1. Develop a c program to implement the Process system calls (fork (), exec(), wait(), create process, terminate process)

fork ()

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
main(void) {
    pid_t pid = 0;
    pid = fork();
    if (pid == 0) {
        printf("I am the child.\n");
    }
    if (pid > 0) {
        printf("I am the parent, the child is %d.\n", pid);
    }
    if (pid < 0) {
        perror("In fork():");
    }
    exit(0);
}
```

Output

I am the parent, the child is 1389.

I am the child.

exec()

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
main(void) {
```

```
    pid_t pid = 0;
    int   status;
    pid = fork();
    if (pid == 0) {
        printf("I am the child.\n");
        execl("/bin/ls", "ls", "-l", "/tmp/igDp5xYjSj.o", (char *) 0);
        perror("In exec(): ");
    }
    if (pid > 0) {
        printf("I am the parent, and the child is %d.\n", pid);
        pid = wait(&status);
        printf("End of process %d: ", pid);
        if (WIFEXITED(status)) {
            printf("The process ended with exit(%d).\n", WEXITSTATUS(status));
        }
            }
    if (pid < 0) {
        perror("In fork():");
    }
    exit(0);
}
```

<u>Output</u>

I am the child.

I am the parent, and the child is 1916.

-rwxr-xr-x 1 compiler compiler 16680 Sep 21 06:13 /tmp/igDp5xYjSj.o

End of process 1916: The process ended with exit(0).

create process, terminate process

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

main(void) {
```

```
    pid_t pid = 0;
    pid = fork();
    if (pid == 0) {
        printf("I am the child.\n");
    }
    if (pid > 0) {
        printf("I am the parent, the child is %d.\n", pid);
    }
    if (pid < 0) {
        perror("In fork():");
    }
    exit(0);
}
```

Output:

I am the parent, the child is 3637.

I am the child.

----

2. Simulate the following CPU scheduling algorithms to find turnaround time and waiting time a) FCFS b) SJF c) Round Robin d) Priority.

a)FCFS

```
#include<stdio.h>
void findWaitingTime(int processes[], int n, int bt[], int wt[])
{
        // waiting time for first process is 0
        wt[0] = 0;
        // calculating waiting time
        for (int i = 1; i < n ; i++ )
                wt[i] = bt[i-1] + wt[i-1] ;
}
// Function to calculate turn around time
```

```
void findTurnAroundTime( int processes[], int n, int bt[], int wt[], int tat[])
{
	// calculating turnaround time by adding  bt[i] + wt[i]
	for (int i = 0; i < n ; i++)
		tat[i] = bt[i] + wt[i];
}
//Function to calculate average time
void findavgTime( int processes[], int n, int bt[])
{
	int wt[n], tat[n], total_wt = 0, total_tat = 0;
	//Function to find waiting time of all processes
	findWaitingTime(processes, n, bt, wt);
	//Function to find turn around time for all processes
	findTurnAroundTime(processes, n, bt, wt, tat);
	//Display processes along with all details
	printf("Processes Burst time Waiting time Turn around time\n");
	// Calculate total waiting time and total turn
	// around time
	for (int i=0; i<n; i++)
	{
		total_wt = total_wt + wt[i];
		total_tat = total_tat + tat[i];
		printf(" %d ",(i+1));
		printf(" %d ", bt[i] );
		printf(" %d",wt[i] );
		printf(" %d\n",tat[i] );
	}
	float s=(float)total_wt / (float)n;
	float t=(float)total_tat / (float)n;
	printf("Average waiting time = %f",s);
	printf("\n");
	printf("Average turn around time = %f ",t);
```

```
}
int main()
{
    //process id's
    int processes[] = { 1, 2, 3};
    int n = sizeof processes / sizeof processes[0];
    //Burst time of all processes
    int burst_time[] = {10, 5, 8};
    findavgTime(processes, n, burst_time);
    return 0;
}
```

Output

```
Processes Burst time Waiting time Turn around time
1    10   0    10
2    5    10   15
 3   8    15   23
Average waiting time = 8.333333
Average turn around time = 16.000000
```

b)SJF

