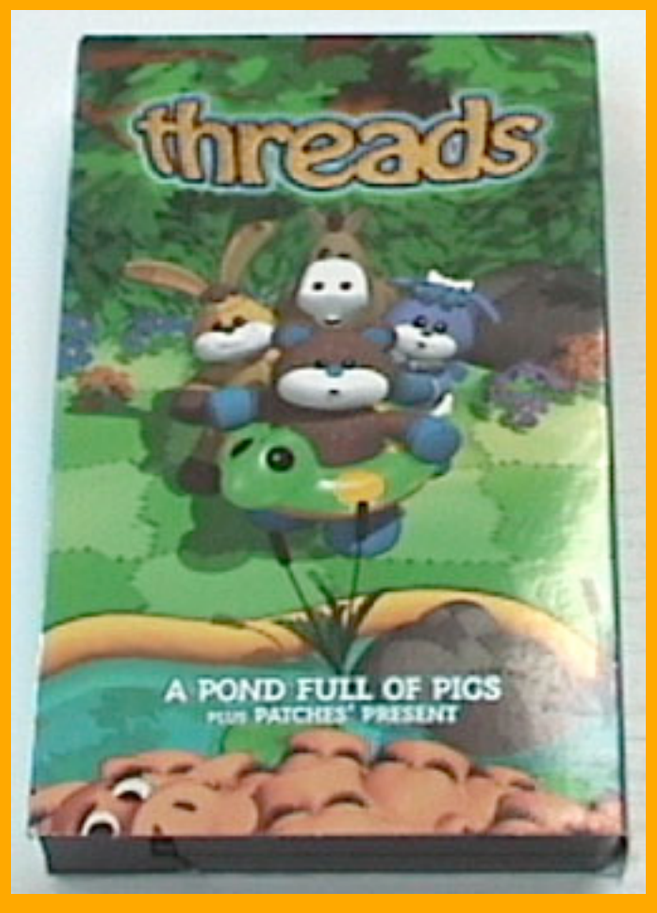
Threading in Perl 5.8(.1)?



Arthur ‘sky’ Bergman
abergman@fotango.com
opensource.fotango.com

Who?

- Arthur Bergman
- abergman@fotango.com
- www.nanisky.com
- perl5-threading-p(orter|imp)
- Fotango, YAPC Sponsor

What will we learn?

- What is threading
- How do we use threading
- Why do we use threading
- Thread safe modules (and XS)

What is threading

DIFFICULT

Different style of thinking

No seriously

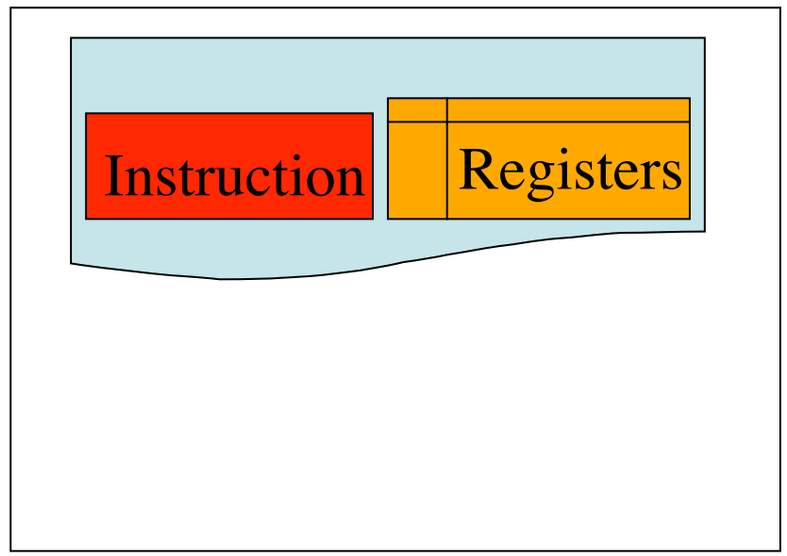
What is a thread

- ‘Anything suggestive of the continuousness and sequence of thread’
  -- American Heritage dictionary

- ‘A set of properties that suggest “continuousness and sequence” within the computer’

