NAME

perlthrtut - Tutorial on threads in Perl

DESCRIPTION

This tutorial describes the use of Perl interpreter threads (sometimes referred to as *ithreads*). In this model, each thread runs in its own Perl interpreter, and any data sharing between threads must be explicit. The user-level interface for *ithreads* uses the threads class.

NOTE: There was another older Perl threading flavor called the 5.005 model that used the threads class. This old model was known to have problems, is deprecated, and was removed for release 5.10. You are strongly encouraged to migrate any existing 5.005 threads code to the new model as soon as possible.

You can see which (or neither) threading flavour you have by running `perl -V` and looking at the `Platform` section. If you have `useithreads=define` you have ithreads, if you have `use5005threads=define` you have 5.005 threads. If you have neither, you don't have any thread support built in. If you have both, you are in trouble.

The threads and threads::shared modules are included in the core Perl distribution. Additionally, they are maintained as a separate modules on CPAN, so you can check there for any updates.

What Is A Thread Anyway?

A thread is a flow of control through a program with a single execution point.

Sounds an awful lot like a process, doesn't it? Well, it should. Threads are one of the pieces of a process. Every process has at least one thread and, up until now, every process running Perl had only one thread. With 5.8, though, you can create extra threads. We're going to show you how, when, and why.

Threaded Program Models

There are three basic ways that you can structure a threaded program. Which model you choose depends on what you need your program to do. For many non-trivial threaded programs, you'll need to choose different models for different pieces of your program.

Boss/Worker

The boss/worker model usually has one *boss* thread and one or more *worker* threads. The boss thread gathers or generates tasks that need to be done, then parcels those tasks out to the appropriate worker thread.

This model is common in GUI and server programs, where a main thread waits for some event and then passes that event to the appropriate worker threads for processing. Once the event has been passed on, the boss thread goes back to waiting for another event.

The boss thread does relatively little work. While tasks aren't necessarily performed faster than with any other method, it tends to have the best user-response times.

Work Crew

In the work crew model, several threads are created that do essentially the same thing to different pieces of data. It closely mirrors classical parallel processing and vector processors, where a large array of processors do the exact same thing to many pieces of data.

This model is particularly useful if the system running the program will distribute multiple threads across different processors. It can also be useful in ray tracing or rendering engines, where the individual threads can pass on interim results to give the user visual feedback.

Pipeline

The pipeline model divides up a task into a series of steps, and passes the results of one step on to the thread processing the next. Each thread does one thing to each piece of data and passes the results to the next thread in line.

This model makes the most sense if you have multiple processors so two or more threads will be executing in parallel, though it can often make sense in other contexts as well. It tends to keep the individual tasks small and simple, as well as allowing some parts of the pipeline to block (on I/O or system calls, for example) while other parts keep going. If you're running different parts of the pipeline on different processors you may also take advantage of the caches on each processor.

This model is also handy for a form of recursive programming where, rather than having a subroutine call itself, it instead creates another thread. Prime and Fibonacci generators both map well to this form of the pipeline model. (A version of a prime number generator is presented later on.)

What kind of threads are Perl threads?

If you have experience with other thread implementations, you might find that things aren't quite what you expect. It's very important to remember when dealing with Perl threads that *Perl Threads Are Not X Threads* for all values of X. They aren't POSIX threads, or DecThreads, or Java's Green threads, or Win32 threads. There are similarities, and the broad concepts are the same, but if you start looking for implementation details you're going to be either disappointed or confused. Possibly both.

This is not to say that Perl threads are completely different from everything that's ever come before. They're not. Perl's threading model owes a lot to other thread models, especially POSIX. Just as Perl is not C, though, Perl threads are not POSIX threads. So if you find yourself looking for mutexes, or thread priorities, it's time to step back a bit and think about what you want to do and how Perl can do it.

However, it is important to remember that Perl threads cannot magically do things unless your operating system's threads allow it. So if your system blocks the entire process on `sleep()`, Perl usually will, as well.

Perl Threads Are Different.

Thread-Safe Modules

The addition of threads has changed Perl's internals substantially. There are implications for people who write modules with XS code or external libraries. However, since Perl data is not shared among threads by default, Perl modules stand a high chance of being thread-safe or can be made thread-safe easily. Modules that are not tagged as thread-safe should be tested or code reviewed before being used in production code.

Not all modules that you might use are thread-safe, and you should always assume a module is unsafe unless the documentation says otherwise. This includes modules that are distributed as part of the core. Threads are a relatively new feature, and even some of the standard modules aren't thread-safe.

Even if a module is thread-safe, it doesn't mean that the module is optimized to work well with threads. A module could possibly be rewritten to utilize the new features in threaded Perl to increase performance in a threaded environment.

