7 Deadlock: An Introduction

Deadlock and livelock are defined and then demonstrated by means of:

- firstly, a trivial example, and
- secondly by means of a more complex but typical situation

Deadlock occurs whenever a network of processes gets into a state where none of the processes is able to continue execution. A similar and related problem is that of livelock, which occurs when part of a process network operates in such a manner as to exclude some of the processes from execution, while others appear to continue execution. A first simple example, based upon the producer – consumer pattern already discussed demonstrates the ease with which a deadlocked system can be created.

7.1 Deadlocking Producer and Consumer

Listing 7-1 gives the coding for a process `BadP`. The process has two channels {12, 13}. Its `run` method initially prints a starting message {16} after which it enters a loop {18}. A message indicating the process is about to write to `outChannel` {19} is printed and then the output takes place {20}. The same actions are then undertaken for a `read` method on `inChannel` {21, 22}. A message indicating that the end of the loop has been reached is printed {23} and the process loops back to {18}.

```
10 class BadP implements CSProcess {
11
12   def ChannelInput inChannel
13   def ChannelOutput outChannel
14
15   def void run() {
16     println "BadP: Starting"
17
18     while (true) {
19       println "BadP: outputting"
20       outChannel.write(1)
21       println "BadP: inputting"
22       def i = inChannel.read()
23       println "BadP: looping"
24     }
25   }
26 }
```

Listing 7-1 BadP Process Coding

Listing 7-2 gives the coding for an equivalent matching process `BadC`, which has an identical structure to `BadP` except that the messages produced are different.

```
10 class BadC implements CSProcess {
11
12   def ChannelInput inChannel
13   def ChannelOutput outChannel
14
15   void run() {
16     println "BadC: Starting"
17
18     while (true) {
19       println "BadC: outputting"
20       outChannel.write(1)
21       println "BadC: inputting"
22       def i = inChannel.read()
23       println "BadC: looping"
24     }
25   }
26 }
```

Listing 7-2 BadC Process Coding

When these processes are executed the console displays the messages shown in Output 7-1.

```
BadC: Starting
BadC: outputting
BadP: Starting
BadP: outputting
```

Output 7-1 Messages Resulting from the Parallel Execution of BadP and BadC

It can be seen that both processes start and achieve an output at lines {19} respectively but cannot make further progress. The reason for this can be seen by inspection because both processes are attempting to output to each other at the same time and neither can undertake the corresponding input operation. It is shown diagrammatically in Figure 7-1, in which increasing time is represented down the page.

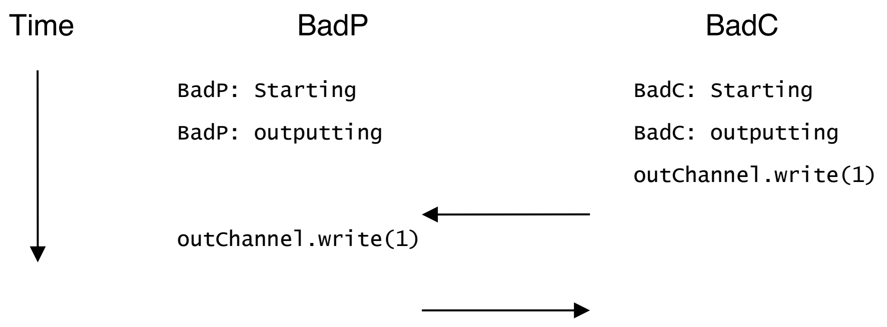

Figure 7-1 Diagrammatic Representation of Deadlocked Process Interactions

In this simple situation the outcome is obvious and easy to see, both processes are trying to output at the same time and thus neither can progress any further because neither can undertake the matching channel read operation. In more complex process networks this is much more difficult to see. Tools are available such as FDR (Formal Systems Europe Ltd, 2013) and Spin (Holzmann G.J., 2013) which can analyse networks of processes for deadlock and livelock but these are limited in the scale of network that can be processed. A different solution is available which can avoid deadlock by engineering design but first we shall investigate a more complex example.

