Rust and systems
Building an OS with rust

ESGI 4A

Maxime BOURY

2024-2025

Summary

I. Day1 – Memory allocator

II. Day 2 – Kernel

III. Day3 – No day 3 I think

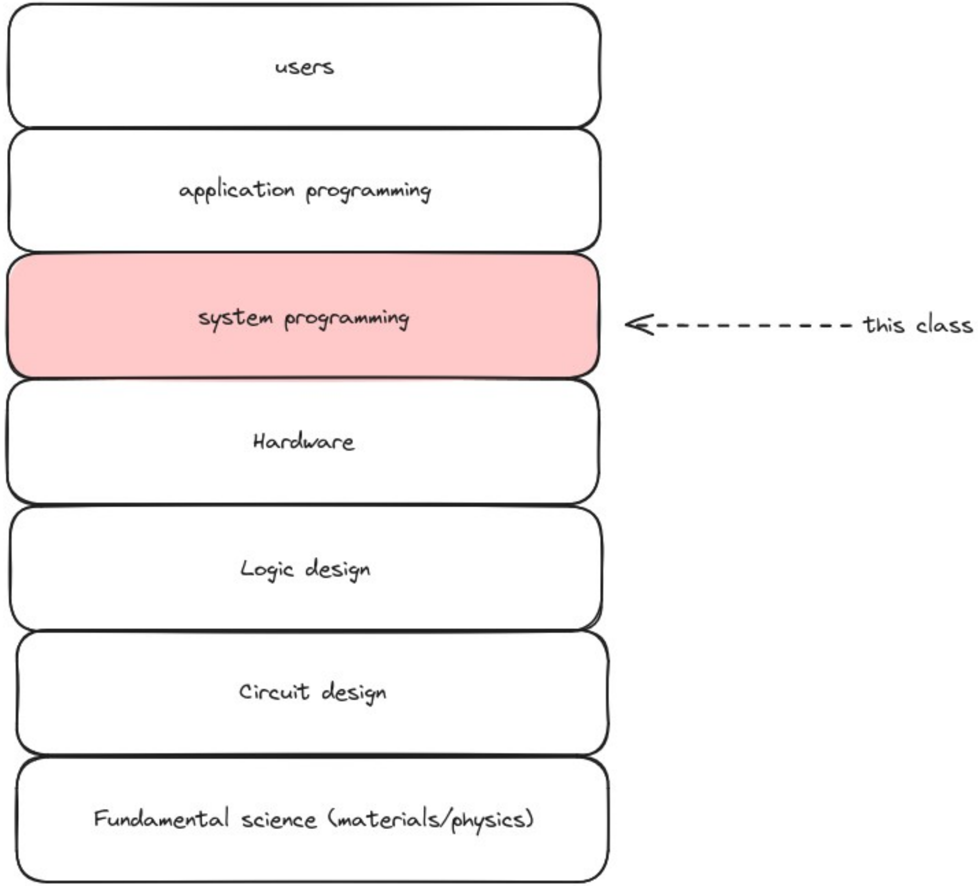
whoami

- Security engineer, Did vulnerability research and exploitation in embedded/Telecom/Blockchains’
- Now Freelance in formation/offensive tool development and audits
- Main focuses are telecom, embedded, Rust and C/C++
- Doing other security related things

1

Introduction

Introduction



inspired by Nostarch's The Secret Life of Programs

Introduction

- What is an operating system?

Introduction

- What is an operating system/kernel?
  - Windows, linux, BSD, Minix, SeL4, Android ….
  - Manages hardware
  - Isolates application and manages them
  - What belongs to the OS and what makes an OS ?

Source: https://en.wikipedia.org/wiki/Microkernel

Introduction

- What is an operating system/kernel?
  - Windows, linux, BSD, Minix, SeL4, Android ….
  - Manages hardware
  - Isolates application and manages them
  - What belongs to the OS and what makes an OS ?
    - Filesystem, process management, Memory management ….

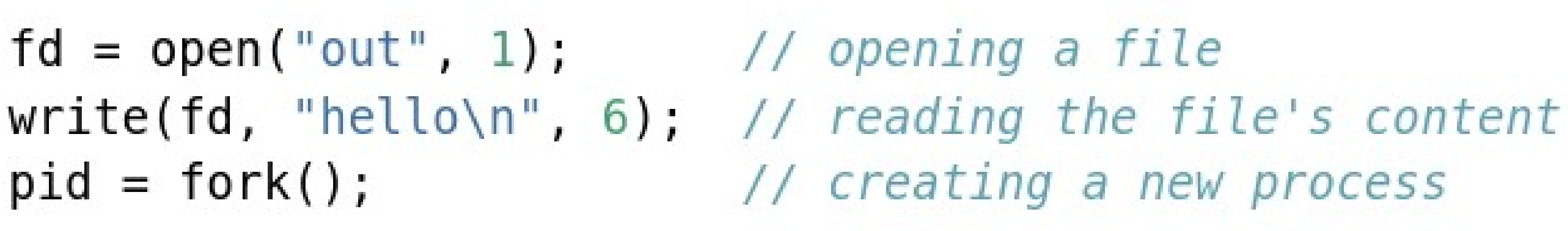
Source: https://en.wikipedia.org/wiki/Microkernel

Introduction

- Another View
  - The OS provides abstraction to applications thus manages core services
    - Users, IPC, memory ….

Example system calls

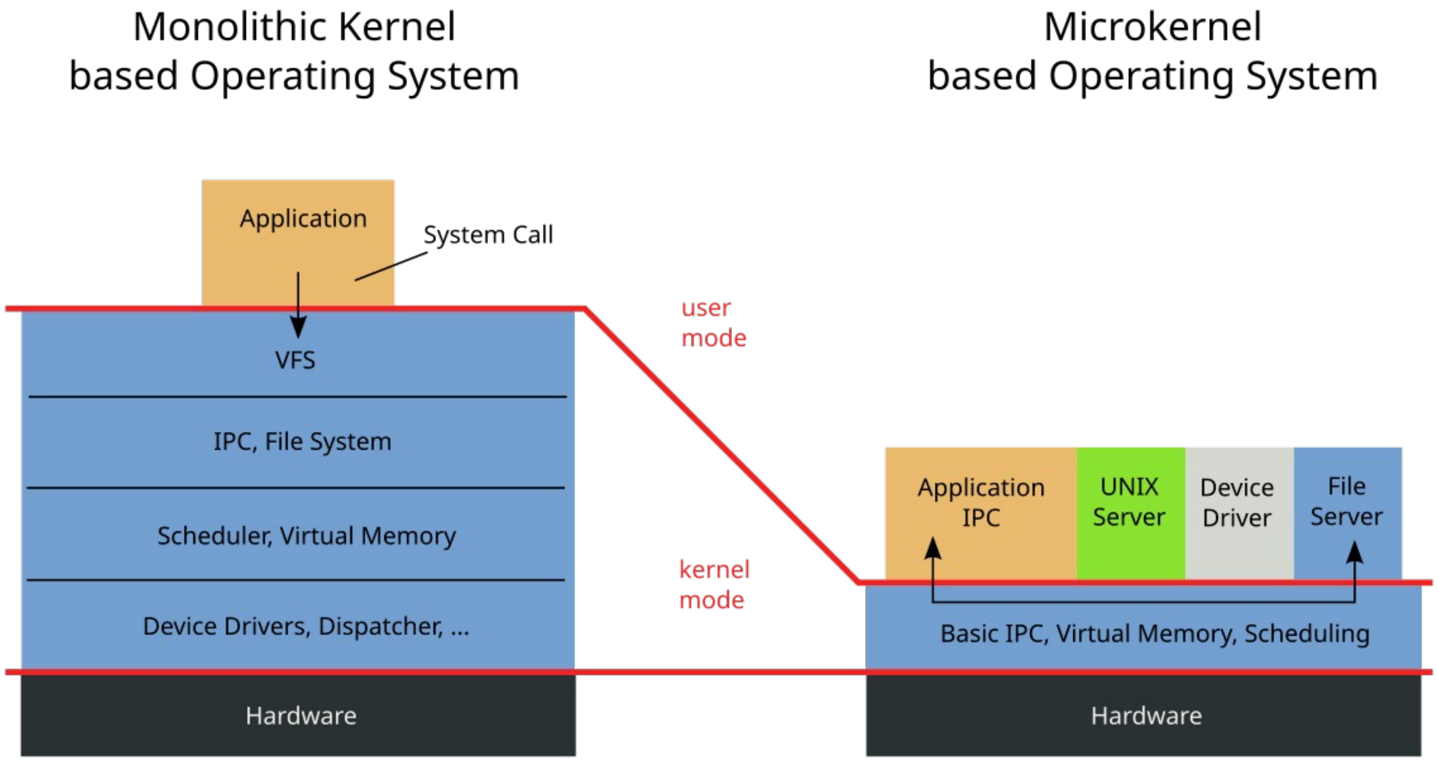
- Interface : application talks to an OS via system calls
- Abstraction : process and file descriptor



```
fd = open("out", 1);        // opening a file
write(fd, "hello\n", 6);    // reading the file's content
pid = fork();               // creating a new process
```

Source : https://tc.gts3.org/cs3210/2020/spring/l/lec01/lec01.html#example-system-calls

Introduction



Source: https://en.wikipedia.org/wiki/Microkernel

Why is designing OS challenging

- Conflicting design goals and trade-offs
  - Performant yet portable
  - Isolated yet sharable
  - Extensible yet secure
  - Compatible yet efficient
  - Many non solved problems: sandboxing, scalability ….

Source : https://tc.gts3.org/cs3210/2020/spring/l/lec01/lec01.html#who-should-take-cs3210

Why studying OS

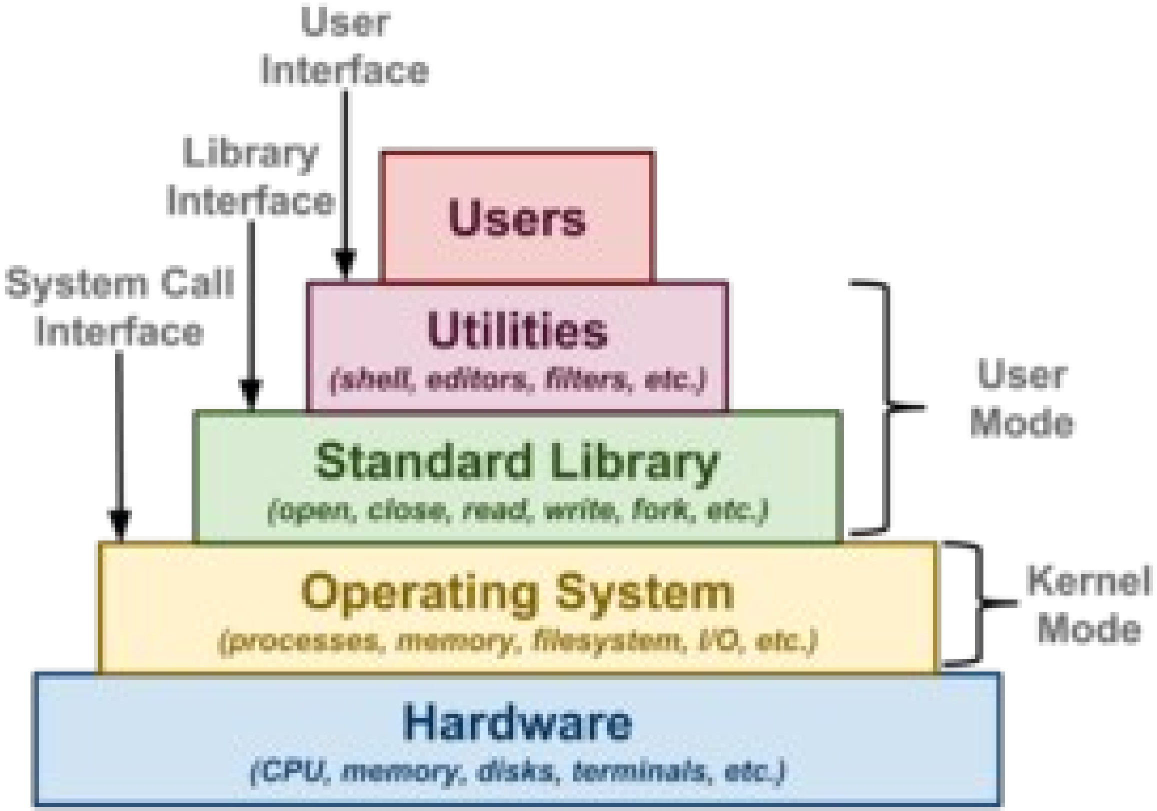
- Anyone wanting to work on previous problems
- Anyone caring about what’s going on under the hood
- Anyone wanting to build high performance systems (I.e cloud)
- Anyone wanting to build systems (I.e embedded/firmware)
- Anyone needing to diagnose bugs or security problems
- Anyone wanting better skill for low level problems (I.e reversing)

Source : https://tc.gts3.org/cs3210/2020/spring/l/lec01/lec01.html#who-should-take-cs3210

Why studying OS

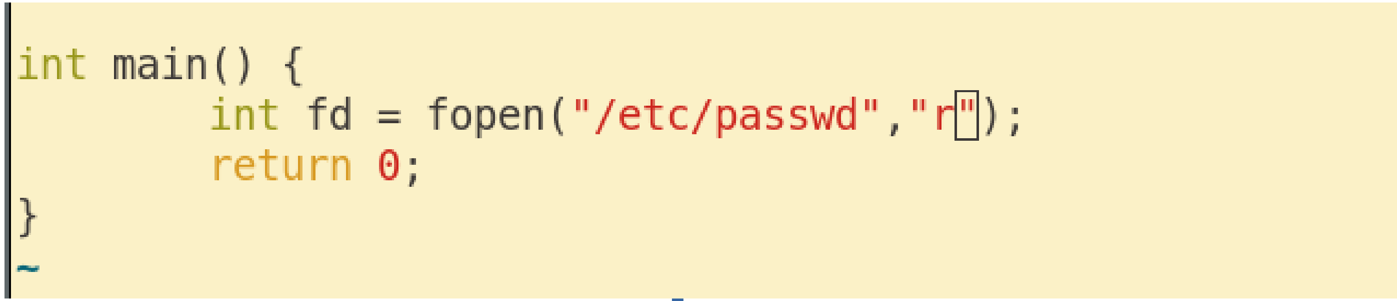
- Allowing you to write tooling, it’s important
- Infosec pro sucking at dev is a problem
- The world is gradually becoming software, you will one day be a software profesionnal (sdn/IAC ….)

Syscall

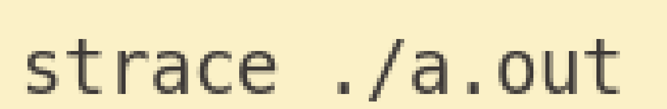

User Interface
Library Interface
System Call Interface
Users
Utilities
(shell, editors, filters, etc.)
Standard Library
(open, close, read, write, fork, etc.)
Operating System
(processes, memory, filesystem, I/O, etc.)
Hardware
(CPU, memory, disks, terminals, etc.)
User Mode
Kernel Mode

Syscall - strace

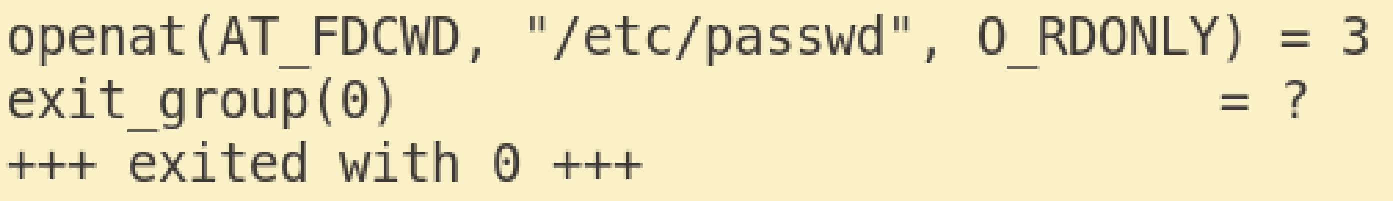
-



```
int main() {
        int fd = fopen("/etc/passwd","r");
        return 0;
}
~
```



```
strace ./a.out
```

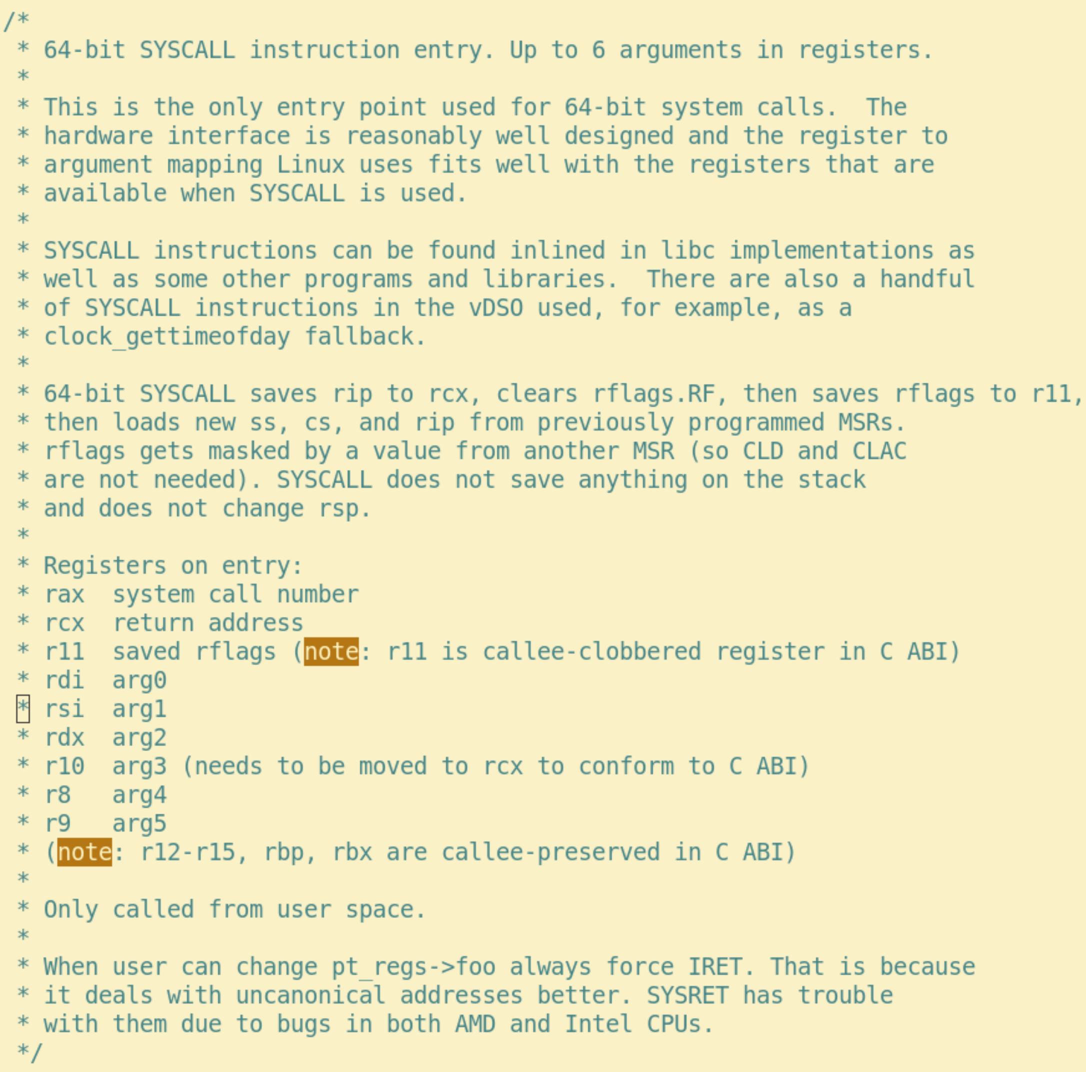


```
openat(AT_FDCWD, "/etc/passwd", O_RDONLY) = 3
exit_group(0)                            = ?
+++ exited with 0 +++
```

