



Lecture 8: 2Q Policy and Multi-Threading



Logistics

- Point Solutions App
 - Session ID: **database**
- Programming assignment 2 released (Gradescope)
- Exercise sheet 1 will be released one week before Ed deadline (Canvas Quiz)
- Mid-term exam on Oct 2 (in-class)

Recap

- Storage Manager
- Buffer Manager
- Cache Replacement Policy
- Buffer Pool Flooding

Lecture Overview

- 2Q Policy
- Multi-Threading
- Fine-Grained Locking

2Q Policy



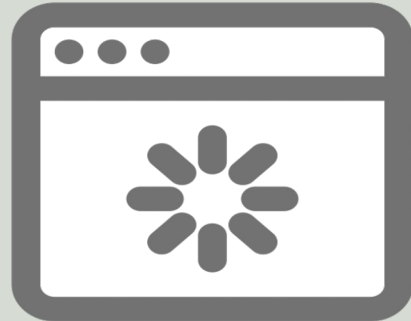
Sequential Scan

sequential scan

when a database reads every table page while processing a query

```
// Sequential scan across the sales_data table  
// Assumption: No index on sale_date column  
SELECT *  
FROM sales_data  
WHERE sale_date BETWEEN '2040-01-01' AND '2040-01-31';
```


Buffer Pool Flooding



Sequential scan floods the buffer pool with less important pages

FIFO Policy

Evicts the oldest page from cache

LRU Policy

Evicts hot pages to store cold pages



FIFO: Buffer Pool Flooding

Page Access
Trace

1 2 1 2 3 4 1 2 5 6 1 2

Buffer Pool Capacity: 3 Pages

FIFO evicts **hot** pages 1 and 2

Access Page	1	2	1	2	3	4	1	2	5	6	1	2
FIFO Queue: 1	1	2	2	2	3	4	1	2	5	6	1	2
FIFO Queue: 2	–	1	1	1	2	3	4	1	2	5	6	1
FIFO Queue: 3	–	–	–	–	1	2	3	4	1	2	5	6
Evict Page						1	2	3	4	1	2	5
Cache Misses	1	2	2	2	3	4	5	6	7	8	9	10

LRU: Buffer Pool Flooding

Page Access Trace

1 2 1 2 3 4 1 2 5 6 1 2

Buffer Pool Capacity: 3 Pages

LRU evicts **hot** pages 1 and 2

Access Page	1	2	1	2	3	4	1	2	5	6	1	2
LRU Queue: 1	1	2	1	2	3	4	1	2	5	6	1	2
LRU Queue: 2	-	1	2	1	2	3	4	1	2	5	6	1
LRU Queue: 3	-	-	-	-	1	2	3	4	1	2	5	6
Evict Page						1	2	3	4	1	2	5
Cache Misses	1	2	2	2	3	4	5	6	7	8	9	10

TwoQ Policy

page transition logic

- Page initially enters FIFO queue
- Page moves to LRU queue on second access

FIFO Queue

Read-Once
Pages
Accessed
During
Sequential
Scan

LRU Queue

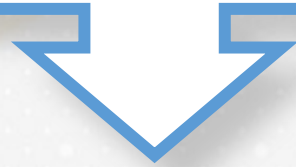
Frequently-
Accessed
Hot Pages
Promoted
to
LRU Queue

TwoQ Policy: touch

Pages Enter **FIFO** Queue



Pages Promoted To **LRU**



Page Position Updated
When Accessed

```
void touch(PageID page_id) {  
    if (pageMap.find(page_id) != pageMap.end()) {  
        auto it = pageMap[page_id];  
        // If in FIFO and accessed again, move to LRU  
        // Otherwise, adjust position within LRU  
        // If cache is full, then evict  
        // New pages added to FIFO  
    }  
}
```


TwoQ Policy: touch

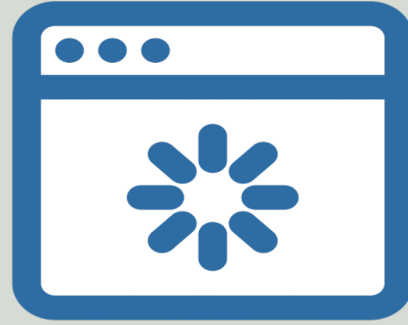
Check **FIFO** Queue When
Page Accessed

Page in FIFO Queue
Upgraded to LRU Queue

Page in LRU Queue Has
Higher Reuse Probability

```
void touch(PageID page_id) {  
    // If the page is already in the cache  
    if (pageMap.find(page_id) != pageMap.end()) {  
        auto it = pageMap[page_id];  
        // If it's in FIFO, move to LRU  
        if (std::find(FIFO.begin(), FIFO.end(), page_id)  
            != FIFO.end()) {  
            FIFO.erase(it);  
            LRU.push_front(page_id);  
            pageMap[page_id] = LRU.begin();  
        }  
        ...  
    }  
}
```

TwoQ Policy: touch



Most Recently Used Pages in **LRU**
Queue Evicted Last

```
void touch(PageID page_id){  
    else {  
        // If it's in LRU, move to the front  
        LRU.erase(it);  
        LRU.push_front(page_id);  
        pageMap[page_id] = LRU.begin();  
    }  
    ...  
}
```


TwoQ Policy: touch

```
void touch(PageID page_id) {  
    ...  
    // If the page is not in the cache  
    else {  
        // If cache is full, evict  
        if (FIFO.size() + LRU.size() >= cacheSize) {  
            evict();  
        }  
        // Add page to FIFO  
        FIFO.push_back(page_id);  
        pageMap[page_id] = std::prev(FIFO.end());  
    }  
}
```

TwoQ Policy: evict

TwoQ evict

- Prioritizes Eviction from FIFO Queue
- Minimizes Cache Pollution from Sequential Scans

```
PageID evict() {  
    // FIFO pages are evicted first to minimize cache pollution,  
    // followed by the least recently used pages in LRU  
}
```


TwoQ Policy: evict

```
PageID evict() {  
    PageID evictedPageId = INVALID_VALUE;  
    if (!FIFO.empty()) {  
        evictedPageId = FIFO.front();  
        FIFO.pop_front();  
        pageMap.erase(evictedPageId);  
    } else if (!LRU.empty()) {  
        evictedPageId = LRU.back();  
        LRU.pop_back();  
        pageMap.erase(evictedPageId);  
    }  
    return evictedPageId;  
}
```

