NAME

threads - Perl interpreter-based threads

VERSION

This document describes threads version 1.86

SYNOPSIS

```
use threads ('yield',
             'stack_size' => 64*4096,
             'exit' => 'threads_only',
             'stringify');

sub start_thread {
    my @args = @_;
    print('Thread started: ', join(' ', @args), "\n");
}
my $thr = threads->create('start_thread', 'argument');
$thr->join();

threads->create(sub { print("I am a thread\n"); })->join();

my $thr2 = async { foreach (@files) { ... } };
$thr2->join();
if (my $err = $thr2->error()) {
    warn("Thread error: $err\n");
}

# Invoke thread in list context (implicit) so it can return a list
my ($thr) = threads->create(sub { return (qw/a b c/); });
# or specify list context explicitly
my $thr = threads->create({'context' => 'list'},
                          sub { return (qw/a b c/); });
my @results = $thr->join();

$thr->detach();

# Get a thread's object
$thr = threads->self();
$thr = threads->object($tid);

# Get a thread's ID
$tid = threads->tid();
$tid = $thr->tid();
$tid = "$thr";

# Give other threads a chance to run
threads->yield();
yield();

# Lists of non-detached threads
my @threads = threads->list();
my $thread_count = threads->list();
```

```
my @running = threads->list(threads::running);
my @joinable = threads->list(threads::joinable);

# Test thread objects
if ($thr1 == $thr2) {
    ...
}

# Manage thread stack size
$stack_size = threads->get_stack_size();
$old_size = threads->set_stack_size(32*4096);

# Create a thread with a specific context and stack size
my $thr = threads->create({ 'context'    => 'list',
                            'stack_size' => 32*4096,
                            'exit'       => 'thread_only' },
                          \&foo);

# Get thread's context
my $wantarray = $thr->wantarray();

# Check thread's state
if ($thr->is_running()) {
    sleep(1);
}
if ($thr->is_joinable()) {
    $thr->join();
}

# Send a signal to a thread
$thr->kill('SIGUSR1');

# Exit a thread
threads->exit();
```

DESCRIPTION

Since Perl 5.8, thread programming has been available using a model called *interpreter threads* which provides a new Perl interpreter for each thread, and, by default, results in no data or state information being shared between threads.

(Prior to Perl 5.8, *5005threads* was available through the `Thread.pm` API. This threading model has been deprecated, and was removed as of Perl 5.10.0.)

As just mentioned, all variables are, by default, thread local. To use shared variables, you need to also load *threads::shared*:

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

When loading *threads::shared*, you must `use threads` before you `use threads::shared`. (`threads` will emit a warning if you do it the other way around.)

It is strongly recommended that you enable threads via `use threads` as early as possible in your script.

If needed, scripts can be written so as to run on both threaded and non-threaded Perls:

```
my $can_use_threads = eval 'use threads; 1';
if ($can_use_threads) {
    # Do processing using threads
    ...
} else {
    # Do it without using threads
    ...
}
```

$thr = threads->create(FUNCTION, ARGS)

This will create a new thread that will begin execution with the specified entry point function, and give it the *ARGS* list as parameters. It will return the corresponding threads object, or `undef` if thread creation failed.

FUNCTION may either be the name of a function, an anonymous subroutine, or a code ref.

```
my $thr = threads->create('func_name', ...);
    # or
my $thr = threads->create(sub { ... }, ...);
    # or
my $thr = threads->create(\&func, ...);
```

The `->new()` method is an alias for `->create()`.

$thr->join()

This will wait for the corresponding thread to complete its execution. When the thread finishes, `->join()` will return the return value(s) of the entry point function.

The context (void, scalar or list) for the return value(s) for `->join()` is determined at the time of thread creation.

```
# Create thread in list context (implicit)
my ($thr1) = threads->create(sub {
                                 my @results = qw(a b c);
                                 return (@results);
                             });
#   or (explicit)
my $thr1 = threads->create({'context' => 'list'},
                           sub {
                                 my @results = qw(a b c);
                                 return (@results);
                           });
# Retrieve list results from thread
my @res1 = $thr1->join();

# Create thread in scalar context (implicit)
my $thr2 = threads->create(sub {
                                 my $result = 42;
                                 return ($result);
                             });
# Retrieve scalar result from thread
my $res2 = $thr2->join();

# Create a thread in void context (explicit)
my $thr3 = threads->create({'void' => 1},
                           sub { print("Hello, world\n"); });
```

```
    # Join the thread in void context (i.e., no return value)
    $thr3->join();
```

See *THREAD CONTEXT* for more details.

