Introducción a Flujo de Control Excepcional

Diego Feroldi

Arquitectura del Computador [*]
Departamento de Ciencias de la Computación
FCEIA-UNR

[*] Actualizado 30 de junio de 2025 (D. Feroldi, `feroldi@fceia.unr.edu.ar`)

Índice

1. Introducción

Desde que se enciende un procesador hasta que se apaga, el contador de programa (PC) toma una secuencia de direcciones que representan instrucciones a ejecutar. Cada paso de una instrucción a la siguiente se llama *transferencia de control*, y la sucesión de estas transferencias constituye el *flujo de control* del procesador.

El flujo de control más simple ocurre cuando las instrucciones están una al lado de la otra en memoria. Sin embargo, los cambios bruscos en este flujo, como los causados por instrucciones de salto, llamada o retorno de función, son necesarios para reaccionar a cambios en el estado interno del programa (por ejemplo, en sus variables).

Pero también existen situaciones externas al programa que requieren atención, como interrupciones de temporizadores, llegada de paquetes de red, lecturas de disco o finalización de procesos hijos. Para atender estos eventos, los sistemas modernos introducen cambios abruptos en el flujo de control, conocidos como **Flujo de Control Excepcional** (*Exceptional Control Flow*, ECF).

El Flujo de Control Excepcional se manifiesta en distintos niveles del sistema:

- En hardware, eventos externos disparan saltos hacia manejadores de excepciones.
- En el sistema operativo, el kernel cambia entre procesos mediante *cambios de contexto*.
- En el nivel de aplicación, los procesos pueden enviarse señales entre sí para ejecutar manejadores.
- Incluso dentro de un mismo programa, es posible realizar saltos no locales frente a errores.

2. Excepciones

Las excepciones son una forma de flujo de control excepcional que se implementa en parte por el hardware y en parte por el sistema operativo. Debido a que su implementación es parcialmente realizada en hardware, los detalles varían según el sistema. Sin embargo, las ideas básicas son las mismas para todos los sistemas. Una **excepción** es un cambio abrupto en el flujo de control en respuesta a alguna variación en el estado del procesador. La Figura 1 muestra la idea básica.

En la figura, el procesador está ejecutando alguna instrucción cuando ocurre un cambio significativo en su estado debido a un *evento*. El evento puede estar directamente relacionado con la ejecución de la instrucción actual. Por ejemplo, puede ocurrir un fallo de página en memoria virtual, un desbordamiento aritmético o que una instrucción intente dividir por cero. Por otro lado, el evento puede no estar relacionado con la ejecución de la instrucción actual. Por ejemplo, puede activarse un temporizador del sistema o completarse una solicitud de E/S.

En cualquier caso, cuando el procesador detecta que ha ocurrido el evento, realiza una llamada a procedimiento indirecto (la excepción), a través de una tabla de saltos llamada tabla de excepciones, hacia una subrutina del sistema operativo (el manejador de excepciones) diseñada específicamente para procesar ese tipo particular de evento.

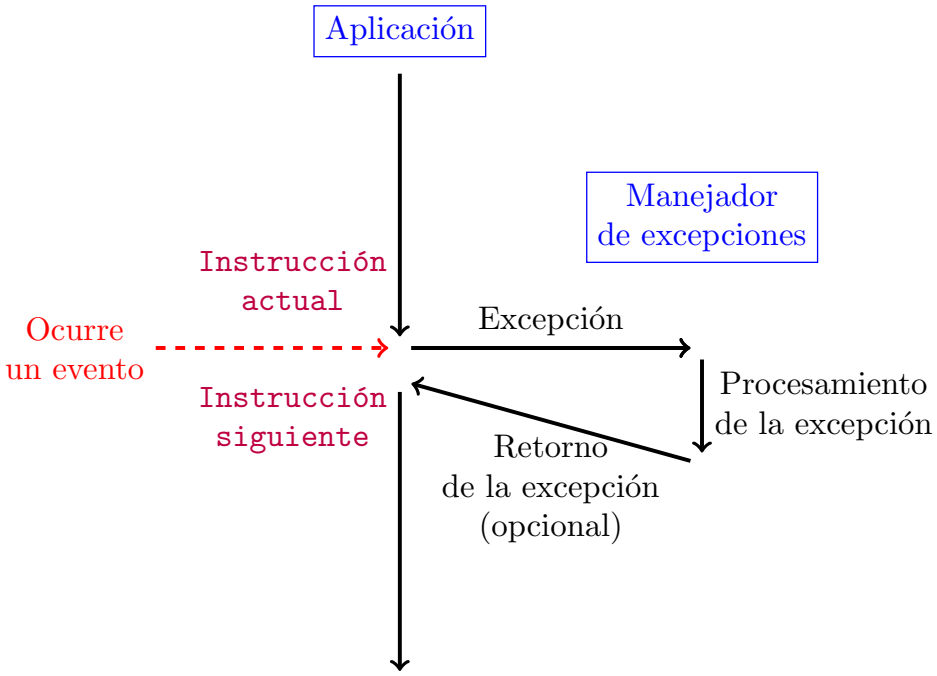

Figura 1: Diagrama esquemático de una excepción.