TwoQ: No Buffer Pool Flooding

TwoQ retains **hot** pages like 1 and 2

Access Page	1	2	1	2	3	4	1	2	5	6	1	2
FIFO Queue:1	1	2	2	-	3	4	4	4	5	6	6	6
FIFO Queue:2	-	1	-	-	-	-	-	-	-	-	-	-
FIFO Queue:3	-	-	-	-	-	-	-	-	-	-	-	-
LRU Queue: 1	-	-	1	2	2	2	1	2	2	2	1	2
LRU Queue: 2	-	-	-	1	1	1	2	1	1	1	2	1
LRU Queue: 3	-	-	-	-	-	-	-	-	-	-	-	-
Evict Page						3			4	5		
Cache Misses	1	2	2	2	3	4	4	4	5	6	6	6

More Policies: LFU

LFU Policy Benefits

Keeps Frequently-Used
Pages in Memory

LFU Policy Limitations

Does Not Consider Changes
to Recent Access Patterns

More Policies: ARC

ARC Policy Benefits

Dynamically Balances between
LRU and LFU

ARC Policy Limitations

Complex
Implementation

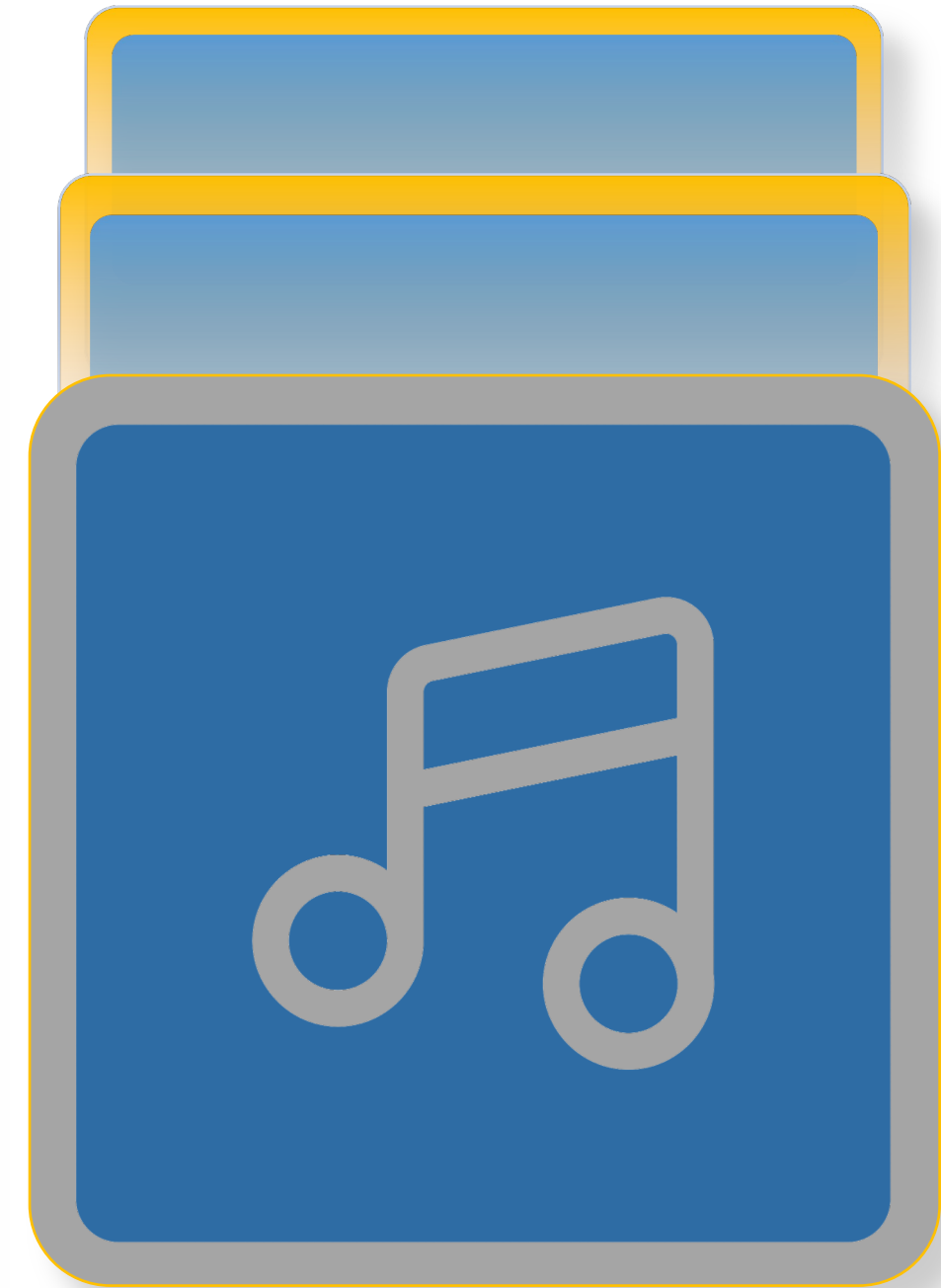
Music Playlist + Cache Eviction Policies

