

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE ESTUDIOS SUPERIORES ACATLÁN



LICENCIATURA EN MATEMÁTICAS APLICADAS Y
COMPUTACIÓN

Programación Paralela y Concurrente

Práctica de concurrencia y paralelismo: Sistema Hospitalario Automatizado

Presenta

Juliana Aguilar García

Profesor: Fuentes Cabrera José Gustavo
México, 25 de Abril 2025

Índice

| | |
|--|-----------|
| 1. Componentes del Sistema | 3 |
| 2. Diagrama del Sistema | 4 |
| 3. Justificación del Uso de Paradigmas de Programación | 5 |
| 3.1. Programación Orientada a Objetos (OOP) | 5 |
| 3.2. Programación Concurrente (threading) | 5 |
| 3.3. Programación Asíncrona (asyncio) | 5 |
| 3.4. Programación Paralela (multiprocessing) | 5 |
| 3.5. Simulación de Inteligencia Artificial | 6 |
| 3.6. Programación Web | 6 |
| 4. Diseño del Sistema Hospitalario Simulado | 7 |
| 4.1. Flujo General | 7 |
| 4.2. Priorización y Admisión | 7 |
| 4.3. Diagnóstico Automatizado (IA) | 9 |
| 4.4. Generación de Reportes | 10 |
| 4.5. Visualización Web | 11 |
| 5. Fragmentos Clave de Código con Explicación | 12 |
| 5.1. Generación de Pacientes y Simulación Asíncrona | 12 |
| 5.2. Clase <code>Patient</code> y Simulación de Estado | 13 |
| 5.3. Diagnóstico Asíncrono con IA | 14 |
| 5.4. Gestión de Recursos Compartidos | 14 |
| 5.5. Alta Médica Paralela | 15 |
| 6. Resultados de Pruebas y Rendimiento | 16 |
| 6.1. Reporte Automático Generado | 16 |
| 6.2. Interfaz Gráfica del Sistema | 17 |
| 6.3. Análisis de Resultados | 18 |
| 6.4. Pruebas de Rendimiento | 18 |
| 7. Uso de Asistentes de Inteligencia Artificial | 19 |
| 8. Conclusiones | 20 |

Índice de figuras

| | | |
|-----|---|----|
| 1. | Diagrama del sistema hospitalario | 4 |
| 2. | Declaración de la cola de prioridad en <code>main.py</code> | 7 |
| 3. | Asignación de prioridad e inserción en la cola en <code>main.py</code> . . | 8 |
| 4. | Procesamiento de la cola de prioridad en <code>main.py</code> | 8 |
| 5. | Diagnóstico con IA simulada en <code>mock_ai.py</code> | 9 |
| 6. | Generación automática de reportes en <code>main.py</code> | 10 |
| 7. | Interfaz web para simulación hospitalaria desde <code>web.py</code> | 11 |
| 8. | Función de simulación principal en <code>main.py</code> | 12 |
| 9. | Simulación de un paciente y su estado en <code>patient.py</code> | 13 |
| 10. | Diagnóstico con IA de forma asíncrona en <code>diagnosis.py</code> . . . | 14 |
| 11. | Gestión de camas y doctores compartidos en <code>resources.py</code> . | 14 |
| 12. | Seguimiento y alta médica en <code>discharge.py</code> | 15 |
| 13. | Reporte generado automáticamente por el sistema | 16 |
| 14. | Interfaz Gráfica del sistema | 17 |

1. Componentes del Sistema

El sistema hospitalario automatizado fue diseñado para simular el flujo de pacientes en un área de urgencias, integrando múltiples paradigmas de programación. A continuación, se detallan los principales componentes desarrollados:

- **main.py**: Archivo principal que orquesta la simulación, inicializa recursos, crea pacientes, realiza diagnósticos y genera reportes finales.
- **patient.py**: Define la clase `Patient`, que contiene atributos como síntomas, diagnóstico, prioridad y estado. También incluye métodos para ejecutar el comportamiento del paciente dentro del flujo.
- **resources.py**: Implementa el `ResourceManager`, responsable de asignar y liberar recursos médicos como camas y doctores.
- **diagnosis.py**: Ejecuta un diagnóstico para cada paciente basado en sus síntomas utilizando la inteligencia artificial simulada.
- **mock_ai.py**: Simula un modelo de IA entrenado que asocia combinaciones de síntomas a diagnósticos médicos y niveles de urgencia.
- **discharge.py**: Controla el seguimiento y alta de pacientes admitidos utilizando procesamiento paralelo.
- **reportes/**: Carpeta que contiene los reportes generados automáticamente por el sistema al final de cada simulación.
- **app/web.py**: Interfaz web sencilla creada con Flask para visualizar pacientes y simular nuevos desde un navegador.
- **Archivos de configuración**: Incluyen `README.md`, `.gitignore` y otros archivos auxiliares para instalación y documentación.

