Mini-Project: Concurrent Database Synchronization

Advanced Operating Systems & Concurrent Programming

Race Conditions, Mutexes, Monitors, and Lock-Free Programming

Due Date: [To be announced]

Abstract

In this project, you will explore the fundamental challenges of concurrent programming by implementing synchronization mechanisms for a multi-threaded in-memory database. You will witness race conditions firsthand, implement multiple synchronization strategies (mutexes, monitors, reader-writer locks, and channels), and analyze their performance trade-offs. This hands-on experience bridges theoretical operating systems concepts with practical concurrent systems engineering.

1 Introduction: The Concurrency Challenge

Modern computing systems are inherently concurrent. Multi-core processors, distributed systems, and cloud infrastructure all rely on concurrent execution to achieve high performance. However, concurrency introduces one of computer science's most challenging problems: **coordinating access to shared resources**.

Key Concept

The Fundamental Problem: When multiple threads access shared data concurrently without proper synchronization, the system enters an undefined state. The outcome depends on the precise timing of thread execution—a phenomenon known as a *race condition*.

1.1 Real-World Impact

Race conditions aren't just theoretical—they cause real production failures:

- **Therac-25 (1985-1987)**: Race conditions in radiation therapy software led to patient deaths
- **Mars Pathfinder (1997)**: Priority inversion caused system resets on Mars
- **Knight Capital (2012)**: A race condition in trading software caused $440 million loss in 45 minutes
- **Cloudflare (2017)**: Memory corruption from race conditions leaked sensitive data

1.2 Learning Objectives

By completing this project, you will:

1. **Understand** the root causes of race conditions and data races

2. **Implement** multiple synchronization primitives from first principles
3. **Analyze** performance trade-offs between different concurrency control mechanisms
4. **Apply** Go's race detector and testing framework
5. **Evaluate** when to use shared-memory vs. message-passing concurrency models

2 Problem Description

You are provided with an **intentionally broken** implementation of a concurrent in-memory key-value database. The database supports basic operations but has *zero synchronization*—making it a perfect case study for understanding race conditions.

2.1 The Database System

The database is a simple key-value store supporting ACID-like transactions:

Listing 1: Database Interface

```go
type Database interface {
    BeginTransaction() *Transaction
    Read(tx *Transaction, key string) (int, bool)
    Write(tx *Transaction, key string, value int)
    Update(tx *Transaction, key string, delta int) bool
    Delete(tx *Transaction, key string) bool
    Commit(tx *Transaction)
}
```

2.2 Demonstration of Race Conditions

The unsynchronized implementation includes three carefully designed scenarios that reliably trigger race conditions:

2.2.1 Scenario 1: The Lost Update Problem

Multiple goroutines increment a shared counter. Without synchronization, updates are lost:

Listing 2: Lost Update Example

```go
// Thread 1 reads: counter = 100
// Thread 2 reads: counter = 100
// Thread 1 writes: counter = 101
// Thread 2 writes: counter = 101  // Lost Thread 1's update!
// Expected: 102, Actual: 101
```

Warning

With 10 goroutines each performing 100 increments, you might expect a final value of 1000. The unsynchronized version typically produces values between 200-800, losing 20-80% of updates!

2.2.2 Scenario 2: The Bank Transfer Anomaly

Money transfers between accounts should preserve the total balance. Race conditions cause money to vanish:

2.2.3 Scenario 3: Inconsistent Reads

Readers observe partial updates when writers modify multiple related values:

Listing 3: Dirty Read Example

```
// Writer: Set data_1 = 500, data_2 = 500
db.Write(tx, "data_1", 500)
// Reader executes here: reads data_1 = 500, data_2 = 100
db.Write(tx, "data_2", 500)
// Reader sees inconsistent state!
```

2.3 Running the Demonstration

Experience the chaos firsthand:

```
cd unsynchronized
go run -race .
```

You will see:

- Race detector warnings showing exact conflict locations
- Lost updates in counter scenario
- Money disappearing in bank transfers
- Inconsistent reads detected

3 Your Task: Implementing Synchronization

Your mission is to fix these race conditions by implementing proper synchronization. You must implement **at least two** of the following approaches.

3.1 Approach 1: Coarse-Grained Mutex (Required)

Concept: Use a single `sync.Mutex` to protect the entire database.

