Universidad San Francisco de Quito

Colegio de Ciencias e Ingenierías

Monitor de Recursos del Sistema

Proyecto Final - Sistemas Operativos

Integrantes:

- Julián León
- Jarod Tierra
- Andrés Vega

Profesor:
Daniel Riofrio

Semestre:
202510 - Primer Semestre 2025/2026

Fecha:
10 de Diciembre 2025

Índice

1. Introducción

1.1. Objetivos del Proyecto

El presente proyecto tiene como objetivo desarrollar una herramienta completa de monitoreo y visualización de recursos del sistema operativo, implementada en Python con interfaz gráfica. Esta herramienta permite monitorear en tiempo real:

- Utilización del CPU (procesador) y sus núcleos individuales
- Utilización de memoria RAM y Swap, incluyendo análisis de fragmentación
- Utilización del almacenamiento (disco duro), velocidad de I/O y fragmentación
- Procesos en ejecución con sus características detalladas
- Consumo de ancho de banda de la red (upload y download)

1.2. Alcance

El proyecto cumple con todos los requisitos especificados en la asignación, incluyendo:

- Aplicación gráfica desarrollada en Python utilizando PyQt5
- Monitoreo de todos los recursos solicitados
- Visualización temporal: estado actual y últimos 60 minutos
- Implementación de hilos para actualización independiente de componentes
- Funcionalidad de terminar procesos desde la aplicación
- **Puntos extra:** Análisis de fragmentación de memoria RAM y disco

1.3. Tecnologías Utilizadas

Tecnología	**Propósito**
Python 3.8+	Lenguaje de programación principal
PyQt5	Framework de interfaz gráfica
psutil	Obtención de información del sistema
matplotlib	Generación de gráficos históricos
threading	Manejo de hilos concurrentes
subprocess	Ejecución de comandos del sistema

Cuadro 1: Tecnologías utilizadas en el proyecto

2. Arquitectura del Sistema

2.1. Estructura General

El proyecto sigue una arquitectura de tres capas bien definida:

1. **Capa de Datos (Monitors):** Módulos que recopilan información del sistema
2. **Capa de Presentación (GUI):** Widgets que muestran la información al usuario
3. **Capa de Control:** Ventana principal que coordina la actualización

2.2. Organización de Archivos

```
ProyectoFinalSO/
|
+-- main.py                        # Punto de entrada
+-- requirements.txt               # Dependencias
+-- README.md                      # Documentacion
|
+-- monitors/                      # Modulos de monitoreo
|   +-- __init__.py
|   +-- cpu_monitor.py            # Monitor de CPU
|   +-- memory_monitor.py         # Monitor de memoria
|   +-- disk_monitor.py           # Monitor de disco
|   +-- process_monitor.py        # Monitor de procesos
|   +-- network_monitor.py        # Monitor de red
|
+-- gui/                           # Interfaz grafica
    +-- main_window.py           # Ventana principal
    +-- widgets/                 # Widgets personalizados
        +-- cpu_widget.py
        +-- memory_widget.py
        +-- disk_widget.py
        +-- process_widget.py
        +-- network_widget.py
```

2.3. Diagrama de Componentes

La aplicación se compone de los siguientes componentes principales:

- **MainWindow:** Coordina todos los widgets y timers
- **Monitors:** Clases que encapsulan la lógica de recopilación de datos
- **Widgets:** Componentes visuales que muestran información

- **Threads:** Hilos que actualizan datos en segundo plano

3. Uso de Hilos

3.1. Importancia de los Hilos

El uso de hilos es un requisito **crucial** en este proyecto. Los hilos permiten:

- Mantener la interfaz gráfica responsiva mientras se recopilan datos
- Actualizar diferentes componentes de forma independiente
- Evitar bloqueos durante operaciones de I/O
- Recopilar datos históricos continuamente en segundo plano

3.2. Implementación de Hilos

Cada monitor implementa su propio hilo de actualización utilizando la clase `Thread` del módulo `threading` de Python. La estructura general es:

Listing 1: Patrón de implementación de hilos

```
from threading import Thread, Lock
import time

class Monitor:
    def __init__(self):
        self._running = False
        self._thread = None
        self._lock = Lock()  # Para acceso thread-safe

    def _monitor_loop(self):
        """Loop que se ejecuta en segundo plano"""
        while self._running:
            self._update_history()
            time.sleep(self.update_interval)

    def start_monitoring(self):
        """Inicia el hilo de monitoreo"""
        if not self._running:
            self._running = True
            self._thread = Thread(
                target=self._monitor_loop,
                daemon=True
            )
```

```
            self._thread.start()

    def stop_monitoring(self):
        """Detiene el hilo"""
        self._running = False
        if self._thread:
            self._thread.join(timeout=2)
```

3.3. Hilos Implementados

Monitor	Archivo	Intervalo
CPU	cpu_monitor.py:215-226	1 segundo
Memoria	memory_monitor.py:221-233	1 segundo
Disco	disk_monitor.py:299-311	5 segundos
Red	network_monitor.py:211-223	1 segundo
Procesos	process_monitor.py:54-66	1 segundo

Cuadro 2: Hilos de monitoreo implementados

3.4. Sincronización Thread-Safe

Para evitar condiciones de carrera (race conditions), todos los monitores utilizan `Lock` para proteger el acceso a datos compartidos:

Listing 2: Ejemplo de acceso thread-safe

```
def get_history(self):
    """Acceso seguro al historial desde multiples hilos"""
    with self._lock:
        return {
            'timestamps': list(self.timestamps),
            'values': list(self.history)
        }
```

3.5. Timers de Qt

Además de los hilos de recopilación, la interfaz gráfica utiliza `QTimer` para actualizar los widgets sin bloquear la UI:

Listing 3: Timers en MainWindow (main_window.py:181-196)

```
# Timer rapido: CPU, Memoria, Red (1 segundo)
self.fast_timer = QTimer()
self.fast_timer.timeout.connect(self.update_fast_resources)
self.fast_timer.start(1000)

# Timer lento: Disco (5 segundos)
self.slow_timer = QTimer()
self.slow_timer.timeout.connect(self.update_slow_resources)
self.slow_timer.start(5000)

# Timer procesos (3 segundos)
self.process_timer = QTimer()
self.process_timer.timeout.connect(self.update_processes)
self.process_timer.start(3000)
```

