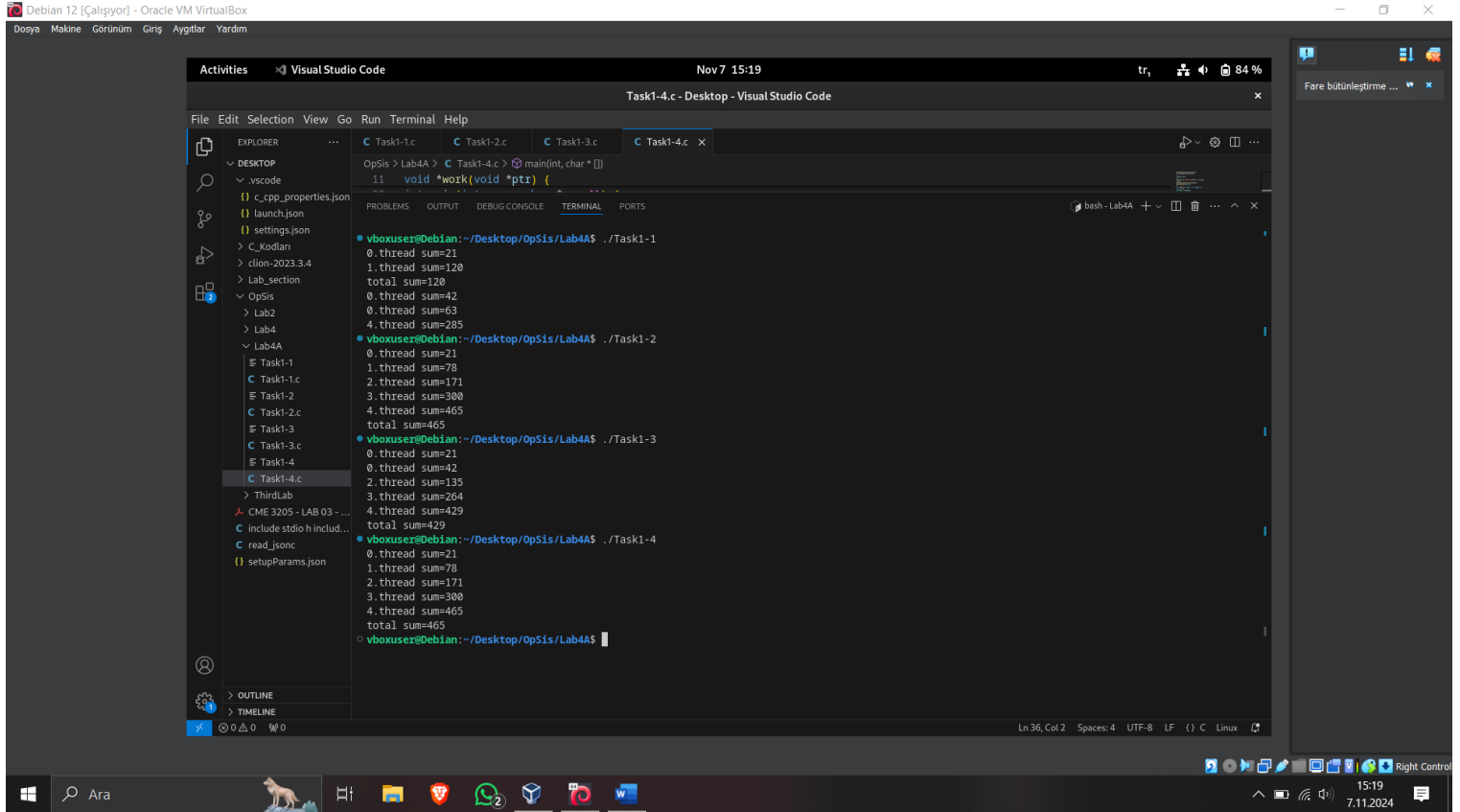


Task 1

Outputs:



```
OpSis > Lab4A > C Task1-4.c > @ main(int, char* [])
11 void *work(void *ptr) {
...
vboxuser@Debian: ~/Desktop/OpSis/Lab4A$ ./Task1-1
0.thread sum=21
1.thread sum=120
total sum=120
vboxuser@Debian: ~/Desktop/OpSis/Lab4A$ ./Task1-2
0.thread sum=21
1.thread sum=78
2.thread sum=171
3.thread sum=300
4.thread sum=465
total sum=465
vboxuser@Debian: ~/Desktop/OpSis/Lab4A$ ./Task1-3
0.thread sum=21
0.thread sum=42
2.thread sum=135
3.thread sum=264
4.thread sum=429
total sum=429
vboxuser@Debian: ~/Desktop/OpSis/Lab4A$ ./Task1-4
0.thread sum=21
1.thread sum=78
2.thread sum=171
3.thread sum=300
4.thread sum=465
total sum=465
vboxuser@Debian: ~/Desktop/OpSis/Lab4A$
```

