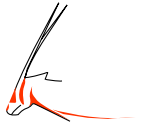
Department of Information Systems
and Operations Management

Multithreaded Programming in ooRexx

Understanding the ooRexx MT Concepts

The 2024 International Rexx Symposium
Brisbane, Queensland, Australia
March 3rd – March 6th 2024



Overview

- Multithreading (MT) concepts in ooRexx
- MT related keyword statements in method routines
  - REPLY keyword statement
  - GUARD keyword statement
- .Message class
  - Allows for dispatching messages synchronously or asynchronously
  - Message objects can be used in the .Alarm class to dispatch messages later
- Roundup

Multithreading Concepts, 1

- Can be triggered using message objects and within method routines
- By default all method routines are guarded
  - A guarded method can only execute, if it has the object's scope lock
  - All methods of a class are in the same “scope”
    - Only one of the guarded methods in the same scope can execute, all other methods are blocked
    - A running guarded method can invoke other guarded methods in the same scope
- It is possible to define a method as UNGUARDED
  - Unguarded methods can always run concurrently
  - Unguarded methods are not controlled (guarded) by the object's scope lock
  - Watch out: unguarded methods can concurrently change attribute values!
    - Synchronize access to attributes, e.g. with an .EventSemaphore or a .MutexSemaphore

Multithreading Concepts, 2

- ooRexx is a powerful interpreter that
  - Allows mutiple Rexx interpreter instances to run concurrently in the same process
    - Each Rexx interpreter instance has a distinct .local environment and shares the global .environment directory
  - Each ooRexx program can take advantage of multithreading where each concurrently executing *activity* gets run on a proper operating system *thread*
  - Maintains an object scope lock for all methods of the same class (“scope”)
    - The object's scope lock is used to guard the execution of guarded methods in the same scope
    - Guarded methods in superclasses are guarded separately according to their scope
    - By defaul,t guarded methods can execute in parallel if they stem from different scopes
      - *Intra* object concurrency
  - Allows safe concurrent execution of methods in different instances (objects)
    - *Inter* object concurrency

Multithreading Concepts, 3

- Object Rexx default behaviour (continued)
  - All methods are GUARDed by default (as a side effect access to attributes gets serialized)
    - Within a class (“scope”) by default only one guarded method can be executed for one and the same object if it acquired the object's scope lock, all other guarded methods of that class (scope) get blocked
      - An object's scope lock is acquired when a guarded method gets invoked
      - An object's scope lock gets released when a guarded method ends execution
    - Methods of one and the same object defined in different superclasses (scopes), are able to run concurrently (intra-multithreading)
- The keyword UNGUARD of a method directive allows that method to run concurrently with any other method in that class for one and the same object
  - There is no exclusive access protection of the object and its attributes!

Cf. rexxref.pdf (12.4. Using Additional Concurrency Mechanisms)

Multithreading Concepts, 4

- Object Rexx default behaviour (continued)
  - It is possible to kick off multithreading at runtime from within methods
    - REPLY keyword statement (only available within a method)
      - Same effect as the RETURN statement
        - Calling program receives execution control (continues to run), **but**
        - **In addition** the remaining statements of the method continue to run as a new activity concurrently on a new thread!
      - Optionally the REPLY statement may return a value to the calling program
      - After the REPLY keyword statement an EXIT or a RETURN keyword statement must not supply a return value
      - Note: the object's scope lock of a guarded method will get released upon executing the REPLY keyword statement and will get reacquired on the new thread for executing the remaining statements

Cf. rexxref.pdf (2.24. REPLY, 2.25. RETURN)

Multithreading Concepts, 5

- It is possible to determine at runtime whether methods are allowed to be executed concurrently with other methods of the same class (scope) for one and the same object
  - GUARD
    - GUARD ON instruction
      - **Waits until it gets the object's scope lock** if another method holds the object's scope lock already, then execution is halted until the other method releases the object's scope lock
    - The GUARD OFF instruction releases the object's scope lock and makes the method unguarded
  - Efficient safeguarding of "critical segments"
    - Waiting for exclusive access can be made dependent on a given value appearing in an attribute of the object (GUARD ON WHEN ...)
    - Waiting for the object's scope lock being relesed can be made dependent on a given value appearing in an attribute of the object (GUARD OFF WHEN ...)

