UNIVERSIDAD DE GRANADA

Sistemas Operativos

Temario de la asignatura

Ismael Sallami Moreno

Recursos Ingeniería Informática y Ade

Licencia

Sistemas Operativos

Ismael Sallami Moreno

https://elblogdeismael.github.io

Índice general

1 Estructuras de sistemas operativos **7**

1.1 Qué es un sistema operativo **7**

1.2 El apoyo del hardware **7**

1.3 La llamada al sistema **8**

1.4 Arquitecturas de sistemas operativos **9**

1.5 Estructura interna: capas y mecanismos **10**

1.6 Arranque **10**

1.7 Sistemas operativos de propósito específico **11**

2 Procesos e hilos **13**

2.1 Proceso **13**

2.2 Hilos **15**

2.3 Diagrama de estados **16**

2.4 Cambio de contexto **17**

2.5 Planificación de la CPU **17**

3 Gestión de memoria **21**

3.1 El problema **21**

3.2 Gestión de memoria para el sistema operativo **21**

3.3 Gestión de memoria para los procesos **22**

3.4 Memoria virtual **24**

4 Gestión de archivos **29**

4.1 Interfaz de los sistemas de archivos **29**

4.2 Diseño software del sistema de archivos **32**

4.3 Implementación de los sistemas de archivos **33**

5 Gestión de entradas y salidas **37**

5.1 El problema **37**

5.2 Comunicación con el dispositivo **37**

5.3 Arquitectura software del sistema de entrada/salida **39**

5.4 Archivos de dispositivos **40**

5.5 Manejadores de dispositivos **41**

6 Mecanismos de seguridad **43**

6.1 Objetivos de protección y amenazas **43**

6.2 Autenticación **45**

6.3 Mecanismos de autorización **46**

6.4 Seguridad en la administración **48**

7 Temario práctico **49**

7.1 Práctica 1. Administración del sistema **49**

7.2 Práctica 2. Servicios del sistema mediante la API **51**

Capítulo 1

Estructuras de sistemas operativos

Tema 1 del programa. Qué papel cumple un sistema operativo, qué apoyo necesita del hardware para cumplirlo, cómo se reparte su código en capas y qué formas toma ese reparto cuando el destino no es un ordenador de propósito general.

1.1 Qué es un sistema operativo

Un sistema operativo es el programa que se interpone entre las aplicaciones y la máquina. Hace dos trabajos que conviene no mezclar:

- **Máquina extendida.** Ofrece abstracciones —proceso, hilo, archivo, dispositivo, espacio de direcciones— que ocultan la interfaz real del hardware. Una aplicación escribe en un archivo; no programa el controlador de disco.
- **Gestor de recursos.** Reparte CPU, memoria, disco y dispositivos entre programas que compiten, y decide en qué orden y durante cuánto tiempo.

Las dos caras se contradicen a menudo. La abstracción quiere que el programa no se entere de nada; la gestión quiere control sobre lo que el programa hace. Casi todas las decisiones de diseño que siguen son un punto de equilibrio entre las dos. La distinción se plantea en estos términos en Tanenbaum, 2009 y en Stallings, 2018.

1.2 El apoyo del hardware

Sin tres mecanismos del procesador, un sistema operativo multiprogramado no se puede construir. No son opcionales: son la frontera que separa un sistema operativo de una biblioteca.

1.2.1 Modos de ejecución

El procesador distingue al menos dos niveles de privilegio:

Modo	Otros nombres	Qué permite
Núcleo	supervisor, kernel, anillo 0	todo el repertorio de instrucciones, incluidas las privilegiadas
Usuario	anillo 3	solo el subconjunto no privilegiado

Son privilegiadas las instrucciones que pueden comprometer al resto del sistema: cambiar el registro que apunta a las tablas de páginas, habilitar o inhibir interrupciones, acceder directamente a puertos de entrada/salida, detener el procesador. Ejecutar una de ellas en modo usuario no la ejecuta:

provoca una excepción, y el control pasa al sistema operativo.

El bit de modo vive en un registro de estado del procesador. Cambiarlo de usuario a núcleo no es una instrucción normal, precisamente porque entonces no protegería nada: solo cambia en los tres eventos que se ven más abajo.

1.2.2 Interrupciones, excepciones y trampas

El procesador abandona el flujo de instrucciones actual en tres situaciones, que se implementan igual y significan cosas muy distintas:

Evento	Origen	Síncrono	Ejemplo
Interrupción	dispositivo externo	no	el disco terminó una transferencia
Excepción	la propia instrucción	sí	división por cero, fallo de página
Trampa (`trap`)	instrucción deliberada	sí	llamada al sistema

En los tres casos el hardware guarda el contador de programa y el registro de estado, pasa a modo núcleo y salta a una dirección que fijó el sistema operativo durante el arranque. Esa dirección sale de una tabla de vectores indexada por el número del evento.

La distinción importa para el diseño: una excepción es reproducible y atribuible a un proceso concreto, una interrupción no. Por eso un fallo de página se resuelve en el contexto del proceso que lo causó, mientras que la rutina de una interrupción de disco no puede suponer nada sobre qué proceso está en la CPU.

1.2.3 El temporizador

Un reloj programable interrumpe periódicamente al procesador. Es lo único que garantiza que el sistema operativo recupere el control de un programa que no lo cede voluntariamente. Sin temporizador no hay multiprogramación con reparto de tiempo: un bucle infinito en modo usuario congelaría la máquina.

El periodo del *tick* fija la granularidad con la que se puede planificar. Los núcleos modernos lo hacen configurable, y los sistemas *tickless* llegan a desprogramarlo cuando la CPU está ociosa, para no despertar un procesador que podría estar en un estado de bajo consumo.

1.2.4 Protección de memoria

La unidad de gestión de memoria traduce cada dirección que emite el procesador y comprueba los permisos de la región a la que pertenece. Un acceso fuera de lo permitido genera una excepción antes de que llegue a la memoria física. El detalle está en el tema 3; aquí basta con retener que la protección es hardware y que el sistema operativo solo programa las tablas.

1.3 La llamada al sistema

Es la única puerta por la que una aplicación pide un servicio al núcleo. La secuencia, con los servicios POSIX como referencia:

1. La aplicación llama a una función de la biblioteca de C, por ejemplo `write()`. Esa función no es la llamada al sistema: es su envoltorio.

2. El envoltorio coloca el número de servicio y los argumentos donde el convenio de la arquitectura manda, casi siempre en registros.
3. Ejecuta la instrucción de trampa (`syscall` en x86-64, `svc` en ARM).
4. El hardware pasa a modo núcleo y salta al manejador.
5. El manejador valida el número, comprueba los argumentos —incluidos los punteros, que apuntan a memoria de usuario y pueden ser inválidos— y llama a la rutina correspondiente.
6. Al volver, el resultado se deja en un registro y el envoltorio lo traduce al convenio de C: valor negativo a `-1` con `errno` puesto.

Ese último punto explica una confusión frecuente. `errno` no lo escribe el núcleo: lo escribe la biblioteca a partir del código de error que el núcleo devuelve.

El coste de una llamada al sistema no está en la trampa, que son unos cientos de ciclos, sino en lo que arrastra: invalidación de cachés, posible cambio de tablas de páginas y, desde las mitigaciones de Spectre y Meltdown, vaciados adicionales del predictor de saltos y de la TLB. Por eso las interfaces modernas de entrada/salida —`io_uring`, `epoll`— se diseñan para amortizar una transición entre muchas operaciones.

1.4 Arquitecturas de sistemas operativos

1.4.1 Monolítico

Todo el sistema operativo se ejecuta en modo núcleo, en un único espacio de direcciones: planificador, gestión de memoria, sistemas de archivos y manejadores de dispositivos comparten memoria y se llaman entre sí como funciones ordinarias.

- **A favor:** rendimiento. Una llamada interna es un salto, no un mensaje.
- **En contra:** un fallo en cualquier parte se lleva el sistema entero, y la superficie de código privilegiado es enorme.

El monolítico puro apenas existe. Linux, Windows NT y los BSD son **monolíticos modulares**: el núcleo carga y descarga módulos en tiempo de ejecución, pero esos módulos siguen ejecutándose en modo núcleo. La modularidad es de compilación y despliegue, no de aislamiento.

1.4.2 Microkernel

En modo núcleo queda lo mínimo indispensable: paso de mensajes, planificación básica y gestión del espacio de direcciones. Sistemas de archivos, pila de red y manejadores pasan a ser procesos de usuario, llamados *servidores*.

- **A favor:** un manejador que falla se reinicia sin tocar el resto. La base de cómputo confiable cabe en unas miles de líneas, y en el caso de seL4 está verificada formalmente.
- **En contra:** lo que en un monolítico era una llamada a función pasa a ser varios mensajes con sus cambios de contexto.

Mach, QNX, MINIX 3 y seL4 son los ejemplos habituales. El coste del paso de mensajes fue el argumento contra el enfoque durante los años noventa; L4 lo redujo un orden de magnitud y el debate se reabrió. La defensa del enfoque, con MINIX 3 como caso de estudio, está en Tanenbaum, 2009.

1.4.3 Híbrido

Un microkernel al que se le devuelven al modo núcleo los servicios cuyo coste de comunicación resultaba inaceptable. XNU, el núcleo de macOS, combina Mach con código BSD en el mismo

espacio de direcciones. La etiqueta describe una decisión de ingeniería, no una arquitectura distinta.

1.4.4 Máquinas virtuales y contenedores

Dos formas de aislar que se confunden a menudo:

	Hipervisor	Contenedor
Qué virtualiza	el hardware	la vista del sistema operativo
Núcleos en juego	uno por máquina virtual, más el del anfitrión	uno solo, compartido
Frontera de aislamiento	tablas de páginas anidadas y modo raíz del procesador	espacios de nombres y grupos de control
Coste de arranque	segundos	milisegundos

Un hipervisor de tipo 1 se ejecuta sobre el hardware desnudo (Xen, ESXi); uno de tipo 2 se apoya en un sistema operativo anfitrión (VirtualBox). Un contenedor no es una máquina virtual: comparte el núcleo, así que una vulnerabilidad de escalada en el núcleo compromete a todos los contenedores del anfitrión.

1.4.5 Exokernel y unikernel

El exokernel lleva la idea contraria al microkernel: en vez de subir las abstracciones a procesos de usuario, las elimina y expone el hardware con protección pero sin abstracción, dejando que cada aplicación enlace la biblioteca de sistema operativo que le convenga. El unikernel es su versión práctica: una única aplicación enlazada con las partes del sistema que necesita, en un solo espacio de direcciones, arrancando directamente sobre el hipervisor.

1.5 Estructura interna: capas y mecanismos

Independientemente de la arquitectura, el código se organiza en cuatro estratos:

1. **Capa dependiente de la máquina.** Arranque, cambio de contexto, tablas de páginas, controlador de interrupciones. Se reescribe por arquitectura.
2. **Mecanismos.** Planificar, asignar marcos, planificar peticiones de disco.
3. **Políticas.** Qué proceso va primero, qué página se expulsa, qué petición se sirve antes.
4. **Interfaz.** Llamadas al sistema y sistemas de archivos virtuales.

La separación entre mecanismo y política es la regla de diseño más rentable del tema: permite cambiar el planificador sin tocar el cambio de contexto. Linux la aplica con las clases de planificación y con los planificadores de bloque intercambiables.

1.6 Arranque

El camino desde el encendido hasta el primer proceso de usuario:

1. **Firmware** (UEFI o BIOS). Inicializa la memoria y los buses, y localiza el cargador.
2. **Cargador de arranque** (GRUB, systemd-boot). Lee el núcleo y, si lo hay, el sistema de archivos inicial en memoria.
3. **Núcleo.** Descomprime, monta la tabla de páginas definitiva, detecta el hardware, monta la raíz.

4. **Primer proceso.** El núcleo crea el proceso 1 —`init`, `systemd`— que arranca todo lo demás. Si ese proceso muere, el núcleo entra en pánico: no hay a quién reasignar los procesos huérfanos.

El sistema de archivos inicial en memoria (`initramfs`) existe por un problema de circularidad: el núcleo necesita el manejador del disco para montar la raíz, y ese manejador está en la raíz. Se resuelve cargando en memoria un sistema mínimo que contiene los módulos imprescindibles. La secuencia completa, con el detalle de qué hace cada fase del núcleo de Linux, está en Mauerer, 2008.

1.7 Sistemas operativos de propósito específico

Cuando el destino no es un ordenador de propósito general, las prioridades cambian y con ellas el diseño.

1.7.1 Tiempo real

Lo que define un sistema de tiempo real no es la velocidad, sino el **determinismo**: la garantía de que una respuesta llega antes de un plazo. Un sistema rápido en el caso medio pero con un caso peor desconocido no sirve.

