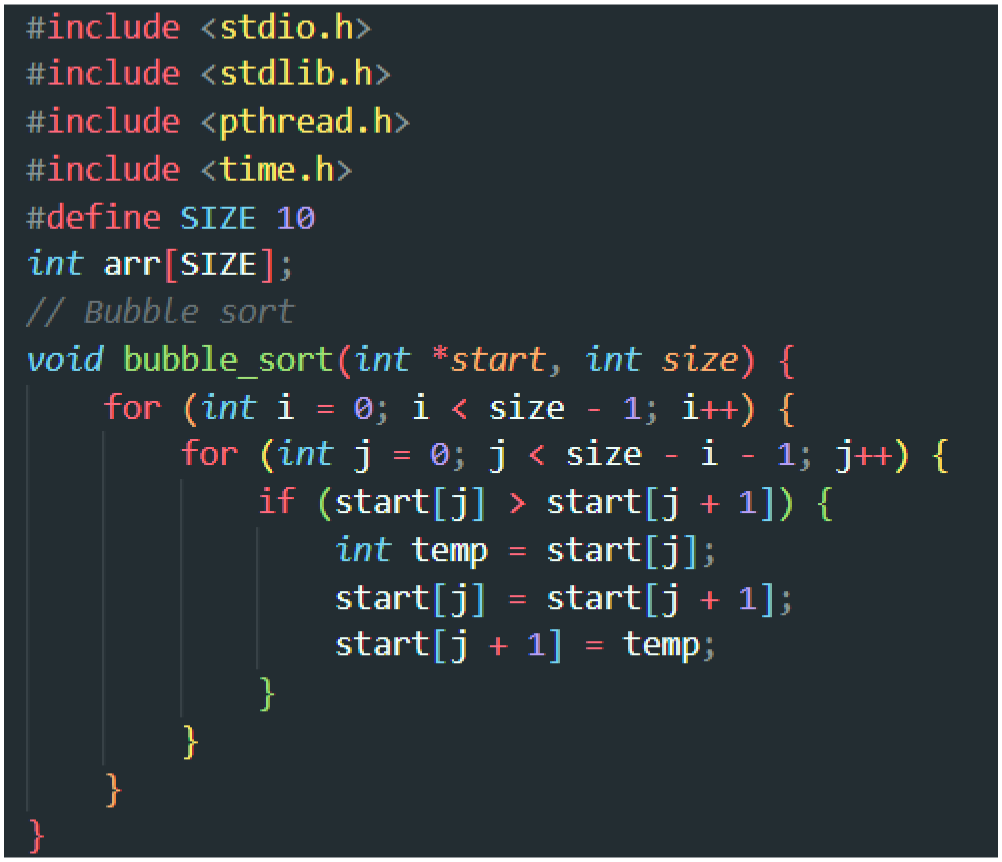
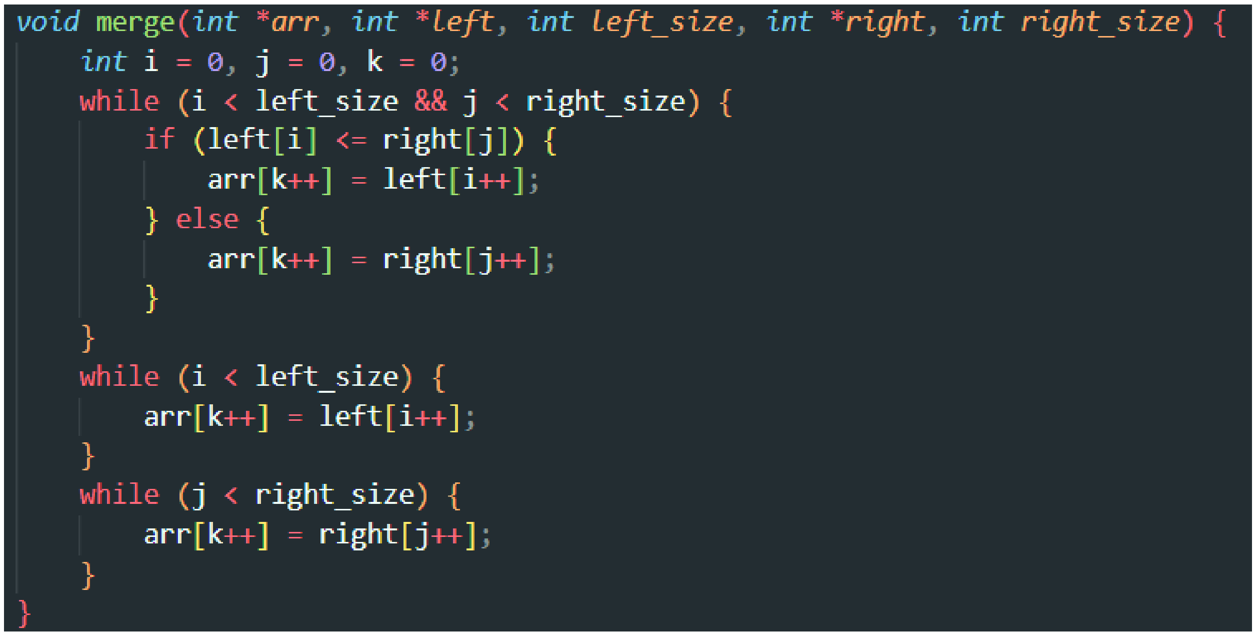
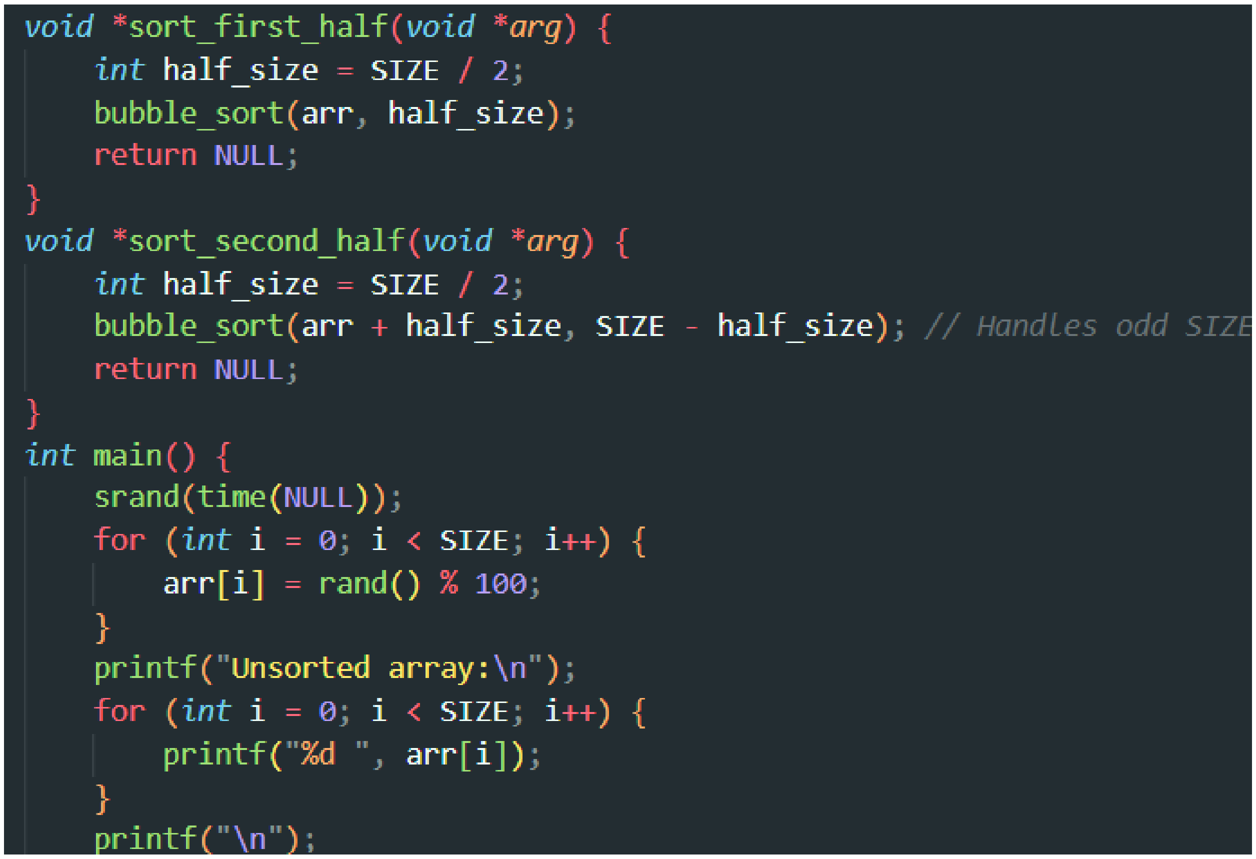
1. Create a multithreaded C program that sorts an array of integers using two threads. The first thread should sort the first half of the array, and the second thread should sort the second half. After both threads finish sorting, the main thread should merge the two sorted halves into a final sorted array.



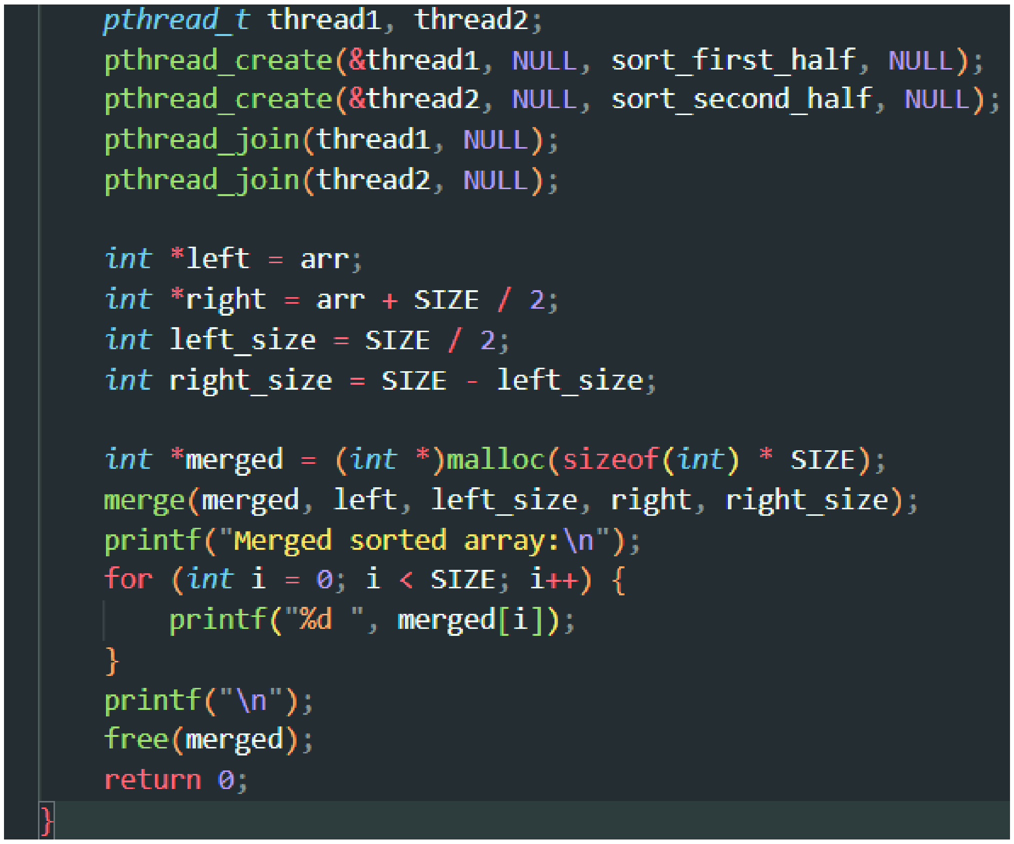
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#define SIZE 10
int arr[SIZE];
// Bubble sort
void bubble_sort(int *start, int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (start[j] > start[j + 1]) {
                int temp = start[j];
                start[j] = start[j + 1];
                start[j + 1] = temp;
            }
        }
    }
}
```



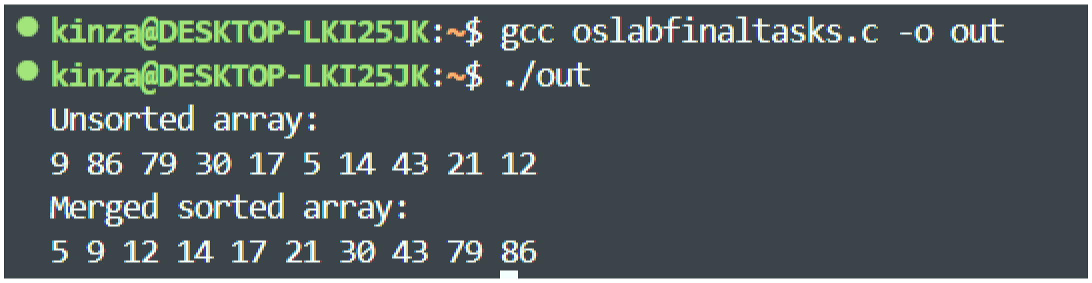
```c
void merge(int *arr, int *left, int left_size, int *right, int right_size) {
    int i = 0, j = 0, k = 0;
    while (i < left_size && j < right_size) {
        if (left[i] <= right[j]) {
            arr[k++] = left[i++];
        } else {
            arr[k++] = right[j++];
        }
    }
    while (i < left_size) {
        arr[k++] = left[i++];
    }
    while (j < right_size) {
        arr[k++] = right[j++];
    }
}
```



