Threading and Perl

Introduction To Perl Threads and Basics of Concurrent Programming

eric.maki for kwpm
24/07/2008
• **Thread Basics**

• **Threads and Shared Memory API**
  – For Perl 5.8.3 and above

• **Synchronization Mechanisms**
  – What’s built in?
  – Generic synchronization primitives
    • What they are and how they are used
    • What/how you can build from Perl’s built-ins (or use from CPAN)
What is a Thread?

• Most generally: an execution context.

• Threads normally provide a mechanism for managing multiple execution contexts within the same OS process.

• Threads can interact via shared memory, rather than via IPC.
Threads and Languages

• Language approaches to Threads:
  – **Java**: Supported in the JVM, thus it is a core language element.
  – **pthreads (POSIX Threads)**: a cross language standard used widely with C/C++. This is an API, not an implementation.
  – **Python**: Core piece of language, implementation is platform dependent.
Threads versus Fibres

• The term ‘thread’ normally refers to execution contexts created by user processes, but scheduled by the kernel.

• ‘Fibres’ are execution contexts managed entirely in user-space.
  – Advantage here is selection and optimization of scheduling and context scope.
  – Obviously, also downsides.
Perl and Threads: Versions

• **Interpreter Threads** introduced in Perl 5.6, and widely available and reasonably stable in Perl 5.8.

• Perl 5.5 had a different threading model, which continued to be supported up to 5.10.
  – It never progressed beyond experimental.
  – I won’t discuss “old style threads” further.
ithreads basic model

- Unlike most other thread implementations, in ithreads **global variables are non-shared by default**.
- A new thread actually gets a copy of **everything**.
- This includes the interpreter instance.
- Model is based around explicit sharing.
• Very basic model
• No notion of thread priority
• Minimal built in synchronization mechanisms:
  – No mutexes
  – No semaphores
• Everything based around shared locks.
Implementation

• Employs pthreads where they are available (everywhere but Win32, I believe)

• pthreads on Linux are treated as ‘special’ processes with a shared memory space

• Win32 uses windows threading model (some impedance mismatch)
Jumping in

- We’ll jump into the code examples in Perl
- Concurrent programming requires a different mindset from strictly sequential programming
Create a thread

use threads;
my $thr = threads->create( \&entry_point, @args );

• A new thread is created, with a
  – new interpreter, stack, etc.,
  – new copies of all unshared globals.
• Starts by calling entry_point( @args )
• When the sub returns, the thread is complete
• No guarantees of ‘first execution’
Thread Death

my $res = $thr->join();

• Return value of a thread is the return value of the function
• When a thread completes it waits for the result to be read
• Joining **blocks** on thread completion
• Thread is destroyed on `join()`
Detaching threads

$thr->detach();

• Thread will not block on completion
• Thread will never become joinable
• However, you must ensure that the thread completes before your program terminates
my $t1 = threads->create( sub { sleep 1; } );
my $t2 = threads->create( sub { sleep 2; } );
$t2->detach();
my $t3 = threads->create( sub { sleep 5; } );
my $t4 = threads->create( sub { sleep 5; } );
$t4->detach();
sleep 3;

# yields:
Perl exited with active threads:
  1 running and unjoined
  1 finished and unjoined
  1 running and detached
use threads;
local $SIG{INT} = sub { die };

my $j1 = threads-&gt;create( sub { sleep 1; } );
my $d2 = threads-&gt;create( sub { sleep 2; } );
my $j3 = threads-&gt;create( sub { sleep 5; } );
my $d4 = threads-&gt;create( sub { sleep 50; } );
$d2-&gt;detach();
$d4-&gt;detach();

$j1-&gt;join();        # joinable are joined
$j3-&gt;join();
$d4-&gt;kill('INT');   # detached are stopped
              # d2 is ignored
sleep 1;            # race!
Other Basic Controls

• `threads->yield()`
  – Kernel hint: schedule something else. Might be a no-op, depending on implementation.

• `threads->list()`
  – Fetch list of all threads, or threads in certain states.

• `my $tid = async {};`
  – Just sugar for creating a thread with anonymous sub.
Shared Variables

• Nothing is shared by default

# compile-time:
my $foo :shared = 8;
my %hash :shared;

# runtime:
share( $bar );         # one level
shared_clone( $baz );  # deep share
Sharing Internals

```perl
my $foo :shared = 42;

SV = PVMG(0x1297048) at 0x1233c10
  REFCNT = 1
  FLAGS = (PADMY,GMG,SMG,pIOK)
  IV = 42
  NV = 0
  PV = 0
  MAGIC = 0x12549a0
    MG_VIRTUAL = 0x528e8040
    MG_TYPE = PERL_MAGIC_shared_scalar(n)
    MG_FLAGS = 0x30
    MG_PTR = 0x12d4910 ""
```
The Implementation

