Tock Operating System

Safety without Processes for Embedded Systems

Amit Levy 1  Brandon Ghena 2  Michael Andersen 3
Brad Campbell 2  Gabe Fierro 3  Pat Pannuto 2
Prabal Dutta 2  David Culler 3  Philip Levis 1

1Stanford University
2University of Michigan
3University of California, Berkeley
Tock is a safe operating system for embedded systems,
Tock

Tock is a safe operating system for embedded systems, designed for low resource consumption:
Tock is a safe operating system for embedded systems, designed for low resource consumption:

- 16KB-512KB memory
- Sub-1mA average current draw
- Order of millisecond timing constraint (O(10000 cycles))
Tock is a safe operating system for embedded systems, designed for low resource consumption:

- 16KB-512KB memory
- Sub-1mA average current draw
- Order of millisecond timing constraint (O(10000 cycles))

with a central focus on isolating untrusted components.
Tock is a safe operating system for embedded systems, designed for low resource consumption:

- 16KB-512KB memory
- Sub-1mA average current draw
- Order of millisecond timing constraint (O(10000 cycles))

with a central focus on isolating untrusted components achieved (primarily) by using a safe language to isolate components in the kernel
Embedded "operating systems" exist (TinyOS, FreeRTOS, Arduino, etc)
Embedded "operating systems" exist (TinyOS, FreeRTOS, Arduino, etc)

But they’re not operating systems like you’re used to:
Embedded "operating systems" exist (TinyOS, FreeRTOS, Arduino, etc)

But they’re not operating systems like you’re used to:

No strict separation between a kernel, drivers and applications.
Embedded "operating systems" exist (TinyOS, FreeRTOS, Arduino, etc)

But they’re not operating systems like you’re used to:

No strict separation between a kernel, drivers and applications.
No mechanism for isolating components from each other
Embedded “operating systems” exist (TinyOS, FreeRTOS, Arduino, etc)

But they’re not operating systems like you’re used to:
No strict separation between a kernel, drivers and applications.
No mechanism for isolating components from each other.
The ”OS” is basically a library.
Embedded "operating systems" exist (TinyOS, FreeRTOS, Arduino, etc)

But they’re not operating systems like you’re used to:
No strict separation between a kernel, drivers and applications.
No mechanism for isolating components from each other.
The "OS" is basically a library.
Think Ruby on Rails for your defibrillator.
How do we build embedded systems?
1. Build a platform

- MCU
- Radio
- Sensors
- Actuators
1. Build a platform

- MCU
- Radio
- Sensors
- Actuators

Each platform is a unique snowflake
2. Choose an "OS"

- Arduino
- TinyOS
- FreeRTOS
3. Pull in drivers for the platform

- Bluetooth driver from Nordic
- 802.15.4 driver from Thingsquare
- Temperature sensor driver from Adafruit
4. Build application(s) on top
5. Optimize for !security

Often modifications to the whole stack to get better performance and energy consumption
Embedded systems are a lot like other systems
Embedded systems are a lot like other systems 
i.e. built from reusable components
This is a recipe for disaster

Mixing code from various sources
This is a recipe for disaster

Mixing code from various sources

+ No isolation mechanisms
This is a recipe for disaster

Mixing code from various sources

+ No isolation mechanisms
  + Optimizing for performance
This is a recipe for disaster

Mixing code from various sources

+ No isolation mechanisms

  + Optimizing for performance

  = Bugs, exploits, meyham
Reusing components is a GOOD thing!

- Less engineering effort
- Fewer bugs overall
- Better interoperability
- Don’t roll your own crypto
- ...

What happens when there is a bug?
Isolation in operating systems

Typically achieved with a thread/process-like abstraction:
Isolation in operating systems

Typically achieved with a thread/process-like abstraction:

- Servers in microkernels
- SIPs in Singularity
- HiStar, Docker, etc...
- Hails, Aeolus, etc...
Why processes?

Provides isolation
Why processes?

Provides isolation

Provides concurrency and parallelism
Why processes?

Provides isolation

Provides concurrency and parallelism

Convenient to enforce using hardware or language
Why *not* processes?

Each process needs its own stack and heap.
Why *not* processes?

Each process needs its own stack and heap.

**Internal fragmentation**: preallocate maximum memory for each process.
Why not processes?

Each process needs its own stack and heap.

**Internal fragmentation**: preallocate maximum memory for each process

**External fragmentation**: dynamically allocate blocks
Why *not* processes?

Each process needs its own stack and heap.

**Internal fragmentation**: preallocate maximum memory for each process

**External fragmentation**: dynamically allocate blocks

Interaction between components requires communication (message passing, RPC...)

Tradeoff granularity for resources
What if we give up concurrency?

We can isolate components, but we can't meet timing requirements.
What if we give up concurrency?

We can isolate components, but we can’t meet timing requirements
Tock is for resource-constrained devices

Microcontrollers often have as little as 16KB of memory
Tock is for resource-constrained devices

Microcontrollers often have as little as 16KB of memory

Timing constraints on the order of a few thousand cycles (apprx 1ms)
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 language mechanisms

- Module boundaries
- Strong encapsulation (hidden constructors)
- etc...
An event system:

- Enqueue all hardware interrupts
- Never block on I/O, instead separate into events
- Deliver results to higher layers through callbacks
An event system:

- Enqueue all hardware interrupts
- Never block on I/O, instead separate into events
- Deliver results to higher layers through callbacks

Built in Rust: type-safe with no runtime a “zero-cost” abstractions

Rust manages memory using affine types (ownership) instead of garbage collection
Small TCB*:

- Hardware abstraction layer (maps I/O registers into types)
- Platform tree
- Event scheduler
Tock Design

Small TCB*: 

- Hardware abstraction layer (maps I/O registers into types)
- Platform tree
- Event scheduler

Most complex components are isolated:

- Peripheral drivers
- Virtualization layers (timers, bus virtualization)
- Applications
Dealing with mutability

Mutability and circular dependencies don’t mix well:

• Unsafe type coercion
• Use after free
• Iterator invalidation
Dealing with mutability

Mutability and circular dependencies don’t mix well:

- Unsafe type coersion
- Use after free
- Iterator invalidation

Solution: Only allow mutability in controlled ways:

- Copy-in/copy-out
- One user-at a time (no "internal" mutability)
Dealing with mutability

Mutability and circular dependencies don’t mix well:

- Unsafe type coercion
- Use after free
- Iterator invalidation

**Solution**: Only allow mutability in controlled ways:

- Copy-in/copy-out
- One user-at a time (no ”internal” mutability)

Enforced using Rust’s ”immutable” references
In progress implementations for two platforms:

**Firestorm**
- SAM4L - 64KB memory
- 802.15.4 and BLE radios
- Sensors - temperature, accelerometer, light intensity

**NRF51822**
- 16KB memory
- Bluetooth low energy system-on-a-chip
Tock Implementation

Kernel is <10K lines of Rust  100 lines of assembly

Requires  6KB memory to include all components
Tock Implementation

Kernel is <10K lines of Rust 100 lines of assembly

Requires 6KB memory to include all components

Performance numbers forthcoming...
Tock also supports a limited number of processes.

- Applications in C
- Legacy drivers

About 8 slots on the SAM4L
Limitations

Legacy code needs to be ported to Rust or take up one of a few processes

Concurrency model does not support parallelism
Conclusion

- Embedded systems need isolation mechanisms
- Traditional mechanisms not appropriate
  - Processes: memory overhead
  - Non-concurrent: timing constraints
- Tock is a single-threaded event system
  - Low/no overhead per component
  - Retains concurrency