ECE 571 – Advanced Microprocessor-Based Design Lecture 12

Vince Weaver

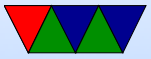
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

10 October 2019

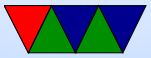
Announcements

- HW#6 will be posted
- Project will be coming up

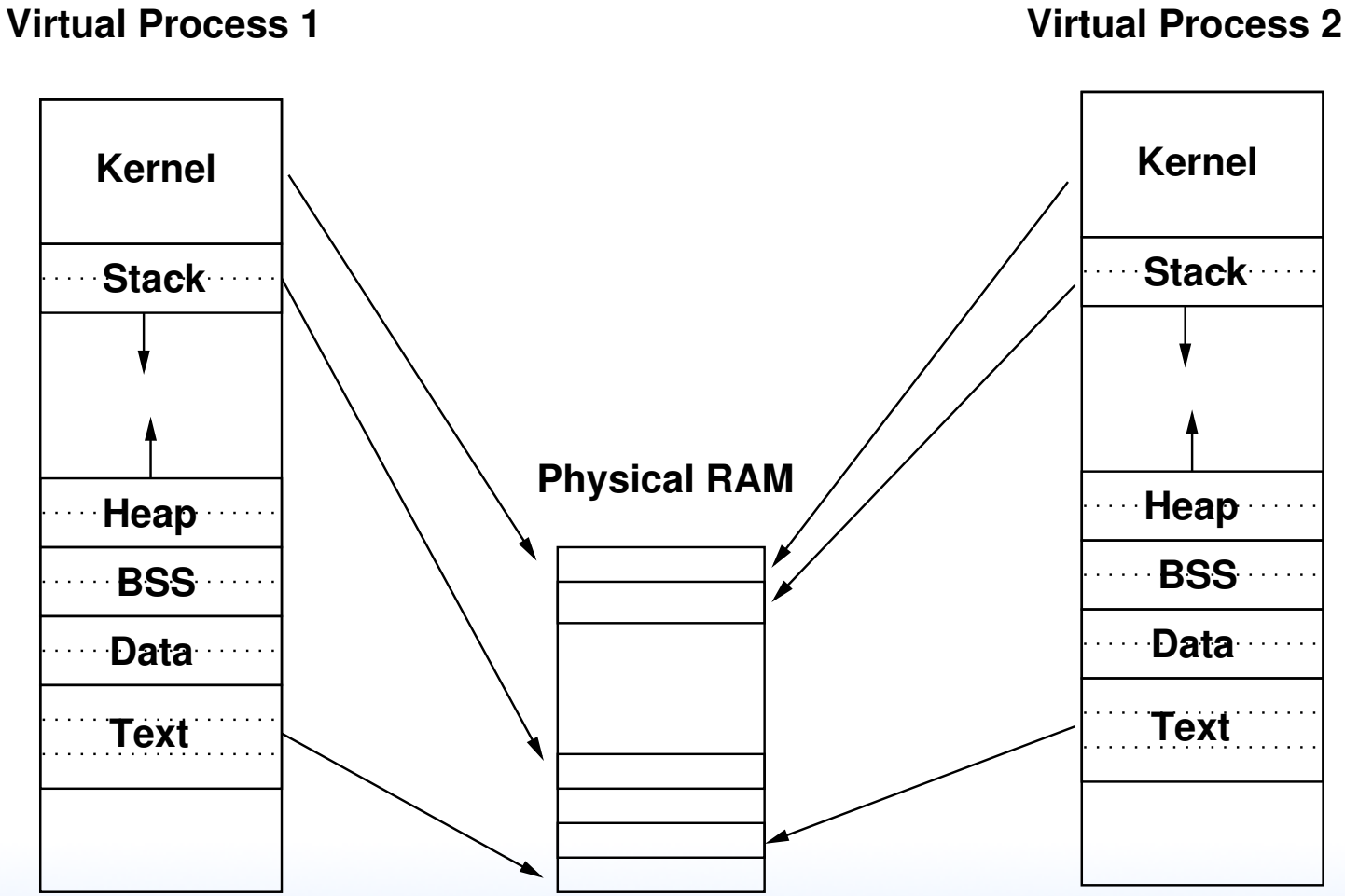


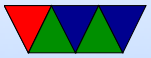
Virtual Memory

- Original purpose was to give the illusion of more main memory than available, with disk as backing store.
- Give each process own linear view of memory.
- Demand paging (no swapping out whole processes).
- Execution of processes only partly in memory, effectively a cache.
- Memory protection
- Security



Diagram

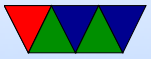

Virtual Process 1
Virtual Process 2
Kernel
Stack
Heap
BSS
Data
Text
Physical RAM
Kernel
Stack
Heap
BSS
Data
Text



Memory Management Unit

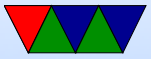
Can run without MMU. There's even MMU-less Linux.

