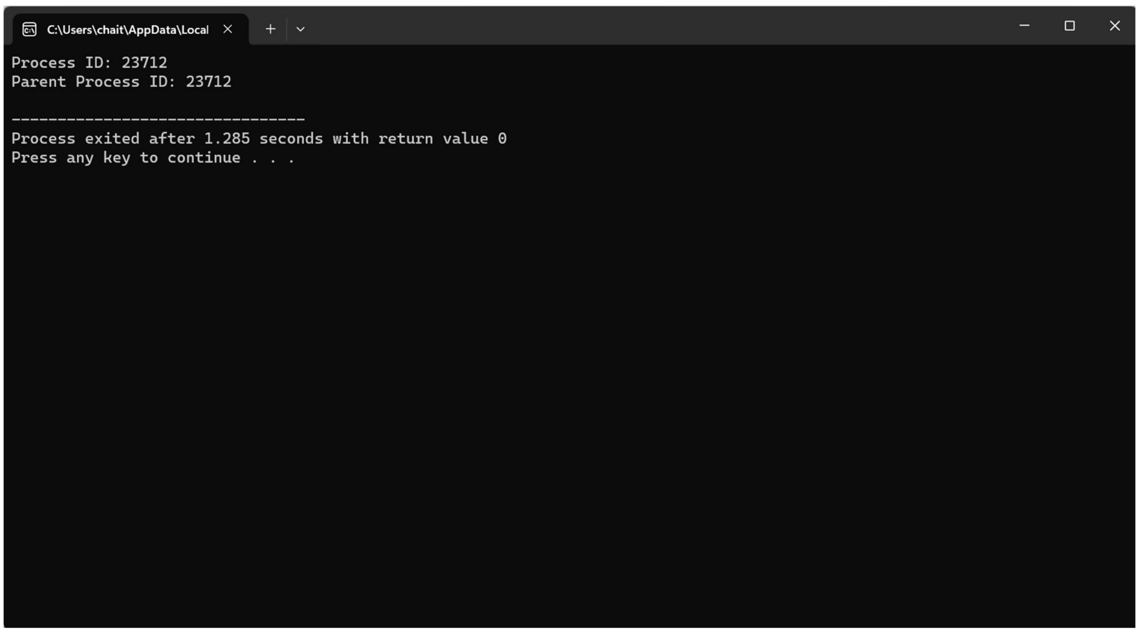
1. **Create a new process by invoking the appropriate system call. Get the process identifier of the currently running process and its respective parent using system calls and display the same using a C program.**

```
#include<stdio.h>

#include<unistd.h>

int main()

{

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

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

return 0;

}
```

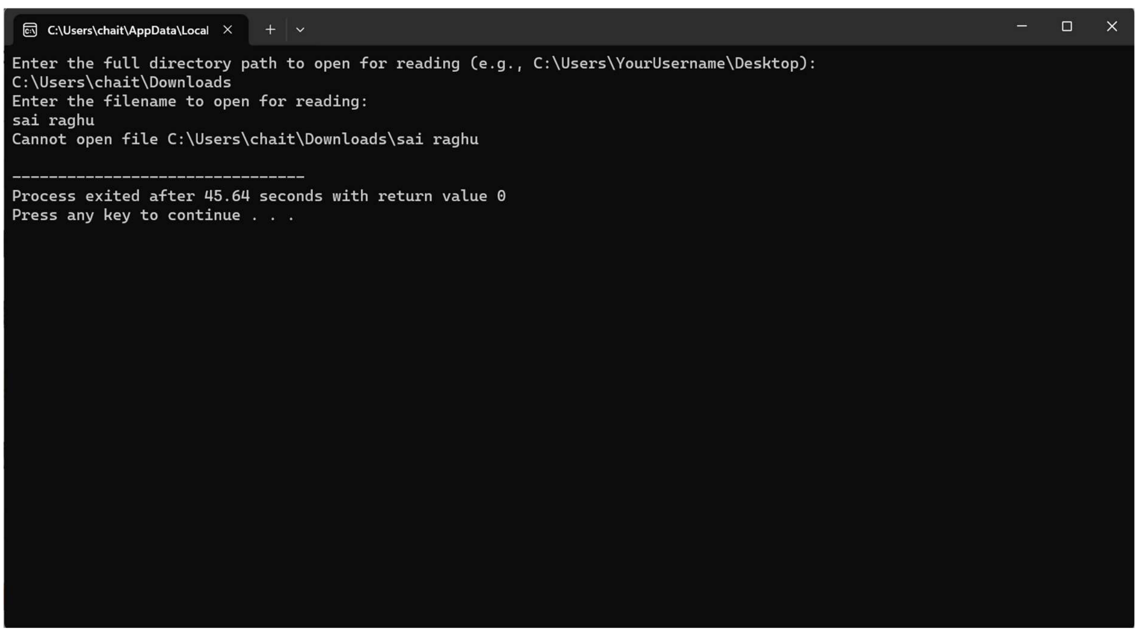


2. **Identify the system calls to copy the content of one file to another and illustrate the same using a C program.**

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void trimNewline(char* string) {

size_t length = strlen(string);

if (length > 0 && string[length - 1] == '\n') {

string[length - 1] = '\0';
```

```
}
}
int main() {
FILE *fptr1, *fptr2;
char readPath[1000], writePath[1000], readFilename[100], writeFilename[100];
char readFullPath[1100], writeFullPath[1100];
// For reading
printf("Enter the full directory path to open for reading (e.g., C:\\Users\\YourUsername\\Desktop):\n");
fgets(readPath, sizeof(readPath), stdin);
trimNewline(readPath); // Remove the newline character
printf("Enter the filename to open for reading:\n");
fgets(readFilename, sizeof(readFilename), stdin);
trimNewline(readFilename); // Remove the newline character
snprintf(readFullPath, sizeof(readFullPath), "%s\\%s", readPath, readFilename);
fptr1 = fopen(readFullPath, "r");
if (fptr1 == NULL) {
printf("Cannot open file %s\n", readFullPath);
exit(0);
}
// For writing
printf("Enter the full directory path to open for writing (e.g., C:\\Users\\YourUsername\\Desktop):\n");
fgets(writePath, sizeof(writePath), stdin);
trimNewline(writePath); // Remove the newline character
printf("Enter the filename to open for writing:\n");
fgets(writeFilename, sizeof(writeFilename), stdin);
trimNewline(writeFilename); // Remove the newline character
snprintf(writeFullPath, sizeof(writeFullPath), "%s\\%s", writePath, writeFilename);
fptr2 = fopen(writeFullPath, "w");
if (fptr2 == NULL) {
printf("Cannot open file %s\n", writeFullPath);
```

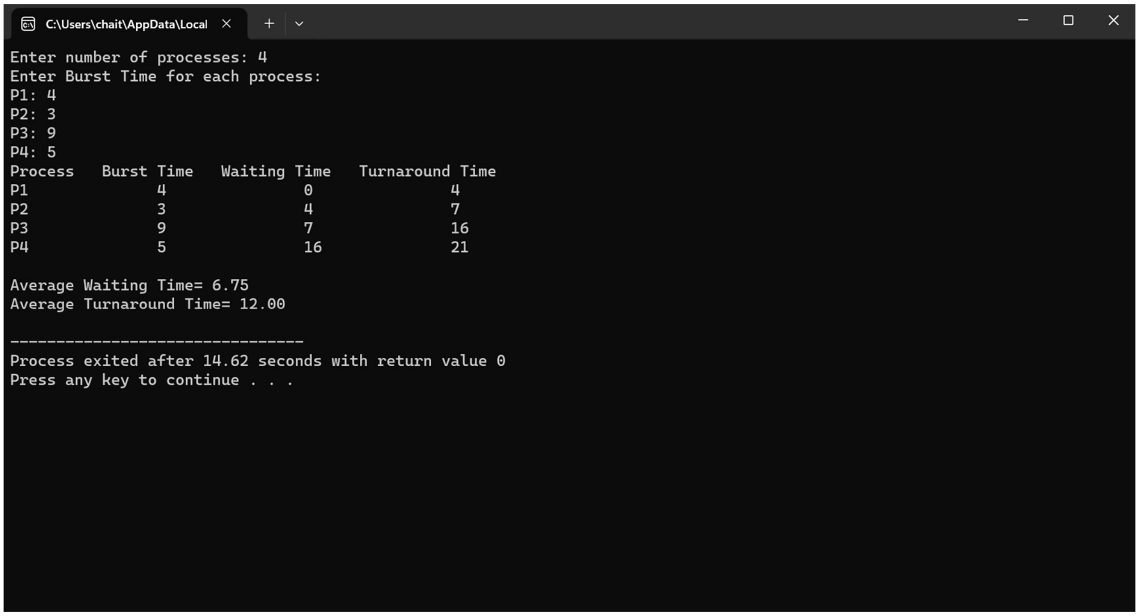
```
exit(0);

}

// Copying contents

char c = fgetc(fptr1);

while (c != EOF) {

fputc(c, fptr2);

c = fgetc(fptr1);

}

printf("\nContents copied to %s\n", writeFullPath);

fclose(fptr1);

fclose(fptr2);

return 0;

}
```



3. **Design a CPU scheduling program with C using First Come First Served technique with the following considerations. a. All processes are activated at time0. b. Assume that no process waits on I/O devices.**

```
#include <stdio.h>

int main() {

int processes[100][3];  // Array to store process details: [Process ID, Burst Time, Waiting Time]

int n, i, j, total_waiting_time = 0, total_turnaround_time = 0;
```

```
printf("Enter number of processes: ");

scanf("%d", &n);

// Input burst times for each process

printf("Enter Burst Time for each process:\n");

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

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

scanf("%d", &processes[i][1]); // Index 1 stores Burst Time

processes[i][0] = i + 1; // Index 0 stores Process ID

}

// Calculate waiting time for each process

processes[0][2] = 0; // First process has 0 waiting time

for (i = 1; i < n; i++) {

processes[i][2] = processes[i - 1][2] + processes[i - 1][1]; // Waiting Time = Previous Waiting Time + Previous Burst Time

total_waiting_time += processes[i][2];

}

// Calculate turnaround time and display process details

printf("Process   Burst Time   Waiting Time   Turnaround Time\n");

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

int turnaround_time = processes[i][1] + processes[i][2]; // Turnaround Time = Burst Time + Waiting Time

total_turnaround_time += turnaround_time;

printf("P%d\t\t%d\t\t%d\t\t%d\n", processes[i][0], processes[i][1], processes[i][2], turnaround_time);

}

// Calculate and display average waiting time and average turnaround time

float avg_waiting_time = (float)total_waiting_time / n;

float avg_turnaround_time = (float)total_turnaround_time / n;

printf("\nAverage Waiting Time= %.2f\n", avg_waiting_time);

printf("Average Turnaround Time= %.2f\n", avg_turnaround_time);

return 0;

}
```



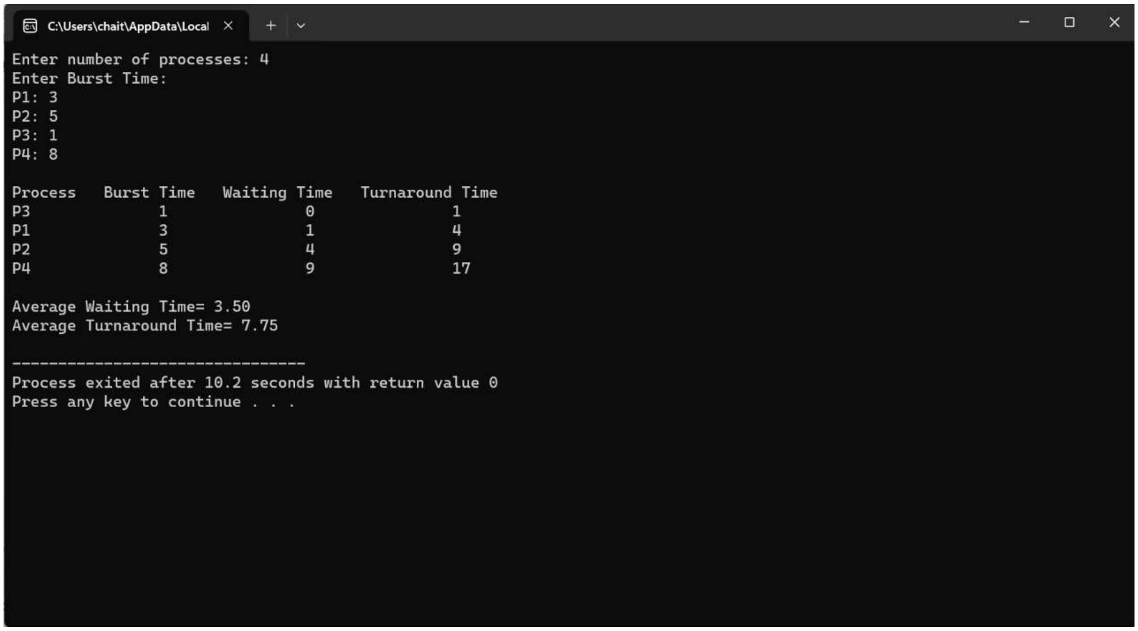
```
C:\Users\chait\AppData\Local
Enter number of processes: 4
Enter Burst Time for each process:
P1: 4
P2: 3
P3: 9
P4: 5
Process   Burst Time   Waiting Time   Turnaround Time
P1            4             0              4
P2            3             4              7
P3            9             7              16
P4            5             16             21

Average Waiting Time= 6.75
Average Turnaround Time= 12.00

--------------------------------
Process exited after 14.62 seconds with return value 0
Press any key to continue . . .
```

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

```
#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 processes: ");

scanf("%d", &n);

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

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

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

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

p[i] = i + 1;

}

// Sort processes based on burst time (Selection Sort)

for (i = 0; i < n - 1; i++) {

pos = i;

for (j = i + 1; j < n; j++) {

if (bt[j] < bt[pos])
```

```
pos = j;
}
// Swap burst time and process IDs
temp = bt[i];
bt[i] = bt[pos];
bt[pos] = temp;
temp = p[i];
p[i] = p[pos];
p[pos] = temp;
}
wt[0] = 0; // Waiting time for the first process is always 0
total = 0;
// Calculate waiting time for each process
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; // Calculate average waiting time
total = 0;
printf("\nProcess   Burst Time   Waiting Time   Turnaround Time\n");
for (i = 0; i < n; i++) {
tat[i] = bt[i] + wt[i]; // Calculate turnaround time
total += tat[i];
printf("P%d\t\t%d\t\t%d\t\t%d\n", p[i], bt[i], wt[i], tat[i]);
}
avg_tat = (float)total / n; // Calculate average turnaround time
printf("\nAverage Waiting Time= %.2f\n", avg_wt);
printf("Average Turnaround Time= %.2f\n", avg_tat);
return 0;
```

```
}
```



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

```
#include<stdio.h>

struct Process {

char process_name;

int burst_time;

int waiting_time;

int turn_around_time;

int priority;

};

int main() {

int number_of_processes;

int total_waiting_time = 0;

struct Process 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_processes);
struct Process processes[number_of_processes];
printf("\nPlease Enter the Burst Time and Priority of each process:\n");
for (int i = 0; i < number_of_processes; i++) {
processes[i].process_name = (char) ASCII_number;
printf("\nEnter the details of the process %c\n", processes[i].process_name);
printf("Enter the burst time: ");
scanf("%d", &processes[i].burst_time);
printf("Enter the priority: ");
scanf("%d", &processes[i].priority);
ASCII_number++;
}
// Sort processes based on priority (Highest priority first)
for (int i = 0; i < number_of_processes - 1; i++) {
position = i;
for (int j = i + 1; j < number_of_processes; j++) {
if (processes[j].priority > processes[position].priority)
position = j;
}
temp_process = processes[i];
processes[i] = processes[position];
processes[position] = temp_process;
}
processes[0].waiting_time = 0;
// Calculate waiting time for each process
for (int i = 1; i < number_of_processes; i++) {
processes[i].waiting_time = 0;
for (int j = 0; j < i; j++) {
processes[i].waiting_time += processes[j].burst_time;
}
```

total_waiting_time += processes[i].waiting_time;

}

average_waiting_time = (float) total_waiting_time / (float) number_of_processes;

// Calculate turnaround time for each process and display process details

printf("\n\nProcess_name\tBurst Time\tWaiting Time\tTurnaround Time\n");

printf("--\n");

int total_turnaround_time = 0;

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

processes[i].turn_around_time = processes[i].burst_time + processes[i].waiting_time;

total_turnaround_time += processes[i].turn_around_time;

printf("\t%c\t\t%d\t\t%d\t\t%d\n", processes[i].process_name, processes[i].burst_time, processes[i].waiting_time, processes[i].turn_around_time);

printf("--\n");

}

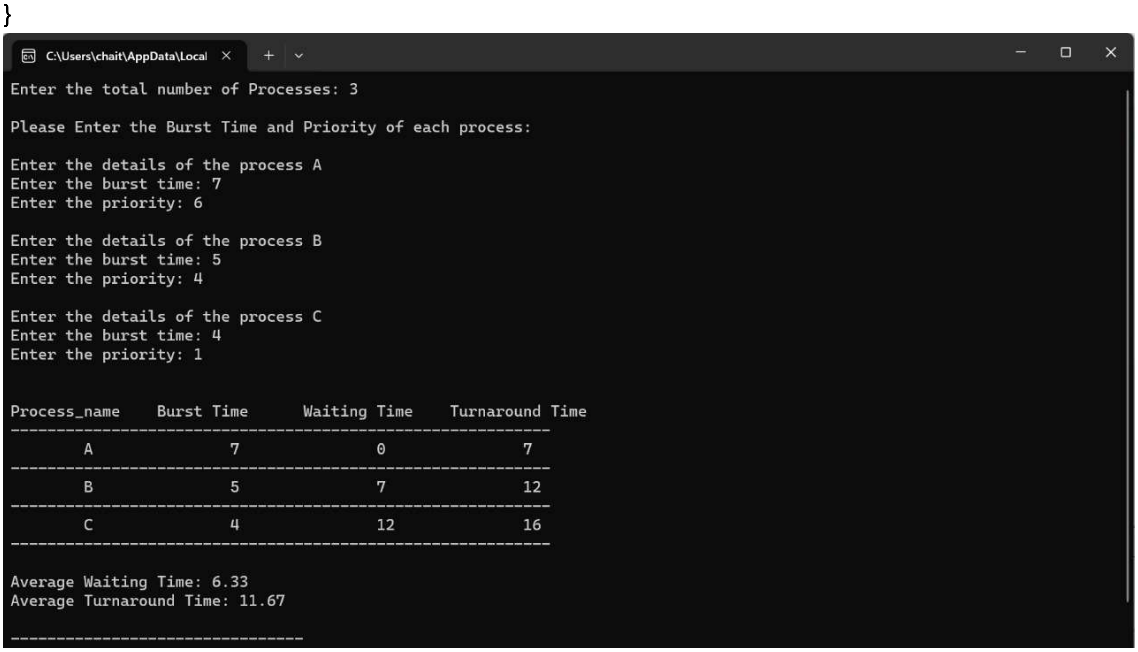
average_turnaround_time = (float) total_turnaround_time / (float) number_of_processes;

printf("\nAverage Waiting Time: %.2f\n", average_waiting_time);

printf("Average Turnaround Time: %.2f\n", average_turnaround_time);

return 0;

}

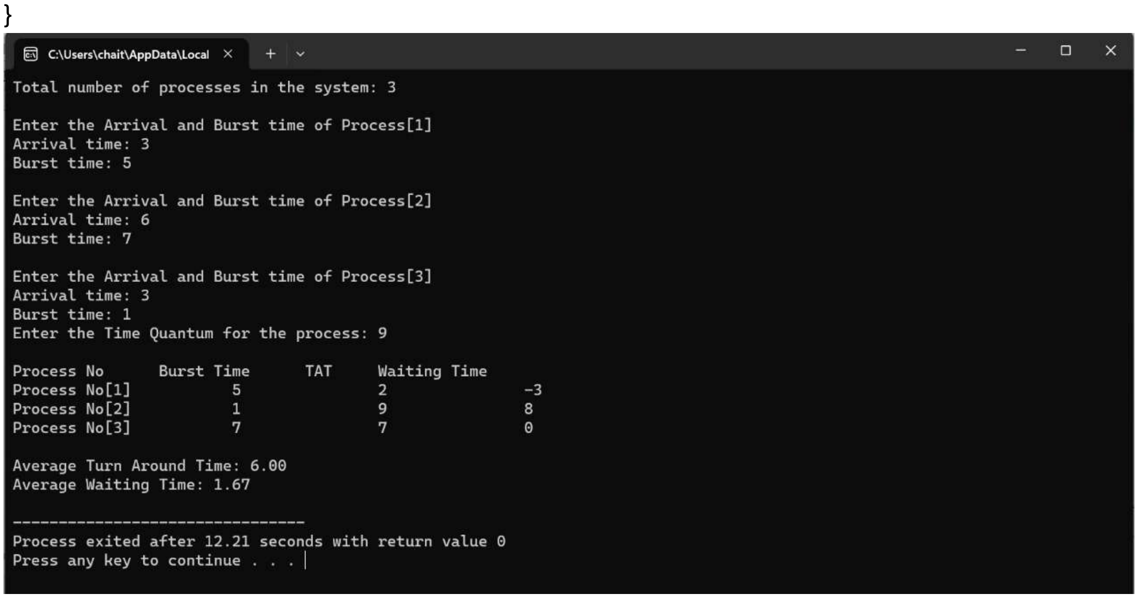


