Thread Synchronization in Concurrent Systems: A Study of Dekker's Algorithm, Peterson's Algorithm, and Semaphore-Based Solutions

Nzenwata U.J.[1], Adegunle I.S.[2], Idume-David J.E.[3], Abu S.E.[4], Akanbi-Bello T.O.[5], Oliyide O.O.[6], Labode O.E.[7], Olatunji O.T.[8]

Department of Computer Science, Babcock University Ilishan Remo, Ogun State, Nigeria.

DOI: https://doi.org/10.51244/IJRSI.2026.1313CS006

Received: 02 April 2026; Accepted: 08 April 2026; Published: 27 April 2026

ABSTRACT

The proliferation of multi-core processors has made concurrency a central paradigm in modern software development. While concurrent execution offers significant performance benefits, it introduces the critical challenge of synchronizing access to shared resources to prevent race conditions, data corruption, and inconsistent states. This paper traces the evolution of thread synchronization solutions, beginning with the pioneering software-only mutual exclusion algorithms of Dekker and Peterson. These algorithms provided the theoretical foundation for ensuring that only one thread can execute in a critical section at a time without relying on hardware-specific instructions. The paper then explores how Dijkstra's semaphore concept extended these ideas into a more practical and powerful tool by eliminating busy waiting and providing a structured mechanism for signaling and coordination. The primary objectives are to analyze the design, properties, and limitations of Dekker's and Peterson's algorithms, demonstrate the application of semaphores to classical synchronization problems; namely the Producer-Consumer, Bounded Buffer, and Readers-Writers problems and provide a comparative analysis of their advantages and relevance in the context of modern operating systems and programming languages. The analysis concludes that while the classical algorithms are primarily of historical and educational value today, the principles they established are directly embodied in the sophisticated synchronization primitives used in contemporary systems.

Keywords: Concurrency, Mutual Exclusion, Dekker's Algorithm, Peterson's Algorithm, Semaphores, Synchronization, Operating Systems.

INTRODUCTION

In computing, concurrency refers to the ability of a system to decompose and make progress on multiple tasks over the same period. This is achieved through multi-threading, where a single process can contain multiple threads of execution. These threads may run in a time-sliced, interleaved fashion on a single processor core or in genuine parallel on multi-core systems [1]. The driving forces behind concurrent programming are performance and responsiveness [2]. It allows applications to continue responding to user input while performing intensive background computations, enabling server systems to handle thousands of simultaneous clients, and fully utilizes the computational power of modern multi-core CPUs. At the heart of concurrent systems lies the management of shared resources, such as global variables, data structures, and files. When threads operate independently on private data, concurrency is straightforward. However, when they need to read and modify shared data, careful coordination is required to ensure the system behaves correctly and predictably.

Problem Statement

The fundamental challenge in concurrent programming arises from the non-atomic nature of most high-level language operations. A single statement like count++ may compile to multiple machine instructions (load, increment, store). If two threads execute this statement concurrently without synchronization, the final value of

count may be incorrect due to interleaving of these instructions. This is known as a race condition [3]. The risks associated with unsynchronized access include:

a. Data Inconsistency: Here, shared data can be left in a corrupted or logically invalid state.

b. Race Conditions: While two threads are concurrently executing the same instruction, the correctness of the program depends on the relative timing of thread execution which leads to non-deterministic and often erroneous behavior.

c. Deadlocks: This is a situation where two or more threads become permanently blocked because each is holding a resource and waiting indefinitely for another resource held by a different thread in the same set.

As Deitel et al. state, "Without proper synchronization, it is possible that one thread will be in the process of updating a shared data structure when another thread attempts to read it. The result is often that the second thread reads inconsistent data" [4, p. 187]. As a result, managing these risks is the central problem addressed by synchronization mechanisms. This paper aims to solve this issue by studying Dekker's and Peterson's algorithm and demonstrating how semaphores can generalize to solve these synchronization problems.

Objectives of the Study

This paper has two objectives:

1. To explain the mechanics, properties, and limitations of classical software-based mutual exclusion algorithms, specifically Dekker's and Peterson's algorithms.

2. To demonstrate how semaphores generalize and extend these foundational concepts to solve practical, real-world synchronization problems beyond simple mutual exclusion.

Historical Background of Mutual Exclusion

The problem of mutual exclusion emerged with the first multiprogramming systems in the 1960s, where multiple processes required controlled access to shared resources. Early solutions took divergent paths: one leveraging the processor's instruction set for simplicity and directness, and another pursuing the more complex challenge of achieving synchronization purely in software. Understanding both trajectories is essential to appreciating the subsequent revolution brought about by the semaphore.

Early Approaches to Mutual Exclusion

Hardware-Based Synchronization Instructions

The most straightforward early solutions to mutual exclusion relied on hardware support in the form of atomic CPU instructions. These instructions execute as a single, uninterruptible operation, providing a fundamental building block [5] for constructing locks and other synchronization primitives.

Test-and-Set (TAS)

The Test-and-Set instruction is one of the simplest hardware primitives. It atomically reads a memory location and sets it to a new value (typically 1), returning the *previous* value.

Pseudocode:

```
function TestAndSet(boolean *lock) {
    boolean old = *lock;
    *lock = 1;
    return old;}
```

This instruction can implement a basic spinlock. A thread attempting to acquire the lock continuously executes TestAndSet in a loop until it returns 0, indicating the lock was free and is now held.

Pseudocode of a basic spinlock:

```
volatile int lock = 0;

void critical() {

   while (test_and_set(&lock) == 1);

   critical section  // only one process can be in this section at a time

   lock = 0;  }
```

The primary limitation of TAS is its high memory bus contention on multi-processor systems [6], as every waiting thread generates a continuous stream of write operations to the lock variable.

Test-and-Test-and-Set (TTAS)

To mitigate the bus contention of pure TAS, the Test-and-Test-and-Set optimization was developed. A thread first spins while reading the lock value (a cheap cacheable operation) and only invokes the atomic TestAndSet when it appears the lock has been released.

Pseudocode of a TTAS Lock:

```
// Shared variables

volatile boolean lock = false

function TTAS_acquire():

   do {        // First test: read without invalidating cache

      while (lock == true):

         // Busy-wait (spin) but cache line remains in shared state

         // This reduces bus traffic compared to constant TAS

   } while (test_and_set(&lock) == true)  // Second test: actual atomic acquisition

      // Lock acquired - enter critical section

  access_shared_resource();

function TTAS_release():

   lock = false
```

This approach significantly reduces expensive atomic operations and associated bus traffic, making it more efficient than TAS on modern architectures.

Compare-and-Swap (CAS)

Compare-and-Swap (known as CMPXCHG in x86) is a more powerful and versatile atomic instruction [7]. It compares the contents of a memory location to an expected old value and, only if they match, writes a new value to that location.

Pseudocode:

```
function CompareAndSwap(pointer *location, expected_value, new_value):
    old_value = *location
    if (old_value == expected_value):
        *location = new_value
        return old_value  // Returns the value that was at the location
```

CAS is the cornerstone of modern non-blocking (lock-free) algorithms, allowing for complex modifications to be attempted without holding a lock.

Pseudocode of a simple atomic counter:

```
volatile int counter = 0
function increment_counter():
    do {
        current = counter
        next = current + 1
    } while (!CompareAndSwap(&counter, current, next))
```

It is used extensively in Java's java.util.concurrent.atomic classes and C++'s std::atomic library.

