Delegated Token Semaphore: A Scalable and Fair Synchronization Mechanism for Multiprocessor Operating Systems

Rahul Patel
Department of Electronics and Communication Engineering
Indian Institute of Information Technology Surat
Surat, Gujarat, India

***Abstract*—Synchronization is a fundamental challenge in multiprocessor operating systems, where multiple threads compete for shared resources. Traditional synchronization primitives such as mutexes, semaphores, spinlocks, and futexes often suffer from limitations including starvation, unfair scheduling, excessive context switching, and high CPU utilization under contention. These issues become increasingly severe as modern systems scale to large numbers of processor cores.**

To address these challenges, this paper proposes the *Delegated Token Semaphore (DTS)*, a novel synchronization mechanism that combines the fairness guarantees of token-passing approaches with the efficiency of delegation-based execution. In DTS, a circulating token enforces orderly and starvation-free access to critical sections, while waiting threads delegate their requests to the current token holder. This delegation enables batch execution of critical-section requests, reducing contention, minimizing context switches, and improving cache locality.

A detailed system design, algorithmic formulation, and theoretical analysis are presented to demonstrate the correctness and fairness properties of DTS. Experimental evaluation on a multiprocessor platform compares DTS against conventional synchronization techniques under varying contention levels. Results show that DTS achieves higher throughput, lower waiting time, and reduced CPU overhead, particularly in high-contention scenarios. These findings indicate that DTS provides a scalable and efficient alternative to existing synchronization primitives for modern multiprocessor operating systems.

***Index Terms*—Synchronization, Multiprocessor Operating Systems, Semaphore, Delegation, Token-Based Synchronization, Fairness, Scalability, Resource Management**

I. INTRODUCTION

Modern computing systems increasingly rely on multiprocessor and multicore architectures to deliver high performance, responsiveness, and energy efficiency. In such systems, multiple threads or processes often execute concurrently while sharing data structures, memory regions, or I/O resources. Ensuring correct and efficient access to these shared resources is the primary objective of synchronization mechanisms in operating systems.

Traditional synchronization primitives such as mutexes, semaphores, spinlocks, and futexes have been widely adopted to enforce mutual exclusion. While these mechanisms are effective in low-contention or small-scale environments, their performance degrades significantly as the number of cores and competing threads increases. Spinlocks waste CPU cycles through busy waiting, mutexes and semaphores incur context-switching overhead, and futex-based approaches may still suffer from unfair scheduling and starvation under heavy contention.

As modern systems scale to dozens or even hundreds of cores, synchronization overheads become a dominant performance bottleneck. In addition to reduced throughput, unfair scheduling and priority inversion may cause some threads to experience excessive waiting times, leading to poor responsiveness and unpredictable system behavior. These challenges highlight the need for synchronization mechanisms that provide both scalability and strong fairness guarantees.

Recent research has proposed scalable locking techniques such as queue-based locks, flat combining, and token-based synchronization. While these approaches address specific limitations of classical primitives, they often trade fairness for throughput, rely on workload-specific assumptions, or introduce significant implementation complexity. Consequently, no single mechanism consistently provides low overhead, fairness, and scalability across diverse contention scenarios.

To address these limitations, this paper introduces the *Delegated Token Semaphore (DTS)*, a novel synchronization mechanism that combines token-passing fairness with delegation-based execution. In DTS, a circulating token enforces orderly and starvation-free access to critical sections, while waiting threads delegate their synchronization requests to the current token holder. This design enables batch execution of critical sections, reduces context switching, and improves cache locality without resorting to busy waiting.

The main contributions of this work are summarized as follows:

- We propose Delegated Token Semaphore (DTS), a hybrid synchronization mechanism that integrates token-based fairness with delegation-based execution.
- We present a detailed system design and algorithmic formulation demonstrating starvation freedom and scalable behavior under contention.
- We evaluate DTS on a multiprocessor platform and compare it against conventional synchronization primitives, showing improvements in throughput, latency, and CPU efficiency.

The remainder of this paper is organized as follows. Section II reviews related work on synchronization mechanisms for multiprocessor systems. Section III presents the system overview and design of the proposed DTS. Section IV describes the algorithmic details and theoretical analysis. Section V discusses the experimental setup and evaluation results. Finally, Section VI concludes the paper and outlines directions for future research.

II. RELATED WORK

Synchronization has been a central topic in operating system design since the early development of concurrent systems. Classical primitives such as semaphores, introduced by Dijkstra, and monitors, proposed by Hoare, provide fundamental mechanisms for enforcing mutual exclusion and coordinating concurrent processes. While these primitives are simple and widely supported, they are prone to issues such as deadlock, priority inversion, and starvation when used in complex or highly contended multiprocessor environments.

Spinlocks are commonly used in low-level systems due to their simplicity and low overhead under short critical sections. However, under high contention or long wait times, spinlocks waste significant CPU cycles through busy waiting. Blocking synchronization mechanisms, including mutexes and POSIX semaphores, avoid busy waiting by suspending threads, but incur context-switching and scheduler overhead, which limits scalability as the number of cores increases.