```c
void *sort_first_half(void *arg) {
    int half_size = SIZE / 2;
    bubble_sort(arr, half_size);
    return NULL;
}
void *sort_second_half(void *arg) {
    int half_size = SIZE / 2;
    bubble_sort(arr + half_size, SIZE - half_size); // Handles odd SIZE
    return NULL;
}
int main() {
    srand(time(NULL));
    for (int i = 0; i < SIZE; i++) {
        arr[i] = rand() % 100;
    }
    printf("Unsorted array:\n");
    for (int i = 0; i < SIZE; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
```

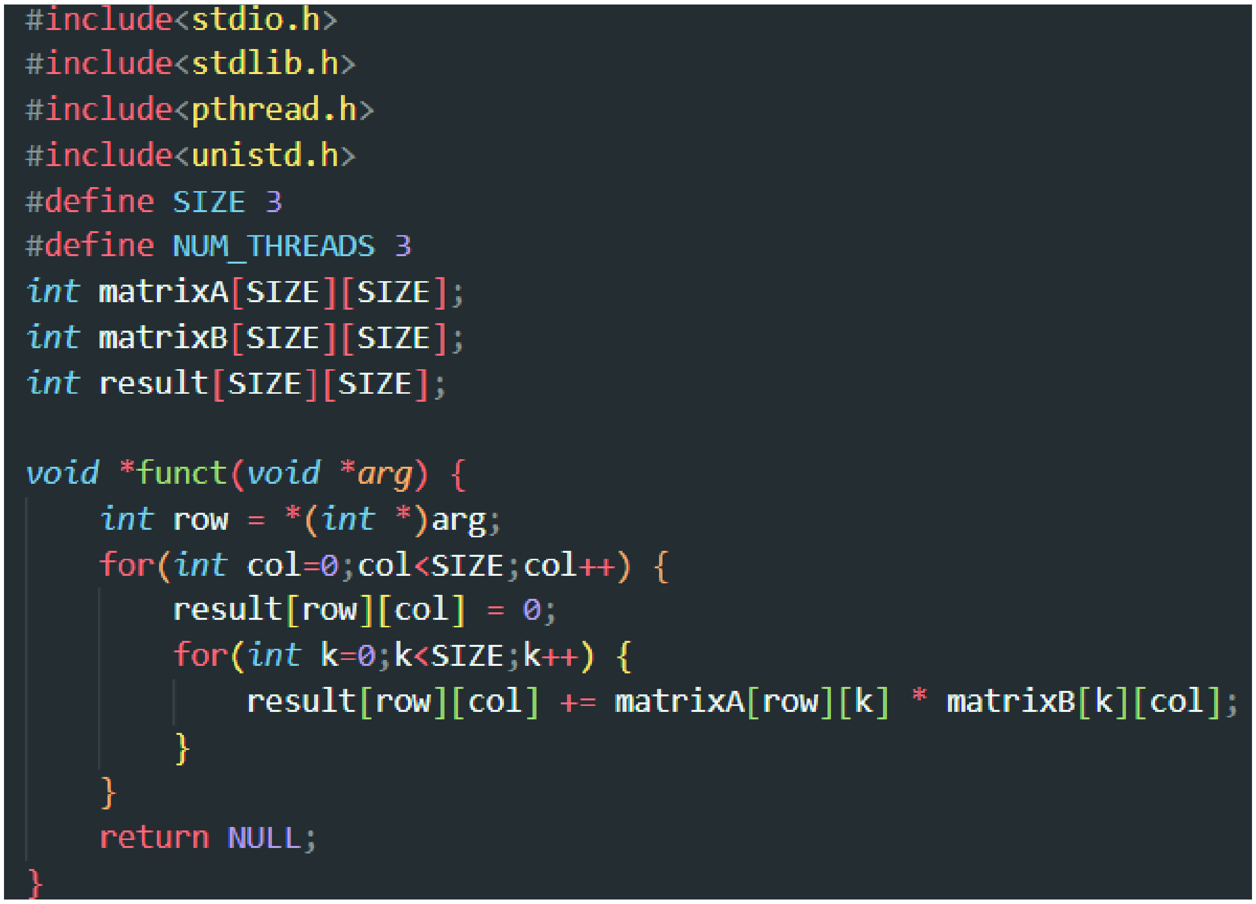


```c
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, sort_first_half, NULL);
    pthread_create(&thread2, NULL, sort_second_half, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    int *left = arr;
    int *right = arr + SIZE / 2;
    int left_size = SIZE / 2;
    int right_size = SIZE - left_size;

    int *merged = (int *)malloc(sizeof(int) * SIZE);
    merge(merged, left, left_size, right, right_size);
    printf("Merged sorted array:\n");
    for (int i = 0; i < SIZE; i++) {
        printf("%d ", merged[i]);
    }
    printf("\n");
    free(merged);
    return 0;
}
```

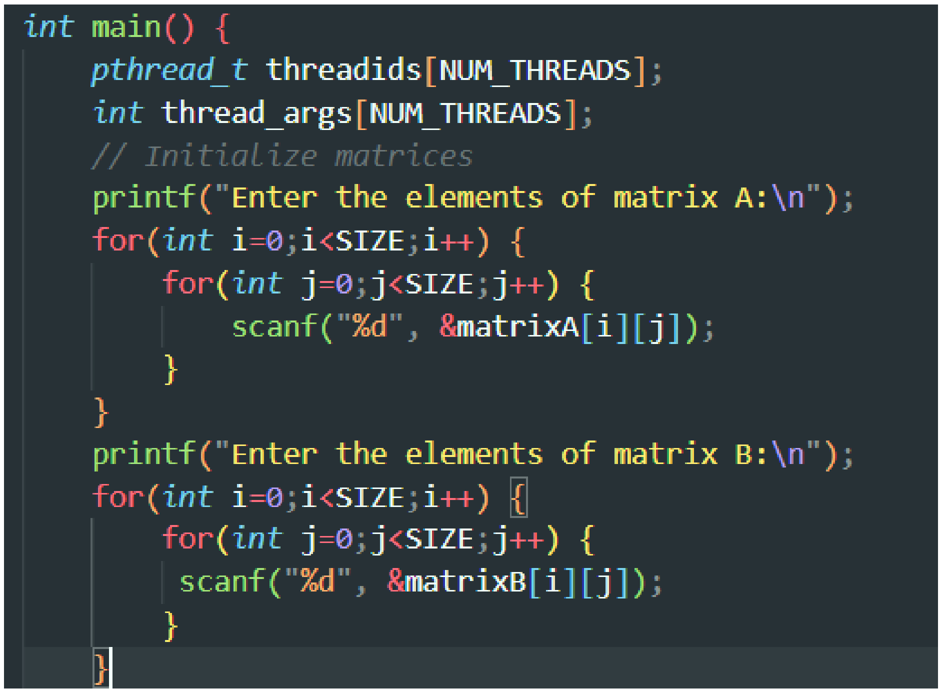


```
kinza@DESKTOP-LKI25JK:~$ gcc oslabfinaltasks.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out
Unsorted array:
9 86 79 30 17 5 14 43 21 12
Merged sorted array:
5 9 12 14 17 21 30 43 79 86
```

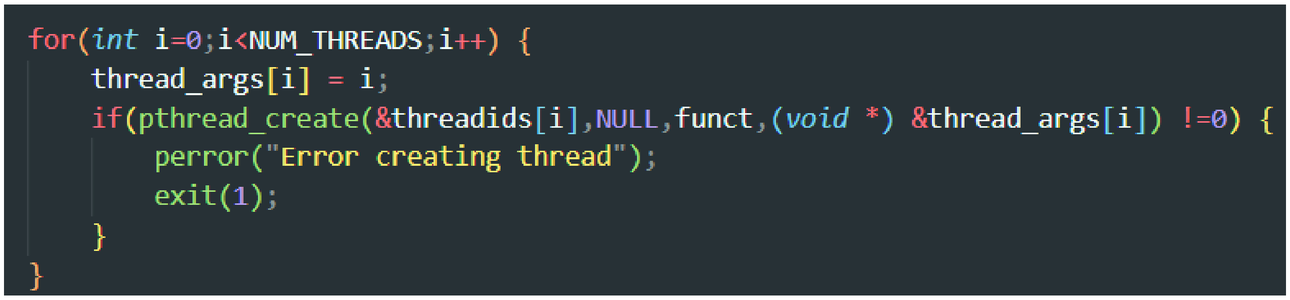
2. Create a multithreaded C program that performs matrix multiplication using threads. Prompt the user to input two square matrices of size 3x3. Use a separate thread to calculate each row of the result matrix. After all threads have completed, display the final multiplied matrix.



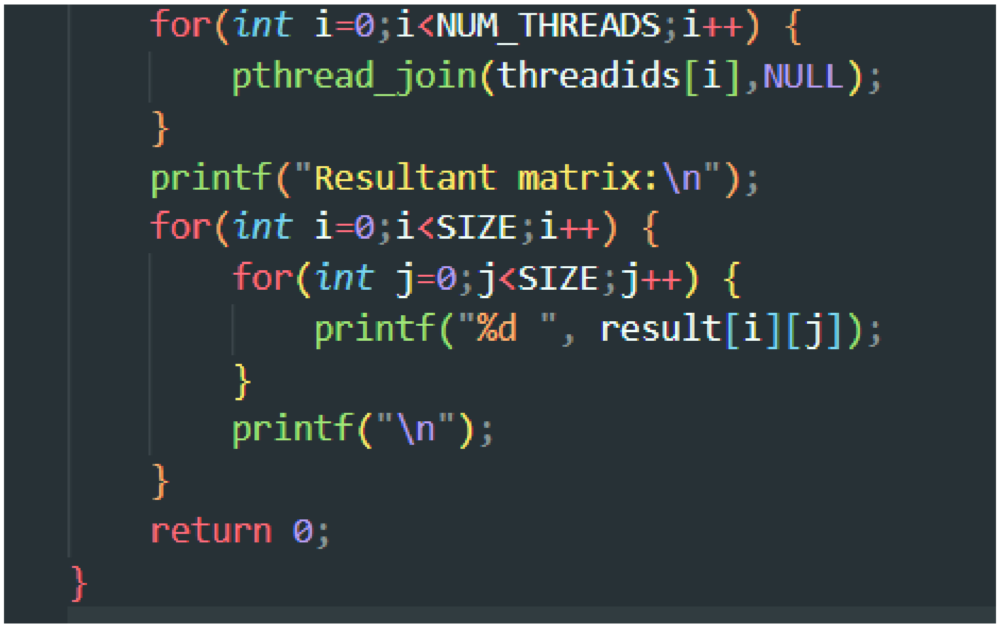
```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#define SIZE 3
#define NUM_THREADS 3
int matrixA[SIZE][SIZE];
int matrixB[SIZE][SIZE];
int result[SIZE][SIZE];

void *funct(void *arg) {
    int row = *(int *)arg;
    for(int col=0;col<SIZE;col++) {
        result[row][col] = 0;
        for(int k=0;k<SIZE;k++) {
            result[row][col] += matrixA[row][k] * matrixB[k][col];
        }
    }
    return NULL;
}
```



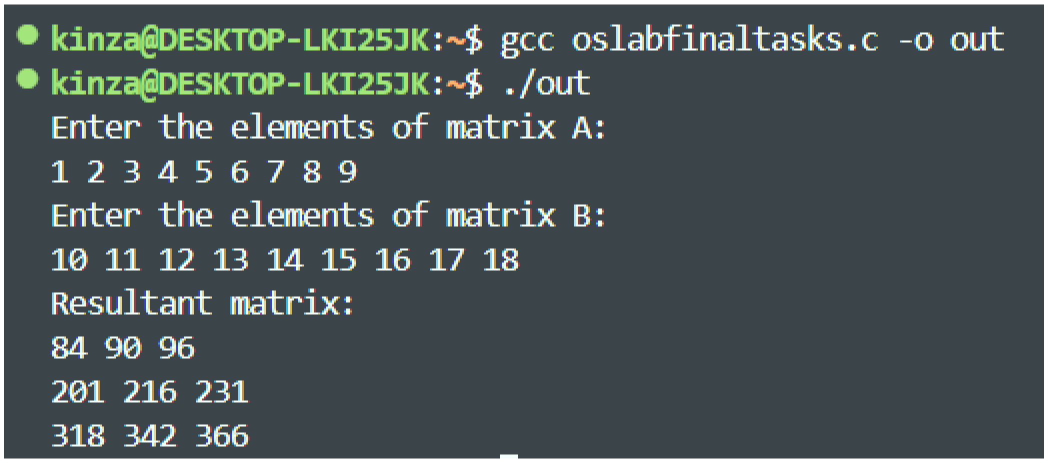
```c
int main() {
    pthread_t threadids[NUM_THREADS];
    int thread_args[NUM_THREADS];
    // Initialize matrices
    printf("Enter the elements of matrix A:\n");
    for(int i=0;i<SIZE;i++) {
        for(int j=0;j<SIZE;j++) {
            scanf("%d", &matrixA[i][j]);
        }
    }
    printf("Enter the elements of matrix B:\n");
    for(int i=0;i<SIZE;i++) {
        for(int j=0;j<SIZE;j++) {
         scanf("%d", &matrixB[i][j]);
        }
    }
```



