Mecanismos de protección de hardware

1. Modos de protección del hardware

Hasta ahora estuvimos viendo cómo el sistema operativo interactúa con los programas de usuario, cómo se estructura el kernel, y cómo se construye ese delicado equilibrio entre aislamiento, control y funcionalidad. Pero para que todo esto funcione —y, sobre todo, para que funcione bien— necesitamos **ayuda del hardware**.

En este apunte vamos a hablar precisamente de eso:
¿Qué hace el hardware para proteger al sistema operativo y a los procesos entre sí?
¿Cómo garantiza que un programa no pueda modificar arbitrariamente la memoria, detener el procesador, o acceder directamente al disco?

La respuesta está en una serie de mecanismos que provee el procesador y que el sistema operativo utiliza como base para implementar su política de protección. Sin ese soporte **no se podría construir un sistema operativo moderno**.

Vamos a repasar tres pilares fundamentales que hacen posible esta protección:

1. **Instrucciones privilegiadas**
2. **Protección de memoria** (a través del modelo de memoria virtual)
3. **Interrupciones de timer** para garantizar el control sobre el uso del CPU

Este apunte va a conectar lo que venimos viendo a nivel de software con las bases de arquitectura de computadoras. Así como el kernel impone reglas, el hardware es quien las hace cumplir.

2. Instrucciones privilegiadas

Uno de los mecanismos fundamentales de esa protección son las llamadas **instrucciones privilegiadas**.

2.1 ¿Por qué existen las instrucciones privilegiadas?

Pensemos en esto: si cualquier programa pudiera ejecutar instrucciones que interactúan directamente con el disco rígido, el procesador o la red, el sistema sería un caos. Bastaría una línea mal escrita —o maliciosa— para sobreescribir archivos importantes, colgar la máquina o comprometer datos sensibles.

Por eso, las arquitecturas modernas, como x86 y RISC-V, definen un conjunto de instrucciones que **solo pueden ejecutarse en modo kernel**. Son las que llamamos **instrucciones privilegiadas**.
Y un ejemplo típico, que ya vimos, es la instrucción `OUT` en x86, que permite escribir en puertos de hardware, como el disco.

Este conjunto de instrucciones está perfectamente documentado. Si alguna vez quieren ir al fondo de esto —por ejemplo, si están tan entusiasmados que quieren escribir su propio sistema operativo como TP profesional—, van a necesitar leer los manuales.

- En **RISC-V**, el **Volumen II** del manual define las instrucciones privilegiadas. Está pensado para quienes desarrollan sistemas operativos.

- En **x86**, Intel publica un manual gigantesco en varios volúmenes. El que nos interesa es el **Volumen III**, conocido como la *System Programming Guide*.
  En este contexto, "system" no significa cualquier sistema, sino *software de sistema*, como compiladores, bases de datos, y, por supuesto, sistemas operativos.

2.2 Ejemplos de instrucciones privilegiadas

Veamos algunas instrucciones reales que están restringidas al modo kernel, y por qué:

- `MOV` a registros de control como `CR0`–`CR4`: estos registros contienen configuraciones fundamentales del procesador. Por ejemplo, un bit en `CR0` activa o desactiva la **memoria virtual**. Si un proceso de usuario pudiera tocar eso, arruinaría todo el sistema.

- `HLT` (halt): detiene el procesador. Imagina que cualquier programa pudiera hacer esto...

- `LGDT` (Load Global Descriptor Table): configura estructuras que definen cómo se accede a la memoria. Otro punto clave para proteger.

Estas instrucciones, mal utilizadas, no solo afectan al proceso que las ejecuta: **afectan al sistema entero**.

2.4 Instrucciones no privilegiadas

En contraposición, el mundo en el que ustedes viven cuando programan (el **modo usuario**) está lleno de instrucciones seguras y no privilegiadas. Estas incluyen:

- Aritmética: `ADD`, `SUB`

- Movimiento de datos: `MOV`, siempre que no toque registros especiales

- Control de flujo: `JMP`, `CALL`, `RET`, mientras no salten a zonas prohibidas
- Stack: `PUSH`, `POP`
- `NOP`: No Operation, literalmente no hace nada, útil para alineación o tiempos de espera

Pero hay una que merece una mención especial:

- `INT`: esta instrucción genera una **interrupción por software**. Y sí, **puede ser ejecutada por el usuario**.

¿Y por qué es importante? Porque es el mecanismo base para implementar las **system calls**. Cuando ustedes hacen un `write()` o un `read()`, por dentro se ejecuta una `INT`, que le dice al procesador: “Necesito hablar con el kernel”. Y el procesador responde: “Muy bien, pasamos a modo kernel”.

