Ruby master - Bug #8185

Thread/fork issue

03/30/2013 03:33 AM - Anonymous

Status:	Open		
Priority:	Normal		
Assignee:			
Target version:			
ruby -v:	2.0.0p100	**Backport:**	

Description

Hello all,

I've found an issue where calling fork inside a thread, and passing a block
to the fork, causes the forked process to continue after the block. I've
reproduced the issue on the following versions of ruby:
ruby 2.0.0p100 (2013-03-27 revision 39954) [x86_64-darwin10.8.0]
ruby 1.9.3p392 (2013-02-22 revision 39386) [x86_64-darwin10.8.0]

Here is the script I used to reproduce:

```
1000.times do |j|
puts "run #{j}"
threads = []
100.times do |i|
threads << Thread.new(i) do |local_i|
opid = fork do
# exit!(true) # fixes the issue
# exit(true) # doesn't fix the issue
# no 'exit' also exhibits issue
end
::Process.waitpid(opid, 0)
File.open("/tmp/test_thread_fork_#{local_i}.pid", "w") {|f| f.write
"1" }
end
end
threads.map { |t| t.join }

borked = false
100.times do |i|
fn = "/tmp/test_thread_fork_#{i}.pid"
contents = File.read(fn)
if contents.size > 1
puts "file #{fn} was written to many times (#{contents})"
borked = true
end
end
exit(false) if borked
end
```

As you can see from the comments inside the fork I can work around the
issue by using "exit!". I am correct in understanding that there should be
no case in which the file is written to multiple times, correct?

Thank you,
Jason Gladish

History

#1 - 03/30/2013 03:34 AM - drbrain (Eric Hodel)

- Category set to core

- ruby -v set to 2.0.0p100

#2 - 03/30/2013 03:53 AM - drbrain (Eric Hodel)

On Mar 29, 2013, at 10:16, Jason Gladish jason@expectedbehavior.com wrote:

> I've found an issue where calling fork inside a thread, and passing a block to the fork, causes the forked process to continue after the block

I've created http://bugs.ruby-lang.org/issues/8185 from this email.

PS: you can create issues directly at http://bugs.ruby-lang.org/projects/ruby-trunk/issues/new

#3 - 03/30/2013 07:53 AM - akr (Akira Tanaka)

2013/3/30 Jason Gladish jason@expectedbehavior.com:

> I've found an issue where calling fork inside a thread, and passing a block
> to the fork, causes the forked process to continue after the block. I've
> reproduced the issue on the following versions of ruby:

It seems buffered IO data is flushed in several child processes.
The contorol itself dosen't continue after the block.

exit!(true) solves the problem because it doesn't flush IO.
Also, if you use f.syswrite instead of f.write, the problem disappears.

The problem is reproduced more reliably by sleeping a second after f.write.

```
1000.times do |j|
puts "run #{j}"
threads = []
100.times do |i|
threads << Thread.new(i) do |local_i|
opid = fork do
# exit!(true) # fixes the issue
# exit(true) # doesn't fix the issue
# no 'exit' also exhibits issue
end
::Process.waitpid(opid, 0)
File.open("/tmp/test_thread_fork_#{local_i}.pid", "w") {|f|
f.write "1"
sleep 1
f.flush
}
end
end
threads.map { |t| t.join }
```

```
borked = false
100.times do |i|
  fn = "/tmp/test_thread_fork_#{i}.pid"
  contents = File.read(fn)
  if contents.length > 1
    puts "file #{fn} was written to many times (#{contents})"
    borked = true
  end
end
exit(false) if borked
```

end

The problem can be solved if Ruby flushes all IO in the fork method.
(Currently only STDOUT and STDERR is flushed.)

I'm not sure we should do it because it slows down fork method
by scanning all objects.
--
Tanaka Akira

#4 - 04/02/2013 10:23 AM - akr (Akira Tanaka)

2013/3/30 Tanaka Akira akr@fsij.org:

> 2013/3/30 Jason Gladish jason@expectedbehavior.com:
>
> > I've found an issue where calling fork inside a thread, and passing a block

> to the fork, causes the forked process to continue after the block. I've
> reproduced the issue on the following versions of ruby:

> It seems buffered IO data is flushed in several child processes.
> The contorol itself dosen't continue after the block.
>
> exit!(true) solves the problem because it doesn't flush IO.
> Also, if you use f.syswrite instead of f.write, the problem disappears.
>
> The problem is reproduced more reliably by sleeping a second after f.write.

