11d. Thread-Safe Operations

Martin Alfaro
PhD in Economics

INTRODUCTION

Multithreading allows multiple threads to run simultaneously in a single process, enabling parallel execution of operations within the same computer. Unlike other forms of parallelization (e.g., multiprocessing), multithreading is distinguished by **the sharing of a common memory space among all tasks**.

This shared memory environment introduces several complexities, implying that **running code in parallel may create side effects if handled improperly**. Essentially, the issue arises when multiple threads access and modify shared data simultaneously, potentially causing unintended consequences in other threads.

These potential problems have led to the distinguish between **thread-safe and thread-unsafe operations**. Characterizing operations in one or the other way depends on their ability to be executed in parallel without causing any issues (e.g., data corruption, inconsistencies, or crashes).

The section will be centered on identifying some key features that render operations unsafe. They'll reveal that common operations like reductions aren't thread safe, leading to incorrect results if multithreading is applied naively. We'll conclude the section by exploring 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

Unsafe operations in multithreaded environments are those that can lead to incorrect behavior, data corruption, or program crashes when executed concurrently. The following examples we present will highlight their potential emergence when tasks exhibit some degree of dependency, either in terms of operations or shared resources.

WRITING ON A SHARED VARIABLE

One of the most basic examples of an unsafe operation is writing to a shared variable. To illustrate this, consider a scenario where a scalar variable `output` is initialized to zero. This value is then 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 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 via `sleep`, it represents the potential gap in time caused by intermediate computations, which could preclude an immediate update of `output`.

The delay is inconsequential for a sequential procedure, where `output` takes on the values 0, 1, and 2 as the program progresses. However, when executed concurrently, the first iteration completes 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 in this simple example, it can manifest in more complex and subtle ways in real-world applications. The core issue is that the order of execution is not guaranteed in a multithreaded environment. Thus, when multiple threads modify the same shared variable, the final value depends on which thread executes last.

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, a parallel for-loop could safely mutate a vector, even though multiple threads are simultaneously modifying a shared object (the vector). This is the case as long as each thread operates on distinct elements of the vector.

However, when **reading and writing shared data is sensitive to the specific order of thread execution**, we can end up with a **data race** (also known as **race condition**). The name reflects that the final output will change every time we execute the operation, depending on which thread finishes and modifies the data last.

To illustrate the issue, let's modify our previous example by introducing a variable `temp`, whose value is updated in each iteration. This variable will additionally be shared across threads and 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
```

The issue can be easily circumvented in this case by defining `temp` as a variable local to the for-loop. This can be achieved by avoiding its initialization outside the for-loop. By doing so, each thread would refer to its own local copy of `temp`, eliminating the data race.

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

Reductions are a prime example when it comes to thread-unsafe operations. To demonstrate this, 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. The following sections will present these strategies.

EMBARRASSINGLY PARALLEL PROBLEMS

We close this section by considering the opposite end of the spectrum: problems that naturally lend themselves to parallel execution without these complications. They're referred to as **embarrassingly parallel problems**.

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`.