LRU

Keeps Only New Discoveries

TwoQ

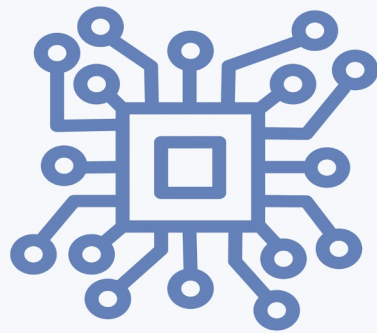
Keeps Favorites and New Discoveries



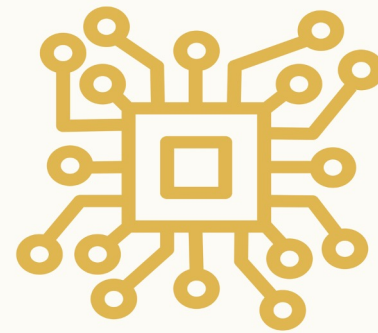
Multi-Threading



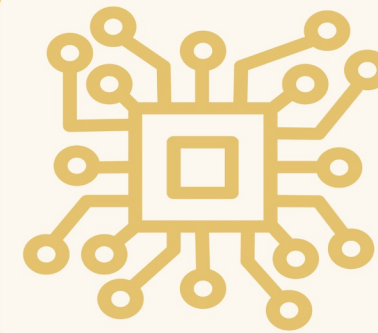
History of CPUs



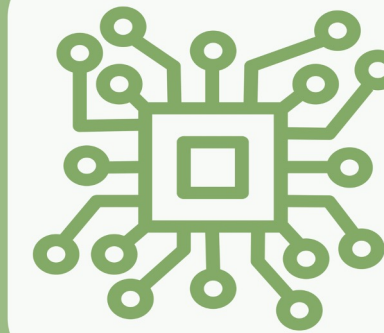
1970-1980s
First CPUs
Few
Megahertz



1990s
100s of
MHz to
over 1 GHz

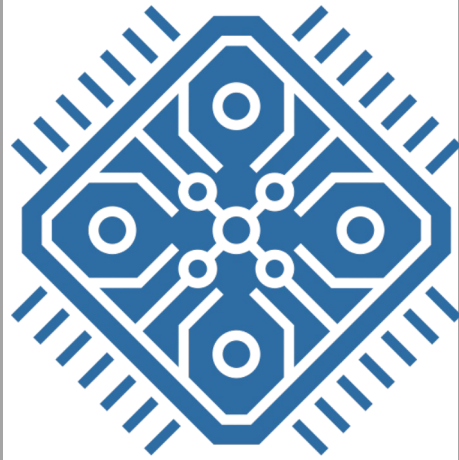


Early 2000s
Top-end
CPUs
over 3 GHz

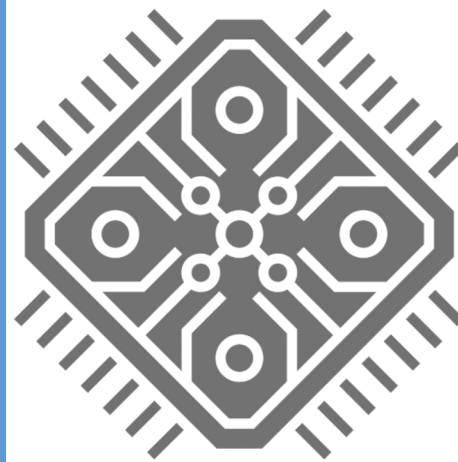


Mid 2000s
3.8 GHz
Thermal
Wall

History of CPUs



2005
First 2-core
CPUs
Parallel
Processing



Today
64-core
CPUs
AMD
Threadripper

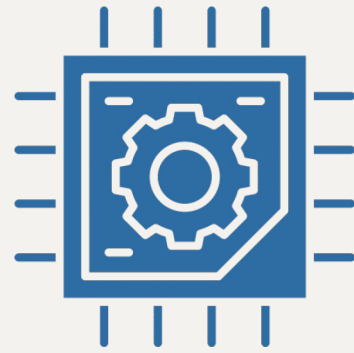
Multi-Core CPUs



Clock Speed

Rhythm of a
Drumbeat

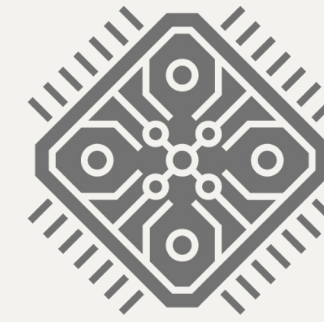
One Beat =
One Instruction



Single-Core CPU

Faster Drumbeat

Causes Overheating



Multi-Core CPU

Multiple
Drummers

Multiple Instructions
at the same time

Threading

Thread: Series of Instructions

**Multi-Core CPUs:
Multiple Threads in Parallel**



Thread 1
Txn 1 from User 1



Thread 2
Txn 2 from User 2

Multi-Threading Example

Context

A shared bank balance variable updated by multiple threads representing different transactions

```
#include <iostream>
#include <thread>
#include <vector>

int bankBalance = 1000; // Initial bank balance

void performTransactions() {
    for (int i = 0; i < 10000; ++i) {
        bankBalance += 10; // Deposit
        bankBalance -= 10; // Withdrawal
    }
}
```

Multi-Threading Example

```
int main() {  
    std::vector<std::thread> threads;  
    for (int i = 0; i < 8; ++i) {  
        // Initiate transactions without synchronization  
        threads.emplace_back(performTransactions);  
    }  
    for (auto &thread : threads) {  
        thread.join(); // Wait for all threads to finish  
    }  
    std::cout << "Expected balance: 1000\n  
                Actual balance: " << bankBalance  
                << std::endl;  
    return 0;  
}
```