Other Key Instructions include:

Fetch-and-Add: Atomically increments a variable and returns its *previous* value. It is the foundation for fair locks like ticket locks, which function like a deli counter to ensure first-come-first-served fairness and prevent starvation.

Load-Link / Store-Conditional (LL/SC): A pair of instructions found in RISC architectures (ARM, MIPS). Load-Link reads a value and monitors the address; a subsequent Store-Conditional succeeds only if no other thread has written to the location since. This pair can be used to implement all other atomic primitives [8].

Limitations of Hardware-Centric Approaches

Despite their simplicity and guaranteed atomicity, hardware-based solutions had significant limitations that motivated the search for software alternatives.

a. Busy Waiting: The most critical drawback, also known as spinning, consumes CPU cycles indefinitely while a thread waits for a lock. As Deitel et al. state, "The main disadvantage of the test-and-set instruction is that it requires busy waiting…This continues to tie up the CPU" [4, p. 229]. This is profoundly inefficient in single-processor systems.

b. Priority Inversion: In systems with thread priorities, a low-priority thread holding a lock could prevent a high-priority thread from running, a situation known as priority inversion. If the low-priority thread is preempted, the high-priority thread will busy-wait indefinitely.

c. Starvation: Simple spinlocks have no inherent fairness guarantee. A thread might be consistently bypassed by others when attempting to acquire a lock.

d. Architectural Dependency: These instructions are processor-specific, making code non-portable across different hardware platforms.

These limitations, particularly the inefficiency of busy waiting, created a clear need for synchronization mechanisms that could block a waiting thread, freeing the CPU for other work.

Dekker and Peterson's Contributions

In response to the drawbacks of hardware-based locking, computer scientists sought pure software solutions. The Dutch mathematician Th. J. Dekker was the first to devise a correct algorithm for mutual exclusion between two processes using only shared memory for communication [9]. His algorithm used a combination of flags and a turn variable to ensure that both processes could not enter their critical sections simultaneously. Gary L. Peterson later published a significantly simpler and more elegant algorithm [10] that achieved the same guarantees. Peterson's algorithm, using just two flags and a turn variable, became the canonical example for teaching software-based mutual exclusion.

These algorithms were monumental for theoretical computer science. They demonstrated that mutual exclusion could be enforced reliably without specialized hardware support, relying solely on standard load/store instructions and careful algorithmic design. They provided the formal foundation for proving essential properties like mutual exclusion, progress, and bounded waiting, establishing the core criteria for all future synchronization solutions. However, they still relied on busy waiting [11] and were too complex to generalize for more than two processes, leaving room for a more practical abstraction.

Emergence of Semaphores

In 1965, Edsger W. Dijkstra introduced the semaphore [12], a variable used to control access to a common resource by multiple processes in a concurrent system. This was a paradigm shift. The semaphore was an abstract data type that moved synchronization from low-level, algorithm-specific code to a high-level, OS-managed primitive. Dijkstra defined two atomic operations on a semaphore **S**:

a. **P()** (from the Dutch *proberen*, to test): Decrements **S**. If **S** becomes negative, the process is blocked and placed in a waiting queue.

b. **V()** (*verhogen*, to increment): Increments **S**. If there are any processes blocked in the queue, one is unblocked.

The key advancement was that the semaphore's implementation within the operating system kernel could move waiting threads from the busy-waiting state to a blocked state, freeing the CPU for other work. This solved the primary inefficiency of both hardware spinlocks and software algorithms like Peterson's. Furthermore, semaphores formalized not only mutual exclusion (via binary semaphores) but also inter-process signaling and coordination (via counting semaphores), providing a unified tool for a wide range of synchronization problems. This set the stage for the practical solutions to classical problems discussed in the following sections.

Dekker's Algorithm

Dekker's algorithm is a seminal solution to the mutual exclusion problem for two processes [2], [13]. It employs three shared variables: two Boolean flags (flag[0] and flag[1]) and a turn variable. The flag for a process indicates its desire to enter the critical section, while the turn variable is used to arbitrate when both processes wish to enter simultaneously, thus ensuring fairness and preventing deadlock. The algorithm works by having a process first express its desire to enter by setting its flag to true. It then checks the other process's flag. If the other process is not interested, it proceeds immediately. If the other process is also interested, the process consults the turn variable. The process whose turn it is proceeds, while the other process resets its flag (to be polite and avoid deadlock) and waits for its turn. After exiting the critical section, the process resets its flag and updates the turn variable.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue XIII April 2026
Special Issue on Emerging Paradigms in Computer Science and Technology

Pseudocode

This pseudocode illustrates Dekker's algorithm for two processes, *P0* and *P1*.

```
// Shared variables - initialized to false

boolean flag[2] = {false, false}

int turn = 0  // 0 or 1 - indicates whose turn it is

// Process 0

function process_0():

    while true:

        // Express interest in critical section

        flag[0] = true

            // Check if other process is interested

        while flag[1] == true:

            // If other process is interested, check whose turn it is

            if turn != 0:

                // Yield to other process

                flag[0] = false

                        // Wait for our turn

                while turn != 0:

                    // Busy wait

                        // Re-express interest

                flag[0] = true

                // Critical Section

        critical_section_0()

            // Exit protocol

        turn = 1          // Give turn to other process

        flag[0] = false   // No longer interested

            // Remainder Section

        remainder_section_0()

// Process 1 (symmetric)
```

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue XIII April 2026
Special Issue on Emerging Paradigms in Computer Science and Technology

```
function process_1():
  while true:
    flag[1] = true
        while flag[0] == true:
      if turn != 1:
        flag[1] = false
        while turn != 1:
          // Busy wait
        flag[1] = true


    // Critical Section
    critical_section_1()
        // Exit protocol
    turn = 0
    flag[1] = false
        // Remainder Section
    remainder_section_1()
```

Properties of Dekker's algorithm

Dekker's algorithm satisfies the three essential criteria for a correct synchronization solution:

a. Mutual Exclusion: The structure of the flags and the turn variable guarantees that at most one process can be in its critical section at any time. It is impossible for both processes to pass the while condition simultaneously.

b. Progress: A process wishing to enter its critical section will eventually be able to do so. The algorithm does not allow a situation where both processes are stuck forever outside their critical sections.

c. Bounded Waiting: A process will not be starved indefinitely. The turn variable ensures that after one process has exited its critical section, it will pass the turn to the other, guaranteeing the waiting process a chance to enter.

Limitations

Despite its correctness, Dekker's algorithm has significant drawbacks that limit its practical use:

a. Busy Waiting Overhead: The algorithm relies on the processes actively spinning in while loops when they are waiting. This consumes CPU cycles that could be used for other productive work, making it inefficient.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue XIII April 2026
Special Issue on Emerging Paradigms in Computer Science and Technology

b. Complexity: The logic is intricate and difficult to understand, verify, and prove correct. A small error in implementation can lead to subtle bugs.

c. Lack of Scalability: The algorithm is designed explicitly for two processes. Extending it to support N processes is extremely complex and inefficient [14], and other solutions are far more suitable for such cases.

Deitel et al. summarize this well: "Although this approach [Dekker's algorithm] guarantees mutual exclusion, it is quite intricate, and its complexity demonstrates how difficult it can be to create software-based solutions to the mutual-exclusion problem" [4, p. 193].

