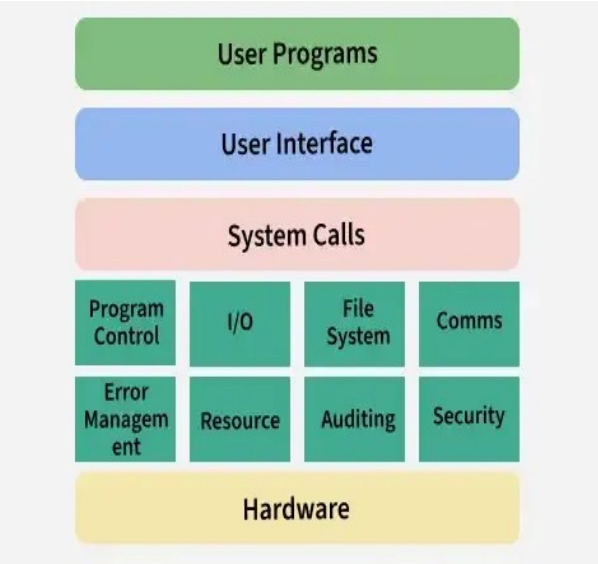
System Calls (Llamadas al Sistema)



¿Qué son exactamente las System Calls?

Las system calls o llamadas al sistema son la interfaz programática entre los procesos de usuario y el kernel del sistema operativo. Son el mecanismo mediante el cual una aplicación solicita servicios privilegiados al sistema operativo.

Definición clave: Las system calls son la única puerta de entrada legítima al kernel. Ningún programa de usuario puede acceder directamente al hardware sin pasar por ellas.

Si recordamos la arquitectura de capas:

NIVEL 4: APLICACIONES DE USUARIO
↓
[INTERFAZ: APIs del sistema]
↓
NIVEL 3: SOFTWARE DEL SISTEMA ← Shell, bibliotecas, utilidades
↓
[INTERFAZ: System Calls (syscalls)] ← ¡ESTA ES LA FRONTERA!
↓
NIVEL 2: KERNEL ← Aquí vive la magia
↓
[INTERFAZ: Instrucciones de hardware]
↓
NIVEL 1: HARDWARE

Las system calls son esa frontera protegida entre el espacio de usuario y el espacio del kernel.

¿Por qué existen? El problema de la seguridad

Recordemos los dos modos de operación:
- *Modo Usuario:* Restringido, sin acceso directo a hardware
- *Modo Supervisor (Kernel):* Privilegiado, control total

¿Qué pasaría sin system calls?

Escenario	**Consecuencia**
Un programa accede directamente al disco	Podría sobrescribir archivos del sistema
Un programa controla la CPU directamente	Podría monopolizarla infinitamente
Un programa accede a memoria de otros	Robo de datos, inestabilidad total

Las system calls son el "portero" que verifica que todas las solicitudes sean legítimas y seguras.

Tipos de System Calls (categorías)

Basándonos en las responsabilidades del kernel:

Categoría	**Ejemplos de syscalls**	**¿Qué hace el kernel?**
Control de procesos	`fork()`, `exec()`, `exit()`, `wait()`	Crear, terminar, sincronizar procesos
Gestión de archivos	`open()`, `read()`, `write()`, `close()`	Abrir, leer, escribir, cerrar archivos
Gestión de dispositivos	`ioctl()`, `read()`, `write()`	Comunicarse con hardware vía drivers
Gestión de información	`getpid()`, `gettimeofday()`, `setrlimit()`	Obtener datos del sistema o del proceso
Comunicación	`pipe()`, `socket()`, `shmget()`, `semop()`	IPC: pipes, sockets, memoria compartida
Proteccion		Controlar el acceso a los recursos. Obtener y fijar permisos Permitir y denegar acceso a usuarios

Setrlimit() : es una llamada al sistema en Unix/Linux que establece o modifica los límites máximos de recursos consumibles por un proceso (como memoria, tiempo de CPU, archivos abiertos). Define un límite "blando" (soft limit - el valor actual) y uno "duro" (hard limit - máximo permitido para el usuario no root) para un recurso específico.

exec(): Si fork() es una clonación, exec() es una mutación total. Esta llamada al sistema permite que un proceso deje de ejecutar su código actual y comience a ejecutar un programa completamente distinto.

1. ¿Cómo funciona la "Metamorfosis" del exec()?

Cuando un proceso invoca la familia de funciones exec() (como execl, execp, execve), ocurren los siguientes cambios radicales:

- Sustitución de Código: El kernel elimina el código binario actual de la memoria del proceso y carga las instrucciones del nuevo programa (por ejemplo, de un editor de texto a un listador de archivos).
- Reinicio de Memoria: Se limpian las variables globales, el montón (heap) y la pila (stack) del programa original para que el nuevo programa comience desde cero.
- Persistencia del Recipiente (PID): A pesar del cambio de "alma" (código), el "cuerpo" (el proceso en el sistema) mantiene el mismo Process ID (PID). Para el sistema operativo, es el mismo proceso, pero haciendo algo totalmente diferente.

2. El Ciclo de Vida: fork() + exec()

En sistemas tipo Unix/Linux, casi ningún programa inicia de la nada. La Shell utiliza una combinación de ambas syscalls para funcionar:

- fork(): La Shell se clona a sí misma, creando un proceso hijo idéntico.
- exec(): El hijo, inmediatamente, muta hacia el comando que el usuario escribió (como ls o gcc), reemplazando el código de la Shell por el del comando.
- wait(): Mientras tanto, el padre (la Shell original) espera a que esa metamorfosis termine y el proceso muera para volver a pedir una orden al usuario.

3. Estados posibles en la arquitectura de Linux/Unix:

- Número negativo (-1): Error (no se pudo crear el proceso).
- Cero (0): "Eres el hijo".
- Número positivo (>0): "Eres el padre, y este es el PID real de tu hijo".

4. Cuadro Comparativo

Característica	**fork()**	**exec()**
Procesos finales	Dos procesos (padre e hijo).	Un solo proceso (el mismo que llamó).
Código	El hijo tiene una copia del código del padre.	El código original se borra y se carga uno nuevo.
PID	El hijo recibe un PID nuevo.	El PID se mantiene igual.
Retorno	Retorna al código que lo llamó (si tiene éxito).	No retorna. Si tiene éxito, el código original ya no existe para recibir el retorno.