Need for Synchronization

Challenge

Without synchronization, simultaneous transactions lead to inaccurate balance

Run 1

Expected Balance: \$1000

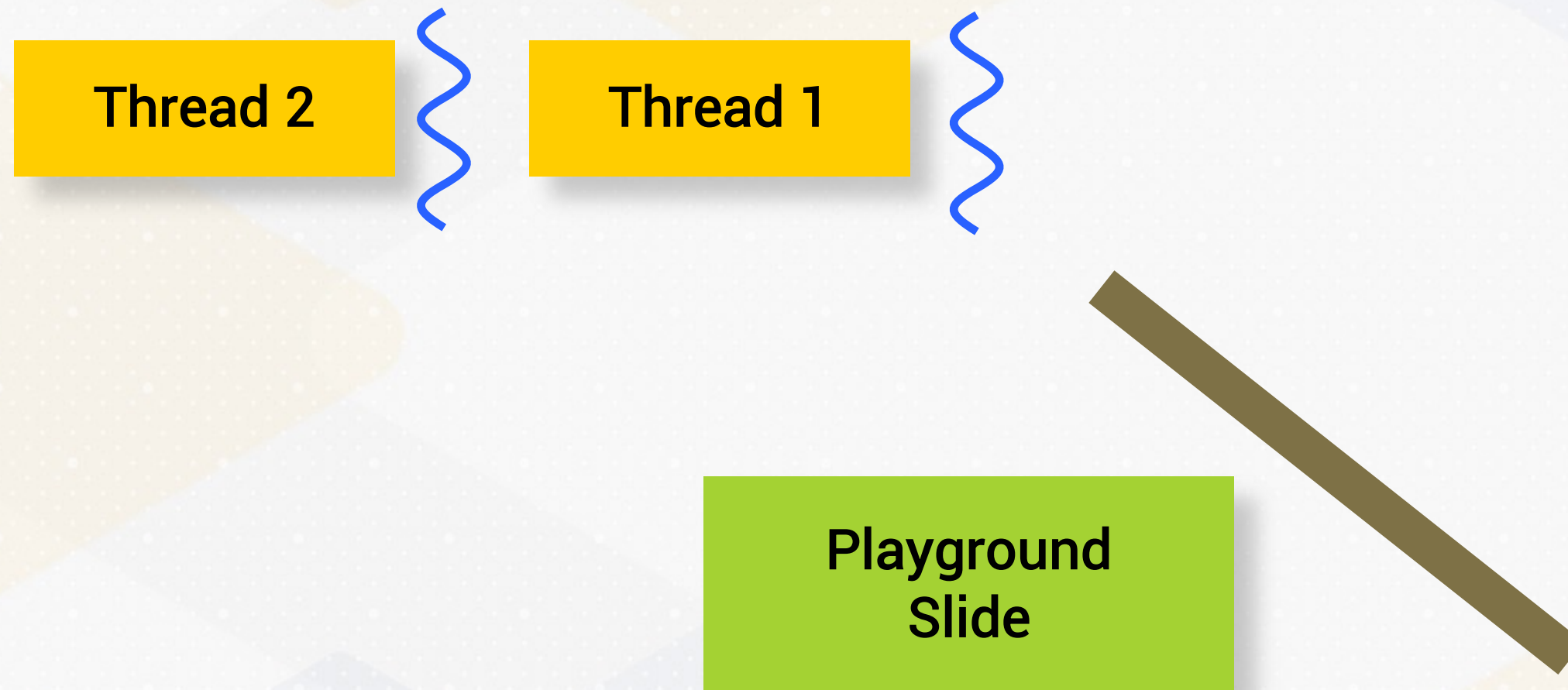
Actual Balance : ~~\$910~~

Run 2

Expected Balance: \$1000

Actual Balance : ~~\$1020~~

Playground Slide Synchronization



Assembly Level Explanation

bankBalance += 10;

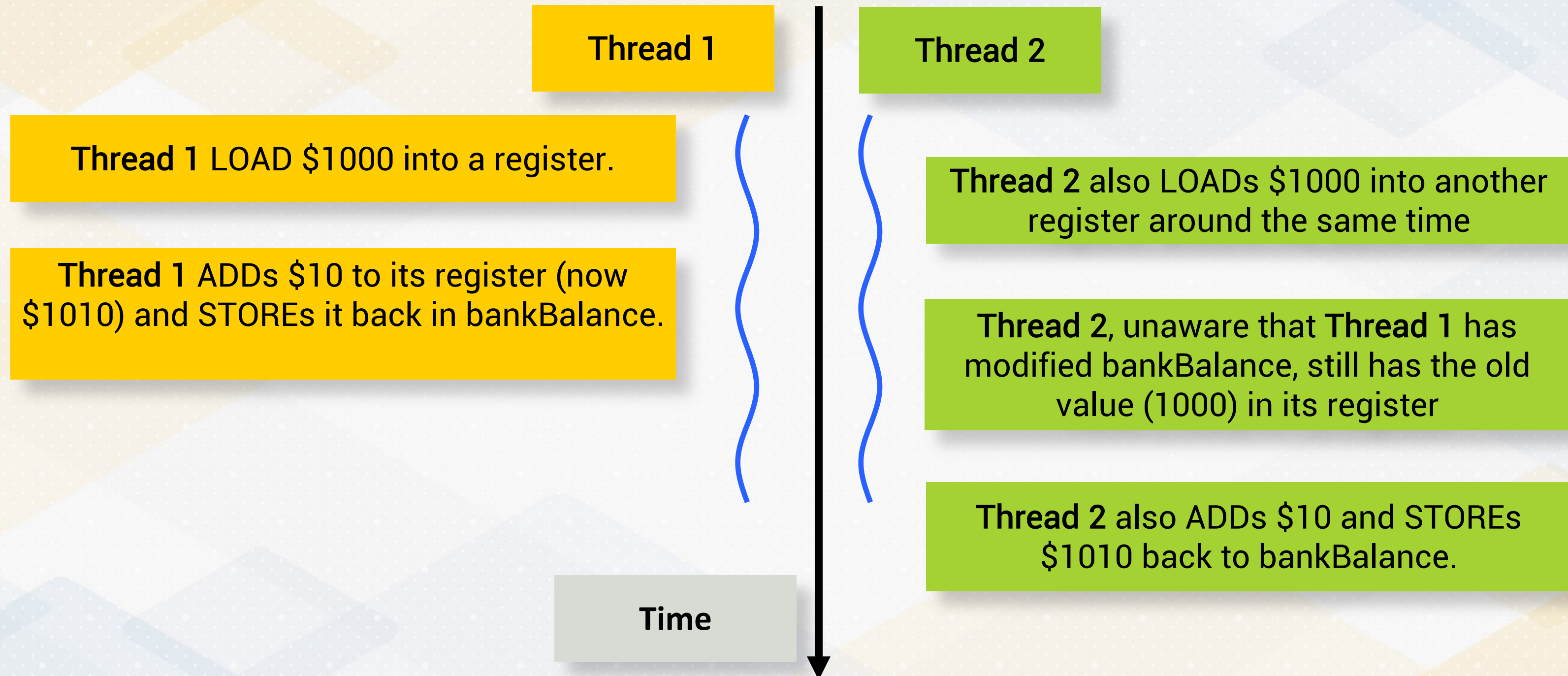


```
// LOAD value of bankBalance from memory to register
MOV EAX, [bankBalance]

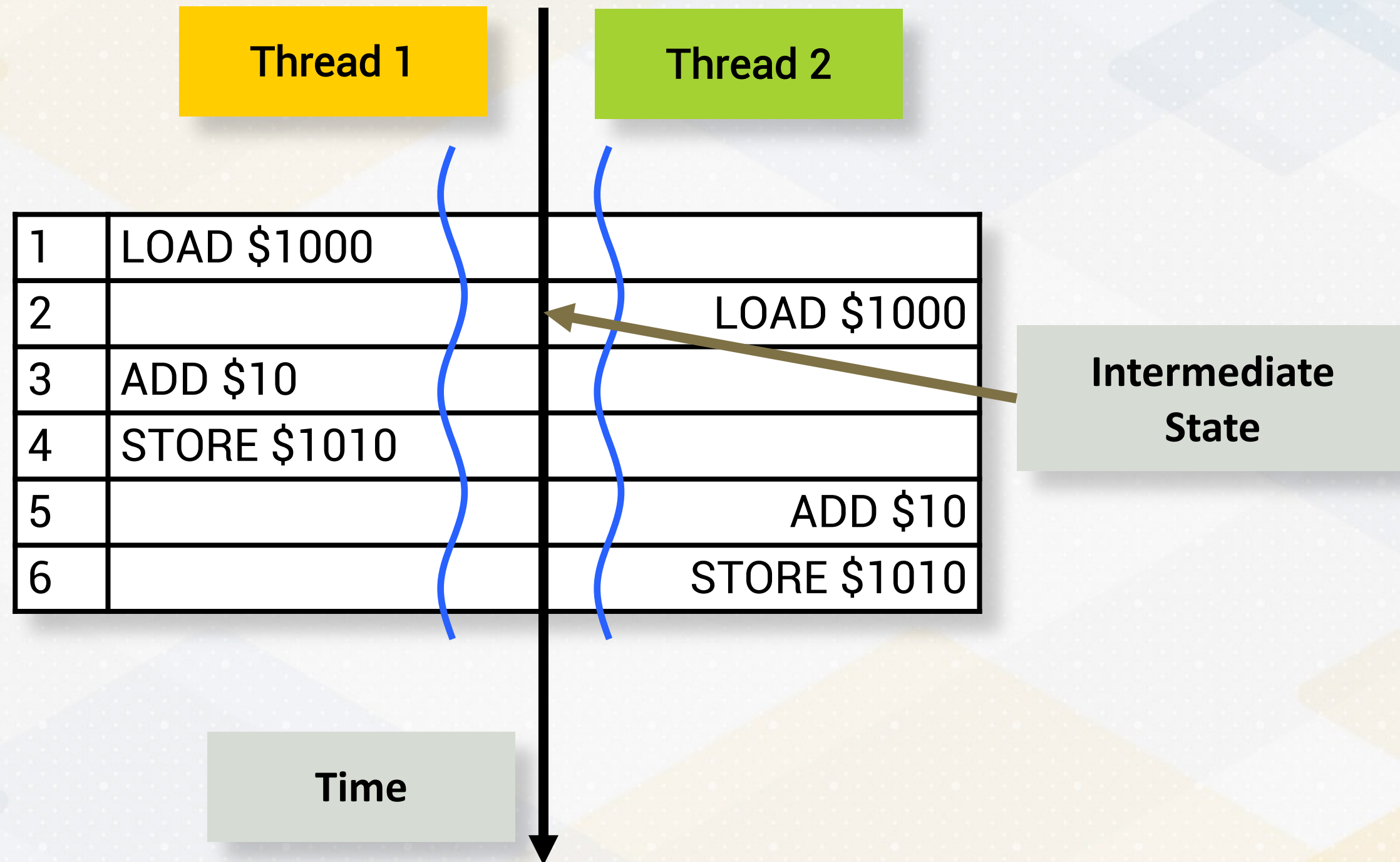
//ADD 10 to increment the value in the register
ADD EAX, 10

//STORE the new value in register back to the memory location of bankBalance
MOV [bankBalance], EAX
```

