|  |
| --- |
| Edinburgh Napier University |
| Fundamentals of Parallel Systems LabBook |
| SET09109 |

|  |
| --- |
| GROUP F 40183743 |

**Ex 2-1**



**Consumer.groovy**

// insert a modified println statement

*println* "the result was: ${i}"

**Multiplier.groovy**

// write i \* factor to outChannel

outChannel.write(i \* factor)

// read in the next value of i

i = inChannel.read()

**RunMultiplier.groovy**

//insert here an instance of multiplier with a multiplication factor of 4

**new** Multiplier (inChannel: connect1.in(),

outChannel: connect2.out(),

factor: 4)

**Output**

next: 1

next: the result was: 4

2

next: the result was: 8

4

next: the result was: 16

3

next: the result was: 12

0

Finished

**Ex 2-2C:\Users\Beej\AppData\Local\Microsoft\Windows\INetCacheContent.Word\ex2.2_diagram.png**

**GenerateSetsOfThree.groovy**

//write the terminating List as per exercise definition

outChannel.write([-1,-1,-1])

**ListToStream.groovy**

// hint: output list elements as single integers

**for** ( i **in** 0 ..< inList.size)outChannel.write(inList[i])

inList = inChannel.read()

**CreateSetsOfEight.groovy**

// put v into outList and read next input

outList[i] = v

v = inChannel.read()

**Output**

Eight Object is [1, 2, 3, 4, 5, 6, 7, 8]

Eight Object is [9, 10, 11, 12, 13, 14, 15, 16]

Eight Object is [17, 18, 19, 20, 21, 22, 23, 24]

Finished

**Questions**

*What change is required to output objects containing six integers?*

**for** ( i **in** 0 .. 7 ) becomes **for** ( i **in** 0 .. 5 ) in CreateSetsOfEight.groovy

*How could you parameterise this in the system to output objects that contain any number of integers (e.g. 2, 4, 8, 12) ?*

Have the number the for loop be a variable that is read in from the console and can be decided by the user by writing it in the console.

*What happens if the number of integers required in the output stream is not a factor of the total number of integers in the input stream (e.g. 5 or 7)*

Numbers are left out and the process does not terminate because ListToStream.groovy can’t finish its for loop, and doesn’t send ‘-1’ to CreateSetsofEight.groovy

**Ex 3-1**

**Process Network Diagram for Differentiate using Minus**



**Minus.groovy**

// output one value subtracted from the other

// be certain you know which way round you are doing the subtraction!!

outChannel.write(read0.value - read1.value)

**Differentiate.groovy**

// insert a constructor for Minus

**new** Minus ( inChannel0: a.in(),

inChannel1: c.in(),

outChannel: outChannel)

**Output**

Differentiated Numbers

0

1

2

3

4

5

6

7

8

9

10

11

12

13

**Process Network Diagram for Differentiate using Negator**



**Negator.groovy**

//output the negative of the input value

outChannel.write(-(inChannel.read()))

**Differentiate.groovy**

//insert a constructor for Negator

**new** Negator ( inChannel: c.in(),

outChannel: d.out() )

**Output**

Differentiated Numbers

0

1

2

3

4

5

6

7

8

9

10

11

**Questions**

*Which is the more pleasing solution? Why?*

I prefer the Minus solution. It seems to be simpler, especially when considering the network process diagram. It directly undoes the integrate step as Minus.groovy is the opposite to GPlus.groovy. The negator adds another process before the GPlus that I feel overcomplicates it.

