Multi-Threading

Visit this blog post for a presentation of Julia multi-threading features.

Starting Julia with multiple threads

By default, Julia starts up with a single thread of execution. This can be verified by using the command `Threads.nthreads()`:

```
julia> Threads.nthreads()
1
```

The number of execution threads is controlled either by using the `-t/--threads` command line argument or by using the `JULIA_NUM_THREADS` environment variable. When both are specified, then `-t/--threads` takes precedence.

> **Julia 1.5**
>
> The `-t/--threads` command line argument requires at least Julia 1.5. In older versions you must use the environment variable instead.

Lets start Julia with 4 threads:

```
$ julia --threads 4
```

Let's verify there are 4 threads at our disposal.

```
julia> Threads.nthreads()
4
```

But we are currently on the master thread. To check, we use the function `Threads.threadid`

```
julia> Threads.threadid()
1
```

Note

If you prefer to use the environment variable you can set it as follows in Bash (Linux/macOS):

```
export JULIA_NUM_THREADS=4
```

C shell on Linux/macOS, CMD on Windows:

```
set JULIA_NUM_THREADS=4
```

Powershell on Windows:

```
$env:JULIA_NUM_THREADS=4
```

Note that this must be done *before* starting Julia.

Note

The number of threads specified with `-t/--threads` is propagated to worker processes that are spawned using the `-p/--procs` or `--machine-file` command line options. For example, `julia -p2 -t2` spawns 1 main process with 2 worker processes, and all three processes have 2 threads enabled. For more fine grained control over worker threads use `addprocs` and pass `-t/--threads` as `exeflags`.

Data-race freedom

You are entirely responsible for ensuring that your program is data-race free, and nothing promised here can be assumed if you do not observe that requirement. The observed results may be highly unintuitive.

The best way to ensure this is to acquire a lock around any access to data that can be observed from multiple threads. For example, in most cases you should use the following code pattern:

```
julia> lock(a) do
           use(a)
       end
```

```
julia> begin
           lock(a)
           try
               use(a)
           finally
               unlock(a)
           end
       end
```

Additionally, Julia is not memory safe in the presence of a data race. Be very careful about reading a global variable (or closure variable) if another thread might write to it! Instead, always use the lock pattern above when changing any data (such as assigning to a global) visible to multiple threads.

```
Thread 1:
global b = false
global a = rand()
global b = true

Thread 2:
while !b; end
bad(a) # it is NOT safe to access `a` here!

Thread 3:
while !@isdefined(a); end
use(a) # it is NOT safe to access `a` here
```

The `@threads` Macro

Let's work a simple example using our native threads. Let us create an array of zeros:

```
julia> a = zeros(10)
10-element Array{Float64,1}:
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
```

Let us operate on this array simultaneously using 4 threads. We'll have each thread write its thread ID into each location.

Julia supports parallel loops using the `Threads.@threads` macro. This macro is affixed in front of a `for` loop to indicate to Julia that the loop is a multi-threaded region:

```
julia> Threads.@threads for i = 1:10
           a[i] = Threads.threadid()
       end
```

The iteration space is split among the threads, after which each thread writes its thread ID to its assigned locations:

```
julia> a
10-element Array{Float64,1}:
 1.0
 1.0
 1.0
 2.0
 2.0
 2.0
 3.0
 3.0
 4.0
 4.0
```

Note that `Threads.@threads` does not have an optional reduction parameter like `@distributed`.

Atomic Operations

Julia supports accessing and modifying values *atomically*, that is, in a thread-safe way to avoid race conditions. A value (which must be of a primitive type) can be wrapped as `Threads.Atomic` to indicate it must be accessed in this way. Here we can see an example:

```
julia> i = Threads.Atomic{Int}(0);

julia> ids = zeros(4);

julia> old_is = zeros(4);

julia> Threads.@threads for id in 1:4
           old_is[id] = Threads.atomic_add!(i, id)
```

```
            ids[id] = id
        end

julia> old_is
4-element Array{Float64,1}:
 0.0
 1.0
 7.0
 3.0

julia> ids
4-element Array{Float64,1}:
 1.0
 2.0
 3.0
 4.0
```

Had we tried to do the addition without the atomic tag, we might have gotten the wrong answer due to a race condition. An example of what would happen if we didn't avoid the race:

```
julia> using Base.Threads

julia> nthreads()
4

julia> acc = Ref(0)
Base.RefValue{Int64}(0)

julia> @threads for i in 1:1000
           acc[] += 1
       end

julia> acc[]
926

julia> acc = Atomic{Int64}(0)
Atomic{Int64}(0)

julia> @threads for i in 1:1000
           atomic_add!(acc, 1)
       end

julia> acc[]
1000
```

> **Note**
>
> Not *all* primitive types can be wrapped in an `Atomic` tag. Supported types are `Int8`, `Int16`, `Int32`, `Int64`, `Int128`, `UInt8`, `UInt16`, `UInt32`, `UInt64`, `UInt128`, `Float16`, `Float32`, and `Float64`. Additionally, `Int128` and `UInt128` are not supported on AAarch32 and ppc64le.

