Tock

A Safe Multi-tasking Operating System for Microcontrollers

Amit Levy
Branden Ghena Bradford Campbell Pat Pannuto
Nicholas Matsakis Prabal Dutta Philip Levis

July 28, 2016

Overview: Tock

- An **operating system** for microcontrollers
  - $< 50\mu A$ average current
  - $16KiB$-$512KiB$ memory
  - $O(1ms)$ timing constraints
- **Rust type system** isolates numerous kernel components
- **Hardware protection** isolates limited # of processes
- Resolves **isolation granularity** vs. **resource consumption**:
  - Single-threaded asynchronous event system
  - Type encapsulation for isolation

Microcontrollers Deserve Protection


ACCU-CHEK®
Aviva Combo
Last 8 hours
Overall
Back
Graph
Good
Mornin
nest
Sulzer Intermedics
ANGLETON, TEXAS
PHILIPS

Existing embedded ”operating systems” are not real operating systems

- No separation of core, drivers and applications.
- No isolation mechanisms
- ”OS” is just a library

Existing embedded ”operating systems” are not real operating systems

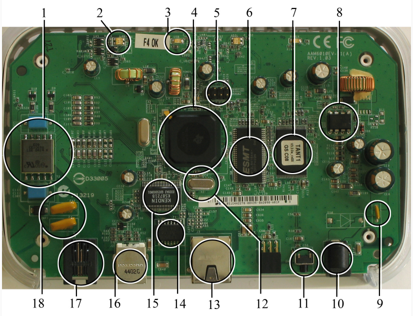
- No separation of core, drivers and applications.
- No isolation mechanisms
- ”OS” is just a library

Ruby on Rails for your defibrillator

How do we build embedded systems?

1. Build hardware platform

- Microcontroller
- Radio, buses...
- Sensors
- Actuators
- LEDs

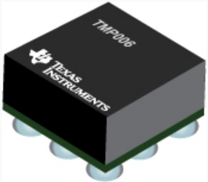


2. Choose an "OS"

- Arduino
- TinyOS
- FreeRTOS
- Atmel Software Framework, Nordic SDK...

3. 3rd-party drivers

- TMP006
- Bluetooth
- ZigBee
- IP networking



4. Build application on top

- Hand-rolled code
- Cryptography libraries
- Statistics/Machine learning
- PID control

5. Optimize

- Energy consumption
- Performance
- Memory usage

5. Optimize

- Energy consumption
- Performance
- Memory usage
- *!Security*

Embedded systems are built like other systems

Embedded systems are built like other systems

built from reusable components

Reusing components is a GOOD!

- Less engineering effort
- Fewer bugs overall
- Better interoperability
- …

Mixing code from various sources

Mixing code from various sources

+ No isolation mechanisms

Mixing code from various sources

+ No isolation mechanisms

+ Optimizing for performance

Mixing code from various sources

+ No isolation mechanisms

+ Optimizing for performance

= Recipe for disaster

Mixing code from various sources

+ No isolation mechanisms

+ Optimizing for performance

= Recipe for disaster

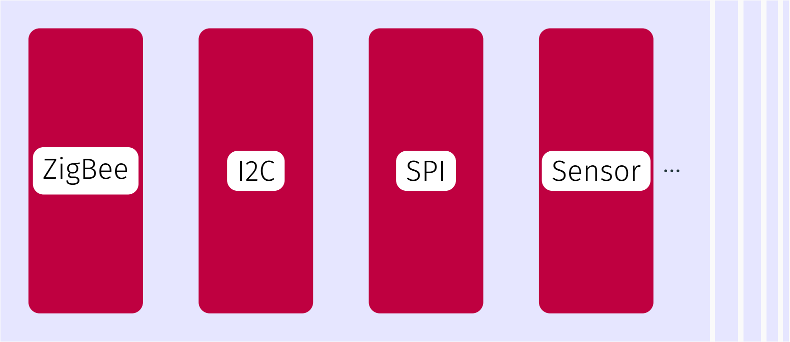
What happens when there is a bug in one of the components?

