16.2 Preemptive coroutines

In this section, we describe *Preemptive coroutines,* which may be used to implement parallel objects and scheduling of parallel objects.

So far coroutines have been cooperative in the sense that a coroutine executes until it voluntarily suspends execution. Using cooperative coroutines, at most one coroutine executes at a given time during program execution. Synchronization between coroutines may thus not be needed.

A preemptive coroutine may be suspended by an external signal at any time during its execution. That is, a preemptive coroutine does not control when it has to give up control. As we shall see later, this makes it possible to implement *time-sharing* between two or more pre-emptive coroutines.

Time-sharing is a standard computing term that describes the situation where two or more parallel objects share the same computing resource (CPU or core). This enables multi-tasking, which is the concurrent execution of multiple tasks (processes) over a certain period of time. In the simple form where only one CPU/core is available, at most one coroutine executes at a given time, but since it may be suspended (the term *interrupted* is the general computing term here) at an arbitrary point during execution, it simulates the effect of the coroutines executing in true parallel by executing segments of the code in an *interleaved* manner. The time slots allocated to a coroutine is typically in the order of milliseconds, and this implies that it is necessary to synchronize access to common data objects.

A cooperative coroutine may be resumed using a `call`-method. For a preemptive coroutine, an `attach`-method is used:

```
s: obj
   ...
   s.suspend
   ...
...
s.attach(100)
...
```

`s` is executing a sequence of statements. We assume that `s` during its instantiation executes a suspend and return to its invoker. Otherwise it does not makes sense to resume `s`.

The statement `s.attach(100)` implies that `s` is resumed. The argument 100 to `attach` implies that `s` is preemptively suspended after 100 units of execution. The unit depends on the actual implementation of qBeta. For here we assume that the unit is a micro second.

We next sketch an example using preemptive coroutines in the domain of file-processing. The object `smallSys` has three preemptive coroutines `reader`, `processor` and `writer`.

The `reader` reads the next integer from `inFile`; the `processor` computes a function from this value; the result of this is written by the `writer` to the file `outFile`.

The three coroutines communicate values through the `buffer`-objects `inBuffer` and `outBuffer`.

```
smallSys: obj
   inFile,outFile: obj File
```

```
   inBuffer, outBuffer: obj
      add(V: var inter): ...
      get -> V: var integer: ...
      ...
   reader: obj
      reader.suspend
      cycle
         inBuffer.add(inFile.read)
   processor: obj
      compute: ...
      processor.suspend
      cycle
         outBuffer.add(compute(inBuffer.get))
   writer: obj
      writer.suspend
      cycle
         outFile.write(nextValue)
   done -> B: var Boolean: ...
-- open inFile and outFile
loop:
   cycle
      reader.attach(10)
      processor.attach(100)
      writer.attach(10)
      if (done) :then
         leave(loop)
-- close inFile and outFile
```

The three coroutines start by executing a suspend; the `reader` then repeatedly reads a value from `inFile` and adds it to the `inBuffer`; the processor repeatedly gets a value from the `inBuffer`, compute a new value, which is added to the `outBuffer`; the `writer` repeatedly reads a value from the `outBuffer` and writes it to the `outFile`. `Compute` is left unspecified.

The `smallSys` object starts by opening the files; it then repeatedly resumes the three coroutines; the `reader` and `writer` get slots of 10 milliseconds; the `processor` gets a slot of 100 milliseconds. This goes on until the `done`-method returns true – `done` is unspecified.

Note here, that `smallSys` is in full control of how to schedule the coroutines and the size of the timeslots given to each of them. This is a feature that is rarely supported by most mainstream object-oriented languages.

The buffers `inBuffer` and `outBuffer` are shared objects and synchronisation is needed, but left unspecified here.

Although the example is sketchy, it does resemble an example that may arise in practice.