Lab/Project 3
Multi-threaded Programming in Java

CPS 310: Operating Systems

This project asks you to develop some thread-safe Java classes, and Java programs to test them. You may use any Java development environment and any operating system of your choice. Correct programs should work on any Java platform.

As always, think before you code. Strive for a simple and elegant solution. If the specification below is incomplete in any way then you are free to resolve the ambiguity as you see fit.

Problem 1. EventBarrier

Create a class to implement *EventBarrier*, which allows a group of threads to wait for an event and respond to it in a synchronized fashion. We may consider the threads that request notification of an event as "consumers" of the event, and threads that generate the event as "producers" of the event. *EventBarrier* is a simple event handling scheme inspired by the foundational synchronization objects in Microsoft Windows.

An EventBarrier represents a sequence of event occurrences through time. The consumer threads *arrive* at the barrier to wait for the next event. A producer thread may *raise* the event, i.e., signal that the next event has occurred: all the consumers wake up, respond to the event, and then notify the producer that their handling of the event is *complete*. The producer and consumers block (in *raise* and *complete*) until all consumers have completed their handling of the previous event. We say that the event is *in progress* while consumers are handling it. The three methods *arrive*, *raise*, and *complete* take no arguments.

EventBarrier requires no external synchronization, and it keeps an internal state to "remember" that an event is in progress. If a new consumer thread arrives at the barrier while an event is in progress, *arrive* returns immediately without blocking, and the new consumer may respond to the event in progress.

EventBarrier is like a drawbridge gate at a castle. Weary traveling minstrels *arrive* outside the gate of the castle for the drawbridge to come down. The gatekeeper lowers the bridge and calls *raise* on the gate to wake up the waiting minstrels. The minstrels respond to the event by crossing the drawbridge to enter the castle through the gate. After entering they invoke *complete* to inform the gatekeeper that they have crossed. When all minstrels have completed the EventBarrier reverts to the unsignaled state and the *raise* call returns: the gate is closed. EventBarrier enables this synchronization in a way that lets any number of minstrels enter the castle while the gate is open, and prevents the gatekeeper from closing the gate while minstrels are still crossing the bridge.

Here is the interface for *EventBarrier:*

***void** arrive()* – Arrive at the barrier and wait until an event is signaled. Return immediately if already in the signaled state. (Called by consumer thread.)

***void** raise()* -- Signal the event and block until all threads that wait for this event have responded. The EventBarrier reverts to the unsignaled state before *raise* returns. (Called by producer thread.)

***void** complete()* -- Indicate that the calling thread has finished responding to a signaled event, and block until all other threads that handle this event have also completed. (Called by a consumer thread.)

***int** waiters()* -- Return a count of threads that are waiting for the event or that have not yet responded to it.

Test your *EventBarrier* implementation in whatever way you think best. Be sure your implementation correctly handles threads that call *arrive* while the *EventBarrier* object is in the signaled state; all participating threads must wait until the late arrival has responded to the event and called *complete.*

You may assume that no awakened consumer calls *arrive* again until all waiters for a given event have completed and the *EventBarrier* has reverted to the unsignaled state. This is an assumption about the program that uses *EventBarrier*, so be careful that your test programs behave this way.

Problem 2. Elevator

Implement an elevator controller for a building with *F* floors. You may use (or modify) your *EventBarrier* class to implement the elevator controller if you choose. The elevator controller is split between two classes, *Elevator* and *Building.* There is a thread for each rider and a thread for each elevator. The elevator and rider threads invoke the methods of the Elevator and Building classes and synchronize through these classes.

The rider threads use methods the *Building* rider interface to call elevators: *CallUp*, *CallDown.* Riders use the *Elevator* rider interface to ride the elevators: *Enter, Exit, RequestFloor*.

These methods behave in a way that is familiar in the real world. *CallUp* and *CallDown* request an elevator for travel in a particular direction, then wait for a called elevator to arrive for travel in the requested direction.

Each elevator runs as a separate thread looping within the *Elevator* class, calling internal elevator control methods *(VisitFloor, OpenDoors*, *CloseDoors)* in the obvious sequence to serve riders in an orderly fashion.

Part 1. Implement the elevator controller for a single elevator, including the *Building* and *Elevator* methods defined here and any new methods needed by your implementation. Your solution should avoid rider starvation (all riders are served “eventually”) as well as all obvious races, e.g., doors opening while the elevator is in transit, riders entering and exiting while the doors are closed, etc. In Part 1 you may assume that the elevator is of

unbounded size, i.e., it can hold all arriving riders. When your elevator visits a floor (*VisitFloor*), it should wait for all exiting riders to exit and all entering riders to board.

Include a test program that demonstrates the operation of your elevator for multiple riders. Your program should be ‘‘well-behaved’’ in the following sense: any rider that calls the elevator for the up or down direction (using *CallUp* or *CallDown)* should wait until the elevator arrives, then get on it (*Enter*), request a floor (*RequestFloor*), then get off (*Exit*) when *RequestFloor* returns, signifying arrival at the correct floor.

Part 2. Extend your solution in Part 1 to handle elevators that can hold only *N* riders at a time, where *N* is a compile-time constant. In this case, your *Enter* primitive should return a failure status if a rider attempts to get on while the elevator is full. The rider should respond by calling another elevator.

Part 3. Extend your solution to handle *E* elevators. The elevators should coordinate through the *Building* class to serve requests efficiently. For example, at most one elevator should go to pick up each set of passengers, and it should be carefully chosen. This is a difficult resource scheduling problem.

For extra credit, extend your elevators to handle badly behaved riders (e.g., practical jokers who call the elevator and don’t wait for it, who get on but do not request a floor, or who do not exit at the requested floor).