Allocating Dynamic Kernel Memory in Low-Memory Microcontrollers

COS 316: Principles of Computer System Design

Amit Levy & Ravi Netravali

Microcontrollers Becoming Platforms

- Fitness watches support different activities
- USB security keys perform multiple functions
  - U2F, SSH, GPG, HOTP
- Sensor networks run several experiments at once

Embedded Software Isn't Ready

- Run all code in a single address space
- Trust all code
- Can't update components
- Can't recover components

TinyOS

freeRTOS

Contiki

The Open Source OS for the Internet of Things

arm MBED

Safe Multiprogramming by Isolating Applications and OS Services

Can't Use Normal Isolation Techniques

- Limited memory: ***64 kB*** of RAM
  - Memory isolation techniques limit granularity
  - `malloc` can fail!
- No page virtualization
  - Instead protection bits for 8 memory regions
- Moore's Law doesn't fix the problem
  - Sleep current is limiting factor
  - Memory capacity < 10x in 15 years

Microcontrollers demand new multiprogramming abstractions.

How to Multiprogram a Microcontroller

- Use *type safety* to isolate most of the system
- Use memory isolation sparingly
  - Preemtive scheduling
  - Recover or update components at runtime
- Support dynamic workloads without `malloc`

Tock

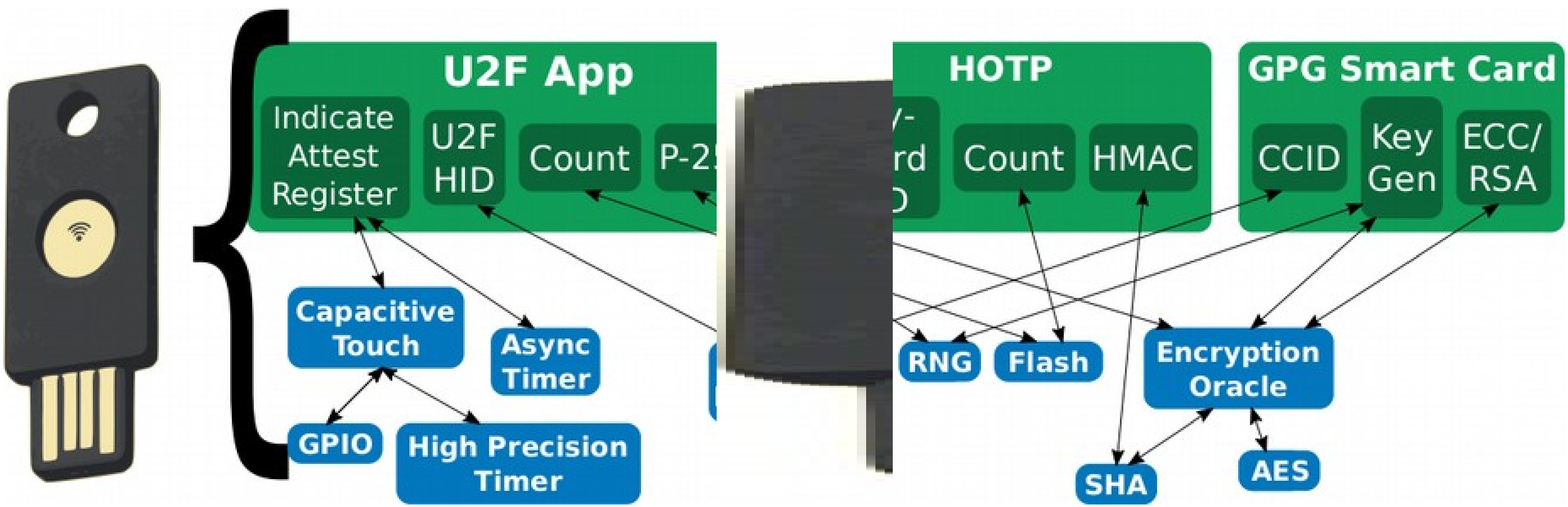
- Kernel written in Rust
- Processes abstraction using Memory Protect Unit (MPU)
- *Grants*: mechanism to account for dynamic workloads

Outline

1. Security Model & Design Principles
2. Two Isolation Mechanisms
3. Grants

Security in a Multiprogrammable MCU

Let's consider a programmable USB security key

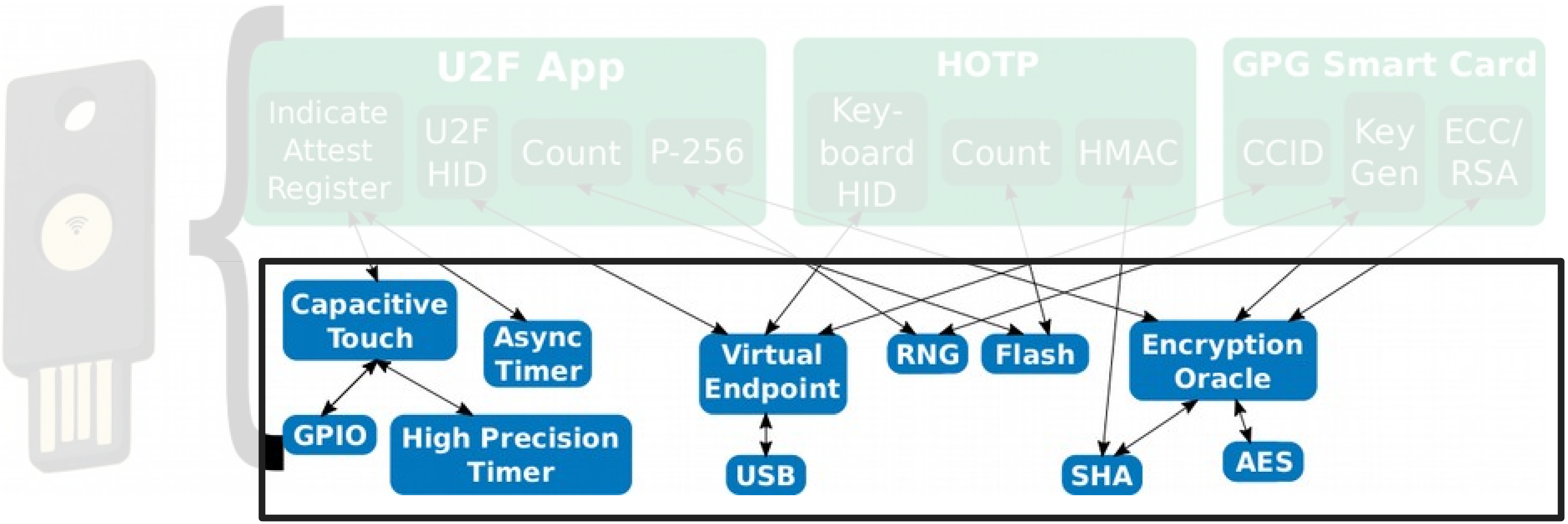


Board Integrators

- Build the hardware
- Combine core kernel, MCU-specific glue code & drivers
- Complete control over firmware