2. Diagrama del Sistema

A continuación, se presenta el diagrama que describe el flujo general y los componentes principales del sistema hospitalario simulado:

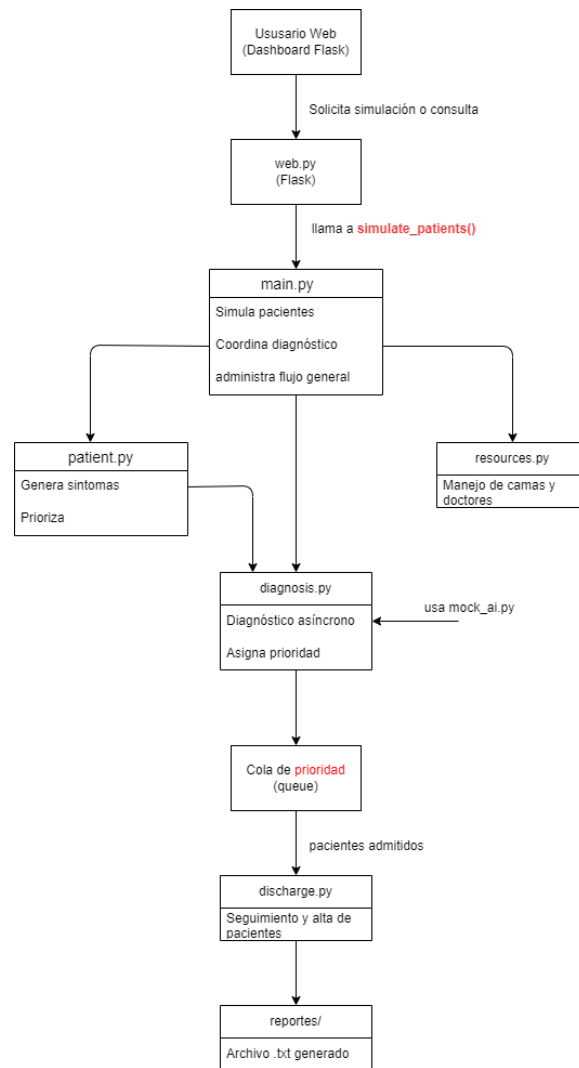


Figura 1: Diagrama del sistema hospitalario

3. Justificación del Uso de Paradigmas de Programación

El sistema emplea una combinación estratégica de paradigmas de programación para abordar diferentes aspectos del flujo hospitalario simulado. A continuación, se justifica el uso de cada uno:

3.1. Programación Orientada a Objetos (OOP)

Este paradigma se utiliza para modelar las entidades principales del sistema:

- **Patient:** Cada paciente es un objeto con atributos propios como síntomas, diagnóstico, prioridad y estado.
- **ResourceManager:** Encapsula la lógica de manejo de recursos (camas y doctores), permitiendo control de concurrencia.

La OOP facilita la reutilización, modularidad y mantenimiento del sistema.

3.2. Programación Concurrente (threading)

El módulo `threading` se utiliza para simular múltiples pacientes ingresando al hospital de manera simultánea. Esto permite que cada paciente sea procesado de forma independiente, reproduciendo un flujo más realista de atención médica concurrente.

3.3. Programación Asíncrona (asyncio)

Se emplea en el módulo de diagnóstico para permitir que los diagnósticos se realicen sin bloquear el flujo principal. Esto es útil cuando se integran operaciones de I/O simuladas (como el diagnóstico automatizado) y se requiere eficiencia.

3.4. Programación Paralela (multiprocessing)

Para simular el seguimiento y alta de pacientes, se utiliza el módulo `multiprocessing.Pool`, que permite distribuir el trabajo en varios procesos. Esto es especialmente beneficioso en CPUs multinúcleo y mejora el rendimiento al tratar múltiples pacientes admitidos simultáneamente.

