

índice

OBJETIVO:	3
INTRODUCCIÓN:	3
DIAGRAMA DEL SISTEMA:	4
DISEÑO DETALLADO	4
ESTRUCTURA MODULAR	4
CÓDIGO CLAVE EXPLICADO	6
RELACIÓN CON LA ARQUITECTURA	11
PRUEBAS Y RENDIMIENTO	11
CONCLUSIÓN:	16
REFERENCIAS:	16

Objetivo:

Que el estudiante aplique y diferencie los paradigmas de programación paralela, concurrente y asíncrona mediante la simulación de un sistema realista que requiere procesamiento distribuido de tareas en distintos tiempos y recursos.

Introducción:

Estamos desarrollando un hospital de manera virtual donde los pacientes no son solo números, sino entidades con síntomas dinámicos, prioridades que son cambiantes y recursos limitados. Eso es lo que armamos aquí: un ecosistema en Python que simula desde el registro de un paciente hasta su alta, pasando por triaje automatizado con IA, diagnóstico predictivo y hasta una lucha asíncrona por camas (¡de las cuales con motivos académicos solo hay 3!).

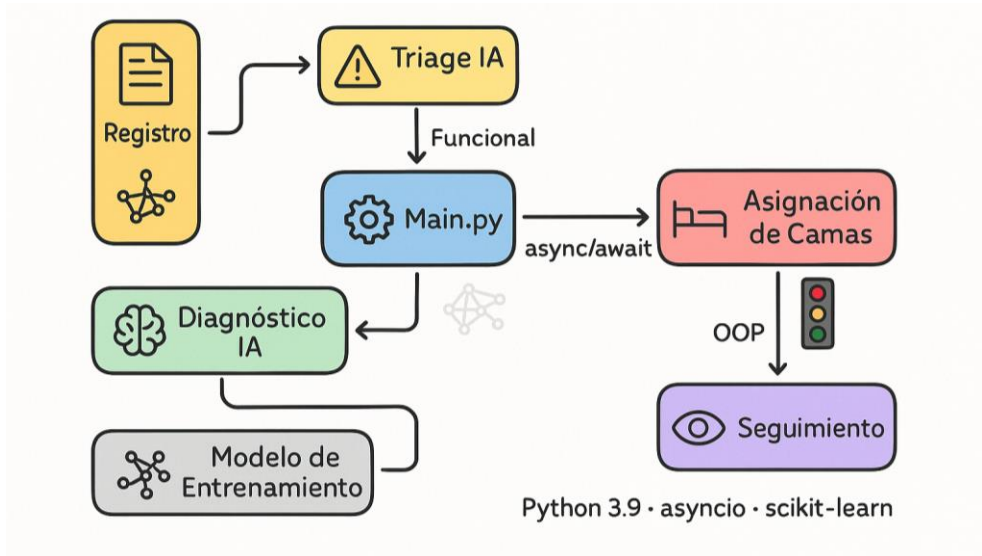
Las herramientas que usamos como `asyncio` para gestionar colas sin bloquear el sistema, `scikit-learn` para predecir enfermedades basadas en síntomas aleatorios, y hasta logs coloridos que parecen sacados de una terminal hacker (gracias al `ColoredFormatter`).

Pero no todo fue código: nos topamos con errores de concurrencia, modelos que no generalizaban, y hasta un deadlock en el semáforo de camas.

Este informe es la historia técnica de esos retos, las soluciones que implementamos (y las que descartamos), y cómo logramos que todo funcione en armonía. Si algo queda claro, es que cada línea de código aquí tiene un propósito, desde el `Semaphore` que controla las camas hasta llegar al `LabelEncoder` que traduce diagnósticos a números. No es perfecto, pero es nuestro. Y ahora, documentado para que otros puedan iterar sin perderse en el camino.



