Modelo de ejecución dual: Kernel/User

1. El sistema operativo como árbitro: espacio usuario vs espacio kernel

Hoy quiero que empecemos por una de las ideas más fundamentales en todo el diseño de sistemas operativos: **la separación entre el espacio de usuario y el espacio de kernel**. Esta distinción, aunque al principio puede parecer una cuestión técnica menor, es en realidad la base de cómo funciona y se organiza todo el sistema operativo moderno.

En cada computadora donde corre un sistema operativo —ya sea Linux, Windows, macOS o cualquier otro— el procesador puede operar en **dos modos distintos**: uno **privilegiado**, que llamamos *modo kernel*, y otro **restringido**, conocido como *modo usuario*. Estos modos no son una idea abstracta: están implementados directamente en el hardware del procesador, mediante un simple bit que indica en qué modo se está ejecutando el sistema en ese momento.

Ahora, ¿por qué necesitamos esta separación? Porque el sistema operativo no es solo un programa más. Es **el programa que administra todo**: la memoria, el acceso al disco, la comunicación con los dispositivos, la planificación de procesos, y más. Si cualquier aplicación pudiera acceder directamente a esos recursos, el sistema sería inestable, inseguro y caótico. Por eso necesitamos que haya un árbitro, una entidad central que tenga el control y decida, con reglas claras, quién puede hacer qué. Ese árbitro es el **kernel**.

Cuando ustedes escriben código —en el TP, en sus proyectos, en un simple `printf`—, todo ese código se ejecuta en lo que llamamos **espacio de usuario**. Ese espacio está diseñado para ser seguro, controlado y limitado. En ese contexto, sus programas no pueden acceder directamente a la memoria del sistema, ni escribir en el disco rígido, ni hablarle directamente a la placa de red. No pueden, porque no confiamos en que el código de usuario esté libre de errores, y tampoco queremos que un error en un programa rompa todo el sistema.

En cambio, el **espacio kernel** es donde vive el código del sistema operativo. Allí no hay restricciones. El kernel puede hacer cualquier cosa: leer y escribir en cualquier parte de la memoria, hablar directamente con el hardware, manejar interrupciones, iniciar y detener procesos. Y, lo más importante: puede protegerse a sí mismo y proteger a los procesos de usuario entre sí.

Esta separación, que vamos a seguir explorando durante la cursada, permite que el sistema operativo brinde algo que llamamos **aislamiento y protección**. Cada proceso se ejecuta como si estuviera solo en la máquina, sin interferir con los demás, y sin poder dañar al sistema. Y aunque todo lo que hace un proceso pasa, eventualmente, por el kernel, ese pasaje se hace de forma controlada y segura, mediante mecanismos especiales que vamos a estudiar en profundidad: las *system calls*.

Entonces, si me tengo que quedar con una idea clave de esta sección, es esta:

> El sistema operativo funciona como un árbitro entre los programas y el hardware, y esa función es posible gracias a la separación entre el espacio de usuario y el espacio de kernel.

Sin esa división, simplemente no podríamos construir sistemas confiables, seguros y multitarea como los que usamos hoy. En la próxima sección, vamos a ver cómo eran los sistemas antes de que existiera esta separación... y por qué fue tan necesario inventarla.

2. Ejecución directa: el modelo de los sistemas antiguos

Para entender bien por qué es tan importante esta separación entre el espacio de usuario y el espacio de kernel, vale la pena mirar un poco hacia atrás y ver cómo eran los sistemas operativos en sus orígenes. Lo que hoy consideramos básico —como ejecutar varios programas a la vez o impedir que un programa dañe a otro— no siempre estuvo ahí. De hecho, durante mucho tiempo, **los sistemas operativos no hacían casi nada de eso**.

Voy a contarles una anécdota personal que les puede dar una idea. Cuando yo era chico y tomaba "clases de computación", usábamos unas PC que tenían instalado **DOS**, el *Disk Operating System*. Era una consola bastante rudimentaria. Uno escribía comandos como `PRINCE.EXE`, y arrancaba el *Prince of Persia*. O con suerte, nos dejaban jugar al *Doom*, que en esa época era lo más avanzado que había. Todo eso corría sobre DOS, un sistema operativo que no tenía ni protección, ni aislamiento, ni multitarea real.