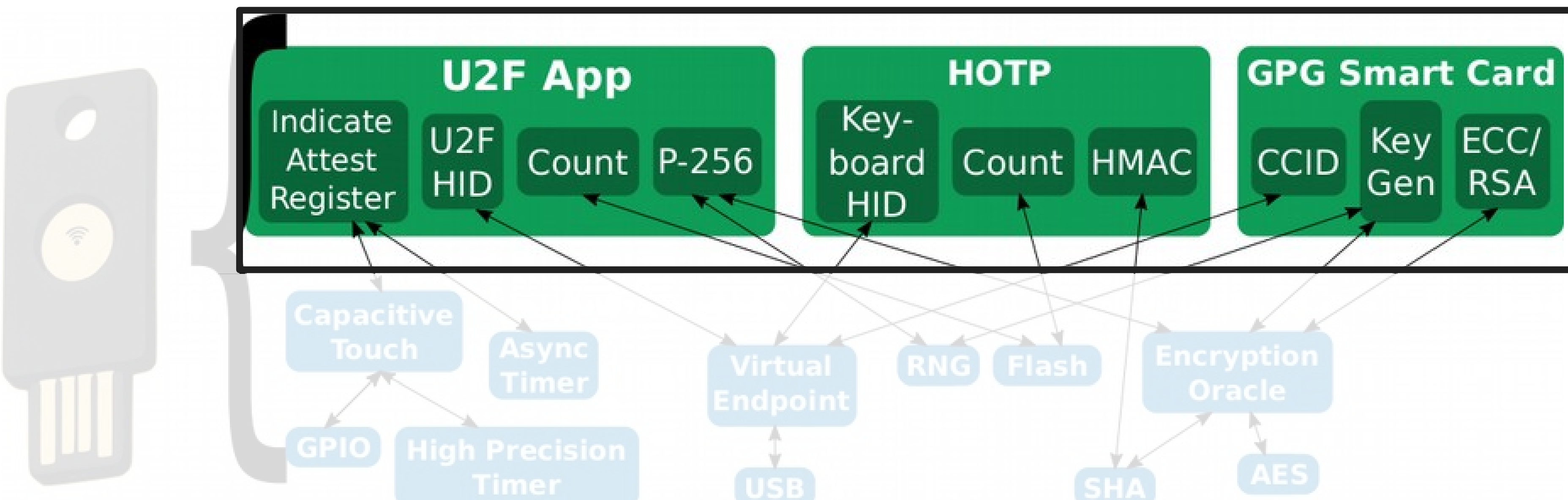
Kernel Component Developers

- Build most kernel functionality
- Source code available to board integrators
- But auditing won't catch all bugs



Application Developers

- Implement end-user functionality
- “Third-party” developers: unknown to board integrators
- Modeled as *malicious*



Design Principles

- **Isolation guarantees should be clear**
  - What *exactly* can a component do?
- **System should be dependable**
  - Unanticipated runtime behavior shouldn't cause crashes
- **Maximize concurrency**
  - I/O operations can overlap
- **Minimize resource consumption**
  - Resources don't dictate isolation granularity
- **Maximize programmability**
  - Applications will have unknown behavior

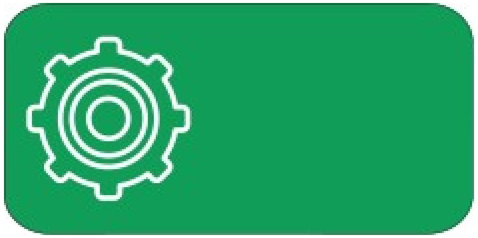
Tock’s Two Isolation Models

Capsules

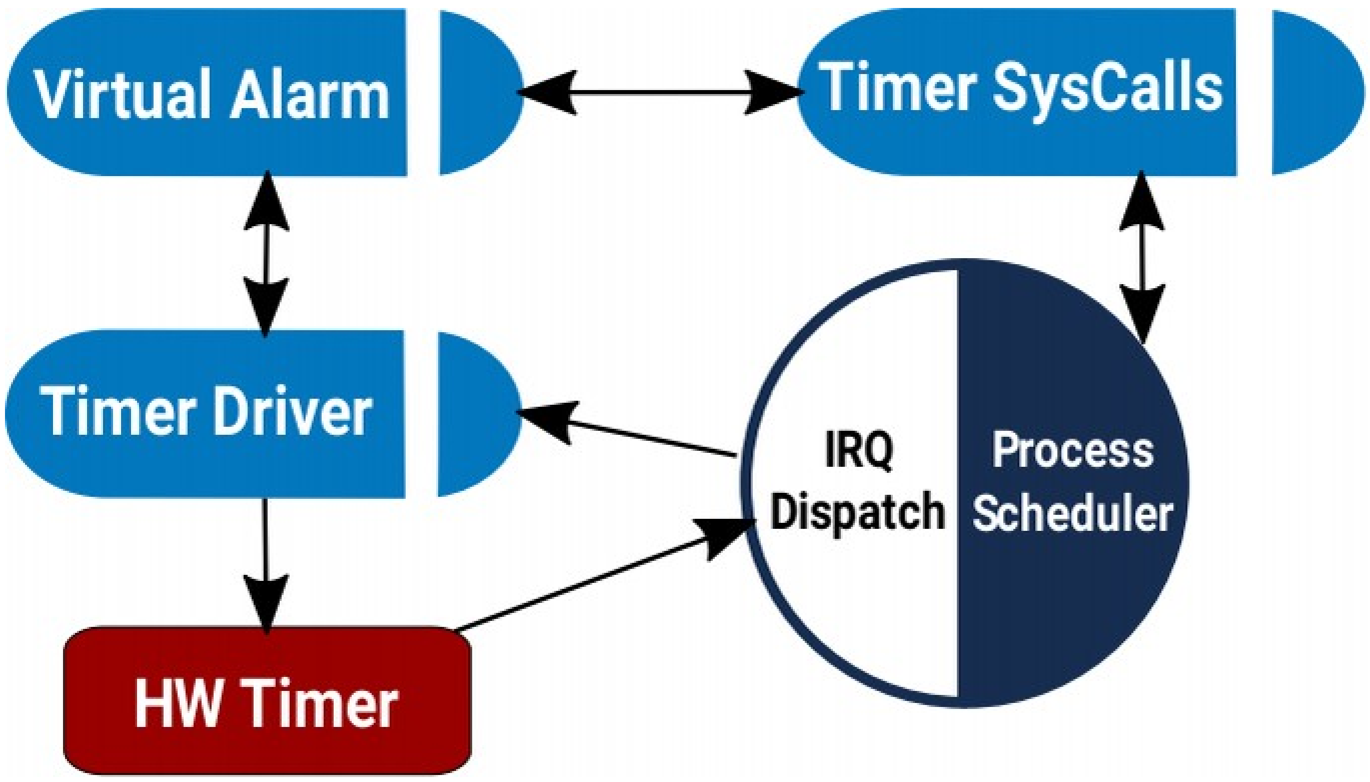
- Compile-time
- Kernel
- Limited trust
- Fine grained

