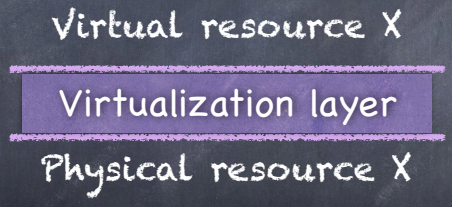
Virtualization

A technique, not a principle

- Applies fundamental principles
  - creates an Abstraction to enforce Modularity through Layering, often relying on an Interpreter
- Main goals
  - reduce complexity
  - enable automation
  - reduce performance mismatches

In its simplest form

- A layer that exports the same abstraction as the layer that it relies upon

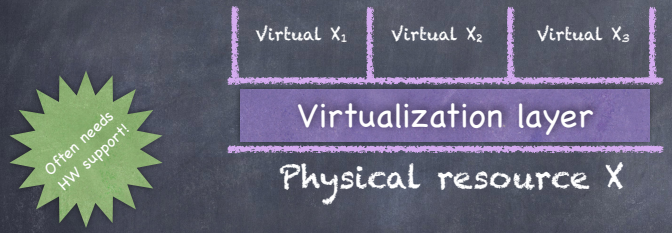


- Adds from consumer of virtual resource names of underlying physical resource
  - enforces modularity through isolation

In general

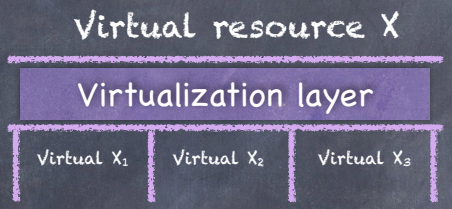
- Three ways to virtualize
  - Multiplexing
    - many virtual objects from one physical object
  - Aggregation
    - one enhanced virtual object from many physical objects
  - Emulation
    - a virtual object form a different kind of physical object

Multiplexing



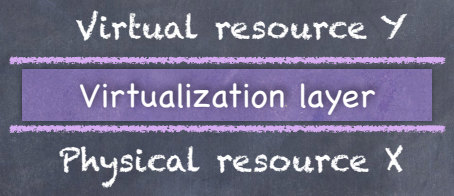
- Thread: One CPU ⟶ N CPUs
- Virtual Memory: One memory ⟶ N memories
- Virtual Circuit: One channel ⟶ N channels
- Virtual Machine: One machine ⟶ N machines

Aggregation



- Raid: N disks ⟶ One disk
- SMR: N replicas ⟶ One replica
- Channel bonding: N channels ⟶ One channel

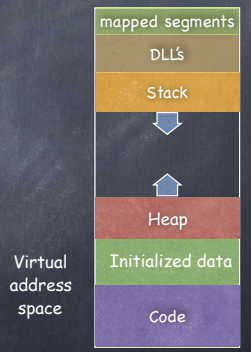
Emulation



- RAM Disk: RAM as a very fast disk
- Apple's Rosetta: x86 as a Power PC
- Virtual Tape: disk as tape

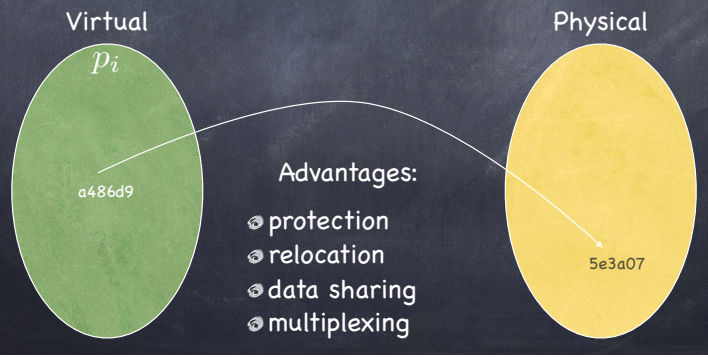
Virtualizing memory

- Virtual address space: set of memory addresses that process can "touch"
  - CPU works with virtual addresses
- Physical address space: set of memory addresses supported by hardware



Address translation

- Implement a function mapping
  $\langle pid, virtual\ address \rangle$ into $physical\ address$

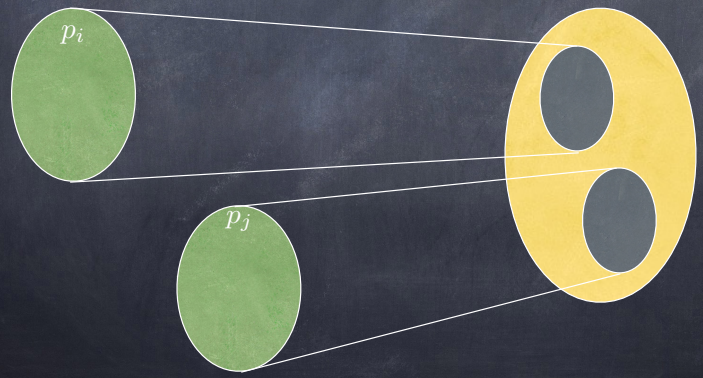


Advantages:

- protection
- relocation
- data sharing
- multiplexing

Protection

- At all times, the functions used by different processes map to disjoint ranges



Relocation

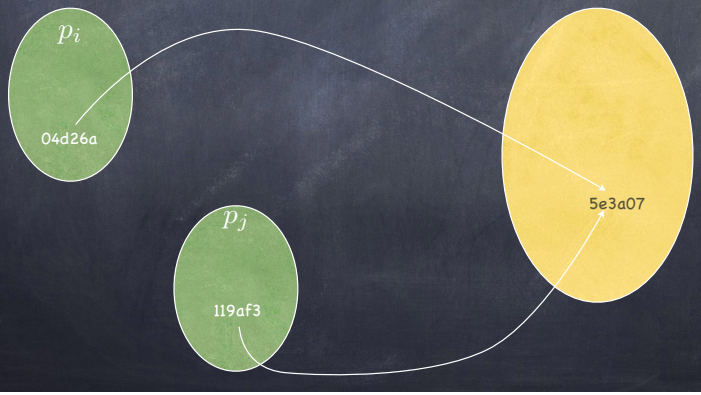
- The range of the function used by a process can change over time