Diagrama del Sistema:



(Referencia: Imagen generada por inteligencia artificial con ChatGPT de OpenAI, 2025. Diagrama de flujo técnico de un sistema hospitalario simulado. Creado el 30 de abril de 2025 mediante prompt personalizado)

Diseño Detallado

Patrones de Diseño Aplicados

Patrón	Módulo/Clase	Propósito	Implementación Clave
Factory	`Paciente` (generación de síntomas)	Creación flexible de pacientes con síntomas aleatorios o predefinidos.	`generar_sintomas()` encapsula la lógica de construcción.
Singleton	`ColoredFormatter` (visualización)	Garantizar una única instancia del logger con configuración consistente.	Configuración centralizada en `main.py`
Semáforo	`asignacion_recursos.py`	Control concurrente de recursos limitados (camas) con `asyncio.Semaphore`.	`camas_disponibles = asyncio.Semaphore(3)`.
Pipeline	`flujo_paciente_async()` (main.py)	Orquestación secuencial de etapas clínicas (Registro → Triage → Diagnóstico → Cama → Seguimiento).	Uso de `await` para encadenar tareas asíncronas.

Estructura Modular

La arquitectura sigue un enfoque por responsabilidades claras:

- **main.py:** Controlador central que coordina el flujo asíncrono.
- **Módulos clínicos:**
 - ✓ registro.py: I/O bound (simulación de bases de datos).
 - ✓ triage_ia.py/diagnostico_ia.py: CPU bound (inferencia con modelos ML).
 - ✓ asignacion_recursos.py: Gestión asíncrona de recursos críticos.
- **Utils:**
 - ✓ visualizacion.py: Configuración de logs y estadísticas.
 - ✓ paciente.py: Entidad central con lógica de generación de datos.

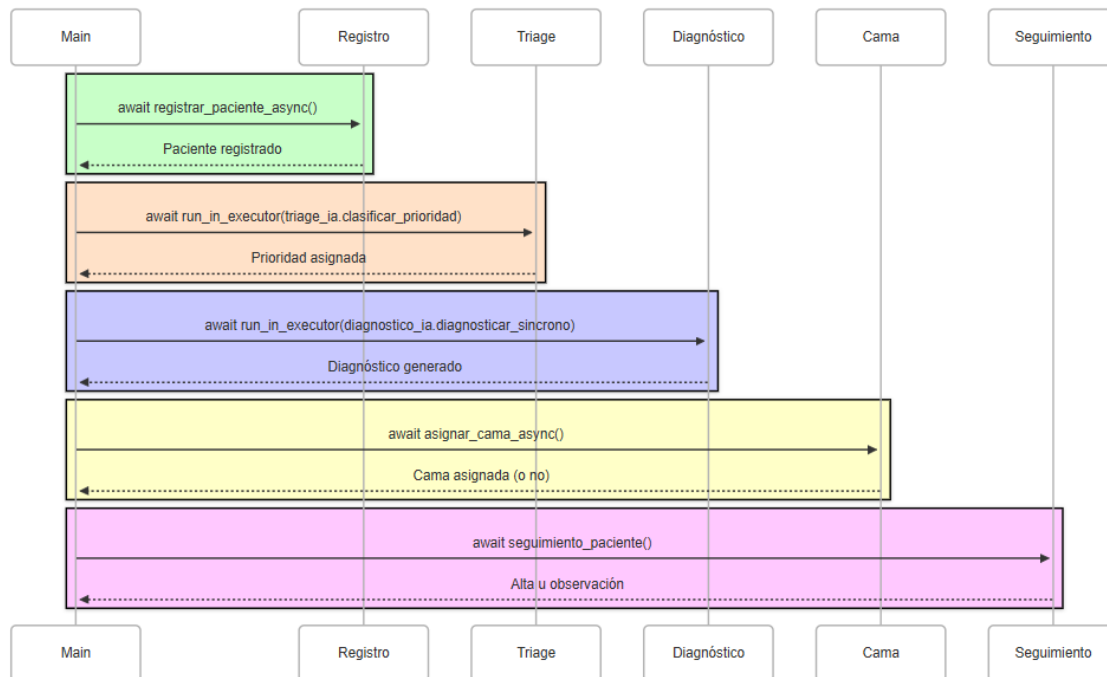
Librerías Clave

Librería	Uso	Ejemplo aplicado
asyncio	Concurrencia para I/O bound (registro, asignación de camas).	async def asignar_cama_async() en asignacion_recursos.py.
joblib	Serialización eficiente de modelos ML para inferencia rápida.	joblib.load("modelo_triage.pkl") en triage_ia.py.
sklearn	Entrenamiento y predicción con DecisionTreeClassifier.	Pipeline en modelo_entrenamiento.py.
pandas	Transformación de datos para diagnóstico (DataFrame de síntomas).	Pipeline en modelo_entrenamiento.py.

