Lab 4 — Concurrency / Synchronization

Student: ■■■■■ ■■■■ Group: ■-26

This report contains: design, code, test data and results for variant #16 (m=2).
Implementation uses std::shared_mutex per field to allow concurrent reads and exclusive writes.

Files included: data_struct_demo.cpp, gen_commands.py, data/*.txt, results/table3x3.txt

Design and Locking Strategy

1) One std::shared_mutex per field (2 fields total).
2) Read operations acquire shared_lock on a single field.
3) Write operations acquire unique_lock on a single field.
4) operator string acquires shared_lock on all fields in increasing order (0,1) to avoid deadlock.

Rationale: high proportion of read/string operations benefits from shared locks allowing concurrent readers; writes are rare and exclusive locks are acceptable.

How to compile and run (short)

1) Compile: g++ -std=c++17 -O2 data_struct_demo.cpp -lpthread -o demo
2) Generate test files (optional): python3 gen_commands.py variant 2000 data/variant
3) Run examples: ./demo multi 3 data/variant_thread0.txt data/variant_thread1.txt data/variant_thread2.txt

Sample results table (fill after experiments):

	variant	uniform	skewed
1 thread	a11	a12	a13
2 threads	a21	a22	a23
3 threads	a31	a32	a33

C++ implementation (start)

```
// data_struct_demo.cpp
// C++17 required (for std::shared_mutex)
// Compile: g++ -std=c++17 -O2 data_struct_demo.cpp -lpthread -o demo
// Example usage:
//  ./demo single 1 data/variant_thread0.txt
//  ./demo multi 3 data/variant_thread0.txt data/variant_thread1.txt data/variant_thread2.txt

#include <bits/stdc++.h>
#include <shared_mutex>
#include <thread>
#include <chrono>

using namespace std;
using Clock = chrono::high_resolution_clock;

struct DataStruct {
    vector<int> fields; // m = 2
    vector<shared_mutex> locks; // one shared_mutex per field

    DataStruct(size_t m = 2) : fields(m, 0), locks(m) {}

    // get value (read) — shared lock on single field
    int get(size_t idx) {
        shared_lock<shared_mutex> lk(locks[idx]); // shared/read lock
        // small pause to simulate some work (comment out for real benchmarks)
        // this_thread::sleep_for(chrono::microseconds(1));
        return fields[idx];
    }

    // set value (write) — exclusive lock on single field
    void set(size_t idx, int val) {
        unique_lock<shared_mutex> lk(locks[idx]);
        fields[idx] = val;
    }

    // operator string — read all fields (acquire shared locks in order to avoid deadlock)
    string to_string_all() {
        // acquire shared_lock on all in increasing order
        shared_lock<shared_mutex> lk0(locks[0]);
        shared_lock<shared_mutex> lk1(locks[1]);
        ostringstream oss;
        oss << "Field0=" << fields[0] << "; Field1=" << fields[1];
        return oss.str();
    }
};

// Command representation
struct Cmd {
    enum Type { READ, WRITE, STRING } type;
    int idx; // for read/write
    int val; // for write
    Cmd(Type t=STRING, int i=0, int v=0) : type(t), idx(i), val(v) {}
};

vector<Cmd> parse_file_to_cmds(const string &filename) {
    ifstream in(filename);
    if (!in) {
        cerr << "Cannot open file: " << filename << "\n";
        return {};
    }
    vector<Cmd> res;
    string op;
    while (in >> op) {
        if (op == "read") {
            int idx; in >> idx;
            res.emplace_back(Cmd::READ, idx, 0);
        } else if (op == "write") {
            int idx, val; in >> idx >> val;
            res.emplace_back(Cmd::WRITE, idx, val);
        } else if (op == "string") {
            res.emplace_back(Cmd::STRING, 0, 0);
        } else {
            // skip unknown line
            string rest; getline(in, rest);
        }
    }
    return res;
}

// Execute commands (the data struct is shared among threads).
void execute_commands(DataStruct &ds, const vector<Cmd> &cmds) {
    for (const auto &c : cmds) {
        if (c.type == Cmd::READ) {
            volatile int v = ds.get(c.idx); (void)v;
```

```
        } else if (c.type == Cmd::WRITE) {
            ds.set(c.idx, c.val);
        } else { // STRING
            string s = ds.to_string_all();
            (void)s;
        }
    }
}

int main(int argc, char** argv) {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    if (argc < 3) {
        cout << "Usage:\n  " << argv[0] << " mode threads file1 [file2 file3]\n";
        cout << "mode: single | multi\n";
        cout << "threads: number of threads to spawn (1,2,3). For single mode, use 1\n";
        return 0;
    }

    string mode = argv[1];
    int threads = stoi(argv[2]);
    vector<string> files;
    for (int i = 0; i < threads; ++i) {
        if (3 + i < argc) files.push_back(argv[3 + i]);
        else files.push_back(""); // missing
    }

    // Read files into memory (we do NOT count this time)
    vector<vector<Cmd>> all_cmds;
    for (int i = 0; i < threads; ++i) {
        if (files[i].empty()) {
            cerr << "Missing filename for thread " << i << "\n";
            return 1;
        }
        auto cmds = parse_file_to_cmds(files[i]);
```

Notes and recommendations

- Repeat each experiment 5 times and use averages.
- Use -O2 optimization and proper compiler flags.
- Ensure your system has enough CPU cores for multi-threading tests.
- If you need help running experiments, contact instructor or check README.