Ruby trunk - Feature #3176

Thread#priority= should actually do something

04/20/2010 06:39 AM - coatl (caleb clausen)

Status:	Closed
Priority:	Normal
Assignee:	ko1 (Koichi Sasada)
Target version:	2.0.0

Description

=begin
Currently, Thread#priority= doesn't seem to do anything useful. See #1169 for more discussion of this. Here's a short program which demonstrates Thread#priority= not working, adapted from one in that issue:

```
$ cat thrprio.rb
c1 = c2 = 0
go=nil
t2 = Thread.new { 1 until go; loop { c2 += 1 } }
t2.priority = -2
t1 = Thread.new { 1 until go; loop { c1 += 1 } }
t1.priority = -1
go=true
sleep 5
t1.kill
t2.kill
puts "#{c1} #{c2} #{(c1-c2).to_f/(c1+c2)} #{c1 > c2}"

$ ruby thrprio.rb
17102855 17276166 -0.005041184855147562 false
$ ruby thrprio.rb
16839456 16977800 -0.0040909291989864585 false
$ ruby thrprio.rb
17063114 17248978 -0.005416858675781125 false
$ ruby thrprio.rb
16809137 17019727 -0.006225157309450296 false
```

When I run it, the 2 counts have very nearly the same value... to within c2. If thread priorities are working, c1 should be much larger than c2. (In ruby 1.8, in which Thread#priority= works, c1 is several orders of magnitude larger than c2.)

Depending on your OS and what order the threads are started in, you may see the reverse situation, where c1 > c2, but occasionally c1 < c2. This is still wrong, tho. The difference between c1 and c2 should be very large.

In the discussion below, GIL means the global interpreter lock (global_vm_lock).

Also, keep in mind that every mutex has inside it somewhere a queue which holds a list of the threads waiting for the mutex.

I have a theory that there's an interaction between the scheduler and the GIL which causes priorities to be effectively ignored. Imagine this:

There are 2 threads running, A and B. A.priority > B.priority.
Initially, A runs (so it holds GIL).
B tries to run, but has to wait on GIL.
After a short time, the scheduler forces a context switch.
So now A unlocks GIL, planning to immediately lock it again. This yields time to other threads.
But before A can lock GIL, B is scheduled (since it was at the head of the queue waiting for the GIL).
Now B owns GIL.
A tries to lock GIL, but can't obtain it, so it waits for it to be available.
After another timeslice, B yields time back to A in the same way.
...and so on

So, A and B are effectively alternating timeslices, and each gets roughly equal amounts of time. Even tho A should have a higher priority, and should get much more time. If the OS uses priority queues rather than normal fifo queues for the internal queue inside a mutex, then there would be no problem. However, I strongly suspect that most operating systems use fifo queues within their mutex implementations.

So far, this is a theory. I have no certain proof. I have poked around in the pthreads implementation inside glibc, and concluded that mutexes in glibc do seem to use fifo queues (not priority queues) internally.

I had assumed that thread_timer() in thread_pthread.c is what causes thread switches to happen... however, as I look at it more closely, I now suspect that that is not the case. I don't know where context switches are going on... but I still think the overall theory is correct.

If I'm right, then this might be fixable by making timeslices variable length, instead of always 10 ms. Lower priority threads would get shorter timeslices. Alternatively, high priority threads could get several timeslices in a row, and lower priority threads only one. There may well be even better ways to address the problem.

Yusuke, I've tried to make my explanation as clear as possible. Please let me know if it's still hard to understand.
=end

History

#1 - 04/21/2010 09:54 PM - coatl (caleb clausen)

=begin
Let me try again to explain this one, from a different point of view. I'm simplifying some things a little bit below, but all the essential parts remain true.

Every platform which implements threads has to have somewhere inside it a queue on which it stores the threads which are ready to run but not actually running yet. The type of this queue influences the behavior of the threading system... if it's a plain FIFO queue, then the threading system can't have thread priorities. In order for thread priorities to work at all, the ready-to-run queue has to be a priority queue.