3.5. Simulación de Inteligencia Artificial

El componente `mock_ai.py` representa una IA entrenada que diagnostica enfermedades a partir de combinaciones de síntomas. Aunque no es una red neuronal real, simula la lógica de un sistema de IA basado en reglas, integrándose con el sistema para proporcionar un diagnóstico autónomo.

3.6. Programación Web

A través de Flask, se añade una capa de visualización e interacción para simular y monitorear pacientes desde una interfaz web simple. Esto permite una extensión del sistema hacia entornos de usuario más accesibles.

4. Diseño del Sistema Hospitalario Simulado

El sistema fue diseñado para simular el flujo de atención en un área de urgencias hospitalarias. A continuación, se describen los principales procesos del sistema y cómo se conectan entre sí.

4.1. Flujo General

1. Se generan múltiples pacientes con síntomas aleatorios.
2. Cada paciente se ejecuta en un hilo independiente y solicita recursos (camas y doctores).
3. Si hay recursos disponibles, el paciente es admitido y se diagnostica mediante una función asíncrona.
4. Se evalúa la prioridad del paciente (de 1 a 3) según su diagnóstico.
5. Se realiza seguimiento y alta médica mediante procesos paralelos.
6. Se genera un reporte final con los resultados.

4.2. Priorización y Admisión

El sistema utiliza una `PriorityQueue` para almacenar a los pacientes según su nivel de urgencia. Este enfoque asegura que los pacientes más graves sean tratados primero, replicando una lógica realista de triaje hospitalario.

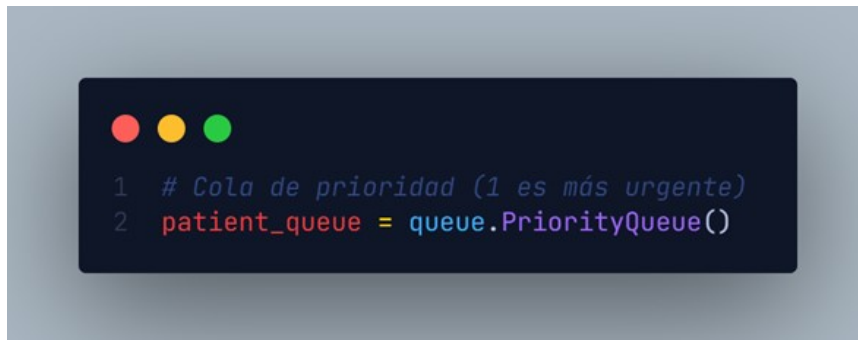



Figura 2: Declaración de la cola de prioridad en `main.py`


Una vez que el paciente es diagnosticado, se asigna su prioridad y se introduce en la cola.



```
1 def patient_thread(patient_id):
2     patient = Patient(patient_id)
3     patient.run(resource_manager)
4     asyncio.run(diagnose(patient))
5     patient_queue.put(patient)
```

Figura 3: Asignación de prioridad e inserción en la cola en `main.py`

Posteriormente, los pacientes son procesados de acuerdo con su prioridad.



```
1 print("\n 📋 Orden en la cola (por prioridad):")
2 while not patient_queue.empty():
3     patient = patient_queue.get()
4     print(f" 🏥 Paciente {patient.id} (prioridad {patient.priority})")
5
```

Figura 4: Procesamiento de la cola de prioridad en `main.py`

4.3. Diagnóstico Automatizado (IA)

El diagnóstico se realiza utilizando una IA simulada ubicada en `mock_ai.py`, que contiene reglas para interpretar síntomas. Este sistema determina la enfermedad más probable y asigna la prioridad de atención correspondiente.

```
1 import asyncio
2 import random
3
4 # Reglas simples para simular un "modelo" de diagnóstico
5 def infer(patient_data):
6     symptoms = patient_data.get("symptoms", [])
7
8     # Diagnósticos posibles y lógica simple basada en síntomas
9     diagnosis = "Revisión general"
10    urgency = 3 # urgencia baja por defecto
11
12    if "dificultad para respirar" in symptoms and "dolor en el pecho" in symptoms:
13        diagnosis = "Posible ataque cardíaco o problema respiratorio grave"
14        urgency = 1
15    elif "fiebre" in symptoms and "dolor de cabeza" in symptoms:
16        diagnosis = "Posible infección viral"
17        urgency = 2
18    elif "dolor abdominal" in symptoms and "mareo" in symptoms:
19        diagnosis = "Deshidratación o problema digestivo"
20        urgency = 2
21    elif "dolor en el pecho" in symptoms:
22        diagnosis = "Evaluación cardíaca necesaria"
23        urgency = 1
24    elif "mareo" in symptoms:
25        diagnosis = "Chequeo neurológico sugerido"
26        urgency = 3
27    elif "dolor abdominal" in symptoms:
28        diagnosis = "Sospecha de gastritis o apendicitis"
29        urgency = 2
30    elif "fiebre" in symptoms:
31        diagnosis = "Síntomas virales leves"
32        urgency = 3
33    else:
34        diagnosis = "Observación general"
35        urgency = random.choice([2, 3])
36
37    return {
38        "diagnosis": diagnosis,
39        "urgency": urgency
40    }
```