Syscall - walkthrough

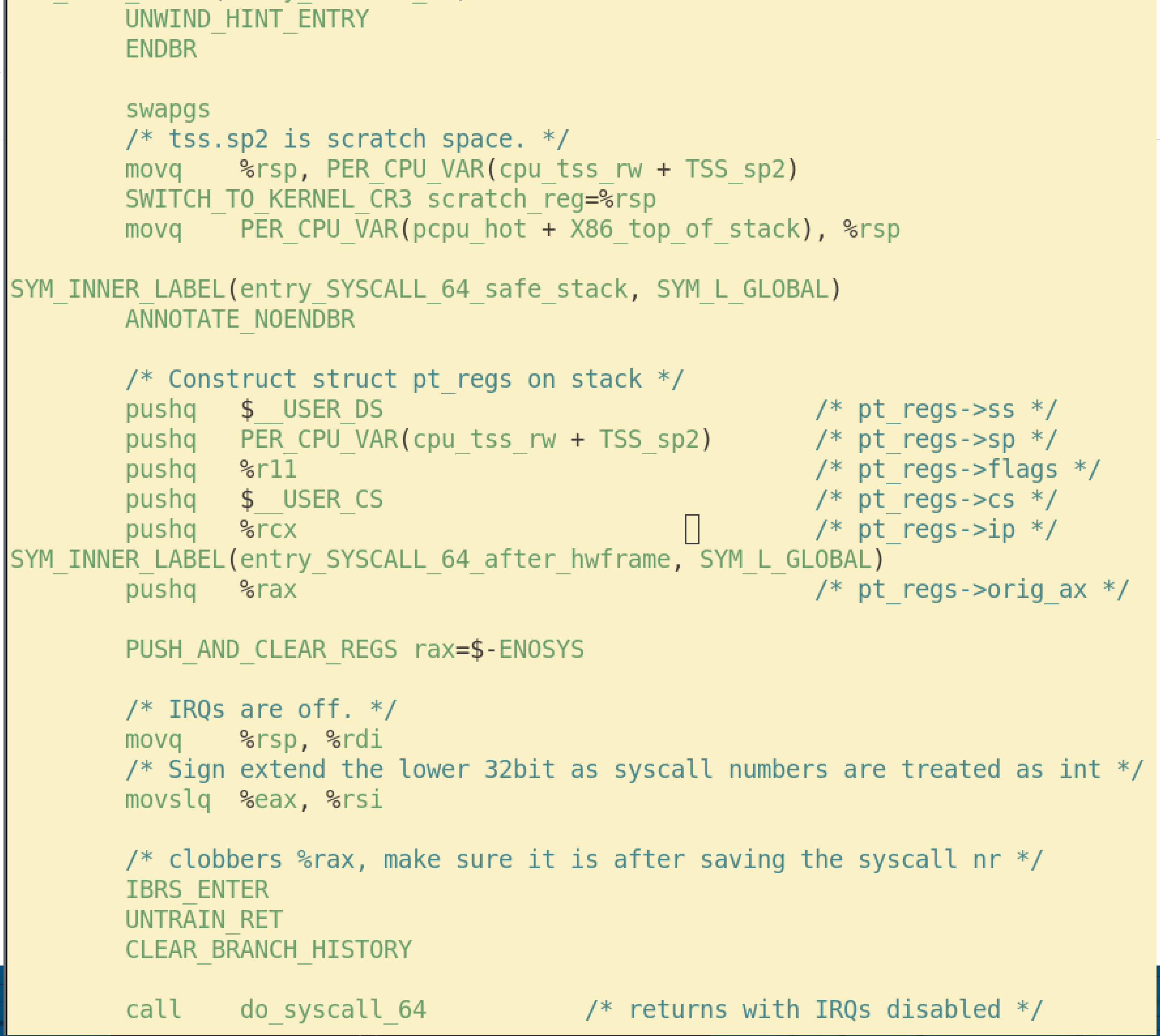
- The interruptions asks the CPU to stop
- Dispatched to kernel
- linux/arch/x86/entry/entry_64.s
- Table is linux/arch/x86/entry/syscalls/syscall_64.tbl

Syscall -



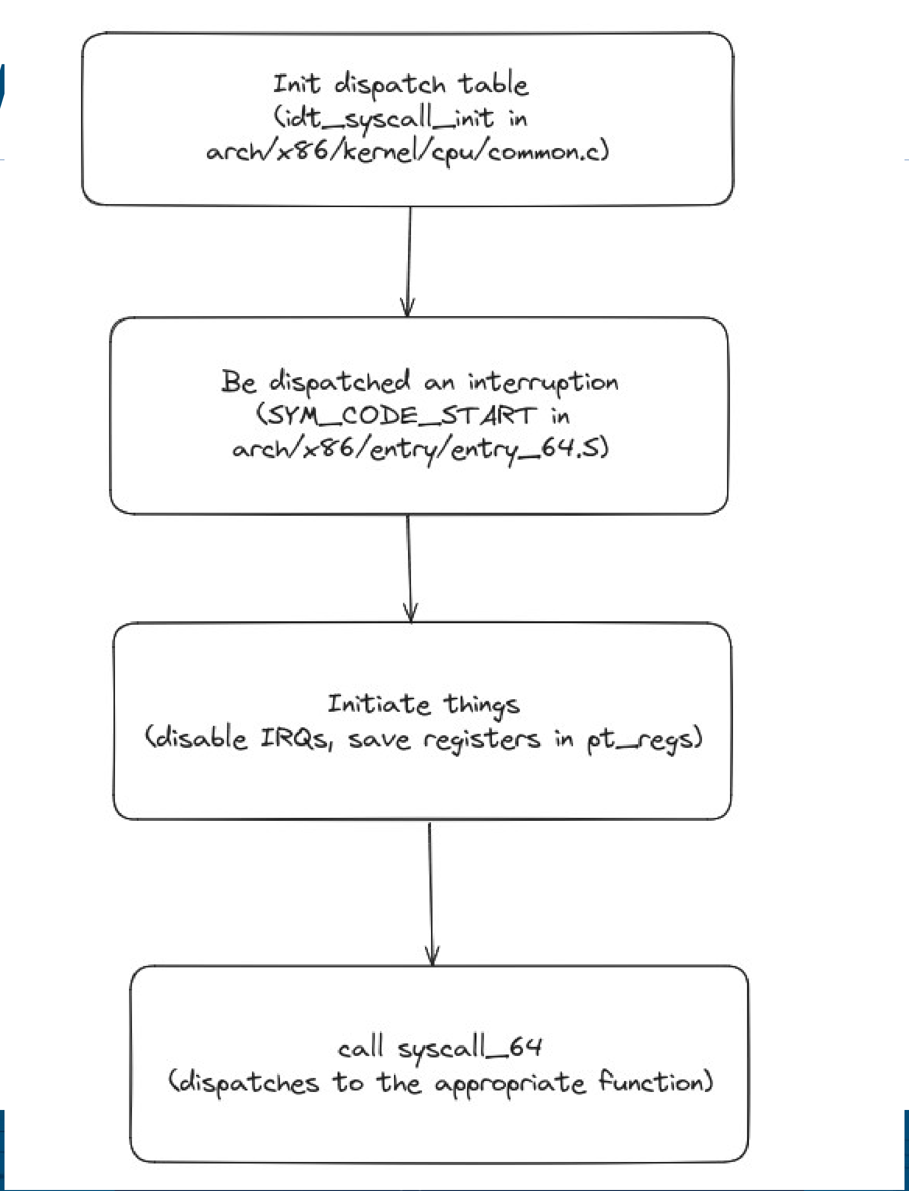
```
/*
 * 64-bit SYSCALL instruction entry. Up to 6 arguments in registers.
 *
 * This is the only entry point used for 64-bit system calls.  The
 * hardware interface is reasonably well designed and the register to
 * argument mapping Linux uses fits well with the registers that are
 * available when SYSCALL is used.
 *
 * SYSCALL instructions can be found inlined in libc implementations as
 * well as some other programs and libraries.  There are also a handful
 * of SYSCALL instructions in the vDSO used, for example, as a
 * clock_gettimeofday fallback.
 *
 * 64-bit SYSCALL saves rip to rcx, clears rflags.RF, then saves rflags to r11,
 * then loads new ss, cs, and rip from previously programmed MSRs.
 * rflags gets masked by a value from another MSR (so CLD and CLAC
 * are not needed). SYSCALL does not save anything on the stack
 * and does not change rsp.
 *
 * Registers on entry:
 * rax  system call number
 * rcx  return address
 * r11  saved rflags (note: r11 is callee-clobbered register in C ABI)
 * rdi  arg0
 * rsi  arg1
 * rdx  arg2
 * r10  arg3 (needs to be moved to rcx to conform to C ABI)
 * r8   arg4
 * r9   arg5
 * (note: r12-r15, rbp, rbx are callee-preserved in C ABI)
 *
 * Only called from user space.
 *
 * When user can change pt_regs->foo always force IRET. That is because
 * it deals with uncanonical addresses better. SYSRET has trouble
 * with them due to bugs in both AMD and Intel CPUs.
 */
```

Syscall - w



```
        UNWIND_HINT_ENTRY
        ENDBR

        swapgs
        /* tss.sp2 is scratch space. */
        movq    %rsp, PER_CPU_VAR(cpu_tss_rw + TSS_sp2)
        SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
        movq    PER_CPU_VAR(pcpu_hot + X86_top_of_stack), %rsp

SYM_INNER_LABEL(entry_SYSCALL_64_safe_stack, SYM_L_GLOBAL)
        ANNOTATE_NOENDBR

        /* Construct struct pt_regs on stack */
        pushq   $__USER_DS                              /* pt_regs->ss */
        pushq   PER_CPU_VAR(cpu_tss_rw + TSS_sp2)       /* pt_regs->sp */
        pushq   %r11                                    /* pt_regs->flags */
        pushq   $__USER_CS                              /* pt_regs->cs */
        pushq   %rcx                                    /* pt_regs->ip */
SYM_INNER_LABEL(entry_SYSCALL_64_after_hwframe, SYM_L_GLOBAL)
        pushq   %rax                                    /* pt_regs->orig_ax */

        PUSH_AND_CLEAR_REGS rax=$-ENOSYS

        /* IRQs are off. */
        movq    %rsp, %rdi
        /* Sign extend the lower 32bit as syscall numbers are treated as int */
        movslq  %eax, %rsi

        /* clobbers %rax, make sure it is after saving the syscall nr */
        IBRS_ENTER
        UNTRAIN_RET
        CLEAR_BRANCH_HISTORY

        call    do_syscall_64           /* returns with IRQs disabled */
```

Syscall - W


Init dispatch table
(idt_syscall_init in
arch/x86/kernel/cpu/common.c)
Be dispatched an interruption
(SYM_CODE_START in
arch/x86/entry/entry_64.S)
Initiate things
(disable IRQs, save registers in pt_regs)
call syscall_64
(dispatches to the appropriate function)

Syscall - analysis

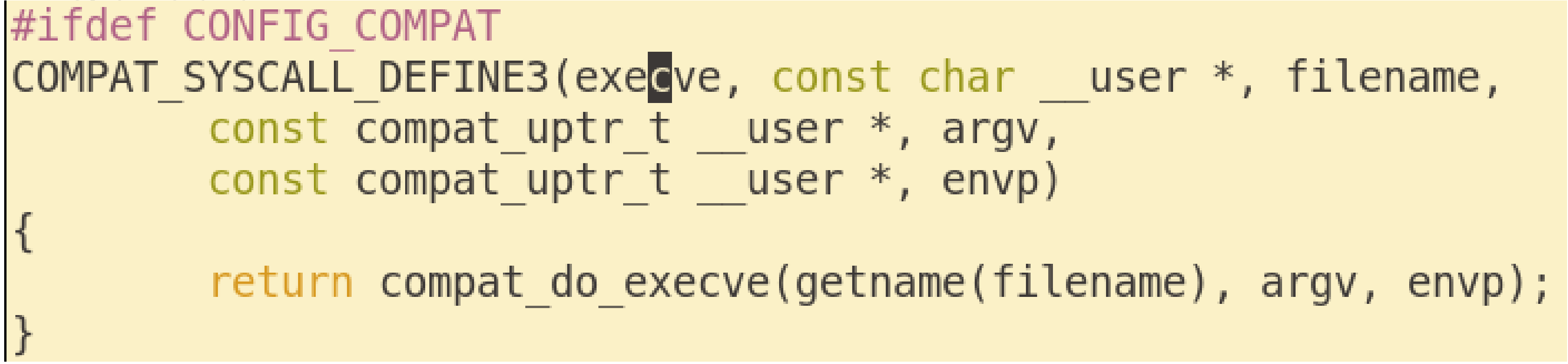
- Execve syscall

Syscall - Execve

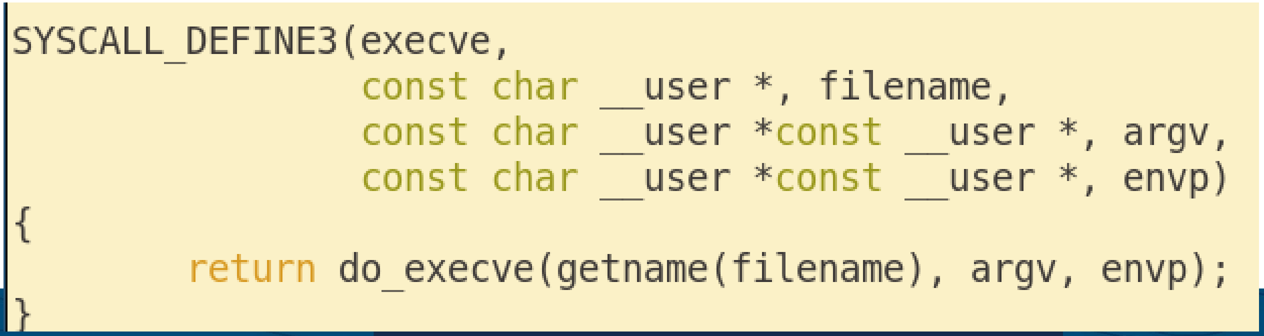
- First find the definition
  - fs/exec.c

Execve - definition

- First find the definition
  - fs/exec.c

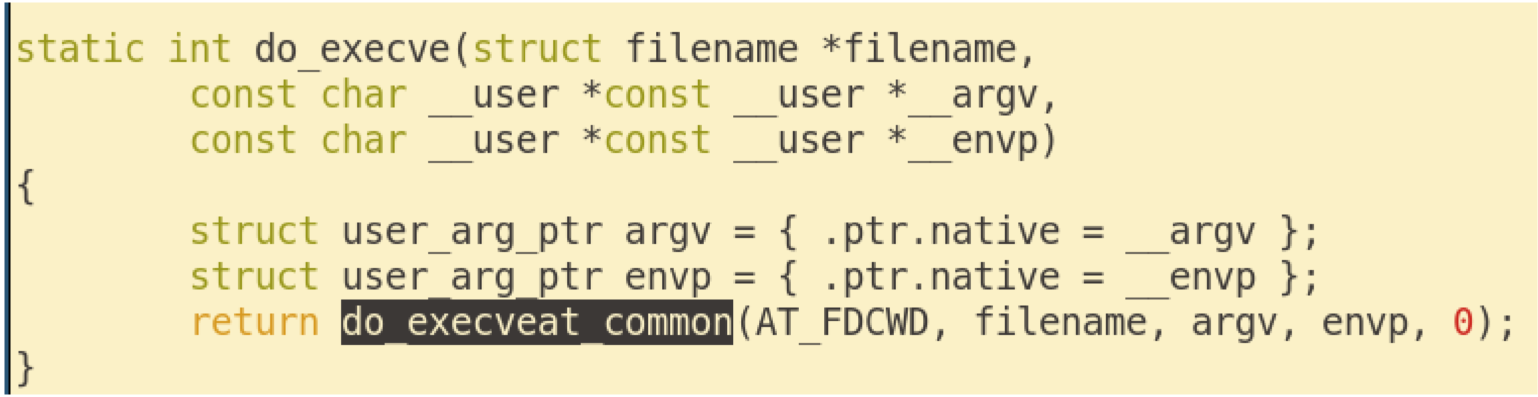


```
#ifdef CONFIG_COMPAT
COMPAT_SYSCALL_DEFINE3(execve, const char __user *, filename,
        const compat_uptr_t __user *, argv,
        const compat_uptr_t __user *, envp)
{
        return compat_do_execve(getname(filename), argv, envp);
}
```



```
SYSCALL_DEFINE3(execve,
                const char __user *, filename,
                const char __user *const __user *, argv,
                const char __user *const __user *, envp)
{
        return do_execve(getname(filename), argv, envp);
}
```

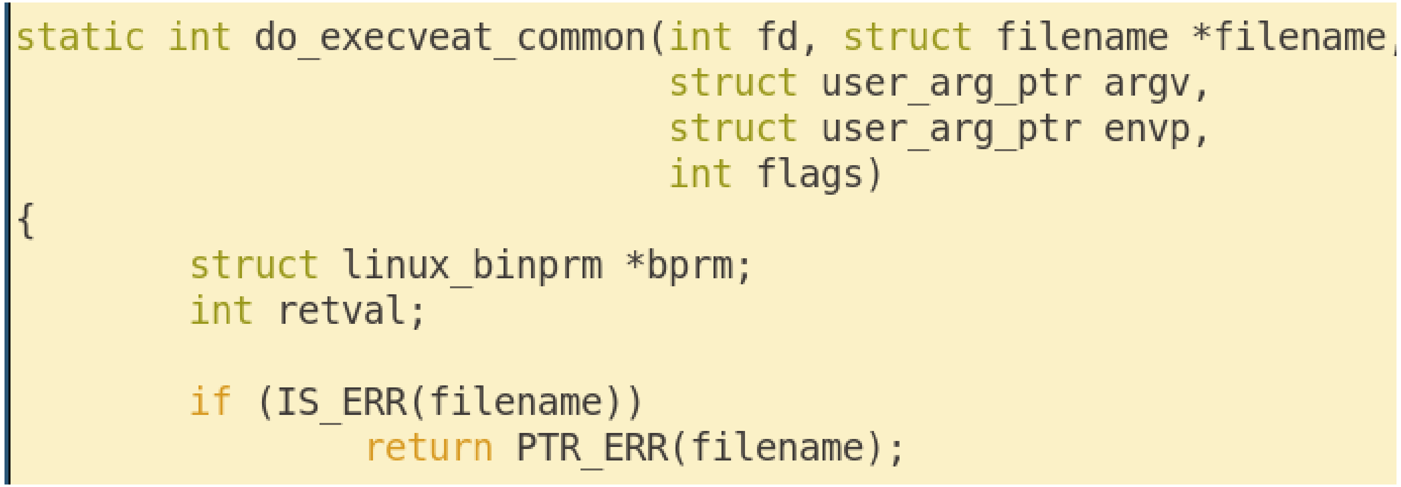
Execve - definition



```
static int do_execve(struct filename *filename,
        const char __user *const __user *__argv,
        const char __user *const __user *__envp)
{
        struct user_arg_ptr argv = { .ptr.native = __argv };
        struct user_arg_ptr envp = { .ptr.native = __envp };
        return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
}
```

