7 Deadlock: An Introduction

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 are able to continue. 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 {2, 3}. Its `run` method initially prints a starting message {5} after which it enters a loop {6}. A message indicating the process is about to write to `outChannel` {7} is printed and then the output takes place {8}. The same action is then undertaken for a `read` method on `inChannel` {9, 10}. A message indicating that the end of the loop has been reached is printed {11} and the process loops back to {6}.

```
01     class BadP implements CSProcess {

02       def ChannelInput inChannel
03       def ChannelOutput outChannel

04       def void run() {

05         println "BadP: Starting"

06         while (true) {

07           println "BadP: outputting"
08           outChannel.write(1)

09           println "BadP: inputting"
10           def i = inChannel.read()

11           println "BadP: looping"
12         }
13       }
14     }
```

Listing 7-1 BadP Process Coding

Listing 7-2 gives the coding for an equivalent process `BadC`, which has an identical structure to `BadP` except that the messages produced are different. When these processes are executed the console displays the messages shown in Output 7-1.

```
15     class BadC implements CSProcess {

16       def ChannelInput inChannel
17       def ChannelOutput outChannel

18       def void run() {

19         println "BadC: Starting"
20         while (true) {

21           println "BadC: outputting"
22           outChannel.write(1)

23           println "BadC: inputting"
24           def i = inChannel.read()

25           println "BadC: looping"
26         }
27       }
28     }
```

Listing 7-2 BadC Process Coding

```
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 attempt to output at lines {8 and 22} 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.

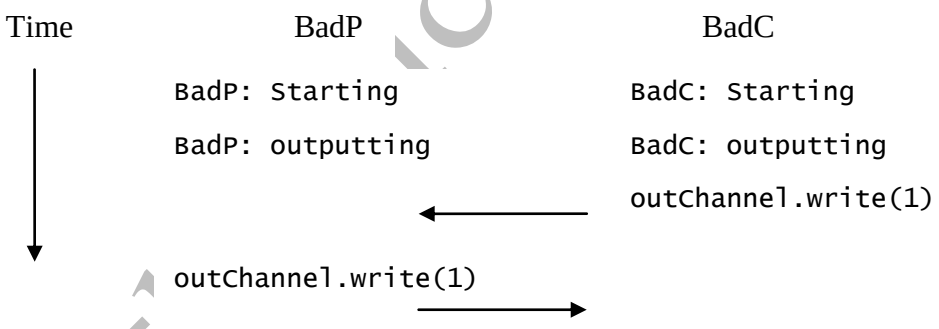

Figure 7-1 Diagrammatic Representation of Deadlocked Process Interactions

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 [ref] and Spin [ref] 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.

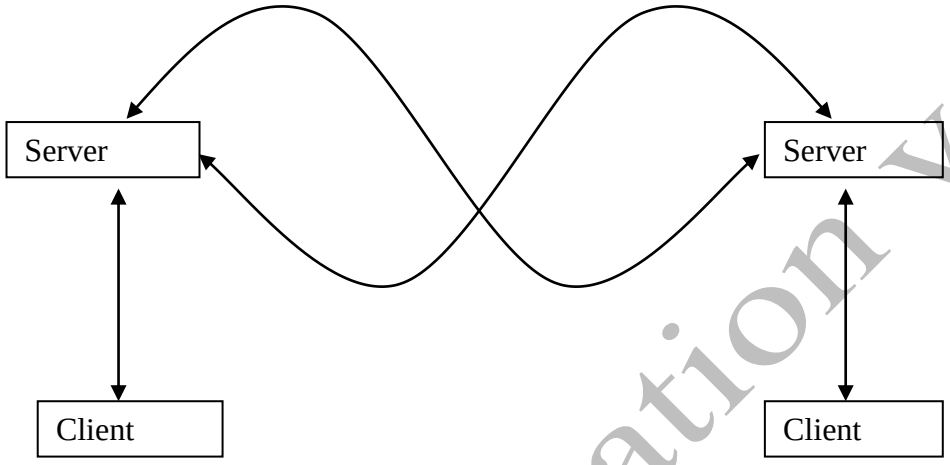

Figure 7-2 Cross-Coupled Clients and Servers System

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` {31}, which is used to send requests to its `Server` and `receiveChannel` {30} used to return results from the `Server`, to the `Client`. The property `selectList` {33} is a `List` initialised to the set of entries in the `Server`'s `Map` that are to be accessed. The number of elements in the `selectList` is found {35} and this is used as the range of a `for` loop {37-42}. Each `key` is found in sequence {38} and this is written to the `requestChannel` {39}. The `Client` then reads the response from the `Server` {40} and then prints out the key and the value of its associated map entry in the `Server` {41}. It should be noted that the `Client` may have to wait for the response from its `Server` {40}, if its `Server` has to access another `Server` because it does not contain the required key itself.