Race Condition



Non-Atomic Load-ADD-STORE Sequence



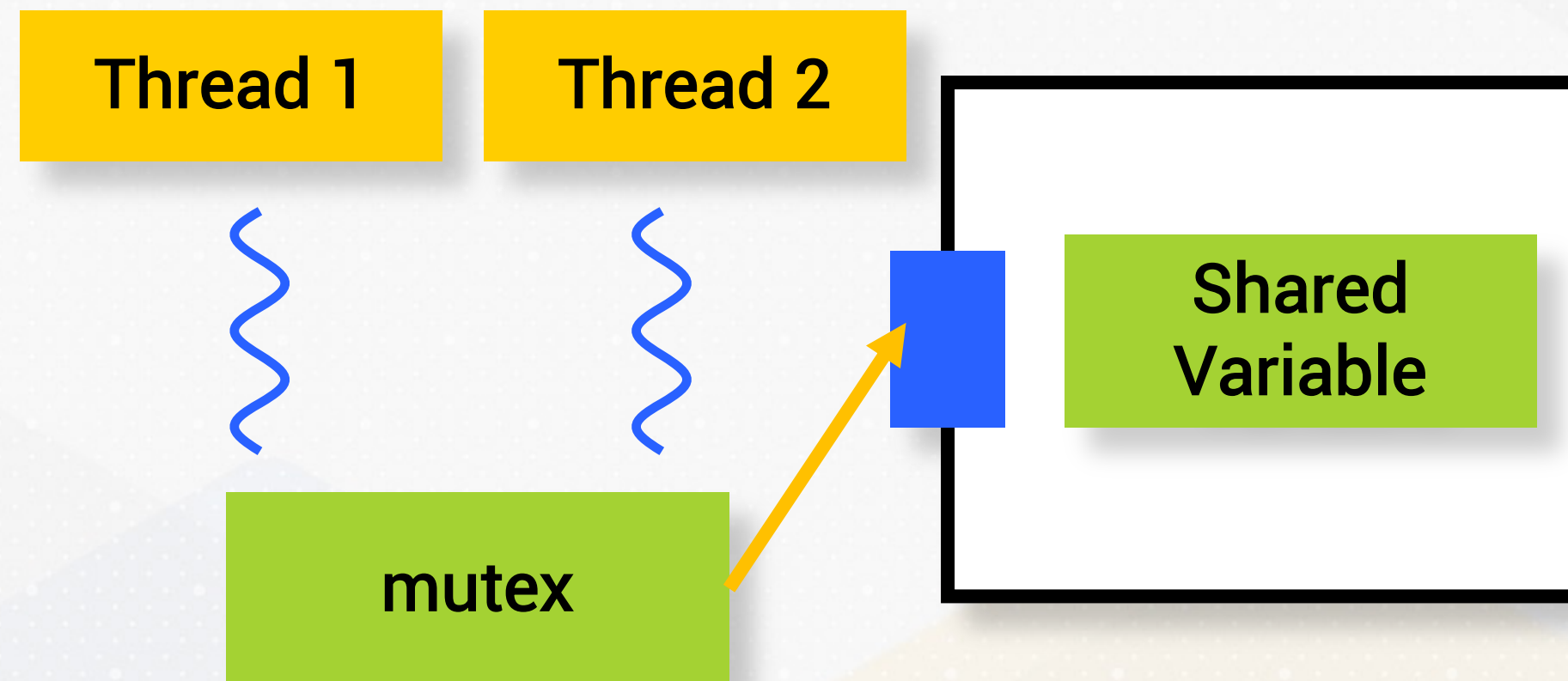
Mutex



std::mutex (mutual exclusion)