```c
for(int i=0;i<NUM_THREADS;i++) {
    thread_args[i] = i;
    if(pthread_create(&threadids[i],NULL,funct,(void *) &thread_args[i]) !=0) {
        perror("Error creating thread");
        exit(1);
    }
}
```

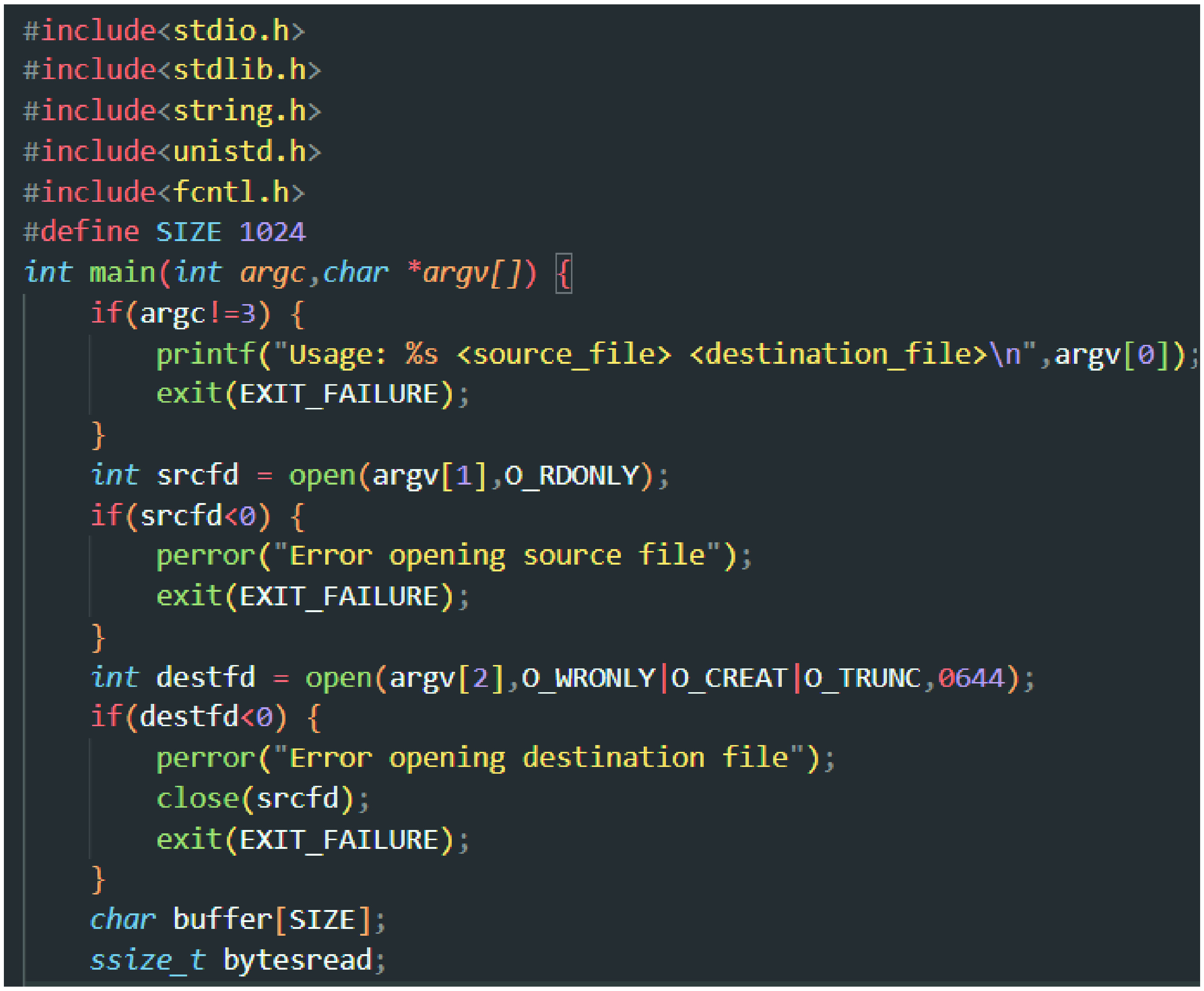


```c
    for(int i=0;i<NUM_THREADS;i++) {
        pthread_join(threadids[i],NULL);
    }
    printf("Resultant matrix:\n");
    for(int i=0;i<SIZE;i++) {
        for(int j=0;j<SIZE;j++) {
            printf("%d ", result[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

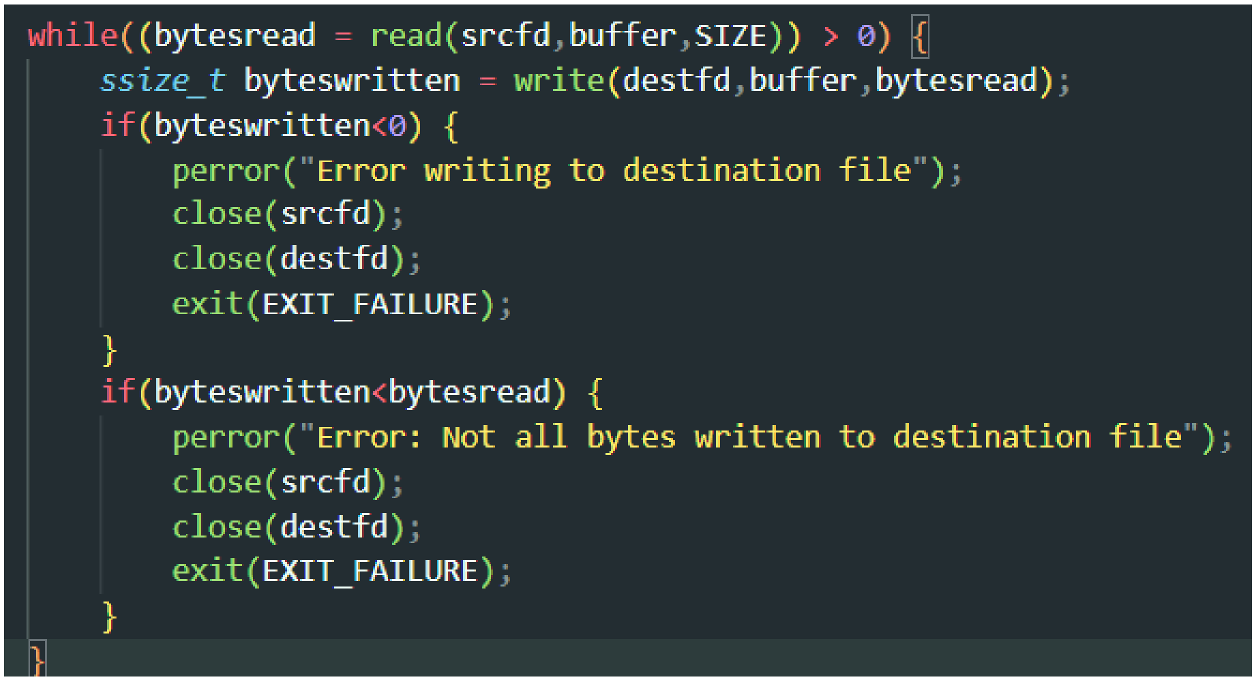


```
● kinza@DESKTOP-LKI25JK:~$ gcc oslabfinaltasks.c -o out
● kinza@DESKTOP-LKI25JK:~$ ./out
  Enter the elements of matrix A:
  1 2 3 4 5 6 7 8 9
  Enter the elements of matrix B:
  10 11 12 13 14 15 16 17 18
  Resultant matrix:
  84 90 96
  201 216 231
  318 342 366
```

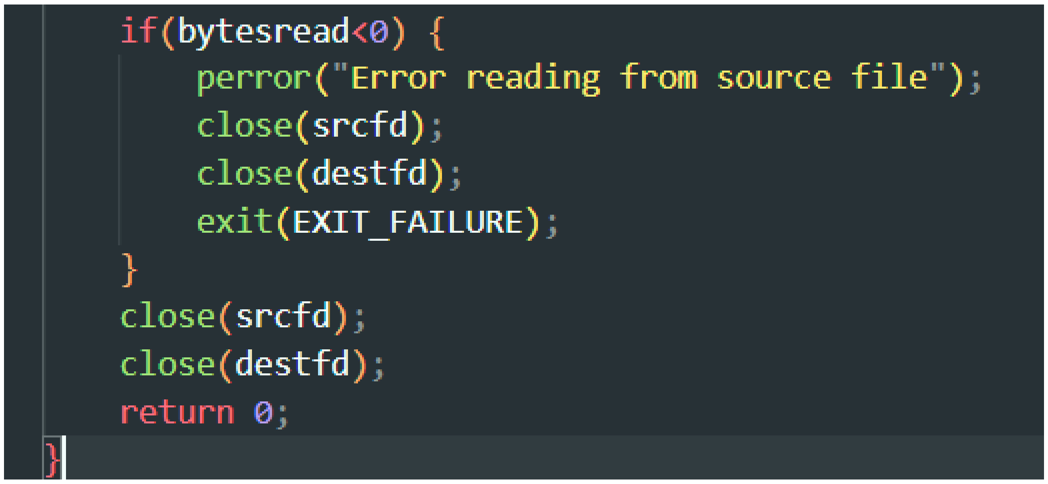
3. Write a C program that replicates a simplified version of the `cp` command using system calls. It should read the source file using `read()` and write to a destination file using `write()`, with error handling.



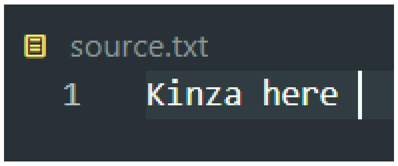
```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<fcntl.h>
#define SIZE 1024
int main(int argc,char *argv[]) {
    if(argc!=3) {
        printf("Usage: %s <source_file> <destination_file>\n",argv[0]);
        exit(EXIT_FAILURE);
    }
    int srcfd = open(argv[1],O_RDONLY);
    if(srcfd<0) {
        perror("Error opening source file");
        exit(EXIT_FAILURE);
    }
    int destfd = open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,0644);
    if(destfd<0) {
        perror("Error opening destination file");
        close(srcfd);
        exit(EXIT_FAILURE);
    }
    char buffer[SIZE];
    ssize_t bytesread;
```



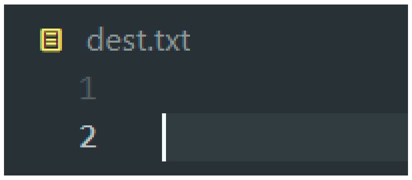
```c
while((bytesread = read(srcfd,buffer,SIZE)) > 0) {
    ssize_t byteswritten = write(destfd,buffer,bytesread);
    if(byteswritten<0) {
        perror("Error writing to destination file");
        close(srcfd);
        close(destfd);
        exit(EXIT_FAILURE);
    }
    if(byteswritten<bytesread) {
        perror("Error: Not all bytes written to destination file");
        close(srcfd);
        close(destfd);
        exit(EXIT_FAILURE);
    }
}
```



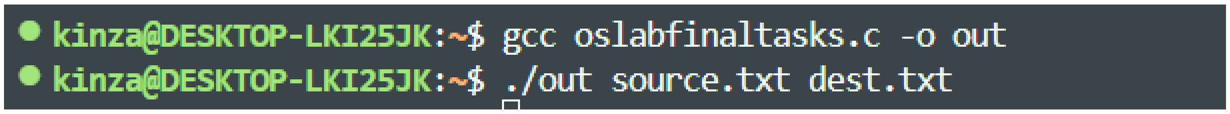
```c
    if(bytesread<0) {
        perror("Error reading from source file");
        close(srcfd);
        close(destfd);
        exit(EXIT_FAILURE);
    }
    close(srcfd);
    close(destfd);
    return 0;
}
```



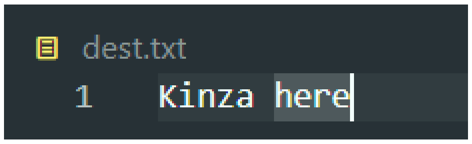
```
source.txt
Kinza here
```



```
dest.txt
```

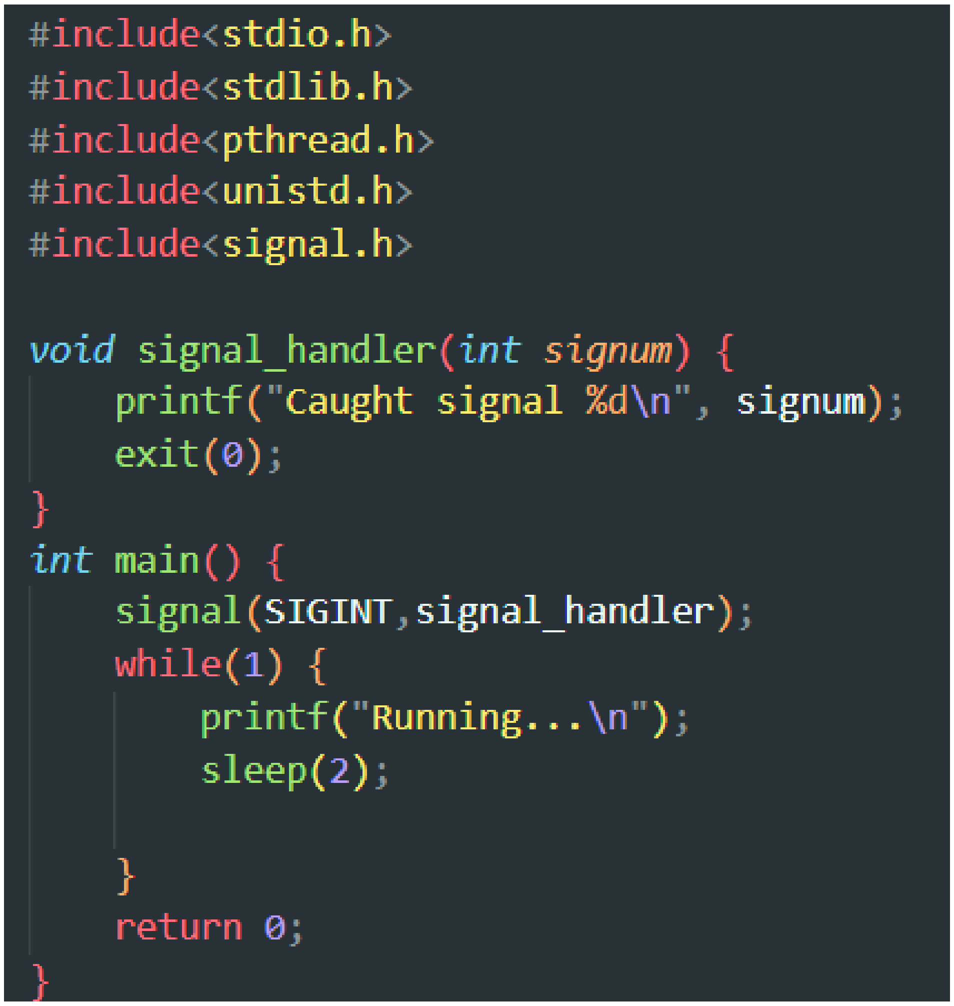


```
kinza@DESKTOP-LKI25JK:~$ gcc oslabfinaltasks.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out source.txt dest.txt
```

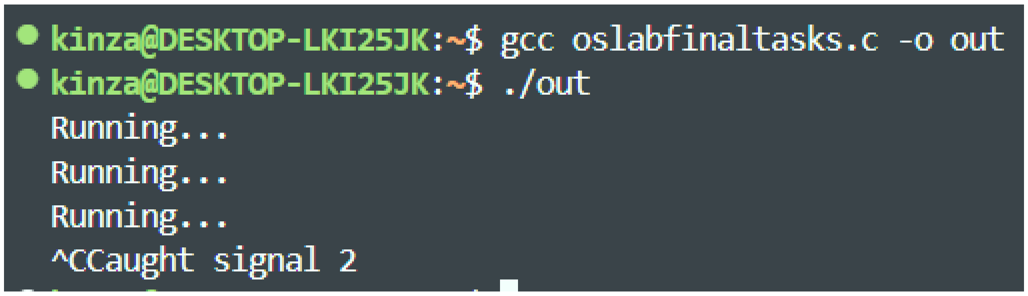


```
dest.txt
Kinza here
```

4. Create a C program that continuously prints a message like "Running…" every 2 seconds. When the user presses Ctrl+C (SIGINT), catch the signal and display "SIGINT caught, terminating safely." Use `signal()` to handle the signal.

