TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

# DESARROLLO DE UN SISTEMA OPERATIVO CON ARQUITECTURA MICROKERNEL


**Autor**
José Luis Amador Moreno

**Directores**
José Luis Garrido Bullejos
Carlos Rodríguez Domínguez

Departamento de Lenguajes y Sistemas Informáticos
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación


—

Granada, julio de 2022

# Desarrollo de un sistema operativo con arquitectura microkernel

José Luis Amador Moreno

**Palabras clave**: sistema_operativo, microkernel, RPC

## Resumen

Aquí va el resumen

# Development of an operating system with microkernel architecture

José Luis Amador Moreno

**Keywords**: operating_system, microkernel, RPC

**Abstract**

Aquí va el resumen

Yo, **José Luis Amador Moreno**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 23834645K, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: José Luis Amador Moreno

Granada a TODO de junio de 2022.

D. **José Luis Garrido Bullejos**, Profesor del Área de TODO del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada.

D. **Carlos Rodríguez Domínguez**, Profesor del Área de TODO del Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada.

**Informan:**

Que el presente trabajo, titulado ***Desarrollo de un sistema operativo con arquitectura microkernel***, ha sido realizado bajo su supervisión por **José Luis Amador Moreno**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a X de mes de 2022.

**Los directores:**

**José Luis Garrido Bullejos** **Carlos Rodríguez Domínguez**

*A mi padre, que en paz descanse.*

# Contenidos

# Términos

**Access Control List** Semántica de permisos en la que se desmenuzan las políticas de acceso sobre un archivo, en lugar de tener un solo propietario.

**Address Space Layout Randomization** Mecanismo por el cual un programa es cargado en distintas partes aleatorias de memoria virtual para fortalecer la seguridad.

**Advanced Configuration and Power Interface** Estándar que aporta tablas con información de la BIOS.

**Advanced Host Controller Interface** Estándar de comunicación con dispositivos SATA.

**Advanced PIC** Chip encargado de manejar las interrupciones en un x86 multiprocesador.

**Advanced Technology Attachment** Protocolo de comandos y transporte para discos duros.

**AHCI Base Memory Register** Base en memoria de un HBA.

**Allocator** Función de reserva de memoria con granularidad de pocos bytes.

**amd64** Otra forma de llamar a x86-64.

**Anillo de protección** Sistema de x86 por el cual existen 4 niveles de privilegios en el procesador.

**Application Processor** Cualquier core distinto al BSP.

**ARM** Arquitectura RISC usada mayoritariamente en dispositivos empotrados.

**ARM64** ISA de ARM de 64 bits.

**ATA Packet Interface** Protocolo de comandos y transporte para unidades CD.

**Basic Input/Output System** Firmware dedicado a inicializar el hardware básico del x86.

**Binario** Ejecutable.

**Bootloader** Programa encargado de preparar la CPU y pasar el control al kernel.

**Bootstrap Processor** Primer core de un sistema SMP iniciado por la BIOS.

**Caché (CPU)** Memoria de rápido acceso entre la CPU y la memoria principal.

**Completely Fair Scheduler** Scheduler usado por Linux basado en un árbol Rojo-Negro.

**Copy on Write** Trampa de page fault para copiar páginas solo cuando se intenta escribir sobre ellas.

**Darwin** Versión libre de XNU.

**Dirección canónica** En x86-64, expansión de signo para completar los 64 bits de direccionamiento cuando la arquitectura soporta menos (48 o 57).

**Dispatcher** Rutina encargada de pasar del kernel a un proceso.

**Double Fault** Excepción causada en un ISR.

**Driver** Programa abstracción sobre hardware.

**Ejecutable** Archivo con formato estándar (por ejemplo, ELF) que contiene el programa resultante del proceso de enlazado.

**Enhanced Host Controller Interface** Interfaz de comunicación con dispositivos USB 2.0.

**Enhanced rep movsb** Mecanismo para realizar copias de memoria aceleradas.

**Enlazado** Unión de los objetos con las librerías.

**Enlazado dinámico** Modo de enlazado en el cual las referencias a librerías se resuelven en tiempo de ejecución.

**Enlazado estático** Modo de enlazado en el cual las librerías se incluyen dentro del binario.

**Entry point** Punto de entrada.

**Esquema de particiones** Forma de organizar particiones sobre un medio de almacenamiento.

**Excepción** Interrupción de naturaleza crítica emitida por la CPU en caso de fallo.

**Executable and Linkable Format** Formato de ejecutable usado por UNIX desde su versión 4.

**Extended Feature Enable Register** MSR para habilitar funcionalidades modernas de x86.

**eXtensible Host Controller Interface** Interfaz de comunicación con dispositivos USB 1.x, 2.0 y 3.x.

**Filosofía UNIX** *Hacer solo una cosa, y hacerla bien.*

**Frame Information Structure** Estructura de AHCI para el envío de órdenes de copia al HBA.

**General Protection Fault** Excepción de x86 de múltiples causas originadas por sistemas de protección.

**Global Descriptor Table** Estructura de IA-32 usada para definir los segmentos a nivel global de la CPU.

**Global Offset Table** Sección del ejecutable.

**Halt** Estado de la CPU en el que no se ejecutan instrucciones, solo se reciben interrupciones.

**Hard real time** Sistema en tiempo real en el que los plazos son inamovibles.

**Heap (tarea)** Región de memoria reservada para un programa para variables dinámicas.

**Herramienta** Utilidad.

**Higher half** Concepto usado para hacer referencia a la mitad superior de la memoria virtual.

**Host Bus Adapter** Controlador AHCI.

**IA-32** ISA de x86 de 32 bits desde el i386, sucesor de x86-16.

**Init** Tarea que se encarga de inicializar el sistema ejecutar el resto de tareas básicas.

**Input/Output APIC** Chip compartido por todos los cores que redirige interrupciones.

**Instruction Set Architecture** Conjunto de instrucciones de una arquitectura.

**Integrated Drive Electronics** Estándar de interfaces para la conexión de dispositivos de almacenamiento.

**Inter-Process Communication** Abanico de métodos para comunicar procesos.

**Inter-processor Interrupt** Mecanismo de la APIC para sincronizar cores por medio de interrupciones.

**Interrupción** Concepto genérico para agrupar las interrupciones hardware y las syscalls.

**Interrupción enmascarable** Interrupción de naturaleza no crítica, deshabilitable.

**Interrupción hardware** Mensaje enviado por el hardware a la CPU.

**Interrupción no enmascarable** Excepción.

**Interrupt Descriptor Table** Estructura de IA-32 y x86-64 usada para manejar las interrupciones.

**Interrupt Request** Petición de interrupción hardware.

**Interrupt Service Routine** Rutina del kernel encargada de resolver una interrupción concreta.

**Interrupt Stack Table** Estructura de IA-32 y x86-64 con pilas libres en caso de interrupción.

**ISO9660** Sistema de archivos usado por los CDs.

**Kernel** Núcleo de un sistema operativo.

**Kernel monolitíco** Kernel que contiene todos los drivers.

**Kernel page-table isolation** Mecanismo del kernel para tener su propio tabla de páginas independiente a los procesos, nunca mapeada en userspace.

**Kernel panic** Fallo irrecuperable del kernel.

**Librería** Conjunto de símbolos externos al programa.

**Librería estándar** Librería con estructuras de datos básicas, usada para comunicación con el kernel.

**Limine** Bootloader independiente que implementa, entre otros, el protocolo de arranque stivale2.

**Llamada al sistema** Syscall.

**Local APIC** Chip adjunto a un core que aporta funcionalidad APIC.

**Local Descriptor Table** Estructura de IA-32 usada para definir segmentos a nivel de tarea.

**Logical Block Addressing** Forma de identificar una dirección en memoria secundaria mediante su número de sector.

**Long IPC** Paso de mensajes grandes, de más de una página.

**Long mode** Modo de un x86 para x86-64.

**M.2** Conector para tarjetas de expansión basado en PCI Express 3.0.

**Mandatory Access Control** Control de acceso genérico en el que se restringen las acciones que puede realizar un programa o usuario.

**Marshalling** Cambio de estructuración de un conjunto de datos.

**Master Boot Record** Primer sector de un dispositivo de almacenamiento, usado por BIOS.

**Memoria compartida** IPC, páginas físicas compartidas entre varias tareas.

**Memoria física** Espacio de direccionamiento cuyas direcciones son emitidas por el bus de datos.

**Memoria virtual** Espacio de direccionamiento virtual, sus direcciones son traducidas a físicas.

**Memory Management Unit** Chip dedicado a traducir direcciones virtuales a físicas.

**Memory-Mapped Input/Output** Mecanismo de comunicación con el hardware por medio de memoria física.

**Message Signaled Interrupts** Modo de configurar las interrupciones en dispositivos PCI.

**Microkernel** Kernel con muy pocos drivers.

**Model Specific Register** Registro de x86 accesible por instrucciones específicas.

**Modo supervisor** Modo de la CPU con acceso a todas las instrucciones.

**Modo usuario** Modo de la CPU con acceso restringido a instrucciones.

**MSI-X** Similar a MSI, pero permite hasta 2048 interrupciones.

**Multi-Level Feedback Queue** MLRR en el que las tareas cambian de prioridad dinámicamente.

**Multi-Level Round-Robin** Scheduler Round-Robin con distintas colas de prioridad.

**Multi-Queue Multiprocessor Scheduler** Rutina encargada de organizar las tareas entre los distintos cores.

**Multi-Threading** Soporte de una CPU para ejecutar varios threads en paralelo.

**NVM Express** Interfaz de comunicación con dispositivos de almacenamiento por PCI Express, alternativa a SATA.

**Open Host Controller Interface** Interfaz de comunicación con algunos dispositivos USB.

**Page Fault** Excepción por acceso indebido a una dirección virtual.

**Paginación** Mecanismo presente en muchas arquitecturas para división de la memoria.

**Paginación multinivel** Esquema de paginación con varias estructuras en forma de árbol.

**Partición** División del espacio de almacenamiento secundario.

**Partición swap** Partición encargada a guardar páginas de procesos cuando no caben en memoria principal.

**Paso de mensajes** IPC, pequeñas secuencias de bits con formato.

**PCI Express** Versión moderna, y retrocompatible, de PCI.

**Peripheral Component Interconnect** Bus estándar de computadores para conectar periféricos a la placa base.

**Physical Address Extension** Mecanismo de x86 desde IA-32 para paginar a tres niveles.

**Pila (tarea)** Región de memoria reservada para un programa para variables locales y direcciones de retorno.

**Pila de almacenamiento** Pila de drivers en cuya cima se encuentra VFS.

**Pila de red** Pila de drivers para dar soporte a red (Ethernet, IP, TPC...).

**Pipe** Unix FIFO.

**Polling** Mecanismo de espera reiterada hasta que un suceso ocurre.

**Port-mapped I/O** Mecanismo de comunicación con el hardware por medio de instrucciones específicas.

**Portable Operating System Interface** Estándar IEEE que define una interfaz y entorno de SO.

**Position-independent code** Código compilado para utilizar exclusivamente referencias relativas al contador de programa, nunca absolutas.

**Primary Volume Descriptor** Superbloque de ISO9660.

**Proceso** Binario cargado en memoria, denominación práctica de tarea.

**Process Control Block** Estructura de datos que representa a un proceso.

**Process Identifier** Identificador numérico de un proceso.

**Program Header** Estructura de un ELF que define regiones de memoria.

**Programmable Interrupt Controller** Chip encargado de manejar las interrupciones en un IBM compatible.

**Programmed I/O** Mecanismo de transferencia de bytes de dispositivos de almacenamiento en el que interviene la CPU.

**Protected mode** Modo de un x86 para IA-32.

**Public Service Namespace** Mecanismo de resolución de PIDs por nombre público de Strife.

**Punto de entrada** Dirección de un programa en la cual comienza la ejecución.

**Página** Particiones de la memoria, física o virtual.

**Quantum** Tiempo máximo consecutivo permitido para la ejecución de una tarea.

**Real mode** Modo de retrocompatibilidad de un x86 con x86-16.

**Real time** Concepto utilizado cuando la ejecución de una tarea tiene un plazo.

**Reentrancia** Toma del poder del kernel tras un quantum.

**Relocation Read-Only** Medida de seguridad por la cual se resuelven las referencias en la GOT de un proceso antes de tiempo para marcarla como solo lectura.

**Remote Procedure Call** IPC, llamada de un proceso a una función de otro.

**Remote Procedure Identifier** En Strife, identificador para una procedimiento remoto, relativo al servidor.

**Returned-Oriented Programming** Técnica de explotación de binarios que permite ejecutar código arbitrario mediante la escritura de direcciones de salto.

**Ring 0** x86 en modo supervisor, en referencia al anillo de protección con identificador 0.

**Ring 3** x86 en modo usuario, en referencia al anillo de protección con identificador 3.

**Root System Description Pointer** Tabla ACPI con puntero a RSDT o XSDT.

**Round-Robin** Algoritmo genérico que representa una cola cíclica.

**Sandwich MLFQ** Scheduler de autoría propia en este trabajo.

**Scheduler** Rutina encargada de seleccionar la siguiente tarea a ejecutarse.

**Scheduler a corto plazo** Definición anterior de scheduler.

**Scheduler a largo plazo** Rutina encargada de seleccionar la siguiente tarea a cargarse en memoria, no ejecutarse.

**Scheduler a medio plazo** Rutina encargada de manejar la entrada y salida de páginas de memoria secundaria.

**Sector** Partición física del espacio de almacenamiento de un dispositivo.

**Segmentación** Mecanismo de IA-32 para división de la memoria, obsoleto.

**Segmento** Divisón de memoria por medio de segmentación.

**Selector** Identificador de segmento que contiene su índice además de ciertas flags.

**Serial ATA** Versión moderna de ATA, retrocompatible.

**Shared Memory Identifier** En Strife, identificador para una página compartida, relativo al cliente.

**Shell** Interfaz básica de órdenes con la que el usuario se comunica con el sistema.

**Single-Queue Multiprocessor Scheduler** Sistema multiprocesador sin MQMS.

**Small Computer Systems Interface** Interfaz de comunicación para transferencia de datos.

**Soft real time** Sistema en tiempo real en el que los plazos no son inamovibles.

**Spinlock** Cerrojo de espera ocupada usado en los kernels, que puede ser implementado con distintos algoritmos.

**Standard Template Library** Parte de la librería estándar que implementa estructuras de datos abstractas.

**Startup IPI** IPI que arranca los APs en modo real con un punto de entrada concreto.

**stivale2** Protocolo de arranque simple y libre, implementado en varios bootloaders independientes.

**Supervisor Memory Access Protection** Mecanismo hardware para prohibir el acceso a memoria de usuario desde ring 0 a menos que se indique lo contrario.

**Supervisor Memory Execute Protection** Mecanismo hardware para prohibir la ejecución de páginas de usuario desde ring 0.

**Symmetric Multiprocessing** Método de multiprocesador en el cual todos los cores acceden a toda la memoria.

**Syscall** Interrupción software generada por una tarea para comunicarse con el kernel.

**System Call Extensions** Bit dentro del EFER que habilita las instrucciones syscall y sysret.

**Tabla de páginas (formal)** Estructura del procesador usada para la traducción de direcciones virtuales a físicas.

**Tarea** Unidad de código y datos.

**Task State Segment** Estructura de IA-32 usada en cambios de contexto hardware.

**Thread Control Block** Sinónimo de PCB en sistemas multi-threading.

**Tiempo real** Real time.

**TLB flush** Proceso por el cual se borran las direcciones de la TLB.

**Translation Lookaside Buffer** Caché de la MMU.

**Triple Fault** Excepción causada en el ISR de #DF, irrecuperable por software.

**Unified Extensible Firmware Interface** Sucesor a BIOS.

**Universal Host Controller Interface** Interfaz de comunicación con dispositivos USB 1.x.

**Universally Unique Identifier** Número de 128 bits que intenta identificar un objeto cualquiera de forma única.

**Unix FIFO** IPC, flujo de bits.

**User Identifier** Representación numérica de un usuario del sistema.

**Userspace** Término práctico de modo usuario.

**Utilidad** Programa para el usuario final que abstrae el sistema operativo.

**Very Soft Real Time** Procesos con prioridad máxima, permitiendo la inanición de otros, pero sin deadlines.

**VESA BIOS Extensions** Interrupción BIOS para manejar los modos de vídeo.

**Virtual File System** Sistema de archivos abstracto, que encapsula al resto.

**x86** Arquitectura CISC diseñada por Intel, usada en la mayoría de ordenadores personales.

**x86-16** ISA original de x86 desde el 8086.

**x86-64** ISA de x86 de 64 bits que usan la mayoría de computadores, sucesor de IA-32.

**XNU** Kernel usado por macOS.

**Yield** Syscall utilizada en sistemas antiguos sin reentrancia para devolver el control al kernel.

# Abreviaciones

**ABAR** AHCI Base Memory Register

**ACL** Access Control List

**ACPI** Advanced Configuration and Power Interface

**AHCI** Advanced Host Controller Interface

**AP** Application Processor

**APIC** Advanced PIC

**ASLR** Address Space Layout Randomization

**ATA** Advanced Technology Attachment

**ATAPI** ATA Packet Interface

**BIOS** Basic Input/Output System

**BSP** Bootstrap Processor

**CFS** Completely Fair Scheduler

**CoW** Copy on Write

**DF** Double Fault

**EFER** Extended Feature Enable Register

**EHCI** Enhanced Host Controller Interface

**ELF** Executable and Linkable Format

**ERMSB** Enhanced rep movsb

**FIS** Frame Information Structure

**GDT** Global Descriptor Table

**GOT** Global Offset Table

**GP** General Protection Fault

**GPF** GP

**HBA** Host Bus Adapter

**IDE** Integrated Drive Electronics

**IDT** Interrupt Descriptor Table

**IOAPIC** Input/Output APIC

**IPC** Inter-Process Communication

**IPI** Inter-processor Interrupt

**IRQ** Interrupt Request

**ISA** Instruction Set Architecture

**ISR** Interrupt Service Routine

**IST** Interrupt Stack Table

**KPTI** Kernel page-table isolation

**LAPIC** Local APIC

**LBA** Logical Block Addressing

**LDT** Local Descriptor Table

**MAC** Mandatory Access Control

**MBR** Master Boot Record

**MLFQ** Multi-Level Feedback Queue

**MLRR** Multi-Level Round-Robin

**MMIO** Memory-Mapped Input/Output

**MMU** Memory Management Unit

**MQMS** Multi-Queue Multiprocessor Scheduler

**MSI** Message Signaled Interrupts

**MSR** Model Specific Register

**MT** Multi-Threading

**NVMe** NVM Express

**NX** No Execute

**OHCI** Open Host Controller Interface

**PAE** Physical Address Extension

**PATA** ATAPI

**PCB** Process Control Block

**PCI** Peripheral Component Interconnect

**PCIe** PCI Express

**PF** Page Fault

**PHDR** Program Header

**PIC (ejecutable)** Position-independent code

**PIC (hardware)** Programmable Interrupt Controller

**PID** Process Identifier

**PIO (disco)** Programmed I/O

**PIO (hardware)** Port-mapped I/O

**POSIX** Portable Operating System Interface

**PSNS** Public Service Namespace

**PVD** Primary Volume Descriptor

**RELRO** Relocation Read-Only

**ROP** Returned-Oriented Programming

**RPC** Remote Procedure Call

**RPID** Remote Procedure Identifier

**RR** Round-Robin

**RSDP** Root System Description Pointer

**SA** Sistema de Archivos

**SATA** Serial ATA

**SCE** System Call Extensions

**SCSI** Small Computer Systems Interface

**SIPI** Startup IPI

**SMAP** Supervisor Memory Access Protection

**SMEP** Supervisor Memory Execute Protection

**SMID** Shared Memory Identifier

**SMLFQ** Sandwich MLFQ

**SMP** Symmetric Multiprocessing

**SO** Sistema operativo

**SQMS** Single-Queue Multiprocessor Scheduler

**stdlib** Librería estándar

**STL** Standard Template Library

**TCB** Thread Control Block

**TLB** Translation Lookaside Buffer

**TSS** Task State Segment

**UEFI** Unified Extensible Firmware Interface

**UHCI** Universal Host Controller Interface

**UID** User Identifier

**UUID** Universally Unique Identifier

**VBE** VESA BIOS Extensions

**VFS** Virtual File System

**VSRT** Very Soft Real Time

**xHCI** eXtensible Host Controller Interface

# Capítulo 1

# Introducción y objetivos

## 1.1. Introducción

### 1.1.1. Dominio del problema

### 1.1.2. Motivación

Mi primer programa lo escribí a los siete años. Fue un script en Batch, el lenguaje que usan los archivos con extensión `.bat` en Windows (herencia de MS-DOS). Con el paso de los años, aprendí muchos otros lenguajes: Visual Basic, Java, PHP, Javascript, Python... Más adelante, con más experiencia, otros como C y C++. Debido a la vaga idea que tenía sobre el funcionamiento interno de un computador, resultaron incontables los intentos fallidos de aprender ensamblador de x86 por mi cuenta antes de entrar en la carrera. En el primer cuatrimestre de primero, allá por octubre de 2018, conseguí al fin entender por mi cuenta el funcionamiento básico de un x86 y, con esto, escribir un *hola mundo* que podía comprender.

El uso de ensamblador, siendo no más que un conjunto de macros y mnemónicos, está muy acotado en la actualidad. Se puede usar para optimizar secciones importantes de código que han de ejecutarse rápido, pero a cambio se pierde portabilidad entre arquitecturas y, dependiendo de cómo de concreta sea la operación a optimizar, también entre generaciones de procesadores. Además, solo es aplicable a lenguajes compilados, y en un mundo en el cual los programas siguen una trayectoria clara de volverse independientes de la máquina (primero con Java, últimamente con Electron), son muy pocas las situaciones en las que resulta útil.

Sin embargo, existe un proyecto, necesario aún hoy en día, que únicamente puede ser escrito en ensamblador. Se trata del *stage 1* de un *bootloader*: las primeras instrucciones controlables por software que ejecuta un procesador después de completar la inicialización más básica del hardware. Con tal de aliviar el esfuerzo necesario para escribir este lenguaje, los bootloaders intentan preparar el entorno más simple posible que permita pasar a C o C++, y de aquí nace el *stage 2*. Los núcleos de los sistemas operativos necesitan también secciones de ensamblador para realizar operaciones de bajo nivel, como las rutinas de interrupción o los cambios de contexto.

Todo esto se une a un interés desde pequeño de crear un sistema operativo, aún con la simplificada idea que era capaz de entender por entonces. Un sistema operativo es el mayor proyecto de software, definido por algunos como *la gran frontera* o *el gran pináculo de la programación* [1].

Este es el fruto de una vida de aprendizaje.

### 1.1.3. Estructura de la memoria

Este documento se organiza en un total de seis capítulos:

- El capítulo 1 se enfoca en presentar el problema a resolver, detallar los objetivos a cumplir, y aproximar una planificación de los pasos necesarios para completar el trabajo.
- El capítulo 2 describe, de forma superficial pero autocontenida, todos los fundamentos necesarios que una persona ajena al mundo del desarrollo de sistemas operativos debe conocer para entender la magnitud del problema y la lógica detrás de las soluciones.
- El capítulo 3 analiza el estado del arte de los proyectos próximos a este, para averiguar qué soluciones se han dado a los problemas aquí descritos en el pasado.
- El capítulo 4, con la inspiración del anterior además de varias ideas novedosas, expresa cómo el proyecto orquestra sendos subproyectos que son capaces de resolver los problemas dados.
- El capítulo 5 completa el anterior, comentando cómo cada subproyecto realiza la tarea de resolver el problema que tiene asignado.
- Para terminar, el capítulo 6 se hacen varias cosas. Primero, se muestran capturas de pantalla del resultado del proyecto funcionando. No merece un capítulo propio: la enorme mayoría del trabajo es completamente invisible. Después, se hace una retrospectiva del trabajo realizado, comentando qué decisiones han dado buenos resultados y cuáles no. Por último, se menciona el trabajo futuro que puede seguir el estado del arte en este aspecto, así como el propio para continuar el proyecto después de la entrega.

## 1.2. Objetivos

**TODO: El objetivo principal y otros más técnicos. Esto es un mini-índice. Estado del arte. Validar, posibles mejoras, trabajo futuro. Fallos cometidos. Metodología.**

### 1.2.1. Terminología

Con tal de evitar malentendidos y siglas imposibles de buscar en internet, se utilizarán términos en inglés, especialmente en aquellos que tienen una abreviación asociada. La terminología se utiliza, así, en español e inglés indistintamente. En este último caso, el género de los sustantivos se eligirá de forma arbitraria, pero se mantendrá consistente durante todo el texto.

## 1.3. Planificación y costes

**TODO: Se puede decir la planificación inicial y luego ya la final en otro capítulo. Gantt. Contar horas, estimación. Cuántas por ECTS.**

# Capítulo 2

# Fundamentos

## 2.1. Definición

Alfred Aho, autor del libro más importante sobre compiladores, *Compilers: Principles Techniques and Tools* [2], así como un libro referente sobre algoritmos, *Data Structures and Algorithms* [3], comenzó una conferencia en 2015 con la siguiente afirmación:

> Tal y como decía Knuth en *The Art of Computer Programming*, [un algoritmo] no es más que una serie finita de instrucciones que termina en un tiempo finito. [...] Yo doy clase de teoría de computadores en Columbia, y usamos dos libros de texto: uno usa esta definición; el otro, afirma que un algoritmo no tiene necesariamente que parar para todas las entradas. Así, los computólogos no pueden estar de acuerdo ni en el término más fundamental del área. — Alfred Aho [4]

Si *algoritmo* es una palabra cuya extensión es difícil de delimitar, hacerlo para *sistema operativo* resulta una tarea más complicada aún. Un sistema es todo aquel conjunto de bloques relacionados entre sí con el propósito de emerger un todo. La intuición es sencilla, se explica en primero de carrera en la Universidad de Granada: programa o conjunto de programas que controla la ejecución de aplicaciones y actúa como interfaz entre el usuario y el hardware.

Es evidente para todo entendido que, si bien esta definición es correcta, y muy certera en el uso del concepto de abstracción, no establece límites. Existen definiciones distintas; por ejemplo, Andrew Tanenbaum en *Modern Operating Systems* aporta dos que no son mutuamente excluyentes: sistema operativo como **máquina extendida**, en el sentido de pila de capas de abstracción, y como **gestor de recursos** [5].

Se trata de un debate abierto al cual el habla popular no ayuda: no es inusual escuchar a alguien ajeno al campo referirse a Linux como un sistema operativo, a pesar de ser un núcleo. Por otra parte, la posición de la *Free Software Foundation*, y especialmente la de su antiguo portavoz Richard Stallman, de ridiculizar al proyecto como una minúscula parte del sistema GNU [6], también resulta inadecuada, en especial sabiendo que existen motivos de conflicto de interés entre ambos proyectos.

En el mundo del *hobby osdev*, es decir, el de aquellos programadores que se dedican a escribir sistemas operativos como actividad recreativa, al cual yo he pertenecido durante varios años y dentro del cual he hecho grandes amigos, también existe esta disputa: es común encontrar a expertos en estos grupos que no consideran a DOS como un SO por no ofrecer un kernel con la suficiente abstracción del hardware.