```
#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");
    // User Input Burst Time and alloting Process Id.
    for (i = 0; i < n; i++) {
        printf("P%d: ", i + 1);
```

```
        scanf("%d", &A[i][1]);
        A[i][0] = i + 1;
}
// Sorting process according to their Burst Time.
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;
// Calculation of Waiting Times
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");
// Calculation of Turn Around Time and printing the data.
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: 2

Enter Burst Time:

P1: 10

P2: 12

P	BT	WT	TAT
P1	10	0	10
P2	12	10	22

Average Waiting Time= 5.000000

Average Turnaround Time= 16.000000

c) Round Robin

```
#include<stdio.h>
#include<limits.h>
#include<stdbool.h>
struct P{
int AT,BT,ST[20],WT,FT,TAT,pos;
};
int quant;
int main(){
int n,i,j;
printf("Enter the no. of processes :");
scanf("%d",&n);
struct P p[n];
printf("Enter the quantum \n");
scanf("%d",&quant);
```

```
printf("Enter the process numbers \n");
for(i=0;i<n;i++)
scanf("%d",&(p[i].pos));
printf("Enter the Arrival time of processes \n");
for(i=0;i<n;i++)
scanf("%d",&(p[i].AT));
printf("Enter the Burst time of processes \n");
for(i=0;i<n;i++)
scanf("%d",&(p[i].BT));
int c=n,s[n][20];
float time=0,mini=INT_MAX,b[n],a[n];
// Initializing burst and arrival time arrays
int index=-1;
for(i=0;i<n;i++){
        b[i]=p[i].BT;
        a[i]=p[i].AT;
        for(j=0;j<20;j++){
        s[i][j]=-1;
        }
}
int tot_wt,tot_tat;
tot_wt=0;
tot_tat=0;
bool flag=false;
while(c!=0){
mini=INT_MAX;
flag=false;
for(i=0;i<n;i++){
        float p=time+0.1;
        if(a[i]<=p && mini>a[i] && b[i]>0){
        index=i;
        mini=a[i];
```

```
                    flag=true;
                    }
}
// if at =1 then loop gets out hence set flag to false
if(!flag){
                    time++;
                    continue;
}
//calculating start time
j=0;
while(s[index][j]!=-1){
j++;
}
if(s[index][j]==-1){
s[index][j]=time;
p[index].ST[j]=time;
}
if(b[index]<=quant){
time+=b[index];
b[index]=0;
}
else{
time+=quant;
b[index]-=quant;
}
if(b[index]>0){
a[index]=time+0.1;
}
// calculating arrival,burst,final times
if(b[index]==0){
c--;
p[index].FT=time;
```

```
p[index].WT=p[index].FT-p[index].AT-p[index].BT;
tot_wt+=p[index].WT;
p[index].TAT=p[index].BT+p[index].WT;
tot_tat+=p[index].TAT;
}
}
printf("Process number ");
printf("Arrival time ");
printf("Burst time ");
printf("\tStart time");
j=0;
while(j!=10){
j+=1;
printf(" ");
}
printf("\t\tFinal time");
printf("\tWait Time ");
printf("\tTurnAround Time \n");
for(i=0;i<n;i++){
printf("%d \t\t",p[i].pos);
printf("%d \t\t",p[i].AT);
printf("%d \t",p[i].BT);
j=0;
int v=0;
while(s[i][j]!=-1){
printf("%d ",p[i].ST[j]);
j++;
v+=3;
}
while(v!=40){
printf(" ");
v+=1;
```

```
}
printf("%d \t\t",p[i].FT);
printf("%d \t\t",p[i].WT);
printf("%d \n",p[i].TAT);
}
double avg_wt,avg_tat;
avg_wt=tot_wt/(float)n;
avg_tat=tot_tat/(float)n;
printf("The average wait time is : %lf\n",avg_wt);
printf("The average TurnAround time is : %lf\n",avg_tat);
return 0;
}
```

