11d. Thread–Safe Operations

[Martin Alfaro](#)
PhD in Economics

INTRODUCTION

Multithreading allows multiple threads to run simultaneously within a single process, enabling parallel execution of operations on the same machine. Unlike other forms of parallelization such as multiprocessing, where each process has its own memory space, **threads in multithreading share a common memory space**.

This shared-memory environment makes parallelization easy, but it also introduces some complexity while writing code. Since all threads can access and modify the same data simultaneously, **parallel execution can lead to unintended side effects if not managed carefully**. In particular, the interaction between threads may cause corrupted results or even program crashes.

To reason about these issues, it's useful to distinguish 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 algorithmic restructuring to prevent corruption.

The current section will identify key features that render operations unsafe under multithreading. In particular, we'll see that common operations like reductions aren't thread-safe, leading to incorrect results when multithreading is applied naively. We'll also explore the concept of embarrassingly parallel programs, which are a prime example of thread-safe operations. These programs can be divided into independent units of work that require no communication or synchronization. Consequently, they can be parallelized directly, without the need to restructure the program.

THREAD-UNSAFE OPERATIONS

In multithreaded environments, unsafe operations are 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 through a for-loop that iterates twice, with `output` set to `i` in the *i*-th iteration.

To starkly lay bare the challenges of concurrent execution, we deliberately introduce a decreasing delay before updating `output`. This is implemented with `sleep(1/i)`, causing the first iteration to pause for 1 second and the second iteration for half a second. Although this delay is artificially introduced via `sleep`, it represents the potential time gap caused by intermediate computations, which could preclude an immediate update of `output`.

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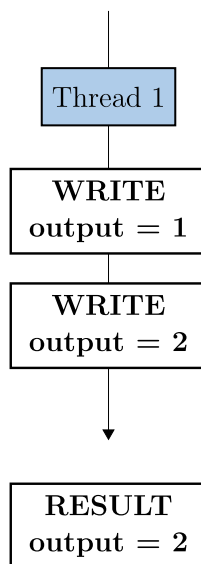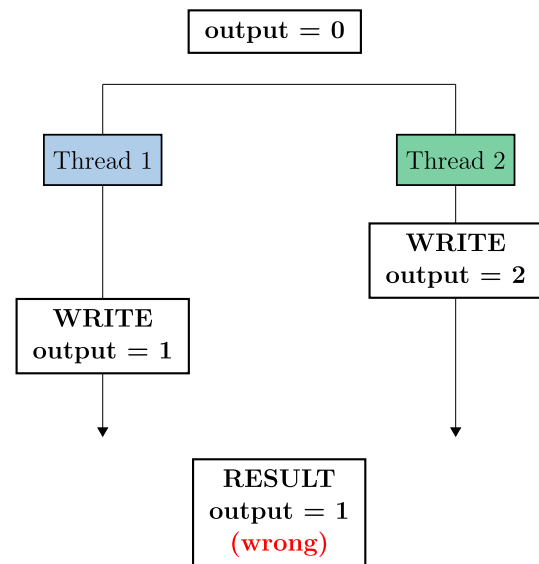```
julia> foo()
1
```

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, it determines that the first iteration completes after the second iteration has finished. As a result, the sequence of values for `output` is 0, 2, and 1.

## SEQUENTIAL                                        PARALLEL



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 such a scenario.

## READING AND WRITING A SHARED VARIABLE

Reading and writing shared data doesn't necessarily yield 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 **race condition**. The 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 demonstrate 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).

```julia
function foo()
    output = Vector{Int}(undef,2)
    temp   = 0

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

    return output
end
```

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

```julia
function foo()
    output = Vector{Int}(undef,2)
    temp   = 0

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

    return output
end
```

```julia
julia> foo()
2-element Vector{Int64}:
 2
 2
```
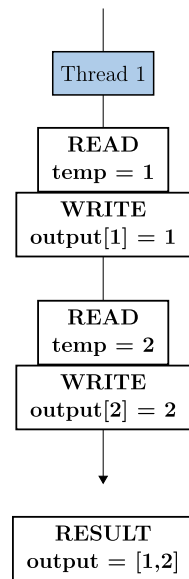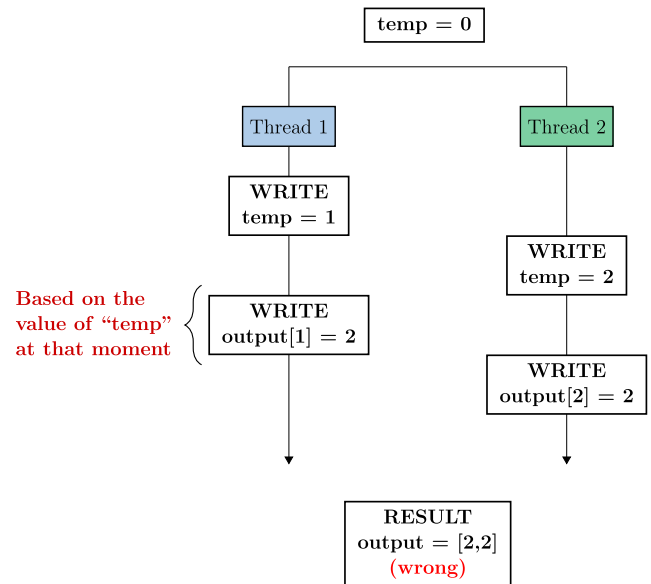
```julia
function foo()
    output = Vector{Int}(undef,2)


