

13.1 A simple search system

As a first simple example, we will consider a problem in the domain of searching. Suppose we have an array of 150,000 records with people. We want to find all persons with an age between 18 and 24. We may do this by writing a method that goes through all the records and check the ages. We may also split the task into three tasks and define three activities where each activity handles one third of the records. This should speed up the process, compared to using just one activity.

The basic elements of a parallel system are objects executing computations in parallel - called *active objects*. Below we show how the the model of the search-activity may be split between active objects:

```
aSimpleSearcher: obj BasicSystem
  searcherA: obj BasicProcess
    search(1,50000)
  searcherB: obj BasicProcess
    search(50001,100000)
  searcherC: obj BasicProcess
    search(10001,150000)
  :::
  searcherA.start
  searcherB.start
  searcherC.start
```

The object `aSimpleSearcher` is subclassed from a class `BasicSystem`, which is a class that defines abstractions for describing objects that may execute in parallel. One of these abstractions is class `BasicProcess`, which may be used as a superclass for objects that execute in parallel. We supply the details of `BasicSystem` in a later chapter.

To simplify the example, we assume that we have 150000 records. In general, it is not a good idea to encode constants like this in the code. Instead such values should be a parameter of a class or method or being read as data from some source.

```
class BasicSystem:
  class BasicProcess:
    start: :::
    :::
  :::
```

The object `aSimpleSearcher` contains three searcher-objects, `searcherA`, `searcherB`, and `searcherC` which all are derived from `BasicProcess`. This means that they may execute in parallel. The execution of the three searcher-objects is started by invocation of the `start` method of class `BasicProcess`: `searcherA.start`, etc. They all invoke a `search`-method on their third of the list of records. The `search`-method searches the range of objects within the arguments of `search`.

Next we supply the details of the searching activity, that is the method `search` - we assume that we have a class `Person` with a name and an age attribute:

```
aSimpleSearcher: obj BasicSystem
  records: obj Array(150000, Person)
  search(first, last: var integer):
    for (first):to(last):repeat
```

```

        if ((18 <= records[inx].age)
            and (records[inx].age <= 24)) :then
            collector.insert(records[inx])
    collector: obj :::
    :::
class Person:
    name: var String
    age: var integer
    ...

```

The object `aSimpleSearcher` has a local array, `records`, which is an array containing 150000 `Person`-objects. The method `search` searches the part of the array as defined by its parameters `first` and `last`. If a record matches the search-criteria, the `Person`-object is added to a `collector`-object.

Note: The class `BasicSystem` is placed within a module as described in chapter later in this chapter. This will require a minor adjustment to this example in order to be able to execute it.

The `collector`-object is a `Set`-object:

```
collector: obj Set(Person)
```

This looks simple and straightforward, but we have now arrived at one of the major problems in making parallel programs. Since the three searcher-objects execute in parallel, two or more of them may invoke `collector.insert` at the same time. This may imply that the result of `collector.insert` has an unexpected result. The problem may be the implementation of the `insert` method of `Set`. As we shall see in a later chapter, `insert` manipulates local references in `Set`. This may not work if two or more searcher-objects does this at the same time.

Race conditions

This problem of two or more parallel objects accessing the same objects at the same time is referred to as a *race condition*.

Race conditions may also appear in real life. One example is booking of hotel rooms: A customer may call a hotel and talk to a receptionist and ask for the booking a room on a specific date – it turns out that there is just one available room at that date. While the receptionist talks with the customer another potential customer calls the hotel and another receptionist answers the phone and it turns out that this customer also wants a room on the same day as the first customer.

The two receptionist can of course not both sell the same room, so in practice they have a means for avoiding this. In the old days they had a book where they recorded reservations and before making a reservation, a receptionist must get this book and only one can have the book at a given time. This ensures that a given room can only be booked once. Today most hotels have a computer system that takes care of reservations and avoiding race conditions.

Avoiding race conditions is a critical issue in parallel programming. For parallel programming the `Set` object, `collector`, used in the above example, plays a similar role to the reservation book of the hotel. At most one parallel object at a time may control `collector` just as at most one receptionist may control the reservation book. In the next section, we introduce a language mechanism that makes it possible to avoid race conditions.

The monitor system

There exists a number of language mechanisms that may help avoiding race conditions. In this section we use a *monitor* to define a safe version of *Collector*.

A monitor is an object where it is only possible to invoke at most one method at a given time. That is two or more methods cannot be invoked at the same time. A new class *MonitorProcess* is defined to represent parallel activity that makes use of *Monitor* objects.

Class *Monitor* and class *MonitorProcess* are defined as part of the class *MonitorSystem*:

```
class MonitorSystem:
  class MonitorProcess: BasicProcess
    start: :::
    :::
  class Monitor:
    entry:
      :::
  :::
```

Class *Monitor* has a local method *entry* that must be used as a supermethod of all methods defined within in a subclass of *Monitor*. Only one method that has *entry* as a supermethod can be executed at a given time. For methods that have *entry* as supermethod, only one can be executed at a given time.