Side effects and mutable function arguments

When using multi-threading we have to be careful when using functions that are not pure as we might get a wrong answer. For instance functions that have a name ending with `!` by convention modify their arguments and thus are not pure.

@threadcall

External libraries, such as those called via `ccall`, pose a problem for Julia's task-based I/O mechanism. If a C library performs a blocking operation, that prevents the Julia scheduler from executing any other tasks until the call returns. (Exceptions are calls into custom C code that call back into Julia, which may then yield, or C code that calls `jl_yield()`, the C equivalent of `yield`.)

The `@threadcall` macro provides a way to avoid stalling execution in such a scenario. It schedules a C function for execution in a separate thread. A threadpool with a default size of 4 is used for this. The size of the threadpool is controlled via environment variable `UV_THREADPOOL_SIZE`. While waiting for a free thread, and during function execution once a thread is available, the requesting task (on the main Julia event loop) yields to other tasks. Note that `@threadcall` does not return until the execution is complete. From a user point of view, it is therefore a blocking call like other Julia APIs.

It is very important that the called function does not call back into Julia, as it will segfault.

`@threadcall` may be removed/changed in future versions of Julia.

Caveats

At this time, most operations in the Julia runtime and standard libraries can be used in a thread-safe manner, if the user code is data-race free. However, in some areas work on stabilizing thread support is ongoing. Multi-threaded programming has many inherent difficulties, and if a program using threads exhibits unusual or undesirable behavior (e.g. crashes or mysterious results), thread interactions should typically be suspected first.

There are a few specific limitations and warnings to be aware of when using threads in Julia:

- Base collection types require manual locking if used simultaneously by multiple threads where at least one thread modifies the collection (common examples include `push!` on arrays, or inserting items into a `Dict`).
- After a task starts running on a certain thread (e.g. via `@spawn`), it will always be restarted on the same thread after blocking. In the future this limitation will be removed, and tasks will migrate between threads.
- `@threads` currently uses a static schedule, using all threads and assigning equal iteration counts to each. In the future the default schedule is likely to change to be dynamic.
- The schedule used by `@spawn` is nondeterministic and should not be relied on.
- Compute-bound, non-memory-allocating tasks can prevent garbage collection from running in other threads that are allocating memory. In these cases it may be necessary to insert a manual call to `GC.safepoint()` to allow GC to run. This limitation will be removed in the future.
- Avoid running top-level operations, e.g. `include`, or `eval` of type, method, and module definitions in parallel.
- Be aware that finalizers registered by a library may break if threads are enabled. This may require some transitional work across the ecosystem before threading can be widely adopted with confidence. See the next section for further details.

Safe use of Finalizers

Because finalizers can interrupt any code, they must be very careful in how they interact with any global state. Unfortunately, the main reason that finalizers are used is to update global state (a pure function is generally rather pointless as a finalizer). This leads us to a bit of a conundrum. There are a few approaches to dealing with this problem:

1. When single-threaded, code could call the internal `jl_gc_enable_finalizers` C function to prevent finalizers from being scheduled inside a critical region. Internally, this is used inside some functions (such as our C locks) to prevent recursion when doing certain operations (incremental package loading, codegen, etc.). The combination of a lock and this flag can be used to make finalizers safe.
2. A second strategy, employed by Base in a couple places, is to explicitly delay a finalizer until it may be able to acquire its lock non-recursively. The following example demonstrates how this strategy could be applied to `Distributed.finalize_ref`:

```
function finalize_ref(r::AbstractRemoteRef)
    if r.where > 0 # Check if the finalizer is already run
        if islocked(client_refs) || !trylock(client_refs)
            # delay finalizer for later if we aren't free to acquire the lock
            finalizer(finalize_ref, r)
```

```
            return nothing
        end
        try # `lock` should always be followed by `try`
            if r.where > 0 # Must check again here
                # Do actual cleanup here
                r.where = 0
            end
        finally
            unlock(client_refs)
        end
    end
    nothing
end
```

3. A related third strategy is to use a yield-free queue. We don't currently have a lock-free queue implemented in Base, but `Base.InvasiveLinkedListSynchronized{T}` is suitable. This can frequently be a good strategy to use for code with event loops. For example, this strategy is employed by `Gtk.jl` to manage lifetime ref-counting. In this approach, we don't do any explicit work inside the `finalizer`, and instead add it to a queue to run at a safer time. In fact, Julia's task scheduler already uses this, so defining the finalizer as `x -> @spawn do_cleanup(x)` is one example of this approach. Note however that this doesn't control which thread `do_cleanup` runs on, so `do_cleanup` would still need to acquire a lock. That doesn't need to be true if you implement your own queue, as you can explicitly only drain that queue from your thread.

« Asynchronous Programming Multi-processing and Distributed Computing »