Proyecto de Curso:
Análisis, Diseño y Construcción de un Sistema Operativo desde Cero

Por

Castro Pari, Rayneld Fidel

Mamani Flores, Natan

Mendoza Quispe, Jose Daniel

Polo Chura, Marco Rosauro

Trabajo académico presentado a la

Facultad de Ingeniería

proyecto final de unidad

Ingeniería de Informatica y Sistemas

Departamento Académico de Ingeniería de Informatica y de Sistemas

Asesor: Ugarte Rojas, Héctor Eduardo

Memorial Universidad Nacional San Antonio Abad del Cusco

Semestre 2025-II

Cusco Perú

Abstract

This document provides information on how to write your thesis using the LaTeX document preparation system. You can use these files as a template for your own thesis, just replace the content, as necessary. You should put your real abstract here, of course.

"The purpose of the abstract, which should not exceed 150 words for a Masters' thesis or 350 words for a Doctoral thesis, is to provide sufficient information to allow potential readers to decide on relevance of the thesis. Abstracts listed in Dissertation Abstracts International or Masters' Abstracts International should contain appropriate key words and phrases designed to assist electronic searches."

— MUN School of Graduate Studies

Acknowledgements

Put your acknowledgements here...

"Intellectual and practical assistance, advice, encouragement and sources of monetary support should be acknowledged. It is appropriate to acknowledge the prior publication of any material included in the thesis either in this section or in the introductory chapter of the thesis."

— MUN School of Graduate Studies

Contents

List of Tables

List of Figures

Chapter 1

Introduccion

Chapter 2

Marco teórico

2.1 ¿Que es un sistema operativo?

Según Stallings, el sistema operativo es el software encargado de controlar la ejecución de los programas y de administrar los recursos del procesador; además actúa como intermediario entre el usuario y el hardware, proporcionando los servicios necesarios para que las aplicaciones puedan ejecutarse correctamente (Stallings, 2023)

Añadiendo esta idea, Tanenbaum y Bos explican que la finalidad del sistema operativo es convertir las interfaces de hardware, que suelen ser complejas e inconsistentes, en abstracciones practicas y sencillas de usar para los programadores y aplicaciones. Ademas de gestionar recursos como CPU, memoria y dispositivos de E/S, permitiendo así que las aplicaciones trabajen sin problemas con el hardware (Tanenbaum and Bos, 2023).

Según el artículo de la Revista Ogma, que es un científica y multidisciplinaria, un sistema operativo es el software principal que actúa como intermediario entre el usuario y el hardware, cuyo propósito es favorecer el uso eficiente de la computadora. Transforma las interfaces de hardware, complejas e inconsistentes, en abstracciones útiles y manejables para y aplicaciones y programadores, administra y mejora recursos (CPU, memoria, dispositivos de E/S y almacenamiento); organiza la ejecución concurrente de tareas, y ofrece espacios accesibles que permite el acceso de las computadoras, permitiendo que usuarios sin preparación instalen programas, administren archivos. (Cusme Vera et al., 2022)

Un sistema operativo es como el sistema nervioso del computador, coordinando componentes hardware y software para que todo funcione de manera ordenada. Se encarga de organizar y administrar recursos como procesador, memoria, dispositivos

de E/S y almacenamiento.

2.2 Arquitecturas de Sistemas Operativos

2.2.1 Arquitectura Monolítica

El sistema operativo monolítico es un sistema operativo muy simple donde el núcleo controla directamente la gestión de dispositivos, memoria, archivos y procesos. Todos los recursos del sistema son accesibles al núcleo. En los sistemas monolíticos, cada componente del sistema operativo está contenido dentro del núcleo.

En una arquitectura monolítica, el núcleo del sistema operativo está diseñado para proporcionar todos los servicios del sistema operativo, incluyendo la gestión de memoria, la programación de procesos, los controladores de dispositivos y los sistemas de archivos, en un único y gran binario. Esto significa que todo el código se ejecuta en el espacio del núcleo, sin separación entre los procesos del núcleo y los del usuario (GeeksforGeeks, 2024b).

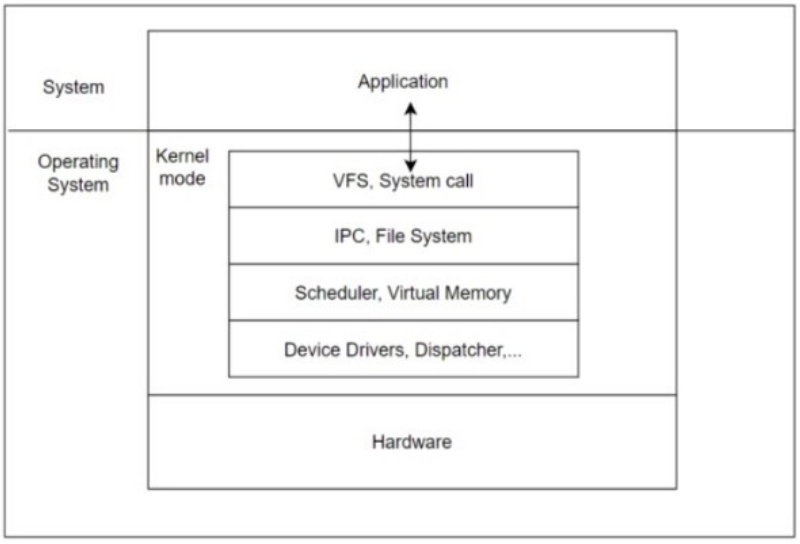

Figure 2.1: Sistema operativo basado en kernel monolítico (Harshvardhan and Irabatti, 2023)

2.2.2 Arquitectura en Microkernel

Un microkernel es un enfoque para diseñar un sistema operativo (SO). El microkernel proporciona servicios fundamentales para su funcionamiento, como la gestión básica de memoria, la programación de tareas.El microkernel es un tipo de sistema operativo que proporciona servicios básicos.

como los controladores de dispositivos y los sistemas de archivos, son gestionados por procesos a nivel de usuario. El proceso a nivel de usuario se comunica con el microkernel mediante el paso de mensajes. Esta forma de gestionar el proceso hace que los microkernels sean más modulares y flexibles que los kernels monolíticos tradicionales (GeeksforGeeks, 2024a).

