Lab 7: Multithreading in Ruby

Learning Objectives:

1. Threads in Ruby
2. Thread Lifecycle
3. Creating and Finishing a Thread
4. Thread Exclusion
5. Additional Information

Supporting files:

1. thread.rb
2. mutex.rb
3. process.rb

---

1. Threads in Ruby

A multithreaded program has more than one thread of execution. Within each thread, statements are executed sequentially, but the threads themselves may be executed in parallel on a multicore CPU, for example. Often on a single CPU machine, multiple threads are not actually executed in parallel, but parallelism is simulated by interleaving the execution of the threads. Ruby makes it easy to write multi-threaded programs with the Thread class. Ruby threads are a lightweight and efficient way to achieve concurrency in our code.

Threads help to run multiple operations within a single process. As an example, the attached file *thread.rb* demonstrates a program that uses 3 threads, each thread prints a fruit name associated with its ID. The output is like the following.

```
Thread.abort_on_exception = true

@fruits = ['apple', 'banana', 'cherry']

def print_name(id)
  5.times do |t|
    puts @fruits[id]
  end
end

threads = []

3.times do |i|
  threads << Thread.new(i) {|id| print_name(id) }
end

threads.each { |t|  t.join }
```

Output:

```
banana
cherry
apple
banana
banana
apple
apple
apple
```

```
banana
cherry
apple
banana
cherry
cherry
cherry
```

The overlap of the fruits names proves that all 3 threads started working together.

---

2. Thread Lifecycle

A new thread is created with Thread.new. We can also use the synonyms Thread.start and Thread.fork. There is no need to start a thread after creating it, it begins running automatically when CPU resources become available. The Thread class defines a number of methods to query and manipulate the thread while it is running. A thread runs the code in the block associated with the call to Thread.new and then it stops running.

The value of the last expression in that block is the value of the thread, and can be obtained by calling the value method of the Thread object. If the thread has run to completion, then the value returns the thread's value right away. Otherwise, the value method blocks and does not return until the thread has completed.

The class method Thread.current returns the Thread object that represents the current thread. This allows threads to manipulate themselves. The class method Thread.main returns the Thread object that represents the main thread. This is the initial thread of execution that began when the Ruby program was started. We can wait for a particular thread to finish by calling that thread's Thread.join method. The calling thread will block until the given thread is finished.

---

3 Creating and Finishing a Thread

In order to create a thread pass an unique integer while creating a thread, e.g.., *Thread.new(0)*. Then a worker function is to be set which will be called asynchronously.
When a thread is done working, in order to finish the thread the *.join()* function is unblocked from execution routine.

---

4 Thread Exclusion

If two threads share access to the same data, and at least one of the threads modifies that data, we must take special care to ensure that no thread can ever see the data in an inconsistent state. This is called thread exclusion. Mutex is necessary to synchronize between threads and avoid deadlocks.

Let's recheck the output from *thread.rb*. We can see some overlap of fruit names which denotes that all 3 threads tried to write to stdout without any control. If we want them to do synchronize properly, an instance of Mutex has to be used. Any mutex must be locked before continuing any task, and must be unlocked after processing the task. So other thread can use that mutex until it is unlocked. For example, looking at the attached file *mutex.rb*:

```
Thread.abort_on_exception = true

@fruits = ['apple', 'banana', 'cherry']
```

```
@mutex = Mutex.new

def print_name(id)
  @mutex.lock

  5.times do |t|
    puts @fruits[id]
  end

  @mutex.unlock
end

threads = []

3.times do |i|
  threads << Thread.new(i) {|id| print_name(id) }
end

threads.each { |t|  t.join }
```

Output:

cherry
cherry
cherry
cherry
cherry
banana
banana
banana
banana
banana
apple
apple
apple
apple
apple

This means, thread 3 (ID 3, cherry) started it worker function first, and locked the mutex, and printed 5 lines. When it released the mutex, thread 2 (ID 2, banana) got the chance to print to stdout and finally thread 1 (ID 1, apple) got access to finish its job.

---

5 Additional Information

Process

Any process has 3 IO channels called stdout, stdin and stderror. Using *open3* module, any command or process can be executed from Ruby program.

capture2
The capture2 function provides direct access to output and status of any command that was ran. Say we want to execute "ls /" and get the output. The following snippet is enough for such purpose.

```
output, status = Open3.capture2("ls /usr")
print output
print status
```

status.pid holds the process ID of the command, and *status.exitstatus* holds the runtime return code which would be 0 if no error was occurred.

popen3

The popen3 function provides access to all 3 channels of a command - input, output and error. Which means, we can write into the process, read it and get error text if there is any.

Say we want to run *ruby* command. Let's check the following snippet.

```
Open3.popen3("ruby") { |stdin, stdout, stderr|
   stdin.write "print RUBY_VERSION"
   stdin.close # Send EOF signal
   output = stdout.read
   print "Output: #{output}\n"
}
```

Here, we wrote a one line Ruby code ‘print RUBY_VERSION’ and sent EOF (Ctrl+D) to Ruby interpreter. Then the output is read, and printed. Error messages could be read using stderr channel also.
