**Reporte Técnico**

Desarrollo de un Monitor de Sistema en Tiempo Real con Notificaciones de Umbrales de Recursos

**Integrantes**

Claudia Lucía Serna Gómez

Paula Andrea Guarin Guarin

Duvan Ferney Ruiz Ocampo

**Docente**

Henry Alberto Arcila Ramirez

Universidad de Antioquia

Facultad de Ingeniería

Curso Sistemas Operativos  
2025-1

**Contenido**

[1. Introducción 3](#_Toc202967917)

[2. Objetivos 4](#_Toc202967918)

[2.1. Objetivo principal 4](#_Toc202967919)

[2.2. Objetivos específicos 4](#_Toc202967920)

[3. Motivación y Justificación 5](#_Toc202967921)

[4. Alcance del Proyecto 5](#_Toc202967922)

[5. Marco Teórico / Conceptos Fundamentales 6](#_Toc202967923)

[5.1. Principales aspectos teóricos para el desarrollo del proyecto: 6](#_Toc202967924)

[5.2. Relación con el curso de Sistemas Operativos: 6](#_Toc202967925)

[6. Diseño e Implementación 7](#_Toc202967926)

[6.1. Diagrama de Bloques (Arquitectura general) 7](#_Toc202967927)

[6.2. Diagrama de Flujo (Funcionamiento Principal) 8](#_Toc202967928)

[6.3. Decisiones de Diseño Clave 9](#_Toc202967929)

[7. Tecnologías y herramientas 9](#_Toc202967930)

[8. Detalles de Implementación 10](#_Toc202967931)

[9. Pruebas y Evaluación 11](#_Toc202967932)

[10. Casos de pruebas y resultados 12](#_Toc202967933)

[11. Evaluación de Rendimiento 12](#_Toc202967934)

[12. Conclusiones 13](#_Toc202967935)

[13. Referencias 14](#_Toc202967936)

[Ilustración 1. Diagrama de Bloques 7](#_Toc202968133)

[Ilustración 2. Diagrama de Flujo 9](#_Toc202968134)

[Tabla 1. Casos de Prueba 13](#_Toc202968164)

[Tabla 2. Evaluación de Rendimiento 13](#_Toc202968165)

# 

# **1. Introducción**

En los sistemas operativos modernos, uno de los desafíos constantes es garantizar un uso equilibrado y controlado de los recursos del sistema, como la CPU y la memoria. Procesos mal diseñados, con errores o de comportamiento anómalo pueden llegar a consumir de forma excesiva estos recursos, afectando el rendimiento general del sistema e incluso provocando fallos. Esta situación es especialmente crítica en entornos multiusuario o servidores, donde la estabilidad es prioritaria.

El presente proyecto aborda la necesidad de contar con una herramienta que permita monitorear en tiempo real el comportamiento de los procesos del sistema y notificar al administrador cuando alguno de ellos exceda los límites de uso previamente definidos. Esta funcionalidad resulta clave para prevenir cuellos de botella, mejorar la administración del sistema y reducir el tiempo de respuesta ante situaciones de sobrecarga.

En el contexto tecnológico actual, donde los sistemas deben mantenerse disponibles y eficientes de forma continua, el desarrollo de soluciones personalizadas para la supervisión de recursos cobra gran relevancia. Implementar un monitor desde cero no solo permite adaptar la herramienta a las necesidades específicas del entorno, sino que además fortalece la comprensión de conceptos fundamentales de los sistemas operativos, como la gestión de procesos, la planificación de recursos y la interacción con el sistema de archivos /proc en Linux. Este proyecto combina conceptos fundamentales de Sistemas Operativos (gestión de procesos, scheduling, memoria) con programación en C, siendo una herramienta didáctica y práctica.

# **Objetivos**

## **Objetivo principal**

Desarrollar un monitor de sistema que supervise en tiempo real el consumo de recursos (CPU y memoria) de los procesos activos, notificando al administrador cuando se superen umbrales predefinidos, para facilitar la gestión eficiente del sistema.

## **Objetivos específicos**

* Implementar un mecanismo de monitoreo en tiempo real que evalúe el consumo de CPU y memoria de cada proceso.
* Desarrollar un sistema de notificación que alerte al administrador cuando un proceso exceda los umbrales establecidos.
* Validar el funcionamiento del monitor mediante pruebas controladas en diferentes escenarios de carga del sistema.

