RUBY - MULTITHREADING

Traditional programs have a single *thread of execution*: the statements or instructions that comprise the program are executed sequentially until the program terminates.

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 your code.

Creating Ruby Threads:

To start a new thread, just associate a block with a call to *Thread.new*. A new thread will be created to execute the code in the block, and the original thread will return from *Thread.new* immediately and resume execution with the next statement:

```
# Thread #1 is running here
Thread.new {
  # Thread #2 runs this code
}
# Thread #1 runs this code
```

Example:

Here is an example, which shows how we can use multi-threaded Ruby program.

```
#!/usr/bin/ruby

def func1
   i=0
   while i<=2
      puts "func1 at: #{Time.now}"
      sleep(2)
      i=i+1
   end
end

def func2
   j=0
   while j<=2
      puts "func2 at: #{Time.now}"
      sleep(1)
      j=j+1
   end
end

puts "Started At #{Time.now}"
t1=Thread.new{func1()}
t2=Thread.new{func2()}
t1.join
t2.join
puts "End at #{Time.now}"
```

This will produce following result:

```
Started At Wed May 14 08:21:54 -0700 2008
func1 at: Wed May 14 08:21:54 -0700 2008
func2 at: Wed May 14 08:21:54 -0700 2008
func2 at: Wed May 14 08:21:55 -0700 2008
func1 at: Wed May 14 08:21:56 -0700 2008
```

```
func2 at: Wed May 14 08:21:56 -0700 2008
func1 at: Wed May 14 08:21:58 -0700 2008
End at Wed May 14 08:22:00 -0700 2008
```

Thread Lifecycle:

A new threads are created with *Thread.new*. You 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.

You 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.

Threads and Exceptions:

If an exception is raised in the main thread, and is not handled anywhere, the Ruby interpreter prints a message and exits. In threads other than the main thread, unhandled exceptions cause the thread to stop running.

If a thread t exits because of an unhandled exception, and another thread **s** calls *t.join or t.value,* then the exception that occurred in **t** is raised in the thread **s**.

If *Thread.abort_on_exception* is *false*, the default condition, an unhandled exception simply kills the current thread and all the rest continue to run.

If you would like any unhandled exception in any thread to cause the interpreter to exit, set the class method *Thread.abort_on_exception* to *true*.

```
t = Thread.new { ... }
t.abort_on_exception = true
```

Thread Variables:

A thread can normally access any variables that are in scope when the thread is created. Variables local to the block of a thread are local to the thread, and are not shared.

Thread class features a special facility that allows thread-local variables to be created and accessed by name. You simply treat the thread object as if it were a Hash, writing to elements using []= and reading them back using [].

In this example, each thread records the current value of the variable count in a threadlocal variable with the key *mycount*.

```
#!/usr/bin/ruby

count = 0
arr = []

10.times do |i|
   arr[i] = Thread.new {
```

```
      sleep(rand(0)/10.0)
      Thread.current["mycount"] = count
      count += 1
   }
end

arr.each {|t| t.join; print t["mycount"], ", " }
puts "count = #{count}"
```

This produces the following result:

```
8, 0, 3, 7, 2, 1, 6, 5, 4, 9, count = 10
```

The main thread waits for the subthreads to finish and then prints out the value of *count* captured by each.

Thread Priorities:

The first factor that affects thread scheduling is thread priority: high-priority threads are scheduled before low-priority threads. More precisely, a thread will only get CPU time if there are no higher-priority threads waiting to run.

You can set and query the priority of a Ruby Thread object with *priority*= and *priority*. A newly created thread starts at the same priority as the thread that created it. The main thread starts off at priority 0.

There is no way to set the priority of a thread before it starts running. A thread can, however, raise or lower its own priority as the first action it takes.

Thread Exclusion:

If two threads share access to the same data, and at least one of the threads modifies that data, you 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 a class that implements a simple semaphore lock for mutually exclusive access to some shared resource. That is, only one thread may hold the lock at a given time. Other threads may choose to wait in line for the lock to become available, or may simply choose to get an immediate error indicating that the lock is not available.

