Patient Monitoring Health System

John Teetz & Alicia Peters

CMPSC 472

12/7/25

Introduction:

In this project, we set out to design and implement a real-time patient monitoring system that simulates how hospitals track changing vital signs and respond to clinical risks. The main objective was to create a web-based application that continuously updates patient vital signs, evaluates their condition using a severity algorithm, and presents the results in a clear dashboard interface. We also wanted the system to demonstrate essential operating system concepts, such as background threading, concurrency, scheduling, and controlled shared state.

Our Patient Monitoring System (called VitalWatch) displays three simulated patients, Alice Miller, John Smith, and Maria Garcia, whose vital signs update dynamically. A monitoring thread simulates new vitals, identifies when they cross unsafe thresholds, and automatically generates warning or critical alerts. The dashboard then allows staff to view patient conditions, acknowledge alerts, and change the simulation mode (Normal, Fast, or Paused) to control how frequently the background thread updates the data.

Significance:

We chose this project because real-time monitoring systems are essential in modern healthcare, and building one provides a meaningful way to apply OS principles to a realistic domain. Hospitals rely on continuous monitoring to detect deterioration early, reduce adverse events, and improve patient outcomes. By simulating this process, we can explore how operating systems manage background tasks, prioritize work, and maintain responsiveness while handling continuous data.

The project is also significant because it demonstrates how OS concepts translate into software that feels interactive and “alive.” Instead of building a static web app, we built a dynamic system where a background worker thread acts like a scheduler updating patient state independently of user requests. The simulation controls (Normal, Fast, Pause) make the system more than just a display; it becomes a tunable environment where users can change the underlying monitoring frequency, similar to tuning interval timers or adjusting process quanta in an OS.

Finally, this project is meaningful because it simulates clinical alert escalation, making it relevant to real-world hospital informatics. The alert system is intentionally simple, but it illustrates expert-system behavior that, in real hospitals, helps prevent missed deterioration.

Code Structure: (Can be found in the README)

The code for this project is organized into two main parts: the backend logic and the web interface. The backend lives in the "patient_monitor" folder and is responsible for things like storing patient data, simulating their vitals, generating alerts, and running the background monitoring thread. The "web" folder handles everything related to the user interface, including the routes, templates, and pages the user actually interacts with. Keeping these two pieces separate made the project easier to work on because I could update the simulation logic without breaking the UI, and vice versa.

Algorithms:

There are two main algorithms in this project: one for simulating vitals and one for deciding whether an alert should be created. For the vitals simulation, the idea is pretty simple. I

take the patient's most recent vitals and make small random changes to them so it looks like the patient is actually being monitored in real time. For example, heart rate might go up or down by a few beats, and the same goes for blood pressure, oxygen levels, and temperature. This creates realistic variation without letting the numbers jump into impossible ranges.

The second algorithm is the alert severity algorithm. After new vitals are generated, the system checks whether any of the values fall into unsafe ranges. If a vital crosses one of these thresholds, the system creates either a warning or critical alert. For instance, if oxygen drops below 90 percent or if heart rate goes unusually high, the algorithm labels that as critical because those conditions often mean the patient is unstable. If the numbers are only slightly concerning, it classifies them as warnings. This is a straightforward rules-based approach, but it works well for demonstrating how a monitoring system decides when something is wrong.

Another smaller algorithm controls the simulation speed. The monitoring thread checks a shared setting that determines whether it should run normally, run faster, or pause. Fast mode reduces the amount of time the system sleeps between updates, which makes vitals change more quickly. Pause mode basically tells the thread to skip updating completely. This part of the project ties back to operating system scheduling ideas, since it changes how often the background work happens.

Verification of Algorithms:

To make sure the algorithms were working correctly, we tested them with a couple of very simple toy examples.

For the vitals simulation, we took a set of normal values for a patient, such as a heart rate of 80 and an oxygen level of 96, and ran them through the simulation function. The output numbers were only slightly different, such as 84 for heart rate or 95 for oxygen. This showed that the function was doing what we intended: adding small, realistic changes instead of extreme or random jumps.

For the alert severity algorithm, we tested a set of vitals that should definitely trigger an alert. For example, a heart rate above 140, oxygen at 88, and a high temperature all at the same time. The algorithm correctly labeled that as a critical alert. This gave confidence that the thresholds were implemented the right way and that the monitoring thread would react when a patient's condition worsened.

Functionalities:

The system includes several main functionalities that work together to simulate a real patient monitoring environment. Each part also connects back to operating system ideas we learned in class, especially around threading, scheduling, and shared state.

- The system continuously updates the vitals of three patients in the background. This is handled by a separate thread that runs on its own schedule, which is similar to how the OS manages background processes.
- Whenever a patient’s vitals become abnormal, the system automatically generates alerts. This part of the project shows how periodic tasks can make decisions based on new data, similar to how the OS checks system status during interrupts or timed events.

- The dashboard displays all patients, their latest vitals, and any related alerts. The UI refreshes based on updated data that comes from the background thread, which demonstrates communication between the main program and a concurrently running worker.
- Users can filter alerts by severity, making it easier to focus on critical conditions. This is helpful when the system produces a lot of activity at once, especially in fast mode.
- The simulation controls allow the user to switch between normal speed, fast speed, or pause. Changing the speed basically adjusts how long the background thread sleeps between cycles, which reflects the idea of scheduling intervals or different priority levels in an OS.
- There is a patient detail page that shows recent vitals for each patient, giving a clearer view of how their condition changes over time.
- A login system was added so only authorized users can access the dashboard. This ties into the idea of protecting resources, which we also discussed in the context of OS security and user permissions.

Adding these small OS elements made the project more than just a simple simulation. It helped show how threads, timing, and shared data structures need to be managed carefully when building software that behaves like real monitoring equipment.

Execution Results: (Images found on the README)

When running the system, the three patients' vitals change constantly, and each one behaves differently depending on how the random simulation affects them. In normal speed, the changes happen slowly and alerts appear at a manageable rate. In fast mode, alerts show up much

more often, which helps demonstrate how a real hospital monitor can become overwhelmed when a patient's condition goes downhill quickly.

During testing, Alice usually stayed fairly stable, although she did trigger occasional warnings. John tended to have more unstable oxygen levels in fast mode, which caused more critical alerts. Maria was generally steady, making her helpful for comparison. These differences made the dashboard feel dynamic rather than repetitive.

The UI clearly showed the severity of alerts through colors and labels, and the filtering buttons helped isolate the important ones. The pause feature also made testing easier by letting me freeze the simulation temporarily. Overall, the results showed that the background thread, alert algorithm, and UI were working together as intended.

Conclusion:

This project taught us a lot about how background processes and web applications can work together. The biggest takeaway was how important it is to manage shared state safely when multiple parts of a program run at the same time. The simulation controls were a good example of this because the user can change the speed while the monitoring thread is already running. I also learned how much small design choices affect the clarity of a user interface, especially when dealing with alerts. There were a few challenges, such as figuring out how to serve static files in Flask when using blueprints and making sure the monitoring thread started only once. But fixing those issues helped me understand the structure of Flask applications better. Overall, the project did what we wanted it to. It simulated live patient monitoring, produced realistic alerts, and used operating system concepts like threading, timing, and concurrency in a meaningful way.