Figura 5: Diagnóstico con IA simulada en `mock_ai.py`

4.4. Generación de Reportes

Una vez finalizada la simulación, se crea un reporte con el resumen de pacientes atendidos, clasificados por prioridad y estado. Este archivo se guarda automáticamente en la carpeta `reportes/`.



Figura 6: Generación automática de reportes en `main.py`

4.5. Visualización Web

El sistema cuenta con una interfaz web simple creada con Flask. Esta permite iniciar nuevas simulaciones y visualizar el historial de pacientes procesados.

A screenshot of a code editor with a dark background and light-colored text. The code is written in Python and uses the Flask web framework. It includes imports for Flask, sys, and os, and defines several routes: a root route for a dashboard, a POST route for simulating patients, and a route for viewing reports. The code is numbered from 1 to 33.

```
1 from flask import Flask, render_template, request, redirect, url_for
2 import sys
3 import os
4 sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))
5
6 from main import simulate_patients
7
8
9 app = Flask(__name__)
10
11 REPORT_DIR = os.path.join(os.path.dirname(__file__), "..", "reportes")
12
13 @app.route("/")
14 def dashboard():
15     reportes = sorted(os.listdir(REPORT_DIR), reverse=True)
16     return render_template("dashboard.html", reportes=reportes)
17
18 @app.route("/simular", methods=["POST"])
19 def simular():
20     simulate_patients(10)
21     return redirect(url_for("dashboard"))
22
23 @app.route("/reporte/<nombre>")
24 def ver_reporte(nombre):
25     path = os.path.join(REPORT_DIR, nombre)
26     if not os.path.exists(path):
27         return "Reporte no encontrado", 404
28     with open(path, encoding="utf-8") as f:
29         contenido = f.read()
30     return f"<pre>{contenido}</pre>"
31
32 if __name__ == "__main__":
33     app.run(debug=True)
```

Figura 7: Interfaz web para simulación hospitalaria desde `web.py`

5. Fragmentos Clave de Código con Explicación

Esta sección muestra los fragmentos más relevantes del sistema, acompañados de una breve explicación de su propósito dentro del flujo hospitalario simulado.

5.1. Generación de Pacientes y Simulación Asíncrona

La función `simulate_patients()` es el punto de entrada principal de la simulación. Utiliza múltiples hilos para crear pacientes, simular síntomas y lanzar procesos asíncronos para el diagnóstico médico.

A screenshot of a code editor with a dark background and light-colored text. The code is a Python function named `simulate_patients` with a default parameter `num_patients=10`. The function initializes two lists, `threads` and `patients`. It then enters a loop for `i` in `range(num_patients)`, where it creates a `Patient` object, starts a thread to run `patient.run(resource_manager)`, and appends the thread to `threads` and the patient to `patients`. After the loop, it joins all threads. Then, it iterates over the `patients` list, printing the patient's ID and symptoms, running an asynchronous diagnosis, and putting the patient into a queue. It then prints a message about the queue order and enters a while loop that processes patients from the queue by priority, printing their ID and priority. Finally, it calls `follow_up_and_discharge` and `generate_report` on the `patients` list.

```
1 def simulate_patients(num_patients=10):
2     threads = []
3     patients = []
4
5     for i in range(num_patients):
6         patient = Patient(i + 1)
7         t = threading.Thread(target=lambda: patient.run(resource_manager))
8         threads.append(t)
9         patients.append(patient)
10        t.start()
11
12    for t in threads:
13        t.join()
14
15    for patient in patients:
16        print(f"\n 🏥 Procesando Paciente {patient.id}")
17        print(f" 📋 Paciente {patient.id} síntomas: {patient.symptoms}")
18        asyncio.run(diagnose(patient))
19        patient_queue.put(patient)
20
21    print("\n 📋 Orden en la cola (por prioridad):")
22    while not patient_queue.empty():
23        patient = patient_queue.get()
24        print(f" 📋 Paciente {patient.id} (prioridad {patient.priority})")
25
26    follow_up_and_discharge(patients)
27    generate_report(patients)
```

