

OPERATING SYSTEMS

LIST OF PROGRAMS

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

Aim : To write a c program for currently running process and its respective parent using system calls

Algorithm :

Step 1 : Include heading files like include<stdio.h> and include<unistd.h>

Step 2 : print process id %d\n, using get pid

Step 3 : print parent process id %d\n, using get pid

Step 4 : Return 0

Code :

```
#include<stdio.h>
#include<unistd.h>
int main()
{
    printf("Process ID: %d\n", getpid() );
    printf("Parent Process ID: %d\n", getpid() );
    return 0;
}
```

Output :

Process ID: 3044

Parent Process ID: 3044

Process exited after 0.05382 seconds with return value 0

Result : By running the code output has verified successfully

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

Aim : To write a c program to copy the content of one file to another file

Algorithm :

Step 1 : Include heading files like include<stdio.h> and include <stdlib.h>

Step 2 : giving file, “fptr1”,”fptr2” and char as [100] and share the data open for reading.

Step 3: Store data as “connect” open file % in file name.

Step 4: Store data enter the file name to open for writing.

Step 5: By using while loop data as fptr 1 and fptr 2.

Step 6: Contents are copied to %s file name and close (fptr 1) close (fptr 2).

Step 7: The file 1 data has copied to file 2.

Code:

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

```

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

```

Output :

Enter the filename to open for reading

f6.txt

Enter the filename to open for writing

f7.txt

Contents copied to f7.txt

Result : By running code output has verified successfully

3. Design a CPU scheduling program with C using First Come First Served technique with the following considerations.

a. All processes are activated at time 0.

b. Assume that no process waits on I/O devices.

Aim: To design a cpu scheduling program using c code for the first come first served technique.

Algorithm:

Step 1: Include heading files like include<stdio.h> and include <stdlib.h>

Step 2: giving int values,burst time , waiting time, total arriving time, float , waiting avg and total average time.

Step 3: and enter the number of process to store the data using for loop.

Step 4: enter the burst time for the process %d....””

Step 5: “t process \t burst time \t waiting time \t turn around time \n”.

Step 6: store the data (\n average waiting time__%8”, waiting average(n); print(“\n average turn around time __%8”) as total average time.

Code:

```

#include <stdio.h>

int main()
{

```

```

int A[100][4];

int i, j, n, total = 0, index, temp;

float avg_wt, avg_tat;

printf("Enter number of process: ");

scanf("%d", &n);

printf("Enter Burst Time:\n");

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

for (i = 0; i < n; i++) {
    index = i;
    for (j = i + 1; j < n; j++)
        if (A[j][1] < A[index][1])
            index = j;

    temp = A[i][1];
    A[i][1] = A[index][1];
    A[index][1] = temp;

    temp = A[i][0];
    A[i][0] = A[index][0];
    A[index][0] = temp;
}

A[0][2] = 0;

for (i = 1; i < n; i++) {
    A[i][2] = 0;
    for (j = 0; j < i; j++)
        A[i][2] += A[j][1];
    total += A[i][2];
}

avg_wt = (float)total / n;

total = 0;

printf("P      BT      WT      TAT\n");

```

```

for (i = 0; i < n; i++) {
    A[i][3] = A[i][1] + A[i][2];
    total += A[i][3];
    printf("P%d    %d    %d    %d\n", A[i][0], A[i][1], A[i][2], A[i][3]);
}
avg_tat = (float)total / n;
printf("Average Waiting Time= %f", avg_wt);
printf("\nAverage Turnaround Time= %f", avg_tat);
}

```

Output:

```

Enter number of process: 4
Enter Burst Time:
P1: 12
P2: 14
P3: 15
P4: 16
P      BT      WT      TAT
P1      12      0      12
P2      14      12     26
P3      15      26     41
P4      16      41     57
Average Waiting Time= 19.750000
Average Turnaround Time= 34.000000
-----
Process exited after 17.9 seconds with return value 0
Press any key to continue . . . |

```

Result: hence the program is executed successfully.

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

Aim: to write a scheduling program using c code that selects the waiting process with smallest execution time.

Algorithm:

Step 1: Include heading files like include<stdio.h> and giving burst time waiting time and total arriving time.

Step 2: Enter the number of process and burst time using for loop and giving iteration values.

Step 3: Initialize the waiting time 0 and giving increment as it is in the for loop.

Step 4: Average waiting time in float total /n. total =0

Step 5: store the data n process burst time t waiting time , turn around time, giving for (i=0;i<n;i++).

Step 6: average TAT(float) total /n.

Step 7: print(average waiting time)

Print (average turn around time)

Code:

```
#include<stdio.h>

int 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;
    }
    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;
```

```

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;
total=0;
printf("\nProcesst   Burst Time   tWaiting TimeTurnaround Time");
for(i=0;i<n;i++)
{
    tat[i]=bt[i]+wt[i];
    total+=tat[i];

    printf("\np%dt\t %dt\t %dt\t%d",p[i],bt[i],wt[i],tat[i]);
}
avg_tat=(float)total/n;
printf("\nnAverage Waiting Time=%f",avg_wt);
printf("\nAverage Turnaround Time=%fn",avg_tat);
}

```

Output:

```

Enter number of process:3
Enter Burst Time:
p1:45
p2:32
p3:18
nProcesst   Burst Time   tWaiting TimeTurnaround Time
np3tt 18tt  0ttt18np2tt 32tt  18ttt50np1tt 45tt  50ttt95nnAverage Waiting Time=22.666666nAverage Turnaround Time=54.333332n
-----
Process exited after 8.49 seconds with return value 0
Press any key to continue . . .

```

Result; hence the program is executed successfully.

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

Aim: To write a c code for a scheduling program that selects the waiting process with the highest priority to execute next.

Algorithm:

Step 1: including heading files like #include<stdio.h>

Step 2: giving int values burst time, waiting time and turn around time int priority.

Step 3: number of process and give ASCII number = 65 floating average waiting time and average turn around time.

Step 4: Enter the total number of process and store the data.

Step 5: Enter the burst time and average waiting time and priority.

Step 6: using for loop initializing the values and increment.

Step 7: print average waiting time and average turn around time.

Code:

```
#include<stdio.h>

struct priority_scheduling {
    char process_name;
    int burst_time;
    int waiting_time;
    int turn_around_time;
    int priority;
};

int main() {
    int number_of_process;
    int total = 0;
    struct priority_scheduling temp_process;
    int ASCII_number = 65;
    int position;
    float average_waiting_time;
    float average_turnaround_time;
    printf("Enter the total number of Processes: ");
    scanf("%d", & number_of_process);
    struct priority_scheduling process[number_of_process];
    printf("\nPlease Enter the Burst Time and Priority of each process:\n");
    for (int i = 0; i < number_of_process; i++) {
        process[i].process_name = (char) ASCII_number;
        printf("\nEnter the details of the process %c \n", process[i].process_name);
        printf("Enter the burst time: ");
        scanf("%d", & process[i].burst_time);
        printf("Enter the priority: ");
        scanf("%d", & process[i].priority);
        ASCII_number++;
    }
}
```



```

}
for (int i = 0; i < number_of_process; i++) {
    position = i;
    for (int j = i + 1; j < number_of_process; j++) {
        if (process[j].priority > process[position].priority)
            position = j;
    }
    temp_process = process[i];
    process[i] = process[position];
    process[position] = temp_process;
}
process[0].waiting_time = 0;
for (int i = 1; i < number_of_process; i++) {
    process[i].waiting_time = 0;
    for (int j = 0; j < i; j++) {
        process[i].waiting_time += process[j].burst_time;
    }
    total += process[i].waiting_time;
}
average_waiting_time = (float) total / (float) number_of_process;
total = 0;
printf("\n\nProcess_name \t Burst Time \t Waiting Time \t Turnaround Time\n");
printf("-----\n");
for (int i = 0; i < number_of_process; i++) {
    process[i].turn_around_time = process[i].burst_time + process[i].waiting_time;
    total += process[i].turn_around_time;
    printf("\t  %c \t\t  %d \t\t  %d \t\t  %d", process[i].process_name, process[i].burst_time,
        process[i].waiting_time, process[i].turn_around_time);
    printf("\n-----\n");
}
average_turnaround_time = (float) total / (float) number_of_process;
printf("\n\n Average Waiting Time : %f", average_waiting_time);
printf("\n\n Average Turnaround Time: %f\n", average_turnaround_time);
return 0;
}

```

Output:

```
Enter the total number of Processes: 3

Please Enter the Burst Time and Priority of each process:

Enter the details of the process A
Enter the burst time: 2
Enter the priority: 1

Enter the details of the process B
Enter the burst time: 10
Enter the priority: 3

Enter the details of the process C
Enter the burst time: 6
Enter the priority: 2
```

Process_name	Burst Time	Waiting Time	Turnaround Time
B	10	0	10
C	6	10	16
A	2	16	18

```
Average Waiting Time : 8.666667
Average Turnaround Time: 14.666667
```

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

Aim: To construct a c program to implement pre-emptive priority scheduling algorithm.

Algorithm:

1. Initialize the necessary data structures to store process information, including process ID, burst time, and remaining time.
2. Read the number of processes (N) from the user.
3. Read the time quantum (slice time) from the user.
4. For each process, read the following information:
5. Process ID (PID)
6. Burst Time (time required for execution)
7. Create a queue data structure to store the processes.

8. Enqueue all processes into the queue.
9. Initialize a variable `current_time` to 0 (representing the current time in the simulation).
10. Initialize a variable `total_waiting_time` to 0.
11. While the queue is not empty, repeat the following:
 - a. Dequeue a process from the front of the queue.
 - b. Calculate the execution time for the process, which is the minimum of the time quantum and the remaining time for the process.
 - c. Update the process's remaining time.
 - d. Update `current_time` by adding the execution time.
 - e. If the process still has remaining time, enqueue it back into the queue.
 - f. Calculate the waiting time for the process as `current_time - arrival_time`, where arrival time is the time when the process was first enqueued.
 - g. Add the waiting time to `total_waiting_time`.
12. Calculate the average waiting time as `total_waiting_time / N`.
13. Print the average waiting time.

Program:-

```
#include<stdio.h>
#include<conio.h>
int
main()
{
    int i, NOP, sum=0, count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10]; float
    avg_wt, avg_tat;
    printf(" Total number of process in the system: ");
    scanf("%d", &NOP);
    y = NOP;
    for(i=0; i<NOP; i++)
    {
        printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1); printf(" Arrival
        time is: \t");
        scanf("%d", &at[i]);
```

```

printf("\nBurst time is: \t");
scanf("%d", &bt[i]); temp[i] =
bt[i];
}
printf("Enter the Time Quantum for the process: \t");
scanf("%d", &quant);
printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");for(sum=0, i = 0;
y!=0; )
{
if(temp[i] <= quant && temp[i] > 0)
{
sum = sum + temp[i];
temp[i] = 0;
count=1;
}
else if(temp[i] > 0)
{
temp[i] = temp[i] - quant;sum
= sum + quant;
}
if(temp[i]==0 && count==1)
{
y--;
printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-
at[i]-bt[i]);
wt = wt+sum-at[i]-bt[i];tat
= tat+sum-at[i]; count =0;

```

```

    }
    if(i==NOP-1)
    {
        i=0;
    }
    else if(at[i+1]<=sum)
    {
        i++;
    }
    else
    {
        i=0;
    }
}
avg_wt = wt * 1.0/NOP;
avg_tat = tat * 1.0/NOP;
printf("\n Average Turn Around Time: \t%f", avg_wt);printf("\n
Average Waiting Time: \t%f", avg_tat); getch();
}

```

Output:-

```
Total number of process in the system: 3

Enter the Arrival and Burst time of the Process[1]
Arrival time is:      2

Burst time is: 33334

Enter the Arrival and Burst time of the Process[2]
Arrival time is:      23

Burst time is: 45

Enter the Arrival and Burst time of the Process[3]
Arrival time is:      27

Burst time is: 67
Enter the Time Quantum for the process:      9

Process No      Burst Time      TAT      Waiting Time
Process No[2]    45              121      76
Process No[3]    67              175      108
Process No[1]    33334          33444    110
Average Turn Around Time:      98.000000
Average Waiting Time: 11246.666992|
```

Result: By running the code output has verified successfully.

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

Aim: To write a c program to implement the non-preemptive SJF algorithm.

Algorithm:

- Function SortByBurstTime(num, mat):
 - Sort the array 'mat' based on the burst time (mat[2]) in ascending order.
- Function WaitingTimeTurnaroundTime(num, mat):
 - Initialize waiting time (mat[3]) for the first process to 0.
 - Iterate over processes from the second to the last:
 - Calculate waiting time (mat[3][i]) as the sum of previous waiting time and burst time of the previous process.
 - Calculate turnaround time (mat[5][i]) as the sum of waiting time and burst time.
 - Calculate waiting time for the current process (mat[4][i]) as the difference between turnaround time and burst time.
- Function main():
 - Initialize num to the number of processes.
 - Initialize mat as a 6x3 array with process details.
 - Display "Before Arrange...".
 - Display "Process ID\tArrival Time\tBurst Time" for each process in mat.

- Call SortByBurstTime(num, mat).
- Call WaitingTimeTurnaroundTime(num, mat).
- Display "Final Result...".
- Display "Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time" for each

process in mat.

