Building real-time applications in Rust with OxidOS

Alexandru Radovici, Cristian Rusu

Building real-time applications in Rust with OxidOS

- Embedded Operating Systems
- OxidOS (Tock)
- Build a Tock Rust Application
- Build OxidOS Drivers
- Q&A

Operating System

Operating System

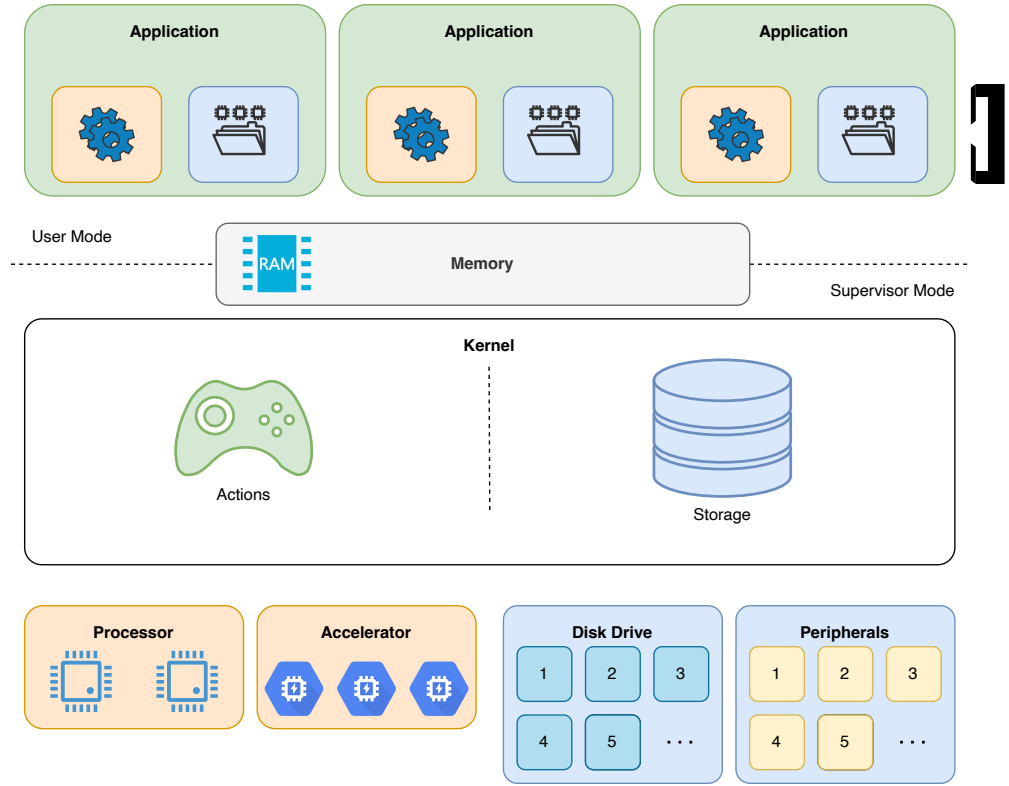
the main role

Allow Portability

- provides a hardware independent API
- applications should run on any hardware

Resources Management and Isolation

- allow applications to access resources
- prevent applications from accessing hardware directly
- isolate applications



Desktop and Server Operating Systems

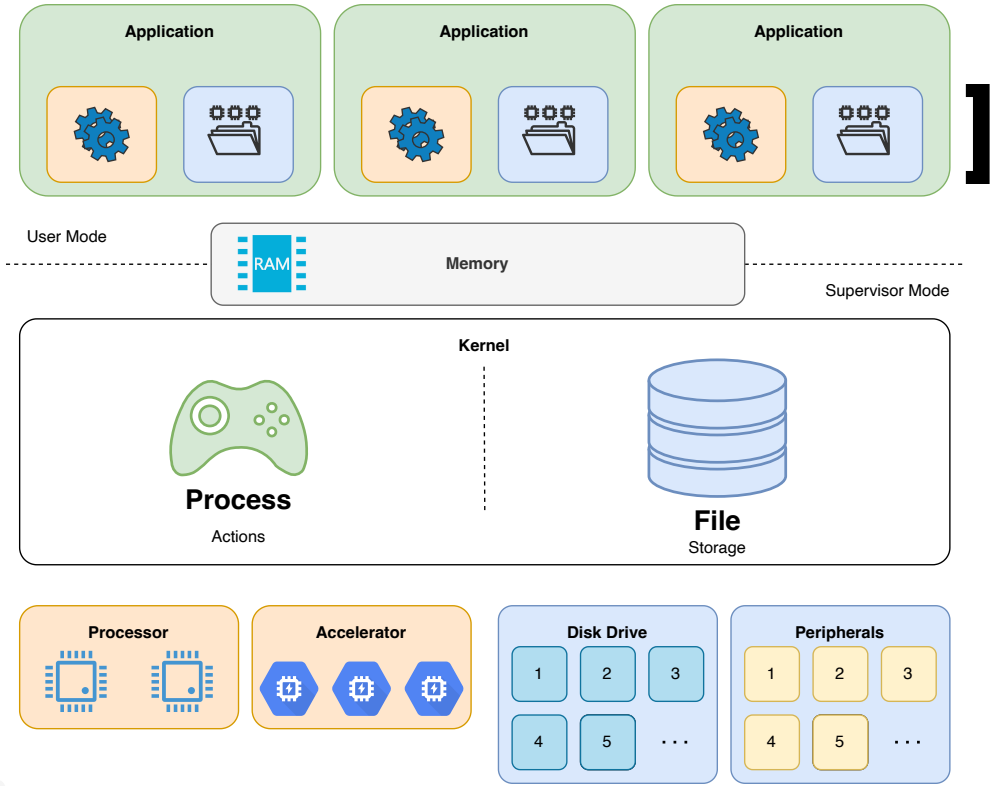
abstractions

Actions

- **process** and **threads**
- use the *Processor* and *Accelerators* (GPU, Neural Engine, etc)

Data

- everything is a file
- peripherals are viewed as files (*POSIX*)
  - `/sys/class/gpio/gpio5/direction`
  - `/sys/class/gpio/gpio5/value`



Embedded Operating Systems

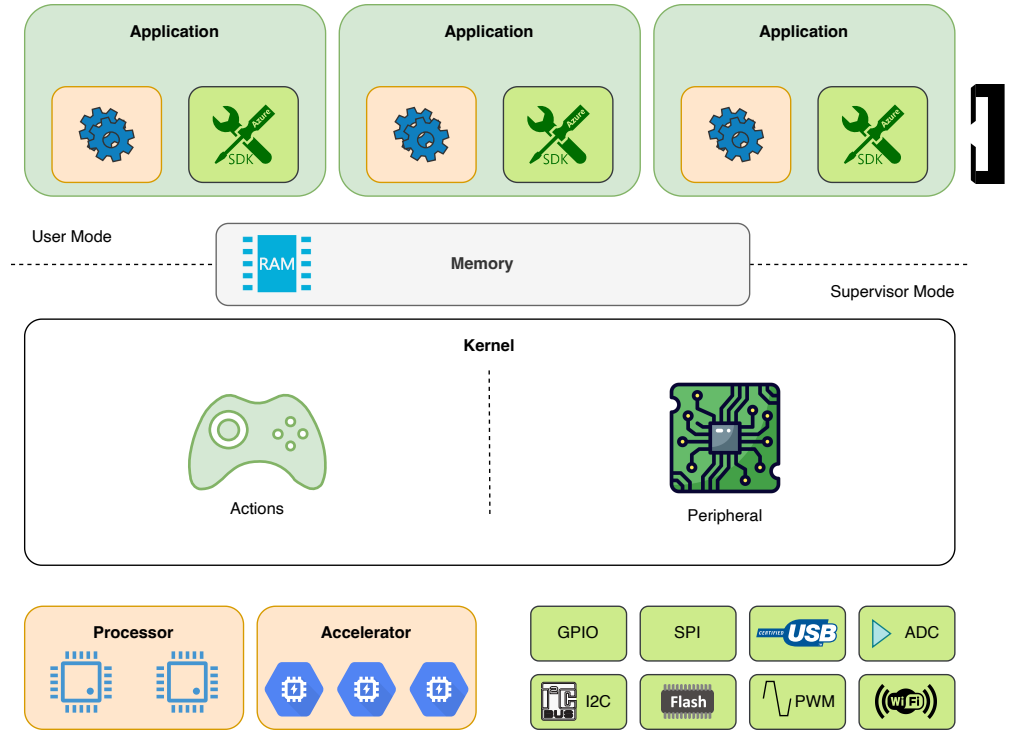
Actions

- **process** or **threads**
- use the *Processor* and *Accelerators* (Crypto Engines, Neural Engine, etc)

