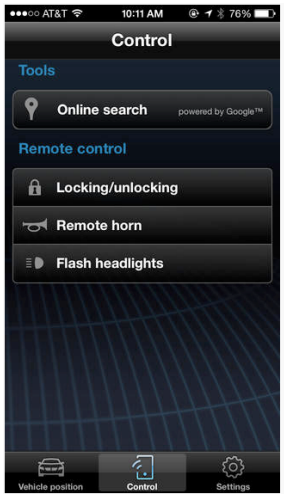
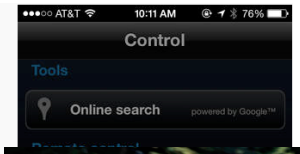
Building a Secure Embedded Kernel in Rust

Amit Levy[a], Branden Ghena[b], Bradford Campbell[b],
Pat Pannuto[b], Nicholas Matsakis[c], Prabal Dutta[b], Philip Levis[a]
Platform Lab
May 8, 2017

[a]Stanford University [b]University of Michigan [c]Mozilla Research

The Internet of Things (IoT)





A Security Disaster

The Economist | World politics | Business & finance | Economics | Science & technology | Culture

Cyber-security

The internet of things (to be hacked)

Hooking up gadgets to the web promises huge benefits. But security must not be an afterthought

Jul 12th 2014 | From the print edition

Timekeeper | Like 217 | Tweet 594

How the Internet of Things Could Kill You

By Fahmida Y. Rashid JULY 18, 2014 7:30 AM - Source: Tom's Guide US | 5 COMMENTS

Hacking the Fridge: Internet of Things Has Security Vulnerabilities

JESS SCANLON | MORE ARTICLES
JUNE 28, 2014

Philips Hue LED smart lights hacked, home blacked out by security researcher

By Sal Cangeloso on August 15, 2013 at 11:45 am | 7 Comments

- HP conducted a security analysis of IoT devices[1]
  - 80% had privacy concerns
  - 80% had poor passwords
  - 70% lacked encryption
  - 60% had vulnerabilities in UI
  - 60% had insecure updates

[1] http://fortifyprotect.com/HP_IoT_Research_Study.pdf

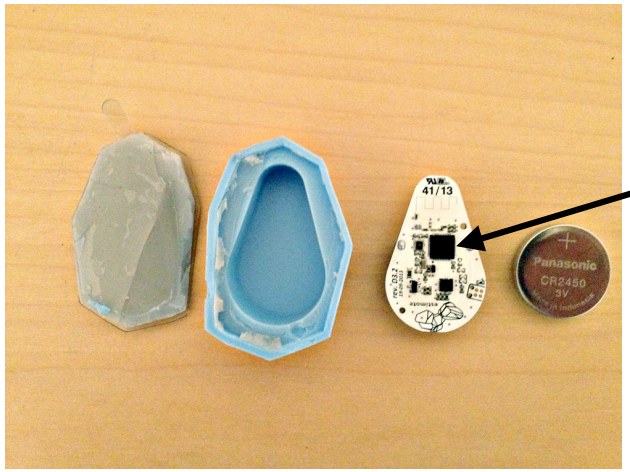
Let’s design and build a secure embedded kernel!

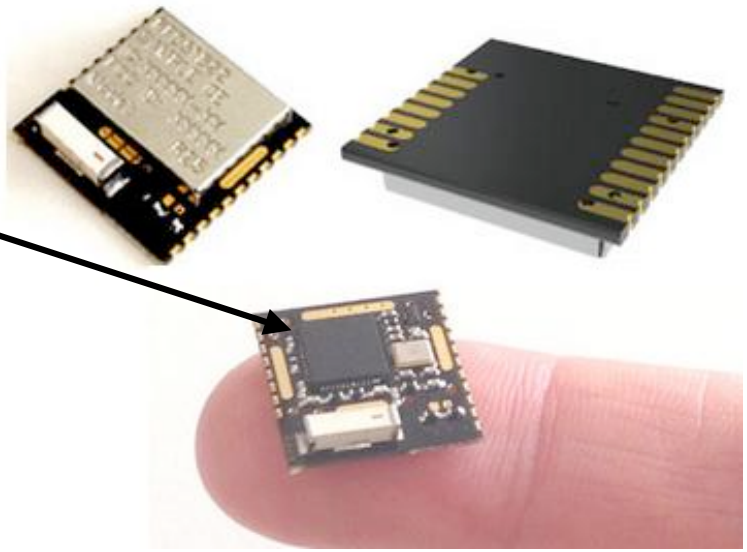
Outline

- Energy rules everything
- Kernel design goals
- Tock operating system design
  - Architecture: Rust, capsules, and processes
  - Rust challenges
  - Resource model: memory and CPU
- Some early observations

Embedded Devices

- Cortex M0+ with integrated 2.4GHz transceiver
  - Supports Bluetooth Low Energy
  - Two models: 32kB/256kB or 16kB/128kB
- DigiKey cost for 25,000: $1.99





Energy Rules Everything

- Most things don't have wires connected to them
  - All energy comes from a battery
  - Energy harvesting is sometimes but often not possible
- For IoT devices, energy is the limiting resource
- Two costs dominate
  - Sleep/idle current
  - Radio communication

Sleep and Wake Up

- nRF51422
- CR2032: 225mAh @ 3.0V
  - Can drive system at 1.8V, use a buck power converter to step down voltage, in this slide assume we just run at 3.0V

Mode	Current	Lifetime
Sleep	10.2 µA	920 days
CPU active	4.4 mA	2.1 days
Radio TX	16 mA	14 hours
Radio RX	13.4 mA	17 hours
5Hz BLE advertisement	~231 µA	40 days

Joules/Bit

- WiFi is more efficient than BLE and 802.15.4!
  - WiFi: 50Mbps (application), 600mW, 83MbpJ
  - 802.15.4: 200kbps (application), 60mW, 3.3MbpJ
  - BLE: 150kbps (application), 30mW, 5MbpJ
- But, transition times can dominate
  - Simpler radios wake up faster
- Simple case: send a 1kB packet
  - 1kB packet is 160μS in WiFi, 5ms in BLE
  - Suppose wakeup is 2ms WiFi, 200μS BLE
  - WiFi is awake 2.16ms (1.2mJ), BLE is awake 5.2ms (0.3mJ)
  - Costs less to send packet with BLE
  - 1kB is very large for IoT

Energy Rules Everything

- Moore's Law/Denard scaling don't drive designs

	TelosB	**imix**
Year	2004	2017
MCU	MSP430F1611	SAM4LC8CA
Sleep current	0.2 µA	1.6 µA
Word size	16-bit	32-bit
CPU clock	8 MHz	12-48 MHz
Flash	48 kB	512 kB
RAM	10kB	64kB

13 years, 6-fold increase in RAM *and* sleep current.

Outline

- Energy rules everything
- Kernel design goals
- Tock operating system design
  - Architecture: Rust, capsules, and processes
  - Rust challenges
  - Resource model: memory and CPU
- Some early observations

Dependability

- Embedded systems need to run for long periods of time without any intervention
- Place a high premium on *dependability*
- Often sacrifice other metrics, like speed or throughput, to improve dependability
  - E.g., if a FitBit crashes, reset it and lose all data; if that fails, return it to the manufacturer
  - What would you do if your USB authentication dongle was flaky/crashed?

