Please look at http://redmine.ruby-lang.org/issues/4535 before this one. That one actually fixes a bug I noticed while working on this patch.

Ruby 1.9 no longer depends on multiplexed non-blocking I/O to do its threading and defaults to blocking file descriptors.

As a result, there is no need to check the fd for read/writability when there is an error check for rb_io_wait_(read|writ|able) after the blocking function.

I also believe the code in io_binwrite() to:

avoid context switch between "a" and "n" in STDERR.puts "a".

...has always been broken under 1.9 with native threads.

Nothing new is broken with test-all and test-rubyspec

Associated revisions
Revision 84c77c15 - 09/09/2012 05:29 PM - kosaki (Motohiro KOSAKI)
  - ext/socket/basicsocket.c (rsock_bsock_send): avoid unnecessary select() calls before doing I/O Patch by Eric Wong. [Feature #4538] [ruby-core:35586]
  - ext/socket/init.c (rsock_s_recvfrom): ditto.
  - ext/socket/init.c (rsock_s_accept): ditto.
  - ext/socket/udpsocket.c (udp_send): ditto.
  - io.c (io_fflush): ditto.
  - io.c (io_binwrite): ditto.
  - io.c (rb_io_syswrite): ditto.

git-svn-id: svn+ssh://ci.ruby-lang.org/ruby/trunk@36944 b2dd03c8-39d4-4d8f-98ff-823fe69b080e

Revision 36944 - 09/09/2012 05:29 PM - kosaki (Motohiro KOSAKI)
  - ext/socket/basicsocket.c (rsock_bsock_send): avoid unnecessary select() calls before doing I/O Patch by Eric Wong. [Feature #4538] [ruby-core:35586]
  - ext/socket/init.c (rsock_s_recvfrom): ditto.
  - ext/socket/init.c (rsock_s_accept): ditto.
  - ext/socket/udpsocket.c (udp_send): ditto.
  - io.c (io_fflush): ditto.
  - io.c (io_binwrite): ditto.
  - io.c (rb_io_syswrite): ditto.

Revision 36944 - 09/09/2012 05:29 PM - kosaki (Motohiro KOSAKI)
  - ext/socket/basicsocket.c (rsock_bsock_send): avoid unnecessary select() calls before doing I/O Patch by Eric Wong. [Feature #4538] [ruby-core:35586]
  - ext/socket/init.c (rsock_s_recvfrom): ditto.
  - ext/socket/init.c (rsock_s_accept): ditto.
  - ext/socket/udpsocket.c (udp_send): ditto.
  - io.c (io_fflush): ditto.
History

#1 - 03/31/2011 03:37 PM - headius (Charles Nutter)

=begin
Is it possible to interrupt/wakeup a thread that's doing a direct blocking IO call? I always understood that as the primary reason for doing the select logic.
=end

#2 - 04/01/2011 07:29 AM - normalperson (Eric Wong)

=begin
Charles Nutter headius@headius.com wrote:

Is it possible to interrupt/wakeup a thread that's doing a direct blocking IO call? I always understood that as the primary reason for doing the select logic.

Yes for non-regular files as long as the signal handlers don't set the SA_RESTART flag. Ruby does not set SA_RESTART anywhere and can interrupt I/O on pipes/sockets at any time.

Regular files are special. select() just returns success immediately on regular files and the IO operation will block (refusing to accept
signals) while waiting for disk. NFS can be mounted to be interruptable, but you still can't rely on select()/poll() for readiness notification.

--
Eric Wong
=end

#3 - 04/01/2011 07:01 PM - headius (Charles Nutter)
=begin
Understood. I read through a bit more code and saw that Ruby uses pthread signalling and RUBY_VM_CHECK_INTS after blocking regions to interrupt blocking operations.