Relocation

- The range of the function used by a process can change over time

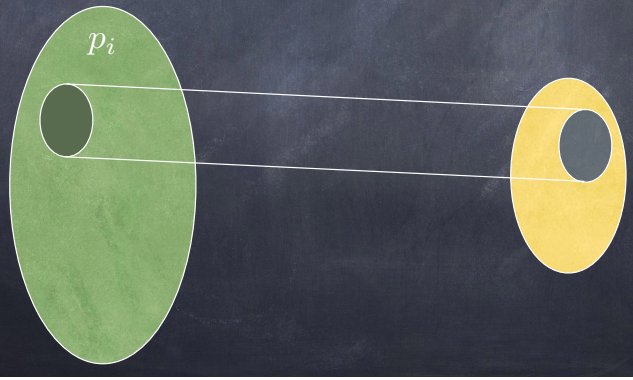
Data Sharing

- Map different virtual addresses of different processes to the same physical address



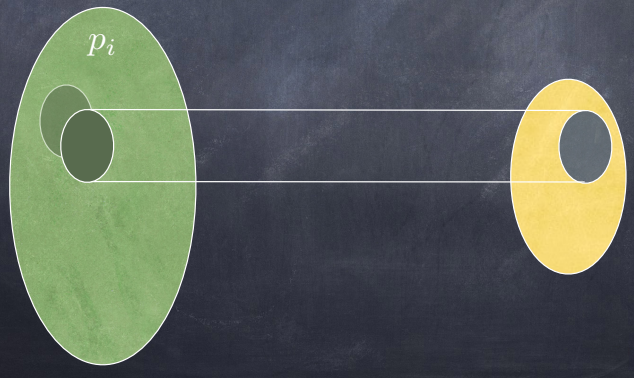
Multiplexing

- Create illusion of almost infinite memory by changing domain (set of virtual addresses) that maps to a given range of physical addresses



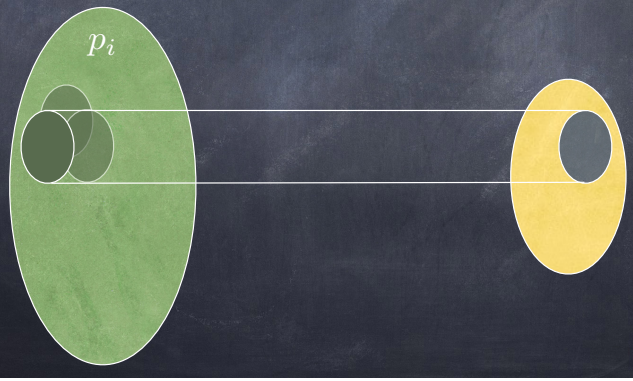
Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



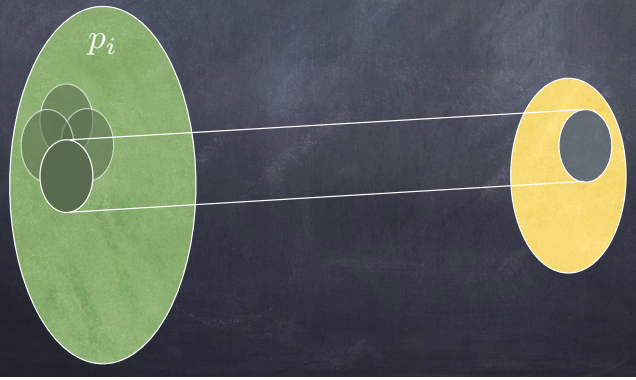
Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



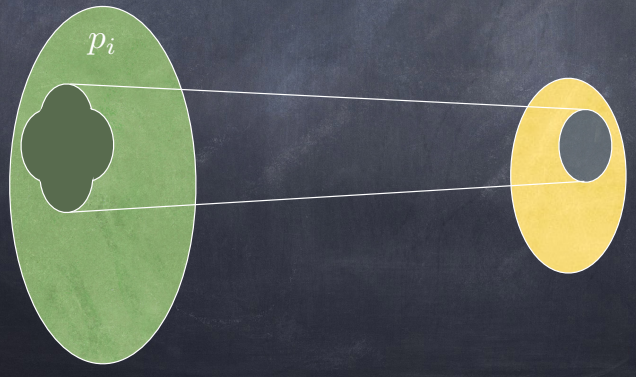
Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time

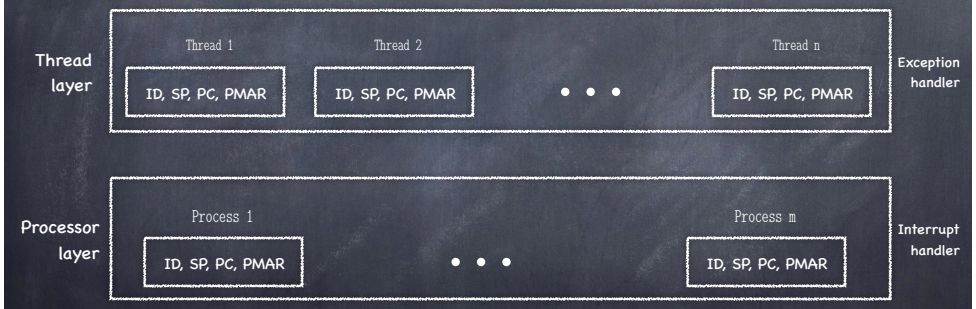


Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



Virtualizing processors via threads



- Threads share control of processor through `YIELD()`
  - Three steps: Save · Schedule · Dispatch

Implementing `YIELD()`

- Threads are sharing address space
- All running
- More threads than processors

```
shared structure processor_table[PROC_NO]
    integer thread_id // running thread's id
```

```
shared structure thread_table[THREAD_NO]
    integer topstack   // stack pointer
    integer state      // RUNNABLE or RUNNING
```

```
shared lock instance thread_table_lock
```

```
procedure YIELD()
    ACQUIRE(thread_table_lock)
    ENTER_PROCESSOR_LAYER(GET_THREAD_ID())
    RELEASE(thread_table_lock)
    return
```

```
procedure ENTER_PROCESSOR_LAYER(this_thread)
    thread_table[this_thread].state ← RUNNABLE
    thread_table[this_thread].topstack ← SP
    SCHEDULER()
    return
```

```
procedure SCHEDULER()
    j ← GET_THREAD_ID()
    do
        j ← (j + 1) mod THREAD_NO
    while thread_table[j].state ≠ RUNNABLE
    thread_table[j].state ← RUNNING
    processor_table[CPUID].thread_id ← j
    EXIT_PROCESSOR_LAYER(j)
    return
```

```
procedure EXIT_PROCESSOR_LAYER(new)
    SP ← thread_table[new].topstack
    return
```

where do we return?

```
procedure GET_THREAD_ID()
    return processor_table[CPUID].thread_id
```

