dirname, current_file->filename);
        current_file = current_file->next;
    }
}

/* Print the subdirectories of the given directory */
void print_subdirectories(directory_t* directory) {
    directory_t* current_directory = directory->child;
```

```
    while (current_directory != NULL) {
        printf("%s/\n", current_directory->dirname);
        current_directory = current_directory->next_sibling;
    }
}

/* Print the directory tree rooted at the given directory */
void print_directory_tree(directory_t* directory, int depth) {
    printf("%*s%s/\n", depth * 2, "", directory->dirname);
    print_files_in_directory(directory);
    print_subdirectories(directory);
    directory_t* current_directory = directory->child;
    while (current_directory != NULL) {
        print_directory_tree(current_directory, depth + 1);
        current_directory = current_directory->next_sibling;
    }
}

int main() {
    /* Create the root directory */
    directory_t *root_directory = create_directory("root", NULL);

    /* Add some files to the root directory */
    add_file_to_directory(root_directory, "rootfile1.txt");
    add_file_to_directory(root_directory, "rootfile2.txt");

    /* Add some subdirectories to the root directory */
    add_child_directory(root_directory, "subdir1");
    add_child_directory(root_directory, "subdir2");

/* Add some files to subdir1 */
    directory_t *subdir1 = root_directory->child;
    add_file_to_directory(subdir1, "subdir1file1.txt");
    add_file_to_directory(subdir1, "subdir1file2.txt");

/* Add a subdirectory to subdir1 */
    add_child_directory(subdir1, "subsubdir");

/* Add some files to subsubdir */
    directory_t *subsubdir = subdir1->child;
    add_file_to_directory(subsubdir, "subsubdirfile1.txt");

/* Print the directory tree */
    print_directory_tree(root_directory, 0);

    return 0;
}
```

OUTPUT:-


untitled26
Version control
Debug
untitled26
Project
untitled26 ~/CLionProjects/untitl
External Libraries
Scratches and Consoles
CMakeLists.txt
main.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct file {
char* filename;
struct file* next;
} file_t;
typedef struct directory {
char* dirname;
struct directory* parent;
Run
untitled26
root/rootfile2.txt
subdir1/
subdir2/
subdir1/
subdir1/subdir1file1.txt
subdir1/subdir1file2.txt
subsubdir/
subsubdir/
subsubdir/subsubdirfile1.txt
subdir2/
Process finished with exit code 0
untitled26
main.c
.clang-tidy
9:1
LF
UTF-8
4 spaces
C: untitled26 | Debug

Lab 11

Aim: - Implement the concepts of Sequential list, Linked List and Indexed File allocations.

Theory:- **Sequential file allocation**

In sequential file allocation, data is stored in contiguous blocks of storage space. This means that all the data in a file is stored in a single contiguous block on the storage medium. This method is simple to implement and allows for fast access to data, but it has a major drawback. If data is deleted from the file or if new data is added to the file, the entire file may need to be reorganized to maintain the contiguous block of storage space. This can be a time-consuming process and can result in wasted storage space.

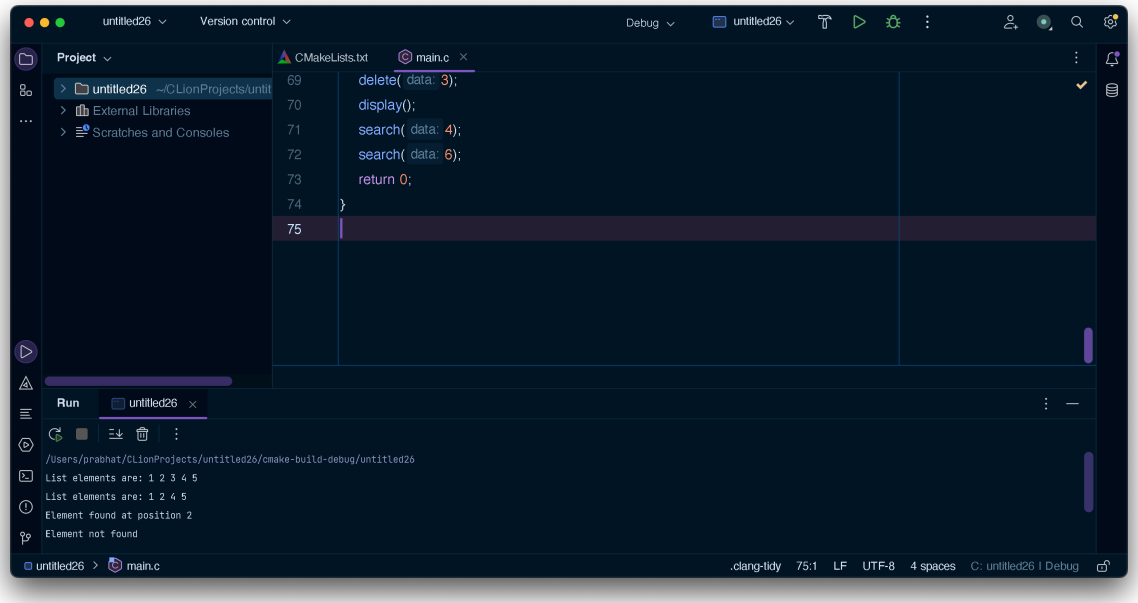
CODE:-

