11b. Introduction to Multithreading

Martin Alfaro
PhD in Economics

INTRODUCTION

A proper implementation of multithreading demands some basic understanding of the inner workings of computers. In particular, it's essential to identify how programming languages handle dependency between operations. Grasping these concepts is especially relevant for multithreading, since the technique exposes the user to the risk of writing unsafe code: a flawed implementation may yield incorrect results.

This section will only present preliminary concepts, setting the stage for subsequent sections. The focus will be on explanations rather than actual implementations of multithreading. In fact, most of the macros and functions introduced here won't be utilized again.

NATURE OF COMPUTATIONS

Operations can be broadly classified into two categories: dependent and independent. Outcomes in **dependent operations** affect or are affected by other operations, while **independent operations** turn the order of execution irrelevant for the outcome obtained. As an illustration, the following code gives rise to dependent or independent operations, depending on which numbers are summed by operation B.

```
job_A()  = 1 + 1
job_B(A) = 2 + A

function foo()
    A = job_A()
    B = job_B(A)

    return A,B
end
```

```
job_A() = 1 + 1
job_B() = 2 + 2

function foo()
    A = job_A()
    B = job_B()

    return A,B
end
```

Likewise, regardless of their dependence status, operations can be computed either sequentially or concurrently. A **sequential** procedure involves executing operations one after the other, thus ensuring that each operation completes before the next one begins. Conversely, **concurrency** allows multiple operations to be processed simultaneously, opening up opportunity for parallel execution.

Like most programming languages, **Julia defaults to sequential execution**. This is a deliberate choice that prioritizes result correctness, grounded in that **concurrent execution with dependent operations can yield incorrect results if mishandled**. Basically, the issue arises because concurrency can deal with dependencies in multiple manners, potentially involving timing inconsistencies for reading and writing data. A sequential approach precludes this possibility, as it guarantees a predictable order of execution and therefore timing.

Despite its advantages regarding safety, a sequential approach can be quite inefficient for independent tasks: by restricting computations to one at a time, the computational resources may go underutilized. In contrast, **a simultaneous approach allows for operations to be calculated in parallel, thereby fully utilizing all our available computational resources**. This can lead to significant reductions in computational time.

Since programming languages typically default to sequential execution, some nuances of concurrency can be challenging to understand (e.g., concurrency doesn't necessarily imply simultaneity). This poses a significant issue if it leads to an inappropriate handling of concurrency. To address this aspect, we next revisit these approaches in light of the fundamental concepts of tasks and threads.

TASKS AND THREADS

Internally, Julia defines instructions to process through the concept of **tasks**. In turn, each of these tasks must be assigned to **a computer thread** for its computation. Since there's a one-to-one relationship between threads and tasks, **the number of threads available on your computer determines the number of tasks that can be computed simultaneously**.

Importantly, each session in Julia begins with a predefined pool of threads, and Julia defaults to a single thread regardless of your computer's hardware. We'll start with this case, as it provides an intuitive starting point for understanding concurrency.

To help our intuition, consider two workers A and B, who we'll think of as employees working for a company. B's job consists of performing the same operation continuously during a certain amount of time. In the code, this is represented by summing `1+1` repeatedly during one second. Instead, A's job consists of receiving some delivery, which will arrive after a certain period of time. In the code, this is represented by performing no computations for two seconds, through the function `sleep(2)`.

```
function job_A(time_working)
    sleep(time_working)        # do nothing (waiting for some delivery in the example)

    println("A completed his task")
end
```

```
function job_B(time_working)
    start_time = time()

    while time() - start_time < time_working
        1 + 1                  # compute `1+1` repeatedly during `time_working` seconds
    end

    println("B completed his task")
end
```

Due to the lazy nature of function definitions, these code snippets simply describe a set of operations without computing anything. It's only when we add lines like `job_A(2)` and `job_B(1)` that the operations are sent for computation. To lay bare the internal steps followed by Julia, let's adopt a more low-level approach and define `job_A(2)` and `job_B(1)` as tasks. As shown below, tasks aren't mere abstractions to organize our discussion, but are actually part of Julia's codebase.

```
A = @task job_A(2)      # A's task takes 2 seconds
B = @task job_B(1)      # B's task takes 1 second
```

Once tasks are defined, the first step for their computation is to **schedule** them. This means that the task is added to the queue of operations that the computer's processor will execute. Essentially, scheduling instructs the machine to compute a task as soon as a thread becomes available.