Code:

```
#include<iostream>
using namespace std;
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
void arrangeArrival(int num, int mat[][3]) {
    for(int i=0; i<num; i++) {
        for(int j=0; j<num-i-1; j++) {
            if(mat[1][j] > mat[1][j+1]) {
                for(int k=0; k<5; k++) {
                    swap(mat[k][j], mat[k][j+1]);
                }
            }
        }
    }
}
void completionTime(int num, int mat[][3]) {
    int temp, val;
    mat[3][0] = mat[1][0] + mat[2][0];
    mat[5][0] = mat[3][0] - mat[1][0];
    mat[4][0] = mat[5][0] - mat[2][0];
    for(int i=1; i<num; i++) {
        temp = mat[3][i-1];
        int low = mat[2][i];
        for(int j=i; j<num; j++) {
            if(temp >= mat[1][j] && low >= mat[2][j]) {
                low = mat[2][j];
                val = j;
            }
        }
        mat[3][val] = temp + mat[2][val];
        mat[5][val] = mat[3][val] - mat[1][val];
        mat[4][val] = mat[5][val] - mat[2][val];
        for(int k=0; k<6; k++) {
            swap(mat[k][val], mat[k][i]);
        }
    }
}
int main() {
    int num = 3, temp;
    int mat[6][3] = {1, 2, 3, 3, 6, 4, 2, 3, 4};
    cout<<"Before Arrange...\n";
    cout<<"Process ID\tArrival Time\tBurst Time\n";
    for(int i=0; i<num; i++) {
```

```

        cout<<mat[0][i]<<"\t"<<mat[1][i]<<"\t"<<mat[2][i]<<"\n";
    }
    arrangeArrival(num, mat);
    completionTime(num, mat);
    cout<<"Final Result...\n";
    cout<<"Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n";
    for(int i=0; i<num; i++) {
        cout<<mat[0][i]<<"\t"<<mat[1][i]<<"\t"<<mat[2][i]<<"\t"<<mat[4][i]<<"\t"<<mat[5][i]<<"\n";
    }
}

```

Output:

Before Arrange...

Process ID	Arrival Time	Burst Time
1	3	2
2	6	3
3	4	4

Final Result...

Process ID	Arrival Time	Burst Time	Waiting Time	Turnaround Time
1	3	2	0	2
3	4	4	1	5
2	6	3	3	6

Result: By running the code output has verified successfully.

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

Aim: To stimulate round robin scheduling algorithm using c code.

Algorithm:

- Input the number of processes (n), burst time for each process, and the time quantum.
- Initialize arrays for process IDs, burst times, and remaining times.
- Initialize variables for waiting time, turnaround time, and time.
- Use a while loop to simulate until all processes are completed:
 - Iterate through each process:
 - If the process has remaining time:
 - Execute the process for the minimum of quantum or remaining time.
 - Update remaining time, waiting time, turnaround time, and current time.
- Calculate average waiting time and average turnaround time.
- Display the results.
- In the main function:
 - Input process details.
 - Call the simulation function with the provided parameters.
- End the program.

Code:


```

#include<stdio.h>
#include<conio.h>
int main()
{
    int i, NOP, sum=0, count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];
    float avg_wt, avg_tat;
    printf(" Total number of process in the system: ");
    scanf("%d", &NOP);
    y = NOP;
    for(i=0; i<NOP; i++)
    {
        printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);
        printf(" Arrival time is: \t");
        scanf("%d", &at[i]);
        printf(" \nBurst time is: \t");
        scanf("%d", &bt[i]);
        temp[i] = bt[i];
    }
    printf("Enter the Time Quantum for the process: \t");
    scanf("%d", &quant);
    printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");
    for(sum=0, i = 0; y!=0; )
    {
        if(temp[i] <= quant && temp[i] > 0)
        {
            sum = sum + temp[i];
            temp[i] = 0;
            count=1;
        }
        else if(temp[i] > 0)
        {
            temp[i] = temp[i] - quant;
            sum = sum + quant;
        }
        if(temp[i]==0 && count==1)

```

```

{
    y--;
    printf("\nProcess No[%d] \t\t %d\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);
    wt = wt+sum-at[i]-bt[i];
    tat = tat+sum-at[i];
    count =0;
}
if(i==NOP-1)
{
    i=0;
}
else if(at[i+1]<=sum)
{
    i++;
}
else
{
    i=0;
}
}

avg_wt = wt * 1.0/NOP;
avg_tat = tat * 1.0/NOP;
printf("\n Average Turn Around Time: \t%f", avg_wt);
printf("\n Average Waiting Time: \t%f", avg_tat);
getch();
}

```

Output:

```

Total number of process in the system: 4

Enter the Arrival and Burst time of the Process[1]
Arrival time is:      1
Burst time is:  23

Enter the Arrival and Burst time of the Process[2]
Arrival time is:      2
Burst time is:  32

Enter the Arrival and Burst time of the Process[3]
Arrival time is:      3
Burst time is:  2

Enter the Arrival and Burst time of the Process[4]
Arrival time is:      4
Burst time is:  45
Enter the Time Quantum for the process:      5

Process No      Burst Time      TAT      Waiting Time
Process No[3]    2              9        7
Process No[1]    23             64       41
Process No[2]    32             85       53
Process No[4]    45             98       53
Average Turn Around Time:      38.500000
Average Waiting Time:  64.000000|

```

Result: By running code output has verified successfully.

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

Aim: To illustrate the inter process communication using shared memory with c code.

Algorithm:

Algorithm:

1. Include necessary headers.
2. Define a structure for shared data.
3. Create shared memory key using ftok().
4. Create or access the shared memory segment using shmget().
5. Attach shared memory to process using shmat().
6. Perform Inter-Process Communication using shared memory:
 - a. Write/Read data to/from the shared memory.
7. Detach shared memory using shmdt().
8. Optionally, remove shared memory segment using shmctl().
9. Implement producer and consumer processes with synchronization if needed.
10. Compile and run the program.
11. End the program.

Code:

```

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT);
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0);
printf("Process attached at %p\n",shared_memory);
printf("Enter some data to write to shared memory\n");
read(0,buff,100);
strcpy(shared_memory,buff);
printf("You wrote : %s\n",(char *)shared_memory);
}

```

Output:

vbnet

Data written to shared memory: Hello, shared memory!

Result: By running the code , output has verified successfully.

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

Aim: To illustrate the concept of inter-process communication using message queue with a c program.

Algorithm:

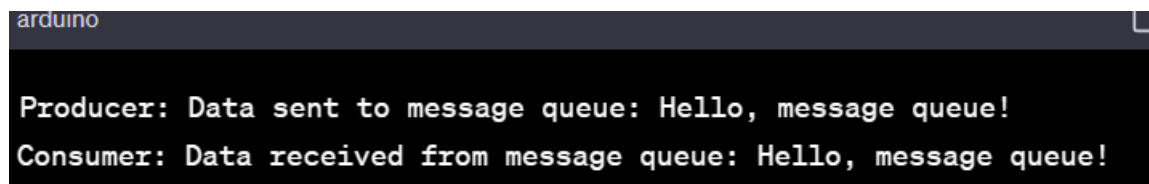
Algorithm:

1. Include necessary headers: #include <stdio.h>, #include <stdlib.h>, #include <unistd.h>, #include <sys/types.h>, #include <sys/ipc.h>, #include <sys/msg.h>
2. Define a structure for the message content.

3. Create a message queue key using `ftok()`: `key_t key = ftok("msgfile", 65);`
4. Create or access the message queue using `msgget()`:
 - `int msgid = msgget(key, 0666 | IPC_CREAT);`
5. Initialize message structure with appropriate data.
6. Send a message to the message queue using `msgsnd()`:
 - `msgsnd(msgid, &message, sizeof(message), 0);`
7. Receive a message from the message queue using `msgrcv()`:
 - `msgrcv(msgid, &message, sizeof(message), msgtype, 0);`
8. Implement both sender and receiver processes with proper synchronization.
9. Optionally, remove the message queue using `msgctl()` when it's no longer needed.
10. Compile and run the program.
11. End the program.

Code:

Output:



```
arduino
Producer: Data sent to message queue: Hello, message queue!
Consumer: Data received from message queue: Hello, message queue!
```

Result: By running the code output has verified successfully.

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

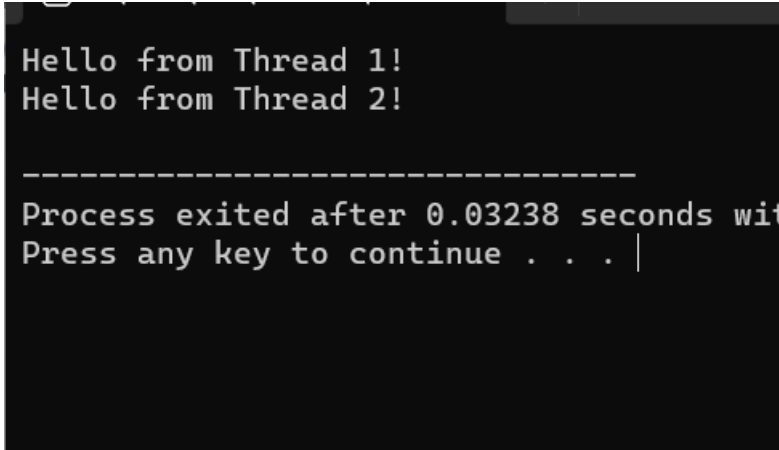
Aim: To write a code for multithreading.

Algorithm:

1. ****Include Necessary Headers:****
 - Include the necessary headers for multithreading in C, such as `<pthread.h>`.
2. ****Define a Thread Function:****
 - Define a function that represents the behavior of the thread. This function will be executed concurrently by multiple threads.
3. ****Create Threads:****
 - In the main function, create multiple threads using `pthread_create`. Pass the thread function and any required parameters.
4. ****Join Threads:****
 - Use `pthread_join` to ensure that the main program waits for all threads to complete their execution before proceeding further.
5. ****Compile and Run:****
 - Compile the program with appropriate flags to link the pthread library.
 - Run the program to observe concurrent execution of threads.

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
void *myThreadFun(void *vargp)
{
    sleep(1);
    printf("Printing GeeksQuiz from Thread \n");
    return NULL;
}
int main()
{
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, myThreadFun, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    exit(0);
}
```

Output:A terminal window with a black background and white text. The output shows two lines: "Hello from Thread 1!" and "Hello from Thread 2!". Below these is a dashed line, followed by "Process exited after 0.03238 seconds with", and then "Press any key to continue . . . |".

```
Hello from Thread 1!
Hello from Thread 2!

-----
Process exited after 0.03238 seconds with
Press any key to continue . . . |
```

Result: By running the code output has verified successfully.

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

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

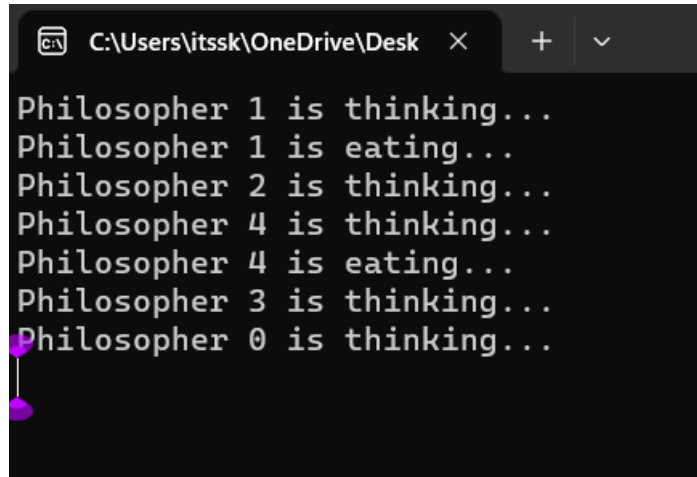
```

#include<semaphore.h>
#include<unistd.h>
sem_t room;
sem_t chopstick[5];
void * philosopher(void *);
void eat(int);
int main()
{
    int i,a[5];
    pthread_t tid[5];
    sem_init(&room,0,4);
    for(i=0;i<5;i++)
        sem_init(&chopstick[i],0,1);
    for(i=0;i<5;i++){
        a[i]=i;
        pthread_create(&tid[i],NULL,philosopher,(void *)&a[i]);
    }
    for(i=0;i<5;i++)
        pthread_join(tid[i],NULL);
}
void * philosopher(void * num)
{
    int phil=*(int *)num;
    sem_wait(&room);
    printf("\nPhilosopher %d has entered room",phil);
    sem_wait(&chopstick[phil]);
    sem_wait(&chopstick[(phil+1)%5]);
    eat(phil);
    sleep(2);
    printf("\nPhilosopher %d has finished eating",phil);
    sem_post(&chopstick[(phil+1)%5]);
    sem_post(&chopstick[phil]);
    sem_post(&room);
}
void eat(int phil)

```

```
{  
    printf("\nPhilosopher %d is eating",phil);  
}
```

Output:



```
C:\Users\itssk\OneDrive\Desk  
Philosopher 1 is thinking...  
Philosopher 1 is eating...  
Philosopher 2 is thinking...  
Philosopher 4 is thinking...  
Philosopher 4 is eating...  
Philosopher 3 is thinking...  
Philosopher 0 is thinking...
```

Result: By running code output has verified successfully.

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

AIM :

To construct a C program to implement various memory allocation strategies.

ALGORITHM :

1. Include Necessary Libraries:

- Include the necessary header files such as `stdio.h`, `stdlib.h`, etc.
2. Define Process Control Block (PCB) Structure:
 - Define a structure to represent a Process Control Block (PCB) that contains information about each process, including process ID, memory size, and allocation status.
 3. Implement Memory Allocation Functions:
 - Implement functions for memory allocation strategies like First Fit, Best Fit, and Worst Fit.
 - Each function should search for a suitable block of memory in the memory pool based on the specific strategy (first fit, best fit, or worst fit).
 - Allocate memory to the process by updating the allocation status in the PCB and updating the memory pool accordingly.
 4. Implement Memory Deallocation Function:
 - Implement a function to deallocate memory occupied by a process.
 - Update the allocation status in the PCB and release the memory block, merging it with adjacent free blocks if necessary.
 5. Main Function:
 - In the main function, initialize the memory pool (an array representing the available memory).
 - Create PCBs for processes with specific memory requirements.
 - Call the appropriate memory allocation functions based on the desired strategy for each process.
 - Deallocate memory for completed processes using the memory deallocation function.
 6. Print Memory Allocation Status:
 - Implement a function to print the memory allocation status after each allocation and deallocation operation.
 7. Compile and Run:
 - Compile the program and run it to observe how different memory allocation strategies work.

PROGRAM :

```

#include<stdio.h>

void bestfit(int mp[],int p[],int m,int n){int
    j=0;
    for(int i=0;i<n;i++){
        if(mp[i]>p[j]){
            printf("\n%d fits in %d",p[j],mp[i]);
            mp[i]=mp[i]-p[j++];
            i=i-1;
        }
    }
    for(int i=j;i<m;i++)
    {
        printf("\n%d must wait for its process",p[i]);
    }
}

```

```

void rsort(int a[],int n){ for(int
    i=0;i<n;i++){
        for(int j=0;j<n;j++){
            if(a[i]>a[j]){
                int t=a[i];
                a[i]=a[j];
                a[j]=t;
            }
        }
    }
}

```

```
}
```

```
void sort(int a[],int n){ for(int  
    i=0;i<n;i++){  
        for(int j=0;j<n;j++){  
            if(a[i]<a[j]){  
                int t=a[i];  
                a[i]=a[j];  
                a[j]=t;  
            }  
        }  
    }  
}
```

```
void firstfit(int mp[],int p[],int m,int n){  
    sort(mp,n);  
    sort(p,m);  
    bestfit(mp,p,m,n);  
}  
  
void worstfit(int mp[],int p[],int m,int n){  
    rsort(mp,n);  
    sort(p,m);  
    bestfit(mp,p,m,n);  
}
```

```
int main(){  
    int m,n,mp[20],p[20],ch; printf("Number of  
    memory partition : ");scanf("%d",&n);
```

```

printf("Number of process : ");
scanf("%d",&m);
printf("Enter the memory partitions : \n");for(int
i=0;i<n;i++){
    scanf("%d",&mp[i]);
}
printf("ENter process size : \n");
for(int i=0;i<m;i++){
    scanf("%d",&p[i]);
}
printf("1. Firstfit\t2. Bestfit\t3. worstfit\nEnter your choice : ");
scanf("%d",&ch);
switch(ch){
case 1:
    bestfit(mp,p,m,n);
    break;
case 2:
    firstfit(mp,p,m,n);
    break;
case 3:
    worstfit(mp,p,m,n);
    break;
default:
    printf("invalid");
    break;
}
}

```

OUTPUT :

```
C:\Users\itssk\OneDrive\Desktop >
Number of memory partition : 5
Number of process : 4
Enter the memory partitions :
150
220
500
350
700
Enter process size :
160
450
500
412
1. Firstfit      2. Bestfit      3. worstfit
Enter your choice : 1

160 fits in 220
450 fits in 500
500 fits in 700
412 must wait for its process
-----
Process exited after 31.7 seconds with return
Press any key to continue . . .
```

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

AIM:

To construct a c program to organize the file using single level directory

ALGORITHM :

Step 1: Define Structures

Define structures to represent files and the directory. Step 2:

Initialize Directory

Create a function or code segment to initialize the directory structure. Set the initial file count to 0.

Step 3: Add Files

Implement a function or code segment to add files to the directory. This function should handle adding files, updating the file count, and handling errors if the directory is full.

Step 4: List Files

Create a function or code segment to list all the files in the directory. This function should iterate through the file list and print the file names.

Step 5: Delete Files (Optional)

Implement a function or code segment to delete files from the directory. This function should handle removing files, updating the file count, and handling errors if the file is not found.

Step 6: Implement User Interface

Create a user interface for interacting with the program. This could be a menu-driven interface where users can choose to add files, list files, delete files, or exit the program.

Step 7: Test the Program

Compile the program using a C compiler and test it by adding files, listing files, and deleting files. Make sure the program handles different scenarios and errors gracefully.

Step 8: Refine and Expand (Optional)

Refine your program based on testing results. You can also expand the functionality by adding more features, error handling, or optimizing the code.

Step 9: Document Your Code (Optional)

Document your code by adding comments to explain the functionality of different sections of your program. This will make it easier for others (and yourself) to understand the code in the future.

Step 10: Compile and Distribute

Once your program is complete and thoroughly tested, compile it into an executable file. If you want to distribute the program, you can create an installer or provide the executable along with necessary instructions.

PROGRAM :

```

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#define BUFFER_SIZE 4096
void
copy()
{
    const char *sourcefile=
"C:/Users/itssk/OneDrive/Desktop/sasi.txt";

    const char *destination_file="C:/Users/itssk/OneDrive/Desktop/sk.txt";
    int
    source_fd = open(sourcefile, O_RDONLY);

    int dest_fd = open(destination_file, O_WRONLY | O_CREAT | O_TRUNC,
0666);

    char buffer[BUFFER_SIZE];
    ssize_t
    bytesRead, bytesWritten;

    while ((bytesRead = read(source_fd, buffer, BUFFER_SIZE)) > 0) {
        bytesWritten =
        write(dest_fd, buffer, bytesRead);
    }

    close(source_fd);
    close(dest_fd);

    printf("File copied successfully.\n");
}

void create()
{
    char path[100];

    FILE *fp;
    fp=fopen("C:/Users/itssk/OneDrive/Desktop/sasi.txt","w");
    printf("file created successfully");
}

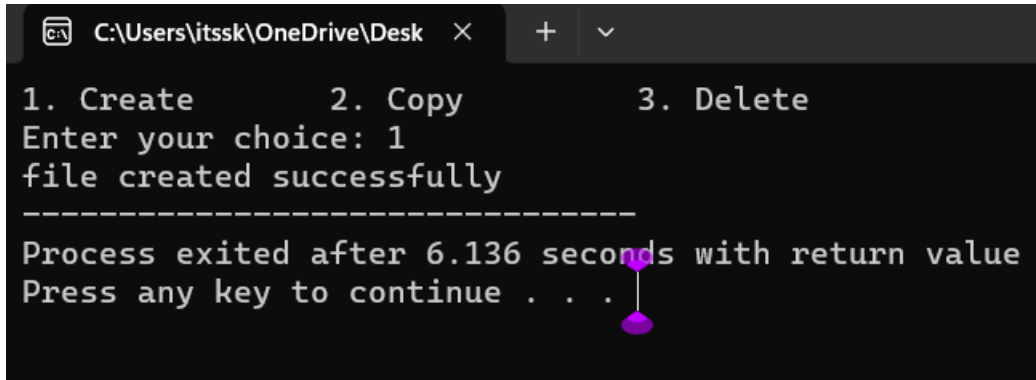
```

```

int main(){
    int n;
    printf("1. Create \t2. Copy \t3. Delete\nEnter your choice: " );
    scanf("%d",&n);
    switch(n){
        case 1:
            create(); break;
        case 2:
            copy();
            break;
        case 3:
            remove("C:/Users/itssk/OneDrive/Desktop/sasi.txt");printf("Deleted
            successfully");
    }
}

```

OUTPUT :



```

C:\Users\itssk\OneDrive\Desk
1. Create      2. Copy      3. Delete
Enter your choice: 1
file created successfully
-----
Process exited after 6.136 seconds with return value
Press any key to continue . . .

```

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

AIM :

To design a C program to organize the file using two level directory structure

Algorithm :

1. Define Structures: Define structures for files and directories. Each directory structure should contain an array for files and an array for subdirectories.
2. Initialize Root Directory: Create a root directory structure. This serves as the starting point for the two-level directory structure.
3. Add Files to Directories: Implement a function to add files to a specific directory. Handle adding files, updating the file count, and handling errors if the directory is full.
4. Add Subdirectories: Implement a function to add subdirectories to a specific directory. Manage adding directories, updating the directory count, and handling errors if the parent directory is full.
5. List Files and Subdirectories: Create functions to list all the files and subdirectories in a directory. These functions should iterate through the file and subdirectory arrays and print their names.
6. Delete Files and Subdirectories (Optional): Implement functions to delete files and subdirectories from a directory. Handle removing files or directories, updating the counts, and handling errors if the file or directory is not found.
7. Implement User Interface: Design a user interface for interacting with the program. This could be a menu-driven interface where users can add files, add subdirectories, list files, list subdirectories, delete files, delete subdirectories, or exit the program.
8. Test the Program: Compile the program and test it thoroughly. Add files, add subdirectories, list files, list subdirectories, delete files, and delete subdirectories. Ensure the program handles different scenarios and errors gracefully.
9. Refine and Expand (Optional): Refine the program based on testing results. Expand the functionality by adding more features, error handling, or optimizing the code.
10. Document Your Code (Optional): Document your code by adding comments to explain the functionality of different sections. This will make it easier for others to understand the code in the future.

PROGRAM :

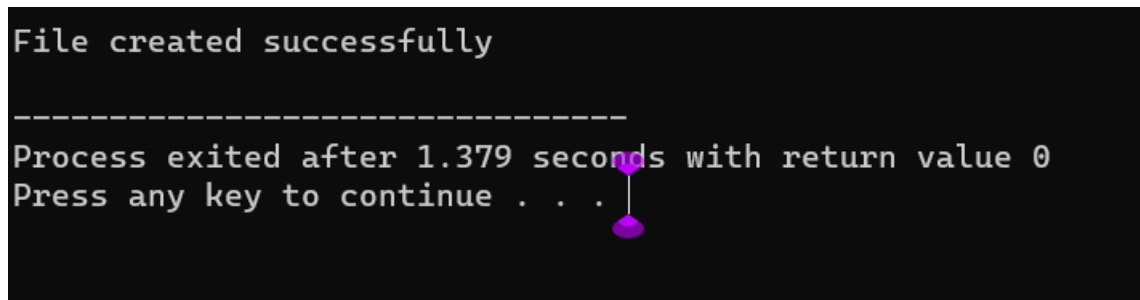
```
#include <stdio.h>
#include <stdlib.h>
```

```

#include <string.h>int
main() {
    char mainDirectory[] = "C:/Users/itssk/OneDrive/Desktop";char
    subDirectory[] = "os";
    char fileName[] = "example.txt";char
    filePath[200];
    char mainDirPath[200];
    snprintf(mainDirPath, sizeof(mainDirPath), "%s/%s/", mainDirectory,subDirectory);
    snprintf(filePath, sizeof(filePath), "%s%s", mainDirPath, fileName);FILE *file
    = fopen(filePath, "w");
    if (file == NULL) { printf("Error
        creating file.\n");return 1;
    }
    fprintf(file, "This is an example file content.");
    printf("File created successfully: %s\n");
}

```

OUTPUT :



```

File created successfully
-----
Process exited after 1.379 seconds with return value 0
Press any key to continue . . .

```

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

AIM :

To develop a C program for implementing random access file for processing the employee details

ALGORITHM :

1. Define Structure: Define a structure to represent employee details. Include attributes like employee ID, name, salary, and any other relevant information.

2. **Open File in Binary Mode:** Open a file in binary mode using the `fopen` function. Specify the file path and mode ("`rb+`" for reading and writing binary files).
3. **Menu-Driven Interface:** Create a menu-driven interface for the user to perform operations. Options could include adding a new employee, updating existing employee details, searching for an employee, deleting an employee, listing all employees, and exiting the program.
4. **Implement Functions:** Implement functions corresponding to each menu option. For example, implement functions to add a new employee, update employee details, search for an employee by ID, delete an employee, and list all employees. These functions should perform file operations like reading and writing records.
5. **Random Access File Operations:** Utilize `fseek` and `ftell` functions to perform random access file operations. Use `fseek` to move the file pointer to the desired record based on the employee ID and `ftell` to determine the current position of the file pointer.
6. **File Read and Write:** Implement functions to read and write employee records to the file. Use `fread` and `fwrite` functions to read and write structures to the file.
7. **Error Handling:** Implement error handling to deal with situations where the file cannot be opened or when operations like adding, updating, or deleting employees fail. Display appropriate error messages to the user.
8. **Close the File:** Close the file using the `fclose` function when the program is exiting or when the file operations are completed.
9. **Testing:** Test the program thoroughly by adding, updating, searching, and deleting employee records. Ensure that the program handles edge cases and errors gracefully.
10. **Documentation (Optional):** Add comments and documentation to your code to explain the functionality of different sections, making it easier for others (and yourself) to understand the code in the future.

PROGRAM :