```
#include <stdio.h>

#define MAX_SIZE 100

int arr[MAX_SIZE];
int last = -1;

void insert(int data) {
    if (last >= MAX_SIZE - 1) {
        printf("Error: Overflow\n");
        return;
    }
    arr[++last] = data;
}
```

```
void delete(int data) {
    int i, pos = -1;
    for (i = 0; i <= last; i++) {
        if (arr[i] == data) {
            pos = i;
            break;
        }
    }
    if (pos == -1) {
        printf("Error: Element not found\n");
        return;
    }
    for (i = pos; i < last; i++) {
        arr[i] = arr[i + 1];
    }
    last--;
}

void search(int data) {
    int i, pos = -1;
    for (i = 0; i <= last; i++) {
        if (arr[i] == data) {
            pos = i;
            break;
        }
    }
    if (pos == -1) {
        printf("Element not found\n");
    } else {
        printf("Element found at position %d\n", pos);
    }
}

void display() {
    int i;
    if (last == -1) {
        printf("List is empty\n");
        return;
    }
    printf("List elements are: ");
    for (i = 0; i <= last; i++) {
        printf("%d ", arr[i]);
    }
```

```
    printf("\n");
}

int main() {
    insert(1);
    insert(2);
    insert(3);
    insert(4);
    insert(5);
    display();
    delete(3);
    display();
    search(4);
    search(6);
    return 0;
}
```

OUTPUT:-



Q2:-

Linked file allocation:

In linked file allocation, data is stored in non-contiguous blocks of storage space. Each block of data contains a pointer to the next block of data in the file. This means that data can be stored anywhere on the storage medium, and new data can be added or old data can be deleted without affecting the rest of the file. However, accessing data in a linked file can be slower than accessing data in a sequential file, because the system needs to follow the pointers to find the data.

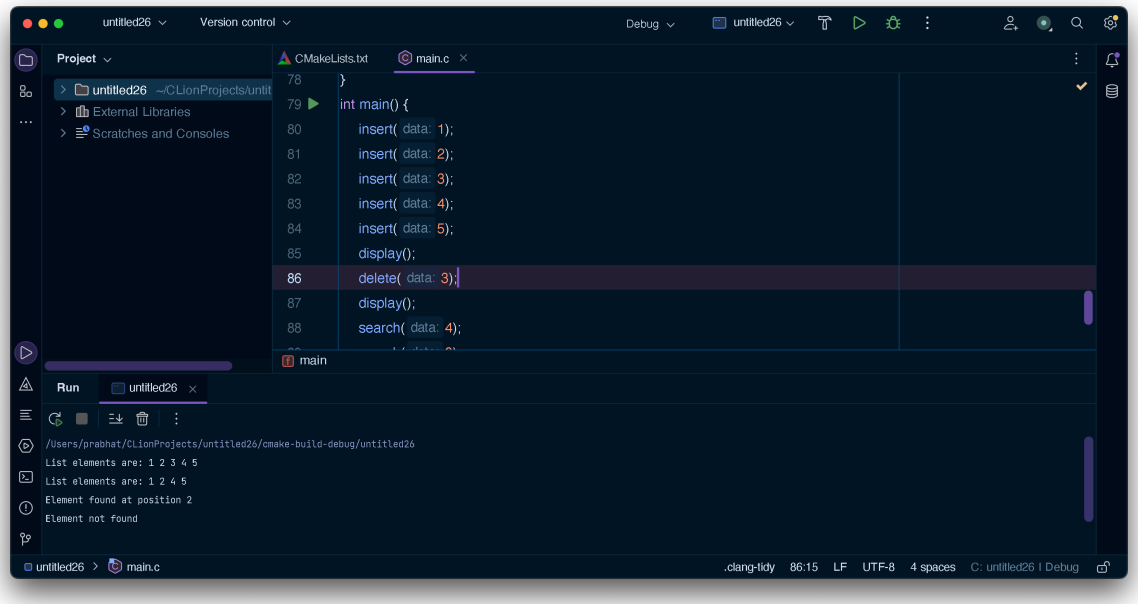
CODE:-

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

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

struct node* head = NULL;

void insert(int data) {
    struct node* new_node = (struct node*)malloc(sizeof(struct node));
    new_node->data = data;
    new_node->next = NULL;
    if (head == NULL) {
        head = new_node;
    } else {
        struct node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = new_node;
    }
```

```
}

void delete(int data) {
    if (head == NULL) {
        printf("Error: List is empty\n");
        return;
    }
    struct node* temp = head;
    struct node* prev = NULL;
    while (temp != NULL && temp->data != data) {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Error: Element not found\n");
        return;
    }
    if (prev == NULL) {
        head = temp->next;
    } else {
        prev->next = temp->next;
    }
    free(temp);
}

void search(int data) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    struct node* temp = head;
    int pos = 0;
    while (temp != NULL && temp->data != data) {
        temp = temp->next;
        pos++;
    }
    if (temp == NULL) {
        printf("Element not found\n");
    } else {printf("Element found at position %d\n", pos);
    }
}

void display() {
```

```
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    printf("List elements are: ");
    struct node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
int main() {
    insert(1);
    insert(2);
    insert(3);
    insert(4);
    insert(5);
    display();
    delete(3);
    display();
    search(4);
    search(6);
    return 0;
}
```

