Universidad Nacional Autónoma de México

Facultad de Ingeniería

Sistemas Operativos

Exposición: Análisis de la integración de Rust en el Kernel de Linux

Medina Villa Samuel

Ávila Martínez Alonso

Fecha de entrega: 23 de septiembre de 2025

Profesor: Gunnar Eyal Wolf Iszaevich

Análisis de la integración de Rust en el Kernel de Linux

El lenguaje C ha estado ligado al desarrollo del kernel de Linux desde su creación en 1991, ya que es un lenguaje eficiente y que otorga un control a bajo nivel del sistema. Sin embargo, la gestión manual de memoria en C es una fuente recurrente de vulnerabilidades de seguridad.
Es por esto que emerge Rust, un lenguaje de programación de sistemas que promete seguridad de memoria sin costo en el rendimiento. La incorporación de Rust al kernel de Linux a partir de la versión 6.1 marca un momento importante en la historia del lenguaje y del sistema operativo. Analizaremos esta transición, las razones principales de su impulso y las implicaciones que tiene en el futuro del kernel.

El dominio de C en el entorno del kernel.

La elección más popular en el desarrollo de sistemas y software de bajo nivel en las últimas décadas ha sido C, gracias al control que proporciona sobre la disposición de memoria y el hardware en general. Pero primero explicaremos que es la memoria y como funciona en C.

En C existen dos principales modelos para poder asignar memoria, una es la estática (pila) y dinámica (Heap). En la asignación estática es automática, en esta la memoria para las variables locales se crea al entrar a una función y digamos se destruye al salir. La principal limitación que tiene este tipo de asignación es que el tamaño de la memoria debe ser conocido en tiempo de compilación y no puede alterarse.
Para la asignación dinámica se le permite al programador el control para gestionar los recursos de memoria durante la ejecución de un programa. Esta memoria se asigna en el heap, la diferencia que podemos notar es la flexibilidad.
Ahora, la pregunta es ¿cómo se realiza esa asignación? En C se realiza a través de cuatro funciones clave que nos proporciona <stdlib.h>.

- malloc (). El propósito es asignar un bloque único y contiguo de memoria del tamaño especificado en bytes, además no se inicializa lo que significa que contiene valores basura.
  Un ejemplo es si queremos crear un arreglo de 10 enteros y dado que el tamaño de uno es de 4 bytes lo que se necesita es 40 bytes de memoria

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *ptr = (int *)malloc(40);

    for (int i = 0; i < 10; i++)
        ptr[i] = i + 1;

    for (int i = 0; i < 10; i++)
        printf("%d ", ptr[i]);
    return 0;
}
```

- calloc (). El propósito de esta función es similar a la anterior, su principal diferencia es que inicializa todos los bytes del bloque de memoria a cero. La sintaxis es:

```c
calloc(n, size);
```

  Donde n es el número de elementos y size es el tamaño de cada elemento en bytes.

- free (). Esta función devuelve la memoria asignada dinámicamente al sistema operativo. Es vital usarla ya que libera la memoria que ya no se necesita al no utilizarla conlleva a fugas de memoria (principal error).
  La sintaxis es la siguiente:

```c
free(ptr);
```

- realloc (). Nos permite expandir o reducir un bloque de memoria que fue asignado. Esta es la sintaxis de la función.

```c
realloc(ptr, new_size);
```

  Un ejemplo sería si asignamos previamente un valor de 3 enteros y queremos redimensionarlo a 10 se vera asi:

```c
#include <stdio.h>
#include <stdlib.h>
```

```c
int main() {
    int *ptr = (int *)malloc(3 * sizeof(int));

    int *temp = (int *)realloc(ptr, 10 * sizeof(int));

    if (temp == NULL)
        printf("Memory Reallocation Failed\n");
    else
        ptr = temp;

