CEThreads: Simulación de Tráfico con Calendarización de Hilos en Tiempo Real

Manuel Emilio Alfaro Mayorga Email: manalfaro@estudiantec.cr
Henry Daniel Nuñez Perez Email: henunez@estudiantec.cr
María Nicole Valverde Jimenez Email: marvalverde@estudiantec.cr
Kun Kin Zheng Liang Email: kzheng@estudiantec.cr
Escuela de Ingeniería en Computadores
Instituto Tecnológico de Costa Rica

***Abstract*—This project involves the development of a custom user-space threading library called *CEthreads*, implemented in the C programming language to replicate basic functionalities of `Pthreads`. The library provides primitives for thread creation, synchronization, and termination, as well as mutual exclusion mechanisms using mutexes. To visualize and validate the functionality, a graphical simulator was built using Python, depicting a bidirectional road with alternating traffic flow, where each vehicle is modeled as a thread. The simulation supports multiple scheduling algorithms (Round Robin, FCFS, SJF, Priority, and Real-Time) and flow control methods (Fairness, Sign-based, and FIFO), all configurable by the user through a settings file. The system was tested on a Linux environment, integrating the backend C library with the Python-based graphical interface via separate processes and shared files. A key insight from the project was that despite Python's limitations in native multithreading within graphical environments, a functional and responsive integration can be achieved through modular design and proper synchronization. This highlights the relevance of abstracting operating system resources effectively in educational simulations and lightweight concurrent systems.**

***Palabras clave*—CEthreads, Calendar Scheduler, server, Street-Manager**

I. Introducción

En el ámbito de los sistemas operativos, la programación concurrente y la planificación de tareas son componentes fundamentales para el diseño eficiente de software moderno [1], [2]. El uso de hilos permite dividir la ejecución de un programa en múltiples unidades de trabajo que pueden ejecutarse simultáneamente, optimizando así el uso del procesador y mejorando la capacidad de respuesta de los sistemas. Las bibliotecas de hilos, como `Pthreads`, proporcionan una interfaz estandarizada para gestionar estos procesos concurrentes; sin embargo, su comprensión profunda requiere más que su uso directo: exige una reimplementación desde cero para internalizar sus mecanismos [2].

Este proyecto tiene como objetivo principal desarrollar una biblioteca propia de hilos, llamada *CEthreads*, implementada en lenguaje C y operando completamente en espacio de usuario. La biblioteca incluye funciones esenciales para la creación y gestión de hilos, así como mecanismos de sincronización mediante mutexes. Para evidenciar su funcionalidad, se diseñó una interfaz gráfica en Python que simula una carretera por la que transitan vehículos representados como hilos, utilizando diversos algoritmos de planificación y control de flujo [3].

A lo largo de este documento se detallará el ambiente de desarrollo, los atributos de aprendizaje reforzados, el diseño del sistema, las instrucciones de uso y las conclusiones derivadas de la implementación, destacando los retos enfrentados y las estrategias empleadas para superarlos.

II. Ambiente de Desarrollo

Esta sección describe todo lo que se necesita para desarrollar y ejecutar el proyecto, desde el sistema operativo hasta las herramientas específicas que se utilizaron.

El proyecto *Scheduling Cars* fue desarrollado sobre un sistema operativo basado en Linux, específicamente utilizando la distribución Ubuntu 22.04 LTS. Este entorno permite el manejo eficiente de hilos en espacio de usuario, así como el uso de bibliotecas estándar del lenguaje C sin depender de bibliotecas externas, cumpliendo así con los requisitos establecidos.

El desarrollo se realizó utilizando el lenguaje de programación C, bajo el estándar C99, y empleando el compilador `GCC` en su versión 11.4.0. Para gestionar la construcción del proyecto, se utilizó `CMake`, el cual permite definir de forma clara y estructurada los archivos fuente, pruebas, módulos y dependencias del sistema.

Como entorno de desarrollo se utilizó `Visual Studio Code`, junto con extensiones específicas para el desarrollo en C y la integración con CMake. Además, se emplearon herramientas como `Valgrind` para la detección de errores de acceso a memoria y fugas, y `GDB` como depurador para analizar el comportamiento en tiempo de ejecución.