Desafíos Técnicos y Soluciones

1. **Sincronización de Recursos:**
 - ✓ **Problema:** Contienda por camas entre pacientes concurrentes.
 - ✓ **Solución:** Semáforos asíncronos (asyncio.Semaphore) para garantizar exclusión mutua.
2. **Integración de Modelos ML en Entorno Asíncrono:**
 - ✓ **Problema:** Inferencia bloqueante (CPU bound) en flujo asíncrono.
 - ✓ **Solución:** Uso de ProcessPoolExecutor para ejecutar modelos en paralelo sin bloquear el event loop.
3. **Consistencia de Logs:**
 - ✓ **Problema:** Formateo centralizado de logs con colores y emojis.
 - ✓ **Solución:** Patrón Singleton en ColoredFormatter (configuración única vía logging.getLogger(__name__)).

Diagrama de Secuencia (Ejemplo: Flujo de un Paciente)



Este diseño optimiza:

- **Escalabilidad:** Módulos independientes para añadir nuevas funcionalidades (ej: módulo de facturación).
- **Mantenibilidad:** Patrones bien definidos y separación de preocupaciones.
- **Rendimiento:** AsyncIO para I/O y multiprocessing para CPU.

Código Clave Explicado

Este informe documenta el desarrollo y la ejecución de una simulación de un flujo de pacientes en un entorno hospitalario.

La simulación utiliza programación asíncrona (`asyncio`), procesamiento paralelo (`ProcessPoolExecutor`) y modelos de aprendizaje automático para simular etapas clave como registro, triaje, diagnóstico, asignación de recursos (camas) y seguimiento. El objetivo es modelar y analizar el rendimiento del sistema bajo diferentes condiciones.

1. Orquestación del Flujo del Paciente (`main.py`)

La función `flujo_paciente_async` es el corazón de la simulación para cada paciente individual. Define la secuencia de etapas por las que pasa un paciente.

`main.py`

```
async def flujo_paciente_async(paciente, cpu_executor: ProcessPoolExecutor):
```

"""

Maneja el flujo completo de un paciente de forma asíncrona.

Orquesta las diferentes etapas del proceso hospitalario.

"""

```
logger.info(f"Paciente {paciente.id} [SIMULACION]: Iniciando flujo...")
```

```
try:
```

```
    # 1. Registro (I/O bound simulado)
```

```
    await registrar_paciente_async(paciente, actualizar_estadistica_global)
```

```
    if paciente.estado == "error_registro" or paciente.estado == "registro_cancelado":
```

```
        logger.warning(f"Paciente {paciente.id} [SIMULACION]: Flujo detenido debido a {paciente.estado}.")
```

```
        return
```

```
    # 2. Triage (CPU bound - ejecutado en ProcessPoolExecutor)
```

```
    logger.info(f"Paciente {paciente.id} [TRIAGE]: Iniciando triaje...")
```

```
    loop = asyncio.get_running_loop()
```

```
    try:
```

```
        paciente.prioridad = await loop.run_in_executor(
```

```
            cpu_executor, clasificar_prioridad, paciente.sintomas
```

```
        )
```

```
        logger.info(f"Paciente {paciente.id} [TRIAGE]: Prioridad {paciente.prioridad}")
```

```
        await actualizar_estadistica_global('triage')
```

```
    except Exception as e:
```

```
        # ... (manejo de errores)
```

```
        return
```

```
    # 3. Diagnóstico (CPU bound - ejecutado en ProcessPoolExecutor)
```

```
    logger.info(f"Paciente {paciente.id} [DIAGNOSTICO]: Iniciando diagnóstico...")
```

```
    try:
```

```

    paciente = await loop.run_in_executor(
        cpu_executor, diagnosticar_paciente_sincrono, paciente
    )
    if paciente.estado == "diagnosticado":
        await actualizar_estadistica_global('diagnostico')
    # ... (manejo de estado y errores)

except Exception as e:
    # ... (manejo de errores del executor)

# 4. Asignación de Cama y Tratamiento (Recurso limitado con asyncio.Semaphore)
recibio_cama = False
if paciente.estado not in ["error_registro", "error_triage", "flujo_cancelado"]:
    try:
        recibio_cama = await asignar_cama_async(paciente,
            actualizar_estadistica_global)
    except Exception as e:
        # ... (manejo de errores)
        recibio_cama = False

# 5. Seguimiento (I/O bound simulado)
if paciente.estado not in ["alta", "error_registro", "error_triage", "flujo_cancelado",
    "cama_cancelada", "error_cama", "error_ejecutor_cama"]:
    logger.info(f"Paciente {paciente.id} [SEGUIMIENTO]: Iniciando etapa de
        seguimiento...")
    try:
        await seguimiento_paciente(paciente, actualizar_estadistica_global,
            recibio_cama)
    except Exception as e:
        # ... (manejo de errores)