Como definir el término parece ser una batalla perdida, es infructuoso dedicarse a lucharla en un trabajo de esta índole, y se tomará una postura de mente abierta, en la que se aceptarán como partes de un sistema operativo todas las capas de abstracción genéricas por debajo de una utilidad (piense en el bloc de notas), así como aquellos programas que actúen únicamente como vista para interactuar de forma directa con una de las capas, como `ls` en GNU, o `dir` en MS-DOS.

### 2.1.1. Portabilidad

Al contrario de lo que puede parecer, un sistema operativo está escrito con el objetivo de soportar una arquitectura o un conjunto de ellas. De forma general, se intenta escribir el código más portable posible, pero partes críticas como ciertas rutinas del kernel, así como la totalidad del bootloader, son, por pura definición, no portables, y son necesarias versiones distintas para cada arquitectura a soportar (denominadas *target architectures*, o *targets* para abreviar). Existe el ejemplo extremo de NetBSD, cuyo objetivo es soportar la mayor cantidad de arquitecturas posibles (en el momento de redactar esto, 8 primarias y 49 secundarias [7]). En el otro extremo, se encuentra Windows 11, con soporte completo para únicamente x86-64 y ARM64. Los sistemas operativos hechos por un grupo reducido de personas, así como los hechos con un propósito muy concreto, suelen intentar soportar solo una. En este caso, se trata de x86, con lo que los fundamentos prácticos explicados en este documento se enfocarán en dicha arquitectura.

## 2.2. Partes comunes

Habiendo establecido una definición, es posible distinguir cuatro partes fundamentales, que en muchas ocasiones se pueden encontrar mezcladas, y en otras extremas pueden faltar. Son: el *kernel*, los *drivers*, las *librerias*, y las *utilidades*. En esta sección se hará un repaso por su significado, se darán ejemplos, y se enunciarán sus partes de haberlas.

### 2.2.1. El kernel

El kernel de un sistema operativo, traducido como *núcleo*, es el soporte sobre el cual reposa todo el sistema. Es el primer software que se ejecuta fuera del bootloader, y se pueden destacar varios objetivos:

- Manejar los distintos recursos de bajo nivel.
- Hacer emerger el concepto de tareas.
- Interconectar tareas y drivers.

Es importante profundizar sobre cada uno de estos aspectos. Para empezar, el hardware proporciona una serie de recursos esenciales para todo programa: memoria, canales de interconexión, periféricos... De todos ellos, la memoria es el único esencial para tener un sistema (discutiblemente aburrido, pero completo). Los procedimientos de reserva y liberación de memoria son manejados por el kernel. Se profundizará en este tema en la sección 2.4.1.

En todo sistema operativo moderno (especialmente aquellos que pertenecen a la familia de los multiprogramados) existe el concepto de *tarea*: una unidad de código y datos que se comunica con diversas partes del sistema. El kernel es el encargado de montarla en memoria, y, usualmente, intercambiarla con otras en cortos periodos de tiempo para dar la impresión de que se están ejecutando simultáneamente, cuando no necesariamente tiene que ser así. De este concepto surge la mayor parte de teoría escrita sobre sistemas operativos, y se suele considerar la parte más importante. Se profundizará mucho en este apartado durante todo el trabajo, pero en la sección 2.3.1 se encontrarán las primeras pinceladas.

Por último, un kernel conecta tareas entre sí y con los drivers presentes. Las tareas se comunican mediante un concepto llamado IPC (*Inter Process Communication*), de las cuales existen varios tipos no necesariamente excluyentes:

- FIFOs. Son flujos de bits que funcionan como tuberías (*pipe*, en inglés). Es lo que usa UNIX y derivados.
- Paso de mensajes. Consiste en hacer envíos de paquetes, como si fueran sockets, a ciertos puntos de recepción de la tarea que actúa como servidor. Su uso para mensajes largos ha quedado en desuso, pues se conoce que la copia de grandes mensajes ralentiza mucho el sistema.

- RPC, *Remote Procedure Call*. En este tipo de IPC, una tarea llama a una función de otra (que puede encontrarse en un otro computador) como si se tratara de una suya propia. Es lo que usa el sistema operativo de este proyecto, así como partes internas de NT bajo el nombre de LPM (*Local Procedure Call*). A diferencia del resto, este es un procedimiento síncrono: la tarea A entra dentro de B, con lo cual la ejecución de A no continúa hasta que la rutina de B haya terminado.
- Memoria compartida. Presente en la gran mayoría de sistemas operativos modernos, el concepto de compartir memoria física es esencial para ocasiones en las que hay que transmitir una gran cantidad de datos entre tareas con mínima latencia.

Un kernel (o, al menos, parte de él) siempre se ejecuta en lo que se conoce de forma genérica como modo supervisor, siendo su contraparte el modo usuario. El supervisor tiene acceso a la totalidad de la CPU: todos los registros, todas las instrucciones, y toda la memoria. En el modo usuario, se restrigen (muchas veces granularmente) estas capacidades.

Gran parte del trabajo del kernel es recibir peticiones. Algunas de ellas son generadas por el hardware, y se denominan interrupciones hardware. De estas hay dos tipos: enmascarables, relativas al hardware no esencial, y no enmascarables, de las cuales el mayor exponente son las excepciones (la más simple: la división por cero). Otras son causadas por el software, y se las conoce como llamadas al sistema (en inglés, *system calls*, o *syscalls* para abreviar).

Existen dos tipos de kernels fundamentales: monolíticos y microkernels.

Los monolíticos se caracterizan por tener todos los drivers dentro. Esto hace que la comunicación entre ellos sea rápida durante la ejecución, aunque, de haber un cambio en uno de ellos, será necesario enlazar de nuevo todo el kernel. Salvo muy finas protecciones que los kernel monolíticos suelen crear al arranque, un fallo de programación, por poco grave que sea, puede promocionar a un fallo irrecuperable del kernel (concepto conocido como *kernel panic*) [8]. Además, un driver malicioso podría tomar control del kernel y, por tanto, de todo el sistema, haciéndose a sí mismo invisible en el proceso; este tipo de malware se conoce como *rootkit*.

Los microkernels se caracterizan por lo opuesto: intentan separar los drivers en tareas independientes siempre que sea posible. La comunicación entre ellos es considerablemente más lenta, pues una petición a un driver requiere cambiar de una tarea a otra, en un proceso llamado cambio de contexto, muy costoso en recursos. A cambio, un fallo en uno de los drivers no tiene por qué resultar terminal, y la tarea correspondiente puede reiniciarse con la esperanza de que siga funcionando sin que vuelva a ocurrir ese comportamiento anómalo. Esto les aporta más robustez, así como seguridad: un driver nunca se ejecutará en modo supervisor, aunque sí puede tener acceso al hardware y causar problemas por ese camino. Los microkernels son conceptualmente más simples, pues mantienen el software separado en proyecto sencillos sin crear un delicioso plato de código spaguetti en el que un driver llama a otro localmente y sin posible detección, registro, y control de privilegios: todos están al mismo nivel. A cambio, son mucho más complejos de escribir, pues parten de un entorno en el que no hay funcionalidad, y han montar todo un sistema en base a eso (*¿Cómo cargar el programa que carga los programas?*) [9]. Este proceso de hacer emerger un sistema de la nada se denomina *bootstrapping*.

Con el objetivo de hacer el plato italiano menos apetitoso y alcanzar un equilibrio entre

separación de drivers y velocidad, surgen los kernels híbridos. La mayoría de kernels comerciales los utilizan, entre ellos NT (Windows), Linux, y XNU/Darwin (macOS). Los drivers separados del kernel se denominan módulos, y se cargan en tiempo de ejecución desde el sistema de archivos: o bien como una tarea como los microkernels, o bien introduciéndolos en el contexto del kernel. Por esto mismo, solo los drivers que no resultan esenciales para el funcionamiento del sistema pueden cargarse modularmente. Nótese que esta decisión se centra en aliviar el tamaño del código fuente, así como del binario final, del kernel, y no está guiada por la seguridad.

### 2.2.2. Los drivers

Un driver (en español, *controlador*) es un programa que implementa una capa de abstracción sobre un dispositivo o concepto de bajo nivel [10]. En un kernel monolítico, no es más que una colección de funciones y estructuras. En un microkernel, se ejecuta como una tarea independiente.

Existe un driver por dispositivo físico al que se quiere conectar, así como otros que agrupan otros drivers y crean abstracciones virtuales. Por ejemplo, un driver de IDE, correspondiente a los distintos dispositivos ATA conectados a la placa base (discos duros clásicos), puede ser accedido mediante otro driver que agrupe los dispositivos físicos y les dé nombres virtuales, como `sda1` en el caso de Linux.

Sin drivers difícilmente puede haber un sistema operativo. Se suele considerar que el driver de vídeo, encargado de mostrar texto o imágenes por la pantalla, es esencial para un sistema operativo útil. Dependiendo del enfoque y el objetivo del proyecto, puede contar con unos y no otros. Si el SO está principalmente enfocado para servidores, puede no contar con un driver de teclado, y en su lugar tener una pila de red (*network stack*) amplia que permita a otros dispositivos comunicarse con el sistema. Si está enfocado a ser usado por usuarios ajenos al área, un driver de vídeo que pueda mostrar gráficos es imprescindible.

A la hora de escribir un driver, se recurre a la especificación del hardware. En ocasiones, esta especificación no es pública y se mantiene como secreto corporativo. En estos casos, es el fabricante el que se encarga de escribir el controlador para un sistema operativo concreto, generalmente Windows. A veces, el fabricante no publica la especificación, pero sí el código fuente del driver, y generalmente el código resulta ilegible, pues su propósito no es ser comprendido. Como gran exponente de esto último cabría destacar el archivo `intel_display.c` de Linux, escrito, naturalmente, por Intel, y que implementa parte del driver en un solo archivo de más de 10,000 líneas [11].

Por esto último, hay grupos de dispositivos cuyo soporte resulta inalcanzable para un desarrollador de sistemas operativos independiente sin llegar a métodos como la ingeniería inversa. Ejemplos de esto son *Wifi* y la aceleración gráfica 3D.

### 2.2.3. Las librerías

Una librería (*library* en inglés, en ocasiones también traducido como *biblioteca*) es una API que proporciona una abstracción sobre un concepto; por ejemplo, permite a un programa la comunicación con otra parte del sistema de forma sencilla. Pueden estar

enfocadas en envolver el funcionamiento de un driver, creando funciones que se comunican con él para hacer el proceso más transparente al programador. También pueden estar escritas con un propósito de más alto nivel, como realizar operaciones matemáticas sobre enteros de múltiple precisión.

Cuando un sistema operativo planea soportar los ejecutables producidos por un lenguaje, construye para él una librería de comunicación con el kernel y el resto del sistema: se denomina la librería estándar (*stdlib*). El ejemplo más claro es C, para el que GNU aporta la `GNU libC` [12], y Windows la API del sistema.

Las librerías se juntan con los archivos objeto en el proceso de enlazado. Este proceso se puede realizar de dos maneras: estático y dinámico.

- En el estático, las librerías se adjuntan en el ejecutable. Esto hace que el binario (ejecutable) resulte independiente del entorno, pues lleva con él todo lo que necesita.
- En el dinámico, las librerías se referencian por su nombre y uso, y es el cargador de programas, en ejecución, quien se encarga de resolver las direcciones mediante un proceso denominado *relocation*. Esto reduce el tamaño del binario, y permite una actualización global de una librería sin reenlazar todos los programas.

### 2.2.4. Las utilidades

Una herramienta (*tool*) o utilidad (*utility*) es todo programa con una función simple que se relaciona con el kernel. Permiten una vista sobre algún aspecto del sistema, y generalmente lo hacen de forma legible para humanos (*human-readable*). Son programas a los que en la mayoría de ocasiones se accede mediante la *shell* (concha), cuyo nombre, originario de UNIX, referencia a cómo oculta en su interior una perla (el kernel). Las utilidades también se pueden combinar con otras en *scripts*, creando complejos procesos encadenados. UNIX inventó el concepto de *pipes*, mediante los cuales la salida de un programa es conectada a la entrada de otro, permitiendo así una armonía de interconexión entre utilidades [13].

Con los años, especialmente en la comunidad Linux, este concepto ha ido en decadencia, y son pocas las utilidades que permiten este tipo de interconexión sin hacer ningún retoque.

Además, aquí aparece la filosofía UNIX: *hacer solo una cosa, y hacerla bien*, refiriéndose a que las utilidades deben mantenerse simples, y en lugar de tener una herramienta para varios propósitos, tener muchas herramientas para cada acción. En el entorno Linux, y especialmente en las utilidades GNU, este concepto nunca ha existido. El código fuente de `ls` es un archivo de cinco mil líneas [14].

Nótese que existen *comandos* que se comportan como utilidades a pesar de no serlo. En su lugar, son órdenes a la shell que se gestionan internamente sin pasar por ejecutar un programa. Ejemplos son `cd` o `echo`.

## 2.3. Teoría de un sistema operativo

Conociendo las partes más esenciales de un sistema operativo, existen ciertas áreas que resultan de interés teórico. Generalmente, son las relativas a las tareas, y son los conceptos que en la Universidad de Granada se impartieron en la asignatura *Sistemas Operativos*. En esta sección se hará un breve repaso de todas estas áreas, con tal de contextualizar el resto del trabajo: las tareas, el scheduler, y el sistema de archivos.

### 2.3.1. Tareas

En la subsección 2.2.1 se explicó superficialmente el concepto de tarea, y este capítulo trata de profundizar en él. Lo más fundamental: *tarea* es el nombre teórico del concepto. Generalmente, se utiliza el término *proceso* para referirse a un binario cuando está cargado en memoria. En sistemas MT (*Multi-threading*), la terminología es *thread* (traducido como *hilo* o *hebra*), de las cuales pueden estar ejecutándose varias que comparten gran parte del contexto concurrentemente.

La forma de representación interna de una tarea en el kernel es el PCB (*Process Control Block*), también llamado TCB (*Thread Control Block*) en sistemas multithreading, una estructura que contiene todo lo necesario para su funcionamiento, incluyendo su contexto y sus regiones de memoria estáticas (cargadas del binario) o las dinámicas como la pila y el *heap*. Las tareas son referenciadas por su PID (*Process Identifier*), un entero sin signo generalmente de 16, 32, o 64 bits [15].

Toda tarea se crea y se ejecuta, la gran mayoría terminan, no se ejecutan indefinidamente y, en los sistemas operativos modernos, además se pausan y se reanudan. El proceso de reanudar una tarea o ejecutarla por primera vez se lleva a cabo por una rutina llamada el *dispatcher*. Esta se encarga de realizar el cambio de contexto, es decir, recuperar el estado del procesador (registros y flags, generalmente) en el que se encontraba la tarea (o el inicial de ser arrancada), así como su tabla de páginas. Después, realiza un cambio a modo usuario y salta al punto donde se pausó la tarea, de haber sido pausada, o el punto de entrada (*entry point*) de ser iniciada.

En UNIX, la primera tarea que se ejecuta es *init*, con PID 1 [13]. En Linux, concretamente, existen varios programas a elegir, siendo el más usado *systemd* [16], y en menor medida otros como *OpenRC* [17], *runit* [18], o *SysV init* [19]. Esta tarea inicia todas las otras, y desde entonces toda tarea tiene un padre, lo cual genera un grafo de hijos trazable. El proceso de creación de una tarea en UNIX se realiza mediante un procedimiento de `fork`, por el cual la tarea hace mitosis y forma dos partes completamente independientes (no threads), seguida de `exec`, por el cual sustituyen todas sus estructuras del PCB por las del binario cargado como parámetro [13]. En Windows, este procedimiento es atómico, y se realiza mediante una llamada a la API a la función `CreateProcess` `createprocess`.

Una tarea generalmente se encuentra en uno de tres estados: preparada para ser ejecutada, bloqueada esperando algún recurso, y ejecutándose. Además, necesariamente todo programa en ejecución cuenta con cuatro secciones: datos, código, pila (*stack*), y *heap*.

### 2.3.2. Scheduler

En un estado usual del sistema hay decenas o cientos de tareas pendientes de ejecutarse. Debe haber, así, una autoridad que decida quién se ejecuta, dónde, y durante cuánto tiempo. De esto se encarga el *scheduler* (traducido como *planificador*): es la rutina del kernel encargada de manejar las tareas en tiempo de ejecución.

En la literatura clásica se definen tres tipos [20]:

- Scheduler a largo plazo (*long-term*). Es el encargado de decidir qué procesos se admiten en memoria principal, esto es, cuando se cargan y ejecutan por primera vez.
- Scheduler a medio plazo (*medium-term*). Decide cuándo los procesos entran y salen de memoria principal para situarse en memoria secundaria (disco duro).
- Scheduler a corto plazo (*short-term*). Decide qué tarea es la siguiente que ha de recibir tiempo de CPU, en base a ciertos criterios.

Con el tiempo, los dos primeros tipos han quedado, o bien en desuso, o bien son muy raramente utilizados. El primer tipo, en la práctica, es raramente referenciado así. Generalmente, gracias a la creación de los procesadores multinúcleo, el kernel carga una tarea de forma inmediata, aunque no necesariamente se ejecute en ese instante.

Cuando la cantidad de memoria RAM estaba en el orden de los MBs o pocos GBs, tenía sentido el scheduler a medio plazo. Existían particiones *swap* (de intercambio), sobre las cuales los procesos entraban y salían por no caber en memoria principal. Cualquier estudiante de ingeniería informática que haya ejecutado un algoritmo pesado y ha estado viendo a la vez la salida de `htop` es consciente de que si se empieza a usar la memoria de intercambio es porque hay un *memory leak* en su código, y no por la pesadez del algoritmo. En otras palabras, si el proceso ha llegado a usar swap, la va a llenar pronto y el kernel lo va a terminar: ¿Para qué usar swap siquiera entonces?

En algunos casos de cómputos extremos para aplicaciones de, por ejemplo, astronomía, es posible que se llegue a usar swap, pero generalmente, por ser tan lenta, suele merecer la pena instalar más memoria principal. Los supercomputadores no son famosos por la cantidad de espacio de almacenamiento que tienen, sino por la velocidad de sus procesadores, GFLOPs, y la amplia RAM. Las particiones swap siguen existiendo, los instaladores de Linux las crean por defecto a día de hoy, pero los sitemas operativos soportan esta función muy principalmente porque *ya estaba ahí*, y tendría poco sentido eliminarla siendo algo que siempre va a estar inactivo, y cuyo *overhead* dentro del kernel es inexistente.

Por todo esto, cuando hoy en día se habla de scheduler, siempre se hace referencia a dos tipos: al scheduler a corto plazo, y a un nuevo tipo que ha surgido con la llegada de los multinúcleo, el MQMS (*Multi-Queue Multiprocessor Scheduler*).

El MQMS es el más amplio, y por lo tanto el que debe explicarse primero. Toda CPU moderna tiene, en mayor o menor medida, caché. La caché L1 es la que está individualizada a los núcleos. Así, tendría sentido repartir las tareas entre los *cores* de tal forma que se maximize el uso de caché, e idealmente quepan todos los programas que han de ejecutarse ahí, lo que conllevaría una velocidad mucho mayor en la ejecución de tareas, pues la copia

de bits de RAM a caché es mucho más lenta que de caché a registros. Varios sistemas operativos, especialmente los indicados para servidores (como Linux) tienen este tipo de scheduler, pero no todos: también se puede mantener una *pool* global de procesos de la que cada core saca uno cuando le toque (SQMS). Implementar un MQMS es complicado, y de hacerse mal puede ser contraproducente: alcanzar un equilibrio siempre es difícil.

El scheduler a corto plazo (a partir de ahora, simplemente scheduler), decide qué se ejecuta y en qué orden. Se pueden clasificar según muchos criterios:

- Con o sin reentrancia (*preemption*). En los schedulers reentrantes, el kernel pausa la ejecución de un proceso tras el paso de cierto tiempo, denominado *quantum*, generalmente en el orden de los pocos milisegundos. Esto evita tener que esperar a que la tarea termine o quede bloqueada por la espera de algún recurso (lectura del disco duro, llegada de paquetes de red...), y permite realizar el intercambio de tareas más a menudo, lo que da una sensación de concurrencia al usuario, a pesar de que exista solo un núcleo en el procesador. En estos últimos casos, si se desea tener una interfaz gráfica moderna, resulta imprescindible.
- Con soporte o no para prioridades. Tareas distintas tienen prioridad sobre otras, y esta prioridad se puede especificar numéricamente en los schedulers con soporte para prioridades. En los schedulers más simples con prioridad surge el riesgo de *inanición*, por el cual procesos de baja prioridad pueden potencialmente estar sin ejecutarse más tiempo del esperado: incluso infinito de haber algún problema con los más prioritarios.
- Según su nivel de tiempo real. Existen kernels muy específicos para tareas de *Safety-Critical Systems*, es decir, aquellos que pueden resultar responsables de pérdidas humanas, que poseen schedulers de tiempo real, en los cuales cada tarea lleva asociada una restricción de tiempo antes de la cual debe concluir. De aquí se diferencian dos tipos: *hard real time*, en el cual es inadmisible que la tarea no concluya en el plazo dado (*deadline*), y *soft real time*, en el cual se toma una política de *best-effort*. Para este último caso, generalmente sirven sistemas operativos de propósito general: por ejemplificar, Linux y NT tienen varios schedulers, y uno de ellos es de tiempo real suave.

Se procede a hacer un muy breve repaso de los schedulers predecesores al que usará el kernel incluido en el sistema operativo de este trabajo.

1. Sistema monotarea. En DOS (y esto incluye a MS-DOS), no existía el concepto de tareas en sí, pues solo podía haber una en ejecución en un momento dado. Cuando la tarea concluía, se volvía al prompt o se continuaba ejecutando el *batch* de tareas especificado en un archivo `.bat`.
2. Llamadas de bloqueo. En las primeras versiones de Windows, anteriores a Windows 95, el scheduler no tenía reentrancia, y las tareas eran responsables de liberar la CPU cuando consideraran oportuno mediante una syscall *yield*.
3. Round-Robin. Ligado al anterior, Round-Robin es un algoritmo genérico que representa una cola cíclica. Corresponde a cualquier tipo de scheduler con reentrancia o yield, cuyo orden de procesamiento sea cíclico: 1, 2, 3, 1, 2, 3, 1, 2...

4. Round-Robin multinivel. Extensión del anterior, pero ahora existen distintas colas para aportar soporte de prioridades. Se intenta tomar un proceso de la cola de máxima prioridad y, de no existir, se prueba la siguiente.

5. MLFQ. *MultiLevel Feedback Queue*, o cola multinivel con retroalimentación. Construido sobre el anterior, con la diferencia de que las prioridades de los procesos cambian dinámicamente dependiendo de si usan todo el quantum o se bloquean antes [21]. Surgen varios parámetros a tener en cuenta:

   - ¿Cuántas veces debe agotarse el quantum para bajar su prioridad?
   - ¿Es posible promocionar una tarea? Algunos schedulers MLFQ *suben* la tarea de cola en caso de que haya estado varios turnos sin concluir su quantum. En cuyo caso, ¿Cuántos turnos?
   - ¿Se permite fijar la prioridad de una tarea?
   - ¿Se usa el mismo quantum en todas las colas?

   NT, de Windows, y muchos derivados de BSD, incluyendo XNU, de macOS, usan variantes de este algoritmo [22] [23]. Por defecto, tiene el posible problema de que tareas de baja prioridad pueden sufrir inanición, y por esto no se suele implementar como tal.

6. Mención honorífica: CFS. Linux, desde su versión 2.6.23, utiliza por defecto CFS (*Completely Fair Scheduler*). Se trata de un *Red-Black tree*, una estructura de datos en forma de árbol similar a un AVL, es decir, un árbol binario de búsqueda autobalanceado. En esta estructura, las tareas pendientes se mantienen ordenadas según la cantidad de nanosegundos que se hayan ejecutado (*virtual runtime*). Además, el quantum es dinámico, varía según la carga del sistema. Resulta subóptimo para microkernels, pues la implementación de un árbol rojo-negro es compleja y termina siendo una estructura que se usa exclusivamente en el scheduler [24].

### 2.3.3. Sistema de archivos

Un sistema de archivos es una organización de la memoria secundaria que permite asignar regiones del espacio disponible a distintos datos, creando así el concepto de *archivo*. Un archivo puede ser de varios tipos, los más fundamentales son los regulares, secuencias de bits de estructura interna arbitraria (como los archivos de texto), y los directorios, agrupaciones de referencias a otros archivos.

Todo archivo tiene una serie de metadatos: nombre, tipo, y tamaño. Dependiendo del diseño, también puede tener su fecha de creación, y distintos valores que definan los permisos de acceso y modificación según el usuario.

Todo sistema de archivos tiene al menos un directorio: la raíz (*root*), dentro del cual se encuentran el resto de archivos. La estructuración de los directorios puede ser de nivel restringido o jerárquico. En la primera, existe una limitación de la profundidad de anidación de directorios (uno o dos); este era el caso de algunas versiones antiguas de DOS y CP/M. La falta de agrupación de los archivos llevó rápidamente a la invención de la estructuración jerárquica, en la cual no existe un límite como tal en la profundidad de los directorios (aunque sí pueden existir otros, como la longitud de ruta). En este último caso, la jerarquía

se puede expresar en forma de árbol, en el cual todo archivo se encuentra únicamente en un directorio, o en forma de grafo, que permite varias referencias a un mismo archivo, encontrándose en varias rutas simultáneamente. La mayoría de sistemas de archivos, como los derivados de UNIX, tienen un árbol de directorios de este tipo.

Existen sistemas de archivos de solo lectura, como ISO9660, el usado por los CDs y los archivos con extensión `.iso`. Estos sistemas permiten una gestión óptima del espacio, puesto que la estructuración de ficheros ocurre solo una vez [25]. Sin embargo, la mayoría no son de este tipo, sino alterables. En estos casos, ha de tenerse especial cuidado con la organización del espacio libre para aprovecharlo lo máximo posible. Los SSAA antiguos estaban basados en la reserva de espacio secuencial, y existía el problema de la *fragmentación*, por el cual algunas regiones del disco quedaban inutilizables, y se requería un proceso de desfragmentación para reorganizar todo el almacenamiento y unir estas secciones inutilizables de forma que se consiguieran juntar todos los *huecos* secuencialmente, maximizando el espacio de almacenamiento secuencial. Otras ramas de sistemas de archivos, como la de UNIX (con UFS) [13], la moderna de Windows (NTFS) [26], las de Linux (ext*) [27], y los BSDs, hacen una estructuración indexada. En el proceso de dar formato al disco por primera vez, se establece una región de memoria para almacenar bloques de datos, y estos se reservan y liberan de forma dinámica cuando es necesario crear uno nuevo. Con tal de no limitar excesivamente el tamaño de los archivos, se suele hacer una indexación multinivel.

Todo archivo tiene un descriptor asociado. En UNIX, este descriptor se denomina *inodo*, y contiene toda la información sobre un archivo (metadatos, tabla de índices de bloques...) salvo el nombre, que se excluye con tal de poder hacer múltiples referencias con distintos nombres al mismo archivo [13]. De igual forma que las tareas con los PIDs, en los SSAA modernos, los archivos se identifican numéricamente. En UNIX y derivados, se utiliza el número de inodo. En Windows con NTFS, existe un concepto similar, el *File ID*.

