CSE 548: Computer Systems Architecture

Virtual Memory

Luis Ceze, Spring 2017

based on slides from friends at UPenn, UIUC, UW, MIT.

Virtualizing Processors

- How do multiple apps (and OS) share the processors?
  - **Goal: applications think there are an infinite # of processors**

Virtualizing Processors

- How do multiple apps (and OS) share the processors?
  - **Goal: applications think there are an infinite # of processors**

- Solution: *time-share* the resource
  - Trigger a **context switch** at a regular interval (~1ms)
    - **Pre-emptive**: app doesn't yield CPU, OS forcibly takes it
      + Stops greedy apps from starving others
  - **Architected state**: PC, registers
    - Save and restore them on context switches
    - Memory state?
  - **Non-architected state**: caches, branch predictor tables, etc.
    - Ignore or flush
- OS responsible to handle context switching
  - Hardware support is just a timer interrupt

Virtualizing Main Memory

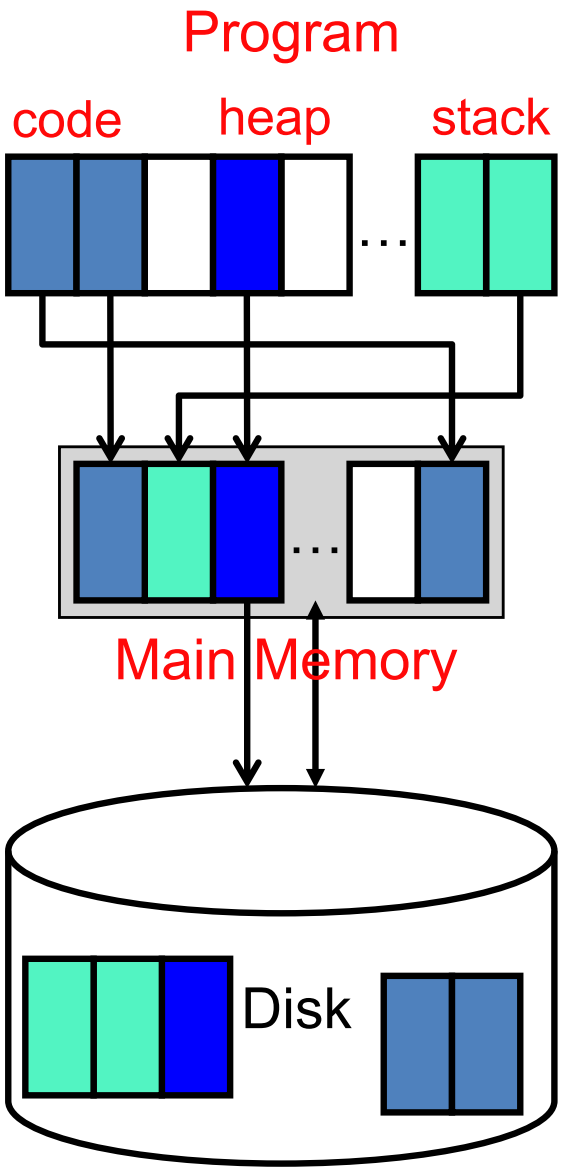
- How do multiple apps (and the OS) share main memory?
  - **Goal: each application thinks it has infinite memory**

Virtualizing Main Memory

- How do multiple apps (and the OS) share main memory?
  - **Goal: each application thinks it has infinite memory**

- One app may want more memory than is in the system
  - App's insn/data footprint may be larger than main memory
  - **Requires main memory to act like a cache**
    - With disk as next level in memory hierarchy (slow)
    - Write-back, write-allocate, large blocks or "pages"
  - No notion of "program not fitting" in registers or caches (why?)
- Solution:
  - Part #1: treat memory as a "cache"
    - Store the overflowed blocks in "swap" space on disk
  - Part #2: add a level of indirection (address translation)