```
C:\Users\chait\AppData\Local
Enter the total number of Processes: 3

Please Enter the Burst Time and Priority of each process:

Enter the details of the process A
Enter the burst time: 7
Enter the priority: 6

Enter the details of the process B
Enter the burst time: 5
Enter the priority: 4

Enter the details of the process C
Enter the burst time: 4
Enter the priority: 1


Process_name    Burst Time      Waiting Time    Turnaround Time
----------------------------------------------------------
        A               7               0               7
----------------------------------------------------------
        B               5               7               12
----------------------------------------------------------
        C               4               12              16
----------------------------------------------------------

Average Waiting Time: 6.33
Average Turnaround Time: 11.67

--------------------------------
```

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

```
#include<stdio.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 processes in the system: ");
scanf("%d", &NOP);
y = NOP;
// Input arrival and burst time for each process
for(i=0; i<NOP; i++)
{
printf("\nEnter the Arrival and Burst time of Process[%d]\n", i+1);
printf("Arrival time: ");
scanf("%d", &at[i]);
printf("Burst time: ");
scanf("%d", &bt[i]);
temp[i] = bt[i];
}
printf("Enter the Time Quantum for the process: ");
scanf("%d", &quant);
// Sorting processes based on arrival time
for(i=0; i<NOP-1; i++) {
for(int j=i+1; j<NOP; j++) {
if(at[i] > at[j]) {
int temp = at[i];
at[i] = at[j];
at[j] = temp;
temp = bt[i];
bt[i] = bt[j];
bt[j] = temp;
}
}
}
printf("\nProcess No\tBurst Time\tTAT\tWaiting Time\n");
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("Process No[%d]\t\t%d\t\t%d\t\t%d\n", 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;
}
i = (i + 1) % NOP;
}
avg_wt = (float)wt / NOP;
avg_tat = (float)tat / NOP;
printf("\nAverage Turn Around Time: %.2f", avg_tat);
printf("\nAverage Waiting Time: %.2f\n", avg_wt);
return 0;
}
```

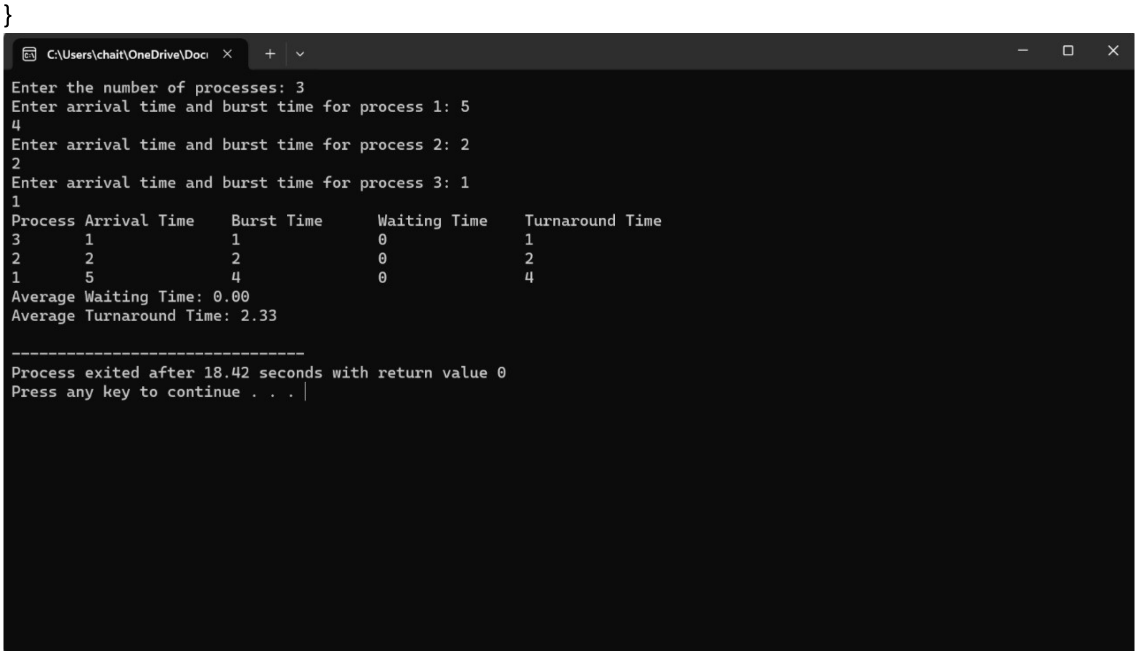


7. **Construct a C program to implement non-preemptive SJF algorithm.**

```
#include <stdio.h>
// Process structure
typedef struct {
int process_id;
int arrival_time;
int burst_time;
} Process;

// Function to perform non-preemptive SJF scheduling
void sjf(Process processes[], int n) {
int waiting_time[n], turnaround_time[n];
```

```
// Initialize waiting time and turnaround time arrays
for (int i = 0; i < n; i++) {
waiting_time[i] = 0;
turnaround_time[i] = 0;
}

// Sort processes based on arrival time
for (int i = 0; i < n - 1; i++) {
for (int j = 0; j < n - i - 1; j++) {
if (processes[j].arrival_time > processes[j + 1].arrival_time) {
Process temp = processes[j];
processes[j] = processes[j + 1];
processes[j + 1] = temp;
}
}
}

int total_waiting_time = 0;
int total_turnaround_time = 0;
int current_time = 0;

// Calculate waiting time and turnaround time
for (int i = 0; i < n; i++) {
// Calculate waiting time for current process
waiting_time[i] = current_time - processes[i].arrival_time;
if (waiting_time[i] < 0)
waiting_time[i] = 0;

// Calculate turnaround time for current process
turnaround_time[i] = waiting_time[i] + processes[i].burst_time;

// Update current time
current_time += processes[i].burst_time;

// Update total waiting time and total turnaround time
total_waiting_time += waiting_time[i];
total_turnaround_time += turnaround_time[i];
}

// Calculate average waiting time and average turnaround time
float avg_waiting_time = (float)total_waiting_time / n;
float avg_turnaround_time = (float)total_turnaround_time / n;

// Print results
printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
for (int i = 0; i < n; i++) {
printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].process_id, processes[i].arrival_time,
processes[i].burst_time, waiting_time[i], turnaround_time[i]);
```

```
}
printf("Average Waiting Time: %.2f\n", avg_waiting_time);
printf("Average Turnaround Time: %.2f\n", avg_turnaround_time);
}

int main() {
int n;
printf("Enter the number of processes: ");
scanf("%d", &n);

// Array to store processes
Process processes[n];

// Input process details
for (int i = 0; i < n; i++) {
printf("Enter arrival time and burst time for process %d: ", i + 1);
scanf("%d %d", &processes[i].arrival_time, &processes[i].burst_time);
processes[i].process_id = i + 1;
}

sjf(processes, n);

return 0;
}
```

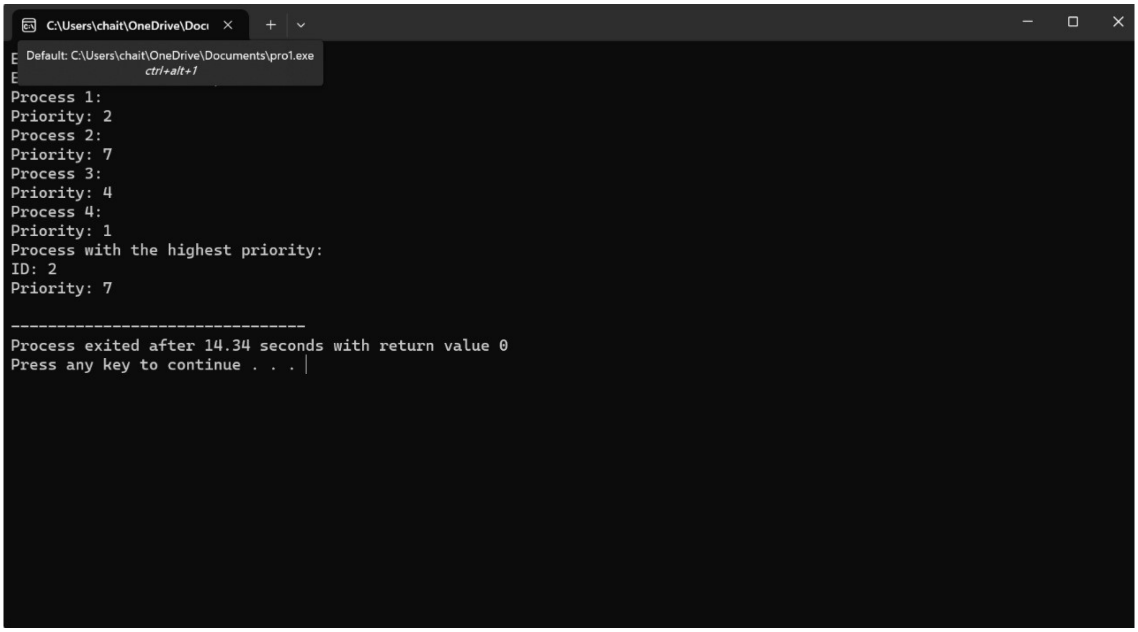


8. Construct a C program to simulate Round Robin scheduling algorithm with C.

```
#include<stdio.h>
#include<stdlib.h>
#define MAX_PROCESSES 10
```

```
struct Process {
int id;
int priority;
};
// Function to select the process with the highest priority
struct Process selectHighestPriority(struct Process processes[], int n) {
struct Process highestPriorityProcess = processes[0];
for (int i = 1; i < n; i++) {
if (processes[i].priority > highestPriorityProcess.priority) {
highestPriorityProcess = processes[i];
}
}
return highestPriorityProcess;
}
int main() {
struct Process processes[MAX_PROCESSES];
int n;
// Input the number of processes
printf("Enter the number of processes: ");
scanf("%d", &n);
// Input details of each process
printf("Enter details of each process:\n");
for (int i = 0; i < n; i++) {
printf("Process %d:\n", i + 1);
processes[i].id = i + 1;
printf("Priority: ");
scanf("%d", &processes[i].priority);
}

// Select the process with the highest priority
struct Process nextProcess = selectHighestPriority(processes, n);

// Display the selected process
printf("Process with the highest priority:\n");
printf("ID: %d\n", nextProcess.id);
printf("Priority: %d\n", nextProcess.priority);

return 0;
```

```
}
```



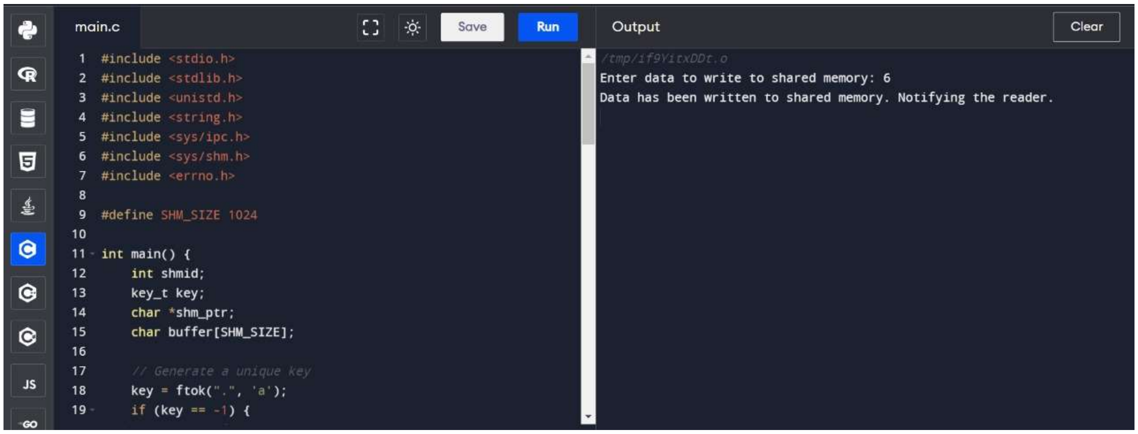
9. **Illustrate the concept of inter-process communication using shared memory with a C program.**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <errno.h>

#define SHM_SIZE 1024

int main() {
int shmid;
key_t key;
char *shm_ptr;
char buffer[SHM_SIZE];

// Generate a unique key
key = ftok(".", 'a');
if (key == -1) {
perror("ftok");
exit(EXIT_FAILURE);
}

// Create a shared memory segment
shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666);
if (shmid == -1) {
```

```
perror("shmget");
exit(EXIT_FAILURE);
}

// Attach the shared memory segment
shm_ptr = shmat(shmid, NULL, 0);
if (shm_ptr == (char *)-1) {
perror("shmat");
exit(EXIT_FAILURE);
}

// Writing data to the shared memory
printf("Enter data to write to shared memory: ");
if (fgets(buffer, sizeof(buffer), stdin) == NULL) {
perror("fgets");
exit(EXIT_FAILURE);
}
strncpy(shm_ptr, buffer, SHM_SIZE - 1); // Ensure null-terminated string
shm_ptr[SHM_SIZE - 1] = '\0'; // Null terminate explicitly

// Notify the reader that data is ready
printf("Data has been written to shared memory. Notifying the reader.\n");
*shm_ptr = '%'; // Notify by writing to shared memory
shm_ptr = NULL; // Reset shm_ptr

// Wait for the reader to finish reading
while (1) {
shm_ptr = shmat(shmid, NULL, 0); // Reattach the shared memory segment
if (shm_ptr == (char *)-1) {
perror("shmat");
exit(EXIT_FAILURE);
}
if (*shm_ptr == '%') {
break;
}
sleep(1);
}

// Detach the shared memory segment
if (shmdt(shm_ptr) == -1) {
perror("shmdt");
exit(EXIT_FAILURE);
}

// Delete the shared memory segment
if (shmctl(shmid, IPC_RMID, NULL) == -1) {
perror("shmctl");
exit(EXIT_FAILURE);
```

```
}

return 0;
}
```

THIS QUESTION CAN YOU USE THE ONLINE COMPILER

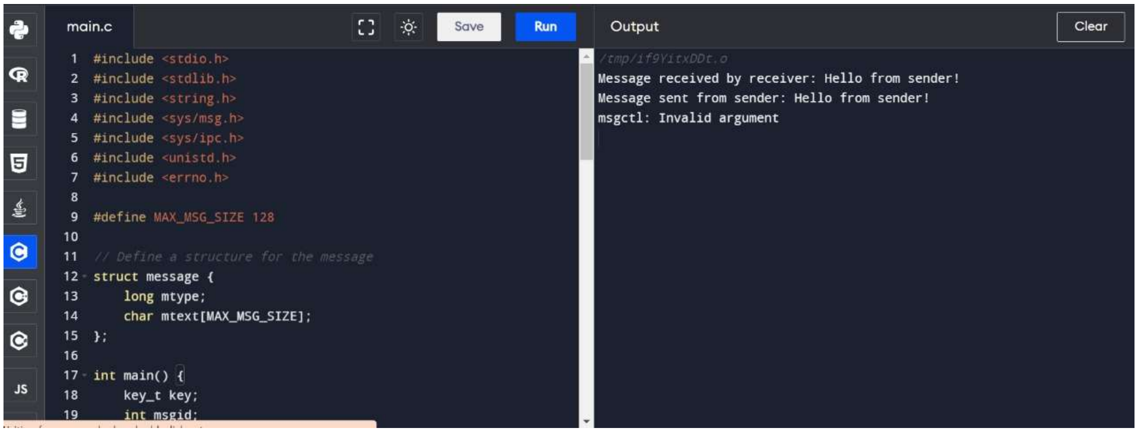


10. Illustrate the concept of inter process communication using message queue with a C program.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/msg.h>
#include <sys/ipc.h>
#include <unistd.h>
#include <errno.h>

#define MAX_MSG_SIZE 128

// Define a structure for the message
struct message {
long mtype;
char mtext[MAX_MSG_SIZE];
};

int main() {
key_t key;
int msgid;
struct message msg;

// Generate a unique key for the message queue
key = ftok(".", 'a');
if (key == -1) {
perror("ftok");
exit(EXIT_FAILURE);
}
```

```
// Create or access the message queue
msgid = msgget(key, 0666 | IPC_CREAT);
if (msgid == -1) {
perror("msgget");
exit(EXIT_FAILURE);
}

// Sender process
if (fork() == 0) {
// Construct message
msg.mtype = 1; // Message type
strcpy(msg.mtext, "Hello from sender!");