Ok, so what does this mean for MRI? The OS-provided ready-to-run queue is present, but it isn't being used. It's always empty. All ruby threads which are ready to run are actually pending on the GIL. That's a mutex, and mutexes have their own queues internally on which they keep all the threads waiting for that mutex. So, the GIL's internal queue is MRI's actual ready-to-run queue. And (I contend) that queue is a FIFO, not a priority queue. Therefore, since MRI's actual ready-to-run queue is a FIFO, MRI does not respect thread priorities.

I have verified that for linux, the queues living inside mutexes are FIFOs. I'm fairly sure this is also the case on nearly every other OS as well.

From this point of view, the fix is fairly obvious. The GIL should be replaced with a priority queue. That is, convert calls to

```
$GIL.unlock
```

into
Thread.current=$ready_to_run.dequeue
Thread.current.wake_up!

and calls to

```
$GIL.lock
```

into
$ready_to_run.enqueue(thread: this_thread, priority: this_thread.priority)

I'm not quite sure what to do with BLOCKING_REGION; I think it requires some slightly special handling but shouldn't be very hard.
=end

#2 - 04/21/2010 10:59 PM - coatl (caleb clausen)

=begin
oh no, I left something off. I should have said:

convert calls to

```
$GIL.unlock
```

into
Thread.current=$ready_to_run.dequeue
Thread.current.wake_up!

and calls to

```
$GIL.lock
```

into
$ready_to_run.enqueue(thread: this_thread, priority: this_thread.priority)
this_thread.go_to_sleep #left off this line, sorry

=end

#3 - 04/21/2010 11:32 PM - mame (Yusuke Endoh)

=begin
Hi.

2010/4/21 caleb clausen redmine@ruby-lang.org:

> I'm not quite sure what to do with BLOCKING_REGION; I think it requires some slightly special handling but shouldn't be very hard.

A patch and benchmark result are welcome :-)

As far as I know, standard pthread does not provide a feature
that wakes up a specified thread (except pthread_kill).
Condition variable may be used to encode the feature, but I'm
afraid if it may degrade whole performance (even if the script
does not use Thread#priority).
I doubt whether Thread#priority is worth the risk and cost.

In addition, ko1 said, it is intented not to fix Thread#priorty.
He said that ruby should depend on thread scheduler of OS
because he is thinking of "parallel ruby" in the future.

The time has come that we should give up Thread#priority as
deprecated, I think.

--
Yusuke Endoh mame@tsg.ne.jp

=end

#4 - 05/12/2010 07:21 AM - coatl (caleb clausen)

=begin
Here's a patch which makes thread priorities work. This was made against the current trunk (1.9.3dev as of 11-may-2010).

Please consider this a preview to show where I want to go with this rather than a final product. Before this can be accepted, the following things should happen:

1) More testing. this passes make test and make test-all, but I haven't tried rubyspecs yet. (what else?)

2) Testing on more platforms. This appears to work on x86-linux, but multicore and 64bit machines should be tested, as well as other posix systems, and PARTICULARLY windows, which is rather more likely to have some kind of problem.

3) Most tests, for both thread priorities, and ffs implementation that I had to add to support platforms where ffs isn't present (windows).

4) Benchmarking. Yusuke was concerned about performance of this feature. I think speed will only be a tiny bit less with this patch, but that has yet to be proven.

5) Review for style, appropriateness, correctness, etc.

6) Patch should be split up into several chunks, for easier digestion.