**Ex 3-2**



**GSCopy.groovy**

// output the input value in sequence to each output channel

outChannel0.write(i)

outChannel1.write(i)

**GSquares.groovy**

// you will need to modify this twice

**new** GSPairsA ( inChannel: I2P.in(),

outChannel: outChannel ),

*and*

// you will need to modify this twice

**new** GSPairsB ( inChannel: I2P.in(),

outChannel: outChannel ),

**Output (with GSPairsA)**

Squares

**Output (with GSPairsB)**

Squares

1

4

9

16

25

36

49

64

81

100

121

144

169

196

**Questions**

*Determine the effect of the change. Why does this happen?*

When using GSPairsA the process halts. This is because GSCopy sequentially writes the value to a.out() then b.out() so GPlus receives the value via a.in() but doesn’t get supplied a value from c.in() yet (as GTail does not write the first value it receives to c.out()). The 2nd value from the inChannel then cannot get sent by GSCopy via the a channel to GPlus as GPlus has not run. Therefore GSCopy is unable to send a 2nd value to GTail and the process halts.

When using GSPairsB, the value is sent to GTail first so GPlus will receive the 1st value via the a channel and then the 2nd value via the c channel from the GTail and run successfully. Therefore the process executes normally, providing the correct output.

**Ex 3-3**

**Questions**

*Why was it considered easier to build* ***GParPrint*** *as a new process rather than using multiple instances of* ***GPrint*** *to output the table of results?*

Building a table using multiple instances of GPrint would require complex routing of the inChannels and complicated formatting in the println statements so that an accurate table could be formed. GParPrint, however, reduces this complexity by taking multiple inChannels at once and printing each piece of information in line such that a table can easily be formed.

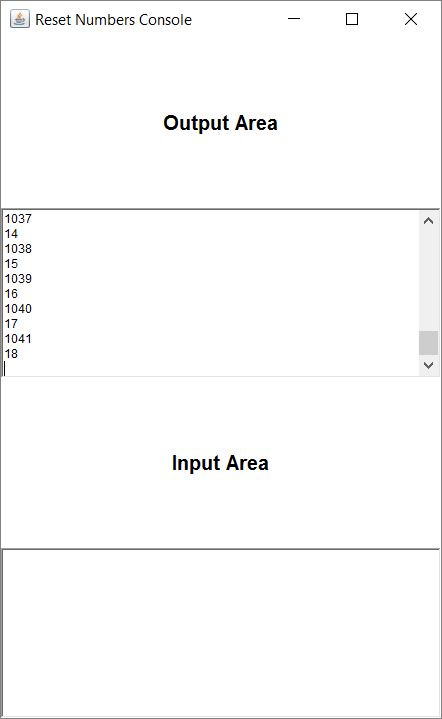
**Ex 4-1**

**Questions**

*What happens if line {25} of* ***ResetPrefix*** *Listing 4-1 is commented out? Why?*

As shown in the output image below when a reset value is inputted the output numbers oscillate between the initial sequence of numbers and the sequence starting from the reset value. If a second reset value is inputted, then the process deadlocks. This is because the old values are still in the system (as the line where they’re read in is commented out), when a third number is placed into the system there is no available process to accept the new number. By adding buffers more numbers could be placed into the system but you would need infinite buffers to allow the process to never deadlock, but this is impossible.

**Output**



**Ex 4-2**



**ResetNumbers.groovy**

// requires a constructor for ResetSuccessor

**new** ResetSuccessor( inChannel: b.in(),

outChannel: c.out(),

resetChannel: resetChannel )

**ResetSuccessor.groovy**

// deal with inputs from resetChannel and inChannel

// use a priSelect

**def** index = alt.priSelect()

**if** (index == 0 ) { // resetChannel input

**def** resetValue = resetChannel.read()

inChannel.read()

outChannel.write(resetValue)

