Monitors

Last updated: February 4th 2019, at 12.39pm

1 Problem Statement

Consider the following problem statement:

- There is *one* resource that may only be accessed by *one* process at any one time;
- There are many processes that access the resource using the methods `requestResource(int priority)` and `releaseResource()`;
- The parameter `priority` is an integer with a value between 0 and 10, i.e. $0 \leq$ `priority` ≤ 10;
- The value of `priority` is not bound to a process, i.e. any process may call `requestResource` with any (legal) value of `priority`;
- The implementation should give priority to resource requests with the highest priority.

Assume that you can use a Java-like programming language (or pseudocode) with the following constructs:

- Monitor definitions of the form:

  ```
  monitor name {
      /* variable declarations, e.g. */
      int variableName = 0;

      /* method declarations, e.g. */
      void proc() {
          ...
      }
  }
  ```

  which may contain any number of method definitions and local variables.

- a **Condition** class, with the methods **wait()** and **signal()**.

Nested monitors are not allowed, and monitors cannot be assigned a priority. The implementation of monitors satisfies the immediate resumption requirement.

2 Exercises

1. (Pen and paper) Using constructs described above define a **monitor** containing (at least) the two methods `requestResource(int priority)` and `releaseResource()`, such that these satisfy the given problem statement. Hint: assign a **Condition** object to each priority.

2. **Logbook question.** Implement your solution in Java, using `Lock`s and `Condition`s. I.e., implement a `LockResourceManager` class that uses `Lock`s and `Condition`s to implement the `requestResource(int priority)` and `releaseResource()` methods described above. Use the code provided (see section 3) to test your implementation, and try some tests of your own.

3. **Model question.** Now implement your solution in “raw” Java — i.e. without using `Lock`s and `Condition`s, only synchronised methods and/or synchronised statements. Hint: define your own `lock()` and `unlock()` methods, giving you a finer control over mutual exclusion in the code. I.e., implement a `RawResourceManager` class that this time uses synchronised methods and/or statements to implement the `requestResource(int priority)` and `releaseResource()` methods described above. Use the code provided (see section 3) to test your implementation, and try some tests of your own.

3 The Code

Some code is provided to get you started with questions 2 and 3. This code defines the following classes:

`Resource` Models a simple resource.

`ResourceManager` Specifies the resource manager interface — i.e. defines the signatures of all the methods that a resource manager should implement.

`BasicResourceManager` Implements all the methods required by `ResourceManager`, *except* `requestResource(int priority)` and `releaseResource()`.

`ResourceUser` Models a resource user.

`ResourceSystem` Combines `Resource`s, `ResourceManager`s and `ResourceUser`s to define a system where there is a set of resources and a set of users, with all users trying to make use of all resources, and with access to these resources managed by the resource managers.

Note that this code will not currently compile — see below.

`ResourceSystemTest` Used to define some tester code. Note that this code does *not* test the resource system in the sense of reporting a pass or a fail in a test. It simply runs the resource system specified in the test. It is up to the user to inspect the output produced by this test to check that the system displays the behaviour expected.

`ResourceError` Basic error handling for errors arising from the management and use of resources.

See the code and its documentation for further details.

Note that `ResourceSystem`, and hence `ResourceSystemTest`, will not currently compile. This is because the `addResource(String name,int maxUseages)` method in `ResourceSystem` includes code that tries to create an instance of `ResourceManager` (this is on line 58 of `ResourceSystem.java`). It is not possible to instantiate an interface. When you have completed your implementation of `LockResourceManager` and/or `RawResourceManager` you can test the implementation by replacing the attempted instantiation of `ResourceManager` by a call of the constructor for your class.

The code bundle also includes an implementation of the car park system discussed in the lecture. This is for information only, and does not form part of the code for this week's exercises.