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

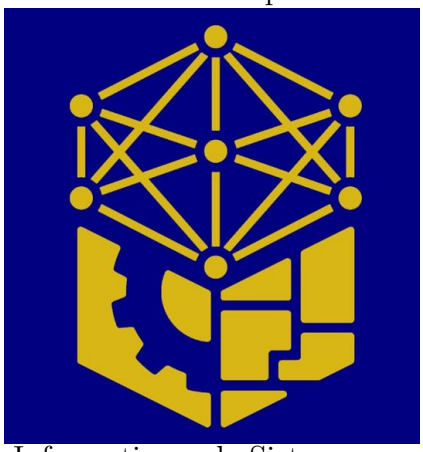
Polo Chura, Marco Rosauro

Trabajo académico presentado a la

Facultad de Ingeniería Eléctrica, Electrónica, Informática y Mecánica

como

Proyecto de Investigación de la Segunda Unidad de Sistemas Operativos



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

Índice general

Índice de tablas

Índice de figuras

Introducción

Para desarrollar un sistema operativo desde sus bases es indispensable conocer, con suficiente detalle, las distintas estructuras y mecanismos que hacen posible su funcionamiento. Más allá de la teoría, resulta especialmente útil examinar sistemas operativos reales —muchos de ellos abiertos y creados con fines educativos o de investigación— que exploran diferentes arquitecturas, modelos de aislamiento y niveles de complejidad.

En este informe se estudian diversas propuestas existentes, entre ellas MINIX, XV6, Theseus, RedoxOS, FlexOS, LibrettOS, FreeRTOS, HelenOS, Haiku OS y FreeDOS. Lo que se busca es aprender de las decisiones de diseño que presentan estos proyectos, contrastar sus propuestas y ver cuáles resultan realmente útiles como referencia para construir un sistema operativo con fines académicos.

Objetivo de la investigación

La idea central del estudio es revisar y comparar técnicamente varios sistemas operativos que responden a diferentes arquitecturas y enfoques de diseño, prestando particular atención a los proyectos de código abierto que se usan para enseñar o

investigar.

Alcance del análisis

Este estudio se limita a sistemas operativos que, por su licencia, documentación y propósito, pueden emplearse adecuadamente como referencia en la construcción de un sistema operativo académico. En particular, se revisan los siguientes proyectos: MINIX, XV6, Theseus, RedoxOS, FlexOS, LibrettOS, FreeRTOS, HelenOS, Haiku OS y FreeDOS, cada uno abordado en el Capítulo 1 bajo un esquema de análisis uniforme.

El alcance comprende únicamente:

1. La revisión de documentación oficial, artículos académicos y repositorios de código.

2. Analizar cualitativamente las decisiones de diseño que se tomaron en cada uno de los sistemas revisados.

3. Elaborar una comparación organizada y un análisis crítico que permita reconocer cuáles de estas propuestas pueden servir mejor como referencia para desarrollar un sistema operativo con fines académicos.

Metodología

Para cumplir con los objetivos planteados, se adopta una metodología dividida en tres fases principales:

1. **Revisión bibliográfica y documental:** Aquí se reúne y examina información procedente de libros especializados, artículos académicos, documentación oficial de los proyectos y repositorios de código. Gracias a esta revisión es posible entender el contexto, los objetivos iniciales y la situación actual de cada uno de los sistemas operativos analizados

2. **Evaluación técnica de cada sistema operativo:** En el Capítulo 1 se estudia cada sistema usando un mismo esquema. Se revisa para qué sirve, cómo está diseñado, qué partes tiene (procesos, memoria, archivos, dispositivos, etc.), qué herramientas se utilizan para trabajarlo, como compiladores o emuladores.y qué tan difícil o accesible es para los estudiantes.

3. **Comparación estructurada y análisis:** Con base en la información recopilada, el Capítulo 2 desarrolla una comparación general entre los sistemas analizados, evaluando criterios como arquitectura, tamaño del kernel, soporte de hardware, comunidad, documentación y su potencial para fines formativos. En el Capítulo 3 se presenta un análisis crítico donde se examinan las ventajas y limitaciones de cada propuesta, reflexionando sobre su idoneidad como punto de partida para la construcción de un sistema operativo.

Capítulo 1

Propuestas analizadas

1.1 MINIX

1.1.1 Nombre del proyecto o sistema operativo

El nombre del proyecto desarrollado por Andrew S. Tanenbaum es **MINIX**. Según (Andrew S. Tanenbaum and Albert S. Woodhull, 2006), este nombre es un acrónimo de *mini-UNIX*, su propósito fue ser una versión educativa y simplificada del sistema UNIX, diseñada para facilitar el aprendizaje de los principios fundamentales de los sistemas operativos. Este sistema se inspiro se inspiró en la UNIX Version 7 y con el tiempo fue mejorado para ajustarse al estándar internacional POSIX.

1.1.2 Enlace al repositorio y/o documentación oficial

- **Pagina oficial:** https://www.minix3.org
- **Enlace al repositorio oficial:** https://github.com/Stichting-MINIX-Research-Foundation/minix
- **Enlace al libro oficial:** https://csc-knu.github.io/sys-prog/books/Andrew%20S.%20Tanenbaum%20-%20Operating%20Systems.%20Design%20and%20Implementation.pdf

1.1.3 Objetivo del proyecto

Segun (Andrew S. Tanenbaum and Albert S. Woodhull, 2006), este sistema operativo MINIX fue diseñado para ser un sistema operativo con fines educativos. Su propósito es ser como una herramienta de laboratorio para enseñar los conceptos fundamentales como: procesos, comunicación entre procesos, semáforos, monitores, paso de

mensajes, algoritmos de programación, entrada/salida, interbloqueos, controladores de dispositivos, gestión de memoria, algoritmos de paginación, diseño de sistemas de archivos, seguridad y mecanismos de protección. MINIX generalmente se centra en lo teórico y lo practico es por ese motivos que fue uno de lo mejores sistemas operativos para aprender sobre sistemas operativos.

1.1.4 Lenguaje(s) de implementación

Segun (Andrew S. Tanenbaum and Albert S. Woodhull, 2006), MINIX fue implementado principalmente en el lenguaje de programación C, lo cual es típico para sistemas operativos debido a la conexión cercana con el hardware y la facilidad para escribir código. Durante años, algunas partes del código también han sido implementadas en ensamblador para aprovechar características específicas del hardware. En la version de MINIX 3, tan solamente consta de 40000 lineas de código ejecutables, que lo hacen mas fácil de entender y realizar modificaciones en el sistema operativo.

1.1.5 Arquitectura del sistema (monolítica, microkernel, etc.)

En el MINIX 3 se usa una arquitectura de microkernel, lo que nos dice que el núcleo o kernel se encarga de funciones mas básicas de sistema. como por ejemplo la planificación de procesos, comunicación, el manejo del hardware. Para poder entender mejor como es la arquitectura de MINIX 3, se muestra la siguiente figura 1.1.

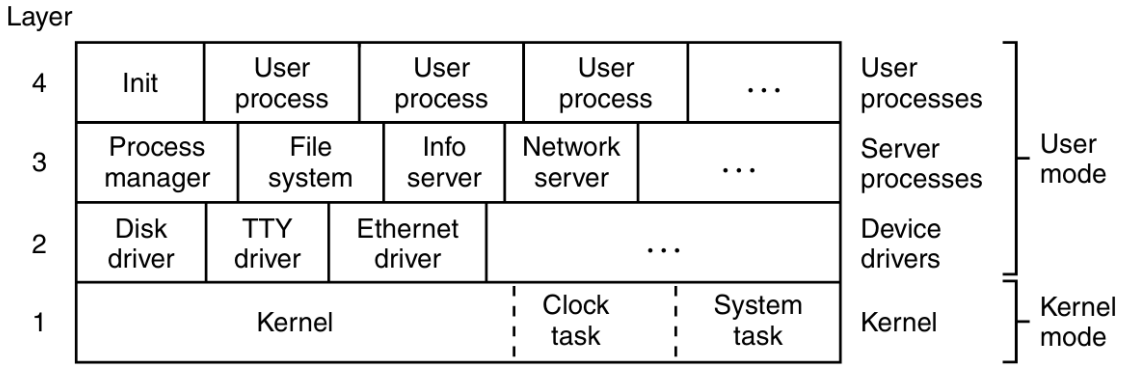


Figura 1.1: Arquitectura de MINIX 3.

Fuente: Obtenido de (Andrew S. Tanenbaum and Albert S. Woodhull, 2006, pag. 113) *OPERATING SYSTEMS DESIGN AND IMPLEMENTACIÓN.*

1.1.6 Componentes implementados (procesos, memoria, archivos, etc.)

MINIX está diseñado para ser modular, donde cada servicio (como la gestión de archivos, la planificación de procesos y los controladores de dispositivos). En resumen podemos ver en la tabla 1.1 los principales subsistemas y su funcionamiento técnico.

Tabla 1.1: Funcionamiento técnico de los principales subsistemas de MINIX 3

Componente / Subsistema	**Nombre en MINIX (archivo)**	**Funcionamiento técnico (idea clave)**
Sistema de archivos	`fs/`	Gestiona la estructura de directorios y el montaje de dispositivos mediante `mount`.
Archivos especiales	`/dev/`	Representan dispositivos como archivos de bloque o carácter.
Tuberías (pipes)	`pipe.c`	Permiten comunicación entre procesos mediante pseudoarchivos.
Shell (intérprete de comandos)	`sh/`	Ejecuta programas, redirige E/S y permite multitarea con `&`.
Llamadas al sistema	`syscall.c`	Interfaz entre programas y el sistema operativo.
Comunicación entre procesos	`ipc.c`	Controla el intercambio seguro de mensajes entre procesos.

Fuente: Elaboración propia con base en (Andrew S. Tanenbaum and Albert S. Woodhull, 2006, pag. 20–42).

Sistema de archivos

Gestiona la estructura jerárquica de archivos. Permite montar diferentes dispositivos (como CD-ROM, discos duros, disco solido, memoria flash y entre otros) en un único árbol de directorios mediante la llamada `mount`. Su trabajo es administra rutas,

directorios y accesos sin depender del dispositivo.

Archivos especiales

Modelan los dispositivos físicos como archivos. Los archivos de bloque permiten acceder a unidades de disco por bloques, mientras que los de carácter representan flujos continuos (impresoras, módems y todo dispositivos de entrada y salida). Con la finalidad de permitir operar los dispositivos usando las mismas llamadas que para archivos comunes.

Tuberías

Implementan la comunicación entre procesos mediante código en pseudoarchivos. Un proceso escribe en la tubería y otro lee, logrando sincronización y transferencia de datos sin necesidad de archivos temporales. Base del uso de tuberías en el shell.

Shell

Proporciona la interfaz principal con el usuario. Tiene como objetivo ejecuta programas, redirige entradas y salidas, conecta procesos con tuberías y permite tareas en segundo plano. Utiliza intensivamente las llamadas al sistema del núcleo.

Llamadas al sistema

Contiene la interfaz entre los programas de usuario y el sistema operativo. Incluyen operaciones para manejo de procesos y archivos. En MINIX 3, la mayoría de las llamadas siguen el estándar POSIX.

Comunicación entre procesos

Gestiona el intercambio de mensajes entre procesos del sistema y el microkernel. Esta arquitectura refuerza el aislamiento y la estabilidad, ya que los servicios del sistema operan como procesos independientes que se comunican de forma controlada.

1.1.7 Herramientas utilizadas (compiladores, emuladores, etc.)

MINIX acepta varios lengujes de programación y compliladores, lo que facilita el desarrollo de software de usuario como compilaciones con el propio sistema operativo. (The MINIX 3 Wiki, 2017)

- **Lenguajes:** Incluyes soporte para varios lenguajes como C/C++, clisp(LISP), mawk(AWK), Perl, Python y otros.
- **Compiladores:** Utiliza el famoso gcc (GNU Compiler Collection) y clang/LLVM como compiladores. Ambos permiten al compilacion nativa.
- **Arquitectura compatibles:** Esta diseñado para procesadores x86, ARM y RISC-V.
- **Emuladores:** MINIX puede ejecutarse en emuladores populares como QEMU, VirtualBox y Bochs, lo que facilita su uso en diferentes entornos de desarrollo.
- **Paquetes:** Tiene una amplia variedad de programas, con mas de 4000 paquetes disponibles, las cuales son Shells, editores de texto, juegos, correos electrónicos.

1.1.8 Nivel de complejidad y accesibilidad para estudiantes

Según (Andrew S. Tanenbaum and Albert S. Woodhull, 2006, pag. 17), MINIX 3 está especialmente enfocado en PCs más pequeños como los que se encuentran comúnmente en países en desarrollo y en sistemas integrados, que siempre tienen recursos limitados. En cualquier caso, este diseño facilita mucho a los estudiantes aprender cómo funciona un sistema operativo que intentar estudiar un sistema monolítico enorme. Por su reducido tamaño y diseño del micronucleo y su documentación, resulta facil de entender para personas que deseen instalar un sistema compatible con UNIX en sus maquinas personal (Colaboradores de Wikipedia, 2025).

1.2 XV6

1.2.1 Nombre del proyecto o sistema operativo

El sistema operativo xv6 es una versión moderna y simplificada del Unix Sexta Edición (V6), creada con fines educativos. Fue desarrollado por el MIT en 2006 para el curso de Engineering Operating Systems (6.828) (Cox, R. and Kaashoek, F. and Morris, R., 2018), con el objetivo de ofrecer una herramienta didáctica más accesible y actual que el Unix original, el cual estaba escrito en un dialecto antiguo de C (pre-ANSI). Esta reimplementación mantiene la esencia y estructura del Unix clásico, pero utiliza un código más claro y compatible con los entornos de aprendizaje contemporáneos.

1.2.2 Enlace al repositorio y/o documentación oficial

- **Pagina oficial:** https://pdos.csail.mit.edu/6.828/2018/xv6.html
- **Enlace al repositorio oficial:** https://github.com/mit-pdos/xv6-public
- **Enlace al libro oficial:** https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev10.pdf

1.2.3 Objetivo del proyecto

El propósito principal de xv6 es educativo y experimental. Fue diseñado para funcionar como un kernel de estudio que permita a los estudiantes comprender los principios fundamentales de Unix dentro de un curso semestral (Wikipedia contributors, 2024b). A diferencia de sistemas operativos más complejos como Linux o BSD, xv6 mantiene una estructura lo bastante simple como para ser analizada y entendida en

su totalidad durante una asignatura, pero al mismo tiempo conserva los componentes esenciales y la lógica interna de un sistema Unix real.

1.2.4 Lenguaje(s) de implementación

El núcleo de xv6 está escrito principalmente en ANSI C, con pequeñas porciones en lenguaje ensamblador (por ejemplo, para rutinas de arranque y manejo de interrupciones) (Cox, R. and Kaashoek, F. and Morris, R., 2018, pag. 7),. La versión original de x86 es C (para x86-32); la versión reciente soporta RISC-V también escrita en C

1.2.5 Arquitectura del sistema (monolítica, microkernel, etc.)

Como explica la documentación, xv6 es un kernel monolítico: toda la funcionalidad del sistema corre con privilegios de kernel, sin servidores externos (Cox, R. and Kaashoek, F. and Morris, R., 2018, pag. 25). En Xv6, el kernel sigue el diseño clásico de los sistemas Unix. Funciona como un núcleo único que brinda servicios a múltiples procesos, los cuales se ejecutan en un espacio de usuario independiente. Cada proceso va alternando entre dos modos de ejecución:

- **En el modo usuario:** el proceso ejecuta su propio código y realiza operaciones computacionales básicas.
- **En el modo kernel:** es el sistema operativo quien toma el control para ejecutar las llamadas al sistema solicitadas.

