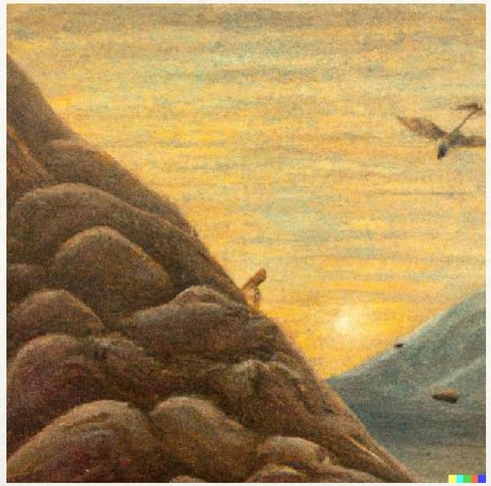


Dynamic Privilege

A Thesis Defense by Thomas Unger

Overview

- Ch 1: Rigidity Prevents Optimization
- Ch 2: Introducing Dynamic Privilege
- Ch 3: Demonstrating Low-Level and OS-Level Access
- Ch 4: Case Studies
- Conclusion

There's a lot more to say here

Overview

- **Ch 1: Rigidity Prevents Optimization**
- Ch 2: Introducing Dynamic Privilege
- Ch 3: Demonstrating Low-Level and OS-Level Access
- Ch 4: Case Studies
- Conclusion

Ch 1: Rigidity Prevents Optimization

- General Purpose Operating Systems (GPOSs) rule the world
- Specialized OSs can outperform GPOSs by making **trade-offs**
- Can we bring some flexibility GPOS?
  - By enabling application specialization

Should We Mind the Gap?

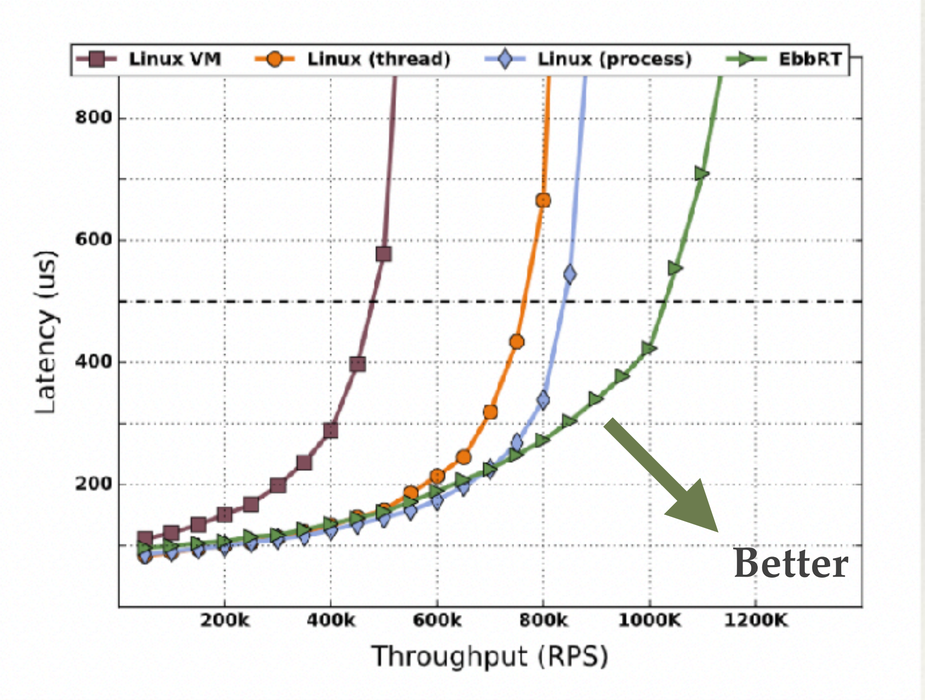
- Is there a gap between what **computing hardware is capable of**, and **what is actually achieved** on modern operating systems?
- Specialized OSs can serve as a lower bound
  - Are we within 5%

From USCC: https://www.uschamber.com/assets/archived/images/ctec_datacenterrpt_lowres.pdf

Should We Mind the Gap?

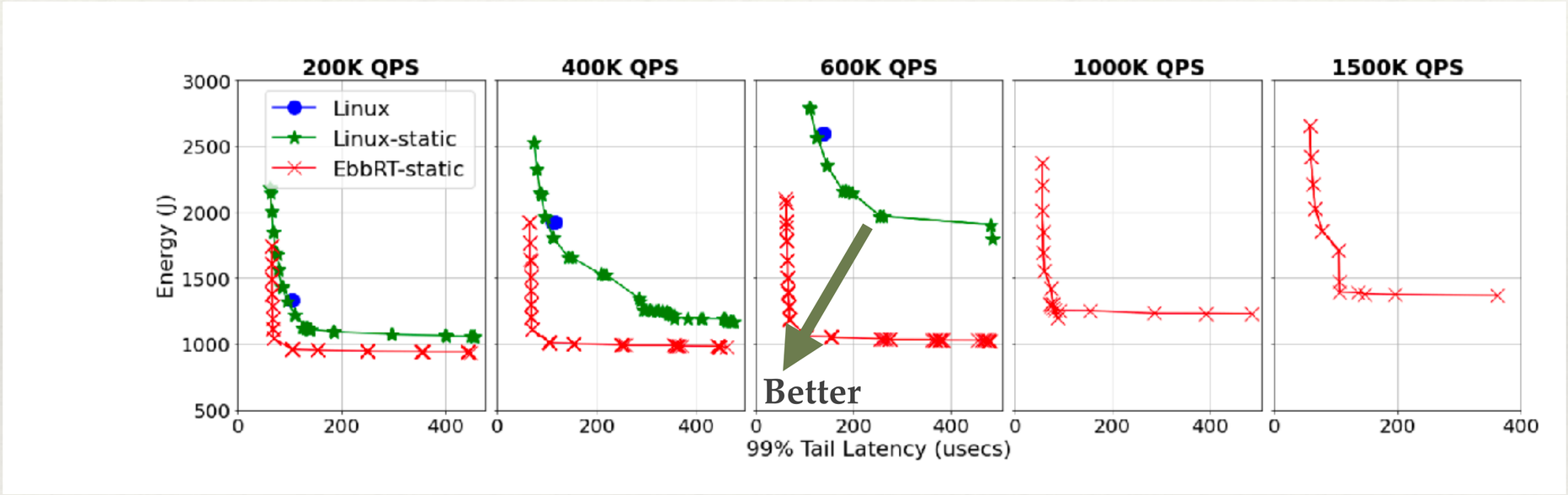
- **Doubled** the throughput of memcached, a heavily optimized networked key-value store
- representative workload
- well tuned baseline

EbbRT: 2.1x better
throughput than Linux on
a multi-core memcached



Should We Mind the Gap?

EbbRT 50% Energy Savings on memcached



Should We Mind the Gap?

- Is there a gap between what **computing hardware is capable of**, and **what is actually achieved** on modern operating systems?
- Specialized OSs can serve as a lower bound
  - Are we within 5%

Rigidity Prevents Tradeoffs

- A **static divide** between application and kernel introduces rigidity
  - Applications face immutable system interfaces, can't trade off
  - Tools only apply to one domain or the other
  - Programmers get siloed

Start with GPOS Whittle Down

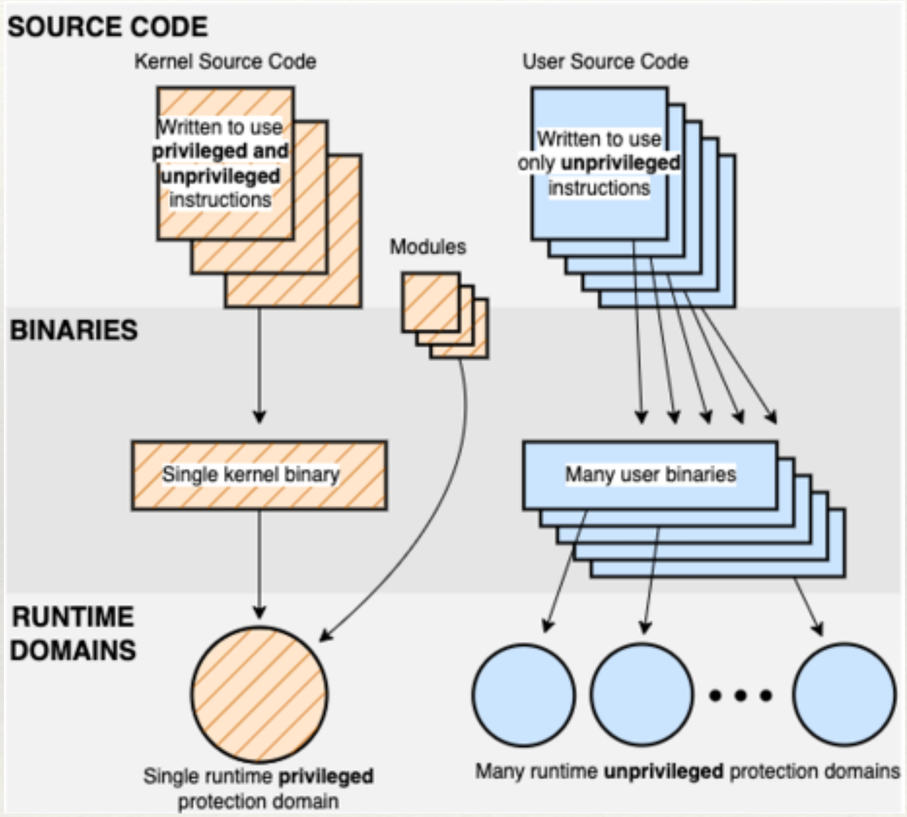
- Start with the full feature application context
- Extend it with OS-Level Access
- Modify the system to meet application requirements

Ch 2: Introducing Dynamic Privilege

- The **static divide**
- It is built from **hardware privilege**
- Dynamic Privilege, a mechanism that orthogonalizes hardware privilege
- A thesis statement

Software is Separated by Privilege

- Applications and kernels are divided at every stage of their lifetimes
  - Separate **code-bases**
  - Independent **binaries**
  - Isolated **runtimes**



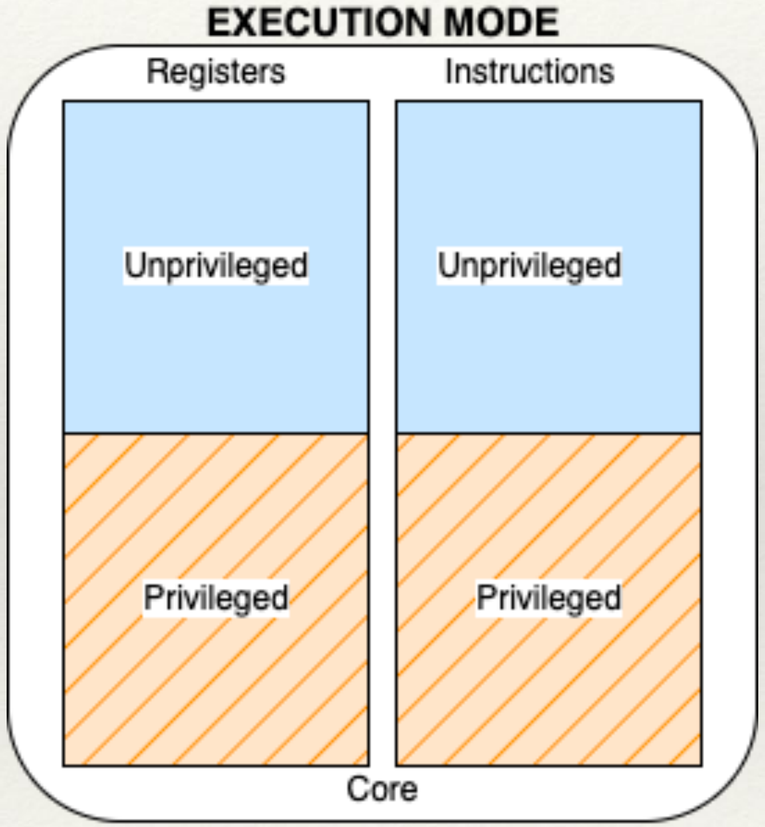
Source to Runtime Spectrum

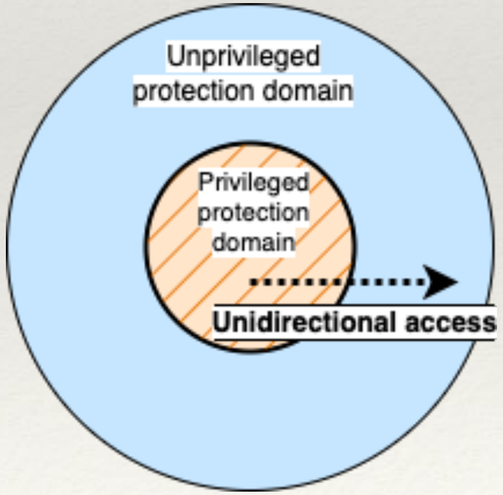
Hardware Privilege

- Processors (Hardware) operate in distinct modes of execution
  - Unprivileged or user-mode
  - Privileged or supervisor mode

Hardware Privilege

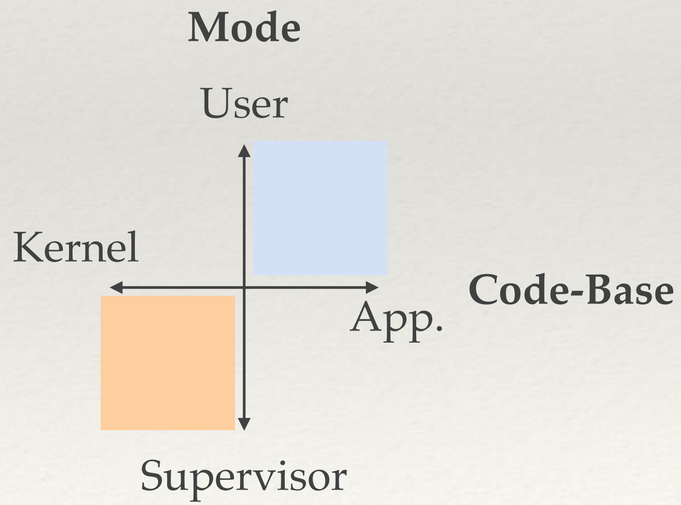
- What can you **access**?
  - Instructions
  - Registers
  - Memory



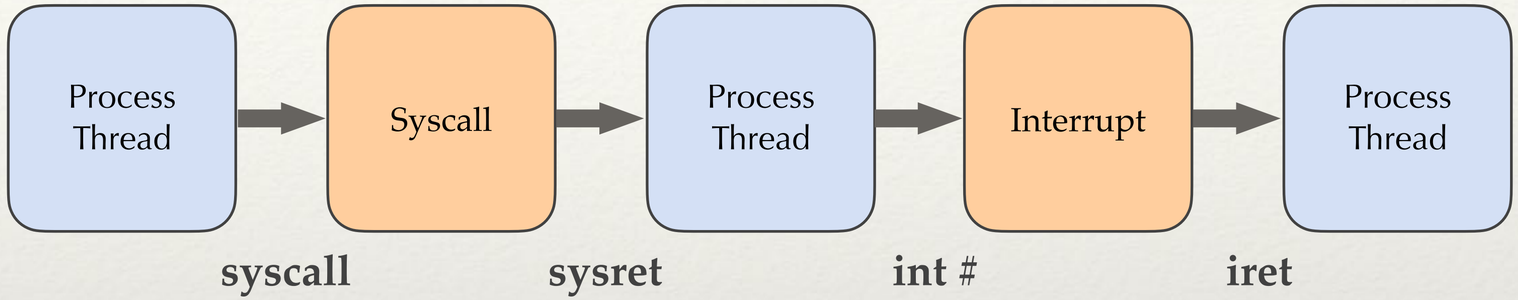


Privilege Separates Execution

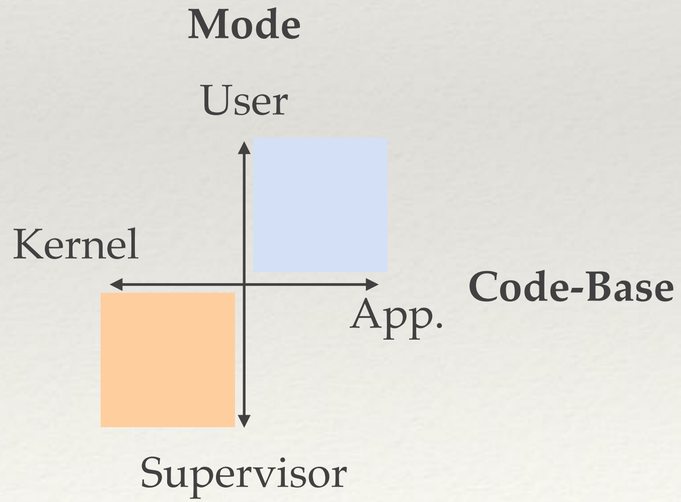
- Coupled transitions
  - Codebase Change <-> Privilege Level Change