If you're using a module that's not thread-safe for some reason, you can protect yourself by using it from one, and only one thread at all. If you need multiple threads to access such a module, you can use semaphores and lots of programming discipline to control access to it. Semaphores are covered in "Basic semaphores".

See also "Thread-Safety of System Libraries".

Thread Basics

The threads module provides the basic functions you need to write threaded programs. In the following sections, we'll cover the basics, showing you what you need to do to create a threaded program. After that, we'll go over some of the features of the threads module that make threaded programming easier.

Basic Thread Support

Thread support is a Perl compile-time option. It's something that's turned on or off when Perl is built at your site, rather than when your programs are compiled. If your Perl wasn't compiled with thread support enabled, then any attempt to use threads will fail.

Your programs can use the Config module to check whether threads are enabled. If your program can't run without them, you can say something like:

```
use Config;
$Config{useithreads} or
die('Recompile Perl with threads to run this program.');
```

A possibly-threaded program using a possibly-threaded module might have code like this:

```
use Config;
use MyMod;

BEGIN {
if ($Config{useithreads}) {
# We have threads
require MyMod_threaded;
import MyMod_threaded;
} else {
require MyMod_unthreaded;
import MyMod_unthreaded;
}
}
```

Since code that runs both with and without threads is usually pretty messy, it's best to isolate the thread-specific code in its own module. In our example above, that's what `MyMod_threaded` is, and it's only imported if we're running on a threaded Perl.

A Note about the Examples

In a real situation, care should be taken that all threads are finished executing before the program exits. That care has **not** been taken in these examples in the interest of simplicity. Running these examples *as is* will produce error messages, usually caused by the fact that there are still threads running when the program exits. You should not be alarmed by this.

Creating Threads

The threads module provides the tools you need to create new threads. Like any other module, you need to tell Perl that you want to use it; `use threads;` imports all the pieces you need to create basic threads.

The simplest, most straightforward way to create a thread is with `create()`:

```
use threads;

my $thr = threads->create(\&sub1);

sub sub1 {
print("In the thread\n");
}
```

The `create()` method takes a reference to a subroutine and creates a new thread that starts executing in the referenced subroutine. Control then passes both to the subroutine and the caller.

If you need to, your program can pass parameters to the subroutine as part of the thread startup. Just include the list of parameters as part of the `threads->create()` call, like this:

```
use threads;

my $Param3 = 'foo';
my $thr1 = threads->create(\&sub1, 'Param 1', 'Param 2', $Param3);
my @ParamList = (42, 'Hello', 3.14);
my $thr2 = threads->create(\&sub1, @ParamList);
my $thr3 = threads->create(\&sub1, qw(Param1 Param2 Param3));

sub sub1 {
my @InboundParameters = @_;
print("In the thread\n");
print('Got parameters >', join('<>',@InboundParameters), "<\n");
}
```

The last example illustrates another feature of threads. You can spawn off several threads using the same subroutine. Each thread executes the same subroutine, but in a separate thread with a separate environment

and potentially separate arguments.

new() is a synonym for create().

Waiting For A Thread To Exit

Since threads are also subroutines, they can return values. To wait for a thread to exit and extract any values it might return, you can use the join() method:

```
use threads;

my ($thr) = threads->create(\&sub1);

my @ReturnData = $thr->join();
print('Thread returned ', join(', ', @ReturnData), "\n");

sub sub1 { return ('Fifty-six', 'foo', 2); }
```

In the example above, the join() method returns as soon as the thread ends. In addition to waiting for a thread to finish and gathering up any values that the thread might have returned, join() also performs any OS cleanup necessary for the thread. That cleanup might be important, especially for long-running programs that spawn lots of threads. If you don't want the return values and don't want to wait for the thread to finish, you should call the detach() method instead, as described next.

NOTE: In the example above, the thread returns a list, thus necessitating that the thread creation call be made in list context (i.e., my ($thr)). See "$thr->*join()*" in threads and "THREAD CONTEXT" in threads for more details on thread context and return values.

Ignoring A Thread

join() does three things: it waits for a thread to exit, cleans up after it, and returns any data the thread may have produced. But what if you're not interested in the thread's return values, and you don't really care when the thread finishes? All you want is for the thread to get cleaned up after when it's done.

