Lecture 12:
Thread-Safe Hash Table

Logistics

- Point Solutions App
  - Session ID: **database**
- Exam
  - Max score will be lowered (curved)
  - 6422 section: 200 points to **160** points (AVG: 133, MAX: 175)
  - 4420 section: 165 points to **130** points (AVG: 110, MAX: 146)
  - You can see your graded paper during office hours

Logistics

- One-page project proposals due on **Oct 12** (extra credit)
  - Rubric: 5% for most submissions, 10% for a small subset of submissions
  - Subjective evaluation
  - Topic should be related to database internals, preferably C++
  - In-class presentations at the end of the semester
  - Top three projects (picked by the students) will receive a prize!

Recap

- Hash Tables
- Hash Function
- Deletion and Position Tracking
- Quadratic Probing
- Double Hashing

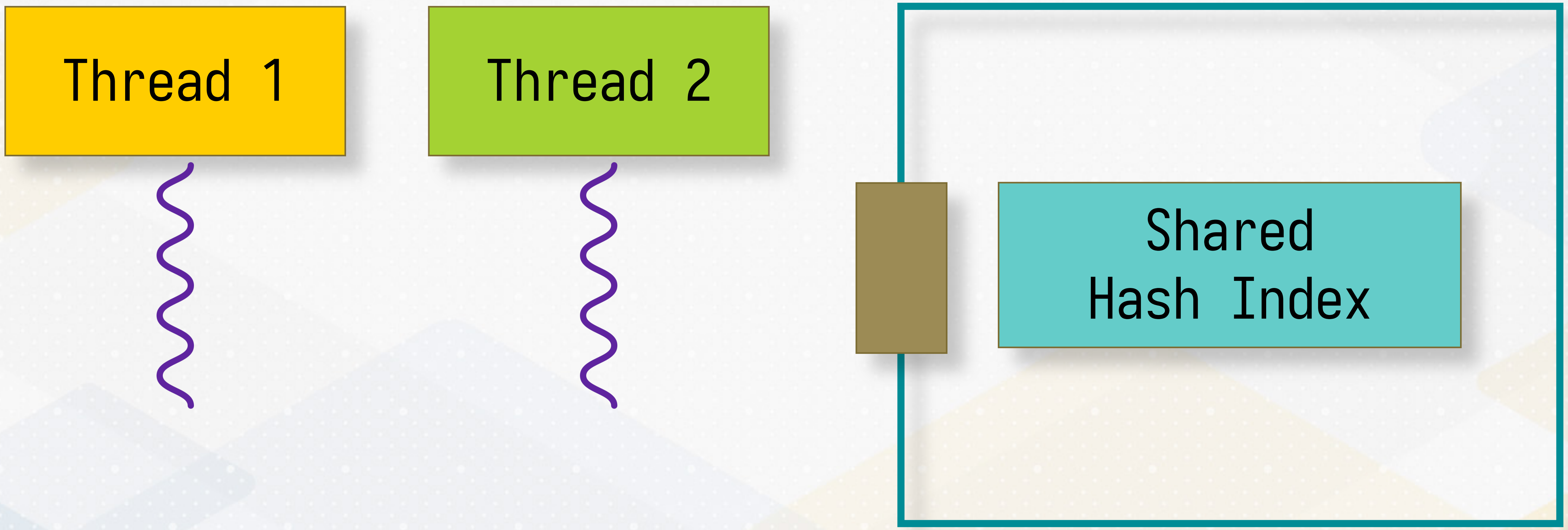
Lecture Overview

- Parallel Index Construction
- Fine-Grained Locking
- Shared Mutex
- Simulation Framework

Parallel Index Construction

Parallel Index Construction

With multi-core CPUs, **parallelizing index construction** offers a significant performance boost by **distributing** the workload across **multiple threads**.

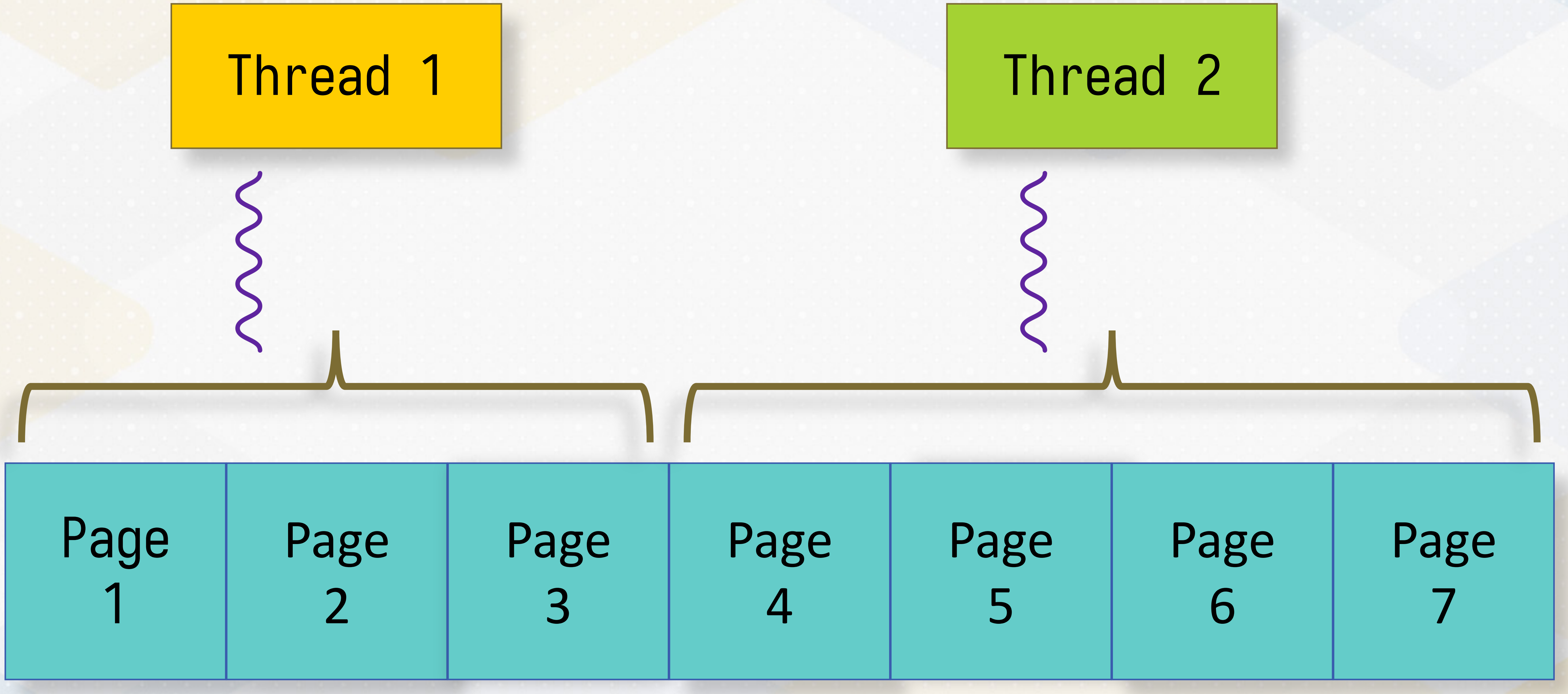


Page Assignment

Divide the total number of pages (num_pages) by the number of available threads (num_threads), assign each thread a specific range of pages to process.

