1. Any shell scripting program.

```
1.if else

Code-


#!/bin/bash

echo "Enter a number: "

read number


if [ $number -gt 0 ]; then

    echo "The number is positive."

elif [ $number -lt 0 ]; then

    echo "The number is negative."

else

    echo "The number is zero."

fi




2.for loop

Code-

#!/bin/bash

echo "Enter a number: "

read number

is_prime=1
```

```
for ((i=2; i<=number/2; i++))
do
    if [ $((number%i)) -eq 0 ]; then
        is_prime=0
        break
    fi
done
if [ $number -eq 1 ]; then
    echo "1 is neither prime nor composite."
elif [ $is_prime -eq 1 ]; then
    echo "$number is a prime number."
else
    echo "$number is not a prime number."
fi
3.while loop
Code-
#!/bin/bash

echo "Enter a number: "
read number

original_number=$number
reverse_number=0
```

```
while [ $number -gt 0 ]
do
    remainder=$((number % 10))
    reverse_number=$((reverse_number * 10 + remainder))
    number=$((number / 10))
done

if [ $original_number -eq $reverse_number ]; then
    echo "$original_number is a palindrome."
else
    echo "$original_number is not a palindrome."
fi
```

4.until

Code-

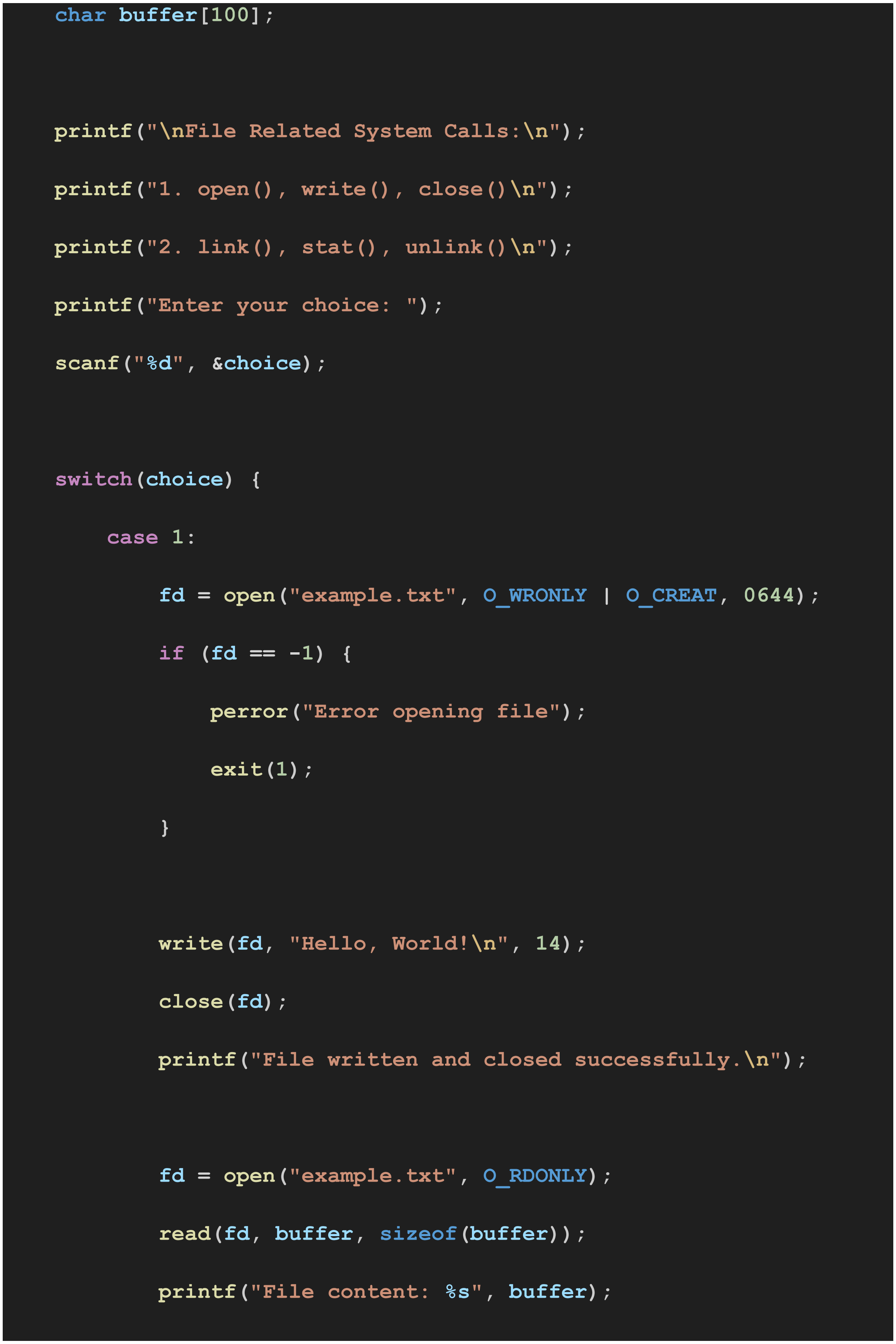
```
#!/bin/bash
echo "Enter a number: "
read number
sum=0
temp=$number
until [ $temp -eq 0 ]
do
    digit=$((temp % 10))
    sum=$((sum + digit ** 3))
    temp=$((temp / 10))
done
if [ $sum -eq $number ]; then
    echo "$number is an Armstrong number."
else
    echo "$number is not an Armstrong number."
fi
5.case
 #!/bin/bash
echo "Enter first number: "
read num1
echo "Enter second number: "
read num2

echo "Enter operation (+, -, *, /): "
read operation

case $operation in
    +)
        result=$((num1 + num2))
        echo "Result: $result"
        ;;
    -)
        result=$((num1 - num2))
        echo "Result: $result"
        ;;
    \*)
        result=$((num1 * num2))
        echo "Result: $result"
        ;;
    /)
        result=$((num1 / num2))
        echo "Result: $result"
        ;;
    *)
        echo "Invalid operation. Please enter +, -, *, or /."
        ;;
esac
```

2. Write a program demonstrating use of different system calls.

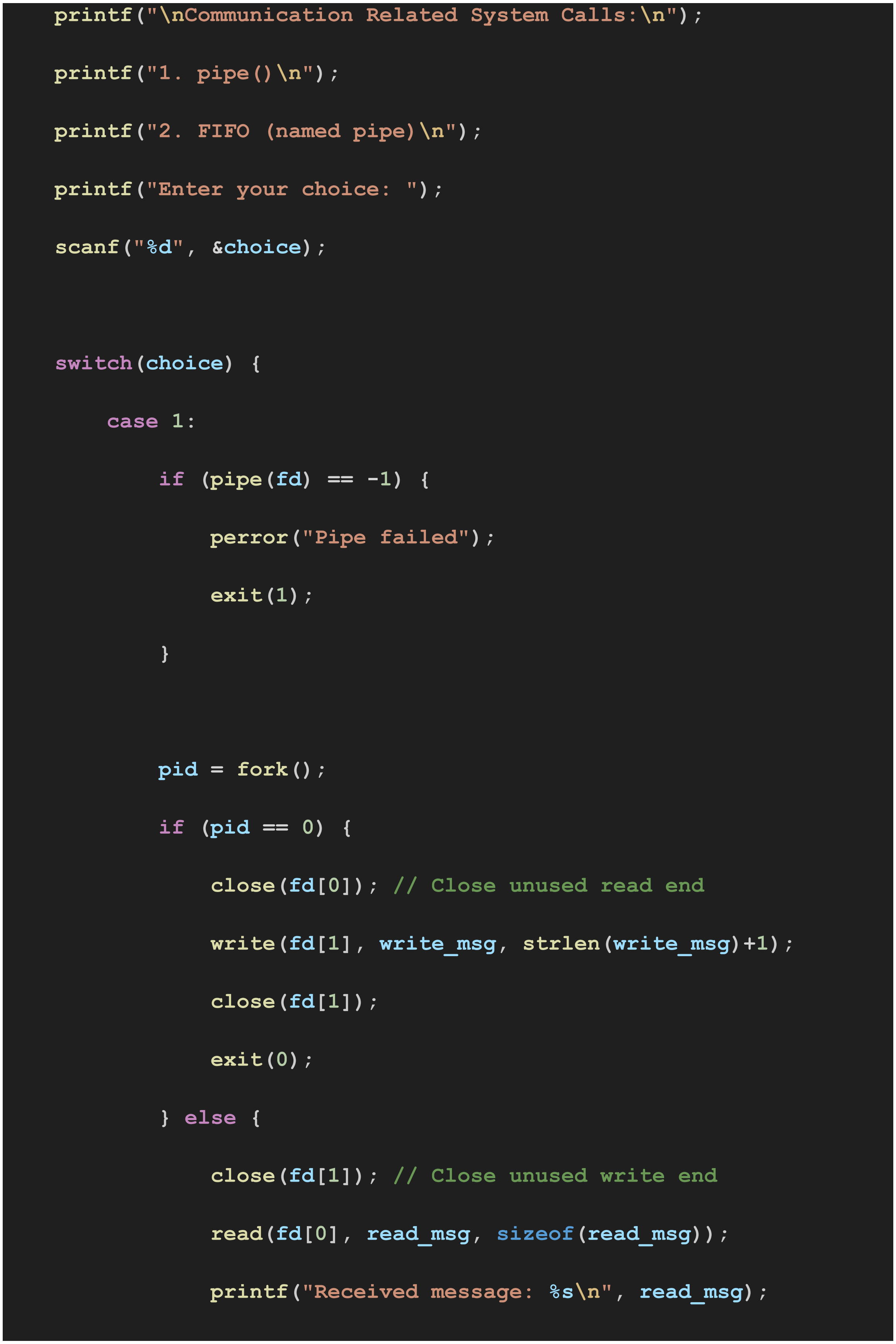
```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
```

```c
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <string.h>

// Function prototypes
void process_related();
void file_related();
void communication_related();
void info_related();

void process_related() {
    int choice;
    pid_t pid;

    printf("\nProcess Related System Calls:\n");
    printf("1. fork()\n");
    printf("2. exit()\n");
    printf("3. wait()\n");
    printf("4. exec()\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
    switch(choice) {
        case 1:
            pid = fork();
            if (pid == 0) {
                printf("Child process. PID = %d\n", getpid());
                exit(0);
            } else {
                printf("Parent process. PID = %d\n", getpid());
                wait(NULL);
            }
            break;
        case 2:
            printf("Exiting process with status 0...\n");
            exit(0);
            break;
        case 3:
            pid = fork();
            if (pid == 0) {
                printf("Child process created. PID = %d\n",
getpid());
                exit(0);
            } else {
                wait(NULL);
```

```c
                printf("Child process has terminated. Parent PID = %d\n", getpid());
            }
            break;
        case 4:
            pid = fork();
            if (pid == 0) {
                execl("/bin/ls", "ls", NULL);
                perror("execl failed");
                exit(0);
            } else {
                wait(NULL);
                printf("Executed ls command in child process.\n");
            }
            break;
        default:
            printf("Invalid choice.\n");
            break;
    }
}

void file_related() {
    int choice;
    int fd;
```



```
    char buffer[100];

    printf("\nFile Related System Calls:\n");
    printf("1. open(), write(), close()\n");
    printf("2. link(), stat(), unlink()\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch(choice) {
        case 1:
            fd = open("example.txt", O_WRONLY | O_CREAT, 0644);
            if (fd == -1) {
                perror("Error opening file");
                exit(1);
            }

            write(fd, "Hello, World!\n", 14);
            close(fd);
            printf("File written and closed successfully.\n");

            fd = open("example.txt", O_RDONLY);
            read(fd, buffer, sizeof(buffer));
            printf("File content: %s", buffer);
            close(fd);
            break;
        case 2:
            link("example.txt", "example_link.txt");
            struct stat file_stat;
            stat("example_link.txt", &file_stat);
            printf("Size of linked file: %ld bytes\n", file_stat.st_size);
            unlink("example_link.txt");
            printf("Link removed.\n");
            break;
        default:
            printf("Invalid choice.\n");
            break;
    }
}


void communication_related() {
    int choice;
    int fd[2];
    char write_msg[] = "Hello, World!";
    char read_msg[20];
    pid_t pid;
```



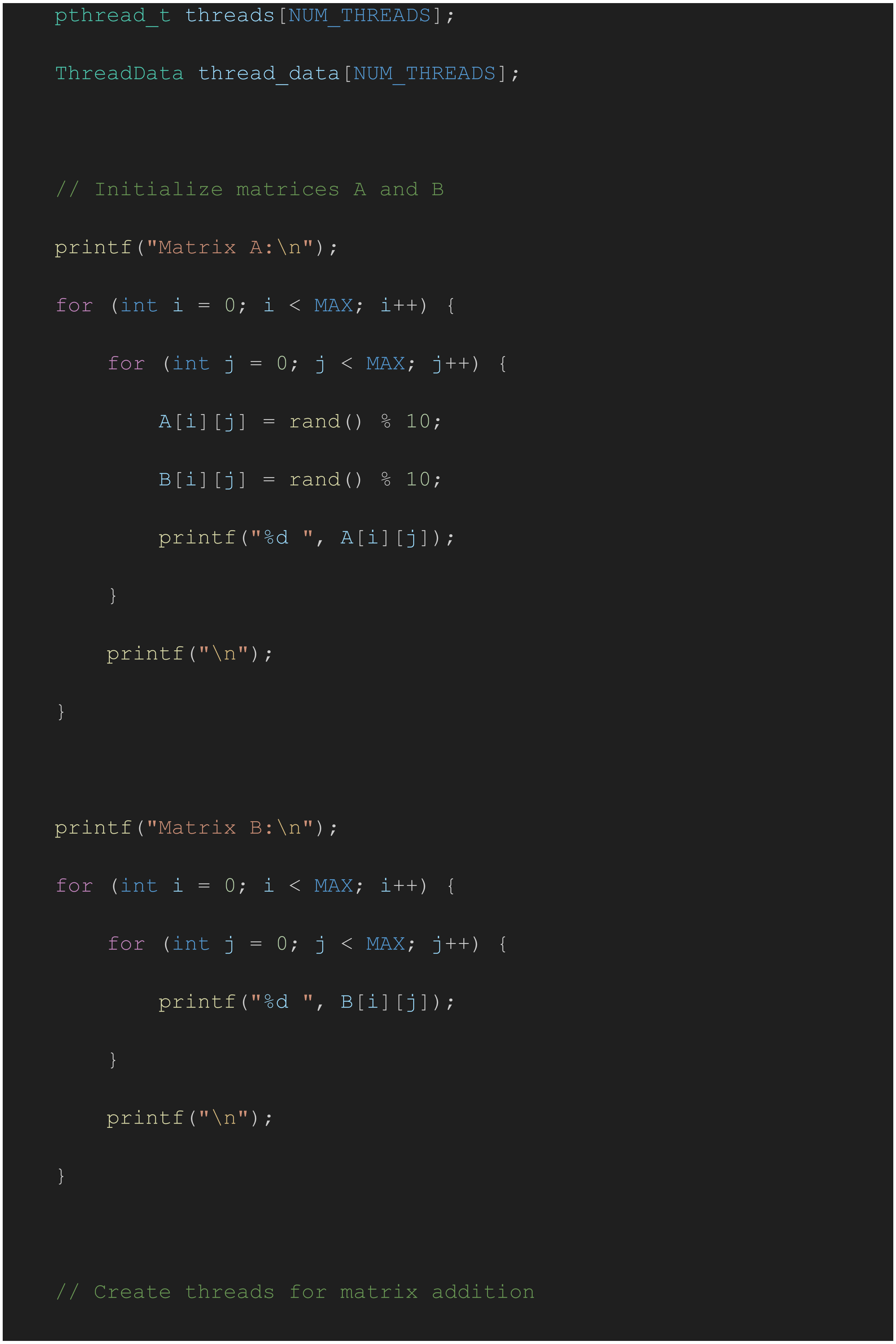
```
printf("\nCommunication Related System Calls:\n");
printf("1. pipe()\n");
printf("2. FIFO (named pipe)\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch(choice) {
    case 1:
        if (pipe(fd) == -1) {
            perror("Pipe failed");
            exit(1);
        }

        pid = fork();
        if (pid == 0) {
            close(fd[0]); // Close unused read end
            write(fd[1], write_msg, strlen(write_msg)+1);
            close(fd[1]);
            exit(0);
        } else {
            close(fd[1]); // Close unused write end
            read(fd[0], read_msg, sizeof(read_msg));
            printf("Received message: %s\n", read_msg);
```

```
                close(fd[0]);
                wait(NULL);
            }
            break;
        case 2:
            mkfifo("/tmp/myfifo", 0666);
            pid = fork();

            if (pid == 0) {
                int fd = open("/tmp/myfifo", O_WRONLY);
                write(fd, write_msg, strlen(write_msg)+1);
                close(fd);
                exit(0);
            } else {
                int fd = open("/tmp/myfifo", O_RDONLY);
                read(fd, read_msg, sizeof(read_msg));
                printf("Received message: %s\n", read_msg);
                close(fd);
                unlink("/tmp/myfifo");
                wait(NULL);
            }
            break;
        default:
```

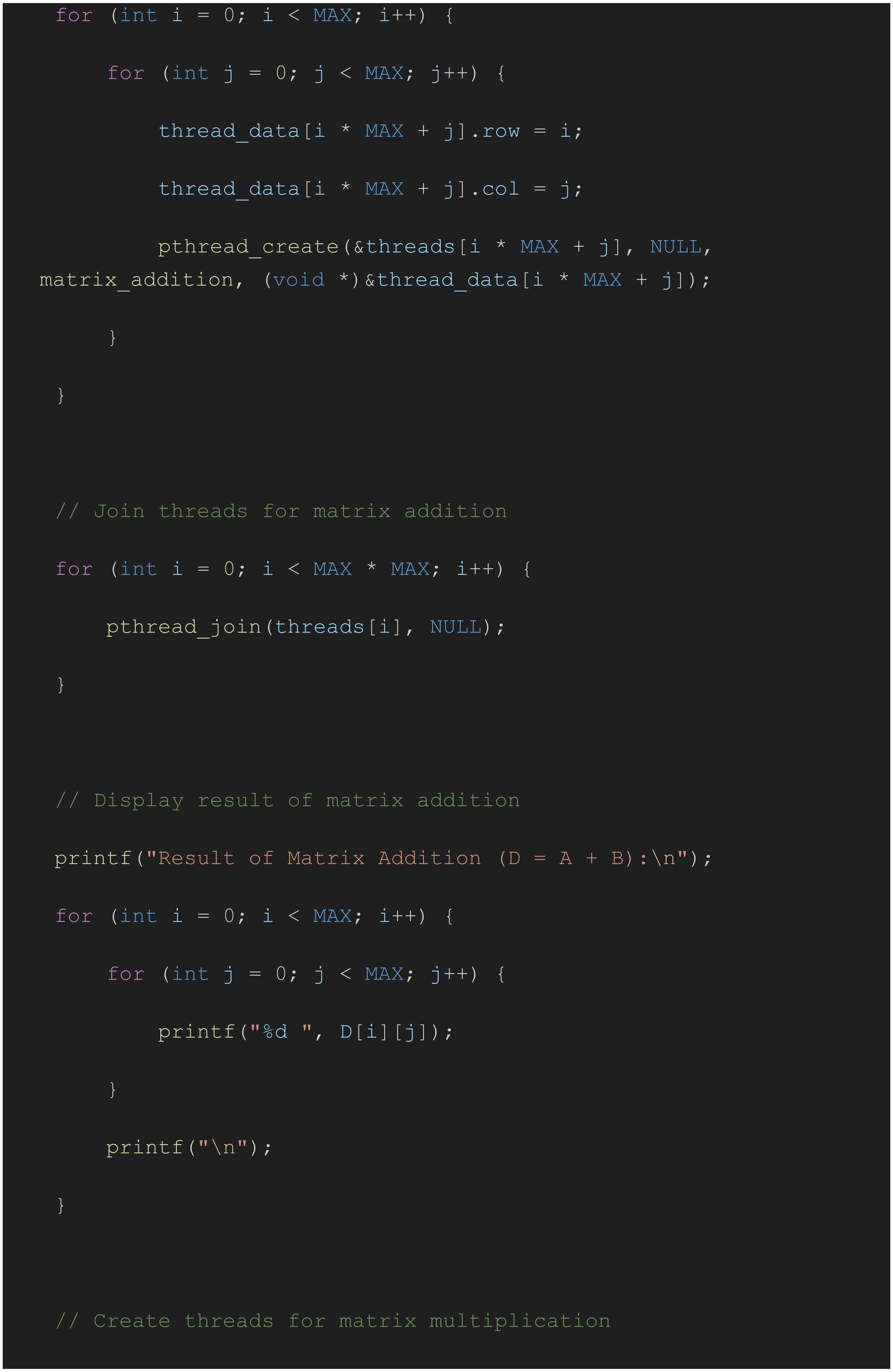
```c
            printf("Invalid choice.\n");

            break;

    }

}



void info_related() {

    int choice;



    printf("\nInformation Related System Calls:\n");

    printf("1. alarm()\n");

    printf("2. sleep()\n");

    printf("Enter your choice: ");

    scanf("%d", &choice);



    switch(choice) {

        case 1:

            printf("Setting an alarm for 5 seconds...\n");

            alarm(5);

            sleep(6); // Wait to see alarm trigger

            break;

        case 2:

            printf("Sleeping for 3 seconds...\n");

            sleep(3);
            printf("Woke up after 3 seconds.\n");
            break;
        default:
            printf("Invalid choice.\n");
            break;
    }
}

int main() {
    int choice;

    while(1) {
        printf("\nMenu:\n");
        printf("1. Process Related System Calls\n");
        printf("2. File Related System Calls\n");
        printf("3. Communication Related System Calls\n");
        printf("4. Information Related System Calls\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
                process_related();
                break;
            case 2:
                file_related();
                break;
            case 3:
                communication_related();
                break;
            case 4:
                info_related();
                break;
            case 5:
                printf("Exiting...\n");
                exit(0);
            default:
                printf("Invalid choice.\n");
                break;
        }
    }

    return 0;
}
```

3. Implement multithreading for Matrix Operations using Pthreads.

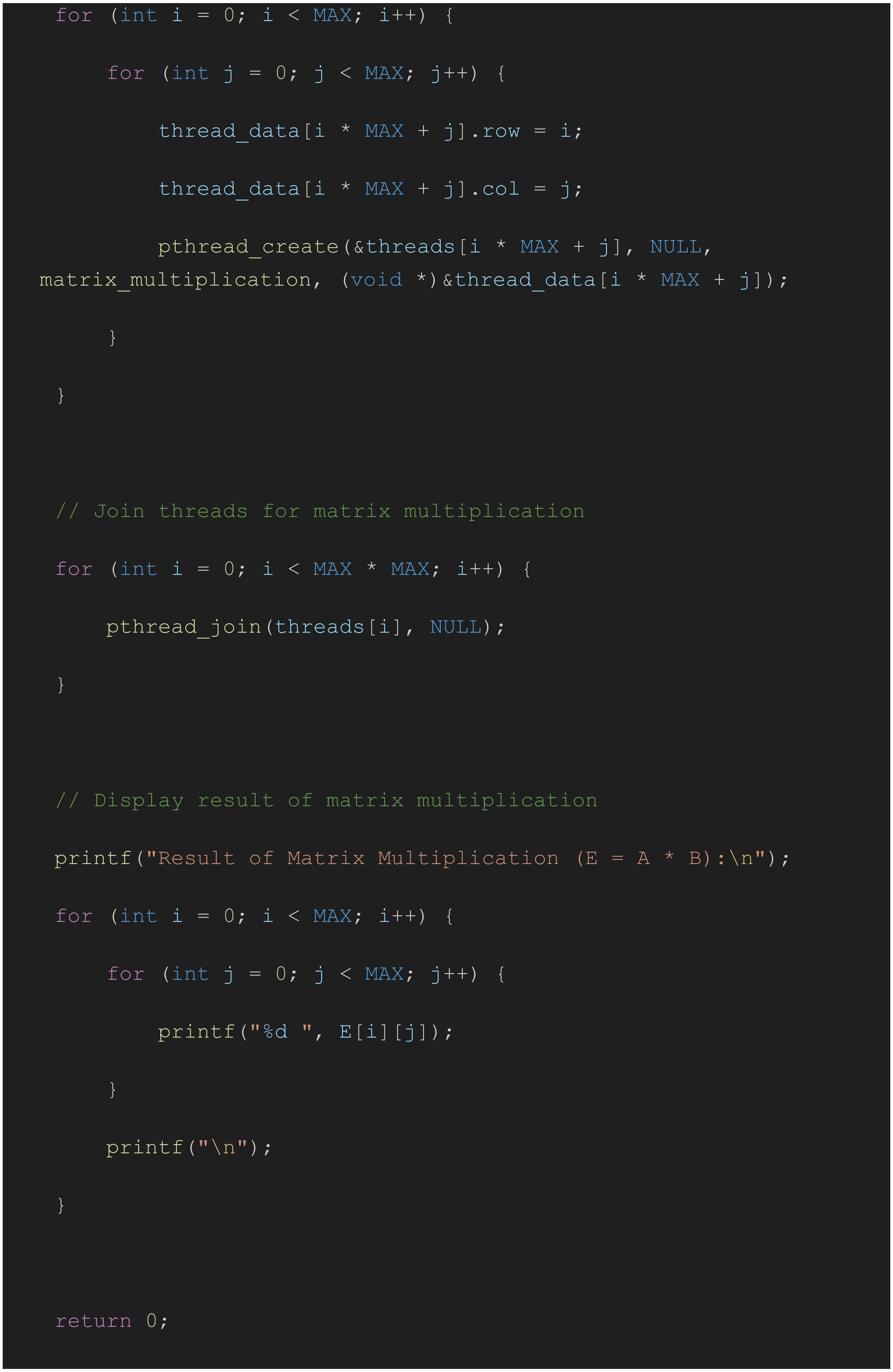
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define MAX 3  // Size of the matrix (MAX x MAX)
#define NUM_THREADS MAX * MAX  // Number of threads

int A[MAX][MAX], B[MAX][MAX], C[MAX][MAX]; // Matrices for operations
int D[MAX][MAX]; // Result matrix for addition
int E[MAX][MAX]; // Result matrix for multiplication

