Lecture 9: Multi-Threading & Synchronization


Logistics

- Point Solutions App
  - Session ID: **database**
- Programming assignment 2 due on **Sep 24** (Gradescope)
- Exercise sheet 1 due on **Sep 24** (Gradescope)

Recap

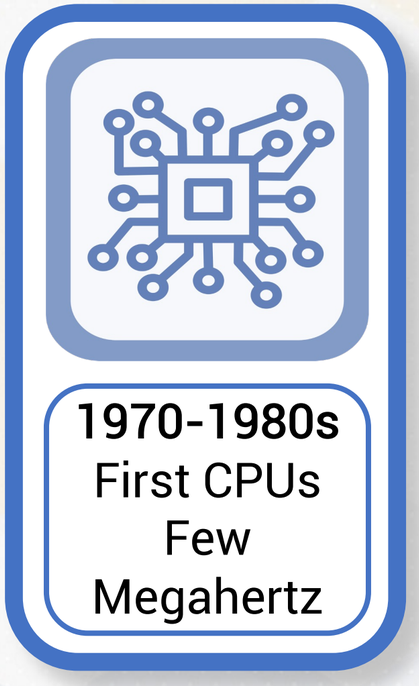
- Cache Replacement Policy
- Buffer Pool Flooding
- 2Q Policy

Lecture Overview

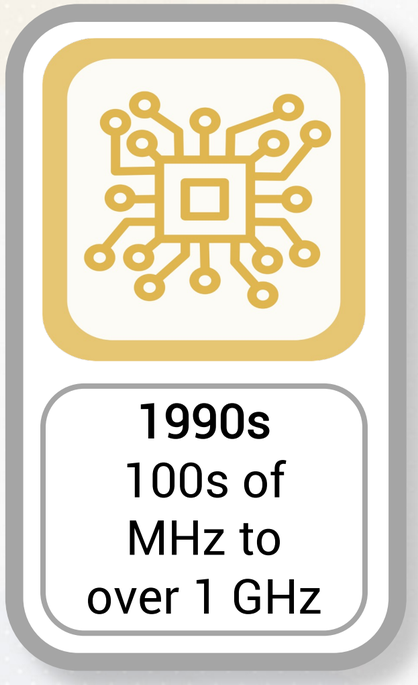
- Multi-Threading
- Synchronization
- Fine-Grained Locking
- Debugging