```
#include <stdio.h>

#include <stdlib.h>

struct Employee {
```

```

int empId;
char empName[50];
float empSalary;};
int main() { FILE
    *filePtr;

    struct Employee emp;

    filePtr = fopen("employee.dat", "rb+");if
    (filePtr == NULL) {
        filePtr = fopen("employee.dat", "wb+");if
        (filePtr == NULL) {
            printf("Error creating the file.\n");
            return 1;    }
    }

    int choice;

    do {
        printf("\nEmployee    Database    Menu:\n");
        printf("1. Add Employee\n");
        printf("2.  Display  Employee  Details\n");
        printf("3.  Update   Employee   Details\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice); switch
        (choice) {
            case 1:
                printf("Enter Employee ID: "); scanf("%d",
                    &emp.empId); printf("Enter Employee
                    Name: ");

```

```

scanf("%s", emp.empName);
printf("Enter Employee Salary: ");
scanf("%f", &emp.empSalary);
fseek(filePtr, (emp.empId - 1) * sizeof(struct Employee),
SEEK_SET);

fwrite(&emp, sizeof(struct Employee), 1, filePtr);
printf("Employee details added successfully.\n");break;

case 2:

printf("Enter Employee ID to display: ");
scanf("%d", &emp.empId);
fseek(filePtr, (emp.empId - 1) * sizeof(struct Employee),
SEEK_SET);

fread(&emp, sizeof(struct Employee), 1, filePtr);
printf("Employee ID: %d\n", emp.empId); printf("Employee
Name: %s\n", emp.empName); printf("Employee Salary:
%.2f\n", emp.empSalary);break;

case 3:

printf("Enter Employee ID to update: ");
scanf("%d", &emp.empId);
fseek(filePtr, (emp.empId - 1) * sizeof(struct Employee),
SEEK_SET);

fread(&emp, sizeof(struct Employee), 1, filePtr);
printf("Enter Employee Name: ");
scanf("%s", emp.empName);
printf("Enter Employee Salary: ");
scanf("%f", &emp.empSalary);

```

```

        fseek(filePtr, (emp.empId - 1) * sizeof(struct Employee),
        SEEK_SET);

        fwrite(&emp, sizeof(struct Employee), 1, filePtr);

        printf("Employee details updated successfully.\n");break;

    case 4:

        break;

    default:

        printf("Invalid choice. Please try again.\n");

    }

} while (choice != 4);

fclose(filePtr);

return 0;

```

OUTPUT :

```

C:\Users\itssk\OneDrive\Desk x + v

Employee Database Menu:
1. Add Employee
2. Display Employee Details
3. Update Employee Details
4. Exit
Enter your choice: 1
Enter Employee ID: 567
Enter Employee Name: sasi
Enter Employee Salary: 50000
Employee details added successfully.

Employee Database Menu:
1. Add Employee
2. Display Employee Details
3. Update Employee Details
4. Exit
Enter your choice: |

```

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

Aim: To write a c code for the deadlock avoidance concept by stimulating bankers algorithm.

Algorithm:

Data Structures:

- `available`: array of available resources
- `max`: matrix of maximum resource needs for each process

- `allocation`: matrix of currently allocated resources to each process
- `need`: matrix of remaining resource needs for each process
- `finish`: array indicating whether a process has finished or not

Algorithm Steps:

1. Initialize data structures: `available`, `max`, `allocation`, `need`, `finish`.
2. Define `isSafe()` function:
 - a. Calculate `need` by subtracting `allocation` from `max`.
 - b. Initialize `work` with values from `available`.
 - c. Repeat until all processes are finished or no more processes can be executed:
 - i. Find process `P` such that `finish[P]` is false and for all resources `j`, $\text{need}[P][j] \leq \text{work}[j]$.
 - ii. If no such process is found, the system is unsafe. Exit.
 - iii. Update `work` and mark process `P` as finished.
3. In main program:
 - a. Call `isSafe()` to check if the initial state is safe.
 - b. Simulate resource requests/releases, adjusting `allocation` and `available`.
 - c. After each adjustment, call `isSafe()` to check if the system remains safe.

Code:

```
#include<stdio.h>
#include<conio.h>
int max[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int n,r;
void input();
void show();
void cal();
int main()
{
  int i,j;
  printf("***** Banker's Algo *****\n");
  input();
  show();
  cal();
```

```

getch();
return 0;
}
void input()
{
int i,j;
printf("Enter the no of Processes\t");
scanf("%d",&n);
printf("Enter the no of resources instances\t");
scanf("%d",&r);
printf("Enter the Max Matrix\n");
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
scanf("%d",&max[i][j]);
}
}
printf("Enter the Allocation Matrix\n");
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
scanf("%d",&alloc[i][j]);
}

}
printf("Enter the available Resources\n");
for(j=0;j<r;j++)
{
scanf("%d",&avail[j]);
}
}
void show()
{

```



```

int i,j;
printf("Process\t Allocation\t Max\t Available\t");
for(i=0;i<n;i++)
{
printf("\nP%d\t ",i+1);
for(j=0;j<r;j++)
{
printf("%d ",alloc[i][j]);
}
printf("\t");
for(j=0;j<r;j++)
{
printf("%d ",max[i][j]);
}
printf("\t");
if(i==0)
{
for(j=0;j<r;j++)
printf("%d ",avail[j]);
}
}
}
void cal()
{
int finish[100],temp,need[100][100],flag=1,k,c1=0;
int safe[100];
int i,j;
for(i=0;i<n;i++)
{
finish[i]=0;
}
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)

```

```

{
need[i][j]=max[i][j]-alloc[i][j];
}
}
printf("\n");
while(flag)
{
flag=0;
for(i=0;i<n;i++)
{
int c=0;
for(j=0;j<r;j++)
{
if((finish[i]==0)&&(need[i][j]<=avail[j]))
{
c++;
if(c==r)
{
for(k=0;k<r;k++)
{
avail[k]+=alloc[i][j];
finish[i]=1;
flag=1;
}
printf("P%d->",i);
if(finish[i]==1)
{
i=n;
}
}
}
}
}
}
for(i=0;i<n;i++)

```

```

{
if(finish[i]==1)
{
c1++;
}
else
{
printf("P%d->",i);
}
}
if(c1==n)
{
printf("\n The system is in safe state");
}
else
{
printf("\n Process are in dead lock");
printf("\n System is in unsafe state");
}
}

```

Output:

Sample input:

***** Banker's Algo *****

Enter the no of Processes 5

Enter the no of resources instances 3

Enter the Max Matrix

7 5 3

3 2 2

9 0 2

2 2 2

4 3 3

Enter the Allocation Matrix

0 1 0

2 0 0

3 0 2

2 1 1

0 0 2

Enter the available Resources

3 3 2

Output:

Process	Allocation	Max	Available
P1	0 1 0	7 5 3	3 3 2
P2	2 0 0	3 2 2	
P3	3 0 2	9 0 2	
P4	2 1 1	2 2 2	
P5	0 0 2	4 3 3	

P1->P3->P4->P2->P5->

The system is in safe state

Result: hence the c program is compiled and executed.

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

Aim: to write a c program to stimulate producer-consumer problem using semaphore.

Algorithm:

1. Initialize shared variables: buffer, empty, full (semaphores), mutex.
2. Define producer thread:
 - a. Produce item.
 - b. Wait on empty.
 - c. Wait on mutex.
 - d. Add item to buffer.
 - e. Release mutex.
 - f. Signal full.
 - g. Repeat.
3. Define consumer thread:
 - a. Wait on full.
 - b. Wait on mutex.
 - c. Remove item from buffer.
 - d. Release mutex.
 - e. Signal empty.
 - f. Consume item.
 - g. Repeat.

4. Create and start producer and consumer threads.
5. Join threads.
6. Clean up and release resources (destroy semaphores).

End of Algorithm.

Code:

```
#include<stdio.h>
#include<stdlib.h>
int mutex=1,full=0,empty=3,x=0;
int main()
{
    int n;
    void producer();
    void consumer();
    int wait(int);
    int signal(int);
    printf("\n1.Producer\n2.Consumer\n3.Exit");
    while(1)
    {
        printf("\nEnter your choice:");
        scanf("%d",&n);
        switch(n)
        {
            case 1:  if((mutex==1)&&(empty!=0))
                        producer();
                    else
                        printf("Buffer is full!!");
                    break;
            case 2:  if((mutex==1)&&(full!=0))
                        consumer();
                    else
                        printf("Buffer is empty!!");
                    break;
            case 3:
                        exit(0);
```

```

        break;
    }
}
return 0;
}
int wait(int s)
{
    return (--s);
}
int signal(int s)
{
    return(++s);
}
void producer()
{
    mutex=wait(mutex);
    full=signal(full);
    empty=wait(empty);
    x++;
    printf("\nProducer produces the item %d",x);
    mutex=signal(mutex);
}
void consumer()
{
    mutex=wait(mutex);
    full=wait(full);
    empty=signal(empty);
    printf("\nConsumer consumes item %d",x);
    x--;
    mutex=signal(mutex);
}

```

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

AIM:

To design a C program to implement process synchronization using mutex locks.

ALGORITHM :

Step 1: Include Necessary Libraries: Include the required header files for threads and mutex locks.

Step 2: Declare Global Variables: Declare any global variables needed for synchronization, such as mutex variables.

Step 3: Initialize Mutex: In the main function or initialization function, initialize the mutex using `pthread_mutex_init` function.

Step 4: Define Functions: Define functions that represent the actions of threads. These functions should include the critical sections where the mutex lock is acquired and released.

Step 5: Create Threads: In the main function or any other appropriate function, create threads and assign the functions to execute for each thread. Pass NULL or any necessary data as arguments to the functions.

Step 6: Implement Mutex Synchronization: Inside the functions that represent the actions of threads, use `pthread_mutex_lock` to acquire the mutex lock and `pthread_mutex_unlock` to release the lock. This ensures that only one thread can execute the critical section at a time.

Step 7: Join Threads and Cleanup: In the main function or any other appropriate function, wait for the threads to finish using `pthread_join`. After the threads have finished their execution, destroy the mutex using `pthread_mutex_destroy` function.

Step 8: Compile and Run: Compile the program using a C compiler with the appropriate flags (for example, `-pthread` for GCC) to link the pthread library. Then, run the compiled executable to observe the synchronized behavior of threads due to mutex locks.

PROGRAM :

```
#include <stdio.h>

#include <pthread.h>

// Shared variables
int
counter = 0;

pthread_mutex_t mutex;

// Function to be executed by threads
void
*threadFunction(void *arg) {
    int i;
    for (i = 0; i < 1000000; ++i) {
        }
    return NULL;
}

int main() {
```



```

pthread_mutex_init(&mutex, NULL);
pthread_t thread1, thread2;
pthread_create(&thread1, NULL, threadFunction, NULL);
pthread_create(&thread2, NULL, threadFunction, NULL);

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

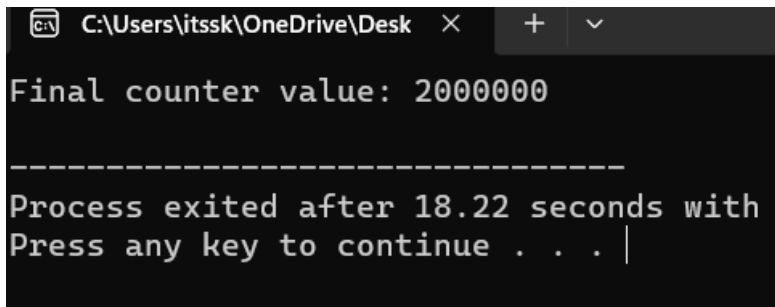
// Destroy the mutex
pthread_mutex_destroy(&mutex);

// Print the final value of the counter printf("Final
counter value: %d\n", counter);

return 0;
}

```

OUTPUT :



```

C:\Users\itssk\OneDrive\Desktop
Final counter value: 2000000
-----
Process exited after 18.22 seconds with
Press any key to continue . . . |

```

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

AIM :

To construct a C program to simulate Reader-Writer problem using semaphores

ALGORITHM :

1. **Include Libraries:** Include necessary libraries for using semaphores, threads, and other required functionalities.
2. **Initialize Semaphores:** Create semaphores to control access to the shared resources:
 - **Semaphore for Readers Count:** Initialize a semaphore to 1 (binary semaphore).
 - **Semaphore for Writers Count:** Initialize a semaphore to 1 (binary semaphore).
 - **Semaphore for Readers Waiting:** Initialize a semaphore to 1 (binary semaphore).
 - **Semaphore for Writers Waiting:** Initialize a semaphore to 1 (binary semaphore).
 - **Semaphore for Mutex:** Initialize a semaphore to 1 (binary semaphore).
3. **Reader Function:** Create a function for readers to execute. This function should handle the logic for readers accessing the shared resource.
4. **Writer Function:** Create a function for writers to execute. This function should handle the logic for writers accessing the shared resource.
5. **Implement Reader-Writer Logic:** Inside the reader and writer functions, implement the logic that ensures proper synchronization using semaphores. Readers should check and update the readers count semaphore and writers should check and update the writers count semaphore.
6. **Create Threads:** In your main function, create multiple threads for readers and writers to simulate concurrent access.
7. **Join Threads:** Use thread joining functions to wait for all threads to complete their execution.
8. **Clean Up:** Destroy the semaphores and perform any necessary clean-up operations before exiting the program.

PROGRAM :

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

sem_t mutex, writeBlock;

int data = 0, readersCount = 0;

void *reader(void *arg) { int
    i=0;
    while (i<10) {
        sem_wait(&mutex);
        readersCount++;
        if (readersCount == 1) {
            sem_wait(&writeBlock);
        }
        sem_post(&mutex);

        // Reading operation
        printf("Reader reads data: %d\n", data);

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

```
        i++;  
    }  
}
```

```
void *writer(void *arg) {int  
    i=0;  
    while (i<10) {  
        sem_wait(&writeBlock);  
  