// Structure to pass data to threads
typedef struct {
    int row;
    int col;
} ThreadData;

// Function for matrix addition
void *matrix_addition(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    int row = data->row;

    int col = data->col;


    D[row][col] = A[row][col] + B[row][col];


    pthread_exit(0);

}


// Function for matrix multiplication

void *matrix_multiplication(void *arg) {

    ThreadData *data = (ThreadData *)arg;

    int row = data->row;

    int col = data->col;


    E[row][col] = 0;

    for (int k = 0; k < MAX; k++) {

        E[row][col] += A[row][k] * B[k][col];

    }


    pthread_exit(0);

}


int main() {
```



```
    pthread_t threads[NUM_THREADS];
    ThreadData thread_data[NUM_THREADS];

    // Initialize matrices A and B
    printf("Matrix A:\n");
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            A[i][j] = rand() % 10;
            B[i][j] = rand() % 10;
            printf("%d ", A[i][j]);
        }
        printf("\n");
    }

    printf("Matrix B:\n");
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            printf("%d ", B[i][j]);
        }
        printf("\n");
    }

    // Create threads for matrix addition
```



```
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            thread_data[i * MAX + j].row = i;
            thread_data[i * MAX + j].col = j;
            pthread_create(&threads[i * MAX + j], NULL,
matrix_addition, (void *)&thread_data[i * MAX + j]);
        }
    }

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

    // Display result of matrix addition
    printf("Result of Matrix Addition (D = A + B):\n");
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            printf("%d ", D[i][j]);
        }
        printf("\n");
    }

    // Create threads for matrix multiplication
```



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

     for (int j = 0; j < MAX; j++) {

         thread_data[i * MAX + j].row = i;

         thread_data[i * MAX + j].col = j;

         pthread_create(&threads[i * MAX + j], NULL,
matrix_multiplication, (void *)&thread_data[i * MAX + j]);

     }

 }



 // Join threads for matrix multiplication

 for (int i = 0; i < MAX * MAX; i++) {

     pthread_join(threads[i], NULL);

 }



 // Display result of matrix multiplication

 printf("Result of Matrix Multiplication (E = A * B):\n");

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

     for (int j = 0; j < MAX; j++) {

         printf("%d ", E[i][j]);

     }

     printf("\n");

 }



 return 0;
```

```
}
```

4. Implementation of Classical problems (reader writer) using Threads and Mutex

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

// Shared resource
int shared_data = 0;

// Counter for readers
int readcnt = 0;  // Equivalent to `read_count` in the original code

// Mutex for controlling access to the shared resource and reader count
pthread_mutex_t mutex;             // Protects readcnt (equivalent to read_count_mutex)
pthread_mutex_t wrt;               // Equivalent to wrt in the pseudocode, controls writers

// Max iterations for readers and writers
int MAX_ITERATIONS = 5;

// Reader function
void* reader(void* arg) {
```

```
int id = *((int*)arg);
free(arg);

for (int i = 0; i < MAX_ITERATIONS; i++) {
    // Reader wants to enter the critical section
    pthread_mutex_lock(&mutex);  // Equivalent to wait(mutex)
    readcnt++;
    if (readcnt == 1) {
        // First reader locks wrt to block writers
        pthread_mutex_lock(&wrt);  // Equivalent to wait(wrt)
    }
    pthread_mutex_unlock(&mutex);  // Allow other readers to enter by unlocking mutex

    // Reader is reading the shared resource
    printf("Reader %d is reading shared data: %d\n", id, shared_data);
    sleep(1);  // Simulating read time

    // Reader finished reading
    pthread_mutex_lock(&mutex);  // Lock to update readcnt
    readcnt--;
    if (readcnt == 0) {
        // Last reader unlocks wrt to allow writers
        pthread_mutex_unlock(&wrt);  // Equivalent to signal(wrt)
    }
```

```
        pthread_mutex_unlock(&mutex);  // Allow other readers/writers to proceed

        sleep(1);  // Simulating delay between reads
    }

    return NULL;
}

// Writer function
void* writer(void* arg) {
    int id = *((int*)arg);
    free(arg);

    for (int i = 0; i < MAX_ITERATIONS; i++) {
        // Writer wants to write
        pthread_mutex_lock(&wrt);  // Equivalent to wait(wrt)

        // Writer is writing to the shared resource
        shared_data += 10;  // Modifying the shared resource
        printf("Writer %d is writing new shared data: %d\n", id, shared_data);

        pthread_mutex_unlock(&wrt);  // Equivalent to signal(wrt) to allow other readers/writers
        sleep(2);  // Simulating write time
    }
    return NULL;
}

int main() {
    pthread_t readers[5], writers[2];

    // Initialize mutexes
    pthread_mutex_init(&mutex, NULL);  // For protecting readcnt
    pthread_mutex_init(&wrt, NULL);    // For controlling writers

    // Create reader threads
    for (int i = 0; i < 5; i++) {
        int* id = malloc(sizeof(int));
        *id = i + 1;
        pthread_create(&readers[i], NULL, reader, id);
    }

    // Create writer threads
    for (int i = 0; i < 2; i++) {
        int* id = malloc(sizeof(int));
        *id = i + 1;
        pthread_create(&writers[i], NULL, writer, id);
    }
    // Wait for threads to complete
    for (int i = 0; i < 5; i++) {
        pthread_join(readers[i], NULL);
    }
    for (int i = 0; i < 2; i++) {
        pthread_join(writers[i], NULL);
    }

    // Destroy mutexes
    pthread_mutex_destroy(&mutex);
    pthread_mutex_destroy(&wrt);

    printf("All readers and writers have finished their operations.\n");

    return 0;
}
```

5. Implementation of Classical problems(producer consumer) using Threads and Mutex

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

```
#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int count = 0;
pthread_mutex_t mutex;
pthread_cond_t not_full, not_empty;

void* producer(void* arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        item = rand() % 100;
        pthread_mutex_lock(&mutex);
        while (count == BUFFER_SIZE)
            pthread_cond_wait(&not_full, &mutex);
        buffer[count++] = item;
        printf("Producer produced: %d\n", item);
        pthread_cond_signal(&not_empty);
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
    return NULL;
}
```

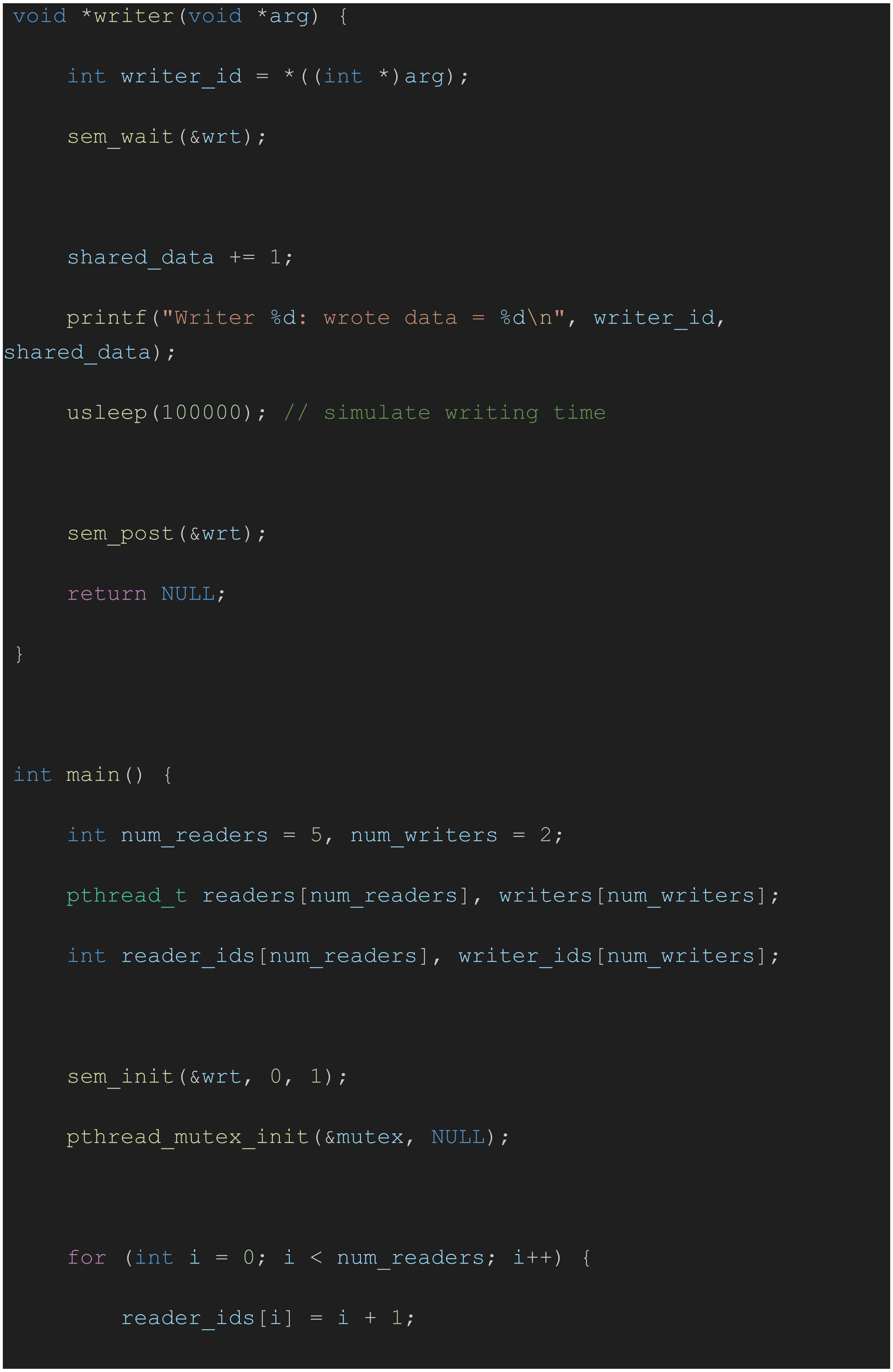
```c
void* consumer(void* arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        pthread_mutex_lock(&mutex);
        while (count == 0)
            pthread_cond_wait(&not_empty, &mutex);
        item = buffer[--count];
        printf("Consumer consumed: %d\n", item);
        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
    return NULL;
}

int main() {
    pthread_t prod_thread, cons_thread;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&not_full, NULL);
    pthread_cond_init(&not_empty, NULL);
    pthread_create(&prod_thread, NULL, producer, NULL);
    pthread_create(&cons_thread, NULL, consumer, NULL);

    pthread_join(prod_thread, NULL);
    pthread_join(cons_thread, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&not_full);
    pthread_cond_destroy(&not_empty);

    return 0;
}
```

6. Implementation of Classical problems (reader writer) using Threads and Semaphore. .(reader writer, producer consumer, dining philosopher)

   1. Reader-Writer Problem:

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

sem_t wrt;
pthread_mutex_t mutex;
```

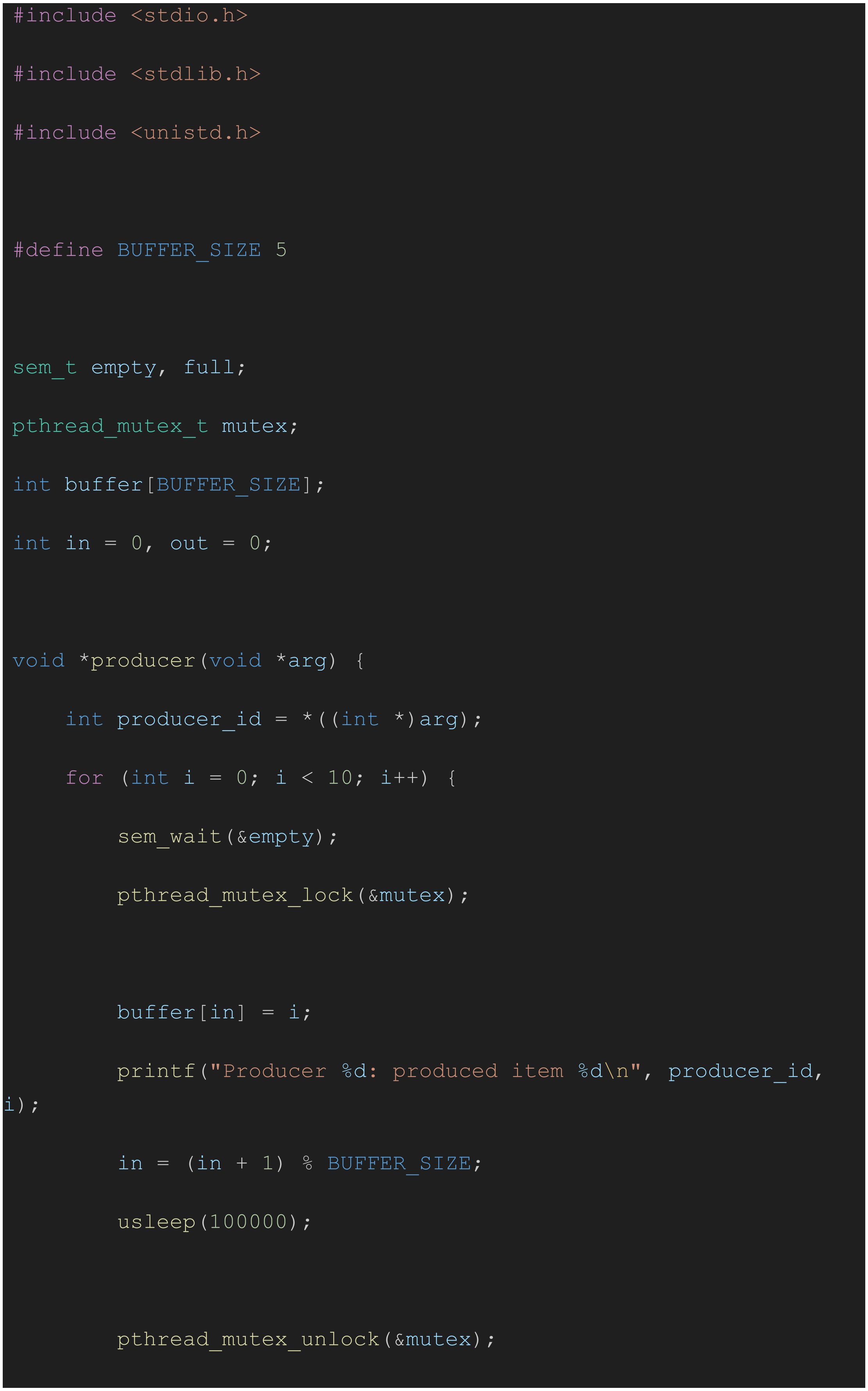
```
int read_count = 0;
int shared_data = 0;

void *reader(void *arg) {
    int reader_id = *((int *)arg);
    pthread_mutex_lock(&mutex);
    read_count++;
    if (read_count == 1)
        sem_wait(&wrt);
    pthread_mutex_unlock(&mutex);

    printf("Reader %d: read data = %d\n", reader_id,
shared_data);
    usleep(100000); // simulate reading time

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

    return NULL;
}
```



```c
void *writer(void *arg) {
    int writer_id = *((int *)arg);
    sem_wait(&wrt);

    shared_data += 1;
    printf("Writer %d: wrote data = %d\n", writer_id,
shared_data);
    usleep(100000); // simulate writing time

    sem_post(&wrt);
    return NULL;
}

int main() {
    int num_readers = 5, num_writers = 2;
    pthread_t readers[num_readers], writers[num_writers];
    int reader_ids[num_readers], writer_ids[num_writers];

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

    for (int i = 0; i < num_readers; i++) {
        reader_ids[i] = i + 1;
        pthread_create(&readers[i], NULL, reader,
&reader_ids[i]);
    }
    for (int i = 0; i < num_writers; i++) {
        writer_ids[i] = i + 1;
        pthread_create(&writers[i], NULL, writer,
&writer_ids[i]);
    }

    for (int i = 0; i < num_readers; i++)
        pthread_join(readers[i], NULL);
    for (int i = 0; i < num_writers; i++)
        pthread_join(writers[i], NULL);

    sem_destroy(&wrt);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

2.Producer-Consumer Problem

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



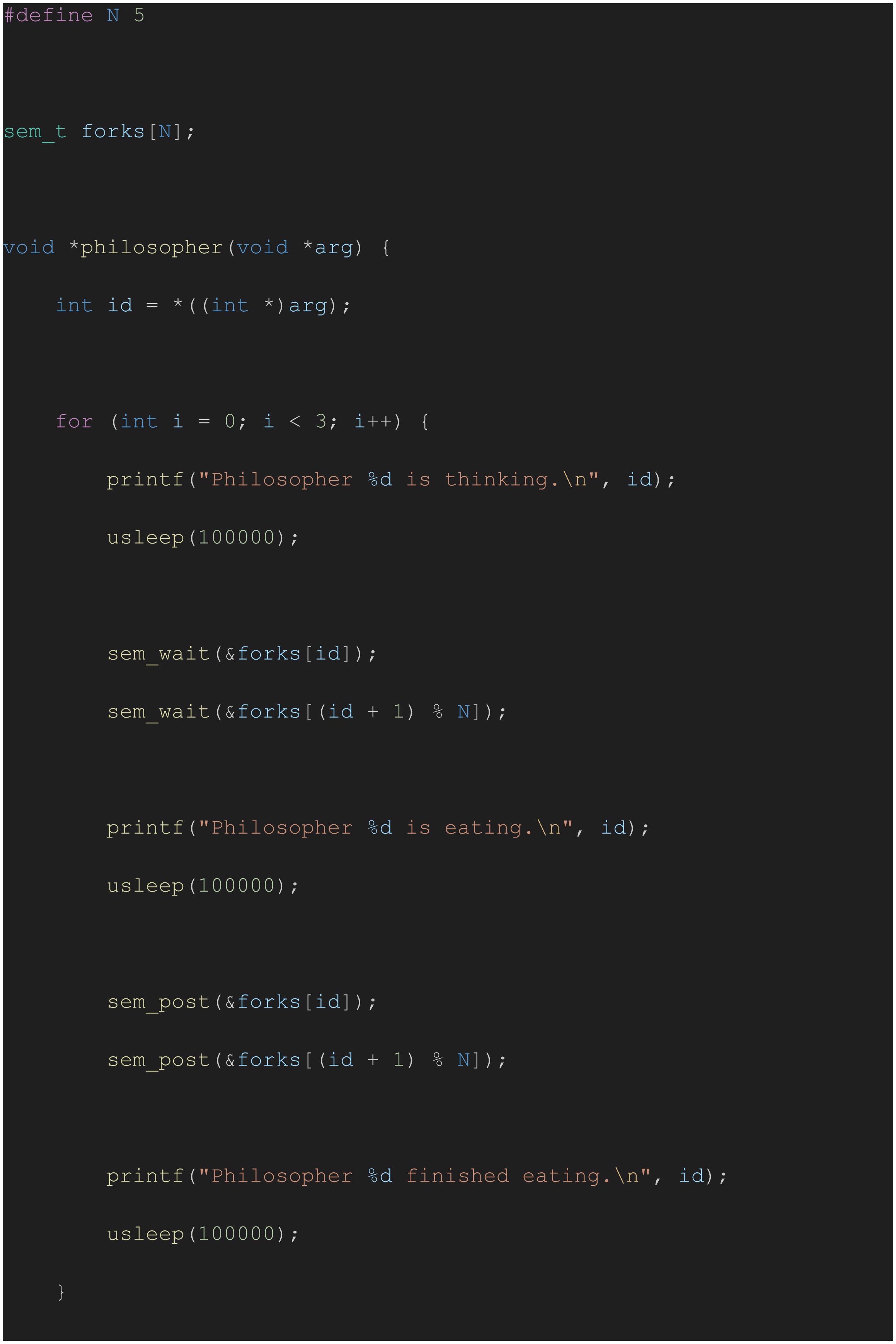
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#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 producer_id = *((int *)arg);
    for (int i = 0; i < 10; i++) {
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);

        buffer[in] = i;
        printf("Producer %d: produced item %d\n", producer_id,
i);
        in = (in + 1) % BUFFER_SIZE;
        usleep(100000);

        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
    return NULL;
}

void *consumer(void *arg) {
    int consumer_id = *((int *)arg);
    for (int i = 0; i < 10; i++) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);

        int item = buffer[out];
        printf("Consumer %d: consumed item %d\n", consumer_id, item);
        out = (out + 1) % BUFFER_SIZE;
        usleep(150000);

        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
    return NULL;
}

int main() {
    int num_producers = 2, num_consumers = 2;
    pthread_t producers[num_producers],
consumers[num_consumers];
    int producer_ids[num_producers],
consumer_ids[num_consumers];


    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);


    for (int i = 0; i < num_producers; i++) {
        producer_ids[i] = i + 1;
        pthread_create(&producers[i], NULL, producer,
&producer_ids[i]);
    }
    for (int i = 0; i < num_consumers; i++) {
        consumer_ids[i] = i + 1;
        pthread_create(&consumers[i], NULL, consumer,
&consumer_ids[i]);
    }


    for (int i = 0; i < num_producers; i++)
        pthread_join(producers[i], NULL);
    for (int i = 0; i < num_consumers; i++)
        pthread_join(consumers[i], NULL);
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

3.Dining Philosophers Problem

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

#define N 5

sem_t forks[N];

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

    for (int i = 0; i < 3; i++) {
        printf("Philosopher %d is thinking.\n", id);
        usleep(100000);
        sem_wait(&forks[id]);
        sem_wait(&forks[(id + 1) % N]);

        printf("Philosopher %d is eating.\n", id);
        usleep(100000);

        sem_post(&forks[id]);
        sem_post(&forks[(id + 1) % N]);

        printf("Philosopher %d finished eating.\n", id);
        usleep(100000);
    }
    return NULL;
}

int main() {
    pthread_t philosophers[N];
    int ids[N];

    for (int i = 0; i < N; i++)
        sem_init(&forks[i], 0, 1);
    for (int i = 0; i < N; i++) {
        ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher, &ids[i]);
    }

    for (int i = 0; i < N; i++)
        pthread_join(philosophers[i], NULL);

    for (int i = 0; i < N; i++)
        sem_destroy(&forks[i]);

    return 0;
}
```