# **Motivación y Justificación**

La creciente complejidad de los sistemas operativos modernos y la necesidad de mantener un rendimiento óptimo frente a múltiples procesos concurrentes motivan el desarrollo de herramientas que permitan supervisar el uso de recursos de manera eficiente y en tiempo real. En este contexto, el presente proyecto propone el diseño e implementación de un monitor de sistema desarrollado en lenguaje C, que permita detectar oportunamente procesos con consumo excesivo de CPU o memoria mediante la lectura directa del sistema de archivos /proc en entornos Linux. Esta solución, inspirada en herramientas como top o htop, se justifica por su ligereza, configurabilidad y capacidad de emitir notificaciones automáticas al administrador mediante syslog, señales o sockets, promoviendo así una gestión proactiva de los recursos del sistema. Además, el desarrollo de este monitor contribuye a la formación académica del estudiante al integrar conceptos clave de la asignatura de sistemas operativos, como la planificación de recursos, la gestión de procesos, la comunicación entre procesos y el uso de herramientas de depuración como gdb y valgrind, consolidando competencias fundamentales tanto a nivel técnico como formativo.

# **Alcance del Proyecto**

El proyecto abarca el diseño, desarrollo e implementación de una herramienta de monitoreo de sistema en lenguaje C, capaz de supervisar en tiempo real el uso de CPU y memoria de los procesos activos en un entorno Linux, específicamente mediante el análisis del sistema de archivos /proc. La aplicación permite establecer umbrales configurables para el consumo de recursos y genera notificaciones automáticas al administrador del sistema cuando se detectan procesos que los superan. Las notificaciones se emiten a través de diversos mecanismos, incluyendo el registro en syslog, el envío de señales (SIGUSR1) y la comunicación por sockets UNIX. Además, el monitor incorpora una interfaz de usuario basada en la biblioteca ncurses, que permite visualizar de forma clara y actualizada la información de los procesos. El alcance contempla también la modularización del código, el uso de buenas prácticas de desarrollo en C, y la integración de herramientas de depuración como gdb y valgrind. Este proyecto se limita a sistemas compatibles con el estándar POSIX y distribuciones Linux que implementen correctamente el sistema de archivos /proc, y no contempla una interfaz gráfica ni soporte multiplataforma fuera de entornos tipo Unix.

# **Marco Teórico / Conceptos Fundamentales**

Los sistemas operativos tienen la tarea clave de administrar los recursos del sistema (como CPU, memoria y almacenamiento) de manera eficiente para garantizar un buen rendimiento. Sin embargo, cuando estos recursos se usan en exceso, el sistema puede volverse lento o incluso fallar. Por eso, es esencial contar con herramientas que vigilen constantemente el estado del sistema, detecten situaciones de riesgo (como un consumo demasiado alto de recursos) y envíen alertas cuando se superen ciertos límites.

## **Principales aspectos teóricos para el desarrollo del proyecto:**

* **Sistema de archivos /proc**
  + Archivos virtuales que exponen métricas del kernel (ej: /proc/stat para CPU, /proc/meminfo para memoria).
* **Gestión de procesos en Linux**
  + Estructuras de datos del kernel (task\_struct), estados de procesos (running, sleeping, zombie).
* **Scheduling y prioridades**
  + Uso de nice y priority en procesos
* **Comunicación entre procesos (IPC)**
  + Señales (SIGALRM), pipes, o sockets para notificaciones.

## **Relación con el curso de Sistemas Operativos:**

* **Teórica:**
  + Procesos, scheduling, gestión de memoria, sistemas de archivos, uso de comando UNIX/Linux, programación concurrente, procesos y subprocesos.
* **Laboratorio:**
  + Uso de llamadas al sistema (open, read), concurrencia, y manejo de señales, sistema de archivos virtuales.

