A Design for Comprehensive Kernel Instrumentation

Peter Feiner
peter@cs.toronto.edu

Angela Demke Brown
demke@cs.toronto.edu

Ashvin Goel
ashvin@eecg.toronto.edu

University of Toronto

Motivation

Transparent fault isolation for device drivers

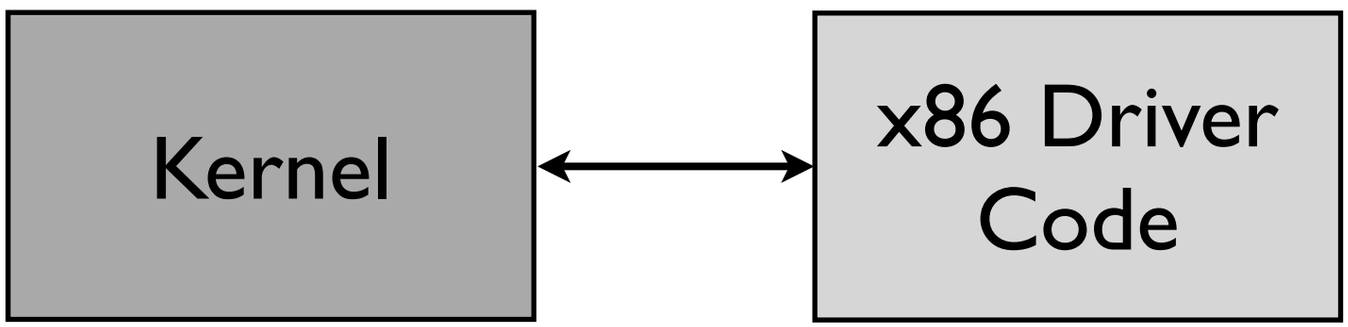
- Want to isolate existing driver binaries

Inspired by Byte Granularity Isolation

- Requires source code

Use Dynamic Binary Instrumentation (DBI)

- Does not require source code
- Inspect & modify instructions before they execute



Motivation

Transparent fault isolation for device drivers

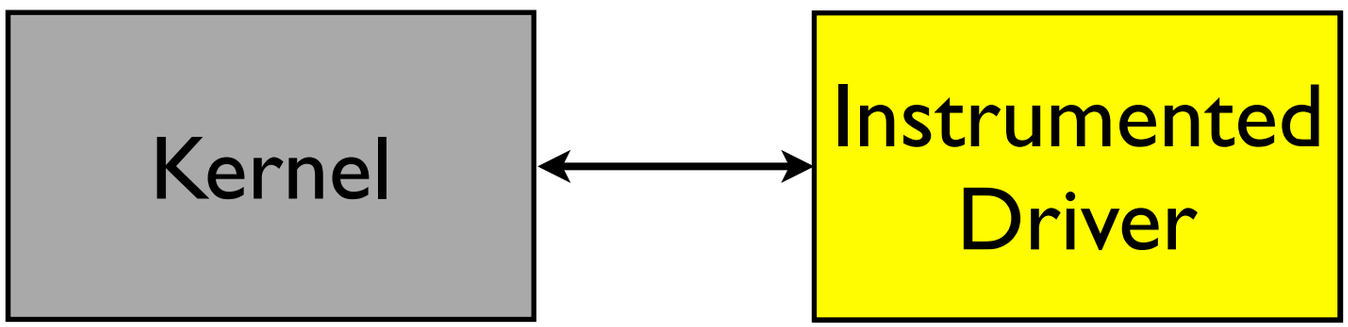
- Want to isolate existing driver binaries

Inspired by Byte Granularity Isolation

- Requires source code

Use Dynamic Binary Instrumentation (DBI)

- Does not require source code
- Inspect & modify instructions before they execute



Motivation

Transparent fault isolation for device drivers

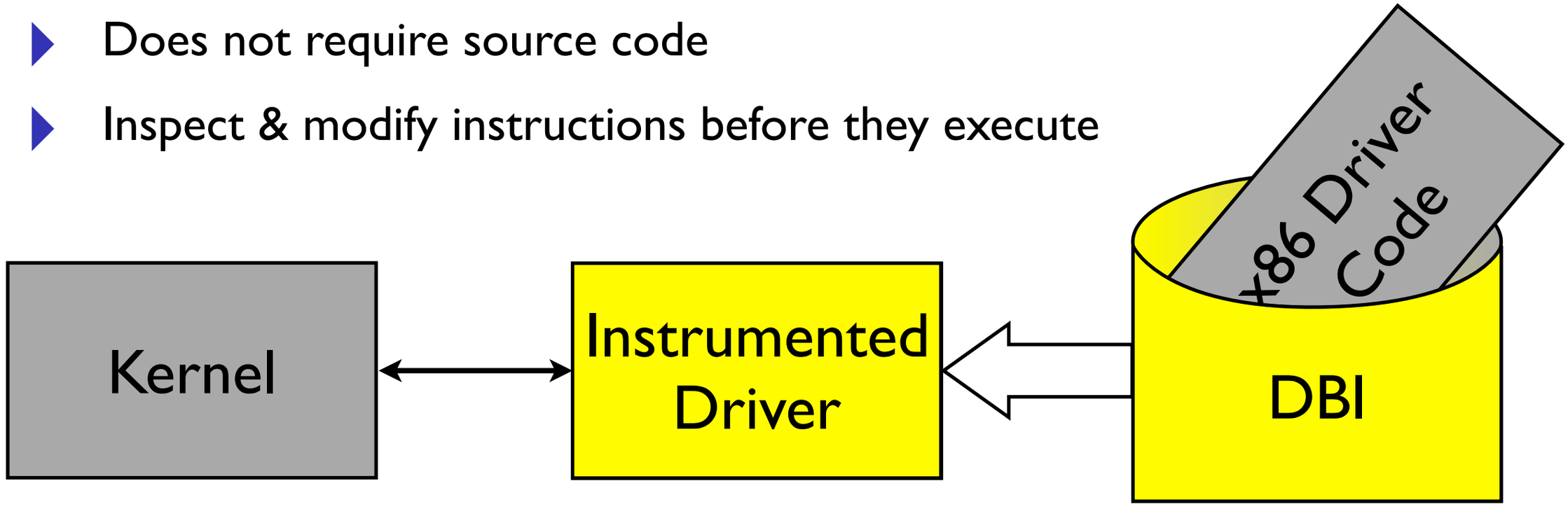
- Want to isolate existing driver binaries

Inspired by Byte Granularity Isolation

- Requires source code

Use Dynamic Binary Instrumentation (DBI)

- Does not require source code
- Inspect & modify instructions before they execute



Motivation

DBI applied for debugging and security at the user level

- Memcheck - checks memory errors
- Program Shepherding - control flow integrity

Various user-level DBI frameworks are available

- APIs for inspecting and modifying instructions
- e.g., Valgrind, DynamoRIO, Pin

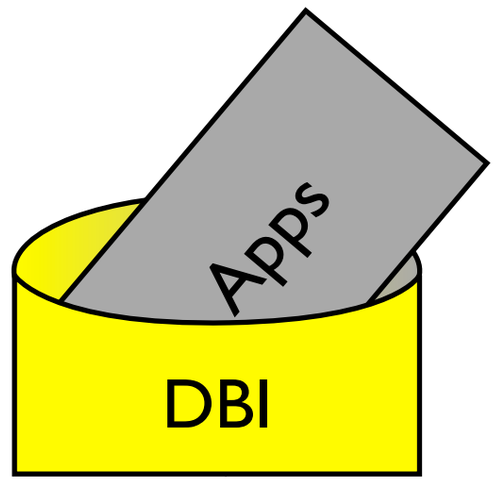
These frameworks don't work in the kernel

- What would it take?

The Key Difference

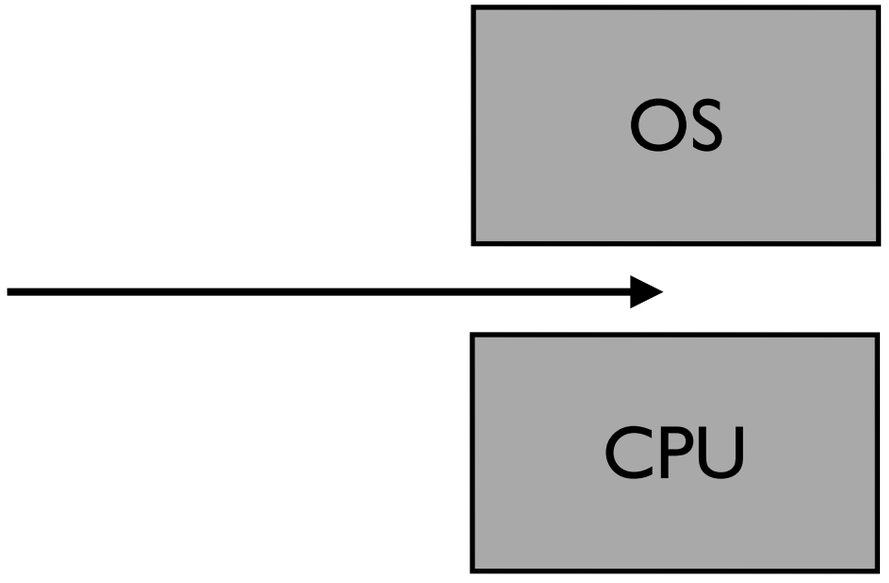
User frameworks sit between applications and the OS

- Interpose on system calls
- Take advantage of OS services, e.g. I/O



Kernel frameworks need to sit between the OS & CPU

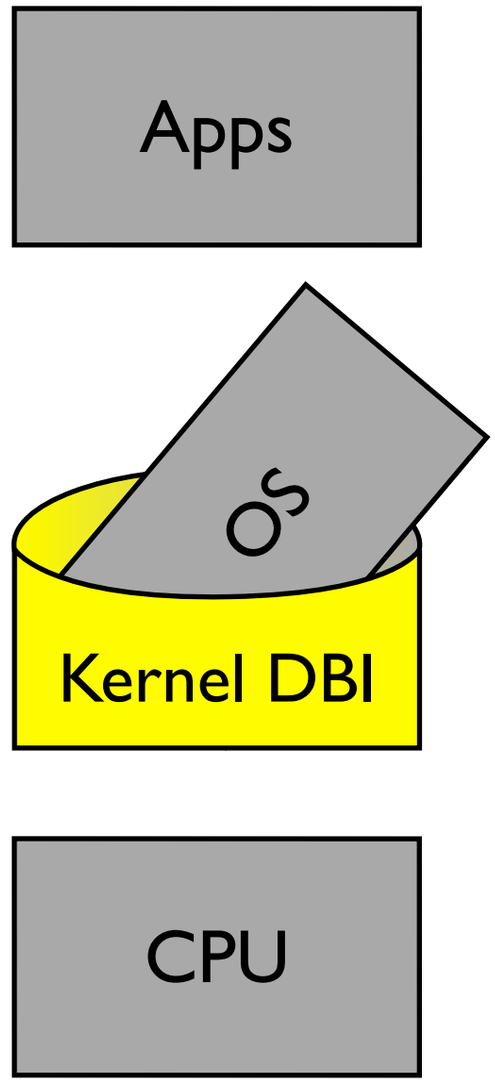
- Isn't that what hypervisors do?