OUTPUT:-



Indexed file allocation:

In indexed file allocation, a file is divided into fixed-size blocks, and each block is given a unique identifier or index. The index is used to look up the block in a table, which contains information about the location of the block on the storage medium. This method allows for fast access to data, because the system can quickly look up the location of a block using the index. It also allows for efficient use of storage space, because blocks can be added or deleted without affecting the rest of the file. However, maintaining the index can be a time-consuming process, and the system needs to keep track of the location of the index table.

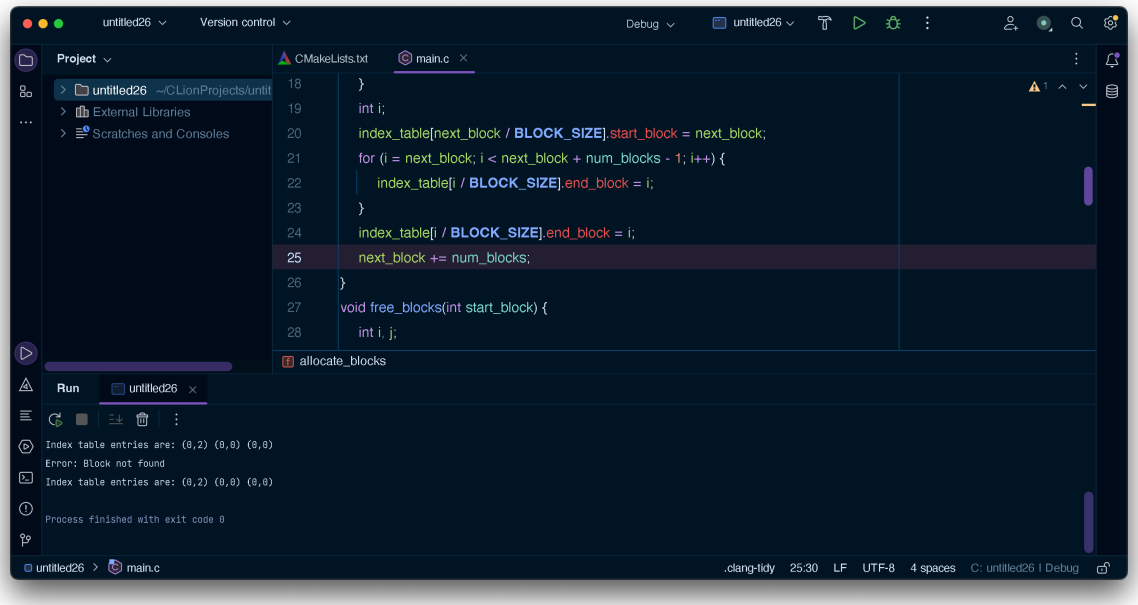
CODE:-

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

#define BLOCK_SIZE 4
#define FILE_SIZE 12

struct index_entry {
```

```
    int start_block;
    int end_block;
};