7. Implementation of Classical problems (producer consumer,) using Threads and Semaphore.

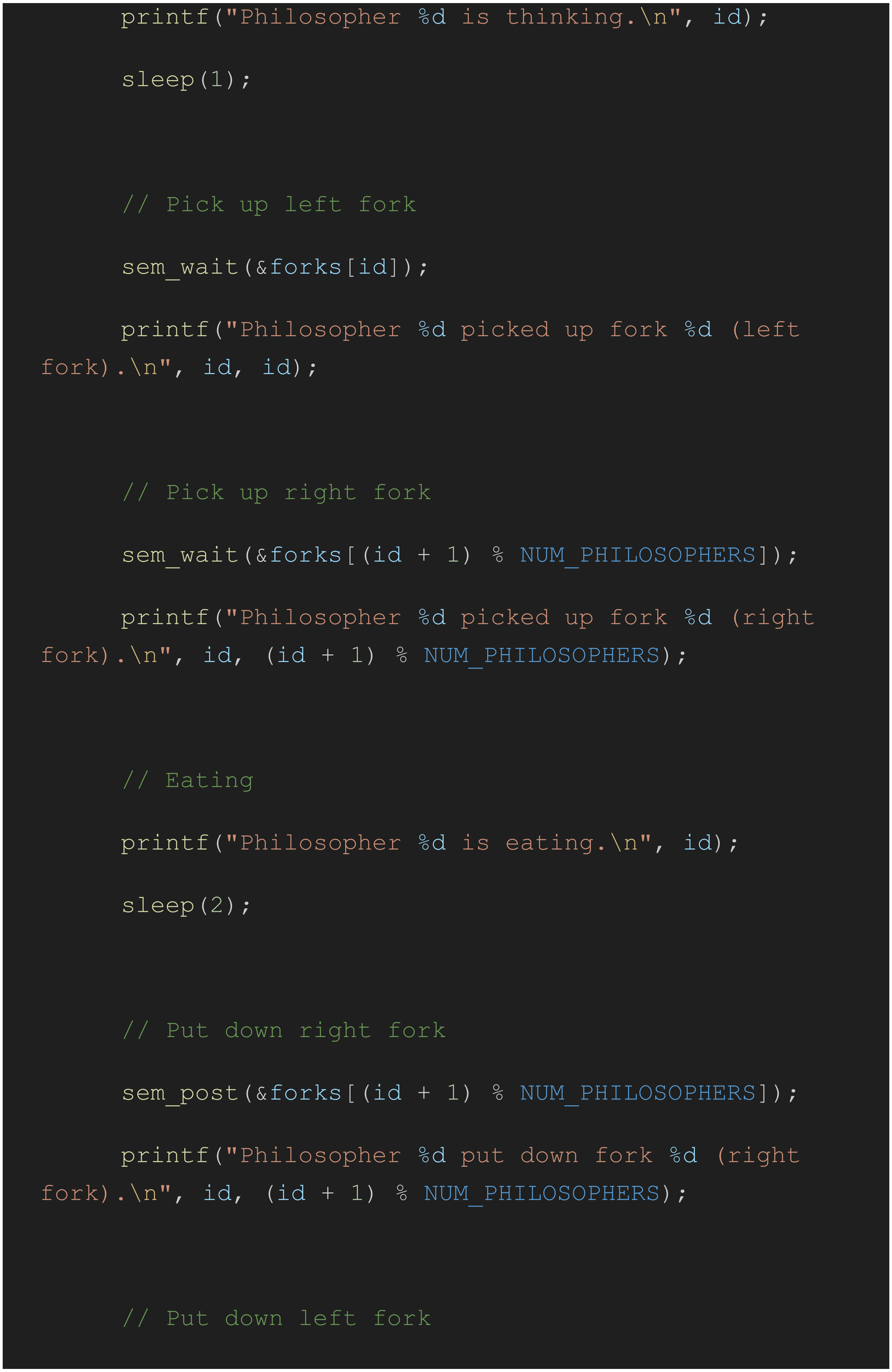
```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <unistd.h>
```



```c
#define N 5

sem_t forks[N];

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

    for (int i = 0; i < 3; i++) {
        printf("Philosopher %d is thinking.\n", id);
        usleep(100000);

        sem_wait(&forks[id]);
        sem_wait(&forks[(id + 1) % N]);

        printf("Philosopher %d is eating.\n", id);
        usleep(100000);

        sem_post(&forks[id]);
        sem_post(&forks[(id + 1) % N]);

        printf("Philosopher %d finished eating.\n", id);
        usleep(100000);
    }
    return NULL;
}

int main() {
    pthread_t philosophers[N];
    int ids[N];

    for (int i = 0; i < N; i++)
        sem_init(&forks[i], 0, 1);

    for (int i = 0; i < N; i++) {
        ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher,
   &ids[i]);
    }

    for (int i = 0; i < N; i++)
        pthread_join(philosophers[i], NULL);

    for (int i = 0; i < N; i++)
        sem_destroy(&forks[i]);

    return 0;
}
```

8. Implementation of Classical problems (dining philosopher) using Threads and Semaphore.

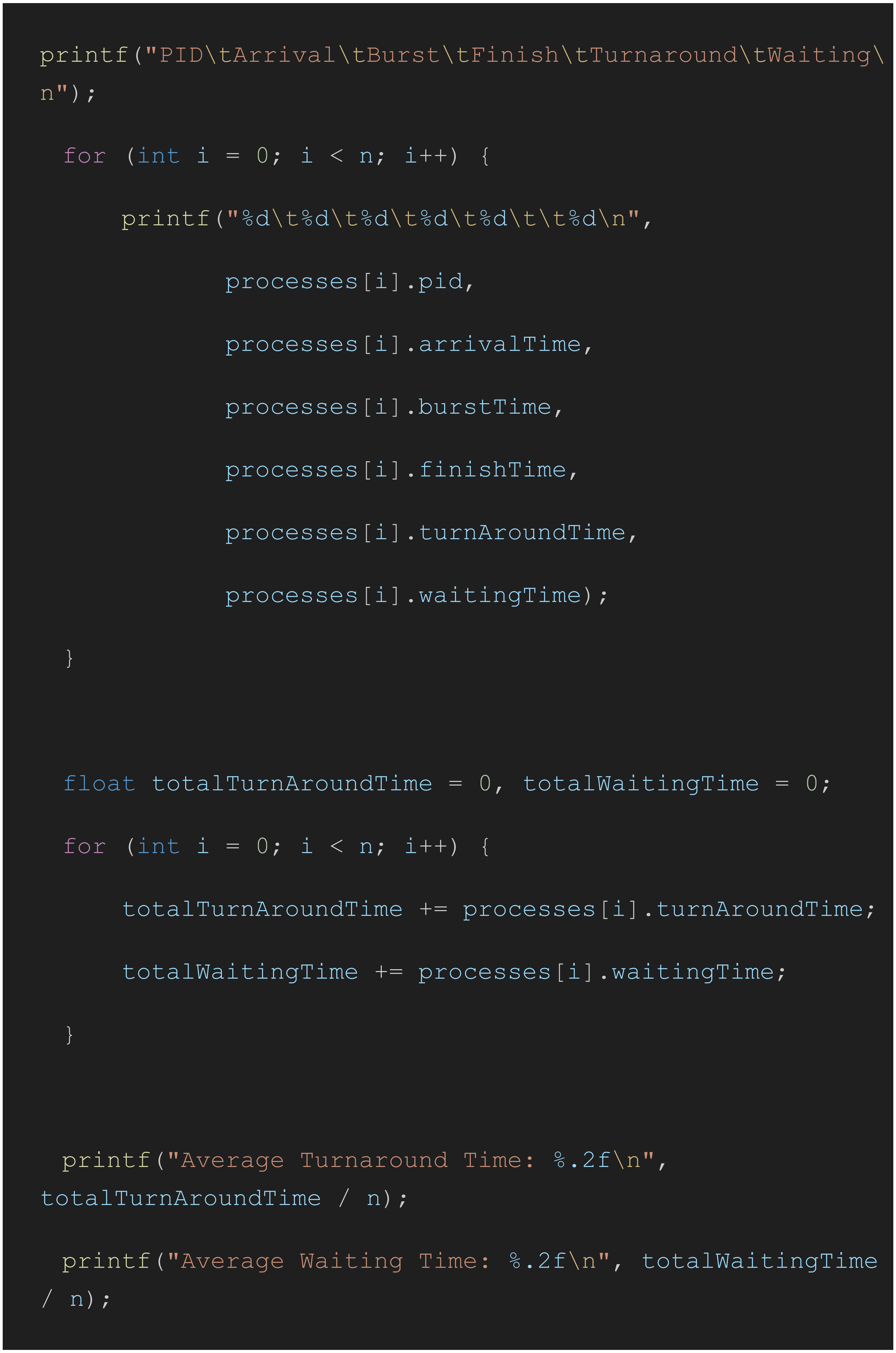
```c
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>



#define NUM_PHILOSOPHERS 5



// Semaphores for each fork

sem_t forks[NUM_PHILOSOPHERS];



// Philosopher function

void *philosopher(void *num) {

    int id = *(int *)num;



    // Each philosopher thinks, picks up forks, eats, and then puts down forks

    while (1) {
```



```c
        printf("Philosopher %d is thinking.\n", id);
        sleep(1);

        // Pick up left fork
        sem_wait(&forks[id]);
        printf("Philosopher %d picked up fork %d (left fork).\n", id, id);

        // Pick up right fork
        sem_wait(&forks[(id + 1) % NUM_PHILOSOPHERS]);
        printf("Philosopher %d picked up fork %d (right fork).\n", id, (id + 1) % NUM_PHILOSOPHERS);

        // Eating
        printf("Philosopher %d is eating.\n", id);
        sleep(2);

        // Put down right fork
        sem_post(&forks[(id + 1) % NUM_PHILOSOPHERS]);
        printf("Philosopher %d put down fork %d (right fork).\n", id, (id + 1) % NUM_PHILOSOPHERS);

        // Put down left fork
        sem_post(&forks[id]);
        printf("Philosopher %d put down fork %d (left
  fork).\n", id, id);

        // Thinking
        printf("Philosopher %d is thinking again.\n", id);
        sleep(1);
    }
    return NULL;
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopher_ids[NUM_PHILOSOPHERS];

    // Initialize semaphores (one for each fork)
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_init(&forks[i], 0, 1);
    }

    // Create philosopher threads
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosopher_ids[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher,
  &philosopher_ids[i]);
    }

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

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

    return 0;
}
```

9. Write a program to compute the finish time, turnaround time and waiting time for the First come First serve

```c
#include <stdio.h>


struct Process {

    int pid;       // Process ID

    int arrivalTime;

    int burstTime;

    int finishTime;

    int turnAroundTime;

    int waitingTime;

};


void calculateTimes(struct Process processes[], int n) {

    int currentTime = 0;


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

        // If the process arrives after the current time, update the current time to the arrival time of the process

        if (currentTime < processes[i].arrivalTime) {

            currentTime = processes[i].arrivalTime;
        }


        // Finish time is the current time plus the burst
  time

        processes[i].finishTime = currentTime +
  processes[i].burstTime;


        // Turnaround time is finish time minus arrival
  time

        processes[i].turnAroundTime =
  processes[i].finishTime - processes[i].arrivalTime;


        // Waiting time is turnaround time minus burst time

        processes[i].waitingTime =
  processes[i].turnAroundTime - processes[i].burstTime;


        // Update current time to finish time of the
  current process

        currentTime = processes[i].finishTime;

    }

}


void displayResults(struct Process processes[], int n) {
```



```c
printf("PID\tArrival\tBurst\tFinish\tTurnaround\tWaiting\n");
  for (int i = 0; i < n; i++) {
      printf("%d\t%d\t%d\t%d\t%d\t\t%d\n",
             processes[i].pid,
             processes[i].arrivalTime,
             processes[i].burstTime,
             processes[i].finishTime,
             processes[i].turnAroundTime,
             processes[i].waitingTime);
  }

  float totalTurnAroundTime = 0, totalWaitingTime = 0;
  for (int i = 0; i < n; i++) {
      totalTurnAroundTime += processes[i].turnAroundTime;
      totalWaitingTime += processes[i].waitingTime;
  }

  printf("Average Turnaround Time: %.2f\n", totalTurnAroundTime / n);
  printf("Average Waiting Time: %.2f\n", totalWaitingTime / n);
```

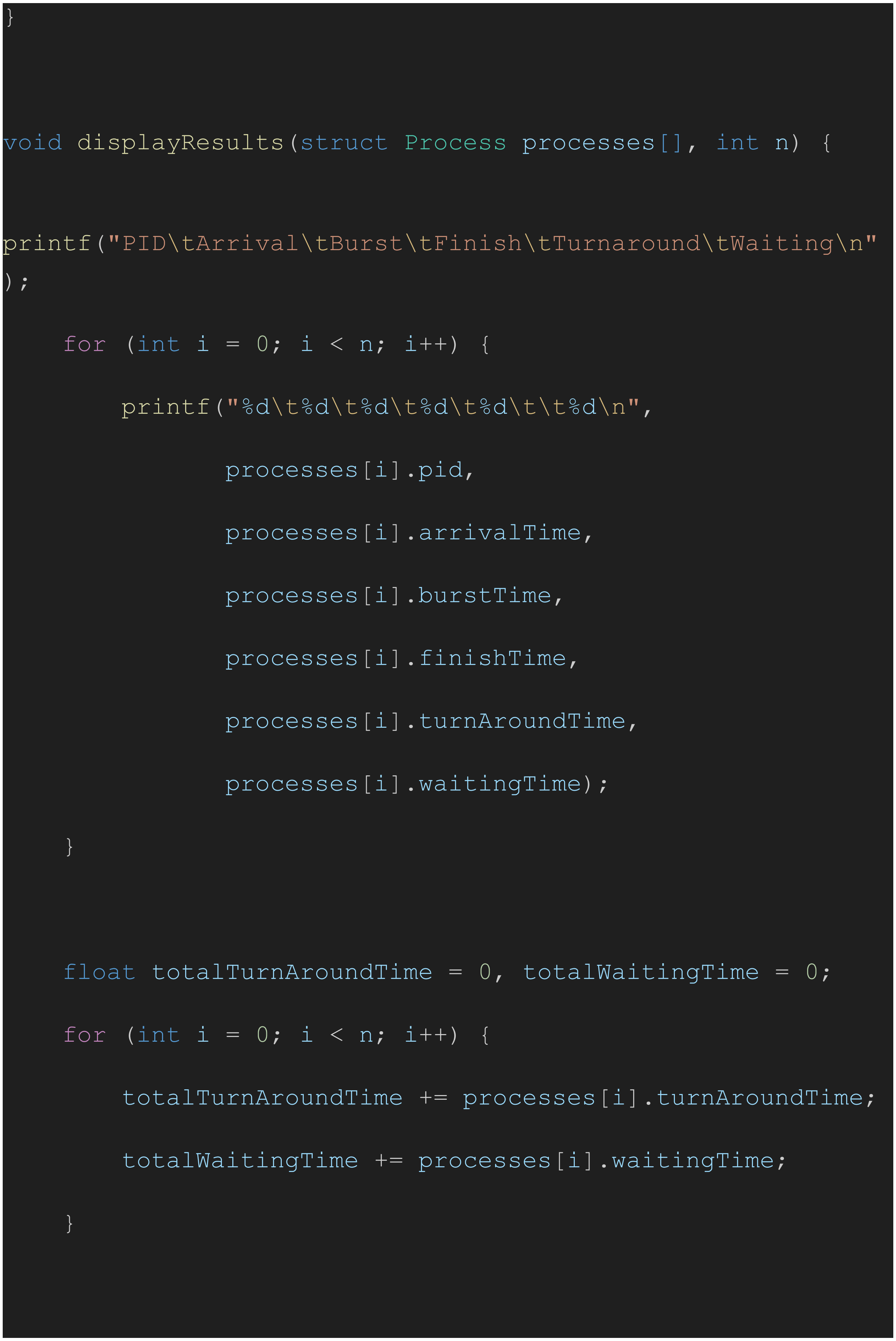
```c
}


int main() {
    int n;

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

    struct Process processes[n];

    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Enter arrival time and burst time for process %d: ", processes[i].pid);
        scanf("%d %d", &processes[i].arrivalTime, &processes[i].burstTime);
    }

    // Sort processes by arrival time (FCFS scheduling)
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (processes[i].arrivalTime > processes[j].arrivalTime) {
                struct Process temp = processes[i];

                processes[i] = processes[j];

                processes[j] = temp;

            }

        }

    }


    calculateTimes(processes, n);

    displayResults(processes, n);


    return 0;

}
```

10. Write a program to compute the finish time, turnaround time and waiting time for the Shortest Job First (Preemptive and Non Preemptive)

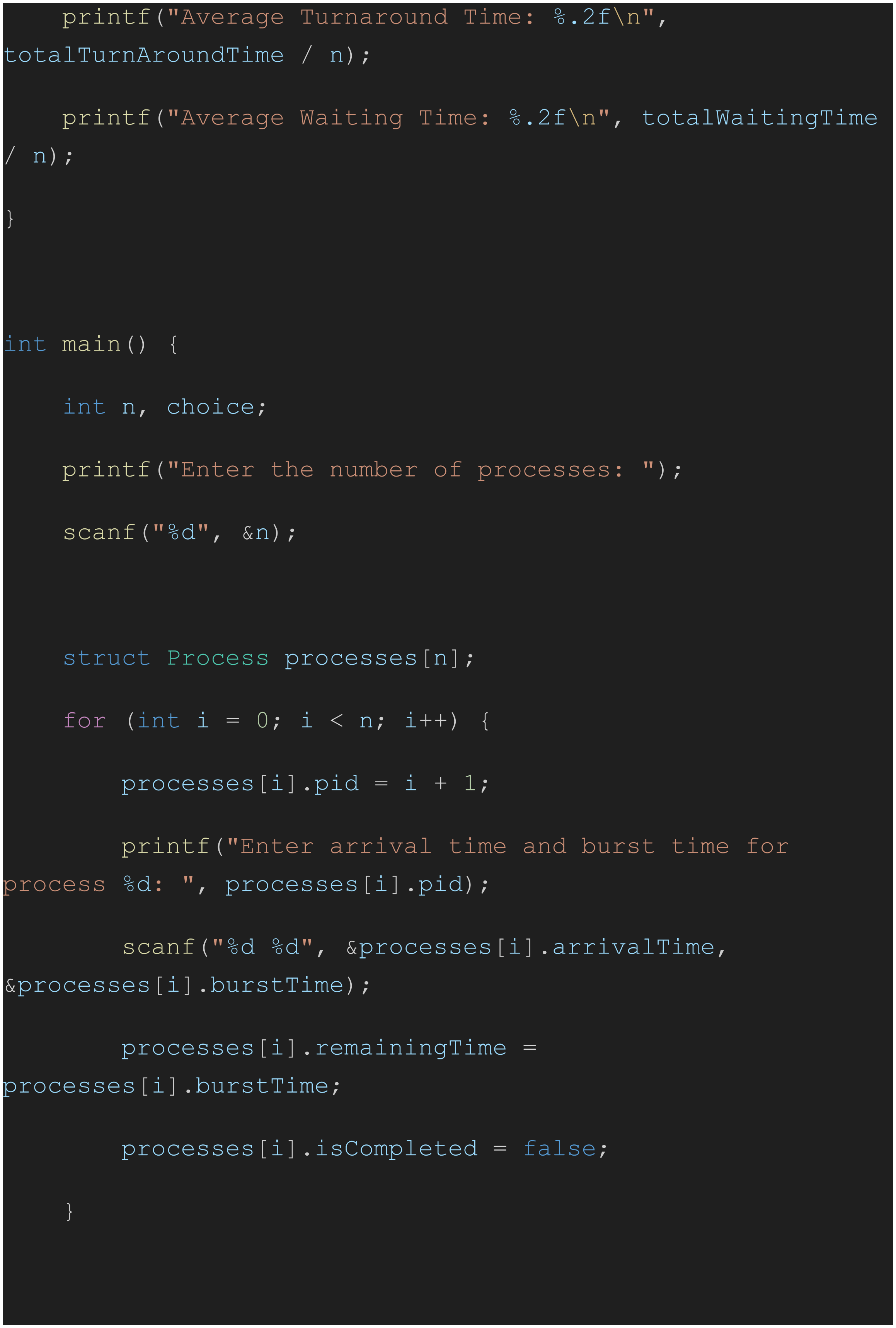
```
#include <stdio.h>

#include <stdbool.h>


struct Process {

    int pid;          // Process ID

    int arrivalTime;
    int burstTime;
    int remainingTime; // For preemptive SJF
    int finishTime;
    int turnAroundTime;
    int waitingTime;
    bool isCompleted;
};

// Function for Non-Preemptive SJF
void sjfNonPreemptive(struct Process processes[], int n) {
    int currentTime = 0, completed = 0;

    while (completed < n) {
        int minIndex = -1;
        int minBurstTime = 1e9;

        // Select the process with the smallest burst time
that has arrived
        for (int i = 0; i < n; i++) {
            if (!processes[i].isCompleted &&
processes[i].arrivalTime <= currentTime &&
                processes[i].burstTime < minBurstTime) {
```

```
                minBurstTime = processes[i].burstTime;
                minIndex = i;
            }
        }

        if (minIndex == -1) {
            currentTime++;
        } else {
            // Calculate the finish time, turnaround time, and waiting time
            processes[minIndex].finishTime = currentTime + processes[minIndex].burstTime;
            processes[minIndex].turnAroundTime = processes[minIndex].finishTime - processes[minIndex].arrivalTime;
            processes[minIndex].waitingTime = processes[minIndex].turnAroundTime - processes[minIndex].burstTime;
            processes[minIndex].isCompleted = true;

            currentTime = processes[minIndex].finishTime;
            completed++;
        }
    }
```

```
}

// Function for Preemptive SJF
void sjfPreemptive(struct Process processes[], int n) {
    int currentTime = 0, completed = 0;
    int minIndex = -1;
    int minRemainingTime = 1e9;

    while (completed < n) {
        minIndex = -1;
        minRemainingTime = 1e9;

        // Select the process with the smallest remaining time that has arrived
        for (int i = 0; i < n; i++) {
            if (processes[i].arrivalTime <= currentTime && !processes[i].isCompleted &&
                processes[i].remainingTime < minRemainingTime) {
                minRemainingTime = processes[i].remainingTime;
                minIndex = i;
            }
        }


        if (minIndex != -1) {

            processes[minIndex].remainingTime--;

            currentTime++;


            // If process is completed

            if (processes[minIndex].remainingTime == 0) {

                processes[minIndex].finishTime =
currentTime;

                processes[minIndex].turnAroundTime =
processes[minIndex].finishTime -
processes[minIndex].arrivalTime;

                processes[minIndex].waitingTime =
processes[minIndex].turnAroundTime -
processes[minIndex].burstTime;

                processes[minIndex].isCompleted = true;

                completed++;

            }

        } else {

            currentTime++;

        }

    }
```



```c
}


void displayResults(struct Process processes[], int n) {

printf("PID\tArrival\tBurst\tFinish\tTurnaround\tWaiting\n"
);
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t\t%d\n",
               processes[i].pid,
               processes[i].arrivalTime,
               processes[i].burstTime,
               processes[i].finishTime,
               processes[i].turnAroundTime,
               processes[i].waitingTime);
    }


    float totalTurnAroundTime = 0, totalWaitingTime = 0;
    for (int i = 0; i < n; i++) {
        totalTurnAroundTime += processes[i].turnAroundTime;
        totalWaitingTime += processes[i].waitingTime;
    }
```

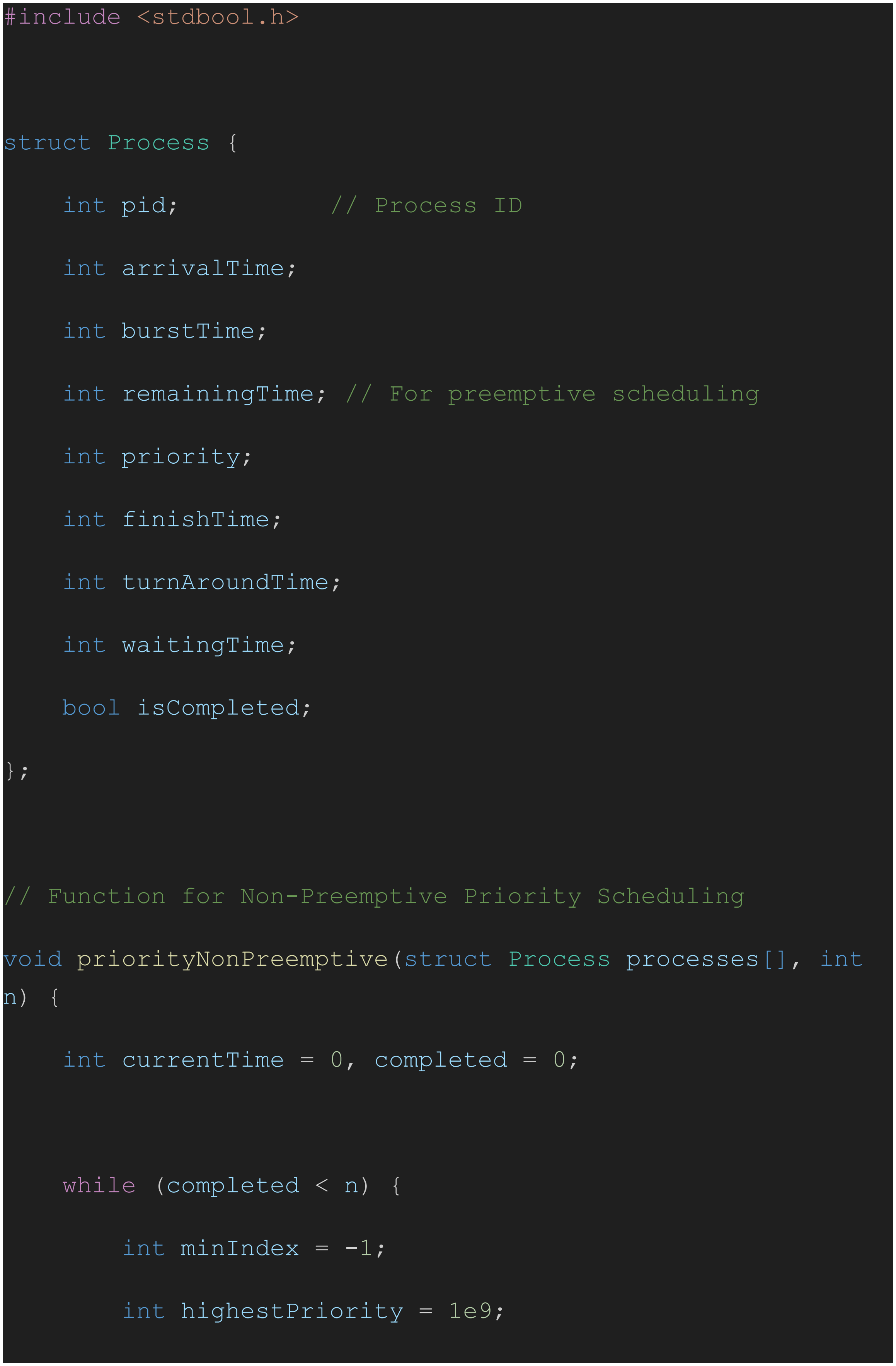


```
    printf("Average Turnaround Time: %.2f\n",
totalTurnAroundTime / n);
    printf("Average Waiting Time: %.2f\n", totalWaitingTime
/ n);
}


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

    struct Process processes[n];
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Enter arrival time and burst time for
process %d: ", processes[i].pid);
        scanf("%d %d", &processes[i].arrivalTime,
&processes[i].burstTime);
        processes[i].remainingTime =
processes[i].burstTime;
        processes[i].isCompleted = false;
    }
    printf("Choose Scheduling:\n1. Non-Preemptive SJF\n2. Preemptive SJF\n");
    scanf("%d", &choice);


    if (choice == 1) {
        sjfNonPreemptive(processes, n);
    } else if (choice == 2) {
        sjfPreemptive(processes, n);
    } else {
        printf("Invalid choice!\n");
        return 0;
    }


    displayResults(processes, n);
    return 0;
}
```

