Getting Started with Multithreaded Perl

Walt Mankowski (walt@cs.drexel.edu)
Drexel University

YAPC::EU 2009
5 August 2009
Why threaded Perl?
The number of transistors on a chip doubles about every two years.

Why threads?
Why threads?

- CPUs not getting any faster, but...
Why threads?

- CPUs not getting any faster, but...
- CPUs continue to get smaller and more numerous, so...
Why threads?

• CPUs not getting any faster, but…
• CPUs continue to get smaller and more numerous, so…
• To run faster, need to use more than 1 CPU.
Parallel Processing
Parallel Processing

pthreads
Parallel Processing

pthreads

MPI
Parallel Processing

pthreads

MPI

OpenMP
Parallel Processing

pthreads
MPI
OpenMP
Linda
Parallel Processing

pthreads
MPI
OpenMP
Linda
Java threads
Parallel Processing

pthreads
MPI
OpenMP
Linda
Java threads
shell pipeline
Parallel Processing

pthreads
MPI
OpenMP
Linda
Java threads
shell pipeline

fork()
Parallel Processing

- pthreads
- MPI
- OpenMP
- Linda
- Java threads
- shell pipeline
- fork()
- semaphores
Parallel Processing

- pthreads
- MPI
- OpenMP
- Linda
- Java threads
- shell pipeline

- fork()
- semaphores
- shared memory
Parallel Processing

- pthreads
- MPI
- OpenMP
- Linda
- Java threads
- shell pipeline
- fork()
- semaphores
- shared memory
- pipes
Parallel Processing

- pthreads
- MPI
- OpenMP
- Linda
- Java threads
- shell pipeline
- fork()
- semaphores
- shared memory
- pipes
- sockets
Parallel Processing

- pthreads
- MPI
- OpenMP
- Linda
- Java threads
- shell pipeline
- fork()
- semaphores
- shared memory
- pipes
- sockets
- etc…
Threads vs Processes
Threads vs Processes

In general (not necessarily in Perl) …
Threads vs Processes

In general (not necessarily in Perl) …

- threads lighter-weight than processes
Threads vs Processes

In general (not necessarily in Perl) …

- threads lighter-weight than processes
- faster to create, faster to switch
Threads vs Processes

In general (not necessarily in Perl) …

• threads lighter-weight than processes
• faster to create, faster to switch
• easier to share data between threads
Threads vs Processes

In general (not necessarily in Perl) …

• threads lighter-weight than processes
  • faster to create, faster to switch
  • easier to share data between threads
  • shared data in same memory location
Threads vs Processes

In general (not necessarily in Perl) …

- threads lighter-weight than processes
- faster to create, faster to switch
- easier to share data between threads
- shared data in same memory location
- harder to debug threads
Threads vs Processes

In general (not necessarily in Perl) …

- threads lighter-weight than processes
- faster to create, faster to switch
- easier to share data between threads
- shared data in same memory location
- harder to debug threads
- race conditions, locks, randomness
A Little History
A Little History

Threads.pm
A Little History

Threads.pm

• introduced in 5.005
A Little History

Threads.pm

- introduced in 5.005
- sucked
A Little History

Threads.pm

• introduced in 5.005
• sucked
• deprecated
A Little History

Threads.pm

- introduced in 5.005
- sucked
- deprecated
- removed in 5.10
A Little History
A Little History

threads.pm
A Little History

threads.pm

• introduced in 5.8
A Little History

threads.pm

• introduced in 5.8
• improved in 5.10
A Little History

threads.pm

• introduced in 5.8
• improved in 5.10
• "interpreter threads" (ithreads)
A Little History

threads.pm

• introduced in 5.8
• improved in 5.10
• "interpreter threads" (ithreads)
• separate Perl interpreter for each threads
Creating threads
#!/usr/bin/perl -w
use strict;
use threads;

my $t1 = threads->new(&tsub);
my $t2 = threads->new(&tsub);

$t1->join;
$t2->join;

sub tsub {
    print "I like pie\n";
}

#!/usr/bin/perl -w
use strict;
use threads;

my $t1 = threads->new(&tsub);
my $t2 = threads->new(&tsub);

$t1->join;
$t2->join;

sub tsub {
  print "I like pie\n";
}

#!/usr/bin/perl -w
use strict;
use threads;

my $t1 = threads->new(&tsub);
my $t2 = threads->new(&tsub);

$t1->join;
$t2->join;

sub tsub {
    print "I like pie\n";
}
#!/usr/bin/perl -w
use strict;
use threads;  # as early as possible

my $t1 = threads->new(\&tsub);
my $t2 = threads->new(\&tsub);