I wonder, though, if depending on this behavior is leading Ruby more and more down the GVL path. The designers of the JVM's core IO libraries, for example, were unable to reconcile concurrent native threads with interruptible IO, due to the impossibility of knowing what state all IO-related data structures are in when the thread is interrupted. As a result, IO channels performing blocking operations are explicitly closed when the thread they block is interrupted.

In JRuby, we simulate interruptible IO by using select as much as possible, and blocking operations against unselectable IO channels are not interruptible. Some of the overhead of select is mitigated by JVM implementers usually using the fastest-possible mechanism to implement it (kqueue, epoll).

It seems that your change (and others like it) makes Ruby even more dependent on kernel-level blocking IO operations always being safely interruptible, and depending on those interruptions to only occur at the exact boundaries defined by the GVL. A future concurrent-threaded Ruby (or other impls that may become concurrent-threaded) may want to consider this, no? And are there any cross-platform concerns from eliminating select in these cases?

I also wonder if there's a race condition here; is it not possible that the interrupt of a thread would fire immediately after the GVL has been released but before the blocking IO operation has fired? Perhaps I'm birdwalking too deep into the vagaries of MRI's IO logic.
=end

#4 - 04/02/2011 07:23 AM - normalperson (Eric Wong)
=begin
Charles Nutter headius@headius.com wrote:

I don't think so, even if threads are interrupted they're resumed after the signal handler is done (or the process is dying anyways and we don't care). If the interrupt is to raise an exception then that could get messy[1], but for the general case of signal handlers it's not an issue.

As a result, IO channels performing blocking operations are explicitly closed when the thread they block is interrupted.

That is terrible. I'd never touch a platform that does that.

It seems that your change (and others like it) makes Ruby even more dependent on kernel-level blocking IO operations always being safely interruptible, and depending on those interruptions to only occur at the exact boundaries defined by the GVL. A future concurrent-threaded Ruby (or other impls that may become concurrent-threaded) may want to consider this, no? And are there any cross-platform concerns from eliminating select in these cases?

If there are cross-platform concerns, the functions that wrap select() should be made no-op on platforms where select() is not needed (on all POSIX-like ones, I expect) and not interfere with platforms where they're not needed.

Regardless, there'll always be a set of IO operations that can never be interrupted. That doesn't bother me at all since the rest of the VM still runs. I'd rather just not use select()/poll() at all for "blocking" I/O calls.
I also wonder if there's a race condition here; is it not possible that the interrupt of a thread would fire immediately after the GVL has been released but before the blocking IO operation has fired? Perhaps I'm birdwalking too deep into the vagaries of MRI's IO logic.

So a signal handler might fire and the syscall would just continue and not fail with EINTR. No big deal, it'll just finish the syscall before checking for interrupts.

The real race condition is relying on select()/poll() at all for readability. select()/poll() returning success never guarantees an operation won't block due to spurious wakeups and shared IO across multiple threads/processes.

[1] - which is why rb_ensure() is used in some places, such as using with(select() for rb_fd_init()/rb_fd_term())

--
Eric Wong
=end

#5 - 04/03/2011 07:23 AM - headius (Charles Nutter)
=begin
On Fri, Apr 1, 2011 at 4:57 PM, Eric Wong normalperson@yhbt.net wrote:

Charles Nutter headius@headius.com wrote:

I wonder, though, if depending on this behavior is leading Ruby more and more down the GVL path. The designers of the JVM's core IO libraries, for example, were unable to reconcile concurrent native threads with interruptible IO, due to the impossibility of knowing what state all IO-related data structures are in when the thread is interrupted.

I don't think so, even if threads are interrupted they're resumed after the signal handler is done (or the process is dying anyways and we don't care). If the interrupt is to raise an exception then that could get messy[1], but for the general case of signal handlers it's not an issue.