¿Y cómo funcionaba? **Era un modelo de ejecución directa.** Eso significa que cuando uno ejecutaba un programa, el sistema operativo le entregaba el control total de la máquina. Así de simple. El programa podía acceder a toda la memoria, leer o escribir directamente en el disco, o manipular los registros del hardware como quisiera. No había intermediarios, ni permisos, ni barreras. El sistema operativo era apenas una colección de utilidades básicas y drivers. Era más un ayudante que un administrador.

Puede parecer una locura hoy en día, pero en ese momento tenía cierto sentido. Solo se ejecutaba **un programa a la vez**, así que no hacía falta preocuparse porque un proceso interfiriera con otro. Si estabas jugando al *Doom*, ese juego era el único dueño de la máquina. Y eso les permitía a los programadores hacer cosas impresionantes para la época. Por ejemplo, John Carmack —uno de los creadores de *Doom* y más tarde del Oculus— logró efectos 3D increíbles para ese momento, porque podía acceder (y "hackear") directamente a la memoria de video y hacer trucos que hoy serían imposibles sin romper todo el sistema.

Pero claro, ese nivel de acceso traía muchos problemas. Por ejemplo:

- **No había protección.** Un programa podía sobrescribir cualquier parte de la memoria, incluso la usada por el sistema o por otros programas.

- **No había aislamiento.** Si por alguna razón se ejecutaban dos programas al mismo tiempo —como en las primeras versiones de Windows—, cualquier cosa podía pasar. No había ningún mecanismo que impidiera que uno modificara o destruyera los datos del otro.

- **No había control de ejecución.** Si un programa quedaba corriendo eternamente, nadie lo podía desalojar para darle lugar a otro.

Todo esto implicaba que el sistema era **frágil y propenso a fallas**. Bastaba con que un programa mal escrito —o con mala intención— hiciera algo indebido para que se rompiera todo.

Por eso, a medida que las computadoras empezaron a tener más memoria, más usuarios y más necesidades de concurrencia, surgió la necesidad de diseñar sistemas operativos que pudieran **controlar mejor la ejecución de los programas**. Sistemas que pudieran decir: “Este programa puede hacer esto, pero no aquello; y si se pasa de la raya, lo corto”.

Ese fue el punto de inflexión que llevó al nacimiento del kernel moderno y a la distinción entre el espacio de usuario y el espacio de kernel. Pero para llegar ahí, primero hizo falta una idea revolucionaria que cambió para siempre la arquitectura de los sistemas operativos. Y esa idea nació con un sistema llamado **Multics**.

En la próxima sección vamos a hablar de eso.

3. El origen del aislamiento y la protección

Hasta acá vimos cómo los sistemas antiguos dejaban todo librado al azar: cualquier programa podía hacer cualquier cosa. Pero eso no escalaba. Cuando las computadoras empezaron a ser compartidas por varios usuarios, cuando necesitábamos ejecutar varios programas a la vez, cuando se volvió crítico que una falla no derrumbara todo… **hubo que inventar una mejor forma de organizar las cosas**.

Y ahí aparece una figura fundamental en la historia de los sistemas operativos: **Fernando Corbató**. Él fue el responsable de liderar el desarrollo de un sistema llamado **Multics**, que podríamos pensar como el abuelo de Unix. Multics fue un experimento ambicioso: quería ofrecer tiempo compartido real, con protección entre programas, y una estructura que permitiera que muchos usuarios pudieran usar la misma máquina sin pisarse entre ellos.

La gran idea de Multics —y que Unix heredó después— fue apoyarse en el **hardware** para implementar una parte crítica de la protección. ¿Por qué? Porque si dejábamos todo en manos del software, sería lento e ineficiente. Imaginen si el sistema operativo tuviera que revisar cada instrucción que ejecuta un programa para ver si tiene permiso: simplemente no sería viable.

En cambio, el diseño moderno se basa en un principio claro:

> El **hardware** provee el **mecanismo** de protección, y el **sistema operativo** aplica la **política**.

¿Qué significa eso? Que el procesador tiene circuitos internos (literalmente, transistores, lógica cableada) que pueden detectar si un programa en modo usuario intenta ejecutar una instrucción prohibida, o acceder a una zona de memoria que no le corresponde. En ese momento, el hardware **interrumpe la ejecución del programa** y salta al kernel para que tome una decisión.

