Lecture 18: Embedded OS

CS343 – Operating Systems

Branden Ghena – Spring 2022

Today’s Goals

- Introduce embedded OS concerns and how they are different from general-purpose computing.
- Provide insight into an alternative OS approach from Unix.
- Explore what OS research looks like.

Outline

- **Embedded Systems**
- Embedded Operating Systems
- Tock
  - Overview
  - Designing a secure kernel
  - Designing secure applications

What are embedded systems?

- An “embedded computer”, as in a computer within something else
  - Interacted with as a device, not as a computer
  - Smart lightbulbs, autonomous vehicles, wearable devices
  - Internet of Things, Robotics, Industry 4.0, Smart Home

- Common desire of interaction with the real world
  - Variety of possible concerns
    - Cost
    - Power
    - Real-time
    - Fault-tolerance

Resource-constrained embedded systems

- Class of embedded systems with onerous constraints
  - Limited memory, compute, and power
  - Often used in battery-operated or energy-harvesting scenarios

- This is the domain of ubiquitous computing
  - Cheap systems that can be deployed anywhere
  - Wireless sensor networks

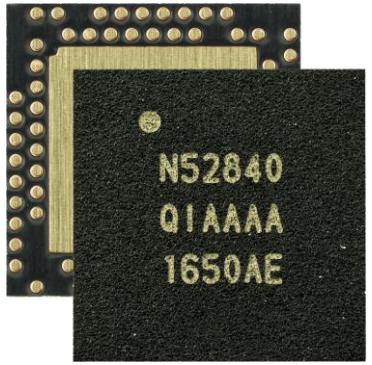
- We'll be exploring this area in this lecture
  - This is where my research has been focused
  - Takes problems that normal computing can mostly ignore
    - And amplifies them to an extreme point

Microcontrollers drive embedded systems

- Microcontroller is ~95% of a computer within a single chip
  - CPU
  - RAM
  - Flash storage
  - Hardware controllers for various signals and buses
    - Digital I/O, Analog I/O, PWM
    - UART, I2C, SPI, sometimes CAN or even Ethernet
  - Radio for wireless communication
    - Bluetooth Low Energy, 802.15.4 (Zigbee), WiFi

- What's missing?
  - Power, Sensors, Connectors, Antennas

7 mm x 7 mm



Microcontrollers are comparatively *very* constrained

- CPU
  - Single core, very simple pipeline
  - 32-bit or 16-bit (although 8-bit systems still exist)
  - 100 MHz or less (e.g. 32 kHz)
- RAM
  - 1-256 KB
  - Systems often disable some RAM to save power…
- Flash storage
  - 32-2048 KB
  - Mapped directly into memory
  - Code usually executes from flash!

Drivers are incredibly important

- Vast variety of sensors that might be used
  - Temperature, Humidity, Light Intensity, Light Color, Air Pressure, Air Quality, Acceleration, Rotation, Magnetic Field, Buttons
- Variety of other systems to communicate with
  - SD cards, radios, GPS, other microcontrollers
- Devices are the core purpose of embedded systems
  - Which makes correct drivers more important
  - But variety means you’re constantly writing new drivers

Energy is often the dominant concern

- Normal computer ~100 W (varies a lot)
  - CPU uses ~60 Watts
  - Powered by plug into mains

- Embedded systems often run on batteries
  - Four AA batteries: 15 Watt-hours
    - Average power to get 1 year of life: 1.7 mW
  - Coin cell battery: 0.36 Watt-hours
    - Average power to get 1 year of life: 41 μW

- Embedded system
  - ~50 mW when active
  - ~10 μW when in sleep mode
    - Goal of an embedded system: get back to sleep mode

Note: this is only one class of embedded

- Robotics: ROS (Robot OS) on top of normal Linux

- Industrial and smart home IoT
  - QNX (Unix-like RTOS)
  - Azure IoT (platform for secure, updatable devices)
    - Hardware/Software design with multiple microcontrollers
    - One just for security and managing resources
    - One just for managing over-the-air updates

Outline

- Embedded Systems
- **Embedded Operating Systems**
- Tock
  - Overview
  - Designing a secure kernel
  - Designing secure applications

Embedded needs its own operating systems

- Can’t use normal Linux!
  - Too little memory and processing
  - Too much concern about power
  - Microcontrollers don’t have the necessary hardware features (virtual memory)

- Important: Linux is *never* going to be the solution
  - Capabilities of general computers are orders of magnitude better
  - Embedded systems are gaining more capabilities
  - But new lower-power, lower-cost systems keep emerging too

Typical embedded OS design

- Assumption: embedded device has one single purpose

- One application and one kernel combined into a single program
  - Application might be multiple cooperative tasks
  - Kernel is mostly drivers, with maybe a scheduler
  - No protection, and usually minimal resource management

Contiki-NG

Two needs not met by traditional embedded OSes

1. Security
   - Protect the core platform from applications

2. Multiprogramming
   - Run multiple, unrelated applications (securely)

Challenge for secure embedded: no virtual memory

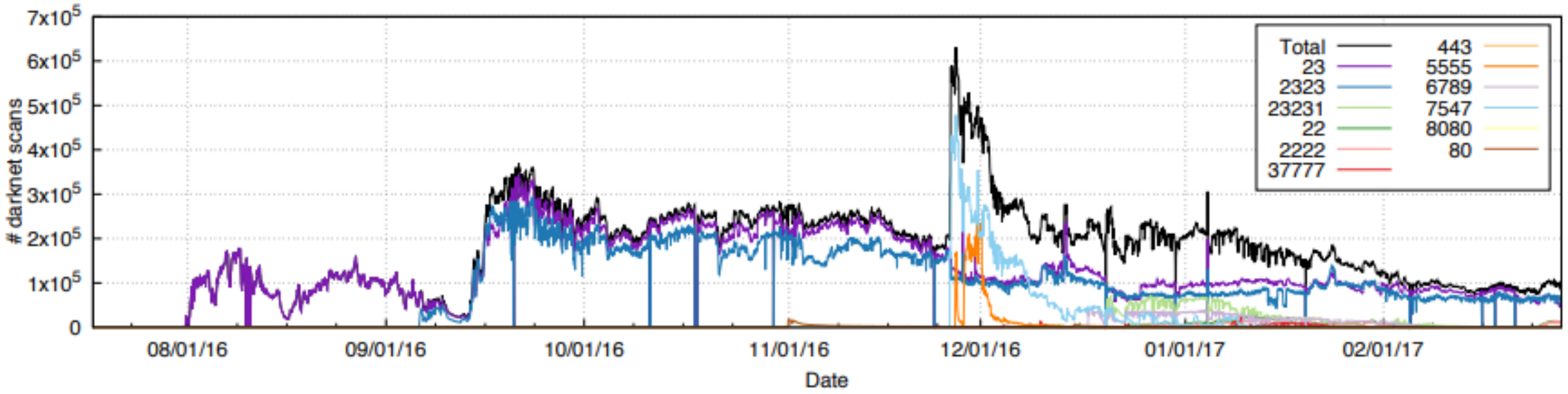
- No hardware support for virtual memory
  - So all addresses on the system are real physical addresses
- Nothing prevents applications from
  - Manipulating kernel data structures
  - Directly accessing hardware
- All applications on the system must be trusted
  - Devices do have one single purpose
  - But any weak link leaves the whole system broken

Embedded devices are a weak link

- Custom, application-specific code written in C
  - Limited code-reuse
  - Low testing coverage

- All code on the system is trusted
  - No isolation: any code can directly access hardware registers
  - Little distinction between “kernel” and “application”

Reminder: Mirai botnet (2016)



- Takes control of up to 600,000 insecure connected devices
  - IP-attached cameras, DVRs, routers, printers

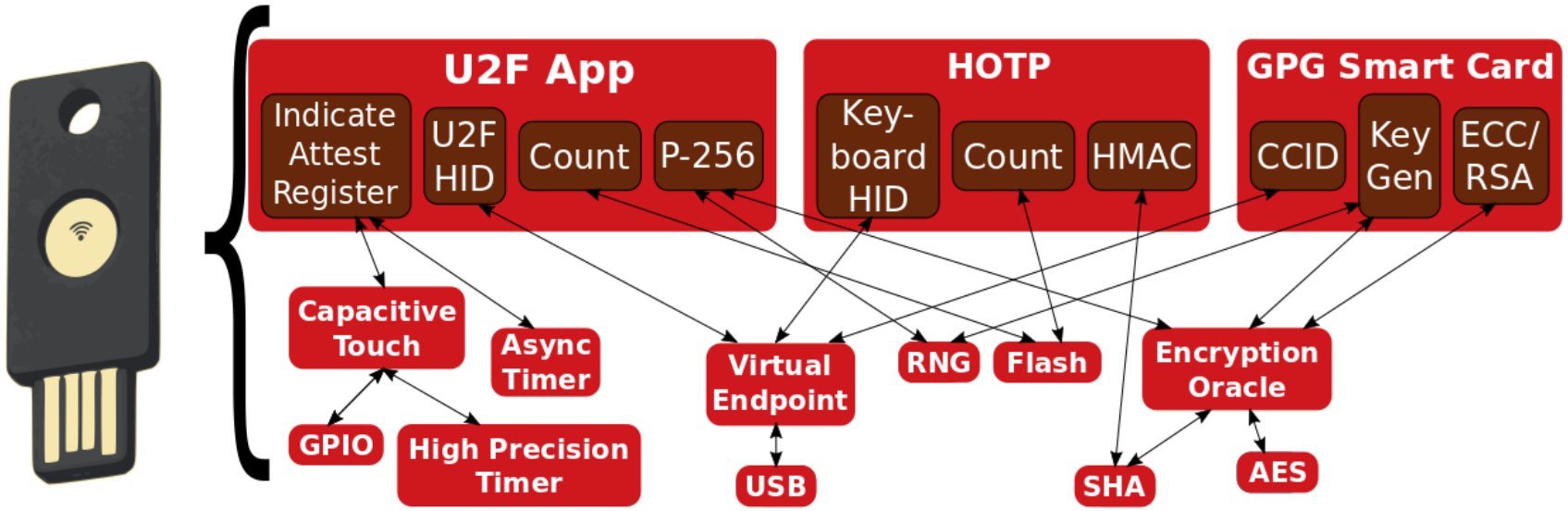
Weak devices provide network entry points

- IoT devices can be used as a network entry point
  - Step one: hack the device
  - Step two: use the device to access information on the private network

- Example: casino high-rollers database obtained through an IoT fish tank thermostat
  - https://interestingengineering.com/a-casinos-database-was-hacked-through-a-smart-fish-tank-thermometer

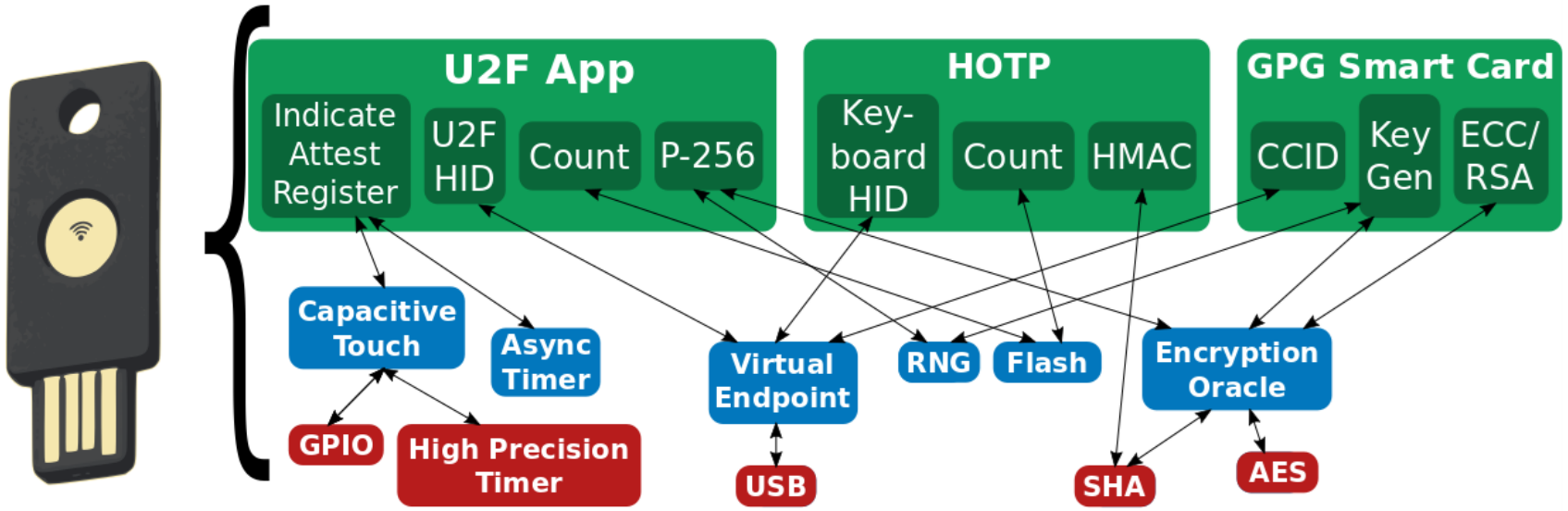
Modern systems increasingly need support for multiprogramming