I'm speaking specifically of Thread#raise and Thread#kill, which if used to interrupt a thread could potentially leave the IO channel in an unknown state (due to interrupting during a system call). On the JVM, all process-level signals are handled by a separate thread, so they are never run on user threads and that's not a concern for us. JRuby has real concurrent threads, so regardless of what blocking calls we make other threads will continue to run (i.e. we have no need for BLOCKING_REGION-tye GVL logic). So ultimately it's only being able to kill or raise in an arbitrary thread that led us to make JRuby IO logic use selection to get around the effects of interrupting blocking IO calls I mentioned below.

Long story short, how does MRI guarantee that the underlying IO is in a reliable state when the thread accessing it can be interrupted permanently? It seems like doing most blocking at a consistent point (like a select call) is safer.

And I am mostly just trying to understand how it's consistently safe to interrupt a system-level IO call.

As a result, IO channels performing blocking operations are explicitly closed when the thread they block is interrupted.

That is terrible. I'd never touch a platform that does that.

Well, I tend not to touch platforms that expose or depend on specific platform details, like MRI does in many places (and now more places with your patch, I think). I like my code to work the same on all platforms.
That said, I admit it's inconvenient, but I understand the reasoning. You have to understand the JVM is trying to smooth over the platform-specific details of IO across lots of platforms, many of them not POSIX. If you can't guarantee to user code what the state of an IO channel will be when interrupting system-level code, it's a pretty clean option to say "don't do that, or we'll close the stream" and point users toward a safely interruptible option like select.

We've managed to work with that situation and mostly emulate MRI's IO behavior, so in practice it's more a nuisance than anything else.

If there are cross-platform concerns, the functions that wrap select() should be made no-op on platforms where select() is not needed (on all POSIX-like ones, I expect) and not interfere with platforms where they're not needed.

Regardless, there'll always be a set of IO operations that can never be interrupted. That doesn't bother me at all since the rest of the VM still runs. I'd rather just not use select()/poll() at all for "blocking" I/O calls.

That seems good on the surface, but it's depending on those blocking operations having consistent state after being interrupted across platforms. That seems like it would be easier to guarantee at a "select" level, but I admit I'm trying to understand if that's true. If you can't guarantee that the underlying IO channels are in a consistent state (ideally the same state regardless of platform) then writing to IO becomes a bunch of platform-specific checks in user code just like you'd have to write in C. The structure of Ruby's APIs has always been to provide a reasonably consistent view of system-level APIs so you don't have to do that.

I also wonder if there's a race condition here; is it not possible that the interrupt of a thread would fire immediately after the GVL has been released but before the blocking IO operation has fired? Perhaps I'm birdwalking too deep into the vagaries of MRI's IO logic.

So a signal handler might fire and the syscall would just continue and not fail with EINTR. No big deal, it'll just finish the syscall before checking for interrupts.

Except that you've now fired your Thread#kill or Thread#raise and the thread is never going to see it. If the contract of kill and raise is that "we'll try to kill or raise in the target thread, but no guarantees if it will do anything at all" I'm fine with that, but that hasn't been the expectation of Ruby users up to now. I'm not sure if this is actually a problem or not...MRI's cross-thread event behavior is rather involved.

The real race condition is relying on select()/poll() at all for readability. select()/poll() returning success never guarantees an operation won't block due to spurious wakeups and shared IO across multiple threads/processes.

That's certainly true, but any code using select would not just blindly proceed to a blocking call after wakeup...it would check that the IO channel is actually ready, and if not go into select again. I don't see how that makes the consistency and reliability of blocking on selection less attractive than interrupting arbitrary kernel-level calls.

- Charlie =end

#6 - 04/05/2011 03:23 AM - normalperson (Eric Wong)

=begin
Charles Oliver Nutter headius@headius.com wrote:

On Fri, Apr 1, 2011 at 4:57 PM, Eric Wong normalperson@yhbt.net wrote:

Charles Nutter headius@headius.com wrote:
I wonder, though, if depending on this behavior is leading Ruby more and more down the GVL path. The designers of the JVM’s core IO libraries, for example, were unable to reconcile concurrent native threads with interruptible IO, due to the impossibility of knowing what state all IO-related data structures are in when the thread is interrupted.

