Concurrency:
Multi-core Programming and Data Processing

Assignment 03

Professor: Pascal Felber
Assistant: Isabelly Rocha

April 2, 2020

Submission: Java code and a pdf file containing your answers.

1 Dining Savages

The dining savages problem considers the following situation: A tribe of savages eats meals from a single large pot that has a capacity of N portions. When a savage eats, he takes a portion from the pot if the pot still has at least one available. If the pot is empty then only one savage orders the cook to refill the pot and waits until this is full again. Other savages should not try to eat or refill the pot once the refilling request was sent to the cook by the first savage that has noticed that the pot was empty. The cook does only full refills of the pot (N portions). To summarize the constraints: the savages can't take a portion from the pot if the pot is empty and the cook can't refill the pot unless the pot is empty.

1.1 Implementation

Write a program (Ex1Savages1.java) to simulate the behavior of the savages and the cook, where each one of them is a thread and the pot is a shared resource, respecting the constraints above. Consider that each savage wants to eat only one meal, but the number of savages should be higher than the capacity of the pot, so the pot will need refillings. Try to write a short description of the reasoning you used as an informal proof for the synchronization constraints required.

1.2 Fair Implementation

Now, consider that the savages are always hungry (i.e., the threads are continuously looping trying to take another portion from the pot after they eat). Modify your program (Ex1Savages2.java) such that every savage will eventually eat from the pot. The number of savages is fixed, and it is known by each one of them. As before, write a short description of the reasoning you used, as an informal proof for the no starvation constraint required.

Notes:

- You are allowed to use volatile / synchronized / ReentrantLock constructs / classes in your solutions.
- You are **not allowed** to use Atomic classes (e.g., AtomicInteger) or Synchronizers (e.g., CountDownLatch, Semaphore and Phaser).
- You **should not** rely on the fairness implementation provided by the Java classes (e.g., using a Semaphore with fairness support initialized from the constructor).
- You are **not allowed** to use the Object.wait()-notify() / Lock.await()-signal() mechanisms presented in class.
- You are **not allowed** to use the Thread.interrupt() / Thread.sleep()

2 Dining Philosophers

A collection of N Philosophers sits at a round table, where N forks are placed on the table, one between each pair of adjacent philosophers. No philosopher can eat unless he has two forks and he can only use the two forks separating him from his two neighbors. Obviously, adjacent philosophers cannot eat at the same time. Each philosopher alternately eats and sleeps, waiting when necessary for the requisite forks before eating.

Your task is to write code simulating the dining philosophers so that no philosopher starves. An obvious protocol that each philosopher might follow is:

```
while true do
    grab left fork;
    grab right fork;
    eat;
    release left fork;
    release right fork;
    sleep;
end
```

For this implementation you are free to use any Java synchronization constructs. Try to think of a solution which attempts to address potential deadlocks that might occur.