Este cambio ocurre cuando un proceso necesita un servicio del sistema operativo: el hardware cambia automáticamente a un modo de mayor privilegio (modo supervisor)

y transfiere la ejecución al kernel. Una vez que este completa la tarea, devuelve el control al proceso en el espacio de usuario. (Cox, R. and Kaashoek, F. and Morris, R., 2018, pag. 9),

1.2.6 Componentes implementados (procesos, memoria, archivos, etc.)

Xv6 implementa los componentes esenciales de un sistema operativo Unix clásico:

Tabla 1.2: Componentes principales del sistema operativo **XV6**

Componente / Subsistema	**Archivo(s) principal(es)**	**Funcionamiento técnico (idea clave)**
Gestión de procesos y planificador	`proc.c`, `proc.h`	Define la estructura `proc` y el planificador round-robin por CPU; maneja el ciclo de vida de procesos (`fork`, `exec`, `exit`, `wait`).
Gestión de memoria virtual	`vm.c`, `kalloc.c`	Implementa tablas de páginas por proceso (Sv39) y asignador físico mediante `kalloc/kfree`.
Sistema de archivos	`fs.c`, `file.c`, `log.c`	FS tipo Unix con `inode`, directorios, caché de bloques y journaling mediante write-ahead log.
Interfaz de llamadas (`syscalls`)	`syscall.c`, `sysproc.c`, `usys.S`	Tabla de `syscalls` y despacho de traps del modo usuario al kernel (`fork`, `read`, `write`, etc.).
subsistemas de E/S y drivers	`console.c`, `uart.c`, `virtio_disk.c`	Drivers básicos UART y VirtIO integrados con la caché de bloques.
Comunicación entre procesos (IPC)	`pipe.c`, `sys_pipe.c`	Implementa pipes unidireccionales en memoria; permite transmisión de datos entre procesos relacionados mediante `read`/`write` sincronizados.

Gestión de procesos y planificación

La creación de procesos usa `fork()` para copiar el proceso padre y `exec()` para cargar nuevos programas. La planificación emplea un algoritmo **round-robin** básico, donde cada CPU asigna turnos de tiempo fijo a los procesos en una lista cíclica, sin prioridades. Todos los procesos comparten el mismo entorno básico, sin distinción entre usuarios (Cox, R. and Kaashoek, F. and Morris, R., 2018, pag. 82).

Gestión de memoria virtual

Xv6 gestiona la memoria mediante paginación, asignando a cada proceso un espacio de direcciones virtual independiente. Utiliza solo 38 bits de los 64 que soporta RISC-V, permitiendo teóricamente hasta 256 GB de memoria por proceso (Cox, R. and Kaashoek, F. and Morris, R., 2018, pag. 38).

El espacio de direcciones de cada proceso sigue la convención Unix clásica: La distribución de la memoria sigue el diseño clásico de Unix:

- **Segmento de Código**: Ubicado en las direcciones más bajas (cerca de 0), con permisos de solo lectura y ejecución para proteger el código del programa.
- **Datos y Heap**: Situados a continuación del código, con permisos de lectura y escritura. El heap puede expandirse dinámicamente cuando el programa solicita más memoria mediante la llamada `sbrk`.
- **Pila del Usuario**: Ubicada en la parte superior de la memoria, crece hacia abajo y tiene una *página guardia* inferior. Cualquier acceso a esta página causa que el kernel termine el proceso, previniendo desbordamientos.

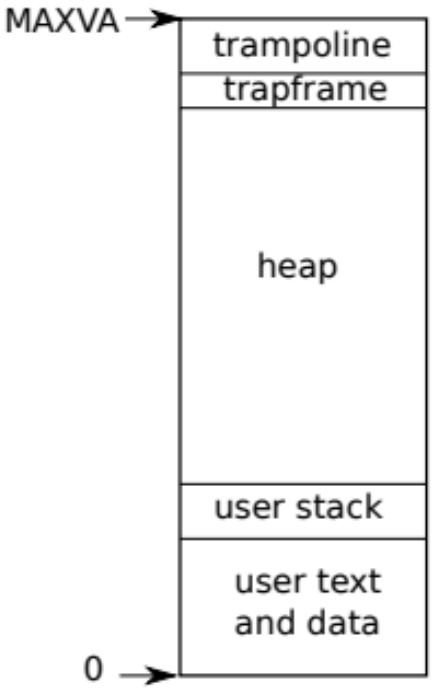


Figura 1.2: Disposicion del espacio de direcciones virtuales de un proceso.

Fuente: Obtenido de (Cox, R. and Kaashoek, F. and Morris, R., 2018) *xv6: a simple, Unix-like teaching operating system.*

Sistema de archivos

Xv6 implementa un sistema de archivos jerárquico que garantiza la integridad de los datos mediante journaling. Su diseño por capas organiza las funciones desde el acceso al disco hasta la gestión de archivos, haciendo que cada componente sea especializado y manejable. (Pekopeko11, 2024).

Organización en Disco El disco se estructura en secciones bien definidas:

- **Bloque 0**: Contiene el código de arranque
- **Bloque 1**: Almacena el "superbloque" con los metadatos del sistema de archivos

- **Bloques siguientes**: Albergan los inodos (que representan archivos)
- **Secciones posteriores**: Incluyen mapas de bits para espacios libres y los bloques de datos reales
- **Zona final**: Reservada para el journal o registro transaccional

Gestión de Archivos y Directorios Los directorios son archivos especiales que asocian nombres con inodos. Las rutas se resuelven secuencialmente (ej: `/usr/bin/ls` busca `usr`, luego `bin`).

Las operaciones básicas (`open`, `read`, `write`, `close`) usan **descriptores de archivo** y una **caché de bloques**. `read` carga datos a memoria, mientras `write` los marca para escritura transaccional. (Cox, R. and Kaashoek, F. and Morris, R., 2018, pag. 85–86).

Interfaz de llamadas (syscalls)

Xv6 implementa llamadas al sistema que siguen el modelo tradicional de Unix, permitiendo a los programas de usuario solicitar servicios del kernel mediante interrupciones de software.(Cox, R. and Kaashoek, F. and Morris, R., 2018, pag. 8–9).

El mecanismo funciona de la siguiente manera:

1. Un proceso en modo usuario invoca una syscall, lo que hace que el hardware genere una interrupción.
2. Esta interrupción transfiere el control al kernel, cambiando el modo de ejecución de la CPU.

3. El kernel identifica la syscall solicitada a través de un número único y ejecuta la función correspondiente.

4. Una vez completada, el kernel devuelve el control al proceso de usuario con los resultados necesarios.

E/S y drivers

Los dispositivos en xv6 se integran en el sistema bajo el principio de que "todo es un archivo". Esto se logra mediante **archivos de dispositivo**, creados con la llamada al sistema `mknod`.

- `mknod(path, major, minor)`: Crea un archivo especial que representa un dispositivo. Los parámetros `major` (mayor) y `minor` (menor) son números que identifican de forma única al controlador (*driver*) del kernel y al dispositivo específico.

Controladores de Dispositivos (Drivers) El kernel incluye controladores mínimos pero funcionales para hardware esencial. Algunos ejemplos son:

- `uart.c` para el puerto serie.
- `virtio_disk.c` para el acceso al disco.
- `timer.c` para el temporizador del sistema.

Comunicación entre procesos (IPC)

Xv6 implementa un mecanismo básico de comunicación entre procesos (IPC) mediante **pipes**. Las pipes son canales unidireccionales en memoria que permiten a dos procesos

relacionados intercambiar datos de forma sincronizada.(Cox, R. and Kaashoek, F. and Morris, R., 2018, pag. 73–74),

1.2.7 Herramientas utilizadas (compiladores, emuladores, etc.)

Para la construcción, compilación y ejecución de *xv6*, se emplean las siguientes herramientas principales:

- **GCC (GNU C Compiler):** Compila el kernel y los programas de usuario escritos en C, generando binarios ELF compatibles con las arquitecturas `x86-32` o `RISC-V`.
- **QEMU:** Emulador de hardware ampliamente utilizado para ejecutar *xv6* en entornos virtuales. Permite probar el sistema operativo sin necesidad de hardware físico, simulando arquitecturas `x86` o `RISC-V`.
- **Cross-compiler:** En sistemas como macOS o Windows, se requiere un compilador cruzado para generar binarios ELF destinados a la arquitectura objetivo de *xv6*.

(MIT PDOS, 2024),

1.2.8 Nivel de complejidad y accesibisslidad para estudiantes

Xv6 fue diseñado específicamente para ser un sistema operativo ligero y educativo. Su código completo consta de aproximadamente 10,000 líneas (equivalente a unas 99 páginas impresas) Su simplicidad relativa (ausencia de capas complejas como módulos dinámicos) lo hace muy accesible para estudiantes de OS. Aun así, cubre los conceptos

esenciales de concurrencia, paginación, sistemas de archivos y llamadas al sistema típicos de un Unix real.(Cox, R. and Kaashoek, F. and Morris, R., 2018, pag. 10–11).

1.3 Theseus

1.3.1 Nombre del proyecto o sistema operativo

El nombre del sistema operativo es "Theseus"; según (Theseus OS Project, 2023), fue nombrado así en honor a la paradoja del barco de Teseo, que plantea la cuestión de, si a un barco le renuevas todas sus piezas, ¿Ese barco renovado sigue siendo el mismo barco? Esta idea de renovación de cada componente es característica de este sistema operativo.

1.3.2 Enlace al repositorio y/o documentación oficial

- **Enlace al repositorio oficial:** https://github.com/theseus-os/Theseus
- **Enlace al libro oficial:** https://www.theseus-os.com/Theseus/book/index.html

1.3.3 Objetivo del proyecto

Según (Boos et al., 2020), este sistema operativo es en esencia experimental, aunque puede usarse como objeto de estudio debido a su innovadora arquitectura. También tiene objetivos técnicos, como reestructurar la modularidad, reducir el "state spill" y la recuperación ante fallos (fault recovery).

1.3.4 Lenguaje(s) de implementación

Según (Boos et al., 2020) y confirmando con el repositorio oficial (Boos, 2024), el sistema operativo Theseus fue casi completamente desarrollado en Rust, aunque también

se ha usado C para una futura implementación de la librería "libc" dirigida a Theseus (tlibc), ya que por el momento se tomó prestado de la biblioteca de REDOX (relibc).

1.3.5 Arquitectura del sistema (monolítica, microkernel, etc.)

Theseus, como sistema operativo no tiene una arquitectura clásica como monolítica o microkernel, ni mucho menos multikernel; sino que basa su estructura en **cells** o células por su traducción en español, las cuales haciendo honor a su definición biológica, son módulos pequeños e independientes que sirven como bloque de construcción para el sistema operativo. Adicionalmente, theseus tiene un componente mínimo el cual es llamado "nano-core", que se encarga de iniciar el sistema, establecer memoria mínima y cargar las demás células; pero a diferencia de un kernel propiamente dicho, este no es un jefe permanente, sino que cada cell es independiente (Boos et al., 2020). A continuación una tabla comparativa entre un kernel clásico y el nano-core de theseus:

Tabla 1.3: Comparación entre un kernel tradicional y el nanocore de Theseus

Aspecto	**Kernel tradicional**	**Nanocore (Theseus)**
Definición	Es el núcleo del sistema operativo. Centraliza la gestión de componentes y recursos.	Unidad mínima del núcleo que gestiona sólo lo esencial para iniciar y mantener cells.
Estructura	Monolítica o microkernel: el kernel controla los servicios del sistema.	No existe un "núcleo único": el sistema está compuesto por módulos cooperativos.
Acoplamiento	Alto. Los módulos dependen del kernel y de otros subsistemas.	Bajo. Cada módulo es reemplazable y se comunica mediante interfaces explícitas.
Objetivo	Proveer servicios centrales de manera estable.	Eliminar el *state spill* (derrame de estado), permite reemplazar componentes en ejecución.

Fuente: Adaptado en base a (Boos et al., 2020), *Theseus: An Experiment in OS Structure for State Management.*

Para entenderlo mejor, se especifica que se impone una arquitectura/organización plana para estas células, todo se corre en un solo SAS (single address space) y un solo nivel de privilegio (no se diferencia entre modo usuario y modo kernel) (Boos & Zhong, 2017). Además, se incluye una figura comparativa entre arquitecturas clásicas y la de theseus basada en células:

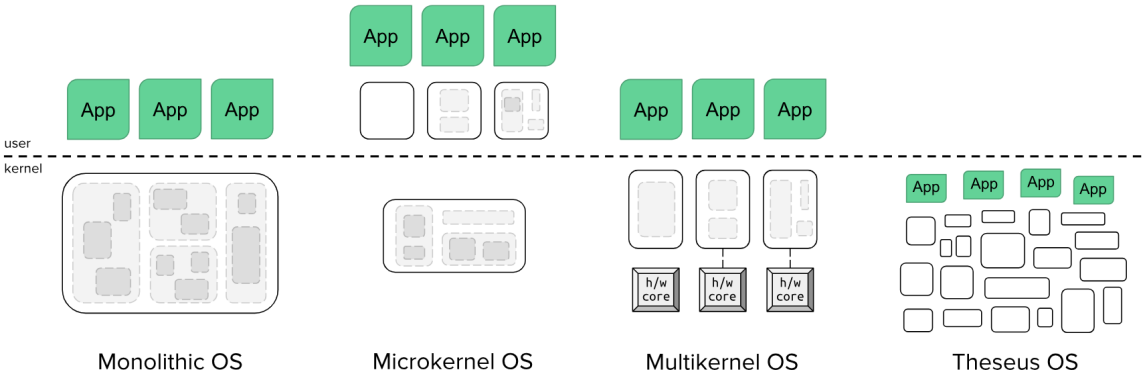


Figura 1.3: Comparativa de la arquitectura de Theseus frente a arquitecturas tradicionales.

Fuente: Obtenido de (Theseus OS Project, 2023) *The Theseus OS Book*.

1.3.6 Componentes implementados (procesos, memoria, archivos, etc.)

En theseus, cada cell encapsula una parte del sistema, encapsula cada componente, como gestión de memoria, planificación/Scheduling, E/S o comunicación. Este trabajo con cells permite aislar y detectar fallos, para poder reemplazar componentes en ejecución y evitar el *state spill* entre módulos (Boos et al., 2020). A continuación y a modo de introducción, se presenta una tabla con los principales componentes y subsistemas de Theseus OS, los nombres de sus archivos en el repositorio oficial y la idea clave detrás de sus funcionamiento.

Tabla 1.4: Funcionamiento técnico (idea clave) de los principales subsistemas de Theseus OS

Componente / Subsistema	**Nombre en Theseus (archivo/crate)**	**Funcionamiento técnico (idea clave)**
Gestión de memoria	`memory` crate	Mapea páginas virtuales a marcos físicos garantizando exclusividad.
Carga dinámica de módulos (módulos / células)	`mod_mgmt` crate	Carga en ejecución módulos ("cells") y gestiona sus dependencias.
Scheduling / multitarea	`scheduler` / `task` crates	Ejecuta tareas en un único espacio de direcciones compartido.
Manejo de archivos / sistema de archivos	`fs` crate	Gestiona operaciones de archivos, bajo la modularidad de Theseus.
Subsistemas de E/S y drivers	Ej. `e1000`, `ata_pio`, `keyboard`	Controladores modulares escritos en Rust, diseñados para reemplazo.
Recuperación ante fallos / actualización en caliente	`loader`, `spawn` crates	Reemplaza módulos activos sin reiniciar el sistema completo.
Comunicación entre módulos	`event_types`, `device_manager` crates	Mensajería sin estado entre módulos para evitar dependencias implícitas.

Fuente: Elaboración propia con base en (Boos et al., 2020; Theseus OS Project, 2023; Boos & Zhong, 2017).

Gestión de memoria

El componente `memory` de Theseus implementa una idea `MappedPages`, que representa una región de páginas virtuales adyacentes, mapeadas a marcos físicos. El mapeo se realiza con la función `Mapper::map(pages, frames, flags, pg_tbl)`, y se aprovechan las utilidades de Rust para asegurar que cada página tenga un único marco asociado (Boos et al., 2020; Theseus OS Project, 2023).

Carga dinámica de módulos (módulos / células)

El subsistema `mod_mgmt` permite que los módulos (cells) se carguen, vinculen y descarguen en tiempo de ejecución. Para ello, se construyen instancias de `CrateNamespace`. Este subsistema nos permite sustituir un módulo (cell) sin reiniciar por completo el sistema (Theseus OS Project, 2023).

Scheduling / multitarea

Los crates `scheduler` y `task` gestionan las tareas en un único entorno de espacio de direcciones (SAS) y un solo nivel de privilegio (SPL). Cada tarea se encapsula mediante una estructura `Task` que contiene el contexto del CPU, el puntero de pila y el estado de ejecución. Este enfoque de unicidad evita el fenómeno del "state spill" entre módulos (Boos, 2024).

Manejo de archivos / sistema de archivos

El crate `fs` implementa las funciones básicas del sistema de archivos, como lectura, escritura y gestión de directorios. Siguiendo la filosofía de Theseus, este módulo

se carga como una cell independiente, permitiendo su reemplazo o actualización sin necesidad de reiniciar el sistema (Boos, 2024).