Cuando el manejador de excepciones termina su procesamiento, ocurre una de tres cosas, dependiendo del tipo de evento que causó la excepción:

1. El manejador devuelve el control a la instrucción actual, es decir, la instrucción que se estaba ejecutando cuando ocurrió el evento.
2. El manejador devuelve el control a la instrucción siguiente, la instrucción que habría ejecutado a continuación si no hubiera ocurrido la excepción.
3. El manejador aborta el programa interrumpido.

3. Clases de excepciones

Las excepciones pueden dividirse en cuatro clases: interrupciones, trampas, fallas y abortos. La Tabla 1 resume las características de estas clases.

3.1. Interrupciones

Las interrupciones ocurren de manera asíncrona como resultado de señales provenientes de dispositivos de E/S que son externos al procesador. Las interrupciones por hardware son asíncronas en el sentido de que no son provocadas por la ejecución de una instrucción en particular. Los manejadores de excepciones para interrupciones por hardware suelen llamarse *manejadores de interrupciones*.

La Figura 2 resume el procesamiento de una interrupción. Los dispositivos de E/S, como adaptadores de red, controladores de disco y temporizadores, generan interrupciones

Tabla 1: Clases de excepciones.

Clase	**Causa**	**Asíncrona/Sínc.**	**Comportamiento al retornar**
Interrupción	Señal de un dispositivo de E/S.	Asíncrona	Siempre retorna a la instrucción siguiente.
Trampa	Excepción intencional.	Síncrona	Siempre retorna a la instrucción siguiente.
Falla	Error potencialmente recuperable.	Síncrona	Puede retornar a la instrucción actual.
Aborto	Error no recuperable.	Síncrona	Nunca retorna.

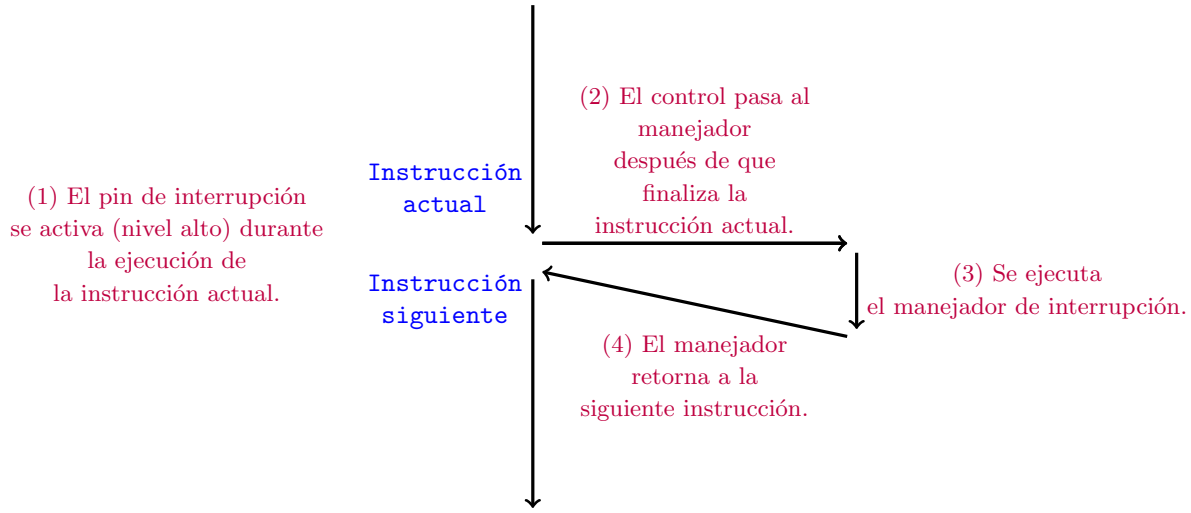

Figura 2: Manejo de interrupciones.

al activar un pin en el chip del procesador y colocan en el bus del sistema el número de excepción que identifica al dispositivo que causó la interrupción.

Después de que la instrucción actual termina de ejecutarse, el procesador detecta que el pin de interrupción se ha activado (nivel alto), lee el número de excepción desde el bus del sistema y luego invoca al manejador de interrupciones correspondiente. Cuando el manejador finaliza, devuelve el control a la siguiente instrucción (es decir, a la instrucción que habría seguido en el flujo de control si la interrupción no hubiera ocurrido). El efecto es que el programa continúa ejecutándose como si la interrupción nunca hubiera ocurrido.

3.2. Trampas y llamadas al sistema

Las trampas son excepciones intencionales que ocurren como resultado de la ejecución de una instrucción. Al igual que los manejadores de interrupciones, los manejadores de trampas devuelven el control a la siguiente instrucción. El uso más importante de las trampas es proporcionar una interfaz similar a una llamada a procedimiento entre los programas de usuario y el núcleo del sistema, conocida como *llamada al sistema (syscall)*.

