Virtual Machines

h/t: Simon Peter, Andrew Baumann

Multiprocessor Recap

- Cache coherence requires extensive bookkeeping and hw messages to manage shared data
- Contention for shared data can slow CPUs
- How do we reduce contention?
  - Per-processor data structures (Barrelfish)
  - Optimize coherence communication (MCS, Barrelfish)
  - Lock free data structures (MCS, RCU, …)

Adve Implementation Rule

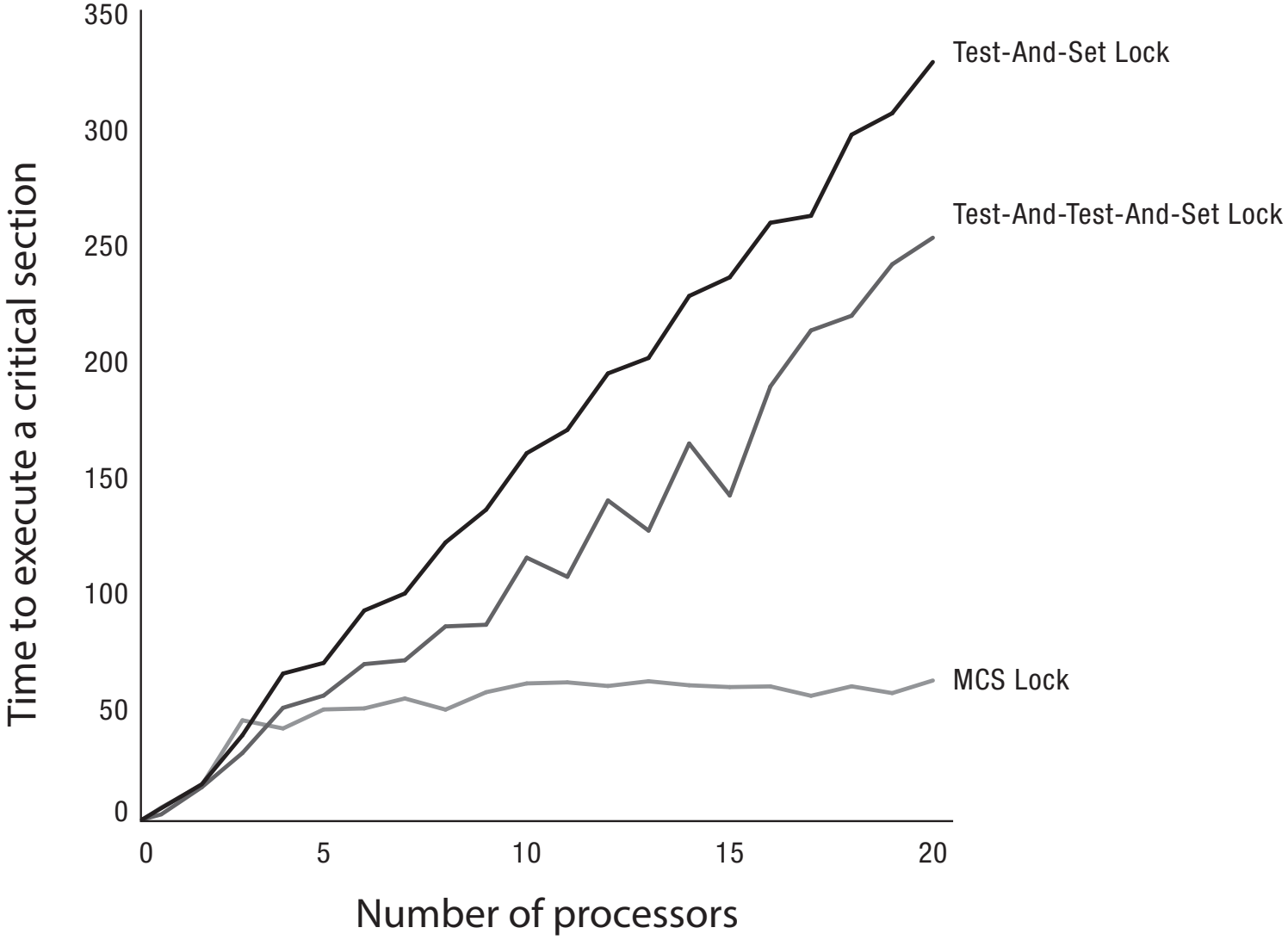
Let's assume (wlog) that each process specifies that its own operations happen in some order

- E.g., read A, write B, append C, …
- If concurrent, system can choose the order

Serializable/sequentially consistent if

1. Operations applied in processor order, and
2. all operations to same memory location are serialized (as if to a single copy)

Test (and Test) and Set Performance



MCS Lock

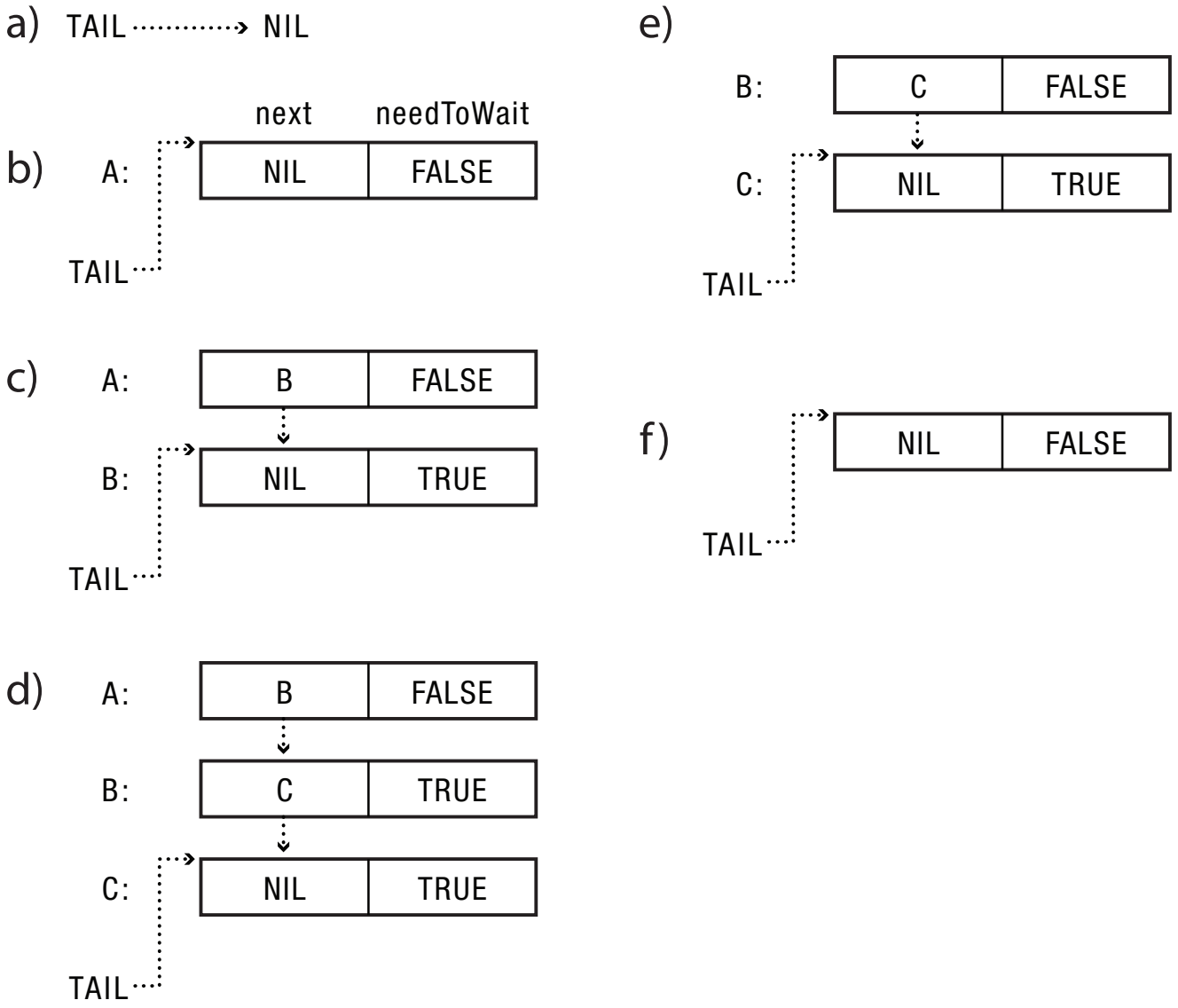
- Maintain a list of threads waiting for the lock
  - Thread at front of list holds the lock
  - MCSLock::tail is last thread in list
  - Add to tail using CompareAndSwap
- Lock handoff: set next->needToWait = FALSE
  - Next thread spins: while needToWait is TRUE

MCS Lock Implementation

```
MCSLock::acquire() {
   myTCB−>next = NULL;
   myTCB−>needToWait = FALSE;
   oldTail = tail;
   while (!compareAndSwap(&tail,
                  oldTail, &myTCB)) {
       oldTail = tail;
   }
   if (oldTail != NULL) {
      myTCB->needToWait = TRUE;
      oldTail−>next = myTCB;
      memory_barrier();
      while (myTCB−>needToWait)
          ;
   }
}
```

```
TCB {
   TCB *next;              // next in line
   bool needToWait;
}
MCSLock {
   Queue *tail = NULL; // end of line
}

MCSLock::release() {
   if (!compareAndSwap(&tail,
                  myTCB, NULL)) {
      while (myTCB−>next == NULL)
          ;
     myTCB−>next −>needToWait=FALSE;
   }
}
```

MCS In Operation



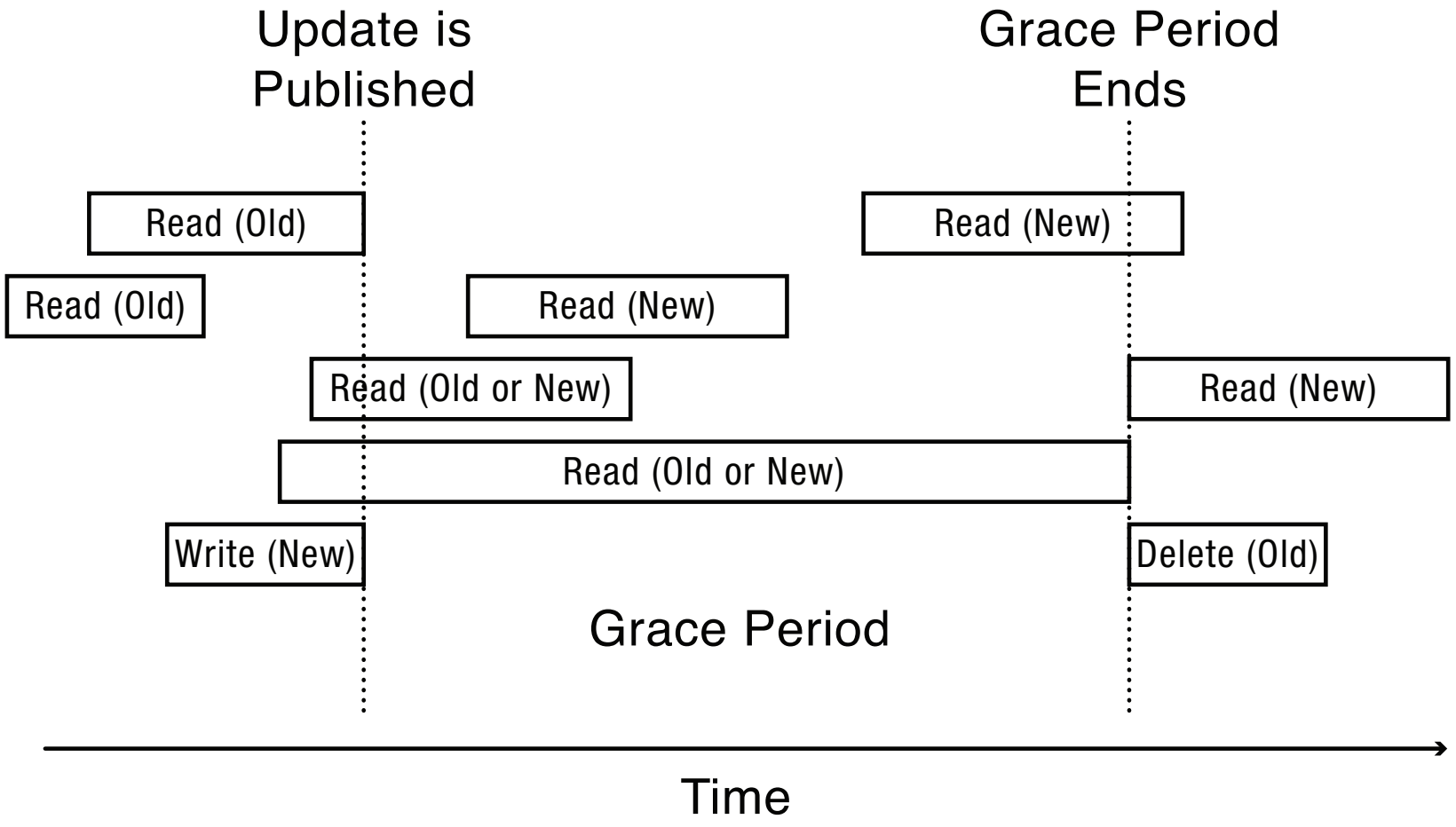
In Practice

- Spin adaptively
  - No delay if few waiting
  - Longer delay if many waiting
  - Guess number of waiters by how long you wait

Read-Copy-Update (RCU) Locks

- Goal: very fast reads to shared data
  - Reads proceed without first acquiring a lock
  - OK if write is (very) slow
- Restricted update
  - Writer computes new version of data structure
  - Publishes new version with a single atomic instruction
- Multiple concurrent versions
  - Readers in progress may see old or new version
  - New readers see new version
- Integration with thread scheduler
  - Readers in progress at previous update must complete within grace period
  - Then ok to garbage collect old version

Read-Copy-Update



Read-Copy-Update Implementation

- Readers disable interrupts on entry
  - Guarantees they complete critical section in a timely fashion
  - No read or write lock
- Writer
  - Acquire write lock
  - Compute new data structure
  - Publish new version with atomic instruction
  - Release write lock
  - Wait for time slice on each CPU
  - Only then, garbage collect old version of data structure

Lock-free Data Structures

- No lock for either read or write
  - No lock contention!
  - No deadlock!
- General method using compareAndSwap
  - Create copy of data structure
  - Modify copy
  - Swap in new version iff no one else has
  - Garbage collect old version (RCU style)
  - Restart if pointer has changed

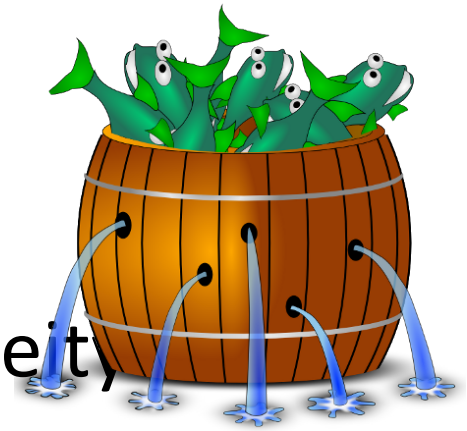
RCU Balanced Tree

- Readers can always walk tree
- Writers construct a new version of tree, use compare and swap to atomically replace
  - New readers see new version of tree
- Rebalancing on addition/deletion is a local operation
  - Compare and swap to atomically replace subtree

RCU Memory Management

- Operations
  - Adding/removing a memory region: mmap, munmap
  - Adjusting memory bounds: sbrk, mmap
  - Lazy allocation of page tables
  - Lazy allocation of pages
  - Lazy page table entry update (e.g., copy on write)
- For multithreaded user programs
  - Concurrent page faults
- Both machine-independent and machine-dependent data structures

Barrelfish:
The OS as Distributed System



- 2007-today, ETH Zurich
- OS for “multicore” systems
- Goals: Scalability, agility, heterogeneity
  - OS can be reconfigured for each new machine
- No shared state
- Message passing
- Software consistency mechanisms

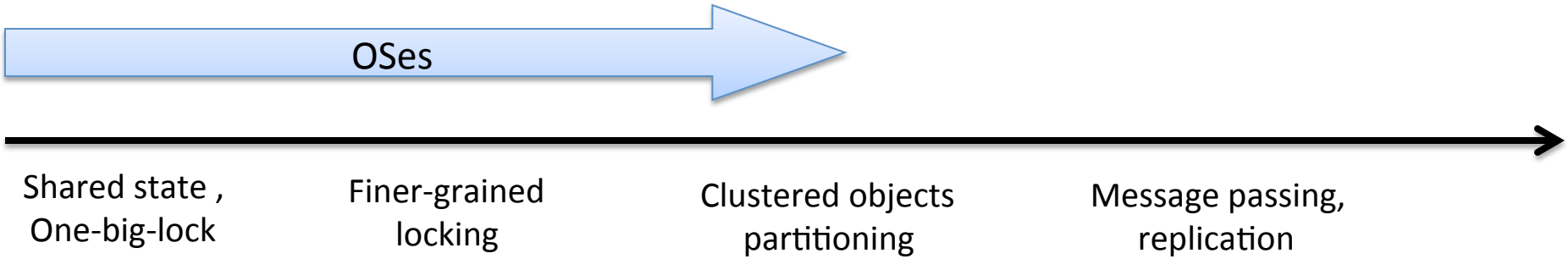
Barrelfish Observations

- Heterogeneous cores, cache coherence protocols
  - Optimizations for one architecture may be counterproductive for another
- Complex core-core interconnect topologies
- Message passing can be faster than locking
  - Cache locality inside the critical section

Barrelfish

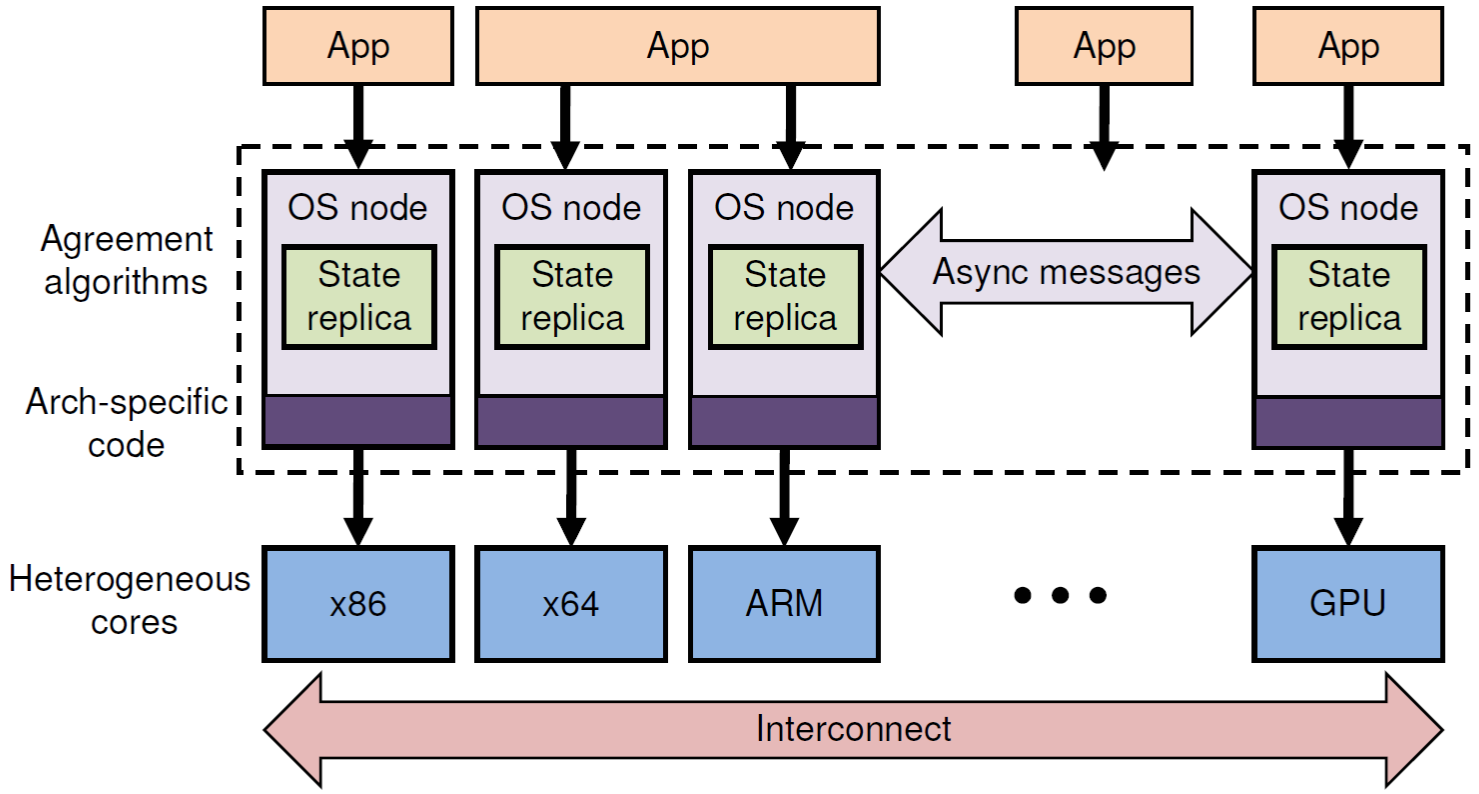
- Explicit inter-core communication
  - Message pipelining vs. synchronous locking
- Hardware neutral OS structure
  - Cache coherence costs vary dramatically
- Replicated state, not shared state

Clear trend….