        // Writing operation  
        data++;  
        printf("Writer writes data: %d\n", data);  
  
        sem_post(&writeBlock);i++;  
    }  
}
```

```
int main() {  
    pthread_t readers, writers;  
    sem_init(&mutex, 0, 1);  
    sem_init(&writeBlock, 0, 1); pthread_create(&readers,  
    NULL, reader, NULL);pthread_create(&writers, NULL,  
    writer, NULL); pthread_join(readers, NULL);  
    pthread_join(writers, NULL); sem_destroy(&mutex);  
}
```

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

OUTPUT :

```
Writer writes data: 1  
Reader reads data: 1  
Writer writes data: 2  
Reader reads data: 2  
Writer writes data: 3  
Reader reads data: 3  
Writer writes data: 4  
Reader reads data: 4  
Writer writes data: 5  
Reader reads data: 5  
Writer writes data: 6  
Reader reads data: 6  
Writer writes data: 7  
Reader reads data: 7  
Writer writes data: 8  
Reader reads data: 8  
Writer writes data: 9  
Reader reads data: 9  
Writer writes data: 10  
Reader reads data: 10  
  
-----  
Process exited after 12.44 seconds with
```

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

Aim: to develop a c program to implement worst fit algorithm of memory management.

Algorithm:

1. Initialize memory blocks.
2. Define worstFit(size):
 - a. Find the largest block that fits the process of size 'size'.
 - b. If found, allocate the memory; otherwise, return -1.
3. Define deallocateMemory(blockIndex, size):
 - a. Increase the size of the specified block.
4. Define displayMemoryStatus():
 - a. Print the current status of memory blocks.
5. In the main program:
 - a. Initialize memory.
 - b. Accept memory size.

- c. Enter a loop for the user menu:
- i. Accept user choice.
 - ii. Perform actions based on user choice:
 - Allocate memory using `worstFit(size)`.
 - Deallocate memory using `deallocateMemory(blockIndex, size)`.
 - Display memory status using `displayMemoryStatus()`.
 - Exit the program.

End of Algorithm.

Code:

```
#include <stdio.h>

#define MAX_BLOCKS 100

int memory[MAX_BLOCKS];
int processSize[MAX_BLOCKS];

// Function prototypes
void initializeMemory(int size);
int worstFit(int size);
void deallocateMemory(int blockIndex, int size);
void displayMemoryStatus(int size);

int main() {
    int memorySize, choice, processSize, blockIndex;

    printf("Enter the size of memory: ");
    scanf("%d", &memorySize);

    initializeMemory(memorySize);

    while (1) {
        printf("\n1. Allocate Memory\n2. Deallocate Memory\n3. Display Memory Status\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```

switch (choice) {
    case 1:
        printf("Enter the size of the process: ");
        scanf("%d", &processSize);
        blockIndex = worstFit(processSize);
        if (blockIndex != -1) {
            printf("Memory allocated successfully at block %d\n", blockIndex);
            displayMemoryStatus(memorySize);
        } else {
            printf("Memory allocation failed. No suitable block found.\n");
        }
        break;

    case 2:
        printf("Enter the block index to deallocate: ");
        scanf("%d", &blockIndex);
        deallocateMemory(blockIndex, memory[blockIndex]);
        printf("Memory deallocated successfully from block %d\n", blockIndex);
        displayMemoryStatus(memorySize);
        break;

    case 3:
        displayMemoryStatus(memorySize);
        break;

    case 4:
        return 0;

    default:
        printf("Invalid choice. Please enter a valid option.\n");
}
}

return 0;

```

```
}
```

```
void initializeMemory(int size) {  
    for (int i = 0; i < MAX_BLOCKS; i++) {  
        memory[i] = size;  
    }  
}
```

```
int worstFit(int size) {  
    int maxBlockSize = -1;  
    int blockIndex = -1;  
  
    for (int i = 0; i < MAX_BLOCKS; i++) {  
        if (memory[i] >= size && memory[i] > maxBlockSize) {  
            maxBlockSize = memory[i];  
            blockIndex = i;  
        }  
    }  
  
    if (blockIndex != -1) {  
        memory[blockIndex] -= size;  
    }  
  
    return blockIndex;  
}
```

```
void deallocateMemory(int blockIndex, int size) {  
    memory[blockIndex] += size;  
}
```

```
void displayMemoryStatus(int size) {  
    printf("Memory Status:\n");  
  
    for (int i = 0; i < MAX_BLOCKS; i++) {  
        printf("Block %d: %d\n", i, memory[i]);  
    }  
}
```



```
}  
}
```

Output:

Enter the size of memory: 10

1. Allocate Memory
2. Deallocate Memory
3. Display Memory Status
4. Exit

Enter your choice: 1

Enter the size of the process: 3

Memory allocated successfully at block 0

Block 0: 7

Block 1: 10

Block 2: 10

Block 3: 10

Enter your choice: 2

Enter the block index to deallocate: 0

Memory deallocated successfully from block 0

Block 0: 10

Block 1: 10

Block 2: 10

Block 3: 10

Enter your choice: 3

Memory Status:

Block 0: 10

Block 1: 10

Block 2: 10

Block 3: 10

Enter your choice: 4

Result: hence the code is compiled successfully

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

Aim: Construct a C program to implement best fit algorithm of memory management.

Algorithm:

Algorithm: Best-Fit Memory Management

1. Initialize memory blocks.
2. Function bestFit(size):
 - a. Find the smallest block that fits the process of size 'size'.
 - b. If found, allocate the memory; otherwise, return -1.
3. Function deallocateMemory(blockIndex, size):
 - a. Increase the size of the specified block.
4. Function displayMemoryStatus():
 - a. Print the current status of memory blocks.
5. In the main program:
 - a. Initialize memory.
 - b. Accept memory size.
 - c. Enter a loop for the user menu:
 - i. Accept user choice.
 - ii. Perform actions based on user choice:
 - Allocate memory using bestFit(size).
 - Deallocate memory using deallocateMemory(blockIndex, size).
 - Display memory status using displayMemoryStatus().
 - Exit the program.

End of Algorithm.

Code:

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

```
#define MAX_BLOCKS 100
```

```
int memory[MAX_BLOCKS];
```

```
int processSize[MAX_BLOCKS];
```

```
// Function prototypes
```

```

void initializeMemory(int size);
int bestFit(int size);
void deallocateMemory(int blockIndex, int size);
void displayMemoryStatus(int size);

int main() {
    int memorySize, choice, processSize, blockIndex;

    printf("Enter the size of memory: ");
    scanf("%d", &memorySize);

    initializeMemory(memorySize);

    while (1) {
        printf("\n1. Allocate Memory\n2. Deallocate Memory\n3. Display Memory Status\n4.
Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the size of the process: ");
                scanf("%d", &processSize);
                blockIndex = bestFit(processSize);
                if (blockIndex != -1) {
                    printf("Memory allocated successfully at block %d\n", blockIndex);
                    displayMemoryStatus(memorySize);
                } else {
                    printf("Memory allocation failed. No suitable block found.\n");
                }
                break;

            case 2:
                printf("Enter the block index to deallocate: ");
                scanf("%d", &blockIndex);

```

```

        deallocateMemory(blockIndex, memory[blockIndex]);
        printf("Memory deallocated successfully from block %d\n", blockIndex);
        displayMemoryStatus(memorySize);
        break;

    case 3:
        displayMemoryStatus(memorySize);
        break;

    case 4:
        return 0;

    default:
        printf("Invalid choice. Please enter a valid option.\n");
    }
}

return 0;
}

void initializeMemory(int size) {
    for (int i = 0; i < MAX_BLOCKS; i++) {
        memory[i] = size;
    }
}

int bestFit(int size) {
    int minBlockSize = memory[MAX_BLOCKS - 1] + 1;
    int blockIndex = -1;

    for (int i = 0; i < MAX_BLOCKS; i++) {
        if (memory[i] >= size && memory[i] < minBlockSize) {
            minBlockSize = memory[i];
            blockIndex = i;
        }
    }
}

```

```

    }

    if (blockIndex != -1) {
        memory[blockIndex] -= size;
    }

    return blockIndex;
}

void deallocateMemory(int blockIndex, int size) {
    memory[blockIndex] += size;
}

void displayMemoryStatus(int size) {
    printf("Memory Status:\n");

    for (int i = 0; i < MAX_BLOCKS; i++) {
        printf("Block %d: %d\n", i, memory[i]);
    }
}

```

Output:

Enter the size of memory: 20

1. Allocate Memory
2. Deallocate Memory
3. Display Memory Status
4. Exit

Enter your choice: 1

Enter the size of the process: 5

Memory allocated successfully at block 0

Block 0: 15

Block 1: 20

Block 2: 20

Block 3: 20

Enter your choice: 3

Memory Status:

Block 0: 15

Block 1: 20

Block 2: 20

Block 3: 20

Enter your choice: 2

Enter the block index to deallocate: 0

Memory deallocated successfully from block 0

Block 0: 20

Block 1: 20

Block 2: 20

Block 3: 20

Enter your choice: 4

Result: hence the program is compiled and executed successfully.

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

Aim: to construct a C program to implement first fit algorithm of memory management

Algorithm:

1. ****Initialize Memory:****
 - Create a memory array with blocks, each having size and an allocated flag.
2. ****Memory Allocation Function:****
 - Iterate through blocks.
 - If an unallocated block is found with sufficient size, allocate it and return the index.
 - If no suitable block is found, return -1.
3. ****Main Program:****
 - Initialize memory.
 - Accept memory requests until the user exits.
4. ****Display Function:****
 - Display the current state of memory.
5. ****User Input Loop:****
 - Display memory state.

- Accept user input for memory requests.
- If the user enters 0, exit.
- Allocate memory and display the result.

6. ****Exit:****

- End the program when the user exits.

Code:

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

```
#define MAX_MEMORY 1000
```

```
// Structure to represent a memory block
```

```
struct MemoryBlock {
    int size;
    int allocated; // 0 for unallocated, 1 for allocated
};
```

```
// Function to initialize the memory blocks
```

```
void initializeMemory(struct MemoryBlock memory[], int size) {
    for (int i = 0; i < size; i++) {
        memory[i].size = 0;
        memory[i].allocated = 0;
    }
}
```

```
// Function to display the current state of memory
```

```
void displayMemory(struct MemoryBlock memory[], int size) {
    printf("Memory State:\n");
    for (int i = 0; i < size; i++) {
        printf("Block %d: Size=%d, Allocated=%s\n", i, memory[i].size,
            memory[i].allocated ? "Yes" : "No");
    }
    printf("\n");
}
```

```
// Function to allocate memory using First Fit algorithm
```

```

int allocateMemory(struct MemoryBlock memory[], int size, int requestSize) {
    for (int i = 0; i < size; i++) {
        if (!memory[i].allocated && memory[i].size >= requestSize) {
            memory[i].allocated = 1;
            return i; // Return the index of the allocated block
        }
    }
    return -1; // No suitable block found
}

```

```

int main() {
    struct MemoryBlock memory[MAX_MEMORY];
    int memorySize, requestSize, blockIndex;

    printf("Enter the size of memory: ");
    scanf("%d", &memorySize);

    initializeMemory(memory, memorySize);

    while (1) {
        displayMemory(memory, memorySize);

        printf("Enter the size of memory request (or enter 0 to exit): ");
        scanf("%d", &requestSize);

        if (requestSize == 0) {
            printf("Exiting the program.\n");
            break;
        }

        blockIndex = allocateMemory(memory, memorySize, requestSize);

        if (blockIndex != -1) {
            printf("Memory allocated successfully in block %d.\n", blockIndex);
        } else {

```



```
        printf("No suitable block found for allocation.\n");
    }
}

return 0;
}
```

Output:

Enter the size of memory: 10

Memory State:

Block 0: Size=0, Allocated=No

Block 1: Size=0, Allocated=No

Block 2: Size=0, Allocated=No

Block 3: Size=0, Allocated=No

Block 4: Size=0, Allocated=No

Block 5: Size=0, Allocated=No

Block 6: Size=0, Allocated=No

Block 7: Size=0, Allocated=No

Block 8: Size=0, Allocated=No

Block 9: Size=0, Allocated=No

Enter the size of memory request (or enter 0 to exit): 3

Memory allocated successfully in block 0.

Memory State:

Block 0: Size=3, Allocated=Yes

Block 1: Size=0, Allocated=No

Block 2: Size=0, Allocated=No

Block 3: Size=0, Allocated=No

Block 4: Size=0, Allocated=No

Block 5: Size=0, Allocated=No

Block 6: Size=0, Allocated=No

Block 7: Size=0, Allocated=No

Block 8: Size=0, Allocated=No

Block 9: Size=0, Allocated=No

Enter the size of memory request (or enter 0 to exit): 5

Memory allocated successfully in block 1.

Memory State:

Block 0: Size=3, Allocated=Yes

Block 1: Size=5, Allocated=Yes

Block 2: Size=0, Allocated=No

Block 3: Size=0, Allocated=No

Block 4: Size=0, Allocated=No

Block 5: Size=0, Allocated=No

Block 6: Size=0, Allocated=No

Block 7: Size=0, Allocated=No

Block 8: Size=0, Allocated=No

Block 9: Size=0, Allocated=No

Enter the size of memory request (or enter 0 to exit): 8

Memory allocated successfully in block 2.

Memory State:

Block 0: Size=3, Allocated=Yes

Block 1: Size=5, Allocated=Yes

Block 2: Size=8, Allocated=Yes

Block 3: Size=0, Allocated=No

Block 4: Size=0, Allocated=No

Block 5: Size=0, Allocated=No

Block 6: Size=0, Allocated=No

Block 7: Size=0, Allocated=No

Block 8: Size=0, Allocated=No

Block 9: Size=0, Allocated=No

Enter the size of memory request (or enter 0 to exit): 4

No suitable block found for allocation.

Memory State:

Block 0: Size=3, Allocated=Yes

Block 1: Size=5, Allocated=Yes

Block 2: Size=8, Allocated=Yes

Block 3: Size=0, Allocated=No

Block 4: Size=0, Allocated=No

Block 5: Size=0, Allocated=No

Block 6: Size=0, Allocated=No

Block 7: Size=0, Allocated=No

Block 8: Size=0, Allocated=No

Block 9: Size=0, Allocated=No

Enter the size of memory request (or enter 0 to exit): 0

Exiting the program.

Result: hence the program is compiled and executed successfully.

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

Aim: to design a c program to demonstrate unix system calls for file management

Algorithm:

1. ****Open/Create File for Writing:****
 - Use `open` to create or open a file for writing.
2. ****Write Data to File:****
 - Use `write` to write data to the file.
3. ****Close File:****
 - Use `close` to close the file.
4. ****Open File for Reading:****
 - Use `open` to open the file for reading.
5. ****Read Data from File:****
 - Use `read` to read data from the file.
6. ****Close File Again:****
 - Use `close` to close the file after reading.
7. ****Display Results:****
 - Print the data written to and read from the file.
8. ****Exit:****
 - End the program.

Code:

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

