National University of Singapore
School of Computing

CS2106 **Tutorial 4** Semester 1 11/12

This tutorial contains a $\star$ question. The $\star$ questions (i) are challenging, (ii) are beyond the scope of CS2106, and (iii) will be not discussed during tutorial. Interested students may however email the solutions to the lecturer. Students who solve three $\star$ questions correctly will earn a coffee and enter CS2106 Hall of Fame.

1. This question concerns the solution to the dining philosophers problem from the textbook (Figure 2-46).

   Does this solution ensure that no philosopher will starve? If yes, argue why it is so. Otherwise, provide a sequence of executation that will starve a philosopher.

2. Figure 2-45 shows an incorrect solution to the dining philosophers problem that leads to deadlock. In this solution, all philosophers are left-handed – they try to pick up the left fork first when they want to eat.

   Suppose there is one philosopher at the table that is right-handed and always try to pick up the right fork first (i.e., swap the order of `take_fork(i)` and `take_fork((i+1)%N)`).

   Will deadlock still occur? Explain.

3. **The Dining Savages problem (Adapted from CS2106 Final 2009/10 Semester 2)**

   A tribe has N savages and a cook. The savages eat communcal dinner from a large pot that can hold M servings of stewed meat. When a savage wants to eat, he helps himself from the pot with a single serving, unless it is empty. If the pot is empty, the savage wakes up the cook and then waits until the cook has refilled the pot. When woken up, the cook refills the pot with M servings of stewed meat, and goes back to sleep. The behavior of these characters repeats infinitely.

   Sketch the pseudocode for the savages and the cook. Use semaphores to synchronize between the actions of the savages and the cook. Note that the only allowed operations on semaphores are initialization (the "=" operator) and the operations up() and down().

4. ($\star$) Section 2.3.9 of the textbook describes an abstraction called *barrier*. A barrier is associated with a set of N processes. The barrier is locked and it does not let any process through until all N processes are at the barrier. In other words, a process will be blocked at the barrier, until all N processes are blocked. When all N processes are blocked, the barrier will unlock and all N processes can continue execution.

   A *repeatable barrier* is a barrier that will lock itself after all the processes pass through the barrier. Thus, a repeatable barrier can be used in a loop.

   Implement a repeatable barrier using semaphores.