



Lecture 9: Fine-Grained Locking & Debugging



Logistics

- Point Solutions App
 - Session ID: **database**
- Programming assignment 2 released (Gradescope)
- Exercise sheet 1 will be released soon (Canvas)

Recap

- 2Q Policy
- Multi-Threading
- Synchronization

Lecture Overview

- Fine-Grained Locking
- Debugging
- Database Configuration

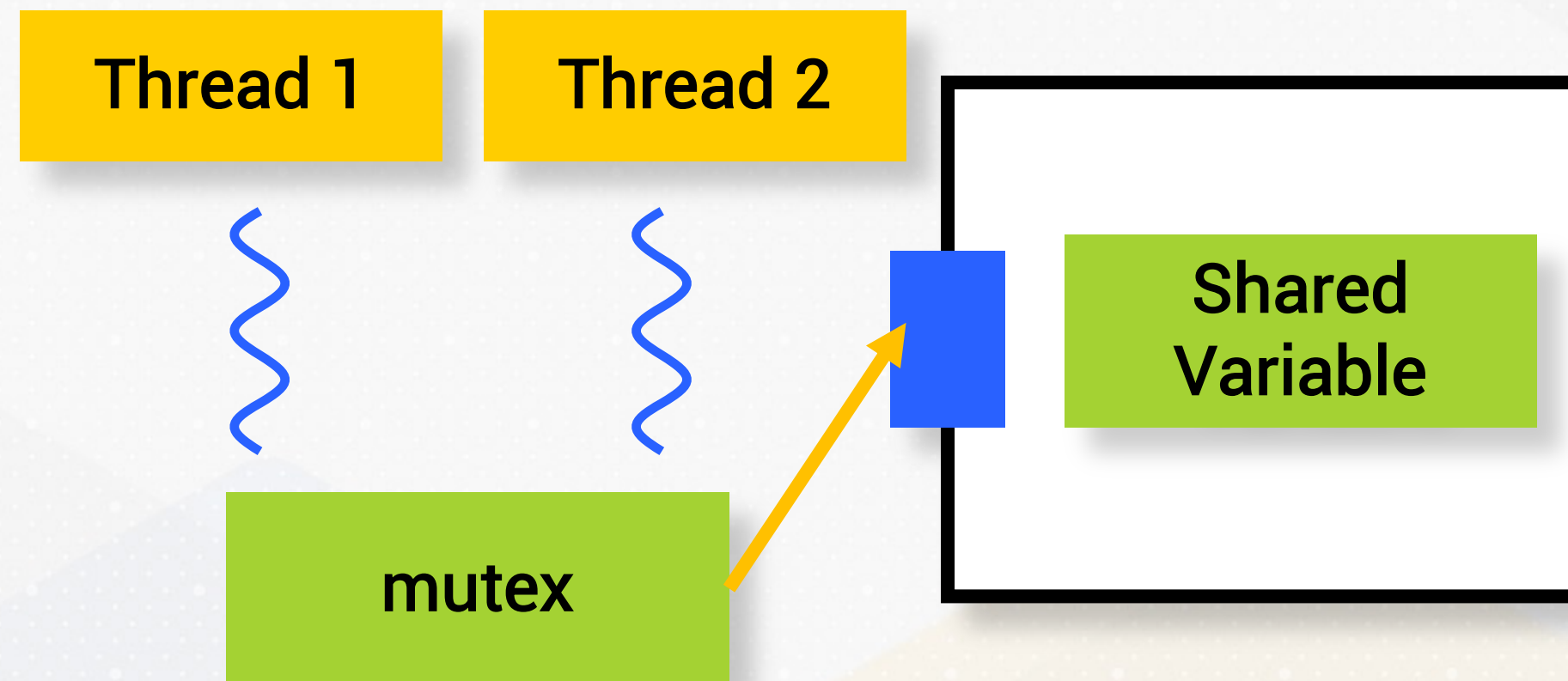
Mutex



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

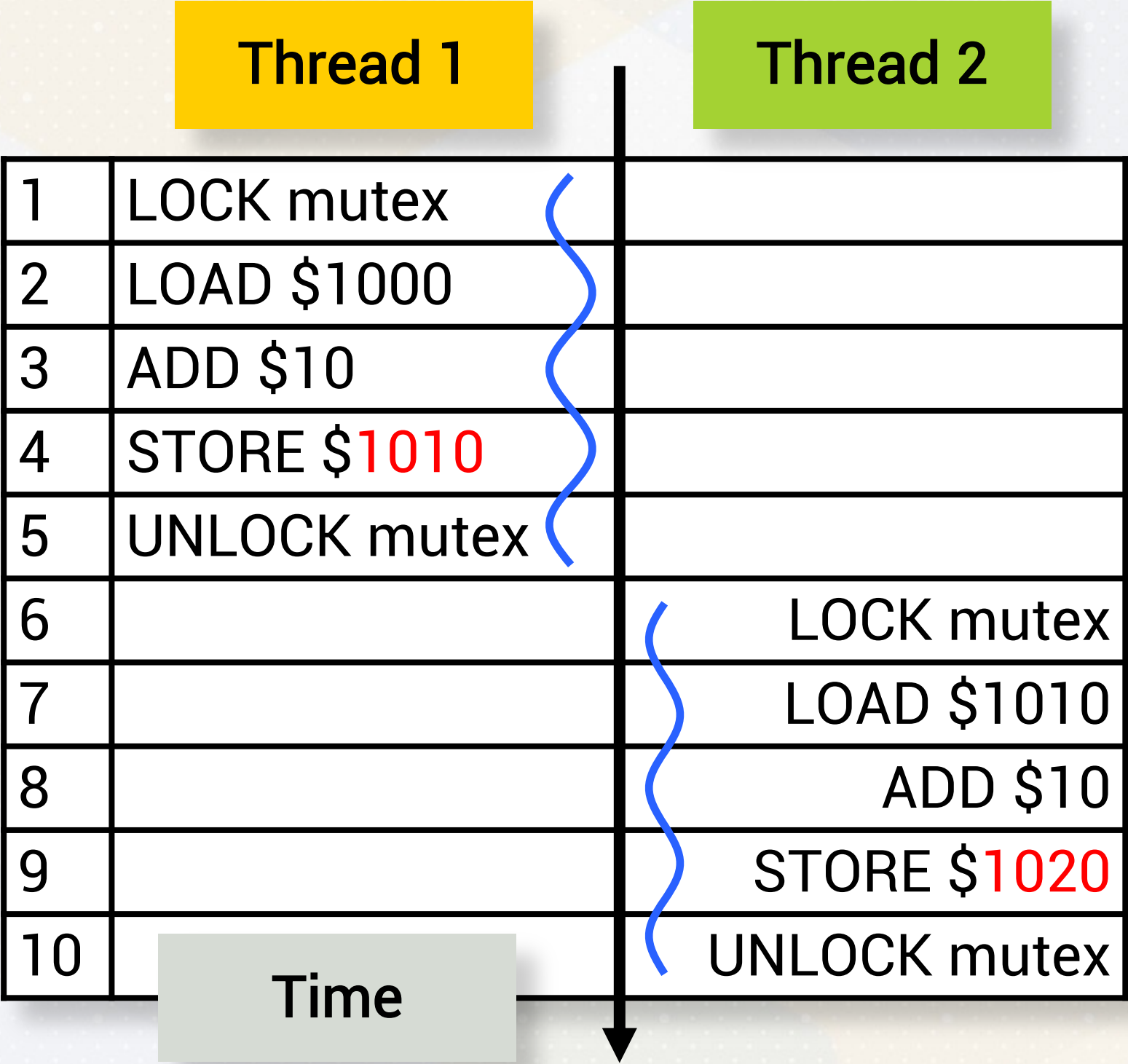
```
; Locking the spinlock
