Synchronized Methods vs Synchronized Statements

We used synchronized methods instead of synchronized statements due to the below reasons.

First, the operations within the method are closely related, and partial synchronization (using synchronized statements on just a part of the method) might not provide the needed consistency. Using synchronized methods ensured that the entire sequence of operations within the method was atomic, maintaining consistency across shared resources or data structures accessed or modified by both the all the threads. Additionally, this reduces the chances of future modifications inadvertently breaking the synchronization by missing necessary code blocks within the method.

Also, by synchronizing entire methods, it is more likely to acquire locks in a consistent order, especially where method calls are structured in a clear hierarchy. This can reduce the risk of deadlocks, which are more likely to occur when multiple synchronized statements involve acquiring multiple locks in different orders across threads. It is also important to note that synchronized methods automatically manage the acquisition and release of locks on the object (for instance methods) or the class (for static methods) they belong to. This ensures that the method will be executed by only one thread at a time for the locked object or class. Automatic lock management is crucial for providing consistency, as it eliminates manual lock handling errors, ensuring that locks are properly acquired and released, thereby preventing deadlocks and ensuring thread safety.

Semaphores vs Busy-Waiting

In high-load scenarios, busy-waiting may be particularly inefficient because there is increased context switching, instead where other threads could utilize these CPU cycles for processing. In other words, semaphore approach is more efficient than busy-waiting because it frees up CPU resources while a thread is waiting. Other threads or processes can use these resources to perform useful work. When a condition is met (e.g., a buffer becomes full or empty), only then is the waiting thread awakened, reducing unnecessary CPU utilization and context switches.

It should be noted that threads may experience a slight delay when waking up compared to the immediate response of busy-waiting. This is perhaps part of the cause of our output running times being different compared to our expectations, especially as this is not a high-load scenario as highlighted above, instead with < 100 transactions.

This may be seen when comparing busy-waiting running times:

Terminating client receiving thread - Running time 330 milliseconds

Terminating server thread1 - Running time 344 milliseconds

Terminating server thread1 - Running time 344 milliseconds

Terminating network thread - Client disconnected Server disconnected

to semaphore running times:

Terminating client sending thread - Running time 227 milliseconds

Terminating client receiving thread - Running time 336 milliseconds

Terminating server thread1 - Running time 351 milliseconds

Terminating server thread1 - Running time 350 milliseconds

Terminating network thread - Client disconnected Server disconnected

2 Threads vs 3 Threads

Introducing a third thread allows for more parallel processing, as seen by the decrease in running times of 3 threads compared to 2. This can reduce the time spent on processing tasks by distributing the workload more evenly among threads. Though not present here, in general if there is an unexpected increase/balance in running time of 3 threads compared to 2, it is because the semaphores must be carefully managed to prevent deadlock and minimize contention. Proper semaphore usage ensures that threads are blocked when necessary and promptly awakened when they can proceed, without unnecessary delays. Also, semaphores must correctly deal with network buffer's increased contention by efficiently scheduling thread access to the shared resources.

Running times for two threads:

Terminating client sending thread - Running time 227 milliseconds

Terminating client receiving thread - Running time 336 milliseconds

Terminating server thread1 - Running time 351 milliseconds

Terminating server thread1 - Running time 350 milliseconds

Terminating network thread - Client disconnected Server disconnected

Running times for three threads:

Terminating client sending thread - Running time 156 milliseconds

Terminating client receiving thread - Running time 272 milliseconds

Terminating server thread - Thread2 Running time 289 milliseconds

Terminating server thread - Thread3 Running time 289 milliseconds

Terminating server thread - Thread1 Running time 289 milliseconds