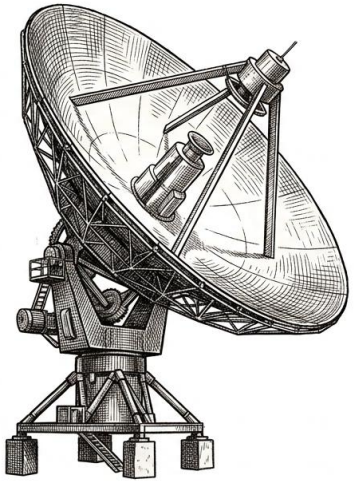
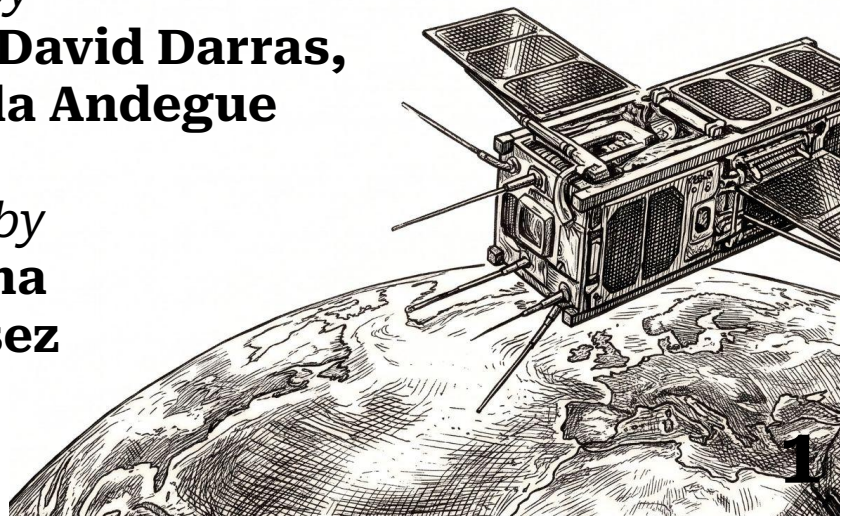
xOS

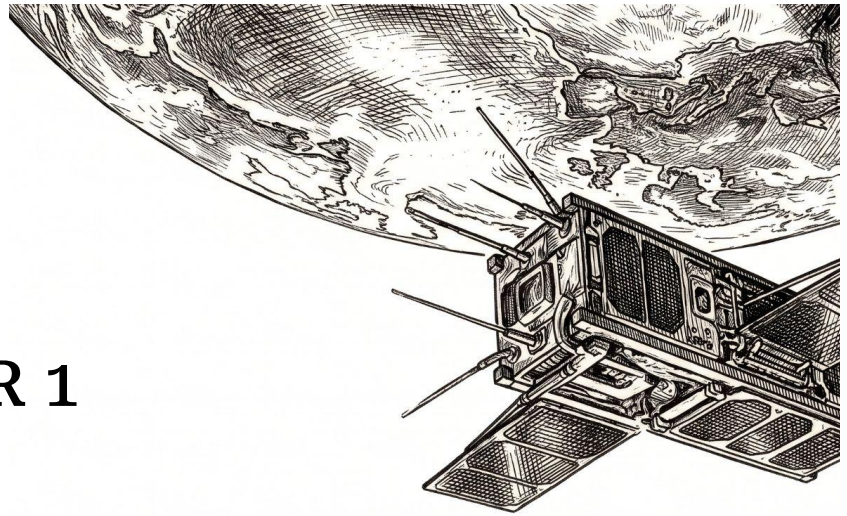
Rethinking Isolation for Nanosatellites

Presented by
Alexander Yanovskyy, David Darras,
François Lionnel Bala Andegue

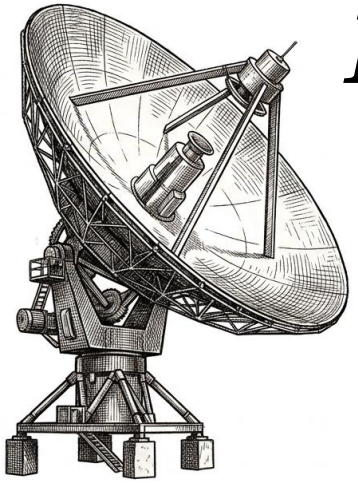
Supervised by
Alain Tchana
Didier Donsez







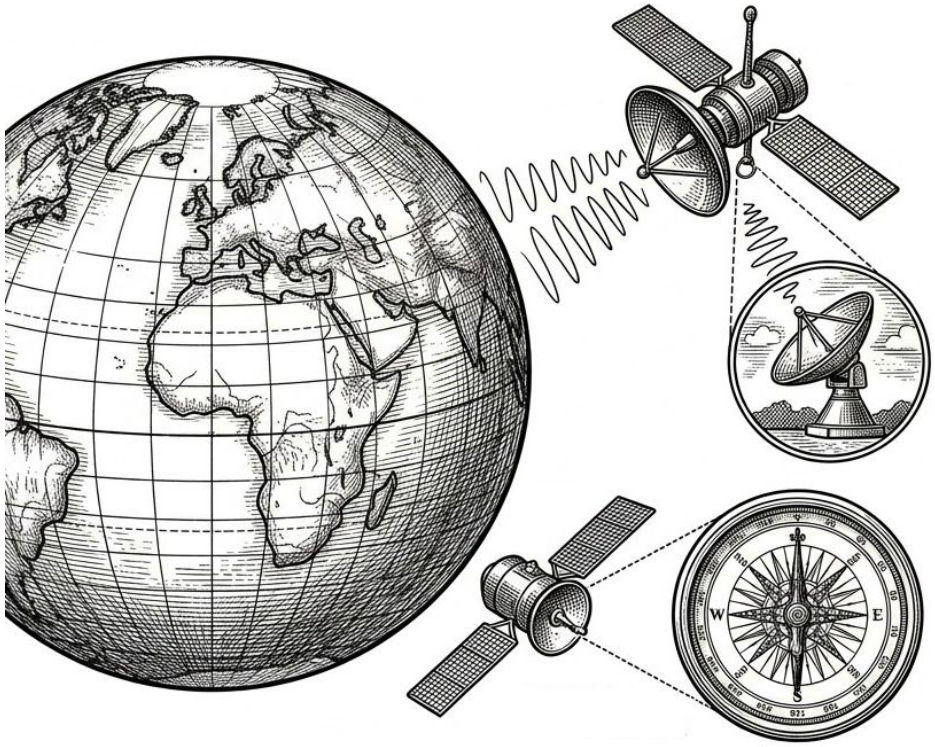
CHAPTER 1



Landscape & Challenges

Why Do We Go to Space ?

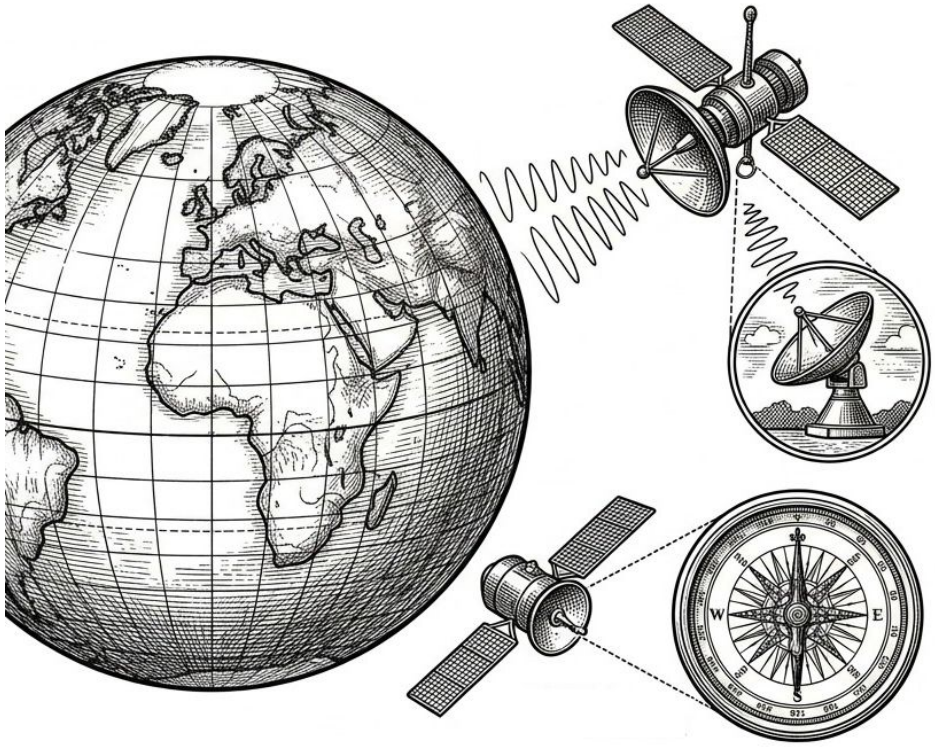
Why Do We Go to Space ?



Earth Observation

Monitoring climate change, optimizing agriculture, maritime tracking, and military surveillance.

Why Do We Go to Space ?



Earth Observation

Monitoring climate change, optimizing agriculture, maritime tracking, and military surveillance.

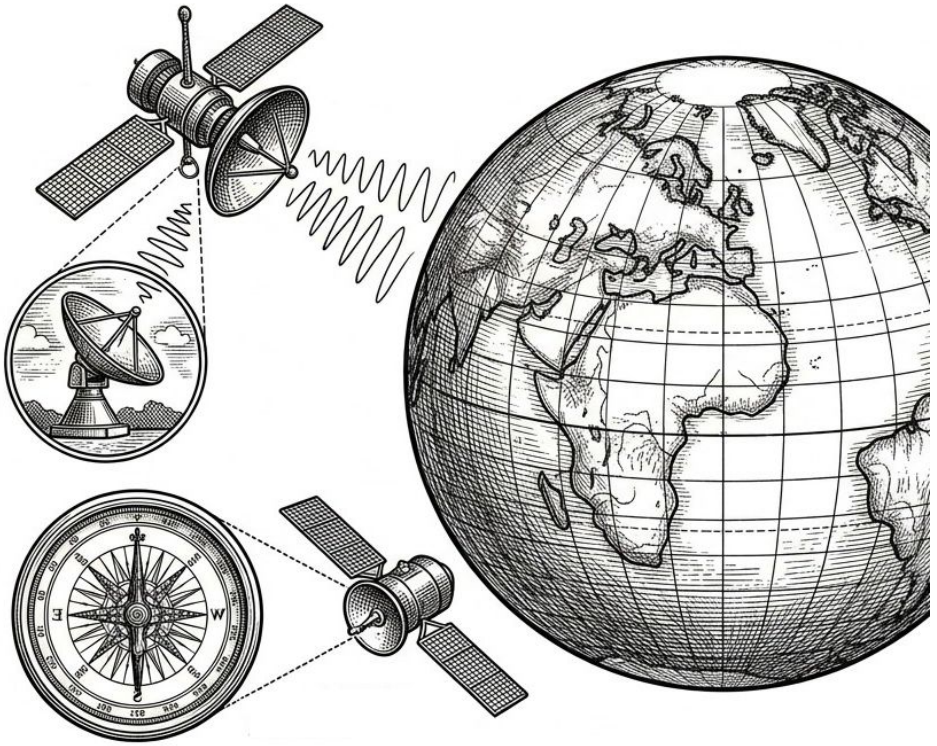
Telecommunications

Providing global broadband internet to remote areas, television broadcasting, and secure defense communications.

Why Do We Go to Space ?

Navigation

Powering Global Navigation Satellite Systems that synchronize global logistics, aviation, and our daily smartphone maps.



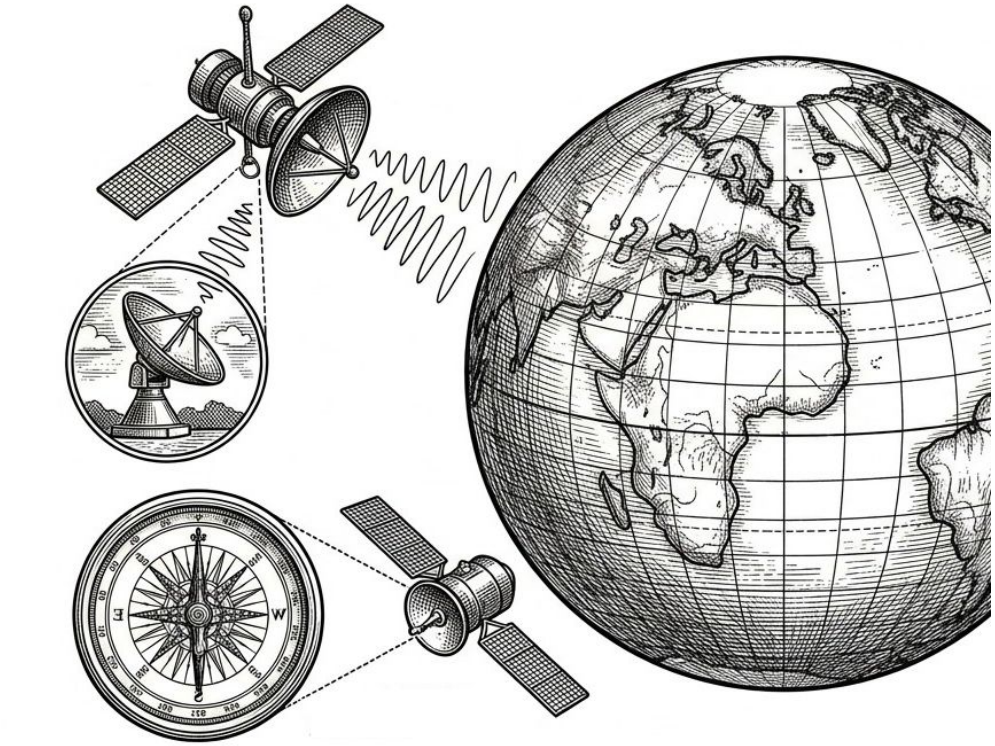
Why Do We Go to Space ?

Navigation

Powering Global Navigation Satellite Systems that synchronize global logistics, aviation, and our daily smartphone maps.

Scientific Research

Deploying space telescopes free from atmospheric distortion, and using the microgravity environment to test new materials or biological behaviors.



The Old Space Paradigm

Objective

• *Zero-Failure Tolerance*

• *Long-Term Lifespan* (15+ years): In deep space with geostationary telecommunications.

The Old Space Paradigm

Objective

• *Zero-Failure Tolerance*

• *Long-Term Lifespan* (15+ years): In deep space with geostationary telecommunications.

Methodology

• *Formal Verification*: Mathematical proof of code correctness using *Ada*

• *V-Model Lifecycle*: 10+ years of exhaustive ground testing before a single launch.

The Old Space Paradigm

Objective

• *Zero-Failure Tolerance*

• *Long-Term Lifespan* (15+ years) : In deep space with geostationary telecommunications.

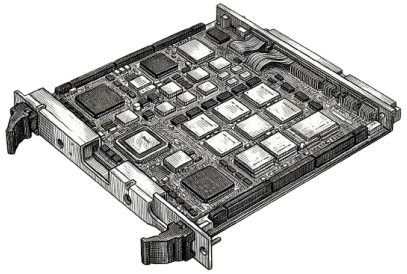
Methodology

• *Formal Verification*: Mathematical proof of code correctness using *Ada*.

• *V-Model Lifecycle*: 10+ years of exhaustive ground testing before a single launch.

Hardware

• *Rad-Hard:* Purpose-built processors immune to space radiation.



The Old Space Paradigm

Objective

• *Zero-Failure Tolerance*

• *Long-Term Lifespan* (15+ years) : In deep space with geostationary telecommunications.

Methodology

• *Formal Verification*: Mathematical proof of code correctness using *Ada*.

• *V-Model Lifecycle*: 10+ years of exhaustive ground testing before a single launch.

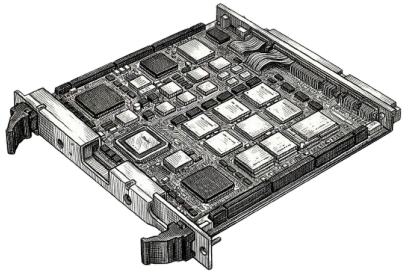
Hardware

• *Rad-Hard:* Purpose-built processors immune to space radiation.

Constraints

• *Astronomical Costs:* Budgets exceeding $100M+ per satellite.

• *Technological Lag:* Flight hardware is often 10 to 15 years behind consumer technology by launch day.



The New Space Paradigm

Objective

• *Fail-Fast & Iterate*

• *Short-Term Missions*: Low Earth Orbit missions lasting 6 months to 2 years.

• *Democratization*: Enabling universities and startups to launch satellite constellations.

The New Space Paradigm

Objective

• *Fail-Fast & Iterate*

• *Short-Term Missions*: Low Earth Orbit missions lasting 6 months to 2 years.

• *Democratization*: Enabling universities and startups to launch satellite constellations.

Hardware

• Using *COTS* electronics (IoT tech).

• *Radiation Vulnerability*: High susceptibility to SEEs and TID.

• *Standardization*: The CubeSat form factor (1U = 10x10x10 cm) drastically reduces launch costs.

The New Space Paradigm

Objective

- *Fail-Fast & Iterate*
- *Short-Term Missions*: Low Earth Orbit missions lasting 6 months to 2 years.
- *Democratization*: Enabling universities and startups to launch satellite constellations.

Hardware

- Using *COTS* electronics (IoT tech).
- *Radiation Vulnerability*: High susceptibility to SEEs and TID.
- *Standardization*: The CubeSat form factor (1U = 10x10x10 cm) drastically reduces launch costs.

Methodology & Software

- *Agile Development*
- *Modern Languages*: C++, Rust, Python

Embedded Systems Challenges

1. Memory & Storage Limits

KB-scale RAM and Flash requiring ultra-lightweight footprint optimization and XIP.

Embedded Systems Challenges

1. Memory & Storage Limits

KB-scale RAM and Flash requiring ultra-lightweight footprint optimization and XIP.

2. Strict Power Budgets

Harsh eclipse cycles and limited energy demanding aggressive duty-cycling.

Embedded Systems Challenges

1. Memory & Storage Limits

KB-scale RAM and Flash requiring ultra-lightweight footprint optimization and XIP.

2. Strict Power Budgets

Harsh eclipse cycles and limited energy demanding aggressive duty-cycling.

3. Deterministic Multitasking

Growing need to run multiple concurrent tasks with strict timing guarantees.

Embedded Systems Challenges

1. Memory & Storage Limits

KB-scale RAM and Flash requiring ultra-lightweight footprint optimization and XIP.

2. Strict Power Budgets

Harsh eclipse cycles and limited energy demanding aggressive duty-cycling.

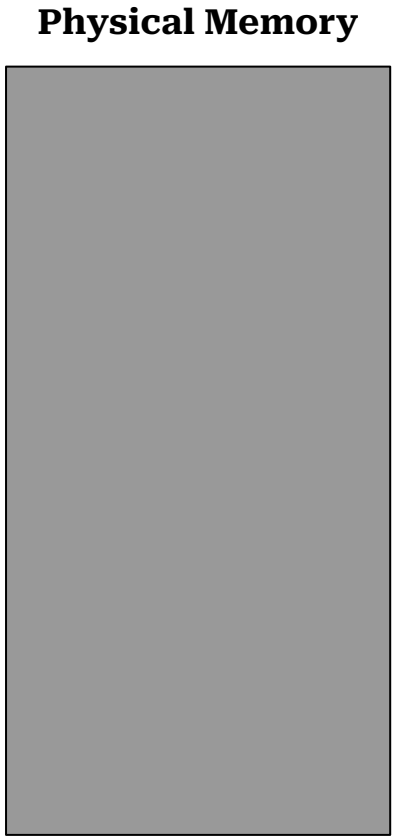
3. Deterministic Multitasking

Growing need to run multiple concurrent tasks with strict timing guarantees.

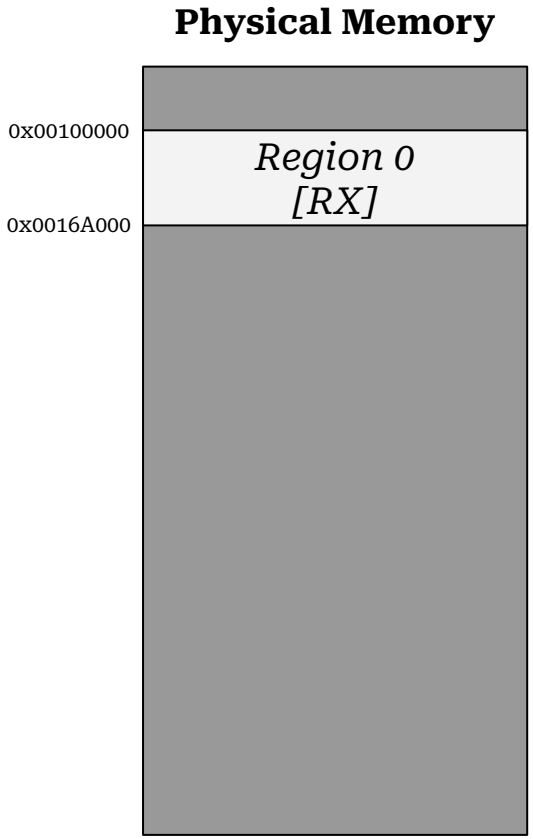
4. Hardware Isolation

No costly MMU; reliance on MPUs (limited to 8 power-of-two aligned regions), leading to memory fragmentation.

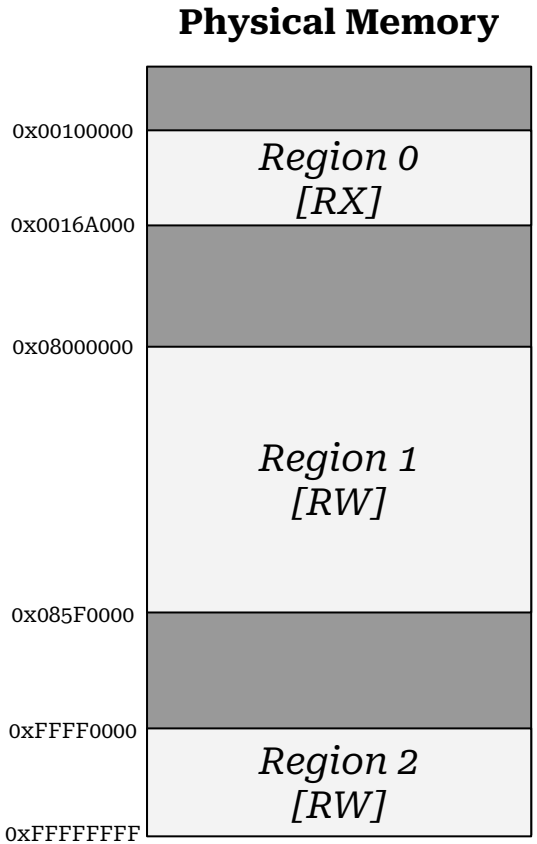
MPU (*Memory Protection Unit*)



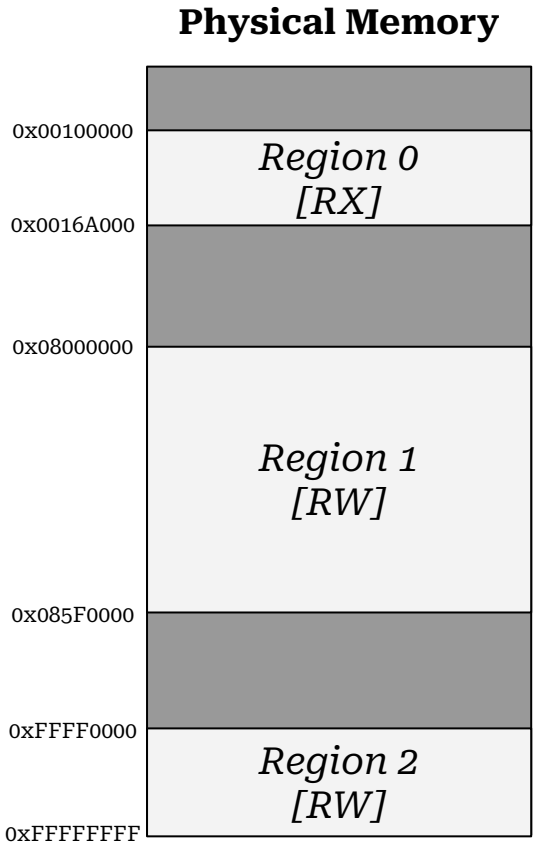
MPU (*Memory Protection Unit*)



MPU (*Memory Protection Unit*)



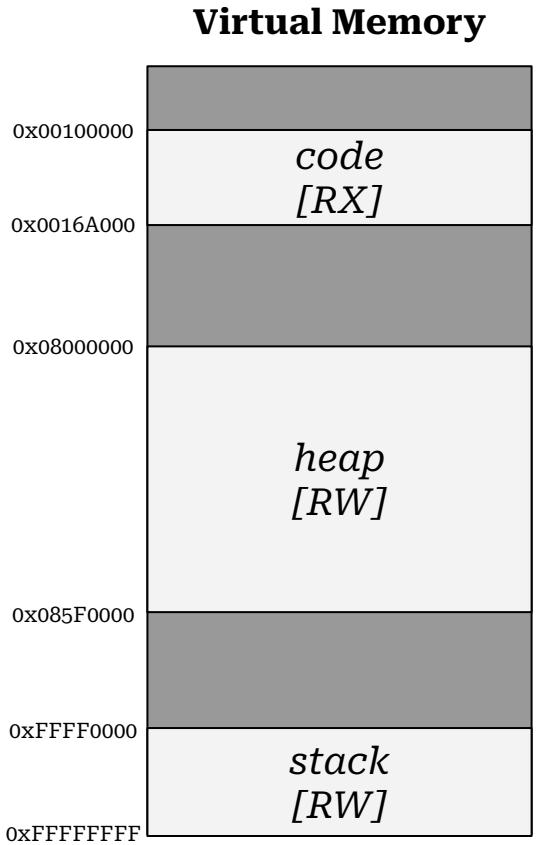
MPU (*Memory Protection Unit*)



