CS311 Operating System Fundamentals

Assignment 2 - Thread Synchronization and Data Races

Note: While you may develop and run your code in any environment you prefer, you must transfer your code to the CS department remote server and ensure it also runs there. All grading will be performed on the remote server environment.

Learning Objectives

- Observe how race conditions occur in shared data structures.
- Use mutex locks to ensure correctness.
- Compare the performance of different synchronization strategies.
- Understand the trade-off between correctness and concurrency.

Overview

You are given a simplified multithreaded user database simulator. Each thread inserts and reads user records from a shared in-memory table.

```
typedef struct {
    int id;
    char name[MAX_NAME];
    char email[MAX_EMAIL];
    volatile int valid;
} User;
```

There are 10,000 total users. The table is shared among all threads, meaning concurrent writes and reads can interfere. Your job is to explore how locking affects correctness and performance.

Provided Files

- user_data.h - Structure and helper declarations (do not modify)
- user_data.c - Utility functions (do not modify)
- main.c - Main program logic (edit this file)
- Makefile - Automates compilation (optional edits allowed)

Compile using:

```
make
```

Run using:

```
./assignment <nthreads>
```

Example:

```
./assignment 4
```

Part 1 - Observe Race Conditions (No Locks)

Run with 1 and 4 threads to observe missing users caused by race conditions:

```
./assignment 1
./assignment 4
```

Expected observations:

- With 1 thread - all users present (0 missing).

- With 4 threads - some users missing, results vary each run.

Part 2 - Add a Global Lock

Add a global mutex to protect the shared table. Declare a lock at the top of main.c:

```
static pthread_mutex_t global_lock = PTHREAD_MUTEX_INITIALIZER;
```

Wrap table access with lock and unlock calls:

```
pthread_mutex_lock(&global_lock);
table[id] = u;
pthread_mutex_unlock(&global_lock);
```

Rebuild and run again with multiple threads. You should now see 0 missing users, but slower performance.

Part 3 - Implement Per-Segment Locks

Replace the global lock with an array of locks, one per 100-user segment. Each segment can be updated independently.

Each segment contains 100 users, arranged in order by their ID. For example, IDs 0–99 belong to segment 0, IDs 100–199 belong to segment 1, and ID 2599 would be in segment 25.

```
static pthread_mutex_t seg_locks[NUM_SEGMENTS];
```

Initialize all locks in main():

```
for (int i = 0; i < NUM_SEGMENTS; i++) pthread_mutex_init(&seg_locks[i], NULL);
```

Then, lock only the segment for each ID:

```
int seg = seg_index_for_id(id);
pthread_mutex_lock(&seg_locks[seg]);
table[id] = u;
pthread_mutex_unlock(&seg_locks[seg]);
```

Reflection Questions

- What causes data races in this program?
- Why does adding locks fix correctness but reduce performance?
- Why is per-segment locking more efficient than a single global lock?
- If you used more than 100 segments, how might performance change?
- How could these principles apply to real operating systems or databases?
- If the design used multiple processes instead of threads, what changes would be needed to maintain data consistency and performance?

Submission

Submit the following: - Your modified main.c. - A written document that includes clear answers to all reflection questions (you may answer them inline or in a separate section), along with a short summary discussing your observations, performance trade-offs, and key insights. Bundle your files as a .zip or .tar.gz archive and submit it through **Brightspace**.

Grading Rubric (100 points)

- **Program Functionality (10 pts):** Base program compiles and runs correctly without errors.
- **Global Lock Implementation (20 pts):** Global mutex correctly added and verified; results are correct under multiple threads.

- **Per-Segment Lock Implementation (30 pts):** Per-segment lock array correctly implemented, initialized, and tested; results remain correct and performance improves.
- **Performance Analysis (15 pts):** Includes timing results and discussion of performance trade-offs across locking methods.
- **Reflection Responses (15 pts):** Written answers fully address all reflection questions, demonstrating understanding of synchronization and design trade-offs.
- **Summary and Presentation (10 pts):** Clear organization, grammar, and readability in the write-up. Submitted properly through Brightspace as a single archive.