El kernel, entonces, es el encargado de aplicar las reglas del juego. A veces decide que fue un intento indebido y termina el proceso (ese típico `Segmentation fault` que todos conocemos). Otras veces, como vamos a ver más adelante, puede usar esa interrupción a su favor, por ejemplo, para implementar estrategias de optimización como la carga diferida de páginas de memoria.

Pero lo más importante es que esta arquitectura —con el kernel como árbitro y el hardware como guardián— **permite construir un entorno seguro y estable**. Y eso da lugar a un segundo concepto fundamental que quiero que se lleven de esta sección: el **aislamiento**.

Cada programa que ustedes corren se ejecuta en su propio espacio, aislado de los demás. Aunque haya muchos procesos en paralelo, cada uno actúa como si estuviera solo en la máquina. No puede leer ni escribir en la memoria de otro, no puede cerrar sus archivos, no puede interferir con su ejecución. El kernel se encarga de que eso no pase, y el hardware lo respalda.

Este aislamiento tiene consecuencias muy prácticas. Por ejemplo, si su programa hace `malloc` y se olvida de hacer `free`, no pasa nada grave: cuando el proceso termina, el kernel recupera toda esa memoria. Lo mismo con los archivos abiertos: el kernel los cierra automáticamente al terminar el proceso. Todo esto es posible porque el kernel **mantiene estructuras internas** que registran lo que cada proceso hace, aunque ustedes no las vean directamente desde su código.

Y esto no se limita solo a la seguridad. También es una forma de diseño robusto: permite que cada parte del sistema funcione sin depender de la corrección o el buen comportamiento del resto. Es decir, aunque un programa falle, **el sistema puede seguir funcionando perfectamente**.

En resumen, el kernel moderno no es solo una colección de drivers, ni una simple biblioteca que facilita el acceso al hardware. Es una pieza crítica que:

- Administra los recursos del sistema,
- Protege a los programas entre sí,

- Y garantiza que todos puedan convivir sin destruirse mutuamente.

Y todo eso es posible gracias a una colaboración muy estrecha entre el kernel y el hardware. En la próxima sección vamos a bajar un poco más al nivel concreto y preguntarnos: ¿qué es realmente un proceso? ¿Qué estructuras lo componen? ¿Y cómo mantiene el kernel el control sobre todos ellos?

4. ¿Qué es y qué hace un proceso?

Ya hablamos bastante del kernel, de su rol como árbitro y de cómo se apoya en el hardware para proteger a los programas entre sí. Pero ahora nos toca hablar de los protagonistas concretos de esa historia: los **procesos**.

Un proceso no es solo un programa. Es, más bien, **una instancia en ejecución de un programa**, con todo lo que eso implica: su propio espacio de memoria, sus archivos abiertos, su estado, su posición en la planificación de la CPU, y un conjunto de estructuras internas que el kernel utiliza para monitorearlo, controlarlo y protegerlo.

Todo proceso vive en lo que llamamos **User Land**, el *espacio de usuario*. Ahí es donde corre el código que ustedes escriben. Cada línea que programaron, cada función que compilaron, se ejecuta ahí. Pero ese mundo, aunque parezca libre, está cuidadosamente delimitado por el sistema operativo.

Cuando miramos por dentro lo que define a un proceso, nos encontramos con varias piezas fundamentales:

- **El Process ID (PID):** un número único que identifica al proceso dentro del sistema. Es útil para que el kernel (y ustedes) puedan referirse a él.

- **Los file descriptors:** una lista que indica qué archivos, sockets o dispositivos tiene abiertos ese proceso. Cada uno tiene un número asociado (0 para stdin, 1 para stdout, 2 para stderr, etc.).

- **La estructura de memoria:** que incluye el código del programa, su pila (stack), el heap (donde vive la memoria dinámica), y otras áreas que el kernel mantiene separadas.

Lo interesante es que muchas de estas estructuras **no viven en el proceso mismo**, sino en el espacio kernel. El kernel mantiene internamente toda una colección de estructuras administrativas, invisibles para el código de usuario, pero fundamentales para que el proceso funcione como debe.

Por ejemplo, cuando hacen un `malloc`, el kernel se entera. Ustedes le están pidiendo más memoria. El kernel se la da, pero también **anota** que se la dio. Si después hacen `free`, genial:

la libera. Pero si no lo hacen y el proceso termina, **el kernel igual recupera esa memoria**, porque tiene registrado todo. Y lo mismo ocurre con los archivos: si abren uno y se olvidan de cerrarlo, el kernel lo cierra por ustedes cuando el proceso muere.