// Send message to the message queue
if (msgsnd(msgid, &msg, sizeof(msg.mtext), 0) == -1) {
perror("msgsnd");
exit(EXIT_FAILURE);
}

printf("Message sent from sender: %s\n", msg.mtext);
}
// Receiver process
else {
// Receive message from the message queue
if (msgrcv(msgid, &msg, sizeof(msg.mtext), 1, 0) == -1) {
perror("msgrcv");
exit(EXIT_FAILURE);
}

printf("Message received by receiver: %s\n", msg.mtext);
}

// Remove the message queue
if (msgctl(msgid, IPC_RMID, NULL) == -1) {
perror("msgctl");
exit(EXIT_FAILURE);
}

return 0;
}
```

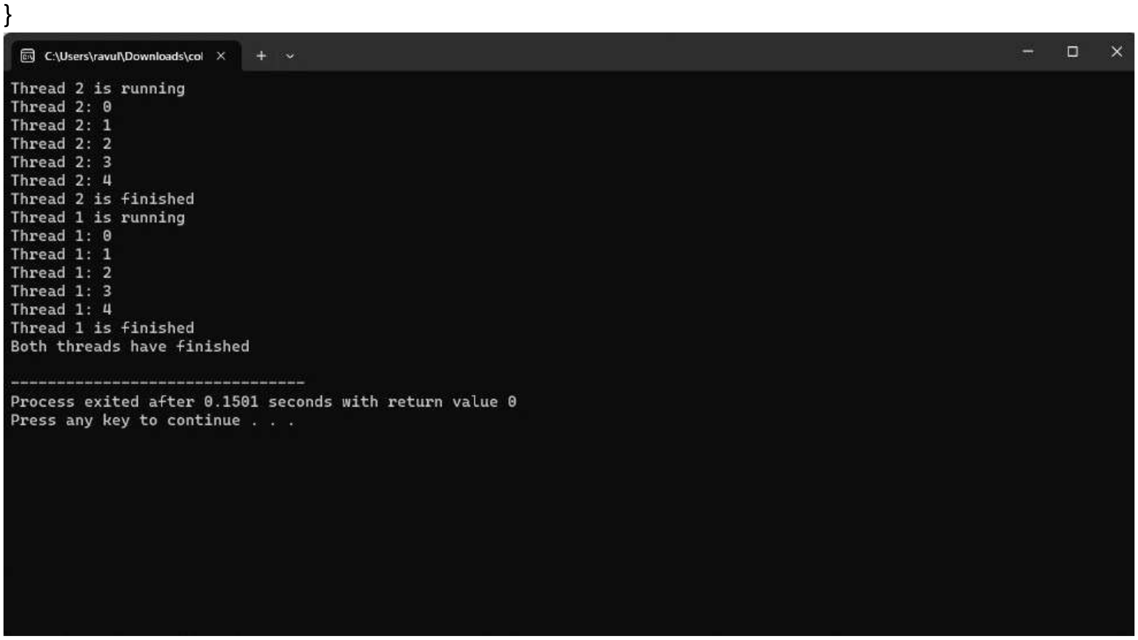
THIS QUESTION CAN YOU USE THE ONLINE COMPILER



11. Illustrate the concept of multi threading using a C program.

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

// Function executed by the first thread
void *threadFunction1(void *arg) {
printf("Thread 1 is running\n");
for (int i = 0; i < 5; i++) {
printf("Thread 1: %d\n", i);
}
printf("Thread 1 is finished\n");
return NULL;
}

// Function executed by the second thread
void *threadFunction2(void *arg) {
printf("Thread 2 is running\n");
for (int i = 0; i < 5; i++) {
printf("Thread 2: %d\n", i);
}
printf("Thread 2 is finished\n");
return NULL;
}

int main() {
pthread_t tid1, tid2; // Thread IDs

// Create the first thread
if (pthread_create(&tid1, NULL, threadFunction1, NULL) != 0) {
perror("pthread_create");
exit(EXIT_FAILURE);
}

// Create the second thread
```

```
if (pthread_create(&tid2, NULL, threadFunction2, NULL) != 0) {
perror("pthread_create");
exit(EXIT_FAILURE);
}

// Wait for the first thread to finish
if (pthread_join(tid1, NULL) != 0) {
perror("pthread_join");
exit(EXIT_FAILURE);
}

// Wait for the second thread to finish
if (pthread_join(tid2, NULL) != 0) {
perror("pthread_join");
exit(EXIT_FAILURE);
}

printf("Both threads have finished\n");

return 0;
}
```

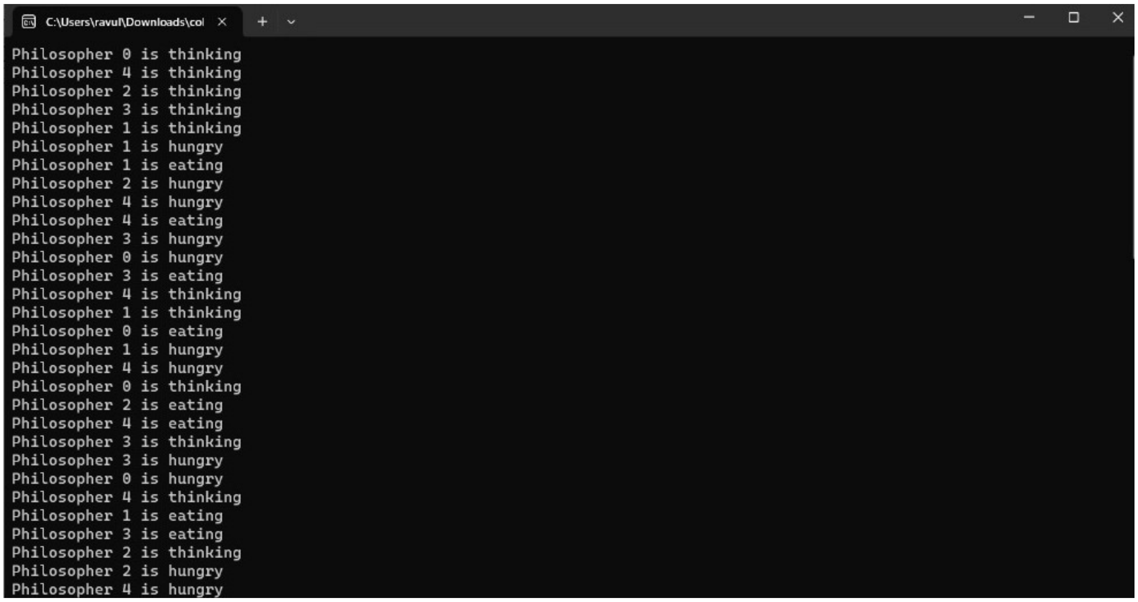


12. **Design a C program to simulate the concept of Dining-Philosophers problem.**

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

#define NUM_PHILOSOPHERS 5
#define LEFT (id + NUM_PHILOSOPHERS - 1) % NUM_PHILOSOPHERS
#define RIGHT (id + 1) % NUM_PHILOSOPHERS
```

```
#define EATING_TIME 2
#define THINKING_TIME 1

pthread_mutex_t forks[NUM_PHILOSOPHERS];

void *philosopher(void *arg) {
int id = *(int *)arg;

while (1) {
printf("Philosopher %d is thinking\n", id);
sleep(THINKING_TIME);

printf("Philosopher %d is hungry\n", id);
pthread_mutex_lock(&forks[LEFT]);
pthread_mutex_lock(&forks[id]);

printf("Philosopher %d is eating\n", id);
sleep(EATING_TIME);

pthread_mutex_unlock(&forks[id]);
pthread_mutex_unlock(&forks[LEFT]);
}

return NULL;
}

int main() {
pthread_t threads[NUM_PHILOSOPHERS];
int ids[NUM_PHILOSOPHERS];

// Initialize mutexes
for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
pthread_mutex_init(&forks[i], NULL);
}

// Create philosopher threads
for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
ids[i] = i;
if (pthread_create(&threads[i], NULL, philosopher, &ids[i]) != 0) {
perror("pthread_create");
exit(EXIT_FAILURE);
}
}

// Join philosopher threads
for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
if (pthread_join(threads[i], NULL) != 0) {
perror("pthread_join");
```

```
exit(EXIT_FAILURE);
}
}

// Destroy mutexes
for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
pthread_mutex_destroy(&forks[i]);
}

return 0;
}
```



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

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

#define MEMORY_SIZE 100

// Memory block structure
struct MemoryBlock {
int id;        // Process ID
int size;      // Size of the memory block
int allocated;  // Flag to indicate whether the block is allocated or free
};

// Function prototypes
void initializeMemory(struct MemoryBlock memory[], int size);
void printMemory(struct MemoryBlock memory[], int size);
void allocateFirstFit(struct MemoryBlock memory[], int size, int pid, int requestSize);
void allocateBestFit(struct MemoryBlock memory[], int size, int pid, int requestSize);
void allocateWorstFit(struct MemoryBlock memory[], int size, int pid, int requestSize);
```

```
int main() {
struct MemoryBlock memory[MEMORY_SIZE];
int choice, pid, requestSize;

// Initialize memory
initializeMemory(memory, MEMORY_SIZE);

// Menu
while (1) {
printf("\nMemory Allocation Strategies\n");
printf("1. First Fit\n");
printf("2. Best Fit\n");
printf("3. Worst Fit\n");
printf("4. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
case 1:
printf("Enter process ID and size: ");
scanf("%d %d", &pid, &requestSize);
allocateFirstFit(memory, MEMORY_SIZE, pid, requestSize);
break;
case 2:
printf("Enter process ID and size: ");
scanf("%d %d", &pid, &requestSize);
allocateBestFit(memory, MEMORY_SIZE, pid, requestSize);
break;
case 3:
printf("Enter process ID and size: ");
scanf("%d %d", &pid, &requestSize);
allocateWorstFit(memory, MEMORY_SIZE, pid, requestSize);
break;
case 4:
exit(EXIT_SUCCESS);
default:
printf("Invalid choice\n");
}

// Print memory after allocation
printMemory(memory, MEMORY_SIZE);
}

return 0;
}

// Initialize memory blocks as free
```

```
void initializeMemory(struct MemoryBlock memory[], int size) {
for (int i = 0; i < size; i++) {
memory[i].id = -1;
memory[i].size = 0;
memory[i].allocated = 0;
}
}

// Print memory blocks
void printMemory(struct MemoryBlock memory[], int size) {
printf("\nMemory Blocks:\n");
printf("%-8s %-8s %-8s %-12s\n", "Block", "ID", "Size", "Allocated");
for (int i = 0; i < size; i++) {
printf("%-8d %-8d %-8d %-12s\n", i, memory[i].id, memory[i].size,
memory[i].allocated ? "Allocated" : "Free");
}
}

// Allocate memory using First Fit strategy
void allocateFirstFit(struct MemoryBlock memory[], int size, int pid, int requestSize) {
int allocated = 0;
for (int i = 0; i < size; i++) {
if (!memory[i].allocated && memory[i].size >= requestSize) {
memory[i].id = pid;
memory[i].allocated = 1;
allocated = 1;
break;
}
}
if (!allocated)
printf("Memory allocation failed for process %d with size %d\n", pid, requestSize);
}

// Allocate memory using Best Fit strategy
void allocateBestFit(struct MemoryBlock memory[], int size, int pid, int requestSize) {
int bestFitIndex = -1;
int minFragmentation = INT_MAX;

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

```c
if (bestFitIndex != -1) {
memory[bestFitIndex].id = pid;
memory[bestFitIndex].allocated = 1;
} else {
printf("Memory allocation failed for process %d with size %d\n", pid, requestSize);
}
}

// Allocate memory using Worst Fit strategy
void allocateWorstFit(struct MemoryBlock memory[], int size, int pid, int requestSize) {
int worstFitIndex = -1;
int maxFragmentation = INT_MIN;

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

if (worstFitIndex != -1) {
memory[worstFitIndex].id = pid;
memory[worstFitIndex].allocated = 1;
} else {
printf("Memory allocation failed for process %d with size %d\n", pid, requestSize);
}
}
```

```
C:\Users\ravul\Downloads\co

Memory Allocation Strategies
1. First Fit
2. Best Fit
3. Worst Fit
4. Exit
Enter your choice: 1
Enter process ID and size: 200
2
Memory allocation failed for process 200 with size 2

Memory Blocks:
Block   ID      Size    Allocated
0       -1      0       Free
1       -1      0       Free
2       -1      0       Free
3       -1      0       Free
4       -1      0       Free
5       -1      0       Free
6       -1      0       Free
7       -1      0       Free
8       -1      0       Free
9       -1      0       Free
10      -1      0       Free
11      -1      0       Free
12      -1      0       Free
13      -1      0       Free
14      -1      0       Free
15      -1      0       Free
16      -1      0       Free
17      -1      0       Free
18      -1      0       Free
19      -1      0       Free
20      -1      0       Free
21      -1      0       Free
22      -1      0       Free
23      -1      0       Free
24      -1      0       Free
25      -1      0       Free
26      -1      0       Free
```

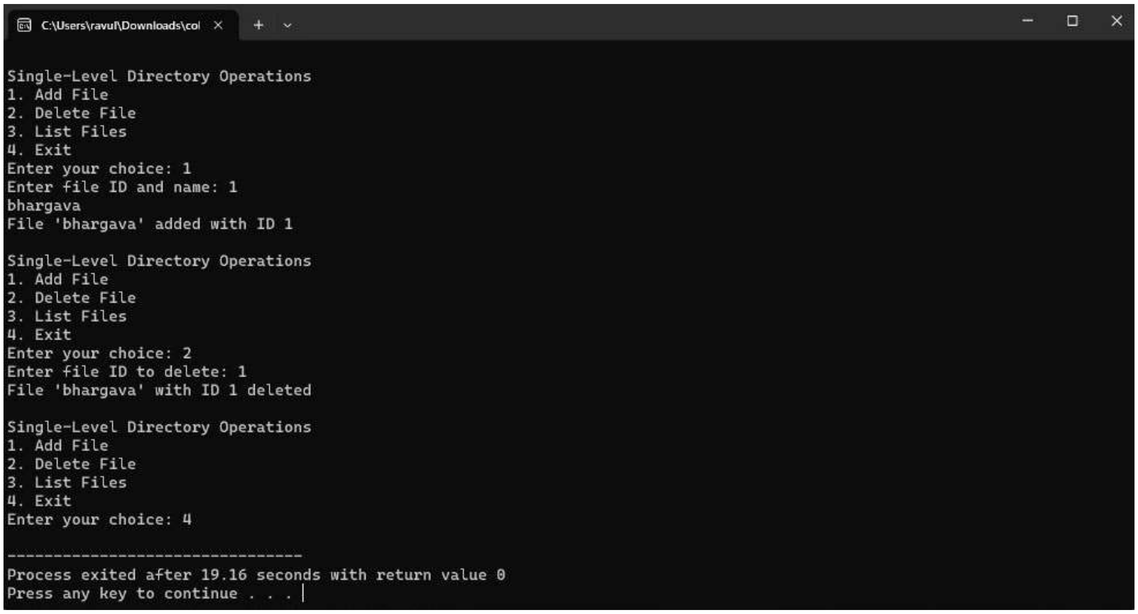
14. Construct a C program to organize the file using single level directory.

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

#define MAX_FILENAME_LENGTH 50

// File structure
struct File {
int id;
char name[MAX_FILENAME_LENGTH];
};

// Directory structure
struct Directory {
struct File *files;
int fileCount;
int maxSize;
};

// Function prototypes
void initializeDirectory(struct Directory *dir, int maxSize);
void addFile(struct Directory *dir, int fileId, const char *fileName);
void deleteFile(struct Directory *dir, int fileId);
void listFiles(const struct Directory *dir);
void clearInputBuffer();

int main() {
struct Directory dir;
int choice, fileId;
char fileName[MAX_FILENAME_LENGTH];

// Initialize directory with a maximum capacity of 100 files
initializeDirectory(&dir, 100);

// Menu
while (1) {
printf("\nSingle-Level Directory Operations\n");
printf("1. Add File\n");
printf("2. Delete File\n");
printf("3. List Files\n");
printf("4. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
clearInputBuffer(); // Clear input buffer

switch (choice) {
case 1:
```

```c
printf("Enter file ID and name: ");
scanf("%d %s", &fileId, fileName);
clearInputBuffer(); // Clear input buffer
addFile(&dir, fileId, fileName);
break;
case 2:
printf("Enter file ID to delete: ");
scanf("%d", &fileId);
clearInputBuffer(); // Clear input buffer
deleteFile(&dir, fileId);
break;
case 3:
listFiles(&dir);
break;
case 4:
// Free dynamically allocated memory before exiting
free(dir.files);
exit(EXIT_SUCCESS);
default:
printf("Invalid choice\n");
clearInputBuffer(); // Clear input buffer
}
}

return 0;
}

// Initialize directory
void initializeDirectory(struct Directory *dir, int maxSize) {
dir->fileCount = 0;
dir->maxSize = maxSize;
dir->files = (struct File *)malloc(maxSize * sizeof(struct File));
if (dir->files == NULL) {
printf("Memory allocation failed\n");
exit(EXIT_FAILURE);
}
}

// Add file to directory
void addFile(struct Directory *dir, int fileId, const char *fileName) {
if (dir->fileCount < dir->maxSize) {
struct File *file = &(dir->files[dir->fileCount]);
file->id = fileId;
strncpy(file->name, fileName, MAX_FILENAME_LENGTH - 1);
file->name[MAX_FILENAME_LENGTH - 1] = '\0'; // Ensure null-terminated string
printf("File '%s' added with ID %d\n", fileName, fileId);
dir->fileCount++;
} else {
```

```
printf("Cannot add file. Directory is full.\n");
}
}

// Delete file from directory
void deleteFile(struct Directory *dir, int fileId) {
int i, found = 0;
for (i = 0; i < dir->fileCount; i++) {
if (dir->files[i].id == fileId) {
printf("File '%s' with ID %d deleted\n", dir->files[i].name, fileId);
found = 1;
break;
}
}

if (found) {
for (int j = i; j < dir->fileCount - 1; j++) {
dir->files[j] = dir->files[j + 1];
}
dir->fileCount--;
} else {
printf("File with ID %d not found\n", fileId);
}
}

// List all files in directory
void listFiles(const struct Directory *dir) {
if (dir->fileCount == 0) {
printf("No files in directory\n");
} else {
printf("Files in Directory:\n");
for (int i = 0; i < dir->fileCount; i++) {
printf("File ID: %d, Name: %s\n", dir->files[i].id, dir->files[i].name);
}
}
}

// Function to clear input buffer
void clearInputBuffer() {
int c;
while ((c = getchar()) != '\n' && c != EOF);
```

}



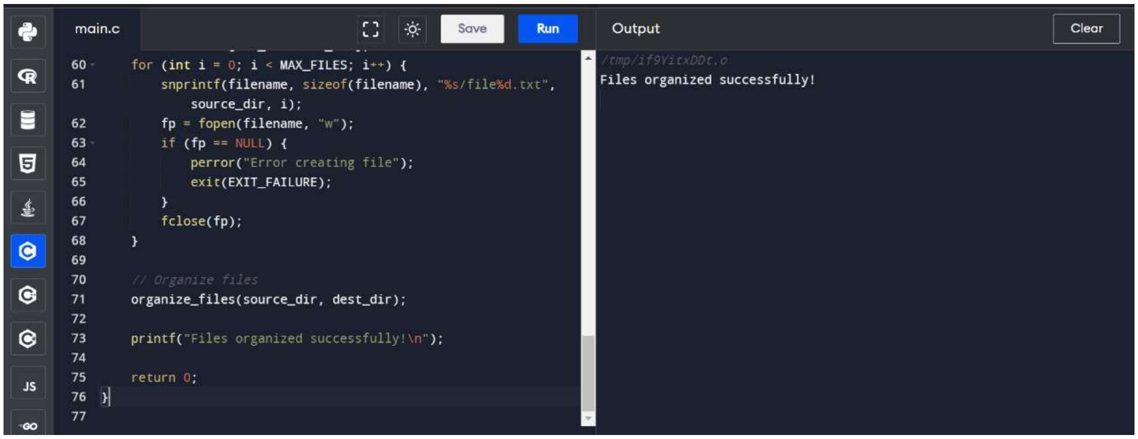
15. Design a C program to organize the file using two level directory structure.

