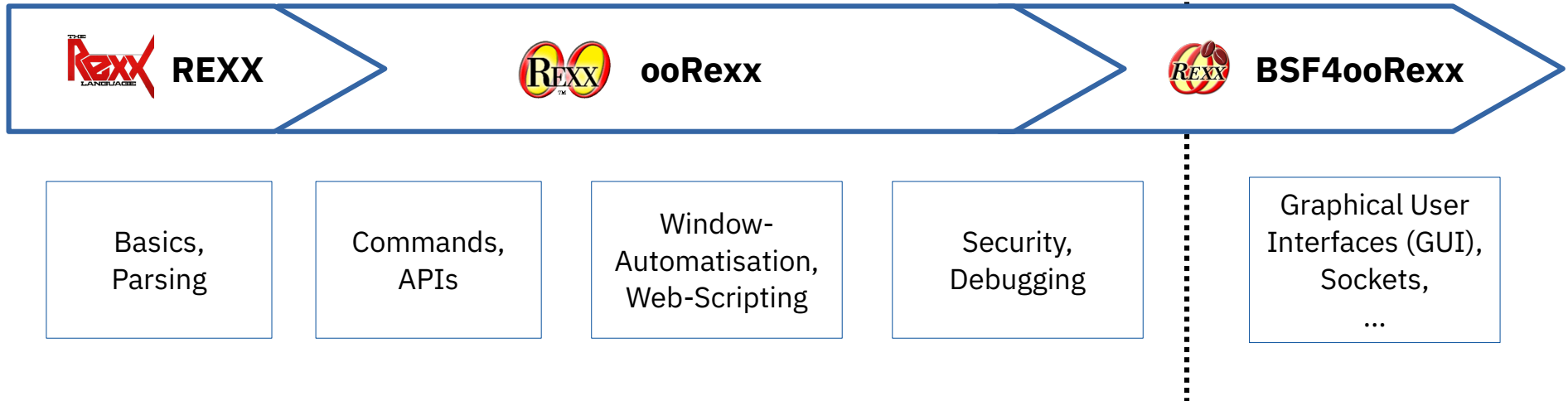


Multithreading in ooRexx

Business Programming 1

Business Programming 2



Overview



- Multithreading (MT)
- Howto kick off MT
- Guard lock
- Semaphores
- Roundup

- Multithreading
 - Parallel (concurrent) execution of parts of a program on different (multiple) "threads of execution" within a process
- ooRexx MT
 - Inter-object MT
 - Different objects can execute methods on different threads concurrently
 - Intra-object MT
 - In a specific object methods from different class (scopes) can execute on different threads concurrently

Howto Kick Off MT, 1



- Externally
 - E.g. a GUI dispatches messages concurrently from the GUI thread
- From within ooRexx programs
 - **REPLY** keyword statement in methods
 - Returns from the method and
 - Remainder of the method gets executed on a separate thread
 - **START** message of the root class **.Object**
 - **START** message to a message object (instance of **.Message**)
 - **.Alarm** class dispatching a message on a separate thread

Howto Kick Off MT, **REPLY**, 2



```
p=.PingPong~new
p~ping          -- will reply
do i=1 to 4
    say ti(.context) "pong #" i "(sleeping 0.10s)"
    call sysSleep 0.1      -- sleep 1/10 second
end

::class PingPong
::method ping
    reply          -- return, remainder on new thread
    do i=1 to 4
        sleepTime=random(1,10)/100  -- sleep between 1/100 and 1/10 seconds
        say ti(.context) "ping #" i "(sleeping" sleepTime"s)"
        call sysSleep sleepTime
    end

::routine ti
    use arg ctxt
    return ["T"ctxt~thread"] ["I"ctxt~invocation"]
```

Possible output:

```
[T1] [I2] pong # 1 (sleeping 0.10s)
[T2] [I1] ping # 1 (sleeping 0.05s)
[T2] [I1] ping # 2 (sleeping 0.09s)
[T1] [I2] pong # 2 (sleeping 0.10s)
[T2] [I1] ping # 3 (sleeping 0.07s)
[T2] [I1] ping # 4 (sleeping 0.09s)
[T1] [I2] pong # 3 (sleeping 0.10s)
[T1] [I2] pong # 4 (sleeping 0.10s)
```