Output:

```
Enter the no. of processes :3
Enter the quantum
5
Enter the process numbers
2
3
4
Enter the Arrival time of processes
10
12
15
Enter the Burst time of processes
5
12
10
Process number Arrival time Burst time  Start time              Final time  Wait
    Time     TurnAround Time
2       10       5    10                                  15        0        5
3       12       12   15  25  35                          37        13       25
4       15       10   20  30                              35        10       20
The average wait time is : 7.666667
The average TurnAround time is : 16.666666
```

d) Priority

3.Develop a C program to simulate producer-consumer problem using semaphores.

```
#include <stdio.h>
#include <stdlib.h>
int mutex = 1,full = 0, empty = 10, x = 0;
void producer()
{
	--mutex;
	++full;
	--empty;
	x++;
	printf("\nProducer produces" "item %d",x);
	++mutex;
}
void consumer()
{
	--mutex;
	--full;
	++empty;
	printf("\nConsumer consumes " "item %d",x);
	x--;
	++mutex;
}
int main()
{
	int n, i;
	printf("\n1. Press 1 for Producer" "\n2. Press 2 for Consumer"
		"\n3. Press 3 for Exit");
#pragma omp critical
	for (i = 1; i > 0; i++) {
		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;
		}
```

```
        }
}
```

Output

```
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
Enter your choice:1
Producer producesitem 1
Enter your choice:2
Consumer consumes item 1
Enter your choice:3
```

---

4. Develop a C program which demonstrates interprocess communication between a reader process and a writer process. Use mkfifo, open, read, write and close APIs in your program.

```
#include<unistd.h>
#include<stdio.h>
#include<sys/types.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<string.h>
#include<errno.h>
int main(int argc, char * argv[])
{
if(argc!=2 && argc!=3)
{
printf("Usage: %s <file> [<arg>]\n",argv[0]);
return 0;
}
int fd;
char buf[256];
(void)mkfifo(argv[1],S_IFIFO | S_IRWXU | S_IRWXG | S_IRWXO);      /*To create FIFO*/
if(argc==2)
```

```
{
fd=open(argv[1],O_RDWR | O_NONBLOCK);
while(read(fd,buf,sizeof(buf))<0)
sleep(1);
printf("%s",buf);
}
else
{
fd=open(argv[1],O_RDWR);
write(fd,argv[2],strlen(argv[2])+1);
}
close(fd);
}
```

Output:

--

5. Develop a C program to simulate Bankers Algorithm for DeadLock Avoidance.

```
// Banker's Algorithm
#include <stdio.h>
int main()
{
	// P0, P1, P2, P3, P4 are the Process names here
	int n, m, i, j, k;
	n = 5; // Number of processes
	m = 3; // Number of resources
	int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix
						{ 2, 0, 0 }, // P1
						{ 3, 0, 2 }, // P2
						{ 2, 1, 1 }, // P3
						{ 0, 0, 2 } }; // P4

	int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix
					{ 3, 2, 2 }, // P1
					{ 9, 0, 2 }, // P2
					{ 2, 2, 2 }, // P3
					{ 4, 3, 3 } }; // P4

	int avail[3] = { 3, 3, 2 }; // Available Resources 0
	int f[n], ans[n], ind = 0;
	for (k = 0; k < n; k++) {
```

```
            f[k] = 0;
}
int need[n][m];
for (i = 0; i < n; i++) {
            for (j = 0; j < m; j++)
                        need[i][j] = max[i][j] - alloc[i][j];
}
int y = 0;
for (k = 0; k < 5; k++) {
            for (i = 0; i < n; i++) {
                        if (f[i] == 0) {
                                    int flag = 0;
                                    for (j = 0; j < m; j++) {
                                                if (need[i][j] > avail[j]){
                                                            flag = 1;
                                                            break;
                                                }
                                    }

                                    if (flag == 0) {
                                                ans[ind++] = i;
                                                for (y = 0; y < m; y++)
                                                            avail[y] += alloc[i][y];
                                                f[i] = 1;
                                    }
                        }
            }
}

int flag = 1;
for(int i=0;i<n;i++)
{
if(f[i]==0)
{
            flag=0;
            printf("The following system is not safe");
            break;
}
}
if(flag==1)
{
printf("Following is the SAFE Sequence\n");
for (i = 0; i < n - 1; i++)
            printf(" P%d ->", ans[i]);
printf(" P%d", ans[n - 1]);
}
return (0);

}
```