```


6. Finalización del Flujo del Paciente

... (log de finalización)

except asyncio.CancelledError:

... (manejo de cancelación)

except Exception as e:

... (manejo de errores desconocidos)

Explicación: Esta función asíncrona define el camino que sigue cada paciente. Utiliza `await` para esperar la completación de tareas asíncronas (registro, cama, seguimiento) y `await loop.run_in_executor` para ejecutar tareas síncronas que consumen CPU (triage, diagnóstico) en un `ProcessPoolExecutor` separado, evitando bloquear el bucle de eventos de `asyncio`.

2. Simulación de Recurso Limitado (Camas - asignacion_recursos.py)

La función `asignar_cama_async` demuestra cómo se simula un recurso limitado (camas) utilizando un semáforo asíncrono (`asyncio.Semaphore`).

```
asignacion_recursos.py > asignar_cama_async
5
6 logger = logging.getLogger(__name__)
7
8 camas_disponibles = asyncio.Semaphore(3)
9
10 async def asignar_cama_async(paciente, actualizar_estadisticas_func):
11     logger.info(f"Paciente {paciente.id} [CAMA]: Esperando cama disponible...")
12
13     try:
14         # Intentar adquirir el semáforo asíncrono
15         async with camas_disponibles:
16             logger.info(f"Paciente {paciente.id} [CAMA]: Cama asignada - Iniciando tratamiento")
17
18             # Registrar cama asignada
19             await actualizar_estadisticas_func('cama_asignada')
20
21             # Simular tiempo de tratamiento
22             tiempo_tratamiento = random.uniform(2, 5)
23             await asyncio.sleep(tiempo_tratamiento) # Usar await asyncio.sleep
24
25             logger.info(f"Paciente {paciente.id} [TRATAMIENTO]: Tratamiento completado en {tiempo_tratamiento:.1f}s")
26             paciente.estado = "tratamiento_completado" # Nuevo estado para indicar fin del tratamiento
27             return True # Indica que sí recibió cama
```

Explicación: El `asyncio.Semaphore(3)` crea un semáforo con un contador inicial de 3, representando 3 camas disponibles. La sentencia `async with camas_disponibles:` intenta adquirir una "licencia" del semáforo. Si el contador es mayor que cero, se adquiere instantáneamente y el contador disminuye. Si es cero, la tarea `async with` espera hasta que otra tarea libere una licencia.