But the World is Changing…

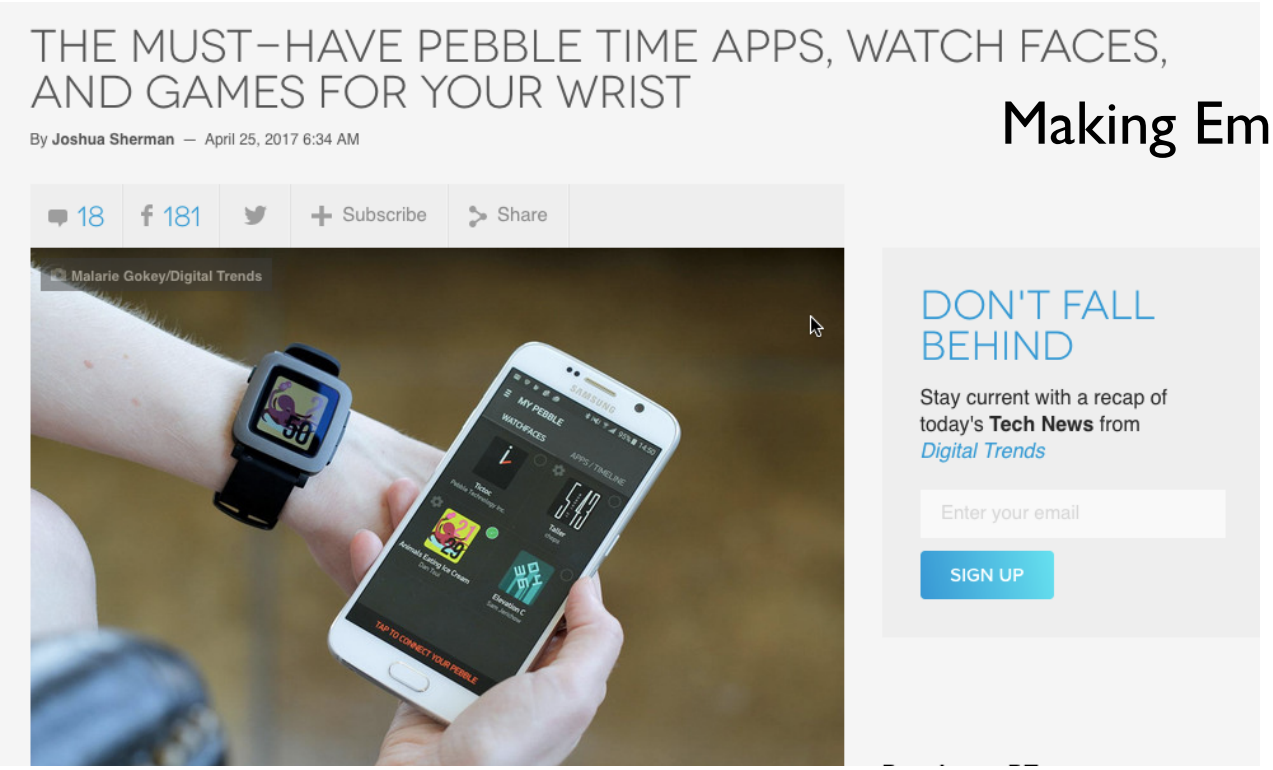
"An embedded system is a computerized system that is purpose-built for its application."

Elicia White
Making Embedded Systems, O'Reilly

But the World is Changing…

"An embedded system is a computerized system that is purpose-built for its application."

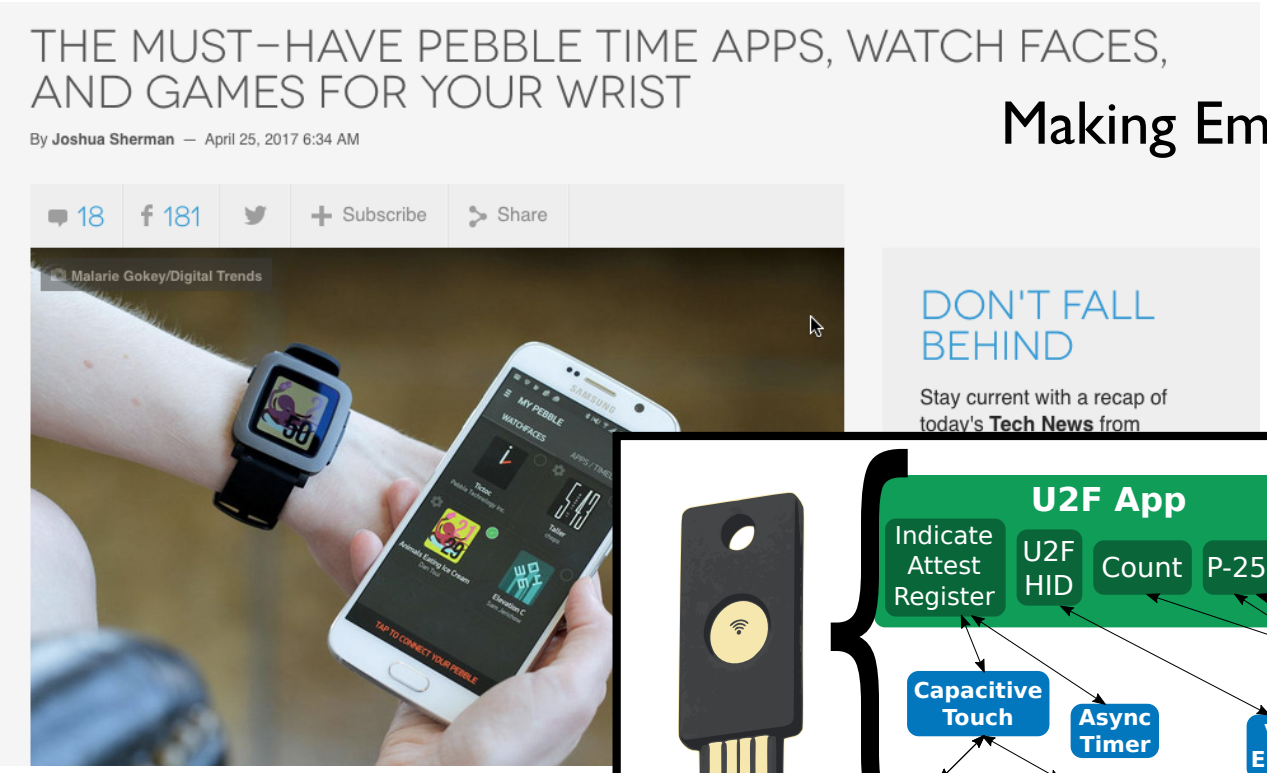
Elicia White
Making Embedded Systems, O'Reilly

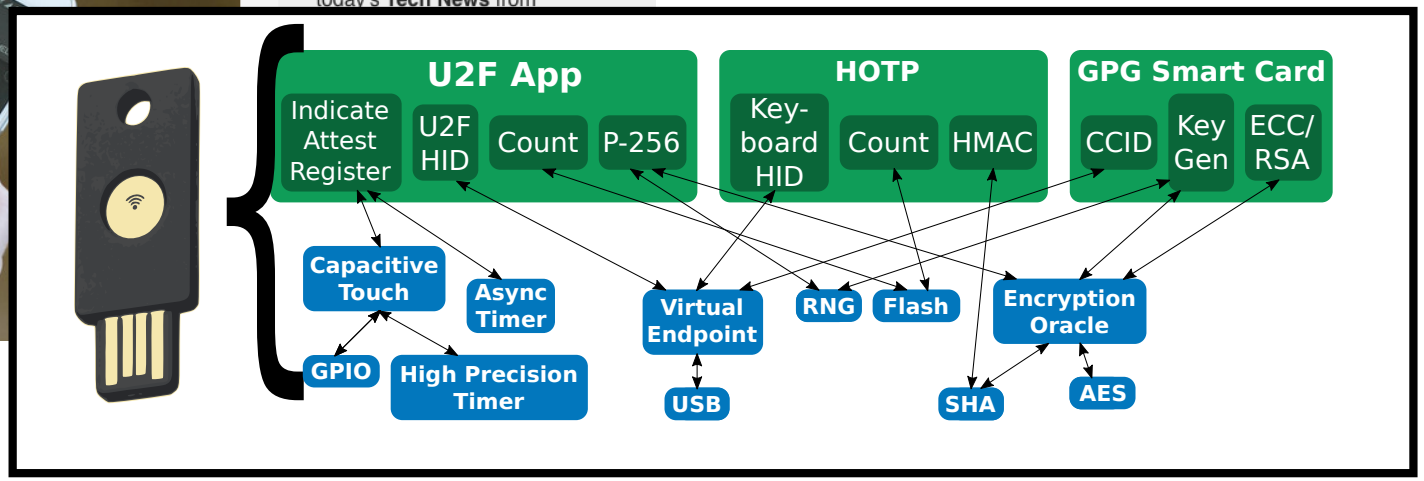


But the World is Changing…

"An embedded system is a computerized system that is purpose-built for its application."

