13 Accessing Shared Resources: CREW

In this chapter and the next, techniques are described that were developed for, and are used most often in shared memory multi-processing systems. In such systems great care has to be taken to ensure that processes running on the same processor do not access an area of shared memory in an uncontrolled manner. Up to now the solutions have simply ignored this problem because all data has been local to and encapsulated within a process. One process has communicated data to another as required by the needs of the solution. The process and channel mechanisms have implicitly provided two capabilities, namely synchronisation between processes and mutual exclusion of data areas. In shared memory environments the programmer has to be fully aware of both these aspects to ensure that neither is violated.

Mutual exclusion ensures that while one process is accessing a piece of shared data no other process will be allowed access regardless of the interleaving of the processes on the processor. Synchronisation ensures that processes gain access to such shared data areas in a manner that enables them to undertake useful work. The simplest solution to both these problems is to use a pattern named CREW, Concurrent Read Exclusive Write, which, as its names suggests, allows any number of reader processes to access a piece of shared data at the same time but only one writer to process to access the same piece of data at one time. The CREW mechanism manages this requirement and in sensible implementations also imposes some concept of fairness. If access is by multiple instances of reader and writer processes then one could envisage a situation where the readers could exclude writers and vice versa and this should be ameliorated as far as is possible. The JCSP implementation of a CREW does exhibit this capability of fairness, as shall be demonstrated.

At the simplest level the CREW has to be able to protect accesses to the shared data and the easiest way of doing this is to surround each access, be it a read or write with a call to a method that allows the start of an operation and subsequently when the operation is finished to indicate that it has ended. Between such pairs of method calls the operation of the CREW is guaranteed. Thus the programmer has to surround access to shared data with the required start and end method calls be they a read or write to the shared data. It is up to the programmer to ensure that all such accesses to the shared data are suitably protected.

In the JCSP implementation of CREW we extend an existing storage collection with a `Crew` class. Then we ensure that each access that `puts` data into the collection is surrounded by a `startWrite`() and `endWrite`() pair of method calls on the `Crew`. Similarly, that each `get` access is surrounded by a `startRead`() and `endRead`() method call. Internally, the `Crew` then ensures that access to the shared storage collection is undertaken in accordance with the required behaviour. Further, fairness can be implemented quite simply by ensuring that if the shared data is currently being accessed by one or more reader processes then as soon as a writer process indicates that it wishes to put some data into the shared collection then no further reader processes are permitted to start reading until the write has finished. Similarly, a sequence of write processes, each of which requires exclusive access, will be interposed by reader process accesses as necessary.

13.1 CrewMap

Listing 13-1 shows a simple extension of a `HashMap` {1} by means of an instance of a `Crew` {2}. The `put` and `get` methods of `HashMap` are then overwritten with new versions that surround them with the appropriate start and end method calls {4, 6} and {9, 11}, between which the normal `HashMap`'s `get` and `put` methods can be called as usual.

```
01     class CrewMap extends  HashMap {

02       def theCrew = new Crew()

03       def Object put ( Object itsKey, Object itsValue ) {
04         theCrew.startWrite()
05         super.put ( itsKey, itsValue )
06         theCrew.endWrite()
07       }

08       def Object get ( Object itsKey ) {
09         theCrew.startRead()
10         def result = super.get ( itsKey )
11         theCrew.endRead()
12         return result
13       }
14     }
```

Listing 13-1 The CrewMap Class Definition

At this point a word of caution has to be given. This arises because Java allows exceptions to be thrown at any point. Thus in the above formulation it might be possible for the lines that represent normal access to the shared resource {5, 10} to fail. In such a case the call to the `end` synchronisation method {6, 11} will never happen and thus the `Crew` will fail in due course as the required locks will not be released. The associated documentation for JCSP Crew discusses this in more detail. The solution is to encapsulate the access in a `try .. catch .. finally` block. The problem arises because Java invokes code sequences that are not part of the coding sequence and thus the programmer has to be very wary of these possibilities. In the following description we shall presume that all access is well behaved and such a fault will not occur.

Once the `CrewMap` has been defined it can be used in a solution that requires multiple processes access to its shared data collection. Figure 13-2 shows such a typical application. In this case two `Read` and two `Write` processes access the shared `DataBase` resource. The coding of the `DataBase` process is shown in Listing 13-2.