Virtual machines

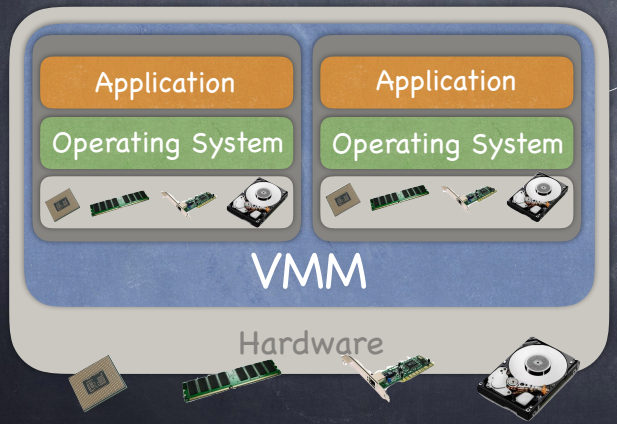
Virtualizing Hardware

How it all began

- Computer are expensive
- OS has very limited capability
  - no time sharing
- Virtual machines give illusion of private computer
  - Allow OS innovation
  - Very popular!
    - over time, hardware support for virtualization!

IBM 360, mid `60

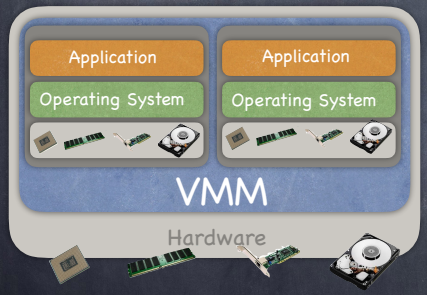
Virtual Machine: A Definition



"An efficient, isolated duplicate of the real machine"

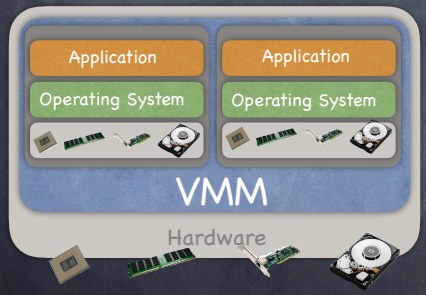
Popek & Goldberg ` 74

What it means for the VMM



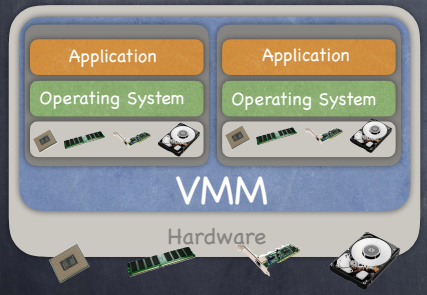
- Environment for programs must be essentially identical to that of original machine

What it means for the VMM



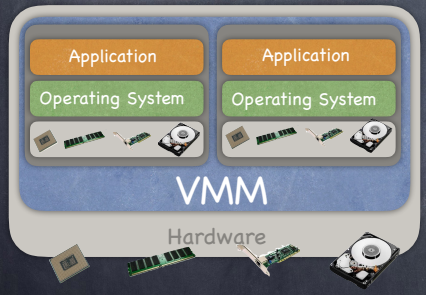
- Environment for programs must be essentially identical to that of original machine

What it means for the VMM



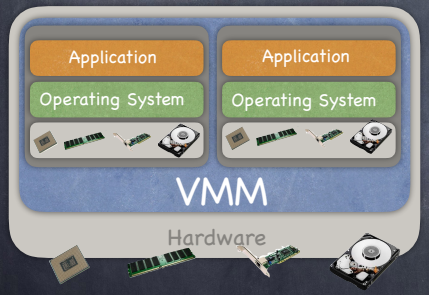
- Environment for programs must be essentially identical to that of original machine
  - effect of running any program under VMM should be identical to running it on the original machine
    - except (possibly) because of differences due to timing dependencies and availability of resources

What it means for the VMM



- Programs running in VM must show at worst only minor decreases in speed
  - most virtual processor's instructions must be executed directly on the real processor, with no software intervention from VMM

What it means for the VMM



- The VMM must be in complete control of system resources
  - programs cannot access resources not allocated to them
  - VMM can regain control of already allocated resources

… and then, it al ended

- The rise of the modern OS
  - anything you can do I can do better!
  - hardware support stopped
  - by the mid 90s, virtual machines were largely dismissed
    - but remember the work on Hypervisor-based FT
      Bressoud and Schneider, SOSP '95

Rising from the ashes

- The Flash multiprocessor (Stanford '94)
  - cache-coherent shared memory and high performance message passing
- Its OS, Hive, proves hard to build
  - OS has become too big and complex
    - hard to adapt for ccNUMA architectures
- VMM can partition resources across VMs in a way that their OSs can manage
  - and even support specialized OSs

And then, of course…

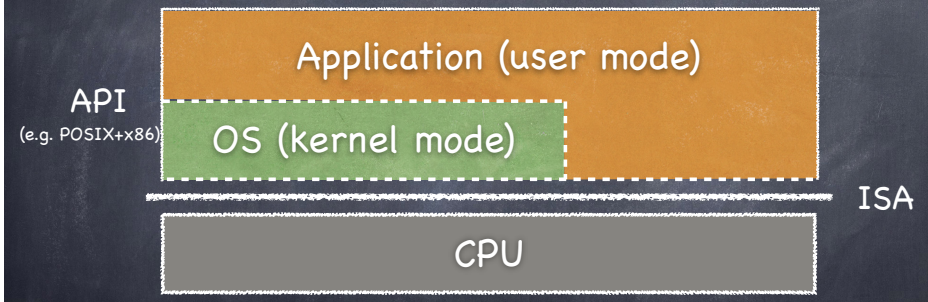
- Cloud computing
  - VMMs simplify sharing resources while maintaining isolation

but how does one build a VMM?

