Ruby Programming Language Threads and Processes

MD.Shabeeruddin

Computer Science and Engineering
IIT Bombay

November 29, 2004

Introduction

- Threading often improves program response time by making heavy procedures run in the background.
- Ruby supports user-level threading.
- Ruby behaves uniformly on all platforms
- Ruby programs are thread compatibility.

Introduction

- Threading often improves program response time by making heavy procedures run in the background.
- Ruby supports user-level threading.
- Ruby behaves uniformly on all platforms
- Ruby programs are thread compatibility.

Introduction

- Threading often improves program response time by making heavy procedures run in the background.
- Ruby supports user-level threading.
- Ruby behaves uniformly on all platforms
- Ruby programs are thread compatibility.

Introduction

- Threading often improves program response time by making heavy procedures run in the background.
- Ruby supports user-level threading.
- Ruby behaves uniformly on all platforms
- Ruby programs are thread compatibility.

Thread Creation

Thread creation syntax

-
  ```
  Thread.new( [ arg ]* ) {| args | block }. }
  ```
- where
  - arg is actual parameters to thread.
  - args is formal parameters.
  - block is sequence of statements.

Thread Creation

An Example

-
  ```
  Thread.new("hello"){|mesg| print mesg;}
  ```
- This creates a new thread, which prints “hello”.

Query Thread Information

- `Thread.current`

  returns current executing thread object.
- `Thread.list`
  returns list of Thread objects.
- `Thread.critical`
  returns true/false depending upon Thread being in Critical region.

Thread Manipulation

1. `Thread.kill(aThread)`

   kills the aThread object.
2. `Thread.stop`
   puts the Thread in sleep state.
3. `Thread.exit`
   Terminates the thread

Thread Manipulation contd..

1. `Thread.pass`

   Invokes the thread scheduler to pass execution to another thread.

2. An example

```
a = Thread.new { print "a"; Thread.pass;
                 print "b"; Thread.pass;
                 print "c" }
b = Thread.new { print "x"; Thread.pass;
                 print "y"; Thread.pass;
                 print "z" }
a.join
b.joinThread.stop
```

3. produces axbycz

Thread Manipulation contd..

- `Thread.stop`
  return true/false whether thread is dead or sleeping.

Process Creation

Process creation syntax

-
  ```
  Process.fork[{block }]. }
  ```
- where
  - block is sequence of statements.
- returns processId of process.

Query Process Information

- `Process.pid`

  returns Process Id of Process.
- `Process.uid`

  returns user id of this process.
- `Process.getpriority( kind, anInteger )`

  ```
  An example
   Process.getpriority(Process::PRIO_USER, 0)
   gets the priority of current user process.
  ```

Process Manipulation

1. `Process.kill( aSignal, [ aPid ] )`
   sends a SIGNAL aSignal to Process with pid=aPid
2. `Process.wait`
   waits for child process to exit.
3. `Process.exit!(returnValue)`
   Terminates the Process immediately and returns returnValue to underlying system.

Process Manipulation contd..

1. `Process.uid=anInteger`

   Sets user id of this process.

2. `Process.setpriority( kind, anInteger, anIntPriority )`

   ```
   Examples
   Process.setpriority(Process::PRIO_USER, 0, 19)
   Process.setpriority(Process::PRIO_PROCESS, 0, 19)
      sets priority of process.
   ```