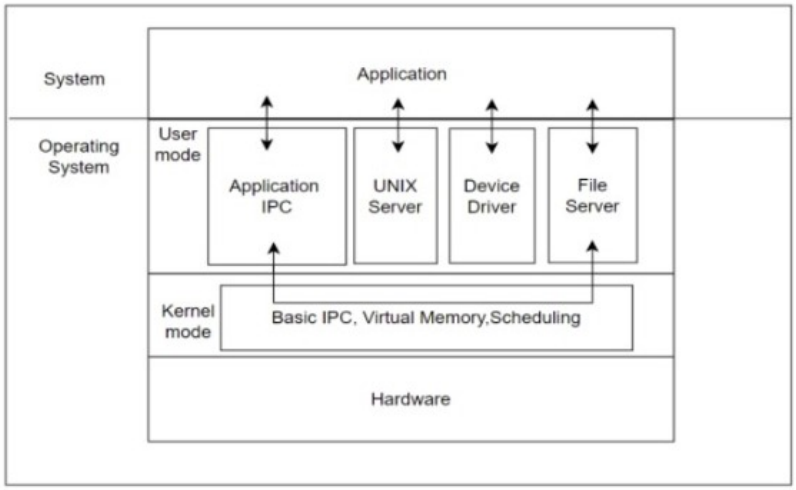

Figure 2.2: Sistema operativo basado en microkernel (Harshvardhan and Irabatti, 2023)

2.2.3 Arquitectura Hibrida

Un núcleo híbrido es una arquitectura de núcleo basada en la combinación de aspectos de las arquitecturas de micronúcleo y núcleo monolítico utilizadas en sistemas operativos .

La idea de esta categoría es tener una estructura de kernel similar a la de un microkernel, pero implementada en términos de un kernel monolítico. A diferencia de un microkernel, todos (o casi todos) los servicios del sistema operativo se encuentran en el espacio del kernel . Si bien no hay sobrecarga de rendimiento para el paso de mensajes ni el cambio de contexto entre el kernel y el modo de usuario, como en los kernels monolíticos , no hay ventajas de rendimiento al tener servicios en el espacio de usuario , como en los microkernels (Microsoft Wiki / Fandom, 2024).

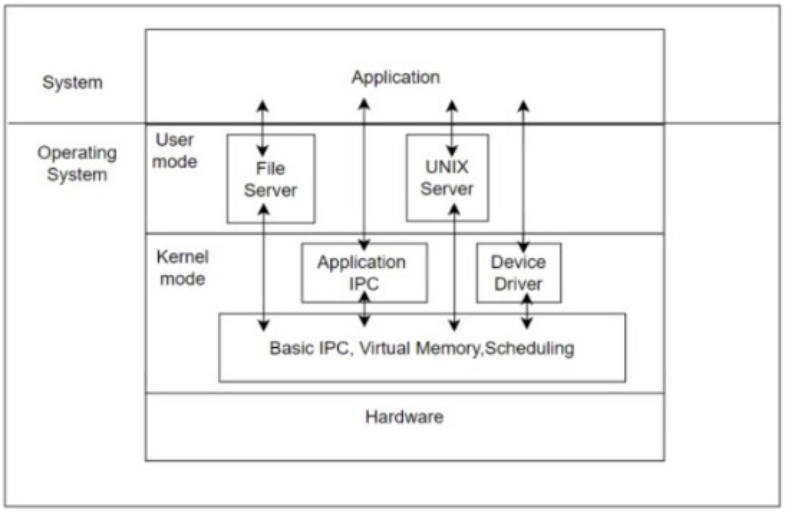

Figure 2.3: Sistema operativo basado en kernel híbrido (Harshvardhan and Irabatti, 2023)

2.2.4 Arquitectura en Exokernel

Exokernel es un sistema operativo desarrollado por el Instituto Tecnológico de Massachusetts (MIT) con el concepto de poner la aplicación bajo control. Los sistemas operativos Exokernel buscan proporcionar gestión de recursos de hardware a nivel de aplicación. La arquitectura de este sistema operativo está diseñada para separar la protección de recursos de la gestión, facilitando así la personalización específica de cada aplicación(Keetmalin, 2017).

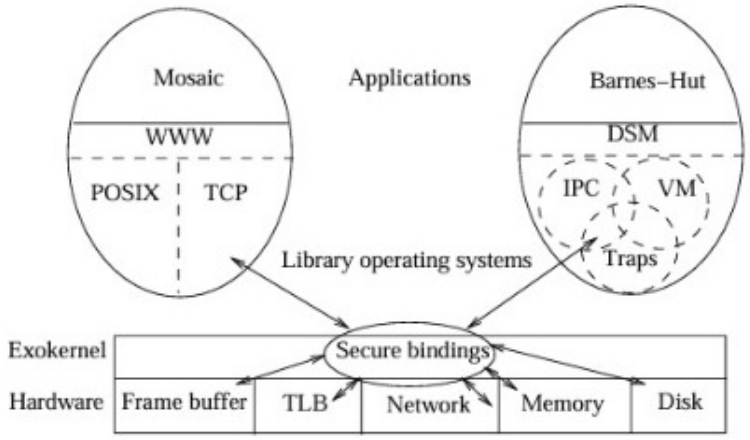

Figure 2.4: Sistema operativo basado en exokernel (Engler et al., 1995)

2.2.5 Comparacion entre arquitecturas de Sistemas Operativos

Table 2.1: Comparación de arquitecturas de SO

Arquitectura	**Ventajas**	**Desventajas**
Monolítica	Alto rendimiento, simple en diseño inicial	Difícil de mantener, poco modular
Microkernel	Modularidad, mayor seguridad	Mayor sobrecarga de comunicación
Hibrida	Combina rendimiento y modularidad	Complejidad de implementacion
Exokernel	Máxima flexibilidad y control	Muy complejo, poco usado en producción

2.3 Componentes principales de un sistema operativo

Un sistema operativo integra diversos componentes que gestionan los recursos de hardware y ofrecen servicios a los procesos que se ejecutan en el sistema.

2.3.1 Gestión de procesos

La gestión de procesos es uno de los pilares fundamentales de cualquier sistema operativo, ya que se encarga de controlar la ejecución de programas y de garantizar un uso eficiente de la CPU. Un proceso, desde su creación hasta su finalización, pasa por diferentes estados a lo largo de su ciclo de vida. Al crearse, entra en el estado nuevo; luego pasa al estado preparado mientras espera acceso a la CPU. Cuando el planificador lo selecciona, pasa a ejecutando. Durante la ejecución, un proceso puede quedar en espera si requiere operaciones de entrada/salida, y finalmente alcanza el estado terminado al completar su tarea, como se muestra en la Figura 2.5.