Outline

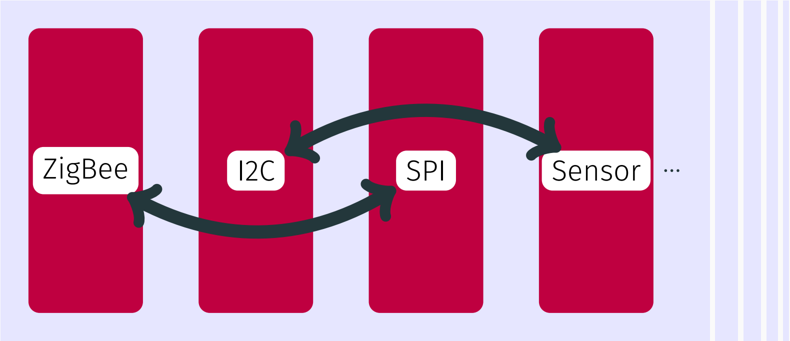
1. Why processes won’t work
2. Tock architecture
3. Capsules
4. Grants
5. Evaluation

Ownership is Theft

Process Isolation



Process Isolation



Why processes?

- Isolation
- Concurrency (parallelism)
- Good programming model
- Convenient to enforce

Why *not* processes?

Resource overhead

- Allocate memory for each process
- Context switch for communication

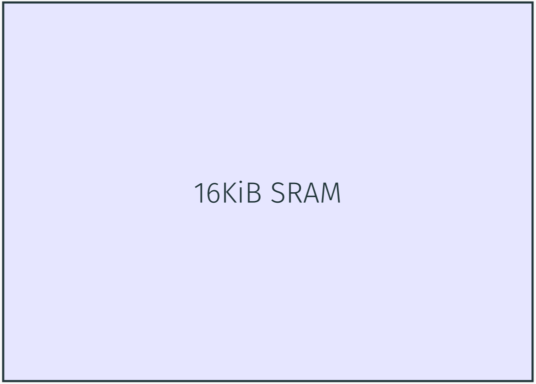
Why *not* processes?

Resource overhead

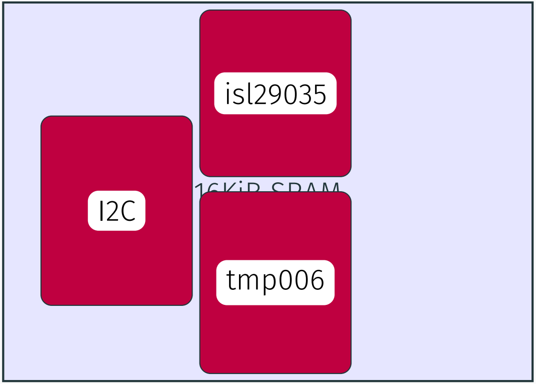
- Allocate memory for each process
- Context switch for communication

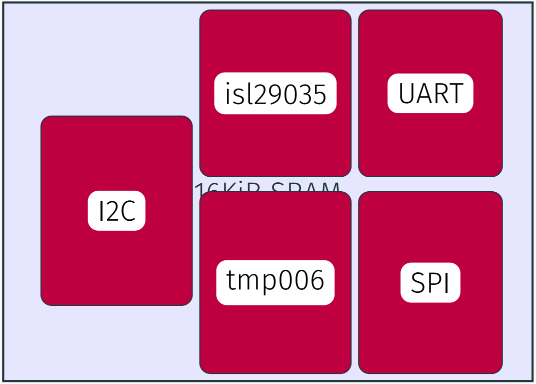
Tock is for resource constrained devices