Esto es posible porque el kernel, como buen administrador, **no confía en el código de usuario**. Lo respeta, le da espacio para actuar, pero siempre lo vigila. El kernel no olvida. Y gracias a eso, aunque su programa esté mal escrito, no va a romper todo el sistema. Lo peor que puede pasar es que muera su proceso —un segmentation fault, por ejemplo—, pero el resto sigue funcionando perfectamente.

Acá es donde entra de nuevo en juego el concepto de **aislamiento**. Y es central.

Cada proceso se ejecuta **como si estuviera solo en la computadora**. Literalmente. Cree que tiene toda la memoria para sí, que es el único que accede al disco, que nadie más está ejecutando nada. Y eso es gracias a que el kernel —con ayuda del hardware— le crea esa ilusión: le da un espacio de memoria virtual exclusivo, le mantiene sus propios recursos, y **le impide interferir con otros procesos**.

Eso significa que, por más que escriban código que accidentalmente intenta acceder a la memoria de otro proceso, simplemente no va a funcionar. No hay forma. La memoria está protegida. Y si lo intentan, el hardware lanza una excepción, y el kernel interviene.

Este aislamiento no solo es bueno desde el punto de vista de la seguridad. También es una gran herramienta de diseño. Por ejemplo, si ustedes están haciendo un servidor, podrían implementarlo usando **threads** (hilos de ejecución dentro del mismo proceso), o usando **fork**, es decir, múltiples procesos. La diferencia clave es que los threads **comparten la memoria**, y eso puede ser más práctico para implementar algunos mecanismos de comunicación entre threads, pero también más riesgoso: un bug en un thread puede afectar a los demás. En cambio, si usan procesos separados con `fork`, **cada uno tiene su propio espacio de memoria**, completamente independiente. La separación o no separación de los espacios de memoria se vuelve una decisión de diseño con sus ventajas y desventajas.

Y eso es posible gracias al kernel, que hace ese trabajo de aislamiento. Cuando hacen un `fork`, el sistema operativo no solo crea una copia del proceso original: también copia su espacio de memoria (o lo simula, como veremos más adelante), y garantiza que **no se compartan datos salvo que ustedes lo permitan explícitamente**.

En definitiva, lo que quiero que se lleven de esta sección es esto:

> Un proceso es mucho más que un programa. Es una entidad viva en el sistema, administrada por el kernel, vigilada por el hardware, y diseñada para estar protegida, restringida y aislada.

Y entender cómo funciona un proceso —cómo se crea, cómo se destruye, qué recursos maneja— es esencial para comprender todo lo que viene: llamadas al sistema, planificación, sincronización, comunicación entre procesos, y mucho más.

5. Protección y soporte de hardware

Ya vimos que el kernel protege a los procesos, que los aísla, que evita que se pisen entre sí o que accedan al hardware directamente. Pero una pregunta natural que surge es:
¿cómo se logra realmente esa protección?

Y acá es donde entra un principio de diseño fundamental en sistemas operativos: la **separación entre mecanismo y política**.

¿Qué significa esto?

El **mecanismo** es el conjunto de herramientas básicas que hacen posible algo. En nuestro caso, esos mecanismos están implementados en el **hardware**: circuitos que saben, por ejemplo, si una instrucción está permitida en modo usuario, o si un programa está accediendo a una dirección de memoria que no le corresponde.

La **política**, en cambio, es el conjunto de decisiones sobre cómo y cuándo usar esos mecanismos. Eso lo decide el **sistema operativo**, es decir, el kernel.

Veámoslo en acción con un ejemplo. Imaginemos que su programa, corriendo en espacio de usuario, intenta escribir directamente en el disco rígido. Si eso estuviera permitido, sería un desastre: podrían sobrescribir archivos del sistema, destruir datos de otros programas o dañar el sistema entero.

Pero no se preocupen: **el hardware no lo permite**.

Hay un mecanismo físico dentro del procesador —un simple bit, un *flag*— que le dice al sistema si está en modo usuario o en modo kernel. Cuando está en modo usuario, muchas instrucciones directamente **no pueden ejecutarse**. Si el programa intenta hacerlo igual, el procesador **lanza una excepción**, es decir, interrumpe su ejecución y transfiere el control al kernel.

