Rastreador de System Calls

Curso: Sistemas Operativos Avanzados
Profesor: Kevin Moraga
Estudiante: Omar Solís
Fecha: Octubre, 2025

---

1. Introducción

El propósito de esta tarea es desarrollar un rastreador de llamadas al sistema (system calls) para programas en GNU/Linux utilizando Rust. Las llamadas al sistema permiten que un programa interactúe con el kernel del sistema operativo, realizando operaciones como leer o escribir archivos, crear procesos, asignar memoria, entre otras. Este rastreador permite ejecutar un programa cualquiera y registrar todas las syscalls que realiza, mostrando un conteo acumulativo al finalizar y ofreciendo modos de visualización detallada para análisis y depuración.

2. Ambiente de desarrollo

El proyecto se desarrolló en un entorno Linux utilizando las siguientes herramientas:

- Sistema operativo: Ubuntu 22.04 LTS
- Lenguaje de programación: Rust 1.71
- Gestor de paquetes y compilación: Cargo
- Librerías utilizadas:
  - `nix` para operaciones de ptrace y manejo de procesos.
  - `clap` para parseo de argumentos de línea de comando.
  - `prettytable` para mostrar tablas de syscalls de forma legible.
  - `libc` para interoperabilidad con funciones del sistema.
- Editor de código: Visual Studio Code

3. Estructuras de datos y funciones principales

- **`HashMap<u64, &'static str>`**: Mapa que relaciona el número de syscall con su nombre.
- **`build_syscall_map()`**: Inicializa el mapa de syscalls conocidas.
- **`syscall_name()`**: Devuelve el nombre de una syscall a partir de su número.
- **`wait_enter()`**: Pausa la ejecución hasta que el usuario presione Enter (modo `-V`).
- **`path_and_args_to_cstrings()`**: Convierte los argumentos de Rust en CStrings para `execv`.
- **`run_tracer(verbose: bool, step: bool, prog_and_args: Vec<String>)`**: Función principal que realiza el fork, ejecuta el programa hijo, rastrea las syscalls con `ptrace`, y muestra la tabla acumulativa al finalizar.

- Uso de enumeraciones y constantes de Rust para manejo de señales (`Signal`) y estado de procesos (`WaitStatus`).

4. Instrucciones para ejecutar el programa

Para compilar el proyecto se utiliza Cargo, el gestor de Rust. Esto generará el binario en `target/debug/rastreador`. Para compilar en modo release (optimizado) se puede usar `cargo build --release`.

Para ejecutar el programa de manera normal, mostrando únicamente la tabla final de system calls al finalizar, se puede usar:

```
sudo ./target/debug/rastreador /bin/ls -l /usr
```

Este comando ejecuta el programa `/bin/ls` con argumento `-l /usr` y al finalizar muestra la tabla acumulativa de system calls utilizadas por el programa. En este modo no se imprime cada syscall durante la ejecución, solo el conteo final.

Para ejecutar en **modo verbose**, donde se imprime información de cada syscall que realiza el programa, se usa:

```
sudo ./target/debug/rastreador -v /bin/ls -l /usr
```

En este modo se muestran el nombre de la syscall y su número, permitiendo observar la secuencia completa de llamadas en tiempo real. Es útil para depuración o análisis detallado del comportamiento del programa.

Para ejecutar en **modo paso a paso**, donde además de mostrar cada syscall se pausa la ejecución hasta que el usuario presione Enter, se usa:

```
sudo ./target/debug/rastreador -V /bin/ls -l /usr
```

Este modo permite inspeccionar detalladamente cada llamada al sistema y entender el flujo exacto de la ejecución del programa.

Como ejemplo práctico con un programa más complejo que realiza muchas llamadas al sistema, se puede rastrear la ejecución de `tar` para comprimir todo el contenido de `/usr/bin`:

```
sudo ./target/debug/rastreador -v /bin/tar -czf /tmp/test.tar.gz -C /usr/bin .
```

Este comando ejecuta `/bin/tar` para generar el archivo `/tmp/test.tar.gz` con todo el contenido de `/usr/bin` y rastrea todas las syscalls utilizadas durante la operación. Permite practicar el rastreo con programas más complejos y verificar cómo interactúan con el sistema operativo.

> **Nota importante:** Se recomienda usar `sudo` para rastrear programas que requieran permisos elevados. Las opciones del programa rastreado (`Prog`) no son analizadas por el rastreador; simplemente se pasan tal cual al ejecutar.

5. Actividades realizadas por estudiante

- 12/10/2025: Configuración del entorno y herramientas, 2 horas.
- 13/10/2025: Implementación del fork y ejecución del programa hijo, 3 horas.
- 14/10/2025: Implementación de rastreo de syscalls con ptrace, 4 horas.
- 15/10/2025: Implementación de modos verbose y step, 3 horas.
- 16/10/2025: Generación de tabla acumulativa con prettytable, 2 horas.
- 17/10/2025: Pruebas con diferentes programas y depuración de errores, 3 horas.
- 18/10/2025: Documentación del código y preparación de entrega, 2 horas.

Total horas: 19 horas.

6. Autoevaluación

El programa quedó funcional cumpliendo todos los requisitos principales: ejecución de cualquier programa hijo, rastreo de syscalls, modos `-v` y `-V`, y generación de tabla final. Problemas encontrados: manejo de señales y errores en ptrace, solucionados con chequeos adicionales. Limitaciones: solo syscalls comunes se muestran con nombre; las no incluidas aparecen como `sys_{num}`.

Reporte de commits de git:

- `commit 3411be3` Initial commit
- `commit 250dd77` This repo was created late the code was previously developed during weekend
- `commit f98196e` Readme file
- `commit 335fbc3` git ignore agregado

Calificación autoevaluada según rúbrica: 5/5 (cumple todas las secciones requeridas).

7. Lecciones Aprendidas

- Comprender cómo funciona `ptrace` y el rastreo de syscalls en Linux.
- Manejo de señales y sincronización entre procesos padre/hijo.
- Conversión de argumentos Rust a C para `execv`.
- Uso de tablas para mostrar información acumulativa de manera clara.
- Importancia de documentar cada parte del código y pruebas exhaustivas.

8. Bibliografía

- Manual de `ptrace` en Linux: https://man7.org/linux/man-pages/man2/ptrace.2.html
- Documentación Rust: https://doc.rust-lang.org/
- Nix crate documentation: https://docs.rs/nix/latest/nix/
- Clap crate documentation: https://docs.rs/clap/latest/clap/
- Prettytable crate documentation: https://docs.rs/prettytable/latest/prettytable/

- Linux System Calls: https://syscalls.w3challs.com/

9. Documentación del código fuente

El código está documentado internamente mediante comentarios explicativos en `tracer.rs`, `parser.rs` y `main.rs`, describiendo funciones, estructuras, y el flujo de ejecución paso a paso. Los modos verbose y step cuentan con instrucciones de uso y macros de debug para facilitar la comprensión del flujo de syscalls y el comportamiento del programa.