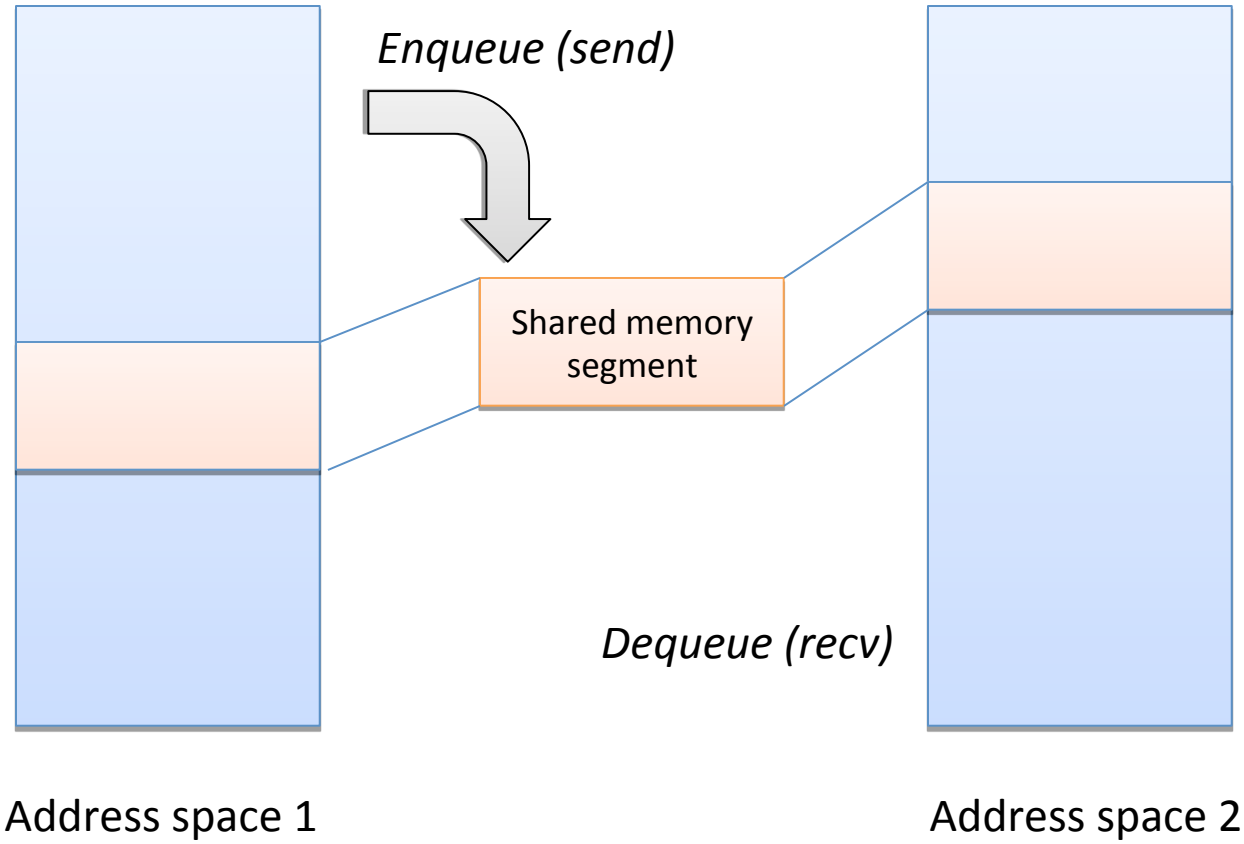


- Finer-grained locking of shared memory
- Replication as an optimization of shared memory

Barrelfish Architecture



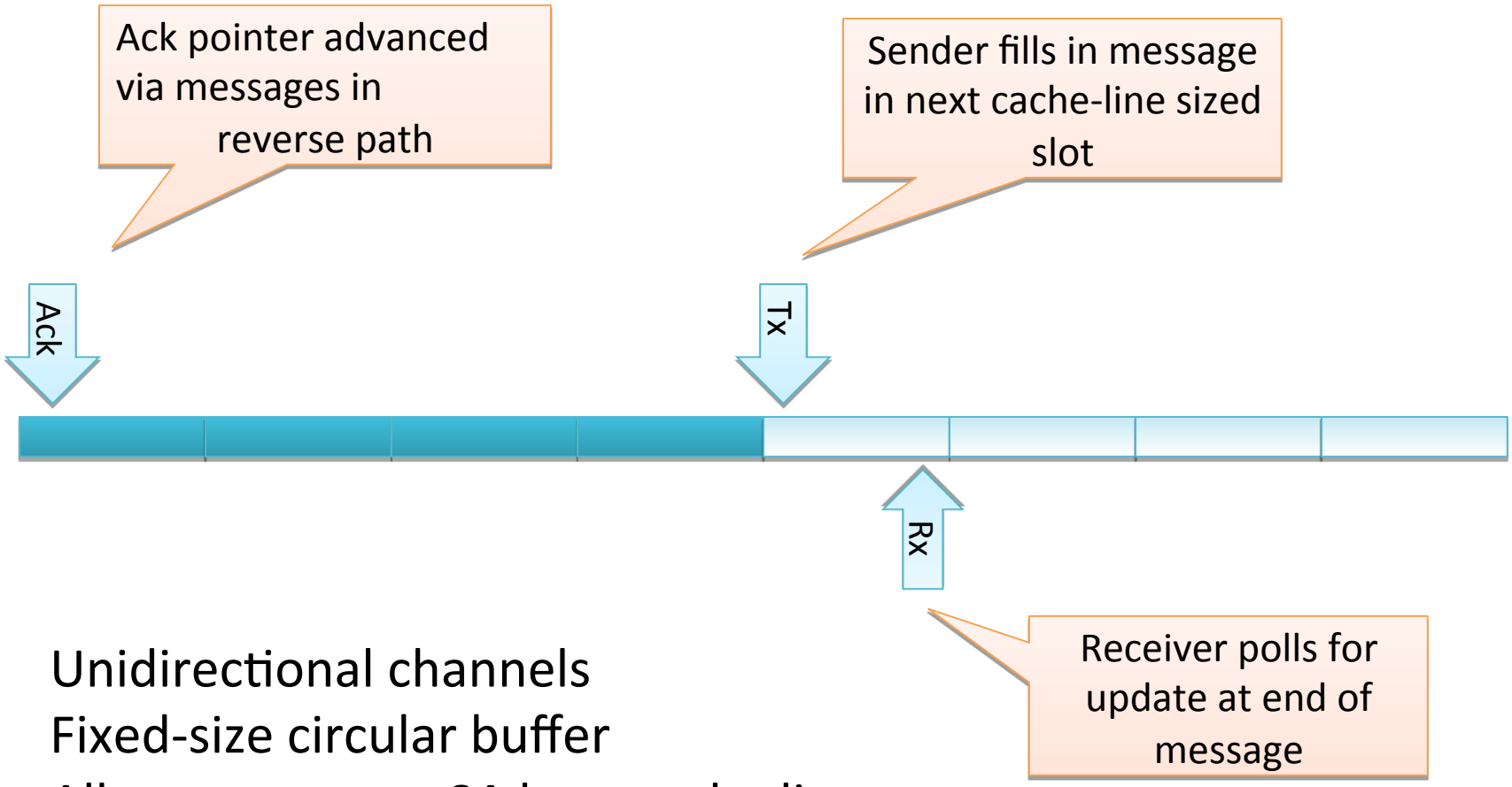
User-space producer-consumer queue



Barrelfish CC-UMP (x86) Interconnect Driver

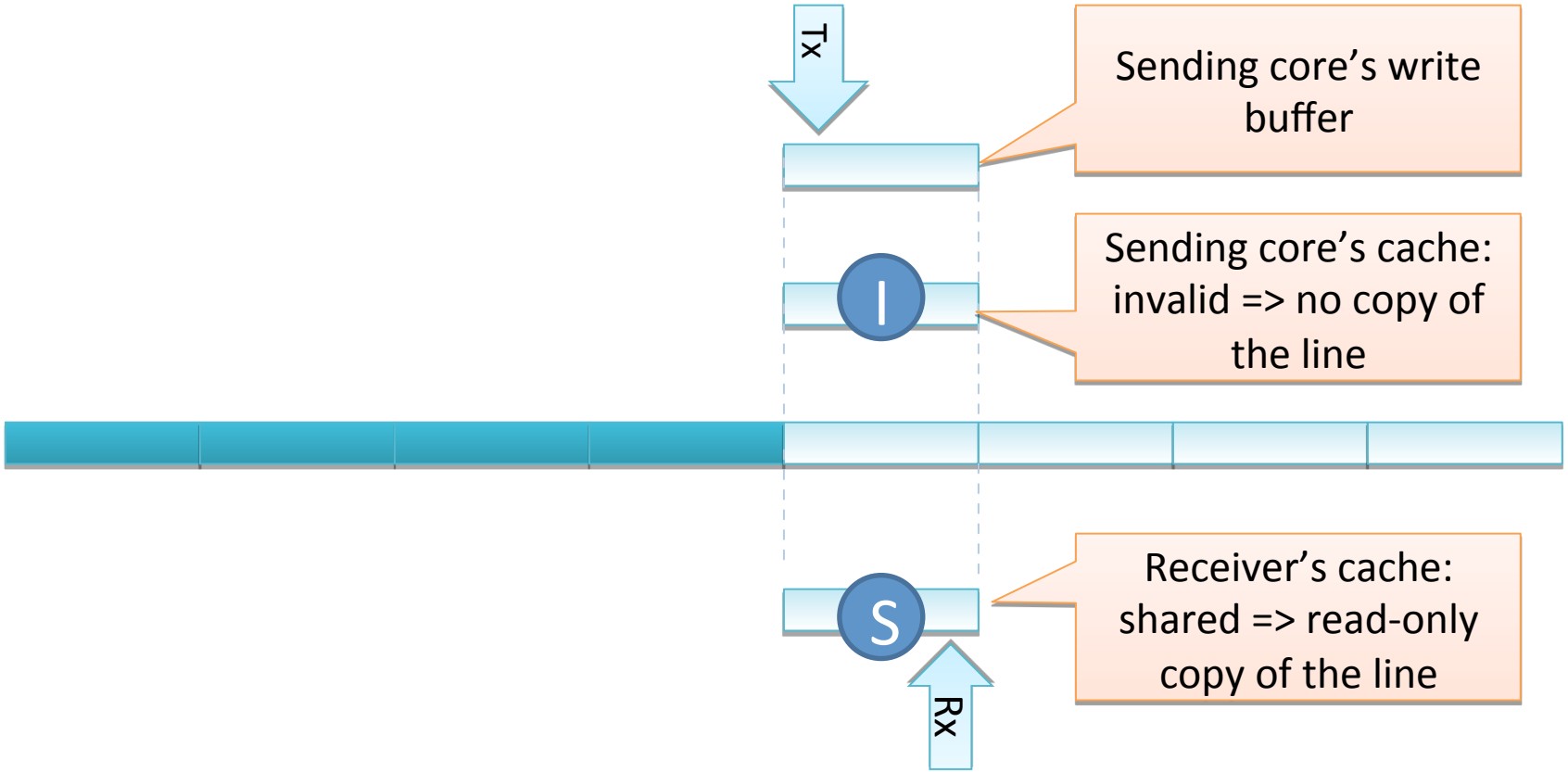
- Cache-coherent shared memory
  - inspired by URPC
- Ring buffer of cache-line sized messages
  - 64 bytes or 32 bytes
  - 1 word for bookkeeping; last one written
- Credit-based flow control out of band
- One channel per IPC binding (not shared)

CC-UMP: cache-coherent user-space messaging

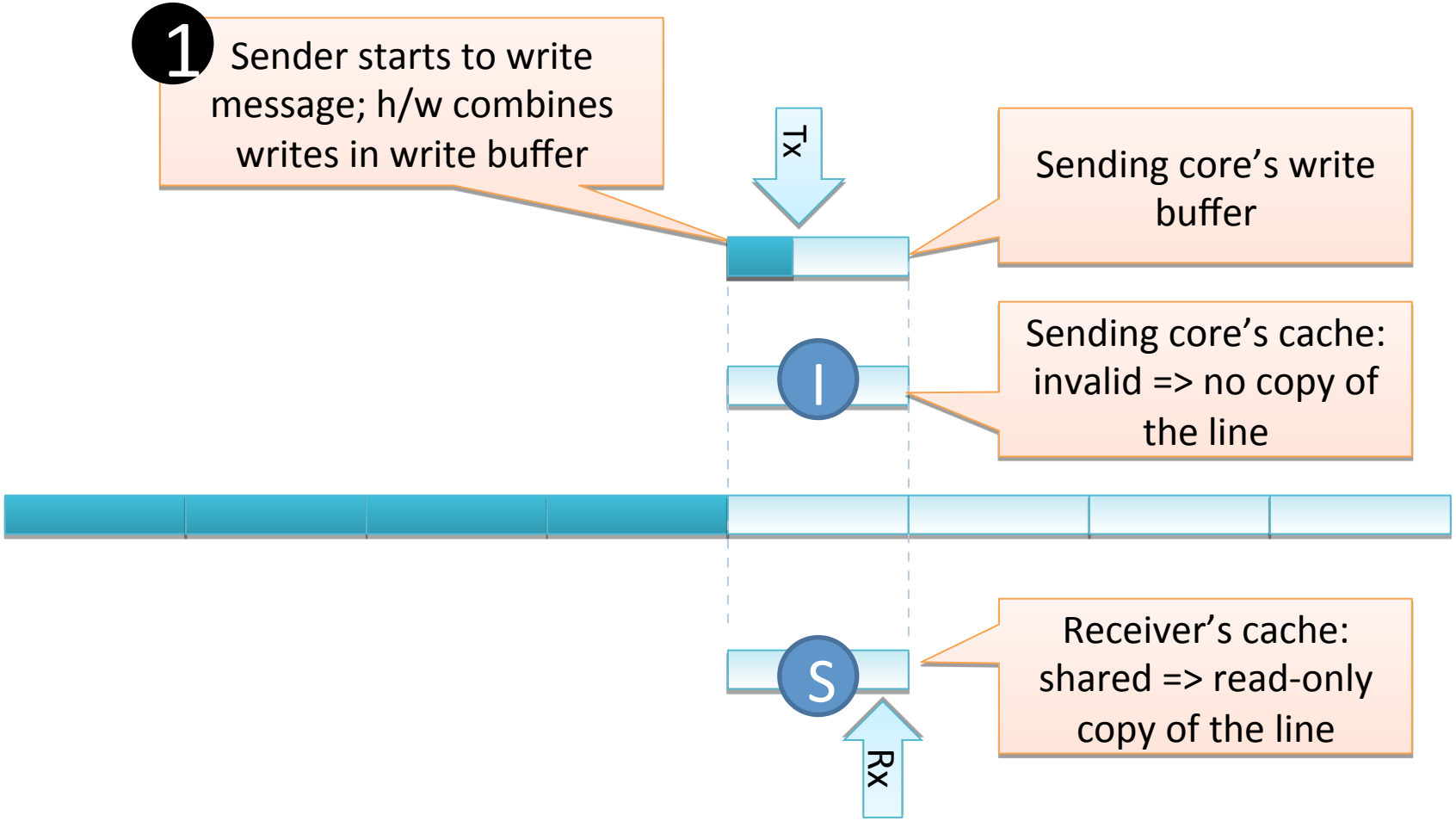


Unidirectional channels
Fixed-size circular buffer
All messages are 64-byte cache lines

