Fundamentals of Parallel Systems

Lab Book

**C2**

**2.1**

**Consumer.groovy:**

//insert a modified println statement

*println* "Your input multiplied by the factor is: " + i

**CreateSetsOfEight.groovy:**

// put v into outList and read next input

outList.add(v)

**Multiplier.groovy:**

// write i \* factor to outChannel

outChannel.write(i \* factor)

// read in the next value of i

i = inChannel.read()

Run Multiplier.groovy:

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

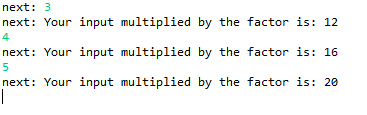
**new** Multiplier ( inChannel: connect1.**in**(),

outChannel: connect2.out(), factor:4 ),

**Diagram:**



**Working Screenshot:**



**2.2**

**ListToStream.groovy:**

// hint: output list elements as single integers

**for**(**int** i=0; i<3; i++)

{

outChannel.write(inList[i])

}

inList = inChannel.read()

**GenerateSetsOfThree.groovy:**

//write the terminating List as per exercise definition

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

**CreateSetsOfEight.groovy:**

// put v into outList and read next input

outList[i] = v

v = inChannel.read()

**Questions:**

What change is required to output objects containing six integers?

Change the for loop in ‘CreateSetsOfEight.groovy’ from 0..7 to 0..5

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

###############################################################

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) ?

Any remainders are put into a list, but the list is not printed until it becomes full

**C3**

**3.1**

**Differentiate.groovy:**

// insert a constructor for Minus

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

inChannel1: c.in(),

outChannel: outChannel)

**Minus.groovy:**

// output one value subtracted from the other

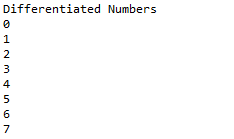
// be certain you know which way round you are doing the subtraction!!

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

**Diagram:**



**Working Screenshot:**



**DifferentiateNeg.groovy:**

//insert a constructor for Negator

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

outChannel: d.out()),

**Negator.groovy:**

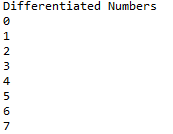
//output the negative of the input value

outChannel.write(inChannel.read() \* -1)

**Diagram:**



**Working Screenshot:**

****

**Questions:**

Which is the more pleasing solution? Why?

The minus approach is more pleasing as it is simpler to follow and requires less steps, whereas using the negator creates the need for another process, and as a result of that things become a little more complex. However, this could simplify problems that are larger and more complex overall, but in terms of this task the simpler solution is better.

**3.2**

**Diagrams:**

GSPairsA:



GSPairsB:



**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

//first modification is to insert a constructor for GSPairsA

// then run the network using TestGSCopy

//second modification replace the constructor for GSPairsA with GSPairsB

// then run the network again using TestGSCopy

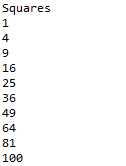
// you will then be able to compare the behaviour and also to

// explain why this happens!

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

//new GSPairsA ( inChannel: I2P.in(), outChannel: outChannel )

**Working Screenshots:**

GSPairsA: GSPairsB

**Questions:**

Determine the effect of the change.

GSPairsA doesn’t work as it causes the system to deadlock, whereas GSPairsB runs smoothly.

Why does this happen?

The difference between GSPairsA and GSPairsB is that the output channels are swapped. GSCopy writes serially, which creates a possibility for the system to deadlock due to an output not being ready to receive data at the correct time, which is exactly what goes wrong when using GSPairsA. Switching the output channels around solves the problem as shown in GSPairsB.

**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?

Because GParPrint works in parallel, it can receive multiple inputs simultaneously. GPrint would need to be used in several different instances to get the same job done. This would make the system much more difficult to write, maintain and use. GParPrint simplifies the program to complete the same task.

**3.4**

###############################################################

**C4**

**4.1**

**Questions:**

What happens if line 25 of ResetPrefix (Listing 4-1) is commented out? Why?

The system uses in the inChannel to check whether outChannel is ready for use. When inChannel.read() returns data, the output channel is ready. If line 25 (inChannel.read()) is commented out, data from resetChannel is written straight to the outChannel, even though it isn’t ready to receive it yet, causing a second set of numbers to be displayed. This is due to a number being sent into the system before the previous reset number has finished its cycle. Explore what happens if you try to send several reset values hence, explain what happens and provide a reason for this.

