**Fixed Thread Pool Design** **Why use BlockingQueue?**  
It handles concurrent access and supports blocking take() when the queue is empty.

 **What if you call submit() after shutdown()?**  
Throws IllegalStateException.

 **Why use interrupt() during shutdown?**  
To unblock threads waiting on taskQueue.take().

 **How is this different from Java’s ExecutorService?**  
This is a simplified custom implementation; ExecutorService offers richer lifecycle control, task result tracking (Future), and more robust shutdown options (awaitTermination).

### **Do we need to use wait() and notify() ✅ Short Answer:**

**No, you do not need to use wait() and notify()** if you're using a **BlockingQueue** like LinkedBlockingQueue — it already handles the waiting and notification internally in a thread-safe manner.

**✅ Why wait()/notify() Aren’t Needed Here**

In your custom fixed thread pool, you're using:

Runnable task = taskQueue.take();

This method:

* Automatically **blocks** if the queue is empty (like wait()).
* Internally **notifies** waiting threads when new tasks are added (like notify()).

So, the coordination between producer (submit()) and consumer (worker.run()) is automatically handled by the blocking queue implementation — **no manual synchronized, wait(), or notify() needed.**

**✅ When Would You Need wait() and notify()?**

You would need them if:

* You were **building your own queue** from scratch (e.g., using List or Array).
* You wanted **fine-grained control** over synchronization between threads.
* You’re managing **custom thread-safe data structures** where Java’s concurrent collections aren't used.

Using BlockingQueue like LinkedBlockingQueue is preferred because it's more readable, robust, and less error-prone. If we wrote our own task queue using synchronized, wait(), and notify(), it would require careful handling to avoid bugs like missed notifications or deadlocks.