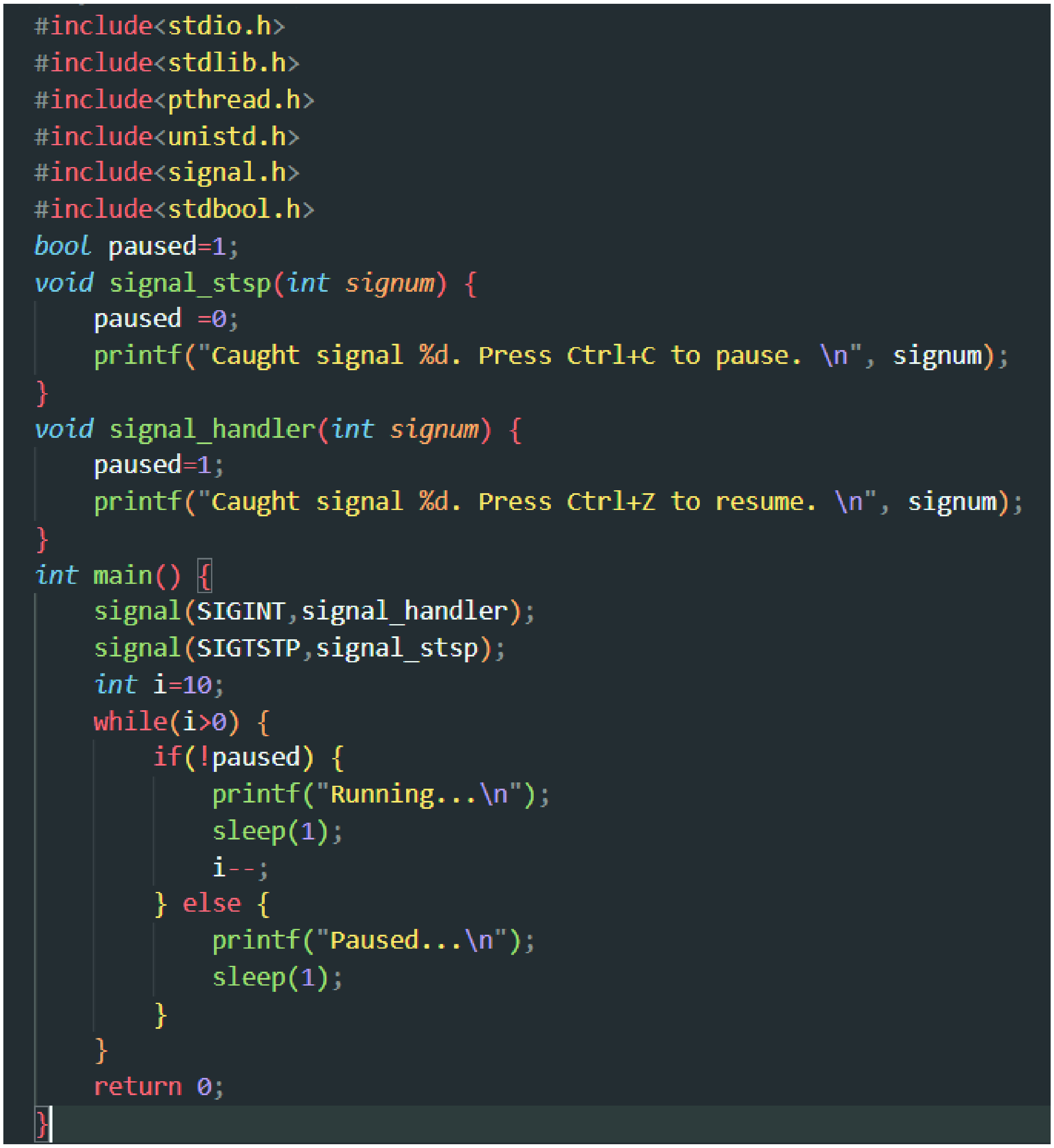


```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#include<signal.h>

void signal_handler(int signum) {
    printf("Caught signal %d\n", signum);
    exit(0);
}
int main() {
    signal(SIGINT,signal_handler);
    while(1) {
        printf("Running...\n");
        sleep(2);

    }
    return 0;
}
```

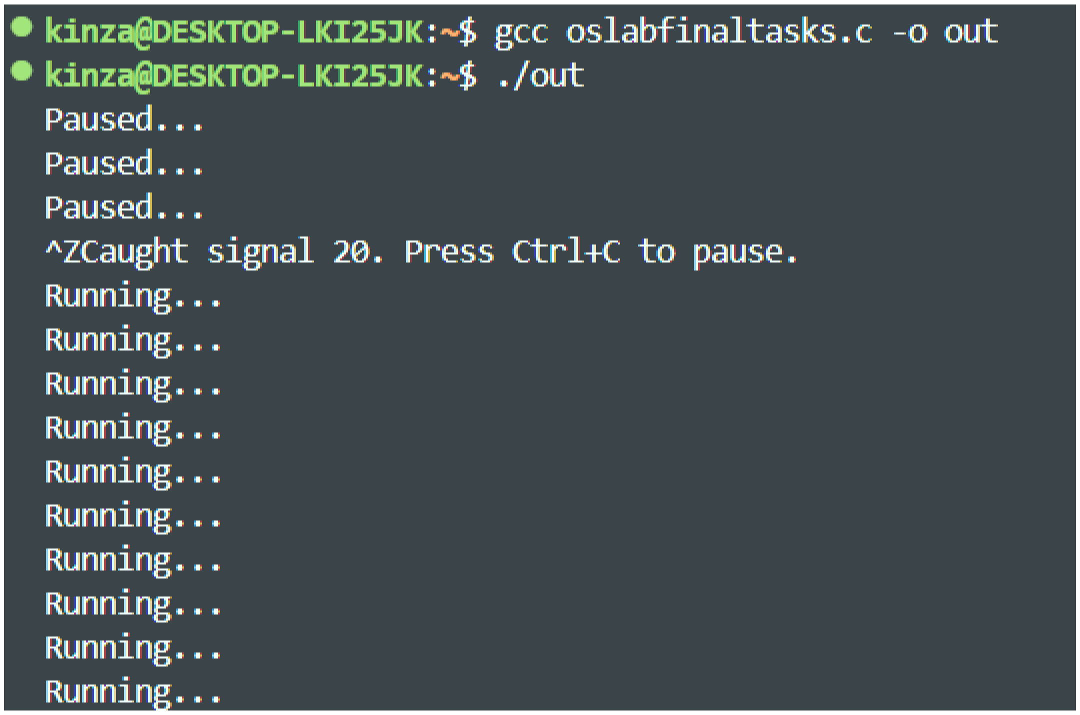


```
kinza@DESKTOP-LKI25JK:~$ gcc oslabfinaltasks.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out
Running...
Running...
Running...
^CCaught signal 2
```

5. Develop a C program that simulates a countdown timer from 10. If the user presses Ctrl+C, catch SIGINT and pause the timer. When the user presses Ctrl+Z, catch SIGTSTP to resume the countdown.

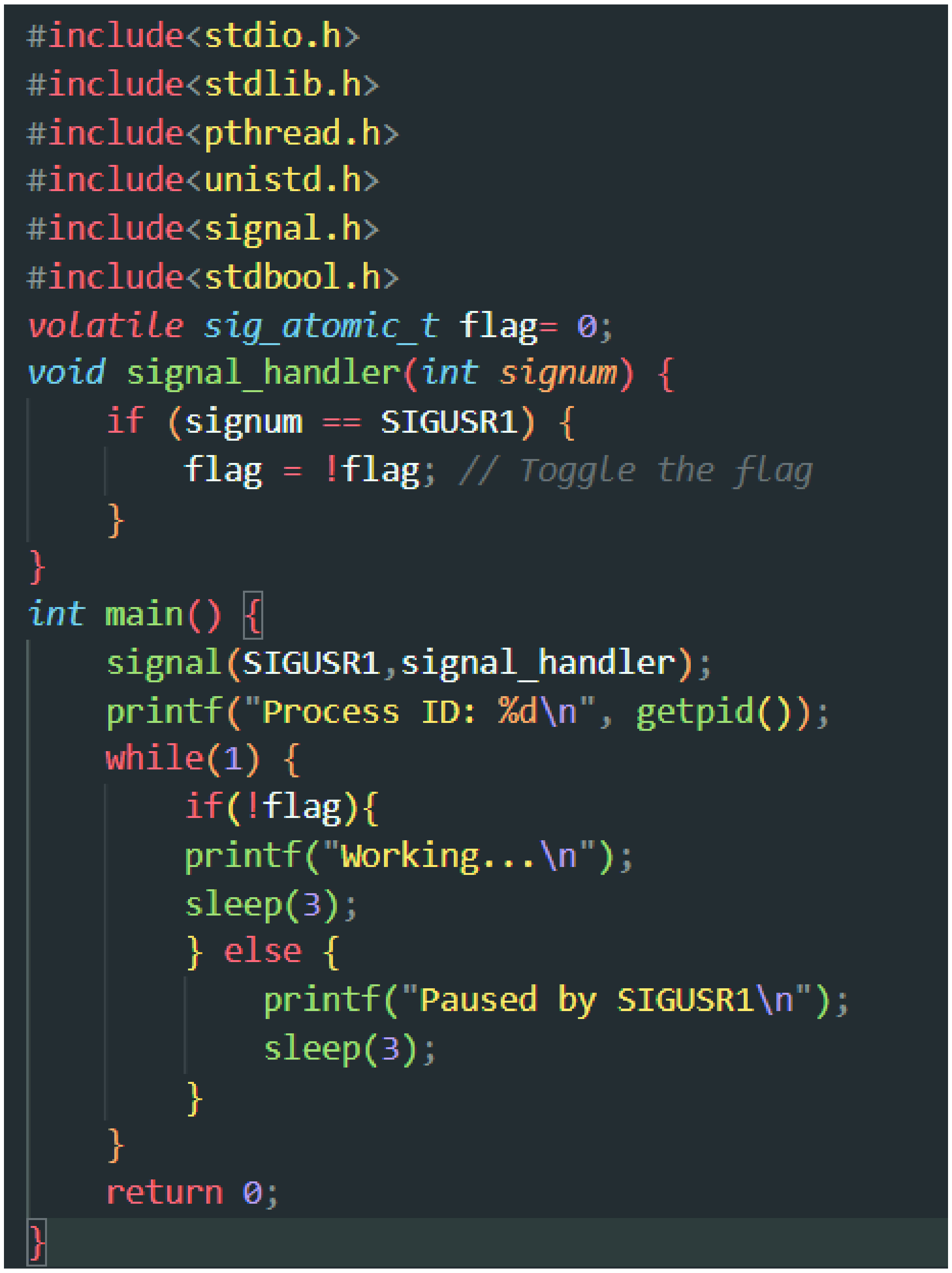


```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#include<signal.h>
#include<stdbool.h>
bool paused=1;
void signal_stsp(int signum) {
    paused =0;
    printf("Caught signal %d. Press Ctrl+C to pause. \n", signum);
}
void signal_handler(int signum) {
    paused=1;
    printf("Caught signal %d. Press Ctrl+Z to resume. \n", signum);
}
int main() {
    signal(SIGINT,signal_handler);
    signal(SIGTSTP,signal_stsp);
    int i=10;
    while(i>0) {
        if(!paused) {
            printf("Running...\n");
            sleep(1);
            i--;
        } else {
            printf("Paused...\n");
            sleep(1);
        }
    }
    return 0;
}
```

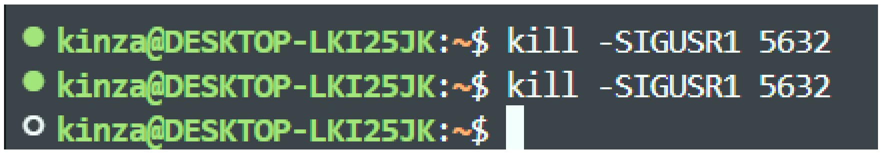


```
● kinza@DESKTOP-LKI25JK:~$ gcc oslabfinaltasks.c -o out
● kinza@DESKTOP-LKI25JK:~$ ./out
  Paused...
  Paused...
  Paused...
  ^ZCaught signal 20. Press Ctrl+C to pause.
  Running...
  Running...
  Running...
  Running...
  Running...
  Running...
  Running...
  Running...
  Running...
  Running...
```

6. Create a C program that runs an infinite loop, printing “Working...” every 3 seconds. Set up a signal handler for SIGUSR1 such that when the signal is received, a global flag is toggled between 1 and 0. If the flag is 1, the process should print “Paused by SIGUSR1” instead of “Working...”. Sending the signal again should resume the normal output.