Processes

- Runtime
- Applications
- Potentially malicious
- Coarse grained



Capsules



- A Rust module and structs
- Event-driven execution
- Communicate via references & method calls

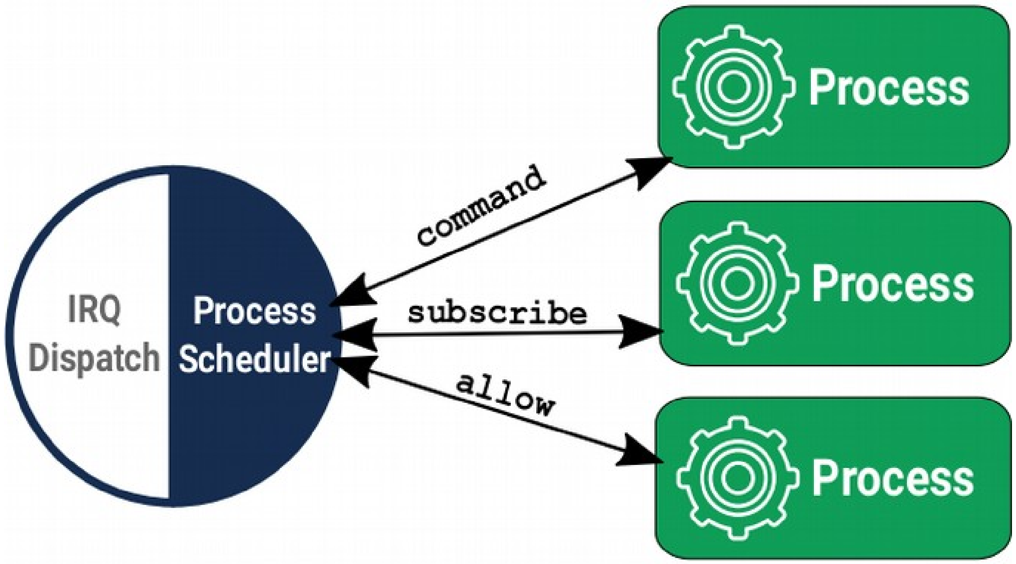
Capsule Isolation

```
struct DMAChannel {
    length: u32,
    base_ptr: *const u8,
}

impl DMAChannel {
  fn set_dma_buffer(&self, buf: &'static [u8]) {
    self.length = buf.len();
    self.base_ptr = buf.as_ref();
  }
}
```

- Exposes the DMA base pointer and length as a Rust *slice*[*]
- Type-safety guarantees user has access to memory

