Software Systems

Lecture 3

Jonathan Dönszelmann
Vivian Roest

Delft University of Technology, The Netherlands

January 24, 2024

Previous Lecture

- Profiling
- Benchmarking
- Optimizing programs for speed

Today

Writing Code without an Operating System or Standard Library (`std`)

Operating System Services

Question:

What services does an operating system provide to a program?

Operating System Services

- Filesystem
- Threads and Subprocesses
- An Allocator
- Peripherals
- Graphics
- Standard in/output
- Program startup routine (`crt0/1`)
- ... and more.

Program Startup

```
extern crate std;
extern crate alloc;
extern crate core;

use std::prelude::*;

fn __start() {
    startup();
    main();
    teardown();
}

// rest of code
```

https://github.com/runtimejs/musl-libc/blob/master/crt/x86_64/crt1.s https://github.com/runtimejs/musl-libc/blob/0a11d7cb13e243782da36e2e5747b8b151933cca/src/env/__libc_start_main.c#L58

no_std

Opt-out of using Rust's standard library by writing:

`main.rs`

```rust
#![no_std]

// rest of main
```

`core` and `alloc`

- `core`: set of functions that *do not* rely on an Operating System
- `alloc`: set of functions that rely on the presence of an Allocator
    - e.g. `Box`, `Vec`, `Rc`
- `std`: set of functions that *do* require an Operating System

`core` and `alloc`

- `core`: set of functions that *do not* rely on an Operating System
- `alloc`: set of functions that rely on the presence of an Allocator
  - e.g. `Box`, `Vec`, `Rc`
- `std`: set of functions that *do* require an Operating System

Question:

Can we use `no_std` **with** an operating system?

no_std

Question:

What do we do if we don't have an operating system?

- How does the program start? Remember the NES.
- What do we do once it did start?

Don't panic!

Question:

What (should) happen if a bare-metal[1] program crashes?

[1] i.e. running without an operating system

Don't panic!

There are only a few things we can do

- Reboot
- Shut down completely
- Halt / Infinite Loop
- Report error using semihosting[2]

```
#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    loop {} //infinte loop
}
```

[2] Only sometimes available: e.g. when using a debugger or virtualization

Allocating memory

Question:

How do we manage memory without an operating system?

Specifying an allocator

```rust
#![no_std]

// enable the alloc crate. Only works if an allocator is provided.
extern crate alloc;

// import some allocator
use some_allocator::SomeAllocator;

// define global allocator
#[global_allocator]
static ALLOCATOR: SomeAllocator = SomeAllocator::new();


// This function is called when an allocation fails.
#[alloc_error_handler]
fn alloc_error(layout: Layout) -> ! {
    panic!("Alloc error! {layout:?}");
}
```

Hardware Abstraction

Many bare-metal programs have similar components

- Booting up
- Interrupt Handlers
- Common Portocols
  - I^2C
  - CAN
  - SPI
  - ...

Traditionally, manufacturers also provide C code to support programming for that device.

The Rust ecosystem has two kind of things that can help

- Platform Abstraction Crates (PACs)
- Hardware Abstraction Layers (HALs)

Aside: Memory Mapped I/O (MMIO)

- Special memory addresses that don't store data but talk to a peripheral device
- Useful to provide a (somewhat) standardized way to talk to peripherals
- However, a lot of "magic" constants to define and use correctly
- Can we make this easier?

Those that followed Software Fundamentals might recognise this from the NES joypad.

Peripheral Access Crates

The datasheet table for the nRF51 temperature sensor

Table 181: Instances

Base address	Peripheral	Instance	Description
0x4000C000	TEMP	TEMP	Temperature Sensor

Table 182: Register Overview

Register	Offset	Description
Tasks		
START	0x000	Start temperature measurement
STOP	0x004	Stop temperature measurement
Events		
DATARDY	0x100	Temperature measurement complete, data ready
Registers		
INTEN	0x300	Enable or disable interrupt
INTENSET	0x304	Enable interrupt
INTENCLR	0x308	Disable interrupt
TEMP	0x508	Temperature in °C

Peripheral Access Crates

The Rust struct for the nRF51 temperature sensor from `nrf51-pac`[3]

```rust
#[repr(C)]
pub struct RegisterBlock {
    pub tasks_start: TASKS_START,
    pub tasks_stop: TASKS_STOP,
    pub events_datardy: EVENTS_DATARDY,
    pub intenset: INTENSET,
    pub intenclr: INTENCLR,
    pub temp: TEMP,
    /* private fields (for alignment) */
}
```

Tasks	
START	0x000
STOP	0x004
Events	
DATARDY	0x100
Registers	
INTEN	0x300
INTENSET	0x304
INTENCLR	0x308
TEMP	0x508

notice how it matches the datasheet 1:1?