By placing all accesses to the shared data under control of a *mutex*, we ensure consistency and atomic operation. Let's try to examples, first one without mutax and second one with mutax:

Example without Mutax:

```
#!/usr/bin/ruby
require 'thread'

count1 = count2 = 0
difference = 0
counter = Thread.new do
   loop do
      count1 += 1
      count2 += 1
   end
end
spy = Thread.new do
   loop do
      difference += (count1 - count2).abs
   end
end
sleep 1
puts "count1 :  #{count1}"
puts "count2 :  #{count2}"
puts "difference : #{difference}"
```

This will produce the following result:

```
count1 :  1583766
count2 :  1583766
difference : 637992
```

```
#!/usr/bin/ruby
require 'thread'
mutex = Mutex.new

count1 = count2 = 0
difference = 0
counter = Thread.new do
   loop do
      mutex.synchronize do
         count1 += 1
         count2 += 1
      end
    end
end
spy = Thread.new do
   loop do
       mutex.synchronize do
          difference += (count1 - count2).abs
       end
   end
end
sleep 1
mutex.lock
puts "count1 :  #{count1}"
puts "count2 :  #{count2}"
puts "difference : #{difference}"
```

This will produce the following result:

```
count1 :  696591
count2 :  696591
difference : 0
```

Handling Deadlock:

When we start using *Mutex* objects for thread exclusion we must be careful to avoid *deadlock*. Deadlock is the condition that occurs when all threads are waiting to acquire a resource held by another thread. Because all threads are blocked, they cannot release the locks they hold. And because they cannot release the locks, no other thread can acquire those locks.

This is where *condition variables* come into picture. A *condition variable* is simply a semaphore that is associated with a resource and is used within the protection of a particular *mutex*. When you need a resource that's unavailable, you wait on a condition variable. That action releases the lock on the corresponding *mutex*. When some other thread signals that the resource is available, the original thread comes off the wait and simultaneously regains the lock on the critical region.

Example:

```
#!/usr/bin/ruby
require 'thread'
mutex = Mutex.new

cv = ConditionVariable.new
a = Thread.new {
   mutex.synchronize {
      puts "A: I have critical section, but will wait for cv"
      cv.wait(mutex)
      puts "A: I have critical section again! I rule!"
   }
}
```

```
puts "(Later, back at the ranch...)"

b = Thread.new {
   mutex.synchronize {
      puts "B: Now I am critical, but am done with cv"
      cv.signal
      puts "B: I am still critical, finishing up"
   }
}
a.join
b.join
```

This will produce the following result:

```
A: I have critical section, but will wait for cv
(Later, back at the ranch...)
B: Now I am critical, but am done with cv
B: I am still critical, finishing up
A: I have critical section again! I rule!
```

Thread States:

There are five possible return values corresponding to the five possible states as shown in the following table. The *status* method returns the state of the thread.

Thread state	Return value
Runnable	run
Sleeping	Sleeping
Aborting	aborting
Terminated normally	false
Terminated with exception	nil

Thread Class Methods:

Following methods are provided by *Thread* class and they are applicable to all the threads available in the program. These methods will be called as using *Thread* class name as follows:

```
Thread.abort_on_exception = true
```

Here is complete list of all the class methods available:

SN	Methods with Description
1	**Thread.abort_on_exception** Returns the status of the global *abort on exception* condition. The default is *false*. When set to *true*, will cause all threads to abort $theprocesswillexit(0)$ if an exception is raised in any thread.
2	**Thread.abort_on_exception=** When set to *true*, all threads will abort if an exception is raised. Returns the new state.
3	**Thread.critical**

Returns the status of the global *thread critical* condition.