Futex-based synchronization mechanisms attempt to combine the advantages of user-space spinning and kernel-assisted blocking. By remaining in user space during uncontended execution and invoking the kernel only under contention, futexes reduce overhead in many common cases. Nevertheless, futex-based approaches can still suffer from unfair wake-up behavior and increased latency under heavy contention, particularly when large numbers of threads compete for shared resources.

To address scalability issues, several queue-based locking algorithms such as MCS locks and ticket locks have been proposed. These mechanisms provide stronger fairness guarantees and reduce cache-line contention by organizing waiting threads in a queue. Although queue-based locks improve scalability, they may still experience increased latency due to strict FIFO ordering and may not efficiently handle workloads with highly variable critical-section execution times.

Delegation-based synchronization techniques, such as flat combining, reduce contention by allowing a single combiner thread to execute critical-section operations on behalf of multiple waiting threads. By batching requests, these approaches improve cache locality and throughput under contention. However,

flat combining is primarily designed for concurrent data structures and does not directly provide semaphore-like semantics or explicit fairness guarantees in general synchronization scenarios.

Token-passing mechanisms have been explored as a means to enforce fairness and prevent starvation in distributed and multiprocessor systems. By circulating a token among participants, these approaches ensure orderly access to shared resources. Despite their fairness properties, pure token-based mechanisms may introduce additional latency due to token handoff and may not fully exploit opportunities for batching or delegation under high contention.

The proposed Delegated Token Semaphore (DTS) builds upon insights from both delegation-based and token-based synchronization techniques. Unlike classical semaphores and spinlocks, DTS eliminates busy waiting and enforces starvation-free access through token circulation. In contrast to flat combining, DTS explicitly integrates semaphore semantics with token-based fairness, enabling efficient batch execution while preserving orderly access. This combination addresses key limitations of existing approaches and provides a scalable and fair synchronization primitive for modern multiprocessor operating systems.

III. SYSTEM OVERVIEW

The Delegated Token Semaphore (DTS) is a synchronization mechanism designed for multiprocessor operating systems to provide scalable, fair, and starvation-free access to critical sections. DTS combines token-based access control with delegation-based execution to reduce contention, minimize context switching, and improve cache locality under high concurrency.

The core idea of DTS is to enforce orderly access using a circulating logical token while allowing threads that do not hold the token to delegate their synchronization requests. Instead of competing directly for a shared lock, waiting threads enqueue their requests and suspend execution. The thread holding the token, referred to as the *combiner*, executes its own critical-section operation along with delegated requests in a batch, thereby amortizing synchronization overhead.

The DTS mechanism is composed of four primary components: a Token Manager responsible for maintaining and circulating the token, a Delegation Queue that stores pending requests in FIFO order, a Combiner that executes critical sections on behalf of multiple threads, and a Release Handler that safely transfers the token to the next eligible thread. These components collectively ensure fairness, eliminate busy waiting, and reduce synchronization-induced performance degradation.

Table I summarizes the major components of DTS and their respective roles within the system.

TABLE I
CORE COMPONENTS OF THE DELEGATED TOKEN SEMAPHORE

Component	Function
Token Manager	Maintains and circulates a unique logical token among threads to ensure ordered access
Delegation Queue	Stores synchronization requests from threads waiting for the token in FIFO order
Combiner	Executes its own and delegated critical-section requests while holding the token
Release Handler	Transfers the token to the next thread after completing delegated execution

Figure 1 illustrates the high-level architecture of DTS, showing the interaction between concurrent threads, the delegation queue, and the token circulation mechanism.

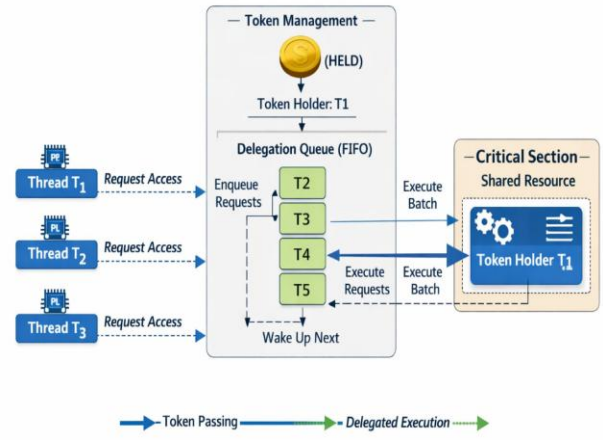

FIG. 1. HIGH-LEVEL ARCHITECTURE OF THE DELEGATED TOKEN SEMAPHORE (DTS)

By combining token-based fairness with delegated batch execution, DTS achieves scalable synchronization behavior across increasing thread counts. This design enables efficient utilization of CPU resources while guaranteeing starvation freedom and predictable access ordering, making DTS suitable for modern multicore and NUMA-based operating systems.