Peripheral

- provide a hardware independent API
- prevent processes from accessing the peripheral

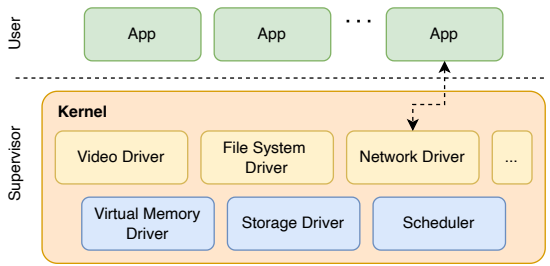
usually the applications and the kernel are compiled together into a **single binary**



Kernel Types

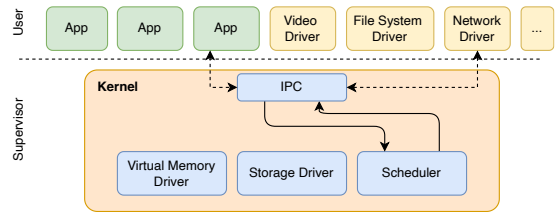
from the **kernel and drivers** point of view

Monolothic



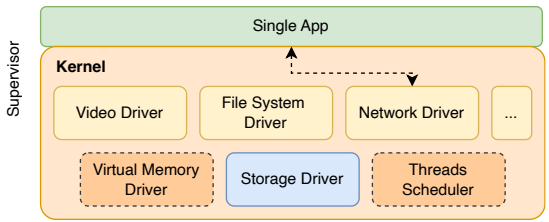
- all drivers in the kernel
- Windows, Linux, MacOS

Microkernel



- all drivers are applications
- Minix

Unikernel



- the kernel is bundled with all the drivers and one single application
- Unikraft/Linux
- Most of the microcontroller RTOSes

System Call

the OS API

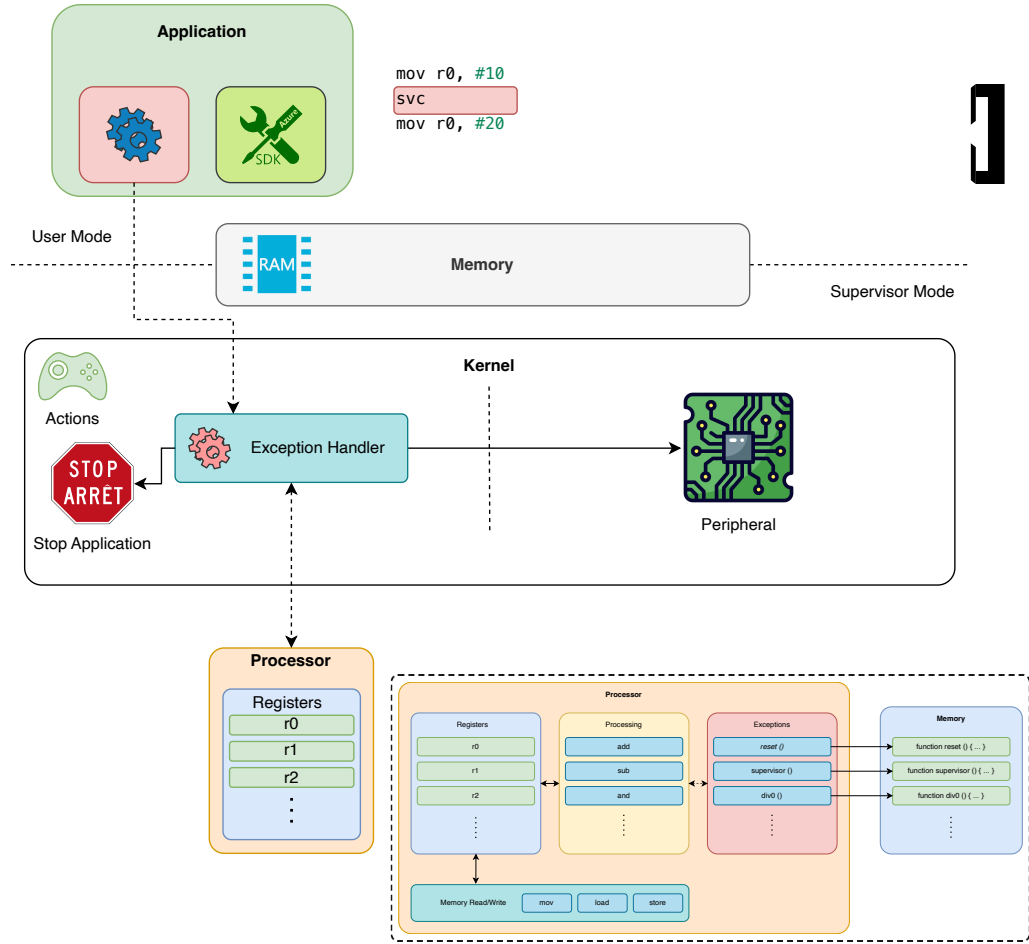
accessing a peripheral can be **performed** only **by the OS**

The application:

1. puts values in the registers
2. triggers an exception
   - `svc` instruction for ARM

The OS:

1. looks at the registers and determines what the required action is
2. performs the action
3. puts the return values into the registers



OxidOS OS

based on Tock

Bibliography

for this section

Alexandru Radovici, Ioana Culic, *Getting Started with Secure Embedded Systems*

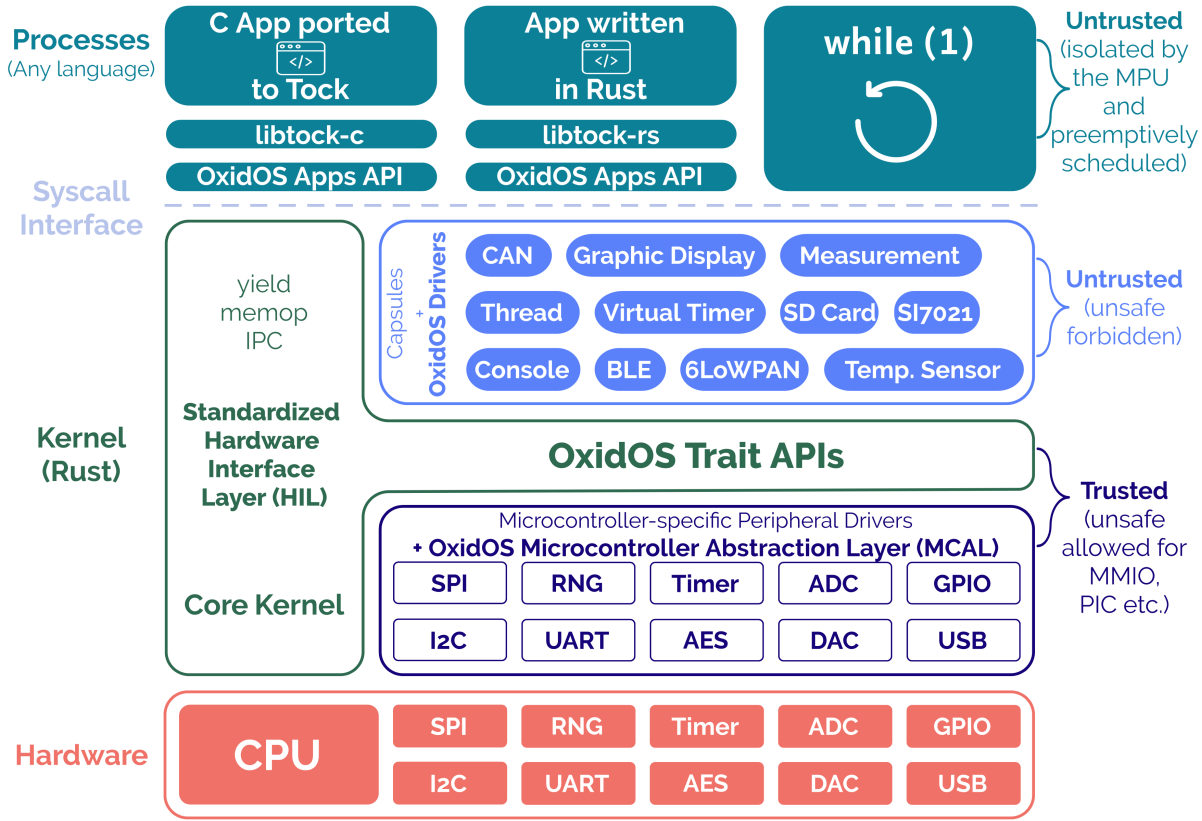
- Chapter 3 - *The Tock system architecture*