Recent formal analyses of Dekker’s algorithm have shed new light on its theoretical significance. Nigro, Cicirelli, and Pupo conducted a rigorous modeling and analysis of Dekker-based mutual exclusion algorithms using formal verification tools, confirming that Dekker’s algorithm correctly satisfies mutual exclusion, progress, and bounded waiting when executed under a sequentially consistent memory model [15]. Their work further highlights that on modern relaxed-memory architectures (such as x86 TSO and ARM), the algorithm’s correctness guarantees break down without explicit memory fences, making it unsuitable for direct deployment in contemporary multi-core systems without hardware-level modification. Kode and Oyemade similarly observe that while classical algorithms like Dekker’s provide foundational theoretical value, their reliance on sequential consistency renders them incompatible with the reordering behaviors of modern processors [16]. These findings reinforce the conclusion that Dekker’s algorithm is best understood as a historical milestone and educational artifact rather than a deployable synchronization solution.

Peterson’s Algorithm

Peterson's algorithm simplifies the approach taken by Dekker while maintaining the same correctness guarantees [3], [12]. It also uses two flag variables and a single turn variable, but its logic is more straightforward and elegant. The key insight is the line turn = j; which a process uses to willingly yield to the other process. A process enters its critical section only if the other process is not interested *or* if it is the process's own turn.

Pseudocode for two processes:

```
// Shared global variables

boolean flag[2] = {false, false}  // Interest flags for both processes

int turn = 0                      // Whose turn it is to enter (0 or 1)

// Process 0

procedure process_0():

    while true:

        // Entry protocol - express interest

        flag[0] = true

        turn = 1                  // let process 1 go first

        // Wait until safe to enter

        while flag[1] == true and turn == 1:

            // Busy wait - cannot enter yet
```

```
            // Process 1 is interested AND it's their turn

                // CRITICAL SECTION

        // Only one process can be here at a time

        critical_section_0()


        // Exit protocol

        flag[0] = false            // No longer interested

                // REMAINDER SECTION

        remainder_section_0()

// Process 1 (symmetric to process 0)

procedure process_1():

    while true:

        flag[1] = true

        turn = 0                   // Let process 0 go first


        while flag[0] == true and turn == 0:

            // Busy wait

            // Process 0 is interested AND it's their turn


        // CRITICAL SECTION

        critical_section_1()

        flag[1] = false

        remainder_section_1()
```

Properties of Peterson's algorithm

Like Dekker's algorithm, Peterson's algorithm satisfies all three critical properties:

a. Mutual Exclusion: The condition in the while loop ensures that if both processes have set their flags to true, the turn variable will break the tie. Only one process can have the condition (flag[j] && turn == j) evaluate to false, allowing it to proceed.

b. Progress: A process cannot be blocked from entering its critical section by a process stuck in its remainder section. If a process wishes to enter, it will eventually set its flag and either find the other's flag is false or wait for its turn, which will be granted.

c. Bounded Waiting: When a process exits its critical section, it sets its flag to false. If the other process is waiting, it will immediately be able to proceed. The turn variable ensures that if both processes continuously contend for the critical section, they will alternate access.

The primary advantage of Peterson's algorithm is its simplicity. It achieves the same robust guarantees as Dekker's algorithm with significantly less complex code. This makes it easier to teach, understand, and prove correct. It serves as a brilliant pedagogical tool for demonstrating the principles of software-based mutual exclusion. However, it shares the same fundamental limitations: it is designed for exactly two processes and relies on busy waiting.

A rigorous property assessment of Peterson’s mutual exclusion algorithm by Nigro and Cicirelli further validates its theoretical correctness. Through formal modeling and model checking, the authors verified that Peterson’s algorithm satisfies all three required properties: mutual exclusion, progress, and bounded waiting under a sequentially consistent memory model, while also demonstrating that it fails under relaxed memory models commonly found in modern multi-core processors [17]. This formally corroborates the long-held assessment that Peterson’s algorithm is ideal for education but impractical for direct deployment on contemporary hardware without explicit memory barriers. In another study, [18] note that in multi-threaded environments, the assumptions underlying classical algorithms such as Peterson’s are rarely satisfied without deliberate memory ordering constraints, which modern synchronization libraries handle transparently through hardware-aware primitives. The simplicity of Peterson’s algorithm which [17] confirm can be fully verified with minimal formal overhead. They further explain its enduring pedagogical appeal even as practical synchronization has long moved to semaphores, mutexes, and higher-level constructs. Below is a table that shows a comparative analysis of Dekker’s and Peterson’s mutual exclusion algorithm:

Feature	Dekker’s Algorithm	Peterson’s Algorithm
Algorithmic Complexity	High — intricate flag/yield logic requiring multiple nested conditions to resolve contention [2], [15]	Low — elegant single-condition wait (flag[j] && turn == j); significantly simpler to implement and verify [3], [17]
Shared Variables	Two Boolean flags (flag[0], flag[1]) and one integer turn variable	Two Boolean flags (flag[0], flag[1]) and one integer turn variable; structurally identical but logically simpler
Mutual Exclusion	Guaranteed — the flag/turn structure prevents simultaneous critical section entry; formally verified [15]	Guaranteed — the combined flag and turn condition ensures only one process proceeds at a time; formally verified [17]
Progress	Satisfied — a process wishing to enter will eventually do so; no indefinite external blocking [2]	Satisfied — a process in its remainder section cannot block another from entering [3], [17]
Bounded Waiting	Satisfied — the turn variable ensures the waiting process is granted access after at most one cycle [2], [15]	Satisfied — alternating turn grants each process access within one contention cycle [3], [17]
Busy Waiting	Yes — waiting threads spin in while loops, consuming CPU cycles proportional to contention duration [4], [15]	Yes — waiting threads spin on (flag[j] && turn == j), wasting CPU; primary practical limitation [4], [18]

Feature	Dekker's Algorithm	Peterson's Algorithm
Memory Model Requirement	Requires sequentially consistent memory; correctness breaks down on modern x86/ARM relaxed-memory architectures without explicit fences [15]	Requires sequentially consistent memory; fails on modern processors without memory barriers; confirmed by formal analysis [17]
Processes Supported	Two only — extension to N processes is extremely complex and impractical [14]	Two only — not designed for N > 2; generalisation requires fundamentally different approaches [3]
Strengths	Historically first correct software-only mutual exclusion solution; established foundational proof that hardware support is not required [2], [9]	Simpler and more elegant than Dekker's; canonical teaching example; easy to prove correct; widely used in OS education [3], [8]
Limitations	Complexity makes implementation error-prone; busy waiting wastes CPU; impractical on real hardware without memory fences [4], [15]	Still relies on busy waiting; limited to two processes; inefficient in multi-core environments without adaptation [3], [18]

Table 1: Comparative Analysis of Dekker's and Peterson's Mutual Exclusion Algorithms

From Dekker/Peterson to Semaphores

The evolution from the classical algorithms to semaphores represents a move from specific, intricate logic to a general, abstract synchronization primitive. The *flag* variables in Dekker's and Peterson's algorithms can be conceptually viewed as a precursor to a binary semaphore (or mutex), which is a lock with two states (0 and 1). The intricate dance of setting flags and yielding via the *turn* variable is abstracted away by the semaphore's wait() and signal() operations. The turn variable's role in ensuring fairness is analogous to the semaphore's queueing mechanism, which unblocks waiting threads in a defined order.

