Laboratory Work #2

on PPC

“Multithreading”

Done by: Ursu Dumitru
Verified by: Ciorbă Dumitru

Chișinău, 2014

Task:
Given the diagram of causal dependencies we have to study the methods of communication using pipes and the methods of sending the serialized objects.

Objectives:
- Use Ruby as the programming language
- Use Unix named pipes to implement the communication
- Respect the Thread order execution and communication from the graph (variant 5) (Annex A, fig.1).

Domain of the problem:
After some research on serialization and deserialization using the Ruby language, I found that there are tools built in the language for such task. The standard tools for this are:
- yaml as the main serialization format (support provided by 'yaml' library)
- Marshal as the main binary object serialization mechanism

For communication, the options are streams provided by the IO class, or as my development environment is a Unix-like operating system - named pipes. They resemble the usual files found in the file system, and are created using the “mkfifo” command. These pipes are persistent after reboots, and that is a nice feature to have.

Source code analysis:
addon.rb
```
# let's make the named pipe in the file system.

`mkfifo communication_pipe`

class Person
  attr_accessor :name, :age
  def initialize(name, age)
    @name = name
    @age = age
  end

  def to_s
    # let's get prettier objects printed out. #<Person:0x007fe1cfc04118> is not
    # fun
    name + ' - ' + age.to_s
  end

  def <=> other
    # let's make our objects sortable :D
    self.name <=> other.name
  end

  def to_xml
```

```
    "<xml>\n<name>#{self.name}</name>\n<age>#{self.age}</age>\n</xml
>"
  end
end
```

Initially we create a pipe, which will be used by all threads to send serialized data between them. Next, the Person class is built, adding getters and setters with atrr_acsessor. Then a few helper methods are created, as it's explained in the comments.

```
def creaza(mode)
  data = []
  $/="\n\n"
  File.open("resource.yml", "r").each do |object|
    data << YAML::load(object)
  end
```

An array of Person objects it obtained from the resource.yml file, and they are appended to the data array, using the shovel “<<” operator.

```
  pipe = open('communication_pipe', 'w+')
  if mode == :binary
    # we use print so we don't get unnecesary newlines in the binary format
    pipe.print Marshal::dump(data)
  elsif mode == :yaml
    pipe.puts YAML::dump(data)
  else
    puts "Choose a serialization technique!"
  end
  pipe.flush
  pipe.close
end
```

For conveniece, we handle to modes: binary and yaml mode, taking care to close the pipe.

```
def afisheaza(mode)
  pipe = open('communication_pipe', 'r+')
  if mode == :binary
    #puts pipe.gets
    puts Marshal::load(pipe.gets)
  elsif mode == :yaml
    puts YAML::load(pipe.gets)
  else
    puts "make your mind!"
  end
  pipe.close
```

```
end
```

The method “afisheaza” just outputs deserializes the binary or yaml content of the pipe. If nothing is available in the pipe, the program just hangs until something is pushed there. Note that there isn't a way to recognize the format by some kind of signature, if we run “afisheaza :yaml” but binary data was sent into the pipe, “bad things” will happen. The program will crash in such scenario, with the following error message:
“/home/dimon/.gem/ruby/2.0.0/gems/psych-2.0.1/lib/psych.rb:369:in `parse': (<unknown>): control characters are not allowed at line 1 column 1 (Psych::SyntaxError)”

```
#! /usr/bin/env ruby
require 'countdownlatch'
require 'yaml'
require_relative 'addons'
require 'debugger'

latch2 = CountDownLatch.new 1
latch3 = CountDownLatch.new 2
latch4 = CountDownLatch.new 2
latch5 = CountDownLatch.new 1
latch6 = CountDownLatch.new 2

t1 = Thread.new {
  creaza :binary
  latch2.countdown!
  latch3.countdown!
}

t2 = Thread.new {
  latch2.wait
  afisheaza :binary
  latch3.countdown!
  latch4.countdown!
}

t3 = Thread.new {
  latch3.wait
  creaza :yaml
  latch4.countdown!
}

t4 = Thread.new {
  latch4.wait
  # sort what we got from t3
  pipe = open('communication_pipe', 'r+')
  persons = YAML::load(pipe.gets)
```

```
  persons.sort!
```
The persons array has a sort method, thanks to inheritance from Array class: all we have to do is to define the “<=>” method.
```
  # sent to t5 for serialization
  w_pipe = open('another_pipe', 'w+')
```
Another pipe was needed, as in this thread the “communication_pipe” was already used, and we don't want the 6'th thread to receive 5'th thread messages.

```
  w_pipe.puts YAML::dump(persons)
  w_pipe.flush

  latch5.countdown!
  # set to t6 to display
  w_pipe.puts persons
  w_pipe.flush

  latch6.countdown!
  pipe.close
  w_pipe.close
}

t5 = Thread.new {
  latch5.wait
  # let's serialize these ones
  pipe = open('communication_pipe', 'r+')
  persons = YAML::load(pipe.gets)
  pipe.close
  file = open('persons.xml', 'w')
  file.puts persons.sort.each.map(&:to_xml)
```
A chaining of methods is used, the last one, map, invokes the given block once for each element of `self`, and so each array element is converted to the XML format. Because the format was pretty simple, I didn't use any library, I just interpolated the values in a xml string.

```
  file.close
  latch6.countdown!
}

t6 = Thread.new {
  latch6.wait
  pipe = open('communication_pipe', 'r+')
  sorted_persons = YAML::load(pipe.gets)
  puts sorted_persons
  pipe.close
}

t1.join
t2.join
```

```
t3.join
t4.join
t5.join
t6.join
```
Joining the threads ensures that the program won't finish before the threads are closed.

Conclusion:

In the process of doin this laboratory work, I've learned how to use unix pipes, serialization and deserialization to communicate between threads. The real cool stuff is that they can be even completely separated processes, even written in different languages. No API required, no language bindings.

Annex A

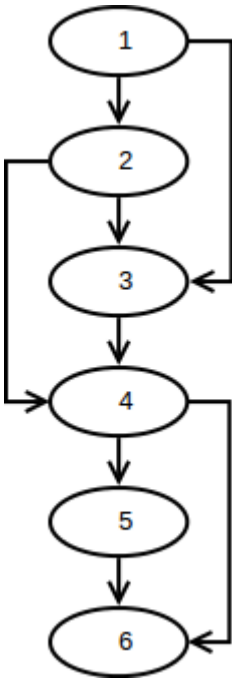

Figure 1. The threads control flow

Bibliography:

1. http://www.skorks.com/2010/04/serializing-and-deserializing-objects-with-ruby/
2. http://www.pauldix.net/2009/07/using-named-pipes-in-ruby-for-interprocess-communication.html