System Calls

1. Introducción: el salto entre mundos

Una de las ideas más poderosas del diseño de los sistemas operativos modernos es la **separación entre el espacio de usuario y el espacio del kernel**. Este límite, que parece tan conceptual al principio, es en realidad un muro bien concreto y necesario. El kernel no es solo un conjunto de funciones sofisticadas. Es **el árbitro, el mediador y el guardián** de todo lo que ocurre en la computadora.

Cuando uno programa, todo lo que escribe—ya sea en C, Python, Java o lo que sea—corre en **modo usuario**. En ese modo, el programa está contenido, limitado. No puede hacer cualquier cosa. No puede acceder a cualquier parte de la memoria. No puede escribir directamente en el disco. No puede tocar el hardware. ¿Por qué? Porque si cada programa pudiera hacer eso, **viviríamos en un caos permanente**: un programa podría corromper datos de otro, o peor, hacer caer todo el sistema.

Para evitar ese caos, se define un **modo kernel**, un entorno privilegiado al que solo ciertos fragmentos de código pueden acceder. El sistema operativo vive allí. Y esa separación no es un detalle menor: **es la base de la protección, la estabilidad y la seguridad** de cualquier sistema multitarea.

Ahora bien, la pregunta natural que surge es: **¿Cómo hace un programa común y corriente para pedirle algo al kernel, si no puede hablarle directamente?** Porque eventualmente, un programa tiene que leer un archivo, enviar datos por red o mostrar algo en pantalla. Todas esas son operaciones que requieren intervención del sistema operativo.

Ahí es donde entra en juego **la transición controlada**. Es decir, el mecanismo mediante el cual un programa de usuario puede solicitar algo al kernel, sin romper las reglas, y de una forma segura y predecible.

Ahora, es importante entender lo siguiente: **no se puede simplemente "saltar" al kernel** con un `call` o un `jmp` como si fuera una función más. Eso no está permitido. No podés escribir en el código: `jmp al_kernel`. El hardware lo impide.

Entonces, ¿cómo se hace? Esa es precisamente la historia que vamos a desarrollar a lo largo de esta clase: **cómo funciona el mecanismo de transición entre el modo usuario y el modo kernel**, cómo se implementa, por qué se hace así, y qué papel juega cada componente del sistema (el hardware, el sistema operativo, las librerías y el programa del usuario) en este delicado y poderoso baile.

Vamos a ver que detrás de un simple `write()` en C hay una **coreografía compleja**: desde el wrapper que setea los registros, hasta la instrucción que genera una interrupción, el vector que

decide a dónde saltar, el kernel que recibe el pedido, lo procesa, y después, con mucho cuidado, **te devuelve el control como si nada hubiera pasado**.

En definitiva, vamos a explorar **el corazón del mecanismo de invocación al kernel**. Lo que permite que el sistema operativo nos ayude, sin comprometer al sistema.

2. ¿Cuándo se salta al kernel?

Para entender bien cómo funciona este mecanismo de transición entre usuario y kernel, primero hay que saber **cuándo ocurre**. Porque este salto no sucede solo cuando ustedes, como programadores, quieren. A veces sí, pero muchas veces el kernel **entra en escena por su cuenta**, sin que nadie lo invite. Y eso tiene consecuencias importantes.

Empecemos por el caso más común y predecible:

System Calls

Cuando ustedes, desde su programa, hacen algo como `read()`, `write()`, `open()`, `fork()`, etc., están haciendo una **System Call**. Esto es una **llamada explícita al sistema operativo**. Básicamente, le están diciendo al kernel: “Hola, necesito que vos hagas esto por mí”. Este es un salto **voluntario y controlado**. El programa sabe que va a entrar al kernel y, en muchos casos, puede prepararse para eso.

Pero no todo es tan previsible. Hay otras situaciones donde **el salto al kernel ocurre de forma involuntaria**, y eso cambia bastante las reglas del juego.

Interrupciones de hardware

Imaginen que están escribiendo código que hace cálculos con matrices y de repente alguien aprieta una tecla o llega un paquete por red. ¿Qué pasa entonces? El kernel tiene que intervenir. Pero su programa no sabe nada de eso. Ustedes siguen multiplicando matrices.
El kernel entonces y sin pedir permiso **interrumpe la ejecución y salta a modo kernel** para atender ese evento. A esto se le llama **interrupción de hardware**.