Processes

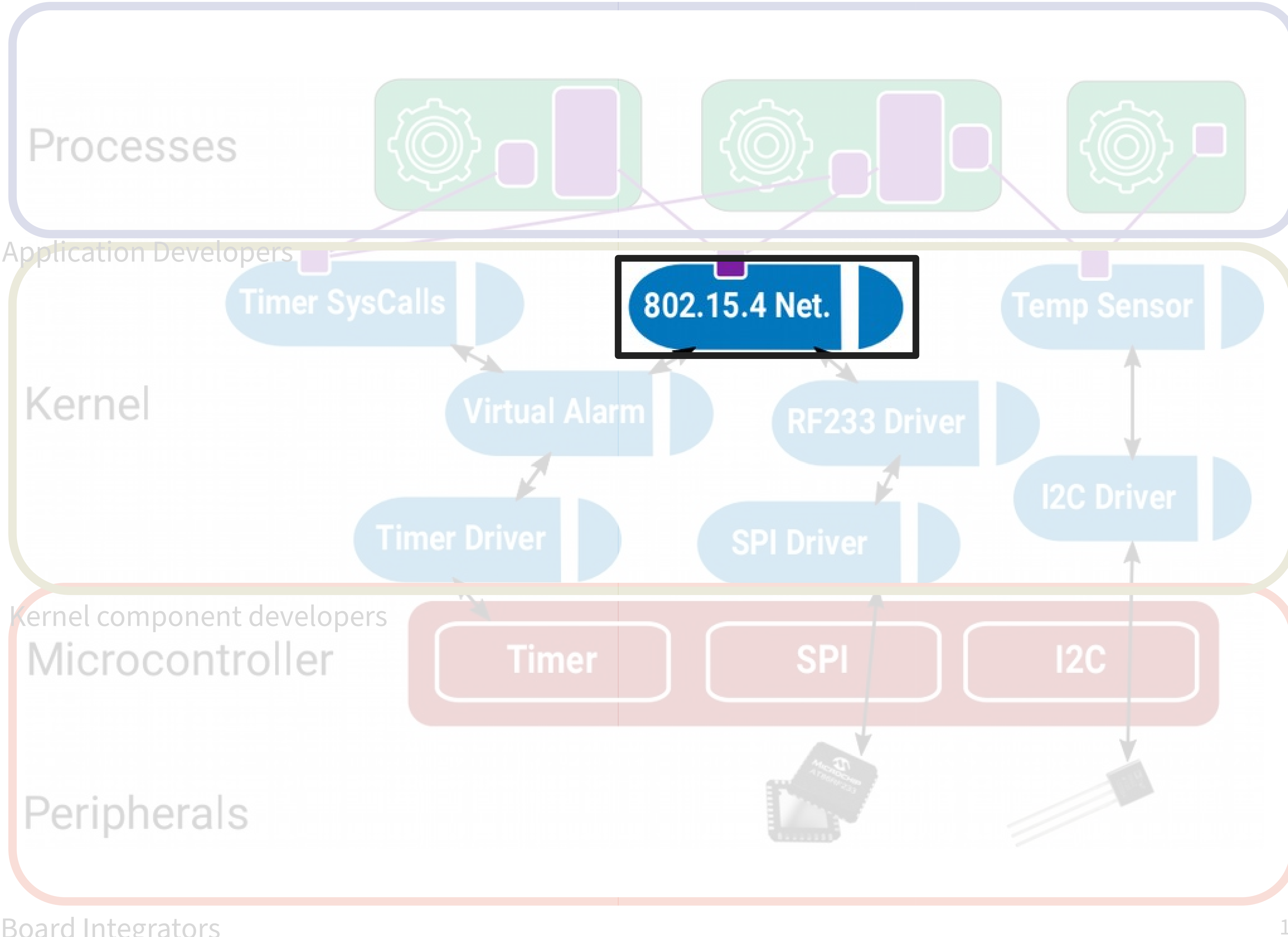


- Hardware-isolated concurrent executions of programs
  - Logical memory region: stack, heap, static variables
  - Uses the ARM *Memory Protection Unit* (MPU) to protect memory regions without virtualization
- Scheduled preemptively
- System calls & IPC for communication
- Updated dynamically

Processes vs. Capsules

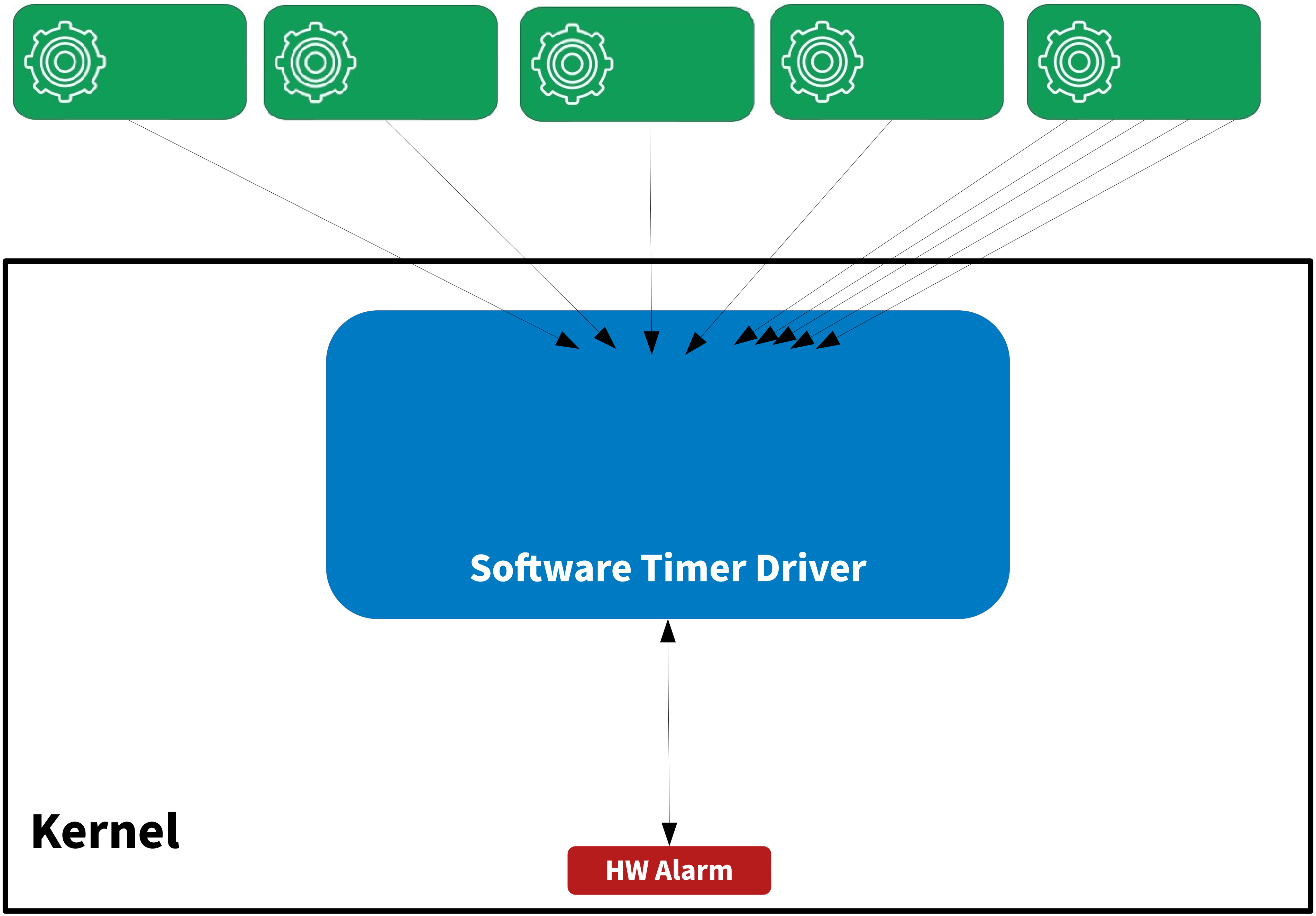
Capsules	**Processes**
• Isolated by compiler	• Isolated at run-time
• Shared stack, no heap	• Dedicated stack & heap
• Cooperative	• Preemptive
• Rust only	• Any language
• Method calls	• Context switch
• Replaceable at compile-time	• Replaceable at runtime