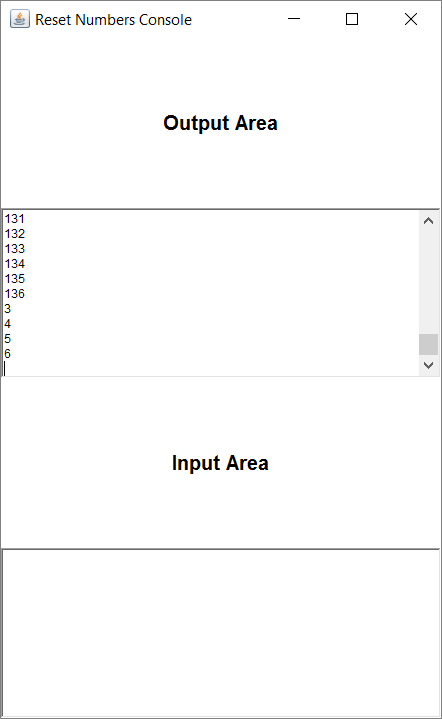
}

**else** { //inChannel input

outChannel.write(inChannel.read() + 1)

}

**Output**



**Questions**

*Does it overcome the problem identified in Exercise 1? If not, why not?*

This different formulation of ResetNumbers does not fix the problem identified in Exercise 1 for the same reasons as before.

**Ex 5-1**

**RunScaler.groovy**

**new** QProducer ( put: QP2Q.out(),

iterations: 50,

delay: 100 ),

**new** QConsumer ( get: QC2Q.out(),

receive: Q2QC.in(),

delay: 100 )

**Output**

QConsumer has started

QProducer has started

QConsumer has read 1

QConsumer has read 2

QConsumer has read 3

QConsumer has read 4

QConsumer has read 5

QConsumer has read 6

.

.

.

QConsumer has read 47

QConsumer has read 48

QConsumer has read 49

QConsumer has read 50

Q finished

QConsumer has read null

**Questions**

*By varying the delay times demonstrate that the system works in the manner expected. What do you conclude from these experiments?*

Changing the delay times did not affect operation from completing correctly, only how long it took to finish. Increasing the delay values for either QProducer, QConsumer or for both increased the time for each line of output to print.

**Ex 5-2**



**Scale.groovy**

**while** (**true**) {

**switch** ( scaleAlt.priSelect(preCon) ) {

**case** SUSPEND :

// deal with suspend input

suspend.read()

factor.write(scaling)

suspended = **true**

println "Suspended"

preCon[INJECT] = **true**

preCon[SUSPEND] = **false**

**break**

**case** INJECT:

// deal with inject input

scaling = injector.read()

println "Injected scaling is $scaling"

suspended = **false**

timeout = timer.read() + DOUBLE\_INTERVAL

timer.setAlarm(timeout)

preCon[SUSPEND] = **true**

preCon[INJECT] = **false**

**break**

**case** TIMER:

// deal with Timer input

timeout = timer.read() + DOUBLE\_INTERVAL

timer.setAlarm ( timeout )

scaling = scaling \* multiplier

println "Normal Timer: new scaling is ${scaling}"

**break**

**case** INPUT:

// deal with Input channel

**def** inValue = inChannel.read()

**def** result = **new** ScaledData()

result.original = inValue

**if** (preCon[SUSPEND] == **true**) {

result.scaled = inValue \* scaling

}

**else** {

result.scaled = inValue

}

outChannel.write ( result )

**break**

} //end-switch

} //end-while

**Output**

Original Scaled

0 0

1 2

2 4

3 6

Normal Timer: new scaling is 4

4 16

5 20

Suspended

6 6

Injected scaling is 5

7 35

8 40

9 45

10 50

11 55

Normal Timer: new scaling is 10

12 120

Suspended

13 13

Injected scaling is 11

14 154

15 165

**Questions**

*Which is the more elegant formulation? Why?*

The scaling device that uses pre-conditions is more elegant than the solution using nested alternatives, this is because nested alternatives has multiple switch statements making it hard to follow, whereas there is only one switch when using pre-conditions and therefore improves readability.

