Assignment 5: Concurrency 5/28/2025

**Pedro Ramirez**

Implement the next release of your term project. You will incorporate *concurrency* if concurrency can fit appropriately (it usually can!); otherwise, create a different project with concurrency.

The same instructions as for Assignment 3 apply, including leveraging AI, showing your value added in red font and by means of explanations, inserting comments (in red) in figures that describe clearly your value added, commenting on what you consider significant prompt contributions, and submitting.

## 1 SUMMARY DESCRIPTION

One- or two-paragraph overall description of your proposed term project. Color red the parts that changed from Assignment 4.Excellent assignments are usually event-driven.

Your response replaces this.

## 2 ADDITIONAL FUNCTIONAL REQUIREMENTS IMPLEMENTED IN THIS RELEASE

Title and one or two sentences per requirement. Don’t repeat requirements implemented for prior assignments unless they are necessary to provide context—in which case, make it clear they are old.

### 🟥 **2.1 Concurrent Appointment Booking (NEW)**

The system allows multiple patients to attempt booking the same appointment slot at the same time. To prevent double bookings, I use a ReentrantLock in the AppointmentManager, which ensures that only one booking goes through if there’s a conflict. This adds realism to the system and prepares it for real-world scenarios where multiple users might act at once.

### 🟥 **2.2 Asynchronous Appointment Saving (NEW)**

After an appointment is booked, it is now saved in the background by a separate thread. Instead of writing directly to the file during booking, the appointment is added to a queue. A background worker listens to the queue and writes to appointments.txt, allowing the booking process to stay fast and responsive even under load.

## 3 I/O SUPPORTING THE NEW REQUIREMENTS LISTED ABOVE

Provide input/output illustrating all new features of your application. Clearly demonstrate every new requirement and tie it back to the corresponding requirement.

Your response replaces this.

## 4 YOUR DIRECTORY

Show a screenshot of your directory. This should include JUnit tests—class-by-class, method-by-method, and covering edge cases, except for trivial and inappropriate ones. Include your “.dat” files (where objects are written) or JSON files. This should include JUnit tests—except for trivial and inappropriate ones.

Your response replaces this.

## 5 TECHNIQUES IMPLEMENTED

Integrate concurrency to tangibly improve the application’s scalability or real-world usability. Because you have AI at your disposal, we hold you to a standard that assumes its use: we reward well-engineered solutions rather than merely deducting points for errors.

## ✅ 5.1 Class Model and Sequence Diagram

### Technique: Asynchronous Appointment Saving

This technique offloads the task of saving booked appointments to a background thread, allowing the system to stay responsive. The appointment is passed to a queue, and a separate thread writes it to the file.

### Use Case: Save Appointment in the Background

**Actors:**  
• Patient  
• System (MEDTRACK)

**Preconditions:**  
• A patient has successfully booked an appointment.  
• AppointmentManager and AppointmentSaverWorker are initialized.

**Main Flow:**

1. The patient books an appointment using the CLI.
2. AppointmentManager.bookAppointment() creates the appointment object.
3. The appointment is passed to AppointmentSaverWorker.saveLater().
4. The background thread reads from the queue using BlockingQueue.take().
5. The appointment is saved to appointments.txt using FileWriter.
6. The main thread continues immediately without waiting.

**Postconditions:**  
• The appointment is written to the file in the background.  
• The worker thread stays active for future bookings.

### Class Model (🟥 = where concurrency is applied)

└── AppointmentManager 🟥

private final AppointmentSaverWorker saverWorker

private final ExecutorService executor

AppointmentManager()

→ executor.submit(saverWorker)

bookAppointment()

→ saverWorker.saveLater(...)

shutdown()

→ saverWorker.stop()

└── AppointmentSaverWorker 🟥 NEW

saveLater(Appointment, String, String)

→ adds to BlockingQueue

run()

→ reads from queue, writes to file (on separate thread)

stop()

→ stops the background loop

record AppointmentSaveRequest(Appointment, String, String)

📌 [Diagram is referenced here](https://docs.google.com/spreadsheets/d/1Srxutu-G_5bswR1xzbCi4zFVPmfwhTKby1di8F7EdTw/edit?gid=0#gid=0)

A screenshot of a computer

AI-generated content may be incorrect.

## ✅ 5.2 Explanation: Why Concurrency Is Appropriate

Concurrency is appropriate here because saving appointments to a file takes time and would slow down the booking process if done on the main thread. Using a separate thread for saving keeps the system fast and responsive, especially when there are many bookings happening at once. It also better simulates how a real-world scheduling system would work in a clinic or hospital.

### ✅ 5.3 Code Showing Where Concurrency Is Defined

This section defines the background thread, executor, lock, and blocking queue.