Multi-Threading

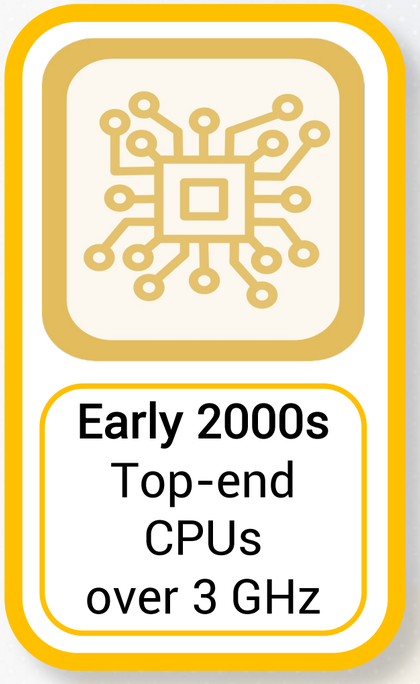
History of CPUs



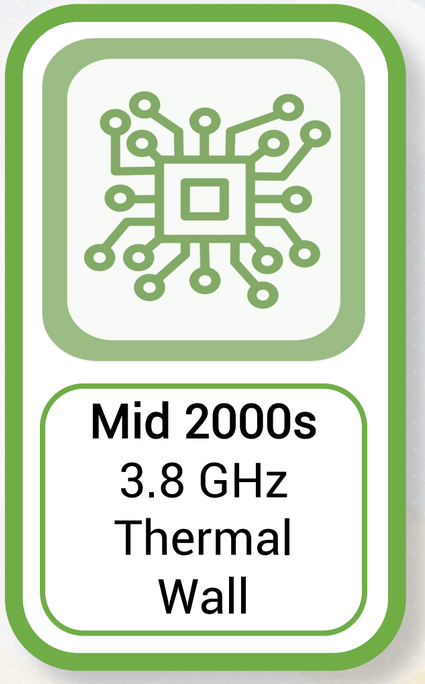
1970-1980s
First CPUs
Few
Megahertz



1990s
100s of
MHz to
over 1 GHz

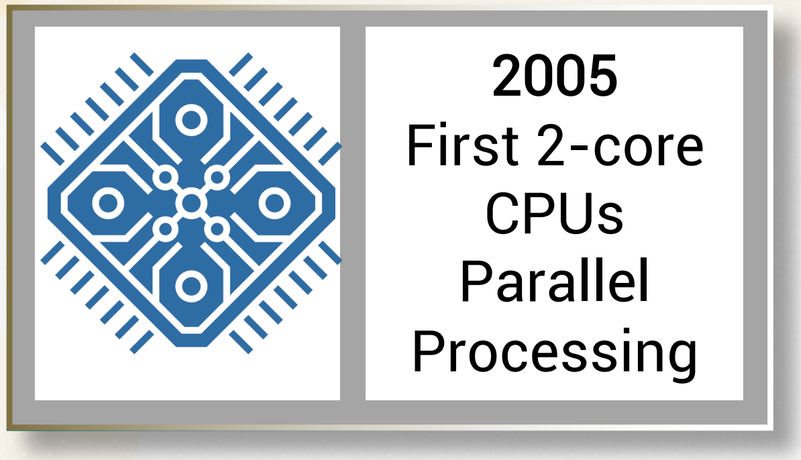


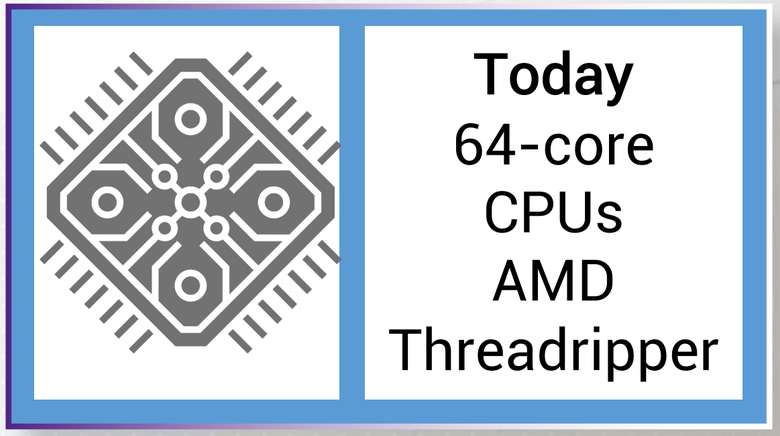
Early 2000s
Top-end
CPUs
over 3 GHz



Mid 2000s
3.8 GHz
Thermal
Wall

History of CPUs



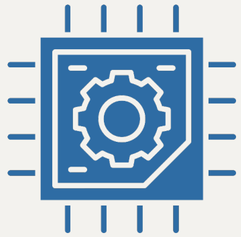


Multi-Core CPUs

Clock Speed

Rhythm of Drumbeat

One Instruction Per Beat



Single-Core CPU

Faster Drumbeat

Caused Overheating & Other Issues

Multi-Core CPU

Multiple Drummers

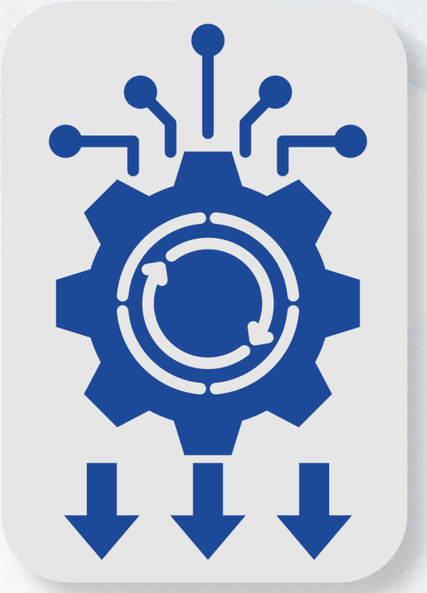
Same-Time Multiple Instructions

Parallel Processing

Threading

Thread: Database Instructions for CPU

Multi-Core CPUs: Threads Used 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 accessed by multiple threads representing different bank 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