Busy Waiting vs Blocking

The most significant practical advancement of semaphores is the move from busy waiting to blocking. In Dekker's and Peterson's algorithms, as seen in figure 1 below, a waiting thread continuously polls the shared variables, consuming CPU cycles [19]. In the Dekker/Peterson (busy-wait) model, a thread waiting to enter the critical section continuously spins consuming 100% of its allocated CPU time until the turn variable signals its entry.

In a semaphore implementation, the wait() operation, when it cannot proceed, will atomically decrement the semaphore's value and place the thread in a waiting queue associated with the semaphore. The thread's state is changed from "running" to "waiting," and the OS scheduler is invoked to select another thread to run. This frees the CPU for useful work. When another thread performs a signal() operation, it increments the semaphore and unblocks one waiting thread, moving it back to the "ready" state. As Deitel et al. explain, "This is a key benefit of semaphores processes that a blocked waiting for a semaphore do not consume processor time... The operating system is responsible for managing the waiting queue and awakening a process when the semaphore becomes available" [4, p. 200].

Figure 1 below illustrates the contrasting thread interaction models of the two synchronization paradigms:

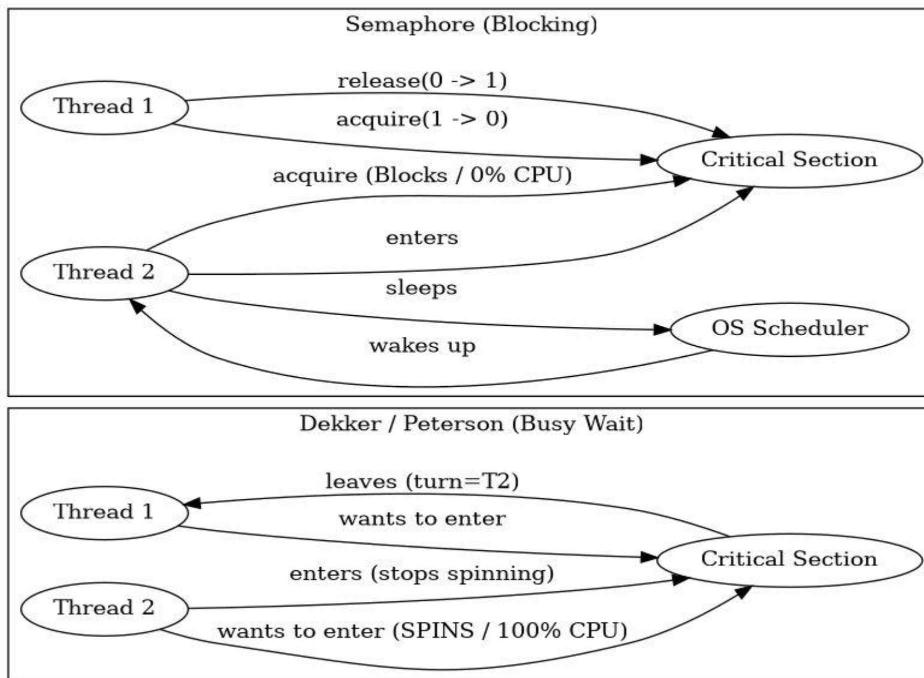

Figure 1: Thread Interaction Flow — Semaphore Blocking vs. Dekker/Peterson Busy Waiting

Semaphore Operations

A semaphore S is an integer variable upon which only two operations are defined. These operations are guaranteed to be atomic; that is, they cannot be interrupted.

a. wait(S) (or P(S)): This operation decrements the value of S. If S becomes negative as a result, the thread executing the wait is blocked and placed in the semaphore's waiting queue. Otherwise, the thread continues execution.

```
wait(S) {
   S--;
   if (S < 0) {
      block(); // Add this process to S's waiting queue    }}
```

b. signal(S) (or V(S)): This operation increments the value of S. If there are any threads blocked in the semaphore's waiting queue (meaning S was negative or zero), one of them is unblocked and moved to the ready queue.

```
signal(S) {
   S++;
   if (S <= 0) {
      wakeup(); // Remove a process P from S's waiting queue    }}
```

A binary semaphore can only take the values 0 and 1 and is used for mutual exclusion. A counting semaphore can take any integer value and is used to control access to a resource with a finite number of instances or to coordinate threads based on the number of a particular event.

Application to Producer–Consumer Problem

The Producer-Consumer problem is a classic synchronization paradigm [20]. This is where one or more producer threads generate data items and place them into a shared buffer and also one or more consumer threads remove items from the buffer and process them. The problem requires two forms of synchronization:

a. Mutual Exclusion: The buffer is a shared data structure; therefore, puts and gets must be performed atomically to prevent race conditions.

b. Event Signaling: A consumer must not try to remove an item from an empty buffer, and a producer must not try to add an item to a full buffer.

Semaphore Solution

A semaphore-based solution elegantly handles both requirements using three semaphores:

a. mutex: A binary semaphore (initialized to 1) that enforces mutual exclusion for accesses to the buffer.

b. full: A counting semaphore (initialized to 0) that counts the number of filled slots in the buffer.

c. empty: A counting semaphore (initialized to BUFFER_SIZE) that counts the number of empty slots in the buffer.

The pseudocode for the producer and consumer is as follows:

Producer Thread:

```
while (true) {
   // produce an item
   item = produce_item();
   wait(empty);    // Wait for an empty slot (decrements empty count)
   wait(mutex);    // Acquire lock to the buffer
   // Critical Section: add item to buffer
   insert_item(item);
   signal(mutex);  // Release lock on the buffer
   signal(full);   // Signal that a slot is now full (increments full count)}
```

Consumer Thread:

```
while (true) {
   wait(full);     // Wait for a full slot (decrements full count)
   wait(mutex);    // Acquire lock to the buffer
   // Critical Section: remove item from buffer
   item = remove_item();
   signal(mutex);  // Release lock on the buffer
   signal(empty);  // Signal that a slot is now empty (increments empty count)
```

```
// consume the item

consume_item(item);}
```

Relation to Dekker/Peterson

The mutex semaphore in this solution directly fulfills the singular role that Dekker's and Peterson's algorithms were designed for: ensuring mutual exclusion for the critical section (the buffer). The classical algorithms, however, have no inherent mechanism to handle the additional coordination of waiting for a non-full or non-empty buffer. The full and empty semaphores demonstrate the power of semaphores to go beyond mutual exclusion and provide a general signaling mechanism between threads, a capability absent from the earlier software solutions.

Application to Bounded Buffer Problem

The Bounded Buffer problem is a more precise name for the Producer-Consumer problem [21] when the buffer has a fixed size, N. The key addition is that the solution must actively block producers when the buffer is full and block consumers when the buffer is empty, rather than simply having them check and retry.

Semaphore-based Solution

The solution is identical to the one presented above. The counting semaphores full and empty elegantly manage the blocking conditions for both producers and consumers. The initial value of empty is N (the buffer size). Each time a producer wishes to add an item, it performs a wait(empty). If empty is zero (meaning the buffer is full), the producer is blocked. Similarly, a consumer performing wait(full) will block if no items are available. The signal operations performed by the complementary thread are what unblock the waiting threads.