The basics

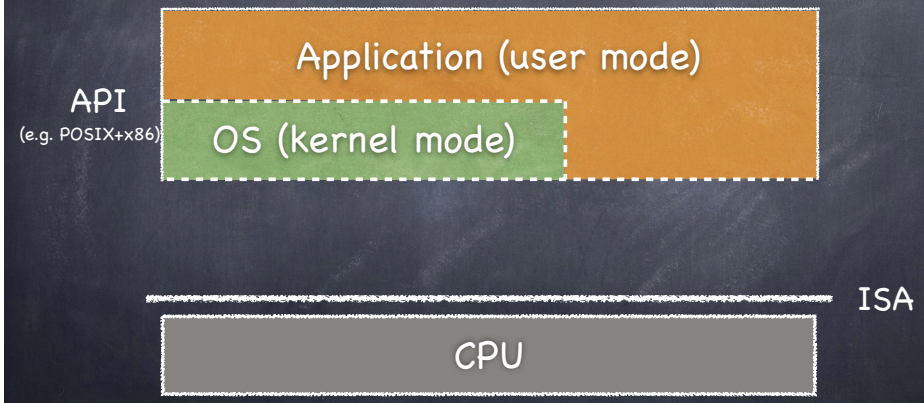
- Multiplex
  - CPU
    - VM has virtual CPUs/Cores
  - Memory
    - VM has Virtual physical memory (real memory)
- Emulate
  - Sensitive instructions, even in systems with architectural support for virtualization
  - I/O devices
    - virtual disk, nic, screen, keyboard…

An exercise in layering



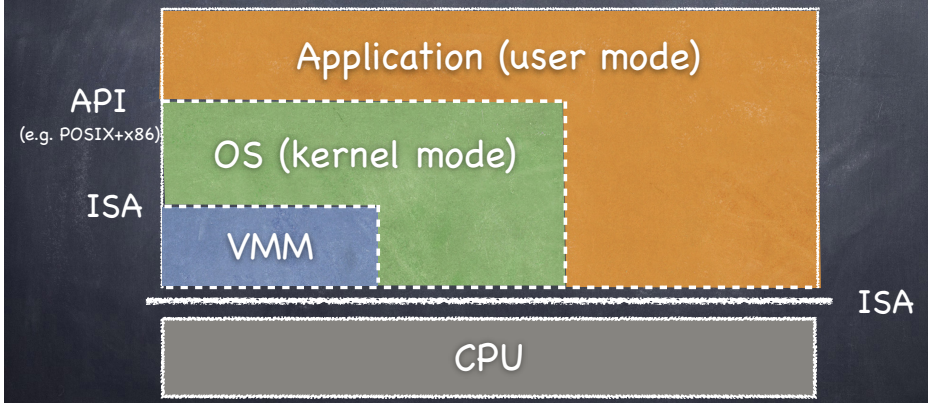
Applications execute unprivileged instruction directly on the CPU, without OS intervention

An exercise in layering



In principle, all you need!

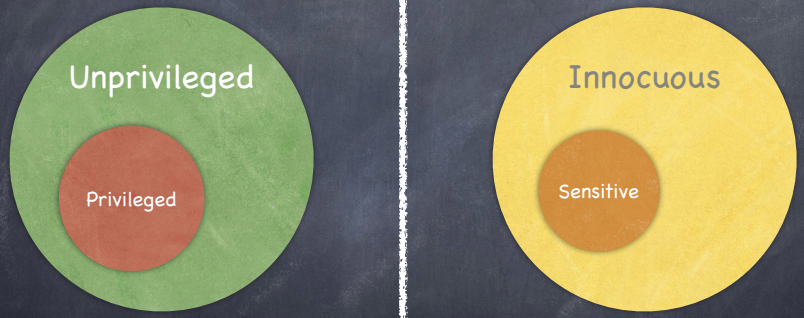
An exercise in layering



Virtualizing the ISA

- Machine modeled as a tuple
  - E (executable storage)
  - M (mode of operation – user or kernel)
  - P (program counter)
  - R (relocation registers, bound contiguous physical memory used by virtual memory)
    - straightforward extension to paging etc.
- A trap is generated for
  - attempts to access addresses out of bound
  - privileged instructions executed in user mode

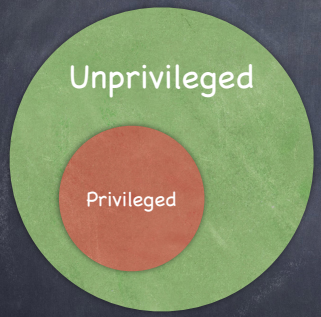
Types of instructions



LPSW: Load Process Status Word

SPT: Set CPU Timer

Types of instructions

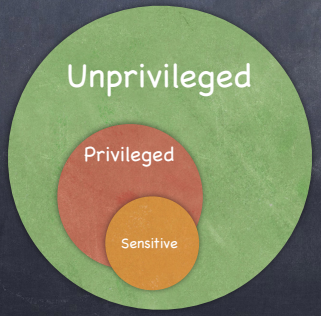


LPSW: Load Process Status Word

SPT: Set CPU Timer

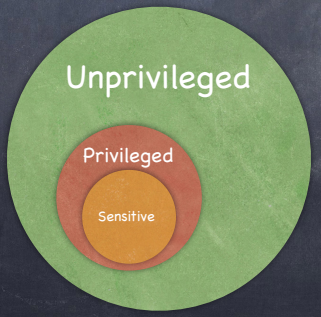
- Sensitive instructions
  - Control sensitive
    - Change system's resource configuration
    - LPSW, SPT
  - Behavior sensitive
    - Behavior or results depend on system's configuration
    - LRA (IBM 370): Load Real Address
      - Result depends on mapping of real memory
    - POPF (Intel): Pop Stack into EFlags Register
      - One of the flags disables interrupt; no-op in user mode

A sufficient condition for building a VMM



A sufficient condition for building a VMM

P&G Theorem 1:
Sensitive instructions must be privileged



Every sensitive instruction generates a trap, so VMM can emulate it

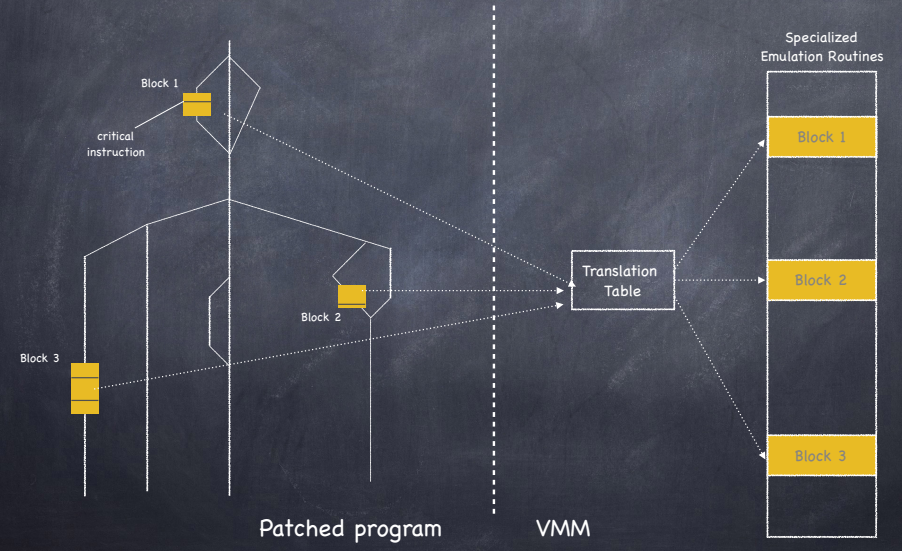
What if the theorem does not hold?