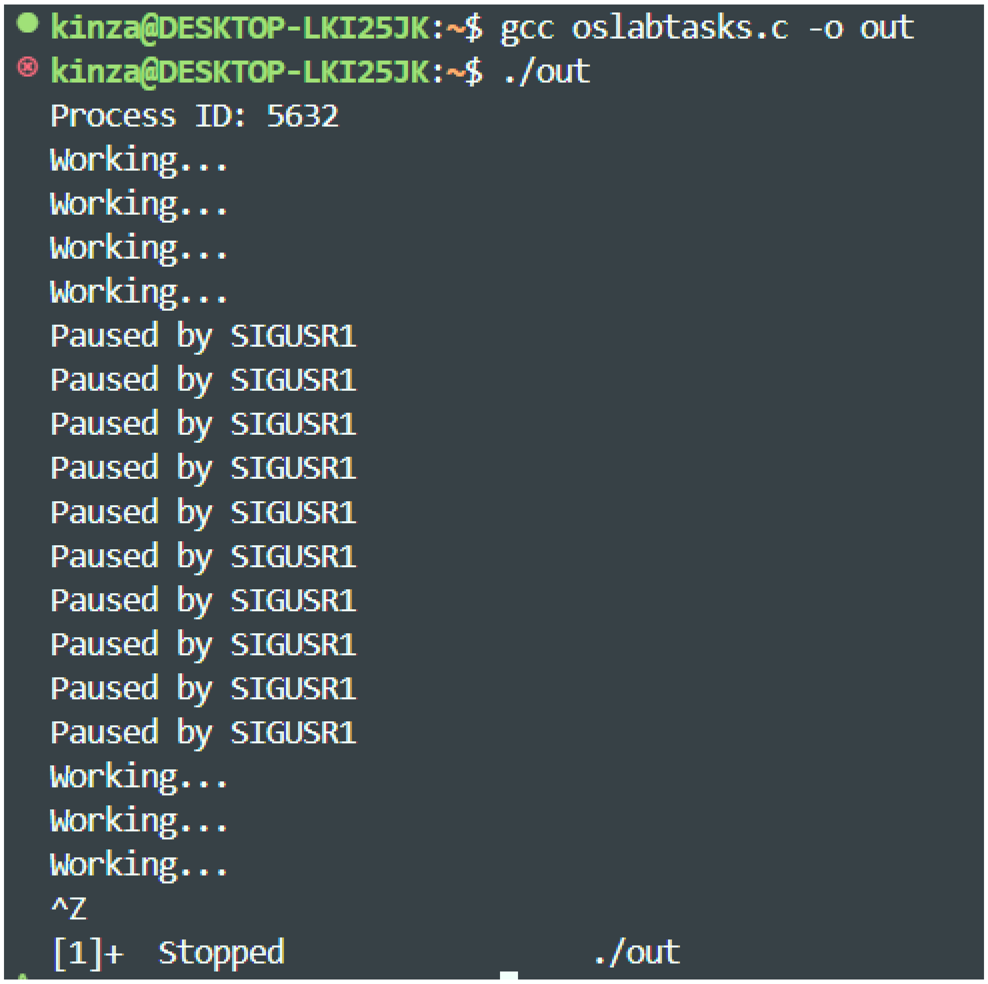


```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#include<signal.h>
#include<stdbool.h>
volatile sig_atomic_t flag= 0;
void signal_handler(int signum) {
    if (signum == SIGUSR1) {
        flag = !flag; // Toggle the flag
    }
}
int main() {
    signal(SIGUSR1,signal_handler);
    printf("Process ID: %d\n", getpid());
    while(1) {
        if(!flag){
        printf("Working...\n");
        sleep(3);
        } else {
            printf("Paused by SIGUSR1\n");
            sleep(3);
        }
    }
    return 0;
}
```



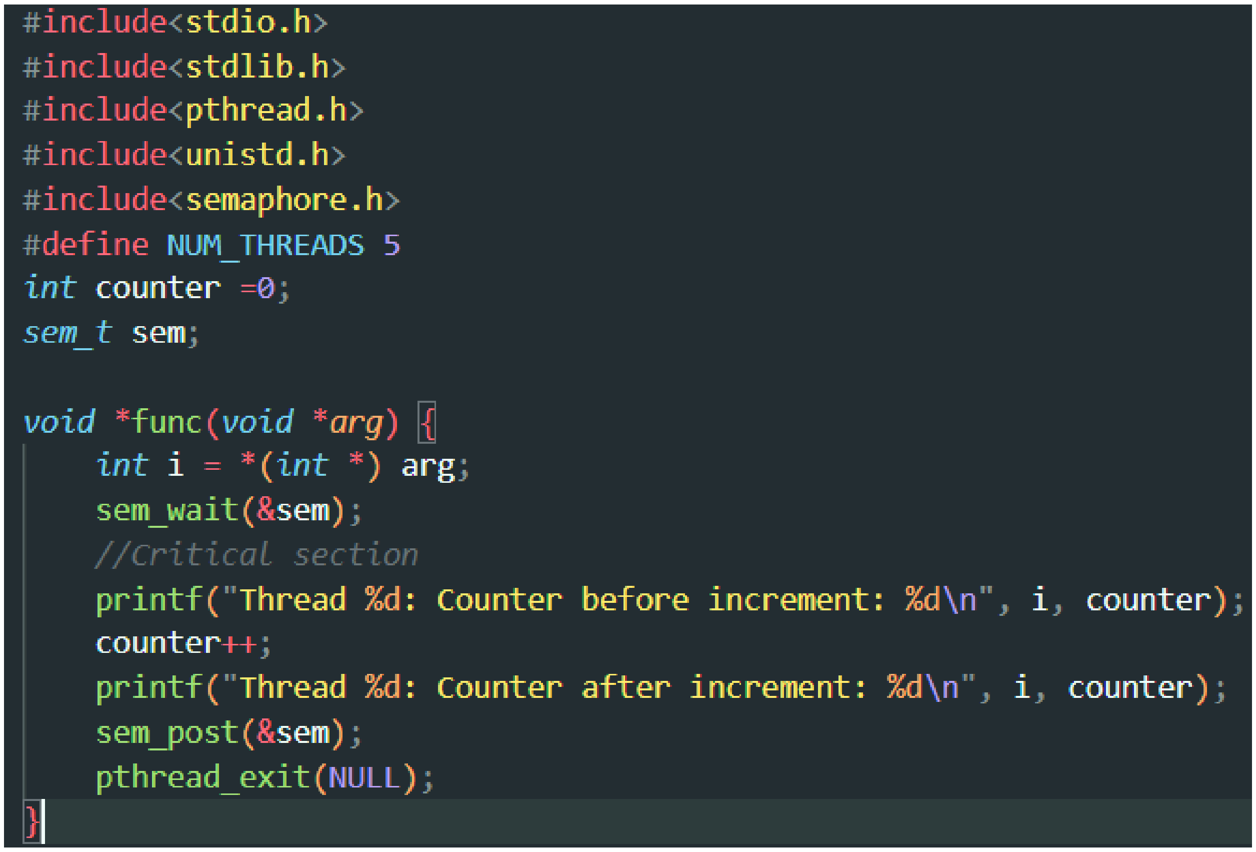
```
kinza@DESKTOP-LKI25JK:~$ kill -SIGUSR1 5632
kinza@DESKTOP-LKI25JK:~$ kill -SIGUSR1 5632
kinza@DESKTOP-LKI25JK:~$
```

Sending the signal kill -SIGUSR1 <pid> is used to pause and sending the signal kill -SIGUSR1 <pid> resumes it.

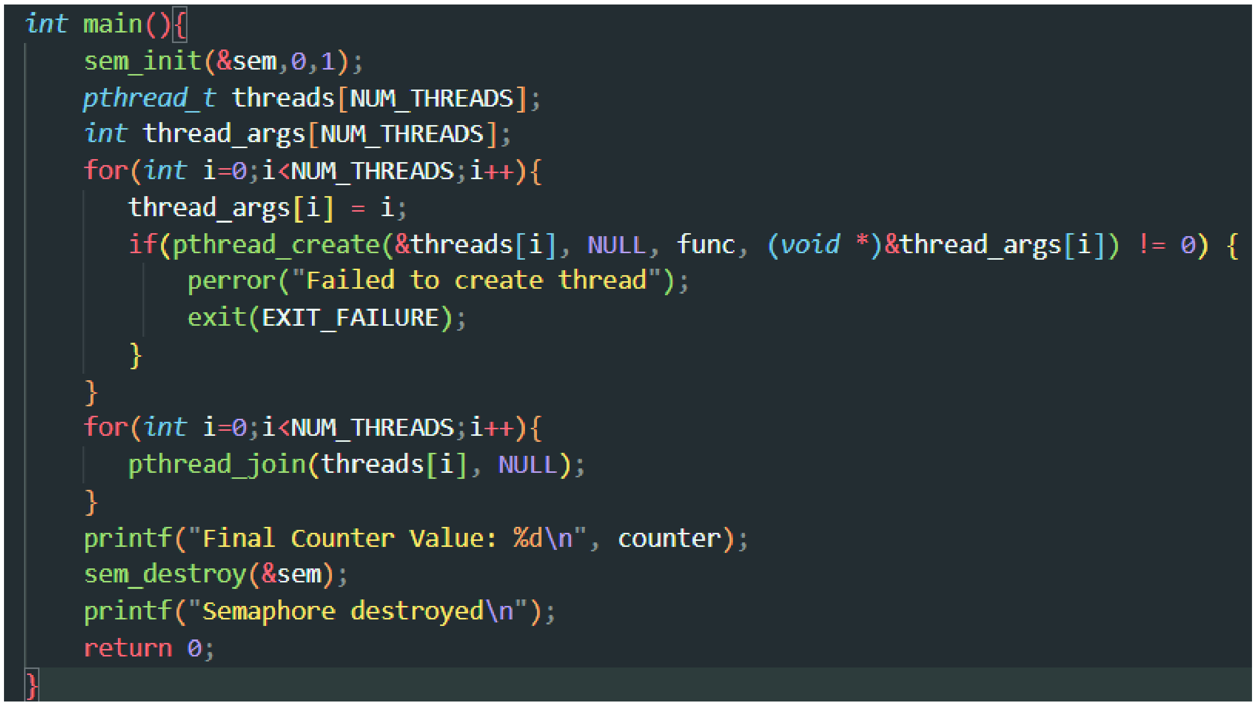


```
kinza@DESKTOP-LKI25JK:~$ gcc oslabtasks.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out
Process ID: 5632
Working...
Working...
Working...
Working...
Paused by SIGUSR1
Paused by SIGUSR1
Paused by SIGUSR1
Paused by SIGUSR1
Paused by SIGUSR1
Paused by SIGUSR1
Paused by SIGUSR1
Paused by SIGUSR1
Paused by SIGUSR1
Paused by SIGUSR1
Working...
Working...
Working...
^Z
[1]+  Stopped                 ./out
```

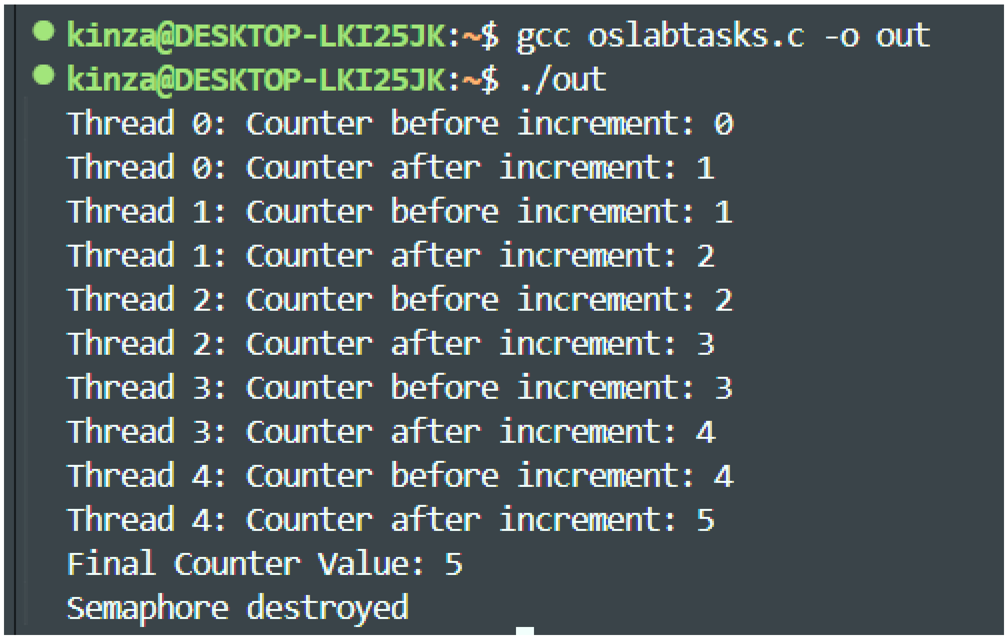
7. Create a C program that uses semaphores to control access to a shared counter variable updated by multiple threads. Ensure that only one thread can update the counter at a time.



```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#include<semaphore.h>
#define NUM_THREADS 5
int counter =0;
sem_t sem;

void *func(void *arg) {
    int i = *(int *) arg;
    sem_wait(&sem);
    //Critical section
    printf("Thread %d: Counter before increment: %d\n", i, counter);
    counter++;
    printf("Thread %d: Counter after increment: %d\n", i, counter);
    sem_post(&sem);
    pthread_exit(NULL);
}
```