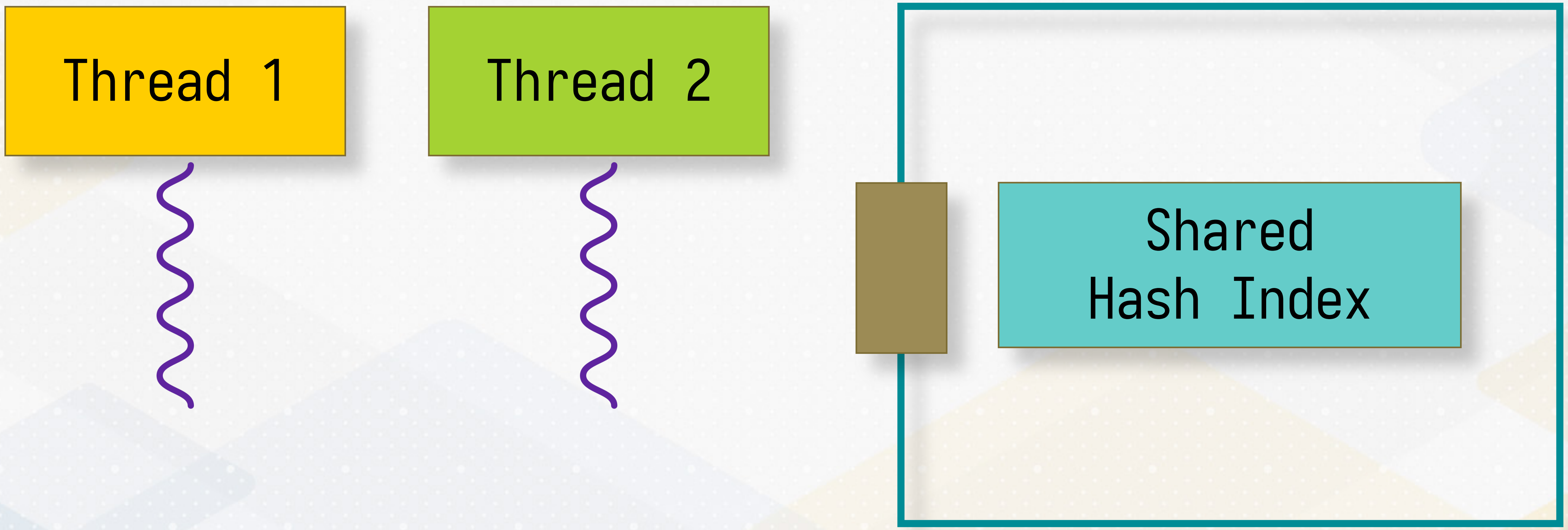
```
void parallelProcessPages(size_t num_threads = 5) {
    auto num_pages = buffer_manager.getNumPages();
    size_t pages_per_thread = num_pages / num_threads;
    std::vector<std::thread> threads;
    for (size_t i = 0; i < num_threads; i++) {
        size_t start_page = i * pages_per_thread;
        size_t end_page =...; // Last thread gets any remaining pages
        threads.emplace_back(&BuzzDB::processPageRange, this, start_page, end_page);
    }
...
}
```

Page Assignment



Thread Safety

With **parallel index construction**, the challenge is that **concurrent operations** on the index by multiple threads may lead to **inconsistent state**.



Thread Safety

Thread 1: Insert (15, Fig)   Thread 2: Insert (25, Pear)

0	1	2	3	4	5	6	7	8	9
					25				
					Pear				

HASH(KEY) = KEY % 10

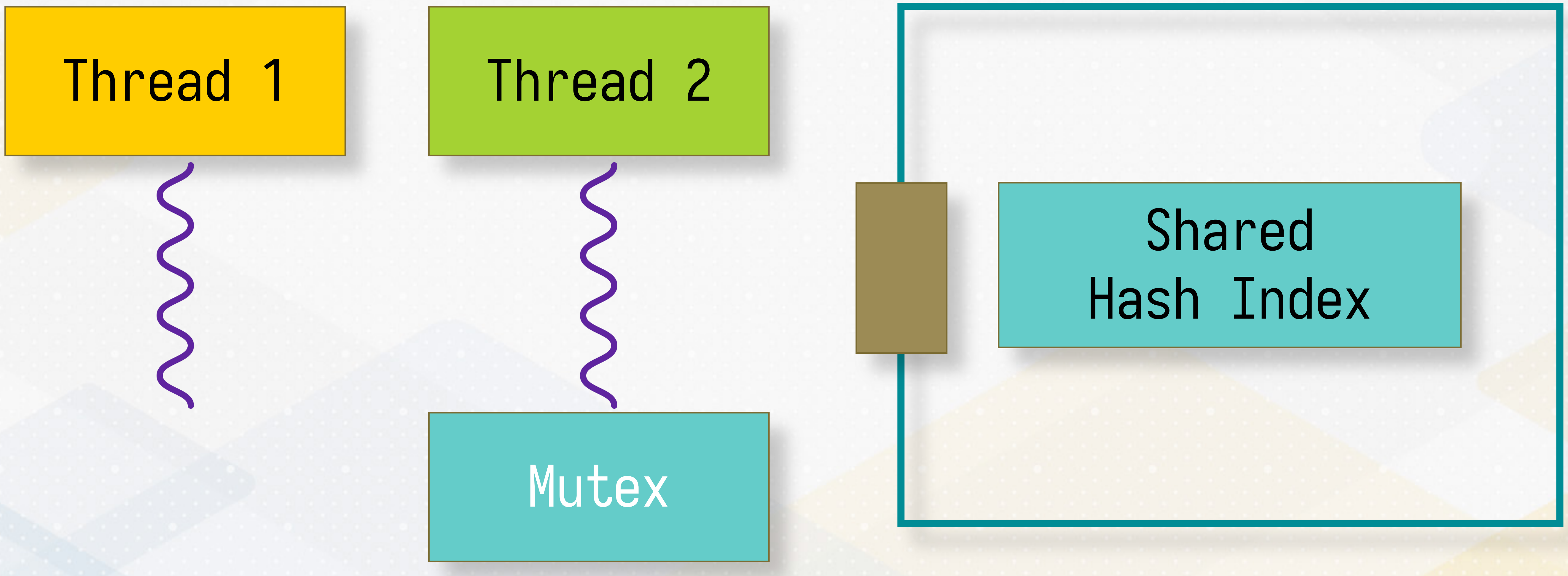
Race Condition

	Thread 1	Thread 2
1	HASH key 15	
2	PROBE slot 5	
3		HASH key 25
4		PROBE slot 5
5	STORE key 15 in slot 5	
6		STORE key 25 in slot 5