Tock OS

an embedded operating systems that works like a desktop or server one

- A **preemptive** embedded OS (runs on MCUs)
  - Cortex-M
  - RISC-V
- Uses memory protection (**MPU required**)
- Has separate **kernel and user space**
  - most embedded OS have the one piece software philosophy
- Runs untrusted apps in user space
- **Hybrid** architecture
- Kernel (and drivers) written in Rust
- Apps written in C/C++ or Rust (any language that can be compiled)

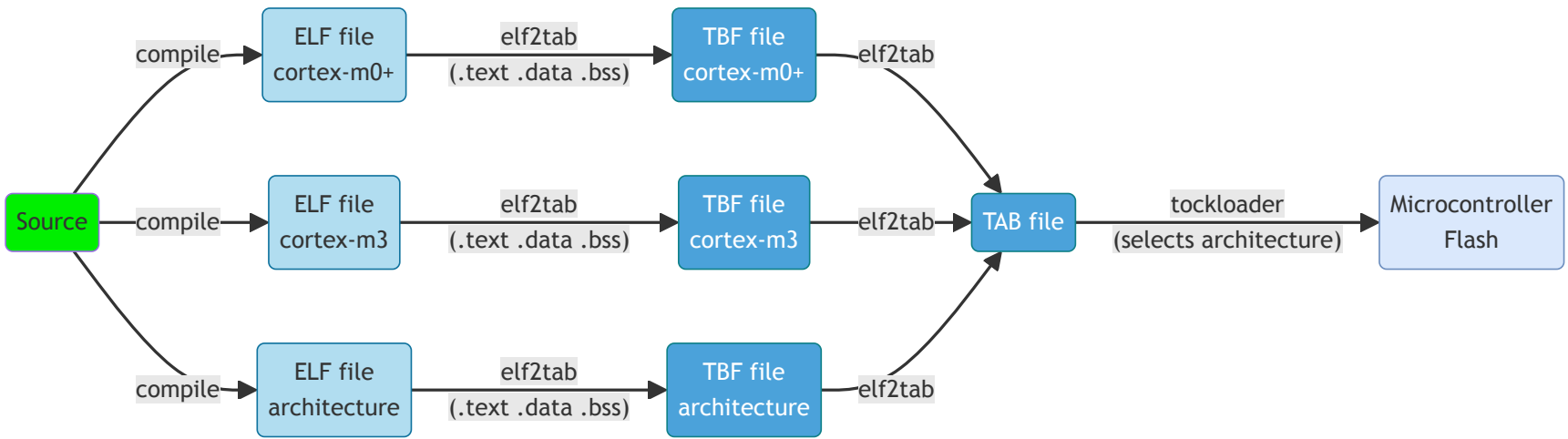
The Stack


Processes
(Any language)
C App ported to Tock
App written in Rust
while (1)
Untrusted
(isolated by the MPU and preemptively scheduled)
libtock-c
libtock-rs
OxidOS Apps API
OxidOS Apps API
Syscall Interface
yield
memop
IPC
Kernel (Rust)
Standardized Hardware Interface Layer (HIL)
Core Kernel
Capsules + OxidOS Drivers
CAN
Graphic Display
Measurement
Thread
Virtual Timer
SD Card
SI7021
Console
BLE
6LoWPAN
Temp. Sensor
Untrusted
(unsafe forbidden)
OxidOS Trait APIs
Microcontroller-specific Peripheral Drivers
+ OxidOS Microcontroller Abstraction Layer (MCAL)
SPI
RNG
Timer
ADC
GPIO
I2C
UART
AES
DAC
USB
Trusted
(unsafe allowed for MMIO, PIC etc.)
Hardware
CPU
SPI
RNG
Timer
ADC
GPIO
I2C
UART
AES
DAC
USB

Processes

separate binaries

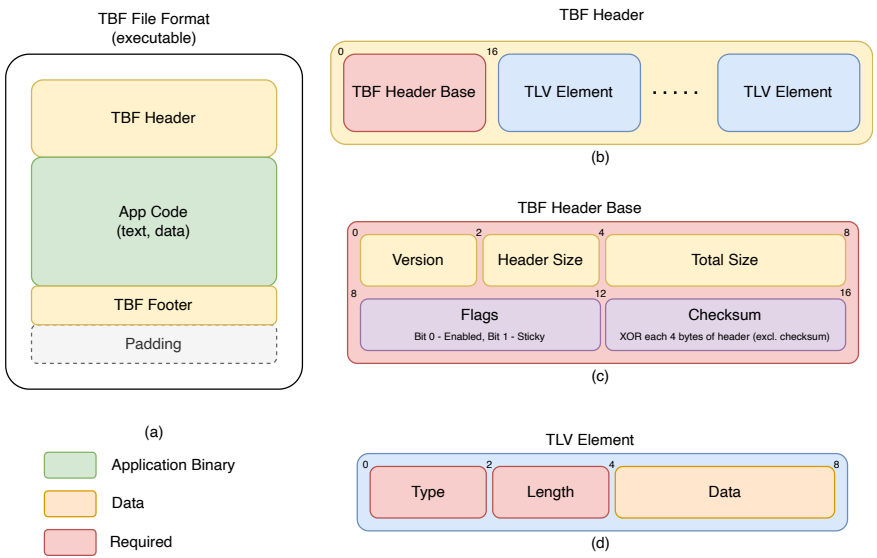
- compiled separately from the kernel
- written in any language that compiles (C, Rust, ...)
- saved into the *Tock Binary Format* (*TBF*) / *Tock Application Bundle* (*TAB*)

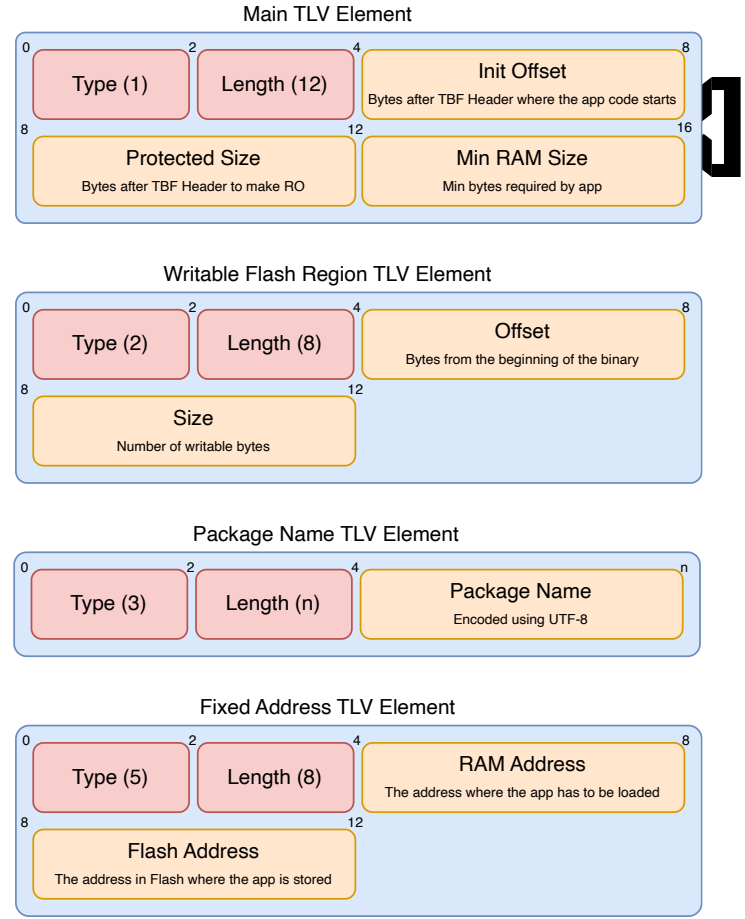


Tock Binary Format

stores

- **headers** about how to load the application
- the **binary code** and **data**
- **credential** footers