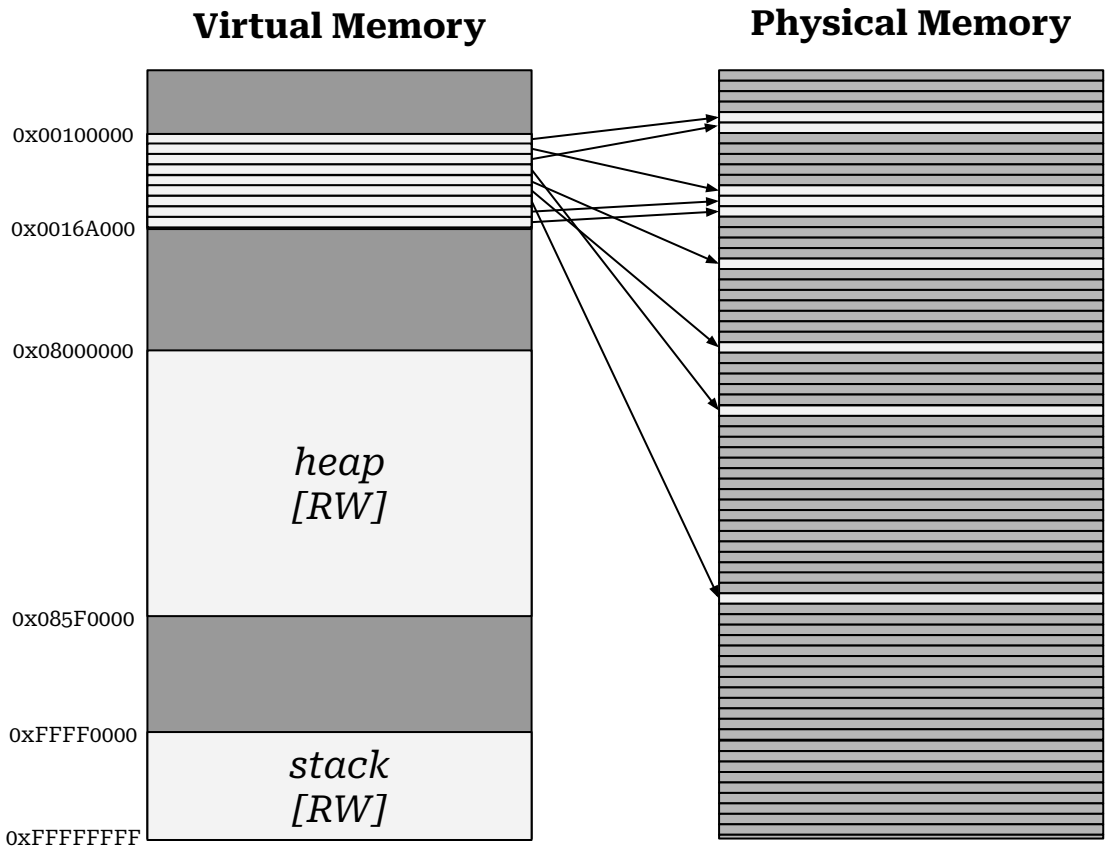
- No Address Translation
- Coarse-Grained Granularity (Regions)
- Fixed Region Limit (8 or 16)

- Deterministic Real-Time Performance
- Low Power Consumption

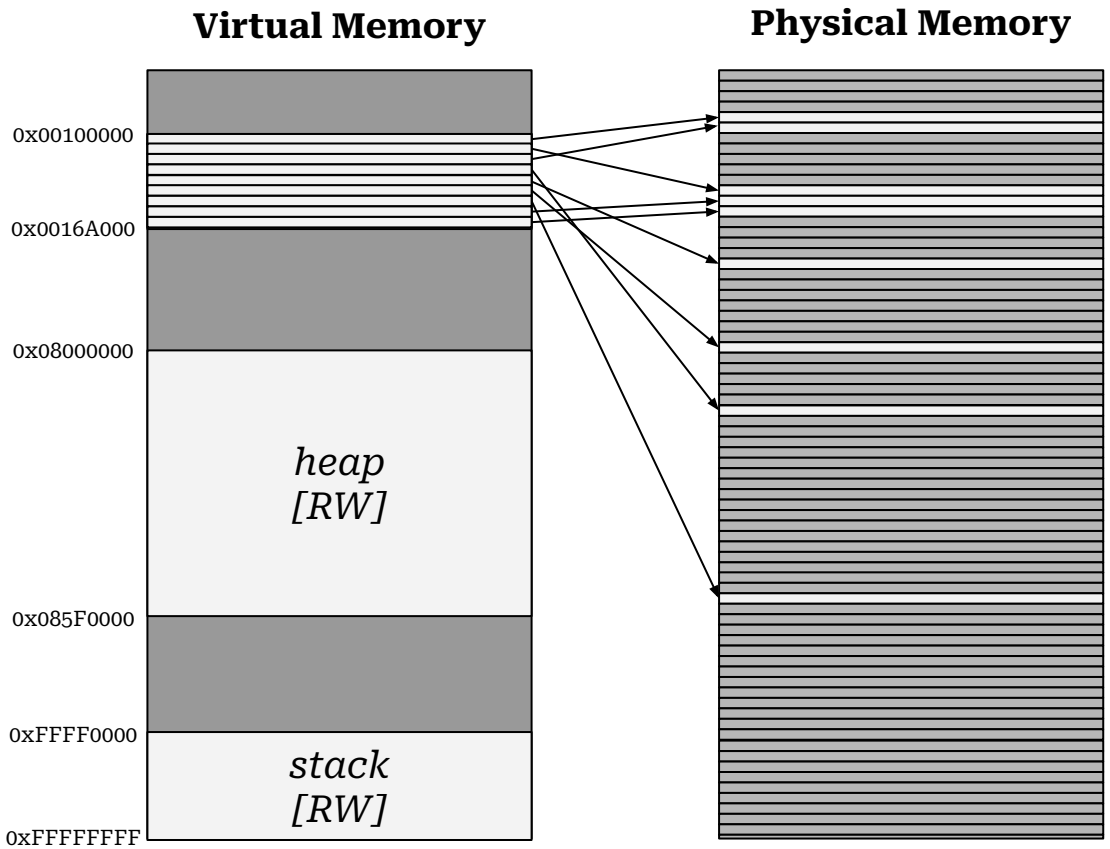
MMU (*Memory Management Unit*)



MMU (*Memory Management Unit*)



MMU (*Memory Management Unit*)



- Address Translation
- Fine-Grained Granularity (Pages)

⇒ Strong Process Isolation

- Non-Deterministic (TLB misses / Page faults)
- High Power Consumption
- Memory Overhead (Page Tables)

FreeRTOS - The Lightweight Standard

FreeRTOS - The Lightweight Standard

Highly Deterministic

Guarantees strict execution timing for critical operations.

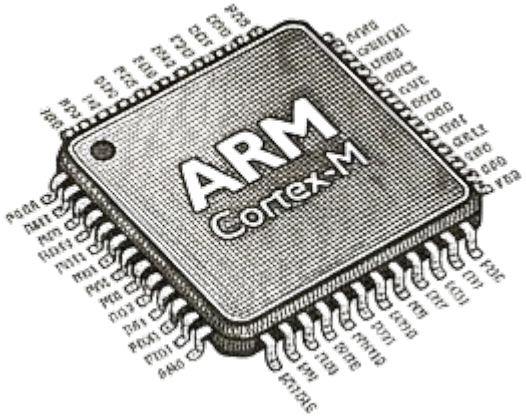
FreeRTOS - The Lightweight Standard

Highly Deterministic

Guarantees strict execution timing for critical operations.

Minimal Footprint

Can run on tiny, low-power MCU (ARM Cortex-M) with only a few kilobytes of RAM.



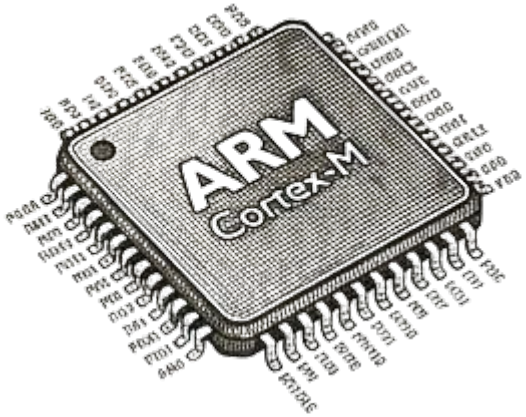
FreeRTOS - The Lightweight Standard

Highly Deterministic

Guarantees strict execution timing for critical operations.

Minimal Footprint

Can run on tiny, low-power MCU (ARM Cortex-M) with only a few kilobytes of RAM.



No Native Isolation (SAS)

All tasks share the same physical memory space by default.

FreeRTOS - The Lightweight Standard

Highly Deterministic

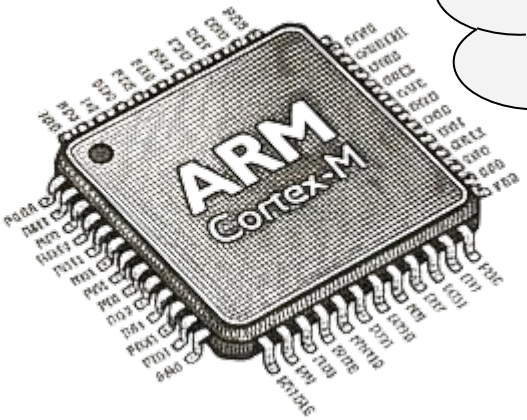
Guarantees strict execution timing for critical operations.

Minimal Footprint

Can run on tiny, low-power MCU (ARM Cortex-M) with only a few kilobytes of RAM.

FreeRTOS-MPU exists, but it is far too restrictive for developers, making it largely unused in practice.

.



No Native Isolation (SAS)

All tasks share the same physical memory space by default.

FreeRTOS - The Lightweight Standard

Highly Deterministic

Guarantees strict execution timing for critical operations.

Minimal Footprint

Can run on tiny, low-power MCU (ARM Cortex-M) with only a few kilobytes of RAM.

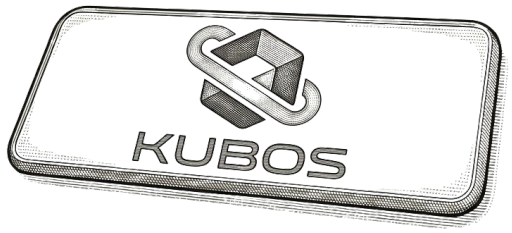
Memory-Unsafe Languages (C/C++)

A crash in a single task instantly leads to a catastrophic, system-wide failure.

No Native Isolation (SAS)

All tasks share the same physical memory space by default.

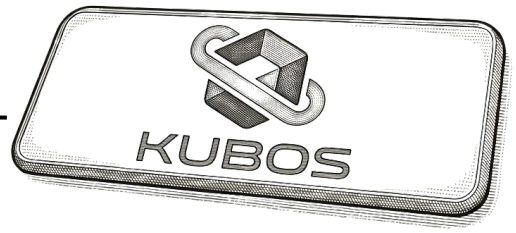
KubOS Linux - The Heavyweight Challenger



KubOS Linux - The Heavyweight Challenger

Process & Thread Model

Based on a desktop-class architecture



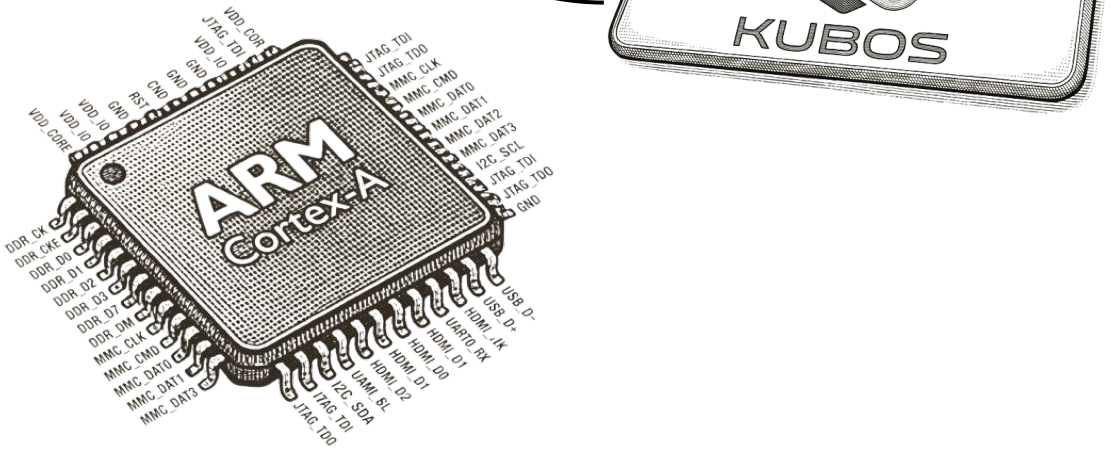
KubOS Linux - The Heavyweight Challenger

Process & Thread Model

Based on a desktop-class architecture

Hardware & Cost Constraints

- Requires external DDR RAM (64–128 MB) and SPI Flash (8–32 MB).
- Power consumption increases to 500 mW – 2 W



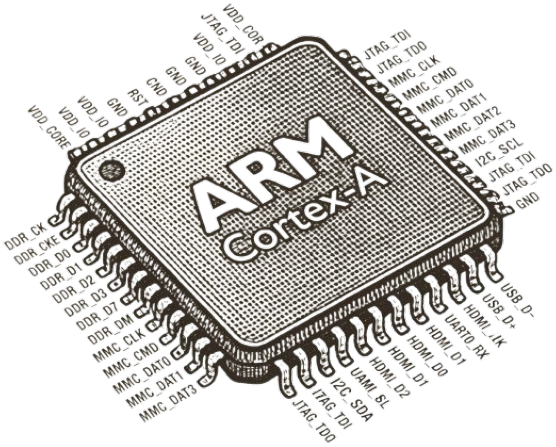
KubOS Linux - The Heavyweight Challenger

Process & Thread Model

Based on a desktop-class architecture

Hardware & Cost Constraints

- Requires external DDR RAM (64–128 MB) and SPI Flash (8–32 MB).
- Power consumption increases to 500 mW – 2 W



KUBOS

Rock-Solid Isolation

- Mandatory MMU
- Memory-safe language (Rust)

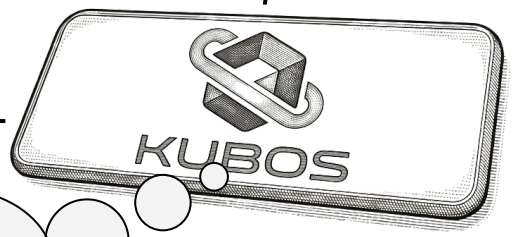
KubOS Linux - The Heavyweight Challenger

Process & Thread Model

Based on a desktop-class architecture

Hardware & Cost Constraints

- Requires external DDR RAM (64–128 MB) and SPI Flash (8–32 MB).
- Power consumption increases to 500 mW – 2 W

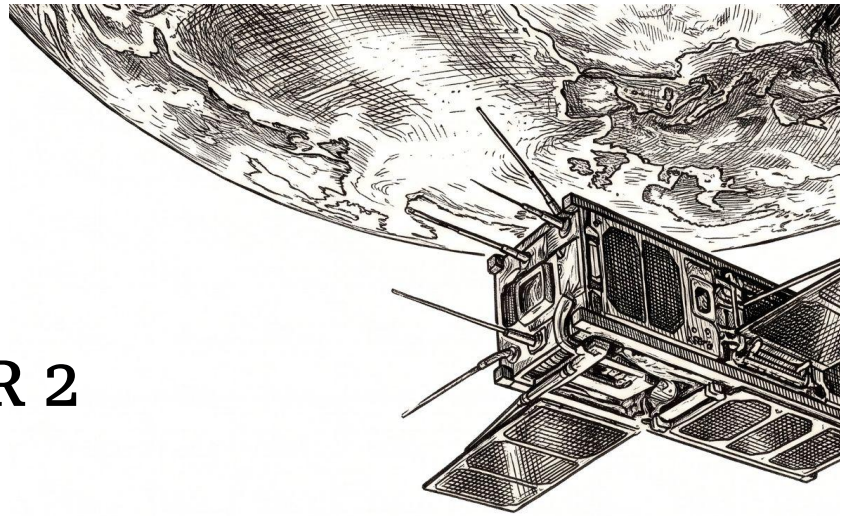


Inflexibility

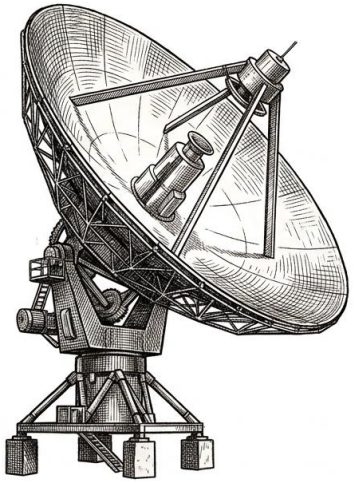
KubOS Linux is hardware -bound; KubOS-RT was built for MPUs, yet requiring a new OS per isolation mechanism kills flexibility.

Rock-Solid Isolation

- Mandatory MMU
- Memory-safe language (Rust)



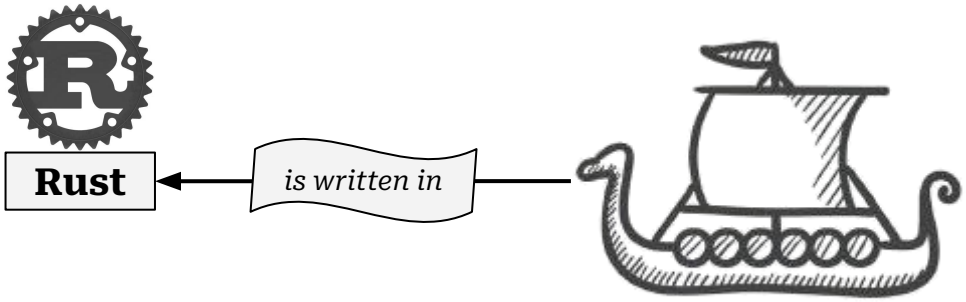
CHAPTER 2



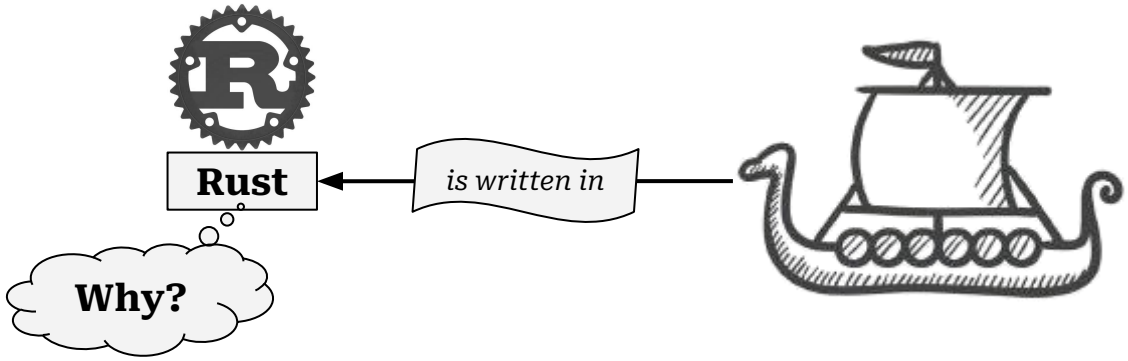
Our Inspirations

Theseus OS

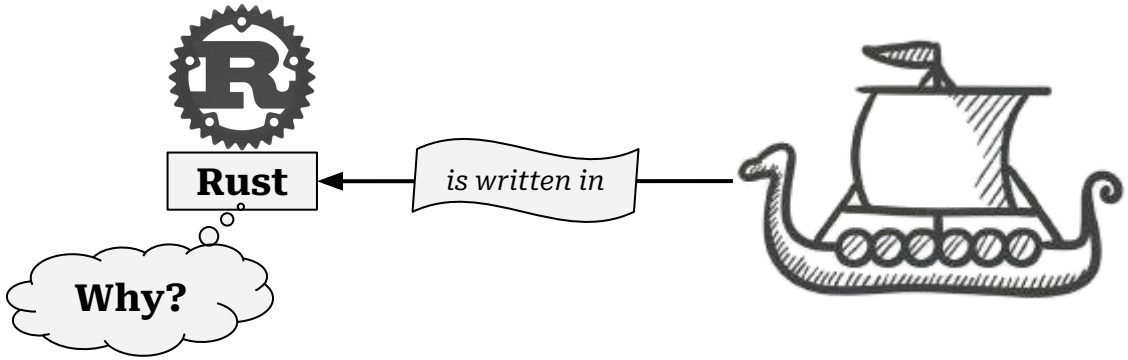
Theseus OS



Theseus OS

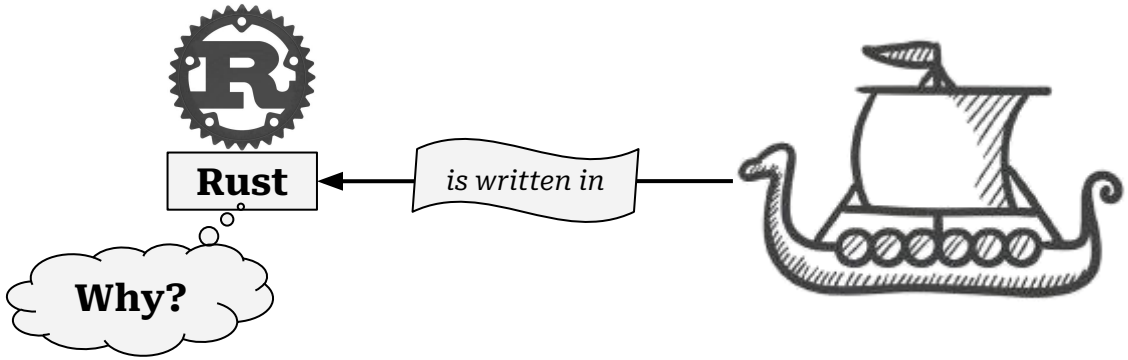


Theseus OS



Compile-time[1]
memory safety[2]
& **type safety[3]**

Theseus OS



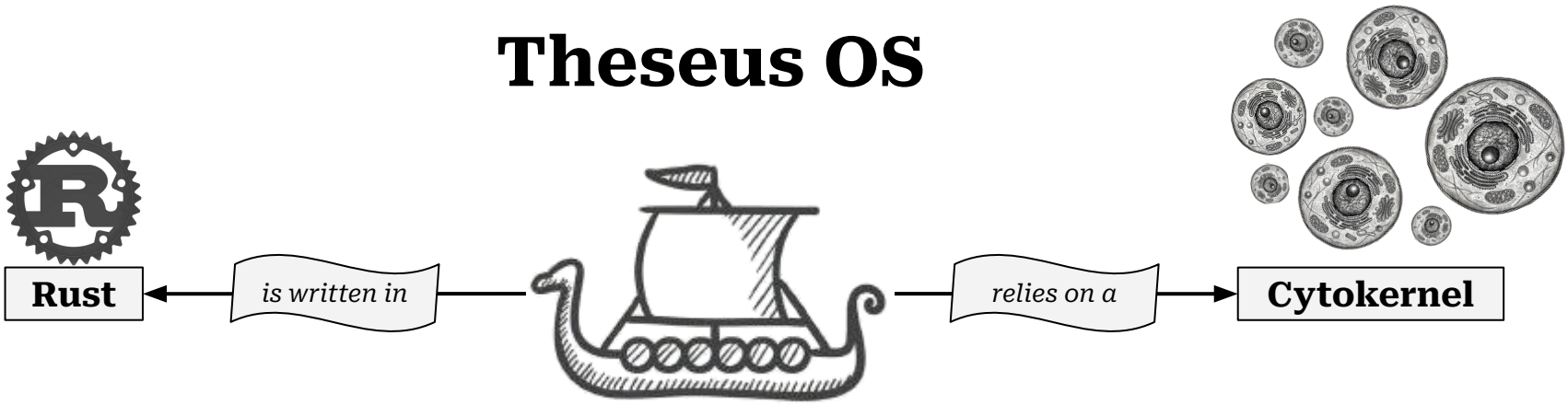
Compile-time[1]
memory safety[2]
& **type safety**[3]

[1] ⇒ No runtime overhead

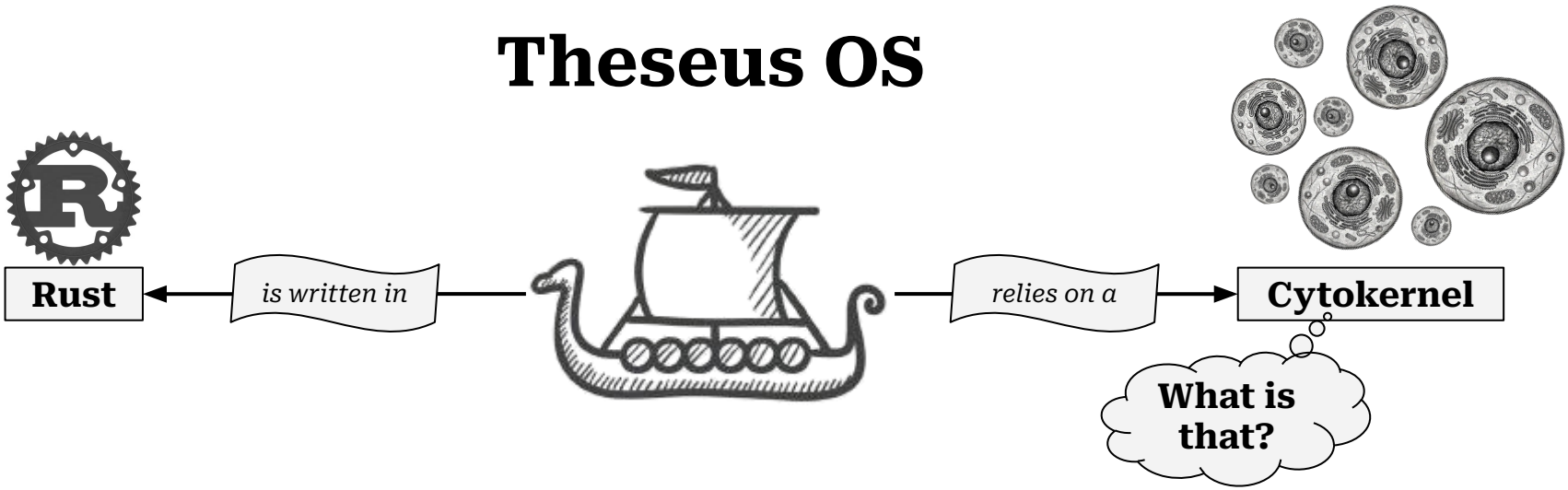
[2] ⇒ No *memory leaks*, *null pointer dereferences*, *use-after-free*, *buffer overflows*, *double frees*, etc., like in C, nor the need for a *garbage collector* like Ocaml thanks to **Ownership**, **Borrowing** & **Lifetimes** (enforced by the borrow checker)