- **Tiempo real estricto** (*hard*): incumplir el plazo es un fallo del sistema. Control de vuelo, frenado ABS.
- **Tiempo real flexible** (*soft*): incumplirlo degrada la calidad. Reproducción de audio y vídeo.

Consecuencias de diseño: núcleo apropiativo, latencia de interrupción acotada y publicada, ausencia de memoria virtual con paginación bajo demanda —un fallo de página tiene un coste impredecible—, y protocolos de herencia de prioridad para acotar la inversión de prioridades. FreeRTOS, VxWorks y QNX son ejemplos; Linux se acerca con el parche `PREEMPT_RT`, ya integrado en la rama principal.

La inversión de prioridades merece atención porque es el fallo clásico del área. Una tarea de baja prioridad toma un cerrojo; una de alta prioridad se bloquea esperándolo; una de prioridad intermedia, que no necesita el cerrojo, expulsa a la de baja. La de alta prioridad queda esperando indefinidamente a la de baja, que no avanza. Es lo que dejó al Mars Pathfinder reiniciándose en 1997. La herencia de prioridad lo corrige elevando temporalmente la prioridad del poseedor del cerrojo a la del bloqueado más prioritario.

1.7.2 Empotrados

Recursos escasos y función fija. Muchos no tienen unidad de gestión de memoria, así que no hay espacios de direcciones separados: todo el código comparte memoria y un puntero corrupto lo corrompe todo. El sistema operativo suele ser una biblioteca enlazada con la aplicación, sin llamadas al sistema ni cambios de modo.

1.7.3 Móviles

Comparten la base con los de propósito general —Android es Linux, iOS comparte XNU con macOS— y se diferencian en la gestión de la energía y en el modelo de ciclo de vida de las aplicaciones. Ninguna aplicación decide cuánto vive: el sistema la suspende o la mata según la memoria disponible, y espera que guarde su estado cuando se le avisa. La política de permisos por aplicación, y no por usuario, es la otra diferencia estructural: el modelo Unix de usuarios no resuelve el problema de aislar aplicaciones que pertenecen al mismo usuario.

1.7.4 Distribuidos y de red

Un sistema operativo de red ofrece recursos remotos que el usuario sabe que son remotos: se monta un volumen, se abre una sesión. Uno distribuido oculta la distribución y presenta un único sistema. La segunda categoría apenas ha salido de la investigación —Amoeba, Plan 9—, porque la transparencia total choca con los fallos parciales: en una sola máquina, o funciona todo o no funciona nada; en un conjunto de máquinas, una parte falla mientras el resto sigue.

Capítulo 2

Procesos e hilos

Tema 2 del programa. La abstracción de proceso y la de hilo, cómo se representan dentro del núcleo, el diagrama de estados por el que pasan y los algoritmos con los que se decide quién ocupa la CPU.

2.1 Proceso

Un proceso es un programa en ejecución junto con todo el estado que necesita para continuar: memoria, registros, archivos abiertos, identidad y credenciales. La diferencia entre programa y proceso es la que hay entre una receta y una comida en preparación. El programa es pasivo y vive en disco; el proceso es activo y vive mientras se ejecuta.

Un mismo programa puede dar lugar a muchos procesos simultáneos, y cada uno tiene su propio estado. Comparten el texto —el código, que es de solo lectura— y no comparten nada más.

2.1.1 El bloque de control de proceso

El núcleo representa cada proceso con una estructura, el **PCB** (*Process Control Block*), que en Linux es `struct task_struct`. Contiene:

Grupo	Contenido
Identificación	PID, PID del padre, identificadores de usuario y grupo
Estado	estado actual, código de salida, señales pendientes y bloqueadas
Contexto de CPU	contador de programa, puntero de pila, registros generales y de estado
Memoria	puntero a la estructura del espacio de direcciones, tablas de páginas
Archivos	tabla de descriptores, directorio actual, raíz, máscara de creación
Planificación	prioridad, clase, tiempo consumido, CPU preferida
Contabilidad	tiempos de usuario y de sistema, fallos de página, límites de recursos

El tamaño de esa estructura importa: es lo que hay que recorrer y actualizar en cada cambio de contexto y en cada decisión de planificación.

2.1.2 El espacio de direcciones

La memoria de un proceso Unix se divide en regiones con permisos distintos:

Región	Permisos	Contenido
Texto	lectura y ejecución	código del programa, compartible entre procesos
Datos inicializados	lectura y escritura	variables globales con valor inicial
BSS	lectura y escritura	variables globales a cero, no ocupa espacio en el ejecutable
Montículo	lectura y escritura	memoria dinámica, crece hacia direcciones altas
Pila	lectura y escritura	marcos de llamada, crece hacia direcciones bajas

Entre montículo y pila queda un hueco que ambos consumen desde extremos opuestos. Que el texto sea de solo lectura y no ejecutable la pila —el bit NX— es la contramedida básica contra la inyección de código.

2.1.3 Creación: `fork` y `exec`

Unix separa dos operaciones que otros sistemas juntan en una:

- **`fork()`** duplica el proceso llamante. Devuelve dos veces: 0 en el hijo y el PID del hijo en el padre. Es el único servicio con esa propiedad, y de ahí sale el patrón de programación habitual.
- **`exec()`** sustituye la imagen del proceso actual por otro programa. No crea un proceso nuevo: reutiliza el que llama. Si tiene éxito no vuelve.

La separación es lo que permite que el hijo ajuste su entorno —redirigir descriptores, cambiar de directorio, bajar privilegios— entre las dos llamadas. Es exactamente lo que hace un intérprete de órdenes al montar una tubería. El repertorio completo de servicios implicados y sus casos límite está en Kerrisk, 2010 y en Stevens y Rago, 2005.

```c
pid_t pid = fork();
if (pid < 0) {
    perror("fork");
} else if (pid == 0) {
    /* Hijo: se redirige la salida antes de reemplazar la imagen. */
    int fd = open("salida.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    dup2(fd, STDOUT_FILENO);
    close(fd);
    execlp("ls", "ls", "-l", NULL);
    _exit(127);          /* solo se llega aqui si exec fallo */
} else {
    int estado;
    waitpid(pid, &estado, 0);
}
```

Duplicar el espacio de direcciones entero sería carísimo, y casi siempre inútil porque el hijo va a llamar a `exec` inmediatamente. Se resuelve con **copia al escribir**: padre e hijo comparten los

marcos físicos marcados de solo lectura, y el primer intento de escritura provoca una excepción que el núcleo atiende duplicando solo esa página.

2.1.4 Terminación, zombis y huérfanos

Un proceso termina llamando a `exit()` o recibiendo una señal que lo mata. Al terminar no desaparece del todo: el núcleo conserva su entrada en la tabla de procesos con el código de salida hasta que el padre lo recoge con `wait()`. En ese intervalo el proceso es un **zombi**: no consume CPU ni memoria, pero ocupa un PID.

Un padre que no llama a `wait()` acumula zombis hasta agotar la tabla de procesos. Es un fallo de programación, no del sistema.

El caso simétrico es el **huérfano**: el padre termina antes que el hijo. El núcleo reasigna el huérfano al proceso 1, que llama a `wait()` en un bucle precisamente para recogerlos. Un huérfano no es un problema; un zombi con un padre vivo que nunca lo recoge, sí.

2.2 Hilos

Un hilo es un flujo de ejecución dentro de un proceso. Los hilos de un proceso comparten espacio de direcciones, descriptores de archivo y señales, y tienen propios el contador de programa, los registros y la pila.

Recurso	Compartido entre hilos	Propio de cada hilo
Código y datos globales	sí	—
Montículo	sí	—
Descriptores de archivo	sí	—
Pila	—	sí
Registros y contador de programa	—	sí
Máscara de señales	—	sí
`errno`	—	sí, por almacenamiento local al hilo

Que `errno` sea local al hilo es un detalle con historia: en las primeras bibliotecas era una variable global, y dos hilos que fallaban a la vez se pisaban el código de error.

2.2.1 Por qué hilos y no procesos

- Crear un hilo cuesta un orden de magnitud menos que crear un proceso, porque no hay que duplicar el espacio de direcciones.
- El cambio entre hilos del mismo proceso no invalida la TLB: las tablas de páginas son las mismas.
- La comunicación es memoria compartida directa, sin llamadas al sistema.

Y el precio: **no hay aislamiento**. Un puntero corrupto en un hilo corrompe a todos, y toda estructura compartida necesita sincronización explícita. Un fallo que mata un hilo mata el proceso entero.

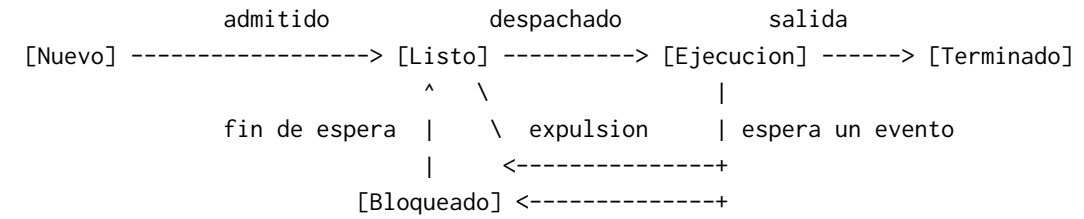
2.2.2 Modelos de implementación

Modelo	Dónde se planifican	Consecuencia
N:1, en biblioteca	espacio de usuario	cambio rapidísimo, pero una llamada bloqueante detiene todos los hilos y no hay paralelismo real
1:1, en núcleo	núcleo	cada hilo es planificable y bloquea solo a sí mismo; el coste de creación es mayor
M:N, híbrido	ambos	teóricamente lo mejor de los dos; en la práctica la complejidad no compensó

Linux implementa 1:1: `clone()` crea una tarea que comparte con la llamante los recursos que se indiquen mediante banderas, y `fork()` y la creación de hilos son la misma llamada con distintas banderas. Solaris intentó M:N y acabó volviendo a 1:1. Las **corrutinas** y las *goroutines* de Go recuperan la idea M:N en el espacio de usuario, pero para tareas que se bloquean en entrada/salida, no para paralelismo de CPU.

2.3 Diagrama de estados

Un proceso o hilo pasa por cinco estados:



Estado	Significado
Nuevo	creado, aún sin admitir en la cola de listos
Listo	podría ejecutarse, espera turno de CPU
En ejecución	ocupa una CPU
Bloqueado	espera un evento; aunque hubiera CPU libre no podría avanzar
Terminado	acabó, se conserva su registro hasta que el padre lo recoge

Las transiciones que conviene distinguir:

- **Listo → Ejecución.** La decide el planificador. Es la única transición que elige una política.
- **Ejecución → Listo.** Expulsión: se agotó el cuanto o llegó alguien más prioritario. El proceso podía seguir.
- **Ejecución → Bloqueado.** El proceso pidió algo que no está: una lectura de disco, un dato de red, un cerrojo ocupado. Es voluntaria.
- **Bloqueado → Listo.** Llegó el evento. **No pasa a ejecución directamente**: vuelve a la cola de listos y compite por la CPU. Confundir estas dos es el error clásico del tema.

Los sistemas con memoria virtual añaden dos estados suspendidos, para procesos cuyas páginas se han expulsado a disco: *listo y suspendido* y *bloqueado y suspendido*. La suspensión la decide el planificador de medio plazo.

Linux nombra los estados de otra forma, y la diferencia tiene consecuencias prácticas: `TASK_INTERRUPTIBLE` es un bloqueo que una señal puede romper, y `TASK_UNINTERRUPTIBLE` uno que no. Un proceso atascado en el segundo estado, que `ps` muestra con `D`, no se puede matar ni con `SIGKILL`; es lo que ocurre cuando un disco o un montaje de red dejan de responder.

2.4 Cambio de contexto

Guardar el estado del proceso saliente y restaurar el del entrante. Los pasos:

1. Un evento entra en el núcleo: interrupción, excepción o llamada al sistema.
2. Se guardan los registros del proceso saliente en su PCB.
3. Se actualiza su estado y su contabilidad.
4. El planificador elige al siguiente.
5. Se conmuta el espacio de direcciones —cargar el registro que apunta a las tablas de páginas—, lo que invalida entradas de la TLB.
6. Se restauran los registros del entrante y se vuelve a modo usuario.

El coste directo es de unos pocos microsegundos. El indirecto es mayor y no se mide fácilmente: las cachés quedan llenas de datos del proceso anterior, y el entrante avanza despacio hasta volver a llenarlas. Los identificadores de espacio de direcciones (ASID, PCID) reducen la parte de la TLB, porque permiten que convivan entradas de varios espacios sin vaciarlas.

2.5 Planificación de la CPU

2.5.1 Criterios

Cinco medidas, y ninguna política las optimiza todas a la vez:

Criterio	Definición	Se quiere
Uso de CPU	fracción de tiempo ocupada	máximo
Productividad	procesos terminados por unidad de tiempo	máxima
Tiempo de retorno	desde la llegada hasta la terminación	mínimo
Tiempo de espera	tiempo total en la cola de listos	mínimo
Tiempo de respuesta	desde la llegada hasta la primera salida	mínimo

En un sistema interactivo pesa el tiempo de respuesta; en uno por lotes, la productividad. Son objetivos incompatibles, y por eso hay más de un algoritmo.