Note: As for *BasicSystem*, class *MonitorSystem* is placed within a module as described in chapter and this will also require a minor adjustment to this example in order to be able to execute it.

We may now define the *collector* object as a *Monitor* by encapsulating the *Set*-object within the *Monitor*:

```
collector: obj Monitor
  insert(p: ref Person): entry
    matches.insert(p)
  matches: obj Set(Person)
```

As can be seen, *collector* is subclassed from *Monitor* and *insert* is a submethod of *entry*. This guarantees that at most one invocation of *insert* may be executed at a given time even though it may be called in parallel by the three searcher processes.

A *MonitorProcess* in a *MonitorSystem* can only access data-items defined *locally* in the *MonitorProcess* object, but it may invoke methods of a *Monitor* object. The reason for this is to avoid race conditions if two or more *MonitorProcess* object access the same objects at the same time.

In the simple searcher, the objects to be searched are in the *global* object records. This objects can thus not be accessed by a *MonitorProcess* object. We thus have to organize the records in another way.

In this simple example, we split the *records* array object into three array objects and place one in each searcher object. We define a general *Searcher* class that contains the records and the search method of a given *MonitorProcess*:

```

class Searcher: MonitorProcess
  records: obj Array(50000, Person)
  search(first, last: var integer):
    -"-
  inner(Searcher)

```

We may now define the searcher objects as subclassed from Searcher:

```

searcherA: obj Searcher
  search(1,50000)
searcherB: obj Searcher
  search(1,50000)
searcherC: obj Searcher
  search(1,50000)

```

In this way each Searcher object has its own records to be searched.

The search methods defined above is the same as the one defined in the `aSimpleSearcher`. The difference is that here it is placed locally to the three searcher-objects. Each of them has its own search method although the code is identical and defined in class Searcher, which is a superclass of all three objects.

We now add a MonitorProcess-object that presents the results of the searching. Such a MonitorProcess has to wait for all three Searcher-objects to have finished searching their part of the objects.

```

presenter: obj MonitorProcess
  waitTermination(searcherA, SearcherB, SearcherC)
  collector.scan
  console.print("Found" + current.asText)

```

The method `waitTermination` waits until all its arguments searcherA, SearcherB, and SearcherC have terminated execution.

A scan-method has been added to collector. It scans though all elements in the set of matches, and execute an inner for each element:

```

collector: obj Monitor
  -"-
  scan:
    current: ref Person
    matches.scan
      this(scan).current := current
    inner(scan)

```

The complete system then consists of the following elements:

```

aSafeSearcher: obj MonitorSystem
  class Searcher: -"-
  searcherA: obj Searcher
    search(1,50000)
  searcherB: obj Searcher
    search(1,50000)

```

```

searcherC: obj Searcher
  search(1,50000)
collector: obj Monitor
  insert(P: ref Person): entry
  -"-
  scan:
  -"-
  matches: obj Set(Person)
presenter: obj MonitorProcess
  -"-
searcherA.start
searcherB.start
searcherC.start
presenter.start

```

As being said before, most of the examples in this book is for illustrating basic programming and modelling. In this example we have not dealt with how to add content to the records of the Searcher object. Also organizing the records as done here may not be the best solution in a practical system.

Using a bounded buffer

Computers in general have limited storage with respect to RAM and disk space, and programs have to take this into account.

The next example is also in the domain of searching. We consider a number of clients that makes requests to a server for searching Person-objects with specific values of its attributes, such as name and age above.

The overall structure of this system is:

```

usingBoundedBuffer: obj MonitorSystem
  class Client: MonitorProcess
    mkRequest(field: var String, V: var String) -> R: ref Request:
      :::
  class Request(sendedr: ref Client, field: var String, V: var String):
    :::
  ...
clientA: obj Client("clientA")
  :::
clientB: obj Client("clientB")
  :::
clientC: obj Client("clientC")
  :::

server: obj Monitor
  addRequest(R: ref Request): entry
  :::
  getRequest -> R: ref Request: entry
  :::
  requests: obj BoundedBuffer(#Request,100)

```

```

Searcher: MonitorProcess
  decodeAndSearch(R: ref Request):
    ...
  :::
searcherA: obj Searcher("searcherA")
searcherB: obj Searcher("searcherB")
searcherC: obj Searcher("searcherC")

```

The following code defines a class `Client` that may be used to specify the clients of the system:

```

class Client: MonitorProcess
  mkRequest(field, value: var String) -> R: ref Request:
    R := Request(this(Client),field, value)
    server.addRequest(r)
  inner(Client)
class Request(sender: ref Client, field, V: var String):
  ...
clientA: obj Client
  mkRequest("age","18-24")
  mkRequest("name","John Smith")
  ...
clientB: obj Client
  mkRequest("age","60-65")
  ...
clientC: obj Client
  ...