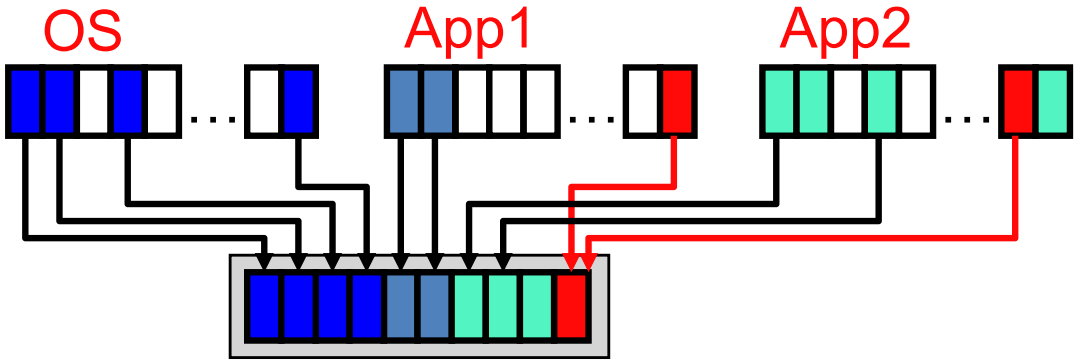
Virtual Memory (VM)



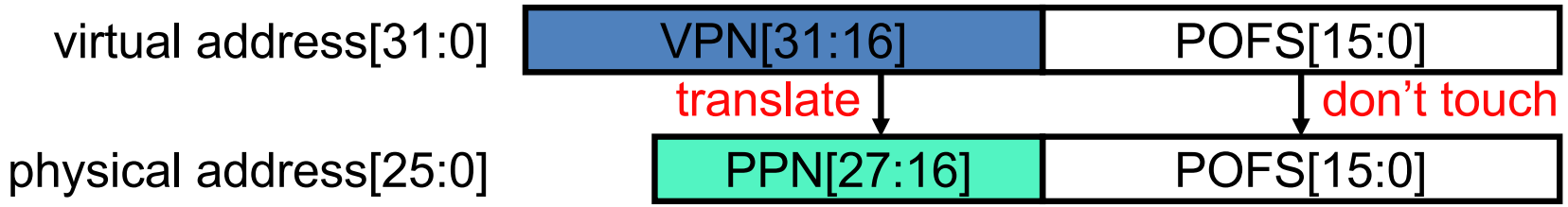
- Programs use **virtual addresses (VA)**
  - $0…2^N–1$
  - VA size also referred to as machine size
  - E.g., Pentium4 is 32-bit, Alpha is 64-bit
- Memory uses **physical addresses (PA)**
  - $0…2^M–1$ (typically M<N, especially if N=64)
  - 2^M is most physical memory machine supports
- VA→PA at **page** granularity (VP→PP)
  - By "system"
  - Mapping need not preserve contiguity
  - VP need not be mapped to any PP
  - Unmapped VPs live on disk (swap) (or unallocated)
- *What is virtual memory used for?*

Uses of Virtual Memory

- Key uses: **isolation** and **multi-programming**
  - Each app thinks it has 2^N B of memory, its stack starts 0xFFFFFFFF,…
  - Apps prevented from reading/writing each other's memory
    - Can't even address the other program's memory!
- **Protection**
  - Each page with a read/write/execute permission set by OS
  - Enforced by hardware
- **Inter-process communication**.
  - Map same physical pages into multiple virtual address spaces
  - Or share files via the UNIX **`mmap()`** call



