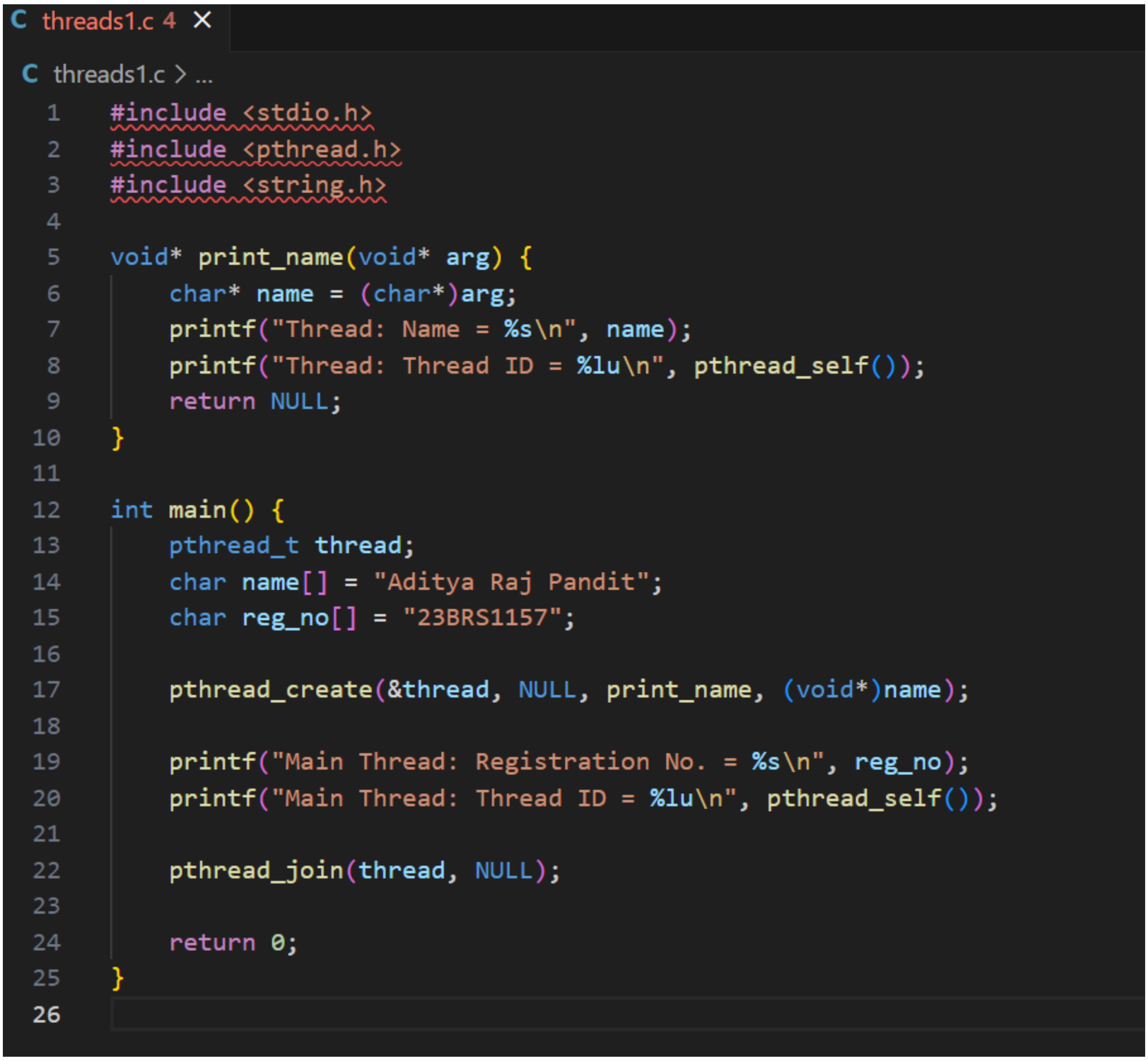
NAME : ADITYA RAJ PANDIT

REG NO : 23BRS1157

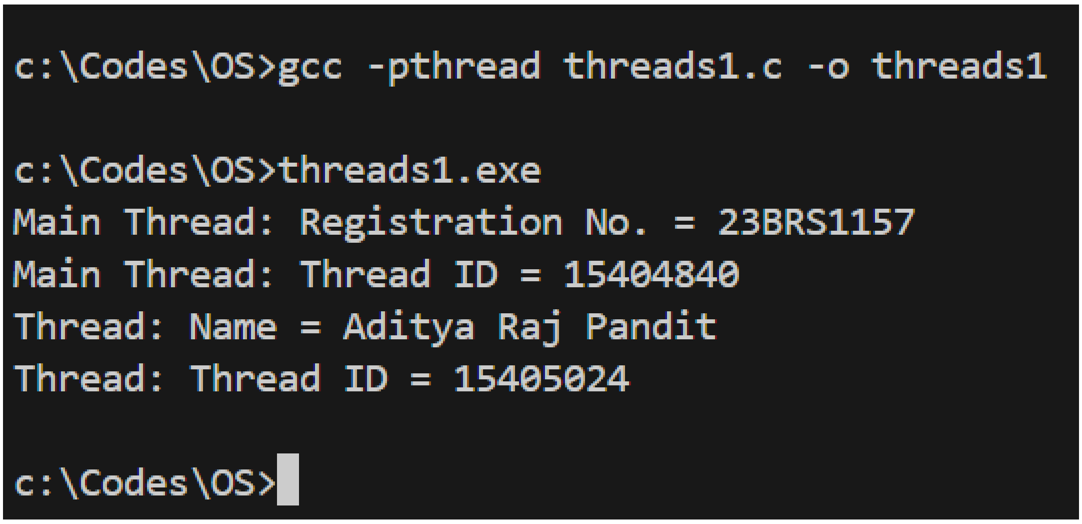
1) **Create a thread pass string as your name to the thread and also print your registration number to main thread and print their IDs.**

CODE :



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

void* print_name(void* arg) {
    char* name = (char*)arg;
    printf("Thread: Name = %s\n", name);