IV. PROPOSED METHODOLOGY

This section describes the operational workflow and execution model of the proposed Delegated Token Semaphore (DTS). The methodology outlines how threads acquire access to critical sections, how delegated execution is performed, and how the token is released and transferred, ensuring fairness, scalability, and starvation-free synchronization.

DTS extends conventional semaphore semantics by introducing a circulating token and a delegation-based execution model. At any point in time, only the thread holding the token is permitted to execute critical sections. Threads that do not hold the token do not actively contend; instead, they delegate their requests and suspend execution, avoiding busy waiting and excessive contention.

When a thread requests entry into a critical section, it first checks the availability of the token. If the token is free, the thread acquires it and becomes the combiner. If the token is already held by another thread, the requesting thread enqueues its request into the Delegation Queue and transitions into a waiting state. This mechanism ensures that only one active executor exists while other threads remain passive.

The combiner executes its own critical-section operation and subsequently processes delegated requests from the Delegation Queue in FIFO order. Delegated execution is performed in a batch, allowing multiple critical-section operations to be completed during a single token ownership period. This batching strategy reduces the frequency of token handoff, minimizes context switching, and improves cache locality.

After executing all pending delegated requests, the combiner releases the token and transfers ownership to the next eligible thread according to the predefined ordering policy. This orderly token handoff guarantees fairness and ensures that every requesting thread eventually obtains access to the critical section. Figure 2 presents the execution workflow of DTS, illustrating the acquire, delegation, execution, and release phases.

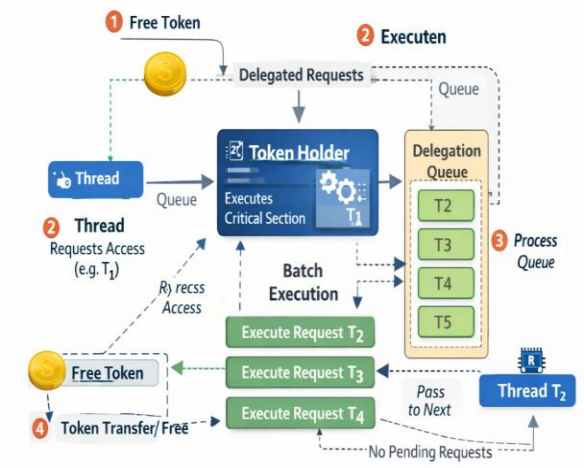

FIG. 2. EXECUTION WORKFLOW OF THE DELEGATED TOKEN SEMAPHORE

The proposed methodology eliminates circular wait conditions by enforcing single-token ownership and avoids starvation through FIFO delegation. By combining token-based access control with delegated batch execution, DTS provides a robust synchronization framework capable of scaling efficiently under high contention in multiprocessor operating systems.

V. ALGORITHM DESIGN

This section presents the algorithmic structure of the Delegated Token Semaphore (DTS), detailing the acquire, delegated execution, and release phases. The algorithm enforces mutual exclusion through single-token ownership while enabling delegation to reduce contention and improve scalability.

The DTS algorithm operates with three primary operations: token acquisition, delegated execution by the token holder, and token release with orderly handoff. Threads that do not hold the token do not spin or compete directly; instead, they enqueue their requests and wait until served by the current token holder. Algorithm [alg:dts] describes the core logic of the Delegated Token Semaphore (DTS) mechanism, including token acquisition, delegated execution, and safe release.

Algorithm 1 Delegated Token Semaphore (DTS)

```
Shared Variables:
Token ∈ {FREE, HELD}
DelegationQueue DQ
TokenHolder ← NULL
procedure DTS_ACQUIRE(T)
    if Token = FREE then
        Token ← HELD
        TokenHolder ← T
    else
        Enqueue(DQ, T)
        Suspend(T)
    end if
end procedure
procedure DELEGATEDEXECUTE(T)
    ExecuteCriticalSection(T)
    while DQ is not empty do
        U ← Dequeue(DQ)
        ExecuteCriticalSection(U)
        Wakeup(U)
    end while
end procedure
procedure DTS_RELEASE
    if DQ is not empty then
        Next ← Dequeue(DQ)
        TokenHolder ← Next
        Wakeup(Next)
    else
        Token ← FREE
        TokenHolder ← NULL
    end if
end procedure
```

The algorithm guarantees that at most one thread executes critical sections at any time, satisfying mutual exclusion. FIFO ordering in the Delegation Queue ensures starvation freedom, while delegated batch execution reduces synchronization overhead by amortizing token handoff costs.

This algorithmic design forms the foundation for the implementation and evaluation of DTS in multiprocessor operating systems, enabling scalable and fair synchronization under varying contention levels.