Ejercicio: El Enigma del Proceso Fantasma

syscall_fork.c

c

```
#include <stdio.h>
#include <unistd.h>

int main() {
   printf("1. Iniciando el programa principal (PID: %d)...\n", getpid());

   if (fork() == 0) {
      // --- BLOQUE DEL HIJO ---
      printf("2. Soy el hijo. Mi PID es %d. Voy a transformarme...\n", getpid());

      execl("/bin/ls", "ls", NULL); // Llamada al sistema EXEC [cite: 322, 323]

      printf("3. ¡Esta línea es un éxito! He terminado la transformación.\n");
   } else {
      // --- BLOQUE DEL PADRE ---
      sleep(2); // El padre espera un poco
      printf("4. Soy el padre. Mi hijo ya debe haber terminado su metamorfosis.\n");
   }

   printf("5. Fin del ciclo del programa.\n");
   return 0;
}
```

```
gcc syscall_fork.c -o syscall_fork
```

```
./syscall_fork
```

Preguntas :

1. ¿Se imprimirá la línea número 3? - *printf("3. ¡Esta línea es un éxito! He terminado la transformación.\n");*

   Respuesta esperada: No. Una vez que execl tiene éxito, el código original del hijo (incluyendo esa línea 3) es borrado de la memoria y reemplazado por el código de ls. Si el exec funciona, el programa original "muere" para dar paso al nuevo.

2. ¿Cuántas veces se imprimirá la línea número 5? - *printf("5. Fin del ciclo del programa. \n");*

Respuesta esperada: Solo una vez. Aunque hay dos procesos (padre e hijo), el hijo nunca llega a la línea 5 porque su "camino" fue desviado por el exec. Solo el padre, que no ejecutó el exec, llegará al final del código original.

3. Si el comando ls muestra archivos en la pantalla, ¿quién los está imprimiendo técnicamente?

   Respuesta esperada: El proceso hijo, pero usando el código nuevo de /bin/ls que cargó en su memoria.

Ejercicio: El Enigma del Proceso Fantasma

syscall_fork.c

c

```
#include <stdio.h>
#include <unistd.h>

int main() {
   printf("1. Iniciando el programa principal (PID: %d)...\n", getpid());

   if (fork() == 0) {
      // --- BLOQUE DEL HIJO ---
      printf("2. Soy el hijo. Voy a transformarme en 'ls -la'...\n");

      // execl(ruta, comando, argumento1, argumento2, ..., NULL)
      execl("/bin/ls", "ls", "-la", NULL);

      printf("3. ¡Esta línea es un éxito! He terminado la transformación.\n");
   } else {
      // --- BLOQUE DEL PADRE ---
      sleep(2); // El padre espera un poco
      printf("4. Soy el padre. Mi hijo ya debe haber terminado su metamorfosis.\n");
   }

   printf("5. Fin del ciclo del programa.\n");
   return 0;
}
```

```
gcc syscall_fork.c -o syscall_fork

./syscall_fork
```

Explicación de los parámetros:

Cuando usas execl, los argumentos tienen un orden estricto que a veces confunde:

- "/bin/ls": Es la ruta completa al ejecutable. El sistema necesita saber exactamente dónde está el archivo en el disco.
- "ls": Por convención, el primer argumento (arg0) es el nombre del programa. Aunque parezca repetitivo, es lo que el proceso verá como su propio nombre.
- "-la": Este es el argumento real que modifica el comportamiento de ls (formato largo y mostrar archivos ocultos).
- NULL: Es obligatorio. Le dice a la función execl que ya no hay más argumentos que leer.

Pregunta : Como seria para ejecutar *ls -l -a /home* ?

¿Cómo funciona una System Call paso a paso?

Ejemplo, un programa quiere leer un archivo. Secuencia completa:

PASO 1: Aplicación en User Space
└── El programa ejecuta: read (archivo, buffer, tamaño)
└── Biblioteca C (glibc) intercepta la llamada

PASO 2: Preparación de la syscall
└── La biblioteca coloca los parámetros en registros específicos
└── Número de syscall (ej: 0 para read en Linux x86-64)
└── Descriptor de archivo
└── Dirección del buffer
└── Cantidad de bytes

PASO 3: Interrupción/Trap
└── Se ejecuta la instrucción especial: SYSCALL (o int 0x80 en sistemas antiguos)
└── ¡CAMBIO DE MODO! Usuario → Kernel
└── El CPU salta a una dirección específica del kernel

PASO 4: Ejecución en Kernel Space
└── El kernel verifica los parámetros (¿el proceso tiene permiso?)
└── Localiza el archivo en el sistema de archivos (VFS)
└── Comunica con el driver del disco
└── Espera la lectura física (el proceso puede dormir aquí)
└── Copia los datos del kernel al buffer del usuario

PASO 5: Retorno
└── Se coloca el resultado en registros (bytes leídos o código de error)
└── CAMBIO DE MODO: Kernel → Usuario
└── La aplicación continúa su ejecución

Dato clave: El cambio de modo (context switch) tiene un costo. Por eso existen técnicas como vsyscall y vDSO para ciertas operaciones frecuentes (como `gettimeofday`).

vsyscall (Virtual System Call)

- Primera generación de esta optimización
- Mapea una página de memoria de solo lectura del kernel en el espacio de usuario
- Permite lecer datos del kernel sin cambiar a modo kernel
- Limitación: es estático y poco flexible

vDSO (Virtual Dynamic Shared Object)

- Evolución moderna que reemplaza vsyscall
- Es una biblioteca compartida dinámica proporcionada por el kernel
- Contiene código ejecutable en espacio de usuario para operaciones de solo lectura seguras
- Ventajas: más flexible, seguro y extensible

Tabla de Syscall esenciales (Linux x86-64)

ID (rax)	Nombre	Descripción
0	read	Leer datos de un archivo o terminal.
1	write	Escribir datos (ej: lo que usa printf por detrás).
2	open	Abrir un archivo para leer/escribir.
3	close	Cerrar un descriptor de archivo.
57	fork	Crear un proceso clon.
59	execve	Reemplazar el programa actual (la metamorfosis).
60	exit	Terminar el proceso actual.
61	wait4	Esperar a que un proceso hijo termine.

execve es la reina: Aunque en C usamos execl, execp o execv, a nivel de Kernel todas terminan llamando a la syscall número 59 (execve). Las demás son solo "envoltorios" (wrappers) de la biblioteca estándar para hacernos la vida más fácil.

El Registro RAX: Antes de la instrucción syscall, el programa "esconde" el número (ej. el 59) en un compartimento de la CPU llamado RAX. El Kernel entra, mira ese compartimento y dice: "Ah, quieres una metamorfosis".

bash

strace -e trace=process ls

Salida esperada

execve("/usr/bin/ls", ["ls"], 0x7ffdf5e14dc0 / 27 vars */) = 0*

- Veremos cómo aparece execve al principio del todo. Es la primera syscall que ocurre cuando lanzas cualquier comando, porque el sistema tiene que transformar el proceso vacío en el programa ls.
- 0x7ffdf5e14dc0 es la direccion de memoria, puntero hacia el Entorno del Proceso (Environment Variables). Ahí están guardadas todas las variables de entorno que el programa necesita para

funcionar, como el PATH (donde buscar comandos), el LANG (el idioma del sistema), o el USER.