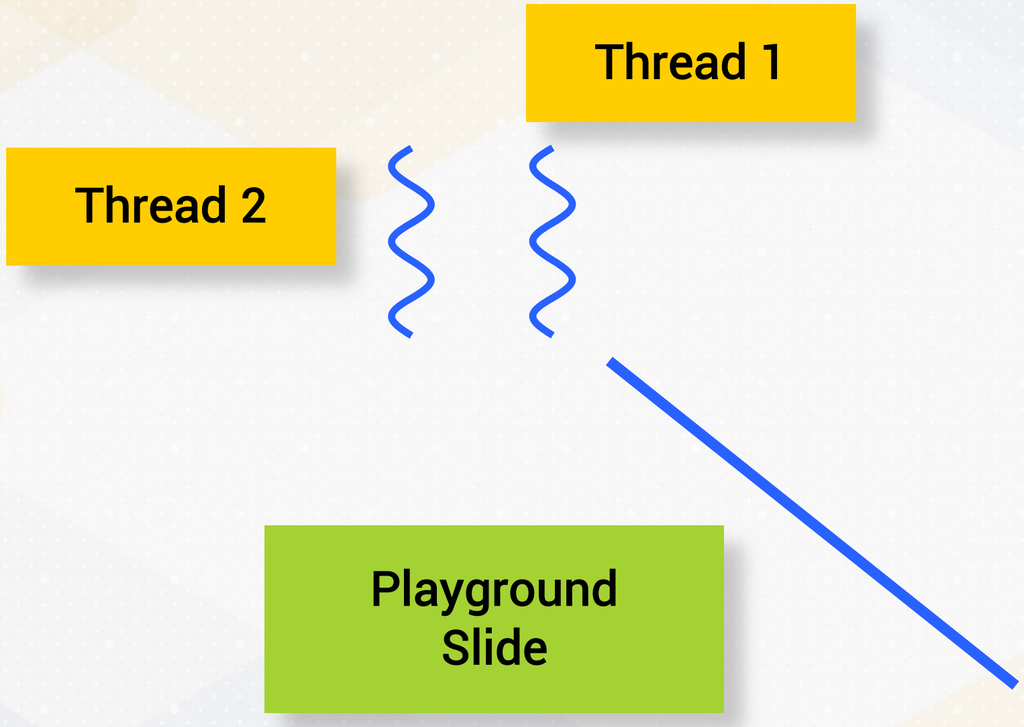
```
int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 8; ++i) {
        // Initiate transactions without synchronization
        threads.emplace_back(performTransactions);
    }
    for (auto& thread : threads) {
        thread.join(); // Wait for all threads to finish
    }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

Need for Synchronization

Challenge	Without proper synchronization, simultaneous deposits and withdrawals can lead to an inaccurate balance – race condition