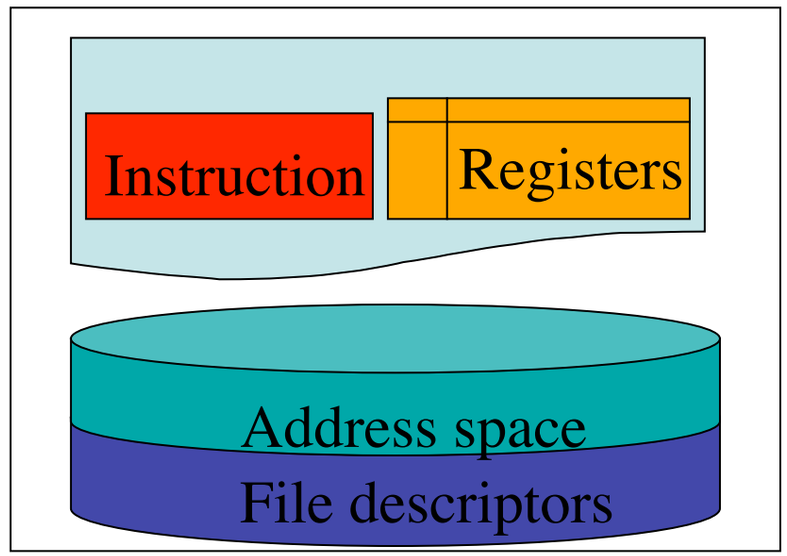
the thread

- A set of instructions
- A set of address and data registers
- The current instruction
- Attached to a process



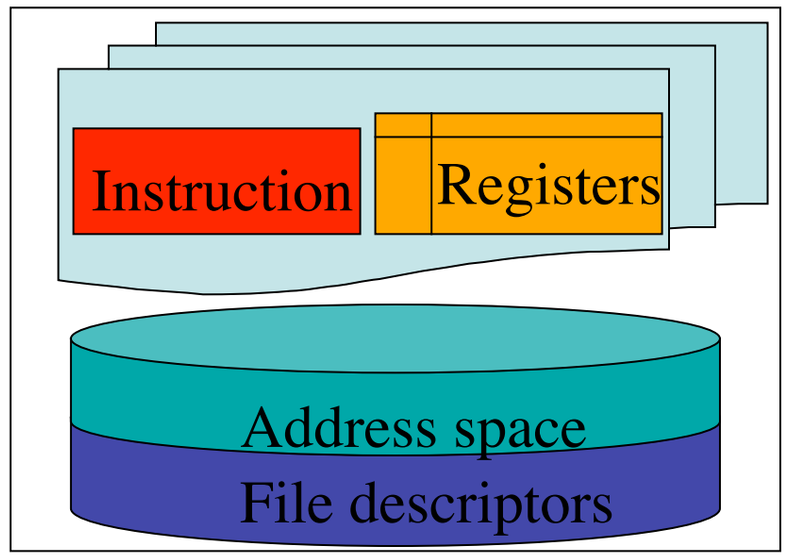
the process

- A set of instructions
- A set of address and data registers
- The current instruction
- An address space
- File descriptors
- Meta data



the multithreaded process

- Several threads running independently
  - An address space
  - File descriptors
  - Meta data



In perl speak

- Several perl interpreters in the same process
  - Each thread gets it's own interpreter
  - Each interpreter is like a plain old perl interpreter

Clone, just like fork

- We start with a main interpreter
- We then clone the main interpreter
  - Each threads interpreter is a clone of a previously existing interpreter
- Cloning is close to forking
  - Except we don't use the operating system
- In fact, on Win32, fork is implemented using clone, the so called pseudo-fork

NO MEMORY SHARING

Enough, how is it done

- Need a perl 5.8 configured for threading
- Win32 usually is so you should be safe
- Configure -Dusethreads -Duseithreads
  - Jaguar users, add -Dusereentrant
- Works on Pthreads, oldPthreads, Win32 and Netware threads.

?threads?