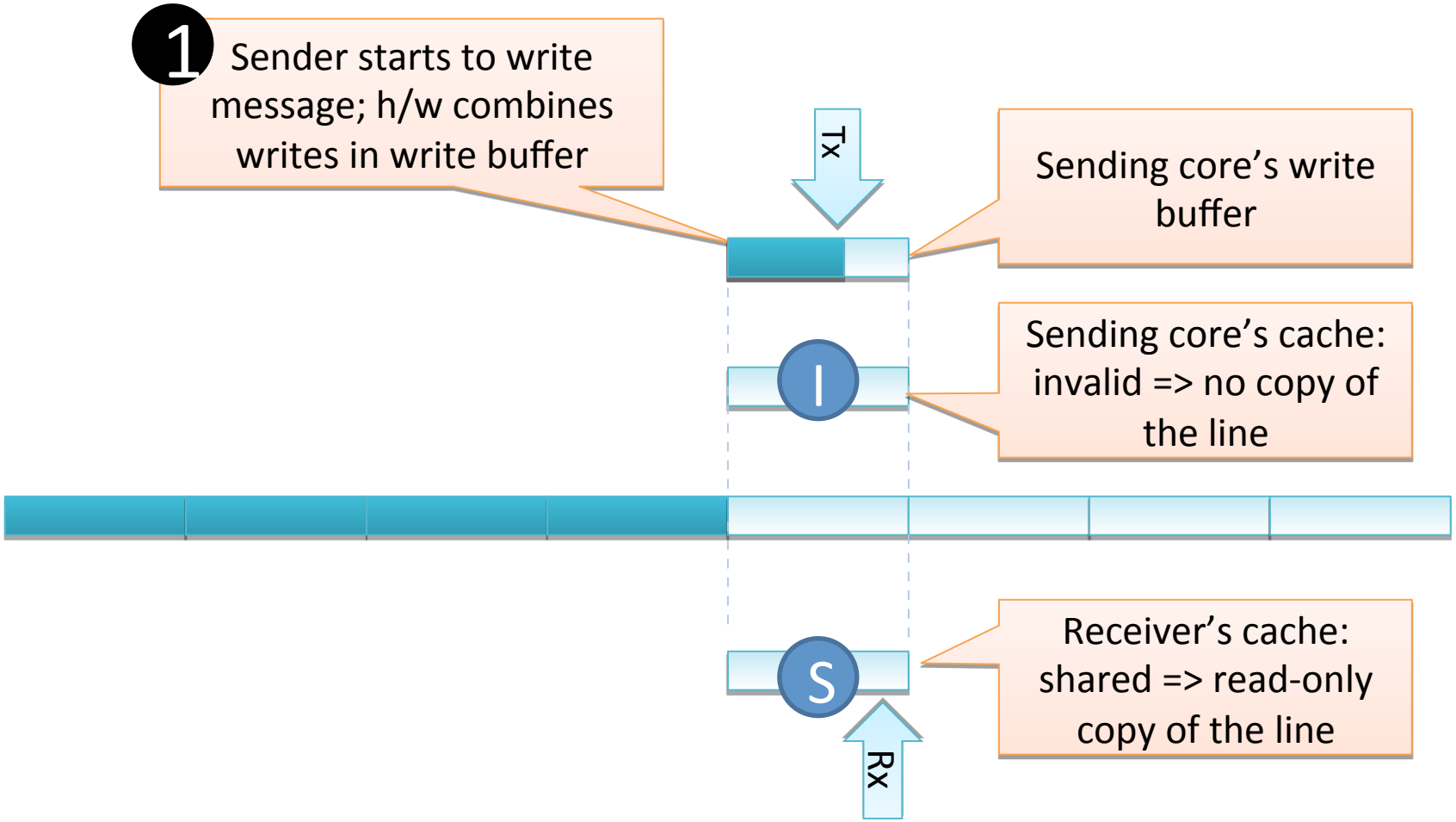
CC-UMP: cache-coherent user-space messaging



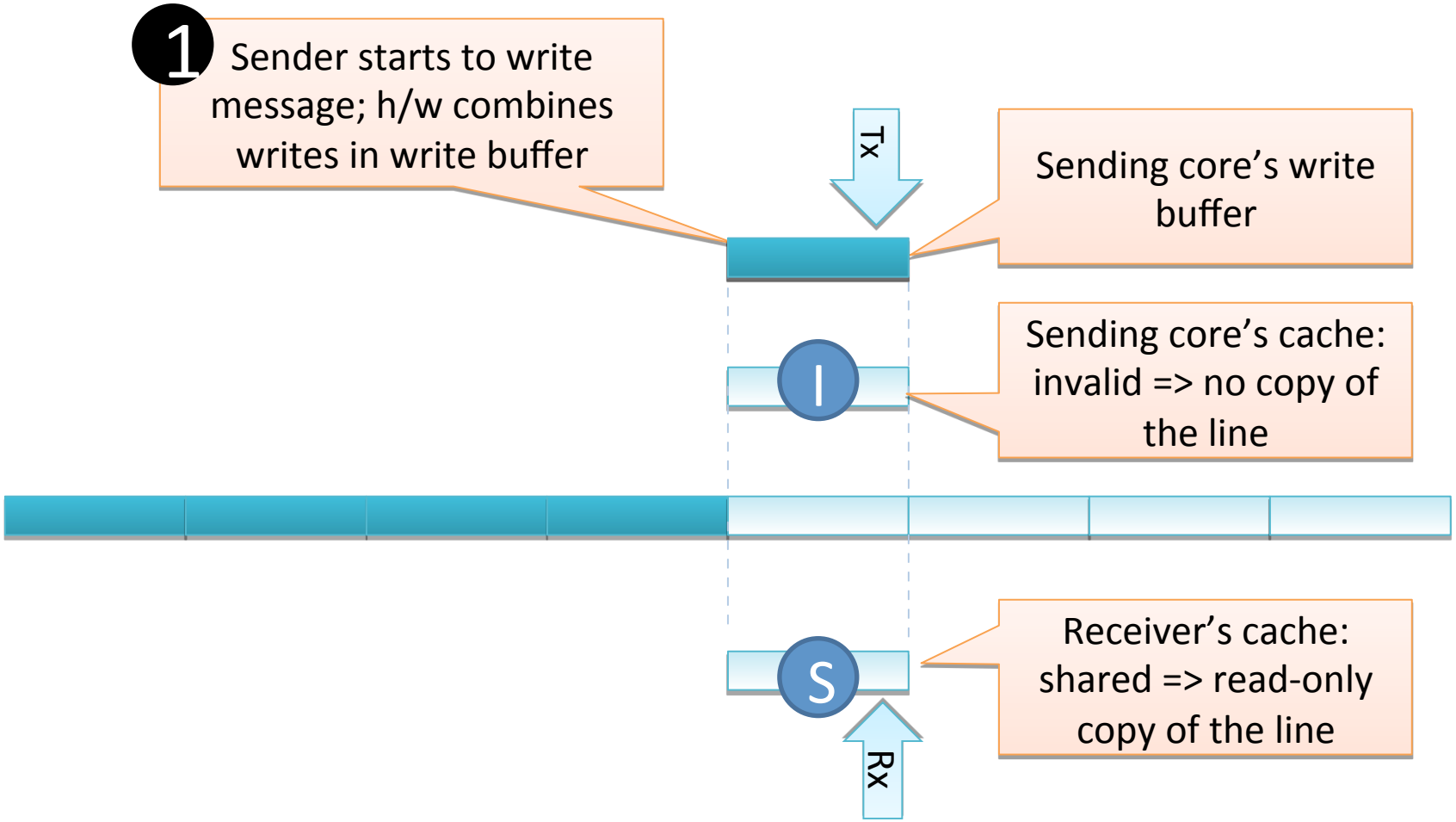
CC-UMP: cache-coherent user-space messaging



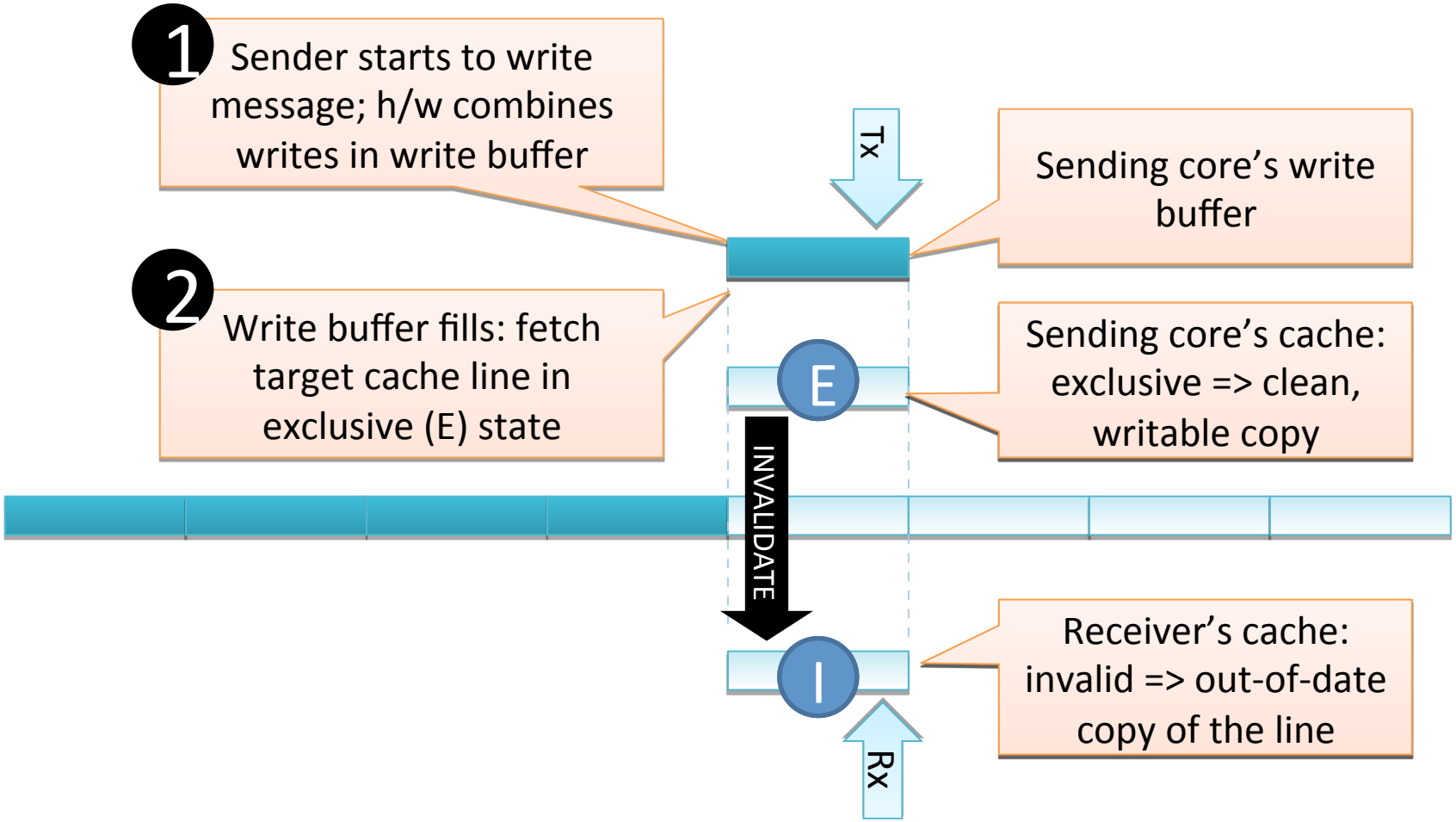
CC-UMP: cache-coherent user-space messaging



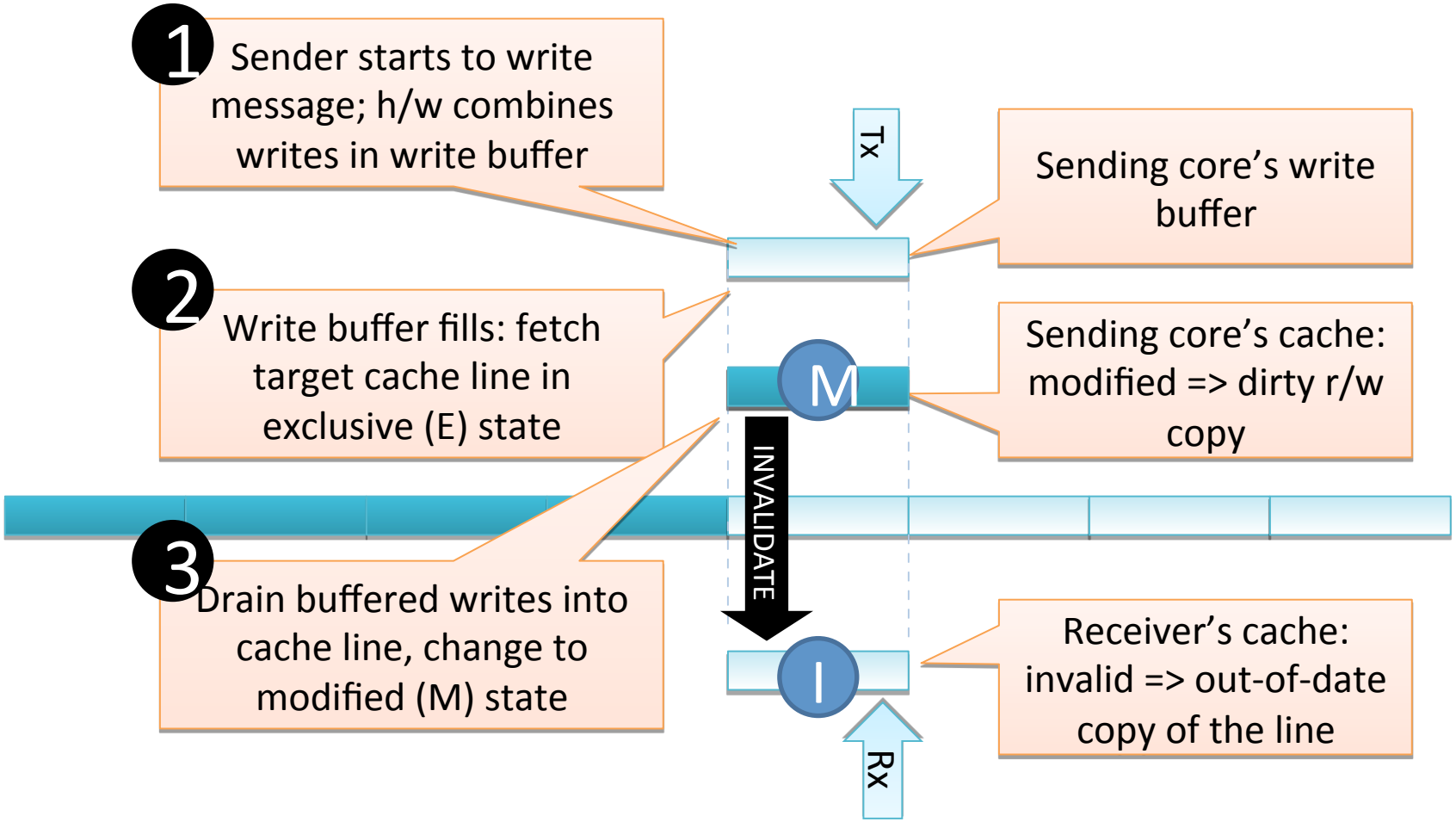
CC-UMP: cache-coherent user-space messaging



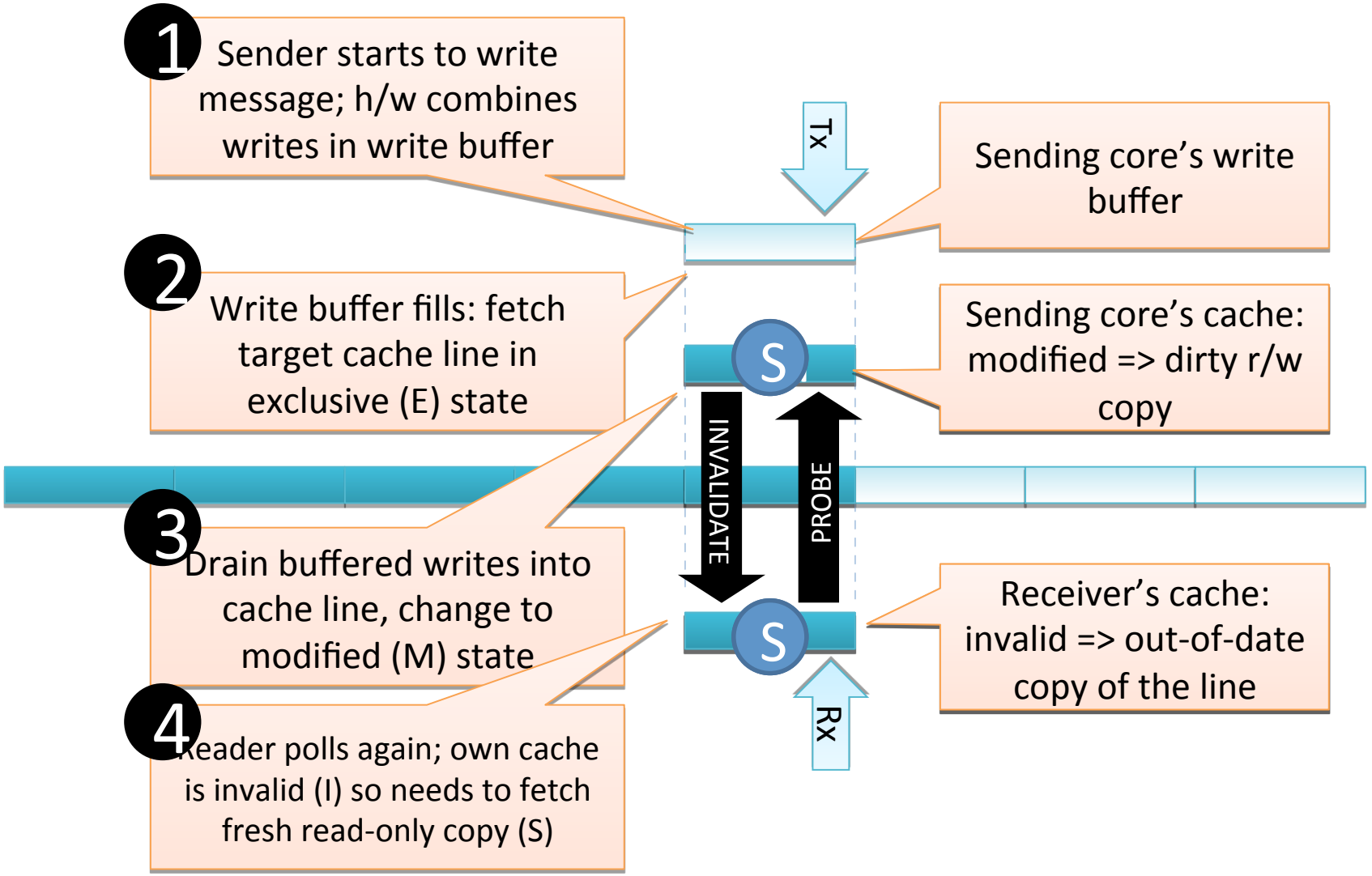
CC-UMP: cache-coherent user-space messaging