Elicia White
Making Embedded Systems, O'Reilly



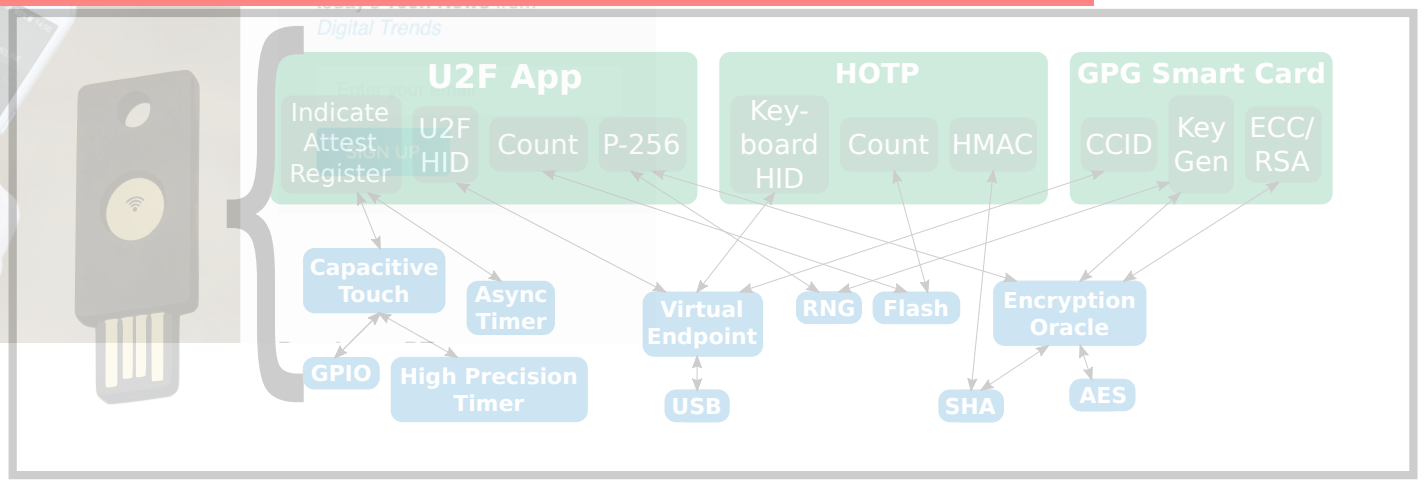


But the World is Changing…

"An embedded system is a computerized system that is purpose-built for its application."

a White
O'Reilly

A new class of embedded devices, that act as platforms supporting loadable programs within a particular application domain.



5 Kernel Design Goals

- No Resource Exhaustion
  - Don't run out of memory at runtime
  - Application infinite loop doesn't hang system
- Memory Isolation
  - Protect independent pieces of code from each other
  - Historically impossible: written in C, no MMU
- Memory Efficiency
  - Conserve precious RAM
- Concurrency
  - Can perform multiple operations at once (energy efficiency)
- Loadable Applications
  - Can be dynamically reprogrammed without kernel reinstall

Previous Systems

	No Resource Exhaustion	Memory Isolation	Memory Efficiency	Concurrency	Loadable Applications
Arduino			✓		
TinyOS			✓	✓	
SOS			✓	✓	✓
TOSThreads	✓			✓	✓
Contiki			✓	✓	✓
FreeRTOS				✓	
RiotOS			✓		

Previous Systems

	No Resource Exhaustion	Memory Isolation	Memory Efficiency	Concurrency	Loadable Applications
Arduino			✓		
T					
S					
T					
Contiki			✓	✓	✓
FreeRTOS				✓	
RiotOS			✓		

All of these are written in C, designed for 8-bit or 16-bit MCUs. Can more modern hardware and progamming languages remove these tradeoffs?