Importantly, multiple tasks can be *processed* concurrently, without implying that they'll be *computed* simultaneously. Indeed, this is the case in a single-thread session. The distinction can be understood through an analogy with juggling: a juggler manages multiple balls at the same time, but only holds one ball at any given moment. Similarly, multiple tasks can be processed simultaneously, even when only one is actively executing on the CPU.

Although true parallelism isn't feasible in single-threaded sessions, concurrency can still offer some benefits. This is due to the possibility of **task switching**, which is enabled by a **task yielding** mechanism. When a task becomes idle, it can voluntarily relinquish control of the thread, allowing other tasks to utilize the thread's time. By fostering a cooperative approach, concurrency ensures plenty of computers' resource utilization at any given time.

In the following, we describe this mechanism in more detail.

SEQUENTIAL AND CONCURRENT COMPUTATIONS

While code is executed sequentially by default, *tasks* are designed to compute concurrently. As a result, adopting a sequential approach requires instructing Julia to execute tasks one at a time. This is achieved by introducing a wait instruction immediately after scheduling a task, thus ensuring that the task completes its calculation before proceeding.

The code snippet below demonstrates this mechanism by introducing the functions `schedule` and `wait`.

```
A = job_A(2)            # A's task takes 2 seconds
B = job_B(1)            # B's task takes 1 second
```

```
A = @task job_A(2)      # A's task takes 2 seconds
B = @task job_B(1)      # B's task takes 1 second

schedule(A) |> wait
schedule(B) |> wait
```

```
A = @task job_A(2)      # A's task takes 2 seconds
B = @task job_B(1)      # B's task takes 1 second

(schedule(A), schedule(B)) .|> wait
```

Note that `wait` was added even for the concurrent case, although after both tasks are scheduled. The intention is that both tasks must be processed at the same time, without implying that any subsequent operation is also processed concurrently.

The example reveals the benefits of task switching under concurrency: although only one task can run at any given time, task A can yield control of the thread to task B when it becomes idle. In the code, the idle state is simulated by the function `sleep`, during which the computer isn't performing operations. Once task A becomes idle, its state is saved, allowing it to eventually resume execution from where it left off. In the meantime, task B can utilize that thread's processing time, explaining why B finishes first.

By taking turns efficiently and sharing the single available thread, tasks can make the most of the CPU's processing power. This contrasts with a sequential approach, where task A must finish before moving to the next task. The difference is reflected in their execution times, resulting in 2 seconds for a concurrent approach and 3 seconds for the sequential one.

Real-world examples of idle states emerge naturally in various scenarios. For instance, it's common when a program is waiting for a user input, such as a keystroke or mouse click. It can also arise when browsing the internet, where the CPU may idle while waiting for a server to send data. Task switching is so ubiquitous in certain contexts that we often take it for granted. For instance, I bet you never questioned whether you could use the computer while a document is printing in the background!

Note, though, that concurrency with a single thread offers no benefits if both tasks require active computations. This is because the CPU would be fully utilized, leaving no opportunity for task switching. In such cases, the sequential and concurrent approaches become equivalent. In our example, this would occur if task B consisted of computing `1+1` repeatedly, determining an execution time of 3 seconds for both approaches.

```
function job(name_worker, time_working)
    start_time = time()

    while time() - start_time < time_working
        1 + 1                  # compute `1+1` repeatedly during `time_working` seconds
    end

    println("$name_worker completed his task")
end
```

```
function schedule_of_tasks()
    A = @task job("A", 2)      # A's task takes 2 seconds
    B = @task job("B", 1)      # B's task takes 1 second

    schedule(A) |> wait
    schedule(B) |> wait
end
```

```
function schedule_of_tasks()
    A = @task job("A", 2)      # A's task takes 2 seconds
    B = @task job("B", 1)      # B's task takes 1 second

    (schedule(A), schedule(B)) .|> wait
end
```

Nonetheless, the key insight from the examples isn't the ineffectiveness of concurrency in a session with a single thread. Instead, the main takeaway is the underlying procedure outlined: **when a task is scheduled, the computer will attempt to find an available thread for its computation**. For concurrency, this implies that **starting a session with multiple threads will result in parallel code execution**, which is simply referred to as **multithreading**. In the following, we explain this case in more detail.

MULTITHREADING

Let's continue considering the last scenario, where both workers A and B perform meaningful computations. The only change we introduce is that Julia's session now starts with more than one thread available. For the concurrent approach, the only modification we make to the code is that tasks

aren't "sticky". This is just a technicality indicating that a task can be run on any thread, rather than only the thread they were first scheduled on. Non-sticky tasks allow for a better use of resources, as the task can be computed as soon as a thread is available.

