Extracted from:

The dRuby Book

Distributed and Parallel Computing with Ruby

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

The dRuby Book

Distributed and Parallel Computing with Ruby

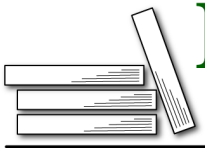
Masatoshi Seki

Translated by Makoto Inoue

Foreword by Yukihiro “Matz” Matsumoto

Edited by Susannah Davidson Pfalzer

The Facets of Ruby Series



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Susannah Pfalzer (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Original Japanese edition:
"dRuby niyoru Bunsan Web Programming" by Masatoshi Seki
Copyright © 2005. Published by Ohmsha, Ltd

Printed in the United States of America.
ISBN-13: 978-1-934356-93-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2012

In many systems, each thread shares the same memory address space. Systems can switch the flow of control, but they can't switch memory space. So, threads tend to have less overhead and are often called *lightweight* processes.

With multithreading, you can easily write applications that handle multiple events, such as a network or GUI application. However, multithreading causes problems that don't happen in single-threaded mode.

dRuby-based systems are often composed of multiple processes. When serving as a server, dRuby has no idea when clients will call the server methods. This situation is similar to what happens when programming in a multithreaded environment. In this chapter, we'll learn about multithreading in Ruby, communication between threads, and how to apply these techniques when programming in dRuby.

Before you start this chapter, keep in mind that multithreading is a difficult topic in general; don't be surprised if you feel overwhelmed by the number of topics covered in this chapter. Feel free to skip whenever you like. However, do read Section 5.4, *Passing Objects via Queue*, on page ? to understand how to use `Queue` to communicate in a multithreaded environment, because we'll compare `Queue` with other concepts in Chapter 6, *Coordinating Processes Using Rinda*, on page ? and Chapter 9, *Drip: A Stream-Based Storage System*, on page ?.

5.1 dRuby and Multithreading

Multithreading is vital in dRuby. In this section, we'll take a look at the relationship between dRuby and multithreading.

Always in Multithreading Mode

dRuby generates threads by first having `DRb.start_service` generate a server thread that waits for method calls from other processes. Then, every time the server thread receives a method call from the client, the server generates a new thread that takes care of executing the method call.

While the server is handling other remote method calls, it creates a new thread and executes it when there's a call from a client.

Let's see how process A calls process B (see Figure 19, *How remote method calls work*, on page 6).

```
there = DRbObject.new_with_uri('...')
there.foo()
```

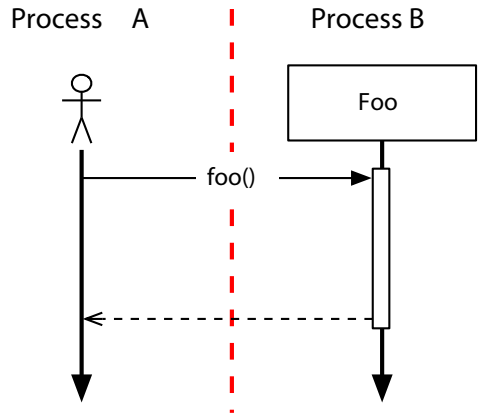


Figure 19—How remote method calls work

When there is a request to process B, then a server thread at DRbServer receives the request. DRbServer then creates a new method invocation and delegates the task to it in a separate thread. As soon as it gets delegated, the server thread comes back and prepares for the next method call (see Figure 20, *Implementing a remote method call*, on page 7).

Thanks to this mechanism, dRuby can receive different calls and execute them while it's in the middle of processing other calls. Let's think about the following code:

```
ary = DRbObject.new_with_uri(..)
ary.each do |x|
  x.foo()
end
```

Because dRuby has no constraints—such as blocking other method calls while performing a method call—it doesn't cause a deadlock when two processes call each other (see Figure 21, *Two processes calling each other*, on page 8). This architecture enables you to call remote methods with block arguments.

The main goal of dRuby is to extend Ruby's method invocation to a distributed environment. I implemented dRuby so that objects can call each other just as they do in normal Ruby scripts. For this reason, you have to pay particular attention to multithreading when using dRuby. Always bear in mind that the server script may receive method calls at any time.

