Assignment 4 Report - Use of Monitors

Dean Tsankov, Kacper Lisik

February 24, 2025

Introduction

In this assignment, we learn how to program and use concurrent objects, e.g., synchronized monitors, in a multi-threaded application, as well as to program and use monitors in a concurrent program in Java and C++.

Problem 1 - A Fuel Space Station

Task description

```
Consider a future fuel station in space that supplies nitrogen and
↪ quantum fluid. Assume that the station can handle V space vehicles in
↪ parallel (simultaneously) and maximum storage for N liters of
↪ nitrogen and Q liters of quantum fluid. When a vehicle arrives at the
↪ station, it requests the needed amounts of fuel of the two different
↪ types or only one of the types. If there is not enough fuel of either
↪ type, the vehicle has to wait without blocking other vehicles.
↪ Special supply vehicles deliver fuel in fixed quantities that far
↪ exceed the fuel tank capacity of any vehicle. When a supply vehicle
↪ arrives at the station, it delays until there is enough space to
↪ deposit the fuel delivered. To travel back, the supply vehicles also
↪ request a certain amount of fuel of the two different types or one of
↪ the types, just like an ordinary vehicle, not necessarily the type it
↪ supplies.
```

Key points of solution

We decided to use the Java and its `concurrent` library to solve this task. As such the solution consisted of simply applying the standard object oriented programming principles to the problem and making one of the objects behave as a monitor namely the Station object. When other objects access it and its functions they will have to comply with its concurrency management

mechanisms.

The structure of the solution is the following: There is the one Station monitor object which implements the `FuelUp()` method for when a vehicle wants to get fuel and the `RechargeFuelTanksAndFuelUp()` method which replenishes the stations own supply of fuel as well as give fuel to the supplier vehicle. Both of them include the key word `synchronized` as they are used as implicit locks and only a certain number of threads can execute their functionality simultaneously.

Due to the length of the code here we will present only the more involved of the two functions but they behave fairly similarly in any case:

```
(...)
public synchronized void RechargeFuelTanksAndFuelUp(int
↪  nitroSupplyAmount, int quantSupplyAmount,
        int nitroRequestAmount, int quantRequestAmount) {

    while (this.docked >= this.maxVehicles) {
        try {
            wait();
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }

    // succesfully docked
    docked++;
    System.out.println(Thread.currentThread().getName() + " docked
    ↪  ");
    long beginTime = System.currentTimeMillis();
    while ((nitroSupplyAmount > 0 && this.nitroAmount +
    ↪  nitroSupplyAmount > this.maxNitroStorage)
            || (quantSupplyAmount > 0 && this.quantAmount +
            ↪  quantSupplyAmount > this.maxQuantStorage)) {
        try {
            wait(this.generalTimeout);
            // This timeout measure is simply here for testing
            ↪  purposes since, the fuel and
            // request amonts of each vehicle are random it often
            ↪  happens that they do not
            // coincide and reach a stall. in a proper program these
            ↪  amounts would somehow
            // be properly balanced
            if (System.currentTimeMillis() - beginTime >=
            ↪  this.generalTimeout) {
```

```
                    System.out.println(
                            Thread.currentThread().getName() + " - waited
                            ↪  too long whithout being able to
                            ↪  recharge");
                    docked--;
                    notifyAll();
                    return;
                }
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }

        try {
            Thread.sleep(rnd.nextInt(900)); // stall for believable
            ↪  recharging
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }

        this.nitroAmount += nitroSupplyAmount;
        this.quantAmount += quantSupplyAmount;

        System.out.println("Recharge Station: The supply vehicle: " +
        ↪  Thread.currentThread().getName() + " supplied "
                + nitroSupplyAmount
                + " units nitrogen;" + quantSupplyAmount + " units
                ↪  quantum fuel to the station");
        System.out.println("N: " + this.nitroAmount + " Q: " +
        ↪  this.quantAmount + "\n");

        // fuel up the supplier vehicle
        this.CoreFuleUpFuncrionality(nitroRequestAmount,
        ↪  quantRequestAmount);

        System.out.println(Thread.currentThread().getName() + " flew
        ↪  away");
        docked--;
        notifyAll();
    }
    (...)
```