Este detalle es tan importante que le dedicaremos un apunte especial. Es el mecanismo que permite que los programas pidan ayuda al kernel sin romper las reglas. Y como es una interrupción controlada, el kernel siempre sabe desde dónde vino la llamada, quién la pidió, y qué permisos tiene.

En resumen, las instrucciones privilegiadas son una pieza esencial del mecanismo de protección. Le dan al kernel el control exclusivo sobre las operaciones críticas del sistema, y se aseguran —con ayuda del hardware— de que ningún proceso de usuario pueda tomar atajos o forzar comportamientos indebidos.

3. Protección de memoria

Ya entendimos cómo el procesador protege al sistema restringiendo el acceso a instrucciones privilegiadas. Ahora vamos a ver otra parte fundamental del mecanismo de protección: **la memoria**. En nuestro curso tendremos toda una unidad dedicada a la memoria en general y a la protección en particular. Pero esta sección servirá de introducción.

No alcanza con que el código de un proceso no pueda detener el procesador o escribir en un puerto. **También es clave que no pueda leer ni escribir en la memoria de otro proceso**, ni siquiera por error.

Y ahí entra en juego la **memoria virtual**.

3.1 Memoria virtual como abstracción

Cuando un programa corre, no accede directamente a la memoria física de la máquina. En cambio, accede a lo que se llama su **espacio de direcciones virtual**. Esa memoria virtual es una **abstracción** que le hace creer al proceso que tiene su propio bloque continuo de memoria —como si estuviera solo en la máquina.

Este espacio virtual está dividido en **páginas**. Por convención, una página suele tener 4 KB de tamaño. Lo importante es que cada página virtual se puede **mapear** a una página física real (del mismo tamaño). El sistema operativo y el procesador trabajan juntos para mantener ese mapeo.

Entonces, un proceso podría tener asignadas 20 páginas virtuales, y cada una podría estar mapeada a una ubicación distinta en memoria física. O incluso, algunas podrían no estar mapeadas todavía.

3.2 Tablas de mapeo: páginas virtuales a físicas

¿Y cómo se sabe qué página virtual apunta a qué página física? A través de una estructura conocida como la **tabla de páginas**.

La tabla de páginas es como un directorio. Dice:

- Página virtual 0x0000 → Página física 0xA000
- Página virtual 0x0001 → Página física 0xF000
- Y así...

El procesador consulta esta tabla **cada vez que el programa accede a memoria**. Lo hace tan rápido y de forma tan optimizada que no lo notamos. Nuevamente, tenemos al hardware acelerando un proceso que por software sería demasiado lento.

La tabla está configurada por el sistema operativo, y puede cambiar dinámicamente. De hecho, cuando ustedes hacen un `malloc`, puede que el sistema operativo les asigne nuevas páginas físicas y actualice la tabla en consecuencia.

Existe una tabla de páginas para cada proceso y cada proceso sólo accede a su propia tabla de páginas (de forma implícita, a través del procesador cada vez que accede a una dirección de memoria). Con esto se logra que un proceso no pueda acceder a páginas físicas de otro proceso; ni siquiera sabe donde encontrarlas!

3.3 Flags de protección por página

Cada entrada de la tabla de páginas no solo indica la correspondencia entre páginas virtuales y físicas, sino que también contiene **banderas (flags) de protección**. Algunas de las más importantes son:

- **Read (lectura):** ¿se puede leer esta página?
- **Write (escritura):** ¿se puede escribir?
- **Execute (ejecución):** ¿se puede ejecutar código en esta página?
- **User:** ¿puede acceder el usuario, o solo el kernel?

Esto permite, por ejemplo, que:

- La sección de **código (`text`)** de un proceso tenga solo permisos de **ejecución y lectura**, pero no de escritura.
- La sección de **datos (`data`)** tenga permisos de **lectura y escritura**, pero no de ejecución.

  Si un proceso trata de ejecutar código en `data`, o escribir en `text`, se lanza una excepción. El sistema lo mata antes de que cause daño.

El flag `user` cumple otro rol importante. Las páginas donde está `user = 0` son **exclusivas del kernel**. El código de usuario **ni siquiera puede leerlas**. Así se protege, por ejemplo, la memoria donde vive el kernel o las estructuras internas que administra.