- /* 27 vars */ en esa dirección de memoria hay un arreglo con 27 variables de entorno que el proceso padre (la Shell) le está heredando al proceso hijo (ls). Es como la "maleta" que el padre le entrega al hijo con las reglas del hogar antes de que empiece a correr.
- = 0 Si execve tiene éxito, ¡técnicamente nunca debería devolver nada al programa original! Porque el programa original ya no existe. El = 0 que ves en strace indica que el Kernel logró cargar el nuevo programa correctamente en el espacio de memoria y el proceso comenzó su nueva vida con éxito. Si vieras un -1, significaría que el exec falló (por ejemplo, si el archivo no tuviera permisos de ejecución o no existiera).

Pregunta: ¿Por qué ls sabe que mi terminal está en español o inglés?

La respuesta está en ese tercer parámetro 0x7f... que acabas de encontrar: ahí va la variable LANG.

Formas de entrar al kernel

Mecanismo	**¿Quién lo usa?**	**¿Cuándo?**
Trap (cambio de modo Usuario → Kernel) (SYSCALL)	Programas de usuario	Para pedir servicios al kernel (open, read, fork...)
Interrupt (IRQ)	Hardware	Cuando un dispositivo necesita atención
Exception	CPU automáticamente	Cuando ocurre un error (divison por 0, page fault (acceso a memoria inválida), instrucción ilegal)
SYSRET/IRET	Kernel	Para volver a modo usuario después de atender una syscall, interrupción o excepción

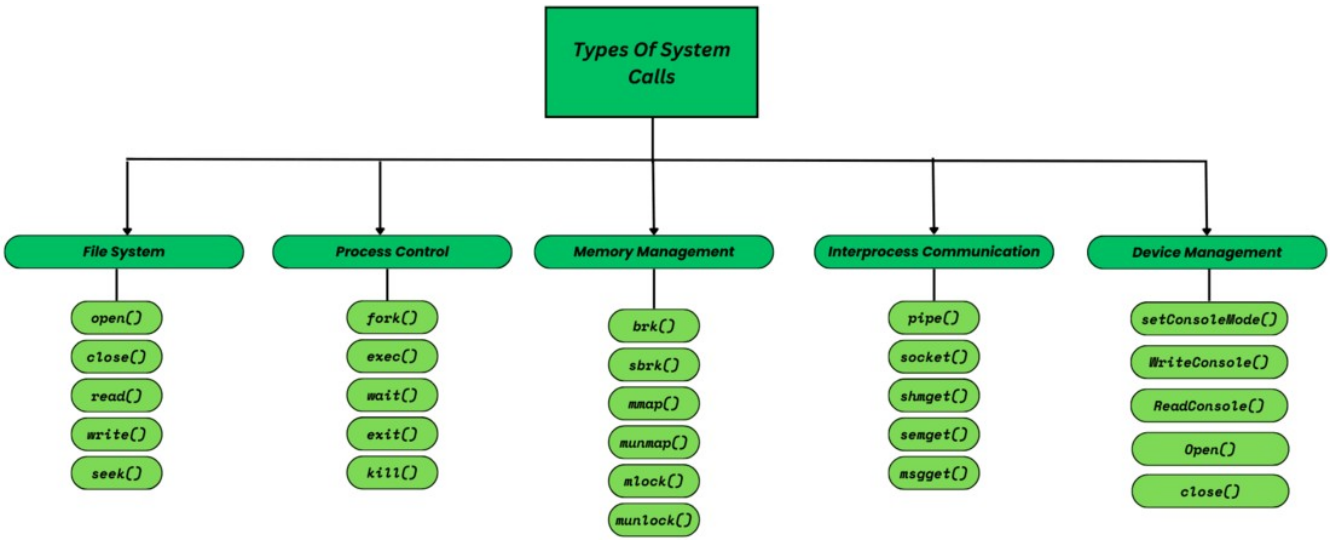
Costo del Trap en System Calls

Trap es una instrucción de CPU que genera una interrupción controlada para cambiar de modo Usuario a modo Kernel y ejecutar una system call.

Operación	**¿Usa trap?**	**Costo**
`pow()` (biblioteca matemática)	No	~ns (nanosegundos)
Variable local	No	~ns (nanosegundos)
open()	Sí	~µs (microsegundos, 1000x más)

Tipos de llamadas al sistema

Los servicios proporcionados por un sistema operativo suelen estar relacionados con cualquier tipo de operación que un programa de usuario pueda realizar, como creación, terminación, bifurcación, movimiento, comunicación, etc. Tipos similares de operaciones se agrupan en una única categoría de llamada al sistema. Las llamadas al sistema se clasifican en las siguientes categorías:



Operaciones del sistema de archivos

Estas llamadas al sistema se realizan mientras se trabaja con archivos en el sistema operativo, operaciones de manipulación de archivos como creación, eliminación, terminación, etc.

- **open():** Abre un archivo para leer o escribir (texto, audio, etc.).
- **read():** Lee datos de un archivo abierto.
- **write():** Escribe o guarda datos en un archivo.
- **close():** Cierra un archivo abierto.
- **lseek():** Mueve el puntero del archivo a una posición específica en un archivo (por ejemplo, salta a la línea 47 para leer desde allí).

Control de procesos

Este tipo de llamadas al sistema se ocupan de la creación de procesos, la terminación de procesos, la asignación de procesos, la desasignación, etc. Básicamente, gestiona todos los procesos que forman parte del sistema operativo.

- **fork():** Crea un nuevo proceso secundario duplicando el proceso principal.
- **exec():** Reemplaza el proceso actual con un nuevo programa (ejecución superpuesta usando el mismo PID).
- **wait():** Hace que el proceso principal espere hasta que finalice su proceso secundario.
- **exit():** Termina el proceso actual.

- **kill():** Envía una señal a un proceso (por ejemplo, finalizar, detener o reiniciar).

Gestión de memoria

Este tipo de llamadas al sistema se ocupan de la asignación y desasignación de memoria y del cambio dinámico del tamaño de la memoria asignada a un proceso. En resumen, la gestión general de la memoria se realiza mediante la realización de estas llamadas al sistema.

- **brk():** Establece la dirección final del heap / montón, cambiando directamente el tamaño de la memoria del montón.
- **sbrk():** Ajusta el tamaño del heap / montón mediante un valor positivo (aumento) o negativo (disminución).
- **mmap():** Asigna un archivo o dispositivo al espacio de memoria de un proceso para que se pueda acceder a él como a la memoria normal.
- **munmap():** Elimina un mapeo de memoria del espacio de direcciones de un proceso.
- **mlock() / munlock():** Bloquea páginas en la memoria para evitar intercambios; el desbloqueo las libera y las devuelve a la administración normal.
- **malloc(size) :** Asigna memoria sin inicializar

Recordar que Heap es Montón ó pila desordenada en español y es la región de memoria donde se asigna memoria dinámicamente durante la ejecución del programa

Característica	Stack (Pila)	Heap (Montón)
Orden	Ordenado, LIFO	Desordenado, libre
Gestión	Automática (compilador)	Manual (programador)
Asignación	`int x;` (declarar)	`malloc()` (solicitar)
Liberación	Automática al salir de función	Manual con `free()`
Velocidad	Muy rápida	Más lenta
Tamaño	Limitado (MBs)	Grande (GBs)
Fragmentación	No	Sí (huecos de memoria)
Uso típico	Variables locales, parámetros	Estructuras grandes, datos dinámicos