Subsistemas de E/S y drivers

Los drivers como `e1000`, `ata_pio` o `keyboard` escritos en Rust; gestionan hardware de red, discos PATA/IDE y teclados PS/2 respectivamente. Fiel al enfoque de Theseus, son independientes y reemplazables en tiempo de ejecución; para facilitar el "fault recovery" (Boos, 2024).

Recuperación ante fallos / actualización en caliente

Los crates `loader` y `spawn` son las cells que permiten el tan aclamado "fault recovery" y "live update"; o sea recuperación ante fallos y actualización, de cells, sin reiniciar todo el sistema. El funcionamiento contiene detención de tareas, liberación de recursos y recarga de cells (Theseus OS Project, 2023).

Comunicación entre módulos

Los subsistemas `event_types` y `device_manager` se encargan de la correcta mensajería entre cells mediante canales tipados y sin depender de estado compartido. Las dependencias se definen con metadatos para eliminar el "state spill" en la comunicación entre componentes (Boos & Zhong, 2017).

1.3.7 Herramientas utilizadas (compiladores, emuladores, etc.)

Theseus emplea y modifica herramientas de y para Rust; también utiliza máquinas virtuales. Las principales herramientas son las siguientes:

- **Rust toolchain:** Uso de `cargo`, `rustc` y crates estándar del ecosistema Rust para la compilación, gestión de dependencias y construcción modular de Theseus.

- **Sistema de carga dinámica y enlazado en tiempo de ejecución:** Theseus modifica el compilador (rustc) y el "linker" para permitir la carga y sustitución de (cells) en ejecución (Boos & Zhong, 2017).

- **Emuladores y máquinas virtuales:** Se utilizan entornos como "QEMU" para probar, depurar y validar el funcionamiento del sistema (Boos, 2024).

- **Librería estandar de C:** Actualmente, Theseus utiliza gran parte de **relibc** de Redox OS como su biblioteca estándar de C, pero planea desarrollar su propia versión llamada **tlibc** (Theseus libc) en el futuro (Boos, 2024).

1.3.8 Nivel de complejidad y accesibilidad para estudiantes

Considerando un entorno académico, que es justamente donde rust brilla, es un proyecto digno de estudio, con un altísimo nivel de complejidad debido también y en mayor medida, a su arquitectura basada en cells; también cabe mencionar que la dificultad sube debido al uso de la versión de la librería estandar de C (relibc) y el uso de QEMU, el cual es un emulador no tan intuitivo y del cual no hay tanta documentación en español.

Sin embargo, se ve compensado debido a su alta accesibilidad, la cual es posible gracias a su extensa documentación y estudios en torno a este proyecto, propiciados por el mismo creador de Theseus.

1.4 RedoxOS

1.4.1 Nombre del proyecto o sistema operativo

El sistema operativo examinado es "Redox OS". De acuerdo con (Redox OS Project, 2025c), su denominación deriva de la palabra "Redox", que es la abreviatura de "*reduction–oxidation*", un término de química que representa el intercambio equilibrado de electrones; esto refleja la meta del proyecto: alcanzar un sistema equilibrado en seguridad, rendimiento y simplicidad, a través de un diseño contemporáneo y fiable, redactado totalmente en Rust.

1.4.2 Enlace al repositorio y/o documentación oficial

- **Enlace al repositorio oficial:** https://github.com/redox-os
- **Enlace al libro oficial:** https://doc.redox-os.org/book/

1.4.3 Objetivo del proyecto

Según (Redox OS Project, 2025c,g), Redox OS tiene un enfoque fundamentalmente educativo y experimental, pero también pretende ofrecer una opción práctica y segura en comparación con sistemas Unix tradicionales. Su objetivo es probar que un sistema operativo que esté totalmente desarrollado en Rust puede lograr la misma estabilidad y eficiencia que los sistemas comerciales, mientras elimina categorías completas de errores relacionados con la memoria y la concurrencia.

1.4.4 Lenguaje(s) de implementación

Redox OS está predominantemente desarrollado en Rust, abarcando su kernel, controladores, gestor de memoria, sistema de archivos y gran parte de las herramientas de usuario.

Ciertos elementos específicos, como el cargador de arranque y aspectos de la compatibilidad con C, emplean "ensamblador y C" de manera restringida.

1.4.5 Arquitectura del sistema

Redox OS adopta una arquitectura híbrida de microkernel, inspirada en el diseño de Minix 3 y el modelo de Unix, pero con un enfoque mucho más fuerte en la seguridad y la modularidad. El kernel de Redox es extremadamente pequeño: se encarga solo de la planificación, la gestión básica de memoria y la comunicación entre procesos. Todos los servicios del sistema, como el manejo de archivos, controladores, red y entorno gráfico, se ejecutan en el espacio de usuario como procesos independientes llamados schemes.

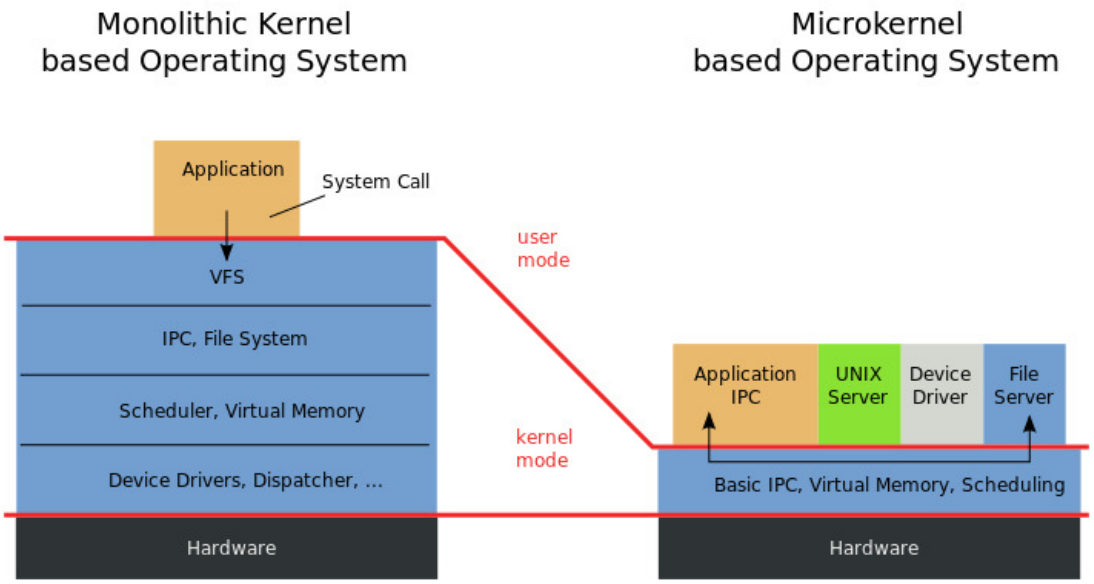


Figura 1.4: Comparativa de la arquitectura de Redox frente a arquitecturas tradicionales.

Fuente: Obtenido de (Redox OS Project, 2025c).

1.4.6 Componentes implementados

En Redox OS, los componentes principales del sistema se implementan como crates escritos en `Rust`, siguiendo una arquitectura microkernel. Cada crate encapsula un servicio o subsistema (procesos, memoria, sistema de archivos, E/S, etc.) con una estricta separación de privilegios. El microkernel se encarga de la comunicación por mensajes entre estos servicios, garantizando seguridad, aislamiento y estabilidad.

A continuación, se presenta una tabla con los principales componentes y subsistemas de Redox OS, los nombres de sus crates en el repositorio oficial y la idea técnica detrás de su funcionamiento.

Tabla 1.5: Funcionamiento técnico de los principales subsistemas de Redox OS

Componente / Subsistema	**Nombre en Redox (archivo/crate)**	**Funcionamiento técnico (idea clave)**
Gestión de memoria	`memory`, `paging`, `rmm`	Maneja paginación virtual y asignación segura de marcos físicos.
Gestión de procesos y multitarea	`kernel`, `scheme::proc`	Ejecuta procesos aislados con planificación cooperativa.
Sistema de archivos	`redoxfs`	Sistema nativo en Rust con soporte concurrente.
Subsistemas de E/S y drivers	`drivers`, `schemes`	Drivers en espacio de usuario con namespaces restringidos.
Comunicación entre procesos (IPC)	`syscall`, `scheme`, `libredox`	IPC basado en mensajes mediante estructuras SQE/CQE.
Planificación y temporización	`scheduler`, `timer`	Planificador por prioridad con temporizador interno.
Gestión de usuarios y permisos	`user`, `authd`	Controla autenticación y separación de privilegios.
Interfaz gráfica (GUI)	`orbital`, `orbtk`	Servidor de ventanas y toolkit gráfico modular.
Red y conectividad	`netstack`, `ethernetd`, `ipd`	Pila de red completa en espacio de usuario.
Gestión del sistema y servicios	`init`, `logd`, `backgroundd`	Controla inicio, logs y servicios del sistema.
Bibliotecas del sistema y compatibilidad	`relibc`, `redox_rt`	Implementan llamadas POSIX sobre los *schemes*.

Fuente: Elaboración propia con base en (Redox OS Project, 2025g,c).

Gestión de memoria

La gestión de memoria en Redox OS utiliza paginación virtual y asignación segura de marcos físicos. El crate `rmm` implementa el gestor de memoria del kernel, garantizando aislamiento entre procesos y eficiencia en la asignación dinámica (OSDev Wiki, 2025a).

Gestión de procesos y multitarea

El kernel de Redox implementa multitarea cooperativa, donde cada proceso se ejecuta en espacio de usuario y el microkernel gestiona la planificación y sincronización básica entre tareas (OSDev Wiki, 2025b).

Sistema de archivos

`RedoxFS` es el sistema de archivos nativo, completamente escrito en Rust. Ofrece acceso concurrente, soporte para permisos y comunicación mediante el modelo de schemes.

Subsistemas de E/S y drivers

En Redox OS, los controladores de dispositivos se ejecutan como demonios en espacio de usuario. Cada driver tiene su propio namespace restringido, evitando daños al sistema principal. Esta separación aumenta la seguridad y la estabilidad del sistema (Redox OS Project, 2025f).

Comunicación entre módulos

La comunicación se realiza mediante el sistema de schemes. Las llamadas del usuario se traducen en mensajes SQE/CQE, enviados entre el kernel y los servicios de usuario. Esto asegura modularidad y aislamiento total entre componentes (Redox OS Project, 2025a).

Planificación y temporización

El planificador emplea colas de prioridad para asignar tiempo de CPU. Busca mantener equidad, prioridad y escalabilidad con una complejidad cercana a $O(1)$ (OSDev Wiki, 2025b).

Gestión de usuarios y permisos

Los servicios `authd` y `user` manejan autenticación, control de accesos y separación de privilegios, previniendo escaladas no autorizadas (Redox OS Project, 2025h).

Interfaz gráfica (GUI)

`Orbital` y `OrbTK` implementan el entorno gráfico de Redox. Incluyen un servidor de ventanas, gestión de eventos y un toolkit modular, todo en Rust (Redox OS Project, 2025d).

Red y conectividad

El sistema de red, compuesto por `netstack`, `ethernetd` e `ipd`, implementa una pila TCP/IP segura en espacio de usuario, sin requerir acceso directo al kernel (Redox OS Project, 2025b).

Gestión del sistema y servicios

Servicios como `init`, `logd` y `backgroundd` administran la inicialización, el registro de eventos y procesos en segundo plano, similares al sistema init de Unix (Redox OS Project, 2025j).

Bibliotecas del sistema y compatibilidad

`relibc` y `redox_rt` ofrecen compatibilidad POSIX. Estas bibliotecas traducen funciones estándar (como `open`, `read`, `write`) a operaciones sobre schemes del sistema (Redox OS Project, 2025e).

1.4.7 Herramientas utilizadas

El ecosistema de desarrollo de Redox OS está construido íntegramente sobre las herramientas del lenguaje `Rust` y un conjunto de utilidades de compilación y virtualización que garantizan portabilidad y reproducibilidad en distintos entornos.

Entre las principales herramientas empleadas se destacan:

- **Rust y Cargo:** Redox OS está escrito completamente en Rust, utilizando `cargo` como sistema de compilación y gestión de dependencias. Cada componente del sistema (kernel, drivers, librerías y servicios) se organiza como un *crate*.
- **Make y scripts de automatización:** El proceso de construcción global se gestiona mediante archivos `Makefile` y scripts en Bash, que coordinan la compilación cruzada de todos los crates y bibliotecas.

- **Podman y Docker:** Se utilizan para entornos de compilación reproducibles. Estas herramientas permiten construir el sistema dentro de contenedores sin necesidad de alterar la configuración del host.

- **QEMU:** Es el emulador principal para ejecutar y depurar Redox OS sin requerir hardware físico. Permite probar el kernel, los controladores y las aplicaciones en un entorno controlado.

- **Git y GitLab:** Todo el código fuente está alojado en GitLab (anteriormente en GitHub). Git se usa para control de versiones, colaboración y revisión de código.

- **Herramientas de compilación cruzada (cross):** Permiten compilar Redox para distintas arquitecturas como `x86_64`, `i686` o `ARM64`, manteniendo el mismo entorno de desarrollo.

Estas herramientas no solo simplifican el desarrollo, sino que también facilitan la experimentación y el aprendizaje, ya que permiten reconstruir completamente el sistema operativo desde el código fuente, probarlo en un entorno virtual y modificar cualquier parte del kernel o de los servicios del usuario.

Corresponde a la sección 6.1–6.8 del manual oficial de Redox OS sobre el proceso de construcción y herramientas de compilación (Redox OS Project, 2025i).

1.4.8 Nivel de complejidad y accesibilidad para estudiantes

Redox OS presenta un equilibrio notable entre complejidad técnica y accesibilidad educativa. Aunque es un sistema operativo funcional y moderno, su diseño modular

en torno a un microkernel y su implementación en `Rust` facilitan el estudio y la modificación de sus componentes por parte de estudiantes o investigadores.

- **Código fuente legible y seguro:** Rust impone reglas de seguridad de memoria y control de concurrencia, reduciendo errores comunes de punteros y sincronización. Esto hace que el código de Redox sea más fácil de entender para quienes se inician en el desarrollo de sistemas.

- **Arquitectura microkernel:** Su estructura simplificada y modular permite analizar cada subsistema de manera aislada (memoria, procesos, archivos, drivers, etc.), lo que favorece la comprensión progresiva del sistema operativo.

- **Facilidad de compilación y pruebas:** Gracias a `QEMU` y a los contenedores de `Podman`, los estudiantes pueden compilar y ejecutar Redox en sus propias máquinas sin riesgo de dañar su sistema anfitrión.

- **Documentación educativa:** El Redox OS Book ofrece guías detalladas para compilar, depurar y extender el sistema, convirtiéndolo en una herramienta pedagógica ideal para cursos de sistemas operativos o arquitectura de software.

- **Comunidad activa:** Existe una comunidad abierta de desarrolladores y estudiantes en GitLab y Matrix que brindan soporte, facilitan la colaboración y fomentan el aprendizaje conjunto.

Corresponde a las secciones 1.1–1.7 y 6.1–6.8 del manual oficial de Redox OS (Redox OS Project, 2025c,i).

1.5 FlexOS

1.5.1 Nombre del proyecto o sistema operativo

El sistema operativo analizado es "FlexOS". Según (Kuenzer, Simon and Lefeuvre, Hugo and Huici, Felipe and others, 2023), su nombre proviene del término "Flexible Operating System", reflejando su principal objetivo: ofrecer una plataforma operativa flexible y configurable que permita ajustar dinámicamente el nivel de aislamiento entre componentes. A diferencia de los sistemas tradicionales con arquitecturas rígidas (monolíticas o microkernel), FlexOS propone una arquitectura adaptable donde el desarrollador puede equilibrar seguridad, aislamiento y rendimiento según las necesidades del sistema.

1.5.2 Enlace al repositorio y/o documentación oficial

- **Repositorio oficial:** https://github.com/project-flexos
- **Sitio y documentación técnica:** https://project-flexos.github.io/
- **Publicación principal:** *"FlexOS: Making OS Isolation Flexible"*, presentada en EuroSys 2023 (Kuenzer, Simon and Lefeuvre, Hugo and Huici, Felipe and others, 2023).

1.5.3 Objetivo del proyecto

El objetivo central de FlexOS (educativo, experimental) es redefinir el equilibrio entre rendimiento y aislamiento en los sistemas operativos. Mientras los sistemas tradicionales deben optar entre un kernel monolítico rápido o un microkernel seguro pero

más costoso en rendimiento, FlexOS permite ajustar el nivel de aislamiento de manera configurable. Esto se logra mediante su diseño modular, que permite que cada componente (como controladores, memoria o IPC) se ejecute con distintos niveles de separación temporal y espacial. De esta forma, se busca proporcionar un entorno operativo capaz de adaptarse a distintos contextos: desde sistemas embebidos con recursos limitados, hasta entornos de alta seguridad o investigación académica (Kuenzer, Simon and Lefeuvre, Hugo and Huici, Felipe and others, 2023).