Output:

```
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
```

6.Develop a C program to simulate the following contiguous memory allocation Techniques: a) Worst fit b) Best fit c) First fit.

```
#include <stdio.h>
#include<limits.h>
// Define the maximum number of memory blocks
#define MAX_BLOCKS 100

// Structure to represent a memory block
struct MemoryBlock {
    int id;          // Block ID
    int size;        // Block size
    int allocated;   // 1 if allocated, 0 if free
};

// Function to initialize memory blocks
void initializeMemory(struct MemoryBlock memory[], int numBlocks) {
    for (int i = 0; i < numBlocks; i++) {
        memory[i].id = i + 1;
        memory[i].size = rand() % 10 + 1;  // Initialize with random sizes (1 to 10)
        memory[i].allocated = 0;  // All blocks are initially free
    }
}

// Function to display memory status
void displayMemory(struct MemoryBlock memory[], int numBlocks) {
    printf("Memory Status:\n");
    printf("ID\tSize\tStatus\n");
    for (int i = 0; i < numBlocks; i++) {
        printf("%d\t%d\t", memory[i].id, memory[i].size);
        if (memory[i].allocated) {
            printf("Allocated\n");
        } else {
            printf("Free\n");
        }
    }
    printf("\n");
}

// Function to allocate memory using First Fit
int firstFit(struct MemoryBlock memory[], int numBlocks, int blockSize) {
```

```
    for (int i = 0; i < numBlocks; i++) {
        if (!memory[i].allocated && memory[i].size >= blockSize) {
            memory[i].allocated = 1;
            return i;
        }
    }
    return -1; // No suitable block found
}

// Function to allocate memory using Best Fit
int bestFit(struct MemoryBlock memory[], int numBlocks, int blockSize) {
    int bestFitIdx = -1;
    int minFragmentation = INT_MAX;

    for (int i = 0; i < numBlocks; i++) {
        if (!memory[i].allocated && memory[i].size >= blockSize) {
            int fragmentation = memory[i].size - blockSize;
            if (fragmentation < minFragmentation) {
                bestFitIdx = i;
                minFragmentation = fragmentation;
            }
        }
    }

    if (bestFitIdx != -1) {
        memory[bestFitIdx].allocated = 1;
    }
    return bestFitIdx;
}

// Function to allocate memory using Worst Fit
int worstFit(struct MemoryBlock memory[], int numBlocks, int blockSize) {
    int worstFitIdx = -1;
    int maxFragmentation = -1;

    for (int i = 0; i < numBlocks; i++) {
        if (!memory[i].allocated && memory[i].size >= blockSize) {
            int fragmentation = memory[i].size - blockSize;
            if (fragmentation > maxFragmentation) {
                worstFitIdx = i;
                maxFragmentation = fragmentation;
            }
        }
    }

    if (worstFitIdx != -1) {
        memory[worstFitIdx].allocated = 1;
    }
    return worstFitIdx;
}
```

```
int main() {
    struct MemoryBlock memory[MAX_BLOCKS];
    int numBlocks, blockSize, choice;

    printf("Enter the number of memory blocks: ");
    scanf("%d", &numBlocks);

    initializeMemory(memory, numBlocks);

    while (1) {
        printf("\nMemory Allocation Techniques:\n");
        printf("1. First Fit\n");
        printf("2. Best Fit\n");
        printf("3. Worst Fit\n");
        printf("4. Display Memory Status\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the size of the block to allocate: ");
                scanf("%d", &blockSize);
                if (firstFit(memory, numBlocks, blockSize) == -1) {
                    printf("Memory allocation failed!\n");
                } else {
                    printf("Memory allocated successfully.\n");
                }
                break;
            case 2:
                printf("Enter the size of the block to allocate: ");
                scanf("%d", &blockSize);
                if (bestFit(memory, numBlocks, blockSize) == -1) {
                    printf("Memory allocation failed!\n");
                } else {
                    printf("Memory allocated successfully.\n");
                }
                break;
            case 3:
                printf("Enter the size of the block to allocate: ");
                scanf("%d", &blockSize);
                if (worstFit(memory, numBlocks, blockSize) == -1) {
                    printf("Memory allocation failed!\n");
                } else {
                    printf("Memory allocated successfully.\n");
                }
                break;
            case 4:
                displayMemory(memory, numBlocks);
```

```
            break;
        case 5:
            printf("Exiting program.\n");
            return 0;
        default:
            printf("Invalid choice!\n");
        }
    }

    return 0;
}
```

