Multithreading & Asynchronous Programming in Julia

Plan

1 Setup and Context

- What is multithreading and parallel computing
- Julia Tasks

2 Multithreading

- Basic multithreaded loop
- Scheduling strategies
- Data race condition
- Locks (Mutex)
- Atomic operations

3 Asynchronous Programming

- async/sync
- Channel & Tasks

Setup and Context

What is Multithreading?

Multithreading allows a program to run **multiple computations at the same time** on separate CPU cores. This is especially useful for **CPU-bound tasks** where raw computing power is the bottleneck.

In Julia, multithreading is enabled with the environment variable `JULIA_NUM_THREADS`, and used via macros like `Threads.@threads` and `Threads.@spawn`.

The goal is to split work into smaller, independent parts that can be safely computed in parallel.

We'll also talk about asynchronous programming, which defines coroutines running at the same time on a single thread, and able to yield control to each other while they wait for something.

16

```
1 Threads.nthreads()
```

Julia Tasks

I will refer to Tasks many times during this workshop because they're the unit of computational work when it comes to dispatching work across threads or process it asynchronously.

From the julia documentation:

> You can think of a Task as a handle to a unit of computational work to be performed. It has a create-start-run-finish lifecycle. Tasks are created by calling the Task constructor on a 0-argument function to run, or using the @task macro:

```
t = Task (runnable) @0x000001ad8e9409c0
```

```
1 t = @task begin; sleep(5); println("done"); end
```

A task is defined, but not executed by default. It needs to be scheduled to be executed (see scheduling strategies in the multithreading section)

```
1 schedule(t); wait(t)
```

```
done
```

this executes the task on whatever thread scheduled it. But since our CPU has many threads, we can send any task to be handled by another thread!

Multithreading

Basic multithreaded loop

Threads like to be dispatched on repetitive CPU bound process, which is why we generally dispatch independant operations of a loop on a thread.

```julia
begin
    max_val = 10_000_000
    output = zeros(Int, max_val)

    # Single Thread
    @time  for i in 1:max_val
        output[i] = i^2
    end
    println(output[1:10])

    output = zeros(Int, max_val)
    # Dispatching threads with :dynamic scheduling
    @time Threads.@threads for i in 1:max_val
        output[i] = i^2
    end
    println(output[1:10])
end
```

```
  1.689201 seconds (40.00 M allocations: 762.924 MiB, 4.79% gc time)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
  0.189529 seconds (20.04 M allocations: 307.298 MiB, 37.50% gc time, 167.45% comp
ilation time)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- Tasks that can be done indepentendly from each other can easiliy be multi-threaded.
- There's an overhead cost to doing things with multiple threads. Meaning that if your tasks are not very long or you don't have a lot of them, you might slow the process with multiple threads.
    - Try with a max_val of 10_000_000, then 100.

Scheduling Strategies

julia supports three scheduling strategies since 1.11. Those dictate how your tasks are assigned to threads in the threadpool.

:static

The iterator is divided upfront into equal chunk of tasks, with each thread assigned a chunk once. This assumes that every task takes the exact same computing resources. This leads to very low overhead costs but will lead to idle threads if every task is not uniform in required computing time. Requires the iterator to have a known length.

:greedy

Every thread in the threadpool is assigned the "next available" small chunk of tasks (typically a single task), meaning they take the next iteration as soon as they're spawned or done working. This reassigns threads quickly, which handles uneven workloads better but leads to higher scheduling overhead. Does not require the iterator to have a known length. Good if workload is not uniform

:dynamic (default)

Threads pull fixed-size chunks of work dynamically from a shared pool. Each thread takes a chunk, processes it, then takes another when finished. This strikes a balance between static by having better thread reassignement to handle uneven workload, while having less overhead than :greedy. Requires the iterator to have a known length.