CC-UMP: cache-coherent user-space messaging

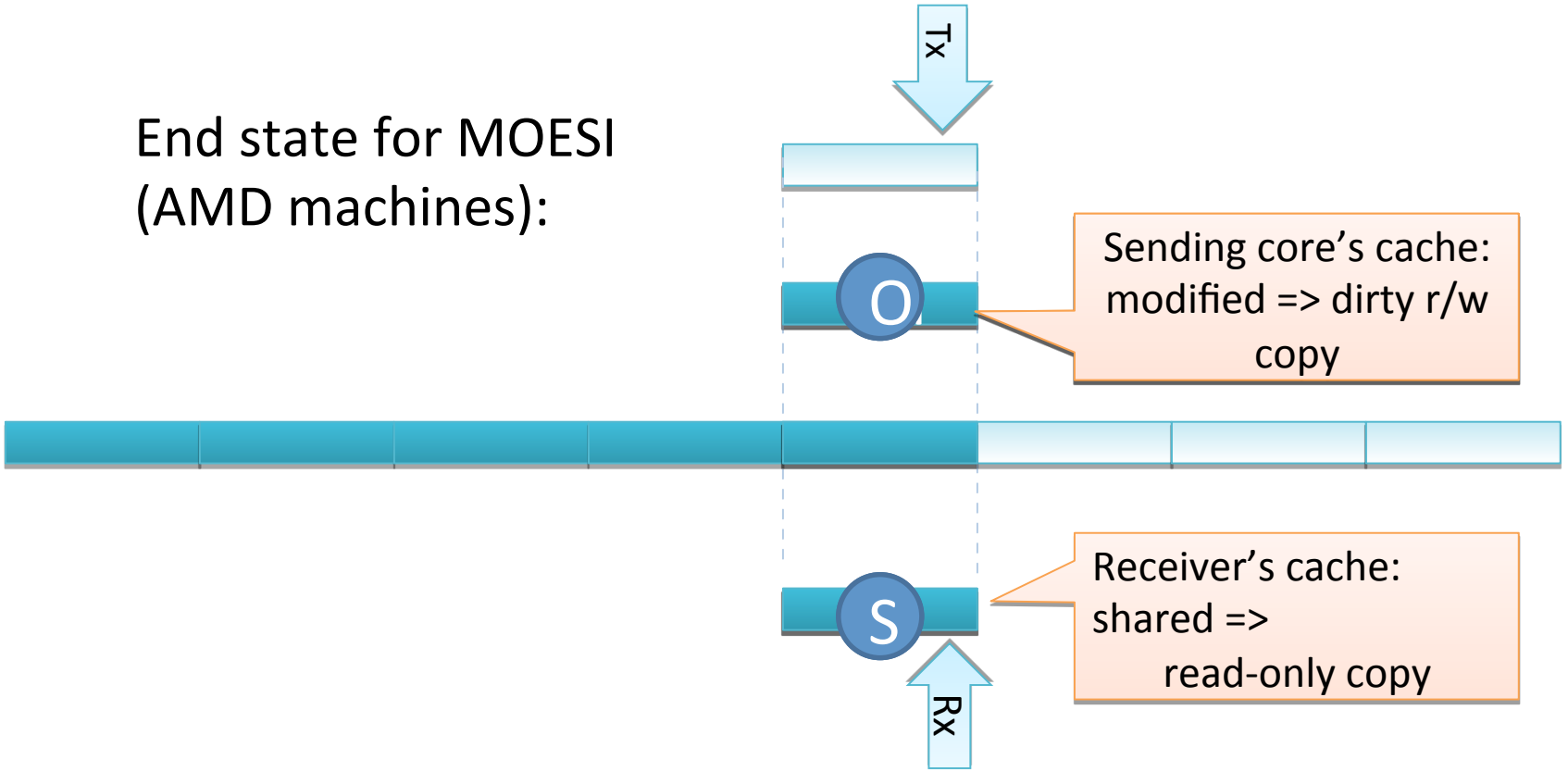


CC-UMP: cache-coherent user-space messaging



CC-UMP: cache-coherent user-space messaging

End state for MOESI (AMD machines):



Virtual Machine

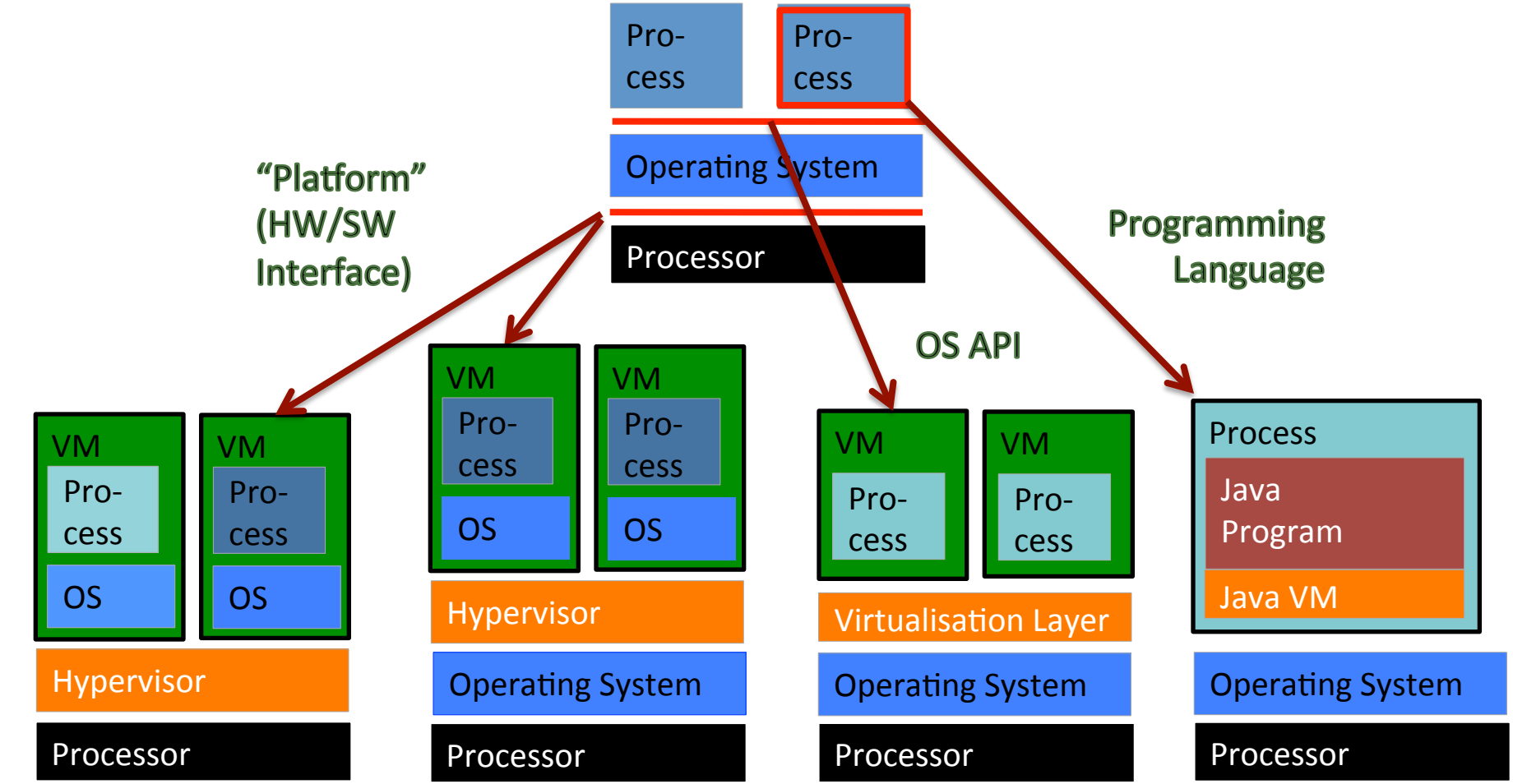
"A VM is an efficient, isolated duplicate of a real machine"

[Popek & Goldberg, 1974]

- **Duplicate:** VM behaves identically to real machine
  - Programs can't tell the difference
  - Caveats: resources, timing differences
- **Isolated:** several VMs execute without interference
- **Efficient:** speed close to that of real hardware
  - Requires that most instructions are executed directly by hardware

Hypervisor aka *virtual-machine monitor* (VMM): software implementing the VM

Types of Virtualisation



Platform VM or System VM

Type-1 "Native"

Type-2 "Hosted"

OS-level VM / Containers

Process VM

(Main topic of this lecture)

VMM History

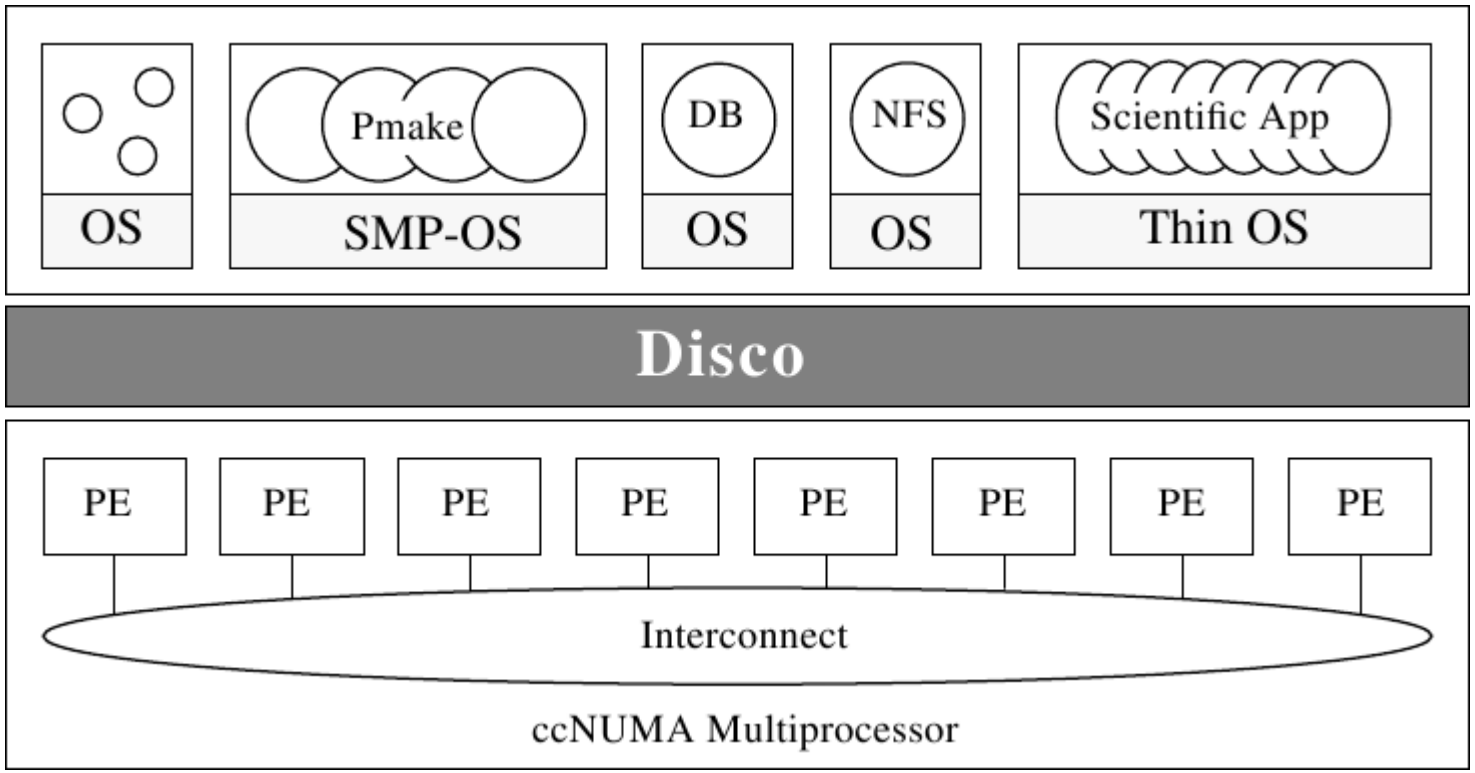
- Appeared concurrently with timesharing, late 60's
  - Multics and IBM's competitor (TSS) were both late
  - IBM hacked together a system known as CP/CMS
    - CP = control program (virtual machine monitor)
    - CMS was an existing single-user OS
  - Precursor of IBM mainframe VMMs
- Confined mostly to mainframes for decades
  - PC hardware was not efficiently virtualisable

Disco

Running commodity OSes on scalable multiprocessors [Bugnion et al., 1997]

- Context: ca. 1995, large ccNUMA multiprocessors appearing
- Problem: scaling OSes to run efficiently on these was hard
  - Extensive modification of OS required
  - Complexity of OS makes this expensive
  - Availability of software and OSes trailing hardware
- Idea: implement a scalable VMM, run multiple OS instances
- VMM has most of the features of a scalable OS, e.g.:
  - NUMA-aware allocator
  - Page replication, remapping, etc.
- VMM substantially simpler/cheaper to implement
- Run multiple (smaller) OS images, for different applications

Disco architecture



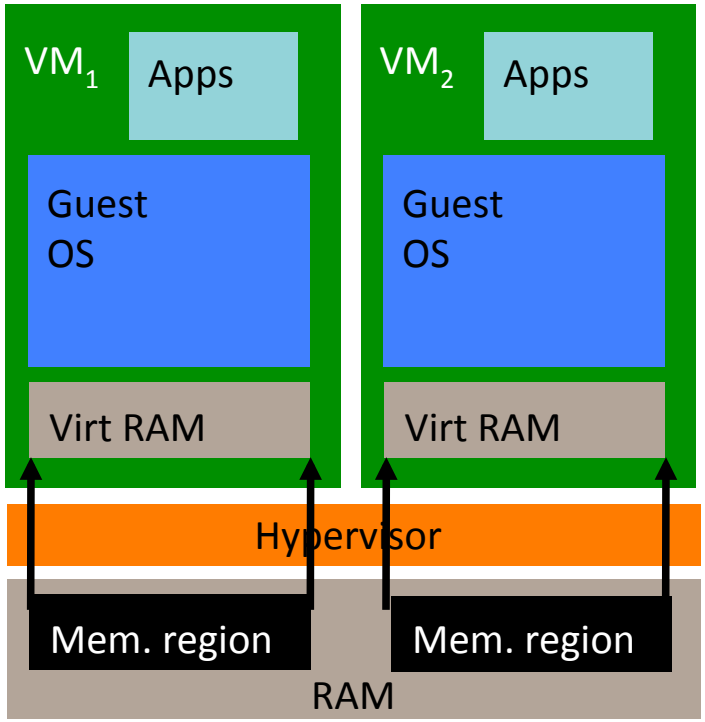
[Bugnion et al., 1997]

VMM History

- Disco authors went on to found VMware
  - Instead of pursuing scalability research
- Shipped v1 in 1999, showed it was practical on x86
  - Big challenge because x86 was not virtualizable
  - Research version on MIPS
- Hugely important today
  - OS security concerns
  - Cloud

Why Virtual Machines?

- Historically, used to share mainframes
  - Run several (even different) *guest OSes*
  - Each gets a **static** subset of physical resources
- Recent renaissance; many reasons:
  - Strong isolation (e.g. in the cloud)
  - Security (thinner interface)
  - Complete encapsulation of app/OS
    - Decouples OS from hardware
    - Migration/consolidation
    - Checkpointing, debugging
  - Run multiple OSes concurrently
- A band-aid for OS limitations?

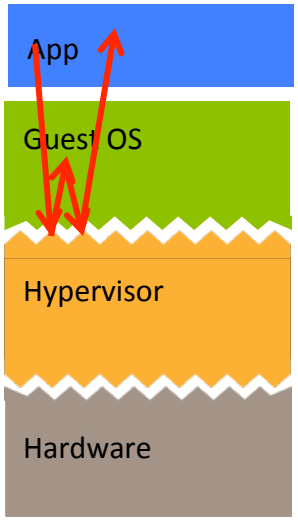


VMs in the cloud

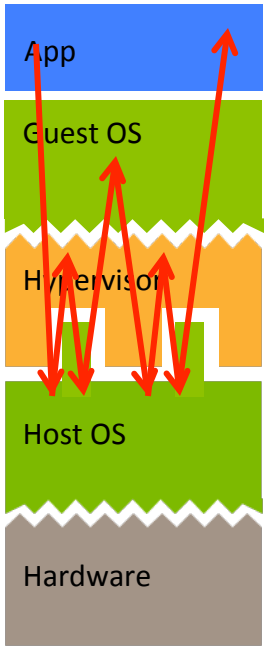
- Utility computing is an old idea, but VMs made it practical
- Infrastructure as a Service (IaaS)
  - Rent VMs from cloud provider
  - Provider increases utilisation by sharing hardware
  - On-demand provisioning
  - Dynamic load balancing / live migration

Native vs. Hosted VMM

Native/Classic/ Bare-metal/Type-I



Hosted/Type-II



- Hosted VMM beside native apps
  - Sandbox untrusted apps
  - Convenient for running alternative OS on desktop
  - Leverage host drivers
- Less efficient
  - Double mode switches
  - Double context switches
  - Host not optimised for exception forwarding

Virtualisation mechanics

- Trap and emulate
- Binary translation
- Paravirtualisation
- Hardware assistance

Trap and emulate virtualisation

- Traditional approach
  - Run guest OS and applications in unprivileged mode
  - Guest attempts to access physical resource
  - Hardware raises exception (trap), invokes hypervisor
  - Hypervisor emulates instruction
    - e.g. updates virtual CPU state
- Most instructions don't trap
  - Prerequisite for efficient virtualisation

Trap and emulate virtualisation

- Formalised by Popek & Goldberg (1974)
- Definitions:
  - Assume HW user/system mode (e.g. x86 ring 0)
  - *Privileged state* determines resource allocation
    - Privilege mode, address space, etc.
  - *Privileged instructions* trap in user but not system mode
    (e.g. `cli`, `sti`)
  - *Sensitive instructions* change or expose privileged state
    (e.g. `mov` to `CR3`, `int` , `iret`)
- Theorem:
  - Can construct an effective VMM if sensitive instructions are a subset of privileged instructions
  - Can also perform recursive virtualisation (run VMM in a VM)

Trap and emulate example: Virtual interrupts and CLI/STI

- Virtual machine monitor:
  - Controls hardware interrupt flag (IF)
    - Enabled during guest execution
  - Maintains virtual IF
  - Uses virtual IF to decide when to interrupt guest
- When guest executes CLI or STI
  - Protection violation trap, since guest is in user mode
  - VMM looks at virtual privilege level
    - If 0 (kernel mode), changes virtual IF
    - Else emulates virtual protection violation trap to guest kernel
- VMM must only expose virtual IF to guest

Trap and emulate limitations

- Problem 1: not all architectures are T&E virtualisable
  - e.g.: x86 CS exposes privilege level
  - `pushf` reveals real (host) interrupt flag
  - `popf` modifies flags, but silently ignores some in user mode
- Problem 2: high overhead/frequency of traps
- Solutions
  1. Rewrite problem instructions before they execute
     - Binary translation
     - (Classic) VMware
  2. Change guest OSes
     - "Paravirtualisation"
     - Xen, Denali, …
  3. Change the architecture
     - x86 virtualisation extensions, first shipped in 2005

Binary translation

- Basic idea:
  - Translate all instructions before they are executed
  - Most innocuous instructions are identical
  - Replace sensitive instructions (with traps, or other code)
- Very complex, especially for x86!
  - Code and data are intermingled
  - Code might be 64-, 32- or 16-bit
  - Introspection needs to show original code
  - Variable-length instructions
    - What happens if you branch to the “middle” of an instruction?
  - Self-modifying code
- Can also think of this as a JIT-compiling emulator
  - Can sometimes improve performance vs. native!

