RustOS - A Custom Operating System in Rust

Konstantin Unterweger, Köppl Michael, Loidolt Lukas, Pritzl Johannes

RustOS - Overview

- no_std environment (no standard library)
- Bare-metal x86-64 OS development
- Rust and WebAssembly
- Built from scratch following OS development principles

Requirements That Demand Low-Level Control

Key challenges requiring low-level programming:

- Direct hardware access (keyboard, memory, interrupts)
- No operating system underneath - RustOS IS the OS
- Manual memory management without allocator
- Interrupt handling at CPU level
- Performance-critical operations

CPU Segmentation Setup

```
pub fn initialize_global_descriptor_table() {
    GLOBAL_DESCRIPTOR_CONTEXT.gdt.load();
    unsafe {
        CS::set_reg(GLOBAL_DESCRIPTOR_CONTEXT.kernel_code);
        SS::set_reg(GLOBAL_DESCRIPTOR_CONTEXT.kernel_data);
        DS::set_reg(GLOBAL_DESCRIPTOR_CONTEXT.kernel_data);
        ES::set_reg(GLOBAL_DESCRIPTOR_CONTEXT.kernel_data);
        load_tss(GLOBAL_DESCRIPTOR_CONTEXT.task_state);
    }
}
```

Why low-level?

- Direct CPU segment register manipulation
- Required for x86-64 protected mode
- Task State Segment (TSS) configuration

Reading Keyboard Input - The Hardware Way

```
extern "x86-interrupt" fn keyboard_interrupt_handler(_stack_frame: InterruptStackFrame) {
    let mut ps2_port: Port<u8> = Port::new(0x60);
    let scancode = unsafe { ps2_port.read() };

    add_scancode(scancode);

    unsafe {
        PICS.lock()
            .notify_end_of_interrupt(u8::from(InterruptIndex::Keyboard));
    }
}
```

Why low-level?

- Direct port I/O (0x60 = PS/2 keyboard port)
- Custom interrupt descriptor table (IDT)
- Manual interrupt acknowledgment to PIC
- unsafe blocks for hardware access

Handling Catastrophic Errors

```
lazy_static! {
    static ref TASK_STATE_SEGMENT: TaskStateSegment = {
        let mut task_state_segment = TaskStateSegment::new();
        task_state_segment.interrupt_stack_table[DOUBLE_FAULT_IST_INDEX as usize] = {
            const STACK_SIZE: usize = 4096 * 5;
            static mut STACK: [u8; STACK_SIZE] = [0; STACK_SIZE];
            let stack_start = VirtAddr::from_ptr(&raw const STACK);
            stack_start + STACK_SIZE as u64
        };
        task_state_segment
    };
}
```

Handling Catastrophic Errors

Why low-level?

- Custom interrupt stack table (IST)
- Manual stack allocation for fault handlers
- Prevents infinite crashes from stack overflow

Building Our Own Heap

```
pub fn init_heap(mapper: &mut impl Mapper<Size4KiB>, frame_allocator: &mut impl
FrameAllocator<Size4KiB>) -> Result<(), MapToError<Size4KiB>> {
    ...

    for page in page_range {
        let frame = frame_allocator.allocate_frame().ok_or(MapToError::FrameAllocationFailed)?;
        let flags = PageTableFlags::PRESENT | PageTableFlags::WRITABLE;
        unsafe { mapper.map_to(page, frame, flags, frame_allocator)?.flush() };
    }

    unsafe {
        ALLOCATOR.lock().init(HEAP_START as *mut u8, HEAP_SIZE);
    }

    Ok(())
}
```

Building Our Own Heap

Why low-level?

- Manual page table manipulation
- Virtual to physical memory mapping
- No malloc/free - we implement it!

Custom Async Executor (Shell Task)

```
impl Stream for ScanCodeStream {
    type Item = u8;
    fn poll_next(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Option<u8>> {
        let queue = SCANCODE_QUEUE.try_get().expect("scancode queue not initialized");

        if let Some(scancode) = queue.pop() { return Poll::Ready(Some(scancode)); }

        WAKER.register(cx.waker());
        match queue.pop() {
            Some(scancode) => {
                WAKER.take();
                Poll::Ready(Some(scancode))
            }
            None => Poll::Pending,
        }
    }
}
```

Custom Async Executor (Shell Task)

Why low-level?

- Custom async runtime (no tokio!)
- Manual waker implementation
- Lock-free queue for interrupt → task communication

Running WASM without an OS

```
pub fn init_wasm_game(wasm_bytes: &'static [u8]) {
    let engine = Engine::default();
    let module = Module::new(&engine, wasm_bytes).expect("Failed to parse WASM module");
    let mut linker = Linker::new(&engine);
    linker
        .func_wrap("env", "put_pixel",
            |_caller: Caller<T>, x: i32, y: i32, r: i32, g: i32, b: i32| {
                let color = Rgb {
                    r: r as u8,
                    g: g as u8,
                    b: b as u8,
                };
                framebuffer::put_pixel(x as usize, y as usize, color);
            },
        )
        .unwrap();
}
```

Running WASM without an OS

Why low-level?

- WASM interpreter in no_std environment
- Custom host function bindings
- Manual memory bridge between Rust and WASM

Filesystem in Memory

```
pub fn init_filesystem(ramdisk: &'static [u8]) -> Result<()> {
    let fs = FileSystem::from_tar(ramdisk.into())?;
    FILE_SYSTEM.init_once(|| Mutex::new(fs));
    Ok(())
}
```

Why low-level?

- Parse format manually
- No disk driver - ramdisk only
- Custom error handling without std::io

Testing Without a Test Framework

Cannot use the `test` crate since no std!

Still wanted to make use of Rust's test capabilities!

1. Compile each test as a bootable kernel
2. QEMU boots the test kernel (headless mode)
3. Test runner executes all #[test_case] functions
4. Results printed via serial port (not screen!)
5. QEMU exits with success/failure code
6. Build system interprets exit code

The Complete Picture

Shell (Async Task)
Custom Executor
Scancode Stream (Queue)
Keyboard Interrupt Handler
IDT + GDT + TSS
Memory Manager + Allocator
Bootloader

How we benefited from Rust

1. **Hardware Control**: Direct PS/2 port access, interrupt handling
2. **No Runtime**: No std lib, custom allocator, manual memory management
3. **Performance**: Zero-cost abstractions, no GC pauses
4. **Safety**: Rust's type system prevents common bugs
5. **Complete Control**: We decide EVERYTHING
6. **Testing**: Unit test capabilities built-in

Result: A functioning OS with shell, filesystem, and even WASM games!