**Lab Book**

**Exercise 2.1**

**Diagram:**

Connect2

Connect1

Multiplier

Consumer

Producer

**Code:**

**def** processList =

[

**new** Producer ( outChannel: connect1.out() ),

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

**new** Multiplier(inChannel: connect1.in(), factor: 4, outChannel: connect2.out()),

**new** Consumer ( inChannel: connect2.in() )

]

**while** (i > 0)

{

// write i \* factor to outChannel

outChannel.write(i\*factor);

// read in the next value of i

i = inChannel.read();

}

**while** ( i > 0 )

{

//insert a modified println statement

println i;

i = inChannel.read()

}

**Output:**

3

next: 12

5

next: 20

6

next: 24

**Exercise 2.2**

**Diagram:**

Connect2

Connect1

CreateSetsOfEight

ListToStream

GenerateSetsOfThree

**Code:**

**GenerateSetsOfThree**

//write the terminating List as per exercise definition

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

**ListToStream**

// hint: output list elements as single integers

**for**(j **in** 0..<inList.size)

{

outChannel.write(inList[j])

}

inList = inChannel.read()

**CreateSetsOfEight**

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

{

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

{

// put v into outList and read next input

outList.add(v)

v = inChannel.read()

}

*println* " Eight Object is ${outList}"

outList.clear()

}

**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?

Change the

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

in CreateSetsOfEight to

**for** ( i **in** 0 .. 5 )

How could you paramaterise this in the system to output objects that contain any number of integers?

Instead of changing it to a static value, create a variable setSize and change the loop to:

**for** ( i **in** 0 .. (setSize – 1) )

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?

The numbers that go beyond the maximum factor are excluded from the output stream.

**Exercise 3.1**

**Minus:**

**Diagram:**

out

in

a

c

Minus

GPrefix

b

GPCopy

**Code:**

**Minus.groovy**

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

**Differentiate.groovy**

**def** differentiateList = [ **new** GPrefix(prefixValue: 0,

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

outChannel: c.out() ),

**new** GPCopy ( inChannel: inChannel,

outChannel0: a.out(),

outChannel1: b.out() ),

// 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

**Negator:**

**Diagram:**

out

a

in

GPCopy

d

c

b

GPlus

GPrefix

Negator

**Code:**

**Negator.groovy**

outChannel.write(-inChannel.read())

**DifferentiateNeg.groovy**

**def** differentiateList = [ **new** GPrefix ( prefixValue: 0,

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

outChannel: c.out() ),

**new** GPCopy ( inChannel: inChannel, outChannel0: a.out(),

outChannel1: b.out() ),

//insert a constructor for Negator

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

outChannel: d.out()),

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

inChannel1: d.**in**(),

outChannel: outChannel )

]

**Output:**

Differentiated Numbers

0

1

2

3

4

5

6

7

**Questions:**

I find that the minus method is more pleasing, as it is more sensible to have a method for subtracting than it is to negate the value and then add it.

Exercise 3.2

**Diagram:**

GSPairsA

a

out

in

c

GPlus

b

GSCopy

GTail

GSPairsB

GTail

in

b

c

out

a

GPlus

GSCopy

**Code:**

**GSCopy**

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

{

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

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

outChannel0.write(i)

outChannel1.write(i)

}

**GSquares**

**With GSPairsA:**

**def** testList = [ **new** GNumbers ( outChannel: N2I.out() ),

**new** GIntegrate ( inChannel: N2I.**in**(),

outChannel: I2P.out() ),

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

outChannel: outChannel)

]

**With GSPairsB:**

**def** testList = [ **new** GNumbers ( outChannel: N2I.out() ),

**new** GIntegrate ( inChannel: N2I.**in**(),

outChannel: I2P.out() ),

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

outChannel: outChannel)

]

Output:

With GSPairsA:

Squares

With GSPairsB:

Squares

1

4

9

16

25

36

49

**Questions:**

When GSPairsA is used the channel a is sent first from GSCopy, and reaches GPlus, which is in turn waiting for the other in channel, which will never arrive as GSCopy is still sending through a, as such the process never completes. In GSPairsB the b channel is sent first, reaches GTail which then sends out c which reaches GPlus, which then waits for the a channel, which is available.

**Exercise 3.3:**

GParprint outputs the data in order while GPrint would not, as such the GParPrint method will make it easier to present the data in a table.