Binary translation example

Program:

```
int isPrime(int a) {
  for (int i = 2; i < a; i++) {
    if (a % i == 0) return 0;
  }
  return 1;
}
```

We're executing isPrime(49)…

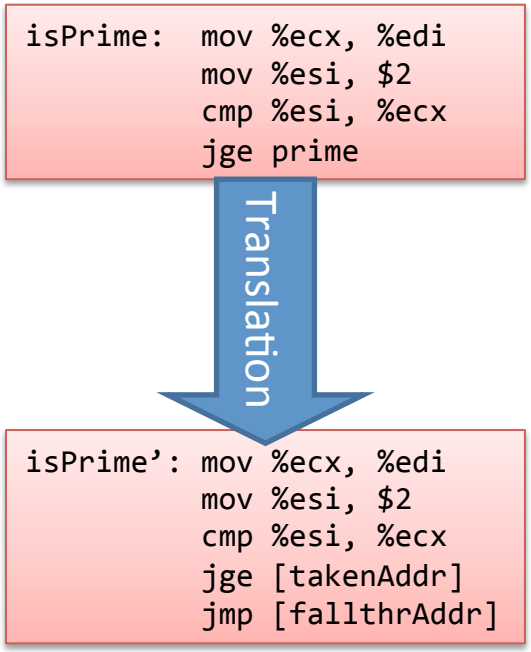
Compiles to:

```
isPrime: mov %ecx, %edi  ; %ecx = %edi
(a)
         mov %esi, $2    ; i = 2
         cmp %esi, %ecx  ; is i >= a?
         jge prime       ; jump if yes
nexti:   mov %eax, %ecx  ; set %eax = a
         cdq             ; sign-extend
         idiv %esi       ; a % i
         test %edx, %edx ; zero
remainder?
         jz notPrime     ; jump if yes
         inc %esi        ; i++
         cmp %esi, %ecx  ; is i >= a?
         jl nexti        ; jump if no
prime:   mov %eax, $1    ; return in
%eax
         ret
notPrime: xor %eax, %eax ; %eax = 0
         ret
```

"Basic block"

[Adams and Agesen, 2006]

Binary translation example



- Translate and execute first basic block
  - Branch targets point back into translator
- Translate next (reached) basic block, …
- Chain translations
  - Update jump targets
  - Elide fall-through jumps

Binary translation example

```
isPrime':  mov %ecx, %edi
           mov %esi, $2
           cmp %esi, %ecx
           jge [takenAddr]
                          ; fall-thru into next CCF
nexti':    mov %eax, %ecx
           cdq
           idiv %esi
           test %edx, %edx
           jz notPrime'
                          ; fall-thru into next CCF
           inc %esi
           cmp %esi, %ecx
           jl nexti'
           jmp [fallthrAddr3]

notPrime': xor %eax, %eax
           pop %r11        ; RET
           mov %gs:0xff39eb8(%rip), %rcx ; spill rcx
           movzx %ecx, %r11b
           jmp %gs:0xfc7dde0(8*%rcx) ; (restores rcx)
```

- Untaken branches never translated
- I-cache locality good
- GS refers to VMM state
- CALL/RET are complex
  - Stack contains native addresses
- 64-bit registers handy for a 32-bit guest

Paravirtualisation

– Denali [Whitaker et al, OSDI'02] and Xen [Barham et al, SOSP'03]

– Basic idea: "enlighten" guest OS to run in VM

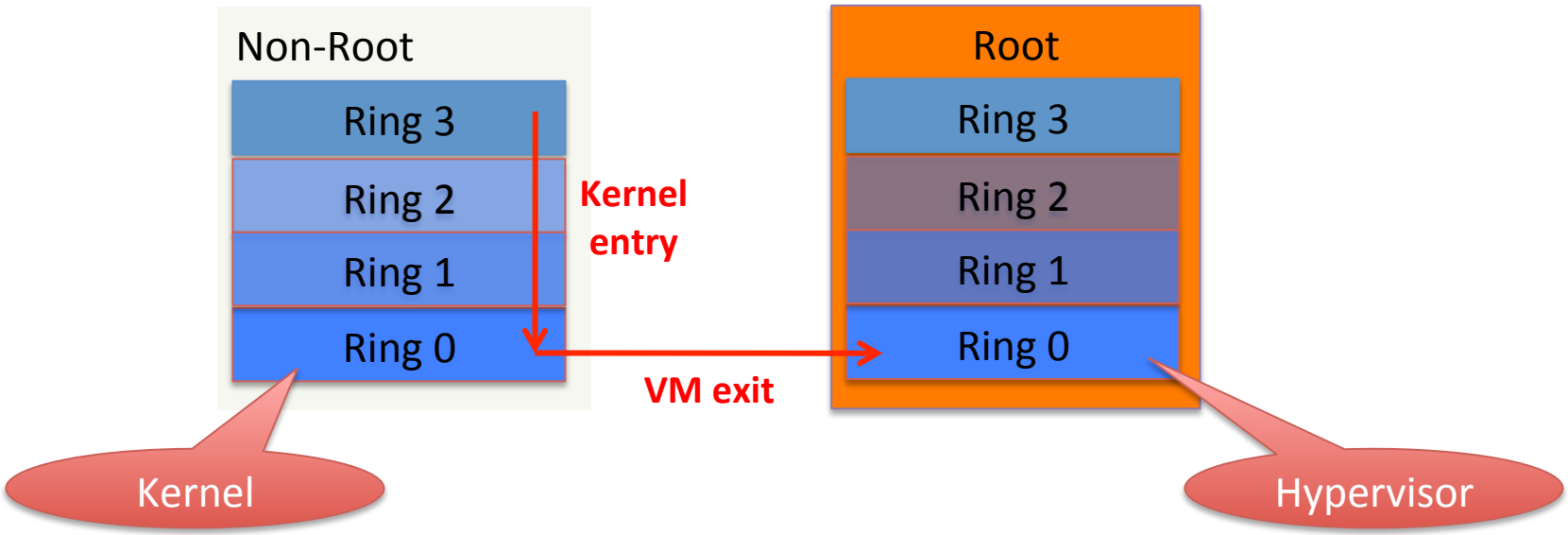
  - Augment processor ISA with explicit hypercalls
  - Remove sensitive instructions
  - Reduce number of traps

– Generally outperforms trap and emulate, binary translation

- Requires source modifications for each guest / host

Hardware extensions

- Intel VT / AMD SVM (circa 2006)
- Changed architecture to enable virtualisation
  - New privilege mode (guest mode / “root” mode)
  - New instructions (vmrun, vmexit) – a bit like iret / syscall
  - New data structure: VM control block (for guest state)

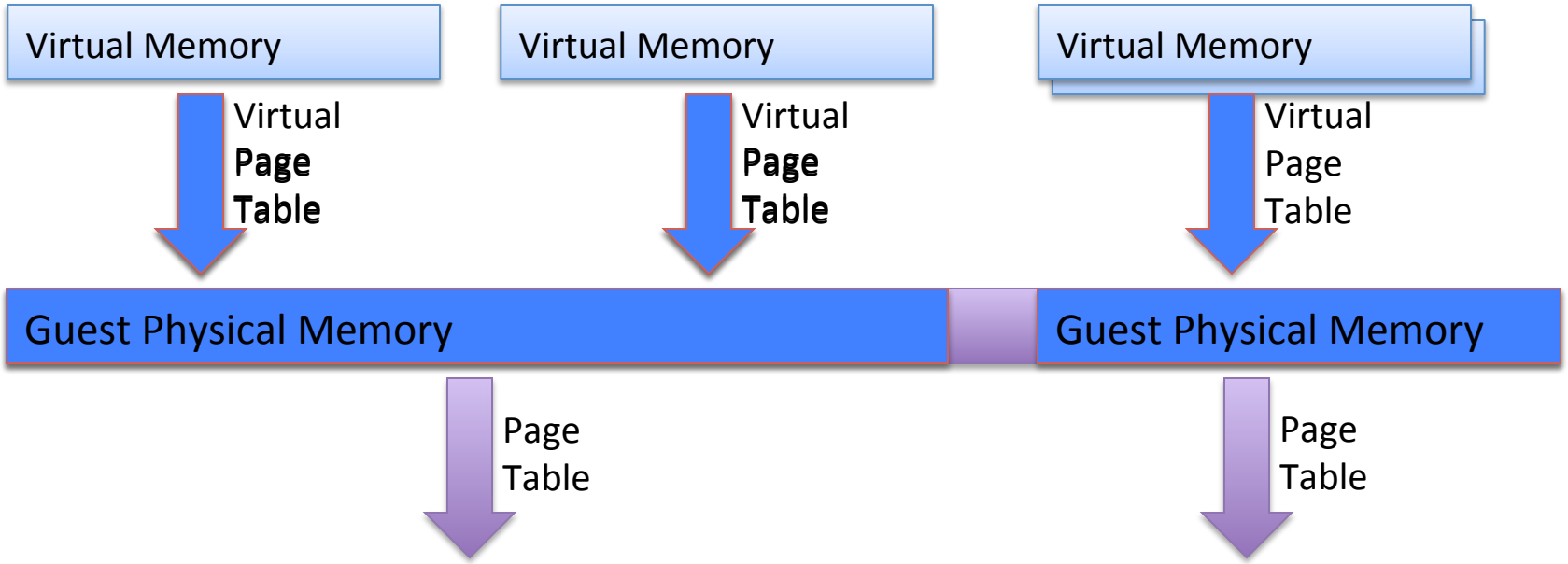


Virtualisation overheads

- VMM must maintain virtualised privileged machine state
  - processor status
  - addressing context
  - device state
- VMM needs to emulate privileged instructions
  - translate between virtual and real privileged state
  - e.g. guest ↔ real page tables
- Virtualisation traps are expensive
  - >1000 cycles on some Intel processors!
  - But improving (Haswell round-trip < 500 cycles)
- Some OS operations involve frequent traps
  - STI/CLI for mutual exclusion
  - frequent page table updates during fork()

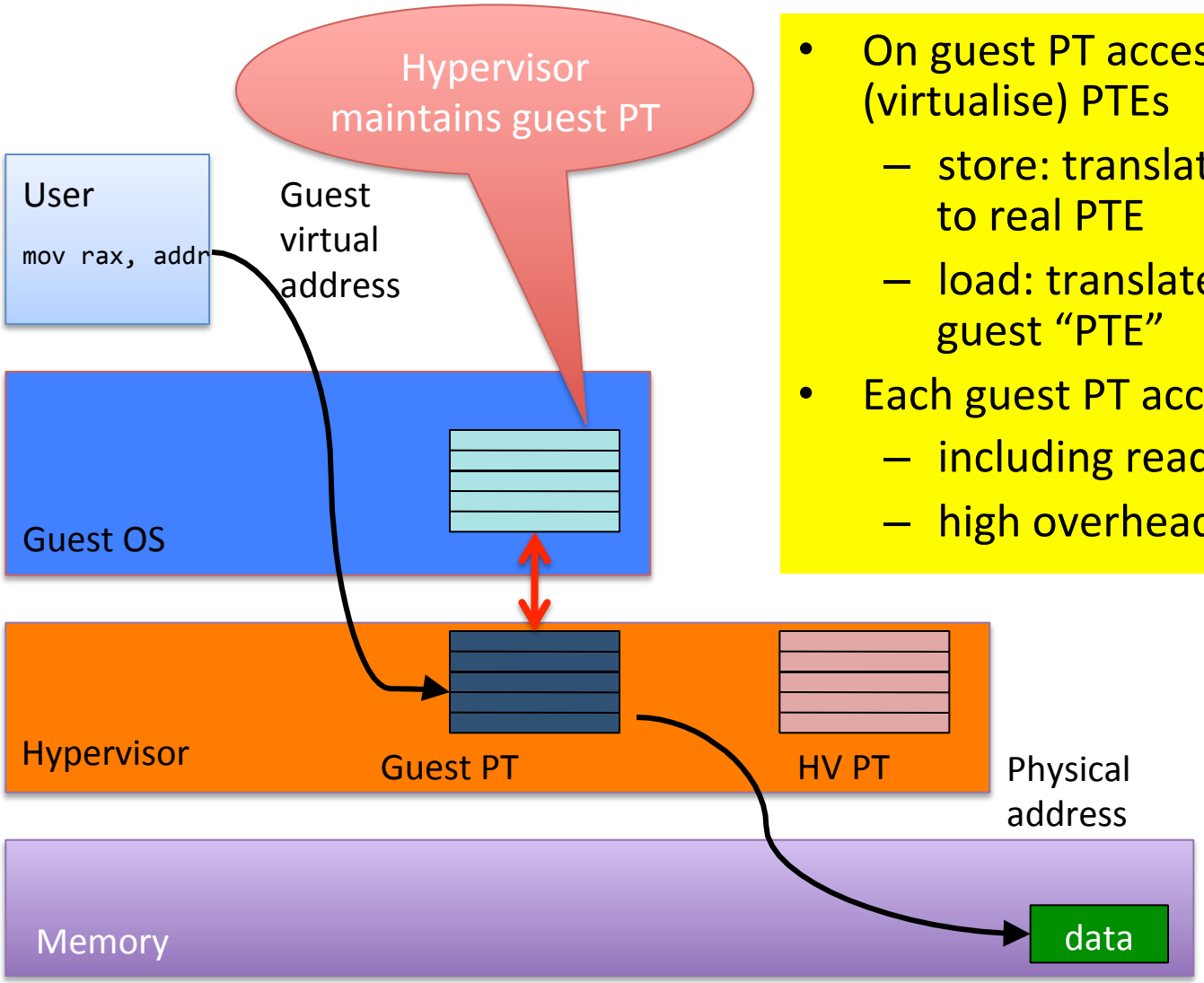
Memory Virtualization

VM address translation



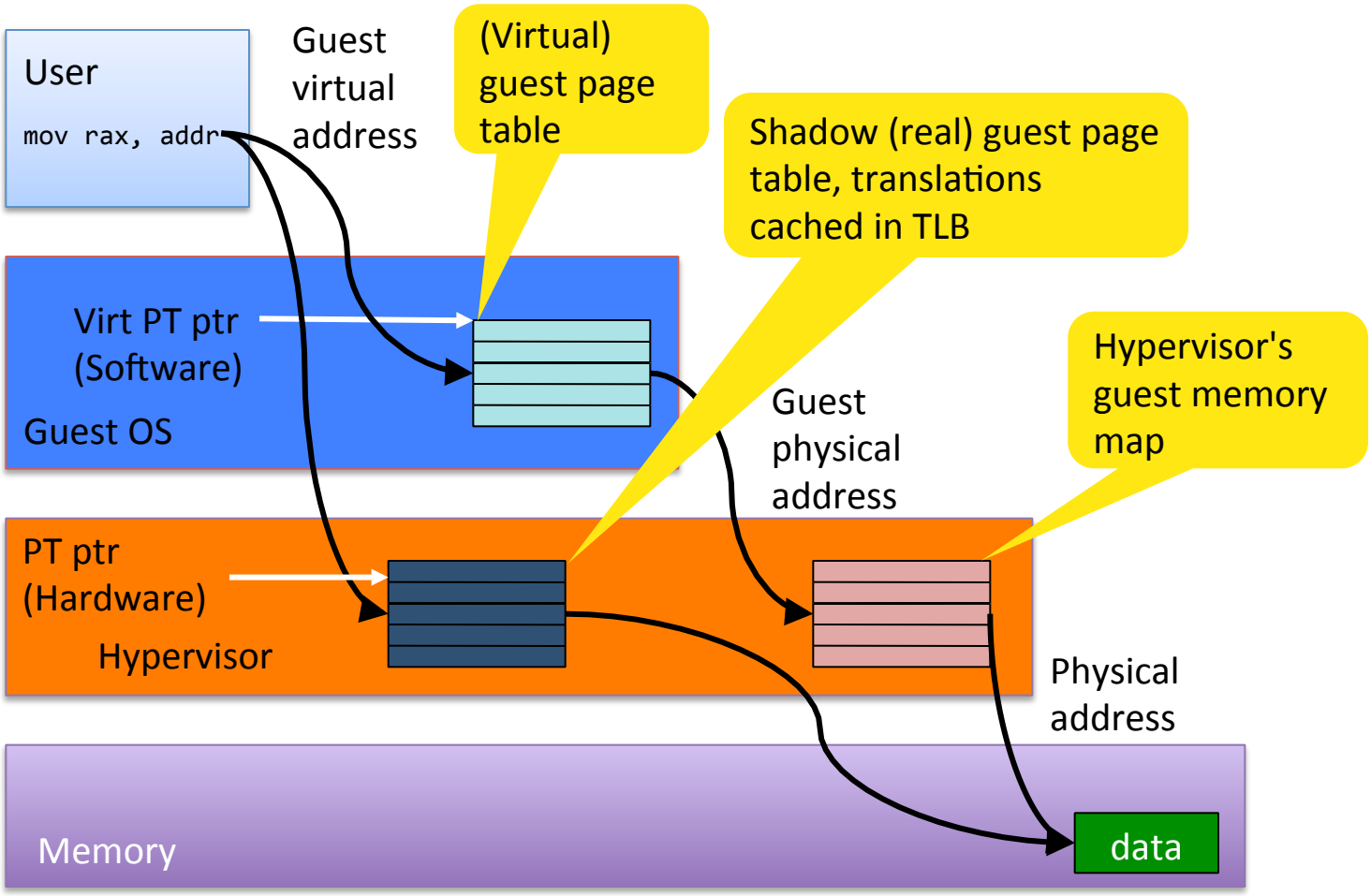
Must implement with single MMU translation!

Real Guest PT

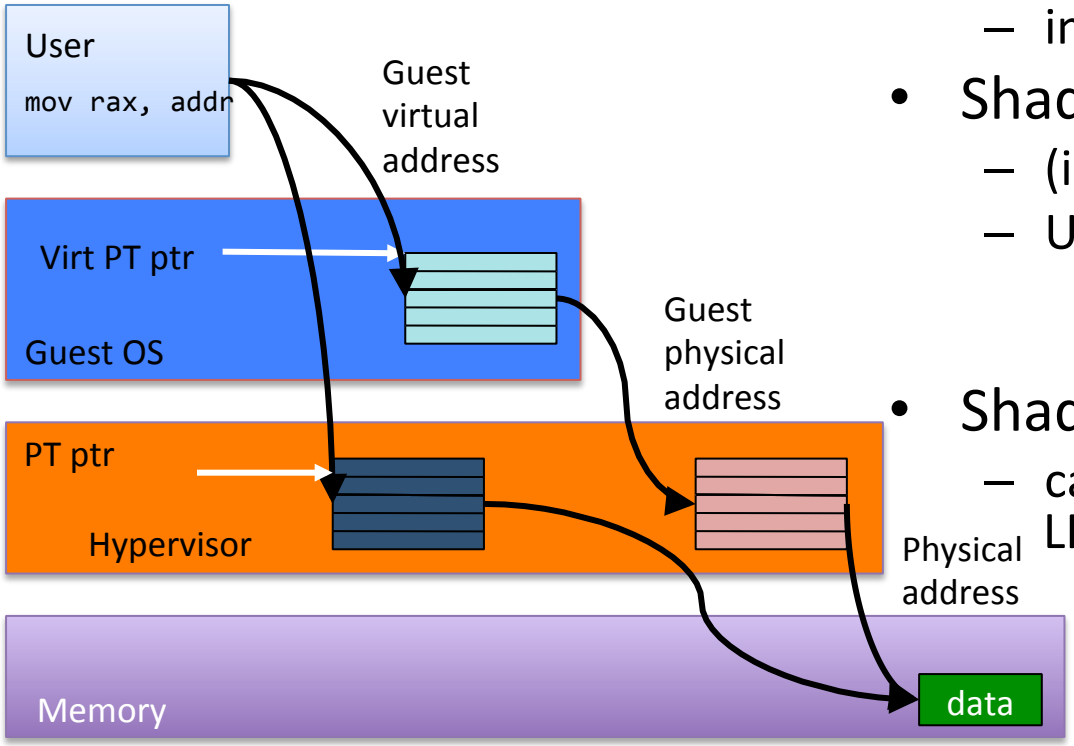


- On guest PT access must translate (virtualise) PTEs
  - store: translate guest "PTE" to real PTE
  - load: translate real PTE to guest "PTE"
- Each guest PT access traps!
  - including reads
  - high overhead

