System Calls (Llamadas al Sistema)

Antes de ver los System Calls, entendamos primero los IPC

IPC: Inter-Process Communication - Comunicación entre Procesos en Sistemas Operativos

IPC (Inter-Process Communication) es el mecanismo que permite que dos o más procesos se comuniquen, sincronicen y compartan datos entre sí.

Concepto clave: Los procesos en sistemas operativos modernos están aislados en espacios de memoria separados (por seguridad). El IPC es la "puerta controlada" que permite que colaboren cuando es necesario.

¿Por qué es necesario?

Situación	**Ejemplo real**
Un proceso produce datos, otro los consume	Navegador descarga archivo → Explorador de archivos lo muestra
División de tareas complejas	Editor de video: un proceso renderiza, otro reproduce preview
Servicios del sistema	Aplicación solicita impresión → Servicio de impresión la gestiona
Múltiples instancias coordinadas	Navegador con varias pestañas compartiendo caché

Mecanismos de IPC

Responsabilidades del Kernel: Comunicación entre procesos (IPC)

Comunicación IPC: Pipes, señales, shared memory, sockets

Mecanismo	**Descripción**	**Uso típico**
Pipes (Tuberías)	Canal unidireccional de flujo de datos	`ls | grep txt` (comandos encadenados)
Señales (Signals)	Notificación asíncrona de eventos	Ctrl+C (SIGINT), kill (SIGTERM)
Memoria Compartida	Región de RAM accesible por múltiples procesos	Bases de datos, aplicaciones de alto rendimiento
Sockets	Comunicación entre procesos (misma máquina o red)	Servidores web, aplicaciones distribuidas

Colas de mensajes	Mensajes estructurados con prioridad	Sistemas embebidos, comunicación formal
Semáforos	Sincronización de acceso a recursos compartidos	Prevenir condiciones de carrera

Mecanismos explicados en detalle

1. Pipes (Tuberías)

Proceso A (escribe) ——————► [PIPE] ——————► Proceso B (lee)
ls -la datos grep ".txt"

Características:
- Unidireccional (uno escribe, otro lee)
- FIFO (First In, First Out)
- El kernel gestiona el buffer intermedio
- Anónimos (entre procesos padre-hijo) o con nombre (named pipes/FIFOs)

Ejemplo en shell:
bash

ps aux | grep firefox | wc -l
ps → pipe → grep → pipe → wc
Tres procesos comunicándose por pipes

wc significa "word count" (contador de palabras), pero es mucho más versátil. La opción -l especifica que queremos contar líneas (lines).

2 Señales (Signals)

Notificaciones asíncronas que interrumpen un proceso para informarle de un evento.

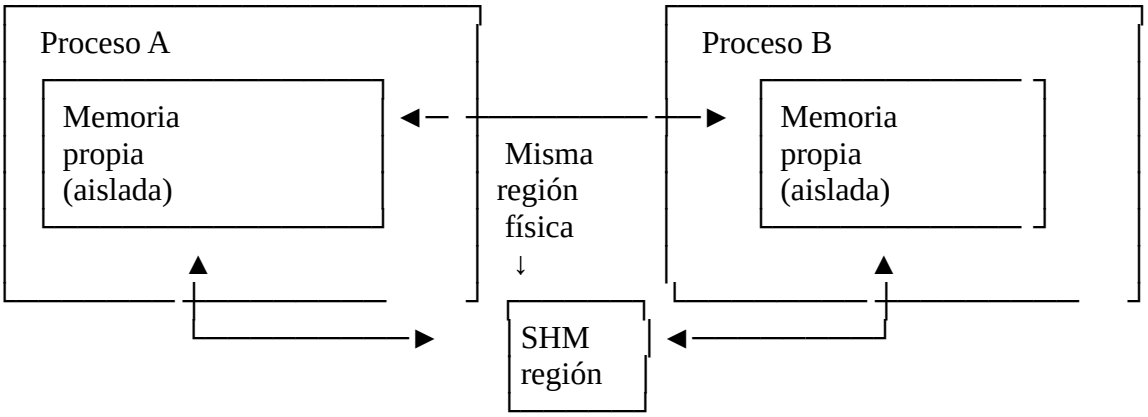
Señal	**Número**	**Significado**	**Origen típico**
SIGINT	2	Interrupción	Ctrl+C
SIGKILL	9	Terminación forzada	kill -9
SIGTERM	15	Terminación ordenada	kill
SIGSTOP	19	Pausar proceso	Ctrl+Z
SIGCONT	18	Continuar proceso	fg, bg

