Informe: Rastreador de Syscalls en Rust

Introducción

En las lecciones en clase hemos introducido conceptos clave como son las syscalls, una llamada al sistema que permite a una aplicación de usuario solicitar acciones que requieren privilegios especiales. Lo hemos visto mucho en teoría, pero es necesario comprenderlo en la práctica.

La presente tarea aborda la necesidad de realmente comprender el comportamiento de estos programas a nivel de sistema operativo, específicamente a través del monitoreo de estas llamadas durante la ejecución de un programa al que vamos a llamar "Prog".

La herramienta está desarrollada en **Rust** y permite rastrear en tiempo real todas las interacciones entre un programa y el kernel de Linux.
Se puede ejecutar con la línea:

```
rastreador r \[-v | -V ] Prog \[args...]
```

- **-v** → despliega todas las syscall en un instante.
- **-V** → espera el *prompt* del usuario para continuar con la ejecución.
- **args** → se refiere a los argumentos necesarios para que el programa `Prog` funcione.

Con esto, el programa irá proporcionando información detallada sobre cada llamada al sistema, sus argumentos y valores de retorno. Esta capacidad es fundamental para la comprensión de cómo los programas interactúan con el sistema operativo.

Ambiente de desarrollo

- **Sistema Operativo:** Linux (Ubuntu)
- **Arquitectura:** x86
- **Lenguaje de programación:** Rust (versión estable 1.89.0)
- **Editor/IDE:** RustRover

Bibliotecas principales

- `nix` : Para interfaces con el sistema Unix
- `clap` : Para parsing de argumentos de línea de comandos

- `owo-colors` : Para output coloreado en terminal
- `serde_json` : Para procesamiento de datos JSON

Herramientas

- Cargo (gestor de paquetes de Rust)
- Git (control de versiones)

Estructuras de datos y funciones

Estructuras de datos utilizadas:

- `HashMap<u64, String>` : Para mapear números de system calls a sus nombres.
- `Pid` : Tipo de dato para identificadores de proceso de la biblioteca `nix` .
- `Command` : Para ejecución y control de procesos hijos.
- `Json` : Contiene el id de las syscall, nombre, parámetros y archivo (referencia al kernel de Linux).

Funciones principales:

- `wait_for_key()` : Pausa la ejecución hasta que el usuario presione una tecla.
- `main()` : Función principal que coordina todo el proceso de *tracing*.
- Parsing de argumentos con `clap::Parser` .
- Carga de tabla de system calls desde archivo JSON.
- Configuración de proceso hijo con `ptrace::traceme()` .
- Bucle principal de *tracing* con `ptrace::syscall()` y `waitpid()` .

Ejecución del programa

Compilación

```
cargo build --release
```

Ejemplos de ejecución

Programa sencillo en c++:

```
g++ -o ejecutable prueba_elevador.cpp
./target/release/rastreador -v ./ejecutable
./target/release/rastreador -V ./ejecutable
```

Crear archivo:

```
sudo ./target/release/rastreador -v /usr/bin/touch /filetest.txt
sudo ./target/release/rastreador -V /usr/bin/touch /filetestV.txt
```

Borrar archivo:

```
sudo ./target/release/rastreador -v /bin/rm /filetest.txt
sudo ./target/release/rastreador -V /bin/rm /filetestV.txt
```

Registro de Actividades

Estudiante	Actividad	Horas	Fecha
Josh Lis	Investigación de ptrace y system calls en Linux	5	27/08/2025
Josh Lis	Familiarización con Rust	2	28/08/2025
Josh Lis	Inicio del ambiente para el proyecto y pruebas	2	29/08/2025
Josh Lis	Desarrollo inicial del programa	3	29/08/2025
Josh Lis	Desarrollo del programa	2	30/08/2025
Josh Lis	Pruebas y depuración del programa	2	30/08/2025
Josh Lis	Documentación del proyecto	3	30/08/2025
Total		**19**	

Autoevaluación

Estado final del programa:

- El programa funciona correctamente, cumpliendo con todos los requisitos especificados.
- Rastrea system calls de programas arbitrarios.
- Soporta dos modos de operación: -v (sin pausas) y -V (con pausa).
- Muestra información detallada de cada system call, así como cuantos llamó.
- Usa colores para mejorar la legibilidad del output.

Problemas encontrados:

- Conflicto entre el flag `-V` personalizado y el flag de versión auto-generado por `clap`.
- Dificultades con el manejo de memoria en Rust.
- Problemas con rutas de archivos JSON en tiempo de compilación.
- Problemas con sintaxis de Rust.

Limitaciones:

- Solo funciona en sistemas Linux.
- Requiere permisos de superusuario para rastrear algunos programas.
- El output muestra argumentos en hexadecimal sin decodificación.

Lecciones aprendidas:

- Rust tiene una sintaxis distinta a las que he visto anteriormente, pero ofrece grandes beneficios en seguridad.
- `ptrace` es una herramienta increíble para análisis de programas a nivel de sistema operativo.
- La gestión de dependencias en Cargo es más sencilla que en muchos otros lenguajes.
- El versionado semántico es importante para evitar conflictos entre bibliotecas.
- La documentación sigue siendo igual de fundamental.
- Los registros específicos que poseen las llamadas al sistema.
- Un mejor entendimiento del funcionamiento de los procesos padres e hijos.
- Una comprensión más extensa de las llamadas al sistema.

Recomendaciones para futuros estudiantes

- Familiarizarse con los conceptos de sistemas operativos antes de abordar el proyecto.
- Comenzar con ejemplos simples de `ptrace` antes de implementar la solución completa.
- Leer la documentación oficial de Rust, las bibliotecas utilizadas y familiarizarse con la sintaxis.
- Probar el programa con diferentes tipos de aplicaciones para entender los patrones de system calls.
- Preferiblemente haber programado en ensamblador antes.

Bibliografía

- "The Rust programming language - The Rust programming language." https://doc.rust-lang.org/book/

- “nix - Rust.” https://docs.rs/nix/
- “clap - Rust.” https://docs.rs/clap/
- “ptrace(2) - Linux manual page.” https://man7.org/linux/man-pages/man2/ptrace.2.html
- Fasterthanlime, “GitHub - fasterthanlime/rue: A bad version of strace in Rust,” GitHub. https://github.com/fasterthanlime/rue/