---

Stack (ordenado):

```
Entra función A → [A]          ← Llega primero
Entra función B → [B][A]       ← Se apila arriba
Sale función B  → [A]            ← Se desapila en orden inverso
Sale función A  → []             ← LIFO perfecto
```

Heap (desordenado):

```
malloc(100)  →  bloque en dirección 0x1000
malloc(50)   →  bloque en dirección 0x1064
free(100)    →  hueco de 100 bytes en 0x1000
malloc(30)   →  ocupa PARTE del hueco (0x1000)
malloc(80)   →  busca otro lugar, 0x1064+50...
                  ↑
          ¡Fragmentación! Huecos dispersos
```

memoria.c

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   // malloc: memory allocation de 100 bytes
   char *ptr = malloc(100);

   if (ptr == NULL) {
      printf("Error: malloc falló\n");
      return 1;
   }

   // Usar la memoria...
   sprintf(ptr, "Hola mundo");
   printf("Antes %s\n", ptr);

   // Liberar la memoria (free = liberar)
   free(ptr);


   printf("Despues %s\n", ptr);
   return 0;
}
```

Compilamos

```
gcc memoria.c -o memoria
```

Que ocurre?

Comunicación entre procesos (IPC)

Cuando se requieren dos o más procesos para comunicarse, entonces varios IPC El sistema operativo utiliza mecanismos que implican realizar numerosas llamadas al sistema. Algunos de ellos son:

- **pipe():** Crea un canal de comunicación unidireccional entre procesos (por ejemplo, padre → hijo).
- **socket():** Crea un socket de red para la comunicación entre procesos en el mismo sistema o en sistemas diferentes.
- **shmget():** Asigna un segmento de memoria compartida para que múltiples procesos puedan acceder al mismo espacio de memoria.
- **semget():** Crea o accede a un semáforo, utilizado para la sincronización de procesos al compartir recursos.
- **msgget():** Crea o accede a una cola de mensajes, lo que permite que los procesos intercambien mensajes de forma estructurada.

Gestión de dispositivos

Las llamadas al sistema de gestión de dispositivos se utilizan para interactuar con varios dispositivos periféricos conectados al PC o incluso con la gestión del dispositivo actual.

- **SetConsoleMode():** Establece el modo de entrada/salida de la consola (por ejemplo, controla el comportamiento de la línea de comandos en Windows).
- **WriteConsole():** Escribe datos en la pantalla de la consola.
- **ReadConsole():** Lee datos de la entrada de la consola.
- **open():** Abre un archivo o dispositivo y devuelve un descriptor de archivo para acceder.
- **close():** Cierra un archivo o dispositivo previamente abierto.

Algunos valores:

Constante	**Significado**	**Descripción**
O_RDONLY	Read Only	Solo lectura
O_WRONLY	Write Only	Solo escritura
`O_RDWR	Read Write	Lectura y escritura

https://www.geeksforgeeks.org/operating-systems/different-types-of-system-calls-in-os/

Valores del FD, File Descripción:

Número	**Nombre**	**Significado**
0	`STDIN_FILENO`	Entrada estándar (teclado)
1	`STDOUT_FILENO`	Salida estándar (pantalla)
2	`STDERR_FILENO`	Error estándar (pantalla, para errores)
3+	—	Archivos/dispositivos que **tú** abres

Ejemplo:

gestion_dispositivos.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <termios.h>

int main() {
    int fd;
    char buffer[100];
    ssize_t bytes_leidos, bytes_escritos;

    printf("=== DEMOSTRACIÓN: Gestión de Dispositivos ===\n\n");

    //
================================================================
===
    // 1. INTERACCIÓN CON DISPOSITIVO DE ENTRADA: TECLADO (/dev/tty)
    //
================================================================
===
    printf("1. Abriendo dispositivo de terminal (teclado/pantalla)...\n");

    // open() - SYSCALL: Abre el dispositivo de terminal actual
    // En Linux, todo es archivo. /dev/tty representa la terminal actual
    fd = open("/dev/tty", O_RDWR);
    if (fd == -1) {
        perror("Error al abrir /dev/tty");
        return 1;
    }
    printf("   ✓Dispositivo abierto. Descriptor de archivo: %d\n\n", fd);

    //
================================================================
===
    // 2. ESCRITURA EN DISPOSITIVO DE SALIDA: PANTALLA
    //
================================================================
===
    printf("2. Escribiendo en dispositivo de salida...\n");

    const char *mensaje = "   ✓Mensaje escrito directamente al dispositivo!\n";

    // write() - SYSCALL: Escribe bytes al dispositivo
```

```
    // Equivalente a WriteConsole() en Windows
    bytes_escritos = write(fd, mensaje, strlen(mensaje));
    if (bytes_escritos == -1) {
        perror("Error en write");
        close(fd);
        return 1;
    }
    printf("   Bytes escritos: %zd\n\n", bytes_escritos);

    //
================================================================
===
    // 3. LECTURA DESDE DISPOSITIVO DE ENTRADA: TECLADO
    //
================================================================
===
    printf("3. Leyendo desde dispositivo de entrada (teclado)...\n");
    printf("   Escribe algo y presiona ENTER: ");
    fflush(stdout);

    // read() - SYSCALL: Lee bytes desde el dispositivo
    // Equivalente a ReadConsole() en Windows
    bytes_leidos = read(fd, buffer, sizeof(buffer) - 1);
    if (bytes_leidos == -1) {
        perror("Error en read");
        close(fd);
        return 1;
    }

    // Eliminar el salto de línea
    buffer[bytes_leidos] = '\0';
    if (buffer[bytes_leidos - 1] == '\n') {
        buffer[bytes_leidos - 1] = '\0';
    }

    printf("   ✓Bytes leídos: %zd\n", bytes_leidos);
    printf("   ✓Texto ingresado: '%s'\n\n", buffer);

    //
================================================================
===
    // 4. INTERACCIÓN CON OTRO DISPOSITIVO: NULL (descarte)
    //
================================================================
===
    printf("4. Demostración con dispositivo /dev/null (agujero negro)...\n");

    int fd_null = open("/dev/null", O_WRONLY);
    if (fd_null != -1) {
```

```
        const char *datos = "Este mensaje desaparece...";
        write(fd_null, datos, strlen(datos));
        printf("   ✓Datos escritos a /dev/null (se descartan silenciosamente)\n");
        close(fd_null);  // SYSCALL: Cierra el dispositivo
    }
    printf("\n");

    //
================================================================
===
    // 5. CIERRE DEL DISPOSITIVO
    //
================================================================
===
    printf("5. Cerrando dispositivo de terminal...\n");

    // close() - SYSCALL: Cierra el descriptor de archivo
    // Libera recursos del kernel asociados al dispositivo
    if (close(fd) == -1) {
        perror("Error al cerrar");
        return 1;
    }
    printf("   ✓Dispositivo cerrado correctamente\n\n");

    //
================================================================
===
    // RESUMEN DE SYSCALLS USADAS
    //
================================================================
===
    printf("=== SYSTEM CALLS UTILIZADAS ===\n");

printf("┌──────────┬──────────────────────────────────────────────┐
──────────┐ \n");
    printf("│  Syscall  │           Función                │ \n");

printf("├──────────┼──────────────────────────────────────────────┤
──────────┤ \n");
    printf("│  open()   │ Abrir dispositivo/archivo            │ \n");
    printf("│  read()   │ Leer desde dispositivo de entrada    │ \n");
    printf("│  write()  │ Escribir a dispositivo de salida     │ \n");
    printf("│  close()  │ Cerrar dispositivo, liberar recursos │ \n");

printf("└──────────┴──────────────────────────────────────────────┘
──────────┘ \n");

    return 0;
}
```

