



## **Tarea 2: Rastreador de System Calls**

Instituto Tecnológico de Costa Rica  
Escuela de Ingeniería en Computación  
Centro Académico de Alajuela

**Estudiante:**

William Alfaro Quirós

**Asignatura:**

IC-6600: Principios de Sistemas Operativos

**Grupo:**

20

**Profesor:**

Ing. Kevin Moraga, MSc.

**Fecha de entrega:**

30 de marzo del 2025

Semestre I

# Índice

1. Introducción	2
2. Ambiente de desarrollo	2
3. Estructuras de datos usadas y funciones	3
4. Instrucciones para ejecutar el programa	4
5. Actividades realizadas por estudiante	5
6. Autoevaluación	5
7. Lecciones Aprendidas	6
8. Bibliografía	8
9. Anexos	9

## 1. Introducción

En los sistemas operativos modernos, las llamadas al sistema (*system calls*) representan el mecanismo principal mediante el cual los programas en espacio de usuario solicitan servicios al núcleo del sistema operativo. Estas llamadas permiten acceder a recursos fundamentales como archivos, memoria y procesos, siendo un componente esencial para la ejecución de cualquier aplicación. Comprender y analizar estas llamadas es clave para el diagnóstico, la depuración y la seguridad del sistema.

Esta tarea consistió en el desarrollo de una herramienta en lenguaje C que permite interceptar y registrar las llamadas al sistema realizadas por un programa especificado. A través del uso de la función `ptrace()`, comúnmente empleada por depuradores como `gdb`, el rastreador implementado puede observar en tiempo real la ejecución del programa objetivo, mostrando un resumen de syscalls, un seguimiento detallado (`-v`) o un modo paso a paso (`-V`) que detiene la ejecución entre cada llamada. Este tipo de herramienta resulta particularmente útil para conocer el comportamiento interno de programas, así como para tareas de análisis forense y de seguridad informática [1].

Además, la implementación de este rastreador ofrece una experiencia práctica con técnicas de bajo nivel relacionadas con la creación y el seguimiento de procesos, fortaleciendo el entendimiento de la interfaz entre espacio de usuario y kernel, especialmente en entornos UNIX/Linux, donde la introspección de procesos es ampliamente utilizada tanto por herramientas de sistema como por malware avanzado [2].

## 2. Ambiente de desarrollo

Para la elaboración de esta tarea se utilizó un entorno de desarrollo controlado basado en sistemas GNU/Linux, el cual permitió realizar pruebas exhaustivas del rastreador de llamadas al sistema y garantizar su correcto funcionamiento. A continuación, se detallan las principales herramientas y configuraciones empleadas:

- **Lenguaje de programación:** C (C99), por su eficiencia, control a bajo nivel y compatibilidad directa con llamadas al sistema y bibliotecas POSIX.
- **Sistema operativo:** Ubuntu 22.04.3 LTS de 64 bits, sobre el cual se realizaron tanto el desarrollo como las pruebas, aprovechando su compatibilidad con herramientas de bajo nivel como `ptrace()`.
- **Compilador:** GCC versión 11.4.0, utilizado para compilar el código fuente del rastreador mediante el `Makefile` proporcionado.
- **Editor de código:** Visual Studio Code, empleado para facilitar la edición, navegación y depuración del código fuente.
- **Herramientas auxiliares:**
  - `make`: Para la automatización de la compilación.
  - `gdb`: Utilizado ocasionalmente para inspeccionar el comportamiento del proceso rastreador.
  - `strace`: Como herramienta de referencia para validar la salida del rastreador implementado.

- **Control de versiones:** Git, empleado para llevar el historial de cambios y organizar el trabajo de forma modular y ordenada a lo largo del proyecto.
- **Plataforma de ejecución:** Procesador Intel x86\_64 sobre máquina virtual con 4 GB de RAM y entorno de terminal Bash.

### 3. Estructuras de datos usadas y funciones

Durante el desarrollo del rastreador de llamadas al sistema se implementaron estructuras de datos simples pero funcionales que permitieron organizar, mapear y presentar la información capturada de manera eficiente. A continuación se describen las estructuras clave y las funciones principales empleadas en la implementación.