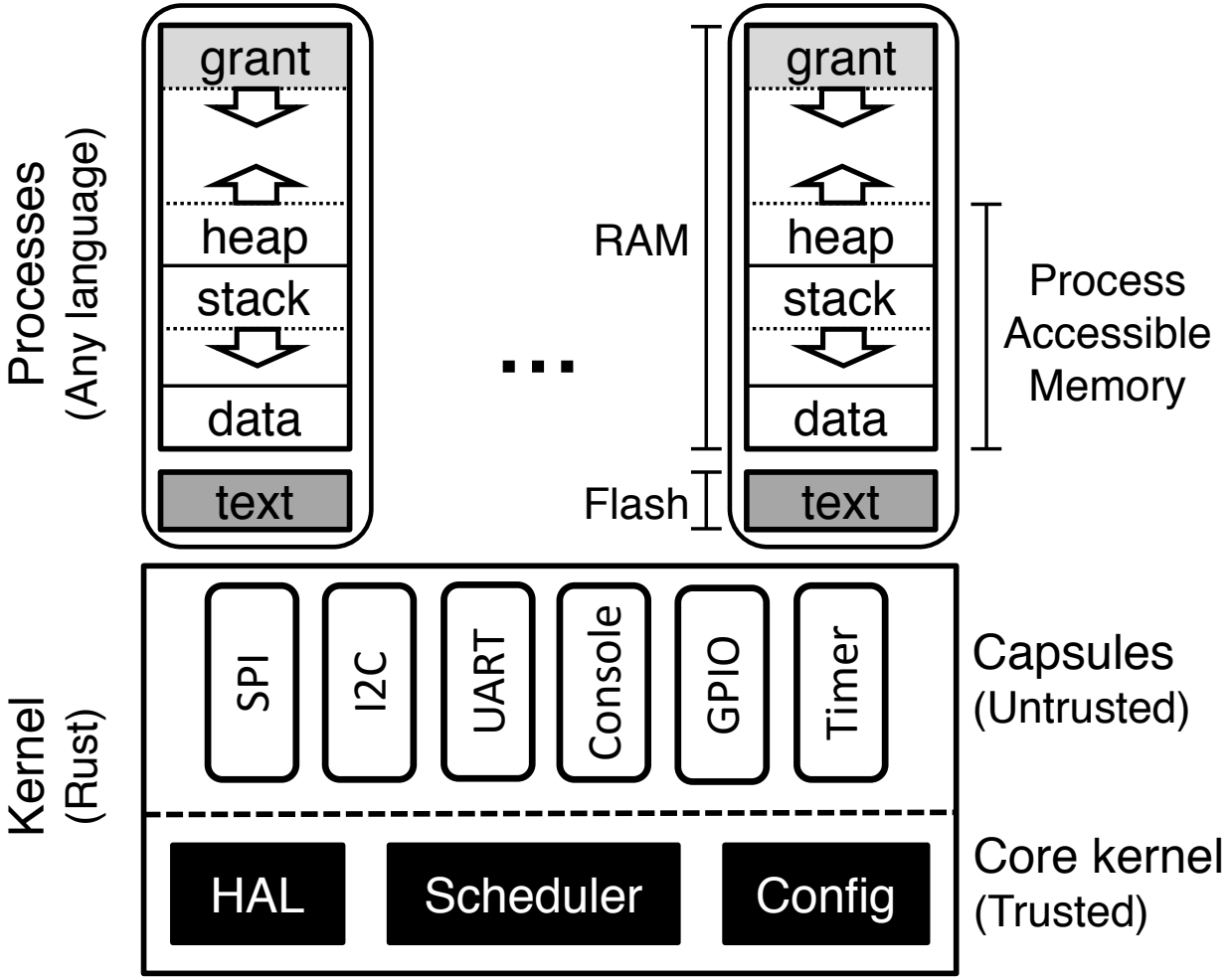
Outline

- Energy rules everything
- Kernel design goals
- Tock operating system design
  - Architecture: Rust, capsules, and processes
  - Rust challenges
  - Resource model: memory and CPU
- Some early observations

Tock Operating System

- Safe, multi-tasking operating system for memory-constrained devices
- Core kernel written in Rust, a safe systems language
  - Small amount of trusted code (can do unsafe things)
    - Rust bindings for memory-mapped I/O
    - Core scheduler, context switches
- Core kernel can be extended with *capsules*
  - Safe, written in Rust
  - Run inside kernel
- *Processes* can be written in any language (asm, C)
  - Leverage Cortex-M memory protection unit (MPU)
  - User-level, traps to kernel with system calls

Tock Architecture



Processes vs. Capsules

Category	**Capsule**	**Process**
Protection	Language	Hardware
Memory Overhead	None	Separate stack
Protection Granularity	Fine	Coarse
Concurrency	Cooperative	Preemptive
Update at Runtime	No	Yes

Rust Safety

- Tackles two problems:
  - Thread safety (concurrent access)
  - Memory safety (address contains proper type)
- Rule 1: a memory location can have one read/write pointer or multiple read-only pointers
  - mutable references and references in Rust parlance
- Rule 2: a reference can only point to memory that is assured to outlive the reference
  - prevents dangling pointers

Rust Rule 1

- Rule 1: a memory location can have one read/write pointer or multiple read-only pointers
  - mutable references and references in Rust parlance

```
let mut x = 5;
let y = &x;
let z = &x;
```

OK

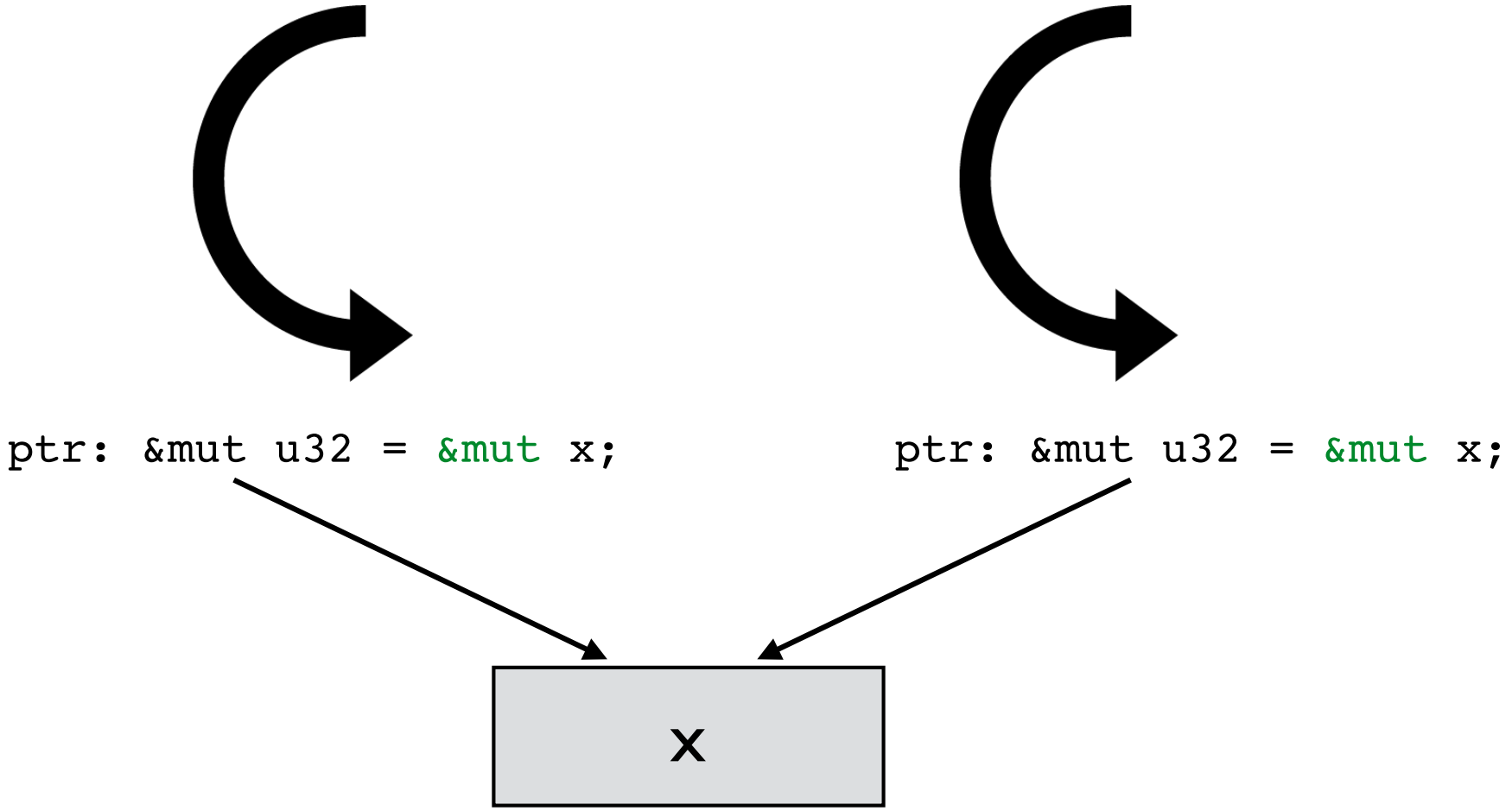
```
let mut x = 5;
let y = &mut x;
let z = &x;
```

No

```
let mut x = 5;
let y = &mut x;
let z = &mut x;
```

No

Why: Thread Safety



Why: Memory Safety

```c
union NumOrPointer {
  uint32_t Num;
  uint32_t* Pointer;
};

union NumOrPointer* external;
uint32_t* numptr = &external->Num;
*numptr = 0xdeadbeef;
*external->Pointer = 12345;
```