Memory Layout

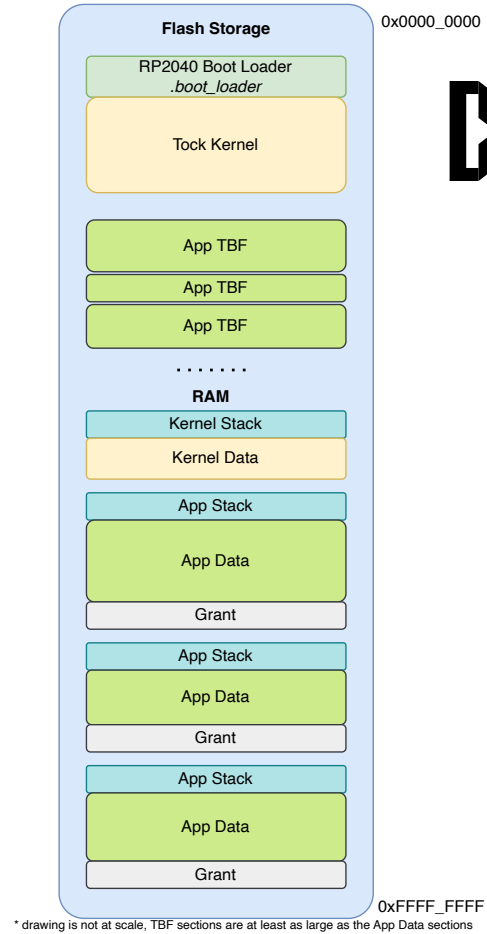
for the RP2040

Kernel

- is written in flash separated from the apps
- loads each app at boot

Applications

- each application TBF is written to the flash separately
- each application has a separate
  - *stack* in RAM
  - *grant* section where the kernel stores data about the app
  - *data* section in RAM



Memory Layout

for the RP2040 at runtime

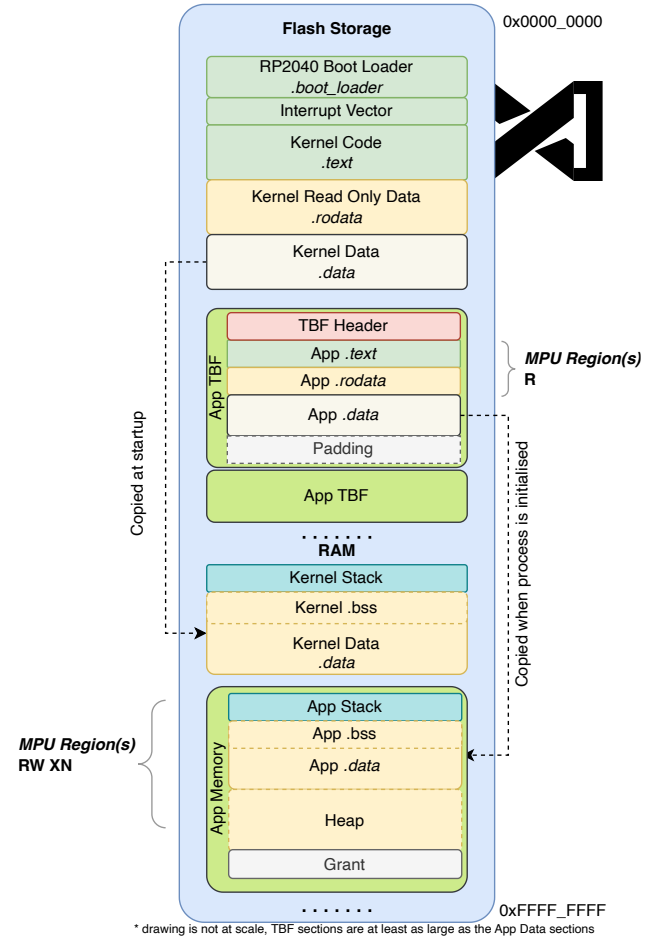
Kernel

- sets up the MPU every time it switches to a process

Applications

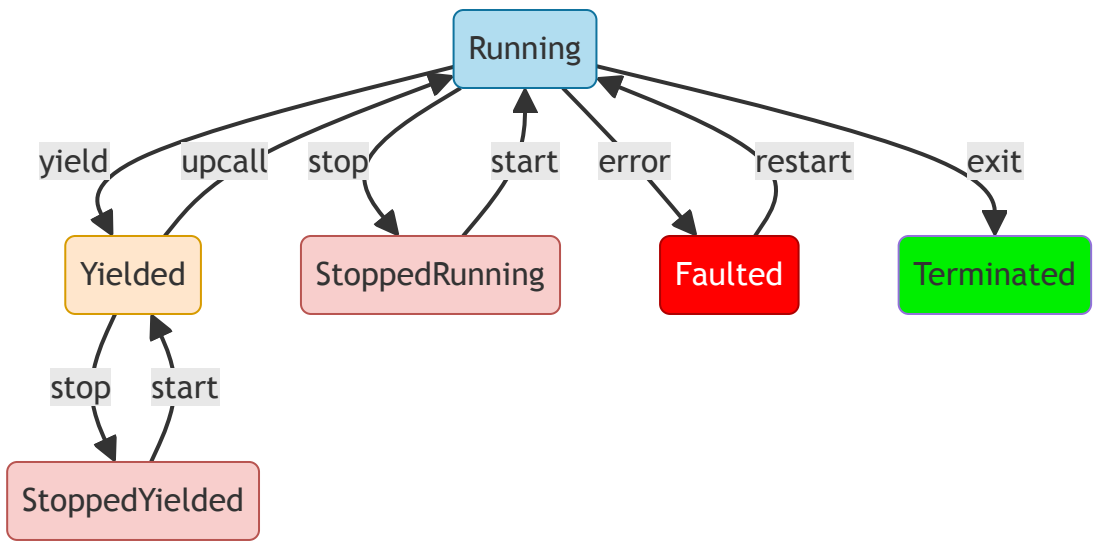
- can read and execute its code
- can read and write its *stack* and *data*
- can read and write the *allocated heap*

Applications are **not allowed** to access the **kernel's memory** or **the peripherals**.



Process States

- Tock runs only on *single core*
- *Running* state means the process is ready to run
- *Yielded* means the process waits for an event (*upcall*)
- *start* and *stop* are user commands
- a process is stopped only if the user asked it



Application API

libraries

Tock provides two libraries:

- `libtock-c` that is fully supported
- `libtock-rs` that is in development ⚠ [1]

1. Due to a Rust compiler issue, Rust applications are not relocatable. This means that developers have to know at compile time the load addresses for Flash and RAM. ↩

Example Application (C)

```c
#include <libtock-sync/services/alarm.h>
#include <libtock/interface/led.h>

int main(void) {
  // Ask the kernel how many LEDs are on this board.
  int num_leds;
  int err = libtock_led_count(&num_leds);
  if (err < 0) return err;

  // Blink the LEDs in a binary count pattern and scale
  // to the number of LEDs on the board.
  for (int count = 0; ; count++) {
    for (int i = 0; i < num_leds; i++) {
      if (count & (1 << i)) {
        libtock_led_on(i);
      } else {
        libtock_led_off(i);
      }
    }

    // This delay uses an underlying alarm in the kernel.
    libtocksync_alarm_delay_ms(250);
  }
}
```

Example Application (Rust)

```rust
//! A simple libtock-rs example. Just blinks all the LEDs.

#![no_main]
#![no_std]

use libtock::alarm::{Alarm, Milliseconds};
use libtock::leds::Leds;
use libtock::runtime::{set_main, stack_size};

set_main! {main}
stack_size! {0x200}

fn main() {
    if let Ok(leds_count) = Leds::count() {
        loop {
            for led_index in 0..leds_count {
                let _ = Leds::toggle(led_index as u32);
            }
            Alarm::sleep_for(Milliseconds(250)).unwrap();
        }
    }
}
```

Faults

similar to segfaults

- the kernel and apps can fault
- a detailed debug message can be displayed
- due to MPU usage Tock apps fault on:
  - trying to access memory outside its data (includes peripheral access)
  - stack overflow
  - trying to perform privileged operations

```
---| Fault Status |---
Data Access Violation:              true
Forced Hard Fault:                  true
Faulting Memory Address:            0x00000000
Fault Status Register (CFSR):       0x00000082
Hard Fault Status Register (HFSR):  0x40000000

---| App Status |---
App: crash_dummy   -   [Fault]
 Events Queued: 0   Syscall Count: 0   Dropped Callback Cou
 Restart Count: 0
 Last Syscall: None

  Address  | Region Name    Used | Allocated (bytes)
0x20006000
           | ▼ Grant         948 |    948
0x20005C4C
           | Unused
0x200049F0
           | ▲ Heap            0 |   4700               S
0x200049F0                                               R
           | Data            496 |    496               A
0x20004800                                               M
           | ▼ Stack          72 |   2048
0x200047B8
           | Unused
0x20004000
```

