ASSIGNMENT 2

u21516261

OCTOBER 19, 2025

BYRON PILLAY

Task 1

Which lock was chosen and Why?

The Reentrant Lock was chosen. This lock was chosen because:

- It provides finer control than using synchronised,
- It has support for Conditions which have necessary features for a reusable barrier like await() and signalAll(),
- It avoids race conditions,
- The lock is reusable, and
- It allows the same thread to acquire the lock multiple times for better scalability.

The Reentrant Lock is more suitable than a Semaphore Lock or Read-Write Locks because the Semaphore is not designed to manage barriers in which all threads wait and a Read-Write Lock is not suitable for synchronization.

Overall, the reusable and condition-based features of a Reentrant lock is desirable in a cycle barrier system like the Task 1.

Task 2

Which Lock was chosen and Why?

Again the Reentrant Lock was chosen. Task 2 requires the Producers to sleep when the buffer is full and consumers to sleep when the buffer is empty. The Reentrant lock allows for both conditions to be monitored.

Furthermore, the Reentrant Lock allows the waits to be interrupted which means there is no redundant waiting time when the buffer is full or empty.

Coarse Grained Implementations vs. Fine Grained Implementations

The Coarse Grained Implementation uses just 1 Reentrant Lock which protects access to the head, tail and count member variables, while the Fine Grained Implementation requires 2 locks and these locks the head and tail independently.

The fine grained implementation allows producers and consumers to work in parallel when the queue is neither empty nor full. Meanwhile, the coarse grained implementation only allows the producers to run when the queue is not full and the consumer is only allowed to run when the queue is not empty.

Which is Better for the Pizzeria?

The Fine Grained Implementation makes sense in the context of a Pizzaria since the cooks (or producers) will probably not wait to produce an order of pizzas until the delivery man (consumer) returns.

Task 3

A Method to Use in Place of the Poison Pill

Instead of a Posion Pill, an Atomic Integer can be used to track how many products are in the queue. When a producer produces a pizza (via enqueue), the producer will increment the Atomic Integer, and the consumer will check if the Atomic Integer is 0 before trying to take a pizza (dequeue).

How Do Producers Avoid Overwriting and How do Consumers Avoid an Empty Queue?

The safety comes from BlockingQueue<Integer>.

The producers avoid overwriting because pizzas are added to the queue using queue.put(pizza). The bounded BlockingQueue blocks the producer when the queue is full, and only allows the producer to produce when there is space.

Meanwhile, consumers do not read from an empty queue because the BlockingQueue method queue.take() blocks when the queue is empty.