    printf("Thread: Thread ID = %lu\n", pthread_self());
    return NULL;
}

int main() {
    pthread_t thread;
    char name[] = "Aditya Raj Pandit";
    char reg_no[] = "23BRS1157";

    pthread_create(&thread, NULL, print_name, (void*)name);

    printf("Main Thread: Registration No. = %s\n", reg_no);
    printf("Main Thread: Thread ID = %lu\n", pthread_self());

    pthread_join(thread, NULL);

    return 0;
}

```

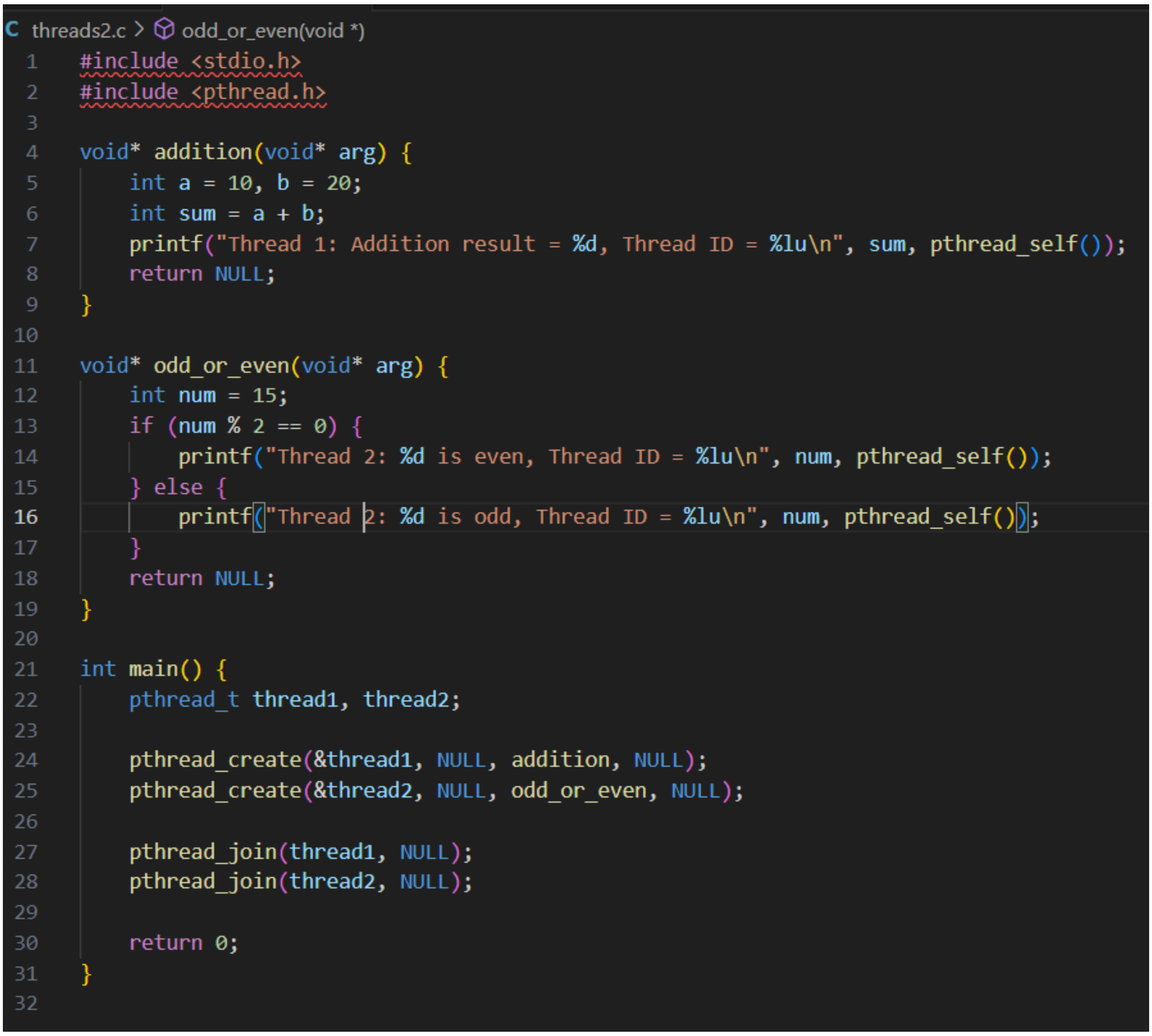
OUTPUT :