In this case, you use the detach() method. Once a thread is detached, it'll run until it's finished; then Perl will clean up after it automatically.

```
use threads;

my $thr = threads->create(\&sub1); # Spawn the thread

$thr->detach(); # Now we officially don't care any more

sleep(15)
# Let thread run for awhile

sub sub1 {
my $count = 0;
while (1) {
$count++;
print("\$count is $count\n");
sleep(1)
}
}
```

Once a thread is detached, it may not be joined, and any return data that it might have produced (if it was done and waiting for a join) is lost.

detach() can also be called as a class method to allow a thread to detach itself:

```
use threads;

my $thr = threads->create(\&sub1);
```

```
sub sub1 {
threads->detach();
# Do more work
}
```

Process and Thread Termination

With threads one must be careful to make sure they all have a chance to run to completion, assuming that is what you want.

An action that terminates a process will terminate *all* running threads. *die()* and *exit()* have this property, and perl does an exit when the main thread exits, perhaps implicitly by falling off the end of your code, even if that's not what you want.

As an example of this case, this code prints the message "Perl exited with active threads: 2 running and unjoined":

```
use threads;
my $thr1 = threads->new(\&thrsub, "test1");
my $thr2 = threads->new(\&thrsub, "test2");
sub thrsub {
my ($message) = @_;
sleep 1;
print "thread $message\n";
}
```

But when the following lines are added at the end:

```
$thr1->join();
$thr2->join();
```

it prints two lines of output, a perhaps more useful outcome.

Threads And Data

Now that we've covered the basics of threads, it's time for our next topic: Data. Threading introduces a couple of complications to data access that non-threaded programs never need to worry about.

Shared And Unshared Data

The biggest difference between Perl *ithreads* and the old 5.005 style threading, or for that matter, to most other threading systems out there, is that by default, no data is shared. When a new Perl thread is created, all the data associated with the current thread is copied to the new thread, and is subsequently private to that new thread! This is similar in feel to what happens when a Unix process forks, except that in this case, the data is just copied to a different part of memory within the same process rather than a real fork taking place.

To make use of threading, however, one usually wants the threads to share at least some data between themselves. This is done with the threads::shared module and the :shared attribute:

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

my $foo :shared = 1;
my $bar = 1;
threads->create(sub { $foo++; $bar++; })->join();

print("$foo\n"); # Prints 2 since $foo is shared
print("$bar\n"); # Prints 1 since $bar is not shared
```

In the case of a shared array, all the array's elements are shared, and for a shared hash, all the keys and values are shared. This places restrictions on what may be assigned to shared array and hash elements: only simple values or references to shared variables are allowed - this is so that a private variable can't accidentally become shared. A bad assignment will cause the thread to die. For example:

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

my $var = 1;
my $svar :shared = 2;
my %hash :shared;

... create some threads ...

$hash{a} = 1; # All threads see exists($hash{a})
# and $hash{a} == 1
$hash{a} = $var; # okay - copy-by-value: same effect as previous
$hash{a} = $svar; # okay - copy-by-value: same effect as previous
$hash{a} = \$svar; # okay - a reference to a shared variable
$hash{a} = \$var; # This will die
delete($hash{a}); # okay - all threads will see !exists($hash{a})
```

Note that a shared variable guarantees that if two or more threads try to modify it at the same time, the internal state of the variable will not become corrupted. However, there are no guarantees beyond this, as explained in the next section.

Thread Pitfalls: Races

While threads bring a new set of useful tools, they also bring a number of pitfalls. One pitfall is the race condition:

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

my $x :shared = 1;
my $thr1 = threads->create(\&sub1);
my $thr2 = threads->create(\&sub2);

$thr1->join();
$thr2->join();
print("$x\n");

sub sub1 { my $foo = $x; $x = $foo + 1; }
sub sub2 { my $bar = $x; $x = $bar + 1; }
```

What do you think $x will be? The answer, unfortunately, is *it depends*. Both sub1() and sub2() access the global variable $x, once to read and once to write. Depending on factors ranging from your thread implementation's scheduling algorithm to the phase of the moon, $x can be 2 or 3.

Race conditions are caused by unsynchronized access to shared data. Without explicit synchronization, there's no way to be sure that nothing has happened to the shared data between the time you access it and the time you update it. Even this simple code fragment has the possibility of error:

```
use threads;
my $x :shared = 2;
my $y :shared;
my $z :shared;
my $thr1 = threads->create(sub { $y = $x; $x = $y + 1; });
my $thr2 = threads->create(sub { $z = $x; $x = $z + 1; });
$thr1->join();
$thr2->join();
```

Two threads both access $x. Each thread can potentially be interrupted at any point, or be executed in any order. At the end, $x could be 3 or 4, and both $y and $z could be 2 or 3.

Even `$x += 5` or `$x++` are not guaranteed to be atomic.

Whenever your program accesses data or resources that can be accessed by other threads, you must take steps to coordinate access or risk data inconsistency and race conditions. Note that Perl will protect its internals from your race conditions, but it won't protect you from you.

Synchronization and control

Perl provides a number of mechanisms to coordinate the interactions between themselves and their data, to avoid race conditions and the like. Some of these are designed to resemble the common techniques used in thread libraries such as `pthreads`; others are Perl-specific. Often, the standard techniques are clumsy and difficult to get right (such as condition waits). Where possible, it is usually easier to use Perlish techniques such as queues, which remove some of the hard work involved.

Controlling access: *lock()*

The `lock()` function takes a shared variable and puts a lock on it. No other thread may lock the variable until the variable is unlocked by the thread holding the lock. Unlocking happens automatically when the locking thread exits the block that contains the call to the `lock()` function. Using `lock()` is straightforward: This example has several threads doing some calculations in parallel, and occasionally updating a running total:

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

my $total :shared = 0;

sub calc {
while (1) {
my $result;
# (... do some calculations and set $result ...)
{
lock($total); # Block until we obtain the lock
$total += $result;
} # Lock implicitly released at end of scope
last if $result == 0;
}
}

my $thr1 = threads->create(\&calc);
my $thr2 = threads->create(\&calc);
my $thr3 = threads->create(\&calc);
$thr1->join();
$thr2->join();
$thr3->join();
print("total=$total\n");
```

`lock()` blocks the thread until the variable being locked is available. When `lock()` returns, your thread can be sure that no other thread can lock that variable until the block containing the lock exits.

It's important to note that locks don't prevent access to the variable in question, only lock attempts. This is in keeping with Perl's longstanding tradition of courteous programming, and the advisory file locking that `flock()` gives you.

You may lock arrays and hashes as well as scalars. Locking an array, though, will not block subsequent locks on array elements, just lock attempts on the array itself.

Locks are recursive, which means it's okay for a thread to lock a variable more than once. The lock will last until the outermost `lock()` on the variable goes out of scope. For example:

```
my $x :shared;
doit();

sub doit {
{
{
lock($x); # Wait for lock
lock($x); # NOOP - we already have the lock
{
lock($x); # NOOP
{
lock($x); # NOOP
lockit_some_more();
}
}
} # *** Implicit unlock here ***
}
}

sub lockit_some_more {
lock($x); # NOOP
} # Nothing happens here
```

Note that there is no unlock() function - the only way to unlock a variable is to allow it to go out of scope.

A lock can either be used to guard the data contained within the variable being locked, or it can be used to guard something else, like a section of code. In this latter case, the variable in question does not hold any useful data, and exists only for the purpose of being locked. In this respect, the variable behaves like the mutexes and basic semaphores of traditional thread libraries.

A Thread Pitfall: Deadlocks

Locks are a handy tool to synchronize access to data, and using them properly is the key to safe shared data. Unfortunately, locks aren't without their dangers, especially when multiple locks are involved. Consider the following code:

```
use threads;

my $x :shared = 4;
my $y :shared = 'foo';
my $thr1 = threads->create(sub {
lock($x);
sleep(20)
lock($y);
});
my $thr2 = threads->create(sub {
lock($y);
sleep(20)
lock($x);
});
```

This program will probably hang until you kill it. The only way it won't hang is if one of the two threads acquires both locks first. A guaranteed-to-hang version is more complicated, but the principle is the same.

The first thread will grab a lock on $x, then, after a pause during which the second thread has probably had time to do some work, try to grab a lock on $y. Meanwhile, the second thread grabs a lock on $y, then later tries to grab a lock on $x. The second lock attempt for both threads will block, each waiting for the other to release its lock.

This condition is called a deadlock, and it occurs whenever two or more threads are trying to get locks on resources that the others own. Each thread will block, waiting for the other to release a lock on a resource. That never happens, though, since the thread with the resource is itself waiting for a lock to be released.

There are a number of ways to handle this sort of problem. The best way is to always have all threads acquire locks in the exact same order. If, for example, you lock variables $x, $y, and $z, always lock $x before $y, and $y before $z. It's also best to hold on to locks for as short a period of time to minimize the risks of deadlock.

The other synchronization primitives described below can suffer from similar problems.

Queues: Passing Data Around

A queue is a special thread-safe object that lets you put data in one end and take it out the other without having to worry about synchronization issues. They're pretty straightforward, and look like this:

```
use threads;
use Thread::Queue;

my $DataQueue = Thread::Queue->new();
my $thr = threads->create(sub {
while (my $DataElement = $DataQueue->dequeue()) {
print("Popped $DataElement off the queue\n");
}
});

$DataQueue->enqueue(12);
$DataQueue->enqueue("A", "B", "C");
sleep(10)
$DataQueue->enqueue(undef);
$thr->join();
```

You create the queue with Thread::Queue->new(). Then you can add lists of scalars onto the end with enqueue(), and pop scalars off the front of it with dequeue(). A queue has no fixed size, and can grow as needed to hold everything pushed on to it.