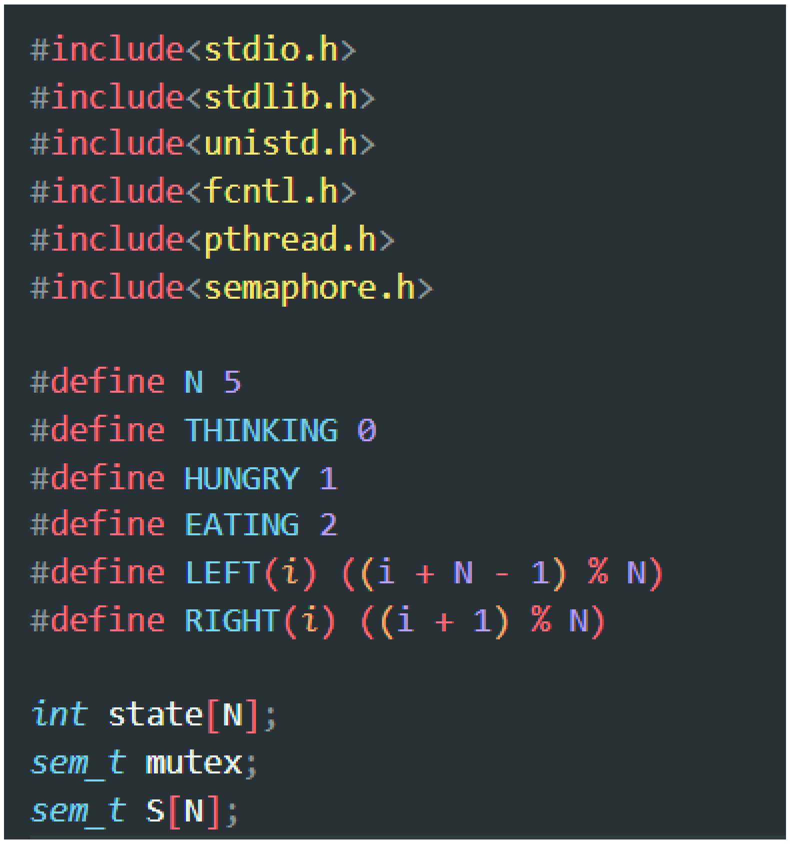


```c
int main(){
    sem_init(&sem,0,1);
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    for(int i=0;i<NUM_THREADS;i++){
        thread_args[i] = i;
        if(pthread_create(&threads[i], NULL, func, (void *)&thread_args[i]) != 0) {
            perror("Failed to create thread");
            exit(EXIT_FAILURE);
        }
    }
    for(int i=0;i<NUM_THREADS;i++){
        pthread_join(threads[i], NULL);
    }
    printf("Final Counter Value: %d\n", counter);
    sem_destroy(&sem);
    printf("Semaphore destroyed\n");
    return 0;
}
```

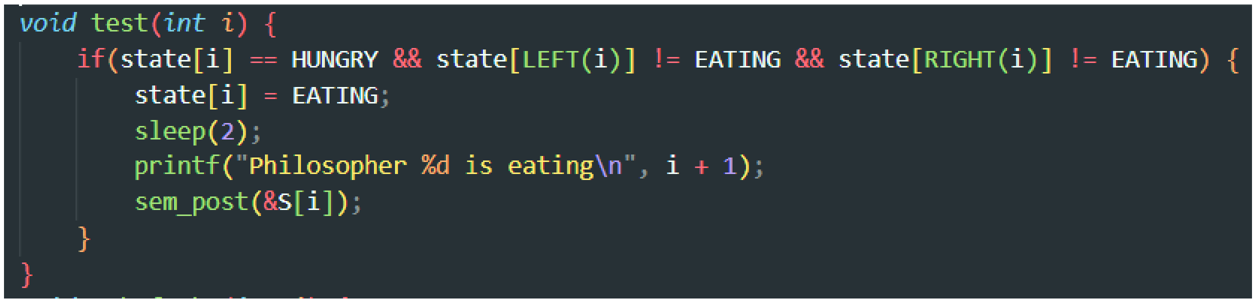


```
kinza@DESKTOP-LKI25JK:~$ gcc oslabtasks.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out
Thread 0: Counter before increment: 0
Thread 0: Counter after increment: 1
Thread 1: Counter before increment: 1
Thread 1: Counter after increment: 2
Thread 2: Counter before increment: 2
Thread 2: Counter after increment: 3
Thread 3: Counter before increment: 3
Thread 3: Counter after increment: 4
Thread 4: Counter before increment: 4
Thread 4: Counter after increment: 5
Final Counter Value: 5
Semaphore destroyed
```

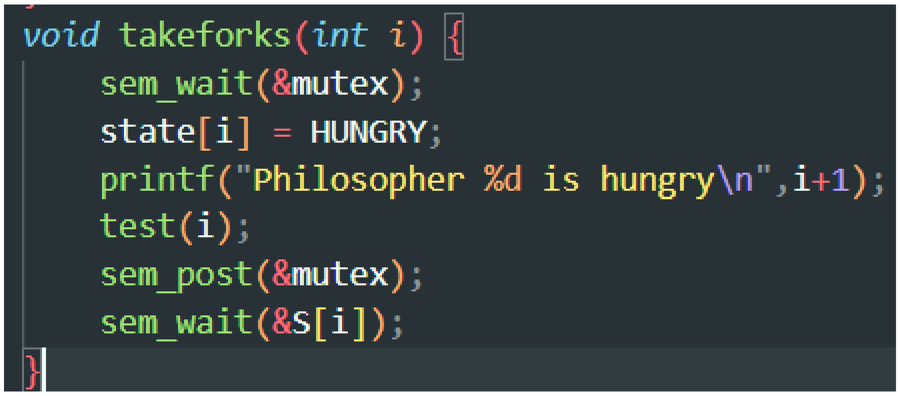
8. Implement the Dining Philosophers problem using semaphores. Ensure no deadlock occurs and neighboring philosophers do not eat simultaneously.



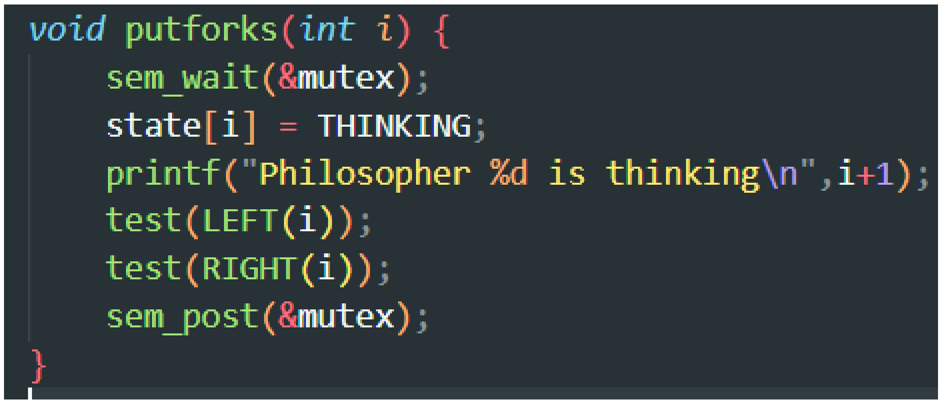
```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>
#include<pthread.h>
#include<semaphore.h>

#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT(i) ((i + N - 1) % N)
#define RIGHT(i) ((i + 1) % N)

int state[N];
sem_t mutex;
sem_t S[N];
```



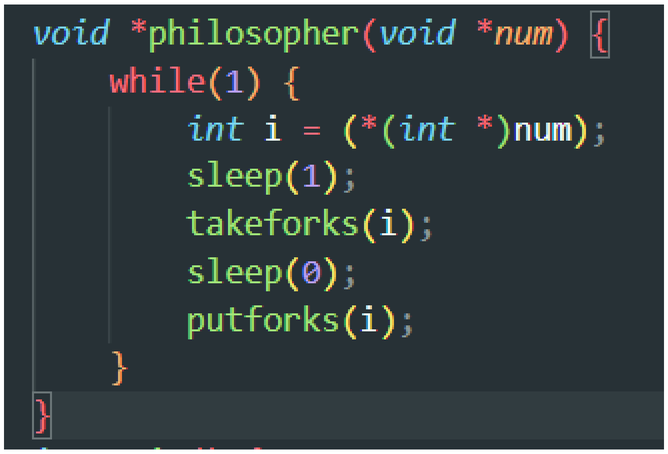
```c
void test(int i) {
    if(state[i] == HUNGRY && state[LEFT(i)] != EATING && state[RIGHT(i)] != EATING) {
        state[i] = EATING;
        sleep(2);
        printf("Philosopher %d is eating\n", i + 1);
        sem_post(&S[i]);
    }
}
```



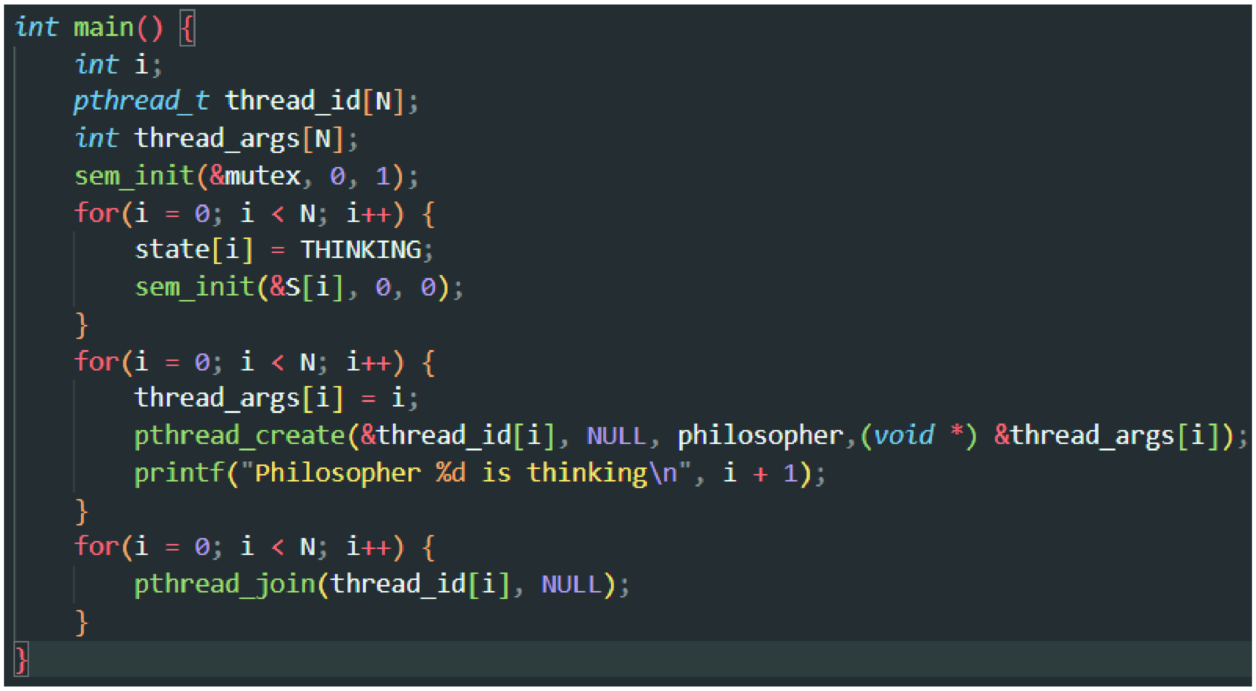
```c
void takeforks(int i) {
    sem_wait(&mutex);
    state[i] = HUNGRY;
    printf("Philosopher %d is hungry\n",i+1);
    test(i);
    sem_post(&mutex);
    sem_wait(&S[i]);
}
```



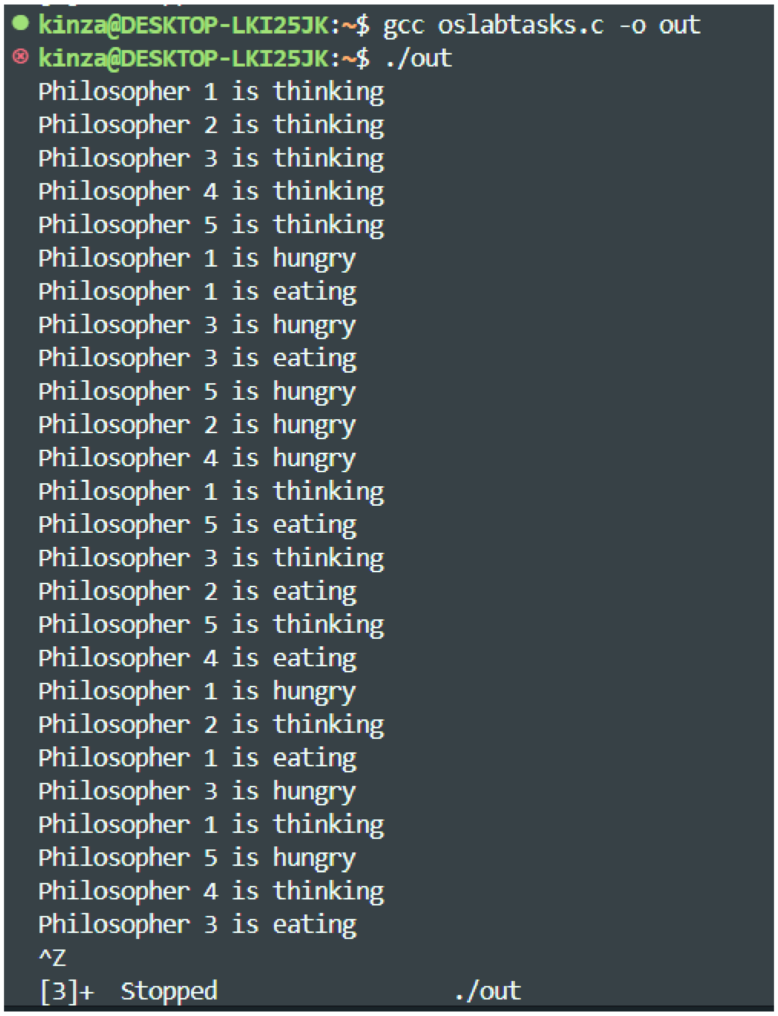
```c
void putforks(int i) {
    sem_wait(&mutex);
    state[i] = THINKING;
    printf("Philosopher %d is thinking\n",i+1);
    test(LEFT(i));
    test(RIGHT(i));
    sem_post(&mutex);
}
```



```c
void *philosopher(void *num) {
    while(1) {
        int i = (*(int *)num);
        sleep(1);
        takeforks(i);
        sleep(0);
        putforks(i);
    }
}
```

