Análisis de la integración de Rust en el kernel de Linux

Medina Villa Samuel-320249538
Ávila Martínez Alonso-320237988

C

El lenguaje C ha estado ligado al desarrollo del kernel de Linux desde su creación en 1991

- 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

El dominio de C en el entorno del kernel.

¿Cómo funciona la memoria en C?

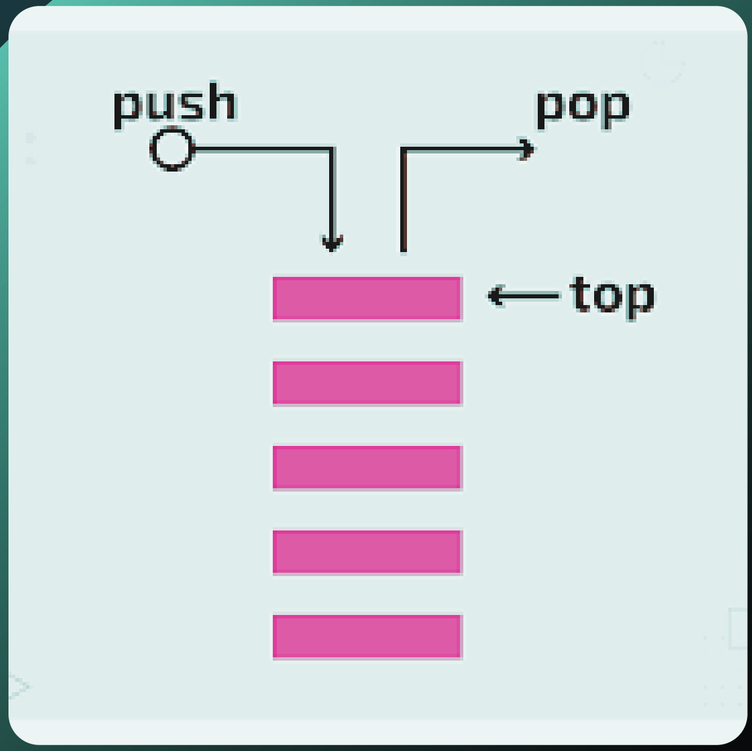
Existen dos principales modelos para poder asignar memoria:

- Estática (Pila)
- Dinámica (Heap)

Estática (Pila)

Es automática, en esta la memoria para las variables locales se crea al entrar a una función y digamos se destruye al salir.

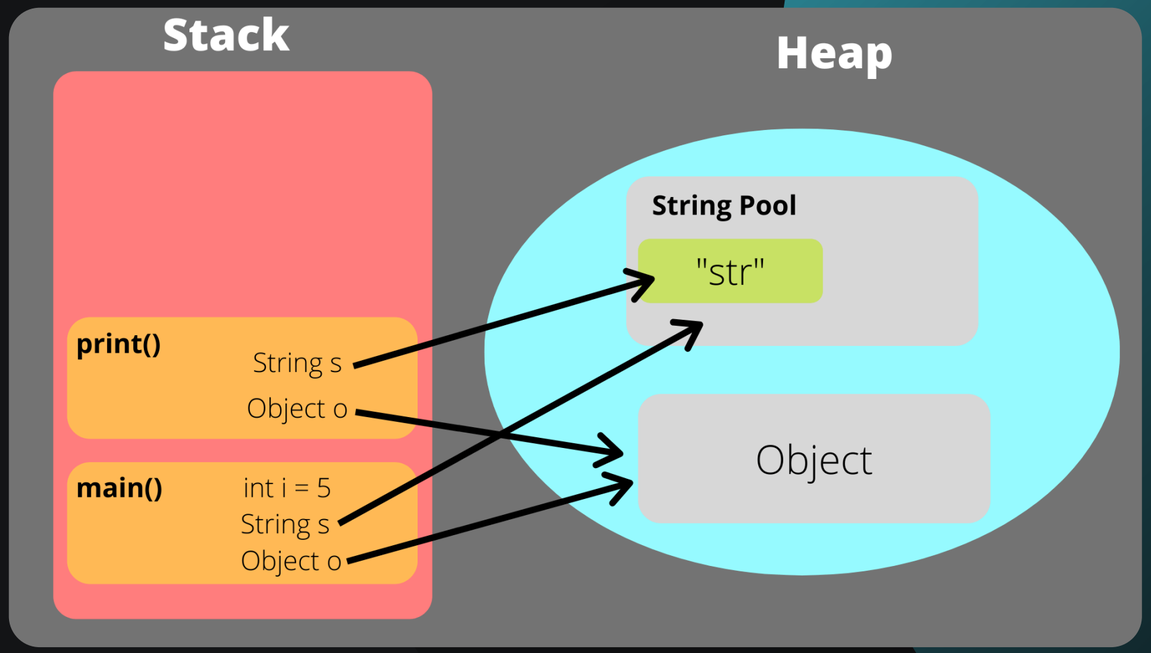
La limitación que tiene este tipo de asignación es que el tamaño de la memoria debe ser conocido en tiempo de compilación y no puede alterarse.