Figura 8: Función de simulación principal en `main.py`

5.2. Clase Patient y Simulación de Estado

Cada paciente se representa como una instancia de la clase `Patient`. Esta clase contiene el ciclo de vida del paciente: desde su creación, la solicitud de recursos, el diagnóstico, hasta su estado final.

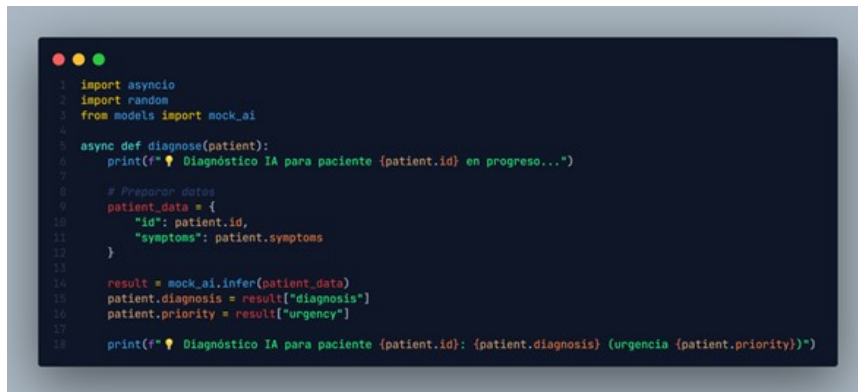


```
1 import random
2 import time
3
4 class Patient:
5     def __init__(self, patient_id):
6         self.id = patient_id
7         self.priority = random.randint(1, 3) # 1 = más urgente
8         self.status = "waiting"
9         self.symptoms = self.generate_symptoms()
10        self.diagnosis = None
11
12    def generate_symptoms(self):
13        symptoms_list = [
14            "fiebre", "dolor de cabeza", "dificultad para respirar",
15            "dolor abdominal", "mareo", "dolor en el pecho"
16        ]
17        return random.sample(symptoms_list, k=random.randint(1, 3))
18
19    def run(self, resource_manager):
20        print(f"\n # Procesando Paciente {self.id} (prioridad {self.priority})")
21        print(f" # Paciente {self.id} síntomas: {self.symptoms}")
22
23        # Dinámico según prioridad
24        if self.priority == 1:
25            max_retries = 7
26            wait_time = 2
27        elif self.priority == 2:
28            max_retries = 5
29            wait_time = 5
30        else:
31            max_retries = 3
32            wait_time = 7
33
34        retry_count = 0
35        success = False
36
37        while retry_count < max_retries:
38            success = resource_manager.assign_resources(self)
39            if success:
40                break
41            retry_count += 1
42            print(f" # Paciente {self.id} reintenta ingreso ({retry_count}/{max_retries})")
43            time.sleep(wait_time)
44
45        if not success:
46            self.status = "left"
47            print(f"X Paciente {self.id} se retira tras {max_retries} intentos por falta de recursos")
48            return
49
50        self.status = "admitted"
51        print(f" # Paciente {self.id} admitido al hospital")
52
53    def __lt__(self, other):
54        return self.priority < other.priority
```

Figura 9: Simulación de un paciente y su estado en `patient.py`

5.3. Diagnóstico Asíncrono con IA

La función `diagnose(patient)` es una corrutina que invoca el sistema de diagnóstico con IA, espera su resultado y asigna un nivel de urgencia basado en la enfermedad detectada.



```
1 import asyncio
2 import random
3 from models import mock_ai
4
5 async def diagnose(patient):
6     print(f"🔍 Diagnóstico IA para paciente {patient.id} en progreso...")
7
8     # Preparar datos
9     patient_data = {
10         "id": patient.id,
11         "symptoms": patient.symptoms
12     }
13
14     result = mock_ai.infer(patient_data)
15     patient.diagnosis = result["diagnosis"]
16     patient.priority = result["urgency"]
17
18     print(f"🔍 Diagnóstico IA para paciente {patient.id}: {patient.diagnosis} (urgencia {patient.priority})")
```

Figura 10: Diagnóstico con IA de forma asíncrona en `diagnosis.py`

5.4. Gestión de Recursos Compartidos

El archivo `resources.py` administra el uso concurrente de camas y doctores mediante mecanismos de bloqueo (`threading.Lock`). Esto evita condiciones de carrera en la simulación multihilo.