- POPF
  - one of 17 sensitive but unprivileged (critical) instruction on x86

What if the theorem does not hold?

- POPF
  - one of 17 sensitive but unprivileged (critical) instruction on x86
- Binary translation: Scan and Patch
  - Trap to VMM at start of each basic block
  - Sensitive instruction and location saved in VMM side table
  - Trap to VMM at end of basic block, repeat
    - if patched all downstream basic blocks, switch back to jump

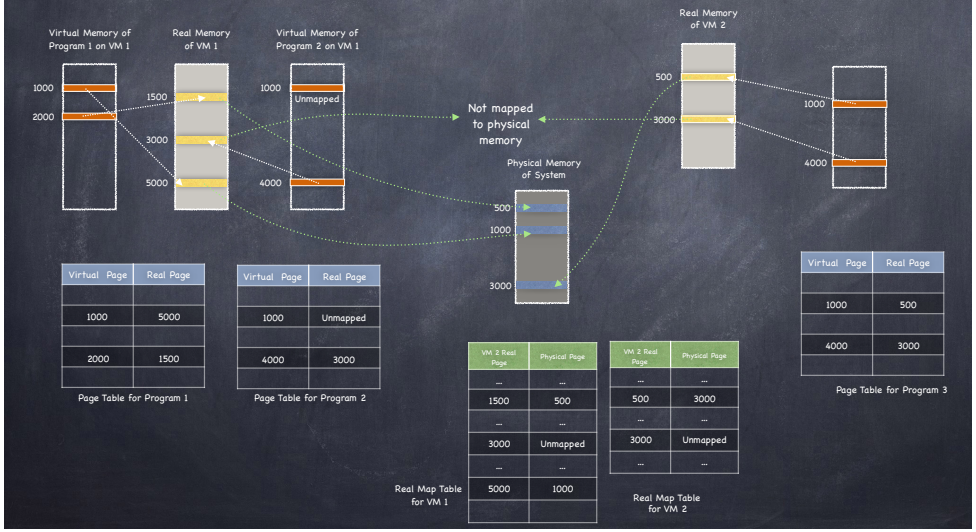
Caching Emulation Code



Virtualizing memory

- OS assumes full control over memory…
  - Managing it: it owns it all
  - Mapping it: any page to any frame
- … but VMM is the real boss
  - Assigns frames to VMs
  - Controls mapping for isolation
- Two levels of indirection!

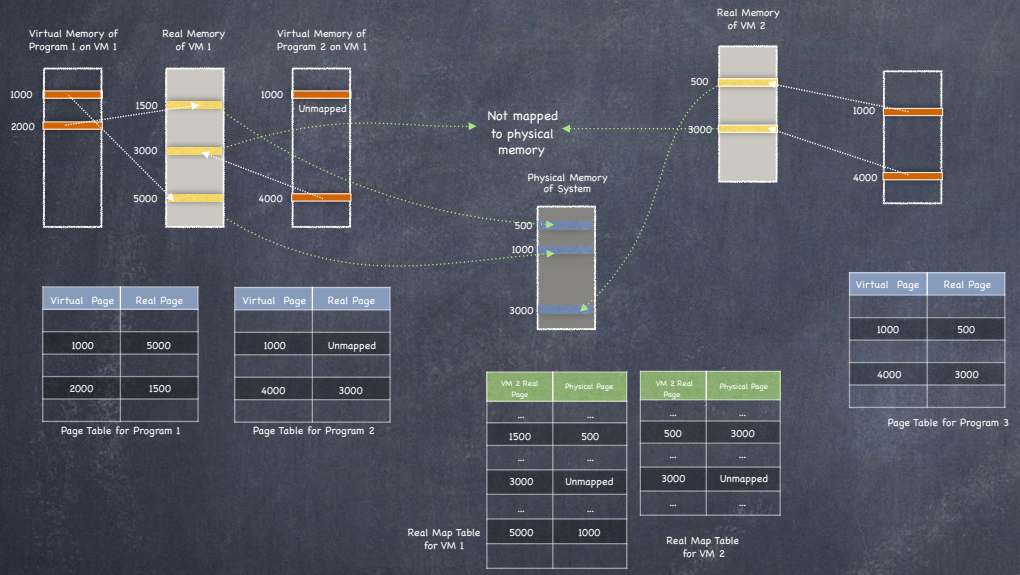
Virtualizing memory



The TLB challenge

- TLB maps directly virtual to physical addresses
  - if we can map program memory on guest to host physical memory, we can reduce performance hit
- Solution depends on what is architected (can be manipulated through ISA)
  - Page table or
  - TLB itself

Architected PT: Shadow Page Tables



Program 1 on VM 1 is currently active

Page Table Pointer

Virtual Page	Physical Page
...	...
1000	1000
...	...
2000	500
...	...

Shadow Page Table for Program 1 on VM 1

Virtual Page	Physical Page
...	...
1000	Unmapped
...	...
4000	Unmapped
...	...

Shadow Page Table for Program 2 on VM 1

Virtual Page	Physical Page
...	...
1000	3000
...	...
4000	Unmapped
...	...