How do you keep processes separate? Very carefully...



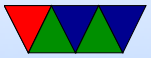
Page Lookup

Simplest would just be a table, with virtual page as index and physical page as value.



Page Tables

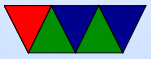
- Collection of Page Table Entries (PTE)
- Some common components:
  - ID of owner
  - Virtual Page Number
  - valid bit,
  - location of page (memory, disk, etc)
  - protection info (read only, etc)
  - page is dirty, age (how recent updated, for LRU)



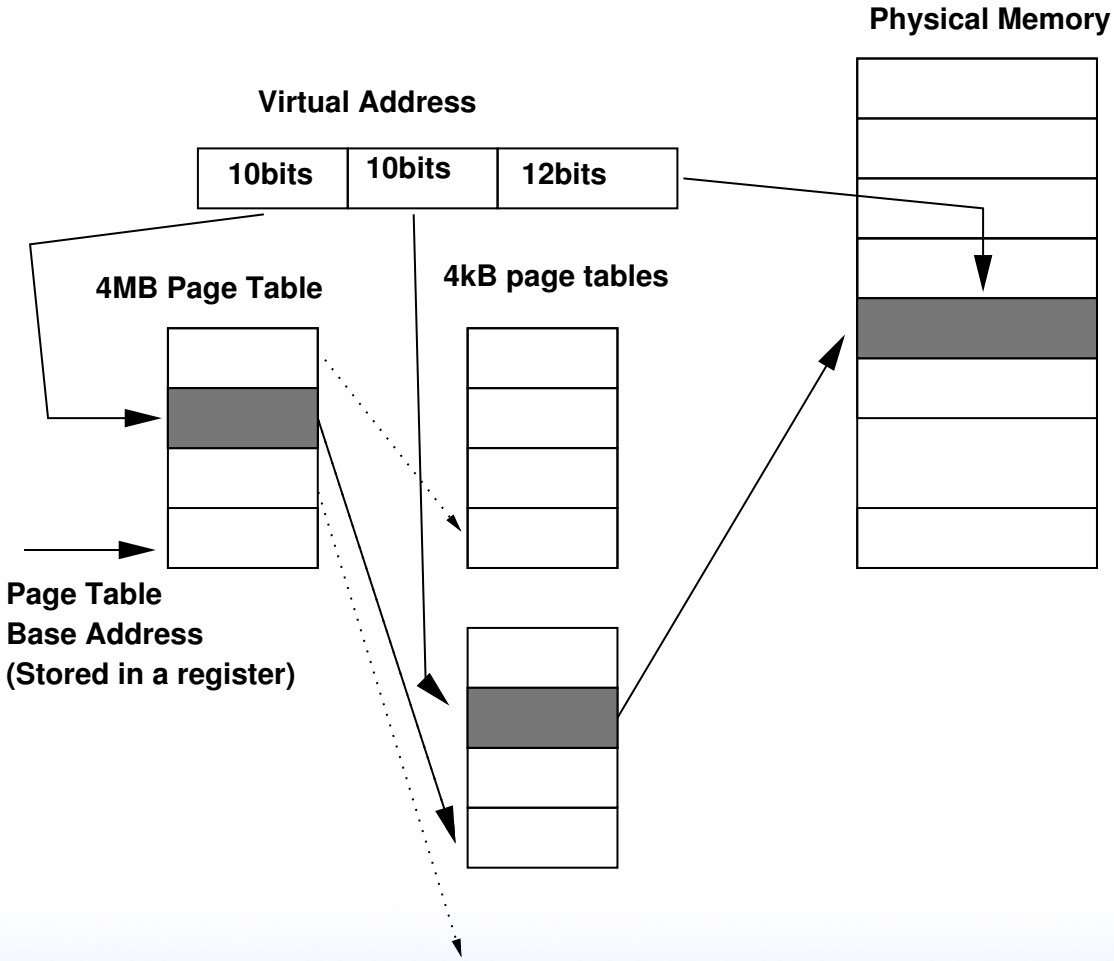
Hierarchical Page Tables

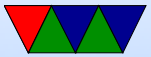
- With 4GB memory and 4kb pages, you have 1 Million pages per process. If each has 4-byte PTE then 4MB of page tables per-process. Too big.