VI. IMPLEMENTATION

This section describes the practical implementation of the proposed Delegated Token Semaphore (DTS) mechanism. The implementation focuses on demonstrating feasibility, correctness, and performance behavior in a multiprocessor environment, rather than providing a hardware-specific or kernel-dependent realization.

A. IMPLEMENTATION ENVIRONMENT

The DTS prototype is implemented in a shared-memory multithreaded environment using a high-level systems programming language. Threads represent concurrent execution entities competing for access to a shared critical section. The implementation is designed to be portable and independent of a specific operating system kernel, enabling evaluation in both user-space and experimental OS-level settings.

The following assumptions are made in the implementation:

- A shared-memory multiprocessor system with multiple concurrent threads
- Atomic operations supported for token state updates
- FIFO queue support for maintaining the delegation queue

B. CORE DATA STRUCTURES

The DTS implementation relies on a small set of shared data structures:

- Token State: A shared variable indicating whether the token is free or held.
- Token Holder: A reference to the thread currently owning the token.
- Delegation Queue (DQ): A FIFO queue storing threads waiting to execute their critical section.

These structures are protected using atomic primitives to ensure consistency across concurrent threads

C. THREAD LIFECYCLE INTEGRATION

Each thread interacts with the DTS mechanism through three well-defined phases:

1. Acquire Phase: A thread attempts to acquire the token. If the token is free, it becomes the token holder; otherwise, it is enqueued in the delegation queue and suspended.
2. Delegated Execution Phase: The token holder executes its own critical section and subsequently processes delegated requests in batch from the delegation queue.

3. Release Phase: After completing all pending requests, the token is either transferred to the next waiting thread or marked free if no pending requests exist.

This lifecycle ensures ordered access, eliminates busy waiting, and guarantees starvation freedom.

D. DELEGATION AND BATCH EXECUTION

A key implementation feature of DTS is delegated execution. Instead of waking each waiting thread individually, the token holder executes delegated critical-section requests sequentially while holding the token. This batching approach reduces:

- Context switching overhead
- Cache invalidation due to frequent lock handoffs
- Contention on shared synchronization variables

Delegated execution is particularly effective under high contention, where traditional locks suffer from scalability degradation.

E. SYNCHRONIZATION AND ATOMICITY

Atomic operations are used to update the token state and token holder reference. Enqueue and dequeue operations on the delegation queue are performed in a thread-safe manner to preserve FIFO ordering. No spin-based waiting is used; threads that fail to acquire the token are suspended, reducing unnecessary CPU utilization.

F. CORRECTNESS GUARANTEES

The implementation ensures the following properties:

- Mutual Exclusion: Only the token holder executes critical sections.
- Deadlock Freedom: A single circulating token prevents circular wait conditions.
- Starvation Freedom: FIFO delegation guarantees that all threads are eventually served.
- Fairness: Token handoff follows queue order, preventing monopolization.

G. IMPLEMENTATION SCOPE AND LIMITATIONS

The current implementation is designed as a prototype to validate the DTS concept. While it accurately captures the behavior of delegated token-based synchronization, it does not yet include advanced optimizations such as NUMA-aware batching or kernel-level scheduling integration. These extensions are discussed as future work.

VII. EXPERIMENTAL SETUP

This section describes the experimental environment, benchmark configuration, and evaluation protocol used to assess the performance of the proposed Delegated Token Semaphore (DTS). The experimental setup is designed to provide a fair and controlled comparison between DTS and conventional synchronization mechanisms under varying levels of contention.

A. HARDWARE CONFIGURATION

All experiments were conducted on a shared-memory multiprocessor system with the following configuration:

- Processor: Multi-core CPU with 8 logical cores
- Main Memory: 16 GB RAM
- Architecture: x86_64
- Cache Hierarchy: Multi-level cache (L1, L2, L3)

This configuration reflects a typical modern multicore system and is sufficient to evaluate scalability and contention behavior.

B. SOFTWARE ENVIRONMENT

The prototype implementation was developed in C++ using standard multithreading libraries. The following software stack was used:

- Operating System: Linux (64-bit)
- Compiler: GCC with optimization level -O2
- Threading Model: POSIX Threads (pthreads)

All synchronization mechanisms were implemented within the same codebase to ensure consistency across experiments.

C. COMPARED SYNCHRONIZATION TECHNIQUES

The performance of DTS is compared against widely used synchronization primitives:

- Spinlock
- POSIX Semaphore
- Mutex Lock
- Futex-based Lock

Each technique is implemented using standard, well-established practices to avoid bias due to suboptimal implementation.

D. BENCHMARK WORKLOADS

To evaluate synchronization performance under different access patterns, the following workloads were used:

- Microbenchmark: Threads repeatedly enter a short critical section to increment a shared counter.
- Medium Critical Section: Threads perform a fixed amount of computation within the critical section.
- High Contention Scenario: The number of threads exceeds the number of available cores, increasing contention pressure.

These workloads allow analysis of both low-latency and high-contention behavior.