    return 0;
}
```

En este caso podemos ver los riesgos que se pueden presentar al asignar memoria, entonces ¿por qué un sistema como un kernel utiliza un modelo que es propenso a errores?
Este control sobre la memoria es un requisito para realizar tareas de más bajo nivel. Es precisamente este poder para manipular direcciones de memoria directamente lo que permite implementar técnicas fundamentales como:

- Mapeo de registros de hardware: Los **structs** en C permiten definir estructuras de datos cuya disposición en memoria puede ser controlada para que coincida exactamente con los registros de un dispositivo de hardware. Esto le permite al kernel interactuar directamente con el hardware a través de operaciones de punteros sobre estas estructuras, esta técnica es conocida como “entrada/salida mapeada en memoria” (MMIO).
- Gestión de memoria a bajo nivel: El kernel no solo es responsable de asignar espacios de memoria a los procesos, sino que configura directamente cómo es que el procesador traduce direcciones de memoria virtuales a direcciones físicas. Esto implica trabajar con registros especiales del procesador, manipular estructuras de datos internas, entre otros procesos.
  La MMU (Memory Management Unit) es un componente del procesador encargado de traducir las direcciones de memoria virtuales (las que utilizan los programas) en las direcciones de memoria físicas (las que están en la RAM), y las tablas de páginas son estructuras de datos que se encargan de almacenar ese mapeo entre página virtual y física.

Además de contar con la capacidad de implementar estas técnicas, el runtime de C es mínimo, lo cual es una cualidad importante para los manejadores de

interrupciones, que deben ejecutarse con la menor latencia posible y de forma predecible.

A pesar de todas las grandes ventajas que ofrece el lenguaje (de las cuales Rust también cumple casi en su totalidad), la seguridad del software escrito en C siempre ha requerido debida atención, pues el compilador no te protege contra **segfaults**, **use-after-free**, **double free**, tienes que manejar manualmente la alocación/de-alocación de memoria, y en general es propenso a riesgos de seguridad importantes.

Garantías de seguridad

La propuesta que le da valor a Rust es su capacidad para erradicar clases enteras de errores de memoria a través de un análisis estático, asegurando así la prevención de errores en tiempo de compilación, en lugar de ejecución.
Dentro de este análisis estático riguroso, el protagonista es el **borrow checker**, un conjunto de reglas y mecanismos del compilador de Rust que se encargan de verificar que los programas respeten las reglas de **propiedad** y **préstamos de memoria** (ownership & borrowing), garantizando que no se acceda a memoria inválida sin necesidad de un recolector de basura.

- **Ownership:** Este principio establece que los valores tienen un único dueño (la variable que lo posee), y cuando el dueño sale del scope, el valor se libera automáticamente.

- **Borrowing:** Un valor es intransferible de dueño, pero se pueden **prestar** referencias, éstas pueden ser mutables (modificar y leer) o inmutables (únicamente leer). El borrow checker se encarga de verificar que sólo exista una referencia mutable a la vez, y prohíbe el uso de cualquier tipo de referencia si el valor referenciado ya ha sido liberado.

```rust
fn main() {
    let mut s = String::from("Hola");

    let r1 = &mut s;
    let r2 = &mut s; // Error: dos préstamos mutables al mismo tiempo
    println!("{}, {}", r1, r2);
}
```

El compilador arrojaría un mensaje como:

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
```

Es importante mencionar que Rust **no erradica completamente** los errores de memoria, solo una clase de ellos, como:

- **Use-after-free:** Es el uso de un bloque de memoria después de haberla liberado. Si la memoria ya ha sido reasignada a otra cosa, se pueden modificar datos que no le pertenecen al proceso, y ocasiona comportamientos indefinidos.

```c
int *ptr = malloc(sizeof(int));
*ptr = 42;
free(ptr);

// ERROR: uso después de liberar
printf("Valor: %d\n", *ptr);
```

- **Double free:** Es cuando se intenta liberar dos veces el mismo bloque de memoria. La segunda vez que se intenta liberar un bloque de memoria, éste ya puede estar reasignado, lo que corrompe la estructura interna del heap (se intenta agregar dos veces a la lista de bloques de memoria disponibles, o freelist).

```c
int *ptr = malloc(sizeof(int));
free(ptr);

// ERROR: doble liberación
free(ptr);

return 0;
```

- **Dangling pointers:** Son punteros que apuntan a bloques de memoria que ya no son válidos. Si se utilizan para lectura, se obtienen datos basura, en cambio si se utilizan para escritura, se pueden corromper datos ajenos o en otras partes del programa.

```c
int *ptr = malloc(sizeof(int));
*ptr = 99;
free(ptr);
```

```c
// Ahora ptr es "dangling", pero todavía lo usamos.
if (ptr != NULL) {
    printf("Dangling pointer: %d\n", *ptr); // ERROR
}
```

- **Buffer overflows:** Es la escritura de datos fuera de los límites de un arreglo o bloque de memoria.

```c
char buffer[5] = "abcd";

// ERROR: escribimos fuera de los 5 bytes
buffer[5] = 'X';
printf("Buffer: %s\n", buffer);
```