2.5.2 Apropiativa y no apropiativa

Una política es **no apropiativa** si el proceso conserva la CPU hasta que se bloquea o termina, y **apropiativa** si el núcleo puede quitársela. La segunda necesita el temporizador del tema 1 y complica el núcleo: si se expulsa un proceso que estaba dentro de una llamada al sistema, las

estructuras que estuviera modificando quedan a medias, y hace falta sincronización dentro del propio núcleo.

2.5.3 Los algoritmos

FCFS, primero en llegar primero en ser servido. No apropiativo, cola FIFO. Trivial y justo en apariencia, pero sufre el **efecto convoy**: un proceso largo al frente hace esperar a todos los cortos detrás, y el tiempo medio de espera se dispara.

SJF, primero el más corto. Minimiza demostrablemente el tiempo medio de espera. Su problema es que exige conocer de antemano la duración de la siguiente ráfaga, que no se conoce; se estima con una media exponencial de las anteriores:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\, \tau_n$$

donde t_n es la ráfaga real observada y $\alpha \in [0,1]$ pondera el pasado reciente frente a la historia. Con $\alpha = 0$ la estimación nunca cambia; con $\alpha = 1$ solo cuenta la última ráfaga. Su versión apropiativa, **SRTF**, es mejor todavía en tiempo medio de espera y hace morir de hambre a los procesos largos.

Turno rotatorio (*round robin*). FCFS con un cuanto de tiempo: agotado el cuanto, el proceso vuelve al final de la cola. La elección del cuanto lo decide todo:

- Cuanto grande: degenera en FCFS.
- Cuanto pequeño: buen tiempo de respuesta, pero los cambios de contexto se comen la CPU.

La regla práctica, recogida en Silberschatz et al., 2006, es que el 80 % de las ráfagas quepan dentro de un cuanto, lo que sitúa el valor típico entre 10 y 100 milisegundos.

Por prioridades. A cada proceso un número; primero el más prioritario. El riesgo es la **inanición**: un proceso de baja prioridad puede no ejecutarse nunca. Se corrige con **envejecimiento**, subiendo la prioridad de los que llevan mucho esperando.

Colas multinivel. Varias colas con políticas propias —una para procesos interactivos con turno rotatorio, otra por lotes con FCFS— y una política entre colas. En su versión **realimentada** los procesos cambian de cola según su comportamiento: quien agota el cuanto baja, quien se bloquea antes sube. Así el sistema deduce quién es interactivo y quién no, sin que nadie se lo diga. Es el esquema que usaron las variantes clásicas de Unix.

2.5.4 Un ejemplo comparado

Cuatro procesos que llegan en el instante 0 con ráfagas de 8, 4, 9 y 5 unidades, en ese orden, y un cuanto de 4 para el turno rotatorio:

Algoritmo	Orden de servicio	Tiempo medio de espera
FCFS	P1, P2, P3, P4	$(0+8+12+21)/4 = 10{,}25$
SJF	P2, P4, P1, P3	$(0+4+9+17)/4 = 7{,}50$
Turno rotatorio	P1, P2, P3, P4, P1, P3, P4, P3	$(12+4+16+15)/4 = 11{,}75$

SJF gana en espera media; el turno rotatorio pierde en esa medida y gana en la que no aparece en la tabla, el tiempo de respuesta: P4 empieza a ejecutarse en el instante 12 en vez de en el 21.

2.5.5 Multiprocesadores

Con varias CPU aparecen problemas que no existían:

- **Afinidad.** Migrar un proceso a otra CPU pierde su caché. El planificador prefiere devolverlo donde estaba; la afinidad blanda es una preferencia y la dura, una imposición del programador.
- **Equilibrado de carga.** Una CPU ociosa con otra saturada desperdicia la máquina. Se corrige empujando trabajo desde la cargada o tirando de él desde la ociosa, y ambas cosas rompen la afinidad.
- **NUMA.** Si la memoria está repartida entre nodos, un proceso debe ejecutarse en el nodo donde está su memoria. Planificar y asignar memoria dejan de ser decisiones independientes.
- **Multihilo simultáneo.** Dos hilos hardware en un mismo núcleo comparten unidades funcionales y caché: no son dos CPU. Un planificador que los trate como tales reparte mal.

2.5.6 Tiempo real

Dos algoritmos con garantías demostrables para tareas periódicas, cada una con periodo T_i y tiempo de cómputo C_i:

- **Tasa monótona** (RM). Prioridad estática, mayor cuanto menor es el periodo. Es óptimo entre los de prioridad fija. Admite el conjunto de tareas si

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

  cota que tiende a $\ln 2 \approx 0{,}693$ cuando n crece. La condición es suficiente, no necesaria: un conjunto que no la cumpla puede seguir siendo planificable.
- **Plazo más próximo primero** (EDF). Prioridad dinámica: la mayor para la tarea cuyo plazo está más cerca. Planifica cualquier conjunto que cumpla $\sum C_i/T_i \leq 1$, así que aprovecha el 100 % de la CPU. A cambio, su comportamiento al sobrecargarse es mucho peor: pierde plazos de forma impredecible, mientras que RM sacrifica primero las tareas de periodo largo.

2.5.7 Linux

El planificador por omisión desde 2007 es **CFS**, el planificador completamente justo. No usa cuantos fijos: mantiene por tarea un **tiempo virtual de ejecución** que avanza a un ritmo inversamente proporcional a su peso, y siempre elige la tarea con el menor. Las tareas se guardan en un árbol rojinegro ordenado por ese tiempo, así que elegir es tomar el nodo más a la izquierda, en tiempo constante amortizado.

El diseño de esa estructura y el detalle de cómo se calcula el peso están descritos en Love, 2010. El valor `nice`, de -20 a 19, se traduce en un peso, y cada punto supone aproximadamente un 10 % de CPU. Junto a CFS conviven las clases de tiempo real `SCHED_FIFO` y `SCHED_RR`, siempre más prioritarias, y `SCHED_DEADLINE`, que implementa EDF. Desde 2024 CFS ha sido reemplazado por **EEVDF**, que añade a la justicia una noción explícita de plazo para mejorar la latencia de las tareas interactivas.

Capítulo 3

Gestión de memoria

Tema 3 del programa. Cómo se reparte la memoria física entre el núcleo y los procesos, cómo se traduce una dirección lógica en una física y qué hace el sistema cuando la memoria que los procesos piden supera a la que hay.

3.1 El problema

La memoria física es un vector de bytes con direcciones consecutivas. Sobre él hay que sostener tres exigencias a la vez:

- **Reubicación.** Un programa se compila sin saber en qué dirección se cargará, y puede cargarse en una distinta cada vez.
- **Protección.** Un proceso no puede leer ni escribir la memoria de otro, ni la del núcleo.
- **Compartición.** Y sin embargo dos procesos que ejecutan el mismo programa deben poder compartir su código, y dos que cooperan, una región de datos.

Las tres se resuelven con lo mismo: **traducir**. Las direcciones que el proceso emite no son las de la memoria; hay una capa que las convierte, y esa capa es el lugar natural donde comprobar permisos y donde hacer que dos direcciones lógicas distintas apunten al mismo sitio.

Dirección	Otros nombres	Quién la ve
Lógica	virtual	el proceso, el compilador, el depurador
Física	real	el bus de memoria

La traducción la hace la **MMU**, la unidad de gestión de memoria, en hardware y en cada acceso. Que sea hardware no es un detalle de implementación: si fuera software, cada acceso a memoria costaría una llamada al sistema.

3.2 Gestión de memoria para el sistema operativo

El núcleo tiene sus propias necesidades y no puede usar las mismas abstracciones que ofrece a los procesos. No puede permitirse un fallo de página al atender un fallo de página, así que su memoria está fija en marcos físicos.

3.2.1 El sistema colega

Para asignar bloques físicos contiguos, Linux usa el **sistema colega** (*buddy system*). La memoria se divide en bloques de tamaño potencia de dos, de una página hasta 1024 páginas. Al pedir un bloque:

1. Se busca en la lista del tamaño exacto.

2. Si está vacía, se toma uno del siguiente tamaño y se parte en dos mitades, *colegas*.
3. Una mitad se entrega; la otra pasa a la lista del tamaño inferior.

Al liberar, si el colega del bloque también está libre, los dos se fusionan y la fusión se propaga hacia arriba. La dirección del colega se calcula con un `XOR` sobre el bit correspondiente al tamaño, así que encontrarlo es una operación, no una búsqueda.

- **A favor:** fusión barata y fragmentación externa contenida.
- **En contra:** fragmentación interna de hasta el 50 %. Pedir 33 páginas da un bloque de 64.

3.2.2 Asignadores de objetos

Casi todo lo que el núcleo asigna son estructuras pequeñas y del mismo tamaño, que se crean y destruyen constantemente: descriptores de archivo, entradas de directorio, sockets. Para eso el sistema colega es una herramienta demasiado gruesa.

La respuesta es el **asignador de losas** (*slab*): una caché por tipo de objeto, con los objetos ya construidos y listos para reutilizar. Ahorra la inicialización, y mantiene los objetos del mismo tipo juntos, lo que mejora el comportamiento de la caché del procesador. Linux ha usado tres implementaciones de la idea —SLAB, SLOB y SLUB—; la vigente es SLUB, más simple y con mejor escalabilidad en máquinas con muchos núcleos.

3.2.3 Zonas

No toda la memoria física es equivalente. Linux la divide en zonas porque ciertos dispositivos solo pueden hacer DMA sobre los primeros megabytes, y porque en máquinas de 32 bits no toda la memoria física cabía en el espacio de direcciones del núcleo. En 64 bits la distinción casi desaparece, pero la estructura sigue ahí y explica que un sistema con memoria libre pueda fallar al asignar: la memoria libre puede estar en la zona equivocada.

3.3 Gestión de memoria para los procesos

3.3.1 Asignación contigua

El esquema más simple: cada proceso ocupa un bloque contiguo, delimitado por un registro base y un registro límite. La traducción es una suma y una comparación, y la protección sale gratis.

Su problema es la **fragmentación externa**: tras varias asignaciones y liberaciones queda memoria libre repartida en huecos, ninguno lo bastante grande. Con particiones dinámicas hay que elegir hueco:

Algoritmo	Criterio	Comportamiento
Primer ajuste	el primer hueco que sirva	el más rápido; fragmenta el principio de la memoria
Siguiente ajuste	igual, pero desde donde se quedó	reparte la fragmentación; algo peor en ocupación
Mejor ajuste	el hueco más pequeño que sirva	deja restos minúsculos e inservibles
Peor ajuste	el hueco más grande	el peor de los cuatro en la práctica

La regla del 50 %, obtenida por análisis estadístico del primer ajuste, dice que con n bloques

asignados quedan del orden de $n/2$ huecos: un tercio de la memoria se pierde. La compactación —mover procesos para juntar los huecos— la resuelve, pero exige reubicación dinámica y detener el sistema mientras se copia.

3.3.2 Paginación

La memoria física se divide en **marcos** de tamaño fijo y el espacio lógico en **páginas** del mismo tamaño. Cualquier página puede ir a cualquier marco, así que la asignación deja de necesitar contigüidad y la fragmentación externa desaparece. Queda solo la interna, acotada por debajo de una página.

Una dirección lógica se parte en dos campos:

```
|<---- numero de pagina (p) ---->|<--- desplazamiento (d) --->|
```

Con páginas de 2^k bytes, el desplazamiento son los k bits bajos y el número de página el resto. No hace falta dividir: el reparto es un desplazamiento de bits, y de ahí que el tamaño de página sea siempre potencia de dos.

La **tabla de páginas** traduce p en un número de marco. Cada entrada guarda, además del marco:

Bit	Para qué
Presencia	la página está en memoria o no
Protección	lectura, escritura, ejecución
Modificado (*dirty*)	ha sido escrita desde que se cargó
Referenciado	ha sido accedida recientemente
Usuario/núcleo	accesible en modo usuario o solo en núcleo
Caché	si es cacheable; se desactiva para memoria de dispositivos

El bit de modificado es el que evita escrituras inútiles: una página que no se ha modificado desde que se leyó no hace falta volver a escribirla al expulsarla.