If a queue is empty, dequeue() blocks until another thread enqueues something. This makes queues ideal for event loops and other communications between threads.

Semaphores: Synchronizing Data Access

Semaphores are a kind of generic locking mechanism. In their most basic form, they behave very much like lockable scalars, except that they can't hold data, and that they must be explicitly unlocked. In their advanced form, they act like a kind of counter, and can allow multiple threads to have the *lock* at any one time.

Basic semaphores

Semaphores have two methods, down() and up(): down() decrements the resource count, while up() increments it. Calls to down() will block if the semaphore's current count would decrement below zero. This program gives a quick demonstration:

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

my $semaphore = Thread::Semaphore->new();
my $GlobalVariable :shared = 0;

$thr1 = threads->create(\&sample_sub, 1);
$thr2 = threads->create(\&sample_sub, 2);
$thr3 = threads->create(\&sample_sub, 3);

sub sample_sub {
```

```
    my $SubNumber = shift(@_);
    my $TryCount = 10;
    my $LocalCopy;
    sleep(1)
    while ($TryCount--) {
    $semaphore->down();
    $LocalCopy = $GlobalVariable;
    print("$TryCount tries left for sub $SubNumber "
    ."(\$GlobalVariable is $GlobalVariable)\n");
    sleep(2)
    $LocalCopy++;
    $GlobalVariable = $LocalCopy;
    $semaphore->up();
    }
    }


    $thr1->join();
    $thr2->join();
    $thr3->join();
```

The three invocations of the subroutine all operate in sync. The semaphore, though, makes sure that only one thread is accessing the global variable at once.

Advanced Semaphores

By default, semaphores behave like locks, letting only one thread down() them at a time. However, there are other uses for semaphores.

Each semaphore has a counter attached to it. By default, semaphores are created with the counter set to one, down() decrements the counter by one, and up() increments by one. However, we can override any or all of these defaults simply by passing in different values:

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

    my $semaphore = Thread::Semaphore->new(5);
    # Creates a semaphore with the counter set to five

    my $thr1 = threads->create(\&sub1);
    my $thr2 = threads->create(\&sub1);

    sub sub1 {
    $semaphore->down(5); # Decrements the counter by five
    # Do stuff here
    $semaphore->up(5); # Increment the counter by five
    }

    $thr1->detach();
    $thr2->detach();
```

If down() attempts to decrement the counter below zero, it blocks until the counter is large enough. Note that while a semaphore can be created with a starting count of zero, any up() or down() always changes the counter by at least one, and so $semaphore->down(0) is the same as $semaphore->down(1).

The question, of course, is why would you do something like this? Why create a semaphore with a starting count that's not one, or why decrement or increment it by more than one? The answer is resource availability. Many resources that you want to manage access for can be safely used by more than one thread at once.

For example, let's take a GUI driven program. It has a semaphore that it uses to synchronize access to the

display, so only one thread is ever drawing at once. Handy, but of course you don't want any thread to start drawing until things are properly set up. In this case, you can create a semaphore with a counter set to zero, and up it when things are ready for drawing.

Semaphores with counters greater than one are also useful for establishing quotas. Say, for example, that you have a number of threads that can do I/O at once. You don't want all the threads reading or writing at once though, since that can potentially swamp your I/O channels, or deplete your process's quota of filehandles. You can use a semaphore initialized to the number of concurrent I/O requests (or open files) that you want at any one time, and have your threads quietly block and unblock themselves.

Larger increments or decrements are handy in those cases where a thread needs to check out or return a number of resources at once.

Waiting for a Condition

The functions `cond_wait()` and `cond_signal()` can be used in conjunction with locks to notify co-operating threads that a resource has become available. They are very similar in use to the functions found in `pthreads`. However for most purposes, queues are simpler to use and more intuitive. See threads::shared for more details.

Giving up control

There are times when you may find it useful to have a thread explicitly give up the CPU to another thread. You may be doing something processor-intensive and want to make sure that the user-interface thread gets called frequently. Regardless, there are times that you might want a thread to give up the processor.