Enviar señales (SIGSTOP, SIGCONT) al kernel

Flujo:

Usuario presiona Ctrl+C
↓
Terminal envía SIGINT al proceso foreground
↓
Kernel interrumpe al proceso
↓
Proceso ejecuta manejador de señal (o termina por defecto)

3 Memoria Compartida (Shared Memory)



Kernel gestiona

Características:
- Más rápido que pipes/sockets (no copia datos)
- Requiere sincronización (semáforos/mutex) para evitar conflictos
- El kernel solo interviene al inicio (mapeo) y final (desmapeo)

4 Sockets

El mecanismo más versátil: permite IPC local y en red.

Tipo	**Familia**	**Uso**
Unix Domain	AF_UNIX	Misma máquina (más eficiente)
Internet	AF_INET	Diferentes máquinas (TCP/IP)

User Space: Process → sendmsg() → Syscall → Sockets → TCP/IP → Network Device → HW

Ejemplo real: Tu navegador (proceso) se comunica con el servidor web (otro proceso, posiblemente otra máquina) vía sockets.

Comparativa de mecanismos IPC

Característica	Pipe	Señal	Mem. Compartida	Socket
Velocidad	Media	Alta (solo aviso)	Muy alta	Media
Dirección	Unidireccional	Asíncrona	Bidireccional	Bidireccional
Complejidad	Simple	Simple	Compleja	Media
Datos	Flujo de bytes	Solo número (señal)	Estructura libre	Flujo o datagramas
Red	No	No	No	Sí
Sincronización implícita	Sí (bloqueante)	No	No	Opcional

Ejemplo práctico: Productor-Consumidor

Demostración de IPC (Inter-Process Communication) usando pipes, uno de los mecanismos que el kernel proporciona para que procesos se comuniquen.

PIPE - TUBERÍA DE AGUA

[Depósito A] ========► [Depósito B]
↑ ↑
Padre escribe Hijo lee
(llena la tubería) (vacia la tubería)

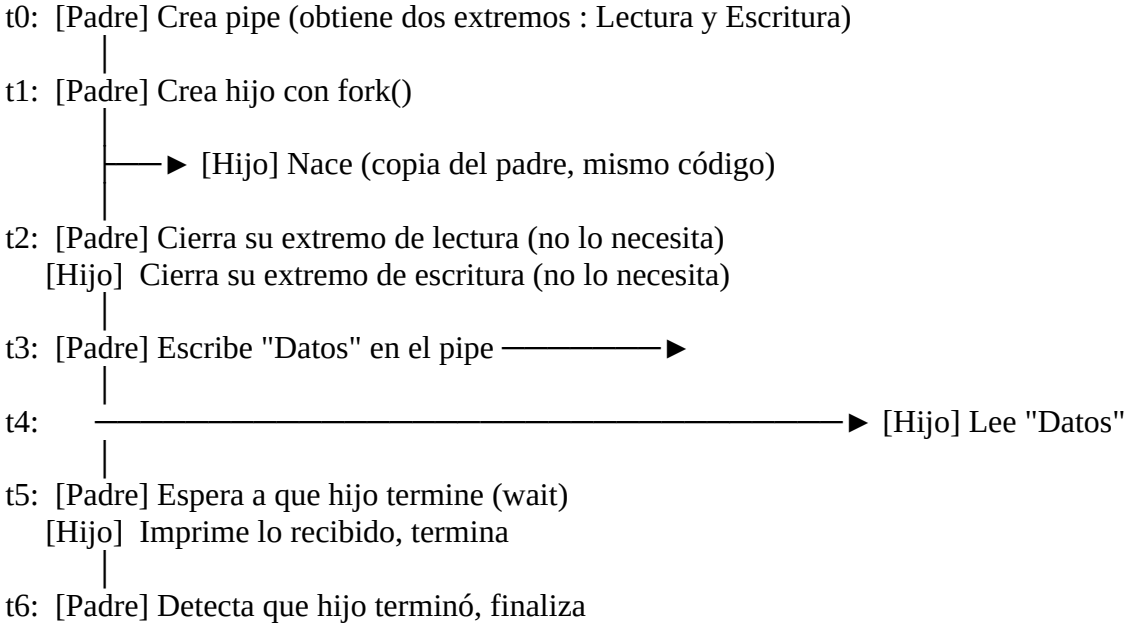
- Un solo sentido (unidireccional)
- Si está vacía, quien lee espera
- Si está llena, quien escribe espera

¿Quiénes participan?