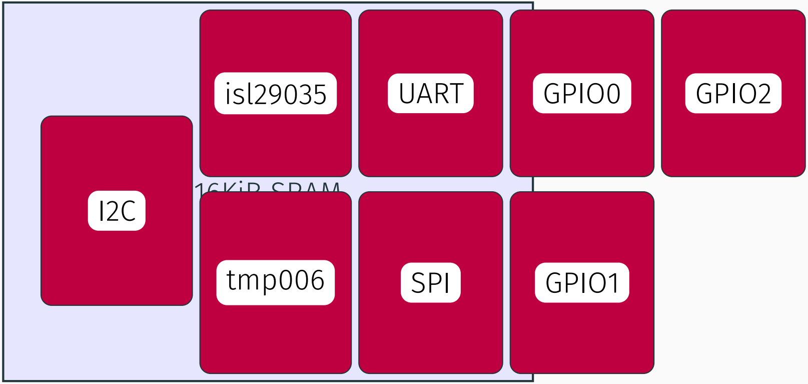
- 16KiB memory
- $O(1ms)$ timing constraints

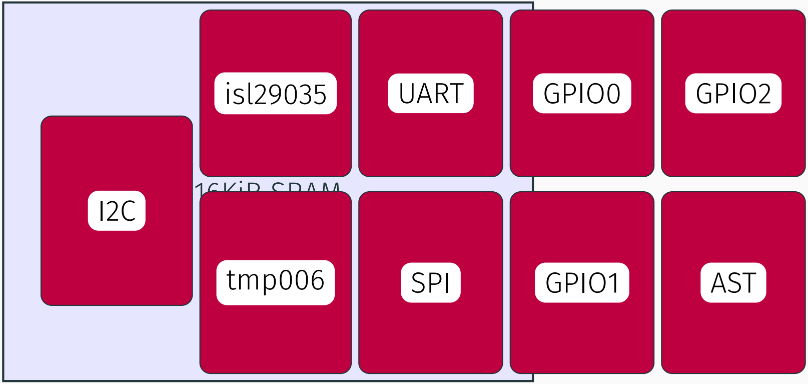

16KiB SRAM


I2C
16KiB SRAM


isl29035
I2C
tmp006


isl29035
UART
I2C
tmp006
SPI


I2C
isl29035
UART
GPIO0
GPIO2
tmp006
SPI
GPIO1


I2C
isl29035
tmp006
UART
SPI
GPIO0
GPIO1
GPIO2
AST

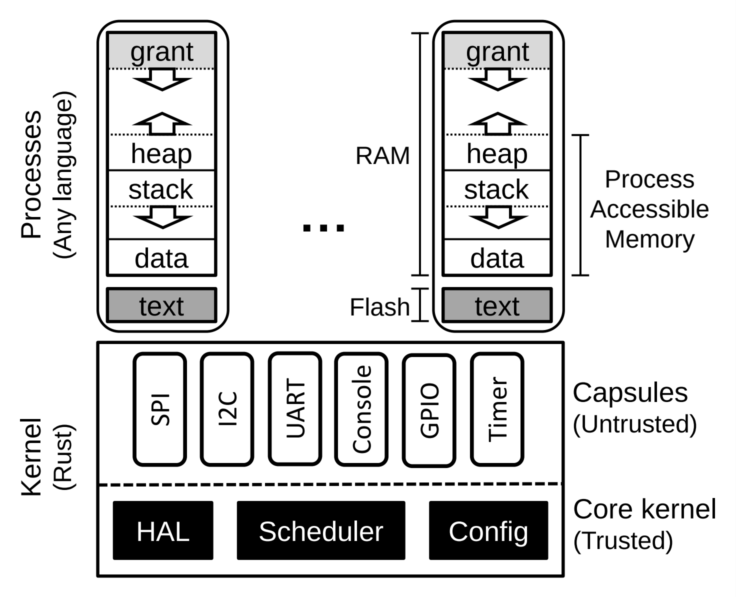
Tradeoff *granularity* for *resources*

Architecture

Challenge: How do we isolate concurrent components *without* incurring a memory/performance overhead for each component?