Entonces ahí, recién ahí, **el kernel decide qué hacer**. Tal vez termina el proceso con un `segmentation fault`. Tal vez, en otros casos más sofisticados, aprovecha esa interrupción para hacer una optimización. Pero siempre es el kernel el que toma la decisión. El hardware solo hace sonar la alarma.

> En otras palabras: el **hardware detecta**, el **kernel decide**.

Esto permite que los programas de usuario se ejecuten de manera eficiente, como si tuvieran el control del procesador, pero sin comprometer la seguridad del sistema. El procesador **no está**

chequeando cada instrucción con el sistema operativo: simplemente las ejecuta directamente... hasta que algo sea sospechoso. Y entonces, y solo entonces, interviene el kernel.

Modos de ejecución en el hardware

Para implementar todo esto, los procesadores modernos ofrecen **distintos modos de ejecución**. Lo más común es que tengan al menos dos:

- **Modo usuario (user mode):** restringido, sin acceso directo al hardware ni a instrucciones privilegiadas.
- **Modo kernel (kernel mode):** sin restricciones, con acceso total a los recursos del sistema.

Por ejemplo, el procesador **RISC-V**, que vamos a usar bastante en esta materia, tiene tres modos:

1. **Modo máquina (machine mode):** el más privilegiado, usado principalmente en sistemas embebidos.
2. **Modo supervisor:** donde vive el kernel de sistemas operativos modernos.
3. **Modo usuario:** donde se ejecutan las aplicaciones normales.

Cuando su programa corre en userland, el procesador está literalmente en **modo usuario**. Tiene el bit correspondiente activado. Si su código hace una llamada al sistema, como un `write()`, el sistema salta al **modo kernel**, y el procesador cambia de contexto: ahora ese bit está desactivado, lo que permite ejecutar instrucciones privilegiadas.

En procesadores más antiguos o complejos, como **x86**, hay hasta cuatro niveles de privilegio, conocidos como *rings*, aunque en la práctica se usan dos: el ring 0 (kernel) y el ring 3 (usuario). Todo lo demás queda reservado o no se utiliza en los sistemas operativos modernos.

Sin soporte de hardware, no hay sistema operativo moderno

Esto es clave: **sin estos mecanismos de hardware**, todo lo que queremos hacer desde el sistema operativo sería imposible. No podríamos implementar protección, aislamiento, ni multitarea confiable. El kernel estaría completamente a ciegas, sin forma de saber si un programa está rompiendo las reglas.

Por eso, al estudiar sistemas operativos, inevitablemente tenemos que hablar un poco de arquitectura de computadoras. Porque muchos de los principios que usamos —los modos de

ejecución, los registros, las interrupciones— están implementados **físicamente** en el procesador.

Entonces, si quieren una frase para resumir esta sección, es esta:

> El sistema operativo impone las reglas, pero **es el hardware el que las hace cumplir**.

Y juntos, kernel y hardware, logran que múltiples procesos, múltiples usuarios, y múltiples dispositivos convivan de forma estable, segura y eficiente.

En la próxima sección vamos a ver cómo se produce esa transición entre espacio de usuario y espacio de kernel, y cómo se estructuran las famosas **system calls**, esos saltos que conectan ambos mundos.

6. System Calls: puente entre el usuario y el kernel

Hasta ahora, vimos que el kernel tiene acceso total al sistema, y que los procesos de usuario están restringidos. Pero… si los procesos no pueden acceder directamente al disco, a la memoria compartida o a la red, ¿cómo hacen cualquier cosa útil?

La respuesta es a través de un **puente cuidadosamente controlado**: las *system calls*, o llamadas al sistema.

Una *system call* es, en términos simples, **una forma que tiene un programa de usuario de pedirle algo al kernel**. Pero no es una función cualquiera. No basta con hacer un `jump` al código del kernel, como si fuera una función normal. Eso sería peligroso y rompería todo el aislamiento que tanto nos costó construir. En cambio, el paso de usuario a kernel debe hacerse de forma **controlada, segura y verificada**.

La definición formal que más me gusta dice que una *system call* es:

> *Un punto de entrada controlado al kernel, que permite que un proceso solicite que el sistema operativo realice una operación en su nombre.*

Así de claro. No estamos ejecutando directamente una operación privilegiada: estamos **pidiendo que el kernel la ejecute por nosotros**.