OxidOS Developer Platform

Developer Platform

online development environment

- uses WASM to emulate a virtual ECU
- runs ARM Cortex-M TBF applications
- dashboard

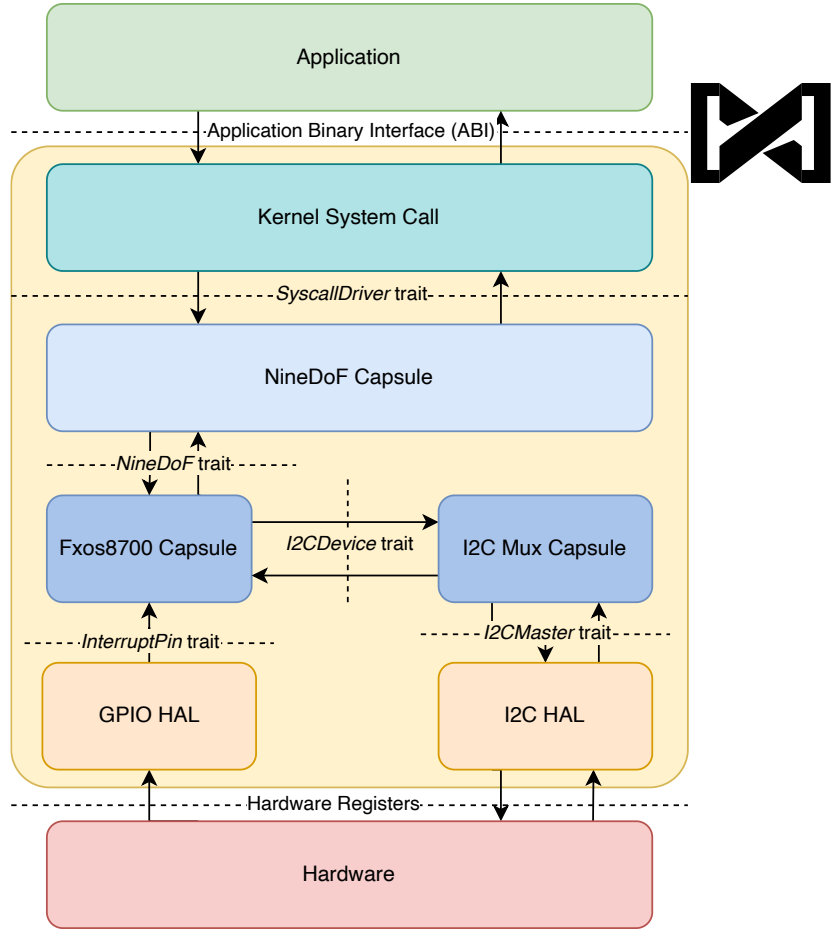
Singup

1. Singup to OxidOS Developer Platform
2. Create a new project and open it
3. Open the Workshop

System Calls

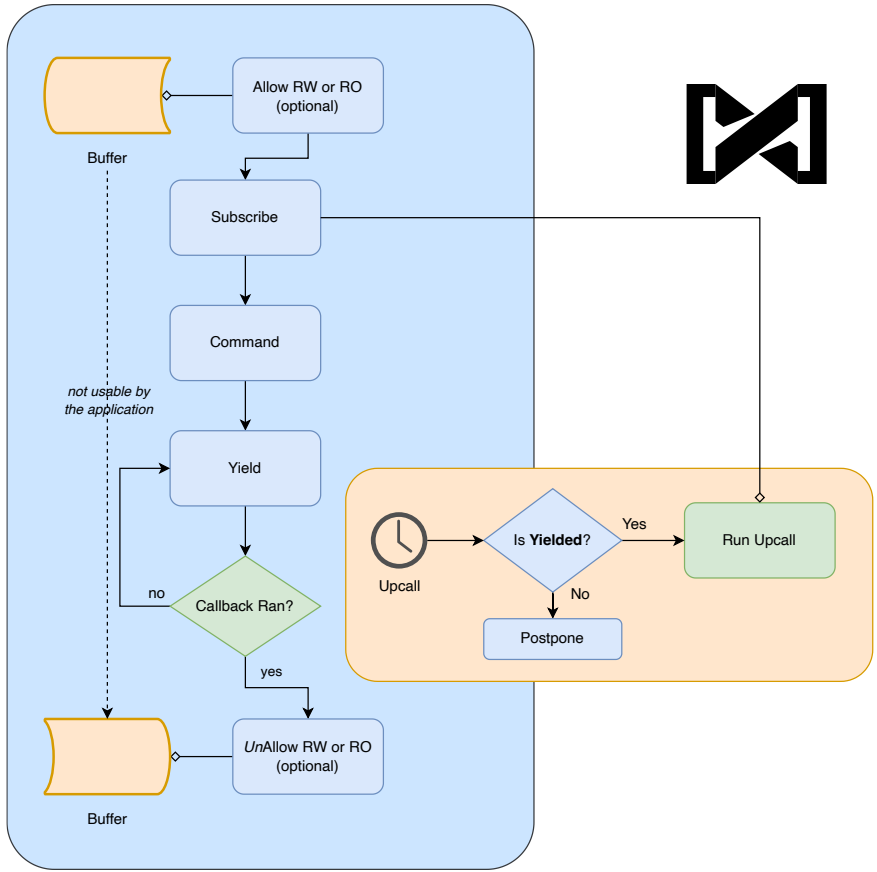
Syscall Stack

- *kernel* exposes a set of traits called *Hardware Interface Layer* (*HIL*)
- drivers use *HIL* to communicate in between each other
- *SyscallDrivers* expose system calls to user space API



System Calls

0. Yield
1. Subscribe
2. Command
3. ReadWriteAllow
4. ReadOnlyAllow
5. Memop
6. Exit
7. UserspaceReadableAllow



5: Memop

Memop expands the memory segment available to the process, allows the process to retrieve pointers to its allocated memory space, provides a mechanism for the process to tell the kernel where its stack and heap start, and other operations involving process memory.

```
memop(op_type: u32, argument: u32) -> [[ VARIES ]] as u32
```

Arguments

- `op_type`: An integer indicating whether this is a `brk` (0), a `sbrk` (1), or another memop call.
- `argument`: The argument to `brk`, `sbrk`, or other call.

Each memop operation is specific and details of each call can be found in the memop syscall documentation.

Return

- Dependent on the particular *memop* call.

6: Exit

The process signals the kernel that it has no more work to do and can be stopped or that it asks the kernel to restart it.

```
tock_exit(completion_code: u32)
tock_restart(completion_code: u32)
```

Return

None

2: Command

Command instructs the driver to perform a specific action.

```
command(driver: u32, command_number: u32, argument1: u32, argument2: u32) -> CommandReturn
```

Arguments

- `driver` : integer specifying which driver to use
- `command_number` : the requested command.
- `argument1` : a command-specific argument
- `argument2` : a command-specific argument

One Tock convention with the *Command* system call is that command number 0 will always return a value of 0 or greater if the driver is present.