- Different kinds of threads
- Perl has perl threads
- Not
  - pthreads
  - zthreads
  - win32threads
  - mach threads
- All similarity is accidental

How do we use threading

A simple example

```
use threads;
sub my_sub {
  for(1..$_[0]) {
    print "Hi, I am thread ", threads->tid,
    "\n";
    sleep 1;
  }
}
my $thr1 = threads->create(\&my_sub, 10);
my $thr2 = threads->create(”my_sub”, 10);
threads->new(sub {print “hi” })->join();
$thr1->join(); $thr2->join();
exit;
```

```
threads->create(\&my_sub, 10);
```

- Creates a thread
- Passes the argument 10 to the thread
- Starts executing the thread in my_sub()
  - Subroutine reference

```
threads->create("my_sub", 10);
```

- Creates a thread
- Passes the argument 10 to the thread
- Starts executing the thread in my_sub()
  - Subroutine reference
  - Named subroutine

```
threads->new(sub { })->join
```

- Creates a thread
- Passes the argument 10 to the thread
- Starts executing the thread in my_sub()
  - Subroutine reference
  - Named subroutine
  - Anonymous subroutine

```
threads->new(sub { })->join
```

- Creates a thread
- Passes the argument 10 to the thread
- Starts executing the thread in my_sub()
  - Subroutine reference
  - Named subroutine
  - Anonymous subroutine
- new is an alias to create
  - For historic reasons

```
my $thread = threads->new(sub
```

- Creates a thread
- Passes the argument 10 to the thread
- Starts executing the thread in my_sub()
  - Subroutine reference
    - Named subroutine
    - Anonymous subroutine
- new is an alias to create
  - For historic reasons
- Returns a thread object
  - Used to control the thread

The thread object

- Each thread is represented by a an object
- Accessible from all other threads
  - No built in security

The thread object

- Each thread is represented by a an object
- Accessible from all other threads
  - No built in security
- Get it from
  - `$thread = threads->create(`
  - `$thread = threads->self()`
  - `@threads = threads->list()`

The thread object

- Each thread is represented by a an object
- Accessible from all other threads
  - No built in security
- Get it from
  - `$thread = threads->create(`
  - `$thread = threads->self()`
  - `@threads = threads->list()`

The thread object

- Each thread is represented by a an object
- Accessible from all other threads
  - No built in security
- Get it from
  - `$thread = threads->create(`
  - `$thread = threads->self()`
  - `@threads = threads->list()`

A simple example

```
use threads;
sub my_sub {
 for(1..$_[0]) {
 print "Hi, I am thread ", threads->tid,
"\n";
 sleep 1;
 }
}
my $thr1 = threads->create(\&my_sub, 10);
my $thr2 = threads->create(”my_sub”, 10);
threads->new(sub {print “hi” })->join();
$thr1->join(); $thr2->join();
exit;
```

join

```
$thread = $thread->join();
```

- Gets the return value
- Waits for the thread to return
- Tells perl it can destroy this thread
  - You can only do this once

detach

```
$thread->detach();
```

- If you don't care about the thread
- As soon as the thread is done it will go away
  - You can only do this once

Importance of cleaning

- Memory leaks
- Unclean shutdown
- `END {$->join()for(thread->lists)}`
  - Still won't get detached threads
- Might get

  "A thread exited while 2 threads were running."

Basic threading

- create
  - a new thread
  - data to the thread
- join
  - a thread is done
  - get data back
- detach
  - we don’t care

Warning

Temporally uncertain programming

Importance of cleaning

- Memory leaks
- Unclean shutdown
- `END {$->join()for(thread->lists)}`
  - Still won't get detached threads
- Might get

  "A thread exited while 2 threads were running."

Might get

- Might?
- This makes threads hard
- Humans can usually do one thing at a time
- Computers can do many
- Processes do one thing at the time
  - Hence humans can understand processes
- Threaded programs do multiple things
  - Humans are totally lost

- Never assume a thread you create will wait for you
- Each thread might go to sleep, at any arbitrary point, for an unbounded period of time
- There is no order
- All apparent order is random
  - CPU
  - OS
  - Weather
  - Gods

NEVER TRUST THREAD INERTIA

"Threads will run in the most evil order possible" -- Bill Gallmeister

Unshared data

```
my $foo = 1;
threads(sub {$foo++ })->join();
print "$foo\n";
```

- Results in

  1

- Data is not shared by default

Shared data

```
use threads::shared;
my $foo :shared = 1;
threads(sub {$foo++ })->join();
print "$foo\n";
```