Time

std::mutex

Mutex "serializes" access to the shared index structure.



std::lock_guard

lock_guard automatically acquires a lock on creation and releases it on destruction.

```
class HashIndex {
    private:
    mutable std::mutex indexMutex; // Mutex for thread-safe access

    void insertOrUpdate(int key, int value) {
    std::lock_guard<std::mutex> guard(indexMutex); // RAII-style mutex
    management
    // Perform thread-safe update on the index
    }
};
```

Thread Safety with Mutex

	Thread 1	Thread 2
1	Lock Index Mutex	
2	HASH key 15	
3	PROBE slot 5	WAIT on Index Mutex
4	STORE key 15 in slot 5	WAIT on Index Mutex
5	Unlock Index Mutex	WAIT on Index Mutex
6		Lock Index Mutex
7		HASH key 25
8		PROBE slot 5 and see key 15
9		PROBE slot 6
10		STORE key 25 in slot 6
11		Unlock Index Mutex

Time

Thread Safety with Mutex

Time	Thread 1	Thread 2
1		Lock Index Mutex
2		HASH key 25
3	WAIT on Index Mutex	PROBE slot 5
4	WAIT on Index Mutex	STORE key 25 in slot 5
5	WAIT on Index Mutex	Unlock Index Mutex
6	Lock Index Mutex	
7	HASH key 15	
8	PROBE slot 5 and see key 25	
9	PROBE slot 6	
10	STORE key 15 in slot 6	
11	Unlock Index Mutex	