Listing 4: Mutex Synchronization Pattern

```
type DatabaseMutex struct {
    mu      sync.Mutex
    records map[string]*Record
}

func (db *DatabaseMutex) Update(key string, delta int) {
    db.mu.Lock()
    defer db.mu.Unlock()

    // Critical section - only one goroutine at a time
    db.records[key].Value += delta
}
```

Characteristics:

- **Pros**: Simple, easy to verify correctness, no deadlocks
- **Cons**: Low concurrency, serializes all operations
- **Best for**: Write-heavy workloads, small critical sections

3.2 Approach 2: Monitor Pattern with Condition Variables

Concept: Implement fine-grained per-key locking using `sync.Cond`.

Key Concept

A **monitor** is a synchronization construct that combines mutual exclusion with condition variables, allowing threads to wait for specific conditions to become true.

Listing 5: Monitor Pattern

```go
type DatabaseMonitor struct {
    mu     sync.Mutex
    cond   *sync.Cond
    locked map[string]bool  // Per-key locks
}

func (db *DatabaseMonitor) acquireKey(key string) {
    for db.locked[key] {
        db.cond.Wait()  // Wait until key is free
    }
    db.locked[key] = true
}

func (db *DatabaseMonitor) releaseKey(key string) {
    db.locked[key] = false
    db.cond.Broadcast()  // Wake waiting goroutines
}
```

Characteristics:

- **Pros**: Better concurrency, operations on different keys proceed in parallel
- **Cons**: More complex, potential for deadlocks with multiple keys
- **Best for**: Demonstrating classic OS synchronization patterns

3.3 Approach 3: Reader-Writer Locks

Concept: Allow multiple concurrent readers but exclusive writers using `sync.RWMutex`.

Listing 6: Reader-Writer Lock Pattern

```go
type DatabaseRWLock struct {
    mu sync.RWMutex
    records map[string]*Record
}

func (db *DatabaseRWLock) Read(key string) int {
    db.mu.RLock()          // Shared lock
    defer db.mu.RUnlock()
    return db.records[key].Value
}

func (db *DatabaseRWLock) Write(key string, value int) {
    db.mu.Lock()           // Exclusive lock
    defer db.mu.Unlock()
    db.records[key].Value = value
}
```

Performance Comparison:
Characteristics:

- **Pros**: Excellent for read-heavy workloads (10x+ speedup)
- **Cons**: Writers can starve, more overhead than simple mutex
- **Best for**: Read-heavy workloads (90%+ reads)

3.4 Approach 4: Channel-Based Synchronization (Bonus)

Concept: Use Go channels to implement message-passing concurrency.

Key Concept

Go's philosophy: *"Don't communicate by sharing memory; share memory by communicating."* Channels serialize access by design.

Listing 7: Channel-Based Synchronization

```go
type DatabaseChannel struct {
    opChan chan operation
}

func (db *DatabaseChannel) run() {
    for op := range db.opChan {
        // Single goroutine processes all operations
        switch op.opType {
        case "read":
            op.resultChan <- db.records[op.key]
        case "write":
            db.records[op.key] = op.value
        }
    }
}
```

Characteristics:

- **Pros**: Idiomatic Go, naturally thread-safe, no explicit locks
- **Cons**: All operations serialized, channel overhead
- **Best for**: Learning message-passing concurrency models

4 Requirements

4.1 Implementation Requirements

1. **Correctness**: All provided tests must pass
2. **No Race Conditions**: Must pass `go test -race` with zero warnings
3. **API Compatibility**: Maintain the same interface as unsynchronized version
4. **Documentation**: Explain your synchronization strategy in comments
5. **Error Handling**: Proper use of `defer` to prevent lock leaks

4.2 Testing Requirements

Your implementation will be tested against:

1. **Functional Tests**: Basic CRUD operations
2. **Concurrency Tests**: 20+ concurrent goroutines
3. **Race Detection**: `go test -race -v`
4. **Stress Tests**: High-contention scenarios
5. **Consistency Tests**: Invariant preservation (e.g., bank transfer totals)

4.3 Analysis Requirements

Write a technical report (2-3 pages) including:

1. **Race Condition Analysis** (1 page):
   - Identify 3+ specific race conditions in unsynchronized code
   - Explain the memory ordering that causes each race
   - Describe potential consequences (data corruption, crashes, etc.)
2. **Synchronization Strategy** (1 page):
   - Explain your chosen approach(es) and why
   - Discuss critical sections and lock granularity
   - Justify design decisions (e.g., why RWLock over Mutex)
3. **Performance Analysis** (1 page):
   - Run benchmarks: `go test -bench=. -benchmem`
   - Compare throughput of different approaches
   - Analyze scalability with varying goroutine counts
   - Include graphs showing performance vs. concurrency level

5 Getting Started

5.1 Project Structure

```
mini-project/
|-- unsynchronized/          # Provided starter code
|    |-- database.go         # UNSAFE implementation
|    |-- client.go           # Test scenarios
|    |-- main.go             # Demo runner
|    +-- go.mod
+-- solution/                # Your code goes here
     |-- database_mutex.go   # Your implementation
     |-- database_*.go       # Additional approaches
     |-- *_test.go           # Tests (provided)
     +-- go.mod
```

5.2 Development Workflow

1. **Observe**: Run unsynchronized version with race detector
2. **Understand**: Analyze where and why races occur
3. **Implement**: Start with mutex approach (simplest)
4. **Test**: Run `go test -race -v` frequently
5. **Benchmark**: Compare performance of approaches
6. **Iterate**: Implement second approach
7. **Document**: Write analysis report

6 Grading Rubric

Component	**Points**	**Criteria**
Correctness (40 points)		
Tests Pass	20	All provided tests pass
No Races	20	Zero race conditions with `-race` flag
Implementation (30 points)		
Mutex	15	Correct coarse-grained locking
Second Approach	15	Monitor, RWLock, or Channel
Code Quality (10 points)		
Documentation	5	Clear comments explaining strategy
Best Practices	5	Proper `defer`, error handling
Analysis Report (20 points)		
Race Analysis	8	Identifies and explains races
Strategy	6	Justifies synchronization choices
Performance	6	Benchmarks and analysis
Bonus (up to +20 points)		
Extra Approaches	+10 each	Third/fourth implementation
Total	**100**	

7 Advanced Challenges (Optional)

For students seeking deeper understanding:

1. **Lock-Free Data Structures**: Implement using `sync/atomic`
2. **Deadlock Detection**: Add timeout-based deadlock detection
3. **Transaction Isolation**: Implement SERIALIZABLE isolation level
4. **Performance Optimization**: Fine-tune lock granularity
5. **Distributed Synchronization**: Extend to multiple processes

8 Submission Guidelines

8.1 Deliverables

Submit a ZIP file named `group#_database_sync.zip` containing:

1. Source code (all `.go` files)
2. Analysis report (PDF, 2-3 pages)
3. Benchmark results (text file or screenshots)
4. README with build/test instructions
5. **Group contribution statement** (who did what)

8.2 Evaluation Criteria

Your submission will be evaluated on:

- **Correctness**: Does it work without races?
- **Understanding**: Do you understand *why* it works?
- **Engineering**: Is the code clean and well-documented?
- **Analysis**: Can you explain trade-offs?

9 Resources

- Go Concurrency: `https://go.dev/doc/effective_go#concurrency`
- Go Race Detector: `https://go.dev/doc/articles/race_detector`
- Sync Package: `https://pkg.go.dev/sync`
- *The Little Book of Semaphores* by Allen Downey
- *Operating Systems: Three Easy Pieces* - Concurrency chapters

10 Collaboration Guidelines

This is a **group project for teams of 2 students**. Both team members are expected to contribute meaningfully to the implementation, testing, and analysis.

10.1 Team Expectations

- **Equal Contribution**: Both members should contribute to coding and analysis
- **Code Reviews**: Review each other's code before submission
- **Pair Programming**: Consider pair programming for complex sections
- **Communication**: Use version control (Git) to coordinate work

10.2 Contribution Statement

Include a brief statement (1 paragraph) in your README describing:

- How you divided the work
- What each team member contributed
- How you collaborated (meetings, pair programming, code reviews, etc.)

10.3 Academic Honesty

While this is a group project, collaboration between different groups is not permitted. Your team's code and analysis must be your own work. You may discuss high-level concepts with other groups, but sharing code or detailed implementation strategies is not allowed.

This project will challenge you, but it's one of the most important concepts in systems programming.

Understanding concurrency is what separates good programmers from great systems engineers.

Good luck!