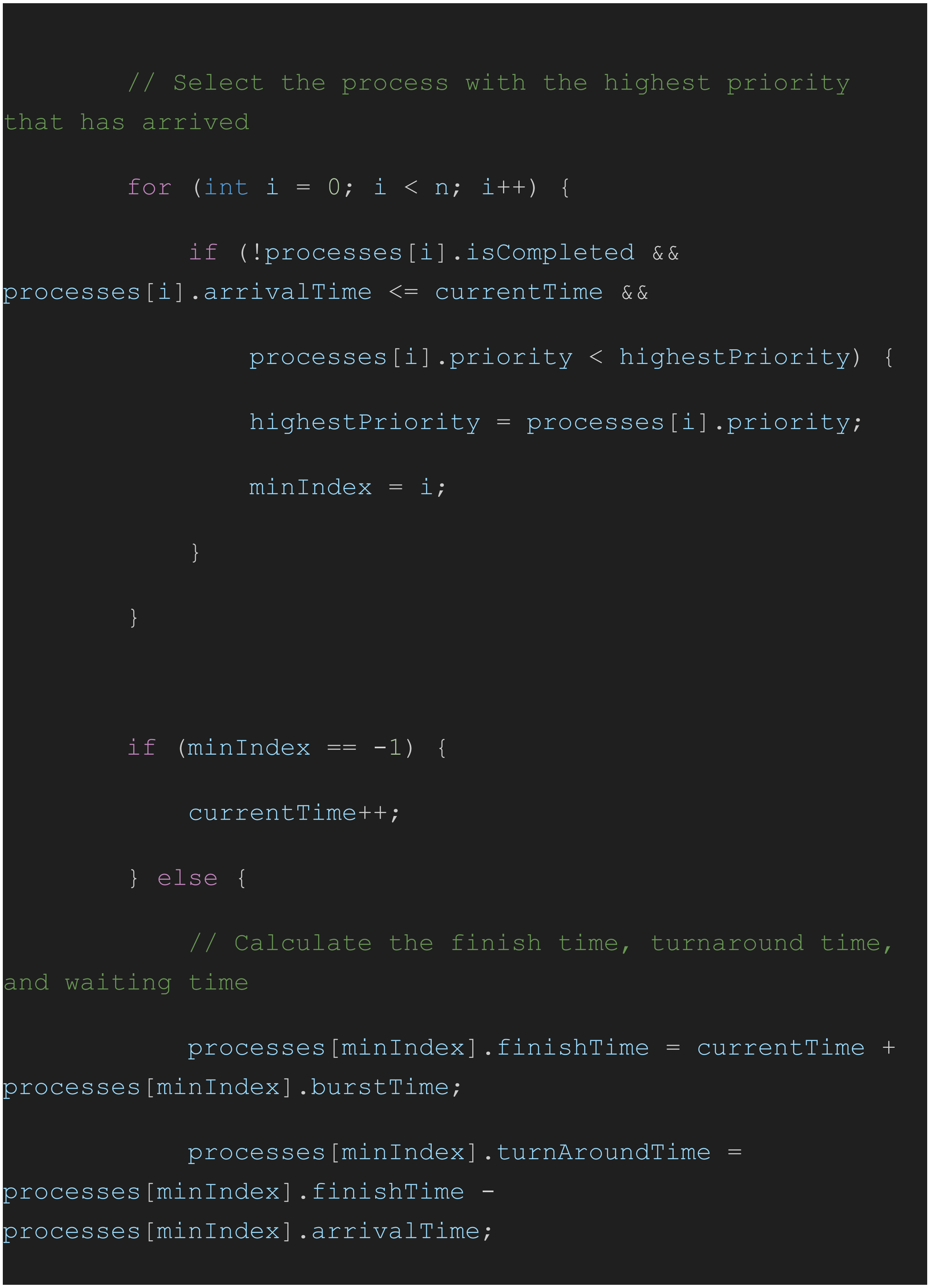
11. Write a program to compute the finish time, turnaround time and waiting time for the

Priority (Preemptive and Non Preemptive)

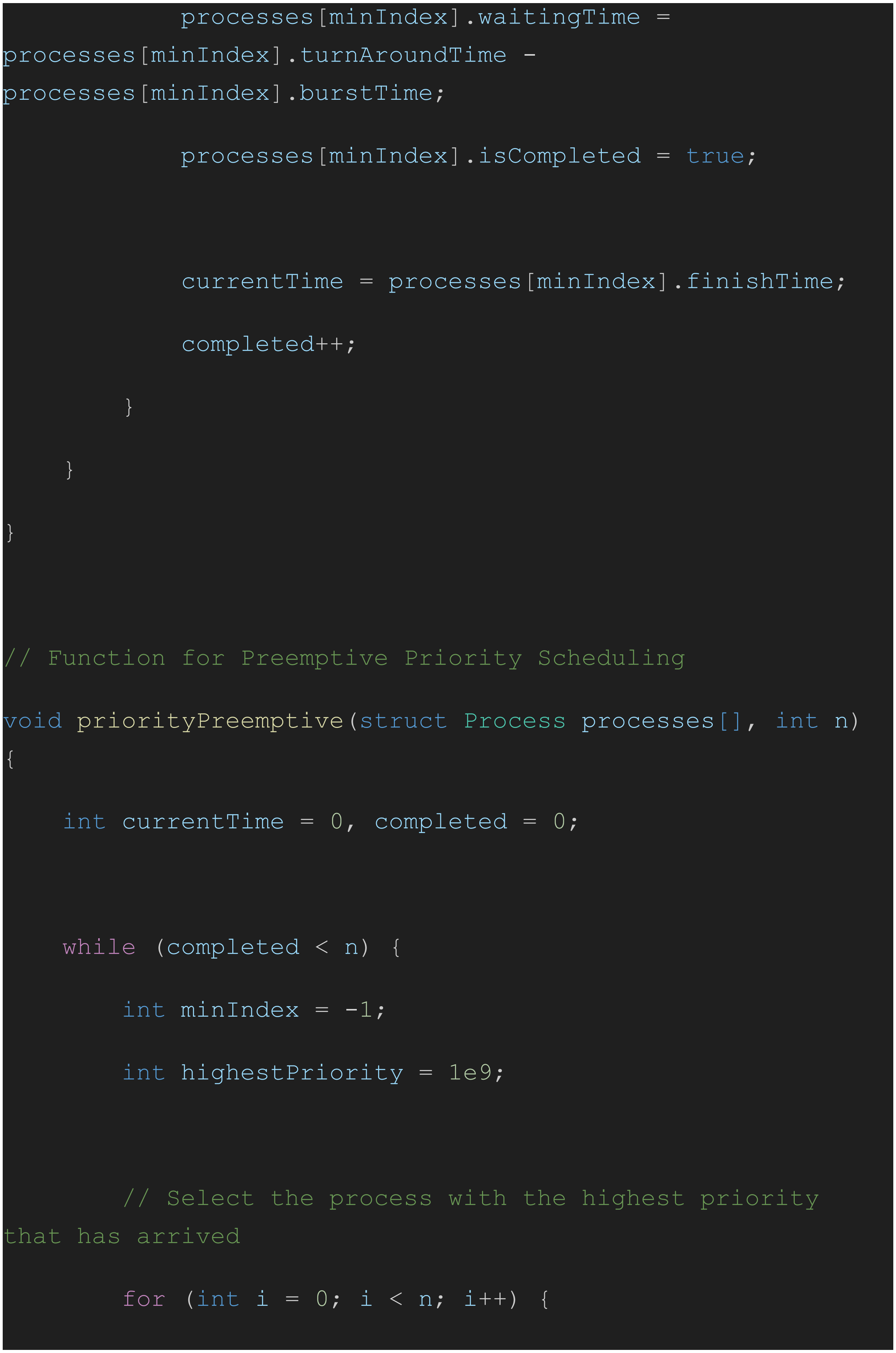
```
#include <stdio.h>
```



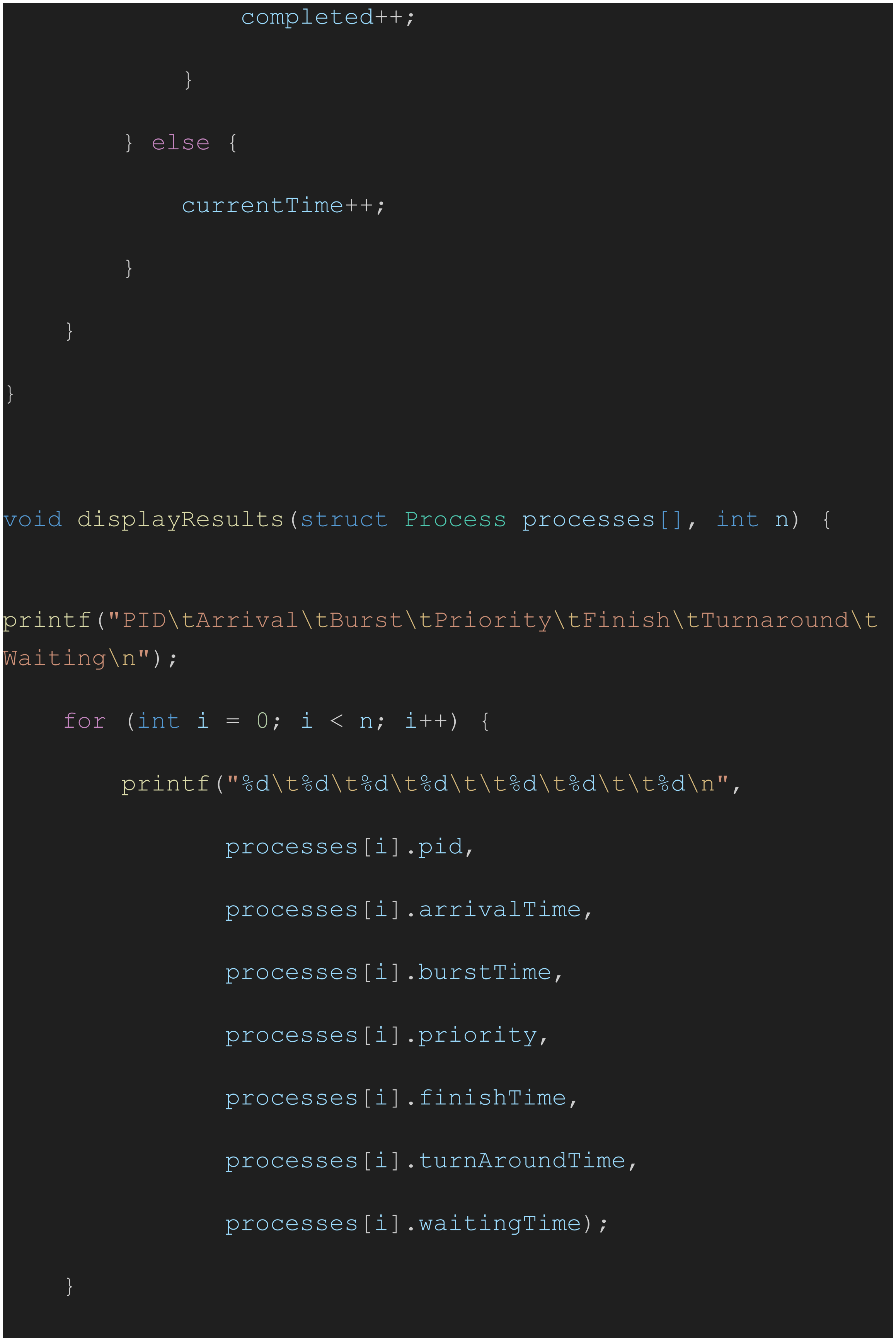
```c
#include <stdbool.h>

struct Process {
    int pid;          // Process ID
    int arrivalTime;
    int burstTime;
    int remainingTime; // For preemptive scheduling
    int priority;
    int finishTime;
    int turnAroundTime;
    int waitingTime;
    bool isCompleted;
};

// Function for Non-Preemptive Priority Scheduling
void priorityNonPreemptive(struct Process processes[], int
n) {
    int currentTime = 0, completed = 0;

    while (completed < n) {
        int minIndex = -1;
        int highestPriority = 1e9;
```



```java
        // Select the process with the highest priority that has arrived
        for (int i = 0; i < n; i++) {
            if (!processes[i].isCompleted && processes[i].arrivalTime <= currentTime &&
                processes[i].priority < highestPriority) {
                highestPriority = processes[i].priority;
                minIndex = i;
            }
        }

        if (minIndex == -1) {
            currentTime++;
        } else {
            // Calculate the finish time, turnaround time, and waiting time
            processes[minIndex].finishTime = currentTime + processes[minIndex].burstTime;
            processes[minIndex].turnAroundTime = processes[minIndex].finishTime - processes[minIndex].arrivalTime;
```



```
            processes[minIndex].waitingTime =
processes[minIndex].turnAroundTime -
processes[minIndex].burstTime;
            processes[minIndex].isCompleted = true;

            currentTime = processes[minIndex].finishTime;
            completed++;
        }
    }
}

// Function for Preemptive Priority Scheduling
void priorityPreemptive(struct Process processes[], int n)
{
    int currentTime = 0, completed = 0;

    while (completed < n) {
        int minIndex = -1;
        int highestPriority = 1e9;

        // Select the process with the highest priority
that has arrived
        for (int i = 0; i < n; i++) {
            if (processes[i].arrivalTime <= currentTime &&
!processes[i].isCompleted &&
                processes[i].priority < highestPriority) {
                highestPriority = processes[i].priority;
                minIndex = i;
            }
        }

        if (minIndex != -1) {
            // Process one unit of the burst time
            processes[minIndex].remainingTime--;
            currentTime++;

            // If process is completed
            if (processes[minIndex].remainingTime == 0) {
                processes[minIndex].finishTime =
currentTime;
                processes[minIndex].turnAroundTime =
processes[minIndex].finishTime -
processes[minIndex].arrivalTime;
                processes[minIndex].waitingTime =
processes[minIndex].turnAroundTime -
processes[minIndex].burstTime;
                processes[minIndex].isCompleted = true;
```

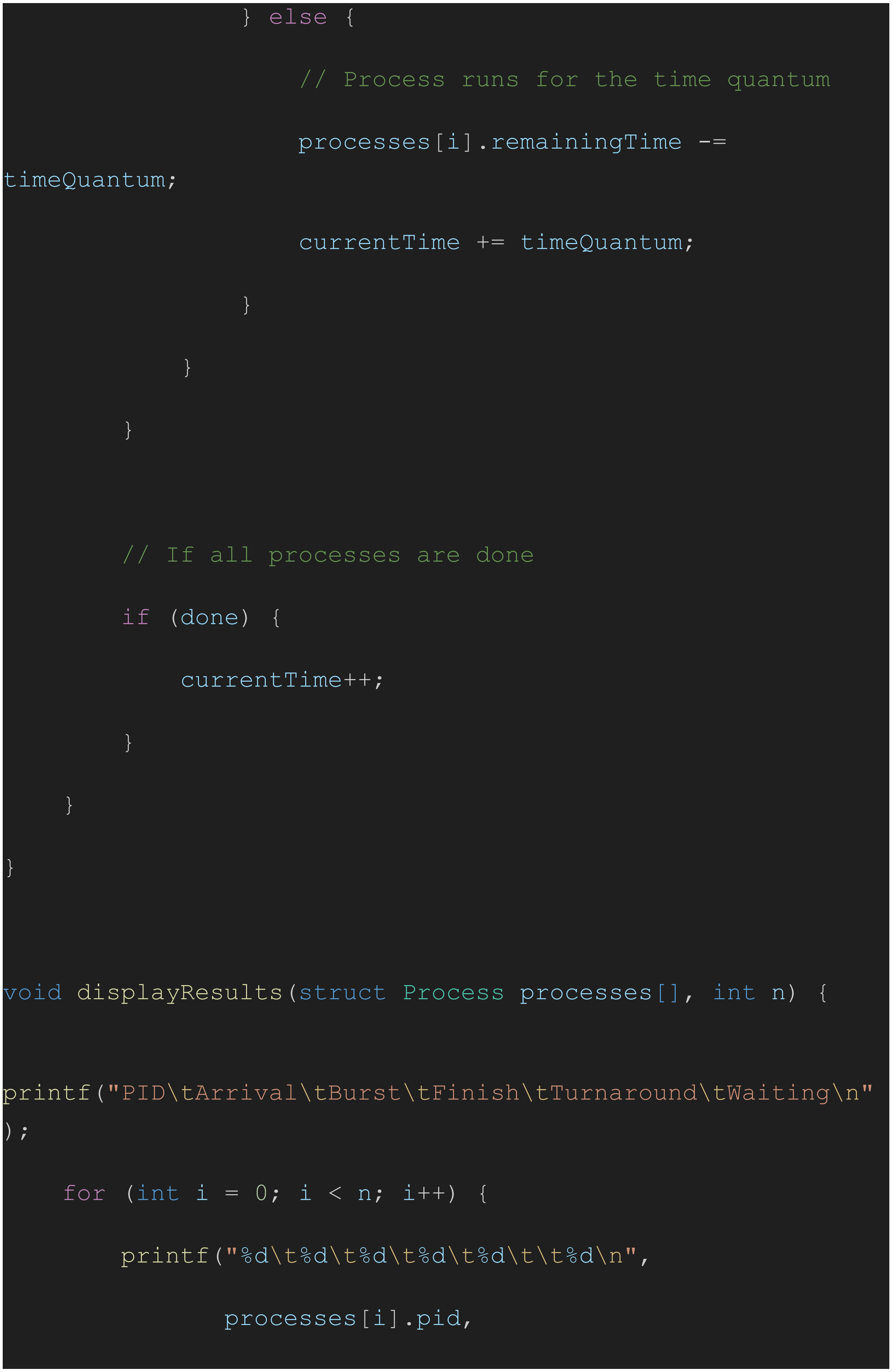


```c
                completed++;
            }
        } else {
            currentTime++;
        }
    }
}

void displayResults(struct Process processes[], int n) {
printf("PID\tArrival\tBurst\tPriority\tFinish\tTurnaround\tWaiting\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t\t%d\t%d\t\t%d\n",
               processes[i].pid,
               processes[i].arrivalTime,
               processes[i].burstTime,
               processes[i].priority,
               processes[i].finishTime,
               processes[i].turnAroundTime,
               processes[i].waitingTime);
    }
    float totalTurnAroundTime = 0, totalWaitingTime = 0;
    for (int i = 0; i < n; i++) {
        totalTurnAroundTime += processes[i].turnAroundTime;
        totalWaitingTime += processes[i].waitingTime;
    }

    printf("Average Turnaround Time: %.2f\n", totalTurnAroundTime / n);
    printf("Average Waiting Time: %.2f\n", totalWaitingTime / n);
}

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

    struct Process processes[n];
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Enter arrival time, burst time, and priority for process %d: ", processes[i].pid);
        scanf("%d %d %d", &processes[i].arrivalTime, 
&processes[i].burstTime, &processes[i].priority);
        processes[i].remainingTime = 
processes[i].burstTime;
        processes[i].isCompleted = false;
    }

    printf("Choose Scheduling:\n1. Non-Preemptive 
Priority\n2. Preemptive Priority\n");
    scanf("%d", &choice);

    if (choice == 1) {
        priorityNonPreemptive(processes, n);
    } else if (choice == 2) {
        priorityPreemptive(processes, n);
    } else {
        printf("Invalid choice!\n");
        return 0;
    }

    displayResults(processes, n);
    return 0;
}
```

12. Write a program to compute the finish time, turnaround time and waiting time for the

Round robin

```
#include <stdio.h>


struct Process {
    int pid;           // Process ID
    int arrivalTime;   // Arrival time
    int burstTime;     // Burst time
    int remainingTime; // Remaining burst time
    int finishTime;    // Finish time
    int turnAroundTime; // Turnaround time
    int waitingTime;   // Waiting time
};


void roundRobin(struct Process processes[], int n, int quantum) {
    int currentTime = 0;
    int completed = 0;
    int timeQuantum = quantum;
    while (completed < n) {
        int done = 1;

        for (int i = 0; i < n; i++) {
            // Check if process has remaining time and has arrived
            if (processes[i].remainingTime > 0 && processes[i].arrivalTime <= currentTime) {
                done = 0;

                // If remaining time is less than or equal to time quantum, process will finish
                if (processes[i].remainingTime <= timeQuantum) {
                    currentTime += processes[i].remainingTime;
                    processes[i].finishTime = currentTime;
                    processes[i].turnAroundTime = processes[i].finishTime - processes[i].arrivalTime;
                    processes[i].waitingTime = processes[i].turnAroundTime - processes[i].burstTime;
                    processes[i].remainingTime = 0;
                    completed++;
```