**Exercise 4.1:**

**Questions:**

If line 25 is removed the previous value will continue to be printed along with the new value. This occurs because the previous value still exists in the channel and must be read in order to be discarded.

When multiple reset values are entered the program deadlocks. This is because the number of values being entered from the reset channel outnumber those coming from GSuccessor. This works when there are two values in the system because there will always be a free process, but when more are added each process will be used and the deadlock occurs.

**Exercise 4.2:**

Diagram:

in

a

out

GSCopy

GPrefix

Code:

resetChannel

b

c

ResetSuccessor

**ResetNumbers:**

**def** testList = [ **new** GPrefix ( prefixValue: initialValue,

outChannel: a.out(),

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

**new** GPCopy ( inChannel: a.**in**(),

outChannel0: outChannel,

outChannel1: b.out() ),

// requires a constructor for ResetSuccessor

**new** ResetSuccessor( resetChannel: resetChannel,

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

outChannel: c.out())

]

**ResetSuccessor:**

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

{

// deal with inputs from resteChannel and inChannel

// use a priSelect

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

**if** (index == 0 )

{

inChannel.read()

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

outChannel.write(resetValue)

}

**else**

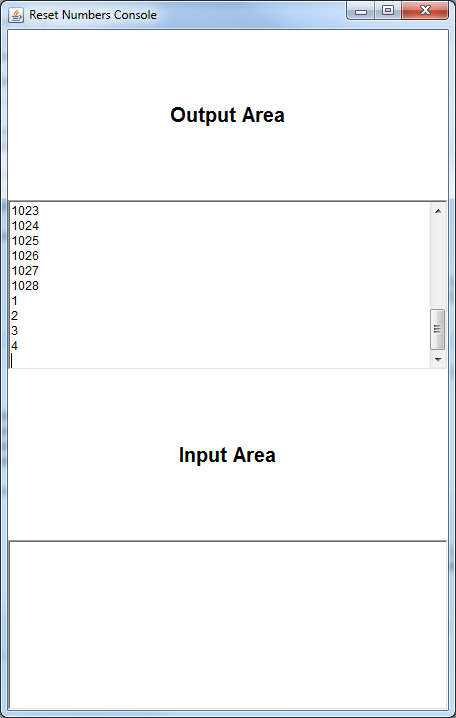
{

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

}

}

Output:



Questions:

If inchannel.read() is removed from ResetSuccessor the same issue encountered in 4.1 will occur and the program will deadlock.

**Exercise 5.1:**

If the delay is increased for either QProducer or QConsumer the time between the process outputting messages will be equal to that delay. If the delay for both is increased the process will take as long as the greatest delay.

Exercise 5.2:

Controller

Suspend

factor

injector

GPrint

Scale

GFixedDelay

GNumbers

scaledData

timedData

data

Code:

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

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

**case** SUSPEND :

// deal with suspend input

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

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

result.original = inValue

result.scaled = inValue

outChannel.write ( result )

**break**

**case** INJECT:

// deal with inject input

scaling = injector.read() //this is the resume signal as well

*println* "Injected scaling is $scaling"

suspended = **false**

timeout = timer.read() + DOUBLE\_INTERVAL

timer.setAlarm ( timeout )

**break**

**case** TIMER:

// deal with Timer input

timeout = timer.read() + DOUBLE\_INTERVAL

timer.setAlarm ( timeout )

scaling = scaling \* 2

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

**break**

**case** INPUT:

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

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

result.original = inValue

result.scaled = inValue \* scaling

outChannel.write ( result )

// deal with Input channel

**break**

} //end-switch

} //end-while

Output:

Original Scaled

0 0

1 2

2 4

3 6

4 8

Normal Timer: new scaling is 4

5 20

6 24

7 28

8 32

Normal Timer: new scaling is 8

9 72

10 80

11 88

12 96

13 104

Question:

I find that for this exercise using the preconditions is a more elegant solution as due to there being less conditions to check it will be cheaper to preform.

Exercise 6.1

Diagram:

Code:

Output:

Questions:

Exercise 7.1

Diagram:

client0

Client1

server0

Server1

C02S0request

S02S1request

S12S0send

S12S0request

S02S1send

C12S1request

S12C1send

S02C0send

Question:

The deadlock occurs when the servers each try to access each other at the same time. As they are both anticipating a response, neither will be able to progress until the response from the other arrives, which it never will.

Exercise 8.1