Figure 2.5: Diagrama de estados de un proceso. Fuente: (jesustorres, 2024)

Entre las funciones principales de la gestión de procesos se encuentra la creación y finalización de procesos. El sistema operativo se encarga de iniciar nuevos procesos mediante llamadas al sistema (por ejemplo, `fork` en Unix) y de eliminar aquellos que ya han concluido su ejecución (Juan jose costa prats, 2015).

Otra función esencial es la planificación o *scheduling*, que consiste en decidir el orden en el que los procesos acceden a la CPU. Para ello se emplean algoritmos como FIFO, round robin o planificación basada en prioridades, según los objetivos de rendimiento y equidad del sistema (Juan jose costa prats, 2015).

Asimismo, la gestión de procesos incluye mecanismos de sincronización y comunicación, conocidos como IPC (Inter-Process Communication). Estos permiten que los procesos se coordinen y compartan información de manera segura, evitando condiciones de carrera y conflictos al acceder a recursos comunes (Wikipedia, 2023b).

2.3.2 Gestión de memoria

La gestión de memoria es esencial para que el sistema operativo asigne y libere espacio en la memoria principal (RAM) a los procesos en ejecución. Su función principal consiste en aislar los espacios de direcciones de cada programa, de modo que puedan ejecutarse de manera concurrente sin interferir entre sí. Gracias a este control, la CPU puede cargar en memoria las instrucciones y datos necesarios de cada proceso en el momento oportuno. En términos generales, el administrador de memoria organiza los procesos de tal manera que se obtenga la máxima utilidad del espacio disponible, trasladando a la memoria principal la información que debe ejecutarse en cada instante (Wikipedia, 2023a).

En los sistemas modernos esta gestión se fundamenta en el concepto de memoria virtual, que proporciona a cada proceso una memoria lógica mayor que la memoria física disponible. Para ello se emplean técnicas como la paginación, en las que cada proceso dispone de su propia tabla de páginas que mapea direcciones lógicas hacia marcos de la memoria física. En la Figura 2.6 se ilustra este esquema, donde las páginas virtuales de los procesos se corresponden con bloques de memoria física compartida de manera ordenada.

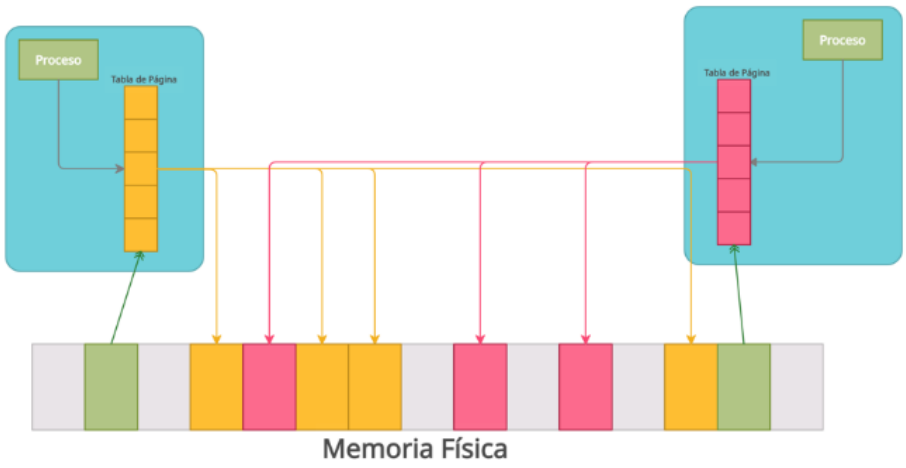

Figure 2.6: Traducción por paginación. Cada proceso (cajas turquesa) posee su propia tabla de páginas (columnas amarilla y rosada) que asigna páginas lógicas a marcos de la memoria física (bloques coloreados). Fuente: (Wikipedia, 2023)

Entre las funciones principales del gestor de memoria se encuentran el control del uso de la memoria, la asignación y liberación de espacio para los procesos, la reubicación de datos en caso necesario, la gestión de la fragmentación y la compactación, la implementación de la memoria virtual y la protección de los espacios de direcciones

para evitar accesos indebidos entre procesos. Estas tareas en conjunto garantizan un uso eficiente y seguro del recurso más limitado y compartido del sistema.

La traducción mediante paginación, como se aprecia en la figura, permite que cada proceso perciba su espacio de direcciones como contiguo, aunque en la práctica las páginas estén distribuidas en diferentes marcos físicos de la RAM. El sistema operativo se encarga de mantener actualizadas las tablas de páginas y, cuando es necesario, puede intercambiar páginas entre la memoria principal y el disco, asegurando así la abstracción de una memoria virtual continua (jjruz, 2020).

2.3.3 Sistema de archivos

2.3.3.1 Estructura jerárquica

El sistema de archivos organiza los archivos y directorios en forma de árbol invertido. A nivel lógico existe un único directorio raíz (/), nodo principal que contiene todos los demás. Cada nodo del árbol corresponde a un directorio que puede contener subdirectorios, archivos normales o especiales. Este diseño jerárquico con nombres únicos por directorio facilita la navegación y la gestión, evitando colisiones de nombres.

2.3.3.2 Gestión de datos persistentes

El sistema de archivos provee los medios para almacenar y recuperar datos de forma ordenada en la memoria secundaria (discos, SSD, etc.). La memoria se divide en bloques de tamaño fijo, y el sistema de archivos asigna a cada archivo los bloques necesarios. Además, controla la consistencia de los datos mediante técnicas como journaling, asegurando que la información persista aunque finalicen los procesos o

falle el sistema. También gestiona la integridad ante fallos, permitiendo recuperar estructuras de disco sin pérdida de datos (rguirado, 2019).

2.3.3.3 Operaciones de archivo

El sistema de archivos ofrece llamadas básicas para crear, abrir, leer, escribir, renombrar y eliminar archivos y directorios. Por ejemplo, al invocar `open`, el kernel devuelve un descriptor de archivo que apunta a una entrada en la tabla de archivos abiertos y al inodo correspondiente. Según Bach (1986), esta tabla relaciona descriptores, entradas de acceso y estructuras de inodos internos. En otras palabras, cada proceso maneja un descriptor representado como un entero, mientras que el sistema mantiene internamente una tabla que referencia los inodos de los archivos abiertos (rguirado, 2019).