Difference between these programs:

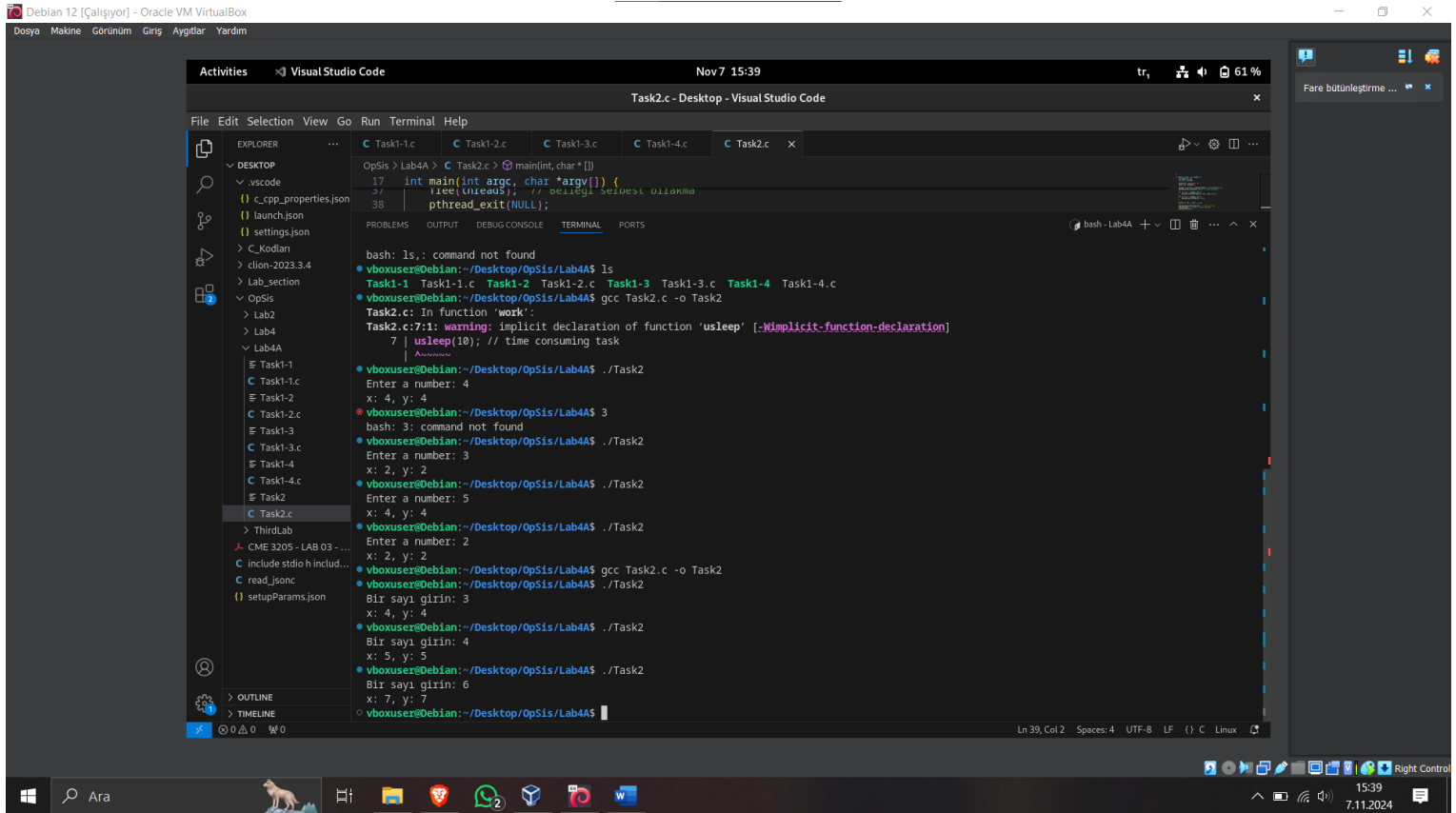
First program creates five threads, each calculating a portion of the sum from a shared array. However, `pthread_join` is not used within the loop that creates threads, meaning the main thread doesn't wait for individual threads to finish. As a result, the `printf` for total sum may run before the threads complete, leading to an incomplete or incorrect final sum.

