NAME

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

VERSION

This document describes threads::shared version 1.14

SYNOPSIS

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


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


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 pseudo-forks on Win32). It is used together with the *threads* module.

EXPORT

share, cond_wait, cond_timedwait, cond_signal, cond_broadcast, is_shared

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.

A variable can also be marked as shared at compile time by using the :shared attribute: my $var :shared;.

Due to problems with Perl's prototyping, if you want to share a newly created reference, you

need to use the `&share([])` and `&share({})` syntax.

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

```
my $var :shared;
$var = 1;                # ok
$var = [];               # error
$var = &share([]);       # ok
```

`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 nodes, and then adding them to a shared hash or array.

```
my %hash :shared;
$hash{'meaning'} = &share([]);
$hash{'meaning'}[0] = &share({});
$hash{'meaning'}[0]{'life'} = 42;
```

is_shared VARIABLE

> `is_shared` checks if the specified variable is shared or not. If shared, returns the variable's internal ID (similar to *refaddr()*). Otherwise, returns `undef`.
>
> ```
> if (is_shared($var)) {
>     print("\$var is shared\n");
> } else {
>     print("\$var is not shared\n");
> }
> ```

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. Multiple calls to `lock` by the same thread from within dynamically nested scopes are safe -- the variable will remain locked until the outermost lock on the variable goes out of scope.
>
> Locking a container object, such as a hash or array, doesn't lock the elements of that container. For example, if a thread does a `lock(@a)`, any other thread doing a `lock($a[12])` won't block.
>
> `lock()` follows references exactly *one* level. `lock(\$a)` is equivalent to `lock($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. This is most easily accomplished by locking the variable inside a block.
>
> ```
> my $var :shared;
> {
>     lock($var);
>     # $var is locked from here to the end of the block
>     ...
> }
> # $var is now unlocked
> ```
>
> If you need more fine-grained control over shared variable access, 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 re-block 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 re-locking 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 suppress 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.

OBJECTS

threads::shared exports a version of *bless()* that works on shared objects such that *blessings* propagate across threads.

```
# Create a shared 'foo' object
my $foo;
share($foo);
$foo = &share({});
bless($foo, 'foo');


# Create a shared 'bar' object
my $bar;
share($bar);
$bar = &share({});
bless($bar, 'bar');


# Put 'bar' inside 'foo'
$foo->{'bar'} = $bar;


# Rebless the objects via a thread
threads->create(sub {
    # Rebless the outer object
    bless($foo, 'yin');


    # Cannot directly rebless the inner object
    #bless($foo->{'bar'}, 'yang');


    # Retrieve and rebless the inner object
    my $obj = $foo->{'bar'};
    bless($obj, 'yang');
    $foo->{'bar'} = $obj;


})->join();


print(ref($foo),          "\n");    # Prints 'yin'
print(ref($foo->{'bar'}), "\n");    # Prints 'yang'
print(ref($bar),          "\n");    # Also prints 'yang'
```

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 AND LIMITATIONS

When `share` is used on arrays, hashes, array refs or hash refs, any data they contain will be lost.

```
my @arr = qw(foo bar baz);
share(@arr);
# @arr is now empty (i.e., == ());


# Create a 'foo' object
my $foo = { 'data' => 99 };
bless($foo, 'foo');
```

```
    # Share the object
    share($foo);        # Contents are now wiped out
    print("ERROR: \$foo is empty\n")
        if (! exists($foo->{'data'}));
```

Therefore, populate such variables **after** declaring them as shared. (Scalar and scalar refs are not affected by this problem.)

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, once for each thread's scope exit. Another danger is that the contents of hash-based objects will be lost due to the above mentioned limitation. See *examples/class.pl* (in the CPAN distribution of this module) for how to create a class that supports object sharing.

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})`.

View existing bug reports at, and submit any new bugs, problems, patches, etc. to:
http://rt.cpan.org/NoAuth/Bugs.html?Dist=threads-shared

SEE ALSO

threads::shared Discussion Forum on CPAN: *http://www.cpanforum.com/dist/threads-shared*

Annotated POD for *threads::shared*:
http://annocpan.org/~JDHEDDEN/threads-shared-1.14/shared.pm

Source repository: *http://code.google.com/p/threads-shared/*

threads, *perlthrtut*

http://www.perl.com/pub/a/2002/06/11/threads.html and
http://www.perl.com/pub/a/2002/09/04/threads.html

Perl threads mailing list: *http://lists.cpan.org/showlist.cgi?name=iThreads*

AUTHOR

Artur Bergman <sky AT crucially DOT net>

threads::shared is released under the same license as Perl.

Documentation borrowed from the old Thread.pm.

CPAN version produced by Jerry D. Hedden <jdhedden AT cpan DOT org>.