[3] https://docs.rs/nrf51-pac/latest/nrf51_pac/temp/struct.RegisterBlock.html

Peripheral Access Crates: working with peripherals

```rust
let mut p = nrf51_hal::pac::Peripherals::take().unwrap();

// take reference since peripheral is owner
let t = &mut p.TEMP;

// write the bits 101 to this register
t.tasks_start.write(|w| unsafe {w.bits(5)})

// reset the value (safe)
t.tasks_start.write(|w| w.reset_value())

// read the temperature (safe)
t.temp.read(|r| r.bits())
```

- writing is unsafe
- resetting and reading is safe

Peripheral Access Crates: data validation

- Some registers can only contain certain values
- Writing other values is unsafe
- Using the type system for data validation

```
let mut p = nrf51_hal::pac::Peripherals::take().unwrap();

// take reference since peripheral is owner
let u = &mut p.UART;

// safe!
u.write(|w| w.baudrate().baud1200());
```

- Using baudrate, we can only write “valid” values

Hardware abstraction layers

- Built on top of PACs
- Higher level
- Usually much better documented
- “Create your own operating system”

```rust
use nrf_hal::temp::Temp;

let mut p = nrf51_hal::pac::Peripherals::take().unwrap();

let mut t = Temp::new(p.TEMP);

// don't bother with the individual registers
let temperature = t.measure();
```

Hardware abstraction layers

Some examples:

- abstract over device: `nrf_usbd`
- abstract over chip: `nrf51_hal`
- abstract over whole architecture: `cortex_m`

Cortex_m

- Abstracts details of ARM Cortex M processors
- CPU registers, interrupts, standard peripherals
- extensions:
  - `alloc_cortex_m`: an allocator
  - `cortex_m_rt`: a startup runtime
  - `cortex_m_interrupt`: easy interrupt setup
  - `cortex_m_semihosting`: communication with hypervisor[4]

With HALs and PACs, embedded development starts to feel like programming **with** an operating system

[4] or even hardware debugger

Unsafe code

Definition

Safe: Rust can check it

Sound: Rust can't check it but it does work!

- Soundness may depend on assumptions
  - A pointer is not NULL
  - An index is in bounds
- If assumptions are checked, unsafe operations may be *always* sound

```rust
// unsound when a is null
unsafe fn dereference(a: *mut usize) -> usize {
    *a
}
```

Unsafe code

Definition

Safe: Rust can check it

Sound: Rust can't check it but it does work!

- Safety does not mean 'can't crash'
- Safety means no undefined behavior

```
// unsound when a is null
unsafe fn dereference(a: *mut usize) -> usize {
    *a
}

// is this the only assumption?
fn safe_dereference(a: *mut usize) -> usize {
    assert_ne!(a as usize, 0);
    dereference(a)
}
```

Safe abstractions

- Some things are safe to do but the compiler won't allow us
- Certain things are only safe if certain conditions are met

A safe abstraction is a way to do traditionally unsafe things through an always-safe interface.

Check our assumptions if we cannot guarantee them!

Question:

Have we seen any safe abstractions in this course?

Safe abstractions

Rust is built on unsafe abstractions

- `Box` abstracts memory management
- `Mutex` abstracts shared mutability
- `Vec` abstracts growing and shrinking memory
- `println!` and `File` abstracts system calls
- `std::io::Error` abstracts errno
- `Channel` abstracts communication
- … the list goes on

Important

Document why unsafe code is safe! Write down your assumptions!

Safe abstractions

Safe abstractions often rely on certain assumptions about a system

Question:

Can you think of operations that are safe on some systems and unsafe on others?

Safe abstractions: A Mutex for a single core machine

- Only one core: no locking required!
- Or is it?

Safe abstractions: A Mutex for a single core machine

- Only one core: no locking required!
- Or is it?

Pseudocode:

```rust
struct OurMutex<T>(UnsafeCell<T>);

impl<T> OurMutex<T> {
    fn update(v: impl FnOnce(&mut T)) {
        // turn off interrupts
        v(self.0.get())
        // turn interrupts back on if they were on before
    }
}
```

Turning off interrupts makes shared mutability safe on single core machines! But check (and comment) if we can make such assumptions!

Assignment

- Design and create a UART driver (from template)
- Design a protocol to communicate with the microcontroller
- Serializing and Deserializing messages
- Detect transmission errors
- Drive a simulated step counter

All in an emulator

Advent of Code

- 1st of December until 25th of December
- Starting Friday!
- Every day a programming puzzle
- A bit harder every day
- First to solve ⟶ most points
- Leaderboard: `adventofcode.com` `356604-2d88ced0`[5]

[5] also on the course website