Except the station object there are two types of vehicls with their respective classes but since part of their functionality inherently overlaps, we

decided to first create an abstract vehicle class which the specific ones extend to behave in their own ways:

```
(...)
public abstract class Vehicle extends Thread {
    protected int nitroRequestAmount;
    protected int quantRequestAmount;
    protected Station fuelStation;

    protected int lifeSpan; //how many request cycles can the
    ↪  vehicles do

    protected Random rnd;

    public Vehicle(int nitroRequestAmount, int quantRequestAmount,
    ↪  Station fuelStation, int lifeSpan) {
        this.nitroRequestAmount = nitroRequestAmount;
        this.quantRequestAmount = quantRequestAmount;
        this.fuelStation = fuelStation;
        this.rnd = new Random();
        this.lifeSpan = lifeSpan;
    }

    /**
     * @param delay the amount of time the vehicle spends away from
     ↪  the station
     * might be posible to rework this function to also work for the
     ↪  supplier somehow by passing a line of code as an argument?
    */
    public void StandardFuelUpProcedure(int delay) {
        while(this.lifeSpan > 0){
            this.fuelStation.FuelUp(this.nitroRequestAmount,
            ↪  this.quantRequestAmount);
            this.lifeSpan--;
            try {
                Thread.sleep(rnd.nextInt(delay)); //do stuff before
                ↪  next request
            }
            catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

```
    (...)
```

The final part of the solution is the main class itself which we will not show here, but it simply reads console arguments in order to set up a test scenario and creates all the necessary objects for the simulation. Then it invokes each vehicle's `start()` method, since all of them are children of the `Thread` class, this is what causes the threads to begin their routines. Ultimately we wait for all of them to finish by calling `join()` on them and that concludes our program which simply prints out the actions each thread takes and the status of the station at various points in time.

Problem

Task description

```
A vehicle repair station can repair vehicles of three types: A, B, and C.
↪  The station has the following capacity:

It can repair in parallel at most a, b, and c vehicles of type A, B, and
↪  C, correspondingly;
It can repair in parallel at most v vehicles of different types.
If a vehicle cannot be repaired because of any (or both) of the above
↪  limitations, the vehicle has to wait until it can get a place to be
↪  repaired.
```

Key points of solution

Thread Synchronization

The program uses `std::mutex` to ensure only one thread modifies shared data at a time. A `std::condition_variable` is used to make vehicles wait if the station is full.

Vehicle Entry and Exit

Each vehicle tries to enter the repair station. If there is enough space, it enters; otherwise, it waits. When a vehicle exits, space is freed, and waiting vehicles are notified.

Use of OpenMP

OpenMP is used to create multiple threads, each representing a vehicle. Each thread randomly decides when to request access to the repair station.

Output and Debugging

The program prints messages showing when vehicles enter and exit the station. It also displays the thread number and remaining space, making it easier to track execution.

```
    (...)
class RepairStation
{
private:
    condition_variable cv;
    mutex mtx;
    int repair_type[3];
    int all;

    void enter_station(char type)
    {
        type -= 'a';
        unique_lock<mutex> lock(mtx);
        cv.wait(lock, [this, type]
                { return (repair_type[type] > 0 && all > 0); });
        repair_type[type]--;
        all--;
        cout << "vechicle " << char('a' + type) << " " << " entered the
        ↪  station. Thread: " << omp_get_thread_num() << " free space
        ↪  overall after entering: " << all << " vechicle specific free
        ↪  space: " << repair_type[type] << "\n";
    }
    void exit_station(char type)
    {
        type -= 'a';
        unique_lock<mutex> lock(mtx);
        repair_type[type]++;
        all++;
        cout << "vechicle " << char('a' + type) << " " << " exited the
        ↪  station. Thread: " << omp_get_thread_num() << " free space
        ↪  overall: " << all << " vechicle specific free space: " <<
        ↪  repair_type[type] << "\n";
        cv.notify_all();
    }

public:
    RepairStation(int _a, int _b, int _c, int _all)
    {
        repair_type[0] = _a;
        repair_type[1] = _b;
```

```
        repair_type[2] = _c;
        all = _all;
    }
    ~RepairStation() {}

    void use_station(char type)
    {
        enter_station(type);
        cout << "Bzzt... Bzzzzt Thread: " << omp_get_thread_num() <<
        ↪  "\n";
        usleep(rand() % 3000 + 1);
        cout << "Thread: " << omp_get_thread_num() << " Repaired\n";
        exit_station(type);
    }
};
    (...)
```