Para compilar y ejecutar el proyecto correctamente, se deben cumplir los siguientes requisitos en el sistema:

- **Sistema operativo:** Distribución Linux compatible (recomendado: Ubuntu 22.04 LTS)
- **Lenguaje C:** Compilador `GCC` versión 11 o superior
- **Sistema de construcción:** `CMake` versión 3.25 o superior
- **Herramientas recomendadas:**
  - `make`
  - `Valgrind` (opcional, para detección de errores de memoria)
  - `GDB` (opcional, para depuración)

Una vez instaladas las herramientas anteriores, se deben seguir los pasos de la Sección V.

III. Atributos

1) Indicar las estrategias para el trabajo individual y en equipo de forma equitativa e inclusiva en las etapas del proyecto (planificación, ejecución y evaluación).
   En el caso de la planificación, se hizo una reunión como equipo y establecimos los objetivos técnicos. Además, definimos la estructura base del proyecto y los lenguajes de programación que se iban a usar. En la ejecución, a cada miembro del equipo se le asignó responsabilidades específicas según la especialización de cada uno: quién realizaba la interfaz, quién realizaba los algoritmos de calendarización, quien hacía los algoritmos de flujo y quien creada los CEThreads. Finalmente, en la etapa de evaluación, entre todos se revisó los resultados en conjunto, así probando que el sistema cumpliera las funciones propuestas. Además, se comprobó que el codigo fuera legible y ordenado.
2) Indicar la planificación del trabajo mediante la identificación de roles, metas y reglas.
   Para la planificación a cada miembro del equipo se le estableció un rol. Un miembro realiza la interfaz, otro los algoritmos de calendarización, otro los CEThreads y el último los algoritmos de flujo. Además, en esta etapa se establecieron las metas, estas se centraron en construir un repositorio y crear el proyecto con su respectivo CMake para la ejecución del código en C. Finalmente, se establecieron las reglas de colaboración estas se basan en la comunicación constante y en la validación mutua de código, acordando que las decisiones se tomarían por consenso y priorizando la sencillez de implementación cuando surgieran dudas.
3) Indicar cuáles acciones promueven la colaboración entre los miembros del equipo durante el desarrollo del proyecto.
   Una de las acciones fue que se tuviera una comunicación consistente para esto se creó un grupo de WhatsApp para reportar avances. Además, que este nos permitió compartir problemas que surgieran en tiempo real y asegurar que ambos estuvieran al tanto de cada modificación. Asimismo, se realizaron reuniones virtuales periódicas para revisar el progreso, evaluar las tareas en seguimiento y discutir y ajustar la hoja de ruta cuando era necesario. Esta organizanizacion permitió una retroalimentación rápida, evitando cuellos de botella en el desarrollo.
4) Indicar cómo se ejecutan las estrategias planificadas para el logro de los objetivos.
   Para el logro de los objetivos, se intento cumplir con las metas una por una. Primero, se garantizó tener un CMake funcional. Luego, se integró como la parte de C y de la interfaz (Python) de la aplicación, prestando especial atención en la generación de carros. Además, para validar estos avances, se realizaron constantes pruebas con diferentes patrones, asegurando que la aplicación corriera sin fallos y que todo estuviera correctamente interconectado.
5) Indicar la evaluación para la el desempeño del trabajo individual y en equipo
   Para la evaluación se planteó pequeñas metas semanales para cada uno de los miembros del equipo, esto fue supervisado a través de repositorios de control de versiones. Además, esto permitió que se evaluará tanto el desempeño individual, esto por la calidad y puntualidad en sus contribuciones, como la eficacia del equipo esto al observar si las funcionalidades del proyecto estaban completas.
6) Indicar la evaluación para las estrategias utilizadas de equidad e inclusión.
   Para promover la equidad, las responsabilidades se asignaron conforme a las fortalezas de cada uno, maximizando su aporte y motivación. Además, toda decisión se tomó tras escuchar los argumentos de ambas partes, y al haber desacuerdos, se daba preferencia a la solución más simple o a la que implicara menor complejidad de mantenimiento. A nivel de accesibilidad, se mantuvieron los colores de alto contraste (verde, rojo, amarillo, azul y morado) indicados en la especificación sobre un fondo negro para que resultaran distinguibles incluso por usuarios con daltonismo.
7) Indicar la evaluación para las acciones de colaboración entre los miembros del equipo.
   La principal herramienta para este proyecto para medir la colaboración fue un repositorio en GitHub, en este se crearon ramas, una para cada uno de los requerimientos del proyecto, esto permitió que cada miembro del equipo pudiera trabajar de manera paralela. También, en esta herramienta se puede observar las estadísticas de commits y pull requests de cada uno de los miembros.