Order of Thread Execution

Thread 1 → Thread 2

0	1	2	3	4	5	6	7	8	9
					15	25			
					Fig	Pear			

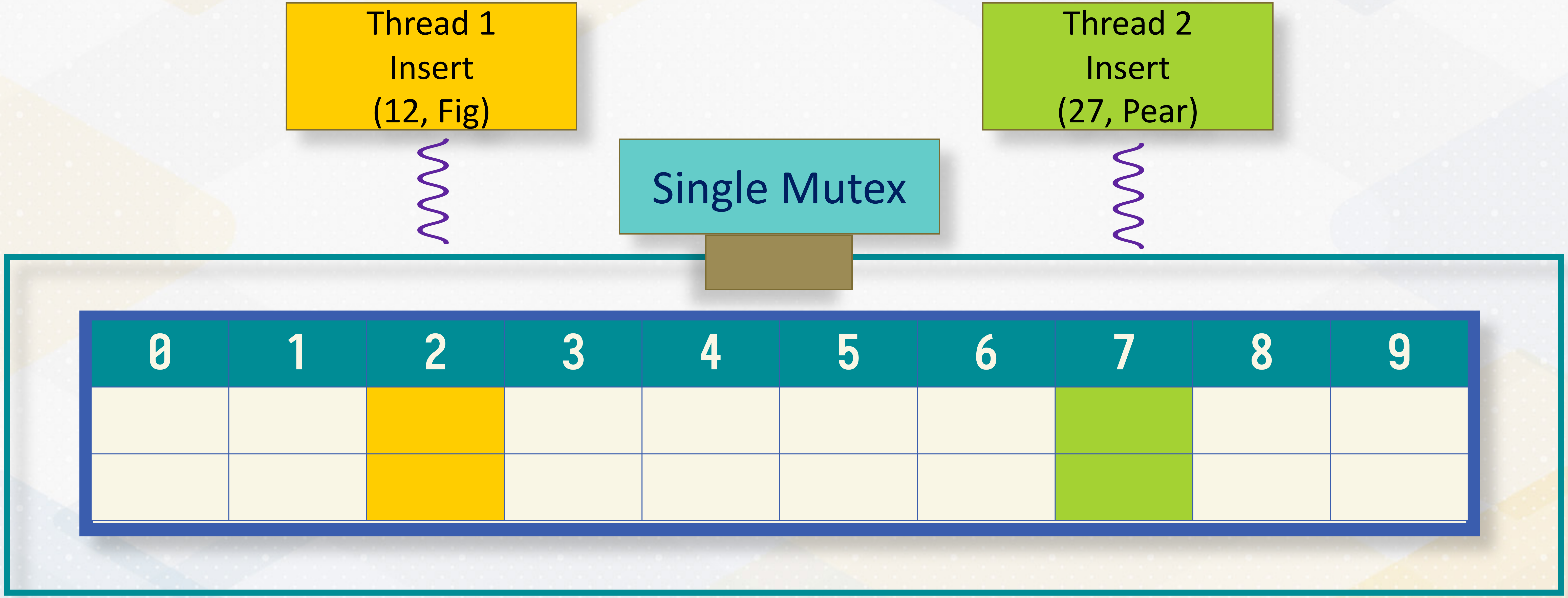
Thread 1 → Thread 2

0	1	2	3	4	5	6	7	8	9
					25	15			
					Pear	Fig			

Fine-Grained Locking

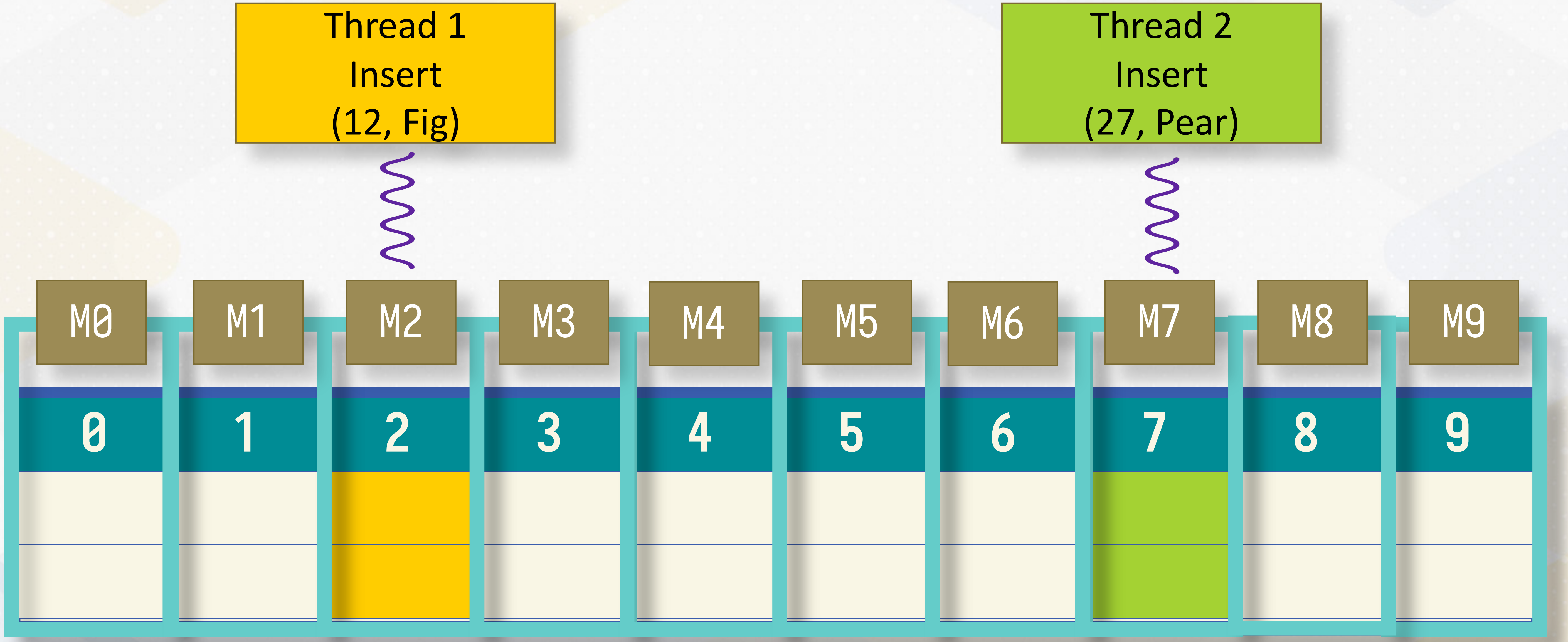
Limited Concurrency

Using a single lock for the entire hash table severely limits parallelism.



Fine-Grained Locking

Using a lock for each slot increases parallelism.



Fine-Grained Locking

Each slot in the hash table has an associated mutex.

```
std::vector<std::unique_ptr<std::mutex>> mutexes;
void insertOrUpdate(int key, int value) {
size_t index = hashFunction(key); // Determine the slot index for the
key
do {
// Lock only the mutex for the specific slot
std::lock_guard<std::mutex> lock(*mutexes[index]);
// Attempt to insert or update the slot
if (conditions_met) { // Insert or update logic }
// Handle collision and calculate next slot index
} while (not_inserted);
}
```

vector<mutex> vs vector<unique_ptr<mutex>>

Vector<element> requires element to be movable.

Mutex NOT MOVABLE	unique_ptr<mutex> MOVABLE

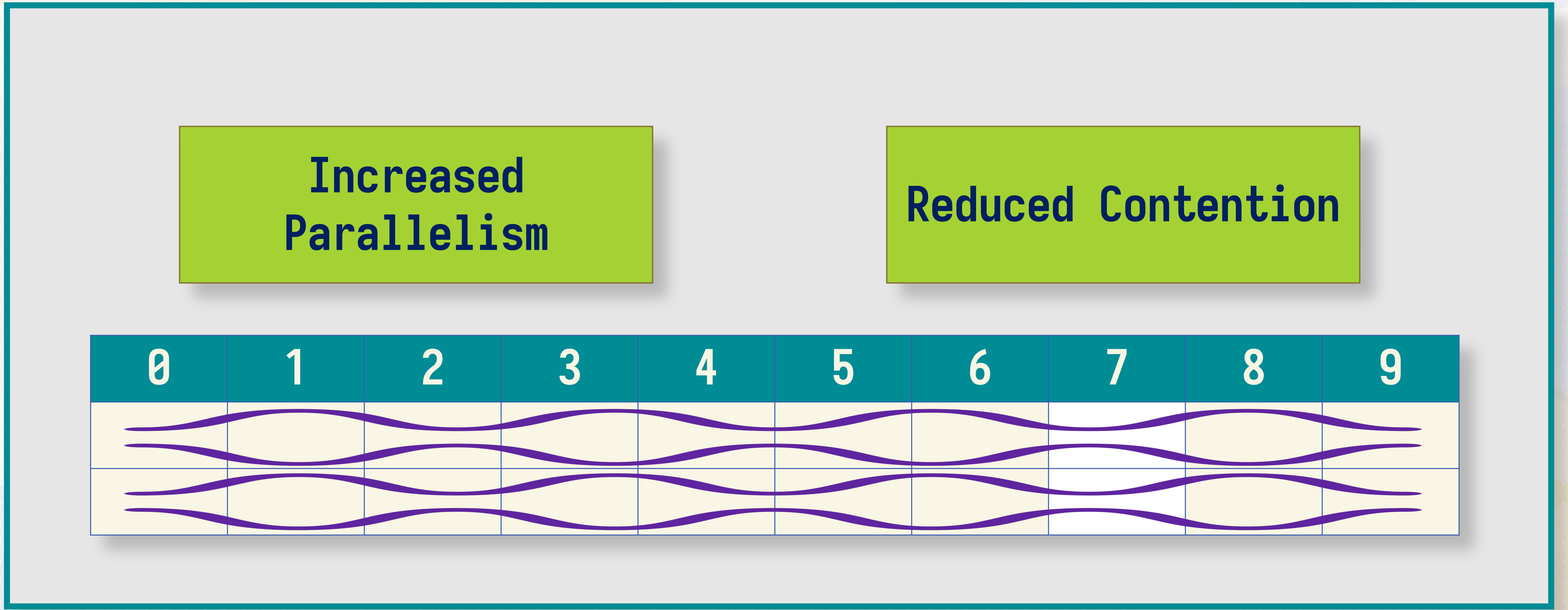
Fine-Grained Locking

By locking only the specific slot being accessed, rather than the entire hash table, fine-grained locking enables higher concurrency.

```
int getValue(int key) const {
size_t index = hashFunction(key);
size_t originalIndex = index;

do {
// Lock only the specific slot's mutex
std::lock_guard<std::mutex> lock(*mutexes[index]);
// Check if the key is inside the slot or not
// Calculate next slot index
} while (index != originalIndex);
}
```