3.4 Acciones prohibidas que generan excepción

Vamos a reforzar este punto con ejemplos concretos. Cada vez que el proceso intenta algo que **no tiene permiso para hacer**, el procesador lanza una **excepción**, y el control salta al kernel.

Situaciones típicas:

- Acceder a una página que no tiene permisos de lectura → excepción
- Escribir en una página marcada como solo lectura → excepción
- Ejecutar código en una página sin permisos de ejecución → excepción
- Acceder a una página no mapeada (inexistente o inválida) → excepción
- Acceder a una página donde `user = 0` desde código de usuario → excepción

Este comportamiento es lo que permite el aislamiento **total entre procesos**, y entre procesos y kernel. Cada proceso vive en su burbuja virtual, y **no hay forma de romper esa burbuja salvo mediante una system call controlada**.

En resumen, la memoria virtual no solo es una comodidad para el programador (que cree que tiene memoria contigua), sino que es **una herramienta central para proteger y aislar procesos**. Y de nuevo, el sistema operativo decide las reglas, pero es el **hardware quien las hace cumplir**.

En la próxima sección vamos a cerrar esta clase hablando de **excepciones** en general y cómo el kernel se organiza para responder a ellas.

4. Excepciones y control del sistema operativo

A esta altura ya vimos varias veces que, cuando un proceso intenta hacer algo que no tiene permitido —ya sea ejecutar una instrucción privilegiada, acceder a memoria protegida o interactuar directamente con hardware— el sistema **no lo deja pasar**. Pero, ¿qué pasa exactamente cuando algo así ocurre?

Ahí entra en juego uno de los mecanismos más potentes y delicados del sistema: **las excepciones**.

4.1 Excepciones como interrupciones

Una **excepción** es una especie particular de **interrupción** generada por el **hardware** cuando algo no está bien. Podés pensarla como un **salto forzado** que interrumpe el flujo de ejecución normal de un programa para entregarle el control al sistema operativo.

Por ejemplo:

- Intentás ejecutar `OUT` desde modo usuario → excepción
- Querés escribir en una página de solo lectura → excepción
- Ejecutás código en una región sin permisos de ejecución → excepción

Y así con muchos otros casos. En todos ellos, lo que ocurre es lo siguiente:

1. El procesador detecta el problema (por hardware, sin intervención previa del sistema operativo).
2. Cambia a modo supervisor y salta automáticamente a una dirección **preconfigurada** que el kernel registró al iniciar el sistema.

3. En esa dirección está el **handler de excepción**, que es simplemente una función del kernel que se encarga de manejar ese error.

Esto significa que, **cada vez que ocurre una excepción, el kernel recupera el control del sistema**.

Y esto es fundamental. Así se garantiza que, si un programa se "porta mal" —voluntaria o accidentalmente—, **no puede dañar al resto del sistema**, ni siquiera a sí mismo más allá de su propio proceso.

4.2 El kernel configura qué hacer ante cada excepción

Cuando arranca el sistema operativo, una de las primeras cosas que hace es registrar lo que se conoce como la **tabla de excepciones** o **vector de interrupciones**. Esa tabla le indica al procesador:

- Qué hacer si hay una violación de segmento
- Qué hacer si hay una instrucción ilegal
- Qué hacer si hay un page fault
- Qué hacer si el timer genera una interrupción
- Y así sucesivamente...

Para cada tipo de excepción, hay una **función de manejo específica**, que puede:

- Terminar el proceso culpable
- Loguear el evento
- Tomar medidas especiales (como cargar una página desde disco si es una page fault y está permitido)

Esto significa que el **control del sistema está siempre en manos del kernel**. Incluso cuando las cosas se salen de curso, el kernel es el que decide cómo responder.

Entonces, cada excepción es una oportunidad para que el sistema operativo:

- Detecte errores o accesos indebidos
- Proteja la integridad del sistema

- Mantenga el aislamiento entre procesos
- Administre recursos de forma segura

Sin este mecanismo, el sistema no podría garantizar ni seguridad, ni estabilidad, ni rendimiento.

En la próxima —y última— sección de la clase, vamos a ver cómo este mismo principio de "salto automático al kernel" se usa para **gestionar el uso del procesador**: cuando un proceso quiere (o no quiere) soltar la CPU. Ahí es donde entra el famoso **timer interrupt**.

5. Timer interrupt: el kernel retoma el control

Hasta ahora estuvimos viendo cómo el kernel **recupera el control** del sistema cada vez que algo sale mal: una excepción, una violación de permisos, una instrucción no autorizada. Pero… ¿qué pasa si todo sale bien?