• An extra thread is created for shared memory
• Each thread that has access to a shared variable gets a handle variable
• Which is essentially tie()ed to the shared thread’s version.
The Costs

- Shared memory is thus horribly expensive.
- Somewhat unavoidable
  - Perl variables are complex, and internal consistency needs to be arbitrated
  - Each shared variable has a mutex guarding it
  - No atomic types, strictly speaking, but this is close
Atomic assignment, but no atomic test

my $cnt :shared = 0;
my $t1 = threads->create( \&work, \$cnt );
my $t2 = threads->create( \&work, \$cnt );
$t1->join();
$t2->join();

sub work {  
    my $cnt = shift;
    do {  
        $$cnt++;
        print "$$cnt\n";
    } while $$cnt < 5;
    
}  
# prints 1–6, add a sleep, and 1–5
# locks are very basic:

sub work {
    my $cnt = shift;
    while (1) {
        lock( $$cnt ); # lock held by scope
        print $$cnt++ . "\n"; # meaning inc and
        last if $$cnt >= 5; # cmp are now atomic
    }
}

# will always print 1..5
Locks

- Held only by scope
- Take a shared variable as argument
- Block until thread has exclusive lock

- There is no 'try_lock'
- Deadlocks are easy
Deadlock

my $foo :shared = 0;
my $bar :shared = 0;

my $t1 = threads->create(
    sub { lock( $foo ); sleep 1; lock( $bar ); } );
my $t2 = threads->create(
    sub { lock( $bar ); sleep 1; lock( $foo ); } );

# threads block until killed
cond_wait and cond_broadcast

• Only one more real set of primitives in threaded Perl:
  – `cond_wait( $var );`
    • Block until another thread broadcasts for this shared $var
  – `cond_broadcast( $var );`
    • Notify all users waiting on this shared $var
• No guarantee the value changed, just as simple as that.
Busy Wait versus Intelligent Wait

- Why?

```perl
# busy wait:
1 until ( $shr_flag == 1 );
# lazy wait:
sleep 1 until ( $flag == 1 );

# smart:
lock( $flag );
cond_wait( $flag ) until $flag == 1;
# but all modifiers need to broadcast
```
Threadsafe Queue

- Can build complex structures from these primitives
- I’ll illustrate this and provide an example for cond_* at the same time

- Thread safe queue: want a FIFO pipe that can be used by arbitrary number of threads
Queue implementation

my @queue : shared;

sub enqueue {  
    lock( @queue );
    push @queue, @_;  
    cond_broadcast( @queue );
}

sub dequeue {  
    lock( @queue );
    cond_wait( @queue ) until @queue > 0;  
    return shift @queue;
}
cond_signal()

my @queue :shared;

sub enqueue {
    lock( @queue );
    push @queue, @_; 
    cond_signal( @queue );
}

sub dequeue {
    lock( @queue );
    cond_wait( @queue ) until @queue > 0;
    cond_signal( @queue ) if @queue > 1;
    return shift @queue;
}
package Queue;
use threads;
use threads::shared;

sub new {
    my $class = shift;
    my @queue :shared;
    return bless [@queue, $class];
}

sub enqueue {
    my $self = shift;
    lock( @$self );
    push @$self, @_;  
    cond_signal( @$self );
}  # etc...
Best Practices in Perl threads

- Best to abstract shared access into objects
- Avoid having multiple locks at the same time
  - More generally, never do something that might block while you have a lock.
- Minimize shared memory
- Avoid having APIs that will share user variables: copy instead
Beautiful Queues

• Queues turn out to be supremely useful in communication between threads:
  – One thread can pump a thread full of tasks, many can pull from queue and satisfy tasks
  – Workers can shove results into a queue for some reducer/reporter
Thread::Queue

• Thread::Queue, don’t need to build your own
• Also implements
  – non-blocking dequeue()
  – pending()
  – peek()
Graceful degradation

• All of our Queue code (and \texttt{Thread::Queue}) works in non-threaded applications

• \texttt{thread::shared (lock(), share(), :share, cond_*)} degrades meaningfully in the absence of threads

• Can have threading-safe logic in modules
Concurrent Programming

• **Synchronization ‘Primitives’**
  – Semaphores
  – Read/Write Locks
  – Thread Barriers
  – Critical Sections/Mutexes
  – Spinlocks

• None are provided as core primitives, but can be constructed from what we are given
A Helpful Exercise

• Concurrent programming requires different mental model
• Execution environment is not necessarily consistent
• Race conditions are very hard to find in testing
• I found it a very helpful exercise to carefully implement generic synchronization mechanisms
Semaphores

• The classic synchronization primitive [Dijkstra74]

• A type with an integer value, and two operations: up and down
  – up() increases the value
  – down() decreases the value, or blocks until the value can be decreased
Building a Semaphore

package Semaphore;

sub new {
    my ( $class, $initial ) = @_;  
    my $val :shared = $initial;
    return bless \$val, $class;
}

sub up {
    my $self = shift;
    lock( $$self );
    $$self++;
    cond_signal( $$self );
}

sub down {
    my $self = shift;
    lock( $$self );
    cond_wait( $$self ) while ( $$self <= 0 );
    $$self--;
Thread::Semaphore

- CPAN implementation: **Thread::Semaphore**
- Very useful for allocating from resource pools, particularly collaborative resources
- Dining Philosophers
Read/Write Locks

• Resource guarded by a read write lock can:
  – Have many simultaneous readers, or
  – One writer.