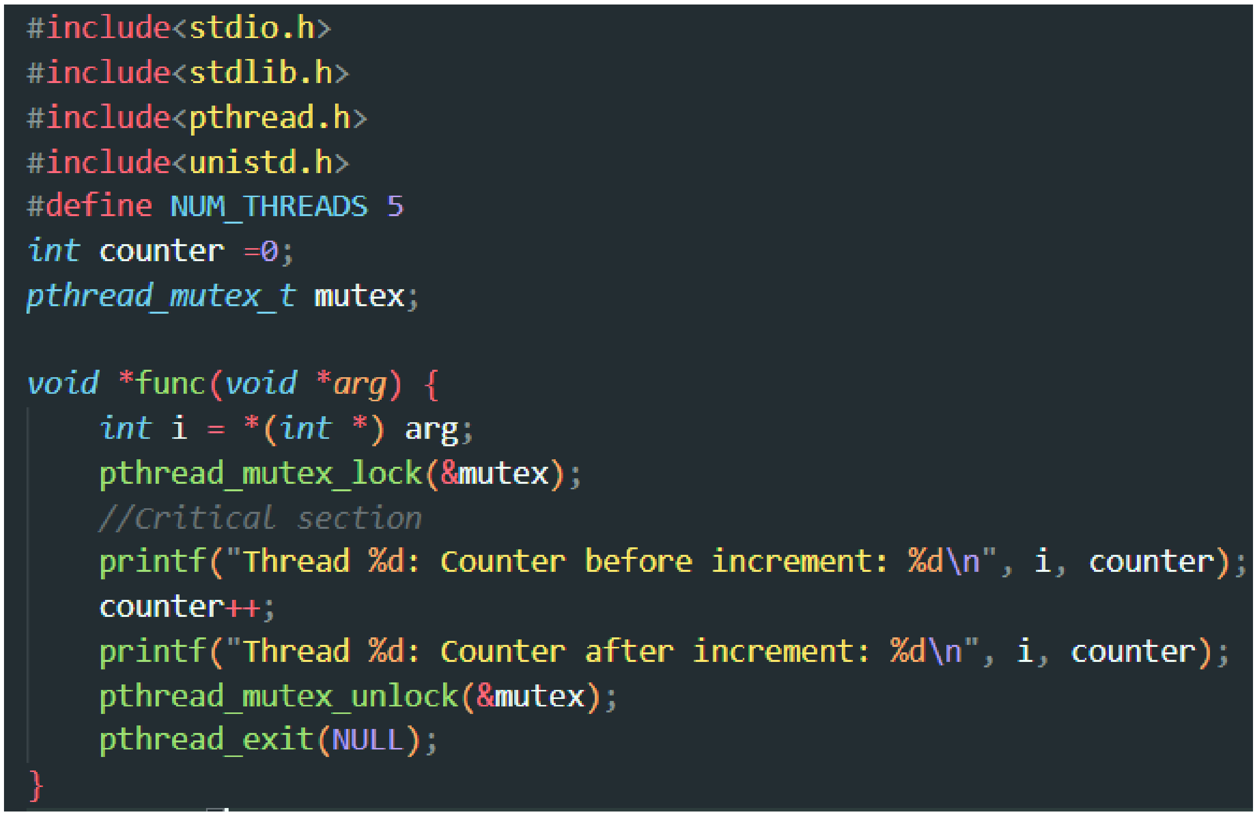


```c
int main() {
    int i;
    pthread_t thread_id[N];
    int thread_args[N];
    sem_init(&mutex, 0, 1);
    for(i = 0; i < N; i++) {
        state[i] = THINKING;
        sem_init(&S[i], 0, 0);
    }
    for(i = 0; i < N; i++) {
        thread_args[i] = i;
        pthread_create(&thread_id[i], NULL, philosopher,(void *) &thread_args[i]);
        printf("Philosopher %d is thinking\n", i + 1);
    }
    for(i = 0; i < N; i++) {
        pthread_join(thread_id[i], NULL);
    }
}
```

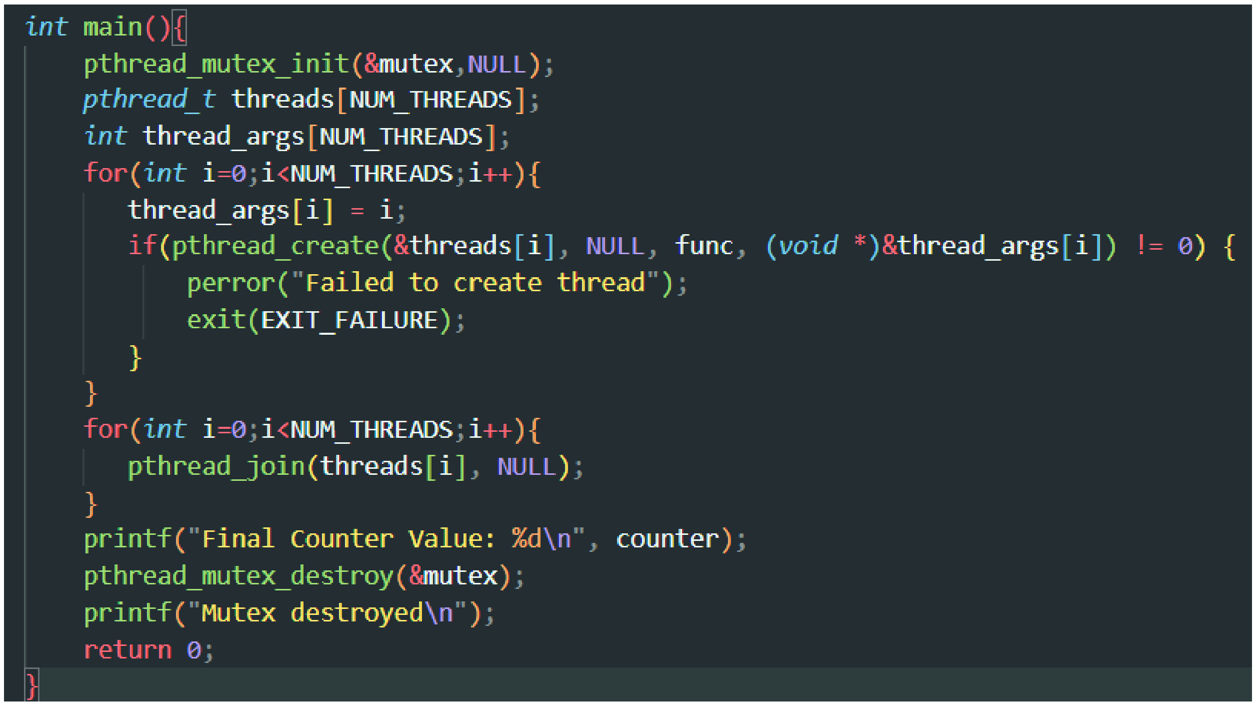


```
kinza@DESKTOP-LKI25JK:~$ gcc oslabtasks.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is hungry
Philosopher 1 is eating
Philosopher 3 is hungry
Philosopher 3 is eating
Philosopher 5 is hungry
Philosopher 2 is hungry
Philosopher 4 is hungry
Philosopher 1 is thinking
Philosopher 5 is eating
Philosopher 3 is thinking
Philosopher 2 is eating
Philosopher 5 is thinking
Philosopher 4 is eating
Philosopher 1 is hungry
Philosopher 2 is thinking
Philosopher 1 is eating
Philosopher 3 is hungry
Philosopher 1 is thinking
Philosopher 5 is hungry
Philosopher 4 is thinking
Philosopher 3 is eating
^Z
[3]+  Stopped                 ./out
```

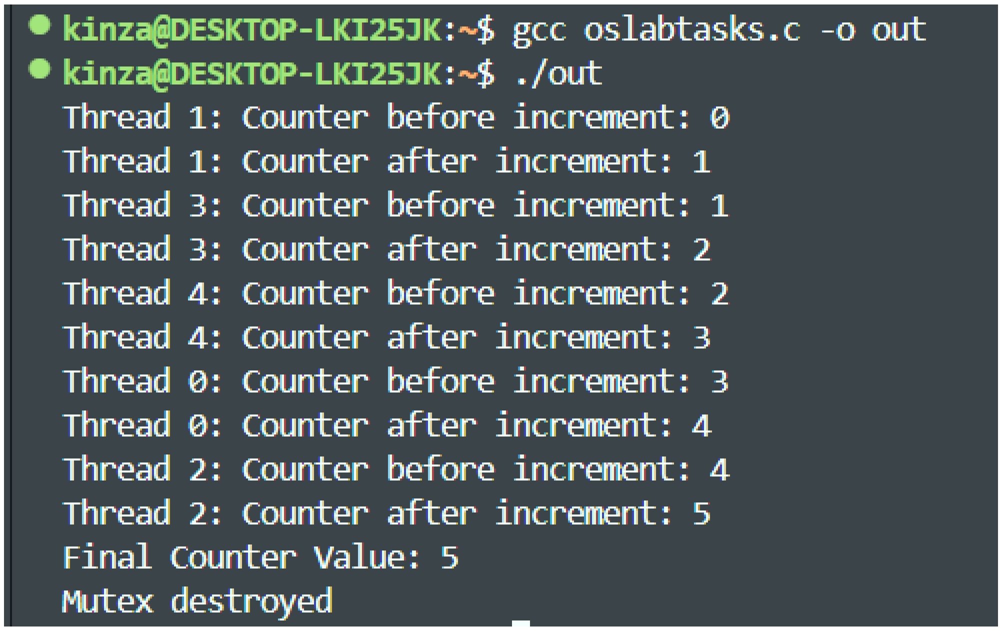
9. Create a C program where multiple threads increment a global variable. Use a mutex to ensure the updates are synchronized and no race condition occurs.



```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#define NUM_THREADS 5
int counter =0;
pthread_mutex_t mutex;

void *func(void *arg) {
    int i = *(int *) arg;
    pthread_mutex_lock(&mutex);
    //Critical section
    printf("Thread %d: Counter before increment: %d\n", i, counter);
    counter++;
    printf("Thread %d: Counter after increment: %d\n", i, counter);
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}
```

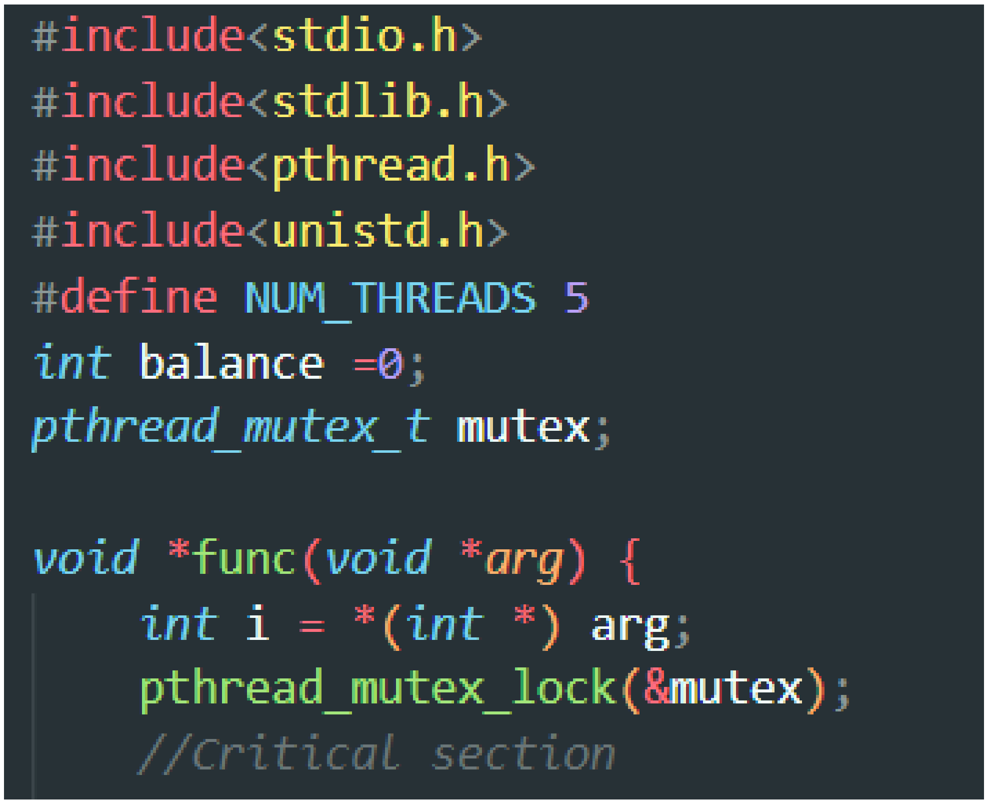


```c
int main(){
    pthread_mutex_init(&mutex,NULL);
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    for(int i=0;i<NUM_THREADS;i++){
        thread_args[i] = i;
        if(pthread_create(&threads[i], NULL, func, (void *)&thread_args[i]) != 0) {
            perror("Failed to create thread");
            exit(EXIT_FAILURE);
        }
    }
    for(int i=0;i<NUM_THREADS;i++){
        pthread_join(threads[i], NULL);
    }
    printf("Final Counter Value: %d\n", counter);
    pthread_mutex_destroy(&mutex);
    printf("Mutex destroyed\n");
    return 0;
}
```