4. Módulos Implementados

4.1. Monitor de CPU

4.1.1. Funcionalidades

- Porcentaje de uso total del CPU
- Uso individual por cada núcleo/core
- Frecuencia actual del procesador
- Load average (1, 5 y 15 minutos)
- Distribución de tiempo (usuario, sistema, inactivo)
- Gráfico histórico de los últimos 60 minutos

4.1.2. Captura de Pantalla

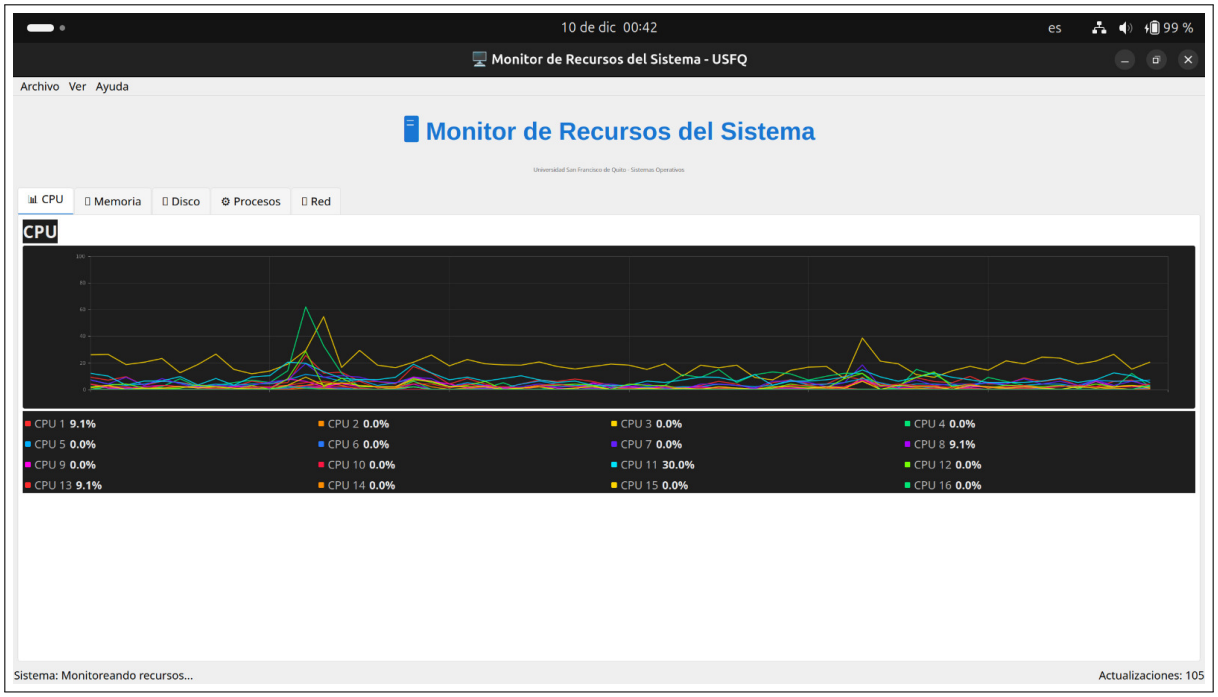


Figura 1: Interfaz del monitor de CPU mostrando uso por núcleo y gráfico histórico

4.1.3. Implementación Técnica

El monitor de CPU utiliza `psutil.cpu_percent()` para obtener el uso total y por núcleo. El historial se mantiene en una `deque` con capacidad para 3600 muestras (1 hora a 1 segundo por muestra).

4.2. Monitor de Memoria RAM

4.2.1. Funcionalidades

- Memoria total, disponible, usada y libre
- Uso de memoria Swap
- **Análisis de fragmentación de memoria (punto extra)**
- Gráfico histórico del uso de RAM y Swap

4.2.2. Capturas de Pantalla

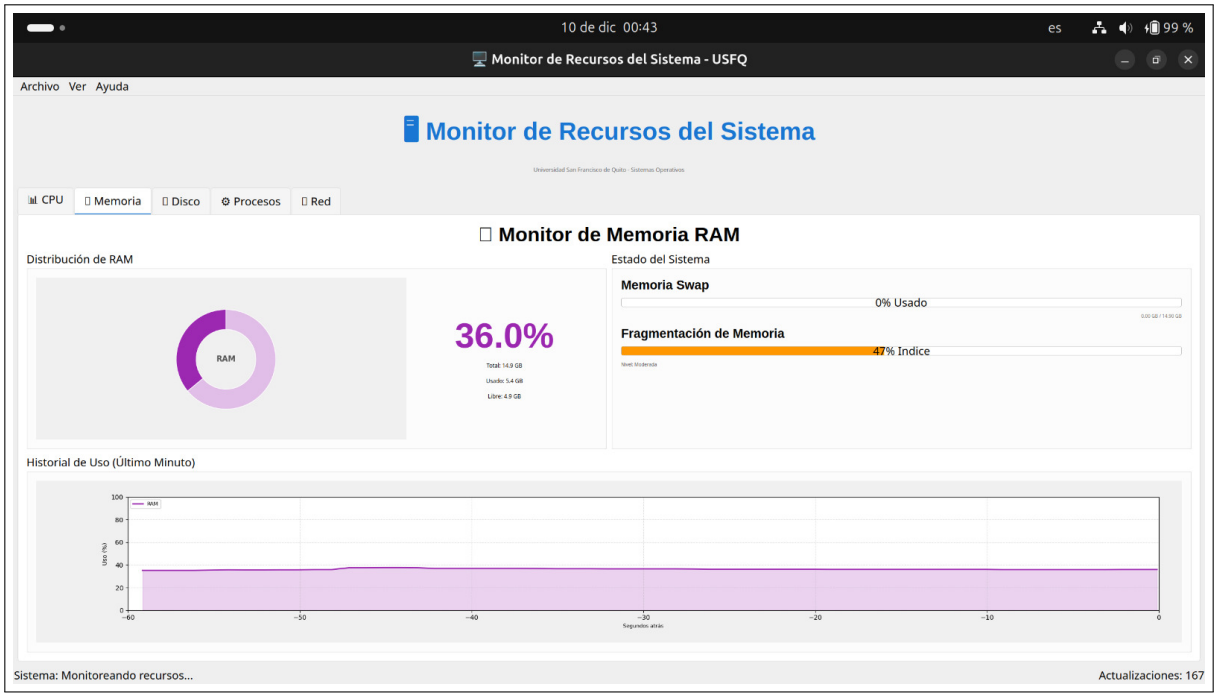


Figura 2: Monitor de memoria mostrando uso de RAM, Swap y gráficos históricos

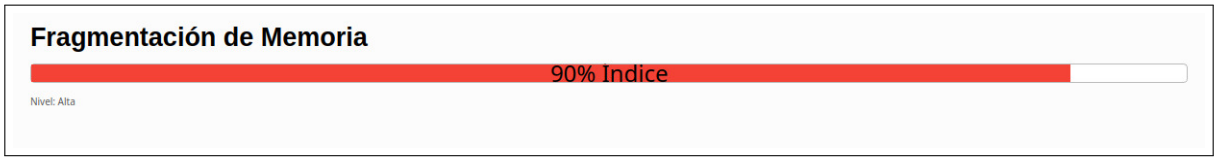


Figura 3: Análisis de fragmentación de memoria RAM (punto extra)