Proceso	Rol	Acción
Padre	Productor	Crea el pipe, crea al hijo, envía datos
Hijo	Consumidor	Recibe datos del pipe, los procesa

Secuencia esperada

TIEMPO ──►



Clave: Son procesos independientes con espacios de memoria separados. No pueden compartir variables directamente.

Ver video

https://www.youtube.com/watch?v=Y2mDwW2pMv4

pipe_ejemplo.c

Un proceso padre envía datos a un proceso hijo a través de un pipe (tubería), que es un canal de comunicación unidireccional gestionado por el kernel.

```
#include <stdio.h>      // printf
#include <stdlib.h>     // exit
#include <unistd.h>     // pipe, fork, write, read, close
#include <sys/types.h>  // pid_t
#include <sys/wait.h>   // wait

int main() {
 int pipefd[2];
 pid_t pid;
 char buffer[10];

 // SYSCALL: Crear pipe
 if (pipe(pipefd) == -1) {
 perror("Error al crear pipe");
 exit(1);
 }

 // SYSCALL: Crear proceso (fork)
```

```
    pid = fork();

    if (pid < 0) {
    // Error
    perror("Error en fork");
        exit(1);
      }

      if (pid > 0) {
        // ========== PROCESO PADRE (Productor) ==========
        printf("[Padre] PID: %d, Hijo PID: %d\n", getpid(), pid);

        close(pipefd[0]);  // Cerrar extremo de lectura (no lo usa)

        // SYSCALL: Escribir en pipe
        write(pipefd[1], "Datos", 5);
        printf("[Padre] Enviado: 'Datos'\n");

        close(pipefd[1]);  // Cerrar extremo de escritura

        // SYSCALL: Esperar a que el hijo termine
        wait(NULL);
        printf("[Padre] Hijo terminado. Fin.\n");

      } else {
        // ========== PROCESO HIJO (Consumidor) ==========
        printf("[Hijo] PID: %d, Padre PID: %d\n", getpid(), getppid());

        close(pipefd[1]);  // Cerrar extremo de escritura (no lo usa)

        // SYSCALL: Leer de pipe (bloquea si está vacío)
        read(pipefd[0], buffer, 5);
        buffer[5] = '\0';  // Terminar string para printf

        printf("[Hijo] Recibido: '%s'\n", buffer);

        close(pipefd[0]);  // Cerrar extremo de lectura

        exit(0);  // SYSCALL: Terminar proceso hijo
      }

      return 0;
    }
```

Compilamos con gcc:

```
gcc pipe_ejemplo.c -o pipe_ejemplo
```

Si hay errores, instalar gcc primero:

sudo apt update
sudo apt install gcc build-essential

Ahora lo ejecutamos:

./pipe_ejemplo

Para ver las syscalls reales que hace la aplicación que acabamos de crear:

Instalar strace si no lo tienes

sudo apt install strace

Ver syscalls en tiempo real

strace ./pipe_ejemplo

O solo contar cuáles usa

strace -c ./pipe_ejemplo

¿Qué system calls se usan aquí?

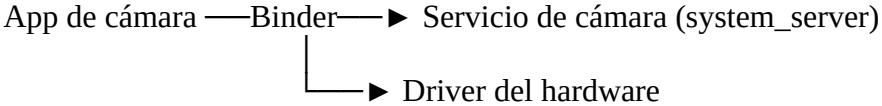
- pipe() → Crear canal IPC
- fork() → Crear proceso
- write() / read() → Transferir datos
- close() → Liberar recursos

IPC en sistemas modernos

Android: Binder IPC

Android reemplazó los mecanismos tradicionales de Linux por Binder:

- Más eficiente para móviles (batería, memoria limitada)
- Seguridad basada en permisos (cada app es usuario Linux diferente)
- Usado para toda comunicación con servicios del sistema



Windows: Mecanismos propietarios

- COM/DCOM: Objetos distribuidos
- Named Pipes: Similar a Unix

- Mailslots: Comunicación unidireccional broadcast
- Memory-mapped files: Equivalente a shared memory

Problemas clásicos en IPC

Problema	Descripción	Solución
Condición de carrera	Dos procesos modifican datos simultáneamente	Semáforos, mutex
Deadlock	Procesos se bloquean mutuamente esperando	Orden de adquisición de recursos
Starvation	Un proceso nunca accede al recurso	Prioridades, envejecimiento
Buffer overflow	Productor más rápido que consumidor	Colas circulares, control de flujo