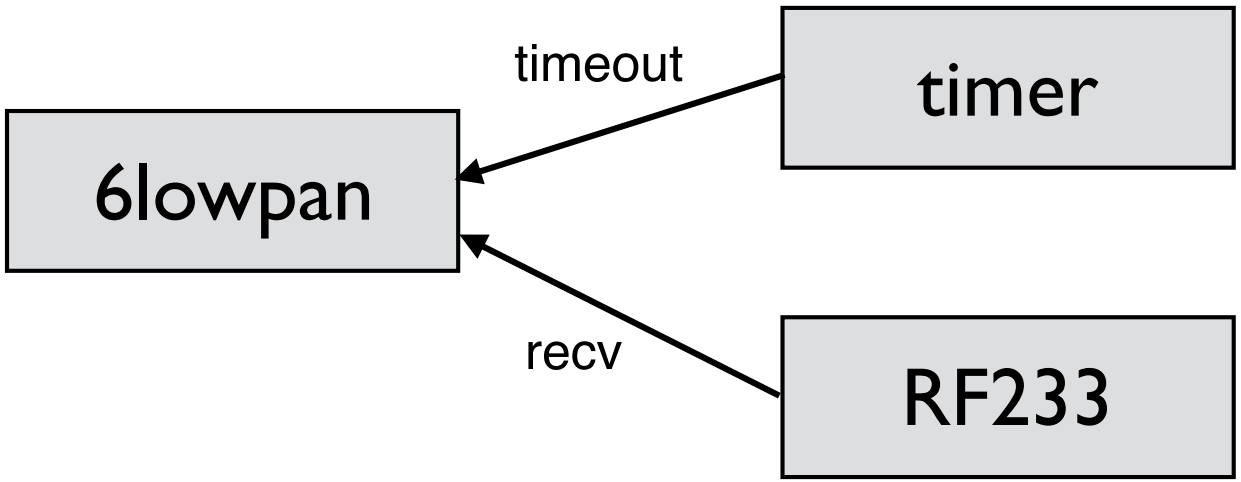
C

```rust
enum NumOrPointer {
  Num(u32),
  Pointer(&'static mut u32)
}

// n.b. compile error
let external : &mut NumOrPointer;
match external {
  &mut Pointer(ref mut internal) => {
    // This would violate safety and
    // write to memory at 0xdeadbeef
    *external = Num(0xdeadbeef);
    *internal = 12345;
  },
  ...
}
```

Rust

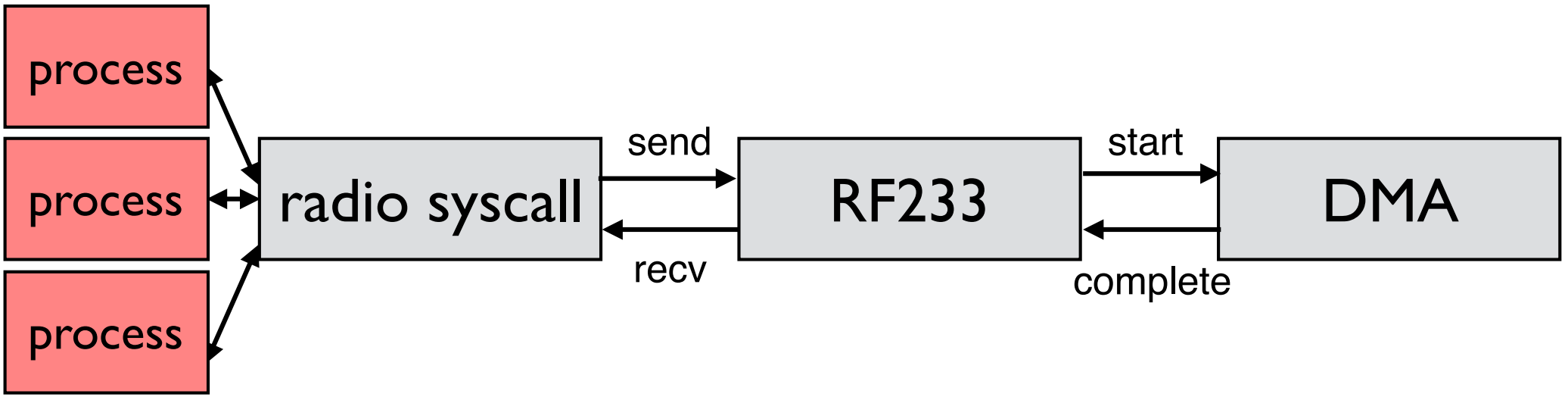
Problem 1: Events

- Often want to register multiple event callbacks on a single structure
  - E.g., networking stack has packet reception and timers
- Each callback needs a mutable reference



Problem 2: System Calls

- Decompose abstractions into kernel components (Rust APIs) with an optional system call interface
- Both upper and lower layers need to be able to call component: two mutable references



Strawman: Change Language

- Problem is that Rust is conflating memory safety and thread safety
- Introduce *execution contexts* into the language
- Allow multiple mutable references within a single execution context

Good: Solves thread safety.

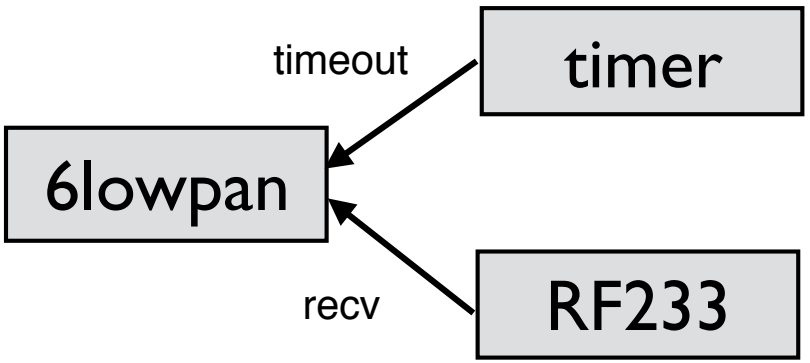
Bad: Doesn't solve memory safety.

```
// Does not compile: Cannot borrow across contexts
let x = 0;
spawn(|| { println!("In thread: {}", x); });

// Does compile: Can take sole ownership with move
let x = 0;
spawn(move || { println!("In thread: {}", x); });
```

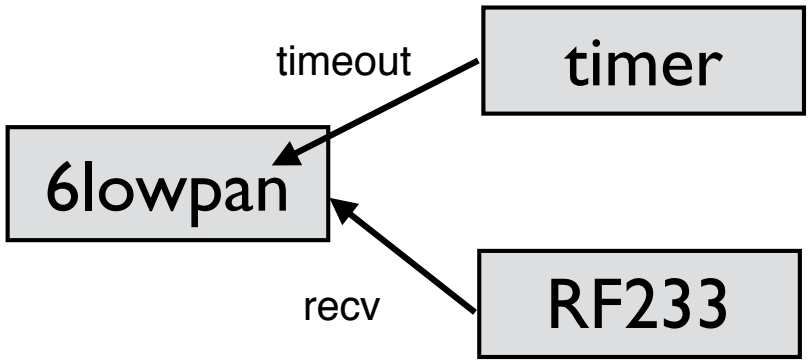
Ownership is Theft: Experiences Building an Embedded OS in Rust. Amit Levy, Michael P Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis and Pat Pannuto. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS 2015)*, 2015.

Events: Insight

- If we can ensure memory outlives reference, then multiple mutable references can be safe
- Rule: if there is a reference to memory block M, there cannot be any references *inside* M



Safe

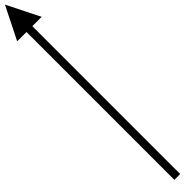


Unsafe

First Step: Cells

- Rust type: allows `get`/`set` with immutable references
- Good: multiple references can `set` values
- Bad: requires memory copies

```
pub struct Spi {
    registers: *mut SpiRegisters,
    client: Cell<Option<&'static SpiMasterClient>>,
    dma_read: Cell<Option<&'static DMAChannel>>,
    dma_write: Cell<Option<&'static DMAChannel>>,
    transfers_in_progress: Cell<u8>,
    dma_length: Cell<usize>,
}
```

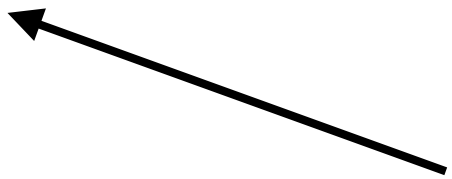


Fields accessed only a few times per operation (async DMA)

First Step: Cells

- Rust type: allows `get`/`set` with immutable references
- Good: multiple references can `set` values
- Bad: requires memory copies

```
pub struct DMAChannel {
    registers: *mut DMARegisters,
    nvic: nvic::NvicIdx,
    pub client: Option<&'static mut DMAClient>,
    enabled: Cell<bool>,
    buffer: Cell<'static, [u8]>,
}
```



Copying a whole buffer!