If the program exits without all threads having either been joined or detached, then a warning will be issued.

Calling `->join()` or `->detach()` on an already joined thread will cause an error to be thrown.

$thr->detach()

Makes the thread unjoinable, and causes any eventual return value to be discarded. When the program exits, any detached threads that are still running are silently terminated.

If the program exits without all threads having either been joined or detached, then a warning will be issued.

Calling `->join()` or `->detach()` on an already detached thread will cause an error to be thrown.

threads->detach()

Class method that allows a thread to detach itself.

threads->self()

Class method that allows a thread to obtain its own *threads* object.

$thr->tid()

Returns the ID of the thread. Thread IDs are unique integers with the main thread in a program being 0, and incrementing by 1 for every thread created.

threads->tid()

Class method that allows a thread to obtain its own ID.

"$thr"

If you add the `stringify` import option to your `use threads` declaration, then using a threads object in a string or a string context (e.g., as a hash key) will cause its ID to be used as the value:

```
    use threads qw(stringify);

    my $thr = threads->create(...);
    print("Thread $thr started...\n");  # Prints out: Thread 1
started...
```

threads->object($tid)

This will return the *threads* object for the *active* thread associated with the specified thread ID. If `$tid` is the value for the current thread, then this call works the same as `->self()`. Otherwise, returns `undef` if there is no thread associated with the TID, if the thread is joined or detached, if no TID is specified or if the specified TID is undef.

threads->yield()

This is a suggestion to the OS to let this thread yield CPU time to other threads. What actually happens is highly dependent upon the underlying thread implementation.

You may do `use threads qw(yield)`, and then just use `yield()` in your code.

threads->list()

threads->list(threads::all)

threads->list(threads::running)

threads->list(threads::joinable)

With no arguments (or using `threads::all`) and in a list context, returns a list of all non-joined, non-detached *threads* objects. In a scalar context, returns a count of the same.

With a *true* argument (using `threads::running`), returns a list of all non-joined, non-detached *threads* objects that are still running.

With a *false* argument (using `threads::joinable`), returns a list of all non-joined, non-detached *threads* objects that have finished running (i.e., for which `->join()` will not *block*).

$thr1->equal($thr2)

Tests if two threads objects are the same thread or not. This is overloaded to the more natural forms:

```
if ($thr1 == $thr2) {
    print("Threads are the same\n");
}
# or
if ($thr1 != $thr2) {
    print("Threads differ\n");
}
```

(Thread comparison is based on thread IDs.)

async BLOCK;

`async` creates a thread to execute the block immediately following it. This block is treated as an anonymous subroutine, and so must have a semicolon after the closing brace. Like `threads->create()`, `async` returns a *threads* object.

$thr->error()

Threads are executed in an `eval` context. This method will return `undef` if the thread terminates *normally*. Otherwise, it returns the value of `$@` associated with the thread's execution status in its `eval` context.

$thr->_handle()

This *private* method returns the memory location of the internal thread structure associated with a threads object. For Win32, this is a pointer to the `HANDLE` value returned by `CreateThread` (i.e., `HANDLE *`); for other platforms, it is a pointer to the `pthread_t` structure used in the `pthread_create` call (i.e., `pthread_t *`).

This method is of no use for general Perl threads programming. Its intent is to provide other (XS-based) thread modules with the capability to access, and possibly manipulate, the underlying thread structure associated with a Perl thread.

threads->_handle()

Class method that allows a thread to obtain its own *handle*.

EXITING A THREAD

The usual method for terminating a thread is to *return()* from the entry point function with the appropriate return value(s).

threads->exit()

If needed, a thread can be exited at any time by calling `threads->exit()`. This will cause the thread to return `undef` in a scalar context, or the empty list in a list context.

When called from the *main* thread, this behaves the same as `exit(0)`.

threads->exit(status)

When called from a thread, this behaves like `threads->exit()` (i.e., the exit status code is ignored).

When called from the *main* thread, this behaves the same as `exit(status)`.

die()

Calling `die()` in a thread indicates an abnormal exit for the thread. Any `$SIG{__DIE__}` handler in the thread will be called first, and then the thread will exit with a warning message that will contain any arguments passed in the `die()` call.

exit(status)

Calling *exit()* inside a thread causes the whole application to terminate. Because of this, the use of `exit()` inside threaded code, or in modules that might be used in threaded applications, is strongly discouraged.