Perl's threading package provides the `yield()` function that does this. `yield()` is pretty straightforward, and works like this:

```
use threads;

sub loop {
my $thread = shift;
my $foo = 50;
while($foo--) { print("In thread $thread\n"); }
threads->yield();
$foo = 50;
while($foo--) { print("In thread $thread\n"); }
}

my $thr1 = threads->create(\&loop, 'first');
my $thr2 = threads->create(\&loop, 'second');
my $thr3 = threads->create(\&loop, 'third');
```

It is important to remember that `yield()` is only a hint to give up the CPU, it depends on your hardware, OS and threading libraries what actually happens. **On many operating systems, *yield()* is a no-op.** Therefore it is important to note that one should not build the scheduling of the threads around `yield()` calls. It might work on your platform but it won't work on another platform.

General Thread Utility Routines

We've covered the workhorse parts of Perl's threading package, and with these tools you should be well on your way to writing threaded code and packages. There are a few useful little pieces that didn't really fit in anyplace else.

What Thread Am I In?

The `threads->self()` class method provides your program with a way to get an object representing the thread it's currently in. You can use this object in the same way as the ones returned from thread creation.

Thread IDs

`tid()` is a thread object method that returns the thread ID of the thread the object represents. Thread IDs are integers, with the main thread in a program being 0. Currently Perl assigns a unique TID to every thread

ever created in your program, assigning the first thread to be created a TID of 1, and increasing the TID by 1 for each new thread that's created. When used as a class method, `threads->tid()` can be used by a thread to get its own TID.

Are These Threads The Same?

The `equal()` method takes two thread objects and returns true if the objects represent the same thread, and false if they don't.

Thread objects also have an overloaded `==` comparison so that you can do comparison on them as you would with normal objects.

What Threads Are Running?

`threads->list()` returns a list of thread objects, one for each thread that's currently running and not detached. Handy for a number of things, including cleaning up at the end of your program (from the main Perl thread, of course):

```
# Loop through all the threads
foreach my $thr (threads->list()) {
$thr->join();
}
```

If some threads have not finished running when the main Perl thread ends, Perl will warn you about it and die, since it is impossible for Perl to clean up itself while other threads are running.

NOTE: The main Perl thread (thread 0) is in a *detached* state, and so does not appear in the list returned by `threads->list()`.

A Complete Example

Confused yet? It's time for an example program to show some of the things we've covered. This program finds prime numbers using threads.

```
1 #!/usr/bin/perl
2 # prime-pthread, courtesy of Tom Christiansen
3
4 use strict;
5 use warnings;
6
7 use threads;
8 use Thread::Queue;
9
10 sub check_num {
11 my ($upstream, $cur_prime) = @_;
12 my $kid;
13 my $downstream = Thread::Queue->new();
14 while (my $num = $upstream->dequeue()) {
15 next unless ($num % $cur_prime);
16 if ($kid) {
17 $downstream->enqueue($num);
18 } else {
19 print("Found prime: $num\n");
20 $kid = threads->create(\&check_num, $downstream, $num);
21 if (! $kid) {
22 warn("Sorry. Ran out of threads.\n");
23 last;
24 }
25 }
26 }
27 if ($kid) {
28 $downstream->enqueue(undef);
```

```
29  $kid->join();
30  }
31  }
32
33  my $stream = Thread::Queue->new(3..1000, undef);
34  check_num($stream, 2);
```

This program uses the pipeline model to generate prime numbers. Each thread in the pipeline has an input queue that feeds numbers to be checked, a prime number that it's responsible for, and an output queue into which it funnels numbers that have failed the check. If the thread has a number that's failed its check and there's no child thread, then the thread must have found a new prime number. In that case, a new child thread is created for that prime and stuck on the end of the pipeline.

This probably sounds a bit more confusing than it really is, so let's go through this program piece by piece and see what it does. (For those of you who might be trying to remember exactly what a prime number is, it's a number that's only evenly divisible by itself and 1.)

The bulk of the work is done by the check_num() subroutine, which takes a reference to its input queue and a prime number that it's responsible for. After pulling in the input queue and the prime that the subroutine is checking (line 11), we create a new queue (line 13) and reserve a scalar for the thread that we're likely to create later (line 12).

The while loop from line 14 to line 26 grabs a scalar off the input queue and checks against the prime this thread is responsible for. Line 15 checks to see if there's a remainder when we divide the number to be checked by our prime. If there is one, the number must not be evenly divisible by our prime, so we need to either pass it on to the next thread if we've created one (line 17) or create a new thread if we haven't.

The new thread creation is line 20. We pass on to it a reference to the queue we've created, and the prime number we've found. In lines 21 through 24, we check to make sure that our new thread got created, and if not, we stop checking any remaining numbers in the queue.

Finally, once the loop terminates (because we got a 0 or undef in the queue, which serves as a note to terminate), we pass on the notice to our child, and wait for it to exit if we've created a child (lines 27 and 30).