TakeCell

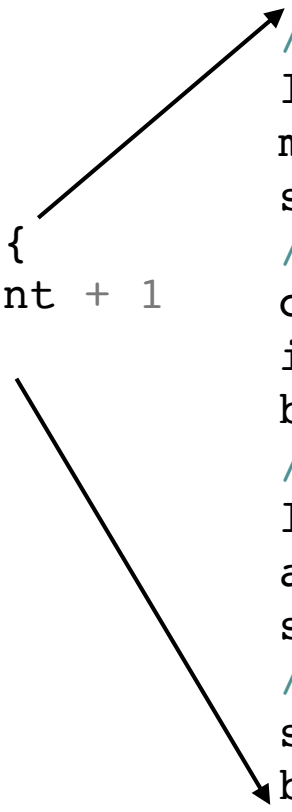
- Rather than copy data in/out, a TakeCell uses a closure
  - Pass code in
  - Thread safety: until first closure completes, other invocations see TakeCell as empty (None)
- Holds a reference to a type

```
struct App { /* many vars */ }

app: TakeCell<App>,

self.app.map(|app| {
  // code can read/write
  // app's variables
});
```

```
pub struct DMAChannel {
    registers: *mut DMARegisters,
    nvic: nvic::NvicIdx,
    pub client: Option<&'static mut DMAClie
    enabled: Cell<bool>,
    buffer: TakeCell<'static, [u8]>,
}
```

Example Code and ASM

```
struct App {
  count: u32,
  tx_callback: Callback,
  rx_callback: Callback,
  app_read: Option<AppSlice<Shared, u8>>,
  app_write: Option<AppSlice<Shared, u8>>,
}
pub struct Driver {
  busy: Cell<bool>,
  app: TakeCell<App>,
}
driver.app.map(|app| {
  app.count = app.count + 1
});
```



```
/* Load App address into r1, replace with null
ldr r1, [r0, 0]
movs r2, 0
str r2, [r0, 0]
/* If TakeCell is empty (null) return */
cmp r1, 0
it eq
bx lr
/* Non-null: increment count */
ldr r2, [r1, 0]
add r2, r2, 1
str r2, [r1, 0]
/* Store App back to TakeCell */
str r1, [r0, 0]
bx lr
```

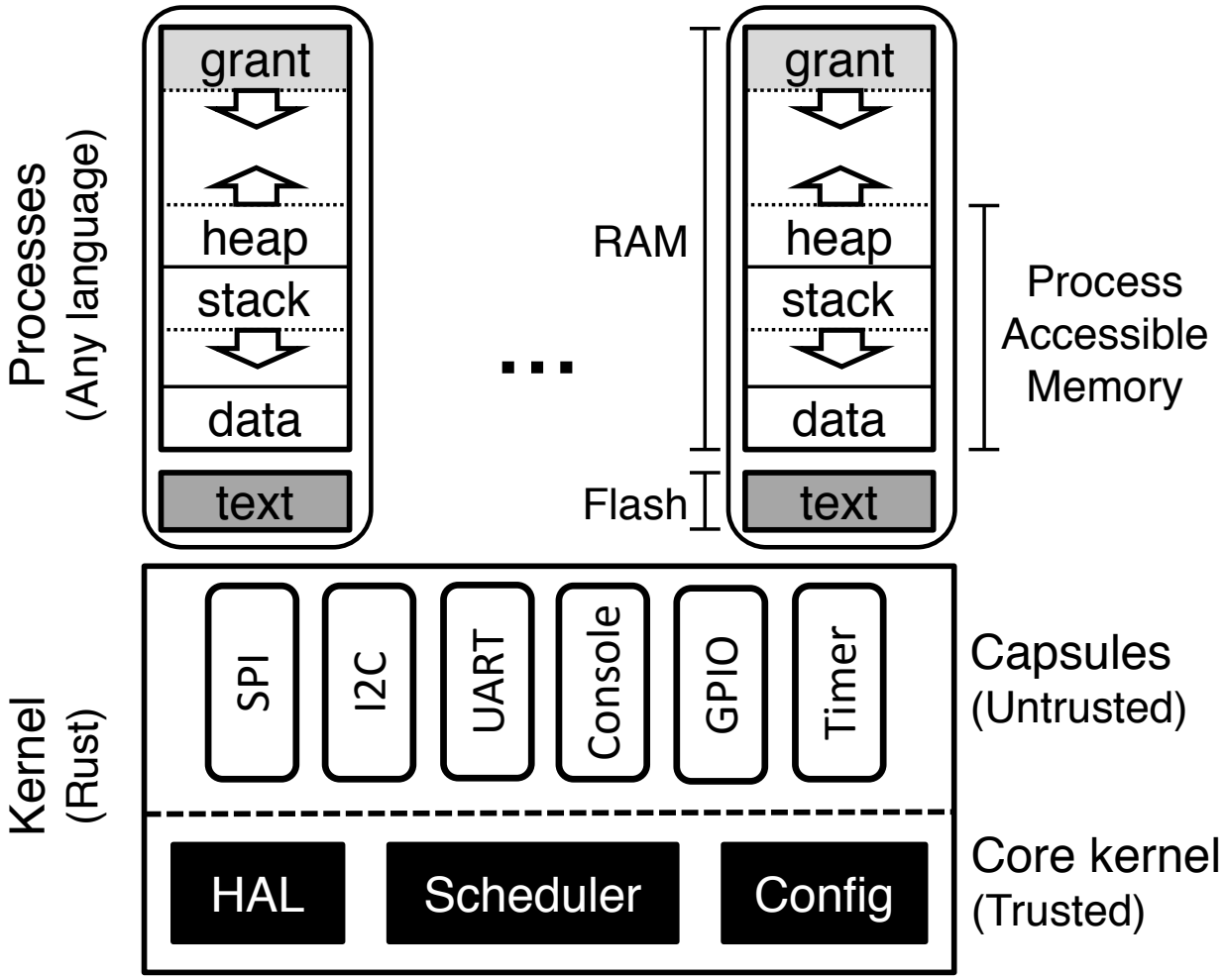
Resource Model

- Kernel is a single thread of control
  - All I/O operations are non-blocking
  - Interrupts enqueue events
  - System calls enqueue events
- Kernel has no heap
  - Prevent resource exhaustion from shared pool