If `exit()` really is needed, then consider using the following:

```
threads->exit() if threads->can('exit');   # Thread friendly
exit(status);
```

use threads 'exit' => 'threads_only'

This globally overrides the default behavior of calling `exit()` inside a thread, and effectively causes such calls to behave the same as `threads->exit()`. In other words, with this setting, calling `exit()` causes only the thread to terminate.

Because of its global effect, this setting should not be used inside modules or the like.

The *main* thread is unaffected by this setting.

threads->create({'exit' => 'thread_only'}, ...)

This overrides the default behavior of `exit()` inside the newly created thread only.

$thr->set_thread_exit_only(boolean)

This can be used to change the *exit thread only* behavior for a thread after it has been created. With a *true* argument, `exit()` will cause only the thread to exit. With a *false* argument, `exit()` will terminate the application.

The *main* thread is unaffected by this call.

threads->set_thread_exit_only(boolean)

Class method for use inside a thread to change its own behavior for `exit()`.

The *main* thread is unaffected by this call.

THREAD STATE

The following boolean methods are useful in determining the *state* of a thread.

$thr->is_running()

Returns true if a thread is still running (i.e., if its entry point function has not yet finished or exited).

$thr->is_joinable()

Returns true if the thread has finished running, is not detached and has not yet been joined. In other words, the thread is ready to be joined, and a call to `$thr->join()` will not *block*.

$thr->is_detached()

Returns true if the thread has been detached.

threads->is_detached()

Class method that allows a thread to determine whether or not it is detached.

THREAD CONTEXT

As with subroutines, the type of value returned from a thread's entry point function may be determined by the thread's *context*: list, scalar or void. The thread's context is determined at thread creation. This is necessary so that the context is available to the entry point function via *wantarray()*. The thread may then specify a value of the appropriate type to be returned from `->join()`.

Explicit context

Because thread creation and thread joining may occur in different contexts, it may be desirable to state the context explicitly to the thread's entry point function. This may be done by calling `->create()` with a hash reference as the first argument:

```
my $thr = threads->create({'context' => 'list'}, \&foo);
...
my @results = $thr->join();
```

In the above, the threads object is returned to the parent thread in scalar context, and the thread's entry point function `foo` will be called in list (array) context such that the parent thread can receive a list (array) from the `->join()` call. (`'array'` is synonymous with `'list'`.)

Similarly, if you need the threads object, but your thread will not be returning a value (i.e., *void* context), you would do the following:

```
my $thr = threads->create({'context' => 'void'}, \&foo);
...
$thr->join();
```

The context type may also be used as the *key* in the hash reference followed by a *true* value:

```
threads->create({'scalar' => 1}, \&foo);
...
my ($thr) = threads->list();
my $result = $thr->join();
```

Implicit context

If not explicitly stated, the thread's context is implied from the context of the `->create()` call:

```
# Create thread in list context
my ($thr) = threads->create(...);

# Create thread in scalar context
my $thr = threads->create(...);

# Create thread in void context
threads->create(...);
```

$thr->wantarray()

This returns the thread's context in the same manner as *wantarray()*.

threads->wantarray()

Class method to return the current thread's context. This returns the same value as running *wantarray()* inside the current thread's entry point function.

THREAD STACK SIZE

The default per-thread stack size for different platforms varies significantly, and is almost always far more than is needed for most applications. On Win32, Perl's makefile explicitly sets the default stack to 16 MB; on most other platforms, the system default is used, which again may be much larger than is needed.

By tuning the stack size to more accurately reflect your application's needs, you may significantly reduce your application's memory usage, and increase the number of simultaneously running threads.

Note that on Windows, address space allocation granularity is 64 KB, therefore, setting the stack smaller than that on Win32 Perl will not save any more memory.

threads->get_stack_size();

Returns the current default per-thread stack size. The default is zero, which means the system default stack size is currently in use.

$size = $thr->get_stack_size();

Returns the stack size for a particular thread. A return value of zero indicates the system default stack size was used for the thread.

$old_size = threads->set_stack_size($new_size);

Sets a new default per-thread stack size, and returns the previous setting.

Some platforms have a minimum thread stack size. Trying to set the stack size below this value will result in a warning, and the minimum stack size will be used.