Howto Kick Off MT, .Object's START, 3



```
p=.PingPong~new
p~start("ping")      -- Object's start method
do i=1 to 4
  say ti(.context) "pong #" i "(sleeping 0.10s)"
  call sysSleep 0.1 -- sleep 1/10 second
end

::class PingPong
::method ping      -- will run on separate thread
do i=1 to 4
  sleepTime=random(1,10)/100 -- sleep between 1/100 and 1/10 seconds
  say ti(.context) "ping #" i "(sleeping" sleepTime"s)"
  call sysSleep sleepTime
end

::routine ti
use arg ctxt
return ["T"ctxt~thread] ["I"ctxt~invocation"]
```

Possible output:

```
[T1] [I1] pong # 1 (sleeping 0.10s)
[T2] [I2] ping # 1 (sleeping 0.02s)
[T2] [I2] ping # 2 (sleeping 0.01s)
[T2] [I2] ping # 3 (sleeping 0.05s)
[T1] [I1] pong # 2 (sleeping 0.10s)
[T2] [I2] ping # 4 (sleeping 0.01s)
[T1] [I1] pong # 3 (sleeping 0.10s)
[T1] [I1] pong # 4 (sleeping 0.10s)
```

Howto Kick Off MT, .Message's START, 3



```
p=.PingPong~new
m=.message~new(p,"ping")
m~start           -- Message's start method
do i=1 to 4
  say ti(.context) "pong #" i "(sleeping 0.10s)"
  call sysSleep 0.1 -- sleep 1/10 second
end

::class PingPong
::method ping           -- will run on separate thread
do i=1 to 4
  sleepTime=random(1,10)/100 -- sleep between 1/100 and 1/10 seconds
  say ti(.context) "ping #" i "(sleeping" sleepTime"s)"
  call sysSleep sleepTime
end

::routine ti
use arg ctxt
return ["T"ctxt~thread"] ["I"ctxt~invocation"]
```

Possible output:

```
[T1] [I1] pong # 1 (sleeping 0.10s)
[T2] [I2] ping # 1 (sleeping 0.02s)
[T2] [I2] ping # 2 (sleeping 0.08s)
[T1] [I1] pong # 2 (sleeping 0.10s)
[T2] [I2] ping # 3 (sleeping 0.03s)
[T2] [I2] ping # 4 (sleeping 0.09s)
[T1] [I1] pong # 3 (sleeping 0.10s)
[T1] [I1] pong # 4 (sleeping 0.10s)
```

Howto Kick Off MT, **.Alarm**, 3



```
p=.PingPong~new
do i=1 to 4
  sleepTime=random(1,10)/100 -- sleep between 5/100 and 1/10 seconds
  .alarm~new(sleepTime,.message~new(p,'ping','i',i,"in an alarm" sleepTime))
  say ti(.context) "pong #" i "(sleeping 0.10s)"
  call sysSleep 0.1 -- sleep 1/10 second, no alarms can be created for 0.1s !
end

::class PingPong
::method ping -- alarm will run on a separate thread
  use arg i, sleepTime
  say ti(.context) "ping #" i "(sleeping" sleepTime"s)"

::routine ti
  use arg ctxt
  return ["T"ctxt~thread] ["I"ctxt~invocation"]
```

Possible output:

```
[T1] [I2] pong # 1 (sleeping 0.10s)
[T2] [I3] ping # 1 (sleeping in an alarm 0.02s)
[T1] [I2] pong # 2 (sleeping 0.10s)
[T2] [I5] ping # 2 (sleeping in an alarm 0.02s)
[T1] [I2] pong # 3 (sleeping 0.10s)
[T2] [I7] ping # 3 (sleeping in an alarm 0.04s)
[T1] [I2] pong # 4 (sleeping 0.10s)
[T2] [I9] ping # 4 (sleeping in an alarm 0.1s)
```


Guard Lock, 1



- By default all methods of a class are guarded
- Only one of the guarded method of a class can execute, after acquiring the *guard lock*, all other guarded methods of the class are blocked
 - Therefore all guarded methods of a class can only execute sequentially!
- A guarded method can invoke other methods which get the guard lock and increase the guard lock count by one
 - Upon return the guard lock count gets reduced by one
- If a guarded method returns and the guard lock count drops to 0, then one of the blocked methods will become executable and gain the guard lock