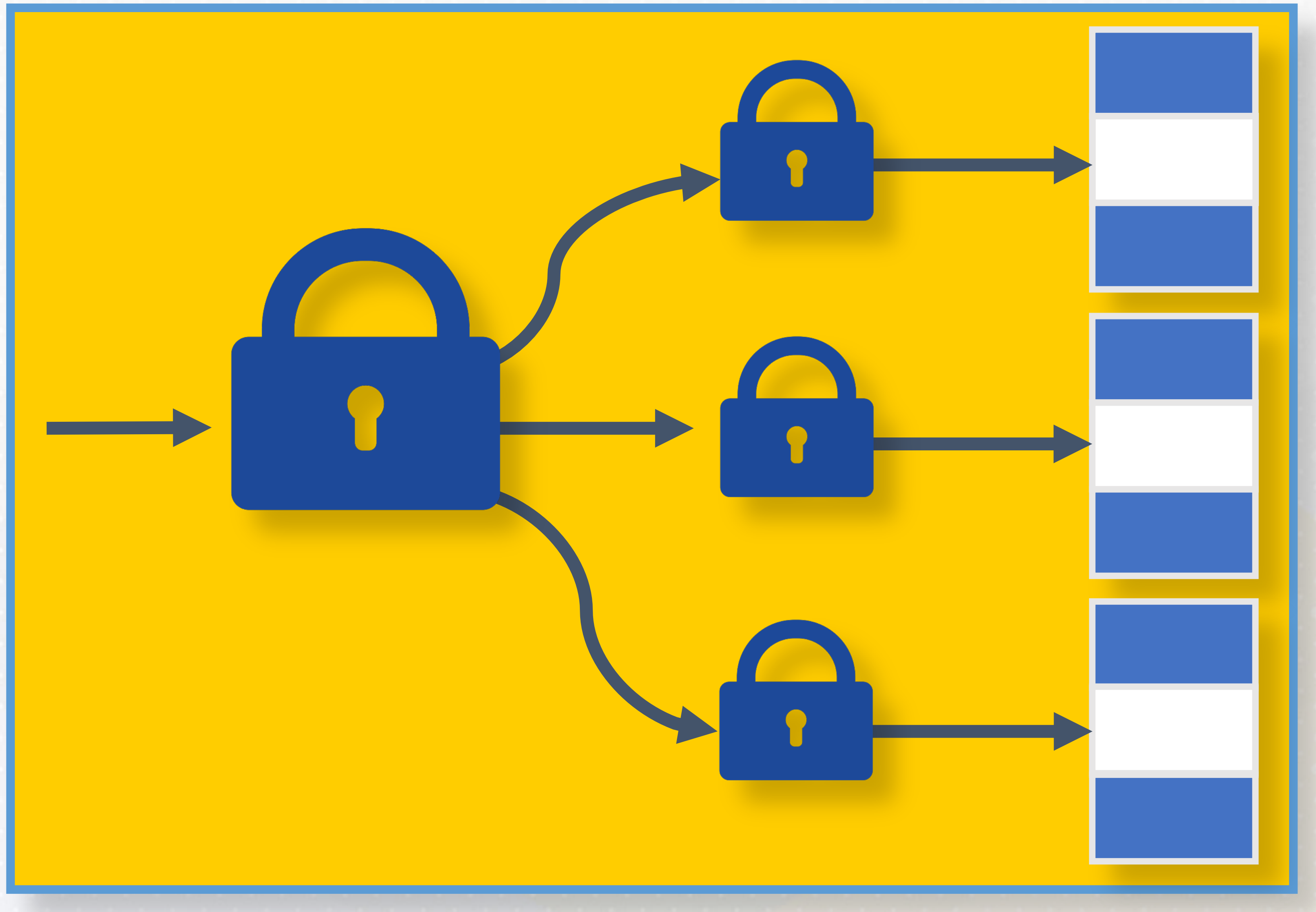
Benefits of Fine-Grained Locking



Limitations of Fine-Grained Locking

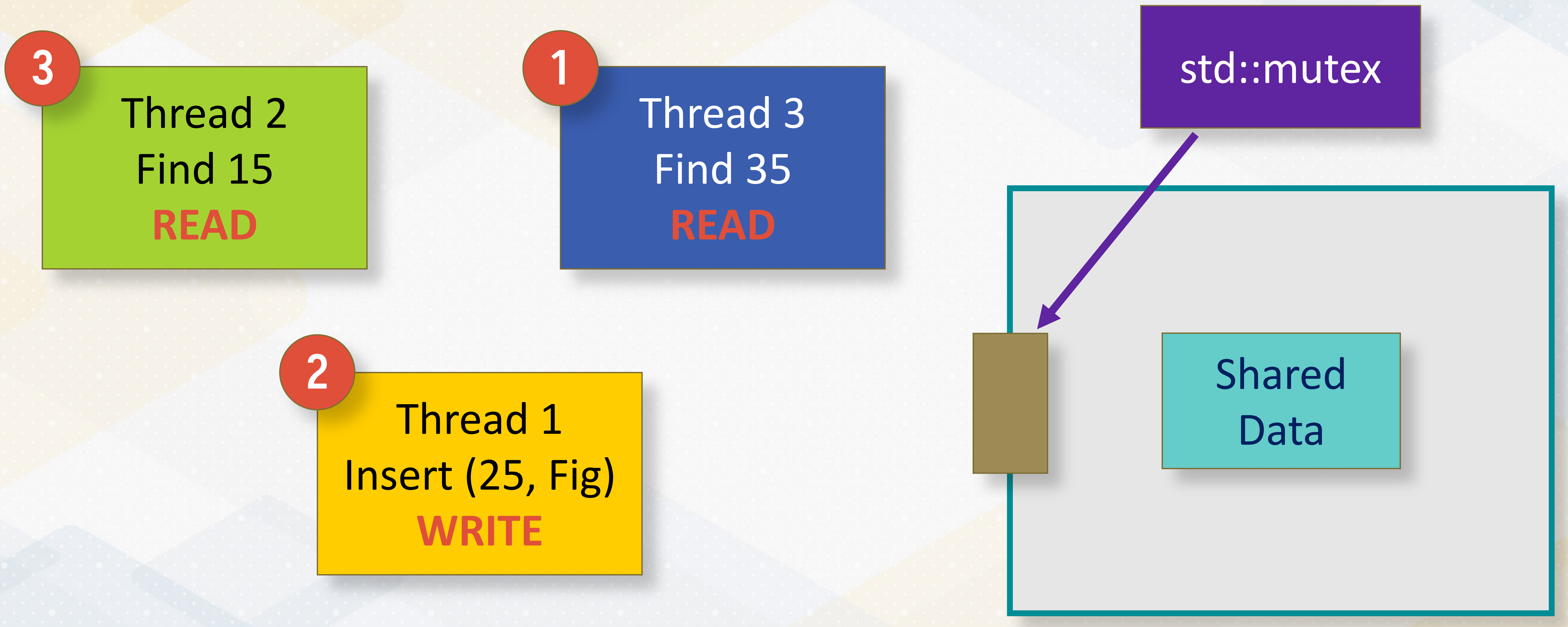
Increased Lock Acquisition/ Release Cost