Address Translation

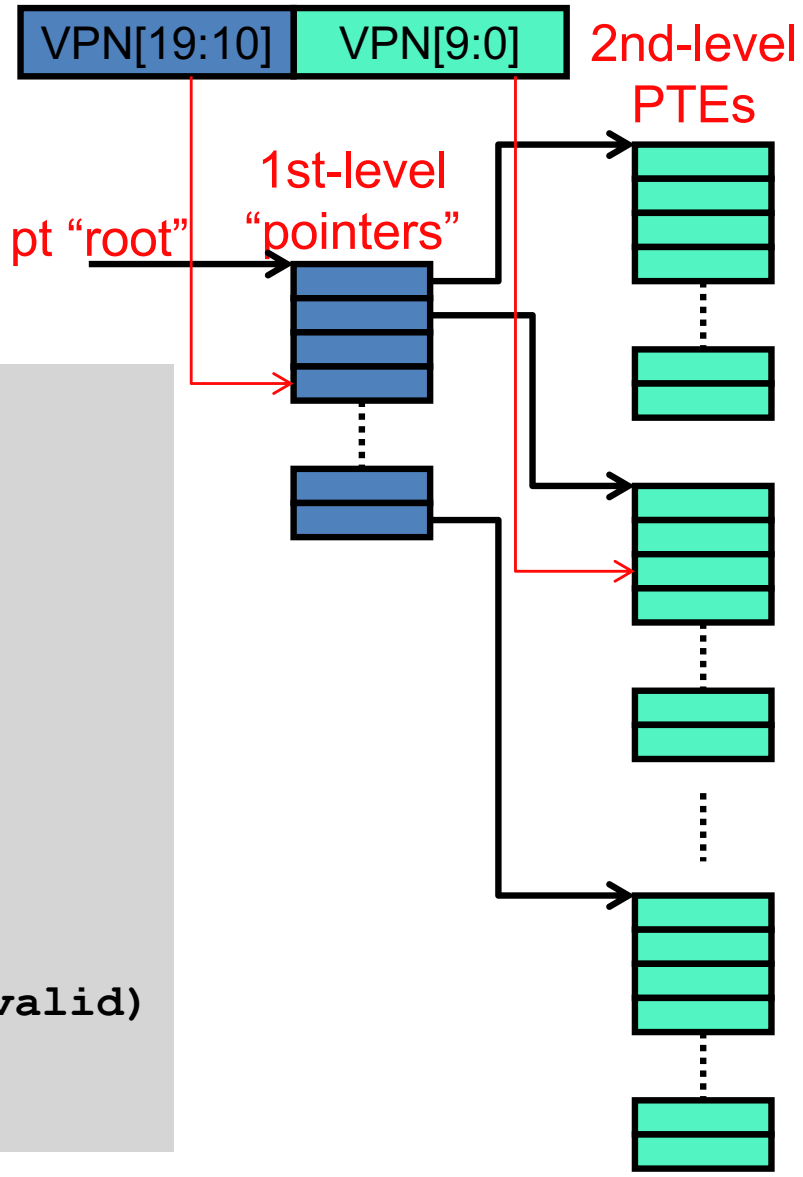


- VA→PA mapping called **address translation**
  - Split VA into **virtual page number (VPN)** & **page offset (POFS)**
  - Translate VPN into **physical page number (PPN)**
  - POFS is not translated
  - VA→PA = [VPN, POFS] → [PPN, POFS]

- Example above
  - 64KB pages → 16-bit POFS
  - 32-bit machine → 32-bit VA → 16-bit VPN
  - Maximum 256MB memory → 28-bit PA → 12-bit PPN

Multi-Level Page Table (PT)

- 20-bit VPN
  - Upper 10 bits index 1st-level table
  - Lower 10 bits index 2nd-level table




```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty;
} PTE;
struct {
   struct PTE ptes[1024];
} L2PT;
struct L2PT *pt[1024];

int translate(int vpn) {
  struct L2PT *l2pt = pt[vpn>>10];
  if (l2pt && l2pt->ptes[vpn&1023].is_valid)
     return l2pt->ptes[vpn&1023].ppn;
}
```