Con esto, todo directorio referencia a los archivos por su identificador numérico, y esta secuencia de referencias se extiende hasta la raíz. En UNIX, el descriptor a este directorio especial está situado en una estructura global al sistema de archivos, el superbloque, que en versiones modernas se encuentra replicado a lo largo del espacio para aportar tolerancia a fallos.

Por último, los SSAA modernos implementan el concepto de *journaling*, mediante el cual se mantiene un registro de las operaciones pendientes de escritura al disco. Esto logra hacer estas operaciones de forma semiatómica (esto es, minimizando el tiempo de escritura). Así, en caso de corte inesperado del sistema operativo (por ejemplo, fallo en la alimentación), se consigue evitar en la mayoría de los casos la corrupción de las estructuras.

Generalmente, y salvo ocasiones concretas, los dispositivos de almacenamiento se encuentran divididos en particiones, cada cual formateada con un sistema de archivos, que usualmente el mismo. Las particiones se organizan siguiendo un esquema de particiones, de los cuales existen dos más importantes: DOS (también llamado MBR) y GPT. Las limitaciones del primero (cuatro particiones salvo *hacks* y tamaño máximo reducido) han hecho que, con los años, GPT sea la opción más usual.

## 2.4. Práctica de un sistema operativo

En la sección 2.3 se han visto los aspectos más teóricamente relevantes del campo. Sin embargo, a la hora de comenzar un proyecto de esta magnitud, el programador no tarda en darse cuenta de que las áreas mencionadas son una parte muy pequeña del código necesario para conseguir escribir el código del sistema operativo más básico.

Esta sección incluirá explicaciones independientes de la arquitectura sobre estos conceptos que han faltado por explicar: la gestión de memoria, las interrupciones, y los drivers necesarios.

### 2.4.1. Memoria

Tras la preparación inicial del procesador, al inicio de la ejecución del bootloader, la memoria es un espacio contiguo de palabras, se denomina *memoria física*, pues la dirección se emite por el bus de direcciones. Para tener un mejor manejo sobre ella, se crea el concepto de *memoria virtual* (también llamada *lineal*). Esta aparece con el concepto de paginación, un mecanismo que ofrecen la mayoría de arquitecturas: la memoria se divide en páginas, de tamaño dependiente del ISA (siempre potencia de dos bytes), y cada página virtual corresponde a una física del mismo tamaño, aunque pueden existir varias virtuales que apunten a la misma física. Este mecanismo permite al kernel crear una estructuración propia de la memoria principal y manejarla con los rangos que él considere [28].

Es importante notar que una dirección física no tiene por qué corresponder a una región en RAM, sino que puede usarse para realizar MMIO (*Memory Mapped Input/Output*) con tal de comunicarse con otros chips conectados.

El *mapping* de virtual a física se realiza mediante una estructura denominada genéricamente como *tabla de páginas*. Toda arquitectura tiene, además, un registro protegido (acceso únicamente permitido al modo supervisor) que apunta a esta estructura. Es muy usual que, debido a la gran cantidad de memoria física disponible, esta tabla de páginas se realice de forma multinivel. Cada tarea tiene una tabla de páginas propia que forma parte de su contexto [28].

Las entradas de la tabla de páginas no solo tienen la dirección física a la que apunta la virtual, sino varias *flags* que actúan como atributos. La mayoría de estas flags tienen propósitos de protección, y en caso de incumplirse generan una excepción. Son imprescindibles la de solo lectura y la que indica si es accesible en modo usuario. En ocasiones, también pueden aparecer flags más concretas, como puede ser la de no-ejecución o la de no-caché (que evita que se almacene en L1-L3).

En el momento en que se habilita la memoria virtual, todas las direcciones que se emitan al bus de direcciones pasan primero por un chip aparte (originalmente; hoy en día todo se implementa dentro de la CPU), la MMU (*Memory Management Unit*), que se encarga de hacer la traducción. La MMU tiene una caché para almacenar las traducciones más ocurrentes, se trata del TLB (*Translation Lookaside Buffer*). El procesador cambia la tabla de páginas en el proceso de cambio de tarea en ejecución (cambio de contexto), y es conocida como una operación costosa especialmente porque conlleva un *TLB flush*, es

decir, el borrado de todas las traducciones cacheadas, salvo las marcadas como globales mediante una de las flags.

La enorme mayoría de kernels modernos se cargan a sí mismos en la mitad superior de la memoria virtual, región conocida como *higher half*. Esto permite que las estructuras multinivel más amplias sean marcadas como globales, pues se mantienen constantes entre contextos.

### 2.4.2. Interrupciones y excepciones

Todo procesador de cualquier arquitectura recibe interrupciones. En la subsección 2.2.1 se mencionó que existen dos tipos: enmascarables, y no enmascarables. El proceso de enmascarar se refiere a la capacidad de desactivar una de ellas.

Antes de nada, un breve comentario sobre la notación: las interrupciones hardware se suelen denominar por sus siglas terminadas en `#`, y las excepciones por sus siglas comenzando con `#`. Así, un error de protección general, que es excepción, se denomina #GP (o #GPF), mientras que la interrupción legacy `A` de PCI se denomina INTA#.

Las interrupciones no enmascarables (en otras palabras, no evitables) se denominan excepciones: fallos captados por la CPU durante la ejecución de una instrucción. Se mencionó con anterioridad la división por cero, pero la más significativa es el fallo de página (*page fault*, #PF), que ocurre cuando una de las protecciones de una página virtual no se ha cumplido. Un fallo de página no es necesariamente malo, pueden usarse como herramienta para detectar situaciones en las que el kernel debe realizar una acción sobre el proceso como, por ejemplo, ampliar el tamaño de la pila reservado a la tarea, lo que permite reservar una sola página de stack al iniciar el programa, y solo si lo necesita aportarle más.

Las interrupciones enmascarables (evitables) pueden ser de dos tipos:

- Causadas por el usuario. En algunas arquitecturas donde el ISA no contiene una instrucción de llamada al sistema (como IA-32), se utilizan interrupciones en su lugar.
- Causadas por el hardware, generalmente ajeno a la CPU. Se las denomina IRQ (*Interrupt Request*). El ejemplo más fácil de entender es el reloj del sistema, configurado con el kernel con tal de implementar un mecanismo de reentrancia. Se configura para disparar interrupciones cada quantum, y esto hace que se vuelva al código del kernel.

Toda interrupción debe asociarse a una rutina de interrupción (ISR, *Interrupt Service Routine*), que contiene el código que se ejecuta al recibirla. Es muy usual que este código se ejecuta en modo supervisor, y en ese caso el procesador cambia de modo de ser necesario, carga una stack del kernel para dicha interrupción en concreto, y hace el salto.

Las arquitecturas cuentan con un controlador (*controller*, no confundir con driver) de interrupciones programable, y todo kernel debe contener un driver para soportarlo, y así enmascarar y desenmascarar interrupciones, así como hacerle saber al chip cuál es la dirección del ISR, y el puntero de pila a usar. De haber un problema en el ISR, o directamente no existir, las arquitecturas suelen producir una excepción concreta denominada *double*

*fault* (#DF), que se espera que tenga un ISR asociado y funcional. De volver a encontrarse un fallo procesando un #DF, se produce un *triple fault*, que es la única interrupción de una arquitectura a la que no se puede asociar un ISR. Dependiendo del procesador, la acción a tomar es congelar la CPU (*halt*), o reiniciar el sistema.

### 2.4.3. Comunicación con el hardware

Se mencionó en la subsección 2.4.1 que algunas comunicaciones con el hardware pueden producirse por MMIO. En estos casos, el controlador correspondiente redirige las direcciones al posarse sobre el bus de direcciones de la placa base al chip que tenga asociado (*hardwired*). Por esto mismo, esas porciones de la memoria física están garantizadas que no corresponden a una porción de RAM usable, aunque pueden ser movidas en tiempo de ejecución mediante *bank switching*. Estas regiones de memoria física son desconocidas para el programador, y cambian de ordenador en ordenador, con lo cual las arquitecturas aportan un método de descubrir la distribución de las regiones mediante una estructura conocida como *mapa de memoria* (*memory map*). Este mapa de memoria suele ser leído y reestructurado por el bootloader con tal de pasarle al kernel una versión más manejera.

MMIO no es el único método de comunicación con hardware. En algunas arquitecturas, existe PIO (*Port-mapped IO*), una clase especial de instrucciones dadas por el ISA con este único propósito. No son necesarias más de dos, y sus mnemónicos suelen ser `IN` y `OUT`, aunque con distintos sufijos para indicar el tamaño de palabra a transferir, nunca más del tamaño del registro. Cada controlador de entrada/salida establece un conjunto de direcciones PIO por chip al que está conectado. Suelen ser direcciones pequeñas de 16 bytes.

Para asentar los conocimientos dados, el controlador más famoso, aunque no el único que existe en un ordenador de sobremesa convencional, es el de PCI Express.

### 2.4.4. Drivers necesarios

Para terminar la sección, se procede a comentar qué drivers resultan imprescindibles en un sistema operativo moderno, para conseguir orientar al lector un poco en el trabajo requerido para realizar un proyecto de esta índole.

Independientemente del propósito, es necesario el driver del controlador de interrupciones, que varía según la arquitectura. Se debe configurar y desenmascarar las interrupciones nescesarias, así como proveer las direcciones de los ISRs y reservar los marcos de pila de cada una, por cada núcleo.

Si se desea acceder a archivos, es necesario una pila de almacenamiento (*storage stack*), es decir, los módulos que juntos ofrecen las abstracciones necesarias para acceder a archivos.

- En la parte más baja de esta pila de drivers, se encuentra el driver del bus al que está conectado el dispositivo de almacenamiento a acceder. En sistemas modernos suele ser PCI Express.
- Sobre ello, el driver en sí del controlador al que está conectado el dispositivo de alma-

cenamiento masivo. Este último puede ser un disco duro (en sus múltiples formas), un CD, un pendrive... Dependiendo del dispositivo de almacenamiento en concreto, el controlador al que está conectado es de un tipo u otro. Por ejemplificar, un disco duro SATA puede estar conectado a un controlador AHCI, un ATA a un IDE, un M.2 a un NVMe, o un pendrive a xHCI. Muchos de estos controladores son retrocompatibles, con lo que AHCI puede emular IDE, y xHCI a EHCI (y este a su vez a OHCI o UHCI).

- No es inusual encontrar sobre el driver del controlador una abstracción que nombre los dispositivos de almacenamiento, como se mencionó en la sección 2.2.2.
- Encima de esto último, puede encontrarse un programa que interprete el esquema de particiones.
- Luego, la implementación del sistema de archivos a usar.
- Y, finalmente, el VFS (*Virtual File System*), que abstrae todos los sistemas de archivos disponibles en el sistema para aportar una API uniforme e independiente de todo lo que está bajo la cima de la pila.

Se profundizará en la avalancha de siglas en capítulos posteriores.

## 2.5. Fundamentos de x86-64

En la subsección 2.1.1 se comentó que x86-64 es el target del sistema operativo propuesto. Así, resulta natural ofrecer una contextualización de la arquitectura tan pronto como sea posible. Esta sección cumplirá ese objetivo. Se presentará la arquitectura, su forma de arranque y sus estructuras, como los descriptores que necesita y su forma de realizar paginación. Todo de forma muy breve y superficial, sin llegar a detalles de la implementación.

### 2.5.1. Presentación

El mundo de la informática tuvo un punto de inflexión en 1981 con la salida del *IBM Personal Computer* en Estados Unidos [29]. Su procesador, el Intel 8088, lanzado al mercado 5 años antes, fue la primera pieza de hardware en usar la arquitectura x86. Rápidamente empezó a ganar popularidad, y, por su alto precio, poco después de la salida al mercado, otras compañías productoras de hardware y software crearon las denominadas *compatibles*, computadores cuyo hardware permitía la ejecución del software diseñado para la máquina de IBM, de las cuales cabe destacar las de Compaq. La enorme presencia en mercado de las compatibles ha desencadenado en que, para finales de la década, x86 fuera la arquitectura más utilizada, y mantiene ese puesto a día de hoy.

Como x86 data de tan atrás, muchas de las decisiones originales de diseño se han ido quedado obsoletas, y las subsiguientes generaciones de procesadores aportaron nuevos ISAs que han ido dejando abandonando funcionalidades e introduciendo otras. Sin embargo, muchas de ellas, por el propio diseño de la arquitectura, siguen siendo necesarias hoy en día, y el programador del sistema debe implementarlas. Esto hace que desarrollar un kernel desde cero para x86 sea un proyecto bastante más complejo que el de una arquitectura moderna como puede ser ARM64.

La mayoría de computadores personales de hoy día usan la arquitectura x86, sobre el ISA (*Instruction Set Architecture*, conjunto de instrucciones) de 64 bits llamado x86-64 (también conocido como x64 o amd64). Existe un resurgimiento de la arquitectura ARM fuera de móviles por parte de los procesadores Apple Silicon publicados desde 2020, pero a día de hoy su presencia no consigue alcanzar la de los x86-64.

### 2.5.2. Introducción al arranque x86

Cuando un x86 arranca, se ejecuta un programa aportado por un chip ROM sobre la placa base. Se denomina BIOS (*Basic Input Output System*) en su versión original, aunque en la década pasada fue poco a poco reemplazado por UEFI (*Unified Extensible Firmware Interface*) hasta apoderarse del mercado. UEFI suele tener en la gran mayoría de ocasiones un modo *legacy* para simular ser una BIOS y así mantener la retrocompatibilidad. Esta sección se referirá solo a BIOS con tal de acotar un cierto nivel de simplicidad.

La BIOS realiza tareas de preparación del hardware, como inicializar el controlador de la memoria DRAM y puertos PCI, aunque su forma de hacerlo varía entre fabricantes y modelos. Cuando el hardware esencial ha sido inicializado, se prepara una interfaz de bajo nivel que puede usar el programador del sistema: se trata de las llamadas de interrup-

ción BIOS, ampliamente usadas en la época de MS-DOS, cuando no existía un kernel lo suficientemente amplio como para abstraerse del hardware.

Tras montar este sistema de interrupciones, selecciona un disco de arranque, proceso que ha presenciado todo entusiasta de la informática a la hora de instalar un sistema operativo. De este disco, sea magnético, en estado sólido, unidad CD, o USB, BIOS lee el MBR (*Master Boot Record*), su primer *sector* (conjunto pequeño de bytes, usualmente 512 en discos duros y 2048 en CDs). El MBR es copiado a una región de memoria que comienza en `0x7C00`, por convenio de IBM, y BIOS hace el salto a esta dirección [30]. A partir de este punto, el programador del sistema está en control.

Cuando la BIOS salta al punto de entrada, el procesador se encuentra en un estado conocido como *real mode*, o modo real. Este modo es plenamente compatible con un procesador 80186 de Intel, y su ISA es x86-16, es decir, tiene un tamaño de palabra de 16 bits. Para desbloquear el verdadero potencial de la CPU, el procesador debe de cambiar al *protected mode* (modo protegido), capacidad que apareció por primera vez en el Intel 80386 (también llamado i386), que usa el conjunto de instrucciones IA-32, con una longitud de palabra de 32 bits. Eventualmente, también tendrá que pasar al *long mode* (modo largo), con el ISA x86-64, que corresponde a lo usado hoy en día [31].

Todo este proceso de cambio de modos es realizado por una pieza de software: el *bootloader*, o cargador de arranque. GRUB [32] es el que posee el nombre más conocido, pero existen multitud. Por ejemplo, las versiones modernas de Windows usan BOOTMGR [33]. El bootloader utiliza las interrupciones BIOS para reconocer los discos conectados y poder acceder a ellos posteriormente. Tras hacer el cambio de modos, reconoce los esquemas de particiones, así como las particiones en sí, y carga los archivos necesarios del kernel, para después darle el control, ofreciéndole en el proceso información vital para la posterior preparación del sistema (por ejemplo, el mapa de memoria mencionado en la sección 2.4.3.

### 2.5.3. La memoria en un x86

Desde su comienzo, x86 ha tenido un modo de manejo de memoria: la segmentación. Está obsoleta en x86-64 y se desaconseja su uso a los programadores de sistemas. Sin embargo, es obligatorio implementar una mínima funcionalidad por retrocompatibilidad. Segmentación divide la memoria, como indica su nombre, en segmentos. Así, un programa es una colección de unidades lógicas: código, pila, heap...

En segmentación, una dirección lógica es un par `selector:desplazamiento`. Un selector es un índice de segmento con flags de protección. Los segmentos son de 64KB en modo real, y de hasta 4GB en modo protegido. Existen, además, 6 registros de segmentos, que se usan para formar direcciones cuando se quiere hacer una lectura/escritura de memoria. Son: `CS` (*Code Segment*), `DS` (*Data Segment*), `SS` (*Stack Segment*), `ES` (*Extra Segment*) y `FS` y `GS`, ambos de propósito general [34].

IA-32 incluye nuevas instrucciones con microcódigo que afecta a los registros de segmentación. De ellas, *far jump* e *IRET* son las más usadas. La primera se suele utilizar en los bootloaders para realizar el cambio de segmento en `CS` junto a un salto de forma atómica (hacerlo por separado dispararía una excepción), y la segunda se solía usar en los kernels para realizar cambios de contexto, pues además de cambiar `CS` realiza el cambio de

`EFLAGS`, el registro que tiene las flags del sistema [35].

La segmentación en x86 se define por dos estructuras: en mayor medida, la GDT (*Global Descriptor Table*), y en menor, la LDT (*Local Descriptor Table*). Se comentará solo la GDT por ser relevante a día de hoy, pero la LDT es similar.

El registro GDTR contiene un puntero físico a la GDT, y se carga por medio de la instrucción LGDT. La GDT es un array de entradas, *GDT entries*, cuyo índice forma parte del selector del segmento. El primero siempre es nulo, y el resto contienen descriptores de segmentos, que están formados por una dirección base del segmento, su tamaño, y algunas flags. Esta estructura es un claro indicativo del paso del tiempo en x86: extensiones en IA-32 y x86-64 han dejado bits de la base y el tamaño (límite) desperdigados en los 64 bits que la componen, véase la figura 2.1 [36].

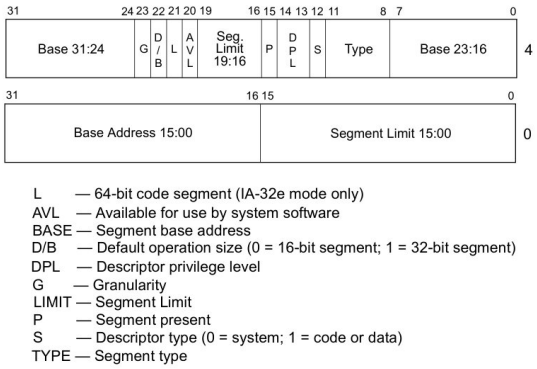



Figura 2.1: Descriptor de segmento para direcciones de 64 bits [36]

Con la GDT aparece el modo usuario en x86. El campo `DPL` que se aprecia en la figura 2.1 es un número denominado anillo de protección (en inglés, *protection ring*). En x86-64 existen 4 anillos, comenzándose en el *ring 0*, y dando libertad al kernel de elegir si los demás están en modo supervisor, usuario, o incluso una mezcla de ambos. Generalmente, para modo usuario se usa el cuarto anillo (*ring 3*), y los anillos 1 y 2 no se utilizan en absoluto, pero un SO podría usarlo para drivers, a costa de complicar la portabilidad. Por ejemplo, en ARM no existen los anillos, sino como tal los modos supervisor y usuario, entre otros.

Si bien segmentación está obsoleta, un kernel de x86-64 debe aportar una GDT válida, con, como mínimo, dos entradas: código y datos del kernel. Si espera implementar tareas en modo usuario (userspace), entonces necesita otras dos: código y datos de usuario.

El modo protegido del i386 trae a x86 el ISA IA-32. Entre las nuevas funcionalidades, se encuentran otra forma de manejo de memoria: la paginación. IA-32 tiene direcciones de 32 bits, con lo que se tienen 4 GBs de memoria virtual direccionables, que se distribuyen generalmente en páginas de 4 KBs (también existen las páginas *huge* de 4MB). Como hacer un array en memoria de cada página virtual con su física equivalente resulta inasequible, se usa paginación multinivel. Se define una tabla de páginas como una página con un array de 1024 entradas, cada una de la cual corresponde a una página virtual, y en cada una se encuentra su correspondiente memoria física. Sobre esto, se crea el directorio de páginas, otra página con un array de 1024 punteros (físicos) a tablas de páginas. En la figura 2.2 se encuentra una representación. La dirección de esta última página se la conoce usualmente

como el *puntero a la tabla de páginas*, aunque realmente apunte al directorio. Para utilizar estas estructuras, los procesadores x86 tienen el registro `cr3` donde se sitúa el puntero.

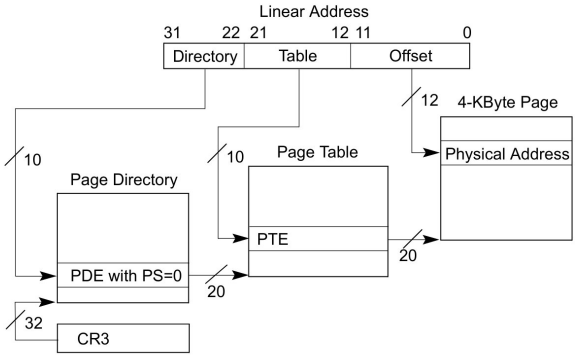



Figura 2.2: Paginación para direcciones de 32 bits [37]

Con el tiempo, se veía venir la obsolescencia inminente de IA-32. Para alargar su vida, apareció la tecnología PAE (*Physical Address Extension*), presente en todo x86 moderno de 32 bits (y todos los de 64), mediante la cual se permite un acceso a una memoria física de más de 4GB por medio de paginación a 3 niveles.

Después de PAE, con x86-64 el espacio de direccionamiento se vuelve de 64 bits, y se tiene una cantidad de direcciones virtuales cuatro mil millones de veces mayor a la de IA-32. Por esto, son necesarios más niveles. Los procesadores generalmente no soportan direcciones de 64 bits, sino de al menos 48; el resto de bits se producen por expansión de signo, y las direcciones de este tipo se denominan direcciones canónicas. Las direcciones de 48 bits se representan por paginación a 4 niveles. Ahora, una tabla de páginas tiene 512 entradas, y un directorio de páginas 512 punteros. Aparecen sobre los directorios de páginas los PDPs (*Page Descriptor Pointer*), y sobre estos últimos los PML4 (*Page Map Level 4*). En la figura 2.3 se encuentra una representación. Algunos procesadores permiten paginación a 5 niveles para acceder a más memoria virtual aún (57 bits), y estos usan los PML5. Suponiendo ahora una paginación a 4 niveles, el registro `cr3` contiene el puntero que apunta a la página con el PML4.

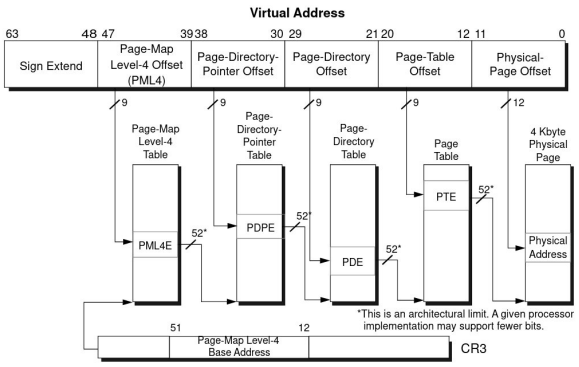



Figura 2.3: Paginación para direcciones de 48 bits [38]

A día de hoy, la paginación a cuatro niveles de x86-64 pone a disposición del programador del sistema algunas flags. Estas son las más relevantes:

- Presente. Se alza cuando la página virtual tiene una física correspondiente.
- R/W. Se pone a uno cuando se desea permitir la escritura.
- U/S. Ondea cuando la página es accesible tanto por el kernel como en userspace.
- PCD. Está activa cuando se desea deshabilitar que la página se almacene en la caché de la CPU. Es útil para MMIO.
- G. Alzada cuando la página es global, y por tanto no debe resetarse durante un TLB flush.

Existen más, algunos de ellos enfocados a protecciones modernas de memoria, y se guardan en la PAT (*Page Attribute Table*), pero es mejor quedarse aquí. Cuando una de las flags no corresponde al estado del procesador durante un acceso, se dispara #PF.

### 2.5.4. Interrupciones y syscalls

Se conoce x86 como una arquitectura guiada por interrupciones, y su correcto manejo es una parte crítica del kernel.

Lo más fundamental: las interrupciones en x86 están numeradas con un identificador del 0 al 255, llamado vector de interrupción [39]. Originalmente, llegaban a un x86 mediante pines propios en la CPU, desde un chip encargado de manejar las interrupciones. Se trata de la PIC 8259 (*Programmable Interrupt Controller*), usado por el IBM PC, y hoy en día, como es esperable, está dentro de la CPU. El hardware envía IRQs a la PIC. Esta hace la transformación a su identificador de x86, comprueba si está enmascarada, y, de no estarlo, la manda al procesador. En el próximo ciclo de reloj, el ciclo fetch/execute comprobará si hay interrupciones pendientes. Como la hay, guardará en la pila el estado (flags y registros fundamentales), hará el cambio de contexto, y saltará al ISR.

¿Cómo sabe la CPU dónde está el ISR? La estructura que los maneja es la IDT (*Interrupt Descriptor Table*), y es muy similar a la GDT. IDTR es un registro, cargado con la instrucción LIDT, que apunta a la IDT. Es un array de entradas, *gate descriptors*, que indican, sobre todo, la dirección del ISR y un índice de la IST (*Interrupt Stack Table*), otra estructura manejada por la CPU que contiene un array de pilas disponibles para la CPU en caso de interrupción [40].

En x86-64, la IST se encuentra dentro de otra estructura de IA-32 que quedó obsoleta, la TSS (*Task State Segment*), cuyo objetivo original era el cambio de contexto por hardware, idea que ha quedado en desfase por implicaciones de velocidad y control del kernel. Sabiendo su contexto no resultará extraño lo siguiente: cada TSS necesita una entrada en la GDT. Tras este cambio, la TSS tiene 6 entradas de IST (del 1 al 7) para interrupciones específicas, así como otras 3 que se usan en caso de que el índice de IST sea cero [41].

Un rango de identificadores de interrupción está dedicado a las excepciones, del 0 al 30. Por ejemplo, #PF es 14, y #GP es 13 [42]. Las demás, las interrupciones hardware, son traducidas por la PIC desde su valor de interrupción hardware, IRQ $n$, a su vector de interrupción x86. El kernel se encarga de establecer esta tabla de traducción al arranque.

Con el surgimiento de los multiprocesadores x86, la PIC se ha quedado atrás. Ha de mantenerse retrocompatible, y no se puede extender para soportar varios procesadores. En su lugar, se ve obligada a emitir la interrupción hardware a todos los cores, y los cambios de contexto innecesarios ralentizan todo el SO. Por esto, se diseñó una sucesora, la APIC (*Advanced PIC*).