Output:

```
Enter the number of memory blocks: 5
Memory Allocation Techniques:
1. First Fit
2. Best Fit
3. Worst Fit
4. Display Memory Status
5. Exit
Enter your choice: 4
Memory Status:
ID  Size    Status
1   4   Free
2   7   Free
3   8   Free
4   6   Free
5   4   Free


Memory Allocation Techniques:
1. First Fit
2. Best Fit
3. Worst Fit
4. Display Memory Status
5. Exit
Enter your choice: 1
Enter the size of the block to allocate: 8
Memory allocated successfully.

Memory Allocation Techniques:
1. First Fit
2. Best Fit
3. Worst Fit
4. Display Memory Status
5. Exit
Enter your choice: 4
Memory Status:
ID  Size    Status
1   4   Free
2   7   Free
3   8   Allocated
4   6   Free
5   4   Free


Memory Allocation Techniques:
1. First Fit
2. Best Fit
3. Worst Fit
4. Display Memory Status
5. Exit
Enter your choice: 2
Enter the size of the block to allocate: 3
Memory allocated successfully.
```

7.Develop a C program to simulate page replacement algorithms: a) FIFO b) LRU

```
#include <stdio.h>

// Define the maximum number of page frames
#define MAX_FRAMES 3

// Function to initialize the page table
void initializePageTable(int pageTable[], int numFrames) {
    for (int i = 0; i < numFrames; i++) {
        pageTable[i] = -1; // Initialize with -1 (indicating an empty frame)
    }
}

// Function to display the page table
void displayPageTable(int pageTable[], int numFrames) {
    printf("Page Table: ");
    for (int i = 0; i < numFrames; i++) {
        if (pageTable[i] == -1) {
            printf("- ");
        } else {
            printf("%d ", pageTable[i]);
        }
    }
    printf("\n");
}

// Function to find the index of the page in the page table
int findPageIndex(int pageTable[], int numFrames, int page) {
    for (int i = 0; i < numFrames; i++) {
        if (pageTable[i] == page) {
            return i;
        }
    }
    return -1; // Page not found
}

// FIFO Page Replacement Algorithm
void fifo(int pages[], int numPages, int pageTable[], int numFrames) {
    int currentIndex = 0;

    for (int i = 0; i < numPages; i++) {
        int currentPage = pages[i];
        int pageIndex = findPageIndex(pageTable, numFrames, currentPage);

        if (pageIndex == -1) {
            // Page fault, replace the oldest page in the table (FIFO)
            pageTable[currentIndex] = currentPage;
            currentIndex = (currentIndex + 1) % numFrames;
        }

        displayPageTable(pageTable, numFrames);
```

```
    }
}

// LRU Page Replacement Algorithm
// LRU Page Replacement Algorithm
void lru(int pages[], int numPages, int pageTable[], int numFrames) {
    int pageOrder[MAX_FRAMES];
    int pageOrderIndex[MAX_FRAMES];

    for (int i = 0; i < numFrames; i++) {
        pageOrder[i] = -1;
        pageOrderIndex[i] = -1;
    }

    for (int i = 0; i < numPages; i++) {
        int currentPage = pages[i];
        int pageIndex = findPageIndex(pageTable, numFrames, currentPage);

        if (pageIndex == -1) {
            // Page fault, find the least recently used page in the table (LRU)
            int lruPageIndex = 0;

            for (int j = 1; j < numFrames; j++) {
                if (pageOrderIndex[j] < pageOrderIndex[lruPageIndex]) {
                    lruPageIndex = j;
                }
            }

            // Replace the LRU page with the new page
            pageTable[lruPageIndex] = currentPage;
            pageOrder[lruPageIndex] = currentPage;
            pageOrderIndex[lruPageIndex] = i;
        } else {
            // Page hit, update the page order index
            for (int j = 0; j < numFrames; j++) {
                if (pageTable[j] == currentPage) {
                    pageOrderIndex[j] = i;
                    break;
                }
            }
        }

        displayPageTable(pageTable, numFrames);
    }
}

int main() {
    int pages[] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5};
    int numPages = sizeof(pages) / sizeof(pages[0]);
```

```
    int fifoPageTable[MAX_FRAMES];
    int lruPageTable[MAX_FRAMES];

    initializePageTable(fifoPageTable, MAX_FRAMES);
    initializePageTable(lruPageTable, MAX_FRAMES);

    printf("FIFO Page Replacement:\n");
    fifo(pages, numPages, fifoPageTable, MAX_FRAMES);

    printf("\nLRU Page Replacement:\n");
    lru(pages, numPages, lruPageTable, MAX_FRAMES);

    return 0;
}
```