3. Integración de Modelos ML (Triage - main.py y triage_ia.py)

La simulación integra modelos pre-entrenados para tareas de triaje y diagnóstico. La ejecución de estos modelos se realiza en un `ProcessPoolExecutor` porque son operaciones que consumen CPU y no deberían bloquear el bucle de eventos.

```
129 # 2. Triage (CPU rápido)
130 logger.info(f"Paciente {paciente.id} [TRIAGE]: Iniciando triaje...")
131 loop = asyncio.get_running_loop()
132 try:
133     paciente.prioridad = await loop.run_in_executor(
134         cpu_executor, clasificar_prioridad, paciente.sintomas
135     )
136     logger.info(f"Paciente {paciente.id} [TRIAGE]: Prioridad {paciente.prioridad}")
137     await actualizar_estadistica_global('triage')
138 except Exception as e:
139     logger.error(f"Paciente {paciente.id} [ERROR_TRIAGE]: Error en triaje: {type(e).__name__} - {e}", exc_info=True)
140     paciente.estado = "error_triage"
141     await actualizar_estadistica_global('error_triage')
142     logger.warning(f"Paciente {paciente.id} [SIMULACION]: Flujo detenido debido a error en triaje.")
143     return
144
```

```
22 def clasificar_prioridad(sintomas_dict):
23     sintomas = [
24         sintomas_dict.get("fiebre", 0),
25         sintomas_dict.get("tos", 0),
26         sintomas_dict.get("dolor", 0),
27         sintomas_dict.get("fatiga", 0),
28         sintomas_dict.get("respirar", 0),
29     ]

```

Explicación: `triage_ia.py` contiene la función síncrona `clasificar_prioridad` que carga un modelo de Decision Tree (entrenado previamente en `modelo_entrenamiento.py`) y realiza una predicción basada en los síntomas.

En `main.py`, `await loop.run_in_executor(cpu_executor, ...)` es crucial. Envía la ejecución de la función síncrona `clasificar_prioridad` (junto con los argumentos) al `ProcessPoolExecutor` (`cpu_executor`), el cual maneja un *pool* de procesos separados. Esto permite que la tarea de triaje, que podría ser intensiva en CPU, se ejecute en otro proceso sin detener el bucle de eventos principal de `asyncio`. La misma lógica se aplica para la etapa de diagnóstico.

4. Lógica de Seguimiento y Alta (Modificada - seguimiento.py)

Para asegurar que todos los pacientes que pasan por seguimiento eventualmente obtengan el alta, modificamos la lógica de la etapa "requiere_observacion"

```
10 async def seguimiento_paciente(paciente, actualizar_estadisticas_func, recibio_cama=False):
11     if paciente.estado in ['alta', 'error_registro', 'error_diagnostico', 'error_cama', 'seguimiento_cancelado']:
12         return
13
14     logger.info(f"Paciente {paciente.id} [SEGUIMIENTO]: Iniciando seguimiento...")
15
16     try:
17         # Simular latencia de red
18         tiempo_latencia = random.uniform(1, 3)
19         await asyncio.sleep(tiempo_latencia)
20
21         resultados = ["estable", "mejorando", "requiere_observacion"]
22         pesos = [0.4, 0.4, 0.2]
23         resultado = random.choices(resultados, weights=pesos, k=1)[0]
24
25         logger.info(f"Paciente {paciente.id} [SEGUIMIENTO]: Resultado: {resultado} ((tiempo_latencia:.1f)s)")
26
```

```

27 # Actualizar estado y registrar alta si corresponde
28 if resultado != "requiere_observacion":
29     paciente.estado = "alta"
30     logger.info(f"Paciente {paciente.id} [ALTA]: Alta médica completada")
31     await actualizar_estadisticas_func('alta')
32     if not recibio_cama:
33         await actualizar_estadisticas_func('alta_sin_cama')
34 else:
35     # --- INICIO DE MODIFICACIÓN SUGERIDA ---
36     paciente.estado = "observacion"
37     logger.info(f"Paciente {paciente.id} [SEGUIMIENTO]: Paciente requiere observación adicional.")
38     await actualizar_estadisticas_func('observacion') # Opcional: registrar cuántos pasan a observación
39
40     # Simular período de observación adicional
41     tiempo_observacion = random.uniform(1, 3) # Tiempo adicional en observación
42     logger.info(f"Paciente {paciente.id} [SEGUIMIENTO]: Iniciando período de observación ({tiempo_observacion:.1f}s)...")
43     await asyncio.sleep(tiempo_observacion)
44

```

Explicación: En la versión original, "requiere_observacion" era un estado final para el paciente en la simulación. Con la modificación, cuando el resultado del seguimiento es "requiere_observacion", el paciente pasa a un estado intermedio ('observacion'), simula un tiempo adicional (await asyncio.sleep), y luego automáticamente transita al estado 'alta'. Esto asegura que la tarea asíncrona de cada paciente (flujo_paciente_async) termine con el estado 'alta' si llega a la etapa de seguimiento sin errores graves.

