Ruby Programming Lab 6

## Suryakumar P 21MIS1146

1. Write separate program using:
2. Yield and resume

Code:

counter = Fiber.new do

  count = 0

  loop do

    count += 1

    Fiber.yield(count)

  end

end

# Example 1: Basic resume and yield

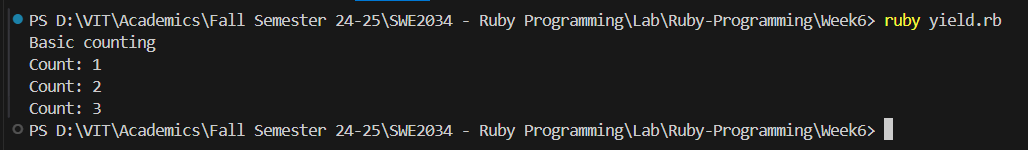
puts "Example 1: Basic counting"

3.times do

  puts "Count: #{counter.resume}"

end

Output:



1. Transfer

Code:

fiber1 = Fiber.new do

    puts "Fiber 1: Starting"

    Fiber.yield

    puts "Fiber 1: Resuming"

  end

  fiber2 = Fiber.new do

    puts "Fiber 2: Starting"

    fiber1.resume

    puts "Fiber 2: Resuming"

  end

  fiber2.resume

Output:



1. Raise

Code:

fiber1 = Fiber.new do

    begin

      puts "Fiber 1: Starting"

      raise "An error occurred in Fiber 1"

    rescue => e

      puts "Fiber 1: Caught exception: #{e.message}"

    ensure

      puts "Fiber 1: Cleaning up"

    end

  end

  fiber2 = Fiber.new do

    begin

      puts "Fiber 2: Starting"

      fiber1.resume

      puts "Fiber 2: Resuming"

    rescue => e

      puts "Fiber 2: Caught exception: #{e.message}"

    ensure

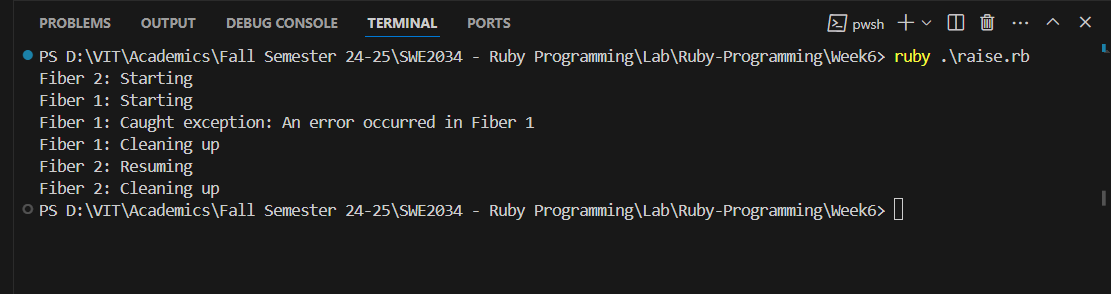
      puts "Fiber 2: Cleaning up"

    end

  end

  fiber2.resume

Output:



2. Create 10 threads, each of which sleep for a random amount of time and then prints a message.

Code:

threads = []

10.times do |i|

  threads << Thread.new do

    sleep\_duration = rand(1..5)

    sleep(sleep\_duration)

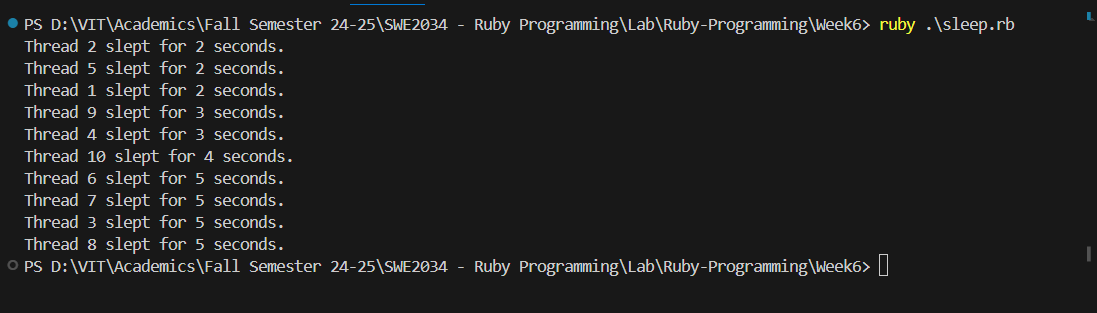
    puts "Thread #{i + 1} slept for #{sleep\_duration} seconds."