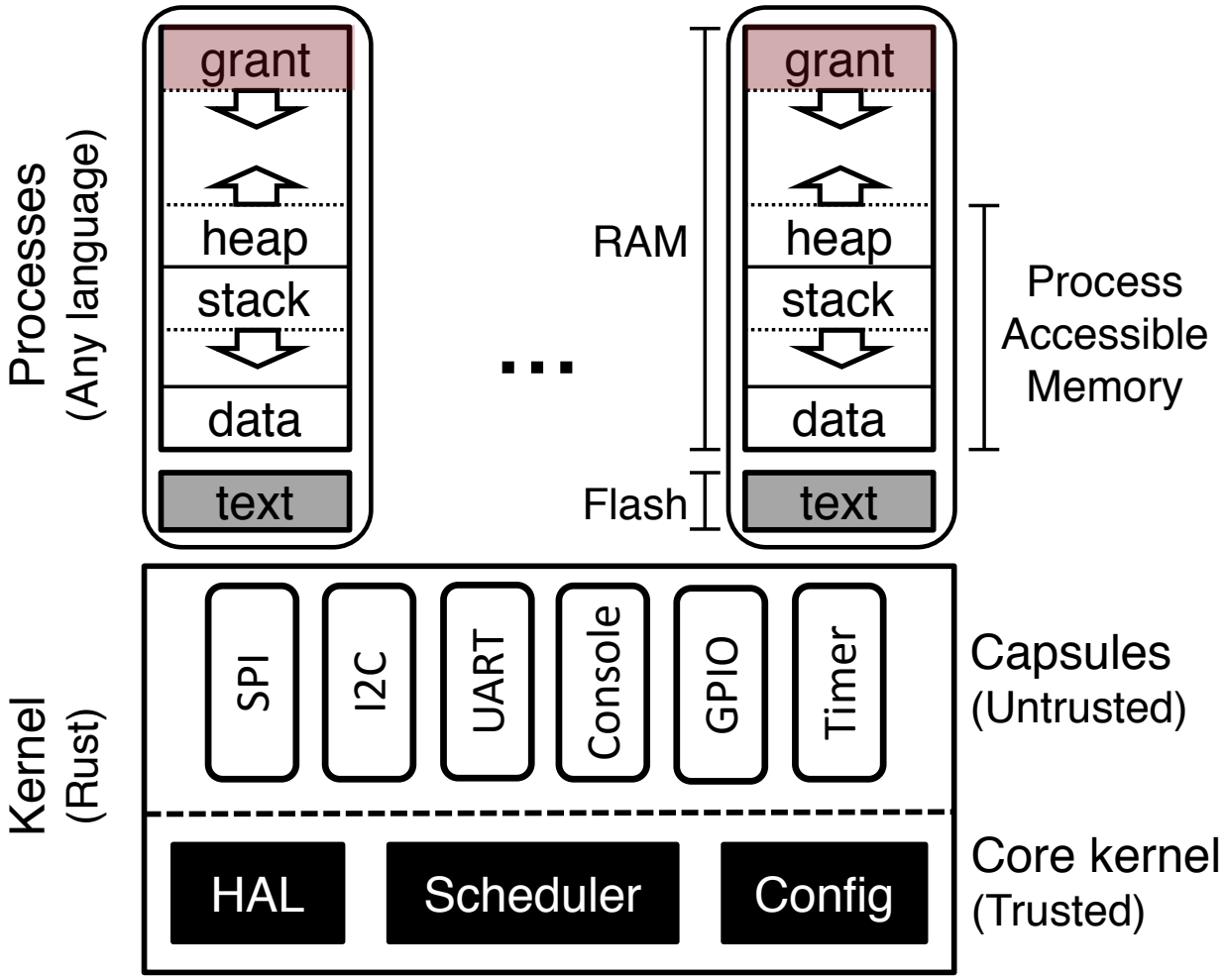
Problem 2: System Calls

- System calls need to dynamically allocate memory
  - Create a timer, kernel needs to keep timer's state
  - Enqueue a packet to send, kernel needs reference to packet
- Kernel can't dynamically allocate memory!
  - Otherwise a process can exhaust kernel memory
  - Fragmentation
  - Cleaning up after process failures

System Call Insight



System Call Insight

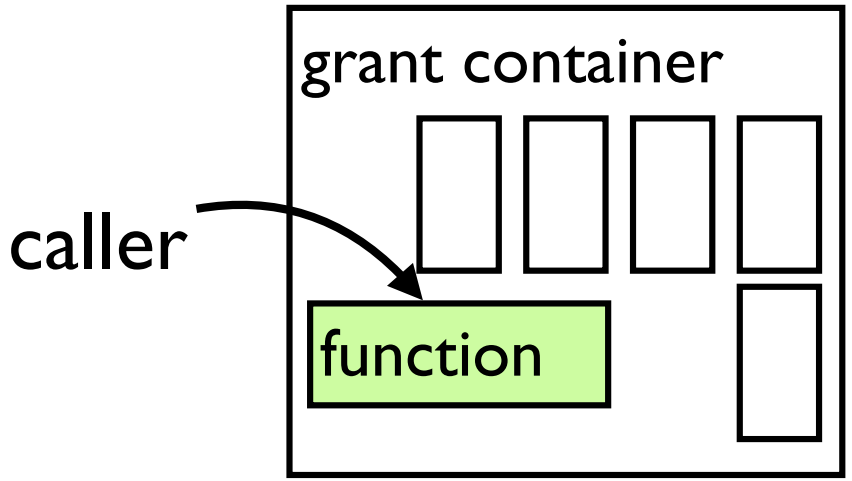


- Processes given block of memory
- Dynamically allocated when process loaded
- Kernel can allocate memory from process

Memory Grants

- Each process has a growable container of *grant memory*
- Kernel can allocate objects from the grant block
- References to objects cannot escape the block
  - Process failure/crash does not lead to dangling pointers
- Users pass a function to the container with `enter`

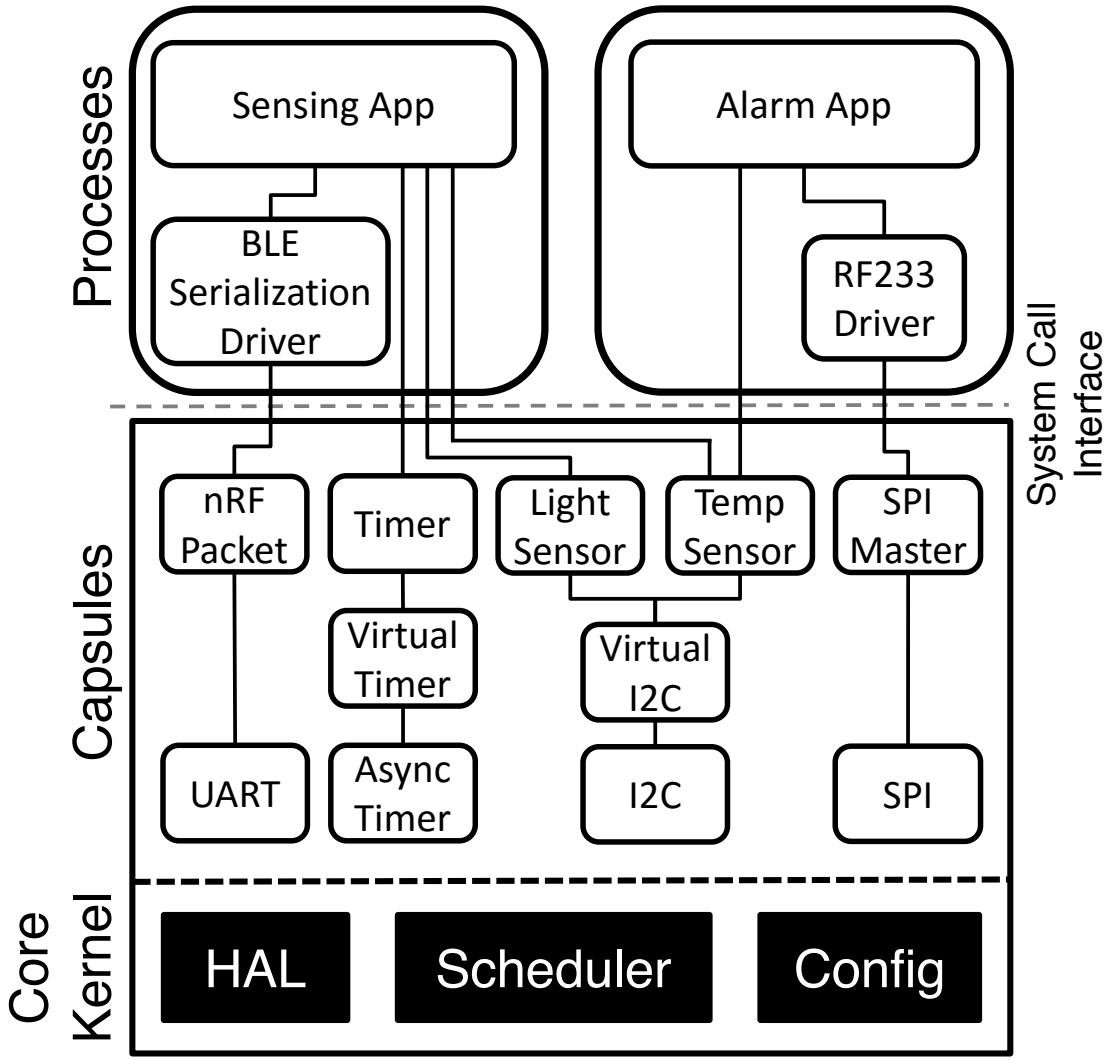
```
self.apps.enter(appid, |app, _| {
    app.read_buffer = Some(slice);
    app.read_idx = 0;
    0
}).unwrap_or(-1)
```