```
busy (generic function with 1 method)
```

```
function busy(i; verbose=true)
    sleep(0.001 * rand())
    verbose && println("i=$i done by thread $(Threads.threadid())")
end
```

```
begin
    @time Threads.@threads :static for i in 1:32
        busy(i, verbose=true)
    end
end
```

```
i=1 done by thread 1
i=27 done by thread 14
i=15 done by thread 8
i=25 done by thread 13
i=3 done by thread 2
i=21 done by thread 11
i=13 done by thread 7
i=29 done by thread 15
i=9 done by thread 5
i=17 done by thread 9
i=19 done by thread 10
i=5 done by thread 3
i=23 done by thread 12
i=31 done by thread 16
i=11 done by thread 6
i=7 done by thread 4
i=4 done by thread 2
i=8 done by thread 4
i=24 done by thread 12
i=10 done by thread 5
i=6 done by thread 3
i=28 done by thread 14
i=32 done by thread 16
i=26 done by thread 13
i=12 done by thread 6
i=22 done by thread 11
i=20 done by thread 10
i=14 done by thread 7
i=16 done by thread 8
i=30 done by thread 15
i=2 done by thread 1
```

With static, each thread was assigned 2 iterations at the very start (thread 1 did operation 1 & 2, thread 2 did 3 & 4, etc...)

```
begin
    @time Threads.@threads :greedy for i in 1:32
        busy(i, verbose=true)
    end
end
```

```
i=10 done by thread 3
i=14 done by thread 15
i=5 done by thread 12
i=12 done by thread 2
i=2 done by thread 4
i=13 done by thread 16
i=9 done by thread 11
i=15 done by thread 14
i=7 done by thread 1
i=8 done by thread 6
i=3 done by thread 8
i=6 done by thread 5
i=4 done by thread 13
i=24 done by thread 4
i=21 done by thread 15
i=29 done by thread 8
i=20 done by thread 5
i=18 done by thread 14
i=31 done by thread 6
i=28 done by thread 10
i=32 done by thread 10
i=23 done by thread 2
i=27 done by thread 9
i=17 done by thread 1
i=19 done by thread 12
i=22 done by thread 3
i=25 done by thread 13
i=30 done by thread 7
i=26 done by thread 16
  0.150138 seconds (184.42 k allocations: 9.438 MiB, 3.48% gc time, 39 lock con
flicts, 524.74% compilation time)
```

```
begin
    @time Threads.@threads :dynamic for i in 1:32
        busy(i, verbose=true)
    end
end
```

```
i=3 done by thread 14
i=31 done by thread 12
i=9 done by thread 1
i=27 done by thread 5
i=7 done by thread 16
i=25 done by thread 11
i=1 done by thread 2
i=11 done by thread 4
i=15 done by thread 3
i=23 done by thread 9
i=19 done by thread 10
i=17 done by thread 15
i=21 done by thread 8
i=13 done by thread 6
i=29 done by thread 7
i=5 done by thread 13
i=32 done by thread 12
i=22 done by thread 8
i=4 done by thread 1
i=28 done by thread 2
i=8 done by thread 14
i=18 done by thread 15
i=16 done by thread 16
i=30 done by thread 7
i=12 done by thread 4
i=2 done by thread 13
i=10 done by thread 5
i=6 done by thread 10
i=24 done by thread 9
i=26 done by thread 11
i=20 done by thread 3
```

With greedy or dynamic, there's not much sense to what thread does what. All of them are either greedily grabbing the next iteration as soon as they can, or the chunks are interleaved among threads as they become available. Try with 10_000 iterations to see a difference.

Data Race Conditions

The bane of multithreading is data-racing. The core rule of multithreading and parallel computing is **never let multiple threads change the same thing at the same time**.

Single thread incrementing a variable (expected behaviour)

```
singlethread_count (generic function with 1 method)
```

```
function singlethread_count()
    total_single = 0
    for i in 1:1_000_000
        total_single += 1
    end
    println("Total (single thread): $total_single")
end
```

Multiple threads incrementing a variable (data race)

```
multithread_count (generic function with 1 method)
```

```
function multithread_count()
    total_threads = 0
    Threads.@threads for i in 1:1_000_000
        total_threads += 1
    end
    println("Total (multithreaded): $total_threads")
end
```