- Example: USB authentication key
  - Universal Second Factor (U2F)
  - HMAC One-time Password (HOTP)
  - GPG Key (GNU Privacy Guard)



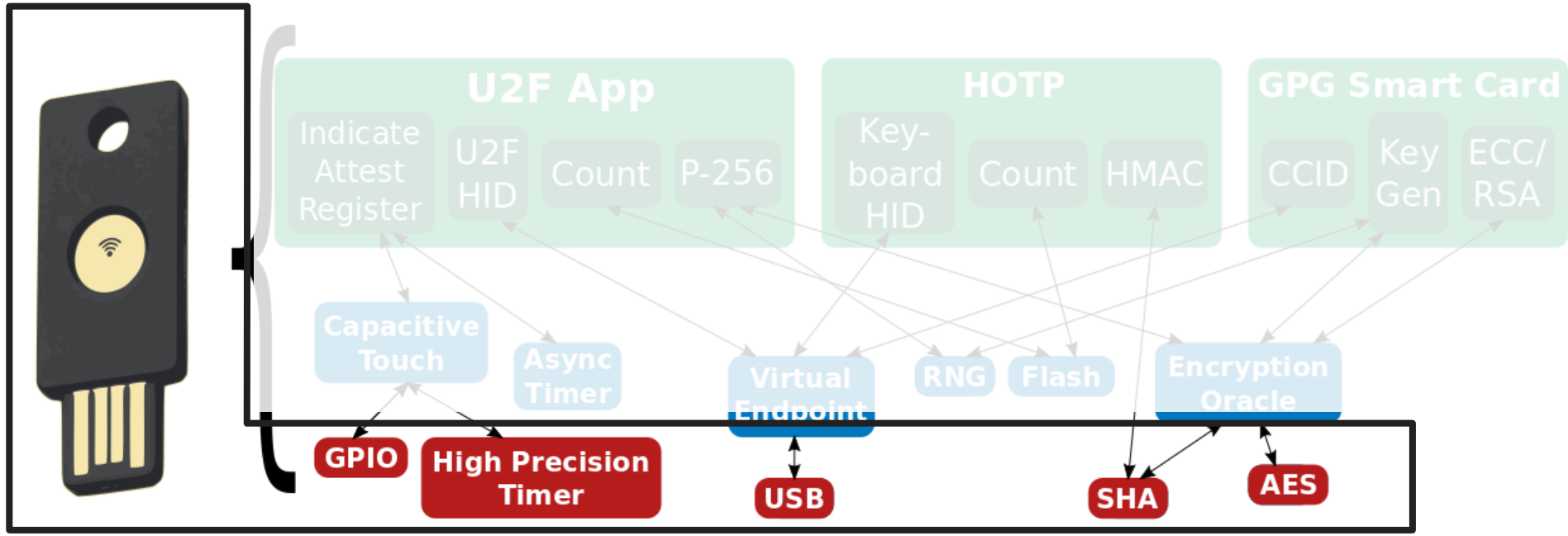
Security layering is desirable

- Different domains with different expectations
  - Applications, services, and platform



Platform layer

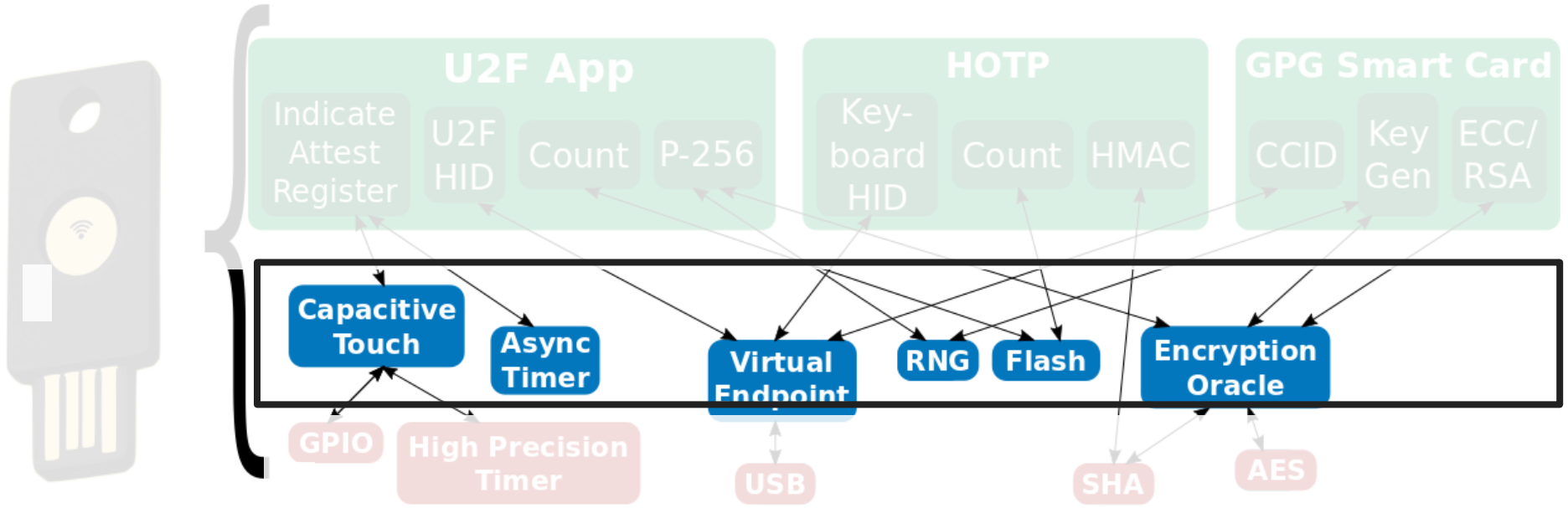
- Core kernel plus microcontroller-specific code
  - ~10 developers
  - Trusted compute base



Goal: possible to correctly extend TCB

Services layer

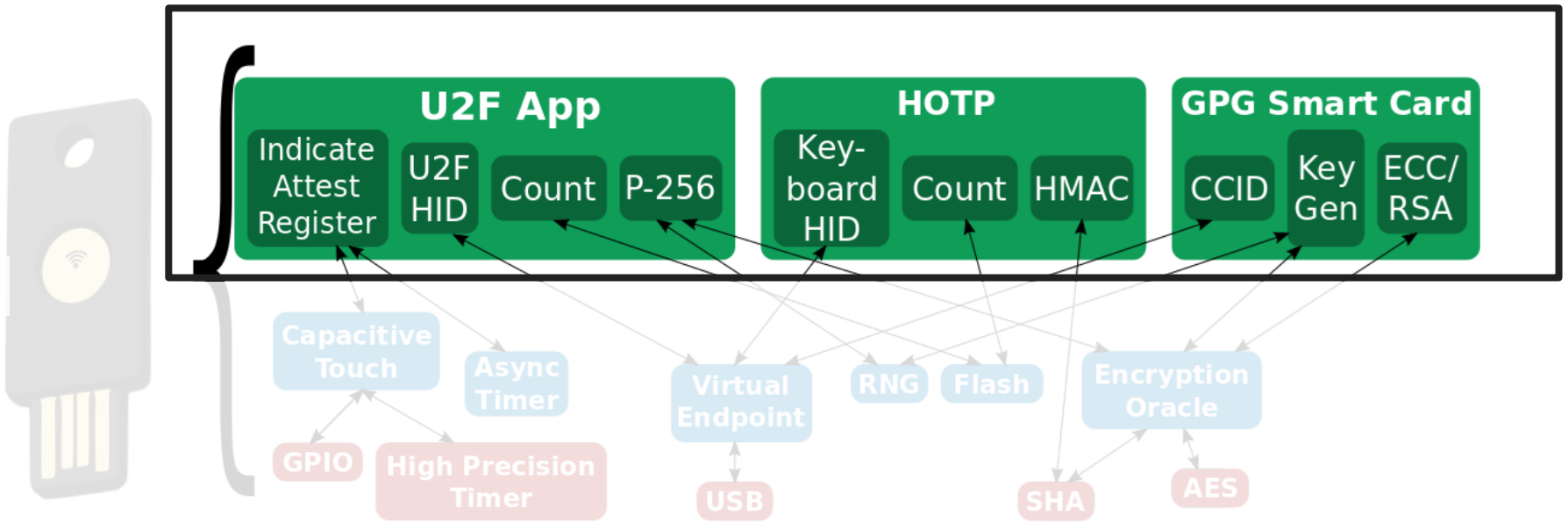
- Device drivers, networking, libraries
  - ~100 developers
  - Auditable, but possibly still buggy



Goal: protect kernel from safety violations

Applications layer

- End-user functionality
  - ~1000 developers
  - Third-party applications, potentially malicious



Goal: end-users can install 3rd-party apps

Break + Question

- What are the challenges of Virtual Memory on an embedded device?
  - How many pages of memory are there? (64 kB RAM, 1 MB Flash)
    - How many bytes would the page table require?
  - What are the timing challenges of Virtual Memory?

Break + Question

- What are the challenges of Virtual Memory on an embedded device?
  - How many pages of memory are there? (64 kB RAM, 1 MB Flash)
    - How many bytes would the page table require?
    - 64/4 + 1024/4 = 272 pages @ 4 bytes per entry = 1088 bytes
  - What are the timing challenges of Virtual Memory?
    - Two memory accesses per access, unless there's a TLB
    - Then, 1 or 2 memory access depending on TLB contents
      - Unpredictable access times!

Outline

- Embedded Systems
- Embedded Operating Systems

- **Tock**
  - **Overview**
  - Designing a secure kernel
  - Designing secure applications

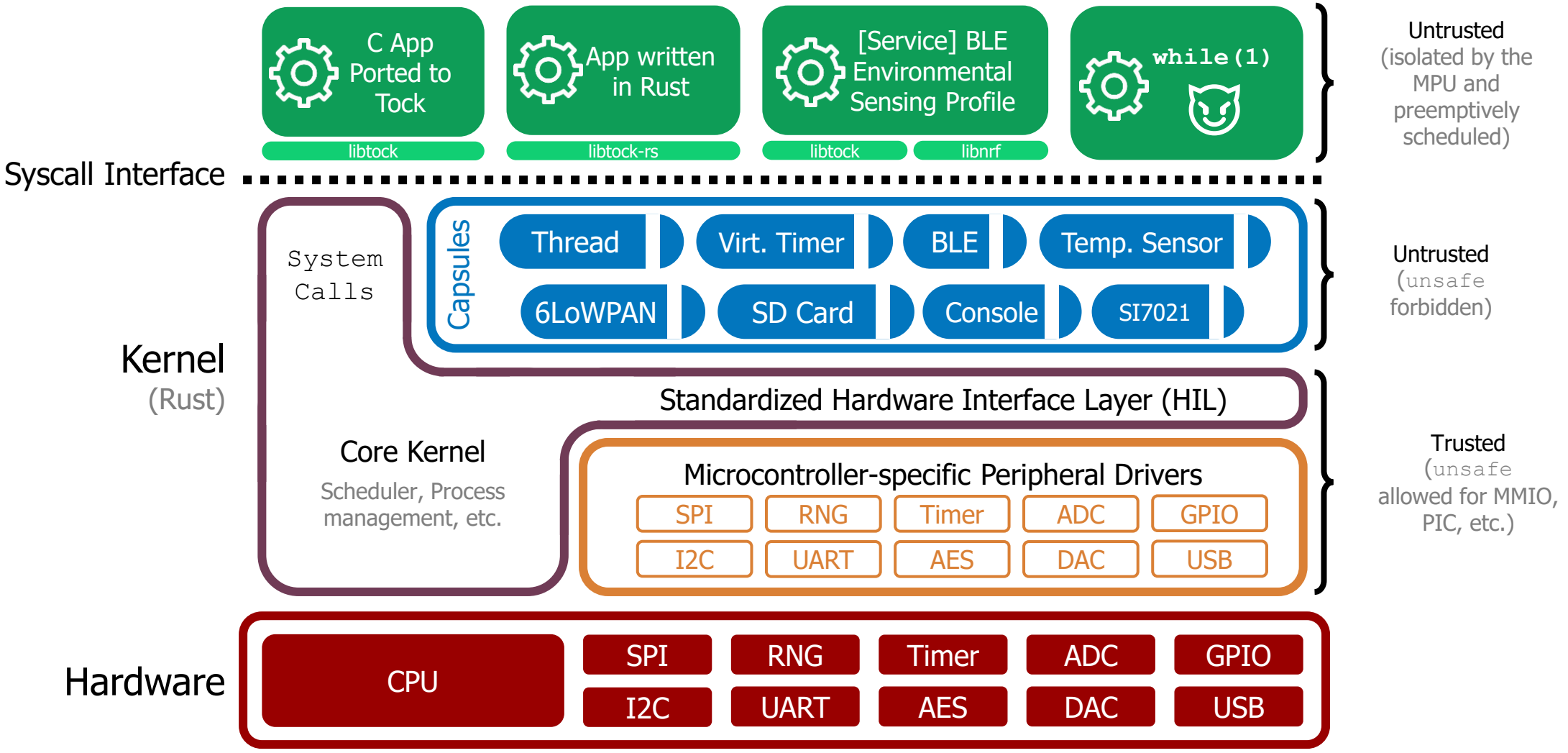
Tock

- Embedded OS designed for secure multiprogramming
  - Kernel written in Rust programming language
  - Applications written in any language
  - Runs on multiple hardware platforms