$t1->join;
$t2->join;

sub tsub {
    print "I like pie\n";
}
#!/usr/bin/perl -w
use strict;
use threads;

my $t1 = threads->new(\&tsub);
my $t2 = threads->new(\&tsub);
$t1->join;
$t2->join;

sub tsub {
    print "I like pie\n";
}
#!/usr/bin/perl -w
use strict;
use threads;

my $t1 = threads-&gt;create(&tsub);
my $t2 = threads-&gt;create(&tsub);
$t1-&gt;join;
$t2-&gt;join;

sub tsub {
    print "I like pie\n";
}
#!/usr/bin/perl -w
use strict;
use threads;

my $t1 = threads->new("tsub");
my $t2 = threads->new("tsub");

$t1->join;
$t2->join;

sub tsub { 
    print "I like pie\n";
}

#!/usr/bin/perl -w
use strict;
use threads;

my $t1 = threads->new(&tsub);
my $t2 = threads->new(&tsub);

$t1->join;
$t2->join;

sub tsub {
    print "I like pie\n";
}

#!/usr/bin/perl -w
use strict;
use threads;

my $t1 = threads->new(&tsub);
my $t2 = threads->new(&tsub);

$t1->detach;
$t2->detach;

sub tsub {
    print "I like pie\n";
}
#!/usr/bin/perl -w
use strict;
use threads;

threads->new(&tsub)->detach;
threads->new(&tsub)->detach;

sub tsub {
    print "I like pie\n";
}

#!/usr/bin/perl -w
use strict;
use threads;

my $t1 = threads->new(&tsub);
my $t2 = threads->new(&tsub);

$t1->join;
$t2->join;

sub tsub {
    print "I like pie\n";
}
Anonymous subs
Anonymous subs

my $t1 = threads->new(
    sub {print "I like pie\n";}
);

Anonymous subs

my $t1 = threads-&gt;new(
    sub {print "I like pie\n";}
);

my $t2 = threads-&gt;new(
    sub {print "I like cake\n";}
);
Passing parameters

my $t1 = threads->new(&tsub, "apple");
my $t2 = threads->new(&tsub, "cherry");
$t1->join;
$t2->join;

sub tsub {
    my ($fruit) = @_; 
    print "I like $fruit pie\n";
}
Returning from a thread

my $t1 = threads->new(&tsub, "apple");
my $t2 = threads->new(&tsub, "cherry");

my $len1 = $t1->join;
print "thread 1 returned $len1\n";
my $len2 = $t2->join;
print "thread 2 returned $len2\n";

sub tsub {
    my ($fruit) = @_;  
    print "I like $fruit pie\n";
    return length $fruit;
}

Return context
Return context

# scalar context
Return context

# scalar context
my $thr = threads->new(...);
Return context

# scalar context
my $thr = threads->new(...);

# list context
Return context

# scalar context
my $thr = threads->new(...);

# list context
my ($thr) = threads->new(...);
Return context

# scalar context
my $thr = threads->new(...);

# list context
my ($thr) = threads->new(...);

# void context
Return context

# scalar context
my $thr = threads->new(...);

# list context
my ($thr) = threads->new(...);

# void context
threads->new(...);
Return context

# scalar context
my $thr = threads->new(...);

# list context
my ($thr) = threads->new(...);

# void context
threads->new(...);

# inside thread
Return context

# scalar context
my $thr = threads->new(...);

# list context
my ($thr) = threads->new(...);

# void context
threads->new(...);

# inside thread
if (threads->wantarray()) {...}
Are threads enabled?
Are threads enabled?

- threads not compiled into Perl by default
Are threads enabled?

- threads not compiled into Perl by default
- many (most?) distributions build Perl with threads
Are threads enabled?

- threads not compiled into Perl by default
- many (most?) distributions build Perl with threads
  - Debian
Are threads enabled?

• threads not compiled into Perl by default
• many (most?) distributions build Perl with threads
  - Debian
  - Ubuntu
Are threads enabled?

- threads not compiled into Perl by default
- many (most?) distributions build Perl with threads
  - Debian
  - Ubuntu
  - OS X
Are threads enabled?
Are threads enabled?

from command line…

• perl -V

• look for “useithreads=define”
Are threads enabled?

from command line…

• perl -V

• look for “useithreads=define”

from program…
Are threads enabled?

from command line...
- perl -V
- look for "useithreads=define"

from program...

use Config;
...
if ($Config{useithreads})...
Sharing Data
Sharing basics
Sharing basics

• all variables are *private* by default
Sharing basics

• all variables are *private* by default
• variables must be *explicitly shared*
Sharing basics

• all variables are **private** by default
• variables must be **explicitly shared**
• thread creation relatively expensive
Explicitly Sharing

use threads;
Explicitly Sharing

use threads;
use threads::shared;
Explicitly Sharing

use threads;
use threads::shared;

# compile time
my $foo : shared;
Explicitly Sharing

use threads;
use threads::shared;

# compile time
my $foo :shared;

# run time
my $bar;
share($bar);
What can be shared?
What can be shared?

scalars
What can be shared?