- Results in

  2

- Attribute declares shared values

threads::shared

- `my $scalar : shared;`
- `my @array : shared;`
- `my %hash : shared;`
- `my %hash; share(%hash);`
- share() is a runtime alias to ':shared'
  - use for pre 5.8 compatible code
- `my $ref = &share({})`

threads::shared

- `my $scalar : shared;`
- `my @array : shared;`
- `my %hash : shared;`
- `my %hash; share(%hash);`
- share() is a runtime alias to ':shared'
  - use for pre 5.8 compatible code
- `my $ref = &share({})`
  - Must be used to defeat faulty prototypes

threads::shared

- share() is not currently recursive
  - `my $foo = share({hi => bar})` does not work
  - Might become recursive in future versions
- Does not work on code, packages or io
- Implemented partly using tie, so you can't tie shared variables
- Object destructors will get run in every thread

Object destructors

```
use threads;
use threads::shared;
my $object = bless &share({});
sub DESTROY { print "bye bye\n" }
async {$object; }->join();
async {$object; }->join();
```

- Gives us five “bye bye”

Object destructors

```
use threads;
use threads::shared;
my $object = bless &share({});
sub DESTROY { print "bye bye\n" }
async {$object; }->join();
async {$object; }->join();
```

- Gives us five “bye bye”
- async is a handy shortcut for

```
threads->new(sub {});
```

Object destructors

```
use threads;
use threads::shared;
my $object = bless &share({});
sub DESTROY { print "bye bye\n" }
async {$object; }->join();
async {$object; }->join();
```

- Gives us five “bye bye”

Object destructors

```
use threads;
use threads::shared;
my $object = bless &share({});
sub DESTROY { print "bye bye\n" }
async {$object; }->join();
async {$object; }->join();
```

- Gives us five “bye bye”
  - A subroutine returns it’s last statement

Object destructors

```
use threads;
use threads::shared;
my $object = bless &share({});
sub DESTROY { print "bye bye\n" }
async {$object; }->join();
async {$object; }->join();
```

- Gives us five “bye bye”
  - A subroutine returns it’s last statement
  - `join` retrieves it

Object destructors

```
use threads;
use threads::shared;
my $object = bless &share({});
sub DESTROY { print "bye bye\n" }
async {$object; }->join();
async {$object; }->join();
```

- Gives us five “bye bye”
  - A subroutine returns it’s last statement
  - `join` retrieves it
  - Nothing wants it, so it gets destroyed

Object destructors

```
use threads;
use threads::shared;
my $object = bless &share({});
sub DESTROY { print "bye bye\n" }
async {$object; 0}->join();
async {$object; 0}->join();
```

- Gives us three “bye bye”
- Slightly better
- Still not ok for all cases

Dirty hack

```
use threads;
use threads::shared;
my $object = bless &share({});
sub DESTROY {
  print "bye bye\n"
    if(threads::shared::_refcnt(%{$_[0]})
    == 1);
}
async {$object; 0}->join();
async {$object; 0}->join();
```

- Gives us one “bye bye”
- “Hidden” feature
- Gives the process global refcount

Shared data are ‘proxies’

```
use threads;
use threads::shared;
my %data : shared;
my $data_item = &share([]);
$data{item } = $data_item;
print "$data{item } != $data_item\n";
$data{item }->[0] = 'hi';
print "$data{item }->[0] eq
   $data_item->[0]\n";
```

- Same shared data can be represented by different front end data objects

Create order

Chaos

```
use threads;
use threads::shared;
my $i : shared;
async {for(1..100000){
  $i++
  }}for(1..10);
$_->join for(threads->list);
print "$i\n";
```

- What will this print?

Chaos

```
use threads;
use threads::shared;
my $i : shared;
async {for(1..100000){
  $i++
  }}for(1..10);
$_->join for(threads->list);
print "$i\n";
```

- What will this print?

Chaos

```
use threads;
use threads::shared;
my $i : shared;
async {for(1..100000){
  $i++
  }}for(1..10);
$_->join for(threads->list);
print "$i\n";
```

- What will this print?