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

#define MAX_FILENAME_LEN 256
#define MAX_FILES 100

void create_directory(const char *path) {
mkdir(path, 0777); // Creates a directory with full permissions
}

void organize_files(const char *source_dir, const char *dest_dir) {
// Create level1 directory
create_directory(dest_dir);

// Create level2 directories
for (int i = 0; i < 10; i++) {
char level2_dir[MAX_FILENAME_LEN];
snprintf(level2_dir, sizeof(level2_dir), "%s/level%d", dest_dir, i);
create_directory(level2_dir);
}

// Traverse source directory
DIR *dir;
```

```
struct dirent *entry;
dir = opendir(source_dir);
if (dir == NULL) {
perror("Error opening directory");
exit(EXIT_FAILURE);
}

// Process files
while ((entry = readdir(dir)) != NULL) {
if (entry->d_type == DT_REG) { // If it's a regular file
char source_file[MAX_FILENAME_LEN];
snprintf(source_file, sizeof(source_file), "%s/%s", source_dir, entry->d_name);
char dest_file[MAX_FILENAME_LEN];
snprintf(dest_file, sizeof(dest_file), "%s/level%d/%s", dest_dir, (entry->d_name[4] - '0'),
entry->d_name); // Changed (entry->d_name[0] - '0') to (entry->d_name[4] - '0') to extract
the file number correctly
rename(source_file, dest_file); // Move file to appropriate level2 directory
}
}

closedir(dir);
}

int main() {
const char *source_dir = "source";
const char *dest_dir = "level1";

// Create source directory
create_directory(source_dir);

// Simulate files in the source directory
FILE *fp;
char filename[MAX_FILENAME_LEN];
for (int i = 0; i < MAX_FILES; i++) {
snprintf(filename, sizeof(filename), "%s/file%d.txt", source_dir, i);
fp = fopen(filename, "w");
if (fp == NULL) {
perror("Error creating file");
exit(EXIT_FAILURE);
}
fclose(fp);
}

// Organize files
organize_files(source_dir, dest_dir);

printf("Files organized successfully!\n");
```

```
return 0;
}
```

THIS QUESTION CAN YOU USE THE ONLINE COMPILER



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

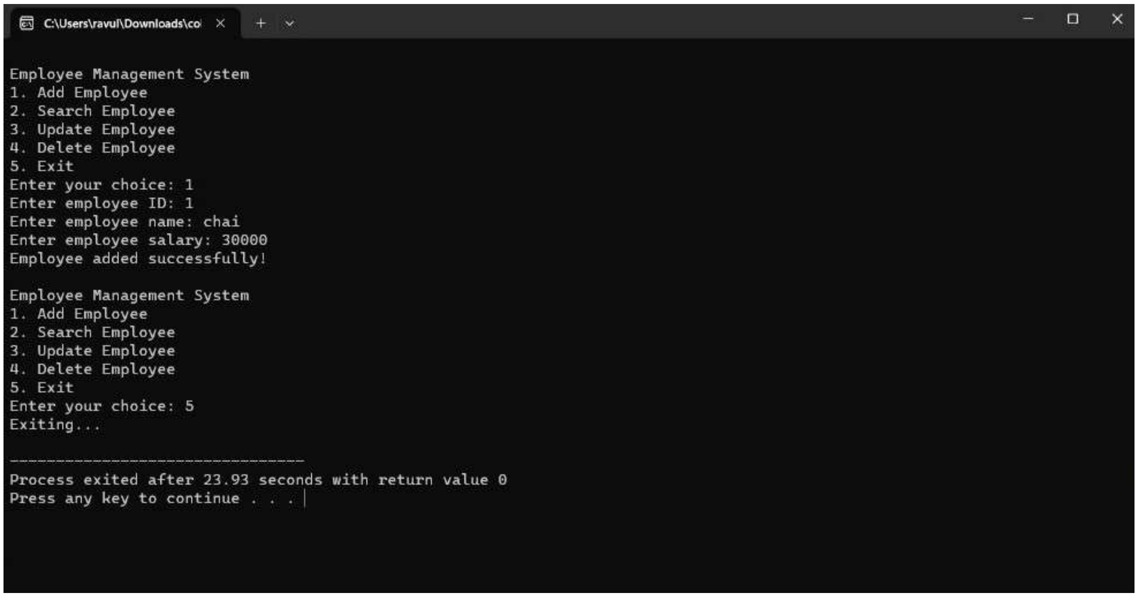
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_EMPLOYEES 100
#define MAX_NAME_LENGTH 50
#define FILENAME "employees.dat"

// Structure to represent an employee
struct Employee {
int id;
char name[MAX_NAME_LENGTH];
float salary;
};

// Function to add a new employee record to the file
void addEmployee() {
FILE *file = fopen(FILENAME, "ab");
if (file == NULL) {
perror("Error opening file");
exit(EXIT_FAILURE);
}

struct Employee emp;

printf("Enter employee ID: ");
scanf("%d", &emp.id);
printf("Enter employee name: ");
scanf("%s", emp.name);
```

```c
printf("Enter employee salary: ");
scanf("%f", &emp.salary);

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

fclose(file);

printf("Employee added successfully!\n");
}

// Function to search for an employee record by ID
void searchEmployee() {
FILE *file = fopen(FILENAME, "rb");
if (file == NULL) {
perror("Error opening file");
exit(EXIT_FAILURE);
}

int targetId;
printf("Enter employee ID to search: ");
scanf("%d", &targetId);

struct Employee emp;
int found = 0;
while (fread(&emp, sizeof(struct Employee), 1, file) == 1) {
if (emp.id == targetId) {
found = 1;
printf("Employee found!\n");
printf("ID: %d\n", emp.id);
printf("Name: %s\n", emp.name);
printf("Salary: %.2f\n", emp.salary);
break;
}
}

if (!found) {
printf("Employee not found!\n");
}

fclose(file);
}

// Function to update an existing employee record
void updateEmployee() {
FILE *file = fopen(FILENAME, "r+b");
if (file == NULL) {
perror("Error opening file");
exit(EXIT_FAILURE);
```

```c
}

int targetId;
printf("Enter employee ID to update: ");
scanf("%d", &targetId);

struct Employee emp;
int found = 0;
while (fread(&emp, sizeof(struct Employee), 1, file) == 1) {
if (emp.id == targetId) {
found = 1;
printf("Enter new name: ");
scanf("%s", emp.name);
printf("Enter new salary: ");
scanf("%f", &emp.salary);

fseek(file, -sizeof(struct Employee), SEEK_CUR);
fwrite(&emp, sizeof(struct Employee), 1, file);
printf("Employee updated successfully!\n");
break;
}
}

if (!found) {
printf("Employee not found!\n");
}

fclose(file);
}

// Function to delete an existing employee record
void deleteEmployee() {
FILE *file = fopen(FILENAME, "r+b");
if (file == NULL) {
perror("Error opening file");
exit(EXIT_FAILURE);
}

int targetId;
printf("Enter employee ID to delete: ");
scanf("%d", &targetId);

struct Employee emp;
int found = 0;
while (fread(&emp, sizeof(struct Employee), 1, file) == 1) {
if (emp.id == targetId) {
found = 1;
emp.id = -1; // Marking as deleted
```

```
fseek(file, -sizeof(struct Employee), SEEK_CUR);
fwrite(&emp, sizeof(struct Employee), 1, file);
printf("Employee deleted successfully!\n");
break;
}
}

if (!found) {
printf("Employee not found!\n");
}

fclose(file);
}

// Main function to demonstrate the functionality
int main() {
int choice;
do {
printf("\nEmployee Management System\n");
printf("1. Add Employee\n");
printf("2. Search Employee\n");
printf("3. Update Employee\n");
printf("4. Delete Employee\n");
printf("5. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
case 1:
addEmployee();
break;
case 2:
searchEmployee();
break;
case 3:
updateEmployee();
break;
case 4:
deleteEmployee();
break;
case 5:
printf("Exiting...\n");
break;
default:
printf("Invalid choice! Please try again.\n");
}
} while (choice != 5);
```

```
return 0;
}
```

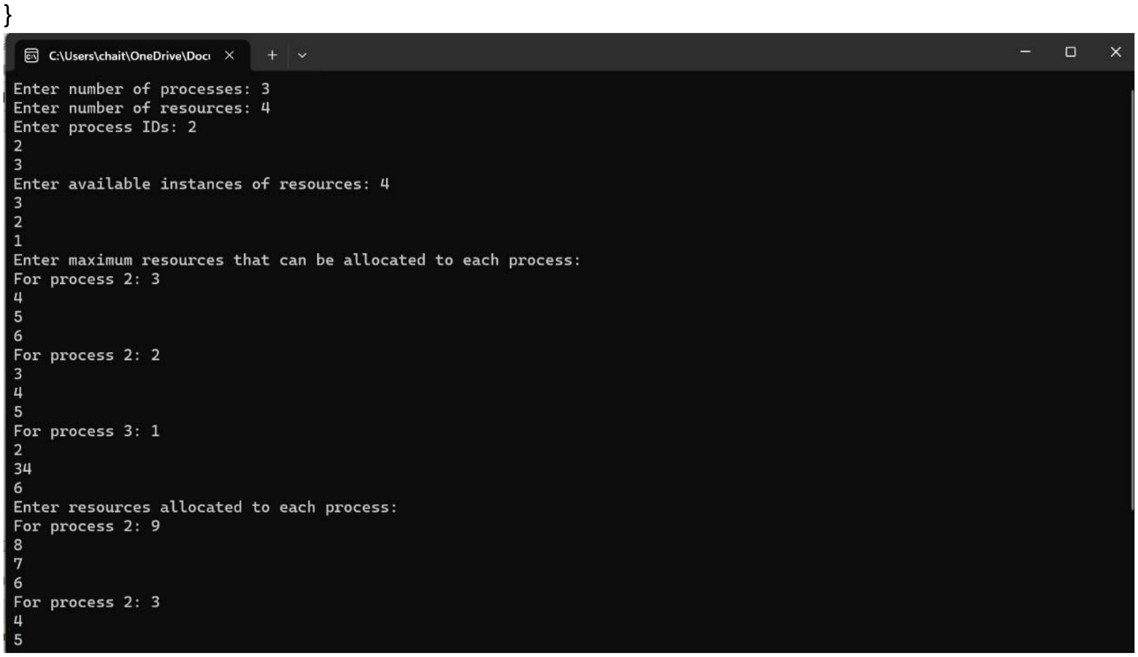


17. Illustrate the deadlock avoidance concept by simulating Banker's algorithm with C.

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

// Define the maximum number of processes and resources
#define MAX_P 10
#define MAX_R 10

// Function to calculate the need matrix
void calculateNeed(int need[MAX_P][MAX_R], int max[MAX_P][MAX_R], int
alloc[MAX_P][MAX_R], int P, int R) {
for (int i = 0; i < P; i++)
for (int j = 0; j < R; j++)
need[i][j] = max[i][j] - alloc[i][j];
}

// Function to check if the requested resources are available
bool isSafe(int processes[], int avail[], int max[][MAX_R], int alloc[][MAX_R], int P, int R) {
int need[MAX_P][MAX_R];
calculateNeed(need, max, alloc, P, R);

bool finish[MAX_P] = {0};
int safeSeq[MAX_P];
int work[MAX_R];
for (int i = 0; i < R; i++)
work[i] = avail[i];

int count = 0;
while (count < P) {
```

```
bool found = false;
for (int p = 0; p < P; p++) {
if (!finish[p]) {
int j;
for (j = 0; j < R; j++)
if (need[p][j] > work[j])
break;
if (j == R) {
for (int k = 0; k < R; k++)
work[k] += alloc[p][k];
safeSeq[count++] = p;
finish[p] = true;
found = true;
}
}
}
if (!found) {
printf("System is not in safe state");
return false;
}
}

printf("System is in safe state.\nSafe sequence is: ");
for (int i = 0; i < P; i++)
printf("%d ", safeSeq[i]);
printf("\n");
return true;
}

// Main function
int main() {
int P, R;
printf("Enter number of processes: ");
scanf("%d", &P);
printf("Enter number of resources: ");
scanf("%d", &R);

int processes[MAX_P];
printf("Enter process IDs: ");
for (int i = 0; i < P; i++)
scanf("%d", &processes[i]);

// Available instances of resources
int available[MAX_R];
printf("Enter available instances of resources: ");
for (int i = 0; i < R; i++)
scanf("%d", &available[i]);
```

```
// Maximum R that can be allocated to processes
int max[MAX_P][MAX_R];
printf("Enter maximum resources that can be allocated to each process:\n");
for (int i = 0; i < P; i++) {
printf("For process %d: ", processes[i]);
for (int j = 0; j < R; j++)
scanf("%d", &max[i][j]);
}

// Resources allocated to processes
int allocation[MAX_P][MAX_R];
printf("Enter resources allocated to each process:\n");
for (int i = 0; i < P; i++) {
printf("For process %d: ", processes[i]);
for (int j = 0; j < R; j++)
scanf("%d", &allocation[i][j]);
}

// Check if the system is in safe state or not
isSafe(processes, available, max, allocation, P, R);

return 0;
}
```

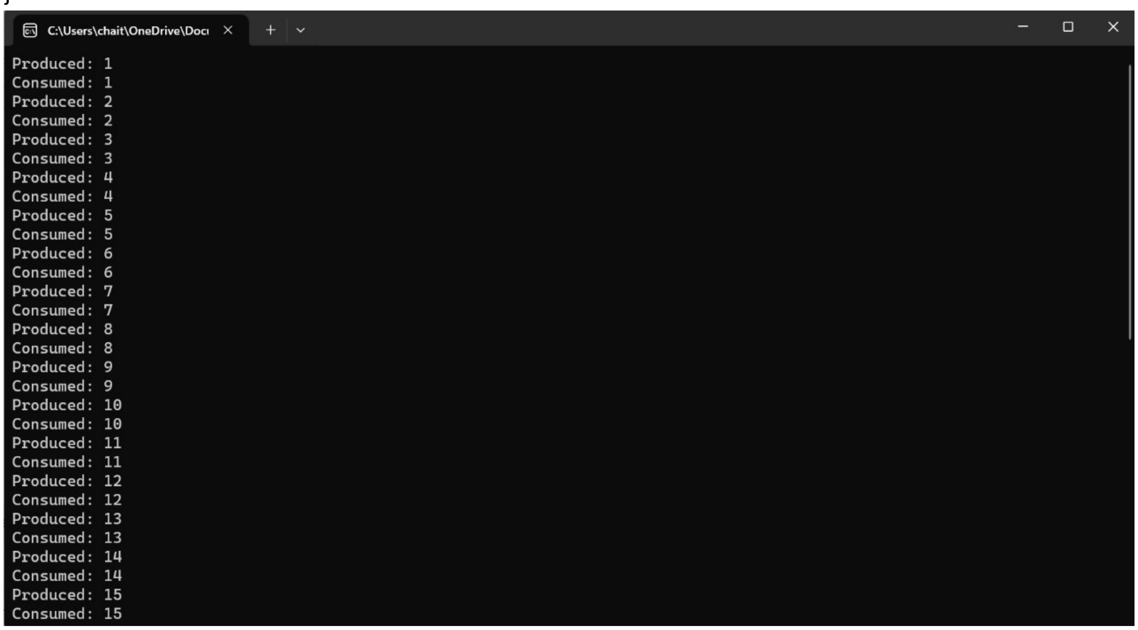


18. Construct a C program to simulate producer-consumer problem using semaphores.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h> // for usleep function

#define BUFFER_SIZE 5
```

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

void *producer(void *arg) {
int item = 1;
while (1) {
sem_wait(&empty); // Wait for an empty slot in the buffer
pthread_mutex_lock(&mutex);

// Produce item
buffer[in] = item;
printf("Produced: %d\n", item);
item++;
in = (in + 1) % BUFFER_SIZE;

pthread_mutex_unlock(&mutex);
sem_post(&full); // Increment the count of full slots

usleep(500000); // Sleep for 500 milliseconds
}
pthread_exit(NULL);
}

void *consumer(void *arg) {
while (1) {
sem_wait(&full); // Wait for a full slot in the buffer
pthread_mutex_lock(&mutex);

// Consume item
int item = buffer[out];
printf("Consumed: %d\n", item);
out = (out + 1) % BUFFER_SIZE;

pthread_mutex_unlock(&mutex);
sem_post(&empty); // Increment the count of empty slots

usleep(500000); // Sleep for 500 milliseconds
}
pthread_exit(NULL);
}

int main() {
pthread_t producer_thread, consumer_thread;
pthread_mutex_init(&mutex, NULL);
sem_init(&empty, 0, BUFFER_SIZE); // Initialize empty semaphore with buffer size
sem_init(&full, 0, 0); // Initialize full semaphore with 0
```

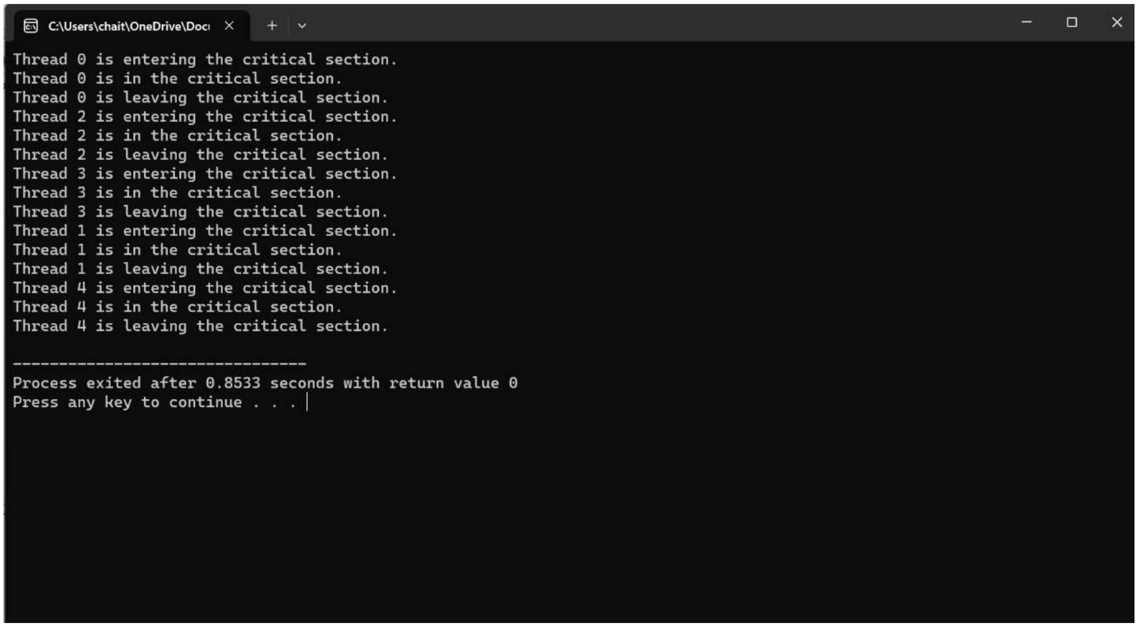
```
// Create producer and consumer threads
pthread_create(&producer_thread, NULL, producer, NULL);
pthread_create(&consumer_thread, NULL, consumer, NULL);

