NAME

perlothrtut - old tutorial on threads in Perl

DESCRIPTION

WARNING: This tutorial describes the old-style thread model that was introduced in release 5.005. This model is deprecated, and has been removed for version 5.10. The interfaces described here were considered experimental, and are likely to be buggy.

For information about the new interpreter threads ("ithreads") model, see the *perlthrtut* tutorial, and the *threads* and *threads::shared* modules.

You are strongly encouraged to migrate any existing threads code to the new model as soon as possible.

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 5.005, 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.)

Native threads

There are several different ways to implement threads on a system. How threads are implemented depends both on the vendor and, in some cases, the version of the operating system. Often the first implementation will be relatively simple, but later versions of the OS will be more sophisticated.

While the information in this section is useful, it's not necessary, so you can skip it if you don't feel up to it.

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 `$a = $a + 2` can behave unpredictably with kernel threads if $a is visible to other threads, as another thread may have changed $a 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.)

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.

Threadsafe Modules

The addition of threads has changed Perl's internals substantially. There are implications for people who write modules--especially modules with XS code or external libraries. While most modules won't encounter any problems, modules that aren't explicitly tagged as thread-safe should be tested 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 beta feature, and even some of the standard modules aren't thread-safe.

If you're using a module that's not thread-safe for some reason, you can protect yourself by using semaphores and lots of programming discipline to control access to the module. Semaphores are covered later in the article. Perl Threads Are Different

Thread Basics

The core Thread 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 Thread 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.

Remember that the threading support in 5.005 is in beta release, and should be treated as such. You should expect that it may not function entirely properly, and the thread interface may well change some before it is a fully supported, production release. The beta version shouldn't be used for mission-critical projects. Having said that, threaded Perl is pretty nifty, and worth a look.

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:

```
  $Config{usethreads} 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;

    if ($Config{usethreads}) {
```

```
    # 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.

Creating Threads

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

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

```
use Thread;

$thr = Thread->new( \&sub1 );

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

The new() method takes a reference to a subroutine and creates a new thread, which 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 `Thread::new` call, like this:

```
use Thread;
$Param3 = "foo";
$thr = Thread->new( \&sub1, "Param 1", "Param 2", $Param3 );
$thr = Thread->new( \&sub1, @ParamList );
$thr = Thread->new( \&sub1, qw(Param1 Param2 $Param3) );

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

The subroutine runs like a normal Perl subroutine, and the call to new Thread returns whatever the subroutine returns.

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.

The other way to spawn a new thread is with async(), which is a way to spin off a chunk of code like eval(), but into its own thread:

```
use Thread qw(async);
```

```
$LineCount = 0;

$thr = async {
    while(<>) {$LineCount++}
    print "Got $LineCount lines\n";
};

print "Waiting for the linecount to end\n";
$thr->join;
print "All done\n";
```

You'll notice we did a use Thread qw(async) in that example. async is not exported by default, so if you want it, you'll either need to import it before you use it or fully qualify it as Thread::async. You'll also note that there's a semicolon after the closing brace. That's because async() treats the following block as an anonymous subroutine, so the semicolon is necessary.

Like eval(), the code executes in the same context as it would if it weren't spun off. Since both the code inside and after the async start executing, you need to be careful with any shared resources. Locking and other synchronization techniques are covered later.

Giving up control

There are times when you may find it useful to have a thread explicitly give up the CPU to another thread. Your threading package might not support preemptive multitasking for threads, for example, or you may be doing something compute-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 Thread qw(yield async);
async {
    my $foo = 50;
    while ($foo--) { print "first async\n" }
    yield;
    $foo = 50;
    while ($foo--) { print "first async\n" }
};
async {
    my $foo = 50;
    while ($foo--) { print "second async\n" }
    yield;
    $foo = 50;
    while ($foo--) { print "second async\n" }
};
```