Shadow Page Table



Shadow Page Table

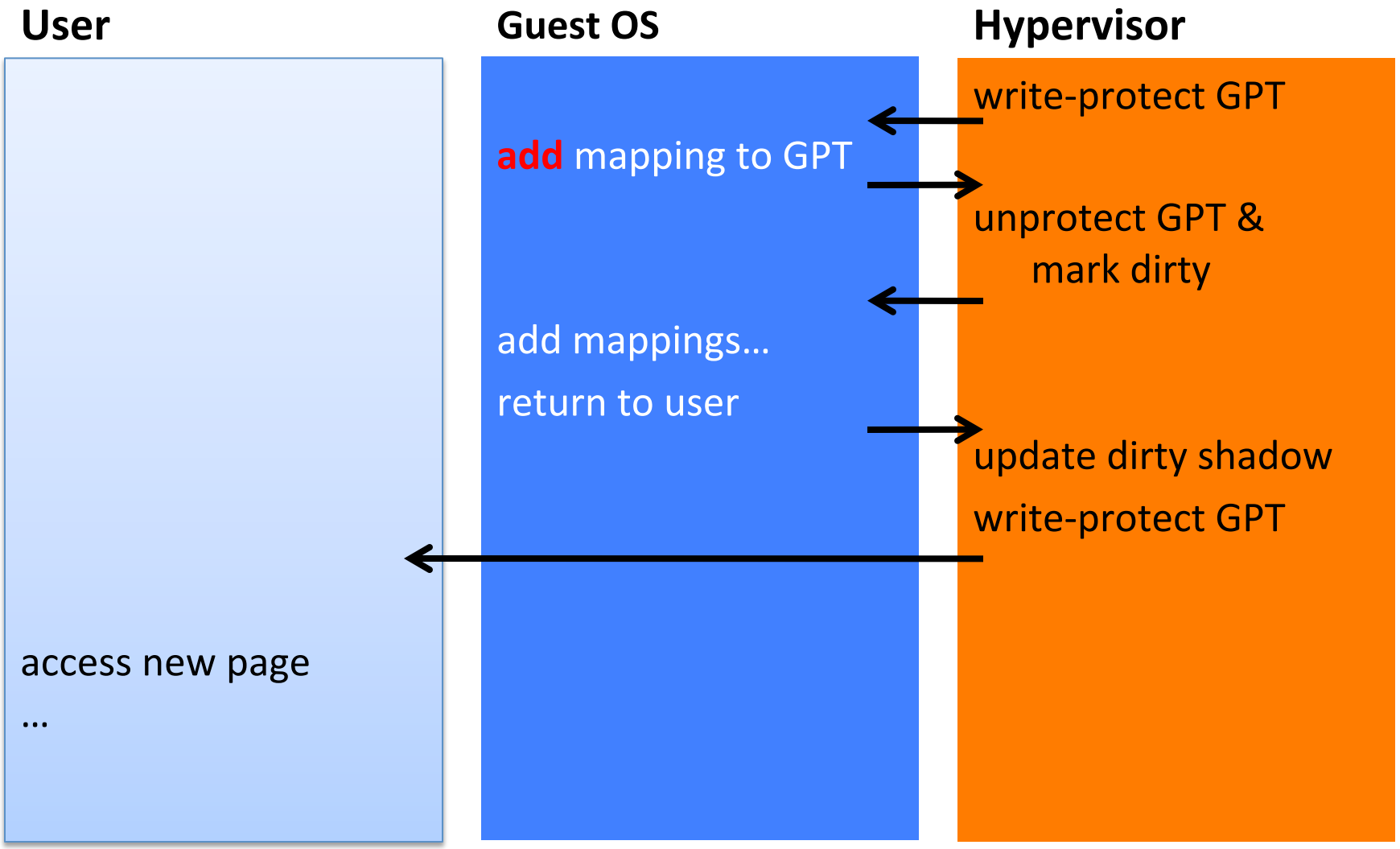


- Hypervisor must shadow (virtualise) all guest PT updates:
  - trap when guest writes to PT
  - translate guest PA (virtual) using guest memory map
  - insert translated PTE in shadow PT
- Shadow PT has TLB semantics
  - (i.e. weak consistency)
  - Update at synchronisation points:
    - page faults
    - TLB flushes
- Shadow PT as virtual TLB
  - can be incomplete: LRU translation cache

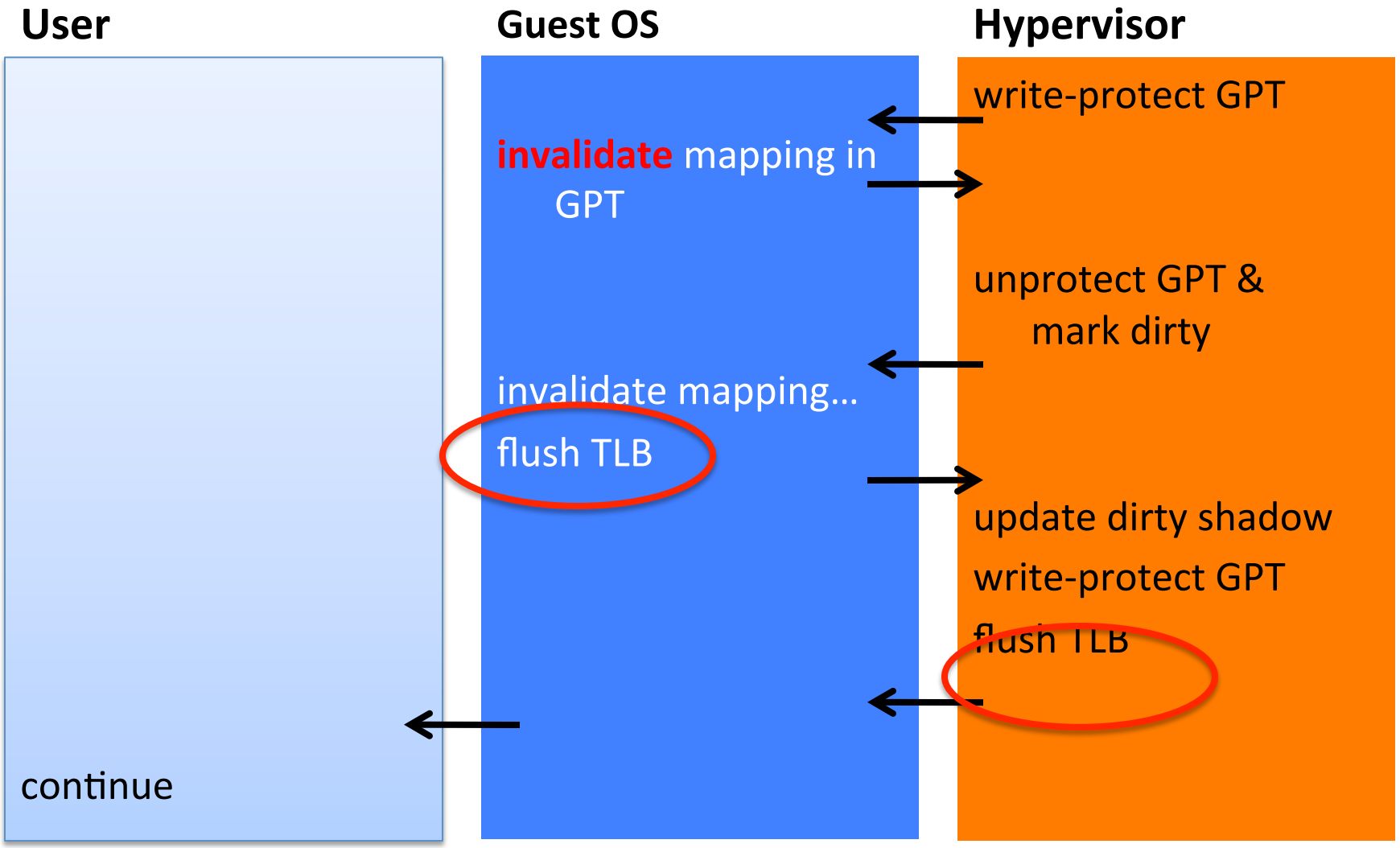
Segment Table		Page Table A		Page Table B	
0	Page Table A	0	0002	0	0001
1	Page Table B	1	0006	1	0004
x	(rest invalid)	2	0000	2	0003
		3	0005	x	(rest invalid)
		x	(rest invalid)		

Segment Table		Page Table K	
0	Page Table K	0	BEEF
x	(rest invalid)	1	F000
		2	CAFE
		3	3333
		4	(invalid)
		5	BA11
		6	DEAD
		7	5555
		x	(rest invalid)

Lazy Shadow Update



Lazy Shadow Update

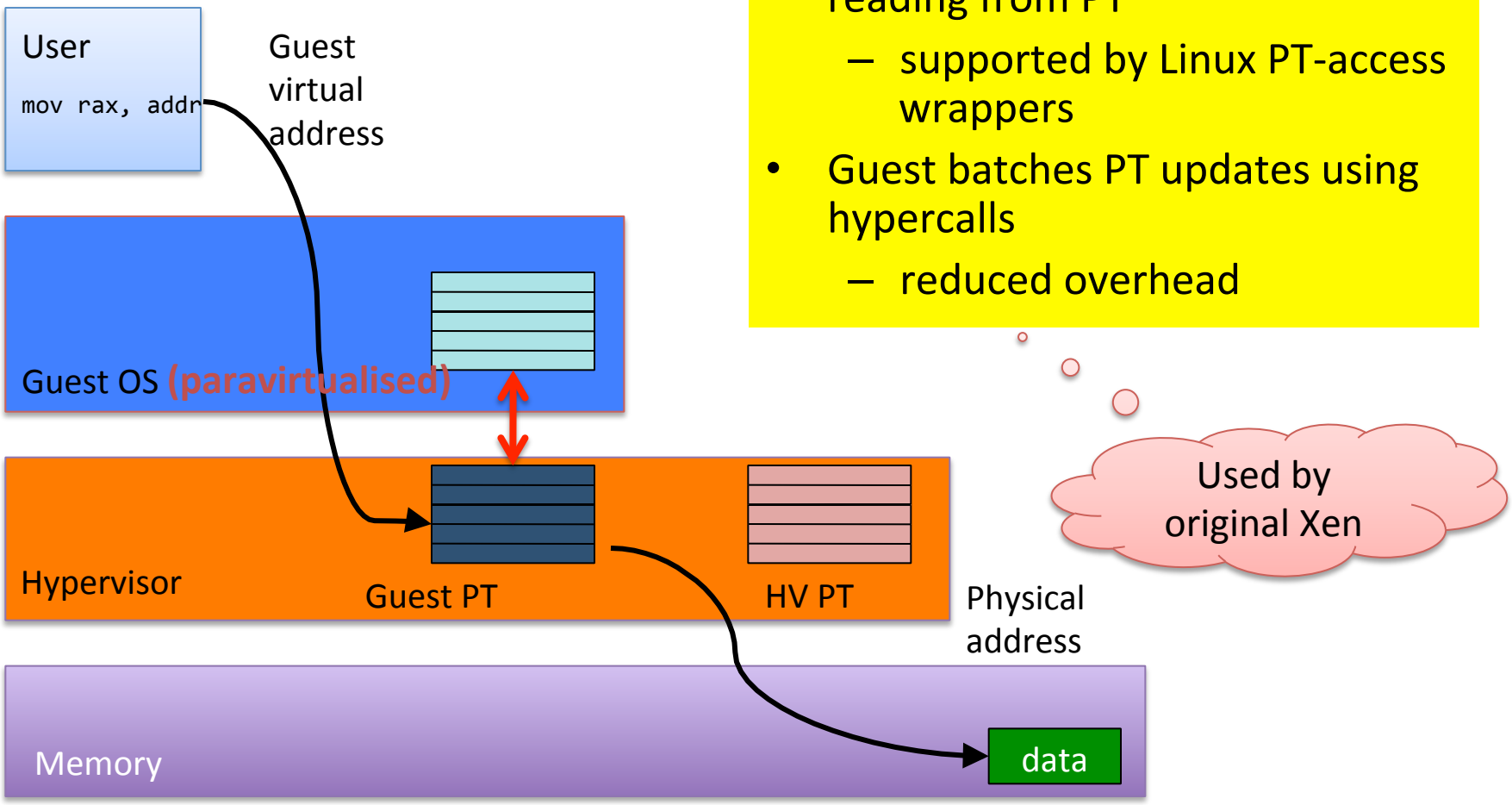


Paravirtualization

- Impure virtualisation methods (aka paravirtualization) enable new optimisations
  - avoid traps through ability to control the ISA
  - changed contract between guest and hypervisor
- Example: para-virtualised guest page table

Paravirtual Guest PT

- Guest translates PTEs itself when reading from PT
  - supported by Linux PT-access wrappers
- Guest batches PT updates using hypercalls
  - reduced overhead

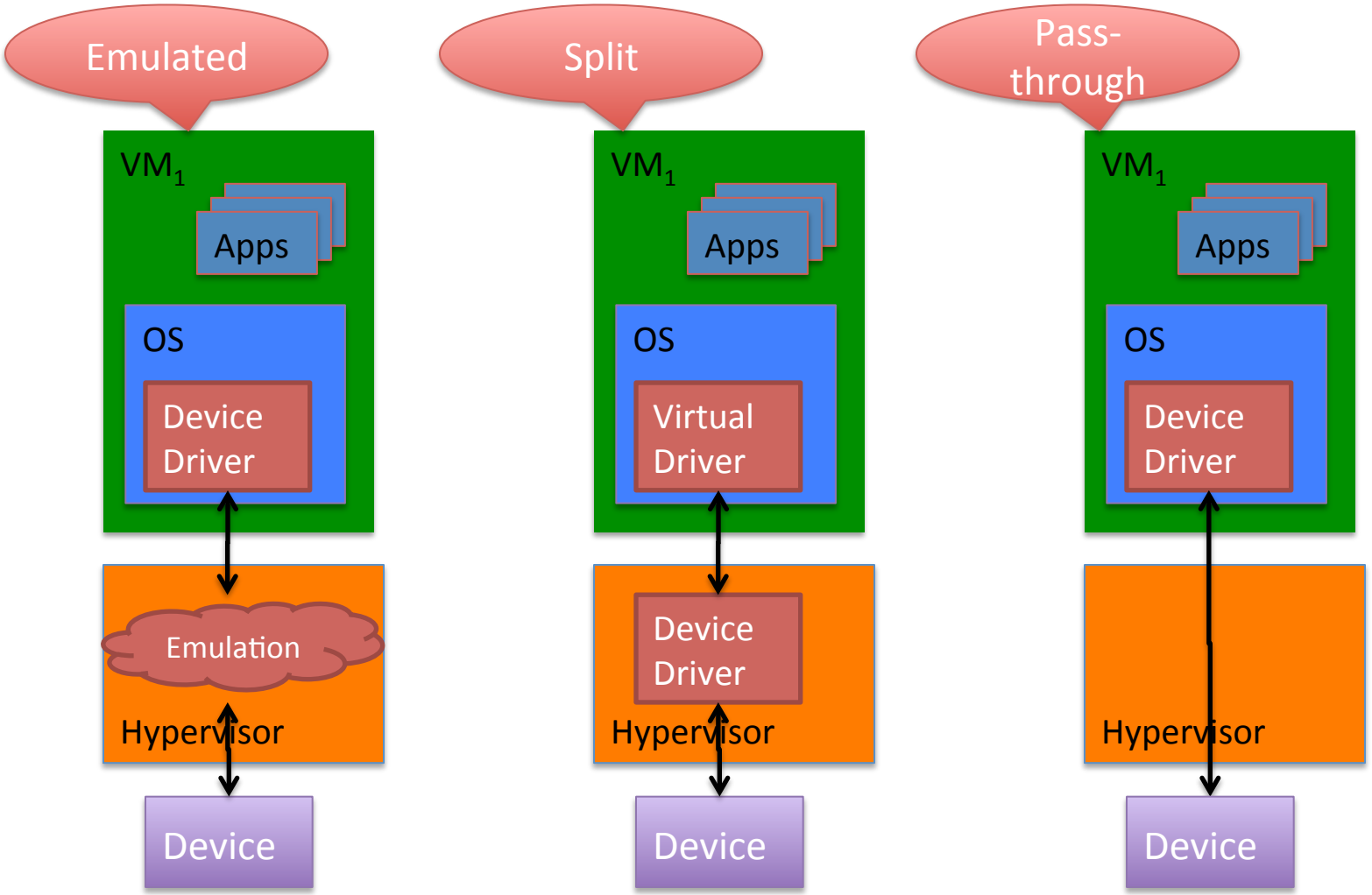


Problems with shadow paging

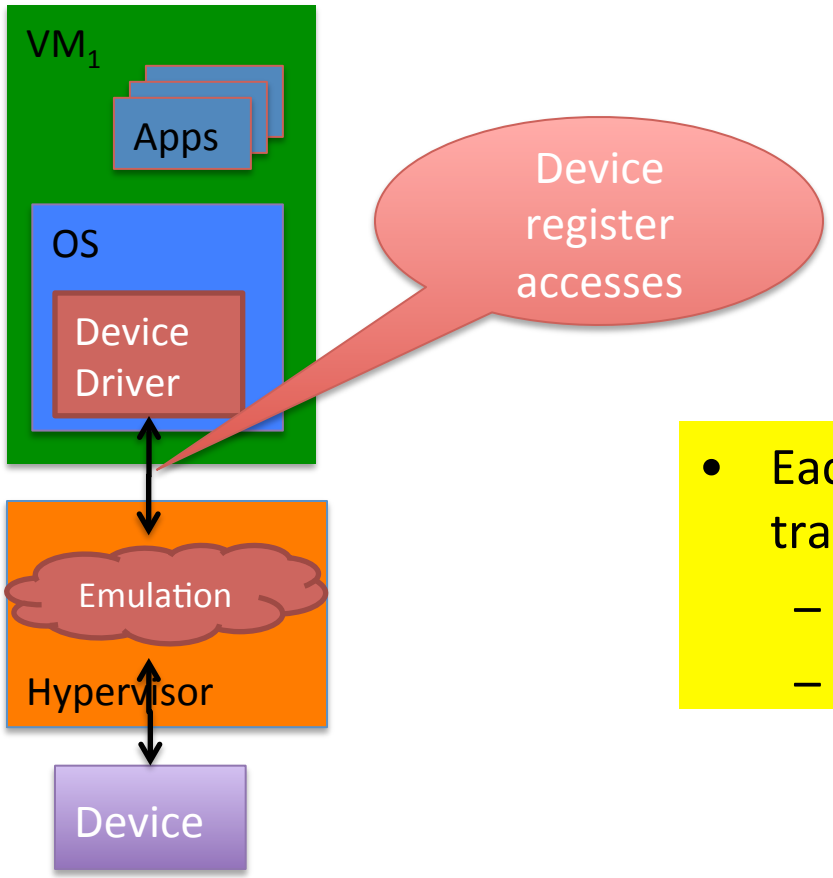
- Significant overhead for PT updates
  - Workload dependent performance impact
- Lots of complexity in the hypervisor
- Solution: more hardware!
  - Intel / AMD introduced *nested paging* circa 2008
- Hardware walks both guest and host page table
  - A bit scary: for x64 (4-level PT), a single memory access might require 16 page table fetches!
  - In practice, they are cached in the TLB

