11d. Thread-Safe Operations

Martin Alfaro
PhD in Economics

INTRODUCTION

Multithreading allows running multiple threads simultaneously in a single process, enabling operations to be executed in parallel within the same computer. Unlike other forms of parallelization such as multiprocessing, multithreading is distinguished by **the sharing of a common memory space among all tasks**.

This shared memory environment introduces several complexities, determining that **running code in parallel may create side effects if handled improperly**. Basically, the issue arises when multiple threads access and modify shared data, potentially causing unintended consequences in other threads. These potential issues have led to the concept of **thread-safe operations**. They're characterized by the possibility of being executed in parallel without causing any issues (e.g., data corruption, inconsistencies, or crashes).

The section starts by identifying features that make operations unsafe. They'll reveal that common operations such as reductions aren't thread safe, giving rise to incorrect results if multithreading is applied naively. We'll also explore the concept of embarrassingly parallel problems, which are a prime example of thread-safe operations. As the name suggests, these problems can be parallelized directly, without requiring significant program adaptations.

UNSAFE OPERATIONS

We start by presenting some operations that aren't thread-safe. The examples highlight the need for caution when tasks exhibit some degree of dependency, either in terms of operations or shared resources.

WRITING ON A SHARED VARIABLE

The following example highlights the potential pitfalls of writing to a shared variable in a concurrent environment. The scenario considered is such that a scalar variable `output` is initialized to zero. Then, this value is updated within a for-loop that iterates twice, with `output` set to `i` in the *i*-th iteration. The script is as follows.

```
function foo()
    output = 0

    for i in 1:2
        sleep(1/i)
        output = i
    end

    return output
end
```

```
julia> foo()
2
```

```
function foo()
    output = 0

    @threads for i in 1:2
        sleep(1/i)
        output = i
    end

    return output
end
```

```
julia> foo()
1
```

To illustrate the challenges of concurrent execution, we've deliberately introduced a decreasing delay before updating `output`. This delay is implemented using `sleep(1/i)`, causing the first iteration to pause for 1 second and the second iteration to pause for half a second. Although this delay is artificially introduced through `sleep`, it represents the potential delays caused by intermediate computations, which could prevent an immediate update of `output`.

The delay is inconsequential for a sequential procedure, with `output` taking on the values 0, 1, and 2 as the program progresses. However, when executed concurrently, the first iteration completes only after the second iteration has finished. As a result, the sequence of values for `output` is 0, 2, and 1.

While the problem may seem apparent, it can manifest in more complex and subtle ways. In fact, the issue can be exacerbated when each iteration additionally involves reading a shared variable. Next, we consider a scenario like this.

READING AND WRITING A SHARED VARIABLE

Reading and writing shared data doesn't necessarily cause problems. For instance, we'll demonstrate that a parallel for-loop can safely mutate a vector, even though multiple threads are simultaneously modifying a shared object (the vector). However, in scenarios where **reading and writing shared data is sensitive to the specific order of thread execution**, it can give rise to a **data race** (also known as **race condition**). The name reflects that the final output will change in each execution, depending on which thread finishes and modifies the data last.

To illustrate the issue, let's keep using an example similar to the outlined above. We modify the example by introducing the variable `temp`, whose value is updated in each iteration. Moreover, this is a variable shared across threads, and is used to mutate the *i*-th entry of a vector `output`. By introducing a delay before writing each entry of `output`, the example shows that all threads end up using the last value of `temp`, which is 2.

```
function foo()
    out  = zeros(Int, 2)
    temp = 0

    for i in 1:2
        temp   = i; sleep(i)
        out[i] = temp
    end

    return out
end
```

```
julia> foo()
2-element Vector{Int64}:
 1
 2
```

```
function foo()
    out  = zeros(Int, 2)
    temp = 0

    @threads for i in 1:2
        temp   = i; sleep(i)
        out[i] = temp
    end

    return out
end
```

```
julia> foo()
2-element Vector{Int64}:
 1
 1
```

```
function foo()
    out  = zeros(Int, 2)


    @threads for i in 1:2
        temp   = i; sleep(i)
        out[i] = temp
    end

    return out
end
```

```
julia> foo()
2-element Vector{Int64}:
 1
 2
```

As the last tab shows, the issue can be easily circumvented in this case. The solution simply requires defining `temp` as a local variable, which is achieved by avoiding its initialization out of the for-loop. By doing so, each thread will refer to its own local copy of `temp`.

Beyond this specific solution, the example aims to highlight the subtleties of parallelizing operations. To further illustrate it, we next examine a more common scenario where data races occur: reductions.

RACE CONDITIONS WITH REDUCTIONS

To illustrate the issue with reductions, let's consider the sum operation. The gist of the problem lies in that the variable accumulating the sum is accessed and modified by all threads in each iteration.

```
x = rand(1_000_000)

function foo(x)
    output = 0.

    for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
julia> foo(x)
500658.01158503356
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.

    @threads for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
julia> foo(x)
21534.22602627773
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.

    @threads for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
julia> foo(x)
21342.557817155746
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.

    @threads for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
julia> foo(x)
21664.133622716112
```

The key insight from this example isn't that reductions are incompatible with multithreading. Rather, that the strategy to apply multithreading needs to be adapted accordingly.

In the following, we'll consider the simplest case to apply multithreading, which is referred to as embarrassingly parallel. Its distinctive feature is that multithreading can be applied without any transformation of the data. After that, we'll show scenarios that can handle dependent operations like reductions.

EMBARRASSINGLY PARALLEL PROBLEMS

The simplest scenario in which multithreading can be applied is known as an **embarrassingly parallel problem**. The term highlights the ease with which code can be divided for parallel execution. It comprises programs consisting of multiple independent and identical subtasks, not requiring interaction with one another to produce the final output. This independence allows for seamless parallelization, providing complete flexibility in the order of task execution.

In for-loops, one straightforward way to parallelize these problems is given by the macro `@threads`. This is a form of thread-based parallelism, where the distribution of work is based on the number of threads available. Specifically, `@threads` attempts to evenly distribute the iterations, in an effort to balance the workload. The approach contrasts with `@spawn`, which is a task-based parallelism where iterations are divided according how the user has manually defined tasks. Unlike `@spawn`, which requires a manual synchronization of the tasks, `@threads` automatically schedules the tasks and waits for their completion before proceeding with any further operations. This is demonstrated below.

```
x_small  = rand(    1_000)
x_medium = rand(  100_000)
x_big    = rand(1_000_000)

function foo(x)
    output = similar(x)

    for i in eachindex(x)
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> @btime foo($x_small)
  3.043 µs (1 allocations: 7.938 KiB)
julia> @btime foo($x_medium)
  315.751 µs (2 allocations: 781.297 KiB)
julia> @btime foo($x_big)
  3.326 ms (2 allocations: 7.629 MiB)
```

```
x_small  = rand(    1_000)
x_medium = rand(  100_000)
x_big    = rand(1_000_000)

function foo(x)
    output = similar(x)

    @threads for i in eachindex(x)
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> @btime foo($x_small)
  10.139 μs (122 allocations: 20.547 KiB)
julia> @btime foo($x_medium)
  42.044 μs (123 allocations: 793.906 KiB)
julia> @btime foo($x_big)
  340.589 μs (123 allocations: 7.642 MiB)
```

In the next section, we provide a thorough analysis of the differences between `@threads` and `@spawn`.