Tasks

`Core.Task` — Type

```
Task(func)
```

Create a `Task` (i.e. coroutine) to execute the given function `func` (which must be callable with no arguments). The task exits when this function returns.

Examples

```
julia> a() = sum(i for i in 1:1000);

julia> b = Task(a);
```

In this example, `b` is a runnable `Task` that hasn't started yet.

`Base.@task` — Macro

```
@task
```

Wrap an expression in a `Task` without executing it, and return the `Task`. This only creates a task, and does not run it.

Examples

```
julia> a1() = sum(i for i in 1:1000);

julia> b = @task a1();

julia> istaskstarted(b)
false

julia> schedule(b);
```

```
julia> yield();

julia> istaskdone(b)
true
```

`Base.@async` — Macro

```
@async
```

Wrap an expression in a `Task` and add it to the local machine's scheduler queue.

Values can be interpolated into `@async` 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.

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

`Base.asyncmap` — Function

```
asyncmap(f, c...; ntasks=0, batch_size=nothing)
```

Uses multiple concurrent tasks to map `f` over a collection (or multiple equal length collections). For multiple collection arguments, `f` is applied elementwise.

`ntasks` specifies the number of tasks to run concurrently. Depending on the length of the collections, if `ntasks` is unspecified, up to 100 tasks will be used for concurrent mapping.

`ntasks` can also be specified as a zero-arg function. In this case, the number of tasks to run in parallel is checked before processing every element and a new task started if the value of `ntasks_func` is less than the current number of tasks.

If `batch_size` is specified, the collection is processed in batch mode. `f` must then be a function that must accept a `Vector` of argument tuples and must return a vector of results. The input vector will have a length of `batch_size` or less.

The following examples highlight execution in different tasks by returning the `objectid` of the tasks in which the mapping function is executed.

First, with `ntasks` undefined, each element is processed in a different task.

```
julia> tskoid() = objectid(current_task());

julia> asyncmap(x->tskoid(), 1:5)
5-element Array{UInt64,1}:
 0x6e15e66c75c75853
 0x440f8819a1baa682
 0x9fb3eeadd0c83985
 0xebd3e35fe90d4050
 0x29efc93edce2b961

julia> length(unique(asyncmap(x->tskoid(), 1:5)))
5
```

With `ntasks=2` all elements are processed in 2 tasks.

```
julia> asyncmap(x->tskoid(), 1:5; ntasks=2)
5-element Array{UInt64,1}:
 0x027ab1680df7ae94
 0xa23d2f80cd7cf157
 0x027ab1680df7ae94
 0xa23d2f80cd7cf157
 0x027ab1680df7ae94

julia> length(unique(asyncmap(x->tskoid(), 1:5; ntasks=2)))
2
```

With `batch_size` defined, the mapping function needs to be changed to accept an array of argument tuples and return an array of results. `map` is used in the modified mapping function to achieve this.

```
julia> batch_func(input) = map(x->string("args_tuple: ", x, ", element_val: ", x
batch_func (generic function with 1 method)

julia> asyncmap(batch_func, 1:5; ntasks=2, batch_size=2)
5-element Array{String,1}:
 "args_tuple: (1,), element_val: 1, task: 9118321258196414413"
 "args_tuple: (2,), element_val: 2, task: 4904288162898683522"
 "args_tuple: (3,), element_val: 3, task: 9118321258196414413"
```

```
"args_tuple: (4,), element_val: 4, task: 4904288162898683522"
"args_tuple: (5,), element_val: 5, task: 9118321258196414413"
```

> **Note**
>
> Currently, all tasks in Julia are executed in a single OS thread co-operatively. Consequently, `asyncmap` is beneficial only when the mapping function involves any I/O - disk, network, remote worker invocation, etc.

`Base.asyncmap!` — Function

```
asyncmap!(f, results, c...; ntasks=0, batch_size=nothing)
```

Like `asyncmap`, but stores output in `results` rather than returning a collection.

`Base.current_task` — Function

```
current_task()
```

Get the currently running `Task`.

`Base.istaskdone` — Function

```
istaskdone(t::Task) -> Bool
```

Determine whether a task has exited.

