Global Interpreter Lock

This lesson discusses the global interpreter lock, also known as GIL and its effects.

Global Interpreter Lock

The global interpreter lock or GIL for short, is a limitation in the MRI Ruby interpreter that allows for only one thread to execute at any given time. Unlike C# or Java, which are truly multithreaded, Ruby can only run a single thread when multiple cores are available. Note that only the standard Ruby MRI implementation suffers from the GIL restriction. Another well known programming language with GIL limitation is the CPython interpreter.

Reasons for GIL

The reason to have GIL is that of Ruby users too. But more importantly, it eliminates race conditions for Ruby developers, although it does not completely solve the issues of multithreading.

- Makes the implementation of the Ruby interpreter and the standard library much easier.
- Non-thread safe C library can be easily integrated.
- Eliminates need for locks or synchronization around the internal state/data-structures maintained by Ruby.

One of the reasons Python needs the GIL is because the garbage collector uses a reference counting algorithm that would need a major overhaul to get it to work in a thread safe way without the GIL. Ruby, however, uses a different garbage collection algorithm called mark and sweep but language implementation becomes much easier when not having to deal

with thread-safety issues.

Ruby threads release GIL when executing blocking I/O operations such as HTTP requests, DB queries, writing or reading from disk and even when invoking `sleep()`.

GIL as a Lock?

Since GIL only allows a single thread to execute at any time, does that imply that our Ruby code is guaranteed to be thread-safe? GIL only serves to protect the internals of MRI and we can never be relied upon to write thread-safe code. Very briefly, under the hood, a timer thread is spawned whenever a thread is waiting to acquire the GIL. Once the timer thread expires, an interrupt flag is set on the thread currently holding the GIL. The GIL-holding thread checks for the interrupts after finishing the execution of and just before returning the result of the currently executing C method. If it finds the interrupt set, it releases the GIL and invokes the scheduler to schedule a different thread. This behavior assures that Ruby methods implemented in C are atomic. However, methods implemented with Ruby code have no atomicity guarantees in MRI as they can expand into multiple native C methods and the thread executing the Ruby method can get context-switched after executing any of the constituent native C methods.

One commonly quoted online example is the snippet `array << 1` which translates to the following C code:

```
rb_ary_push(VALUE ary, VALUE item)
{
    long idx = RARRAY_LEN(ary);

    ary_ensure_room_for_push(ary, 1);
    RARRAY_ASET(ary, idx, item);
    ARY_SET_LEN(ary, idx + 1);
    return ary;
}
```

Because the way GIL works currently, the snippet `array << 1` is atomic because it will expand into a native C method that will be executed atomically. In contrast consider the following Ruby method:

```
def isThreadSafe()
  array << 1

  array << 2
end
```

The above method `isThreadSafe()` isn't thread-safe because another thread can be scheduled right after the first array insertion statement is executed. In other words composition of thread-safe operations doesn't imply thread-safety.

Last but not the least, working of the GIL is an implementation detail, prone to change and without any documentation and should never be relied upon for ensuring thread-safety of Ruby code.