#### Estructuras de datos utilizadas

- **Arreglo `syscall_names[]`:** Se implementó un arreglo de tipo `char *` que contiene los nombres de las llamadas al sistema para la arquitectura `x86_64`, indexado por el número de `syscall`. Esta estructura facilita la conversión de números de `syscall` a nombres legibles por el usuario, permitiendo así que el rastreador imprima resultados interpretables.
- **Mapa de frecuencia (arreglo de enteros):** Se utilizó un arreglo de tipo `int syscall_count[512]` donde cada índice representa una `syscall`, y su valor indica cuántas veces se ha invocado dicha `syscall` durante la ejecución del programa objetivo. Esta estructura se emplea para generar el resumen estadístico al finalizar el rastreo.

#### Funciones principales implementadas

- `main(int argc, char *argv[])` Función principal que interpreta los argumentos proporcionados por el usuario, determina el modo de operación (`normal`, `-v` o `-V`) y lanza el proceso rastreador.
- `child_process(char *prog[], int verbose)` Esta función representa el proceso hijo. Se encarga de ejecutar el programa objetivo mediante `execvp()` y notifica al proceso padre que se encuentra listo para ser rastreado.
- `parent_process(pid_t child_pid, int verbose, int step)` Ejecutada por el proceso padre, esta función monitorea al hijo utilizando `ptrace()` y detecta cuándo ocurre una `syscall`. Según el modo activo, imprime la información correspondiente y actualiza el mapa de frecuencias.
- `handle_syscall(pid_t pid)` Extrae el número de `syscall` desde los registros del proceso hijo usando `PTTRACE_PEEKUSER`, lo mapea con su nombre a través del arreglo `syscall_names[]` y lo imprime en pantalla si es requerido.
- `print_summary()` Al finalizar la ejecución del proceso objetivo, esta función recorre el arreglo de conteo de `syscalls` e imprime únicamente aquellas llamadas que fueron utilizadas durante la ejecución, junto con su cantidad respectiva.

## 4. Instrucciones para ejecutar el programa

Esta sección describe el procedimiento para compilar el rastreador de llamadas al sistema y ejecutarlo en sus distintos modos de operación. El programa fue diseñado para facilitar su uso a través de una interfaz de línea de comandos intuitiva.

### Paso 1: Compilación del programa

Desde la raíz del repositorio, ejecutar el siguiente comando:

```
make
```

Este comando compilará el código fuente y generará el ejecutable `rastreador`.

### Paso 2: Ejecución del rastreador

La sintaxis general para utilizar el programa es:

```
./rastreador [opciones del rastreador] Prog [argumentos de Prog]
```

Donde:

- `[opciones del rastreador]` controlan el comportamiento del rastreador.
- `Prog` es el programa objetivo a rastrear.
- `[argumentos de Prog]` son los parámetros que se le pasan al programa objetivo (no son procesados por el rastreador).

### Paso 3: Modos de operación disponibles

El rastreador soporta tres modos de ejecución:

- **Modo normal:** No se indican opciones. El rastreador ejecuta el programa objetivo y, al finalizar, muestra un resumen de las llamadas al sistema utilizadas.
- **Modo detallado -v:** Muestra en tiempo real cada syscall detectada durante la ejecución del programa.
- **Modo paso a paso -V:** Similar al anterior, pero requiere que el usuario presione una tecla para continuar después de cada syscall.

### Paso 4: Ejemplos de uso

- Ejecutar `ls` y mostrar solo el resumen de syscalls:

```
./rastreador ls
```

- Ejecutar `ls -l` y mostrar cada syscall en tiempo real:

```
./rastreador -v ls -l
```

- Ejecutar `sleep 1` en modo paso a paso:

```
./rastreador -V sleep 1
```

## Consideraciones adicionales

- El rastreador debe ejecutarse desde una terminal con permisos suficientes.
- Para rastrear programas que requieren privilegios, se debe anteponer `sudo` al comando.
- El rastreador ha sido validado con múltiples utilidades estándar como `ls`, `cat`, `echo` y `sleep`.