Compilamos

gcc gestion_dispositivos.c -o gestion_dispositivos

Ejecutamos

./gestion_dispositivos

Ver las system calls en tiempo real (opcional, muy educativo)

strace ./gestion_dispositivos

Salida esperada:

=== DEMOSTRACIÓN: Gestión de Dispositivos ===

1. Abriendo dispositivo de terminal (teclado/pantalla)...
  ✓ Dispositivo abierto. Descriptor de archivo: 3

2. Escribiendo en dispositivo de salida...
  ✓ Mensaje escrito directamente al dispositivo!
  Bytes escritos: 49

3. Leyendo desde dispositivo de entrada (teclado)...
  Escribe algo y presiona ENTER: Hola SO
  ✓ Bytes leídos: 9
  ✓ Texto ingresado: 'Hola SO'

4. Demostración con dispositivo /dev/null (agujero negro)...
  ✓ Datos escritos a /dev/null (se descartan silenciosamente)

5. Cerrando dispositivo de terminal...
  ✓ Dispositivo cerrado correctamente

=== SYSTEM CALLS UTILIZADAS ===

Syscall	Función
open()	Abrir dispositivo/archivo
read()	Leer desde dispositivo de entrada
write()	Escribir a dispositivo de salida
close()	Cerrar dispositivo, liberar recursos

Cómo ver las system calls

En Linux (terminal):

bash

```
# Ver qué syscalls hace un comando
strace -c ls
```

Ver en tiempo real

strace ls

Ver todo

strace -f

Contar cuántas syscalls usa un programa

strace -c ./mi_programa

Salida típica:

% time	*seconds*	*usecs/call*	*calls*	*errors*	*syscall*
35.45	*0.000106*	*3*	*35*		*mmap*
20.40	*0.000061*	*2*	*30*		*openat*
15.05	*0.000045*	*1*	*25*		*close*

En Windows:

Usar Monitor de Recursos o Usar Process Monitor (ProcMon) de Sysinternals para ver operaciones de sistema equivalentes.

Código de demostración

demo_costo_trap.c

Compara el rendimiento de tres tipos de operaciones para demostrar que las system calls son mucho más lentas que las operaciones en user space, debido al costo del trap (cambio de modo Usuario → Kernel).

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <math.h>
#include <sys/time.h>

// Función para medir tiempo en microsegundos
long long tiempo_actual_us() {
 struct timeval tv;
 gettimeofday(&tv, NULL);
 return tv.tv_sec * 1000000LL + tv.tv_usec;
```

```
}

int main() {
    long long inicio, fin;
    long long tiempo_pow, tiempo_open, tiempo_variable;
    double resultado;
    int fd;
    int variable = 0;

    printf("=== DEMOSTRACIÓN: Costo del TRAP ===\n\n");
    printf("Medición de 100,000 operaciones de cada tipo:\n\n");

    // ========================================
    // 1. POW() - Biblioteca matemática (SIN trap)
    // ========================================
    inicio = tiempo_actual_us();

    for (int i = 0; i < 100000; i++) {
        resultado = pow(2.0, 10.0);  // 2^10 = 1024
    }

    fin = tiempo_actual_us();
    tiempo_pow = fin - inicio;

    printf("1. POW() (biblioteca matemática):\n");
    printf("   Resultado: %.0f\n", resultado);
    printf("   Tiempo total: %lld µs\n", tiempo_pow);
    printf("   Tiempo por operación: %.3f µs\n", tiempo_pow / 100000.0);
    printf("   → Sin trap: solo cálculo en CPU\n\n");

    // ========================================
    // 2. VARIABLE LOCAL (SIN trap)
    // ========================================
    inicio = tiempo_actual_us();

    for (int i = 0; i < 100000; i++) {
        variable = i * 2;  // Operación simple
    }

    fin = tiempo_actual_us();
    tiempo_variable = fin - inicio;

    printf("2. VARIABLE LOCAL (operación simple):\n");
    printf("   Resultado final: %d\n", variable);
    printf("   Tiempo total: %lld µs\n", tiempo_variable);
    printf("   Tiempo por operación: %.3f µs\n", tiempo_variable / 100000.0);
    printf("   → Sin trap: solo acceso a registro/CPU\n\n");

    // ========================================
```

```
// 3. OPEN() - System call (CON trap)
// ========================================
// Crear archivo de prueba primero
int fd_temp = open("test_temp.txt", O_CREAT | O_WRONLY, 0644);
close(fd_temp);

inicio = tiempo_actual_us();

for (int i = 0; i < 100000; i++) {
   fd = open("test_temp.txt", O_RDONLY);  // SYSCALL con TRAP
   close(fd);                             // Otra syscall
}

fin = tiempo_actual_us();
tiempo_open = fin - inicio;

printf("3. OPEN() + CLOSE() (system calls):\n");
printf("   Operaciones: 100,000 open + 100,000 close\n");
printf("   Tiempo total: %lld µs\n", tiempo_open);
printf("   Tiempo por open(): %.3f µs\n", tiempo_open / 100000.0);
printf("   → CON trap: cambio Usuario→Kernel→Usuario\n\n");

// ========================================
// COMPARACIÓN
// ========================================
printf("=== COMPARACIÓN DE RENDIMIENTO ===\n");
printf("Variable local:  %lld µs\n", tiempo_variable);
printf("pow():           %lld µs\n", tiempo_pow);
printf("open()+close():  %lld µs\n", tiempo_open);
printf("\n");
printf("open() es %.0fx más lento que pow()\n",
      (double)tiempo_open / tiempo_pow);
printf("open() es %.0fx más lento que variable local\n",
      (double)tiempo_open / tiempo_variable);
printf("\n");

// ========================================
// ¿POR QUÉ?
// ========================================
printf("=== ¿POR QUÉ ES TAN LENTO OPEN()? ===\n\n");

printf("POW() y variable local:\n");
printf("  [CPU] Ejecuta instrucciones ←── Todo en User Space\n");
printf("        ↓\n");
printf("  [CPU] Resultado listo\n\n");

printf("OPEN() (con TRAP):\n");
printf("  [User] Prepara parámetros\n");
printf("        ↓\n");
```