retry:
    ; EAX is set to the expected old value (unlocked = 0)
    MOV EAX, 0
    ; EBX is set to the new value to store if comparison is successful
    (locked = 1)
    MOV EBX, 1
    ; Atomically compare [mutex] to 0, if equal replace [mutex] with 1
    LOCK CMPXCHG [mutex], 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

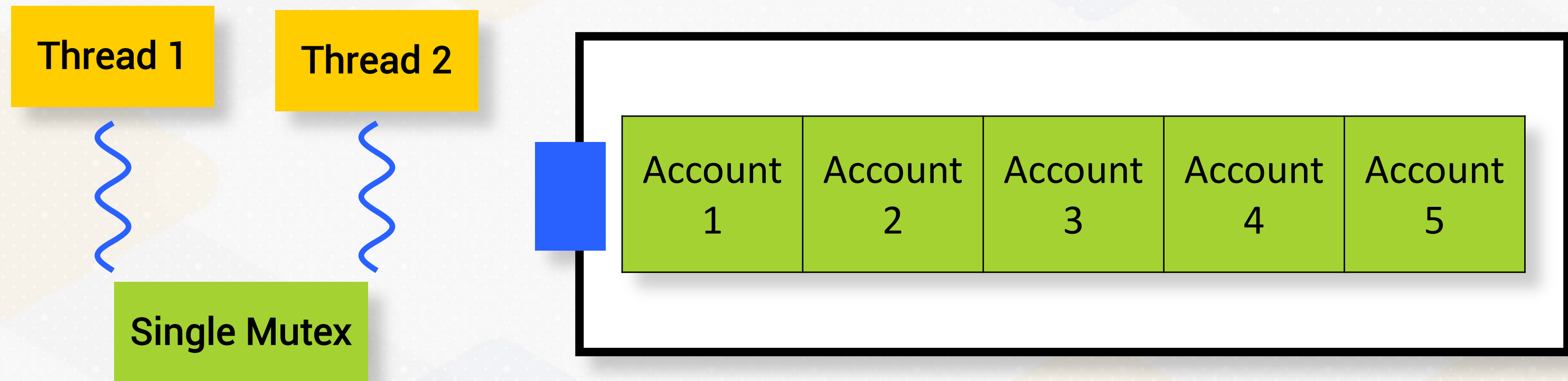


Multiple Bank Accounts

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

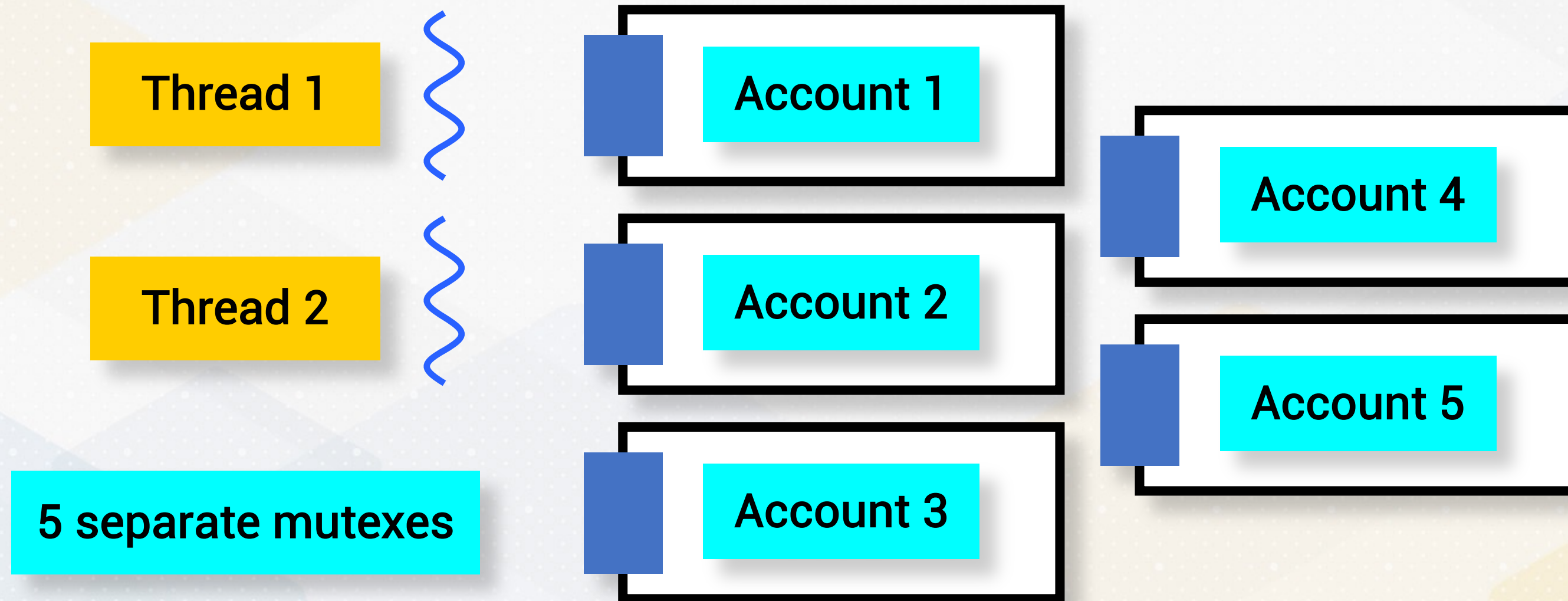