En este nuevo sistema SMP (*Symmetric Multiprocessing*) que es x86, donde cada core tiene acceso a toda la memoria, el procesador es una combinación de pares `<core, LAPIC>`. Esto es, todo núcleo tiene un chip asociado, una LAPIC (*Local APIC*), y todos los cores comparten un único IOAPIC (*Input/Output APIC*). Cuando una IRQ llega al procesador, la maneja la IOAPIC. Esta se comporta como la PIC, en el sentido de que la traduce a su identificador y comprueba si está enmascarada, con la gran diferencia de que, en caso de no estarlo, redirige la interrupción a una de las LAPIC. Se puede configurar de dos formas: destino fijo, en el cual se especifica uno de los cores como receptor, o por prioridad, en el que la CPU intenta averiguar qué núcleo tiene menos trabajo (por tiempo en HALT) y se la envía. Además, la APIC permite las denominadas IPIs (*Inter-processor Interrupts*), una forma de generar interrupciones software entre distintos cores para sincronizarlos [43]. Además, incluye un reloj que emite interrupciones en un intervalo configurable, el *LAPIC timer*.

Con x86-64 aparecen dos nuevas instrucciones muy relevantes: *syscall* y *sysret* [44]. *Syscall* realiza el cambio de modo, desde usuario a supervisor, y en el proceso establece RCX=RIP (se guarda el contador de programa) y R11=RFLAGS (se guardan las flags). *Sysret* lo deshace, con RIP=RCX y RFLAGS=R11, y vuelve a modo usuario. El kernel inicializa este comportamiento por medio de un MSR (*Model Specific Register*), un registro accedido con una instrucción específica (WRMSR). En concreto, el MSR relevante a *syscall* es el EFER (*Extended Feature Enable Register*), existiendo una flag para SCE (*System Call Extensions*). Cuando está habilitado, en otros MSRs (STAR y LSTAR) se le hace saber a la CPU qué selectores se usan para modo supervisor y usuario, así como cuál es el punto de entrada, que se podría entender como la dirección a un ISR genérico [44].

Nótese cómo estas instrucciones no escriben nada en la pila, ni siquiera cambian RSP (puntero de pila) ni la tabla de páginas. Por esto, se realiza un cambio de contexto muy limitado, el kernel considerará posteriormente si es oportuno cambiar estos valores. Esta nueva forma de realizar llamadas al sistema supone un incremento de velocidad mucho mayor al que existía anteriormente en x86, y resulta una herramienta muy útil para el programador del sistema.

# Capítulo 3

# Estado del arte

En este capítulo se analizarán algunos sistemas operativos existentes, más antiguos o más modernos, con tal de ofrecer al lector una visión general del abanico de elecciones posibles a la hora de diseñar un sistema operativo. En el capítulo siguiente, se referenciarán muchas de las opciones aquí presentes y se justificarán las decisiones.

## 3.1. UNIX: el primer mejor SO

En 1925, Western Electric crea Bell Telephone Laboratories, conocido usualmente como *Bell Labs* [45]. Doce años después, se produce el primer premio Nobel del lugar por el descubrimiento de la difracción de electrones [46]. Los laboratorios Bell no han sido otra cosa a lo largo de su historia que una gran fábrica de premios Nobel, y grandes descubrimientos que han hecho avanzar la humanidad, como el transistor, el láser, y la célula fotovoltaica vienen de allí. Entre todos estos logros, existe uno relativo a este trabajo: UNIX.

Ken Thompson, conocido también por la invención de UTF-8, y Dennis Ritchie, conocido por la creación de C, comienzan en 1969 UNIX, un proyecto de sistema operativo para el PDP-11, poco después del fracaso de otro intento anterior, Multics. UNIX trae consigo varias novedades:

- Los *pipes*, o tuberías, ya mencionados en 2.2.1, un método de IPC por el cual los programas se comunican mediante flujos de bytes.
- La filosofía UNIX, mencionada en 2.2.4, en la cual cada programa se enfoca en resolver un solo problema, manteniéndose simple. Estos programas posteriormente se combinan mediante pipes por medio de la shell, como explica el propio Brian Kernighan, también eminencia de Laboratorios Bell, en [47].
- Un sistema de archivos, UFS, basado en inodos.

El gran pilar de UNIX es la intercomunicación. Por esto, todo es un archivo: desde un disco duro en sí hasta sus particiones, y desde el chip Ethernet hasta la configuración del kernel. Hasta las tareas, en cierto modo, son archivos (en `/proc`).

## 3.2. AmigaOS

UNIX tenía un kernel monolítico, pero no es el concepto de monolito que existe hoy en día. En aquella época, si bien diseñar un sistema operativo tenía una dificultad aproximable a la actual, desarrollarlo era extremadamente más simple: existían muchos menos dispositivos con los que comunicarse y los procesadores eran mucho más simples. Por todo esto, la cantidad de líneas de código que tomaban los drivers era mucho menor, y no se consideraba la idea de separarlas.

No sería así, al menos, hasta bien entrados los 70. Se suele considerar el primer microkernel el de *RC 4000 Multiprogramming System*, un SO para máquinas con un propósito muy específico (una planta fertilizadora) creado en 1969 [48]. Durante las décadas de los 70 y los 80, gran parte de la investigación de SSOO se enfocó en microkernels, por ser teóricamente más interesantes que los monolíticos.

El primer producto de este estilo en triunfar económicamente fue la Commodore Amiga, en 1986, con su sistema operativo AmigaOS y su microkernel Exec. Utilizaba un IPC basado en paso de mensajes. Este mensaje estaba preparado en el proceso A. Como no existía en aquel entonces el concepto de protección de memoria, solo existía un espacio de direccionamiento compartido por todas las tareas, así como el kernel. Por esto, el paso del mensaje al proceso B consistía en solo transmitir el puntero a estos datos, lo que implicaba cero copias de los datos [49].

AmigaOS supo utilizar el entorno a su favor para crear lo que es a día de hoy el microkernel más rápido que ha existido. Los SSOO basados en paso de mensajes generalmente requieren copias, como mínimo una si se diseña bien, lo que supone una gran diferencia con respecto al método de Exec.

## 3.3. Familia L4

### 3.3.1. L3

Jochen Liedtke se podría considerar el tutor legal de los microkernels. Si bien no es el padre, fue uno de los grandes investigadores al respecto, y su trabajo e ideas perdurarán en todos los microkernels por venir. En 1987, tras varios años de experiencia con SSOO, comenzó el diseño de L3, con la intención de demostrar que un IPC liviano y muy enfocado en un diseño *machine-specific* podía realizar paso de mensajes sin *overhead* añadido. En L3, el diseño e implementación de las políticas de seguridad (quién puede comunicarse con quién) se delega a las tareas en sí, lo que libera mucho la carga del kernel [50].

### 3.3.2. L4

A principios de los 90, microkernels mal diseñados, y, sobre todo, lentos, acabaron dando mala popularidad al área. Un ejemplo es IBM Workplace OS, con su núcleo Mach, famoso por ser de los peores microkernels escritos [51]. Liedtke denominó a este suceso *el desastre de los 100 microsegundos*, por ser el tiempo mínimo necesario en Mach para realizar paso de mensajes. Por esto, decidió reimplementar L3 desde cero en ensamblador, lo que resultó en la reducción del overhead en un orden de magnitud. Este kernel se denominó L4 [52].

Dos años después de la publicación de L4, enunció por primera vez en 1995 el concepto conocido como *principio de minimalidad de los microkernels*:

> Un concepto es aceptado dentro del microkernel solo si moverlo fuera prevendría la implementación de la funcionalidad requerida del sistema. — Jochen Liedtke [53].

Uno de los puntos claves de L4 es el modelo de IPC síncrono. En él, el paso de mensajes de A a B se realiza bloqueando el proceso A con la syscall de envío. A partir de este punto, el kernel puede copiar, de forma segura, el mensaje desde el espacio de direccionamiento de A a B. Además, toma algunos registros como *registros de mensaje*, y su valor no se altera durante el cambio de contexto, lo que implica que parte del mensaje puede ser pasado con cero copias.

Han surgido numerosos sistemas operativos basados en microkernels derivados de este. Así, surge la familia de microkernels L4, similar (aunque mucho más pequeña) a la de UNIX. Una representación de este árbol genealógico se puede encontrar en la figura 3.1.

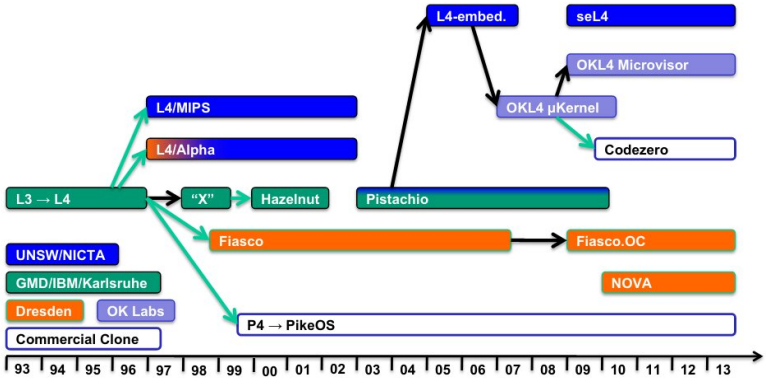



Figura 3.1: Árbol genealógico de L4 hasta 2013 [54]

### 3.3.3. seL4: el último mejor SO

El proyecto *seL4* arranca en 2007 a raíz de Gernot Heiser. Aparece, así, junto a un proyecto similar, EROS, la denominada *tercera generación de microkernels*. Está caracterizada por una API orientada a la seguridad con acceso a recursos controlados por *capabilities*, así como un foco en la virtualización y en enfoques modernos para el manejo de recursos del kernel.

La tercera generación de microkernels también tiene un objetivo de diseño que permita los análisis formales, es decir, una demostración matemática de que la implementación del kernel es consistente con su especificación. seL4 es de los muy pocos kernels que cuentan con una verificación formal completa en términos de seguridad, y además en términos de *timeliness* (temporización), haciéndolo apropiado para aplicaciones de hard real time [55].

El IPC de L4, aunque rápido, era complejo, y contaba con muchos objetivos. Entre ellos, el paso de mensajes pequeños (usual), el de mensajes grandes (*Long IPC*), y el de sincronización. seL4 abandona el *Long IPC*, pues con la experiencia se demostró que este mecanismo ralentizaba todo el sistema, sobre todo porque, en L4, el ISR de page faults se encontraba en userspace, lo que implicaba la necesidad de salir del kernel para resolver las copias de más de una página. En su lugar, si se desea pasar grandes mensajes, se utiliza memoria compartida [54].

L4 utilizaba PIDs para dirigir IPC, y con esto aparece un problema de canal encubierto. En su lugar, seL4 reemplaza esta semántica con *endpoints*, parecidos a puertos, donde cada uno puede tener un número arbitrario de procesos que envían y reciben.

# Capítulo 4

# Propuesta de solución

## 4.1. Metodología

### 4.1.1. Cascada

Un sistema operativo es el mayor exponente de un proyecto con metodología en cascada. A partir de cierto punto durante el desarrollo, es posible comenzar a paralelizar el trabajo, pero cuando lo que se está construyendo es la base, definida en 2.1 como una alta pila de capas de abstracción, donde cada paso adelante requiere la totalidad de los anteriores, se vuelve imposible. El proceso de desarrollo ha de ser necesariamente secuencial. Por esto, la metodología de desarrollo estuvo planeado desde el primer momento, y ha terminado siendo, una metodología en cascada.

### 4.1.2. Tests

Además, dada la naturaleza del proyecto, es imposible realizar vertificaciones informales, como podrían ser los tests unitarios, para comprobar que el funcionamiento es correcto. En un proyecto cuyo entorno natural no conoce los conceptos de *segmentation fault*, se estimaba que los bugs de corrupción de memoria fueran usuales, y así ha sido. Ninguna herramienta, aparte de la verificación formal, que, evidentemente, no se ha realizado por disponer únicamente de una vida para trabajar, es siquiera teóricamente capaz de comprobar que el funcionamiento de los subproyectos es el adecuado, porque un fallo de corrupción de memoria es indetectable.

### 4.1.3. Filosofía

La implementación del proyecto en este capítulo descrito está bajo la licencia de izquierdos de autor GNU General Public License 3.0 o superior. No es *Open Source*, sino software libre. Atendiendo a la defínción de la FSF, cumple con las cuatro libertades esenciales del software [56].

### 4.1.4. Herramientas utilizadas

Como herramienta para gestionar el trabajo a realizar, se eligió desde un primer momento *git*, y su publicación está en el portal GitHub. En el caso de este proyecto, en lugar de un repositorio con directorios para cada proyecto, se ha creado una organización (https://github.com/the-strife-project): un usuario virtual con su propio perfil y repositorios. De esta manera, todos los proyectos que componen el sistema operativo se encuentran separados.

El proyecto se ha realizado en C++11, utilizando GNU Make como herramienta para orquestrar el proceso de construcción del ISO final. Existe un único Makefile que utilizan todos los subproyectos: se ha denominado *helper*, y cada repositorio contiene un Makefile que lo incluye y customiza por medio de variables de entorno. Durante todo el desarrollo, las pruebas se han llevado a cabo con *qemu* como frontend para KVM, el hipervisor moderno de Linux.

Una *toolchain* (compilador, linker, ensamblador...) de C o C++, sea la de GCC o clang (frontend de LLVM), se compila para soportar un único target. La versión de `g++` incluida en las distribuciones GNU/Linux tiene como objetivo, para sorpresa de nadie, GNU/Linux. Está fuertemente conectada con la GNU stdlib y con el entorno userspace y kernel existentes en este sistema operativo. Con tal de poder empezar a escribir un sistema operativo, es necesaria una toolchain propia. Se ha utilizado, así, una versión compilada de G++ con el target `amd64-elf`, genérico, que se enfoca en producir binarios en formato ELF para x86-64. El repositorio *toolchain* organización contiene scripts para compilar estas herramientas.

Se ha usado CI/CD, en forma de GitHub Actions, en el repositorio principal del repositorio (explicado más adelante), así como para la toolchain. De esta forma, están públicamente disponibles archivos ISO para ser descargados.

## 4.2. Contexto

En el verano de 2019 me adentré en el mundo del desarrollo de sistemas operativos. Proyectos personales de tal magnitud no se comienzan de forma intencionada, sino que, con el tiempo, y cientos de horas de pruebas, uno se da cuenta de que ya dispone de todos los conocimientos necesarios para intentar aproximar el problema de forma seria y con asertividad. Un sistema operativo de prueba, y con el objetivo de aprender, comenzó dicho verano: jotadOS (posteriormente renombrado a jotaOS). Se trataba de un SO con IA-32 como target, con un kernel monolítico, escrito en C (posteriormente, C++), con su propio bootloader escrito en ensamblador (JBoot [57]), y sin mucho razonamiento e intenciones detrás más que las de *hacer cosas*.

Este proyecto llegó a tener una complejidad elevada: implementaba una pila de almacenamiento y tenía una shell megalítica (dentro del kernel, para demostrar la funcionalidad). Desde junio de 2020, cuando mis conocimientos ya habían alcanzado cierto umbral, aparecieron las dudas. El proyecto no era lo que quería ahora que fuera. Ahora que conocía los conceptos y los había puesto en práctica, sabía lo que quería:

- Un SO con microkernel, puesto que estaba dispuesto a *sacrificar velocidad por belleza*.
- Cambiar el target a x86-64, puesto que es aporta soluciones mucho más elegantes, modernas, y rápidas que IA-32. Además, el espacio de direccionamiento de 64 bits permitía un ASLR funcional (explicado en la sección siguiente).
- Abandonar el bootloader propio. Un bootloader es difícil de mantener, pues requiere gran parte de los drivers que hay en un SO convencional, con lo cual hay que escribir el mismo código dos veces. Además, JBoot, escrito en ensamblador, era especialmente difícil de manejar. En retrospectiva, agradezco haberlo hecho en su momento, pues ahora conozco a la perfección la secuencia de arranque de x86, pero llegó el momento de cambiar.

Estos cambios resultaban tan sustanciales que, en enero de 2021, consideré que valía la pena hacer *borrón y cuenta nueva*. Por aquel entonces, el SO era un único repositorio, con lo que, el 1 de febrero de 2021, se cambió la rama principal, y se renombró la anterior a *old*. Allí sigue a día de hoy [58].

Teniendo el 1 de febrero un repositorio vacío, comenzó el desarrollo. Primero, un `printf`, luego una klibc simple (*Kernel Standard Library*, con estructuras de datos básicas), y poco a poco se fueron construyendo capas sobre capas de abstracción. Pocos meses después, en mayo de 2021, tras debatirme durante unas semanas, decidí que quería que este proyecto fuera mi trabajo de fin de grado, en lugar de dejarlo para más tarde (tesis doctoral, si la terminara haciendo), principalmente porque, aunque este trabajo tiene sus pinceladas de investigación, es más un proyecto de ingeniería que otra cosa.

En el plazo desde febrero a mayo no me dió tiempo de hacer mucho, tenía una funcionalidad muy básica del kernel, que no llegaba ni de cerca a tener la capacidad de cargar programas, ni IPC, ni nada. Muchas decisiones no estaban siquiera aún tomadas.

## 4.3. Decisiones fundamentales

El proyecto se denomina *The Strife Project*. Su nombre proviene de Empédocles, filósofo presocrático de la Grecia clásica, cuya metafísica (arjé, esencia del ser) estaba basada en elementos. De esta forma, existían dos fuerzas que formaban el universo: el amor (*love*), que unía los elementos, y el odio (*strife*), que los separaba [59].

Un sistema operativo se comienza y se abandona, jamás se termina. No está dentro de mis intenciones abandonarlo en el futuro, sino seguir trabajando en él en los próximos años de ser posible. Por esto, esta memoria retrata el estado de Strife en su versión TODO, con tag en GitHub `TFG`.

### 4.3.1. Focos de interés

Strife intenta ser tres cosas, por orden de prioridad:

- Seguro. Ver subsección 4.3.5.
- Bello. Cuando es posible y apropiado, Strife da soluciones simples a problemas complejos, aún si no son las más rápidas. Se ha hecho el mayor esfuerzo por escribir código muy legible.
- Modular. Ver siguiente subsección, 4.3.2.

### 4.3.2. Forma del proyecto

Strife se publica como una distribución. Como se dejó ver anteriormente, la organización de GitHub contiene un repositorio por proyecto, y existe uno especial, llamado, como es de esperar, *Strife*, que mediante tecnología de *git submodules* combina todos los otros proyectos, en sus versiones concretas, utilizando un Makefile específico para ello. La última versión generada por liberación continua está disponible aquí.

### 4.3.3. ¿Por qué x86?

Si el lector llega a este punto, es perfectamente consciente de cuál es la arquitectura objetivo de Strife: PowerPC. Es broma, x86-64. En la sección 4.2 se explicó que comenzó con el target IA-32 y posteriormente se hizo el cambio. Por esto, en esta subsección la pregunta a responder, más que por qué dicha ISA es el objetivo, por qué considerar x86. La respuesta es sencilla: es el por defecto. A fecha de hoy, y de las últimas décadas, cuando un individuo comienza su primer sistema operativo, lo hace en x86, pues le hace ilusión probarlo eventualmente en hardware real, y x86 es de lo que dispone. Este caso no es una excepción. Además, esta arquitectura tiene cierto valor histórico. De las que no están muertas (como podría ser el MOS 6502/6510), es la más antigua, y entre las líneas de los manuales de Intel y AMD, que aparecen multitud de veces en las referencias al final de la memoria, uno puede ver el pasado.

### 4.3.4. Elección del bootloader y protocolo de arranque

En la sección 4.2 se expresó la intención de abandonar JBoot como bootloader. Su sustituto es Limine en su versión 3, un proyecto de bootloader liderado por una de las personas que conozco del mundo de *hobby osdev* [60].

Limine soporta varios protocolos de arranque. A fecha de hoy, son: *stivale*, *stivale2*, *Linux*, y el protocolo de arranque propio de Limine. Strife está diseñado para *stivale2*, un protocolo muy simple que carga un ELF del kernel, y, en una sección del binario, detecta qué información desea recibir del bootloader [61]. Así, el kernel de Strife es compatible con todo bootloader que soporte el protocolo stivale2, no exclusivamente Limine. Entre estas opciones de recepción, Strife usa las siguientes:

- Text mode, para ejecutar el kernel en el modo gráfico de texto. En la subsección 4.3.6 se encuentra su justificación.
- Memory map, para recibir de la BIOS las regiones de memoria física existentes. Posteriormente se explicará cómo se utiliza.
- Módulos, para resolver el problema del bootstrapping, explicado en la sección 4.9.

### 4.3.5. Mecanismos generales de seguridad

Es foco prioritario del proyecto la seguridad. Por esto, Strife toma todas sus decisiones con la seguridad en mente, en lugar de la velocidad. Estas son las medidas más generales:

- Diseño del kernel resistente a ataques Meltdown y Spectre por KPTI (*Kernel page-table isolation*). Esta es la primera decisión comentada en la memoria que se ve reflejada como tal en el código. En un sistema operativo usual, el kernel se encuentra en el higher half (2.4.1), y marca sus páginas como globales. De esta forma, las páginas del modo supervisor están presentes en toda tabla de páginas, con lo que no es necesario hacer un cambio de contexto completo cuando ocurre una syscall, y todo va más rápido. En 2019, se publicaron dos artículos que definían ataques sobre los procesadores por bugs en el hardware. El primero, Meltdown, por culpa de la ejecución fuera de orden que utilizan las CPUs modernas [62]. El segundo, Spectre, por culpa de la predicción de saltos [63]. KPTI mitiga estos ataques haciendo que el kernel tenga su propio contexto, su propia tabla de páginas. Esto ralentiza mucho la mayoría de syscalls, a cambio de conseguir mitigar las vulnerabilidades.
- ASLR (*Address Space Layout Randomization*) obligatorio. Por este mecanismo, el cargador de programas monta el ejecutable y sus librerías en distintas regiones de memoria generadas aleatoriamente. ASLR está habilitado por defecto en todos los sistemas operativos. Sin embargo, en Linux es posible deshabilitarlo, y en Strife es obligatorio.
- SMAP (*Supervisor Memory Access Protection*) y SMEP (*Supervisor Memory Execute Protection*). Se trata de dos mecanismos de seguridad hardware. SMAP aparece en Broadwell (quinta generación), aunque SMEP existía desde Ivy Bridge. Así, únicamente se activa de estar en una arquitectura reciente, de otro modo se ignora, no

es mandatorio tener Broadwell para arrancar Strife. SMEP previene que el kernel, ejecutándose en ring 0, ejecute código que es accesible para el usuario. Es evidente que esto nunca debe suceder, pues el código accesible para el usuario viene de un ejecutable arbitrario. SMAP, en cambio, previene cualquier tipo de acceso a memoria de usuario desde ring 0. En ocasiones sí es necesario realizar un acceso, y por tanto la prevención solo actúa cuando la flag AC (*Alignment Check*) de la CPU está a 0. Cuando el kernel desee acceder, alza la flag, lee o escribe en la memoria de usuario, para después bajarla de nuevo.

- NX stack obligatorio. Similar a ASLR, es un mecanismo dado en la mayoría de SSOO por defecto, aunque es posible desactivarlo, mientras que en Strife es mandatorio. Consiste en marcar las páginas de la pila como no ejecutables, lo que mitiga varios ataques de *buffer overflow*.
- Full RELRO (*RELocation Read-Only*) mandatorio. A la hora de resolver las referencias a funciones de librerías enlazadas dinámicamente, el kernel puede optar por resolverlas todas de golpe, o conforme vaya siendo necesario (por medio de trampas page fault). Resolverlas todas de golpe aumenta el tiempo que tarda el ejecutable en cargarse, pero permite la posibilidad de aplicar el mecanismo conocido como Full RELRO. Las referencias a funciones dinámicas se encuentran en una sección propia del ejecutable, la GOT (*Global Offset Table*). Cuando Full RELRO está activo, se garantiza que esta sección estará en páginas independientes, con tal de poder ser marcadas como solo lectura una vez se hayan resuelto las referencias, lo que imposibilita ciertos ataques ROP (*Returned Oriented Programming*).

### 4.3.6. Gráficos en modo texto

El apartado gráfico no entra dentro de los intereses del proyecto. Durante el arranque, las BIOS modernas (desde 1989) ofrece un rango de interrupciones para VBA (*VESA BIOS Extensions*, INT 10h), que aporta una funcionalidad simple, pero completa, para obtener una lista reducida de modos gráficos VGA y poder cambiar entre ellos [64]. Los sistemas operativos más avanzados contienen sus propios drivers de vídeo para cada tarjeta gráfica y se comunican con ella así, lo que permite hacerlo dinámicamente y no durante el arranque, pero para los proyectos independientes es usual usar VBA, o GOP (*Graphics Output Protocol*) en el caso de UEFI [65]. De forma general existen dos modos gráficos:

- Modo vídeo. Es el usado por todos los sistemas operativos modernos. Se selecciona una resolución gráfica (ancho x alto) entre las soportadas, y la BIOS asigna un área de la memoria para un *framebuffer* de colores, según la profundidad de bits, aunque generalmente son 3 bytes por píxel.
- Modo texto. Retrocompatible con el IBM PC original, son modos gráficos en los que el framebuffer está constituido por pares `<caracter, color>`. Se selecciona por BIOS un modo según sus filas y columnas, y a partir de ahí se pueden escribir caracteres en pantalla.

En modo vídeo, es necesario rasterizar el texto si se quiere trabajar con una terminal y no un entorno gráfico completo, pues todo lo que se tienen son píxeles. Sin embargo, en modo texto, el programador del sistema tan solo selecciona la fuente y los caracteres. De

no seleccionar la fuente, se tomará una por defecto, que generalmente resulta ser la famosa *code page 437*, apreciable en la figura 4.1.

Figura 4.1: Página de códigos 437, IBM

### 4.3.7. Loader en userspace

Una de las partes más importantes del sistema operativo es el *loader*, o cargador de programas (no confundir con bootloader). Su función es la de, a partir de un ejecutable, comprender las estructuras que lo forman y llegar a tener una representación final en memoria principal, el proceso. En Strife, como en varios otros SO microkernel, el loader se encuentra en userspace, es decir, es un proceso independiente. Esto supone un reto técnico muy superior a tenerlo dentro del kernel, pues complica mucho el problema del bootstrapping (sección 4.9).

Es muy positivo sacar el loader del kernel. El objetivo de un microkernel es que los fallos de un servicio no afecten a otro, y un loader es muy propenso a fallos. No solo es difícil de escribir (por el relocation), sino que es, sobre todo, muy fácil hacerlo mal: hay muchos punteros a distintas secciones del ejecutable, y cada uno de ellos debe ser comprobado para no acceder a memoria indebida. En muchas ocasiones tienen offsets, con lo cual la suma ha de ser hecha comprobando si se ha producido acarreo, y lo mismo pasa con los índices, cuyas multiplicaciones han de computar de forma segura.

Como se dejó entrever en 4.1.4, el formato elegido para los ejecutables es ELF (*Executable and Linkable Format*), originario de UNIX 4, y que utilizan muchos UNIX-like, entre ellos GNU/Linux y los BSDs. ELF es un formato extremadamente bien diseñado, pero sus fundamentos se dejan para el siguiente capítulo.