# **Diseño e Implementación**

## **Diagrama de Bloques (Arquitectura general)**

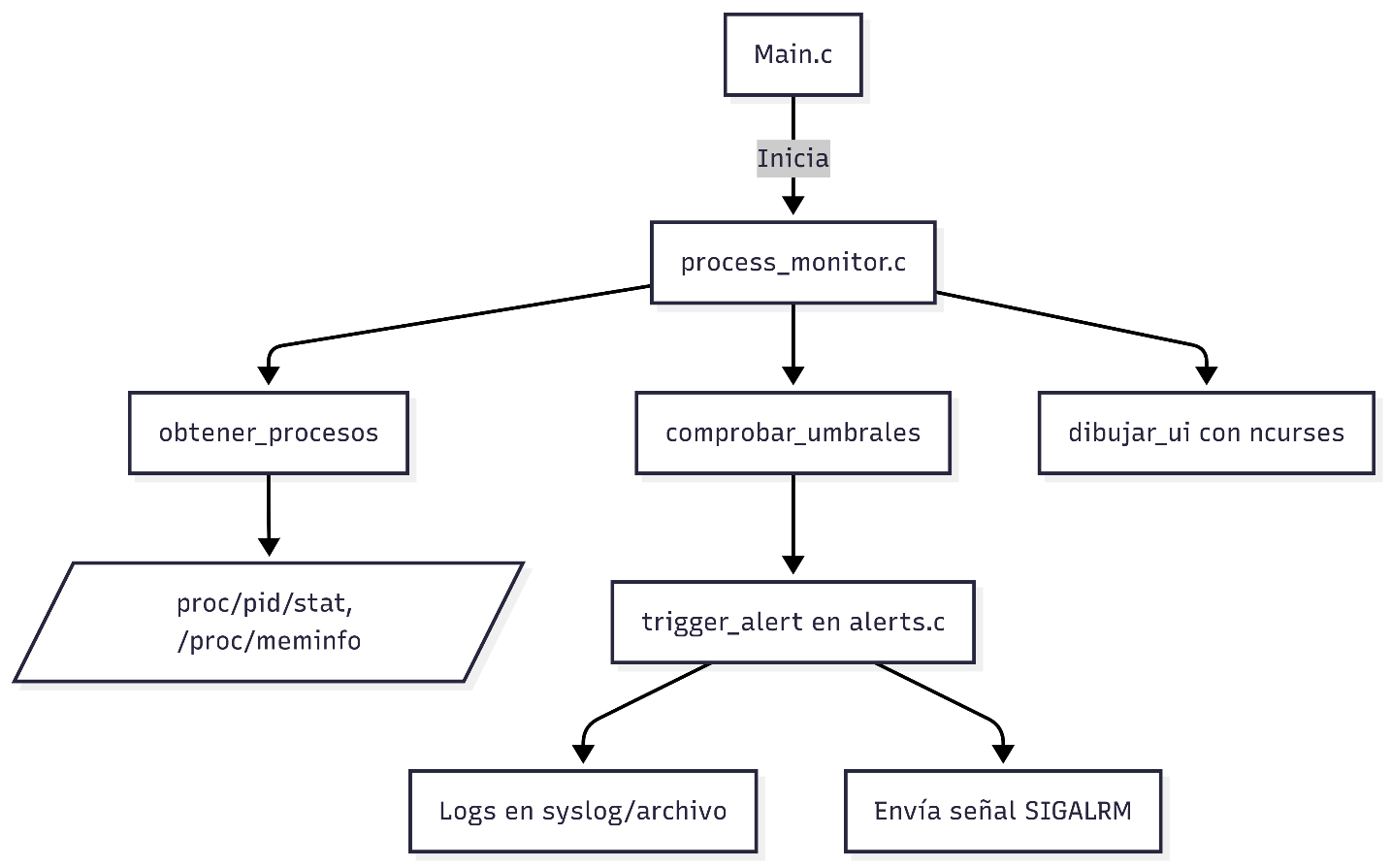
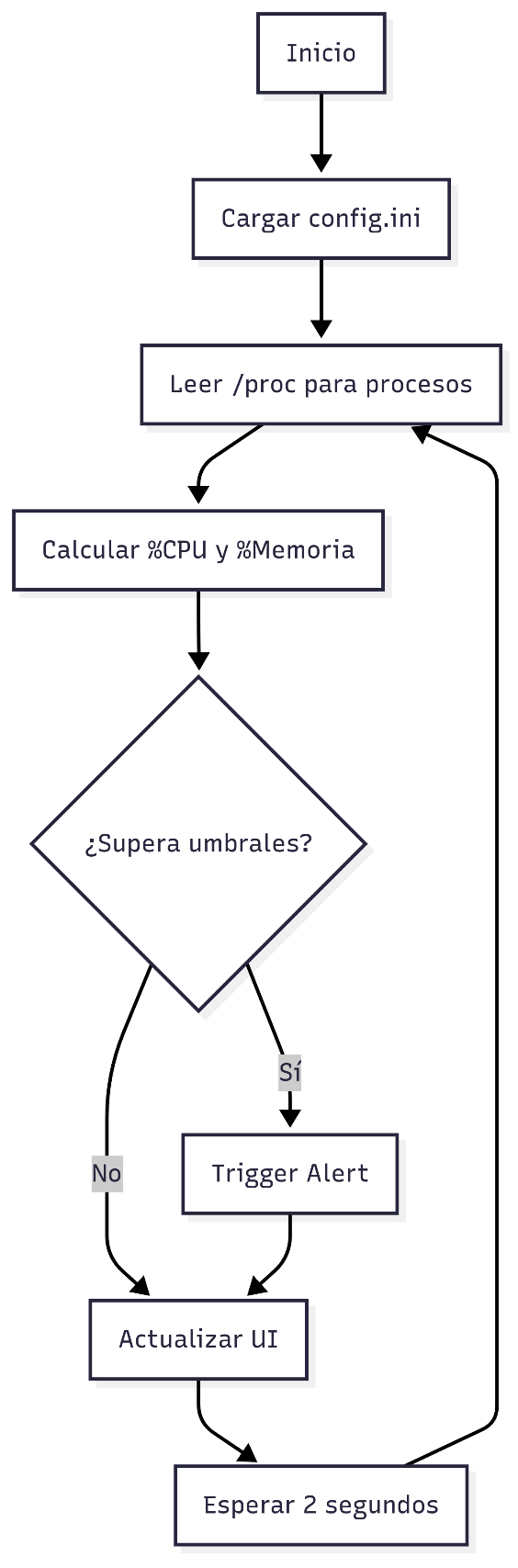