IV. Diseño

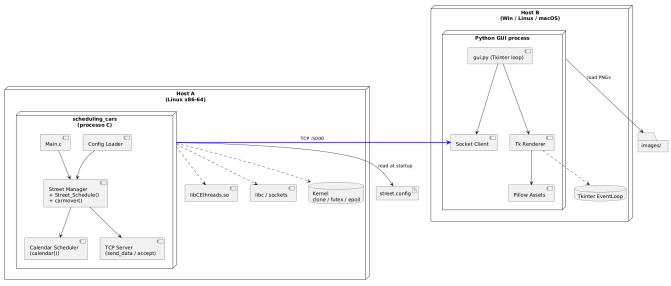

Figura 1. Diagrama de Arquitectura.

El diagrama de arquitectura que se muestra la Figura 1 sitúa la aplicación en su contexto físico y resulta esencial para comprender cómo se distribuye la carga entre procesos y máquinas. Allí se aprecia que el núcleo de la simulación corre como un binario en C dentro de un host Linux; ese proceso encapsula Main.c, el gestor de la calle, el planificador de hilos, el servidor TCP y el cargador de configuración. Todos ellos dependen de bibliotecas nativas, la capa de hilos ligera libCEthreads, la pila de sockets de libc y las primitivas de sincronización del propio kernel, lo que garantiza un control preciso sobre la concurrencia y la latencia. En un segundo host, la GUI en Python ejecuta su propio bucle de eventos y se comunica por medio de un socket TCP con el backend, recibiendo instantáneamente los estados que necesita para

redibujar la escena. La lectura del archivo street.config y la carga de sprites se hace de forma local al arrancar, de modo que la única interacción remota en tiempo de ejecución se da en ese canal TCP.

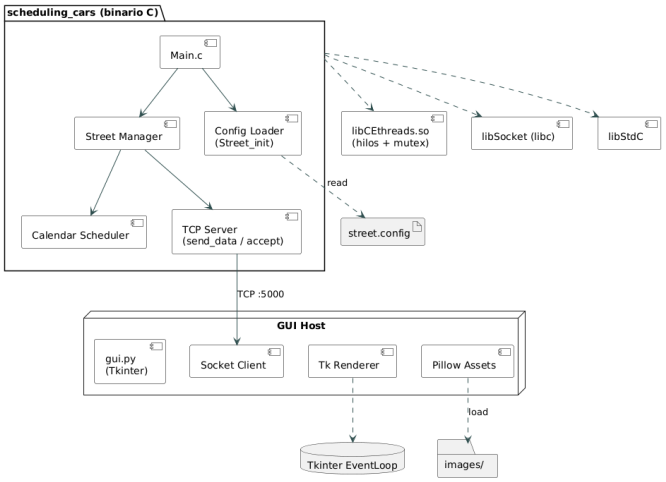

Figura 2. Diagrama de Componentes.

En la Figura 2 se presenta el diagrama de componentes que desciende un nivel y muestra cómo se agrupan las responsabilidades dentro de cada proceso. En el ejecutable C, Main.c funciona como coordinador y delega inmediatamente en cuatro módulos especializados: el Street Manager se encarga de las colas y del arreglo que modela la vía; el Calendar Scheduler alberga las políticas FCFS, SJF, Prioridad, Round–Robin y Tiempo Real; el TCP Server expone el estado hacia el exterior; y el Config Loader hace el enlace entre el archivo de parámetros y la estructura interna. Alrededor de ese núcleo se sitúan dependencias claras, como hilos, sockets y la biblioteca estándar, de manera que cualquier cambio en la lógica de negocio no afecta ni a la comunicación ni al threading. En la otra orilla, la GUI se descompone en cuatro piezas: el script gui.py que gobierna, un cliente de sockets que recibe los mensajes, un renderizador en Tkinter y un pequeño módulo que administra los recursos gráficos con Pillow; de esta forma, cada componente mantiene responsabilidades únicas y reemplazables.

Mientras tanto, el diagrama de clases de la Figura 3 muestra la anatomía exacta de los datos que circulan. Cada objeto Car posee su propio hilo de CEThreads. Los Car viven dentro de una instancia Street, que además rastrea la cantidad de vehículos en tránsito, la dirección de circulación, la luz amarilla y la bandera de cumplimiento de Tiempo Real; todo está pensado para que un único lugar concentre el estado crítico y así simplificar el bloqueo con mutex. A los costados se sitúan dos estructuras WaitLine, una por sentido, que almacenan los carros a la espera de entrar. El StreetManager actúa como fachada: recibe peticiones de la CLI o de la GUI, consulta al planificador, mueve los carros y finalmente orquesta el envío del estado hacia la interfaz. La independencia entre StreetManager y Calendar permite que se introduzcan nuevos algoritmos sin tocar el resto de la arquitectura, reforzando la extensibilidad.