[3] ⇒ No *invalid casts* and prevention of *data races* thanks to the type system (**Send/Sync traits)**

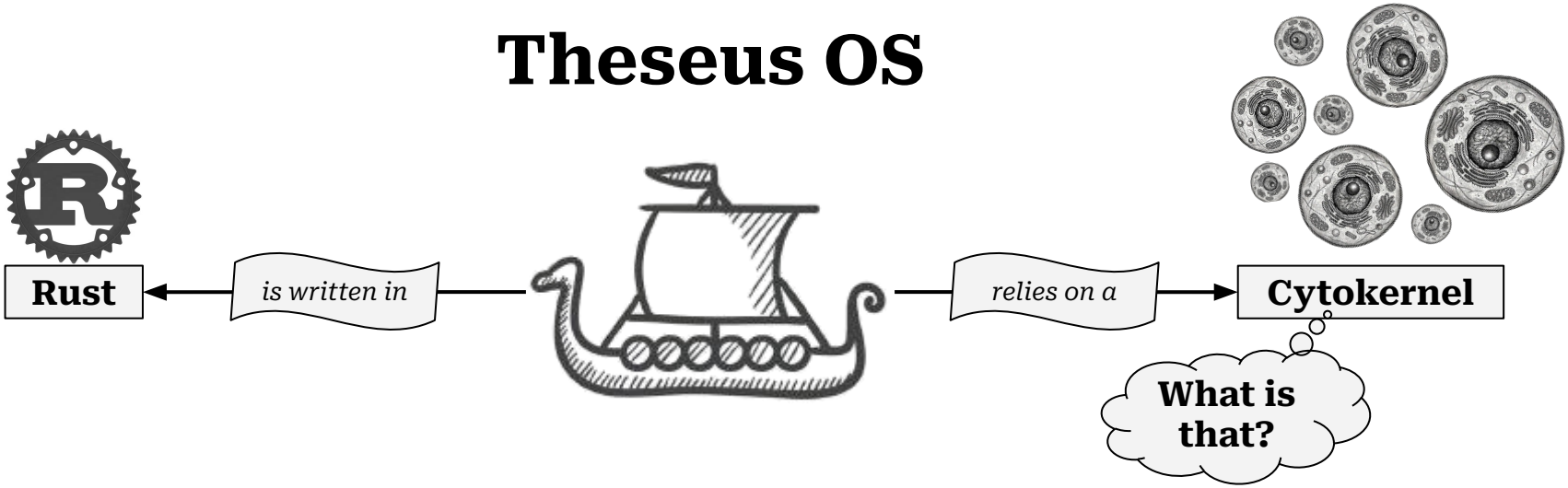
Theseus OS



Theseus OS



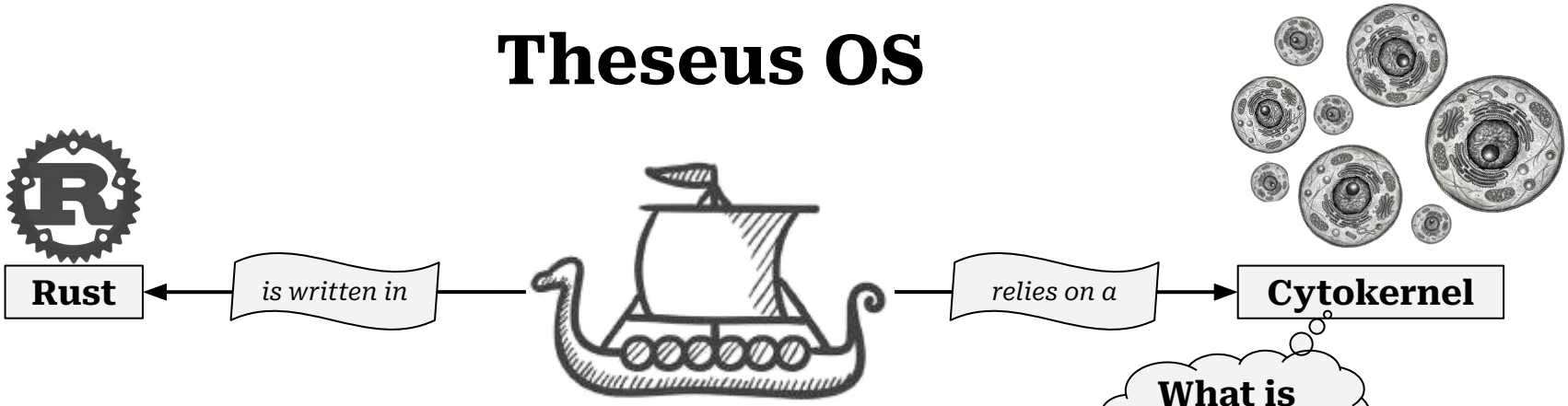
Theseus OS



- An **ultra-minimalist kernel** tasked solely with *state management* and *component loading*.

- An OS broken down into *tiny*, *independent logical units* called Cells (= Rust crates).

Theseus OS

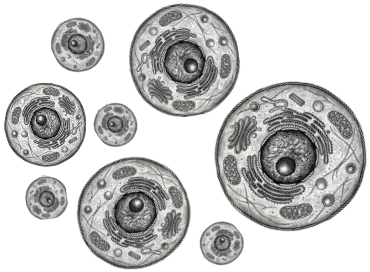


- An **ultra-minimalist kernel** tasked solely with *state management* and *component loading*.

- An OS broken down into *tiny*, *independent logical units* called Cells (= Rust crates).

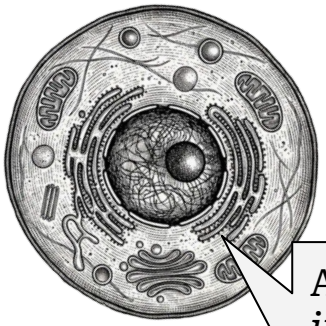
Theseus OS

Rust — *is written in* —  — *relies on a* — **Cytokernel**



What is that?

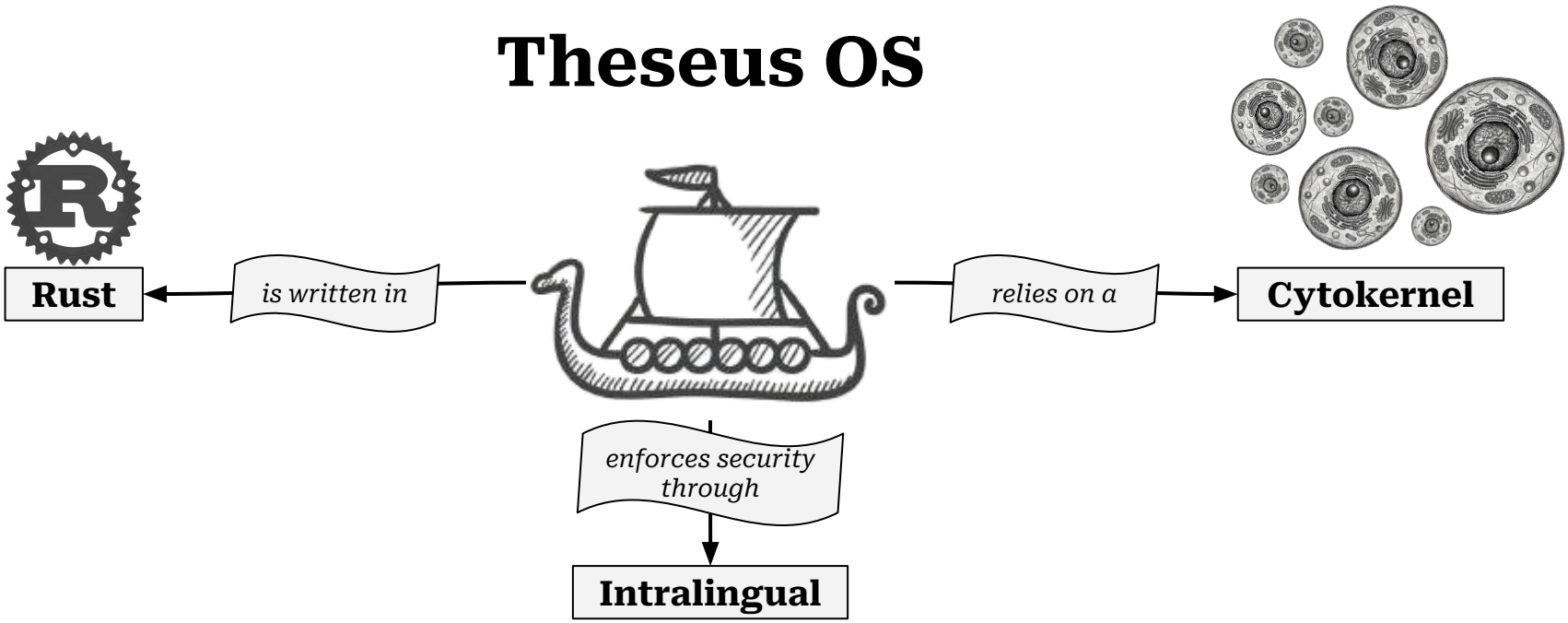
Why is this powerful?



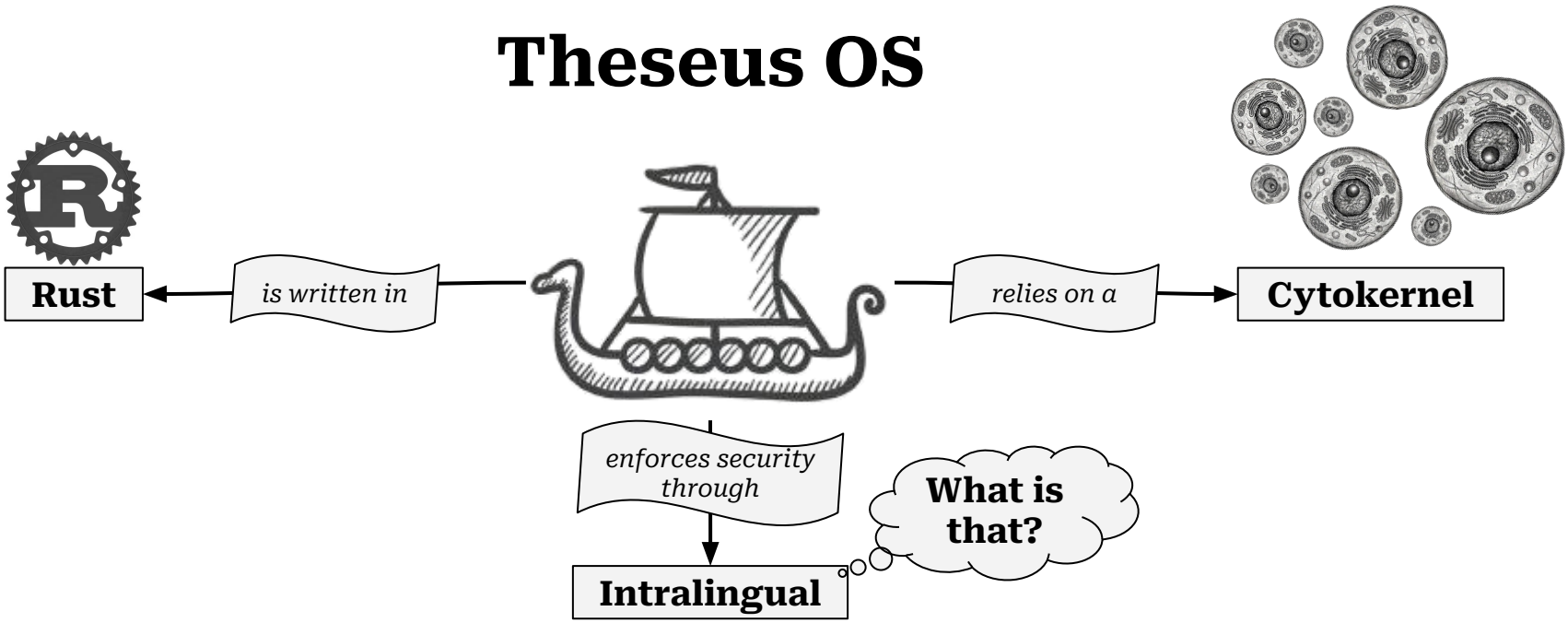
A **cell** is a *runtime-observable, independently-loadable module* that enables *fault isolation*, *live evolution*, *zero-copy state transfer*, and *automatic resource cleanup* without reboot.

- An **ultra-minimalist kernel** tasked solely with *state management* and *component loading*.
- An OS broken down into *tiny*, *independent logical units* called Cells (= Rust crates).

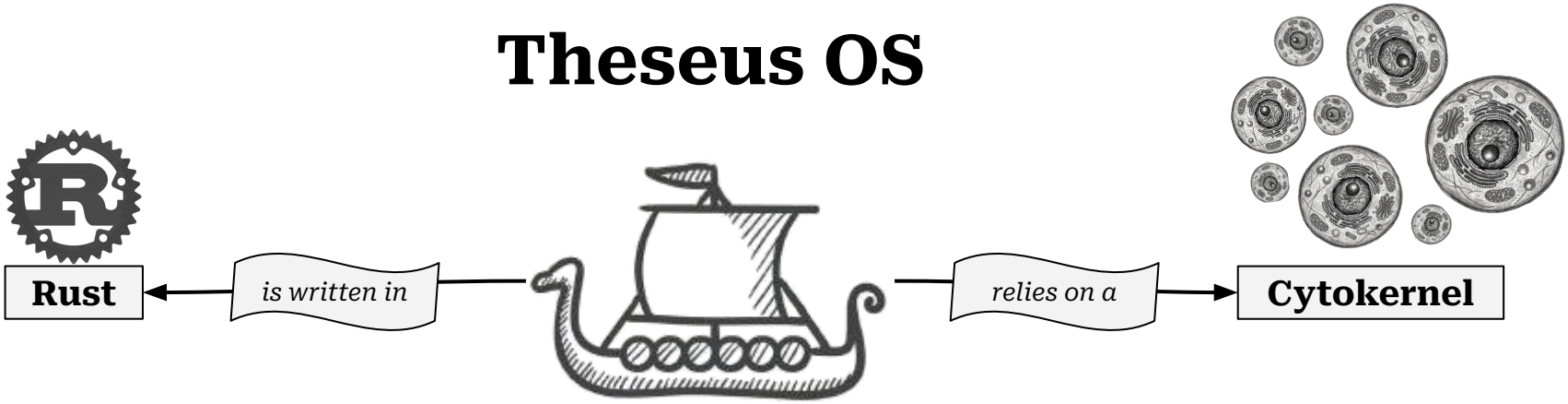
Theseus OS


Rust
is written in
relies on a
Cytokernel
enforces security through
Intralingual

Theseus OS


Rust
is written in
relies on a
Cytokernel
enforces security through
What is that?
Intralingual

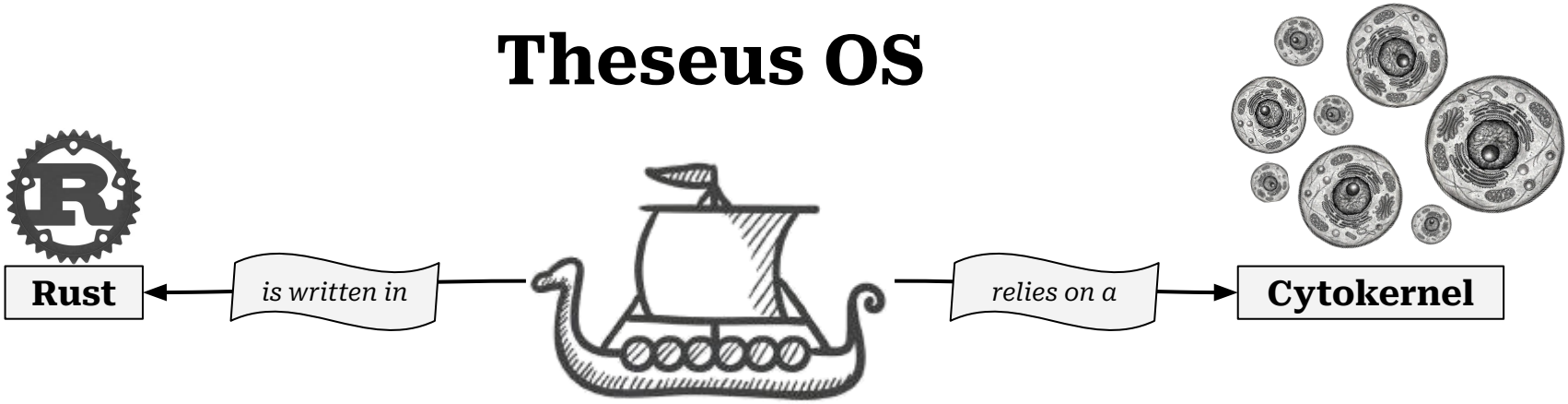
Theseus OS



- **Single Address Space** (SAS): Everything runs in the same memory space.

- **Single Privilege Level** (SPL): No hardware separation between user and kernel, nor slow context switches.

Theseus OS

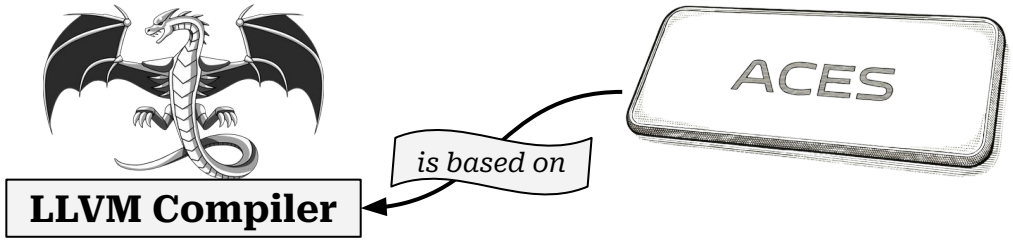


- **Single Address Space** (SAS): Everything runs in the same memory space.
- **Single Privilege Level** (SPL): No hardware separation between user and kernel, nor slow context switches.

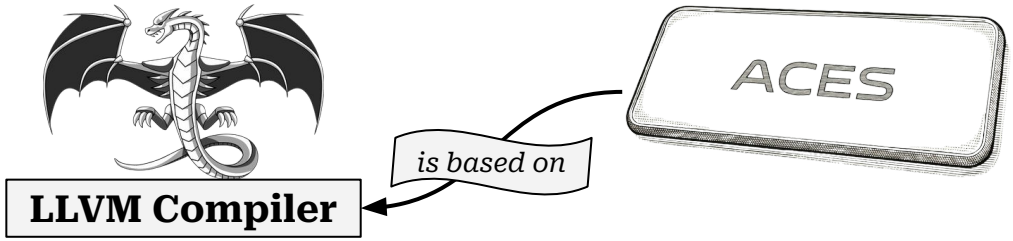
- **Isolation** is no longer enforced by hardware (*MPU/MMU/security rings*), but by the compiler (via Rust's type system and memory management).

ACES (*Automatic Compartments for Embedded Systems*)

ACES (*Automatic Compartments for Embedded Systems*)

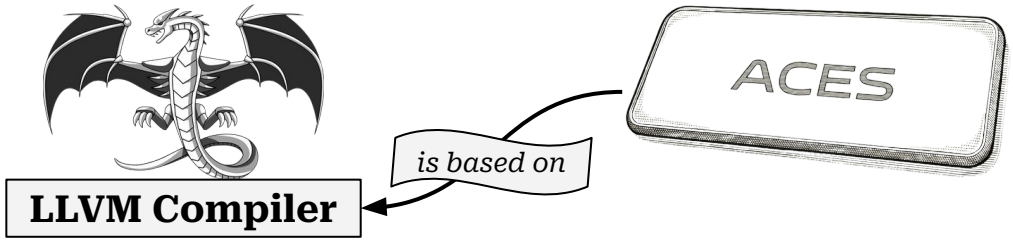


ACES (*Automatic Compartments for Embedded Systems*)



• ACES is a compiler extension that **automatically**[1] instruments binaries to isolate compartments using a MPU.

ACES (*Automatic Compartments for Embedded Systems*)



• ACES is a compiler extension that **automatically**[1] instruments binaries to isolate compartments using a MPU.

[1] ⇒ It is transparent to the developer, they just need to provide their policy. They don't need to modify a single line of code!

ACES (*Automatic Compartments for Embedded Systems*)

• ACES is a compiler extension that **automatically**[1] instruments binaries to isolate compartments using a MPU.

[1] ⇒ It is transparent to the developer, they just need to provide their policy. They don't need to modify a single line of code!

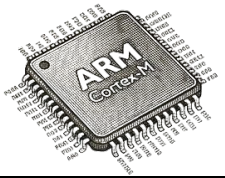
ACES (*Automatic Compartments for Embedded Systems*)

LLVM Compiler

is based on

ACES

targets



Bare-Metal Systems

- ACES is a compiler extension that **automatically**[1] instruments binaries to isolate compartments using a MPU.

[1] ⇒ It is transparent to the developer, they just need to provide their policy. They don't need to modify a single line of code!

- No OS support
- C language only
- Exclusive to the ARMv7-M architecture

ACES (*Automatic Compartments for Embedded Systems*)

LLVM Compiler

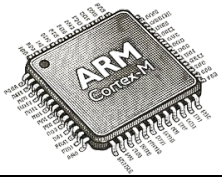
is based on

ACES

enforces

Isolation

targets



Bare-Metal Systems

- ACES is a compiler extension that **automatically**[1] instruments binaries to isolate compartments using a MPU.

[1] ⇒ It is transparent to the developer, they just need to provide their policy. They don't need to modify a single line of code!

- No OS support
- C language only
- Exclusive to the ARMv7-M architecture

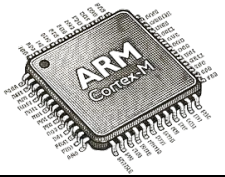
ACES (*Automatic Compartments for Embedded Systems*)

LLVM Compiler

is based on

ACES

targets



Bare-Metal Systems

enforces

Isolation

• ACES is a compiler extension that **automatically**[1] instruments binaries to isolate compartments using a MPU.

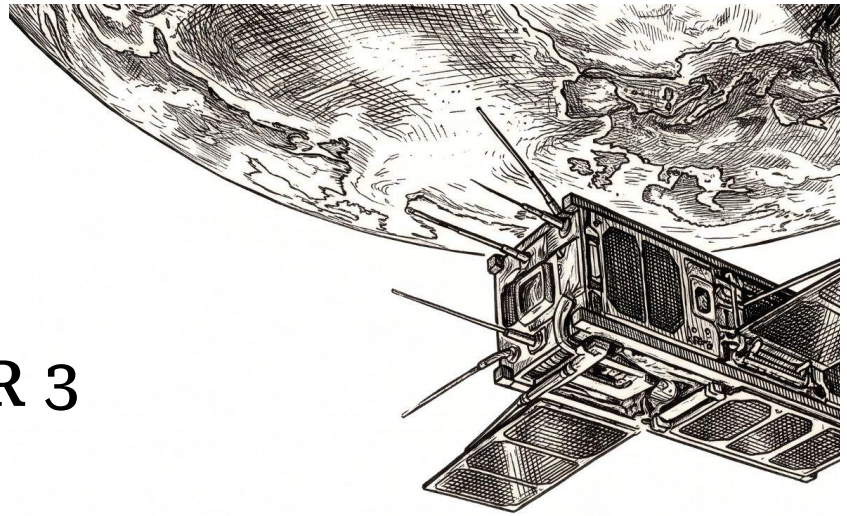
[1] ⇒ It is transparent to the developer, they just need to provide their policy. They don't need to modify a single line of code!

Code-centric rather than control-flow -centric isolation to mitigate ROP attacks.

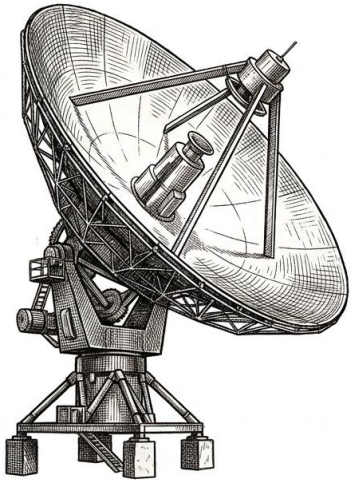
• No OS support

• C language only

• Exclusive to the ARMv7-M architecture



CHAPTER 3



Our Contribution

Unveiling xOS

Unveiling xOS

An OS that **adapts to your hardware**:

Whether it's your isolation setup (None, MPU, MMU, CHERI capabilities, etc.)

or your storage medium (running applications directly from flash or RAM).

No need to create a new OS!

Unveiling xOS

An OS that **adapts to your hardware**:

Whether it's your isolation setup (None, MPU, MMU, CHERI capabilities, etc.)

or your storage medium (running applications directly from flash or RAM).

No need to create a new OS!

Zero developer overhead:

Developers don't have to handle security. They simply choose:

mark their Rust app as safe to run in the kernel,

or indicate it's unsafe (C or Rust) to be compartmentalized via ACES.

Unveiling xOS

An OS that **adapts to your hardware**:

Whether it's your isolation setup (None, MPU, MMU, CHERI capabilities, etc.)

or your storage medium (running applications directly from flash or RAM).

No need to create a new OS!