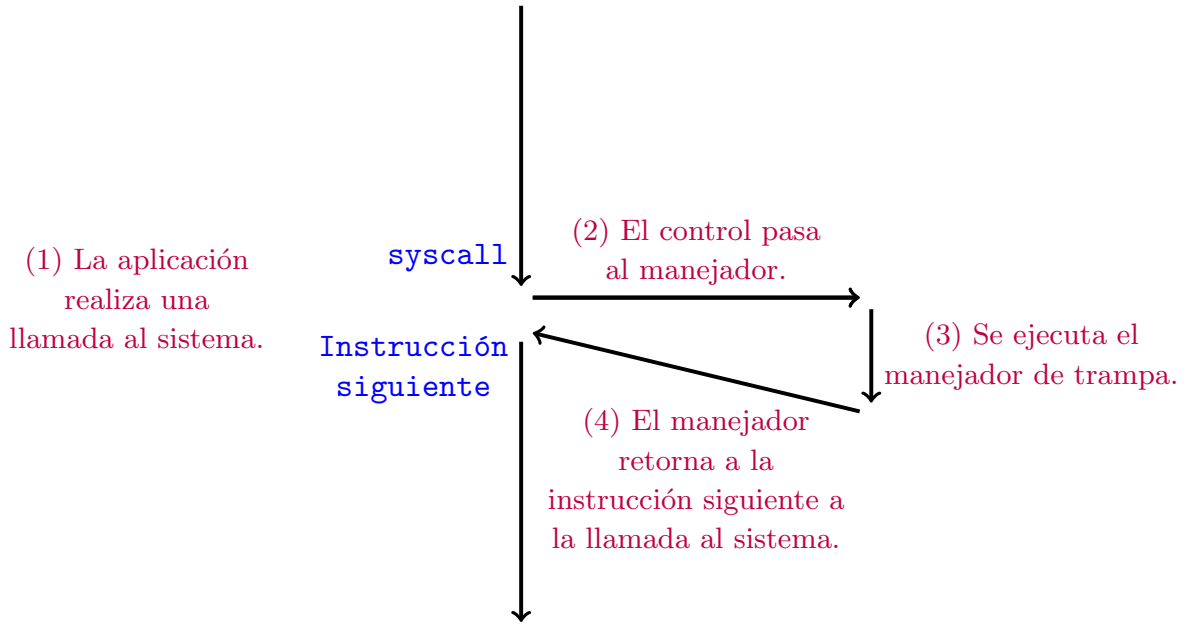

Figura 3: Manejador de trampas.

Los programas de usuario a menudo necesitan solicitar servicios al núcleo, como leer un archivo (*read*), crear un nuevo proceso (*fork*), cargar un nuevo programa (*execve*) o finalizar el proceso actual (*exit*). Para permitir un acceso controlado a estos servicios del núcleo, los procesadores proporcionan una instrucción especial `syscall n` que los programas de usuario pueden ejecutar cuando desean solicitar el servicio número n. La ejecución de la instrucción `syscall` provoca una trampa hacia un manejador de excepciones que decodifica el argumento y llama a la rutina del núcleo correspondiente. La Figura 3 resume el procesamiento de una llamada al sistema.

Desde la perspectiva del programador, una llamada al sistema es idéntica a una llamada a función regular. Sin embargo, sus implementaciones son bastante diferentes. Las funciones regulares se ejecutan en modo usuario, lo que restringe los tipos de instrucciones que pueden ejecutar, y acceden a la misma pila que la función que las llamó. Una llamada al sistema se ejecuta en *modo kernel*, lo que le permite ejecutar instrucciones privilegiadas y acceder a una pila definida en el núcleo.

3.3. Fallas

Las fallas resultan de condiciones de error que un manejador podría corregir. Cuando ocurre una falla, el procesador transfiere el control al manejador de fallas. Si el manejador puede corregir la condición de error, devuelve el control a la instrucción que causó la falla, reejecutándola. De lo contrario, el manejador retorna a una rutina de aborto en el núcleo que termina el programa de aplicación que causó la falla. La Figura 4 resume el procesamiento de una falla.

Un ejemplo clásico de falla es la excepción de fallo de página (*page fault*), que ocurre cuando una instrucción referencia una dirección virtual cuya página correspondiente no