### 4.3.8. Libre de POSIX

POSIX (*Portable Operating System Interface*) es un estándar del IEEE que define una interfaz y entorno de sistema operativo, así como una shell y un conjunto de utilidades, que sigue la línea de UNIX. Un sistema operativo es POSIX si cumple con el estándar, y entre ellos se podrían encontrar GNU/Linux, OpenBSD, FreeBSD, NetBSD, y macOS.

Muchos SSOO independientes se adhieren a POSIX, porque gran parte de las decisiones

de diseño están ya tomadas. Strife no acepta POSIX, y rediseñan grandes partes de la API, shell, y entorno. Toma ideas, por supuesto, como la existencia de una shell similar y algunas de las funciones estándar (`memcpy`, `memset`...), la sintaxis de rutas, la idea de puntos de montaje, y una semántica de `printf` similar, por poner algunos ejemplos.

### 4.3.9. Pilar: kernel O(1)

Una decisión fundamental que define el microkernel de Strife es que la gran mayoría de funciones son $\mathcal{O}(1)$. Después del arranque, la gran mayoría de syscalls, reservas de memoria, y pasos por el scheduler, de ciclar, tienen un número de iteraciones acotable por una constante.

Como consecuencia de esto, toda reserva de memoria que se hace en la tabla de páginas del kernel, ya sea para memoria física o virtual, se hace por medio de *free lists*, y, por tanto, con una granularidad fija de una página. Se explicará esto en más detalle en el siguiente capítulo, pues ya es algo más cercano al código.

La única excepción es la reserva de áreas en el espacio de direccionamiento del proceso, por ASLR, que, evidentemente, requiere páginas virtuales consecutivas para funcionar, uno no puede trocear la sección de código de un programa como le plazca.

Por ahora, lo relevante es saber que, a la hora de reservar memoria física, se tiene que hacer de exactamente una página. Cuando se reserva memoria virtual dentro del kernel, también se tiene que hacer de una página, pero sobre este mecanismo de reserva paginal se construyen *allocators* (funciones de reserva de memoria) para aportar una granularidad menor. En esta decisión, se sacrifica uso de memoria a cambio de velocidad del kernel.

**Aportación posiblemente original**: no he sido capaz de encontrar ningún otro microkernel (o, más improbable aún, kernel monolítico) cuyo funcionamiento esté mayoritariamente compuesto por funciones $\mathcal{O}(1)$. A la pregunta de si se trata verdaderamente de una idea buena o no se intentará responder en las conclusiones del trabajo.

## 4.4. IPC

### 4.4.1. Memoria compartida

Strife tiene dos formas de IPC: memoria compartida, y RPC. Memoria compartida es la más simple, y se explicará en esta subsección. La idea es trivial: un proceso A quiere compartir memoria con otro proceso B; para ello, el kernel mapea la región de memoria física que A quiere compartir en el espacio de direccionamiento de B. Debido a la reserva de memoria física $\mathcal{O}(1)$, esta compartición debe de ser exclusivamente de una página.

POSIX utiliza un mecanismo de memoria compartida *global*. No es complejo de entender: en POSIX, el proceso de compartir memoria entre procesos se lleva a cabo por medio de claves (*keys*), y estas están vinculadas a un fichero del sistema de archivos (en UNIX todo es un archivo, la traducción se hace mediante la función `ftok` [66]). Así, A debe hacerle saber a B de forma indirecta que dicho archivo es el que tiene la clave vinculada a la región de memoria a compartir. Esto es un mecanismo global porque utiliza un contexto global del sistema operativo: el sistema de archivos. Teóricamente, cualquier proceso que tenga permisos de lectura sobre ese archivo, puede obtener la clave y montar la región en su espacio de direccionamiento, realizando *snooping*. La solución a este problema no es elegante: más usuarios. Se definen más UIDs (*User Identifier*) de ser necesario. Este problema existe también, aunque en menor medida, con los pipes (las denominadas *named pipes*, existen alternativas no globales llamadas *nameless pipes*) y sockets. Windows hace algo muy similar con las funciones de la API `CreateFileMapping` y `MapViewOfFile` [67].

En Strife se busca un mecanismo de memoria compartida local, en el que el proceso de compartir memoria se realiza sin claves asignadas a archivos, de forma que verdaderamente solo A y B conozcan que están compartiendo memoria, y no sea posible el *snooping* desde fuera. Esto se hace por medio de syscalls específicas, y se explicarán en el siguiente capítulo.

Por ahora, es crítico conocer una cosa, que es la más importante sobre memoria compartida en Strife: en algún momento, A o B morirán. Cuando esto ocurra, su memoria será liberada, y, para cada, página se deberá comprobar si está compartida o no. En caso de que sí, es necesario algún contador que indique el número de referencias a dicha página física, para no liberarla antes de tiempo de forma global. Para mantener la localidad, es decir, no tener una estructura global dentro del kernel dedicada a esto (lo que conllevaría, de una forma u otra, el uso de claves), este contador debe guardarse en uno de los campos libres de las entradas de la tabla de página del kernel (la paginación en x86-64 deja algunos bits libres para el programador del sistema). Esto es absolutamente incompatible con compartir más de una página, puesto que este tipo de reservas conllevan por debajo otras independientes de memoria física, con sus consecuentes direcciones arbirtarias. Por todo esto, la memoria compartida solo es posible realizarla en tamaños de una página, y esto complica el proceso de compartir grandes regiones de bytes entre dos procesos: su tamaño máximo está acotado. Soluciones a este problema se encuentran en el capítulo siguiente, y las toman, por ejemplo, los servicios que componen la pila de almacenamiento.

**Aportación posiblemente original**: hasta donde soy consciente, no existe otro SO con esta mentalidad. De nuevo, como pasaba con el kernel $\mathcal{O}(1)$, al final se comentará si esta decisión resultó adecuada o no.

### 4.4.2. RPC

La decisión más importante de Strife radica en la elección de RPC como IPC base, algo inusual en los sistemas operativos. RPC se puede entender como un caso específico de paso de mensajes síncrono, en el cual los mensajes son muy cortos, y están dirigidos a una función en concreto (como pasaba con los IRQs y los ISRs).

Durante la rutina de syscall que tiene el kernel, la absoluta primera cosa que se hace es comprobar si el identificador de syscall es el de RPC, en cuyo caso todo el flujo de la rutina cambia y se ejecuta otro, escrito puramente en ensamblador, y que asegura jamás pasar por el scheduler. De forma un poco abstracta y sin llegar al nivel de instrucción, será explicada en detalle, qué hace paso a paso, en el capítulo siguiente.

Cuando el cliente realiza RPC, no queda bloqueado en el proceso. En su lugar, su flujo de ejecución *entra* dentro de la tarea remota. Esto hace que sea un mecanismo de IPC muy veloz, puesto que el salto se produce con un cambio de contexto limitado (se mantienen todos los registros del cliente, aunque se guardan para restaurarse luego, idea tomada de L4) y cero copias. Este uso de registros deja al programador un total de 4 registros para argumentos del procedimiento remoto, lo que son 32 bytes de datos.

Además, al contrario de lo que suele ocurrir en muchas implementaciones de paso de mensajes, se libera al servidor de la carga de abrir distintos threads para escuchar, y solo tiene que manejar las secciones críticas mediante cerrojos, semáforos, o directamente mediante el uso de algoritmos *lock-free*.

Por último, como es el cliente el que se está ejecutando dentro del servidor, y no el servidor en sí, el scheduler considera que es precisamente el cliente el que está en ejecución, no el servidor, y por tanto es este el que sube y baja en las colas de prioridad retroalimentadas. Esto hace que comportamientos costosos en tiempo del cliente no pasen desapercibidos y se culpe de ellos al servidor.

### 4.4.3. PSNS

Strife, además, tiene un mecanismo que se ha denominado PSNS (*Public Service Namespace*). Se trata de un servicio, el primero que se ejecuta durante el bootstrapping, que asigna nombres de hasta 8 caracteres a PIDs en ejecución. De esta forma, los servicios se *publican* al PSNS mediante RPC, dando un nombre para ser reconocidos por los clientes. Los clientes resuelven el nombre, obteniendo el PID en el proceso, y ya pueden comenzar la comunicación por medio de RPC. Para obtener el PID del PSNS existe una syscall específica, que se explica en el capítulo siguiente.

Este mecanismo evita la existencia de los archivos `.pid` muy usuales en entornos GNU/Linux, y a penas complica el kernel.

## 4.5. Scheduler propio

De igual forma que NT y XNU utilizan schedulers derivados de MLFQ, Strife sigue por esta línea. Como se explicó en 2.3.2, se trata de un scheduler con estructuras de datos extremadamente simples (lista enlazada), y esto lo hace perfecto para un microkernel. Como también se comentó, sufre de un problema de inanición a posibles tareas de alta prioridad, cuyo valor dinámico puede decrecer con el paso de las ráfagas.

Strife utiliza un scheduler de autoría propia: SMLFQ (*Sandwich MLFQ*). La idea es muy simple, se separa el scheduler en tres independientes:

- MLRR con $n_v$ colas con prioridad
- MLFQ con $n_r$ colas con prioridad retroalimentadas (dinámicas)
- MLRR con $n_b$ colas con prioridad

El pan de arriba es un scheduler VSRT (*Very Soft Real Time*), esto es, un scheduler simple de máxima prioridad que, por diseño e intencionadamente, permite la inanición de procesos con menor prioridad, sin llegar a implementar deadlines. El contenido del sandwich es un MLFQ usual (*regular*), configurable de distintas maneras según el sistema operativo. El pan de abajo es un MLRR simple para permitir procesos en segundo plano (*background*), asegurando que nunca quitarán recursos si hay procesos regulares disponibles.

Concretando para Strife, se utilizan los valores $n_v = n_b = 3$, $n_r = 10$. Por ahora, la implementación se mantiene simple y no busca ser óptima: un mismo valor de quantum para todas las colas, 10ms; dentro del MLFQ, se promocionan y democionan colas usando únicamente si, en la última ráfaga, se bloqueó o se terminó el quantum completo, respectivamente. Estos valores se suelen tomar por experimentación y no existe una aproximación teórica universal para obtenerlos, aunque existen modelos que intentan aproximarlo [68].

**Aportación original**: si bien SMLFQ es un algoritmo de scheduling simple, y hasta *naïve* por lo intuitivo, tomando NT y XNU soluciones muy parecidas, no había recibido un nombre hasta ahora.

## 4.6. Sistema de archivos propio

Un sistema operativo soporta uno o más sistemas de archivos. En el caso de la mayoría de SSOO, estos SSAA son grandes ideas de décadas pasadas (como UFS, el SA de UNIX), a veces alteradas con conceptos modernos de almacenamiento (familia ext*). Los SSOO gigantescos en los que trabajan miles de personas de forma diaria suelen desarrollar su propio sistema de archivos para plasmar correctamente la visión abstracta que tienen de un entorno asociado a la filosofía del sistema. Curiosamente, esto también pasa en los SSOO independientes, que son proyectos personales, puesto que su objetivo primario es aprender y experimentar. Todo el rango de sistemas operativos que hay en medio, especialmente aquellos dedicados a sistemas empotrados o que tienen una finalidad específica, utilizan ya existentes.

Strife aporta su granito de arena al mundo de los sistemas de archivos con StrifeFS (*Strife File System*), que toma ideas de UFS (en su forma de ext2), el de UNIX, pero también de NTFS, el de Windows. No soporta journaling.

Para empezar: los conceptos base, las capas más físicas del sistema de archivos, se las debo a UFS. La indexación multinivel de los datos, así como el concepto de inodos, son ideas que han sentado precedente, posiblemente llegando a categorizarse como *inmejorables*. Quizá se podría criticar la dispersión de los bloques en memoria secundaria (que ocurre en el resto y se puede solucionar por desfragmentación), y requerirían que el cabezal de lectura del disco trazara trayectorias innecesariamente amplias, pero con la llegada, ya bien instaurada en el mercado, de los SSD, este problema desaparece. En StrifeFS, los inodos se reservan de manera FIFO.

UFS tiene un problema fundamental: el control de acceso. En el sistema de archivos UNIX y derivados, existe una lista de usuarios (UIDs) y grupos (GIDs) a los que pueden pertenecer, definidas respectivamente en los archivos `/etc/passwd` y `/etc/group` [69]. Aquí surge un problema: los grupos están definidos de forma completamente independiente. Un usuario $U$ puede pertenecer a la vez a los grupos $G_1$ y $G_2$, aún si $G_1$ y $G_2$ estuvieran fuertemente relacionados. Este problema lo resuelve Microsoft con el directorio activo (*Active Directory*), donde los grupos pueden heredar de otros grupos, con lo cual $U$ podría pertenecer solo a $G_2$ y, en el proceso, indirectamente estar en $G_1$, con las ventajas que ello supone.

Sin embargo, existe un problema mucho más crítico: la propiedad. En el inodo de un archivo se encuentran dos campos: el UID y el GID del propietario. Esto funciona perfectamente para entornos simples, en los que pueden haber muchos usuarios y, por ejemplo, los grupos de *alumno* y *profesor*, que, en realidad, tendrían más sentido en los antiguos mainframes para los que fue ideado UNIX que en los sistemas actuales. En la actualidad, multiples usuarios son solo necesarios en servidores, y en workstations se usan como trucos para hacer divisiones complejas de permisos. Todo lector que haya administrado un servidor CentOS conoce el usuario `http` o `nginx`, y aquellos que hayan administrado Debian/Ubuntu conocen `www-data`.

Considérse el siguiente escenario: un directorio `notas` tiene como propietario el grupo `etsiit`, del que forma parte todo usuario que está en el grupo `alumnos` o en el grupo `profesores`. Esta restricción de consistencia ($alumnos \rightarrow etsiit \wedge profesores \rightarrow etsiit$) se hace de forma independiente al sistema y recae sobre la responsabilidad del administrador.

Supongamos que se busca que los `profesores` puedan escribir sobre `notas` mientras que `alumnos` tan solo puedan leer. Este escenario es imposible de representar en UFS de esta forma, y habría que recurrir a complicar la jerarquía de directorios.

La solución se denomina ACL (*Access Control Lists*), una semántica de permisos en la que se desmenuzan las políticas de acceso sobre un archivo, en lugar de tener un solo propietario. De esta forma sí es representable el escenario anterior. Una posible solución: sobre `notas` se especifica que todo usuario de `etsiit` puede leer, y todo usuario de `profesores` puede leer y escribir.

Los sistemas de archivos ext*, derivados de UFS, tienen una extensión, *acl*, que implementa este mecanismo, y está disponible en Linux [27]. Sin embargo, lo hace de forma subóptima: un ACL por archivo. Esto implicaría que, en el escenario anterior, se debería de aplicar la política de seguridad recursivamente, lo que complica el mantenimiento. Para solucionarlo, también se permiten aplicar políticas por defecto que son heredadas por todos los archivos nuevos del directorio, pero no los existentes: tan solo aplicará la política, que está contenida en cada archivo, a los nuevos que se creen. La solución quizá la haya descubierto el lector entre las líneas: ACLs por herencia. Por defecto, y a menos que se especifique lo contrario, todo archivo parte del ACL con identificador 0, que no indica otra cosa sino *ningún cambio* y, desde ese archivo, se recorre la jerarquía de directorios hacia arriba aplicando todos los cambios, teniendo más prioridad los más cercanos al archivo, y menos los de la raíz. Por ejemplo, `privado`, dentro de `notas`, puede tener un ACL que incluya una única regla: prohibida la lectura a `alumnos`. Subiendo la jerarquía, la prohibición de `privado` tiene más prioridad que el permiso de `notas`, con lo que la combinación de todos estos ACLs, hasta la raíz, sería la lista de control de acceso efectiva del archivo.

Esta solución, aunque más lenta (pese a acelerable por cachés), simplica mucho la jerarquía de directorios de todo el sistema operativo, y evitan la necesidad de usuarios arbitrarios y sin sentido intuitivo como `www-data` en Debian. Como nota, NTFS, el sistema de archivos de Windows, sí implementa ACLs, pero también sufre el problema de la herencia. Por defecto, un archivo hereda la herencia del padre, pero no puede realizar cambios sobre él que se sumen a los de los niveles superiores [26].

**Aportación original**: mezclar las mejores decisiones de UFS y NTFS, así como una propia, resulta en StrifeFS, un sistema de archivos seguro, local, y fácilmente mantenible en entornos tanto multiusuario como workstation.

## 4.7. El registro: un entorno innovador

Habiendo resuelto el problema de la granularidad de permisos del sistema de archivos en la sección anterior, hay uno que no debe pasar desapercibido: los permisos en tiempo de ejecución. En UNIX y derivados, como Linux, por defecto, la granularidad es binaria: se permite todo (*root*) o no se permite nada (usuario convencional). Esto es matizable: el popular programa *sudo* permite a un usuario ejecutar una orden como otro, según unas políticas definidas en el archivo de configuración (`/etc/sudoers`). En `sudoers`, se pueden especificar reglas del estilo: *el usuario jlxip puede ejecutar la orden* `/bin/programa -s *` *como root*. Si bien esto es mejor que un todo o nada estricto, es producto de un fallo fundamental en el diseño de UNIX. Este uso de wildcards (`*`), junto a el uso de binarios `SETUID` (que funcionan de forma muy parecida), son la mayor fuente de fallos de seguridad que resultan en escalación de privilegios en servidores. Como comentario, el programa *doas* de OpenBSD hace algo similar, pero no soporta el uso de wildcards en los comandos permitidos, lo que, aunque minimiza la superficie de ataques, reduce mucho la funcionalidad.

A raíz de esto nació SELinux (*Security-Enhanced Linux*), un módulo de Linux de principios de los 2000 que implementa políticas de control de acceso como MAC (*Mandatory Access Control*): un control de acceso en el que se restringen las acciones que puede realizar un sujeto (programa o usuario). Por ejemplo, una regla MAC podría prohibir a un programa escuchar en puertos TCP o UDP. SELinux es ampliamente utilizado en servidores, y es, hasta cierto punto, customizable. En las distribuciones que lo incluyen se suelen añadir un conjunto de reglas por defecto para permitir la funcionalidad más básica del sistema. SELinux, sin embargo, no es popular en distribuciones orientadas a workstations, como puede ser Ubuntu, y su inclusión es opcional, lo que hace, en este aspecto, a Ubuntu un sistema *insecure by default*.

En la sección anterior se comentó el uso en GNU/Linux de usuarios específicos para las aplicaciones, con tal de hacerles propietarios de los directorios que utilizan y que solo puedan acceder a ellos. SELinux extiende esta capacidad de restringir operaciones, haciendo que los programas concretos puedan realizar operaciones como las mencionadas anteriormente. Sin embargo, estas políticas generalmente se aplican a programas muy concretos, y el resto del sistema, por defecto, sigue funcionando como si el módulo no estuviera activo, lo que, de nuevo, por defecto, limita la granularidad. Por todo esto, por muy bueno que sea SELinux, no deja de ser un parche, una tirita sobre fallos de diseño fundamentales.

Strife propone un nuevo sistema de MAC: el *registro*. Siguiendo la línea del microkernel L3 de Liedtke, gestionar dentro del kernel los permisos de IPC es subóptimo y ralentiza todo el sistema, así que, en su lugar, se delega esta responsabilidad a los servicios individuales. Strife toma esta idea y, además, aporta el registro como un mecanismo global que pueden usar los servicios, de así desearlo, para tener de forma uniforme todos sus permisos en ellos. El identificador de procedimiento que recibe un servicio en cada IPC sería, así, comprobado en el registro al llegar, y cacheado en el servicio para minimizar los cambios de contexto.

Además, el registro no solo administra las capacidades de los programas, sino también de los usuarios. De esta forma, un usuario solo puede ejecutar un programa si los permisos que requiere son un subconjunto de los que posee. Este mecanismo se denomina DAC (*Discretionary Access Control*), en el sentido de que los permisos del usuario pasan a

los permisos del programa. Siguiendo esta línea, si el programa ejecutara otro, también se aplicaría una máscara. De esta forma, un usuario jamás puede ejecutar un programa, directa o indirectamente, que requiera privilegios que no tiene: los privilegios solo pueden reducirse en el camino.

El registro de Strife es un árbol al que pueden acceder los programas para depositar información y después obtenerla. Un punto clave fundamental es que se trata de una base de datos volátil, que únicamente se encuentra en RAM. El administrador puede configurar el registro por medio de scripts escritos por él que rellenan la *tabula rasa* con quién tiene permitido hacer qué, y no al revés: se trata de un sistema seguro por defecto, en el que, antes de la ejecución de estos scripts, nadie (excepto el superusuario y los programas que ejecuta) puede realizar ninguna acción.

Como nota adicional, el registro es una base de datos clave-valor arbitraria, no enfocada específicamente a estos servicios. Así, los programas lo utilizan también para obtener su configuración, homogeneizando el método de ajustes de los programas en una sola forma, sin distintos archivos de configuración con distintas sintaxis. Como consecuencia lógica, los valores de configuración se deben establecer por medio de scripts, y esto hace que toda la configuración del sistema sea Turing-completa, lo que permitiría al administrador del sistema reutilizar los mismos guiones en distintos computadores, y que configuren el entorno que detecten.

La comunicación con el registro se encuentra encapsulada en la librería estándar, con lo que el programador accede a la configuración de su programa de forma transparente, sin necesidad de leer archivos y parsearlos.

## 4.8. Resumen de los proyectos

Para tener una vista general del sistema operativo para las siguientes secciones, los proyectos que forman la distribución oficial de Strife, en la versión indicada al principio del capítulo (entrega del TFG), se encuentran mencionados en esta sección.

### 4.8.1. Servicios

- `loader`, el cargador de programas en userspace.
- `term`, primer servicio que carga el loader. Es el driver del modo texto, que controla el framebuffer que da la BIOS (pasando por el bootloader). Escribe en pantalla y maneja el cursor y el scrolling.
- `registry`, el registro tal y como fue explicado en 4.7.
- `keyboard`, driver que implementa la funcionalidad del teclado.

### 4.8.2. Pila de almacenamiento

- `PCI`, un driver simple para la conexión con periféricos y tarjetas de expansión del conocido bus estándar.
- `AHCI`, driver para conectar con dispositivos SATA, usado para ATAPI (CD de arranque).
- `ramblock`, servicio que implementa un dispositivo de bloques en RAM, simula ser un disco duro.
- `block`, servicio que abstrae los dispositivos de bloques (`AHCI` y `ramblock`), y los nombra por UUIDs.
- `ISO9660`, servicio que implementa el sistema de archivos ISO9660, usado universalmente por los CDs.
- `StrifeFS`, servicio que implementa el sistema de archivos explicado en 4.6.
- `VFS`, servicio que abstrae todos los sistemas de archivos por medio de puntos de montaje sobre una misma raíz.

### 4.8.3. Programas

- `init`, el último programa que ejecuta el kernel para completar el arranque. Se encarga de iniciar el resto.
- `splash`, un programa muy simple, el primero ejecutado por `init` en la distribución oficial, muestra el logo del proyecto Strife.
- `shell`, el intérprete de comandos por defecto, funciona de forma similar al de UNIX.

- `coreutils`, una pequeña colección de programas básicos para interactuar con el sistema (`ls`, `cp`, `cat`...).

### 4.8.4. Grafo de colaboración

Conociendo todos los programas que forman parte de la distribución oficial, resulta de especial interés tener una forma gráfica de ver cómo los procesos que en todo momento están activos se comunican entre sí, se encuentra en la figura 4.2. Todos ellos están conectados al registro, con lo que no aparece en el grafo para tener una representación visualmente más limpia.

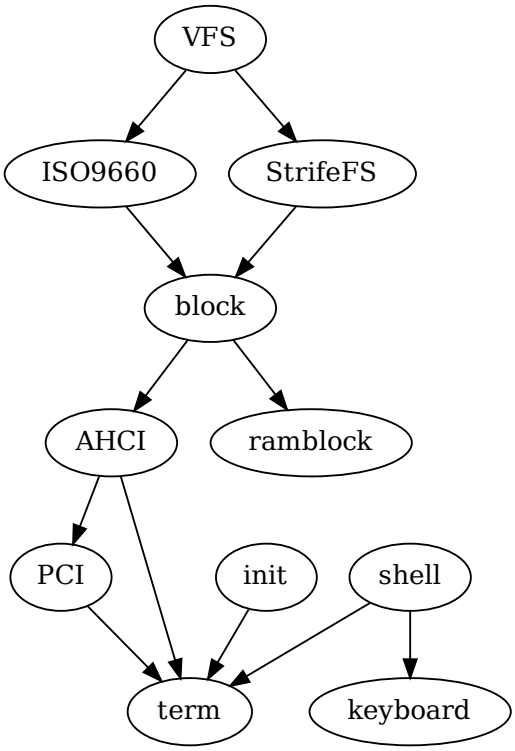



Figura 4.2: Grafo de colaboración de la distribución oficial de Strife

## 4.9. Bootstrapping

El bootstrapping es el proceso de emerger un entorno de la nada. Es un problema propio de los microkernels, porque, en el momento del arranque, el kernel no tiene capacidad para completar el inicio por sí solo. Cuando un kernel monolítico o híbrido arranca, cuenta con una pila de almacenamiento completa que le permite comenzar a cargar programas, y el primero que suelen cargar es `init`. En un microkernel, es necesario cargar los servicios que componen la pila para ello: ¿Cómo cargar el programa que permite acceder a los programas?. El problema se vuelve más complejo aún cuando el loader se saca del kernel y se hace un programa independiente: ¿Cómo cargar el programa que carga los programas?

Existen varias soluciones a la primera pregunta:

1. Repetir los drivers, usando solo las versiones del kernel durante el proceso de bootstrapping, y abandonándolas después. Esto es muy difícil de mantener, porque las dos versiones han de ser necesariamente distintas: el entorno del kernel es muy distinto al userspace. Se podrían abstraer las diferencias en la librería estándar, pero se complicaría más de lo debido dicho proyecto.
2. Propagar el problema hacia arriba. Se repiten los drivers, pero en el bootloader, donde necesariamente se encuentran ya repetidos. Muchos protocolos de arranque (entre ellos, stivale2) permiten enviar ficheros del sistema de archivos cargados durante el arranque al kernel, y, de esta forma, el kernel solo tiene que buscarlos.

El kernl de Strife utiliza la segunda opción. Para responer a la segunda pregunta (carga del loader), tan solo hay una respuesta: tener un cargador de programas simplificado dentro del kernel, no queda otra. Con tal de simplificar el código, se suele restringir el tipo de binario a cargar para que sea enlazado estáticamente. En el caso de ELF, esto quita mucha de la complejidad del proceso de parsing.

El proyecto Strife va un paso más allá y diseña un formato propio de ejecutable extremadamente simple para ser cargado con poco código. Se trata de SUS, y su estructura es la siguiente:

- Número mágico para ser reconocido: `7F555355` (4 bytes).
- Entry point del ejecutable (8 bytes).
- Binario plano. Se trata de un formato de ejecutable sin estructura, cuyas páginas en memoria son iguales a las páginas del binario. Es así como ha de escribirse el stage 1 de un bootloader.