Anything between 100000 - 1000000

What is `$i++`

- temp = $i
- $i = temp + 1
- If $i is 2
- 2 = 2
- $i = 2 + 1
- Result is of course 3

Threads change this

- t1: temp = $i
- t2: temp = $i
- t1: temp is 2
- t2: temp is 2
- t1: $i = temp + 1 (2 + 1)
- t2: $i = temp + 1 (2 + 1)
- $i is 3, not 4 as it should be!

`lock()`

- Takes a shared variable
- Locks it for this scope
- Advisory locking only
- If variable is locked, it will wait for it to be unlocked
- Be careful when locking at file scope level, might not unlock before end of program

Order

```
use threads;
use threads::shared;
my $i : shared;
async {for(1..100000){
  lock($i); $i++
  }}for(1..10);
$_->join for(threads->list);
print "$i\n";
```

- This will give us 1000000

File scope locking

```
use threads;
use threads::shared;
my %data : shared;
lock(%data);
my $thread = threads->new( sub {
lock(%data) });
$thread->join();
```

- This deadlocks

File scope locking

```
use threads;
use threads::shared;
my %data : shared;
lock(%data);
my $thread = threads->new( sub {
lock(%data) });
$thread->join();
```

- This deadlocks
- Outer level scope is never left, no unlock

File scope locking

```
use threads;
use threads::shared;
my %data : shared;
{
  lock(%data);
  my $thread = threads->new( sub {
  lock(%data) });
}
$thread->join();
```

- This deadlocks
- This works

Conditions

- Lock is for locking not waiting
- Use
  - cond_wait
  - cond_signal
  - cond_broadcast

Let us burn some CPU

```
my $i : shared;
async {
  $i = sleep int rand 10 || -1;
}
until($i != 0) {};
print “slept $i\n”;
```

- The beginners first naïve attempt

A second attempt

```
my $i : shared;
{lock($i);
async {
  lock($i);
  $i = sleep int rand 10 || -1;
}}
lock($i);
print “slept $i\n”;
```

- Better approach
- But a bit unclear and faulty
- What if the thread hasn’t begun execution before the second lock?

The good approach

```
my $i : shared;
async {lock($i);
  $i = sleep int rand 10;
  cond_signal($i);
}
lock($i);
cond_wait($i);
print "$i\n";
```

- Use cond_wait to wait for the signal
- Use cond_signal to signal the waiting threads

The perfect approach

```
my $i : shared;
async {lock($i);
  $i = sleep int rand 10;
  cond_signal($i);
}
lock($i);
cond_wait($i) until($i);
print "$i\n";
```

- Use cond_wait to wait for the signal
- Use cond_signal to signal the waiting threads
- Double check the value

cond_wait

- Takes a locked shared variable
- Unlocks the variable
- Blocks at most until it receives a signal
- Can wake up at any time
- Reacquires the lock before proceeding

cond_signal

- Takes a shared variable
  - Preferably locked
- Sends a signal to at least one waiting thread
- Can signal more than one waiting thread

cond_broadcast

- Takes a shared variable
  - Preferably locked
- Sends a signal to all waiting threads
- All threads will then reacquire the lock in a first come first served order
- cond_signal is a cond_broadcast optimisation to avoid the thundering hurd problem

Broadcast

```
my $i : shared;
async {
  lock($i);
  cond_wait($i);
  print “Hello I am thread: “ .
  threads->tid . “\n”;
}
sleep 1;
cond_broadcast($i);
```

threads

- create (new)
- join
- detach
- self
- tid (threads->tid == threads->self->tid)
- list

threads::shared

- : shared
- share()
- &share()
- lock
- cond_wait
- cond_signal
- cond_broadcast

Why do we use threading

We want to do multiple things at the same time!

The competition

- Event loops
  - POE.pm
  - Event.pm
    - Gtk/Tk/Your GUI
- Coroutines
- Fork + shared memory

Unique Selling Points

- Can use that expensive extra processor
- Doesn't block on File IO
- Portable
- Lets you share data
- Easy

Can use that extra CPU

- Fork is the only other solution that lets you do this
  - Forking is messy, non portable and makes it very hard to share data
- POE will get SMP support soon
  - But then it will just use threads

Doesn't block on File IO

- Many solutions support nonblocking IO
  - Only for sockets
- Only threads lets your program continue while you block waiting for your file IO

Portable

- We work very hard to support multiple platforms
- All platforms should act the same
- Some platforms sadly still have very buggy threads, but we try hard
- Recently patches were applied for z/OS !