Increased Lock Memory Consumption

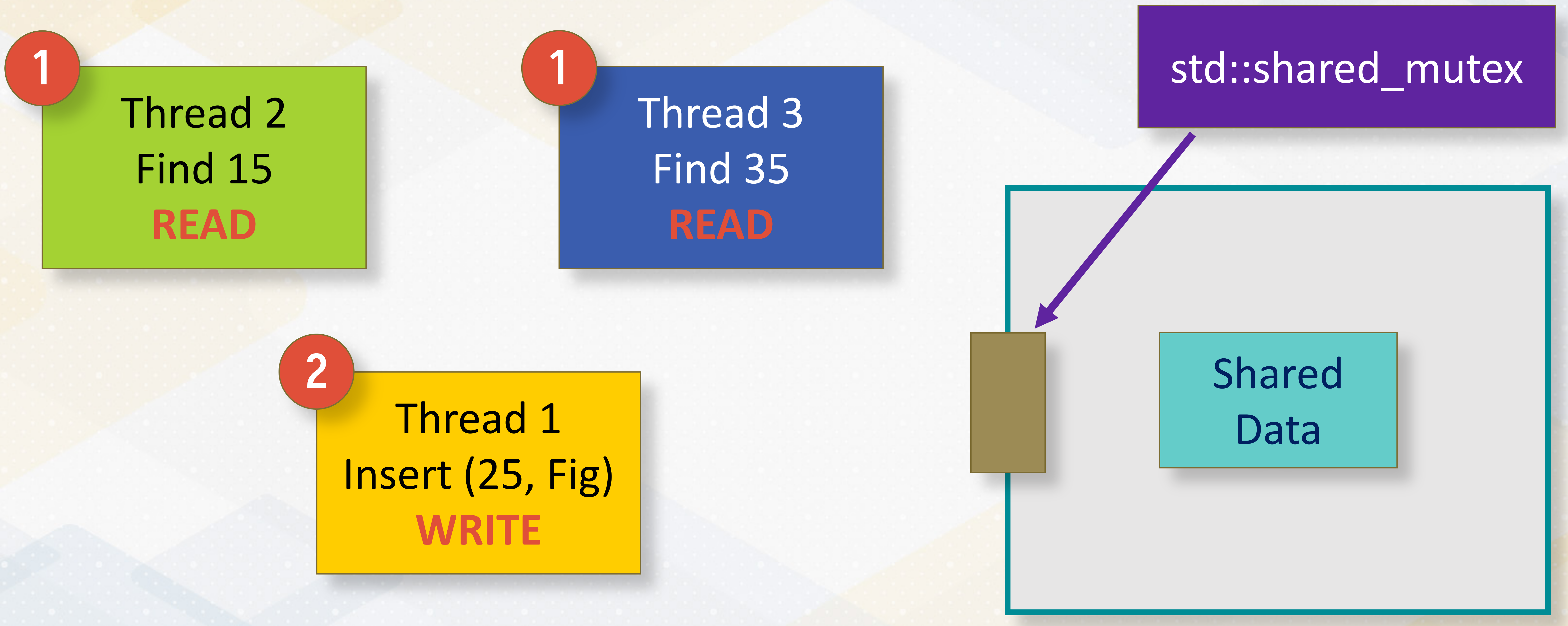


Shared Mutex

Limitations of std::mutex



Limitations of std::mutex



std::unique_lock<std::shared_mutex

std::shared_mutex allows multiple threads to hold a read (shared) lock simultaneously while ensuring exclusive access for write operations.

```
mutable std::shared_mutex mutexes[capacity];

std::vector<std::unique_ptr<std::mutex>> mutexes;
void insertOrUpdate(int key, int value) {
size_t index = hashFunction(key); // Determine the slot index for the key
do {
// Exclusive lock for writing
std::unique_lock<std::shared_mutex> lock(mutexes[index]);
// Attempt to insert or update the slot
if (conditions_met) { // Insert or update logic }
// Handle collision and calculate next slot index
} while (not_inserted);
}
```

std::unique_lock<std::shared_mutex

std::shared_lock<std::shared_mutex> allows multiple threads to read from the same slot concurrently, provided no thread is writing to it.

```
int getValue(int key) const {
size_t index = hashFunction(key);
size_t originalIndex = index;

do {
// Shared lock for reading
std::shared_lock<std::shared_mutex> lock(mutexes[index]);
// Check if the key is inside the slot or not
// Calculate next slot index
} while (index != originalIndex);
}
```

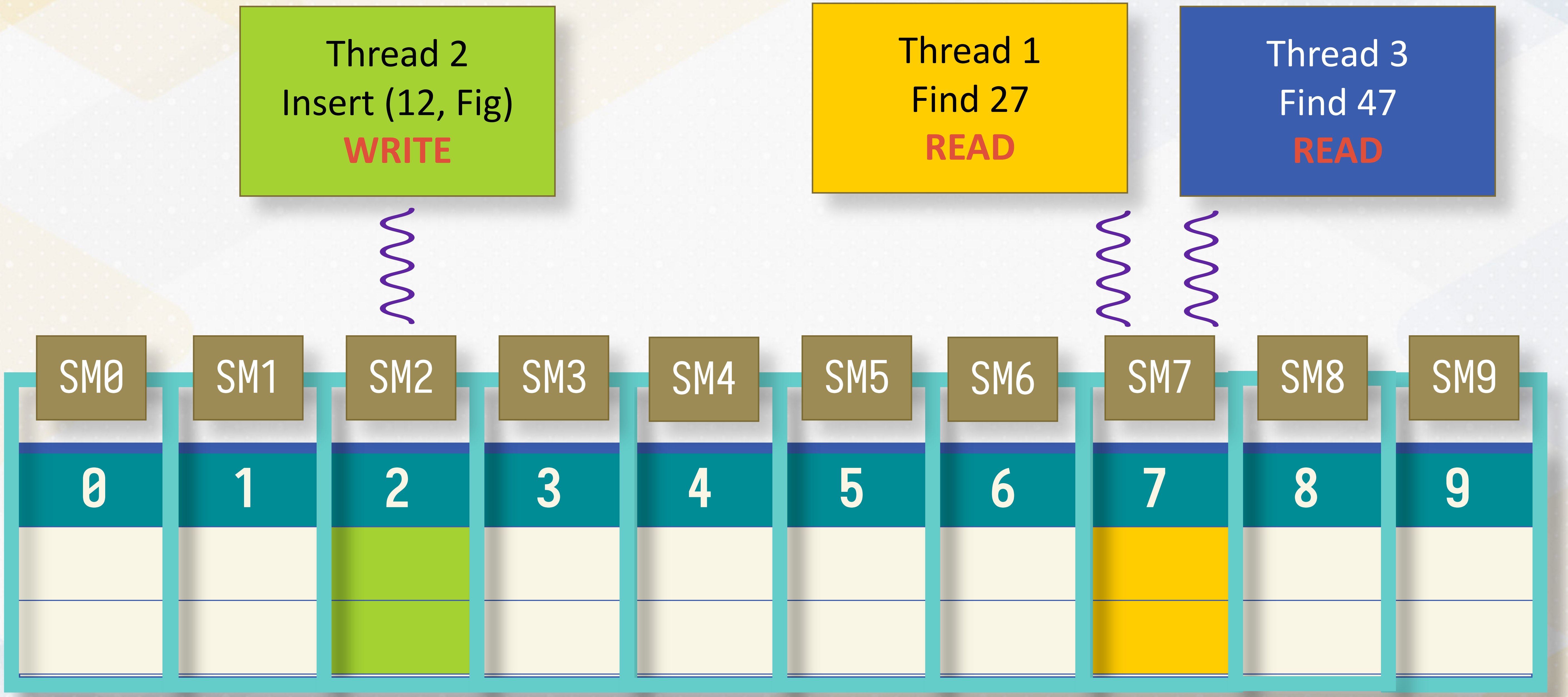
Exclusive Write Lock

Shared Resource	One Hash Table Slot
Mutex Type	std::unique_lock <std::shared_mutex>
Currently Accessing Threads	Thread 2 (**WRITE**)
Waiting Threads	Thread 1 (**READ**) Thread 3 (**READ**) …

Shared Read Lock

Shared Resource	One Hash Table Slot
Mutex Type	std::shared_lock <std::shared_mutex>
Currently Accessing Threads	Thread 1 (**READ**) Thread 3 (**READ**)
Waiting Threads	Thread 2 (**WRITE**) ...