I don’t think so, even if threads are interrupted they’re resumed after the signal handler is done (or the process is dying anyways and we don’t care). If the interrupt is to raise an exception then that could get messy[1], but for the general case of signal handlers it’s not an issue.

I’m speaking specifically of Thread#raise and Thread#kill, which if used to interrupt a thread could potentially leave the IO channel in an unknown state (due to interrupting during a system call).

Long story short, how does MRI guarantee that the underlying IO is in a reliable state when the thread accessing it can be interrupted permanently? It seems like doing most blocking at a consistent point (like a select call) is safer.

MRI should (already appears to) define rb_thread_blocking_region() as a cancellation point for Thread#raise and Thread#kill so any C code should tidy things up before entering/leaving a blocking region.

And I am mostly just trying to understand how it’s consistently safe to interrupt a system-level IO call.

The actual syscall are usually very trivial and has few (if any) user-visible internal structures to worry about unless memory was malloc()-ed for it (in the case of select() + rb_fd_init()).

The kernel is expected to handle all the internal structures for interruptibility (it only exposes an opaque integer descriptor to userspace).

As a result, IO channels performing blocking operations are explicitly closed when the thread they block is interrupted.

That is terrible. I’d never touch a platform that does that.

Well, I tend not to touch platforms that expose or depend on specific platform details, like MRI does in many places (and now more places with your patch, I think). I like my code to work the same on all platforms.

We’ll have to agree to differ here :

I choose to work on my platform (Linux) because I see more benefits to it than alternatives and would like to take advantage of strengths of it.

If there are cross-platform concerns, the functions that wrap select() should be made no-op on platforms where select() is not needed (on all POSIX-like ones, I expect) and not interfere with platforms where they’re not needed.

Regardless, there’ll always be a set of IO operations that can never be interrupted. That doesn’t bother me at all since the rest of the VM still runs. I’d rather just not use select()/poll() at all for “blocking” IO calls.

That seems good on the surface, but it’s depending on those blocking operations having consistent state after being interrupted across platforms. That seems like it would be easier to guarantee at a “select” level, but I admit I’m trying to understand if that’s true. If you can’t guarantee that the underlying IO channels are in a
consistent state (ideally the same state regardless of platform)
then writing to IO becomes a bunch of platform-specific checks in user
code just like you'd have to write in C. The structure of Ruby's APIs
has always been to provide a reasonably consistent view of
system-level APIs so you don't have to do that.

Upon further inspection, I see MRI uses select() only in 100ms
increments while checking for interrupts on win32. That may be because
win32 can't interrupt syscalls like select(), but all other platforms
MRI supports can...

I shall update my patch to only select() before I/O on win32 is somebody
can confirm it is needed. On POSIX, select() never wakes up unless it
receives a signal (or a descriptor is ready).

I also wonder if there's a race condition here: is it not possible
that the interrupt of a thread would fire immediately after the GVL
has been released but before the blocking IO operation has fired?
Perhaps I'm birdwalking too deep into the vagaries of MRI's IO logic.

So a signal handler might fire and the syscall would just continue and
not fail with EINTR. No big deal, it'll just finish the syscall before
checking for interrupts.

Except that you've now fired your Thread#kill or Thread#raise and the
thread is never going to see it.

I don't think "never" is correct, but seeing it too late seems to be
a current problem...

If the contract of kill and raise is
that "we'll try to kill or raise in the target thread, but no
guarantees if it will do anything at all" I'm fine with that, but that
hasn't been the expectation of Ruby users up to now. I'm not sure if
this is actually a problem or not...MRI's cross-thread event behavior
is rather involved.