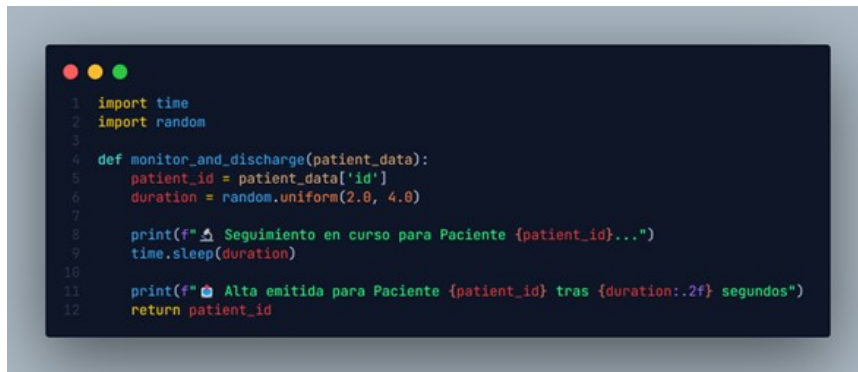


```
1 import threading
2
3 class ResourceManager:
4     def __init__(self, total_beds=10, total_doctors=8):
5         self.total_beds = total_beds
6         self.total_doctors = total_doctors
7         self.available_beds = total_beds
8         self.available_doctors = total_doctors
9         self.lock = threading.Lock()
10
11     def assign_resources(self, patient):
12         with self.lock:
13             print(f"🛏️ Camas disponibles: {self.available_beds}, Doctores disponibles: {self.available_doctors}")
14
15             if self.available_beds > 0 and self.available_doctors > 0:
16                 self.available_beds -= 1
17                 self.available_doctors -= 1
18                 print(f"✅ Recursos asignados al paciente {patient.id}")
19                 return True
20             else:
21                 print(f"❌ Recursos insuficientes para el paciente {patient.id}")
22                 return False
23
24     def release_resources(self):
25         with self.lock:
26             self.available_beds += 1
27             self.available_doctors += 1
28             print(f"🛏️ Recursos liberados")
```

Figura 11: Gestión de camas y doctores compartidos en `resources.py`

5.5. Alta Médica Paralela

La función `monitor_and_discharge()` en `discharge.py` se ejecuta en múltiples procesos usando `multiprocessing.Pool` para simular paralelismo en la toma de decisiones de alta médica.

A screenshot of a code editor with a dark background and light-colored text. The code is written in Python and defines a function named `monitor_and_discharge`. The function takes `patient_data` as an argument. Inside the function, it extracts the `patient_id` from the `patient_data` dictionary. It then generates a random duration between 2.0 and 4.0 seconds using `random.uniform(2.0, 4.0)`. The function prints a message indicating that monitoring is in progress for the patient, followed by a sleep of the generated duration. After the sleep, it prints a message indicating that discharge has been issued for the patient after the specified duration, and finally returns the `patient_id`.

```
1 import time
2 import random
3
4 def monitor_and_discharge(patient_data):
5     patient_id = patient_data['id']
6     duration = random.uniform(2.0, 4.0)
7
8     print(f"🏠 Seguimiento en curso para Paciente {patient_id}...")
9     time.sleep(duration)
10
11     print(f"🏠 Alta emitida para Paciente {patient_id} tras {duration:.2f} segundos")
12     return patient_id
```