Zero developer overhead:

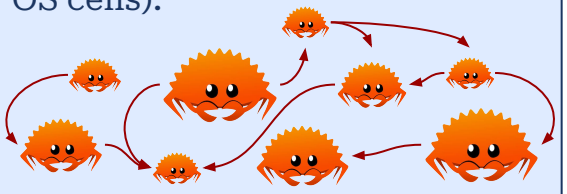
Developers don't have to handle security. They simply choose:

mark their Rust app as safe to run in the kernel,

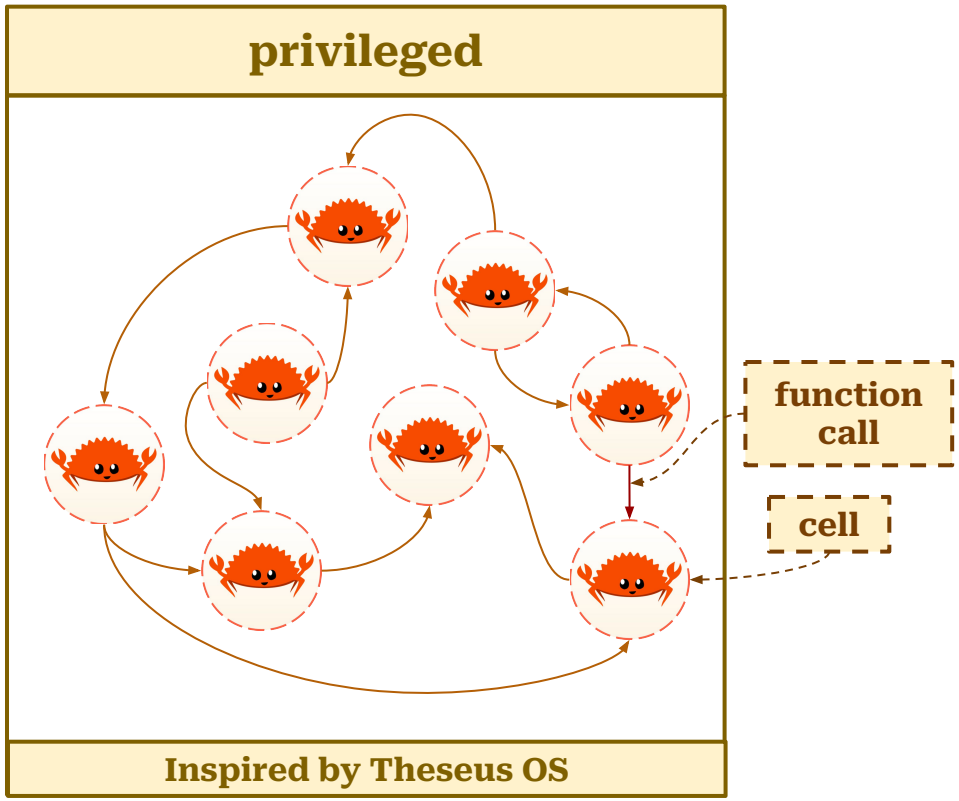
or indicate it's unsafe (C or Rust) to be compartmentalized via ACES.

Modular live-updates & fault recovery:

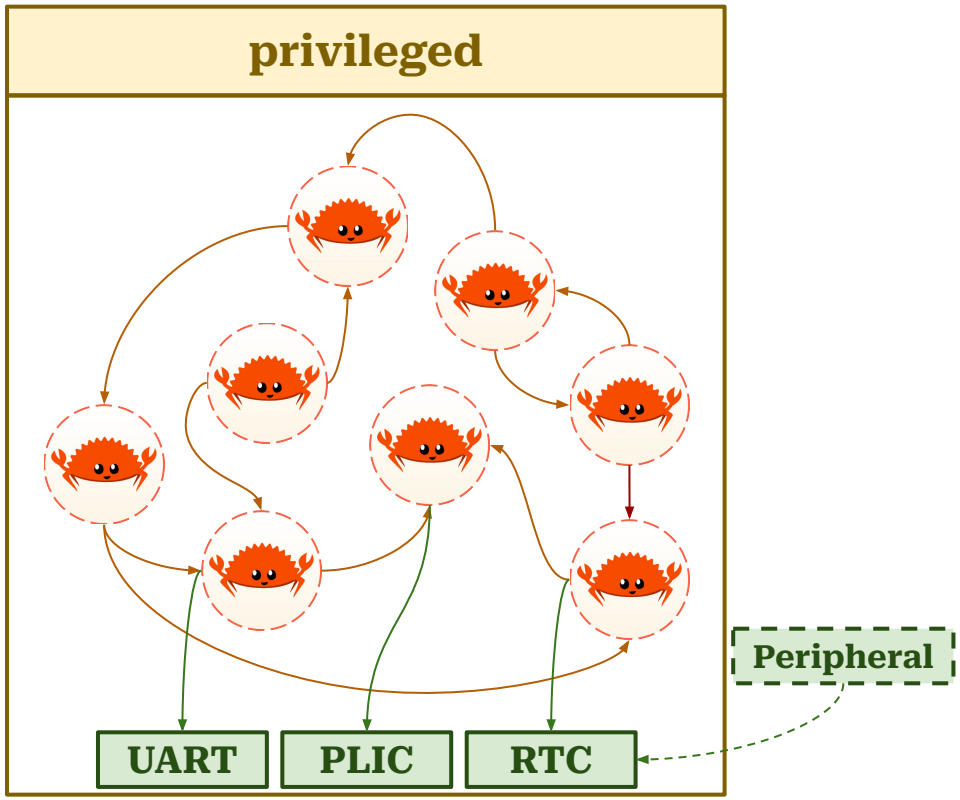
Update or restart individual OS modules on the fly without rebooting (inspired by Theseus OS cells).



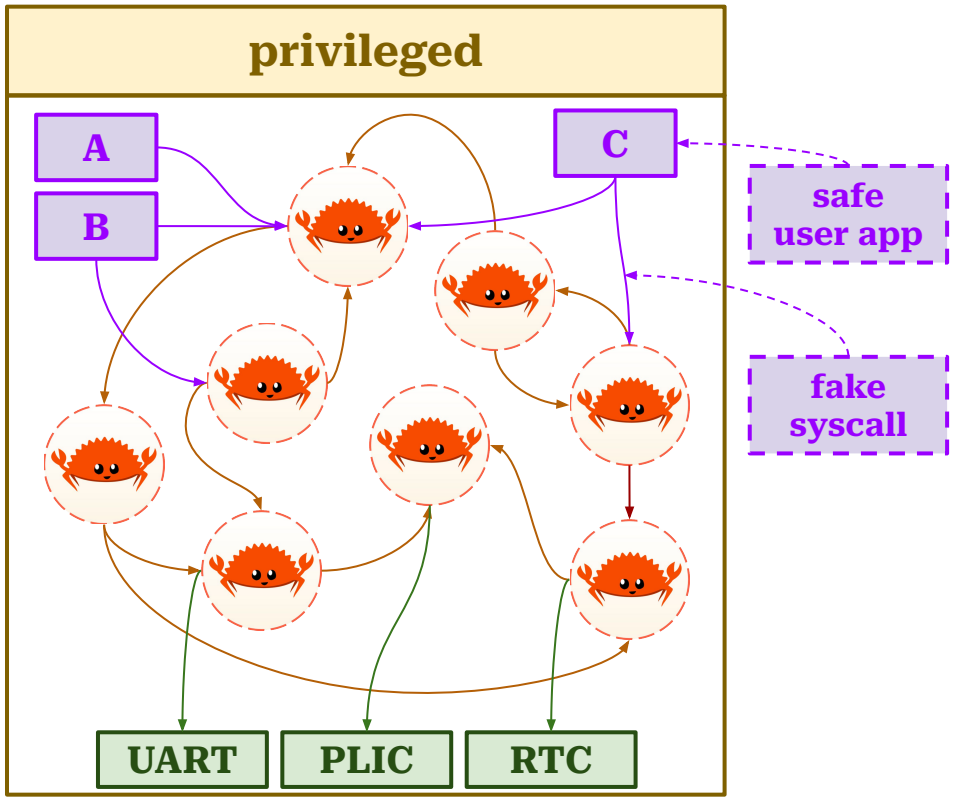
High-Level Overview



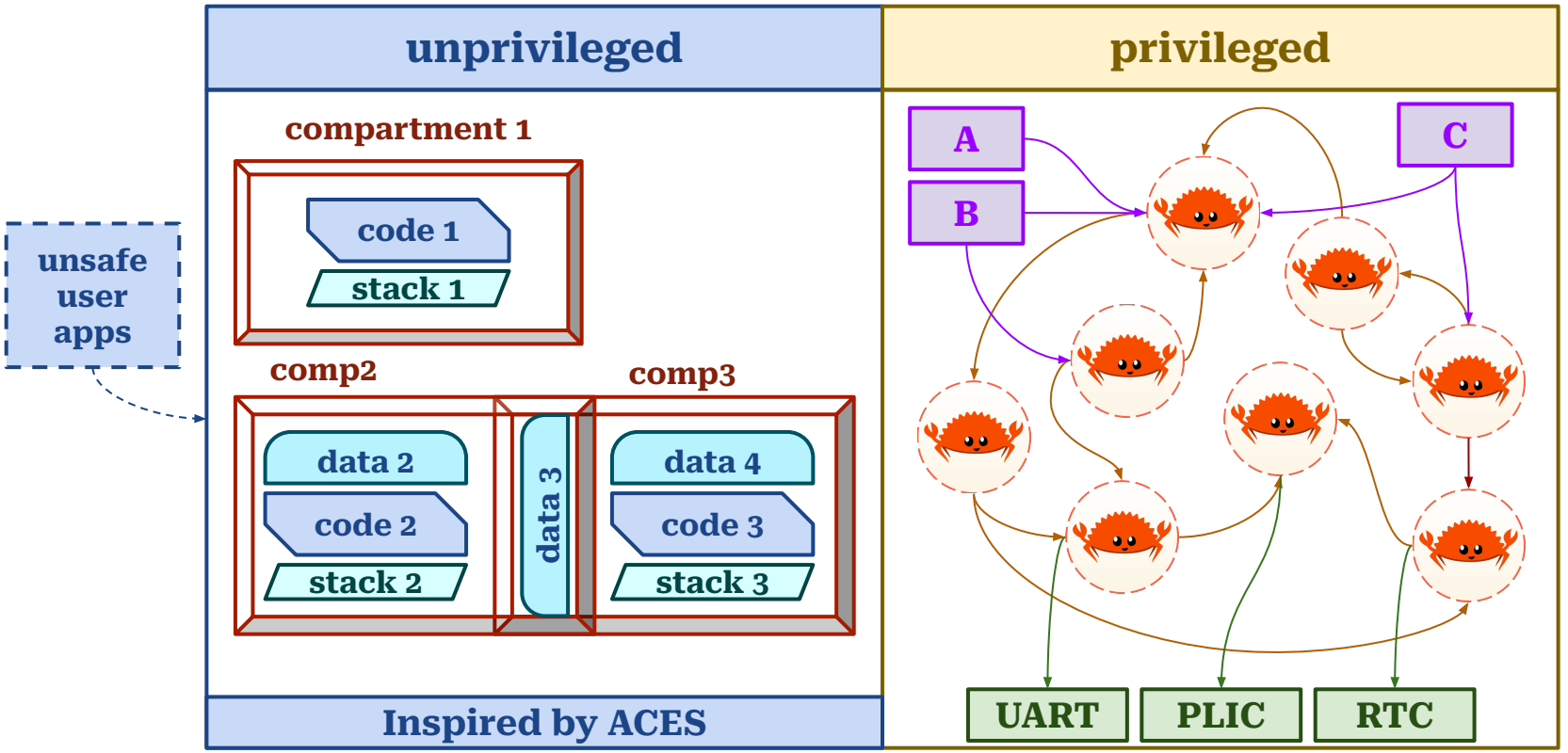
High-Level Overview



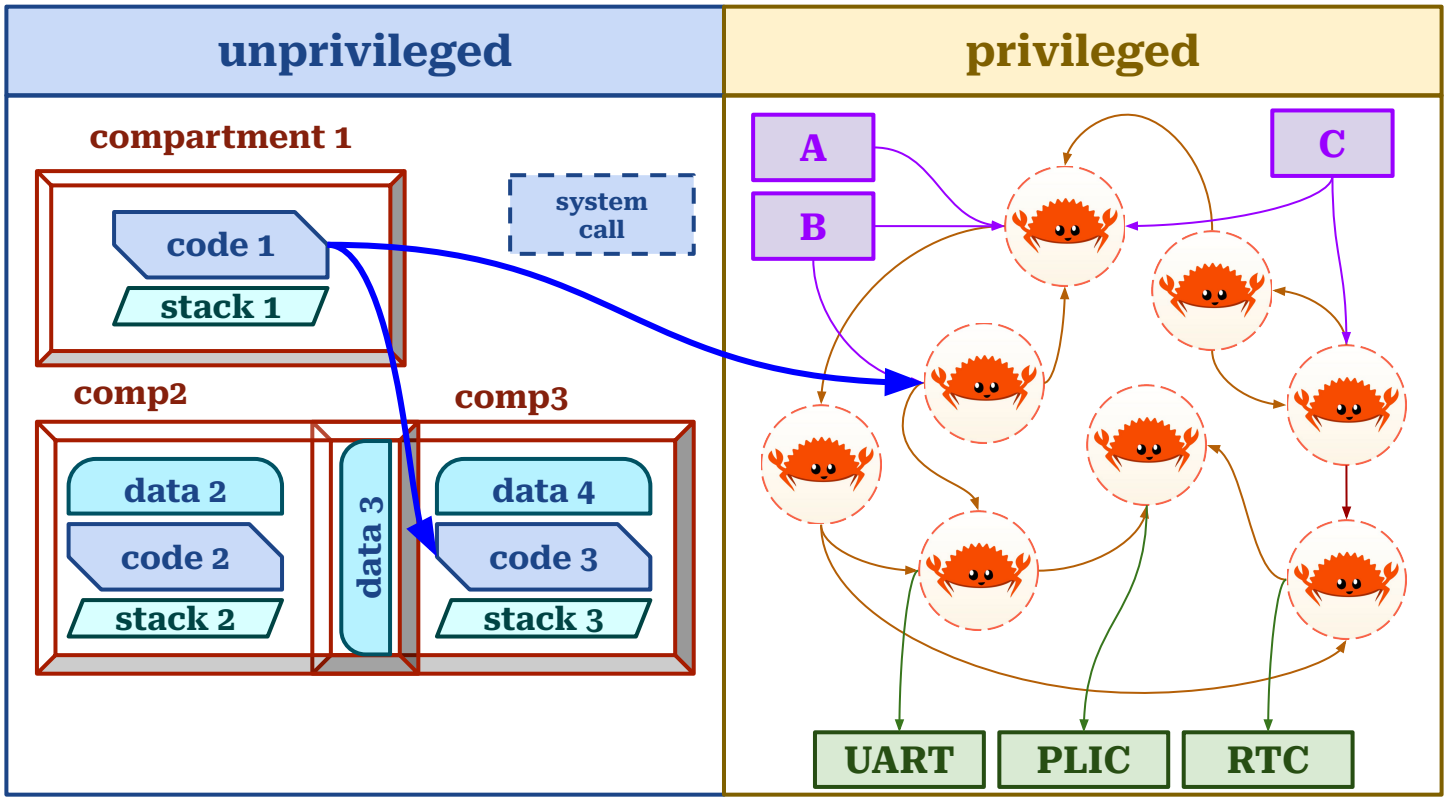
High-Level Overview



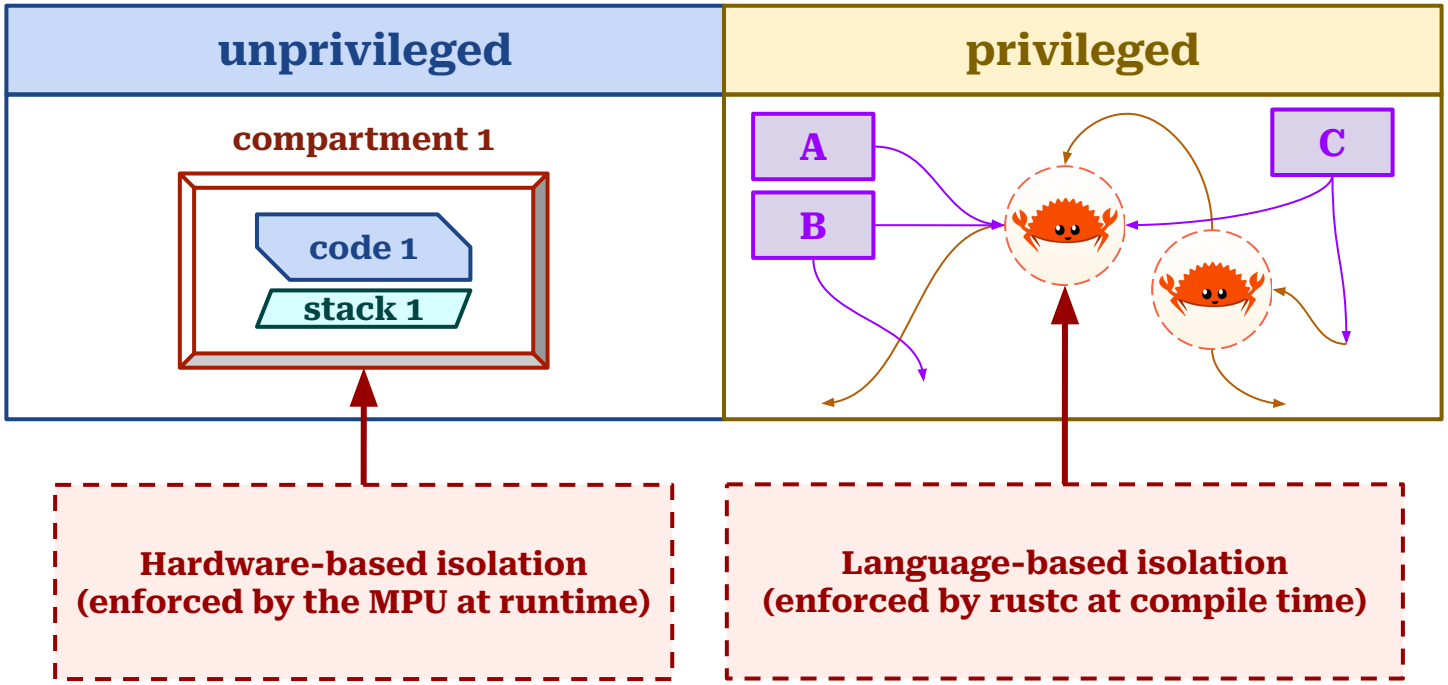
High-Level Overview



High-Level Overview

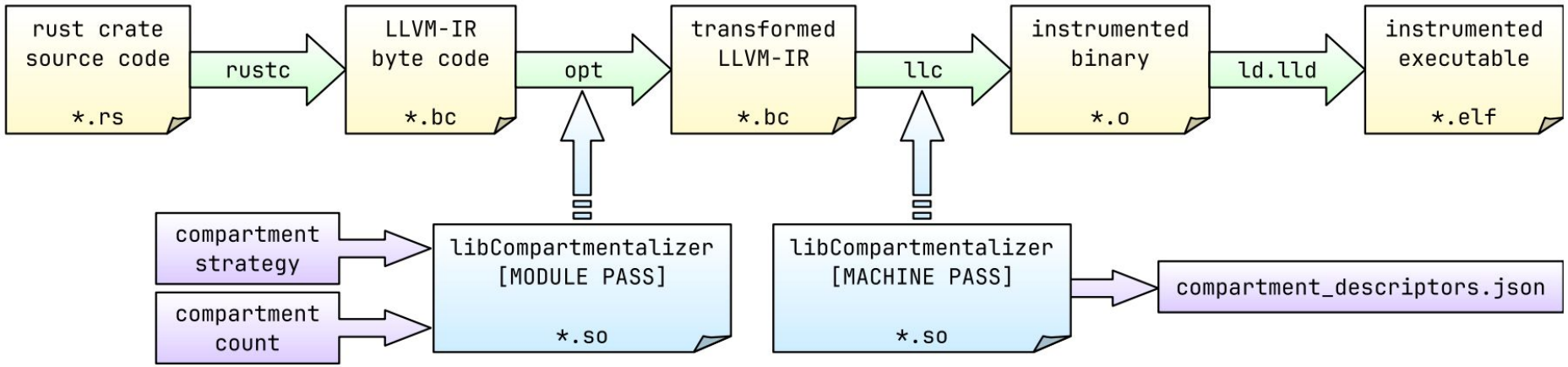


High-Level Overview



Compartmentalizer

How to protect unsafe applications using the ACES mechanism?



1. Compatible with all compilers that can produce LLVM-IR, such as **rustc** or **clang**.

2. Everything is **automatic**, the developer has nothing to do except choose their compartmentalization policy.

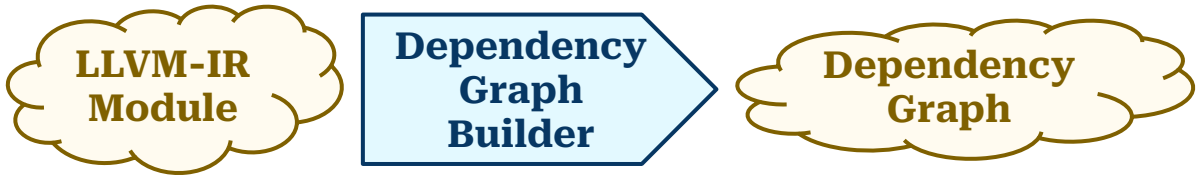
LLVM Module Pass — Step 1


```
define dso_local
i32 @foo(i32 %arg) {
entry:
  ; DIRECT CALL
  %1 = call i32 @bar(i32 %arg)
  ret i32 %1
}

define dso_local
i32 @bar(i32 %arg) {
entry:
  ; INDIRECT CALL
  %fp = load i32 (i32)*, i32 (i32)** @func_ptr
  %result = call i32 %fp(i32 %arg)
  ret i32 %result
}
```

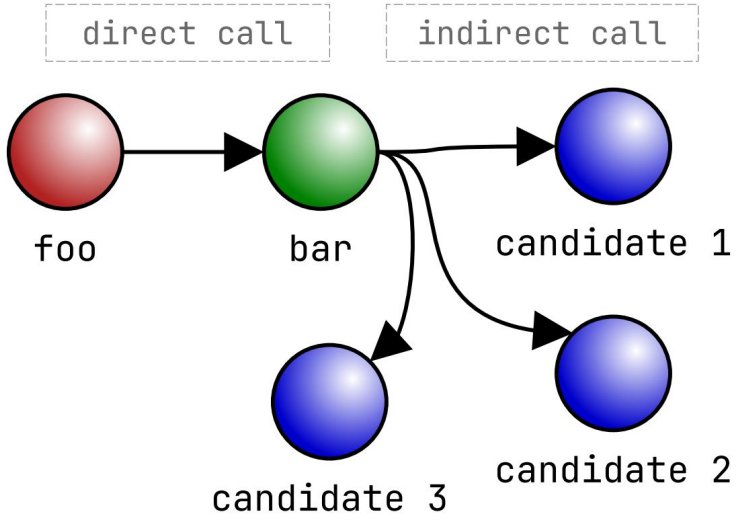
LLVM Module Pass — Step 1


```
define dso_local
i32 @foo(i32 %arg) {
entry:
  ; DIRECT CALL
  %1 = call i32 @bar(i32 %arg)
  ret i32 %1
}

define dso_local
i32 @bar(i32 %arg) {
entry:
  ; INDIRECT CALL
  %fp = load i32 (i32)*, i32 (i32)** @func_ptr
  %result = call i32 %fp(i32 %arg)
  ret i32 %result
}
```

LLVM Module Pass — Step 1

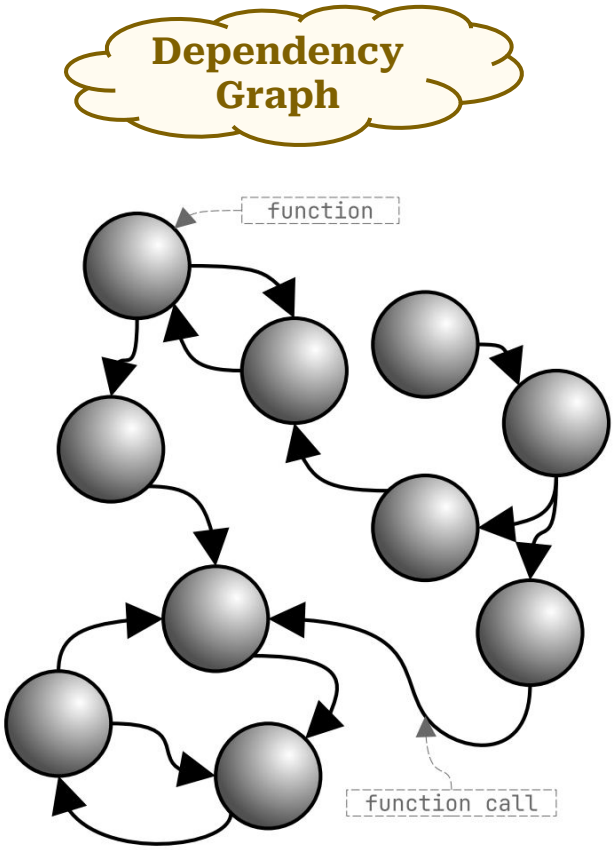



```
define dso_local
i32 @foo(i32 %arg) {
entry:
  ; DIRECT CALL
  %1 = call i32 @bar(i32 %arg)
  ret i32 %1
}

define dso_local
i32 @bar(i32 %arg) {
entry:
  ; INDIRECT CALL
  %fp = load i32 (i32)*, i32 (i32)** @func_ptr
  %result = call i32 %fp(i32 %arg)
  ret i32 %result
}
```

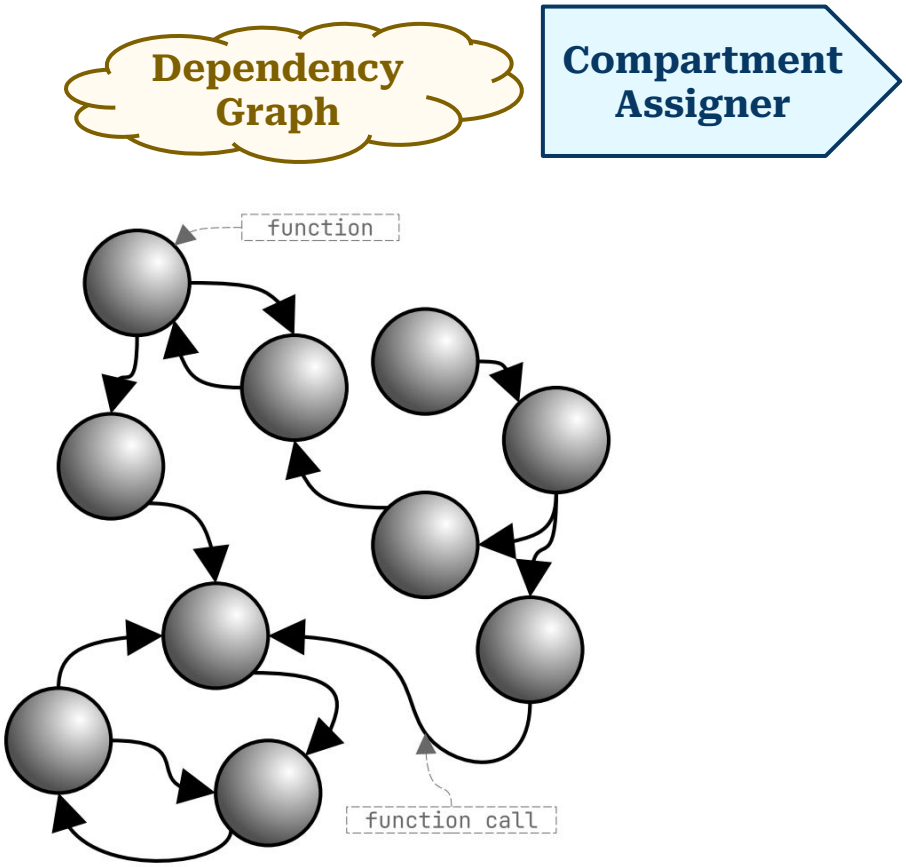


Note: The candidates share the same signature: i32 → i32.