Guard Lock, 2



- A guarded method owning the guard lock can free the guard lock with the **GUARD OFF** keyword statement and turns into an *unguarded* method
- An unguarded method may wish to acquire the guard lock by issuing the **GUARD ON** keyword statement and turns into a guarded method that gets blocked until the guard lock gets acquired

Guard Lock, 3



```
p=.PingPong~new
p~start("ping")  -- dispatch message on a new thread
p~start("pong")  -- dispatch message on a new thread
do until p~done=2
  call sysssleep 0.01
  say "p~done="p~done
end

::class PingPong
::method init
  self~done=0      -- initialize attribute
::method ping      -- alarm will run on a separate thread
  reply
  do i=1 to 4
    say ti(.context) "ping #" i
  end
  self~done+=1

::method pong      -- alarm will run on a separate thread
  reply
  do i=1 to 4
    say ti(.context) "pong #" i
  end
  self~done+=1

::attribute done

::routine ti
  use arg ctxt
  return ["T"ctxt~thread"] ["I"ctxt~invocation"]
```

Possible output:

```
[T3] [I2] ping # 1
[T3] [I2] ping # 2
[T3] [I2] ping # 3
[T3] [I2] ping # 4
[T2] [I3] pong # 1
[T2] [I3] pong # 2
[T2] [I3] pong # 3
[T2] [I3] pong # 4
p~done=2
```

Guard Lock, 4



```
.traceObject~option="Standard"
```

```
p=.PingPong~new
```

```
p~start("ping")    -- dispatch message on a new thread
```

```
p~start("pong")    -- dispatch message on a new thread
```

```
do until p~done=2
```

```
    call sysssleep 0.01
```

```
    say "p~done="p~done
```

```
end
```

```
::class PingPong
```

```
::method init
```

```
    self~done=0      -- initialize attribute
```

```
::method ping      -- alarm will run on a separate thread
```

```
    reply
```

```
    do i=1 to 4
```

```
        trace all
```

```
        say ti(.context) "ping #" i
```

```
        trace normal
```

```
    end
```

```
    self~done+=1
```

```
::method pong      -- alarm will run on a separate thread
```

```
    reply
```

```
    do i=1 to 4
```

```
        trace all
```

```
        say ti(.context) "pong #" i
```

```
        trace normal
```

```
    end
```

```
    self~done+=1
```

```
::attribute done
```

```
::routine ti
```

```
    use arg ctxt
```

```
    return ["T"ctxt~thread"] ["I"ctxt~invocation"]
```

Possible output:

```
[T3 I2 G A1 L1 * ] 17 ** say ti(.context) "ping #" i
[T3] [I2] ping # 1
[T3 I2 G A1 L1 * ] 18 ** trace normal
[T3 I2 G A1 L1 * ] 17 ** say ti(.context) "ping #" i
[T3] [I2] ping # 2
[T3 I2 G A1 L1 * ] 18 ** trace normal
[T3 I2 G A1 L1 * ] 17 ** say ti(.context) "ping #" i
[T3] [I2] ping # 3
[T3 I2 G A1 L1 * ] 18 ** trace normal
[T3 I2 G A1 L1 * ] 17 ** say ti(.context) "ping #" i
[T3] [I2] ping # 4
[T3 I2 G A1 L1 * ] 18 ** trace normal
[T2 I3 G A1 L1 * ] 26 ** say ti(.context) "pong #" i
[T2] [I3] pong # 1
[T2 I3 G A1 L1 * ] 27 ** trace normal
[T2 I3 G A1 L1 * ] 26 ** say ti(.context) "pong #" i
[T2] [I3] pong # 2
[T2 I3 G A1 L1 * ] 27 ** trace normal
[T2 I3 G A1 L1 * ] 26 ** say ti(.context) "pong #" i
[T2] [I3] pong # 3
[T2 I3 G A1 L1 * ] 27 ** trace normal
[T2 I3 G A1 L1 * ] 26 ** say ti(.context) "pong #" i
[T2] [I3] pong # 4
[T2 I3 G A1 L1 * ] 27 ** trace normal
p~done=2
```

