Multi-Threading

`Base.Threads.@threads` — Macro

```
Threads.@threads [schedule] for ... end
```

A macro to parallelize a `for` loop to run with multiple threads. Splits the iteration space among multiple tasks and runs those tasks on threads according to a scheduling policy. A barrier is placed at the end of the loop which waits for all tasks to finish execution.

The `schedule` argument can be used to request a particular scheduling policy. The only currently supported value is `:static`, which creates one task per thread and divides the iterations equally among them. Specifying `:static` is an error if used from inside another `@threads` loop or from a thread other than 1.

The default schedule (used when no `schedule` argument is present) is subject to change.

> Julia 1.5
>
> The `schedule` argument is available as of Julia 1.5.

`Base.Threads.@spawn` — Macro

```
Threads.@spawn expr
```

Create and run a `Task` on any available thread. To wait for the task to finish, call `wait` on the result of this macro, or call `fetch` to wait and then obtain its return value.

Values can be interpolated into `@spawn` via `$`, which copies the value directly into the constructed underlying closure. This allows you to insert the *value* of a variable, isolating the aysnchronous code from changes to the variable's value in the current task.

> Note

See the manual chapter on threading for important caveats.

Julia 1.3

This macro is available as of Julia 1.3.

Julia 1.4

Interpolating values via `$` is available as of Julia 1.4.

`Base.Threads.threadid` — Function

```
Threads.threadid()
```

Get the ID number of the current thread of execution. The master thread has ID `1`.

`Base.Threads.nthreads` — Function

```
Threads.nthreads()
```

Get the number of threads available to the Julia process. This is the inclusive upper bound on `threadid()`.

Synchronization

`Base.Threads.Condition` — Type

```
Threads.Condition([lock])
```

A thread-safe version of `Base.Condition`.

To call `wait` or `notify` on a `Threads.Condition`, you must first call `lock` on it. When `wait` is called, the lock is atomically released during blocking, and will be reacquired before `wait` returns. Therefore idiomatic use of a `Threads.Condition` `c` looks like the following:

```
lock(c)
try
    while !thing_we_are_waiting_for
        wait(c)
    end
finally
    unlock(c)
end
```

> **Julia 1.2**
>
> This functionality requires at least Julia 1.2.

`Base.Event` — Type

```
Event()
```

Create a level-triggered event source. Tasks that call `wait` on an `Event` are suspended and queued until `notify` is called on the `Event`. After `notify` is called, the `Event` remains in a signaled state and tasks will no longer block when waiting for it.

> **Julia 1.1**
>
> This functionality requires at least Julia 1.1.

See also Synchronization.

Atomic operations

> **Warning**

The API for atomic operations has not yet been finalized and is likely to change.

`Base.Threads.Atomic` — Type

```
Threads.Atomic{T}
```

Holds a reference to an object of type `T`, ensuring that it is only accessed atomically, i.e. in a thread-safe manner.

Only certain "simple" types can be used atomically, namely the primitive boolean, integer, and float-point types. These are `Bool`, `Int8`...`Int128`, `UInt8`...`UInt128`, and `Float16`...`Float64`.

New atomic objects can be created from a non-atomic values; if none is specified, the atomic object is initialized with zero.

Atomic objects can be accessed using the `[]` notation:

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> x[] = 1
1

julia> x[]
1
```

Atomic operations use an `atomic_` prefix, such as `atomic_add!`, `atomic_xchg!`, etc.

`Base.Threads.atomic_cas!` — Function

```
Threads.atomic_cas!(x::Atomic{T}, cmp::T, newval::T) where T
```

Atomically compare-and-set `x`

Atomically compares the value in `x` with `cmp`. If equal, write `newval` to `x`. Otherwise, leaves `x` unmodified. Returns the old value in `x`. By comparing the returned value to `cmp` (via `===`) one knows whether `x` was modified and now holds the new value `newval`.

For further details, see LLVM's `cmpxchg` instruction.

This function can be used to implement transactional semantics. Before the transaction, one records the value in `x`. After the transaction, the new value is stored only if `x` has not been modified in the mean time.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_cas!(x, 4, 2);

julia> x
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_cas!(x, 3, 2);

julia> x
Base.Threads.Atomic{Int64}(2)
```

`Base.Threads.atomic_xchg!` — Function

```
Threads.atomic_xchg!(x::Atomic{T}, newval::T) where T
```

Atomically exchange the value in `x`

Atomically exchanges the value in `x` with `newval`. Returns the old value.