Our Approach

We need to combine a DBI framework with a hypervisor

- Choice 1: Port DBI to an existing hypervisor
  - Pros: both exist
  - Cons: both very complex
- Choice 2: Create a minimal hypervisor, similar to SecVisor's approach
  - Pros: easier to do
  - Pros: possibly higher performance



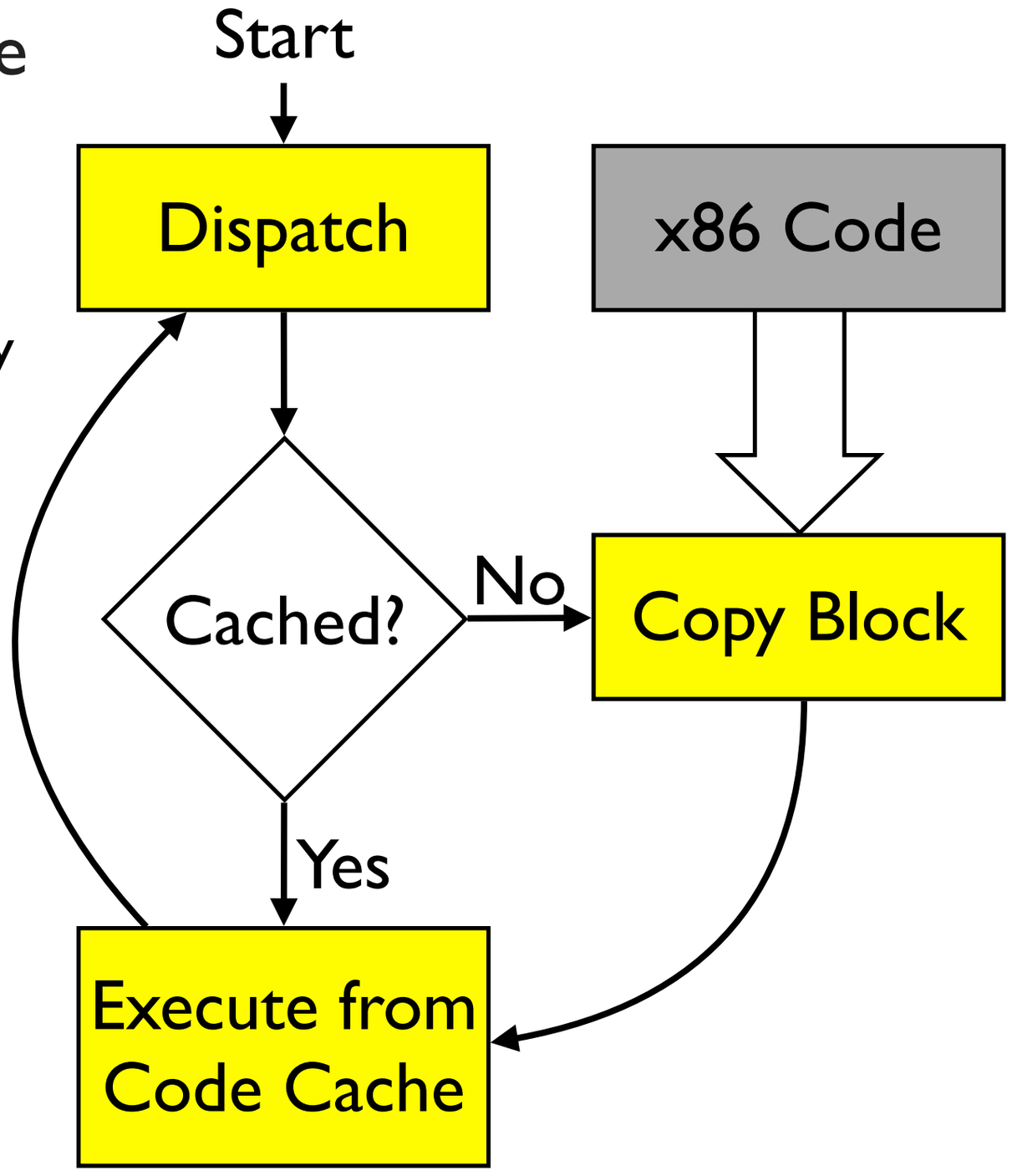
We designed a minimal hypervisor around a DBI framework

- Let's see how DBI works & what it needs

DBI Technique

Copy basic blocks of x86 code into code cache before execution

- Code executed from cache
- Instrumentation added to copy
- Manipulate copies to return control to the dispatcher



DBI Requirements

Never execute machine's original code

- Necessary for security applications

Hide framework from instrumented code

- Instrumented code should observe un-instrumented machine state

Dispatcher should use instrumented code with care

- Implementation cannot use non-reentrant instrumented code

Detect changes to the original code

- Invalidate stale code in the cache

Preserve multicore concurrency

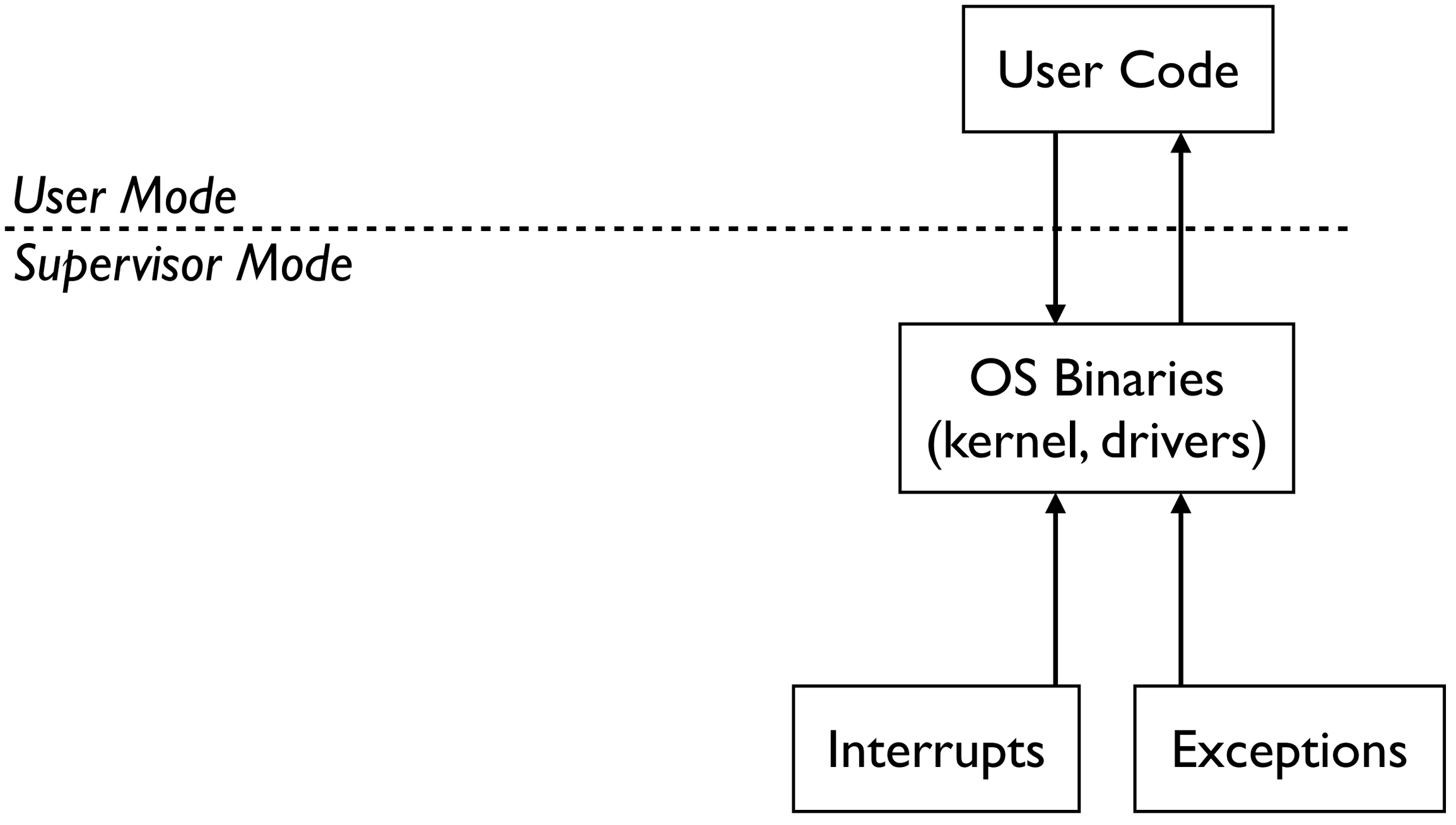
- Essential for performance and accuracy

Meeting DBI Requirements

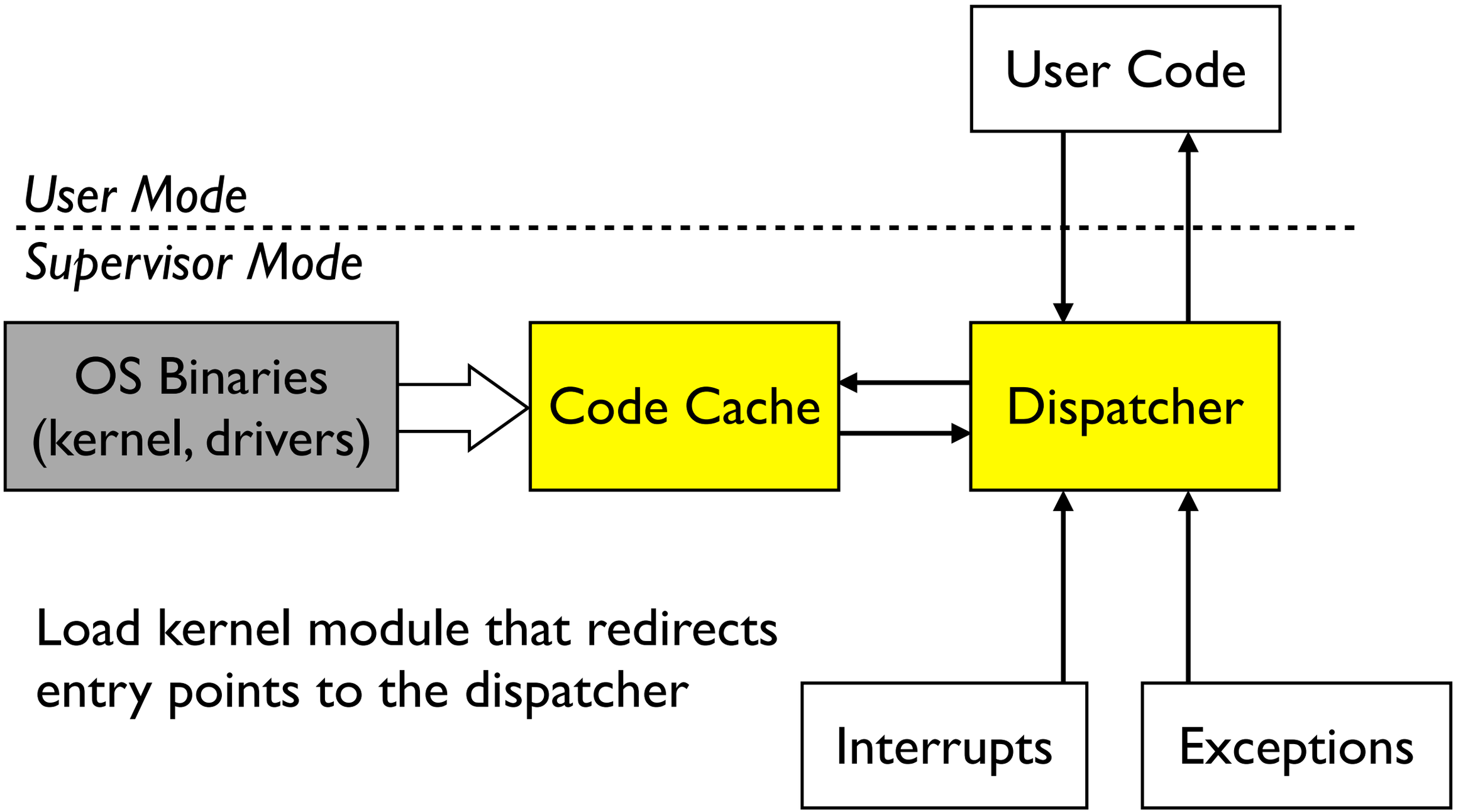
	User	Kernel
Never Execute Original Code	New Threads, Signals	Kernel Entry Points
Transparency	Signals	Interrupts, Exceptions
Reentrance	Use OS Code	Implement Everything From Scratch
Detect Code Changes	System Calls mmap, mprotect, etc.	Shadow Page Tables
Concurrency	Locking, Thread Private	CPU Private