Fine-Grained Locking with Shared Mutex



Simulation

Need for Simulation

REAL THREADS & MUTEXES	SIMULATED THREADS & MUTEXES
Too fast for qualitative analysis	Controlled environment
Non-deterministic	Deterministic
Real deal	Only an approximation

Simulated Mutex

Allows only one thread to access the shared resource at a time.

```
def try_acquire(self):
if not self.is_locked:
self.is_locked = True
return 'acquired'
return 'waiting’
```

Simulated Shared Mutex

Allows multiple readers or a single writer.

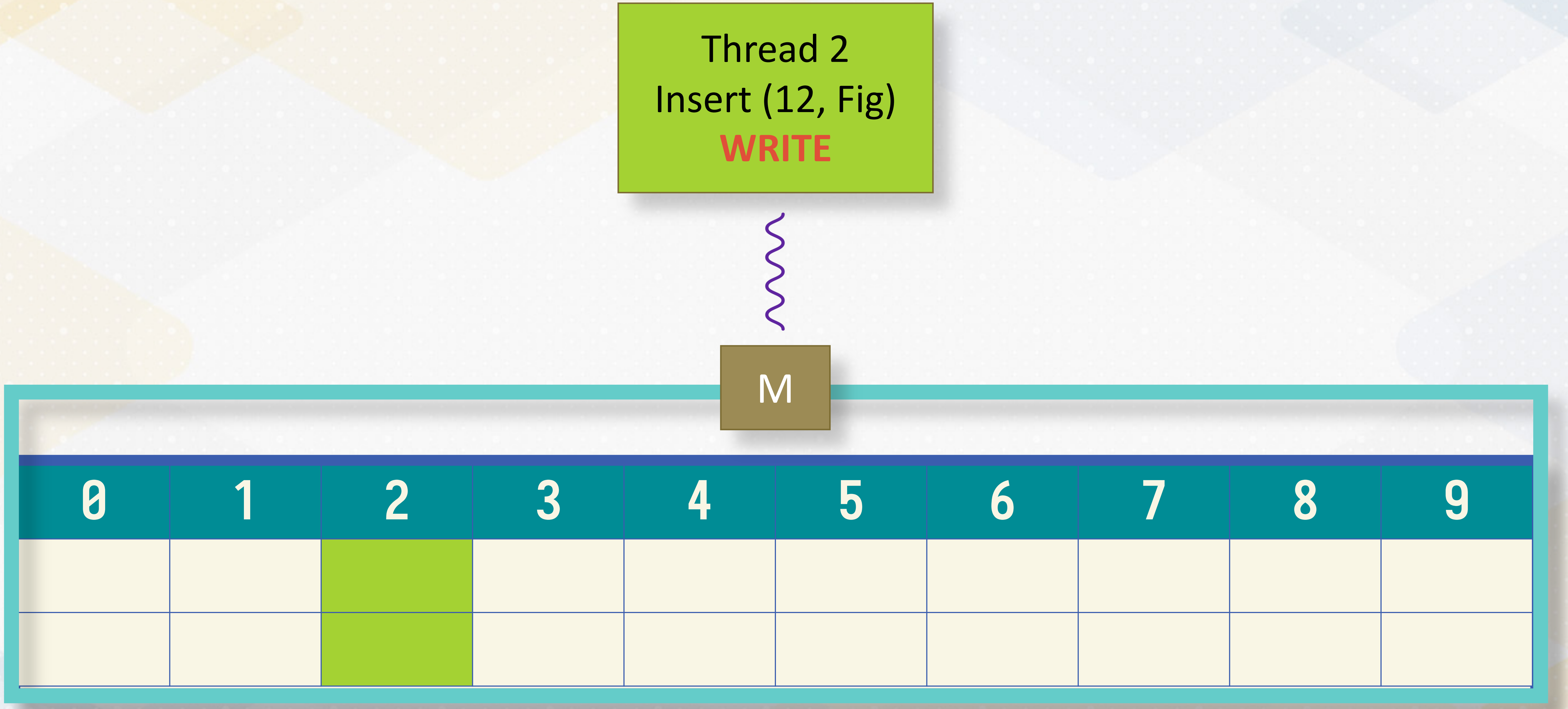
try_acquire_read	Grants read access unless a write lock is held
try_acquire_write	Grants write access if no reads or writes are active
release_read	Releases read lock
release_write	Releases write lock

Concurrent Hash Table

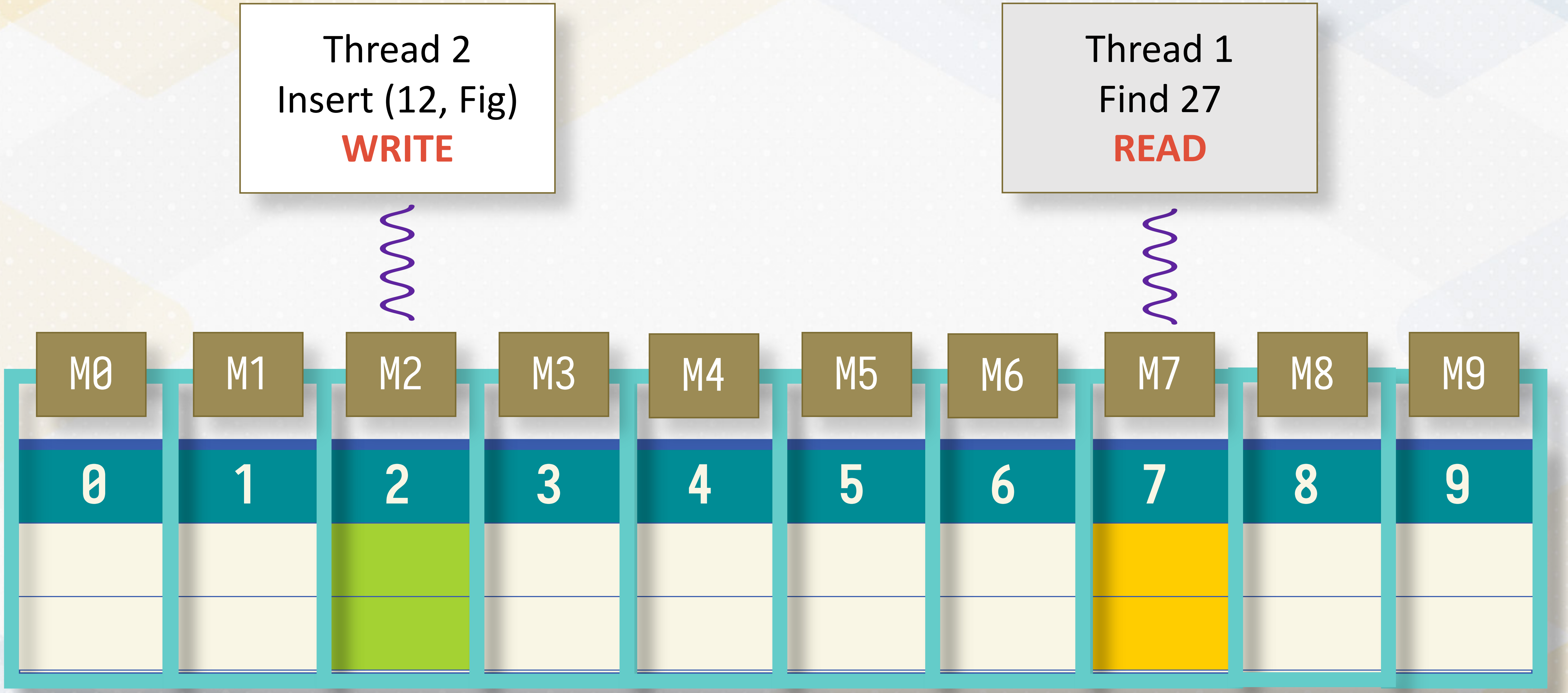
Models a hash table with three concurrency control protocols:

GLOBAL MUTEX	One mutex for entire table
MUTEX_PER_SLOT	One mutex for each slot
SHARED_MUTEX_PER_SLOT	One shared mutex for each slot

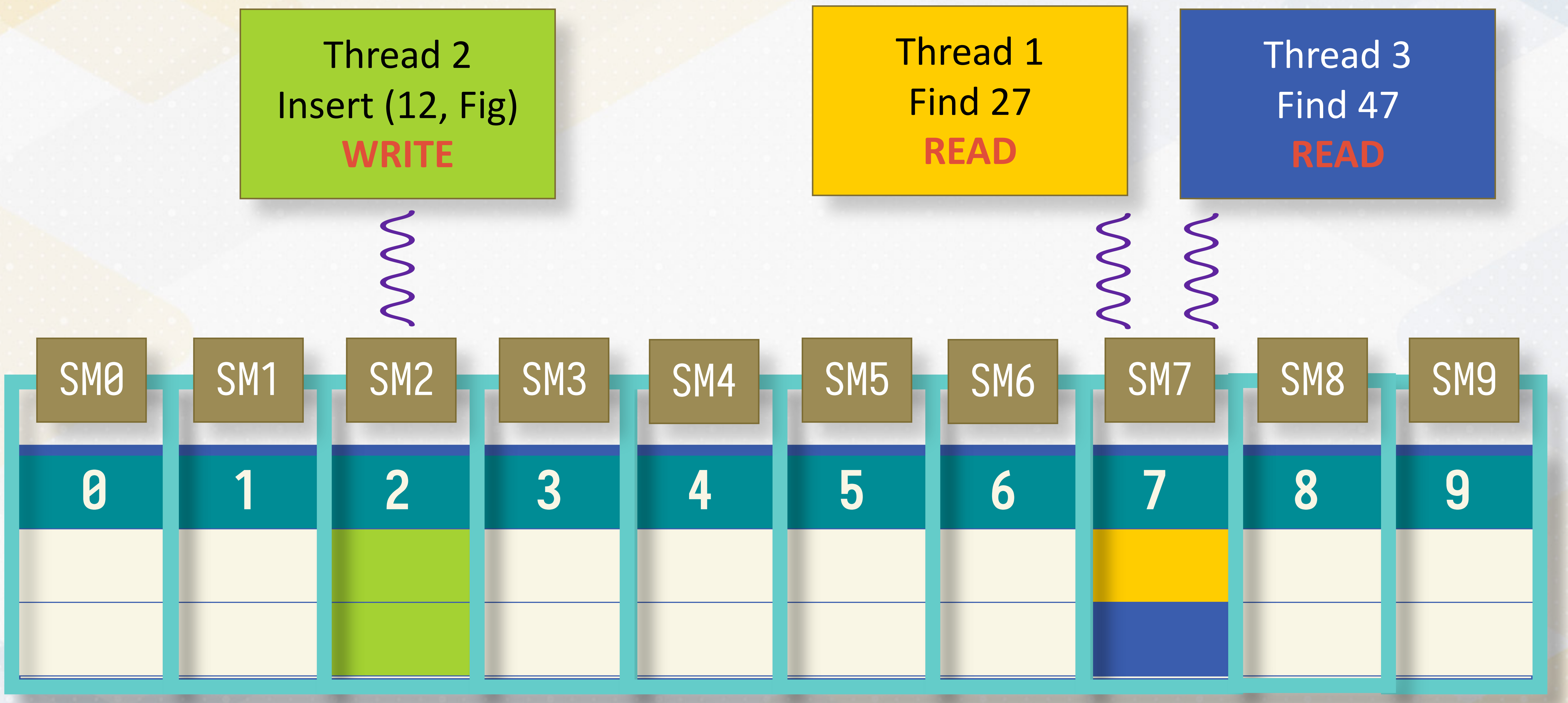
Protocol #1: global_mutex



Protocol #2: mutex_per_slot



Protocol #3: shared_mutex_per_slot



SimulatedThread

Simulates a thread performing a series of operations on the hash table. Step function manages the thread's operations based on its current state.

State	Description
Lock	Acquire lock
Find	Read to current slot
Insert	Write to current slot
Unlock	Release lock
Unlock_and_relock	Release lock and move to next slot

Simulator

Each thread can take only one step in a logical time step, and a thread can acquire a lock only if it was released in prior time step.

```
Thread-0: Attempt to lock for insert on slot 2 -
waiting
Thread-1: Key 25 found at slot 5
Thread-2: Released lock on slot 3 – released
```

Operation Trace

Each thread performs a sequence of operations:

```
Thread-0: insert (25, v25), find 25, find 15
Thread-1: find 35, insert (35, v35), and find
25
Thread-2: find 45, find 25
```

Logical Time Step Count

Comparing the total logical time steps taken by different concurrency control protocols reveals insights into their efficiency for various operation traces.

GLOBAL MUTEX	30 steps
MUTEX_PER_SLOT	21 steps
SHARED_MUTEX_PER_SLOT	15 steps

Global Mutex

```
Time Step: 15
Thread-1: Released lock on table - released
Thread-2: Attempt to lock table - waiting
```

```
Time Step: 16
Thread-1: Attempt to lock table - acquired
Thread-2: Attempt to lock table - waiting
```

Mutex Per Slot

```
Time Step: 7
Thread-0: Attempt to lock slot 6 - acquired
Thread-1: Attempt to lock slot 5 - acquired
Thread-2: Attempt to lock slot 5 - waiting
```

```
Time Step: 8
Thread-0: Key 15 not found.
Thread-1: Found another key 25 at slot 5
Thread-2: Attempt to lock slot 5 - waiting
```

Shared Mutex Per Slot

```
Time Step: 7
Thread-0: Attempt to lock for find on slot 6 - acquired_read
Thread-1: Attempt to lock for insert on slot 5 - acquired_write
Thread-2: Attempt to lock for find on slot 6 - acuired_read
```

```
Time Step: 8
Thread-0: Key 15 not found.
Thread-1: Found another key 25 at slot 5
Thread-2: Key 45 not found.
```

Conclusion

- Parallel Index Construction
- Fine-Grained Locking
- Shared Mutex
- Simulation Framework