Shadow Page Table for Program 3 on VM 2

- One shadow page table per VM
  - if mapping in shadow page, then mapping in virtual page
- Used by hardware to keep TLB up-to-date

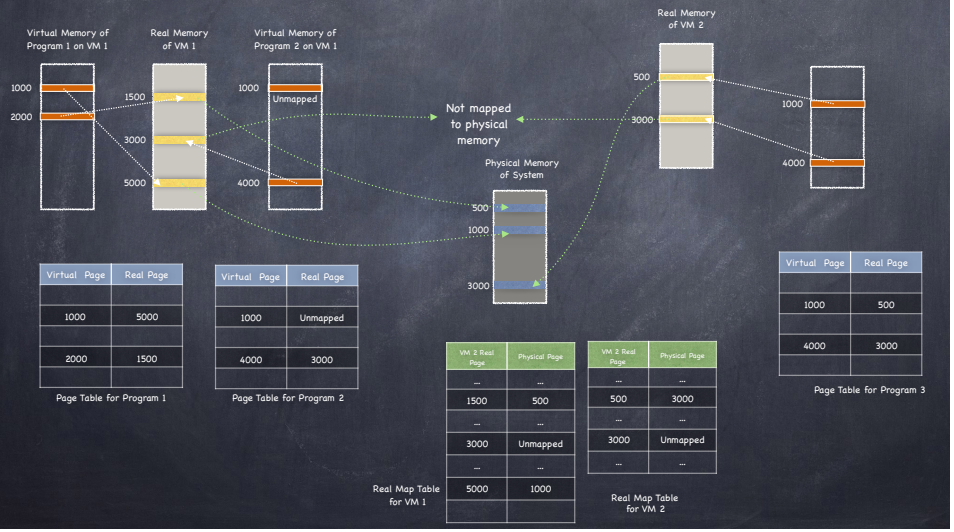
Serving page faults

- If page mapped in virtual table
  - handled entirely by VMM
    - VMM maps page to a frame
    - updates appropriately real map table and shadow table
- If page not mapped in virtual table
  - VMM passes control to guest's handler, notifying a page fault
    - Guest OS issues I/O requests
    - Guest OS issues instructions to modify virtual page table
    - VMM intercepts them and makes the changes - tricky

Architected TLB

- Rewrite TLB when switching to a VM
  - copy VM's virtual TLB to physical TLB
    - inefficient
- Leverage Address Space Identifiers (ASIDs)
  - virtualize ASID register

Virtualizing memory



Architected TLB

- Rewrite TLB when switching to a VM
  - copy VM's virtual TLB to physical TLB
    - inefficient
- Leverage Address Space Identifiers (ASIDs)
  - virtualize ASID register

ASID Map Table

Virtual ASID	Real ASID
...	...
VM1:3	9
...	...
VM1:7	...
...	...
VM2:3	4

ASID mapping:
Prog 1 – ASID 3
Prog 2 – ASID 7

ASID	Virtual Page	Real Page
...	...	
3	1000	1000
...	...	...
3	2000	1500
...	...	...
7	4000	3000
...	...	...

Virtual TLB of VM1

ASID mapping:
Prog 1 – ASID 3

ASID	Virtual Page	Real Page
...	...	
3	1000	3000
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...

Virtual TLB of VM2

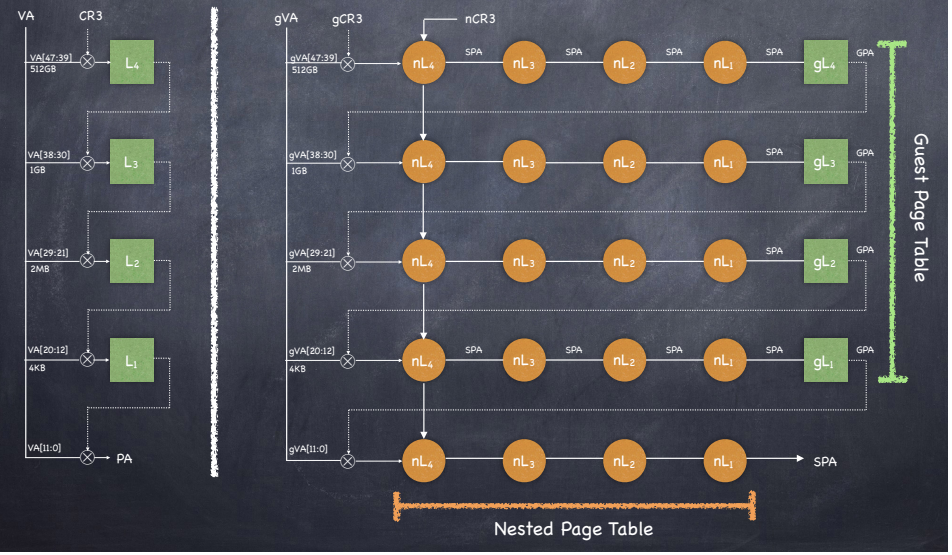
Real TLB

ASID	Virtual Page	Physical Page
...	...	
9	1000	1000
4	1000	3000
9	2000	500
...	...	...
...	...	...

Architectural support for virtualization

- Nested page tables (or extended page tables)
- Hardware walks two different sets of page table structures simultaneously
  - two PT registers
    - one accessible in privileged more
    - the other not

2-D page walk



Virtualizing Events and I/O

- VMM receives interrupts, exceptions
  - must redirect to appropriate VM
  - craft handler invocation, emulate registers
- OS can no longer interact directly with I/O devices
  - intercept communication between OS driver and device — complex and expensive
  - add special driver to OS
    - can reduce number of traps when passing parameters

Type I and II VMMs

- Type I (Bare Metal)
  - allocate and schedule physical resources among the virtual machines
    - Xen, VMWare vSphere, MSoft Hyper-V
- Type II (Hosted)
  - rely on a separate host operating system for resource scheduling
    - KVM (part of Linux)
    - VMware Workstation and Fusion

Disco

Bugnion, Devine, Rosenblum

- Extend OS to run efficiently on cc-NUMA with minimal OS changes, aiming for
  - scalability
    - add a VM!
    - share memory regions across VM boundaries
  - flexibility
    - only (small) VMM needs to scale
    - support for specialized, simple OS (scientific apps)
  - fault containment
    - structure VMM into cells (borrowing from Hive)
  - compatibility with legacy applications
    - hide NUMA effects

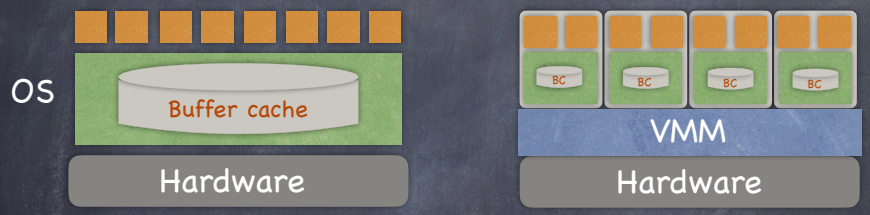
Virtualizing memory

- Architected TLB
  - leverages ASIDs when switching to a different VM on the same virtual CPU
  - but TLB is flushed when scheduling a different virtual CPU