4.2.3. Análisis de Fragmentación (Punto Extra)

La fragmentación de memoria se analiza de forma específica según el sistema operativo:

- **Linux:** Lee `/proc/buddyinfo` para obtener información real de fragmentación
- **macOS:** Utiliza `vm_stat` para analizar páginas especulativas
- **Windows:** Estima basándose en memoria disponible vs. libre

El código se encuentra en `memory_monitor.py:79-183`.

4.3. Monitor de Almacenamiento

4.3.1. Funcionalidades

- Lista de todas las particiones del sistema
- Espacio total, usado y disponible por partición
- Velocidad de lectura y escritura (I/O)
- **Análisis de fragmentación de disco (punto extra)**
- Gráfico histórico de velocidad de I/O

4.3.2. Capturas de Pantalla

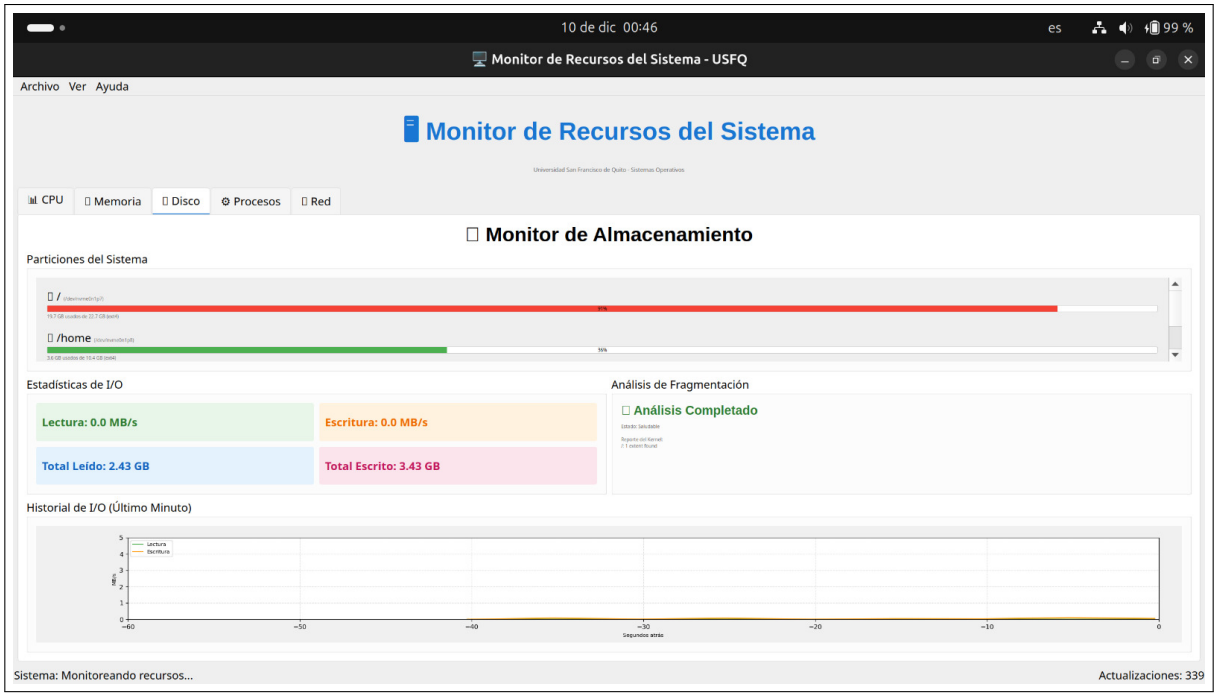


Figura 4: Monitor de disco mostrando particiones y velocidad de I/O

Figura 5: Información de fragmentación del disco (punto extra)

4.3.3. Análisis de Fragmentación (Punto Extra)

La fragmentación del disco se analiza según el sistema de archivos:

- **Linux ext4:** Utiliza `filefrag` y `dumpe2fs`
- **macOS APFS:** Utiliza `diskutil` (APFS no requiere desfragmentación)
- **Windows NTFS:** Utiliza `defrag /A`

El código se encuentra en `disk_monitor.py:153-262`.

4.4. Monitor de Procesos

4.4.1. Funcionalidades

- Lista completa de procesos en ejecución
- Información detallada: PID, nombre, estado, CPU %, memoria, disco
- Búsqueda y filtrado de procesos
- Ordenamiento por diferentes criterios
- **Capacidad de terminar procesos** (SIGTERM y SIGKILL)
- Vista detallada de cada proceso
- Gráfico histórico de cantidad de procesos activos