We'll look at the first three in more detail

Never Execute Original Code

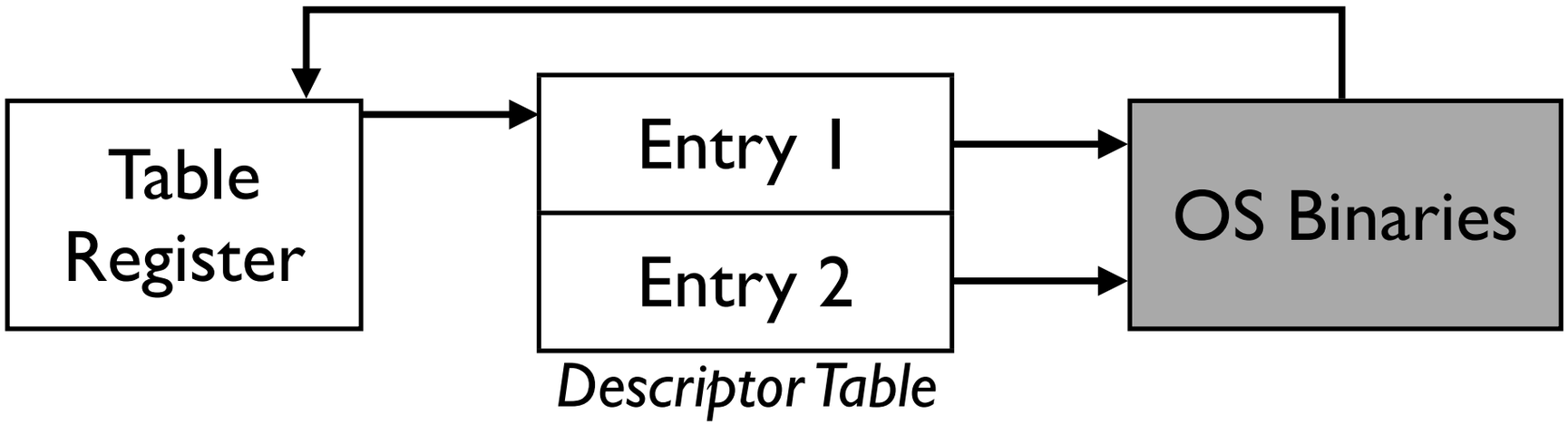


Never Execute Original Code

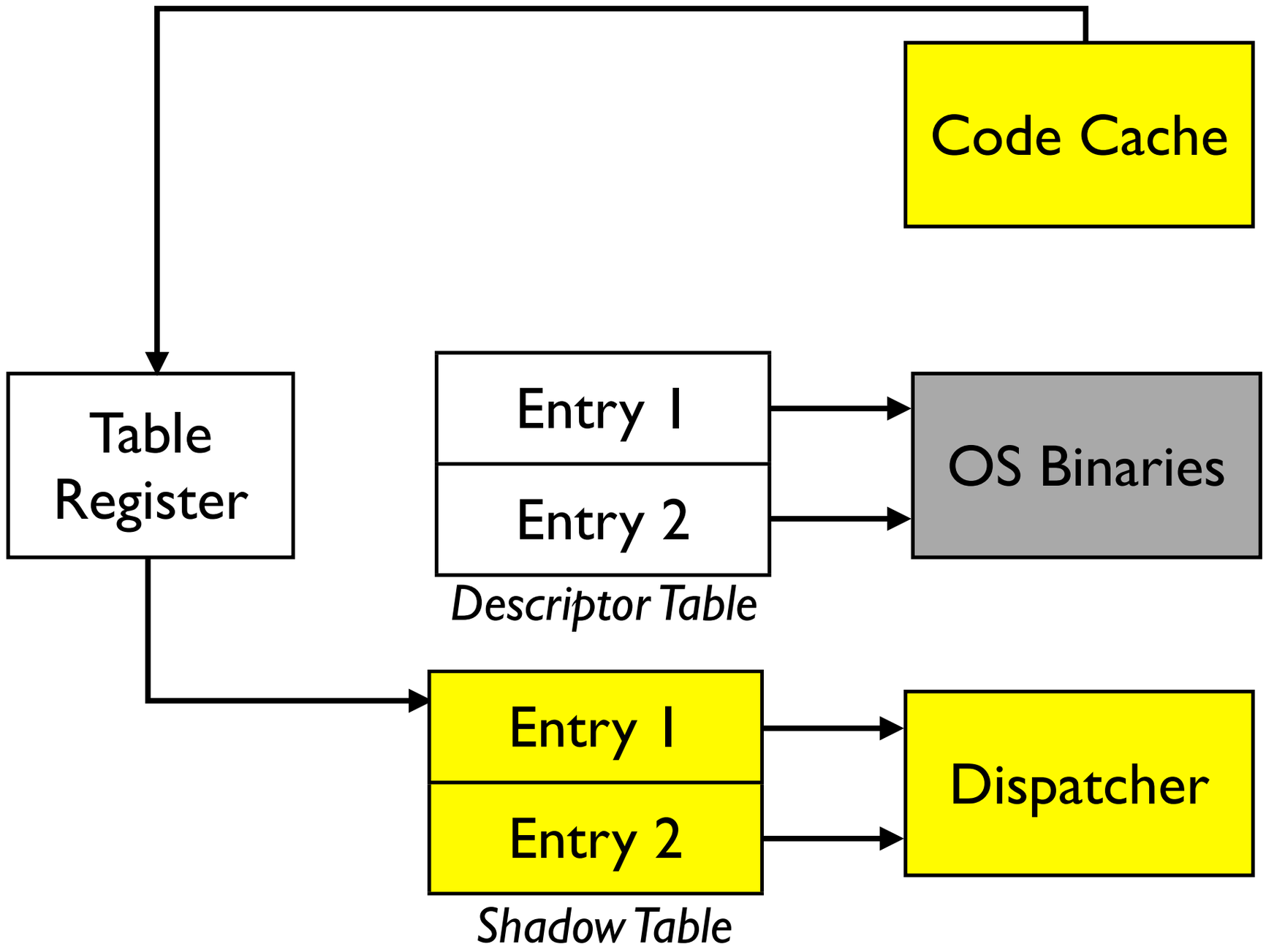


- Load kernel module that redirects entry points to the dispatcher

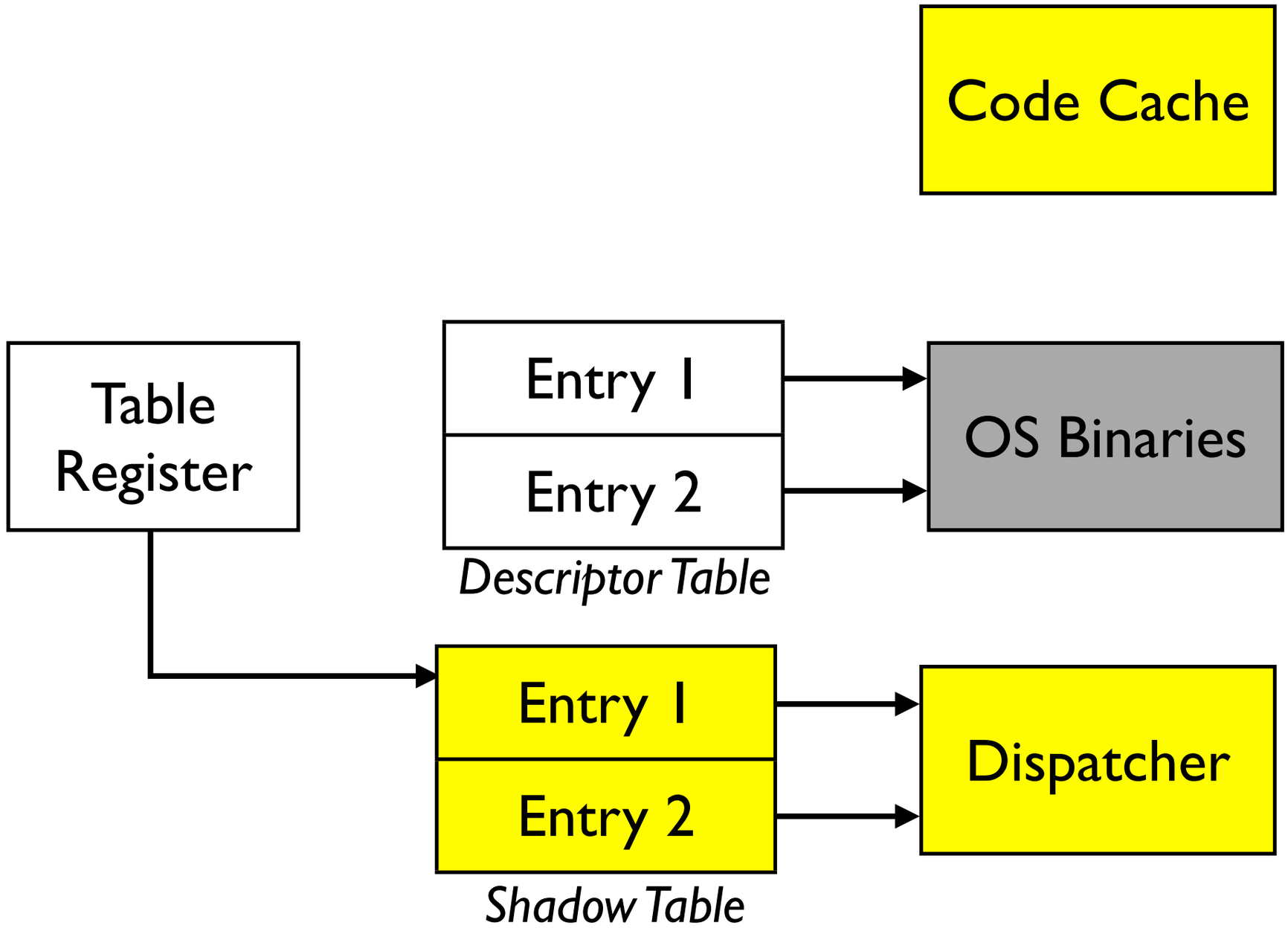
Redirecting Entry Points



Redirecting Entry Points

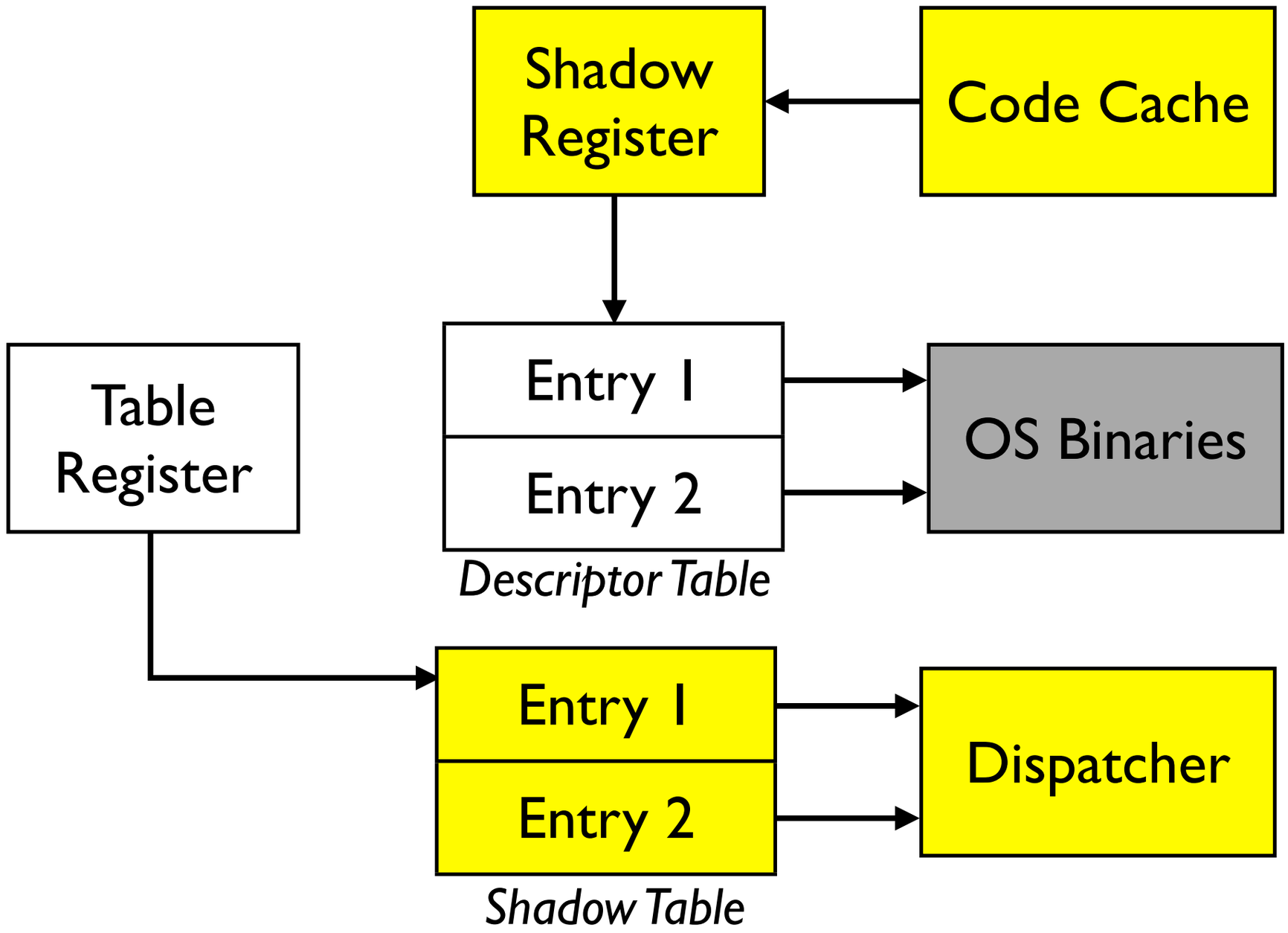


Redirecting Entry Points



- Can't write to table register, otherwise loose control

Redirecting Entry Points



- Can't write to table register, otherwise loose control
- Can't drop the write, otherwise you loose transparency