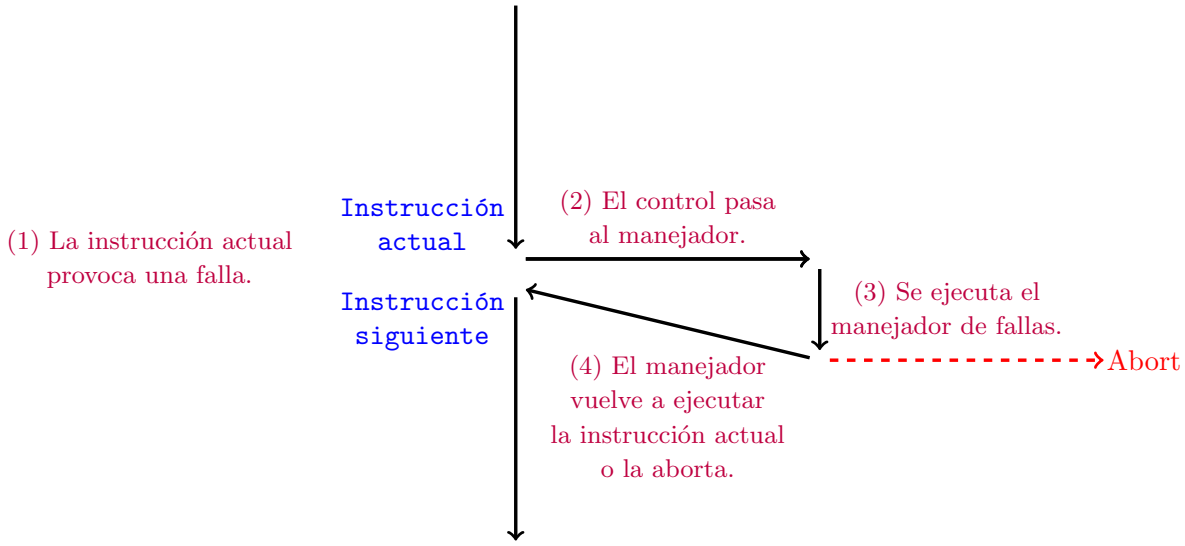

Figura 4: Manejo de fallas.

está presente en la memoria y, por lo tanto, debe ser recuperada desde el disco.

3.4. Abortos

Los abortos resultan de errores fatales irrecuperables, típicamente errores de hardware como errores de paridad que ocurren cuando se corrompen bits en la memoria DRAM o SRAM. Los manejadores de abortos nunca devuelven el control al programa de aplicación. Como se muestra en la Figura 5, el manejador retorna el control a una rutina de aborto que termina el programa de aplicación.

3.5. Excepciones en sistemas Linux/x86-64

Para concretar un poco más, veamos algunas de las excepciones definidas para sistemas x86-64. Existen hasta 256 tipos diferentes de excepciones. Los números en el rango de 0 a 31 corresponden a excepciones definidas por los arquitectos de Intel y, por lo tanto, son idénticas para cualquier sistema x86-64. Los números en el rango de 32 a 255 corresponden a interrupciones y trampas definidas por el sistema operativo. La Tabla 2 muestra algunos ejemplos.

Tabla 2: Ejemplos de excepciones en sistemas x86-64.

Número de excepción	**Descripción**	**Clase de excepción**
0	Error de división	Falla
13	Falla de protección general	Falla
14	Falla de página	Falla
18	Verificación de máquina	Aborto
32–255	Excepciones definidas por el SO	Interrupción o trampa

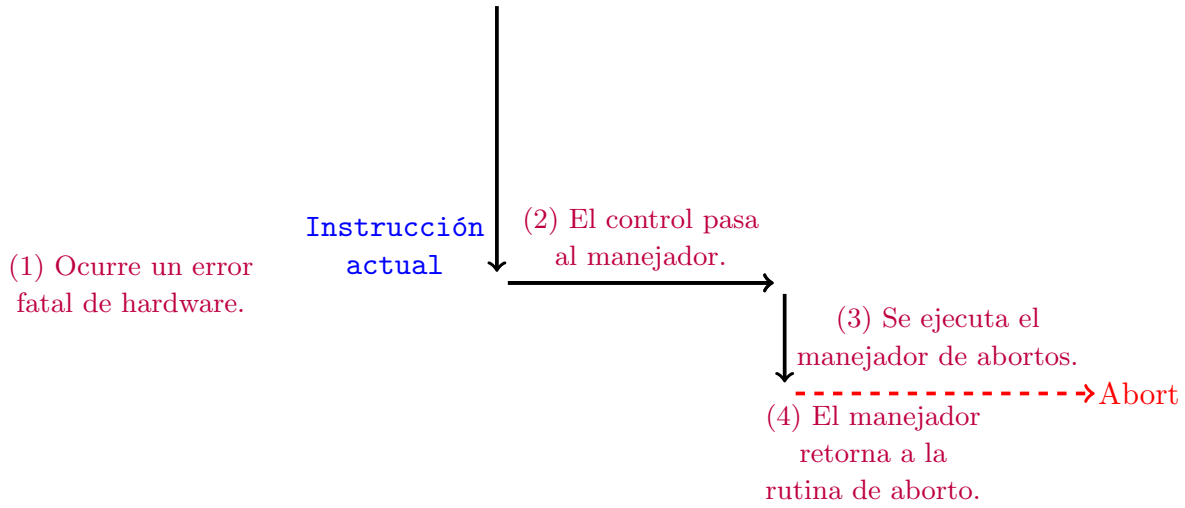

Figura 5: Manejador de abortos.

4. Llamadas al sistema en Linux/x86-64

Una **syscall** (o llamada al sistema) es un mecanismo que permite a los programas en modo usuario interactuar con el sistema operativo, accediendo a recursos o servicios que requieren permisos especiales o privilegios del kernel[1]. Las syscalls son esenciales porque proporcionan una interfaz controlada y segura para que las aplicaciones puedan realizar tareas críticas como leer y escribir en archivos, asignar memoria, comunicarse con dispositivos de hardware o gestionar procesos. En resumen, una syscall es un puente entre las aplicaciones y los servicios privilegiados del sistema operativo.

4.1. Funcionamiento básico

Desde la perspectiva de un programador, una llamada al sistema es similar a una llamada a una función regular. Sin embargo, sus implementaciones son bastante diferentes. La diferencia clave entre una llamada al sistema y una llamada a procedimiento es que una llamada al sistema transfiere el control (es decir, realiza un salto) al sistema operativo mientras simultáneamente eleva el nivel de privilegio del hardware. Las aplicaciones de usuario se ejecutan en lo que se conoce como modo de usuario, lo que significa que el hardware restringe lo que las aplicaciones pueden hacer; por ejemplo, una aplicación en modo de usuario generalmente no puede iniciar una solicitud de E/S al disco, acceder a cualquier página de memoria física ni enviar un paquete en la red.