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 scalars it might return, you can use the join() method.

```
use Thread;
$thr = Thread->new( \&sub1 );

@ReturnData = $thr->join;
print "Thread returned @ReturnData";
```

```
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. detach() is covered later in the article.

Errors In Threads

So what happens when an error occurs in a thread? Any errors that could be caught with eval() are postponed until the thread is joined. If your program never joins, the errors appear when your program exits.

Errors deferred until a join() can be caught with eval():

```
use Thread qw(async);
$thr = async {$b = 3/0;   # Divide by zero error
$foo = eval {$thr->join};
if ($@) {
    print "died with error $@\n";
} else {
    print "Hey, why aren't you dead?\n";
}
```

eval() passes any results from the joined thread back unmodified, so if you want the return value of the thread, this is your only chance to get them.

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 Thread;
$thr = Thread->new( \&sub1 ); # Spawn the thread

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

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

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

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 single most important thing to remember when using threads is that all threads potentially have access to all the data anywhere in your program. While this is true with a nonthreaded Perl program as well, it's especially important to remember with a threaded program, since more than one thread can be accessing this data at once.

Perl's scoping rules don't change because you're using threads. If a subroutine (or block, in the case of async()) could see a variable if you weren't running with threads, it can see it if you are. This is especially important for the subroutines that create, and makes `my` variables even more important. Remember--if your variables aren't lexically scoped (declared with `my`) you're probably sharing them between threads.

Thread Pitfall: Races

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

```
use Thread;
$a = 1;
$thr1 = Thread->new(\&sub1);
$thr2 = Thread->new(\&sub2);

sleep 10;
print "$a\n";

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

What do you think $a will be? The answer, unfortunately, is "it depends." Both sub1() and sub2() access the global variable $a, once to read and once to write. Depending on factors ranging from your thread implementation's scheduling algorithm to the phase of the moon, $a 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 Thread qw(async);
$a = 2;
async{ $b = $a; $a = $b + 1; };
async{ $c = $a; $a = $c + 1; };
```

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

Whenever your program accesses data or resources that can be accessed by other threads, you must take steps to coordinate access or risk data corruption and race conditions.

Controlling access: lock()

The lock() function takes a variable (or subroutine, but we'll get to that later) and puts a lock on it. No other thread may lock the variable until the locking thread exits the innermost block containing the lock. Using lock() is straightforward:

```
use Thread qw(async);
$a = 4;
$thr1 = async {
    $foo = 12;
```

```
        {
            lock ($a); # Block until we get access to $a
            $b = $a;
            $a = $b * $foo;
        }
        print "\$foo was $foo\n";
    };
    $thr2 = async {
        $bar = 7;
        {
            lock ($a); # Block until we can get access to $a
            $c = $a;
            $a = $c * $bar;
        }
        print "\$bar was $bar\n";
    };
    $thr1->join;
    $thr2->join;
    print "\$a is $a\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 innermost 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. Locked subroutines behave differently, however. We'll cover that later in the article.

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.

Finally, 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.

Thread Pitfall: Deadlocks

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

```
    use Thread qw(async yield);
    $a = 4;
    $b = "foo";
    async {
        lock($a);
        yield;
        sleep 20;
        lock ($b);
    };
    async {
        lock($b);
        yield;
        sleep 20;
        lock ($a);
    };
```

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

is the same.

The first thread spawned by async() will grab a lock on $a then, a second or two later, try to grab a lock on $b. Meanwhile, the second thread grabs a lock on $b, then later tries to grab a lock on $a. 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 $a, $b, and $c, always lock $a before $b, and $b before $c. It's also best to hold on to locks for as short a period of time to minimize the risks of deadlock.

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 Thread qw(async);
use Thread::Queue;

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

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

You create the queue with new Thread::Queue. 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.

Threads And Code

In addition to providing thread-safe access to data via locks and queues, threaded Perl also provides general-purpose semaphores for coarser synchronization than locks provide and thread-safe access to entire subroutines.

Semaphores: Synchronizing Data Access

Semaphores are a kind of generic locking mechanism. Unlike lock, which gets a lock on a particular scalar, Perl doesn't associate any particular thing with a semaphore so you can use them to control access to anything you like. In addition, semaphores can allow more than one thread to access a resource at once, though by default semaphores only allow one thread access at a time.