Relación con la Arquitectura

- **AsyncIO:** Usado en asignar_cama_async y registrar_paciente_async para I/O bound.
- **OOP:** Encapsulación en Paciente y modelos ML (diagnostico_ia.py).
- **Funcional:** triage_ia.py evita efectos secundarios.
- **Concurrencia:** Semáforos para recursos y ProcessPoolExecutor para CPU.

Pruebas y Rendimiento

Se realizó una ejecución de la simulación con 4 pacientes.

Comando Ejecutado:

```

PS C:\Users\ferna\Desktop\Hospital_Sim> python main.py 4


```


Salida Completa de la Simulación:


```


03:29:44,873 [INFO] === SIMULACIÓN HOSPITALARIA INICIADA ===
03:29:44,874 [INFO] Pacientes a simular: 4
03:29:44,874 [INFO] =====
03:29:44,875 [INFO] Paciente 1 [SIMULACION]: Paciente llega al hospital.
03:29:44,875 [INFO] Paciente 1 [SIMULACION]: Iniciando flujo...
03:29:45,304 [INFO] Paciente 2 [SIMULACION]: Paciente llega al hospital.
03:29:45,304 [INFO] Paciente 2 [SIMULACION]: Iniciando flujo...


```


03:29:45,519  [INFO] Paciente 3 [SIMULACION]: Paciente llega al hospital.


03:29:45,519  [INFO] Paciente 3 [SIMULACION]: Iniciando flujo...


03:29:45,768  [INFO] Paciente 4 [SIMULACION]: Paciente llega al hospital.


03:29:45,768  [INFO] Paciente 4 [SIMULACION]: Iniciando flujo...


03:29:45,983  [INFO] Paciente 1 [REGISTRO]: Registrado en 1.11s | Avg: 1.11s


03:29:45,983  [INFO] Paciente 1 [TRIAGE]: Iniciando triaje...


03:29:46,422  [INFO] Paciente 3 [REGISTRO]: Registrado en 0.90s | Avg: 1.01s


03:29:46,422  [INFO] Paciente 3 [TRIAGE]: Iniciando triaje...


03:29:46,674  [INFO] Paciente 2 [REGISTRO]: Registrado en 1.37s | Avg: 1.13s


03:29:46,674  [INFO] Paciente 2 [TRIAGE]: Iniciando triaje...


03:29:47,065  [INFO] Paciente 4 [REGISTRO]: Registrado en 1.30s | Avg: 1.17s


03:29:47,067  [INFO] Paciente 4 [TRIAGE]: Iniciando triaje...


03:29:48,221  [INFO] Paciente 1 [TRIAGE]: Prioridad Baja


03:29:48,222  [INFO] Paciente 1 [DIAGNOSTICO]: Iniciando diagnóstico...


03:29:48,223  [INFO] Paciente 3 [TRIAGE]: Prioridad Media


03:29:48,223  [INFO] Paciente 3 [DIAGNOSTICO]: Iniciando diagnóstico...


03:29:48,224  [INFO] Paciente 2 [TRIAGE]: Prioridad Alta


03:29:48,224  [INFO] Paciente 2 [DIAGNOSTICO]: Iniciando diagnóstico...


03:29:48,225  [INFO] Paciente 4 [TRIAGE]: Prioridad Media


03:29:48,225  [INFO] Paciente 4 [DIAGNOSTICO]: Iniciando diagnóstico...


03:29:48,227  [INFO] Paciente 1 [CAMA]: Esperando cama disponible...


03:29:48,227  [INFO] Paciente 1 [CAMA]: Cama asignada - Iniciando tratamiento

03:29:48,228  [INFO] Paciente 3 [CAMA]: Esperando cama disponible...

03:29:48,228  [INFO] Paciente 3 [CAMA]: Cama asignada - Iniciando tratamiento

03:29:48,229  [INFO] Paciente 2 [CAMA]: Esperando cama disponible...

03:29:48,230  [INFO] Paciente 2 [CAMA]: Cama asignada - Iniciando tratamiento

03:29:48,231  [INFO] Paciente 4 [CAMA]: Esperando cama disponible...