- It is likely each process does not use all 4GB at once. (sparse) So put page tables in swappable virtual memory themselves!
  4MB page table is 1024 pages which can be mapped in 1 4KB page.



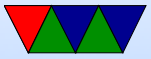
Hierarchical Page Table Diagram





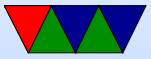
Hierarchical Page Table Diagram

- 32-bit x86 chips have hardware 2-level page tables
- ARM 2-level page tables
- What do you do if you have more than 32-bits?
  - 64-bit x86 has 4-level page tables (256TBv/64TBp)
    44/40-bits?
  - Push by Intel for 5-level tables (128PBv/4PBp)
    57 bits?
  - What if you have unused bits? People use them, causes problems later. AMD64 canonical addresses.

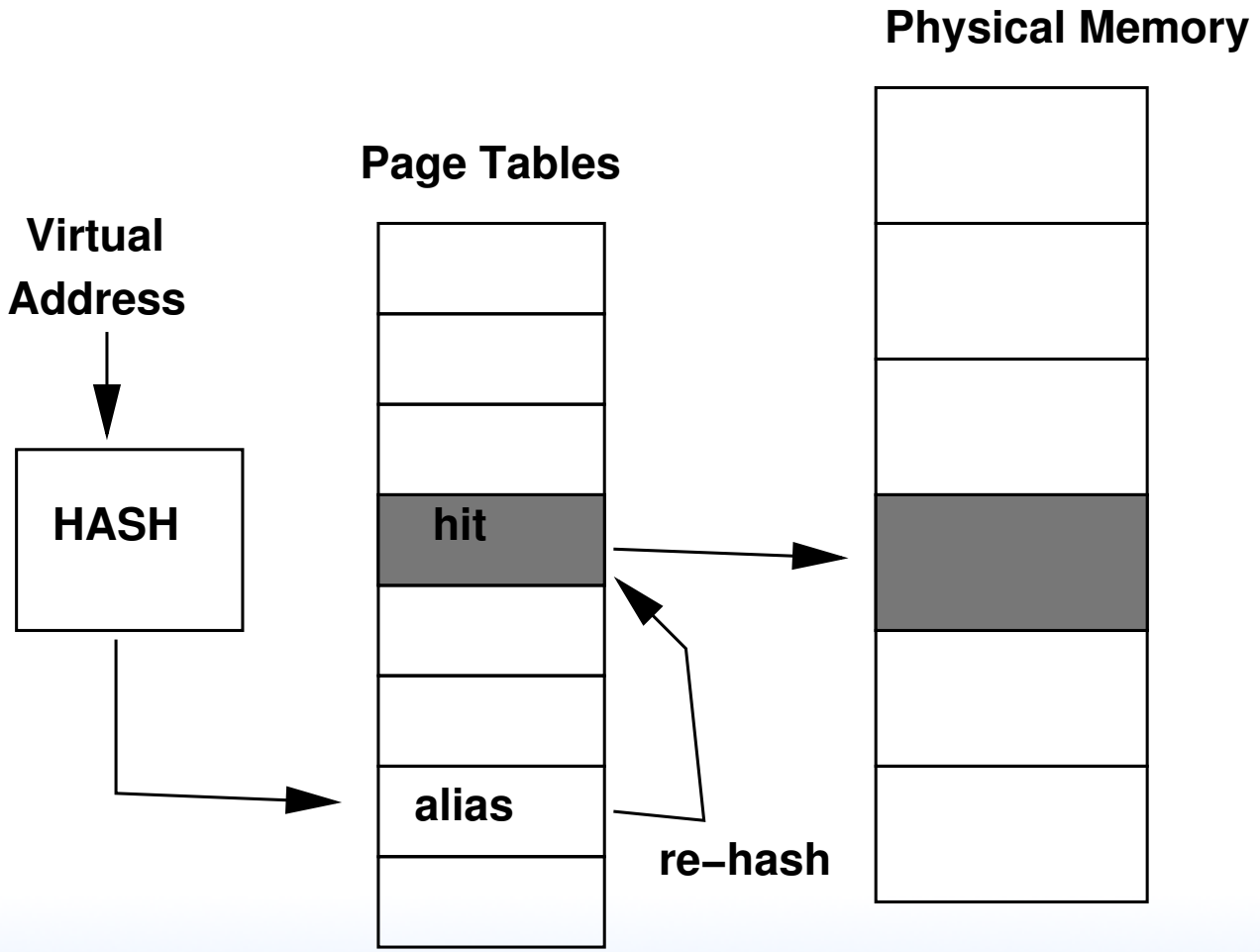


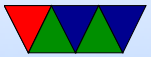
Inverted Page Table

- How to handle larger 64-bit address spaces?
- Can add more levels of page tables (4? 5?) but that becomes very slow
- Can use hash to find page. Better best case performance, can perform poorly if hash algorithm has lots of aliasing.



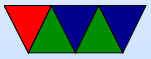
Inverted Page Table Diagram





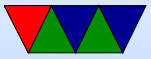
Walking the Page Table

- Can be walked in Hardware or Software
- Hardware is more common
- Early RISC machines would do it in Software. Can be slow. Has complications: what if the page-walking code was swapped out?



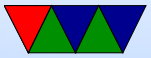
TLB

- Translation Lookaside Buffer
  (Lookaside Buffer is an obsolete term meaning cache)
- Caches page tables
- Much faster than doing a page-table walk.
- Historically fully associative, recently multi-level multi-way
- TLB shootdown – when change a setting on a mapping and TLB invalidated on all other processors



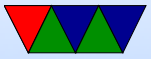
Page Table Caches

- Why walk the whole page table if likely you've walked similar before
- Many processors have page table caches
- Translation Caching: Skip, Don't Walk (the Page Table) (ISCA'10)



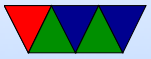
Flushing the TLB

- May need to do this on context switch if doesn't store ASID or ASIDs run out (intel only added ASID support recently)