struct index_entry index_table[FILE_SIZE / BLOCK_SIZE];
int next_block = 0;

void allocate_blocks(int num_blocks) {
    if (next_block + num_blocks > FILE_SIZE / BLOCK_SIZE) {
        printf("Error: Not enough free blocks\n");return;
    }
    int i;
    index_table[next_block / BLOCK_SIZE].start_block = next_block;
    for (i = next_block; i < next_block + num_blocks - 1; i++) {
        index_table[i / BLOCK_SIZE].end_block = i;
    }
    index_table[i / BLOCK_SIZE].end_block = i;
    next_block += num_blocks;
}
void free_blocks(int start_block) {
    int i, j;
    for (i = 0; i < FILE_SIZE / BLOCK_SIZE; i++) {
        if (index_table[i].start_block == start_block) {
            for (j = i; j < FILE_SIZE / BLOCK_SIZE - 1; j++) {
                index_table[j].start_block = index_table[j + 1].start_block;
                index_table[j].end_block = index_table[j + 1].end_block;
            }
            index_table[j].start_block = index_table[j].end_block = -1;
            next_block -= (index_table[i].end_block - index_table[i].start_block +
1);
            return;
        }
    }
    printf("Error: Block not found\n");
}

void display_index_table() {
    printf("Index table entries are: ");
    int i;
    for (i = 0; i < FILE_SIZE / BLOCK_SIZE; i++) {
        printf("(%d,%d) ", index_table[i].start_block, index_table[i].end_block);
    }
```

```
    printf("\n");
}
int main() {
    allocate_blocks(3);
    display_index_table();
    allocate_blocks(2);
    display_index_table();
    free_blocks(4);
    display_index_table();
    free_blocks(1);
    display_index_table();
    return 0;
}
```

OUTPUT:-



Lab 12

Aim: - Implement the concepts of Sequential list, Linked List and Indexed File allocations.

Theory:- Disk scheduling is a technique used by the operating system to order the access to disk blocks or sectors in order to improve the performance of the system. When a process requests data from the disk, the disk arm must move to the correct location to access the data. Disk scheduling algorithms determine the order in which these requests are serviced in order to minimize the average access time and maximize the overall throughput of the system.

There are several disk scheduling algorithms available, including:

First-Come-First-Serve (FCFS): This algorithm services the requests in the order in which they arrive, without regard to their location on the disk. This is a simple and easy-to-implement algorithm, but it can lead to poor performance if the requests are scattered across the disk.

Shortest-Seek-Time-First (SSTF): This algorithm services the request that is closest to the current position of the disk arm. It minimizes the seek time and provides better performance than FCFS.

SCAN: This algorithm moves the disk arm in one direction, servicing all the requests in that direction, and then moves in the opposite direction and services all the requests in that direction. It is a preemptive algorithm and can provide better performance than FCFS and SSTF.

C-SCAN: This algorithm is similar to SCAN but it only services requests in one direction, and when it reaches the end of the disk, it moves back to the beginning of the disk and services requests in the same direction. This algorithm can reduce the average waiting time for requests.

LOOK: This algorithm is similar to SCAN, but it only services requests in the direction of the next closest request, instead of moving all the way to the end of the disk. It is a preemptive algorithm and provides better performance than SCAN.

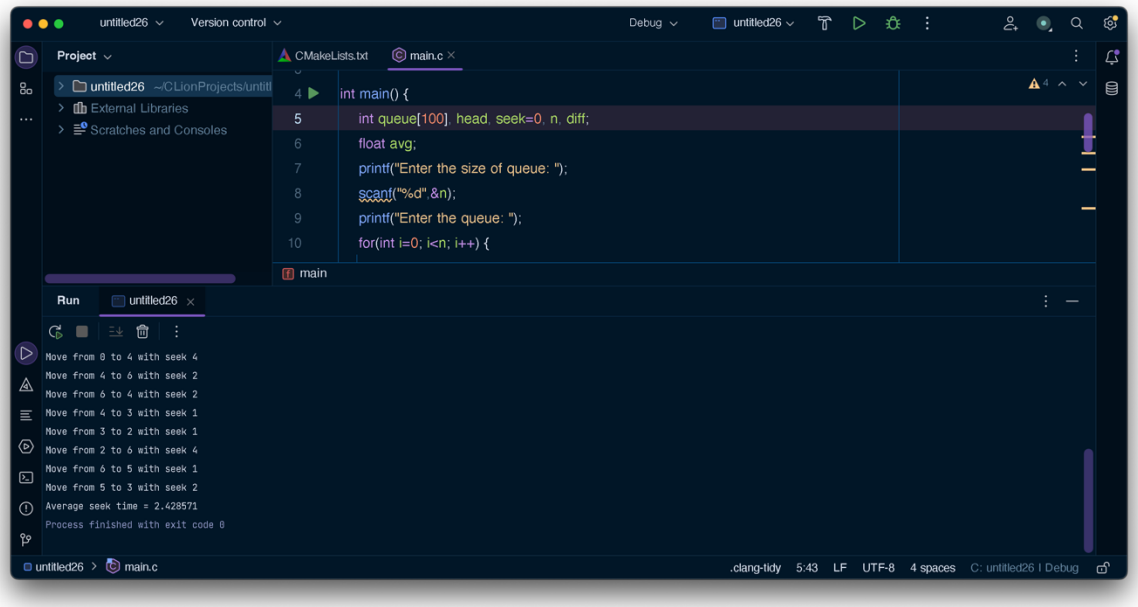
The FCFS disk scheduling algorithm is a non-preemptive algorithm in which the disk arm starts from the initial position and moves towards the first request in the queue. Once it reaches the first request, it serves it and moves to the next request in the queue. The process continues until all the requests have been served. In the implementation, the user is prompted to input the size of the queue, the queue itself, and the initial head position. The algorithm calculates the seek time for each movement of the disk arm and prints it along with the starting and ending positions. Finally, the average seek time is calculated and printed.

CODE:-

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

int main() {
    int queue[100], head, seek=0, n, diff;
    float avg;
    printf("Enter the size of queue: ");
    scanf("%d",&n);
    printf("Enter the queue: ");
    for(int i=0; i<n; i++) {
        scanf("%d",&queue[i]);
    }
    printf("Enter the initial head position: ");
    scanf("%d",&head);
    queue[n]=head;
```

```
    for(int i=0; i<=n; i++) {
        diff=abs(queue[i]-queue[i-1]);
        seek+=diff;
        printf("Move from %d to %d with seek %d\n",queue[i-
1],queue[i],diff);
    }
    avg=(float)seek/n;
    printf("Average seek time = %f",avg);
    return 0;
}
```