Fine-Grained Locking



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

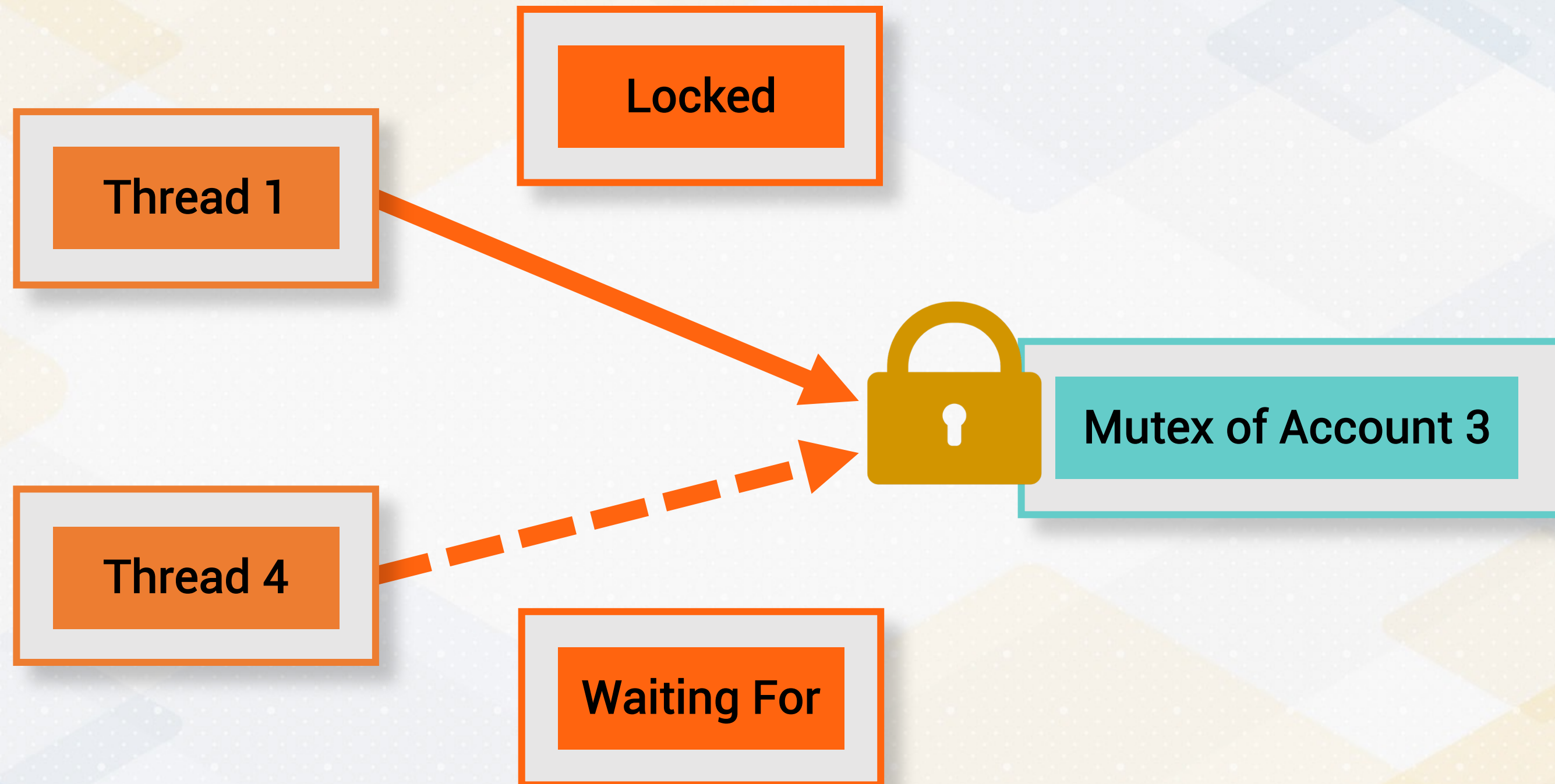
void performTransactions(int account) {
    for (int i = 0; i < 10000; ++i) {
        bankAccountMutexes[account].lock(); // Lock only the mutex for specified account
        bankAccounts[account] += 10; // Deposit
        bankAccounts[account] -= 10; // Withdrawal
        bankAccountMutexes[account].unlock();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



Debugging Policies & Mechanisms