- Open-source research project
  - 2015 collaboration between Stanford, UC Berkeley, and Michigan
    - Since expanded to 6-7 universities
    - Plus several companies (Google, Western Digital)

- https://github.com/tock/tock

Tock software organization



Tock's isolation mechanisms

Totally untrusted

Trusted for liveness, not safety

Processes

- Standalone executable in any language
  - C, C++, Rust, Lua
- Isolation enforced at runtime
- Higher overhead
- Applications

Capsules

- Rust code linked into kernel
- Isolation enforced at compile-time
- Lower overhead
- Used for device drivers, protocols, timers...

How do applications access devices?

- System calls are used to access devices

- Three generic syscalls
  - Command – takes a 32-bit numerical argument
  - Allow – takes a pointer to a buffer to read/write
  - Subscribe – takes a pointer to a function to callback

- First two arguments to all syscalls
  1. Driver number (the driver it wants to interact with)
  2. Minor number (driver-specific identifier)

Example: console capsule (driver number: 1)

- Command syscall values
  1. Transmit from buffer, argument is length
  2. Receive into buffer, argument is length
  3. Cancel request

- Allow syscall values
  1. Pointer to buffer for sending
  2. Pointer to buffer for receiving

- Subscribe syscall values
  1. Send complete handler
  2. Receive complete handler

Writing bytes to console:

1. uint8_t buffer[20] = {…data…}
2. Allow(1, 1, buffer)
3. Command(1, 1, 20)

Event-driven programming

- Tock adopts an event-driven model to avoid concurrency issues
  - Single core system, so the source of concurrency is interrupts
  - Exposed to applications through “subscribe” callbacks

- Callbacks never occur during normal operation
  - Even if application is descheduled due to timeslice

- Additional syscall: yield – no arguments
  - Application pauses until a callback is ready for it
  - Once one or more events are ready
    - Call the callback handlers, one at a time
    - Return from yield statement

Yield example

```
void sensor_callback (int value) {
  printf("Got sensor reading %i\n", value);
}


int main () {
  sensor_register(sensor_callback); // calls allow
  sensor_sample(); // calls command
  yield();
}
```

Synchronous interactions with event model

```
bool sensor_flag = false;

void sensor_callback (int value) {
  printf("Got sensor reading %i\n", value);
  sensor_flag = true;
}

int main () {
  sensor_flag = false;
  sensor_register(sensor_callback); // calls allow
  sensor_sample(); // calls command
  while (!flag) {
    yield();
  }
}
```

Outline

- Embedded Systems
- Embedded Operating Systems

- **Tock**
  - Overview
  - **Designing a secure kernel**
  - Designing secure applications

Tock threat models

- Threat model – the universe of concerns for a system design
  - Systems can’t defend against every possible attack
  - So what attacks is the OS actually concerned about?

- Tock splits threat model into application and kernel parts

Kernel threat model

- Confidentiality
  - Secrets may not be accessed by applications or capsules.
- Integrity
  - Applications and capsules may not modify kernel data except through exposed APIs.
- Availability
  - Applications cannot starve the kernel of resources or deny service.
  - Capsules may deny service but should be designed to prevent starvation.

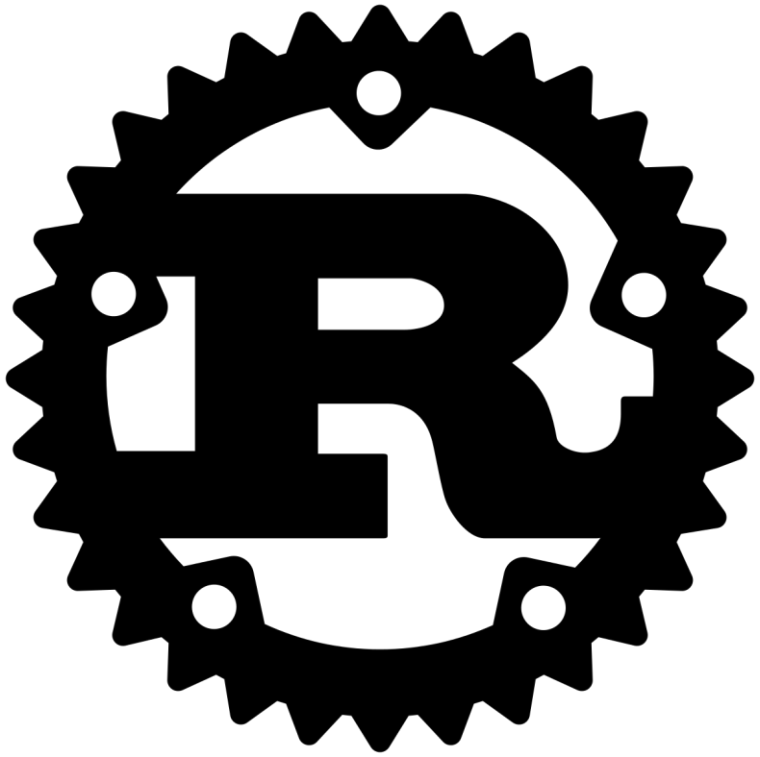
Guarantees of safety from parts of the kernel

- How do we guarantee these without virtual memory?
  - Secrets **may not be accessed** by applications or capsules.
  - Applications and capsules **may not modify kernel data except through exposed APIs.**

- In normal operating systems, drivers are run in kernel mode
  - Full access to memory and hardware on the system

- Otherwise, they would be in userspace with limited access
  - Likely resulting in slower operation

Tock solution: use language features



- Tock uses the Rust programming language for the kernel

- Systems language
  - Represents how hardware actually interacts
  - Strong type system and memory safety
  - Runtime behavior similar to C

- Result: capsules *cannot* access memory they do not own
  - Cannot access application secrets
  - Cannot modify other kernel structures (even by accident)

What problems is Rust trying to solve?

- Memory lifetime
  - Use-after-free
  - Common example: pass a reference into a function, then free it

- Data races
  - Multiple access to data and at least one modifies it
  - Problems we were solving with locks

Rust has a strong notion of “ownership”

- Ownership is a notion of managing memory and sharing.

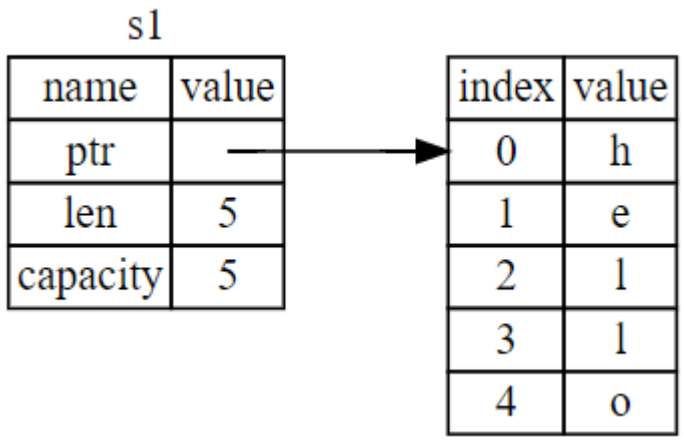
- Rust ownership rules
  - Each value has a variable that’s called its owner.
  - There can only be one owner at a time.
  - When the owner goes out of scope, the value is dropped.

- Last one is straightforward:
  - Values on the stack go away at the end of the function (or any block { })
  - This *lifetime* works just like C or C++

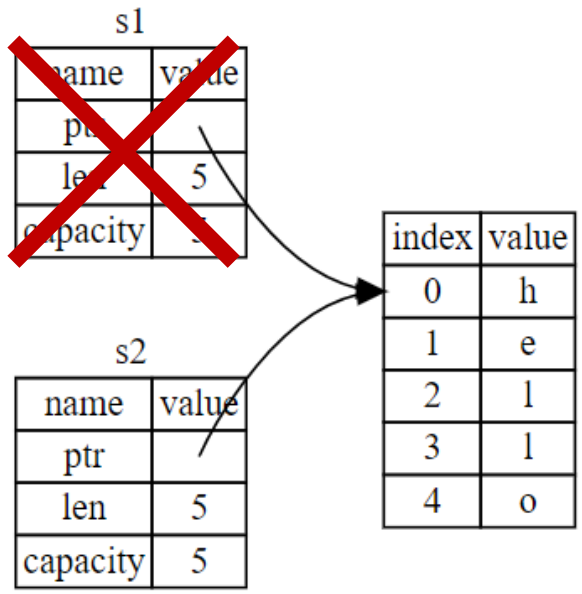
Example: ownership with strings

- Rust ownership rules
  - Each value has a variable that's called its owner.
  - There can only be one owner at a time.

```
let s1 =
String::from("hello");
```




```
let s1 =
String::from("hello");
let s2 = s1;
```



Ownership is transferred through function calls

```
fn main() {
  let s = String::from("hello"); // s comes into scope
  takes_ownership(s); // s's value moves into the function...

  // s is no longer valid here

} // Here, x goes out of scope, then s. But because s's value
  // was moved, nothing special happens.


fn takes_ownership(some_string: String) {
  // some_string comes into scope
  println!("{}", some_string);

} // Here, some_string goes out of scope and `drop` is called.
  // The backing memory is freed.
```

Ownership model prevents data races

- References allow multiple read-only access to a value
  - For example: passing a value into a function by reference
  - Any number of references may exist to a value

- Requesting a writable (mutable) reference requires sole ownership
  - No other references may exist
  - Original owner cannot access value

- This is enforced by the compiler!!!
  - So there is no runtime cost
  - And programmers cannot do bad things by accident

Tock downside: all drivers *must* be written in Rust

- Why hasn't *every* OS taken the "language support" path?
  - Primarily, it wasn't really available. C or C++ were the only real options.

- Also, the vast majority of developers know C but not Rust
  - And the vast majority of existing code is in C and not Rust

- One of Tock's major challenges is that things like networking stacks need to be re-written in Rust
  - BLE, 802.15.4, WiFi

My feelings about writing code in Rust

- Three steps of Rust acceptance

1. You are confused about little weird syntax things.

2. You are frustrated by so many compilation errors.

3. You realize your code mostly works if it compiles.

Sidebar: the future of Rust

- Rust is still growing in popularity but still young
  - Many people are excited to have a safe alternative to C
  - But there is a steep learning curve to using Rust
  - And a vast body of existing code already in C

- How dominant would it have to be to switch our curriculum?
  - Unclear. We've taught C for decades.