Relation to Dekker/Peterson

This problem highlights a fundamental limitation of Dekker's and Peterson's algorithms. They are purely mutual exclusion algorithms. They could be used to protect the buffer itself, ensuring that only one thread inserts or removes at a time. However, they provide no facility for a producer to efficiently wait for a "buffer not full" condition. One would have to resort to a busy-waiting loop that repeatedly checks for space, which is highly inefficient. The semaphore solution, by combining mutual exclusion with conditional blocking, provides a complete and efficient solution to the problem. This demonstrates why semaphores and higher-level abstractions have superseded the classical algorithms in practical system design.

Application to Readers–Writers Problem

The Readers-Writers problem models access to a shared database, such as a file. Multiple reader threads can access the database simultaneously without interfering with each other, as they do not modify the data. However, any writer thread must have exclusive access to the database, both with respect to other writers and to readers. This leads to several variants, primarily distinguished by their policy for preventing starvation. The "first readers-writers problem" prioritizes readers, which can lead to writer starvation. The "second readers-writers problem" prioritizes writers.

Semaphore-based Solution

A common solution for the first readers-writers problem uses two semaphores [20], [22] and an integer counter:

a. rw_mutex: A binary semaphore used by writers and the first/last reader to ensure mutual exclusion for writers.

b. mutex: A binary semaphore used to protect the critical section where the read counter (read_count) is updated.

c. read_count: An integer that tracks the number of active readers.

Writer Thread:

```
while (true) {
   wait(rw_mutex);     // Acquire exclusive access for writing
   // Critical Section for Writing
   // ... perform writing ...
   signal(rw_mutex);   // Release exclusive access}
```

Reader Thread:

```
while (true) {
   wait(mutex);              // Acquire lock to update read_count
   read_count++;
   if (read_count == 1) {     // If I am the first reader...
      wait(rw_mutex);        // ...lock out the writers    }
   signal(mutex);            // Release lock on read_count
   // Critical Section for Reading
   // ... perform reading ...
   wait(mutex);              // Acquire lock to update read_count
   read_count--;
   if (read_count == 0) {     // If I am the last reader...
      signal(rw_mutex); }      // ...unlock for writers
   signal(mutex);            // Release lock on read_count}
```

Relation to Dekker/Peterson

The rw_mutex semaphore acts as the gatekeeper for exclusive access. Its behavior is conceptually similar to the turn variable in Peterson's algorithm that arbitrates between two competing processes. Here, it arbitrates between the collective group of readers and the writers. The "turn" is given to the first reader that arrives, which blocks writers until all readers are finished. While this solution can starve writers, more complex solutions using additional semaphores can enforce fairness. This demonstrates how semaphores can be composed to build sophisticated synchronization policies that are far more complex than what the two-process classical algorithms could express.

Table 2 provides a structured summary of the synchronization mechanisms from classical software algorithms to hardware primitives and semaphore-based abstractions. This comparison is informed by multiple recent studies in the synchronization literature. [16] identify the core trade-offs between hardware-based and software-based synchronization, noting that hardware primitives offer atomicity guarantees but require careful consideration of memory ordering on modern processors. [23] provide a broader survey of concurrency and

synchronization tools, cataloguing practical scenarios where each technique proves most effective. [24] offer a theoretical grounding for semaphore-based synchronization, reinforcing the move from ad hoc algorithms to structured primitives as the foundational transition in synchronization design. Overall, these studies affirm that the choice of synchronization mechanism depends critically on the application context, hardware architecture, and required performance characteristics.

Technique	Advantages	Disadvantages	Primary Use Cases
Dekker's Algorithm [2], [15]	Pure software solution requiring no hardware support; provably satisfies mutual exclusion, progress, and bounded waiting under a sequentially consistent model	Relies on busy waiting, wasting CPU cycles; limited to two processes; complex logic prone to implementation errors; fails on relaxed-memory hardware without fences	Historical study and formal verification research; academic demonstrations of software-only mutual exclusion
Peterson's Algorithm [3], [17]	Simpler and more elegant than Dekker's; software-only with no hardware dependency; formally verifiable with minimal overhead; ideal for teaching	Busy waiting imposes CPU overhead; strictly two-process; incompatible with modern processor memory models without explicit barriers [18]	Operating systems education; mutual exclusion coursework; formal methods research and property verification
Test-and-Set (TAS) [5], [16]	Simple atomic hardware primitive; easy to implement as a spinlock; guaranteed atomicity by the processor	High memory bus contention under multi-core load; busy waiting wastes CPU; no fairness guarantee; starvation possible	Low-contention spinlocks; kernel-level bootstrap synchronization where OS blocking is unavailable
Compare-and-Swap (CAS) [7], [16]	Versatile atomic primitive enabling lock-free algorithm design; foundation of non-blocking data structures; used in Java Atomics and C++ std::atomic	Susceptible to ABA problem; still involves busy-wait retry loops; complex correctness reasoning in lock-free designs	Lock-free and wait-free data structures; atomic counters and accumulators; concurrent collections in modern runtimes
Binary Semaphore [5], [24]	Eliminates busy waiting through OS-managed blocking; supports mutual exclusion and inter-thread signaling; efficient CPU usage when threads are waiting	Risk of deadlock if signal() is omitted or misplaced; no ownership tracking; programmer discipline required to avoid misuse [24]	Critical section protection; producer-consumer coordination; OS kernel synchronization and resource guarding
Counting Semaphore [5], [23], [24]	Controls concurrent access to multiple resource instances; flexible signaling model;	More complex to reason about than binary semaphores; potential for starvation if signal() is	Thread pool management; bounded buffers; readers-writers problems; rate-

Technique	Advantages	Disadvantages	Primary Use Cases
	naturally expresses resource pool management	never called; requires careful initialization	limiting access to shared resource pools
Mutex / Lock [10], [18]	Ownership tracking prevents incorrect unlocking; simpler API than raw semaphores (lock/unlock); language-level support in Java, Python, C++	Deadlock if a thread exits without releasing; priority inversion risk in real-time systems; not suitable for signaling between threads	General-purpose mutual exclusion in application code; protecting shared objects in object-oriented concurrent programs
Monitors and Condition Variables [18], [22]	Encapsulates synchronization within the object; structured wait/notify model reduces programmer error; supports complex conditional blocking	Requires language or runtime support; can introduce overhead from spurious wakeups; over-engineering risk for simple synchronization needs	High-level concurrent object-oriented programming; Java synchronized blocks; Python threading. Condition; real-time scheduling constructs

Table 2: Comparative Summary of Synchronization Techniques: Advantages, Disadvantages, and Use Cases

Modern OS Approaches

While semaphores are still used in kernel development and some low-level programming, modern high-level concurrent programming often employs even more structured abstractions to reduce the chance of programmer error (e.g., forgetting to call signal).

a. Monitors: A monitor is a programming language construct that bundles shared data [22] with the procedures that operate on it, along with the necessary synchronization mechanisms. Only one thread may be active within a monitor at a time. This encapsulates synchronization within the object itself.

b. Condition Variables: Used within monitors, condition variables (wait, signal, broadcast) allow threads to wait for a specific condition to become true, providing more granular control than a simple semaphore.

c. Locks/Mutexes: Many programming languages provide explicit Lock or Mutex objects [10], [25] (e.g., in Java, Python, C++). These are essentially binary semaphores with a more intuitive API (lock()/unlock()).

d. Synchronized Methods/Blocks: Languages like Java and C# offer the synchronized keyword, which allows the compiler to automatically insert lock acquisition and release code, making critical section management less error-prone.