Examples

```
julia> a2() = sum(i for i in 1:1000);

julia> b = Task(a2);

julia> istaskdone(b)
false
```

```
julia> schedule(b);

julia> yield();

julia> istaskdone(b)
true
```

`Base.istaskstarted` — Function

```
istaskstarted(t::Task) -> Bool
```

Determine whether a task has started executing.

Examples

```
julia> a3() = sum(i for i in 1:1000);

julia> b = Task(a3);

julia> istaskstarted(b)
false
```

`Base.istaskfailed` — Function

```
istaskfailed(t::Task) -> Bool
```

Determine whether a task has exited because an exception was thrown.

Examples

```
julia> a4() = error("task failed");

julia> b = Task(a4);

julia> istaskfailed(b)
false
```

```
julia> schedule(b);

julia> yield();

julia> istaskfailed(b)
true
```

`Base.task_local_storage` — Method

```
task_local_storage(key)
```

Look up the value of a key in the current task's task-local storage.

`Base.task_local_storage` — Method

```
task_local_storage(key, value)
```

Assign a value to a key in the current task's task-local storage.

`Base.task_local_storage` — Method

```
task_local_storage(body, key, value)
```

Call the function `body` with a modified task-local storage, in which `value` is assigned to `key`; the previous value of `key`, or lack thereof, is restored afterwards. Useful for emulating dynamic scoping.

Scheduling

`Base.yield` — Function

```
yield()
```

Switch to the scheduler to allow another scheduled task to run. A task that calls this function is still runnable, and will be restarted immediately if there are no other runnable tasks.

```
yield(t::Task, arg = nothing)
```

A fast, unfair-scheduling version of `schedule(t, arg); yield()` which immediately yields to `t` before calling the scheduler.

`Base.yieldto` — Function

```
yieldto(t::Task, arg = nothing)
```

Switch to the given task. The first time a task is switched to, the task's function is called with no arguments. On subsequent switches, `arg` is returned from the task's last call to `yieldto`. This is a low-level call that only switches tasks, not considering states or scheduling in any way. Its use is discouraged.

`Base.sleep` — Function

```
sleep(seconds)
```

Block the current task for a specified number of seconds. The minimum sleep time is 1 millisecond or input of `0.001`.

`Base.schedule` — Function

```
schedule(t::Task, [val]; error=false)
```

Add a `Task` to the scheduler's queue. This causes the task to run constantly when the system is otherwise idle, unless the task performs a blocking operation such as `wait`.

If a second argument `val` is provided, it will be passed to the task (via the return value of `yieldto`) when it runs again. If `error` is `true`, the value is raised as an exception in the woken task.

Examples

```
julia> a5() = sum(i for i in 1:1000);

julia> b = Task(a5);

julia> istaskstarted(b)
false

julia> schedule(b);

julia> yield();

julia> istaskstarted(b)
true

julia> istaskdone(b)
true
```

Synchronization

`Base.@sync` — Macro

```
@sync
```

Wait until all lexically-enclosed uses of `@async`, `@spawn`, `@spawnat` and `@distributed` are complete. All exceptions thrown by enclosed async operations are collected and thrown as a `CompositeException`.

`Base.wait` — Function

```
wait(r::Future)
```

Wait for a value to become available for the specified `Future`.

```
wait(r::RemoteChannel, args...)
```

Wait for a value to become available on the specified `RemoteChannel`.

Special note for `Threads.Condition`:

The caller must be holding the `lock` that owns `c` before calling this method. The calling task will be blocked until some other task wakes it, usually by calling `notify` on the same Condition object. The lock will be atomically released when blocking (even if it was locked recursively), and will be reacquired before returning.

```
wait([x])
```

Block the current task until some event occurs, depending on the type of the argument:

- `Channel`: Wait for a value to be appended to the channel.
- `Condition`: Wait for `notify` on a condition.
- `Process`: Wait for a process or process chain to exit. The `exitcode` field of a process can be used to determine success or failure.
- `Task`: Wait for a `Task` to finish. If the task fails with an exception, a `TaskFailedException` (which wraps the failed task) is thrown.
- `RawFD`: Wait for changes on a file descriptor (see the `FileWatching` package).

If no argument is passed, the task blocks for an undefined period. A task can only be restarted by an explicit call to `schedule` or `yieldto`.