For further details, see LLVM's `atomicrmw xchg` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_xchg!(x, 2)
3

julia> x[]
```

```
2
```

`Base.Threads.atomic_add!` — Function

```
Threads.atomic_add!(x::Atomic{T}, val::T) where T <: ArithmeticTypes
```

Atomically add `val` to `x`

Performs `x[] += val` atomically. Returns the old value. Not defined for `Atomic{Bool}`.

For further details, see LLVM's `atomicrmw add` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_add!(x, 2)
3

julia> x[]
5
```

`Base.Threads.atomic_sub!` — Function

```
Threads.atomic_sub!(x::Atomic{T}, val::T) where T <: ArithmeticTypes
```

Atomically subtract `val` from `x`

Performs `x[] -= val` atomically. Returns the old value. Not defined for `Atomic{Bool}`.

For further details, see LLVM's `atomicrmw sub` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_sub!(x, 2)
```

```
3

julia> x[]
1
```

`Base.Threads.atomic_and!` — Function

```
Threads.atomic_and!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-and `x` with `val`

Performs `x[] &= val` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw and` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_and!(x, 2)
3

julia> x[]
2
```

`Base.Threads.atomic_nand!` — Function

```
Threads.atomic_nand!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-nand (not-and) `x` with `val`

Performs `x[] = ~(x[] & val)` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw nand` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
```

```
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_nand!(x, 2)
3

julia> x[]
-3
```

`Base.Threads.atomic_or!` — Function

```
Threads.atomic_or!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-or `x` with `val`

Performs `x[] |= val` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw or` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(5)
Base.Threads.Atomic{Int64}(5)

julia> Threads.atomic_or!(x, 7)
5

julia> x[]
7
```

`Base.Threads.atomic_xor!` — Function

```
Threads.atomic_xor!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-xor (exclusive-or) `x` with `val`

Performs `x[] $= val` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw xor` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(5)
Base.Threads.Atomic{Int64}(5)

julia> Threads.atomic_xor!(x, 7)
5

julia> x[]
2
```

`Base.Threads.atomic_max!` — Function

```
Threads.atomic_max!(x::Atomic{T}, val::T) where T
```

Atomically store the maximum of `x` and `val` in `x`

Performs `x[] = max(x[], val)` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw max` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(5)
Base.Threads.Atomic{Int64}(5)

julia> Threads.atomic_max!(x, 7)
5

julia> x[]
7
```

`Base.Threads.atomic_min!` — Function

```
Threads.atomic_min!(x::Atomic{T}, val::T) where T
```

Atomically store the minimum of `x` and `val` in `x`

Performs `x[] = min(x[], val)` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw min` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(7)
Base.Threads.Atomic{Int64}(7)

julia> Threads.atomic_min!(x, 5)
7

julia> x[]
5
```

`Base.Threads.atomic_fence` — Function

```
Threads.atomic_fence()
```

Insert a sequential-consistency memory fence

Inserts a memory fence with sequentially-consistent ordering semantics. There are algorithms where this is needed, i.e. where an acquire/release ordering is insufficient.

This is likely a very expensive operation. Given that all other atomic operations in Julia already have acquire/release semantics, explicit fences should not be necessary in most cases.

For further details, see LLVM's `fence` instruction.

ccall using a threadpool (Experimental)

`Base.@threadcall` — Macro

```
@threadcall((cfunc, clib), rettype, (argtypes...), argvals...)
```

The `@threadcall` macro is called in the same way as `ccall` but does the work in a different thread. This is useful when you want to call a blocking C function without causing the main `julia` thread to become blocked. Concurrency is limited by size of the libuv thread pool, which defaults to 4 threads but can be increased by setting the `UV_THREADPOOL_SIZE` environment variable and restarting the `julia` process.

Note that the called function should never call back into Julia.

Low-level synchronization primitives

These building blocks are used to create the regular synchronization objects.

`Base.Threads.SpinLock` — Type

```
SpinLock()
```

Create a non-reentrant, test-and-test-and-set spin lock. Recursive use will result in a deadlock. This kind of lock should only be used around code that takes little time to execute and does not block (e.g. perform I/O). In general, `ReentrantLock` should be used instead.

Each `lock` must be matched with an `unlock`.

Test-and-test-and-set spin locks are quickest up to about 30ish contending threads. If you have more contention than that, different synchronization approaches should be considered.

« Tasks Constants »