Privilege Separates Execution

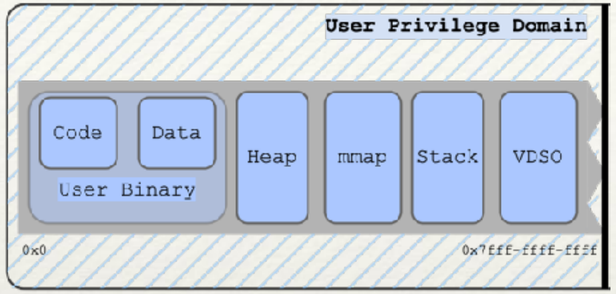


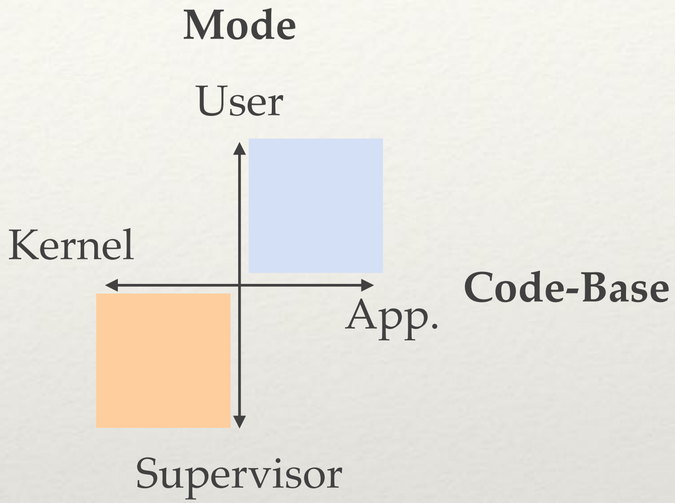
- Coupled transitions
  - Codebase Change <-> Privilege Level Change



Privilege Separates State (Memory)

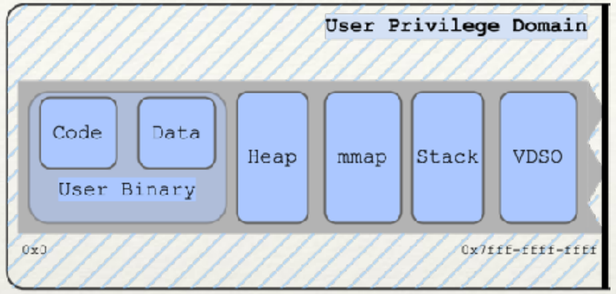
Process

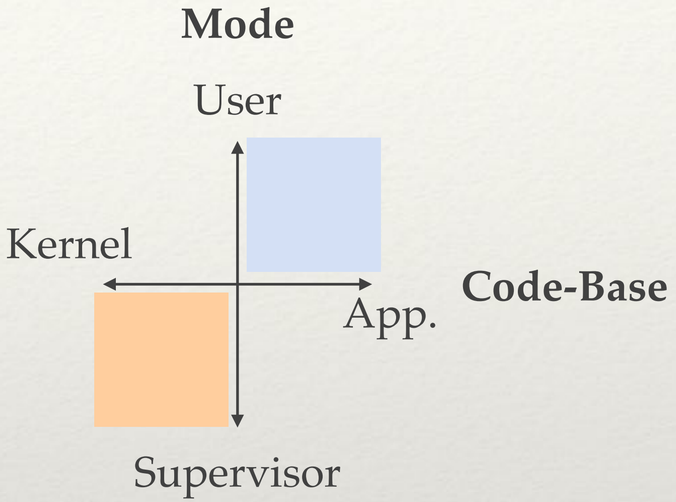




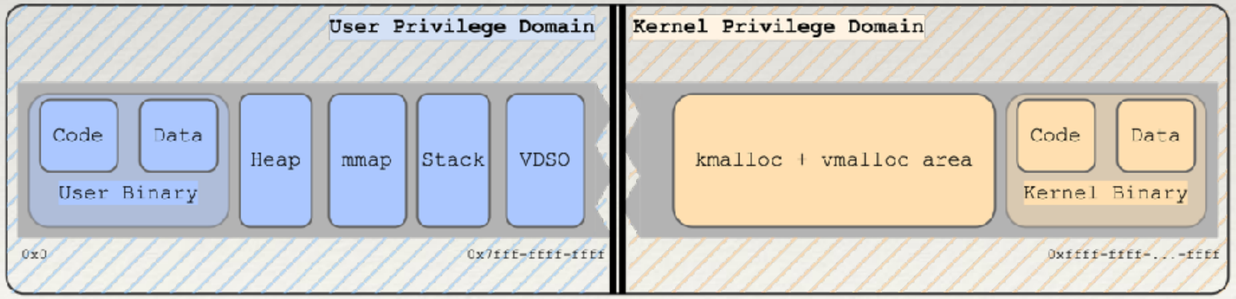
Privilege Separates State (Memory)

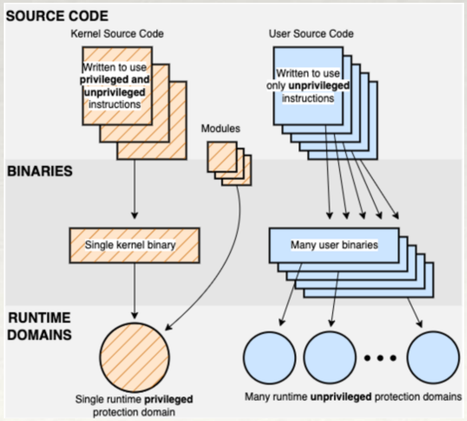
Process



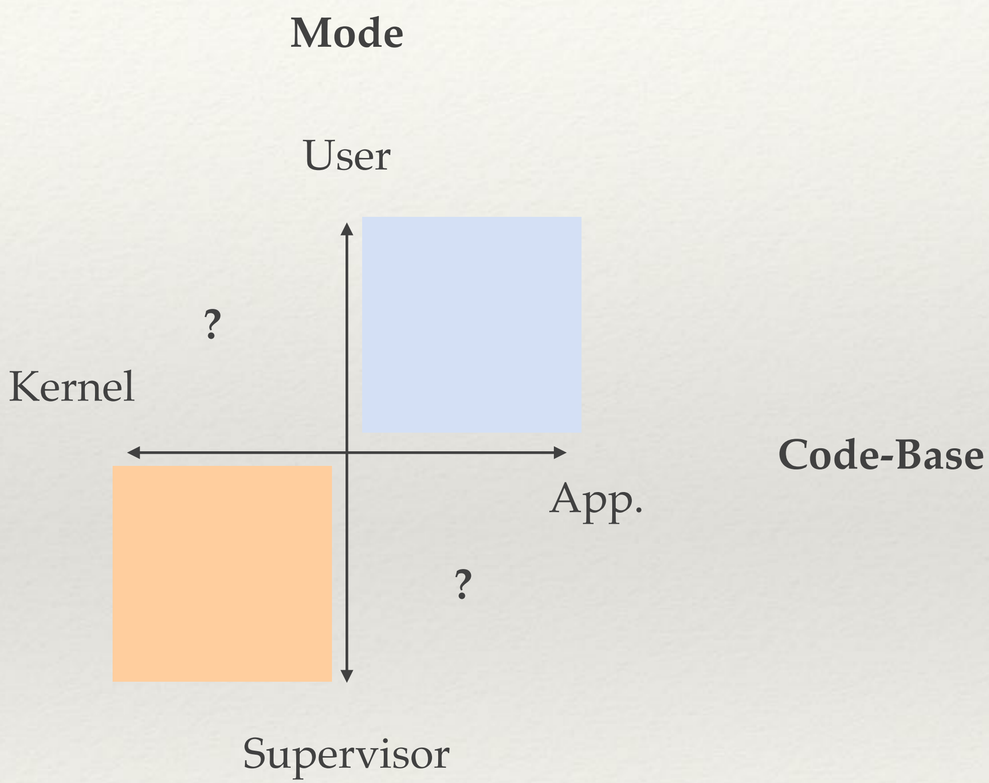


Kernel

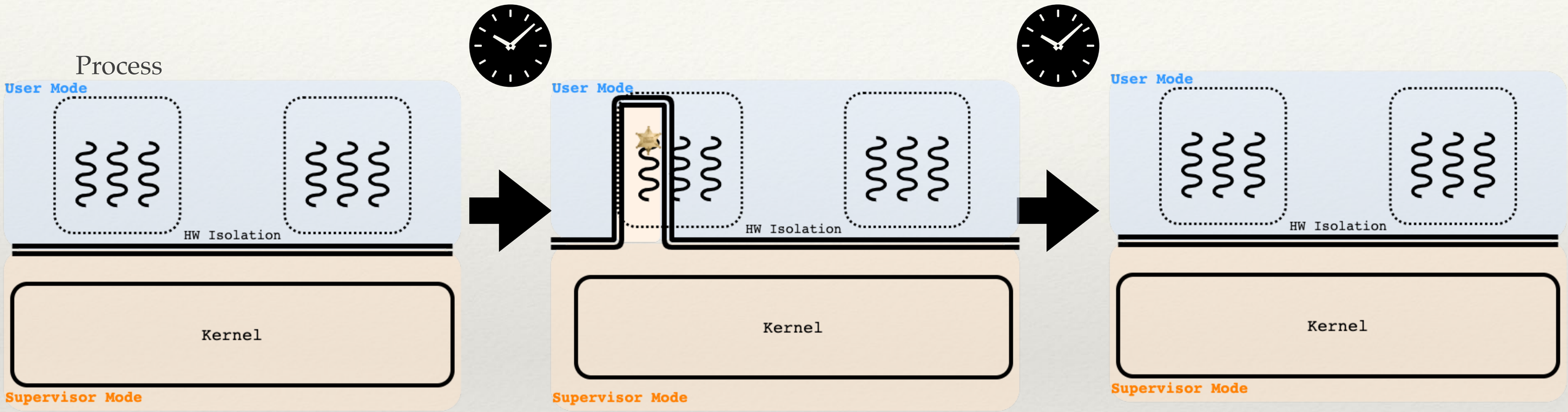




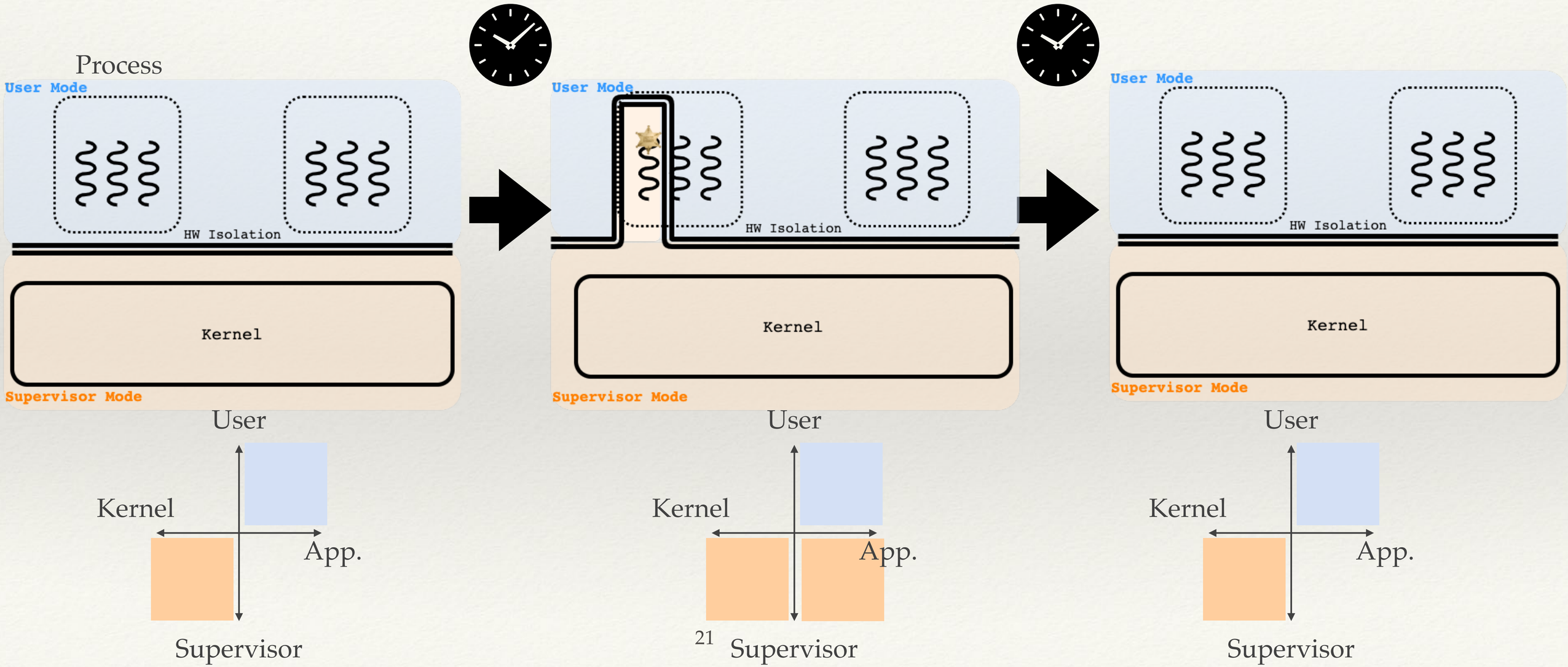
Dynamic Privilege:



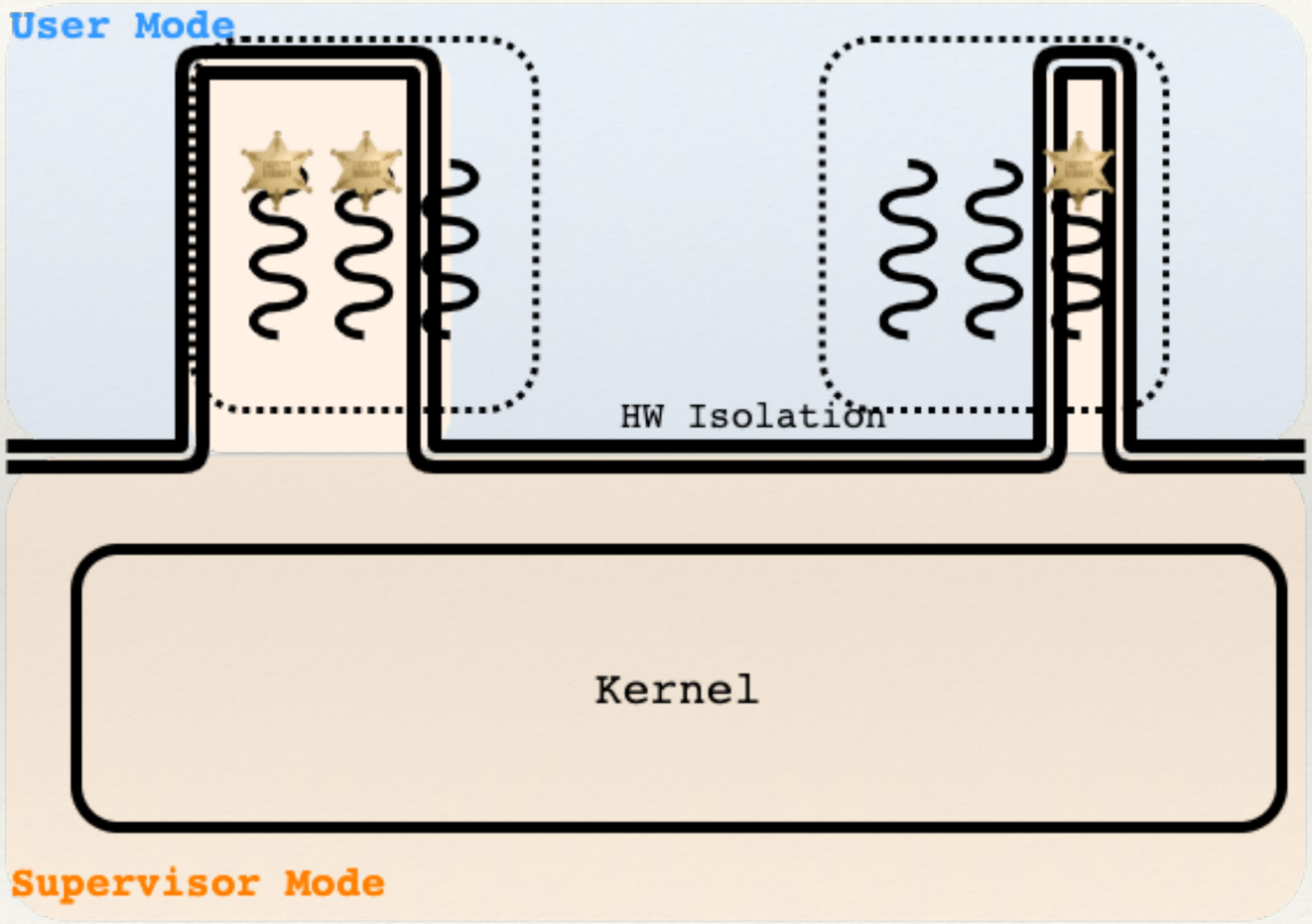
Dynamic Privilege

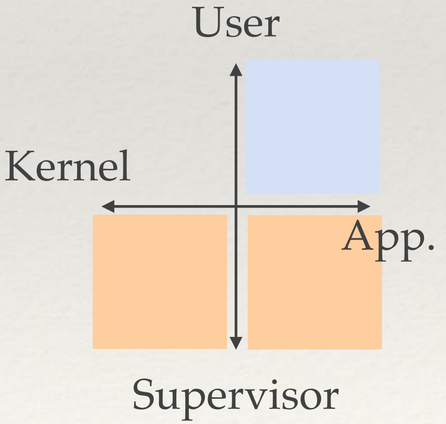


Dynamic Privilege

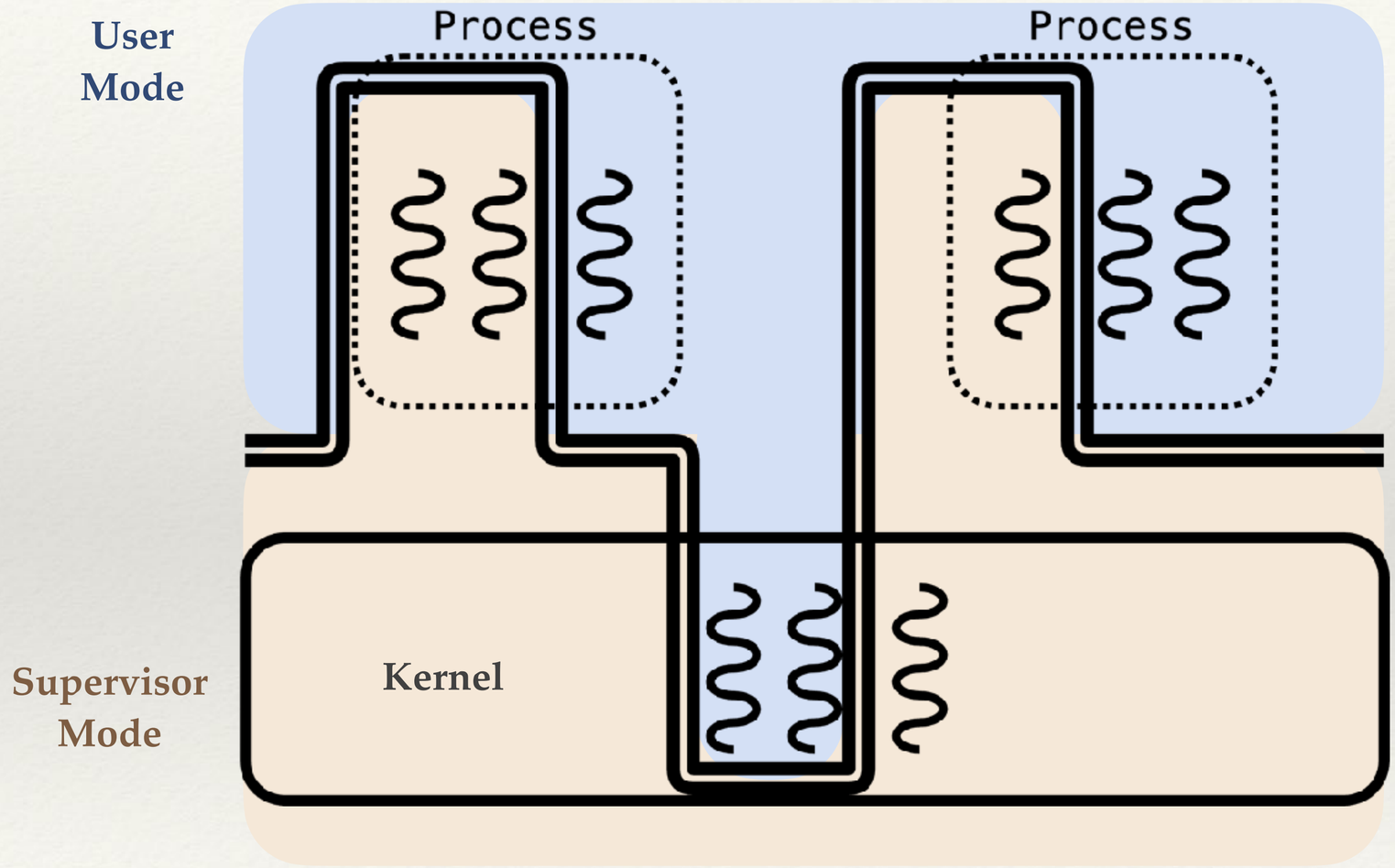


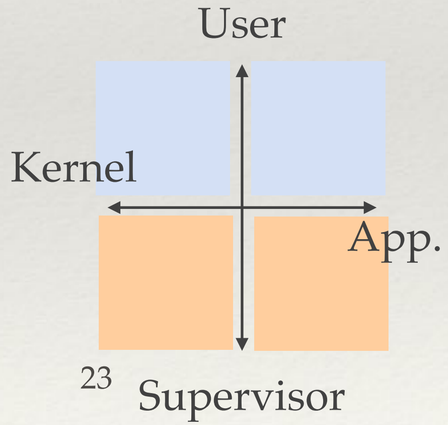
Dynamic Privilege





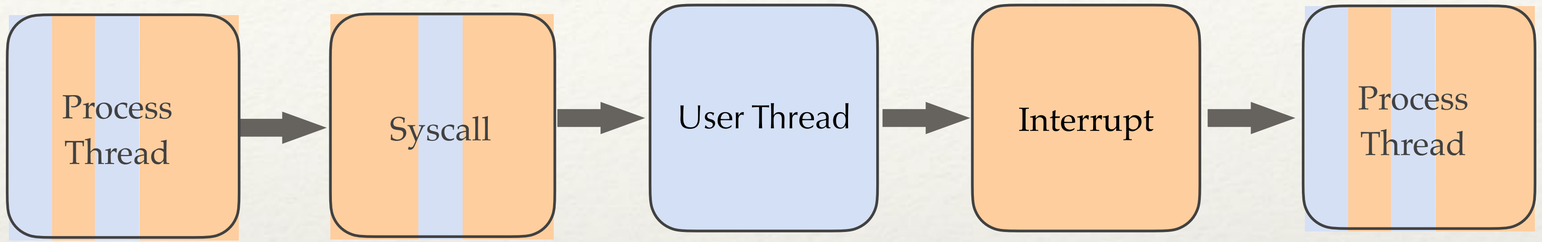
Dynamic Privilege



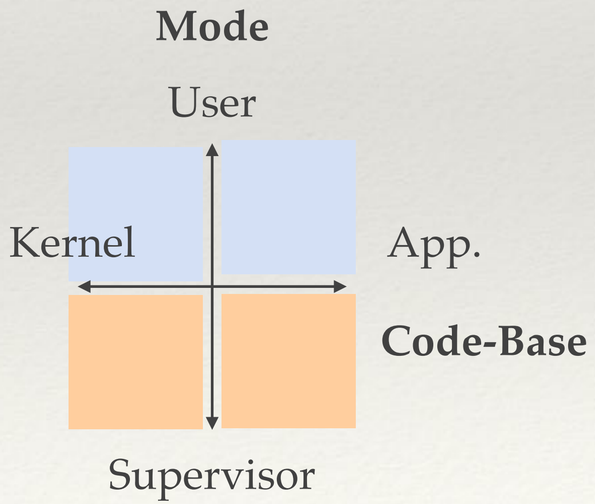


23

Decoupling Mode and Code-Base

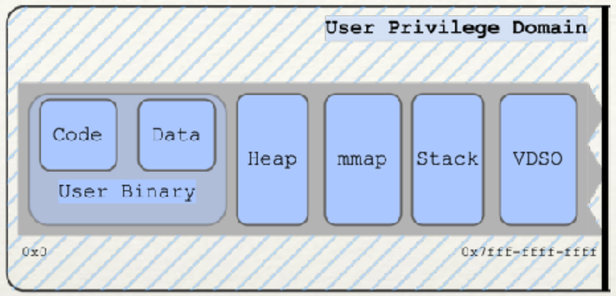


- Decoupled transitions
  - Codebase Change =/= Privilege Level Change

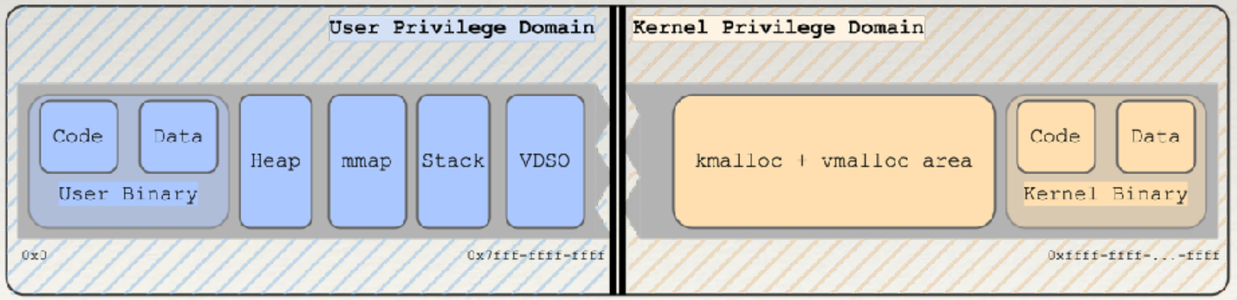


Virtual Address Space (In Memory)

User No Privilege

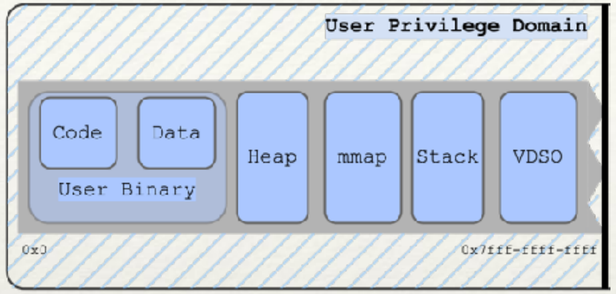


Kernel

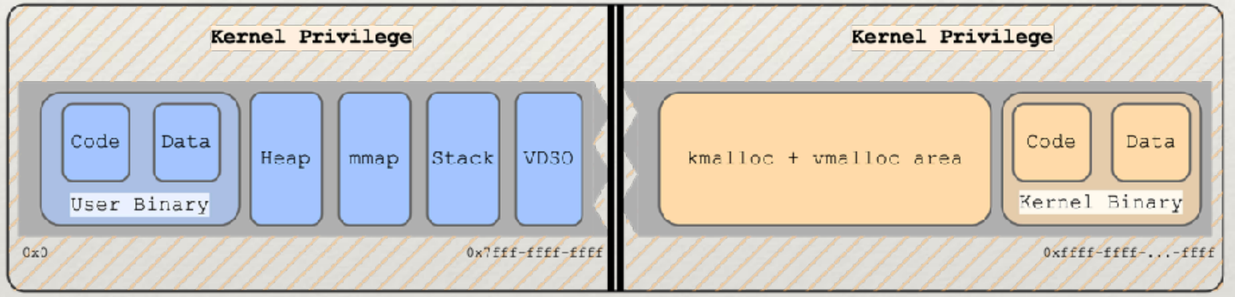


Virtual Address Space (In Memory)

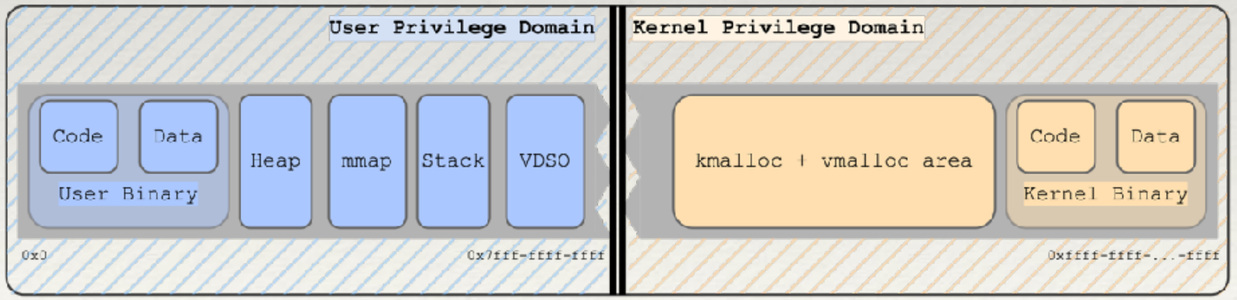
User No Privilege



User With Privilege



Kernel



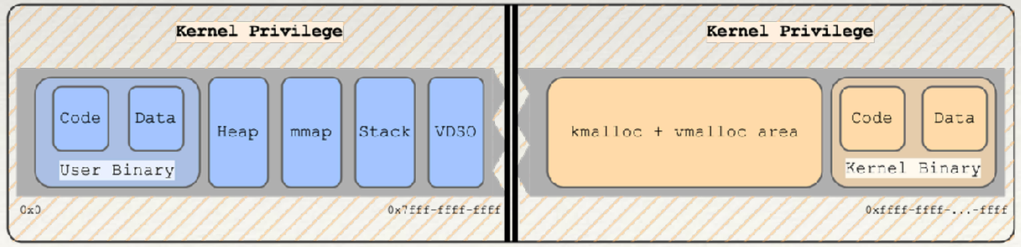
A Definition

Dynamic Privilege: The ability for a unit of execution (e.g. a thread) to independently transition between hardware modes of execution.

A Definition

Dynamic Privilege Attributes:

- Hardware-centric, OS-agnostic
- Address Space Neutral
- Separates Mechanism from Policy



A Definition

Dynamic Priv

- Hardware-centric,
- Address Space Neu
- Separates Mechanism from Policy

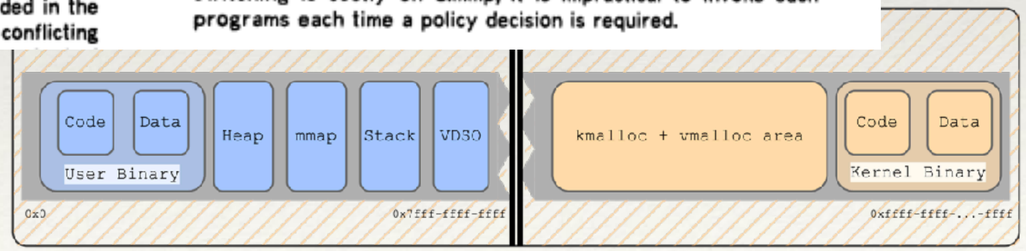
POLICY/MECHANISM SEPARATION IN HYDRA

R. Levin, E. Cohen, W. Corwin, F. Pollack, W. Wulf[1]
Carnegie-Mellon University
Pittsburgh, Pa.

Abstract

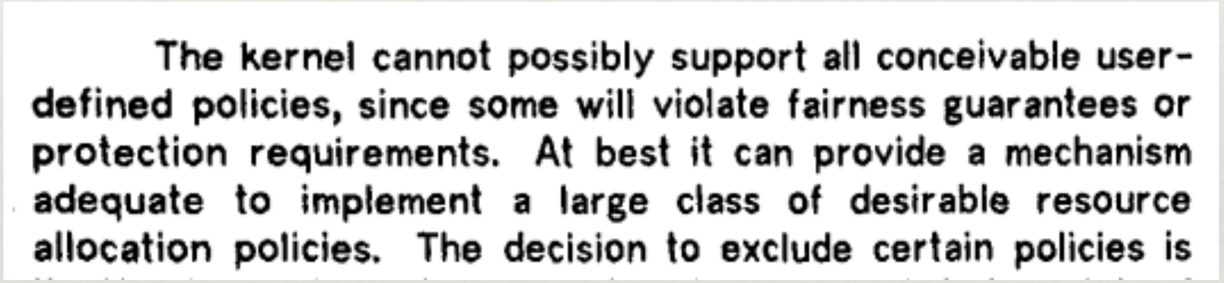
The extent to which resource allocation policies are entrusted to user-level software determines in large part the degree of flexibility present in an operating system. In Hydra the determination to separate mechanism and policy is established as a basic design principle and is implemented by the construction of a kernel composed (almost) entirely of mechanisms. This paper presents three such mechanisms (scheduling, paging, protection) and examines how external policies which manipulate them may be constructed. It is shown that the policy decisions which remain embedded in the kernel exist for the sole purpose of arbitrating conflicting

implication, we must prevent programs which define policies direct access to hardware or data which could be (mis)used to destroy another program. That is, such programs must execute in a protected environment.[2] Further, we must not permit such a program to monopolize any resource, whether it does so intentionally or not. We must assure some "fairness" among competing policies. In addition, we must recognize that many policy decisions must be made rapidly (e.g. fast scheduling decisions are essential in order to achieve reasonable response). Given that user-level policy programs must execute in their own protection domains, and that domain switching is costly on C.mmp, it is impractical to invoke such programs each time a policy decision is required.