// Wait for threads to finish (which will never happen, but for demonstration purpose)
pthread_join(producer_thread, NULL);
pthread_join(consumer_thread, NULL);

// Destroy semaphores and mutex
sem_destroy(&empty);
sem_destroy(&full);
pthread_mutex_destroy(&mutex);

return 0;
}
```



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

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

#define NUM_THREADS 5

pthread_mutex_t mutex;

void *thread_function(void *thread_id) {
int tid = *((int*)thread_id);

// Lock the mutex before accessing shared resources
pthread_mutex_lock(&mutex);

// Critical section
```

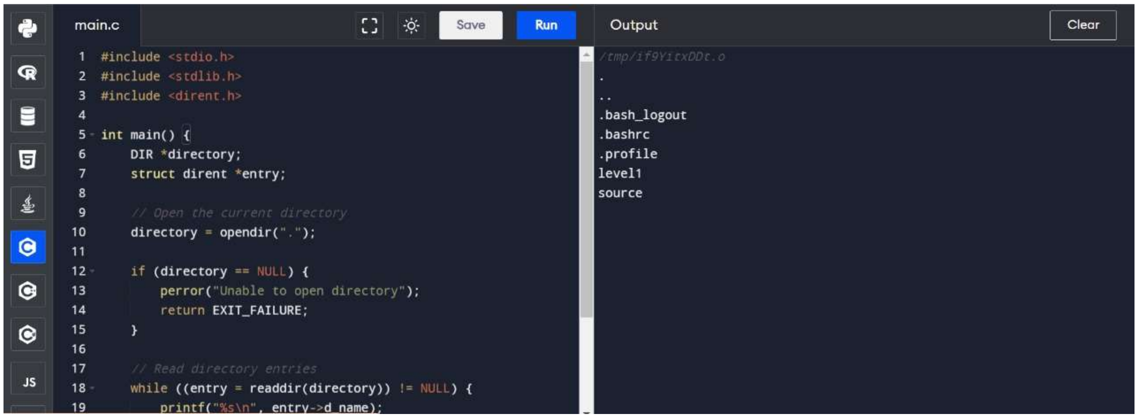
```
printf("Thread %d is entering the critical section.\n", tid);
printf("Thread %d is in the critical section.\n", tid);
printf("Thread %d is leaving the critical section.\n", tid);

// Unlock the mutex after accessing shared resources
pthread_mutex_unlock(&mutex);

pthread_exit(NULL);
}

int main() {
pthread_t threads[NUM_THREADS];
int thread_ids[NUM_THREADS];

// Initialize the mutex
pthread_mutex_init(&mutex, NULL);

// Create threads
for (int i = 0; i < NUM_THREADS; ++i) {
thread_ids[i] = i;
pthread_create(&threads[i], NULL, thread_function, (void *)&thread_ids[i]);
}

// Join threads
for (int i = 0; i < NUM_THREADS; ++i) {
pthread_join(threads[i], NULL);
}

// Destroy the mutex
pthread_mutex_destroy(&mutex);

return 0;
}
```



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

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

int main() {
DIR *directory;
struct dirent *entry;

// Open the current directory
directory = opendir(".");

if (directory == NULL) {
perror("Unable to open directory");
return EXIT_FAILURE;
}

// Read directory entries
while ((entry = readdir(directory)) != NULL) {
printf("%s\n", entry->d_name);
}

// Close the directory
closedir(directory);

return EXIT_SUCCESS;
}
```

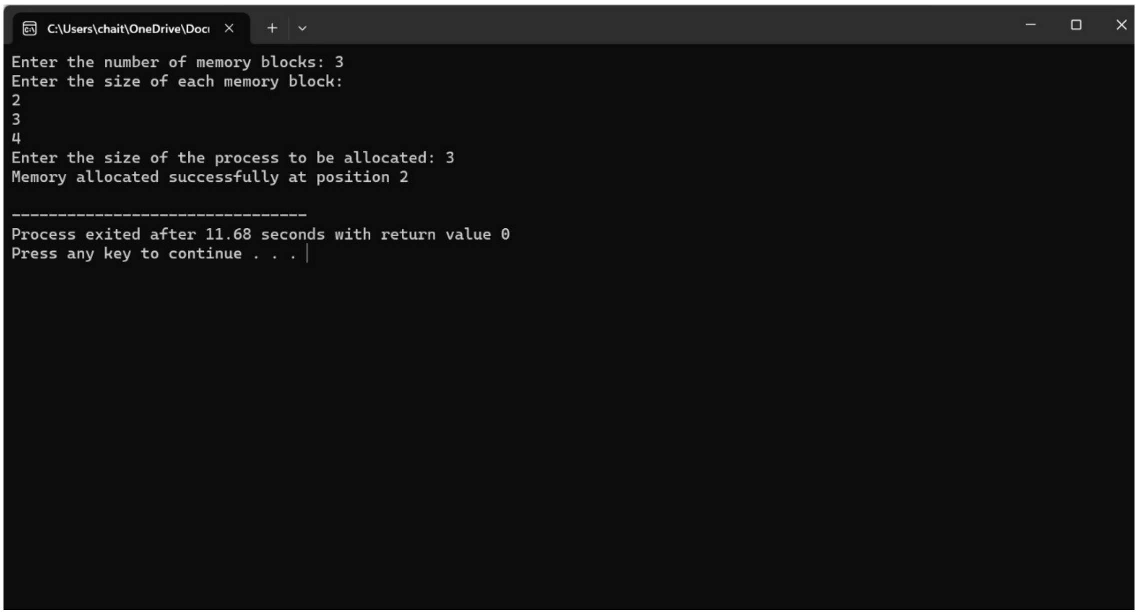
THIS QUESTION CAN YOU USE THE ONLINE COMPILER



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

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

#define MAX_MEMORY 100

struct memory_block {
int size;
int allocated;
};

void worstFit(struct memory_block mem[], int n, int process_size) {
int i, worstFitIdx = -1;
for (i = 0; i < n; i++) {
if (!mem[i].allocated && mem[i].size >= process_size) {
if (worstFitIdx == -1 || mem[i].size > mem[worstFitIdx].size) {
worstFitIdx = i;
}
}
}
if (worstFitIdx != -1) {
mem[worstFitIdx].allocated = 1;
printf("Memory allocated successfully at position %d\n", worstFitIdx);
} else {
printf("No memory block available for allocation\n");
}
}

int main() {
int n, i, process_size;
printf("Enter the number of memory blocks: ");
scanf("%d", &n);

struct memory_block mem[MAX_MEMORY];
```

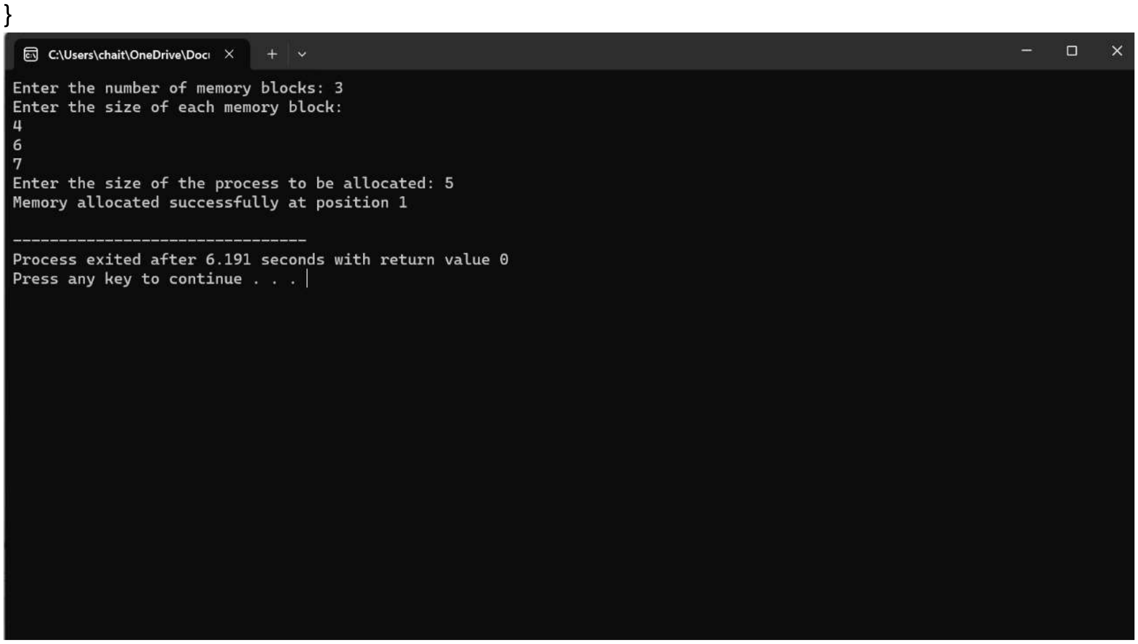
```
printf("Enter the size of each memory block:\n");
for (i = 0; i < n; i++) {
scanf("%d", &mem[i].size);
mem[i].allocated = 0;
}

printf("Enter the size of the process to be allocated: ");
scanf("%d", &process_size);

worstFit(mem, n, process_size);

return 0;
}
```



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

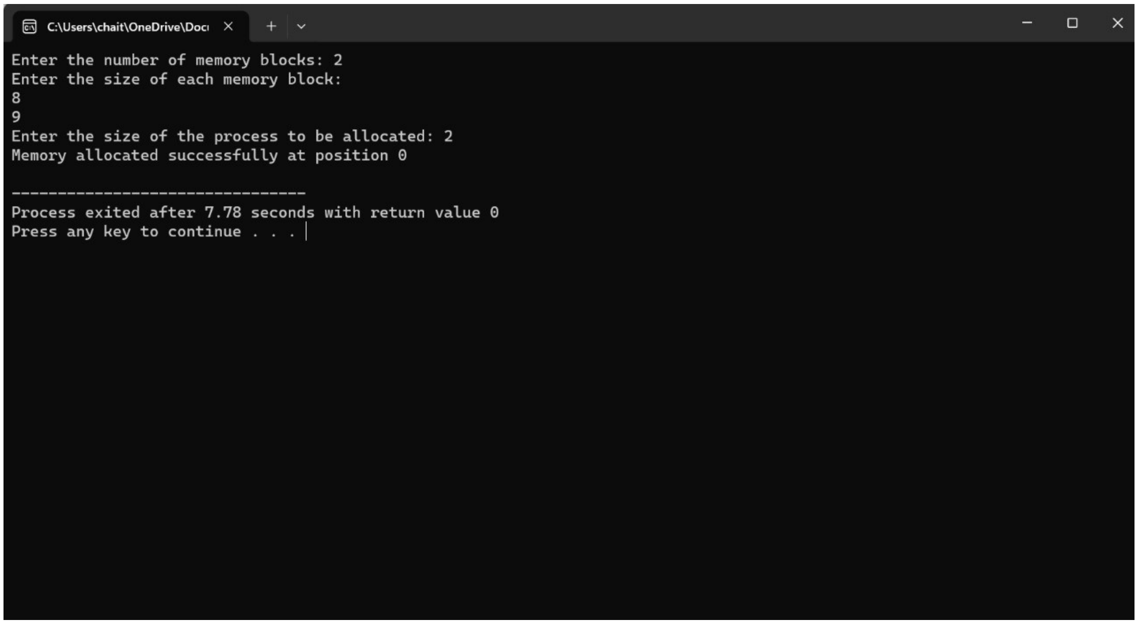
```
#include <stdio.h>
#include <stdlib.h>

#define MAX_MEMORY 100

struct memory_block {
int size;
int allocated;
};

void bestFit(struct memory_block mem[], int n, int process_size) {
int i, bestFitIdx = -1;
for (i = 0; i < n; i++) {
if (!mem[i].allocated && mem[i].size >= process_size) {
if (bestFitIdx == -1 || mem[i].size < mem[bestFitIdx].size) {
bestFitIdx = i;
}
```

```
}
}
if (bestFitIdx != -1) {
mem[bestFitIdx].allocated = 1;
printf("Memory allocated successfully at position %d\n", bestFitIdx);
} else {
printf("No memory block available for allocation\n");
}
}

int main() {
int n, i, process_size;
printf("Enter the number of memory blocks: ");
scanf("%d", &n);

struct memory_block mem[MAX_MEMORY];

printf("Enter the size of each memory block:\n");
for (i = 0; i < n; i++) {
scanf("%d", &mem[i].size);
mem[i].allocated = 0;
}

printf("Enter the size of the process to be allocated: ");
scanf("%d", &process_size);

bestFit(mem, n, process_size);

return 0;
}
```


```
C:\Users\chait\OneDrive\Doc
Enter the number of memory blocks: 3
Enter the size of each memory block:
4
6
7
Enter the size of the process to be allocated: 5
Memory allocated successfully at position 1

--------------------------------
Process exited after 6.191 seconds with return value 0
Press any key to continue . . .
```

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

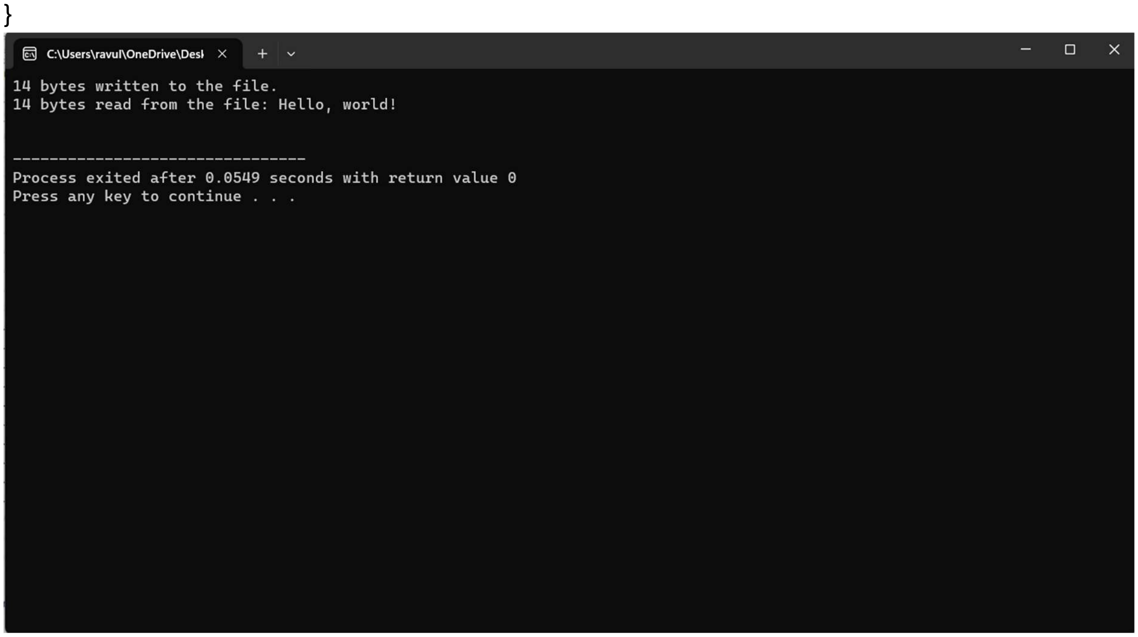
```
#include <stdio.h>
#include <stdlib.h>

#define MAX_MEMORY 100

struct memory_block {
int size;
int allocated;
};

void firstFit(struct memory_block mem[], int n, int process_size) {
int i;
for (i = 0; i < n; i++) {
if (!mem[i].allocated && mem[i].size >= process_size) {
mem[i].allocated = 1;
printf("Memory allocated successfully at position %d\n", i);
return;
}
}
printf("No memory block available for allocation\n");
}

int main() {
int n, i, process_size;
printf("Enter the number of memory blocks: ");
scanf("%d", &n);

struct memory_block mem[MAX_MEMORY];

printf("Enter the size of each memory block:\n");
for (i = 0; i < n; i++) {
scanf("%d", &mem[i].size);
mem[i].allocated = 0;
}

printf("Enter the size of the process to be allocated: ");
scanf("%d", &process_size);

firstFit(mem, n, process_size);

return 0;
}
```



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

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

#define BUFFER_SIZE 1024

int main() {
int fd; // File descriptor
ssize_t bytes_written, bytes_read;
char buffer[BUFFER_SIZE];
const char *file_path = "C:/Users/ravul/Downloads/collage detiles/R.collage/operating system sem-6/os lab/DAY3/Q24/Q24.cpp"; // Use forward slashes and escape backslashes

// Open a file (create if it doesn't exist, truncate if it does)
fd = open(file_path, O_WRONLY | O_CREAT | O_TRUNC, 0644); // Specify permissions manually
if (fd == -1) {
perror("open");
exit(EXIT_FAILURE);
}

// Write data to the file
const char *data_to_write = "Hello, world!\n";
bytes_written = write(fd, data_to_write, strlen(data_to_write));
if (bytes_written == -1) {
perror("write");
close(fd);
```

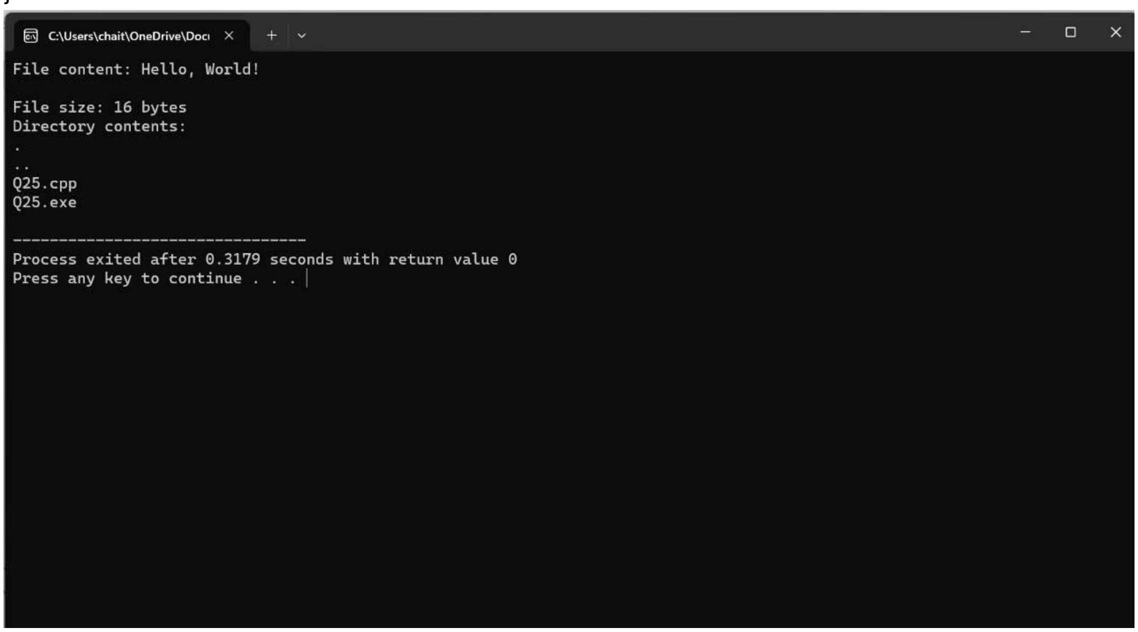
```
exit(EXIT_FAILURE);
}
printf("%ld bytes written to the file.\n", bytes_written);