4.4.2. Capturas de Pantalla

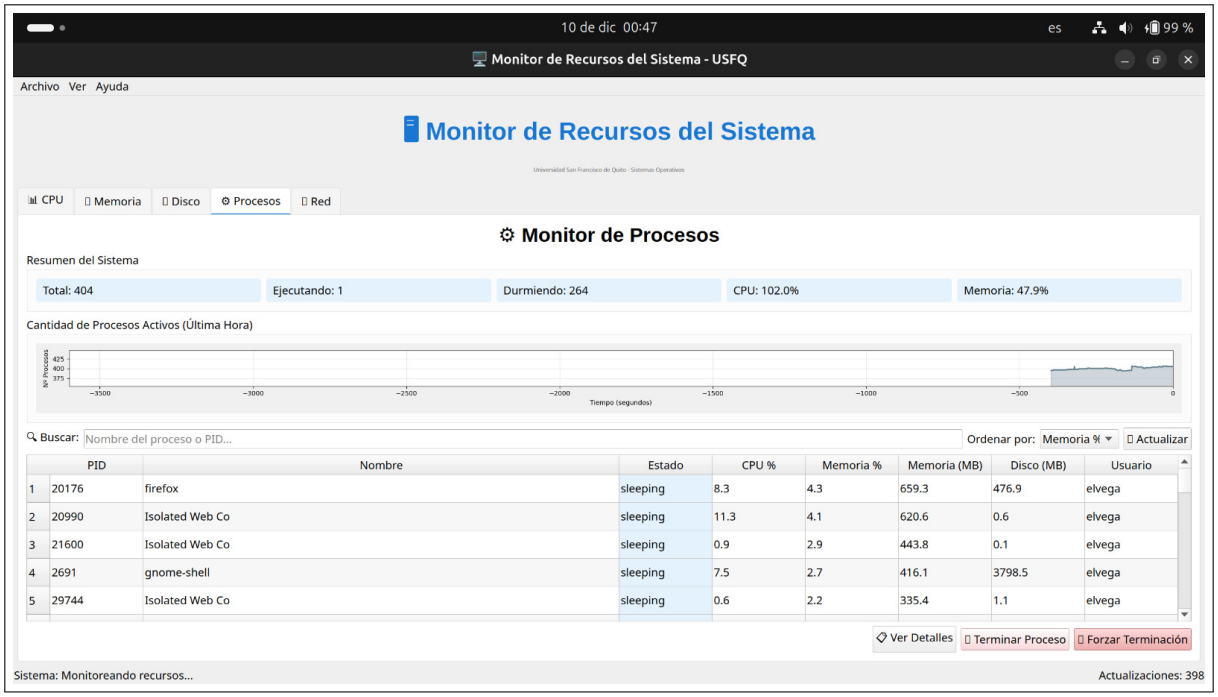


Figura 6: Monitor de procesos mostrando lista completa y gráfico histórico

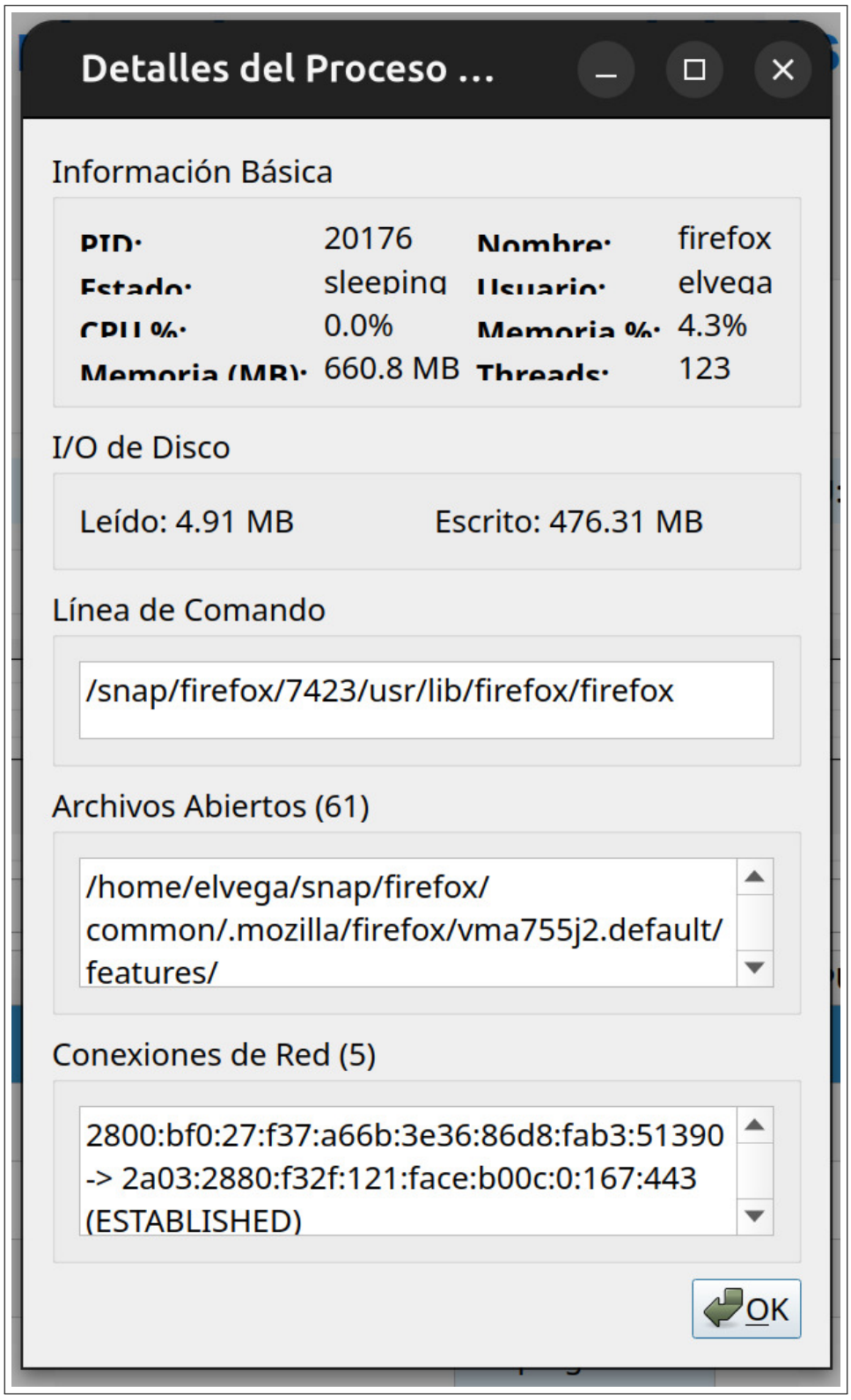


Figura 7: Vista detallada de un proceso individual

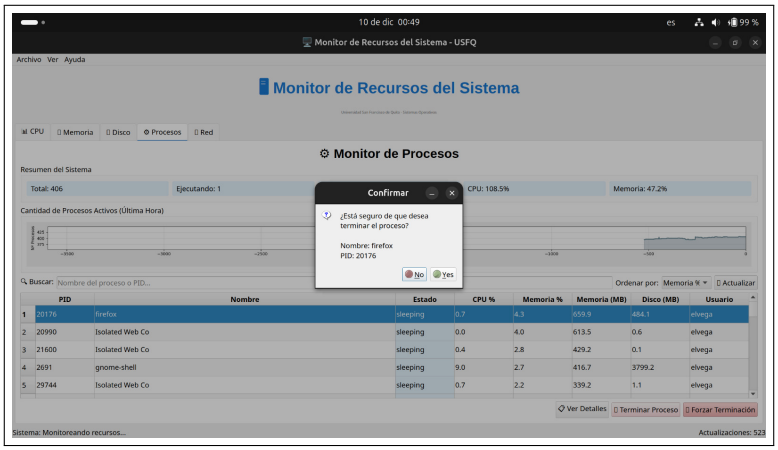


Figura 8: Diálogo de confirmación para terminar un proceso

4.4.3. Terminación de Procesos

La aplicación permite terminar procesos de dos formas:

1. **Terminación normal (SIGTERM):** Permite al proceso cerrar limpiamente
2. **Terminación forzada (SIGKILL):** Fuerza el cierre inmediato

Implementado en `process_monitor.py:265-322` y `process_widget.py:403-453`.