Cuando se inicia una llamada al sistema (generalmente mediante una instrucción especial de hardware llamada **trap**), el hardware transfiere el control a un controlador de

[1]En sistemas operativos, el **kernel** es el núcleo del software que actúa como intermediario entre el hardware y las aplicaciones. Gestiona recursos como la CPU, memoria y dispositivos de entrada/salida, garantizando el acceso controlado y eficiente a estos componentes.

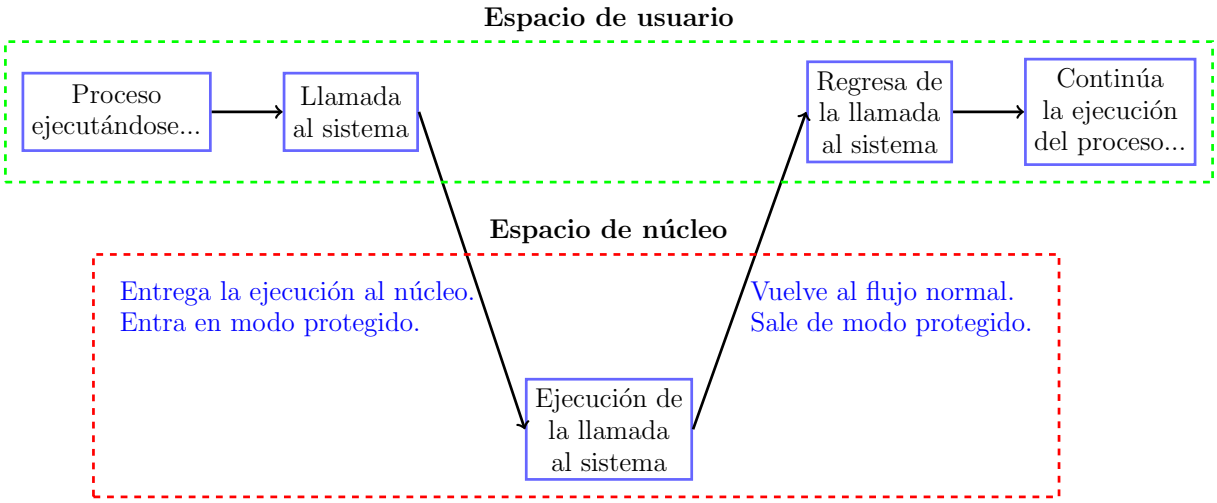

Figura 6: Transición del flujo de ejecución entre el espacio de usuario y el espacio de núcleo durante una llamada al sistema.

trap predefinido (configurado previamente por el sistema operativo) y simultáneamente eleva el nivel de privilegio al modo núcleo (**kernel mode**). En modo núcleo, el sistema operativo tiene acceso completo al hardware del sistema, lo que le permite realizar tareas como iniciar una solicitud de E/S o asignar más memoria a un programa. Cuando el sistema operativo termina de atender la solicitud, devuelve el control al usuario mediante una instrucción especial de retorno de **trap**, que restaura el modo de usuario mientras transfiere el control al punto donde la aplicación se había detenido.

La Fig. 6 muestra de manera esquemática el flujo de ejecución y los cambios entre el espacio de usuario y el espacio del kernel durante una llamada al sistema:

1. **Modo usuario a modo kernel**: Cuando un programa necesita realizar una operación que requiere acceso al kernel (por ejemplo, abrir un archivo), invoca una syscall. Esto genera una interrupción o trampa que transfiere el control al sistema operativo, cambiando el modo de ejecución de usuario a kernel.

2. **Ejecución de la syscall**: El kernel identifica qué servicio se está solicitando (normalmente a través de un número de syscall) y ejecuta el código correspondiente.

3. **Retorno al programa**: Una vez completada la operación, el kernel devuelve el control al programa en modo usuario, junto con un valor que puede indicar éxito o error.

4.2. Ventajas de las syscalls

Seguridad: Restringen el acceso directo al hardware y a las operaciones críticas, reduciendo riesgos de fallos o usos indebidos.

Abstracción: Proveen una interfaz estandarizada para que las aplicaciones interactúen con el sistema operativo, sin preocuparse por detalles de hardware.

4.3. Ejemplos de syscalls comunes

Linux proporciona una amplia variedad de llamadas al sistema para realizar diferentes operaciones. A continuación, se presentan algunos ejemplos de las llamadas al sistema más comunes. Un listado más completo y detallado está disponible en [5, 6].