Es decir, ¿qué pasa si un proceso está perfectamente bien escrito, no comete errores, y simplemente **no quiere soltar la CPU**?

En esos casos, el kernel necesita **una forma de retomar el control de todos modos**, para evitar que un único proceso acapare el procesador indefinidamente.
Ahí es donde entra en escena uno de los grandes aliados del sistema operativo: el **timer interrupt**.

5.1 ¿Por qué hace falta interrumpir procesos?

Las computadoras modernas tienen múltiples procesadores, pero también tienen **muchos más procesos que CPUs disponibles**. En una laptop común puede haber cientos de procesos activos —algunos visibles, otros en segundo plano— y todos quieren su momento para ejecutarse.

El sistema operativo resuelve esto haciendo **time sharing**, es decir, **repartiendo el tiempo del procesador** entre todos los procesos. Para lograrlo:

- Ejecuta un proceso por un pequeño intervalo de tiempo.
- Luego lo interrumpe, guarda su estado.
- Y pone a ejecutar otro.

Este proceso de quitarle el CPU a un proceso se llama **desalojo** (*eviction*), y puede ocurrir de varias formas.

5.2 Formas de ceder el procesador

A veces, el mismo proceso **cede voluntariamente** el procesador. Por ejemplo:

- Llama a `yield()`, una *system call* que literalmente le dice al sistema: "podés darme un descanso, dejá ejecutar a otro".

- Realiza una **llamada bloqueante,** por ejemplo, llama a `read()` o `write()` y queda **bloqueado** esperando E/S (por ejemplo, leer del disco). Mientras espera, el kernel puede ejecutar otro proceso.

Pero… no siempre pasa eso. Algunos programas —sobre todo los que hacen cálculos intensivos— **no tienen ninguna razón para frenar**. El kernel **no puede confiar en que los procesos se comporten bien y cedan voluntariamente el procesador.** Entonces, necesita un plan B.

5.3 Timer interrupt como mecanismo de desalojo forzado

Ese plan B es el **timer interrupt**. Es una interrupción generada **periódicamente por hardware**, cada cierta cantidad de milisegundos.

El mecanismo es así:

1. Cuando se inicia el sistema operativo, el kernel configura el timer (que suele estar embebido en el procesador).

2. Le dice: “Generame una interrupción cada X milisegundos” (por ejemplo, cada 10 ms).

3. Cuando el timer genera la interrupción, **el procesador salta automáticamente al kernel**, igual que con cualquier excepción.

4. El kernel chequea:

    - ¿Quién estaba ejecutando?

    - ¿Cuánto tiempo lleva ejecutándose?

    - ¿Se le terminó su *time slice*?

5. Si corresponde, el kernel guarda el estado del proceso actual y **pone a otro a ejecutar**.

Así, incluso si el proceso no se quiere ir, el kernel **lo saca por la fuerza**. En todo este proceso participa el Scheduler, que recibirá también una unidad completa del programa de nuestro curso.

5.4 Configuración del timer

Este timer, que genera las interrupciones periódicas, **es un dispositivo de hardware real**. Antiguamente era un chip separado. Hoy en día suele estar **integrado dentro del procesador**, como muchas otras funciones (hasta la placa de video, en laptops).

Conceptualmente, sigue siendo un hardware aparte. Su único trabajo es **avisar al kernel cada cierto tiempo**. No decide nada, solo dice: "¡Hey kernel! Pasaron 10 milisegundos, por si querías hacer algo…"

Y el kernel, que siempre está atento a estas interrupciones, aprovecha ese aviso para aplicar su política de planificación, decidir quién ejecuta y cuándo.

Este mecanismo es fundamental para que los sistemas operativos modernos logren:

- Multitarea
- Reparto justo del CPU
- Respuesta rápida a eventos
- Estabilidad frente a procesos voraces de CPU

Más sobre esto en la unidad sobre scheduling.

6. Conclusión

Con esto cerramos el recorrido por **los mecanismos de protección del sistema operativo**, y cómo el **hardware colabora activamente** en hacerlos cumplir.

Vimos:

- Espacios de usuario y kernel
- Instrucciones privilegiadas
- Protecciones de memoria
- Excepciones automáticas
- E interrupciones programadas, como el timer interrupt

Todo esto permite que muchos programas, escritos por muchas personas (y algunos mal escritos), puedan convivir en una computadora sin destruirse entre sí. Y en el centro de todo, está el kernel. Un árbitro. Un protector. Un administrador. Pero, sobre todo, **un programa privilegiado que sabe cuándo intervenir y cómo hacerlo.**