  - Certainly considering it for CS211, but no immediate plan to change
    - Still can't *stop* teaching C (too useful)
    - Rust in 5 weeks at the end of CS211 would be much harder than C++

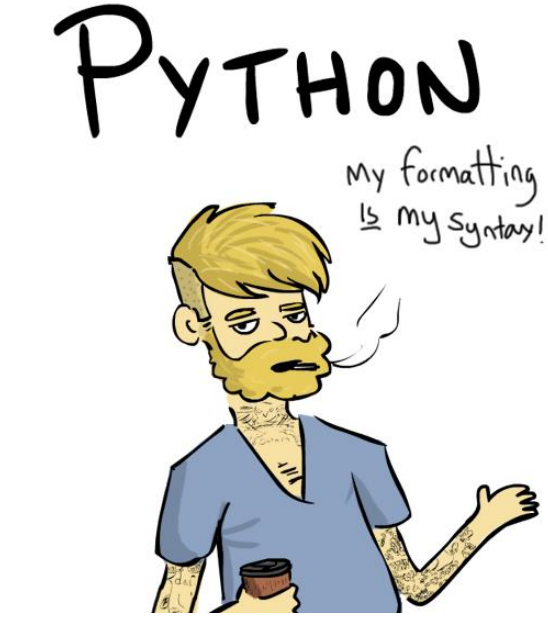
Learning Rust

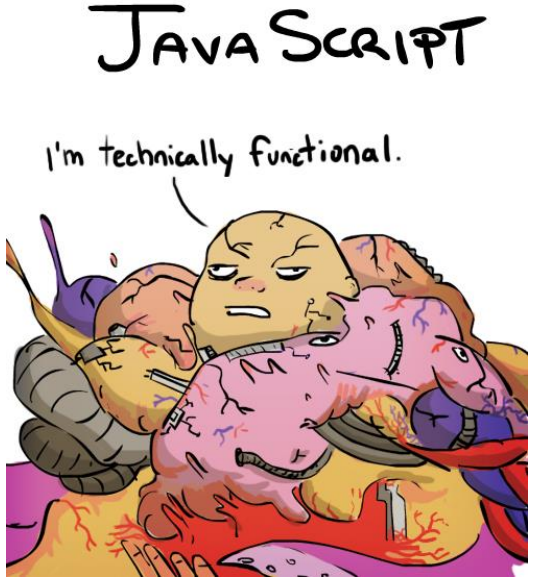
- No class at Northwestern uses it right now 😢

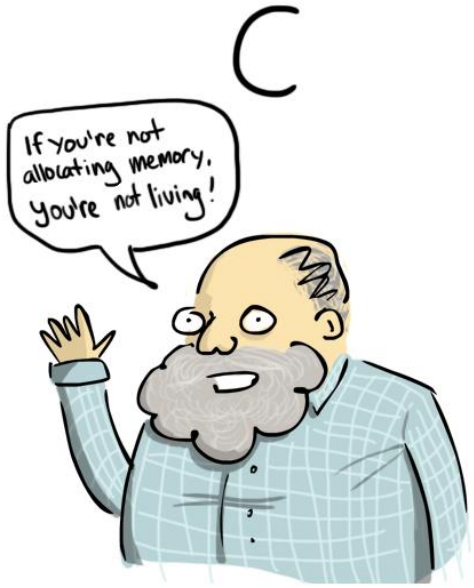
- So you'll definitely need to learn it on your own
  - https://www.rust-lang.org/learn

  - https://serokell.io/blog/learn-rust

Break + Programming languages personified







https://leftoversalad.com/c/015_programmingpeople/

Outline

- Embedded Systems
- Embedded Operating Systems

- **Tock**
  - Overview
  - Designing a secure kernel
  - **Designing secure applications**

Application threat model

- Confidentiality
  - Secrets may not be accessed by other applications or capsules.
- Integrity
  - Data may not be modified by other applications or capsules except when the applications allows access.
- Availability
  - Applications may not deny service to one another, except that finite resources may be granted in a first-come, first-served order.

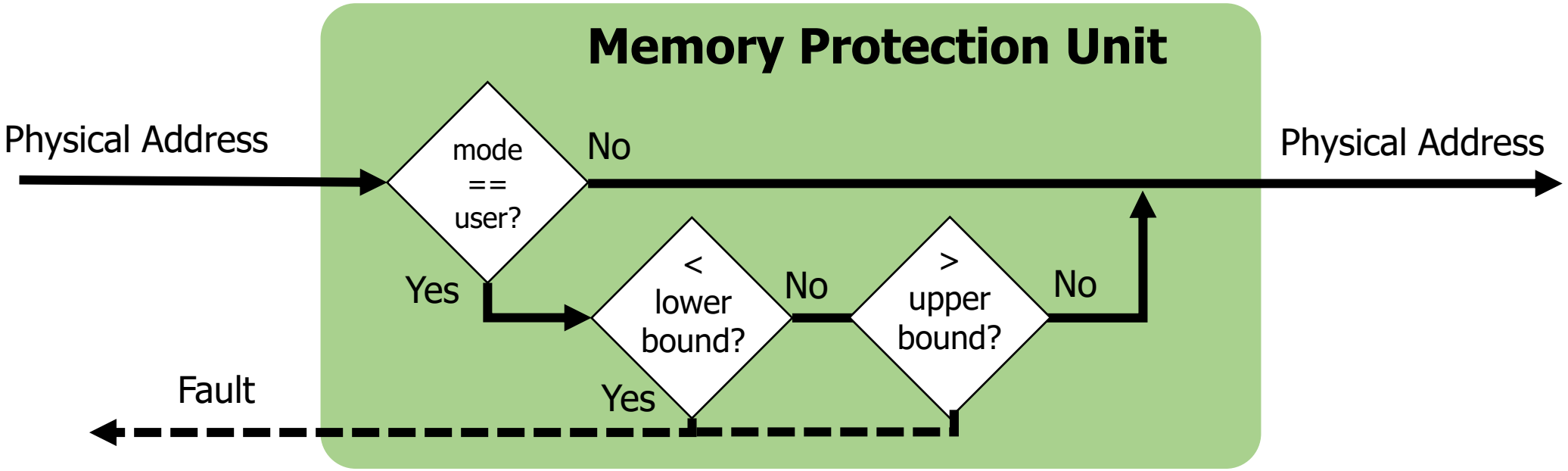
Problems to deal with in applications

1. How do we prevent malicious accesses without virtual memory?
   - Secrets **may not be accessed** by other applications
   - Data **may not be modified by other applications**

Memory access protection with hardware

- Modern microcontrollers have a memory safety feature
  - Memory Protection Unit (MPU)

- Base-and-bounds model of security
  - Bounds checking, but no translation



Problems to deal with in applications

1. How do we prevent malicious accesses without virtual memory?
   - With Memory Protection Unit hardware

2. How do we load multiple applications without virtual memory?
   - We do not want to compile programs for specific physical addresses

Application addresses managed by Position-Independent Code

- Position Independent Code (PIC)
  - Compile program using only *relative* assembly instructions
  - Plus a register (base pointer) that is set to point to data section
  - Feature available in modern GCC for ARM

- All accesses in program become relative
  - Stack accesses relative to stack pointer
  - Data accesses relative to base pointer
  - Heap accesses relative to base pointer (plus size of data)
  - Code accesses relative to current instruction pointer

- OS kernel sets up stack, base, and instruction pointers for where program was really placed in memory

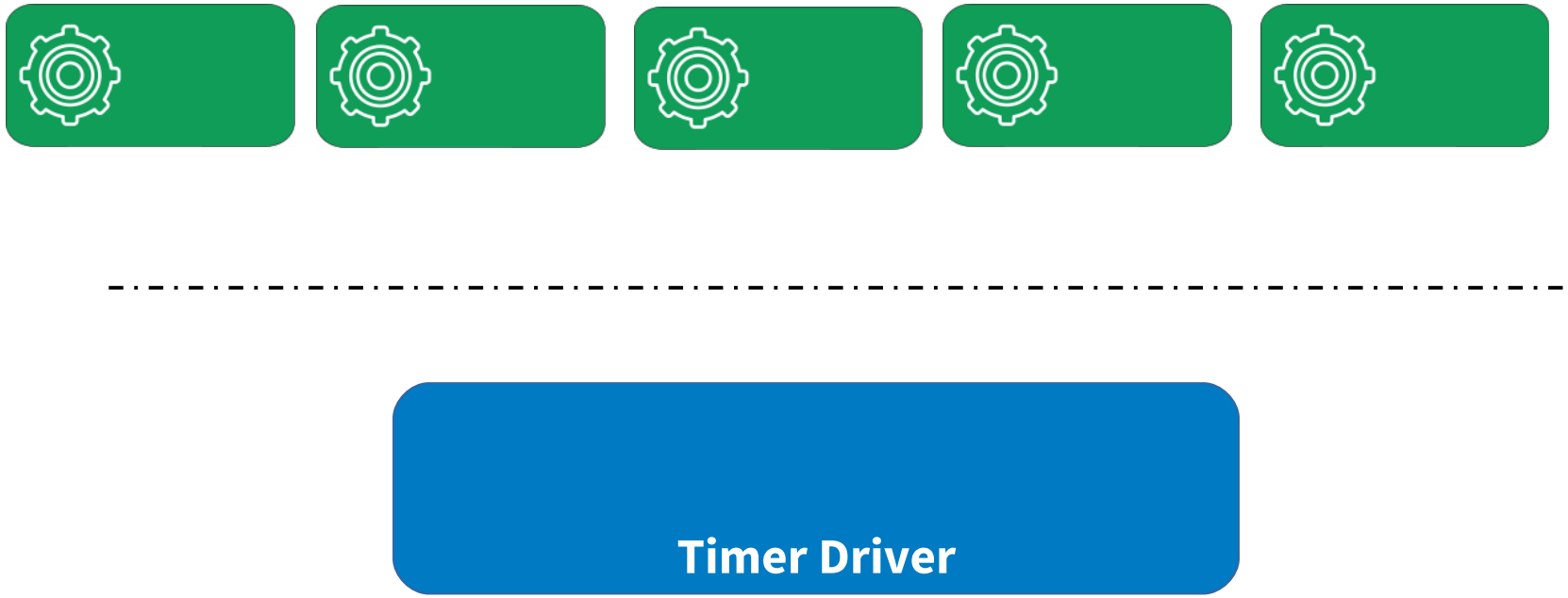
Problems to deal with in applications

1. How do we prevent malicious accesses without virtual memory?
   - With Memory Protection Unit hardware

2. How do we load multiple applications without virtual memory?
   - With Position Independent Code

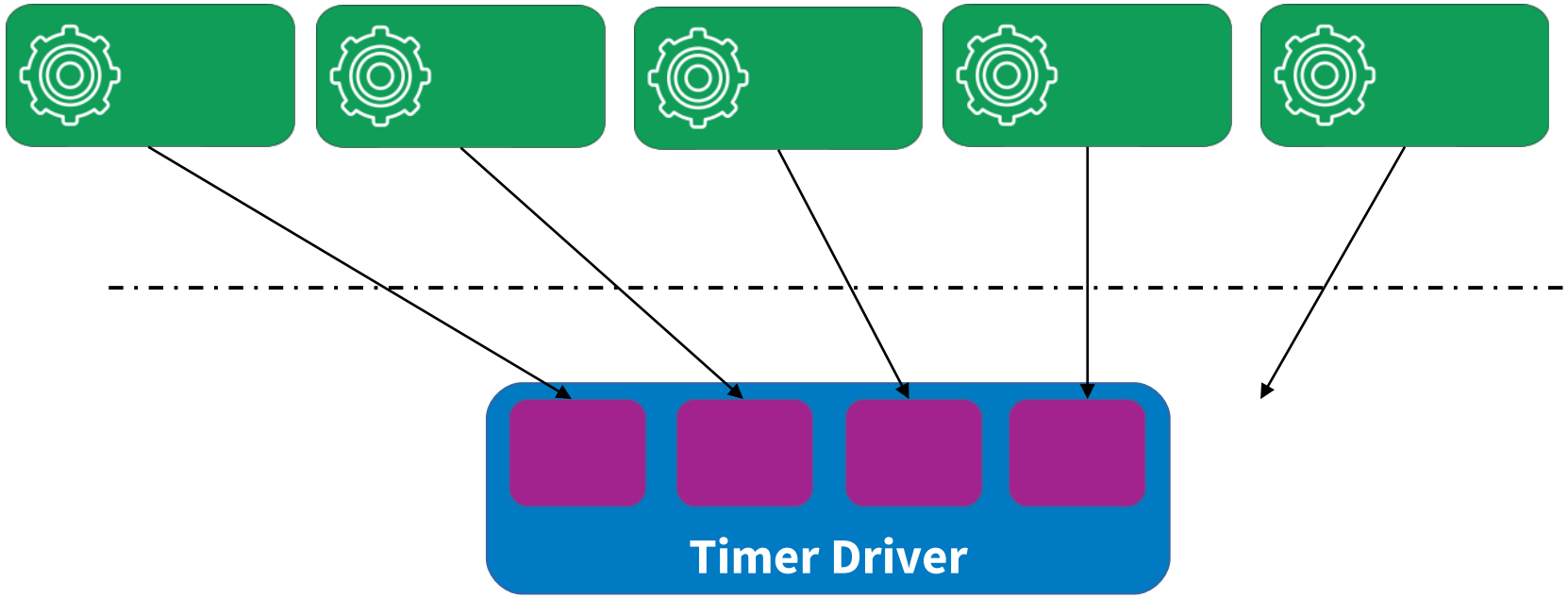
3. How do we manage having so little memory?
   - Applications **may not deny service** to one another