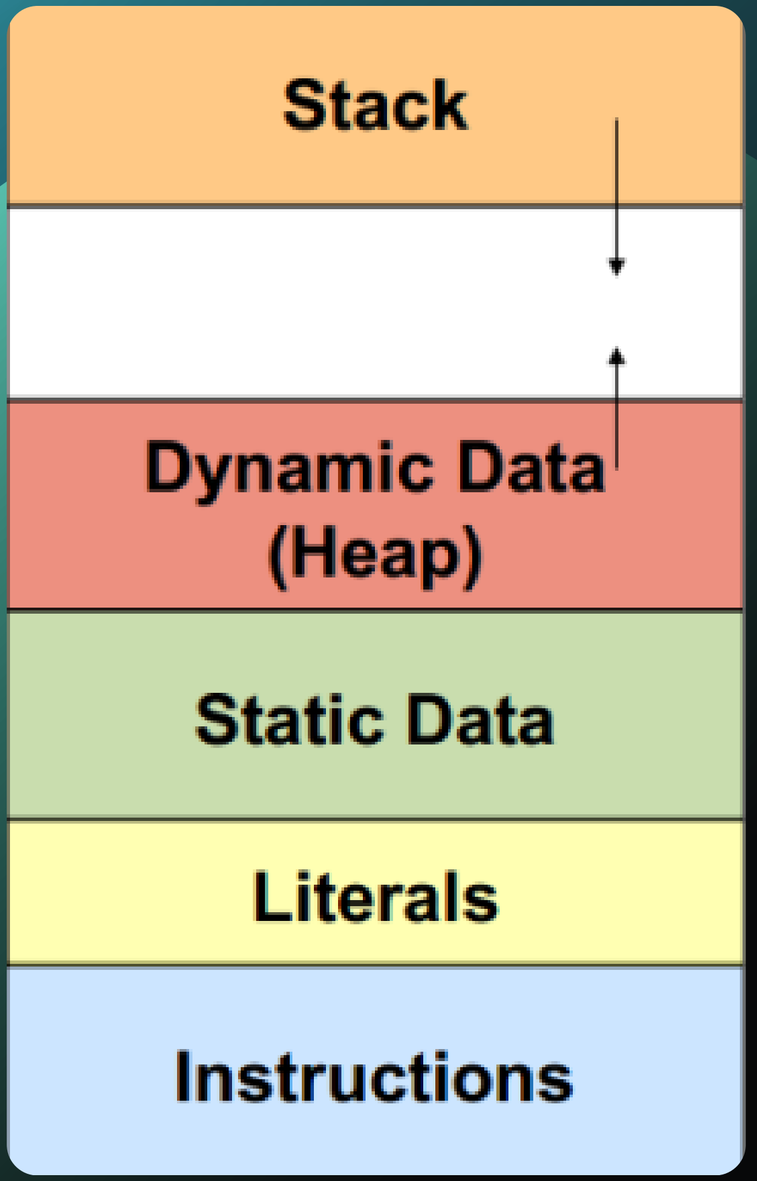


Dinámica (Heap)

Para la asignación dinámica se le permite al programador el control para gestionar los recursos de memoria durante la ejecución de un programa.