Challenge: How do we isolate concurrent components *without* incurring a memory/performance overhead for each component?

Key idea: Use a single-threaded event system and isolate using the Rust type and module system


Processes
(Any language)
grant
heap
stack
data
text
...
RAM
Flash
grant
heap
stack
data
text
Process
Accessible
Memory
Kernel
(Rust)
SPI
I2C
UART
Console
GPIO
Timer
Capsules
(Untrusted)
HAL
Scheduler
Config
Core kernel
(Trusted)

Kernel Design

Event-based concurrency:

- Enqueue all hardware interrupts & poll
- Never block on I/O
- Communicate between components with function calls
- ”Static” callback binding

Kernel Design

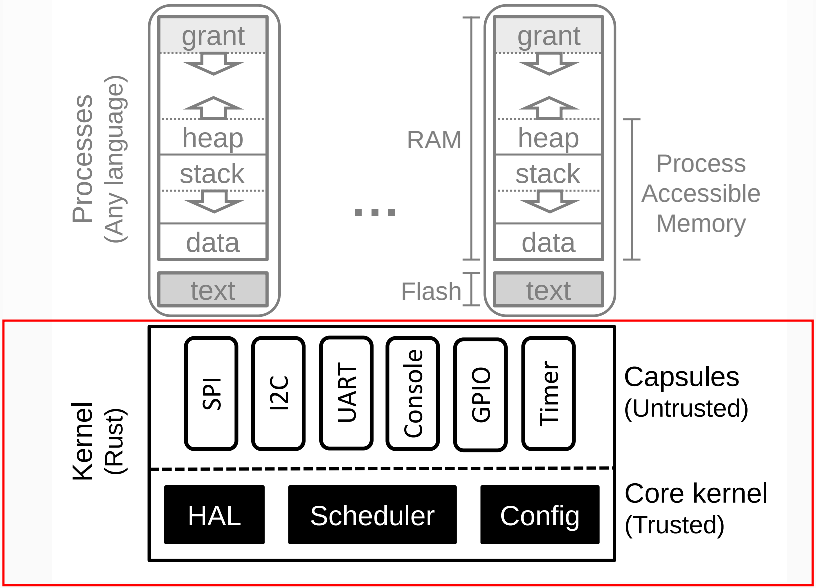
Event-based concurrency:

- Enqueue all hardware interrupts & poll
- Never block on I/O
- Communicate between components with function calls
- ”Static” callback binding

Isolation and safety from Rust

- Type-safe
- No garbage collection
- ”Zero-cost” abstractions

Capsules


Processes
(Any language)
grant
heap
stack
data
text
...
RAM
Flash
grant
heap
stack
data
text
Process
Accessible
Memory
Kernel
(Rust)
SPI
I2C
UART
Console
GPIO
Timer
Capsules
(Untrusted)
HAL
Scheduler
Config
Core kernel
(Trusted)