4 **Thread.critical=**

Sets the status of the global *thread critical* condition and returns it. When set to *true*, prohibits scheduling of any existing thread. Does not block new threads from being created and run. Certain thread operations *suchasstoppingorkillingathread*, *sleepinginthecurrentthread*, *andraisinganexception* may cause a thread to be scheduled even when in a critical section.

5 **Thread.current**

Returns the currently executing thread.

6 **Thread.exit**

Terminates the currently running thread and schedules another thread to be run. If this thread is already marked to be killed, *exit* returns the *Thread.* If this is the main thread, or the last thread, exit the process.

7 **Thread.fork { block }**

Synonym for Thread.new .

8 **Thread.kill***aThread*

Causes the given *aThread* to exit

9 **Thread.list**

Returns an array of *Thread* objects for all threads that are either runnable or stopped. Thread.

10 **Thread.main**

Returns the main thread for the process.

11 **Thread.new**[*arg*] * **{| args | block }**

Creates a new thread to execute the instructions given in block, and begins running it. Any arguments passed to *Thread.new* are passed into the block.

12 **Thread.pass**

Invokes the thread scheduler to pass execution to another thread.

13 **Thread.start**[*args*] * **{| args | block }**

Basically the same as *Thread.new* . However, if class *Thread* is subclassed, then calling *start* in that subclass will not invoke the subclass's *initialize* method.

14 **Thread.stop**

Stops execution of the current thread, putting it into a *sleep* state, and schedules execution of another thread. Resets the *critical* condition to false.

Thread Instance Methods:

These methods are applicable to an instance of a thread. These methods will be called as using an instance of a *Thread* as follows:

```
#!/usr/bin/ruby

thr = Thread.new do   # Calling a class method new
   puts "In second thread"
   raise "Raise exception"
end
thr.join   # Calling an instance method join
```

Here is complete list of all the instance methods available:

SN	Methods with Description
1	**thr[aSymbol]** Attribute Reference - Returns the value of a thread-local variable, using either a symbol or a *aSymbol* name. If the specified variable does not exist, returns *nil*.
2	**thr[aSymbol] =** Attribute Assignment - Sets or creates the value of a thread-local variable, using either a symbol or a string.
3	**thr.abort_on_exception** Returns the status of the *abort on exception* condition for *thr*. The default is *false*.
4	**thr.abort_on_exception=** When set to *true*, causes all threads *includingthemainprogram* to abort if an exception is raised in *thr*. The process will effectively *exit0*.
5	**thr.alive?** Returns *true* if *thr* is running or sleeping.
6	**thr.exit** Terminates *thr* and schedules another thread to be run. If this thread is already marked to be killed, *exit* returns the *Thread*. If this is the main thread, or the last thread, exits the process.
7	**thr.join** The calling thread will suspend execution and run *thr*. Does not return until *thr* exits. Any threads not joined will be killed when the main program exits.
8	**thr.key?** Returns *true* if the given string *orsymbol* exists as a thread-local variable.
9	**thr.kill** Synonym for *Thread.exit* .

10 **thr.priority**

Returns the priority of *thr*. Default is zero; higher-priority threads will run before lower priority threads.

11 **thr.priority=**

Sets the priority of *thr* to an Integer. Higher-priority threads will run before lower priority threads.

12 **thr.raise***anException*

Raises an exception from *thr*. The caller does not have to be *thr*.

13 **thr.run**

Wakes up *thr*, making it eligible for scheduling. If not in a critical section, then invokes the scheduler.

14 **thr.safe_level**

Returns the safe level in effect for *thr*.

15 **thr.status**

Returns the status of *thr*: *sleep* if *thr* is sleeping or waiting on I/O, *run* if *thr* is executing, false if *thr* terminated normally, and *nil* if *thr* terminated with an exception.

16 **thr.stop?**

Returns *true* if *thr* is dead or sleeping.

17 **thr.value**

Waits for thr to complete via *Thread.join* and returns its value.

18 **thr.wakeup**

Marks *thr* as eligible for scheduling, it may still remain blocked on I/O, however.