Example of denial of service with timers



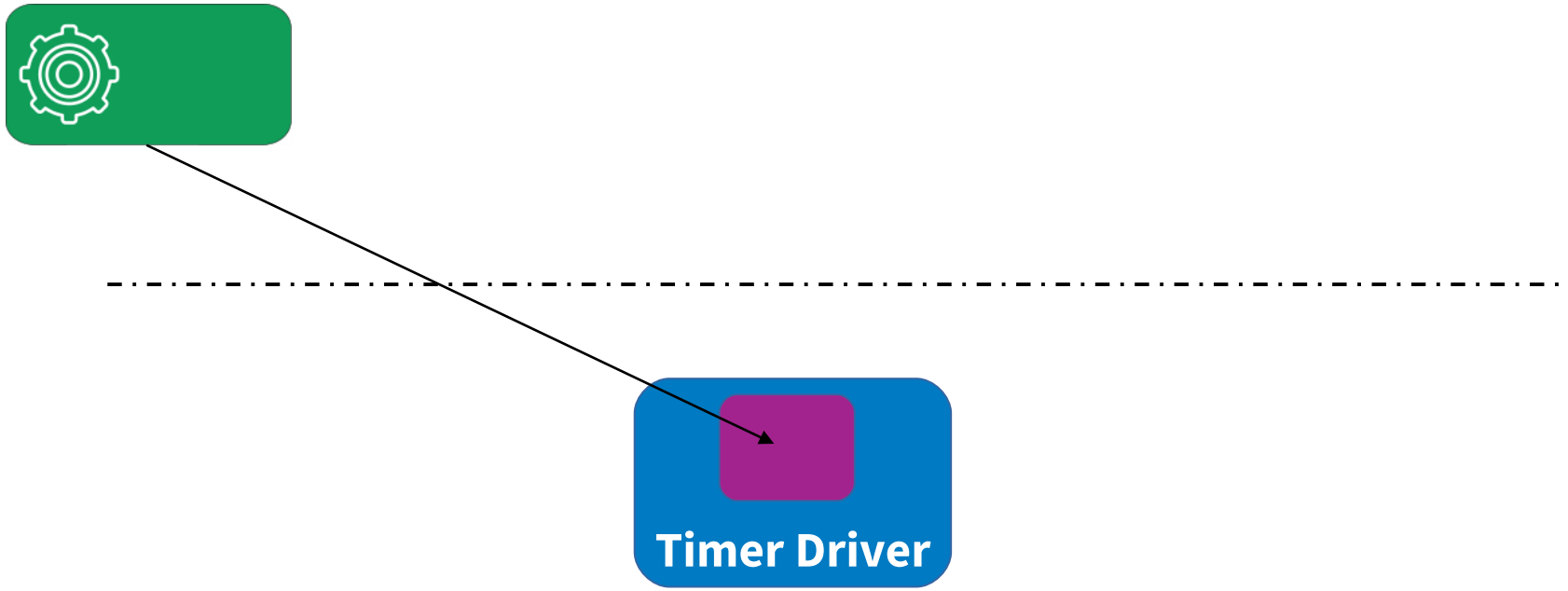
- Multiple applications like want their own individual timers

Example of denial of service with timers



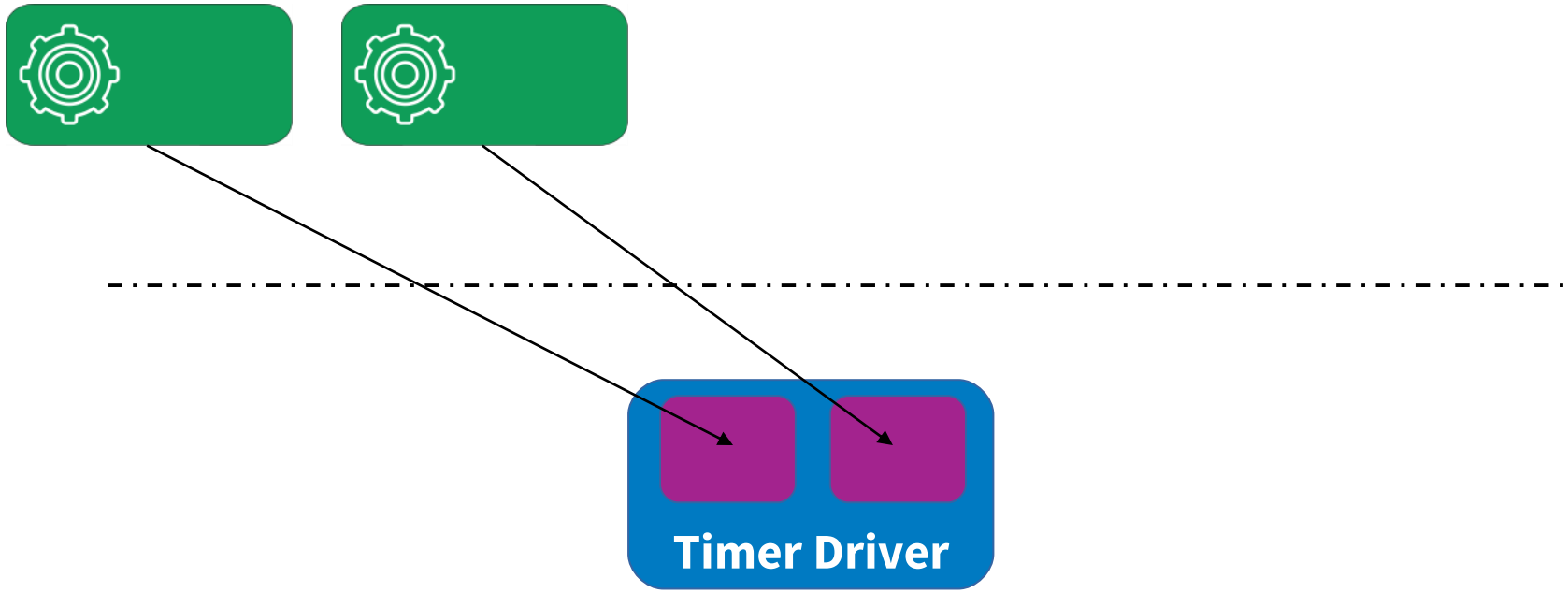
- Static allocation of timers results in needs of some applications denying service to others

Example of denial of service with timers



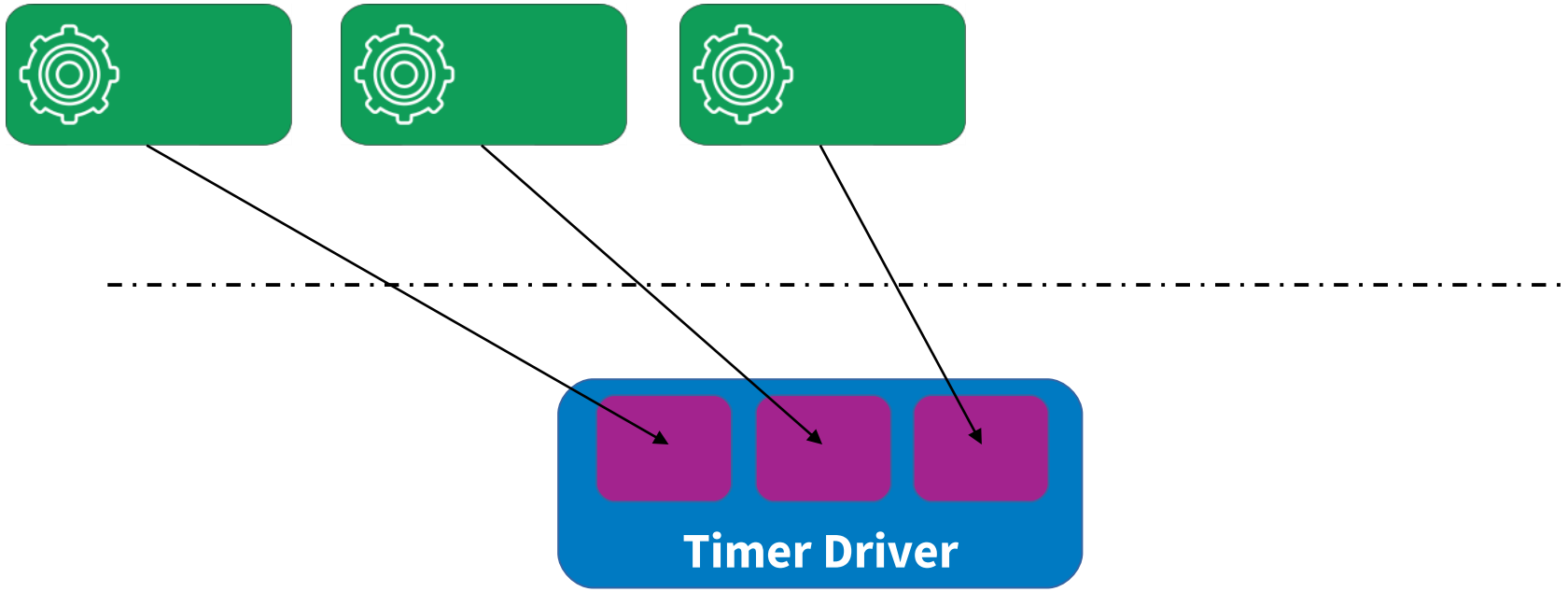
- Dynamic allocation leads to needs of unrelated drivers denying service to applications

Example of denial of service with timers



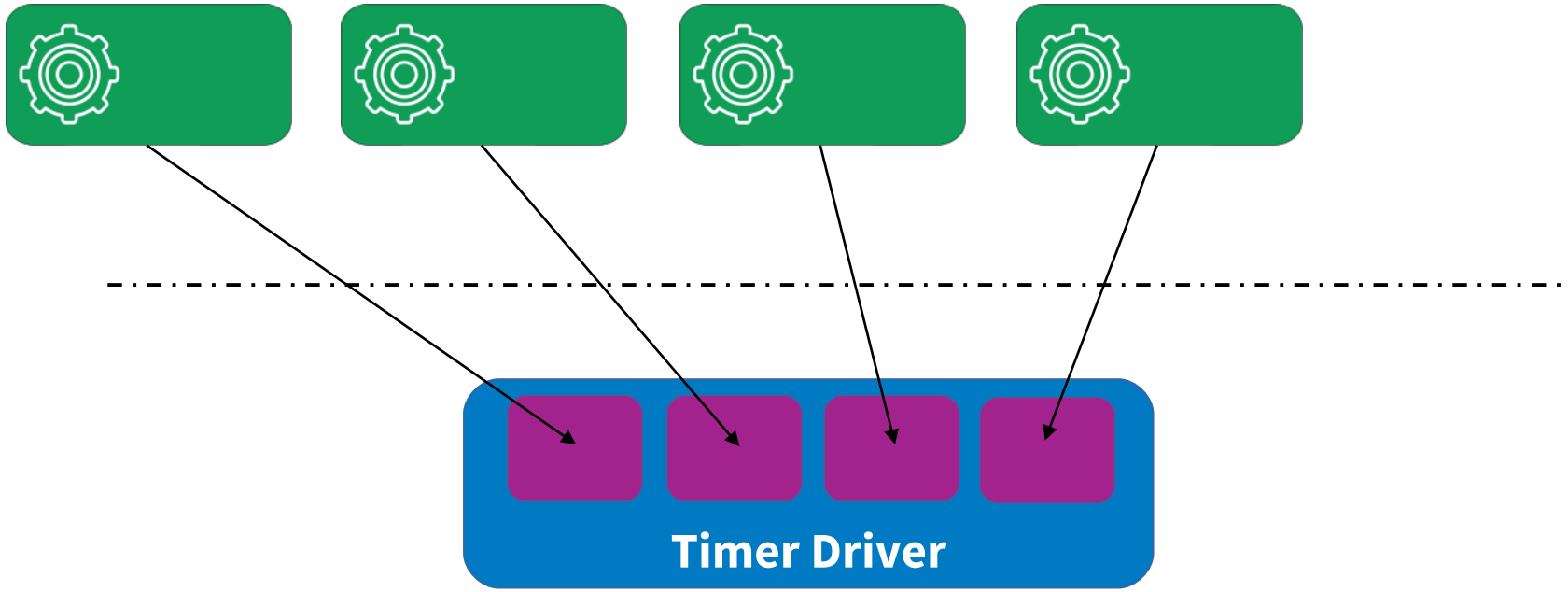
- Dynamic allocation leads to needs of unrelated drivers denying service to applications