Some notes on the implementation:

I've added a priority queue using the multi-level queue data structure, with bitmap scanned by ffs to quickly find the levels actually in use on dequeue. This data structure features O(1) insertion and deletion times, but has a moderately high fixed memory cost for even an empty queue versus tree-based priority queues. (33 words vs 1 word. BFD, I say.) And it only allows 32 different priority levels.
See http://en.wikipedia.org/wiki/Multilevel_queue

Ffs is a bit-scanning routine present on most (all?) unix systems. On most processors, there is an instruction to do this, making this operation nice and fast. Windows does not have ffs, so I wrote a (rather slower) version of it in c as a backup. (The x86 does have this instruction (it's called BSF, I think), but windows just doesn't make a nice programmer interface to it like unix does. If anyone wants to help me write the appropriate inline assembler to make this fast on windows too, I'd appreciate it.) I've made an ffs method available on Fixnum and Bignum at the ruby level as well.
See http://linux.die.net/man/3/ffs

Ruby traditionally has had fair thread priorities, meaning low priority threads still get a little time even when higher priority threads are trying to hog the processor. I've tried (I think successfully) to preserve this fairness by occasionally and temporarily boosting the priority of low-priority threads. But I'm not entirely certain that part of the algorithm works perfectly. The ratios of time given to low vs high priority threads in 1.8 are not preserved. A lowest priority thread may have to wait several seconds before it will get access to the cpu, if a highest priority thread is also running. (Actually, there's no upper limit on how long it will have to wait, given enough other threads contending for the cpu. Not sure it's possible to fix that.)

I'm going to take a break from this for a few days, then get back and finish it off after that.

=end

#5 - 05/12/2010 07:35 AM - coatl (caleb clausen)

- File thread-priorities-try2.diff added

=begin
Aaaag, I uploaded an empty patch. Sorry. Here's the real one.
=end

#6 - 05/14/2010 02:26 AM - mame (Yusuke Endoh)

- Assignee set to ko1 (Koichi Sasada)

- Target version set to 2.0.0

=begin
Hi, Caleb

Great. I glanced over your patch. I think the biggest change is how
thread waits GVL. It is quite funny because ko1 recently does the
same to fix another issue (thread starvation on many core environment).
(The fix have not been committed yet.)
I had concerned its performance cost, but if ko1 agrees with it, I
also agree.

However, ko1 seems to still dislike the priority support for some
reason. I don't know the precise reason. He said he would answer to
this ticket, so please wait for him.

Anyway, thank you for your writing a patch.

One comment for the patch: not-static functions (like pqueue_*) should
prefix "rb_" to avoid conflict with symbols of other projects, even if
they are just for internal.

--
Yusuke Endoh mame@tsg.ne.jp
=end

#7 - 05/16/2010 01:51 AM - kosaki (Motohiro KOSAKI)

=begin

> Issue #3176 has been updated by Yusuke Endoh.
>
> Assigned to set to Koichi Sasada
> Target version set to 1.9.x
>
> Hi, Caleb
>
> Great. I glanced over your patch. I think the biggest change is how
> thread waits GVL. It is quite funny because ko1 recently does the
> same to fix another issue (thread starvation on many core environment).
> (The fix have not been committed yet.)
> I had concerned its performance cost, but if ko1 agrees with it, I
> also agree.
>
> However, ko1 seems to still dislike the priority support for some
> reason. I don't know the precise reason. He said he would answer to
> this ticket, so please wait for him.
>
> Anyway, thank you for your writing a patch.
>
> One comment for the patch: not-static functions (like pqueue_*) should
> prefix "rb_" to avoid conflict with symbols of other projects, even if
> they are just for internal.

Alternative patch is here. I mean we should use os's thread priority
interface if possible.

The test result of caleb's testcase is,

```
./ruby projects/thrprio/thrprio.rb
315662770 209679 0.9986723818385312 true
```

Hmm...
Current MRI internal seems to have too many yield() and sleep(). Linux
setpriority()
give a thread to 1.25 times timeslice bonus per a nice. but this test but now
t1 got about 1500 times bonus against t2. ;-)

diff --git a/thread_pthread.c b/thread_pthread.c
index e6295db..dd66c40 100644
--- a/thread_pthread.c
+++ b/thread_pthread.c
@@ -535,7 +535,23 @@ native_thread_join(pthread_t th)
static void
native_thread_apply_priority(rb_thread_t *th)
{
-#if defined(_POSIX_PRIORITY_SCHEDULING) && (_POSIX_PRIORITY_SCHEDULING > 0)
+#ifdef linux