// Close the file
close(fd);

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

// Read data from the file
bytes_read = read(fd, buffer, BUFFER_SIZE);
if (bytes_read == -1) {
perror("read");
close(fd);
exit(EXIT_FAILURE);
}
// Null-terminate the buffer to treat it as a string
buffer[bytes_read] = '\0';
printf("%ld bytes read from the file: %s\n", bytes_read, buffer);

// Close the file
close(fd);

return 0;
}
```



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

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

int main() {
// Open file using open
int fd = open("C:\\Users\\chait\\Downloads\\sse.txt", O_RDWR | O_CREAT, S_IRUSR |
S_IWUSR);
if (fd == -1) {
perror("Error opening file");
exit(EXIT_FAILURE);
}

// Write to file
char *text = "Hello, World!\n";
write(fd, text, strlen(text));

// Move file pointer using lseek
off_t offset = lseek(fd, 0, SEEK_SET);
if (offset == -1) {
perror("Error seeking file");
exit(EXIT_FAILURE);
}

// Read from file
char buffer[100];
ssize_t bytes_read = read(fd, buffer, sizeof(buffer));
if (bytes_read == -1) {
perror("Error reading file");
exit(EXIT_FAILURE);
}
buffer[bytes_read] = '\0';
printf("File content: %s", buffer);

// Get file information using stat
struct stat file_info;
if (fstat(fd, &file_info) == -1) {
perror("Error getting file info");
exit(EXIT_FAILURE);
}
printf("File size: %lld bytes\n", (long long)file_info.st_size);
```

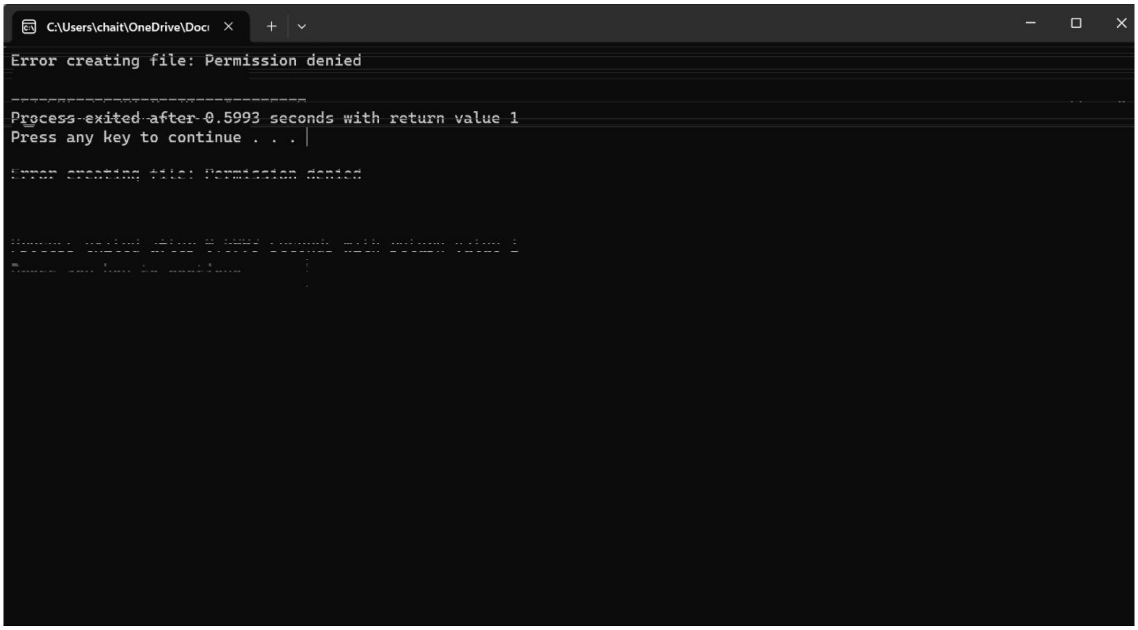
```
// Open directory using opendir
DIR *dir = opendir(".");
if (dir == NULL) {
perror("Error opening directory");
exit(EXIT_FAILURE);
}

// Read directory entries using readdir
struct dirent *entry;
printf("Directory contents:\n");
while ((entry = readdir(dir)) != NULL) {
printf("%s\n", entry->d_name);
}

// Close file and directory
close(fd);
closedir(dir);

return 0;
}
```



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

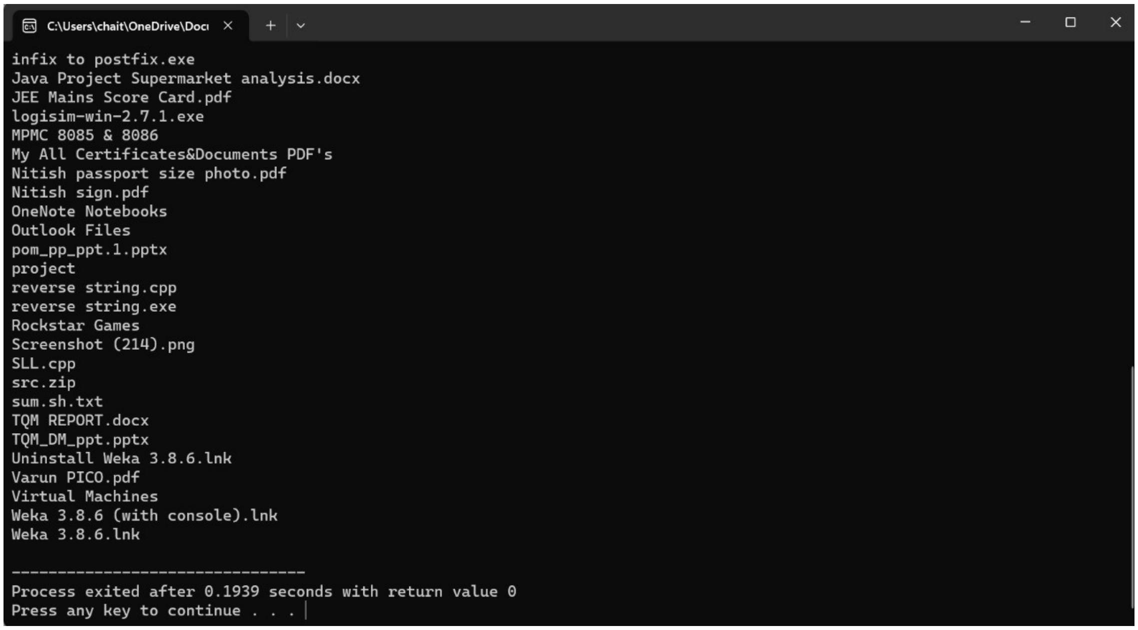
```
#include <stdio.h>
#include <stdlib.h>

int main() {
FILE *file;
char filename[] = "C:\\Users\\chait\\OneDrive\\Documents\\Day 4\\Q32";
char buffer[100];

// Create a file
```

```
file = fopen(filename, "w");
if (file == NULL) {
perror("Error creating file");
exit(EXIT_FAILURE);
}
printf("File created successfully.\n");

// Write to the file
fprintf(file, "Hello, world!\n");
printf("Data written to file.\n");

// Close the file
fclose(file);
printf("File closed.\n");

// Open the file for reading
file = fopen(filename, "r");
if (file == NULL) {
perror("Error opening file for reading");
exit(EXIT_FAILURE);
}
printf("File opened for reading.\n");

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

// Close the file
fclose(file);
printf("File closed.\n");

return 0;
}
```



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

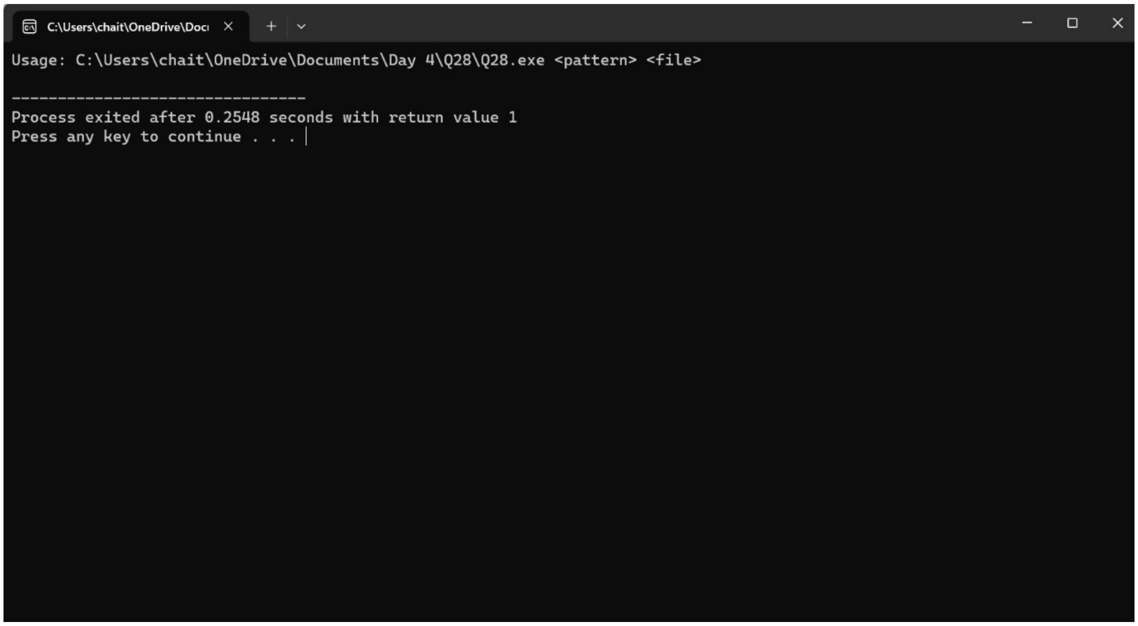
```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>

int main() {
DIR *directory;
struct dirent *entry;

// Open the current directory
directory = opendir("C:\\Users\\chait\\OneDrive\\Documents");

if (directory == NULL) {
perror("Unable to open directory");
return EXIT_FAILURE;
}

// Read directory entries
while ((entry = readdir(directory)) != NULL) {
printf("%s\n", entry->d_name);
}

// Close the directory
closedir(directory);

return EXIT_SUCCESS;
}
```



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

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

#define MAX_LINE_LENGTH 1024

int main(int argc, char *argv[]) {
if (argc != 3) {
fprintf(stderr, "Usage: %s <pattern> <file>\n", argv[0]);
return EXIT_FAILURE;
}

const char *pattern = argv[1];
const char *filename = argv[2];
FILE *file = fopen(filename, "r");

if (file == NULL) {
perror("Error opening file");
return EXIT_FAILURE;
}

char line[MAX_LINE_LENGTH];
while (fgets(line, MAX_LINE_LENGTH, file) != NULL) {
if (strstr(line, pattern) != NULL) {
printf("%s", line);
}
}

fclose(file);
return EXIT_SUCCESS;
```

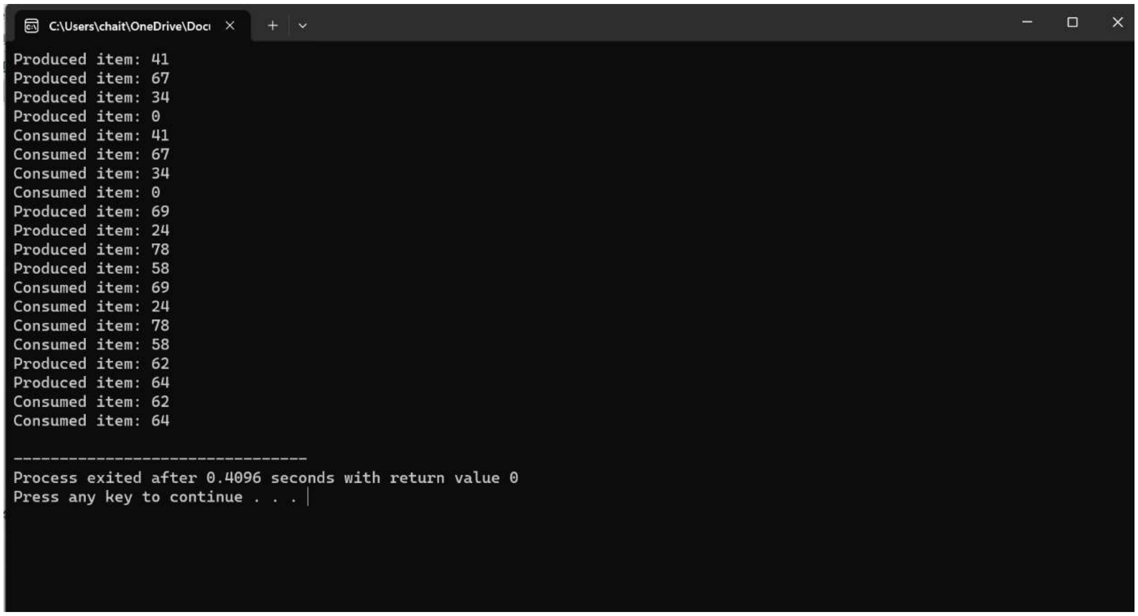
```
}
```



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

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

#define BUFFER_SIZE 5
#define NUM_ITEMS 10

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;

pthread_mutex_t mutex;
pthread_cond_t full, empty;

void *producer(void *arg) {
int item;
for (int i = 0; i < NUM_ITEMS; ++i) {
item = rand() % 100; // Generate a random item
pthread_mutex_lock(&mutex);
while (((in + 1) % BUFFER_SIZE) == out) // Buffer is full
pthread_cond_wait(&empty, &mutex);

buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
printf("Produced item: %d\n", item);

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

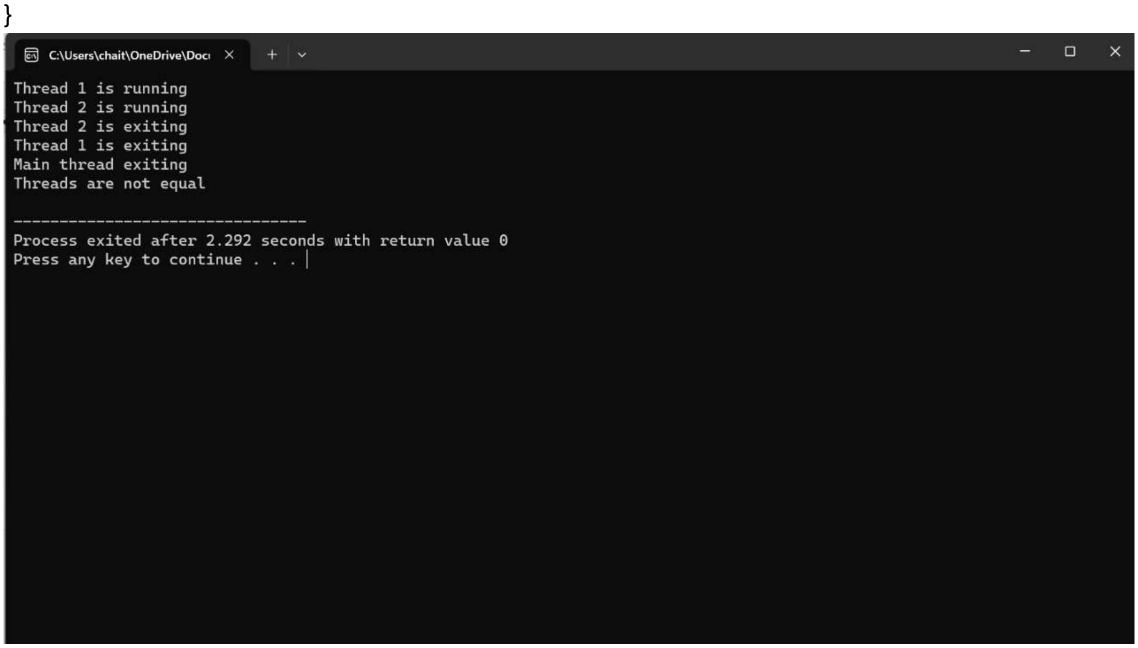
```
}
pthread_exit(NULL);
}

void *consumer(void *arg) {
int item;
for (int i = 0; i < NUM_ITEMS; ++i) {
pthread_mutex_lock(&mutex);
while (in == out) // Buffer is empty
pthread_cond_wait(&full, &mutex);

item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
printf("Consumed item: %d\n", item);

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

int main() {
pthread_t producer_thread, consumer_thread;

pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&full, NULL);
pthread_cond_init(&empty, NULL);

pthread_create(&producer_thread, NULL, producer, NULL);
pthread_create(&consumer_thread, NULL, consumer, NULL);

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

pthread_mutex_destroy(&mutex);
pthread_cond_destroy(&full);
pthread_cond_destroy(&empty);

return 0;
}
```



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

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

void *thread_function(void *arg) {
int thread_id = *((int *)arg);
printf("Thread %d is running\n", thread_id);
sleep(2);
printf("Thread %d is exiting\n", thread_id);
pthread_exit(NULL);
}

int main() {
pthread_t thread1, thread2;
int id1 = 1, id2 = 2;

// Create thread 1
if (pthread_create(&thread1, NULL, thread_function, &id1) != 0) {
perror("Error creating thread 1");
exit(EXIT_FAILURE);
}

// Create thread 2
if (pthread_create(&thread2, NULL, thread_function, &id2) != 0) {
perror("Error creating thread 2");
exit(EXIT_FAILURE);
}
```

```
// Join thread 1
if (pthread_join(thread1, NULL) != 0) {
perror("Error joining thread 1");
exit(EXIT_FAILURE);
}

// Join thread 2
if (pthread_join(thread2, NULL) != 0) {
perror("Error joining thread 2");
exit(EXIT_FAILURE);
}

printf("Main thread exiting\n");

// Check if threads are equal
if (pthread_equal(thread1, thread2)) {
printf("Threads are equal\n");
} else {
printf("Threads are not equal\n");
}

return 0;
}
```