I wrote a simple script to reproduce the problem.
This script doesn't use multi-threads.

```
% ./ruby -ve '
open("zz", "w") {|f|
f.print "foo\n"
Process.wait fork {}
p $$
}
'
ruby 2.1.0dev (2013-04-01 trunk 40040) [x86_64-linux]
4784
% cat zz
foo
foo
```

The buffered data in f, "foo\n", is flushed in the child process and
the parent process.
So zz has two "foo\n".
--
Tanaka Akira

#5 - 04/03/2013 04:53 AM - kosaki (Motohiro KOSAKI)

> I wrote a simple script to reproduce the problem.
> This script doesn't use multi-threads.
>
> ```
> % ./ruby -ve '
> open("zz", "w") {|f|
> f.print "foo\n"
> Process.wait fork {}
> p $$
> }
> '
> ruby 2.1.0dev (2013-04-01 trunk 40040) [x86_64-linux]
> 4784
> % cat zz
> foo
> foo
> ```
>
> The buffered data in f, "foo\n", is flushed in the child process and
> the parent process.
> So zz has two "foo\n".

I believe fork() should flush all IO objects automatically. 1) Why user
should know how ruby beffering? 2) fork() is not common method and
performance disadvantage is not much though.

#6 - 04/03/2013 07:23 AM - akr (Akira Tanaka)

2013/4/3 KOSAKI Motohiro kosaki.motohiro@gmail.com:

> I believe fork() should flush all IO objects automatically. 1) Why user
> should know how ruby beffering? 2) fork() is not common method and
> performance disadvantage is not much though.

I see. Agreed.
--

Tanaka Akira

#7 - 04/07/2013 10:23 AM - akr (Akira Tanaka)

2013/4/3 Tanaka Akira akr@fsij.org:

> 2013/4/3 KOSAKI Motohiro kosaki.motohiro@gmail.com:
>
> > I believe fork() should flush all IO objects automatically. 1) Why user should know how ruby beffering? 2) fork() is not common method and performance disadvantage is not much though.
>
> I see. Agreed.

On second thought, I feel exit! is more essential solution.

Because there may be other objects which behaves as "flush at finalization",
flushing IO objects is not enough.

The problem is finalization is occur twice (at parent and child).
If ruby uses exit! for forked process, we can avoid dual finalization.

However it means users of fork must finalize objects explicitly in
forked process.
For example, fork { print "a" } may lost "a" because buffered "a" is
not flushed.
--
Tanaka Akira

#8 - 04/07/2013 11:29 PM - kosaki (Motohiro KOSAKI)

> On second thought, I feel exit! is more essential solution.
>
> Because there may be other objects which behaves as "flush at finalization",
> flushing IO objects is not enough.
>
> The problem is finalization is occur twice (at parent and child).
> If ruby uses exit! for forked process, we can avoid dual finalization.
>
> However it means users of fork must finalize objects explicitly in
> forked process.
> For example, fork { print "a" } may lost "a" because buffered "a" is
> not flushed.

Also, Process.exit! escape to run at_exit handler. I think exit! is too drastic.
it help many situation than flushing, but also makes much side effect. I suspect
"fork { print "a" }" type output lost makes lots test-all break.

Instead of, providing fork callback likes pthread_atfork() and uses it?

#9 - 04/07/2013 11:53 PM - akr (Akira Tanaka)

2013/4/7 KOSAKI Motohiro kosaki.motohiro@gmail.com:

> Also, Process.exit! escape to run at_exit handler. I think exit! is too drastic.
> it help many situation than flushing, but also makes much side effect. I suspect
> "fork { print "a" }" type output lost makes lots test-all break.

We can flush stdout (and stderr).
Because they are flushed just before fork, flush on child and parent
is harmless.
This doesn't solve problems about other objects, though.

Of course, there is a less-drastic document-only solution:
"Use exit! in forked child process to avoid dual finalization."

> Instead of, providing fork callback likes pthread_atfork() and uses it?

I'm not sure how such calllback is used.
--
Tanaka Akira