All these modern constructs are built upon the same fundamental principles established by Dekker, Peterson, and Dijkstra [1], [26]. They are the practical, evolved descendants of these pioneering synchronization techniques. However, the evolution of synchronization mechanisms reflects broader trends observed across previous literature. [16] provide a comprehensive analysis of synchronization mechanisms in operating systems, affirming that modern OS primitives such as mutexes, condition variables, and spinlocks trace their theoretical lineage directly to the classical algorithms of Dekker, Peterson, and Dijkstra. Their analysis highlights that contemporary operating systems employ hybrid synchronization strategies that combine short spinning phases

with blocking, a design that mitigates both the CPU waste of pure busy waiting and the context-switching overhead of immediate blocking.

Similarly, [18] observe that multi-threaded application performance is critically dependent on selecting appropriate synchronization primitives: over-synchronization leads to unnecessary contention and reduced throughput, while under-synchronization risks data races and non-deterministic behavior. They emphasize that thread scheduling policies must be co-designed with synchronization mechanisms to avoid priority inversion and ensure bounded latency in real-time systems. [23] further extend this perspective, identifying concurrency bugs including deadlocks, livelocks, and data races as persistent sources of software failure whose root causes lie in the misuse or misunderstanding of the very primitives built upon these classical foundations. Their survey of detection tools and mitigation strategies underscores the practical importance of the theoretical properties namely mutual exclusion, progress, and bounded waiting which was first rigorously established by Dekker and Peterson and formalized through semaphore theory.

Experimental Analysis and Performance Evaluation

To complement the theoretical analysis presented, a simulation framework was developed in Python. The experiments were conducted on a Linux-based system running kernel version 6.12.54-linuxkit on an ARM64 (AArch64) architecture. The machine was equipped with 12 physical processor cores and 8 GB of available RAM. All simulations were executed within this environment, with no hardware multithreading beyond the available physical core count. The core simulation logic comprises a SimulationEngine responsible for executing each algorithm, a MetricsTracker recording CPU utilization, execution time, and average wait time per thread, and separate modules for chart generation (Matplotlib/Seaborn) and behavioral flow diagrams (Graphviz). Figure 2 illustrates the architecture of the simulation framework.

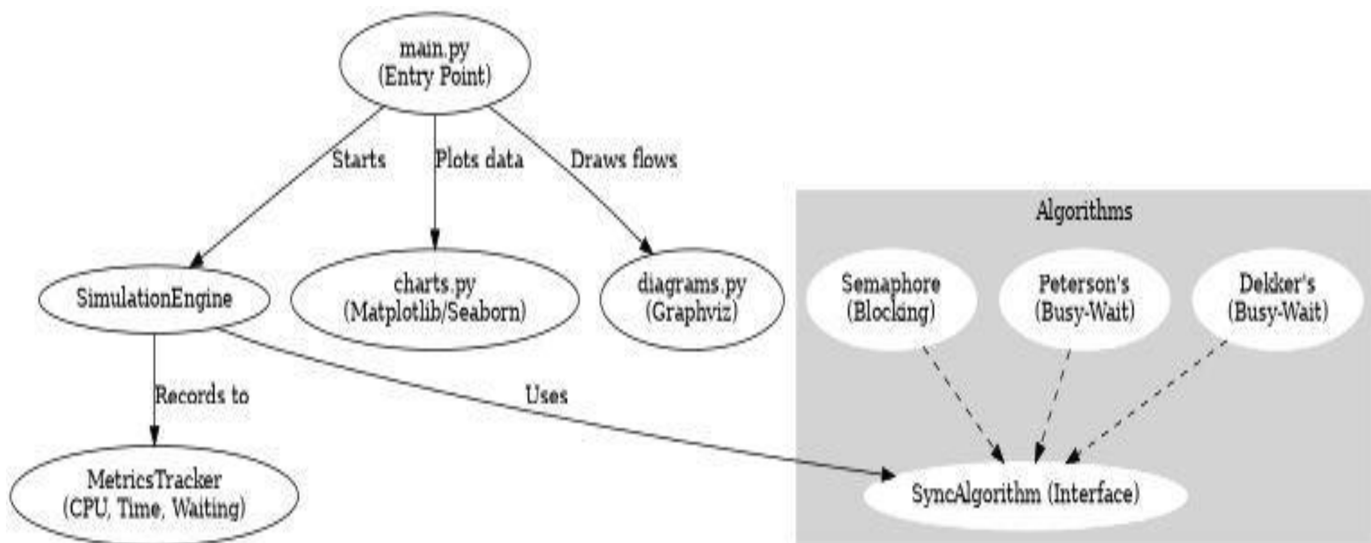

Figure 2: Architecture of the Simulation Framework

CPU Utilization

Figure 3 shows the percentage of CPU utilization actively used by the Python threads during the wait states. Dekker's algorithm recorded the highest utilization at approximately 51%, reflecting the cost of its more complex flag-yielding spin logic. Peterson's algorithm recorded approximately 27%, benefiting from its simpler single-condition wait. The semaphore-based approach recorded approximately 4%, consistent with the blocking behavior depicted in Figure 1. These results directly corroborate the theoretical limitation that busy waiting imposes a measurable and avoidable CPU overhead even at the minimal two-thread scale [15], [16]. While Dekker's and Peterson's algorithms drastically spike CPU usage, this is why "busy waiting" software algorithms are rarely used in modern operating systems as they burn battery and processing power just asking "is it my turn yet?". Semaphores on the other hand, efficiently block and yield the CPU back to the OS.

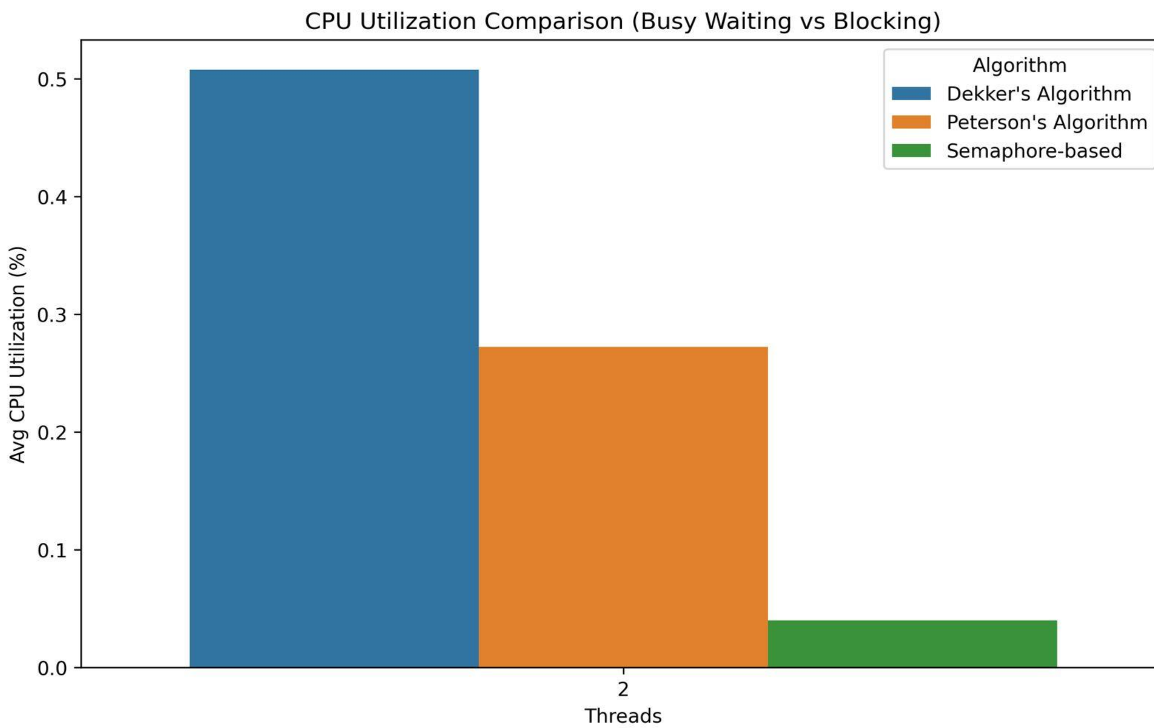

Figure 3: Average CPU Utilization Comparison (Busy Waiting vs. Blocking) at 2 Threads