Different isolation mechanisms for different use cases

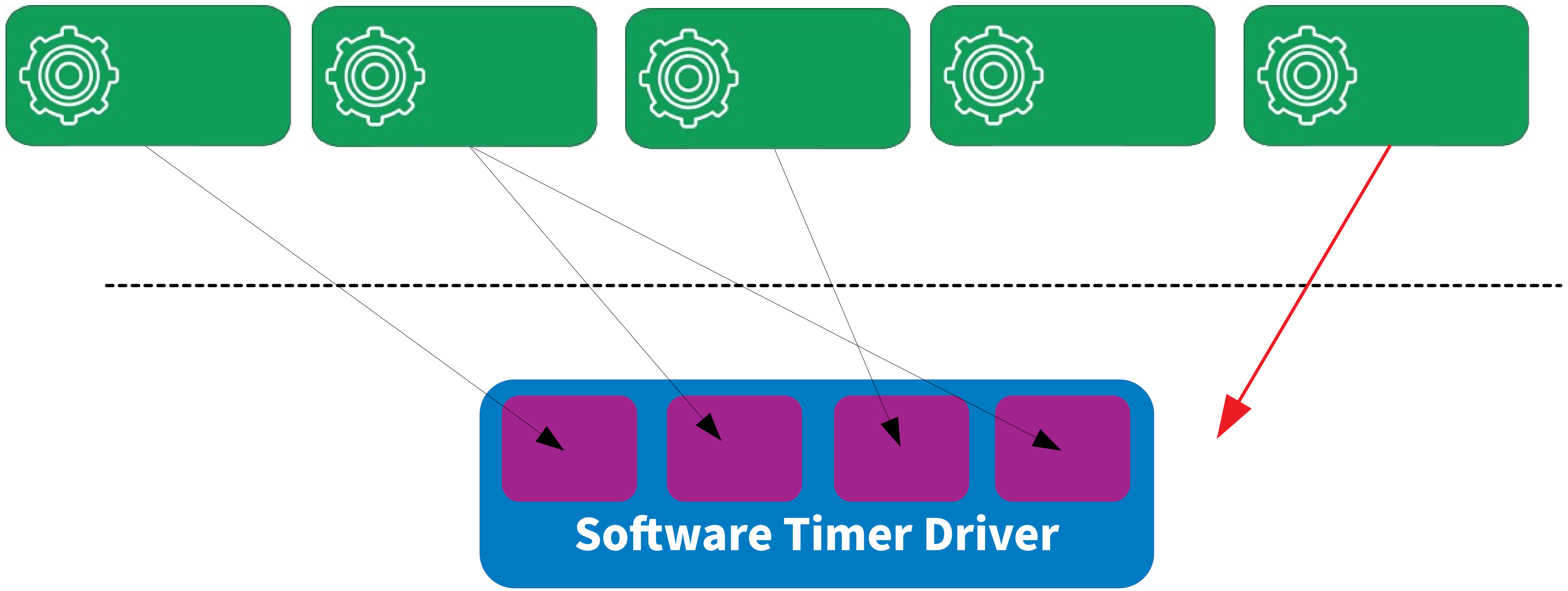

Processes
Application Developers
Timer SysCalls
802.15.4 Net.
Temp Sensor
Kernel
Virtual Alarm
RF233 Driver
I2C Driver
Timer Driver
SPI Driver
Kernel component developers
Microcontroller
Timer
SPI
I2C
Peripherals
Board Integrators

A static kernel needs resources to respond to unpredictable process requests

Working Example: Timer Driver

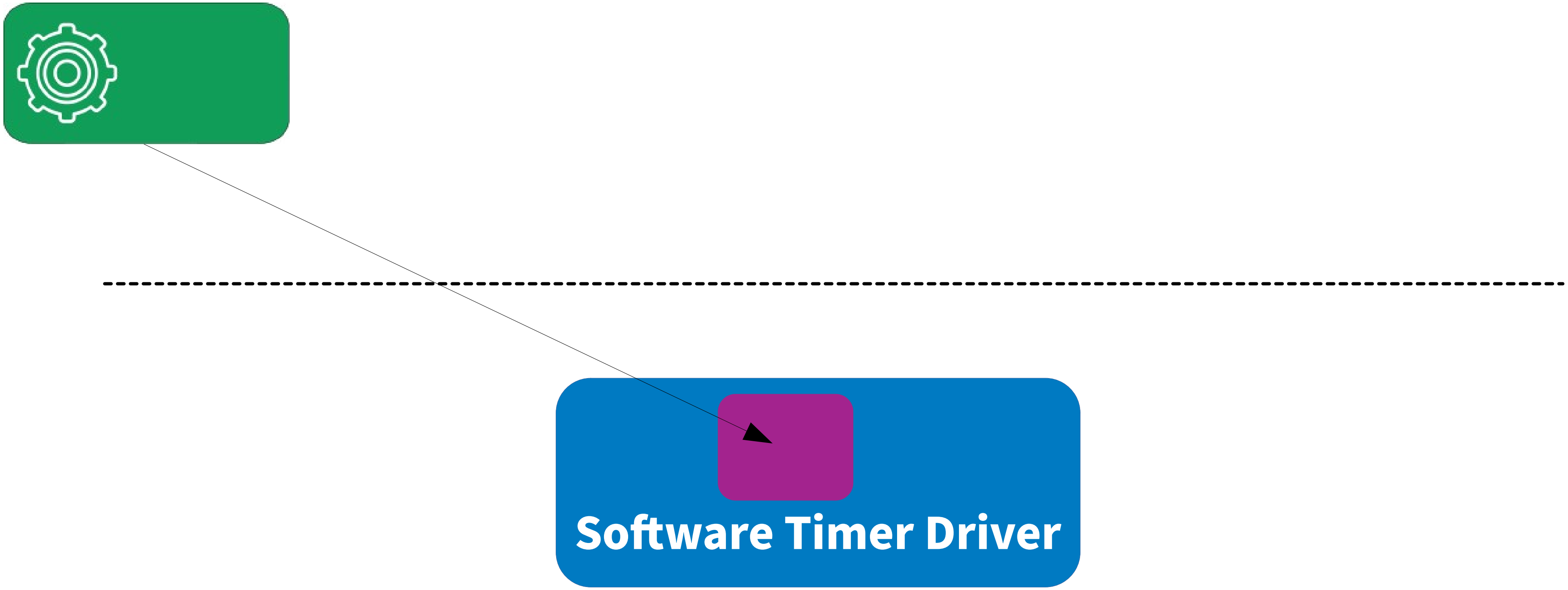


Statically allocating timer state?