- La diferencia que podemos notar es la flexibilidad.





¿Cómo se realiza la asignacion de memoria en C?

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.

```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;
}
```

¿Cómo se realiza la asignacion de memoria en C?

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

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

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

¿Cómo se realiza la asignacion de memoria en C?

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).

Su sintaxis:

```
free(ptr);
```

realloc (). Nos permite expandir o reducir un bloque de memoria que fue asignado

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

¿Cómo se realiza la asignacion de memoria en C?

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

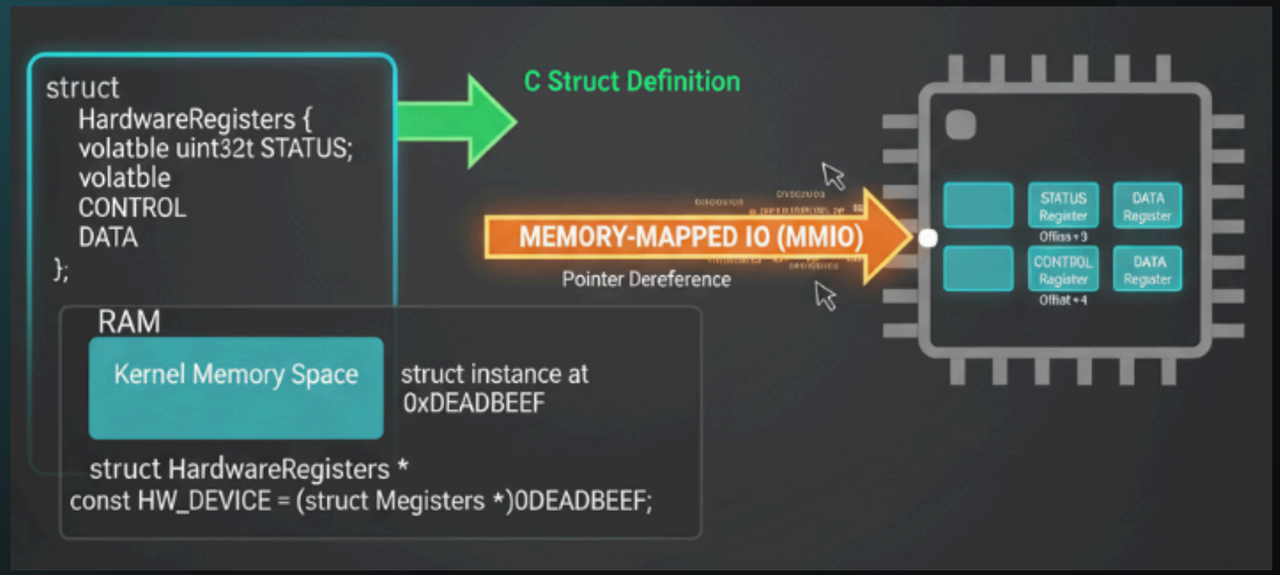
```c
#include <stdio.h>
#include <stdlib.h>

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?**

Técnicas fundamentales

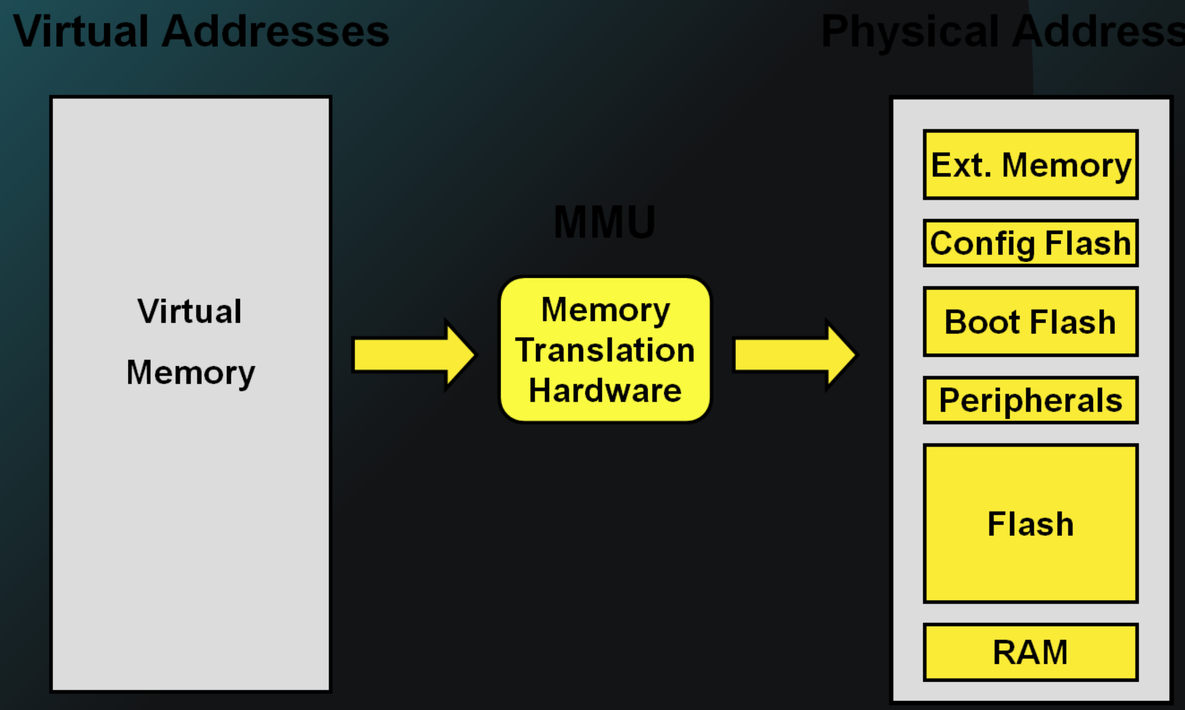
Este control sobre la memoria es un requisito para realizar tareas de más bajo nivel. Es precisamente este poder para manipular direcciones de memoria directamente lo que permite implementar técnicas fundamentales como:

- 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).



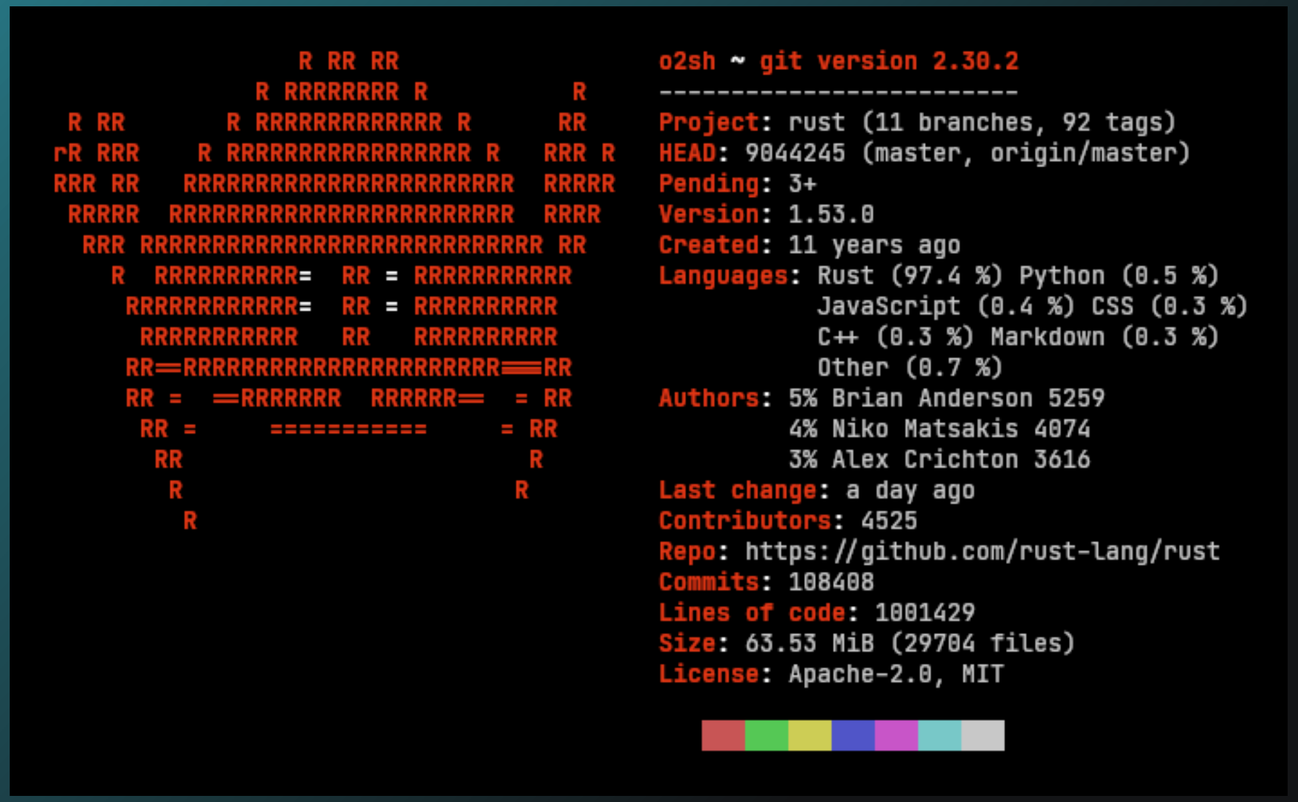
Técnicas fundamentales