```
29     class Client implements CSProcess{

30       def ChannelInput receiveChannel
31       def ChannelOutput requestChannel
32       def clientNumber

33       def selectList = [ ]

34       void run () {
35         def iterations = selectList.size
36         println "Client $clientNumber has $iterations values in $selectList"

37         for ( i in 0 ..< iterations) {
38           def key = selectList[i]
39           requestChannel.write(key)
40           def v = receiveChannel.read()
41           println "Client $clientNumber: with $key has value $v"
42         }
43       }
44     }
```

Listing 7-3 Client Process Structure

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 {46-51}, 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`. The channels `otherServerRequest` and `otherServerRequest` 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` holds a map of the keys and values stored in the `Server`.

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 {54-56} and an alternative over the three input channels {57-59}. The `Server` then loops over which ever alternative guard is enabled followed by executing the related code body in a `switch` statement.

In the case of a `CLIENT` request, the requested key value is read {64} and then a test is made to see if that key is present in the `Server` {65}. If the key is present then value of the map entry corresponding to the key value is sent back to the `Client` {66}; otherwise the request is passed to the other server {69}.

In the case of an `OTHER_REQUEST` from the other server {72-80}, the `CLIENT` code body described above is repeated except that a -1 value is returned {78} 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 {81-83}, the received value is returned to the `Client` {82}.

```
45     class Server implements CSProcess{

46       def ChannelInput clientRequest
47       def ChannelOutput clientSend

48       def ChannelOutput thisServerRequest
49       def ChannelInput thisServerReceive

50       def ChannelInput otherServerRequest
51       def ChannelOutput otherServerSend

52       def dataMap = [ : ]

53       void run () {
54         def CLIENT = 0
55         def OTHER_REQUEST = 1
56         def THIS_RECEIVE = 2
57         def serverAlt = new ALT ([ clientRequest,
58                                    otherServerRequest,
59                                    thisServerReceive])

60         while (true) {
61           def index = serverAlt.select()
62           switch (index) {

63             case CLIENT :
64               def key = clientRequest.read()
65               if ( dataMap.containsKey(key) ) {
66                 clientSend.write(dataMap[key])
67               }
68               else {
69                 thisServerRequest.write(key)
70               }
71               break

72             case OTHER_REQUEST :
73               def key = otherServerRequest.read()
74               if ( dataMap.containsKey(key) ) {
75                 otherServerSend.write(dataMap[key])
76               }
77               else {
78                 otherServerSend.write(-1)
79               }
80               break

81             case THIS_RECEIVE :
82               clientSend.write(thisServerReceive.read() )
83               break

84           }
85         }
86       } //end run
87     }
```

Listing 7-4 The Server Process Coding

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 {88-95}, 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 {96-98}. The list of key values that each client is to access is then specified and it should be noted that both clients read values from both servers.

The processes are then defined such that `client0` is connected to `server0` and `client1` is connected to `server1`. The process network is then defined {123} and the network run.