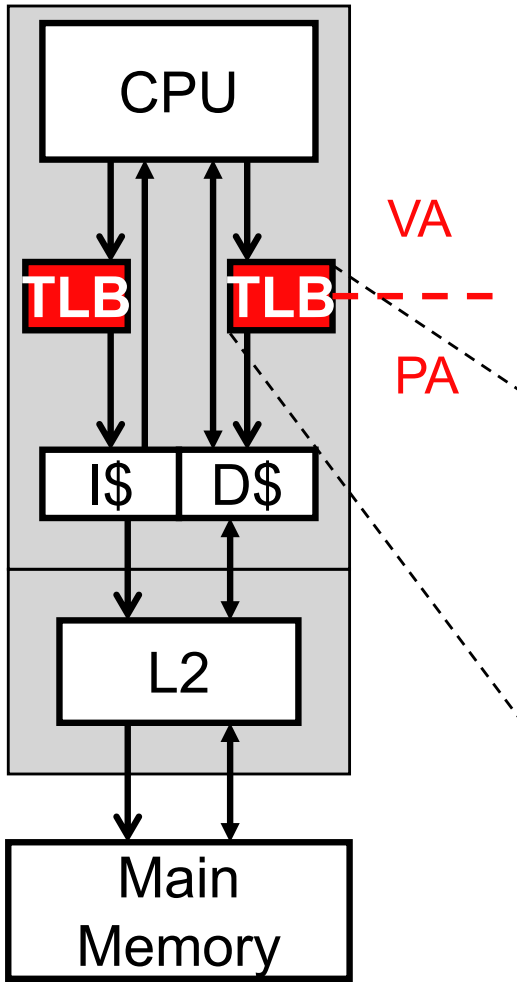
Address Translation Mechanics II

- Conceptually
  - Translate VA to PA before every cache access
  - Walk the page table before every load/store/insn-fetch
- Really? Is this fast?

Address Translation Mechanics II

- Conceptually
  - Translate VA to PA before every cache access
  - Walk the page table before every load/store/insn-fetch
  - Would be terribly inefficient (even in hardware)
- In reality
  - **Translation Lookaside Buffer (TLB)**: cache translations
  - Only walk page table on TLB miss
- Hardware truisms
  - Functionality problem? Add indirection (e.g., VM)
  - Performance problem? Add cache (e.g., TLB)

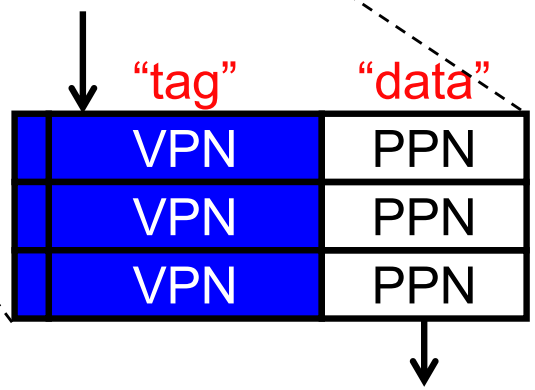
Translation Buffer



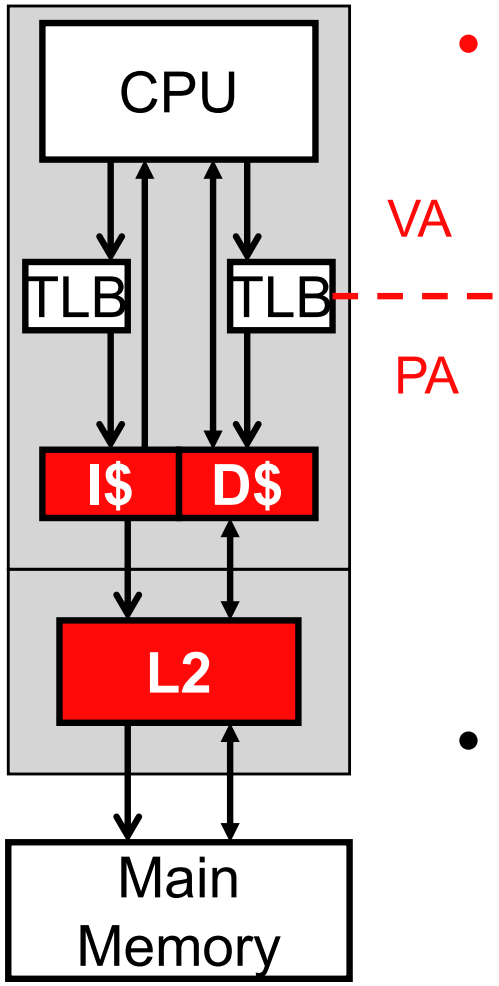
VA

PA

- **Translation buffer (TLB)**
  - Small cache: 16–64 entries
  - Associative (4+ way or fully associative)
  - \+ Exploits temporal locality in page table
  - What if an entry isn't found in the TLB?
    - Invoke TLB miss handler