```
Run 1
Expected Output: $1000
Actual Output  : $910

Run 2
Expected Output: $1000
Actual Output  : $1020
```

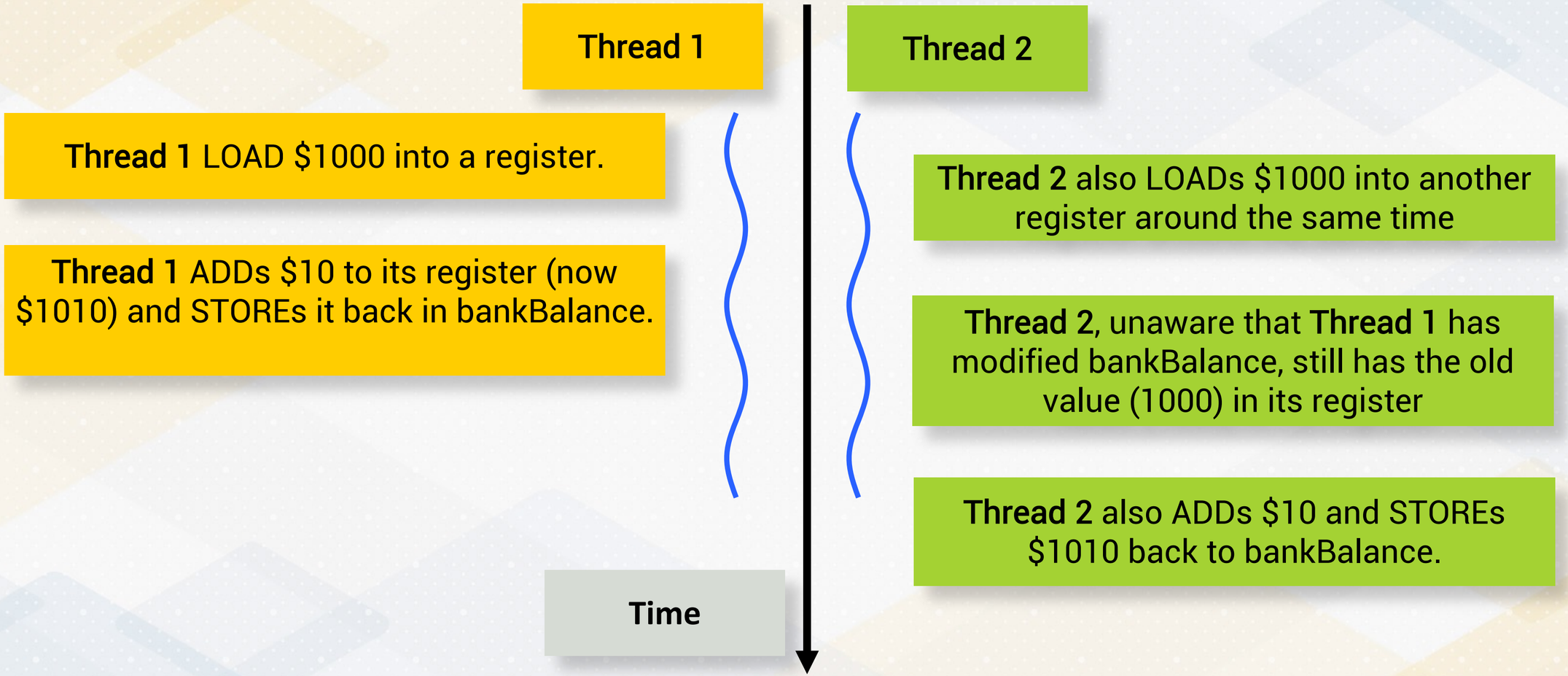
Playground Slide Example



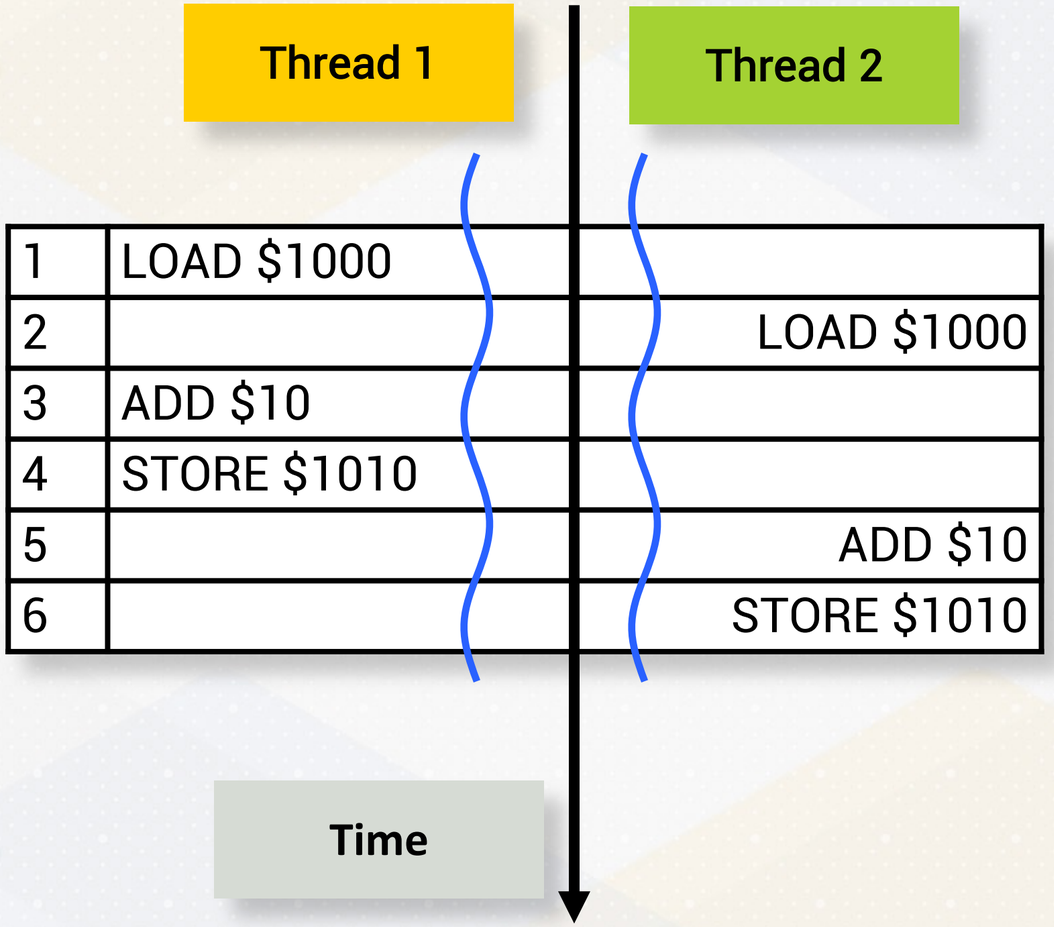
Assembly Level Explanation

- bankBalance += 10 maps to three assembly instructions.
- LOAD value of bankBalance from memory to register
  - Assembly: MOV EAX, [bankBalance]
- ADD 10 to increment the value in the register
  - Assembly: ADD EAX, 10
- STORE the new value in register back to the memory location of bankBalance
  - Assembly: MOV [bankBalance], EAX

Race Condition



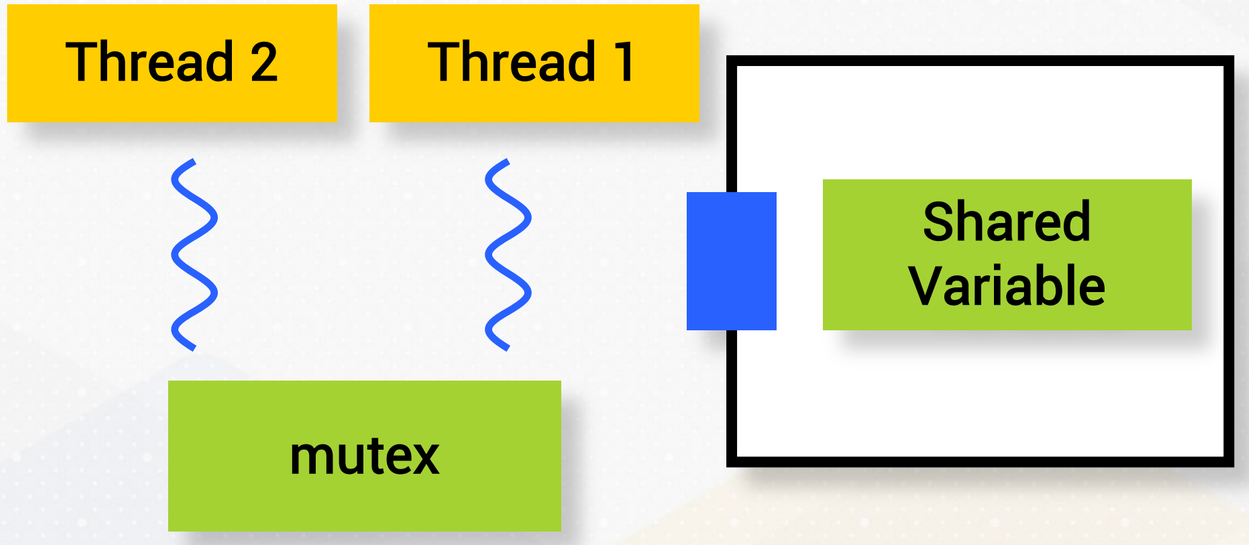
Non-Atomic Load-ADD-STORE Sequence



Synchronization

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

- Lock and unlock operations form critical code sections
- Only one thread enters 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
```