**Ex 6-1**

C:\Users\Beej\AppData\Local\Microsoft\Windows\INetCacheContent.Word\ex6.1_diagram (1).png

**CreatSetsOfEightForTest.groovy**

**def** ChannelInput inChannel

**def** outList = []

**void** run(){

**def** v = inChannel.read()

**while** (v != -1){

**for** ( i **in** 0 .. 7 ) {

// put v into outList and read next input

outList[i] = v

v = inChannel.read()

}

println "Eight Object is ${outList}"

}

println "Finished"

}

**RunThreeToEightTest.groovy**

**class** RunThreeToEightTest **extends** GroovyTestCase {

**void** testRunThreeToEight() {

**def** G2S = Channel.*one2one*()

**def** S2C = Channel.*one2one*()

**def** GSo3 = **new** GenerateSetsOfThree ( outChannel: G2S.out() )

**def** L2S = **new** ListToStream ( inChannel: G2S.in(),

outChannel:S2C.out() )

**def** CSo8 = **new** CreateSetsOfEightForTest ( inChannel: S2C.in())

**def** processList = [ GSo3, L2S, CSo8 ]

**new** PAR ( processList ).run()

**def** expected = [17, 18, 19, 20, 21, 22, 23, 24]

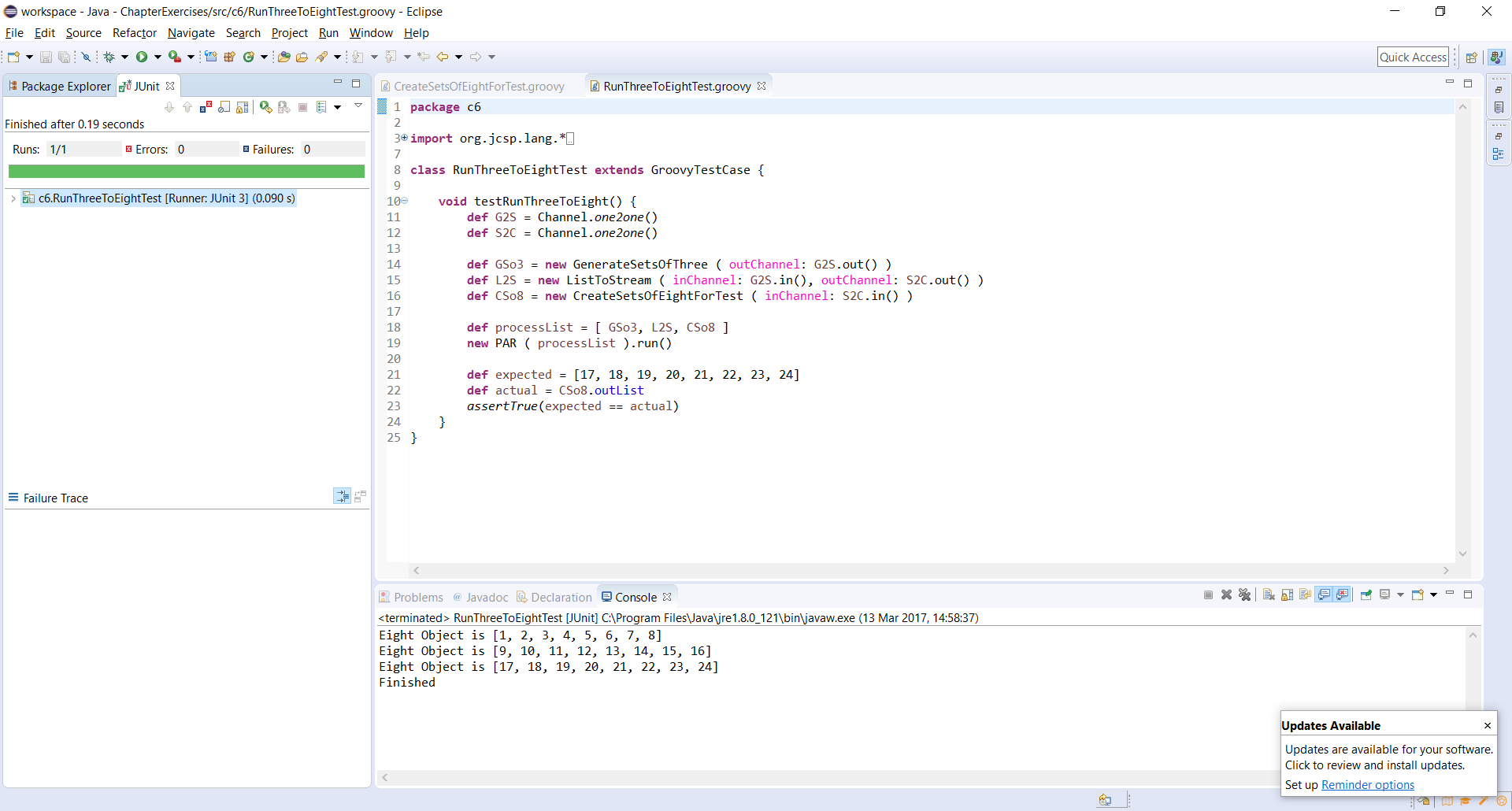
**def** actual = CSo8.outList

*assertTrue*(expected == actual)

}

}

**Output**



**Ex 7-1**

**Server.groovy**

**def** serverID = ""

**switch** (index) {

**case** CLIENT :

**def** key = clientRequest.read()

println "client$serverID requested value for key $key from server$serverID"

**def** value = dataMap[key]

**if** ( dataMap.containsKey(key) ) {

clientSend.write(dataMap[key])

println "server$serverID sent $value to client$serverID"

}

**else** {

println "server$serverID requested value for key $key from other server"

thisServerRequest.write(key)

} //end if

**break**

**case** OTHER\_REQUEST :

**def** key = otherServerRequest.read()

**def** value = dataMap[key]

**if** ( dataMap.containsKey(key) ) {

otherServerSend.write(dataMap[key])

println "server$serverID sent $value back to other server"

}

**else** {

otherServerSend.write(-1)

println "server$serverID did not have $key "

} //end if

**break**

**case** THIS\_RECEIVE :

**def** value = thisServerReceive.read()

clientSend.write(value)