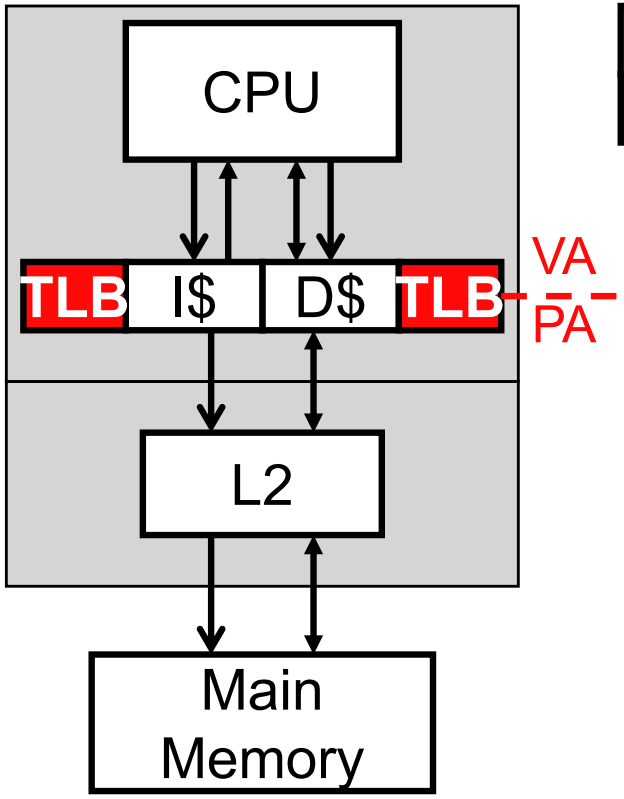


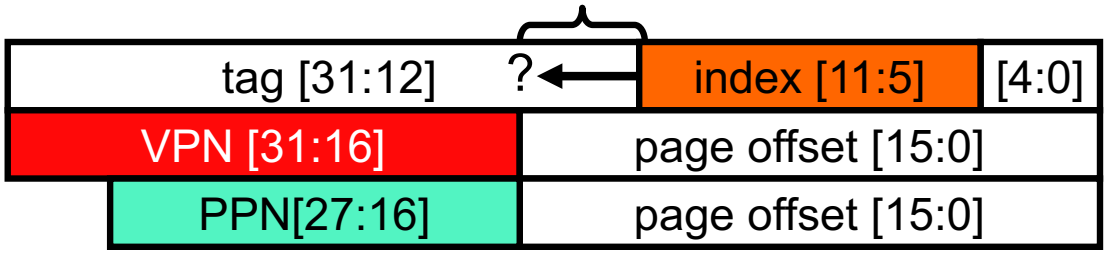
Serial TLB & Cache Access



- **"Physical" caches**
  - – Indexed and tagged by **physical addresses**
  - \+ Natural, "lazy" sharing of caches between apps/OS
    - VM ensures isolation (via **physical addresses**)
    - No need to do anything on context switches
    - Multi-threading works too
  - \+ Cached inter-process communication works
    - Single copy indexed by physical address
  - – Slow: adds at least one cycle to t_{hit}
- Note: **TLBs are by definition virtual**
  - – Indexed and tagged by virtual addresses
  - – Flush across context switches
  - – Or extend with process id tags
- Does this have to be serial?

Parallel TLB & Cache Access





- What about parallel access?
  - What if
    (cache size) / (associativity) ≤ page size
  - Index bits same in virt. and physical addresses!
- Access TLB in parallel with cache
  - Cache access needs tag only at very end
  + Fast: no additional t_{hit} cycles
  + No context-switching/aliasing problems
  - Dominant organization used today

TLB Organization

- **Like caches**: TLBs also have ABCs
  - Capacity
  - Associativity (At least 4-way associative, fully-associative common)
  - What does it mean for a TLB to have a block size of two?
    - Two consecutive VPs share a single tag
  - **Like caches**: there can be L2 TLBs

TLB Misses

- **TLB miss:** translation not in TLB, but in page table
  - Two ways to "fill" it, both relatively fast
- **Software-managed TLB**: e.g., Alpha, Embedded PPC
  - Short (~10 insn) OS routine walks page table, updates TLB
  - \+ Keeps page table format flexible
  - Latency: one or two memory accesses + OS call (pipeline flush)
- **Hardware-managed TLB**: e.g., x86
  - Page table root pointer in hardware register, FSM "walks" table
  - \+ Latency: saves cost of OS call (pipeline flush)
  - Page table format is hard-coded
- **TLB misses becoming a huge problem as physical memory grows**
  - Direct Segments [ISCA'13]

Page Faults

- **Page fault**: PTE not in TLB or page table
  - → page not in memory
  - Starts out as a TLB miss, detected by OS/hardware handler

- **OS software routine**:
  - Choose a physical page to replace
    - **"Working set"**: refined LRU, tracks active page usage
  - If dirty, write to disk
  - Read missing page from disk
    - Takes so long (~10ms), OS schedules another task
  - Requires yet another data structure: **frame map** (why?)
  - Treat like a normal TLB miss from here

Ok, now how do we provide protection?