2.3.3.4 Asignación de espacio

Los datos de cada archivo se almacenan en bloques de disco. El sistema de archivos controla qué bloques están libres y asigna los necesarios a cada archivo. En el esquema FAT (File Allocation Table), por ejemplo, cada disco dispone de una tabla con una entrada por bloque; cada entrada indica el siguiente bloque del archivo o un marcador especial que puede señalar bloque libre, bloque defectuoso o fin de archivo.

2.3.3.5 Seguridad y atributos

El sistema de archivos gestiona los atributos (metadatos) de cada archivo, así como la aplicación de permisos. En sistemas Unix, cada archivo se describe mediante un inodo que almacena permisos de acceso, propietario (UID/GID), fechas de modificación y

tipo de archivo (Departamento de lenguajes y computacion, 2005).

2.3.3.6 Journaling

El journaling es un mecanismo que permite implementar transacciones en los sistemas de archivos. También se conoce como registro por diario, ya que mantiene un journal donde se almacena la información necesaria para restablecer los datos afectados en caso de fallo. Por ejemplo, ante un corte de energía, al reiniciar se reproduce el journal para completar o deshacer operaciones, restaurando así la consistencia. Gracias a este enfoque, los sistemas de archivos pueden volver rápidamente a producción con menor riesgo de corrupción (wikipedia, 2023).

2.3.4 Gestión de dispositivos (E/S)

El sistema de entrada y salida del sistema operativo funciona como interfaz entre el hardware de los dispositivos (discos, teclados, impresoras, etc.) y el software, ocultando las particularidades de cada dispositivo al resto del sistema (jesustorres, 2024).

Sus componentes principales son:

- Controladores de dispositivo (drivers). Son programas específicos, normalmente provistos por el fabricante, que conocen los detalles del hardware y traducen las peticiones genéricas del sistema operativo en operaciones concretas sobre el dispositivo (jesustorres, 2024).

- Interfaz genérica de E/S. El sistema operativo ofrece llamadas estándar como `open`, `read`, `write`, `close` o `ioctl`, de modo que los programas interactúan con

los dispositivos sin necesidad de conocer sus características físicas. En sistemas UNIX, todos los dispositivos de E/S se representan como archivos dentro del directorio `/dev`, lo que permite operaciones uniformes (jesustorres, 2024).

- Buffering y caching. El buffering compensa diferencias de velocidad durante transferencias de datos en curso, mientras que el caching se centra en la reutilización de datos accedidos recientemente, anticipando futuras solicitudes para acelerar accesos posteriores.

- Spooling. En dispositivos secuenciales no compartibles, como impresoras, el sistema operativo encola los trabajos de varios procesos en un almacenamiento intermedio, gestionándolos en orden sin bloquear a los procesos que los enviaron (jesustorres, 2024).

En conjunto, la gestión de dispositivos permite a los procesos leer y escribir en hardware diverso mediante una interfaz uniforme, mientras el sistema operativo optimiza el flujo de datos y protege el acceso concurrente.

2.3.5 Interfaz de usuario

La interfaz de usuario es el medio a través del cual las personas interactúan con el sistema operativo.

Figure 2.7: Evolución de las interfaces de usuario (wikipedia, 2016)

Las interfaces de usuario han evolucionado en diferentes formas de interacción:

- CLI (Command Line Interface). El usuario escribe comandos en una consola o terminal.
- GUI (Graphical User Interface). Interfaz gráfica con ventanas, iconos y menús, más intuitiva y exploratoria.
- NUI (Natural User Interface). Interacción más directa e intuitiva, basada en mecanismos como el tacto, la voz o los gestos.

Un buen diseño de interfaz busca usabilidad e intuición, permitiendo al usuario dar órdenes al sistema operativo y visualizar información de estado. Esto incluye terminales, escritorios, menús de configuración o iconos de carpeta. El estilo de interfaz puede variar mucho entre sistemas, desde texto puro en algunos Unix básicos hasta entornos gráficos completos en sistemas modernos.

2.3.6 Seguridad y protección

La seguridad en un sistema operativo busca garantizar que solo los usuarios autorizados utilicen los recursos y lo hagan de la manera adecuada. Uno de los principios fundamentales es el de menor privilegio: cada proceso recibe únicamente los permisos

estrictamente necesarios. La autenticación permite identificar al usuario, generalmente mediante contraseña, y asignar permisos correspondientes. En sistemas como Multics, un servicio de autenticación asocia cada proceso a un usuario; si este servicio falla, un proceso podría recibir permisos indebidos al quedar vinculado con un usuario equivocado (trent jaeger, 2008).

2.3.6.1 Control de acceso

El sistema operativo aplica un modelo de control de acceso que asigna permisos a los distintos recursos (archivos, dispositivos, regiones de memoria, etc.). Cada solicitud de un proceso se comprueba mediante un monitor de referencia, que valida las operaciones de acuerdo con los permisos asignados. De manera conceptual, esta verificación puede representarse como una matriz de acceso, donde las filas corresponden a procesos o dominios, las columnas a objetos del sistema y las celdas a los permisos de lectura, escritura o ejecución.

En Unix, por ejemplo, cada archivo tiene bits de permiso y un propietario, siendo este último quien puede modificar los bits de acceso del archivo (trent jaeger, 2008).

2.3.6.2 Aislamiento de procesos y protección de memoria

El sistema operativo protege la memoria y el contexto de cada proceso para impedir interferencias entre ellos. Cada proceso se ejecuta en modo usuario (no privilegiado) dentro de su propio espacio virtual de direcciones. De esta manera, ningún proceso puede leer o escribir directamente en la memoria de otro, ni en la memoria del núcleo. La protección de memoria asegura que los datos de un usuario no puedan ser accedidos por otro y que el código crítico del kernel permanezca intacto.

Un ejemplo de este mecanismo es la arquitectura de privilegios o *rings*, donde las instrucciones sensibles solo se permiten en modo kernel (nivel 0). En este nivel privilegiado se ejecutan las operaciones críticas sobre memoria y dispositivos (Computer Science Department, 2018).