println "server$serverID sent $value received from other server to client$serverID"

**break**

} // end switch

**Output**

Client 1 has 10 values in [11, 12, 13, 14, 15, 6, 17, 8, 19, 20]

Client 0 has 10 values in [1, 12, 3, 14, 15, 16, 7, 18, 9, 10]

client1 requested value for key 11 from server1

client0 requested value for key 1 from server0

server0 sent 10 to client0

server1 sent 110 to client1

client0 requested value for key 12 from server0

client1 requested value for key 12 from server1

server0 requested value for key 12 from other server

server1 sent 120 to client1

server1 sent 120 back to other server

server0 sent 120 received from other server to client0

client1 requested value for key 13 from server1

client0 requested value for key 3 from server0

server1 sent 130 to client1

server0 sent 30 to client0

client1 requested value for key 14 from server1

client0 requested value for key 14 from server0

server1 sent 140 to client1

server0 requested value for key 14 from other server

client1 requested value for key 15 from server1

server1 sent 150 to client1

server1 sent 140 back to other server

server0 sent 140 received from other server to client0

client1 requested value for key 6 from server1

client0 requested value for key 15 from server0

server1 requested value for key 6 from other server

server0 requested value for key 15 from other server

*(deadlock)*



**Questions**

*By placing print statements in the coding for the Server and Client processes see if you can determine the precise nature of the deadlock in the Client Server system.*

Using the output from the print statements and by looking at the diagram above it is clear that the deadlock occurs when both the servers request information from each other at the same time. Both the servers received a request from their clients for a value from a key that is outside their own server map. This causes each server to try to write to the other, but as they can’t read in anything new until they both successfully write then neither server will be able to receive the desired value and deadlock occurs.