Ejemplos típicos:

- **Teclado**: el usuario presiona una tecla.
- **Placa de red**: llega un paquete que hay que procesar.
- **Timer**: se cumplió el tiempo asignado a un proceso.

Y hay más.

Excepciones del procesador

Estas son situaciones donde el mismo procesador detecta una situación anormal y lo eleva al kernel. Un clásico y sencillo de entender: la **división por cero**. Si su código intenta dividir por cero, el procesador **lanza una excepción** y transfiere el control al kernel. No hay forma de que lo eviten o lo controlen desde el código común. El kernel se encarga de decidir qué hacer.

Lo importante es que **ustedes no eligieron saltar al kernel**, pero **el sistema los llevó igual**.

¿Por qué es importante todo esto?

Porque no es lo mismo tener una llamada voluntaria a una función (como en un `call` normal), que ser interrumpido de golpe, **sin haber guardado el estado de tu programa**. En una función común, vos sabés que vas a saltar, te podés preparar, guardar tus registros, armar tus argumentos.

En cambio, cuando llega una interrupción de hardware o una excepción, **no hay preparación posible del lado del usuario**. El kernel los saca de circulación de golpe, guarda su estado por su cuenta, y les garantiza devolverlo más tarde.

Este comportamiento nos obliga, como diseñadores del sistema operativo, a **tomar muchas más precauciones**. La transición no es trivial. No podemos asumir que el programa estaba listo para ser interrumpido.
Por eso, más adelante, vamos a ver cómo el kernel **guarda y restaura el contexto** de ejecución. Pero antes, tenemos que entender mejor **cómo se produce ese salto técnico**, y ahí entra el concepto fundamental del **vector de interrupciones**.

3. Interrupciones y el vector de interrupciones

Ya dijimos que hay muchas formas de saltar desde el espacio de usuario al modo kernel. Pero… ¿cómo sabe el procesador *a dónde* saltar cuando ocurre una interrupción o una excepción? ¿Y cómo distingue si lo que ocurrió fue un `read()`, una división por cero o un tick del timer?

La respuesta está en una estructura fundamental: el **vector de interrupciones**.

¿Qué es el vector de interrupciones?

Es una **tabla de entradas que el procesador consulta** cuando ocurre una interrupción o excepción. Cada tipo de evento (una tecla apretada, una división por cero, una llamada a `write()`, etc.) tiene asignado un **número de interrupción**, y ese número funciona como un índice en esta tabla.
En cada entrada del vector, hay una **dirección de memoria** que apunta a una rutina en el kernel: el **handler** de esa interrupción.

En otras palabras, cuando el procesador recibe una interrupción:

1. Obtiene el número de la interrupción.
2. Consulta la tabla.
3. Salta al código que le dice esa entrada.

Todo esto ocurre **automáticamente, por hardware**, sin intervención del programa en ejecución.

El rol del controlador de interrupciones (PIC, APIC)

Antes de que el procesador se entere de que hay una interrupción, alguien tiene que organizar el procesamiento de todas las señales que el hardware esta enviando.
Ahí entra el **Programmable Interrupt Controller (PIC)**. Este componente, que hoy suele estar embebido dentro del procesador (o en su cercanía), se encarga de:

- Recibir los pedidos de interrupción de múltiples dispositivos (teclado, red, timer...).
- Priorizarlos si hay varios al mismo tiempo.
- Notificar al procesador cuál debe atenderse primero.

En arquitecturas modernas como x86, el PIC fue reemplazado por el **APIC** (Advanced PIC), pero la idea general es la misma.

¿Qué hay en el vector de interrupciones?

Veamos una organización típica en sistemas como Linux en arquitecturas x86:

- **Entradas 0–31**: Excepciones del procesador (ej: división por cero, page faults, etc.)
- **Entradas 32–127**: Interrupciones de hardware (timer, teclado, red…)
- **Entrada 128 (0x80 en hexadecimal)**: **Interrupción por software usada para System Calls**

Ese último es el que más nos interesa ahora.

Cuando llaman a una System Call desde C (por ejemplo `read()`), el wrapper de la libc termina ejecutando una instrucción como `int 0x80` (en arquitecturas más modernas se usa `syscall`, pero la idea es la misma). Ese `int 0x80` genera una interrupción por software con el número

128. El procesador consulta la entrada 128 del vector de interrupciones, encuentra la dirección del handler correspondiente, y salta al kernel.