```

- Class `Client` describes the general structure of a client submitting request to the server.
- Class `Request` describes the structure of a request. It contains the sender of the `Request`, a field holding the name of the attribute to search for, and a parameter `V` holding the value (or interval of values) that must match the `field`.
- The method `mkRequest` creates a `Request` and sends it to the server using `server.addRequest(R)`.
- Three sub of `Client`, `clientA`, `clientB`, and `clientC` subclassed from `Client` are declared, each submitting different requests.

We implement the server as a `Monitor` that keeps track of the various requests.

```

server: obj Monitor
  addRequest(R: ref Request): entry
    request.insert(R)
  getRequest -> R: ref Request: entry
    R := requests.next
  requests: obj BoundedBuffer(100,Request)
  :::

```

- The method `addRequest`, inserts the `Request R` in the array `requests`.
- The method `getRequest` is used by a searcher to get a request. The `requests`-object is of type `BoundedBuffer`, which is a list where a limited number of objects may be stored - in this

case 100.

We have to define a new version of a searcher-object to be used in this example since the searching is more complicated than just searching for a person with an age between 18 and 24.

```
class Searcher: MonitorProcess
  decodeAndSearch(R: ref Request):
    ...
  cycle
    R: ref Request
    R := server.getRequest
    decodeAndSearch(R)
```

```
searcherA: obj Searcher
searcherB: obj Searcher
searcherC: obj Searcher
```

A Searcher-process retrieves a request from the Server. It then has to decode the data-items field and V to find out what to search for. This is done by the method decodeAndSearch.

For a request with field = "age" and V = "60-65", it must read the string field to find out that it is the age attribute that is to be used in the search. And it must decode the String "60-65" to find out that the age must be between 60 and 65. Finally the search method above must have the ages to search for defined as parameters. We don't show the details here.

For a request with field = "name" and "V = "John Smith", a similar decision must be made and another search-method must be written to search for Person-objects where the name-attributes has the given value. Again we don't show the details.

Handling the capacity of the server

As mentioned, the Server stores the request in the requests-object, which can hold a maximum of 100 objects. A client trying to add a Request must therefore check if the buffer is not full, and if it is, wait until some space is available.

The same is the case for a Searcher-object. When calling getRequest, the requests-buffer may be empty and thus no Request can be returned to the Searcher. In this case the method getRequest must wait until a client inserts a Request.

To handle this, a Monitor-object has a method wait that delays execution of an entry-method until a given condition becomes true:

```
wait(condition)
```

We may use wait in the server-object as follows:

```
server: obj Monitor
  addRequest(R: ref Request): entry
    wait(not requests.full)
    request.insert(R)
  getRequest -> R: ref Request: entry
```

```

    wait(not requests.isEmpty)
    R := requests.next
requests: obj BoundedBuffer(100,Request)

```

In the beginning of the `addRequest`-method, a `wait(not requests.full)` is inserted. The invocation `requests.full` returns true if the buffer is full. The method `wait` simply delays the execution of `addRequest` until the buffer is not full. This may happen if a `Searcher`-object removes a `Request` using `getRequest`.

In a similar way, a `wait(not requests.isEmpty)` is inserted in the beginning of `getRequest`. It delays execution of `getRequest` until the buffer is not empty.

Organization of BasicSystem and MonitorSystem

In this section, we describe how the classes `BasicSystem` and `MonitorSystem` are organized in modules. This is of course necessary to understand in order to be able to execute the above programs.

Class `BasicSystem` is placed within the module `BasicSystemLib`:

```

BasicSystemLib: obj
  class BasicSystem:
    class BasicProcess:
      :::

```

Class `MonitorSystem` is similar within a module `MonitorSystemLib`:

```

MonitorSystem: obj BasicSystemLib.BasicSystem
  class MonitorSystem:
    class MonitorProcess:
      :::
    class Monitor:
      :::

```

The necessary adjustments to the above examples implies that the programs must be derived from class `BasicSystem` or class `MonitorSystem`.

The program `aSimpleSearcher` must thus be derived from class `BasicSystemLib.BasicSystem`:

```

aSimpleSearcher: obj BasicSystemLib.BasicSystem
  - " -

```

The program `aSafeSearcher` must be derived from class `MonitorSystemLib.MonitorSystem`:

```

aSafeSearcher: obj MonitorSystemLib.MonitorSystem
  - " -

```

Finally, `usingBoundedBuffer` must also be derived from class `MonitorSystemLib.MonitorSystem`:

```

usingBoundedBuffer: obj MonitorSystemLib.MonitorSystem
  - " -

```