Cf. rexxref.pdf (12.4. Using Additional Concurrency Mechanisms)

REPLY Keyword Instruction, 1

- REPLY returns control to the caller and can have a return value
- Remaining method statements constitute a separate *activity* being executed on a separate *thread*
- Notes ad the following example
  - The execution is not necessarily sequential (synchronous) anymore
    - The main program may end before the concurrently executing activities end
  - As all the methods are guarded, only the one holding the object's scope lock can execute blocking all others
    - All the other guarded methods have to wait until the object's scope lock gets released such that one of the next guarded methods can acquire the object's scope lock and becomes eligible to run

REPLY Keyword Instruction, 2

```
a=.x~new
b=.x~new
c=.x~new
fifo=.fifo~new  -- a FIFO buffer
.local~repetitions = 50
a~testwrite(fifo, "from_a")
b~testwrite(fifo, "FROM_B")
c~testread(fifo)
say "after testread"
```

```
::class X

::method testwrite  -- guarded
  use arg fifo, msg1
  REPLY
  do i=1 to .repetitions
    fifo~write(msg1 i)
  End

::method testread   -- guarded
  use arg fifo
  REPLY
  do while fifo~items > 0
    i=fifo~read
    say i
  end
```

```
::class FIFO    -- first-in, first-out
::method init       -- guarded
  expose buffer
  buffer=.queue~new

::method write      -- guarded
  expose buffer
  use arg tmp
  buffer~queue(tmp)

::method read       -- guarded
  expose buffer
  return buffer~pull

::method items      -- guarded
  expose buffer
  return buffer~items
```

Output:

```
after testread
from_a 1
from_a 2
 …
from_a 50
FROM_B 1
...
FROM_B 50
```

REPLY and GUARD ON|OFF, 1

- REPLY returns control to the caller the remaining statements get executed on a new activity (thread)
- The FIFO class uses GUARD ON WHEN and GUARD OFF WHEN
  - Demonstrates how to use some lock attribute to control execution in critical sections of code
  - Attribute lock gets defined in constructor and is accessed from the method routines sheltering critical sections of code with the help of the GUARD keyword instruction
  - Notes
    - Changing the value of the attribute lock is done only when the object's scope lock could be obtained such that no concurrent change of the attribute is possible
    - This is a pedagogical example, code could be simpler

REPLY and GUARD ON|OFF, 2

```
a=.x~new
b=.x~new
c=.x~new
fifo=.fifo~new  -- a FIFO buffer
.local~repetitions = 50
a~testwrite(fifo, "from_a")
b~testwrite(fifo, "FROM_B")
c~testread(fifo)
say "after testread"
```

```
::class X

::method testwrite  -- guarded
  use arg fifo, msg1
  REPLY
  do i=1 to .repetitions
    fifo~write(msg1 i)
  End

::method testread   -- guarded
  use arg fifo
  REPLY
  do while fifo~items > 0
    i=fifo~read
    say i
  end
```

```
::class FIFO
::method init       -- guarded
  expose buffer lock
  buffer=.queue~new
  lock=.false

::method write UNGUARDED
  expose buffer lock
  GUARD ON WHEN lock=.false
  lock=.true
  GUARD OFF
  use arg tmp
  buffer~queue(tmp) -- queue item
  GUARD ON
  lock=.false

::method read  UNGUARDED
  expose buffer lock
  GUARD ON WHEN lock=.false
  lock=.true
  GUARD OFF
  data=buffer~pull  -- get item
  GUARD ON
  lock=.false
  return data

::method items      -- guarded
  expose buffer
  return buffer~items
```

Output:

```
after testread
FROM_B 1
from_a 1
FROM_B 2
...
FROM_B 50
from_a 19
...
from_a 50
```

Class MESSAGE, 1

- .Message class
  - Two possibilities to dispatch messages
    - SEND - synchronous execution
      - Execution proceeds, after the message was completely carried out
    - START - asynchronous execution (multithreading)
      - Message is dispatched and invokes the method as an activity on a sepearate thread
      - Execution of the calling program proceeds concurrently
  - Additional interesting methods in the Message class
    - COMPLETED – returns .true or .false, indicating whether the message has completed, i.e. the invoked method has completed
    - RESULT - waits for and returns the result of an (asynchronously) executing method
    - NOTIFY - allows sending a message to an object to notify it that the message has finished executing

Cf. rexxref.pdf (5.1.2. Message Class)

Class MESSAGE, 2

- **.Alarm** class expects a message object as its first argument
  - Allows for sending the message at a later time
  - Allows for notification callbacks
  - Dispatching the message can be cancelled (cf. CANCEL method)

Cf. rexxref.pdf (5.4.1. Alarm Class)