  end

end

threads.each(&:join)

Output:



3. Create a local variable for a main thread, additional threads and fiber and prints the value of it.

Code:

main\_thread\_variable = "Main Thread Variable"

fiber = Fiber.new do

  fiber\_variable = "Fiber Variable"

  puts "Inside Fiber: #{fiber\_variable}"

  puts "Inside Fiber accessing main thread variable: #{main\_thread\_variable}"

end

threads = 3.times.map do |i|

  Thread.new do

    thread\_variable = "Thread #{i + 1} Variable"

    puts "Inside Thread #{i + 1}: #{thread\_variable}"

    puts "Inside Thread #{i + 1} accessing main thread variable: #{main\_thread\_variable}"

  end

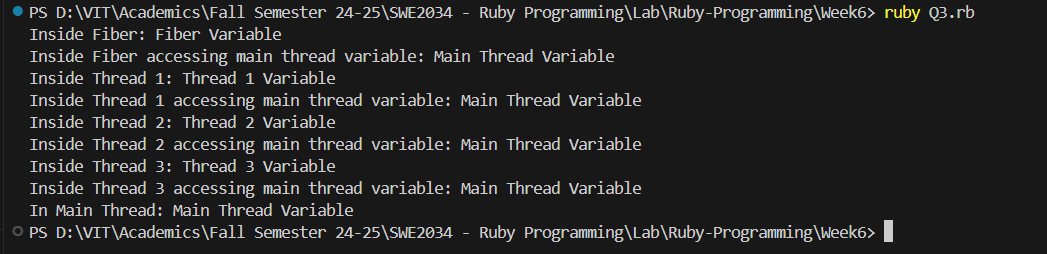
end

fiber.resume

threads.each(&:join)

puts "In Main Thread: #{main\_thread\_variable}"

Output:



4. Local variable values in Nested Thread within a Fiber.

Code:

# Create a fiber

fiber = Fiber.new do

    fiber\_variable = "Fiber Variable"

    puts "Inside Fiber: #{fiber\_variable}"

    # Create a thread within the fiber

    thread = Thread.new do

      thread\_variable = "Thread Variable"

      puts "Inside Thread within Fiber: #{thread\_variable}"

      puts "Inside Thread within Fiber accessing fiber variable: #{fiber\_variable}"

    end

    # Wait for the thread to complete

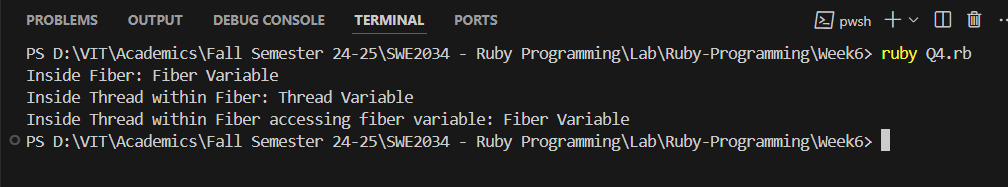
    thread.join

  end

  # Resume the fiber

  fiber.resume

Output:



5. Local variable values in Nested Fiber within a Thread.

Code:

# Create a thread

thread = Thread.new do

    thread\_variable = "Thread Variable"

    puts "Inside Thread: #{thread\_variable}"