- scalars
  - my $foo : shared;
What can be shared?

- scalars
  - my $foo : shared;
- arrays
What can be shared?

- scalars
  - my $foo : shared;
- arrays
  - my @bar : shared;
What can be shared?

- scalars
  - my $foo : shared;
- arrays
  - my @bar : shared;
- hashes
What can be shared?

scalars
• my $foo : shared;
arrays
• my @bar : shared;
hashes
• my %baz : shared;
Restrictions on shared variables

basic rule – a shared variable can't store a reference to an unshared variable
Restrictions on shared variables
Restrictions on shared variables

my ($foo, $bar, $baz);
Restrictions on shared variables

my ($foo, $bar, $baz);
share($foo); share($bar);
Restrictions on shared variables

my ($foo, $bar, $baz);
share($foo); share($bar);

$foo = 42;    # ok
Restrictions on shared variables

my ($foo, $bar, $baz);
share($foo); share($bar);

$foo = 42;     # ok
$foo = "pony"; # ok
Restrictions on shared variables

my ($foo, $bar, $baz);
share($foo); share($bar);

$foo = 42;    # ok
$foo = "pony"; # ok
$foo = $bar;  # ok
Restrictions on shared variables

my ($foo, $bar, $baz);
share($foo); share($bar);

$foo = 42;     # ok
$foo = "pony"; # ok
$foo = $bar;   # ok
$foo = $baz;   # ok
Restrictions on shared variables

```perl
my ($foo, $bar, $baz);
share($foo); share($bar);

$foo = 42;      # ok
$foo = "pony";  # ok
$foo = $bar;    # ok
$foo = $baz;    # ok
$foo = \$bar;   # ok
```
Restrictions on shared variables

my ($foo, $bar, $baz);
share($foo); share($bar);

$foo = 42;    # ok
$foo = "pony"; # ok
$foo = $bar;  # ok
$foo = $baz;  # ok
$foo = \$bar; # ok
$foo = \$baz; # ERROR
Locking
my $n : shared = 0;
my $t1 = threads->new(\&tsub);
my $t2 = threads->new(\&tsub);

$t1->join;
$t2->join;

print "\$n = $n\n";

sub tsub {
    for (1..100_000) {
        $n++;
    }
}
my $n :shared = 0;
my $t1 = threads->new(&tsub);
my $t2 = threads->new(&tsub);

$t1->join;
$t2->join;

print "\$n = $n\n";

sub tsub {
    for (1..100_000) {
        $n++;
        # RACE CONDITION
    }
}
use threads::shared;
my $foo :shared;
...
{
    lock($foo);
    # $foo is locked until end of block
}
# $foo is now unlocked
my $n : shared = 0;
my $t1 = threads->new(\&tsub);
my $t2 = threads->new(\&tsub);

$t1->join;
$t2->join;

print "\$n = $n\n";

sub tsub {  
    for (1..100_000) {  
        lock($n);
        $n++;  
        $n++;  
    }  
}
Lock caveats
Lock caveats

- locks are only advisory
Lock caveats

- locks are only advisory
- can't lock elements of a container:
Lock caveats

- locks are only advisory
- can't lock elements of a container:

```php
lock($foo{bar});  # error
```
Lock caveats

- locks are only advisory
- can't lock elements of a container:

  ```
  lock($foo{bar});  # error
  lock(%foo);       # works
  ```
Semaphores
Thread::Semaphore

Compared to locks:

- not tied to scalars/containers
- not limited by scope
- more flexible than “locked” and “unlocked”
Thread::Semaphore

->new()
->new(N)

• makes new semaphore
• initializes counter to N
• defaults to 1
Thread::Semaphore

- `down()`
- `down(N)`
  - decreases semaphore's counter by N (defaults to 1)
  - blocks if counter would become negative
Thread::Semaphore

- ->up()
- ->up(N)

  • increases semaphore's counter by N (defaults to 1)
  • unblocks threads waiting on down() if possible
use Thread::Semaphore;

my $n :shared = 0;
my $semaphore = Thread::Semaphore->new;
my $t1 = threads->new(&tsub);
my $t2 = threads->new(&tsub);

$t1->join;
$t2->join;

sub tsub { 
    for (1..100_000) { 
        $semaphore->down;
        $n++;
        $semaphore->up;
    }
}
# n readers, 1 writer

my $READERS = 5;
my $sem = new Thread::Semaphore($READERS);
my $t1 = threads->new(\&tsub);
my $t2 = threads->new(\&tsub);