Resumen

Mecanismos IPC

Simples	Avanzados
Pipes	Memoria compartida
Señales	Sockets
Colas mensajes	RPC/RMI
	Binder (Android)

Gestionados por el Kernel

- Crear canales
- Sincronizar acceso
- Verificar permisos
- Copiar datos entre espacios de memoria

Preguntas de comprensión

1. ¿Por qué el PDF menciona IPC como responsabilidad del kernel y no del software de sistema?
   - Pista: ¿Quién controla la memoria y los recursos compartidos?
2. Si los pipes son unidireccionales, ¿cómo logra una conversación bidireccional entre procesos?
   - Respuesta: Usando DOS pipes (A→B y B→A).
3. ¿Qué mecanismo de IPC usarías para que un servidor web (nginx) se comunique con una base de datos (PostgreSQL) en la misma máquina?
   Opciones: Unix domain sockets (más rápido) o TCP sockets (más flexible).

4. ¿por qué las operaciones IPC requieren "buffering en kernel space"?
   Respuesta: Porque los procesos no comparten memoria directamente; el kernel media.

¿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

¿Cómo funciona una System Call paso a paso?

Ejenplo, 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`).

Formas de entrar al kernel

Mecanismo	**¿Quién lo usa?**	**¿Cuándo?**
Trap (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 (div/0, page fault)
SYSRET/IRET	Kernel	Para volver a modo usuario

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)

Cómo ver las system calls en tu sistema

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()?

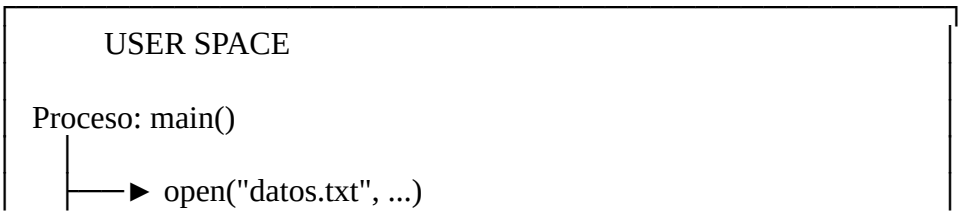
Respuesta: Dos: uno para open() y otro para close().