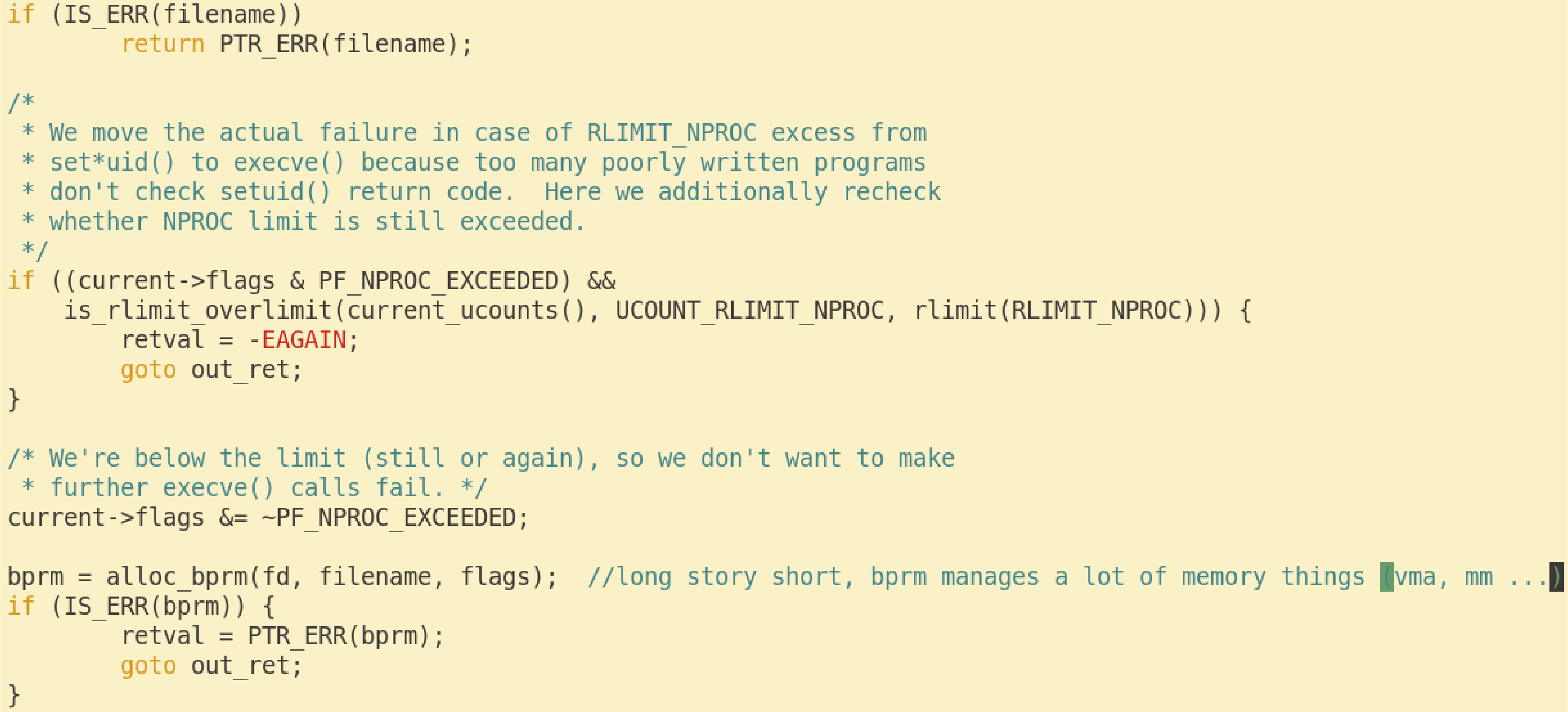
Execve - Initialization

- And finally we arrived



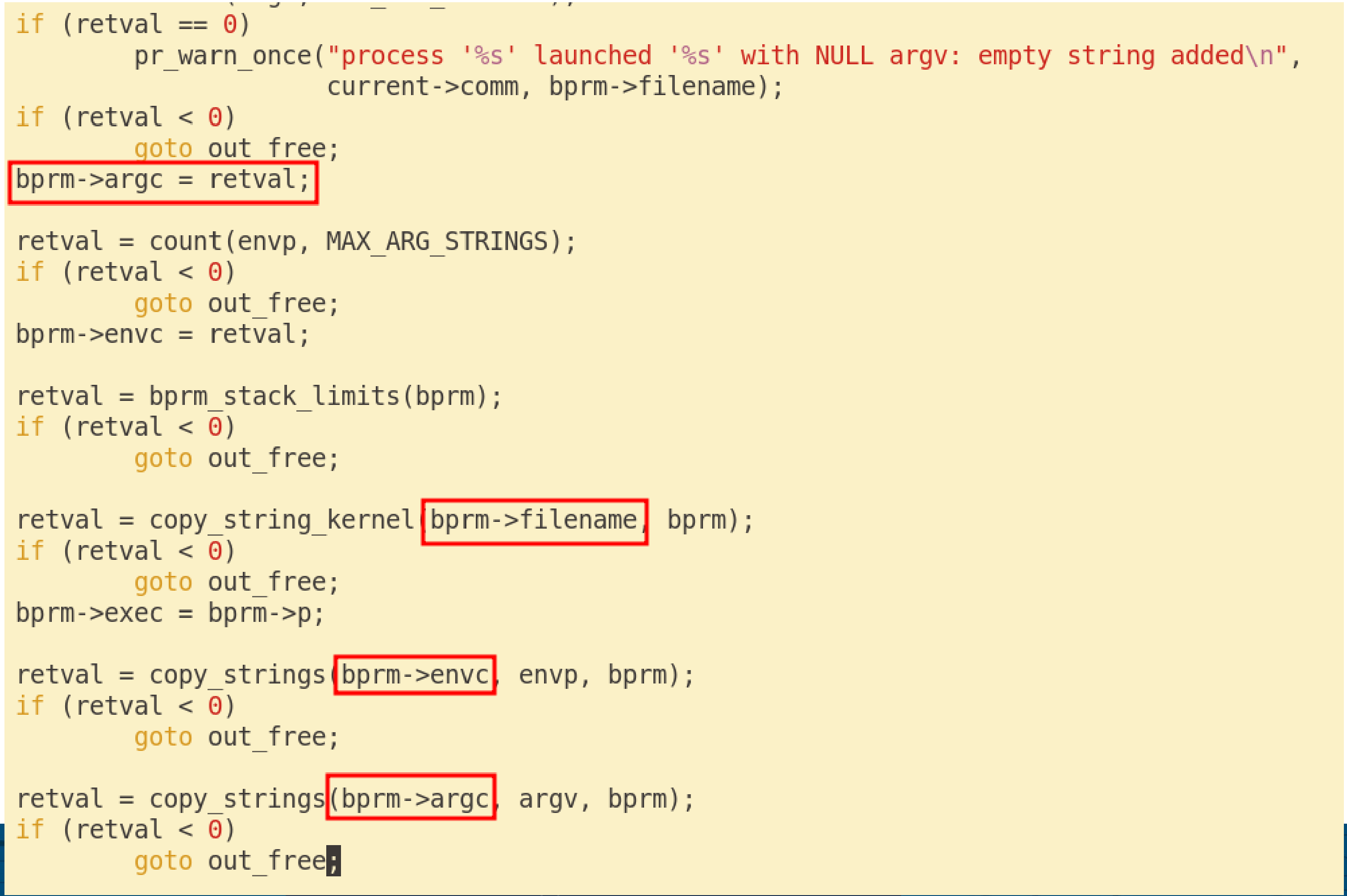
```c
static int do_execveat_common(int fd, struct filename *filename,
                              struct user_arg_ptr argv,
                              struct user_arg_ptr envp,
                              int flags)
{
        struct linux_binprm *bprm;
        int retval;

        if (IS_ERR(filename))
                return PTR_ERR(filename);
```

Execve - initialization



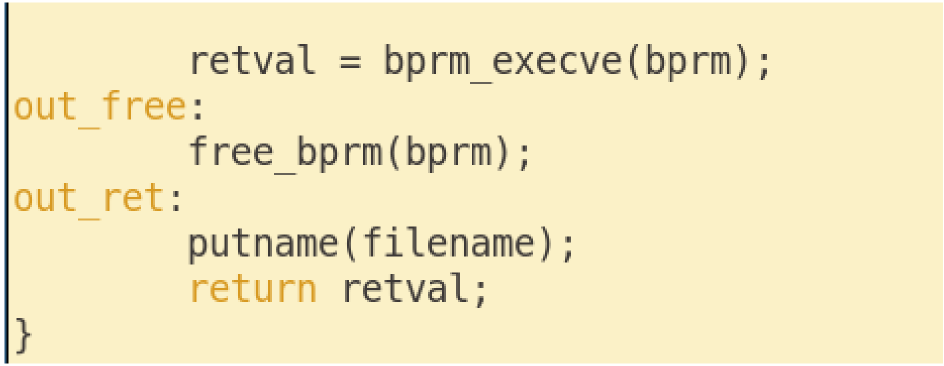
```
if (IS_ERR(filename))
        return PTR_ERR(filename);

/*
 * We move the actual failure in case of RLIMIT_NPROC excess from
 * set*uid() to execve() because too many poorly written programs
 * don't check setuid() return code.  Here we additionally recheck
 * whether NPROC limit is still exceeded.
 */
if ((current->flags & PF_NPROC_EXCEEDED) &&
    is_rlimit_overlimit(current_ucounts(), UCOUNT_RLIMIT_NPROC, rlimit(RLIMIT_NPROC))) {
        retval = -EAGAIN;
        goto out_ret;
}

/* We're below the limit (still or again), so we don't want to make
 * further execve() calls fail. */
current->flags &= ~PF_NPROC_EXCEEDED;

bprm = alloc_bprm(fd, filename, flags);  //long story short, bprm manages a lot of memory things (vma, mm ...)
if (IS_ERR(bprm)) {
        retval = PTR_ERR(bprm);
        goto out_ret;
}
```

Execve – argc,argv,envp



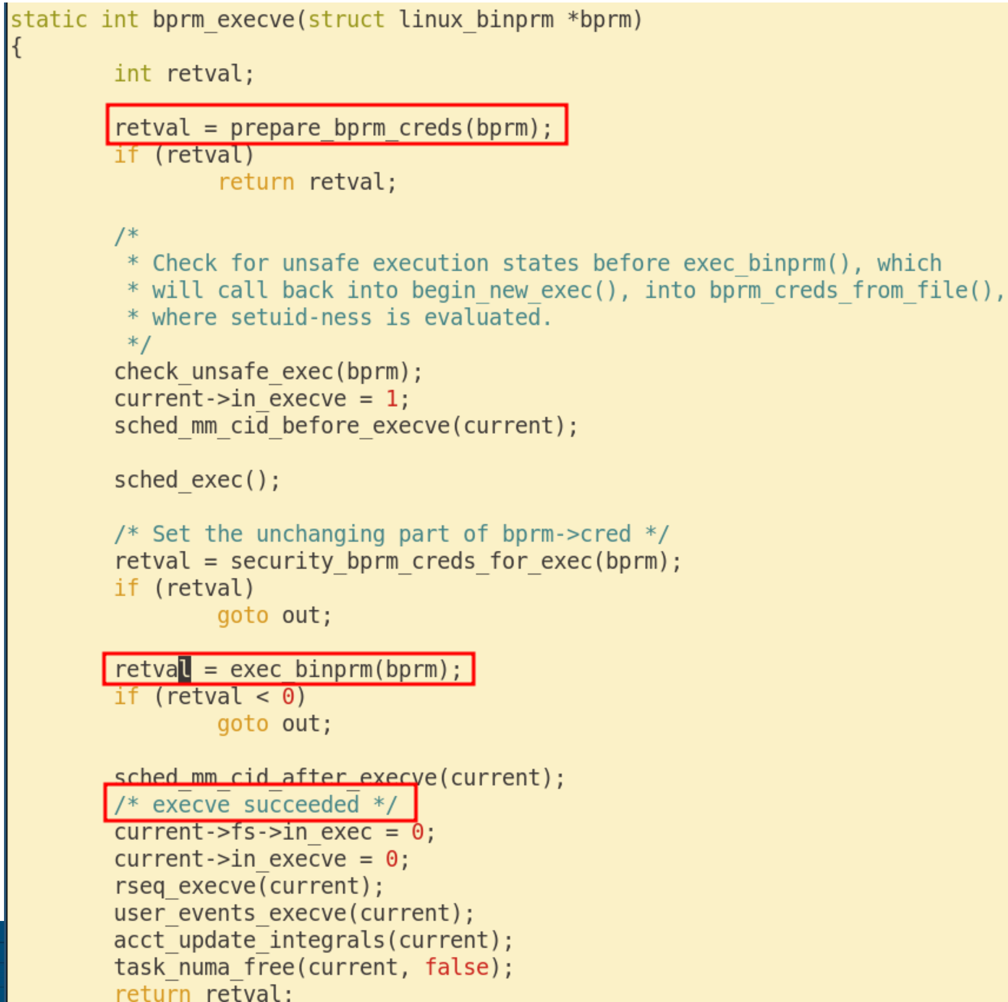
```
if (retval == 0)
        pr_warn_once("process '%s' launched '%s' with NULL argv: empty string added\n",
                     current->comm, bprm->filename);
if (retval < 0)
        goto out_free;
bprm->argc = retval;

retval = count(envp, MAX_ARG_STRINGS);
if (retval < 0)
        goto out_free;
bprm->envc = retval;

retval = bprm_stack_limits(bprm);
if (retval < 0)
        goto out_free;

retval = copy_string_kernel(bprm->filename, bprm);
if (retval < 0)
        goto out_free;
bprm->exec = bprm->p;

retval = copy_strings(bprm->envc, envp, bprm);
if (retval < 0)
        goto out_free;

retval = copy_strings(bprm->argc, argv, bprm);
if (retval < 0)
        goto out_free;
```

Execve – execution



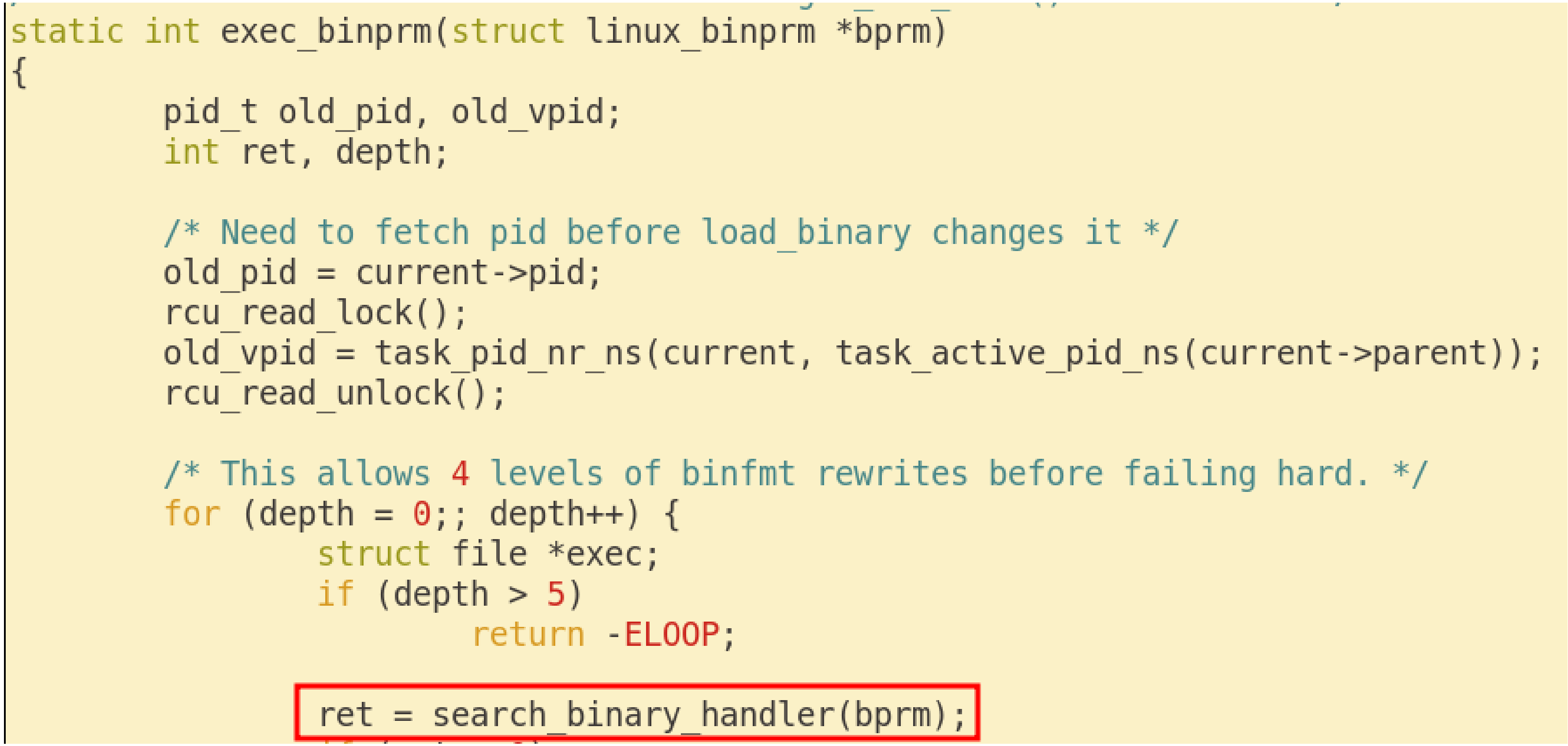
```
        retval = bprm_execve(bprm);
out_free:
        free_bprm(bprm);
out_ret:
        putname(filename);
        return retval;
}
```

Execve – credential then execution



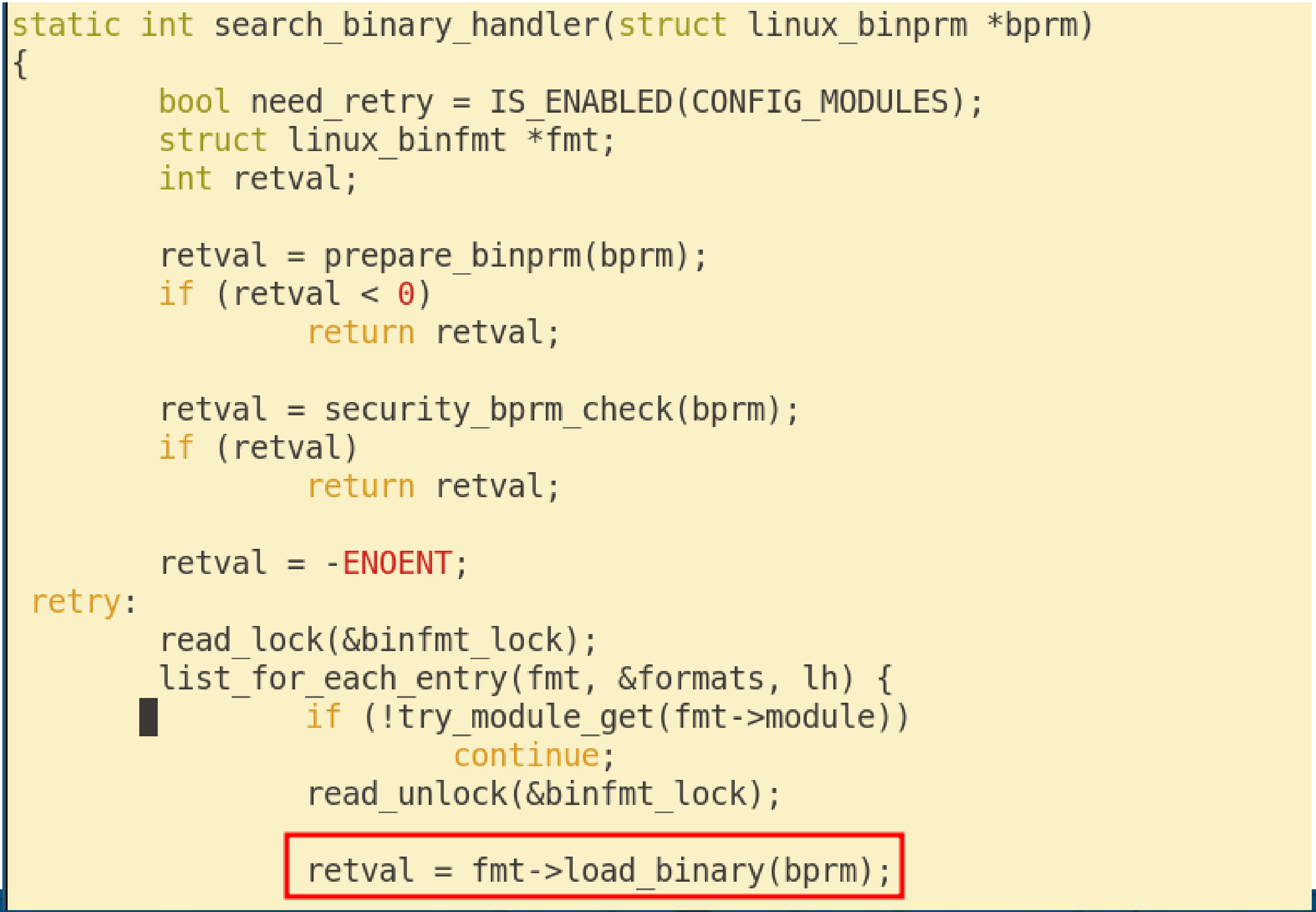
```
static int bprm_execve(struct linux_binprm *bprm)
{
        int retval;

        retval = prepare_bprm_creds(bprm);
        if (retval)
                return retval;

        /*
         * Check for unsafe execution states before exec_binprm(), which
         * will call back into begin_new_exec(), into bprm_creds_from_file(),
         * where setuid-ness is evaluated.
         */
        check_unsafe_exec(bprm);
        current->in_execve = 1;
        sched_mm_cid_before_execve(current);

        sched_exec();

        /* Set the unchanging part of bprm->cred */
        retval = security_bprm_creds_for_exec(bprm);
        if (retval)
                goto out;

        retval = exec_binprm(bprm);
        if (retval < 0)
                goto out;

        sched_mm_cid_after_execve(current);
        /* execve succeeded */
        current->fs->in_exec = 0;
        current->in_execve = 0;
        rseq_execve(current);
        user_events_execve(current);
        acct_update_integrals(current);
        task_numa_free(current, false);
        return retval;
```