Device Virtualization

VMM Device Models

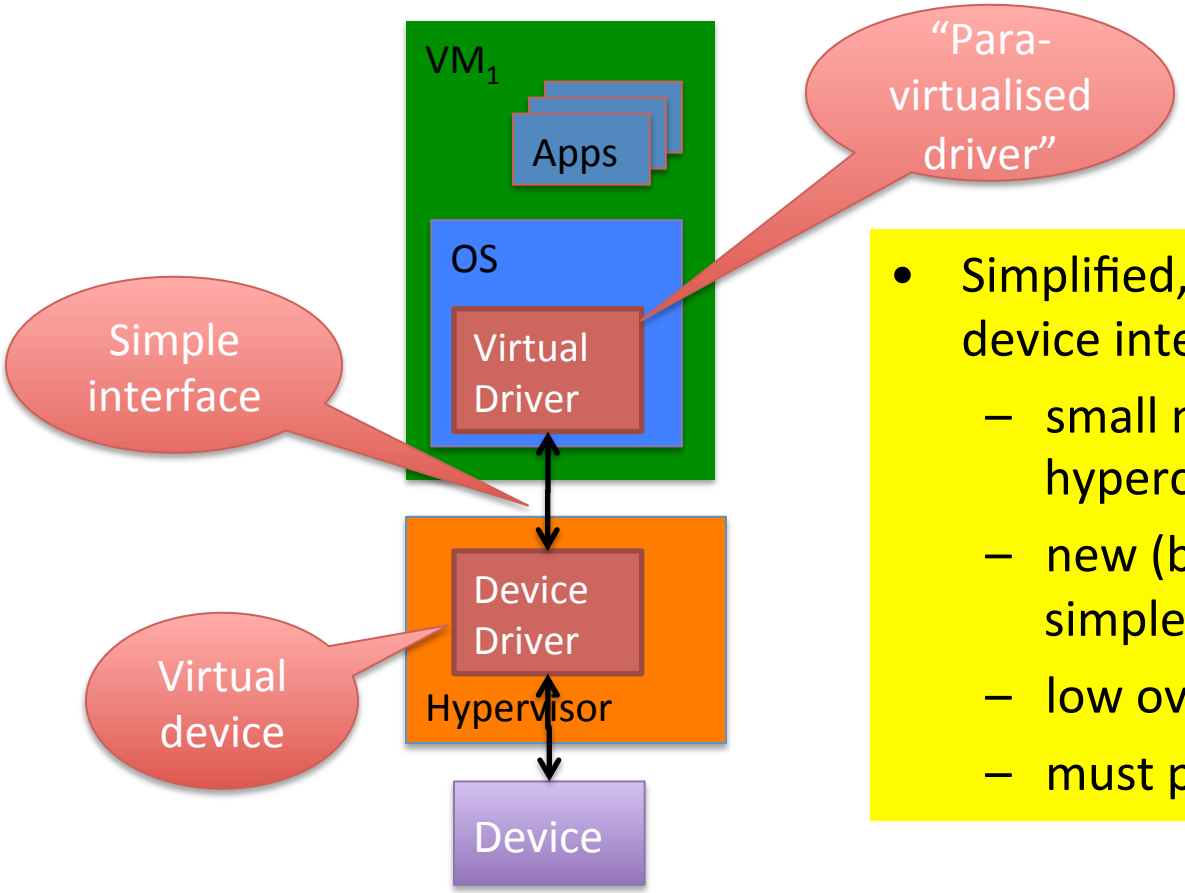


Emulated Device



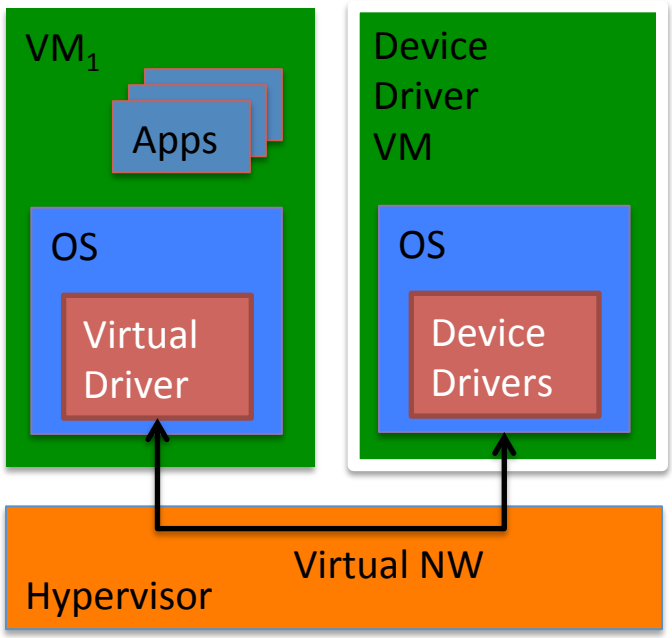
- Each device access must be trapped and emulated
  - unmodified native driver
  - high overhead!

Split Driver (Xen speak)



- Simplified, high-level device interface
  - small number of hypercalls
  - new (but very simple) driver
  - low overhead
  - must port drivers

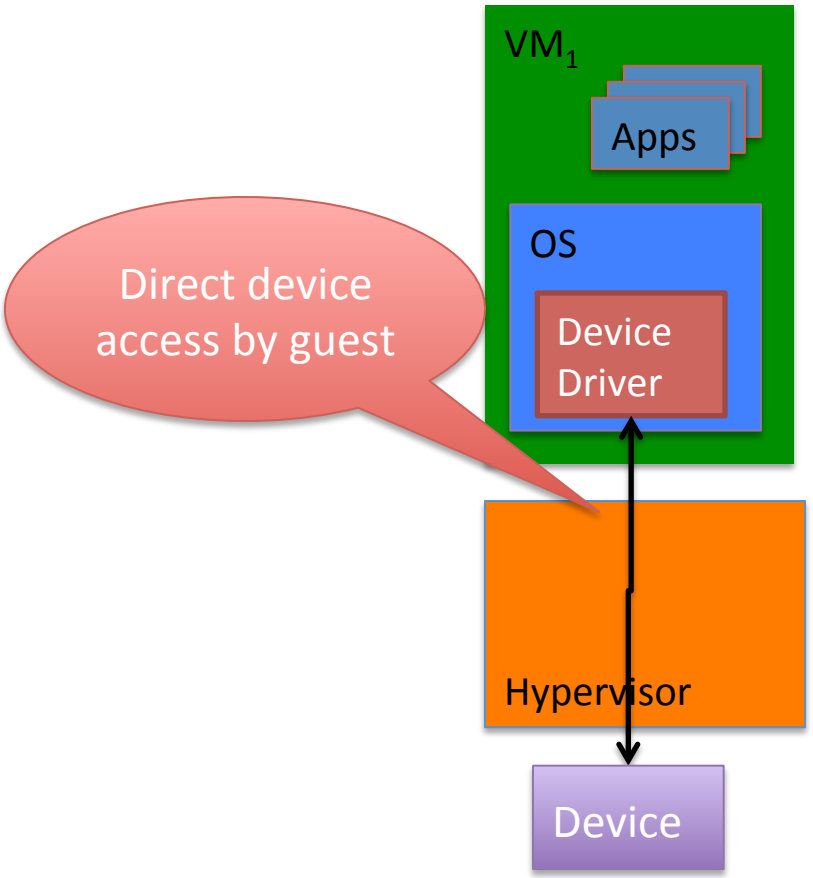
Driver OS (Xen Dom0)



- Leverage Driver-OS native drivers
  - no driver porting
  - must trust complete Driver OS!
  - huge TCB!

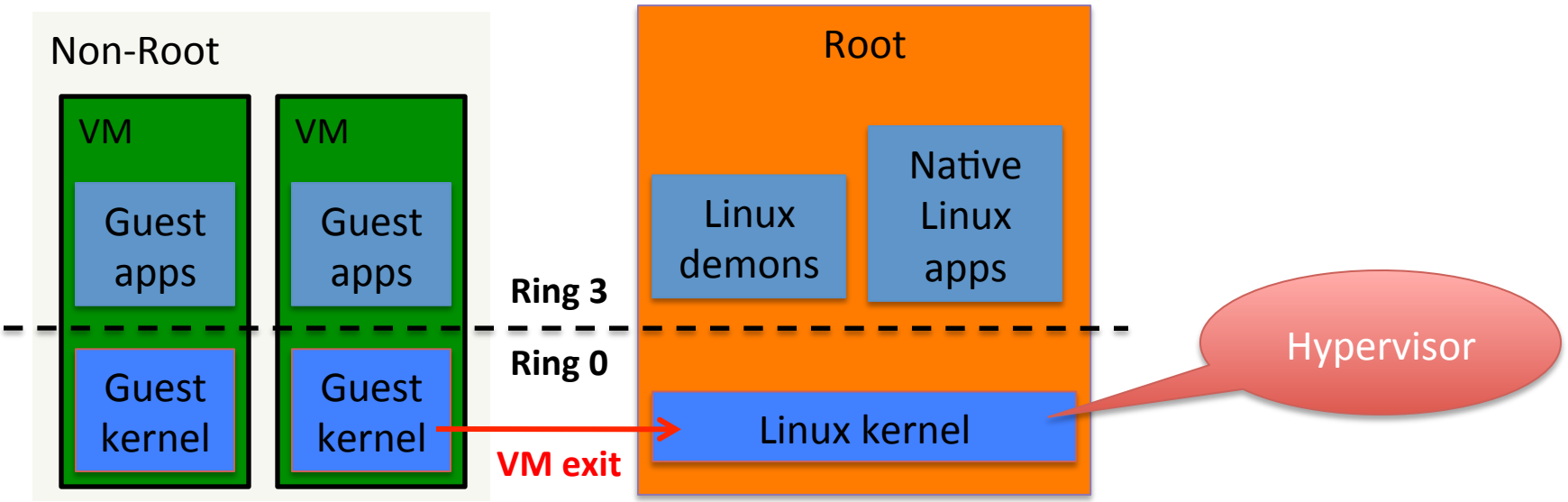
Pass-Through Driver

- Unmodified native driver
- Only secure with hardware support
  - I/O MMU and VM-safe devices



Hybrid Hypervisor OSes

- Idea: turn standard OS into hypervisor
  - … by running in VT-x root mode
  - eg: KVM (“kernel-based virtual machine”)
- Can re-use Linux drivers etc
- *Huge trusted computing base!*
- Often falsely called a Type-2 hypervisor



Multilevel Memory Allocation

- Guest OS has fixed memory size, set at boot
- Host VMM multiplexes memory between VM's
  - Host VMM evicts an unused page
  - Guest OS flushes unused dirty page to virtual disk
  - Host VMM brings back unused page to write to virtual disk

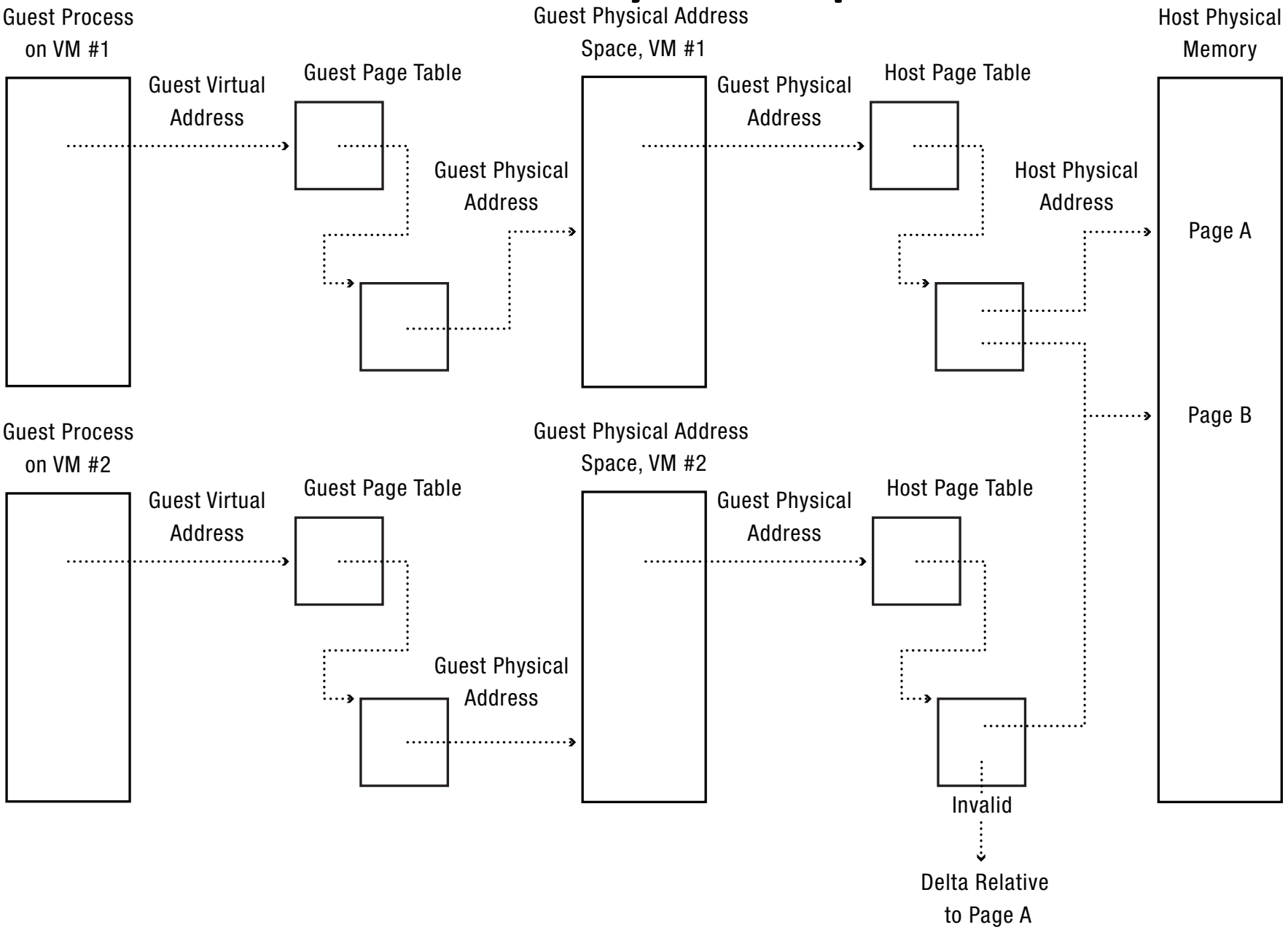
Multilevel Memory Allocation

- Balloon driver: load a dummy I/O device driver into the guest OS to allocate/deallocate frames
- Paravirtualization: modify guest OS to handle dynamic resource allocation
- What about multiplexing guest OS processors?
  - Balloon driver?
  - Scheduler activations?

Memory Compression

- Guest OS's are likely to contain copies of the same page
  - code pages if running same linux version
  - App pages if running the same applications
- ESX solution:
  - Keep hash of recent version of every page
  - Update randomly
  - If match, remap to same page, copy-on-write

VMM memory compression



Fun and Games with VMMs

- Time-travelling virtual machines [King '05]
  - debug backwards by replay VM from checkpoint, log state changes
- SecVisor: kernel integrity by virtualisation [Seshadri '07]
  - controls modifications to kernel (guest) memory
- Overshadow: protect apps from OS [Chen '08]
  - make user memory opaque to OS by transparently encrypting
- Turtles: Recursive virtualisation [Ben-Yehuda '10]
  - virtualise VT-x to run hypervisor in VM
- CloudVisor: mini-hypervisor underneath Xen [Zhang '11]
  - isolates co-hosted VMs belonging to different users
  - leverages remote attestation (TPM) and Turtles ideas

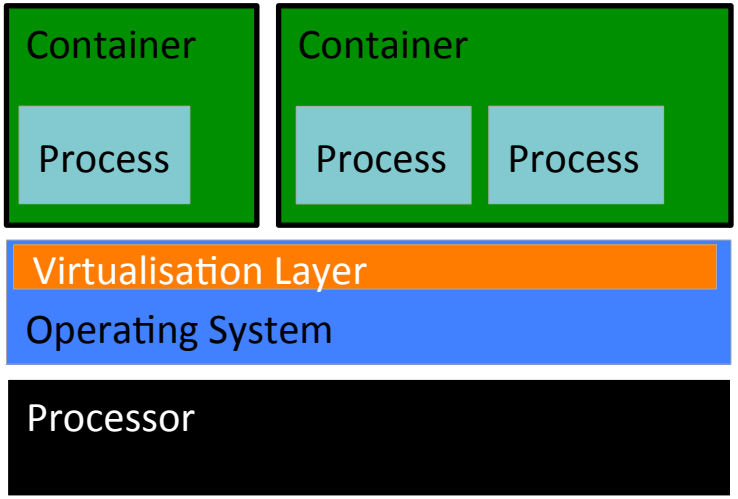
… and many more!

Motivation: overhead of isolation

- VMs are great for isolation, but have significant overheads
  - ***resource*** overheads: disk (GBs) and memory (512 MB+) per VM
  - ***runtime*** overheads: CPU virtualization, I/O virtualization, etc.
  - ***administrative*** overheads: one new OS to manage per VM
  - ***ingress/egress*** overheads: moving large VHDs to/from the cloud
- …but they offer great benefits!
  - Securely isolate guest from host
  - Support live migration
  - Only (?) isolation mechanism strong enough to enable the cloud
- Can we retain their benefits with less overhead?
  - Most apps don't need to see virtualized hardware
  - Most apps don't require their own OS + drivers

OS Containers

- OS kernel modified to virtualise at syscall interface
  - Files
  - Networking
  - PIDs
  - IPC
  - User & group IDs
  - …
- Additional controls on resource allocation
  - Not just best effort
- e.g. Docker, Solaris Zones, …



Container Example: UNIX stat

stat structure, which contains the

/* ID of device containing file */

/* inode number */

/* protection */

/* number of hard links */

/* user ID of owner */

/* group ID of owner */

/* device ID (if special file) */

/* total size, in bytes */

/* blocksize for filesystem I/O */

/* number of 512B blocks allocated */

Linux Containers History

- Chroot
  - Change the root of file system
  - Originally to develop new software releases
- Jail
  - Execute process with restricted set of system calls
  - Ex: postscript viewer in web browser
- Namespaces/cgroups
  - Restrict process visibility and resource usage
  - Per-container network address translation

Containers pros/cons

- Much lower overhead
  - Only one copy of the OS kernel
  - Single level of address translation
  - Drivers not an issue – trusted in the host OS
- Tight(er) coupling between guest/host
  - Can't run different guest OS
  - Harder to encapsulate and migrate state
- … but are they secure?
  - Full OS kernel and drivers in TCB of all containers
  - Syscall interface more complex than VM interface

Threat models for isolation

- Traditional enterprise ("friendly multi-tenant") threat model: *employees run code of their choosing on your system*
- Cloud (multi-tenant) threat model: *anonymous hackers with unlimited access run any code of their choosing on your systems, alongside your most valued customers*
  - Do you trust an OS kernel to isolate them?
  - Do you even trust a hypervisor to isolate them?
    - (More on this later…)

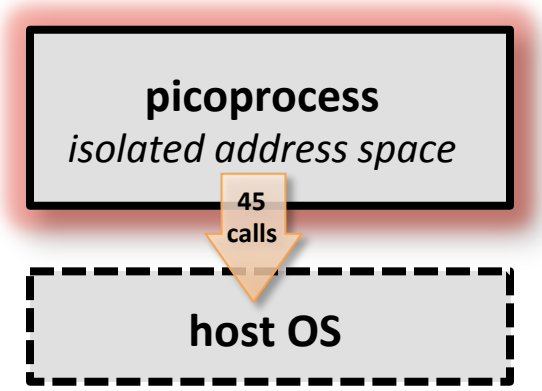
What's the Drawbridge approach?