Execution Time

Figure 4 reports total execution time spent completely suspended/blocked before it was allowed to enter its critical section. Dekker's algorithm completed in approximately 2.67 seconds, Peterson's in approximately 3.01 seconds, and the semaphore-based solution in approximately 2.07 seconds.

The semaphore approach achieves the lowest execution time by eliminating spinning entirely, allowing threads to be scheduled productively while others hold the lock [5], [24]. Semaphores therefore tend to have smoother and more predictable wait time because the OS handles the queue, compared to Dekker's which relies purely on timing and thread context switching.

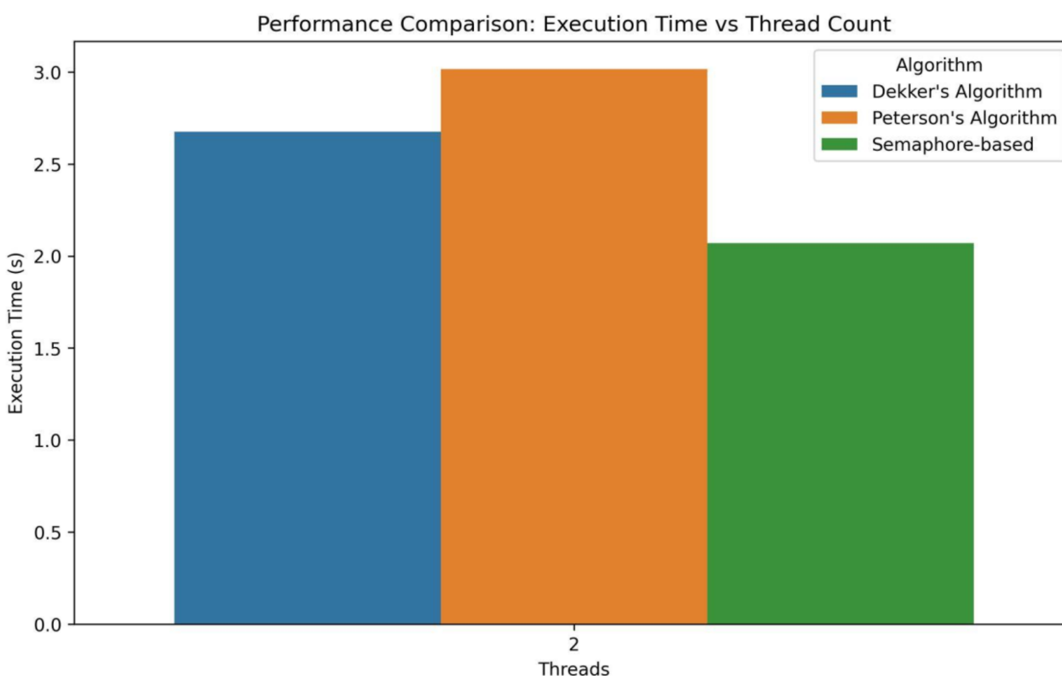

Figure 4: Performance Comparison — Execution Time vs. Thread Count

Average Wait Time per Thread

Figure 5 shows the average wait time experienced per thread. Dekker's algorithm recorded approximately 1.63 seconds, Peterson's approximately 1.91 seconds, and the semaphore-based solution approximately 0.51 seconds which is a reduction of roughly 69% and 73% respectively. The elevated wait time under Peterson's algorithm reflects the overhead of its polite turn-yielding during contention. The semaphore's blocking mechanism suspends waiting threads entirely, so recorded wait time reflects only OS scheduling latency rather than active spinning [5], [18].

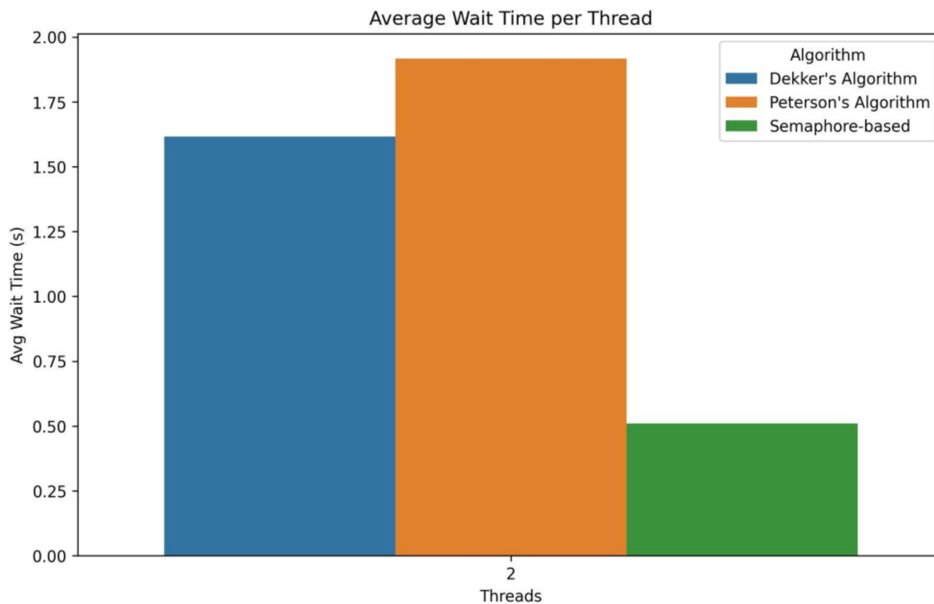

Figure 5: Average Wait Time per Thread at 2 Threads

Scalability of Semaphore-Based Synchronization

Since Dekker's and Peterson's algorithms are architecturally limited to two processes, scalability analysis was conducted exclusively for the semaphore-based implementation. Figure 6 shows execution time growth as thread count increases from 2 to 16. The near-linear growth observed from approximately 2.07 seconds at 2 threads to 16.5 seconds confirms that semaphores scale gracefully with thread count, a property that classical algorithms cannot provide by design [1], [15], [23].