```
int main() {
```

```
// File descriptor
int fd;

// Create a new file or open an existing file for writing
fd = open("example.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);

if (fd == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}

// Write data to the file
const char *data = "Hello, UNIX System Calls!";
ssize_t bytes_written = write(fd, data, strlen(data));

if (bytes_written == -1) {
    perror("write");
    close(fd);
    exit(EXIT_FAILURE);
}

printf("Data written to the file: %s\n", data);

// Close the file
if (close(fd) == -1) {
    perror("close");
    exit(EXIT_FAILURE);
}

// Open the file for reading
fd = open("example.txt", O_RDONLY);
if (fd == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}
```

```

// Read data from the file
char buffer[100];
ssize_t bytes_read = read(fd, buffer, sizeof(buffer) - 1);

if (bytes_read == -1) {
    perror("read");
    close(fd);
    exit(EXIT_FAILURE);
}

buffer[bytes_read] = '\0'; // Null-terminate the string

printf("Data read from the file: %s\n", buffer);

// Close the file
if (close(fd) == -1) {
    perror("close");
    exit(EXIT_FAILURE);
}

return 0;
}

```

Output:

Data written to the file: Hello, UNIX System Calls!

Data read from the file: Hello, UNIX System Calls!

Result: hence the program is compiled and executed successfully.

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

Aim: to construct a c program to implement the I/O system calls of UNIX (fcntl,seek,stat,opendir,readir).

Algorithm:

1. ****File Operations:****

- Open "example.txt" with `open`, set non-blocking flag with `fcntl`, move file pointer with `lseek`, and display file size with `fstat`.

- Close the file with `close`.

2. ****Directory Operations:****

- Open the current directory with `opendir`.
- Read and display file names with `readdir`.
- Close the directory with `closedir`.

3. ****Exit:****

- End the program.

Code:

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

int main() {
    // Demonstrate fcntl
    int fd = open("example.txt", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    // Use fcntl to set the file status flags (here, set it to non-blocking)
    int flags = fcntl(fd, F_GETFL);
    flags |= O_NONBLOCK;
    if (fcntl(fd, F_SETFL, flags) == -1) {
        perror("fcntl");
        close(fd);
        exit(EXIT_FAILURE);
    }

    printf("File 'example.txt' opened with non-blocking flag.\n");
```

```

// Demonstrate lseek
off_t offset = lseek(fd, 5, SEEK_SET);
if (offset == -1) {
    perror("lseek");
    close(fd);
    exit(EXIT_FAILURE);
}

printf("File pointer moved to offset 5.\n");

// Demonstrate stat
struct stat fileStat;
if (fstat(fd, &fileStat) == -1) {
    perror("fstat");
    close(fd);
    exit(EXIT_FAILURE);
}

printf("File Size: %lld bytes\n", (long long)fileStat.st_size);

// Close the file
if (close(fd) == -1) {
    perror("close");
    exit(EXIT_FAILURE);
}

// Demonstrate opendir and readdir
DIR *dir = opendir(".");
if (dir == NULL) {
    perror("opendir");
    exit(EXIT_FAILURE);
}

printf("\nFiles in the current directory:\n");

```

```

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

// Close the directory
closedir(dir);

return 0;
}

```

Output:

File 'example.txt' opened with non-blocking flag.

File pointer moved to offset 5.

File Size: 5 bytes

Files in the current directory:

.

..

example.txt

other_file.txt

subdirectory

result: hence the program is compiled and executed successfully.

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

Aim: to construct a c program to implement the file management operations.

Algorithm:

1. **File Creation and Writing:**

- Open a file named "example.txt" for writing using `fopen`.
- Check if the file is successfully opened; if not, display an error message and exit.
- Write the string "Hello, File Management!" to the file using `fprintf`.
- Close the file using `fclose`.

2. **File Reading:**

- Open the same file for reading using `fopen`.
- Check if the file is successfully opened; if not, display an error message and exit.
- Read content from the file using `fgets` into a buffer.
- Display the content read from the file.

- Close the file using `fclose`.

3. ****Exit:****

- End the program.

Code:

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

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

```
int main() {
```

```
    FILE *file; // File pointer
```

```
    // File creation and writing
```

```
    file = fopen("example.txt", "w");
```

```
    if (file == NULL) {
```

```
        perror("Error creating file");
```

```
        exit(EXIT_FAILURE);
```

```
    }
```

```
    fprintf(file, "Hello, File Management!\n");
```

```
    fclose(file);
```

```
    printf("File 'example.txt' created and written to.\n");
```

```
    // File reading
```

```
    file = fopen("example.txt", "r");
```

```
    if (file == NULL) {
```

```
        perror("Error opening file for reading");
```

```
        exit(EXIT_FAILURE);
```

```
    }
```

```
    char buffer[100];
```

```
    fgets(buffer, sizeof(buffer), file);
```

```
    printf("Content read from file: %s", buffer);
```

```
    fclose(file);
```

```
    return 0;
}
```

Output:

File 'example.txt' created and written to.

Content read from file: Hello, File Management!

Result:

Hence the program is executed successfully.

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

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

int main(int argc, char *argv[]) {
    DIR *dir;
    struct dirent *entry;

    // Open the specified directory or use the current directory if none is provided
    const char *path = (argc > 1) ? argv[1] : ".";

    dir = opendir(path);

    if (dir == NULL) {
        perror("Error opening directory");
        exit(EXIT_FAILURE);
    }

    // List files in the directory
    printf("Files in directory '%s':\n", path);
    while ((entry = readdir(dir)) != NULL) {
        printf("%s\n", entry->d_name);
    }
}
```

```
closedir(dir);
```

```
return 0;
```

```
}
```

Output:

```
./ls_simulation /path/to/directory
```

```
...
```

or if no directory is specified:

```
```bash
```

```
./ls_simulation
```

```
...
```

Example output:

```
...
```

Files in directory '/path/to/directory':

```
file1.txt
```

```
file2.c
```

```
subdirectory
```

```
...
```

**Result:** hence the program is compiled and executed successfully.

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

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

**Code:**

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

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

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

```
#define BUFFER_SIZE 5
```

```
int buffer[BUFFER_SIZE];
```

```
int in = 0, out = 0, count = 0;
```

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t full = PTHREAD_COND_INITIALIZER;
pthread_cond_t empty = PTHREAD_COND_INITIALIZER;
```

```
void *producer(void *arg) {
 int item;
 for (int i = 0; i < 10; i++) {
 item = rand() % 100; // Simulate producing an item
 pthread_mutex_lock(&mutex);

 while (count == BUFFER_SIZE) {
 pthread_cond_wait(&empty, &mutex);
 }

 buffer[in] = item;
 in = (in + 1) % BUFFER_SIZE;
 count++;

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

 pthread_cond_signal(&full);
 pthread_mutex_unlock(&mutex);
 }

 pthread_exit(NULL);
}
```

```
void *consumer(void *arg) {
 int item;
 for (int i = 0; i < 10; i++) {
 pthread_mutex_lock(&mutex);

 while (count == 0) {
 pthread_cond_wait(&full, &mutex);
```

```

 }

 item = buffer[out];
 out = (out + 1) % BUFFER_SIZE;
 count--;

 printf("Consumed: %d\n", item);

 pthread_cond_signal(&empty);
 pthread_mutex_unlock(&mutex);
}

pthread_exit(NULL);
}

int main() {
 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);

 return 0;
}

```

### **Output:**

```

Produced: 24
Produced: 56
Produced: 92
Produced: 11
Produced: 78
Consumed: 24
Produced: 35
Produced: 99

```

Produced: 43  
Produced: 72  
Consumed: 56  
Consumed: 92  
Consumed: 11  
Produced: 64  
Produced: 89  
Produced: 33  
Consumed: 78  
Consumed: 35  
Produced: 54  
Produced: 21  
Produced: 66  
Produced: 48  
Consumed: 99  
Consumed: 43  
Consumed: 72  
Produced: 87  
Consumed: 64  
Consumed: 89  
Consumed: 33  
Produced: 12  
Produced: 67  
Produced: 23  
Produced: 44  
Produced: 79  
Consumed: 54  
Consumed: 21  
Consumed: 66  
Consumed: 48  
Consumed: 87  
Consumed: 12  
Consumed: 67  
Consumed: 23  
Consumed: 44

Consumed: 79

Result: hence the program is compiled and executed successfully.

**30.** Write C programs to demonstrate the following thread related concepts.

**PROGRAM:**

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void* func(void* arg)
{
 pthread_detach(pthread_self());printf("Inside
 the thread\n"); pthread_exit(NULL);
}
void fun()
{
 pthread_t ptid;
 pthread_create(&ptid, NULL, &func, NULL);
 printf("This line may be printed"
 " before thread terminates\n");
 if(pthread_equal(ptid, pthread_self()))
 {
 printf("Threads are equal\n");
 }

 else
```

```

 printf("Threads are not equal\n");
 pthread_join(ptid, NULL);
 printf("This line will be printed""
 after thread ends\n");
 pthread_exit(NULL);
}
int main()
{
 fun();
 return 0;
}

```

```

This line may be printed before thread terminates
Inside the thread
Threads are not equal
This line will be printed after thread ends

```

**OUTPUT:**

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

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

#### **ALGORITHM:**

2. Create an array to represent the page frames in memory.
3. Initialize all page frames to -1, indicating that they are empty.
4. Initialize a queue to keep track of the order in which pages are loaded into memory.
5. Initialize variables for page hits and page faults to zero.
6. Read the reference string (sequence of page numbers) from the user or use a predefined array.
7. For each page in the reference string, do the following:
  8. Check if the page is already in memory (a page hit).
  9. If it's a page hit, update the display and move to the next page.
  10. If it's a page fault (page not in memory), do the following:
    11. Increment the page fault count.
    12. Remove the oldest page in memory (the one at the front of the queue).
    13. Load the new page into the memory and enqueue it.
    14. Update the display to show the page replacement.



15. Continue this process for all pages in the reference string. 16. After processing all pages, display the total number of page faults.

**PROGRAM:**

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

```
#define MAX_FRAMES 3 // Maximum number of frames in memory
```

```
void printFrames(int frames[], int n) { for
```

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

```
 if (frames[i] == -1) {
```

```
 printf(" - ");
```

```
 } else {
```

```
 printf(" %d ", frames[i]);
```

```
 }
```

```
 }
```

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

```
}
```

```
int main() {
```

```
 int referenceString[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2}; int n =
```

```
 sizeof(referenceString) / sizeof(referenceString[0]);
```

```
 int frames[MAX_FRAMES];
```

```
 int framePointer = 0; // Points to the current frame to be replaced
```

```
 for (int i = 0; i < MAX_FRAMES; i++) {
```

```
 frames[i] = -1; // Initialize all frames to -1 (indicating empty)
```

```
 }
```

```
printf("Reference String: ");for
(int i = 0; i < n; i++) {
 printf("%d ", referenceString[i]);
}
printf("\n\n");
```

```
printf("Page Replacement Order:\n");
```

```
for (int i = 0; i < n; i++) {
 int page = referenceString[i];int
 pageFound = 0;

 // Check if the page is already in memoryfor (int j
 = 0; j < MAX_FRAMES; j++) {
 if (frames[j] == page) {
 pageFound = 1; break;
 }
 }
}
```

```
if (!pageFound) {
 printf("Page %d -> ", page);
 frames[framePointer] = page;
 framePointer = (framePointer + 1) % MAX_FRAMES;
 printFrames(frames, MAX_FRAMES);
}
```

```

 }

 return 0;
}

```

### OUTPUT:

```

Reference String: 7 0 1 2 0 3 0 4 2 3 0 3 2

Page Replacement Order:
Page 7 -> 7 - -
Page 0 -> 7 0 -
Page 1 -> 7 0 1
Page 2 -> 2 0 1
Page 3 -> 2 3 1
Page 0 -> 2 3 0
Page 4 -> 4 3 0
Page 2 -> 4 2 0
Page 3 -> 4 2 3
Page 0 -> 0 2 3

Process exited after 0.0623 seconds with return value 0
Press any key to continue . . . |

```

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

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

#### ALGORITHM:

1. Create an array to represent the page frames in memory.
2. Initialize all page frames to -1, indicating that they are empty.
3. Create a queue or a data structure (e.g., a doubly-linked list) to maintain the order of pages based on their usage history.
4. Initialize a counter for page hits and page faults to zero.
5. Read the reference string (sequence of page numbers) from the user or use a predefined array.