Shared data

- One process
- All data is accessible without a speed hit
- File handles can be used by all threads

Easy

- At least compared to the alternatives
- Fork is just confusing
- Event loops requires you to think backwards
- All other solutions also have race conditions

A simple example

```
use threads;
use IO::Socket;
my $listen = IO::Socket::INET->new(
 LocalPort => 4545,
 ReuseAddr => 1,
 Listen    => 10,
)
sub handle_connection {
 my $socket = shift;
 my $output = shift || $output;
 print $output “$_” while(<$socket>);
}
async {while(my $socket = $listen->accept) {
 threads->new(\&handle_connection, $socket);
};
handle_connection(\*STDIN,\*STDOUT);
```

Thread safe modules

Applications are in control

- Threads are usually created by the applications
- Modules should just work
- Most modules will just work
- Modules usually will not create threads

Perl helps you

- threads::shared exports noops unless threads has been loaded
- Same code branch can thus work for threaded and non threaded code
- A module shouldn't use threads unless threads has been loaded
- Some modules, might of course be written just to use threads

Perl helps you II

- Since no data is shared
- Most data doesn't need to be shared
- Your work to make your module safe is
  - Share the correct data
  - Add locking

Making Test::More threadsafe

- Not needed, Test::More uses Test::Builder
- Special care needed because Test::Builder works all the way back to 5.004
- Test numbers should be shared between all threads

Wrong result by default

```
for(1..10) {
  async {is(1,1) };
}
ok 1
ok 1
ok 1
ok 1
ok 1
ok 1
ok 1
ok 1
ok 1
ok 1
```

Backwards compatible

```
BEGIN {
    use Config;
    if( $] >= 5.008 &&
$Config{useithreads} ) {
        require threads;
        require threads::shared;
        threads::shared->import;
    }
    else {
        *share = sub { 0 };
        *lock  = sub { 0 };
    }
}
```

Share the right variables

```
my $Curr_Test = 0;
my @Test_Results = ();
my @Test_Details = ();
```

Share the right variables

```
my $Curr_Test = 0;
my @Test_Results = ();
my @Test_Details = ();
```

```
share($Curr_Test);
share(@Test_Results);
share(@Test_Details);
```

locking it up

- Identify the uses of the shared variables
- Add locking
- Share any references that gets added to the shared arrays

```
$Curr_Test++;
my $result = {};
...
$Test_Results[$Curr_Test-1] = $result;
```

locking it up

- Identify the uses of the shared variables
- Add locking
- Share any references that gets added to the shared arrays

```
lock($Curr_Test);
$Curr_Test++;
my $result = {};
...
share($result);
$Test_Results[$Curr_Test-1] = $result;
```

One final problem

- Array slices do not auto expand the array!
- Tie/Magic limitation!
- So no autovivification of not run tests

```
my $num_failed = grep !$_->{'ok'},
@Test_Results[0..$Expected_Tests-1];
$num_failed += abs($Expected_Tests -
@Test_Results);
```

Hack to go around the issue

```
for my $idx
($#Test_Results..$Expected_Tests-1) {
  my %empty_result = ();
  share(%empty_result);
  $Test_Results[$idx] = \%empty_result
    unless defined $Test_Results[$idx];
}
```

All done

```
for(1..10) {
  async {is(1,1) };
}
ok 1
ok 2
ok 3
ok 4
ok 5
ok 6
ok 7
ok 8
ok 9
ok 10
```

Modules that help you

- Thread::Semaphore
- Thread::Queue
- Thread::Signal
- Several CPAN modules

Thread::Queue

```
use Thread::Queue;
my $q = new Thread::Queue;
$q->enqueue("foo", "bar");
my $foo = $q->dequeue;
my $bar = $q->dequeue_nb;
my $left = $q->pending;
```

Thread::Semaphore

```
use Thread::Semaphore;
my $s = new Thread::Semaphore;
$s->up;      # Also known as the semaphore
V -operation.
# The guarded section is here
$s->down;    # Also known as the semaphore
P -operation.

# The default semaphore value is 1.
my $s = new
Thread::Semaphore($initial_value);
$s->up($up_value);
$s->down($up_value);
```

Thread::Signal

- For Linux (ab)users
- Safe signals
- Allows you to map signals between different threads

Elizabeth Mattisen

- Thread::Tie
- Thread::Use
- Thread::Need
- Thread::Status
- Thread::Rand
- Thread::Deadlock
- Thread::Pool
- And more.....