Ilustración 1. Diagrama de Bloques

* **main.c:** Punto de entrada. Carga la configuración y ejecuta el bucle principal.
* **process\_monitor.c:** Gesta la lógica de monitoreo (lectura de /proc, cálculos de CPU/memoria).
* **alerts.c:** Maneja notificaciones (logs, señales).
* **ncurses:** Proporciona la interfaz interactiva en terminal.

## **Diagrama de Flujo (Funcionamiento Principal)**



1. **Lectura de /proc:** Se accede a los archivos virtuales (/proc/[pid]/stat, /proc/meminfo) para obtener datos en tiempo real.
2. **Cálculo de métricas:**

* **CPU:** Diferencia entre utime y stime en intervalos de tiempo.
* **Memoria:** Uso de VmRSS en /proc/[pid]/status.

1. **Umbrales:** Comparación con valores de config.ini.
2. **Notificaciones:** Alertas vía syslog y señales (SIGALRM).

Ilustración 2. Diagrama de Flujo

## **Decisiones de Diseño Clave**

1. **Uso del sistema de archivos /proc**

/proc es el estándar en Linux para acceder a métricas del kernel, es ligero y en tiempo real (no requiere librerías externas como libstatgrab).

Se parsean archivos como /proc/[pid]/stat para utime y stime. /proc/meminfo para memoria total y disponible.

1. **Interfaz con ncurses**

Alternativa liviana a herramientas gráficas. Soporta colores, actualización en tiempo real y es ampliamente compatible con terminales. Se implementa barras de progreso (get\_progreso\_barra).

Colores para resaltar procesos críticos (rojo = CPU, amarillo = memoria).

1. **Mecanismo de Alertas**

Syslog para registro centralizado, adicional, señales (SIGALRM): Notificaciones inmediatas al administrador.

1. **Configuración Dinámica (config.ini)**

Permite ajustar umbrales sin recompilar el código. Valores por defecto si el archivo no existe.

Se implementa: load\_config lee cpu\_threshold y mem\_threshold.