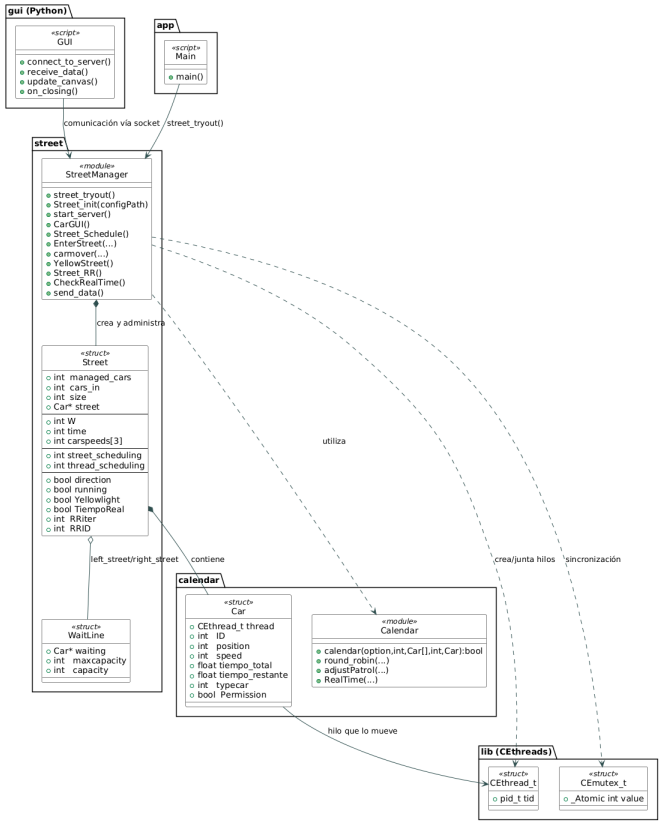

Figura 3. Diagrama UML.

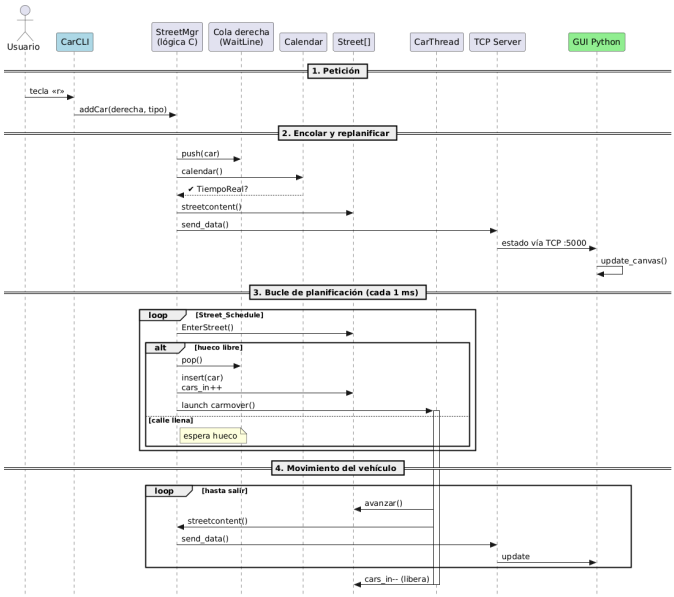

Figura 4. Diagrama de Secuencia.

Finalmente, la Figura 4 presenta el diagrama de secuencia donde ilustra la dinámica del caso de uso más representativo: la adición de un vehículo por el costado derecho. El flujo muestra cómo el comando introducido por el usuario viaja desde la línea de comandos hasta el gestor de la calle, quien primero encola el carro, luego invoca al planificador para recalcular prioridades y, de inmediato, publica el nuevo estado a través del servidor TCP. Ese mensaje llega a la GUI, que actualiza su lienzo sincrónicamente. Paralelamente, un bucle de un milisegundo intenta insertar el vehículo desde la cola al tramo crítico; si la posición inicial está libre, se lanza un hilo dedicado que avanza el coche y envía actualizaciones, manteniendo así la animación en tiempo real. Por último, el bucle alternativo "calle llena" muestra que la lógica trabaja sin bloquear la simulación como tal.