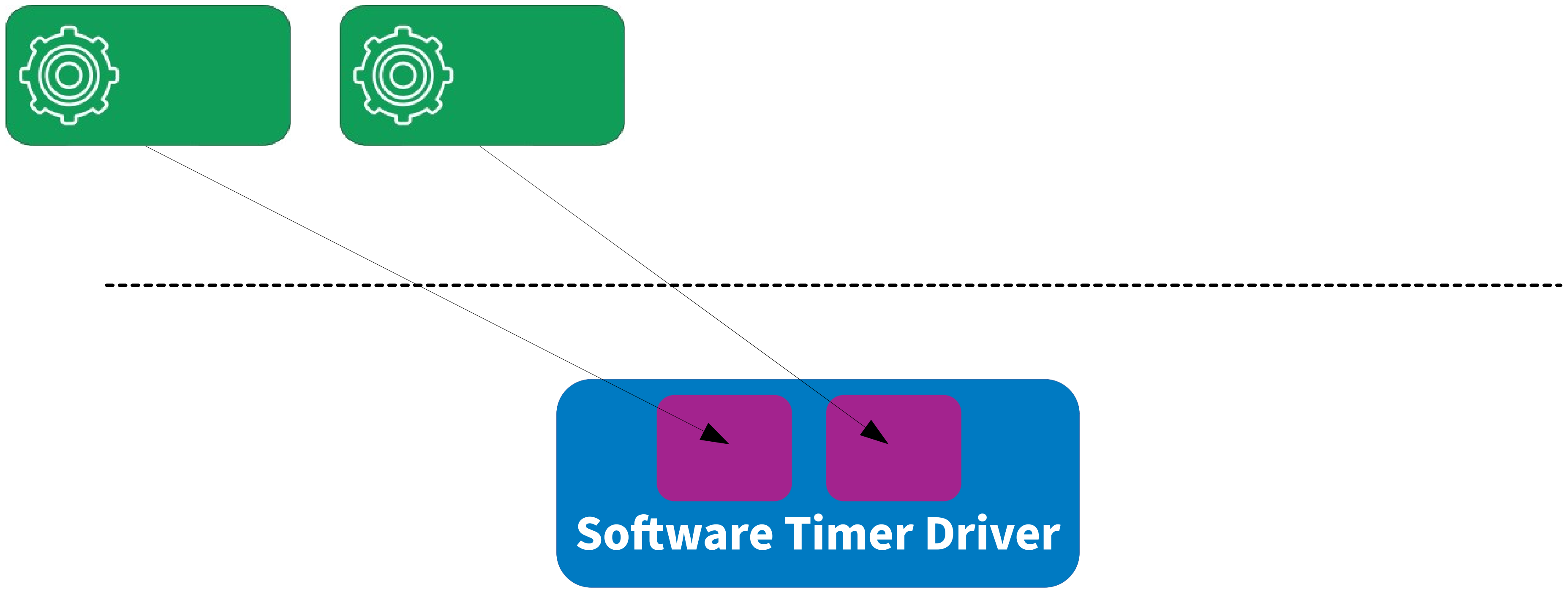


Static allocation must trade off memory efficiency and maximum concurrency

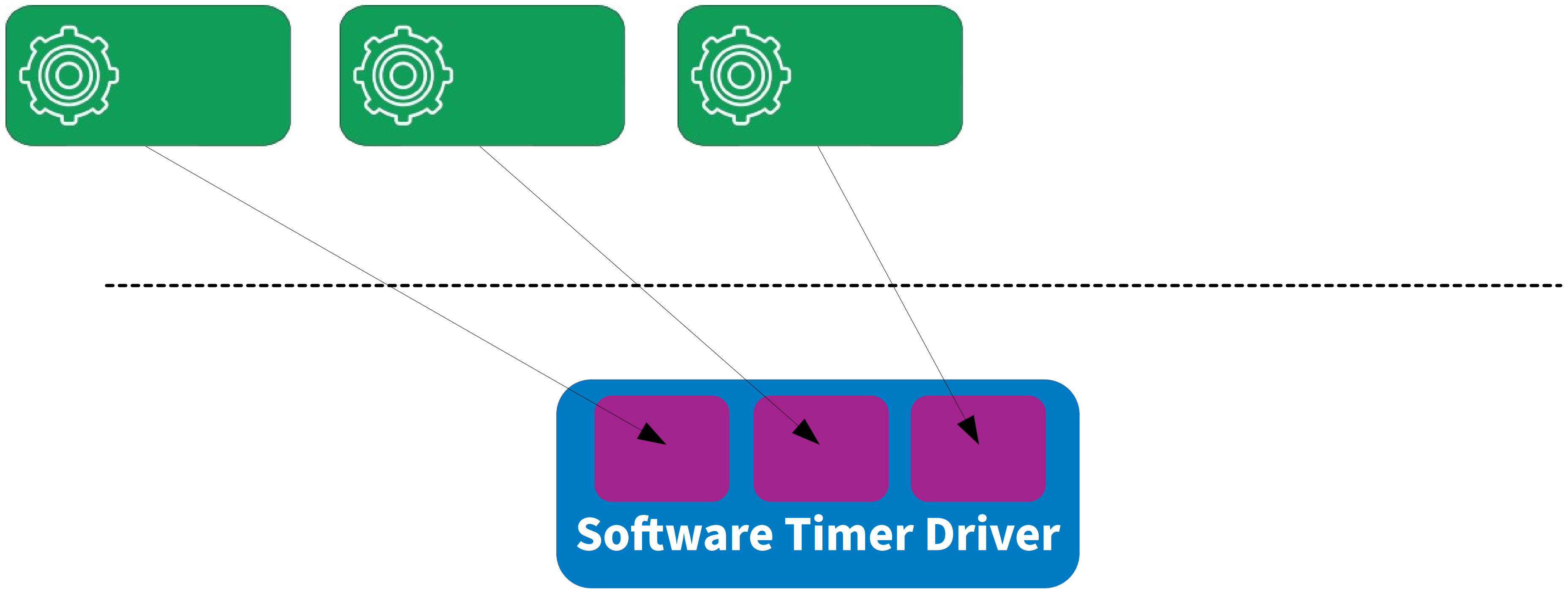
What About Dynamic Allocation?