```
    printf("  [TRAP] SYSCALL instruction ──► Cambio a Kernel Mode\n");
    printf("         ↓\n");
    printf("  [Kernel] Guarda registros del usuario\n");
    printf("  [Kernel] Valida permisos del proceso\n");
    printf("  [Kernel] Busca archivo en VFS → ext4 → driver\n");
    printf("  [Kernel] Asigna file descriptor\n");
    printf("  [Kernel] Restaura registros\n");
    printf("         ↓\n");
    printf("  [TRAP] SYSRET ──► Vuelve a User Mode\n");
    printf("         ↓\n");
    printf("  [User] Recibe fd (o error)\n\n");

    printf("El trap añade: cambio de modo + validaciones +\n");
    printf("               saltos de contexto + trabajo del kernel\n\n");

    // Limpieza
    unlink("test_temp.txt");

    return 0;
}
```

Compilar con biblioteca matemática (-lm)

```
gcc demo_costo_trap.c -o demo_costo_trap -lm
```

Ejecutar

```
./demo_costo_trap
```

Salida esperada

```
=== DEMOSTRACIÓN: Costo del TRAP ===

Medición de 100,000 operaciones de cada tipo:

1. POW() (biblioteca matemática):
 Resultado: 1024
 Tiempo total: 2345 µs
 Tiempo por operación: 0.023 µs
 → Sin trap: solo cálculo en CPU

2. VARIABLE LOCAL (operación simple):
 Tiempo total: 89 µs
 Tiempo por operación: 0.001 µs
 → Sin trap: solo acceso a registro/CPU

3. OPEN() + CLOSE() (system calls):
 Operaciones: 100,000 open + 100,000 close
```

Tiempo total: 456789 µs
Tiempo por open(): 4.568 µs
→ CON trap: cambio Usuario→Kernel→Usuario

=== COMPARACIÓN DE RENDIMIENTO ===
Variable local: 89 µs
pow(): 2345 µs
open()+close(): 456789 µs

open() es 195x más lento que pow()
open() es 5132x más lento que variable local

=== ¿POR QUÉ ES TAN LENTO OPEN()? ===

POW() y variable local:
[CPU] Ejecuta instrucciones ←── Todo en User Space
↓
[CPU] Resultado listo

OPEN() (con TRAP):
[User] Prepara parámetros
↓
[TRAP] SYSCALL instruction ──► Cambio a Kernel Mode
↓
[Kernel] Guarda registros del usuario
[Kernel] Valida permisos del proceso
[Kernel] Busca archivo en VFS → ext4 → driver
[Kernel] Asigna file descriptor
[Kernel] Restaura registros
↓
[TRAP] SYSRET ──► Vuelve a User Mode
↓
[User] Recibe fd (o error)

El trap añade: cambio de modo + validaciones +
saltos de contexto + trabajo del kernel

Ver cada syscall individualmente

strace -c ./demo_costo_trap 2>&1

Salida típica

```
# % time     seconds  usecs/call     calls    errors syscall
# ------ ----------- ----------- --------- --------- ----------------
#  98.50    0.456789           4    100000           openat
#   1.20    0.005678           0    100000           close
#   0.10    0.000234           0      1000           write
# ...
```

¿Por qué pow() es más lento que la variable local si ambos están en user space?

Respuesta: pow() es una función compleja con muchas operaciones de punto flotante; la variable es solo una asignación de registro.

Si open() es tan lento, ¿por qué no hacemos todo en memoria?

Respuesta: La persistencia requiere disco; el kernel debe mediar por seguridad y coordinación.

Vemos que printf() hace buffering. ¿Cómo reduce eso los traps?

Respuesta: Acumula caracteres y hace un solo write() syscall en lugar de uno por carácter.

¿Cuántos traps ocurren en open() + close()?

Respuesta: Dos: uno para open() y otro para close().

VFS: Virtual File System

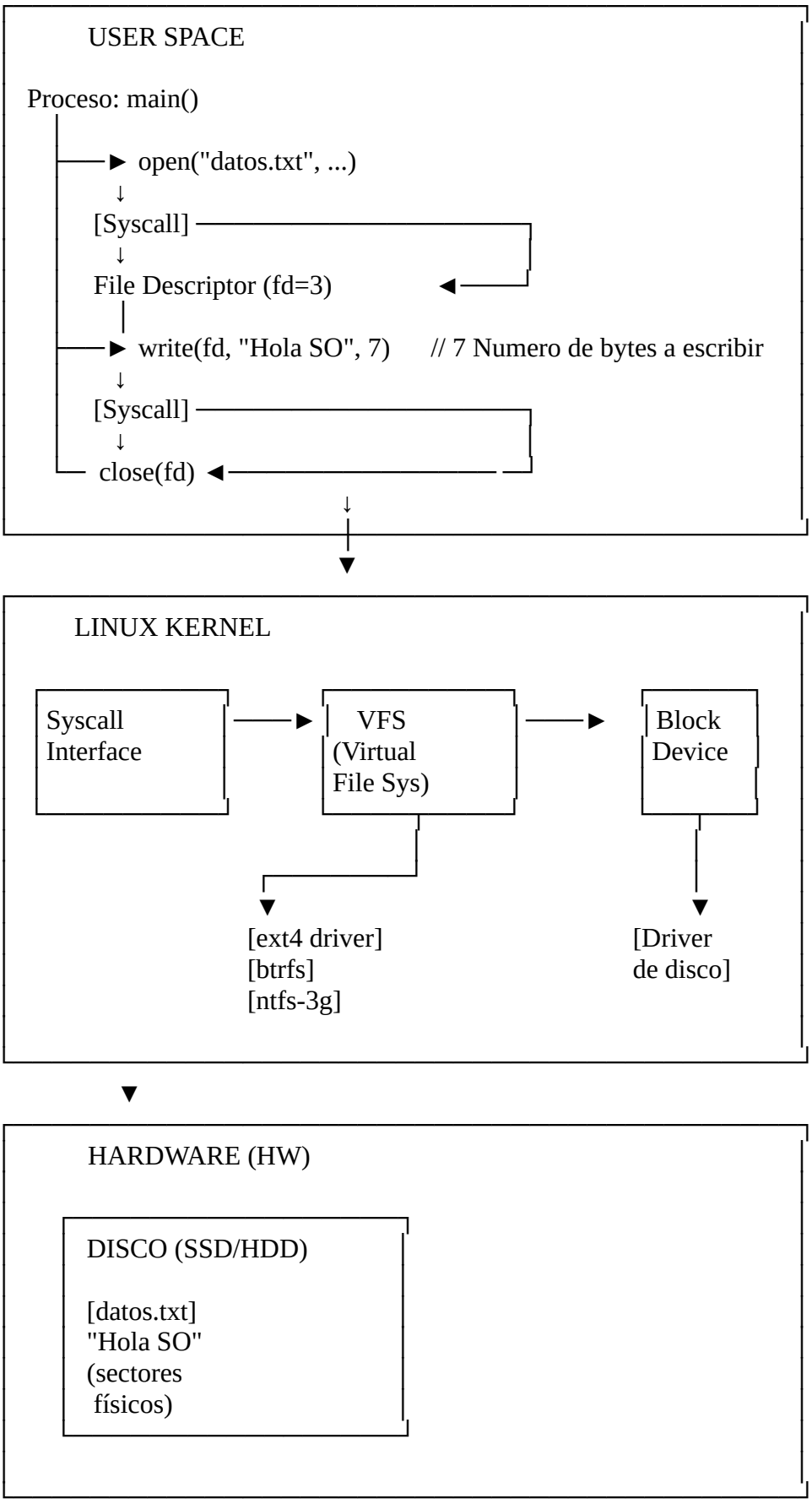
Capa de abstracción del kernel que permite que las mismas operaciones de archivo (open, read, write, close) funcionen sobre cualquier tipo de filesystem sin que la aplicación sepa cuál es.

¿Qué hace cada línea?

Línea de código	**Acción**	**Capa**
open(...)	Crea/abre archivo, obtiene permiso para escribir	User Space → Syscall
write(...)	Envía bytes al kernel	Syscall → VFS → Block Device
close(...)	Libera recurso, asegura escritura física	Syscall → Kernel limpia

Valores estándar reservados file descriptor

Número	**Nombre**	**Significado**
0	`STDIN_FILENO`	Entrada estándar (teclado)
1	`STDOUT_FILENO`	Salida estándar (pantalla)
2	`STDERR_FILENO`	Error estándar (pantalla, para errores)
3+	—	Archivos/dispositivos que **tú** abres


USER SPACE
Proceso: main()
open("datos.txt", ...)
[Syscall]
File Descriptor (fd=3)
write(fd, "Hola SO", 7) // 7 Numero de bytes a escribir
[Syscall]
close(fd)
LINUX KERNEL
Syscall Interface
VFS (Virtual File Sys)
Block Device
[ext4 driver]
[btrfs]
[ntfs-3g]
[Driver de disco]
HARDWARE (HW)
DISCO (SSD/HDD)
[datos.txt]
"Hola SO"
(sectores físicos)

System Calls en la práctica: Ejemplos visibles

Ejemplo 1: Creación de Procesos y Reemplazo de Programa

Demostración de cómo la shell ejecuta comandos, replicando lo que sucede cuando escribes ls -l en la terminal.

proceso.c

Replica lo que hace la shell cuando ejecutas un comando: crea un proceso hijo, lo transforma en otro programa (ls -l), y espera a que termine.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
 pid_t pid = fork();  // SYSCALL: clona el proceso actual

 if (pid == 0) {
 // Proceso hijo
 execl("/bin/ls", "ls", "-l", NULL);  // SYSCALL: reemplaza el programa
 } else {
 // Proceso padre
 wait(NULL);  // SYSCALL: espera a que termine el hijo
 }
 return 0;
}
```