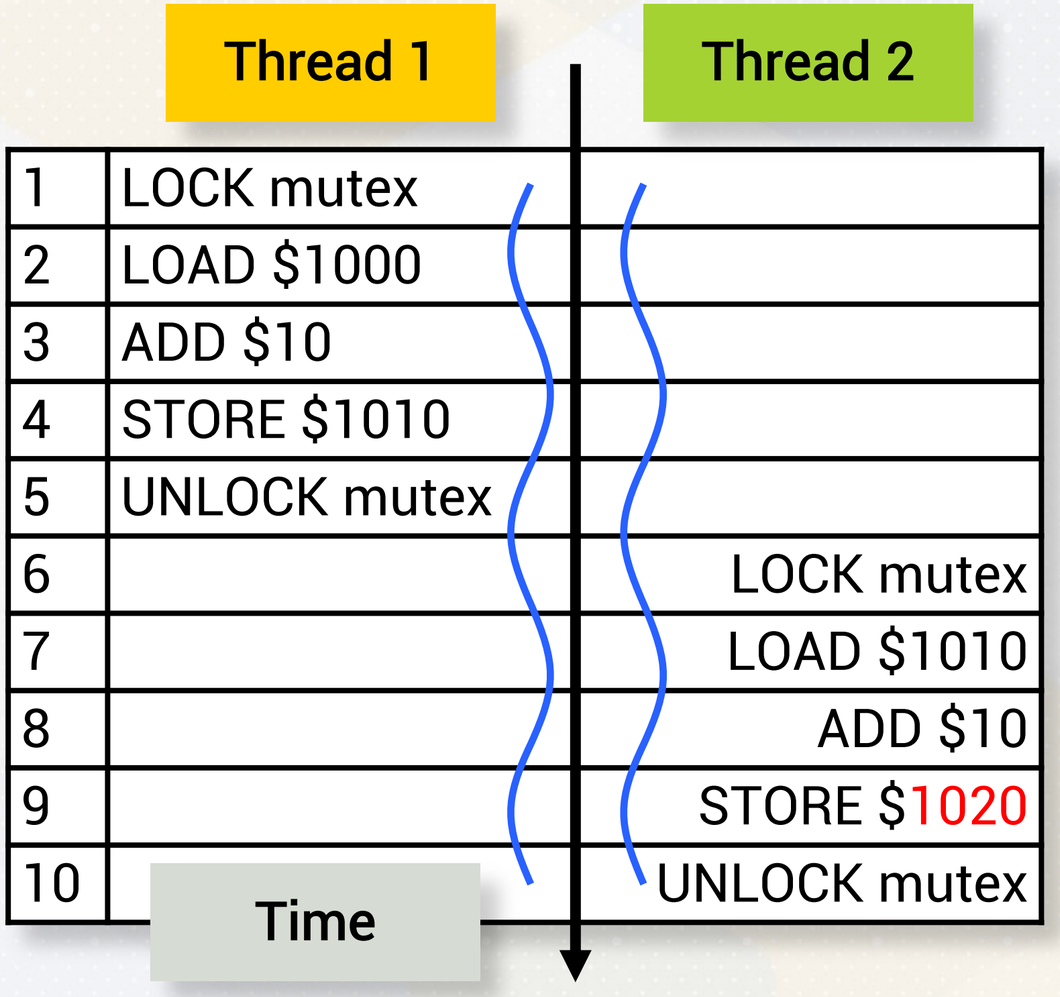
Mutex Operations at Assembly Level

```
; Locking the Mutex
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
```

