Chapter 7: Spin Locks and Contention

This chapter introduces the practical challenges of implementing synchronization primitives in a real-world multiprocessor environment, where hardware architecture and contention significantly impact performance. The core topics revolve around **spin locks**, which are a type of lock where a thread waiting to acquire the lock simply keeps checking in a loop until the lock becomes available.

7.1 Welcome to the Real World: The Failure of Theoretical Proofs

In a theoretical, idealized model, synchronization primitives like the Peterson lock can be proven correct to provide mutual exclusion[1111]. However, in practice, these proofs often fail[2]. This is because real-world multiprocessors do not always guarantee

sequential consistency[33].

- **Sequential Consistency:** A memory model where all operations from all threads appear to execute in a single, global, sequential order that respects the program order of each individual thread[4].

- **The Problem:** Modern multiprocessors and their compilers reorder instructions to enhance performance, which can break the assumption of program order among reads and writes to different variables[5]. This can lead to race conditions and incorrect behavior even with algorithms that were theoretically proven to be correct[6666].

7.2 Test-And-Set Locks (TASLock)

The
Test-and-Set (TAS) lock is a simple spin lock that uses the atomic getAndSet() operation to acquire a lock[7].

- **The Issue:** The TAS lock performs poorly on real-world systems because when multiple threads are contending for the lock, they all repeatedly attempt to acquire it, causing a high volume of cache misses[8]. The shared cache line containing the lock's state is constantly being invalidated and updated, leading to significant contention on the memory bus.

7.3 TTASLock: The Test-and-Test-and-Set Lock

The
Test-and-Test-and-Set (TTAS) lock is an improved version of the TAS lock that addresses the performance issue of high contention[999].

- **The Mechanism:** Instead of repeatedly using getAndSet(), the TTAS lock first performs a simple get() read of the lock's state in a loop[10]. The thread only attempts to acquire the lock with

  getAndSet() once the lock appears to be free.
- **The Benefit:** Threads spin on a locally cached copy of the lock's state, only causing a cache invalidation when the lock is released[11]. This significantly reduces the memory bus traffic and improves performance under high contention[12].

7.4 Exponential Backoff

This is another strategy to reduce contention in spin locks[13].

- **The Mechanism:** After a failed attempt to acquire the lock, a thread waits for an exponentially increasing amount of time before trying again.
- **The Benefit:** This approach helps to spread out the threads' lock acquisition attempts, reducing the likelihood of a "thundering herd" problem where all threads contend for the lock at the same time.

7.5 Queue Locks

Queue locks are designed to be more fair and efficient than simple spin locks by explicitly managing a waiting queue of threads[141414].

- **Array-Based Locks:** These locks use a circular array to assign a "ticket" to each thread. A thread waits until its ticket number is the same as the current turn, which is incremented by the lock holder upon release[15].

- **The CLH (Craig, Landin, and Hagersten) Queue Lock:** This is a popular and efficient queue lock. Each thread spins on a separate, locally cached node in a linked list, avoiding contention on a single shared variable[16161616]. When a thread releases the lock, it

simply updates its node, signaling to the next thread in the queue that the lock is free[17].

- **The MCS (Mellor-Crummey and Scott) Queue Lock:** This lock also uses a linked list, but each thread spins on its own node[1818]. This is more efficient because it completely eliminates network contention.

7.6 A Queue Lock with Timeouts

This section deals with extensions to queue locks to handle scenarios where a thread holding a lock may fail, preventing other threads from ever acquiring the lock[1919]. The timeout mechanism allows a waiting thread to give up on acquiring the lock after a certain amount of time, preventing indefinite waiting.

7.7 A Composite Lock

A
Composite Lock combines different locking strategies to optimize for different scenarios[2020]. For example, a lock might use a fast path for low-contention scenarios and a slower, more robust queue lock for high-contention scenarios[2121].

7.8 Hierarchical Locks

Hierarchical locks are designed to improve scalability in large-scale systems[2222].

- **The Mechanism:** Threads are grouped into "clusters," and each cluster has a local lock[2323]. When a thread can't acquire the local lock, a designated thread from that cluster (a "cluster master") competes for a global lock[2424].

- **The Benefit:** This approach reduces contention on the global lock by a factor of the cluster size, as many threads can acquire the lock locally without needing to access the global state[25252525]. An example is the

  Hierarchical CLH Queue Lock[26262626].