13.2 The DataBase Process

The `DataBase` process has two channel list properties {16, 17} comprising the channels used by the `Read` and `Write` processes to access it. Additionally, properties are required that define the number of such `Read` and `Write` processes, `readers` and `writers` respectively {18, 19}.

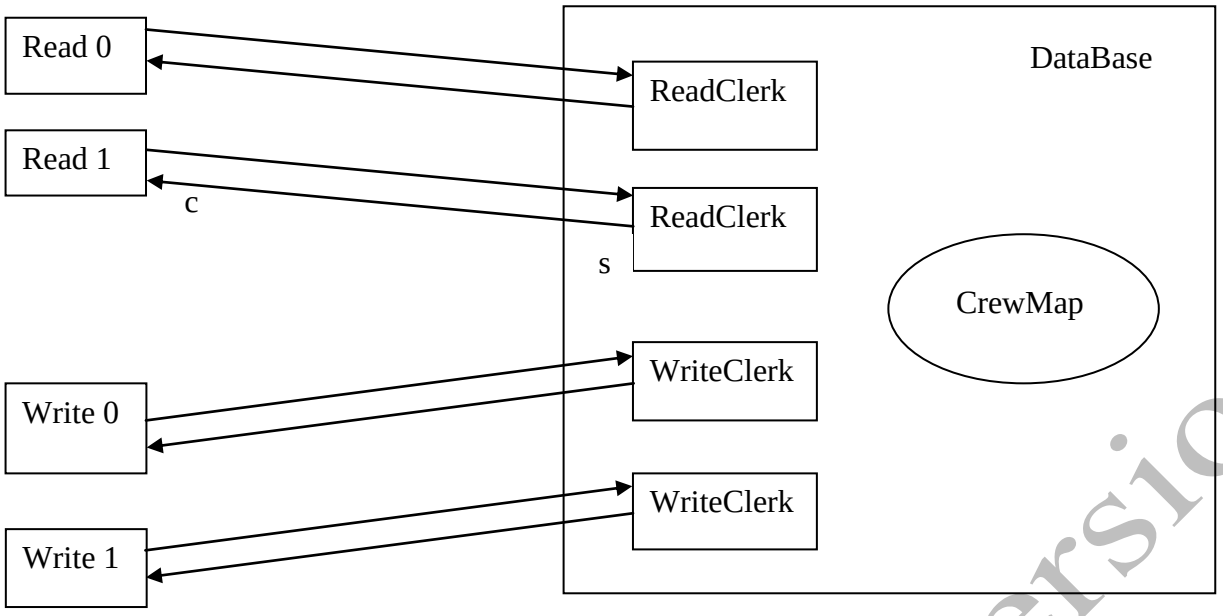

Figure 13-1 A Simple Use of CrewMap

```
15    class DataBase implements CSProcess {

16      def ChannelInputList inChannels
17      def ChannelOutputList outChannels
18      def int readers
19      def int writers

20      void run () {
21        def crewDataBase = new CrewMap()
22        for ( i in 0 ..< 10 ) {
23          crewDataBase.put ( i, 100 + i)
24        }

25        def processList = []
26        for (i in 0..< readers) {
27          processList.putAt (i, new ReadClerk ( cin: inChannels.getAt(i),
28                                                cout: outChannels.getAt(i),
29                                                data: crewDataBase) )
30        }

31        for ( i in 0 ..< writers ) {
32          processList.putAt ( ( i + readers),
33                    new WriteClerk ( cin: inChannels.getAt ( i + readers ),
34                                     cout: outChannels.getAt (i + readers ),
35                                     data: crewDataBase ) )
36        }

37        new PAR (processList).run()
38      }
39    }
```

Listing 13-2 The DataBase Process definition

The `run` method {20} essentially creates the structure shown in Figure 13-1. An instance of `CrewMap` is defined called `crewDataBase` {21}. The shared resource `crewDataBase` is then populated with initial values {22-24}, which initialises the first ten locations with the values 100 to 109 in sequence. An empty `processList` {25} is then defined that will hold instances of the required `ReadClerk` and `WriteClerk` processes. The required number of `ReadClerk` processes are then created {27} and placed in `processList`. Each `ReadClerk` is allocated the corresponding element of the `inChannels` and `outChannels` channel lists {27, 28}. Finally, the `ReadClerk` process has its `data` property initialised to the `crewDataBase` itself {29}. The `WriteClerk` processes are instantiated in the same manner {31-36} ensuring that the correct elements of the `inChannels` and `outChannels` lists are allocated to the

processes {32}. This means that the all the ReadClerk and WriteClerk processes have shared access to the crewDataBase. The processList can now be passed to a PAR for running {37}.