1.5.4 Lenguaje(s) de implementación

FlexOS está implementado principalmente en C y C++, con soporte parcial en Rust para módulos experimentales que requieren mayor seguridad en memoria. Su infraestructura está construida sobre el microkernel Unikraft, al que extiende con nuevas bibliotecas y mecanismos de aislamiento. La elección de C y C++ permite mantener compatibilidad con Unikraft y con componentes de sistemas POSIX existentes, mientras que Rust se emplea en zonas críticas de seguridad o donde la verificación de memoria resulta esencial (Kuenzer, Simon and Lefeuvre, Hugo and Huici, Felipe and others, 2023).

1.5.5 Arquitectura del sistema

FlexOS presenta una arquitectura modular y configurable, inspirada en los principios de los microkernels, pero con un grado de flexibilidad superior. Cada componente puede ejecutarse en un dominio de aislamiento, que define si comparte espacio de direcciones, hilos de ejecución o recursos temporales con otros módulos (Kuenzer,

Simon and Lefeuvre, Hugo and Huici, Felipe and others, 2023). Estos dominios pueden configurarse de tres maneras principales:

El núcleo de FlexOS gestiona la coordinación de estos dominios, la planificación del tiempo de CPU, la comunicación entre módulos (IPC) y el control de errores.

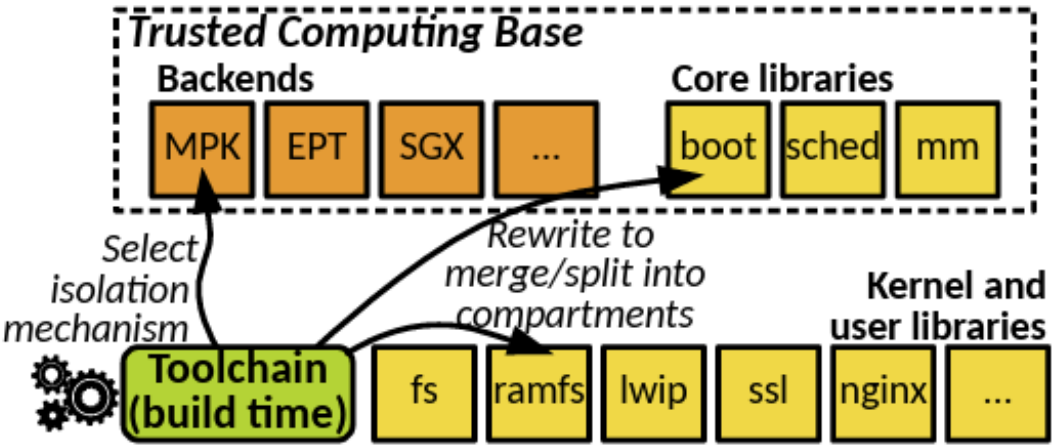


Figura 1.5: Arquitectura modular e híbrida de FlexOS.

Fuente: Obtenido de (Kuenzer, Simon and Lefeuvre, Hugo and Huici, Felipe and others, 2023).

1.5.6 Componentes implementados

En FlexOS, los componentes del sistema se diseñan para maximizar el aislamiento y la modularidad, aplicando principios de sistemas operativos seguros y configurables. Su estructura permite ajustar el nivel de aislamiento (temporal y espacial) entre componentes según los requisitos de rendimiento o seguridad, convirtiéndolo en un sistema operativo híbrido altamente flexible.

Los componentes se implementan principalmente en C, C++ y Rust, utilizando microbibliotecas y servicios del kernel configurables. A continuación, se presenta una tabla con los principales subsistemas y módulos de FlexOS, junto con su función técnica principal (Kuenzer, Simon and Lefeuvre, Hugo and Huici, Felipe and others, 2023).

Tabla 1.6: Funcionamiento técnico de los principales subsistemas de FlexOS

Componente / Subsistema	**Nombre o módulo en FlexOS**	**Funcionamiento técnico (idea clave)**
Núcleo y planificación	`flex_kernel`	Controla la ejecución y distribución del tiempo de CPU entre tareas.
Gestión de memoria	`mem_mgr`, `isolation_lib`	Maneja memoria virtual y niveles configurables de aislamiento.
Procesos y tareas	`task_mgr`	Crea y coordina tareas con comunicación segura.
Sistema de IPC	`ipc_core`, `shared_mem`	Permite intercambio de datos mediante colas y memoria compartida.
Seguridad y aislamiento	`sandbox`, `trusted_domain`	Define dominios seguros y políticas de acceso.
Gestión de E/S y drivers	`io_layer`, `device_mgr`	Ejecuta controladores aislados para proteger el kernel.
Monitoreo y fallos	`fault_handler`, `recovery_svc`	Detecta errores y reinicia módulos sin afectar el sistema.
Configuración del sistema	`flex_config`	Ajusta el nivel de aislamiento y rendimiento.
Bibliotecas de usuario	`libflex`, `api_runtime`	Facilita la comunicación segura con el kernel.

Fuente: Elaboración propia con base en (Kuenzer, Simon and Lefeuvre, Hugo and Huici, Felipe and others, 2023).

Gestión de memoria

FlexOS implementa un modelo de gestión de memoria basado en *espacial isolation*, donde cada dominio o componente puede tener su propio espacio de direcciones. El sistema utiliza tanto protección por hardware (MMU) como políticas de software para garantizar independencia entre procesos y minimizar los efectos de fallos de memoria.

Procesos y multitarea

El administrador de tareas controla el ciclo de vida de los procesos mediante colas de ejecución. Los procesos pueden compartir recursos limitadamente o ejecutarse en dominios completamente aislados, dependiendo de la configuración elegida.

Comunicación entre módulos

FlexOS utiliza un sistema híbrido de comunicación: colas seguras, llamadas IPC síncronas y memoria compartida protegida. Esto permite un equilibrio entre rendimiento y seguridad, ajustable según la política de aislamiento establecida.

Gestión de E/S y drivers

Los controladores en FlexOS se ejecutan como módulos independientes o en dominios restringidos, minimizando el riesgo de fallos. Cada driver puede ser reiniciado o reemplazado sin afectar el núcleo principal.

Seguridad y aislamiento

Una de las características clave del sistema es su capacidad para reconfigurar el aislamiento entre componentes, ofreciendo niveles ajustables de separación espacial y

temporal. Esto permite ejecutar aplicaciones críticas en entornos seguros sin comprometer el rendimiento global del sistema.

1.5.7 Herramientas utilizadas (compiladores, emuladores, etc.)

FlexOS se construye sobre la infraestructura de **Unikraft**, por lo que emplea un conjunto de herramientas especializadas para la compilación, configuración y ejecución de sus módulos. El proceso de construcción del sistema operativo se basa en la cadena de herramientas de `GNU Make` y `Kconfig`, lo que permite personalizar la inclusión o exclusión de componentes del kernel según el nivel de aislamiento deseado.

Entre las herramientas más relevantes utilizadas en FlexOS se encuentran:

- **Compilador principal:** `clang/LLVM`, utilizado para la instrumentación del código, generación de binarios seguros y soporte para aislamiento entre dominios.
- **Infraestructura de construcción:** `Unikraft Build System`, que proporciona un entorno modular basado en `KBuild` y `Makefiles` para definir configuraciones del kernel y librerías.
- **Simuladores y entornos de prueba:** ejecución de instancias de FlexOS sobre `QEMU` y máquinas virtuales `KVM`, permitiendo validar configuraciones con diferentes niveles de aislamiento y medir overheads de seguridad.
- **Instrumentación y análisis:** uso de `AddressSanitizer (ASan)`, `Clang Control Flow Integrity (CFI)` y trazadores de rendimiento para detectar vulnerabilidades y medir costos de comunicación entre dominios.

- **Herramientas de fuzzing y validación:** integración con `ConfFuzz`, el framework de fuzzing desarrollado por los mismos autores, usado para explorar automáticamente configuraciones de aislamiento y detectar fallos en el espacio de diseño.

1.5.8 Nivel de complejidad y accesibilidad para estudiantes

FlexOS presenta un nivel de complejidad intermedio a avanzado, dado su enfoque experimental y su dependencia de la infraestructura de Unikraft y del compilador LLVM. Aunque está diseñado como un sistema de investigación y enseñanza sobre *microkernels híbridos y aislamiento flexible*, su curva de aprendizaje es considerablemente mayor que la de sistemas educativos tradicionales como XV6 o MINIX.

Su accesibilidad depende del objetivo académico:

- Para estudiantes de pregrado, el entorno de FlexOS puede resultar complejo por su uso intensivo de compilación cruzada, configuración Kconfig y dependencias de LLVM, pero es útil para entender conceptos de modularidad y aislamiento.
- Para estudiantes de posgrado o investigadores, ofrece un entorno ideal para estudiar la relación entre *rendimiento y seguridad*, mediante experimentación con distintos niveles de compartmentalización, IPC y aislamiento de memoria.

1.6 LibrettOS

1.6.1 Nombre del proyecto o sistema operativo

Seguna la sitio oficial (Systems Software Research Group, 2021), El nombre de este proyecto es LibrettOS, un sistema operativo de código abierto desarrollado por el Grupo de Investigación de Software de Sistemas de Virginia Tech. Se basa en la combinación de dos paradigmas un microkernel y el modelo de biblioteca.

1.6.2 Enlace al repositorio y/o documentación oficial

- **Pagina oficial:** https://librettos.org/
- **Enlace al repositorio oficial:** https://github.com/ssrg-vt/librettos-src
- **Articulo:** https://ssrg.ece.vt.edu/papers/vee20-librettos.pdf

1.6.3 Objetivo del proyecto

Según (Ruslan Nikolaev and Mincheol Sung and Binoy Ravindran, 2021), LibrettOS es un sistema operativo experimental de investigaciones. Su objetivo es explorar el paradigma del núcleo híbrido que combine con el microkernel multiserver, para mantener mayor seguridad y tolerancia a errores a fallos y permitir a las aplicaciones el acceso a directamente al hardware para un alto rendimiento. Especificamente no fue creado para fines educativo ni para uso constante, sino como un prototipo academico para diseñar nuevas ideas acerca de sistemas operativos.

1.6.4 Lenguaje(s) de implementación

Según el repositorio (The rumpkernel project, 2025), LibrettOS fue implementado principalmente en el lenguaje de programación C (55.8%), C++(0.5%), Assembly(9.5%), Makefile(4.2%) y objetive-C(0.3%), lo cual es clásico para sistemas operativos debido a la conexión cercana con el hardware y la facilidad para escribir código. Durante años, algunas partes del código también han sido implementadas en ensamblador para aprovechar características específicas del hardware.

1.6.5 Arquitectura del sistema (monolítica, microkernel, etc.)

Según (Ruslan Nikolaev and Mincheol Sung and Binoy Ravindran, 2021, pag. 5–6), la arquitectura de LibrettOS combina modos de ejecucion para optimizar el rendimiento y el aislamiento. La aplicación 1 se ejecuta en el modo de sistema operativo de biblioteca, accediendo a todos los dispositivos de hardware. La aplicación 2 se comunica a través de de servidores de red. LibrettOS se ejecuta sobre Xen, un hypervisor de la arquitectura microkernel, el sistema permite varios accesos directos como indirectos al hardware. Para el almacenamiento, permite compartir archivos mediante un servidor NFS, actualmente LibrettOS es compatible con API POSIX/BSD. Para entender mejor la arquitectura, se muestra en la figura 1.6.

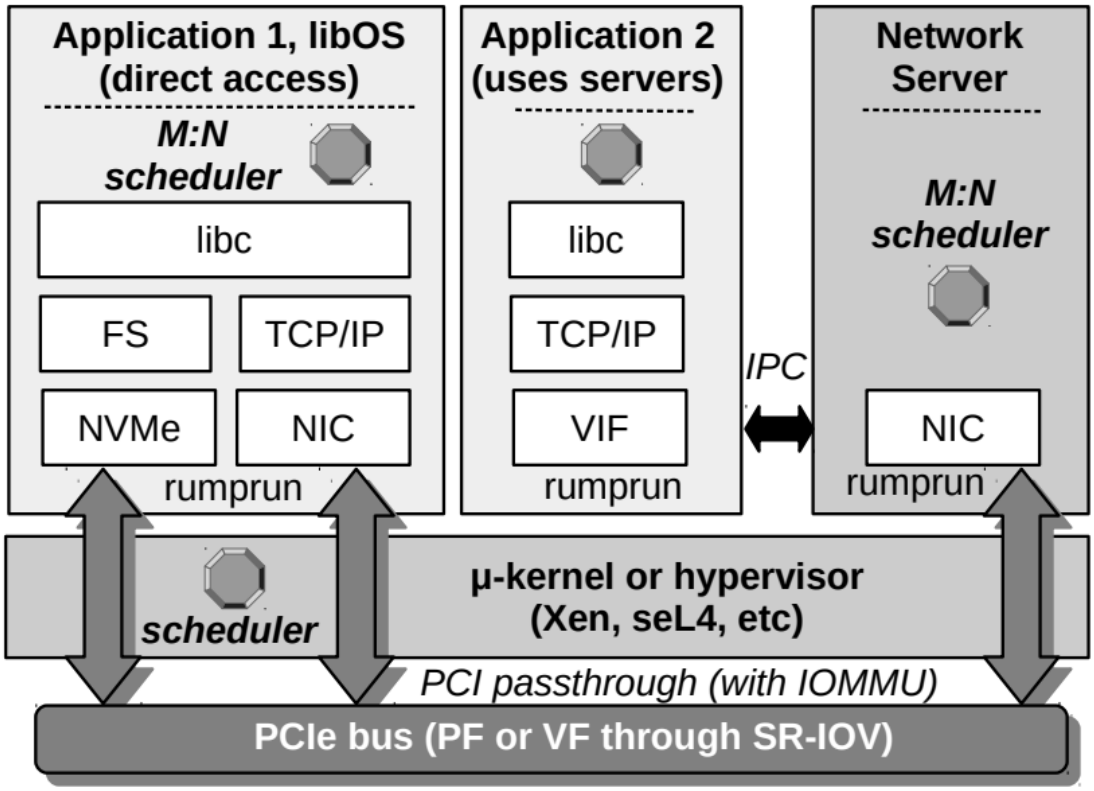


Figura 1.6: Arquitectura de LibrettOS.

Fuente: Obtenido de (Ruslan Nikolaev and Mincheol Sung and Binoy Ravindran, 2021, pag. 5) *LibrettOS: A Dynamically Adaptable Multiserver-Library OS*.

1.6.6 Componentes implementados (procesos, memoria, archivos, etc.)

LibrettOS es un sistema modular que distribuye los servicios del sistema operativo (procesos, memoria, archivos, red y controladores) entre servidores de usuario y librerías en espacio de aplicación. La tabla 1.7 resume sus principales subsistemas y su funcionamiento técnico.

Tabla 1.7: Funcionamiento técnico de los principales subsistemas de LibrettOS

Componente / Sub-sistema	**Implementación en LibrettOS**	**Funcionamiento técnico (idea clave)**
Gestión de procesos y planificación	`rumpkernel / rumprun-smp`	Cada dominio ejecuta un *rumpkernel* multi-hilo con planificación M:N. Xen gestiona los vCPUs y Rumprun programa los hilos POSIX.
Gestión de memoria	`NetBSD VM / Xen / IOMMU`	Usa memoria virtual de NetBSD con soporte para *PCI passthrough*, *IOMMU* y *SR-IOV* para aislar y compartir hardware.
Sistema de archivos	`ext3 / NFS / NVMe`	Emplea ext3 sobre NVMe y un servidor NFS en Rumprun para compartir volúmenes por red con buen rendimiento.
Controladores de dispositivos	`NetBSD drivers / rumprun-smp`	Reutiliza drivers de NetBSD (NIC, NVMe), operando igual en modo aislado o *library-OS*.
Red (Network stack)	`NetBSD TCP/IP / servidor de red VIF`	Ejecuta la pila TCP/IP en la app o un servidor de red, usando canales virtuales (VIF) para el reenvío seguro.
Servicios y compatibilidad POSIX	`rumprun / NetBSD libc`	Soporta servicios POSIX/BSD comunes; algunas llamadas como `fork()` aún no están implementadas.