mutex

a door lock to a room with the shared variable



std::mutex (mutual exclusion)



Mutex manual
locking &
unlocking

Transactions
processed
atomically

Prevents data
corruption

```
std::mutex bankMutex;  
  
void performTransactions(int account) {  
    for (int i = 0; i < 10000; ++i) {  
        bankMutex.lock();    // Manually lock the mutex  
        bankBalance += 10;    // Deposit  
        bankBalance -= 10;    // Withdrawal  
        bankMutex.unlock();  // Manually unlock the mutex  
    }  
}
```


Critical Section



- Operations between lock and unlock form critical section
- Only one thread enters the section at a time

```
bankMutex.lock(); // Manually lock the mutex
// CRITICAL SECTION STARTS
bankBalance += 10; // Deposit
bankBalance -= 10; // Withdrawal
// CRITICAL SECTION ENDS
bankMutex.unlock(); // Manually unlock the mutex
```

Spinlock Operations at Assembly Level

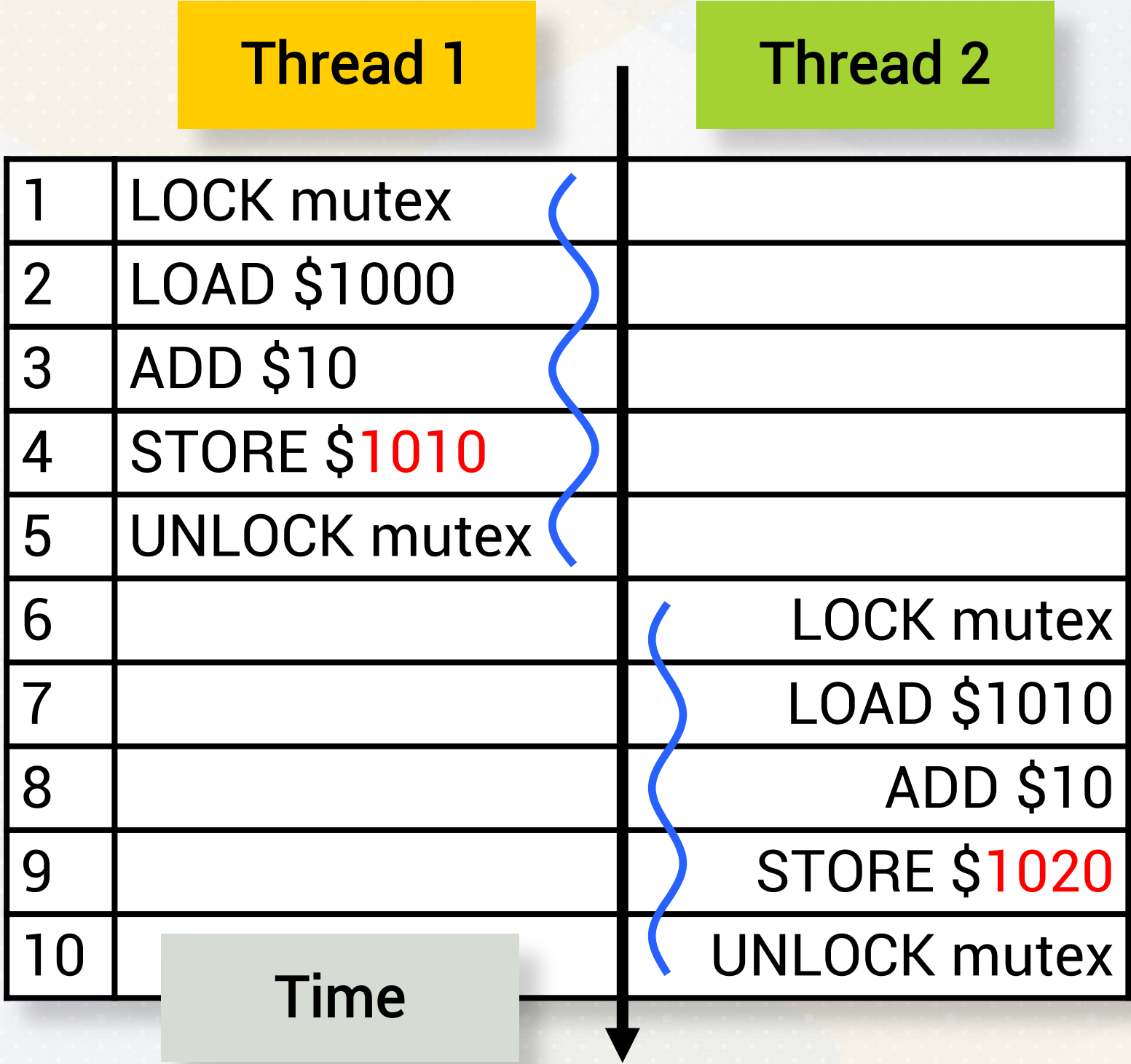
```
; Locking the spinlock
retry:
    ; EAX is set to the expected old value (unlocked = 0)
    MOV EAX, 0
    ; EBX is set to the new value to store if comparison is successful
    (locked = 1)
    MOV EBX, 1
    ; Atomically compare [mutex] to 0, if equal replace [mutex] with 1
    LOCK CMPXCHG [mutex], EBX;
    ; Test if the original value of [mutex] (now in EBX) was 1
    TEST EBX, EBX;
    ; If the mutex was already locked (EBX was 1), jump to retry
    JNZ retry
```