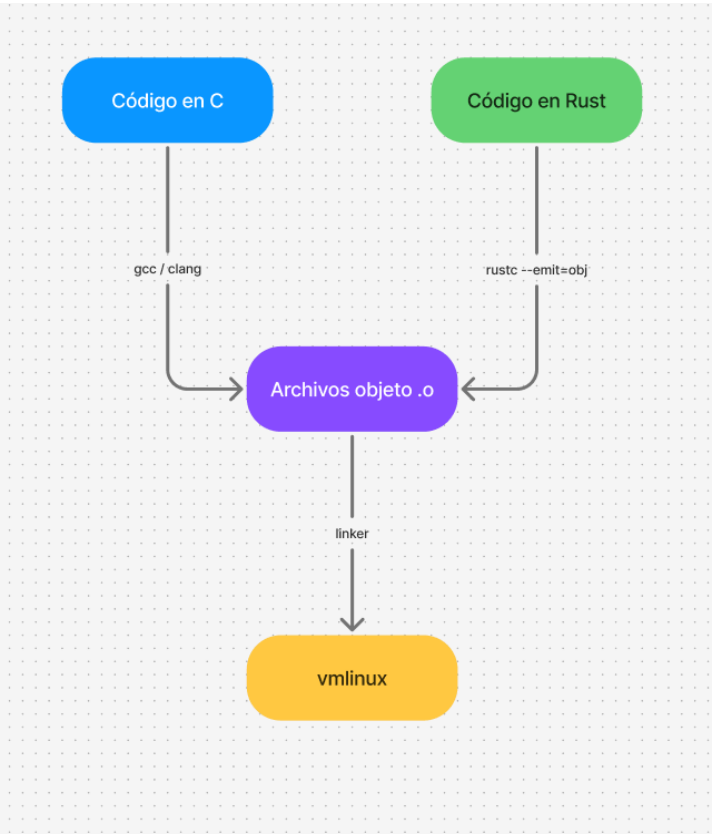
- **Data races:** Las condiciones de carrera en los sistemas concurrentes son errores donde el resultado de una ejecución depende del orden impredecible en el que se intercalan las operaciones de múltiples hilos.

La mayoría de las vulnerabilidades del kernel de Linux son producto de errores de memoria, y aunque cuenta con herramientas como el Kernel Address Sanitizer (KASan), cuyo propósito es detectar estos errores en tiempo de ejecución, la seguridad debe ser proactiva, no reactiva. Ejemplos históricos de la importancia de este último punto son los exploits famosos **Dirty COW**, en el que se aprovecha una condición de carrera para escalar privilegios, **Stack clash**, que aprovechaba desbordamientos de pila para sobrescribir memoria adyacente, y múltiples **use-after-free** en drivers de **red**, **archivos** y **subsistemas**.

Coexistencia entre Rust y C

Entonces, si Rust ofrece tantas ventajas, ¿Por qué se sigue usando C?, si bien las capacidades a bajo nivel de Rust y C son bastante similares, partes críticas del kernel requieren control absoluto, como la inicialización del hardware, la administración de interrupciones, el MMU, además de la optimización extrema en tamaño de binarios y ciclos del CPU, aspectos en los que C destaca.
Es por esto que la relación entre Rust y C en el kernel de Linux no es de rivalidad, sino que se complementan entre sí en las áreas que destacan con el propósito de obtener un kernel más robusto.

El proceso de compilado del kernel, visto desde una perspectiva muy general, es el siguiente:



Tanto el código de C como de Rust son compilados a **archivos objeto** (cada uno con su propio compilador, claro) y posteriormente se pasa a la etapa del **linker** o **vinculador**, un programa que se encarga de procesar los archivos .o generados para obtener un único binario ejecutable. Este binario ejecutable llamado **vmlinux** contiene el kernel completo de Linux, el cual pasa por otro conjunto de procesos para hacerlo una imagen booteable y que funcione en las distintas arquitecturas de los procesadores soportados.

El futuro de Linux (Conclusiones)

La incorporación de Rust en el kernel de Linux marca el inicio de una nueva etapa en su desarrollo, en donde la seguridad y robustez son los puntos principales de enfoque. Parece ser que el kernel va encaminado a convertirse en un sistema híbrido, en donde C aporta madurez y cercanía al hardware, mientras que Rust deja al alcance un desarrollo más seguro y sostenible en ciertas áreas.
La tendencia que se está siguiendo en el desarrollo del kernel promete ser beneficiosa a largo plazo, y tal vez crítica para la continuidad del mismo.

Referencias

- Contreras, Felipe. (2025). *Rust is not for Linux*. https://felipec.wordpress.com/2025/02/13/rust-not-for-linux/

- Fireship. (2022). *Rust in 100 Seconds*. YouTube. https://www.youtube.com/watch?v=YyRVOGxRKLg

- Bean, Noah. (2024). *The Pragmatic Future of the Linux Kernel: Balancing Rust and C*. Medium. https://medium.com/@noahbean3396/the-pragmatic-future-of-the-linux-kernel-balancing-rust-and-c-e518117328b7

- Security Boulevard. (2025). *Rust-Linux War*. https://securityboulevard.com/2025/02/rust-linux-war-richixbw/

- Programiz. (s.f.). *C dynamic memory allocation*. https://www.programiz.com/c-programming/c-dynamic-memory-allocation
- Tanenbaum, A. S., & Bos, H. (2015). Modern operating systems (4th ed.). Pearson.
- Love, R. (2010). Linux kernel development (3rd ed.). Addison-Wesley Professional.
- Linux Kernel Organization. (2025). The Linux Kernel documentation. https://docs.kernel.org/
- MITRE. (2025). Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/