- Gestión de memoria a bajo nivel:El kernel gestiona la memoria a bajo nivel, configurando al procesador para que traduzca las direcciones de memoria virtuales (de los programas) a físicas (en la RAM). Este proceso es realizado por la MMU (Unidad de Gestión de Memoria), un componente del procesador que utiliza tablas de páginas para saber cómo mapear o corresponder cada dirección virtual con su dirección física real.



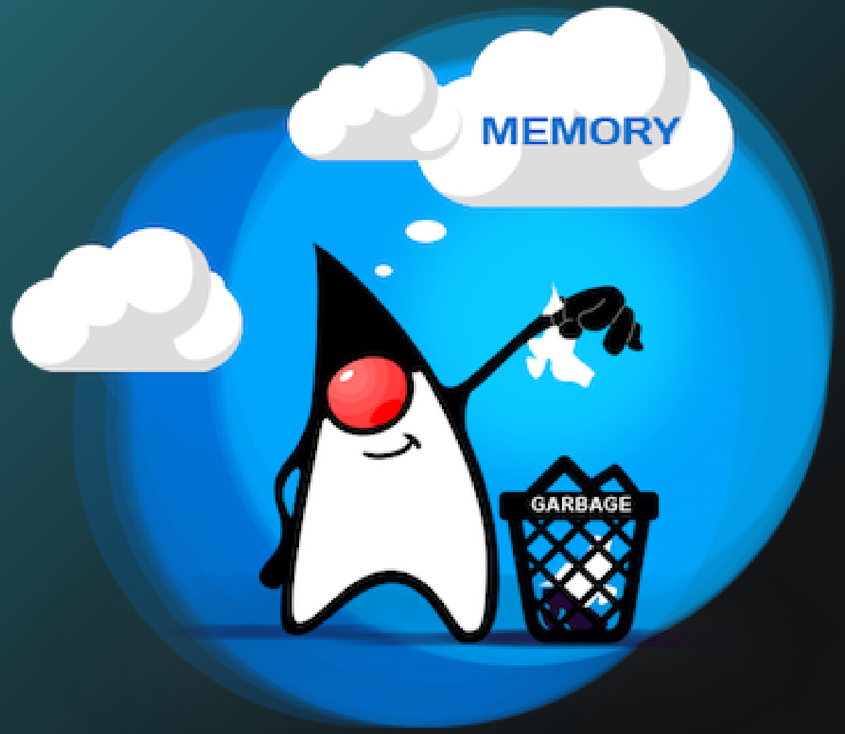
Garantías de seguridad

Prevención de errores de memoria en tiempo de compilación.



Borrow Checker

Conjunto de reglas y mecanismos del compilador, hacen cumplir las reglas de propiedad y préstamos de memoria (ownership & borrowing)



Ownership

Único dueño, se libera la memoria al salir del scope