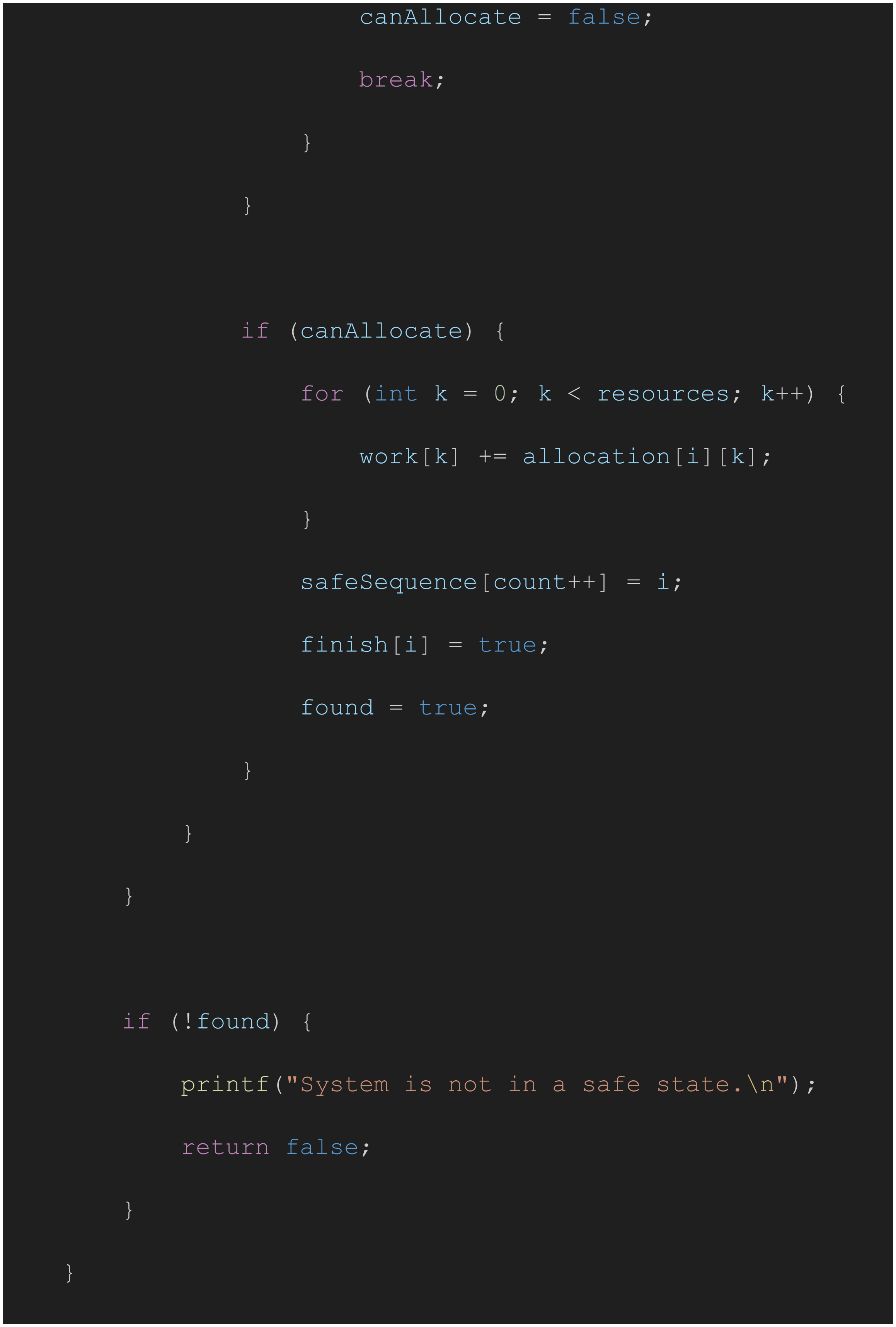


```
                } else {
                    // Process runs for the time quantum
                    processes[i].remainingTime -=
timeQuantum;
                    currentTime += timeQuantum;
                }
            }
        }

        // If all processes are done
        if (done) {
            currentTime++;
        }
    }
}

void displayResults(struct Process processes[], int n) {

printf("PID\tArrival\tBurst\tFinish\tTurnaround\tWaiting\n"
);
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\t\t%d\n",
                processes[i].pid,
               processes[i].arrivalTime,
               processes[i].burstTime,
               processes[i].finishTime,
               processes[i].turnAroundTime,
               processes[i].waitingTime);
    }

    float totalTurnAroundTime = 0, totalWaitingTime = 0;
    for (int i = 0; i < n; i++) {
        totalTurnAroundTime += processes[i].turnAroundTime;
        totalWaitingTime += processes[i].waitingTime;
    }

    printf("Average Turnaround Time: %.2f\n", totalTurnAroundTime / n);
    printf("Average Waiting Time: %.2f\n", totalWaitingTime / n);
}


int main() {
    int n, quantum;
    printf("Enter the number of processes: ");

    scanf("%d", &n);

    struct Process processes[n];



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

        processes[i].pid = i + 1;

        printf("Enter arrival time and burst time for
process %d: ", processes[i].pid);

        scanf("%d %d", &processes[i].arrivalTime,
&processes[i].burstTime);

        processes[i].remainingTime =
processes[i].burstTime;

    }



    printf("Enter the time quantum: ");

    scanf("%d", &quantum);



    roundRobin(processes, n, quantum);

    displayResults(processes, n);



    return 0;

}
```

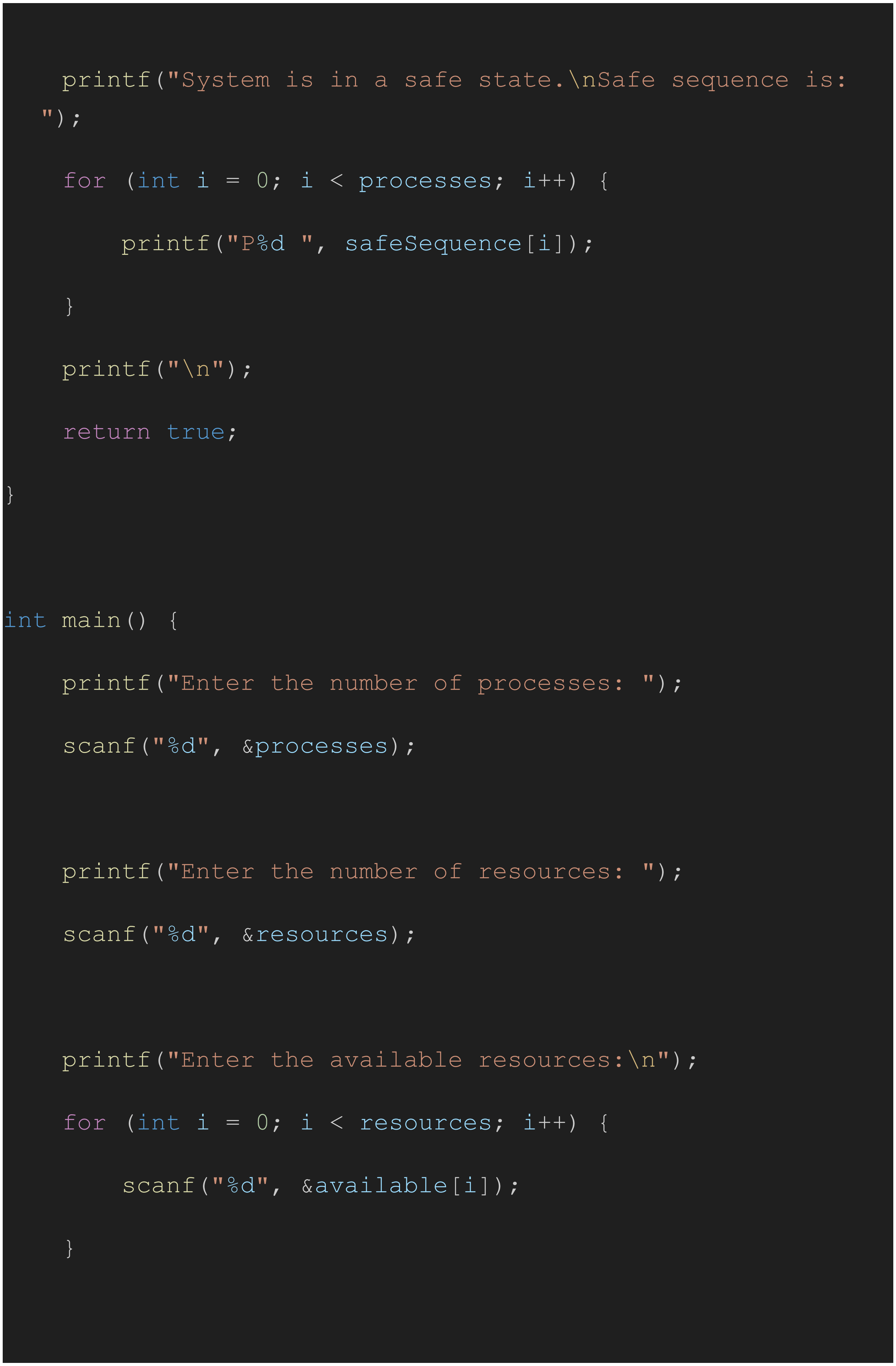
13. Write a program to check whether given system is in safe state or not using Banker’s Deadlock Avoidance algorithm.

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

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

int processes, resources;
int available[MAX_RESOURCES];
int max[MAX_PROCESSES][MAX_RESOURCES];
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];

void calculateNeed() {
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}
```

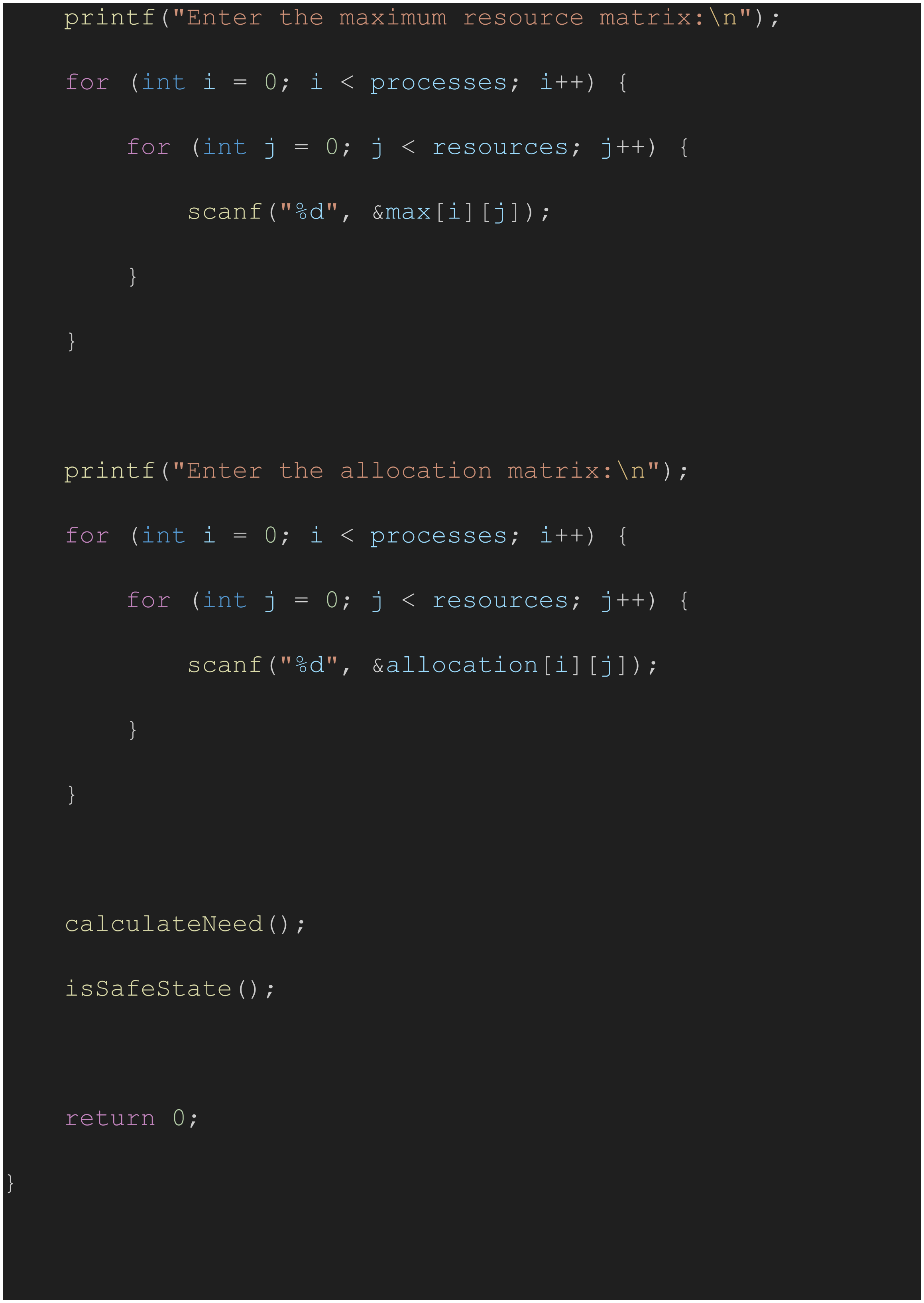
```
bool isSafeState() {
    int work[MAX_RESOURCES];
    bool finish[MAX_PROCESSES] = {false};
    int safeSequence[MAX_PROCESSES];
    int count = 0;

    // Initialize work as a copy of available resources
    for (int i = 0; i < resources; i++) {
        work[i] = available[i];
    }

    while (count < processes) {
        bool found = false;

        for (int i = 0; i < processes; i++) {
            if (!finish[i]) {
                bool canAllocate = true;

                for (int j = 0; j < resources; j++) {
                    if (need[i][j] > work[j]) {
```



```c
                        canAllocate = false;
                        break;
                    }
                }

                if (canAllocate) {
                    for (int k = 0; k < resources; k++) {
                        work[k] += allocation[i][k];
                    }
                    safeSequence[count++] = i;
                    finish[i] = true;
                    found = true;
                }
            }
        }

        if (!found) {
            printf("System is not in a safe state.\n");
            return false;
        }
    }
```



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

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

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

    printf("Enter the available resources:\n");
    for (int i = 0; i < resources; i++) {
        scanf("%d", &available[i]);
    }
```

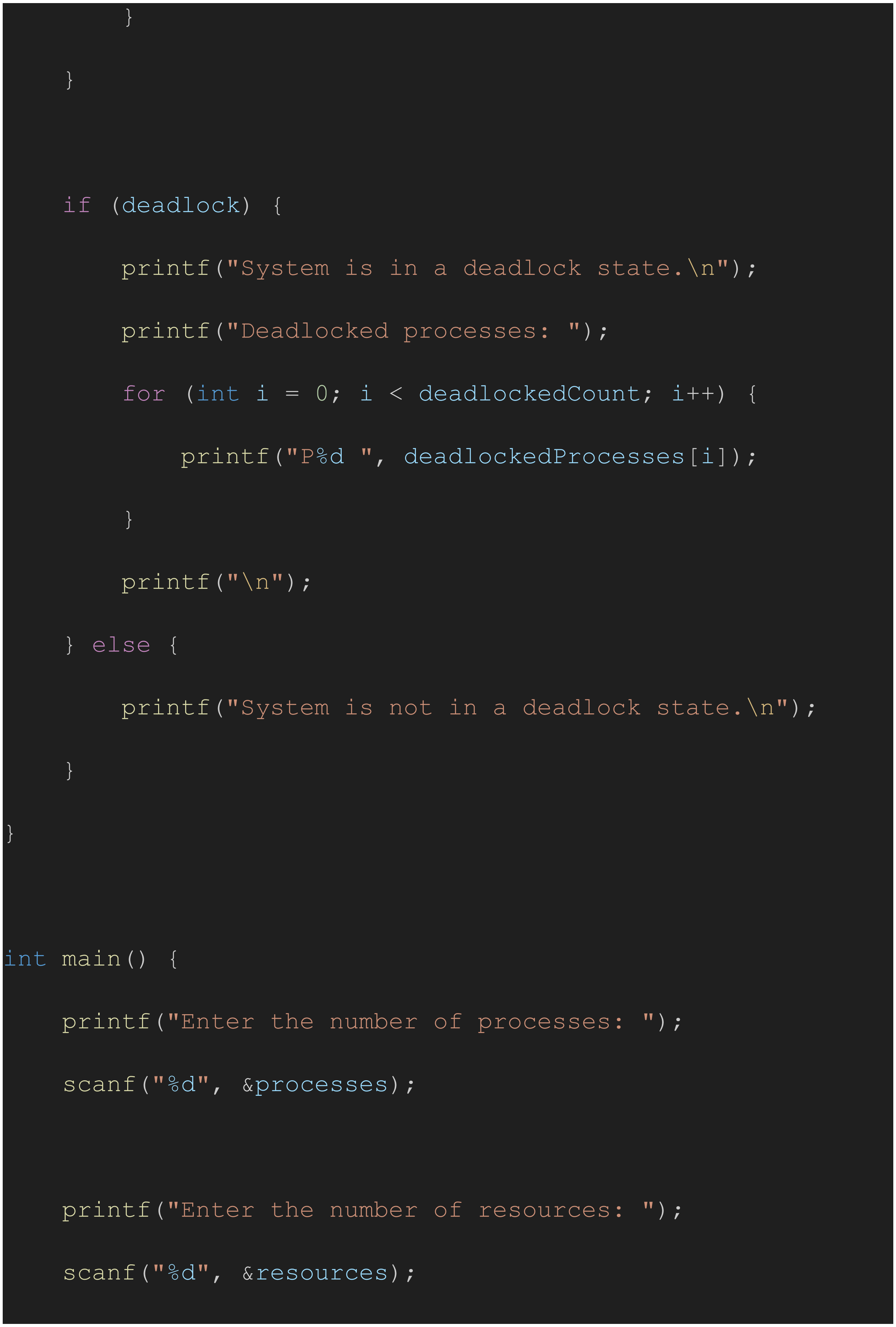


```c
    printf("Enter the maximum resource matrix:\n");
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            scanf("%d", &max[i][j]);
        }
    }

    printf("Enter the allocation matrix:\n");
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            scanf("%d", &allocation[i][j]);
        }
    }

    calculateNeed();
    isSafeState();

    return 0;
}
```

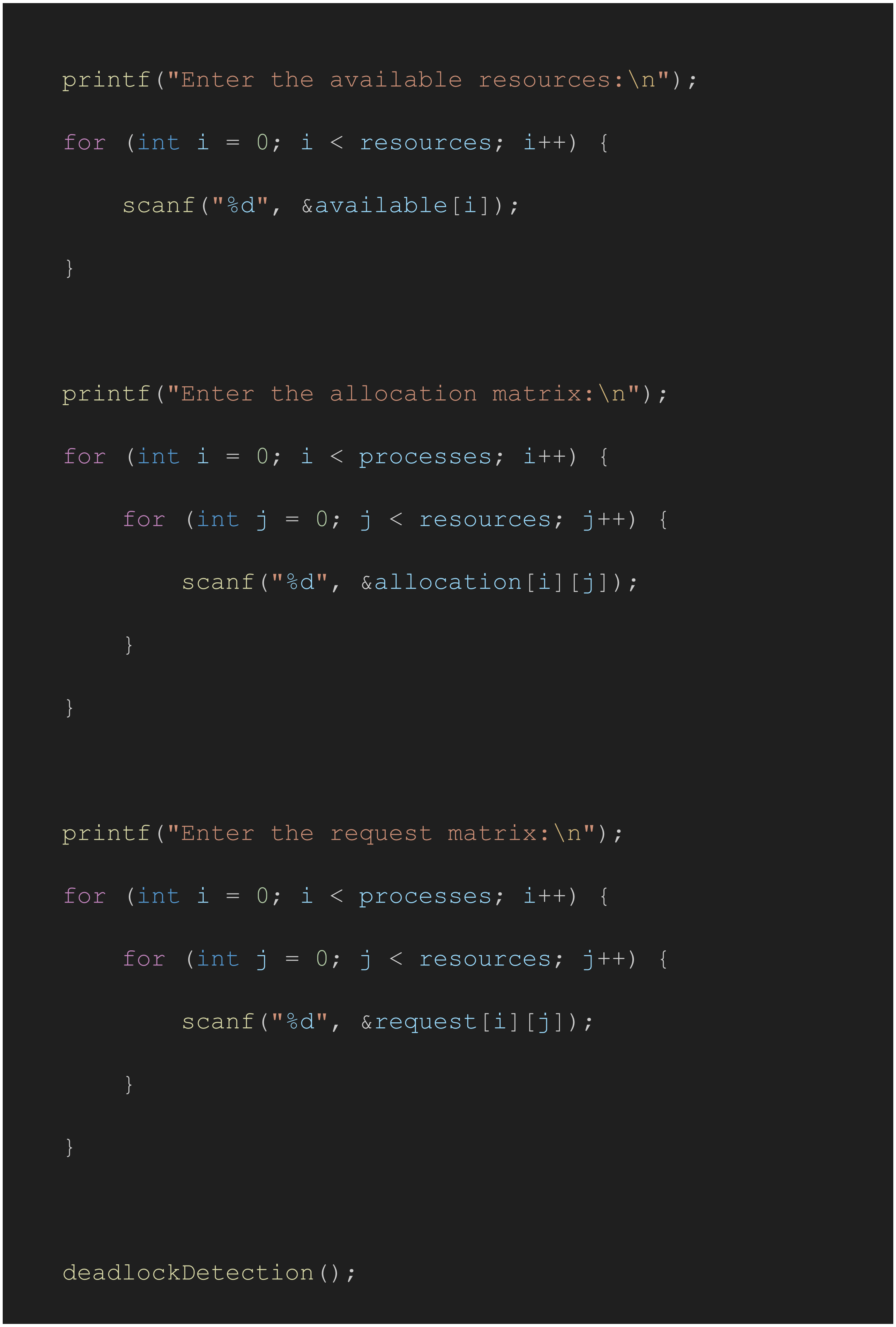
14. Write a program for Deadlock detection algorithm

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

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

int processes, resources;
int available[MAX_RESOURCES];
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int request[MAX_PROCESSES][MAX_RESOURCES];

void deadlockDetection() {
    bool finish[MAX_PROCESSES] = {false};
    int work[MAX_RESOURCES];

    // Initialize work as a copy of available resources
    for (int i = 0; i < resources; i++) {
        work[i] = available[i];
    }

    bool deadlock = false;
    int deadlockedProcesses[MAX_PROCESSES];
    int deadlockedCount = 0;

    for (int count = 0; count < processes; count++) {
        bool found = false;

        for (int i = 0; i < processes; i++) {
            if (!finish[i]) {
                bool canProceed = true;

                // Check if the process's request can be
satisfied
                for (int j = 0; j < resources; j++) {
                    if (request[i][j] > work[j]) {
                        canProceed = false;
                        break;
                    }
                }

                // If the request can be satisfied,
allocate resources temporarily
```

```
              if (canProceed) {

                  for (int j = 0; j < resources; j++) {

                      work[j] += allocation[i][j];

                  }

                  finish[i] = true;

                  found = true;

              }

          }

      }


      // If no process could proceed in this round, break
out

      if (!found) {

          break;

      }

  }


  // Check for processes still marked as unfinished

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

      if (!finish[i]) {

          deadlockedProcesses[deadlockedCount++] = i;

          deadlock = true;
```



```
        }
    }

    if (deadlock) {
        printf("System is in a deadlock state.\n");
        printf("Deadlocked processes: ");
        for (int i = 0; i < deadlockedCount; i++) {
            printf("P%d ", deadlockedProcesses[i]);
        }
        printf("\n");
    } else {
        printf("System is not in a deadlock state.\n");
    }
}

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

    printf("Enter the number of resources: ");
    scanf("%d", &resources);
```



```c
printf("Enter the available resources:\n");
for (int i = 0; i < resources; i++) {
    scanf("%d", &available[i]);
}

printf("Enter the allocation matrix:\n");
for (int i = 0; i < processes; i++) {
    for (int j = 0; j < resources; j++) {
        scanf("%d", &allocation[i][j]);
    }
}

printf("Enter the request matrix:\n");
for (int i = 0; i < processes; i++) {
    for (int j = 0; j < resources; j++) {
        scanf("%d", &request[i][j]);
    }
}

deadlockDetection();
```

```
    return 0;

}
```

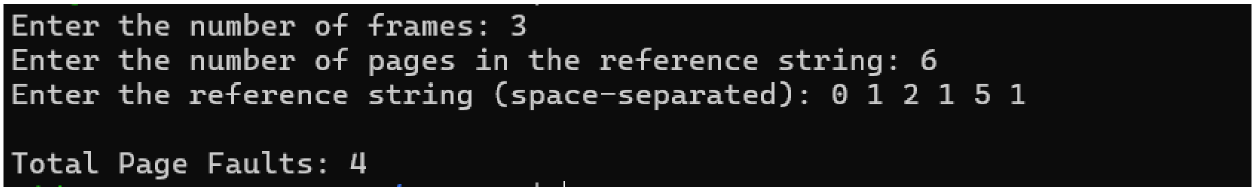
15. Write a program to calculate the number of page faults for a reference string for the FIFO page replacement algorithms:

```
#include <stdio.h>


#define MAX_FRAMES 10


int isPageInFrames(int frames[], int frameCount, int page) {

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

        if (frames[i] == page) {

            return 1; // Page found in frames

        }

    }

    return 0; // Page not found

}


int main() {

    int frameCount, pageCount;

    int pageFaults = 0;

     int nextFrameToReplace = 0; // To keep track of which frame to
replace next

```

```
    // Input: Number of frames and number of pages in the reference
  string
    printf("Enter the number of frames: ");
    scanf("%d", &frameCount);

    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &pageCount);

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

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

    // Processing each page in the reference string
    for (int i = 0; i < pageCount; i++) {
        int currentPage = pages[i];

        // Check if the current page is already in the frames
```

```c
        if (!isPageInFrames(frames, frameCount, currentPage)) {
            // Page fault occurs as the page is not in frames
            frames[nextFrameToReplace] = currentPage; // Replace the page at nextFrameToReplace
            pageFaults++; // Increment page faults
            nextFrameToReplace = (nextFrameToReplace + 1) % frameCount; // Move to the next frame
        }
    }