En el repositorio del loader se aporta un programa, `elf2sus.py`, que realiza la transformación de un formato a otro. Este programa es llamado en el Makefile del subproyecto, de forma que el fichero resultante del proceso de compilación es `loader.sus`.

**Aportación original**: el tipo de archivo SUS es el más simple posible que contiene un cargador de programas para ser cargado como primer paso de bootstrapping de un microkernel. En realidad, se podría hacer directamente un binario plano, pero requeriría

que el punto de entrada estuviera en 0, y cargar una tarea con dirección base 0 es muy mala idea, puesto que la derreferencia del puntero nulo es válida.

# Capítulo 5

# Diseños detallados

## 5.1. Kernel

El kernel, siendo el proyecto más grande y complejo de todo el trabajo, toma muchas decisiones que son importantes de mencionar para entender cómo realiza las intenciones expresadas en el capítulo anterior. Se hará primero un repaso por lo más fundamental: los descriptores, tanto de segmentos como de interrupciones. Sin ellos, no existe sistema operativo. Tener una GDT válida permite la gestión de la memoria, y este será el siguiente tema a atacar. Después, comienza el userspace: syscalls, IPC, scheduler, y, para finalizar, cómo funciona el equivalente a `main` en el kernel.

### 5.1.1. GDT

En un x86, en todo momento debe de haber presente una GDT válida. Al arranque, BIOS proporciona una, que varía entre fabricantes, con lo que no se puede confiar en ella; en general, hay que desconfiar de las BIOS, la gran mayoría tienen bugs en unas partes u otras. Por esto, el bootloader, en su stage 1, hace el cambio a una GDT conocida, con segmentos especificados (generalmente muy simples), y, conforme va cambiando el modo del procesador, la va alterando. Cuando se llega al kernel, este debe establecer la suya propia, para ser independiente del bootloader.

Strife inicializa la segmentación con tan solo cuatro selectores generales:

- Kernel code, para el registro CS (*Code Segment*). Ring 0.
- Kernel data, para el resto de registros de segmentación. Ring 0.
- User code, también para CS cuando se está en userspace. Ring 3.
- User data, análogamente. Ring 3.

Después de estos cuatro, se añade un selector por cada core para las TSS, como se explicó en 2.5.4.

### 5.1.2. IDT e ISRs

La IDT se configura para que todo vector de interrupción tenga asociado un ISR por defecto, que todo lo que hace es causar un kernel panic. Para los vectores que sí se implementan, se introducen los punteros a sus ISRs correspondientes. Todo ISR comienza con un código en ensamblador, con tal de reorganizar los registros y el marco de pila que establece el procesador para traducirlos en parámetros a la parte del ISR escrito en el lenguajde alto nivel.

La mayoría de excepciones implementadas concluyen en que debe terminarse el proceso que la ha causado, si se ha producido en ring 3, o en kernel panic, si ha sido en ring 0. Esto incluye, por ejemplificar, con #GP (error de protección general), #DE (división por cero), #DF (double fault), y #UD (opcode inexistente). La única excepción es page fault, que se utiliza para varias cosas:

- Aumentar el tamaño de la pila. Cuando la página objetivo que ha causado la excepción está una por encima de la última página de pila reservada, y no supera el límite, se procede asignando más pila al proceso.
- Si el page fault fue causado en ring 3, en la mitad inferior de la memoria, su causa es una violación de la protección de la página, y se ha producido en el primer byte de esta, entonces se trata de una petición implícita de vuelta de RPC. Este ingenioso mecanismo se explicará en 5.1.6.

### 5.1.3. PMM

El PMM (*Physical Memory Manager*) de un sistema operativo es la sección del kernel que se dedica a reservar y liberar la memoria física. Como se comentó en 4.3.9, este conjunto de rutinas son $\mathcal{O}(1)$, porque están implementadas en forma de *free list*, también llamada *free block chain*: una estructura de datos en la que las regiones libres de memoria se unen formando una lista enlazada. Considere un escenario en el que existen dos regiones libres de memoria separadas:

- 1MB $\rightarrow$ 16MB.
- 18MB $\rightarrow$ 64MB.

En este caso, una free list representa el espacio manteniendo un puntero a la primera página física libre. En ella, se escriben `15MB`, el tamaño de esta región, y `18MB`, el inicio de la siguiente. Después, en los primeros bytes de la altura de `18MB`, se escriben `46MB`, el tamaño, y `0`, puesto que no hay una región siguiente. Este proceso de inicialización es técnicamente $\mathcal{O}(n)$, pero muy rara vez suele haber más de 4 o 5 regiones distintas de memoria física, con lo que es informalmente aproximable a $\mathcal{O}(1)$, y además solo se realiza este procedimiento durante el arranque.

Con la free list inicializada con la memoria usable, la reserva es trivial, y $\mathcal{O}(1)$:

**Algoritmo 1:** Reserva de página en free list

```
Entrada: Puntero a la primera región libre (ptr)
Salida: Página libre, nueva primera región libre
Resultado: Modificación del estado de la memoria
inicio
    si ptr = 0 entonces
        devolver ERROR
    fin
    libre ⟵ ptr
    si ptr.tamaño ≤ TAMAÑO_PÁGINA entonces
        ptr' ⟵ ptr.siguiente
    fin
    en otro caso
        ptr.tamaño ⟵ ptr.tamaño - TAMAÑO_PÁGINA
        ptr' ⟵ ptr
    fin
    devolver libre, ptr'
fin
```

Si la reserva devuelve *ERROR*, entonces no hay memoria física disponible. En este caso, si es posible, se vuelve error. De no ser posible, por necesitar memoria en el kernel, debe existir un mecanismo de liberación forzada. Se usa el mecanismo *timber*, que consiste en, de forma Greedy, terminar el proceso no-esencial que más memoria esté ocupando [70].

El algoritmo de liberación de páginas es trivial, y consiste en inicializar esta página liberada: establecer el tamaño de la región (una página, 4KBs), y apuntar al siguiente. Después, se establece esta dirección como nuevo puntero, y será la próxima en usarse. Esto hace que el algoritmo sea LIFO. Se podría modificar para hacerlo FIFO, aunque hacerlo LIFO aporta una pequeña protección extra contra *use-after free*.

Estas funciones están implementadas bajo los nombres de `PMM::alloc` y `PMM::free`. También existe otra, `PMM::calloc`, que rellena la página física con ceros antes de devolverla, para evitar posibles filtraciones de datos.

### 5.1.4. VMM

Siendo capaz el kernel de reservar memorias físicas, y después de haber habilitado la paginación, se necesita hacer lo mismo en el espacio de direccionamiento virtual. En el caso de Strife, para mitigar los ataques Meltdown y Spectre, el VMM se separa en dos distintos: uno privado, que gestiona las páginas privadas del contexto del kernel, y otro público, que trabaja con páginas marcadas como globales en todo contexto.

El privado no es otra cosa que la llamada a las funciones de reserva o liberación del PMM. En la tabla de páginas del kernel, la mitad inferior de la memoria está *mapeada* uno-a-uno; es decir, las direcciones son las mismas en memoria física y virtual. Por esto, no es necesario realizar ningún paso más.

El público es más de lo mismo. Se hace la llama a la reserva del PMM, y después se

*mapea* la dirección física $F$ a la dirección virtual $V = F + \text{HIGHER\_HALF}$, marcándola en el proceso como global y no-ejecutable. De igual forma que en el PMM, existe `VMM::calloc` para limpiar las páginas.

### 5.1.5. Syscalls

Antes de continuar con las explicaciones sobre el kernel, ha llegado el momento de enumerar las syscalls. Existen 19 TODO de ellas, y algunas requieren privilegios de ejecución que se comprueban en el kernel desde el registro. En realidad, es al revés: el registro envía al kernel qué usuarios y procesos pueden realizar qué acciones antes de que estas ocurran. Esto se debe al descubrimiento de seL4 que afirma que un microkernel jamás debe ejecutar una rutina del userspace, como se mencionó en 3.3.3.

En la tabla 5.1 se encuentran las diferentes syscalls que existen en el microkernel de Strife con información sobre cada una.

Tabla 5.1: Syscalls de Strife

| ID | Nombre | Área | Permisos necesarios | Argumentos |
|---|---|---|---|---|
| 0 | `EXIT` | General | Ninguno | 1 |
| 1 | `MORE_HEAP` | General | Ninguno | 1 |
| 2 | `MMAP` | General | Ninguno | 2 |
| 3 | `BACK_FROM_LOADER` | Loader | Solo loader | 3 |
| 4 | `MAKE_PROCESS` | Loader | Solo loader | 0 |
| 5 | `ASLR_GET` | Loader | Solo loader | 3 |
| 6 | `MAP_IN` | Loader | Solo loader | 3 |
| 7 | `FIND_PSNS` | General | Ninguno | 1 |
| 8 | `HALT` | General | Ninguno | 0 |
| 9 | `RPC` | RPC | Ninguno | 2-6 |
| 10 | `ENABLE_RPC` | RPC | Ninguno | 1 |
| 11 | `RPC_MORE_STACKS` | RPC | Solo kernel | 1 |
| 12 | `SM_MAKE` | Memoria compartida | Ninguno | 0 |
| 13 | `SM_ALLOW` | Memoria compartida | Ninguno | 2 |
| 14 | `SM_REQUEST` | Memoria compartida | Ninguno | 2 |
| 15 | `SM_MAP` | Memoria compartida | Ninguno | 1 |
| 16 | `ALLOW_IO` | General | `IO_ALLOWED` | 0 |
| 17 | `ALLOW_PHYS` | General | `PHYS_ALLOWED` | 0 |
| 18 | `GET_PHYS` | General | `PHYS_ALLOWED` | 1 |
| 19 | `MAP_PHYS` | General | `PHYS_ALLOWED` | 3 |
| 20 | `LOAD_EXE` | Tareas | Ninguno | 2 |
| 21 | `LAST_LOADER_ERROR` | Tareas | Ninguno | 0 |
| 22 | `GET_KILL_REASON` | Tareas | Ninguno | 1 |
| 23 | `GET_EXIT_VALUE` | Tareas | Ninguno | 1 |

Se procede a explicar las que están marcadas como área general, las otras se explicarán en sus correspondientes secciones.

- `EXIT` es la syscall que permite que un proceso termine. Recibe un argumento, su valor de retorno. Al ocurrir, se libera la memoria del proceso, y su valor de retorno pasa

al proceso padre.

- `MORE_HEAP` reserva más pila. Tiene un parámetro que indica el número de páginas adicionales a reservar. De ocurrir la reserva, devuelve el puntero a la primera página reservada. De fallar, devuelve *nullptr* (0).
- `MMAP` toma 2 parámetros, y hace una reserva de $a$ páginas en una dirección arbitraria dada por ASLR, asignando en el proceso los permisos `b`, que son un mapa de flags:
  - `MMAP_RW` permite la escritura sobre las páginas.
  - `MMAP_EXEC` permite la ejecución.
  - `MMAP_RWX = MMAP_RW | MMAP_EXEC`
- `FIND_PSNS` devuelve el PID, conocido por el kernel, del servicio PSNS, para que los procesos puedan hacerle RPC y resolver nombres. Más sobre esto en 5.2.
- `HALT` se usa para parar el flujo de ejecución del programa sin terminarlo. Al ocurrir, se libera la pila y el proceso queda en estado bloqueado indefinidamente. La intención es que se use para quedar esperando a recibir un RPC. Posible causa de confusión: el proceso ya debe estar preparado para recibir RPCs (mediante `ENABLE_RPC`, explicado posteriormente en 5.1.6), `HALT` tan solo detiene el flujo principal del programa para que no termine.
- `ALLOW_IO` se usa para pedirle al kernel que habilite la flag `IOPL=3` en el contexto del proceso. Esta flag en realidad son dos, pues es un valor numérico de dos bits (de 0 a 3), que indica al procesador el anillo de protección máximo que es capaz de ejecutar las instrucciones `in` y `out` para comunicarse con el hardware mediante PIO. La usa, por ejemplo, el driver de PCI. Realizar esta syscall requiere que exista una clave `IO_ALLOWED` en el registro para este programa y usuario.
- `ALLOW_PHYS` se usa para pedirle al kernel que acepte las siguientes syscalls. Realizar esta requiere que exista otra clave `PHYS_ALLOWED` en el registro para este programa y usuario.
- `GET_PHYS` obtiene la dirección física de una página del espacio de direccionamiento virtual del proceso. La usa, por ejemplo, el driver de AHCI para comunicarse con el hardware por MMIO. Require haber llamado antes a `ALLOW_PHYS`.
- `MAP_PHYS` también requiere `ALLOW_PHYS`. Mapea, no reserva, una región consecutiva de $b$ páginas físicas, comenzando en la dirección física $a$, en la memoria virtual del proceso, con el mapa de flags $c$:
  - `MAP_PHYS_RW` para permitir la escritura sobre esta página.
  - `MAP_PHYS_DONT_CACHE` para evitar que la página se mantenga en caché (necesario para MMIO).

### 5.1.6. RPC en detalle

Esta es la sección más compleja del proyecto, la explicaré lo mejor que pueda. Para empezar, las syscalls. IPC por RPC en Strife usa tres:

- `RPC` realiza una llamada a procedimiento remoto de forma síncrona. Toma 6 argumentos, aunque solo 2 de ellos son necesarios desde el punto de vista del kernel: el PID destino, y el RPID (*Remote Procedure ID*), es decir, un identificador, que el cliente debe conocer de antemano, que representa a la función que intenta llamar. El resto de argumentos son los parámetros que recibirá el procedimiento remoto (el por qué son 4 en seguida).
- `ENABLE_RPC` efectúa el mecanismo descrito arriba: permite que otros procesos entren dentro de él. Además, recibe un parámetro: el entry point de RPC, como si fuera un ISR. Debe estar escrito en ensamblador.
- `RPC_MORE_STACKS` se explicará en breves instantes.

Sabiendo esto, se puede definir el flujo de comunicación del cliente y del servidor. Primero, el del cliente:

1. De no conocer el PID remoto, se efectúa un RPC primero a PSNS (utilizando `FIND_PSNS` de no haber cacheado el PID aún). Podría ya conocerse para evitar públicamente este valor, por ejemplo, si el proceso con el que se quiere comunicar es hijo del cliente.
2. Realiza el RPC con los parámetros dados.

El del servidor es ligeramente más complejo:

1. Primero, define el entry point para las llamadas. En 5.7.4 se explicará cómo la stdlib encapsula este comportamiento.
2. Después, habilita la entrada de RPC mediante la syscall `ENABLE_RPC`, pasando como argumento el puntero a la función de entrada.
3. De desearlo, realiza un RPC al PSNS para publicar su nombre.
4. Finalmente, efectúa la syscall `HALT` para detener el flujo de ejecución del main.

Todo proceso tiene en su PCB un vector de 256 pilas para su uso en RPC. El código en ensamblador recorre esta lista buscando una pila libre y, de estar todas las presentes en uso (o directamente no haber ninguna presente), se intenta reservar una nueva. Esta reserva sí se hace desde el código en C++, y resultaría tan complejo hacerlo en ensamblador que no es factible. Por esto, si hacen falta más pilas, toda expectativa de hacer la ida rápido se desvanece, y el código efectúa una syscall especial, `RPC_MORE_STACKS`, únicamente disponible para el kernel en esta situación tan concreta. Si durante el intento de reservar resulta que todas las 256 pilas están en uso simultáneamente, el cliente se bloquea y queda en la cola de un semáforo hasta que una de ellas se libere. Este es el absoluto peor de los casos; en el más usual, hay una pila libre, se toma, y se marca como en uso, haciendo que durante toda la rutina no sea necesario cambiar a la tabla de páginas del kernel, lo que hace que la ida de RPC tome únicamente un cambio de contexto, aún con la protección de Meltdown presente.

Un RPC está compuesto por dos pasos: una ida y una vuelta. Un concepto fundamental en Strife es el *return ticket* (billete de vuelta): una estructura en la pila del servidor que

guarda la información sobre cómo volver. Aquí surge un problema: ¿Cómo guardar la información necesaria para volver sin la posibilidad de que el servidor escriba sobre ella? Resultaría catastrófico: permitiría a cualquier programa saltar a cualquier dirección de cualquier proceso sin comprobación de privilegios. Para mantener la localidad solo hay una forma: hacer la página solo accesible por el kernel. De esta forma, es necesaria una página únicamente para el billete de vuelta, cuyos privilegios de acceso cambian antes de terminar la ida.

A la vuelta, es necesario comprobar que la página en la que se encuentra este billete de vuelta es, efectivamente, del kernel, para evitar falsas vueltas de RPC que nunca han tenido una ida. Esta comprobación resultaría costosa en tiempo y difícil de programar: teniendo la página virtual, habría que recorrer los cuatro niveles de paginación para comprobar que efectivamente existe y además es propiedad del kernel. Además, en ensamblador, porque todo lo relativo a RPC está en dicho lenguaje. Existe una solución mucho más elegante: causar un page fault desde el servidor que indique la vuelta. Este page fault se disparará al intentar leer bytes de esta página, pues es propiedad del kernel, y el ISR comprobará, como se presagió en 5.1.2, si efectivamente se cumplen las condiciones que identifican a un return ticket: ser una página en la mitad inferior de la memoria que a su vez es únicamente accesible por el kernel. De darse el caso, el kernel hará el salto a la rutina de vuelta de RPC, y se habrá evitado hacer la comprobación por software: la habrá hecho la MMU.

Las bases de RPC han quedado, con esto, explicadas. El código que las realiza es difícil de entender y lleno de trucos de ensamblador que no se consideran de relevancia para explicar aquí. Las rutinas de ida y vuelta de RPC contienen el código más complejo que he escrito en mi vida, y estoy orgulloso de ellas. Por si el lector quisiera echarles un ojo, se encuentran aquí [71].

### 5.1.7. Memoria compartida en detalle

El procedimiento de compartir memoria se hace parcialmente en el kernel, parcialmente fuera. El PCB tiene un campo con un puntero a una página específica para memoria compartida. En ella, se almacenan cuartetos `<SMID, kptr, tptr, allowed>`; es decir, la página virtual en el espacio de direccionamiento del proceso que la pidió, así como el proceso con el que quiere compartirla. Por esto, solo es posible compartir una página con un proceso, y una tarea solo puede compartir 128 páginas simultáneamente (son los cuartetos que caben en una página, 4*8 bytes sobre 4096).

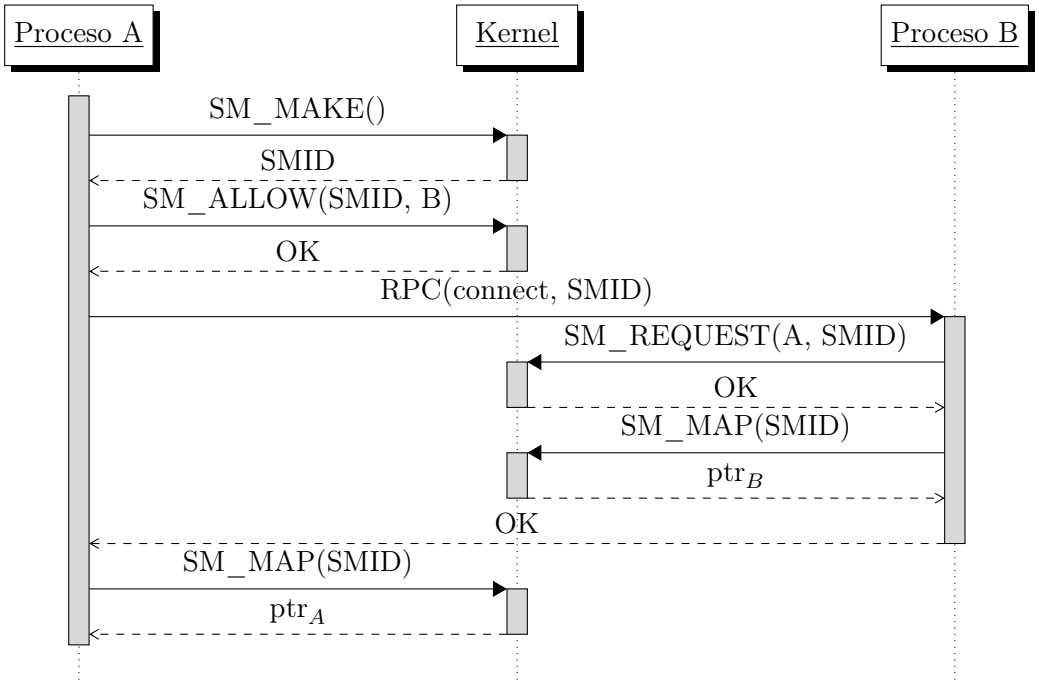
Se definen cuatro syscalls:

- `SM_MAKE` crea una nueva página compartida. No recibe argumentos, y devuelve un SMID (*Shared Memory Identifier*) de 64 bits generado aleatoriamente. Se espera que la ejecute el cliente.
- `SM_ALLOW` recibe como argumentos el SMID y un nuevo valor para el campo `allowed` al PID dado.
- `SM_REQUEST` es ejecutada por el servidor, y tiene como argumentos el PID del cliente y el el SMID. Esta syscall copia el par de memoria compartida del cliente al PCB del servidor, permitiendo su acceso.

- `SM_MAP`, ejecutada tanto por el cliente como el servidor, toma el SMID y, finalmente, monta la página compartida en una página, dada por ASLR, del espacio de direccionamiento virtual del proceso que efectúa la syscall.

El mecanismo de `SM_REQUEST` está hecho para que el servidor pida expresamente que quiere esa página en su lista (recordemos, finita) de regiones de memoria compartida, y evitar un posible inundamiento malicioso de SMIDs. En estas syscalls, el lector se habrá dado cuenta de que en ningún momento el servidor conoce el SMID del cliente. Esta información ha de ser pasada de forma externa, y se recomienda el uso de un RPC para ello. La mayoría de servicios aportados en la distribución oficial contienen un procedimiento público (con su RPID asociado) denominado `connect` que sirve para esto mismo.

Habiendo explicado todo esto, la semántica de memoria compartida es la siguiente:




Esto rellena los campos poco a poco. Después de `SM_MAKE`, uno de los índices aparece con su campo SMID distinto de cero, y `kptr` (puntero del kernel, se podría entender como el físico), se reserva. Tras `SM_ALLOW`, se rellena `allowed`, y solo tras `SM_MAP` se establece el valor de `tptr` (task pointer). Unificar `SM_MAKE` y `SM_MAP` haría que el campo `kptr` dejara de ser necesario, pues siempre habría un `tptr` cuya página física es computable. Sin embargo, tendría la desventaja de que la reserva de memoria virtual solo se podría llevar a cabo antes de saber si ha habido un error en la comunicación por RPC, y los servicios perderían esta libertad.

### 5.1.8. Los procesos

Como se viene diciendo desde el principio de este trabajo, la estructura que representa un proceso es el Process Control Block. En Strife, el PCB forma una clase Task, y está

compuesto por una serie de campos, muchos de los cuales han mencionado ya. Son los siguientes:

- Punto de entrada de RPC. Se define al principio de la estructura para tener un offset nulo, lo cual facilita su acceso desde ensamblador.
- Flags de RPC. Se encuentra en el segundo campo por la misma razón, y contiene el valor que debe tomar el registro RFLAGS a la entrada de un RPC.
- 257 punteros a pilas de RPC, todos inicializados a 1. Se dijeron que eran 256 anteriormente, pero se mantiene un último cuyo valor es siempre 1 para detectar cuando no queden más pilas disponibles para asignar. Cada entrada puede tener tres valores:
  - 0: esta pila en concreto existe, pero está en uso.
  - 1: esta pila no existe, no está reservada.
  - Cualquier otro valor: puntero a la pila, en el espacio de direccionamiento virtual del proceso.
- El puntero al PML4, administrado por una clase `Paging` que encapsula la complejidad de la paginación multinivel.
- Estado salvaguardado. A la hora de interrumpirse el proceso, se guarda el estado aquí. Está formado por todos los registros de propósito general, así como RFLAGS.
- Valores de RIP (contador de programa) y RSP (puntero de pila) para ser restaurados cuando el proceso vuelva a ejecutarse.
- `heapBottom` y `stackTop` representan, respectivamente, cuáles son las páginas que representan el estado mayor asignado actualmente de la heap y la pila. En un punto dado de la ejecución del programa, podría tenerse una página de heap y dos de pila. En este caso, `heapBottom` apuntaría a la base de la pila + tamaño de página, y `stackTop` a la base del stack - 2 * tamaño de página.
- `prog`, `heap`, y `stack` son las bases ASLR para estas regiones de memoria. `prog` es el lugar de la primera página del programa, `heap` es la primera página de la heap (esté reservada o no), y `stack` es la primera página después de la pila que no forma parte de ella; esto es, el valor inicial de RSP (recordemos que un PUSH en x86 primero decrementa RSP y después escribe).
- `maxHeapBottom` y `maxStackTop` precomputan cuáles son las páginas máxima y mínima de la heap y el stack respectivamente, con tal de hacer la comprobación rápidamente en caso de page fault.
- Una instancia de `ASLR` que contiene las regiones libres de memoria virtual consecutiva para poder hacer reservas aleatorias.
- Un puntero a la página de SMIDs, inicializado a `nullptr` y asignado en caso de ser necesario.
- Un booleano que indica si se permiten las syscalls de memoria física. El por defecto es false, y se establece a true si la syscall `ALLOW_PHYS` concluyó exitosamente.

De esta forma, el tamaño del PCB es grande, pero esto no es un problema, porque toda instancia se almacena en su propia página de memoria privada del kernel.

Si el lector se fija, este PCB no tiene información relativa al scheduler. Esto se debe a que existe otra estructura que engloba a Task, y se denomina SchedulerTask. Tiene los siguientes campos:

- Una copia del puntero al PML4 como primer elemento, para ser accedido desde ensamblador (rutina RPC) de forma fácil cuando no es posible derrefernciar la tarea por poder usar exclusivamente memoria pública.
- PID del proceso padre. Los servicios de bootstrapping tienen este valor a 0.
- El código de error que devolvió el loader la última vez que se hizo `LOAD_EXE`. Este es el valor que devuelve `LAST_LOADER_ERROR`.
- Lista enlazada de hijos. Se trata de cuartetos `<pid, kr, ret, waiting>`, donde:
  - `pid` es el PID del proceso hijo.
  - `kr`, kill reason, es la razón por la que el kernel ha matado a este proceso. Los valores se encuentran en la tabla 5.2.
  - `ret` es el valor de retorno del proceso cuando ha terminado.
  - `waiting` indica si el padre está actualmente esperando a este proceso.
- Puntero a la página que contiene el PCB, memoria privada del kernel.
- Un booleano indicando si la última ráfaga terminó consumiendo todo el quantum (falso) o si acabó en espera de entrada/salida (verdadero).
- La prioridad actual.
- El tipo de prioridad, es decir, en cuál de los tres pisos del sandwich se sitúa.

Las syscalls `GET_KILL_REASON` y `GET_EXIT_VALUE` iteran la lista de hijos y devuelven respectivamente los valores `kr` y `ret` del PID que coincide con el argumento.