4.3.1. Gestión de archivos

- **open()**: Esta llamada al sistema se utiliza para abrir un archivo y obtener un descriptor de archivo, que es un identificador único para el archivo abierto. Recibe como parámetros la ruta del archivo y banderas adicionales, devolviendo un descriptor de archivo.
- **read()**: La llamada al sistema **read()** se utiliza para leer datos de un archivo en un búfer. Recibe como parámetros el descriptor de archivo, la dirección del búfer y el número de bytes a leer, devolviendo la cantidad de bytes leídos.
- **write()**: Esta llamada al sistema se utiliza para escribir datos desde un búfer en un archivo. Recibe como parámetros el descriptor de archivo, la dirección del búfer y el número de bytes a escribir, devolviendo la cantidad de bytes escritos.
- **close()**: La llamada al sistema **close()** se utiliza para liberar un descriptor de archivo y cerrar el archivo correspondiente. Recibe como parámetro el descriptor de archivo y devuelve 0 si tiene éxito.
- **stat()**: La llamada al sistema **stat()** recupera información sobre un archivo, como su tamaño, permisos, propiedad y marcas de tiempo. Recibe como parámetro la ruta del archivo y llena una estructura `struct stat` con la información del archivo. Esta llamada se utiliza frecuentemente para consultar metadatos de archivos y realizar operaciones basadas en los atributos del archivo.

4.3.2. Gestión de procesos

- **fork()**: La llamada al sistema **fork()** crea un nuevo proceso duplicando el proceso que la invoca. Devuelve valores diferentes al proceso padre y al proceso hijo. El proceso hijo recibe un valor de retorno de 0, mientras que el proceso padre recibe el ID del proceso hijo. Esta llamada permite la creación de ejecución concurrente o paralela en un programa.
- **execve()**: La llamada al sistema **execve()** reemplaza el proceso actual con un nuevo programa. Carga un nuevo programa en el espacio de memoria del proceso actual e inicia su ejecución. Esta llamada se utiliza típicamente para ejecutar programas o scripts externos.
- **wait()**: Bloquea al proceso padre hasta que un proceso hijo termina.
- **exit()**: Termina el proceso actual y devuelve un código de salida al sistema operativo.

- **kill()**: Se utiliza para enviar señales a un proceso o grupo de procesos. Aunque su nombre sugiere que se usa exclusivamente para terminar procesos, en realidad es una interfaz general para la comunicación mediante señales.

4.3.3. Gestión de memoria

- **malloc()**: La llamada al sistema **malloc()** se utiliza para la asignación dinámica de memoria. Reserva una cantidad específica de memoria y devuelve un puntero al bloque de memoria asignado. Esta llamada se utiliza comúnmente cuando el tamaño de la memoria requerida no se conoce en tiempo de compilación.
- **mmap()**: Se utiliza para mapear un archivo o un área de memoria a un espacio de direcciones en el proceso que la invoca.
- **munmap()**: Se utiliza para desmapear un área de memoria previamente mapeada con **mmap()**.
- **brk()**: Se utiliza para gestionar el límite superior del segmento de datos de un proceso, también conocido como el heap.

4.3.4. Comunicación

- **socket**, **send**, **recv**, etc.

Ejemplo

En sistemas basados en Unix/Linux, la syscall `write` se utiliza para escribir datos en un archivo o en la salida estándar. En código C, su uso podría representarse de la siguiente manera:

```c
#include <unistd.h>

int main() {
    const char *msg = "Hola, syscall!\n";
    write(1, msg, 15); // 1 representa la salida estándar
    return 0;
}
```

El primer argumento de `write` es el descriptor de archivo, que identifica el archivo o recurso en el que se desea escribir. Este descriptor puede representar un archivo, un dispositivo o incluso un flujo estándar, como la salida estándar (`stdout`), cuyo valor es 1. En este ejemplo, el descriptor corresponde a `stdout`. El segundo argumento es un puntero al buffer que contiene la secuencia de bytes a escribir, mientras que el tercer argumento especifica la cantidad de bytes a escribir.

Observación

Los programas en C pueden invocar cualquier llamada al sistema directamente utilizando la función syscall correspondiente. Sin embargo, esto rara vez es necesario en la práctica. La biblioteca estándar de C proporciona un conjunto de funciones envolventes convenientes para la mayoría de las llamadas al sistema. Las funciones envolventes empaquetan los argumentos, realizan la llamada al núcleo con la instrucción de llamada al sistema apropiada, y luego devuelven el estado de retorno de la llamada al sistema al programa que la invocó. Por ejemplo, si queremos imprimir por pantalla es más usual usar la instrucción `printf` en lugar de usar la syscall `write`.

4.4. Llamadas al sistema en Assembler

Las llamadas al sistema se proporcionan en sistemas x86-64 mediante una instrucción llamada `syscall`. Es bastante interesante estudiar cómo los programas pueden utilizar esta instrucción para invocar llamadas al sistema de Linux directamente. Todos los argumentos de las llamadas al sistema de Linux se pasan a través de registros de propósito general en lugar de la pila. Por convención, el registro `rax` contiene el número de la llamada al sistema, con hasta seis argumentos en `rdi`, `rsi`, `rdx`, `r10`, `r8` y `r9`. El primer argumento está en `rdi`, el segundo en `rsi`, y así sucesivamente. Finalmente, la instrucción `syscall` transfiere el control al kernel.