¿Cómo ocurre esta transición?

Cuando escriben algo como `write(fd, buffer, size)` en C, están llamando a una función. Pero esa función no es una parte de su código, ni tampoco es parte del kernel directamente. Lo que realmente están invocando es una función dentro de una **librería del sistema**, como la famosa `glibc` (GNU C Library).

Esa librería hace varias cosas:

1. Prepara los argumentos de la llamada (por ejemplo, el descriptor de archivo, el puntero al buffer, etc.).

2. Invoca una instrucción especial que **genera una interrupción por software** (por ejemplo, `int 0x80` en x86 o `ecall` en RISC-V).

3. Esa interrupción **hace que el procesador cambie de modo**: de usuario a kernel.

4. El control del flujo pasa al kernel, que ahora puede ejecutar instrucciones privilegiadas.

5. El kernel realiza la operación (escribe en el disco, envía por red, etc.) y retorna el resultado.

6. Finalmente, el control vuelve al espacio de usuario, donde la ejecución continúa.

Todo eso, entre líneas de código, ocurre en microsegundos. Y muchas veces ni lo notamos. Pero es un salto importante: estamos saliendo del espacio de usuario y entrando en el núcleo mismo del sistema.

¿Por qué no podemos escribir nuestras propias *system calls*?

Porque el kernel **define y limita cuáles son las llamadas disponibles**. Ustedes no pueden inventar una *system call* nueva desde su programa. Solo pueden usar las que el kernel expone. Y eso es intencional: es una cuestión de seguridad.

Si el kernel aceptara cualquier código o cualquier tipo de llamada, los programas podrían hacer cualquier cosa. Pero en vez de eso, el kernel dice:

> "Estas son mis reglas. Estas son las operaciones que podés pedirme. Si necesitás algo que no está acá, escribí tu propio kernel."

Por eso, *system calls* como `read`, `write`, `open`, `close`, `fork`, `exec`, `wait`, `kill`, entre muchas otras, son las únicas formas oficiales de interactuar con el sistema operativo. Y por lo general, son suficientes para hacer casi cualquier cosa.

¿Y qué pasa con la performance?

No todo es perfecto. Cada vez que hacemos una *system call*, estamos generando una interrupción, cambiando de modo, accediendo a estructuras del kernel, volviendo al modo usuario… Todo eso tiene un **costo en performance**. No es enorme, pero tampoco es gratuito.

Por eso, por ejemplo, no conviene hacer un `write` por cada carácter de un archivo. Es mejor armar un buffer grande y escribir todo junto. Así evitamos cientos o miles de saltos innecesarios al kernel. Un buen diseño de programas en C, o en cualquier lenguaje que acceda al sistema, siempre tiene en cuenta este tipo de detalles.

En resumen, las *system calls* son el único canal legítimo que tienen los procesos de usuario para interactuar con el kernel. Son una especie de frontera vigilada, donde el usuario puede pedir cosas, pero no puede invadir el territorio del sistema.

En la siguiente sección, vamos a hablar un poco más sobre las **instrucciones privilegiadas**, esas instrucciones especiales que solo el kernel puede ejecutar, y cómo todo esto se relaciona con la performance y el diseño de sistemas.

7. Instrucciones privilegiadas y niveles de privilegio

Ahora que ya tenemos bastante claro qué es una *system call* y cómo funciona ese puente entre el espacio de usuario y el espacio de kernel, es momento de que bajemos un poco más al nivel del hardware para entender **por qué** todo este sistema es seguro. Y la respuesta está en algo que ya nombramos antes, pero que ahora vamos a explorar mejor: las **instrucciones privilegiadas** y los **niveles de privilegio**.

7.1 ¿Qué es una instrucción privilegiada?

La definición más directa es esta:

> Una **instrucción privilegiada** es una instrucción de la CPU que **no puede ejecutarse en modo usuario** porque puede comprometer la estabilidad o la seguridad del sistema.

Por ejemplo, una instrucción que escriba directamente en un puerto del hardware, o que modifique registros internos del procesador, es considerada privilegiada. Si un programa de usuario intentara ejecutar una de estas instrucciones, lo más probable es que termine con una excepción, y eventualmente, con su proceso muerto.

