

16.1 Implementing Lock and Semaphore

As mentioned, we need a basic synchronization mechanism in order to define both the low-level synchronization mechanisms and the high-level synchronization mechanisms.

In order to implement synchronization mechanisms, we rely on using an *atomic operation*. An operation is *atomic* if it cannot be interrupted while its is being executed. We have previously seen (section) that incrementing an integer like $n := n + 1$ is not an atomic operation. Reading a value is atomic, just as writing a value is atomic.

A common example of an atomic operation available on computers and accessible from a programming language is *compare-and-swap*. It compares the contents of a memory location with a given value and, only if they are the same, modifies the location to the given value. The operation returns a value that indicates if the operation succeeded.

Our programming language has a `cmpAndSwap` operation that may be applied to an `integer`. Suppose we have an `integer` variable `mutex`:

```
mutex: var integer
```

We may apply `cmpAndSwap` in the following way:

```
B: var Boolean
B := mutex.cmpAndSwap(v1,v2)
```

- If `mutex = v1`, then nothing happens and `cmpAndSwap` returns true.
- If `mutex <> v1`, then `cmpAndSwap` assigns `v2` to `mutex` and returns false.
- The execution of `cmpAndSwap` is atomic, i.e. the execution cannot be interrupted.

Suppose we want to use an integer variable to keep track of whether or not a parallel object is in a critical region. In th example below, the variable `mutex` may have the value 0 or 1. Zero (0) indicates that no other parallel objects in the critical regions, and one (1) indicates the opposite.

A parallel object attempting to reserve the `mutex` then repeatedly tries to write a one (1) to `mutex` until a zero is returned:

```
mutex: var integer
...
loop: do
  if (mutex.cmpAndSwap(1,1)) :then
    -- operation failed, try again
    restart(loop)
-- operation succeeded, enter critical region
...
mutex := 0
```

If a one is returned by `cmpAndSwap`, the `mutex` is busy and som other parallel object is in its critical region. If zero is returned the `mutex` is free and `cmpAndSwap` assigns it a one and enters it critical region.

Implementing Lock

We may now show how to implement the Lock class:

```
class Lock:
  mutex: var integer
  get:
    loop: do
      if (mutex.cmpAndSwap(1,1)) :then
        restart(loop)
  free:
    mutex := 0
  mutex := 0
```

Class Lock has two method get and free. Get does the same as in the previous example, and free assigns zero to mutex.

Implementing Semaphore

Here we show how to implement a counting semaphore using a lock. Class Semaphore has the two methods wait and signal, and it has two private objects mutex and Q (short for Queue).

```
class Semaphore(cnt: var integer)
  -- Implementation of a counting semaphores
  -- cnt is the initial value of the semaphore
  -- requires: cnt > 0
  wait:
    mutex.get -- attempt to get the Lock
    cnt := cnt - 1
    if (cnt < 0) :then
      disable -- is this needed?
      Q.insert(TheActiveProcess)
      enable
      mutex.free
      theActiveProcess.suspend
    :else
      mutex.free
  signal:
    mutex.get
    cnt := cnt + 1
    if(cnt <= 0) :then
      P := Q.removeNext
      disable
      P.mkActive -- P.wakeUp?
      enable
    mutex.free
  %private
  mutex: obj Lock
  Q: obj OrderedList(GeneralProcess)
```

The wait-method attempts to get the mutex. If it succeeds it decrements the counter cnt.

If `cnt` becomes less than 0, the Semaphore cannot grant more resources. It then inserts `theActiveProcess` into its local queue, releases the mutex and suspends execution. `theActiveProcess` is a predefined variable (so its declaration is not shown above) that refers to the current active Process.

If `cnt` is greater than or equal to zero, it just releases the mutex since there was at least one free resource.

The `signal`-method also attempts to get the mutex and if it succeeds it increments `cnt`.

If `cnt` is less than or equal to zero it means that some processes is waiting in the queue `Q` to get the lock. It thus removes the next element in `Q` and makes it active and releases the mutex.

if `cnt` is greater than zero, no processes are waiting to request a resource - it then just releases the mutex.