7.2 Multiple Network Servers

A common feature of modern networks is the ability to access many servers from the same workstation. In the background, the network administrator may implement some form of mirror system so that the servers are backed up on each other. When such systems were installed in early network designs there often occurred periods when the network ran very slowly or actually came to halt. The only recourse was to reboot the servers. The problem was this situation was unpredictable.

In this section we shall build a pair of servers that operate in a naïve manner and exhibit this behaviour simply by the order in which data is accessed. The behaviour only requires two clients, which are able to access data from each of the servers but which, crucially access the data through only one server. The system structure is shown in Figure 7-2. Each server has one client and if the client needs to access data that is not available on its own server an automatic access from its server is made to the other server. In order to improve performance of the servers we will allow their clients to initiate another request for data if the previous request has been passed to the other server. Thus requests are interleaved. The connections are shown as double ended arrows to indicate there is a request phase and the subsequent return of a data value corresponding to the request.

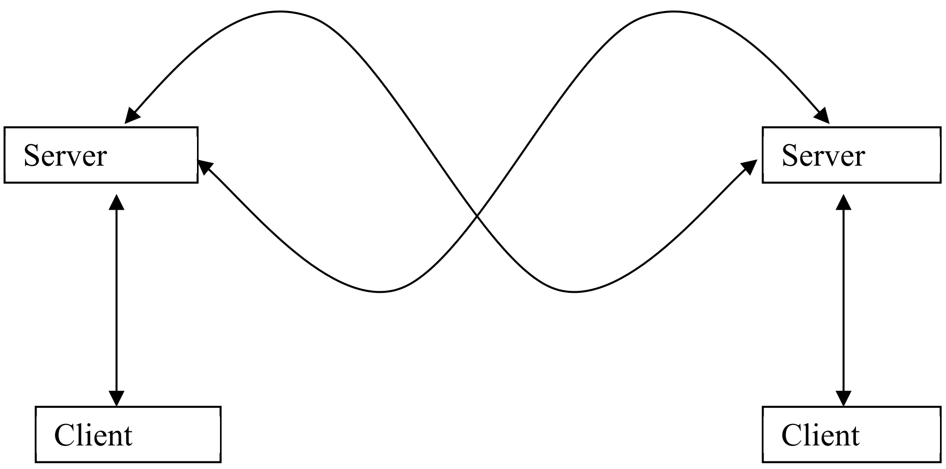

Figure 7-2 Cross-Coupled Clients and Servers System

The servers are implemented as `Maps` comprising 10 elements in each server. The keys for the map entries are distinct. Each client has a list of map entries it wishes to access by key value.

7.2.1 The Client Process

The coding for a Client process is shown in Listing 7-3. The `Client` process has two channel properties `requestChannel` {13}, which is used to send requests to its `Server` and `receiveChannel` {12} used to return results from the `Server,` to the `Client`. The property `selectList` {15} is a `List` initialised to the set of entries in the `Server`'s `Map` that are to be accessed.

```
10 class Client implements CSProcess{
11
12    def ChannelInput receiveChannel
13    def ChannelOutput requestChannel
14    def clientNumber
15    def selectList = [ ]
16
17    void run () {
```

```
18      def iterations = selectList.size
19      println "Client $clientNumber has $iterations values in $selectList"
20
21      for ( i in 0 ..< iterations) {
22        def key = selectList[i]
23        requestChannel.write(key)
24        def v = receiveChannel.read()
25      }
26
27      println "Client $clientNumber has finished"
28    }
29 }
```

Listing 7-3 Client Process Structure

The number of elements in the `selectList` is found {18} and this is used as the range of a `for` loop {21–25}. The client process identity and the list of keys it is going to access are then printed out. Each `key` is found in sequence {22} and this is written to the `requestChannel` {23}. The `Client` then reads the response from the `Server` {24}. At this point the process could print out the returned value but we choose not to. It should be noted that the `Client` may have to wait for the response from its `Server` {24}, if its `Server` has to access the other `Server` because it does not contain the required key itself. A message is printed when the `Client` finishes {21}.