Interrupciones vs Excepciones vs Traps

La nomenclatura acá puede ser confusa, así que vale la pena aclararlo:

- **Interrupciones**: eventos generados por **hardware** (timer, teclado, red).
- **Excepciones**: eventos generados por **software o el procesador**, típicamente errores (divide by zero, page fault).
- **Traps**: nombre más general, que a veces engloba ambos conceptos (interrupciones + excepciones).

En algunos libros o arquitecturas, todo esto se agrupa bajo el nombre de “traps”. En otros, se hace una distinción más precisa. Lo importante es entender el mecanismo: **algo provoca un evento**, el procesador **interrumpe la ejecución actual**, y el kernel **toma el control**.

Con este esquema, el sistema operativo puede atender eventos externos, manejar errores, y recibir pedidos explícitos de servicio provenientes de los programas. Todo con una sola estructura bien pensada y bien organizada.

Profundicemos sobre este último caso. Ahora que sabemos cómo se llega al kernel, veamos **qué ocurre cuando uno lo hace voluntariamente**, a través de una System Call.

4. System Calls: cómo entramos al kernel voluntariamente

Hasta ahora vimos varias formas en que el kernel toma el control del sistema: interrupciones de hardware, excepciones del procesador… Pero en el caso más habitual para nosotros como programadores, **queremos pedirle algo al kernel de manera explícita y ordenada**. Por ejemplo, leer un archivo, escribir en pantalla, abrir un socket. Para eso existen las **System Calls**.

¿Qué es una System Call?

Una System Call es una **puerta de entrada controlada al kernel**. Es un mecanismo formalizado, establecido por el sistema operativo, que permite que el código de usuario le solicite una operación al kernel sin violar las reglas de protección. Podemos pensarlo como una especie de **API de bajo nivel del sistema operativo**.

Y es importante repetirlo: **solo se puede entrar al kernel mediante los mecanismos que él mismo permite**. Ustedes no pueden inventar una nueva System Call desde tu código. Solo podés usar las que el kernel ya tiene implementadas.

¿Y cómo se invoca una System Call?

Acá viene lo interesante. Desde tu código en C, se hace algo como:

```
write(fd, buffer, size);
```

Y eso parece una función más. Pero **por debajo pasa algo mucho más complejo**. Lo que estás llamando no es directamente el código del kernel, sino un **wrapper**: una función intermedia que forma parte de la librería estándar de C (la `glibc`, en Linux).

Este wrapper se encarga de:

1. **Cargar los parámetros** de la llamada (por ejemplo: qué escribir, a dónde, cuánto) en ciertos **registros** del procesador.

2. **Indicar qué System Call estás invocando**, también mediante un número en un registro.

3. Ejecutar una instrucción especial que **genera una interrupción por software**, generalmente `int 0x80` o `syscall` (dependiendo de la arquitectura y el sistema).

¿Y qué pasa después?

Cuando se ejecuta la instrucción `int 0x80` (o `syscall`), el procesador:

- Detecta que hay una interrupción.

- Consulta el vector de interrupciones, entrada 128 (para arquitecturas x86).

- Salta al handler correspondiente dentro del kernel.

Ahí arranca el código del kernel que maneja las System Calls. El kernel mira el número de System Call que le pasaron en el registro `EAX` (en x86), y de acuerdo a ese número, llama a la rutina interna que implementa la operación. Por ejemplo:

- `1` → `sys_exit`

- `3` → `sys_read`

- `4` → `sys_write`

Además, usa otros registros (`EBX`, `ECX`, `EDX`, etc.) para acceder a los parámetros que tambien le pasaron.
Ejemplo: si llamás a `read(fd, buffer, count)`, el wrapper:

- pone el número `3` (System Call `read`) en `EAX`
- pone `fd` en `EBX`
- pone la dirección de `buffer` en `ECX`
- pone `count` en `EDX`

Y ahí sí, el kernel ejecuta el código de la system call en sí.

¿Qué ventajas tiene este mecanismo?

- Es **seguro**: el usuario no puede hacer nada que el kernel no haya autorizado.
- Es **estructurado**: cada llamada tiene un número y parámetros bien definidos.
- Es **eficiente**: se implementa usando instrucciones de bajo nivel optimizadas.