Transparency

Need to hide DBI framework from instrumented code

- Sometimes essential for correctness

Many transparency issues, including

- Code cache return addresses
- Shadowed registers
- Exception stack frame
- Interrupt stack frame

Exception Transparency

Dispatching kernel's exception handlers is tricky because they inspect machine state

- Registers stolen by instrumentation
- Address of instruction that triggers the exception
  - Handlers need to see original instruction addresses
  - Linux panics on page faults from non white-listed instructions
  - Problem is that code cache isn't on the white list
  - Solution is to translate from code cache to original address

Solution for interrupt handlers is similar

Interrupt Transparency

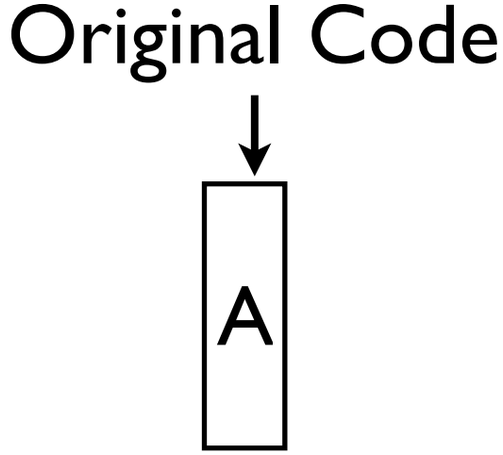
H = Interrupt Handler

I = Instrumentation

I = Interrupt

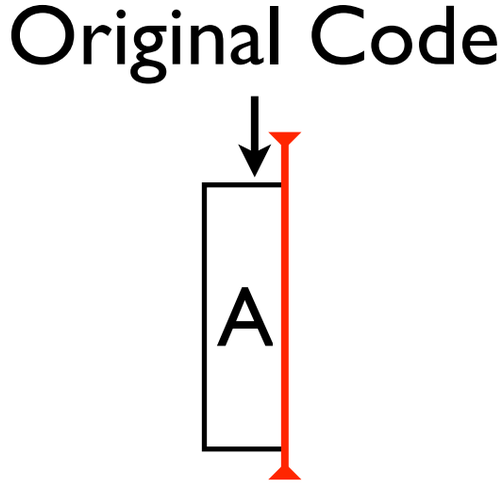
Interrupt Transparency

H = Interrupt Handler
I = Instrumentation
I = Interrupt



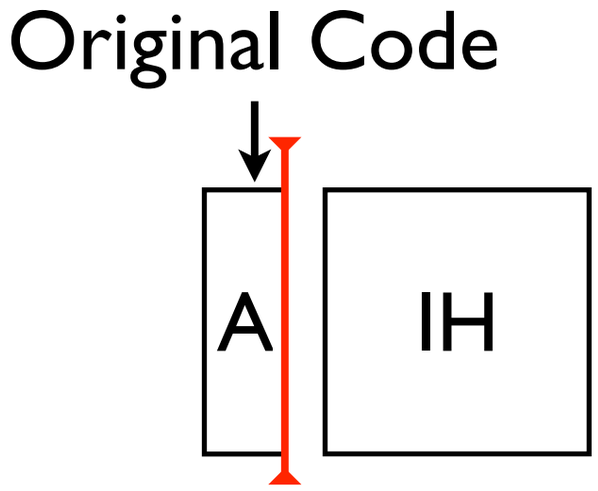
Interrupt Transparency

H = Interrupt Handler
I = Instrumentation
I = Interrupt



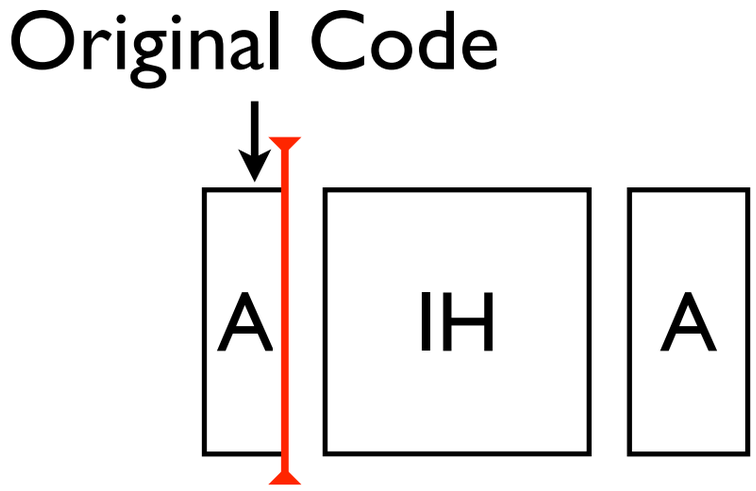
Interrupt Transparency

H = Interrupt Handler
I = Instrumentation
I = Interrupt



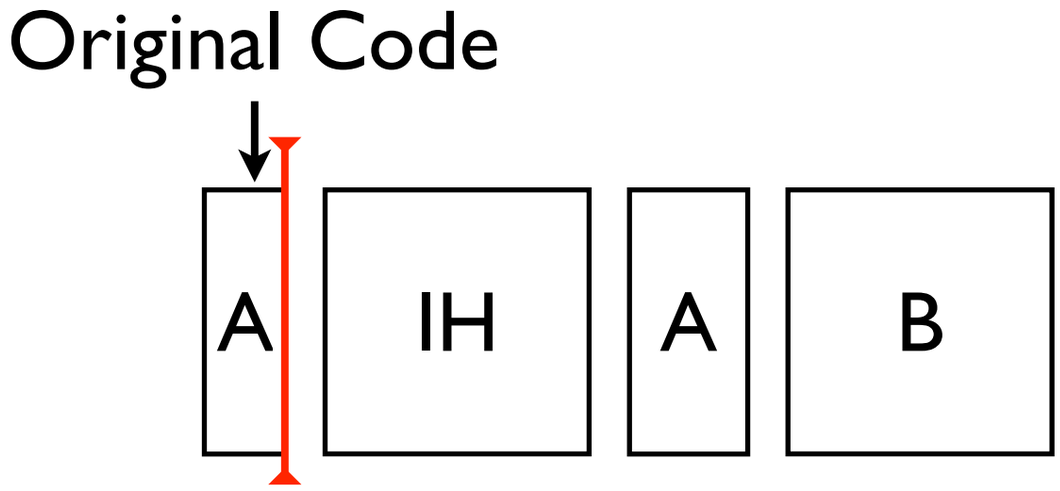
Interrupt Transparency

H = Interrupt Handler
I = Instrumentation
I = Interrupt



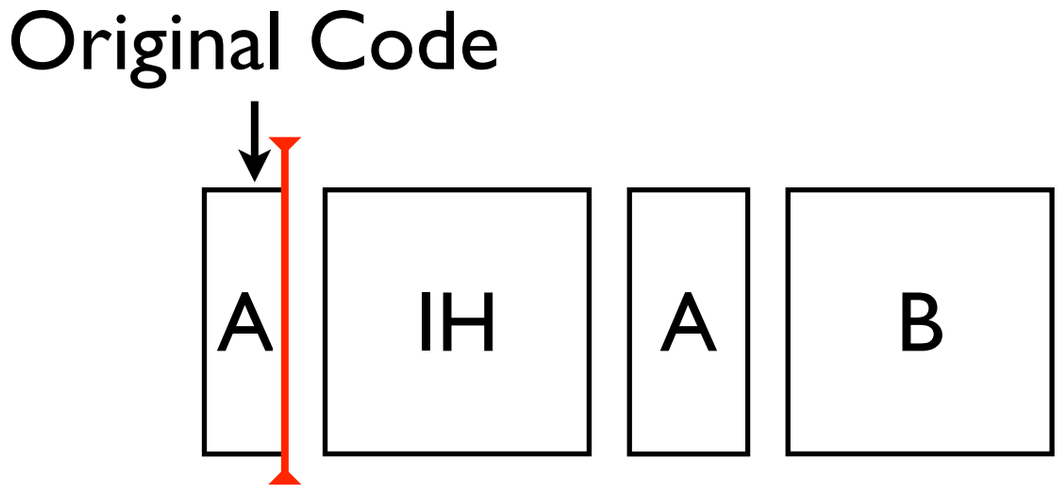
Interrupt Transparency

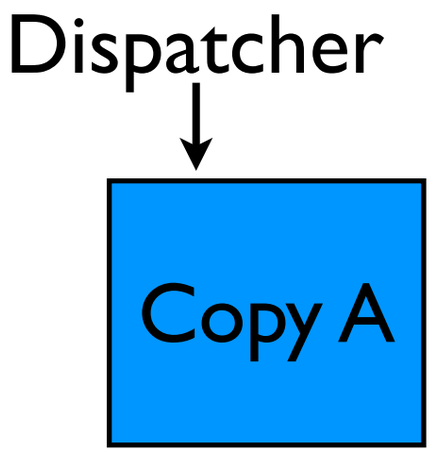
H = Interrupt Handler
I = Instrumentation
I = Interrupt



Interrupt Transparency

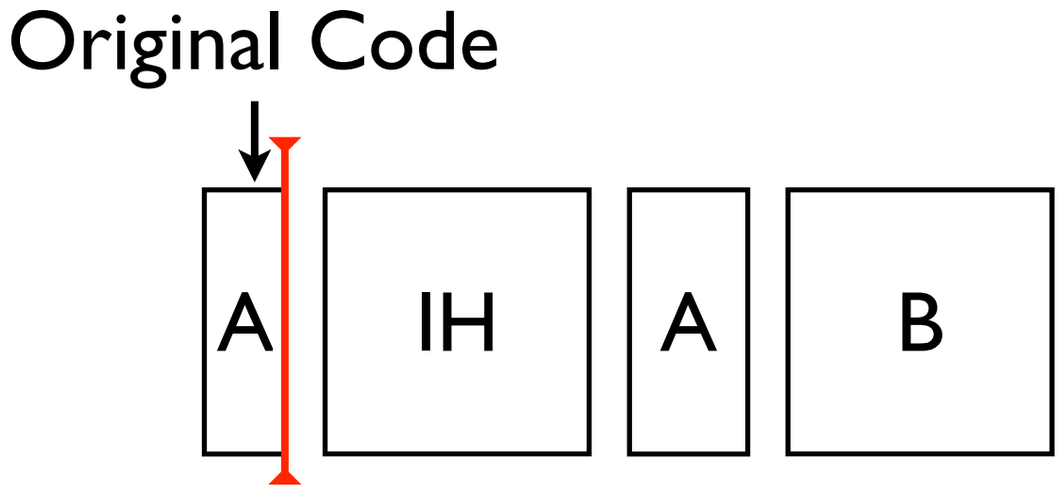
H = Interrupt Handler
I = Instrumentation
I = Interrupt