Este mecanismo es lo que impide que cualquier programa pueda hacer lo que quiera. Si el procesador detecta que algo indebido se está ejecutando en modo usuario, **lanza una excepción**, que no es otra cosa que una forma elegante de decir:

> “Esto no se puede hacer. Pasemos el control al kernel para que decida qué hacer con esta situación.”

7.2 La excepción como mecanismo de protección

Cuando el hardware detecta que se ejecutó una instrucción no permitida —como una instrucción privilegiada en modo usuario— se produce una **excepción** o **trap**. Esto fuerza un

salto al kernel. Lo interesante es que esto ocurre **automáticamente**, sin que el kernel tenga que estar mirando lo que hace cada programa.

Este comportamiento es clave. Permite que el programa de usuario se ejecute a toda velocidad, directamente sobre el procesador, sin intervención del sistema operativo... **hasta que intenta hacer algo que no debe**. En ese momento, el procesador interrumpe todo y le da el control al kernel.

Una vez en el kernel, se ejecuta un manejador de excepciones, que puede hacer muchas cosas:

- Terminar el proceso (por ejemplo, con un `Segmentation Fault` o un `General Protection Fault`)
- Registrar el error
- O incluso, en casos más avanzados, resolver el problema (como ocurre en optimizaciones como el *copy-on-write*)

7.3 Niveles de privilegio en hardware

Los procesadores modernos, como **x86** y **RISC-V**, implementan **niveles de privilegio** para reforzar esta idea de protección.

En x86, se usan los famosos anillos de protección:

- **Ring 0:** el más privilegiado. Donde corre el kernel.
- **Ring 3:** el menos privilegiado. Donde corre el usuario.
- **Ring 1 y 2:** existen, pero **no se usan** en la mayoría de los sistemas operativos modernos.

Visualmente se los suele representar como anillos concéntricos. El más interno (ring 0) puede hacer todo. Cada nivel más externo tiene menos permisos. Pero en la práctica, los sistemas modernos (como Linux y Windows) **usan solo ring 0 y ring 3**. El kernel vive en ring 0, y las aplicaciones de usuario, en ring 3.

En RISC-V, los modos se llaman de otra forma:

- **User Mode**

- **Supervisor Mode**
- **Machine Mode**

Cada uno representa un nivel distinto de control. En sistemas operativos modernos, nos movemos principalmente entre *User* y *Supervisor*. *Machine Mode* queda reservado para firmware o sistemas embebidos.

Estos niveles se representan internamente con bits en un registro especial del procesador. Por ejemplo, un **Current Privilege Level (CPL)** de 3 en x86 indica que estamos en ring 3, es decir, en modo usuario. Si tratamos de hacer algo reservado para niveles más bajos, el procesador compara ese CPL con el nivel requerido... y si no alcanza, lanza una excepción.

7.4 Acceso controlado al hardware mediante `IOPL`

Un ejemplo concreto de cómo se controla el acceso al hardware lo encontramos en la instrucción `OUT` de x86, que se usa para escribir directamente en puertos del hardware, como el del disco rígido.

Esta instrucción **no puede ser ejecutada por procesos de usuario**, y la razón está en un registro especial llamado `IOPL` (Input/Output Privilege Level). Este campo define qué nivel mínimo de privilegio se necesita para ejecutar instrucciones de entrada/salida.

El procesador compara dos cosas:

- El **CPL**, que indica en qué anillo está ejecutándose el proceso.
- El **IOPL**, que indica qué nivel se necesita para hacer una instrucción como `OUT`.

Si el CPL es **mayor** al IOPL (por ejemplo, CPL=3 e IOPL=0), entonces se lanza una **General Protection Fault**, y se salta al kernel.

Esto también nos permite entender mejor por qué **la única manera legítima de que un proceso de usuario acceda al hardware es haciendo una system call**. Esa llamada hace que el sistema entre en modo kernel (anillo 0), y desde ahí sí se pueden ejecutar instrucciones privilegiadas como `OUT`.

En resumen, esta sección nos deja varias ideas importantes:

- El procesador no solo ejecuta código: también **supervisa que el código cumpla con las reglas**.

- Los **niveles de privilegio** permiten separar claramente el código seguro (kernel) del potencialmente inseguro (usuario).

- Las **instrucciones privilegiadas** están prohibidas en modo usuario, y violarlas desencadena excepciones automáticas.

- Todo esto está **implementado en hardware**, no en software, y por eso funciona tan eficientemente.