Output

```
FIFO Page Replacement:
Page Table: 1 - -
Page Table: 1 2 -
Page Table: 1 2 3
Page Table: 4 2 3
Page Table: 4 1 3
Page Table: 4 1 2
Page Table: 5 1 2
Page Table: 5 1 2
Page Table: 5 1 2
Page Table: 5 3 2
Page Table: 5 3 4
Page Table: 5 3 4

LRU Page Replacement:
Page Table: 1 - -
Page Table: 1 2 -
Page Table: 1 2 3
Page Table: 4 2 3
Page Table: 4 1 3
Page Table: 4 1 2
Page Table: 5 1 2
Page Table: 5 1 2
Page Table: 5 1 2
Page Table: 3 1 2
Page Table: 3 4 2
Page Table: 3 4 5
```

---

8. Simulate following File Organization Techniques a) Single level directory b) Two level directory

a) Single level directory

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

#define MAX_FILES 100

struct File {
    char name[20];
    int size;
};

struct Directory {
    struct File files[MAX_FILES];
    int fileCount;
};

void initializeDirectory(struct Directory* dir) {
    dir->fileCount = 0;
}

void createFile(struct Directory* dir, const char* name, int size) {
    if (dir->fileCount < MAX_FILES) {
        struct File newFile;
        strcpy(newFile.name, name);
        newFile.size = size;
        dir->files[dir->fileCount++] = newFile;
        printf("File '%s' created with size %d\n", name, size);
    } else {
        printf("Directory is full. Cannot create more files.\n");
    }
}

void listFiles(struct Directory* dir) {
    if (dir->fileCount == 0) {
        printf("No files in the directory.\n");
    } else {
        printf("Files in the directory:\n");
        for (int i = 0; i < dir->fileCount; i++) {
            printf("%s (%d bytes)\n", dir->files[i].name, dir->files[i].size);
        }
    }
}

int main() {
    struct Directory singleLevelDirectory;
    initializeDirectory(&singleLevelDirectory);

    createFile(&singleLevelDirectory, "file1.txt", 100);
    createFile(&singleLevelDirectory, "file2.txt", 200);
    createFile(&singleLevelDirectory, "file3.txt", 150);
```

```
    listFiles(&singleLevelDirectory);