```
begin
    @time singlethread_count()
    @time multithread_count()
end
```

```
Total (single thread): 1000000
  0.000078 seconds (23 allocations: 672 bytes)
Total (multithreaded): 63554
  0.033598 seconds (1.01 M allocations: 15.878 MiB, 663.38% compilation time)
```

Solving Data Race Conditions

There are three ways of solving a data race:

- Adapting the algorithm in such a way that the data race is solved (best solution, think parallel merging of sorted tables)
- Using locks to prevent threads from writing in the same space at the same time
- Using atomic operations when possible

Using Locks

```
begin
    function locked_multithread_count()
        total_threads = 0
        lock = Threads.SpinLock()
        Threads.@threads for i in 1:1_000_000
            Threads.lock(lock) do
                total_threads += 1
            end
        end
        println("Total (multithreaded with lock): $total_threads")
    end
    @time locked_multithread_count()
end
```

```
Total (multithreaded with lock): 1000000
  0.216347 seconds (1.05 M allocations: 18.056 MiB, 3.48% gc time, 202.07% compila
tion time)
```

Each thread can only execute whatever is in the locked block if another thread isn't already doing it. In this case this is absolutely horrendous since the only operation of the loop is then done one thread at a time, which means it's equivalent to using 1 thread while paying the overhead of multithreading and keeping track of a lock

Using Atomic

```
begin
    function atomic_multithread_count()
        total = Threads.Atomic{Int}(0)

        Threads.@threads for i in 1:1_000_000
            Threads.atomic_add!(total, 1)
        end

        println("Total (multithreaded with atomic): $(total[])")
    end
    @time atomic_multithread_count()
end
```

```
Total (multithreaded with atomic): 1000000
  0.047034 seconds (52.08 k allocations: 2.618 MiB, 483.52% compilation time)
```

Atomic operations honestly kinda magical. Whenever what you're doing is a short serie of simple CPU instructions (like a "add" here), you can use atomic operations to ensure it's indivisible and not interuptable, which garanties that no thread can corrupt the result. This is generally much faster than locking if you only have simple instructions, though here it's still slower than a single thread because there's no benefit to multithreading in this example

Asynchronous Programming

Unlike multi-threading, async programming is usually described as "I/O bound". It's a way make code that awaits a flux of input and runs asynchronously from whatever code launched it. Functions that can run asynchronously are called coroutines.

Note that asynchronous coroutines usually still run on a single thread (though you can mix multithreading and asynchrous programming), but coroutines can yield control while they wait for something (typically I/O), allowing other coroutines to work in the meantime.

The `@async` macro allows you to create Tasks that will be immediately scheduled, independently from the main thread.

```julia
begin
    function async_subprocess_demo()
        println("Starting async I/O from subprocesses...")

        # Start two processes that echo lines slowly
        p1 = open(`julia -e 'for i in 1:5; println("A $i"); sleep(0.5); end'`, "r")
        p2 = open(`julia -e 'for i in 1:5; println("B $i"); sleep(0.3); end'`, "r")

        # This coroutine reads from p1 and is schedule immediately
        t1 = @async for line in eachline(p1)
            println("From t1: line $line")
        end

        # This coroutine reads from p2 and is also scheduled immediately
        t2 = @async for line in eachline(p2)
            println("From t2: line $line")
        end

        println("Main thread is free to do other things...")

        # Wait allows you to stop main thread until a Task is done
        wait(t1)
        wait(t2)
        println("All subprocess output received.")
    end

    async_subprocess_demo()
end
```

```
Starting async I/O from subprocesses...
Main thread is free to do other things...
From t1: line A 1
From t2: line B 1
From t2: line B 2
From t1: line A 2
From t2: line B 3
From t2: line B 4
From t1: line A 3
From t2: line B 5
From t1: line A 4
From t1: line A 5
All subprocess output received.
```

Channels are queues that are synchronized and accessible accross your coroutines. You can then organise code in a "producer/consumer" way. A producer Task puts items in a queue, a consumer tasks takes items from the queue whenever one is available (exploiting the I/O bound nature of coroutines).