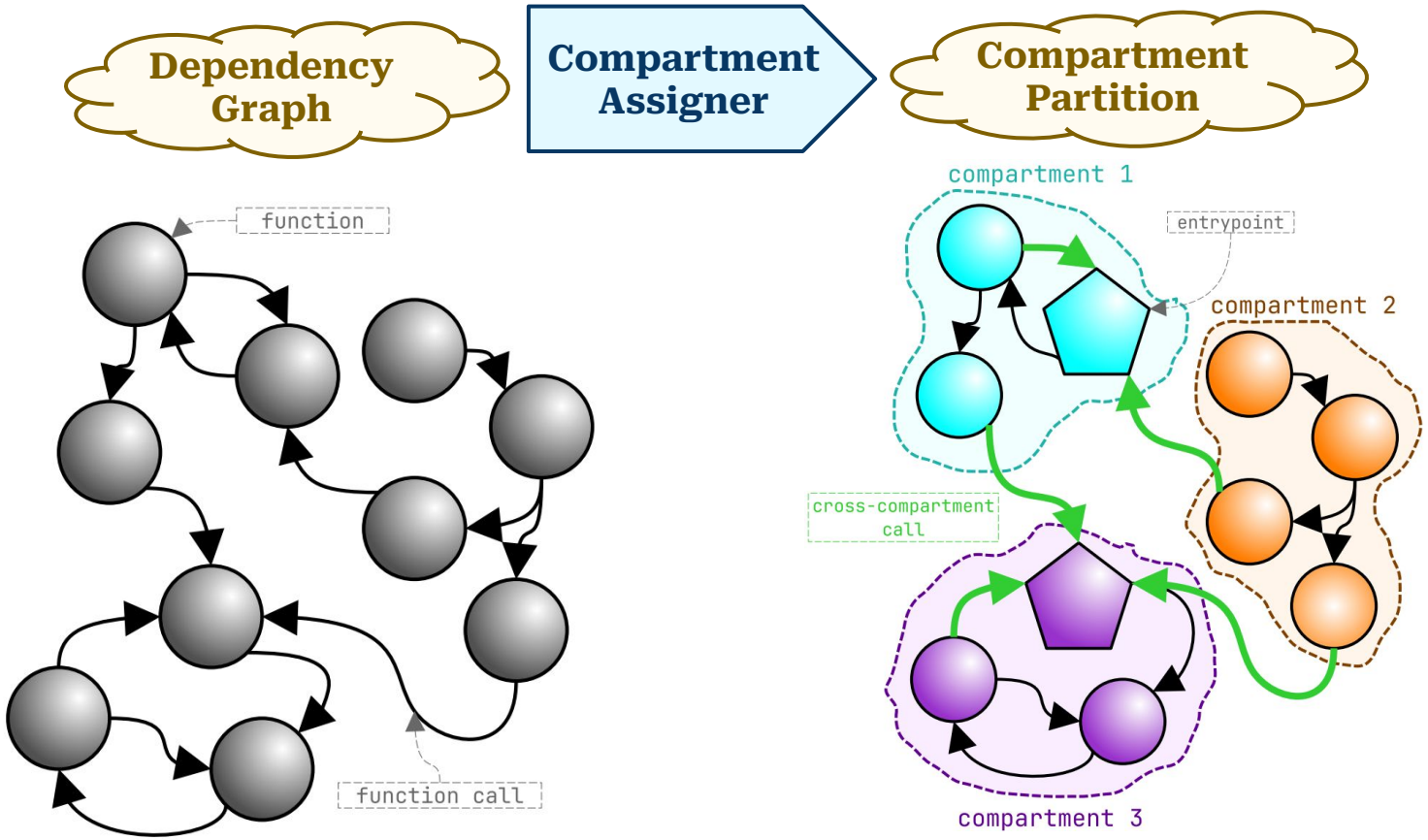
LLVM Module Pass — Step 2



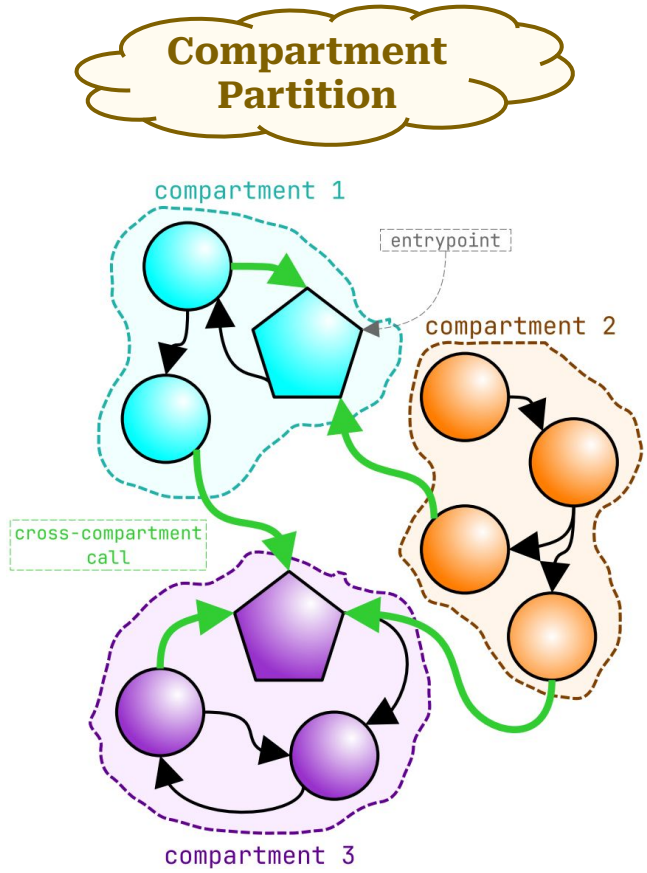
LLVM Module Pass — Step 2



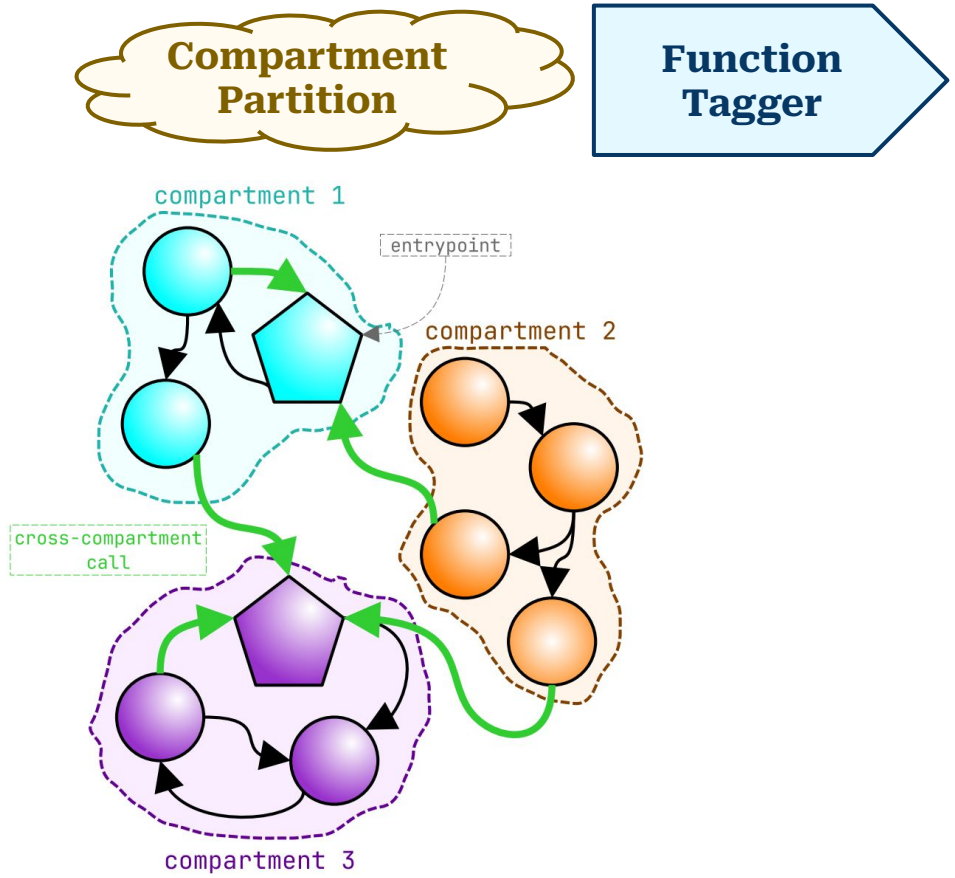
LLVM Module Pass — Step 2



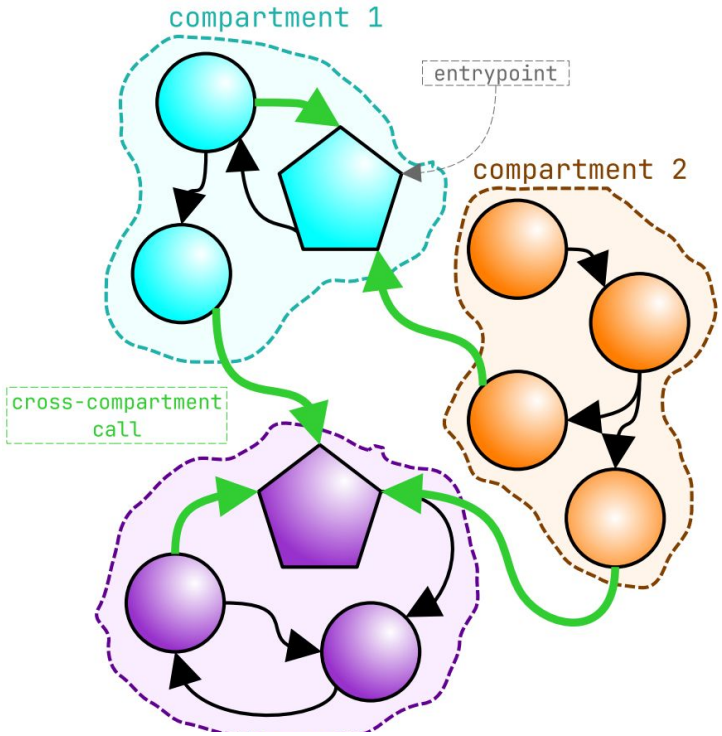
LLVM Module Pass — Step 3



LLVM Module Pass — Step 3



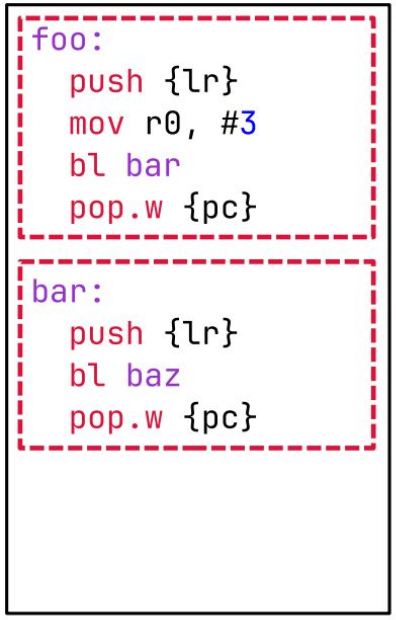
LLVM Module Pass — Step 3



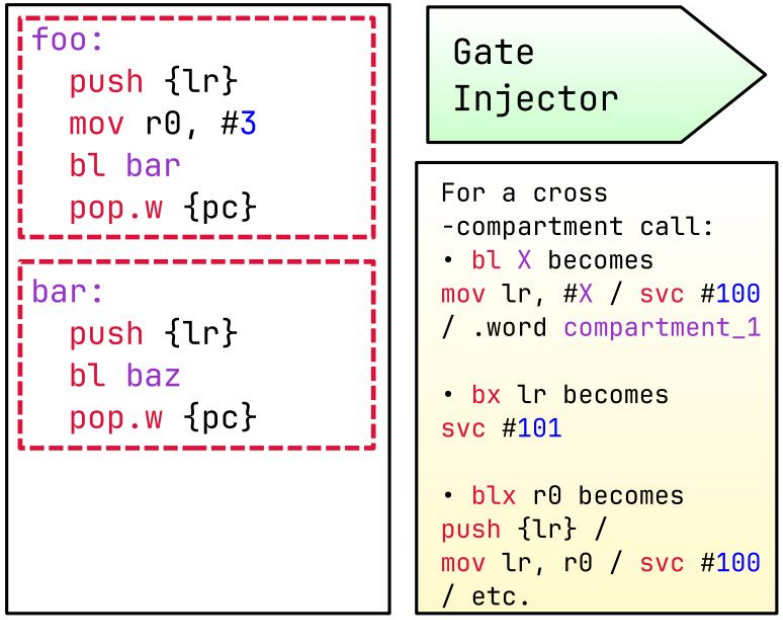

```
define dso_local
i32 @caller(i32 %arg) section
".text.comp.C1" #1 {
entry:
 %1 = call i32 @callee(i32 %arg)
 ret i32 %1
}

define dso_local
i32 @calle(i32 %arg) section
".text.comp.C2" #2 {
entry:
 ret i32 %arg
}
```

```
attributes #1 = {
 "compartment-id"="C1"
 "compartment-entry"="false"
 optsize
}

attributes #2 = {
 "compartment-id"="C2"
 "compartment-entry"="true"
 optsize
}
```