Often `wait` is called within a `while` loop to ensure a waited-for condition is met before proceeding.

`Base.fetch` — Method

```
fetch(t::Task)
```

Wait for a Task to finish, then return its result value. If the task fails with an exception, a `TaskFailedException` (which wraps the failed task) is thrown.

`Base.timedwait` — Function

```
timedwait(testcb::Function, timeout::Real; pollint::Real=0.1)
```

Waits until `testcb` returns `true` or for `timeout` seconds, whichever is earlier. `testcb` is polled every `pollint` seconds. The minimum duration for `timeout` and `pollint` is 1 millisecond or `0.001`.

Returns :ok or :timed_out

`Base.Condition` — Type

```
Condition()
```

Create an edge-triggered event source that tasks can wait for. Tasks that call `wait` on a `Condition` are suspended and queued. Tasks are woken up when `notify` is later called on the `Condition`. Edge triggering means that only tasks waiting at the time `notify` is called can be woken up. For level-triggered notifications, you must keep extra state to keep track of whether a notification has happened. The `Channel` and `Threads.Event` types do this, and can be used for level-triggered events.

This object is NOT thread-safe. See `Threads.Condition` for a thread-safe version.

`Base.notify` — Function

```
notify(condition, val=nothing; all=true, error=false)
```

Wake up tasks waiting for a condition, passing them `val`. If `all` is `true` (the default), all waiting tasks are woken, otherwise only one is. If `error` is `true`, the passed value is raised as an exception in the woken tasks.

Return the count of tasks woken up. Return 0 if no tasks are waiting on `condition`.

`Base.Semaphore` — Type

```
Semaphore(sem_size)
```

Create a counting semaphore that allows at most `sem_size` acquires to be in use at any time. Each acquire must be matched with a release.

`Base.acquire` — Function

```
acquire(s::Semaphore)
```

Wait for one of the `sem_size` permits to be available, blocking until one can be acquired.

`Base.release` — Function

```
release(s::Semaphore)
```

Return one permit to the pool, possibly allowing another task to acquire it and resume execution.

`Base.AbstractLock` — Type

```
AbstractLock
```

Abstract supertype describing types that implement the synchronization primitives: `lock`, `trylock`, `unlock`, and `islocked`.

`Base.lock` — Function

```
lock(lock)
```

Acquire the `lock` when it becomes available. If the lock is already locked by a different task/thread, wait for it to become available.

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

```
lock(f::Function, lock)
```

Acquire the `lock`, execute `f` with the `lock` held, and release the `lock` when `f` returns. If the lock is already locked by a different task/thread, wait for it to become available.

When this function returns, the `lock` has been released, so the caller should not attempt to `unlock` it.

`Base.unlock` — Function

```
unlock(lock)
```

Releases ownership of the `lock`.

If this is a recursive lock which has been acquired before, decrement an internal counter and return immediately.

`Base.trylock` — Function

```
trylock(lock) -> Success (Boolean)
```

Acquire the lock if it is available, and return `true` if successful. If the lock is already locked by a different task/thread, return `false`.

Each successful `trylock` must be matched by an `unlock`.

`Base.islocked` — Function

```
islocked(lock) -> Status (Boolean)
```

Check whether the `lock` is held by any task/thread. This should not be used for synchronization (see instead `trylock`).

`Base.ReentrantLock` — Type

```
ReentrantLock()
```

Creates a re-entrant lock for synchronizing `Task`s. The same task can acquire the lock as many times as required. Each `lock` must be matched with an `unlock`.

Calling 'lock' will also inhibit running of finalizers on that thread until the corresponding 'unlock'. Use of the standard lock pattern illustrated below should naturally be supported, but beware of inverting the try/lock order or missing the try block entirely (e.g. attempting to return with the lock still held):

```
lock(l)
try
    <atomic work>
finally
    unlock(l)
end
```

Channels

`Base.Channel` — Type

```
Channel{T=Any}(size::Int=0)
```

Constructs a `Channel` with an internal buffer that can hold a maximum of `size` objects of type `T`. `put!` calls on a full channel block until an object is removed with `take!`.

`Channel(0)` constructs an unbuffered channel. `put!` blocks until a matching `take!` is called. And vice-versa.