```julia
begin
    function async_pipeline_basic()
        jobs = Channel{Int64}(10)  # This channel has a maximum of 10 items

        # Producer task
        prod = @async try
            for i in 1:10
                println("Producing: Task $i")
                put!(jobs, i)
                sleep(0.5)  # simulate spacing out jobs
            end
            close(jobs)  # signal no more jobs
        catch e
            @error "producer crashed..." exception=(e, catch_backtrace())
        end

        # Consumer task
        cons = @async try
            for task in jobs
                println("Processing: Task $task")
                sleep(1.0)  # simulate longer work
            end
            println("All jobs processed.")
        catch e
            @error "consumer crashed..." exception=(e, catch_backtrace())
        end

        println("Pipeline running... (main thread)")
        wait(cons)  # Wait for the consumer to finish
    end

    @time async_pipeline_basic()
end
```

```
Pipeline running... (main thread)
Producing: Task 1
Processing: Task 1
Producing: Task 2
Processing: Task 2
Producing: Task 3
Producing: Task 4
Processing: Task 3
Producing: Task 5
Producing: Task 6
Processing: Task 4
Producing: Task 7
Producing: Task 8
Processing: Task 5
Producing: Task 9
Producing: Task 10
Processing: Task 6
Processing: Task 7
Processing: Task 8
Processing: Task 9
Processing: Task 10
All jobs processed.
```

```
 10.169881 seconds (79.81 k allocations: 4.162 MiB, 1 lock conflict, 0.47% compila
tion time)
```

Note the following:

- Channels are buffered, meaning they have a limited capacity. If a Tasks tries to put an item in a full queue, it will be paused until space is freed.
- Asynchronous code crashes *silently* unless you use a try block. If it seems your code is doing nothing for a long time, check that your coroutines are still alive

If you produce more than you consume, you can even create **multiple** consumers

```
[nothing, nothing, nothing]
```

```
begin
    function async_pipeline_multiconsume()
        jobs = Channel{Int64}(10)  # This channel has a maximum of 10 items

        # Producer task
        prod = @async try
            for i in 1:10
                println("Producing: Task $i")
                put!(jobs, i)
                sleep(0.5)  # simulate spacing out jobs
            end
            close(jobs)  # signal no more jobs
        catch e
            @error "producer crashed..." exception=(e, catch_backtrace())
        end

        # We want to store all consumers
        consumers = Vector{Task}()
        for cid in 1:3  # Making 3 consumers
            cons = @async try
                for task in jobs
                    println("Processing: Task $task")
                    sleep(1.0)  # simulate longer work
                end
                println("All jobs processed.")
            catch e
                @error "consumer crashed..." exception=(e, catch_backtrace())
            end
            push!(consumers, cons)
        end

        println("Pipeline running... (main thread)")
        wait.(consumers)  # Wait for the consumers to finish
    end

    @time async_pipeline_multiconsume()
end
```

```
Pipeline running... (main thread)
Producing: Task 1
Processing: Task 1
Producing: Task 2
Processing: Task 2
Producing: Task 3
Processing: Task 3
Producing: Task 4
Processing: Task 4
Producing: Task 5
Processing: Task 5
Producing: Task 6
Processing: Task 6
Producing: Task 7
Processing: Task 7
Producing: Task 8
Processing: Task 8
```

```
Producing: Task 9
Processing: Task 9
Producing: Task 10
Processing: Task 10
All jobs processed.
All jobs processed.
All jobs processed.
  5.650571 seconds (212.10 k allocations: 10.914 MiB, 2 lock conflicts, 1.16% comp
ilation time)
```

Note that this is already faster, but this is not even the most beneficicial scenario where the queue becomes eventually full. With a much smaller queue, or a situation where the queue gets filled fast, having multiple consumers can prevent the producer from idling, saving a lot of time.