## 5. Actividades realizadas por estudiante

A continuación, se detallan las principales actividades llevadas a cabo por el estudiante durante el desarrollo de la Tarea 2, incluyendo el trabajo de investigación, implementación, pruebas y documentación del rastreador de llamadas al sistema:

Cuadro 1: Actividades realizadas por estudiante

Fecha	Descripción breve de la actividad	Horas
22 de marzo	Investigación sobre llamadas al sistema, uso de <code>ptrace</code> y análisis de herramientas similares como <code>strace</code> .	3
23 de marzo	Configuración del entorno de desarrollo, creación del repositorio e implementación inicial de <code>rastreador.c</code> .	5
24 de marzo	Construcción del arreglo de nombres de syscalls y creación de <code>syscall_map.c</code> y <code>syscall_map.h</code> .	3
25 de marzo	Implementación de los modos <code>-v</code> y <code>-V</code> , manejo de argumentos y sincronización padre-hijo.	4
28 de marzo	Pruebas con distintos programas del sistema, validación del rastreador y manejo de errores.	4
28 de marzo	Revisión final del código, limpieza y documentación. Preparación del informe en LaTeX.	3
29 de marzo	Ajustes en la documentación del código y entrega.	1
Total de horas invertidas		23

## 6. Autoevaluación

El programa implementado cumplió satisfactoriamente con los objetivos planteados al inicio del proyecto. Se logró desarrollar un rastreador de llamadas al sistema funcional, que permite observar la ejecución de un proceso y registrar cada syscall invocada, ya sea de forma resumida, detallada o en modo paso a paso. Además, se implementó un sistema de mapeo para más de 450 llamadas al sistema de la arquitectura `x86_64`, y se obtuvo experiencia práctica con el uso de `ptrace`, la manipulación de procesos y el control del flujo de ejecución.

El resultado final cumple con los requisitos planteados y ofrece una base sólida para los requisitos de la tarea y análisis de procesos en sistemas Linux.

Considerando los objetivos cumplidos y los resultados obtenidos, se autoevalúa esta tarea con una calificación de **100 (nota máxima)**.

## Limitaciones identificadas

Entre las principales limitaciones del programa destacan:

- El rastreador no interpreta los argumentos de cada syscall, únicamente muestra el nombre de la llamada detectada.
- La herramienta fue diseñada para programas de ejecución rápida y sencilla; con procesos más complejos podría requerirse optimización de rendimiento o mejoras en la visualización.

A continuación, se resume en la Tabla 2 el registro principal de commits realizados durante el desarrollo del proyecto. El repositorio completo del proyecto se encuentra disponible en el siguiente enlace:

<https://github.com/BillyyBoyy/Tarea2-S0>

Cuadro 2: Registro de Commits realizados en el repositorio

Fecha	Descripción del commit
28 de marzo	Creación de directorio de archivos y primeros avances.
28 de marzo	Creación de rastreador.
28 de marzo	Cambios y lista de syscalls.
28 de marzo	Creación de Makefile y el header para la lista de syscalls.
28 de marzo	Implementación de nueva función y documentación interna.
28 de marzo	Documentación en LaTeX incluída.
29 de marzo	Ajustes en documentación.
29 de marzo	Ajustes en la documentación del código y entrega.

## 7. Lecciones Aprendidas

Durante el desarrollo de esta tarea se obtuvieron aprendizajes significativos que contribuyen tanto a la formación técnica como al fortalecimiento de habilidades prácticas en sistemas operativos. A continuación se resumen las lecciones más relevantes:

- **Interacción entre usuario y kernel:** Se comprendió en mayor profundidad cómo los programas en espacio de usuario interactúan con el núcleo del sistema mediante llamadas al sistema, lo que permitió visualizar el comportamiento interno de herramientas comunes como `ls`, `cat` o `sleep`.
- **Uso práctico de `ptrace()`:** La implementación del rastreador permitió familiarizarse con la función `ptrace()`, ampliamente utilizada por depuradores. Se comprendió su ciclo de control sobre procesos hijo, así como su capacidad para interceptar señales y obtener registros de CPU.