- int priority = 0 - th->priority; +
- if (priority > 19)
- priority = 19; +
- /*
- * No privileged can't use <0 priority. But we don't need care it.
- * Setpriority() naturally ignore such call.
- */
- if (priority < -20)
- priority = -20; +
- /* Strangely, Linux's setpriority(PRIO_PROCESS) change per-thread priority. */
- setpriority(PRIO_PROCESS, 0, priority);
- return; +#elif defined(_POSIX_PRIORITY_SCHEDULING) && (_POSIX_PRIORITY_SCHEDULING > 0) struct sched_param sp; int policy;
  int priority = 0 - th->priority;

=end

#8 - 05/16/2010 02:48 AM - kosaki (Motohiro KOSAKI)

=begin

The test result of caleb's testcase is,

./ruby projects/thrprio/thrprio.rb
315662770 209679 0.9986723818385312 true

Oops, please use following test instead caleb's one. because my
experimental patch haven't implement the way to chnage another thread
priority yet.

```
==================================================
c1 = c2 = 0
go=nil
t2 = Thread.new {
t2.priority = -2;
loop { c2 += 1 }
}
t1 = Thread.new {
t1.priority = -1;
loop { c1 += 1 }
}
go=true
sleep 5
t1.kill
t2.kill
puts "#{c1} #{c2} #{(c1-c2).to_f/(c1+c2)} #{c1.to_f/c2} #{c1 > c2}"
==================================================
```

Note: if we use /proc/sys/kernel/sched_compat_yield=1, we can get better result.

```
% taskset -c 1  ./ruby projects/thrprio/thrprio2.rb
16989126 12636726 0.14691223057483713 1.3444246555634742 true
```

=end

#9 - 05/16/2010 05:21 AM - kosaki (Motohiro KOSAKI)

=begin

Hi

> a slick little patch to just use setpriority
>
> And that works?!?

Do you mean my test result is not enough?

> Frankly, I should have tried this first, but I assumed it had been
> tried already and my groping around in NPTL source code seemed to
> indicate that it wouldn't. As long as this works, it is a much better
> solution than what I did.

The setpriority() is pure kernel feature. NPTL is unrelated.

Plus, Java JDK also is using setpriority(PRIO_PROCESS) for
supporting per-thread priority.

Strictly speaking, JavaVM don't have GVL, and it doesn't have
yield() flood issue either. but we have. This difference is important.
because GVL oftern makes confuse kernel thread scheduling.
Then, your patch works better in some case.

> How universal is this? You've got #ifdef linux in there, does that
> mean it won't work on any other posix system? Is this likely to work
> on windows? (I'm trying desperately to think of an excuse for keeping
> my patch around, can you tell?)

MacOS X, Solaris, Windows have each different changing way.
There isn't POSIX standard in this area unfortunatelly.

Plus we already windows one, see thread_win32.c#native_thread_apply_priority().

However, Some OS doesn't have per-thread priority feature at all. So,
I don't think your patch is useless. My intention was, I hoped to clarify
why you don't use setpriority() nor another os feature. your patch
didn't describe it.

=end

#10 - 05/16/2010 06:28 AM - kosaki (Motohiro KOSAKI)

=begin

> Note: if we use /proc/sys/kernel/sched_compat_yield=1, we can get better result.
>
> % taskset -c 1 ./ruby projects/thrprio/thrprio2.rb
> 16989126 12636726 0.14691223057483713 1.3444246555634742 true

And, if we apply following patch, tuning sched_compat_yield knob is
not necessary anymore.

```
diff --git a/thread_pthread.c b/thread_pthread.c
index e6295db..ddef0b3 100644
--- a/thread_pthread.c
+++ b/thread_pthread.c
@@ -130,7 +130,14 @@ native_cond_timedwait(pthread_cond_t *cond,
pthread_mutex_t *mutex, struct times

#define native_cleanup_push pthread_cleanup_push
#define native_cleanup_pop  pthread_cleanup_pop
-#ifdef HAVE_SCHED_YIELD
+
+#ifdef linux
+/*
```