Refiring signal pthread_kill() is probably needed if the syscall blocks
a long time:

blocking_thread     interrupting_thread

check ints => nothing
release gvl
Thread#raise
set interrupt flag
pthread_kill()
long syscall
(long time passes...)
check ints => finally sees Thread#raise

In this case for delivering Thread#raise in timely fashion, the
interrupting thread will need to set a timer to refire pthread_kill() in
a loop if it detects blocking_thread hasn't reacted to the signal, yet.
Multiple EINTRs from pthread_kill() wouldn't be any more/less harmful
than one EINTR.

I realize doing a short 100ms poll()/select() like win32 does is
possible for timelier delivery of Thread#raise/Thread#kill, but I'd
rather avoid those expensive syscalls for general case since
Thread#raise/Thread#kill is not common.

The real race condition is relying on select()/poll() at all for
readability. select()/poll() returning success never guarantees an
operation won't block due to spurious wakeups and shared IO across
multiple threads/processes.

That's certainly true, but any code using select would not just
blindly proceed to a blocking call after wakeup...it would check that
the IO channel is actually ready, and if not go into select again. i
do n't see how that makes the consistency and reliability of blocking
on selection less attractive than interrupting arbitrary kernel-level
calls.

Blocking syscalls can be better for some cases (e.g. accept() under
Linux) since the kernel can implement behavior to wake up exactly one
waiter on a ready client, whereas with select()/poll(), all the waiters
get woken. I try to take advantage of that (avoids doubling up on
syscalls made) when possible.

--
Eric Wong
=end

#7 - 06/10/2011 10:53 PM - ko1 (Koichi Sasada)

Hi,

Any action on this proposal?

This thread is too long and difficult to understand....

(2011/03/30 3:22), Eric Wong wrote:

Issue #4538 has been reported by Eric Wong.

Feature #4538: [PATCH (cleanup)] avoid unnecessary select() calls before doing I/O
http://redmine.ruby-lang.org/issues/4538

Author: Eric Wong
Status: Open
Priority: Low
Assignee: 
Category: core
Target version: 1.9.x

Please look at http://redmine.ruby-lang.org/issues/4535 before
this one. That one actually fixes a bug I noticed while working
on this patch.

Ruby 1.9 no longer depends on multiplexed non-blocking I/O
to do its threading and defaults to blocking file descriptors.

As a result, there is no need to check the fd for read/writability when
there is an error check for rb_io_wait_(read|writ)able after the
blocking function.

I also believe the code in io_binwrite() to:
avoid context switch between "a" and "\n" in STDERR.puts "a".
[ruby-dev:25080]
...has always been broken under 1.9 with native threads.

Nothing new is broken with test-all and test-rubyspec

--
// SASADA Koichi at atdot dot net

#8 - 06/10/2011 11:05 PM - kosaki (Motohiro KOSAKI)

- Status changed from Open to Assigned
- Assignee set to kosaki (Motohiro KOSAKI)

Assigned.

#9 - 09/10/2012 02:29 AM - kosaki (Motohiro KOSAKI)

- Status changed from Assigned to Closed
- % Done changed from 0 to 100
This issue was solved with changeset r36944.
Eric, thank you for reporting this issue.
Your contribution to Ruby is greatly appreciated.
May Ruby be with you.

- ext/socket/basicsocket.c (rsock_bsock_send): avoid unnecessary select() calls before doing I/O Patch by Eric Wong. [Feature #4538]
  [ruby-core:35586]
- ext/socket/init.c (rsock_s_recvfrom): ditto.
- ext/socket/init.c (rsock_s_accept): ditto.
- ext/socket/udpsocket.c (udp_send): ditto.
- io.c (io_flush): ditto.
- io.c (io_binwrite): ditto.
- io.c (rb_io_syswrite): ditto.

**Files**

<table>
<thead>
<tr>
<th>File Name</th>
<th>Size</th>
<th>Date</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001-avoid-unnecessary-select-calls-before-doing-I-O.patch</td>
<td>5.31 KB</td>
<td>03/30/2011</td>
<td>normalperson (Eric Wong)</td>
</tr>
</tbody>
</table>