I will potentially focus on using threads in few of my coding examples…

A good example would be currency exchange application.

Initiating currency exchange by producer as an opportunity for arbitrage.
The shared object / resource would be the denomination offered by the thread instance (taken to be representation of the conversion identified for opportunity such as \$=>£=>\$)

The consumer might block off the transaction if they are exceeding a certain amount of arbitrage… Or there might be insufficient intermediary funds available along the chain.

I am unsure how the consumer will dictate availability of intermediary denomination (£). I am unsure if this is a concern at all.
The consumer can perhaps perform producerThread.interrupt() to momentarily freeze any further action or producerThread.wait(). At this point, the Producer has committed to the transaction.
The consumer might also perform a notify() or notifyAll() to any threads dealing with large denomination of £ in order to gain currency.
This might weaken or strengthen the £ prospectively.

Alternatively the consumer might await certain amount of profit before completing transaction (i.e. if multiple threads

with instances are participating concurrently). The consumer might identify several high volume currency transactions (with minimal risk) and prioritise these threads first….

I am unsure if this can be classified as deadlock since there would be threads locking other threads… However it will not occur indefinitely.

And it just does not seem possible they can all lock each other out..

Also, it seems unethical to have such a mechanism in place in order to support fair play. I believe use locks in Java will support this area VS synchronise.

Another possible situation is starvation in which threads with lower profits / less volume of currencies are monopolising hardware resources unexpectedly. For instance, it can be consisting of large chain (£=>$=>€=>¥=>€=>$=>£)
Subsequently they are getting a lock of the other threads which are competing for larger arbitrage.

Alternatively, the threads operating on larger compute (realising less financial gain) are starving threads which are on less compute BUT making more money (i.e. their conversion chain might be shorter such as £=>$=>€=>$=>£)

Based on research, it appears that large CPU utilised threads get precedence.
There was also mention of utilizing CyclicBarriers which ensures certain threads reach a certain point before performing action. This was as part of the lock. It might be that a certain threshold of currency is exchanged.

The whole situation can be a typical livelock (envisaged from consumer perspective) in which buyer waits for the broker to declare availability of the exchanged currency in order to perform transaction.

Whereas the broker is waiting for the buyer to relinquish denomination for which he/she intends to convert.

But it has to be realised, these are typically utilised in high frequency trading and there is a negligible delay for transaction to occur. So any thread would expect immediate transaction also given the fluctuating exchange rates.

In terms of fluctuation the conversion rates, this in itself would require a major consideration.

Also since I can not emulate a real world without several influences affecting rates, perhaps it would be good to create logic based on a closed system of finite currency and decision making has to be made

on availability of denominations. This will permit utilization of certain locks or even deadlocks without overcomplicating the system.