- * Linux sched_yield() have difference behavior against other OS's one.
- * In many case, it does as NOP. We don't hope it.
- */ +#define native_thread_yield() (usleep(0)) +#elif defined(HAVE_SCHED_YIELD) #define native_thread_yield() (void)sched_yield() #else #define native_thread_yield() ((void)0)

=end

#11 - 05/18/2010 02:41 AM - coatl (caleb clausen)

- File thread-priorities-try3.diff added

=begin
Here's an updated version of my patch. I've incorporated changes according to Yusuke's and Tanaka'a comments and taken out the old way of implementing priorities (via the slice field of rb_thread_t; it was breaking my implementation in some cases).

I tried to make it use pthread_setschedparam on linux. That would seem to be the 'right' way to do what Kosaki was trying to do, since it allows you to set a thread priority from another thread. But it caused problems; the test program never finished, and wouldn't obey my C-c. So I had to back all that code out. I'll keep trying with that idea, since it would be preferable to have the OS handle priorities if at all possible.

This patch is still to be considered preliminary. I've tested it somewhat more, but I haven't made the tests automated yet. Benchmarks are still pending too.
=end

#12 - 05/18/2010 04:26 AM - kosaki (Motohiro KOSAKI)

=begin

> I tried to make it use pthread_setschedparam on linux. That would seem to be the 'right' way to do what Kosaki was trying to do, since it allows you to set a thread priority from another thread. But it caused problems; the test program never finished, and wouldn't obey my C-c. So I had to back all that code out. I'll keep trying with that idea, since it would be preferable to have the OS handle priorities if at all possible.

setschedparam()? no, it's for real time thread mess. please don't use it.

=end

#13 - 05/18/2010 07:47 AM - coatl (caleb clausen)

- File prio_test.rb added

=begin
Here's the test code I'm using right now. All the different parameters to this I've tried are working right now.

I also use it as a benchmark. Under both trunk and my patch, I see a total of about 151 million after running this test, tho with quite a bit of variation (as much as 5 million) around that. That indicates to me that in both cases the same amount of work is being done in the 10 seconds allotted.

Naturally, trunk fails the actual test.
=end

#14 - 05/18/2010 02:24 PM - kosaki (Motohiro KOSAKI)

=begin

> > setschedparam()? no, it's for real time thread mess. please don't use it.
>
> But setpriority always modifies the current thread. :-/ The API of #priority= is supposed to allow changing the priority from another thread.

Nope. setpriority() have pid argument, it can be passed thread-id.
see gettid() syscall. pthread_t is pthread level thread identifier. tid is kernel level thread identifier.

=end

#15 - 05/18/2010 03:40 PM - coatl (caleb clausen)

=begin
More test results:
With the latest patch, make test and make test-all both pass. (Well, make test-all dies with *** stack smashing detected ***, but I get that same error with trunk.) Rubyspec also passes, more or less (with the same set of errors as I see with trunk. (These are due to idiosyncrasies of my environment... missing openssl headers and the like.))

I'm still working on a coming up with more benchmarks.... any suggestions?

This patch needs testing on more operating systems, and on multi-core systems. Windows tests would be especially welcome.

=end

#16 - 05/19/2010 02:35 PM - coatl (caleb clausen)

- File thread-priorities-final.tgz added

=begin
I've divided my patch into 13 pieces for easier consumption. Since I can only upload 1 file at a time, I tarred them together. I hope this will be the final version of this patch, but I'm always willing to revise if there is more feedback. Koichi, I am most of all hoping to hear from you, since it has been said that you had some kind of objection to this patch.

I did try to create a more comprehensive benchmark, but I ended up crashing the interpreter, so I'm kind of stuck on that front at the moment. (See bug #3312.)