```
c:\Codes\OS>gcc -pthread threads1.c -o threads1

c:\Codes\OS>threads1.exe
Main Thread: Registration No. = 23BRS1157
Main Thread: Thread ID = 15404840
Thread: Name = Aditya Raj Pandit
Thread: Thread ID = 15405024

c:\Codes\OS>
```

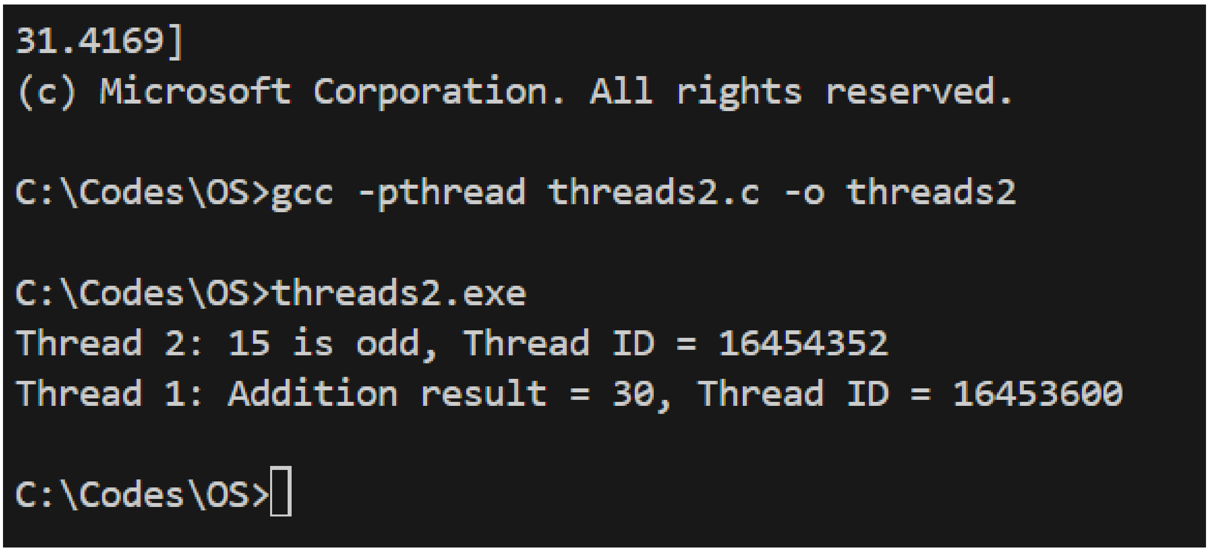
2) **Create two threads and display the two different function (Addition/ Odd or even ete.,) along with the corresponding thread_id. (pthread_self() function returns thread id)**

CODE:



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

void* addition(void* arg) {
    int a = 10, b = 20;
    int sum = a + b;
    printf("Thread 1: Addition result = %d, Thread ID = %lu\n", sum, pthread_self());
    return NULL;
}

void* odd_or_even(void* arg) {
    int num = 15;
    if (num % 2 == 0) {
        printf("Thread 2: %d is even, Thread ID = %lu\n", num, pthread_self());
    } else {
        printf("Thread 2: %d is odd, Thread ID = %lu\n", num, pthread_self());
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, addition, NULL);
    pthread_create(&thread2, NULL, odd_or_even, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}