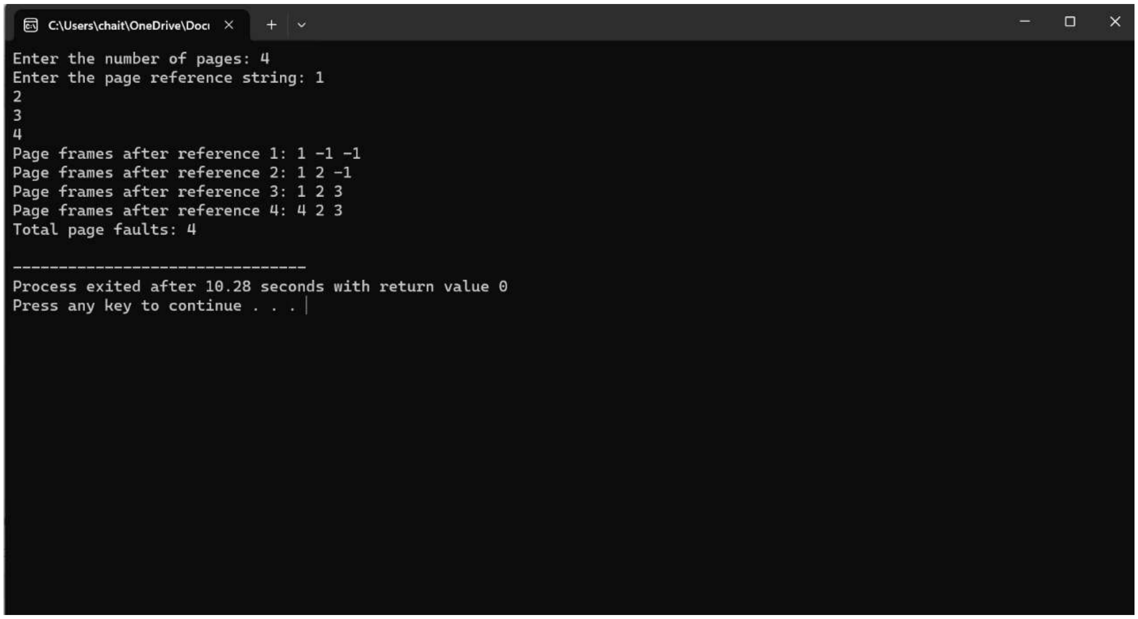


31. Construct a C program to simulate the First in First Out paging technique of memory management.

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

#define PAGE_FRAMES 3

int main() {
```

```
int page_frames[PAGE_FRAMES];
int page_faults = 0;
int page_count;
int oldest_index = 0; // Index of the oldest page in the page frames

printf("Enter the number of pages: ");
scanf("%d", &page_count);

int pages[page_count];

printf("Enter the page reference string: ");
for (int i = 0; i < page_count; i++) {
scanf("%d", &pages[i]);
}

// Initialize page frames to -1 (indicating empty)
for (int i = 0; i < PAGE_FRAMES; i++) {
page_frames[i] = -1;
}

// Simulate FIFO paging
for (int i = 0; i < page_count; i++) {
int page = pages[i];
int found = 0;

// Check if page is already in page frame
for (int j = 0; j < PAGE_FRAMES; j++) {
if (page_frames[j] == page) {
found = 1;
break;
}
}

// Page fault: Replace the oldest page
if (!found) {
page_frames[oldest_index] = page;
oldest_index = (oldest_index + 1) % PAGE_FRAMES; // Update oldest page index
page_faults++;
}

// Print current state of page frames
printf("Page frames after reference %d: ", page);
for (int j = 0; j < PAGE_FRAMES; j++) {
printf("%d ", page_frames[j]);
}
printf("\n");
}
```

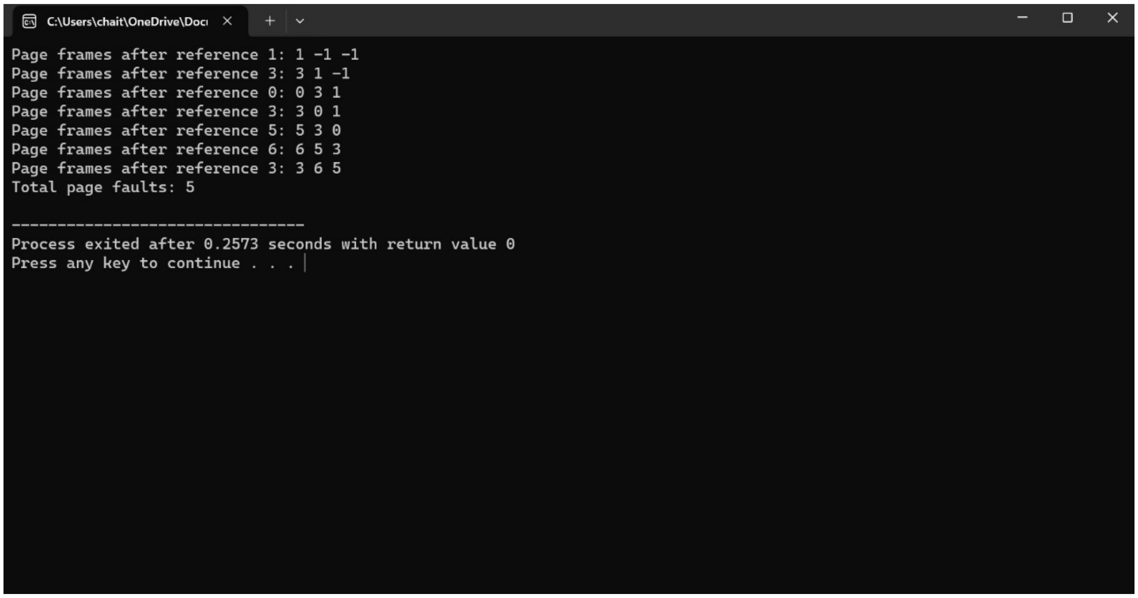
```
printf("Total page faults: %d\n", page_faults);

return 0;
}
```



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

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

#define PAGE_FRAMES 3

int main() {
int page_frames[PAGE_FRAMES];
int page_faults = 0;
int pages[] = {1, 3, 0, 3, 5, 6, 3}; // Reference string
int page_count = sizeof(pages) / sizeof(pages[0]);

// Initialize page frames to -1 (indicating empty)
for (int i = 0; i < PAGE_FRAMES; i++) {
page_frames[i] = -1;
}

// Simulate LRU paging
for (int i = 0; i < page_count; i++) {
int page = pages[i];
int found = 0;

// Check if page is already in page frame
for (int j = 0; j < PAGE_FRAMES; j++) {
if (page_frames[j] == page) {
found = 1;
```

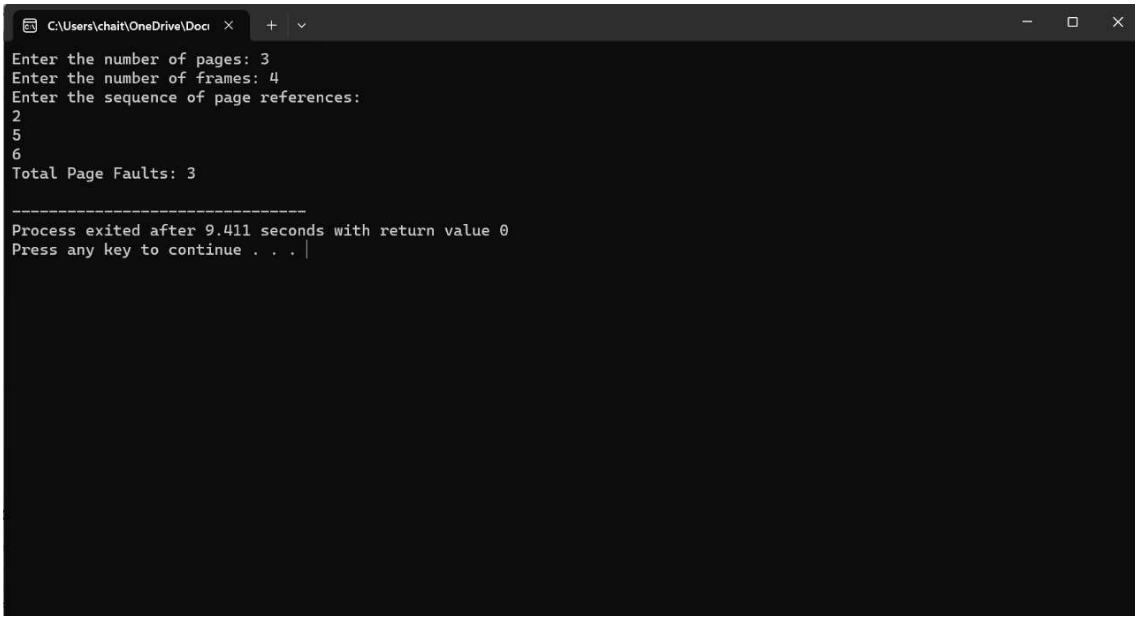
```
// Update the page's position in page_frames (move to front)
for (int k = j; k > 0; k--) {
page_frames[k] = page_frames[k - 1];
}
page_frames[0] = page;

break;
}
}

// Page fault: Replace the least recently used page
if (!found) {
// Move all pages one step forward to make space for the new page
for (int j = PAGE_FRAMES - 1; j > 0; j--) {
page_frames[j] = page_frames[j - 1];
}
page_frames[0] = page;
page_faults++;
}

// Print current state of page frames
printf("Page frames after reference %d: ", page);
for (int j = 0; j < PAGE_FRAMES; j++) {
printf("%d ", page_frames[j]);
}
printf("\n");
}

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

return 0;
}
```



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

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

#define NUM_FRAMES 3 // Number of frames in memory

// Function to find the optimal page to replace
int optimalPage(int pages[], int numOfPages, int frames[], int numOfFrames, int index) {
int res = -1, farthest = index;
for (int i = 0; i < numOfFrames; i++) {
int j;
for (j = index; j < numOfPages; j++) {
if (frames[i] == pages[j]) {
if (j > farthest) {
farthest = j;
res = i;
}
break;
}
}
if (j == numOfPages)
return i;
}
return (res == -1) ? 0 : res;
}

// Function to simulate optimal paging
void optimalPaging(int pages[], int numOfPages, int numOfFrames) {
int frames[numOfFrames], pageFaults = 0;
bool isPageFault;
```

```
for (int i = 0; i < numOfFrames; i++)
frames[i] = -1;

for (int i = 0; i < numOfPages; i++) {
isPageFault = true;

for (int j = 0; j < numOfFrames; j++) {
if (frames[j] == pages[i]) {
isPageFault = false;
break;
}
}

if (isPageFault) {
int index = optimalPage(pages, numOfPages, frames, numOfFrames, i + 1);
frames[index] = pages[i];
pageFaults++;
}
}

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

int main() {
int numOfPages, numOfFrames;

printf("Enter the number of pages: ");
scanf("%d", &numOfPages);

printf("Enter the number of frames: ");
scanf("%d", &numOfFrames);

int pages[numOfPages];

printf("Enter the sequence of page references:\n");
for (int i = 0; i < numOfPages; i++)
scanf("%d", &pages[i]);

optimalPaging(pages, numOfPages, numOfFrames);

return 0;
}
```

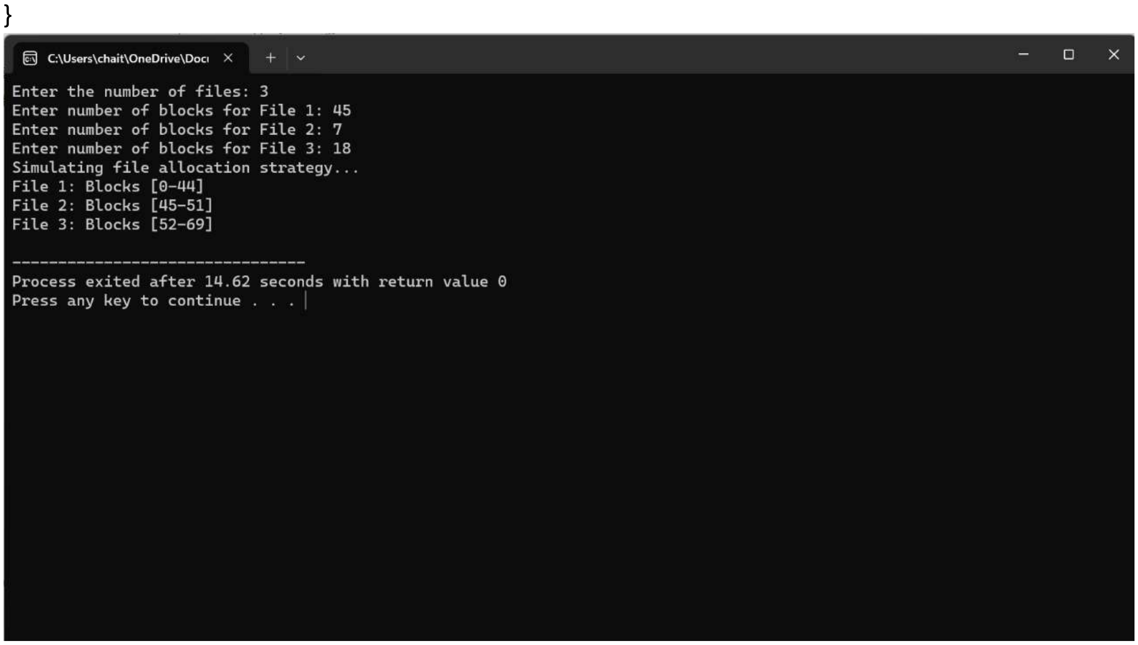


34. Consider a file system where there cords of the file are stored one after another both physically and logically. Are cord of the file can only be accessed by reading all the previous records. Design a C program to simulate the file allocation strategy.

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

#define DISK_SIZE 1000  // Total number of blocks on the disk
#define BLOCK_SIZE 512   // Size of each block (in bytes)
#define MAX_FILES 10     // Maximum number of files in the system

// Structure to represent a file
typedef struct {
int startBlock;     // Starting block number of the file
int numBlocks;      // Number of blocks allocated to the file
} File;

// Structure to represent the disk
typedef struct {
bool allocated[DISK_SIZE];  // Array to track allocated blocks on the disk
} Disk;

// Function to initialize the disk
void initializeDisk(Disk *disk) {
for (int i = 0; i < DISK_SIZE; i++) {
disk->allocated[i] = false;
}
}

// Function to allocate blocks to a file
bool allocateBlocks(Disk *disk, File *file) {
```

```c
int blocksNeeded = file->numBlocks;

// Find contiguous free blocks
int consecutiveBlocks = 0;
for (int i = 0; i < DISK_SIZE; i++) {
if (!disk->allocated[i]) {
consecutiveBlocks++;
if (consecutiveBlocks == blocksNeeded) {
file->startBlock = i - blocksNeeded + 1;
break;
}
} else {
consecutiveBlocks = 0;
}
}

// Check if enough contiguous free blocks are found
if (consecutiveBlocks == blocksNeeded) {
// Mark allocated blocks on the disk
for (int i = file->startBlock; i < file->startBlock + blocksNeeded; i++) {
disk->allocated[i] = true;
}
return true;
} else {
printf("Error: Not enough contiguous free blocks.\n");
return false;
}
}

// Function to simulate file allocation
void simulateFileAllocation(Disk *disk, File files[], int numFiles) {
printf("Simulating file allocation strategy...\n");
for (int i = 0; i < numFiles; i++) {
printf("File %d: Blocks [%d-%d]\n", i+1, files[i].startBlock, files[i].startBlock +
files[i].numBlocks - 1);
}
}

int main() {
Disk disk;
initializeDisk(&disk);

int numFiles;
printf("Enter the number of files: ");
scanf("%d", &numFiles);

if (numFiles > MAX_FILES) {
printf("Error: Exceeded maximum number of files.\n");
```

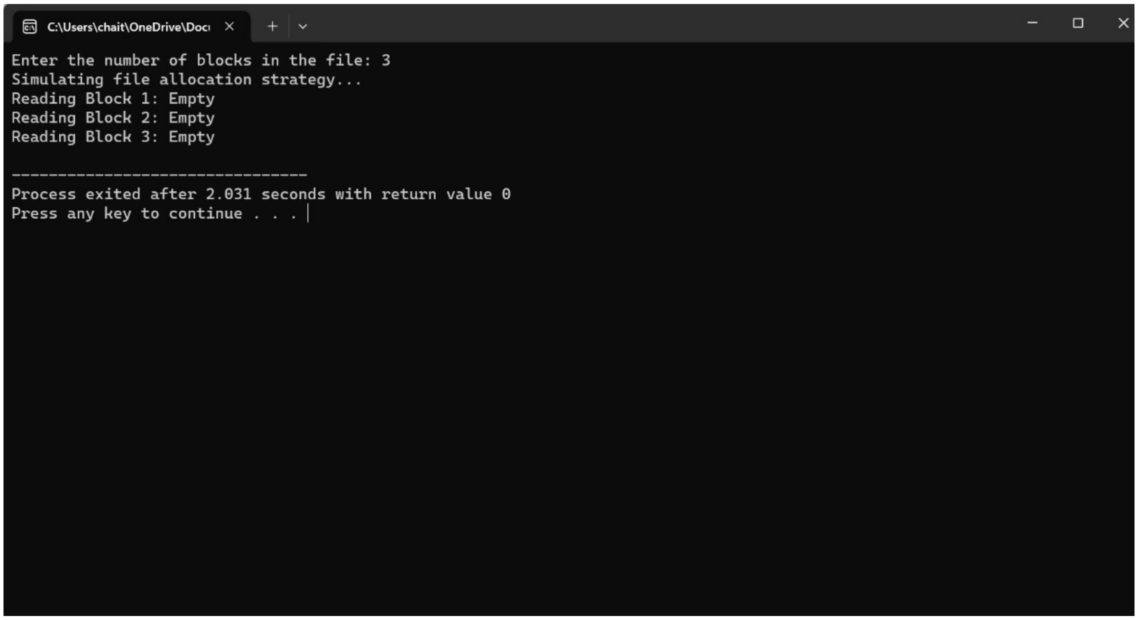
```
return 1;
}

File files[numFiles];

for (int i = 0; i < numFiles; i++) {
printf("Enter number of blocks for File %d: ", i+1);
scanf("%d", &files[i].numBlocks);

if (files[i].numBlocks <= 0 || files[i].numBlocks > DISK_SIZE) {
printf("Error: Invalid number of blocks for File %d.\n", i+1);
return 1;
}

if (!allocateBlocks(&disk, &files[i])) {
return 1;
}
}

// Simulate file allocation
simulateFileAllocation(&disk, files, numFiles);

return 0;
}
```



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

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

#define MAX_BLOCKS 100 // Maximum number of blocks in the file
```

```
#define BLOCK_SIZE 512 // Size of each block (in bytes)

// Structure to represent a block in the file
typedef struct Block {
char data[BLOCK_SIZE];
} Block;

// Structure to represent the index block
typedef struct IndexBlock {
int blockPointers[MAX_BLOCKS]; // Array to store block pointers
int numOfBlocks; // Number of blocks in the file
} IndexBlock;

// Function to simulate the file allocation strategy
void simulateFileAllocation(IndexBlock *indexBlock) {
printf("Simulating file allocation strategy...\n");