Kosaki, I have tried using setpriority on linux to get the OS to handle priorities, but it failed in the same way that pthread_setschedparam did; the test script would never finish and wouldn't even respond to ^C. I have no idea why this is failing so badly. (Maybe a bug in linux?? More likely a bug in my understanding.) In any case, another problem with this approach is that you won't readily be able to set a thread to higher priority than the main thread. Altho there may be ways around this, they come with their own issues. For instance, if a ruby process is started at the lowest possible priority (highest nice level) it won't be possible to have threads within that process be a different priority at all. On reflection, it seems better for ruby to manage its own thread priority queue (at least on linux) and have a guaranteed level of support for priorities and fixed number of available priorities. Your other patch (for native_thread_yield) looks pretty good, but I haven't tried it.

I've tested this on linux, but not anything else. I don't really have access to any other systems to test with. Please, can someone help me test this patch on other OS's?
=end

#17 - 05/19/2010 07:16 PM - kosaki (Motohiro KOSAKI)

=begin
Hi

> Kosaki, I have tried using setpriority on linux to get the OS to handle priorities, but it failed in the same way that pthread_setschedparam did; the test script would never finish and wouldn't even respond to ^C. I have no idea why this is failing so badly. (Maybe a bug in linux?? More likely a bug in my understanding.)

Which test case do you use? I'll investigate the issue.

> In any case, another problem with this approach is that you won't readily be able to set a thread to higher priority than the main thread. Altho there may be ways around this, they come with their own issues. For instance, if a ruby process is started at the lowest possible priority (highest nice level) it won't be possible to have threads within that process be a different priority at all. On reflection, it seems better for ruby to manage its own thread priority queue (at least on linux) and have a guaranteed level of support for priorities and fixed number of available priorities.

JVM have the same limitation. Then, I guess JRuby too.

http://java.sun.com/j2se/1.5.0/docs/guide/vm/thread-priorities.html#150

So, I think we can accept the limitation too.

=end

#18 - 05/22/2010 09:34 AM - coatl (caleb clausen)

- File setpriority_wont_work.diff added

=begin
I'm attaching a patch for using setpriority on linux and thereby bypassing (on that platform) all the priority queue stuff which my patch implements. setpriority still doesn't work for me, so this patch is to be considered highly experimental. (Just warning those who might not otherwise know.) This patch should be applied on top of my other patches.

I am using ubuntu 9.04. My uname -a shows this:
Linux baytree 2.6.28-18-generic #60-Ubuntu SMP Fri Mar 12 04:40:52 UTC 2010 i686 GNU/Linux

I may have missed the point again with this patch. I have been remarkably dense when it comes to this whole setpriority issue.

All my patches are relative to 27911 (??I think??), and therefore slightly stale. Lemme know if this is problem and I can update them again.
=end

#19 - 09/14/2010 04:01 PM - shyouhei (Shyouhei Urabe)

- Status changed from Open to Assigned

=begin

=end

#20 - 03/02/2012 11:36 PM - kosaki (Motohiro KOSAKI)

- Status changed from Assigned to Closed

1.9.3 has new thread scheduler and this problem is no longer reprodusable. closed then.

% ruby-193 thprio.rb
77219459 3027217 0.9245522144742793 true

% ruby-193 thprio.rb
78194222 2953676 0.9272026491678195 true

% ruby-193 thprio.rb
78585760 1466848 0.9633528991335298 true

Files

thread-priorities-try2.diff	21.8 KB	05/12/2010	coatl (caleb clausen)
thread-priorities-try3.diff	19.5 KB	05/18/2010	coatl (caleb clausen)
prio_test.rb	1.01 KB	05/18/2010	coatl (caleb clausen)
thread-priorities-final.tgz	8.78 KB	05/19/2010	coatl (caleb clausen)
setpriority_wont_work.diff	7.42 KB	05/22/2010	coatl (caleb clausen)