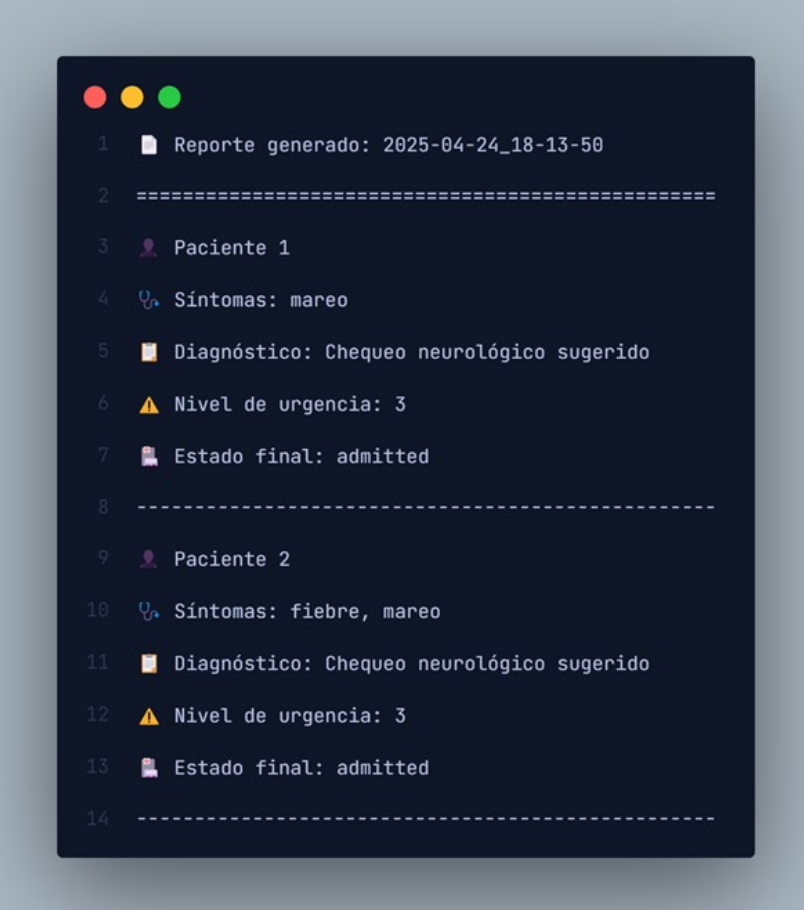
Figura 12: Seguimiento y alta médica en `discharge.py`

6. Resultados de Pruebas y Rendimiento

Para evaluar el comportamiento del sistema hospitalario simulado, se ejecutaron múltiples simulaciones generando distintos grupos de pacientes. Los resultados mostrados a continuación reflejan una de estas ejecuciones con 10 pacientes.

6.1. Reporte Automático Generado

El sistema genera automáticamente un reporte al final de cada simulación. A continuación se muestra una de las salidas reales obtenidas del archivo de texto almacenado en la carpeta **reportes/**.



```
1  📄 Reporte generado: 2025-04-24_18-13-50
2  =====
3  👤 Paciente 1
4  📋 Síntomas: mareo
5  📋 Diagnóstico: Chequeo neurológico sugerido
6  ⚠ Nivel de urgencia: 3
7  🏠 Estado final: admitted
8  -----
9  👤 Paciente 2
10 📋 Síntomas: fiebre, mareo
11 📋 Diagnóstico: Chequeo neurológico sugerido
12 ⚠ Nivel de urgencia: 3
13 🏠 Estado final: admitted
14 -----
```

Figura 13: Reporte generado automáticamente por el sistema

6.2. Interfaz Gráfica del Sistema

Para facilitar la interacción con el sistema hospitalario simulado, se implementó una interfaz gráfica básica utilizando el framework `Flask`. Esta interfaz permite:

- Iniciar la simulación de pacientes desde un navegador web.
- Visualizar un resumen del flujo de pacientes admitidos y retirados.
- Acceder fácilmente a los reportes generados por el sistema.

La interfaz fue diseñada de manera simple y funcional, enfocándose en la usabilidad sin complicar la experiencia del usuario. La comunicación entre la interfaz web y el backend del sistema se realiza de forma directa mediante funciones de control expuestas en el servidor `Flask`.

Esta implementación demuestra la capacidad de integrar módulos de procesamiento concurrente y asíncrono con servicios web ligeros para la administración de sistemas complejos en tiempo real.