```

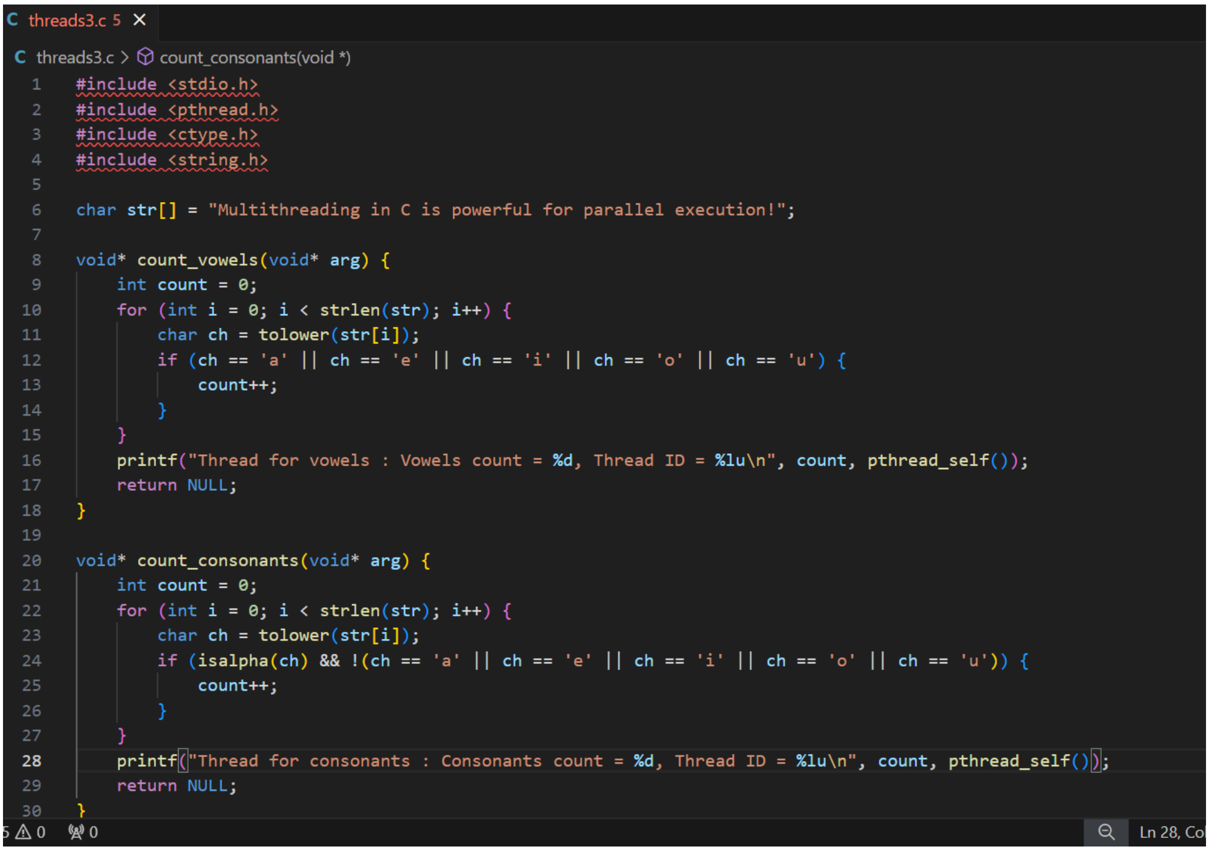
OUTPUT :