Observaciones

- **Convenciones de llamadas**: En x86-64, las syscalls usan una convención distinta a la de las funciones estándar de C:
  - Las syscalls usan solamente registros para argumentos:
    - `rdi` (primer argumento),
    - `rsi` (segundo argumento),
    - `rdx` (tercer argumento),
    - `r10` (cuarto argumento),
    - `r8` (quinto argumento),
    - `r9` (sexto argumento).
  - El registro `rax` se utiliza para indicar el número de la llamada al sistema.
  - Las funciones C utilizan la pila para algunos argumentos (dependiendo de la convención ABI).
- **Códigos de syscalls**: Los números de syscall son específicos del sistema operativo. En Linux, se pueden encontrar en los archivos de cabecera, como `/usr/include/asm/unistd_64.h`.
- **Errores**: Si la syscall falla, el valor devuelto en `rax` será un número negativo que representa un código de error.

Ejemplo

La syscall `exit` (número 60) se utiliza para terminar un programa con un código de salida.

```
.section .text
.global main
main:
    movq $60, %rax        # Número de syscall: exit
    movq $0, %rdi         # Código de salida: 0
    syscall               # Llamada al sistema
```

Notar que el código anterior es equivalente al siguiente código usando la instrucción `ret`:

```
.section .text
.global main
main:
    movq $0, %rax
    ret
```

Ejemplo

El sistema operativo utiliza la syscall número 1 (`write`) para escribir en un archivo o en la salida estándar.

```
.section .data
msg:  .asciz "Hola, syscall!\n" # Mensaje con terminador nulo
len:  .long 15                  # Longitud del mensaje

.section .text
.global main
main:
    #Primero, llamamos a write
    movq $1, %rax             # Número de syscall: write
    movq $1, %rdi             # File descriptor: 1 (stdout)
    leaq msg, %rsi            # Dirección del mensaje
    movq len, %rdx            # Longitud del mensaje
    syscall                   # Llamada al sistema

    #Luego, llamamos a exit
    movq $60, %rax            # Número de syscall: exit
    xorq %rdi, %rdi           # Código de salida: 0
    syscall                   # Llamada al sistema
```

Ejemplo

La syscall número 0 (`read`) lee datos desde un archivo o la entrada estándar.

```
.section .data
msg:  .asciz "Hola, syscall!\n"  # Mensaje con terminador nulo
len:  .long 15                   # Longitud del mensaje

.section .bss
buffer: .skip 64                  # Reservar un búfer de 64 bytes

.section .text
.global main
main:
    # Primero, llamamos a read
    movq $0, %rax              # Número de syscall: read
    movq $0, %rdi              # File descriptor: 0 (stdin)
    leaq buffer, %rsi          # Dirección del búfer
    movq $64, %rdx             # Tamaño del búfer
    syscall                    # Llamada al sistema

    # Luego, llamamos a write
    movq $1, %rax             # Número de syscall: write
    movq $1, %rdi             # File descriptor: 1 (stdout)
    syscall                   # Llamada al sistema

    # Finalmente, llamamos a exit
    movq $60, %rax             # Número de syscall: exit
    xorq %rdi, %rdi            # Código de salida: 0
    syscall                    # Llamada al sistema
```

Apéndice A: Códigos de syscalls

Existen varios cientos de llamadas al sistema, que pueden agruparse en diferentes categorías. A continuación se muestran los códigos de syscalls más comunes en Linux para la arquitectura **x86-64**. Esta información se basa en el archivo de cabecera `/usr/include/asm/unistd_64.h` y está organizada por categorías para mayor claridad. Para acceder a las definiciones de las llamadas al sistema, utilizar el comando `man` desde la línea de comandos: Por ejemplo: `man 2 exit`. El 2 en la línea de comandos especifica la sección 2 de las páginas del manual.

Syscall	Código	Descripción
Gestión de archivos		
`read`	0	Leer de un archivo o stdin.
`write`	1	Escribir en un archivo o stdout.

`open`	2	Abrir un archivo.
`close`	3	Cerrar un archivo.
`stat`	4	Obtener información sobre un archivo.
`fstat`	5	Obtener información sobre un descriptor de archivo.
`lstat`	6	Similar a `stat`, pero sigue enlaces simbólicos.
`lseek`	8	Cambiar la posición de lectura/escritura en un archivo.

Gestión de procesos

`fork`	57	Crear un nuevo proceso.
`vfork`	58	Similar a `fork`, pero más eficiente.
`execve`	59	Ejecutar un programa.
`exit`	60	Salir de un proceso.
`wait4`	61	Esperar a que un proceso hijo termine.

Gestión de memoria

`mmap`	9	Mapear memoria en el espacio de usuario.
`munmap`	11	Desmapear memoria.
`brk`	12	Cambiar el tamaño del heap del proceso.
`mprotect`	10	Cambiar permisos de una región de memoria.

Gestión de directorios

`getcwd`	79	Obtener el directorio actual.
`chdir`	80	Cambiar el directorio actual.
`mkdir`	83	Crear un directorio.
`rmdir`	84	Eliminar un directorio.

Tiempos y fechas

`time`	201	Obtener la hora actual.
`gettimeofday`	96	Obtener el tiempo real y la zona horaria.
`nanosleep`	35	Suspender la ejecución por un tiempo especificado.

Comunicaciones

`socket`	41	Crear un socket.
`connect`	42	Conectar un socket.
`send`	44	Enviar datos a través de un socket.
`recv`	45	Recibir datos desde un socket.