Other constructors:

- `Channel()`: default constructor, equivalent to `Channel{Any}(0)`
- `Channel(Inf)`: equivalent to `Channel{Any}(typemax(Int))`
- `Channel(sz)`: equivalent to `Channel{Any}(sz)`

> **Julia 1.3**
>
> The default constructor `Channel()` and default `size=0` were added in Julia 1.3.

`Base.Channel` — Method

```
Channel{T=Any}(func::Function, size=0; taskref=nothing, spawn=false)
```

Create a new task from `func`, bind it to a new channel of type `T` and size `size`, and schedule the task, all in a single call.

`func` must accept the bound channel as its only argument.

If you need a reference to the created task, pass a `Ref{Task}` object via the keyword argument `taskref`.

If `spawn = true`, the Task created for `func` may be scheduled on another thread in parallel, equivalent to creating a task via `Threads.@spawn`.

Return a `Channel`.

Examples

```
julia> chnl = Channel() do ch
           foreach(i -> put!(ch, i), 1:4)
       end;

julia> typeof(chnl)
Channel{Any}

julia> for i in chnl
           @show i
       end;
i = 1
i = 2
i = 3
i = 4
```

Referencing the created task:

```
julia> taskref = Ref{Task}();

julia> chnl = Channel(taskref=taskref) do ch
           println(take!(ch))
       end;
```

```
julia> istaskdone(taskref[])
false

julia> put!(chnl, "Hello");
Hello

julia> istaskdone(taskref[])
true
```

Julia 1.3

The `spawn=` parameter was added in Julia 1.3. This constructor was added in Julia 1.3. In earlier versions of Julia, Channel used keyword arguments to set `size` and `T`, but those constructors are deprecated.

```
julia> chnl = Channel{Char}(1, spawn=true) do ch
           for c in "hello world"
               put!(ch, c)
           end
       end
Channel{Char}(sz_max:1,sz_curr:1)

julia> String(collect(chnl))
"hello world"
```

`Base.put!` — Method

```
put!(c::Channel, v)
```

Append an item `v` to the channel `c`. Blocks if the channel is full.

For unbuffered channels, blocks until a `take!` is performed by a different task.

Julia 1.1

`v` now gets converted to the channel's type with `convert` as `put!` is called.

`Base.take!` — Method

```
take!(c::Channel)
```

Remove and return a value from a `Channel`. Blocks until data is available.

For unbuffered channels, blocks until a `put!` is performed by a different task.

`Base.isready` — Method

```
isready(c::Channel)
```

Determine whether a `Channel` has a value stored to it. Returns immediately, does not block.

For unbuffered channels returns `true` if there are tasks waiting on a `put!`.

`Base.fetch` — Method

```
fetch(c::Channel)
```

Wait for and get the first available item from the channel. Does not remove the item. `fetch` is unsupported on an unbuffered (0-size) channel.

`Base.close` — Method

```
close(c::Channel[, excp::Exception])
```

Close a channel. An exception (optionally given by `excp`), is thrown by:

- `put!` on a closed channel.
- `take!` and `fetch` on an empty, closed channel.

`Base.bind` — Method

```
bind(chnl::Channel, task::Task)
```

Associate the lifetime of `chnl` with a task. `Channel` `chnl` is automatically closed when the task terminates. Any uncaught exception in the task is propagated to all waiters on `chnl`.

The `chnl` object can be explicitly closed independent of task termination. Terminating tasks have no effect on already closed `Channel` objects.

When a channel is bound to multiple tasks, the first task to terminate will close the channel. When multiple channels are bound to the same task, termination of the task will close all of the bound channels.

Examples

```
julia> c = Channel(0);

julia> task = @async foreach(i->put!(c, i), 1:4);

julia> bind(c,task);

julia> for i in c
           @show i
       end;
i = 1
i = 2
i = 3
i = 4

julia> isopen(c)
false
```

```
julia> c = Channel(0);

julia> task = @async (put!(c, 1); error("foo"));

julia> bind(c, task);

julia> take!(c)
1

julia> put!(c, 1);
ERROR: TaskFailedException:
```

```
foo
Stacktrace:
[...]
```

« Arrays Multi-Threading »

Powered by Documenter.jl and the Julia Programming Language.