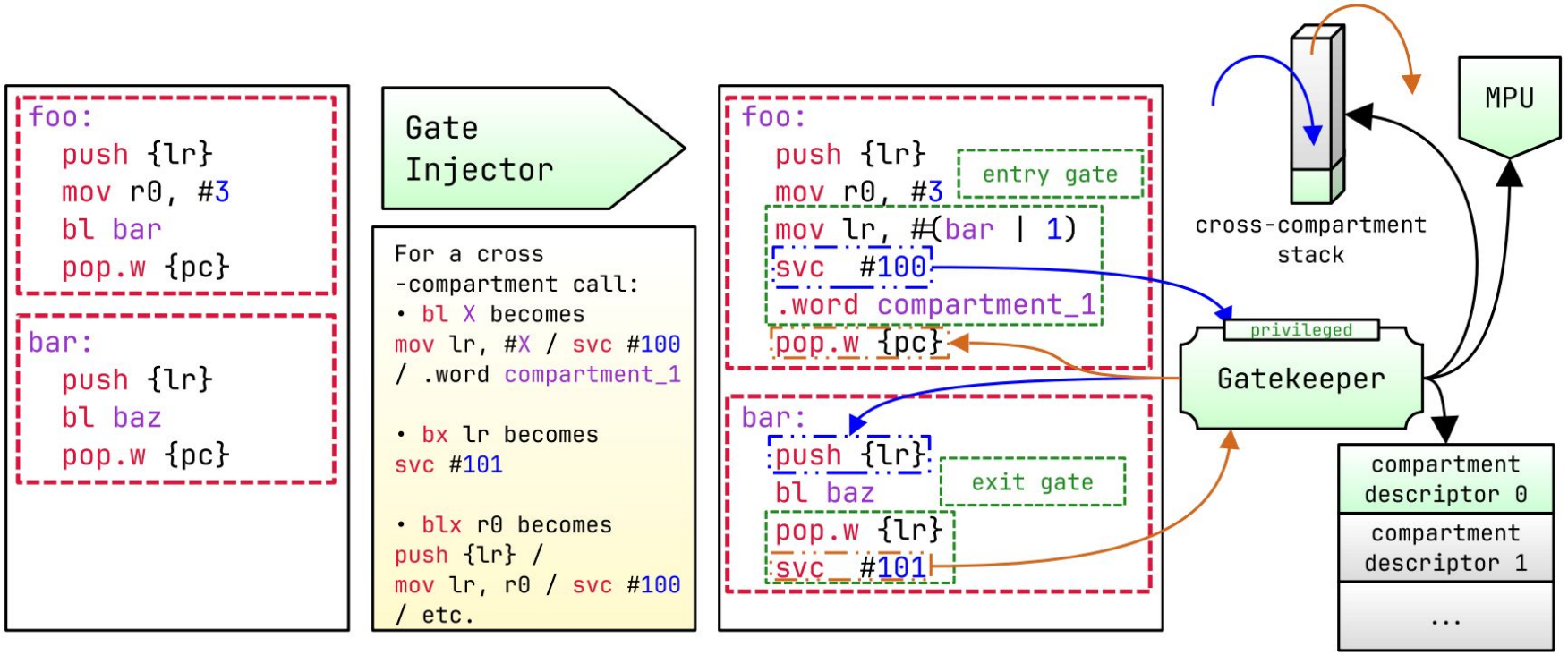
LLVM Machine Pass — Step 4



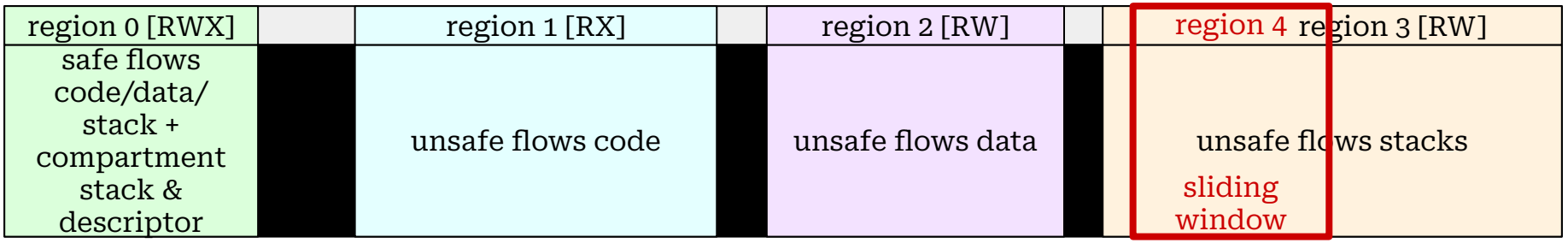
LLVM Machine Pass — Step 4



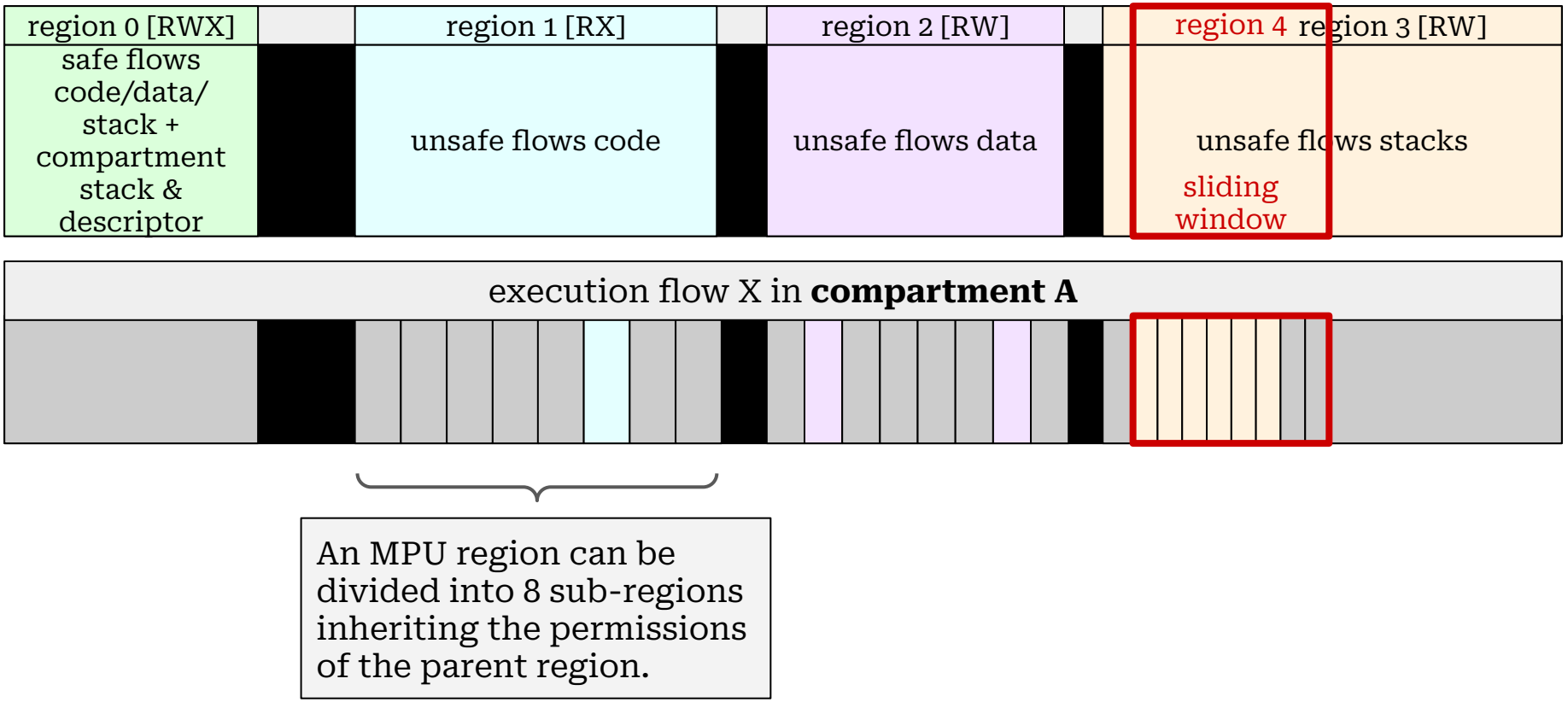
LLVM Machine Pass — Step 4



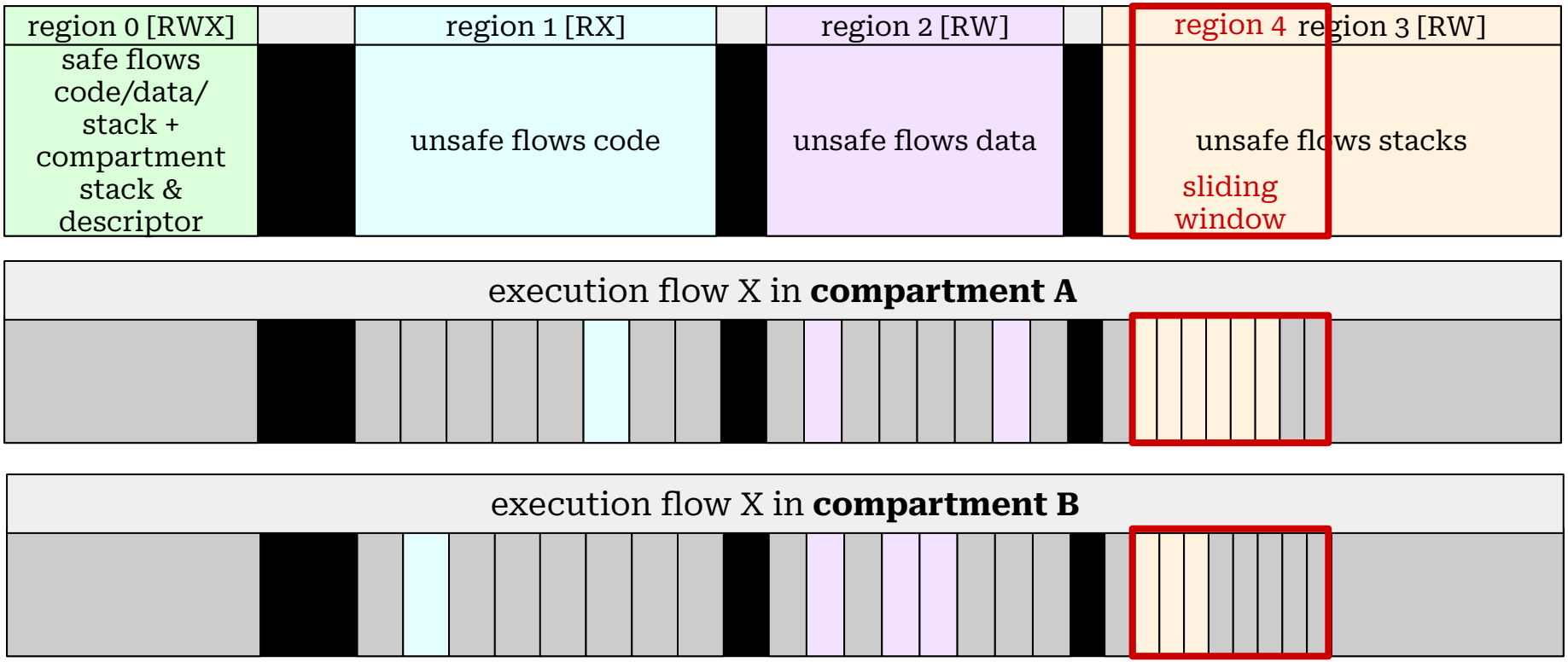
Memory Layout



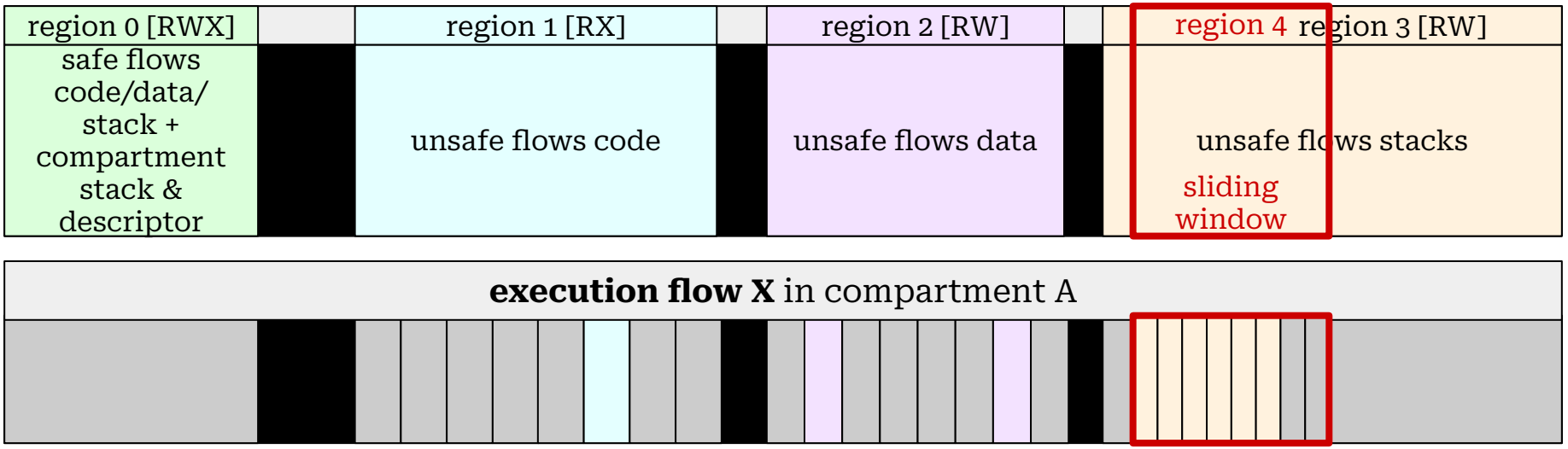
Memory Layout



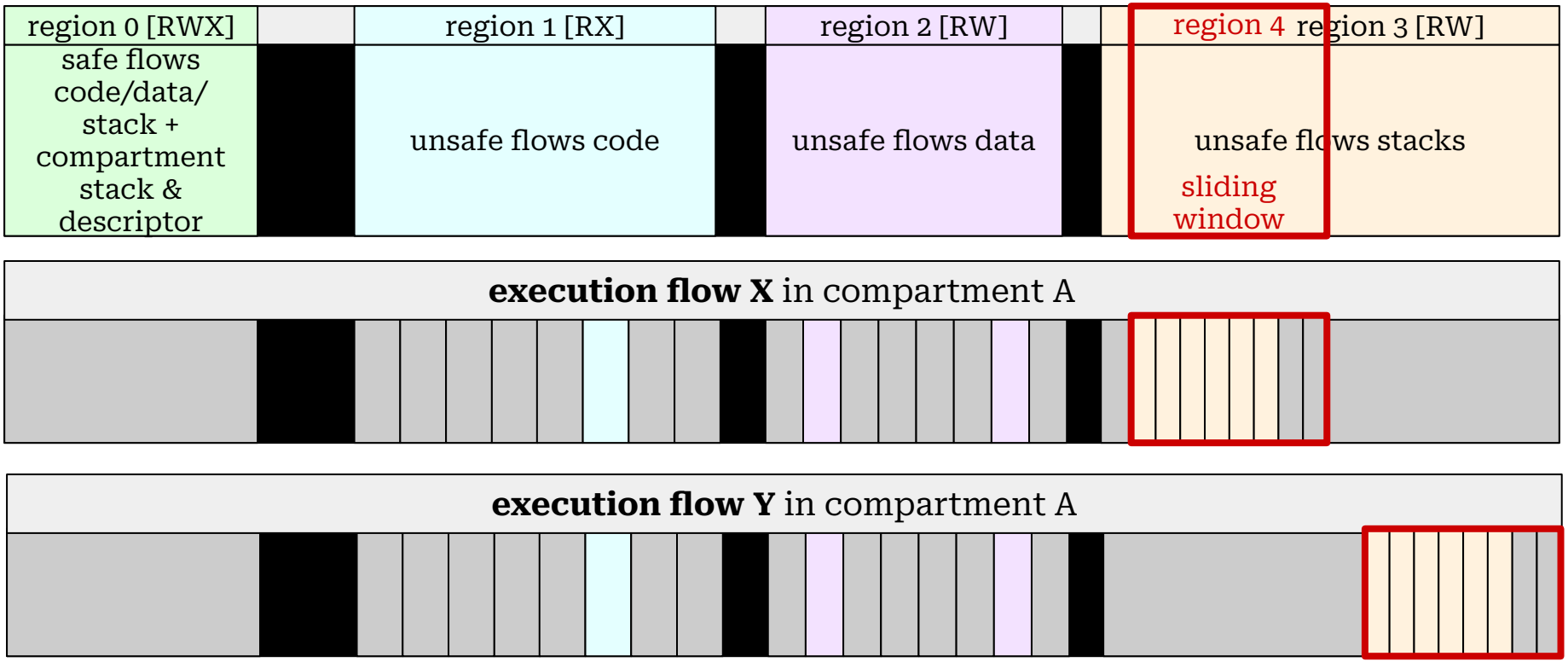
Memory Layout



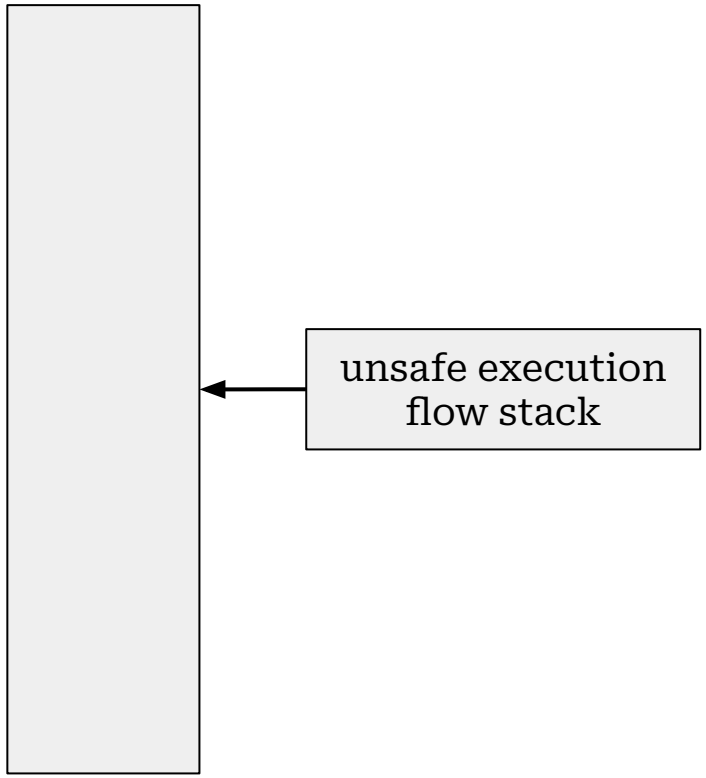
Memory Layout



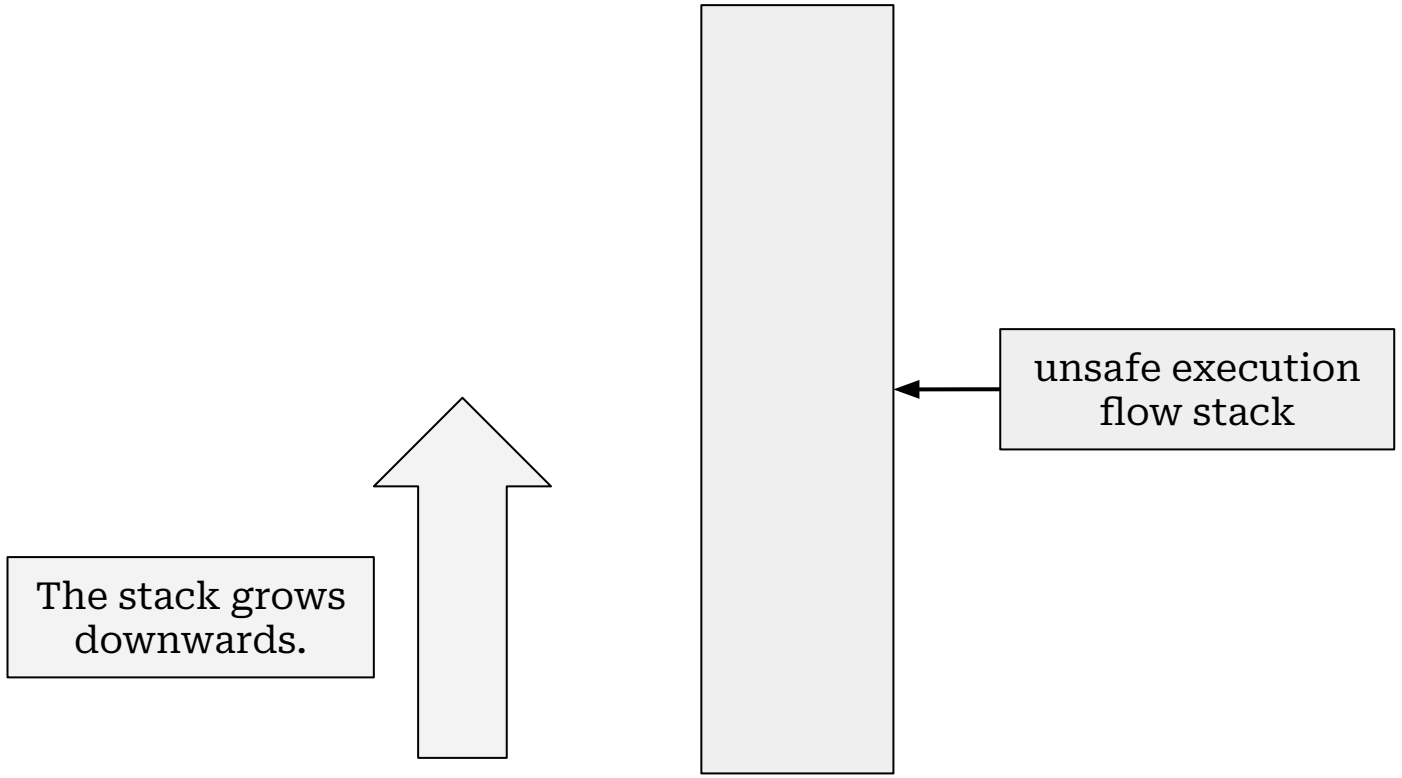
Memory Layout



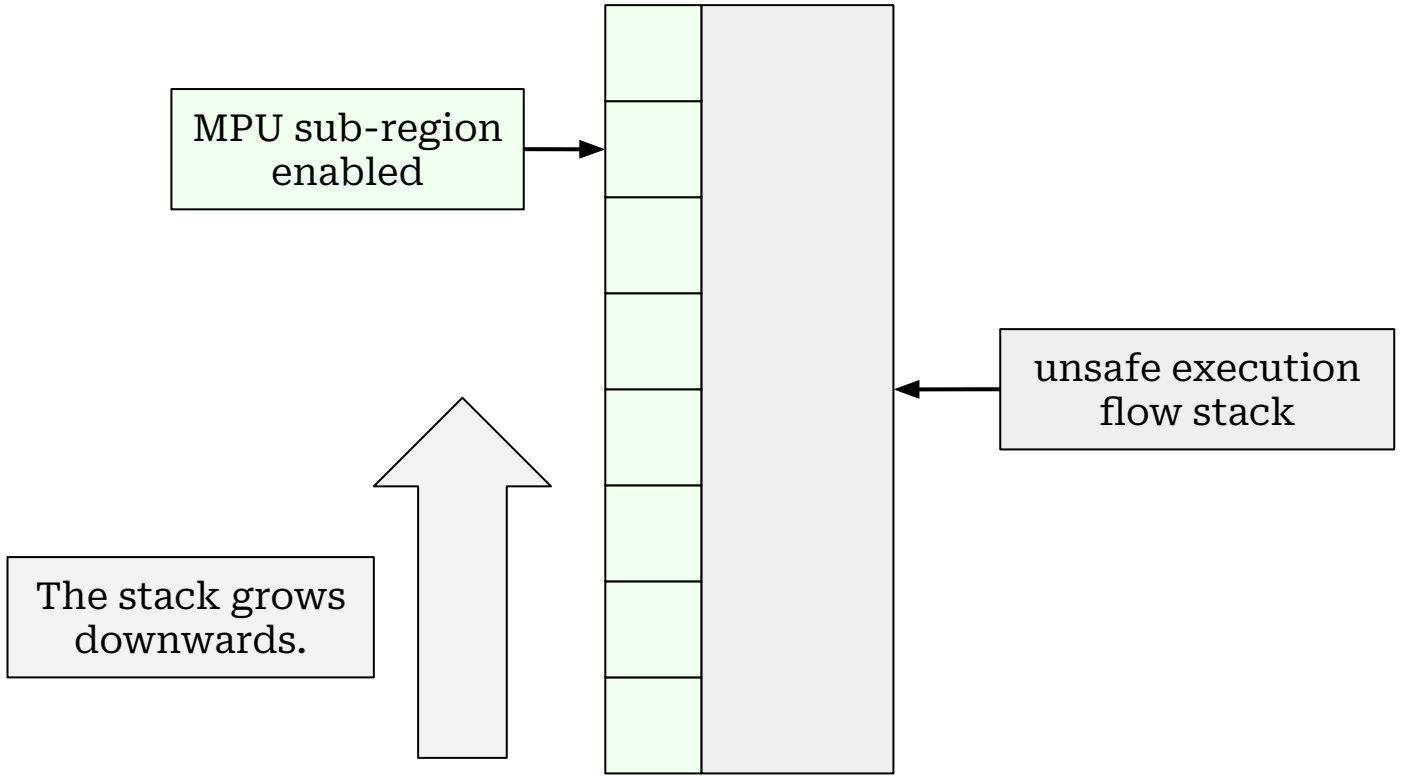
Stack Protection



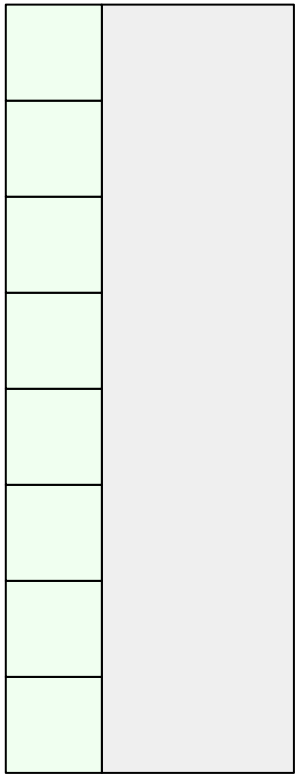
Stack Protection



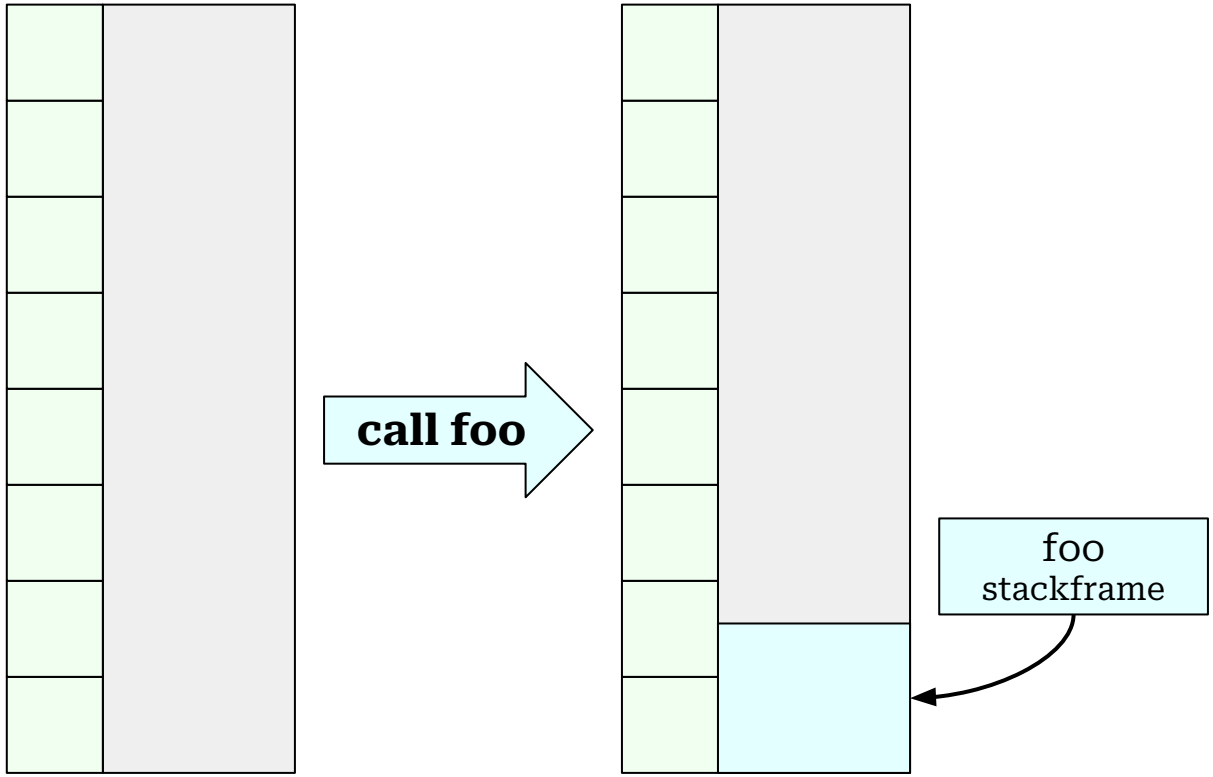
Stack Protection



Stack Protection

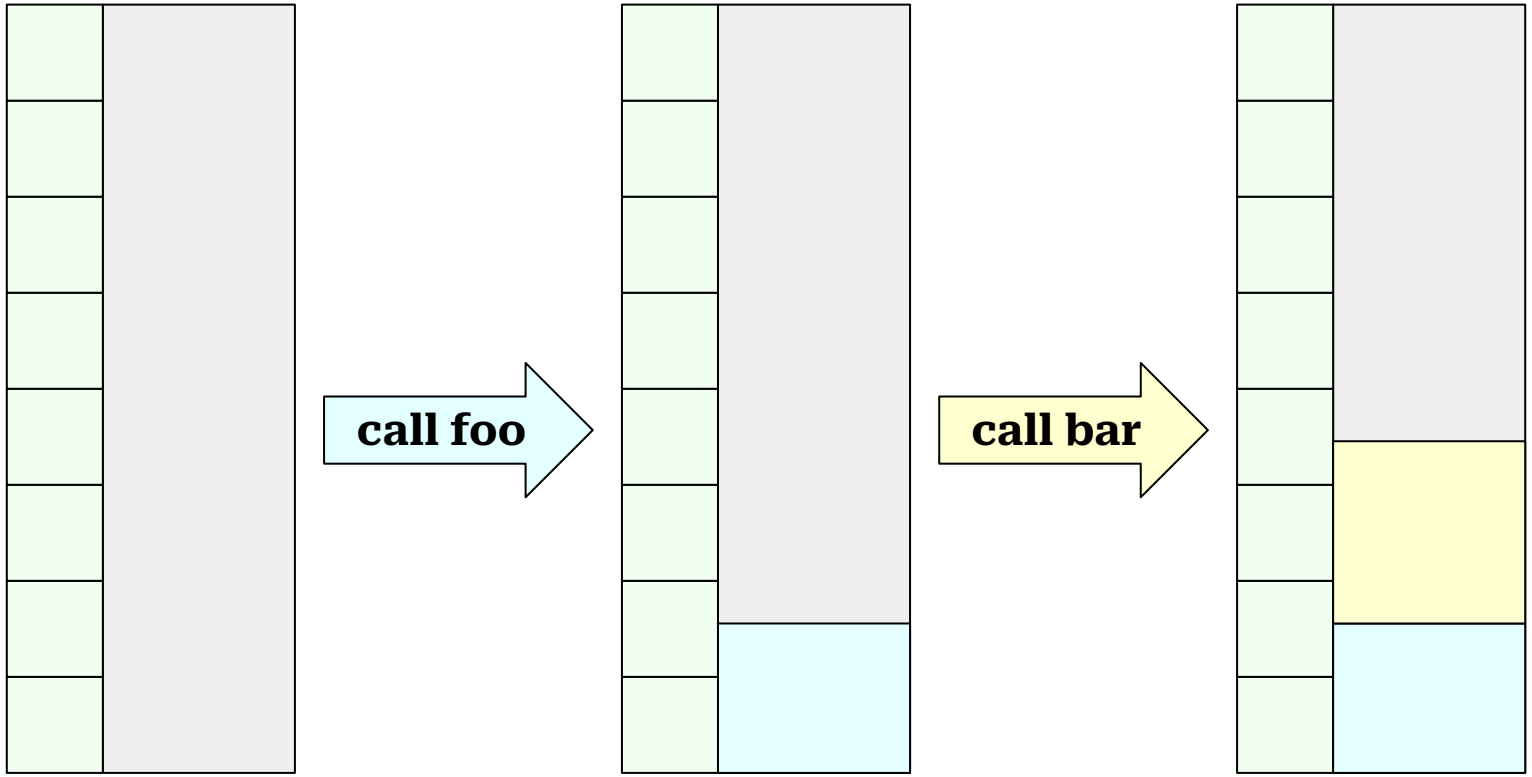


Stack Protection

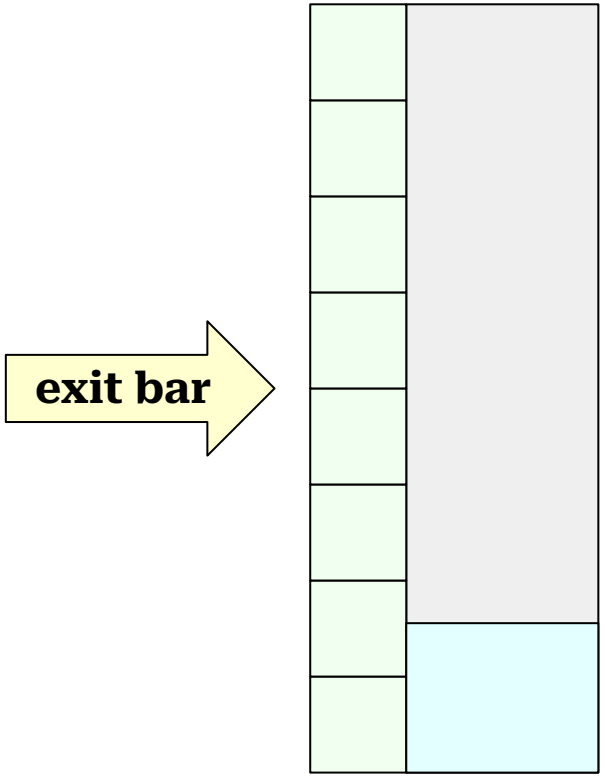


Stack Protection

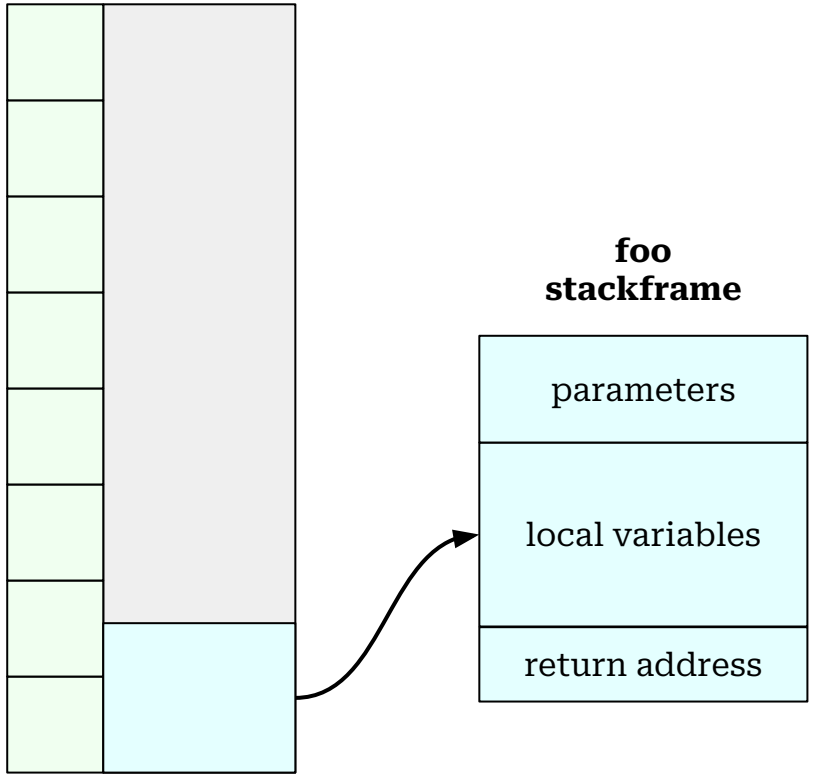
Note: foo and bar reside in the same compartment A.