El tamaño de la tabla

Con direcciones de 64 bits y páginas de 4 KiB, una tabla lineal por proceso tendría 2^{52} entradas. Es inviable, y de ahí las tres soluciones:

- **Tablas multinivel.** El número de página se parte en varios campos, uno por nivel. Solo se crean las tablas de los niveles intermedios que se usan, así que un espacio de direcciones disperso ocupa poco. x86-64 usa cuatro niveles, y cinco desde que se amplió el espacio direccionable.
- **Tabla invertida.** Una entrada por marco físico en vez de por página lógica, indexada por una función *hash* del par proceso-página. El tamaño pasa a depender de la memoria instalada y no del espacio lógico; a cambio, compartir memoria se complica.
- **Tabla de páginas *hash*.** Variante de la anterior para espacios de direcciones grandes y dispersos.

La TLB

Con tablas de cuatro niveles, una traducción cuesta cuatro accesos a memoria más el acceso real: cinco veces el precio. La **TLB** es una caché asociativa de traducciones recientes que lo evita.

Si la tasa de aciertos es α, el acceso cuesta en la TLB t, el acceso a memoria m y la tabla tiene n niveles, el tiempo efectivo es

$$T = \alpha\,(t + m) + (1 - \alpha)\,\big(t + (n + 1)\,m\big)$$

Con $\alpha = 0{,}99$, $n = 4$, $t = 1$ ns y $m = 100$ ns el acceso efectivo son unos 105 ns frente a los 101 ideales: un 4 % de sobrecoste. Con $\alpha = 0{,}90$ sube a 141 ns, un 40 %. La eficacia del sistema entero descansa en esa tasa de aciertos, y por eso existen las **páginas grandes** de 2 MiB y 1 GiB: una sola entrada de TLB cubre lo que antes cubrían quinientas.

Al cambiar de proceso, las traducciones de la TLB dejan de valer. Vaciarla entera en cada cambio de contexto es caro, así que los procesadores etiquetan cada entrada con un identificador de espacio de direcciones y así conviven traducciones de varios procesos.

3.3.3 Segmentación

La paginación divide por tamaño; la segmentación divide por significado. Un proceso se ve como un conjunto de segmentos —código, datos, pila, cada biblioteca— de longitud variable, y una dirección es el par (segmento, desplazamiento).

La ventaja es que la unidad de protección y de compartición coincide con la unidad lógica: el segmento de código de una biblioteca compartida se protege y se comparte como un todo. La desventaja es que vuelve la fragmentación externa, porque los segmentos son de tamaño variable.

La combinación **segmentación paginada** aplica las dos: se segmenta por significado y cada segmento se pagina. Es lo que hacía x86 en 32 bits. En x86-64 la segmentación quedó reducida a casi nada —las bases de los segmentos se fuerzan a cero— y solo sobreviven `FS` y `GS`, que se usan para el almacenamiento local al hilo.

3.4 Memoria virtual

La idea: un proceso no necesita estar entero en memoria para ejecutarse. Basta con la parte que está usando. Lo que permite:

- Ejecutar programas mayores que la memoria física.
- Aumentar el grado de multiprogramación, porque cada proceso ocupa menos.
- Cargar más rápido, porque no hace falta leer el ejecutable entero.

3.4.1 Paginación bajo demanda

Las páginas se traen cuando se referencian, no antes. Al arrancar, la tabla de páginas del proceso tiene todas las entradas marcadas como no presentes; el primer acceso a cada una provoca un **fallo de página**.

El tratamiento:

1. La MMU ve el bit de presencia a cero y genera una excepción.
2. El núcleo comprueba si la dirección es válida para ese proceso. Si no lo es, la señal es `SIGSEGV` y el proceso muere.
3. Si es válida, busca un marco libre. Si no lo hay, aplica el algoritmo de reemplazo.
4. Programa la lectura desde disco y bloquea el proceso.
5. Cuando la transferencia termina, actualiza la tabla de páginas y devuelve el proceso a la cola de listos.
6. Se reejecuta **la instrucción que falló**, no la siguiente.

Ese último punto obliga a que las instrucciones sean reiniciables, y no todas lo son de forma trivial: una instrucción que copia bloques y modifica registros de índice sobre la marcha puede haber dejado el estado a medias. Los procesadores lo resuelven comprobando de antemano que todas las páginas implicadas están presentes, o guardando el estado intermedio en registros ocultos.

El coste de un fallo de página domina el rendimiento. Con un tiempo de acceso a memoria de 100 ns y un fallo servido en 8 ms, el tiempo de acceso efectivo con probabilidad de fallo p es

$$T_{ef} = (1 - p) \cdot 100 \text{ ns} + p \cdot 8 \text{ ms}$$

Para que la degradación se quede por debajo del 10 % hace falta $p < 2{,}5 \cdot 10^{-6}$: menos de un fallo cada 400 000 accesos. La memoria virtual funciona porque los programas tienen **localidad**, no porque el mecanismo sea barato.

3.4.2 Copia al escribir

Ya apareció con `fork`. Padre e hijo comparten los marcos, marcados de solo lectura aunque la región sea escribible. El primer intento de escritura provoca una excepción, el núcleo duplica esa página, la marca escribible en el proceso que escribió y deja al otro con la original. Solo se copia lo que se modifica.

El mismo mecanismo sostiene las páginas a cero: todas las páginas de BSS de todos los procesos apuntan a un único marco lleno de ceros hasta que alguien escribe.

3.4.3 Algoritmos de reemplazo

Cuando no hay marco libre hay que elegir víctima. Se evalúan sobre una cadena de referencias contando fallos de página.

Óptimo (OPT). Expulsa la página que tardará más en volver a usarse. No es implementable —exige conocer el futuro— y sirve como cota inferior contra la que medir a los demás.

FIFO. Expulsa la más antigua. Barato y malo: expulsa páginas muy usadas solo por llevar tiempo. Además sufre la **anomalía de Belady**, que es el resultado más contraintuitivo del tema: darle más marcos puede producir más fallos. Con la cadena `1 2 3 4 1 2 5 1 2 3 4 5`, FIFO da 9 fallos con tres marcos y 10 con cuatro.

LRU. Expulsa la que lleva más tiempo sin usarse. Es una buena aproximación al óptimo, y pertenece a la familia de algoritmos de pila, que por construcción no sufren la anomalía de Belady. Su problema es el coste: una implementación exacta exige actualizar una marca de tiempo o una lista enlazada en **cada acceso a memoria**, lo que solo es viable en hardware que no existe.

Las aproximaciones que sí se implementan:

- **Bit de referencia.** El hardware lo pone a uno en cada acceso; el sistema lo limpia periódicamente. Distingue usada de no usada, pero no ordena.
- **Segunda oportunidad, o reloj.** FIFO con bit de referencia: si la candidata lo tiene a uno, se le pone a cero y se pasa a la siguiente en vez de expulsarla. Las páginas se recorren en círculo, de donde el nombre.
- **Reloj mejorado.** Considera el par (referenciado, modificado) y prefiere expulsar las no modificadas, que no hay que escribir a disco.
- **Envejecimiento.** Un registro de desplazamiento por página, al que se le inyecta el bit de referencia por la izquierda. Aproxima el orden de LRU con coste acotado.

Comparación sobre la cadena `7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1` con tres marcos:

Algoritmo	Fallos
Óptimo	9
LRU	12
FIFO	15

3.4.4 Asignación de marcos

Cuántos marcos recibe cada proceso. La asignación **equitativa** reparte por igual e ignora que los procesos tienen tamaños muy distintos; la **proporcional** reparte según el tamaño del espacio lógico; la **por prioridad** favorece a los procesos prioritarios.

Y una decisión transversal: el reemplazo **local** obliga a cada proceso a elegir víctima entre sus propios marcos, y el **global** le permite quitárselo a otro. El global aprovecha mejor la memoria y hace que el rendimiento de un proceso dependa del comportamiento de los demás, lo que lo vuelve difícil de reproducir.

3.4.5 Hiperpaginación

Si un proceso no tiene marcos suficientes para su conjunto de páginas activas, falla, expulsa una página que necesita enseguida, vuelve a fallar y así indefinidamente. Es la **hiperpaginación** (*thrashing*): el sistema pasa más tiempo moviendo páginas que ejecutando.

Con reemplazo global se realimenta sola, y de la peor manera. La CPU baja de ocupación porque todos los procesos están bloqueados esperando disco; el planificador de largo plazo interpreta que hay capacidad libre y admite más procesos; los nuevos procesos piden marcos y la situación empeora. La ocupación de CPU cae en vertical mientras el grado de multiprogramación sube.

Dos modelos para evitarlo:

- **Conjunto de trabajo.** El conjunto de páginas referenciadas en las últimas Δ referencias. Si la suma de los conjuntos de trabajo de todos los procesos supera los marcos disponibles, hay que suspender alguno. La elección de Δ es empírica: demasiado pequeño no cubre la localidad, demasiado grande solapa varias.
- **Frecuencia de fallos.** Se mide directamente la tasa de fallos de cada proceso y se le dan más marcos si supera un umbral alto, o se le quitan si baja del umbral bajo. Ataca el síntoma directamente y es más fácil de medir.

3.4.6 Archivos proyectados en memoria

`mmap()` asocia un archivo a un rango de direcciones del proceso. A partir de ahí el archivo se lee y se escribe con accesos a memoria, y el sistema de paginación se encarga del resto: el primer acceso a cada página provoca un fallo que la trae de disco, y las escrituras se propagan al archivo según el modo.

Frente a `read` y `write`:

- No hay copia entre el búfer del núcleo y el del usuario.
- No hay una llamada al sistema por operación.
- Varios procesos que proyectan el mismo archivo en modo compartido comparten los marcos físicos, lo que sirve a la vez como memoria compartida.

Y los inconvenientes, que también los tiene: el coste por fallo de página se paga aun cuando el acceso es secuencial, no hay forma limpia de gestionar un error de entrada/salida —llega como `SIGBUS`, no como un valor de retorno—, y cambiar el tamaño del archivo mientras está proyectado

es terreno resbaladizo.

Es el mecanismo con el que se cargan los ejecutables y las bibliotecas compartidas: el núcleo no lee el binario, lo proyecta. El tratamiento en Linux, con el detalle de las estructuras implicadas, está en Mauerer, 2008; la interfaz y sus casos límite, en Kerrisk, 2010.

Capítulo 4

Gestión de archivos

Tema 4 del programa. La interfaz que el sistema de archivos ofrece a las aplicaciones, cómo se estructura por dentro y qué estructuras de datos lo sostienen en disco.

4.1 Interfaz de los sistemas de archivos

4.1.1 El archivo

Un archivo es una secuencia de bytes con nombre y almacenamiento persistente. Unix no impone más estructura: no distingue entre archivos de texto y binarios, ni conoce registros. Toda interpretación corre por cuenta de la aplicación. Otros sistemas sí impusieron estructura —registros de longitud fija, claves indexadas— y acabaron obligando a las aplicaciones a rodearla.

Los atributos que el sistema guarda de cada archivo:

Atributo	Contenido
Identificador	número de i-nodo, único dentro del sistema de archivos
Tipo	regular, directorio, enlace simbólico, dispositivo, tubería con nombre, socket
Tamaño	en bytes
Propietario	usuario y grupo
Permisos	lectura, escritura y ejecución para propietario, grupo y resto
Marcas de tiempo	acceso, modificación del contenido, modificación del i-nodo
Enlaces	cuántos nombres apuntan a él

El nombre no está en la lista, y eso es deliberado: en Unix el nombre no es un atributo del archivo, sino una entrada de directorio que apunta a él. Un archivo puede tener varios nombres o ninguno.

4.1.2 Operaciones

Las llamadas al sistema básicas, con la semántica que conviene retener:

Servicio	Qué hace	Detalle que suele fallar
`open`	abre y devuelve un descriptor	comprueba permisos solo aquí, no en cada `read`

Servicio	Qué hace	Detalle que suele fallar
`read`, `write`	transfieren desde el desplazamiento actual	pueden transferir menos de lo pedido, y hay que reintentar
`lseek`	mueve el desplazamiento	permite pasar del final y crear un hueco
`close`	libera el descriptor	no garantiza que los datos estén en disco
`fsync`	fuerza la escritura a disco	esto sí lo garantiza; `close` no
`unlink`	borra un nombre	el archivo sobrevive mientras algún proceso lo tenga abierto
`stat`	consulta atributos	sin abrir el archivo

Dos comportamientos que causan errores reales:

- **`write` puede escribir menos de lo pedido** sin que haya error. Ignorar el valor de retorno pierde datos en silencio, sobre todo con tuberías y sockets.
- **`close` no garantiza persistencia.** Los datos están en la caché del núcleo. Un corte de corriente los pierde. Solo `fsync` obliga a bajarlos, y en un archivo recién creado hay que sincronizar además el directorio que lo nombra, o el archivo puede quedar sin nombre.

4.1.3 Descriptores

Al abrir un archivo el núcleo maneja tres estructuras encadenadas, y confundirlas lleva a razonar mal sobre `fork` y sobre `dup`:

1. **Tabla de descriptores**, una por proceso. Un vector de punteros indexado por el número que devuelve `open`. Los descriptores 0, 1 y 2 son entrada, salida y error estándar por convenio.
2. **Tabla de archivos abiertos**, del sistema. Una entrada por cada `open`, con el desplazamiento actual y el modo de apertura.
3. **Tabla de i-nodos**, del sistema. Una entrada por archivo distinto en uso.