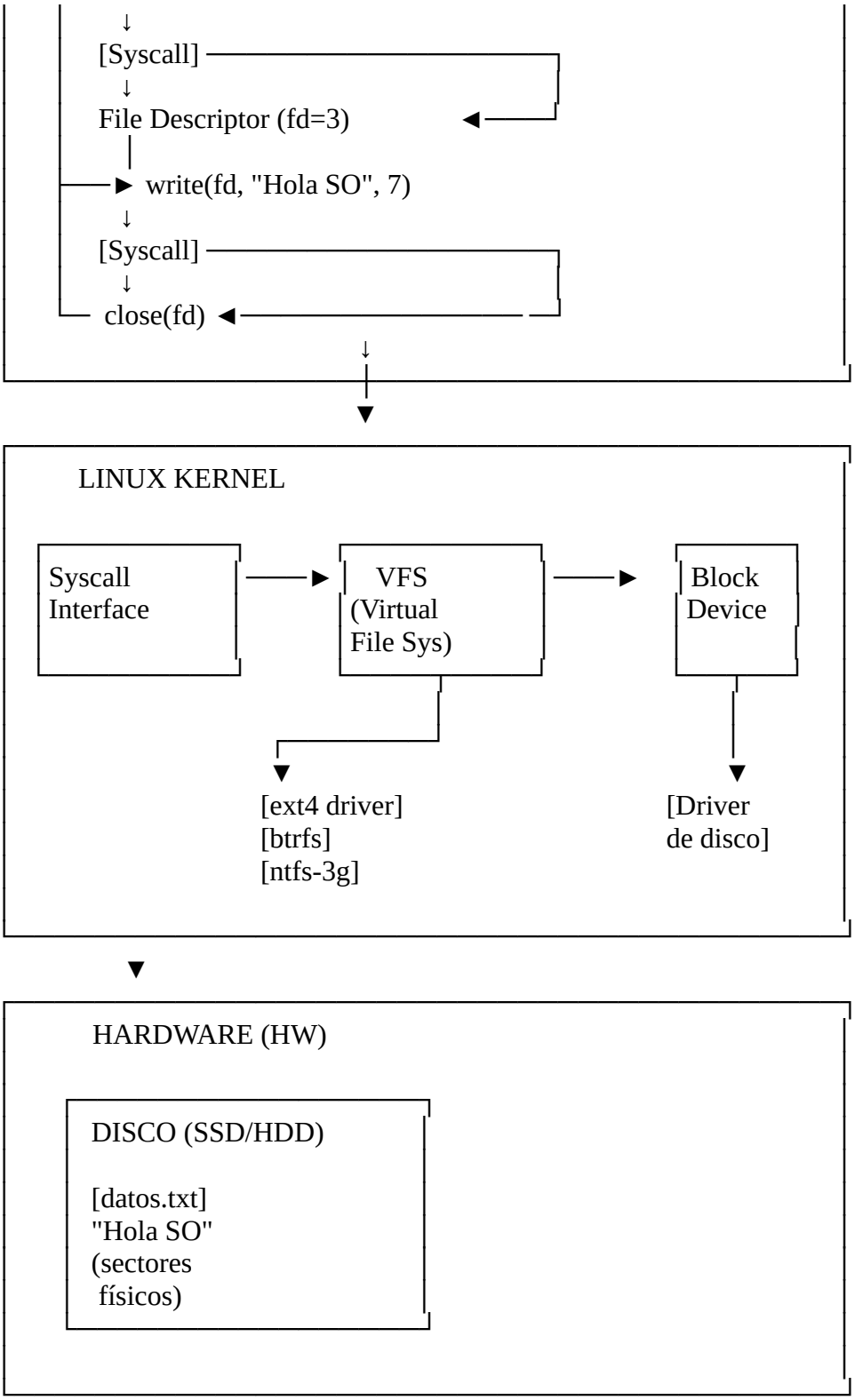
VFS: Virtual File System

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





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

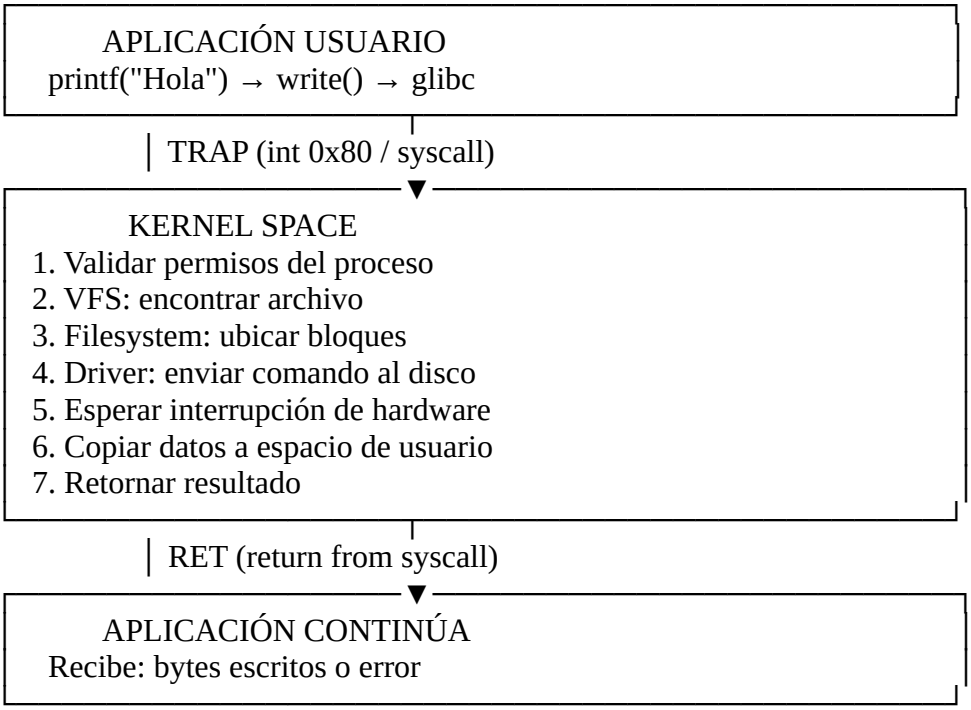
Concepto	Conexión con System Calls
Kernel	Es quien implementa y ejecuta las syscalls
Modo Usuario/Supervisor	La syscall es el puente entre ambos modos
Software de sistema	Proporciona wrappers convenientes sobre syscalls crudas

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.