In second program, each thread is created, and immediately after creation, `pthread_join` is called to wait for that specific thread to complete. This ensures that each thread completes its operation before the next one starts. Consequently, the output will be consistent since only one thread runs at a time.

Third program creates all threads first, then waits for all of them to finish by calling `pthread_join` for each thread in a separate loop after creation. Since `x` and `sum` are shared without protection, race condition occurs, potentially leading to inconsistent outputs.

Fourth program adds a mutex lock around the critical section where the sum is calculated, and x is updated. By using `pthread_mutex_lock` and `pthread_mutex_unlock`, it prevents race conditions, ensuring that only one thread accesses the shared resources at a time.

TASK 2



```
OpSis > Lab4A > C Task2.c > main(int, char*[])
17 int main(int argc, char *argv[]) {
18     pthread_t t1, t2, t3, t4;
19     pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
20     pthread_exit(NULL);
21 }
```

```
bash: ls: command not found
vboxuser@Debian:~/Desktop/OpSis/Lab4A$ ls
Task1-1 Task1-1.c Task1-2 Task1-2.c Task1-3 Task1-3.c Task1-4 Task1-4.c
vboxuser@Debian:~/Desktop/OpSis/Lab4A$ gcc Task2.c -o Task2
Task2.c: In function 'work':
Task2.c:7:1: warning: implicit declaration of function 'usleep' [-Wimplicit-function-declaration]
7 |     usleep(10); // time consuming task
  |     ^~~~~~
vboxuser@Debian:~/Desktop/OpSis/Lab4A$ ./Task2
Enter a number: 4
x: 4, y: 4
vboxuser@Debian:~/Desktop/OpSis/Lab4A$ 3
bash: 3: command not found
vboxuser@Debian:~/Desktop/OpSis/Lab4A$ ./Task2
Enter a number: 3
x: 2, y: 2
vboxuser@Debian:~/Desktop/OpSis/Lab4A$ ./Task2
Enter a number: 5
x: 4, y: 4
vboxuser@Debian:~/Desktop/OpSis/Lab4A$ ./Task2
Enter a number: 2
x: 2, y: 2
vboxuser@Debian:~/Desktop/OpSis/Lab4A$ gcc Task2.c -o Task2
vboxuser@Debian:~/Desktop/OpSis/Lab4A$ ./Task2
Bir sayı girin: 3
x: 4, y: 4
vboxuser@Debian:~/Desktop/OpSis/Lab4A$ ./Task2
Bir sayı girin: 4
x: 5, y: 5
vboxuser@Debian:~/Desktop/OpSis/Lab4A$ ./Task2
Bir sayı girin: 6
x: 7, y: 7
vboxuser@Debian:~/Desktop/OpSis/Lab4A$
```

(In first run I used given code after using gcc the output is my own code)

The problem with the initial code is that multiple threads access shared variables `x` and `y` at the same time, which creates a race condition and can lead to incorrect results. To fix this, the section where `y = x + 1;` and `x = y;` are updated should be protected with a mutex lock. In the improved code, `pthread_mutex_lock` and `pthread_mutex_unlock` are used to make sure only one thread can access this section at a time. This way, race conditions are prevented, and `x` and `y` are updated safely, leading to consistent and expected results even with multiple threads.

Improved Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

int x = 1, y = 0;

pthread_mutex_t lock; // Mutex tanımlaması

void *work(void *ptr) {

    pthread_mutex_lock(&lock); // Kritik bölgeye giriş

    y = x + 1;

    x = y;

    pthread_mutex_unlock(&lock); // Kritik bölgeden çıkış

    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {

    int number, i = 0;

    pthread_t *threads;

    printf("Bir sayı girin: ");

    scanf("%d", &number);

    threads = (pthread_t *)malloc(number * sizeof(pthread_t));

    pthread_mutex_init(&lock, NULL); // Mutex başlatma

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

        pthread_create(&threads[i], NULL, work, NULL);

    }

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

        pthread_join(threads[i], NULL);

    }

    printf("x: %d, y: %d\n", x, y);

    pthread_mutex_destroy(&lock); // Mutex yok etme

    free(threads); // Belleği serbest bırakma

    pthread_exit(NULL);
}
```