```
var n = 1000000009;
var isPrime = true;
// I can create a Task with operations to be executed in parallel,
// similarly to what I do with Thread: I pass an Action to the constructor.
var task1 = new Task(() => {
    for (var div = 2; div < n; div++) {
        if (n % div == 0) {
            isPrime = false;
            break;
        }
    }
});

// I can create a Task<T>, where T is the type of the result
// returned by the task.
var task2 = new Task<bool>(() => {
    for (var div = 2; div < n; div++) {
        if (n % div == 0) {
            return false;
        }
    }
    return true;
});

// I can start a Task as I start a Thread:
task2.Start();
// (a non-started task doesn't execute the Action
// an is awaited indefinitely)

// I can await a Task as I join a Thread:
task2.Wait();

// If it's a Task<T>, I can read the task result in the Result property.
// If the task was not awaited, it is awaited now:
isPrime = task2.Result;

// Both Wait() and Result wait for the result in a synchronous way,
// blocking the calling thread.

// I can use a task in an asynchronous way, using async/await:

var task3 = new Task<bool>(() => {
    for (var div = 2; div < n; div++) {
        if (n % div == 0) {
            return false;
        }
    }
    return true;
});

task3.Start();
```

```
52
53  // If I want to create a Task that is already started and ready to run,
54  // I can use the factory method:
55
56  task3 = Task.Run(() => {
57      for (var div = 2; div < n; div++) {
58          if (n % div == 0) {
59              return false;
60          }
61      }
62      return true;
63  });
64
65  // I can await a Task in an asynchronous way
66  // using the keyword 'await':
67  isPrime = await task3;
68
69  // The execution awaits for the task completion before
70  // continuing the execution of the method, but meanwhile the calling
      Thread
71  // is free to operate elsewhere.
72
73  Console.WriteLine($"{n} is {(isPrime ? "" : "not ")}prime");
74
75  var task4 = Task.Run(() => {
76      for (var div = 2; div < n; div++) {
77          if (n % div == 0) {
78              return false;
79          }
80      }
81      return true;
82  });
83  var task5 = Task.Run(() => {
84      for (var div = 2; div < n; div++) {
85          if (n % div == 0) {
86              return false;
87          }
88      }
89      return true;
90  });
91
92  // The thread that continues the execution after the 'await' could be
      another one.
93  // More likely, here the Console will print different ids:
94  Console.WriteLine($"Current Thread:
      {Environment.CurrentManagedThreadId}");
95  await task4;
96  Console.WriteLine($"Current Thread:
      {Environment.CurrentManagedThreadId}");
97  await task5;
98  Console.WriteLine($"Current Thread:
      {Environment.CurrentManagedThreadId}");
99  ;
```

```
100
101 // I can execute an asynchronous sleep, through:
102 await Task.Delay(1000);
103
```

```
1  namespace S06_Advanced_P08_Tasks3;
2
3  /*
4  Exercise: implement a console application that reads a number from the
     console,
5  and then calculates if the number is prime.
6  The calculation must be stopped if it takes too much time.
7  To implement the exercise, use Task.WhenAny().
8  The first task is the one with the calculation.
9  The second task is just a Delay of N milliseconds.
10 When WhenAny() returns (id est, when the fastest task has the result),
11 the system must stop the other Task.
12 You can use CancellationTokenSource and CancellationToken.
13 */
14
15 class Program
16 {
17     static async Task Main() {
18         Console.WriteLine("*** App to calculate if a number is prime,
             with timeout! ***");
19         var number = ReadNumberFromConsole();
20         var isPrime = false;
21         // When I want to coordinate the work between threads or tasks,
22         // I create a single CancellationTokenSource:
23         var cts = new CancellationTokenSource();
24         // Then I pass to each task its CancellationToken.
25         // .NET async methods usually have an overload that accepts a
             CancellationToken;
26         // they stop as soon as a cancellation is requested on their
             source:
27         var calculationTask = Task.Run(() => isPrime = IsPrime(number),
             cts.Token);
28         var timeoutTask = Task.Delay(500, cts.Token);
29         // Now I use WhenAny() to start them all concurrently:
30         var resultTask = await Task.WhenAny(new[] { timeoutTask,
             calculationTask });
31         // resultTask is the Task that finished first.
32         // At this point I call Cancel() on the source so that all the
             other tasks are stopped:
33         cts.Cancel();
34         // Now I check which task won:
35         if (resultTask == calculationTask) {
36             Console.WriteLine($"{number} is {(isPrime ? "" : "not ")}
                 prime");
37         } else {
38             Console.WriteLine("TIMEOUT! The computation required too
                 much time.");
39         }
40         // I could also check the ids of the Task (every Task has a
             unique Id).
41     }
42
43     private static long ReadNumberFromConsole() {
```

```
44          while (true) {
45              Console.Write("Enter a number: ");
46              var value = Console.ReadLine();
47              if (long.TryParse(value, out var number)) {
48                  return number;
49              } else {
50                  Console.WriteLine("Invalid number! Retry!");
51              }
52          }
53      }
54
55      static bool IsPrime(long n) {
56          for (long div = 2; div < n; div++) {
57              if (n % div == 0) {
58                  return false;
59              }
60          }
61          return true;
62      }
63  }
64
```

```
1  using System.Diagnostics;
2
3  var sw = Stopwatch.StartNew();
4
5  var number = 1000000009;
6
7  bool hasDivisorsMainThread = false;
8  bool hasDivisorsSecondaryThread = false;
9
10 // When a process is executed, there is a main Thread,
11 // i.e. a first stack of operations.