Meanwhile, back in the main thread, we first create a queue (line 33) and queue up all the numbers from 3 to 1000 for checking, plus a termination notice. Then all we have to do to get the ball rolling is pass the queue and the first prime to the check_num() subroutine (line 34).

That's how it works. It's pretty simple; as with many Perl programs, the explanation is much longer than the program.

Different implementations of threads

Some background on thread implementations from the operating system viewpoint. There are three basic categories of threads: user-mode threads, kernel threads, and multiprocessor kernel threads.

User-mode threads are threads that live entirely within a program and its libraries. In this model, the OS knows nothing about threads. As far as it's concerned, your process is just a process.

This is the easiest way to implement threads, and the way most OSes start. The big disadvantage is that, since the OS knows nothing about threads, if one thread blocks they all do. Typical blocking activities include most system calls, most I/O, and things like sleep().

Kernel threads are the next step in thread evolution. The OS knows about kernel threads, and makes allowances for them. The main difference between a kernel thread and a user-mode thread is blocking. With kernel threads, things that block a single thread don't block other threads. This is not the case with user-mode threads, where the kernel blocks at the process level and not the thread level.

This is a big step forward, and can give a threaded program quite a performance boost over non-threaded programs. Threads that block performing I/O, for example, won't block threads that are doing other things. Each process still has only one thread running at once, though, regardless of how many CPUs a system might have.

Since kernel threading can interrupt a thread at any time, they will uncover some of the implicit locking assumptions you may make in your program. For example, something as simple as `$x = $x + 2` can behave unpredictably with kernel threads if `$x` is visible to other threads, as another thread may have changed `$x` between the time it was fetched on the right hand side and the time the new value is stored.

Multiprocessor kernel threads are the final step in thread support. With multiprocessor kernel threads on a machine with multiple CPUs, the OS may schedule two or more threads to run simultaneously on different CPUs.

This can give a serious performance boost to your threaded program, since more than one thread will be executing at the same time. As a tradeoff, though, any of those nagging synchronization issues that might not have shown with basic kernel threads will appear with a vengeance.

In addition to the different levels of OS involvement in threads, different OSes (and different thread implementations for a particular OS) allocate CPU cycles to threads in different ways.

Cooperative multitasking systems have running threads give up control if one of two things happen. If a thread calls a yield function, it gives up control. It also gives up control if the thread does something that would cause it to block, such as perform I/O. In a cooperative multitasking implementation, one thread can starve all the others for CPU time if it so chooses.

Preemptive multitasking systems interrupt threads at regular intervals while the system decides which thread should run next. In a preemptive multitasking system, one thread usually won't monopolize the CPU.

On some systems, there can be cooperative and preemptive threads running simultaneously. (Threads running with realtime priorities often behave cooperatively, for example, while threads running at normal priorities behave preemptively.)

Most modern operating systems support preemptive multitasking nowadays.

Performance considerations

The main thing to bear in mind when comparing Perl's *ithreads* to other threading models is the fact that for each new thread created, a complete copy of all the variables and data of the parent thread has to be taken. Thus, thread creation can be quite expensive, both in terms of memory usage and time spent in creation. The ideal way to reduce these costs is to have a relatively short number of long-lived threads, all created fairly early on (before the base thread has accumulated too much data). Of course, this may not always be possible, so compromises have to be made. However, after a thread has been created, its performance and extra memory usage should be little different than ordinary code.

Also note that under the current implementation, shared variables use a little more memory and are a little slower than ordinary variables.

Process-scope Changes

Note that while threads themselves are separate execution threads and Perl data is thread-private unless explicitly shared, the threads can affect process-scope state, affecting all the threads.

The most common example of this is changing the current working directory using `chdir()`. One thread calls `chdir()`, and the working directory of all the threads changes.

Even more drastic example of a process-scope change is `chroot()`: the root directory of all the threads changes, and no thread can undo it (as opposed to `chdir()`).

Further examples of process-scope changes include `umask()` and changing uids and gids.

Thinking of mixing `fork()` and threads? Please lie down and wait until the feeling passes. Be aware that the semantics of `fork()` vary between platforms. For example, some Unix systems copy all the current threads into the child process, while others only copy the thread that called `fork()`. You have been warned!

Similarly, mixing signals and threads may be problematic. Implementations are platform-dependent, and even the POSIX semantics may not be what you expect (and Perl doesn't even give you the full POSIX API). For example, there is no way to guarantee that a signal sent to a multi-threaded Perl application will get intercepted by any particular thread. (However, a recently added feature does provide the capability to send

signals between threads. See ''THREAD SIGNALLING'' in threads for more details.)