• Readers block until no writer
• Writer blocks until no readers
• Useful for compound objects
  – Don’t want to read from something in an inconsistent state
sub new {
    my ( $class ) = @_;  
    my %self :shared = (  
        readers => 0,  
        writers => 0,  
        writers_waiting => 0,  
    );
    return bless \%self, $class;
}

sub read_lock {
    my $self = shift;
    lock( \$self );
    cond_wait( \$self )
        until ( $self->{writers_waiting} == 0  
            and $self->{writers} == 0 );
    $self->{readers}++;
}
Read/Write locks con’t

sub read_release {
    my $self = shift;
    lock( %$self );
    $self->{readers}--;
    cond_signal( %$self )
        if ( $self->{writers_waiting} > 0 );
}

sub write_lock {
    my $self = shift;
    lock( %$self );
    $self->{writers_waiting}++;
    cond_wait( %$self )
        until ( $self->{readers} == 0
            and $self->{writers} == 0 );
    $self->{writers_waiting}--;
    $self->{writers}++;
}
sub write_release {
    my $self = shift;
    lock( %$self );
    $self->{writers}--;
    cond_broadcast( %$self );
}

Using Read/Write Locks

# in a worker thread
$wrlock->read_lock();
# make a decision based on complex object
$wrlock->read_release();

# in some maintenance thread
$wrlock->write_lock();
# incrementally update object
$wrlock->write_release();

• Common pattern where you have lots of readers, occasional updates
Critical Sections and Claims

- **Critical section**: only one thread can execute a section of code at a time.
- Can make claim objects for scoped claims
- Trivially implemented with Semaphores, won’t illustrate implementation
Critical Section usage

```perl
my $cs = CriticalSection->new();

sub some_non_reentrant_sub
{
    my $claim = CriticalSection::Claim->new( $cs );

    # update some file
    # claim’s destructor release critical section
}
```

- Common pattern where you are altering a static file, or using some non-threadsafe code.
Thread::CriticalSection

• There is a beta module on CPAN, with a different approach:

```perl
my $cs = Thread::CriticalSection->new;
$cs->execute(
    sub {
        # your code is protected by $cs
    }
);
```

• Probably a much Perl-ier API
Thread Barriers

• A call that blocks until a certain threshold of waiters is met

```perl
my $barrier = Barrier->new( 5 );

# ...
# thread initialization

if ( $barrier->wait() ) {
    print "All set, starting!\n";
}
# ...
```
Thread::Barrier

- CPAN module with more or less that interface
- Unblocks all, but returns true for the thread that hit the threshold, so you can execute code once
- Barriers can be re-used, thresholds adjusted
- Likely most useful to block all threads of a class
Spinlocks

• Tight loop test and set:
  
  1 until my_try_lock( $lock );

• Common in OS context, make little sense in user threads

• Mostly used for creating tiny critical sections in kernel structures in MP systems
  – Want to wait, but don’t want to release CPU
Why use Threads?

• **Cons:**
  - Not lightweight
  - Shared memory very expensive
  - Very hard to debug
  - Less portable
  - Suck the cost of context switching
  - Issues with XS modules
  - Core dumps and instability
  - Way slower than multiplexing
  - Threaded Perl is a compile-time option
Why use Threads? (cont)

• **Pros:**
  - Communication/sharing easier than between processes
  - Take advantage of multi-core systems
  - Easier to use an existing module in threads than to implement with non-blocking IO and multiplexing
    • Assuming module is thread safe
  - Sort of neat*
What’s on CPAN?

• **Thread::Pool** and friends
  – Cool idea – worker pools with monitored queues, snapshots, collectors, etc.
  – Poorly executed.

• Various synchronization methods discussed

• Some task-specific things, like threaded DNS resolvers

• Not much in the way of usable frameworks
List of Mentioned Modules

• Core Modules:
  – threads
  – threads::shared
  – Thread::Queue
  – Thread::Semaphore

• Non-core CPAN modules:
  – Thread::RWLock
  – Thread::CriticalSection
  – Thread::Barrier
References and Useful Links

• Core threads module docs
  – http://perldoc.perl.org/threads.html

• Core threads::shared module docs
  – http://perldoc.perl.org/threads/shared.html

• perlthrtut threads tutorial
  – http://perldoc.perl.org/perlthrtut.html
  – This is an extensive and strongly recommended doc