03:29:50,680  [INFO] Paciente 3 [TRATAMIENTO]: Tratamiento completado en 2.4s

03:29:50,681  [INFO] Paciente 3 [SEGUIMIENTO]: Iniciando etapa de seguimiento...

03:29:50,682  [INFO] Paciente 3 [SEGUIMIENTO]: Iniciando seguimiento...

03:29:50,683  [INFO] Paciente 4 [CAMA]: Cama asignada - Iniciando tratamiento

03:29:52,039  [INFO] Paciente 1 [TRATAMIENTO]: Tratamiento completado en 3.8s

03:29:52,039  [INFO] Paciente 1 [SEGUIMIENTO]: Iniciando etapa de seguimiento...


03:29:52,040  [INFO] Paciente 1 [SEGUIMIENTO]: Iniciando seguimiento...

03:29:52,819  [INFO] Paciente 3 [SEGUIMIENTO]: Resultado: mejorando (2.1s)

03:29:52,819  [INFO] Paciente 3 [ALTA]: Alta médica completada

03:29:52,820  [INFO] Paciente 3 [SIMULACION]: Flujo completado con estado final 'alta'.

03:29:52,882  [INFO] Paciente 2 [TRATAMIENTO]: Tratamiento completado en 4.6s

03:29:52,882  [INFO] Paciente 2 [SEGUIMIENTO]: Iniciando etapa de seguimiento...

03:29:52,883  [INFO] Paciente 2 [SEGUIMIENTO]: Iniciando seguimiento...

03:29:53,928  [INFO] Paciente 2 [SEGUIMIENTO]: Resultado: estable (1.0s)

03:29:53,929  [INFO] Paciente 2 [ALTA]: Alta médica completada


03:29:53,929  [INFO] Paciente 2 [SIMULACION]: Flujo completado con estado final 'alta'.

03:29:54,007  [INFO] Paciente 1 [SEGUIMIENTO]: Resultado: estable (2.0s)

03:29:54,007  [INFO] Paciente 1 [ALTA]: Alta médica completada


03:29:54,007  [INFO] Paciente 1 [SIMULACION]: Flujo completado con estado final 'alta'.


03:29:54,883  [INFO] Paciente 4 [TRATAMIENTO]: Tratamiento completado en 4.2s

03:29:54,883  [INFO] Paciente 4 [SEGUIMIENTO]: Iniciando etapa de seguimiento...


03:29:54,884  [INFO] Paciente 4 [SEGUIMIENTO]: Iniciando seguimiento...


03:29:56,395  [INFO] Paciente 4 [SEGUIMIENTO]: Resultado: requiere_observacion (1.5s)

03:29:56,395  [INFO] Paciente 4 [SEGUIMIENTO]: Paciente requiere observación adicional.

03:29:56,396  [INFO] Paciente 4 [SEGUIMIENTO]: Iniciando período de observación (2.8s)...

03:29:59,217  [INFO] Paciente 4 [ALTA]: Observación completada, alta médica.

03:29:59,218  [INFO] Paciente 4 [SIMULACION]: Flujo completado con estado final 'alta'.

03:29:59,453  [INFO]

=== SIMULACIÓN COMPLETADA ===



=====

ESTADÍSTICAS FINALES DETALLADAS

=====

Total registrados:	4
Procesados en triage:	4
Diagnosticados:	4
Pacientes que recibieron cama:	4
Pacientes con alta médica:	4
Altas sin cama asignada:	0
Errores durante registro:	0
Errores durante diagnóstico:	0
Errores durante cama:	0
Errores durante seguimiento:	0
Pacientes con errores/cancelados:	0

=====

03:29:59,456  [INFO]  Tiempo total del proceso: 14.58 segundos

Análisis de Resultados:

Total Pacientes Simulados: 4

Pacientes que Iniciaron Flujo: 4

Pacientes por Etapa:

1. Registrados: 4

2. Procesados en Triage: 4
3. Diagnosticados: 4
4. Recibieron Cama: 4

Estados Finales:

- Alta Médica: 4
- Observación: 0 (Todos transitaron a alta tras observación si fue necesario)
- Errores/Cancelados: 0

Tiempo Total de Simulación: 14.58 segundos

Interpretación del Rendimiento:

La simulación con 4 pacientes y la lógica de alta tras observación demuestra la ejecución concurrente de tareas y la correcta aplicación de la modificación.