**Ex 8-1**

**Client.groovy**

**void** run () {

**def** iterations = selectList.size

println "Client $clientNumber has $iterations values in $selectList"

**def** passed = **true**

**def** failedClient

**def** timesFailed = 0

**for** ( i **in** 0 ..< iterations) {

**def** key = selectList[i]

**def** desiredValue = 10 \* key

requestChannel.write(key)

println "Client $clientNumber requested value stored at $key."

**def** v = receiveChannel.read()

**if** (v != desiredValue) {

println "Client $clientNumber has received the wrong

value. $v was returned instead of $desiredValue."

failedClient = clientNumber

timesFailed = timesFailed + 1

passed = **false**

} **else** {

println "Client $clientNumber received the desired

value, $desiredValue."

}

}

**if** (passed) {

println "\nAll values successfully returned to Client

$clientNumber."

} **else** {

println "\nClient $failedClient was not successful. It failed

$timesFailed time(s)."

}

println "\n \*\*\* Client $clientNumber has finished \*\*\* \n"

}

**Output** (using wrong values in server map to test the system)

Number of clients per server; 1 to 9 ? 1

Client 10 has 10 values in [11, 12, 13, 14, 15, 6, 17, 8, 19, 20]

Client 0 has 10 values in [1, 12, 3, 14, 15, 16, 7, 18, 9, 10]

Client 0 requested value stored at 1.

Client 10 requested value stored at 11.

Client 10 received the desired value, 110.

Client 0 received the desired value, 10.

Client 10 requested value stored at 12.

Client 0 requested value stored at 12.

Client 0 received the desired value, 120.

Client 0 requested value stored at 3.

Client 10 received the desired value, 120.

Client 0 has received the wrong value. 31 was returned instead of 30.

Client 10 requested value stored at 13.

Client 10 received the desired value, 130.

Client 10 requested value stored at 14.

Client 10 received the desired value, 140.

Client 10 requested value stored at 15.

Client 0 requested value stored at 14.

Client 10 received the desired value, 150.

Client 0 received the desired value, 140.

Client 10 requested value stored at 6.

Client 0 requested value stored at 15.

Client 10 received the desired value, 60.

Client 0 received the desired value, 150.

Client 10 requested value stored at 17.

Client 0 requested value stored at 16.

Client 10 received the desired value, 170.

Client 0 has received the wrong value. 162 was returned instead of 160.

Client 10 requested value stored at 8.

Client 0 requested value stored at 7.

Client 10 received the desired value, 80.

Client 0 received the desired value, 70.

Client 10 requested value stored at 19.

Client 0 requested value stored at 18.

Client 10 received the desired value, 190.

Client 0 received the desired value, 180.

Client 0 requested value stored at 9.

Client 10 requested value stored at 20.

Client 10 received the desired value, 200.

Client 0 received the desired value, 90.

All values successfully returned to Client 10.

\*\*\* Client 10 has finished \*\*\*

Client 0 requested value stored at 10.

Client 0 received the desired value, 100.

Client 0 was not successful. It failed 2 time(s).

\*\*\* Client 0 has finished \*\*\*

**Ex 9-1**



**MissedEventsCheck.groovy**

**class** MissedEventsCheck **implements** CSProcess {

**def** ChannelInput inChannel

**def** ChannelOutput outChannel

**void** run(){

**def** previousData = 0

**def** count = 0

**def** e = **new** EventData()

**while**(**true**) {

e = inChannel.read().copy()

**if** (count > 0) {

**if** (e.data == previousData + 1 + e.missed) {

println "Number of missed data is

correct."

} **else** {

println "Number of missed data is

incorrect."

}

}

previousData = e.data

count = count + 1

outChannel.write(e)

}

}

}

**RunSingleStream.groovy**

**new** MissedEventsCheck ( inChannel: udd2mec.in(),

outChannel: mec2prn.out()),

**Output**

Event Output

Event Generator for source 1 has started

EventData -> [source: 1, data: 100, missed: 0]

Number of missed data is correct.