- **Manejo de procesos padre-hijo:** Se reforzaron los conocimientos sobre la creación de procesos con `fork()` y su monitoreo desde el proceso padre, destacando la importancia de una sincronización precisa para evitar condiciones de carrera.
- **Diseño modular y escalable:** El diseño del arreglo de syscalls y el uso de estructuras separadas como `syscall_map.c` y `syscall_map.h` permitió mantener un código limpio, organizado y fácilmente ampliable.
- **Importancia del control de versiones:** El uso de Git desde el inicio del proyecto facilitó el manejo ordenado del código, la experimentación con distintas ideas y el seguimiento de cambios importantes.
- **Valor del testing incremental:** Realizar pruebas frecuentes con distintos programas objetivo permitió detectar errores rápidamente, mejorar la robustez del rastreador y validar el comportamiento de cada modo de operación.



## 8. Bibliografía

- [1] A. Silberschatz, P. B. Galvin y G. Gagne, *Operating System Concepts*, 10.<sup>a</sup> ed., Hoboken, NJ, USA: Wiley, 2020.
- [2] R. Love, *Linux System Programming*, 1.<sup>a</sup> ed., Sebastopol, CA, USA: O'Reilly Media, 2005.

## 9. Anexos

### Anexo A: Ejemplo de ejecución del rastreador en modo -v

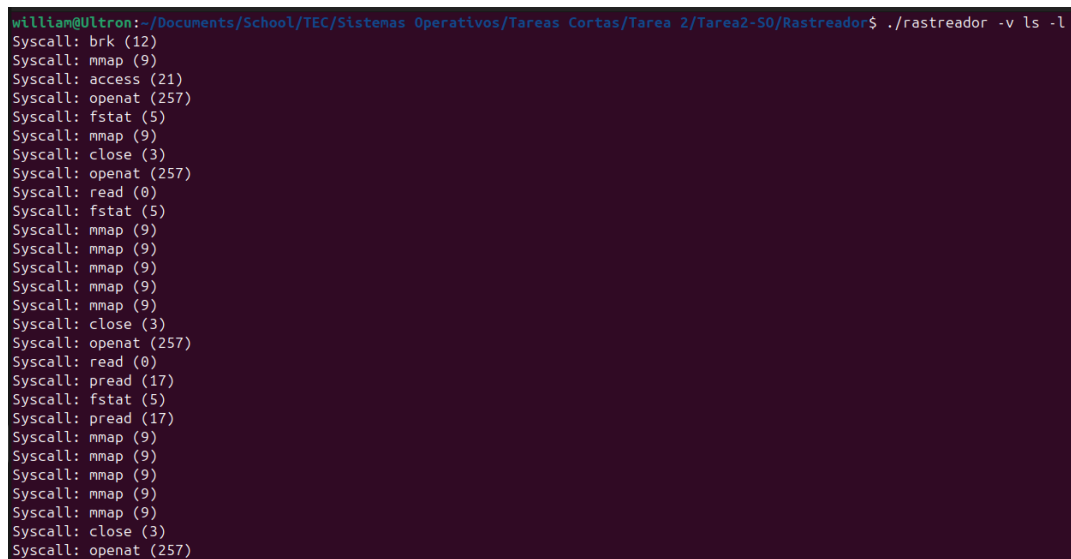
En este anexo se presenta una captura de pantalla ilustrativa del funcionamiento del programa *rastreador* en modo detallado (-v). Para este ejemplo, se ejecutó el siguiente comando desde la terminal:

```
./rastreador -v ls -l
```

El rastreador inicia un proceso hijo con el comando indicado, y mediante el uso de `ptrace()` intercepta todas las llamadas al sistema que este realiza. En el modo -v, cada syscall detectada es impresa en pantalla de forma inmediata, junto con su número respectivo en la arquitectura x86\_64.

Este tipo de monitoreo en tiempo real es útil para observar el comportamiento interno de utilidades estándar como `ls`, revelando llamadas comunes como `brk`, `mmap`, `read`, `fstat`, `openat`, entre muchas otras.