Policy/Mechanism Separation

Policy / Mechanism Separation in Hydra

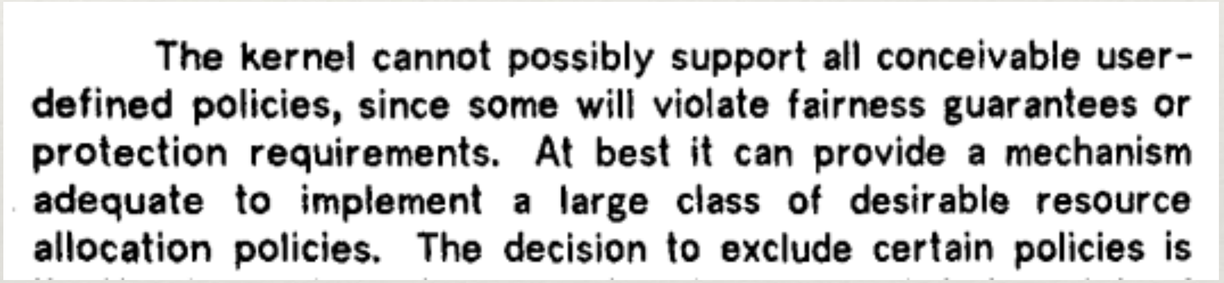


The kernel cannot possibly support all conceivable user-defined policies, since some will violate fairness guarantees or protection requirements. At best it can provide a mechanism adequate to implement a large class of desirable resource allocation policies. The decision to exclude certain policies is

Policy/Mechanism Separation

- Dynamic Privilege could be used to implement
  - Static Privilege
  - Just-in-Time Privilege
  - On-Demand Privilege
  - Lazy Privilege

Policy / Mechanism Separation in Hydra



The kernel cannot possibly support all conceivable user-defined policies, since some will violate fairness guarantees or protection requirements. At best it can provide a mechanism adequate to implement a large class of desirable resource allocation policies. The decision to exclude certain policies is

Implementing Dynamic Privilege

- Many options
  - Add a new instruction to hardware
  - Implemented a new OS
  - **Enabling this in an existing OS**

Intersecting Research

Kernel architecture

Monolith
Microkernel
Exokernel
Anykernel
Unikernel
SAOSs

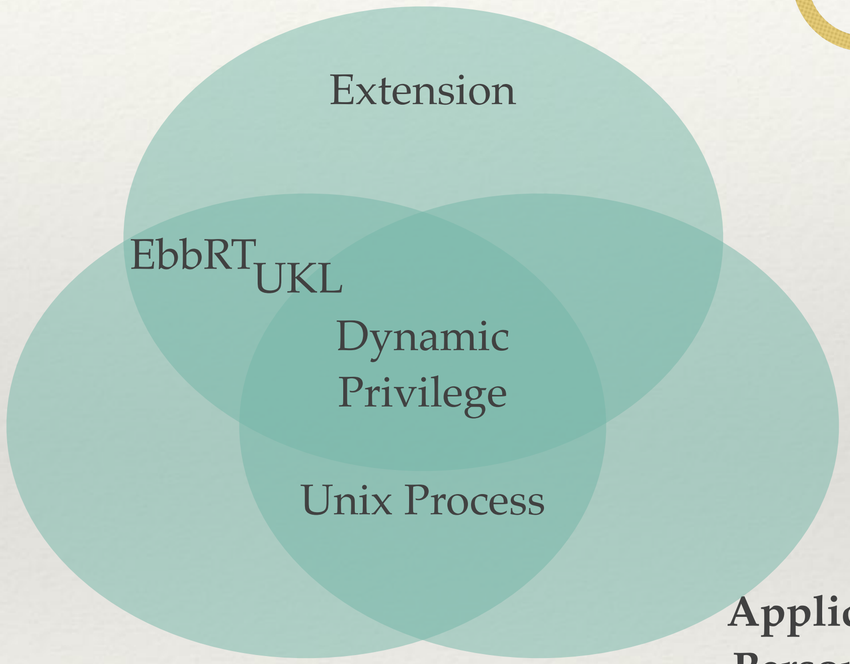
Existing OS Mechanisms

Kernel Modules
FUSE
VMMs
e-BPF

Control Flow

Capabilities

Mungi
Grasshopper
Protected Shared Libraries
Capability Hardware Enhanced RISC Instructions (CHERI)



Extension

K42
Vino
Code Downloading
Software Fault Isolation
Hardware Transactions
OO OSs

Playing with privilege

Kernel Mode Linux
Dune
Privbox
Lupine
UKL

Thesis Statement

We introduce Dynamic Privilege, an OS model enabling threads to adjust their hardware privilege levels on the fly, challenging the traditional separation between kernel and application.

Dynamic Privilege provides granular control and requires only minor modifications to support it in an existing OS.

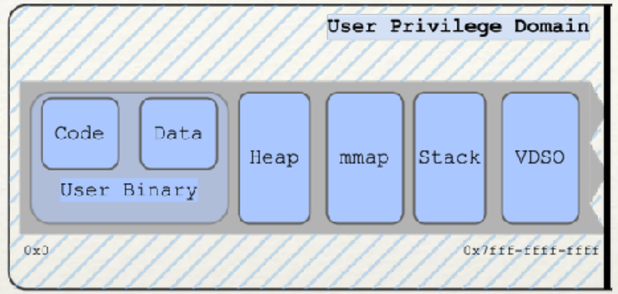
Further, Dynamic Privilege enables new system evolution and optimization while maintaining compatibility with existing software ecosystems.

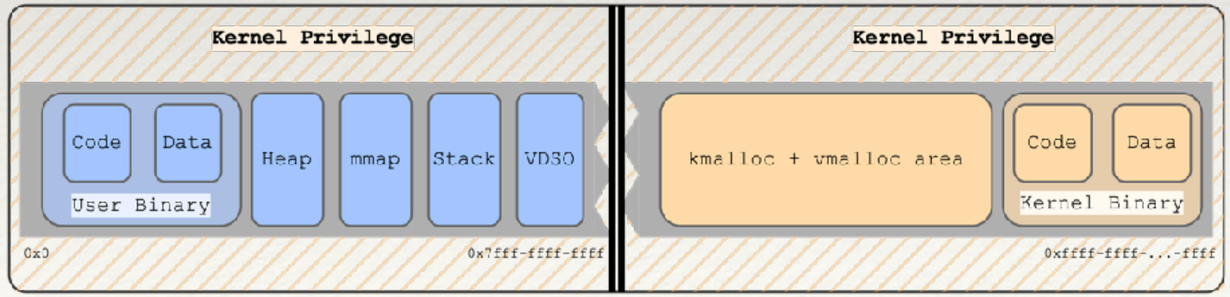
Ch 3: Demonstrating Low-Level and OS-Level Access

- Implementing Dynamic Privilege as system call, kElevate
- Demonstrating Access
  - Low-Level (Hardware)
  - Runtime OS (Symbol Table)
  - Compile Time OS (Macros / Types / Inline-Functions)

Ch 3: Demonstrating Low-Level and OS-Level Access

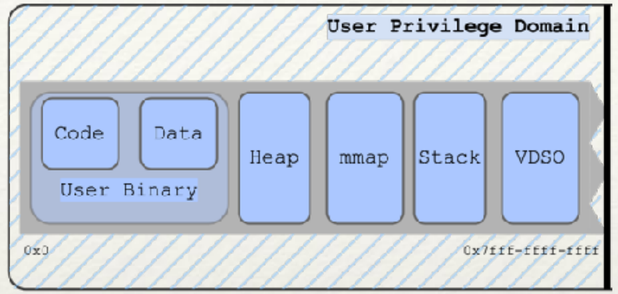
- Implementing Dynamic Privilege as system call, kElevate
- Demonstrating Access
  - Low-Level (Hardware)
  - Runtime OS (Symbol Table)
  - Compile Time OS (Macros / Types / Inline-Functions)

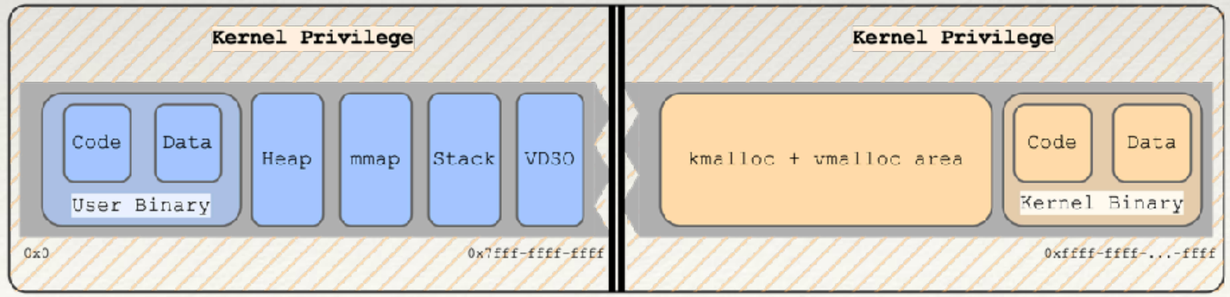




Ch 3: Demonstrating Low-Level and OS-Level Access

- Implementing Dynamic Privilege as system call, kElevate
- Demonstrating Access
  - Low-Level (Hardware)
  - Runtime OS (Symbol Table)
  - Compile Time OS (Macros / Types / Inline-Functions)





Quick Implementation

- While this could be implemented in hardware…
- Until then, we'll emulate it with a system call
- kElevate() is the name of our syscall, the **mechanism** we use to implement Dynamic Privilege
- It's about as fast as our lowest latency syscall getppid() ~100ns

```
static long do_kElevate(int target_mode){
  return syscall(NR_ELEVATE_SYSCALL, target_mode);
}

long k_elevate(){
  return do_kElevate( SYM_ELEVATE_FLAG );
}

long k_lower(){
  return do_kElevate( SYM_LOWER_FLAG );
}

long sym_check_elevate(){
  return do_kElevate( SYM_QUERY_FLAG);
}
```

Using kElevate to access and relinquish hardware privilege

Quick Implementation

- The entire kElevate patch for Linux (x86_64) is 173 lines of new kernel code.
  - What does this really enable?
  - Adaptors
- Developed for the x86_64 architecture
- ARM64 prototype

```c
static long do_kElevate(int target_mode){
  return syscall(NR_ELEVATE_SYSCALL, target_mode);
}

long k_elevate(){
  return do_kElevate( SYM_ELEVATE_FLAG );
}

long k_lower(){
  return do_kElevate( SYM_LOWER_FLAG );
}

long sym_check_elevate(){
  return do_kElevate( SYM_QUERY_FLAG);
}
```

Using kElevate to access and relinquish hardware privilege

Similar to Kernel Modules Security Model

- The only thing protecting system calls from unprivileged users is a check w/ the capability system
  - systemctl, mount, insmod
- We introduce a use a very similar check to determine if the executable has the corresponding capability

```
if (!capable(CAP_SYS_ADMIN))
    return -EPERM;
```

```
1095        struct SymbiReg sreg;
1096        sreg.raw = flags;
1097
1098        if (!capable(CAP_SYMBI_ELEV))
1099            return -EPERM;
1100
```

kElevate, the Gory Details

- Kernel Changes:
  - Add system call kElevate()
  - Add kernel config KELEVATE
  - Add conditional system call return path
    - Modified IRET frame to return with privilege
    - Alternative RET implementation
  - Add 2 bits to the (per thread) task_struct
    - Elevated / Migrated
  - Migration path
    - Update GS register to new CPU
- Boot Arguments:
  - No SMEP / No SMAP / No KASLR

```c
static long do_kElevate(int target_mode){
  return syscall(NR_ELEVATE_SYSCALL, target_mode);
}

long k_elevate(){
  return do_kElevate( SYM_ELEVATE_FLAG );
}

long k_lower(){
  return do_kElevate( SYM_LOWER_FLAG );
}

long sym_check_elevate(){
  return do_kElevate( SYM_QUERY_FLAG);
}
```

Using kElevate to access and relinquish hardware privilege

Low-Level Access

- With Dynamic Privilege, all we get is hardware access
  - Bracketing a thread's privilege access in time
  - Accessing / Executing:
    - instructions
    - registers
    - memory

Low-Level Access

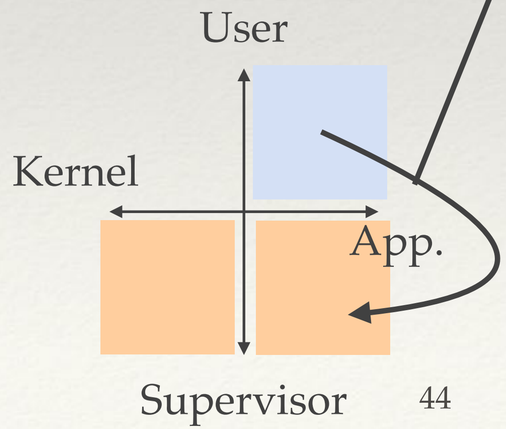
- Bracketing privilege access in time
- Accessing a privileged register
  - x86_64 Control Register 3
- Executing privileged instructions
  - mov cr3, %rax

```
priv = loadMod(priv_lib_path)
kele = loadMod(kele_lib_path)
kele.kElevate()
cr3 = priv.getcr3()
kele.kLower()
print(f"CR3 = 0x{cr3:x}")
```

```
CR3 = 0x2e868002
```

Low-Level Access

- Bracketing privilege access in time
- Accessing a privileged register
  - x86_64 Control Register 3
- Executing privileged instructions
  - mov cr3, %rax




```
priv = loadMod(priv_lib_path)
kele = loadMod(kele_lib_path)
kele.kElevate()
cr3 = priv.getcr3()
kele.kLower()
print(f"CR3 = 0x{cr3:x}")
```

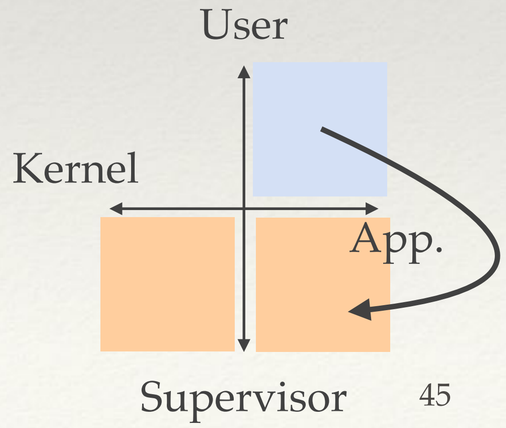
```
CR3 = 0x2e868002
```

Low-Level Access

- Bracketing privilege access in time
- Accessing a privileged register
  - x86_64 Control Register 3
- Executing privileged instructions
  - mov cr3, %rax

```
priv = loadMod(priv_lib_path)
kele = loadMod(kele_lib_path)
kele.kElevate()
cr3 = priv.getcr3()
kele.kLower()
print(f"CR3 = 0x{cr3:x}")
```

```
CR3 = 0x2e868002
```