// 🟥 **Inside AppointmentManager.java**

// Lock for protecting shared booking logic

private final ReentrantLock lock = new ReentrantLock();

// Worker that handles saving appointments in the background

private final AppointmentSaverWorker saverWorker = new AppointmentSaverWorker();

// Executor to run the worker on a separate thread

private final ExecutorService executor = Executors.newSingleThreadExecutor();

// Constructor starts the background thread

public AppointmentManager() {

executor.submit(saverWorker); // 🟥 Start background worker thread

}

// Shutdown method stops background saving cleanly

public void shutdown() {

saverWorker.stop(); // 🟥 Signal the saver to stop

executor.shutdown(); // 🟥 Stop the executor service

}

// 🟥 **Inside AppointmentSaverWorker.java**

// Thread-safe queue to hold save requests

private final BlockingQueue<AppointmentSaveRequest> queue = new LinkedBlockingQueue<>();

// Flag to keep the thread running

private volatile boolean running = true;

// Called by AppointmentManager to queue up saves

public void saveLater(Appointment appointment, String patientName, String doctorName) {

queue.offer(new AppointmentSaveRequest(appointment, patientName, doctorName));

}

// Stops the worker loop

public void stop() {

running = false;

}

// Background thread execution

@Override

public void run() {

System.out.println("🟢 Appointment saver started...");

while (running || !queue.isEmpty()) {

try {

AppointmentSaveRequest request = queue.take(); // Waits for next appointment

Appointment a = request.appointment();

try (FileWriter writer = new FileWriter("data/appointments.txt", true)) {

writer.write(a.getConfirmationCode() + " | " +

request.patientName() + " | " +

request.doctorName() + " | " +

a.getDate() + " | " +

a.getTime() + "\n");

}

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

System.err.println("⚠️ Saver thread interrupted");

} catch (IOException e) {

System.err.println("❌ Error saving appointment: " + e.getMessage());

}

}

System.out.println("🛑 Appointment saver stopped.");

}

// Inner record used to wrap appointment data

public record AppointmentSaveRequest(Appointment appointment, String patientName, String doctorName) {

}

### ✅ 5.4 Code Showing Where Concurrency Is Used

This section shows how the lock and background saver are used at runtime during booking.

// 🟥 **Inside AppointmentManager.java**

public Appointment bookAppointment(Patient patient, Doctor doctor, String date, String time) {

lock.lock(); // 🟥 Begin critical section to prevent race conditions

try {

// Validate input (simplified here)

if (!doctor.isAvailable(date, time)) {

throw new InvalidInputException("Doctor is already booked at this time.", doctor.getName());

}

// Create the appointment

Appointment appointment = new Appointment(patient.getId(), doctor.getId(), date, time);

// Add appointment to both doctor and patient

doctor.addAppointment(appointment);

patient.addAppointment(appointment);

// 🟥 Use background worker to save without blocking

saverWorker.saveLater(appointment, patient.getName(), doctor.getName());

return appointment;

} finally {

lock.unlock(); // 🟥 End of critical section

}

}

## 6 Evaluation



### ✅ Part 2.1 – Concurrent Appointment Booking (NEW)

**Prompt:**  
Describe how to implement concurrency in the current project. How would an excellent solution look?

**Answer:**  
To implement concurrency in the booking process, use a ReentrantLock in the bookAppointment method inside AppointmentManager. This ensures that when multiple patients try to book the same doctor and time slot, only one booking is allowed through. Set up a demo where 10 patients attempt to book at the same time, and the system correctly allows only one to succeed.

**An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:**

* ⬜ Clearly demonstrates the race condition with additional test cases or scenarios
* ⬜ Extends support for concurrent booking with multiple doctors and multiple time slots
* ⬜ Provides clear user feedback when a booking fails due to unavailability
* ⬜ Uses modular, reusable locking logic that could be applied to other shared data or services
* ⬜ Adds logging or tracing of booking attempts for better debugging and visibility

### 🟥 Value Added – Making the booking process thread-safe prevents double bookings and supports multiple users at the same time. This makes the system more reliable and better suited for real-world use.

### ✅ Part 2.2 – Asynchronous Appointment Saving (NEW)

**Prompt:**  
Describe another new functional requirement that we can implement in this release, including context if needed.

**Answer:**  
To avoid blocking the booking process with file writing, introduce a background worker that saves appointments asynchronously. When an appointment is booked, it is added to a thread-safe queue. A separate worker thread processes the queue and writes appointments to the file. This keeps the user experience smooth and models how a real-world system might handle high activity.

**An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:**

* ⬜ Automatically shuts down the worker thread when the program exits without requiring a manual call
* ⬜ Confirms and logs each save to provide assurance that the appointment was written
* ⬜ Handles a high volume of queued appointments without loss or delay
* ⬜ Prevents duplicate or corrupted entries through additional checks
* ⬜ Adds retry logic and error handling if the file cannot be written
* ⬜ Ensures the worker thread shuts down automatically when the program exits, without requiring manual calls
* ⬜ Confirms and logs each successful appointment save, so the user knows it was written
* ⬜ Handles many appointments queued in quick succession without delay or data loss
* ⬜ Prevents duplicate entries or overlapping writes by handling edge cases or adding file locks
* ⬜ Includes error handling and retry logic if file writing fails (e.g., disk error, permission issues)

### 🟥 Value Added – Saving appointments in the background keeps the app responsive during booking. It avoids delays caused by file writing and improves the user experience during busy activity.