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

    return 0;
}
```

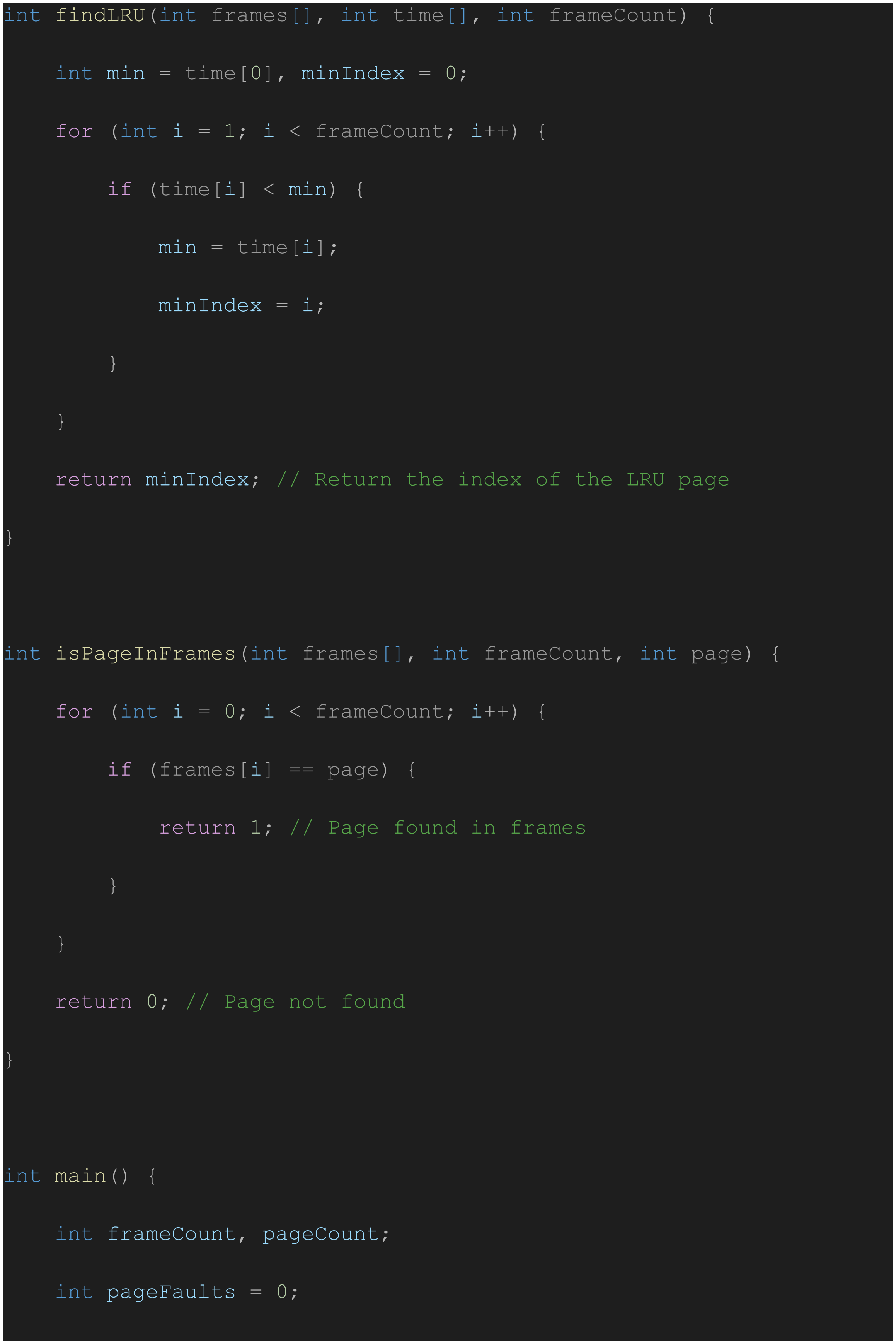
Output:-



```
Enter the number of frames: 3
Enter the number of pages in the reference string: 6
Enter the reference string (space-separated): 0 1 2 1 5 1

Total Page Faults: 4
```

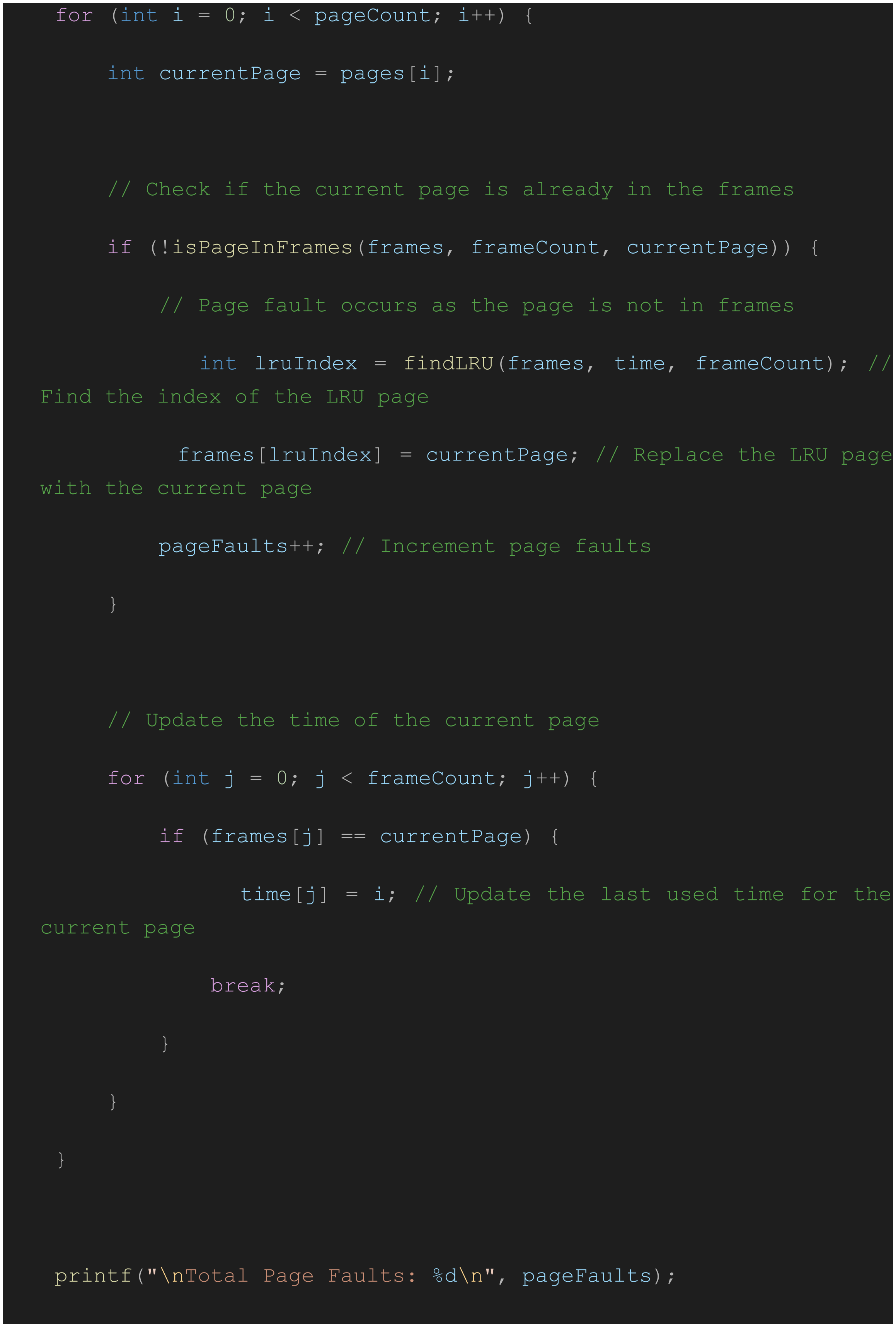
16. Write a program to calculate the number of page faults for a reference string for the LRU page replacement algorithms:

```c
#include <stdio.h>

#define MAX_FRAMES 10

```



```
int findLRU(int frames[], int time[], int frameCount) {
    int min = time[0], minIndex = 0;
    for (int i = 1; i < frameCount; i++) {
        if (time[i] < min) {
            min = time[i];
            minIndex = i;
        }
    }
    return minIndex; // Return the index of the LRU page
}

int isPageInFrames(int frames[], int frameCount, int page) {
    for (int i = 0; i < frameCount; i++) {
        if (frames[i] == page) {
            return 1; // Page found in frames
        }
    }
    return 0; // Page not found
}

int main() {
    int frameCount, pageCount;
    int pageFaults = 0;
```

```
  // Input: Number of frames and number of pages in the reference
string
 printf("Enter the number of frames: ");
 scanf("%d", &frameCount);


 printf("Enter the number of pages in the reference string: ");
 scanf("%d", &pageCount);


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


 int frames[MAX_FRAMES];
  int time[MAX_FRAMES]; // Array to keep track of the last used
time of each frame
 for (int i = 0; i < frameCount; i++) {
     frames[i] = -1; // Initialize frames as empty
     time[i] = 0; // Initialize the last used time
 }


 // Processing each page in the reference string
```



```
    for (int i = 0; i < pageCount; i++) {
        int currentPage = pages[i];


        // Check if the current page is already in the frames
        if (!isPageInFrames(frames, frameCount, currentPage)) {
            // Page fault occurs as the page is not in frames
              int lruIndex = findLRU(frames, time, frameCount); // Find the index of the LRU page
             frames[lruIndex] = currentPage; // Replace the LRU page with the current page
            pageFaults++; // Increment page faults
        }


        // Update the time of the current page
        for (int j = 0; j < frameCount; j++) {
            if (frames[j] == currentPage) {
                time[j] = i; // Update the last used time for the current page
                break;
            }
        }
    }


    printf("\nTotal Page Faults: %d\n", pageFaults);
    return 0;
}
```

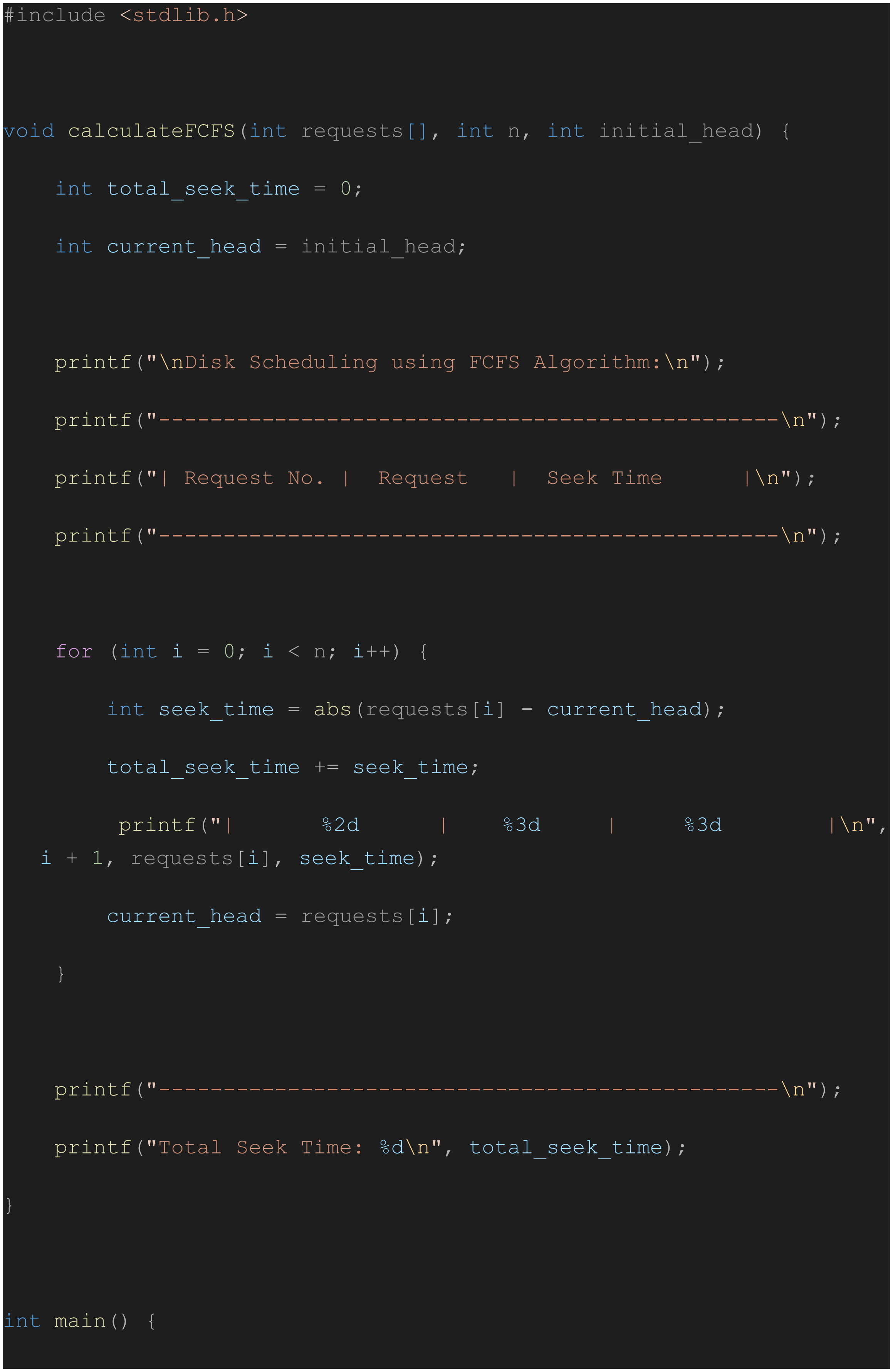
17. Write a program to calculate the number of page faults for a reference string for the Optimal page replacement algorithms:

```
#include <stdio.h>

#define MAX_FRAMES 10

int findOptimal(int frames[], int frameCount, int pages[], int pageCount, int currentIndex) {
    int farthest = currentIndex, indexToReplace = -1;

    for (int i = 0; i < frameCount; i++) {
        int j;
        for (j = currentIndex; j < pageCount; j++) {
            if (frames[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    indexToReplace = i;
                }
                break;
```

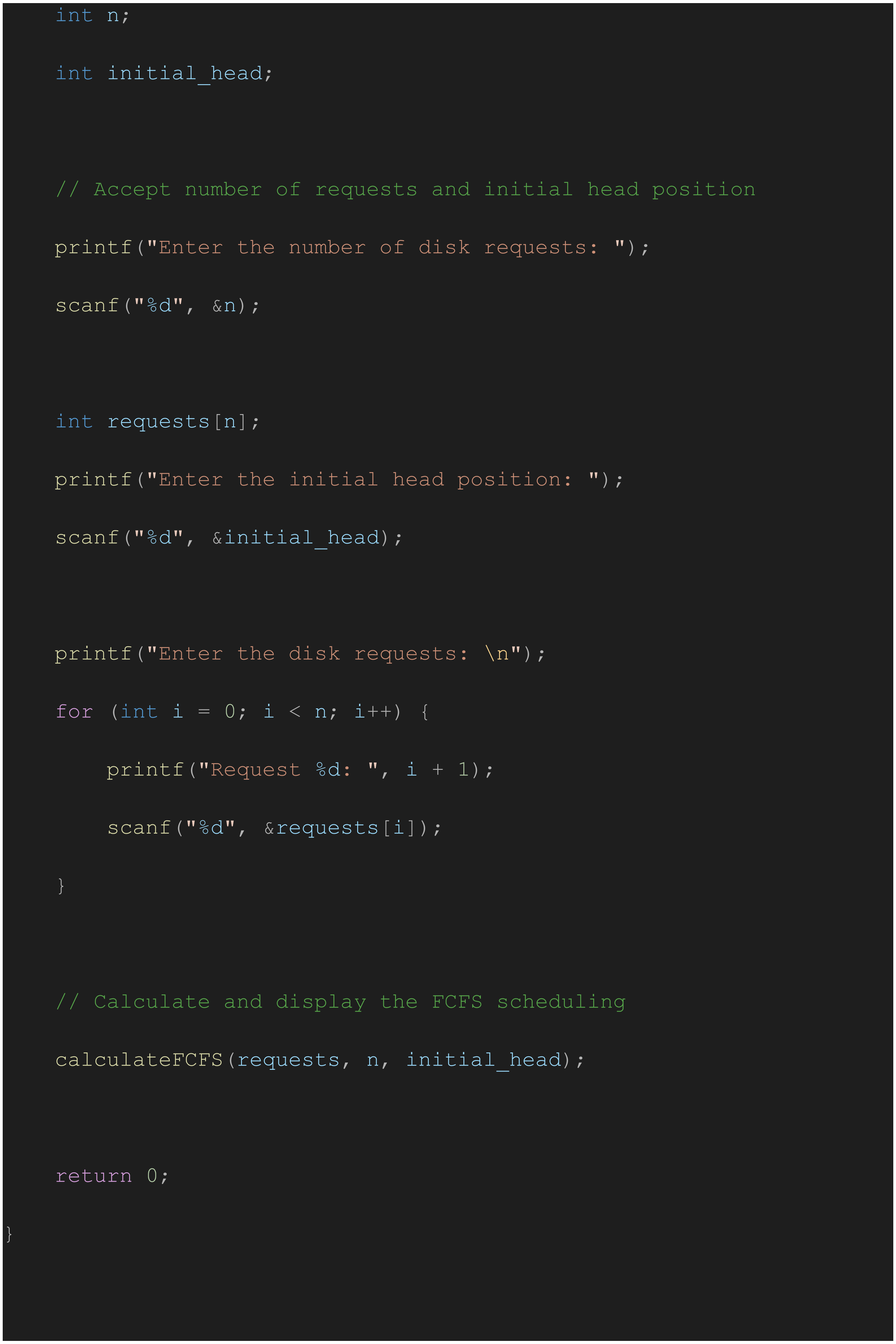
```c
            }

        }

        // If the frame is never going to be used again

        if (j == pageCount) {

            return i; // Replace this frame

        }

    }



    // If all pages are used in the future, replace the one that is used the farthest in the future

    return (indexToReplace != -1) ? indexToReplace : 0;

}



int isPageInFrames(int frames[], int frameCount, int page) {

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

        if (frames[i] == page) {

            return 1; // Page found in frames

        }

    }

    return 0; // Page not found

}



int main() {

    int frameCount, pageCount;
    int pageFaults = 0;


    // Input: Number of frames and number of pages in the reference
string

    printf("Enter the number of frames: ");

    scanf("%d", &frameCount);


    printf("Enter the number of pages in the reference string: ");

    scanf("%d", &pageCount);


    int pages[pageCount];

    printf("Enter the reference string (space-separated): ");

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

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

    }


    int frames[MAX_FRAMES];

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

        frames[i] = -1; // Initialize frames as empty

    }


    // Processing each page in the reference string

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

        int currentPage = pages[i];
        // Check if the current page is already in the frames
        if (!isPageInFrames(frames, frameCount, currentPage)) {
            // Page fault occurs as the page is not in frames
            int indexToReplace = findOptimal(frames, frameCount, pages, pageCount, i); // Find the optimal frame to replace
            frames[indexToReplace] = currentPage; // Replace the optimal frame with the current page
            pageFaults++; // Increment page faults
        }
    }

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

    return 0;
}
```

18. Write a program to simulate FCFS disk scheduling. Calculate total seek time.Print accepted input and output in tabular format

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



```
#include <stdlib.h>


void calculateFCFS(int requests[], int n, int initial_head) {

    int total_seek_time = 0;

    int current_head = initial_head;


    printf("\nDisk Scheduling using FCFS Algorithm:\n");

    printf("------------------------------------------------\n");

    printf("| Request No. |  Request   |  Seek Time      |\n");

    printf("------------------------------------------------\n");


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

        int seek_time = abs(requests[i] - current_head);

        total_seek_time += seek_time;

         printf("|       %2d      |    %3d     |     %3d        |\n",
   i + 1, requests[i], seek_time);

        current_head = requests[i];

    }


    printf("------------------------------------------------\n");

    printf("Total Seek Time: %d\n", total_seek_time);

}


int main() {
```



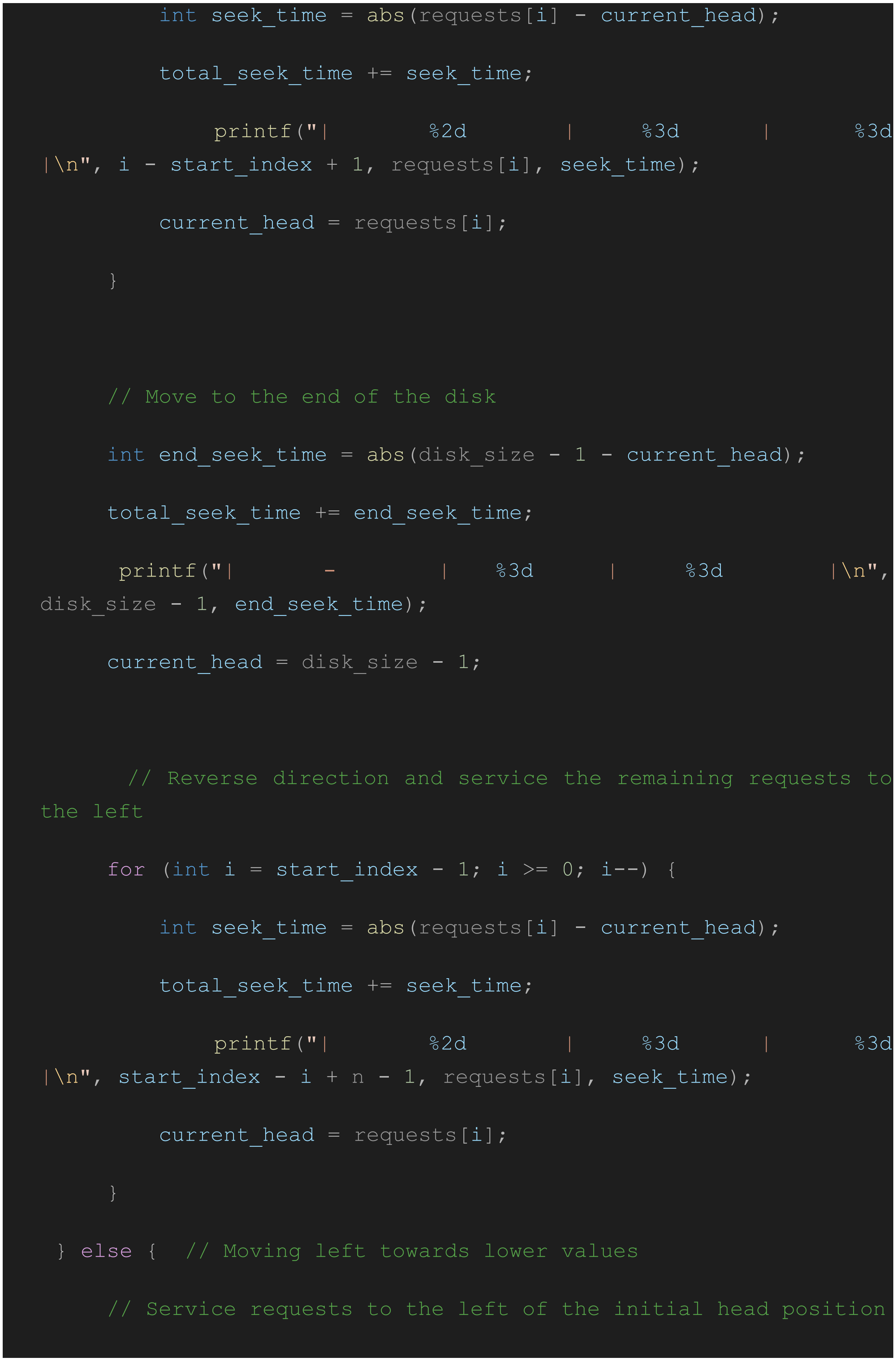
```c
    int n;
    int initial_head;

    // Accept number of requests and initial head position
    printf("Enter the number of disk requests: ");
    scanf("%d", &n);

    int requests[n];
    printf("Enter the initial head position: ");
    scanf("%d", &initial_head);

    printf("Enter the disk requests: \n");
    for (int i = 0; i < n; i++) {
        printf("Request %d: ", i + 1);
        scanf("%d", &requests[i]);
    }

    // Calculate and display the FCFS scheduling
    calculateFCFS(requests, n, initial_head);

    return 0;
}
```

19. Write a program to simulate SSTF disk scheduling. Calculate total seek time.Print accepted input and output in tabular format

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

void calculateSSTF(int requests[], int n, int initial_head) {
    int total_seek_time = 0;
    int current_head = initial_head;
    int visited[n];  // Array to keep track of visited requests
    int i, count = 0;

    // Initialize the visited array
    for (i = 0; i < n; i++) {
        visited[i] = 0;
    }

    printf("\nDisk Scheduling using SSTF Algorithm:\n");
    printf("------------------------------------------------\n");
    printf("| Request No. |  Request   |  Seek Time      |\n");
    printf("------------------------------------------------\n");

    while (count < n) {
        int min_seek_time = 10000; // Arbitrary large value
        int min_index = -1;

        // Find the closest request
        for (i = 0; i < n; i++) {
            if (!visited[i]) {
                int seek_time = abs(requests[i] - current_head);
                if (seek_time < min_seek_time) {
                    min_seek_time = seek_time;
                    min_index = i;
                }
            }
        }

        // Process the closest request
        if (min_index != -1) {
            total_seek_time += min_seek_time; // Update total seek time
              printf("|       %2d        |    %3d      |      %3d |\n", count + 1, requests[min_index], min_seek_time);
            current_head = requests[min_index]; // Move head to the current request
            visited[min_index] = 1; // Mark this request as visited
            count++;
        }

    }


    printf("-------------------------------------------------\n");

      printf("Total Seek Time: %d\n", total_seek_time); // Display
   total seek time

}


int main() {

    int n;

    int initial_head;


    // Accept number of requests and initial head position

    printf("Enter the number of disk requests: ");

    scanf("%d", &n);


    int requests[n];

    printf("Enter the initial head position: ");

    scanf("%d", &initial_head);


    printf("Enter the disk requests: \n");

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

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

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

    // Calculate and display the SSTF scheduling
    calculateSSTF(requests, n, initial_head);

    return 0;
}
```