Memory resource management

- Memory is precious
  - in NUMA, make sure memory is local to CPU
  - avoid unnecessary replication of information
    - memory can be overcommitted!
- Exploit the additional level of indirection offered by VMM
  - mapping between guest physical (real) and host physical (physical) addresses can change without impacting VM

Memory resource management



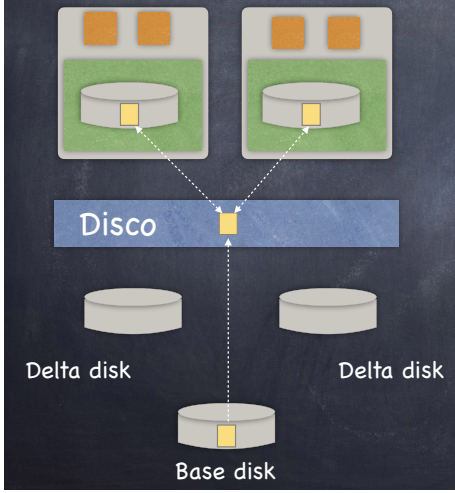
- Buffer caches occupy a significant fraction of memory
  - can they be transparently shared without involving OS?

The answer is layering

- Mapping between real and physical addresses can change
- And the type of mapping can change
  - W, R/W, R only
- Leveraged in
  - DMA to disc traffic
  - network traffic

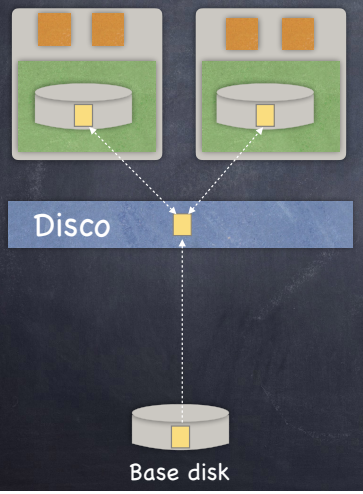
  to identify pages that are identical

Interpose on DMA - shared "base" disk



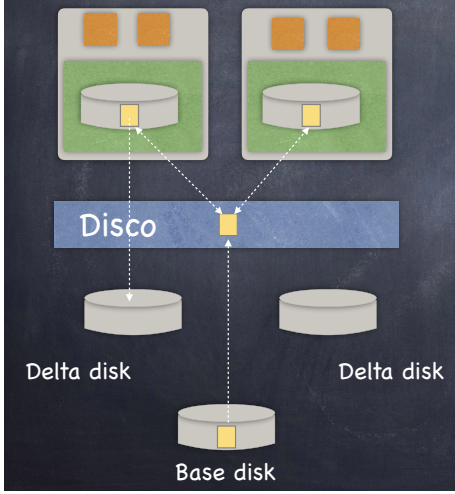
- Each VM has own logical disk
  - Base disk (shared) + Delta disk (private)
- Ex: nearly identical file systems
  - most reads from Base disk
  - on SCSI read from Base disk
    - associate page with disk offset
    - make page read-only
    - avoids future SCSI reads
  - what if one VM modifies the page?

Interpose on DMA - shared "base" disk



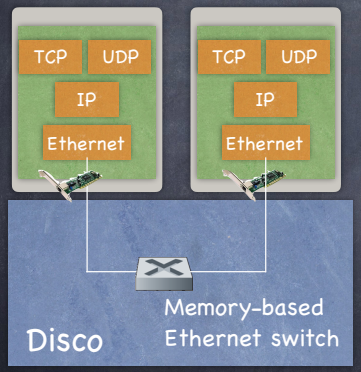
- Example: nearly identical file systems
  - most reads from Base disk
  - on SCSI read from Base disk
    - associate page with disk offset
    - make page read-only
    - avoids future SCSI reads
  - what if one VM modifies the page?

Interpose on DMA - shared "base" disk



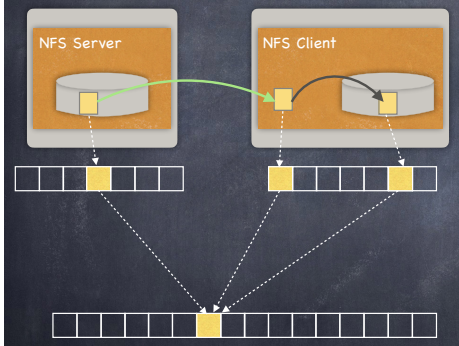
- Copy-on-write
  - Memory (triggered by page fault, since page is read only)
    - add new page mapping
  - Disc (triggered by SCSI write)
    - write new version to Delta disk
    - in Disco, implemented with logs in memory

VM to VM networking



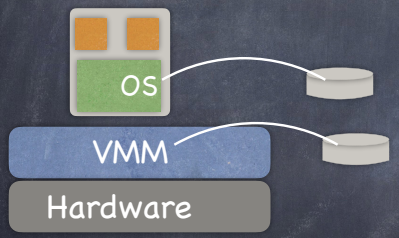
- Disco relies on existing network stack
- Emulates abstraction of Ethernet NIC for each VM
- The twist:
  - Disco can remap the fraction of a packet that straddles a full page

Interpose on network traffic



- Disco's in memory Ethernet switch
  - copies small fragments
  - remaps page-size fragments
- OS modification
  - IRIX NFS code — copy using call to VMM

The ~~answer~~ problem is layering



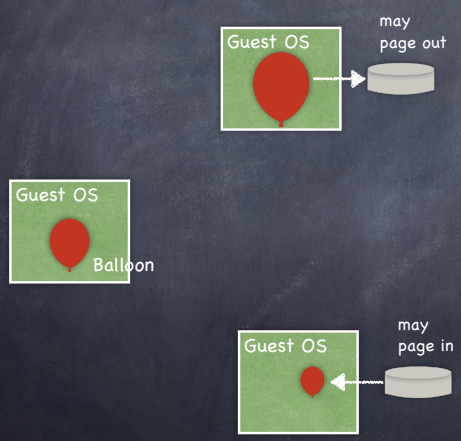
- Double swapping!
  - VMM, under memory pressure, swaps page X
  - OS, under memory pressure, decides to swap page X
    - VMM must (i) swap out another page and (ii) swap X back in
    - Only then can OS swap X out
- Disco does not deal with this
  - nor does the mainframe literature

Key issue

- VMM does not have the information needed to determine the best page to evict
- Likely to create unexpected interaction with guest OSes page replacement mechanisms in guest OSes
- The hip approach
  - coax guest into doing the page replacement
  - avoid meta-level policy decisions