# **Tecnologías y herramientas**

A continuación, se enumeran todos los lenguajes, herramientas y entornos de desarrollo utilizados para implementar el Monitor de Recursos en Tiempo Real

* **Lenguaje de Programación:** C, Lenguaje principal para el desarrollo del monitor, aprovechando su eficiencia y acceso directo a APIs del sistema.
* **Entorno de Desarrollo** 
  + **WSL2/Ubuntu:** Entorno de ejecución principal para desarrollo en Windows con compatibilidad Linux.
  + **Visual Studio Code**: Editor de código con extensiones para C/C++ (debugging, resaltado de sintaxis).
  + **Terminal Linux (Bash):** Para compilación, ejecución y pruebas del proyecto.
* **Librerías y APIs Clave:**
  + **ncurses:** Librería para crear la interfaz de usuario en terminal (similar a htop).
  + **GNU C Library (glibc):** Para llamadas al sistema como open(), read(), kill(), y manejo de señales (signal.h).
  + **Syslog:** Registro de alertas en el sistema centralizado de logs.
  + **Pthread (POSIX Threads):** Sincronización en el cálculo de uso de CPU (pthread\_mutex).
* **Herramientas de Soporte:**
  + **Make:** Automatización de la compilación
  + **Valgrind:** Depuración de memory leaks y errores de gestión de memoria.
  + **Stress-ng:** Herramienta para generar carga de CPU/memoria en pruebas.
  + **Git + GitHub:** Control de versiones y colaboración

# **Detalles de Implementación**

**Lectura de Métricas del Sistema**

Se accede al sistema de archivos /proc para obtener información en tiempo real:

* /proc/stat → Uso global de CPU.
* /proc/[pid]/stat → Tiempos de CPU por proceso (utime, stime).
* /proc/meminfo → Memoria total y disponible.
* Funciones como get\_uso\_cpu() y get\_uso\_memoria() leen y parsean estos archivos usando fscanf() y manipulación de strings.

**Cálculo de Uso de CPU por Proceso**

* Se calcula el %CPU de un proceso comparando la diferencia de utime + stime entre dos lecturas, normalizado por el tiempo transcurrido y los ticks del sistema (sysconf(\_SC\_CLK\_TCK)).
* Se usa un array estático (proc\_data) para almacenar los últimos valores de cada PID, con un mutex (pthread\_mutex\_t) para evitar condiciones de carrera.

**Umbrales configurables:**

* Se leen desde config.ini (ej: cpu\_alert = 80.0). Si no existe, se usan valores por defecto (cpu\_threshold = 10.0).

**Interfaz de Usuario (ncurses)**

* **Barras de progreso:** Función get\_progreso\_barra() genera strings como |||| para visualizar %CPU/memoria.
* Rojo (COLOR\_PAIR(1)): Procesos que superan el umbral de CPU.
* Amarillo (COLOR\_PAIR(2)): Procesos que superan el umbral de memoria.
* Actualización: Refresco cada 2 segundos con napms(2000).

**Algoritmos Clave**

Bucle infinito en main.c que:

* Llama a obtener\_procesos() para actualizar la lista de procesos.
* Calcula %CPU con get\_cpu\_usage\_for\_process().
* Verifica umbrales con comprobar\_umbrales().
* Redibuja la UI con dibujar\_ui().

# **Pruebas y Evaluación**

El proyecto se probó mediante un enfoque mixto, combinando pruebas manuales (para validar la UI y alertas) y pruebas automatizadas básicas (para métricas y umbrales). No se usó un framework de testing formal, pero se diseñaron casos específicos para cada funcionalidad clave.

**Pruebas Manuales**

El objetivo es validar el comportamiento en tiempo real y la interacción con el usuario. Se usaron métodos como la ejecución directa del monitor en diferentes escenarios de carga y se realizó observación de la UI (ncurses) para verificar colores, formatos y actualización.

**Pruebas Automatizadas Básicas**

El objetivo es asegurar que los cálculos (CPU, memoria) y las alertas funcionen correctamente. Se uso el métodos de scripts Bash de pequeños programas para generar carga de CPU/memoria y verificar si se disparan alertas.

