

16.3 Implementing a Monitor System

Here we will show how to implement a Monitor system using preemptive coroutines and Semaphore.

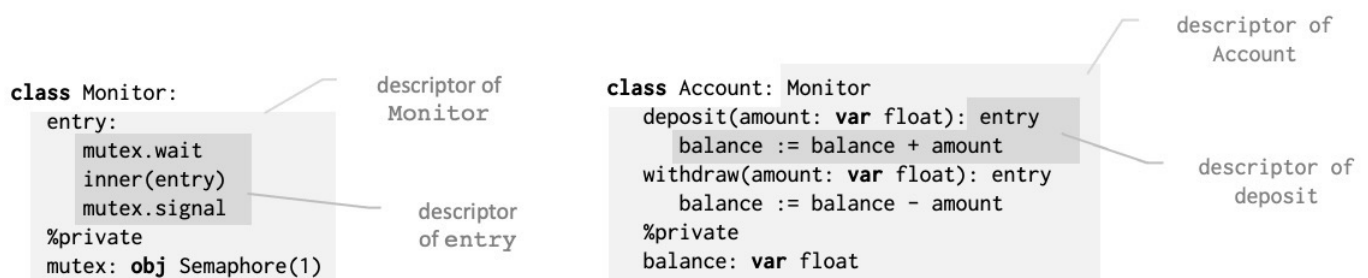
Below we show our first version of class Monitor:

```
class Monitor:
  entry:
    mutex.wait
    inner(entry)
    mutex.signal
  %private
  mutex: obj Semaphore(1)
```

A Monitor-object has a Semaphore attribute and a local method pattern, entry, which is to be used as a supermethod for methods in subclasses of Monitor. As can be seen, entry has an inner(entry); mutex.wait is executed before inner(entry) and mutex.signal is executed after inner(entry).

The Account class may be described as a subclass of Monitor in the following way:

```
class Account: Monitor
  deposit(amount: var float): entry
    balance := balance + amount
  withdraw(amount: var float): entry
    balance := balance - amount
  %private
  balance: var float
```



If anAccount refers to an Account-object, then execution of `anAccount.deposit(200)` has the effect that the supermethod entry of deposit works like a wrapper around the statement `balance := balance + amount`. This ensures that `mutex.wait` is executed before the statement and `mutex.signal` is executed afterwards. The same is the case for execution of a `anAccount.withDraw(300)`. All in all, using entry as a supermethod ensures that at most one deposit or withdraw may be executed at the same time, which guarantees exclusive access to data-items within the Monitor object.

In section , Monitor is used as part of class that also defines a MonitorProcess class. We will show how to define such a system. It will contain the elements shown below:

```
class MonitorSystem
```

```

class Monitor: ...
class MonitorProcess: ...
%private
scheduler: obj ...
SQS: obj ProcessQueue

```

Class `Monitor` is defined as shown above. Class `MonitorProcess` is supposed to be a superclass of all parallel objects in the `MonitorSystem`. The scheduler object handles scheduling of `MonitorProcess`-objects and `SQS` is a queue of `MonitorProcess`-objects that are ready for being executed.

Class `MonitorProcess` may be defined as follows:

```

class MonitorProcess:
  start:
    status := ACTIVE
    SQS.insert(this(MonitorProcess))
  status: var integer
  inner(MonitorProcess)
  status := TERMINATED

```

A `MonitorProcess` has four attributes:

- A `start` method that sets the status of the `MonitorProcess` object to `ACTIVE`.
- An integer variable `status` holding the status of the `MonitorProcess`.
- An `inner`-statement that implies execution of items in a subclass of `MonitorProcess`.
- Finally a statement setting `status` to `TERMINATED` whereafter execution of the `MonitorProcess` object ends.

Next we describe the Scheduler object:

```

scheduler: obj
  active: ref MonitorProcess
  cycle
    active := SQS.next
    if (active <> none) :then
      active.attach(100)
      if (active.status = ACTIVE) then
        SQS.insert(active)

```

The scheduler has a data-item `active` that is a reference to the `MonitorProcess` object currently being executed.

Then it has a `cycle`-statement that forever executes:

- A reference to the next `MonitorProcess` in the queue `SQS` of active `MonitorProcess`-objects is assigned to `active`.
- If there are no references in `SQS`, `active` will get the datum `none`.
- If `active` is not `none`, the method `active.attach(100)` is invoked implying that execution of `active` is resumed. If `active` has not been executed before, execution starts from the

beginning of `active`; if `active` has been executed before and execution has been suspended, execution is resumed after the point of suspension.

- `Active` will be preemptively suspended after 100 time units, but it may also terminate before the 100 time units have appeared.
- If `active` is preemptively suspended, `active.status = ACTIVE` and `active` is re-inserted into SQS.
- Otherwise `active` has terminated (`status = TERMINATED`) execution and thus not re-inserted into SQS.