Basic semaphores

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

```
    use Thread qw(yield);
    use Thread::Semaphore;
    my $semaphore = Thread::Semaphore->new();
    $GlobalVariable = 0;

    $thr1 = Thread->new( \&sample_sub, 1 );
    $thr2 = Thread->new( \&sample_sub, 2 );
    $thr3 = Thread->new( \&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";
            yield;
            sleep 2;
            $LocalCopy++;
            $GlobalVariable = $LocalCopy;
            $semaphore->up;
        }
    }
```

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. down() decrements the counter and up() increments the counter. By default, semaphores are created with the counter set to one, down() decrements by one, and up() increments by one. 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. $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/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' 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.

Attributes: Restricting Access To Subroutines

In addition to synchronizing access to data or resources, you might find it useful to synchronize access to subroutines. You may be accessing a singular machine resource (perhaps a vector processor), or find it easier to serialize calls to a particular subroutine than to have a set of locks and semaphores.

One of the additions to Perl 5.005 is subroutine attributes. The Thread package uses these to provide several flavors of serialization. It's important to remember that these attributes are used in the compilation phase of your program so you can't change a subroutine's behavior while your program is actually running.

Subroutine Locks

The basic subroutine lock looks like this:

```
sub test_sub :locked {
}
```

This ensures that only one thread will be executing this subroutine at any one time. Once a thread calls this subroutine, any other thread that calls it will block until the thread in the subroutine exits it. A more elaborate example looks like this:

```
use Thread qw(yield);

new Thread \&thread_sub, 1;
new Thread \&thread_sub, 2;
new Thread \&thread_sub, 3;
new Thread \&thread_sub, 4;

sub sync_sub :locked {
    my $CallingThread = shift @_;
    print "In sync_sub for thread $CallingThread\n";
    yield;
    sleep 3;
    print "Leaving sync_sub for thread $CallingThread\n";
}

sub thread_sub {
    my $ThreadID = shift @_;
    print "Thread $ThreadID calling sync_sub\n";
    sync_sub($ThreadID);
    print "$ThreadID is done with sync_sub\n";
}
```

The `locked` attribute tells perl to lock sync_sub(), and if you run this, you can see that only one thread is in it at any one time.

Methods

Locking an entire subroutine can sometimes be overkill, especially when dealing with Perl objects. When calling a method for an object, for example, you want to serialize calls to a method, so that only one thread will be in the subroutine for a particular object, but threads calling that subroutine for a different object aren't blocked. The method attribute indicates whether the subroutine is really a method.

```
use Thread;
```

```
sub tester {
    my $thrnum = shift @_;
    my $bar = Foo->new();
    foreach (1..10) {
        print "$thrnum calling per_object\n";
        $bar->per_object($thrnum);
        print "$thrnum out of per_object\n";
        yield;
        print "$thrnum calling one_at_a_time\n";
        $bar->one_at_a_time($thrnum);
        print "$thrnum out of one_at_a_time\n";
        yield;
    }
}

foreach my $thrnum (1..10) {
    new Thread \&tester, $thrnum;
}

package Foo;
sub new {
    my $class = shift @_;
    return bless [@_], $class;
}

sub per_object :locked :method {
    my ($class, $thrnum) = @_;
    print "In per_object for thread $thrnum\n";
    yield;
    sleep 2;
    print "Exiting per_object for thread $thrnum\n";
}

sub one_at_a_time :locked {
    my ($class, $thrnum) = @_;
    print "In one_at_a_time for thread $thrnum\n";
    yield;
    sleep 2;
    print "Exiting one_at_a_time for thread $thrnum\n";
}
```

As you can see from the output (omitted for brevity; it's 800 lines) all the threads can be in per_object() simultaneously, but only one thread is ever in one_at_a_time() at once.