V. Instrucciones de cómo se utiliza el proyecto

A. Pasos de instalación y ejecución

1) **Clonar repositorio:**

```
git clone
    https://github.com/EmilioTec10/
Proyecto1-Operativos.git
cd Proyecto1-Operativos
```

2) **Configurar archivo `canal.config`:**

```
cd src/street/
nano canal.config # Editar con parametros
    deseados
```

Ejemplo de configuración:

```
length=5 # Longitud de la calle
c_schedule=1 # 1=Equidad, 2=Letrero,
    3=FIFO
t_schedule=1 # 1=FCFS, 2=SJF,
    3=Prioridad, 4=RR, 5=Tiempo Real
W=2 # Carros por ciclo (Equidad)
speed=1,2,4 # Velocidades [normal,
    deportivo, emergencia]
```

3) **Compilar proyecto:**

```
mkdir -p build && cd build
cmake ..
make
```

4) **Ejecutar servidor:**

```
cd bin
./Proyecto1-Operativos
```

5) **Lanzar interfaz gráfica (en terminal separada):**

```
cd build/bin
python3 gui.py
```

B. Interacción con el sistema

- **Comandos en el servidor:** Estos comandos se pueden visualizar en la Figura 5, los cuales permiten interactuar con la interfaz gráfica de la Figura 6.

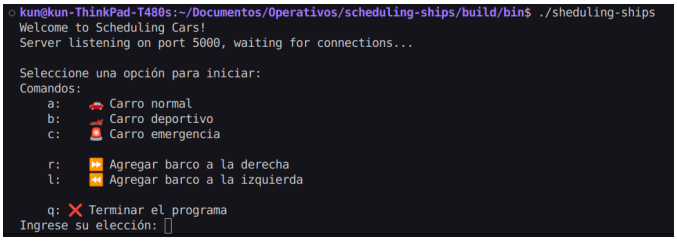

Figura 5. Comandos en consola para interacción con Scheduling Cars.

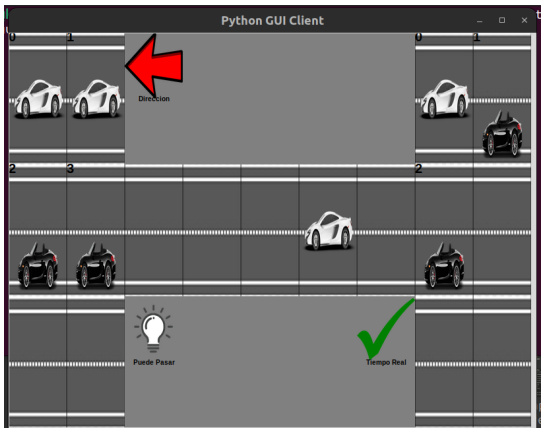

Figura 6. Interfaz gráfica de Scheduling Cars.

C. Solución de problemas

- **Error de compilación:**

```
cd build && make clean && make
```

- **Interfaz no responde:**
  - Verificar que el puerto 5000 esté disponible
  - Asegurar rutas correctas de imágenes en `images/`
- **Finalización elegante:**
  - Usar comando `q` en servidor
  - Cerrar ventanas en orden: interfaz primero, luego servidor

VI. Conclusiones

El desarrollo de Scheduling Cars demostró que es posible construir, exclusivamente en espacio de usuario, una biblioteca ligera de hilos capaz de exponer primitivas de creación y sincronización comparables a Pthreads, sin sacrificar el control fino sobre la planificación. La instrumentación de la calle bidireccional confirmó que el modelo de un vehículo = un hilo no sólo es viable en términos de rendimiento sino también pedagógicamente esclarecedor: la visualización permitió constatar, en tiempo real, los efectos de cada algoritmo de calendarización y las implicaciones de las decisiones de control de flujo en la latencia percibida.

Los resultados sugieren que la combinación de un backend nativo en C y una GUI en Python ofrece una relación costo-beneficio óptima para simulaciones docentes: el proceso C se encarga de la tarea intensiva en CPU y de la sincronización a nivel de kernel, mientras que la capa Python se limita a

renderizar, evitando los cuellos de botella habituales de la GIL. La comunicación asíncrona vía sockets resultó suficientemente rápida para mantener una tasa de refresco fluida, incluso con cientos de hilos concurrentes, y al mismo tiempo desacopla por completo lógica y presentación, lo que facilita portar la interfaz a otros entornos gráficos o incluso a la web.

