CMPSC 472 – Operating Systems--Section 001

Professor: Janghoon Yang

Author: Erand Vejseli & Jomiloju Odumosu

Project 2:

Community Clinics Scheduler

1. Description of the Project

The Community Clinic Load Manager is a simulation system that applies Operating System scheduling concepts to the real-world problem of distributing patient appointments across

multiple community clinics. Each clinic is modeled as a "node" with limited capacity, and each appointment request is treated as a scheduling job. The system routes incoming requests to the clinic with the lowest predicted wait time, similar to Shortest Job First (SJF) scheduling. A live dashboard displays clinic load, utilization, and routing history in real time.

2. Significance of the Project

Healthcare systems often experience uneven workloads, where some clinics become overloaded while others remain underutilized. We understand that this is mainly because people do not really trust small clinics, and they try to go for the more well-known ones, believing that they have better doctors as well. But most of the times the waiting time is very very long, usually patients after a few months. This leads to inefficient resource usage and helping patients to get the needed appointments in time. This project demonstrates how operating system scheduling principles can directly improve healthcare resource allocation:

- SJF reduces overall waiting time.
- Priority scheduling ensures urgent patients receive care faster.
- Load balancing prevents one clinic from becoming overburdened.
- Real-time updates mirror how OS schedulers operate under continuous job arrivals.

The model shows that technical scheduling theories from OS courses can provide real-world value in healthcare environments. In addition to smarter routing, the model also captures how different clinics process patients at different speeds. Clinics with higher capacity clear their load faster because their "service rate" is higher in the simulation. This mirrors a multiprocessor or multi-core system in operating systems, where more powerful cores can complete more work per unit time. By combining routing and processing behavior, the project gives a more realistic view of how OS scheduling principles can be applied to healthcare operations.

3. Code Structure

The code is organized into clear, modular components:

main.py

- Defines the FastAPI routes
- Hosts the live dashboard

- Integrates scheduling logic and state

models.py

- Data models for clinics, appointment requests, routing events

state.py

- Global system state
- Tracks clinics, loads, requests, and history
- Implements load decay (simulates clinics working through their backlog)

scheduler.py

- Implements the hybrid SJF + Priority algorithm
- Includes override logic for urgent requests
- Records the reason for each scheduling decision

simulation.py

- Background worker that periodically decreases each clinic's load
- Simulates clinics completing patient cases over tim

Code Structure Diagram

Community Clinic Load Manager

```
│
├── main.py
│     ├─ API routes
│     └─ Dashboard UI
│
├── models.py
│     ├─ Clinic model
│     ├─ Appointment model
│     └─ Routing event model
│
├── scheduler.py
│     └─ SJF/predicted wait time routing
│
├── simulation.py
```

```
│    └── Load decay background thread
│
└── state.py
    └── Shared state + mutex synchronization
```

4. Description of Algorithms

Shortest Job First (SJF) Component

For routine patients (urgency 1–5), scheduling is based on predicted wait time:

predicted_wait = (current_load + expected_duration) / capacity

The clinic with the lowest predicted wait is selected.

Priority Scheduling Override (High-Urgency Patients)

- For high-urgency patients (urgency 6–10), the system applies a **priority override**:
- It first computes the SJF choice (which clinic has the smallest predicted wait).
- Then, for high-urgency requests, it prefers the clinic with the **highest capacity** (the "largest hospital").
- If that clinic is different from the pure SJF choice, the system records the reason as priority_override in the routing history.
- This models **Priority Scheduling** layered on top of SJF: most patients follow SJF, but urgent cases are routed to the most capable clinic, even if the wait there is slightly longer.

Capacity-Based Processing Rate (Load Decay Algorithm)

- Each clinic also processes its existing workload over time. This is modeled with a **load decay algorithm** in the background worker:

  effective_step = (decay_step * center.capacity) / 10
  center.current_load = max(0.0, center.current_load - effective_step)
- center.current_load represents the total "work" waiting at that clinic.
- center.capacity controls how much work it can clear per decay tick.
- Higher-capacity clinics remove more load per tick, so they "treat" patients faster.

- This behavior is analogous to a **multi-core or multiprocessor system** in operating systems, where cores with more resources or higher performance can complete more CPU bursts in the same time interval. In our model, small clinics behave like slower cores, and large clinics behave like faster cores.

Synchronization

All updates to clinic load occur inside a mutex-protected critical section:

```
with center.lock:
    center.current_load += expected_duration
```

This prevents race conditions when multiple requests arrive concurrently.

5. Verification of Algorithms with Toy Examples

Example 1: SJF Routing

Clinics:

- Clinic A: capacity 3, current_load = 30
- Clinic B: capacity 5, current_load = 10

Request: expected_duration = 10

Predicted wait:

- A → (30 + 10) / 3 = 13.3
- B → (10 + 10) / 5 = 4

Result: Request is routed to Clinic B (shorter predicted completion time).

Example 2: Priority Tie-Break

Clinic loads produce equal predicted wait:

- Clinic A: predicted_wait = 5
- Clinic B: predicted_wait = 5

Request A (urgency 2) is routed before Request B (urgency 7).

Example 3: Capacity-based processing (load decay)

Clinics:

- Clinic A: capacity = 2, current_load = 20
- Clinic B: capacity = 10, current_load = 20

With the decay rule:

effective_step = (decay_step * capacity) / 10
(decay_step = 1.0 for illustration)

Per tick:

- Clinic A removes (1.0 × 2) / 10 = 0.2 load
- Clinic B removes (1.0 × 10) / 10 = 1.0 load

After 10 ticks:

- Clinic A: 20 − (0.2 × 10) = 18
- Clinic B: 20 − (1.0 × 10) = 10

This confirms that higher-capacity clinics process work significantly faster, just like faster CPU cores in an OS. In the dashboard, this is visible as the higher-capacity clinic's load bar decreasing more quickly.

6. Functionalities

- **Add Clinic:** Define clinic name and capacity.
- **Submit Appointment:** Provide urgency and expected duration.
- **Live Load Balancing:** System routes each request to the least-loaded clinic.
- **Load Decay Simulation:** Background process reduces clinic loads over time.
- **Live Dashboard:** Displays current loads, utilization bars, clinic status, and routing events.

- **API Documentation:** Interactive Swagger UI at /docs.

7. Execution Results & Analysis

The system successfully demonstrates real-time load balancing across community clinics. As appointments are submitted, the dashboard shows the load increasing instantly and decreasing over time through decay. The decay behavior also clearly shows that high-capacity clinics clear their backlogs faster than low-capacity clinics, which matches the intuition of multi-core CPU scheduling where more powerful cores complete more work per time unit. The routing history confirms that requests consistently choose the clinic with the shortest predicted wait, matching SJF behavior. By visualizing load distribution and algorithm decisions, the tool highlights how OS scheduling methods can reduce congestion and create balanced appointment flows. Screenshots of the dashboard during test runs show clear differences between high-capacity and low-capacity clinics, confirming the meaningfulness of the algorithm.

8. Conclusions

The Community Clinic Load Manager shows how Operating Systems concepts such as scheduling, concurrency, and critical-section protection can be applied to real-world healthcare challenges. The project demonstrates effective load distribution, reduced wait times, and dynamic updates using an SJF-inspired algorithm. Some limitations include the simplicity of the simulation and the lack of real-world data. Future improvements could include additional scheduling modes, predictive modeling, or integration with real clinic datasets. Overall, the project strengthened our understanding of OS scheduling, synchronization, and system design.