8. Caso práctico: escribir en disco y la instrucción `OUT`

Hasta ahora hablamos bastante de conceptos: niveles de privilegio, instrucciones privilegiadas, protección, excepciones, system calls… Pero nada como un **caso práctico** para ver cómo todas estas piezas encajan. Vamos a seguir un ejemplo real: ¿qué pasa cuando un programa quiere leer (o escribir) en el disco?

Spoiler: el programa no accede directamente al disco. **Nunca**.

8.1 ¿Qué pasa cuando hacemos un `read` o un `write`?

Cuando ustedes llaman a `read()` desde su código en C —por ejemplo, para leer el contenido de un archivo— parece algo trivial. Pero por debajo, están ocurriendo muchas cosas. Lo que realmente está pasando es que están haciendo una **system call** que genera un **salto controlado al kernel**.

Ese salto, como vimos, cambia el modo de ejecución del procesador: pasamos de **modo usuario (anillo 3)** a **modo kernel (anillo 0)**. Una vez allí, el kernel es el que **decide cómo cumplir con ese `read`**: puede acceder al disco, al caché del sistema de archivos, a la memoria, o a una combinación de todo eso.

Esto es fundamental: **el programa de usuario nunca toca el disco directamente**. Ni sabe dónde está el disco, ni con qué protocolo comunicarse, ni qué registros hay que configurar. Todo eso lo maneja el kernel. ¿Y cómo lo hace? Usando instrucciones privilegiadas.

8.2 El rol de la instrucción `OUT` en x86

En la arquitectura x86, una de esas instrucciones privilegiadas es `OUT`. Su trabajo es escribir datos en un **puerto de E/S** (entrada/salida), que es una dirección especial que representa un dispositivo de hardware, como un disco rígido.

Veamos un ejemplo simple:

```
OUT puerto, valor
```

Esta instrucción le dice al procesador: “Escribí este valor en este puerto”. Por ejemplo, en el caso de un disco IDE, el kernel tiene que escribir una **secuencia de comandos** en una serie de puertos específicos para que el disco entienda qué tiene que hacer: leer, escribir, buscar, etc. No es un único `OUT`, sino una **coreografía de varias instrucciones** que siguen un protocolo rígido definido por el hardware.

¿Puede un programa de usuario ejecutar esa instrucción `OUT` directamente? **No.**

Y no es una cuestión de confianza: **el procesador no lo permite**. Tiene un sistema de validación basado en dos valores:

- **CPL (Current Privilege Level):** el anillo actual donde se ejecuta el código.
- **IOPL (I/O Privilege Level):** el nivel requerido para ejecutar instrucciones de E/S como `OUT`.

Si el `CPL > IOPL`, el procesador lanza una **excepción de protección general** (*General Protection Fault*), que transfiere el control al kernel. Esa es la forma en que el hardware impide que el usuario se meta con los dispositivos directamente.

8.3 El procesador como guardián

Volvamos al ejemplo. Supongamos que un proceso de usuario intenta ejecutar `OUT` desde el anillo 3. Lo que ocurre es lo siguiente:

1. El procesador compara el `CPL` (3) con el `IOPL` (0).
2. Como `3 > 0`, se lanza una **excepción**.
3. Esa excepción interrumpe el programa y **salta automáticamente al kernel**.
4. El kernel puede decidir qué hacer: terminar el proceso, registrar un error, o ignorarlo si corresponde (aunque en este caso no).

Entonces, ¿cómo logra un programa de usuario que se escriba algo en el disco? La respuesta ya la conocés:

- Hace una **system call** (`read`, `write`, etc.)
- El kernel toma el control

- El kernel ejecuta, entre otras cosas, **instrucciones privilegiadas como `OUT`**

- El kernel luego devuelve el control al programa

Este ejemplo resume perfectamente **todo el ecosistema de protección y colaboración entre hardware y sistema operativo**. El usuario no tiene acceso directo, pero puede pedirle al kernel que lo ayude. Y el kernel, que sí tiene permisos, se encarga de hacer las cosas como corresponde, bajo reglas estrictas.

Con esto completamos la primera etapa del recorrido: cómo se estructura el kernel, cómo se diferencia del espacio de usuario, y cómo interactúan ambos mediante System Calls.
A partir del próximo apunte, vamos a entrar en el soporte que brinda el hardware para que esta arquitectura sea posible y eficiente.