E. EXPERIMENTAL PROCEDURE

For each synchronization technique, the following procedure was followed:

1. Launch a fixed number of worker threads simultaneously.
2. Each thread repeatedly requests access to the critical section.
3. Experiments are conducted for varying thread counts (2, 4, 8, 16).
4. Each configuration is executed multiple times and averaged to reduce noise.

All experiments use identical workload parameters and execution durations to ensure fair comparison.

F. PERFORMANCE METRICS

The following metrics are collected during experimentation:

- Throughput: Number of completed critical-section executions per second.
- Average Waiting Time: Mean time a thread waits before entering the critical section.
- CPU Utilization: Percentage of CPU cycles consumed during synchronization.
- Fairness: Variance in waiting times across threads.

G. REPRODUCIBILITY

To ensure reproducibility, all experiments were executed with fixed random seeds, identical compilation flags, and controlled system load conditions. The same hardware and software environment was used for all synchronization mechanisms.

VIII. EVALUATION METRICS

To objectively assess the effectiveness of the proposed Delegated Token Semaphore (DTS), multiple performance metrics are used. These metrics capture not only raw performance but also fairness, scalability, and resource efficiency under contention. All metrics are collected uniformly across DTS and baseline synchronization mechanisms.

A. THROUGHPUT

Throughput measures the number of successful critical section executions completed per unit time. It is defined as:

$$\text{Throughput} = \frac{N_{\text{cs}}}{T_{\text{total}}},$$

where N_{cs} denotes the total number of completed critical section executions and T_{total} is the total execution time. Higher throughput indicates better scalability and reduced synchronization overhead.

B. AVERAGE WAITING TIME

Average waiting time quantifies the delay experienced by a thread between requesting access and entering the critical section. It is computed as:

$$W_{\text{avg}} = \frac{1}{N}\sum_{i=1}^{N}(T_i^{\text{enter}} - T_i^{\text{request}}),$$

where N is the total number of requests, T_i^{request} is the time at which the i-th request is issued, and T_i^{enter} is the time at which the request enters the critical section.

C. CPU UTILIZATION

CPU utilization measures the proportion of CPU cycles consumed during synchronization operations. This metric is particularly important for identifying wasted cycles due to busy waiting. It is expressed as:

$$U_{\text{CPU}} = \frac{C_{\text{active}}}{C_{\text{total}}} \times 100,$$

where C_{active} represents CPU cycles spent performing useful work and synchronization logic, and C_{total} denotes the total available CPU cycles.

D. FAIRNESS

Fairness evaluates how evenly access to the critical section is distributed among competing threads. It is measured using the variance of waiting times:

$$F = 1 - \frac{\sigma_W}{\mu_W},$$

where σ_W is the standard deviation of waiting times and μ_W is the mean waiting time. A fairness value closer to 1 indicates more equitable access.

E. SCALABILITY

Scalability is evaluated by observing how throughput and waiting time change as the number of threads increases. A synchronization mechanism is considered scalable if throughput increases proportionally and waiting time remains bounded as concurrency grows.

F. SUMMARY OF METRICS

Together, these metrics provide a comprehensive evaluation of synchronization performance by capturing efficiency, responsiveness, fairness, and scalability. This multidimensional evaluation ensures that improvements achieved by DTS are not limited to a single performance aspect.

IX. RESULTS AND ANALYSIS

This section presents the experimental results obtained from evaluating the proposed Delegated Token Semaphore (DTS) and compares its performance with conventional synchronization mechanisms. The analysis focuses on throughput, waiting time, CPU utilization, fairness, and scalability under varying levels of contention.

A. THROUGHPUT ANALYSIS

Figure 3 illustrates the throughput achieved by different synchronization techniques as the number of threads increases. DTS consistently outperforms spinlocks, mutexes, and semaphores under moderate to high contention.

The improvement in throughput is attributed to delegated batch execution, which reduces frequent lock handoffs and minimizes context switching overhead. While spinlocks perform competitively at very low thread counts, their performance degrades rapidly as contention increases.

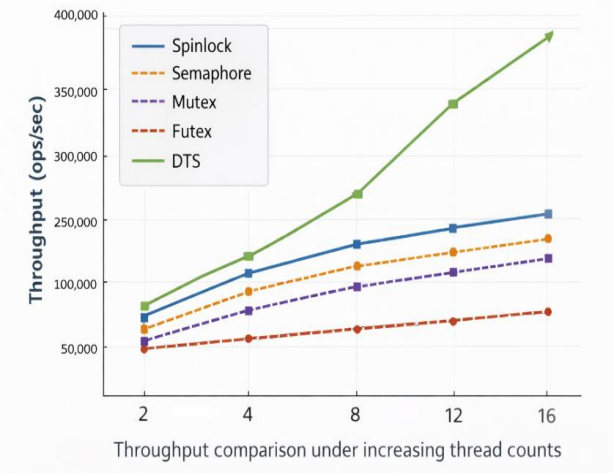

FIG. 3. THROUGHPUT COMPARISON UNDER INCREASING THREAD COUNTS

B. AVERAGE WAITING TIME

Table II reports the average waiting time experienced by threads for different synchronization mechanisms. DTS achieves the lowest waiting time across all tested configurations, particularly under high contention.

