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
  - < 50μA 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
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?
1. Why processes won’t work
2. Tock architecture
3. Capsules
4. Grants
5. Evaluation
Ownership is Theft
Process Isolation

ZigBee  I2C  SPI  Sensor  ...
Process Isolation

- ZigBee
- I2C
- SPI
- Sensor
...
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
16KiB SRAM
I²C
isl29035
UART
GPIO0
GPIO2
tmp006
SPI
GPIO1
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)

Kernel (Rust)

Core kernel (Trusted)

Capsules (Untrusted)

Process Accessible Memory

RAM

Flash

Heap

Stack

Text

Data

Grant

Text

Data

Grant

Heap

Stack
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)

Kernel (Rust)

Core kernel (Trusted)

Capsules (Untrusted)

- SPI
- I2C
- UART
- Console
- GPIO
- Timer

- HAL
- Scheduler
- Config

memory:
- heap
- stack
- data
- grants

RAM
Flash

Process Accessible Memory
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
• 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.
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 architecture to achieve higher-level safety goals.
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 \texttt{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 its 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

<table>
<thead>
<tr>
<th>Event Source</th>
<th>Core Kernel</th>
<th>Capsule</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIO Input</td>
<td>0.623 μs</td>
<td>8.54 μs</td>
<td>33.4 μs</td>
</tr>
<tr>
<td>Timer Expiration</td>
<td>0.623 μs</td>
<td>8.67 μs</td>
<td>36.8 μs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>CPU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch to kernel</td>
<td>111</td>
</tr>
<tr>
<td>Call capsule</td>
<td>83</td>
</tr>
<tr>
<td>Switch back to process</td>
<td>146</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>340</strong></td>
</tr>
</tbody>
</table>
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