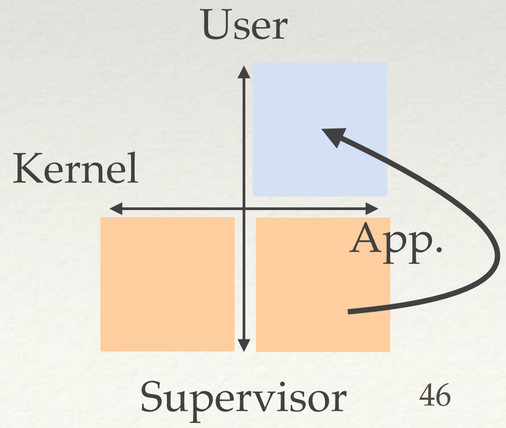


Low-Level Access

- Bracketing privilege access in time
- Accessing a privileged register
  - x86_64 Control Register 3
- Executing privileged instructions
  - mov cr3, %rax

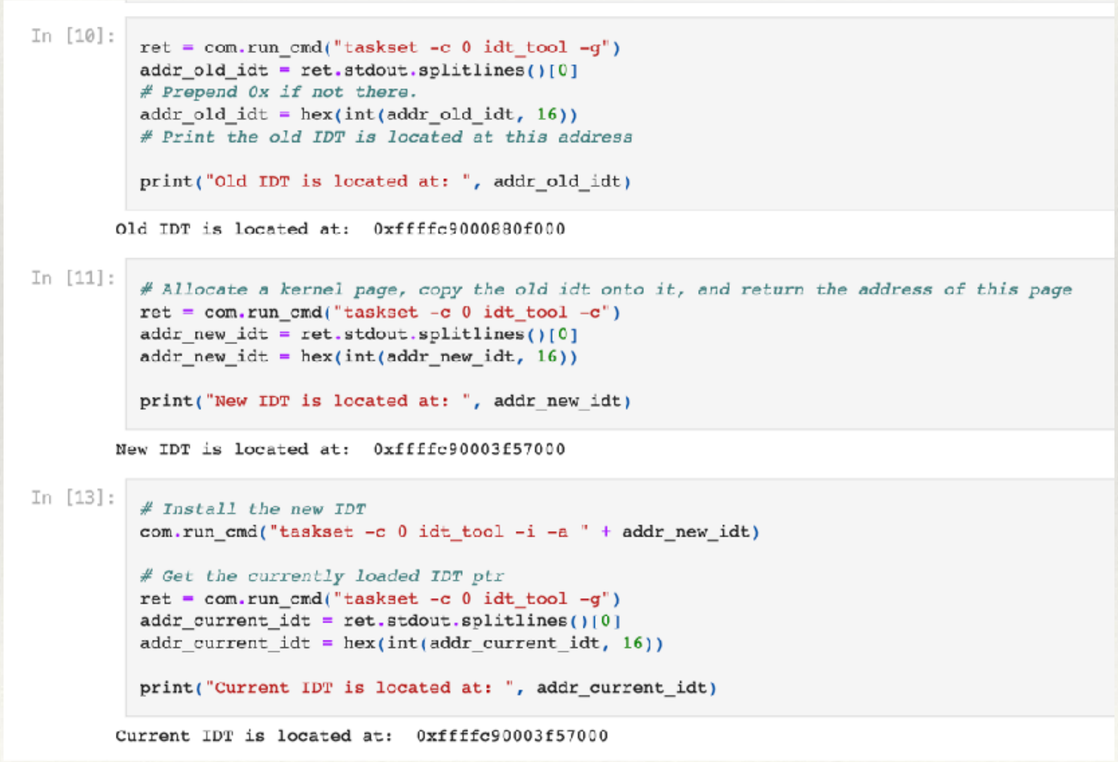
```
priv = loadMod(priv_lib_path)
kele = loadMod(kele_lib_path)
kele.kElevate()
cr3 = priv.getcr3()
kele.kLower()
print(f"CR3 = 0x{cr3:x}")
```

```
CR3 = 0x2e868002
```



Dynamic Privilege from Anywhere

- We've mainly prototyped in C, C++, and Python, but have written prototypes to demonstrate kElevate can be used in others
  - Java
  - Node
  - Rust
  - Go
  - Assembly

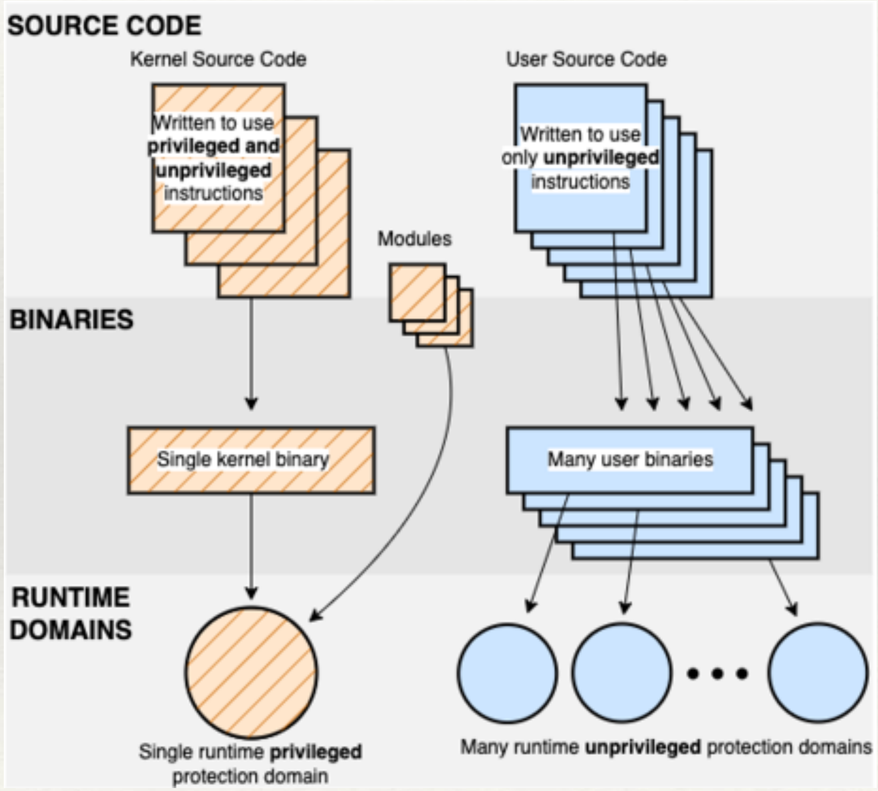


```
In [10]:
ret = com.run_cmd("taskset -c 0 idt_tool -g")
addr_old_idt = ret.stdout.splitlines()[0]
# Prepend 0x if not there.
addr_old_idt = hex(int(addr_old_idt, 16))
# Print the old IDT is located at this address

print("Old IDT is located at: ", addr_old_idt)

Old IDT is located at:  0xffffc9000880f000

In [11]:
# Allocate a kernel page, copy the old idt onto it, and return the address of this page
ret = com.run_cmd("taskset -c 0 idt_tool -c")
addr_new_idt = ret.stdout.splitlines()[0]
addr_new_idt = hex(int(addr_new_idt, 16))

print("New IDT is located at: ", addr_new_idt)

New IDT is located at:  0xffffc90003f57000

In [13]:
# Install the new IDT
com.run_cmd("taskset -c 0 idt_tool -i -a " + addr_new_idt)

# Get the currently loaded IDT ptr
ret = com.run_cmd("taskset -c 0 idt_tool -g")
addr_current_idt = ret.stdout.splitlines()[0]
addr_current_idt = hex(int(addr_current_idt, 16))

print("Current IDT is located at: ", addr_current_idt)

Current IDT is located at:  0xffffc90003f57000
```

Who says you can't hack the kernel from a Jupyter Notebook

OS-Level Access

- What kernel developers need to get their work done
  - Runtime
    - Executing standalone kernel functions
    - Accessing kernel data



Runtime OS Access

- Intermixing user and kernel codepaths!

```
with open("output.txt", "w") as f:
    # Standard write.
    f.write("Hello ")

    kele.kElevate()
    # Invoking kernel's ksys_write directly.
    kele.ksys_write(f.fileno(), "World!", 7)
    kele.kLower()
```

```
# Open the file and read the contents
with open("output.txt", "r") as f:
    print(f.read())
```

```
Hello world!
```

Runtime OS Access

- Intermixing user and kernel codepaths!

```
with open("output.txt", "w") as f:
    # Standard write.
    f.write("Hello ")

    kele.kElevate()
    # Invoking kernel's ksys_write directly.
    kele.ksys_write(f.fileno(), "World!", 7)
    kele.kLower()
```

```
# Open the file and read the contents
with open("output.txt", "r") as f:
    print(f.read())
```

```
Hello world!
```

Runtime OS Access

- Intermixing user and kernel codepaths!

```
with open("output.txt", "w") as f:
    # Standard write.
    f.write("Hello ")

    kele.kElevate()
    # Invoking kernel's ksys_write directly.
    kele.ksys_write(f.fileno(), "World!", 7)
    kele.kLower()
```

```
# Open the file and read the contents
with open("output.txt", "r") as f:
    print(f.read())
```

```
Hello world!
```

Runtime OS Access

- Intermixing user and kernel codepaths!
- Location of ksys_write() known by kallsyms subsystem (or the symbol table)

```
with open("output.txt", "w") as f:
    # Standard write.
    f.write("Hello ")

    kele.kElevate()
    # Invoking kernel's ksys_write directly.
    kele.ksys_write(f.fileno(), "World!", 7)
    kele.kLower()
```

```
# Open the file and read the contents
with open("output.txt", "r") as f:
    print(f.read())
```

```
Hello world!
```

Runtime OS Access

- Intermixing user and kernel codepaths!

```
with open("output.txt", "w") as f:
    # Standard write.
    f.write("Hello ")

    kele.kElevate()
    # Invoking kernel's ksys_write directly.
    kele.ksys_write(f.fileno(), "World!", 7)
    kele.kLower()
```

```
# Open the file and read the contents
with open("output.txt", "r") as f:
    print(f.read())
```

```
Hello world!
```

Runtime OS Access

- Intermixing user and kernel codepaths!

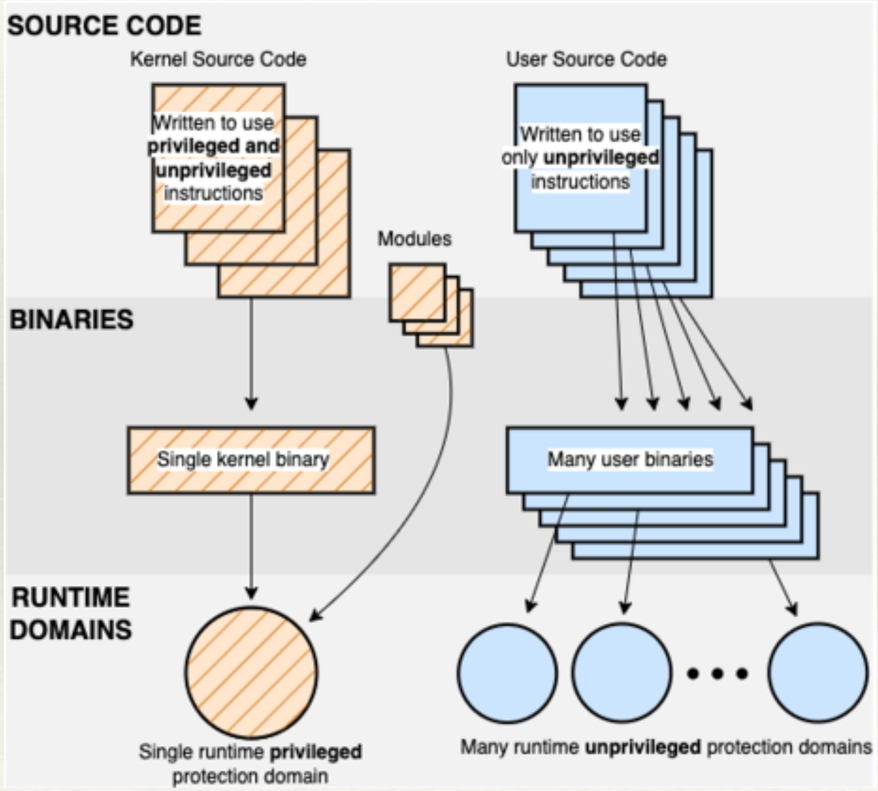
```
with open("output.txt", "w") as f:
    # Standard write.
    f.write("Hello ")

    kele.kElevate()
    # Invoking kernel's ksys_write directly.
    kele.ksys_write(f.fileno(), "World!", 7)
    kele.kLower()
```

```
# Open the file and read the contents
with open("output.txt", "r") as f:
    print(f.read())
```

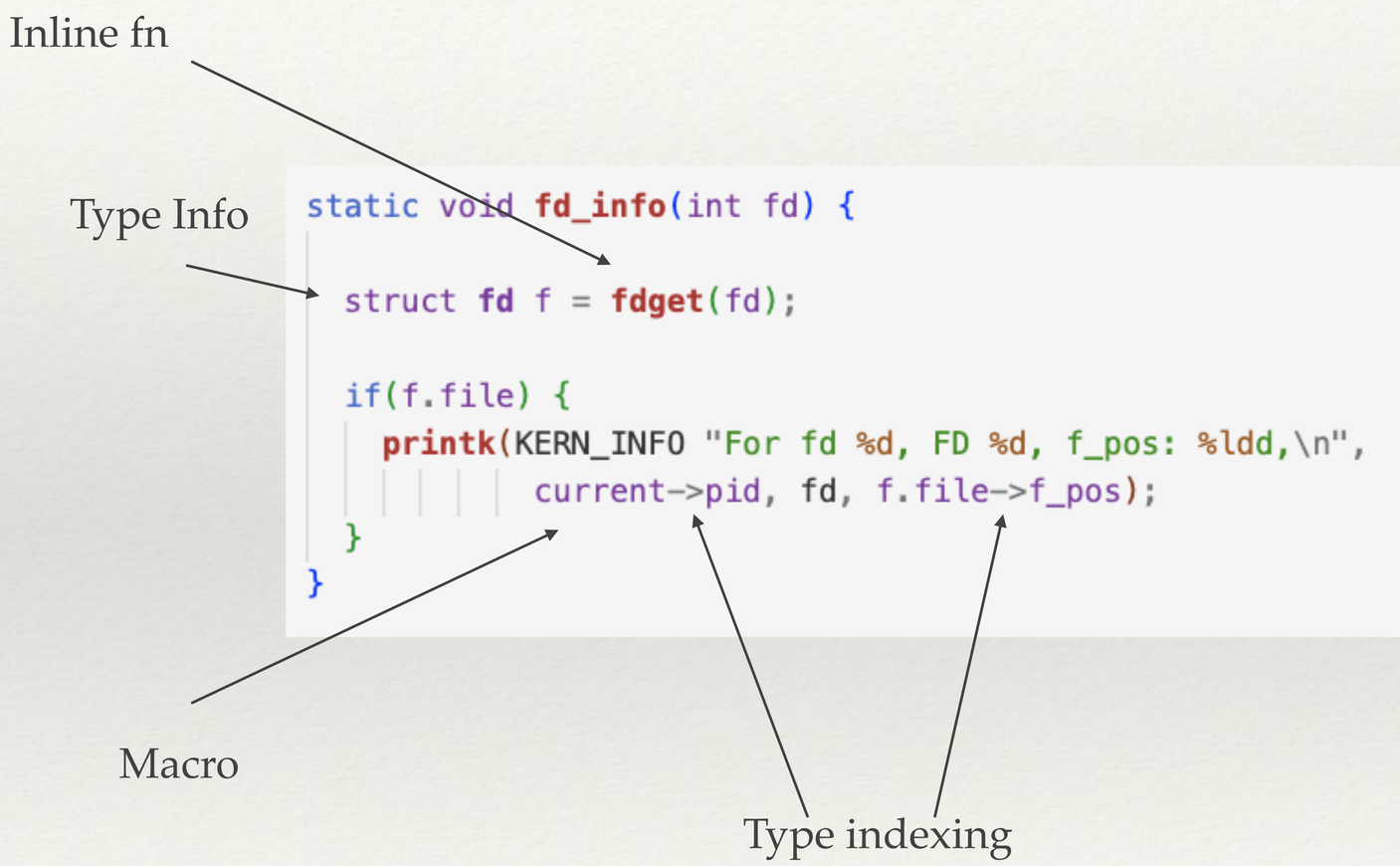
```
Hello world!
```

OS-Level Access

- What kernel developers need to get their work done
  - Runtime
    - Executing standalone kernel functions
    - Accessing kernel data
  - Compile time
    - Macros
    - Types
    - Inline Functions

