Operating Systems **CSI3131** ***Lab 3*** *Winter 2011*

Semaphores – Simulating a ferry

Objective
To gain experience with Java semaphores in creating a simulation of a ferry that services automobiles and an ambulance.

Description

Please read the lab document completely before starting.

Complete Java program provided, Lab3.java that simulates the operation of a ferry.

The ferry offers services between two ports, port 0 and port 1. Ten automobiles travel between these two ports, as well as an ambulance. Each automobile arrives at a port, boards the ferry (when it arrives at the port), crosses to the other port, disembarks the ferry, travels around for a bit, and then comes to the port to cross again.

The ambulance is a vehicle (the automobiles and ambulance are both vehicles) that functions like the automobile with an exception. When an ambulance boards the ferry, the ferry leaves immediately without waiting to be full.

The ferry travels between the two ports (0 and 1): when it arrives at a port, vehicles on board disembark and any waiting vehicles board. When the ferry is full, or an ambulance boards, it leaves for the other port.

The objective of the lab is to simulate the ferry and the vehicles traveling between the ports. See the template, Lab3.java, that accompanies this document.

You will be modifying, with semaphores, the three classes, Auto, Ambulance, and Ferry, to simulate the automobiles, the ambulance and the ferry respectively. Your simulation must meet the following guidelines:

1. The capacity of the ferry is 5 vehicles (5 automobiles or 4 automobiles and one ambulance).
2. A vehicle at port "p" boards the ferry if
   a. The ferry is at the same port p,
   b. All vehicles that arrived with the ferry on board have all disembarked the ferry,
   c. There still exists room on the ferry for the vehicle.
3. A vehicle at port « p » must wait
   a. If the ferry is not at the same port « p »,
   b. Vehicles are disembarking from the ferry,
   c. The ferry is full and is ready to leave.
4. If an ambulance boards the ferry, the ferry leaves immediately.
5. If the ferry is not full and with no ambulance on board, it waits for other automobiles to arrive.
6. A vehicle can disembark only at the arrival at the next port.
7. Treat the output statements (println) as if they are actions. For example, it is not necessary to program a method board() that is simulated by "System.out.println("Auto " + id_auto + " boards the ferry at port "+port);".

Your task is to:

1. Modify all classes for the automobiles, ambulance and ferry.
2. Use Java semaphores (see the description below of Semaphore – the Class for creating semaphores) to control the access to the ferry and for synchronization of the threads.
3. Note that the template provide the code that does the following:
    a. Creates a Java thread for executing the ferry code,
    b. Creates 10 Java threads, each thread executing the automobile code, with an identifier that varies between 0 and 9.
    c. Creates a Java thread that executes the ambulance code,
    d. Runs the simulation for 10 ferry crossings, with automobiles 0 to 6 and the ambulance initially at port 0 and automobiles 7 to 9 initially at port 1.
    e. Once the ferry has made 10 crossings, the ferry thread shall terminate. The main thread (executing main()) with then interrupt (cancel) the automobile threads and the ambulance thread with the interrupt() method.
4. A sample output is provided (see the file log). Note that output varies from execution to execution.

The Java semaphore

- Java provides a class Semaphore (that is part of the package java.util.concurrent). This class offers the services of a counting semaphore with all characteristics of the semaphores in class. In addition:
    - It is possible to specify the use of FIFO for use with the semaphore queue. With FIFO, no starvation is possible. But the use of a queue without FIFO is more efficient.
    - The semaphore value in the Semaphore class works a bit differently than what was presented in class. In Java, the semaphore value represents a number of available permits (see acquire() and release() for how this number is manipulated).
    - The number of permits can be initialized to a negative value. This means that a number of “signals” (calls to release) is required to increase the number of permits to a positive value that will prevent threads from blocking on the semaphore.
- Constructors :
    - `Semaphore(int permits)`
    - `Semaphore(int permits, boolean fair)`
    - The parameter `permits` defines the initial value of the semaphore, i.e., the initial number of permits.
    - The parameter `fair` controls the semaphore queue discipline. If faire is true,, the FIFO discipline is used.
    - The first constructor gives the value false to the faire parameter, that is, does not use the FIFO discipline with the semaphore queue.
- The method `release()`
    - This method plays the same role as the function `signal()` studied in class.
    - It increments the number of permits in the semaphore (i.e. its value). Note that the number of semaphores is never negative unless initialized to a negative value (negative values are only temporary). Also not that this way of operating resembles more the spinlocks rather than the blocking semaphores studied in class.

- The method `acquire()`
    - This method plays the same role as `wait()` studied in class.
    - It checks the number of available permits in the semaphore.
    - If a permit is available (semaphore value > 0), the number of permits is decremented and the method returns; the thread can continue executing.
    - If no permit is available (semaphore value <= 0), the thread blocks and is placed in the semaphore queue. When a permit does become available and threads are in the queue, a thread in the queue is unblocked to consume the permit.
    - This method can be interrupted when a thread is interrupted, that is, it throws the exception `InterruptedException`. It is necessary to place the call to the method in a `try/catch` structure as follows:

      ```
      try{sem.acquire();} catch (InterruptedException e) { }
      ```

      If you execute the structure in a loop (such as the one in your lab exercise), add the break; statement as shown below, to exit the loop.

      ```
      try{sem.acquire();} catch (InterruptedException e) { break; }
      ```
- The method `acquireUninterruptibly()`
    - This method gives the same functionality as `acquire()`, but does not throw an exception if the thread is interrupted. Thus, it is not necessary to enclose it in a `try/catch` structure. Use it if the thread containing the call to this method is not to be interrupted or you wish the thread to acquire a permit before terminating.
- Other methods : The Class `Semaphore` offers the following methods. A brief description is provided here. For more detail, consult Java documentation.
    - `tryAcquire()` – If a permit is available, acquire the permit and return `true`; otherwise return `false`. This method does not bock the calling thread. A second version of this method exists to allow the definition of a time limit to wait for a permit before returning.
    - There exists other versions of `acquire(), acquireUninterruptibly(), tryAcquire()`, and `release()` with a parameter to define a number of permits. These versions provide the means to acquire or release more than one permit.
    - `availablePermits()` – Returns the number of permits available in the semaphore. Can be useful for debugging.
    - `drainPermits()` and `reducePermits()` reduce the number of permits in the semaphore. These methods differ from `acquire()` in that they do not block the calling thread if the number of required permits are not available.
    - `isFair()` – Returns `true` if FIFO is being used.
    - `hasQueuedThreads(), getQueueLength(), getQueuedThreads()` are methods used to obtain information on the semaphore queue.