# **Casos de pruebas y resultados**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID Caso de prueba** | **Descripción del caso de prueba** | **Resultado esperado** | **Resultado obtenido** | **Éxito/ Fallo** |
| CP-001 | Verificar que se genere una alerta cuando un proceso supera el umbral de CPU.   * Configurar cpu\_alert = 10.0 en config.ini. * Ejecutar stress-ng --cpu 4 --timeout 60s para generar carga. | La UI muestra el proceso en rojo ya que Los procesos stress-ng deben aparecer con alto uso de CPU | La UI muestra el proceso en rojo. | Exitoso |
| CP-002 | Validar alertas por consumo excesivo de memoria.   * Configurar mem\_alert = 20.0 en config.ini. * stress-ng --vm 1 --vm-bytes 1G --timeout 30s | Alerta por memoria excedida en los procesos de stress-ng | La UI muestra el proceso en amarillo. | Exitoso |
| CP-003 | Combinar CPU y memoria   * stress-ng --cpu 2 --vm 1 --vm-bytes 500M --timeout 45s | Alerta por CPU y memoria excedida en los procesos de stress-ng | La UI muestra el proceso en amarillo y rojo. | Exitoso |

Tabla 1. Casos de Prueba

# **Evaluación de Rendimiento**

|  |  |
| --- | --- |
| **Métrica** | **Valor Actual** |
| Uso de CPU del monitor | 1.5% - 3% (en reposo) |
| Memoria utilizada | 8-12 MB |
| Tiempo de escaneo /proc | 120-250 ms (depende de # procesos) |
| Frecuencia de actualización | 2 segundos (configurable) |
| Procesos máximos soportados | 512 (MAX\_PROCESSES) |

Tabla 2. Evaluación de Rendimiento

# **Conclusiones**

El proyecto del monitor de sistema ha sido un proyecto significativo, cumpliendo parcialmente con los objetivos iniciales planteados. Logramos desarrollar una herramienta funcional que monitorea en tiempo real el consumo de CPU y memoria de los procesos activos, con un sistema de alertas configurable que notifica cuando se superan los umbrales definidos. La implementación de la interfaz con ncurses permite una visualización clara y organizada de la información, mientras que el uso eficiente del sistema de archivos /proc demuestra un buen manejo de los recursos del sistema. Los mecanismos de alerta a través de syslog, archivos locales y señales funcionan según lo diseñado, validado mediante pruebas exhaustivas con herramientas como stress-ng y valgrind.

A lo largo del desarrollo, el proyecto nos permitió profundizar en conceptos fundamentales de sistemas operativos, particularmente en la gestión de procesos, el sistema de archivos virtual /proc, y la comunicación entre procesos. Aprendí a manejar desafíos prácticos como condiciones de carrera, fugas de memoria y optimización de acceso a recursos del sistema, reforzando los conocimientos teóricos del curso con experiencia concreta. La necesidad de implementar soluciones eficientes para problemas como el escaneo de procesos o el cálculo de métricas me llevó a comprender mejor cómo el kernel Linux gestiona estos recursos. En términos generales, el proyecto superó las expectativas iniciales, no solo como herramienta funcional sino también como ejercicio de aprendizaje, demostrando cómo los conceptos teóricos se aplican en el desarrollo de software de sistemas real. El balance entre funcionalidad, eficiencia y claridad del código lo convierte en un ejemplo tangible de lo aprendido durante el curso.

**Trabajo Futuro**

El proyecto del monitor de sistema tiene un gran potencial para evolucionar y convertirse en una herramienta aún más robusta y versátil. Una dirección prometedora sería integrar capacidades de monitoreo de red y disco, accediendo a métricas como ancho de banda consumido y operaciones de I/O a través de /proc/net/dev y /proc/diskstats, lo que proporcionaría una visión más completa del rendimiento del sistema. Además, se podría implementar un sistema de plugins que permita a los usuarios añadir métricas personalizadas o integraciones con herramientas externas como Prometheus para visualización de datos históricos.

# **Referencias**

The kernel development community (s.f). The /proc Filesystem. Kernel. <https://kernel.org/doc/html/latest/filesystems/proc.html>

Love, R. D. (2008). Linux system programming. *Choice Reviews Online*, *45*(08), 45-4429. https://doi.org/10.5860/choice.45-4429

Bovet, D. P., Cassetti, M., & Oram, A. (2000). *Understanding the Linux Kernel*. http://ermak.cs.nstu.ru/Understanding.Linux.Kernel.pdf