Fuente: Elaboración propia con base en (Ruslan Nikolaev and Mincheol Sung and Binoy Ravindran, 2021).

Gestión de procesos y planificación

Gestiona el intercambio de mensajes entre procesos del sistema y el microkernel. Esta arquitectura refuerza el aislamiento y la estabilidad, ya que los servicios del sistema operan como procesos independientes que se comunican de forma controlada.

Gestion de memoria

Cada dominio maneja su propia memoria virtual usando los mecanismos estándar de NetBSD sobre Rumprun. Para acceso directo a hardware usa PCI passthrough e IOMMU, lo que permite apartamiento seguro entre dominios. Además, soporta SR-IOV para dividir dispositivos físicos NICs o NVMe en funciones virtuales asignables a cada VM.

Sistema de archivos

Reutiliza el sistema de archivos de NetBSD. Una instancia de Rumprun con un volumen ext3 sobre NVMe actúa como servidor NFS, exportando volúmenes por red. Los clientes montan este recurso mediante NFS, demostrando buen rendimiento en pruebas de E/S.

Controladores de dispositivo

Reutiliza los controladores de NetBSD como por ejemplo, ixgbe para NICs Intel 10GbE y NVMe. Los mismos drivers funcionan en ambos modos de operación como para aislado o library-OS, permitiendo cambiar dinámicamente de modo sin reemplazar controladores.

Red

Ejecuta la pila TCP/IP de NetBSD dentro del espacio de aplicación o en un servidor de red dedicado. Este servidor reenvía tramas L2 mediante canales virtuales VIF. Si el servidor falla, el estado TCP se mantiene en la aplicación, lo que permite su recuperación.

Servicios y compatibilidad POSIX

LibrettOS es compatible con POSIX/BSD, soportando servicios como SSH, bases de datos y utilidades estándar. Algunas llamadas complejas como fork() y pipe(), aún no están implementadas, pero se consideran alcanzables en futuras versiones.

1.6.7 Herramientas utilizadas (compiladores, emuladores, etc.)

Según (Ruslan Nikolaev and Mincheol Sung and Binoy Ravindran, 2021), LibrettOS se basa en tecnologías como rump kernels y el uso de POSIX.

- **Rump kernels**: Utilizados para ejecutar sistemas operativos como *anykernels* fuera del kernel monolítico de NetBSD.
- **Rumprun**: Un unikernel basado en *rump kernels* para ejecutar aplicaciones directamente con menos sobrecarga del sistema operativo.
- **Hipervisores como Xen y KVM**: Usados para gestionar máquinas virtuales.
- **IOMMU y PCI Passthrough**: Herramientas para dar acceso directo a dispositivos físicos desde VMs.

- **DPDK y SPDK**: Bibliotecas de alto rendimiento para el acceso directo a hardware en aplicaciones de red y almacenamiento.

1.6.8 Nivel de complejidad y accesibilidad para estudiantes

Según el repositorio (The rumpkernel project, 2025), describen que LibrettOS es un prototipo experimental y advierte en un nota que no esta diseñado para uso diario normal, usar bajo su propia responsabilidad. Entonces esto quiere decir que no esta diseñado como un proyecto educativo fácil de usar, sino como una plataforma de investigación. Por lo tanto su dificultada esta alta, pero si es accesible a toda el código mediante el repositorio y a la web oficial de LibrettOS.

1.7 freeRTOS

1.7.1 Nombre del proyecto o sistema operativo

FreeRTOS — Kernel de tiempo real (RTOS) de código abierto orientado a microcontroladores y microprocesadores de recursos limitados. Su historia y mantenimiento han estado ligados a Richard Barry / Real Time Engineers Ltd., y desde 2017/2018 AWS mantiene y distribuye componentes y bibliotecas asociadas (Amazon FreeRTOS/AWS FreeRTOS). (Wikipedia contributors, 2024a) sistema operativo en tiempo real basado en UNIX, diseñado para ser pequeño y eficiente, ideal para sistemas embebidos y dispositivos con recursos limitados.

1.7.2 Enlace al repositorio y/o documentación oficial

- **Pagina oficial:** https://www.freertos.org/?
- **Enlace al repositorio oficial:** https://github.com/FreeRTOS/FreeRTOS?
- **Enlace al libro oficial:** https://www.freertos.org/media/2018/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf?

1.7.3 Objetivo del proyecto

FreeRTOS es un sistema operativo de tiempo real (RTOS) de código abierto, diseñado para ser ligero y portable en microcontroladores. Su kernel provee planificación de tareas, comunicación y temporizadores para sistemas embebidos. Además de su uso industrial, es una herramienta educativa y experimental fundamental. Permite aprender y experimentar con conceptos de sistemas en tiempo real de manera práctica.

(Nannan He, Han-Way Huang, 2020, pag. 19)

1.7.4 Lenguaje(s) de implementación

El núcleo de FreeRTOS está escrito principalmente en C (para facilitar portabilidad) con pequeñas partes en ensamblador según la arquitectura (rutinas de cambio de contexto)(Wikipedia contributors, 2024a),

1.7.5 Arquitectura del sistema (monolítica, microkernel, etc.)

- FreeRTOS tiene una arquitectura de tipo **microkernel**, es decir, el núcleo es muy reducido y solo gestiona lo esencial (planificación de tareas, comunicación entre procesos y gestión de memoria).
- El sistema se organiza en **capas**:
  - Una capa independiente de hardware (código común para todas las arquitecturas)
  - Una capa dependiente de hardware (abstracción para cada CPU/compilador)
  - Por encima de ellas el código de usuario (tareas e ISRs)

(Amy Brown, Greg Wilson, 2014, pag. 38-40) la siguiente figura muestra las capas de sofware: 1.7.

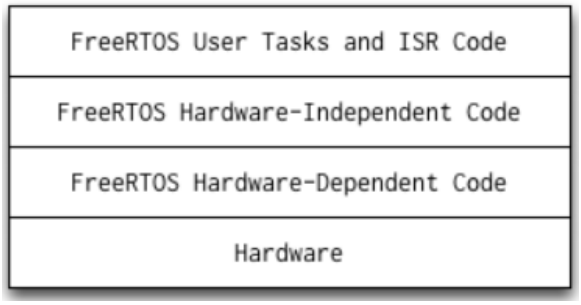


Figura 1.7: Capas de FreeRTOS (usuario/ISRs, núcleo independiente de HW, capa dependiente de HW, hardware)

Fuente: Obtenido de (Amy Brown, Greg Wilson, 2014, pag. 40) *The Architecture of Open Source Applications, Volume II.*

1.7.6 Componentes implementados (procesos, memoria, archivos, etc.)

FreeRTOS es un RTOS minimalista, por lo que implementa principalmente:

Tabla 1.8: Componentes principales de FreeRTOS

Componente / Sub-sistema	**Nombre en FreeRTOS**	**Funcionamiento técnico (idea clave)**
Gestión de tareas (threads)	tasks.c, list.c	Creación y planificación de tareas con prioridades preemptivas. Manejo de cambio de contexto en cada tick del sistema.
Comunicación / sincronización	queue.c, queue.h	Colas de mensajes FIFO, semáforos binarios y mutexes para sincronizar tareas/ISRs.
Temporización	timers.c	Temporizadores por software (delay, callbacks periódicos) con tarea de servicio propio. Tick del sistema configurable (10-1000 Hz).
Gestión de memoria	heap_1.c a heap_5.c	Múltiples esquemas de heap para asignación dinámica. Sin memoria virtual ni paginación.
Interrupciones	port*.c	Mínimo soporte en núcleo: tick del hardware y enmascaramiento de interrupciones críticas. APIs especiales para ISRs.
Eventos y temporales avanzados	event_groups.c	Grupos de flags (eventos) para sincronización múltiple entre tareas.

Gestión de tareas

Las tareas son las unidades básicas de ejecución en FreeRTOS, gestionadas por el planificador del sistema. Cada una posee su propia pila (*stack*) y estructura de control (TCB). Se crean mediante `xTaskCreate()` y se sincronizan usando los mecanismos del kernel.(Richard Barry, 2018)

Comunicación / sincronización

FreeRTOS proporciona diversos mecanismos para la coordinación entre tareas e interrupciones. Entre estos se incluyen colas (`xQueueSend`/`xQueueReceive`), semáforos (binarios y de contaje), mutexes y grupos de eventos. Estas primitivas constituyen la base fundamental para la comunicación y sincronización dentro del sistema.(Richard Barry, 2018)

Temporización

FreeRTOS incorpora temporizadores software que permiten programar la ejecución de funciones callback después de un intervalo específico o de forma periódica. Estos timers son gestionados a través de una API especializada que facilita la implementación de acciones temporizadas dentro de las aplicaciones.(Richard Barry, 2018)

Gestión de memoria

FreeRTOS dispone de cinco implementaciones de memoria dinámica (`heap_1` a `heap_5`) con diferentes estrategias de asignación. Estas van desde métodos básicos hasta esquemas avanzados que gestionan la fragmentación. La selección permite adaptarse a los requerimientos específicos de cada aplicación embebida.(Richard Barry, 2018)

Interrupciones

FreeRTOS gestiona las interrupciones mediante archivos `port*.c` con soporte mínimo en el núcleo. Se encarga del tick del sistema y del enmascaramiento de interrupciones críticas. Proporciona APIs específicas para una comunicación segura entre las ISRs y las tareas.(Richard Barry, 2018)

Eventos y temporales avanzados

La gestión de eventos avanzados se realiza mediante `event_groups.c`, que implementa grupos de flags para sincronización múltiple entre tareas. Este mecanismo permite que varias tareas esperen o señalen múltiples condiciones de forma eficiente, facilitando la coordinación de operaciones complejas en el sistema.

1.7.7 Herramientas utilizadas (compiladores, emuladores, etc.)

Compiladores

FreeRTOS soporta prácticamente cualquier herramienta estándar del ecosistema embebido:

- GCC (`arm-none-eabi-gcc`, AVR-GCC)
- IAR Embedded Workbench
- Keil uVision (ARMCC)
- CodeWarrior
- Microchip XC

En general, se utiliza el *toolchain* propio de la plataforma objetivo.

Simuladores y Emuladores

- Puertos para Windows (Visual C++) y POSIX (Linux) que permiten ejecutar aplicaciones FreeRTOS en simulación.
- Demos en QEMU para varias arquitecturas (ARM, RISC-V, etc.).
- Demos preconfiguradas para placas reales (STM32, ESP32, etc.).

(Amy Brown, Greg Wilson, 2014, pag. 40–41).

1.7.8 Nivel de complejidad y accesibilidad para estudiantes

FreeRTOS posee un kernel minimalista que facilita su aprendizaje inicial al ser más simple que sistemas como Linux. Sin embargo, su uso efectivo como sistema de tiempo real exige dominio de programación en C y arquitectura de microcontroladores para manejar concurrencia e interrupciones.

El proyecto ofrece documentación extensa y ejemplos prácticos como "blinky" que facilitan el aprendizaje. Es adecuado para estudiantes avanzados con bases en sistemas embebidos, aunque dominar sus conceptos avanzados representa un desafío formativo significativo. (Nannan He, Han-Way Huang, 2020).

1.8 HelenOS

1.8.1 Nombre del proyecto o sistema operativo

El nombre del sistema operativo es "HelenOS". El proyecto fue iniciado por Jakub Jermar en 2001, en ese momento era un código independiente que funcionaba como kernel para IA-32. (Děcký, 2015)

1.8.2 Enlace al repositorio y/o documentación oficial

- **Enlace al repositorio oficial:** https://github.com/HelenOS/helenos
- **Documentación oficial y Wiki:** https://www.helenos.org/

1.8.3 Objetivo del proyecto

Según (Děcký, 2015), este sistema operativo, tiene un enfoque educativo y experimental. Además, busca servir como una plataforma para la investigación y desarrollo de sistemas operativos de propósito general, teniendo muy en cuenta la fiabilidad y practicidad.

1.8.4 Lenguaje(s) de implementación

Según (Děcký, 2010) y confirmando con el repositorio oficial (Jermář, 2025), el sistema operativo HelenOS está principalmente implementado en C, con algo de apoyo de C++, Meson como "build system", algunos componentes escritos en ensamblador para tareas de bajo nivel y python para scripts auxiliares.

1.8.5 Arquitectura del sistema (monolítica, microkernel, etc.)

El sistema operativo HelenOS se baso en la arquitectura multiserver de microkernel, multiserver es un tipo de arquitectura microkernel, donde el núcleo del sistema operativo (microkernel) es responsable de las funciones básicas para el funcionamiento del sistema operativo como gestión de memoria, comunicación de procesos y scheduling; y por otro lado, los demás servicios del sistema operativo se implementan como "servidores" y se ejecutan en el espacio de usuario. De esta manera, los servidores pueden ser desarrollados, cambiados y reiniciados de manera independiente sin afectar al kernel (Děcký, 2015). A continuación, se incluye una vista general de la arquitectura/organización del kernel de HelenOS:

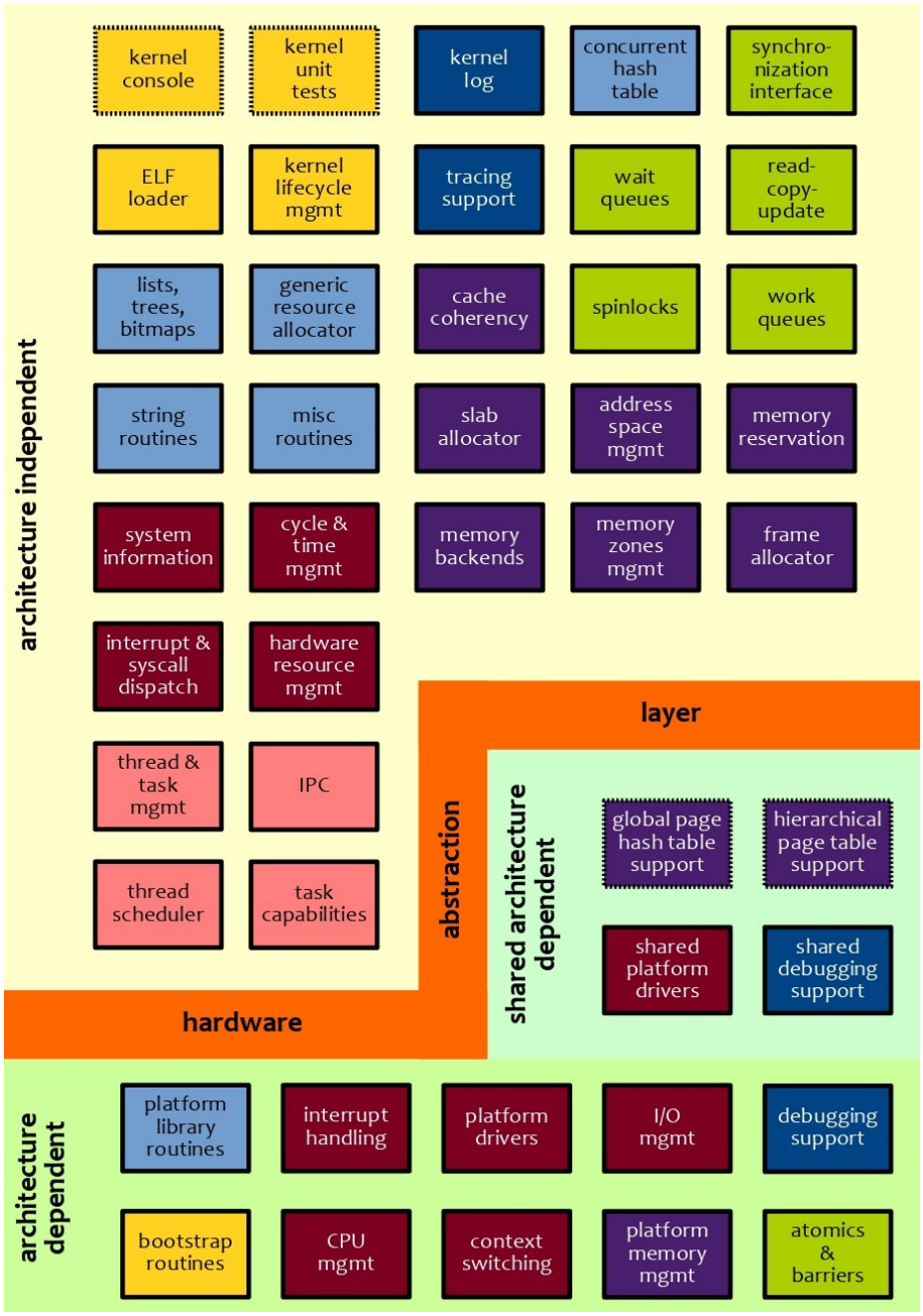


Figura 1.8: Vista general de la arquitectura/organización del kernel de HelenOS.