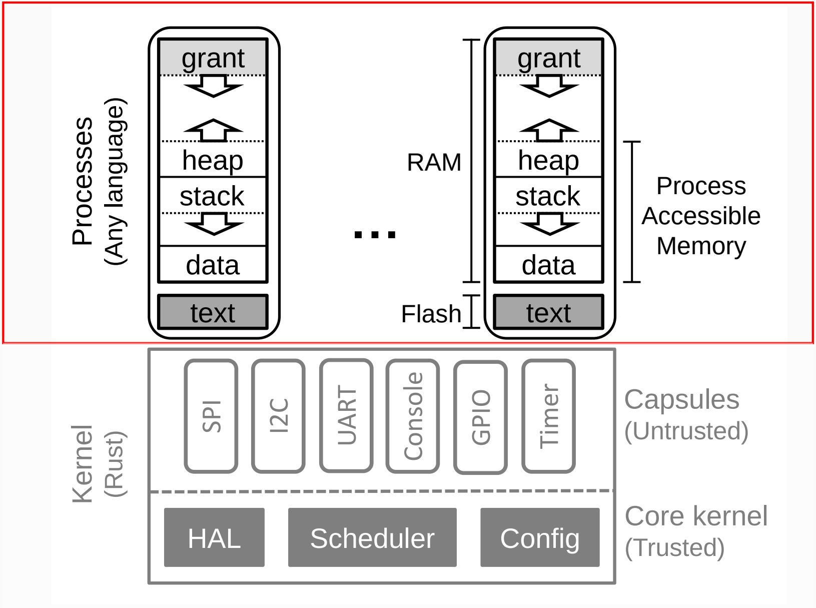
```
mod light_sensor {
  pub struct LightSensor {
      i2c: &I2CDevice,
      state: State,
      buffer: &[u8],
      callback: Option<Callback>,
  }

  impl LightSensor {
    pub fn start_read_lux(&self) { ...  }
  }

  impl I2CClient for LightSensor {
    fn command_complete(&self, buffer: &[u8]) { ... }
  }
}
```

Capsules are *untrusted* for access but *trusted* for liveness.

Dynamic Memory with Grants


Processes
(Any language)
grant
heap
stack
data
text
...
RAM
Flash
grant
heap
stack
data
text
Process
Accessible
Memory
Kernel
(Rust)
SPI
I2C
UART
Console
GPIO
Timer
Capsules
(Untrusted)
HAL
Scheduler
Config
Core kernel
(Trusted)

- No heap in the kernel
- But capsules must allocate memory for process requests
- Remember: single-threaded execution

Grant Regions

- Process-specific kernel-heap
- Not accessible to process
- Capsules can allocate there dynamically
- Deallocation on process exit is $O(1)$

Grant Regions

Need to enforce three invariants:

1. Allocated memory does not allow capsules to break the type system.
2. Capsules can only access pointers to process memory while the process is alive.
3. The kernel must be able to reclaim memory from terminated process.

Key Challenge

Processes can die and their memory needs to be reclaimed dynamically.

Rust determines memory reclamation statically.

We can use **type system** to enforce **simple properties** that interact with the **system architecutre** to achieve **higher-level safety goals**.

```rust
impl<T: Default> Grant {
  fn enter<F,R>(&self, appid: AppId, func: F)
    -> Result<R, Error> where
    F: for<'b> FnOnce(&'b mut Owned<T>, &'b mut Allocator)
      -> R, R: Copy
}

impl Allocator {
  fn alloc<T>(&mut self, data: T) -> Result<Owned<T>, Error>
}

struct Owned<T: ?Sized> { data: Unique<T>, app_id: AppId }
impl Drop, Deref, DerefMut for Owned { ... }
```

What do we know:

1. `'b` lifetime is existential
2. `Allocator` and `Owned` do not implement `Copy`
3. `Allocator` and `enter` are the only way to create an `Owned` type.

What do we know:

1. `'b` lifetime is existential
2. `Allocator` and `Owned` do not implement `Copy`
3. `Allocator` and `enter` are the only way to create an `Owned` type.

`Owned` types can never escape the closure passed to `enter`.

`Owned` types can never escape the closure passed to `enter`.

`Owned` types can never escape the closure passed to `enter`.

When the process scheduler is executing, all capsules have returned.

`Owned` types can never escape the closure passed to `enter`.

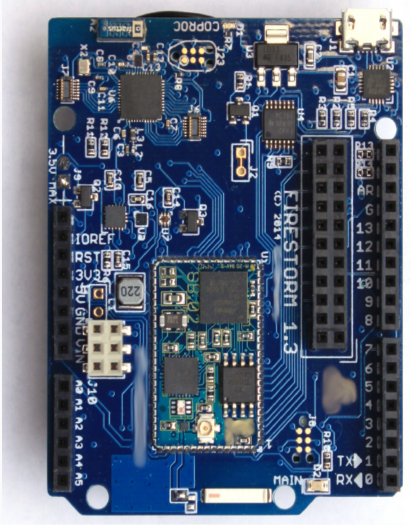
When the process scheduler is executing, all capsules have returned.