Paso	Quien	Acción	Syscall	Resultado
1	Padre	Se divide en dos	fork()	Dos procesos idénticos
2	Hijo	Se transforma en `ls -l`	execl()	Ejecuta comando de listado
3	Padre	Espera pacientemente	wait()	Bloqueado hasta que hijo termine
4	Hijo	Termina al finalizar `ls`	exit() (implícito)	Desaparece
5	Padre	Despierta y continúa		Retorna al prompt

Compilamos con gcc:

```
gcc proceso.c -o proceso
```

Si hay errores, instalar gcc primero:

```
sudo apt update
sudo apt install gcc build-essential
```

Ahora lo ejecutamos:

./proceso

Ver syscalls en tiempo real

strace ./proceso

O solo contar cuáles usa

strace -c ./proceso

Strace full

strace -f ./programa

Salida ejemplo de strace -f ./programa

clone(...) = 12345 ← FORK (padre)
strace: Process 12345 attached ← Hijo detectado
[pid 12345] execve("/bin/ls", ...) = 0 ← EXEC (hijo)
[pid 12345] write(1, "total 128\n...", ...) ← ls escribe
[pid 12345] exit_group(0) = ? ← Hijo termina
[pid 1234] --- SIGCHLD ... ← Padre notificado
[pid 1234] wait4(-1, ...) = 12345 ← WAIT (padre)
[pid 1234] exit_group(0) = ? ← Padre termina

¿Qué system calls involucra la shell?

- fork() → Crear proceso ("Crear procesos: fork(), clone(), execve())
- execve() → Cargar programa
- wait() → Controlar ejecución
- clone() con flags SIGCHLD → equivalente a fork().

Algunos syscalls del ejemplo

Syscall	**usecs/call (mucro segundos por call)**	**¿Por qué varía?**
close	1 µs	Simple: libera descriptor
openat	2 µs	Busca archivo en caché
mmap	3 µs	Configura páginas de memoria
Read	5-50 µs	Depende: ¿está en caché o disco?
Write	5-100 µs	Puede requerir escritura física
fork	100-500 µs	Copia espacio de memoria

Si strace -c muestra:

usecs/call = 500
calls = 1000

¿Cuánto tiempo total se gastó en esa syscall?

Respuesta: 500 µs × 1000 = 500,000 µs = 0.5 segundos

El cuello de botella está en el kernel

Tu código puede ser perfecto, pero si hace syscalls ineficientes, será lento.

Capa	Velocidad	¿Dónde está el problema?
Tu algoritmo	Nanosegundos	Raramente aquí
Bibliotecas	Nanosegundos	Optimizadas
System calls	Micro-milisegundos	⚠ Aquí está el costo
Hardware	Variable	Fuera de control

Ejemplo: Un printf por carácter = 1000 syscalls vs. uno con buffering = 1 syscall.

Lo que revela strace que el profiler no ve

Herramienta	Mide	No detecta
Profiler de CPU	Tu código	Tiempo esperando al kernel
strace	Syscalls	Cuánto pides al sistema

Caso real:
- Profiler dice: "Tu función tarda 5 segundos"
- `strace` revela: Haces 50,000 `open()` innecesarios
- Solución: Abrir una vez, reutilizar descriptor

Problemas clásicos detectables

Patrón en strace	Problema	Solución
Miles de read() de 1 byte	Lectura sin buffer	fread(), aumentar buffer
open()/close() repetidos	No reutilizar archivos	Mantener descriptores abiertos
write() sin fsync	¿Datos seguros?	Decidir: velocidad vs. durabilidad
Muchos stat()	Búsquedas innecesarias	Cachear metadatos
fork() excesivo	Creación masiva de procesos	Usar threads o pool de procesos

Caso práctico: Antes y después

Código ineficiente (detectado con strace -c)
c

```
// 10,000 syscalls de write
for (int i = 0; i < 10000; i++) {
   write(fd, &datos[i], 1);  // 1 byte cada vez
}
// strace -c: 10,000 calls, usecs/call = 2, total = 20,000 µs
```

Código optimizado
c

```
// 1 syscall de write
write(fd, datos, 10000);  // Todo de una vez
// strace -c: 1 call, usecs/call = 5, total = 5 µs
```

Mejora: 4,000x más rápido.

strace te dice:

- ¿Cuántas syscalls haces?
- ¿Cuáles son las más costosas?
- ¿Dónde optimizar?