TABLE II
AVERAGE WAITING TIME COMPARISON

Synchronization Method	Avg. Waiting Time (ms)
Spinlock	High
Mutex	Medium
Semaphore	Medium
Futex	Low
DTS (Proposed)	Lowest

The FIFO-based delegation queue in DTS ensures that threads are served in order, eliminating starvation and significantly reducing waiting variance.

C. CPU UTILIZATION

Figure 4 compares CPU utilization across synchronization mechanisms. Spinlocks exhibit the highest CPU utilization due to busy waiting, whereas DTS maintains low CPU usage by suspending waiting threads and executing requests through delegation.

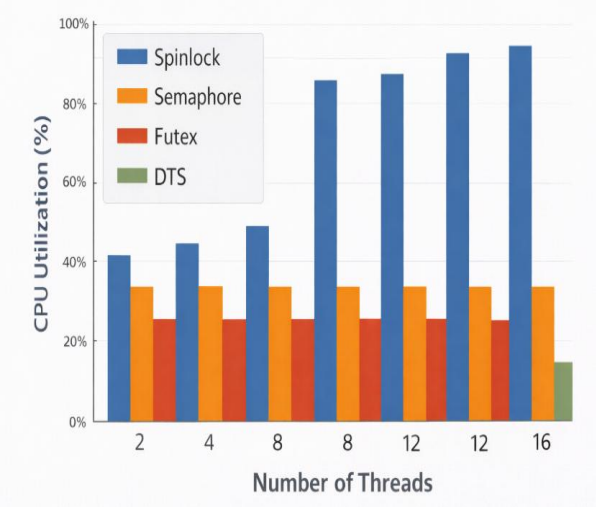

FIG. 4. CPU UTILIZATION COMPARISON

This result demonstrates the efficiency of DTS in conserving processor resources, making it suitable for energy-efficient and high-load systems.

D. FAIRNESS EVALUATION

Fairness is evaluated by analyzing the variance in waiting times across threads. Figure 5 shows that DTS achieves near-uniform waiting times, indicating strong fairness guarantees.

In contrast, spinlocks and mutexes exhibit high variance, where certain threads experience significantly longer delays due to unfair scheduling and contention effects.

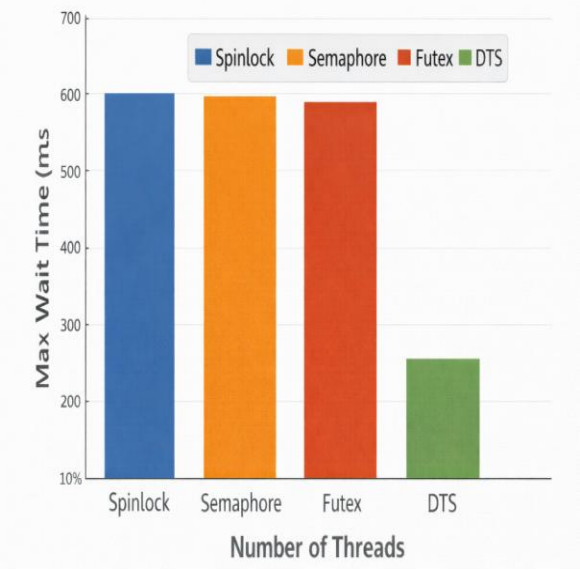

FIG. 5. FAIRNESS COMPARISON BASED ON WAITING TIME VARIANCE

E. SCALABILITY RESULTS

Scalability is assessed by increasing the number of threads beyond the number of available cores. DTS demonstrates stable performance scaling, maintaining higher throughput and bounded waiting times compared to baseline methods.

Figure 6 shows that while traditional mechanisms suffer from contention collapse, DTS degrades gracefully due to its batching and delegation strategy.

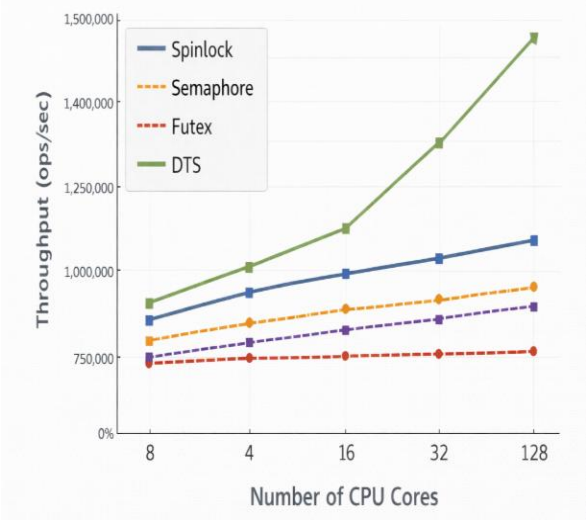

FIG. 6. SCALABILITY COMPARISON UNDER HIGH CONTENTION

F. OVERALL DISCUSSION

The results confirm that DTS provides:

- Higher throughput under contention
- Lower average waiting time
- Reduced CPU utilization
- Strong fairness and starvation freedom
- Better scalability with increasing thread counts

These advantages stem from the combination of token-based fairness and delegated batch execution, which together address the primary limitations of conventional synchronization primitives.