EventData -> [source: 1, data: 101, missed: 0]

Number of missed data is correct.

EventData -> [source: 1, data: 110, missed: 8]

Number of missed data is correct.

EventData -> [source: 1, data: 121, missed: 10]

Number of missed data is correct.

EventData -> [source: 1, data: 131, missed: 9]

Number of missed data is correct.

EventData -> [source: 1, data: 140, missed: 8]

Number of missed data is correct.

EventData -> [source: 1, data: 150, missed: 9]

Number of missed data is correct.

EventData -> [source: 1, data: 158, missed: 7]

Number of missed data is correct.

EventData -> [source: 1, data: 167, missed: 8]

Source 1 has finished

Number of missed data is correct.

EventData -> [source: 1, data: 177, missed: 9]

Number of missed data is correct.

EventData -> [source: 1, data: 189, missed: 11]

Number of missed data is correct.

EventData -> [source: 1, data: 198, missed: 8]

Number of missed data is correct.

EventData -> [source: 1, data: 199, missed: 0]

**Ex 9-2**

(Examples of the modifications made to the times associated with event generation and processing)

**RunMultiStream.groovy**

//High delay in event generation

minTimes = [ 1000, 1020, 1030, 1040, 1050, 1010, 1020, 1030, 1040 ]

maxTimes = [ 1100, 1150, 1200, 1050, 1060, 1030, 1060, 1100, 1080 ]

**def** eventProcess = **new** EventProcessing (eventStreams: eventsList,

//Default delay in processing

minTime: 10,

maxTime: 400 )

//Low delay in event generation

minTimes = [ 10, 20, 30, 40, 50, 10, 20, 30, 40 ]

maxTimes = [ 20, 30, 40, 50, 60, 70, 80, 90, 100 ]

**def** eventProcess = **new** EventProcessing ( eventStreams: eventsList,

//High delay in processing

minTime: 100,

maxTime: 1000 )

//Default delay in event generation

minTimes = [ 10, 20, 30, 40, 50, 10, 20, 30, 40 ]

maxTimes = [ 100, 150, 200, 50, 60, 30, 60, 100, 80 ]

**def** eventProcess = **new** EventProcessing ( eventStreams: eventsList,

//Low delay in processing

minTime: 10,

maxTime: 20 )

**Output** (example of output using high delay in event generation and default delay in processing)

Number of event sources between 1 and 9 ? 4

Event Generator for source 3 has started

Event Output

Event Generator for source 1 has started

Event Generator for source 2 has started

Event Generator for source 4 has started

…

…

EventData -> [source: 1, data: 195, missed: 0]

EventData -> [source: 4, data: 496, missed: 0]

EventData -> [source: 3, data: 390, missed: 0]

EventData -> [source: 2, data: 293, missed: 0]

Source 4 has finished

EventData -> [source: 1, data: 196, missed: 0]

EventData -> [source: 4, data: 497, missed: 0]

EventData -> [source: 3, data: 391, missed: 0]

Source 1 has finished

EventData -> [source: 2, data: 294, missed: 0]

EventData -> [source: 1, data: 197, missed: 0]

EventData -> [source: 4, data: 498, missed: 0]

EventData -> [source: 3, data: 392, missed: 0]

EventData -> [source: 2, data: 295, missed: 0]

EventData -> [source: 1, data: 198, missed: 0]

EventData -> [source: 3, data: 393, missed: 0]

EventData -> [source: 2, data: 296, missed: 0]

EventData -> [source: 3, data: 394, missed: 0]

Source 2 has finished

EventData -> [source: 2, data: 297, missed: 0]

EventData -> [source: 3, data: 395, missed: 0]

EventData -> [source: 2, data: 298, missed: 0]

EventData -> [source: 3, data: 396, missed: 0]