Fuente: Obtenido de (Děcký, 2015) *Application of Software Components in Operating System Design.*

1.8.6 Componentes implementados (procesos, memoria, archivos, etc.)

En HelenOS, cada componente del sistema operativo se implementa como una tarea aislada con su propio espacio de direcciones (siguiendo la arquitectura multiserver), identificador único y comunicación mediante interfaces bien definidas (IPC o syscall API) (Děcký, 2015). A continuación se presenta una tabla con los principales componentes y subsistemas de HelenOS, su nombre de archivo en el repositorio oficial y la idea técnica detrás de su implementación.

Tabla 1.9: Funcionamiento técnico (idea clave) de los principales subsistemas de HelenOS

Componente / Subsistema	**Nombre en HelenOS (tarea / servicio)**	**Funcionamiento técnico (idea clave)**
Gestión de memoria	`frame allocator`, `slab allocator`, `address space mgmt`	Asignación de marcos físicos con separación por tareas.
Planificación / multitarea	`thread scheduler`, `task mgmt`	Aislamiento por espacio de direcciones y comunicación por IPC.
Sistema de archivos	`TMPFS`, `FAT`, `ext4`, `MINIX FS`	Implementados como tareas independientes, comunicadas vía VFS.
Subsistemas de E/S y drivers	Ej. `rtl8139`, `ahci`, `ps2`, `usb hid`	Se usan como tareas separadas, organizados en el árbol de dispositivos.
Servicios de red	`inetsrv`, `tcp`, `udp`, `ethip`	Componentes aislados, comunicados por IPC.
Servicios de usuario / consola	`console`, `clipboard`, `compositor`, `bdsh`	Tareas independientes con comunicación por IPC.
Servicios de nombres y localización	`naming service`, `location service`	Servicios singleton para identificar y vincular componentes.

Fuente: Elaboración propia con base en (Děcký, 2015; Jermář, 2025).

Gestión de memoria

En HelenOS, se implementan los componentes `frame allocator`, `slab allocator` y `address space management` para asignar marcos físicos, gestionar zonas de memoria y mantener espacios de direcciones separados por tarea. Esta separación es para encapsular el estado de cada componente y facilitar la verificación de corrección (Děcký, 2015, p. 39).

Planificación / multitarea

La planificación se realiza mediante `thread scheduler` y `task management`. Le generan a cada tarea su propio espacio de direcciones y permiten su comunicación con otras mediante IPC (Děcký, 2015, p. 38).

Sistema de archivos

HelenOS soporta múltiples sistemas de archivos como `TMPFS`, `FAT`, `ext4`, `MINIX FS`, cada uno implementado como una tarea independiente; se comunican con el sistema mediante el `vfs`, el cual es un intermediario entre drivers y aplicaciones (Děcký, 2015, p. 40).

Subsistemas de E/S y drivers

Los drivers están organizados jerárquicamente en el árbol de dispositivos (como se muestra en la figura). Cada driver (como `rtl8139`, `ahci`, `ps2`, `usb hid`) se implementa como una tarea separada, para permitir modularidad, aislamiento y reemplazo en ejecución (Děcký, 2015, p. 41).

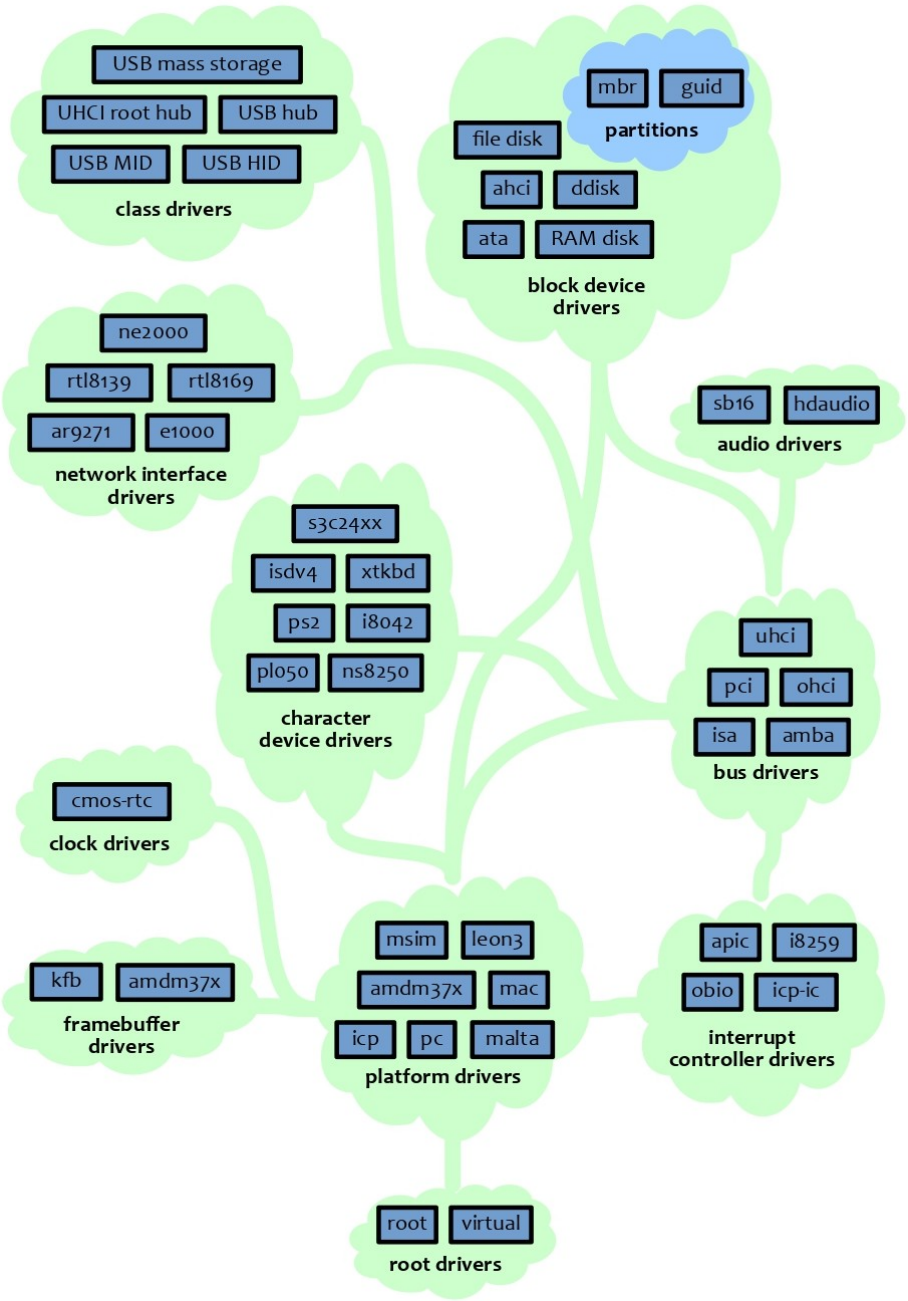


Figura 1.9: Árbol de drivers en HelenOS.

Fuente: Obtenido de (Děcký, 2015) *Application of Software Components in Operating System Design.*

Servicios de red

Los protocolos de red `tcp`, `udp`, `ethip` y el servicio `inetsrv` se implementan como componentes independientes. Siguiendo así la arquitectura multiserver (Děcký, 2015, p. 40).

Servicios de usuario / consola

Los componentes `console`, `clipboard`, `compositor` y el shell `bdsh` se ejecutan como tareas separadas (Děcký, 2015, p. 40).

Servicios de nombres y localización

Son servicios singleton, específicamente `naming service` y el `location service`, que permiten identificar y vincular componentes en el sistema (Děcký, 2015, p. 38).

1.8.7 Herramientas utilizadas (compiladores, emuladores, etc.)

HelenOS utiliza herramientas para el lenguaje C, también usa máquinas virtuales, sistemas de construcción e implementaciones. Entre sus principales herramientas están:

- **Compilador:** Uso de GCC y Clang, con toolchains cruzadas para las diferentes arquitecturas soportadas (x86, ARM, MIPS, SPARC, PowerPC e Itanium) (Jermář, 2025).
- **Emuladores y máquinas virtuales:** Para ejecutar HelenOS se usa el entorno "QEMU", para probar, depurar y validar el ISO del microkernel (Jermář, 2025).

- **Sistema de construcción:** Makefile con scripts Bash, Python y uso de MESON (experimental) para apoyar la compilación cruzada de componentes (Jermář, 2025).

- **Herramienta de portabilidad:** *HelenOS Coastline*, un repositorio desarrollado por los creadores del microkernel, puede ser usado para la integración de software externo más fácilmente.

1.8.8 Nivel de complejidad y accesibilidad para estudiantes

Considerando un entorno de estudio de sistemas operativos, justamente el entorno en el que se propicia esta investigación, con conocimientos previos de C y fundamentos de sistemas operativos; es un proyecto con un nivel de complejidad media, más que nada por su tamaño (es extenso), además del uso de servicios mediante IPC y también considerar que la arquitectura multiserver, que implica microkernel, no es la más sencilla tampoco. Y eso sin contar la diversidad de arquitecturas soportadas, lo cual añade dificultad al análisis del código.

Se consideró en la categorización del nivel de complejidad que, la accesibilidad para los estudiantes es bastante buena, ya que el proyecto es open source y cuenta con una gran cantidad de estudios en torno a él. Sin embargo, una consideración a tomar en cuenta es que, aunque HelenOS dispone de documentación formal (guías, artículos, documentación generada), esta puede no detallar cada componente del sistema operativo con la exhaustividad que otros sistemas operativos grandes ofrecen (por ejemplo un libro completo o wiki ultra-detallada).

1.9 Haiku OS

1.9.1 Nombre del proyecto o sistema operativo

Según (Ryan Leavengood, 2012), Haiku Os es un proyecto visionario con el nombre del sistema operativo Haiku OS en la actualidad, pero que antes era OpenBeOS, un sistema operativo fundado en la División de Corporaciones del Estado de Nueva York como una organización sin fines de lucro en julio del 2003. por su fundador Michael Phipps. El nombre del sistema operativo proviene de la palabra japoness "Haiku", por que era usada en los mensajes de error de BeOS, eran mensajes presentados de forma poética japonesa.

1.9.2 Enlace al repositorio y/o documentación oficial

- **Repositorio oficial (espejo en GitHub):** https://github.com/haiku/haiku — funciona como un *mirror* del repositorio principal alojado en el servidor de Haiku, donde se gestiona el desarrollo activo del sistema operativo. (Haiku Project)
- **Repositorio principal (Gerrit):** https://review.haiku-os.org/haiku — servidor oficial utilizado para la revisión y aprobación del código fuente. (Haiku Project)
- **Sitio web oficial del proyecto:** https://www.haiku-os.org/get-haiku/installation-guide/

1.9.3 Objetivo del proyecto

Según (Stimac, Miroslav, 2011) Haiku es un proyecto comunitario orientado al escritorio experimental, como meta ofrecer un sistema operativo de escritorio abierto, ligero y fácil de usar, diseñado especificamente para la computación personal inspirado en BeOS, Haiku se creo con la finalidad de recrear experiencia de uso para entender el funcionamiento y realizar mejoras.

1.9.4 Lenguaje(s) de implementación

Haiku esta implementado en C++, haciendo uso de una API de Be orientada a objetos (The kits) que facilita el desarrollo de aplicaciones y controladores de dispositivos (Ryan Leavengood, 2012). incluye contenedores de la Biblioteca de Plantillas Estandar de C++ (STL) y otras bibliotecas de terceros como OpenSSL, SQLite y zlib (Haiku, Inc., 2025).

1.9.5 Arquitectura del sistema (monolítica, microkernel, etc.)

El sistema operativo esta basado en BeOS un Kernel hibrido modular. Su núcleo se desarrollo de una rama de NewOS, la modularidad permite a los controladores y otros componentes que se carguen de manera dinamica todo lo proceso segun sea necesario. La siguientes son las capas principales que conforman el sistema operativo, según (Nakrani, Jatinkumar, 2025):

- **Capa de nucleo:** El nucleo de NewOS sirve para servicio como: multitarea, gestión de memoria, controladores y administrador de archivos.

- **Servidor de aplicaciones:** El servido de aplicaciones gestiona las ventanas, el dibujo y el renderizado. Esta diseñado para un alto rendimiento y se integra con el servidor de entrada y el kit de herramientas.

- **Gestión de paquetes:** Haiku es compatible con Be File System (BFC) que incluye atributos de archivos extendidos.

- **Controlador de red:** Haiku incluye una pila de red modular compatible con estaciones.

En la siguiente imagen es al arquitectura del sistema operativo BeOS (el precursor de HaikuOS). Se puede observar que entre el hardware y las aplicaciones del software se encuentra el software BeOS. El software del sistema operativo tiene 3 capas, capa de micronucleo, capa de servidor y capa de software.

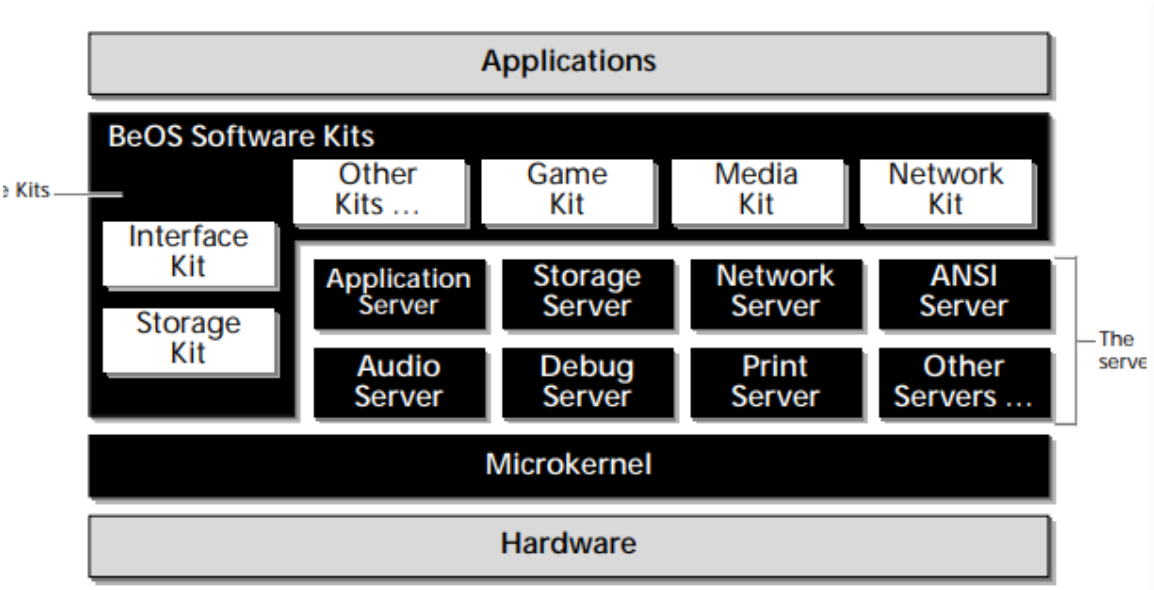


Figura 1.10: Arquitectura del sistema operativo BeOS.

Fuente: Obtenido de (Be, Inc., 2025)

1.9.6 Componentes implementados (procesos, memoria, archivos, etc.)

En HaikuOS, como es característico en sistemas operativos de arquitectura híbrida modular, los componentes principales están implementados y se ejecutan en modo kernel, pero con una alta modularidad organizada en "kits" y servidores. A continuación se presenta una tabla con los principales componentes y subsistemas de HaikuOS, su ubicación en el repositorio oficial y la idea técnica detrás de su implementación.

Tabla 1.10: Funcionamiento técnico (idea clave) de los principales subsistemas de HaikuOS

Componente / Sub-sistema	**Nombre / Ruta en repositorio**	**Funcionamiento técnico (idea clave)**
Gestión de memoria	`src/system/kernel/vm/`, `paging.cpp`	Asignar páginas, espacio de direcciones por proceso.
Scheduling / multitarea	`src/system/kernel/thread.cpp`, `scheduler.cpp`	Soporte para multiprocesador, planificación preventiva y cambio de contexto eficiente.
Sistema de archivos	`src/add-ons/kernel/file_systems/`	BFS extendido, con journalling; soporte FAT para compatibilidad.
Drivers / E/S	`src/add-ons/kernel/drivers, bus_managers/`	Implementados como módulos independientes y jerarquizados.
Servicios de red	`src/add-ons/kernel/network/`, `src/servers/net_server/`	Pila TCP/IP modular, soporte para sockets BSD, DHCP y servicios de red como servidores.
Interfaz gráfica / servicios de usuario	`src/servers/app/`, `src/servers/input/`, `src/kits/interface/`, `app_server`, `input_server`	Servidores para gestión gráfica y entrada, comunicación mediante kits orientados a objetos (GUI toolkit).