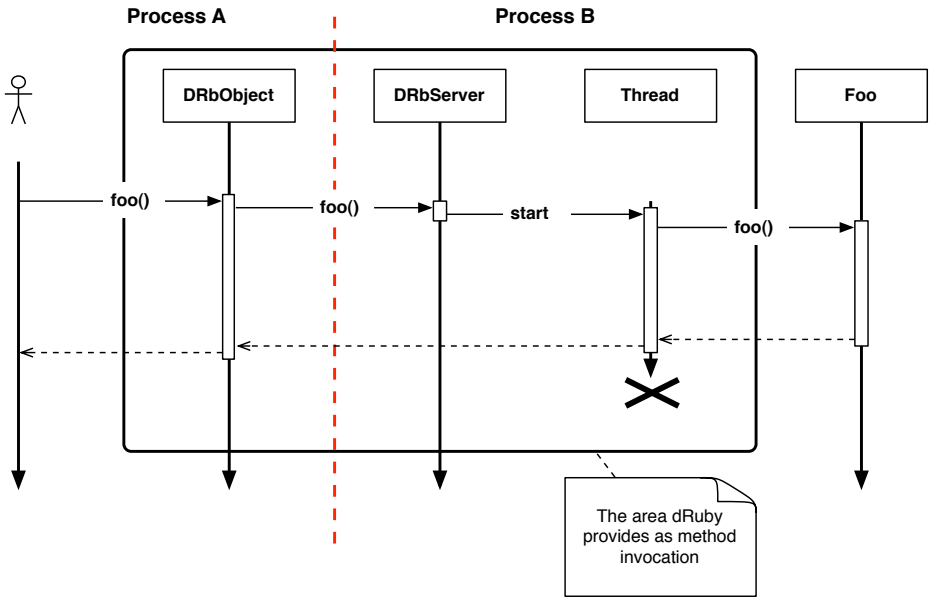


Figure 20—Implementing a remote method call

5.2 Understanding the Thread Class

When a Ruby interpreter executes a script, a main thread starts (see Figure 22, *Main thread at start-up*, on page 8). The main thread is in charge of processing the main script. The main thread is invoked in the very beginning, and when the main thread ends, then the script also ends. The main thread is the longest-living thread within a script's life cycle.

Ruby threading works by passing a block to the `Thread` class. You can pass parameters to the block by passing arguments in `Thread.new`.

```
thread = Thread.new(1,2,3) do |x, y, z|
  # Operations to be threaded.
end
```

This creates a "program flow" (see Figure 23, *Launching the second thread*, on page 9).

A thread has various states, such as `running` or `sleep`. When the thread is waiting for `IO` or other threads, then it goes into sleep mode. A thread often goes back and forth between `run` and `sleep` mode and eventually finishes (see Figure 24, *States of a thread*, on page 9).