Some Linux platforms have a maximum stack size. Setting too large of a stack size will cause thread creation to fail.

If needed, `$new_size` will be rounded up to the next multiple of the memory page size (usually 4096 or 8192).

Threads created after the stack size is set will then either call `pthread_attr_setstacksize()` *(for pthreads platforms)*, or supply the stack size to `CreateThread()` *(for Win32 Perl)*.

(Obviously, this call does not affect any currently extant threads.)

use threads ('stack_size' => VALUE);

This sets the default per-thread stack size at the start of the application.

$ENV{'PERL5_ITHREADS_STACK_SIZE'}

The default per-thread stack size may be set at the start of the application through the use of the environment variable `PERL5_ITHREADS_STACK_SIZE`:

```
PERL5_ITHREADS_STACK_SIZE=1048576
export PERL5_ITHREADS_STACK_SIZE
perl -e'use threads; print(threads->get_stack_size(), "\n")'
```

This value overrides any `stack_size` parameter given to `use threads`. Its primary purpose is to permit setting the per-thread stack size for legacy threaded applications.

threads->create({'stack_size' => VALUE}, FUNCTION, ARGS)

To specify a particular stack size for any individual thread, call `->create()` with a hash reference as the first argument:

```
    my $thr = threads->create({'stack_size' => 32*4096}, \&foo,
@args);
```

$thr2 = $thr1->create(FUNCTION, ARGS)

This creates a new thread (`$thr2`) that inherits the stack size from an existing thread (`$thr1`). This is shorthand for the following:

```
    my $stack_size = $thr1->get_stack_size();
    my $thr2 = threads->create({'stack_size' => $stack_size},
FUNCTION, ARGS);
```

THREAD SIGNALLING

When safe signals is in effect (the default behavior - see *Unsafe signals* for more details), then signals may be sent and acted upon by individual threads.

$thr->kill('SIG...');

Sends the specified signal to the thread. Signal names and (positive) signal numbers are the same as those supported by *kill()*. For example, 'SIGTERM', 'TERM' and (depending on the OS) 15 are all valid arguments to `->kill()`.

Returns the thread object to allow for method chaining:

```
    $thr->kill('SIG...')->join();
```

Signal handlers need to be set up in the threads for the signals they are expected to act upon. Here's an example for *cancelling* a thread:

```
use threads;

sub thr_func
{
    # Thread 'cancellation' signal handler
    $SIG{'KILL'} = sub { threads->exit(); };

    ...
}

# Create a thread
my $thr = threads->create('thr_func');

...

# Signal the thread to terminate, and then detach
# it so that it will get cleaned up automatically
$thr->kill('KILL')->detach();
```

Here's another simplistic example that illustrates the use of thread signalling in conjunction with a semaphore to provide rudimentary *suspend* and *resume* capabilities:

```
use threads;
use Thread::Semaphore;

sub thr_func
{
    my $sema = shift;

    # Thread 'suspend/resume' signal handler
    $SIG{'STOP'} = sub {
        $sema->down();      # Thread suspended
```

```
            $sema->up();        # Thread resumes
        };

        ...
    }

    # Create a semaphore and pass it to a thread
    my $sema = Thread::Semaphore->new();
    my $thr = threads->create('thr_func', $sema);

    # Suspend the thread
    $sema->down();
    $thr->kill('STOP');

    ...

    # Allow the thread to continue
    $sema->up();
```

CAVEAT: The thread signalling capability provided by this module does not actually send signals via the OS. It *emulates* signals at the Perl-level such that signal handlers are called in the appropriate thread. For example, sending `$thr->kill('STOP')` does not actually suspend a thread (or the whole process), but does cause a `$SIG{'STOP'}` handler to be called in that thread (as illustrated above).

As such, signals that would normally not be appropriate to use in the `kill()` command (e.g., `kill('KILL', $$)`) are okay to use with the `->kill()` method (again, as illustrated above).

Correspondingly, sending a signal to a thread does not disrupt the operation the thread is currently working on: The signal will be acted upon after the current operation has completed. For instance, if the thread is *stuck* on an I/O call, sending it a signal will not cause the I/O call to be interrupted such that the signal is acted up immediately.

Sending a signal to a terminated thread is ignored.

WARNINGS

Perl exited with active threads:

> If the program exits without all threads having either been joined or detached, then this warning will be issued.
>
> NOTE: If the *main* thread exits, then this warning cannot be suppressed using `no warnings 'threads';` as suggested below.