Las consecuencias:

- `fork` duplica la tabla de descriptores, así que padre e hijo **comparten el desplazamiento**: si uno lee, el otro continúa desde donde el primero se quedó.
- Dos `open` del mismo archivo dan dos entradas distintas en la segunda tabla, con desplazamientos independientes, y una sola en la tercera.
- `dup2` hace que dos descriptores apunten a la misma entrada de la segunda tabla. Por eso redirige: el descriptor 1 pasa a apuntar donde apunta el archivo abierto.

4.1.4 Directorios

Un directorio es un archivo cuyo contenido es una lista de pares (nombre, i-nodo). Que sea un archivo es lo que permite que el sistema de archivos sea un grafo y no una tabla plana.

La organización del directorio determina el coste de buscar un nombre:

Organización	Búsqueda	Uso
Lista lineal	$O(n)$	sistemas antiguos, directorios pequeños
Tabla *hash*	$O(1)$ en el caso medio	ext3 y ext4 con `dir_index`
Árbol B	$O(\log n)$	XFS, Btrfs, NTFS

La diferencia se nota: un directorio con cien mil entradas y lista lineal convierte cada `open` en un recorrido completo.

4.1.5 Enlaces

Dos formas de dar más de un nombre a un archivo, y no son intercambiables:

	Enlace duro	Enlace simbólico
Qué guarda	el mismo número de i-nodo	una ruta, como texto
Cruza sistemas de archivos	no	sí
Puede apuntar a un directorio	no, salvo `.` y `..`	sí
Si se borra el destino	el archivo sigue vivo	queda colgado
Cuenta en el contador de enlaces	sí	no

Que un enlace duro no pueda apuntar a un directorio evita ciclos en el grafo, y con ellos que un recorrido no termine y que el contador de referencias no llegue nunca a cero. Los enlaces simbólicos sí permiten ciclos, y por eso el núcleo limita a cuarenta el número de enlaces que resuelve al recorrer una ruta.

Un archivo se borra del disco cuando su contador de enlaces llega a cero **y** ningún proceso lo tiene abierto. De ahí el patrón de crear un temporal y desenlazarlo inmediatamente: el archivo existe mientras el proceso viva y desaparece solo cuando termina, incluso si termina de forma anómala.

4.1.6 Montaje

Un sistema de archivos se incorpora al árbol global sobre un directorio existente, el punto de montaje, cuyo contenido queda oculto mientras dure el montaje. Unix presenta así un único árbol con raíz `/`, en lugar de una letra por volumen.

Los espacios de nombres de montaje de Linux permiten que cada proceso vea un árbol distinto. Es la base de los contenedores: no hay virtualización de por medio, solo una vista diferente del mismo núcleo.

4.1.7 Permisos

El modelo Unix clásico son nueve bits, en tres grupos de tres, para propietario, grupo y resto. Además:

- **`setuid`**: el ejecutable corre con los privilegios de su propietario, no de quien lo lanza. Es lo que permite que `passwd` escriba en un archivo que el usuario no puede tocar, y la fuente histórica de escaladas de privilegios.
- **`setgid`**: lo mismo con el grupo; sobre un directorio, hace que lo que se cree dentro herede el grupo.
- **Bit pegajoso**: sobre un directorio, solo el propietario de un archivo puede borrarlo. Es lo que hace que `/tmp` sea utilizable por todos sin que nadie pueda borrar los archivos de otro.

El modelo es demasiado grueso para muchos casos —no permite dar acceso a un usuario concreto que no sea el propietario—, y de ahí las listas de control de acceso, que asocian permisos a usuarios y grupos arbitrarios.

4.2 Diseño software del sistema de archivos

4.2.1 Capas

De arriba abajo:

1. **Llamadas al sistema.** `open`, `read`, `write`.
2. **Sistema de archivos virtual (VFS).** Una interfaz común que oculta qué sistema de archivos concreto hay debajo.
3. **Sistema de archivos concreto.** ext4, XFS, Btrfs, NFS.
4. **Caché de bloques.** Mantiene en memoria los bloques recientes.
5. **Planificador de bloque y manejador del dispositivo.**

El **VFS** es la pieza que merece atención. Define cuatro objetos —superbloque, i-nodo, entrada de directorio y archivo— y una tabla de operaciones por objeto. Cada sistema de archivos concreto rellena esas tablas. El resultado es que `read` funciona igual sobre ext4, sobre NFS y sobre `/proc`, que no es un sistema de archivos sobre disco sino una vista de las estructuras del núcleo.

Es el mismo patrón de polimorfismo que un lenguaje orientado a objetos resuelve con clases abstractas, escrito con punteros a función porque el núcleo está en C.

4.2.2 La caché de bloques

Toda lectura pasa por una caché en memoria. La política es escritura diferida: `write` marca el bloque como sucio y vuelve; un hilo del núcleo lo baja a disco más tarde.

- **A favor:** las escrituras repetidas al mismo bloque se agrupan, y las que se sobrescriben antes de bajar no llegan a hacerlo nunca.
- **En contra:** un corte de corriente pierde lo que no se haya bajado, y el orden en que los bloques llegan al disco no es el orden en que se escribieron.

La alternativa, escritura inmediata, es más segura y mucho más lenta. Los sistemas ofrecen las dos y dejan elegir por montaje o por archivo.

4.2.3 Registro por diario

El problema que resuelve: una operación como crear un archivo toca varias estructuras —mapa de bits de i-nodos, i-nodo, bloque de datos, entrada de directorio— y no hay forma de escribirlas todas a la vez. Un corte a mitad deja el sistema de archivos inconsistente: i-nodos asignados que nadie nombra, o nombres que apuntan a i-nodos libres.

La solución clásica era `fsck`, que recorría el sistema entero al arrancar. Con discos de terabytes eso pasó a durar horas.

El **diario** escribe primero, en un área reservada y de forma secuencial, lo que va a hacer; luego lo hace; y al terminar marca la transacción como completada. Tras un corte, basta con releer el diario: las transacciones completas se reaplican y las incompletas se descartan. El tiempo de recuperación pasa a depender del tamaño del diario, no del disco.

Los tres modos de ext4, de menos a más seguro:

Modo	Al diario van	Consecuencia
`writeback`	solo metadatos, sin orden con los datos	el archivo puede quedar con metadatos nuevos y datos viejos
`ordered`	solo metadatos, pero los datos se bajan antes	el compromiso por omisión
`journal`	metadatos y datos	todo se escribe dos veces, y va a la mitad de velocidad

Una alternativa es **copia al escribir**: nunca se sobrescribe un bloque vivo, se escribe una versión nueva y se actualiza el puntero de la raíz. Es lo que hacen ZFS y Btrfs, y da instantáneas casi gratis, porque conservar una versión antigua es no liberar sus bloques.

4.3 Implementación de los sistemas de archivos

4.3.1 Asignación de bloques

Cómo se registran los bloques que ocupa un archivo:

Contigua. Bloque inicial y longitud. Lectura secuencial óptima y acceso directo trivial; a cambio, fragmentación externa y la imposibilidad de crecer si el vecino está ocupado. Sobrevive donde el contenido no cambia: sistemas de archivos de solo lectura y medios ópticos.

Enlazada. Cada bloque apunta al siguiente. No hay fragmentación externa ni límite de crecimiento, pero el acceso directo obliga a recorrer la cadena y un puntero corrupto pierde el resto del archivo. La variante FAT saca los punteros a una tabla en memoria, lo que arregla el acceso directo y hace que el tamaño de esa tabla limite el del volumen.

Indexada. Un bloque índice con los punteros a todos los bloques del archivo. Acceso directo en un salto, y el coste es un bloque por archivo aunque el archivo sea diminuto.

Extensiones. En vez de un puntero por bloque, tríos (bloque inicial, primer bloque lógico, longitud). Un archivo grande y contiguo se describe con unas pocas extensiones en lugar de miles de punteros. Es lo que usan ext4, XFS y NTFS.

4.3.2 El i-nodo

La estructura que representa un archivo en Unix. Contiene los atributos y los punteros a los bloques, organizados como índice multinivel:

Punteros	Cuántos	Cubren, con bloques de 4 KiB
Directos	12	48 KiB
Indirecto simple	1	4 MiB
Indirecto doble	1	4 GiB
Indirecto triple	1	4 TiB

El diseño está calibrado a la distribución real de tamaños: la inmensa mayoría de los archivos son pequeños y caben en los punteros directos, con un solo acceso. Los archivos grandes son pocos y pueden permitirse los saltos adicionales. Es un esquema deliberadamente asimétrico, y por eso funciona.

Un **archivo disperso** es el caso en que algunos punteros están a cero: hay huecos que no ocupan bloques y que se leen como ceros. Se crean saltando con `lseek` más allá del final. Por eso el espacio que un archivo ocupa puede ser mucho menor que su tamaño, y por eso copiarlo sin cuidado lo materializa entero.

4.3.3 Gestión del espacio libre

Método	Cómo	Cuándo conviene
Mapa de bits	un bit por bloque	encontrar bloques contiguos es barato; el mapa ocupa espacio fijo
Lista enlazada	cada bloque libre apunta al siguiente	no ocupa espacio extra; encontrar contigüidad es imposible
Agrupación	bloques que guardan direcciones de bloques libres	compromiso entre los dos
Recuento	pares (primer bloque, cuántos consecutivos)	eficiente si el espacio libre está agrupado

Con bloques de 4 KiB, el mapa de bits de un disco de 1 TiB ocupa 32 MiB: cabe en memoria y por eso es lo habitual. Los sistemas de archivos modernos lo dividen en grupos de bloques con su propio mapa, para no serializar todas las asignaciones sobre una única estructura.

4.3.4 Planificación de peticiones de disco

En un disco mecánico, el tiempo de servicio son tres sumandos: búsqueda —desplazar el brazo—, latencia rotacional y transferencia. El primero domina y es el único sobre el que el planificador puede actuar, reordenando la cola de peticiones.

Algoritmo	Criterio	Problema
FCFS	orden de llegada	movimiento del brazo desordenado
SSTF	la petición más cercana a la posición actual	inanición de las peticiones lejanas
SCAN, el ascensor	barre hasta un extremo y vuelve	los extremos esperan más
C-SCAN	barre en un solo sentido y salta al principio	tiempos de espera más uniformes
LOOK, C-LOOK	igual, pero sin llegar al extremo si no hay peticiones	las versiones que se implementan de verdad

Sobre una cola `98 183 37 122 14 124 65 67` con el brazo en la pista 53, el recorrido total es de 640 pistas con FCFS, 236 con SSTF y 208 con C-LOOK.

En un SSD todo esto sobra. No hay brazo ni rotación, y el tiempo de acceso no depende de la dirección, así que reordenar por cercanía no aporta nada. El planificador `none` es el adecuado, y los problemas se desplazan a otro sitio: la escritura solo puede hacerse sobre celdas borradas, el

borrado se hace en bloques mucho mayores que la página de escritura, y las celdas se desgastan. De ahí la capa de traducción interna del dispositivo, la recogida de basura y el nivelado de desgaste, y de ahí `TRIM`, que es cómo el sistema de archivos le dice al dispositivo qué bloques ya no contienen nada útil.

4.3.5 Consistencia y comprobación

`fsck` recorre el sistema de archivos comprobando invariantes: que cada bloque esté en un solo archivo o en la lista de libres, que el contador de enlaces de cada i-nodo coincida con las entradas de directorio que lo nombran, que no haya directorios inalcanzables. Lo que encuentra sin dueño va a `lost+found`.

Con diario deja de ser necesario en el caso normal, pero sigue siendo la única respuesta ante corrupción por hardware defectuoso: el diario garantiza que las operaciones son atómicas, no que el disco escriba lo que se le manda. Descripciones detalladas de estas estructuras y de su evolución están en Bach, 1986 y en Vahalia, 1996, y el tratamiento moderno en Tanenbaum, 2009.

Capítulo 5

Gestión de entradas y salidas

Tema 5 del programa. Cómo se comunica el procesador con los dispositivos, cómo se organiza el software que los maneja y cómo se consigue que una aplicación no tenga que saber sobre qué dispositivo escribe.

5.1 El problema

La entrada/salida es la parte del sistema operativo con más código y menos uniformidad, y por una razón estructural: los dispositivos no se parecen entre sí en nada.

Dimensión	Extremos
Velocidad	teclado, unos bytes por segundo; NVMe, gigabytes por segundo
Unidad de transferencia	carácter frente a bloque
Modo de acceso	secuencial frente a directo
Sentido	solo lectura, solo escritura, ambos
Sincronía	síncrono frente a asíncrono
Compartición	dedicado frente a compartible

Cinco órdenes de magnitud separan al teclado del disco de estado sólido. Escribir una capa que los presente igual a las aplicaciones, sin perder el rendimiento de los rápidos ni bloquear el sistema por los lentos, es el trabajo de este tema.