Mutex 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

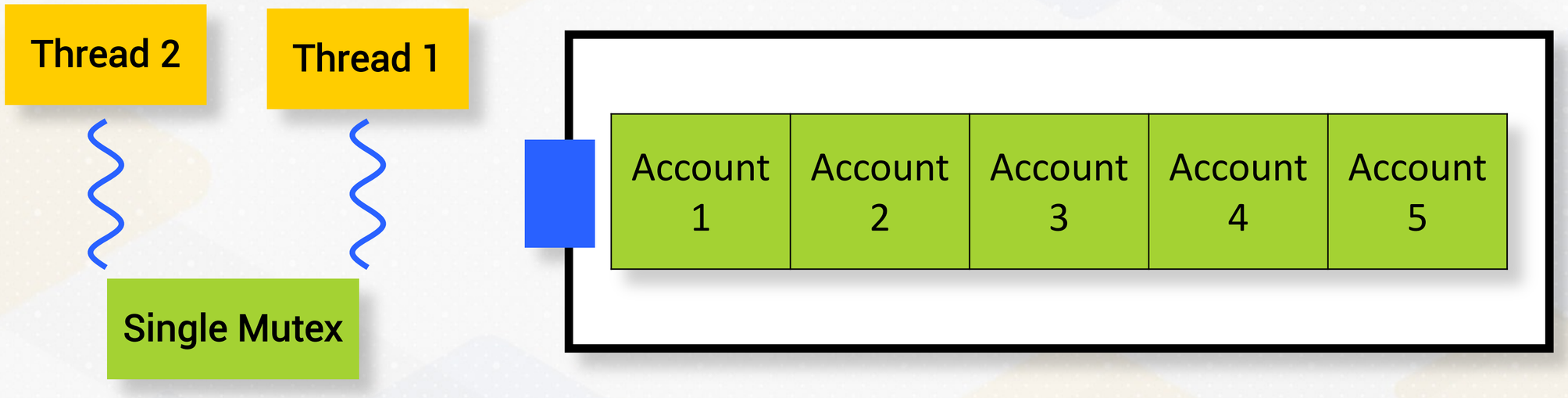


Multiple Bank Accounts

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

```
std::mutex bankMutex;
int bankAccounts[5] = {1000, 2000, 3000, 4000, 5000}; // Initial balances

void performTransactions(int account) {
    for (int i = 0; i < 10000; ++i) {
        bankMutex.lock(); // Single mutex for all accounts
        bankAccounts[account] += 10; // Deposit
        bankAccounts[account] -= 10; // Withdrawal
        bankMutex.unlock();
    }
}
```