When a process dies, we can reclaim all of it’s grants immediately, since no references can be outstanding!

Evaluation

Firestorm Platform

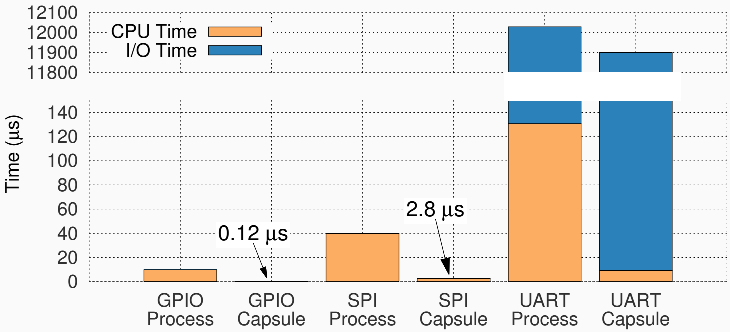
- Atmel SAM4L Cortex-M4
  - 64KiB SRAM
  - 512KiB flash
  - 48Mhz
  - USARTs, SPI, I2C, USB, LCD, AES...
- Bluetooth Low Energy, 802.15.4
- Light, temperature, acceleration



Firestorm Platform

- >100 capsule instances
  - e.g. for each of 75 GPIO pins
- $7KiB$ memory
- $30KiB$ flash
- 7 processes with $8KiB$ memory each
- Drivers for BLE & 802.15.4 in processes

Capsule Operations are Cheap

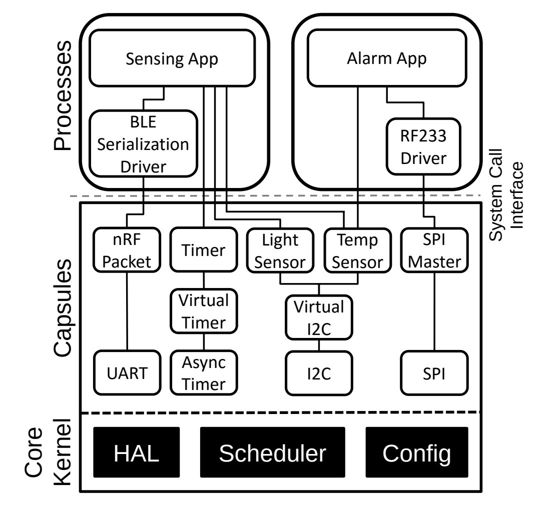


Capsule Operations are Cheap

Event Source	Core Kernel	Capsule	Process
GPIO Input	0.623 μs	8.54 μs	33.4 μs
Timer Expiration	0.623 μs	8.67 μs	36.8 μs

Operation	CPU Cycles
Switch to kernel	111
Call capsule	83
Switch back to process	146
Total	340

Case Study: Sensing Application



Conclusion

We didn't discuss

- Design principles for a **safe** TCB interface?
- Design principles for a **secure** TCB interface?
- Challenges using an ownership type system
    - *Existential Types for Imperative Languages*
- Syscall interface
- User space concurrency model

Limitations

- Capsules are trusted for liveness
- Won’t work with shared-memory multiprocessors
- Trusted configuration module for each platform
- IPC, dynamic reprogramming, multi-SoC platforms
- Potential benefits from type-safe processes

Summary

- Embedded systems growing in complexity
- Providing isolation and safety is critical
- Current OSs inadequate
- Tock:
    - Prioritizes safety by keeping TCB small
    - Leverages language & hardware mechanisms
    - Memory grants to allow safe dynamic allocation
- Tradeoffs between granularity, concurrency and safety