Scheduling

`sched_setparam`	142	Establece los parámetros de planificación.
`sched_get_priority_max`	146	Devuelve el valor máximo de prioridad.
`sched_setscheduler`	144	Establece la política de planificación.
`sched_yield`	24	El proceso puede ceder voluntariamente el procesador.

Otras

`uname`	63	Obtener información sobre el sistema.
`getpid`	39	Obtener el PID del proceso actual.

`getuid`	102	Obtener el UID del usuario actual.
`kill`	62	Enviar una señal a un proceso.

Apéndice B: Errores en syscalls

En las syscalls de Linux, cuando ocurre un error, normalmente se devuelve un valor negativo (generalmente `-1`) y se establece la variable global `errno` para indicar el tipo de error. A continuación, se muestra una lista de los errores comunes devueltos en las syscalls, junto con sus significados:

Código de Error	Número	Descripción
`EPERM`	-1	Operación no permitida.
`ENOENT`	-2	No existe el archivo o directorio.
`ESRCH`	-3	No existe el proceso.
`EINTR`	-4	Llamada al sistema interrumpida por una señal.
`EIO`	-5	Error de entrada/salida.
`ENXIO`	-6	No hay tal dispositivo o dirección.
`E2BIG`	-7	El argumento es demasiado largo.
`ENOEXEC`	-8	Formato ejecutable erróneo.
`EBADF`	-9	Descritor de archivo no válido.
`ECHILD`	-10	El proceso hijo no existe.
`EAGAIN`	-11	El recurso está temporalmente no disponible.
`ENOMEM`	-12	No hay suficiente memoria.
`EACCES`	-13	Permiso denegado.
`EFAULT`	-14	Dirección errónea.
`ENOTBLK`	-15	Dispositivo de bloque inexistente.
`EBUSY`	-16	Recurso ocupado.
`EEXIST`	-17	El archivo ya existe.
`EXDEV`	-18	Dispositivo cruzado.
`ENODEV`	-19	No existe el dispositivo.
`ENOTDIR`	-20	No es un directorio.
`EISDIR`	-21	Es un directorio.
`EINVAL`	-22	Argumento no válido.
`ENFILE`	-23	Demasiados archivos abiertos en el sistema.
`EMFILE`	-24	Demasiados archivos abiertos por el proceso.
`ENOTTY`	-25	No es un dispositivo de terminal.
`ETXTBSY`	-26	Archivo de texto ocupado.
`EFBIG`	-27	Archivo demasiado grande.
`ENOSPC`	-28	No hay espacio en el dispositivo.
`ESPIPE`	-29	Operación no permitida en un descriptor de archivo.
`EROFS`	-30	Solo lectura en el sistema de archivos.
`EMLINK`	-31	Demasiados enlaces duros.
`EPIPE`	-32	El canal está roto.
`EDOM`	-33	Argumento fuera del dominio de la función.
`ERANGE`	-34	Resultado fuera de rango.

EDEADLK	-35	Condición de interbloqueo.
ENAMETOOLONG	-36	El nombre del archivo es demasiado largo.
ENOLCK	-37	No hay bloqueos disponibles.
ENOSYS	-38	Llamada al sistema no implementada.
ENOTEMPTY	-39	El directorio no está vacío.
ELOOP	-40	Demasiados enlaces simbólicos.
ENOMSG	-42	No hay mensajes.
EIDRM	-43	Identificador de mensaje eliminado.
ECHRNG	-44	Rango de secuencia de caracteres erróneo.
EL2NSYNC	-45	Sincronización de nivel 2 requerida.
EL3HLT	-46	Nivel 3 de detención requerido.
EL3RST	-47	Reinicio de nivel 3 requerido.
EIGN	-48	Identificador de señales incorrecto.
EUNATCH	-49	No se puede desvincular el archivo.
ENOCSI	-50	No hay control de señales.
EL2HLT	-51	Detención de nivel 2 requerida.
EBADE	-52	Error en la estructura de archivo de mensajes.
EBADR	-53	Error de solicitud de mensaje.
EXFULL	-54	El almacén de mensajes está lleno.
ENOANO	-55	No hay identificador de la dirección.
EBADRQC	-56	Solicitud de dirección errónea.
EBADSLT	-57	Subíndice de tabla erróneo.
EDEADLOCK	-58	Condición de interbloqueo.
ENOLCK	-59	Sin recursos para realizar bloqueos.

Referencias

[1] Bryant, R. E., & O'Hallaron, D. R. (2011). Computer systems: a programmer's perspective. Prentice Hall.

[2] Wolf, G., Ruiz, E., Bergero, F., & Meza, E. (2015). Fundamentos de sistemas operativos.

[3] Stallings, W. (2011). Operating systems: internals and design principles. Prentice Hall Press.

[4] Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2018). Operating systems: Three easy pieces.

[5] The Chromium Projects. Linux System Call Table. URL: `https://www.chromium.org/chromium-os/developer-library/reference/linux-constants/syscalls/#x86_64_1`. Fecha de consulta: 27 de diciembre de 2024.

[6] Marat Pekker. Linux System Calls. URL: `https://mpdev.hashnode.dev/linux-system-calls`, Fecha de consulta: 26 de diciembre de 2024.