5.2 Comunicación con el dispositivo

5.2.1 El controlador

El procesador no habla con el dispositivo, sino con su **controlador**: un circuito con registros que el procesador lee y escribe. Los registros son siempre de las mismas cuatro clases:

Registro	Sentido	Contenido
Estado	lectura	ocupado, listo, error
Control	escritura	qué operación hacer
Datos de entrada	lectura	lo que el dispositivo entrega
Datos de salida	escritura	lo que se le manda

5.2.2 Cómo se accede a los registros

Dos esquemas:

- **Espacio de entrada/salida separado.** Instrucciones específicas —`in` y `out` en x86— y un espacio de direcciones propio. La separación protege por construcción: son instrucciones privilegiadas.
- **Proyección en memoria.** Los registros aparecen como direcciones de memoria ordinarias, y se leen y escriben con instrucciones normales. Es lo dominante, porque no exige instrucciones especiales y permite escribir manejadores en C.

La proyección en memoria obliga a una precaución: esas direcciones **no se pueden cachear**, porque su valor cambia sin que el procesador escriba, y el compilador no puede reordenar ni eliminar los accesos. De ahí que los manejadores usen funciones específicas de lectura y escritura con barreras, y no punteros ordinarios.

5.2.3 Tres formas de transferir

Sondeo (*polling*). El procesador lee el registro de estado en un bucle hasta que el dispositivo está listo. Simple y con la latencia más baja posible; a cambio consume una CPU entera esperando. Solo compensa cuando la espera es brevísima o el sistema no tiene nada mejor que hacer, y por eso los manejadores de red de alto rendimiento vuelven a él bajo carga alta: con millones de paquetes por segundo, una interrupción por paquete cuesta más que el sondeo.

Interrupciones. El dispositivo avisa cuando termina, y mientras tanto el procesador ejecuta otra cosa. Es el esquema general. Su coste es la latencia de la interrupción y el cambio de contexto, y su límite la **tormenta de interrupciones**: un dispositivo muy rápido puede interrumpir tan a menudo que el sistema no avance. Los manejadores modernos se defienden alternando entre interrupciones y sondeo según la carga.

DMA. Un controlador de acceso directo a memoria transfiere entre el dispositivo y la memoria sin pasar por el procesador. La CPU programa la dirección, el tamaño y el sentido, y recibe una sola interrupción cuando todo ha terminado. Es imprescindible para cualquier dispositivo de bloque: sin DMA, copiar un megabyte costaría un cuarto de millón de accesos del procesador.

El DMA obliga a coherencia de caché: el dispositivo escribe en memoria sin que las cachés del procesador se enteren. En arquitecturas con cachés coherentes lo resuelve el hardware; en las demás, el manejador tiene que invalidar o vaciar explícitamente el rango antes y después de la transferencia. Es una de las fuentes clásicas de errores intermitentes en manejadores.

Y un problema de seguridad: un dispositivo con DMA puede leer y escribir toda la memoria física, lo que convierte un puerto externo en una vía de ataque. La respuesta es la **IOMMU**, que hace con los dispositivos lo que la MMU con los procesos: traduce y comprueba permisos de las direcciones que emiten.

5.2.4 El tratamiento de la interrupción

La rutina de servicio se ejecuta en un contexto incómodo: no pertenece a ningún proceso, no puede bloquearse y no puede dormir esperando memoria. Cuanto más dure, más se retrasa todo lo demás.

De ahí el reparto en dos mitades, que todos los sistemas hacen con nombres distintos:

	Mitad superior	Mitad inferior
Cuándo	inmediata, con interrupciones inhibidas	diferida
Qué hace	reconoce la interrupción y guarda los datos	el procesamiento real
Restricciones	no puede dormir ni bloquearse	puede planificarse como el resto
En Linux	manejador registrado con `request_irq`	`softirq`, `tasklet`, cola de trabajo

Una interrupción puede compartir línea con otros dispositivos, así que la rutina tiene que empezar comprobando si el dispositivo que atiende es realmente el que interrumpió; si no, devuelve que no era suya. Con MSI y MSI-X, que envían la interrupción como una escritura en memoria en vez de por una línea física, la compartición desaparece y cada cola de un dispositivo puede tener su propio vector, dirigido a un núcleo distinto.

5.3 Arquitectura software del sistema de entrada/salida

Cuatro capas, de arriba abajo:

1. **Software de usuario.** La biblioteca de C con su propio búfer. `printf` acumula y llama a `write` cuando el búfer se llena o llega un salto de línea, y por eso la salida de un programa que aborta puede perderse: estaba en ese búfer, no en el núcleo.
2. **Software independiente del dispositivo.** Nombrado, protección, tamaño de bloque uniforme, almacenamiento intermedio, asignación y liberación, informe de errores. Es la capa que hace que `read` signifique lo mismo para todos.
3. **Manejadores.** El código específico de cada dispositivo. Es donde vive el conocimiento del hardware concreto, y donde está la mayor parte del código del núcleo.
4. **Rutinas de interrupción.**

El **objetivo de la capa 2** es la independencia del dispositivo: que el mismo programa funcione leyendo de un archivo, de una tubería o de un terminal. Unix lo consigue con una decisión de diseño muy simple, que es el apartado siguiente.

5.3.1 Almacenamiento intermedio

Por qué se copian los datos en vez de transferirlos directamente:

- **Desacoplar velocidades.** El productor y el consumidor no van al mismo ritmo.
- **Adaptar unidades de transferencia.** La aplicación escribe 10 bytes; el dispositivo trabaja con bloques de 4096.
- **Semántica de copia.** Cuando `write` vuelve, la aplicación puede reutilizar su búfer aunque el dato aún no esté en el disco.

Y el coste: cada copia consume ancho de banda de memoria. En un servidor que sirve archivos, los datos se copian del disco a la caché del núcleo, de ahí al búfer de la aplicación, de ahí al búfer del socket y de ahí a la tarjeta de red. Las técnicas de **copia cero** —`sendfile`, `splice`, proyección en memoria— existen para eliminar los pasos intermedios, y con ellos el paso por el espacio de usuario.

Un caso particular es el **spooling**: para dispositivos que no se pueden compartir, como una

impresora, los trabajos se acumulan en disco y un proceso demonio los sirve de uno en uno. El dispositivo dedicado se convierte así en compartible.

5.3.2 Tratamiento de errores

Los errores de entrada/salida son la norma, no la excepción, y se tratan en niveles: el controlador reintenta, el manejador reintenta un número acotado de veces, y solo si todo falla el error sube a la aplicación como código de retorno. Un sector defectuoso se remapea de forma transparente por el propio disco, que mantiene una reserva para eso; la aplicación no se entera hasta que la reserva se agota.

5.4 Archivos de dispositivos

Aquí está la decisión de diseño que define a Unix: **los dispositivos se presentan como archivos**. Viven en `/dev`, tienen i-nodo, propietario y permisos, y se manipulan con `open`, `read`, `write` y `close`.

Las consecuencias:

- Un programa que copia archivos copia también de un dispositivo a otro sin una línea de código adicional.
- Los permisos de acceso a un dispositivo son los permisos de un archivo, con el mismo modelo y las mismas herramientas.
- La redirección del intérprete de órdenes funciona igual con dispositivos.

Cada archivo de dispositivo lleva dos números: el **mayor**, que identifica al manejador, y el **menor**, que identifica la unidad concreta dentro de ese manejador.

Clase	Cómo se transfiere	Ejemplos
Carácter	flujo de bytes, sin caché de bloques	terminales, `/dev/null`, `/dev/random`, ratón
Bloque	bloques de tamaño fijo, con caché y planificación	discos, unidades ópticas
Red	fuera del esquema: usa sockets, no `/dev`	interfaces de red

Las interfaces de red son la excepción reconocida al principio de que todo es un archivo. No aparecen en `/dev` porque su modelo de acceso —paquetes con direcciones, sin nombre en el sistema de archivos— no encajaba, y Berkeley resolvió el problema con una abstracción aparte, el socket. Plan 9 llevó el principio hasta el final e hizo que también la red fuera un sistema de archivos.

5.4.1 `ioctl` y sus sucesores

Lo que no cabe en `read` y `write` —cambiar la velocidad de un puerto serie, expulsar un disco, consultar la geometría— se hace con `ioctl`, una llamada con un código de operación y un puntero a una estructura que depende del código.

Es reconocidamente la parte fea de la interfaz: no tiene tipos, cada manejador inventa sus códigos, y validar el puntero es responsabilidad de cada uno. Las alternativas modernas —`sysfs`, `netlink`, `/proc`— exponen la configuración como archivos de texto o como mensajes con formato, y son preferibles cuando la operación no está en un camino crítico.

5.4.2 Dispositivos virtuales

No todo archivo de `/dev` corresponde a hardware:

- `/dev/null` descarta lo que se le escribe y devuelve fin de archivo al leer.
- `/dev/zero` entrega ceros indefinidamente.
- `/dev/random` y `/dev/urandom` entregan bytes del generador de números aleatorios del núcleo.
- Los dispositivos de bucle presentan un archivo como si fuera un disco, que es como se monta una imagen sin grabarla.

5.5 Manejadores de dispositivos

Un manejador es el módulo que traduce las operaciones genéricas del sistema en las órdenes concretas del controlador. Su interfaz con el núcleo es una estructura de punteros a función: `open`, `release`, `read`, `write`, `unlocked_ioctl`, `mmap`, `poll`. Rellenar esa estructura y registrarla es lo que convierte un módulo en un manejador.

5.5.1 Estructura

- **Inicialización.** Detectar el dispositivo, reservar memoria y líneas de interrupción, registrar el manejador.
- **Operaciones.** Se ejecutan en el contexto del proceso que llamó, así que pueden dormir.
- **Rutina de interrupción.** No se ejecuta en el contexto de ningún proceso, así que no puede dormir ni copiar a memoria de usuario.
- **Descarga.** Liberar en orden inverso al de la inicialización.

La regla que más errores previene: **el contexto determina lo que se puede hacer**. Copiar a memoria de usuario puede provocar un fallo de página, y un fallo de página puede dormir; hacerlo desde una rutina de interrupción bloquea el sistema.

5.5.2 Módulos cargables

Compilar cada manejador dentro del núcleo obligaría a recompilar y reiniciar para añadir hardware. Los módulos se cargan y descargan en ejecución, se enlazan contra los símbolos que el núcleo exporta y pasan a ejecutarse en modo núcleo con todos los privilegios.

Eso último es lo importante: un módulo no está aislado. Un fallo en un manejador es un fallo del núcleo. Por eso las arquitecturas de microkernel sacan los manejadores a espacio de usuario, y por eso Linux ofrece marcos como FUSE y UIO para escribir en espacio de usuario los que no necesitan rendimiento extremo.

5.5.3 Independencia del dispositivo

El mecanismo que la sostiene es el mismo que el VFS del tema anterior: una tabla de operaciones por dispositivo, rellenada por cada manejador, y una capa superior que solo conoce la tabla. Añadir un dispositivo nuevo es rellenar una estructura y registrarla; ninguna capa por encima cambia.

De ahí que el sistema pueda montar un sistema de archivos sobre un disco SATA, un NVMe o un archivo de imagen sin distinguirlos, y que `cat` funcione igual sobre un archivo, una tubería o un puerto serie.

5.5.4 Detección y nombrado dinámicos

Con hardware que se conecta y desconecta en caliente, los archivos de `/dev` no pueden ser estáticos. El núcleo publica los eventos, y un demonio de espacio de usuario —`udev` en Linux— crea y borra los nodos, aplica permisos y asigna nombres estables a partir de los atributos del dispositivo.

Los nombres estables resuelven un problema real: el orden en que se detectan los discos no está garantizado, así que `/dev/sda` puede ser un disco distinto en el siguiente arranque. Por eso los sistemas de archivos se montan por UUID o por etiqueta, y no por nombre de dispositivo. El detalle de la implementación en Linux está en Love, 2010, y la interfaz que ve el programador, en Kerrisk, 2010.

Capítulo 6

Mecanismos de seguridad

Tema 6 del programa. Qué hay que proteger y de qué, cómo se comprueba quién es alguien y cómo se decide lo que puede hacer una vez comprobado.

6.1 Objetivos de protección y amenazas

6.1.1 Los tres objetivos

Objetivo	Qué garantiza	Ejemplo de violación
Confidencialidad	la información solo la ve quien debe	lectura de un archivo ajeno
Integridad	la información solo la modifica quien debe	alteración de un registro
Disponibilidad	el servicio está cuando se necesita	agotamiento de recursos

Los tres se contraponen en la práctica. Cifrar mejora la confidencialidad y empeora la disponibilidad si se pierde la clave; replicar mejora la disponibilidad y multiplica los sitios desde donde se puede filtrar información.