2.3.6.3 Prevención de intrusiones

Además de la protección interna, el sistema operativo incorpora mecanismos activos frente a amenazas externas o internas. Un sistema de prevención de intrusiones (IPS) supervisa el tráfico y la actividad en busca de comportamientos anómalos, ya sea mediante firmas conocidas o por detección de desviaciones. Si se identifica una amenaza, el IPS puede bloquearla antes de que cause daño, por ejemplo cerrando conexiones peligrosas o eliminando contenido malicioso.

Estos mecanismos suelen incluir filtros y cortafuegos integrados en el sistema operativo, que rechazan accesos no autorizados —ya sea por red o de manera local— según las políticas definidas. En conjunto, el sistema previene ataques como virus, intrusiones de red o escaladas de privilegios, al analizar cada petición de acceso y denegarla cuando coincide con patrones maliciosos (IBM, 2019).

2.3.6.4 Auditoría y registro

Con el fin de detectar incidentes y comportamientos sospechosos, el sistema operativo mantiene registros de auditoría de eventos relevantes: inicios y cierres de sesión, errores de autenticación, intentos fallidos de acceso, cambios en permisos, entre otros. Cada evento se almacena con la fecha, el usuario y la acción realizada, lo que permite revisar los logs para identificar accesos indebidos o intrusiones después de que

ocurran.

Los sistemas de auditoría más avanzados incluso generan alertas automáticas al equipo de seguridad cuando detectan anomalías en el comportamiento (IBM, 2019).

2.3.6.5 Modelos de seguridad clásicos

En sistemas operativos con seguridad reforzada también se emplean modelos formales que definen con rigor las reglas de acceso. El modelo Bell–LaPadula se centra en la confidencialidad, aplicando el principio de "no leer hacia arriba, no escribir hacia abajo" (*read down, write up*). Con ello, un sujeto solo puede leer información de igual o menor nivel de clasificación y escribir en niveles iguales o superiores, evitando fugas de datos sensibles.

Por otro lado, el modelo Biba protege la integridad mediante las reglas opuestas: "read up, write down". Así se evita que un proceso de baja integridad contamine a uno de mayor integridad, o que datos críticos se vean corrompidos por información poco confiable.

En la práctica, estos modelos, junto con variantes como Clark–Wilson, proporcionan una base sólida para garantizar tanto la confidencialidad como la integridad en sistemas multigrado (trent jaeger, 2008).

Chapter 3

Revision de S.O. existentes

3.1 RedoxOs

Figure 3.1: Logo del sistema operativo Redox OS (Wikipedia contributors, 2025)

Redox es un sistema operativo tipo Unix que está escrito en el lenguaje de programación Rust, con el objetivo de implementar un microkernel y un sistema de aplicaciones. Rust es un lenguaje enfocado en la seguridad, rendimiento, gratuito y fácil de navegar. Redox se enfocó en la mejora de varios sistemas anteriores que presentaban errores. Uno de sus proyectos es Redox OS. Es compatible con POSIX, y la comunidad está trabajando para escribir la biblioteca libc en Rust, llamada relibc (Saini, 2018).

Dado que para los sistemas operativos es muy importante la seguridad, porque los sistemas operativos tienen un alto nivel de abstracción sobre los recursos del sistema, más aún en el caso de que Linux tuviera muchos errores en diferentes bibliotecas ocasionados por la seguridad de memoria. Rust en cambio evita todos esos problemas al tener seguridad de memoria en tiempo de compilación. El kernel de Redox contiene más de 20 mil líneas de código, y su diseño es de alto nivel por eso aún puede tener problemas (Ellmann and Emmerich, 2019).

El sistema operativo Redox OS se apoya en un microkernel y está inspirado en el sistema operativo MINIX. Las funciones tradicionales se implementan en kernels monolíticos, como controladores de dispositivo, pilas y el sistema de archivos. Los

microkernels son más seguros y ofrecen mayor estabilidad, pero una de las principales desventajas es que el rendimiento que ofrecen no está optimizado para la mayoría del hardware actual (Ritter, 2019).

En cuanto a sus componentes clave, Redox OS integra un microkernel responsable de los procesos del sistema, un **sistema de archivos** implementado en espacio de usuario, **drivers** que también se ejecutan en espacio de usuario para incrementar la estabilidad, y compatibilidad POSIX que permite correr aplicaciones de otros sistemas Unix. También, añade **la biblioteca relibc**, escrita en Rust para mejorar la seguridad y la compatibilidad, ademas de **gestión de memoria** segura que evita vulnerabilidades comunes como los desbordamientos(The Redox OS Community, 2025).

3.2 Mac OS

Figure 3.2: Logo del sistema operativo mac OS (GQ Informática, 2023)

El proyecto Mach dio comienzo en 1985 en la universidad de Carnegie Mellon con la finalidad de crear un microkernel que reemplace al BSD. Su diseño buscaba más seguridad para los usuarios, nose genero lo resultados esperados y se cancelo en 1994. Entonces el proyecto fue adoptado pro OSF/1 Y NeXTSTEP por lo que completo al combinar los componente de BSD, dando origen al kernel XNU de tipo monolítico, pero presento limitaciones con el System 7 y fracaso el proyecto, por ultimo este último avance fue adquirida por Steve Jobs, lo que permitió integrar NeXTSTEP y su kernel XNU como nuevo sistema operativo a Mac OS X, lanzado en 24 marzo del 2001 con el kernel de XNU, que integraba Mach 3.0 y el marco I/O kit, además de ser integrado en Intel Y ARM (Keuper, 2012). El kernel de XNU prosee una arquitectura hibrida que tiene 4 componentes clave **Mach** que se encarga de administrar las tares, que contine hilos. Su caracteristica principal es la mensajería, mediante puntos de comunicación (IPC), usa el Mach Interface Generator (MIG) para acortar lo procesos, tenemos tambien **BDS** que implementa procesos y señales UNIX, adema de sistema de archivos y redes, **I/O Kit** es el marco de controladores orientado a objetos y por ultimo **KEXTs** que encargar de enlazar no solo con el I/O Kit, sino tambien con

otras partes del kernel, extensión de red, sistema de archivos (Levin, 2007). Varios de los primeros trabajos de software abierto de Apple son Darwin y WebKit. Darwin es un sistema operativo público que Apple introdujo en el año 2000. Este sistema fue liberado bajo la Licencia Pública de Apple (APSL), que ha sido aceptada por la OSI y la FSF (Levin, 2007)

3.3 FreeRTOS

Figure 3.3: Logo del sistema operativo FreeRTOS (Llamas, 2025)

FreeRTOS es un sistema operativo en tiempo real con un kernel planificador, desarrollado para ejecutarse con microcontroladores. Está compuesto funcionalidades de planificación en tiempo real, comunicación entre tareas, análisis temporal y primitivas de sincronización con el propósito de cumplir con las tareas con el plazos estrictos de ejecución (He and Huang, 2020). En cuanto a su arquitectura hace uso de un microkernel para administrar tareas en tiempo real, casi similar al RTLinux en que admiten núcleo dual, lo que permita separar tareas critica y no criticas (Serino and Cheng, 2019) Esta escrito en lenguaje C estándar y con algunas lineas de código en ensamblador para que se acople a diferentes arquitecturas. Dentro de su componentes claves tenemos el **planificador**, **mecanismos de comunicación**, **sincronización** gracias al uso de colas y semáforos. Incluyendo la estructura de **Task Control Block (TCB)** para la administración de procesos y tareas (Barry, 2018).