If another reset value is entered into the system, a deadlock will occur as no processes will be available to accommodate it.

**4.2**

**Diagram:**



**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)

{

inChannel.read();

outChannel.write(resetChannel.read())

}

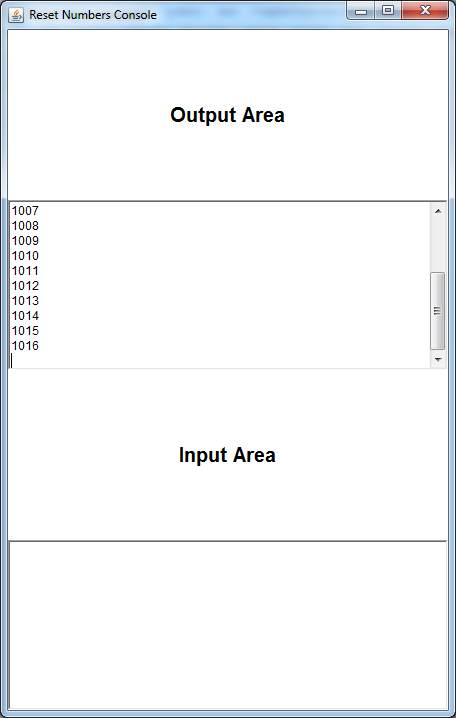
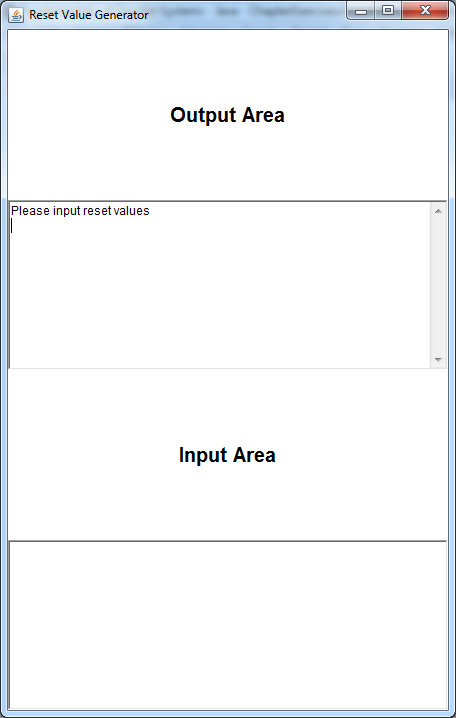
**else**

{

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

}

**Working Screenshots:**

**Questions:**

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

The problem remains as the number of processes and channels within the system has not changed. Deadlocks would still occur if 3 numbers were in the system at once.

**C5**

**5.1**

**Questions:**

What do you conclude from these experiments?

Altering the delay on QProducer and QConsumer doesn’t affect the program as the processes run independently, meaning they don’t have to wait on one another.

**5.2**

**Diagram:**



**Scale.groovy:**

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

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

**case** SUSPEND :

// deal with suspend input

suspend.read()

factor.write(scaling)

suspended = **true**

preCon[INJECT] = **true**

println "Suspended!"

**break**

**case** INJECT:

// deal with inject input

scaling = injector.read()

println "Injected scaling: $scaling"

timeout = timer.read() + DOUBLE\_INTERVAL

timer.setAlarm (timeout)

suspended = **false**

preCon[INJECT] = **false**

**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:

// deal with Input channel

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

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

result.original = inValue

**if** (suspended)

{

result.scaled = inValue

}

**else**

{

result.scaled = inValue \* scaling

}

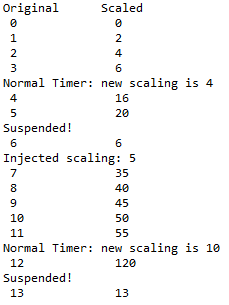
outChannel.write(result)

**break**

} //end-switch

} //end-while

**Working Screenshot:**



**Questions:**

Which is the more elegant formulation? Why?

Using pre-conditions is the most elegant option as it simplifies the code into a much more accessible and readable format without sacrificing the way that it operates. It also allows for easier expansion in future.

**C6**

**6.1**

**Diagram:**

**Scale.groovy:**

**Working Screenshot:**

**Questions:**

Which is the more elegant formulation? Why?

Using pre-