Guard Lock, 5



```
ep=.PingPong~new
p~start("ping")  -- dispatch message on a new thread
p~start("pong")  -- dispatch message on a new thread
do until p~done=2
  call sysssleep 0.01
  say "p~done="p~done
end

::class PingPong
::method init
  self~done=0      -- initialize attribute
::method ping      -- alarm will run on a separate thread
  reply
  do i=1 to 4
    guard on      -- acquire guard lock -> guarded
    say ti(.context) "ping #" i
    guard off     -- release guard lock -> unguarded
  end
  guard on
  self~done+=1

::method pong      -- alarm will run on a separate thread
  reply
  do i=1 to 4
    guard on      -- acquire guard lock -> guarded
    say ti(.context) "pong #" i
    guard off     -- release guard lock -> unguarded
  end
  guard on
  self~done+=1

::attribute done

::routine ti
  use arg ctxt
  return "[T"ctxt~thread"] [I"ctxt~invocation"]"
```

Possible output:

```
[T3] [I2] ping # 1
[T2] [I3] pong # 1
[T3] [I2] ping # 2
[T2] [I3] pong # 2
[T3] [I2] ping # 3
[T2] [I3] pong # 3
[T3] [I2] ping # 4
[T2] [I3] pong # 4
p~done=2
```

Guard Lock, 6



```
.traceObject-option="Standard"
p=.PingPong~new
p-start("ping") -- dispatch message on a new thread
p-start("pong") -- dispatch message on a new thread
do until p-done=2
  call sysleep 0.01
  say "p~done="p-done
end

::class PingPong
::method init
  self~done=0 -- initialize attribute
::method ping
  reply -- alarm will run on a separate thread
  do i=1 to 4
    trace all
    guard on -- acquire guard lock -> guarded
    say ti(.context) "ping #" i
    guard off -- release guard lock -> unguarded
    trace normal
  end
  trace all
  guard on
  self~done+=1

::method pong
  reply -- alarm will run on a separate thread
  do i=1 to 4
    trace all
    guard on -- acquire guard lock -> guarded
    say ti(.context) "pong #" i
    guard off -- release guard lock -> unguarded
    trace normal
  end
  trace all
  guard on
  self~done+=1

::attribute done

::routine ti
  use arg ctxt
  return ["T"ctxt~thread] ["I"ctxt~invocation]"
```

Possible output:

```
[T3 I2 G A1 L1 * ] 17 ** guard on -- acquire guard lock -> guarded
[T3 I2 G A1 L1 * ] 18 ** say ti(.context) "ping #" i
[T3] [I2] ping # 1
[T3 I2 G A1 L1 * ] 19 ** guard off -- release guard lock -> unguarded
[T3 I2 Gu A1 L1 ] 20 ** trace normal
[T2 I3 G A1 L1 * ] 30 ** guard on -- acquire guard lock -> guarded
[T3 I2 Gu A1 L1 ] 17 ** guard on -- acquire guard lock -> guarded
[T2 I3 G A1 L1 * ] 31 ** say ti(.context) "pong #" i
[T2] [I3] pong # 1
[T2 I3 G A1 L1 * ] 32 ** guard off -- release guard lock -> unguarded
[T2 I3 Gu A1 L1 ] 33 ** trace normal
[T3 I2 G A1 L1 * ] 18 ** say ti(.context) "ping #" i
[T3] [I2] ping # 2
[T2 I3 Gu A1 L1 ] 30 ** guard on -- acquire guard lock -> guarded
[T3 I2 G A1 L1 * ] 19 ** guard off -- release guard lock -> unguarded
[T3 I2 Gu A1 L1 ] 20 ** trace normal
[T2 I3 G A1 L1 * ] 31 ** say ti(.context) "pong #" i
[T2] [I3] pong # 2
[T3 I2 Gu A1 L1 ] 17 ** guard on -- acquire guard lock -> guarded
[T2 I3 G A1 L1 * ] 32 ** guard off -- release guard lock -> unguarded
[T2 I3 Gu A1 L1 ] 33 ** trace normal
[T3 I2 G A1 L1 * ] 18 ** say ti(.context) "ping #" i
[T3] [I2] ping # 3
[T2 I3 Gu A1 L1 ] 30 ** guard on -- acquire guard lock -> guarded
[T3 I2 G A1 L1 * ] 19 ** guard off -- release guard lock -> unguarded
[T3 I2 Gu A1 L1 ] 20 ** trace normal
[T2 I3 G A1 L1 * ] 31 ** say ti(.context) "pong #" i
[T2] [I3] pong # 3
[T3 I2 Gu A1 L1 ] 17 ** guard on -- acquire guard lock -> guarded
[T2 I3 G A1 L1 * ] 32 ** guard off -- release guard lock -> unguarded
[T2 I3 Gu A1 L1 ] 33 ** trace normal
p-done=0
[T3 I2 G A1 L1 * ] 18 ** say ti(.context) "ping #" i
[T3] [I2] ping # 4
[T2 I3 Gu A1 L1 ] 30 ** guard on -- acquire guard lock -> guarded
[T3 I2 G A1 L1 * ] 19 ** guard off -- release guard lock -> unguarded
[T3 I2 Gu A1 L1 ] 20 ** trace normal
[T2 I3 G A1 L1 * ] 31 ** say ti(.context) "pong #" i
[T2] [I3] pong # 4
[T3 I2 Gu A1 L1 ] 23 ** guard on
[T2 I3 G A1 L1 * ] 32 ** guard off -- release guard lock -> unguarded
[T2 I3 Gu A1 L1 ] 33 ** trace normal
p-done=0
[T3 I2 G A1 L1 * ] 24 ** self~done+=1
[T2 I3 Gu A1 L1 ] 36 ** guard on
[T2 I3 G A1 L1 * ] 37 ** self~done+=1
p-done=2
```