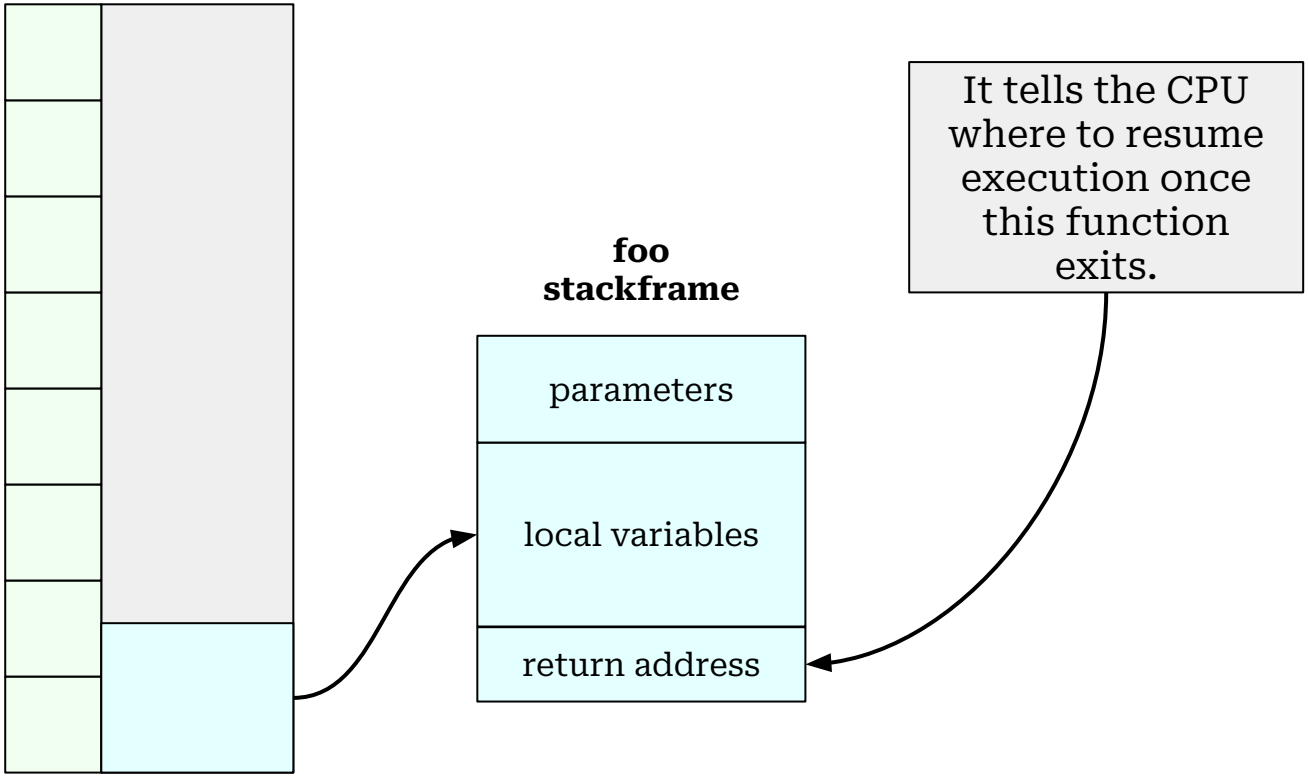
Stack Protection



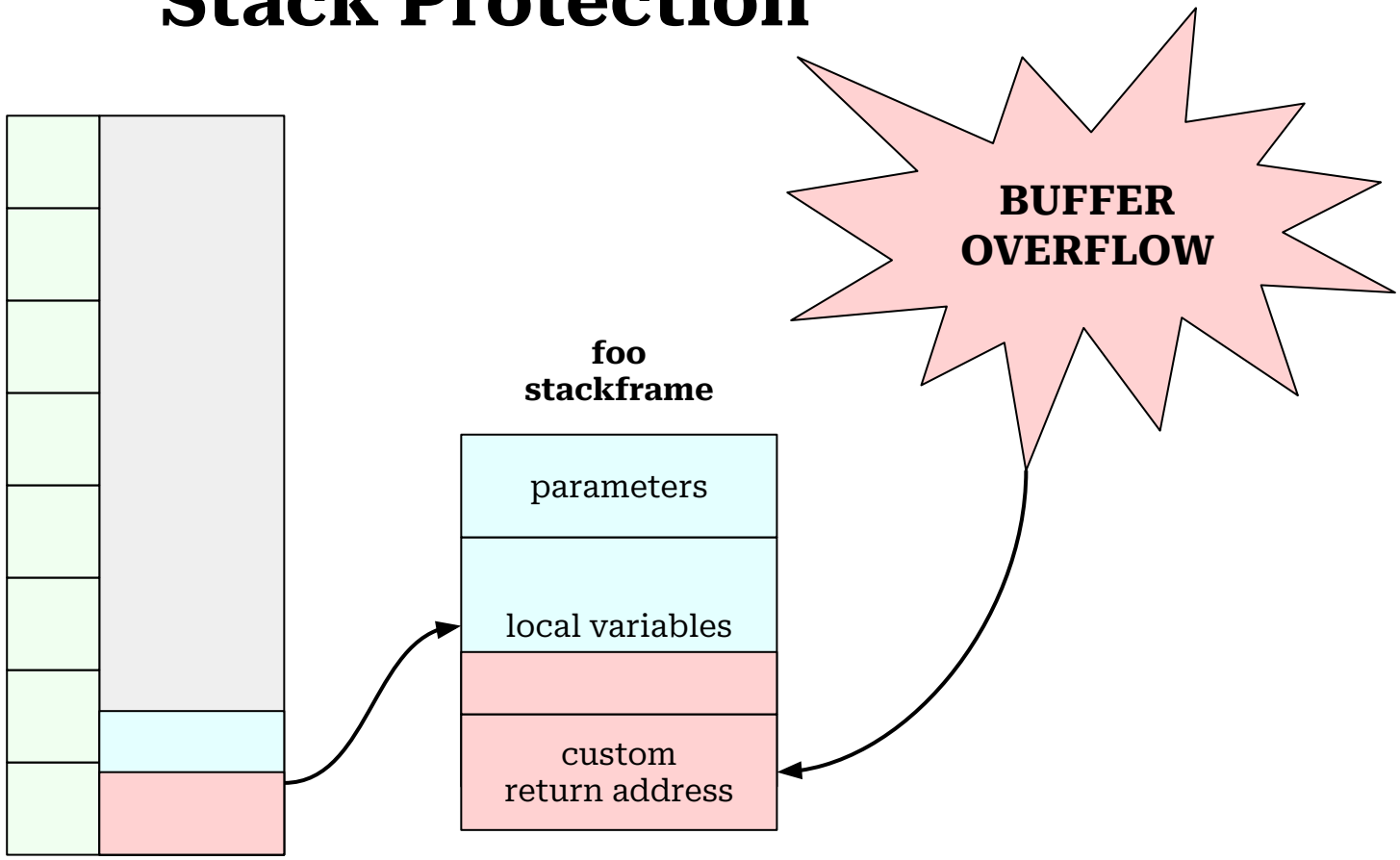
Stack Protection



Stack Protection

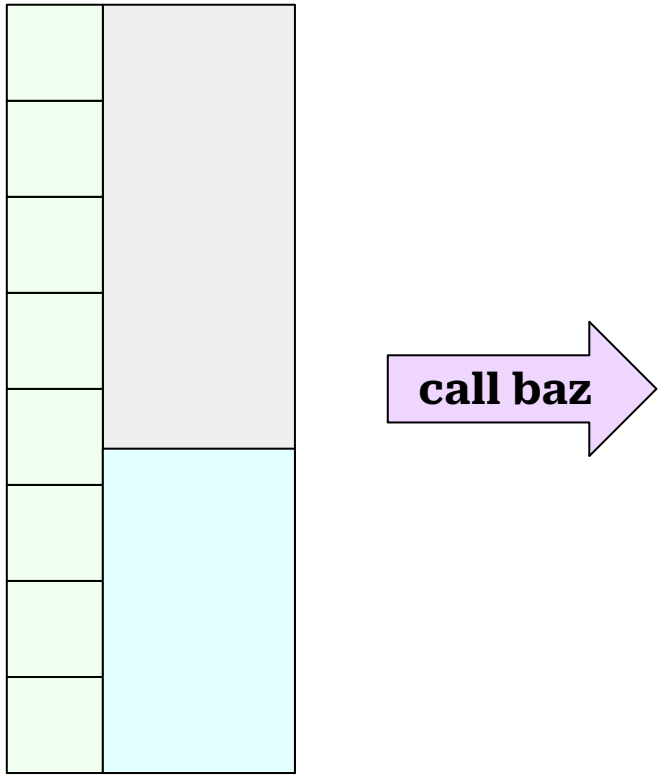


Stack Protection



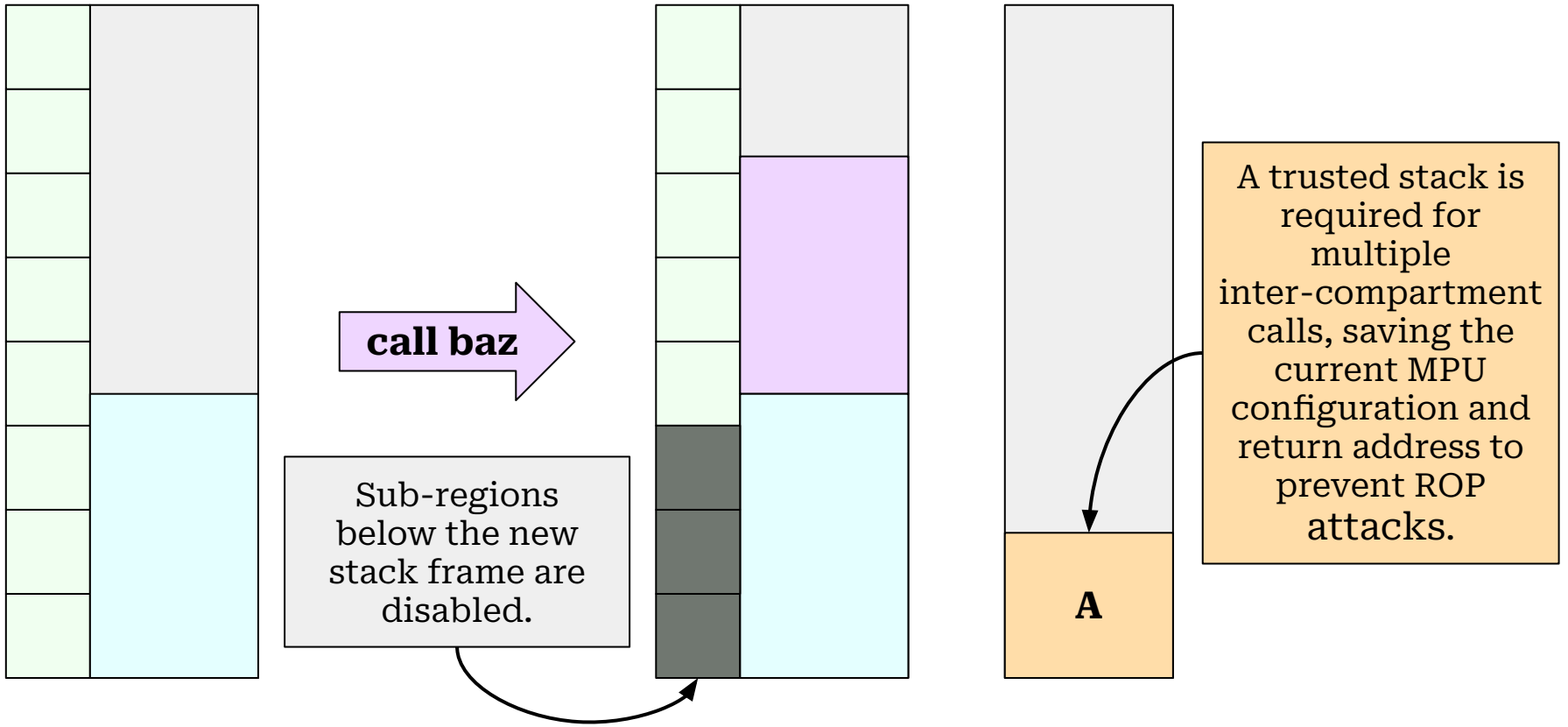
Stack Protection

foo is in compartment A
and
baz in compartment B.

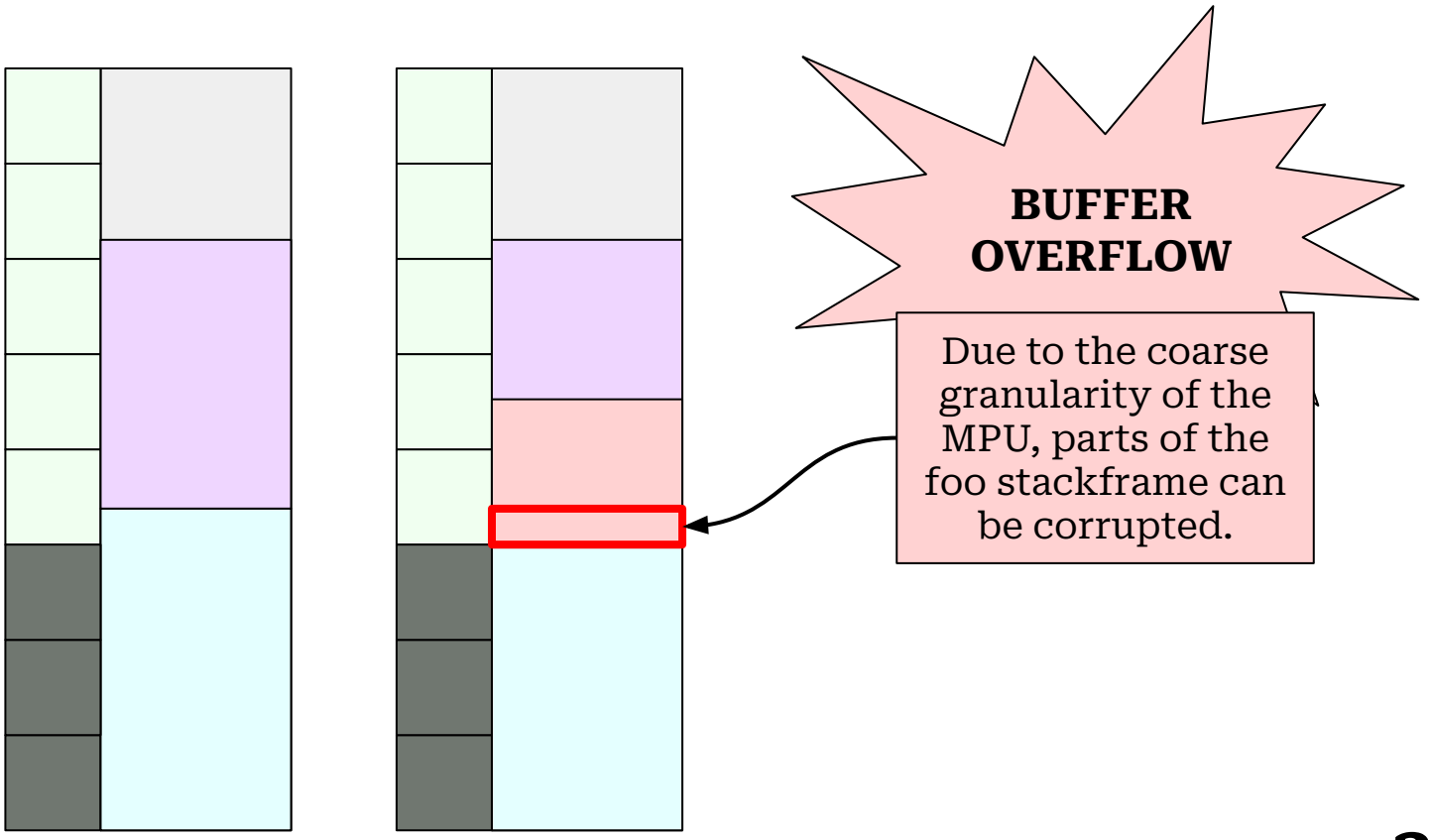


Stack Protection

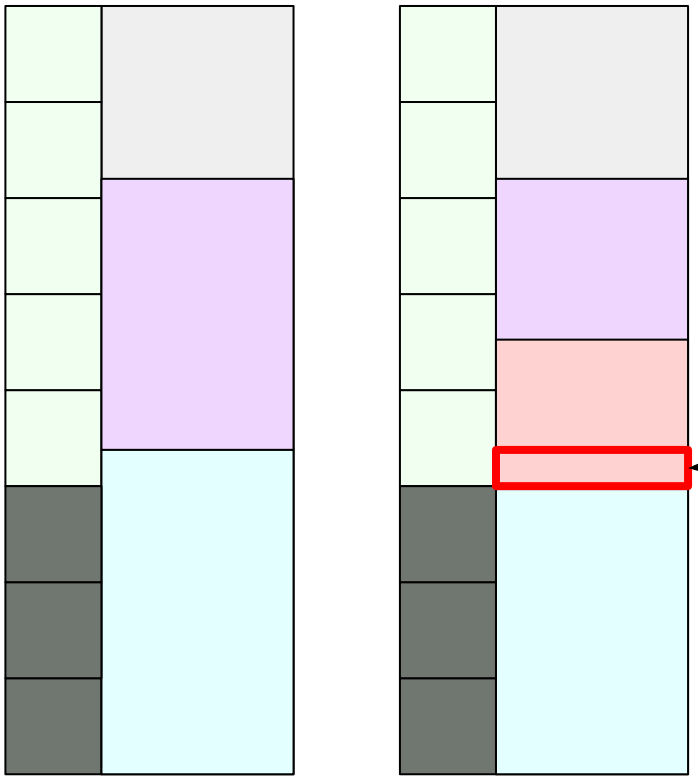
foo is in compartment A
and
baz in compartment B.

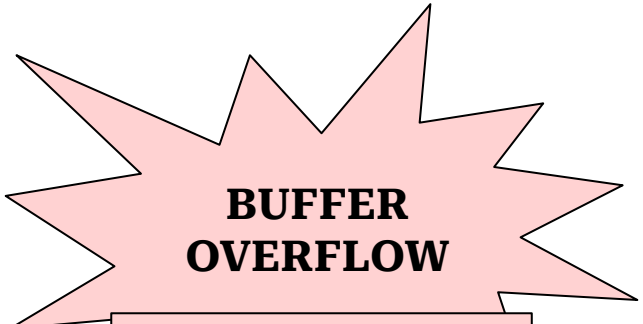


Stack Protection



Stack Protection



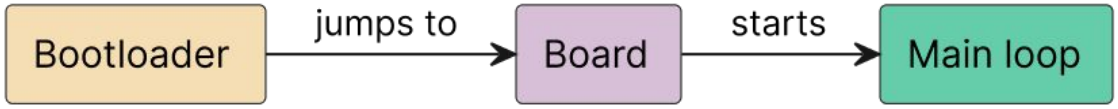


Due to the coarse granularity of the MPU, parts of the foo stackframe can be corrupted.

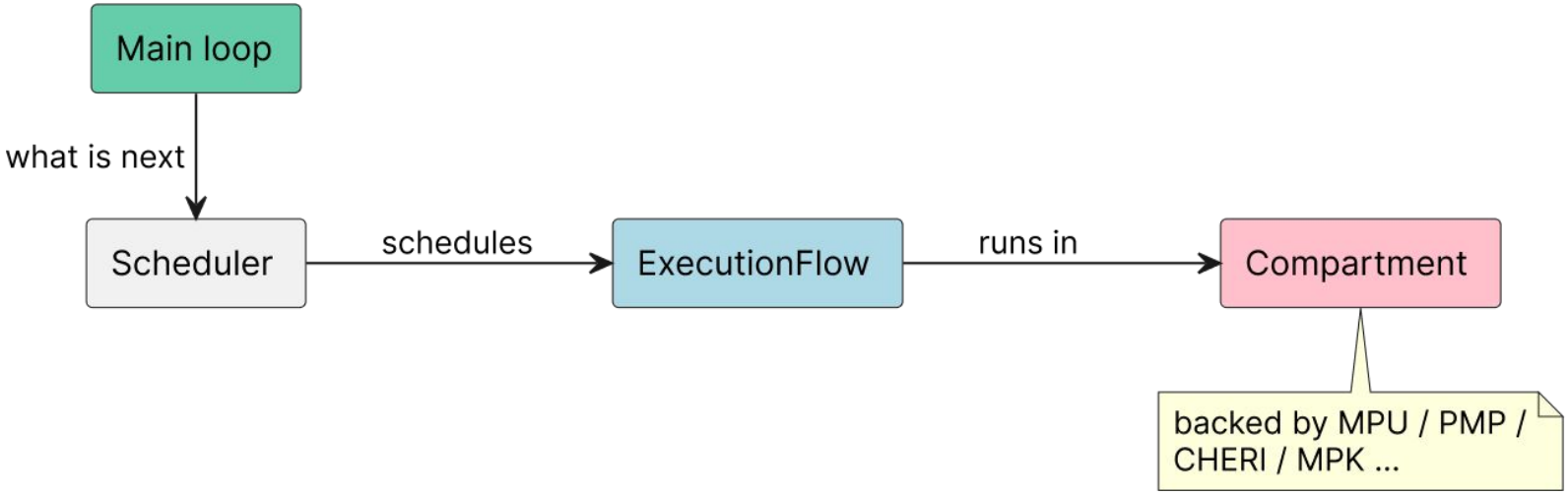
However, since the return address is stored on the trusted stack, an attacker cannot jump into another compartment.

Core Components

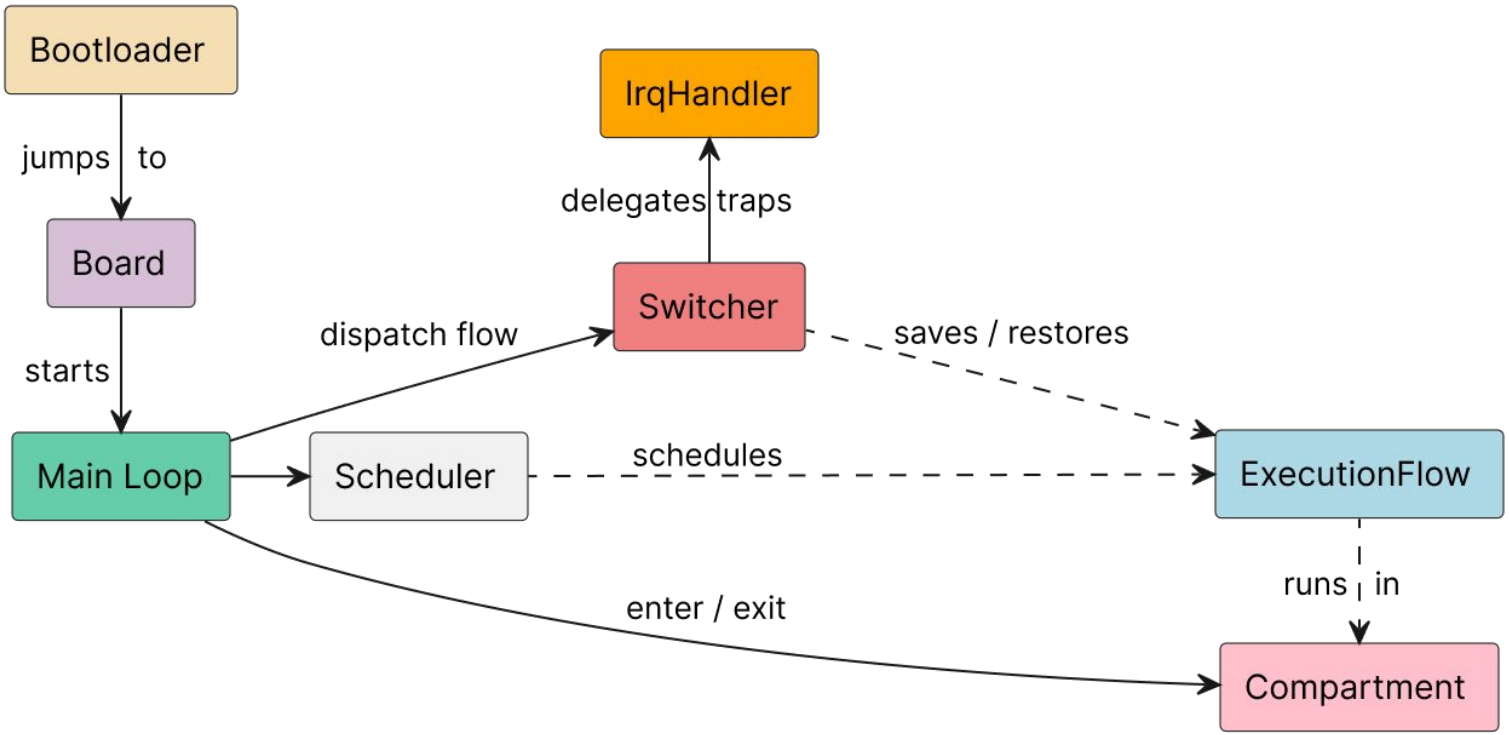
Core Components



Core Components



Core Components



Core Components

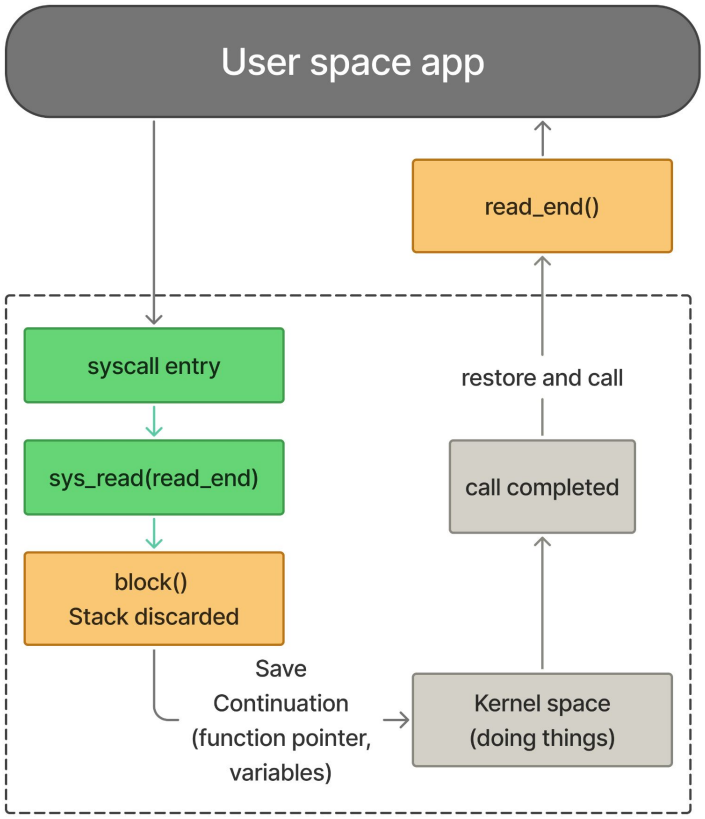
- Event driven execution model
- One kernel stack for every applications

Core Components

- Event driven execution model
- One kernel stack for every applications

Saves Memory	Introduces complexity in the kernel

Execution model



Continuation / Callback

- Current stack is discarded for for resuming
- Used in TockOS, seL4, OKL4, ...