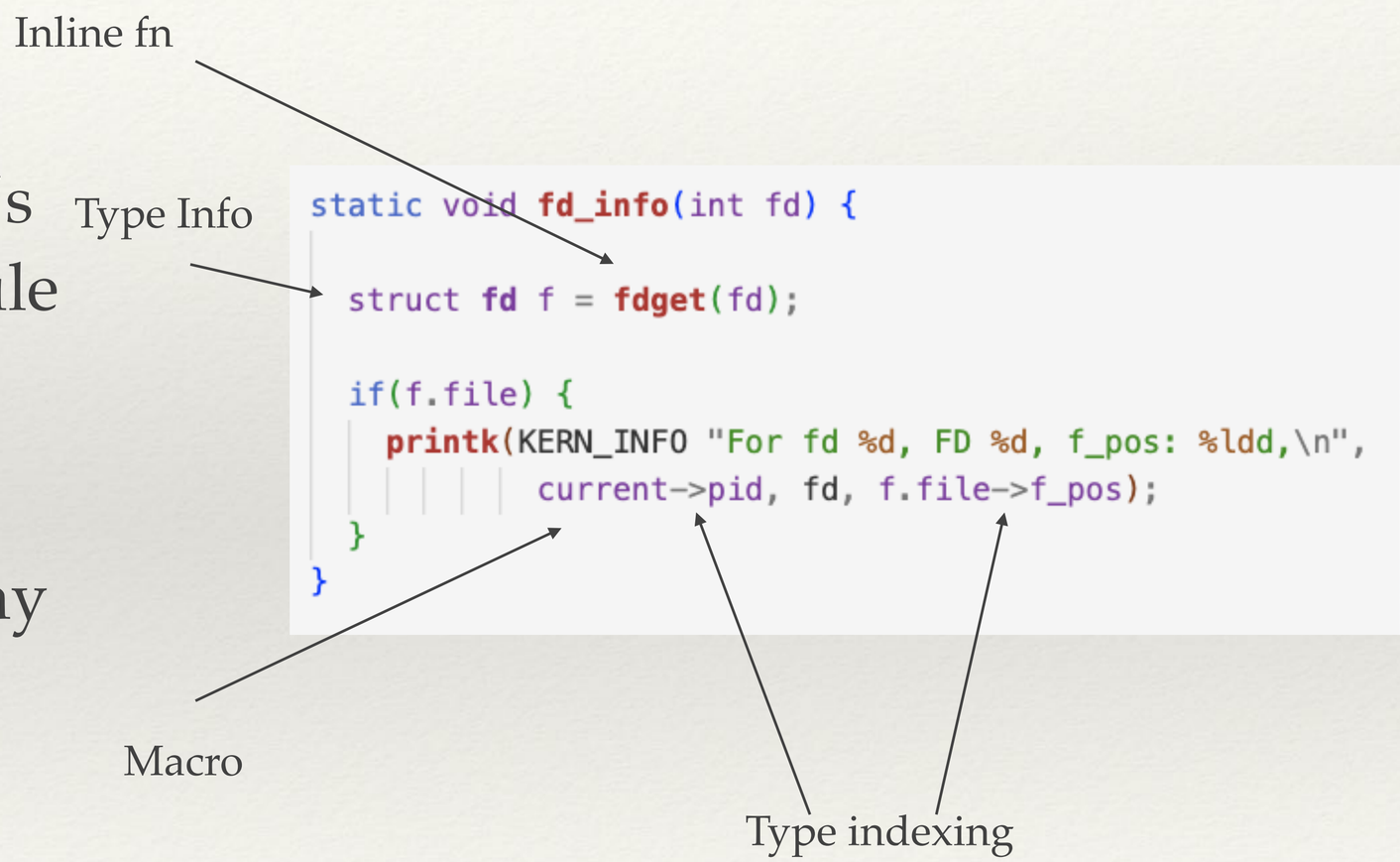


Compile Time OS-Level Access

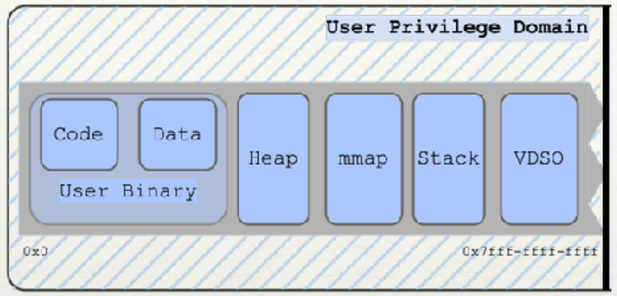


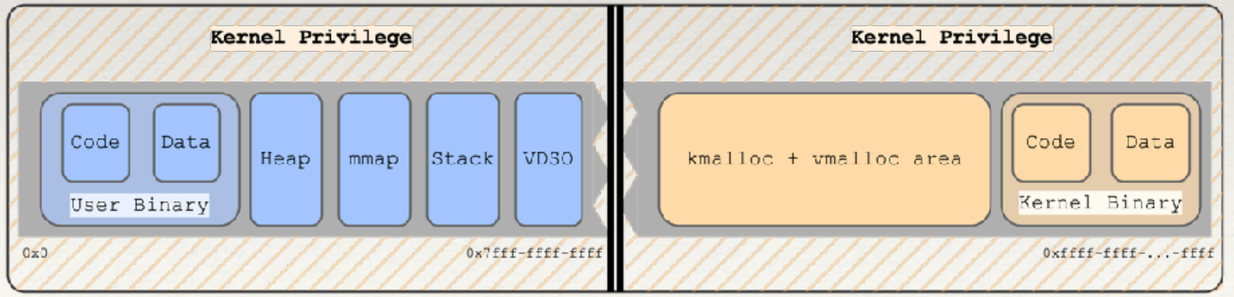
Compile Time OS-Level Access

- How: Utilizing the kernel's "out-of-tree" kernel module build system
- Building this first would have knocked a year off my phd

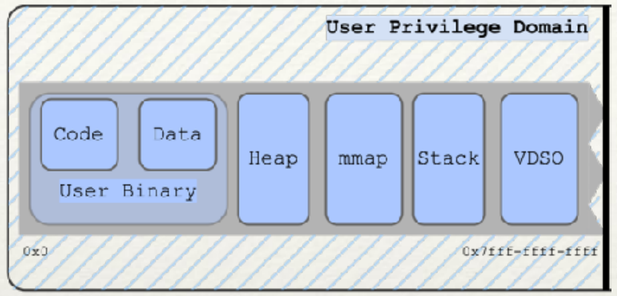


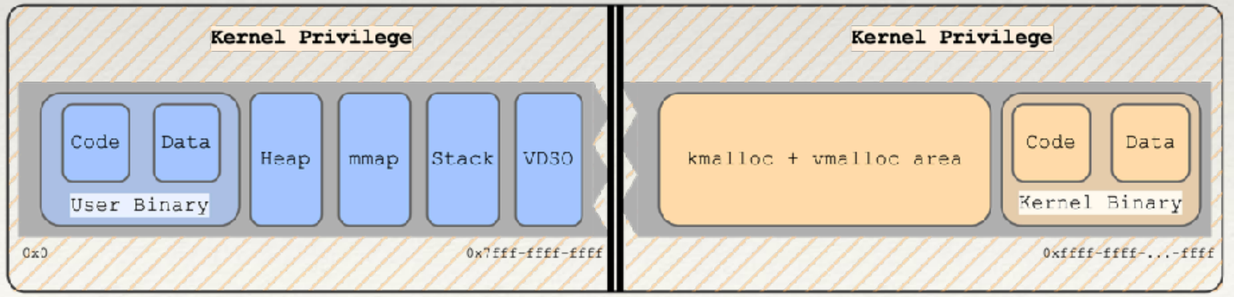
Low-Level to OS-Level Access





Low-Level to OS-Level Access





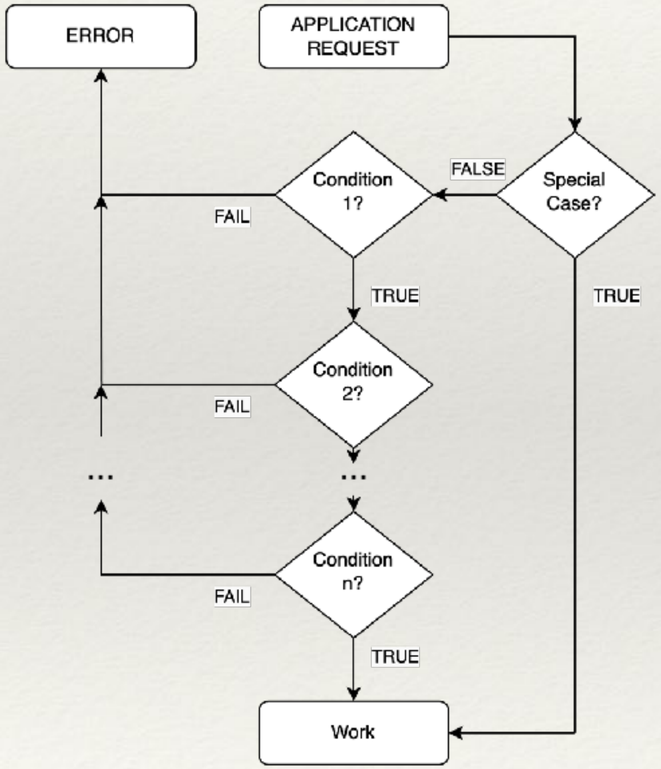
Looks a bit like dynamic loading / linking…

Ch 4: Case Studies

- *Application performance optimization
  - System call shortcutting
- Kernel Code Reuse
  - Observing page table structure

Case Study: Shortcutting

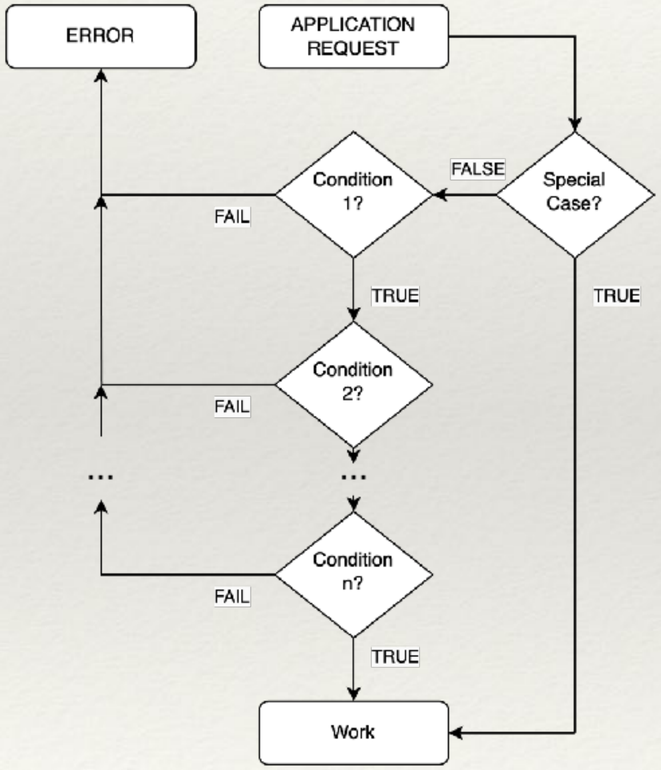
Can we just build this?



Opportunity

Shortcutting

Can we just build this?



Opportunity

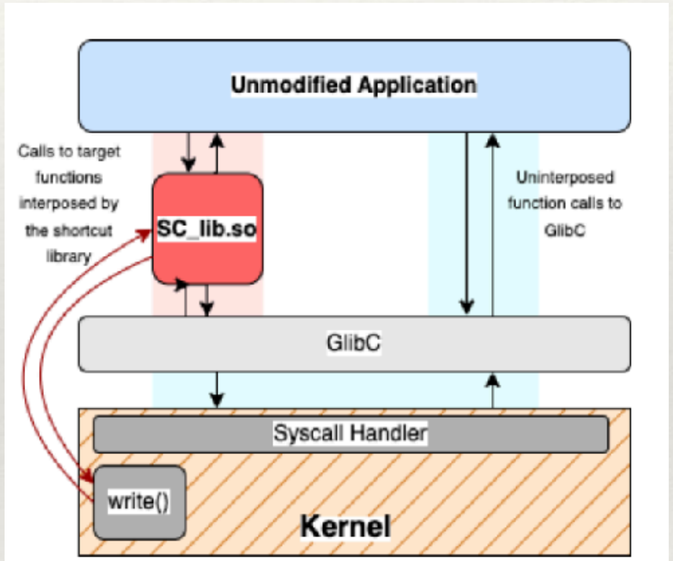
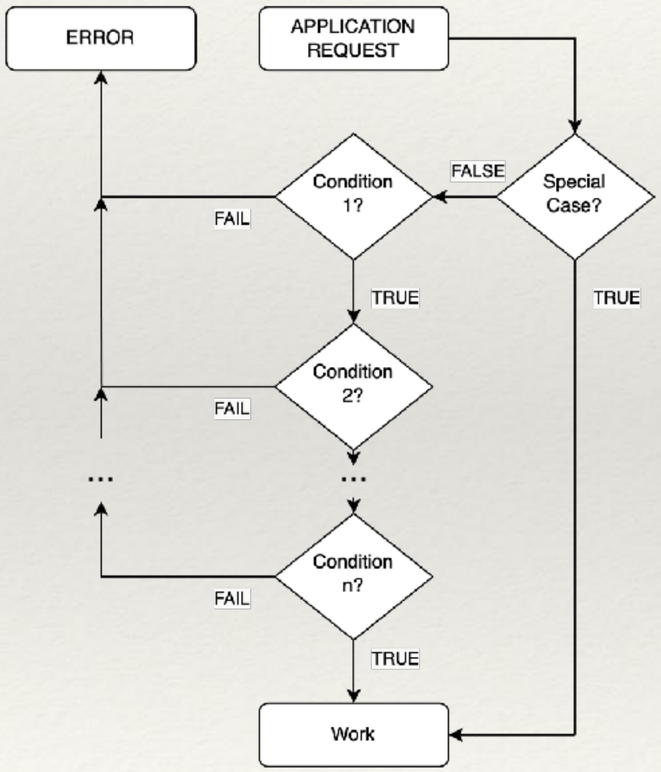


Figure 7. shortcut interpose

Architecture

Shortcutting

Can we just build this?