Tabla 5.2: Razones de terminación de un proceso por el kernel

| Código | Nombre | Descripción |
|---|---|---|
| 0 | `INEXSTING` | PID no reconocido |
| 1 | `NONE` | El proceso terminó de forma natural |
| 2 | `SEGFAULT` | Acceso no permitido a memoria |
| 3 | `LOADER_SYSCALL` | Intentó ejecutar una syscall del loader |
| 4 | `UNKNOWN_SYSCALL` | Efectuó una syscall inexistente |
| 5 | `RPC_BAD_PID` | Se efectuó un RPC a un PID inválido |
| 6 | `KERNEL_SYSCALL` | Intentó ejecutarse una syscall del kernel |
| 7 | `PHYS_NOT_ALLOWED` | Se intentó ejecutar una syscall de memoria física sin permiso |

Existe un array global público de punteros a SchedulerTask, `tasks`, cuyo índice es el PID. Así es como se resuelven PIDs en procesos. Como no existe la posibilidad de reservar memoria consecutiva dentro del kernel, esta estructura se mantiene en el último GB de la memoria virtual.

Estos punteros a SchedulerTask están implementados sobre una clase `ProtPtr`, en la que se usa el bit menos significativo del puntero (como está alineado con la página, ha de ser cero) como cerrojo. Así, el PCB solo es derreferenciable por exclusión mutua y no pueden existir condiciones de carrera relativas, sobre todo, a la muerte del proceso.

Los PIDs en Strife son enteros de 16 bits, así que como máximo pueden existir 65535 procesos en ejecución, dado que el PID 0 está reservado. Se asignan de manera FIFO.

Existe un mecanismo especial utilizado en RPC para obtener el puntero al PCB desde ensamblador de forma rápida, `generalTask`. Al inicio del arranque, el kernel reserva una página en la memoria virtual pública del kernel que queda sin su traducción física. En la tabla de páginas de cada proceso, esta página virtual se encuentra mapeada a la página física que contiene el PCB de la tarea correspondiente. Así, en todo contexto, derreferenciar este puntero obtendrá el PCB únicamente por medio de memoria pública, sin tener que pasar a la tabla de páginas del kernel. La página virtual de `generalTask` es la única que no está marcada como global en toda la región higher half.

El scheduler no tiene mucha complicación práctica. Su justificación teórica se aportó en 4.5, y, en realidad, lo único que el lector debe conocer es que existe una función, `schedule()`, que obtiene la próxima tarea que debe ejecutarse y llama al dispatcher.

Resulta de interés en esta sección discutir este último punto; en realidad, existen dos dispatchers: `dispatch` y `dispatchSaving`, ambas funciones de la clase Task. El primero está contenido dentro del segundo por medio de *fallthrough* de instrucciones. `dispatch` es el intuitivo: realiza el cambio de contexto a la tarea, restaurando los registros y cambiando la tabla de páginas y selectores en el proceso. `dispatchSaving`, además, primero guarda el flujo de ejecución del kernel en variables globales para ser restaurado luego. Este mecanismo se usa en el proceso de bootstrapping, cuando el microkernel tiene una serie de programas que ejecutar secuencialmente. También se usa cuando se comprueban algunos permisos en el registro, pero eso se explicará en 5.4.

### 5.1.9. Drivers necesarios

Un microkernel, para sorpresa de nadie, tiene drivers dentro. Hay algunos que se usan tan extremadamente a menudo (en cada syscall o cada quantum) que ralentizaría excesivamente el sistema sacarlos. Hay otros, además, que son imposibles de extraer pues aportan funcionalidades básicas de un x86 moderno. Por suerte, resultan pocos todos estos casos combinados, y en Strife solo se destacan dos: ACPI y APIC.

APIC se explicó en 2.5.4. Es imposible de extraer pues el kernel necesita realizar IPIs para sincronizar los cores, y, además, implementa las funcionalidades de LAPIC e IOAPIC con tal de configurar las interrupciones externas (hardware), incluyendo el *LAPIC timer* para configurar interrupciones periódicas para los quanta.

ACPI (*Advanced Configuration Power Interface*) es un estándar para descubrir y configurar componentes, así como para gestionar la energía. Es la versión moderna de APM (*Advanced Power Management*). Su interés reside en que la BIOS aporta ciertas tablas ACPI al bootloader, cuyos punteros después se envían al kernel.

Entre estas tablas, se recibe en el kernel el puntero a la RSDP (*Root System Description Pointer*), que, además de tener un campo para diferenciar entre ACPI 1.0 y 2.0 (o en adelante), contiene el puntero a RSDT (*Root System Description Table*, en el caso de 32 bits y ACPI 1.0) o a la XSDT (*Extended System Description Table*, para 64 bits y ACPI 2.0+) [72].

RSDT/XSDT son tablas que contienen, de nuevo, punteros a un número arbitrario de otras tablas, que se utilizan para propósitos concretos. En el caso de Strife, la única que resulta de interés para Strife es la MADT (*Multiple APIC Description Pointer*), que contiene información extensa sobre cada core de la CPU, y, especialmente, el APIC ID y la base MMIO de la, o las, IOAPIC, necesaria para la configuración de interrupciones hardware.

### 5.1.10. SMP

La arquitectura x86 implementa multiprocesamiento simétrico, SMP. Algunos modelos también soportan NUMA para casos concretos, pero no es relevante para el proyecto. Saber que existen varios procesadores sobre una misma memoria da lugar a que todo el kernel tenga que ser diseñado con mecanismos de cerrojos para garantizar la exclusión mutua de secciones críticas. El algoritmo de cerrojo (*Spinlock*) del kernel de Strife es *test, test and set*, y utiliza la instrucción propia de x86 `pause` para evitar el consumo de energía excesivo producto de la espera ocupada [73].

Todos los cores comienzan en modo real. Uno de ellos, elegido por la placa base, se denomina el BSP (*Bootstrap Processor*), también llamado CPU0, y es el único que comienza la ejecución en `0x7C00`. Después de los cambios de modo del bootloader y el arranque del kernel, leyendo la MADT y configurando APIC se puede aplicar potencia al resto de cores (denominados APs, *Application Processors*) mediante SIPIs (*Startup IPIs*), arrancándolos en modo real en una dirección arbitraria, no necesariamente la estándar de IBM. Estos cores tendrían que recorrer la misma ruta que el BSP, teniendo cuidado con la exclusión mutua, hasta llegar al kernel y, a partir de ahí, coordinarse para repartirse las tareas a realizar. Esto resulta muy tedioso para el kernel; se podría entender que el kernel tiene que implementar su propio bootloader.

Para evitar esta tarea, algunos protocolos de arranque, entre ellos, stivale2, promocionan este problema hacia el bootloader, y él se encarga de inicializar a todos. El bootloader pasa al kernel el puntero de una estructura, `stivale2_smp_info`, con un campo `goto_address` que es sondeado (*polling*) por los APs hasta que tiene un valor no nulo, en cuyo caso se salta a esa dirección y se completa el arranque de los cores, de manera transparente al kernel [61].

### 5.1.11. ¿Cómo es el main de un kernel?

Esta sección puede satisfacer la curiosidad del lector: *¿Cómo será el `main()` de un kernel?*. En el caso de Strife, se llama `kmain()`, y se encuentra en el archivo `kernel.cpp` [74]. `kmain` recibe como parámetro un puntero a la estructura que le pasa Limine al kernel con tal de cumplir la especificación de stivale2. Aquí está un resumen superficial de qué

pasos lleva a cabo:

1. Primero, y antes de nada, `kmain` mueve el mapa de memoria que recibe del bootloader a un lugar seguro. Esto se debe a que las regiones marcadas como *usables* dentro del mapa pueden contener (o no, la especificación es ambigua, con lo que depende del bootloader) los propios datos pasados al kernel, con lo que es preciso moverlos a un lugar seguro antes de empezar a sobrescribir la memoria.
2. Seguidamente, se hace lo mismo, pero con los módulos; es decir, los ejecutables cargados por el bootloader para completar el proceso de bootstrapping.
3. Se inicializan los descriptores. Primero, la GDT, y, después, la IDT.
4. Se inicializa el PMM usando el mapa de memoria.
5. Se inicializa la tabla de páginas del kernel, también usando el mapa de memoria.
6. En este punto, se parsean las tablas ACPI.
7. Se inicializa la LAPIC del BSP y la IOAPIC.
8. Se finaliza la inicialización de la memoria con las páginas marcadas por el mapa de memoria como *bootloader reclaimable*.
9. Se inicializan los allocators del kernel para tener una menor granularidad de reservas de memoria dinámica.
10. Se preparan las pilas que usarán el resto de cores del sistema.
11. Se crean y cargan las estructuras TSS, una para cada core.
12. Se inicializa el scheduler.
13. Se habilitan las syscalls.
14. Se habilitan los mecanismos de seguridad SMEP y SMAP, de estar soportados.
15. Comienza el bootstrapping del loader, con la carga del binario SUS.
16. Comienza todo el bootstrapping del userspace, empezando por `PSNS` y terminando con `VFS`.
17. Se desbloquean el resto de cores, y todos llaman a `schedule()`.

## 5.2. Flujo PSNS

El Public Service Namespace es un mecanismo muy simple para asignar nombres a procesos concretos, como se dijo en 4.4.3. Utiliza tan solo una syscall, `FIND_PSNS`, que obtiene del kernel el PID del servicio para comunicarse con él por RPC y realizar las siguientes resoluciones de nombres.

Existen dos procedimientos públicamente disponibles para comunicarse con el servicio PSNS:

- `PUBLISH` publica el nombre, pasado por parámetro, del proceso que hace. De existir ya el nombre, simplemente devuelve *false*. Sería una terrible idea sobrescribir el PID, pues esto permitiría que un proceso malicioso actuara como *Man in the Middle* entre los clientes y el servidor, dándole, además, la libertad de crashear el cliente, que podría ser un servicio imprescindible.
- `RESOLVE` resuelve el nombre, también pasado por parámetro. De no existir, devuelve 0, que es un PID que se garantiza que jamás será asignado.

Se comentó en 4.4.2 que los parámetros de un RPC son a lo sumo 4 enteros de tamaño de registro (`size_t`, 64 bits). Para poder enviar el nombre, se limitan los nombres públicos a 8 caracteres, y el nombre pasa por un proceso de *marshalling* (cambio de forma) para transformarlo en un entero. Si el nombre es de menos de 8 caracteres, los restantes se dejan a cero. Esta funcionalidad está encapsulada en la librería estándar, con lo que el programador no tiene que preocuparse de esto.

El PSNS, dejando de lado su importancia en todo el entorno del sistema operativo, es un programa muy sencillo. La traducción de nombre a PID se realiza por medio de una tabla hash, cuya implementación está en la STL de la librería estándar (explicado al final de este capítulo).

## 5.3. Loader

### 5.3.1. ¿Cómo se carga un programa?

Todo cargador de programas sigue la misma secuencia de pasos para cargar un ejecutable en memoria:

1. Parsear el archivo y copiar las regiones en memoria tal y como pide la estructura.
2. Analizar las funciones exportadas, en caso de ser una librería dinámica.
3. Listar las librerías dinámicas que el programa necesita y cargarlas en el espacio de direccionamiento.
4. *Relocation*, resolver las referencias dinámicas con las funciones requeridas.
5. Cerrar los permisos de las regiones de memoria.

### 5.3.2. El formato ELF

Un ELF se divide de dos formas: cabeceras de programa y secciones.

Para empezar, las cabeceras de programa (PHDRs, *Program Headers*) define las secciones de memoria virtual y cómo corresponden con los contenidos del fichero. Contienen instrucciones para su carga, y permisos. Las instrucciones pueden ser, por ejemplo, *esto es memoria no inicializada*, o *esta es una metacabecera y debe ignorarse a la hora de cargar*. Los permisos suelen definri si las regiones permiten la escritura y/o la ejecución del código. En el caso de ELF-64, una cabecera de programa está definida de la siguiente forma [75]:

- `p_type`, las instrucciones de carga, con valores como `PT_LOAD` (*cárgame*) o `PT_NOTE` (*ignorar en la carga*).
- `p_flags`, con los permisos.
- `p_offset`, dónde en el fichero comienza esta región.
- `p_vaddr`, dónde en memoria comienza esta región.
- `p_paddr`, obsoleto.
- `p_filesz`, cuánto ocupa esta región dentro del archivo.
- `p_memsz`, cuánto ocupa esta región cuando está en memoria.
- `p_align`, restricción de alineamiento para la región.

Conociendo los campos de un PHDR, se entiende cómo funcionaría una región de memoria no inicializada (famosamente conocida como `.bss`): `p_filesz` < `p_memsz`.

Cuando el binario está compilado con ASLR en mente, todas las direcciones virtuales son relativas a la base 0, pero el código está compilado para ser PIC (*Position Independent Code*), utilizando exclusivamente instrucciones con referencias relativas al contador de programa.

Las secciones dividen el ejecutable de forma que se entienda qué contiene cada región, no solo cómo cargarlas. Es necesario analizar las secciones para encontrar aquellas que contienen las referencias dinámicas y los procedimientos exportados, como `.dynsym` y `.rela.dyn`. Con tal de identificar las secciones, cada una contiene un campo `sh_name` con un offset a otras secciones específicas que contienen solo nombres, como `.shstrtab` o `.dynstr`.

Existe una sección, `.dynamic`, que contiene la lista de librerías dinámicas necesarias para el ejecutable, identificadas por nombre (por ejemplo, `libstd.so`), que el sistema operativo debe cargar de forma independiente y posar sobre el espacio de direccionamiento.

Generalmente, las librerías no se copian, porque sería un gasto innecesario de memoria física, sino que se referencian. Esta trampa se denomina CoW (*Copy on Write*), donde las páginas se marcan como solo lectura, y solo se copian si ocurre un page fault de escritura sobre ellas.

Las relocations en un ELF, o sea, el enlazamiento dinámico de funciones requeridas en tiempo de ejecución, tienen varios tipos que el procesador elige dependiendo del contexto. No varían muchos entre ellos, y algunos son muy poco comunes, con lo que Strife únicamente implementa aquellos que han aparecido empíricamente. Ejemplos pueden ser `R_X86_64_JUMP_SLOT` o `R_X86_64_GLOB_DAT`.

### 5.3.3. Flujo de carga de programas

Hablando estrictamente de Strife, los programas son cargados por petición del kernel al loader. En el proceso de bootstrapping, esto se realiza numerosas veces para arrancar secuencialmente los servicios imprescindibles. La semántica consiste en que el loader tiene una región de su espacio de direccionamiento virtual asignado para los ejecutables que entran desde el kernel, una dirección estática, que no cambia con cada carga. El kernel, al arrancar el loader y pedirle cargar la stdlib, que es su primera tarea, establece el registro RDI, correspondiente al primer argumento de una función según la ABI Sys-V, el puntero a esta región y el tamaño de la stdlib. Cuando el kernel ha completado su carga, efectúa la syscall `BACK_FROM_LOADER`, que recibe como parámetros:

- El PID del proceso creado.
- Un código de error, definidos en la tabla 5.3.
- El punto de entrada de la tarea.

El loader está programado de tal forma que no se reserve el PID hasta que se haya comprobado que no existen errores en el binario. Por esto, de haber un código de error distinto de `NONE`, el PID devuelto al loader es nulo.

Tras esto, el loader queda en estado bloqueado hasta que el kernel quiera cargar el siguiente ejecutable. En ese momento, la syscall devolverá el tamaño del nuevo ejecutable situado en la región estática.

Para efectuar la carga, el loader necesita comunicarse con el kernel, y esto lo hace mediante tres syscalls:

- `MAKE_PROCESS`, que crea un PCB vacío (con una tabla de páginas que solo contiene las referencias al kernel y `generalTask`), pero con un PID reservado.
- `ASLR_GET`, que se comunica con el objeto ASLR del PCB. Esto se hace por medio de regiones, identificadas numéricamente, que representan cada región. La syscall reibe el PID del proceso a consultar, el identificador de la región, y el número de páginas a reservar, en caso de que la región no exista anteriormente. Devuelve la dirección en el espacio virtual del nuevo proceso.
- `MAP_IN`, que toma una página de la memoria del loader y la sitúa en un punto concreto de la memoria del nuevo proceso.

Tabla 5.3: Códigos de error devueltos por el loader

| Código | Nombre | Descripción |
|---|---|---|
| 0 | `NONE` | La carga se completó con éxito |
| 1 | `NO_MEMORY` | No se ha podido realizar alguna reserva de memoria |
| 2 | `NOT_ELF` | El archivo recibido no es un ELF |
| 3 | `NOT_64` | El ELF no usa direcciones de 64 bits |
| 4 | `NOT_LE` | El ELF usa Big-Endian en lugar de Little-Endian |
| 5 | `BAD_ARCH` | La arquitectura objetivo no es x86 |
| 6 | `INVALID_OFFSET` | Uno de los offsets del ELF es inválido |
| 7 | `NO_PHDRS` | El ELF no contiene PHDRs |
| 8 | `NO_SECTIONS` | El ELF no define secciones |
| 9 | `NO_SHSTRTAB` | El ELF carece de la sección de nombres de secciones |
| 10 | `NO_DYNSTR` | El ELF no tiene la sección de nombres dinámicos |
| 11 | `UNSUPPORTED_RELOCATION` | El mecanismo de relocation no está implementado |
| 12 | `FAILED_RELOCATION` | No existe la función referenciada dinámicamente |

Una tarea puede iniciar este mecanismo mediante la syscall `LOAD_EXE`, pasándole al kernel un puntero donde está cargado el ejecutable así como su tamaño. No se permite la comunicación directa con el loader, pues no debe requerir permisos, y el cliente podría llegar a congelar el cargador de programas (o saturar su memoria virtual) mediante el envío de páginas vacías.

## 5.4. Registro

### 5.4.1. Jerarquía

En la sección 4.7 se presentó el mecanismo de registro para la configuración de servicios. Aquí se describe su implementación y comunicación. De forma intuitiva, solo existe una forma de hacer este tipo de implementación, sobre todo teniendo en cuenta la volatilidad de los datos: un árbol de tablas hash.

La representación multinivel se separa usando el caracter `/`. La raíz se denomina como `/`. Bajo la raíz existen tres directorios principales, que representan las configuraciones de distintos servicios. Estos son: `self`, `kernel` y `data`.

El objetivo de `self` y `kernel` es guardar privilegios:

- En el caso de `self`, los permisos son respecto al propio registro: ¿Qué claves pueden editar?
- En el de `kernel`, corresponde a los permisos de syscalls protegidas.

En ambos casos, las hojas del árbol representan los permisos:

- `self` tiene como hojas un caracter seguido de una ruta. El caracter puede ser `R`, si se permite su lectura, o `W`, si además se permite su lectura. Por ejemplo, `W/` indica que se tiene control total del registro, y `R/kernel` indica que se pueden leer los permisos relativos al kernel.
- `kernel` tiene dos hojas posibles: `IO_ALLOWED`, para permitir la syscall `ALLOW_IO`, y `PHYS_ALLOWED`, para permitir la syscall `ALLOW_PHYS`.

Tienen a su vez tres subdirectorios: `p`, que afecta a los programas, `g`, a los grupos, y `u`, a los usuarios.

- `p` contiene rutas de programas, separadas por `/`. Como ejemplo, la existencia de `/kernel/p/'/bin/programa'/IO_ALLOWED` permite que el ejecutable situado en `/bin/programa` pueda realizar la syscall `ALLOW_IO`.
- `g` contiene identificadores numéricos de grupos del sistema de archivos raíz. Si existe `/self/g/admins/'W/'` , entonces los usuarios que estén en el grupo `admins` pueden escribir en todo el registro.
- `u` es exactamente lo mismo que `g` pero para usuarios.

Los permisos se combinan de la forma $P \wedge (G \vee U)$. Esto es, el programa necesita tener dicho permiso, y, además, o bien el usuario, o bien uno de los grupos de los que sea parte, tiene que tenerlo también.

`/data` es especial: almacena datos arbitrarios de los programas, que pueden ser permisos si ellos así lo definen. Sus subdirectorios son sus propias rutas. El programa `/bin/editor` tiene permisos de lectura y escritura implícitamente sobre `/data/'/bin/editor'`.

Cuando el kernel recibe una syscall `ALLOW_IO` o `ALLOW_PHYS`, se ve obligado a consultar al registro. Esto se hace por medio de `dispatchSaving`, dejando al otro proceso en estado bloqueado hasta conocer la respuesta.

### 5.4.2. Procedimientos del registro

Los procedimientos públicos que ofrece el registro para su comunicación con PCI son los siguientes:

**TODO: Pendiente de definir**

## 5.5. Pila de memoria secundaria

### 5.5.1. PCI

PCI (*Peripheral Component Interconnect*) es el driver de Strife que implementa la comunicación con el bus homónimo. PCI Express, su sucesor, es retrocompatible con PCI, con lo que el driver sirve para ambos.

No es objetivo de este trabajo entrar a explicar las profundidades del bus, pero sí hay ciertas cosas básicas que hay que conocer: la comunicación se hace por medio de PIO, donde se utilizan dos direcciones de puertos: `0xCF8`, la de configuración, y `0xCFC`, la de datos. Sobre el puerto de configuración se escriben las denominadas direcciones PCI, que son *rutas* de 32 bits que identifican los periféricos conectados. Esto lo hacen por su número de bus (8 bits), su número de dispositivo (5 bits), y su número de función (3 bits). Además, los ocho bits menos significativos de la dirección representan el offset dentro del descriptor del dispositivo [76].

Al leer el offset 0 de un dispositivo, se obtiene una cabecera. Las hay de distintos tipos, pero la más importante es la cabecera `0x0`. En ella, existen campos como *Vendor ID* y *Device ID* con los cuales se puede identificar la procedencia del dispositivo. Otros campos como *class code* y *subclass* sirven para identificar qué tipo de dispositivo es exactamente.

El driver de PCI de Strife tiene como principal objetivo hacer el *probe* de dispositivos; esto es, detectar qué hay conectado a la placa base. La forma más simple de hacerlo es por fuerza bruta. Se prueban todas las direcciones cambiando el número de bus y el número de dispositivo, dejando el número de función a cero. Si el dispositivo no existe, el controlador PCI establece el campo *Vendor ID* a `0xFFFF`, que está reservado para este propósito [76]. Así, hay que hacer fuerza bruta de $2^{8+5} = 8192$ opciones, que, en realidad, toman bastante poco tiempo.

Habiendo identificado qué pares `<bus, dispositivo>` son válidos, se comprueban las funciones (se podrían entender como *subdispositivos*). Para cada una de ellas, se obtiene la clase y subclase que identifican el tipo. El driver guarda toda esta información para otros drivers que la necesiten.

Generalmente, estos dispositivos con los que se quiere comunicar van a generar interrupciones hardware. En tiempos de la PIC, se recibían de forma simple. Con la APIC, se complica, y la forma más sencilla de recibir interrupciones PCI por APIC es usando uno de dos mecanismos muy similares: MSI (*Message Signaled Interrupts*), y MSI-X, sobre los cuales no se entrará a describir en detalle. Sin embargo, sepa el lector que esta es la razón por la cual el driver de acceso a disco implementado en Strife es AHCI y no IDE, puesto que IDE es antiguo (década de los 80) y no soporta MSI/MSI-X, mientras que AHCI funciona sobre PCI Express, y todo dispositivo de dicho bus está obligado por el estándar a soportar uno de los dos.

Para comunicarse con los dispositivos se utilizan las denominadas BAR (*Base Address*), y cada dispositivo tiene hasta seis de ellas. Pueden ser PIO o MMIO, dependiendo del dispositivo.

Los procedimientos públicos del driver se especifican en la tabla 5.4.

Tabla 5.4: Procedimientos públicos de PCI

| RPID | Nombre | Permiso | Argumentos |
|---|---|---|---|
| 0 | `GET_DEVICE` | `PCI_LIST` | 3 |
| 1 | `GET_BAR` | `PCI_FULL` | 2 |
| 2 | `DO_MSI` | `PCI_FULL` | 2 |
| 3 | `ENABLE_MMIO` | `PCI_FULL` | 1 |
| 4 | `BECOME_BUSMASTER` | `PCI_FULL` | 1 |

Una explicación un poco más en detalle:

- `GET_DEVICE` es un procedimiento con los parámetros: clase, subclase, índice. Devuelve la dirección PCI del dispositivo que pasa el filtro de pertener a la clase y subclase dadas. El índice se utiliza para iterar, las tareas que quieran buscar dispositivos PCI de un tipo concreto comienzan el índice en cero y continúan hasta que se devuelva un error `BAD_DEVICE`.
- `DO_MSI` tiene dos parámetros: una dirección PCI y un vector de la IOAPIC. Enlaza las interrupciones del dispositivo al vector dado, usando MSI o MSI-X, según corresponda.
- `GET_BAR` recibe una dirección PCI y un índice $i$, devolviendo BAR$i$ para el dispositivo dado.
- `ENABLE_MMIO` habilita la captura de direcciones en el bus de la placa base, para poder comunicarse con el driver en una región de memoria.
- `BECOME_BUSMASTER` vuelve al dispositivo *maestro del bus*, es decir, por simplificar, selecciona el dispositivo.

### 5.5.2. AHCI

AHCI (*Advanced Host Controller Interface*) es el driver que se comunica con este estándar. Sirve para comunicarse con discos duros (SATA, *Serial ATA*) y unidades de disco (ATAPI, *ATA Packet Interface*). La implementación actual del driver tan solo implementa ATAPI, pues no estaba dentro de los objetivos del proyecto implementar un mecanismo de instalación del sistema operativo.

Una placa base que soporta AHCI contiene un chip, el controlador de AHCI, también denominado HBA (*Host Bust Adapter*), conectado por PCI Express. Se identifica como un dispositivo de clase `0x01` (almacenamiento masivo) y subclase `0x06` (SATA). Mediante comunicación con el driver de PCI, este servicio enumera estos dispositivos, y para cada uno:

- Obtiene el ABAR (*AHCI Base Memory Register*), una dirección física con la que comunicarse con el dispositivo, reconocida como el BAR5.
- Hace maestro del bus al HBA y habilita MMIO.

- Mapea la región física (ABAR) en su memoria virtual; en concreto, son necesarias dos páginas.
- Habilita el dispositivo y enumera sus puertos.

Todo HBA tiene una serie de puertos disponibles; como máximo, 32. En concreto, como se intenta soportar únicamente ATAPI, se buscan aquellos que, en su campo de firma (*signature*) tengan el valor `0xeb140101`, que los identifica como tal [77].

Habiendo reconocido la unidad de disco, se comprueba si tiene un disco dentro; de ser el caso, recuerda a este puerto del dispositivo como una unidad ATAPI con disco, para acceder a ella posteriormente.

Como ATAPI es para CDs, y los CDs son solo lectura, el driver tan solo implementa la lectura. Existen dos formas de acceder a un dispositivo masivo de datos: PIO (*Programmed Input/Output*, no confundir con *Port Input/Output*) y DMA (*Direct Memory Access*).

En la primera, la CPU es responsable de hacer la copia de cada byte, o pequeño múltiplo de bytes (como mucho, el tamaño de registro, 8 bytes). Esto mantiene a la CPU ocupada, y, además, hace que el procedimiento sea mucho más lento, porque los datos tienen que pasar del disco a la CPU y de la CPU a la memoria.