12 // Every thread has an unique Id:
13 Console.WriteLine($"Current Thread Id:
     {Environment.CurrentManagedThreadId}");
14
15 // I create a Thread giving an Action as input.
16 var secondaryThread = new Thread(() => {
17     Console.WriteLine($"Current Thread Id:
         {Environment.CurrentManagedThreadId}");
18     hasDivisorsSecondaryThread = HasDivisorsInRange(number, number/4+1,
         number/2);
19 });
20 // When I start the Thread, that Action is executed:
21 secondaryThread.Start();
22
23 // The Start() method exits immediately, so that the execution on the
     main Thread
24 // can continue immediately:
25 hasDivisorsMainThread = HasDivisorsInRange(number, 2, number/4);
26
27 // Now, to calculate the final result, I need to wait the secondary
     thread:
28 secondaryThread.Join();
29 // The Join() method does not exit until the secondary thread has
     finished.
30
31 // Now that I waited for both partial results, I can compose the final
     result:
32 var isPrime = !hasDivisorsMainThread && !hasDivisorsSecondaryThread;
33
34 sw.Stop();
35 Console.WriteLine($"Time taken: {sw.ElapsedMilliseconds} ms");
36 Console.WriteLine($"{number} is {(isPrime ? "" : "not ")}prime");
37
38 // I can put a Thread "to sleep": the Thread stops to work for N
     milliseconds:
39 Thread.Sleep(1000);
40
41 static bool HasDivisorsInRange(int n, int start, int end) {
42     for (var div = start; div < end; div++) {
43         if (n % div == 0) {
44             return true;
45         }
```

```
46      }
47      return false;
48  }
49
```

```
1  using System.Diagnostics;
2
3  class Program
4  {
5      private static readonly int LIMIT = (int)Math.Pow(2, 16);
6
7      private static readonly object _lock = new();
8
9      static void Main() {
10         CalculateOptimalThreads();
11         CalculateWithParallel();
12     }
13
14     private static void CalculateOptimalThreads() {
15         // I want to find the optimal number of Threads to parallelize
16         // a sequence of operations.
17         // I discover that the optimal number is a multiple of the
             number
18         // of cores of my CPU.
19         // Every CPU core nowadays can execute more flows in parallel.
20         for (var exp = 0; exp < 8; exp++) {
21             var threadCount = (int)Math.Pow(2, exp);
22             CalculateAndMeasurePrimes(threadCount);
23         }
24     }
25
26     private static void CalculateWithParallel() {
27         // Every computer has an optimal number of parallel threads.
28         // I can delegate to the framework to calculate that optimal
             number:
29         var sw = Stopwatch.StartNew();
30         int result = 0;
31         Parallel.ForEach(
32             Enumerable.Range(2, LIMIT - 1),
33             n => {
34                 if (IsPrime(n)) {
35                     // this is wrong: result is shared between the
                       threads,
36                     // so a race condition might happen:
37                     // more threads access the same variable at the same
                       time,
38                     // reading the same value, so the double ++ results
                       in a single increment.
39                     // As a consequence, the final count is less than
                       the correct one,
40                     // and is different every time (RACE CONDITION):
41                     // result++;
42
43                     // To solve this problem, we can use a LOCK:
44                     lock (_lock) {
45                         result++;
46                     }
47                     // Only one Thread at a time can access the locked
```

```
                    section.
48                    // If a second Thread attempts to enter it, the
                    Thread
49                    // is blocked until the previous Thread has
                    finished.
50                }
51            }
52        );
53        sw.Stop();
54        Console.WriteLine($"Time taken: {sw.ElapsedMilliseconds} ms");
55        Console.WriteLine($"Primes up to {LIMIT} = {result}");
56    }
57
58    private static void CalculateAndMeasurePrimes(int threadCount) {
59        Console.WriteLine($"THREAD COUNT: {threadCount}");
60        var sw = Stopwatch.StartNew();
61        var subrangeSize = LIMIT / threadCount;
62        var results = new int[threadCount];
63        var threads = Enumerable
64            .Range(1, threadCount)
65            .Select(i => new Thread(() => {
66                var start = Math.Max(subrangeSize * (i - 1) + 1, 2);
67                var end = subrangeSize * (i - 1) + subrangeSize;
68                for (var n = start; n < end; n++) {
69                    if (IsPrime(n)) {
70                        results[i - 1]++;
71                    }
72                }
73            }))
74            .ToList();
75        foreach (var t in threads) {
76            t.Start();
77        }
78        foreach (var t in threads) {
79            t.Join();
80        }
81        var result = results.Sum();
82        sw.Stop();
83        Console.WriteLine($"Time taken: {sw.ElapsedMilliseconds} ms");
84        Console.WriteLine($"Primes up to {LIMIT} = {result}");
85    }
86
87    static bool IsPrime(int n) {
88        for (var div = 2; div < n; div++) {
89            if (n % div == 0) {
90                return false;
91            }
92        }
93        return true;
94    }
95 }
96
```