Example of denial of service with timers



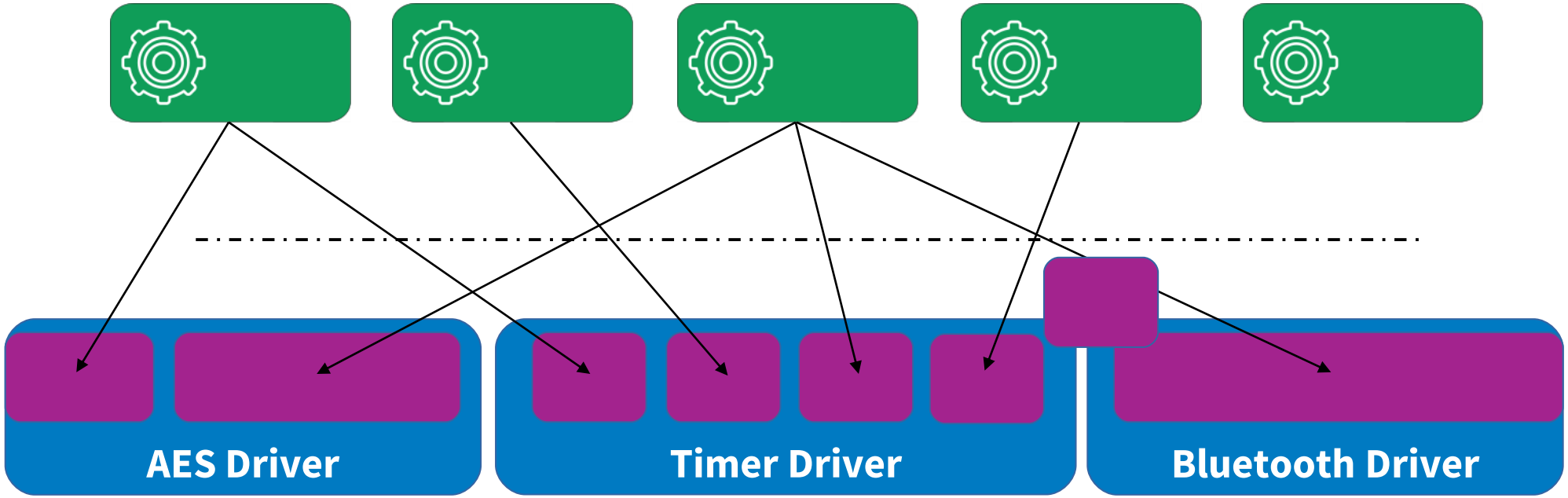
- Dynamic allocation leads to needs of unrelated drivers denying service to applications

Example of denial of service with timers



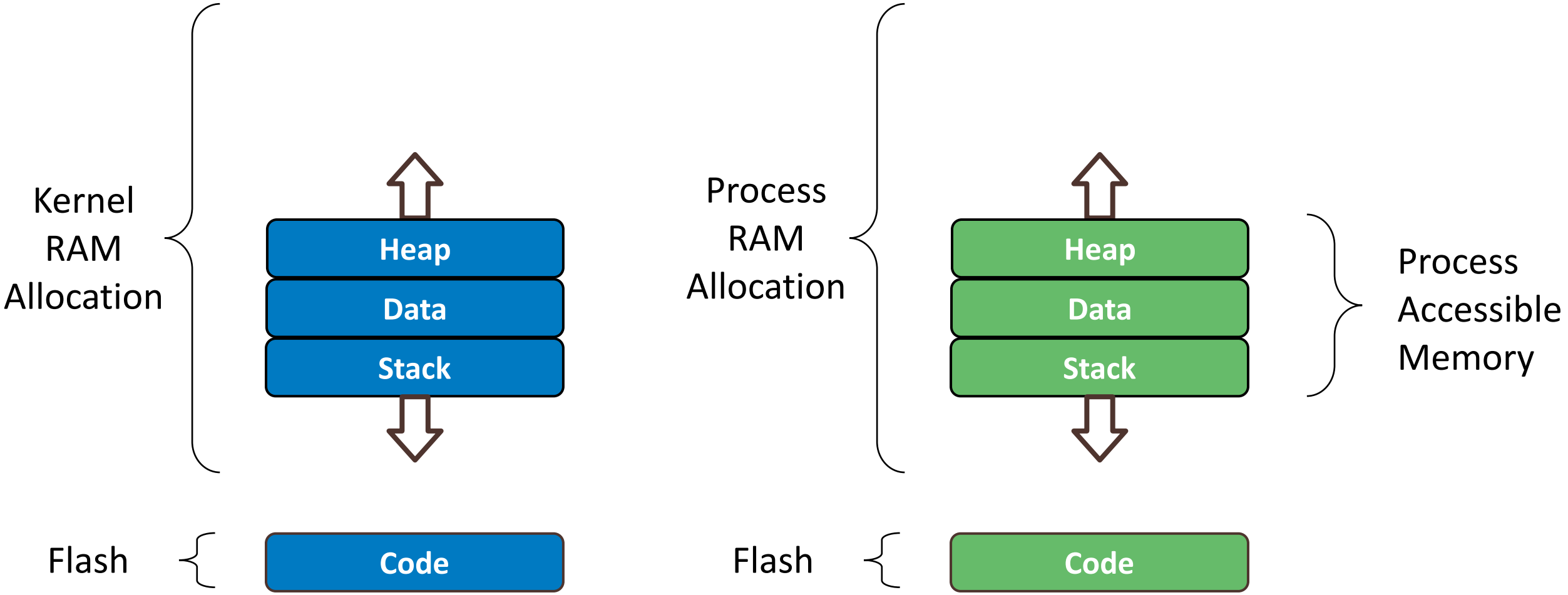
- Dynamic allocation leads to needs of unrelated drivers denying service to applications

Example of denial of service with timers

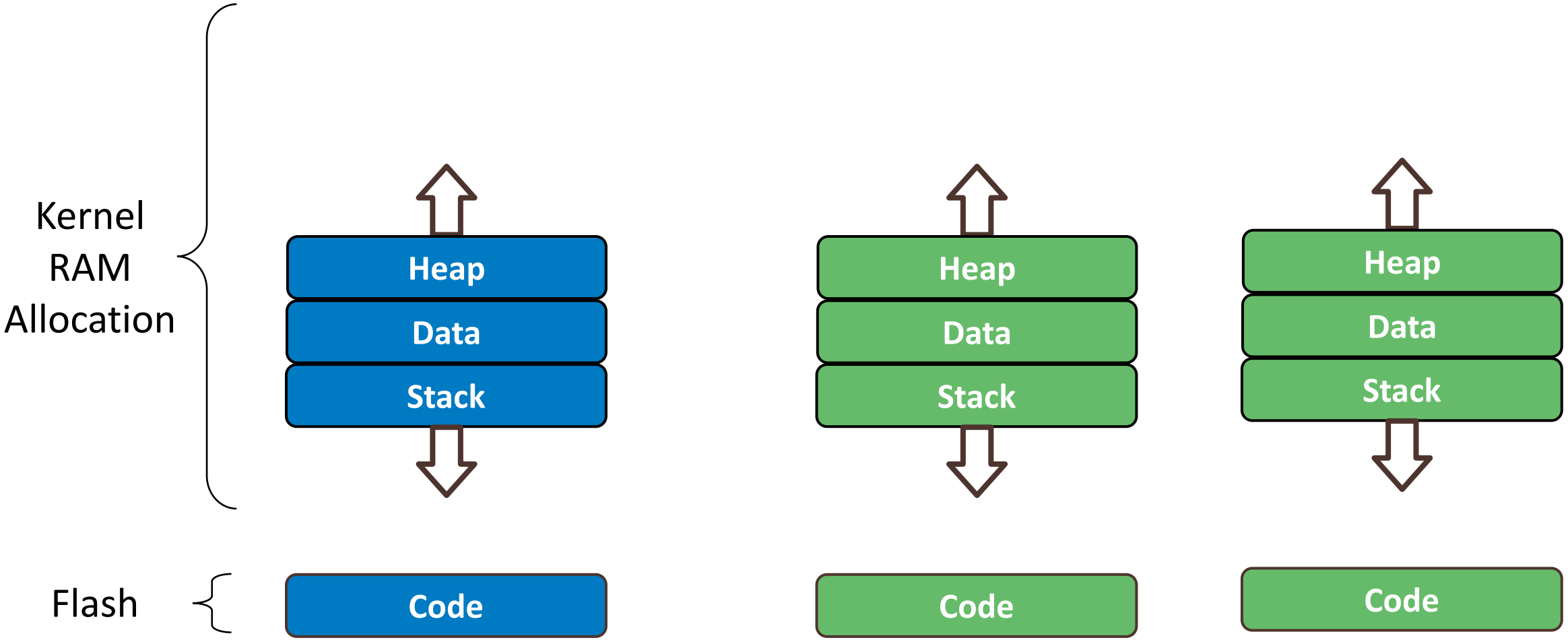


- Dynamic allocation leads to needs of unrelated drivers denying service to applications

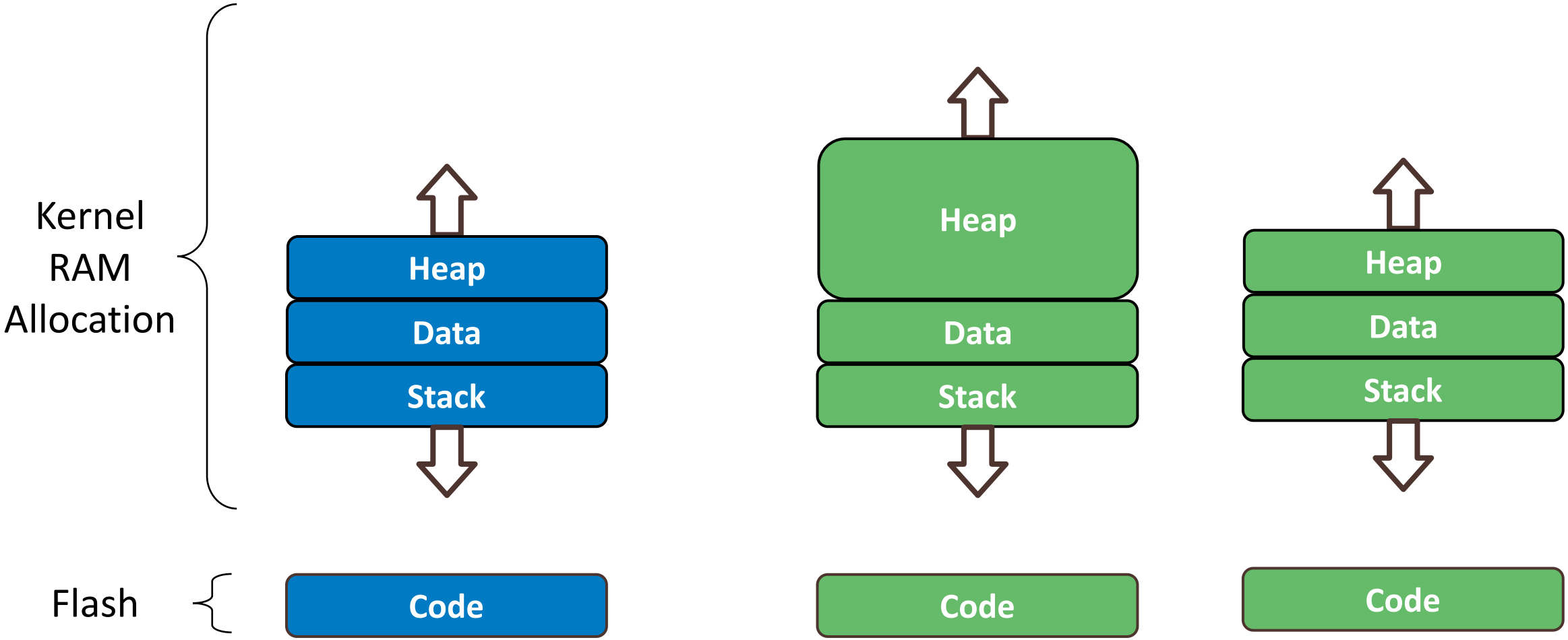
Traditional kernel heap allocation results in shared fate across all processes