6.1.2 Protección y seguridad

Dos palabras que no significan lo mismo:

- **Protección** son los mecanismos internos que controlan el acceso a los recursos del sistema. Es un problema técnico con solución técnica.
- **Seguridad** es el problema completo, e incluye el entorno: quién tiene acceso físico a la máquina, cómo se administra, qué hace el usuario cuando le piden la contraseña por teléfono.

Un sistema con protección perfecta puede ser inseguro. El sistema operativo solo puede resolver la primera mitad.

6.1.3 Principios de diseño

Los que Saltzer y Schroeder formularon en 1975 y siguen vigentes:

1. **Mínimo privilegio.** Cada componente con los permisos justos para su función, y ni uno más.
2. **Economía de mecanismo.** Lo simple se puede auditar; lo complejo, no.
3. **Valores por omisión seguros.** Denegar salvo permiso explícito. Una lista de lo prohibido siempre está incompleta.

4. **Mediación completa.** Comprobar cada acceso, no solo el primero.
5. **Diseño abierto.** La seguridad debe descansar en la clave, no en el secreto del mecanismo.
6. **Separación de privilegios.** Exigir dos condiciones independientes para las operaciones críticas.
7. **Menor mecanismo común.** Reducir lo que se comparte entre usuarios: lo compartido es un canal.
8. **Aceptabilidad psicológica.** Si el mecanismo estorba, se rodea. Una política de contraseñas imposible produce contraseñas apuntadas en un papel.

La **mediación completa** es la que explica una diferencia de diseño de Unix: los permisos de un archivo se comprueban en `open` y no en cada `read`, así que un cambio de permisos no afecta a un descriptor ya abierto. Es una desviación consciente del principio, aceptada por rendimiento.

6.1.4 El dominio de protección

Un proceso ejecuta dentro de un **dominio**, que es un conjunto de pares (objeto, operaciones permitidas). En Unix el dominio lo definen el identificador de usuario efectivo y los grupos.

El cambio de dominio ocurre al ejecutar un binario `setuid`, y es la operación más delicada del modelo: el proceso pasa a tener los privilegios del propietario del ejecutable. Un fallo en un programa `setuid` de root es una escalada de privilegios inmediata, y por eso la tendencia moderna es sustituirlos por **capacidades**, que trocean el privilegio de root en unas cuarenta piezas independientes. Un programa que solo necesita abrir un puerto bajo recibe `CAP_NET_BIND_SERVICE` y nada más.

6.1.5 Amenazas

Clase	Descripción
Caballo de Troya	programa con una función anunciada y otra oculta
Puerta trasera	acceso deliberado que salta la autenticación
Bomba lógica	código que actúa al cumplirse una condición
Escalada de privilegios	pasar de usuario sin privilegios a administrador
Denegación de servicio	agotar un recurso hasta que el servicio deja de responder
Canal encubierto	transmitir información por un medio no previsto para ello

Y las técnicas de explotación que atacan directamente al sistema operativo:

Desbordamiento de pila. Escribir más allá del final de un vector local sobrescribe la dirección de retorno del marco de llamada, y al volver la función el control salta donde el atacante quiera. Las contramedidas se han ido acumulando, y ninguna basta por sí sola:

Contramedida	Qué hace	Cómo se rodea
Canario	valor testigo antes de la dirección de retorno	fuga de información que lo revele
Pila no ejecutable (NX)	impide ejecutar lo escrito en la pila	reutilizar código ya presente (ROP)

Contramedida	Qué hace	Cómo se rodea
ASLR	aleatoriza la posición de pila, montículo y bibliotecas	fuga de direcciones, o fuerza bruta en 32 bits
RELRO	pone de solo lectura las tablas de enlace tras cargar	—

Condición de carrera en tiempo de comprobación. Un programa privilegiado comprueba que un archivo es del usuario y luego lo abre; entre las dos operaciones el atacante sustituye el archivo por un enlace simbólico. La única solución correcta es no separar comprobación y uso: abrir primero y comprobar sobre el descriptor, con `fstat` en vez de `stat`.

Canales laterales. Meltdown y Spectre no leen memoria protegida directamente: obligan al procesador a ejecutar especulativamente un acceso y después deducen el valor midiendo el tiempo de acceso a la caché. Son interesantes aquí porque rompen la frontera que el tema 1 daba por sólida: la protección de memoria seguía funcionando, y aun así la información se filtraba por un efecto secundario del hardware. Las mitigaciones —aislar las tablas de páginas del núcleo, vaciar predictores en cada transición— tienen un coste medible en cada llamada al sistema.

6.2 Autenticación

Comprobar que alguien es quien dice ser. Tres factores, que se combinan:

Factor	Basado en	Debilidad
Conocimiento	contraseña, PIN	se adivina, se reutiliza, se filtra
Posesión	tarjeta, llave física, generador de códigos	se pierde o se roba
Inherencia	huella, iris, voz	no se puede cambiar tras una filtración

Esa última debilidad es la que se subestima. Una contraseña comprometida se cambia; una huella dactilar, no.

6.2.1 Contraseñas

Nunca se almacenan. Se almacena el resultado de una función que no se puede invertir, y la comparación se hace sobre ese resultado.

La función no puede ser una función resumen criptográfica corriente. SHA-256 está diseñada para ser rápida, y esa velocidad juega a favor de quien prueba candidatos: una tarjeta gráfica calcula miles de millones por segundo. Lo que se usa son funciones deliberadamente lentas y con coste ajustable —bcrypt, scrypt, Argon2—, y las dos últimas además exigen memoria, lo que anula la ventaja del hardware especializado.

La **sal** es un valor aleatorio distinto para cada usuario que se concatena antes de aplicar la función. Sin ella, dos usuarios con la misma contraseña tienen el mismo resultado almacenado, y una tabla precalculada sirve para todos los sistemas del mundo a la vez. Con ella, cada contraseña hay que atacarla por separado. La sal no es secreta, y se guarda junto al resultado.

En Unix los resultados no están en `/etc/passwd`, que es legible por todos, sino en `/etc/shadow`, que solo lee root. La separación se introdujo cuando el crecimiento de la potencia de cálculo hizo

que dar acceso público a los resultados dejara de ser aceptable.

6.2.2 Ataques y defensas

Ataque	Descripción	Defensa
Fuerza bruta	probar todas las combinaciones	longitud, y retardo tras cada fallo
Diccionario	probar palabras y variantes frecuentes	función lenta, y comprobar contra listas filtradas
Tabla arcoíris	resultados precalculados	sal
Reutilización	probar credenciales filtradas de otro sitio	segundo factor
Suplantación	pantalla de acceso falsa	camino de confianza al sistema
Registro de teclas	capturar lo tecleado	segundo factor, y contraseñas de un solo uso

La política de caducidad obligatoria cada pocas semanas ha sido **retirada de las recomendaciones**: producía variaciones triviales de la contraseña anterior. Lo que sí se recomienda es longitud, comprobación contra listas de contraseñas filtradas y cambio solo cuando hay indicio de compromiso.

6.2.3 PAM

Los módulos de autenticación conectables separan el mecanismo de autenticación de las aplicaciones que lo usan. `login`, `sshd` y `sudo` no saben cómo se autentica: llaman a PAM, y la configuración decide si detrás hay contraseñas locales, un directorio LDAP, una tarjeta o un segundo factor. Es la separación entre mecanismo y política del tema 1 aplicada aquí, y evita que añadir un método nuevo obligue a tocar cada programa.

6.3 Mecanismos de autorización

Decidida la identidad, queda decidir lo que puede hacer.

6.3.1 La matriz de acceso

El modelo formal: una matriz con un dominio por fila, un objeto por columna y las operaciones permitidas en cada celda. Es completa y no se implementa nunca, por tamaño y dispersión. Lo que se implementa son sus dos proyecciones:

	Lista de control de acceso	Lista de capacidades
Se guarda por	columna, es decir, con el objeto	fila, es decir, con el sujeto
Responde barato a	quién puede acceder a este objeto	a qué puede acceder este sujeto
Revocar	fácil, se edita la lista del objeto	difícil, hay que perseguir las capacidades
Ejemplos	permisos y ACL de POSIX, NTFS	descriptores de archivo, seL4, `pledge`

Un descriptor de archivo abierto es exactamente una capacidad: se obtiene tras una comprobación, se puede pasar a otro proceso —por un socket de dominio Unix— y quien lo tiene puede usarlo sin volver a demostrar nada.

6.3.2 Control de acceso discrecional

Es el modelo Unix: el propietario de un objeto decide sus permisos. Se llama discrecional porque la decisión queda a discreción del usuario.

Su límite es estructural: nada impide que un usuario, voluntariamente o engañado, dé acceso a lo que no debía. Un caballo de Troya ejecutado por un usuario tiene todos los permisos de ese usuario, y puede regalar sus archivos.

6.3.3 Control de acceso obligatorio

La política la fija el administrador y el usuario no puede relajarla, ni sobre sus propios archivos. Cada objeto y cada sujeto llevan una etiqueta, y las reglas se aplican sobre las etiquetas.

El modelo **Bell-LaPadula**, orientado a confidencialidad, impone dos reglas que son menos obvias de lo que parecen:

- **No leer hacia arriba.** Un sujeto no lee objetos de nivel superior.
- **No escribir hacia abajo.** Un sujeto no escribe objetos de nivel inferior.

La segunda sorprende, y es la que de verdad protege: sin ella, un proceso con acceso a información clasificada podría copiarla a un archivo público. El modelo **Biba** es su simétrico para integridad, y sus reglas van en el sentido contrario: no leer hacia abajo, no escribir hacia arriba.

En Linux esto lo implementan SELinux y AppArmor a través de los módulos de seguridad del núcleo, que insertan una comprobación adicional después de la comprobación discrecional. Nunca la sustituyen: si los permisos clásicos deniegan, la etiqueta no rescata nada.

6.3.4 Control de acceso basado en roles

Los permisos se asignan a roles y los roles a usuarios. Escala mejor en organizaciones porque un cambio de puesto es un cambio de rol y no una revisión de cada objeto. `sudo` es su aproximación en Unix: en vez de repartir la contraseña de root, se autoriza a cada usuario un conjunto de órdenes concretas, y cada ejecución queda registrada.

6.3.5 Aislamiento

Cuando ni siquiera se confía en el proceso, se le limita lo que puede pedir:

Mecanismo	Qué acota
`chroot`	la raíz del sistema de archivos visible; no es una frontera de seguridad por sí solo
Espacios de nombres	qué procesos, montajes, red e identificadores ve
Grupos de control	cuánta CPU, memoria y ancho de banda consume
`seccomp`	qué llamadas al sistema puede ejecutar
Máquina virtual	la frontera más fuerte, y la más cara

Un contenedor es la combinación de los cuatro primeros. Que `chroot` no sea una frontera de

seguridad tiene una razón concreta: un proceso con privilegios puede salir de él, y el mecanismo se diseñó para aislar builds, no atacantes.

6.3.6 Registro y auditoría

Lo que no se registra no se puede investigar, y un registro que el atacante puede borrar no sirve. De ahí que los registros se envíen a una máquina distinta, y que se firmen o se encadenen para detectar manipulaciones.

Los tres extremos que hay que evitar: registrar tan poco que no se pueda reconstruir un incidente, registrar tanto que nadie mire, y registrar datos sensibles —contraseñas tecleadas donde no debían, tokens— que convierten el propio registro en un objetivo.

6.4 Seguridad en la administración

Cierra el tema y enlaza con la práctica 1. La mayor parte de los compromisos no explota un fallo del núcleo, sino una configuración:

- Servicios escuchando en la red que nadie usa.
- Actualizaciones sin aplicar sobre vulnerabilidades conocidas y publicadas.
- Permisos demasiado amplios, y directorios escribibles por todos en el camino de búsqueda de un programa privilegiado.
- Copias de seguridad que nunca se han probado restaurando.
- Cuentas de servicio con intérprete de órdenes y contraseña.

El principio del mínimo privilegio se aplica igual aquí: instalar lo que se usa, abrir lo que se necesita, dar los permisos justos. Las guías de administración que la asignatura recomienda —Nemeth et al., 2010 y Frisch, 2002— desarrollan esta parte, y el detalle de los mecanismos del núcleo está en Love, 2010.

Capítulo 7

Temario práctico

Las dos prácticas del programa. La primera administra el sistema desde fuera; la segunda lo programa desde dentro, con la interfaz que el tema 2 describía.

7.1 Práctica 1. Administración del sistema

7.1.1 Herramientas básicas

Usuarios y grupos. La información vive en tres archivos con formato de campos separados por dos puntos:

Archivo	Contiene	Quién lo lee
`/etc/passwd`	nombre, UID, GID, directorio, intérprete	todos
`/etc/shadow`	resultado de la función sobre la contraseña, caducidad	solo root
`/etc/group`	grupos y sus miembros	todos

Se editan con `useradd`, `usermod`, `groupadd` y `passwd`, no a mano: las herramientas bloquean los archivos y validan el formato. Para editarlos de verdad está `vipw`, que hace lo mismo.