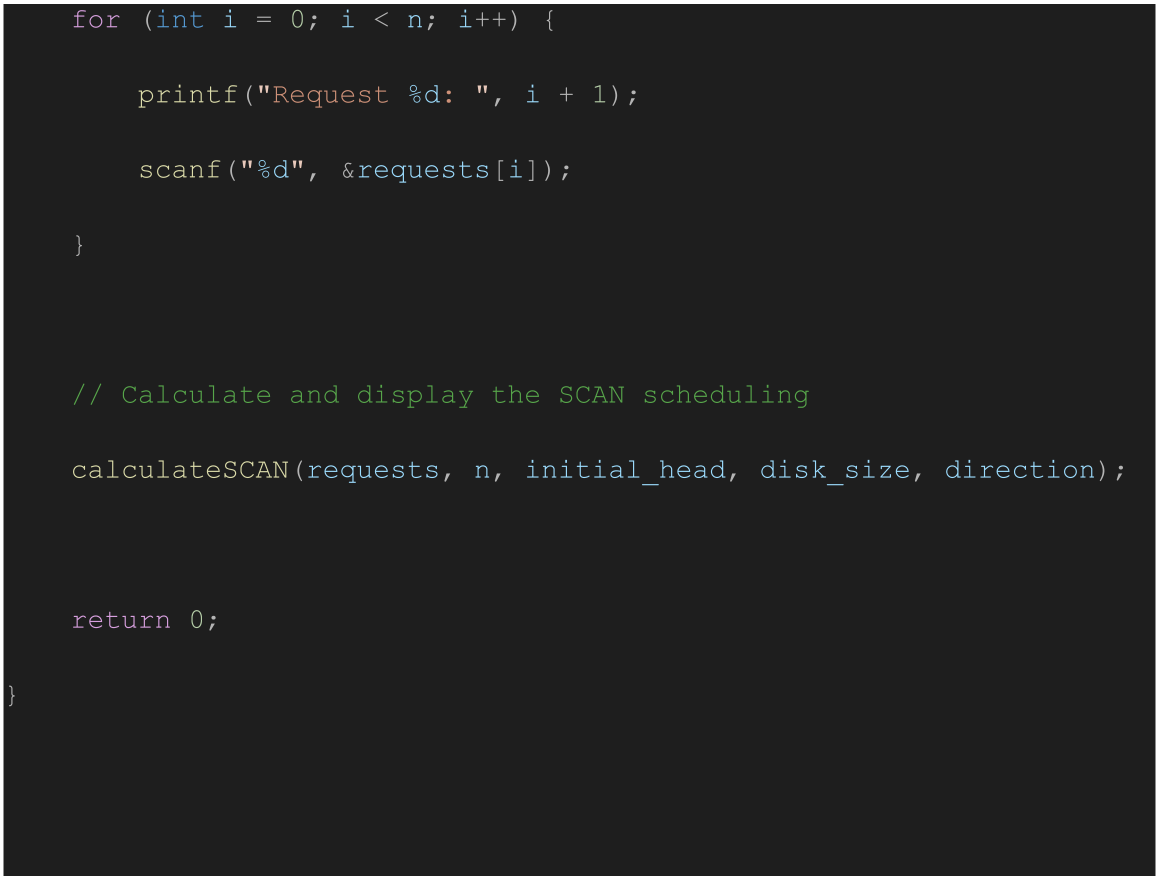
20. Write a program to simulate SCAN disk scheduling. Calculate total seek time.Print accepted input and output in tabular format

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

void calculateSCAN(int requests[], int n, int initial_head, int disk_size, int direction) {
    int total_seek_time = 0;
    int current_head = initial_head;

    // Sort the requests in ascending order
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (requests[j] > requests[j + 1]) {
                int temp = requests[j];
                requests[j] = requests[j + 1];
                requests[j + 1] = temp;
            }
        }
    }


    // Find the starting point where the head should start servicing requests
    int start_index = 0;
    while (start_index < n && requests[start_index] < initial_head) {
        start_index++;
    }


    printf("\nDisk Scheduling using SCAN Algorithm:\n");
    printf("-------------------------------------------------\n");
    printf("| Request No. |  Request   |  Seek Time      |\n");
    printf("-------------------------------------------------\n");


    // Move in the selected direction
    if (direction == 1) {  // Moving right towards higher values
            // Service requests to the right of the initial head position
        for (int i = start_index; i < n; i++) {
```



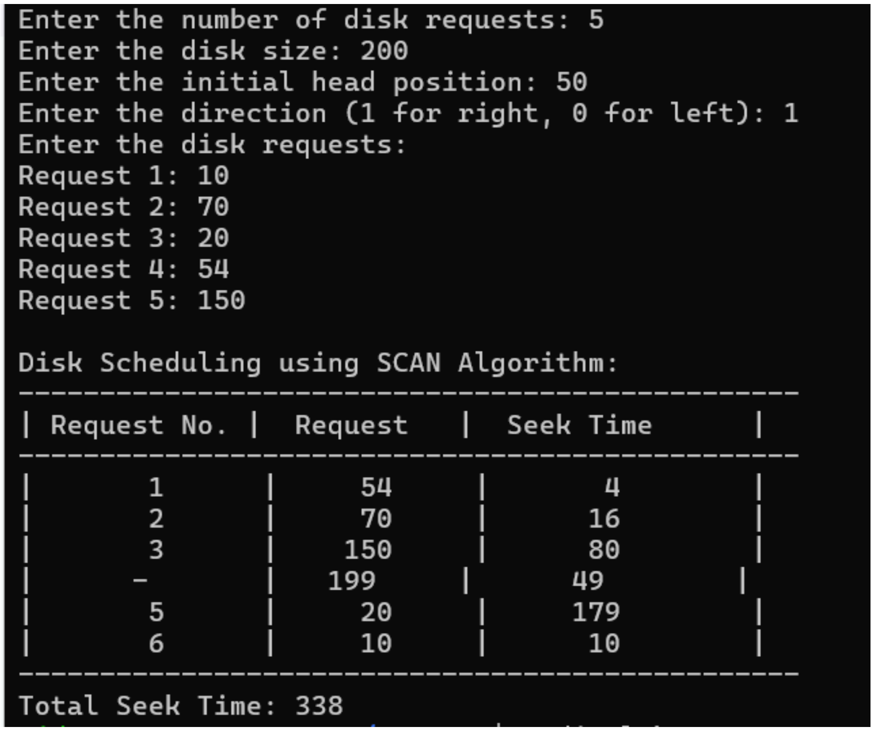
```
         int seek_time = abs(requests[i] - current_head);
         total_seek_time += seek_time;
             printf("|       %2d        |     %3d      |      %3d
|\n", i - start_index + 1, requests[i], seek_time);
         current_head = requests[i];
     }

     // Move to the end of the disk
     int end_seek_time = abs(disk_size - 1 - current_head);
     total_seek_time += end_seek_time;
      printf("|       -        |   %3d     |     %3d        |\n",
disk_size - 1, end_seek_time);
     current_head = disk_size - 1;

       // Reverse direction and service the remaining requests to
the left
     for (int i = start_index - 1; i >= 0; i--) {
         int seek_time = abs(requests[i] - current_head);
         total_seek_time += seek_time;
             printf("|       %2d        |     %3d      |      %3d
|\n", start_index - i + n - 1, requests[i], seek_time);
         current_head = requests[i];
     }
 } else {  // Moving left towards lower values
     // Service requests to the left of the initial head position
```

```
        for (int i = start_index - 1; i >= 0; i--) {

            int seek_time = abs(requests[i] - current_head);

            total_seek_time += seek_time;

                printf("|        %2d        |     %3d      |       %3d
   |\n", start_index - i, requests[i], seek_time);

            current_head = requests[i];

        }



        // Move to the start of the disk

        int end_seek_time = abs(current_head - 0);

        total_seek_time += end_seek_time;

         printf("|        -        |     0       |     %3d        |\n",
   end_seek_time);

        current_head = 0;



         // Reverse direction and service the remaining requests to
   the right

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

            int seek_time = abs(requests[i] - current_head);

            total_seek_time += seek_time;

                printf("|        %2d        |     %3d      |       %3d
   |\n", i + 1, requests[i], seek_time);

            current_head = requests[i];

        }

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

    printf("Total Seek Time: %d\n", total_seek_time); // Display
total seek time

}



int main() {

    int n, initial_head, disk_size, direction;


    // Accept number of requests and initial head position

    printf("Enter the number of disk requests: ");

    scanf("%d", &n);


    int requests[n];

    printf("Enter the disk size: ");

    scanf("%d", &disk_size);


    printf("Enter the initial head position: ");

    scanf("%d", &initial_head);


    printf("Enter the direction (1 for right, 0 for left): ");

    scanf("%d", &direction);


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



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

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

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

    }



    // Calculate and display the SCAN scheduling

    calculateSCAN(requests, n, initial_head, disk_size, direction);



    return 0;

}
```

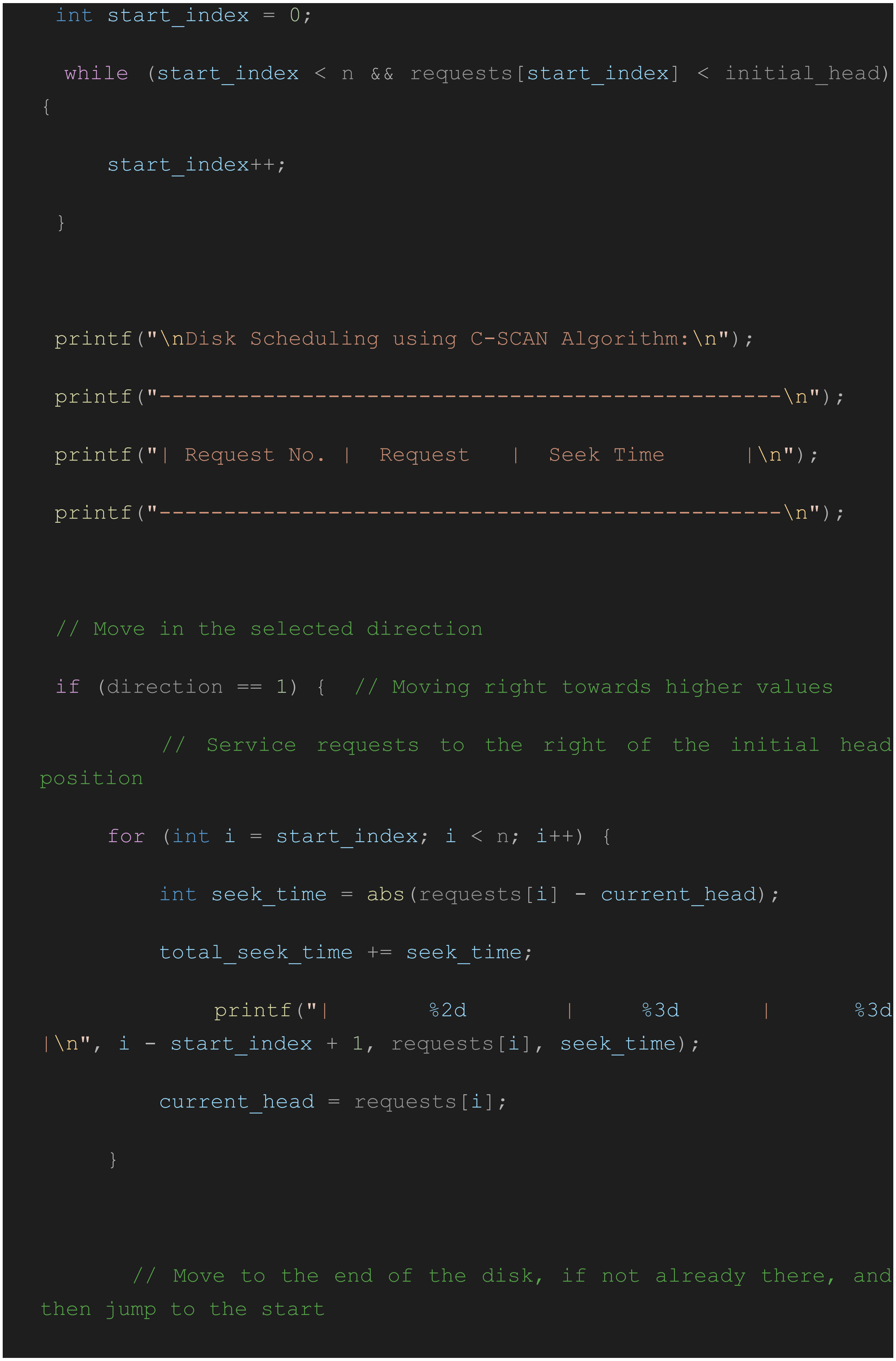
Output :-



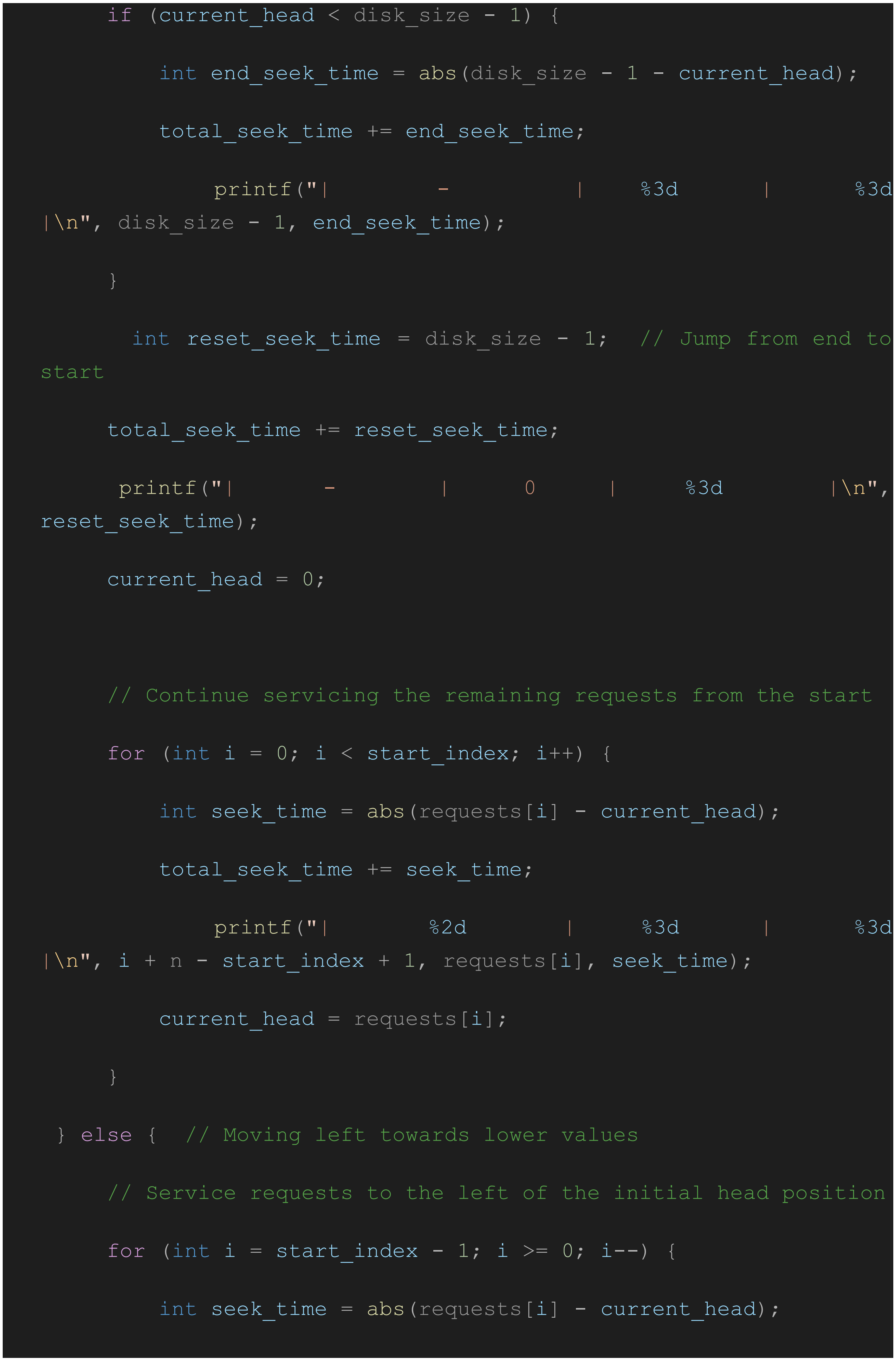
```
Enter the number of disk requests: 5
Enter the disk size: 200
Enter the initial head position: 50
Enter the direction (1 for right, 0 for left): 1
Enter the disk requests:
Request 1: 10
Request 2: 70
Request 3: 20
Request 4: 54
Request 5: 150

Disk Scheduling using SCAN Algorithm:
------------------------------------------------
| Request No. |  Request   |  Seek Time      |
------------------------------------------------
|       1      |     54     |       4        |
|       2      |     70     |      16        |
|       3      |    150     |      80        |
|      –       |   199     |      49        |
|       5      |     20     |     179        |
|       6      |     10     |      10        |
------------------------------------------------
Total Seek Time: 338
```

21. Write a program to simulate C-SCAN disk scheduling. Calculate total seek time.Print accepted input and output in tabular format.

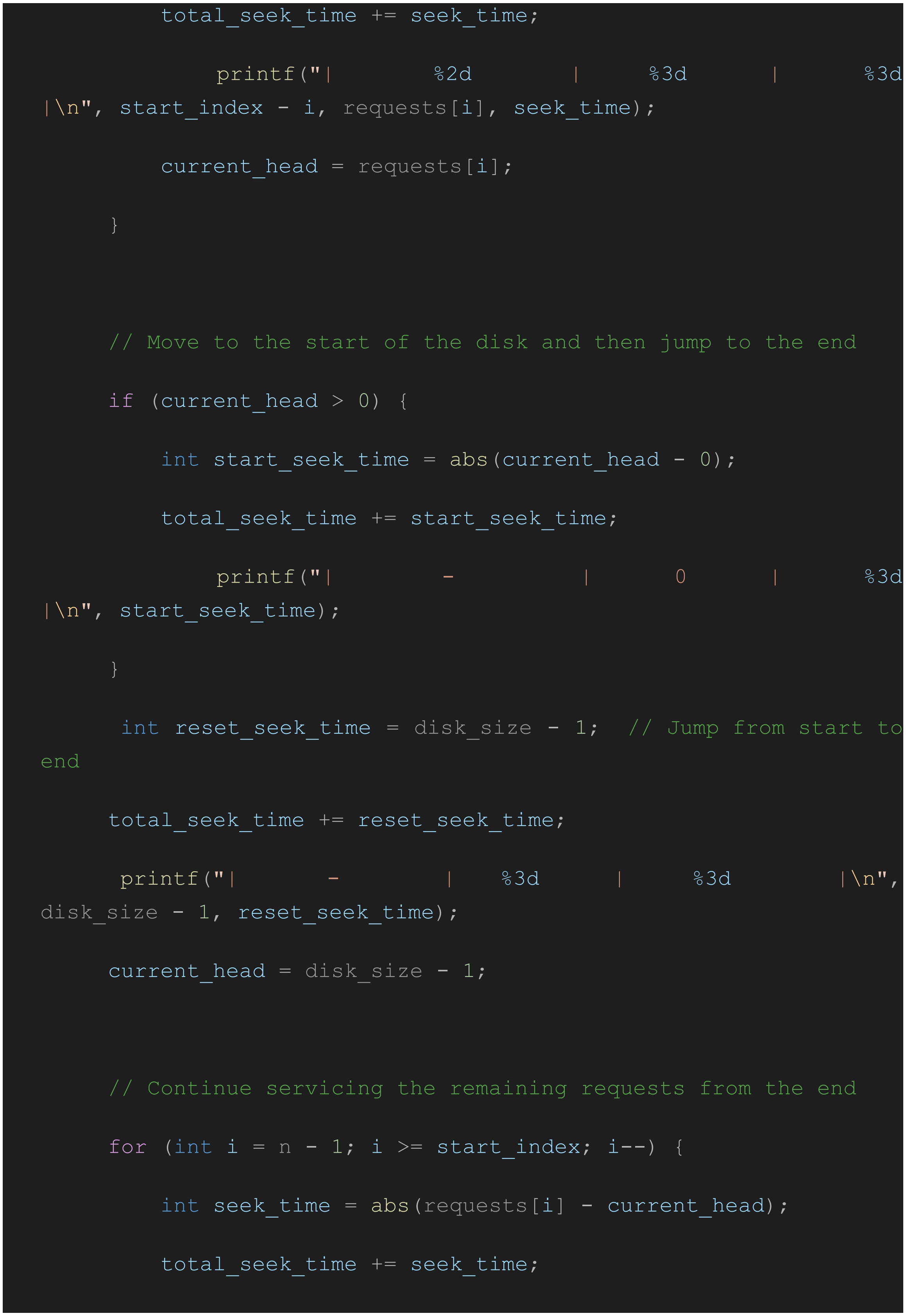
```
#include <stdio.h>
#include <stdlib.h>

void calculateCSCAN(int requests[], int n, int initial_head, int disk_size, int direction) {
    int total_seek_time = 0;
    int current_head = initial_head;

    // Sort the requests in ascending order
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (requests[j] > requests[j + 1]) {
                int temp = requests[j];
                requests[j] = requests[j + 1];
                requests[j + 1] = temp;
            }
        }
    }

     // Find the starting index where the head should start servicing requests
```



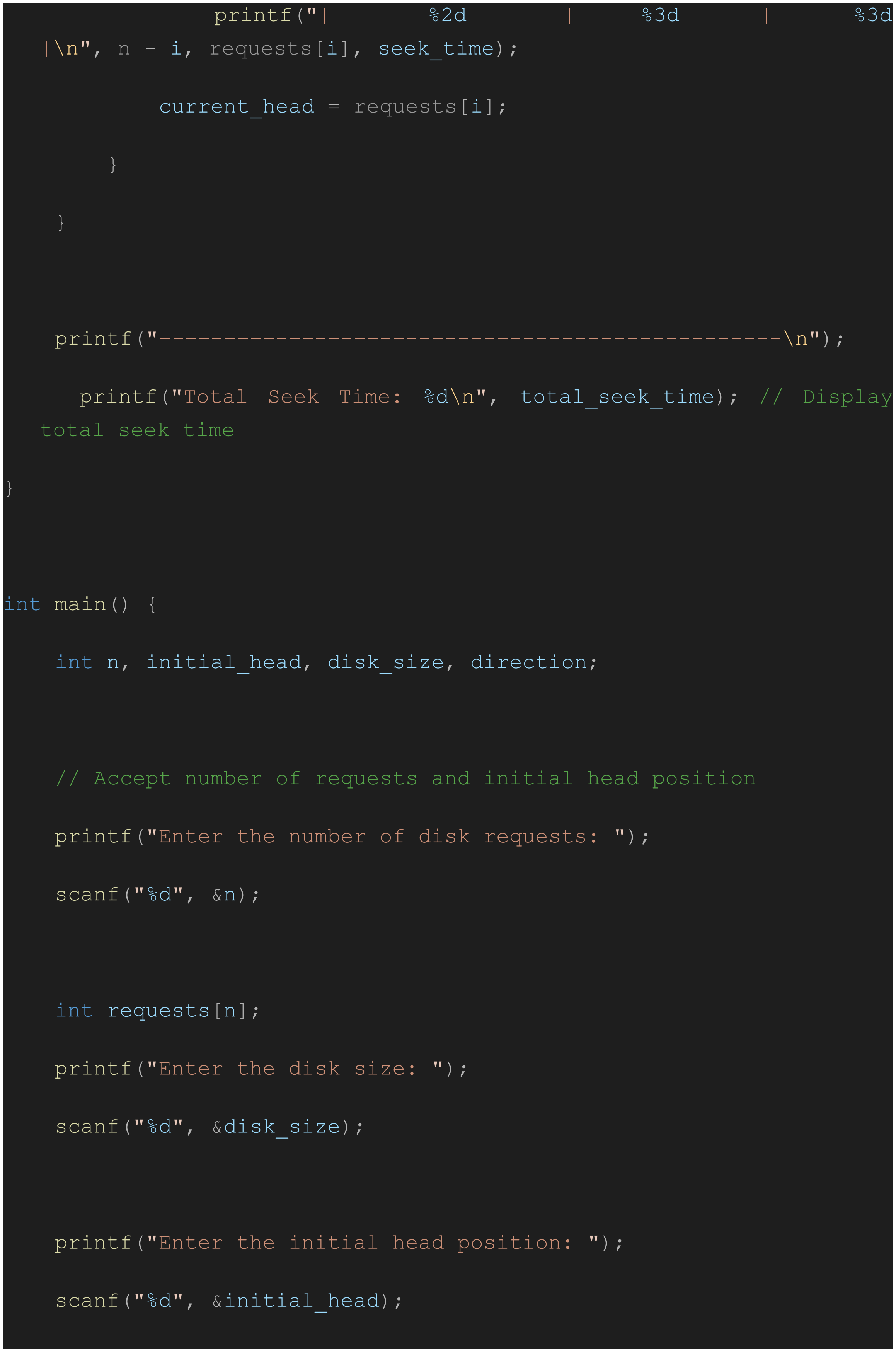
```
 int start_index = 0;

  while (start_index < n && requests[start_index] < initial_head)
{

     start_index++;