// Reading blocks using index block pointers
for (int i = 0; i < indexBlock->numOfBlocks; i++) {
printf("Reading Block %d: %s\n", i + 1, indexBlock->blockPointers[i] == -1 ? "Empty" :
"Data");
}
}

int main() {
IndexBlock indexBlock;
indexBlock.numOfBlocks = 0;

// Initialize block pointers to -1 (indicating empty)
for (int i = 0; i < MAX_BLOCKS; i++) {
indexBlock.blockPointers[i] = -1;
}

printf("Enter the number of blocks in the file: ");
scanf("%d", &indexBlock.numOfBlocks);

// Check for valid number of blocks
if (indexBlock.numOfBlocks <= 0 || indexBlock.numOfBlocks > MAX_BLOCKS) {
printf("Invalid number of blocks.\n");
return 1;
}

// Simulate file allocation strategy
simulateFileAllocation(&indexBlock);

return 0;
}
```

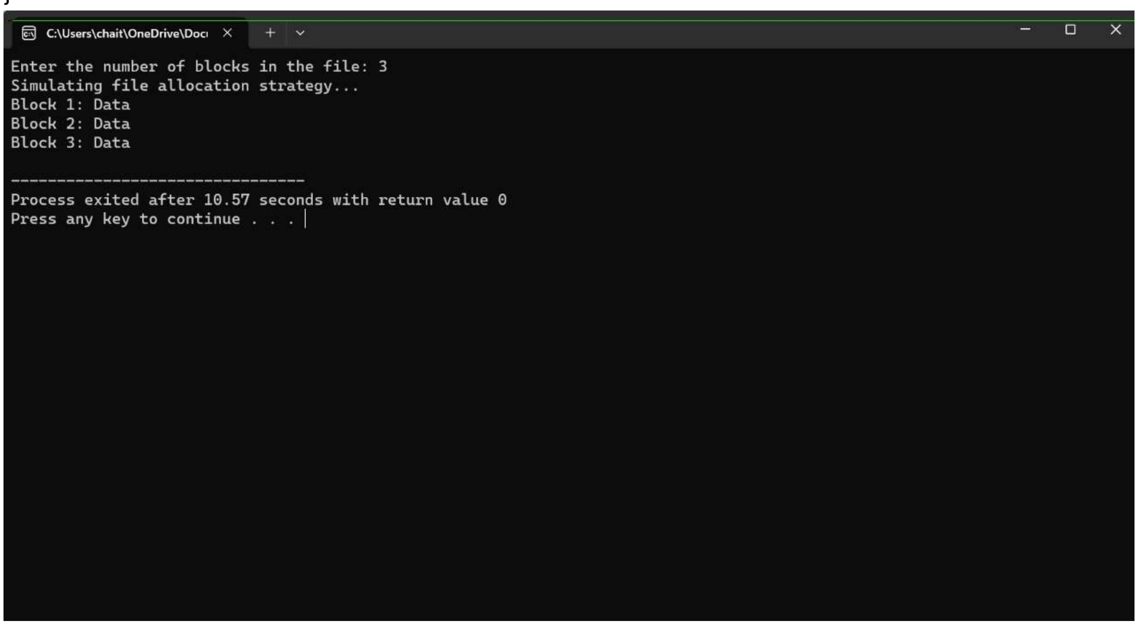


36. **With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block. Design a C program to simulate the file allocation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

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

#define BLOCK_SIZE 512 // Size of each block (in bytes)

// Structure to represent a block in the file
typedef struct Block {
char data[BLOCK_SIZE];
struct Block *next; // Pointer to the next block
} Block;

// Structure to represent a file
typedef struct File {
Block *firstBlock; // Pointer to the first block of the file
Block *lastBlock; // Pointer to the last block of the file
} File;

// Function to initialize a file
void initFile(File *file) {
file->firstBlock = NULL;
file->lastBlock = NULL;
}

// Function to add a block to the end of a file
void addBlockToFile(File *file, Block *block) {
if (file->firstBlock == NULL) {
```

```
// If file is empty, set the first block
file->firstBlock = block;
} else {
// Link the new block to the last block
file->lastBlock->next = block;
}
// Update the last block to the new block
file->lastBlock = block;
}

// Function to simulate the file allocation strategy
void simulateFileAllocation(File *file) {
printf("Simulating file allocation strategy...\n");

Block *currentBlock = file->firstBlock;
int blockCount = 0;

// Traverse the linked list of blocks
while (currentBlock != NULL) {
printf("Block %d: Data\n", ++blockCount);
currentBlock = currentBlock->next;
}
}

// Function to free memory allocated for file blocks
void freeFile(File *file) {
Block *currentBlock = file->firstBlock;
Block *nextBlock;

// Traverse the linked list of blocks and free memory
while (currentBlock != NULL) {
nextBlock = currentBlock->next;
free(currentBlock);
currentBlock = nextBlock;
}

// Reset file pointers
file->firstBlock = NULL;
file->lastBlock = NULL;
}

int main() {
File file;
initFile(&file);

int numOfBlocks;

printf("Enter the number of blocks in the file: ");
```

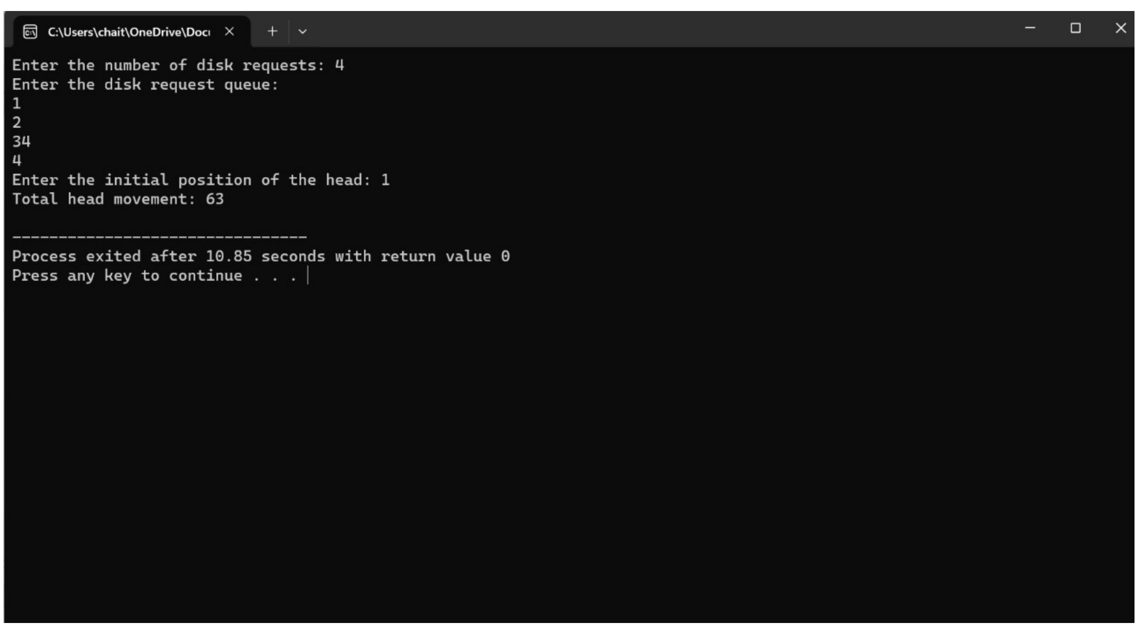
```
scanf("%d", &numOfBlocks);

// Create and add blocks to the file
for (int i = 0; i < numOfBlocks; i++) {
Block *block = (Block *)malloc(sizeof(Block));
block->next = NULL; // Initialize next pointer to NULL
addBlockToFile(&file, block);
}

// Simulate file allocation strategy
simulateFileAllocation(&file);

// Free memory allocated for file blocks
freeFile(&file);

return 0;
}
```



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

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

// Function to calculate total head movement
int calculateHeadMovement(int queue[], int head, int size) {
int totalMovement = 0;

// Traverse the queue and calculate head movement
for (int i = 0; i < size; i++) {
totalMovement += abs(queue[i] - head);
head = queue[i];
```

```
}

return totalMovement;
}

int main() {
int n; // Number of disk requests
printf("Enter the number of disk requests: ");
scanf("%d", &n);

int requestQueue[n]; // Disk request queue

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

int initialHead; // Initial position of the head
printf("Enter the initial position of the head: ");
scanf("%d", &initialHead);

int totalHeadMovement = calculateHeadMovement(requestQueue, initialHead, n);

printf("Total head movement: %d\n", totalHeadMovement);

return 0;
}
```

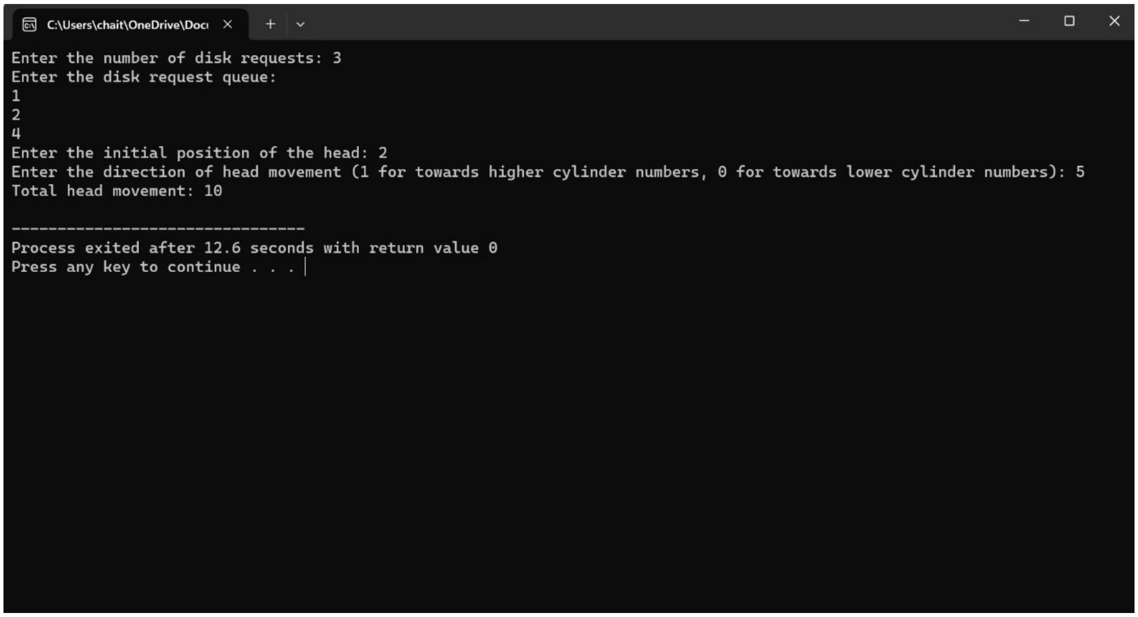


38. Design a C program to simulate SCAN disk scheduling algorithm.

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

```
#define MAX_REQUESTS 1000

// Function to sort an array in ascending order
void sort(int arr[], int n) {
for (int i = 0; i < n - 1; i++) {
for (int j = 0; j < n - i - 1; j++) {
if (arr[j] > arr[j + 1]) {
int temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
}
}
}
}

// Function to simulate SCAN disk scheduling algorithm
int SCAN(int queue[], int head, int size, int direction) {
int totalMovement = 0;
int currentIndex = 0;
int i;

if (direction == 1) { // Moving towards higher cylinder numbers
// Find the index where head movement should change direction
for (i = 0; i < size; i++) {
if (queue[i] >= head) {
currentIndex = i;
break;
}
}
} else { // Moving towards lower cylinder numbers
// Find the index where head movement should change direction
for (i = size - 1; i >= 0; i--) {
if (queue[i] <= head) {
currentIndex = i;
break;
}
}
}

// Calculate head movement
for (i = currentIndex; i < size; i++) {
totalMovement += abs(queue[i] - head);
head = queue[i];
}

if (direction == 1) {
totalMovement += abs(head - 0); // Move to cylinder 0
```

```
head = 0;
for (i = currentIndex - 1; i >= 0; i--) {
totalMovement += abs(queue[i] - head);
head = queue[i];
}
} else {
totalMovement += abs(head - 0); // Move to cylinder 0
head = 0;
for (i = currentIndex + 1; i < size; i++) {
totalMovement += abs(queue[i] - head);
head = queue[i];
}
}

return totalMovement;
}

int main() {
int n; // Number of disk requests
printf("Enter the number of disk requests: ");
scanf("%d", &n);

int requestQueue[MAX_REQUESTS]; // Disk request queue

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

int initialHead; // Initial position of the head
printf("Enter the initial position of the head: ");
scanf("%d", &initialHead);

int direction; // Direction of head movement (1: towards higher cylinder numbers, 0: towards lower cylinder numbers)
printf("Enter the direction of head movement (1 for towards higher cylinder numbers, 0 for towards lower cylinder numbers): ");
scanf("%d", &direction);

sort(requestQueue, n); // Sort the request queue

int totalHeadMovement = SCAN(requestQueue, initialHead, n, direction);

printf("Total head movement: %d\n", totalHeadMovement);

return 0;
}
```

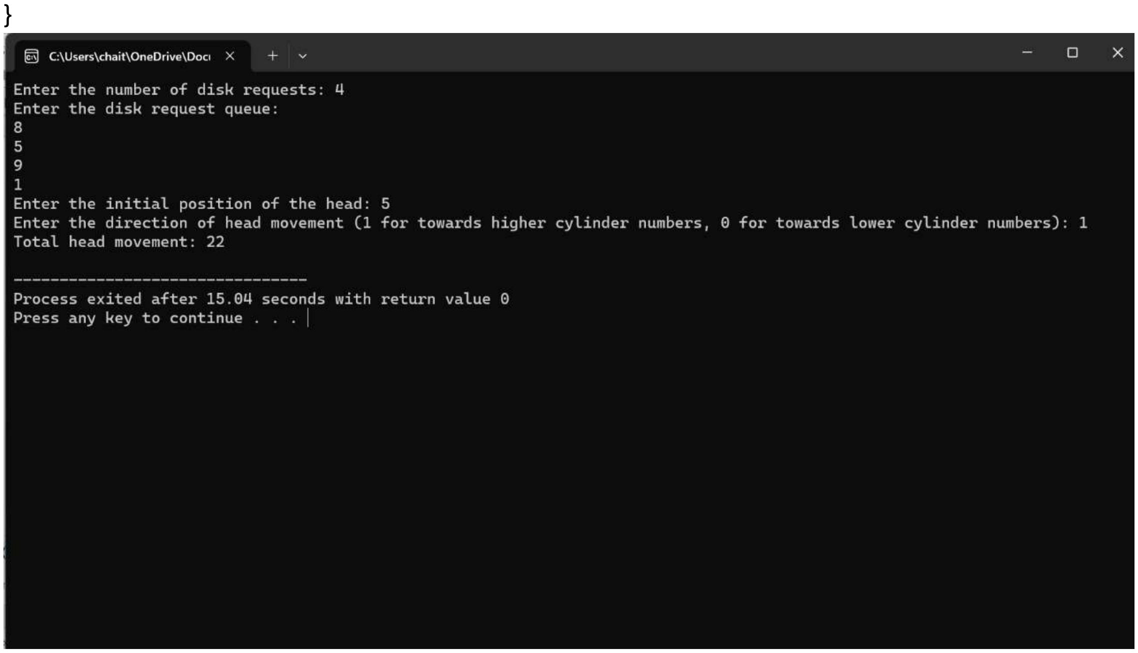


39. Develop a C program to simulate C-SCAN disk scheduling algorithm.

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

#define MAX_REQUESTS 1000

// Function to sort an array in ascending order
void sort(int arr[], int n) {
for (int i = 0; i < n - 1; i++) {
for (int j = 0; j < n - i - 1; j++) {
if (arr[j] > arr[j + 1]) {
int temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
}
}
}
}

// Function to simulate C-SCAN disk scheduling algorithm
int CSCAN(int queue[], int head, int size, int direction) {
int totalMovement = 0;
int currentIndex = 0;
int i;

sort(queue, size); // Sort the request queue

if (direction == 1) { // Moving towards higher cylinder numbers
// Find the index where head movement should change direction
for (i = 0; i < size; i++) {
if (queue[i] >= head) {
```

```
currentIndex = i;
break;
}
}
} else { // Moving towards lower cylinder numbers
// Find the index where head movement should change direction
for (i = size - 1; i >= 0; i--) {
if (queue[i] <= head) {
currentIndex = i;
break;
}
}
}

// Calculate head movement
for (i = currentIndex; i < size; i++) {
totalMovement += abs(queue[i] - head);
head = queue[i];
}

if (direction == 1) {
totalMovement += abs(head - 0); // Move to cylinder 0
head = 0;
totalMovement += abs(head - queue[0]); // Move to the first cylinder after 0
head = queue[0];
for (i = 1; i < size; i++) {
totalMovement += abs(queue[i] - head);
head = queue[i];
}
} else {
totalMovement += abs(head - 0); // Move to cylinder 0
head = 0;
totalMovement += abs(head - queue[size - 1]); // Move to the last cylinder before 0
head = queue[size - 1];
for (i = size - 2; i >= 0; i--) {
totalMovement += abs(queue[i] - head);
head = queue[i];
}
}

return totalMovement;
}

int main() {
int n; // Number of disk requests
printf("Enter the number of disk requests: ");
scanf("%d", &n);
```

```
int requestQueue[MAX_REQUESTS]; // Disk request queue

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

int initialHead; // Initial position of the head
printf("Enter the initial position of the head: ");
scanf("%d", &initialHead);

int direction; // Direction of head movement (1: towards higher cylinder numbers, 0: towards lower cylinder numbers)
printf("Enter the direction of head movement (1 for towards higher cylinder numbers, 0 for towards lower cylinder numbers): ");
scanf("%d", &direction);

int totalHeadMovement = CSCAN(requestQueue, initialHead, n, direction);

printf("Total head movement: %d\n", totalHeadMovement);

return 0;
}
```



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

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

```
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

int main() {
char *filename = "testfile.txt";
char *content = "Hello, world!\n";
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH; // Permissions: rw-
rw-r--

// Create the file
int fd = open(filename, O_CREAT | O_WRONLY | O_TRUNC, mode);
if (fd == -1) {
perror("Failed to create file");
exit(EXIT_FAILURE);
}

// Write content to the file
if (write(fd, content, strlen(content)) == -1) {
perror("Failed to write to file");
exit(EXIT_FAILURE);
}

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

// Simulate different users attempting to access the file
printf("Simulating different users accessing the file:\n");

// Owner
printf("Owner:\n");
if (access(filename, R_OK | W_OK) == 0) {
printf("  Read and write access granted\n");
} else {
printf("  Read and write access denied\n");
}

// Group
printf("Group:\n");
if (access(filename, R_OK | W_OK) == 0) {
printf("  Read and write access granted\n");
} else {
printf("  Read and write access denied\n");
}
```

```
// Others
printf("Others:\n");
if (access(filename, R_OK) == 0) {
printf("  Read access granted\n");
} else {
printf("  Read access denied\n");
}

return 0;
}
```