OUTPUT:-



SCAN:-

The SCAN disk scheduling algorithm is a preemptive algorithm in which the disk arm starts from the initial position and moves towards the last request in the queue in one direction. Once it reaches the last request, it changes its direction and moves towards the first request in the opposite direction, serving all the requests along the way. In the

implementation, the user is prompted to input the size of the queue, the queue itself, the initial head position, and the size of the disk. The algorithm sorts the queue in ascending order and then calculates the seek time for each movement of the disk arm and prints it along with the starting and ending positions. Finally, the average seek time is calculated and printed.

CODE:-

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

int main() {
    int queue[100], head, seek=0, n, end, max;
    float avg;
    printf("Enter the size of queue: ");
    scanf("%d",&n);
    printf("Enter the queue: ");
    for(int i=0; i<n; i++) {
        scanf("%d",&queue[i]);
    }
    printf("Enter the initial head position: ");
    scanf("%d",&head);
    printf("Enter the size of disk: ");
    scanf("%d",&end);
    queue[n]=head;
    n++;
    max=end-1;
    for(int i=0; i<n; i++) {
        for(int j=i+1; j<n; j++) {
            if(queue[i]>queue[j]) {
                int temp=queue[i];
                queue[i]=queue[j];
                queue[j]=temp;
            }
        }
    }
    int pos;
    for(int i=0; i<n; i++) {
        if(queue[i]==head) {
```

```
            pos=i;
            break;
        }
    }
    for(int i=pos; i>=0; i--) {
        seek+=abs(queue[i]-queue[i-1]);
        printf("Move from %d to %d with seek %d\n",queue[i],queue[i-
1],abs(queue[i]-queue[i-1]));
    }
    seek+=queue[pos+1];
    printf("Move from %d to %d with seek
%d\n",queue[pos],queue[pos+1],queue[pos+1]-queue[pos]);
    for(int i=pos+1; i<n-1; i++) {
        seek+=abs(queue[i]-queue[i+1]);
        printf("Move from %d to %d with seek
%d\n",queue[i],queue[i+1],abs(queue[i]-queue[i+1]));
    }
    avg=(float)seek/n;
    printf("Average seek time = %f",avg);
    return 0;
}
```

OUTPUT:-