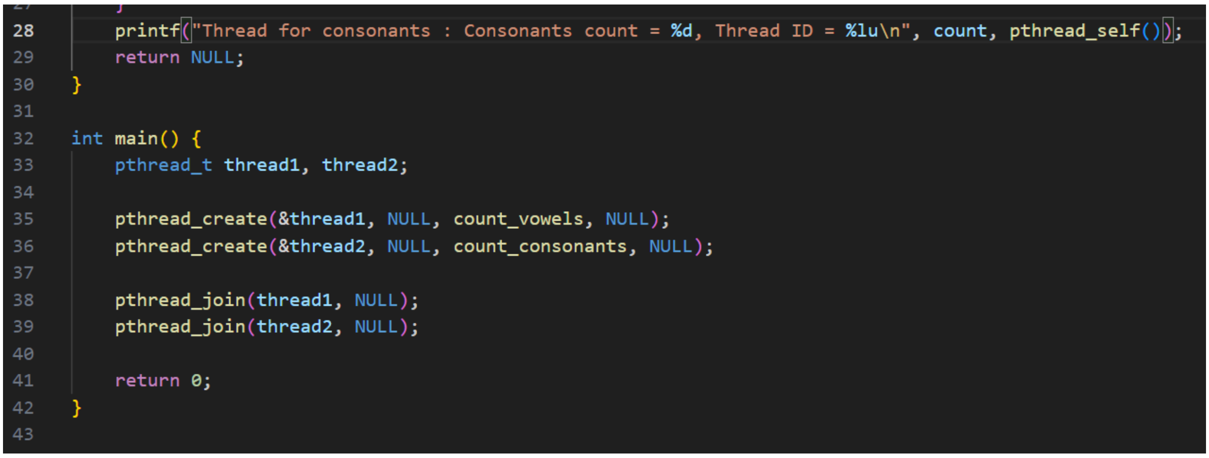
```
31.4169]
(c) Microsoft Corporation. All rights reserved.

C:\Codes\OS>gcc -pthread threads2.c -o threads2

C:\Codes\OS>threads2.exe
Thread 2: 15 is odd, Thread ID = 16454352
Thread 1: Addition result = 30, Thread ID = 16453600

C:\Codes\OS>
```

3) Design two threads to count the vowels and consonants either from text file or a given string.

CODE:

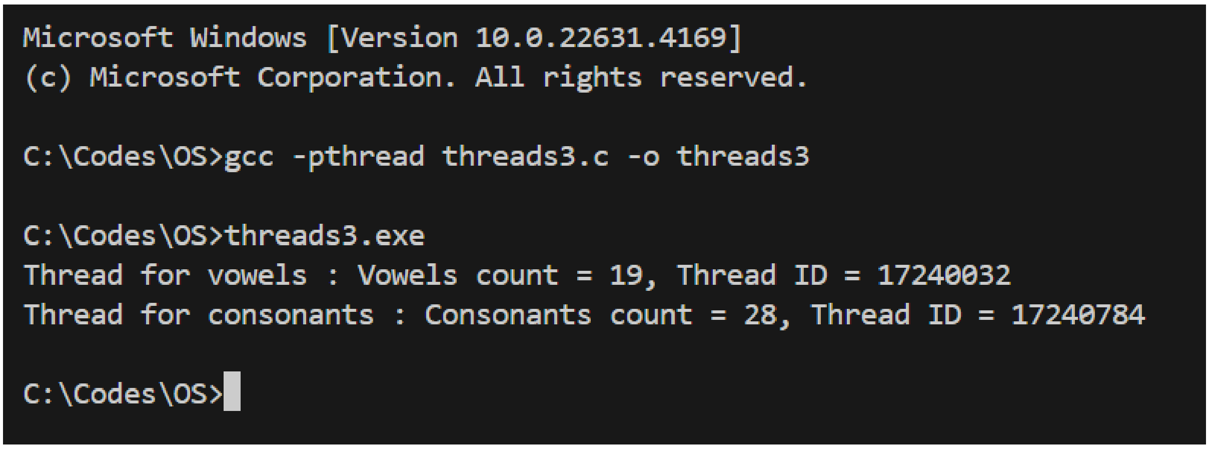


```c
#include <stdio.h>
#include <pthread.h>
#include <ctype.h>
#include <string.h>

char str[] = "Multithreading in C is powerful for parallel execution!";

void* count_vowels(void* arg) {
    int count = 0;
    for (int i = 0; i < strlen(str); i++) {
        char ch = tolower(str[i]);
        if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') {
            count++;
        }
    }
    printf("Thread for vowels : Vowels count = %d, Thread ID = %lu\n", count, pthread_self());
    return NULL;
}

void* count_consonants(void* arg) {
    int count = 0;
    for (int i = 0; i < strlen(str); i++) {
        char ch = tolower(str[i]);
        if (isalpha(ch) && !(ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u')) {
            count++;
        }
    }
    printf("Thread for consonants : Consonants count = %d, Thread ID = %lu\n", count, pthread_self());
    return NULL;
}
```



```c
    printf("Thread for consonants : Consonants count = %d, Thread ID = %lu\n", count, pthread_self());
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, count_vowels, NULL);
    pthread_create(&thread2, NULL, count_consonants, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

OUTPUT :

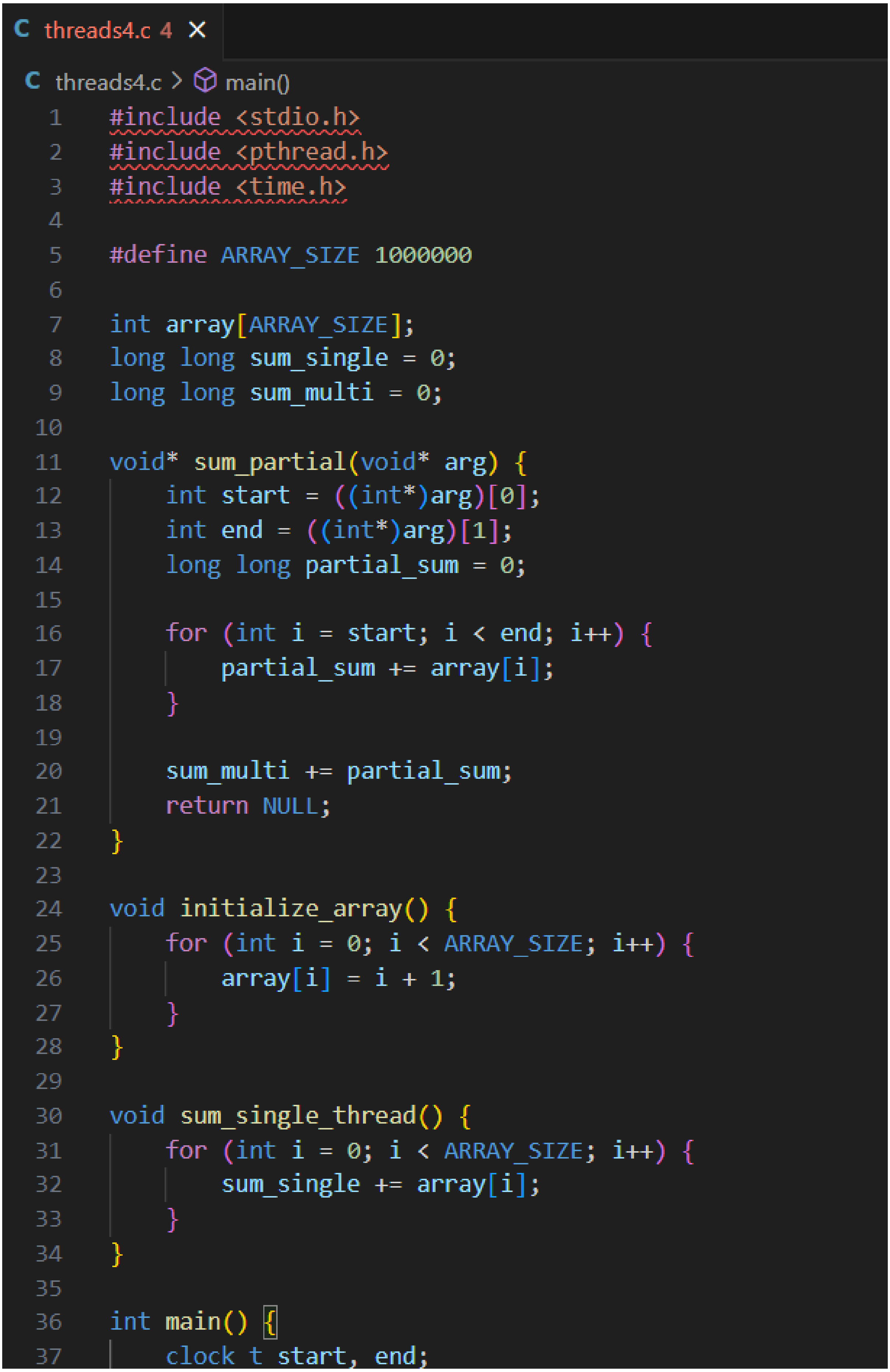


```
Microsoft Windows [Version 10.0.22631.4169]
(c) Microsoft Corporation. All rights reserved.