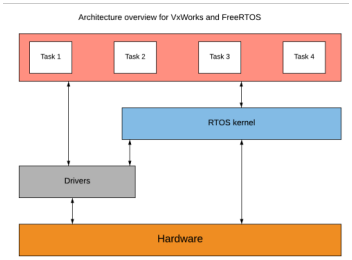

Figure 3.4: Diseño de la arquitectura de FreeRTOS (Llamas, 2025)

FreeRTOS es código abierto con una licencia GPL, lo que permite su uso en aplicaciones comerciales. El código está disponible públicamente y cuenta con documentación extensa en el código fuente en su sitio oficial (The FreeRTOS Project, 2025).

3.4 Linux

Figure 3.5: Logo del núcleo Linux (wikipedia, 2022)

Linux es un sistema operativo de tipo Unix iniciado en 1991 por Linus Torvalds como un kernel monolítico, modular y multitarea. Originalmente escrito para PCs i386, ha evolucionado hasta ofrecer compatibilidad POSIX y soporte extenso de hardware, siendo desarrollado bajo el modelo de código abierto, donde las mejoras provienen de contribuyentes individuales y corporativos, mientras que la dirección general la marca la comunidad y no un único proveedor (wikipedia, 2022; oracle latam, 2024). Su arquitectura se basa en un kernel monolítico moderno que integra características modulares. Todo el núcleo, incluidos los controladores de dispositivos, se ejecuta en un único espacio de direcciones en modo kernel. Sin embargo, soporta la carga y descarga dinámica de módulos en tiempo de ejecución, lo que le otorga flexibilidad similar a la de los micronúcleos sin sacrificar el rendimiento propio de un diseño monolítico (Love, 2010, pp. 7). El núcleo de Linux está escrito principalmente en C, con secciones críticas en ensamblador para optimizar la interacción directa con el hardware. Las aplicaciones en espacio de usuario, en cambio, pueden desarrollarse en diversos lenguajes como C, C++, Python o Rust.

Entre sus componentes principales se encuentran la gestión de memoria con asignación y protección, la planificación de procesos que regula el uso de la CPU, los controladores de dispositivos para el hardware, el sistema de llamadas al sistema con sus mecanismos de seguridad, y el sistema de archivos virtual (VFS), encargado de unificar múltiples formatos y módulos adicionales (oracle latam, 2024).

Linux posee una de las comunidades de código abierto más amplias y activas del mundo, integrada por miles de desarrolladores pertenecientes a empresas como Red Hat, Google, Intel o IBM, así como por colaboradores independientes. La documentación es igualmente extensa e incluye las `man pages`, el proyecto de documentación oficial del kernel (`https://docs.kernel.org/`), wikis como la Arch Wiki, foros especializados y numerosos libros técnicos.

3.5 MINIX 3

Figure 3.6: Estructura de MINIX 3 (usenix, 2010)

MINIX fue creado en 1987 por Andrew Tanenbaum como herramienta docente para la enseñanza de sistemas operativos. En 2005 se anunció MINIX 3, una reimplementación centrada en la fiabilidad y la modularidad, cuyo propósito principal es ofrecer un sistema operativo compatible con POSIX, recuperable ante fallos y adecuado tanto para aplicaciones críticas como para entornos educativos. A diferencia de otros sistemas, fue escrito completamente desde cero, sin incluir código de AT&T, manteniendo compatibilidad con la versión 7 de Unix y los estándares POSIX (usenix, 2010, pp. 10).

Su arquitectura está basada en un microkernel muy reducido que se encarga únicamente de interrupciones, gestión básica de procesos y comunicación mediante IPC. Todo lo demás se ejecuta en el espacio de usuario como servidores independientes: controladores de dispositivos, gestor de archivos, gestor de procesos y gestor de memoria. Este diseño multiserver permite que cada componente esté aislado, de modo que un fallo en un servidor no comprometa al sistema completo. Incluso cuenta con un servidor de reencarnación que reinicia automáticamente los componentes de-

fectuosos para mejorar la tolerancia a fallos (Jorrit N. Herder and Herbert Bos and Jakob Lichtenberg and Peter M. Chen and Andrew S. Tanenbaum, 2007).

MINIX 3 está escrito principalmente en C, con secciones en ensamblador dedicadas a la inicialización del hardware y a rutinas críticas de bajo nivel.

Entre sus componentes principales se encuentran la gestión de procesos, a cargo de un proceso supervisor que controla creación, terminación y recolección de fallos; la gestión de memoria, donde el microkernel provee segmentación y paginación básica mientras un servidor en espacio de usuario administra la memoria virtual; el servidor de archivos (Mini-FS) con soporte POSIX; controladores de dispositivos que se ejecutan como procesos de usuario aislados; y soporte para interfaces gráficas como X11 o entornos ligeros tipo EDE, siguiendo la tradición Unix (usenix, 2010, pp. 12).

La comunidad de MINIX 3, aunque más pequeña en comparación con otros proyectos de código abierto, ha tenido impacto en el ámbito académico. Hasta 2010 su web oficial registraba cerca de 1.7 millones de visitas y más de 300,000 descargas de CD [1]. El proyecto participó en Google Summer of Code entre 2008 y 2010, y mantiene recursos como wiki, grupos de discusión y repositorios en GitHub. Su documentación se apoya en el libro *Operating Systems: Design and Implementation* de Tanenbaum (3ª edición), artículos académicos y materiales de congresos, lo que lo convierte en una referencia educativa sólida en el campo de los sistemas operativos.