Thread creation failed: pthread_create returned #

> See the appropriate *man* page for `pthread_create` to determine the actual cause for the failure.

Thread # terminated abnormally: ...

> A thread terminated in some manner other than just returning from its entry point function, or by using `threads->exit()`. For example, the thread may have terminated because of an error, or by using `die`.

Using minimum thread stack size of #

> Some platforms have a minimum thread stack size. Trying to set the stack size below this value will result in the above warning, and the stack size will be set to the minimum.

Thread creation failed: pthread_attr_setstacksize(*SIZE*) returned 22

The specified *SIZE* exceeds the system's maximum stack size. Use a smaller value for the stack size.

If needed, thread warnings can be suppressed by using:

```
no warnings 'threads';
```

in the appropriate scope.

ERRORS

This Perl not built to support threads

The particular copy of Perl that you're trying to use was not built using the `useithreads` configuration option.

Having threads support requires all of Perl and all of the XS modules in the Perl installation to be rebuilt; it is not just a question of adding the *threads* module (i.e., threaded and non-threaded Perls are binary incompatible.)

Cannot change stack size of an existing thread

The stack size of currently extant threads cannot be changed, therefore, the following results in the above error:

```
$thr->set_stack_size($size);
```

Cannot signal threads without safe signals

Safe signals must be in effect to use the `->kill()` signalling method. See *Unsafe signals* for more details.

Unrecognized signal name: ...

The particular copy of Perl that you're trying to use does not support the specified signal being used in a `->kill()` call.

BUGS AND LIMITATIONS

Before you consider posting a bug report, please consult, and possibly post a message to the discussion forum to see if what you've encountered is a known problem.

Thread-safe modules

See *"Making your module threadsafe" in perlmod* when creating modules that may be used in threaded applications, especially if those modules use non-Perl data, or XS code.

Using non-thread-safe modules

Unfortunately, you may encounter Perl modules that are not *thread-safe*. For example, they may crash the Perl interpreter during execution, or may dump core on termination. Depending on the module and the requirements of your application, it may be possible to work around such difficulties.

If the module will only be used inside a thread, you can try loading the module from inside the thread entry point function using `require` (and `import` if needed):

```
sub thr_func
{
    require Unsafe::Module
    # Unsafe::Module->import(...);

    ....
}
```

If the module is needed inside the *main* thread, try modifying your application so that the module is loaded (again using `require` and `->import()`) after any threads are started, and in such a way that no other threads are started afterwards.

If the above does not work, or is not adequate for your application, then file a bug report on *http://rt.cpan.org/Public/* against the problematic module.

Memory consumption

On most systems, frequent and continual creation and destruction of threads can lead to ever-increasing growth in the memory footprint of the Perl interpreter. While it is simple to just launch threads and then `->join()` or `->detach()` them, for long-lived applications, it is better to maintain a pool of threads, and to reuse them for the work needed, using *queues* to notify threads of pending work. The CPAN distribution of this module contains a simple example (*examples/pool_reuse.pl*) illustrating the creation, use and monitoring of a pool of *reusable* threads.

Current working directory

On all platforms except MSWin32, the setting for the current working directory is shared among all threads such that changing it in one thread (e.g., using `chdir()`) will affect all the threads in the application.

On MSWin32, each thread maintains its own the current working directory setting.

Environment variables

Currently, on all platforms except MSWin32, all *system* calls (e.g., using `system()` or back-ticks) made from threads use the environment variable settings from the *main* thread. In other words, changes made to `%ENV` in a thread will not be visible in *system* calls made by that thread.

To work around this, set environment variables as part of the *system* call. For example:

```
my $msg = 'hello';
system("FOO=$msg; echo \$FOO");   # Outputs 'hello' to STDOUT
```

On MSWin32, each thread maintains its own set of environment variables.

Catching signals

Signals are *caught* by the main thread (thread ID = 0) of a script. Therefore, setting up signal handlers in threads for purposes other than *THREAD SIGNALLING* as documented above will not accomplish what is intended.