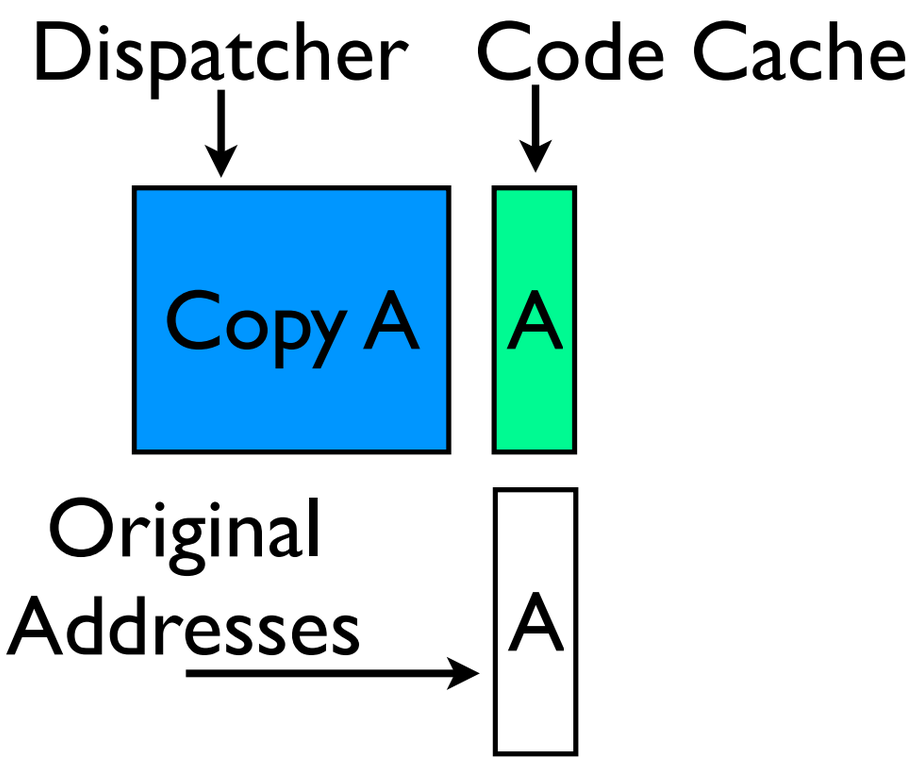




Interrupt Transparency

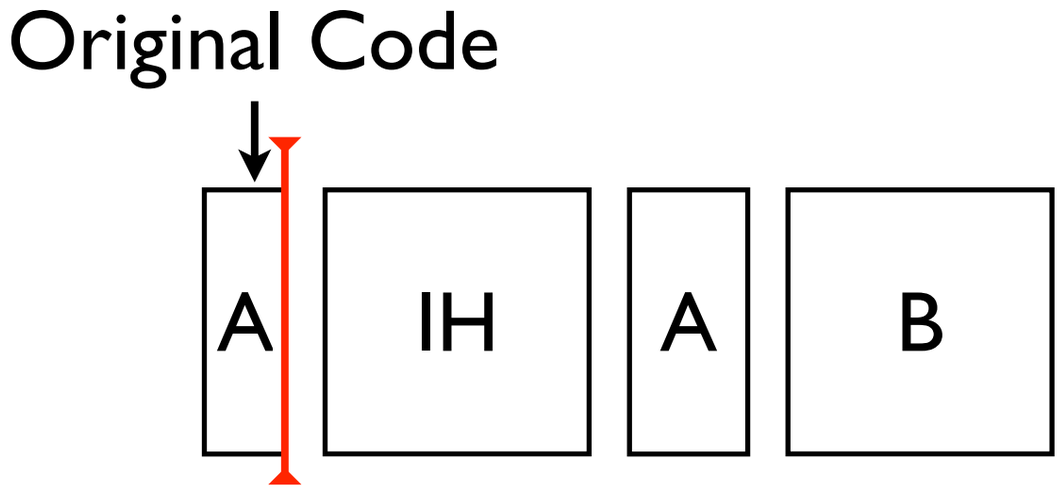
H = Interrupt Handler
I = Instrumentation
I = Interrupt

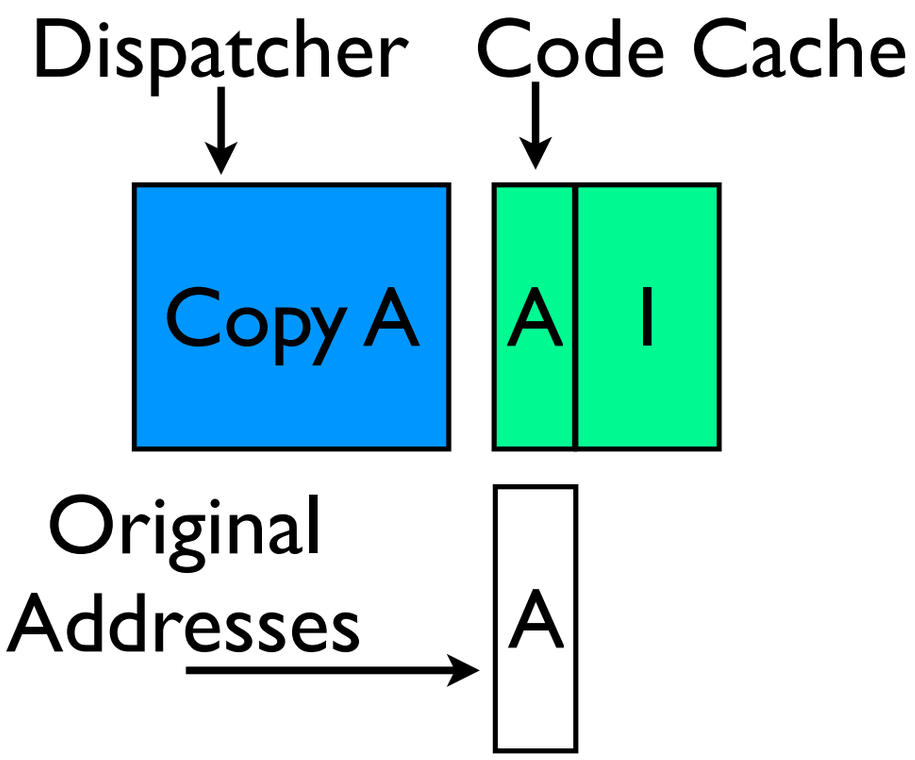




Interrupt Transparency

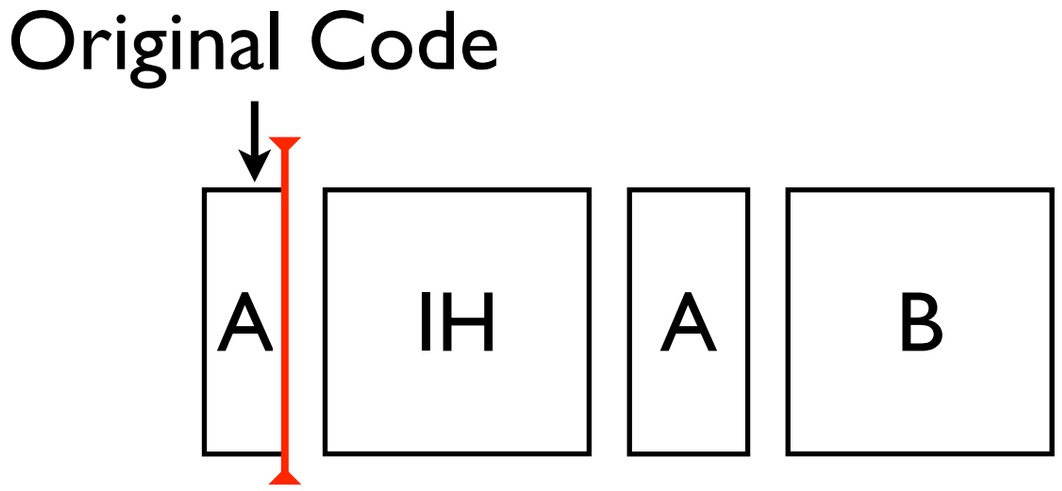
H = Interrupt Handler
I = Instrumentation
I = Interrupt

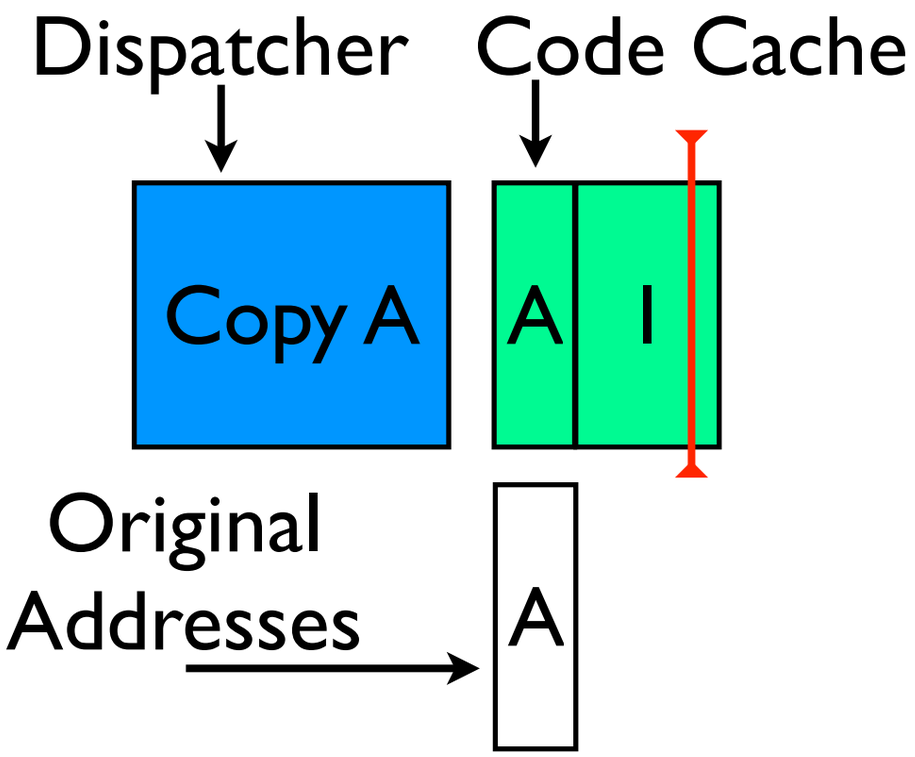




Interrupt Transparency

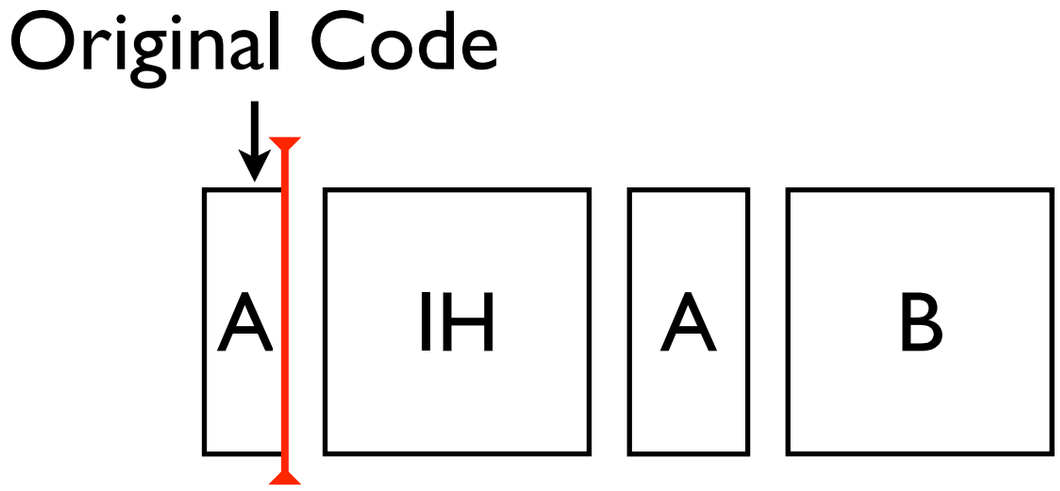
H = Interrupt Handler
I = Instrumentation
I = Interrupt