Locking A Subroutine

You can lock a subroutine as you would lock a variable. Subroutine locks work the same as specifying a `locked` attribute for the subroutine, and block all access to the subroutine for other threads until the lock goes out of scope. When the subroutine isn't locked, any number of threads can be in it at once, and getting a lock on a subroutine doesn't affect threads already in the subroutine. Getting a lock on a subroutine looks like this:

```
lock(\&sub_to_lock);
```

Simple enough. Unlike the `locked` attribute, which is a compile time option, locking and unlocking a

subroutine can be done at runtime at your discretion. There is some runtime penalty to using lock(\&sub) instead of the `locked` attribute, so make sure you're choosing the proper method to do the locking.

You'd choose lock(\&sub) when writing modules and code to run on both threaded and unthreaded Perl, especially for code that will run on 5.004 or earlier Perls. In that case, it's useful to have subroutines that should be serialized lock themselves if they're running threaded, like so:

```
package Foo;
use Config;
$Running_Threaded = 0;

BEGIN { $Running_Threaded = $Config{'usethreads'} }

sub sub1 { lock(\&sub1) if $Running_Threaded }
```

This way you can ensure single-threadedness regardless of which version of Perl you're running.

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 Thread->self 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 the 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.

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.

What Threads Are Running?

Thread->list returns a list of thread objects, one for each thread that's currently running. Handy for a number of things, including cleaning up at the end of your program:

```
# Loop through all the threads
foreach $thr (Thread->list) {
    # Don't join the main thread or ourselves
    if ($thr->tid && !Thread::equal($thr, Thread->self)) {
        $thr->join;
    }
}
```

The example above is just for illustration. It isn't strictly necessary to join all the threads you create, since Perl detaches all the threads before it exits.

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 -w
2  # prime-pthread, courtesy of Tom Christiansen
3
4  use strict;
5
6  use Thread;
7  use Thread::Queue;
8
9  my $stream = Thread::Queue->new();
10 my $kid    = Thread->new(\&check_num, $stream, 2);
11
12 for my $i ( 3 .. 1000 ) {
13     $stream->enqueue($i);
14 }
15
16 $stream->enqueue(undef);
17 $kid->join();
18
19 sub check_num {
20     my ($upstream, $cur_prime) = @_;
21     my $kid;
22     my $downstream = Thread::Queue->new();
23     while (my $num = $upstream->dequeue) {
24         next unless $num % $cur_prime;
25         if ($kid) {
26            $downstream->enqueue($num);
27           } else {
28            print "Found prime $num\n";
29               $kid = Thread->new(\&check_num, $downstream, $num);
30         }
31     }
32     $downstream->enqueue(undef) if $kid;
33     $kid->join()  if $kid;
34 }
```

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 that it funnels numbers that have failed the check into. 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 lets 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's checking (line 20), we create a new queue (line 22) and reserve a scalar for the thread that we're likely to create later (line 21).

The while loop from lines 23 to line 31 grabs a scalar off the input queue and checks against the prime this thread is responsible for. Line 24 checks to see if there's a remainder when we modulo the number to be checked against 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 26) or create a new thread if we haven't.

The new thread creation is line 29. We pass on to it a reference to the queue we've created, and the

prime number we've found.

Finally, once the loop terminates (because we got a 0 or undef in the queue, which serves as a note to die), we pass on the notice to our child and wait for it to exit if we've created a child (Lines 32 and 37).

Meanwhile, back in the main thread, we create a queue (line 9) and the initial child thread (line 10), and pre-seed it with the first prime: 2. Then we queue all the numbers from 3 to 1000 for checking (lines 12-14), then queue a die notice (line 16) and wait for the first child thread to terminate (line 17). Because a child won't die until its child has died, we know that we're done once we return from the join.

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

Conclusion

A complete thread tutorial could fill a book (and has, many times), but this should get you well on your way. The final authority on how Perl's threads behave is the documentation bundled with the Perl distribution, but with what we've covered in this article, you should be well on your way to becoming a threaded Perl expert.

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 http://www.research.digital.com/SRC/staff/birrell/bib.html (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.

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).

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 <sugalskd@ous.edu>

Copyrights