Y aunque parece complejo, **para ustedes como programadores es casi invisible**. Usán `write()` y punto. Pero debajo de eso, hay todo un viaje entre espacios de memoria, registros, interrupciones y código privilegiado, que iremos desarrollando con el correr de las clases.

5. Del otro lado: qué hace el kernel cuando recibe una System Call

Ya vimos cómo un programa de usuario entra al kernel, ya sea por una interrupción, una excepción o una System Call. Ahora vamos a ver **qué pasa dentro del kernel cuando eso ocurre**. Porque, como dijimos antes, no es simplemente “hacer algo” y volver. El kernel tiene que cuidar muchas cosas para que todo funcione sin romper nada.

Primero, salvar el mundo (o al menos el contexto)

Al momento en que el procesador salta al kernel por una System Call (o por cualquier otro motivo), **todos los registros del procesador siguen conteniendo datos del programa de usuario**. Y esos datos no se pueden perder, porque cuando volvamos, el programa tiene que seguir como si nada hubiese pasado.

Entonces, **el kernel lo primero que hace es guardar el contexto del proceso**. Es decir:

- Guarda el valor de todos los registros de propósito general (en Intel, `EAX`, `EBX`, `ECX`, etc.)
- Guarda también el valor del program counter (la dirección de la instrucción siguiente a ejecutar)
- Guarda el estado de las banderas del procesador (carry, overflow, etc.)

Todo esto va a algún lugar seguro en memoria, controlado por el kernel. Cuando el kernel termine su trabajo, va a restaurar todos esos valores exactamente como estaban.

Después, verificar que todo tenga sentido

Una vez salvado el estado del proceso, el kernel tiene que decidir **qué hacer**. Para eso, mira el número de la System Call que el programa de usuario dejó en el registro `EAX` (o el que corresponda en la arquitectura que se esté usando).

Ahí verifica:

- ¿Ese número corresponde a una System Call válida?
- ¿Los parámetros que llegaron (en los otros registros) tienen sentido?
- ¿El proceso tiene permiso para hacer lo que está pidiendo?

Esto es clave. El kernel **no puede confiar en lo que le dice el usuario**, incluso si parece bien armado. Cualquier error, ya sea accidental o malicioso, puede comprometer el sistema. Por eso **hay validaciones en cada paso**.

Luego, ejecutar la System Call

Si todo está en orden, el kernel llama internamente a la rutina que implementa la System Call solicitada. Estas rutinas tienen nombres del estilo `sys_read`, `sys_write`, `sys_fork`, etc., y están escritas para trabajar en el contexto privilegiado del kernel.

Ahí es donde ocurre la magia: se accede a archivos, a memoria, a sockets, a drivers. Por ejemplo, en el caso de `read()`, la rutina puede:

- Leer datos desde un archivo abierto,
- O desde una tubería (`pipe`),
- O desde un socket,
- O desde el teclado...

Y coloca esos datos en el buffer que le indicó el programa de usuario (siempre validando que ese buffer esté en una zona de memoria legal y accesible).

Si durante la ejecución de la System Call ocurre algún problema (por ejemplo, acceso a memoria no permitida, file descriptor inválido, etc.), el kernel:

- Corta la ejecución de la System Call,
- Devuelve un error,
- O, en los casos más graves, **mata el proceso** con un `segmentation fault` o similar.

Finalmente, volver al usuario sin dejar rastros

Cuando termina de ejecutar la System Call, el kernel tiene que devolver el control al programa. Pero **no puede hacerlo así nomás**. Tiene que dejar todo como estaba:

1. **Restaura los registros** que había guardado.
2. **Coloca el valor de retorno** de la System Call (por ejemplo, la cantidad de bytes leídos) en el registro de resultado (`EAX`, en x86).
3. **Salta de nuevo al modo usuario**, en la dirección exacta donde el programa había quedado.

Desde el punto de vista del programa, fue como llamar a una función. Pero por debajo hubo un viaje completo por el espacio del kernel y muchas de verificaciones, movimientos de datos y cambios de contexto. Todo eso ocurrió en microsegundos.

¿Y el wrapper? ¿Qué pasó con él?

El wrapper que mencionamos antes (por ejemplo, el `write()` de la `glibc`) es el que hizo todo el trabajo de preparar la llamada. Cuando el kernel termina y devuelve el control, el wrapper simplemente continúa su ejecución:

- Mira si la System Call devolvió un valor válido o un error.
- Devuelve ese valor al programa que hizo la llamada.

