

## Exercise 11: Synchronisation mechanisms in SCOOP: semaphores and barriers

Hand-out: 13 June  
Due: 20 June

The goal of the exercise is to implement semaphores and barriers in SCOOP. You will also get acquainted with the use of inheritance, deferred classes, and *once* features in SCOOP.

Before solving this exercise, make sure you have downloaded the latest version of SCOOP tools, i.e. scoop2scoopli 0.2.2008 and SCOOPLI 0.3.2005.

### 1. Semaphores

Download the semaphore example from the course website. Your project directory should contain the following classes:

- ROOT\_CLASS
- WORKER
- MY\_SEMAPHORE
- SEMAPHORE\_USER
- PROCESS

Class WORKER implements continuous processes whose task is to enter the critical section protected by a semaphore, print some text (character by character), and leave the critical section. Class MY\_SEMAPHORE implements semaphores of arbitrary size.

### Questions:

- 1.1. How would you implement a mutex using class MY\_SEMAPHORE?
  - 1.1.1. How does the application behave when a mutex is used? What happens when a semaphore of a larger size is used?
- 1.2. Feature {WORKER}.step does not contain any loops. How do we achieve continuous execution of a worker's activity?
  - 1.2.1. What OO techniques are used to achieve this?
  - 1.2.2. How would you reimplement WORKER so that it terminates?
- 1.3. Features P () and V () have been placed in class SEMAPHORE\_USER. Why don't we include them directly in WORKER?
- 1.4. Feature semaphore is implemented as *once function*. What are the advantages of such a solution?
  - 1.4.1. How would you reimplement semaphore to avoid the use of once mechanism?
  - 1.4.2. What is the semantics of once functions with separate result type? With non-separate result type? Try it out.

## 2. Barriers

Download the barrier example from the course website. Your project directory should contain the following classes:

- ROOT\_CLASS
- WORKER
- MY\_BARRIER
- BARRIER\_USER
- PROCESS

Class WORKER implements continuous processes whose task is to join the group protected by a barrier, and leave the group when it has been formed. (While waiting for his buddies to complete the group, a worker reads a newspaper.) Class MY\_BARRIER implements barriers of arbitrary size.

### Questions:

- 2.1. What are the major differences between a barrier and a semaphore? Can you see any similarities in the implementation?
  - 2.1.1. Can you implement feature *ok\_to\_join* based solely on *count* and *size*?
  - 2.1.2. Why do we need two Boolean flags: *ok\_to\_join* and *ok\_to\_leave*?
  - 2.1.3. If the whole group of workers had to perform some action, how would you express it?
- 2.2. How would you implement a mutex using class MY\_BARRIER?
- 2.3. How does the application behave when the size of the barrier is greater than the number of workers?

**Have fun!**