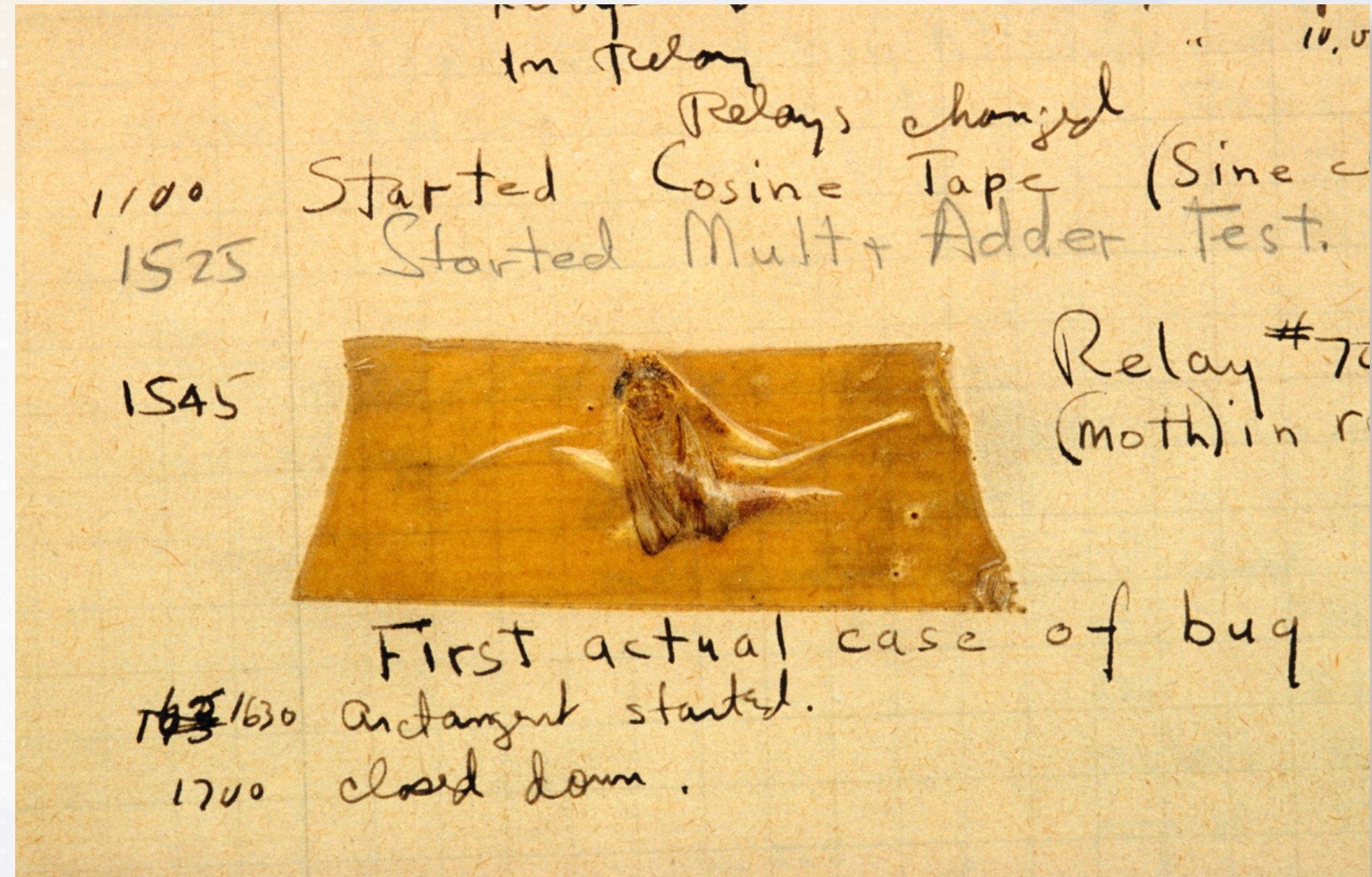
Origin of Debugging



GRACE HOPPER

James S. Davis

Public domain, via Wikimedia Commons



Log Book with Computer Bug

National Museum of American History

Public Domain under Creative Commons Zero (CC0) license

Debugging in Database Systems



Bugs

Data Corruption

Performance
Degradation

System Crashes



Tools

GDB

Print Statements

Using GDB for Debugging

GDB

a tool to see what is going on 'inside' the program while it executes or at the moment it crashed