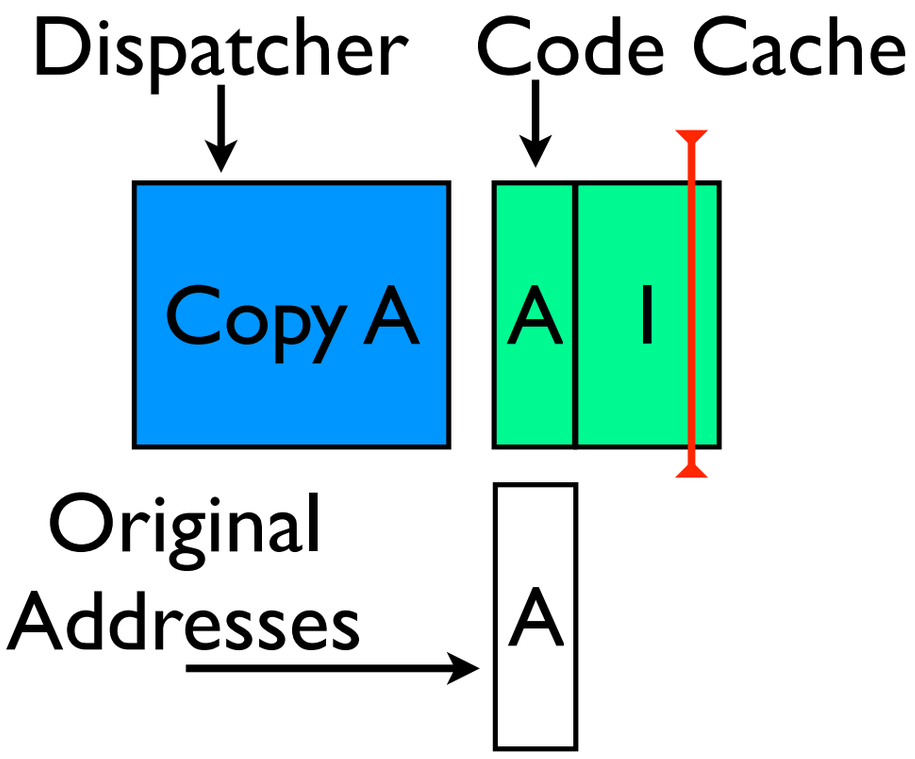




Interrupt Transparency

H = Interrupt Handler
I = Instrumentation
I = Interrupt

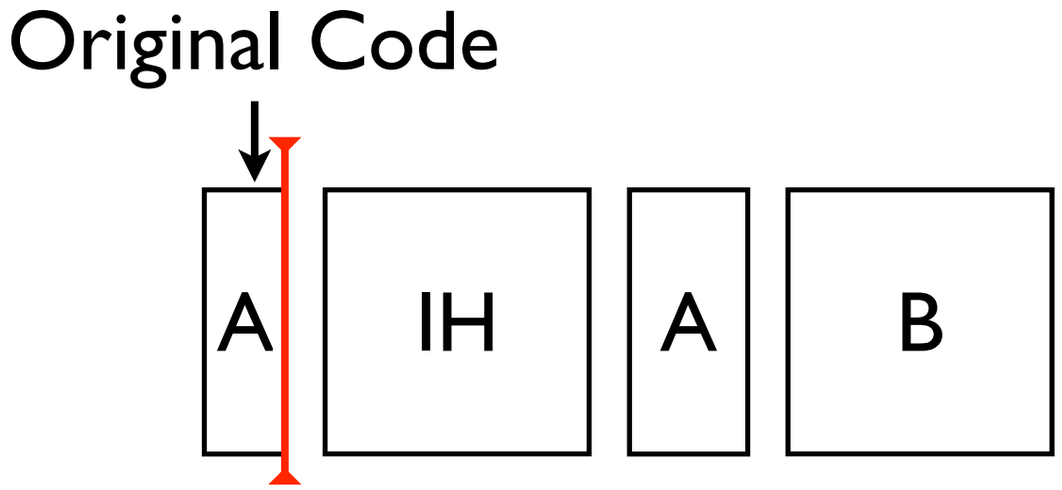


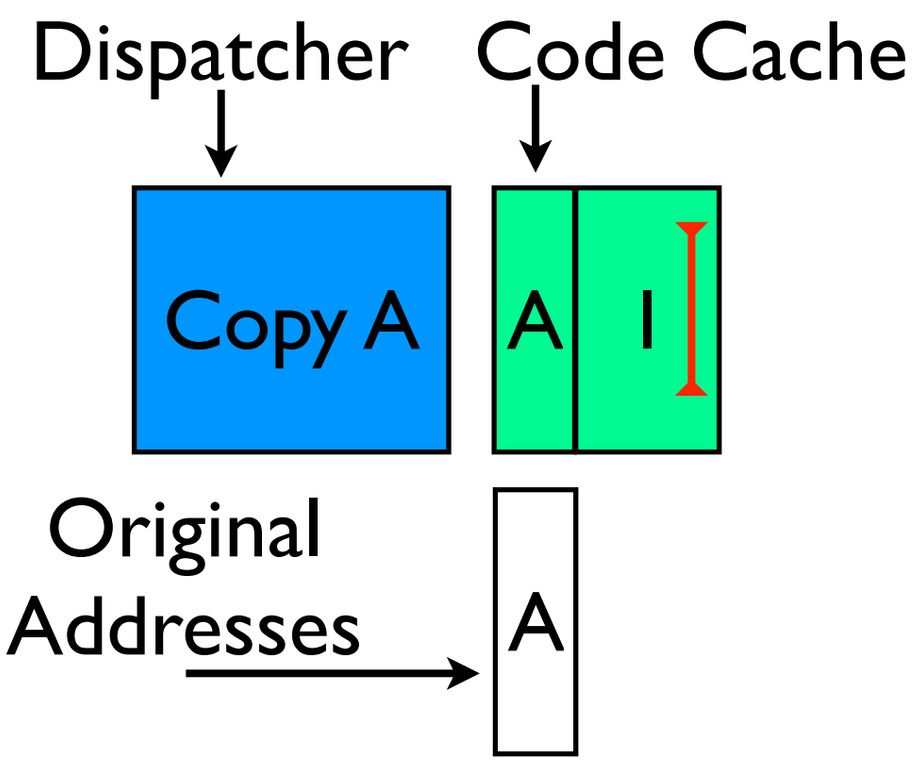


Delay interrupts until next code-cache exit

Interrupt Transparency

H = Interrupt Handler
I = Interrupt
I = Interrupt

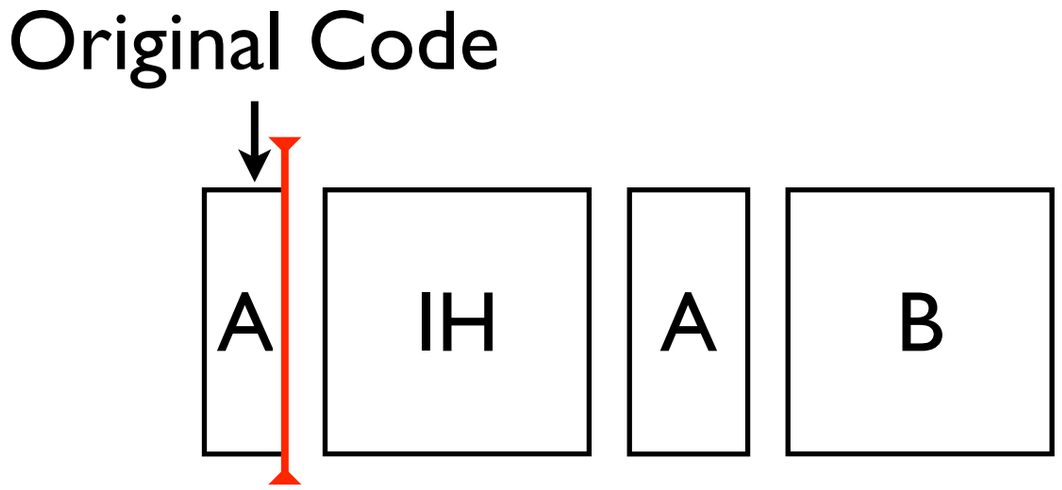


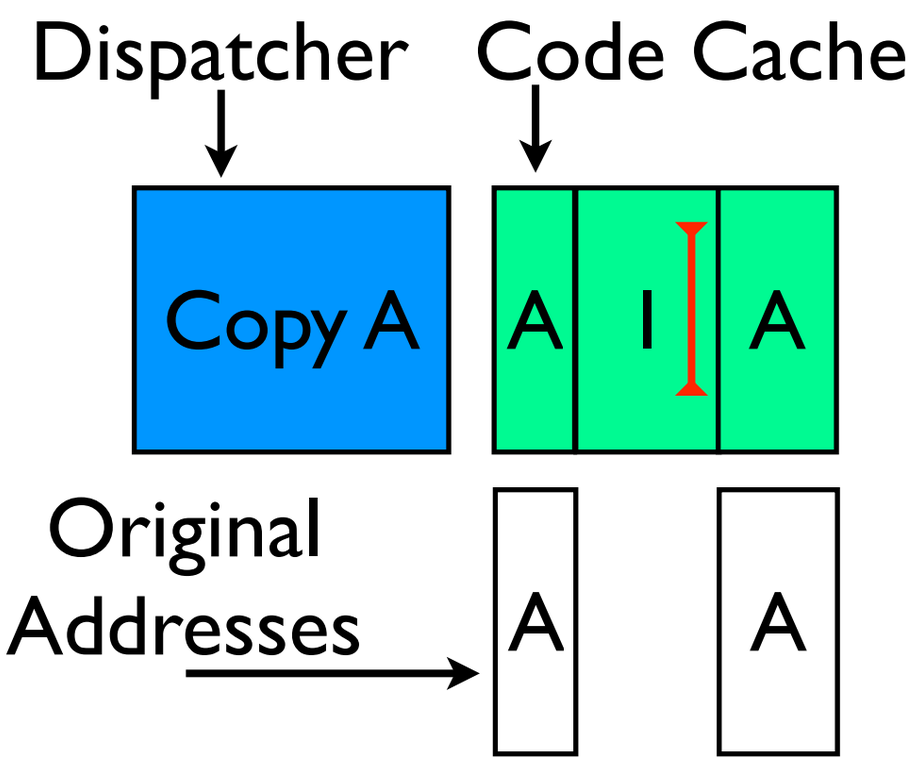


Delay interrupts until next code-cache exit

Interrupt Transparency

H = Interrupt Handler
I = Instrumentation
I = Interrupt

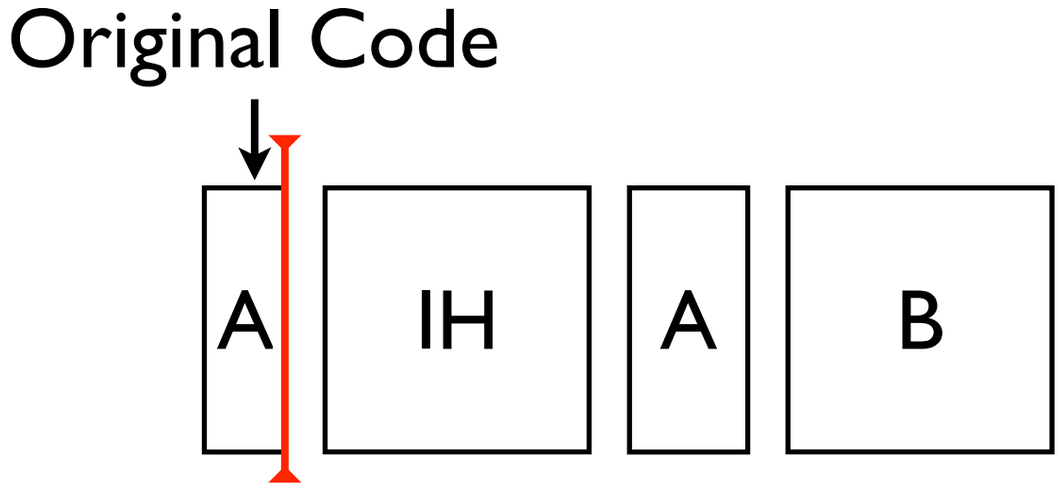


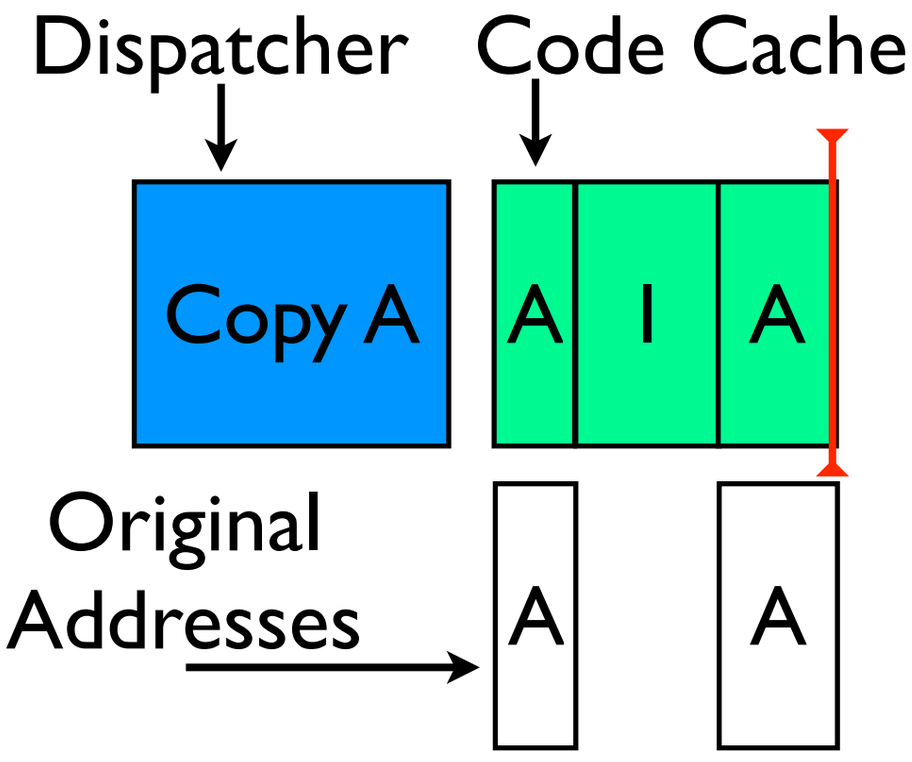


Delay interrupts until next code-cache exit

Interrupt Transparency

H = Interrupt Handler
I = Instrumentation
I = Interrupt

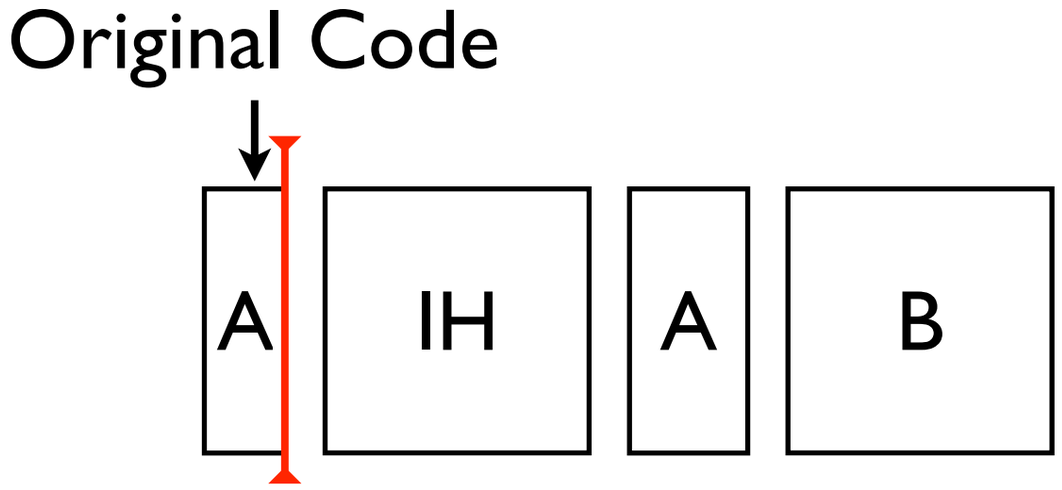


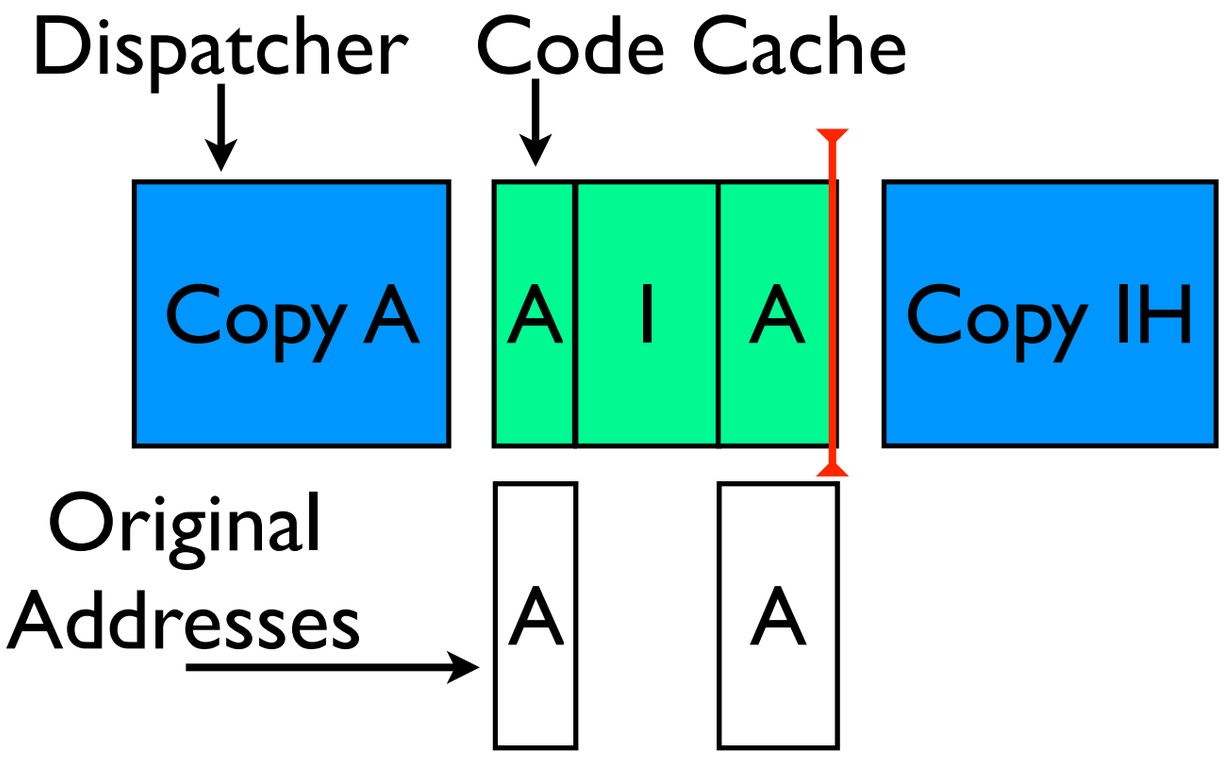


Delay interrupts until next code-cache exit

Interrupt Transparency

H = Interrupt Handler
I = Instrumentation
I = Interrupt

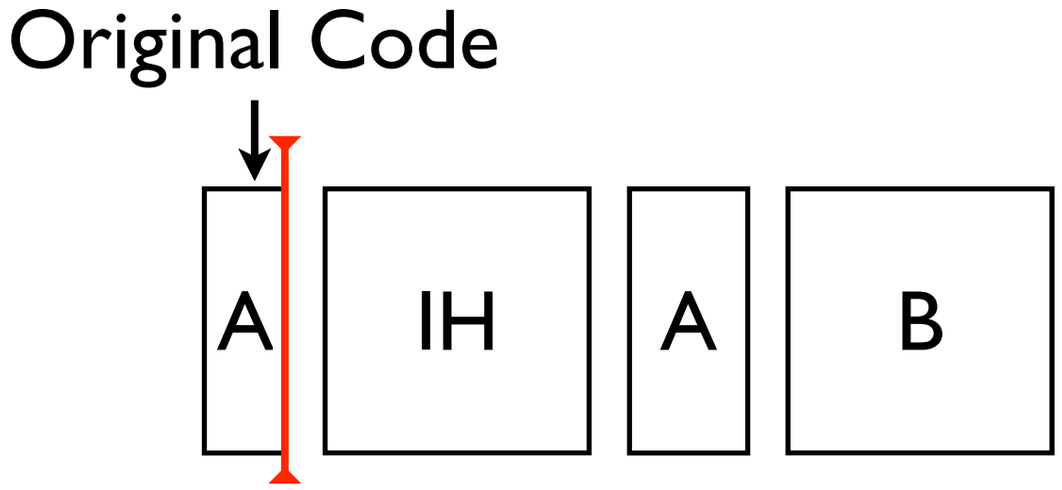


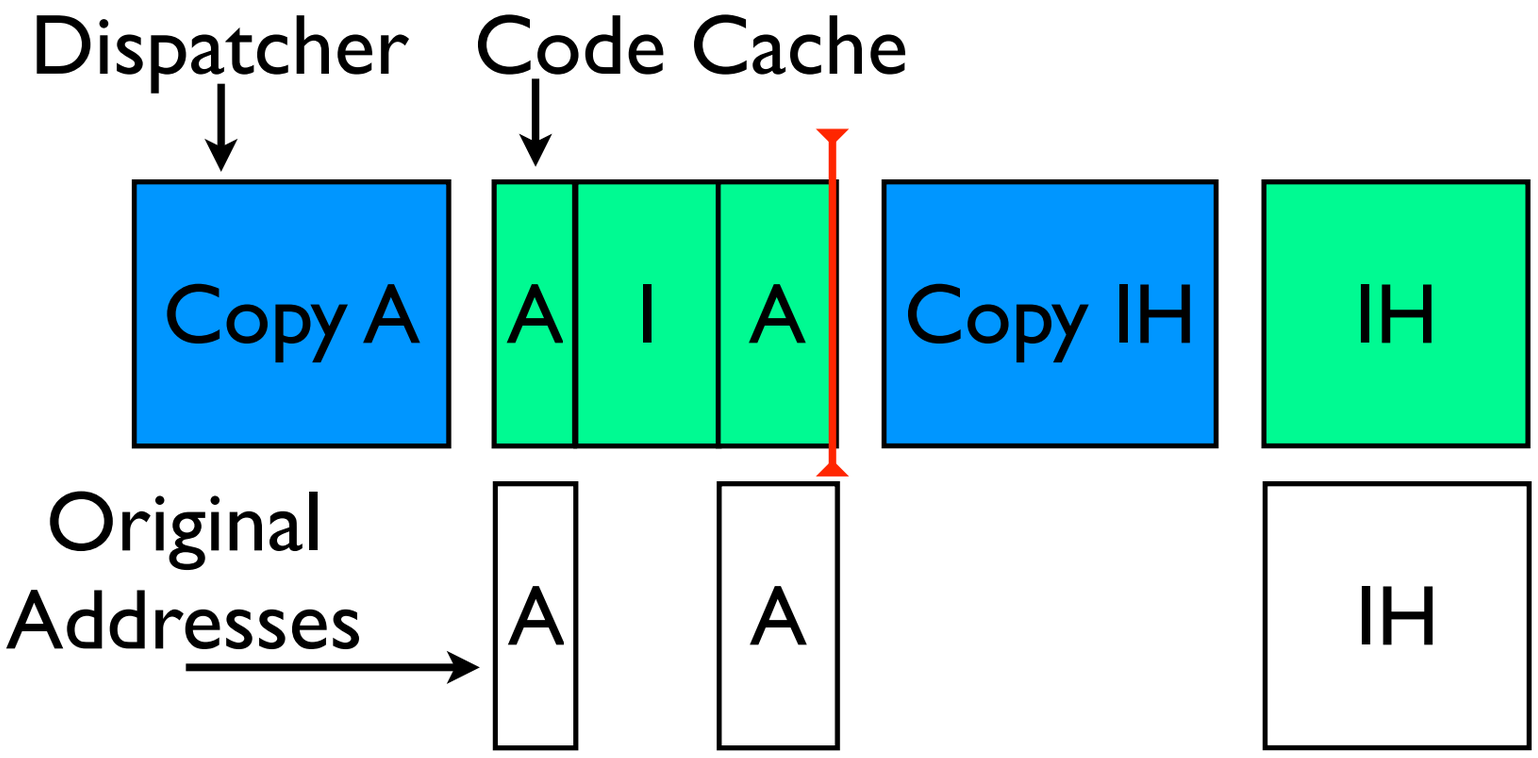


Delay interrupts until next code-cache exit

Interrupt Transparency

H = Interrupt Handler
I = Interrupt
I = Interrupt

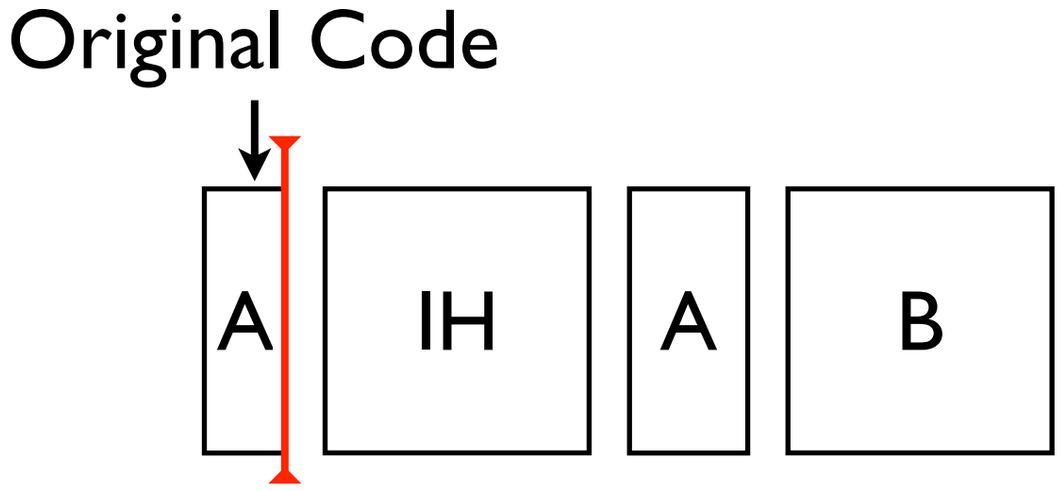


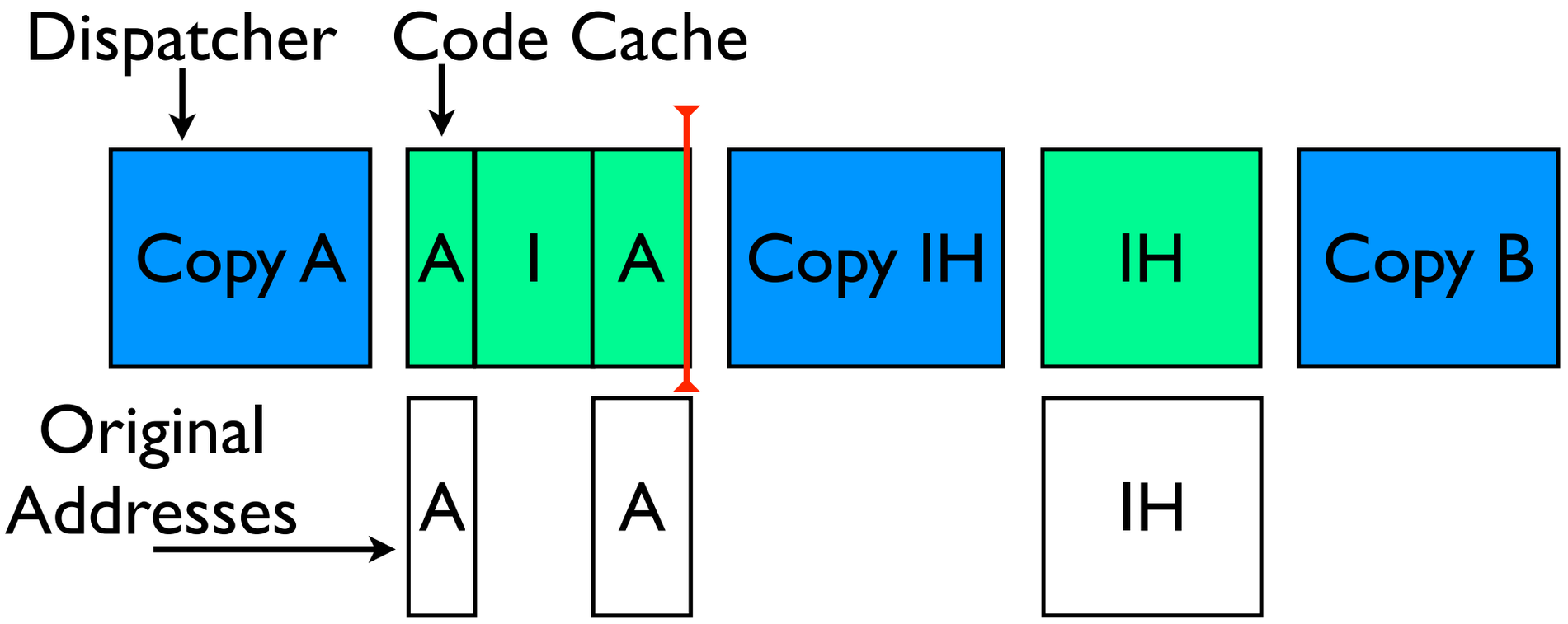


Delay interrupts until next code-cache exit

Interrupt Transparency

H = Interrupt Handler
I = Instrumentation
I = Interrupt

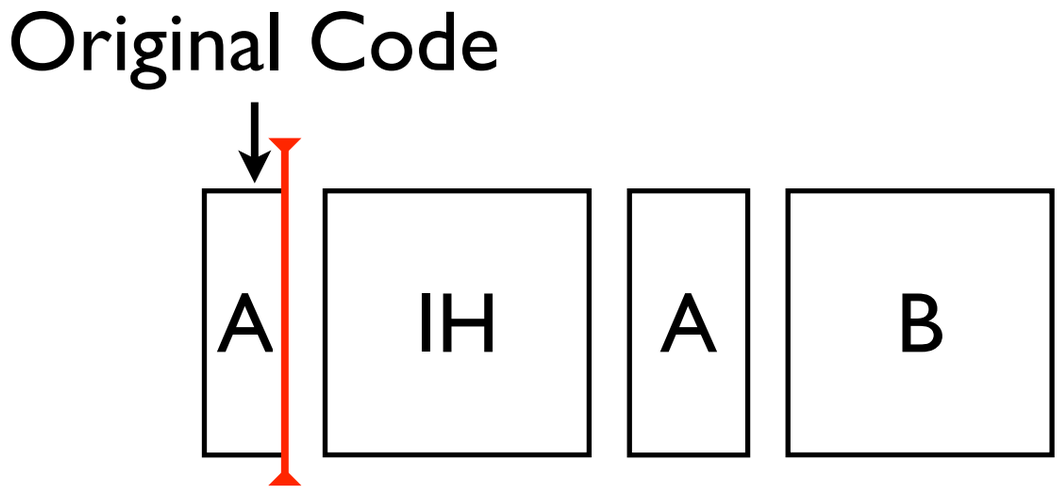


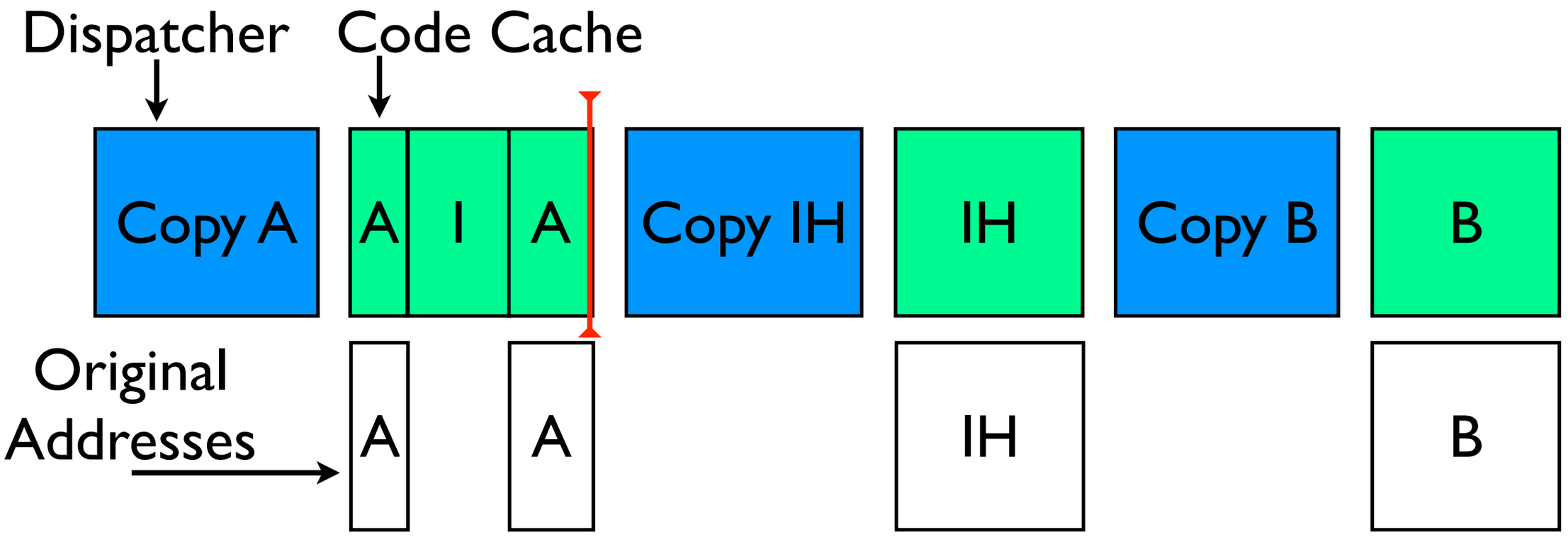


Delay interrupts until next code-cache exit