Opportunity

```
root: [~/Symbi-OS/Tools/bin/shortcut - [master]]
$ ./shortcut.sh -h
###################################

shortcut.sh [args] --- <app> [app args]

Script for launching apps with options for elevation and shortcutting.

Flags:

        -p: passthrough mode, run application without interposer library.
                Ignores all other args,
The following options may interact with kElevate

        -be / -bl: begin with executable elevated / lowered

These ones  can be repeated to specify multiple functions:
        -e <fns>: Elevate around these functions
        -l <fns>: Lower around these functions
        -s <fn1->fn2>: Shortcut between these functions: eg: 'write->ksys_write'
                Don't forget to use quotes around the function names!

Debugging options

        -v: verbose mode for debugging
        -d: dry run, do everything EXCEPT running the application

###################################
```

Interface

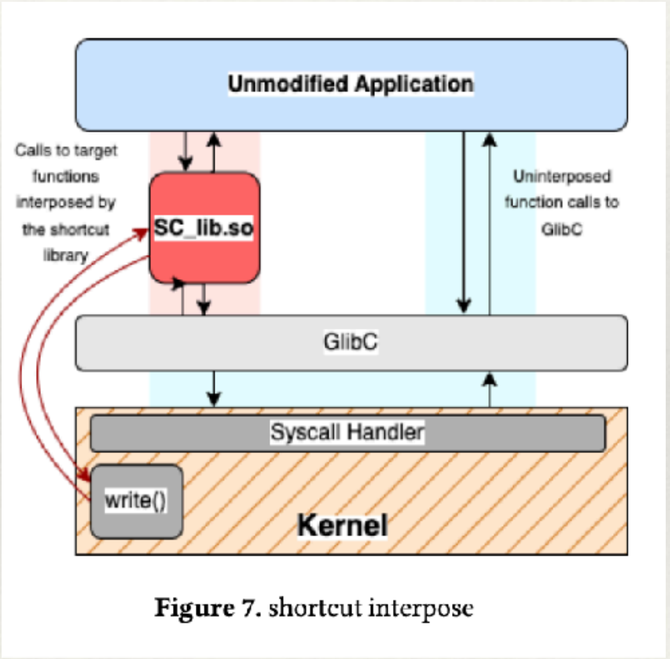


Figure 7. shortcut interpose

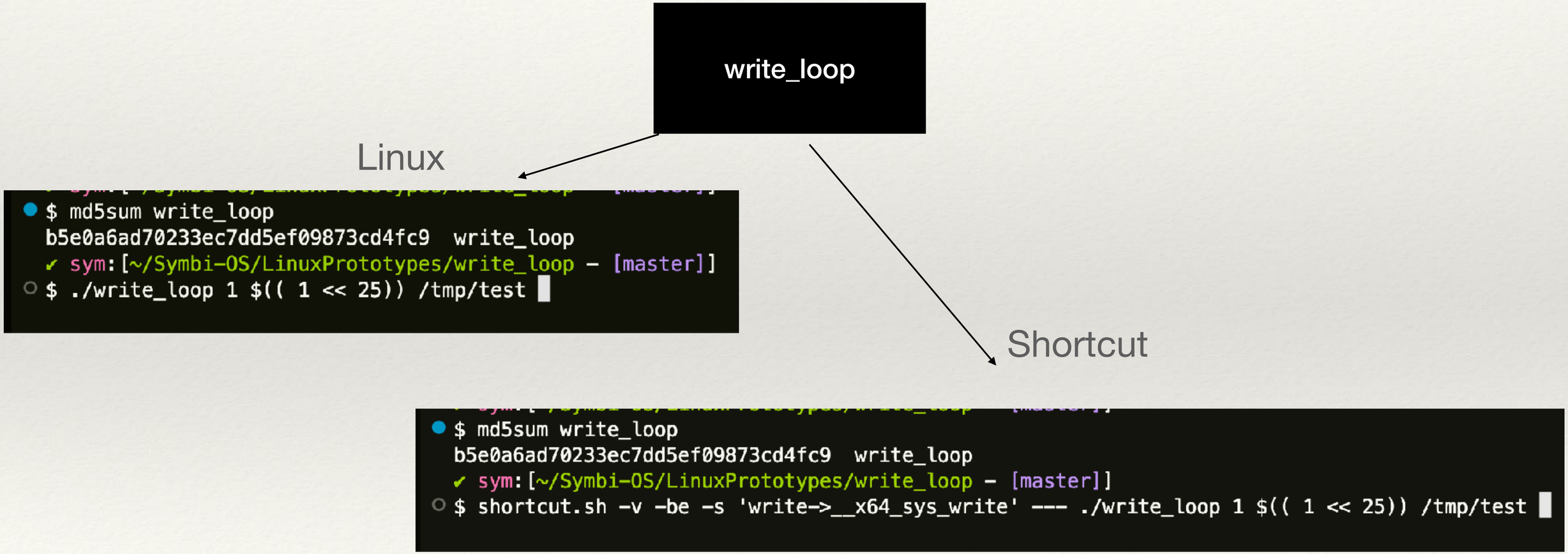
Architecture

Workload

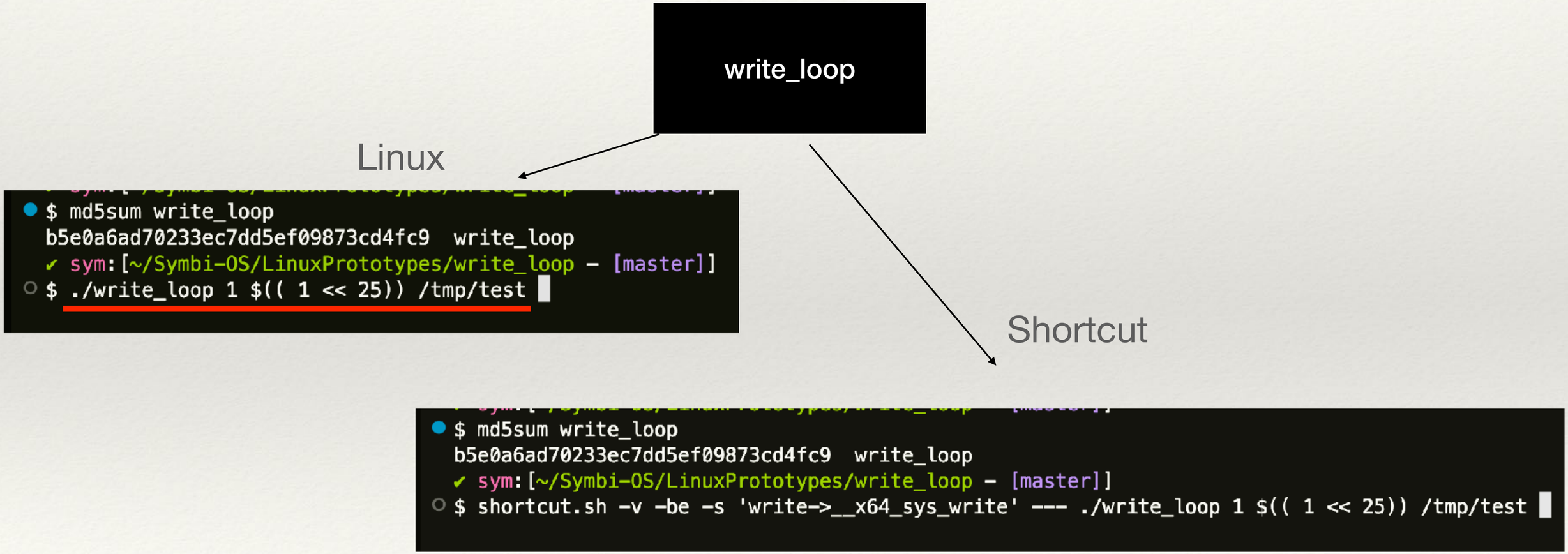
- A C program, *write_loop,* makes write function calls
- Built with a standard toolchain
- With writes destined for a Ramdisk FS

```c
void write_loop(int fd, char *buf, int size, int iter){

    for (int i = 0; i < iter; i++) {
        int rc = write(fd, buf, size);
        if (rc == -1) {
            printf("write failed\n");
            exit(1);
        }
    }

}
```