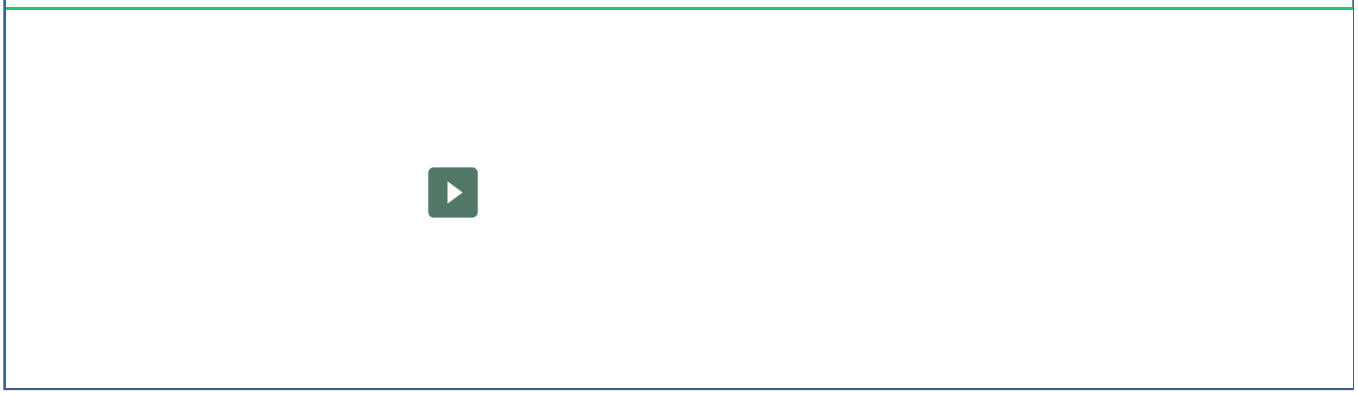
```
function schedule_of_tasks()
    A = @task job("A", 2)                          # A's task takes 2 seconds
    B = @task job("B", 1)                          # B's task takes 1 second

    schedule(A) |> wait
    schedule(B) |> wait
end
```

```
function schedule_of_tasks()
    A = @task job("A", 2) ; A.sticky = false       # A's task takes 2 seconds
    B = @task job("B", 1) ; B.sticky = false       # B's task takes 1 second

    (schedule(A), schedule(B)) .|> wait
end
```

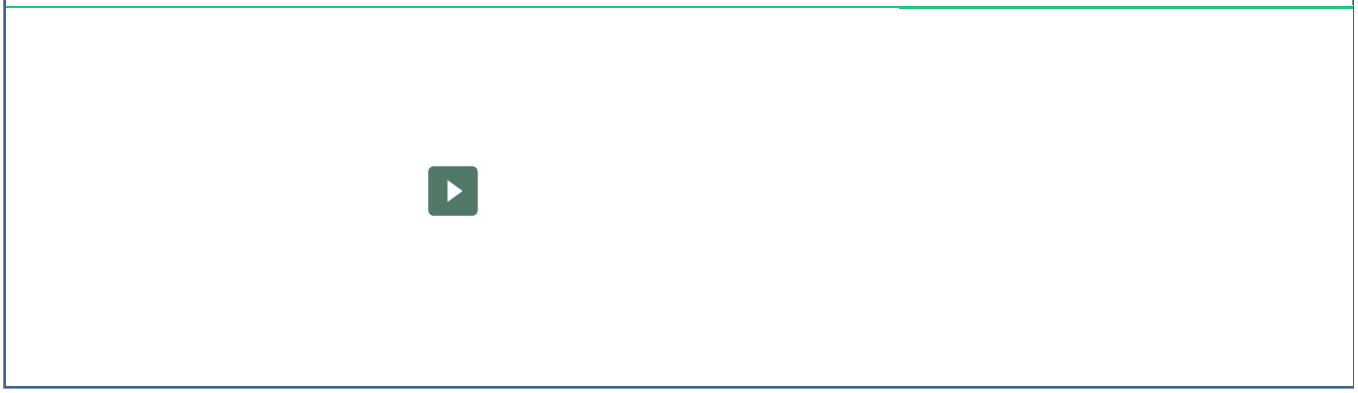
Once there's more than one thread available, concurrency implies simultaneity. This means that each task is run on a different thread, explaining why task B finishes first.