Execve - execution



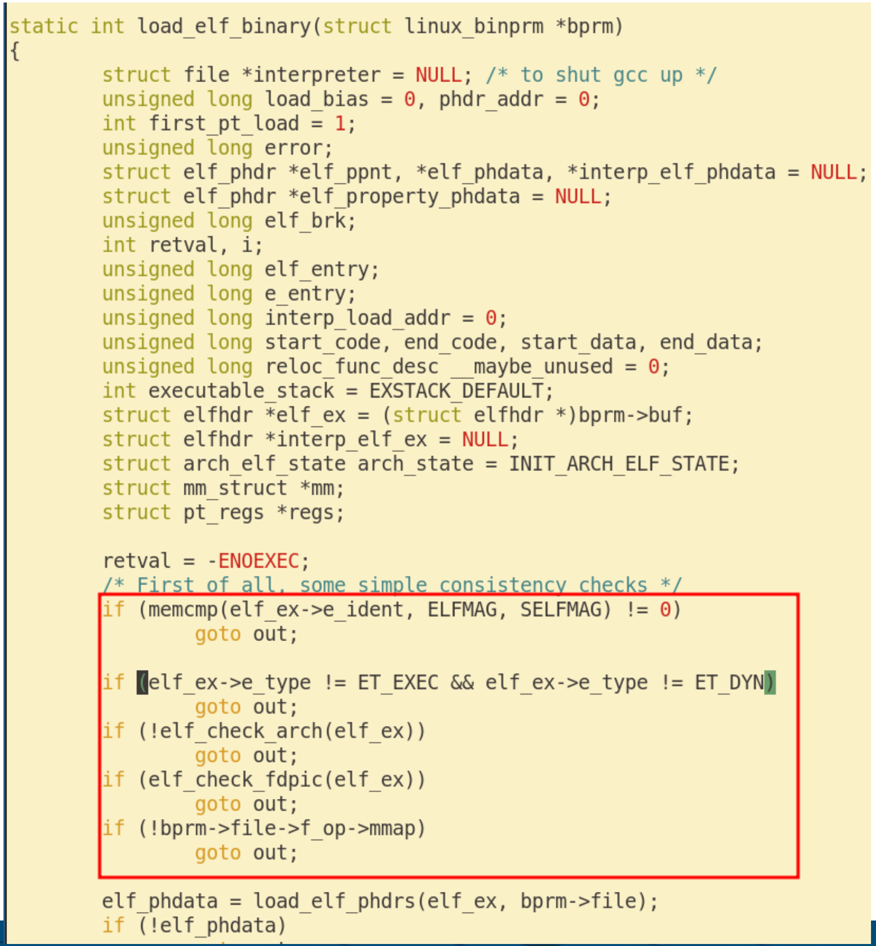
```
static int exec_binprm(struct linux_binprm *bprm)
{
        pid_t old_pid, old_vpid;
        int ret, depth;

        /* Need to fetch pid before load_binary changes it */
        old_pid = current->pid;
        rcu_read_lock();
        old_vpid = task_pid_nr_ns(current, task_active_pid_ns(current->parent));
        rcu_read_unlock();

        /* This allows 4 levels of binfmt rewrites before failing hard. */
        for (depth = 0;; depth++) {
                struct file *exec;
                if (depth > 5)
                        return -ELOOP;

                ret = search_binary_handler(bprm);
```

Execve – loading a binary



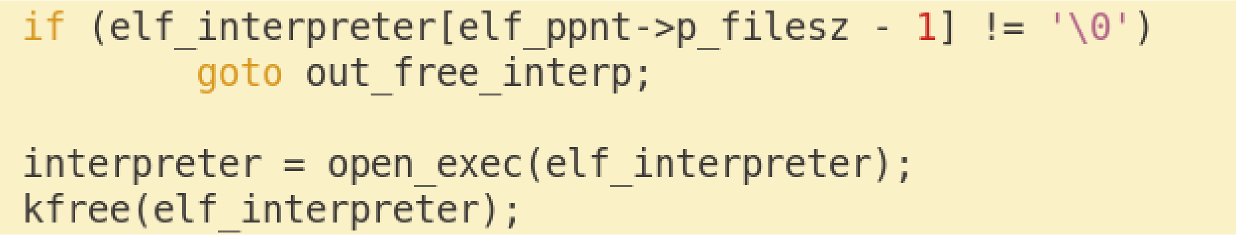
```
static int search_binary_handler(struct linux_binprm *bprm)
{
        bool need_retry = IS_ENABLED(CONFIG_MODULES);
        struct linux_binfmt *fmt;
        int retval;

        retval = prepare_binprm(bprm);
        if (retval < 0)
                return retval;

        retval = security_bprm_check(bprm);
        if (retval)
                return retval;

        retval = -ENOENT;
 retry:
        read_lock(&binfmt_lock);
        list_for_each_entry(fmt, &formats, lh) {
                if (!try_module_get(fmt->module))
                        continue;
                read_unlock(&binfmt_lock);

                retval = fmt->load_binary(bprm);
```

Execve – loading an ELF



```c
static int load_elf_binary(struct linux_binprm *bprm)
{
        struct file *interpreter = NULL; /* to shut gcc up */
        unsigned long load_bias = 0, phdr_addr = 0;
        int first_pt_load = 1;
        unsigned long error;
        struct elf_phdr *elf_ppnt, *elf_phdata, *interp_elf_phdata = NULL;
        struct elf_phdr *elf_property_phdata = NULL;
        unsigned long elf_brk;
        int retval, i;
        unsigned long elf_entry;
        unsigned long e_entry;
        unsigned long interp_load_addr = 0;
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long reloc_func_desc __maybe_unused = 0;
        int executable_stack = EXSTACK_DEFAULT;
        struct elfhdr *elf_ex = (struct elfhdr *)bprm->buf;
        struct elfhdr *interp_elf_ex = NULL;
        struct arch_elf_state arch_state = INIT_ARCH_ELF_STATE;
        struct mm_struct *mm;
        struct pt_regs *regs;

        retval = -ENOEXEC;
        /* First of all, some simple consistency checks */
        if (memcmp(elf_ex->e_ident, ELFMAG, SELFMAG) != 0)
                goto out;

        if (elf_ex->e_type != ET_EXEC && elf_ex->e_type != ET_DYN)
                goto out;
        if (!elf_check_arch(elf_ex))
                goto out;
        if (elf_check_fdpic(elf_ex))
                goto out;
        if (!bprm->file->f_op->mmap)
                goto out;

        elf_phdata = load_elf_phdrs(elf_ex, bprm->file);
        if (!elf_phdata)
```

Execve – starting the interpreter

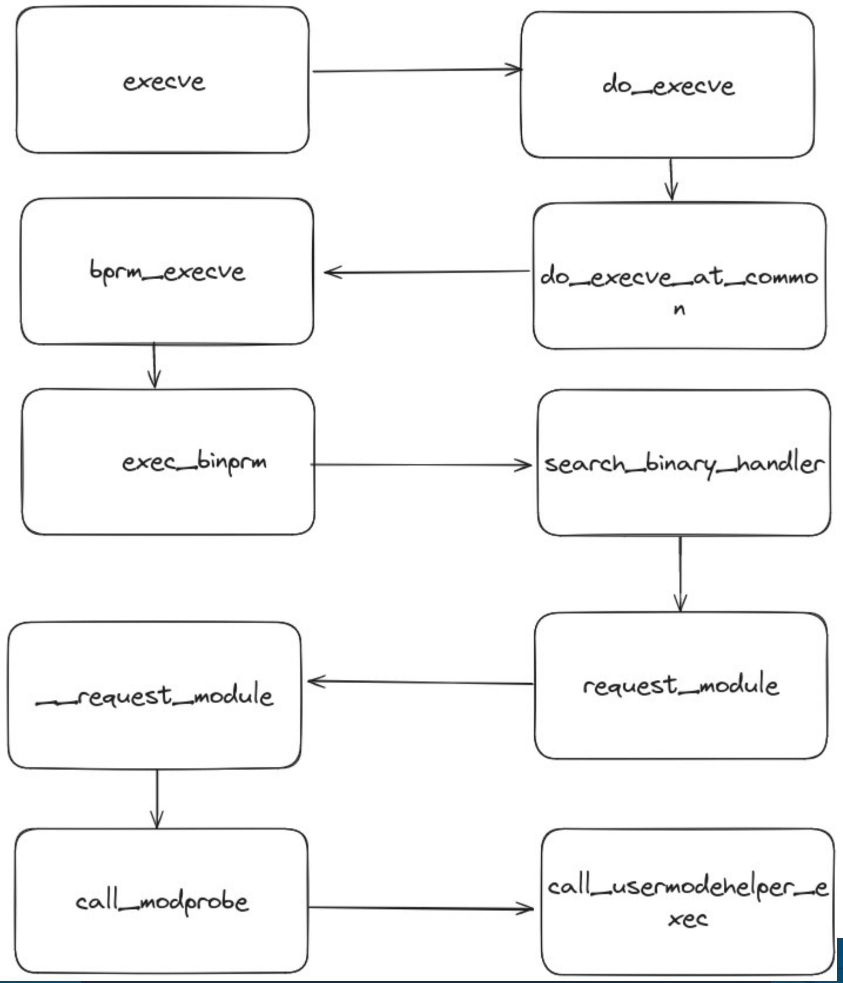


```c
if (elf_interpreter[elf_ppnt->p_filesz - 1] != '\0')
        goto out_free_interp;

interpreter = open_exec(elf_interpreter);
kfree(elf_interpreter);
```

Execve – please no more

- It goes on and on for at least thousand lines of codes so I’ll spare the trouble in the end it start the thread

Execve - summary



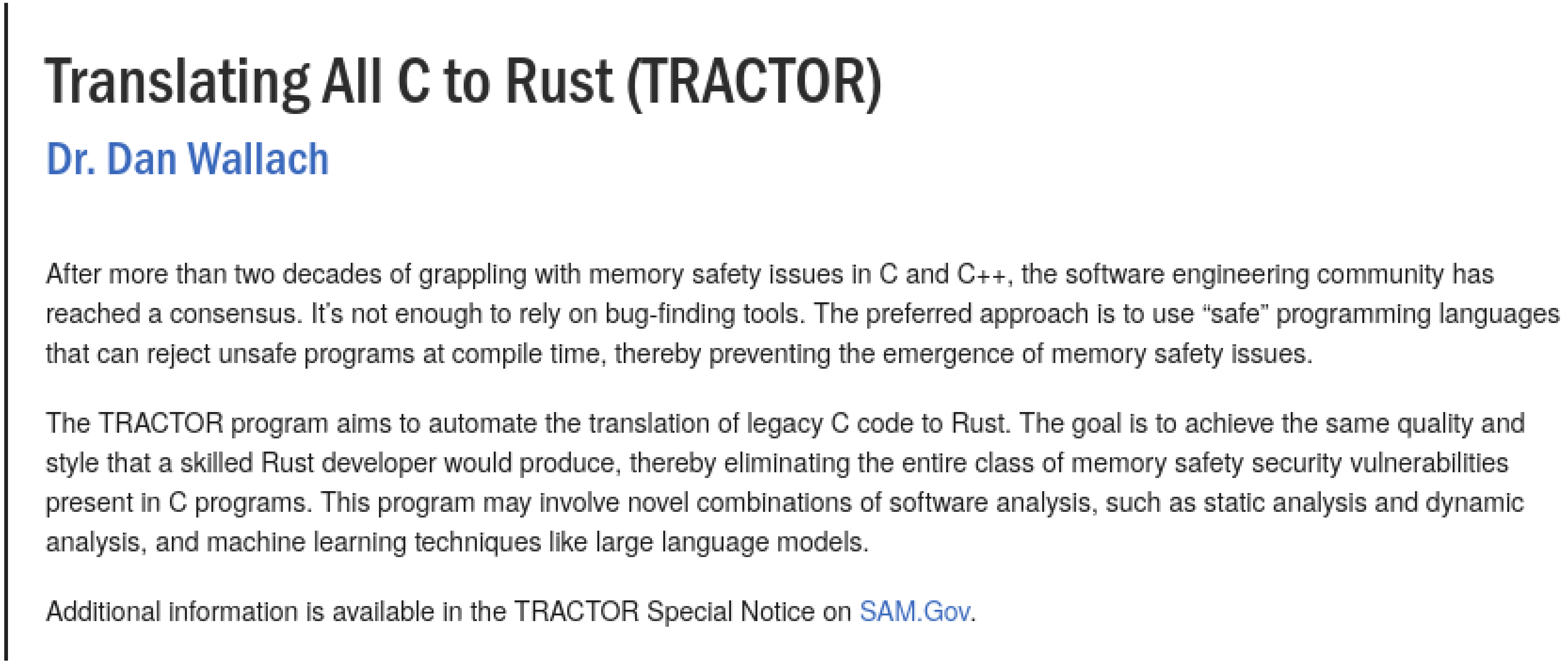
2

Rust advanced

Why rust

- Language that is rising
  - Starting to be adopted by Microsoft (kernel)
  - Android (kernel)
  - Linux (kernel)
  - MacOS and iOS was in reflexion (not planned afaik)
  - Other places?

Why rust



Translating All C to Rust (TRACTOR)

Dr. Dan Wallach

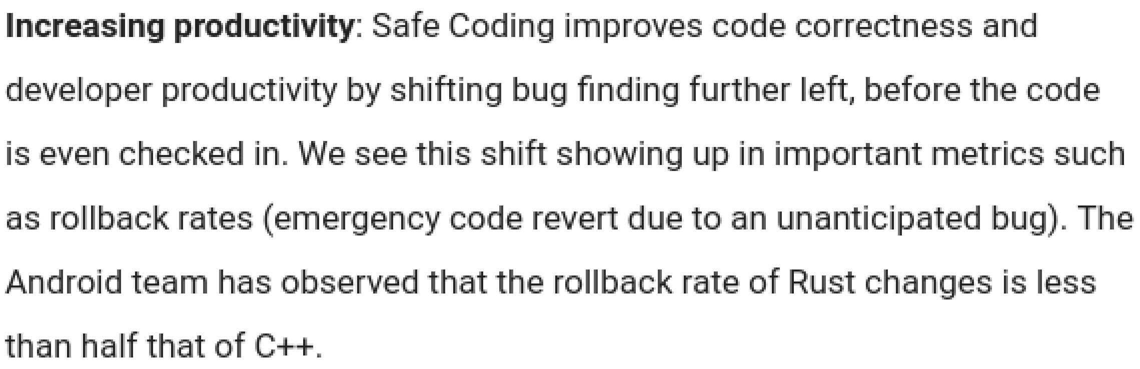
After more than two decades of grappling with memory safety issues in C and C++, the software engineering community has reached a consensus. It's not enough to rely on bug-finding tools. The preferred approach is to use “safe” programming languages that can reject unsafe programs at compile time, thereby preventing the emergence of memory safety issues.

The TRACTOR program aims to automate the translation of legacy C code to Rust. The goal is to achieve the same quality and style that a skilled Rust developer would produce, thereby eliminating the entire class of memory safety security vulnerabilities present in C programs. This program may involve novel combinations of software analysis, such as static analysis and dynamic analysis, and machine learning techniques like large language models.

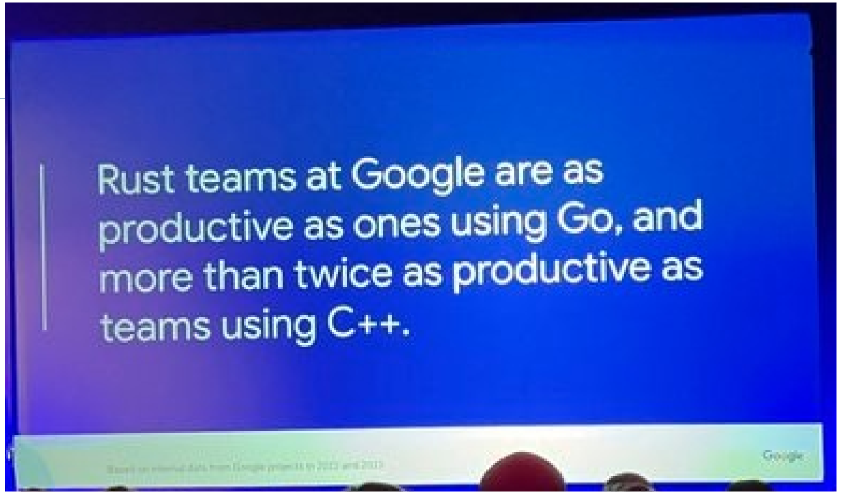
Additional information is available in the TRACTOR Special Notice on SAM.Gov.

Source: https://www.darpa.mil/program/translating-all-c-to-rust

Rust – more productiv

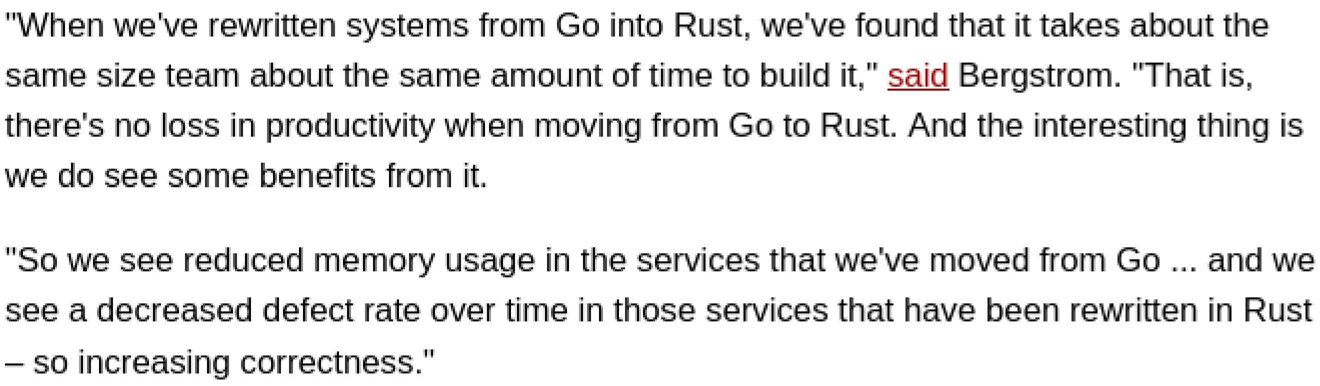


Increasing productivity: Safe Coding improves code correctness and developer productivity by shifting bug finding further left, before the code is even checked in. We see this shift showing up in important metrics such as rollback rates (emergency code revert due to an unanticipated bug). The Android team has observed that the rollback rate of Rust changes is less than half that of C++.



Rust teams at Google are as productive as ones using Go, and more than twice as productive as teams using C++.

Based on internal data from Google projects in 2022 and 2023



"When we've rewritten systems from Go into Rust, we've found that it takes about the same size team about the same amount of time to build it," said Bergstrom. "That is, there's no loss in productivity when moving from Go to Rust. And the interesting thing is we do see some benefits from it.

"So we see reduced memory usage in the services that we've moved from Go ... and we see a decreased defect rate over time in those services that have been rewritten in Rust – so increasing correctness."