Spinlock Operations at Assembly Level

```
; Critical section to update bankBalance
MOV EAX, [bankBalance] ; Load bank balance
ADD EAX, 10             ; Modify bank balance
MOV [bankBalance], EAX ; Store bank balance

; Unlocking the Mutex
MOV EBX, 0              ; Set EBX to 0, which represents the unlocked state
MOV [mutex], EBX        ; Store 0 into the mutex, effectively unlocking it
```

Mutex Prevents Data Race

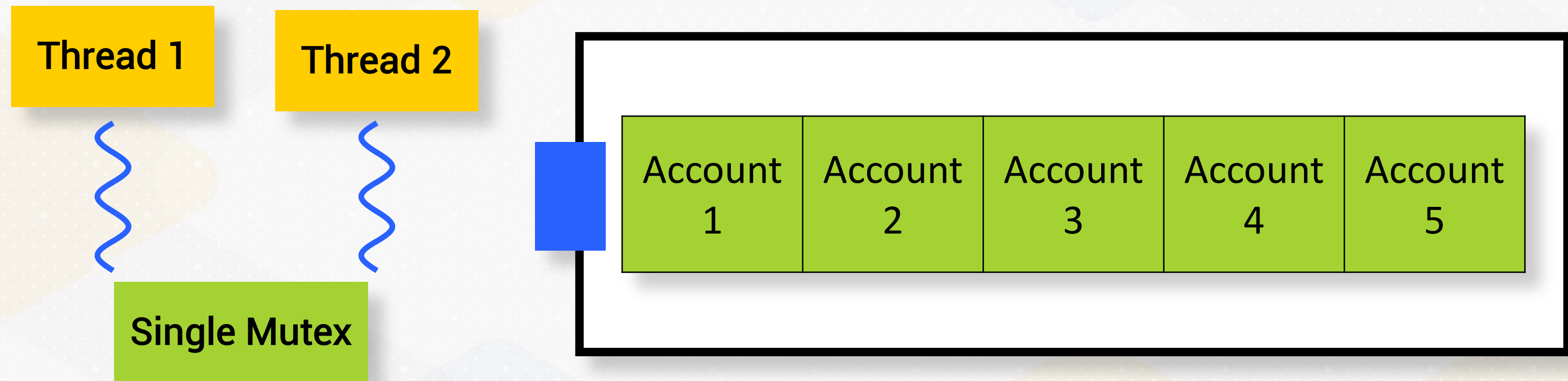


Multiple Bank Accounts

Managing transactions across 5 bank accounts in a multi-threaded application

```
std::mutex bankMutex;  
// Initial balances of five bank accounts  
int bankAccounts[5] = {1000, 2000, 3000, 4000, 5000};  
  
void performTransactions(int account) {  
    for (int i = 0; i < 10000; ++i) {  
        bankMutex.lock();           // Single mutex for all five accounts  
        bankAccounts[account] += 10; // Deposit  
        bankAccounts[account] -= 10; // Withdrawal  
        bankMutex.unlock();  
    }  
}
```