    # Create a fiber within the thread

    fiber = Fiber.new do

      fiber\_variable = "Fiber Variable"

      puts "Inside Fiber within Thread: #{fiber\_variable}"

      puts "Inside Fiber within Thread accessing thread variable: #{thread\_variable}"

    end

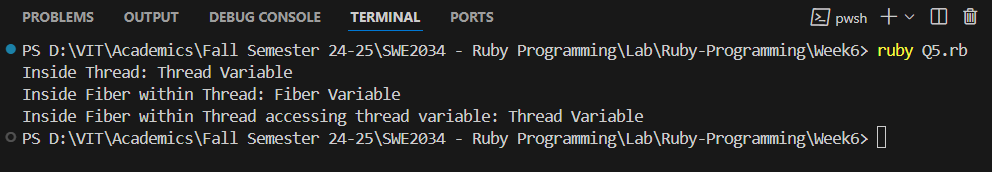
    # Resume the fiber

    fiber.resume

  end

  # Wait for the thread to complete

  thread.join

Output:  


6. Multi Thread sharing same variable address space.

Code:

shared\_counter = 0

mutex = Mutex.new

threads = 10.times.map do |i|

  Thread.new do

    10.times do

      mutex.synchronize do

        shared\_counter += 1

      end

    end

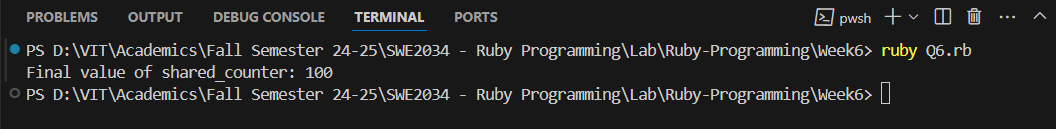
  end

end

threads.each(&:join)

puts "Final value of shared\_counter: #{shared\_counter}"

Output:



7. Write a separate program using the following functions:

a. Thread – Stop and Run

Code:  
running = true

thread = Thread.new do

  while running

    puts "Thread is running..."

    sleep(1)

  end

  puts "Thread has stopped."

end

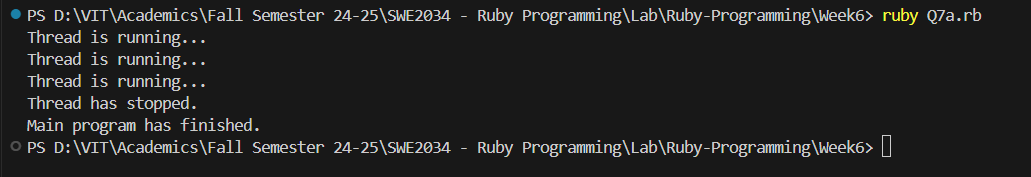
sleep(3)

running = false

thread.join

puts "Main program has finished."

Output:



b. Thread – wake up

Code:

thread = Thread.new do

    puts "Thread is going to sleep..."

    sleep

    puts "Thread has been woken up!"

  end

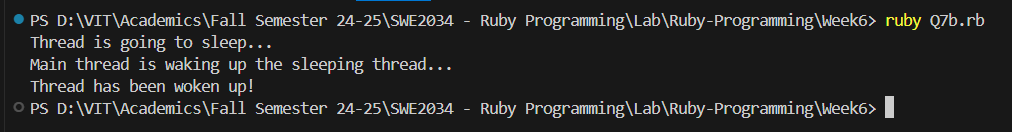
  sleep(2)

  puts "Main thread is waking up the sleeping thread..."

  thread.wakeup

  thread.join

Output:



c. Thread Value

Code:

thread = Thread.new do

    sum = 0

    1.upto(10) do |i|

      sum += i

    end

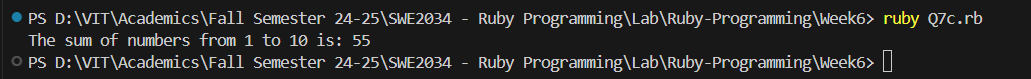
    sum

  end

  result = thread.value

  puts "The sum of numbers from 1 to 10 is: #{result}"

Output:



1. Thread – pass

Code:

t1 = Thread.new do

    5.times do |i|

      puts "Thread 1 - Count: #{i}"

      Thread.pass

    end

  end

  t2 = Thread.new do

    5.times do |i|

      puts "Thread 2 - Count: #{i}"