Ballooning

Waldspurger '02

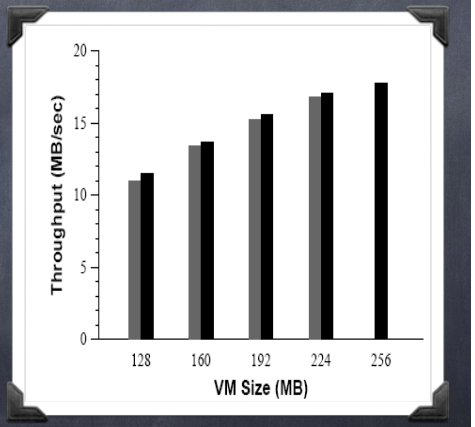


- Balloon
  - a pseudo device driver in guest OS
  - when inflated, allocates pinned physical pages within VM
    - increases memory pressure for guest OS
    - if guest OS deallocates a real page, VMM can then reclaim the physical page

Balloon performance

- Black bar: VM configured from 128 to 256 MB
- Gray bar: 256MB, ballooned down to specified size

Once memory is reclaimed, VM closely tracks performance of same VM with less memory



Throughput of VM running dbench (40 clients)

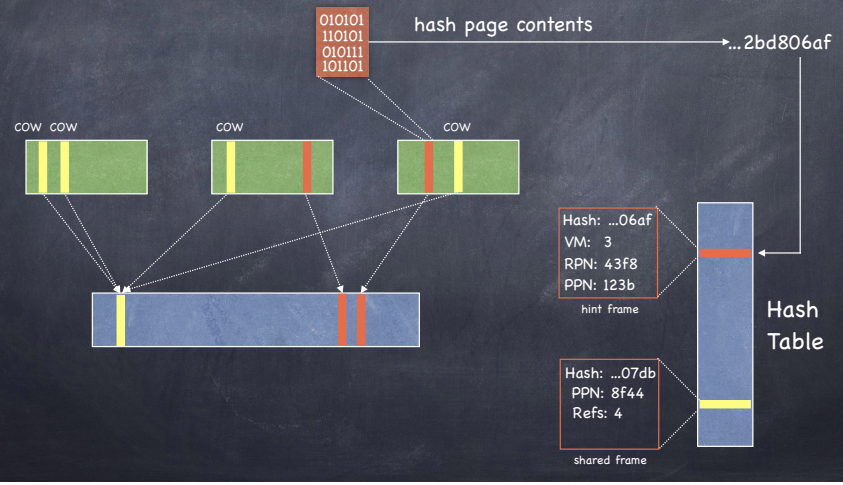
Sometimes, it does not work...

- During guest OS boot time
- If the driver is not installed or disabled
- May not reclaim memory fast enough

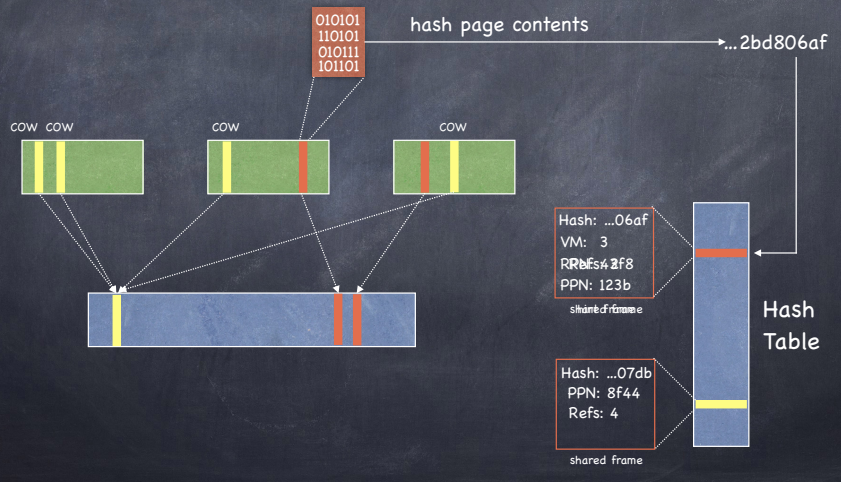
Page sharing

- Disco had transparent page sharing, but...
  - it required guest OS changes
  - it required nonstandard interfaces
- ESX Server introduces content-based page sharing Waldspurger '02
  - pages with identical content can always be shared, independent of how they were generated
    - no need to modify (or even understand) guest OS
    - more opportunities for sharing
    - need scanning to identify sharing opportunities

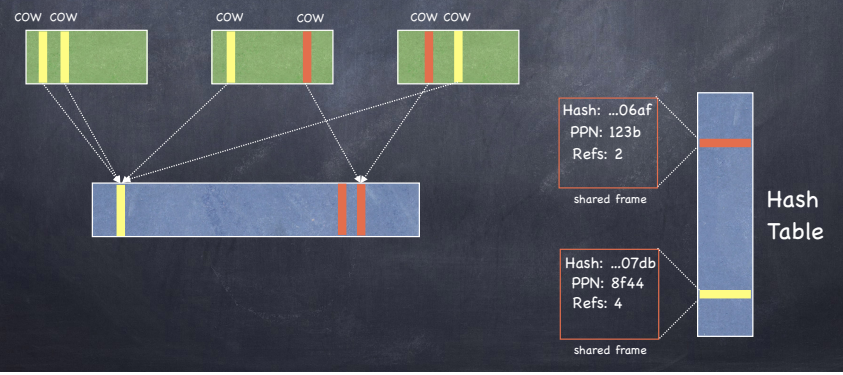
Content-based page sharing



Content-based page sharing



Content-based page sharing



Share-based memory allocation

- Resource rights expressed through shares
  - one is entitled to use resources proportionally to its share allocation
  - Graceful degradation in overloaded situations
  - Benefits distributed proportionally if more resources become available
  - When a VM needs more space, a victim is needed
    - min-funding revocation: choose the one that owns fewest shares per allocated page
    - how do we choose a victim in OS with dynamic allocation?

Approximating a working-set policy

- Tax idle memory
  - charge more for idle page than active one
  - tax rate specifies max fraction of idle pages that may be reclaimed
- Adjusted share-per-page ratio with tax rate τ

$$\rho = \frac{S}{P \cdot (f + k \cdot (1 - f))} \qquad k = 1/(1 - \tau)$$

- Pages in each VM are statistically sampled to estimate active memory usage