4.5. Monitor de Red

4.5.1. Funcionalidades

- Velocidad de descarga (download) en tiempo real
- Velocidad de subida (upload) en tiempo real
- Mostrados de forma **separada** (requisito del proyecto)
- Total de bytes enviados y recibidos
- Lista de interfaces de red
- Conexiones activas
- Gráfico histórico separado para upload y download

4.5.2. Capturas de Pantalla

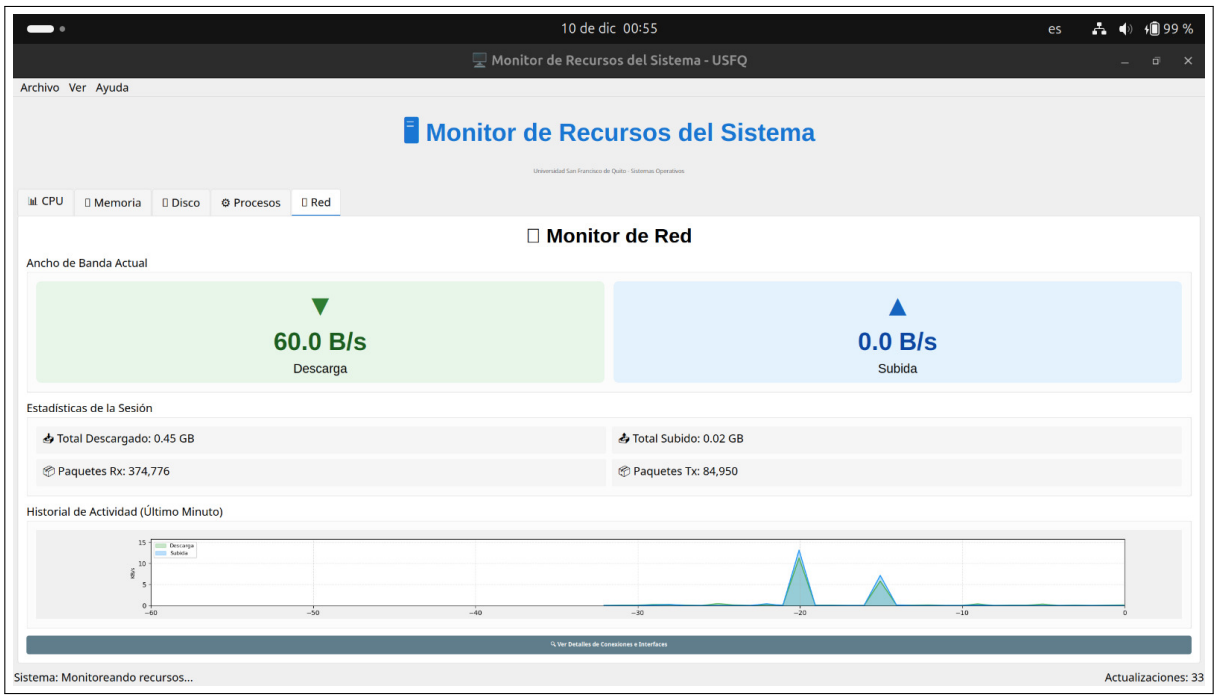


Figura 9: Monitor de red mostrando upload/download separados y gráficos históricos

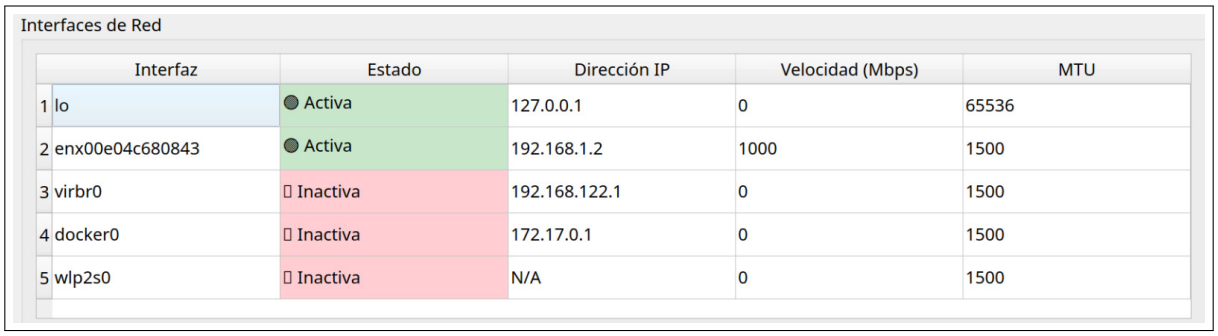


Figura 10: Lista de interfaces de red y conexiones activas

4.5.3. Separación de Upload y Download

Como se requiere en el proyecto, el ancho de banda de upload y download se muestra completamente separado:

- Velocidades mostradas en etiquetas independientes
- Gráficos separados en el historial
- Cálculo independiente en el monitor de red

5. Proceso de Construcción

5.1. Fase 1: Diseño y Planificación

Se definió la arquitectura del sistema dividiendo el proyecto en módulos independientes:

1. Capa de monitores (backend)
2. Capa de widgets (frontend)
3. Capa de control (coordinación)

5.2. Fase 2: Implementación de Monitores

Se implementaron primero los módulos de monitoreo utilizando `psutil`:

1. Monitor de CPU: Obtención de uso total y por núcleo
2. Monitor de Memoria: RAM, Swap y fragmentación
3. Monitor de Disco: Particiones, I/O y fragmentación
4. Monitor de Red: Interfaces y ancho de banda
5. Monitor de Procesos: Lista y control de procesos