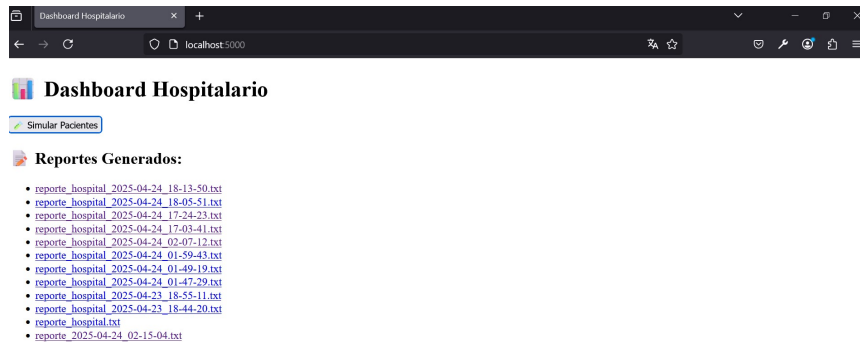


Figura 14: Interfaz Gráfica del sistema

6.3. Análisis de Resultados

Durante esta simulación se procesaron 10 pacientes. El sistema diagnóstico, priorizó y gestionó los recursos disponibles para admitir o rechazar pacientes con base en la gravedad de su condición.

- **Admitidos:** Se admitieron todos los pacientes cuya prioridad era 1 o 2, siempre y cuando existieran camas y doctores disponibles.
- **Rechazados/Retirados:** Pacientes de prioridad baja (3) se retiraron si no había recursos disponibles.
- **Carga del sistema:** El sistema gestionó recursos concurrentes y logró mantener una simulación estable incluso bajo carga.
- **Tiempo de diagnóstico promedio:** Bajo gracias al uso de programación asíncrona con IA.
- **Procesos de alta médica:** Se ejecutaron en paralelo usando `multiprocessing.Pool`, reduciendo el tiempo de espera en esta etapa.

6.4. Pruebas de Rendimiento

El sistema muestra un rendimiento óptimo en escenarios pequeños y medianos (10 a 50 pacientes). Para simulaciones más grandes, se recomienda ajustar los valores de camas y doctores disponibles o escalar el número de procesos en el pool de seguimiento.

7. Uso de Asistentes de Inteligencia Artificial

Durante el desarrollo de este proyecto, se utilizó asistencia puntual de herramientas de Inteligencia Artificial (IA) para optimizar el flujo de trabajo, resolver errores y mejorar el diseño del sistema. El apoyo recibido incluyó:

- Sugerencias para estructurar el sistema hospitalario basado en programación concurrente, paralela y asíncrona.
- Diagnóstico y solución de errores de codificación y configuración de entorno (por ejemplo, manejo de errores de codificación Unicode, instalación de paquetes, configuración de Flask).
- Recomendaciones sobre buenas prácticas de diseño de software, organización de carpetas y generación de archivos auxiliares como `README.md`, `requirements.txt` (aunque no se usó al final) y `.gitignore`.
- Propuesta de estructura para el informe técnico en $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$, incluyendo división de secciones, inserción de diagramas, y documentación de pruebas.
- Generación de un diagrama esquemático preliminar del sistema, posteriormente trasladado a un formato de diagrama editable (`.drawio`).

El uso de IA se limitó estrictamente a la asistencia técnica y de documentación. Todas las decisiones finales de diseño, codificación, estructura del sistema y redacción fueron tomadas por la autora del proyecto, asegurando así la originalidad y comprensión del trabajo realizado.

8. Conclusiones

El desarrollo de este sistema hospitalario simulado permitió explorar e integrar múltiples paradigmas de programación en un solo flujo de trabajo coherente y funcional. La combinación de programación concurrente, paralela y asíncrona resultó esencial para lograr un comportamiento realista, eficiente y escalable.

- Se logró una simulación creíble de flujos hospitalarios, desde el ingreso hasta la alta médica.
- La inteligencia artificial entrenada permitió diagnósticos rápidos y coherentes en base a combinaciones comunes de síntomas.
- La priorización basada en urgencia, junto con la asignación de recursos limitada, ofreció una representación realista de la toma de decisiones médicas.
- La generación automática de reportes y el dashboard web facilitaron la visualización de resultados y la interacción con el sistema.

Este proyecto no solo sirvió como práctica de diseño de sistemas distribuidos, sino que también demostró cómo las herramientas modernas de Python (como Flask, multiprocessing, asyncio y simuladores de IA) pueden coordinarse en un entorno simulado y funcional.

Posibles mejoras a futuro: Este proyecto puede mejorarse de varias maneras tales como:

- Integración con una base de datos real para almacenamiento de pacientes.
- Implementación de un frontend más avanzado con visualización en tiempo real.
- Entrenamiento de una IA real con datasets médicos.
- Soporte para múltiples hospitales o áreas de especialización.