Chapter 12

Concurrency in Ruby
Intro to Erlang

Concurrency in Ruby

- Ruby has both threads and processes
- Ruby threads are operating system independent
- The `Thread` class provides support for creating and using threads
- Other related classes include `Monitor`, `Sync`, `Mutex`, `Queue`

Mutex - Semaphore

```
class Counter1
  attr_reader :count
  def initialize
    @count = 0
    @mutex = Mutex.new
  end

  def inc
    @mutex.lock
      @count += 1
    @mutex.unlock
  end

c = Counter1.new
t1 = Thread.new {1000.times { c.inc } }
t2 = Thread.new {1000.times { c.inc } }

t1.join; t2.join
puts c.count
end
```

Monitors for Mutual Exclusion

```
require 'monitor'
class Counter2 < Monitor
  attr_reader :count

  def initialize
    @count = 0
    super
  end

  def inc
    synchronize do
      @count += 1
    end
  end
end

c = Counter2.new
t1 = Thread.new {1000.times { c.inc } }
t2 = Thread.new {1000.times { c.inc } }
t1.join; t2.join
puts c.count
```

Monitors and Condition Variables

```
@music = Mutex.new
@violin = ConditionVariable.new

@violins_free = 2

def musician(n)
  loop do
    sleep rand(0)
    @music.synchronize do
      @violin.wait(@music) if @violins_free == 0
      @violins_free -= 1
      puts "#{n} has a violin"
    end

    sleep rand(0)
    puts "#{n}: is playing..."
    sleep rand(0)
    puts "#{n}: is finished..."
    @music.synchronize do
      @violins_free += 1
      @violin.signal if @violins_free == 1
    end
  end
end

threads = []
5.times {|i| threads << Thread.new {musician(i+1)}}

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

Modified example from *The Ruby Way*, by Hal Fulton

Intro to Erlang

- A functional language
- Developed for real-time fault-tolerant distributed telecom applications
- Concurrency uses very lightweight processes (not threads that use shared resources)

Atoms, Lists, Tuples

```
abc                 ← An atom
[1, 2, 3]           ← A list
{one, 1, two, 2}    ← A tuple
Name = "Erlang".    ← A variable (immutable)
```

Pattern Matching

= is really a pattern matching operator

Pattern = *Expression*

Evaluate the *Expression* and match the result against the *Pattern*

```
{Y,M,D} = date().
```

← Variables get bound to year, month, and day

Pattern Matching

```
-module(matching).
-export([number/1]).

number(1) -> one;
number(2) -> two;
number(3) -> three;
number(4) -> four;
number(5) -> five;
number(_) -> "Don't know".
```

Compile:
`c(matching.erl).`

Call:
`matching.number(3).`

Functions

```
-module(math).
-export([fact/1, fib/1]).

fact(0) -> 1;
fact(N) -> N * fact(N-1).

fib(1) -> 1;
fib(2) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
```

Lists

Erlang lists have similarities to Lisp/Scheme car and cdr:

```
[H|T] = [1,2,3,4,5].
```

`[H|T]` is a CONS cell with CAR = H and CDR = t.

`H` gets bound to `1` and `T` gets bound to `[2,3,4,5]`

List Functions

```
lists:append([a,b,c], [1,2,3,4]).
L = [1,2,3,4].
lists:map(fun(X) -> 2*X end, L).
Small = fun(X) -> X < 3 end.
lists:map(Small, L).
lists:all(Small, [0,1,2]).
lists:any(Small, [2,4,6]).
```

List Comprehension:

```
[X || X <- L, X rem 2 == 0]
[2*X || X <- L].
```

Concurrency Basics

Erlang concurrency is based on message passing

It uses *lightweight processes*, not threads

- No shared resources that are bug prone

Everything is done using three concurrency primitives:

spawn – creates a new process

! - send a message to a process

receive – receives a message sent to a process

Example - Translation Process

```
-module(translate).
-export([loop/0]).

loop() ->
  receive
    1 ->
       io:format("one~n"),
       loop();
    2 ->
       io:format("two~n"),
       loop();
    3 ->
       io:format("three~n"),
       loop();
    _ ->
       io:format("I don't know~n"),
       loop()
end.
```

Create process:

```
Pid = spawn(fun translate:loop/0).
```

Send message:

```
Pid ! 3.
```