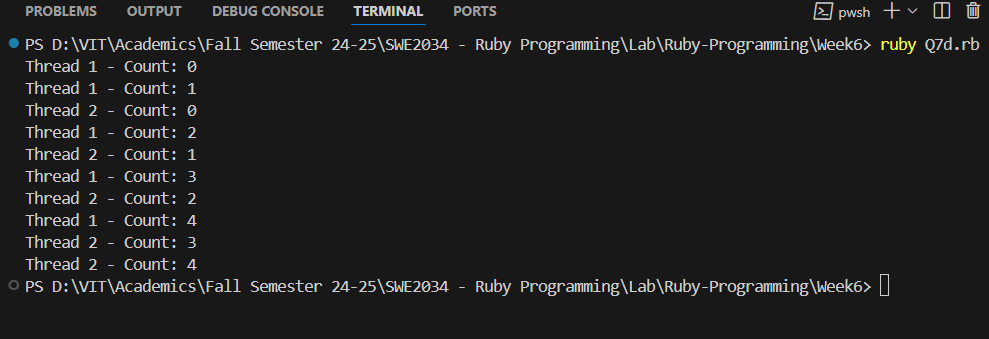
      Thread.pass

    end

  end

  [t1, t2].each(&:join)

Output:



1. Thread – Priority

Code:

# Create a high-priority thread

high\_priority\_thread = Thread.new do

    5.times do

      puts "High priority thread is running"

      sleep(0.1)

    end

  end

  # Create a low-priority thread

  low\_priority\_thread = Thread.new do

    5.times do

      puts "Low priority thread is running"

      sleep(0.1)

    end

  end

  # Set thread priorities

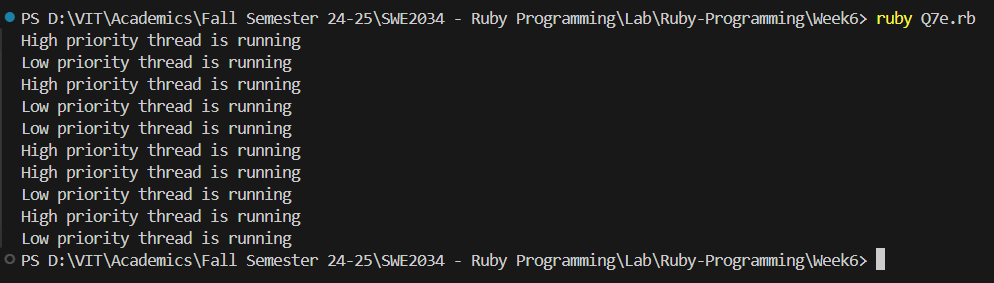
  high\_priority\_thread.priority = 5

  low\_priority\_thread.priority = 1

  # Wait for both threads to complete

  high\_priority\_thread.join

  low\_priority\_thread.join

Output:  


1. Thread – Mutex

Code:

shared\_counter = 0

mutex = Mutex.new

threads = 10.times.map do |i|

  Thread.new do

    10.times do

      mutex.synchronize do

        shared\_counter += 1

      end

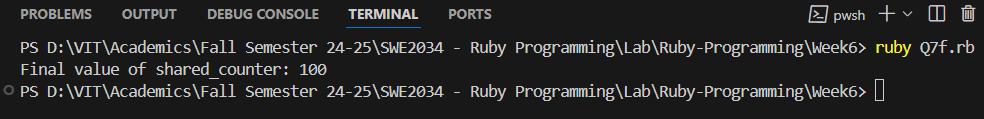
    end

  end

end

threads.each(&:join)

puts "Final value of shared\_counter: #{shared\_counter}"

Output:  


1. Thread – fork

Code:

pid = fork do

    5.times do |i|

      puts "Child process - Count: #{i}"

      sleep(0.5)

    end

  end

  if pid

    5.times do |i|

      puts "Parent process - Count: #{i}"

      sleep(0.5)

    end

    Process.wait(pid)

  end

Output:

Fork works only on Unix based System

Parent process - Count: 0

Child process - Count: 0

Parent process - Count: 1

Child process - Count: 1

Parent process - Count: 2

Child process - Count: 2

Parent process - Count: 3

Child process - Count: 3

Parent process - Count: 4

Child process - Count: 4