[1] usenix.org

3.6 xv6

Figure 3.7: Logo de xv6 (Frans Kaashoek and Robert Morris, 2016)

xv6 es un sistema operativo educativo inspirado en Unix V6, desarrollado en 2006 por el MIT para su curso de sistemas operativos. Su propósito fue proporcionar un kernel moderno y simple que sustituyera al Unix V6 original, cuya complejidad dificultaba la enseñanza. xv6 sigue de cerca la estructura de Unix V6, pero está escrito en C ANSI y se ejecuta en arquitecturas Intel multiprocesador, convirtiéndose en una herramienta práctica y comprensible para la docencia (Frans Kaashoek and Robert Morris, 2016).

Su arquitectura corresponde a un kernel monolítico minimalista, en el que todo el núcleo del sistema operativo se ejecuta con privilegios de kernel, sin separación en servidores independientes. xv6 incluye soporte para multiprocesadores mediante bloqueos y estructuras internas de hilos, aunque carece de controladores de red o capacidades gráficas avanzadas. Sus componentes básicos abarcan la planificación de procesos con un esquema round-robin, la gestión de memoria con paginación simple, un sistema de archivos inspirado en Unix V6 y un conjunto reducido de llamadas al

sistema (Cox et al., 2022, pp. 25).

La implementación de xv6 está escrita principalmente en C, mientras que el código de inicio y las rutinas de interrupción se desarrollan en ensamblador x86. En sus versiones más recientes, también se encuentra una adaptación para la arquitectura RISC-V, lo que amplía su aplicabilidad en contextos educativos.

Entre los componentes principales destacan la gestión de procesos, que otorga a cada uno su propio espacio de direcciones, con un planificador round-robin y un cambio de contexto sencillo; la gestión de memoria mediante paginación básica y un asignador simple para el kernel; un sistema de archivos con inodos y directorios inspirado en Unix V6; y mecanismos elementales de sincronización como `sleep` y `wakeup`, que permiten estudiar las condiciones de carrera. Además, xv6 expone llamadas al sistema tradicionales de UNIX como `fork`, `exit`, `wait`, `open`, `read`, `write` y `close`, lo que facilita comprender la implementación exacta de estas primitivas fundamentales (Cox et al., 2022, pp. 26–30).

xv6 se ha consolidado como una herramienta académica ampliamente utilizada en universidades para cursos de sistemas operativos. Su código fuente está disponible públicamente, y los propios autores distribuyen un libro titulado *xv6: a simple, Unix-like teaching operating system*, que comenta línea por línea el código y explica tanto el funcionamiento como el diseño de sus estructuras y funciones. Gracias a su claridad y abundante documentación, xv6 se ha convertido en uno de los sistemas operativos de referencia para la enseñanza de fundamentos prácticos de diseño de kernels (Cox et al., 2022).

3.7 Fuchsia

Figure 3.8: Icono de S.O. de Fuchsia (Fuchsia Project, 2025)

Fuchsia OS es un sistema operativo desarrollado por Google desde 2016, diseñado como una plataforma de propósito general para dispositivos embebidos, móviles, IoT y potencialmente de escritorio. A diferencia de Android, no se basa en Linux, sino en un núcleo propio denominado Zircon (Google Fuchsia Team, 2025b).

Su arquitectura está basada en un microkernel, en el que Zircon gestiona procesos, hilos, memoria y comunicación mediante objetos e IPC, mientras que servicios como sistemas de archivos y controladores se ejecutan en espacio de usuario, favoreciendo la seguridad y modularidad (Google Fuchsia Team, 2025a).

Fuchsia está escrito principalmente en C++, Rust y Dart. Zircon se implementa en C/C++, mientras que la capa de interfaz gráfica se desarrolla con Flutter (Dart), lo que facilita la portabilidad a múltiples plataformas (Google Inc., 2024).

Entre sus componentes clave se encuentran la gestión de procesos basada en objetos, memoria virtual con aislamiento estricto, soporte para múltiples sistemas de archivos (como MemFS, BlobFS y FAT) y un entorno gráfico con Flutter para experiencias multiplataforma (Google Fuchsia Team, 2025b).

El proyecto es mantenido principalmente por Google, con participación de la comunidad de código abierto. Su documentación oficial está disponible en Fuchsia.dev, que incluye guías técnicas, API y manuales de contribución, aunque parte del desarrollo se mantiene cerrado (Fuchsia Open Source Community, 2025).

3.8 Haiku

Figure 3.9: Icono de S.O. de Haiku (Haiku Project, 2025b)

Haiku es un sistema operativo de código abierto inspirado en BeOS, iniciado en 2001 con el objetivo de proporcionar un entorno moderno, rápido y sencillo de usar, orientado principalmente a equipos de escritorio (Haiku Project, 2025a).

Su arquitectura es híbrida, ya que utiliza un núcleo monolítico modular, pero con elementos que recuerdan a un microkernel, lo que equilibra rendimiento y flexibilidad (Haiku Project Developers, 2024). El sistema está desarrollado principalmente en C++, con partes en C y ensamblador para las interacciones de bajo nivel (Haiku Project Developers, 2024).

Entre sus componentes clave se incluyen un modelo de multitarea preventiva con planificación por prioridades, gestión de memoria virtual con paginación y aislamiento de procesos, y un sistema de archivos propio llamado OpenBFS, optimizado para indexación rápida (Haiku Project, 2023). Además, cuenta con una interfaz gráfica uniforme gestionada por el app_server.

Haiku es mantenido por la Haiku Project Foundation y su comunidad de desarrolladores voluntarios. El proyecto dispone de documentación oficial, foros y repositorios activos en GitHub, con lanzamientos beta periódicos que permiten probar y evaluar su madurez (Haiku Project Community, 2025).

Chapter 4

Comparación técnica

Bibliography

Barry, R. (2018). *Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide*. Real Time Engineers Ltd. For FreeRTOS V10.0.0. Accedido el 20 de septiembre de 2025.

Computer Science Department (2018). Protection and Virtual Memory. Accedido: 17 de septiembre de 2024.

Cox, R., Kaashoek, F., and Morris, R. (2022). xv6: a simple, unix-like teaching operating system. Accedido: 17 de septiembre de 2025.