En términos de diseño de software, la arquitectura por capas y la separación clara de responsabilidades que están reflejadas en los diagramas de componentes y clases, favorecieron la extensibilidad. Añadir un nuevo algoritmo de planificación o un modo de control de tráfico no implica reescribir código existente, sino incorporar un caso extra en el módulo Calendar o StreetManager. La experiencia valida el principio de sustitución abierta/cerrada: el núcleo permanece estable mientras los comportamientos se amplían mediante módulos intercambiables.

Por último, el proyecto pone de relieve el valor de abstraer los detalles del sistema operativo cuando se busca un aprendizaje significativo. Al replicar clone, futex y los mecanismos de espera activa, se comprendió la interacción entre espacio de usuario y kernel más allá de meros llamados de biblioteca. Esa comprensión se tradujo en la capacidad de depurar condiciones de carrera reales y de optimizar el uso de CPU sin recurrir a herramientas externas. En síntesis, Scheduling Cars no sólo alcanzó su objetivo funcional de simular tráfico multihilo, sino que también se consolidó como una plataforma didáctica que evidencia cómo principios de concurrencia, modularidad y portabilidad pueden converger en un sistema compacto y robusto.

VII. Sugerencias y recomendaciones

Un primer frente de mejora consiste en profundizar la instrumentación del sistema. Actualmente la métrica visible para el usuario es el flujo de vehículos y la bandera de “Tiempo Real”; agregar contadores de throughput, tiempos medios de espera por tipo de automóvil y estadísticas de utilización del tramo crítico permitiría cuantificar con mayor precisión el impacto de cada algoritmo de planificación. Una API REST o un exportador Prometheus aportarían datos históricos sin comprometer la ligereza de la visualización Tkinter.

En segundo lugar, se recomienda introducir pruebas de estrés automatizadas que ejecuten cientos o miles de hilos con parámetros aleatorios de velocidad, tamaño de la calle y modo de scheduling. Con ese banco de pruebas podría evaluarse la robustez de la biblioteca libCEthreads frente a situaciones límite (cascada de futex-wakeups, starvation o thrashing de caché). Complementar esta estrategia con sanitizers (ThreadSanitizer/AddressSanitizer) permitiría detectar condiciones de carrera difíciles de reproducir de forma manual.

Desde la perspectiva de usabilidad, convendría abstraer la GUI actual en un servicio Web basado en WebSockets. El backend ya entrega el estado en formato texto; bastaría encapsularlo en JSON y exponerlo a través de un gateway ligero en C o Python (FastAPI, Flask-SocketIO). Esto habilitaría un cliente multiplataforma, ya sea móvil o navegador, y simplificaría la incorporación de animaciones más ricas mediante bibliotecas JavaScript (p. ej. PixiJS o D3.js).

Otra línea a explorar es la adaptación del calendario de planificación a escenarios heterogéneos. El sistema soporta tipos de vehículos fijos, pero la estructura es extensible: se sugiere permitir que el archivo street.config declare dinámicamente nuevos “perfiles” (por ejemplo, ambulancias con prioridad incondicional, vehículos pesados con limitación de velocidad o transportes autónomos que se comuniquen entre sí). Ello transformaría la simulación en una plataforma válida para estudios de política pública o validación de algoritmos de tráfico inteligentes.

En cuanto a la portabilidad y despliegue, empaquetar el backend en un contenedor Docker con capacidades cgroup específicas facilitaría su ejecución en entornos controlados sin requerir privilegios de superusuario, a la vez que permitiría experimentar con distintos schedulers del kernel. Para la GUI, un entorno virtual preconfigurado o un requirements.txt garantizará que quienes revisen el proyecto lo pongan en marcha con un único comando.

Finalmente, para mantener la calidad a medida que el código crece, se aconseja formalizar un flujo de integración continua (CI) que compile el proyecto en modo release y debug, ejecute la suite de pruebas y genere automáticamente la documentación de los diagramas PlantUML. Con ello se asegurará que cada nueva funcionalidad respete las convenciones de diseño y no introduzca regresiones en el comportamiento observable.

Referencias

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. Wiley, 10 ed., 2020.
[2] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*. Pearson, 4 ed., 2014.
[3] Python Software Foundation, *Python 3.12 Documentation – tkinter and threading*. Python Software Foundation, 2024. Disponible en https://docs.python.org/3/library/.