Assignment 4: Semaphores

ETH Zurich

1 Semaphores

1.1 Background

Consider the following synchronisation problem. A new restaurant in town (with the peculiar name of *SemaFood*) features a large communal pot of soup from which diners can help themselves at any point. At full capacity, the pot holds M servings of soup. When a diner wants to eat, they help themselves to a serving from the pot, unless it is empty. In the case that the pot is empty, the diner wakes up the cook and then waits until the cook has refilled the pot before taking a serving.

Each diner process runs the following (unsynchronised) code:

```
while true loop
  -- your code here
  getSoupFromPot()
  -- your code here
  eat()
end
```

and *exactly one* cook process runs the following (unsynchronised) code:

```
while true loop
  -- your code here
  replenishSoup()
  -- your code here
end
```

1.2 Tasks

1. Add (pseudo)code to the diner and cook processes that—through the use of *semaphores*—establishes the following synchronisation constraints, without the possibility of deadlock:

   - at most one diner can have access to the pot at a time;
   - diners cannot invoke *getSoupFromPot* if the pot is empty;
   - the cook can *only* invoke *replenishSoup* once the pot is empty.

   Remember to declare the semaphores and shared resource(s) that you will use, and to initialise them to appropriate values.

2. Explain why your solution is deadlock-free.

3. What assumptions must be made about the semaphores for your solution to be starvation-free too?

1.3 Solution

The tasks and solutions are based on the *dining savages problem* as described in *The Little Book of Semaphores*[1].

1. Following the hint in the task, we declare an integer variable, a mutex, and two condition semaphores:

```
servings := 0
mutex.count := 1 -- protects access to servings
emptyPot.count := 0 -- indicates that pot is empty
fullPot.count := 0 -- indicates that pot is full
```

   Diner processes:

```
while true loop
mutex.down
if servings == 0 then
emptyPot.up
fullPot.down
servings := M
end
servings := servings - 1
getSoupFromPot()
mutex.up
eat()
end
```

   Cook process:

```
while true loop
emptyPot.down
replenishSoup()
fullPot.up
end
```

2. The only opportunity for deadlock comes when a diner holding *mutex* blocks on *fullPot*. While the diner waits, the other diners block on *mutex*. Eventually the cook will run and signal *fullPot*, which allows the waiting diner to resume. That diner will eventually release *mutex*, allowing a blocked diner to unblock.

3. The semaphores would need to be implemented as *strong semaphores*, i.e. semaphores that keep blocked processes in a FIFO queue.

2 Interleaving with Semaphores

2.1 Background

This task is also taken from *Foundations of Multithreaded, Parallel, and Distributed Programming* [1].

[1] http://greenteapress.com/semaphores/

2.2 Task

Given the following processes and code, give the possible outputs of the interleavings:

s.count := 0 r.count := 1 x := 0		
P_1	P_2	P_3
s.down	r.down	r.down
r.down	x := x * (x + 1)	x := x + 2
x := x * 2	r.up	r.up
r.up	s.up	

2.3 Solution

The possible execution orders are
P_2, P_1, P_3 or P_2, P_3, P_1 or P_3, P_2, P_1.
The corresponding outputs are: 2, 4, 12.

3 Unisex bathroom

3.1 Background

This task has been adapted from *Foundations of Multithreaded, Parallel, and Distributed Programming* [1]. In an office there is a unisex bathroom with n toilets. The bathroom is open to both men and women, but it cannot be used by men and women at the same time.

3.2 Task

1. Develop a Java program that simulates the above scenario using semaphores from the Java concurrency library. Your solution should be deadlock free, but it does not have to be starvation free.

2. Justify why your solution is deadlock free.

3.3 Solution

The program and the justifications can be found in the source that comes with this solution.

References

[1] Gregory R. Andrews. Foundations of Multithreaded, Parallel, and Distributed Programming. Addison Wesley, 1999.