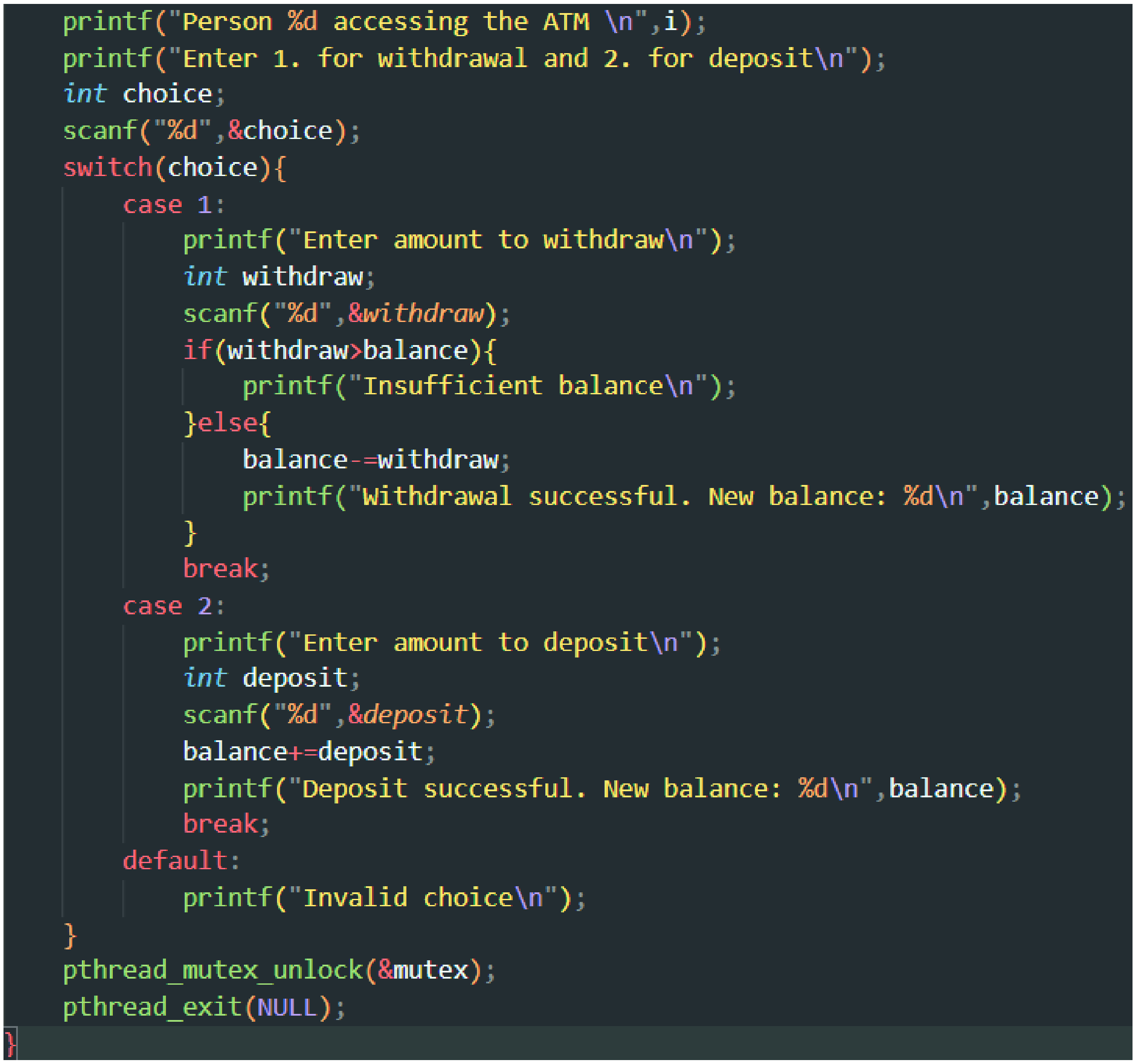


```
kinza@DESKTOP-LKI25JK:~$ gcc oslabtasks.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out
Thread 1: Counter before increment: 0
Thread 1: Counter after increment: 1
Thread 3: Counter before increment: 1
Thread 3: Counter after increment: 2
Thread 4: Counter before increment: 2
Thread 4: Counter after increment: 3
Thread 0: Counter before increment: 3
Thread 0: Counter after increment: 4
Thread 2: Counter before increment: 4
Thread 2: Counter after increment: 5
Final Counter Value: 5
Mutex destroyed
```

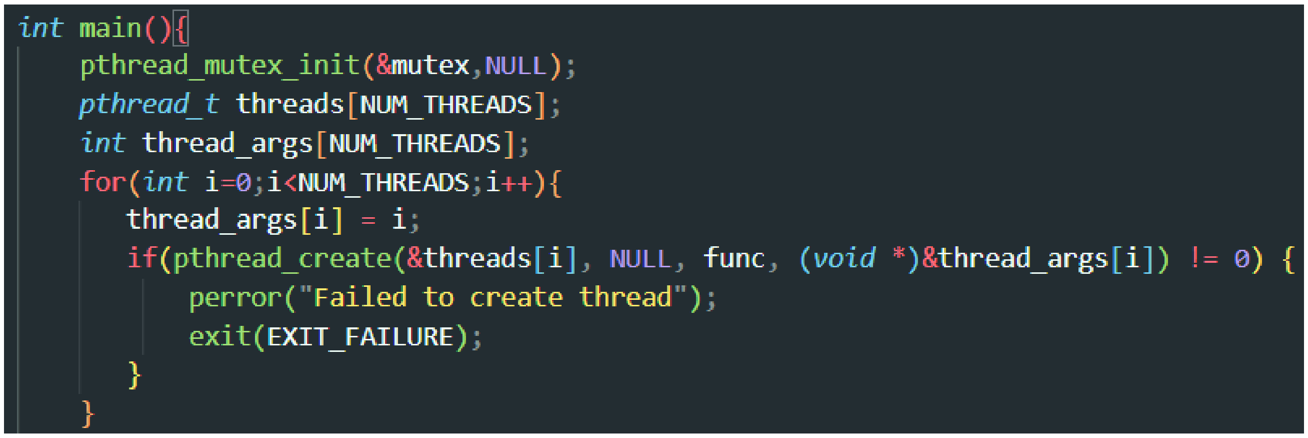
10. Create a program where multiple threads simulate people accessing an ATM. Use a mutex to restrict access to one person at a time, printing messages when each user accesses and leaves the ATM.



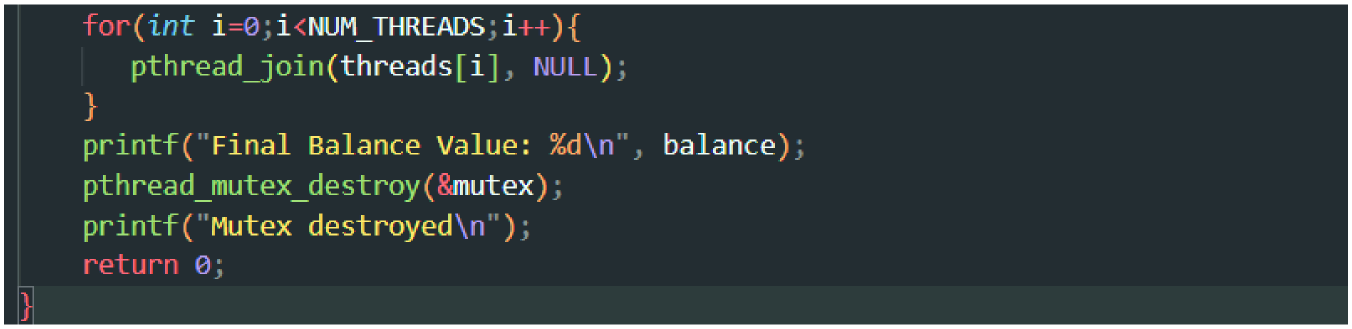
```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
#define NUM_THREADS 5
int balance =0;
pthread_mutex_t mutex;

void *func(void *arg) {
    int i = *(int *) arg;
    pthread_mutex_lock(&mutex);
    //Critical section
```



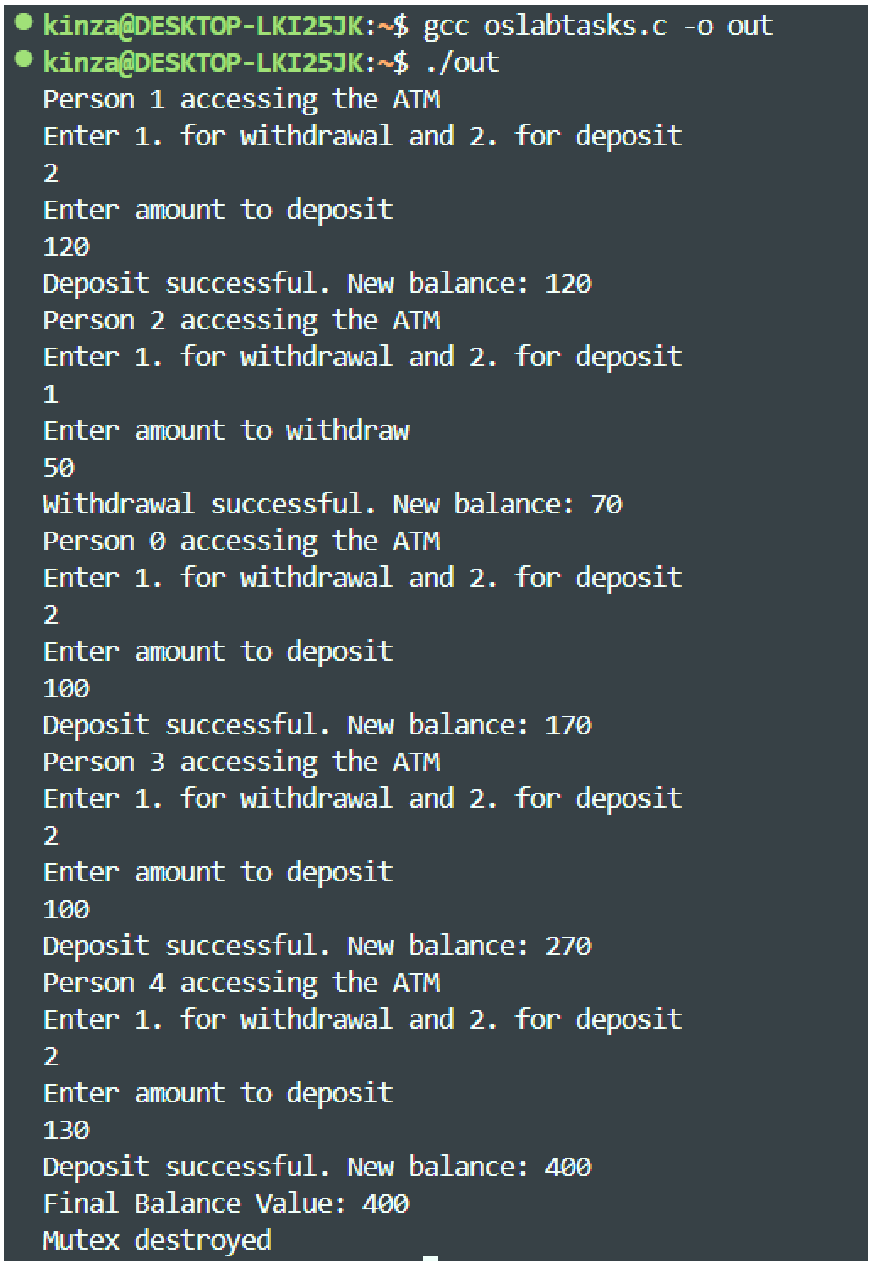
```c
    printf("Person %d accessing the ATM \n",i);
    printf("Enter 1. for withdrawal and 2. for deposit\n");
    int choice;
    scanf("%d",&choice);
    switch(choice){
        case 1:
            printf("Enter amount to withdraw\n");
            int withdraw;
            scanf("%d",&withdraw);
            if(withdraw>balance){
                printf("Insufficient balance\n");
            }else{
                balance-=withdraw;
                printf("Withdrawal successful. New balance: %d\n",balance);
            }
            break;
        case 2:
            printf("Enter amount to deposit\n");
            int deposit;
            scanf("%d",&deposit);
            balance+=deposit;
            printf("Deposit successful. New balance: %d\n",balance);
            break;
        default:
            printf("Invalid choice\n");
    }
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}
```



```c
int main(){
    pthread_mutex_init(&mutex,NULL);
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    for(int i=0;i<NUM_THREADS;i++){
        thread_args[i] = i;
        if(pthread_create(&threads[i], NULL, func, (void *)&thread_args[i]) != 0) {
            perror("Failed to create thread");
            exit(EXIT_FAILURE);
        }
    }
```



```c
    for(int i=0;i<NUM_THREADS;i++){
        pthread_join(threads[i], NULL);
    }
    printf("Final Balance Value: %d\n", balance);
    pthread_mutex_destroy(&mutex);
    printf("Mutex destroyed\n");
    return 0;
}
```



```
kinza@DESKTOP-LKI25JK:~$ gcc oslabtasks.c -o out
kinza@DESKTOP-LKI25JK:~$ ./out
Person 1 accessing the ATM
Enter 1. for withdrawal and 2. for deposit
2
Enter amount to deposit
120
Deposit successful. New balance: 120
Person 2 accessing the ATM
Enter 1. for withdrawal and 2. for deposit
1
Enter amount to withdraw
50
Withdrawal successful. New balance: 70
Person 0 accessing the ATM
Enter 1. for withdrawal and 2. for deposit
2
Enter amount to deposit
100
Deposit successful. New balance: 170
Person 3 accessing the ATM
Enter 1. for withdrawal and 2. for deposit
2
Enter amount to deposit
100
Deposit successful. New balance: 270
Person 4 accessing the ATM
Enter 1. for withdrawal and 2. for deposit
2
Enter amount to deposit
130
Deposit successful. New balance: 400
Final Balance Value: 400
Mutex destroyed
```