Source: https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html
Source: https://x.com/spastorino/status/1773025016822497392
Source : https://www.youtube.com/live/6mZRWFQRvmw?t=27048s

Rust - Introduction

- https://tc.gts3.org/cs3210/2020/spring/l/lec03/lec03.html#next-lecture

Rust - unsafes

- Unsafe
- Transmute
- Unions
- Rust as a TCB (Trusted Computing Base)
- https://www.reddit.com/r/rust/comments/c57oos/evading_rusts_memory_safety/

C runtime

- https://dev.gentoo.org/~vapier/crt.txt
-

Rust – safe but vulnerable

- VecDeque with CVE https://gts3.org/2019/cve-2018-1000657.html
- https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html
- https://huonw.github.io/blog/2016/04/myths-and-legends-about-integer-overflow-in-rust/
- https://huonw.github.io/blog/2014/07/what-does-rusts-unsafe-mean/

Rust panic handlers

- https://doc.rust-lang.org/nomicon/panic-handler.html
- https://doc.rust-lang.org/nomicon/panic-handler.html
- https://llvm.org/docs/ExceptionHandling.html
- https://docs.rust-embedded.org/book/

Rust misc

- Traits and rust language are tightly coupled
- https://pitdicker.github.io/Interior-mutability-patterns/
- https://tc.gts3.org/cs3210/2020/spring/l/lec12/lec12.html#references

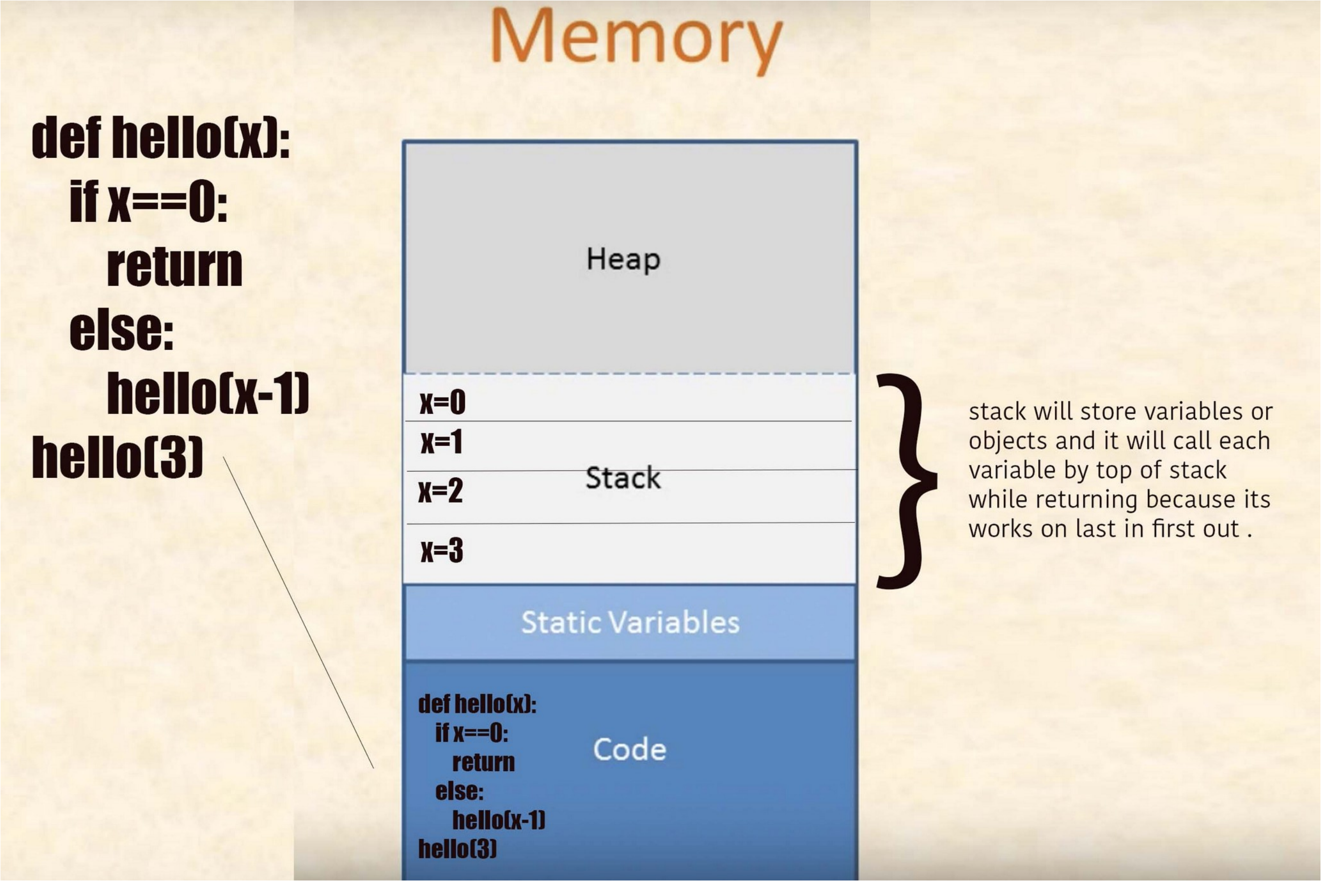
1

Memory management and rust memory model

Memory management in rust

- no_std and libcore https://doc.rust-lang.org/core/index.html or liballoc or libstd
  - Means no dynamic allocation

Memory management in rust



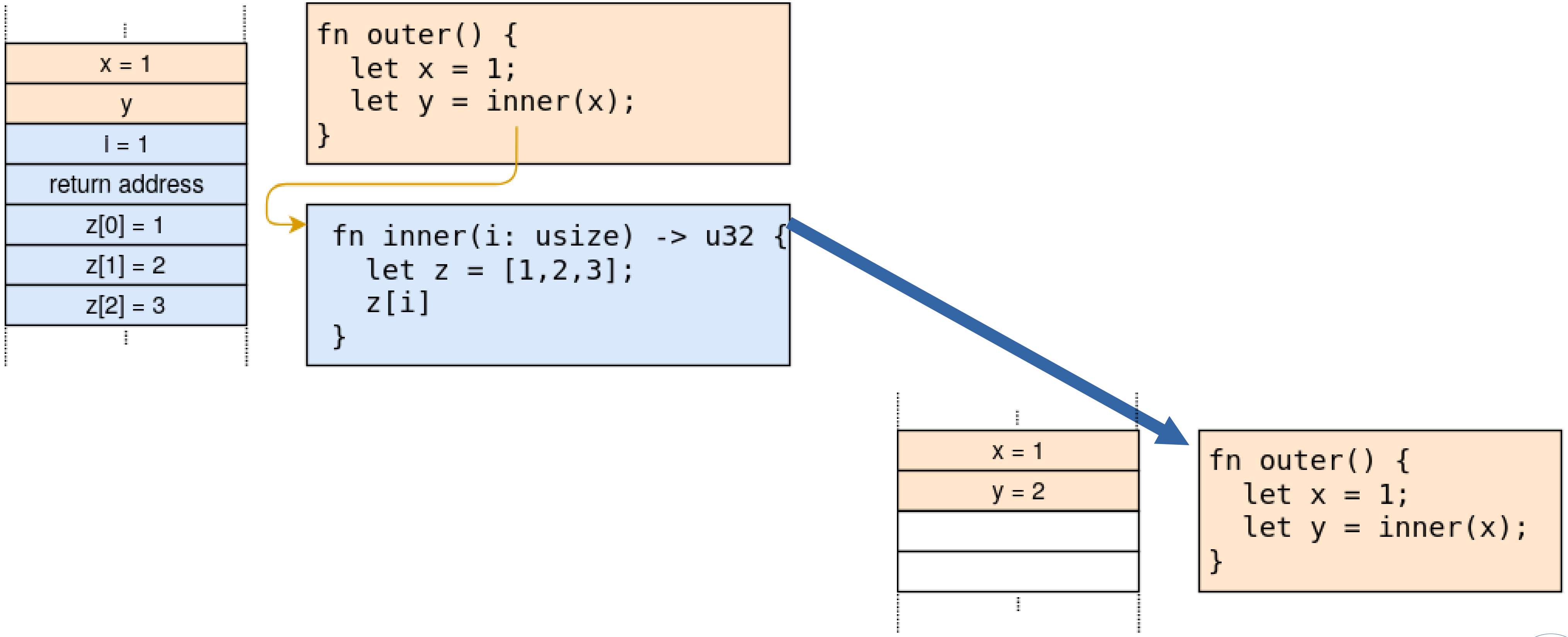
```
def hello(x):
   if x==0:
      return
   else:
      hello(x-1)
hello(3)
```

```
def hello(x):
  if x==0:
    return
  else:
    hello(x-1)
hello(3)
```

stack will store variables or objects and it will call each variable by top of stack while returning because its works on last in first out .

Source : https://os.phil-opp.com/heap-allocation/

Memory management – stack frame



x = 1
y
i = 1
return address
z[0] = 1
z[1] = 2
z[2] = 3

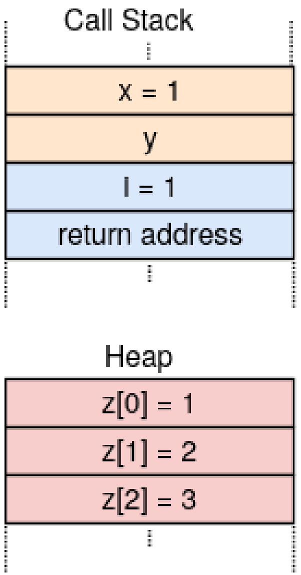
```
fn outer() {
  let x = 1;
  let y = inner(x);
}
```

```
fn inner(i: usize) -> u32 {
  let z = [1,2,3];
  z[i]
}
```

x = 1
y = 2

```
fn outer() {
  let x = 1;
  let y = inner(x);
}
```

Memory management – the heap



Call Stack

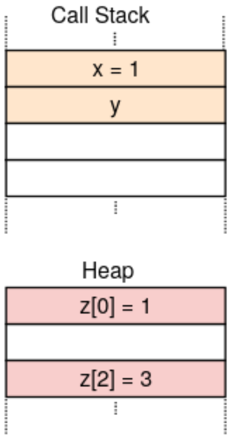
x = 1
y
I = 1
return address

Heap

z[0] = 1
z[1] = 2
z[2] = 3

```
fn outer() {
  let x = 1;
  let y = inner(x);
  deallocate(y, size_of(u32));
}
```

```
fn inner(i: usize) -> *mut u32 {
  let z = allocate(size_of([u32;3]))
  z.write([1,2,3]);
  (z as *mut u32).offset(i)
}
```



Call Stack

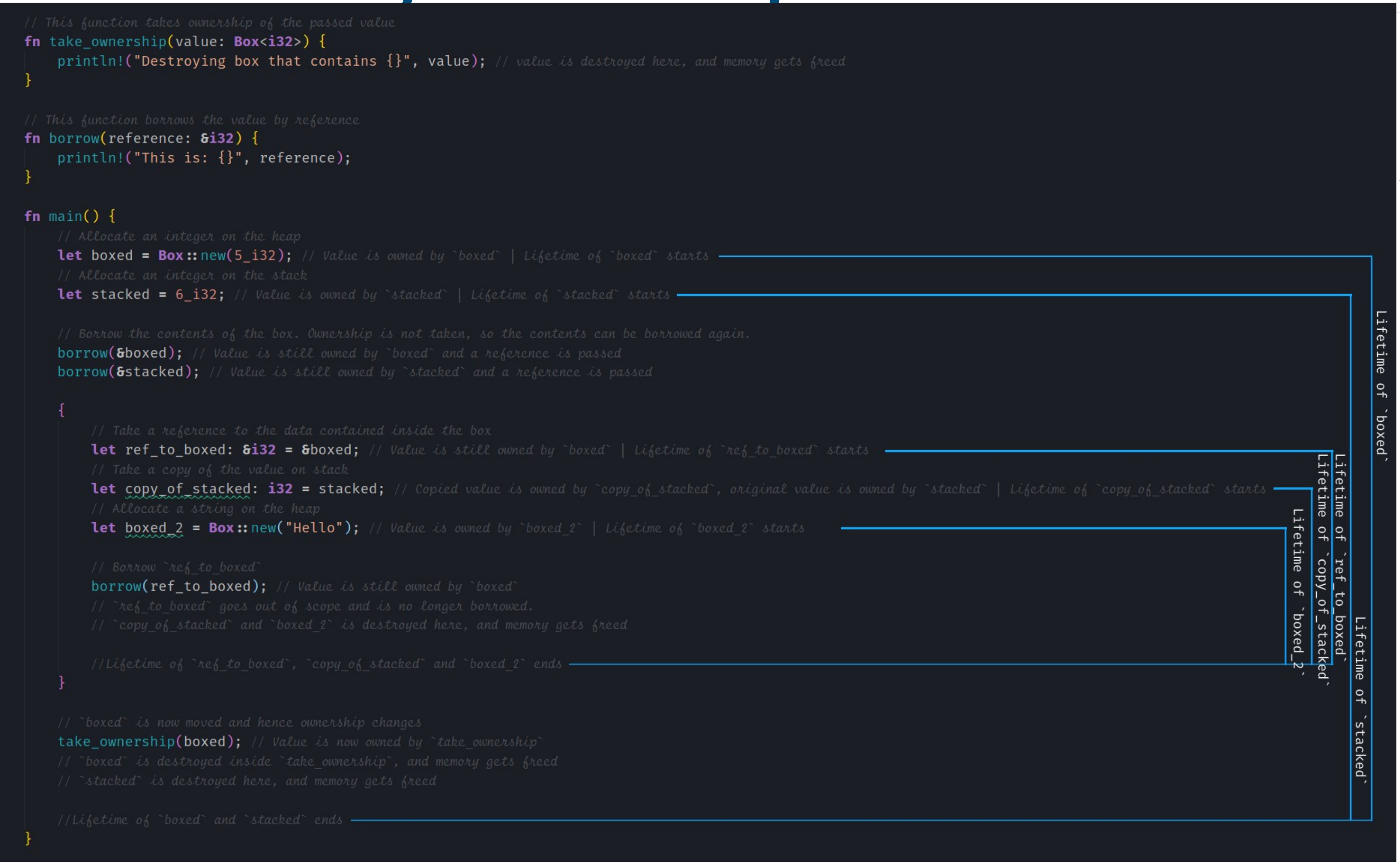
x = 1
y

Heap

z[0] = 1
z[2] = 3

```
fn outer() {
  let x = 1;
  let y = inner(x);
  deallocate(y, size_of(u32));
}
```

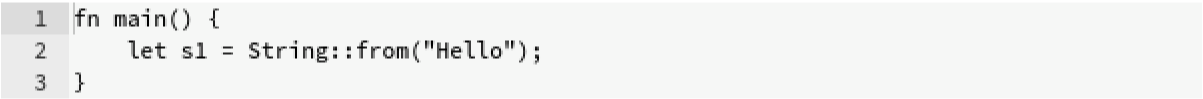
Memory management – The stack



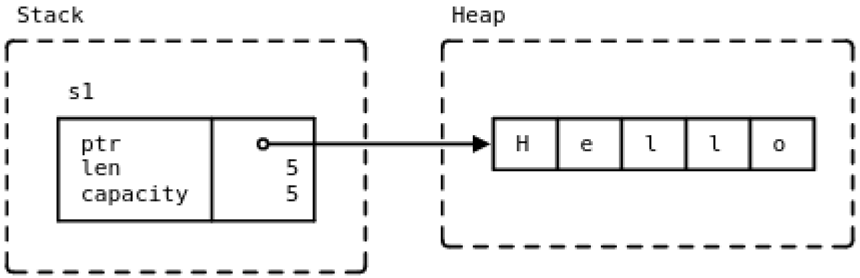
```rust
// This function takes ownership of the passed value
fn take_ownership(value: Box<i32>) {
    println!("Destroying box that contains {}", value); // value is destroyed here, and memory gets freed
}

// This function borrows the value by reference
fn borrow(reference: &i32) {
    println!("This is: {}", reference);
}

fn main() {
    // Allocate an integer on the heap
    let boxed = Box::new(5_i32); // Value is owned by `boxed` | Lifetime of `boxed` starts
    // Allocate an integer on the stack
    let stacked = 6_i32; // Value is owned by `stacked` | Lifetime of `stacked` starts

    // Borrow the contents of the box. Ownership is not taken, so the contents can be borrowed again.
    borrow(&boxed); // Value is still owned by `boxed` and a reference is passed
    borrow(&stacked); // Value is still owned by `stacked` and a reference is passed

    {
        // Take a reference to the data contained inside the box
        let ref_to_boxed: &i32 = &boxed; // Value is still owned by `boxed` | Lifetime of `ref_to_boxed` starts
        // Take a copy of the value on stack
        let copy_of_stacked: i32 = stacked; // Copied value is owned by `copy_of_stacked`, original value is owned by `stacked` | Lifetime of `copy_of_stacked` starts
        // Allocate a string on the heap
        let boxed_2 = Box::new("Hello"); // Value is owned by `boxed_2` | Lifetime of `boxed_2` starts

        // Borrow `ref_to_boxed`
        borrow(ref_to_boxed); // Value is still owned by `boxed`
        // `ref_to_boxed` goes out of scope and is no longer borrowed.
        // `copy_of_stacked` and `boxed_2` is destroyed here, and memory gets freed

        //Lifetime of `ref_to_boxed`, `copy_of_stacked` and `boxed_2` ends
    }

    // `boxed` is now moved and hence ownership changes
    take_ownership(boxed); // Value is now owned by `take_ownership`
    // `boxed` is destroyed inside `take_ownership`, and memory gets freed
    // `stacked` is destroyed here, and memory gets freed

    //Lifetime of `boxed` and `stacked` ends
}
```

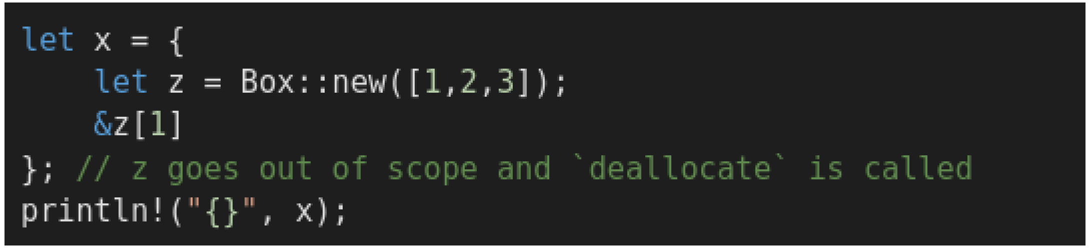
source: https://deepu.tech/memory-management-in-rust/

Memory management – The heap



```
fn main() {
    let s1 = String::from("Hello");
}
```





```
let x = {
    let z = Box::new([1,2,3]);
    &z[1]
}; // z goes out of scope and `deallocate` is called
println!("{}", x);
```

Memory management – The heap

- Mix of automatic and controlled memory
- Memory safety is enforced at compile time
- Tries to have zero cost abstraction on freeing timing as well

Memory management – Further readings

- Understanding ownership in rust
- C/C++ and the stack and heap
- Debugging a high jump in heap consumption of a project in rust for a web project
  (spoiler it uses jemalloc in the end)
- A nice explanation on how drop works based (in french)
- Understanding the heap and the stack
- Debugging a memory leak in rust caused by a bug in Vec
- Using rust for a nice API on heapless datastructures that still can grow wit h little unsafe

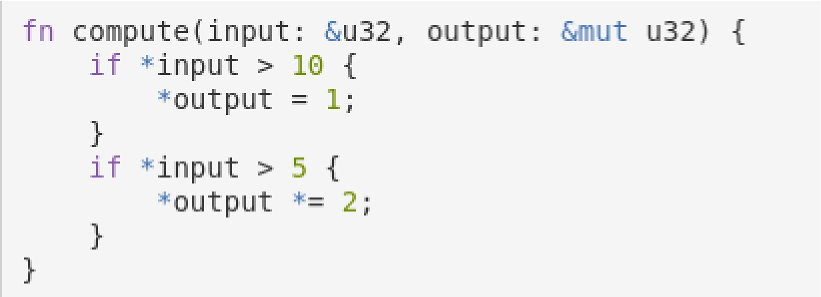
Memory management – Further readings

- Discovering some problems while trying to develop a simple allocator in rust

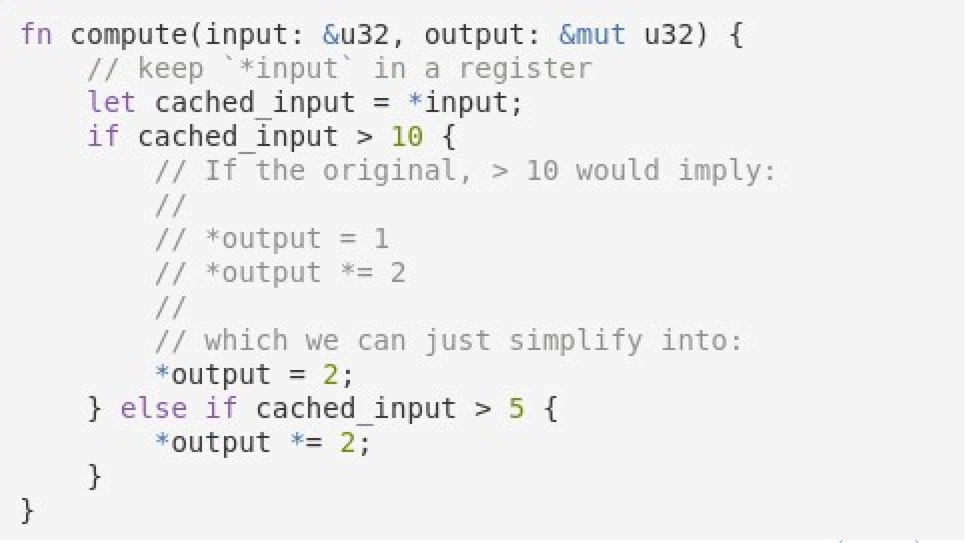
2 Memory model - aliasing

Memory model - Aliasing

- If input is not the same memory region as output, then no problem
- Else aliasing problem
- CPU needs to know if he wants to optimize the first code into the second
- But it’s a hard problem



```rust
fn compute(input: &u32, output: &mut u32) {
    if *input > 10 {
        *output = 1;
    }
    if *input > 5 {
        *output *= 2;
    }
}
```



```rust
fn compute(input: &u32, output: &mut u32) {
    // keep `*input` in a register
    let cached_input = *input;
    if cached_input > 10 {
        // If the original, > 10 would imply:
        //
        // *output = 1
        // *output *= 2
        //
        // which we can just simplify into:
        *output = 2;
    } else if cached_input > 5 {
        *output *= 2;
    }
}
```

Memory management - Aliasing

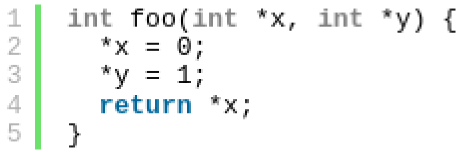
- Rust differentiate unique mutable references (&mut) and shared immutable (&)
- & are “read only” → no problem on aliasing
- &mut can have side effect

Memory management - Aliasing

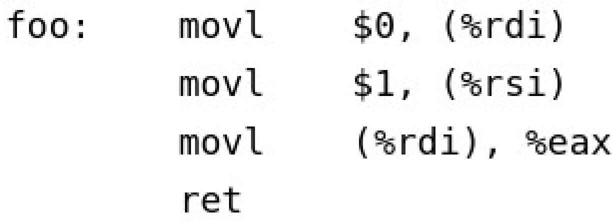
- Quick vocabulary definition
  - memory is anonymous if the programmer cannot refer to it by name or pointer.
  - memory is unaliased if there is currently only one way to refer to it.