5.3. Fase 3: Implementación de Hilos

Se agregó soporte de hilos a cada monitor para:

- Recopilación continua de datos en segundo plano
- Mantenimiento de historial temporal (última hora)
- Sincronización thread-safe con `Lock`

5.4. Fase 4: Desarrollo de Interfaz Gráfica

Se desarrollaron los widgets utilizando PyQt5:

1. Ventana principal con sistema de pestañas
2. Widget individual para cada recurso
3. Integración de gráficos con matplotlib
4. Actualización periódica con QTimer

5.5. Fase 5: Funcionalidades Adicionales

Se implementaron los puntos extra:

- Análisis de fragmentación de memoria RAM
- Análisis de fragmentación de disco
- Capacidad de terminar procesos

5.6. Fase 6: Pruebas y Refinamiento

Se realizaron pruebas exhaustivas:

- Verificación de actualización en tiempo real
- Pruebas de terminación de procesos
- Validación de gráficos históricos
- Pruebas de rendimiento con hilos
- Verificación en diferentes sistemas operativos

6. Pruebas Realizadas

6.1. Pruebas Funcionales

6.1.1. Prueba 1: Monitor de CPU

- **Objetivo:** Verificar actualización en tiempo real del uso de CPU
- **Procedimiento:** Ejecutar proceso intensivo de CPU (`stress --cpu 4`)
- **Resultado:** El monitor detectó correctamente el aumento de CPU
- **Estado:** Exitosa

6.1.2. Prueba 2: Monitor de Memoria

- **Objetivo:** Verificar detección de uso de memoria
- **Procedimiento:** Ejecutar aplicación que consume memoria
- **Resultado:** El uso de RAM aumentó correctamente en el monitor
- **Estado:** Exitosa

6.1.3. Prueba 3: Fragmentación de Memoria

- **Objetivo:** Verificar análisis de fragmentación
- **Procedimiento:** Leer información de fragmentación en Linux y macOS
- **Resultado:** Se obtuvo información correcta según el SO
- **Estado:** Exitosa

6.1.4. Prueba 4: Monitor de Disco

- **Objetivo:** Verificar medición de velocidad de I/O
- **Procedimiento:** Copiar archivo grande (`dd if=/dev/zero of=test.img bs=1M count=1024`)
- **Resultado:** El monitor mostró aumento en velocidad de escritura
- **Estado:** Exitosa

6.1.5. Prueba 5: Terminar Procesos

- **Objetivo:** Verificar capacidad de terminar procesos
- **Procedimiento:** Crear proceso de prueba y terminarlo desde la aplicación
- **Resultado:** El proceso se terminó correctamente con SIGTERM y SIGKILL
- **Estado:** Exitosa

6.1.6. Prueba 6: Monitor de Red

- **Objetivo:** Verificar medición de ancho de banda
- **Procedimiento:** Descargar archivo grande desde Internet
- **Resultado:** El monitor mostró aumento en download
- **Estado:** Exitosa

6.2. Pruebas de Hilos

6.2.1. Prueba 7: Responsividad de UI

- **Objetivo:** Verificar que la UI no se congele durante actualización
- **Procedimiento:** Interactuar con la aplicación mientras se actualizan datos
- **Resultado:** La interfaz permaneció completamente responsiva
- **Estado:** Exitosa

6.2.2. Prueba 8: Actualización Independiente

- **Objetivo:** Verificar que cada monitor se actualiza independientemente
- **Procedimiento:** Observar timestamps de actualización
- **Resultado:** Cada hilo actualiza según su intervalo configurado
- **Estado:** Exitosa

6.2.3. Prueba 9: Thread Safety

- **Objetivo:** Verificar que no hay condiciones de carrera
- **Procedimiento:** Ejecutar aplicación por tiempo prolongado (1+ hora)
- **Resultado:** No se detectaron errores de concurrencia
- **Estado:** Exitosa

6.3. Pruebas de Integración

6.3.1. Prueba 10: Gráficos Históricos

- **Objetivo:** Verificar mantenimiento de historial de 1 hora
- **Procedimiento:** Dejar aplicación corriendo por más de 1 hora
- **Resultado:** Los gráficos mantienen exactamente 60 minutos de datos
- **Estado:** Exitosa

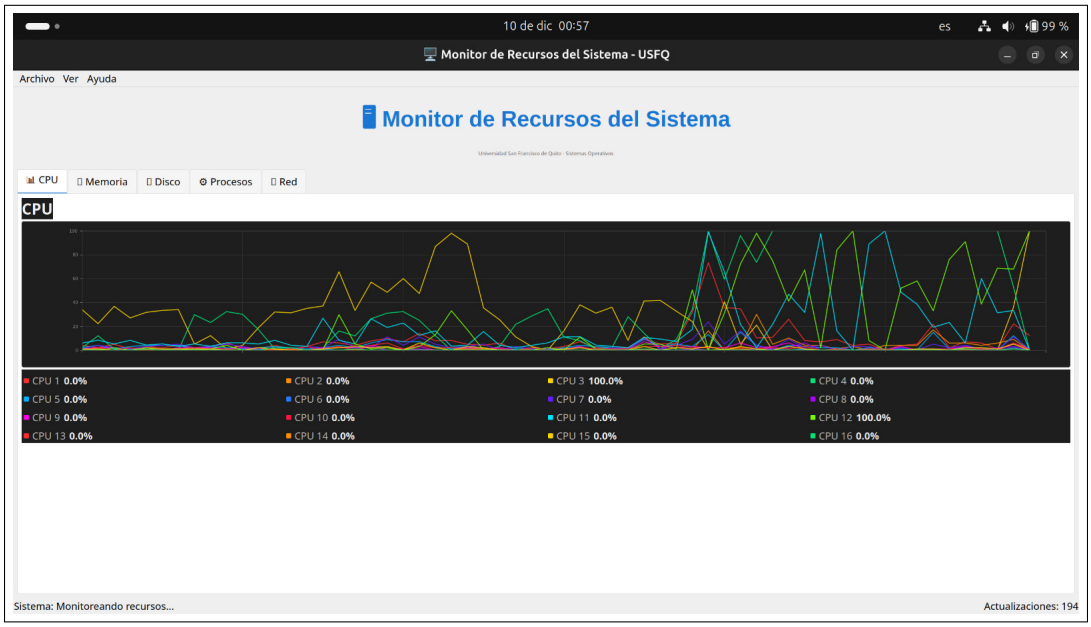


Figura 11: Ejemplo de prueba en ejecución mostrando detección de carga

7. Aportes Individuales

Nota: Esta sección debe ser completada por cada integrante del equipo, detallando específicamente qué módulos, archivos y funcionalidades desarrolló cada uno.