Communication between the Read and Write processes and the DataBase is achieved by a single class called DataObject {40}, see Listing 13-3. DataObject comprises three properties {41-43}, pid hold the identity number of the accessing Read or Write process, location holds the index of the resource element to be accessed and value is either the value read from that element or that is to be written to the element.

```
40    class DataObject implements Serializable {
41      def int pid
42      def int location
43      def int value
44    }
```

Listing 13-3 The Definition of DataObject

It should be noted that this formulation of the DataBase contains no alternative (ALT) as might be expected from previous examples. This arises because we are using a formulation that contains a CREW that essentially provides the same functionality, but only for shared memory applications. The advantage of the alternative is that it can be used to alternate over networked channels and thus is more flexible. It also has the advantage of exposing the alternative concept that is so important in the modelling of parallel systems.

13.3 The Read Clerk Process

Listing 13-4 shows the ReadClerk process, which has channel input and output properties cin {46} and cout {47} respectively and a data property {48} that accesses the CREW resource.

```
45    class ReadClerk implements CSProcess {

46      def ChannelInput cin
47      def ChannelOutput cout
48      def CrewMap data

49      void run () {
50        def d = new DataObject()
51        while (true) {
52          d = cin.read()
53          d.value = data.get ( d.location )
54          cout.write(d)
55        }
56      }
57    }
```

Listing 13-4 The ReadClerk Process

The run method {49-54} defines an instance d of type DataObject {50} after which the value of d is read from cin {52}. The location property of d is then used to access the CrewMap property data {53} to get the corresponding value which is then stored in the value property of d. The revised value of d is then written to the channel cout {54}.

13.4 The Write Clerk Process

The WriteClerk process is shown in Listing 13-5 and is fundamentally the same as that shown in the ReadClerk process except that a new value is put into the shared resource {66}. The unmodified DataObject d is written back to the corresponding Write process to confirm that the operation has taken place {67}.

```
58     class WriteClerk implements CSProcess {

59       def ChannelInput cin
60       def ChannelOutput cout
61       def CrewMap data

62       void run () {
63         def d = new DataObject()
64         while (true) {
65           d = cin.read()
66           data.put ( d.location, d.value )
67           cout.write(d)
68         }
69       }
70     }
```

Listing 13-5 The WriteClerk Process

13.5 The Read Process

The `Read` process is shown in Listing 13-6. It has three properties. A channel by which it writes to the database `r2db` {72} and one by which it reads returned values `db2r` {73}. The last property, `id` {74}, is the identity number of the `Read` process. The `run` method {75} initialises a `DataObject` with the `Read` process' `id` {76} and then reads a value from each location of the shared resource in sequence {77}, printing out each returned value {81}. This is achieved by allocating the loop value `i` to the `location` property of `d` {78}. The instance `d` is then written to the shared resource using the channel `r2db` {79}. The process then waits until it can read the returned `DataObject` into `d` using the channel `db2r` {80}.

```
71     class Read implements CSProcess {

72       def ChannelOutput r2db
73       def ChannelInput db2r
74       def int id

75       void run () {
76         def d = new DataObject(pid:id)
77         for ( i in 0 ..<10 ) {
78           d.location = i
79           r2db.write(d)
80           d = db2r.read()
81           println "Reader ${id}: Location - ${d.location} has value ${d.value}"
82         }
83       }
84     }
```

Listing 13-6 The Read Process

13.6 The Write Process

The `Write` process is shown in Listing 13-7 and is very similar to the `Read` process except that the elements of the shared resource are accessed in reverse order, that is from 9 to 0 {94}. The value written to the shared resource is dependant upon the `id` of the writing process {96}.

```
85     class Write implements CSProcess {
86
87       def ChannelOutput w2db
88       def ChannelInput db2w
89       def int id
90
91       void run () {
92         def d = new DataObject(pid:id)
93         for ( j in 0 ..<10 ) {
94           def i = 9 - j
95           d.location = i
96           d.value = i + ((id+1)*1000)
97           w2db.write(d)
98           d = db2w.read()
99         }
100      }
101    }
```