Execution model

- Rust async could be also used

Execution model

- Rust async could be also used

but

Execution model

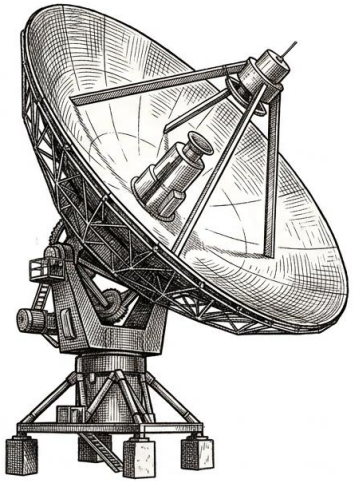
- Rust async could be also used

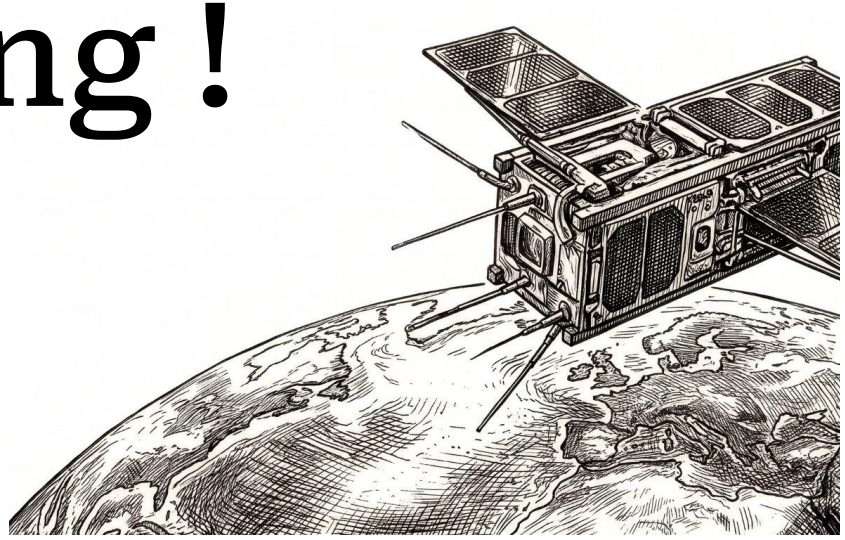
but

" [...] we found that Futures have significant overheads compared to Tock's callback-based design." [1]

[1] *"Tock: From Research to Securing 10 Million Computers"*

Thank you for listening !





Old Slides

Un OS qui prend les spec de l'environnement embarqué des nanosatelite

Les OS qui sont dans ce domaine hérite des OS traditionnel (Process/Thread) alors qu'ils ont différentes caractéristiques

Objectif : faire from scratch qui

Etat de l'art : intègre certaine spécificités mais jamais toutes

La majorité des environnement ont des MPU mais ne les intégre

Besoin d'isolation => on est passé d'un seul service à plusieurs = donc protection nécessaire

On peut avoir différent fonctionnement de sécurité
Hard: Ring level, MPU, MMU, ……
Soft: Rust, …..

Comment on conçoit une interface qui s'adapte à ce choix ?

Système tradi = Process/Thread
=>
Il nous faut quelque chose de générique

Les applications peuvent s'exécuter en flash en ram, pas comme dans OS traditionalismes

WIP, voila ou nous en somme

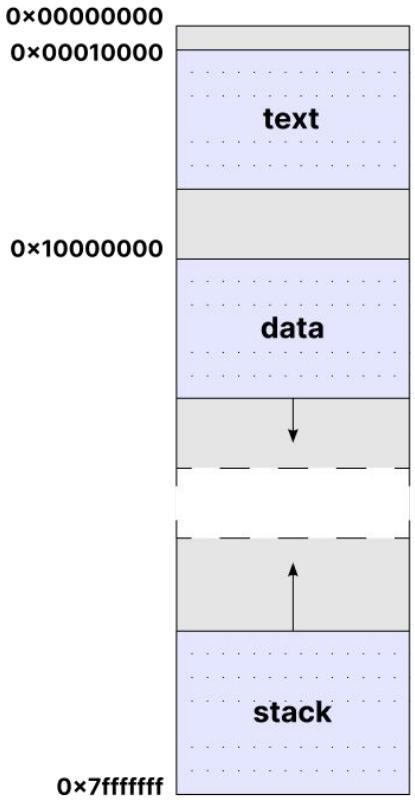
Pourquoi Rust ?
-
-
-

Présentation de l'architecture

Un OS basique, démarrer
Puis avoir l'abstraction de compartmentalisation

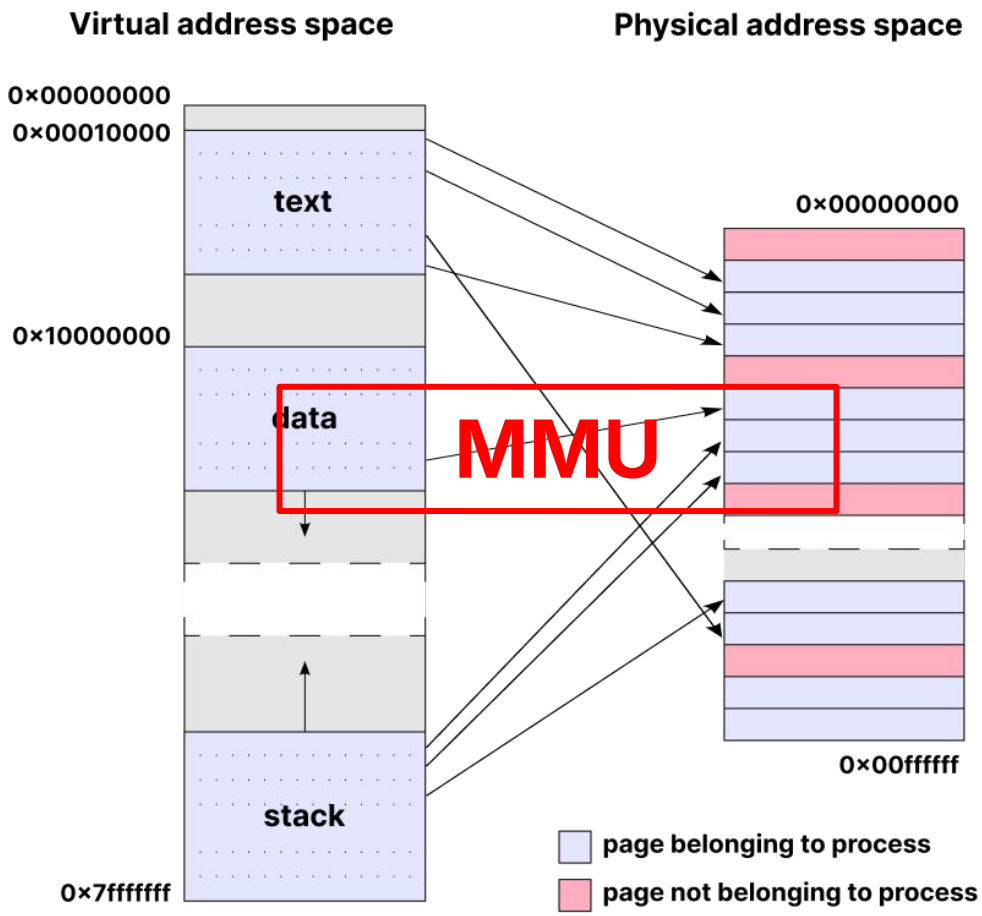
Prise de note

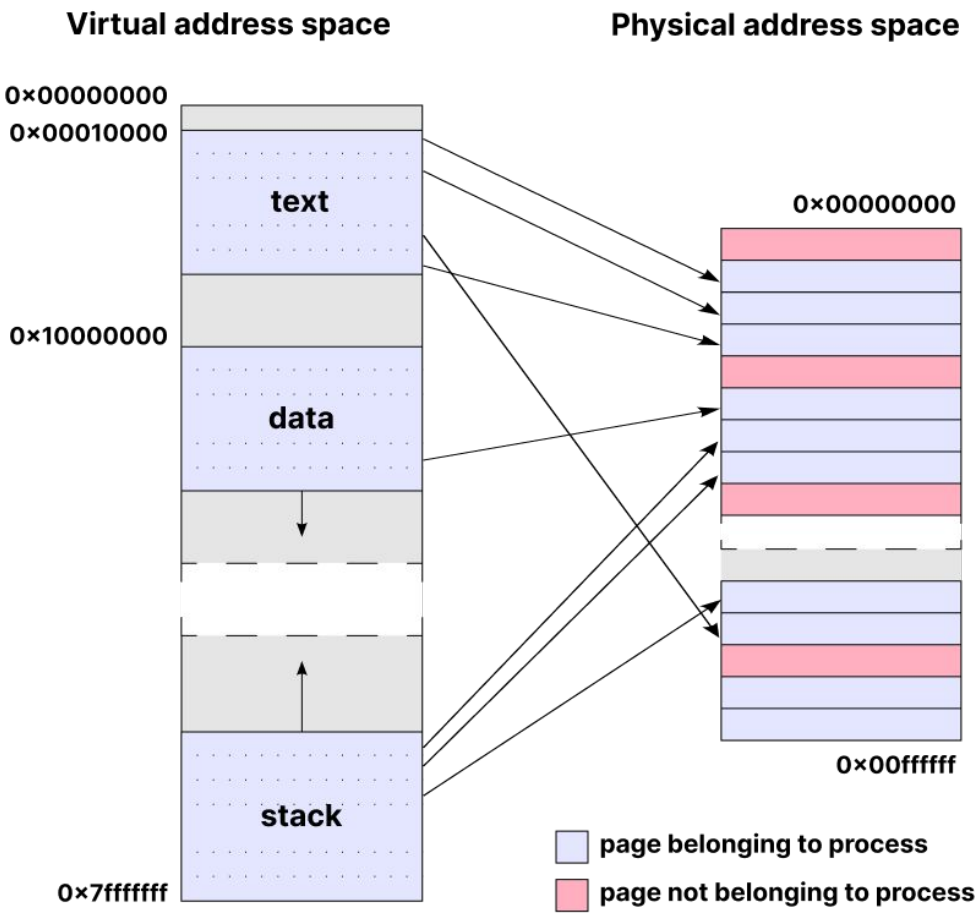
Processus



Thread

- Part of a process


Virtual address space
Physical address space
0×00000000
0×00010000
text
0×10000000
data
MMU
stack
0×7fffffff
0×00000000
0×00ffffff
page belonging to process
page not belonging to process

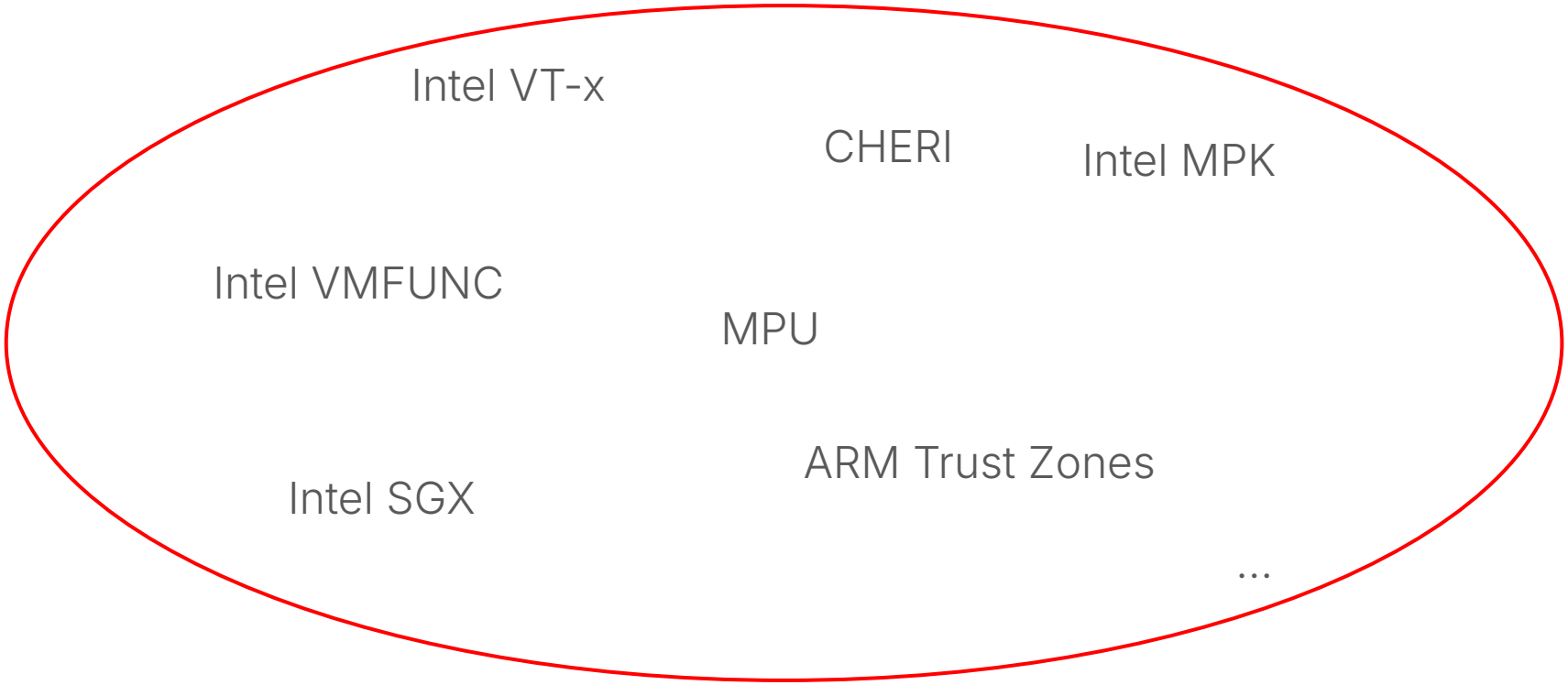


Main goal of MMU

- Prevents a misbehaving or malicious program from modifying or reading data belonging to another program

Memory Management Unit (MMU)

- Foundation of the Processus/Thread model
- Most modern OS use it



Same main goals

Prevents a misbehaving or malicious program from modifying or reading data belonging to another program

Intel VT-x

CHERI

Intel MPK

Intel VMFUNC

MPU

ARM Trust Zones

Intel SGX

...

- How do I map this to the process/thread model ?
- A distinct OS for every hardware feature ?

Memory safety

- No read from none uninitialized memory
- No pointer dereferencement after pointer usage (use-after free)
- No concurrent write without synchronization

Type system

- Each value has a owner (Borrow checker)
- References can’t outlive what they point to (Lifetimes)
- Abstractions with Traits, static by default
- Errors as values

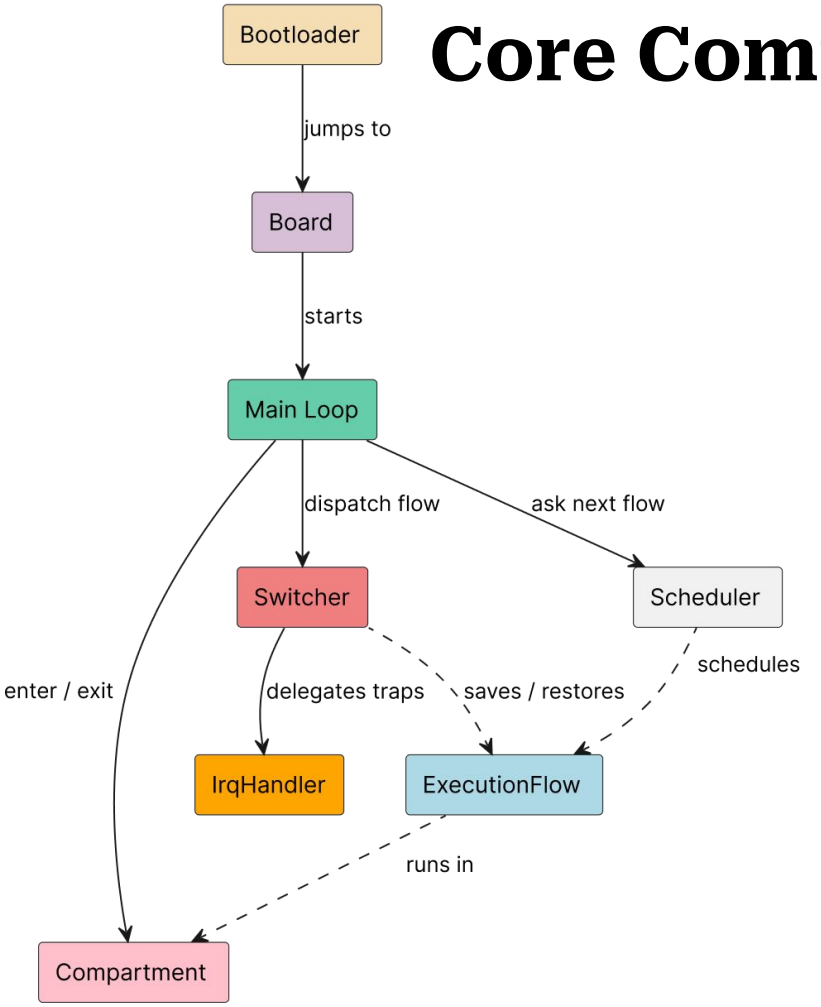
Kernel Execution model

Architecture overview

Current limit

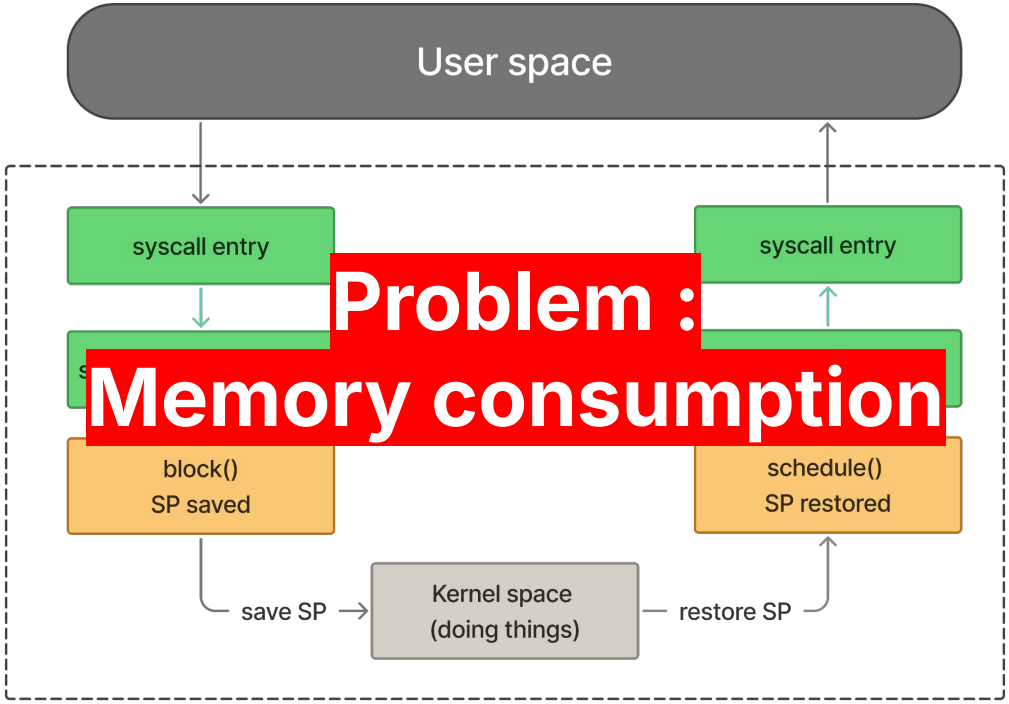
- No support for SMP (only single core)
- Round-robin/Fifo only, no priorities
- No “IPC” (might have another name for it)

Core Components



Per-Thread Kernel Stack

- Every application has one or multiple kernel stack (threads)
- Easy to think about / implement



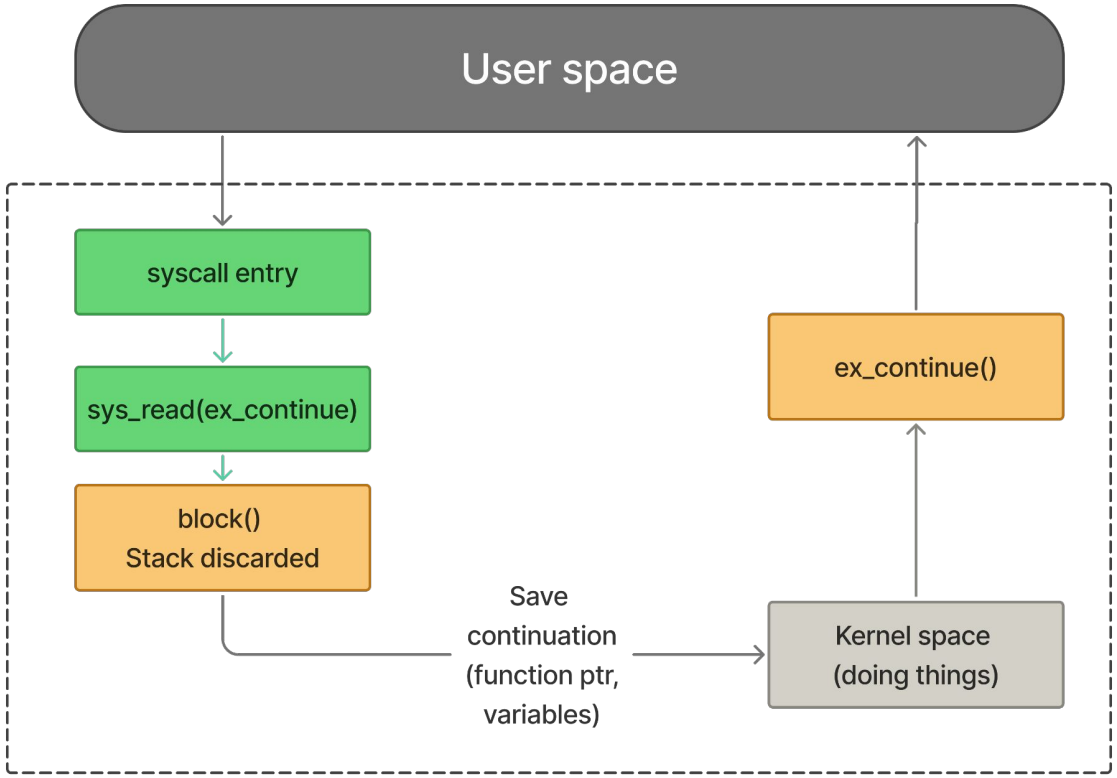
Single Kernel Stack

- Only one kernel stack used for everyone

- **Continuation**
  or
- **Stateless kernel**

Single Kernel Stack

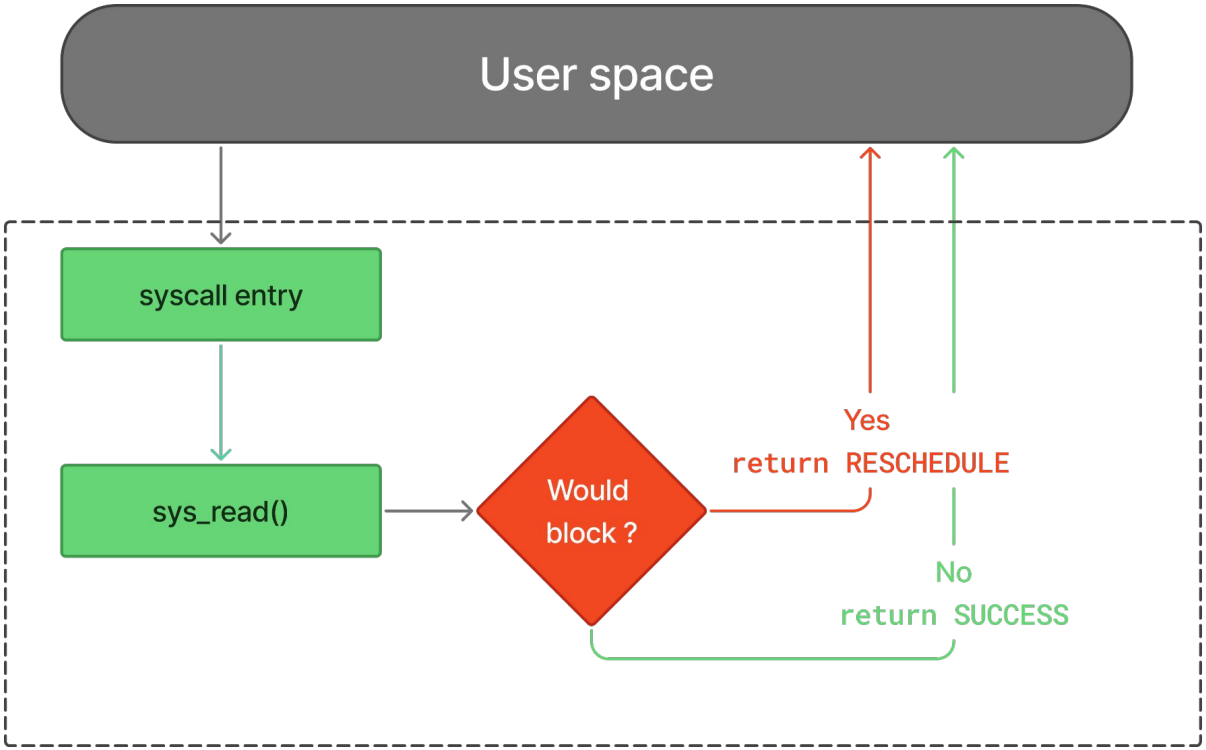
➔ Continuation



- Stack can be reused to support new thread
- Used in seL4, OKL4

Single Kernel Stack

➔ Stateless



- Syscall must be restartable
- No kernel threads
- Kernel not interruptible