Interrupt Transparency

H = Interrupt Handler
I = Instrumentation
I = Interrupt





Delay interrupts until next code-cache exit

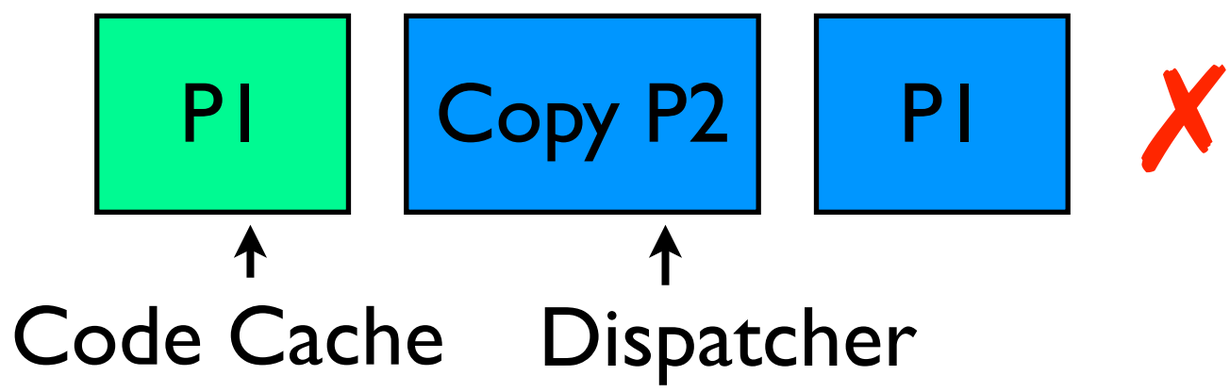
Reentrance

Code is not reentrant if it is unsafe to execute before other executions of the same code finish

- Dispatcher cannot use any non-reentrant OS code, e.g. print, because the non-reentrant code might be currently executing

Say, `print` consists of basic blocks P1, P2

- P1 has executed from code cache
- Dispatcher copies P2
- Dispatcher uses `print` for debugging and invokes P1
- `print` fails because it is non-reentrant



Reentrance Solution

Typical solution is to reimplement non-reentrant code using lower-level uninstrumented code

- e.g., user-level DBI has custom print that makes system calls

OS-level framework has no lower-level code

- Dispatcher must be entirely self sufficient
- Implement our own heap

Some code too difficult to implement from scratch

- Detach and reattach framework to use existing OS code
- Have custom user program make system calls on our behalf
  - Framework cannot depend on user program's correctness

Our Proposal

We chose to port DynamoRIO to a minimal hypervisor because it is

- Open source
- Performance oriented
- Mature

Applications

- Transparent fault isolation
- Dynamic optimization

We will open source our port!

- What would you do with in-kernel DBI?

Backup Slides

Existing Hypervisors

VMWare

- Uses a code cache to translate sensitive instructions
