NAME

Thread - Manipulate threads in Perl (for old code only)

DEPRECATED

The `Thread` module served as the frontend to the old-style thread model, called *5005threads*, that was introduced in release 5.005. That model was deprecated, and has been removed in version 5.10.

For old code and interim backwards compatibility, the `Thread` module has been reworked to function as a frontend for the new interpreter threads (*ithreads*) model. However, some previous functionality is not available. Further, the data sharing models between the two thread models are completely different, and anything to do with data sharing has to be thought differently. With *ithreads*, you must explicitly `share()` variables between the threads.

You are strongly encouraged to migrate any existing threaded code to the new model (i.e., use the `threads` and `threads::shared` modules) as soon as possible.

HISTORY

In Perl 5.005, the thread model was that all data is implicitly shared, and shared access to data has to be explicitly synchronized. This model is called *5005threads*.

In Perl 5.6, a new model was introduced in which all is was thread local and shared access to data has to be explicitly declared. This model is called *ithreads*, for "interpreter threads".

In Perl 5.6, the *ithreads* model was not available as a public API; only as an internal API that was available for extension writers, and to implement fork() emulation on Win32 platforms.

In Perl 5.8, the *ithreads* model became available through the `threads` module, and the *5005threads* model was deprecated.

In Perl 5.10, the *5005threads* model was removed from the Perl interpreter.

SYNOPSIS

```
use Thread qw(:DEFAULT async yield);

my $t = Thread->new(\&start_sub, @start_args);

$result = $t->join;
$t->detach;

if ($t->done) {
    $t->join;
}

if($t->equal($another_thread)) {
    # ...
}

yield();

my $tid = Thread->self->tid;

lock($scalar);
lock(@array);
lock(%hash);
```

```
my @list = Thread->list;
```

DESCRIPTION

The Thread module provides multithreading support for Perl.

FUNCTIONS

$thread = Thread->new(\&start_sub)

$thread = Thread->new(\&start_sub, LIST)

new starts a new thread of execution in the referenced subroutine. The optional list is passed as parameters to the subroutine. Execution continues in both the subroutine and the code after the new call.

Thread->new returns a thread object representing the newly created thread.

lock VARIABLE

lock places a lock on a variable until the lock goes out of scope.

If the variable is locked by another thread, the lock call will block until it's available. lock is recursive, so multiple calls to lock are safe--the variable will remain locked until the outermost lock on the variable goes out of scope.

Locks on variables only affect lock calls--they do *not* affect normal access to a variable. (Locks on subs are different, and covered in a bit.) If you really, *really* want locks to block access, then go ahead and tie them to something and manage this yourself. This is done on purpose. While managing access to variables is a good thing, Perl doesn't force you out of its living room...

If a container object, such as a hash or array, is locked, all the elements of that container are not locked. For example, if a thread does a lock @a, any other thread doing a lock($a[12]) won't block.

Finally, lock will traverse up references exactly *one* level. lock(\$a) is equivalent to lock($a), while lock(\\$a) is not.

async BLOCK;

async creates a thread to execute the block immediately following it. This block is treated as an anonymous sub, and so must have a semi-colon after the closing brace. Like Thread->new, async returns a thread object.

Thread->self

The Thread->self function returns a thread object that represents the thread making the Thread->self call.

Thread->list

Returns a list of all non-joined, non-detached Thread objects.

cond_wait VARIABLE

The cond_wait function takes a **locked** variable as a parameter, unlocks the variable, and blocks until another thread does a cond_signal or cond_broadcast for that same locked variable. The variable that cond_wait blocked on is relocked after the cond_wait is satisfied. If there are multiple threads cond_waiting on the same variable, all but one will reblock waiting to reaquire the lock on the variable. (So if you're only using cond_wait for synchronization, give up the lock as soon as possible.)

cond_signal VARIABLE

The cond_signal function takes a locked variable as a parameter and unblocks one thread that's cond_waiting on that variable. If more than one thread is blocked in a

`cond_wait` on that variable, only one (and which one is indeterminate) will be unblocked.

If there are no threads blocked in a `cond_wait` on the variable, the signal is discarded.

cond_broadcast VARIABLE

The `cond_broadcast` function works similarly to `cond_signal`. `cond_broadcast`, though, will unblock **all** the threads that are blocked in a `cond_wait` on the locked variable, rather than only one.

yield

The `yield` function allows another thread to take control of the CPU. The exact results are implementation-dependent.

METHODS

join

`join` waits for a thread to end and returns any values the thread exited with. `join` will block until the thread has ended, though it won't block if the thread has already terminated.

If the thread being `join`ed `died`, the error it died with will be returned at this time. If you don't want the thread performing the `join` to die as well, you should either wrap the `join` in an `eval` or use the `eval` thread method instead of `join`.

detach

`detach` tells a thread that it is never going to be joined i.e. that all traces of its existence can be removed once it stops running. Errors in detached threads will not be visible anywhere - if you want to catch them, you should use $SIG{__DIE__} or something like that.

equal

`equal` tests whether two thread objects represent the same thread and returns true if they do.

tid

The `tid` method returns the tid of a thread. The tid is a monotonically increasing integer assigned when a thread is created. The main thread of a program will have a tid of zero, while subsequent threads will have tids assigned starting with one.

done

The `done` method returns true if the thread you're checking has finished, and false otherwise.

DEFUNCT

The following were implemented with *5005threads*, but are no longer available with *ithreads*.

lock(\&sub)

With 5005threads, you could also `lock` a sub such that any calls to that sub from another thread would block until the lock was released.

Also, subroutines could be declared with the `:locked` attribute which would serialize access to the subroutine, but allowed different threads non-simultaneous access.

eval

The `eval` method wrapped an `eval` around a `join`, and so waited for a thread to exit, passing along any values the thread might have returned and placing any errors into $@.

flags

The `flags` method returned the flags for the thread - an integer value corresponding to the internal flags for the thread.

SEE ALSO

threads, *threads::shared*, *Thread::Queue*, *Thread::Semaphore*