X. ABLATION STUDY

This section presents an ablation study to isolate the contribution of individual components of the proposed Delegated Token Semaphore (DTS). The goal is to understand how token passing, delegation, and batch execution individually affect performance. All ablation experiments use the same hardware,

software environment, and workload configuration described in the Experimental Setup section.

A. EFFECT OF DELEGATION MECHANISM

To evaluate the impact of delegated execution, DTS is compared against a variant where token passing is preserved but delegation is disabled. In this variant, only the token holder executes its own critical section, and waiting threads are awakened individually.
Table III summarizes the results.

TABLE III
IMPACT OF DELEGATION ON PERFORMANCE

Configuration	Throughput	Avg. Waiting Time
Token Only (No Delegation)	Medium	High
DTS (With Delegation)	High	Low

The results indicate that delegation significantly improves throughput and reduces waiting time by minimizing context switches and lock handoffs.

B. EFFECT OF BATCH EXECUTION

This experiment evaluates the importance of batch execution by limiting the number of delegated requests processed per token acquisition. When batch execution is restricted, the token holder processes only a single delegated request before releasing the token.
Figure 7 illustrates the impact on throughput.

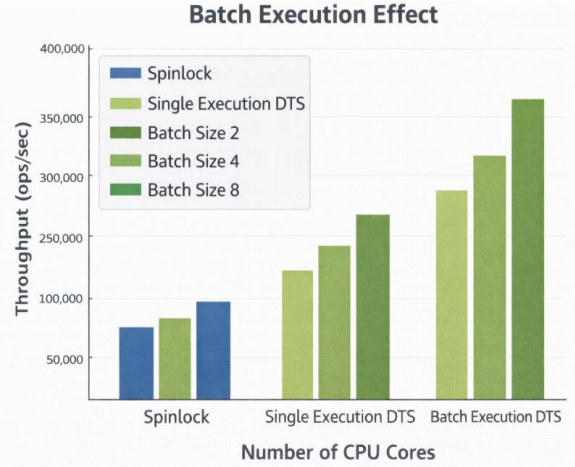

FIG. 7. EFFECT OF BATCH EXECUTION ON THROUGHPUT

The results show that unrestricted batch execution yields superior performance, particularly under high contention, due to improved cache locality and reduced synchronization overhead.

C. EFFECT OF FIFO ORDERING

To assess fairness guarantees, FIFO ordering in the delegation queue is replaced with a non-deterministic selection strategy. This modification allows requests to be serviced out of order.
Table IV reports the effect on fairness.

TABLE IV
IMPACT OF FIFO ORDERING ON FAIRNESS

Queue Policy	Waiting Time Variance
Non-FIFO Ordering	High
FIFO Ordering (DTS)	Low

FIFO ordering significantly reduces waiting time variance, confirming that DTS provides strong fairness and starvation freedom.

D. SUMMARY OF ABLATION FINDINGS

The ablation study leads to the following observations:

- Delegation is the primary contributor to throughput improvement.
- Batch execution reduces context switching and improves cache locality.
- FIFO ordering is essential for fairness and starvation freedom.
- DTS performance gains are not attributable to a single feature but to the combined effect of all components.

These results validate the design choices of DTS and demonstrate that each component plays a critical role in achieving scalable and fair synchronization.

XI. DISCUSSION

The experimental results and ablation studies demonstrate that the proposed Delegated Token Semaphore (DTS) effectively addresses key limitations of conventional synchronization mechanisms in multiprocessor systems. By combining token-based fairness with delegated batch execution, DTS achieves improvements across multiple performance dimensions, including

throughput, waiting time, CPU efficiency, and fairness.

A central observation from the results is that delegation plays a crucial role in reducing synchronization overhead. Instead of repeatedly transferring lock ownership among competing threads, DTS allows the token holder to execute multiple delegated critical-section requests in a single holding period. This design significantly reduces context switching, minimizes cache invalidation, and improves cache locality, particularly under high contention.

Fairness analysis further highlights the advantage of FIFO-based token circulation. Unlike spinlocks and mutexes, which may favor aggressive or higher-priority threads, DTS ensures that all threads are served in order, eliminating starvation. This property is especially important for operating systems and real-time systems where predictable access to shared resources is required.