Multiple Bank Accounts

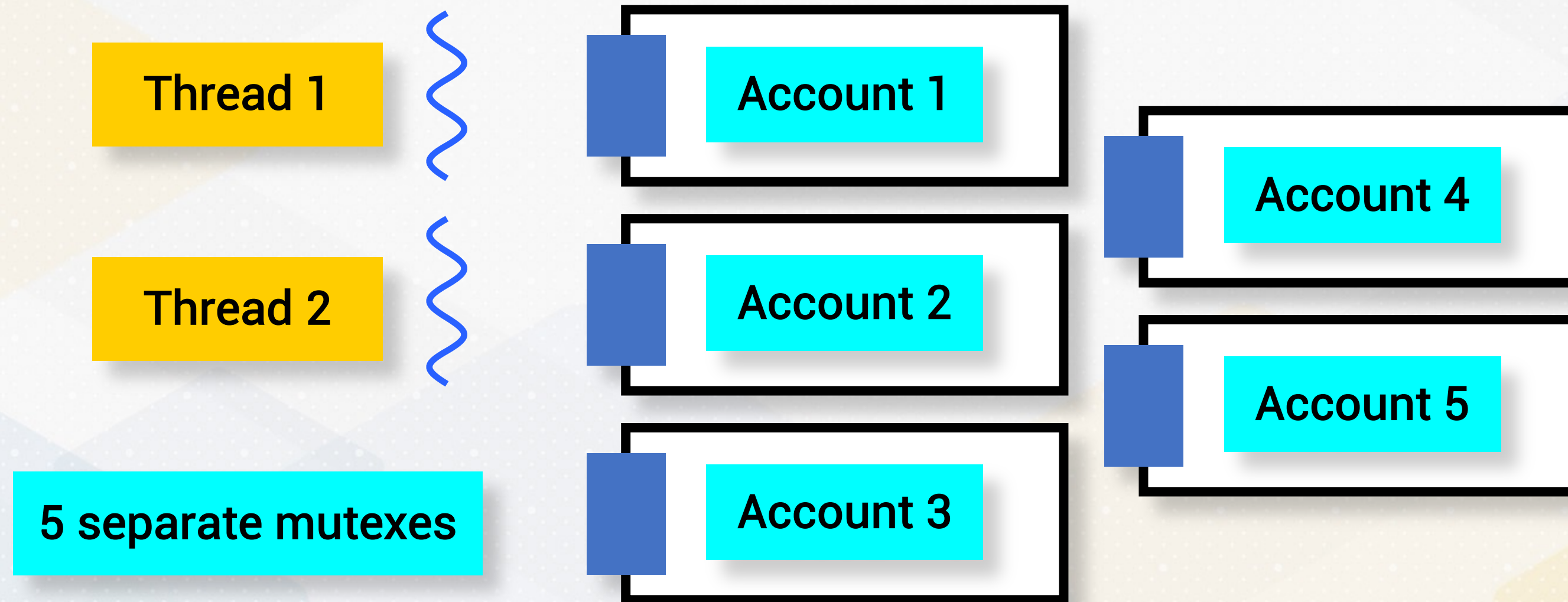


Fine-Grained Locking



Fine-Grained Locking

Use a separate mutex for each bank account to improve concurrency



Fine-Grained Locking

Non-conflicting transactions can run in parallel

```
std::mutex bankAccountMutexes[5]; // A mutex for each bank account
int bankAccounts[5] = {1000, 2000, 3000, 4000, 5000}; // Initial balances

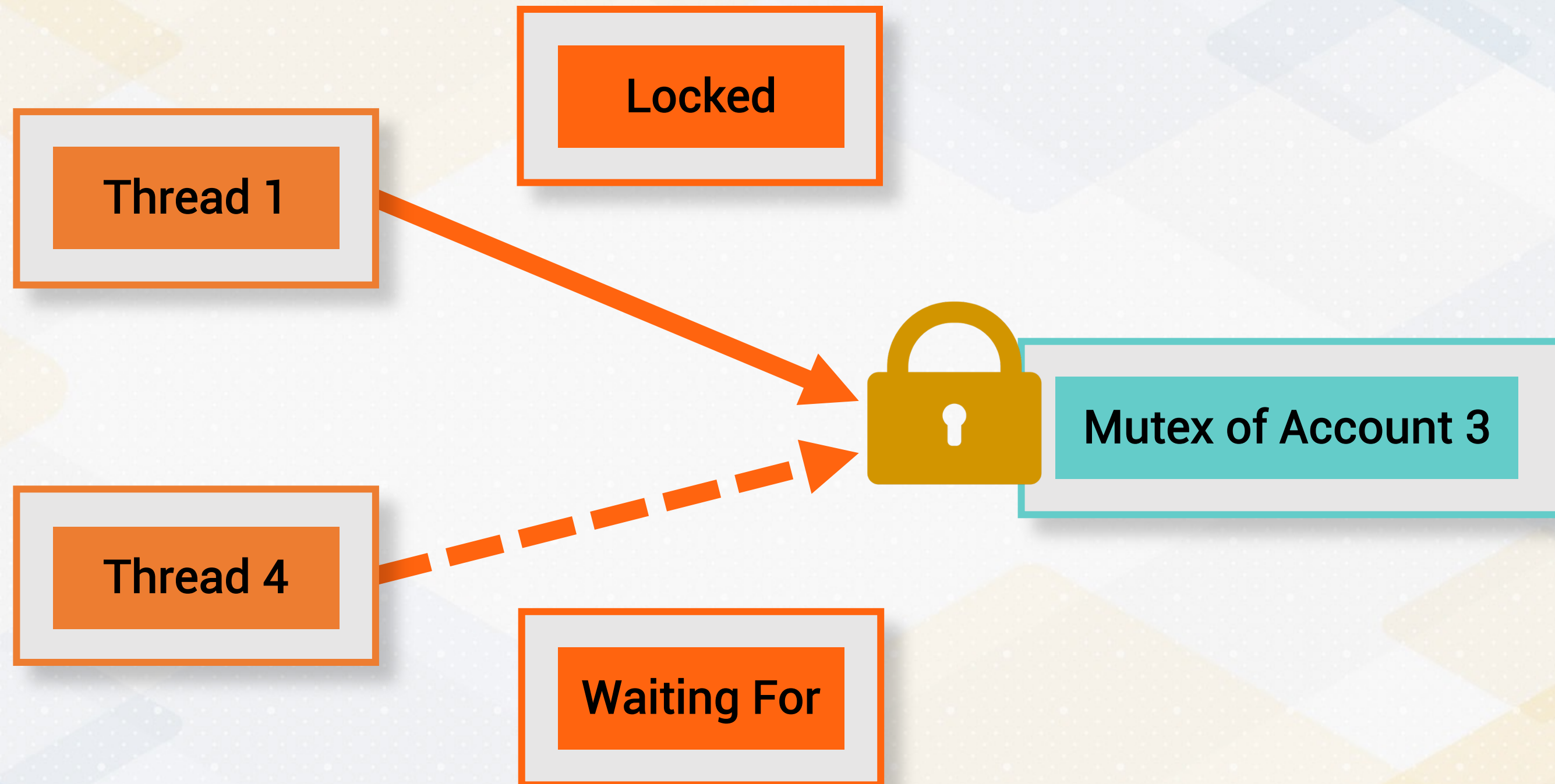
void performTransactions(int account) {
    for (int i = 0; i < 10000; ++i) {
        bankAccountMutexes[account].lock(); // Lock only the mutex for specified account
        bankAccounts[account] += 10; // Deposit
        bankAccounts[account] -= 10; // Withdrawal
        bankAccountMutexes[account].unlock(); // Unlock the mutex for the specified account
    }
}
```

Manual Locking and Unlocking

Manual mutex locking and unlocking comes with risk

```
void performTransactions(int account) {  
    for (int i = 0; i < 10000; ++i) {  
        bankAccountMutexes[account].lock(); // Lock only the mutex for specified account  
        bankAccounts[account] += 10; // Deposit  
        bankAccounts[account] -= 10; // Withdrawal  
        bankAccountMutexes[account].unlock(); // Unlock the mutex for the specified account  
    }  
}
```


Thread Starvation



std::lock_guard

A solution to avoid forgetting to unlock a mutex is to use std::lock_guard

```
void performTransactions(int account) {  
    for (int i = 0; i < 10000; ++i) {  
        // Automatically locks  
        std::lock_guard<std::mutex> lock(bankAccountMutexes[account]);  
        bankAccounts[account] += 10; // Deposit  
        bankAccounts[account] -= 10; // Withdrawal  
        // Mutex automatically unlocked when lock goes out of scope  
    }  
}
```

std::lock_guard automatically manages locking and unlocking

RAII Principle in C++

`std::lock_guard`

another example of RAII

```
std::mutex myMutex;  
std::lock_guard<std::mutex> lock(myMutex); // Object created here
```

Resource

mutex

Object

instance of `std::lock_guard`

RAII Principle in C++

Automate resource management by
tying resource allocation to object lifespan

RAII Object	Resource Managed	Acquisition	Release
<code>std::lock_guard</code>	Mutex	Locks the mutex upon creation	Automatically releases the lock when object is destroyed.
<code>std::unique_ptr</code>	Dynamic memory	Allocates memory and takes ownership	Automatically deallocates memory
<code>std::fstream</code>	File handle	Opens a file and acquires the file handle	Closes the file and releases the file handle

Conclusion

- 2Q Policy
- Multi-Threading
- Fine-Grained Locking