6. For each page in the reference string, do the following:
7. Check if the page is already in memory (a page hit).
8. If it's a page hit, update the position of the page in the usage history data structure to indicate it was recently used.
9. If it's a page fault (page not in memory), do the following:
10. Increment the page fault count.
11. Find the least recently used page in the usage history data structure (e.g., the front of the queue or the tail of the list).
12. Remove the least recently used page from memory and the usage history.
13. Load the new page into memory and add it to the back of the usage history.
14. Update the display to show the page replacement.
15. Continue this process for all pages in the reference string.
16. After processing all pages, display the total number of page faults.

**PROGRAM:**

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

```
#define MAX_FRAMES 3
```

```
void printFrames(int frames[], int n) {
 for (int i = 0; i < n; i++) {
 if (frames[i] == -1) {
 printf(" - ");
 } else {
 printf(" %d ", frames[i]);
 }
 }
 printf("\n");
}
```

```
int main() {
 int frames[MAX_FRAMES];
 int usageHistory[MAX_FRAMES]; // To store the usage history of pages
 for (int i = 0; i < MAX_FRAMES; i++) {
```

```

frames[i] = -1; // Initialize frames to -1 (empty)
usageHistory[i] = 0; // Initialize usage history
}

```

```

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

```

```

printf("Reference String: ");for
(int i = 0; i < n; i++) {
 printf("%d ", referenceString[i]);
}
printf("\n\n");

```

```

printf("Page Replacement Order:\n");for

```

```

(int i = 0; i < n; i++) {
 int page = referenceString[i];int
 pageFound = 0;

 // Check if the page is already in memory (a page hit)for (int j
 = 0; j < MAX_FRAMES; j++) {
 if (frames[j] == page) {
 pageFound = 1;
 // Update the usage history by incrementing other pagesfor (int k
 = 0; k < MAX_FRAMES; k++) {
 if (k != j) {
 usageHistory[k]++;
 }
 }
 usageHistory[j] = 0; // Reset the usage counter for the used pagebreak;
 }
 }
}

```

```

if (!pageFound) {

```

```

printf("Page %d -> ", page);

// Find the page with the maximum usage counter (least recently
used)
int lruPage = 0;
for (int j = 1; j < MAX_FRAMES; j++) {
 if (usageHistory[j] > usageHistory[lruPage]) {lruPage
 = j;
 }
}

int replacedPage = frames[lruPage];
frames[lruPage] = page;
usageHistory[lruPage] = 0;

if (replacedPage != -1) {
 printf("Replace %d with %d: ", replacedPage, page);
} else {
 printf("Load into an empty frame: ");
}

printFrames(frames, MAX_FRAMES);
pageFaults++;
}
}

printf("\nTotal Page Faults: %d\n", pageFaults);

return 0;
}

```

**OUTPUT:**

```

Reference String: 7 0 1 2 0 3 0 4 2 3 0 3 2

Page Replacement Order:
Page 7 -> Load into an empty frame: 7 - -
Page 0 -> Replace 7 with 0: 0 - -
Page 1 -> Replace 0 with 1: 1 - -
Page 2 -> Replace 1 with 2: 2 - -
Page 0 -> Replace 2 with 0: 0 - -
Page 3 -> Replace 0 with 3: 3 - -
Page 0 -> Replace 3 with 0: 0 - -
Page 4 -> Replace 0 with 4: 4 - -
Page 2 -> Replace 4 with 2: 2 - -
Page 3 -> Replace 2 with 3: 3 - -
Page 0 -> Replace 3 with 0: 0 - -
Page 3 -> Replace 0 with 3: 3 - -
Page 2 -> Replace 3 with 2: 2 - -

Total Page Faults: 13

Process exited after 0.05045 seconds with return value 0
Press any key to continue . . . |

```

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

**AIM:** Construct a C program to simulate the optimal paging technique of memory management

**ALGORITHM:**

1. Create an array to represent the page frames in memory.
2. Initialize all page frames to -1, indicating that they are empty.
3. Initialize a variable for page faults to zero.
4. Read the reference string (sequence of page numbers) from the user or use a predefined array.
5. For each page in the reference string, do the following:
6. Check if the page is already in memory (a page hit).
7. If it's a page hit, move to the next page.
8. If it's a page fault (page not in memory), do the following:
9. Increment the page fault count.

10. Calculate the future references of each page in memory by scanning the remaining part of the reference string.
11. Find the page that will not be used for the longest time in the future (the optimal page to replace).
12. Replace the optimal page with the new page.
13. Continue this process for all pages in the reference string.
14. After processing all pages, display the total number of page faults.

**PROGRAM:**

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

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

```
#define MAX_FRAMES 3
```

```
void printFrames(int frames[], int n) {for
 (int i = 0; i < n; i++) {
 if (frames[i] == -1) {
 printf(" - ");
 } else {
 printf(" %d ", frames[i]);
 }
 }
 printf("\n");
}
```

```
int main() {
 int frames[MAX_FRAMES];
 for (int i = 0; i < MAX_FRAMES; i++) { frames[i] = -1;
 // Initialize frames to -1 (empty)
 }

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

 printf("Reference String: ");for
 (int i = 0; i < n; i++) {
```



```

 printf("%d ", referenceString[i]);
}
printf("\n\n");

printf("Page Replacement Order:\n");for

(int i = 0; i < n; i++) {
 int page = referenceString[i];
 int pageFound = 0;

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

 if (!pageFound) {
 printf("Page %d -> ", page);

 int optimalPage = -1; int
 farthestDistance = 0;

 for (int j = 0; j < MAX_FRAMES; j++) {int
 futureDistance = 0;
 for (int k = i + 1; k < n; k++) {
 if (referenceString[k] == frames[j]) {break;
 }
 futureDistance++;
 }

 if (futureDistance > farthestDistance) {
 farthestDistance = futureDistance;
 optimalPage = j;
 }
 }
 }
}

```

```

 }
}

frames[optimalPage] = page;

printFrames(frames, MAX_FRAMES);
pageFaults++;
}
}

printf("\nTotal Page Faults: %d\n", pageFaults);

return 0;
}

```

```

Reference String: 7 0 1 2 0 3 0 4 2 3 0 3 2

Page Replacement Order:
Page 7 -> 7 - -
Page 0 -> 0 - -
Page 1 -> 0 1 -
Page 2 -> 0 2 -
Page 3 -> 0 2 3
Page 4 -> 4 2 3
Page 0 -> 0 2 3

Total Page Faults: 7

Process exited after 0.05286 seconds with return value 0
Press any key to continue . . . |

```

## OUTPUT

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

**AIM:** Consider a file system where the records of the file are stored one after another both physically and logically. A record of the file can only be accessed

by reading all the previous records. Design a C program to simulate the file allocation strategy.

### **ALGORITHM:**

1. Define the structure of a record that will be stored in the file.
2. Create a file to represent the sequential file.
3. Write records to the file sequentially, one after the other.
4. To read a specific record:
5. Prompt the user for the record number they want to access.
6. Read and display all records from the beginning of the file up to the requested record.
7. Continue this process until the user decides to exit.

### **PROGRAM:**

```
#include <stdio.h>

#include <stdlib.h>

// Structure to represent a record struct
Record {
 int recordNumber;
 char data[256]; // Adjust the size as needed for your records
};

int main() {
 FILE *file;
 struct Record record; int
 recordNumber;

 // Open or create a file in write mode (for writing records) file =
 fopen("sequential_file.txt", "w");
 if (file == NULL) {
 printf("Error opening the file.\n");
```

```

 return 1;
 }

 // Write records sequentially to the file
 printf("Enter records (Enter '0' as record number to exit):\n");while (1)
 {
 printf("Record Number: "); scanf("%d",
 &record.recordNumber);if
 (record.recordNumber == 0) {
 break;
 }

 // Input data for the record
 printf("Data: ");
 scanf(" %[^\n]", record.data);

 // Write the record to the file
 fwrite(&record, sizeof(struct Record), 1, file);
 }

 fclose(file);

 // Reopen the file in read mode (for reading records)file =
 fopen("sequential_file.txt", "r");
 if (file == NULL) {
 printf("Error opening the file.\n");
 return 1;
 }

```

```
}
```

```
// Read a specific record from the filewhile
```

```
(1) {
```

```
 printf("Enter the record number to read (0 to exit): ");
```

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

```
 if (recordNumber == 0) {
```

```
 break;
```

```
 }
```

```
// Read and display records up to the requested recordwhile
```

```
(fread(&record, sizeof(struct Record), 1, file)) {
```

```
 printf("Record Number: %d\n", record.recordNumber);
```

```
 printf("Data: %s\n", record.data);
```

```
 if (record.recordNumber == recordNumber) {break;
```

```
 }
```

```
}
```

```
rewind(file); // Reset the file pointer to the beginning of the file
```

```
}
```

```
fclose(file);
```

```
return 0;
```

```
}
```

**OUTPUT:**

```

Enter records (Enter '0' as record number to exit):
Record Number: 389
Data: JASWANTH
Record Number: 0
Enter the record number to read (0 to exit): 389
Record Number: 389
Data: JASWANTH
Enter the record number to read (0 to exit): 0

Process exited after 29.88 seconds with return value 0
Press any key to continue . . . |

```

**35. Consider a file system that brings all the file pointers together into an index block. The *i*th entry in the index block points to the *i*th block of the file. Design a C program to simulate the file allocation strategy.**

**AIM:** Consider a file system that brings all the file pointers together into an index block. The *i*th entry in the index block points to the *i*th block of the file. Design a C program to simulate the file allocation strategy.

**ALGORITHM:**

1. Define the structure of a block that will be stored in the file.
2. Create a file to represent the indexed file.
3. Initialize an index block that contains pointers to data blocks.
4. To write a new block:
5. Prompt the user for the block number and the data to be written to the block.
6. Update the corresponding entry in the index block to point to the new data block.
7. Write the data block to the file.
8. To read a specific block:
9. Prompt the user for the block number they want to access.
10. Use the index block to find the pointer to the requested data block.
11. Read and display the data in the requested data block.
12. Continue this process until the user decides to exit.

**PROGRAM:**

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

```

#include <stdlib.h>

// Structure to represent a block
struct Block {
 int blockNumber;
 char data[256]; // Adjust the size as needed for your blocks
};

int main() {
 FILE *file;
 struct Block block; int
 blockNumber;

 // Create an index block that contains pointers to data blocks int
 indexBlock[100] = {0}; // Adjust the size as needed

 // Open or create a file in write mode (for writing blocks) file =
 fopen("indexed_file.txt", "w");
 if (file == NULL) {
 printf("Error opening the file.\n");
 return 1;
 }

 // Write blocks and update the index block
 printf("Enter blocks (Enter '0' as block number to exit):\n"); while (1)
 {
 printf("Block Number: ");

```

```

scanf("%d", &block.blockNumber);if
(block.blockNumber == 0) {
 break;
}

// Input data for the block
printf("Data: ");
scanf(" %[^\\n]", block.data);

// Write the block to the file
fwrite(&block, sizeof(struct Block), 1, file);

// Update the index block with the pointer to the data block indexBlock[block.blockNumber] =
ftell(file) - sizeof(struct Block);
}

fclose(file);

// Reopen the file in read mode (for reading blocks)file =
fopen("indexed_file.txt", "r");
if (file == NULL) {
 printf("Error opening the file.\\n");
 return 1;
}

// Read a specific block from the filewhile
(1) {

```



```
printf("Enter the block number to read (0 to exit): ");
scanf("%d", &blockNumber);
if (blockNumber == 0) {
 break;
}

if (indexBlock[blockNumber] == 0) {
 printf("Block not found.\n");
} else {
 // Seek to the data block using the index block fseek(file,
 indexBlock[blockNumber], SEEK_SET);fread(&block,
 sizeof(struct Block), 1, file);

 printf("Block Number: %d\n", block.blockNumber);
 printf("Data: %s\n", block.data);
}
}

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

**OUTPUT:**

```

Enter blocks (Enter '0' as block number to exit):
Block Number: 39
Data: JSAWNTH
Block Number: 43
Data: SAI
Block Number: 12
Data: FRIEND
Block Number: 0
Enter the block number to read (0 to exit): 12
Block Number: 12
Data: FRIEND
Enter the block number to read (0 to exit): 0

Process exited after 34.15 seconds with return value 0
Press any key to continue . . . |

```

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

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

**ALGORITHM:**

1. Define the structure of a block that will be stored in the file. Each block contains a pointer to the next block.
2. Create a file to represent the linked allocation.
3. Create a directory entry for the file containing a pointer to the first and last blocks.
4. To write a new block:
5. Prompt the user for the block data.
6. Allocate a new block in the file.
7. If it's the first block, update the directory entry to point to it as both the first and last block.

8. If it's not the first block, update the previous block to point to the newblock.
9. Update the new block's pointer to the next block (usually NULL for the lastblock).
10. To read a specific block:
  11. Prompt the user for the block number they want to access.
  12. Use the directory entry to find the first block of the file.
  13. Traverse the linked list of blocks until you reach the desired block.
  14. Read and display the data in the requested block.
  15. Continue this process until the user decides to exit.

**PROGRAM:**

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

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