```rust
fn main() {
    let s1 = String::from("hola");

    // Ownership se mueve de s1 a s2
    let s2 = s1;

    // println!("{}", s1); // ❌ ERROR: s1 ya no es válido
    println!("{}", s2); // ✅ "hola"
}
```

Borrowing

Puedes prestar referencias mutables (con restricciones) o inmutables

```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);
}
```

NO erradica todos los erroes de memoria, solo un grupo específico de ellos:

Use-after-free

Uso de un bloque de memoria después de haberla liberado.

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

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

Double free

Intento de liberar un mismo bloque de memoria dos veces.

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

// ERROR: doble liberación
free(ptr);
```

Buffer overflow

Escritura de datos fuera de los límites de un bloque de memoria.

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

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

Dangling pointers

Punteros que apuntan a bloques de memoria inválidos.

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

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

Antecedentes históricos

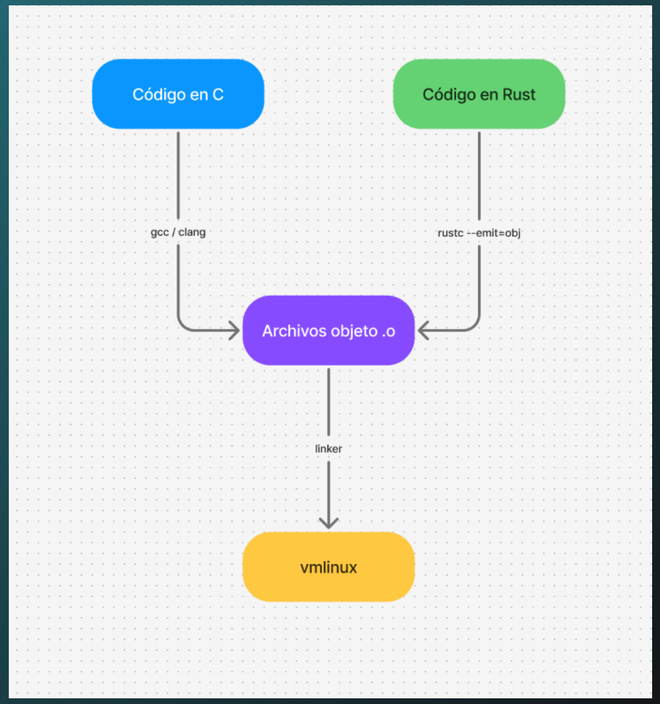
- Dirty COW
- Stack clash
- use-after-free en drivers de red, archivos y subsistemas

Coexistencia entre Rust y C

Seguimos usando C porque hay ciertas partes del kernel que requieren control absoluto, además de optimizaciones en los tamaños de binarios y ciclos del CPU.

Es una relación de coexistencia y en donde se complementan, no de rivalidad.

Compilación de ambos lenguajes en el kernel



Compilador => Código objeto => Vinculador => Binario ejecutable

El futuro de Linux

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.

La tendencia que se está siguiendo en el desarrollo del kernel promete ser beneficiosa a largo plazo, y tal vez crítica para la continuidad de Linux.