Memory management - Aliasing

- Aliasing is hard
  - Strict aliasing (more optimization but less choice for programmer) causes problems on C/C++ : link



```c
int foo(int *x, int *y) {
  *x = 0;
  *y = 1;
  return *x;
}
```

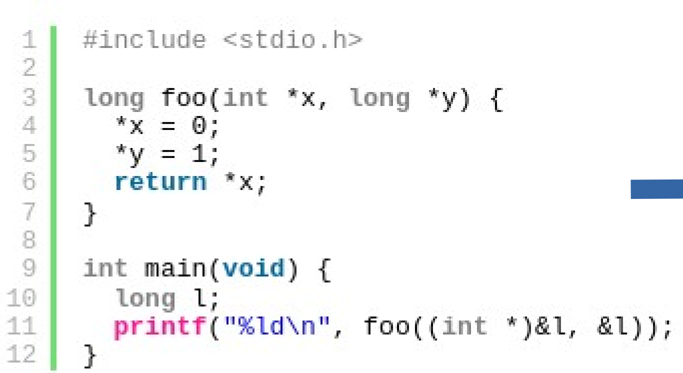


```
foo:     movl    $0, (%rdi)
         movl    $1, (%rsi)
         movl    (%rdi), %eax
         ret
```

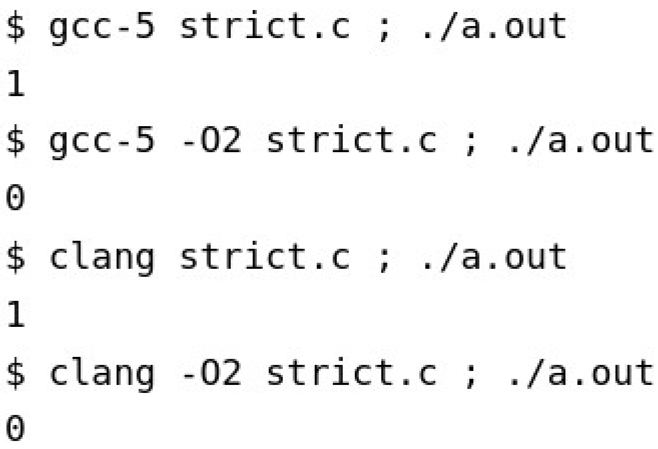
Source: https://blog.regehr.org/archives/1307

Memory management - Aliasing

- Bad inference about aliasing causes bugs



```c
#include <stdio.h>

long foo(int *x, long *y) {
  *x = 0;
  *y = 1;
  return *x;
}

int main(void) {
  long l;
  printf("%ld\n", foo((int *)&l, &l));
}
```



```
$ gcc-5 strict.c ; ./a.out
1
$ gcc-5 -O2 strict.c ; ./a.out
0
$ clang strict.c ; ./a.out
1
$ clang -O2 strict.c ; ./a.out
0
```

Source: https://blog.regehr.org/archives/1307

Memory management - Aliasing

- For more exploration on C/C++ memory model and semantics
  - Cerberus project
  - Cerberus project BMC (test your code for UB)

Memory management - Aliasing

- To ensure two pointers do not alias, rust needs to know provenance
- We’ll stop here – We’re not trying to do an advanced rust class
- But it was there to explain, pointer and aliasing is hard

Memory management – Aliasing further readings

- Fix rust unsafe pointers
- Writing unsafe rust is harder than writing C
- More on aliasing through the rustnomicon
- Memory management in rust
- Pointer aliasing in rust
- Old try to have better unsafe pointer in rust : An RFC
- Unsafe rust is not C

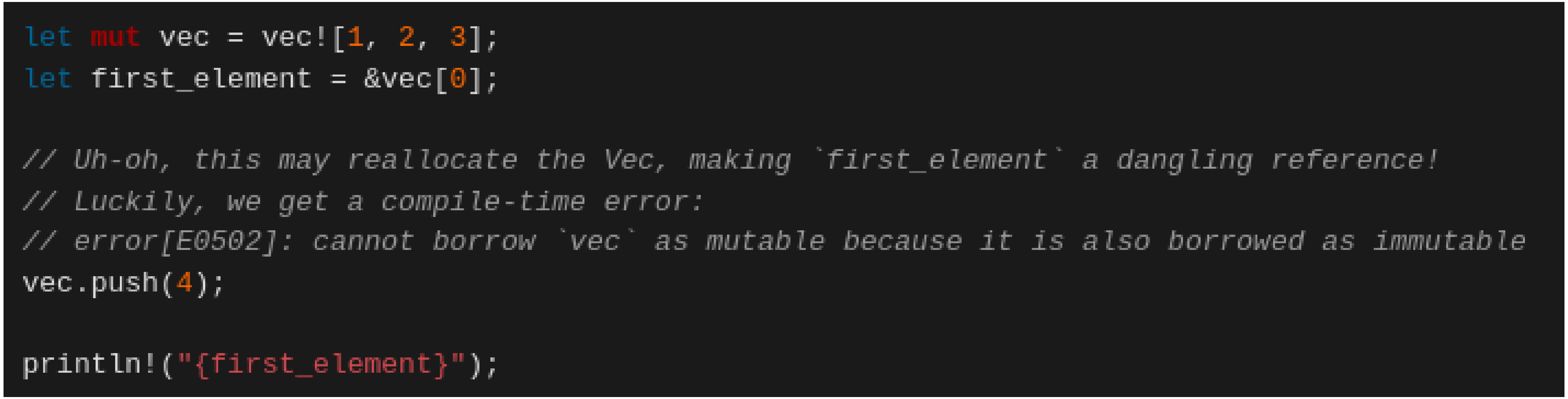
3

Memory management – Ownership and borrowing

Memory management - Borrowing

- Why is it so hard?

Memory management - Borrowing

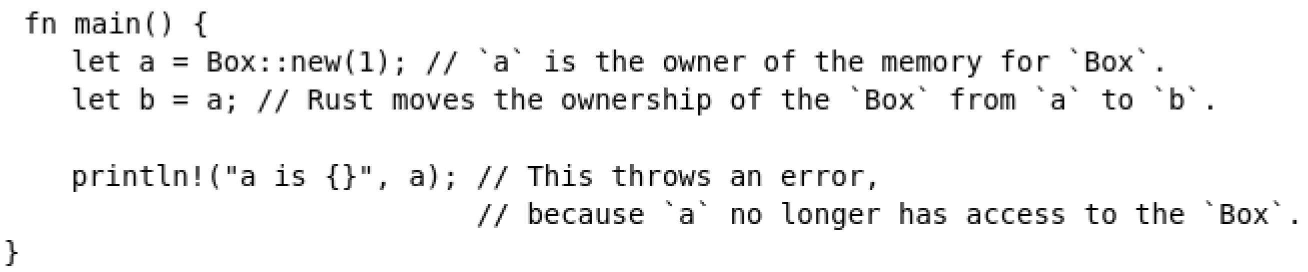


```rust
let mut vec = vec![1, 2, 3];
let first_element = &vec[0];

// Uh-oh, this may reallocate the Vec, making `first_element` a dangling reference!
// Luckily, we get a compile-time error:
// error[E0502]: cannot borrow `vec` as mutable because it is also borrowed as immutable
vec.push(4);

println!("{first_element}");
```

Source: https://antelang.org/blog/safe_shared_mutability/

Memory management - Ownership

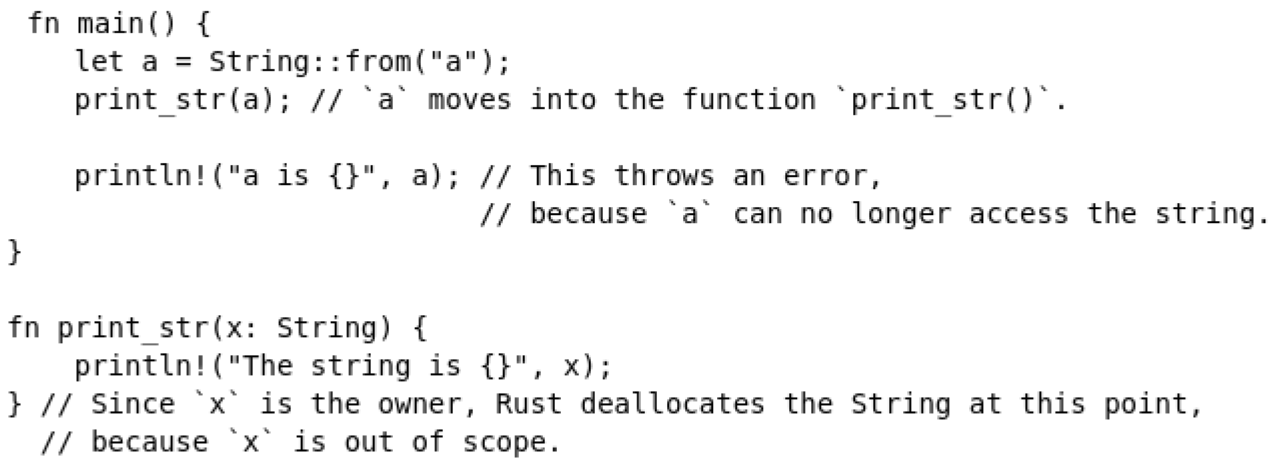
- If variables can be borrowed, it implies some ownership
  - The variable instanciated first own the data
  - If one borrows it the variable is invalidated
  - Else exists until death of variable (drop)
  - In other words, rust is moved by default



```
 fn main() {
   let a = Box::new(1); // `a` is the owner of the memory for `Box`.
   let b = a; // Rust moves the ownership of the `Box` from `a` to `b`.

   println!("a is {}", a); // This throws an error,
                           // because `a` no longer has access to the `Box`.
}
```

Memory management - Ownership

- Functions can take ownership

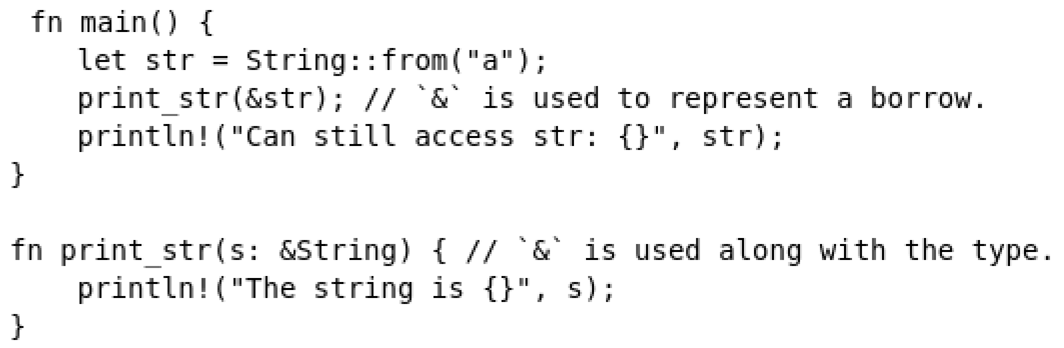


```
fn main() {
    let a = String::from("a");
    print_str(a); // `a` moves into the function `print_str()`.

    println!("a is {}", a); // This throws an error,
                            // because `a` can no longer access the string.
}

fn print_str(x: String) {
    println!("The string is {}", x);
} // Since `x` is the owner, Rust deallocates the String at this point,
  // because `x` is out of scope.
```

https://cmpt-479-982.github.io/week1/safety_features_of_rust.html

Memory management - Ownership

- Borrowing is useful for this



```
 fn main() {
    let str = String::from("a");
    print_str(&str); // `&` is used to represent a borrow.
    println!("Can still access str: {}", str);
}

fn print_str(s: &String) { // `&` is used along with the type.
    println!("The string is {}", s);
}
```

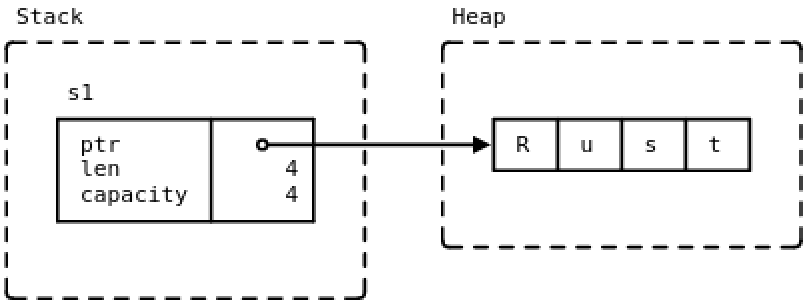
Memory management - Ownership

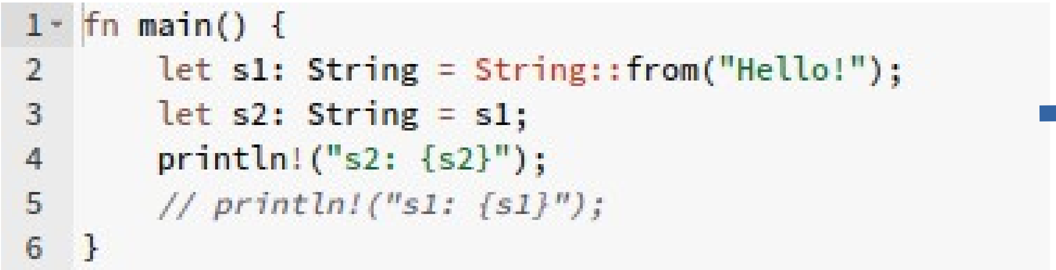
The fundamental concept in rust

Aliasable XOR mutable

The default should be that values can be mutated only if they are not aliased, and there should be no way to introduce unsynchronized aliased mutation. However, the language should support mutating values. The only way to get this is the rest of ownership and borrowing, the distinction between borrows and mutable borrows and the aliasing rules between them.

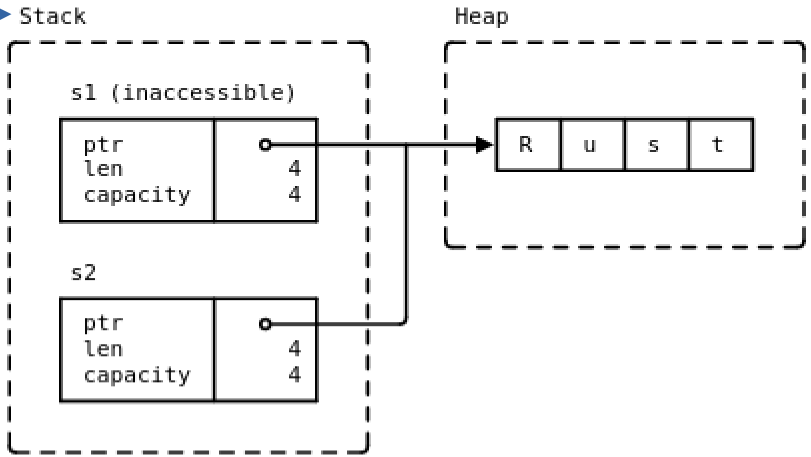
Memory management - Ownership



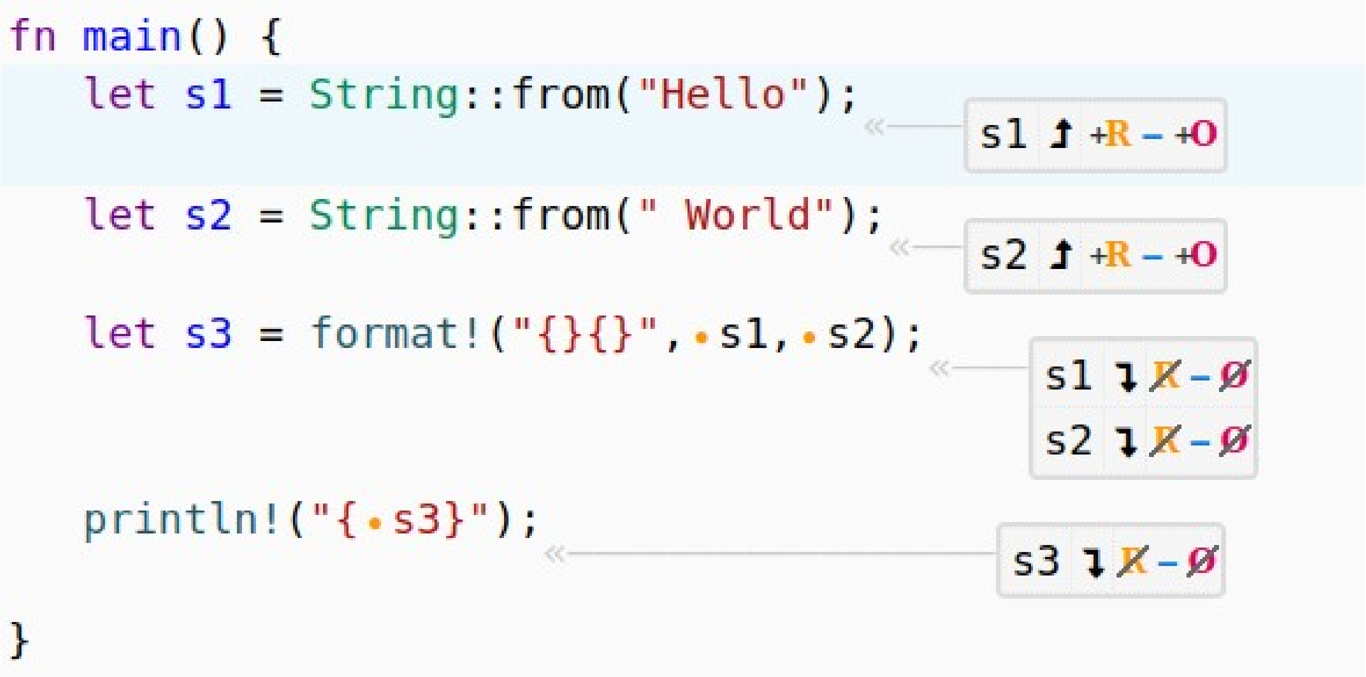


```rust
fn main() {
    let s1: String = String::from("Hello!");
    let s2: String = s1;
    println!("s2: {s2}");
    // println!("s1: {s1}");
}
```

After move to `s2` :



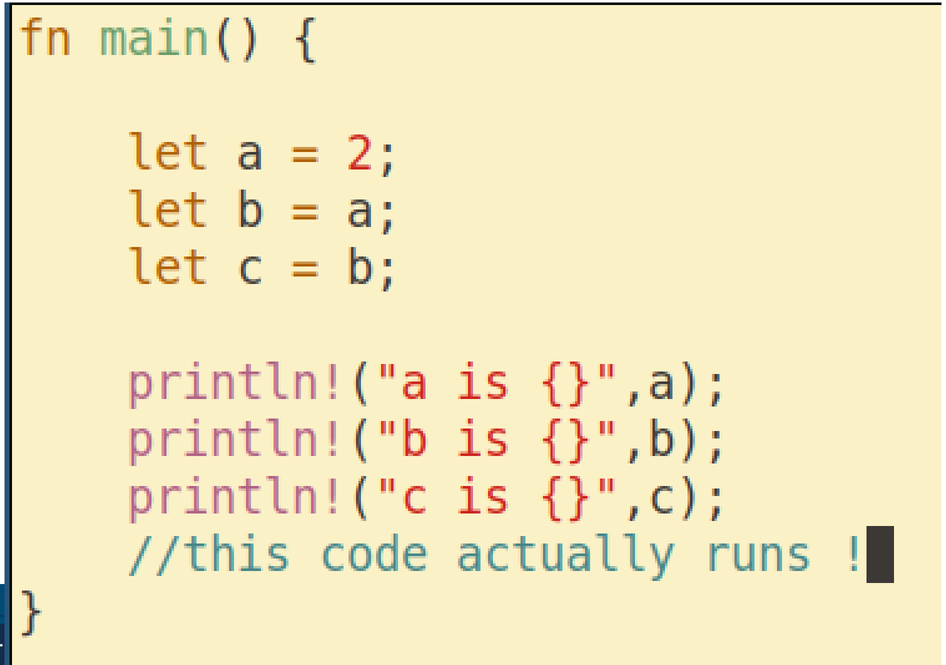
Memory management - Clone



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

    let s2 = String::from(" World");

    let s3 = format!("{}{}", s1, s2);


    println!("{ s3}");