C:\Codes\OS>gcc -pthread threads3.c -o threads3

C:\Codes\OS>threads3.exe
Thread for vowels : Vowels count = 19, Thread ID = 17240032
Thread for consonants : Consonants count = 28, Thread ID = 17240784

C:\Codes\OS>
```

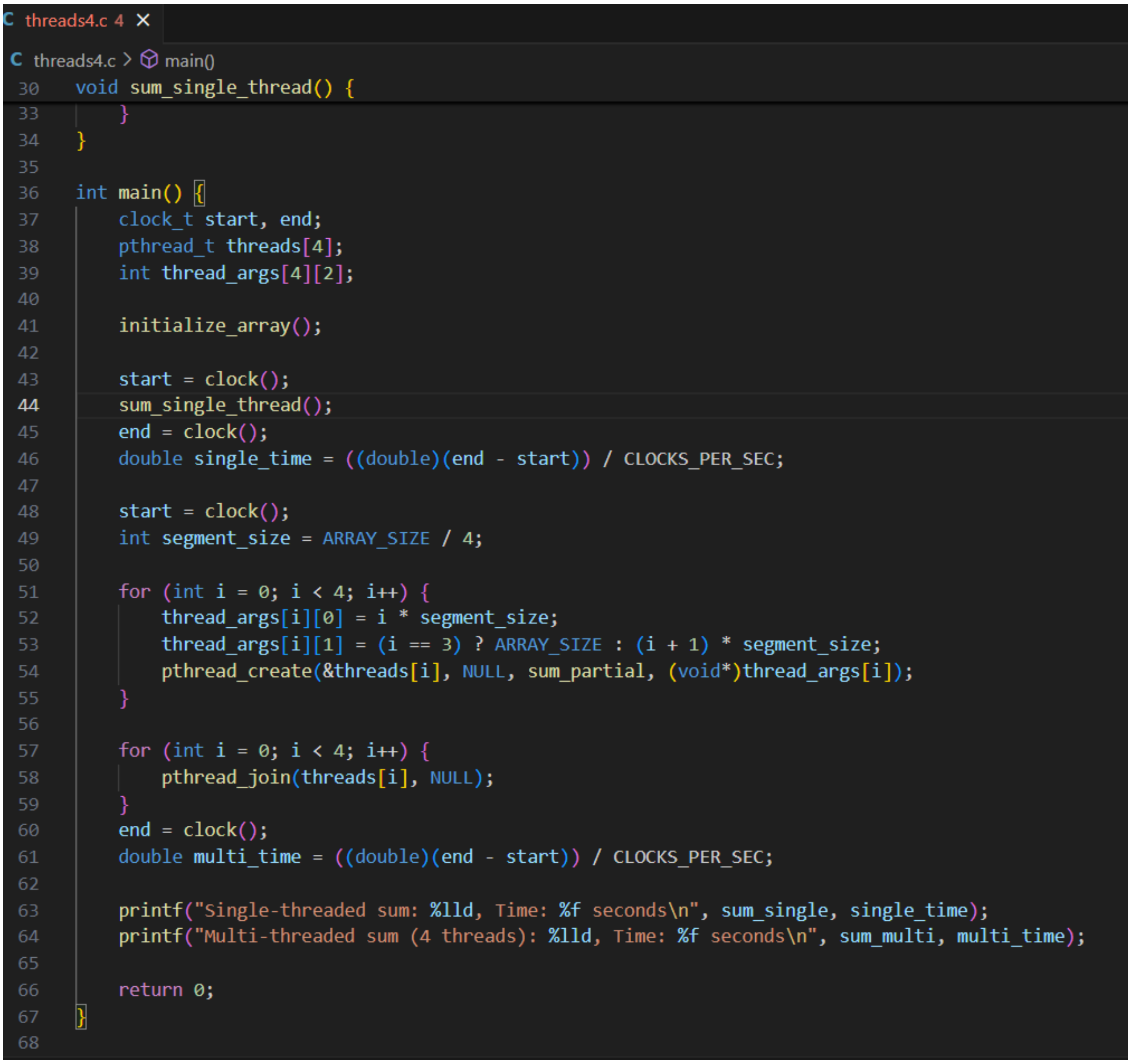
4) Write a program to sum up an array of 1000000 elements where elements are consecutive natural numbers.

ii) Create two or four threads for splitting the sum and find the execution time.

CODE :

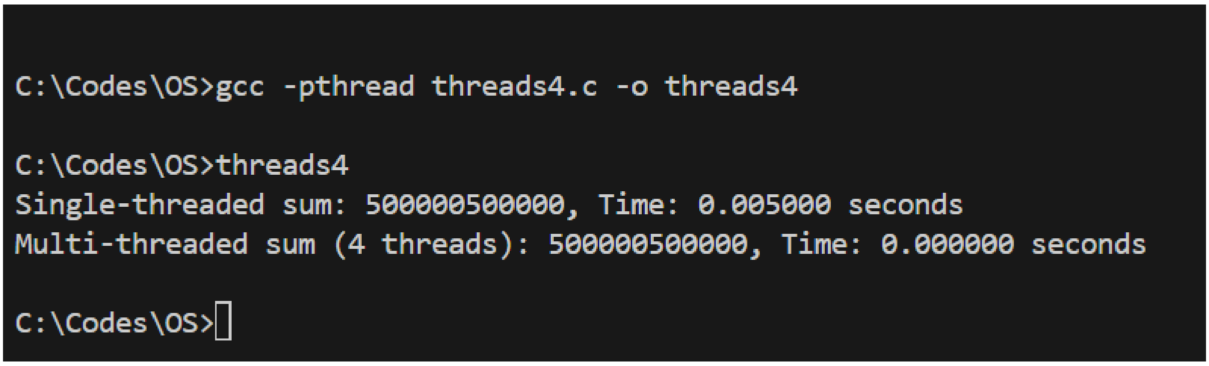


