ASSIGNMENT 2 REPORT

Task 1: Barrier Synchronization

Five friends are preparing different parts of a pizza (dough, kneading, sauce, cheese and toppings).
All must complement their part before the pizza can be baked. Once all are done, they start the next pizza together.

Lock Chosen: ReentrantLock

Reason for this choice:

- ReentrantLock ensures mutual exclusion, allowing only one thread at a time to update the shared variables (completed → counts how many friends have reached the barrier and current_pizza → tracks which pizza (round) is being baked)
- It also provides Condition variables which allow threads to wait (await()) and be signalled (signalAll()) when the barrier condition is met.
- This allows control over thread synchronization compared to synchronized blocks.

Usage of Condition Variable Explanation

- The Condition object makes threads wait until all friends reach the barrier.
- When all threads arrive (completed == friends), the barrier:
    1. Run the task.
    2. Reset completed to 0 for reuse.
    3. Increments current_pizza to start the new 'round' of pizza.
    4. Signals all waiting threads to proceed.

Task 2: Concurrent Data Structure

The pizzeria counter has cashiers (producers) and customers (consumers) who access a shared order queue.

We ensure that:

- ✓ Multiple threads can enqueue and dequeue safely.
- ✓ No overwriting or reading from an empty queue occurs

Lock Chosen: ReentrantLock

Reason for this choice:

- It is for thread safety and access to the Condition variables (notFull, notEmpty) coordination.

Coarse-Grained Queue

- Uses one lock for both enqueue and dequeue operations
- Conditions:
  - notFull ➔ blocks producers when the queue is full.
  - notEmpty ➔ blocks consumers when the queue is empty.

Enqueue steps:

1. Wait while queue is full (notFull.await())
2. Add item to queue
3. Signal notEmpty (a consumer can now take)

Dequeue steps:

1. Wait while queue is empty (notEmpty.await())
2. Remove item
3. Signal notFull (a producer can now add.)

Drawback:

Only one thread (producer or consumer) can access the queue at a time. It results in low concurrency.

Fine-Grained Queue

- Uses two separate locks
  - enqueueLock ➔ for inserting
  - dequeueLock ➔ for removing
- Each has its own condition (notFull and notEmpty)
- Also uses an AtomicInteger size to track the queue size safely across threads.

Effects:

- A producer can enqueue while a consumer is dequeuing if space and items exist.
- Improves throughput and reduces blocking time.

	Coarse-Grained	Fine-Grained
Lock Structure	One lock for all	Separate locks for enqueue/dequeue
Concurrency	Low	High
Blocking	More frequent	Less Frequent
Complexity	Simple	More complex

Best Choice for the Pizzeria

- The fine-grained queue is better because multiple cashiers (producers) can add orders while (customers) consumers collect pizzas simultaneously.
- This reduces waiting times and increases efficiency
- It mirrors a real pizzeria: while one staff member takes an order, another can deliver a pizza without disturbance.

Task 3: Producer-Consumer

- The pizzeria acts as the producer, generating pizzas (orders) while delivery drivers act as consumers, collecting and delivering pizzas.
- Shared structure: BlockingQueue<Integer> manages pizzas between producers and consumers.
- Producer: Creates pizzas and places them in the queue.
- Consumer: Takes pizzas out of the queue for delivery.
- Termination: When production is complete, a "poison pill" (-1) is placed in the queue to signal consumers to stop waiting.

In the code:

- queue.put(pizza) ➔ Blocks if the queue is full (prevents overwriting)
- queue.take() ➔ Blocks if the queue is empty (prevents reading invalid data)

Alternative to the Poison Pill

Instead of using a poison pill, we could use a shared flag:

```
AtomicBoolean running = new AtomicBoolean(true);
```

- Producers set running to false when done.
- Consumers periodically check:
- if (!running.get() && queue.isEmpty()) break;

Let us compare

- The poison pill is simpler and works well for small systems.
- The flag method is cleaner for complex systems with many producers/consumers.

Ensuring Safe Queue Operations

- Producers call put() to block when the queue is full → prevents overwriting.
- Consumers call take() to block when the queue is empty → prevents reading nothing.
- This synchronization ensures no race conditions or missed pizzas.