Listing 13-7 The Write Process

13.7 Creating the System

The script that invokes the `DataBase` system is shown in Listing 13-8.

```
102    def nReaders = Ask.Int ( "Number of Readers ? ", 1, 5)
103    def nWriters = Ask.Int ( "Number of Writers ? ", 1, 5)

104    def connections = nReaders + nWriters

105    One2OneChannel[] toDatabase = Channel.createOne2One(connections)
106    One2OneChannel[] fromDatabase = Channel.createOne2One(connections)
107    One2OneChannel[] consoleData = Channel.createOne2One(connections)

108    def toDB = new ChannelInputList(toDatabase)
109    def fromDB = new ChannelOutputList(fromDatabase)

110    def readers = ( 0 ..< nReaders).collect { r ->
111                            return new Read (id: r,
112                                             r2db: toDatabase[r].out(),
113                                             db2r: fromDatabase[r].in(),
114                                            toConsole: consoleData[r].out())
115                            }

116    def writers = ( 0 ..<nWriters).collect { w ->
117                            def wNo = w + nReaders
118                            return new Write ( id: w,
119                                               w2db: toDatabase[wNo].out(),
120                                               db2w: fromDatabase[wNo].in(),
121                                               toConsole: consoleData[wNo].out())
122                            }

123    def database = new DataBase ( inChannels:  toDB,
124                                  outChannels: fromDB,
125                                  readers: nReaders,
126                                  writers: nWriters)

127    def consoles = ( 0 ..< connections).collect { c ->
128                            def frameString = c < nReaders ?
129                                               "Reader " + c :
130                                               "Writer " + (c - nReaders)
131                           return new GConsole (toConsole: consoleData[c].in(),
132                                                frameLabel: frameString )
                       }
133    def procList = readers + writers + database + consoles

134    new PAR(procList).run()
```

Listing 13-8 The Script to Invoke the DataBase System

Initially, the number of `Read` and `Write` processes is obtained {102, 103} by a console interaction. The total number of connections to the `DataBase` is then calculated as `nConnections` {104}. The system uses a `GConsole` propcessfor each `Read` and `Write` process to display the outcome of the interactions with the `DataBase`. The channels used to connect the `Read` and `Write` processes to the `Database` and the

GConsoles are then defined {105-107}. The corresponding channel lists toDb and fromDb are then defined {108, 109}, which connect the Read and Write processes to the DataBase.

The required number of Read processes is then created in the list readers {110-115}. Each instance uses the closure property r to identify the required element of the previously declared channel arrays that connect the process to the DataBase and its GConsole process. Similarly, the required number of Write processes is defined {116-122}. The property wNo {117} is used to ensure that the index used to associate Write process channel indeces is offset by the number of Read processes.

An instance of the DataBase process is then created {123-126}, using the previously declared channel lists. The list consoles {127-132} contains the instances of GConsole required to connect to the Read and Write processes. Finally, procList is created as the addition of all the process lists and the database process {133} and then run {134}.

Outputs 13-1 and 13-2 show the output from the running of the system when it is started with two Read and two Write processes. The order in which the Write process have been executed can be determined from the values that have been read by the two Read processes. Recall that the Write processes access the database locations in reverse order to the Read processes. The outputs indicate that the implementation of the Crew class is inherently fair because the values read by the Read processes change from the initial values to the modified values about half way through the cycle. The values read from locations 5 and 6 also vary indicating that state of the DataBase was in flux at that point in the access cycles with read and write operation fully interleaved.

```
Location 1 has value 101
Location 2 has value 102
Location 3 has value 103
Location 4 has value 104
Location 5 has value 105
Location 6 has value 1006
Location 7 has value 2007
Location 8 has value 2008
Location 9 has value 2009
Reader has finished
```

Output 13-1 Output From Read process 0

```
Location 1 has value 101
Location 2 has value 102
Location 3 has value 103
Location 4 has value 104
Location 5 has value 1005
Location 6 has value 2006
Location 7 has value 2007
Location 8 has value 2008
Location 9 has value 2009
Reader has finished
```

Output 13-2 Output From Read process 1

13.8 Summary

In this chapter we have investigated a typical mechanism used in shared memory multi-processing system. The formulation tends to hide the interactions that take place because these are captured somewhat remotely in the CrewMap class definition.

13.9 Challenge

Rewrite the system so that a Crew is not used and the DataBase process alternates over the input channels from the Read and Write processes. The system should capture the same concept of fairness as exhibited in the CREW based solution.