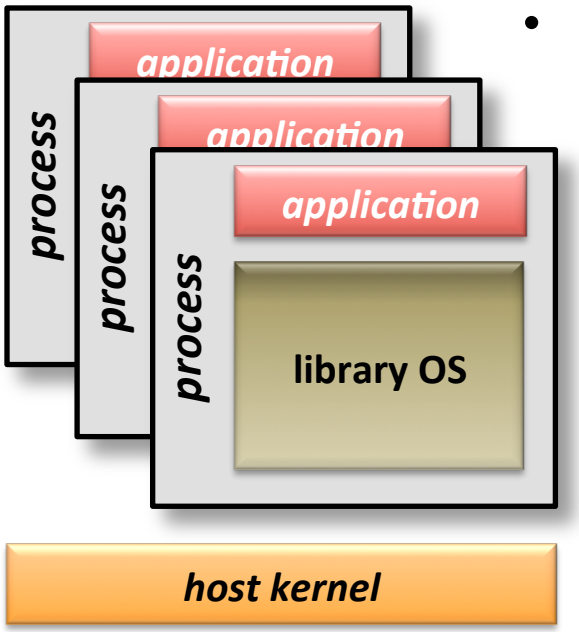
- Key design philosophy:
  - Start with a tight, secure isolation boundary
  - Add app compatibility ***inside*** isolation container
  - ***Not*** plugging holes in a leaky but compatible interface
- Key components:
  - The *picoprocess*, an isolation mechanism
  - The *library OS*, a compatibility mechanism

D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, G. C. Hunt, Rethinking the library OS from the top down, Proc. ASPLOS'11

Picoprocesses and library OSes

- ***Picoprocess*:** concept introduced by MSR's Xax project *(Douceur et al., 2008)*
  - Isolated address space with a *very small*, fixed interface with its host
  - Lightweight, **secure isolation container**

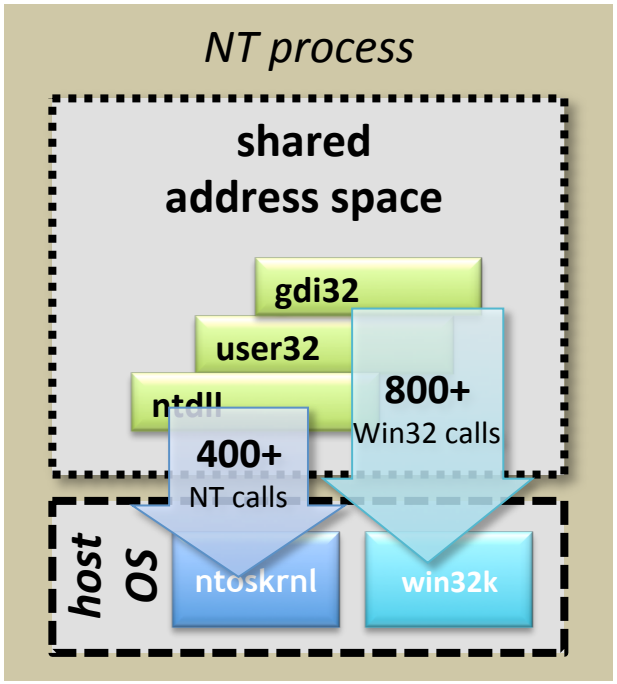


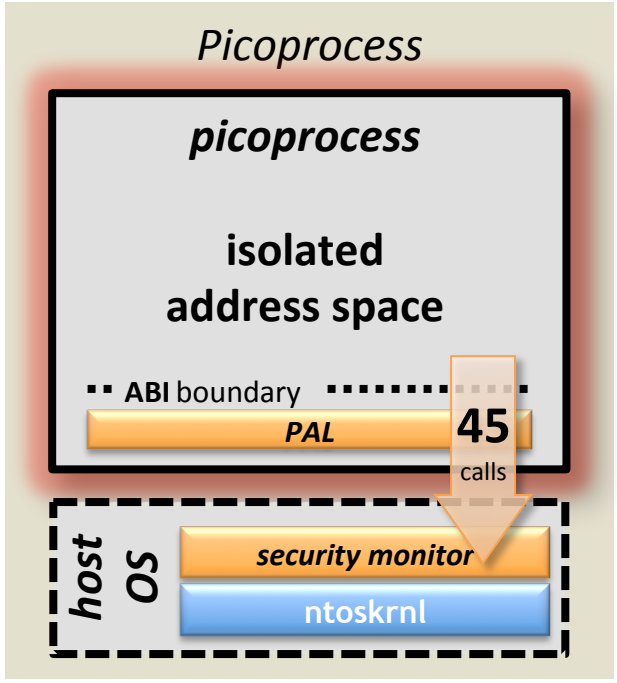


- ***Library OS***: concept championed in CS community in the '90s *(Engler et al., 1995)*
  - Minimal, shared kernel runs in supervisor mode
    - Multiplexes and abstracts hardware resources
    - Enforces cross-application protection
  - Per-app library OS **runs in user mode**
    - Constitutes OS "personality"
    - Provides application services and APIs to application
    - Runs in application's address space (user mode)
    - Each app can choose its own library OS

Drawbridge picoprocess on NT

- NT process with modified service handler
  - **All 1200+ system calls blocked** from user-mode (NTOS and win32k)
  - **45 new system calls added** to process (*Drawbridge system calls*)





The Drawbridge ABI

- ***Drawbridge ABI***: interface between a Drawbridge picoprocess and its host
  - 45 downcalls, 3 upcalls – *everything else is off-limits*
  - Designed from scratch, but heavily inspired by NT
  - APIs have fixed, closed semantics (no IOCTLs)
- Analogous to VM host/guest interface, but with higher-level abstractions
  - **threads** (not virtual CPUs)
  - **virtual memory** (not physical memory)
  - **I/O streams** (not virtual device hardware)
- Design benefits:
  - ***security*** **-** interface is small enough to undergo manual review / inspection
  - ***portability*** - Windows apps run unmodified on any system that implements 45 functions
  - ***flexibility*** – interface allows app's state to live (almost) entirely in process

Drawbridge ABI
(excerpt)

Threading
```
DkThreadCreate
DkSemaphoreCreate
DkSemaphorePeek
DkSemaphoreRelease
DkObjectsWaitAny
...
```

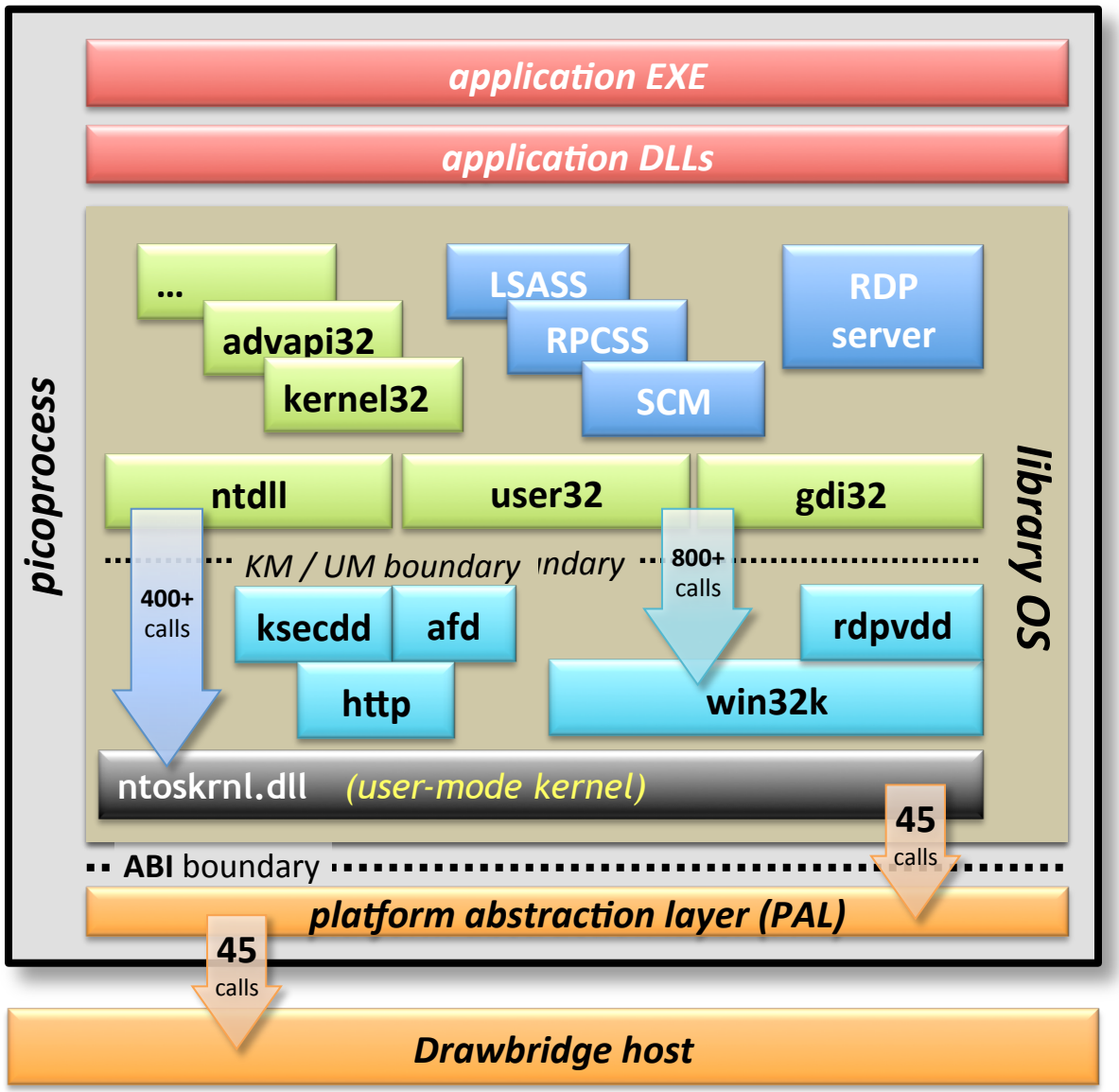
Memory management
```
DkVirtualMemoryAllocate
DkVirtualMemoryFree
DkVirtualMemoryProtect
```

I/O streams
```
DkStreamOpen
DkStreamRead
DkStreamWrite
DkStreamMap
DkStreamFlush
...
```

Upcalls
```
LibOsInitialize
LibOsThreadStart
LibOsExceptionDispatch
```

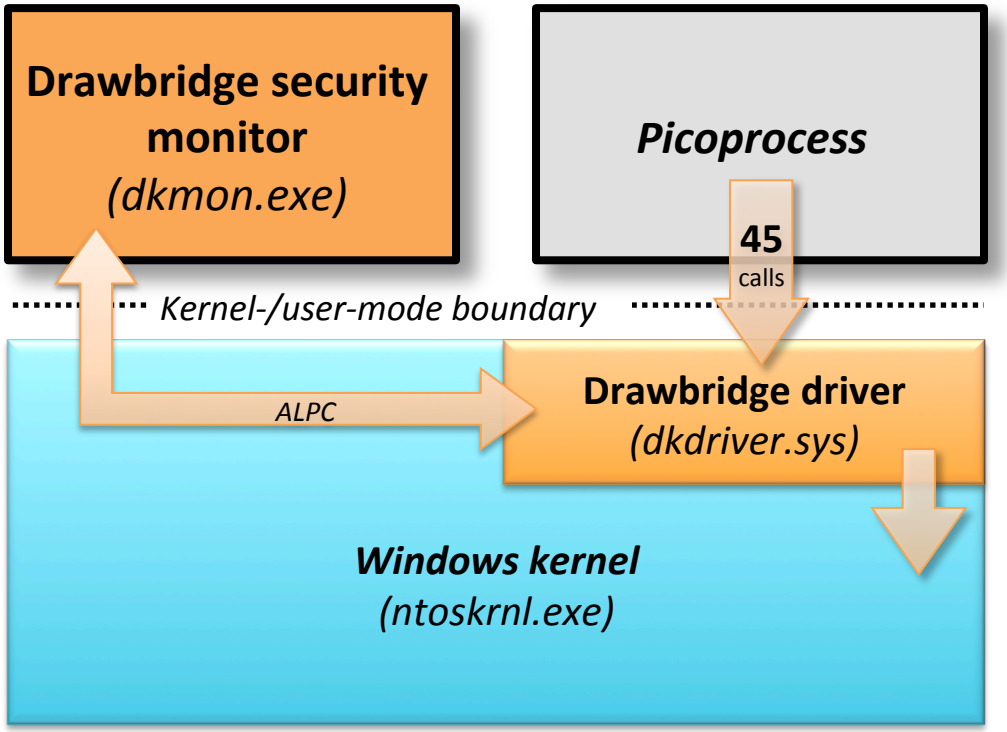
The Windows library OS

- Based on Windows OS
  - Same binaries *(where possible)*
  - Same architecture
- Windows ***enlightened*** **to run in a picoprocess** with the app
  - lifted into user mode
  - most changes in user-mode kernel
- Example library OS: *Win7 SP1*
  - **100MB on disk** (~150 DLLs)
  - **16MB of working set** + app
  - 5.5+ MLoC for 15,000+ Win32 APIs
- Each picoprocess runs its own library OS
  - app chooses its library OS
  - version need not match across picoprocesses or host



The Drawbridge-on-Windows host

- ***Drawbridge host*** implements 45-function ABI atop Windows
- Analogous to Hyper-V's hypervisor + virtualization stack
- Split between kernel-mode driver and user-mode worker
  - Driver implements ABI
  - Driver consults security monitor for policy decisions



The Drawbridge security monitor

- ***Security monitor*** – user-mode half of Drawbridge host
  - launches app in picoprocess
  - makes access policy decisions
  - "normal" NT process
- Policy decisions based on *manifests*
  - All external resources are blocked by default
  - Resources can be white-listed back in by admin
  - Access specified via virtual to physical namespace mappings

Drawbridge security monitor
(dkmon.exe)

Sample Policy

```
[Namespace.Writable]
pipe.server:///RDP=pipe.server:///RDP_Drawbridge          ; expose 'RDP' named pipe server out of
                                                          ; process as 'RDP_Drawbridge'
tcp.server://localhost:3000=tcp.server://localhost:3000   ; allow app to listen (only) on port 3000
tcp.client:=tcp.client:                                   ; allow use of any TCP client socket

[Namespace]
file:///users/jdoe/documents=file:///documents            ; allow R/O access to Documents folder
```

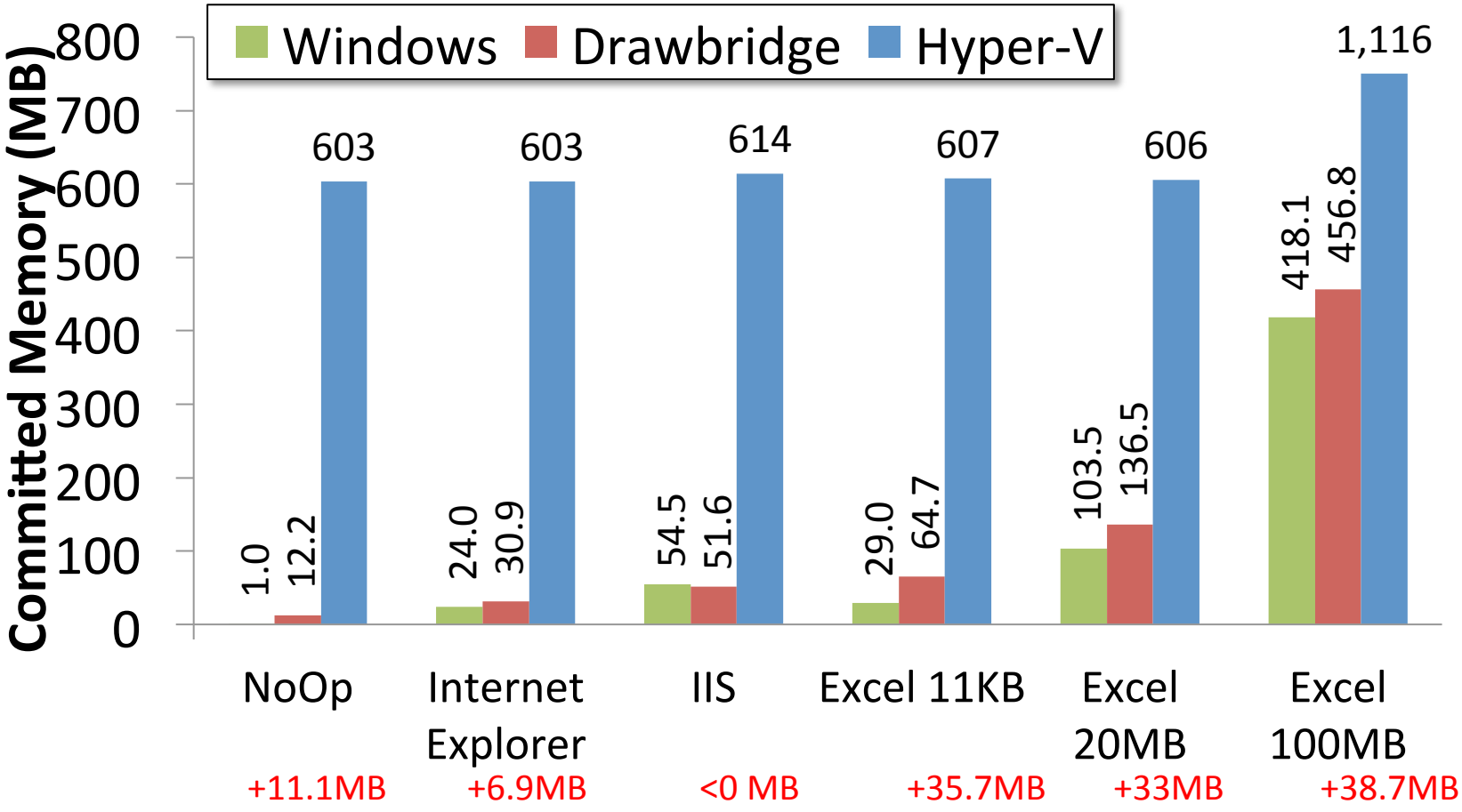
Drawbridge packages

- ***Drawbridge package*** – self-contained, self-describing unit of deployment
- A package contains:
  - Manifest
    - Identity (name, version, options)
    - Dependencies on other packages
    - Access control policy requirements
    - Relative paths to important contained files (e.g. app EXE)
  - Files
  - Registry data (.reg format)
  - Debug resources (e.g. symbols, etc.)
- Everything's a package: app, library, library OS, suspended app
- Security monitor resolves transitive closure of packages and dependencies
  - **File content from packages is unioned** into virtual FS
  - **Registry content from packages is unioned** into virtual registry
  - Packages are read-only, mapped copy-on-write

Sample Manifest

```
[Package]
ManifestVersion=1
PackageRevision=4

[Identity]
Name=IISWorker
MajorVersion=7
MinorVersion=5
BuildNumber=7601
Architecture=x64

[Dependency.Win7]
Name=Windows
MajorVersion=6
MinorVersion=1

[Dependency.CLR4]
Name=MicrosoftNET
MajorVersion=4
MinorVersion=0

[Windows.Application]
Exe=package:///windows/
system32/inetsrv/w3wp.exe

[Windows.Registry]
File:///w3wp.exe.dbreg
```

Committed Memory by Apps



Time to Start Application Package