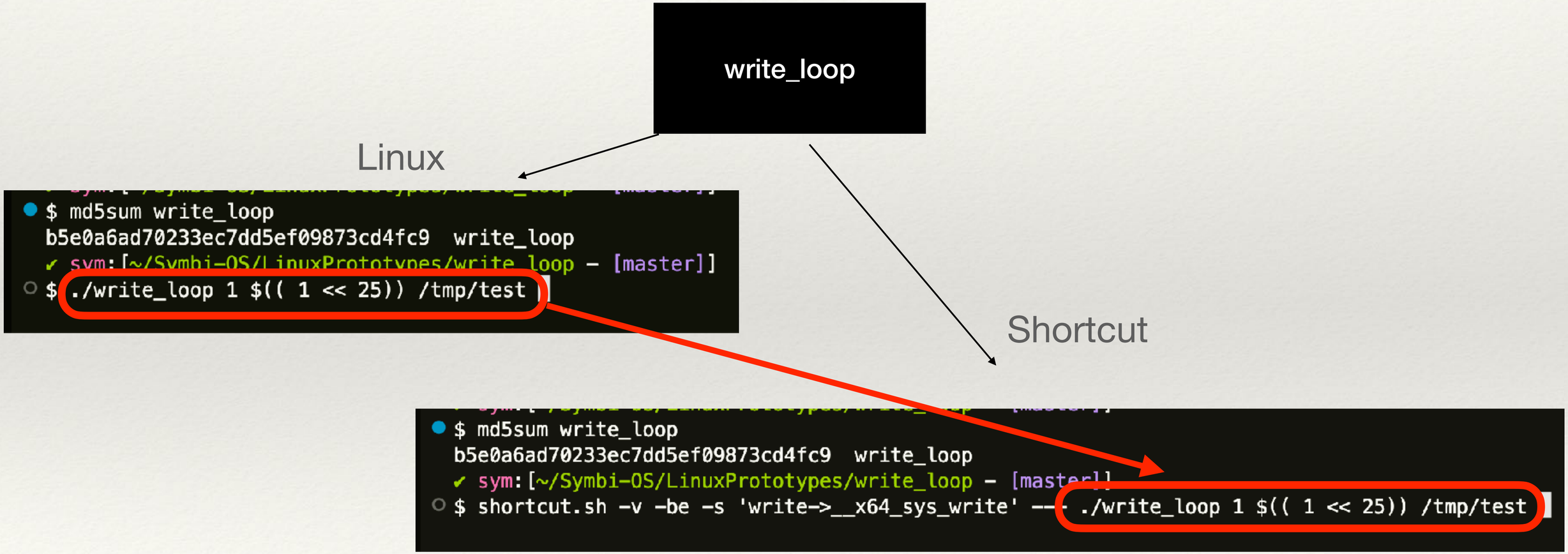
A Comparison



A Comparison

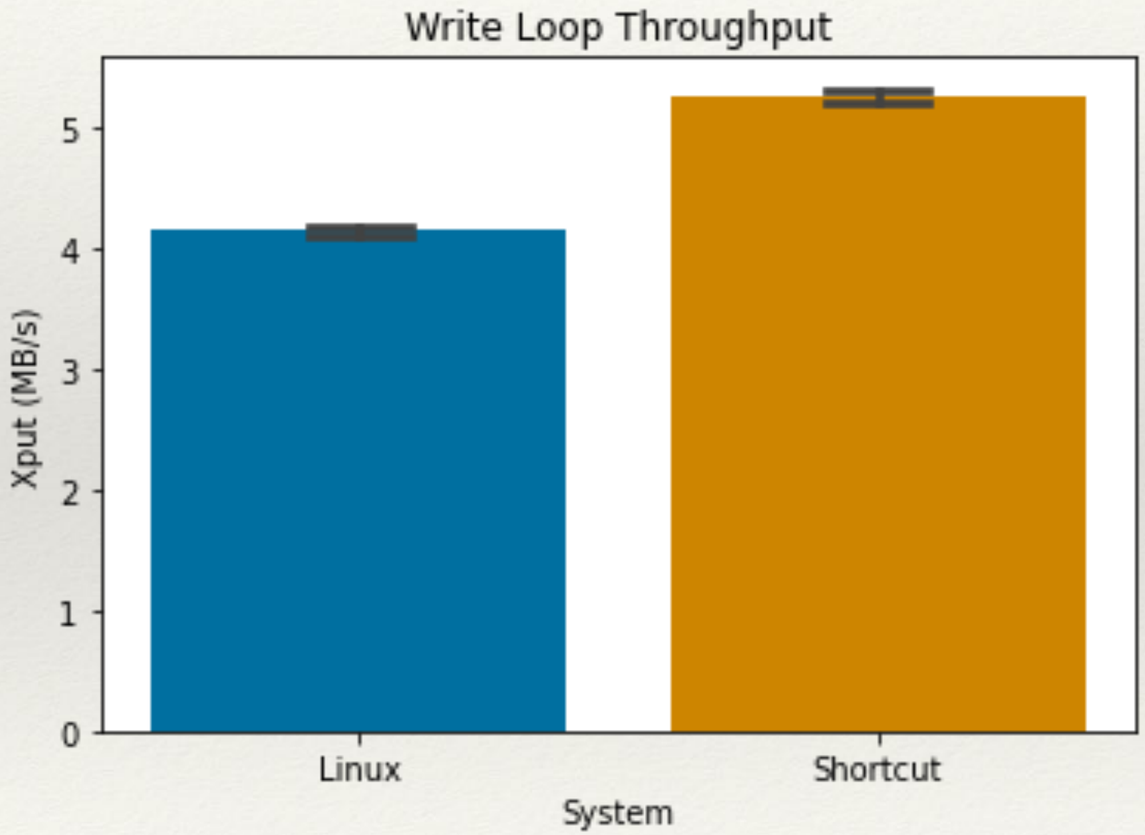


A Comparison



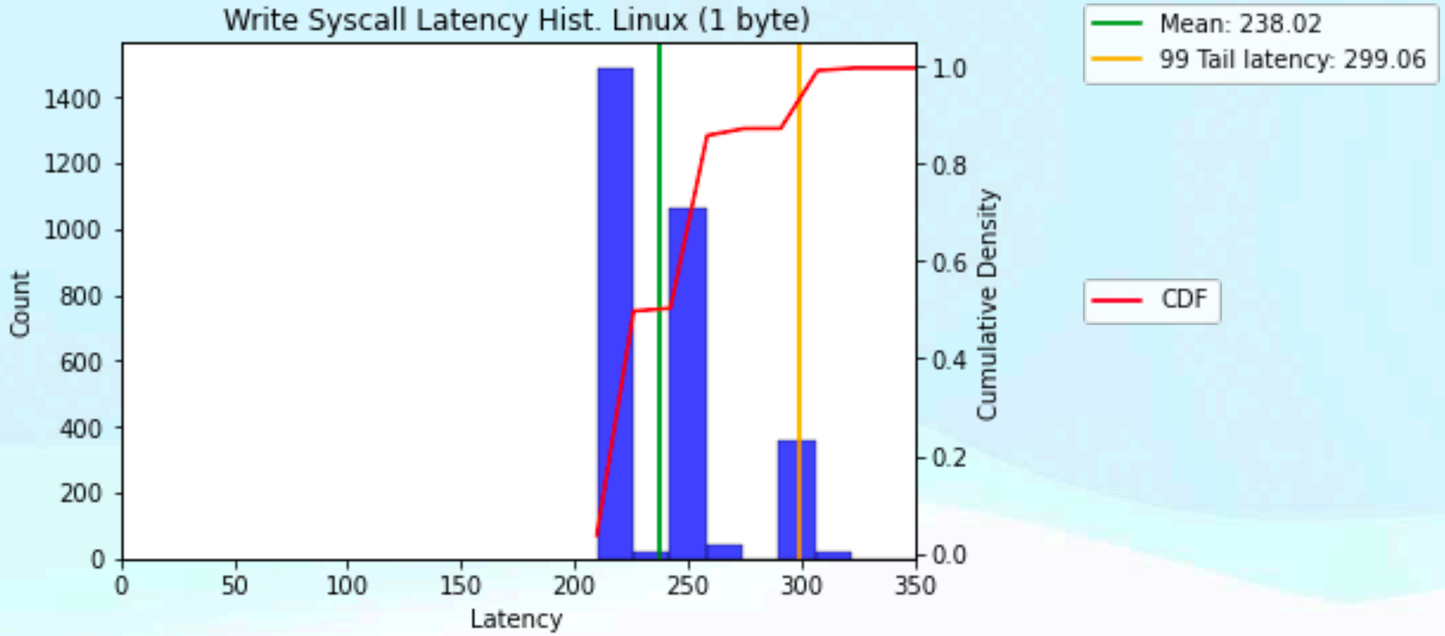
A Comparison

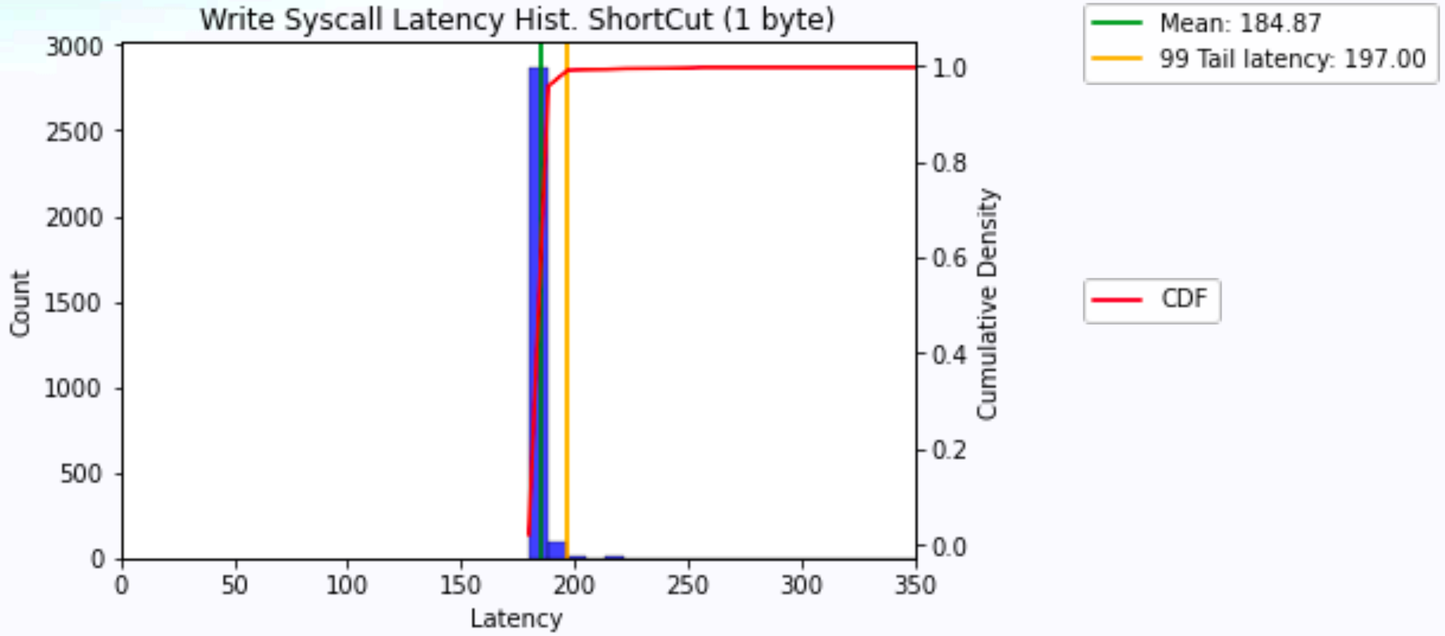
Throughput Comparison



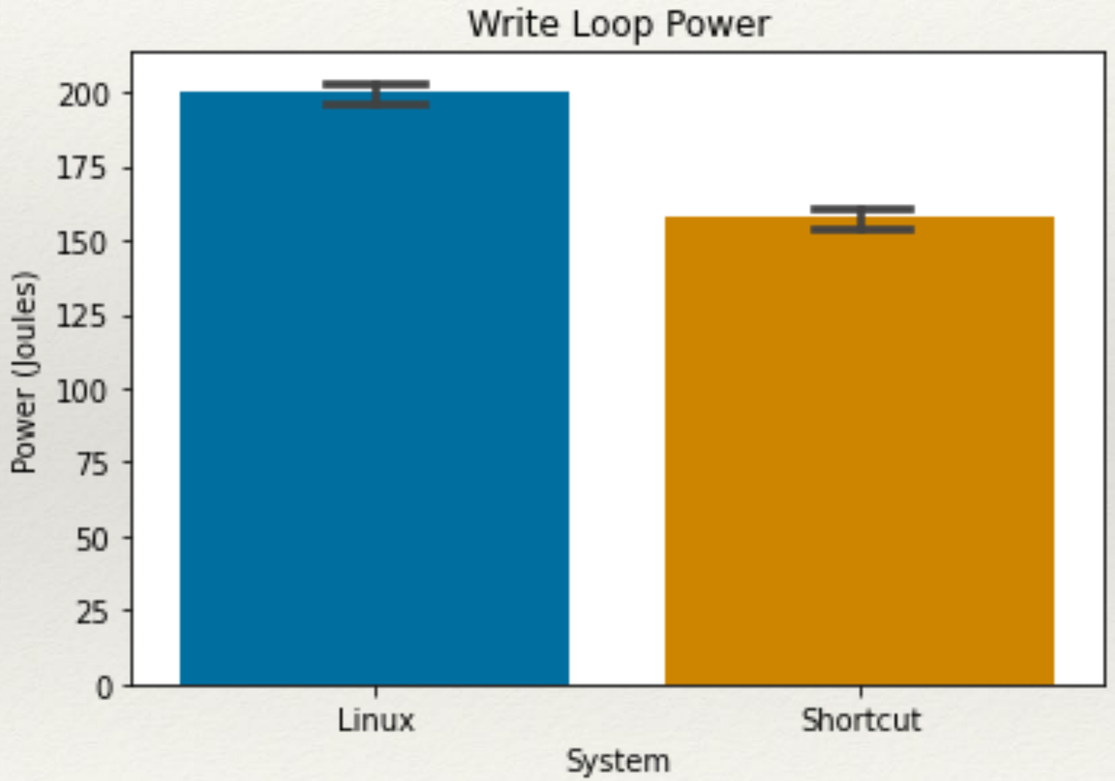
28% Higher x-put

Syscall Latencies





Energy Comparison



24% Reduction in Energy

Shortcutting: Macrobenchmarks

- Refer to dissertation
  - Redis:
    - Shallow shortcuts: 9-11% xput improvement
    - Deep shortcuts: 20-22% xput improvement
  - Memcached
    - Shallow shortcuts: 9-11% xput improvement

Shortcutting: Macrobenchmarks

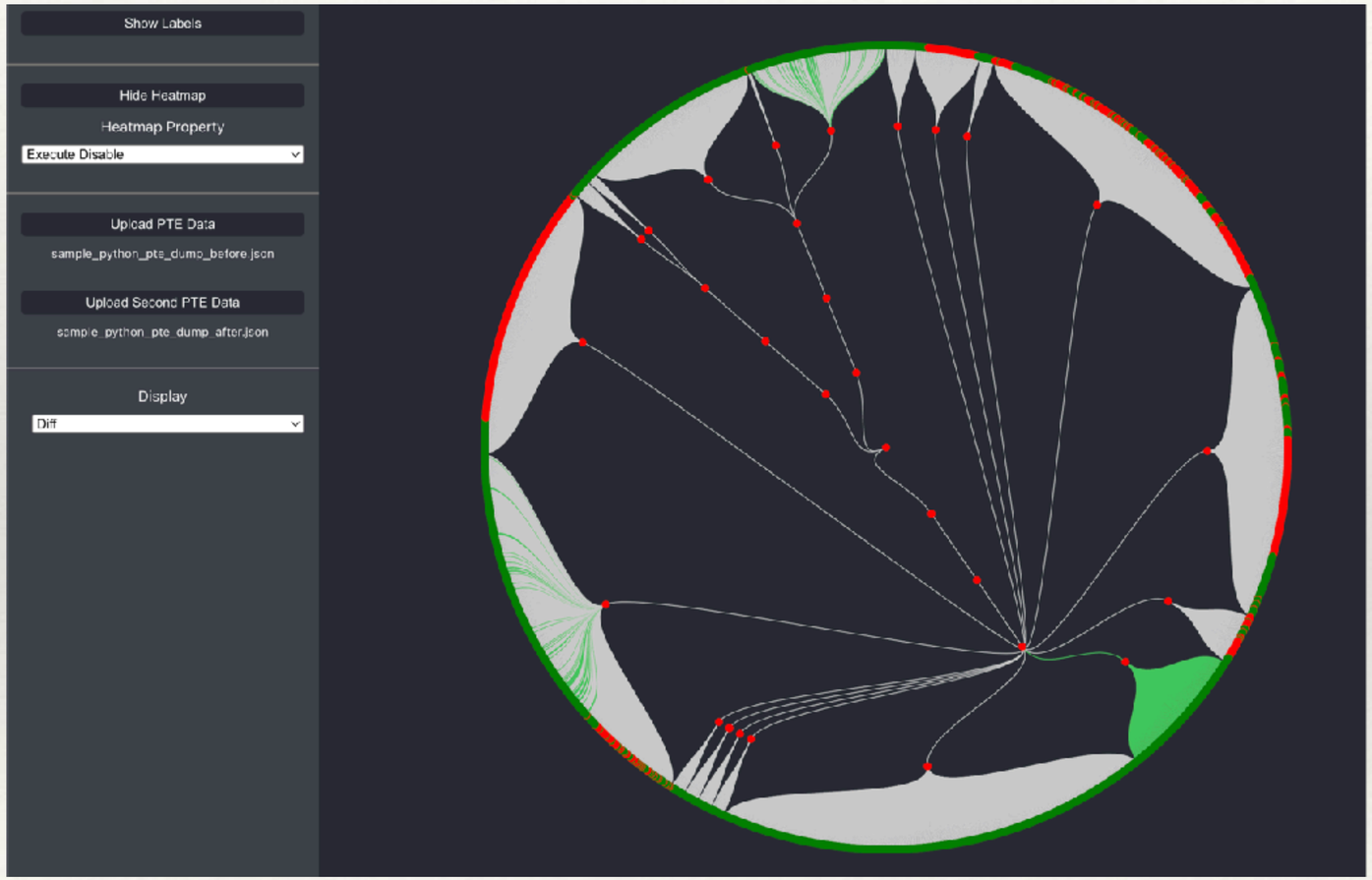
- Refer to dissertation
  - Redis:
    - Shallow shortcuts: 9-11% xput improvement
    - Deep shortcuts: 20-22% xput improvement
  - Memcached
    - Shallow shortcuts: 9-11% xput improvement

```
$ ./sc.sh -s write->__x64_sys_write \
    --- ./wrLoop $SZ $ITER $PATH
```



```
$ ./sc.sh -s \
    write->tcp_sendmsg:0x42ed60 \
    --- ./wrLoop $SZ $ITER $PATH
```

Case Study: Kernel Code Reuse



Page Map Level 5 Entry

63	62 ... 52	51 ... M	M-1 ... 12	11 ... 8	7	6	5	4	3	2	1	0
XD	AVL	Reserved (0)	Bits 12 - (M-1) of address	AVL	RSVD (0)	AVL	A	PCD	PWT	U/S	R/W	P

Page Map Level 4 Entry

63	62 ... 52	51 ... M	M-1 ... 12	11 ... 8	7	6	5	4	3	2	1	0
XD	AVL	Reserved (0)	Bits 12 - (M-1) of address	AVL	RSVD (0)	AVL	A	PCD	PWT	U/S	R/W	P

Page Directory Pointer Table Entry

63	62 ... 52	51 ... M	M-1 ... 12	11 ... 8	7	6	5	4	3	2	1	0
XD	AVL	Reserved (0)	Bits 12 - (M-1) of address	AVL	PS (0)	AVL	A	PCD	PWT	U/S	R/W	P

Page Directory Entry

63	62 ... 52	51 ... M	M-1 ... 12	11 ... 8	7	6	5	4	3	2	1	0
XD	AVL	Reserved (0)	Bits 12 - (M-1) of address	AVL	PS (0)	AVL	A	PCD	PWT	U/S	R/W	P

P: Present
R/W: Read/Write
U/S: User/Supervisor
PWT: Write-Through
PCD: Cache Disable
A: Accessed
PS: Page Size
AVL: Available
M: Maximum Physical Address Bit
XD: Execute Disable

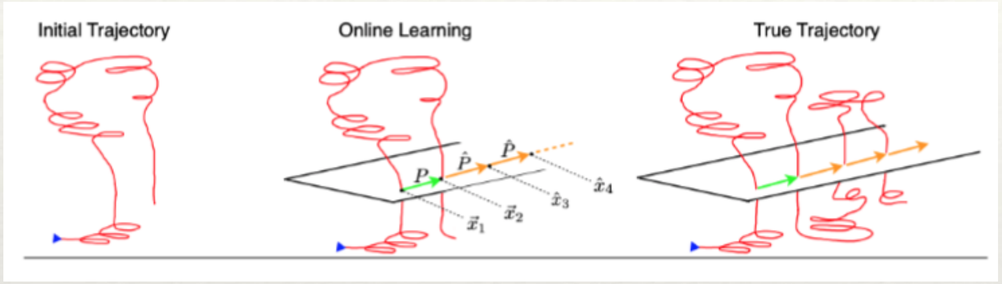
Concluding

Contributions Recap

- Defined the OS model Dynamic Privilege; introduced the notion of fine grained HW privilege switching
- Prototyped a mechanism for extending a GPOS kernel with support for Dynamic Privilege by adding a system call, kElevate
- Produced a library, kele, which codifies our approach to using kelevate and facilitates future use of the mechanism
- Explored the use of Dynamic Privilege through case studies

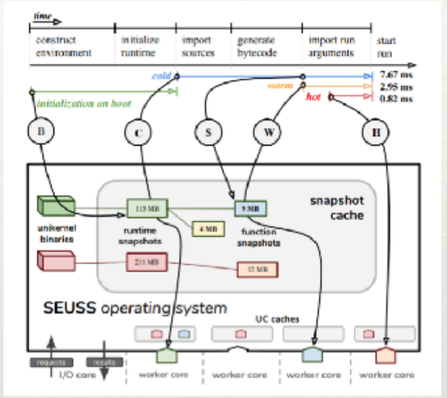
Prior Work

ASC

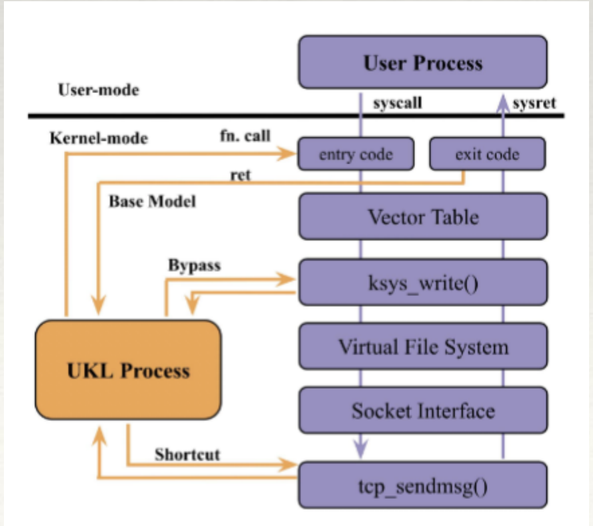


Optimizing Systems

SEUSS



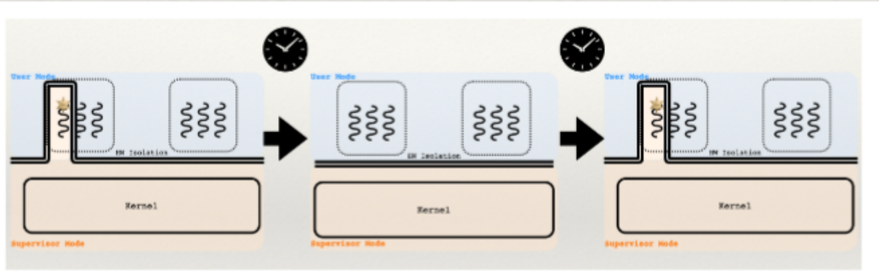
UKL



Realization of a known alternative OS Architecture

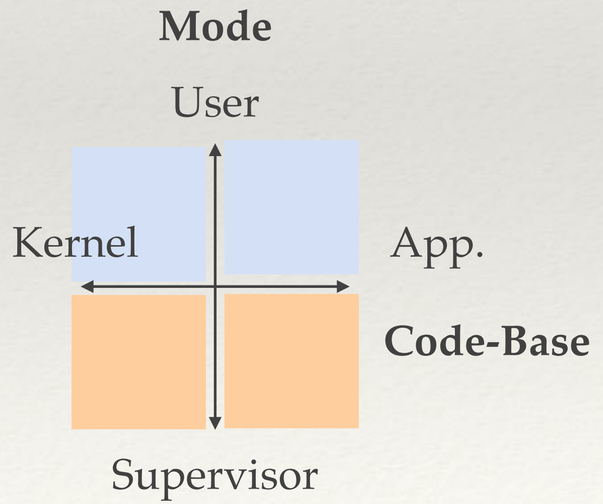
Mechanism

Dynamic Privilege



Future Work

- Lowering kernel code
- Privileged Microkernel Servers
- Safe Privileged Processes
- Combination with capabilities



Collaborators

- Mentors:
  - BU: Jonathan Appavoo / Orran Krieger / Renato Mancuso
  - RH: Larry Woodman / Uli Drepper / Richard Jones / Daniel Bristot de Oliveira
- Collaborators:
  - Arlo Albelli / Eric Munson / Ali Raza / Jim Cadden / Yara Awad
  - Amos Waterland / Han Dong / Parul Sohal / Dan Schatzberg
  - Albert Slepak / Ryan Sullivan / Ke Li / Ting Hsu

Thesis Statement

We introduce Dynamic Privilege, an OS model enabling threads to adjust their hardware privilege levels on the fly, challenging the traditional separation between kernel and application.

Dynamic Privilege provides granular control and requires only minor modifications to support it in an existing OS.

Further, Dynamic Privilege enables new system evolution and optimization while maintaining compatibility with existing software ecosystems.