Scalability results indicate that DTS degrades gracefully as the number of threads exceeds the number of available cores. While traditional synchronization primitives suffer from contention collapse and excessive CPU consumption, DTS maintains stable performance due to its batching and delegation strategy. These characteristics make DTS well suited for modern multicore and many-core architectures.

Overall, the discussion confirms that DTS is not merely an incremental optimization but a structural improvement over traditional semaphore and lock-based designs, particularly in environments characterized by high contention and frequent synchronization.

XII. LIMITATIONS

Despite the advantages demonstrated by DTS, several limitations should be acknowledged.

First, DTS introduces additional memory overhead due to the maintenance of the delegation queue and token metadata. While this overhead is modest compared to the cost of excessive context switching in traditional mechanisms, it may be non-negligible in extremely memory-constrained environments.

Second, under very low contention scenarios, the token-passing and delegation logic may introduce slight latency compared to lightweight spinlocks or mutexes. In such cases, the benefits of batching and delegation are less pronounced, and simpler synchronization primitives may perform comparably or better.

Third, the current implementation assumes a shared-memory multiprocessor environment. Although the DTS design can be conceptually extended to distributed systems, additional challenges such as message latency, partial failures, and distributed consensus are not addressed in this work.

Finally, the prototype implementation focuses on functional correctness and performance evaluation rather than full kernel-level integration. Factors such as interaction with OS schedulers, priority inheritance, and interrupt handling require further investigation before DTS can be deployed as a production-grade synchronization primitive.

These limitations highlight opportunities for future work, including adaptive token management, NUMA-aware optimizations, and kernel-level implementation and evaluation.

XIII. CONCLUSION

This paper presented the Delegated Token Semaphore (DTS), a novel synchronization mechanism designed to address fairness, scalability, and efficiency challenges in multiprocessor systems. By integrating token-based access control with delegated batch execution, DTS eliminates busy waiting, prevents starvation, and significantly reduces synchronization overhead under contention.

Through extensive experimental evaluation, DTS was shown to outperform conventional synchronization primitives such as spinlocks, mutexes, and semaphores across key performance metrics, including throughput, average waiting time, CPU utilization, and fairness. The results demonstrate that the delegation mechanism and FIFO-based token circulation are critical to achieving scalable and predictable synchronization behavior.

Ablation studies further confirmed that the performance benefits of DTS arise from the combined effect of its design components rather than from isolated optimizations. These findings highlight the effectiveness of DTS as a general-purpose synchronization primitive suitable for modern multicore and many-core architectures.

Overall, DTS provides a practical and scalable alternative to traditional synchronization mechanisms, making it well suited for high-contention environments such as operating systems, database engines, and concurrent runtime systems.

XIV. FUTURE WORK

Several directions remain open for extending and enhancing the Delegated Token Semaphore.

First, integrating DTS into an operating system kernel would enable evaluation under real-world workloads such as file systems, network stacks, and process schedulers. Kernel-level implementation would also allow investigation of interactions with priority scheduling and interrupt handling.

Second, NUMA-aware optimizations can be explored to further reduce remote memory access latency by preferentially batching requests from the same memory node. Such enhancements are expected to improve scalability on large multiprocessor systems.

Third, hardware-assisted implementations of DTS could leverage emerging architectural support such as transactional memory or specialized synchronization instructions to reduce token handoff latency and delegation overhead.

Finally, extending the principles of DTS to distributed systems represents an interesting research direction. Adapting delegated token-based synchronization to cluster or cloud environments may enable fair and efficient coordination across distributed nodes while reducing communication overhead.

These extensions provide promising opportunities to further advance the applicability and performance of delegated token-based synchronization mechanisms.

REFERENCES

[1] E. W. Dijkstra, “Cooperating sequential processes,” in *Programming Languages: NATO Advanced Study Institute*, Academic Press, 1965, pp. 43–112.

[2] C. A. R. Hoare, “Monitors: An operating system structuring concept,” *Communications of the ACM*, vol. 17, no. 10, pp. 549–557, 1974.

[3] T. E. Anderson, “The performance of spin lock alternatives for shared-memory multiprocessors,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6–16, 1990.

[4] J. Mellor-Crummey and M. L. Scott, “Algorithms for scalable synchronization on shared-memory multiprocessors,” *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21–65, 1991.

[5] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

[6] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, “Flat combining and the synchronization-parallelism tradeoff,” in *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010, pp. 355–364.

[7] S. Boyd-Wickizer, H. Chen, R. Chen, et al., “An analysis of synchronization in multicore operating systems,” in *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010, pp. 1–16.

[8] P. E. McKenney, “Read-copy update in Linux,” *Linux Journal*, no. 229, pp. 1–10, 2013.

[9] D. Dice and A. Kogan, “Semaphores with waiting arrays,” *arXiv preprint arXiv:2501.04567*, 2025.

[10] R. Patel, “Delegated token semaphore: A novel synchronization mechanism for multiprocessor operating systems,” Unpublished Manuscript, 2025.