- Los pacientes progresan a través de las etapas de registro, triaje, diagnóstico y asignación de cama/tratamiento de forma concurrente, como se esperaba con el uso de asyncio y ProcessPoolExecutor.
- La etapa de seguimiento ahora asegura que incluso si un paciente inicialmente "requiere_observacion" (como fue el caso del Paciente 4), se simula un período adicional y luego se transita al estado 'alta'. Esto se confirma en la salida del log y en las estadísticas finales, donde los 4 pacientes reciben el alta médica.
- El tiempo total de simulación (14.58 segundos para 4 pacientes) es ligeramente mayor que en la ejecución anterior (12.71 segundos), lo cual es esperable y valida la simulación del tiempo adicional para la observación del Paciente 4 antes de su alta final.
- No se registraron errores ni cancelaciones en esta ejecución, lo que indica un flujo completo y exitoso para todos los pacientes dentro de los escenarios simulados.

Resultados con mayor cantidad de pacientes:

Se realizó una ejecución de la simulación con 10 pacientes

```
=== SIMULACIÓN COMPLETADA ===

=====
ESTADÍSTICAS FINALES DETALLADAS
=====
| Total registrados:          10 |
| Procesados en triage:      10 |
| Diagnosticados:           10 |
| Pacientes que recibieron cama: 10 |
| Pacientes con alta médica:  10 |
| Altas sin cama asignada:    0 |
| Errores durante registro:   0 |
| Errores durante diagnóstico: 0 |
| Errores durante cama:      0 |
| Errores durante seguimiento: 0 |
| Pacientes con errores/cancelados: 0 |
=====
03:49:19,753 [INFO] ⚙️Tiempo total del proceso: 19.46 segundos
```

Se realizó una ejecución de la simulación con 30 pacientes

```
==== SIMULACIÓN COMPLETADA ====

=====
ESTADÍSTICAS FINALES DETALLADAS
=====
| Total registrados:                30 |
| Procesados en triage:             30 |
| Diagnosticados:                   30 |
| Pacientes que recibieron cama:    30 |
| Pacientes con alta médica:        30 |
| Altas sin cama asignada:          0  |
| Errores durante registro:         0  |
| Errores durante diagnóstico:      0  |
| Errores durante cama:             0  |
| Errores durante seguimiento:      0  |
| Pacientes con errores/cancelados: 0  |
=====
03:51:21,124 [INFO] ⚡Tiempo total del proceso: 40.76 segundos
```

Conclusión:

La simulación demuestra eficazmente el uso de herramientas de concurrencia en Python (asyncio, ProcessPoolExecutor) para modelar un sistema complejo con diferentes tipos de tareas (I/O vs CPU bound) y recursos limitados.

La integración de modelos de ML se realiza de forma que no degrada el rendimiento del bucle principal. La modificación en la etapa de seguimiento garantiza que el modelo simule un proceso más realista donde los pacientes bajo observación eventualmente reciben el alta, lo cual se valida con los resultados de la ejecución mostrada. La flexibilidad del modelo permite ajustar parámetros para analizar cuellos de botella y el impacto de los recursos en el flujo de pacientes bajo diferentes escenarios.

Referencias:

- Durante el desarrollo de esta práctica, se utilizó una Inteligencia Artificial Gemini, como herramienta auxiliar para diversos fines, siempre de forma ética y documentada. La interacción se llevó a cabo principalmente entre el [Fecha de inicio de la conversación, ej. 01/05/2025].
- OpenAI. (2023). ChatGPT-4 [Modelo de lenguaje avanzado]. <https://chat.openai.com> Consultado durante el periodo de desarrollo del proyecto (2025):
 - Error en asyncio.Semaphore: Se usó para depurar un bloqueo en la asignación de camas (asignacion_recursos.py).
 - Sugerencia de diseño: Patrón Singleton para el logger centralizado (visualizacion.py).