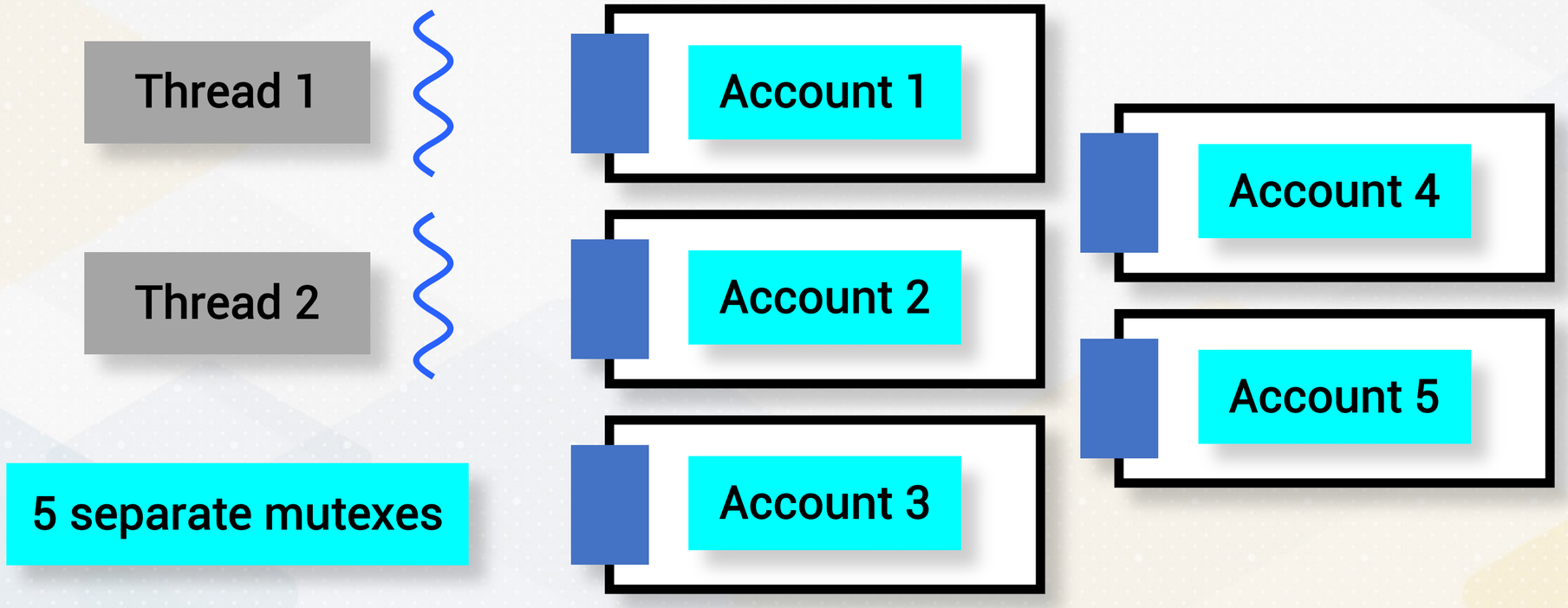
Multiple Bank Accounts



Fine-Grained Locking

Fine-Grain Locking

Use a separate mutex for each bank account to improve concurrency



Fine-Grain Locking

Non-conflicting transactions can now run in parallel

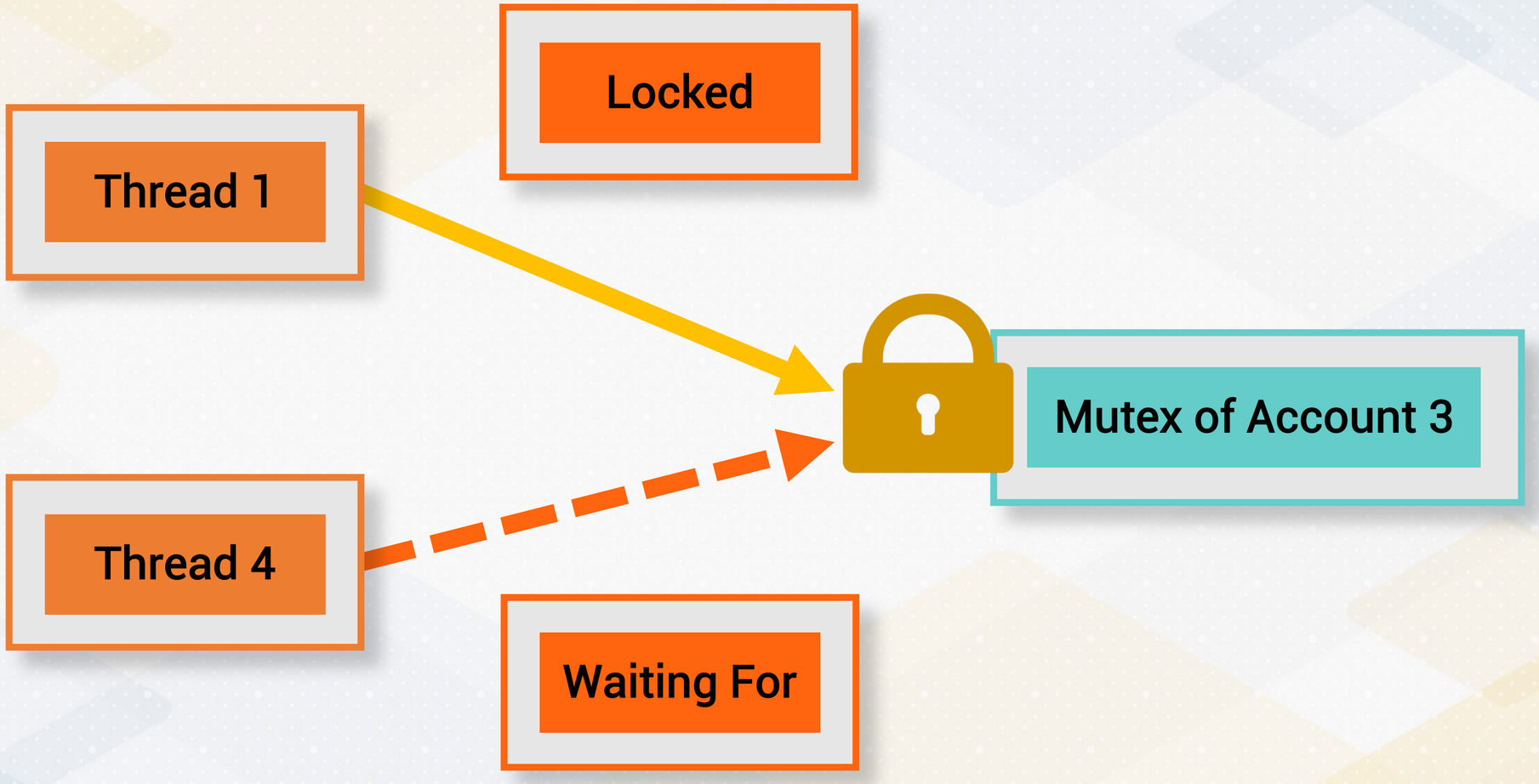
```
std::mutex bankAccountMutexes[5]; // A mutex for each bank account
int bankAccounts[5] = {1000, 2000, 3000, 4000, 5000}; // Initial balances

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 locking and unlocking the mutex 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
        // Forgot to unlock the mutex
    }
}
```

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 mutex locking and unlocking