Fuente: Elaboración propia con base en (Project, 2025; Nakrani, Jatinkumar, 2025).

Gestión de memoria

En HaikuOS se gestiona la memoria virtual con asignación de páginas bajo demanda, con método indexado y vector de bits para administrar espacio libre; y cada proceso tiene su propio espacio de direcciones. El módulo de "Virtual Memory Manager" se encarga de la asignación física y lógica de memoria (Gonzales Calos Alejandro, Lozano Eliseo de Jesús, 2017).

Scheduling / multitarea

La planificación de hilos en el sistema operativo HaikuOS soporta más de un núcleo. Cada hilo tiene su propio contexto; el scheduler organiza prioridades (para usar el CPU) y cambios de contexto (Nakrani, Jatinkumar, 2025).

Sistema de archivos

En HaikuOS, se tiene el sistema de archivos BFS (Be File System) con metadatos extendidos (pares clave-valor) y búsquedas indexadas. Además tiene soporte para otros sistemas como FAT para compatibilidad (Nakrani, Jatinkumar, 2025; Project, 2025).

Drivers / E/S

Los drivers están organizados en forma de árbol con E/S (la cual está muy optimizada para contenido multimedia) en forma de módulos independientes, para permitir actualizarse en ejecución (Haiku, Inc., 2025; Be, Inc., 2025).

Servicios de red

El sistema operativo HaikuOS, cuenta con un kit (Network Kit) muy completo y el servidor `net_server` para comunicaciones. Tiene soporte para los protocolos TCP/IP y UDP, incluso, si le añades complementos, AppleTalk o IPX (Nakrani, Jatinkumar, 2025; Be, Inc., 2025).

Interfaz gráfica / servicios de usuario

La interfaz gráfica de usuario de Haiku es gestionada por los servidores `app_server` e `input_server`. Los "kits" de desarrollo (Application, Interface) nos proporcionan APIs con el paradigma orientado a objetos en C++ para aplicaciones y servicios (Nakrani, Jatinkumar, 2025).

1.9.7 Herramientas utilizadas (compiladores, emuladores, etc.)

HaikuOS utiliza diversas herramientas para ejecutar, compilar y construir. También usa máquinas virtuales e implementaciones con funcionalidades adicionales. Entre las principales herramientas se encuentran:

- **Compilador:** Uso de GCC 2 (modificado), para mantener compatibilidad con BeOS, y GCC 13 para el desarrollo actual (Yusuf, Mohamad and Al Fatah, M Reza and Fami, Asrul and Pratama, Vemas Adi and Z, M Fauzan and Syadam, Muhammad, 2024).

- **Emuladores y máquinas virtuales:** En (Yusuf, Mohamad and Al Fatah, M Reza and Fami, Asrul and Pratama, Vemas Adi and Z, M Fauzan and Syadam, Muhammad, 2024) se hace uso de VirtualBox y en su página oficial (Haiku

Project, 2025) se recomienda usar QEMU, así que HaikuOS puede ejecutarse en ambos entornos.

- **Sistema de construcción:** Haiku modifica la herramienta "Jam" para compilar el kernel, los drivers y la interfaz gráfica. También ofrece compilación cruzada para otras arquitecturas (Haiku Project, 2025).

- **Herramientas de portabilidad:** En el repositorio (Project, 2025), se encuentran scripts *raw disk*, *MMC* o *SD card* para generar imágenes en diferentes arquitecturas.

1.9.8 Nivel de complejidad y accesibilidad para estudiantes

Considerando un entorno académico de estudio de sistemas operativos, justamente el ambiente para el cual se propicia esta investigación, con conocimientos previos de C (el antecesor de C++) y fundamentos de sistemas operativos; HaikuOS es un proyecto con nivel de complejidad media, debido a su arquictectura híbrida modular, la cual no es muy común, y a la cantidad de componentes implementados, y la profundidad de los mismos (por ejemplo, la interfaz gráfica y optimización para multimedia). Además de la herramientas utilizadas para su construcción y portabilidad.

También se consideró para colocarlo en el nivel medio, que la accesibilidad para los estudiantes es buena, ya que el proyecto es open source, aunque no cuenta con tantos estudios en torno a él como otros sistemas. Su página web oficial es bastante completa, y el repositorio oficial está bien organizado. Sin embargo, no posee una documentación formal tan detallada como otros sistemas operativos.

1.10 FreeDOS

1.10.1 Nombre del proyecto o sistema operativo

El sistema operativo FreeDOS, su nombre proviene de la idea de ofrecer una alternativa libre y abierta al sistema operativo MS-DOS, tras el anuncio de Microsoft en 1994 de interrumpir el soporte a este último (FreeDOS Project, 2025b).

1.10.2 Enlace al repositorio y/o documentación oficial

- **Repositorio oficial:** https://github.com/FDOS
- **Sitio web del proyecto:** https://www.freedos.org/

1.10.3 Objetivo del proyecto

El objetivo principal de FreeDOS, además de sus enfoques educativo y experimental, es proporcionar un sistema operativo compatible con MS-DOS completamente libre, de código abierto y mantenido por la comunidad. Busca preservar la capacidad de ejecutar aplicaciones y controladores diseñados originalmente para DOS, especialmente aquellas dependientes de entornos de 16 bits (FreeDOS Project, 2025b).

1.10.4 Lenguaje(s) de implementación

FreeDOS está desarrollado principalmente en `C`, con secciones escritas en `x86 Assembly`, especialmente en el núcleo y los controladores de dispositivo. El lenguaje C se utiliza para la lógica general del sistema, mientras que el ensamblador proporciona control de bajo nivel sobre interrupciones, rutinas BIOS y gestión de hardware. Este enfoque

híbrido permite combinar eficiencia y compatibilidad con el hardware x86, tal como lo hacía MS-DOS (FreeDOS Project, 2025a).

1.10.5 Arquitectura del sistema

FreeDOS hereda la arquitectura clásica de MS-DOS, organizada en capas que separan el hardware del usuario mediante el BIOS y un núcleo DOS. El sistema se ejecuta en modo real (16 bits), operando directamente sobre la arquitectura Intel x86 sin protección de memoria ni multitarea.

Se presenta un esquema adaptado que representa la arquitectura del sistema operativo MS-DOS en que se basa FreeDOS.

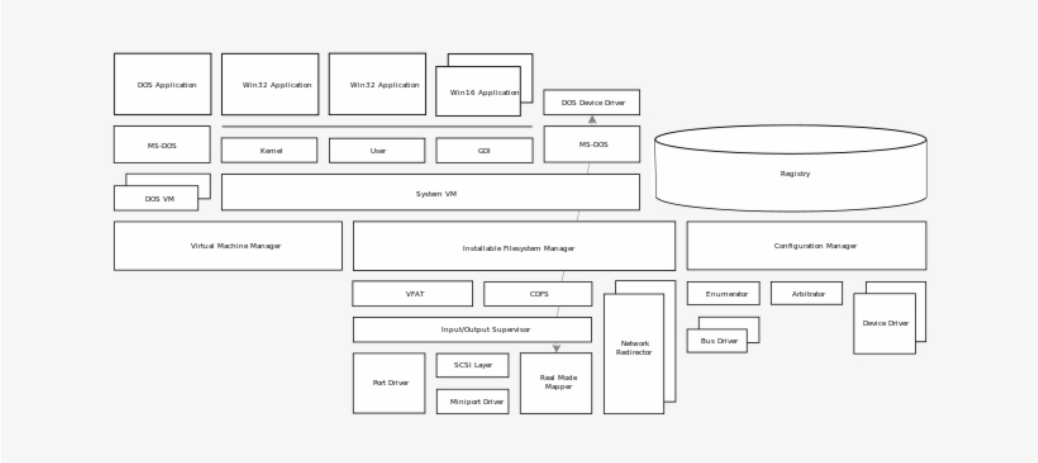


Figura 1.11: Arquitectura clásica de MS-DOS, base de FreeDOS.

Fuente: Adaptado de Microsoft Press, (NicePNG - Image uploader, 2018), *FreeDOS Kernel: An MS-DOS Emulator* (1996).

1.10.6 Componentes implementados (procesos, memoria, archivos, etc.)

FreeDOS es un sistema operativo de código abierto que emula el comportamiento del MS-DOS original. Está diseñado para ser compatible con aplicaciones y controladores de dispositivos escritos para MS-DOS, proporcionando una plataforma para ejecutar software antiguo en hardware moderno. A continuación, se describen los principales componentes y subsistemas implementados en FreeDOS:

Tabla 1.11: Funcionamiento técnico de los principales subsistemas de FreeDOS

Componente / Subsistema	**Nombre en FreeDOS (archivo)**	**Funcionamiento técnico (idea clave)**
Intérprete de comandos	`COMMAND.COM / FreeCOM`	Shell que carga programas, procesa scripts .BAT y maneja redirección/piping básico.
Kernel	`KERNEL.SYS`	Implementa servicios INT 21h (archivos, dispositivos, tiempo), gestión básica de memoria y soporte para TSRs.
Sistema de archivos	Varios drivers FAT	Soporte nativo para FAT12/FAT16/FAT32 sin VFS, con herramientas como COPY, DIR, DEL.
Gestión de memoria	`HIMEM / EMM386`	Modelo clásico DOS: 640KB convencional + UMB, sin protección de memoria ni paginación.
Gestión de procesos	(Parte del kernel)	Multitarea cooperativa nativa, soporte para TSRs, sin procesos aislados ni preemptión.
Drivers y dispositivos	Varios `.SYS / .EXE`	Drivers para discos, CD-ROM, teclado/ratón; algunos actualizados para hardware moderno.

Fuente: Elaboración propia con base en la documentación técnica de FreeDOS.

Intérprete de comandos

Proporciona la interfaz de línea de comandos para interactuar con el sistema operativo. Su función principal es cargar programas ejecutables, procesar scripts por lotes (.BAT) y manejar características básicas como redirección de entrada/salida y piping entre comandos. FreeDOS incluye su propio COMMAND.COM y alternativas como FreeCOM. (Pat Villani, 2017, pag. 240–242)

Kernel

Implementa las funciones centrales compatibles con MS-DOS mediante servicios de interrupción 21h para operaciones de archivos, dispositivos, tiempo y memoria. Gestiona memoria convencional y superior (UMB), soporta programas TSR (Terminate and Stay Resident) y proporciona servicios básicos de disco. El kernel de FreeDOS deriva del proyecto DOS-C y está licenciado bajo GPL v2. (Pat Villani, 2017)

Sistema de archivos

Brinda soporte nativo para sistemas de archivos FAT12, FAT16 y FAT32 en modo DOS real. Permite el manejo de particiones FAT y disquetes mediante drivers, con herramientas tradicionales como COPY, DIR y DEL para operaciones básicas. A diferencia de sistemas modernos, FreeDOS no implementa un VFS ni abstracciones complejas de dispositivos. (Pat Villani, 2017, pag. 30–33)

Gestión de memoria

Administra el modelo clásico de memoria DOS con espacio convencional (primeros 640 KB) y memoria superior (UMB) mediante controladores. Soporta extensiones

de memoria a través de controladores como Himem.sys y emm386.exe en entornos compatibles, pero carece de mecanismos modernos como protección de memoria o paginación presentes como en SO modernos. (Pat Villani, 2017, pag. 48)

Gestión de procesos

Coordina la ejecución de programas de forma cooperativa en lugar de preemptiva. Permite el uso de programas TSR (Terminate and Stay Resident) que permanecen en memoria para proporcionar servicios, pero no implementa procesos aislados ni protección entre espacios de direcciones. Proyectos externos pueden añadir capacidades de multitarea, aunque no forman parte del kernel base. (Pat Villani, 2017, pag. 50)

Drivers y dispositivos

Proporciona soporte para hardware mediante controladores para discos, CD-ROM, teclado, ratón y otro hardware clásico. algunos drivers se han actualizado para soportar hardware moderno a través de emuladores, manteniendo la compatibilidad con el ecosistema tradicional de DOS, (Pat Villani, 2017, pag. 52–53)

1.10.7 Herramientas utilizadas (compiladores, emuladores, etc.)

FreeDOS puede instalarse en cualquier emulador o máquina virtual creando un disco virtual e iniciando desde el CD de instalación. Usando QEMU como ejemplo, el proceso es universal para Linux, Windows y Mac.(jim hall, 2018, pag. 15–17)

- **Compiladores:** Utiliza principalmente GCC (GNU Compiler Collection) para la compilación del kernel y las utilidades del sistema, empleando Makefiles para

gestionar el proceso de construcción tanto nativa como cruzada.

- **Lenguajes:** Soporta principalmente lenguajes de programación C y Assembly para el desarrollo del sistema, junto con herramientas de scripting por lotes a través de archivos BAT.

- **Emuladores:** Puede ejecutarse en emuladores populares como DOSBox, QEMU y VirtualBox, lo que facilita su uso y pruebas en hardware moderno sin necesidad de equipos legacy.

- **Arquitectura compatible:** Está diseñado principalmente para arquitectura x86, con soporte para procesadores desde los 8088/8086 hasta sistemas modernos a través de emulación.

1.10.8 Nivel de complejidad y accesibilidad para estudiantes

FreeDOS enseña conceptos históricos y de bajo nivel de PC (BIOS, boot, interrupciones, FAT); es complementario: excelente para aprendizaje de hardware/arranque y compatibilidad, menos apropiado para enseñar técnicas modernas de diseño de SO (ej. virtual memory, protección por procesos).(JIM HALL, 2018)

- **Complejidad técnica:** Abarca desde nivel básico (comandos DOS, scripting BAT, programación con interrupciones) hasta avanzado (desarrollo de kernel, drivers, porting de hardware).

- **Accesibilidad educativa:** Muy alta para laboratorios de arquitectura de computadores, permitiendo estudiar el proceso de arranque (boot sector), interrupciones BIOS/DOS (int 10h, int 21h) y programación en entornos limitados.

Capítulo 2

Comparación técnica

La tabla siguiente presenta una comparación técnica de varios sistemas operativos destacados, considerando aspectos clave como arquitectura, lenguaje de programación, tamaño del kernel, soporte de hardware, documentación y comunidad de usuarios y desarrolladores.

Tabla 2.1: Tabla comparativa resumida de sistemas operativos educativos y experimentales (formato horizontal)

S.O.	**Arquitectura**	**Lenguaje**	**Documentación**	**Comunidad**	**Dificultad**
MINIX	Microkernel	C, ASM	Amplia y académica	Pequeña–media	Baja
XV6	Monolítico simple	C, ASM	Excelente (MIT)	Grande en educación	Muy baja
Theseus	Cells (nanocore)	Rust	Limitada; académica	Muy pequeña	Muy alta
Redox OS	Microkernel híbrido	Rust	Buena; activa	Media–alta	Media–alta
FlexOS	Híbrida modular	C/C++, Rust	Avanzada; técnica	Pequeña	Alta
LibrettOS	Multiserver + Library OS	C/C++, ASM	Académica avanzada	Muy pequeña	Alta
FreeRTOS	Microkernel RTOS	C	Excelente y abundante	Muy grande	Media
HelenOS	Microkernel multiserver	C/C++	Buena; técnica	Media	Media
Haiku OS	Híbrido modular	C++	Buena; enfocada	Media	Media
FreeDOS	DOS real mode	C, ASM	Media; histórica	Media	Baja

Capítulo 3

Análisis crítico

3.1 Reflexión sobre las propuestas más adecuadas para el contexto educativo

Teniendo en cuenta el contexto educativo específico del curso, que prioriza la comprensión general de los fundamentos detrás del funcionamiento de los sistemas operativos sobre las nuevas ideas y enfoques modernos. También, teniendo en cuenta el nivel de complejidad, accesibilidad y documentación disponible en internet; hemos llegado a las siguientes propuestas:

- **Minix:** Este sistema operativo fue diseñado acorde al objetivo de nuestro curso, el hecho de contar con una arquitectura bien definida (microkernel) y documentada, suman puntos a su favor para ser elegida como objeto de estudio y mejora. Además es conisderado un sistema operativo de complejidad baja debido al pequeño tamaño de su código fuente y sus componentes.