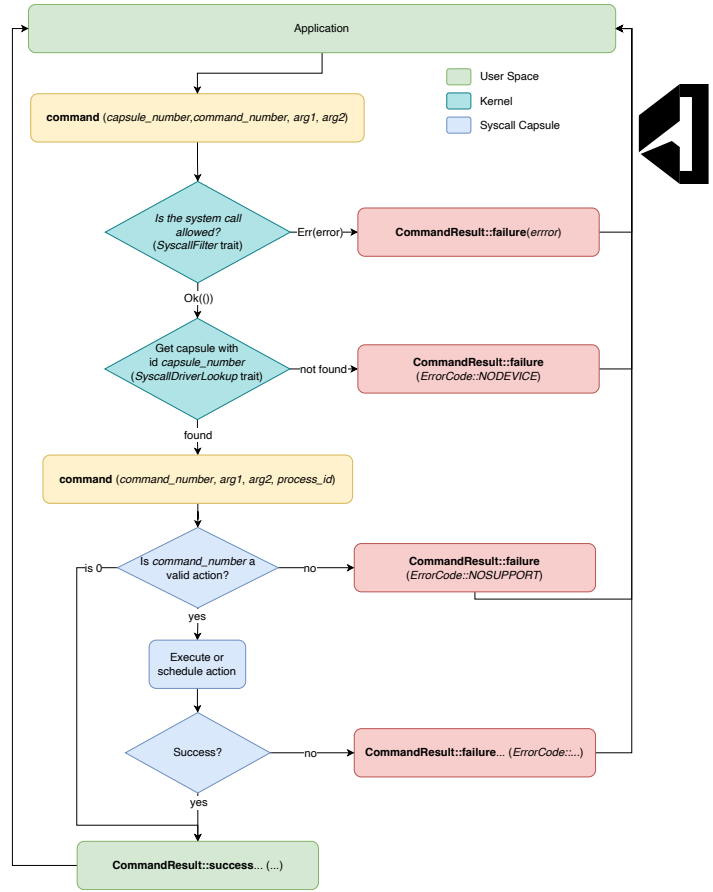
Return

- three `u32` numbers
- Errors
  - `NODEVICE` if `driver` does not refer to a valid kernel driver.
  - `NOSUPPORT` if the driver exists but doesn't support the `command_number` .
  - Other return codes based on the specific driver.

2: Command Flow

the kernel

1. asks the *syscall filter* if the call is allowed
2. searches for the capsule (driver)
3. sends the command to the capsule



1: Subscribe

Subscribe assigns upcall functions to be executed in response to various events.

```
subscribe(driver: u32, subscribe_number: u32, upcall: u32, userdata: u32) -> Result<Upcall, (Upcall, ErrorCode)>
```

Arguments

- `driver` : integer specifying which driver to use
- `subscribe_number` : event number
- `upcall` : function's pointer to call upon event

```
void upcall(int arg1, int arg2, int arg3, void* userdata)
```

- `userdata` : value that will be passed back, usually a pointer

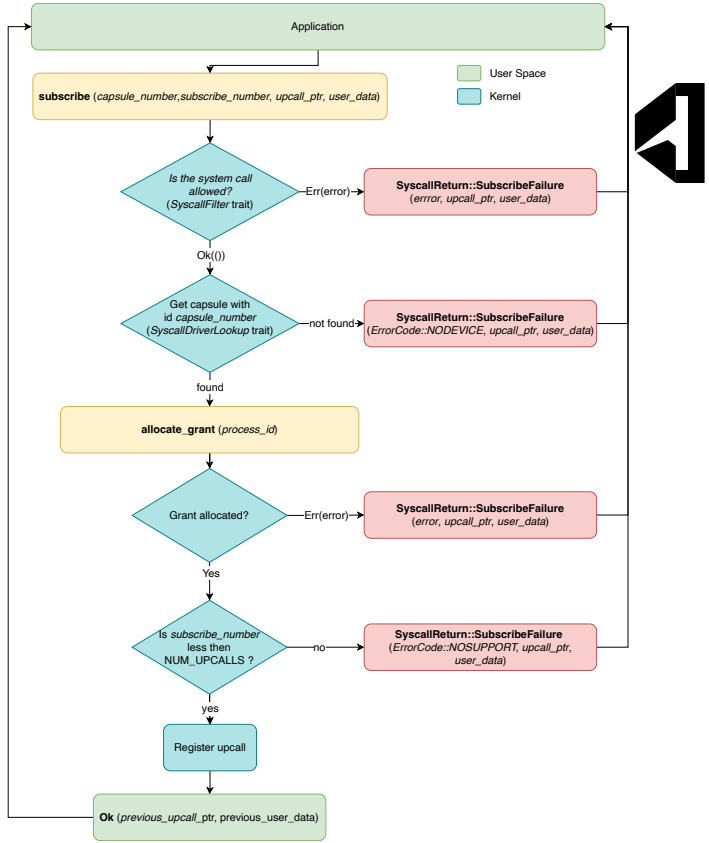
Return

- The previously registered upcall or `TOCK_NULL_UPCALL`
- Errors
  - `NODEVICE` if `driver` does not refer to a valid kernel driver.
  - `NOSUPPORT` if the driver exists but doesn't support the `subscribe_number` .

1: Subscribe Flow

the kernel

1. asks the *syscall filter* if the call is allowed
2. searches for the capsule (driver)
3. asks the capsule to allocate its grant
4. verifies that the *subscribe_number* is in between `0` and `NUM_UPCALLS-1`
5. registers the callback function



0: Yield

Yield transitions the current process from the Running to the Yielded state.

```
// waits for the next upcall
// The process will not execute again until another upcall re-schedules the
// process.
yield()

// does not wait for the next upcall
// If a process has no enqueued upcalls, the
// process immediately re-enters the Running state.
yield_no_wait()
```

Return

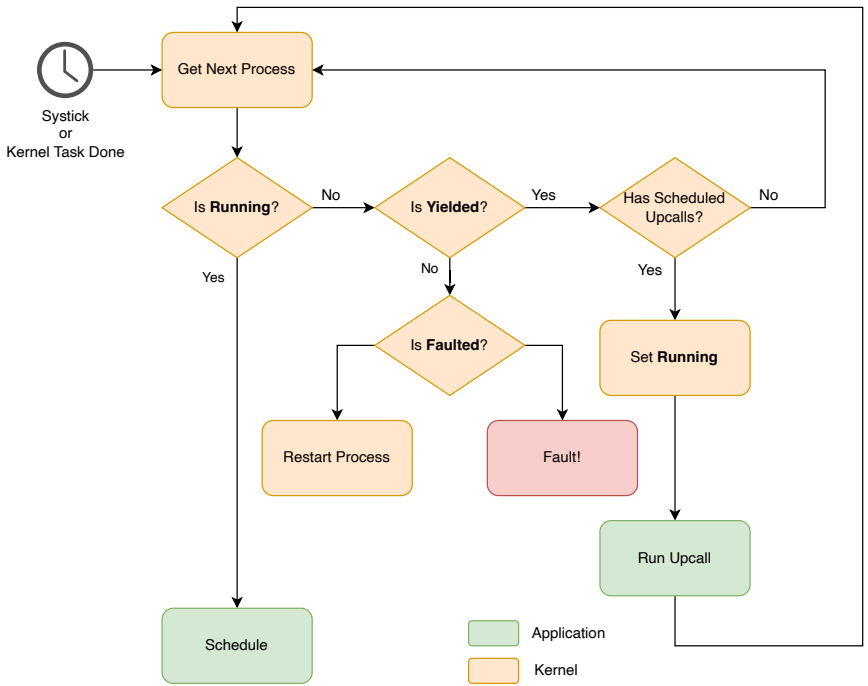
yield: None

yield_no_wait:

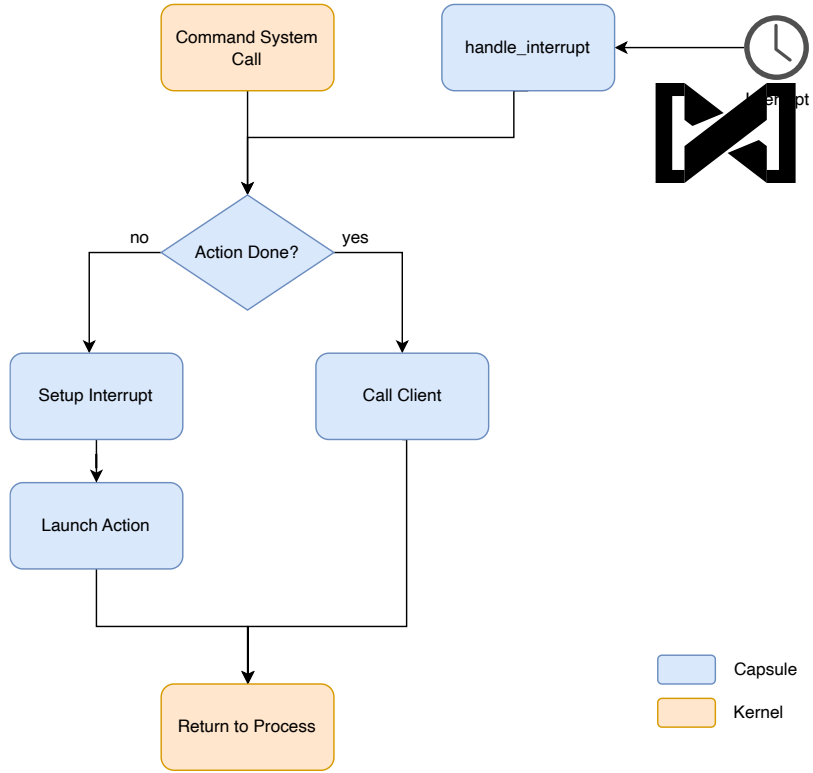
- 1 - *upcall* ran
- 0 - there was no queued *upcall* function to execute