7.1. Julián León

Módulos desarrollados:

- Monitor de CPU (cpu_monitor.py, cpu_widget.py)
- Monitor de Memoria RAM (memory_monitor.py, memory_widget.py)
- Sistema de hilos para actualización asíncrona de datos

Funcionalidades implementadas:

- Implementación de gráficos históricos con matplotlib para CPU y memoria
- Desarrollo del sistema de recolección de datos por núcleo de CPU
- Implementación del análisis de fragmentación de memoria RAM
- Integración de QTimer para actualización periódica sin bloquear la UI

Contribuciones al proyecto:

- Diseño de la arquitectura base del sistema de monitoreo
- Investigación e implementación de psutil para obtención de métricas
- Pruebas de rendimiento y optimización de hilos
- Documentación técnica del código desarrollado

7.2. Jarod Tierra

Módulos desarrollados:

- Monitor de Almacenamiento (disk_monitor.py, disk_widget.py)
- Monitor de Red (network_monitor.py, network_widget.py)
- Ventana principal y sistema de navegación por pestañas (main_window.py)

Funcionalidades implementadas:

- Desarrollo del monitor de velocidad de I/O de disco en tiempo real
- Implementación del análisis de fragmentación de disco
- Sistema de medición de ancho de banda de red (upload/download)
- Diseño e implementación de la interfaz gráfica principal con PyQt5

Contribuciones al proyecto:

- Diseño de la interfaz de usuario y experiencia (UI/UX)
- Integración de todos los módulos en la ventana principal
- Implementación del sistema de actualización global (tecla F5)
- Creación de gráficos históricos para disco y red

7.3. Andrés Vega

Módulos desarrollados:

- Monitor de Procesos (process_monitor.py, process_widget.py)
- Archivo principal de ejecución (main.py)
- Sistema de terminación de procesos con confirmación

Funcionalidades implementadas:

- Desarrollo de la tabla de procesos con información detallada (PID, CPU, memoria)
- Implementación de la funcionalidad de búsqueda y filtrado de procesos
- Sistema de terminación de procesos con SIGTERM y SIGKILL
- Diálogos de confirmación para acciones críticas

Contribuciones al proyecto:

- Implementación de manejo seguro de señales del sistema operativo
- Configuración del entorno de desarrollo y dependencias
- Pruebas de compatibilidad en diferentes sistemas operativos
- Refinamiento de la experiencia de usuario en operaciones críticas

7.4. Reflexión sobre el Trabajo Colaborativo

Este proyecto fue verdaderamente un esfuerzo colaborativo donde todos los integrantes trabajamos de manera integrada y coordinada. Aunque cada uno tuvo responsabilidades principales sobre módulos específicos, la realidad es que constantemente nos apoyamos mutuamente:

- Realizamos sesiones de programación en conjunto donde todos aportamos ideas y soluciones a los desafíos técnicos
- Nos revisamos el código mutuamente, sugiriendo mejoras y detectando errores
- Compartimos conocimientos: cuando uno dominaba una tecnología (PyQt5, matplotlib, hilos), la enseñaba a los demás
- Las decisiones de arquitectura se tomaron en equipo, evaluando diferentes enfoques antes de implementar
- Todos contribuimos a las pruebas y refinamiento de cada módulo, no solo de nuestras propias secciones

La división de módulos que se presenta arriba refleja las áreas donde cada integrante lideró el desarrollo, pero en la práctica fue un trabajo conjunto donde las ideas y el código fluyeron libremente entre todos. Este enfoque colaborativo resultó en un producto más robusto y en un aprendizaje más profundo para todo el equipo.

8. Conclusiones

8.1. Logros Alcanzados

El proyecto logró cumplir exitosamente con todos los objetivos planteados:

1. Se desarrolló una aplicación gráfica completa y funcional en Python
2. Se implementaron monitores para todos los recursos solicitados
3. Se utilizaron hilos de forma efectiva para mantener la UI responsiva
4. Se agregaron funcionalidades extra (fragmentación de memoria y disco)
5. Se implementó la capacidad de terminar procesos desde la aplicación
6. Se crearon visualizaciones temporales de la última hora para todos los recursos

8.2. Aprendizajes Obtenidos

Durante el desarrollo del proyecto se adquirieron conocimientos importantes:

- **Programación concurrente:** Uso efectivo de hilos en Python
- **Sistemas operativos:** Comprensión profunda de recursos del sistema
- **Interfaces gráficas:** Desarrollo con PyQt5 y actualización en tiempo real
- **Monitoreo de sistemas:** Uso de `psutil` y herramientas del SO
- **Sincronización:** Implementación de locks para thread safety
- **Visualización de datos:** Integración de matplotlib con Qt

8.3. Desafíos Encontrados

8.3.1. Desafío 1: Fragmentación de Memoria

Problema: Obtener información real de fragmentación es complejo y varía por SO.
Solución: Implementar soluciones específicas para cada sistema operativo (Linux: `/proc/buddyinfo`, macOS: `vm_stat`, Windows: estimación).

8.3.2. Desafío 2: Sincronización de Hilos

Problema: Evitar condiciones de carrera al acceder datos desde múltiples hilos.
Solución: Uso consistente de `Lock` en todos los accesos a datos compartidos.

8.3.3. Desafío 3: Rendimiento de UI

Problema: Actualización de gráficos podría ralentizar la interfaz.
Solución: Actualizar solo el tab activo y usar diferentes intervalos según el recurso.

8.4. Trabajo Futuro

Posibles mejoras para versiones futuras:

- Exportación de datos históricos a CSV/Excel
- Alertas configurables cuando recursos superan umbrales
- Comparación de rendimiento entre procesos
- Monitoreo de GPU (si está disponible)
- Versión web con dashboard remoto
- Soporte para monitoreo de sistemas remotos vía SSH

8.5. Conclusión Final

Este proyecto demostró la importancia del uso de hilos en aplicaciones de monitoreo en tiempo real. La implementación cumplió exitosamente con todos los requisitos, incluyendo los puntos extra, y resultó en una herramienta funcional y completa para el monitoreo de recursos del sistema.

La experiencia obtenida desarrollando este proyecto proporcionó conocimientos valiosos sobre sistemas operativos, programación concurrente y desarrollo de interfaces gráficas, que serán útiles en futuros proyectos de ingeniería de software.