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

    return output
end
```

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

SEQUENTIAL   **PARALLEL**

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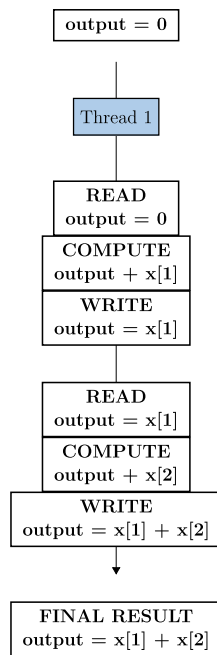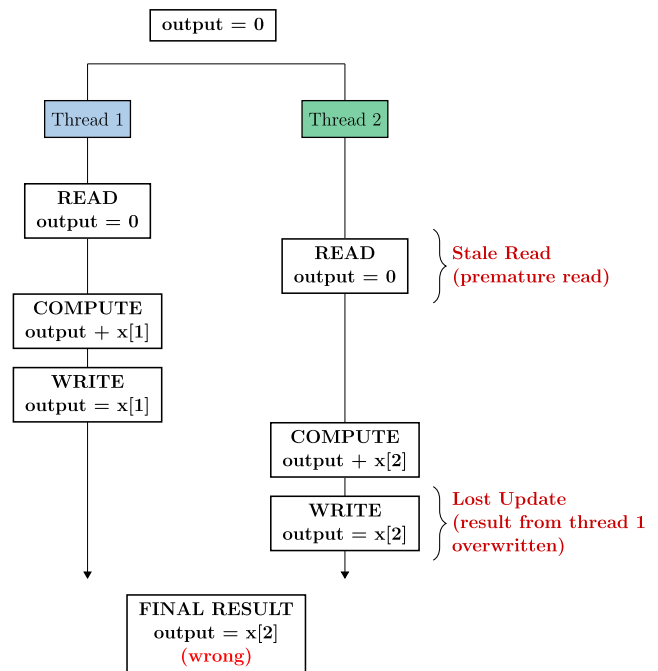
Beyond the solution proposed, the example highlights the subtleties of parallelizing operations. Even seemingly simple patterns can introduce hidden dependencies that lead to unsafe behavior when executed concurrently. To make this clearer, we now turn to a more common scenario where data races occur: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

RACE CONDITIONS WITH REDUCTIONS

Reductions are a prime example of thread-unsafe operations. To illustrate why this is so, consider summing a collection in parallel. At first glance, this seems straightforward: each thread computes a partial contribution and adds it to a shared accumulator. However, the variable holding each partial 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 multiple threads perform these steps concurrently, their actions can overlap in unpredictable ways. One thread's update may overwrite another's, meaning that some contributions are lost rather than combined. As a result, the final sum is nondeterministic and often incorrect, varying from run to run.

Below, we illustrate this issue when we sum the first two elements of a vector `x`.

SEQUENTIAL

output = 0

Thread 1

READ
output = 0

COMPUTE
output + x[1]

WRITE
output = x[1]

READ
output = x[1]

COMPUTE
output + x[2]

WRITE
output = x[1] + x[2]

FINAL RESULT
output = x[1] + x[2]

PARALLEL

output = 0

Thread 1   Thread 2

READ
output = 0

READ   Stale Read
output = 0   (premature read)

COMPUTE
output + x[1]

WRITE
output = x[1]

COMPUTE
output + x[2]

WRITE   Lost Update
output = x[2]   (result from thread 1
                            overwritten)

FINAL RESULT
output = x[2]
(wrong)

When performing reductions in parallel without addressing the underlying race condition, the outcome becomes unpredictable. More specifically, the final sum tends to be lower than the correct value, because some updates are lost when threads overwrite one another's contributions.

```julia
x = rand(1_000_000)

function foo(x)
    output = 0.0

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

    return output
end
```

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

```julia
x = rand(1_000_000)

function foo(x)
    output = 0.0

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

    return output
end
```

```
julia> foo(x)
21436.48668413443
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

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

    return output
end
```

```
julia> foo(x)
21590.997961948713
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

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

    return output
end
```

```
julia> foo(x)
21461.273851717895
```

The key insight from this example isn't that reductions are incompatible with multithreading. Rather, it's that the strategy to apply multithreading needs to be adapted accordingly. While the upcoming sections will present these strategies, we conclude this one by turning to the opposite end of the spectrum: problems that naturally lend themselves to parallel execution.

EMBARRASSINGLY PARALLEL PROGRAMS

The simplest thread-safe programs are those in which tasks have no dependencies at all, which are known as **embarrassingly parallel problems**. Such problems be decomposed into many independent tasks, each of which can be executed in parallel without communication, synchronization, or shared state. This characteristic grants full flexibility in determining the order of task execution.

In the context of for-loops, a simple way to parallelize embarrassingly parallel problems is through the macro `@threads`. This is a form of thread-based parallelism, where the distribution of work is based on the number of threads available. In particular, `@threads` attempts to balance the workload by dividing the iterations as evenly as possible. Unlike `@spawn`, `@threads` automatically schedules the tasks and waits for their completion before execution proceeds. In the next section, we'll present a detailed comparison between `@threads` and `@spawn`. For now, the following example demonstrates the simplicity of `@threads`.

```
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)
  3.319 µs (3 allocations: 7.883 KiB)

julia> @btime foo($x_medium)
  332.609 µs (3 allocations: 781.320 KiB)

julia> @btime foo($x_big)
  3.396 ms (3 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)
  9.092 µs (125 allocations: 20.508 KiB)

julia> @btime foo($x_medium)
  42.710 µs (125 allocations: 793.945 KiB)

julia> @btime foo($x_big)
  336.284 µs (125 allocations: 7.642 MiB)
```