Scheduler

using command, subscribe and yield



how the scheduler works



how drivers work

3 and 4: AllowRead(Write/Only)

Allow shares memory buffers between the kernel and application.

```
allow_readwrite(driver: u32, allow_number: u32, pointer: usize, size: u32) -> Result<ReadWriteAppSlice, (ReadWriteAppSl
allow_readonly(driver: u32, allow_number: u32, pointer: usize, size: u32) -> Result<ReadWriteAppSlice, (ReadWriteAppSlic
```

Arguments

- `driver`: integer specifying which driver to use
- `allow_number`: driver-specific integer specifying the purpose of this buffer
- `pointer`: pointer to the buffer in the process memory space
  - null pointer revokes a previously shared buffer
- `size`: the length of the buffer

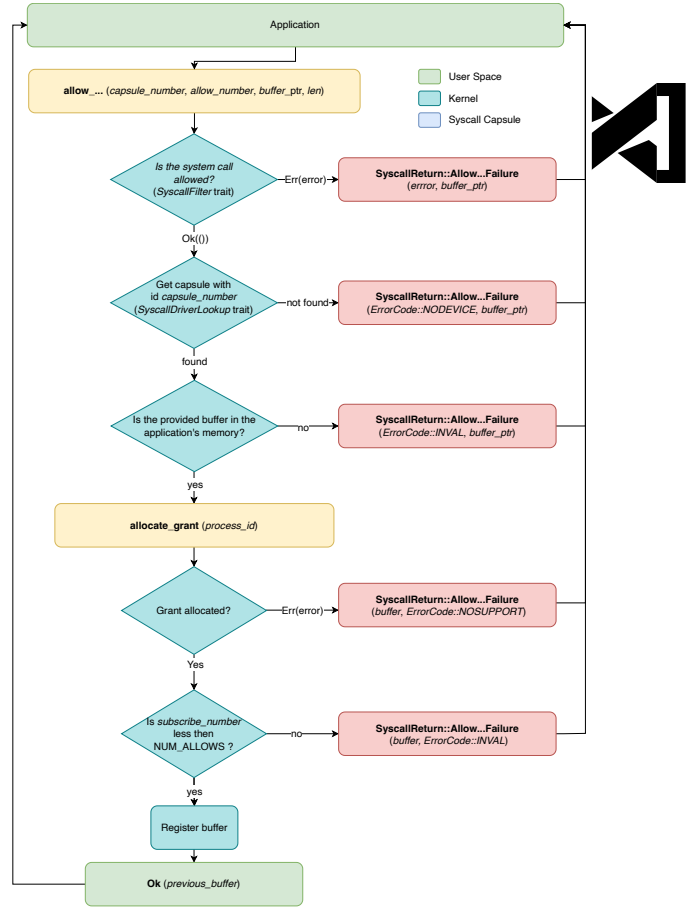
Return

- The previous allowed buffer or NULL
- Errors
  - `NODEVICE` if `driver` does not refer to a valid kernel driver.
  - `NOSUPPORT` if the driver exists but doesn't support the `allow_number`.
  - `INVAL` the buffer referred to by `pointer` and `size` lies completely or partially outside of the processes addressable RAM.

3 and 5: AllowRead(Write/Only) Flow

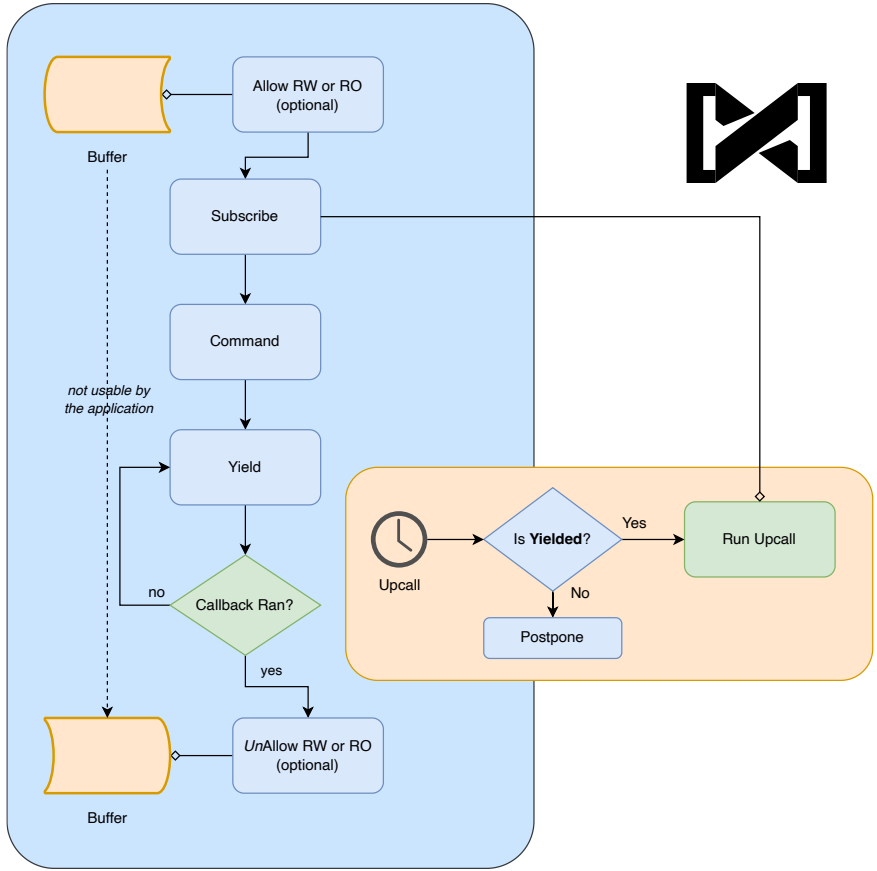
the kernel

1. asks the *syscall filter* if the call is allowed
2. searches for the capsule (driver)
3. asks the capsule to allocate its grant
4. verifies if the *allow_number* if between `0` and `NUM_ALLOW_RW-1` (`NUM_ALLOW_RO-1`)
5. verifies if the whole buffer in the application's memory
6. registers the buffer