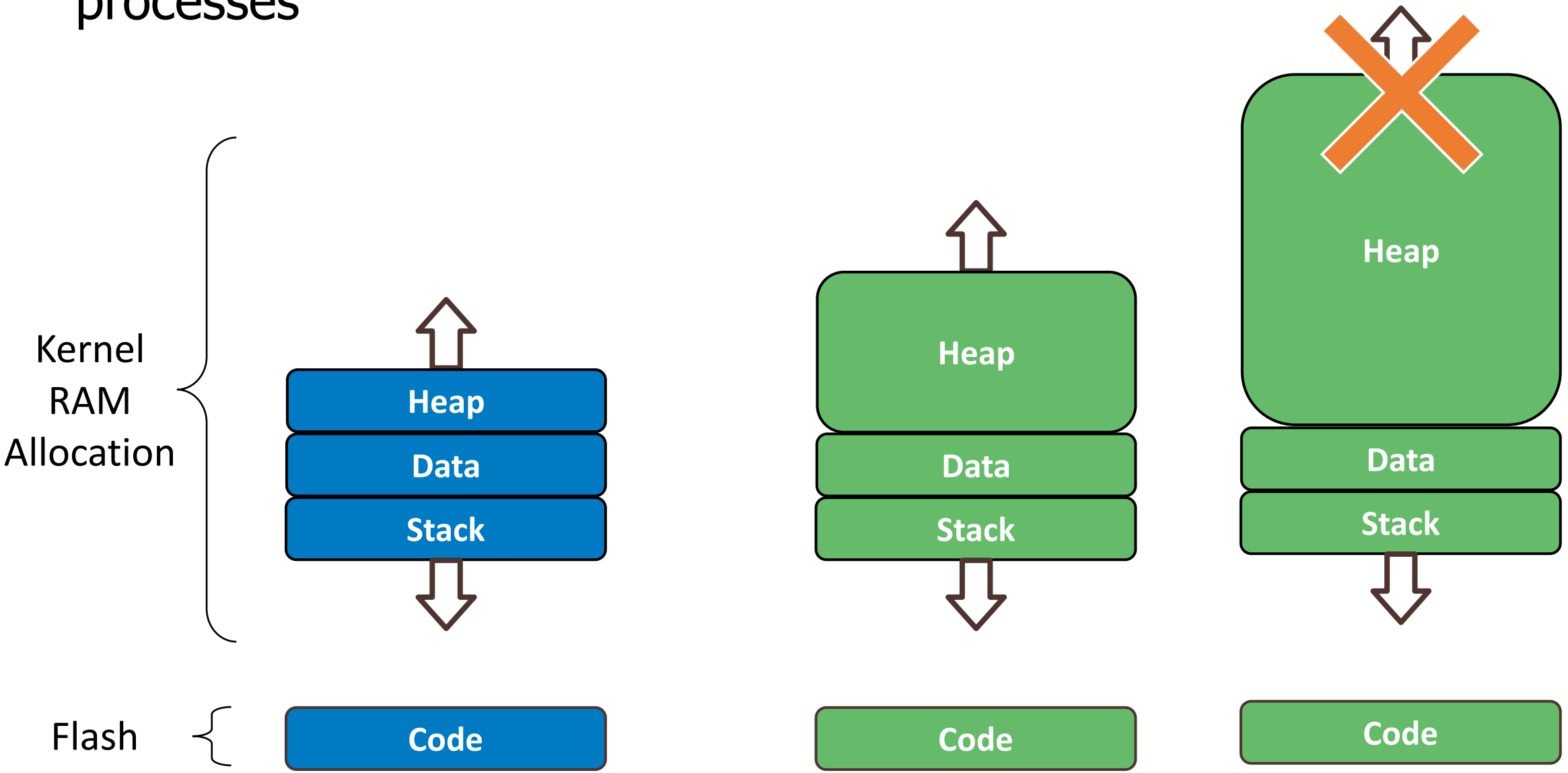
Traditional kernel heap allocation results in shared fate across all processes



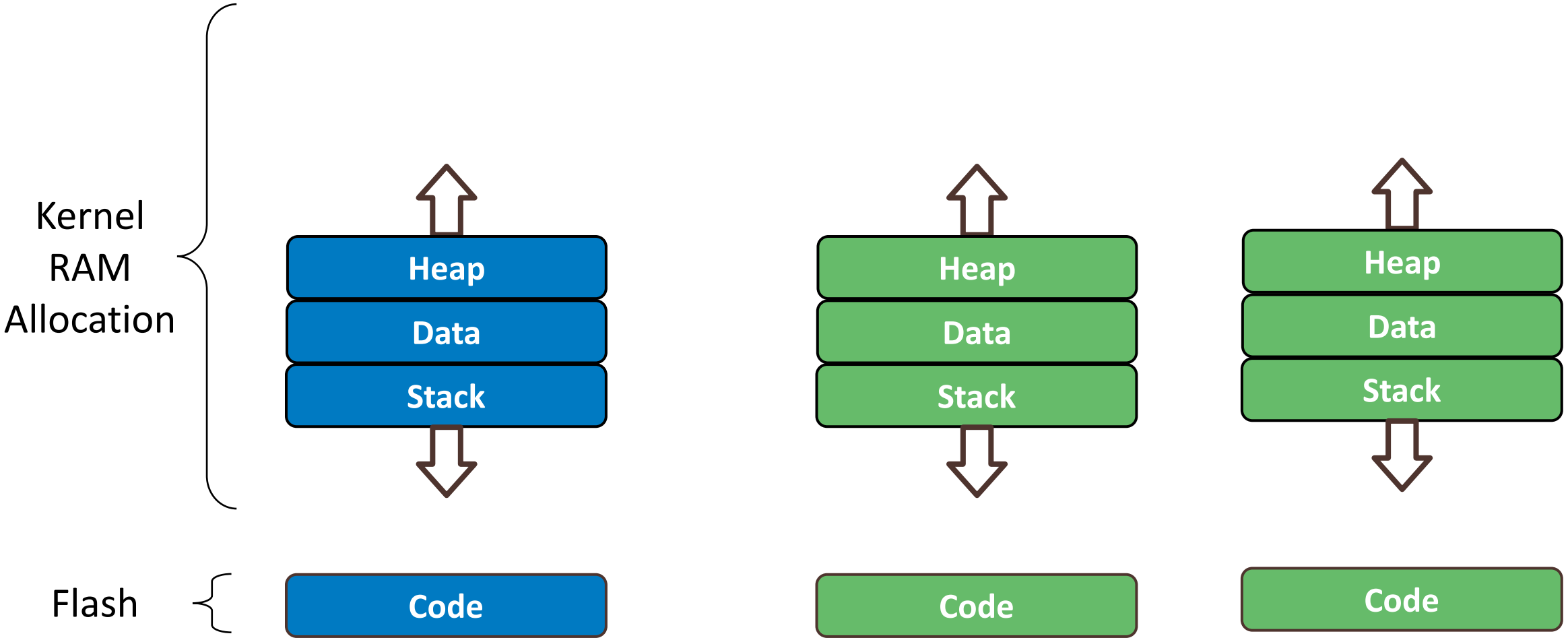
Traditional kernel heap allocation results in shared fate across all processes



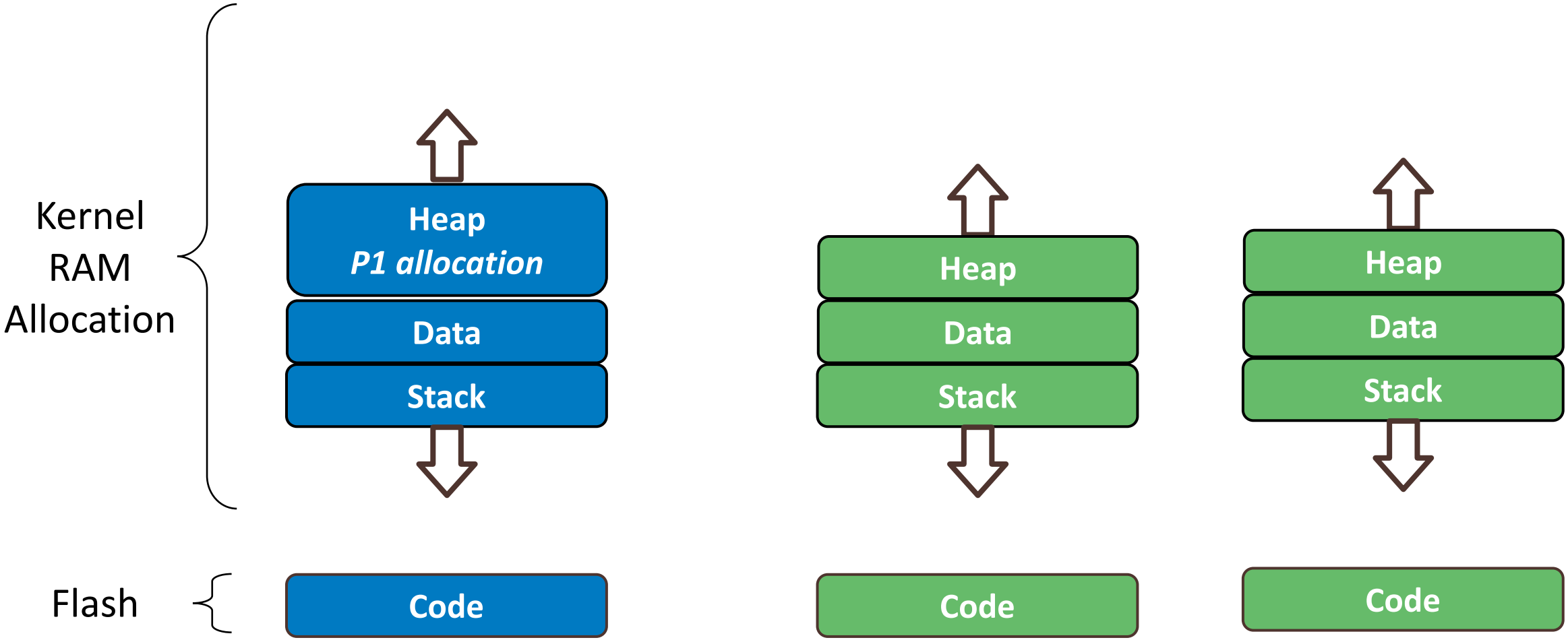
Traditional kernel heap allocation results in shared fate across all processes



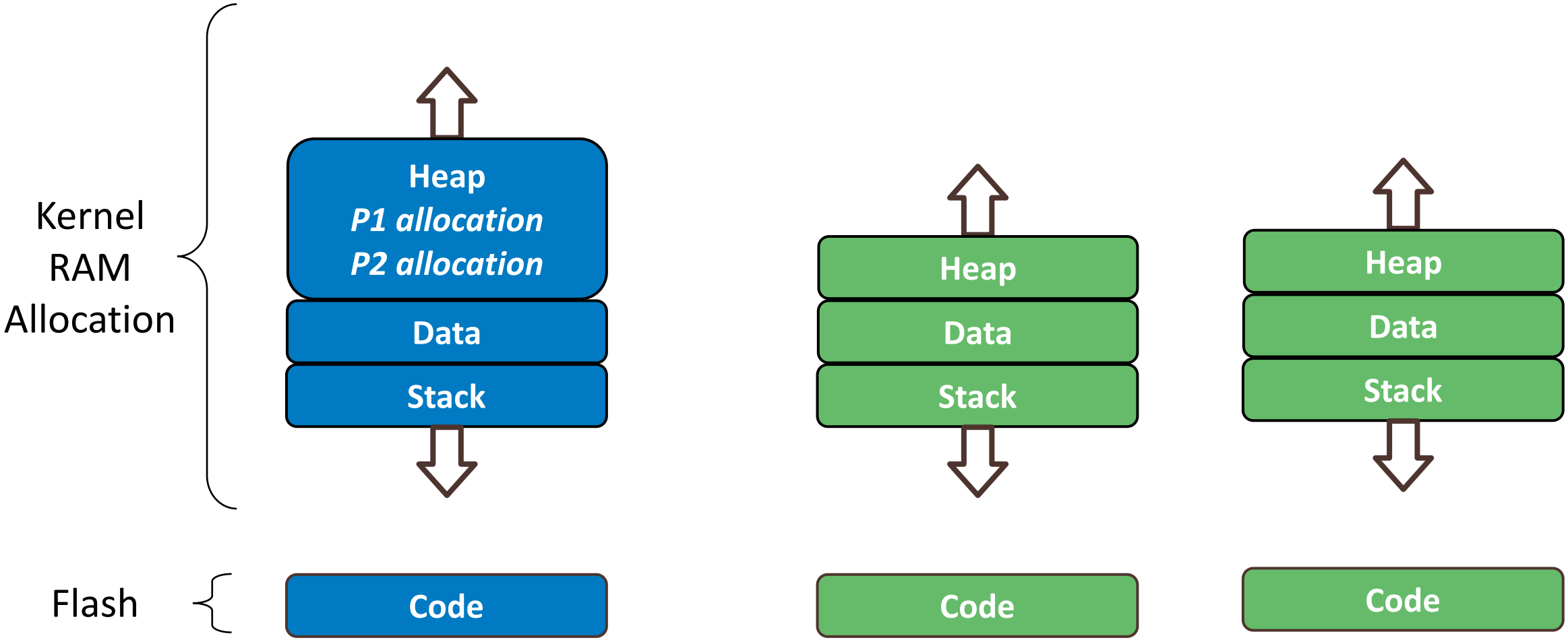
Traditional kernel heap allocation results in shared fate across all processes