Thread-Safety of System Libraries

Whether various library calls are thread-safe is outside the control of Perl. Calls often suffering from not being thread-safe include: `localtime()`, `gmtime()`, functions fetching user, group and network information (such as `getgrent()`, `gethostent()`, `getnetent()` and so on), `readdir()`, `rand()`, and `srand()`. In general, calls that depend on some global external state.

If the system Perl is compiled in has thread-safe variants of such calls, they will be used. Beyond that, Perl is at the mercy of the thread-safety or -unsafety of the calls. Please consult your C library call documentation.

On some platforms the thread-safe library interfaces may fail if the result buffer is too small (for example the user group databases may be rather large, and the reentrant interfaces may have to carry around a full snapshot of those databases). Perl will start with a small buffer, but keep retrying and growing the result buffer until the result fits. If this limitless growing sounds bad for security or memory consumption reasons you can recompile Perl with `PERL_REENTRANT_MAXSIZE` defined to the maximum number of bytes you will allow.

Conclusion

A complete thread tutorial could fill a book (and has, many times), but with what we've covered in this introduction, you should be well on your way to becoming a threaded Perl expert.

SEE ALSO

Annotated POD for threads: <http://annocpan.org/?mode=search&field=Module&name=threads>

Latest version of threads on CPAN: <http://search.cpan.org/search?module=threads>

Annotated   POD   for   threads::shared: <http://annocpan.org/?mode=search&field=Module&name=threads%3A%3Ashared>

Latest   version   of   threads::shared   on   CPAN: <http://search.cpan.org/search?module=threads%3A%3Ashared>

Perl threads mailing list: <http://lists.perl.org/list/ithreads.html>

Bibliography

Here's a short bibliography courtesy of Jürgen Christoffel:

Introductory Texts

Birrell, Andrew D. An Introduction to Programming with Threads. Digital Equipment Corporation, 1989, DEC-SRC Research Report #35 online as ftp://ftp.dec.com/pub/DEC/SRC/research-reports/SRC-035.pdf (highly recommended)

Robbins, Kay. A., and Steven Robbins. Practical Unix Programming: A Guide to Concurrency, Communication, and Multithreading. Prentice-Hall, 1996.

Lewis, Bill, and Daniel J. Berg. Multithreaded Programming with Pthreads. Prentice Hall, 1997, ISBN 0-13-443698-9 (a well-written introduction to threads).

Nelson, Greg (editor). Systems Programming with Modula-3. Prentice Hall, 1991, ISBN 0-13-590464-1.

Nichols, Bradford, Dick Buttlar, and Jacqueline Proulx Farrell. Pthreads Programming. O'Reilly & Associates, 1996, ISBN 156592-115-1 (covers POSIX threads).

OS-Related References

Boykin, Joseph, David Kirschen, Alan Langerman, and Susan LoVerso. Programming under Mach. Addison-Wesley, 1994, ISBN 0-201-52739-1.

Tanenbaum, Andrew S. Distributed Operating Systems. Prentice Hall, 1995, ISBN 0-13-219908-4 (great textbook).

Silberschatz, Abraham, and Peter B. Galvin. Operating System Concepts, 4th ed. Addison-Wesley, 1995, ISBN 0-201-59292-4

Other References

Arnold, Ken and James Gosling. The Java Programming Language, 2nd ed. Addison-Wesley, 1998, ISBN 0-201-31006-6.

comp.programming.threads FAQ, <http://www.serpentine.com/˜bos/threads-faq/>

Le Sergent, T. and B. Berthomieu. ''Incremental MultiThreaded Garbage Collection on Virtually Shared Memory Architectures'' in Memory Management: Proc. of the International Workshop IWMM 92, St. Malo, France, September 1992, Yves Bekkers and Jacques Cohen, eds. Springer, 1992, ISBN 3540-55940-X (real-life thread applications).

Artur Bergman, ''Where Wizards Fear To Tread'', June 11, 2002, <http://www.perl.com/pub/a/2002/06/11/threads.html>

Acknowledgements

Thanks (in no particular order) to Chaim Frenkel, Steve Fink, Gurusamy Sarathy, Ilya Zakharevich, Benjamin Sugars, Jürgen Christoffel, Joshua Pritikin, and Alan Burlison, for their help in reality-checking and polishing this article. Big thanks to Tom Christiansen for his rewrite of the prime number generator.

AUTHOR

Dan Sugalski <dan@sidhe.org<gt>

Slightly modified by Arthur Bergman to fit the new thread model/module.

Reworked slightly by Jörg Walter <jwalt@cpan.org<gt> to be more concise about thread-safety of Perl code.

Rearranged slightly by Elizabeth Mattijsen <liz@dijkmat.nl<gt> to put less emphasis on *yield()*.

Copyrights