Assignment 6: SCOOP

ETH Zurich

1 Breakfast Running Time

1.1 Background

Reasoning about the execution times of a concurrent SCOOP program, in the context of breakfast.

1.2 Task

Consider the following SCOOP program being executed on a processor z:

```
bread.cut
toaster.toast
pan.fry
meal.compose
Result := meal.is_cooked and bread.is_delicious
meal.eat
```

The object-processor associations are given as follows: *pan* is handled by processor p, *bread* and *toaster* by processor q, and *meal* by processor r. The call *bread.cut* takes 20 time units until it returns, *toaster.toast* 30 time units, *pan.fry* 20 time units, *meal.compose* 40 time units, *meal.eat* 20 time units. Assume the queries are instantaneous. What is the minimum time for execution of this program? Justify your answer.

1.3 Solution

The bread and toaster must run in sequence, taking 50 time units. The pan and the first meal take 20 and 40 time units respectively. All 3 of these times are run in parallel, so their combined running time is the maximum, or 50 time units. The program then synchronizes at the **Result** line, waiting for the response of the meal and bread. There is an additional 20 time unit delay at the end.

The total running time is then 70 time units.

2 Interpreting a SCOOP program

2.1 Background

The code in listing 1 shows the participants of a crazy office. Note that the *BOSS* class is the root of this system.

Listing 1: crazy office classes

```
class BOSS

create
```

```
  make

feature
  evil_supervisor : separate EVIL_SUPERVISOR
  nice_supervisor: separate NICE_SUPERVISOR
  worker: separate WORKER

  make
      -- Create supervisors and a worker and use the supervisors to drive the worker.
    do
      create evil_supervisor
      create nice_supervisor
      create worker
      print ("boss: I am about to ask the supervisors to do their job.")
      run ( evil_supervisor , nice_supervisor)
      print ("boss: I am done.")
    end

  run ( a_evil_supervisor : separate EVIL_SUPERVISOR; a_nice_supervisor: separate
      NICE_SUPERVISOR)
      -- Use the supervisors to drive the worker.
    do
      a_evil_supervisor . convince (worker)
      a_nice_supervisor . convince (worker)
      a_evil_supervisor . convince (worker)
      a_nice_supervisor . convince (worker)

      if ( a_evil_supervisor . done and a_nice_supervisor.done) then
        print ("boss: The supervisors are done.")
      end
    end
end

class EVIL_SUPERVISOR

feature
  done: BOOLEAN
    -- Did I convince a worker?

  convince (a_worker: separate WORKER)
      -- Convince 'a_worker' that he is not done as soon as he thinks that he is done.
    require
      a_worker.done
    do
      a_worker.be_not_done
      done := true
      print ("evil supervisor: I am done.")
    end
end

class NICE_SUPERVISOR
```

```
feature
  done: BOOLEAN
    -- Did I convince a worker?

  convince (a_worker: separate WORKER)
      -- Convince 'a_worker' that he is done as soon as he thinks that he is not done.
    require
      not a_worker.done
    do
      a_worker.be_done
      done := true
      print ("nice supervisor: I am done.")
    end
end

class WORKER

create
  make

feature
  make
      -- Create the worker and make him not done.
    do
      done := false
    ensure
      not_done: not done
    end

  done: BOOLEAN
    -- Do I think that I am done with my task?

  be_not_done
      -- Make me realize that I am not done.
    do
      print("worker: I am not done.")
      done := false
    end

  be_done
      -- Make me realize that I am done.
    do
      print("worker: I am done.")
      done := true
    end
end
```

2.2 Task

Write down one possible output of the program. Does this system terminate (i.e. all processors finish their tasks)?

2.3 Solution

The system terminates. One of the possible outputs is:

1. boss: I am about to ask the supervisors to do their job.
2. nice supervisor: I am done.
3. worker: I am done.
4. evil supervisor: I am done.
5. worker: I am not done.
6. nice supervisor: I am done.
7. worker: I am done.
8. evil supervisor: I am done.
9. worker: I am not done.
10. boss: The supervisors are done.
11. boss: I am done.

Variations of the above output are given by the fact that a worker can print its message before the supervisor and the other way around. The remaining orderings are predefined by the program.

3 SCOOP Type Combinators

3.1 Background

Have a look at the code snippets shown in listing 2.

Listing 2: type system

```
class A
class B inherits from A
```

```
a: A
b: separate B
c, d: separate A

r (x: separate T)
  do
    1. a := x.b1
    2. b := x.a1
    3. c := x.d1
    4. d := x.c1
  end

s (x: T)
  do
    5. a := x.b1
    6. b := x.d1
```

```
  end

p (x: separate T)
  do
    7. x.f(a)
    8. x.g(c)
  end

q (x: T)
  do
    9. x.f(a)
    10. x.g(d)
    11. x.g(a)
  end

class T
  feature
    a1: separate A
    b1: separate B
    c1: A
    d1: B

  feature
    f(z: A)
      do
        z.f1
      end

  feature
    g(z: separate A)
      do
        z.f1
      end
end
```

3.2 Task

Are the above numbered statements correct and why/why not?

3.3 Solution

1. No, it is incorrect because a is non-separate and both x and b1 are separate.

2. No, it is incorrect because B inherits from A.

3. Yes, it is correct because x is separate and d1 is non-separate, so c should be separate, and since B $\subseteq$ A, the assignment to c will work.

4. Yes, it is correct because x is separate and c1 is non-separate, so the type of the return value must be separate.

5. No, this is incorrect because a should be separate.

6. Yes, this is fine because b should be non-separate, but since non-separate $\subseteq$ separate, the compiler won't complain.
7. This is not possible because the actual argument should be non-separate with respect to the target, not the client.
8. This is correct because c is separate as it should be.
9. This is correct because everything is non-separate.
10. This is correct because d is separate as it should be.
11. This is correct. The actual argument can be non-separate or separate in case the formal argument is separate.

4 Baboon Crossing

4.1 Background

This task is adapted from Downey [1] and Tanenbaum [2]. There is a deep canyon somewhere in Kruger National Park, South Africa, and a single rope that spans the canyon. Baboons can cross the canyon by swinging hand-over-hand on the rope, but if two baboons going in opposite directions meet in the middle, they will fight and drop to their deaths. Furthermore, the rope is only strong enough to hold n baboons. If there are more baboons on the rope at the same time, it will break.

4.2 Task

Design and implement a SCOOP synchronization scheme with the following properties:

- Once a baboon has begun to cross, it is guaranteed to get to the other side without running into a baboon going the other way.
- There are never more than n baboons on the rope.
- A continuing stream of baboons crossing in one direction should not bar baboons going the other way indefinitely (no starvation).

4.3 Solution

A solution can be found in the SCOOP example directory, which is part of the EiffelStudio installation.

References

[1] Allen B. Downey. The Little Book of Semaphores Second Edition. Green Tea Press, 2005.

[2] Andrew S. Tanenbaum. Modern Operating Systems (2nd Edition). Prentice Hall, 2001.