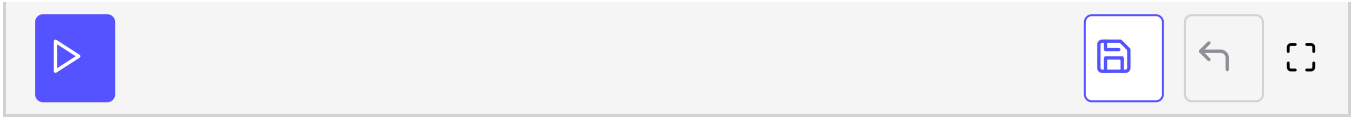
...continued

This lesson continues the discussion of Mutex in Ruby.

> A mistaken belief is that we only need to synchronize access to shared resources when writing to them and not when reading them. Consider the snippet below. We have an array of size two initialized with true values. The writer thread flips the boolean values in all the array elements in a loop. Since the array is shared, we guard the write to the array in a mutex synchronize block. We have another reader thread that outputs the array to the console. With proper synchronization, the output on the console will consist of either all true or all false values. Run the widget below and see what happens:

```
mutex = Mutex.new
array = []

2.times do |i|
  array[i] = true
end


writer = Thread.new do

  while true
    mutex.synchronize {
      array.each_with_index do |el, i|
        array[i] = !array[i]
      end
    }
    # sleep(0.01)
  end
end


reader = Thread.new do

  while true
    puts array.to_s()
    sleep(0.1)
  end
```

```
end


sleep(5)
```

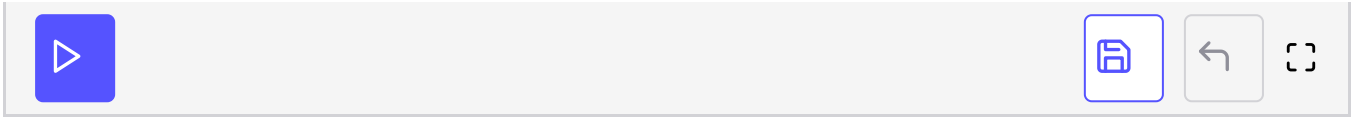


You can observe from the output that the reader thread sees a partially updated array because it reads from the array without synchronization. The reason for this inconsistency is that the writer thread doesn't atomically update the array. It updates the array elements one by one and another thread can get scheduled whilst the writer is still updating the array. If the newly scheduled thread reads the array, it sees a partially updated structure. This is also referred to as reading a **torn value** in some multithreaded languages.

Contrast the above program with the one below. We update a variable `item` with an integer value of 5 or 6 alternatively in a writer thread. A reader thread outputs the value of the variable in a loop. What would be the output of the code widget now? Will you see a torn value? What does a torn value even mean in the context of updating an integer?

```
mutex = Mutex.new
item = 0

writer = Thread.new do

  while true

    mutex.synchronize {
      if item == 0
          item = 2147483647
      else
          item = 0
      end
    }

  end
end


reader = Thread.new do

  while true
```

```
    puts item.to_s
  end
end


sleep(10)
```

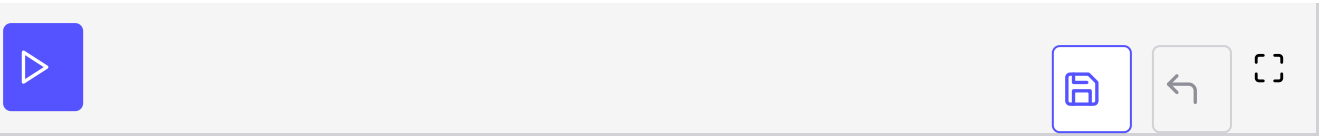


You'll always see a value of either 0 or 2147483647 and nothing arbitrary. If an integer is represented as 32 bits on your machine, it can't happen that the lower 16 bits are updated and then the higher 16 bits are updated in two separate machine instructions. If this was the case, the reader thread could see an arbitrary value other than 0 or 2147483647. For example, the reader thread reads the `item` variable right when the lower 16 bits are updated by the writer thread and the higher 16 bits not yet. As long as an operation can be completed as an atomic machine instruction, we will not see a torn value. However, that doesn't imply the code is thread-safe. If the logic of the program relies on reading the latest value of the variable `item`, then synchronizing using a mutex is must.

In Ruby all types are treated as objects and the language specification doesn't say when writes or reads are atomic. We can ensure atomicity of operations using a mutex block. Note that locking is an expensive operation and in some scenarios implementing atomicity can be an overkill. Consider the typical problem of polling for a boolean flag by a spawned thread to know when to exit. The scenario is setup in the code widget below:

```
keepRunning = true

thread = Thread.new do

  while keepRunning do
    # spin uselessly
  end
  puts "Child thread exiting"
end


sleep(3)
keepRunning = false
puts "Waiting for child thread to terminate"
```

```
thread.join()
```



Can we consider the above code thread-safe? Since only one thread ever reads the boolean flag and only one thread ever writes to it, the program is correct. A similar setup in Java or C# isn't thread-safe since the statements can be reordered for optimization or there can be cache coherence issues.

As an example, in the case of Java, a write to a 64 bit long or double value isn't atomic. It consists of two writes, one to the first half of 32 bits and the second to the second half of the 32 bits. Java's memory model is different and has constructs to mitigate this situation, however, the takeaway from this discussion is to remember that synchronization is required whenever reading or writing shared data.