- **XV6:** Este sistema también es una excelente opción de estudio, también es acorde al propósito del curso, su arquitectura monolítica es la más sencilla de entender, está muy bien documentada y su código fuente es en extremo pequeño en comparación con otras opciones, ya que tiene aproximadamente solo 10,000 líneas de código. Además, al estar basado en Unix es un sistema ricamente educativo.

A modo de justificar nuestra elección, expondremos a continuación la reflexión sobre las demás propuestas, empezando por las que podrían ser candidatas suplentes, en caso de no poder trabajar con las elegidas anteriormente. Considerando opciones que requieren más investigación y tiempo de estudio debido a su tamaño y complejidad, pero que aún son viables para este proyecto:

- **HelenOS:** Es un proyecto bastante bien documentado (incluye bastantes diagramas) y con un contenido altamente educativo, especialmente para entender modularidad, comunicación por IPC y arquitectura microkernel multiserver. La única razon de estar en esta categoría es su tamaño y complejidad.

- **RedoxOS:** Este sistema operativo también tiene una buena cantidad de documentación; en lo educativo, aparte de la modularidad y seguridad (aislamiento), trae un concepto innovador, los drivers en espacio de usuario. Las razones para estar en esta categoría son su tamaño, complejidad y estar escrito en Rust.

- **FreeRTOS:** Este sistema es una opción interesante; sin embargo, no va acorde al objetivo del curso, ya que es de tiempo real y orientado a sistemas embebidos; sus conceptos más educatviso son concurrencia y temporización.

Sobre los demás sistemas, como TheseusOS, FlexOS y LibrettOS, fueron catalogados en este entregable en el rango de dificultad de media-alta a altísima, debido a que son bastante innovadores con paradigmas inexplorados (TheseusOS) o bien son sistemas muy nuevos (Librettos) o tienen dependencias técnicas muy avanzadas (FlexOS); estos tres sistemas están más orientados a proyectos de investigación muy específicos y proyectos de posgrado. Sobre HaikuOS y FreeDOS, HaikuOS fue descartado debido a que la mejor referencia no es accesible de forma gratuita, y la documentación disponible no está bien organizada, además mucha de la documentación se basa más en BeOS (un sistema antiguo), que en lo nuevo de HaikuOS; con FreeDOS, fue descartado para este proyecto debido a su antigüedad y falta de características modernas, sin embargo, aún podría ser estudiado para comprender el arranque de la PC, cabe resaltar que posee una gran documentación y estudios a sus alrededor.

3.2 Identificación de patrones comunes y enfoques divergentes

3.2.1 Patrones Comunes

- **Modularidad y aislamiento:** La mayoría de sistemas analizados usan arquitecturas que permiten modularidad, como microkernel, híbridas o incluso la arquitectura en cells de TheseusOS, en general los proyectos modernos buscan la modularidad en sus componentes para facilitar la corrección de errores sin reiniciar.

- **Lenguaje de implementación:** Casi todos los sistemas operativos analizados

se implementan en lenguajes de bajo nivel, como C, C++ y emsamblador, solo algunos proyectos modernos orientados a seguridad utilizan Rust como lenguaje principal.

- **Virtualización:** Usar herramientas de virtualización como QEMU o VirtualBox es casi universal, para desarrollar, probar y ejecutar; en casi todos los proyectos analizados.

- **Comunicación por IPC:** Los sistemas que tiene una arquitectura modular, usan mensajes IPC para poder comunicar sus componentes entre sí, dentro de los cuales también se encuentran patrones com la implementación de multitarea y scheduling de procesos.

3.2.2 Enfoques Divergentes

- **Arquitectura en cells:** La arquitectura de TheseusOS es una manera única e innovadora de diseñar un sistema operativo, con un enfoque en modularidad extrema para mejorar la recuperación ante fallos y actualización en caliente (hot swap). Consideramos una idea bastante robusta y novedosa para implementar al proyecto propio. Sin embargo, la complejidad de un arquitectura en cells es altísima, razón por la cual no será implementada en este proyecto.

- **Administración de privilegios:** TheseusOS diverge de los demás sistemas analizados, al presentar un único nivel de privilegios acompañado de un único espacio de direcciones, este enfoque es muy atractivo para implementar en el proyecto propio porque permite una gran funcionalidad y robustez con una idea bastante simple (SAS).

- **Flexibilidad de aislamiento:** En FlexOS el nivel de aislamiento de un componente es flexible, o sea puede ser ajustado según conveniencia, para obtener las ventajas de una arquitectura monolítica o microkernel según se quiera.

- **Enfoque en hardware:** LibrettOS se caracteriza por su reutilización de drivers y la pila de NetBSD, utiliza hipervisores y hardware especializado para un aislmiento profundo.

- **Enfoque histórico:** FreeDOS es un proyecto enfocado en comprender la arquitectura x86 y el arranque BIOS de la PC, funciona en 16 bits y carece de casi todos los conceptos de sistemas operativos modernos.

3.3 Dificultades técnicas recurrentes

Estos sistemas en general presentan las siguientes dificultades técnicas:

- **Aprendizaje del lenguaje:** Los sistemas operativos analizados están escritos en C, C++ o Rust, lenguajes cuya curva de aprendizaje es bastante brusca y sobre todo al crear un sistema operativo, con conceptos tan díficiles como templates, ownership y lifetimes.

- **Complejidad de compilación:** Muchos de los sistemas analizados requieren configuraciones de compilación complejas, como toolchains de compilación cruzada, algunos incluso piden entornos de construcción especializados. Además, el más común "QEMU" tampoco es intuitivo de configurar y no posee tanta documentación en español.

- **Virtualización avanzada:** La instalación de LibrettOS requiere del uso de hipervisores y hardware especializado, y si no se cuenta con ello, se debe recurrir a emuladores todavía más complicados y especializados.

- **Abstracción de arquitectura:** Entre los sistemas analizados, resalta la arquitectura microkernel multiserver, la cual exige una organización y planificación exhaustiva de los componentes a implementar, exige pensar el servicios y servidores de estos, implica mucha abstracción y modularidad.

- **Conocimiento de bajo nivel:** Para optimizar el funcionamiento de sistemas operativos, es necesario comprender de manera profunda como funciona la interacción con el hardware y los recursos de la computadora, por lo que es necesario conocer emsamblador para el manejo de interrupciones, gestión de memoria y demás tareas del SO.

3.4 Posibles fuentes de inspiración para el proyecto propio

Para el proyecto propio, en el tercer entregable se podrían extraer ideas de los siguientes sistemas operativos analizados:

- **MINIX:** MINIX sirve como motivación por su arquitectura basada en microkernel, ya que los servicios del sistema operativo se ejecutan en procesos separados, el diseño ayuda en la comprensión de los conceptos de aislamiento e independencia entre componentes. Para nuestro proyecto, podemos emularlo manteniendo un código base reducido con fines educativos, así como su organización modular

del código (por ejemplo, mediante directorios o archivos separados como fs/, , sh/, ipc.c, syscall.c). Además, el uso de IPC como mecanismo principal para una comunicación controlada entre procesos, es clave. También puede desarrollarse un sistema de archivos básico compatible con operaciones de montaje, que siga esta misma lógica de simplicidad y claridad en el diseño.

- **XV6:** La motivación para considerar XV6 es su simplicidad estructural, se trata de un sistema operativo cuyo código fuente es lo suficientemente conciso para ser analizado y comprendido en su totalidad en un semestre académico. Para nuestro proyecto, adoptar este enfoque es conveniente para desarrollar un núcleo (kernel) de tamaño reducido, con un diseño limpio y accesible para su estudio. Sería conveniente el uso de un round-robin y una organización del código en módulos educativos bien diferenciados (por ejemplo: proc.c, vm.c, fs.c o pipe.c). A su vez, se pueden utilizar los pipes (tuberías) como un mecanismo simple y replicable de comunicación entre procesos (IPC), replicando el modelo clásico de espacio de direcciones de Unix (que divide la memoria en regiones para código, datos, heap y stack, incluyendo una página de protección).

- **RedoxOS:** De RedoxOS nos inspira su modernidad, seguridad y modularidad a través de su desarrollo en Rust. El proyecto permite diseñar un kernel mínimo que incluya servicios externos, como drivers, sistemas de archivos o redes, utilizando Rust o integrando principios de seguridad de memoria en las partes más críticas del sistema operativo. Es importante crear un mecanismo de comunicación modular, similar a los schemes, con el objetivo de unificar la API de los servicios y implementar un sistema de archivos sencillo inspirado en RedoxFS.

La separación clara entre el kernel y los servicios debe priorizar el aislamiento y la estabilidad del sistema, y se puede lograr utilizando QEMU junto con contenedores para asegurar un flujo de desarrollo reproducible.

- **HelenOS:** Lo que más nos servirá de inspiración de este sistema operativo es su arquitectura microkernel multiserver, donde la idea clave es implementar un kernel muy pequeño y manejar los servicios en el espacio de usuario. Al igual que en Theseus (el cual también fue una inspiración, aunque no es viable hacer el tercer entregable sobre él), se aplica la modularidad para permitir recuperación ante fallos sin reiniciar el kernel, además de diseñar componentes aislados con su propio espacio de direcciones. Se usará comunicación por mensajes IPC, para comunicar los componentes y es recomendable separar los drivers en forma de un árbol, de manera jerarquizada.

Referencias

Amy Brown, Greg Wilson (2014). The Architecture of Open Source Applications, Volume II. accessed: 22-septiembre-2025.

Andrew S. Tanenbaum and Albert S. Woodhull (2006). *Operating Systems: Design and Implementation*. Prentice Hall, 3rd edition. Accedido el 13 de octubre de 2025.

Be, Inc. (2025). The BeOS Bible: BeOS Kits. Espejo (mirror) de la documentación oficial de BeOS. Documentación original ca. 1998-2000. Accedido el 31 de octubre de 2025.

Boos, K. (2024). Theseus — source code (github repository). https://github.com/theseus-os/Theseus. Repositorio accedido el 22 Octubre 2025.

Boos, K., Liyanage, N., Ijaz, R., & Zhong, L. (2020). Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (pp. 1–19).: USENIX Association.

Boos, K. & Zhong, L. (2017). Theseus: a state spill-free operating system. In *Proceedings of the ACM Workshop on Programming Languages and Operating Systems (PLOS '17)*: ACM.

Colaboradores de Wikipedia (2025). MINIX. Wikipedia, la enciclopedia libre. Página modificada por última vez el 29 de septiembre de 2025. Accedido el 23 de octubre de 2025.

Cox, R. and Kaashoek, F. and Morris, R. (2018). *xv6: a simple, Unix-like teaching operating system*. MIT Press. Accedido: 21 septiembre 2025.

Děcký, M. (2010). A road to a formally verified general-purpose operating system. In *Proceedings of the International Symposium on Architecting Critical Systems (ISARCS '10), Lecture Notes in Computer Science, vol. 6150*. Recuperado el 22 Octubre 2025.

Děcký, M. (2015). *Application of Software Components in Operating System Design*. PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics.

FreeDOS Project (2025a). Developers – The FreeDOS Project. Accedido: 2 noviembre 2025.

FreeDOS Project (2025b). FreeDOS – Sistema operativo libre compatible con DOS. Accedido: 2 noviembre 2025.

Gonzales Calos Alejandro, Lozano Eliseo de Jesús (2017). Administración e instala sistemas operativos Haiku. Presentación en Scribd. Subido el 21 de septiembre de 2017. Accedido el 31 de octubre de 2025.

Haiku, Inc. (2025). Haiku. Sitio web oficial. Copyright 2001-2025. Accedido el 23 de octubre de 2025.

Haiku Project (2025). Haiku — The Open Source Operating System. Sitio web oficial. Accedido: 20 septiembre 2025.

Jermář, J. (2025). Helenos — source code (github repository). https://github.com/HelenOS/helenos. Repositorio accedido el 22 Octubre 2025.

JIM HALL (2018). *Advanced FreeDOS*. creative commons. Accedido: 01 noviembre 2025.

jim hall (2018). *using FreeDOS*. Shareen Mann. Accedido: 01 noviembre 2025.

Kuenzer, Simon and Lefeuvre, Hugo and Huici, Felipe and others (2023). FlexOS: Making OS Isolation Flexible. In *Proceedings of the 18th EuroSys Conference*: ACM. Accedido: 26 octubre 2025.

MIT PDOS (2024). xv6 source and text. Accedido: 21 septiembre 2025.

Nakrani, Jatinkumar (2025). Deployment of Android, Haiku, and Google Fuchsia in Containers and Virtual Machines with Web-UI Access via Apache Guacamole. Tesis de Maestría, Frankfurt University of Applied Sciences. Accedido el 31 de octubre de 2025.

Nannan He, Han-Way Huang (2020). USE OF FreeRTOS IN TEACHING A REAL-TIME EMBEDDED SYSTEMS DESIGN COURSE. accessed: 22-septiembre-2025.

NicePNG - Image uploader (2018). Architectural Diagram - MS-DOS Architecture Diagram. Accedido: 02 noviembre 2025.

OSDev Wiki (2025a). Memory Management – OSDev Wiki. Accedido: 25 octubre 2025.

OSDev Wiki (2025b). Scheduling Algorithms – OSDev Wiki. Accedido: 26 octubre 2025.

Pat Villani (2017). *FreeDOS Kernel*. CRC Press. Accedido: 01 noviembre 2025.

Pekopeko11 (2024). Xv6 File System and Log Design. accessed: 21-septiembre-2025.

Project, T. H. (2025). Haiku — source code (github repository). https://github.com/haiku/haiku. Repositorio accedido el 22 de octubre de 2025.

Redox OS Project (2025a). Communication – The Redox Operating System Book. Accedido: 26 octubre 2025.

Redox OS Project (2025b). Components of Redox – The Redox Operating System Book. Accedido: 26 octubre 2025.

Redox OS Project (2025c). Documentación oficial de Redox OS. Accedido: 22 octubre 2025.

Redox OS Project (2025d). Graphics and Windowing – The Redox Operating System Book. Accedido: 26 octubre 2025.

Redox OS Project (2025e). Programs and Libraries – The Redox Operating System Book. Accedido: 26 octubre 2025.

Redox OS Project (2025f). Repositorio de Drivers de Redox OS en GitLab. Accedido: 26 octubre 2025.

Redox OS Project (2025g). Repositorio de Redox OS en GitHub. Accedido: 22 octubre 2025.

Redox OS Project (2025h). Security – The Redox Operating System Book. Accedido: 26 octubre 2025.

Redox OS Project (2025i). The Build Process – The Redox Operating System Book. Accedido: 26 octubre 2025.

Redox OS Project (2025j). User Space – The Redox Operating System Book. Accedido: 26 octubre 2025.

Richard Barry (2018). Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide. accessed: 22-septiembre-2025.

Ruslan Nikolaev and Mincheol Sung and Binoy Ravindran (2021). LibrettOS: A Dynamically Adaptable Multiserver-Library OS. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC '21)*. Accedido el 23 de octubre de 2025.

Ryan Leavengood (2012). The Dawn of Haiku OS. *IEEE Spectrum*. Accedido el 23 de octubre de 2025.

Stimac, Miroslav (2011). The desktop operating system Haiku®: Analysis of the operating system with focuses on ease of use, GUI, multimedia capability and an empirical research of the Haiku community. Tesis de Maestría, FernUniversität in Hagen (University of Hagen). Accedido el 1 de noviembre de 2025.

Systems Software Research Group (2021). LibrettOS: A research operating system. Sitio web oficial del proyecto, Virginia Tech. Copyright 2021. Accedido el 23 de octubre de 2025.

The MINIX 3 Wiki (2017). MINIX 3 Features. MINIX 3 Official Wiki. Página modificada por última vez el 27 de junio de 2017. Accedido el 23 de octubre de 2025.

The rumpkernel project (2025). rumprun: The rumprun unikernel and toolstack. Repositorio oficial en GitHub. Accedido el 23 de octubre de 2025.

Theseus OS Project (2023). The theseus os book. Recuperado de https://www.theseus-os.com/Theseus/book/index.html.

Wikipedia contributors (2024a). FreeRTOS. accessed: 22-septiembre-2025.

Wikipedia contributors (2024b). Xv6. accessed: 21-septiembre-2025.

Yusuf, Mohamad and Al Fatah, M Reza and Fami, Asrul and Pratama, Vemas Adi and Z, M Fauzan and Syadam, Muhammad (2024). Installing the Haiku Operating System on Virtual Box and Implementing File Management. *Journal Collabits*, 1(2). Accedido el 31 de octubre de 2025.