 }



 printf("\nDisk Scheduling using C-SCAN Algorithm:\n");

 printf("------------------------------------------------\n");

 printf("| Request No. |  Request   |  Seek Time      |\n");

 printf("------------------------------------------------\n");


 // Move in the selected direction

 if (direction == 1) {  // Moving right towards higher values

         // Service requests to the right of the initial head
position

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

         int seek_time = abs(requests[i] - current_head);

         total_seek_time += seek_time;

             printf("|       %2d       |     %3d      |       %3d
|\n", i - start_index + 1, requests[i], seek_time);

         current_head = requests[i];

     }



       // Move to the end of the disk, if not already there, and
then jump to the start
```



```
    if (current_head < disk_size - 1) {
        int end_seek_time = abs(disk_size - 1 - current_head);
        total_seek_time += end_seek_time;
            printf("|        -         |    %3d     |      %3d
|\n", disk_size - 1, end_seek_time);
    }
      int reset_seek_time = disk_size - 1;  // Jump from end to
start
    total_seek_time += reset_seek_time;
     printf("|       -        |      0     |     %3d        |\n",
reset_seek_time);
    current_head = 0;


    // Continue servicing the remaining requests from the start
    for (int i = 0; i < start_index; i++) {
        int seek_time = abs(requests[i] - current_head);
        total_seek_time += seek_time;
            printf("|       %2d        |     %3d     |      %3d
|\n", i + n - start_index + 1, requests[i], seek_time);
        current_head = requests[i];
    }
 } else {  // Moving left towards lower values
    // Service requests to the left of the initial head position
    for (int i = start_index - 1; i >= 0; i--) {
        int seek_time = abs(requests[i] - current_head);
```



```
         total_seek_time += seek_time;

             printf("|       %2d        |     %3d       |       %3d
|\n", start_index - i, requests[i], seek_time);

         current_head = requests[i];

     }


     // Move to the start of the disk and then jump to the end

     if (current_head > 0) {

         int start_seek_time = abs(current_head - 0);

         total_seek_time += start_seek_time;

             printf("|        -          |       0       |       %3d
|\n", start_seek_time);

     }

      int reset_seek_time = disk_size - 1;  // Jump from start to
end

     total_seek_time += reset_seek_time;

      printf("|       -         |    %3d      |     %3d        |\n",
disk_size - 1, reset_seek_time);

     current_head = disk_size - 1;


     // Continue servicing the remaining requests from the end

     for (int i = n - 1; i >= start_index; i--) {

         int seek_time = abs(requests[i] - current_head);

         total_seek_time += seek_time;
```



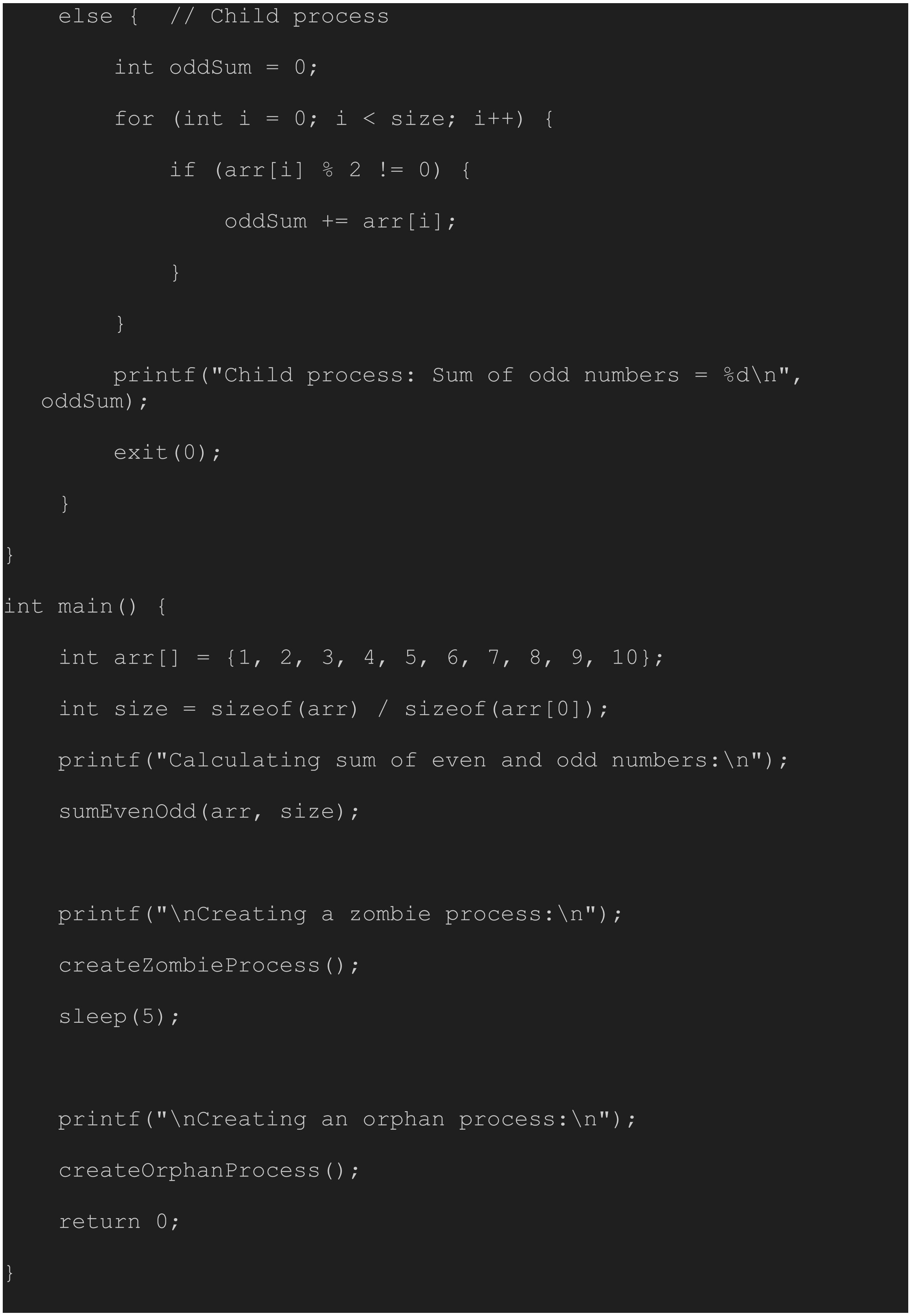
```
                printf("|        %2d        |      %3d      |       %3d
   |\n", n - i, requests[i], seek_time);

            current_head = requests[i];

        }

    }


    printf("------------------------------------------------\n");

      printf("Total  Seek  Time:  %d\n",  total_seek_time); //  Display
   total seek time

}


int main() {

    int n, initial_head, disk_size, direction;


    // Accept number of requests and initial head position

    printf("Enter the number of disk requests: ");

    scanf("%d", &n);


    int requests[n];

    printf("Enter the disk size: ");

    scanf("%d", &disk_size);


    printf("Enter the initial head position: ");

    scanf("%d", &initial_head);
    printf("Enter the direction (1 for right, 0 for left): ");
    scanf("%d", &direction);

    printf("Enter the disk requests: \n");
    for (int i = 0; i < n; i++) {
        printf("Request %d: ", i + 1);
        scanf("%d", &requests[i]);
    }

    // Calculate and display the C-SCAN scheduling
    calculateCSCAN(requests, n, initial_head, disk_size, direction);

    return 0;
}
```

22. Write a program for following 1) zombie process 2),orphan processes 3)sum of even numbers of an array in parent and odd numbers of an array in child process

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

```
void createZombieProcess() {
    pid_t pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        exit(1);
    }

    if (pid > 0) {  // Parent process
        printf("Parent process: Zombie process created. PID = %d\n", pid);
        sleep(10);
    }
    else {  // Child process
        printf("Child process exiting to become zombie.\n");
        exit(0);
    }
}
void createOrphanProcess() {
    pid_t pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        exit(1);
    }

    if (pid > 0) {  // Parent process
```

```
        printf("Parent process exiting to create orphan process.\n");
        exit(0);
    }
    else {  // Child process
        sleep(5);
        printf("Child process (orphan) continuing after parent termination. PID = %d\n", getpid());
    }
}
void sumEvenOdd(int arr[], int size) {
    pid_t pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        exit(1);
    }
    if (pid > 0) {  // Parent process
        int evenSum = 0;
        for (int i = 0; i < size; i++) {
            if (arr[i] % 2 == 0) {
                evenSum += arr[i];
            }
        }
        printf("Parent process: Sum of even numbers = %d\n", evenSum);
        wait(NULL);
    }
```



```
    else {  // Child process
        int oddSum = 0;
        for (int i = 0; i < size; i++) {
            if (arr[i] % 2 != 0) {
                oddSum += arr[i];
            }
        }
        printf("Child process: Sum of odd numbers = %d\n",
   oddSum);
        exit(0);
    }
}
int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("Calculating sum of even and odd numbers:\n");
    sumEvenOdd(arr, size);

    printf("\nCreating a zombie process:\n");
    createZombieProcess();
    sleep(5);

    printf("\nCreating an orphan process:\n");
    createOrphanProcess();
    return 0;
}
```

23. Write a shell script to perform following operations on student database.
a) Insert b) Delete c)Update d)Search

```
#!/bin/bash
DB_FILE="student_database.txt"
show_menu() {
    echo "Select an operation:"
    echo "a) Insert a record"
    echo "b) Delete a record"
    echo "c) Update a record"
    echo "d) Search for a record"
    echo "e) Exit"
    read -p "Enter your choice: " choice
}
insert_record() {
    read -p "Enter Student ID: " id
    read -p "Enter Student Name: " name
    read -p "Enter Student Grade: " grade
    echo "$id,$name,$grade" >> "$DB_FILE"
    echo "Record inserted successfully."
}
delete_record() {
    read -p "Enter Student ID to delete: " id
    if grep -q "^$id," "$DB_FILE"; then
        grep -v "^$id," "$DB_FILE" > temp_file && mv temp_file "$DB_FILE"
        echo "Record with ID $id deleted successfully."
    else
```

```
        echo "Record with ID $id not found."
    fi
}
update_record() {
    read -p "Enter Student ID to update: " id
    if grep -q "^$id," "$DB_FILE"; then
        read -p "Enter new Student Name: " name
        read -p "Enter new Student Grade: " grade
        grep -v "^$id," "$DB_FILE" > temp_file
        echo "$id,$name,$grade" >> temp_file
        mv temp_file "$DB_FILE"
        echo "Record with ID $id updated successfully."
    else
        echo "Record with ID $id not found."
    fi
}
search_record() {
    read -p "Enter Student ID to search: " id
    if grep -q "^$id," "$DB_FILE"; then
        echo "Record found:"
        grep "^$id," "$DB_FILE"
    else
        echo "Record with ID $id not found."
    fi
}
while true; do
```

```
    show_menu

    case $choice in

        a|A) insert_record ;;

        b|B) delete_record ;;

        c|C) update_record ;;

        d|D) search_record ;;

        e|E) echo "Exiting..."; exit 0 ;;

        *) echo "Invalid option. Please try again." ;;

    esac

done
```

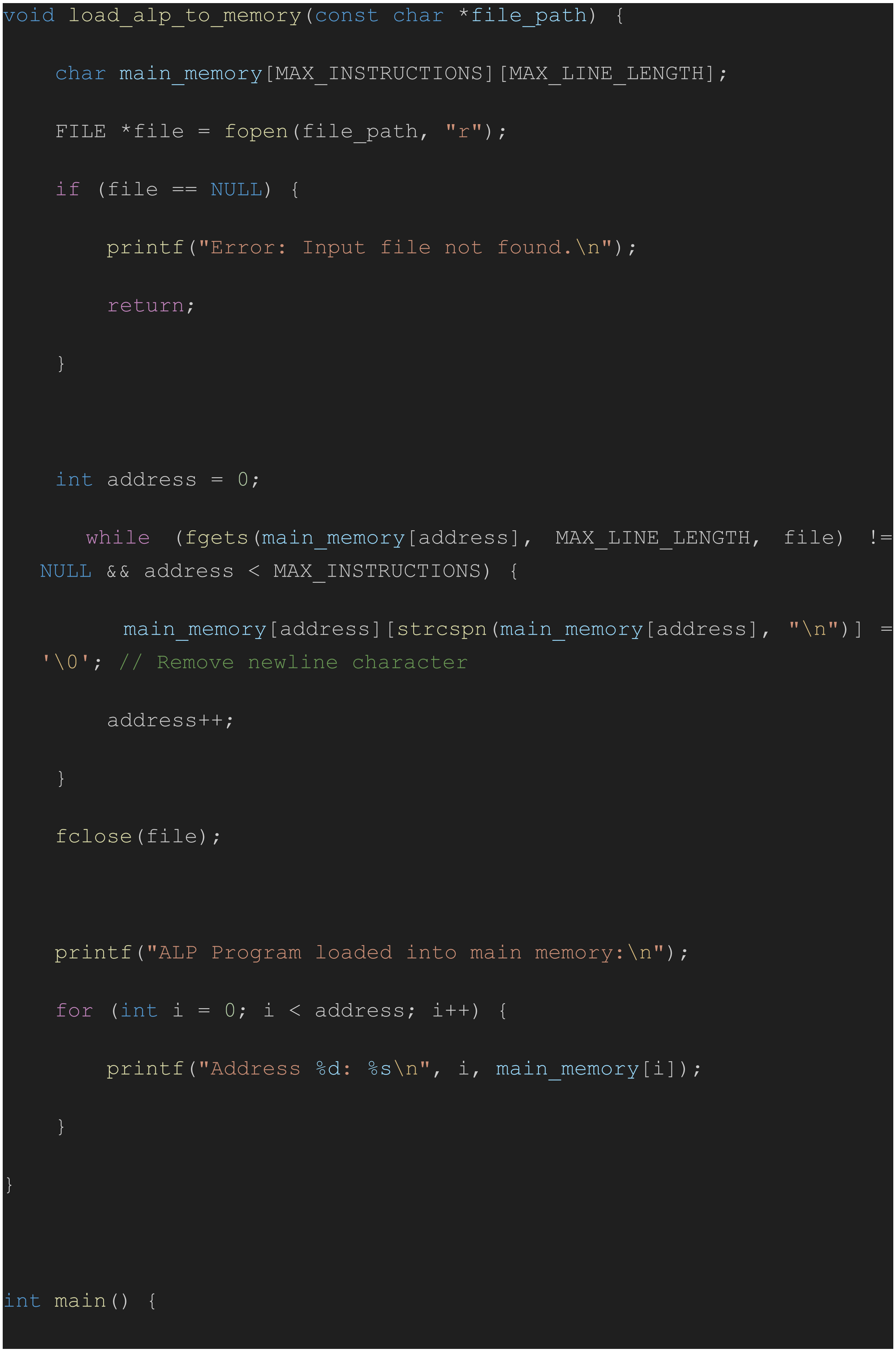
24. Write a program to read and copy the contents of file character by character, line by line.

```
#include <stdio.h>

#include <stdlib.h>



void copyFileCharacterByCharacter(const char *sourceFile, const char *destFile) {

    FILE *src = fopen(sourceFile, "r");

    FILE *dest = fopen(destFile, "w");

    if (src == NULL || dest == NULL) {

        perror("Error opening file");

        exit(1);

    }

    char ch;

    while ((ch = fgetc(src)) != EOF) {

        fputc(ch, dest);
    }
    printf("File copied character by character successfully.\n");
    fclose(src);
    fclose(dest);
}

void copyFileLineByLine(const char *sourceFile, const char
   *destFile) {
    FILE *src = fopen(sourceFile, "r");
    FILE *dest = fopen(destFile, "w");
    if (src == NULL || dest == NULL) {
        perror("Error opening file");
        exit(1);
    }
    char line[1024];
    while (fgets(line, sizeof(line), src) != NULL) {
        fputs(line, dest);
    }
    printf("File copied line by line successfully.\n");
    fclose(src);
    fclose(dest);
}

int main() {
    char sourceFile[100];
    char destFileChar[100];
    char destFileLine[100];
```

```c
    printf("Enter the name of the source file: ");
    scanf("%s", sourceFile);

    printf("Enter the name of the destination file for character-by-character copy: ");
    scanf("%s", destFileChar);

    printf("Enter the name of the destination file for line-by-line copy: ");
    scanf("%s", destFileLine);
    printf("\nCopying file character by character...\n");
    copyFileCharacterByCharacter(sourceFile, destFileChar);
    printf("\nCopying file line by line...\n");
    copyFileLineByLine(sourceFile, destFileLine);
    return 0;
}
```

25. Write a program to load ALP program from input file to main memory.

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


#define MAX_INSTRUCTIONS 100
#define MAX_LINE_LENGTH 50
```



```c
void load_alp_to_memory(const char *file_path) {
    char main_memory[MAX_INSTRUCTIONS][MAX_LINE_LENGTH];
    FILE *file = fopen(file_path, "r");
    if (file == NULL) {
        printf("Error: Input file not found.\n");
        return;
    }

    int address = 0;
      while  (fgets(main_memory[address],  MAX_LINE_LENGTH,  file)  !=
   NULL && address < MAX_INSTRUCTIONS) {
         main_memory[address][strcspn(main_memory[address],  "\n")]  =
   '\0'; // Remove newline character
        address++;
    }
    fclose(file);

    printf("ALP Program loaded into main memory:\n");
    for (int i = 0; i < address; i++) {
        printf("Address %d: %s\n", i, main_memory[i]);
    }
}

int main() {
    load_alp_to_memory("alp_program.txt");
    return 0;
}
```

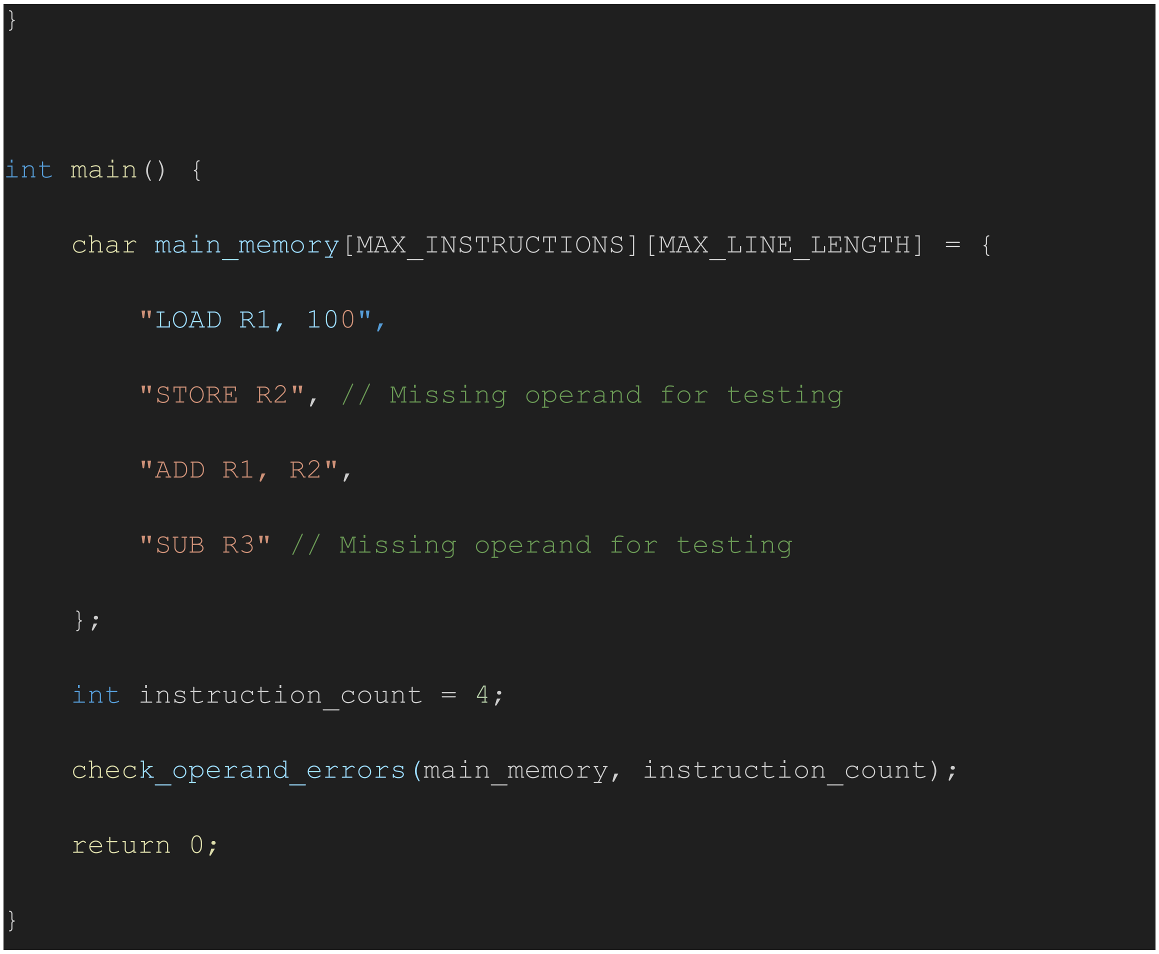
26. Write a program to check Opcode error in a given job and raise an interrupt.

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

#define MAX_INSTRUCTIONS 100
#define MAX_LINE_LENGTH 50

const char *valid_opcodes[] = {"LOAD", "STORE", "ADD", "SUB", "JMP", "HALT"};
#define NUM_OPCODES (sizeof(valid_opcodes) / sizeof(valid_opcodes[0]))

int is_valid_opcode(const char *opcode) {
    for (int i = 0; i < NUM_OPCODES; i++) {
        if (strcmp(opcode, valid_opcodes[i]) == 0) {
            return 1;
        }
    }
    return 0;
}
```

```c
void check_opcode_errors(char main_memory[][MAX_LINE_LENGTH], int instruction_count) {
    for (int i = 0; i < instruction_count; i++) {
        char opcode[MAX_LINE_LENGTH];
         sscanf(main_memory[i], "%s", opcode);  // Extract the first word as opcode
        if (!is_valid_opcode(opcode)) {
                printf("Opcode error at address %d: Invalid opcode '%s'\n", i, opcode);
        }
    }
}


int main() {
    char main_memory[MAX_INSTRUCTIONS][MAX_LINE_LENGTH] = {
        "LOAD R1, 100",
        "STOREE R2, 200", // Invalid opcode for testing
        "ADD R1, R2",
        "INVALID_OP R3, 300" // Invalid opcode for testing
    };
    int instruction_count = 4;
    check_opcode_errors(main_memory, instruction_count);
    return 0;
}
```

27. Write a program to check Oprand error in a given job and raise an interrupt.

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


#define MAX_INSTRUCTIONS 100
#define MAX_LINE_LENGTH 50


void  check_operand_errors(char  main_memory[][MAX_LINE_LENGTH],  int
   instruction_count) {
    for (int i = 0; i < instruction_count; i++) {
        int operand_count = 0;
        char *token = strtok(main_memory[i], " ,");
        // Count tokens after the opcode as operands
        while (token != NULL) {
            operand_count++;
            token = strtok(NULL, " ,");
        }


        if (operand_count < 3) { // Opcode + 2 operands
             printf("Operand error at address %d: Missing operand(s)
   in instruction '%s'\n", i, main_memory[i]);
        }
    }
```



```c
}

int main() {
    char main_memory[MAX_INSTRUCTIONS][MAX_LINE_LENGTH] = {
        "LOAD R1, 100",
        "STORE R2", // Missing operand for testing
        "ADD R1, R2",
        "SUB R3" // Missing operand for testing
    };
    int instruction_count = 4;
    check_operand_errors(main_memory, instruction_count);
    return 0;
}
```