Perl even helps you with XS

```
#define MY_CXT_KEY "DB_File::_guts"
XS_VERSION
typedef struct {
    recno_t     x_Value;
    recno_t     x_zero;
    DB_File     x_CurrentDB;
    DBTKEY      x_empty;
} my_cxt_t;
START_MY_CXT
```

- This defines global data to be copied
- Struct is treated like a PV by perl

Perl even helps you with XS

```
BOOT:
{
  MY_CXT_INIT;
}

void
print_global_value()
  PREINIT:
    dMY_CXT;
  CODE: {
    printf(”%p\n”,MY_CXT.x_Value)
  }
```

- This uses the global data

Magic

- To get a callback to clone your C object at clone time
- mg_dup magic virtual pointer
- Magic is a way to hang data of a normal scalar

Magic from threads.xs

```
MGVTBL ithread_vtbl = {
 ithread_mg_get,          /* get */
 0,                       /* set */
 0,                       /* len */
 0,                       /* clear */
 ithread_mg_free,         /* free */
 0,                       /* copy */
 ithread_mg_dup           /* dup */
}
```

- Magic ~ vtable

Magic from threads.xs

```
SV* obj = newSV(0);
SV* sv = newSVrv(obj,classname);
sv_setiv(sv,PTR2IV(thread));
mg = sv_magicext(sv, Nullsv,
PERL_MAGIC_shared_scalar, &ithread_vtbl,
(char *)thread, 0);
mg->mg_flags |= MGf_DUP;
SvREADONLY_on(sv);
```

- Create a new object with dup magic

Magic from threads.xs

```
ithread_mg_dup(pTHX_ MAGIC *mg,
CLONE_PARAMS *param)
{
    ithread *thread =
        (ithread *) mg->mg_ptr;
    MUTEX_LOCK(&thread->mutex);
    thread->count++;
    MUTEX_UNLOCK(&thread->mutex);
    return 0;
}
```

- Your own custom cloner

XS Mutex

```
static perl_mutex create_destruct_mutex;
MUTEX_INIT(&create_destruct_mutex);
MUTEX_LOCK(&create_destruct_mutex);
MUTEX_UNLOCK(&create_destruct_mutex);
MUTEXT_DESTROY(&create_destruct_mutex);
```

- Maps to the correct platform implementation
- mutex on pthreads, CriticalSection on win32

XS Cond

```
static perl_cond my_cond;
COND_INIT(&my_cond);
COND_LOCK(&my_cond);
COND_UNLOCK(&my_cond);
COND_DESTROY(&my_cond);
```

- Maps to the correct platform implementation
- cond on pthreads, WaitForMultipleObjects on win32

Perl 5.8.1

- Should be with us soon
- Fixes a lot of bugs in 5.8.0
  - srand()
  - $$
  - $0
  - memory leaks
  - hash access race condition

Future

- TPF Grant allows me to concentrate on threads again
- Create an information resource with FAQ/ Module lists
- More use when POE starts abusing it
- PerlInterpreter.pm
- COW

mod_perl2

The main reason for ithreads

Apache 2

- Multiple MPM
  - Multi process modules
    - Fork based
    - Threads based
    - A whole bag of weird mixes

Threaded MPM

- A pool of threads serving your request
  - Beneficial for platforms that have better thread context switching performance than process switching performance
- Win32!!
- Shared memory is good for caching!

How perl fits in this

- A pool of perl threads, ready to be given to an interpreter that needs to serve a mod_perl request
- No more need for front end proxying
- However, confusing, because a perl thread is not tied to a specific apache/OS thread

Actually work

- Crazy Taiwan Perl Hackers!
- Request Tracker 3.0 runs under threaded mod perl
- Biggest application I have heard off