```
g++ -g buzzdb.cpp -o buzzdb  
gdb ./buzzdb
```


Using GDB for Debugging

```
#include <iostream>

int add(int x, int y) {
    return x + y; // Set a breakpoint here
}

int main() {
    int sum = 0;
    for (int i = 1; i <= 10; ++i) {
        sum = add(sum, i);
        std::cout << "Sum: " << sum << endl;
    }
    return 0;
}
```


GDB Commands

Run:
Starts the
program
under GDB

Next: Moves
Execution to
the next line

Print:
Displays
values of
variables

Break: Set a
breakpoint at
a specific line
or function

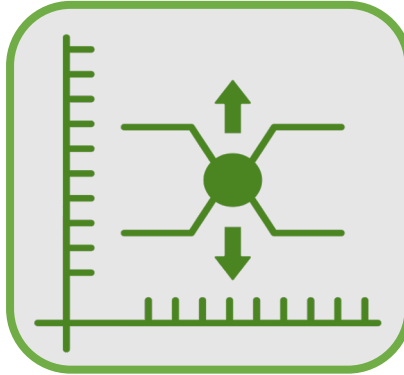
GDB Commands

Continue: Continue running the program until the next breakpoint

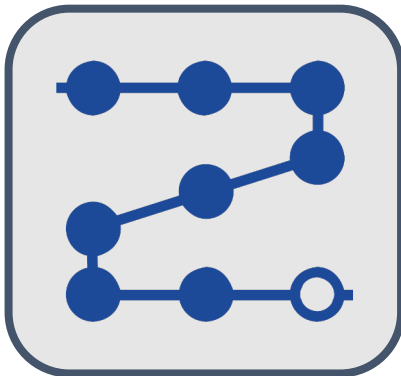
Backtrace (bt):
Shows the call stack to see how the program reached current point

Info:
Displays local variables in the current stack frame

Breakpoints and Backtrace



Breakpoints: Temporarily halt the program execution at a specific point



Backtrace: Reveals the path taken by the program to reach current execution point

Illustrative GDB Session

```
(gdb) break add
Breakpoint 1 at 0x...: file main.cpp, line 4.
(gdb) run
Starting program: /a.out

Breakpoint 1, add (x=0, y=1) at main.cpp:4
4         return x + y;
(gdb) info locals
x = 0
y = 1
(gdb) next
5     }
```


Illustrative GDB Session

```
(gdb) print x
$1 = 0
(gdb) print y
$2 = 1
(gdb) continue
Continuing.
Sum: 1
(gdb) backtrace
#0  add (x=1, y=2) at main.cpp:4
#1  0x... in main () at main.cpp:8
```


Using Print Statements for Debugging

Print statements allow you to track how your program's execution flow and how variables change over time

```
std::cout << "Loading page : " << page_id << std::endl;  
std::cout << "Evicting page: " << evictedPageId << std::endl;
```

Overload << operator

Print statements allow you to track how your program's execution flow and how variables change over time