Con esto ya cerramos el ciclo completo de una System Call. Ahora sabemos:

- Cómo se inicia,
- Qué pasa en el kernel,
- Y cómo volvemos al espacio de usuario.

En la próxima sección, vamos a ver **cómo se prepara todo eso en términos de registros**, usando ejemplos reales con código en ensamblador. Ya no lo veremos solo como teoría: vamos a ver cómo lo haríamos “a mano”.

6. El protocolo de una System Call: registros y ejemplos reales

Ahora que ya entendemos el concepto de System Call y cómo ocurre la transición entre usuario y kernel, vamos a meternos en los **detalles concretos**: ¿Cómo se le pasa información al kernel? ¿Cómo se arma una System Call a mano?

Cada System Call tiene un número

El kernel necesita saber **cuál** es la operación que queremos que haga. Y para eso, cada System Call está identificada con un número entero único. Por ejemplo, en Linux para x86:

Operación	Número
`exit()`	1
`read()`	3
`write()`	4
`open()`	5

Esos números se colocan en un **registro específico del procesador**. En x86, el registro que se usa es `EAX`.

Entonces, para hacer una System Call, lo primero es:

```
mov eax, 4   ; Número de la System Call 'write'
```

¿Y los parámetros? Van en otros registros

La mayoría de las System Calls necesitan más datos: ¿sobre qué archivo? ¿Con qué buffer? ¿Cuántos bytes?
Los primeros tres parámetros suelen pasarse en los registros `EBX`, `ECX` y `EDX`. Así:

```
mov eax, 4     ; Número de la System Call 'write'
mov ebx, 1     ; File descriptor (1 = STDOUT)
mov ecx, msg   ; Dirección del buffer a escribir
mov edx, 13    ; Cantidad de bytes
int 0x80       ; Interrupción que salta al kernel
```

¿Qué hace el kernel con eso?

Cuando el kernel recibe esta llamada:

1. Mira el `EAX` y dice: “Ah, me están pidiendo un `write`”.
2. Mira los otros registros:
    - `EBX`: ¿a qué archivo?
    - `ECX`: ¿de dónde saco los datos?
    - `EDX`: ¿cuántos bytes tengo que escribir?
3. Ejecuta la rutina interna correspondiente (`sys_write`).
4. Devuelve un resultado (por ejemplo, la cantidad de bytes realmente escritos) también en `EAX`.

Cuando vuelve al programa de usuario, ese valor está listo para ser usado. Si hubo un error, el kernel pone un número negativo (como `-1`) y la librería de C lo traduce en un `errno` que podemos consultar.

Ejemplo: llamar a `read()` a mano

Tomemos otro ejemplo. Supongamos que queremos leer desde el teclado (file descriptor 0), y guardar los datos en un buffer. El código se ve así:

asm
CopiarEditar

```
mov eax, 3     ; Número de la System Call 'read'
mov ebx, 0     ; File descriptor (0 = STDIN)
mov ecx, buffer ; Dirección del buffer
mov edx, 100   ; Leer hasta 100 bytes
int 0x80       ; Llamar al kernel
```

Y después de eso, `EAX` va a contener la cantidad de bytes leídos, o un número negativo si hubo un error.

¿Por qué esto importa?

Probablemente nunca tengan que escribir estas System Calls “a mano” con Assembler. Pero entender este protocolo es **clave para comprender cómo se comunican el usuario y el kernel**.

Además, si alguna vez necesitan debuggear un programa o estudiar seguridad, van a ver que muchas herramientas (`strace`, `gdb`, etc.) trabajan a este nivel. Y en sistemas embebidos, o en kernels experimentales como el `xv6`, muchas veces **se programan directamente así**.

Con esto cerramos el recorrido completo de una System Call. Desde el llamado en C, pasando por los registros, la interrupción, la ejecución en el kernel, y el retorno al espacio de usuario.

En la próxima clase, vamos a ver esto implementado **en código real** dentro de `xv6`, el sistema operativo educativo que usamos como base. Ahí van a poder ver cómo están escritas las funciones `sys_write`, `sys_read`, y cómo se configura el vector de interrupciones. Es la mejor manera de terminar de entender cómo funciona todo esto por debajo.