    return 0;
}
```

Output

```
File 'file1.txt' created with size 100
File 'file2.txt' created with size 200
File 'file3.txt' created with size 150
Files in the directory:
file1.txt (100 bytes)
file2.txt (200 bytes)
file3.txt (150 bytes)
```

b)Two level directory

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

#define MAX_FILES_PER_DIRECTORY 10
#define MAX_FILE_NAME_LENGTH 20
#define MAX_DIRECTORIES 10

struct File {
    char name[MAX_FILE_NAME_LENGTH];
    // You can add other file attributes as needed
};

struct Directory {
    char name[MAX_FILE_NAME_LENGTH];
    struct File files[MAX_FILES_PER_DIRECTORY];
    int fileCount;
};

struct FileSystem {
    struct Directory directories[MAX_DIRECTORIES];
    int directoryCount;
};

void createDirectory(struct FileSystem *fs, char *dirName) {
    if (fs->directoryCount < MAX_DIRECTORIES) {
        strcpy(fs->directories[fs->directoryCount].name, dirName);
        fs->directories[fs->directoryCount].fileCount = 0;
        fs->directoryCount++;
        printf("Directory '%s' created successfully.\n", dirName);
    } else {
```

```
        printf("Cannot create more directories. Maximum limit reached.\n");
    }
}

void createFile(struct FileSystem *fs, char *dirName, char *fileName) {
    int dirIndex = -1;
    for (int i = 0; i < fs->directoryCount; i++) {
        if (strcmp(fs->directories[i].name, dirName) == 0) {
            dirIndex = i;
            break;
        }
    }

    if (dirIndex != -1) {
        struct Directory *dir = &fs->directories[dirIndex];
        if (dir->fileCount < MAX_FILES_PER_DIRECTORY) {
            strcpy(dir->files[dir->fileCount].name, fileName);
            dir->fileCount++;
            printf("File '%s' created in directory '%s'.\n", fileName, dirName);
        } else {
            printf("Cannot create more files in directory '%s'. Maximum limit reached.\n",
dirName);
        }
    } else {
        printf("Directory '%s' not found.\n", dirName);
    }
}

void displayFiles(struct FileSystem *fs, char *dirName) {
    int dirIndex = -1;
    for (int i = 0; i < fs->directoryCount; i++) {
        if (strcmp(fs->directories[i].name, dirName) == 0) {
            dirIndex = i;
            break;
        }
    }

    if (dirIndex != -1) {
        struct Directory *dir = &fs->directories[dirIndex];
        printf("Files in directory '%s':\n", dirName);
        for (int i = 0; i < dir->fileCount; i++) {
            printf("- %s\n", dir->files[i].name);
        }
    } else {
        printf("Directory '%s' not found.\n", dirName);
    }
}

int main() {
    struct FileSystem fileSystem;
```

```
    fileSystem.directoryCount = 0;

    int choice;
    char dirName[MAX_FILE_NAME_LENGTH];
    char fileName[MAX_FILE_NAME_LENGTH];

    do {
        printf("\n--- Two-Level Directory File Organization ---\n");
        printf("1. Create Directory\n");
        printf("2. Create File\n");
        printf("3. Display Files in Directory\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the name of the directory: ");
                scanf("%s", dirName);
                createDirectory(&fileSystem, dirName);
                break;

            case 2:
                printf("Enter the name of the directory: ");
                scanf("%s", dirName);
                printf("Enter the name of the file: ");
                scanf("%s", fileName);
                createFile(&fileSystem, dirName, fileName);
                break;

            case 3:
                printf("Enter the name of the directory: ");
                scanf("%s", dirName);
                displayFiles(&fileSystem, dirName);
                break;

            case 4:
                printf("Exiting the program.\n");
                break;

            default:
                printf("Invalid choice. Please enter a number between 1 and 4.\n");
        }

    } while (choice != 4);

    return 0;
}
```

Output:

```
--- Two-Level Directory File Organization ---
1. Create Directory
2. Create File
3. Display Files in Directory
4. Exit
Enter your choice: 1
Enter the name of the directory: Document
Directory 'Document' created successfully.

--- Two-Level Directory File Organization ---
1. Create Directory
2. Create File
3. Display Files in Directory
4. Exit
Enter your choice: 2
Enter the name of the directory: Document
Enter the name of the file: Report.txt
File 'Report.txt' created in directory 'Document'.

--- Two-Level Directory File Organization ---
1. Create Directory
2. Create File
3. Display Files in Directory
4. Exit
Enter your choice: 3
Enter the name of the directory: Document
Files in directory 'Document':
- Report.txt
```

9.Develop a C program to simulate the Linked file allocation strategies.