#### Captura de ejecución:



```
william@Ultron:~/Documents/School/TEC/Sistemas Operativos/Tareas Cortas/Tarea 2/Tarea2-S0/Rastreador$ ./rastreador -v ls -l
Syscall: brk (12)
Syscall: mmap (9)
Syscall: access (21)
Syscall: openat (257)
Syscall: fstat (5)
Syscall: mmap (9)
Syscall: close (3)
Syscall: openat (257)
Syscall: read (0)
Syscall: fstat (5)
Syscall: mmap (9)
Syscall: mmap (9)
Syscall: mmap (9)
Syscall: mmap (9)
Syscall: mmap (9)
Syscall: close (3)
Syscall: openat (257)
Syscall: read (0)
Syscall: pread (17)
Syscall: fstat (5)
Syscall: pread (17)
Syscall: mmap (9)
Syscall: mmap (9)
Syscall: mmap (9)
Syscall: mmap (9)
Syscall: mmap (9)
Syscall: close (3)
Syscall: openat (257)
```

Figura 1: Ejemplo de ejecución del comando `./rastreador -v ls -l`. Imagen recortada para fines ilustrativos.

La Figura 1 muestra una parte del flujo de syscalls capturado durante la ejecución de `ls -l`, donde se puede ver cómo el programa accede a recursos del sistema, abre archivos, mapea memoria, y lee datos antes de mostrar el listado del directorio.

## Anexo B: Tabla de llamadas al sistema (syscalls)

En este anexo se presenta una tabla con las llamadas al sistema (*syscalls*) utilizadas por el programa rastreador. Esta tabla fue construida a partir de la tabla oficial de syscalls para la arquitectura x86\_64 en sistemas Linux, y se utilizó para crear el arreglo `syscall_names[]` dentro del archivo `syscall_map.c`, el cual permite mapear los números de syscall interceptados mediante `ptrace()`.

0 - read	1 - write	2 - open	3 - close	4 - stat
5 - fstat	6 - lstat	7 - poll	8 - lseek	9 - mmap
10 - mprotect	11 - munmap	12 - brk	13 - rt_sigaction	14 - rt_sigprocmask
15 - rt_sigreturn	16 - ioctl	17 - pread	18 - pwrite	19 - readv
20 - writev	21 - access	22 - pipe	23 - select	24 - sched_yield
25 - mremap	26 - msync	27 - mincore	28 - madvise	29 - shmget
30 - shmat	31 - shmctl	32 - dup	33 - dup2	34 - pause
35 - nanosleep	36 - getitimer	37 - alarm	38 - setitimer	39 - getpid
40 - sendfile	41 - socket	42 - connect	43 - accept	44 - sendto
45 - recvfrom	46 - sendmsg	47 - recvmsg	48 - shutdown	49 - bind
50 - listen	51 - getsockname	52 - getpeername	53 - socketpair	54 - setsockopt
55 - getsockopt	56 - clone	57 - fork	58 - vfork	59 - execve
60 - exit	61 - wait4	62 - kill	63 - uname	64 - semget
65 - semop	66 - semctl	67 - shmdt	68 - msgget	69 - msgsnd
70 - msgrcv	71 - msgctl	72 - fcntl	73 - flock	74 - fsync
...	...	...	...	...
424 - pidfd_send_signal	425 - io_uring_setup	426 - io_uring_enter	427 - io_uring_register	428 - open_tree
429 - move_mount	430 - fsopen	431 - fsconfig	432 - fsmount	433 - fspick
434 - pidfd_open	435 - clone3	436 - close_range	437 - openat2	438 - pidfd_getfd
439 - faccessat2	440 - process_madvise	441 - epoll_pwait2	442 - mount_setattr	443 - quotactl_fd
444 - landlock_create_ruleset	445 - landlock_add_rule	446 - landlock_restrict_self	447 - memfd_secret	448 - process_mrelease
449 - futex_waitv	450 - set_mempolicy_home_node	451 - cachestat		