NAME

threads::shared - Perl extension for sharing data structures between threads

SYNOPSIS

```
use threads;
use threads::shared;

my $var : shared;
$var = $scalar_value;
$var = $shared_ref_value;
$var = &share($simple_unshared_ref_value);
$var = &share(new Foo);

my($scalar, @array, %hash);
share($scalar);
share(@array);
share(%hash);
my $bar = &share([]);
$hash{bar} = &share({});

{ lock(%hash); ...  }

cond_wait($scalar);
cond_timedwait($scalar, time() + 30);
cond_broadcast(@array);
cond_signal(%hash);

my $lockvar : shared;
# condition var != lock var
cond_wait($var, $lockvar);
cond_timedwait($var, time()+30, $lockvar);
```

DESCRIPTION

By default, variables are private to each thread, and each newly created thread gets a private copy of each existing variable. This module allows you to share variables across different threads (and pseudoforks on Win32). It is used together with the threads module.

EXPORT

`share`, `cond_wait`, `cond_timedwait`, `cond_signal`, `cond_broadcast`

Note that if this module is imported when `threads` has not yet been loaded, then these functions all become no-ops. This makes it possible to write modules that will work in both threaded and non-threaded environments.

FUNCTIONS

share VARIABLE

`share` takes a value and marks it as shared. You can share a scalar, array, hash, scalar ref, array ref or hash ref. `share` will return the shared rvalue but always as a reference.

`share` will traverse up references exactly *one* level. `share(\$a)` is equivalent to `share($a)`, while `share(\\$a)` is not. This means that you must create nested shared data structures by first creating individual shared leaf notes, then adding them to a shared hash or array.

A variable can also be marked as shared at compile time by using the `shared` attribute: `my`

`$var : shared`.

If you want to share a newly created reference unfortunately you need to use `&share([])` and `&share({})` syntax due to problems with Perl's prototyping.

The only values that can be assigned to a shared scalar are other scalar values, or shared refs, eg

```
    my $var : shared;
    $var = 1;              # ok
    $var = &share([]);     # ok
    $var = [];             # error
    $var = A->new;         # error
    $var = &share(A->new); # ok as long as the A object is not nested
```

Note that it is often not wise to share an object unless the class itself has been written to support sharing; for example, an object's destructor may get called multiple times, one for each thread's scope exit.

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.

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.

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

Note that you cannot explicitly unlock a variable; you can only wait for the lock to go out of scope. If you need more fine-grained control, see *Thread::Semaphore*.

cond_wait VARIABLE

cond_wait CONDVAR, LOCKVAR

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 reacquire the lock on the variable. (So if you're only using `cond_wait` for synchronisation, give up the lock as soon as possible). The two actions of unlocking the variable and entering the blocked wait state are atomic, the two actions of exiting from the blocked wait state and relocking the variable are not.

In its second form, `cond_wait` takes a shared, **unlocked** variable followed by a shared, **locked** variable. The second variable is unlocked and thread execution suspended until another thread signals the first variable.

It is important to note that the variable can be notified even if no thread `cond_signal` or `cond_broadcast` on the variable. It is therefore important to check the value of the variable and go back to waiting if the requirement is not fulfilled. For example, to pause until a shared counter drops to zero:

```
    { lock($counter); cond_wait($count) until $counter == 0; }
```

cond_timedwait VARIABLE, ABS_TIMEOUT

cond_timedwait CONDVAR, ABS_TIMEOUT, LOCKVAR

In its two-argument form, `cond_timedwait` takes a **locked** variable and an absolute timeout as parameters, unlocks the variable, and blocks until the timeout is reached or another thread signals the variable. A false value is returned if the timeout is reached, and a true value otherwise. In either case, the variable is re-locked upon return.

Like `cond_wait`, this function may take a shared, **locked** variable as an additional parameter; in this case the first parameter is an **unlocked** condition variable protected by a distinct lock variable.

Again like `cond_wait`, waking up and reacquiring the lock are not atomic, and you should always check your desired condition after this function returns. Since the timeout is an absolute value, however, it does not have to be recalculated with each pass:

```
lock($var);
my $abs = time() + 15;
until ($ok = desired_condition($var)) {
  last if !cond_timedwait($var, $abs);
}
# we got it if $ok, otherwise we timed out!
```

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. By always locking before signaling, you can (with care), avoid signaling before another thread has entered cond_wait().

`cond_signal` will normally generate a warning if you attempt to use it on an unlocked variable. On the rare occasions where doing this may be sensible, you can skip the warning with

```
{ no warnings 'threads'; cond_signal($foo) }
```

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.

NOTES

threads::shared is designed to disable itself silently if threads are not available. If you want access to threads, you must `use threads` before you `use threads::shared`. threads will emit a warning if you use it after threads::shared.

BUGS

`bless` is not supported on shared references. In the current version, `bless` will only bless the thread local reference and the blessing will not propagate to the other threads. This is expected to be implemented in a future version of Perl.

Does not support splice on arrays!

Taking references to the elements of shared arrays and hashes does not autovivify the elements, and neither does slicing a shared array/hash over non-existent indices/keys autovivify the elements.

share() allows you to `share $hashref->{key}` without giving any error message. But the `$hashref->{key}` is **not** shared, causing the error "locking can only be used on shared values" to occur when you attempt to `lock $hasref->{key}`.

AUTHOR

Arthur Bergman <arthur at contiller.se>

Documentation borrowed from the old Thread.pm

SEE ALSO

threads, *perlthrtut*, *http://www.perl.com/pub/a/2002/06/11/threads.html*