Advanced Parallel Programming Exercise 6

Marvin Kaster

TECHNISCHE UNIVERSITÄT DARMSTADT

Summer term 2025
03.07.2025

Please solve the following tasks by 03.07.2025. The results are not graded, but a solution is discussed on 03.07.2025.

Task 1: Dining philosophers problem

Five philosophers are seated around a circular table. Between each pair of philosophers lies a single fork, meaning there are five forks in total.

Each philosopher can be in one of two states:

- Thinking: The philosopher does not need any forks.
- Eating: The philosopher requires two forks—the one to their left and the one to their right.

A fork can either be on the table (available) or held by a philosopher (in use). To eat, a philosopher must:

1. Pick up the left fork (if available).
2. Then, pick up the right fork (if available).
3. Once both forks are held, the philosopher eats for a while.
4. After eating, the philosopher puts down the right fork, then the left fork, and returns to thinking.

What is the problem of this protocol? How can you change it to avoid it? Implement your strategy in C++.

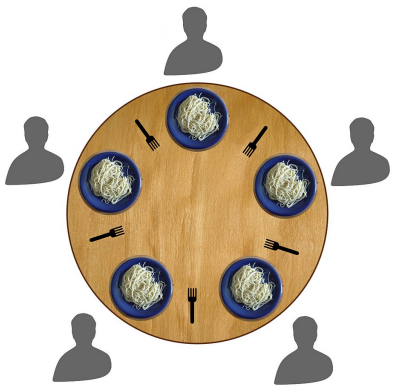

Figure 1: Dining philosophors problem. Taken from https://commons.wikimedia.org/wiki/File:Dining_philosophers_diagram.jpg bdesham

APP – Exercise 6

Solution:

When every philosopher picks up the left fork, every fork is picked up from the table. All philosophers will wait until their right fork is available, which will never happen as everyone is still waiting for a fork. This is called a deadlock. We must ensure that we always pick up the forks (lock the mutexes) in the same order to avoid a deadlock. One strategy would be to assign an index to each fork and always pick up the fork with a lower index first. Another strategy is to assign an index to the philosophers and distinguish between philosophers with an even and an odd index. The philosophers with an even number pick up the left fork first and then the right fork. The odd philosophers pick up the right fork first and then the left one. You can find an implementation of this strategy below.

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>

constexpr int num_philosophers = 5;

std::mutex forks[num_philosophers];

void philosopher(int id) {
    auto left = id;
    auto right = (id + 1) % num_philosophers;

    while (true) {
        std::cout << "Philosopher " << id << " is thinking.\n";
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));

        // Pick up forks
        if (id % 2 == 0) {
            // Pick left then right
            forks[left].lock();
            std::cout << "Philosopher " << id << " picked up left fork " << left << ".\n";

            forks[right].lock();
            std::cout << "Philosopher " << id << " picked up right fork " << right << ".\n";
        } else {
            // Pick right then left
            forks[right].lock();
            std::cout << "Philosopher " << id << " picked up right fork " << right << ".\n";

            forks[left].lock();
            std::cout << "Philosopher " << id << " picked up left fork " << left << ".\n";
        }

        // Eating
        std::cout << "Philosopher " << id << " is eating.\n";
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));

        // Put down forks
        forks[left].unlock();
        forks[right].unlock();

        // Thinking
        std::cout << "Philosopher " << id << " put down forks and is now thinking.\n";
    }
}

int main() {
    std::vector<std::thread> philosophers;
    for (auto i = 0U; i < num_philosophers; i++) {
        philosophers.emplace_back(philosopher, i);
    }

    for (auto &p: philosophers) {
        p.join();
    }
    return 0;
```

```
}
```

Listing 1: Dining philosophers problem

Task 2: Read-write mutex

In this task we want to implement a simplified read-write mutex. It should allow multiple threads to read concurrently, but grant exclusive access for write operations. Thus, it provides different functions to acquire a non-exclusive lock for reading and to acquire an exclusive lock for writing. For unlocking the class may provide two different unlock functions, too.

Please test your implementation with a small program.

Solution:

```cpp
#include <mutex>
#include <thread>
#include <iostream>
#include <vector>

class ReadWriteLock {

public:
    void reader_lock() {
        std::lock_guard<std::mutex> guard(lock);
        reader_count++;
    }

    void writer_lock() {
        while (true) {
            lock.lock();
            if (reader_count == 0) {
                return;
            }
            lock.unlock();
        }
    }

    void reader_unlock() {
        std::lock_guard<std::mutex> guard(lock);
        reader_count--;
    }

    void writer_unlock() {
        lock.unlock();
    }

private:
    std::mutex lock;
    int reader_count = 0;
};

std::uint64_t summand = 1UL;
ReadWriteLock lock;

void sum() {
    lock.reader_lock();
    std::uint64_t result = 0;
    for (int i = 0; i < 100; i++) {
        result += summand;
    }
    lock.reader_unlock();
    std::cout << "result: " << result << std::endl;
    lock.writer_lock();
```

```
    summand++;
    lock.writer_unlock();
}

void thread_func() {
    for (int i = 0; i < 1000; i++) {
        sum();
    }
}

int main() {
    const int num_threads = 8;
    std::vector<std::thread> threads(num_threads);

    for (int i = 0; i < num_threads; i++) {
        threads[i] = std::thread(thread_func);
    }
    for (int i = 0; i < num_threads; i++) {
        threads[i].join();
    }
}
```

Listing 2: Read-write mutex