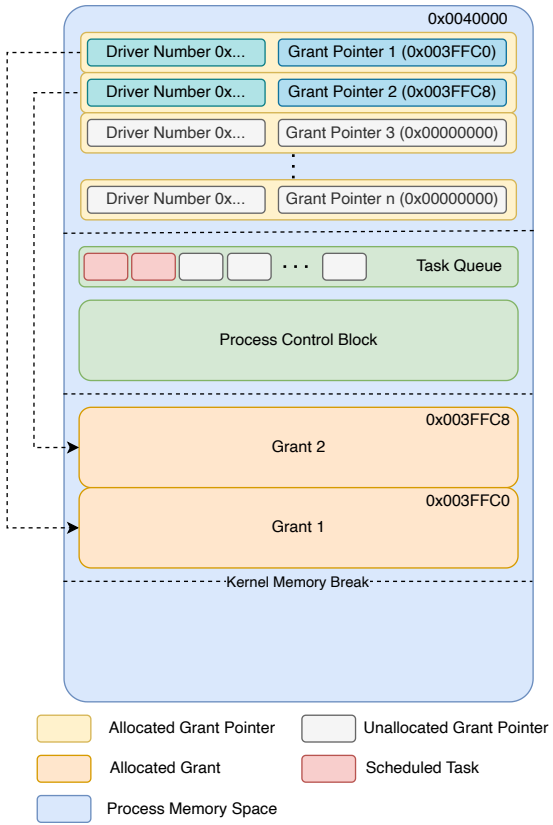


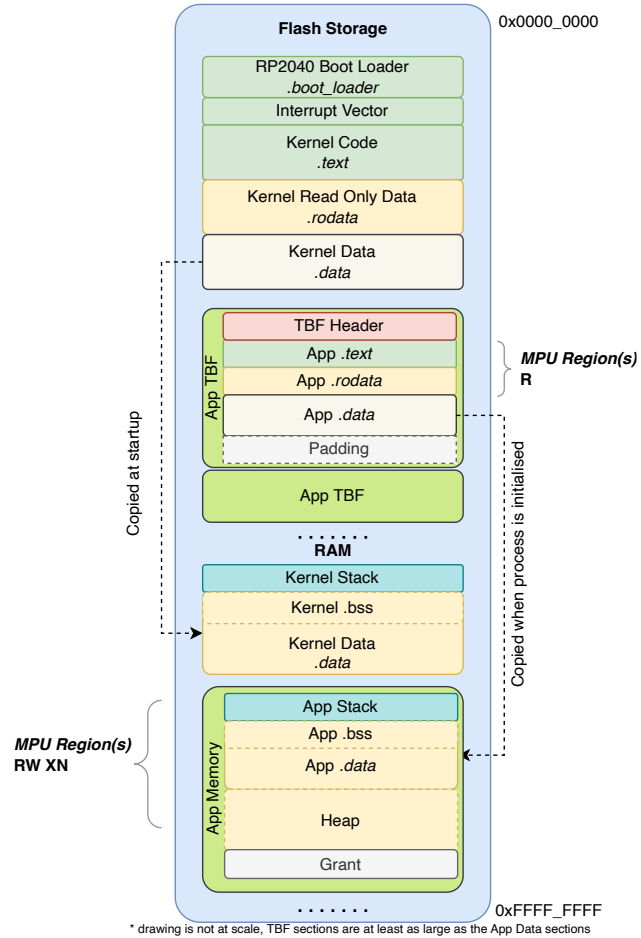
System Call Pattern

1. *allow*: if data exchange is required, share a buffer with a driver
2. *subscribe* to the *action done* event
3. send a *command* to ask the driver to start performing an action
4. *yield* to wait for the *action done* event
   - *the kernel calls a callback*
   - verify if the expected event was triggered, if not *yield*
5. *unallow*: get the buffer back from the driver

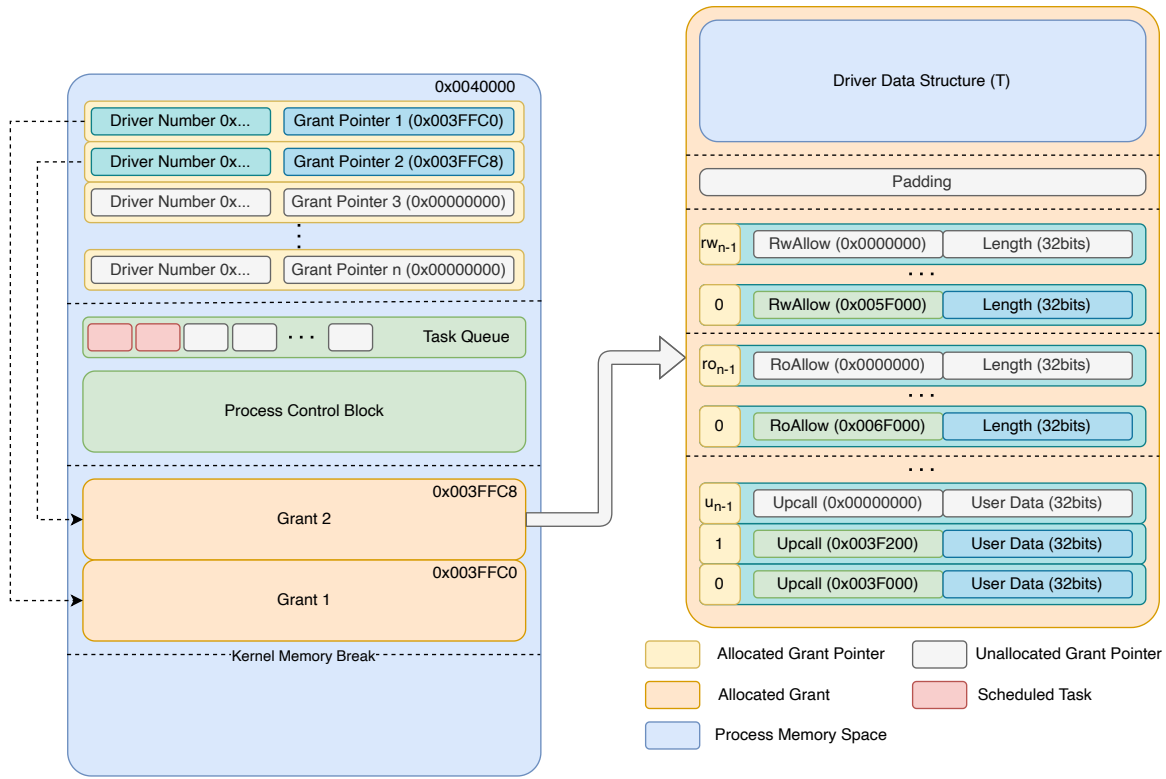


Grant


0x0040000
Driver Number 0x...
Grant Pointer 1 (0x003FFC0)
Driver Number 0x...
Grant Pointer 2 (0x003FFC8)
Driver Number 0x...
Grant Pointer 3 (0x00000000)
Driver Number 0x...
Grant Pointer n (0x00000000)
Task Queue
Process Control Block
0x003FFC8
Grant 2
0x003FFC0
Grant 1
Kernel Memory Break
Allocated Grant Pointer
Unallocated Grant Pointer
Allocated Grant
Scheduled Task
Process Memory Space


Flash Storage
0x0000_0000
RP2040 Boot Loader
.boot_loader
Interrupt Vector
Kernel Code
.text
Kernel Read Only Data
.rodata
Kernel Data
.data
App TBF
TBF Header
App .text
App .rodata
App .data
Padding
App TBF
RAM
Kernel Stack
Kernel .bss
Kernel Data
.data
App Memory
App Stack
App .bss
App .data
Heap
Grant
0xFFFF_FFFF
MPU Region(s)
R
MPU Region(s)
RW XN
Copied at startup
Copied when process is initialised
* drawing is not at scale, TBF sections are at least as large as the App Data sections

Grant


0x0040000
Driver Number 0x...
Grant Pointer 1 (0x003FFC0)
Driver Number 0x...
Grant Pointer 2 (0x003FFC8)
Driver Number 0x...
Grant Pointer 3 (0x00000000)
Driver Number 0x...
Grant Pointer n (0x00000000)
Task Queue
Process Control Block
0x003FFC8
Grant 2
0x003FFC0
Grant 1
Kernel Memory Break
Driver Data Structure (T)
Padding
rw n-1
RwAllow (0x0000000)
Length (32bits)
0
RwAllow (0x005F000)
Length (32bits)
ro n-1
RoAllow (0x0000000)
Length (32bits)
0
RoAllow (0x006F000)
Length (32bits)
u n-1
Upcall (0x00000000)
User Data (32bits)
1
Upcall (0x003F200)
User Data (32bits)
0
Upcall (0x003F000)
User Data (32bits)
Allocated Grant Pointer
Unallocated Grant Pointer
Allocated Grant
Scheduled Task
Process Memory Space

Writing a System Call Driver (Capsule)

```rust
pub trait SyscallDriver {
    /// System call for a process to perform a short synchronous operation
    /// or start a long-running split-phase operation (whose completion
    /// is signaled with an upcall). Command 0 is a reserved command to
    /// detect if a peripheral system call driver is installed and must
    /// always return a CommandReturn::Success.
    fn command(&self, command_num: usize, r2: usize, r3: usize, process_id: ProcessId) -> CommandReturn {
        CommandReturn::failure(ErrorCode::NOSUPPORT)
    }

    /// System call for a process to pass a buffer (a
    /// `UserspaceReadableProcessBuffer`) to the kernel that the kernel can
    /// either read or write. The kernel calls this method only after it checks
    /// that the entire buffer is within memory the process can both read and
    /// write.
    fn allow_userspace_readable(&self, app: ProcessId, which: usize, slice: UserspaceReadableProcessBuffer)
            -> Result<UserspaceReadableProcessBuffer, (UserspaceReadableProcessBuffer, ErrorCode)> {
        Err((slice, ErrorCode::NOSUPPORT))
    }

    /// Request to allocate a capsule's grant for a specific process.
    fn allocate_grant(&self, process_id: ProcessId) -> Result<(), crate::process::Error>;
}
```

Register a Syscall Driver

1. implement the `SyscallDriver` trait
2. add the driver as a field of the board structure
3. associate the driver with a driver number using `SyscallDriverLookup::with_driver` method in `main.rs`

Q&A