Using Class MESSAGE, no REPLY!

```
aa=.x~new
b=.x~new
c=.x~new
fifo=.fifo~new  -- a FIFO buffer
.local~repetitions = 50
.message~new(a, "testwrite", "I", fifo, "from_a")~start
.message~new(b, "testwrite", "I", fifo, "FROM_B")~start
.message~new(c, "testread",  "I", fifo)          ~start
say "after testread"
```

Output:

```
after testread
from_a 1
from_a 2
 …
from_a 50
FROM_B 1
...
FROM_B 50
```

```
::class X
::method testwrite  -- guarded
  use arg fifo, msg1
  do i=1 to .repetitions
    fifo~write(msg1 i)
  end

::method testread   -- guarded
  use arg fifo
  do while fifo~items > 0
    i=fifo~read
    say i
  end
```

```
::class FIFO
::method init       -- guarded
  expose buffer
  buffer=.queue~new

::method write      -- guarded
  expose buffer
  use arg tmp
  buffer~queue(tmp)

::method read       -- guarded
  expose buffer
  return buffer~pull

::method items      -- guarded
  expose buffer
  return buffer~items
```

Using OBJECT's START-method, no REPLY!

```
aa=.x~new
b=.x~new
c=.x~new
fifo=.fifo~new  -- a FIFO buffer
.local~repetitions = 50
a~start("testwrite", fifo, "from_a")
b~start("testwrite", fifo, "FROM_B")
c~start("testread",  fifo)
say "after testread"
```

Output:

```
after testread
from_a 1
from_a 2
 …
from_a 50
FROM_B 1
...
FROM_B 50
```

```
::class X
::method testwrite  -- guarded
  use arg fifo, msg1
  do i=1 to .repetitions
    fifo~write(msg1 i)
  end

::method testread   -- guarded
  use arg fifo
  do while fifo~items > 0
    i=fifo~read
    say i
  end
```

```
::class FIFO
::method init       -- guarded
  expose buffer
  buffer=.queue~new

::method write      -- guarded
  expose buffer
  use arg tmp
  buffer~queue(tmp)

::method read       -- guarded
  expose buffer
  return buffer~pull

::method items      -- guarded
  expose buffer
  return buffer~items
```

Synchronizing Activities

- Executing activities (threads) concurrently
  - How to determine whether all concurrently executing activites (threads) have stopped?
- Example class **Waiter**
  - Simple class whose only *instance* method "wait" is to run in the background for a random length of time
  - Number of running activites (threads) is counted with a class attribute
  - *Class* method "wait" blocks until counter drops to 0 and returns then to the caller/invoker
  - Original idea and code: cf. Ian Collier, news:comp.lang.rexx, 2004-11-09

Class WAITER, Waiting on Threads ...

```
w=.waiter~new -- create an instance
do i=1 to 5
   w~wait(i)  -- invoke instance method
end
say "Waiting for counter to drop to 0..."
.waiter~wait  -- invoke class method
say "--- All done ---"

/* Waiter */
::class waiter
::method init class  -- guarded class method
   expose counter
   counter=0         -- set initial value
::method up   class  -- guarded class method
   expose counter
   counter=counter+1 -- increase counter
::method down class  -- guarded class method
   expose counter
   counter=counter-1 -- decrease counter
::method wait class  -- guarded class method
   expose counter
   guard on when counter=0  -- wait until counter drops to 0

::method wait unguarded -- instance method
   a=random(1,6)  -- get a number between 1 and 6
   reply          -- now concurrency starts
   parse arg n    -- get invocation number
   .waiter~up     -- increase counter
   if n<>'' then say 'Waiter' n 'waiting' a 'seconds'
   call syssleep a   -- sleep a few seconds
   if n<>'' then say 'Waiter' n 'finished'
   .waiter~down   -- decrease counter
```

Possible Output:

```
Waiting for counter to drop to 0...
All done
Waiter 5 waiting 4 seconds
Waiter 1 waiting 2 seconds
Waiter 3 waiting 5 seconds
Waiter 4 waiting 4 seconds
Waiter 2 waiting 1 seconds
Waiter 2 finished
Waiter 1 finished
Waiter 5 finished
Waiter 4 finished
Waiter 3 finished
```

Roundup

- ooRexx makes it easy to create multithreaded programs
  - Keyword statements REPLY and GUARD in method routines
  - .Message class to dispatch messages asynchronously with START
    - Message objects can be used for the .Alarm class to dispatch message later
  - ooRexx root class .Object offers a START method to simplify multithreading
- Have fun exploring multithreading with ooRexx!