7.2.2 The Server Process

The coding for the `Server` process is shown in Listing 7-4. The `Server` process has three pairs of channels {12–17}, as can be observed from Figure 7-2. The channels `clientRequest` and `clientSend` provide the connections to the `Client` process. The channels `thisServerRequest` and `thisServerReceive` are used by this `Server` to make a request to the other `Server` and then receive a response back. The channels `otherServerRequest` and `otherServerReceive` are used by the other server to make a request to this server. Recall that we are only considering a situation in which there are only two servers. The property `dataMap` {18} holds a map of the keys and values stored in the `Server`.

```
10 class Server implements CSProcess{
11
12    def ChannelInput clientRequest
13    def ChannelOutput clientSend
14    def ChannelOutput thisServerRequest
15    def ChannelInput thisServerReceive
16    def ChannelInput otherServerRequest
17    def ChannelOutput otherServerSend
18    def dataMap = [ : ]
19
20    void run () {
21      def CLIENT = 0
```

```
def OTHER_REQUEST = 1
def THIS_RECEIVE = 2
def serverAlt = new ALT ([clientRequest,
                              otherServerRequest,
                              thisServerReceive])
while (true) {
  def index = serverAlt.select()

  switch (index) {
    case CLIENT :
      def key = clientRequest.read()
      if ( dataMap.containsKey(key) )
        clientSend.write(dataMap[key])
      else
        thisServerRequest.write(key)
      //end if
      break
    case OTHER_REQUEST :
      def key = otherServerRequest.read()
      if ( dataMap.containsKey(key) )
        otherServerSend.write(dataMap[key])
      else
        otherServerSend.write(-1)
      //end if
46              break
47            case THIS_RECEIVE :
48              clientSend.write(thisServerReceive.read() )
49              break
50          } // end switch
51        } //end while
52      } //end run
53 }
```

Listing 7-4 The Server Process Coding

The `Server` can receive inputs from three different sources; its `Client`, a request from the other `Server` and a result from the other `Server` in response to a request made by this `Server`. This is reflected in the creation of three case constants {21-23} and an alternative over the three input channels {24–27}. The `Server` then loops over which ever alternative guard is enabled and `Select`ed followed by executing the related code body in a `switch` statement {28, 30}.

In the case of a `CLIENT` request, the requested key value is read {32} and then a test is made to see if that key is present in the `Server` {33}. If the `key` is present then value in the `dataMap` entry corresponding to the `key` value is sent back to the `Client` {34}; otherwise the request is passed to the other server {36}.

In the case of an `OTHER_REQUEST` from the other server {39–46}, the `CLIENT` code body described above is repeated except that a -1 value is returned {44} if a map entry with the requested key value is not found (This should not happen!).

Finally, in the case `THIS_RECEIVE`, which is a response to a request made by this server on the other server {47-49}, the received value is returned to the `Client` {48}.

7.2.3 Running the Network of Clients and Servers

The script used to test the client and server model is shown in Listing 7-5. The eight channels are defined {10-17}, where the notation X2Y implies that the writing (`.out`()) end of the channel is in the process represented by X and the reading (.in()) end of the channel is in the Y process.

The `maps` associated with each server are then defined {19–20}. The list of key values that each client is to access is then specified {22, 23} and it should be noted that both clients read values from both servers.

```
10 def S02S1request = Channel.one2one()
11 def S12S0send = Channel.one2one()
12 def S12S0request = Channel.one2one()
13 def S02S1send = Channel.one2one()
14 def C02S0request = Channel.one2one()
15 def S02C0send = Channel.one2one()
16 def C12S1request = Channel.one2one()
```

```
17 def S12C1send = Channel.one2one()
18
19 def server0Map = [1:10,2:20,3:30,4:40,5:50,6:60,7:70,8:80,9:90,10:100]
20 def server1Map = [11:110,12:120,13:130,14:140,15:150,
21                  16:160,17:170,18:180,19:190,20:200]
22
23 def client0List = [1,12,3,14,15,16,7,18,9,10]
24 def client1List = [11,12,13,14,15,6,17,8,19,20]
25
26 def client0 = new Client ( requestChannel: C02S0request.out(),
27                            receiveChannel: S02C0send.in(),
28                            selectList: client0List,
29                            clientNumber: 0)
30
31 def client1 = new Client ( requestChannel: C12S1request.out(),
32                            receiveChannel: S12C1send.in(),
33                            selectList: client1List,
34                            clientNumber: 1)
35
36 def server0 = new Server ( clientRequest: C02S0request.in(),
37                            clientSend: S02C0send.out(),
38                            thisServerRequest: S02S1request.out(),
39                            thisServerReceive: S12S0send.in(),
40                            otherServerRequest: S12S0request.in(),
41                            otherServerSend: S02S1send.out(),
42                            dataMap: server0Map)
43
44 def server1 = new Server ( clientRequest: C12S1request.in(),
45                            clientSend: S12C1send.out(),
46                            thisServerRequest: S12S0request.out(),
47                            thisServerReceive: S02S1send.in(),
48                            otherServerRequest: S02S1request.in(),
49                            otherServerSend: S12S0send.out(),
50                            dataMap: server1Map)
51
52 def network = [client0, client1, server0, server1]
53 new PAR (network).run()
```