Semaphores, 1



- Semaphores can be used to synchronize multithreaded programs
- ooRexx comes with two built-in semaphore classes
 - **EventSemaphore**, cf. rexxref.pdf, "5.4.7. EventSemaphore Class"
 - **MutexSemaphore**, cf. rexxref.pdf, "5.4.12. MutexSemaphore Class"
- Examples from rexxref.pdf get discussed

Semaphores, 2

EventSemaphore Class



```
event = .EventSemaphore~new -- ooRexx 5.1. sample
say "main starts tasks"
do nr = 1 to 3
  .task~new~waitFor(event, "task" nr)
end
call SysSleep 0.1
say "main posts"
event~post
say "main ends"

::class Task
::method waitFor
  reply
  use strict arg event, name
  ti=ti(.context) -- get thread and invocation IDs
  say ti name "waits"
  event~wait
  say ti name "runs"

::routine ti
  use arg ctxt
  return "[T"ctxt~thread"] [I"ctxt~invocation"]"
```

Possible output:

```
main starts tasks
[T2] [I2] task 2 waits
[T3] [I1] task 1 waits
[T4] [I3] task 3 waits
main posts
main ends
[T3] [I1] task 1 runs
[T2] [I2] task 2 runs
[T4] [I3] task 3 runs
```


Semaphores, 3

MutexSemaphore Class



```
mutex = .MutexSemaphore~new
.Task~new~startWork(mutex, "work 1")
.Task~new~startWork(mutex, "work 2")
say ti(.context) "work tasks started"
```

```
::class Task
::method startWork unguarded
```

```
  expose mutex name
  use strict arg mutex, name
  reply
  self~doWork(1)
```

```
::method doWork unguarded
```

```
  expose mutex name
  use strict arg level
  -- three levels of nested acquires
  if level > 3 then return
```

```
  mutex~acquire
  say ti(.context) name level
  self~doWork(level + 1)
```

```
::routine ti
  use arg ctxt
  return ["T"ctxt~thread"] ["I"ctxt~invocation"]
```

Possible output:

```
[T1] [I3] work tasks started
[T2] [I4] work 1 1
[T2] [I5] work 1 2
[T2] [I6] work 1 3
[T3] [I7] work 2 1
[T3] [I8] work 2 2
[T3] [I9] work 2 3
```

- Easy to create multi-threaded programs in ooRexx
 - **REPLY** keyword statement
 - **START** method in **.Object**
 - **START** method in **.Message**
 - **.Alarm** class
- **Guard locks** to guard execution of guarded methods in a class
 - Unguarded methods can run concurrently to guarded methods
- Semaphores to synchronize multithreaded parts
 - **EventSemaphore**, **MutexSemaphore**
- For debugging with **TRACE** set **.TraceObject**'s **option** class attribute to **"Standard"** to gain insight in multithreaded execution