```
88      def One2OneChannel S02S1request = Channel.createOne2One()
89      def One2OneChannel S12S0send = Channel.createOne2One()

90      def One2OneChannel S12S0request = Channel.createOne2One()
91      def One2OneChannel S02S1send = Channel.createOne2One()

92      def One2OneChannel C02S0request = Channel.createOne2One()
93      def One2OneChannel S02C0send = Channel.createOne2One()

94      def One2OneChannel C12S1request = Channel.createOne2One()
95      def One2OneChannel S12C1send = Channel.createOne2One()

96      def server0Map = [1:10,2:20,3:30,4:40,5:50,6:60,7:70,8:80,9:90,10:100]

97      def server1Map = [11:110,12:120,13:130,14:140,15:150,
98                        16:160,17:170,18:180,19:190,20:200]

99      def client0List = [1,2,3,4,5,6,7,18,9,10]

100     def client1List = [11,12,13,4,15,16,17,18,19,20]

101     def client0 = new Client ( requestChannel: C02S0request.out(),
102                                receiveChannel: S02C0send.in(),
103                                selectList: client0List,
104                                clientNumber: 0)

105     def client1 = new Client ( requestChannel: C12S1request.out(),
106                                receiveChannel: S12C1send.in(),
107                                selectList: client1List,
108                                clientNumber: 1)

109     def server0 = new Server ( clientRequest: C02S0request.in(),
110                                clientSend: S02C0send.out(),
111                                thisServerRequest: S02S1request.out(),
112                                thisServerReceive: S12S0send.in(),
113                                otherServerRequest: S12S0request.in(),
114                                otherServerSend: S02S1send.out(),
115                                dataMap: server0Map)

116     def server1 = new Server ( clientRequest: C12S1request.in(),
117                                clientSend: S12C1send.out(),
118                                thisServerRequest: S12S0request.out(),
119                                thisServerReceive: S02S1send.in(),
120                                otherServerRequest: S02S1request.in(),
121                                otherServerSend: S12S0send.out(),
122                                dataMap: server1Map)

123     def network = [client0, client1, server0, server1]

124     new PAR (network).run()
```

Listing 7-5 The Script Used To Create The Client Server Network

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 functioned correctly. Each client accesses the server map entries in the correct order and every value that should have been printed has been. We can also see that each client accesses entries in both servers.


```
Client 0 has 10 values in [1, 2, 3, 4, 5, 6, 7, 18, 9, 10]
```

```
Client 0: with 1 has value 10
Client 1 has 10 values in [11, 12, 13, 4, 15, 16, 17, 18, 19, 20]
Client 1: with 11 has value 110
Client 1: with 12 has value 120
Client 0: with 2 has value 20
Client 0: with 3 has value 30
Client 1: with 13 has value 130
Client 0: with 4 has value 40
Client 1: with 4 has value 40
Client 1: with 15 has value 150
Client 1: with 16 has value 160
Client 0: with 5 has value 50
Client 1: with 17 has value 170
Client 0: with 6 has value 60
Client 1: with 18 has value 180
Client 0: with 7 has value 70
Client 1: with 19 has value 190
Client 0: with 18 has value 180
Client 0: with 9 has value 90
Client 0: with 10 has value 100
Client 0 has finished
Client 1: with 20 has value 200
Client 1 has finished
```

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

If we replace lines {99, 100} with the following:

```
def client0List = [1,2,3,14,15,6,7,18,9,10]
def client1List = [11,12,13,4,5,16,17,8,19,20]
```

then the result shown in Output 7-3 is generated. That is 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.

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.

```
Client 1 has 10 values in [11, 12, 13, 4, 5, 16, 17, 8, 19, 20]
Client 1: with 11 has value 110
Client 1: with 12 has value 120
Client 0 has 10 values in [1, 2, 3, 14, 15, 6, 7, 18, 9, 10]
Client 1: with 13 has value 130
Client 0: with 1 has value 10
Client 0: with 2 has value 20
Client 1: with 4 has value 40
Client 0: with 3 has value 30
```

Output 7-3 Deadlocked Client Server Results

It should be noted also that different executions of the same network, unaltered in any way, will produce different deadlock situations.

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 fools 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

1. 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.

2. When you have found this out try the same modification to the working version of the system. What happens? Why? What are the implications of this discovery?