Page-Level Protection

- **Page-level protection**
  - Piggy-back page-table mechanism
  - Map VPN to PPN + Read/Write/**Execute** permission bits
  - Attempt to execute data, to write read-only data?
    - Exception → OS terminates program
  - ***When are protection properties checked?***

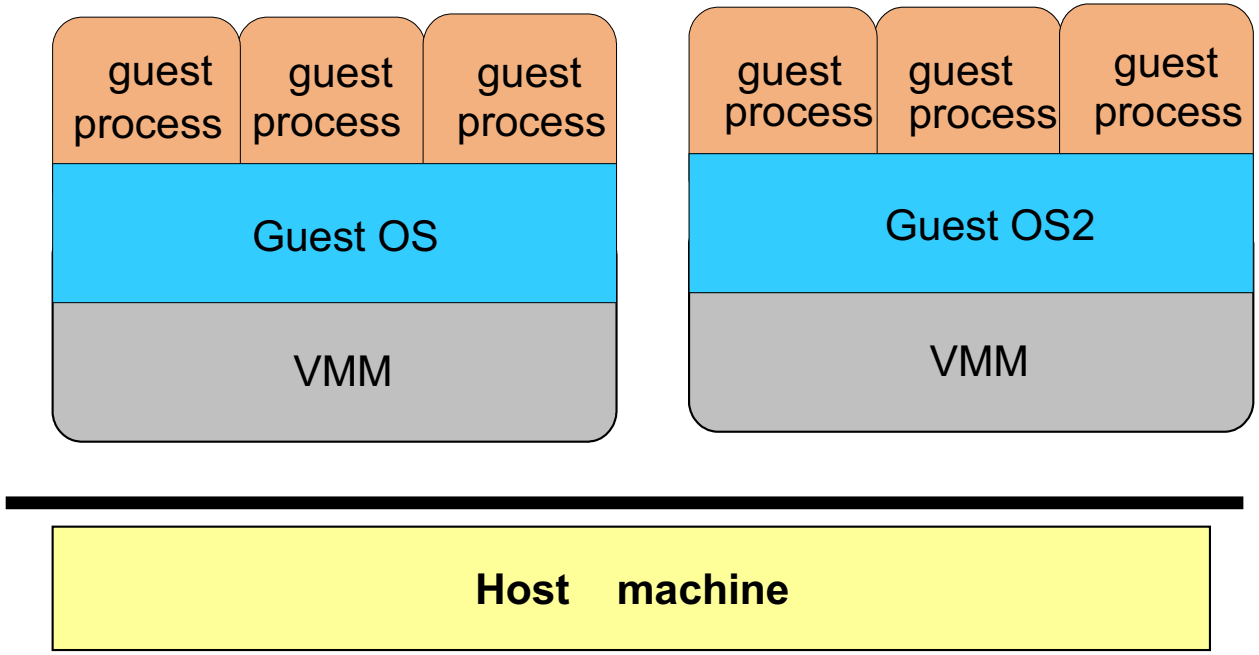
```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty, permissions;
} PTE;
struct PTE pt[NUM_VIRTUAL_PAGES];

int translate(int vpn, int action) {
   if (pt[vpn].is_valid && !(pt[vpn].permissions & action)) kill;
   …
}
```

What could we use protection for?