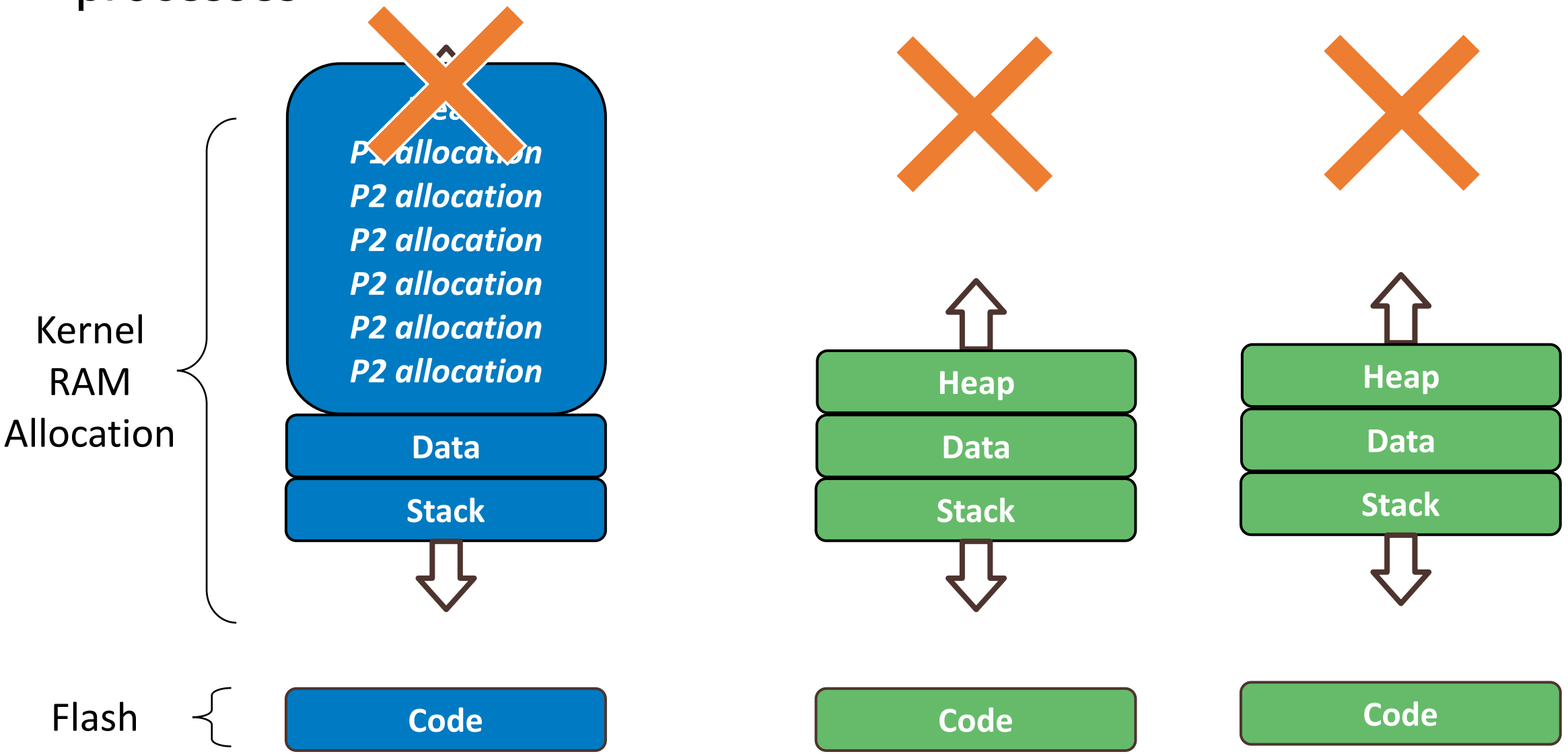
Traditional kernel heap allocation results in shared fate across all processes



Traditional kernel heap allocation results in shared fate across all processes

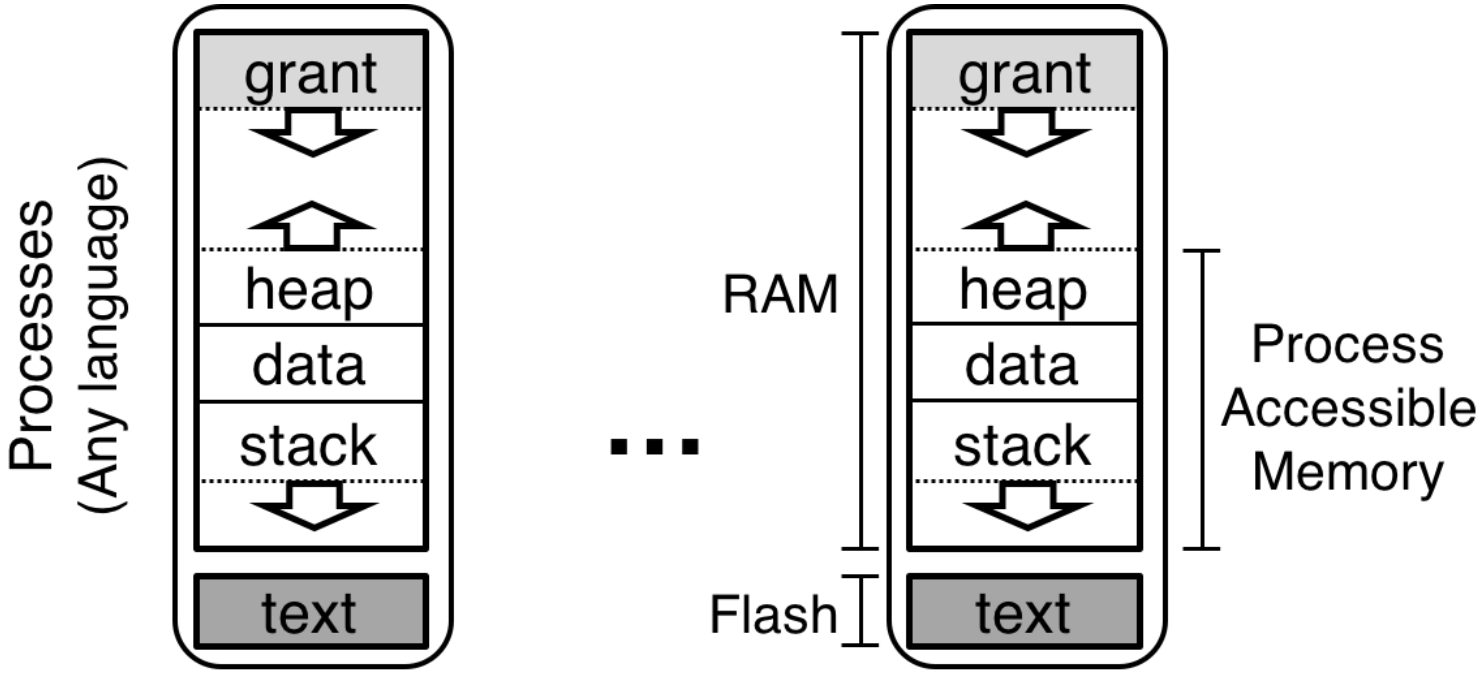


Traditional kernel heap allocation results in shared fate across all processes

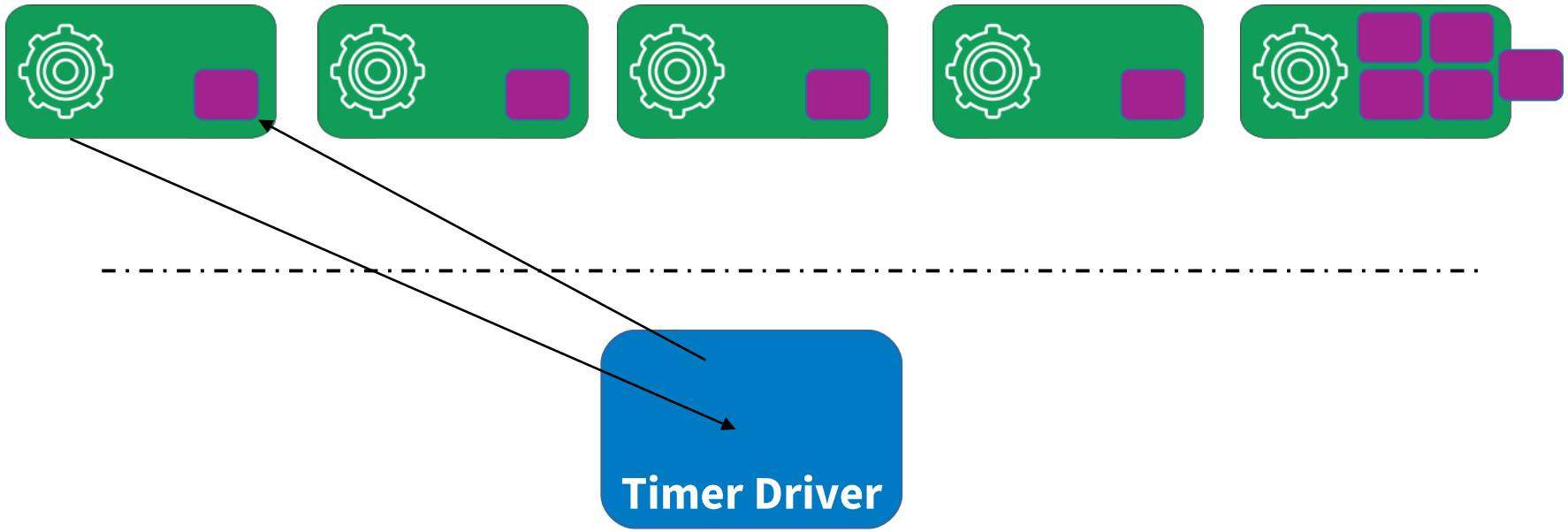


Solution: per-process kernel heaps

- Grant section is only accessible to the kernel
  - Used to store application-specific kernel structures



Grants balance reliability and flexibility



- Using too much memory only affects processes that do so
- No application can deny kernel services to another application

Problems to deal with in applications

1. How do we prevent malicious accesses without virtual memory?
   - With Memory Protection Unit hardware

2. How do we load multiple applications without virtual memory?
   - With Position Independent Code

3. How do we manage having so little memory?
   - With Grant regions

Tock research contributions

- Design of an OS kernel using language safety features
  - What parts of this were particularly challenging?

- Design of a multiprogramming OS under severe memory restrictions
  - No virtual memory
  - ~64 KB of RAM for the entire system (kernel and applications)
    - Grants as a solution for reliable flexibility

- "Multiprogramming a 64 kB Computer Safely and Efficiently" Levy et al. 2017. Symposium on Operating Systems Principles. https://brandenghena.com/projects/tock/levy17multiprogramming.pdf

Outline

- Embedded Systems
- Embedded Operating Systems
- Tock
  - Overview
  - Designing a secure kernel
  - Designing secure applications