Listing 7-5 The Defintion of Channels, Server Maps and Client Key Lists

The processes are then defined such that `client0` is connected to `server0` and `client1` is connected to `server1`. The process network is then defined {52} and the network run {53}.

The result, see Output 7-2, from the execution of the network, shown in Figure 7-2 and Listing 7-5, can be seen, by inspection, to have not completed because the final output line that indicates the client processes have finished and terminated is not printed.

```
Client 0 has 10 values in [1, 12, 3, 14, 15, 16, 7, 18, 9, 10]
Client 1 has 10 values in [11, 12, 13, 14, 15, 6, 17, 8, 19, 20]
```

Output 7-2 Correct Output from the Clients and Servers network

If we replace lines {23, 24} of Listing 7-5 with the following:

```
def client0List = [1,2,3,4,5,6,7,18,9,10]
def client1List = [11,12,13,4,15,16,17,18,19,20]
```

then the result shown in Output 7-3 is generated. This shows that both clients have accessed all the data they require because they have produced the final completion message. Thus the ordering of client requests is significant for the correct operation of the network of processes. This is something that should not be allowed to occur. By inspection we can see that both servers can get into a state where they are trying to access the other server either by making a request or by both waiting to receive a response so that neither of them can complete a communication.

```
Client 1 has 10 values in [11, 12, 13, 4, 15, 16, 17, 18, 19, 20]
Client 0 has 10 values in [1, 2, 3, 4, 5, 6, 7, 18, 9, 10]
Client 0 has finished
Client 1 has finished
```

Output 7-3 Deadlocked Client Server Results

The programmer should resist the temptation to insert print statements to determine what has happened because these can often remove the time critical nature of the interactions and the system appears to work. This occurs because the network of processes is sequentialised by outputting to the common console or other display device. This will be explored in the exercises. It can even depend upon the length of the print statement as to whether a non-working system can be made to work.

It should be noted also that different executions of the same network, unaltered in any way, will often produce different deadlock situations. It is dependent on the dynamics of the network. It can also vary with the combination of processor, operating system, java virtual machine and the number of cores and how they are utilised.

7.3 Summary

In this chapter we have demonstrated how deadlock can occur, first in a very simplistic manner and secondly, in a more complex set of interactions that are very hard to foresee. In the second case the programmer attempted to ensure that the servers were undertaking as many interactions as possible with the clients. A more careful programmer might have decided that instead of separating the request and response made to the other server they would ensure that the response from the other server was received and returned to the client before embarking upon another interaction. This would work in the case where both servers were executing on the same processor because the interactions would be interleaved and thus some distinct ordering might give the impression that the network operated correctly.

However, this would be a fool's paradise in the case of servers running on different machines because in due course the situation would arise where both servers were trying to send a request to each other and the deadlock would occur, possibly after a long period, which had given the impression that the system worked correctly and that the fault lay elsewhere.

Thus we have to find a design pattern that permits the safe design of parallel systems and that is the content of the next chapter.

7.4 Exercises

Exercise 71

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. You will probably find it useful to add a property to the Server process by which you can identify each Server.