Una cuenta de servicio se crea sin intérprete —`/usr/sbin/nologin`— y sin contraseña válida. Es la aplicación directa del mínimo privilegio: ese usuario existe para ser el propietario de unos procesos, no para iniciar sesión.

Permisos. `chmod` en octal o simbólico, `chown` y `chgrp`, y la máscara `umask`, que resta permisos a lo que se crea. Los bits especiales del tema 6 se ven aquí en su forma práctica: `find / -perm -4000` lista los binarios `setuid` del sistema, y esa lista debería ser corta y conocida.

Sistemas de archivos. `df` para el espacio, `du` para el reparto, `lsblk` para la topología de dispositivos, `mount` y `/etc/fstab` para el montaje. En `fstab` los sistemas se identifican por UUID y no por `/dev/sdX`, por lo que explicaba el tema 5: el orden de detección no está garantizado.

Opciones de montaje que cambian la seguridad de una partición: `noexec` impide ejecutar, `nosuid` desactiva los bits `setuid` y `setgid`, `nodev` ignora los archivos de dispositivo. Las tres juntas en `/tmp` cierran vías de escalada conocidas.

Procesos. `ps` para la foto, `top` y `htop` para el seguimiento, `kill` y `pkill` para las señales, `nice` y `renice` para la prioridad. El estado que `ps` muestra es el diagrama del tema 2: `R` en ejecución o listo, `S` bloqueado interrumpible, `D` bloqueado no interrumpible, `Z` zombi, `T` detenido.

Un proceso en `D` no responde ni a `SIGKILL`, y encontrarlo casi siempre apunta a un dispositivo o a un montaje de red que no responde, no al proceso.

Registros. `journalctl` en sistemas con systemd, y los archivos de `/var/log`. La rotación la gestiona `logrotate`, que comprime y descarta los antiguos para que el registro no llene la partición. Un `/var` lleno deja de registrar en silencio, que es exactamente cuando más falta hace.

7.1.2 Monitorización

Cuatro recursos, y una herramienta por recurso:

Recurso	Qué mirar	Con qué
CPU	uso por modo, carga media, cambios de contexto	`vmstat`, `mpstat`, `pidstat`
Memoria	libre real, caché, intercambio, fallos de página	`free`, `vmstat`, `/proc/meminfo`
Disco	operaciones por segundo, latencia, tiempo ocupado	`iostat`, `iotop`
Red	ancho de banda, conexiones, retransmisiones	`ss`, `ip -s`, `nload`

Dos lecturas que se malinterpretan siempre:

- **La memoria «libre» de `free` no es la disponible.** El núcleo usa la memoria no asignada como caché de bloques, y esa caché se libera en cuanto alguien necesita marcos. La columna que importa es `available`, no `free`. Una máquina sana tiene poca memoria libre, y eso está bien.
- **La carga media no es el porcentaje de CPU.** Es el número medio de procesos en estado listo o bloqueado no interrumpible. Con ocho núcleos, una carga de 8 es ocupación plena; con dos, es saturación. Y una carga alta con la CPU ociosa apunta a disco, no a procesador.

`/proc` y `/sys` son la fuente de todo lo anterior. Son sistemas de archivos virtuales —el VFS del tema 4 aplicado a las estructuras del núcleo—, así que la monitorización se reduce a leer archivos de texto.

Cuando algo va lento, el orden que evita perder el tiempo: primero mirar si falta un recurso —CPU, memoria, disco o red—, después qué proceso lo consume, y solo entonces por qué. `strace` muestra las llamadas al sistema que hace un proceso y `perf` dónde gasta los ciclos.

7.1.3 Automatización

Guiones de intérprete de órdenes. Un guion serio empieza igual:

```
#!/usr/bin/env bash
set -euo pipefail
IFS=$'\n\t'
```

`-e` aborta al primer fallo, `-u` convierte en error el uso de una variable no definida y `-o pipefail` propaga el fallo de cualquier etapa de una tubería, no solo de la última. Sin las tres, un guion de copia de seguridad falla a la mitad y devuelve éxito.

Tareas periódicas. `cron` con su formato de cinco campos, o los temporizadores de systemd, que se pueden consultar, tienen registro propio y manejan mejor el caso de una máquina apagada a la hora prevista.

```
# m  h  dom mon dow  orden
  30 3  *   *   *    /usr/local/bin/respaldo.sh
```

Una tarea de `cron` se ejecuta con un entorno mínimo: no hereda el `PATH` de la sesión ni las variables del perfil. Es la causa habitual de que un guion funcione al probarlo y falle a las tres de la mañana. Se resuelve usando rutas absolutas y redirigiendo la salida a un registro.

Servicios. Una unidad de systemd declara qué ejecutar, con qué usuario, qué hacer si falla y de qué depende. Y ofrece aislamiento declarativo, que es la forma barata de aplicar lo del tema 6:

```
[Service]
User=servicio
ExecStart=/usr/local/bin/servicio
Restart=on-failure
NoNewPrivileges=true
ProtectSystem=strict
PrivateTmp=true
```

Copias de seguridad. `tar` y `rsync` con la regla 3-2-1: tres copias, en dos medios distintos, una fuera del sitio. Y la parte que se salta todo el mundo: una copia no está probada hasta que se ha restaurado. Una restauración periódica de prueba es lo único que distingue una copia de seguridad de un archivo grande.

7.2 Práctica 2. Servicios del sistema mediante la API

7.2.1 Gestión y comunicación de procesos

Creación y espera. `fork`, `exec`, `wait` y `waitpid` con la semántica del tema 2. El estado que devuelve `waitpid` se interpreta con macros, no comparando: `WIFEXITED` y `WEXITSTATUS` para la terminación normal, `WIFSIGNALED` y `WTERMSIG` para la muerte por señal.

Tuberías. `pipe` devuelve dos descriptores, lectura y escritura. Montar una tubería entre dos programas es el ejercicio que reúne todo lo anterior:

```
int fd[2];
pipe(fd);

if (fork() == 0) {                  /* productor */
    dup2(fd[1], STDOUT_FILENO);
    close(fd[0]);
    close(fd[1]);
    execlp("ls", "ls", NULL);
    _exit(127);
}
if (fork() == 0) {                  /* consumidor */
    dup2(fd[0], STDIN_FILENO);
    close(fd[0]);
    close(fd[1]);
    execlp("wc", "wc", "-l", NULL);
    _exit(127);
}
close(fd[0]);
close(fd[1]);                       /* imprescindible en el padre */
```

```
wait(NULL);
wait(NULL);
```

Los `close` del padre no son limpieza opcional. Mientras quede **un solo** descriptor de escritura abierto en cualquier proceso, el lector no recibe fin de archivo y `wc` espera indefinidamente. Es el error más frecuente de la práctica, y no se manifiesta como un fallo sino como un programa colgado.

Escribir en una tubería sin lectores genera `SIGPIPE`, que por omisión mata al proceso. Ignorarla convierte el caso en un `write` que devuelve `EPIPE`, que es lo que un programa robusto quiere.

Señales. Se instalan con `sigaction`, no con `signal`, cuya semántica varía entre sistemas. Dentro de un manejador solo se pueden llamar funciones seguras —la lista está en la norma POSIX— y solo se pueden tocar variables declaradas `volatile sig_atomic_t`. Ni `printf` ni `malloc` son seguras: si la señal llega mientras el proceso estaba dentro de `malloc`, el manejador reentra en una estructura a medio modificar.

`SIGKILL` y `SIGSTOP` no se pueden capturar ni ignorar, y esa es justamente su razón de existir.

Otros mecanismos. Colas de mensajes, memoria compartida y semáforos POSIX (`mq_open`, `shm_open`, `sem_open`). La memoria compartida es el mecanismo más rápido —no hay copia— y el que más sincronización exige, porque no la lleva incorporada.

7.2.2 Manejo de archivos y directorios

`open`, `read`, `write`, `lseek`, `close`, `stat`, `opendir` y `readdir`, con las trampas del tema 4:

- Comprobar **siempre** el valor de retorno de `read` y `write`, que pueden transferir menos de lo pedido.
- `errno` solo es válido inmediatamente después de un fallo. Cualquier llamada intermedia, incluida `printf`, puede haberlo cambiado.
- Un recorrido recursivo de directorios tiene que saltarse `.` y `..`, o no termina, y no debe seguir los enlaces simbólicos salvo que sea lo que se quiere: un enlace a un ancestro es un ciclo.

7.2.3 Archivos proyectados en memoria

`mmap` y `munmap`. Un programa que cuenta las apariciones de un byte en un archivo grande se escribe sin un solo `read`:

```
int fd = open(ruta, O_RDONLY);
struct stat st;
fstat(fd, &st);

char *p = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
if (p == MAP_FAILED) { perror("mmap"); return 1; }

size_t n = 0;
for (off_t i = 0; i < st.st_size; i++) {
    if (p[i] == '\n') n++;
}

munmap(p, st.st_size);
close(fd);
```

`MAP_PRIVATE` da copia al escribir: los cambios no llegan al archivo. `MAP_SHARED` sí los propaga, y es lo que permite que dos procesos que proyectan el mismo archivo compartan memoria de verdad.

Un archivo de tamaño cero no se puede proyectar, y leer más allá del final del archivo dentro de la región proyectada produce `SIGBUS`, no `SIGSEGV`.

7.2.4 Gestión de memoria y tiempo

Memoria. `brk` y `sbrk` mueven el final del montículo y casi nunca se usan directamente: `malloc` los usa por debajo para bloques pequeños y `mmap` anónimo para los grandes. Que un `malloc` grande devuelva memoria no significa que haya marcos físicos reservados: Linux permite reservar más de lo que hay, y los marcos llegan al escribir. De ahí que un programa pueda morir por falta de memoria mucho después del `malloc` que la pidió.

Tiempo. `clock_gettime` con el reloj adecuado, que es la decisión que importa:

Reloj	Qué mide	Para qué
`CLOCK_REALTIME`	hora del día	fechas; **da saltos** al ajustarse
`CLOCK_MONOTONIC`	tiempo desde el arranque	medir intervalos
`CLOCK_PROCESS_CPUTIME_ID`	CPU consumida por el proceso	perfilado

Medir una duración con `CLOCK_REALTIME` produce intervalos negativos cuando el demonio de hora ajusta el reloj. Los intervalos se miden siempre con el monótono.

`nanosleep` duerme al menos lo pedido, nunca exactamente: al despertar el proceso pasa a la cola de listos y compite por la CPU, tal como decía el diagrama de estados. Y puede volver antes por una señal, devolviendo el tiempo que faltaba, así que hay que reintentar.

7.2.5 Cómo se entrega

Los programas se compilan con avisos y sin ellos:

```
gcc -Wall -Wextra -std=c11 -g -o programa programa.c
```

Y se comprueban con las herramientas que encuentran lo que la compilación no ve: `valgrind` para accesos inválidos y memoria no liberada, y los desinfectantes del compilador (`-fsanitize=address,undefined`) para lo mismo con menos sobrecoste. Un programa que funciona no es un programa correcto: casi todos los errores de memoria de esta práctica producen resultados correctos hasta que dejan de hacerlo.

El repertorio completo de servicios, con sus casos límite y sus diferencias entre sistemas, está en Kerrisk, 2010 y en Stevens y Rago, 2005; la parte de administración, en Nemeth et al., 2010.

Bibliografía

Bach, M. J. (1986). *The Design of the UNIX Operating System*. Prentice Hall.

Frisch, Æ. (2002). *Essential System Administration* (3.ª ed.). O'Reilly Media.

Kerrisk, M. (2010). *The Linux Programming Interface. A Linux and UNIX System Programming Handbook*. No Starch Press.

Love, R. (2010). *Linux Kernel Development* (3.ª ed.). Addison-Wesley Professional.

Mauerer, W. (2008). *Professional Linux Kernel Architecture*. Wiley.

Nemeth, E., Snyder, G., Hein, T. R., & Whaley, B. (2010). *UNIX and Linux System Administration Handbook* (4.ª ed.). Prentice Hall.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2006). *Fundamentos de Sistemas Operativos* (7.ª ed.). McGraw-Hill.

Stallings, W. (2018). *Operating Systems: Internals and Design Principles* (9.ª ed.). Pearson Education.

Stevens, W. R., & Rago, S. A. (2005). *Advanced Programming in the UNIX Environment* (2.ª ed.). Addison-Wesley Professional.

Tanenbaum, A. S. (2009). *Sistemas Operativos Modernos* (3.ª ed.). Pearson Prentice Hall.

Vahalia, U. (1996). *UNIX Internals. The New Frontiers*. Prentice Hall.