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

// Structure to represent a disk block
struct DiskBlock {
    int blockNumber;
    struct DiskBlock* nextBlock;
};

// Structure to represent a file
struct File {
    char name[50];
    struct DiskBlock* firstBlock;
};

// Function to create a new disk block
struct DiskBlock* createDiskBlock(int blockNumber) {
    struct DiskBlock* newBlock = (struct DiskBlock*)malloc(sizeof(struct DiskBlock));
    newBlock->blockNumber = blockNumber;
    newBlock->nextBlock = NULL;
    return newBlock;
}
```

```c
// Function to create a new file
struct File* createFile(const char* name) {
    struct File* newFile = (struct File*)malloc(sizeof(struct File));
    strcpy(newFile->name, name);
    newFile->firstBlock = NULL;
    return newFile;
}

// Function to add a block to the end of a file
void addBlockToFile(struct File* file, struct DiskBlock* block) {
    if (file->firstBlock == NULL) {
        file->firstBlock = block;
    } else {
        struct DiskBlock* currentBlock = file->firstBlock;
        while (currentBlock->nextBlock != NULL) {
            currentBlock = currentBlock->nextBlock;
        }
        currentBlock->nextBlock = block;
    }
}

// Function to display the blocks in a file
void displayFile(struct File* file) {
    printf("File: %s\n", file->name);
    if (file->firstBlock == NULL) {
        printf("Empty file\n");
    } else {
        printf("Blocks: ");
        struct DiskBlock* currentBlock = file->firstBlock;
        while (currentBlock != NULL) {
            printf("%d ", currentBlock->blockNumber);
            currentBlock = currentBlock->nextBlock;
        }
        printf("\n");
    }
}

int main() {
    struct File* file1 = createFile("file1.txt");
    struct File* file2 = createFile("file2.txt");

    // Create some disk blocks
    struct DiskBlock* block1 = createDiskBlock(1);
    struct DiskBlock* block2 = createDiskBlock(2);
    struct DiskBlock* block3 = createDiskBlock(3);
    struct DiskBlock* block4 = createDiskBlock(4);

    // Add blocks to files
    addBlockToFile(file1, block1);
    addBlockToFile(file1, block2);
```

```
    addBlockToFile(file2, block3);
    addBlockToFile(file2, block4);

    // Display files
    displayFile(file1);
    displayFile(file2);

    // Clean up memory
    free(file1);
    free(file2);
    free(block1);
    free(block2);
    free(block3);
    free(block4);

    return 0;
}
```

Output

```
File: file1.txt
Blocks: 1 2
File: file2.txt
Blocks: 3 4
```

---

10. Develop a C program to simulate SCAN disk scheduling algorithm.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;
    printf("Enter the number of Requests\n");
    scanf("%d",&n);
    printf("Enter the Requests sequence\n");
    for(i=0;i<n;i++)
     scanf("%d",&RQ[i]);
    printf("Enter initial head position\n");
    scanf("%d",&initial);
    printf("Enter total disk size\n");
    scanf("%d",&size);
    printf("Enter the head movement direction for high 1 and for low 0\n");
    scanf("%d",&move);

    // logic for C-Scan disk scheduling

        /*logic for sort the request array */
    for(i=0;i<n;i++)
    {
        for( j=0;j<n-i-1;j++)
```

```
    {
        if(RQ[j]>RQ[j+1])
        {
            int temp;
            temp=RQ[j];
            RQ[j]=RQ[j+1];
            RQ[j+1]=temp;
        }

    }
}

int index;
for(i=0;i<n;i++)
{
    if(initial<RQ[i])
    {
        index=i;
        break;
    }
}

// if movement is towards high value
if(move==1)
{
    for(i=index;i<n;i++)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }
    //  last movement for max size
    TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);
    /*movement max to min disk */
    TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
    initial=0;
    for( i=0;i<index;i++)
    {
         TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
         initial=RQ[i];

    }
}
// if movement is towards low value
else
{
    for(i=index-1;i>=0;i--)
    {
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
        initial=RQ[i];
    }
```

```
        //  last movement for min size
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
        /*movement min to max disk */
        TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
        initial =size-1;
        for(i=n-1;i>=index;i--)
        {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];

        }
    }

    printf("Total head movement is %d",TotalHeadMoment);
    return 0;
}
```

Output:

```
Enter the number of Requests
8
Enter the Requests sequence
95 180 34 119 11 123 62 64
Enter initial head position
50
Enter total disk size
200
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 382
```