```c
#include <stdio.h>
#include <pthread.h>
#include <time.h>

#define ARRAY_SIZE 1000000

int array[ARRAY_SIZE];
long long sum_single = 0;
long long sum_multi = 0;

void* sum_partial(void* arg) {
    int start = ((int*)arg)[0];
    int end = ((int*)arg)[1];
    long long partial_sum = 0;

    for (int i = start; i < end; i++) {
        partial_sum += array[i];
    }

    sum_multi += partial_sum;
    return NULL;
}

void initialize_array() {
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = i + 1;
    }
}

void sum_single_thread() {
    for (int i = 0; i < ARRAY_SIZE; i++) {
        sum_single += array[i];
    }
}

int main() {
    clock_t start, end;
```



```c
void sum_single_thread() {
    }
}

int main() {
    clock_t start, end;
    pthread_t threads[4];
    int thread_args[4][2];

    initialize_array();

    start = clock();
    sum_single_thread();
    end = clock();
    double single_time = ((double)(end - start)) / CLOCKS_PER_SEC;

    start = clock();
    int segment_size = ARRAY_SIZE / 4;

    for (int i = 0; i < 4; i++) {
        thread_args[i][0] = i * segment_size;
        thread_args[i][1] = (i == 3) ? ARRAY_SIZE : (i + 1) * segment_size;
        pthread_create(&threads[i], NULL, sum_partial, (void*)thread_args[i]);
    }

    for (int i = 0; i < 4; i++) {
        pthread_join(threads[i], NULL);
    }
    end = clock();
    double multi_time = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("Single-threaded sum: %lld, Time: %f seconds\n", sum_single, single_time);
    printf("Multi-threaded sum (4 threads): %lld, Time: %f seconds\n", sum_multi, multi_time);

    return 0;
}
```

OUTPUT :



```
C:\Codes\OS>gcc -pthread threads4.c -o threads4

C:\Codes\OS>threads4
Single-threaded sum: 500000500000, Time: 0.005000 seconds
Multi-threaded sum (4 threads): 500000500000, Time: 0.000000 seconds

C:\Codes\OS>
```

iii) Analyse and comment on the execution time and justify the need of multithreaded programming.

Execution Time:

- **Single-threaded execution** processes the array sequentially, meaning it takes more time to sum all the elements as the workload is handled by only one core.
- **Multi-threaded execution** splits the array into segments, distributing the workload across multiple cores or processors. This results in faster execution because modern systems have multiple cores capable of running threads concurrently.

Need for Multithreading:

- **Efficiency**: By using multiple threads, the program can utilize CPU cores more effectively, leading to a reduction in execution time for large tasks.
- **Scalability**: As the size of the array or task increases, the benefits of multithreading become more evident. Multithreading allows for parallelism, which improves performance in computationally intensive applications.
- **Responsiveness**: In many real-world scenarios, tasks such as I/O operations, complex computations, and concurrent handling of multiple tasks can be significantly improved through multithreaded programming.