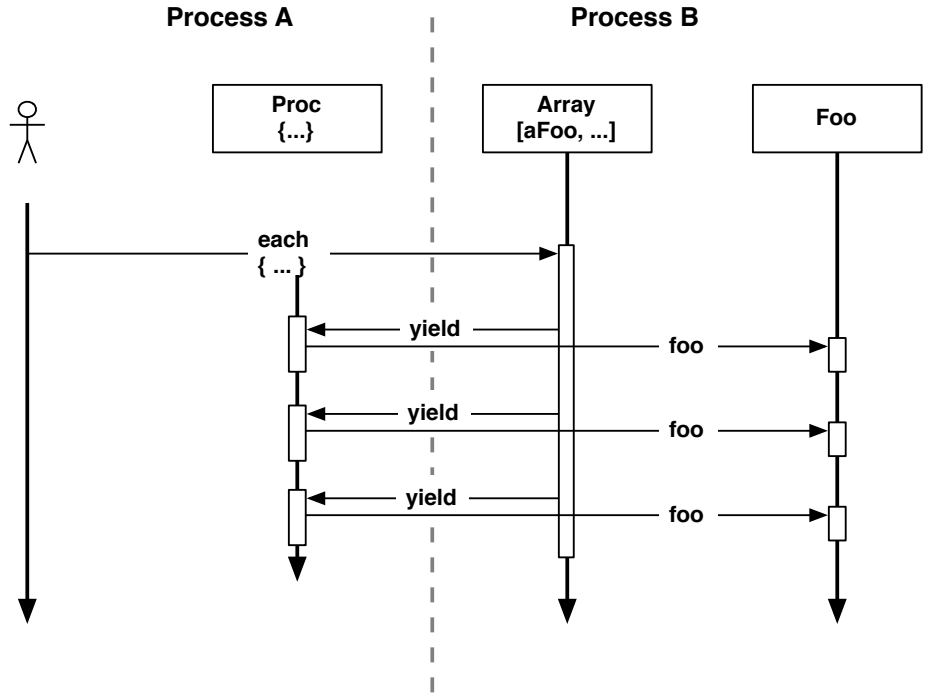


Figure 21—Two processes calling each other

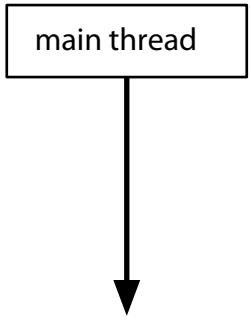


Figure 22—Main thread at start-up

When a thread ends, it contains the end state. If the thread was terminated by an exception, then it will raise an exception when you try to access the value.

You can check the state of a thread using the alive?, status, and value methods.

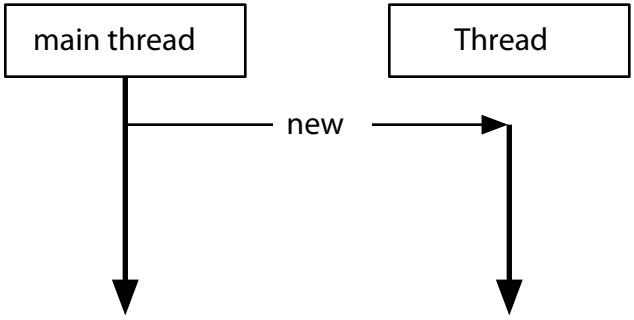


Figure 23—Launching the second thread

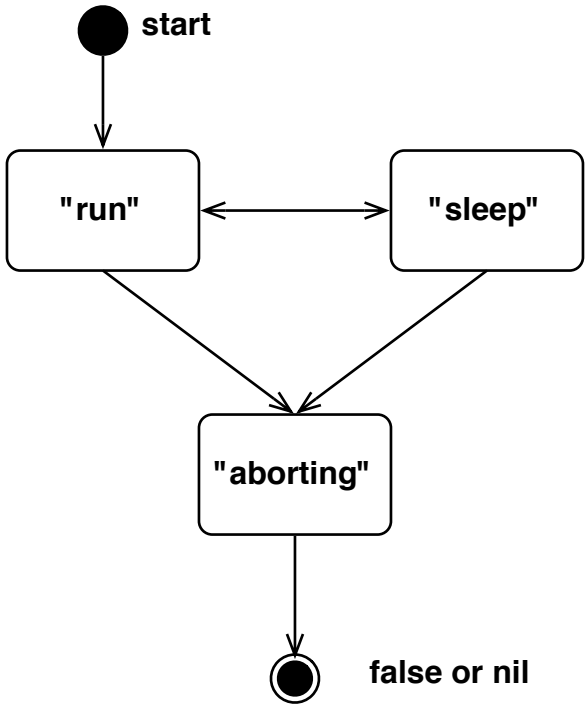


Figure 24—States of a thread

alive?
: Returns true or false

status
: Returns the following state code:

- "run" running.
- sleep sleeping.
- aborting aborting.
- false finished normally.
- nil terminated by exception.

stop?
: Returns true when the thread ended or is asleep.

value
: Waits until the thread ends and returns the value. If the thread is already finished, then it returns the value immediately. If it was terminated by an exception, then it raises an exception. You can access value at any time, and an exception will be raised every time you access the value.

value waits and returns the value. If all you want is to wait until the end of the thread, then you can use the join method. When the join method is called, it blocks the calling thread until the receiver thread ends (see Figure 25, *Threads waiting to join*, on page 11).

To control the state of the thread, you can use the following methods:

exit
: Terminate the thread.

wakeup
: Change the thread in running mode.

run
: Get the thread into running mode. Switch thread.

raise
: Raise exception to the thread.

There are no methods to get other threads into sleep mode. If you want to get your own thread into sleep mode, then you can use the sleep or stop (a class method of the Thread class) method. If you use the sleep method, then you can stop the execution up to the specified time (or forever).