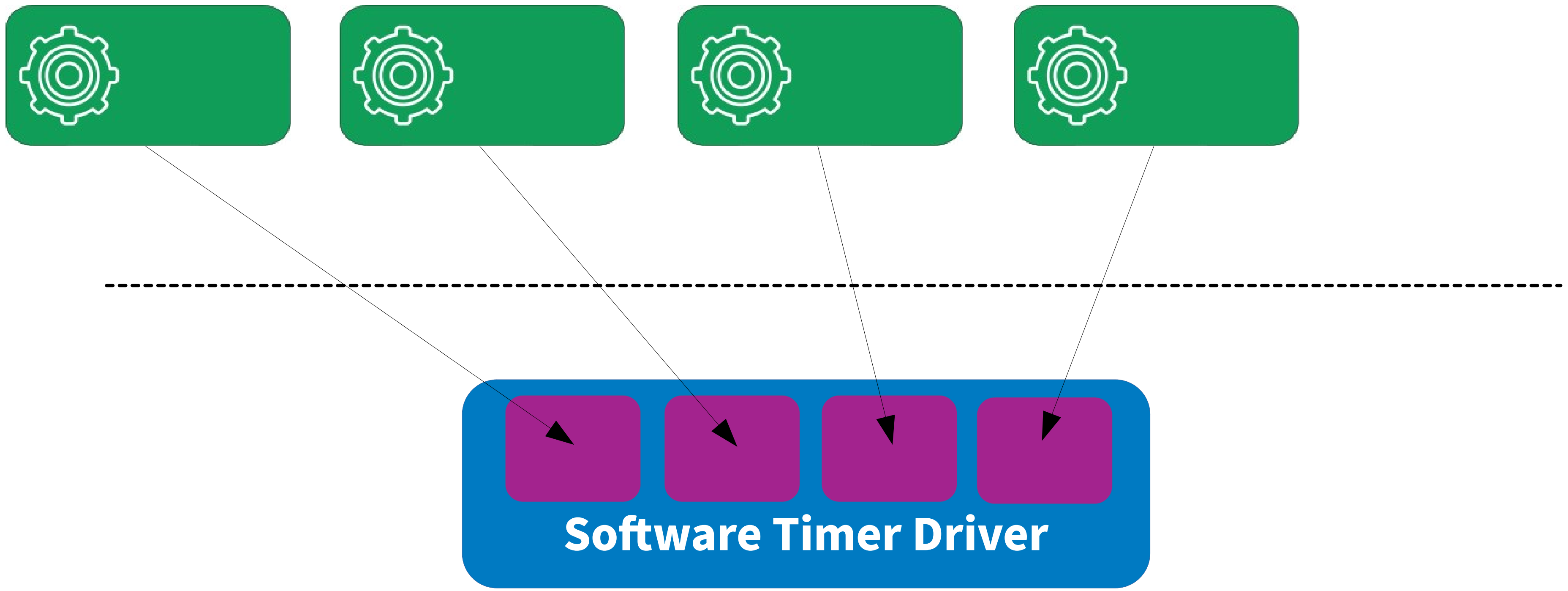
What About Dynamic Allocation?



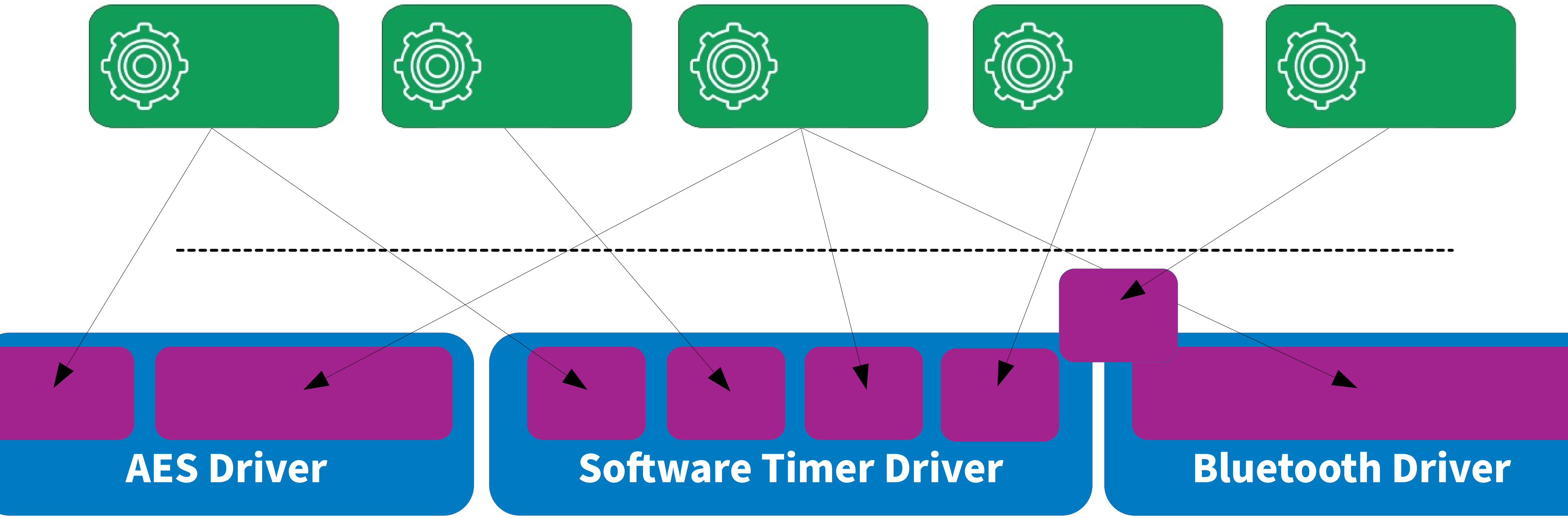
What About Dynamic Allocation?