```
// Define the operator<< function
std::ostream &operator<<(std::ostream &os, const Person &person) {
    os << "Person[name=" << person.name << ", age=" << person.age << "]\n";
    return os;
}

int main() {
    Person alice("Alice", 50);
    std::cout << alice << std::endl;
}
```

OUTPUT: Person[name = Alice, age = 50]

Debugging Principle: Stress Testing

Small Buffer Pool

makes it easier
to observe and
understand
eviction
decisions

Limited Slots

edge-cases such
as buffer pool
arise much
sooner

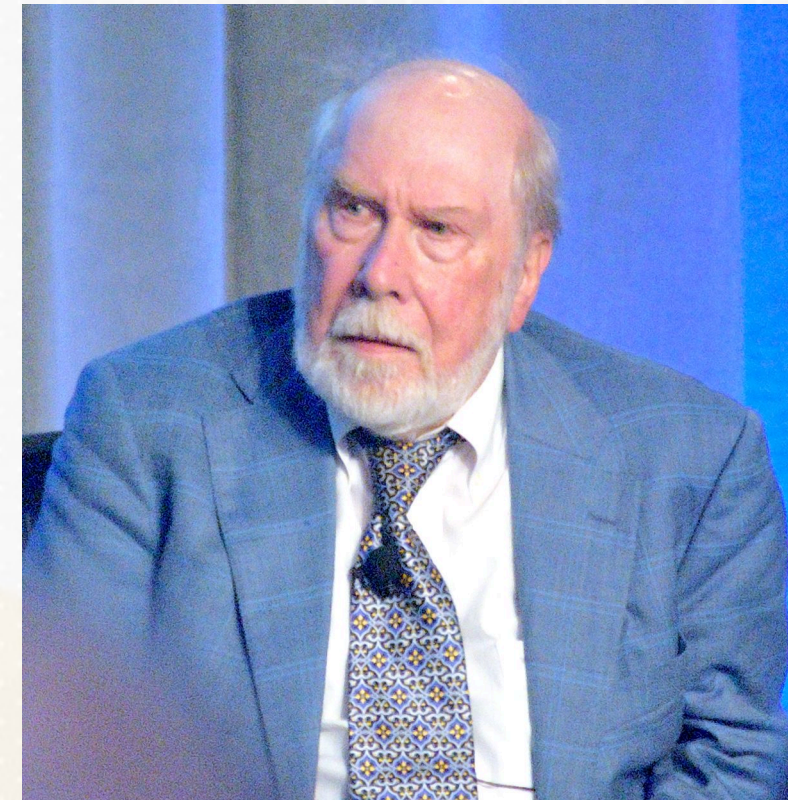
```
constexpr size_t MAX_PAGES_IN_MEMORY = 10; // For stress testing  
BufferManager bufferManager(MAX_PAGES_IN_MEMORY);
```


Debugging Principle: Modular Design

Modularity

clear interfaces and separation of concerns

```
class BufferManager {  
    StorageManager storage_manager;  
    std::unique_ptr<Policy> policy;  
    ...  
};
```



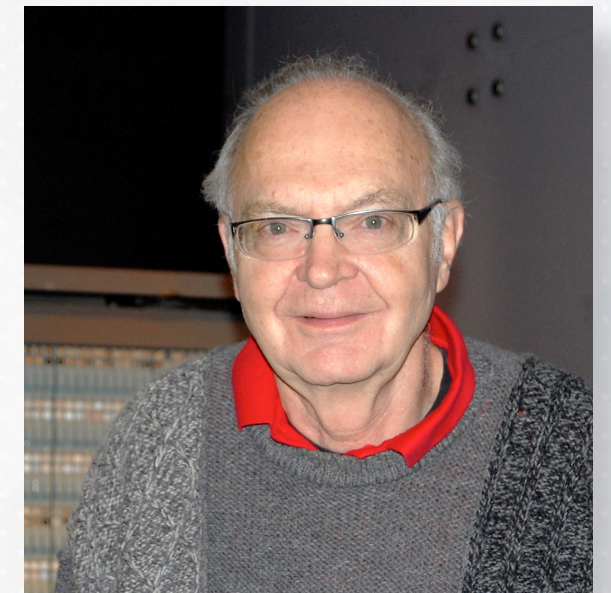
NIKLAUS WIRTH

orcmid, CC BY 2.0
via Wikimedia Commons

Debugging Principle: Literate Programming

programs must be understandable by humans first.
documentation is an integral part of programming.

```
/**
 * Function: readPage
 * Purpose: Reads a page from the buffer pool.
 *          If the page is not currently in memory,
 *          it loads the page from disk into the buffer pool.
 * Inputs: pageId - The ID of the page to be read
 * Outputs: A reference to a Buffer containing the page data
 * Throws: DiskIOException if the page cannot be read
 */
Buffer &BufferManager::readPage(PageId pageId) {
    ...
}
```



DONALD KNUTH

Alex Handy, CC BY-SA 2.0
via Wikimedia Commons

Debugging Principle: Robustness

Robustness

ensure that the system can withstand failures, like failed disk I/O operations