- Does not have an instrumentation API

PinOS

- Pin DBI + Xen Hypervisor
- Does whole-system instrumentation (user + kernel)
- Dispatching is much slower for whole-system (50x slowdown)
- Delegates I/O to a separate uninstrumented VM

Neither is open source

Minimal Hypervisor

Simpler than a full-fledged hypervisor

- No multiplexing
- Shadow page tables have same address mappings, just more restrictive permissions
- Don't need to be completely transparent
  - We can piggy-back on existing OS code, like segment selectors for CPU-private data

Design Assumptions

Once booted, OS runs exclusively in 64-bit long mode

- Emulating obsolete x86 modes would be a pain
- Confirmed validity on Linux by inspection
- We believe it is valid on Windows

Can store dispatcher and code cache in pages that are in all page tables at the same virtual addresses

- Otherwise, we need to steal RAM from the OS at bootup
- Provided by Linux
- We believe this is provided by Windows

Design should work with OS that meets assumptions

- We are currently targeting Linux

Hardware Virtualization Extensions

Do not make implementation simpler

- Removes the need to inspect sensitive instructions
- However, we already can inspect sensitive instructions

Could make implementation more complex

- Need to emulate instructions that cause exits
- Easier for us to emit fix-up code in the code cache

Could improve performance

- Extended page tables might perform better than shadow
- We want to experiment with this