```
// Structure to represent a block
```

```
struct Block {
```

```
 char data[256]; // Adjust the size as needed for your blocks
```

```
 Block* next;
```

```
};
```

```
int main() {
```

```
 struct Block* firstBlock = NULL; // Pointer to the first block in the linkedlist
```

```
 struct Block* lastBlock = NULL; // Pointer to the last block in the linkedlist
```

```
 int blockCount = 0; // Count of blocks in the linked list
```

```
 int blockNumber;
```

```
 char data[256];
```

```
 char choice;
```

```
printf("Linked Allocation Simulation\n");
```

```
while (1) {
```

```
 printf("Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit: ");scanf("%c", &choice);
```

```
 if (choice == 'Q' || choice == 'q') {
 break;
 }
```

```
 if (choice == 'W' || choice == 'w') {
 printf("Enter data for the block: ");
 scanf(" %[^\n]", data);
```

```
 // Create a new block
```

```
 struct Block* newBlock = (struct Block*)malloc(sizeof(struct Block));for (int i =
0; i < 256; i++) {
 newBlock->data[i] = data[i];
 }
```

```
 newBlock->next = NULL;
```

```
 if (blockCount == 0) {
 // This is the first block
 firstBlock = newBlock;
 lastBlock = newBlock;
 } else {
```

```

 // Link the new block to the last block
 lastBlock->next = newBlock; lastBlock =
 newBlock;
 }

 blockCount++;
} else if (choice == 'R' || choice == 'r') {
 printf("Enter the block number to read (1-%d): ", blockCount);
 scanf("%d", &blockNumber);

 if (blockNumber < 1 || blockNumber > blockCount) { printf("Invalid
 block number. The valid range is 1-%d.\n",
blockCount);
 } else {
 struct Block* currentBlock = firstBlock;for
 (int i = 1; i < blockNumber; i++) {
 currentBlock = currentBlock->next;
 }

 printf("Block %d Data: %s\n", blockNumber, currentBlock->data);
 }
}

}

// Free the allocated memory for blocks before exitingstruct
Block* currentBlock = firstBlock;
while (currentBlock != NULL) {
 struct Block* nextBlock = currentBlock->next;

```

```

 free(currentBlock);
 currentBlock = nextBlock;
 }

 return 0;
}

```

## OUTPUT:

```

Linked Allocation Simulation
Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit: W
Enter data for the block: SAI IS WORST
Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit: W
Enter data for the block: JASWANTH IS GOOD
Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit: R
Enter the block number to read (1-2): 2
Block 2 Data: JASWANTH IS GOOD
Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit: R
Enter the block number to read (1-2): 1
Block 1 Data: SAI IS WORST
Enter 'W' to write a block, 'R' to read a block, or 'Q' to quit: Q

Process exited after 46.7 seconds with return value 0
Press any key to continue . . . |

```

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

**AIM:-** Construct a C program to simulate the First Come First Served disk scheduling algorithm.

#### **ALGORITHM:-**

1. Start at the current position of the disk head.
2. For each disk request in the queue:
  - Move the disk head to the requested track.
  - Calculate the seek time as the absolute difference between the new position of the disk head and the previous position.

- Add the seek time to the total seek time.
  - Update the previous position of the disk head to the current position.
3. Repeat step 2 for all disk requests in the queue.
  4. After serving all the requests, calculate and display the total seek time.
  5. Calculate and display the average seek time, which is the total seek time divided by the number of requests.

**PROGRAM:-**

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

int main() {
 int n, head, seek_time = 0;

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

 int request_queue[n];

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

 printf("Enter the initial position of the disk head: ");
 scanf("%d", &head);

 // FCFS Scheduling
 printf("\nFCFS Disk Scheduling:\n");
```

```

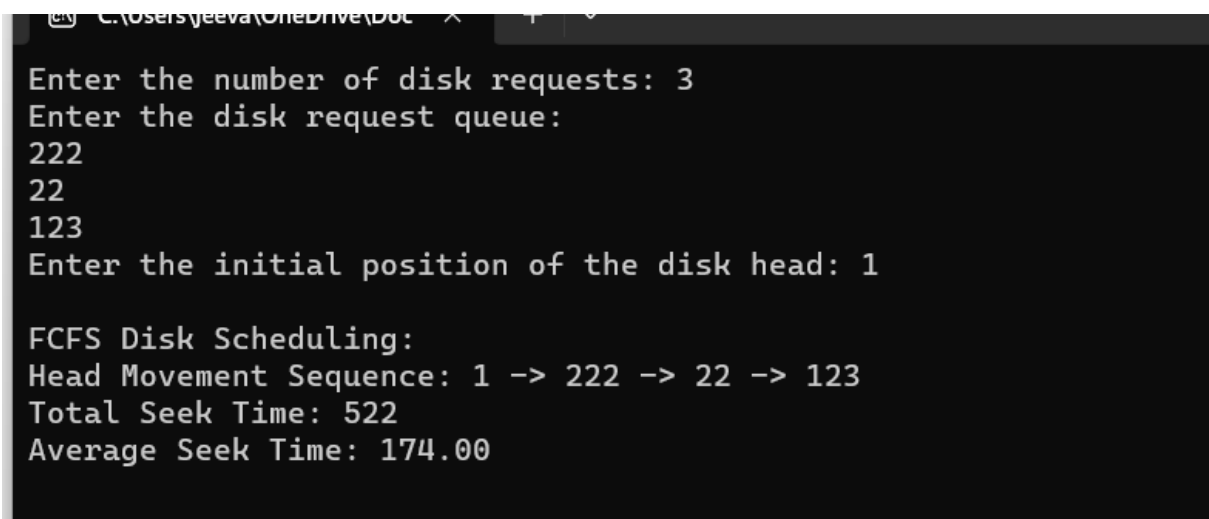
printf("Head Movement Sequence: %d", head);for (int
i = 0; i < n; i++) {
 seek_time += abs(head - request_queue[i]);head =
 request_queue[i];
 printf(" -> %d", head);
}

printf("\nTotal Seek Time: %d\n", seek_time); printf("Average Seek
Time: %.2f\n", (float) seek_time / n);

return 0;
}

```

**OUTPUT:-**



```

C:\Users\jeeva\OneDrive\Doc...
Enter the number of disk requests: 3
Enter the disk request queue:
222
22
123
Enter the initial position of the disk head: 1

FCFS Disk Scheduling:
Head Movement Sequence: 1 -> 222 -> 22 -> 123
Total Seek Time: 522
Average Seek Time: 174.00

```

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

**AIM:-** Design a C program to simulate SCAN disk scheduling algorithm.

**ALGORITHM:-**



1. Determine the direction of movement (inward or outward) based on the queue of pending requests and the current position.
2. While servicing requests in the selected direction:
  - Move the disk head to the next track in the current direction.
  - Calculate the seek time as the absolute difference between the new position of the disk head and the previous position.
  - Add the seek time to the total seek time.
  - Update the previous position of the disk head to the current position.
3. If there are no more requests in the current direction, change direction to the opposite direction.
4. Repeat step 3 until all requests are serviced.
5. After serving all the requests, calculate and display the total seek time.
6. Calculate and display the average seek time, which is the total seek time divided by the number of requests.

**PROGRAM:**

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

int main() {
 int n, head, seek_time = 0;

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

 int request_queue[n];

 printf("Enter the disk request queue:\n");
```

```

for (int i = 0; i < n; i++) { scanf("%d",
 &request_queue[i]);
}

printf("Enter the initial position of the disk head: ");
scanf("%d", &head);

// Sort the request queue to simplify SCAN algorithm
for (int i = 0; i < n - 1; i++) {
 for (int j = i + 1; j < n; j++) {
 if (request_queue[i] > request_queue[j]) {
 int temp = request_queue[i]; request_queue[i] =
 request_queue[j]; request_queue[j] = temp;
 }
 }
}

// SCAN (Elevator) Scheduling
printf("\nSCAN (Elevator) Disk Scheduling:\n");
int start = 0;
int end = n - 1;
int current_direction = 1; // 1 for moving right, -1 for moving left

while (start <= end) {
 if (current_direction == 1) {
 for (int i = start; i <= end; i++) {

```

```

 if (request_queue[i] >= head) {
 seek_time += abs(head - request_queue[i]); head =
 request_queue[i];
 start = i + 1;
 break;
 }
 }
 current_direction = -1; // Change direction
} else {
 for (int i = end; i >= start; i--) {
 if (request_queue[i] <= head) {
 seek_time += abs(head - request_queue[i]); head =
 request_queue[i];
 end = i - 1;
 break;
 }
 }
 current_direction = 1; // Change direction
}
}

printf("Total Seek Time: %d\n", seek_time); printf("Average Seek
Time: %.2f\n", (float)seek_time / n);

return 0;
}

```

### Output:-

```
Enter the number of disk requests: 3
Enter the disk request queue:
12
34
45
Enter the initial position of the disk head: 45

SCAN (Elevator) Disk Scheduling:
Total Seek Time: 0
Average Seek Time: 0.00
```

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

**AIM:-** Develop a C program to simulate C-SCAN disk scheduling algorithm.

#### **ALGORITHM:-**

1. Start at the current position of the disk head.
2. Set the direction of movement to one side (e.g., right).
3. While servicing requests in the selected direction:
  - Move the disk head to the next track in the current direction.
  - Calculate the seek time as the absolute difference between the new position of the disk head and the previous position.
  - Add the seek time to the total seek time.
  - Update the previous position of the disk head to the current position.
4. If there are no more requests in the current direction:
  - Move the disk head to the end of the disk in the current direction.
  - Change direction to the opposite side (e.g., left).
  - Continue servicing requests in the new direction.
5. Repeat step 3 and step 4 until all requests are serviced.
6. After serving all the requests, calculate and display the total seek time.
7. Calculate and display the average seek time, which is the total seek time divided by the number of requests.

#### **PROGRAM:-**

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

```

#include <stdlib.h>

int main() {
 int n, head, seek_time = 0;

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

 int request_queue[n];

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

 printf("Enter the initial position of the disk head: ");
 scanf("%d", &head);

 // Sort the request queue for simplicityfor
(int i = 0; i < n - 1; i++) {
 for (int j = i + 1; j < n; j++) {
 if (request_queue[i] > request_queue[j]) {int
 temp = request_queue[i]; request_queue[i] =
 request_queue[j]; request_queue[j] = temp;
 }
 }
 }

 // C-SCAN Scheduling
 printf("\nC-SCAN Disk Scheduling:\n");int
start = 0;
int end = n - 1;

 while (start <= end) {
 for (int i = start; i <= end; i++) { if
 (request_queue[i] >= head) {
 seek_time += abs(head - request_queue[i]);head =
 request_queue[i];
 start = i + 1;
 }
 }
 }
}

```

```

 }
}
// Move the head to the end in the current direction
seek_time += abs(head - 0);
head = 0;
// Change direction to the opposite side seek_time +=
abs(head - request_queue[end]); head =
request_queue[end];
end = n - 2; // Exclude the last request, as it has already been served

}

printf("Total Seek Time: %d\n", seek_time); printf("Average Seek
Time: %.2f\n", (float)seek_time / n);

return 0;
}

```

#### OUTPUT:-

```

Enter the number of disk requests: 3
Enter the disk request queue:
12
13
14
Enter the initial position of the disk head: 5

C-SCAN Disk Scheduling:
Total Seek Time: 37
Average Seek Time: 12.33

```

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

**AIM:** Illustrate the various File Access Permission and different types users in Linux.

#### ALGORITHM:

1. Create a file or identify an existing file to demonstrate permissions and users.

2. View the file's permissions using the `ls -l` command. The output will look something like this:

.txt

- The first character (-) represents the file type (a dash indicates a regular file).
- The next three characters (rw-) represent the permissions for the file's owner (Read and Write, no Execute).
- The next three characters (r--) represent the permissions for the file's group (Read, no Write or Execute).
- The last three characters (r--) represent the permissions for others (Read, no Write or Execute).
- The number 1 represents the number of hard links to the file.
- owner is the username of the file's owner.
- group is the name of the file's group.
- 1234 is the file's size in bytes.
- Oct 19 10:30 is the last modification timestamp.
- file.txt is the file name.

3. Use the `chmod` command to change the file's permissions. For example, to give the group write permission, use `chmod g+w file.txt`.

4. Re-run `ls -l` to confirm the updated permissions.

5. You can also change the file's owner and group using the `chown` and `chgrp` commands, respectively.

6. To create and manage user accounts, you can use the `useradd` and `passwd` commands.

### **PROGRAM:**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
```

```
int main() {
 char filename[] = "file.txt";
 int new_permissions = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH; // rw-rw-r--

 if (chmod(filename, new_permissions) == 0) { printf("File
 permissions changed successfully.\n");
 } else {
 perror("chmod");
 return 1;
 }
}
```


```
 }

 return 0;
}
```

## OUTPUT:

1. Compile the C program (assuming it's saved in a file named `change_permissions.c`):


bash

 Copy code

```
gcc -o change_permissions change_permissions.c
```

1. Run the program:

bash


 Copy code

```
./change_permissions
```

## Output:

If the program executes successfully, it should display the following output:

arduino

 Copy code

```
File permissions changed successfully.
```