Cusme Vera, R. J., Fuentes Prado, S. N., Bravo Vega, C. D., and Gualotuña Quinga, G. B. (2022). Sistemas operativos libres y sistemas operativos privativos: aplicaciones en el campo educativo. *Revista Científica Multidisciplinaria Ogma*, 1(3):85–97.

Departamento de lenguajes y computacion (2005). tema 4: sistema de archivos. Accedido: 17 de septiembre de 2024.

Ellmann, S. and Emmerich, P. (2019). Porting ixy.rs to redox. Technical report, Technical University of Munich. Accedido el 20 de septiembre de 2025.

Engler, D. R., Kaashoek, M. F., and O'Toole, James, J. (1995). Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Cambridge, MA, USA. ACM.

Frans Kaashoek and Robert Morris (2016). Xv6, a simple Unix-like teaching operating system. Accedido: 16 de septiembre de 2025.

Fuchsia Open Source Community (2025). Community involvement in fuchsia. Foros y repositorios. Accedido: 20 septiembre 2025.

Fuchsia Project (2025). Fuchsia — open source operating system. Sitio web oficial. Accedido: 20 septiembre 2025.

GeeksforGeeks (2024a). Microkernel in operating systems. Accedido: 20 septiembre 2025.

GeeksforGeeks (2024b). Monolithic architecture in operating systems. Accedido: 20 septiembre 2025.

Google Fuchsia Team (2025a). Fuchsia concepts and apis. Documentación técnica. Accedido: 20 septiembre 2025.

Google Fuchsia Team (2025b). Zircon kernel — fuchsia documentation. Documentación técnica. Accedido: 20 septiembre 2025.

Google Inc. (2024). Flutter and dart in fuchsia. Blog técnico de Google. Accedido: 20 septiembre 2025.

GQ Informática (2023). macos, información completa. Accedido el 20 de septiembre de 2025.

Haiku Project (2023). Openbfs — be file system in haiku. Documentación técnica. Accedido: 20 septiembre 2025.

Haiku Project (2025a). Haiku — official website. Sitio web oficial. Accedido: 20 septiembre 2025.

Haiku Project (2025b). Haiku — the open source operating system. Sitio web oficial. Accedido: 20 septiembre 2025.

Haiku Project Community (2025). Haiku community and development. Foros y repositorios. Accedido: 20 septiembre 2025.

Haiku Project Developers (2024). Haiku developer documentation. Documentación técnica. Accedido: 20 septiembre 2025.

Harshvardhan and Irabatti, S. (2023). Study of kernels in different operating systems in mobile devices. *Journal of Online Engineering Education*, 14(1s). Article received: 29 January 2023; Revised: 24 March 2023; Accepted: 20 April 2023.

He, N. and Huang, H.-W. (2020). USE OF FreeRTOS IN TEACHING A REAL-TIME EMBEDDED SYSTEMS DESIGN COURSE. *Computers in Education Journal*, XI(2). Accedido el 20 de septiembre de 2025.

IBM (2019). ¿Qué es un sistema de prevención de intrusiones (IPS)? Accedido: 17 de septiembre de 2024.

jesustorres (2024). Sistemas Operativos. Online; accessed 20-June-2024.

jjruz (2020). tema 7: memoria virtual. Accedido: 17 de septiembre de 2024.

Jorrit N. Herder and Herbert Bos and Jakob Lichtenberg and Peter M. Chen and Andrew S. Tanenbaum (2007). The Architecture of a Reliable Operating System. Accedido: 16 de septiembre de 2024.

Juan jose costa prats (2015). gestion de procesos. Accedido: 17 de septiembre de 2024.

Keetmalin (2017). An introduction on exokernel operating systems. Blog post. Accedido: 20 septiembre 2025.

Keuper, D. (2012). XNU: a security evaluation. Master's thesis, University of Twente. Accedido el 20 de septiembre de 2025.

Levin, J. (2007). Inside the Mac OS X Kernel. In *Proceedings of the 24th Chaos Communication Congress (24C3)*. Accedido el 20 de septiembre de 2025.

Llamas, L. (2025). Freertos cheatsheet, la chuleta con lo imprescindible. Accedido el 20 de septiembre de 2025.

Love, R. (2010). *Linux Kernel Development*. Addison-Wesley Professional, third edition.

Microsoft Wiki / Fandom (2024). Hybrid kernel. Accedido: 20 septiembre 2025.

oracle latam (2024). ¿Qué es Linux? Accedido: 16 de septiembre de 2024.

rguirado (2019). tema 7: sistema de archivos. Accedido: 17 de septiembre de 2024.

Ritter, T. (2019). Disk encryption in redox os. Master's thesis, Masaryk University, Faculty of Informatics. Accedido el 20 de septiembre de 2025.

Saini, R. (2018). Priority based scheduler in rust based redox operating system. Master's thesis, Indian Institute of Technology Madras. Accedido el 20 de septiembre de 2025.

Serino, A. and Cheng, L. (2019). A Survey of Real-Time Operating Systems. Technical Report LU-CSE-19-003, Lehigh University, Department of Computer Science and Engineering, Bethlehem, PA. Accedido el 20 de septiembre de 2025.

Stallings, W. (2023). *Computer Organization and Architecture: Designing for Performance*. Pearson, 12th edition.

Tanenbaum, A. S. and Bos, H. (2023). *Modern Operating Systems*. Pearson, 5th edition.

The FreeRTOS Project (2025). FreeRTOS - Market leading RTOS for embedded systems. Accedido el 20 de septiembre de 2025.

The Redox OS Community (2025). The Redox OS Book. Accedido el 20 de septiembre de 2025.

trent jaeger (2008). operating system security. Accedido: 17 de septiembre de 2024.

usenix (2010). MINIX 3: status report and current research. Accedido: 16 de septiembre de 2024.

wikipedia (2016). Archivo:CLI-GUI-NUI, evolución de interfaces de usuario. Accedido: 17 de septiembre de 2024.

wikipedia (2022). nucleo linux. Accedido: 17 de septiembre de 2024.

Wikipedia (2023a). Gestion de memoria. Accedido: 17 de septiembre de 2024.

Wikipedia (2023b). Inter-process communication. Accedido: 17 de septiembre de 2024.

wikipedia (2023). journaling. Accedido: 17 de septiembre de 2024.

Wikipedia (2023). tabla de paginacion. Accedido: 17 de septiembre de 2024.

Wikipedia contributors (2025). Redox (operating system) — Wikipedia, The Free Encyclopedia. [Online; accessed 20-September-2025].