You can also investigate all threads within a process with the following class methods:

Thread.list
: Lists all live threads

Thread.main
: Returns the main thread

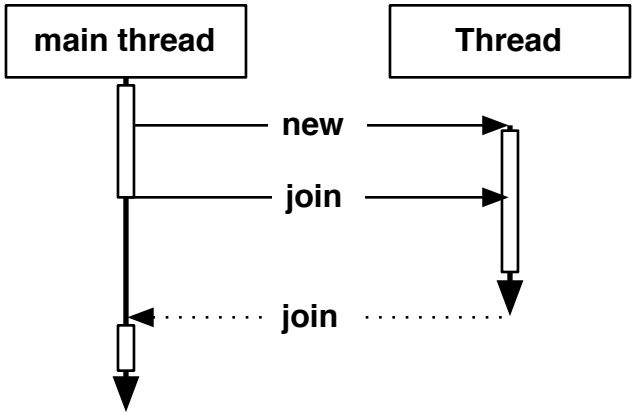


Figure 25—Threads waiting to join

Thread.current
: Returns the currently running thread

Let's try this with irb.

When irb first starts, it should have only one thread as the main thread.

```
% irb --prompt simple
>> Thread.list
=> [#<Thread:0x40.... run>]
>> Thread.list[0] == Thread.main
=> true
>> Thread.current == Thread.main
=> true
```

Here is the code to generate numbers from 0 to 9. However, this uses sleep for each iteration.

```
>> th = Thread.new { 10.times {|x| sleep; p [Thread.current, x]} }
=> #<Thread:0x... sleep>
```

The thread of th is in sleep mode. The Thread.list, status, alive?, and stop? show you the statuses of each thread.

```
>> Thread.list
=> [#<Thread:0x..... sleep>, #<Thread:0x.... run>]
>> th.status
=> "sleep"
>> th.alive?
=> true
```

```
>> th.stop?
=> true
```

Let's wake up th via the wakeup method.

```
>> th.wakeup
=> #<Thread:0x2.... run>
[#<Thread:0x2.... run>, 0]>>
```

th is now in running mode and prints out 0. Because both the main thread and the th thread print out strings, the screen output may not be indented properly. th should now be in sleep mode as it moves to the next iteration.

```
>> th.status
=> "sleep"
```

Let's change the status of th into run mode. run changes the thread immediately, so the output styling may differ from when using wakeup.

```
>> th.run
[#<Thread:0x2..... run>, 1]=> #<Thread:0x2..... run>

>> th.run
[#<Thread:0x2..... run>, 2]=> #<Thread:0x2..... run>

>> th.wakeup
=> #<Thread:0x2..... run>
[#<Thread:0x2..... run>, 3]
```

Next, let's terminate by raising an exception to th.

```
>> th.raise('stop!')
=> nil
>> Thread.list
=> [#<Thread:0x..... run>]
>> th.status
=> nil
>> th.alive?
=> false
>> th.stop?
=> true
```

Since it was terminated by an exception, th.status should return nil. Thread.list returns only a live thread, so it should return only the main thread. th is already terminated, alive? should return false, and stop? should return true.

How about th.value? It should raise the same exception as the one raised by th.raise(RuntimeError: stop!).

```
>> th.value
RuntimeError: stop!
        from (irb):7
```

```
        from (irb):11:in `value'
        from (irb):11
```

What if it terminated normally? This time, print out the numbers from 0 to 9 and end with 'complete' immediately without sleep. Once the main thread is created, call th.join to wait for the end of the thread execution.

```
>> th = Thread.new { 10.times { |x| p x} ; 'complete' }
>> th.join
0
1
2
3
4
5
6
7
8
9
=> #<Thread:0x..... dead>
>> th.status
=> false
>> th.alive?
=> false
>> th.stop?
=> true
```

th.join usually waits for the thread to end. Since it already ended in this case, it returns immediately if you run th.join again.

```
>> th.join
=> #<Thread:0x2ac4d35c dead>
```

th.value should return 'complete', which was evaluated by the thread right before it ended.

```
>> th.value
=> "complete"
```

This gives you basic control of threading; we've tried all the options in irb. In the next section, we'll go through how to safely pass objects among threads.