En la segunda, la CPU tan solo manda las órdenes de copia, y el dispositivo toma el control del bus de datos para copiarlos a la memoria principal sin pasar por el procesador, lo que acelera mucho el proceso. Existen dos formas de DMA: DMA guiado por IRQs, y *DMA polling*. En el primero, el proceso queda bloqueado a la espera de una interrupción hardware de finalización, y mientras tanto puede realizar otras tareas. En la implementación de Strife actual, se realiza DMA polling: se comprueba por espera ocupada si ha finalizado la copia. DMA polling es síncrono, pero, aún así, es mucho más rápido que PIO, porque la ruta que toman los datos es mucho menor.

Estas órdenes que se envían al HBA se forman como FIS (*Frame Information Structure*), una estructura propia de AHCI. Estas órdenes se reparten entre otras estructuras más, como el *Command Header* y la *Command Table*, pero el objetivo de todas es uno: enviar órdenes.

En el caso concreto de ATAPI, las órdenes que se envían son comandos SCSI (*Small Computer Systems Interface*) encapsulados en órdenes ATA, haciendo una gran mezcla de estándares y chips en el proceso, sobre la cual no se va a entrar en detalle.

Existen tres formas de referirse a una localización en memoria secundaria:

- Como una dirección lineal, de la misma forma que si fuera memoria primaria.
- Como un LBA (*Logical Block Addressing*), el número de sector.
- Muy antiguamente, con CHS (*Cylinder-Head-Sector*), una terna `<cabeza, cilindro, sector>`.

ATAPI define los sectores como bloques de 2KBs. El envío de datos a otros procesos se hace por memoria compartida, que es de una página, 4KBs. Esto implica que, a lo sumo,

se pueden leer dos sectores de golpe, 4KBs. De necesitar más, se necesitan hacer sendas peticiones secuenciales.

Lo que sí es preciso comentar son los procedimientos públicos, y están en la tabla 5.5.

Tabla 5.5: Procedimientos públicos de AHCI

| RPID | Nombre | Permiso | Argumentos |
|---|---|---|---|
| 0 | `GET_ATAPIS` | `AHCI_LIST` | 0 |
| 1 | `CONNECT` | `AHCI_CONNECT` | 1 |
| 2 | `READ_ATAPI` | `AHCI_ATAPI_READ` | 3 |

Sus definiciones son simples:

- `GET_ATAPIS` devuelve el número de discos ATAPI conectados al HBA.
- `CONNECT` es la primera ocurrencia del mecanismo comentado en 5.1.7. Recibe un SMID y prepara la memoria compartida, devuelve `true` si todo ha ido bien, `false` en cualquier otro caso.
- `READ_ATAPI` realiza la lectura sobre la página de la memoria compartida. Recibe un identificador de disco (desde 0, menor que el número devuelto por `GET_ATAPIS`), una dirección lineal de inico, y el número de sectores a leer: 1 o 2.

### 5.5.3. ramblock

`ramblock` es un servicio que implementa un dispositivo de bloques en RAM de tamaño infinito. Tiene sectores del tamaño de página. Cuando se pide una que no existe, se devuelve la página llena de ceros. Cuando se escribe en ella, se guarda la página en una tabla hash, usando como clave el LBA.

Los procedimientos públicos se encuentran en la tabla 5.6.

Tabla 5.6: Procedimientos públicos de ramblock

| RPID | Nombre | Permiso | Argumentos |
|---|---|---|---|
| 0 | `CONNECT` | `RAMBLOCK_READ` | 1 |
| 1 | `READ` | `RAMBLOCK_READ` | 1 |
| 2 | `WRITE` | `RAMBLOCK_WRITE` | 1 |

`CONNECT` funciona de la forma usual. `READ` y `WRITE` ambos reciben un LBA, y leen el sector o lo copian respectivamente.

### 5.5.4. block

`block` es la abstracción sobre dispositivos de bloques. Funciona asignando UUIDs (*Universally Unique Identifier*) a los dispositivos, esto es, enteros de 128 bits.

Por ser tan solo una capa de abstracción, todo lo que hace es ofrecer una interfaz homogénea para el acceso a los dispositivos, que posteriormente traduce en otros RPCs a los drivers. Los procedimientos están en la tabla 5.7.

Tabla 5.7: Procedimientos públicos de block

| RPID | Nombre | Permiso | Argumentos |
|---|---|---|---|
| 0 | `CONNECT` | `BLOCK_LIST` | 1 |
| 1 | `LIST_DEVICES` | `BLOCK_LIST` | 1 |
| 2 | `READ` | `BLOCK_READ` | 4 |
| 3 | `WRITE` | `BLOCK_WRITE` | 4 |

- `CONNECT` funciona de la forma usual.
- `LIST_DEVICES` recibe un parámetro, la página. Escribe en la memoria compartida la lista de UUIDs presentes, de forma paginada. Cuando el cliente que está enumerando detecta el UUID `00000000-0000-0000-000000000000`, la secuencia ha concluido. Además, devuelve cuántos UUIDs se han escrito en la memoria.
- `READ`, bueno, lee. Cuatro parámetros: primeros 64 bits del UUID, últimos 64 bits, dirección lineal de inicio, y número de bytes a leer (como máximo el tamaño de página, 4096). Devuelve `true` o `false` según la lectura haya sido correcta o no.
- `WRITE` tiene una interfaz análoga a `READ`, solo cambia la acción que realiza.

### 5.5.5. ISO9660

ISO9660 es un sistema de archivos solo-lectura usado por CDs. Al ser solo lectura, está diseñado para ser muy simple, fácil de leer por reproductores de sonido y DVD. El sistema de archivos comienza en el sector 16 (`0x10`). Contiene el denominado PVD (*Primary Volume Descriptor*), que se podría considerar el superbloque. Contiene ciertas propiedades sobre el sistema de archivos, entre ellas:

- Número de sectores escritos.
- Tamaño del LBA (generalmente 2048 bytes).
- Entrada del directorio raíz.

Una entrada de directorio define un archivo, como si fuera un inodo. Tiene varios campos, los más significativos son:

- LBA del *extent*, es decir, los propios contenidos del fichero (que puede ser a su vez una secuencia de directorios).
- El tamaño del extent.
- Flags. Entre ellas, la más importante es la que define si esta entrada representa a un archivo o a otro directorio.

- Fecha y hora de grabación.
- Longitud del nombre del archivo, seguido del nombre en sí.

Solo con estas dos estructuras se puede recorrer la jerarquía de directorios de todo el sistema de archivos. Como punto a destacar, los valores numéricos que aparecen (LBAs, tamaño...) están en both-endian; es decir, little-endian seguido de big-endian, para que cualquier dispositivo pueda leerlos sin realizar el cambio de endianess.

El directorio raíz tiene nombre de longitud uno, y es solo un byte nulo. El resto de directorios tienen dos entradas como mínimo: la que representa al directorio actual (equivalente a `.`), y la que representa al padre (equivalente a `..`).

Existen dos extensiones a ISO9660 que se suelen usar, *Rock Ridge* y *Joliet*, que permiten cosas como nombres de archivos *case-sensitive* y más profundidad en los directorios. Este driver no implementa ninguna de ellas.

Este servicio abstrae los LBAs en forma de inodos, que realmente son la dirección lineal (lba * 2048 + offset). Además, define una estructura abstracta de fichero, cuyos campos son:

- Inodo.
- Fecha de creación.
- Tamaño del archivo en bytes.
- Flags, solo se usa una: si es o no un directorio.
- Tamaño del nombre, seguido del nombre.

Esta estructura se usa a forma de marshalling para escribir sobre la región de memoria compartida.

La interfaz de procedimientos públicos de este servicio se encuentra en la tabla 5.8.

Tabla 5.8: Procedimientos públicos de StrifeFS

| RPID | Nombre | Permiso | Argumentos |
|---|---|---|---|
| 0 | `SETUP` | `ISO9660_SETUP` | 2 |
| 1 | `CONNECT` | `ISO9660` | 1 |
| 2 | `GET_ROOT` | `ISO9660` | 0 |
| 3 | `LIST` | `ISO9660` | 2 |
| 4 | `READ` | `ISO9660` | 2 |

Funcionan de la siguiente manera:

- `SETUP` inicializa el servicio con un UUID, con tal de comunicarse con block. Los argumentos son los primeros 64 bits del UUID, y los últimos. Devuelve `true` si todo ha ido bien; esto es, si no estaba inicializado ya, y si se ha podido parsear correctamente el PVD.

- `CONNECT` funciona de la manera usual.
- `GET_ROOT` devuelve el inodo de la raíz.
- `LIST` recibe dos parámetros: el inodo, y la página. Así, lista los archivos del directorio con el inodo dado, paginando el proceso, similar a lo que hacía el servicio block. La región de memoria compartida se llena de instancias de la estructura abstracta de fichero definida arriba.
- `READ` recibe el inodo y la página. Sencillamente, lee el archivo de forma paginada, devolviendo en cada paso cuántos bytes se han escrito en la página compartida.

### 5.5.6. StrifeFS

StrifeFS fue ampliamente explicado en 4.6.

**TODO: Marshalling pendiente de definir, según se vea en la implementación**

Su interfaz de procedimientos públicos es similar a la de ISO9660, y se encuentra en la tabla 5.9.

Tabla 5.9: Procedimientos públicos de StrifeFS

| RPID | Nombre | Permiso | Argumentos |
|---|---|---|---|
| 0 | `CONNECT` | `BLOCK_LIST` | 1 |
| 1 | `LIST_DEVICES` | `BLOCK_LIST` | 1 |
| 2 | `READ` | `BLOCK_READ` | 4 |
| 3 | `WRITE` | `BLOCK_WRITE` | 4 |

### 5.5.7. VFS

El VFS es la capa de abstracción sobre todos los sistemas de archivos, quedando así en la cima de la pila de almacenamiento. Homogeneiza los sistemas de archivos, dándoles todos una representación igual a la de StrifeFS.

Está basado en la idea de los puntos de montaje. Para empezar, siempre hay una raíz montada. Bajo ella, en directorios concretos pueden estar montados otros sistemas de archivos. Cuando llega una ruta al servicio, primero ocurre un proceso de simplificación, en el cual:

- Se elimina todo autolado, es decir, las partes `/./` de una ruta. `/cd/./boot` se simplifica a `/cd/boot`.
- Se elevan los directorios con referencias a `..`; `/cd/boot/../libs` se simplifica a `/cd/libs`.
- Se eliminan los separadores duplicados. `/cd//boot` se simplifica a `/cd/boot`.

Tras esto, el VFS intenta identificar a qué punto de montaje corresponde la ruta simplificada. Esto se hace por medio de un algoritmo de máxima coincidencia. Los puntos de

montaje se almacenan como la ruta sobre la cual están montados. Por ejemplo, se pueden tener los siguientes puntos de montaje:

- `/`, StrifeFS sobre `ramblock`.
- `/cd/`, ISO9660 sobre AHCI ATAPI.

La ruta `/cd/boot` coincide en su primer caracter con el primer punto de montaje, y en sus cuatro primeros con el segundo. El que más caracteres coincida, terminando en `/`, determina cuál es el punto de montaje. A partir de ese punto, se elimina la ruta de montaje de la ruta simplificada, y resulta en la ruta relativa, que es la que se envía al servicio correspondiente.

**TODO: faltan los procedimientos públicos**

## 5.6. Del teclado a coreutils

### 5.6.1. term

`term` es el driver que implementa una terminal sobre el framebuffer de modo texto de la BIOS. Esta región de memoria está habitualmente situada en la dirección física `0xB8000`. El servicio implementa el scrolling cuando se pasa de la última línea de texto, así como el cursor, que mantiene, siempre que esté visible, en la posición siguiente al último caracter escrito.

Antes de que comience el proceso de bootstrapping, el driver no se está ejecutando, pero el kernel necesita imprimir cosas por pantalla, especialmente en caso de kernel panic para averiguar qué ha ido mal. Por ello, el kernel contiene un driver similar a este, salvo que es más simple. Este driver del kernel no se desecha una vez `term` se está ejecutando, pues aumentaría muy considerablemente la cantidad de cambios de contexto. En su lugar, se mantienen sincronizados. Cuando el kernel arranca el servicio, le monta una página que es compartida por el kernel, donde se mantiene en todo momento la fila y columna del cursor. Así, pueden escribir sin *pisarse el uno al otro.*

`term` es especialmente simple en cuanto a su interfaz pública. En la tabla 5.10 se enumeran los procedimientos disponibles para RPC.

Tabla 5.10: Procedimientos públicos de term

| RPID | Nombre | Permiso | Argumentos |
|---|---|---|---|
| 0 | `CONNECT` | Ninguno | 1 |
| 1 | `FLUSH` | Ninguno | 1 |

- `CONNECT` funciona de la forma usual. Se establece una página de memoria compartida con el proceso que quiera escribir por pantalla.
- `FLUSH` recibe un parámetro: el número de bytes escritos en la página compartida. Realiza la escritura de ese número de bytes por pantalla.

La comunicación con `term` está abstraída dentro de la librería estándar, se explicará más adelante en 5.7.5.

### 5.6.2. keyboard

### 5.6.3. shell

### 5.6.4. coreutils

## 5.7. La librería estándar

Existen varios puntos de interés a comentar sobre la librería estándar.

### 5.7.1. La STL

Una muy gran parte de la librería estándar es la STL (*Standard Template Library*), un conjunto de clases que implementan estructuras de datos abstractas para ser usadas por el resto de programas de forma transparente. Ejemplos incluyen `std::string` o contenedores como `std::set` o `std::unordered_map`. Son muy similares a las de otros sistemas operativos, pero no compatibles, pues no se adhieren necesariamente al estándar. Aquí se encuentra una lista de las estructuras abstractas de alto nivel implementadas:

- `bitmap`, un mapa de bits sobre una región de memoria dada. Es equivalente a un vector de booleanos, pero mucho más eficiente en memoria.
- `vector`, un contenedor de datos consecutivos en memoria, rápidamente iterables.
- `list`, lista enlazada.
- `dlist`, una lista doblemente enlazada.
- `priority_queue`, un contenedor especializado en mantener rápidamente accesible el elemento mayor o menor de una secuencia. Implementado como una heap.
- `map`, un contenedor clave-valor que mantiene las claves ordenadas, útil si quieren ser iteradas. Implementado como un AVL.
- `set`, un contenedor genérico que mantiene los elementos ordenados, para ser iterados. Implementado como un AVL.
- `unordered_map`, contenedor clave-valor desordenado. Implementado como una tabla hash Robin Hood.
- `unordered_set`, contenedor genérico desordenado. Implementado como una tabla hash Robin Hood.
- `pair`, un par de elmentos arbitrarios.
- `queue`, un contenedor FIFO.
- `stack`, un contenedor LIFO.
- `string`, que, aunque no es una estructura abstracta, se suele incluir en esta clase de proyectos.

### 5.7.2. Allocator

El allocator que usa la librería estándar de C++ de Strife es `liballoc`, una implementación independiente de la plataforma fácil de incluir en sistemas operativos hobby [78]. El programador del sistema solo tiene la responsabilida de establecer un mecanismo de cerrojo para en caso de multithreading, además de funciones para reservar memoria en la heap.

### 5.7.3. La potencia de CISC

La librería estándar aporta algunas funciones escritas para ser especialmente rápidas en los procesadores modernos de x86. Se trata del trío `memcpy`, `memmove`, y `memset`.

Desde los procesadores Ivy Bridge, los procesadores implementan una funcionalidad denominada ERMSB (*Enhanced rep movsb*), que hace que la copia de bytes de una parte de la memoria a otra sea, bajo ciertas circunstancias que se suelen cumplir, más rápida que cualquier otro método [79].

Por esto, las tres funciones estándares mencionadas están implementadas en ensamblador utilizando estas instrucciones. Tanto `memcpy` como `memmove` están implementadas con `rep movsb`, y `memset` está implementado con `rep stosb`.

### 5.7.4. Abstracción sobre RPC

Desde el lado del cliente, hay una fina abstracción sobre la syscall de RPC en la librería estándar. En lugar de tener una única función para hacerla, existen cinco, dependiendo del número de parámetros (de 0 a 4). Esto permite al compilador organizar mejor los registros para conseguir el mínimo movimiento posible.

Desde el lado del servidor, la abstracción es mayor. Se aporta un punto de entrada RPC por defecto, que se usa cuando `std::enableRPC` se llama sin el puntero a la función. Esta tiene su entrada en ensamblador y organiza los RPIDs mediante una tabla hash. Se aporta una función para modificar esta tabla hash y añadir procedimientos públicos, `std::exportProcedure`, que recibe un puntero a función casteado a `void*` y el número de argumentos que espera recibir. Es importante comentar que la tabla hash está implementada *lock-less*, sin exclusión mutua, para acelerar la entrada, con lo cual todas las llamadas a `exportProcedure` deben de realizarse antes de la llamada a `enableRPC()`, para que no existan condiciones de carrera.

Un ejemplo sería de este estilo:

```
std::exportProcedure((void*)connect, 1);
std::exportProcedure((void*)flush, 1);
std::enableRPC();
std::publish("term");
std::halt();
```

**TODO: Abstraer los permisos con el registro?**

### 5.7.5. Abstracción sobre term

Es evidente que el sistema operativo no fuerza al programador a escribir en la página compartida con `term` cada vez que quiera escribir por pantalla. En su lugar, toda la comunicación con el servicio, incluyendo la llamada a `CONNECT` para inicializar la página compartida, se encuentra encapsulada en la librería estándar.

Sobre esta encapsulación existe una implementación de `printf` funcional, que almacena todo lo que se va a imprimir en un buffer (la página compartida), y se hace la llamada a `FLUSH` cuando se llena o cuando ocurre un salto de línea, de igual forma que está implementada en la libC de GNU.

### 5.7.6. Abstracción del registro

**TODO: Pendiente de implementar**

# Capítulo 6

# Experimentos, conclusiones y trabajo futuro

## 6.1. Experimentos

Tamaño ISO, RAM mínima, capturas de pantalla

## 6.2. Conclusiones

## 6.3. Trabajo futuro

**TODO**

## 6.4. Trabajo futuro propio

# Bibliografía


[1] @MessiahAndrw. OS Dev Wiki - Introduction. https://wiki.osdev.org/Introduction#Welcome, February 2007.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Computer Science and Information Processing. Addison-Wesley, Reading, Massachusetts, 1st edition, January 1983.

[4] Alfred Aho. Bell Labs' Role in Programming Languages and Algorithms. 2015. https://www.youtube.com/watch?v=rkuOTgfmH3w.

[5] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*, chapter 1, pages 3–6. Pearson, Boston, MA, 4 edition, 2014.

[6] Richard Stallman. Linux and the GNU System. https://www.gnu.org/gnu/linux-and-gnu.en.html.

[7] NetBSD. Platforms supported. https://wiki.netbsd.org/ports/.

[8] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*, chapter 1.7.1, pages 63–64. Pearson, Boston, MA, 4 edition, 2014.

[9] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*, chapter 1.7.3, pages 65–68. Pearson, Boston, MA, 4 edition, 2014.

[10] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*, chapter 5.3.2, pages 357–361. Pearson, Boston, MA, 4 edition, 2014.

[11] Intel Corporation. intel_display.c. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/gpu/drm/i915/display/intel_display.c?h=v5.18, 2007.

[12] The GNU Project. glibc. https://www.gnu.org/software/libc/.

[13] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Commun. ACM*, 17(7):365–375, jul 1974.

[14] The GNU Project. ls.c. https://github.com/coreutils/coreutils/blob/master/src/ls.c.

[15] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*, chapter 2, page 94. Pearson, Boston, MA, 4 edition, 2014.

[16] freedesktop.org. systemd System and Service Manager. https://www.freedesktop.org/wiki/Software/systemd/.

[17] Gentoo Linux. Project: OpenRC. https://wiki.gentoo.org/wiki/Project:OpenRC.

[18] Gerrit Pape. runit - a UNIX init scheme with service supervision. http://smarden.org/runit/.

[19] mainSystem V style init programs. https://savannah.nongnu.org/projects/sysvinit.

[20] William Stallings. *Computer Organization and Architecture*, chapter 8.2, pages 271–277. Prentice Hall Professional Technical Reference, 6th edition, 2002.

[21] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. An Experimental Time-Sharing System. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, AIEE-IRE '62 (Spring), page 335–344, New York, NY, USA, 1962. Association for Computing Machinery.

[22] David A. Solomon and Helen Custer. *Inside Windows NT*. Microsoft Press, USA, 2nd edition, 1998.

[23] Samuel J. Leffler, Marshall K. McKusick, Michael J. Karels, and John S. Quarterman. *The design and implementation of the 4.3 BSD Unix operating system*. Addison-Wesley series in computer science. Addison-Wesley, 1990.

[24] Ingo Molnar. Modular Scheduler Core and Completely Fair Scheduler [CFS]. https://web.archive.org/web/20170906174308/https://lwn.net/Articles/230501/.

[25] International Organization for Standardization. *Information processing — Volume and file structure of CD-ROM for information interchange*, 1988.

[26] Richard Russon and Yuval Fledel. NTFS Documentation. https://dubeyko.com/development/FileSystems/NTFS/ntfsdoc.pdf.

[27] Dave Poirier. The Second Extended File System. https://www.nongnu.org/ext2-doc/ext2.html.

[28] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*, chapter 3.3, pages 194–208. Pearson, Boston, MA, 4 edition, 2014.

[29] New York Times. BIG I.B.M.'S LITTLE COMPUTER. https://www.nytimes.com/1981/08/13/business/big-ibm-s-little-computer.html, 1981.

[30] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*, chapter 4.3.1, pages 281–282. Pearson, Boston, MA, 4 edition, 2014.

[31] Intel Inc. *Intel® 64 and IA-32 ArchitecturesOptimization Reference Manual*, volume 3A, chapter 2.2, pages 7–8.

[32] The GNU Project. GNU GRUB. https://www.gnu.org/software/grub/.

[33] Microsoft Inc. Windows 10 Mobile partition layout. https://docs.microsoft.com/en-us/windows-hardware/drivers/bringup/partition-layout.

[34] Intel Inc. *Intel® 64 and IA-32 ArchitecturesOptimization Reference Manual*, volume 3A, chapter 3.2, pages 2–6.

[35] Intel Inc. *Intel® 64 and IA-32 ArchitecturesOptimization Reference Manual*, volume 2A, pages 528–536.

[36] Intel Inc. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3A, chapter 3.4.5, page 10.

[37] Intel Inc. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3A, chapter 4.3, page 10.

[38] AMD Inc. *AMD64 Architecture Programmer's Manual*, volume 2, chapter 5, page 143.

[39] Intel Inc. *Intel® 64 and IA-32 ArchitecturesOptimization Reference Manual*, volume 3A, chapter 6.2, pages 6–7.

[40] Intel Inc. *Intel® 64 and IA-32 ArchitecturesOptimization Reference Manual*, volume 3A, chapter 6.10, pages 9–10.

[41] Intel Inc. *Intel® 64 and IA-32 ArchitecturesOptimization Reference Manual*, volume 3A, chapter 7.7, pages 19–20.

[42] Intel Inc. *Intel® 64 and IA-32 ArchitecturesOptimization Reference Manual*, volume 3A, chapter 6.3.1, pages 2–3.

[43] Intel Inc. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3A, chapter 10.

[44] AMD Inc. *AMD64 Architecture Programmer's Manual*, volume 2, chapter 6, pages 171–173.

[45] *Telephony: The American Telephone Journal*, chapter 5, page 20. Number 87. 1925.

[46] Clinton Davisson. The Discovery of Electron Waves. *Nobel Lectures*, 1965.

[47] AT&T. The UNIX Operating System. https://www.youtube.com/watch?v=tc4ROCJYbm0.

[48] Per Brinch Hansen. The Nucleus of a Multiprogramming System. *Communications of the ACM*, 1970.

[49] Carl Sassenrath. Amiga ROM Kernel Reference Manual. https://archive.org/details/1990-beats-steve-amiga-rom-kernel-ref-3rd/mode/2up, 1986.

[50] Jochen Liedtke. A persistent system in real use—experiences of the first 13 years. In *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 2–11, 1993.

[51] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. *SIGOPS Oper. Syst. Rev.*, 27(5):120–133, dec 1993.

[52] Jochen Liedtke. Improving IPC by Kernel Design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. Association for Computing Machinery.

[53] Jochen Liedtke. On Micro-Kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 237–250, New York, NY, USA, 1995. Association for Computing Machinery.

[54] Kevin Elphinstone and Gernot Heiser. From L3 to seL4: What Have We Learnt in 20 Years of L4 Microkernels? In *Proceedings of the Twenty-Fourth ACM, Symposium on Operating Systems Principles*, page 134, 2013.

[55] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[56] Free Software Foundation. The Free Software Definition. https://www.gnu.org/philosophy/free-sw.en.html.

[57] JBoot. https://github.com/the-strife-project/JBoot, 2020.

[58] jotaOS: rama antigua. https://github.com/the-strife-project/Strife/tree/old, 2021.

[59] F.R. Pedraza. *Historia de la filosofía, 2 Bachillerato*. Oxford Educación, 2009.

[60] Limine Bootloader. https://github.com/limine-bootloader/limine.

[61] stivale2 boot protocol specification. https://github.com/stivale/stivale/blob/master/STIVALE2.md.

[62] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[63] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[64] VESA. *VESA BIOS EXTENSION (VBE) Core Functions Standard*.

[65] Intel Inc. *UEFI Driver Development Guide for Graphics Controller Device Classes*, 2011.

[66] The Open Group. ftok - generate an IPC key. https://pubs.opengroup.org/onlinepubs/007904975/functions/ftok.html.

[67] Microsoft Corporation. Creating Named Shared Memory. https://docs.microsoft.com/en-us/windows/win32/memory/creating-named-shared-memory?redirectedfrom=MSDN.

[68] Shahram Saeidi and Hakimeh Baktash. Determining the Optimum Time Quantum Value in Round Robin Process Scheduling Method. *International Journal of Information Technology and Computer Science*, 4, 09 2012.

[69] K. Thompson and D. M. Ritchie. *UNIX PROGRAMMER'S MANUAL*. Bell Telephone Laboratories, Inc., fourth edition, 1973.

[70] Edward Buzzell, Marx Bros, Metro-Goldwyn Mayer. Go West. https://www.youtube.com/watch?v=UlVoZgM4fgI, 1940.

[71] The Strife Project: rpcSwitcher.asm. https://github.com/the-strife-project/kernel/blob/main/src/syscalls/rpcSwitcher.asm.

[72] Inc. UEFI Forum. *Advanced Configuration and Power Interface (ACPI) Specification*, 2019.

[73] Intel Inc. *Intel® 64 and IA-32 ArchitecturesOptimization Reference Manual*, volume 2B, page 235.

[74] The Strife Project: kernel.cpp. https://github.com/the-strife-project/kernel/blob/main/src/kernel.cpp.

[75] *ELF-64 Object File Format*, 1998.

[76] PCI Special Interest Group. *PCI Local Bus Specification*, 1998.

[77] Intel Inc. *Serial ATA: Advanced Host Controller Interface (AHCI) 1.3.1*, chapter 3.3.9, page 29.

[78] Durand. liballoc - a small memory allocator. https://github.com/blanham/liballoc.

[79] Intel Inc. *Intel® 64 and IA-32 ArchitecturesOptimization Reference Manual*, volume 3A, chapter 3.7.6, pages 61–63.