```
// Load frame from disk into memory bool
BufferManager::loadFrame(int frameId, int pageId) {
    try {
        // Call the storage manager to read data from disk into the buffer frame
        storageManager.readPage(pageId, pool[frameId].data);
    } catch (const DiskIOException &e) {
        std::cerr << "Failed to load page from disk: " << e.what() << std::endl;
        return false;
    }
    return true;
}
```


Database Configuration

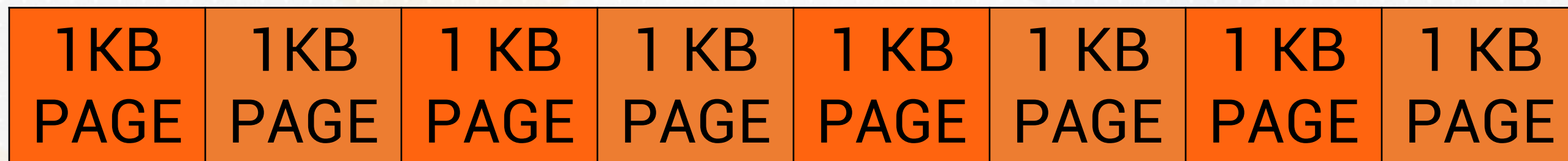


Database Configuration

Parameter	Value
PAGE_SIZE	1 KB
MAX_SLOTS	128 slots per page
MAX_PAGES_IN_MEMORY	10 pages
...	...

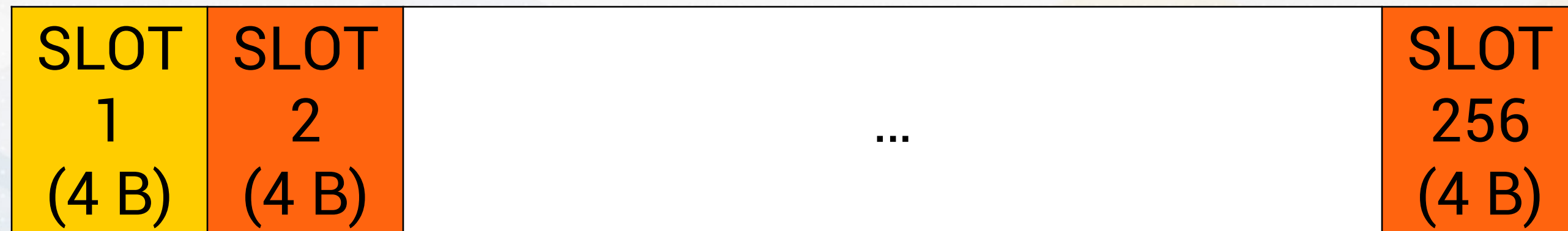
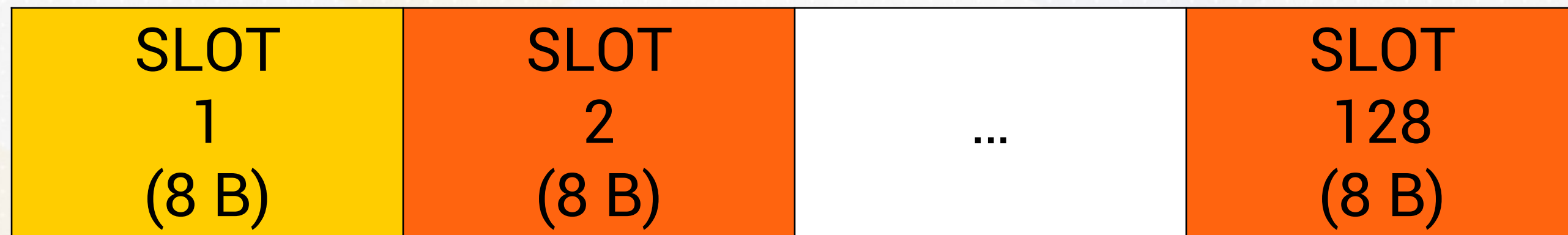
PAGE_SIZE

8KB BUFFER POOL



MAX_SLOTS

1KB SLOTTED PAGE



MAX_PAGES_IN_MEMORY

BUFFER POOL

PAGE 1	PAGE 2	PAGE 3	PAGE 4
-----------	-----------	-----------	-----------

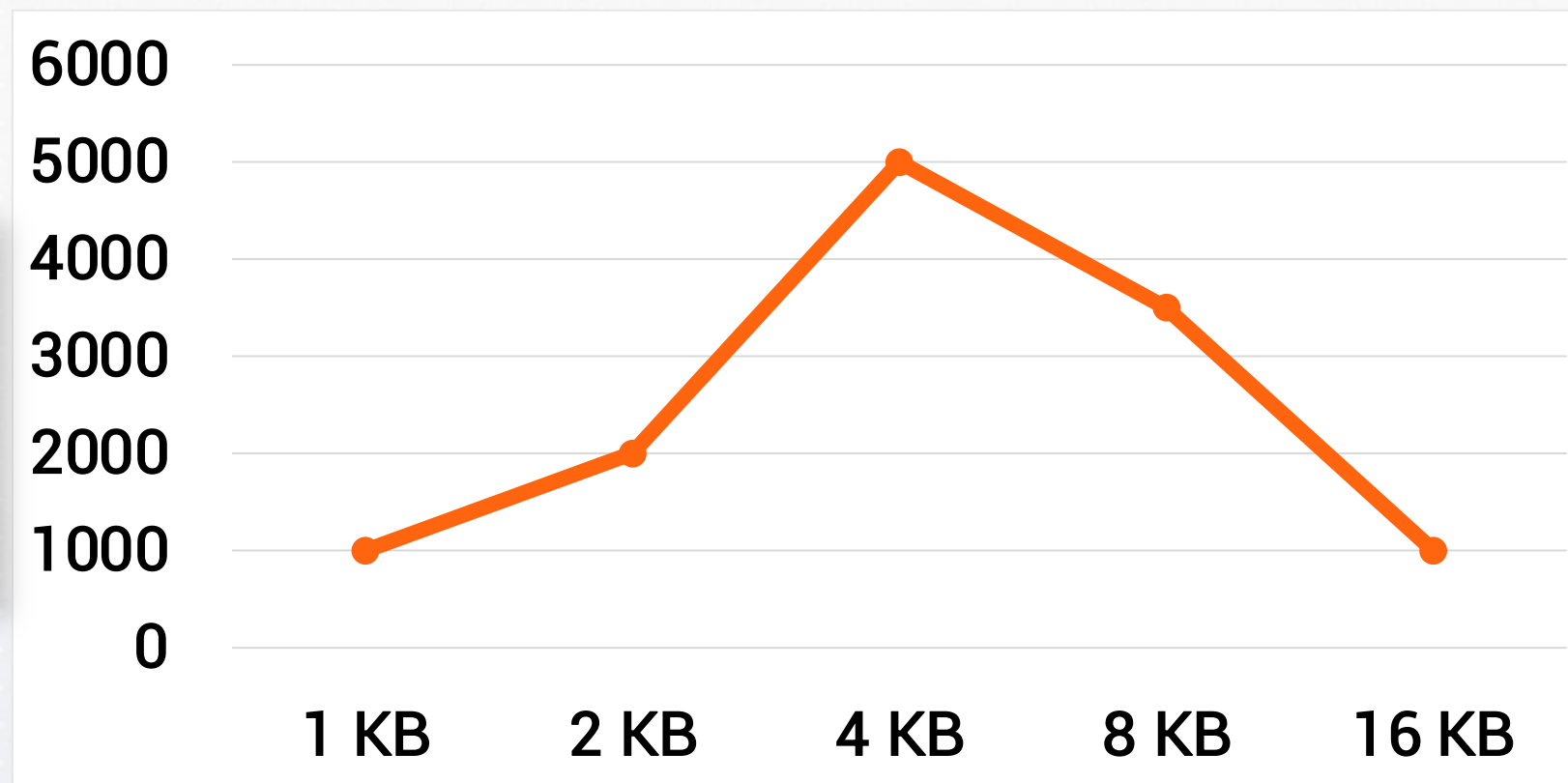


PAGE 1	PAGE 2	PAGE 3	PAGE 4	PAGE 5	PAGE 6	PAGE 7	PAGE 8
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

Challenges in Configuration Tuning

Most parameters like `PAGE_SIZE` have a “sweet spot”

THROUGHPUT
(TRANSACTIONS
PER SECOND)



PAGE SIZE PARAMETER

Challenges in Configuration Tuning

PostgreSQL's default configuration has low buffer pool size

System Parameter	Value
BUFFER_POOL_SIZE	128 MB



System Parameter	Value
BUFFER_POOL_SIZE	16 GB

Immutable vs. Mutable Parameters

System Parameter	Mutable or Immutable
PAGE_SIZE	Immutable
BUFFER_POOL_SIZE	Mutable