This is especially true if trying to catch `SIGALRM` in a thread. To handle alarms in threads, set up a signal handler in the main thread, and then use *THREAD SIGNALLING* to relay the signal to the thread:

```
  # Create thread with a task that may time out
  my $thr->create(sub {
      threads->yield();
      eval {
          $SIG{ALRM} = sub { die("Timeout\n"); };
          alarm(10);
          ...  # Do work here
          alarm(0);
      };
      if ($@ =~ /Timeout/) {
          warn("Task in thread timed out\n");
      }
  };

  # Set signal handler to relay SIGALRM to thread
```

```
    $SIG{ALRM} = sub { $thr->kill('ALRM') };

    ... # Main thread continues working
```

Parent-child threads

On some platforms, it might not be possible to destroy *parent* threads while there are still existing *child* threads.

Creating threads inside special blocks

Creating threads inside `BEGIN`, `CHECK` or `INIT` blocks should not be relied upon. Depending on the Perl version and the application code, results may range from success, to (apparently harmless) warnings of leaked scalar, or all the way up to crashing of the Perl interpreter.

Unsafe signals

Since Perl 5.8.0, signals have been made safer in Perl by postponing their handling until the interpreter is in a *safe* state. See *"Safe Signals" in perl58delta* and *"Deferred Signals (Safe Signals)" in perlipc* for more details.

Safe signals is the default behavior, and the old, immediate, unsafe signalling behavior is only in effect in the following situations:

* Perl has been built with `PERL_OLD_SIGNALS` (see `perl -V`).

* The environment variable `PERL_SIGNALS` is set to `unsafe` (see *"PERL_SIGNALS" in perlrun*).

* The module *Perl::Unsafe::Signals* is used.

If unsafe signals is in effect, then signal handling is not thread-safe, and the `->kill()` signalling method cannot be used.

Returning closures from threads

Returning closures from threads should not be relied upon. Depending of the Perl version and the application code, results may range from success, to (apparently harmless) warnings of leaked scalar, or all the way up to crashing of the Perl interpreter.

Returning objects from threads

Returning objects from threads does not work. Depending on the classes involved, you may be able to work around this by returning a serialized version of the object (e.g., using *Data::Dumper* or *Storable*), and then reconstituting it in the joining thread. If you're using Perl 5.10.0 or later, and if the class supports *shared objects*, you can pass them via *shared queues* .

END blocks in threads

It is possible to add *END blocks* to threads by using *require* or *eval* with the appropriate code. These `END` blocks will then be executed when the thread's interpreter is destroyed (i.e., either during a `->join()` call, or at program termination).

However, calling any *threads* methods in such an `END` block will most likely *fail* (e.g., the application may hang, or generate an error) due to mutexes that are needed to control functionality within the *threads* module.

For this reason, the use of `END` blocks in threads is **strongly** discouraged.

Open directory handles

In perl 5.14 and higher, on systems other than Windows that do not support the `fchdir` C function, directory handles (see *opendir*) will not be copied to new threads. You can use the `d_fchdir` variable in *Config.pm* to determine whether your system supports it.

In prior perl versions, spawning threads with open directory handles would crash the

interpreter. *[perl #75154]*

Perl Bugs and the CPAN Version of *threads*

Support for threads extends beyond the code in this module (i.e., *threads.pm* and *threads.xs*), and into the Perl interpreter itself. Older versions of Perl contain bugs that may manifest themselves despite using the latest version of *threads* from CPAN. There is no workaround for this other than upgrading to the latest version of Perl.

Even with the latest version of Perl, it is known that certain constructs with threads may result in warning messages concerning leaked scalars or unreferenced scalars. However, such warnings are harmless, and may safely be ignored.

You can search for *threads* related bug reports at *http://rt.cpan.org/Public/*. If needed submit any new bugs, problems, patches, etc. to: *http://rt.cpan.org/Public/Dist/Display.html?Name=threads*

REQUIREMENTS

Perl 5.8.0 or later

SEE ALSO

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

threads::shared, *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.perl.org/list/ithreads.html*

Stack size discussion: *http://www.perlmonks.org/?node_id=532956*

AUTHOR

Artur Bergman <sky AT crucially DOT net>

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

LICENSE

threads is released under the same license as Perl.

ACKNOWLEDGEMENTS

Richard Soderberg <perl AT crystalflame DOT net> - Helping me out tons, trying to find reasons for races and other weird bugs!

Simon Cozens <simon AT brecon DOT co DOT uk> - Being there to answer zillions of annoying questions

Rocco Caputo <troc AT netrus DOT net>

Vipul Ved Prakash <mail AT vipul DOT net> - Helping with debugging

Dean Arnold <darnold AT presicient DOT com> - Stack size API