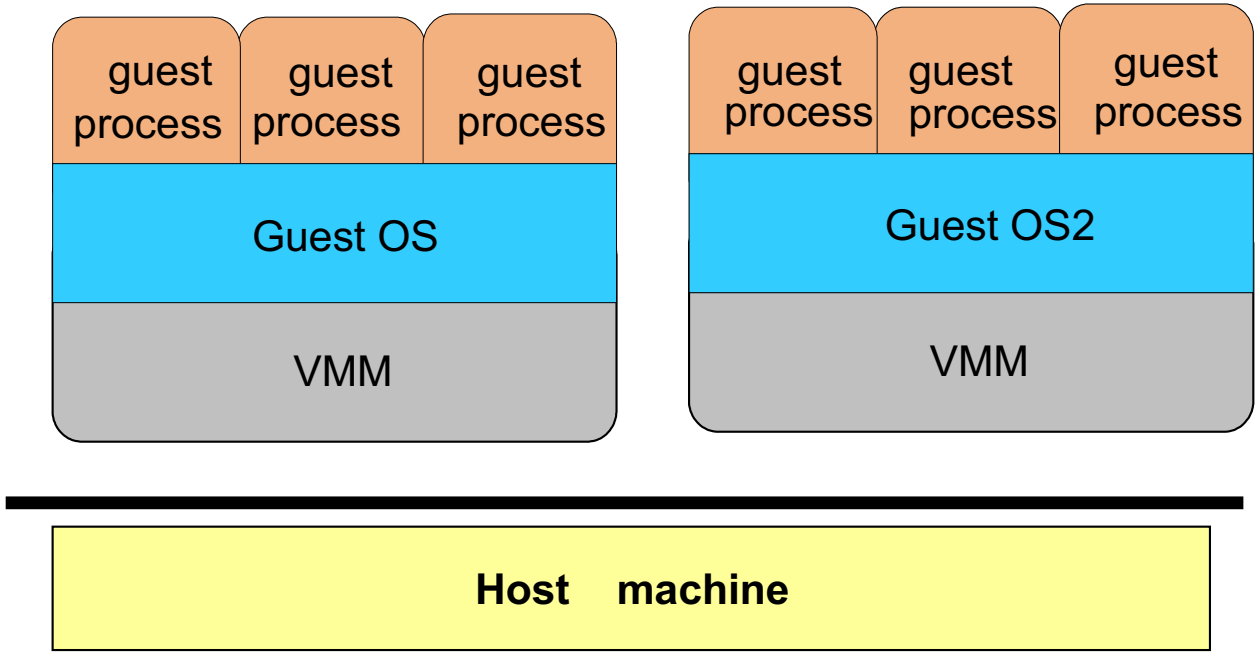
What could we use protection for?

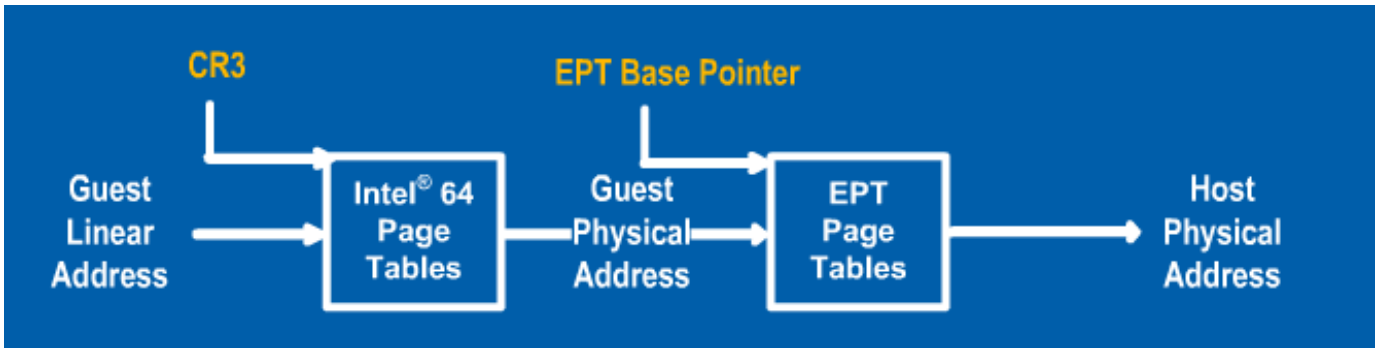
- Virtualization
- Software distributed shared memory
- Garbage collection?
- Optimizations?
- Control program execution in interesting ways

Virtualization



Virtualization





Singularity OS [Larus et al.]

- Can we have a single-address space OS that also supports multiprogramming, is safe etc.
- Exercise: write an OS in a managed language
  - No explicit pointer computation
- Use types and static analysis to isolate program executions
- Minimal low-level code to interact with devices