}
```

Memory management - Copy

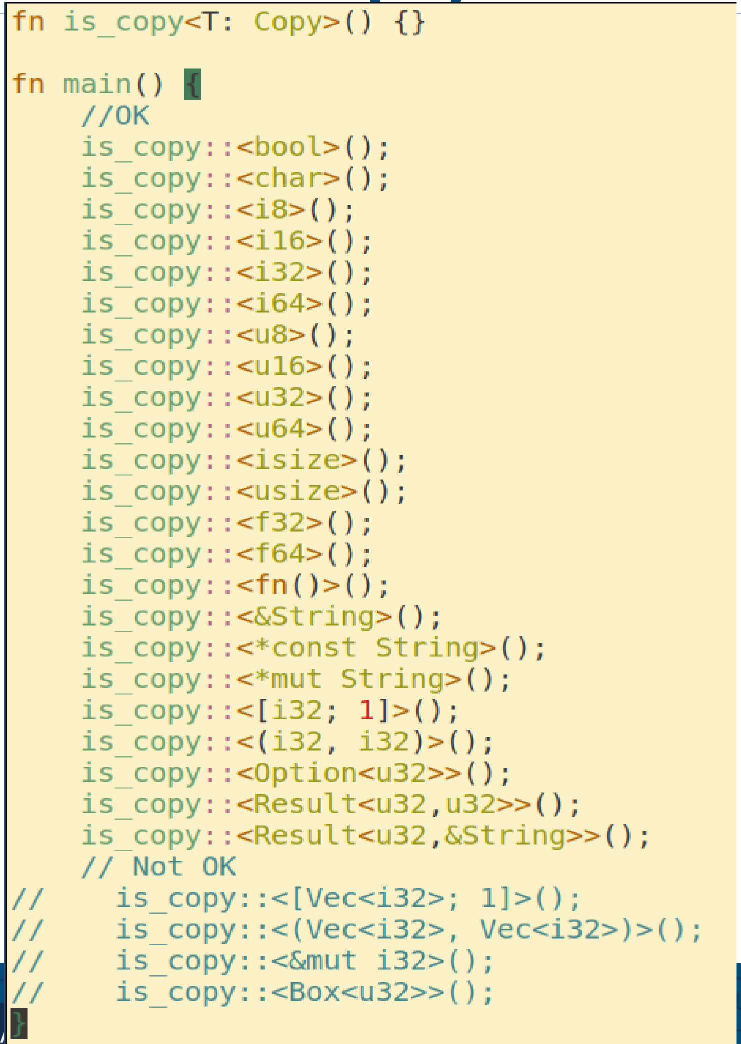
- Some primitive types are copyied by default
- Generalizing the latter case, any type implementing Drop can’t be Copy, because it’s managing some resource besides its own size_of::<T> bytes.
- As usually documentation is good



```
fn main() {

    let a = 2;
    let b = a;
    let c = b;

    println!("a is {}",a);
    println!("b is {}",b);
    println!("c is {}",c);
    //this code actually runs !
}
```

Memory management - Copy

- Quick check through generics



```
fn is_copy<T: Copy>() {}

fn main() {
    //OK
    is_copy::<bool>();
    is_copy::<char>();
    is_copy::<i8>();
    is_copy::<i16>();
    is_copy::<i32>();
    is_copy::<i64>();
    is_copy::<u8>();
    is_copy::<u16>();
    is_copy::<u32>();
    is_copy::<u64>();
    is_copy::<isize>();
    is_copy::<usize>();
    is_copy::<f32>();
    is_copy::<f64>();
    is_copy::<fn()>();
    is_copy::<&String>();
    is_copy::<*const String>();
    is_copy::<*mut String>();
    is_copy::<[i32; 1]>();
    is_copy::<(i32, i32)>();
    is_copy::<Option<u32>>();
    is_copy::<Result<u32,u32>>();
    is_copy::<Result<u32,&String>>();
    // Not OK
//     is_copy::<[Vec<i32>; 1]>();
//     is_copy::<(Vec<i32>, Vec<i32>)>();
//     is_copy::<&mut i32>();
//     is_copy::<Box<u32>>();
}
```

Memory management – Misc insights

- graphs and the observer pattern somewhat inherently require sharing.

Ownership – Further readings

- Ownership controls mutability
- Borrowing in rust when compared to functional programming
- Rust borrowing and ownership
- Recent tries of rust on bettering ownership/borrowing model : Ghost Cell and its associated crate
- Rust aliasing and the relation to borrowing/ownership
- Ownership re explained by brown university (best explanation in my opinion)
- Ante's language trying to explain the shortcomes of rust and how they bypassed th em

Ownership & borrowing – Further readings

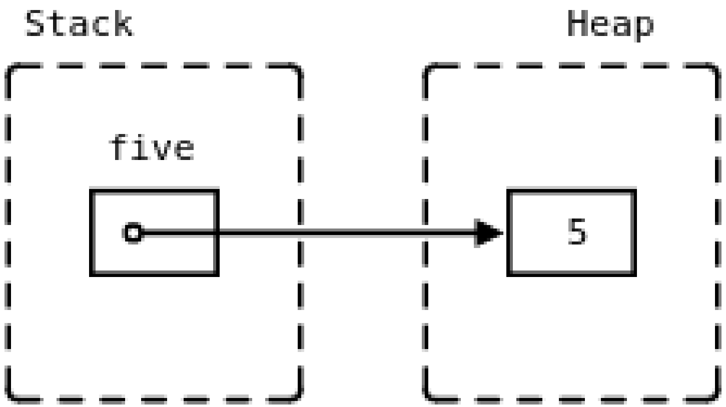
- Nice blog on getting further with borrowing including clone and copy and how to approach it
- Cell and mutability lecture part 12
- Rust memory model behind the nvram prism
- Some advices on how to get your head around rust ownership
- Everything seen summarized
- Hashmap iteration and closures design forced by rust's ownership model

4

Memory management
Smart pointers

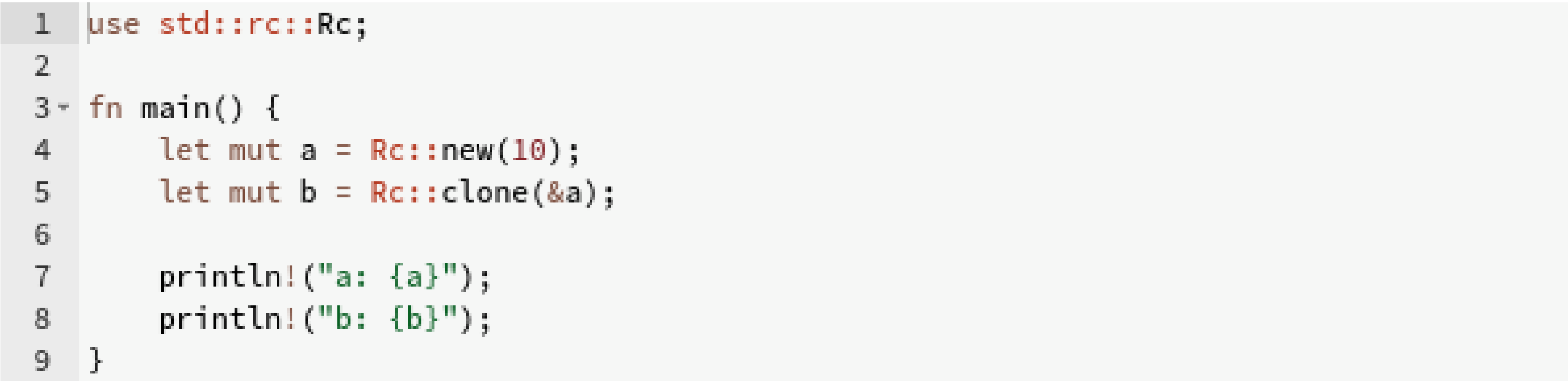
Memory management – Smart pointers

- Box<T> is an owned pointer to data on the heap
- Implement Deref (can use *, method can take Box<Self>, orphan rule does not really apply)

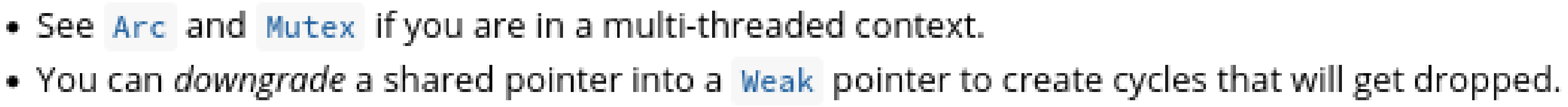


Memory management – Smart pointers

- Rc is a reference counted shared pointer
  - Allows to share ownership
  - Used when runtime can not decide which variable will be the last



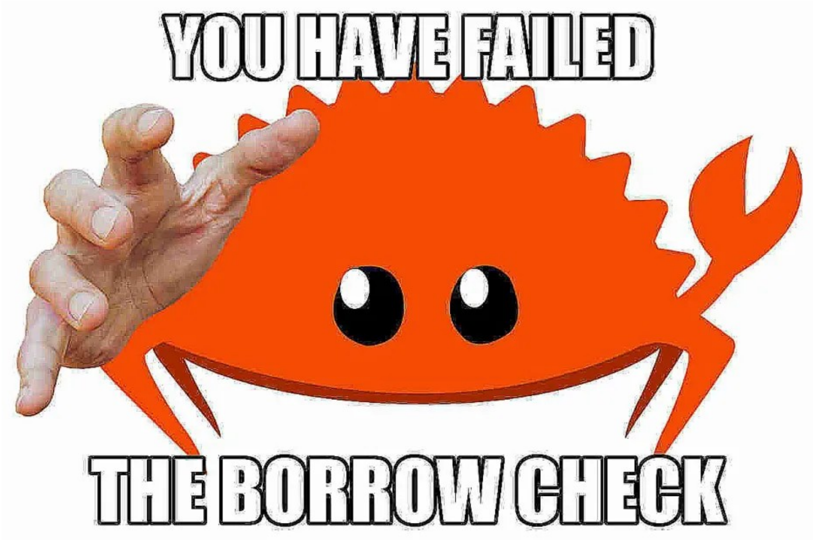
```rust
use std::rc::Rc;

fn main() {
    let mut a = Rc::new(10);
    let mut b = Rc::clone(&a);

    println!("a: {a}");
    println!("b: {b}");
}
```



- See `Arc` and `Mutex` if you are in a multi-threaded context.
- You can *downgrade* a shared pointer into a `Weak` pointer to create cycles that will get dropped.

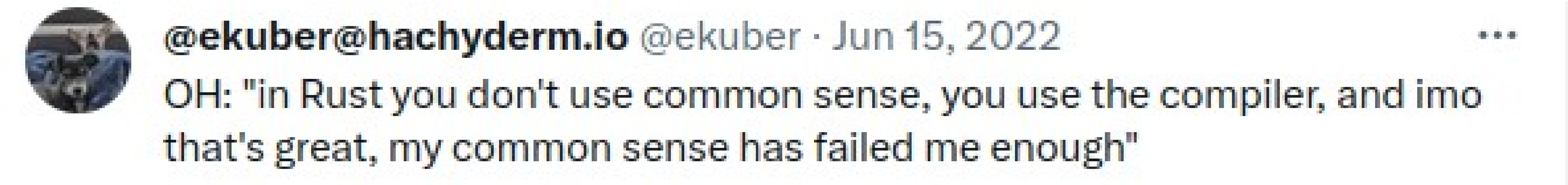
Memory management – Smart pointers

- RefCell
  - Allows breaking borrowing rules
  - Useful for memory safe scenario not allowed by compiler (Mock objects)



Memory management – Smart pointers

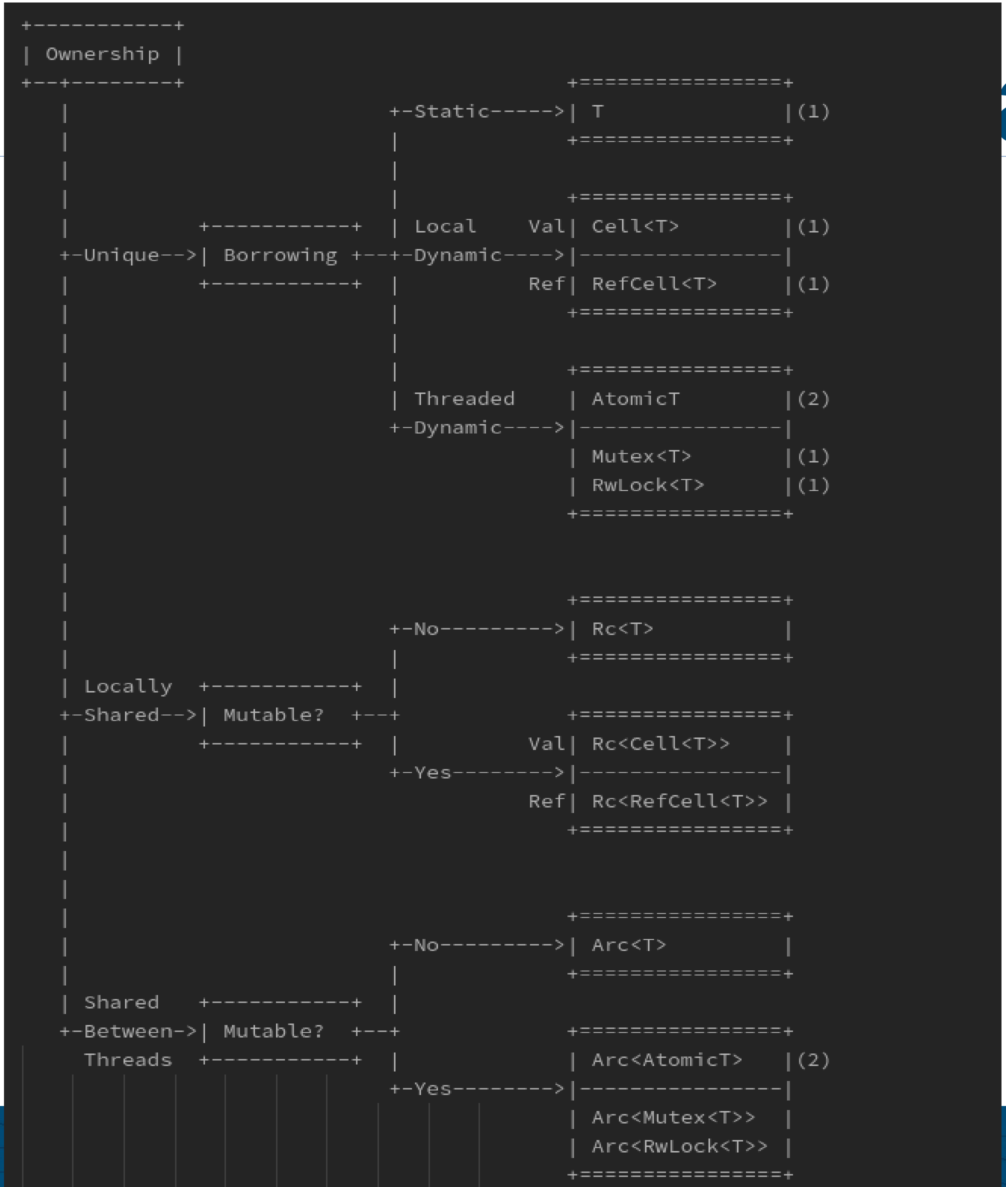
- interior mutability: mutation of values in an immutable context.
- Cell is typically used for simple types, as it requires copying or moving values. More complex interior types typically use RefCell, which tracks shared and exclusive references at runtime and panics if they are misused.



@ekuber@hachyderm.io @ekuber · Jun 15, 2022

OH: "in Rust you don't use common sense, you use the compiler, and imo that's great, my common sense has failed me enough"

Memory art pointer