Cloud Database Tuning

Cost vs Performance

Memory Size	Throughput	Cost	Price-Performance
8 GB	100 transactions per second	\$0.05 per hour	\$0.139 per million transactions
16 GB	200 transactions per second	\$0.15 per hour	\$0.208 per million transactions

Tools for Configuration Tuning

Automatically recommend configuration
based on hardware and application

Parameter	Value
TOTAL_MEMORY	16 GB
NUMBER_OF_CPUS	16
STORAGE_TYPE	SSD / HDD / SAN
APPLICATION_TYPE	Type 1 / Type 2 / Type 3

Database Application Type

Application Type	Performance Bound by	Database Size Relative to DRAM	Workload Characteristics
Online Transaction Processing (OLTP)	CPU- or I/O-bound	DB slightly larger than DRAM ~ 1 TB	20-40% short transactions, some long transactions and queries
Online Analytical Processing (OLAP)	I/O- or DRAM-bound	DB much larger than DRAM ~ 4 TB	Large complex reporting queries; Also known as data warehouse
Web Application (WEB)	CPU-bound	DB much smaller than DRAM ~ 4 GB	90% or more simple queries

Reinforcement Learning for Configuration Tuning

- Agent interacts with database by trying different configuration settings
- Observes the impact on performance such as throughput

State	Current settings of database parameters and the current workload characteristics
Action	Increase or decrease the value of some parameter
Reward	Increase in throughput or decrease in price-performance metric

Conclusion

- Fine-Grained Locking
- Debugging
- Database Configuration