RAII Principle

std::lock_guard	another example of C++ RAII

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

Resource	lock on the mutex

Object	an instance of std::lock_guard

RAII Principle in C++

Simplify resource management by tying resource allocation to object lifespan

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

Debugging

Debugging

- Bugs can lead to data corruption, performance degradation, and system crashes.
- Tools for debugging
  - GDB
  - print statements

Origin of Debugging

GRACE HOPPER

James S. Davis
Public domain, via Wikimedia Commons

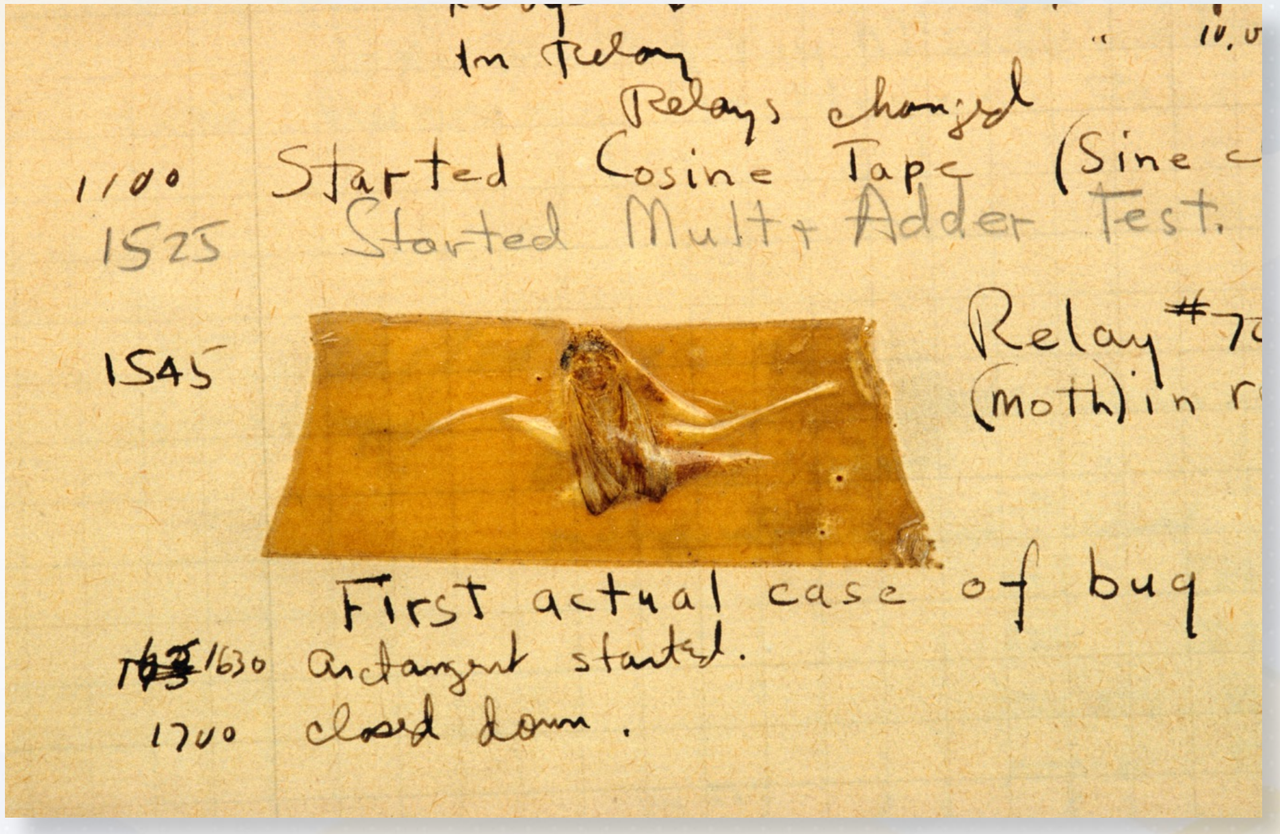


IMAGE NEEDS ATTRIBUTION

Using GDB for Debugging

- **gdb** is a tool that allows developers to see what is going on 'inside' a program while it executes or at the moment it crashed.

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

Using GDB for Debugging

- Consider the following code snippet:

```
#include <iostream>
using namespace std;
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);
        cout << "Sum: " << sum << endl;
    }
    return 0;
}
```

GDB Commands

- run: Start the program.
- next: Execute the next line.
- print: Display the value of a variable.
- break: Set a breakpoint at a specific line or function.
- continue: Continue running the program until the next breakpoint.
- backtrace (bt): Show the call stack to see how the program reached current point.
- info locals: Display 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.

Examples of GDB Session

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

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   }
```

Examples of 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 << "]";
    return os;
}
int main() {
      Person alice("Alice", 30);
      std::cout << alice << std::endl;
}
Person[name=Alice, age=30]
```

Conclusion

- Multi-Threading
- Synchronization
- Fine-Grained Locking
- Debugging