11d. Thread-Safe Operations

Martin Alfaro
PhD in Economics

INTRODUCTION

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

This shared-memory environment introduces additional complexity, since **parallel execution can produce unintended side effects if not managed carefully**. The core issue arises when multiple threads access and modify shared data concurrently, which can cause unintended consequences in other threads.

These problems motivate the distinction between thread-safe and thread-unsafe operations. An operation is considered **thread-safe** if it can be executed in parallel without causing inconsistencies, crashes, or corrupted results. Conversely, **thread-unsafe** operations require explicit synchronization or restructuring to avoid errors.

This section will focus on identifying key features that render certain operations unsafe. In particular, we'll see that common operations like reductions aren't thread safe, leading to incorrect results if multithreading is applied naively. We'll conclude 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 modifications to program structure.

UNSAFE OPERATIONS

In multithreaded environments, unsafe operations correspond to those that can lead to incorrect behavior, data corruption, or program crashes when executed concurrently. Such issues typically arise when tasks exhibit some degree of dependency, either in terms of operations or shared resources.

WRITING ON A SHARED VARIABLE

One of the simplest examples of a thread-unsafe operation is writing to a shared variable. To illustrate, 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 corresponding script is shown below.

```
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 was implemented with `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 isn't 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 incorrect results. For instance, a parallel for-loop could safely mutate a vector: even if multiple threads are simultaneously modifying the same shared object, each thread would be operating on distinct elements of the vector. Thus, no two threads would interfere with one another, making the updates remain independent.

Problems arise, however, **when the correctness of reading and writing shared data depends on the order in which threads execute**. This situation is known as a **data race** (also known as **race condition**). he term reflects the fact that the final output may change from one run to the next, depending on which thread finishes and updates 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 be shared across threads and used to mutate the *i*-th entry of the vector `output`. By introducing a delay before writing each entry of `output`, a race condition is introduced, where all threads end up using the last value of `temp` (in this case, 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
```

In this specific scenario, the issue can be easily circumvented by defining `temp` as a variable local to the for-loop, rather than initializing it outside. This ensures that each thread works with its own private copy of `temp`, thereby eliminating the data race.

Beyond the solution proposed, the example highlights the subtleties of parallelizing operations. Even seemingly simple patterns can introduce hidden dependencies that lead to unsafe behavior when executed concurrently. To make this clearer, we now turn to a more common scenario where data races occur: reductions.

RACE CONDITIONS WITH REDUCTIONS

Reductions are a prime example of thread-unsafe operations. To illustrate, consider summing a collection in parallel. The problem arises because the variable accumulating the sum is shared across all threads. During each iteration, every thread attempts to read its current value, add its contribution, and write the result back. When several threads do this simultaneously, their updates can interleave unpredictably, causing some partial additions to be overwritten rather than combined. As a result, the final sum is nondeterministic and often incorrect, varying from run to run.

```
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. While the upcoming sections will present these strategies, we conclude this section by turning to the opposite end of the spectrum: problems that naturally lend themselves to parallel execution

EMBARRASSINGLY PARALLEL PROBLEMS

Programs that can be paralleled seamlessly are referred to as **embarrassingly parallel problems**. They can be broken down into many completely independent tasks, each of which can be executed in parallel without any need for communication, synchronization, or shared state between them. This independence provides complete flexibility in the order of task execution.

In the context of for-loops, one of the simplest ways to parallelize these problems is with 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 balance the workload by dividing the iterations as evenly as possible.

The approach contrasts with `@spawn`, which implements task-based parallelism. With `@spawn`, the programmer explicitly defines tasks and synchronization must be handled manually. By contrast, `@threads` automatically schedules the tasks and waits for their completion before execution proceeds. The following example demonstrates this behavior.

```
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'll present a detailed comparison of `@threads` and `@spawn`.