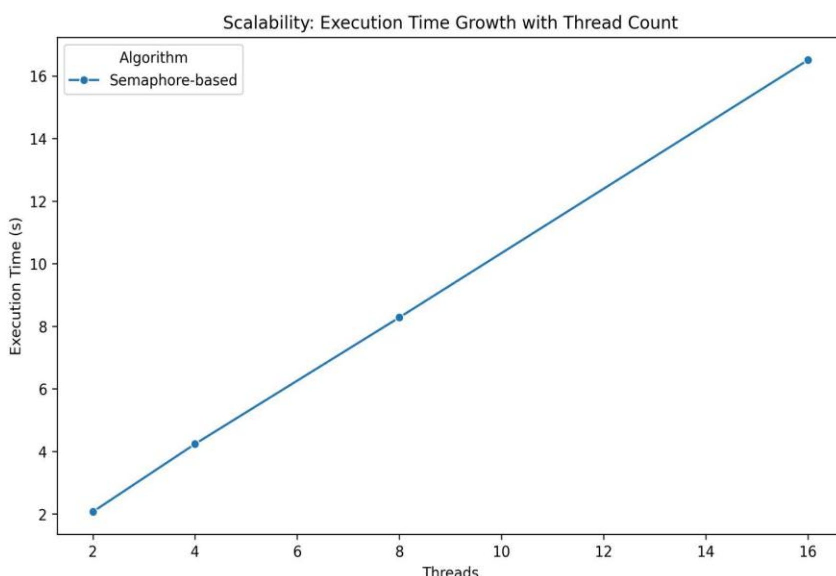

Figure 6: Scalability — Execution Time Growth with Thread Count (Semaphore-Based)

Summary of Experimental Findings

Across all three performance metrics - CPU utilization, execution time, and average wait time, the semaphore-based approach consistently outperforms both classical algorithms at the two-thread baseline. The CPU utilization disparity (51% vs. 27% vs. 4%) is the most striking finding, as it directly quantifies the cost of busy waiting. The scalability analysis further confirms that semaphores are not merely theoretically superior but remain practically efficient as concurrency scales in a dimension entirely inaccessible to Dekker's and Peterson's designs. These findings align with the assessments of [16] and [23], who identify busy-waiting overhead and the inability to scale beyond two processes as the defining practical limitations of classical mutual exclusion algorithms in modern computing environments.

CONCLUSION

The evolution of thread synchronization, from the intricate logic of Dekker's algorithm to the elegant simplicity of Peterson's and finally to the powerful abstraction of semaphores, illustrates a fundamental trend in computer science: the move from complex, specific solutions to general, manageable primitives. Dekker and Peterson provided the essential proof that reliable software-only mutual exclusion was possible, establishing the core properties that any synchronization mechanism must satisfy. Their work remains a vital part of computer science education, teaching the subtle challenges of concurrent programming. Dijkstra's semaphores built upon this foundation, abstract away the complexity and inefficiency of busy waiting. By providing a structured mechanism for both mutual exclusion and inter-thread signaling, semaphores became a versatile tool for solving a wide array of practical synchronization problems, from bounded buffers to reader-writer locks. The principles of atomic operations, waiting, and signaling embedded in these early algorithms are not historical relics; they form the conceptual bedrock upon which all modern synchronization primitives from mutexes and condition variables to concurrent collections and parallel frameworks are built. As concurrency becomes ever more pervasive, understanding this foundational history is key to designing and implementing correct, efficient, and robust software systems.

REFERENCES

1. M. Herlihy and N. Shavit, The Art of Multiprocessor Programming. San Francisco, CA, USA: Morgan Kaufmann, 2012.
2. T. J. Dekker, "Over de sequentialiteit van procesbeschrijvingen (On the sequentiality of process descriptions)," Mathematical Centrum Amsterdam, 1965.
3. G. L. Peterson, "Myths about the mutual exclusion problem," Information Processing Letters, vol. 12, no. 3, pp. 115–116, Jun. 1981.
4. H. M. Deitel, P. J. Deitel, and D. R. Choffnes, Operating Systems, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2004.
5. E. W. Dijkstra, "Cooperating sequential processes," Technische Hogeschool Eindhoven, 1965.
6. D. Dice, "Implementing fast Java monitors with relaxed locks," in Proc. JVM Research and Technology Symposium, 2001, pp. 79–90.
7. Intel Corporation, Intel® 64 and IA-32 Architectures Software Developer's Manual, 2023.
8. W. Stallings, Operating Systems: Internals and Design Principles, 9th ed. Hoboken, NJ, USA: Pearson, 2018.
9. M. L. Scott, Programming Language Pragmatics, 4th ed. San Francisco, CA, USA: Morgan Kaufmann, 2015.
10. B. Goetz et al., Java Concurrency in Practice. Boston, MA, USA: Addison-Wesley, 2006.
11. L. Lamport, "The mutual exclusion problem: Part I—A theory of interprocess communication," Journal of the ACM, vol. 33, no. 2, pp. 313–326, Apr. 1986.
12. L. Lamport, "The mutual exclusion problem: Part II—Statement and solutions," Journal of the ACM, vol. 33, no. 2, pp. 327–348, Apr. 1986.
13. P. Brinch Hansen, Operating System Principles. Englewood Cliffs, NJ, USA: Prentice Hall, 1973.
14. N. G. Leveson, "The evolution of layered synchronization," Software: Practice and Experience, vol. 16, no. 5, pp. 437–454, May 1986.

15. L. Nigro, F. Cicirelli, and F. Pupo, "Modeling and Analysis of Dekker-Based Mutual Exclusion Algorithms," Computers, vol. 13, no. 6, p. 133, 2024. doi: https://doi.org/10.3390/computers13060133
16. O. Kode and T. Oyemade, "Analysis of Synchronization Mechanisms in Operating Systems," arXiv preprint arXiv:2409.11271, Sep. 2024. doi: https://doi.org/10.48550/arXiv.2409.11271
17. L. Nigro and F. Cicirelli, "Property Assessment of Peterson's Mutual Exclusion Algorithms," Applied Computational Intelligence, vol. 4, no. 1, pp. 66–92, Jan. 2024. doi: https://doi.org/10.3934/aci.2024005
18. A. Gupta, A. Arora, and M. Kaur, "Thread Scheduling and Synchronization for Multi-Threaded Applications," Journal of Software Engineering and Simulation, vol. 10, no. 5, pp. 01–06, 2024.
19. J. H. Anderson and Y.-J. Kim, "Adaptive mutual exclusion with local spinning," in Proc. 14th International Symposium on Distributed Computing, 2000, pp. 1–15.
20. P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with 'readers' and 'writers'," Communications of the ACM, vol. 14, no. 10, pp. 667–668, Oct. 1971.
21. M. L. Scott, Programming Language Pragmatics, 4th ed. San Francisco, CA, USA: Morgan Kaufmann, 2015.
22. C. A. R. Hoare, "Monitors: An operating system structuring concept," Communications of the ACM, vol. 17, no. 10, pp. 549–557, Oct. 1974.
23. G. Khandelwal, S. Sonwane, and S. Ware, "Concurrency and Synchronization: Detection, Reasons, Tools and Applications," International Journal of Scientific Research and Engineering Trends, vol. 10, no. 6, pp. 1–8, Nov.–Dec. 2024.
24. V. Sree Dharshni, V. S. Akshaya, M. Sujithra, and A. D. Chitra, "A Theory of Synchronisation Using Semaphores," International Journal of Advanced Engineering and Management (IJAEM), vol. 2, no. 9, pp. 256–263, 2020.
25. S. Oaks and H. Wong, Java Threads, 3rd ed. Sebastopol, CA, USA: O'Reilly Media, 2004.
26. M. A. Arjomandi, "A comprehensive survey of parallel mutual exclusion algorithms," ACM Computing Surveys, vol. 54, no. 9, pp. 1–33.