Outline

- Energy rules everything
- Kernel design goals
- Tock operating system design
  - Architecture: Rust, capsules, and processes
  - Rust challenges
  - Resource model: memory and CPU
- Some early observations

Drivers as Processes



Processes allow easy incorporation of existing library code and drivers: BLE serialization, 802.15.4 radio, etc.

Truly Non-Blocking

- Prior kernels claimed to be non-blocking
  - TinyOS
  - Contiki
  - SOS
- Used blocking calls for very short operations
  - E.g., write a single byte over the SPI
- CortexMs are 8x faster: the cycles for these blocking operations is now significant
  - Every state transition in an FSM is non-blocking…

RF233

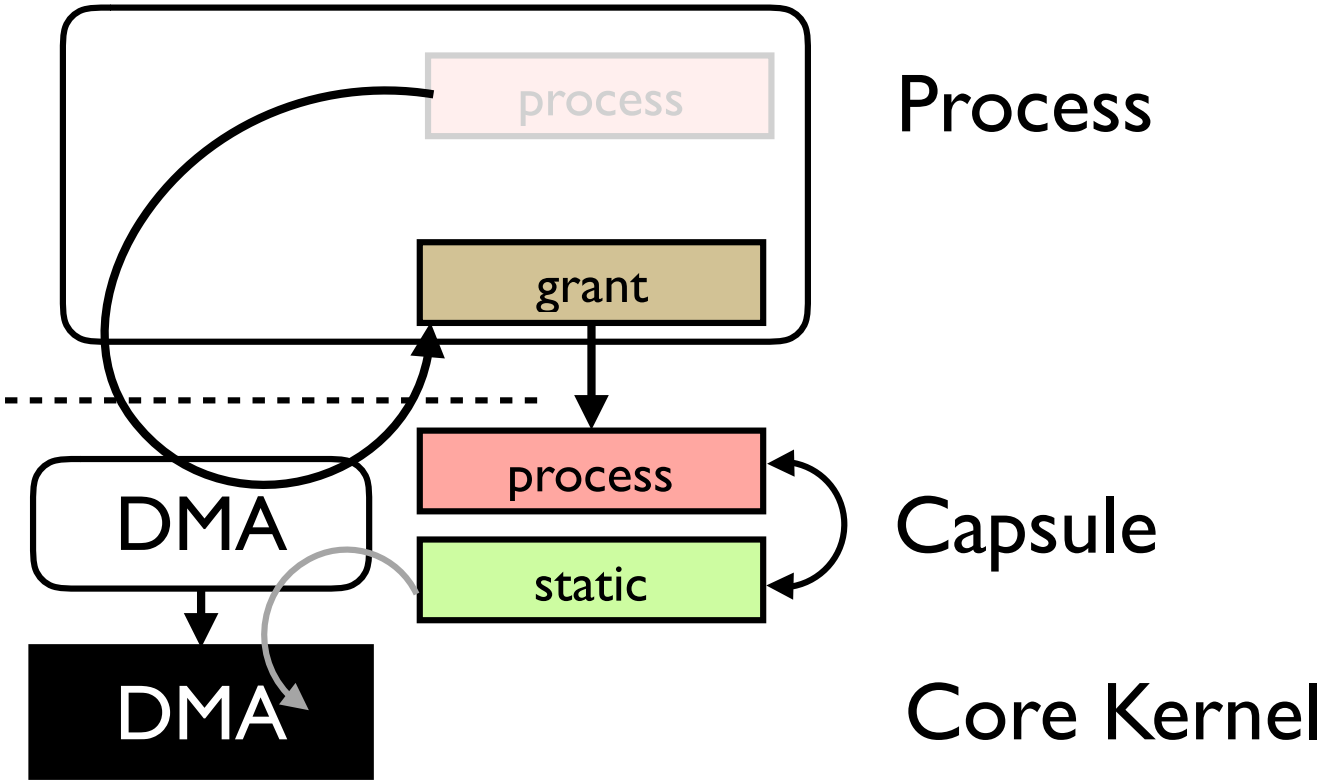
- 802.15.4 radio (used in imix and Nest Protect)
- On an interrupt, have to perform an SPI transaction to read what the source(s) of the interrupt are
  - You could be in the midst of an existing SPI transaction, such as reading the interrupt state
- Client calls can start complex series of SPI transactions
  - Sending a packet
  - Reconfiguring the radio
- End result: 64 states, plus 4 pending flags: transmitting, receiving, configuration, and interrupt

Lifetimes Force Discipline

- A Rust *lifetime* is how long a memory object exists
- If a reference R points to an object O, you must prove that the $O_{lifetime} > R_{lifetime}$: no dangling pointer
- Lifetime is inherently tied to basic blocks and the stack
- Hardware works differently…
- Question: what's the required lifetime of a buffer whose address you put into a DMA register?

DMA Engine

```
pub fn do_xfer(&self,
               pid: DMAPeripheral,
               buf: &'static mut [u8],
               len: usize) {
```



Tock Operating System

- Safe, multi-tasking operating system for memory-constrained devices
- Core kernel written in Rust, a safe systems language
  - Small amount of trusted code (can do unsafe things)
    - Rust bindings for memory-mapped I/O
    - Core scheduler, context switches
- Core kernel can be extended with *capsules*
  - Safe, written in Rust
  - Run inside kernel
- *Processes* can be written in any language (asm, C)
  - Leverage Cortex-M memory protection unit (MPU)
  - User-level, traps to kernel with system calls

Previous Systems

	No Resource Exhaustion	Memory Isolation	Memory Efficiency	Concurrency	Loadable Applications
Arduino			✓		
TinyOS			✓	✓	
SOS			✓	✓	✓
TOSThreads	✓			✓	✓
Contiki			✓	✓	✓
FreeRTOS				✓	
RiotOS			✓		

Previous Systems

	No Resource Exhaustion	Memory Isolation	Memory Efficiency	Concurrency	Loadable Applications
Arduino			✓		
TinyOS			✓	✓	
SOS			✓	✓	✓
TOSThreads	✓			✓	✓
Contiki			✓	✓	✓
FreeRTOS				✓	
RiotOS			✓		
Tock	✓	✓	✓	✓	✓

Thanks!

https://www.tockos.org/