What About Dynamic Allocation?



What About Dynamic Allocation?



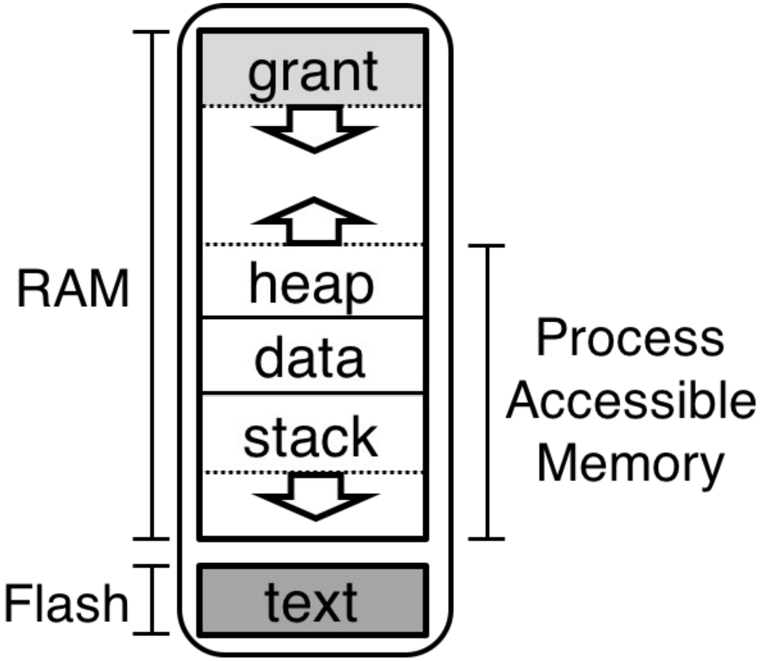
Can lead to unpredictable shortages.

One process's demands impacts capabilities of others.

Separate kernel heap for *each process*

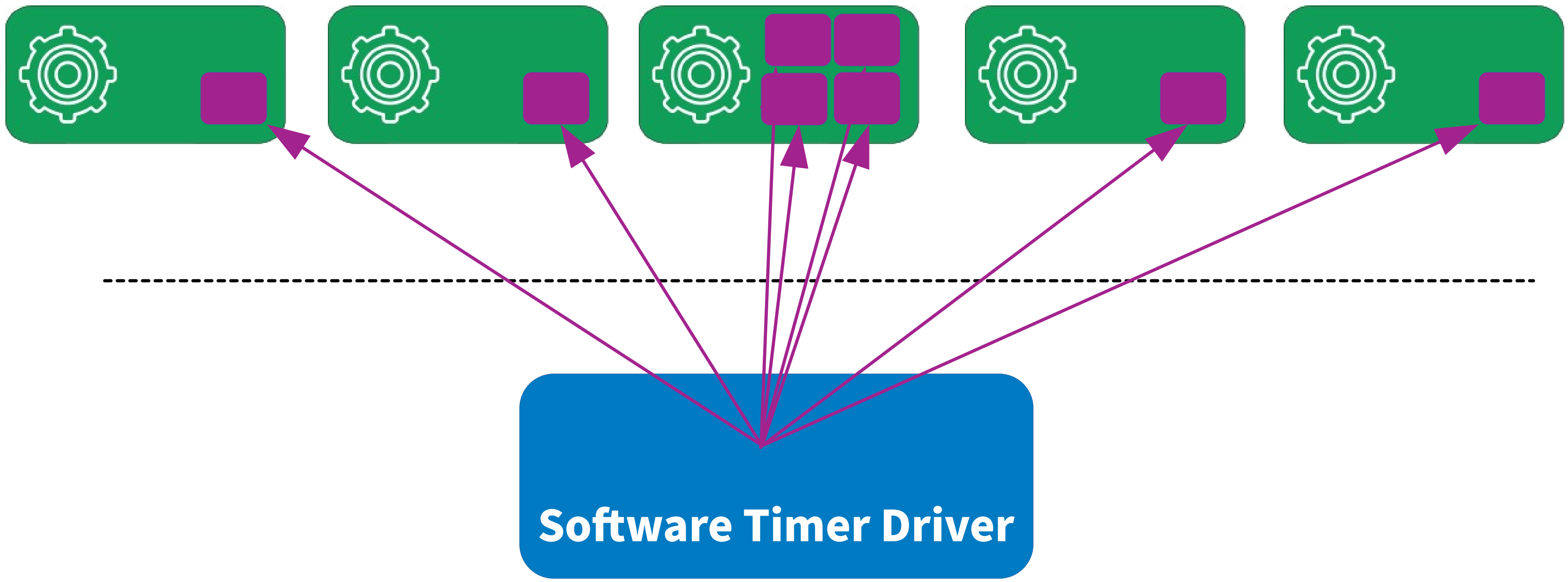
Grants

- Safely account for process-specific kernel heaps
- Allocations for one process do not affect others
- System proceeds if *one* grant section is exhausted
- All process resources freed on process termination



Grants:

Kernel heap *safely* borrowed from processes



Grants balance safety and reliability of static allocation with flexibility of dynamic allocation

Grants uses the type-system to ensure references only accessible when process is live

```
fn enter<'a, F>(&'a self, pid: ProcId, f: F) → where
    F: for<'b> FnOnce(&'b mut T)

      // Can’t operate on timer data here

      timer_grant.enter(process_id, |timer| {
          // Can operate on timer data here
          if timer.expiration > cur_time {
              timer.fired = true;
          }
      });

      // timer data can’t escape here
```

Resource Management in Tock

- Extremely limited memory limits isolation with traditional mechanisms
- Capsules decouple isolation from concurrency
- Still need dynamic allocations in static components
- Grants "borrow" memory from processes to service process requests
- Need to ensure grants for different processes can't reference each other