$t1->join;
$t2->join;

sub tsub {
    if ($do_read) {
        $sem->down(1); read(); $sem->up(1);
    } elsif ($do_write) {
        $sem->down($READERS);
        write();
        $sem->up($READERS);
    }
Queues
Thread::Queue
Thread::Queue

• thread-safe FIFO queues
Thread::Queue

- thread-safe FIFO queues
- any number of threads
Thread::Queue

- thread-safe FIFO queues
- any number of threads
- can pass any datatype supported by threads::shared
Thread::Queue

- thread-safe FIFO queues
- any number of threads
- can pass any datatype supported by threads::shared
- useful for implementing thread pools
my $q = new Thread::Queue;
$q->enqueue($_) for 1..100_000;

my $t1 = threads->new(&tsub);
my $t2 = threads->new(&tsub);

$t1->join;
$t2->join;

sub tsub {
    while (my $x = $q->dequeue) {
        # process $x
    }
}
my $q = new Thread::Queue;
$q->enqueue($_) for 1..100_000;

my $t1 = threads->new(&tsub);
my $t2 = threads->new(&tsub);

$t1->join;
$t2->join;

sub tsub {
  while (my $x = $q->dequeue) {
    # process $x
  }
}
my $q = new Thread::Queue;
$q->enqueue($_) for 1..100_000;

my $t1 = threads->new(&tsub);
my $t2 = threads->new(&tsub);
$t1->join;
$t2->join;

sub tsub {
    while (my $x = $q->dequeue) {
        # process $x
    }
}
my $q = new Thread::Queue;
$q->enqueue($_) for 1..100_000;
$q->enqueue(undef);
$q->enqueue(undef);
my $t1 = threads->new(&tsub);
my $t2 = threads->new(&tsub);

$t1->join;
$t2->join;

sub tsub {
    while (my $x = $q->dequeue) {
        # process $x
    }
}
Thread IDs
Thread IDs

In main thread:
Thread IDs

In main thread:

```perl
my $t = threads->new(&foo);
my $tid = $t->tid();
```
Thread IDs

In main thread:

my $t = threads->new(&foo);
my $tid = $t->tid();

Inside thread:
Thread IDs

In main thread:

my $t = threads->new(&foo);
my $tid = $t->tid();

Inside thread:

# class method
my $tid = threads->tid();
Thread list
Thread list

Call class method threads->list().
Thread list

Call class method threads->list().
Handy for cleanup:
Thread list

Call class method threads->list().
Handy for cleanup:
$_->join for threads->list();
Coding with threads
Threads suck

- hard to debug
- program flow indeterminate
- limit to possible speedup
- hard to code
- threads get in the way of algorithm
Java Threads
Java Threads

• java threads suck, too
Java Threads

- java threads suck, too
- not terribly different from perl threads
Java Threads

• java threads suck, too
• not terribly different from perl threads
• but java programmers use threads all the time
Perl Threads Suck
Perl Threads Suck

- Memory usage
Perl Threads Suck

- Memory usage
  - no copy-on-write
Perl Threads Suck

- Memory usage
  - no copy-on-write
- Many XS modules not thread-safe
Perl Threads Suck

- Memory usage
  - no copy-on-write
- Many XS modules not thread-safe
- Core not designed to be thread-safe
Perl Threads Suck

- Memory usage
  - no copy-on-write
- Many XS modules not thread-safe
- Core not designed to be thread-safe
  - unlike JVM
Perl Threads Suck

- Memory usage
  - no copy-on-write
- Many XS modules not thread-safe
- Core not designed to be thread-safe
  - unlike JVM
- forks.pm
Perl Threads Suck

- Memory usage
  - no copy-on-write
- Many XS modules not thread-safe
- Core not designed to be thread-safe
  - unlike JVM
- forks.pm
  - same syntax as threads.pm, but uses fork instead
double tot = 0;
for (double i = 0; i <= 1; i += 1e-7) {
    tot += f(i);
}
#pragma omp parallel reduction(+:tot)
double tot = 0;
for (double i = 0; i <= 1; i += 1e-7) {
    tot += f(i);
}
More Information

Perl threads:

• perlthrtut
• threads
• threads::shared
• Thread::Queue
• Thread::Semaphore
More Information

pthreads:

- Pthreads Programming (O'Reilly)
- https://computing.llnl.gov/tutorials/pthreads/
More Information

OpenMP

- https://computing.llnl.gov/tutorials/openMP/
More Information

MPI

- https://computing.llnl.gov/tutorials/mpi/
More Information

Future of Parallel Computing

- "Computer Hardware Is Back"
  David Patterson, iTunes U, Stanford Computer Systems Colloquiums, Winter 2007
Thank you!