"¿Por qué printf() es más eficiente que llamar write() repetidamente para cada carácter?"

Respuesta: printf acumula en un buffer y hace una sola write() syscall, reduciendo el número de traps al kernel.

Ejemplo 2: Escribiendo en el Sistema de Archivos

Demostración del camino completo de una escritura a disco, desde la aplicación hasta el hardware, pasando por todas las capas del sistema operativo

escribe.c

Demuestra el camino completo de una escritura a disco, desde la aplicación hasta el hardware, pasando por todas las capas del kernel que gestionan archivos.

```
#include <fcntl.h>
#include <unistd.h>

int main() {
 int fd = open("datos.txt", O_WRONLY | O_CREAT, 0644);  // SYSCALL
 write(fd, "Hola SO", 7);  // SYSCALL → pasa a VFS → Block Device → Storage
 close(fd);  // SYSCALL
```

```
 return 0;
}
```

Proceso:

User Space write() → Syscall → File Descriptor → VFS → Block Device → Storage (HW)

Que hace cada linea?

Línea de código	Acción	Capa
open(...)	Crea/abre archivo, obtiene permiso para escribir	User Space → Syscall
write(...)	Envía bytes al kernel	Syscall → VFS → Block Device
close(...)	Libera recurso, asegura escritura física	Syscall → Kernel limpia

Bash

```
ls -la datos.txt
-rw-r--r-- 1 user user 7 Feb 21 10:30 datos.txt

cat datos.txt
Hola SO
```

Hacer un análisis de rendimiento del codigo.

Esto demuestra el viaje de los datos desde una aplicación hasta el disco físico, mostrando cómo el kernel media cada paso mediante system calls y el Virtual File System.

System Calls vs. APIs de biblioteca

Aspecto	System Call	API de Biblioteca
Ubicación	Kernel	Espacio de usuario
Costo	Alto (cambio de modo)	Bajo (solo código usuario)
Portabilidad	Depende del SO	Más portable (abstrae el SO)
Ejemplo	write()	printf()
Relación	printf() eventualmente llama a write()	printf() eventualmente llama a write()

Regla práctica: Las bibliotecas del sistema (Capa 3) agrupan y optimizan system calls. Por ejemplo, printf() hace buffering antes de llamar write() para reducir syscalls costosas.

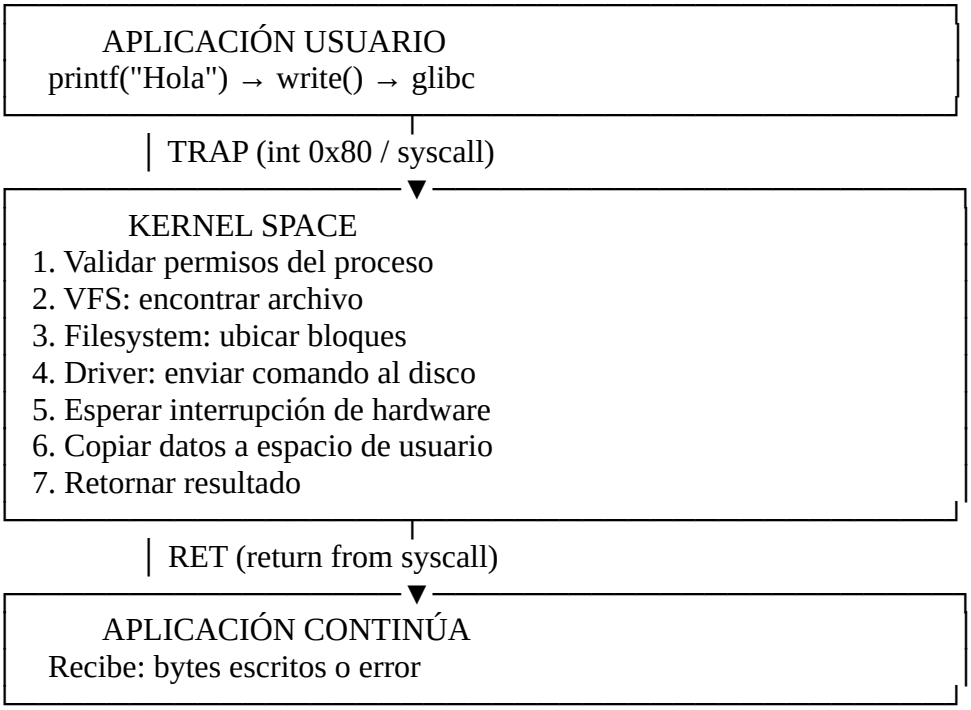
Conceptos

Concepto	Conexión con System Calls
Kernel	Es quien implementa y ejecuta las syscalls
Modo Usuario/Supervisor	La syscall es el puente entre ambos modos
Software de sistema	Proporciona wrappers convenientes sobre syscalls crudas
VFS	Capa del kernel que recibe syscalls de archivos
Protección de E/S	Se implementa verificando permisos en cada syscall
systemd vs init	Ambos usan syscalls para gestionar procesos, pero systemd usa más cgroups (syscalls modernas)

Preguntas

1. ¿Por qué `printf()` es más eficiente que llamar `write()` repetidamente para cada carácter?
   - Pista: Piensa en el costo del cambio de modo usuario→kernel.
2. Si las system calls son la única entrada al kernel, ¿cómo puede un virus dañar el sistema?
   - Pista: El virus es un programa como otro; si el usuario lo ejecuta, tiene sus mismos permisos.
3. ¿Por qué Android (pág. 3) necesita modificar las system calls de Linux tradicional?
   - Pista: Sandboxing de apps, cada app es un usuario Linux diferente.

Resumen visual: La syscall en contexto



Control de Procesos:

INTERRUPIR / (Interrupción)	**TERMINAR FORZADO (SIGKILL)**	**TERMINAR ORDENADO (SIGTERM)**	**PAUSAR (Stop)**
Evento externo (hardware o software)	El kernel mata el proceso inmediatamente	Se pide al proceso que termine él mismo	Se congela temporalmente
El proceso puede volver a ejecutar (ISR termina y vuelve)	NO se puede ignorar	SÍ se puede ignorar o manejar	SÍ se puede ignorar (raro)
Uso: Entrada de teclado, disco listo, timer	Uso: Proceso colgado, malware, emergencia	Uso: Cierre normal de aplicación	Uso: Depuración, control de jobs

ISR: Interrupt Service Routine / Rutina de Servicio de Interrupción: función del kernel que se ejecuta automáticamente cuando ocurre un evento de hardware o software que requiere atención inmediata. Con el ISR la CPU trabaja, el hardware avisa cuando está listo.