Anticipating some of the approaches that we'll use in the next section, we can directly compare Julia's standard code and a multithreaded implementation. Computing tasks in a multithreaded environment can be done in a simpler way through the macro `@spawn`. Its application will be explained in more detail in the next section, but it's basically equivalent to creating and scheduling a non-sticky task. Overall, the following two code snippets internally compute tasks in the ways explained.

```
function schedule_of_tasks()
    A = job("A", 2)             # A's task takes 2 seconds
    B = job("B", 1)             # B's task takes 1 second
end
```



```
function schedule_of_tasks()
    A = @spawn job("A", 2)      # A's task takes 2 seconds
    B = @spawn job("B", 1)      # B's task takes 1 second

    (A,B) .|> wait
end
```



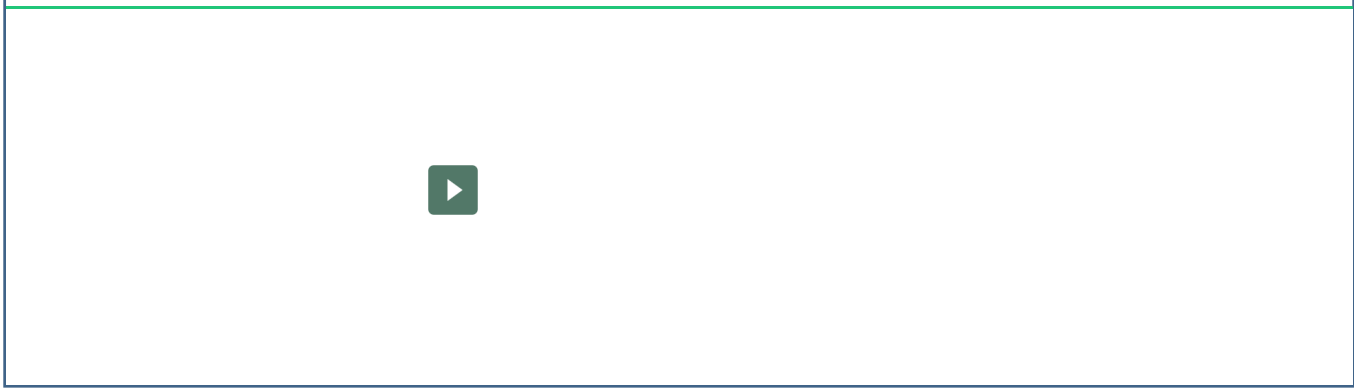
THE IMPORTANCE OF WAITING FOR THE RESULTS

Before finishing this section, let's emphasize an aspect that deserves careful consideration: we should always wait for all the operations to finish before moving forward. This is even necessary for concurrent computations. **Failing to wait for the results can lead to wrong outputs, even with a single thread**.

To demonstrate this possibility as starkly as possible, let's consider a vector mutation in a session with single thread. Moreover, we'll perform each mutation with one second of delay per value. The example reveals that, if we don't wait for the mutation to finish, any subsequent operation will be based on the value of the vector at the moment of execution. This value doesn't necessarily reflect the final output, but merely the value held at the moment it's called.

For instance, suppose we want to mutate the vector `x = [0,0,0]` to obtain `x = [1,2,3]`. As Julia's default approach is sequential, the mutation of the vector must be finished before continuing with any other operation.

```
# Description of job
function job!(x)
    for i in 1:3
        sleep(1)      # do nothing for 1 second
        x[i] = 1      # mutate x[i]

        println("`x` at this moment is $x")
    end
end

# Execution of job
function foo()
    x = [0, 0, 0]

    job!(x)           # slowly mutate `x`

    return sum(x)
end

output = foo()
println("the value stored in `output` is $(output)")
```



Consider now the implementation at a lower level through tasks. In particular, a task performing a mutation is defined in the following way.

```
function job!(x)
    @task begin
        for i in 1:3
            sleep(1)     # do nothing for 1 second
            x[i] = 1     # mutate x[i]

            println("`x` at this moment is $x")
        end
    end
end
```

Given this task, the following code snippets show the consequences of waiting and not waiting for the mutation to complete.

```
function foo()
    x = [0, 0, 0]

    job!(x) |> schedule             # define job, start execution, don't wait for job to be
done

    return sum(x)
end

output = foo()
println("the value stored in `output` is $(output)")
```

```
function foo()
    x = [0, 0, 0]

     job!(x) |> schedule |> wait       # define job, start execution, only continue when
finished

    return sum(x)
end

output = foo()
println("the value stored in `output` is $(output)")
```

As we can see, without waiting for the mutation to take place, the subsequent operation takes the value of `x` at the moment of execution. Since the mutation hasn't started, `x` is still `x = [0,0,0]`.