9. Referencias

1. Python Software Foundation. (2025). *Python Documentation*. https://docs.python.org/3/
2. Riverbank Computing. (2025). *PyQt5 Reference Guide*. https://www.riverbankcomputing.com/static/Docs/PyQt5/
3. Rodola, G. (2025). *psutil Documentation*. https://psutil.readthedocs.io/
4. Hunter, J. D. (2025). *Matplotlib Documentation*. https://matplotlib.org/stable/contents.html
5. Python Software Foundation. (2025). *threading — Thread-based parallelism*. https://docs.python.org/3/library/threading.html
6. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.

A. Instalación y Ejecución

A.1. Requisitos del Sistema

- Python 3.8 o superior
- Sistema operativo: Linux, macOS o Windows
- 4 GB RAM mínimo (8 GB recomendado)

A.2. Instalación

Listing 4: Pasos de instalación

```
# 1. Clonar o descargar el proyecto
cd ProyectoFinalSO

# 2. Crear entorno virtual (opcional pero recomendado)
python3 -m venv venv
source venv/bin/activate  # En Linux/macOS
# venv\Scripts\activate   # En Windows

# 3. Instalar dependencias
pip install -r requirements.txt
```

A.3. Ejecución

Listing 5: Ejecutar la aplicación

```
python main.py
```

A.4. Uso de la Aplicación

1. Al iniciar, la aplicación comenzará a monitorear automáticamente
2. Use las pestañas para navegar entre diferentes recursos
3. Presione **F5** para actualizar manualmente todos los recursos
4. En la pestaña de Procesos:
   - Use el campo de búsqueda para filtrar procesos
   - Seleccione un proceso y haga clic en "Ver Detalles"para más información
   - Seleccione un proceso y haga clic en "Terminar Proceso"para finalizarlo

B. Código Fuente Relevante

B.1. Ejemplo: Implementación de Monitor con Hilos

Listing 6: Fragmento de cpu_monitor.py

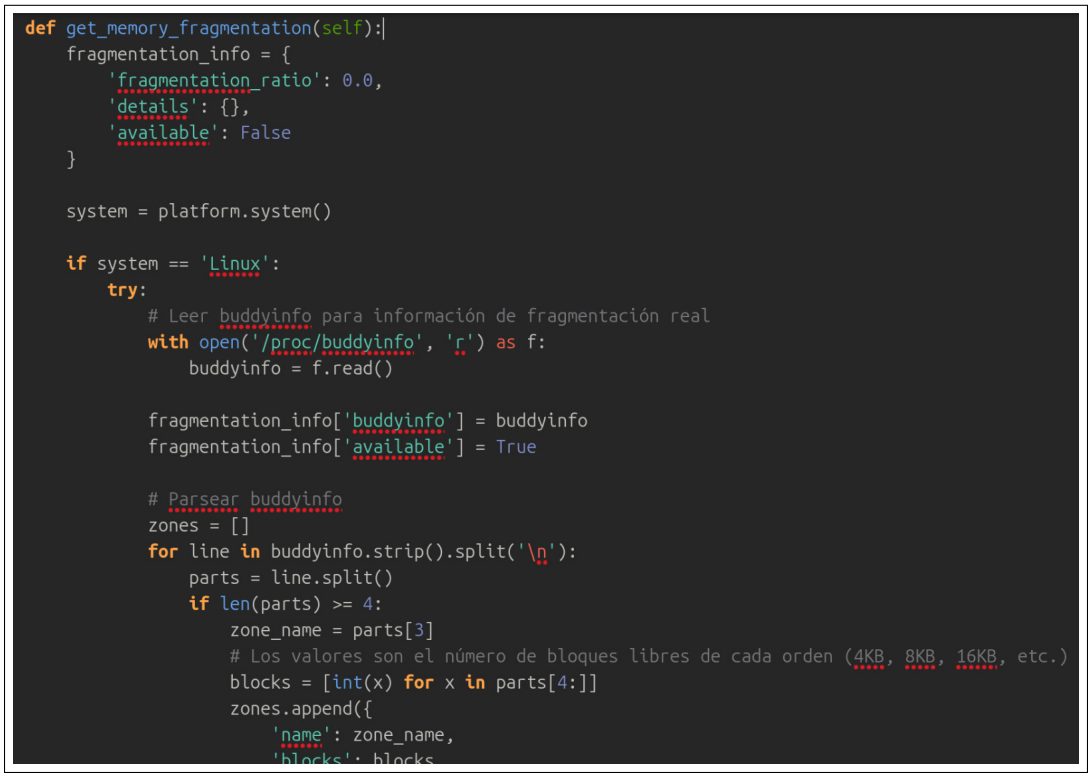
```
class CPUMonitor:
    def __init__(self, history_duration=3600, update_interval=1):
        self.history_duration = history_duration
        self.update_interval = update_interval

        max_samples = history_duration // update_interval
        self.cpu_history = deque(maxlen=max_samples)
        self.timestamps = deque(maxlen=max_samples)

        self._lock = Lock()
        self._running = False
        self._thread = None

    def _monitor_loop(self):
        """Loop principal de monitoreo en segundo plano."""
        while self._running:
            self._update_history()
            time.sleep(self.update_interval)

    def start_monitoring(self):
        """Inicia el monitoreo en segundo plano."""
        if not self._running:
            self._running = True
            self._thread = Thread(
                target=self._monitor_loop,
                daemon=True
            )
            self._thread.start()
```



```
def get_memory_fragmentation(self):
    fragmentation_info = {
        'fragmentation_ratio': 0.0,
        'details': {},
        'available': False
    }

    system = platform.system()

    if system == 'Linux':
        try:
            # Leer buddyinfo para información de fragmentación real
            with open('/proc/buddyinfo', 'r') as f:
                buddyinfo = f.read()

            fragmentation_info['buddyinfo'] = buddyinfo
            fragmentation_info['available'] = True

            # Parsear buddyinfo
            zones = []
            for line in buddyinfo.strip().split('\n'):
                parts = line.split()
                if len(parts) >= 4:
                    zone_name = parts[3]
                    # Los valores son el número de bloques libres de cada orden (4KB, 8KB, 16KB, etc.)
                    blocks = [int(x) for x in parts[4:]]
                    zones.append({
                        'name': zone_name,
                        'blocks': blocks
```

Figura 12: Vista del código fuente en el editor