NAME

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

CAVEAT

Perl has two thread models.

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.

Neither model is configured by default into Perl (except, as mentioned above, in Win32 ithreads are always available.) You can see your Perl's threading configuration by running `perl -V` and looking for the *use...threads* variables, or inside script by `use Config;` and testing for `$Config{use5005threads}` and `$Config{useithreads}`.

For old code and interim backwards compatibility, the Thread module has been reworked to function as a frontend for both 5005threads and ithreads.

Note that the compatibility is not complete: because the data sharing models are directly opposed, anything to do with data sharing has to be thought differently. With the ithreads you must explicitly share() variables between the threads.

For new code the use of the `Thread` module is discouraged and the direct use of the `threads` and `threads::shared` modules is encouraged instead.

Finally, note that there are many known serious problems with the 5005threads, one of the least of which is that regular expression match variables like $1 are not threadsafe, that is, they easily get corrupted by competing threads. Other problems include more insidious data corruption and mysterious crashes. You are seriously urged to use ithreads instead.

SYNOPSIS

```
use Thread;

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

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

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

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

yield();

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

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

lock(\&sub); # not available with ithreads

$flags = $t->flags; # not available with ithreads

my @list = Thread->list; # not available with ithreads

use Thread 'async';
```

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.

With 5005threads you may also lock a sub, using lock &sub. Any calls to that sub from another thread will block until the lock is released. This behaviour is not equivalent to declaring the sub with the locked attribute. The locked attribute serializes access to a subroutine, but allows different threads non-simultaneous access. lock &sub, on the other hand, will not allow *any* other thread access for the duration of the lock.

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.

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_wait`ing 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_wait`ing 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 `die`d, 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`.

eval

The `eval` method wraps an `eval` around a `join`, and so waits for a thread to exit, passing along any values the thread might have returned. Errors, of course, get placed into $@. (Not available with ithreads.)

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.

flags

The `flags` method returns the flags for the thread. This is the integer value corresponding to the internal flags for the thread, and the value may not be all that meaningful to you. (Not available with ithreads.)

done

The `done` method returns true if the thread you're checking has finished, and false otherwise. (Not available with ithreads.)

LIMITATIONS

The sequence number used to assign tids is a simple integer, and no checking is done to make sure the tid isn't currently in use. If a program creates more than 2**32 - 1 threads in a single run, threads may be assigned duplicate tids. This limitation may be lifted in a future version of Perl.

SEE ALSO

threads::shared (not available with 5005threads)

attributes, *Thread::Queue*, *Thread::Semaphore*, *Thread::Specific* (not available with ithreads)