```
+-----------+
| Ownership |
+--+--------+                                +================+
   |                         +-Static----->| T              |(1)
   |                         |             +================+
   |                         |
   |                         |             +================+
   |          +-----------+  | Local    Val| Cell<T>        |(1)
   +-Unique-->| Borrowing +--+-Dynamic---->|----------------|
   |          +-----------+  |          Ref| RefCell<T>     |(1)
   |                         |             +================+
   |                         |
   |                         |             +================+
   |                         | Threaded    | AtomicT        |(2)
   |                         +-Dynamic---->|----------------|
   |                                       | Mutex<T>       |(1)
   |                                       | RwLock<T>      |(1)
   |                                       +================+
   |
   |
   |                                       +================+
   |                         +-No--------->| Rc<T>          |
   |                         |             +================+
   | Locally  +-----------+  |
   +-Shared-->| Mutable?  +--+             +================+
   |          +-----------+  |          Val| Rc<Cell<T>>    |
   |                         +-Yes-------->|----------------|
   |                                    Ref| Rc<RefCell<T>> |
   |                                       +================+
   |
   |
   |                                       +================+
   |                         +-No--------->| Arc<T>         |
   |                         |             +================+
   | Shared   +-----------+  |
   +-Between->| Mutable?  +--+             +================+
     Threads  +-----------+  |             | Arc<AtomicT>   |(2)
                             +-Yes-------->|----------------|
                                           | Arc<Mutex<T>>  |
                                           | Arc<RwLock<T>> |
                                           +================+
```

Source: https://stackoverflow.com/questions/45674479/need-holistic-explanation-about-rusts-cell-and-reference-counted-types/50696381#50696381

Memory management – Smart pointers

- RC
  - A video about RC

Using RC

- When is it nice to use RC
  - Recursive data structure to avoid redundant stores
  - Too difficult to reason about lifetime at compile time

Memory management – Smart pointers

- Interior mutability explained

Smart pointers

- To methods should be overriden
  - Dereferencing (*ptr, through trait Deref/DerefMut)
    - This should never fail
  - Destruction: ptr goes out of scope (through trait Drop)

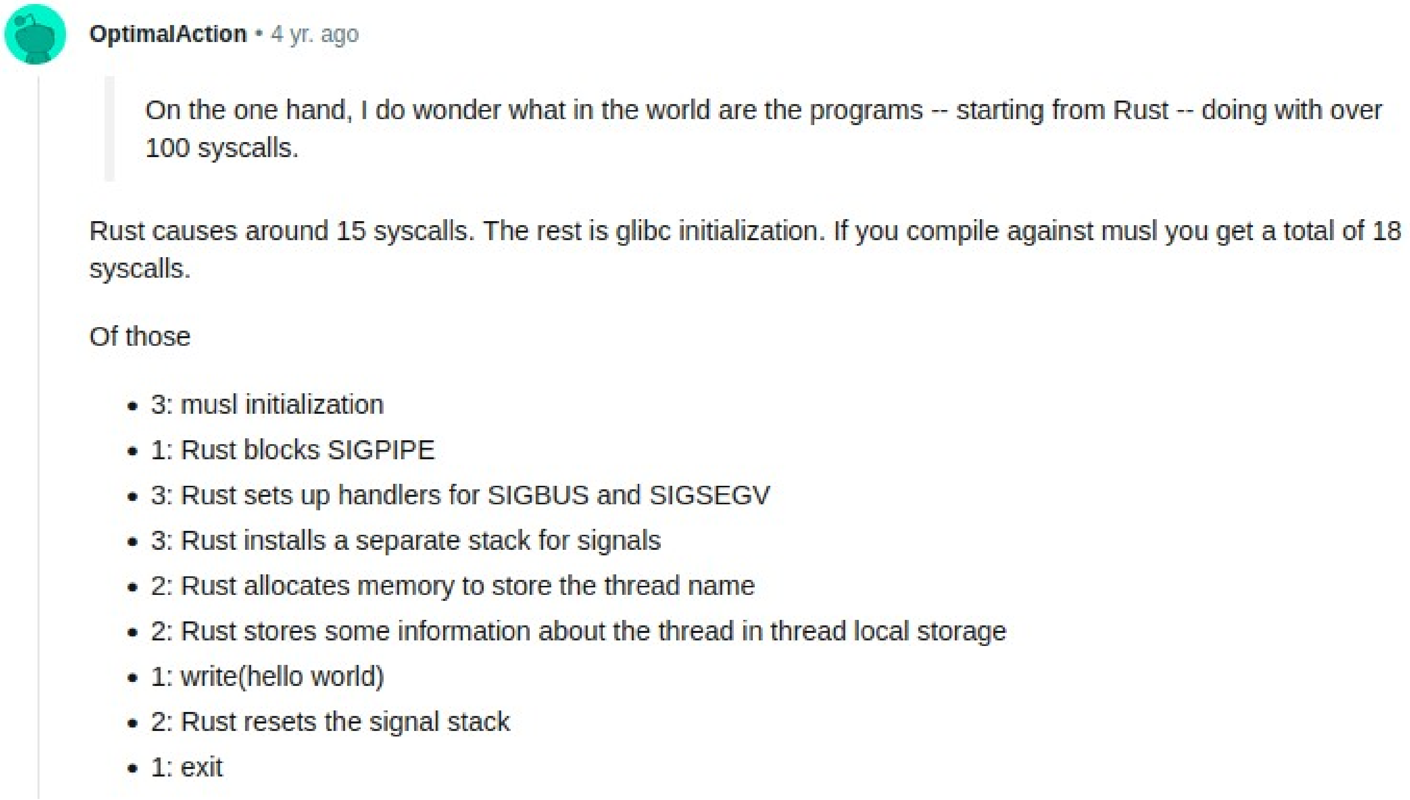
Memory management – Smart pointers

- Combining smart pointers
- Rust smart pointers explanation
- Smart pointer and interior mutability (excellent 2 hour video)
- Rust pointers
- Learning rust with entirely too many linked lists

3

no_std rust

Hello world deep dive



OptimalAction • 4 yr. ago

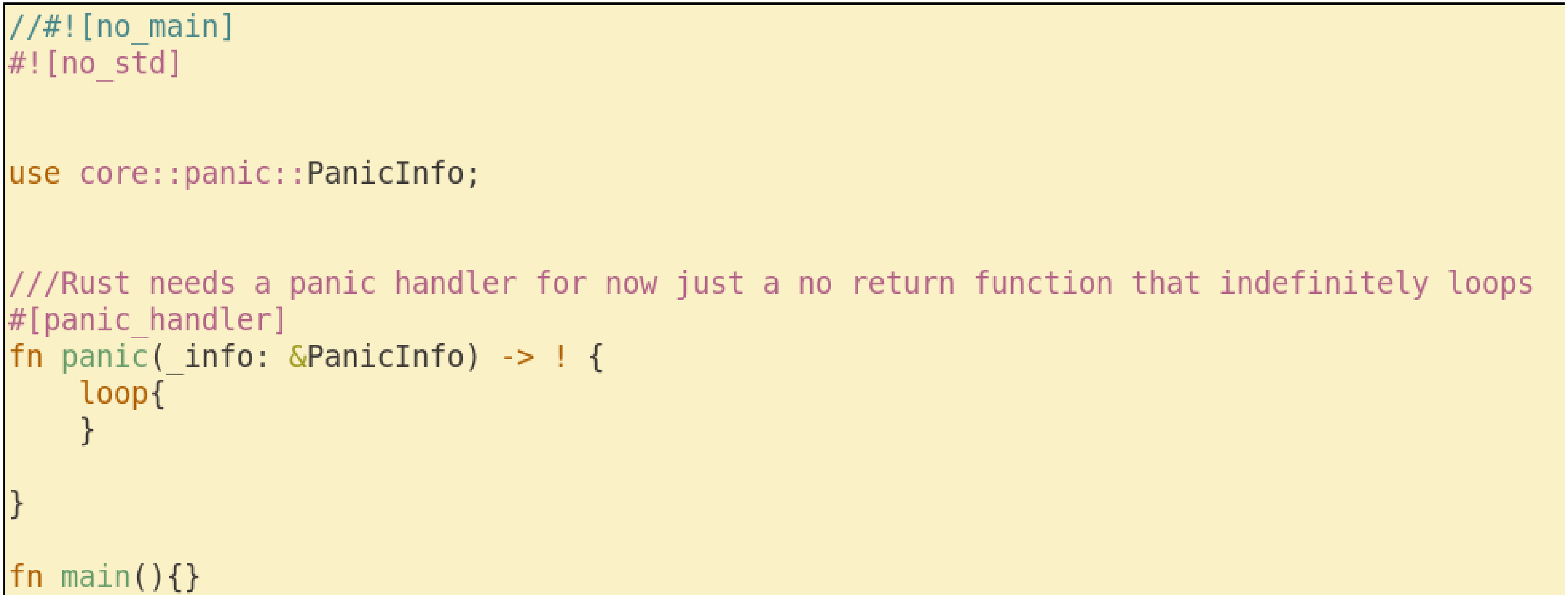
> On the one hand, I do wonder what in the world are the programs -- starting from Rust -- doing with over 100 syscalls.

Rust causes around 15 syscalls. The rest is glibc initialization. If you compile against musl you get a total of 18 syscalls.

Of those

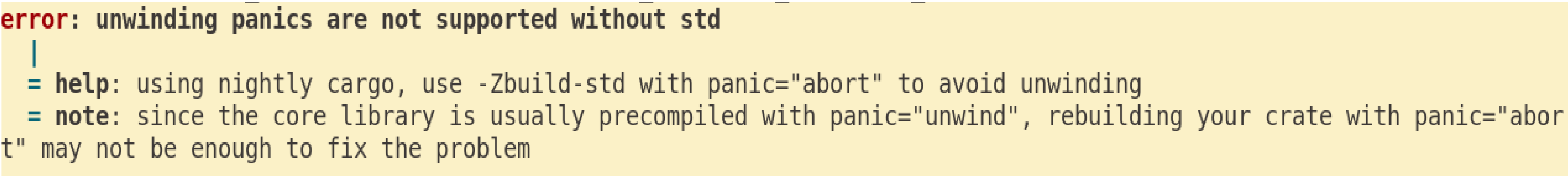
- 3: musl initialization
- 1: Rust blocks SIGPIPE
- 3: Rust sets up handlers for SIGBUS and SIGSEGV
- 3: Rust installs a separate stack for signals
- 2: Rust allocates memory to store the thread name
- 2: Rust stores some information about the thread in thread local storage
- 1: write(hello world)
- 2: Rust resets the signal stack
- 1: exit

no_std rust

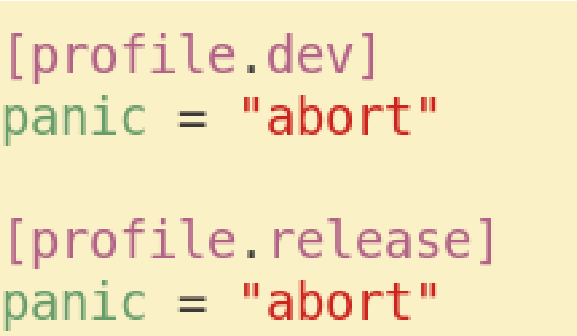


```rust
//#![no_main]
#![no_std]


use core::panic::PanicInfo;


///Rust needs a panic handler for now just a no return function that indefinitely loops
#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop{
    }

}

fn main(){}
```

no_std rust

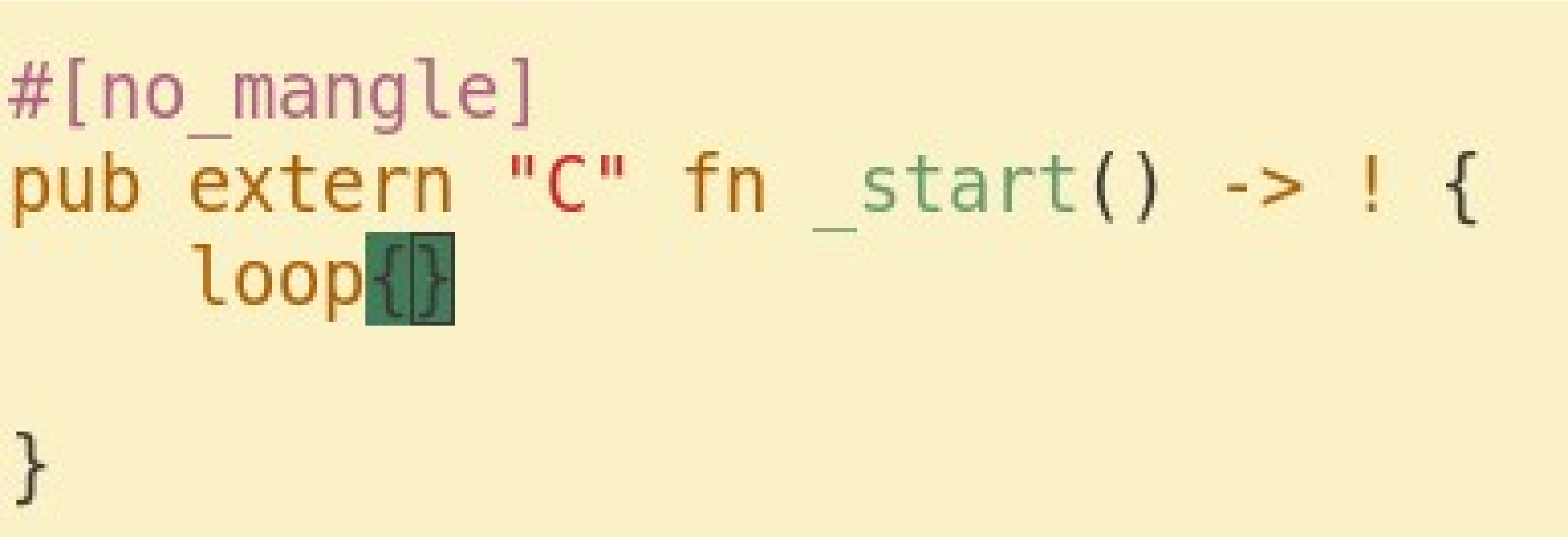


```
error: unwinding panics are not supported without std
  |
  = help: using nightly cargo, use -Zbuild-std with panic="abort" to avoid unwinding
  = note: since the core library is usually precompiled with panic="unwind", rebuilding your crate with panic="abor
t" may not be enough to fix the problem
```



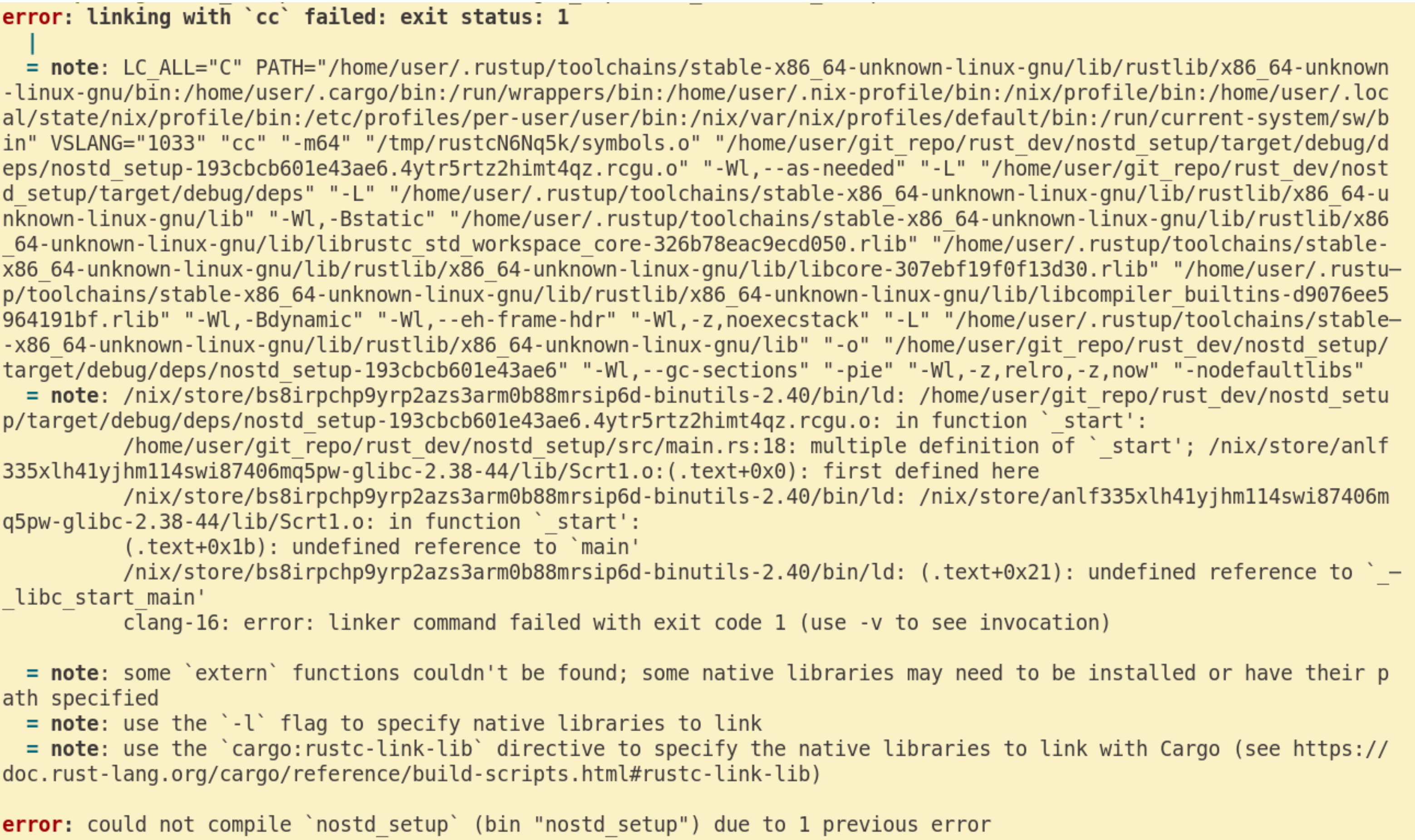
```
[profile.dev]
panic = "abort"

[profile.release]
panic = "abort"
```

no_std rust



```rust
#[no_mangle]
pub extern "C" fn _start() -> ! {
    loop{}

}
```

no_std rust



```
error: linking with `cc` failed: exit status: 1
  |
  = note: LC_ALL="C" PATH="/home/user/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/lib/rustlib/x86_64-unknown
-linux-gnu/bin:/home/user/.cargo/bin:/run/wrappers/bin:/home/user/.nix-profile/bin:/nix/profile/bin:/home/user/.loc
al/state/nix/profile/bin:/etc/profiles/per-user/user/bin:/nix/var/nix/profiles/default/bin:/run/current-system/sw/b
in" VSLANG="1033" "cc" "-m64" "/tmp/rustcN6Nq5k/symbols.o" "/home/user/git_repo/rust_dev/nostd_setup/target/debug/d
eps/nostd_setup-193cbcb601e43ae6.4ytr5rtz2himt4qz.rcgu.o" "-Wl,--as-needed" "-L" "/home/user/git_repo/rust_dev/nost
d_setup/target/debug/deps" "-L" "/home/user/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/lib/rustlib/x86_64-u
nknown-linux-gnu/lib" "-Wl,-Bstatic" "/home/user/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/lib/rustlib/x86
_64-unknown-linux-gnu/lib/librustc_std_workspace_core-326b78eac9ecd050.rlib" "/home/user/.rustup/toolchains/stable-
x86_64-unknown-linux-gnu/lib/rustlib/x86_64-unknown-linux-gnu/lib/libcore-307ebf19f0f13d30.rlib" "/home/user/.rustu-
p/toolchains/stable-x86_64-unknown-linux-gnu/lib/rustlib/x86_64-unknown-linux-gnu/lib/libcompiler_builtins-d9076ee5
964191bf.rlib" "-Wl,-Bdynamic" "-Wl,--eh-frame-hdr" "-Wl,-z,noexecstack" "-L" "/home/user/.rustup/toolchains/stable-
-x86_64-unknown-linux-gnu/lib/rustlib/x86_64-unknown-linux-gnu/lib" "-o" "/home/user/git_repo/rust_dev/nostd_setup/
target/debug/deps/nostd_setup-193cbcb601e43ae6" "-Wl,--gc-sections" "-pie" "-Wl,-z,relro,-z,now" "-nodefaultlibs"
  = note: /nix/store/bs8irpchp9yrp2azs3arm0b88mrsip6d-binutils-2.40/bin/ld: /home/user/git_repo/rust_dev/nostd_setu
p/target/debug/deps/nostd_setup-193cbcb601e43ae6.4ytr5rtz2himt4qz.rcgu.o: in function `_start':
          /home/user/git_repo/rust_dev/nostd_setup/src/main.rs:18: multiple definition of `_start'; /nix/store/anlf
335xlh41yjhm114swi87406mq5pw-glibc-2.38-44/lib/Scrt1.o:(.text+0x0): first defined here
          /nix/store/bs8irpchp9yrp2azs3arm0b88mrsip6d-binutils-2.40/bin/ld: /nix/store/anlf335xlh41yjhm114swi87406m
q5pw-glibc-2.38-44/lib/Scrt1.o: in function `_start':
          (.text+0x1b): undefined reference to `main'
          /nix/store/bs8irpchp9yrp2azs3arm0b88mrsip6d-binutils-2.40/bin/ld: (.text+0x21): undefined reference to `_-
_libc_start_main'
          clang-16: error: linker command failed with exit code 1 (use -v to see invocation)

  = note: some `extern` functions couldn't be found; some native libraries may need to be installed or have their p
ath specified
  = note: use the `-l` flag to specify native libraries to link
  = note: use the `cargo:rustc-link-lib` directive to specify the native libraries to link with Cargo (see https://
doc.rust-lang.org/cargo/reference/build-scripts.html#rustc-link-lib)

error: could not compile `nostd_setup` (bin "nostd_setup") due to 1 previous error
```

no_std rust

- rustc --print target-list
  - Cargo build –target ${target_name}
- cargo rustc -- -C link-arg=-nostartfiles (linux specific)

no_std rust

- Starting files can be found at
  - github.com/Mbahal/didnotpublishyet_need_to_do_it
- Some cumbersome work is in .cargo/config.toml – won’t be explained

Repr and C like structures

- The repr attribute
  - Transparent
  - Packed
  - C

no_std rust

- Volatile
  - https://docs.rust-embedded.org/book/c-tips/index.html#volatile-access
    A nice notion to know although you seldom need it

9

Unsafe rust

Unsafe rust

- Unsafe Rust: can trigger undefined behavior if preconditions are violated.
- You become the anchor of trust
  - YOU are the compiler.

Unsafe rust

- Concept of soundness (can not cause undefined behaviour from safe rust when calling code)
- It allows to :
  - Dereference raw pointers.
  - Access or modify mutable static variables.
  - Access union fields.
  - Call unsafe functions, including extern functions.
  - Implement unsafe traits.

Unsafe rust

- There should be a # Safety section on the Rustdoc for the trait explaining the requirements for the trait to be safely implemented.
- The built-in Send and Sync traits are unsafe.
- Working with strings in ffi is usually unsafe
- In general ffi is unsafed

Unsafe rust

- Out of bond accesses
- Use after free/double free
- Out of bounds pointer arithmetic
- Insufficient alignment
- Invalid values
- Violation of reference aliasing rules
- Data race
- Those are all soundness bug in the beginning

Ecosystem tooling for no_std

- Miri or mirai are apparently not that good with rust no_std

Unsafe – useful links

- Useful links
  - https://doc.rust-lang.org/nomicon/ffi.html
  - https://doc.rust-lang.org/std/ffi/

Unsafe – further readings

- Unsafe rust and miri
- Unsafe rust is not C

5

exercise

Exercise – The heap

- Comeback of 3rd year bonus exercise
  - In Rust no_std we can not use allocators
  - Except if we create one
    - https://bd103.github.io/blog/2023-06-27-global-allocators/
    - As usual it goes with a trait
    - Your goal, implementing a global allocator, and design it in no_std so that you can use it in your kernel next time

Exercise – global alloc

- It’s usually a good thing to keep crates no_std compatible
  - https://www.lurklurk.org/effective-rust/no-std.html
  - https://gist.github.com/tdelabro/b2d1f2a0f94ceba72b718b92f9a7ad7b
  - https://siliconislandblog.wordpress.com/2022/04/24/writing-a-no_std-compatible-crate-in-rust/
  - https://blog.dbrgn.ch/2019/12/24/testing-for-no-std-compatibility/
  - https://github.com/hobofan/cargo-nono
  -

Coding with features - bonus

- So one thing I would like is to keep the global alloc behind a feature
  - https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/conditional-compilation.html
  - https://betterprogramming.pub/compile-time-feature-flags-in-rust-why-how-when-129aada7d1b3?gi=dafd57e2f7c0

exam

- Git repo, add me as contributor
- The project is in **no_std**
- Commit are looked at, do not commit everything in one time, else it’s considered cheating
- If you take code from somewhere **it has to be credited**,else considered cheating as well
- Code quality (miri/mirai/fuzzers, other cargo utils ….)
- Unsafe must be thoroughly documented using rustdoc safety part
- Comment your code and use rust doc, code exemple are appreciated for the allocation library
- A report with your design choice is needed
- Due date : 26/11/2024 23h59

exam

- If you have time (it’s adviced to do so)
  - In the second exam you’ll have to implement a FAT32 filesystem
  - You can start implementing a no_std compatible FAT32 parser
  - Won’t be taken in account for THIS exam, but will help you go faster for part II