Source 3 has finished

EventData -> [source: 3, data: 397, missed: 0]

EventData -> [source: 3, data: 398, missed: 0]

**Output** (example of output using low delay in event generation and high delay in processing)

Number of event sources between 1 and 9 ? 4

Event Generator for source 1 has started

Event Generator for source 4 has started

Event Generator for source 2 has started

Event Output

EventData -> [source: 1, data: 100, missed: 0]

EventData -> [source: 2, data: 200, missed: 0]

EventData -> [source: 3, data: 300, missed: 0]

Source 1 has finished

EventData -> [source: 4, data: 400, missed: 0]

Source 2 has finished

EventData -> [source: 1, data: 101, missed: 0]

Source 3 has finished

EventData -> [source: 2, data: 201, missed: 0]

Source 4 has finished

EventData -> [source: 3, data: 309, missed: 8]

EventData -> [source: 4, data: 423, missed: 22]

EventData -> [source: 1, data: 187, missed: 85]

EventData -> [source: 2, data: 278, missed: 76]

EventData -> [source: 3, data: 381, missed: 71]

EventData -> [source: 4, data: 478, missed: 54]

EventData -> [source: 1, data: 198, missed: 10]

EventData -> [source: 2, data: 298, missed: 19]

EventData -> [source: 3, data: 398, missed: 16]

EventData -> [source: 4, data: 498, missed: 19]

**Questions**

*By modifying the times associated with each event generation stream and also of the processing system explore the performance of the system. What do you conclude?*

If the delay in the event generation is on average higher than the delay in the processing system, then less data is missed and vice versa. The extremes of this are when there is a high delay in the event generation and a low delay in the processing system causing there to be no missed data, and when there is a very low delay in the event generation and a high delay in the processing system causing a very significant portion of the data to be missed (often over 50%). This becomes clear when considering that if the events are being written into the processor faster than it can process them then the data will be missed as the processor is not ready to receive more data, but if the events are written in slowly then they can be processed in time.

**Ex 9-3**

**Output** (using PriMultiplex processing showing priority issue)

…

EventData -> [source: 1, data: 171, missed: 5]

EventData -> [source: 1, data: 172, missed: 0]

EventData -> [source: 1, data: 174, missed: 1]

EventData -> [source: 1, data: 176, missed: 1]

EventData -> [source: 1, data: 180, missed: 3]

Source 1 has finished

EventData -> [source: 1, data: 183, missed: 2]

EventData -> [source: 1, data: 189, missed: 5]

EventData -> [source: 1, data: 195, missed: 5]

EventData -> [source: 1, data: 198, missed: 2]

EventData -> [source: 2, data: 201, missed: 0]

EventData -> [source: 2, data: 261, missed: 59]

EventData -> [source: 2, data: 262, missed: 0]

EventData -> [source: 2, data: 263, missed: 0]

EventData -> [source: 2, data: 264, missed: 0]

…

**Questions**

*By choosing each of the options in turn, comment upon the effect that each multiplexer variation has on overall system performance.*

FairMultiplex: This is the multiplexer used in the previous exercise so it’s performance shall be taken as the base. It selects which event source to process ‘fairly’. When choosing which event source to process, the last one that was selected will have the lowest priority in the current selection.

PriMultiplex: This multiplexer selects whichever source event has the lowest index. Whilst it may reduce the total number of missed events across the all the source event streams, sources with a higher index will have a lot of its events missed until it becomes the source with the lowest index available to the multiplexer. This is shown in the output above where ‘source: 2’ had 59 events missed by the processor whilst it finished processing ‘source: 1’.

Multiplexer: This multiplexer is the same as FairMultiplex. When opening the declaration it can be seen that it simply calls the same select method that FairMulplexer uses (‘fairSelect()’), and is therefore identical. As a result, this multiplexer will have no difference when compared to the FairMultiplex in terms of overall system performance.