- Sometimes called a "TLB Shootdown"
- Hurts performance as the TLB gradually refills
- Avoiding this is why the top part is mapped to kernel under Linux (security issue with Meltdown bug!)



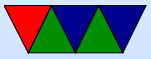
What happens on a memory access

- Cache hit, generally not a problem, see later. To be in cache had to have gone through the whole VM process. Although some architectures do a lookup anyway in case permissions have changed.
- Cache miss, then send access out to memory
- If in TLB, not a problem, right page fetched from physical memory, TLB updated
- If not in TLB, then the page tables are walked



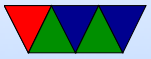
(by the hardware on x86)

- It no physical mapping in page table, then page fault happens



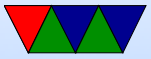
What happens on a page fault

- The OS process structure has info on what memory regions are valid and what should be there (text/data comes from executable on disk, bss zeroed pages, heap/stack might be auto-allocated zeroed pages)
- "minor" – page is already in memory, just need to point a PTE at it. For example, shared memory, shared libraries, etc.
- "major" – page needs to be created or brought in from disk.



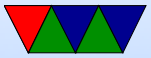
- Demand paging.
- Needs to find room in physical memory.
- If no free space available, needs to kick something out.
  Disk-backed (and not dirty) just discarded.
  Disk-backed and dirty, written back.
- Memory can be paged to disk. Eventually can OOM.
- Memory is then loaded, or zeroed, and PTE updated.
- Can it be shared? (zero page)

- "invalid" – segfault

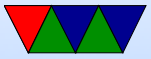


What happens on a fork?

- Do you actually copy all of memory?
  Why would that be bad? (slow, also often exec() right away)
- Page table marked read-only, then shared
- Only if writes happen, take page fault, then copy made
  Copy-on-write

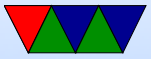


Virtual Memory – Cache Concerns

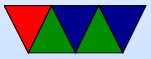


Cache Issues

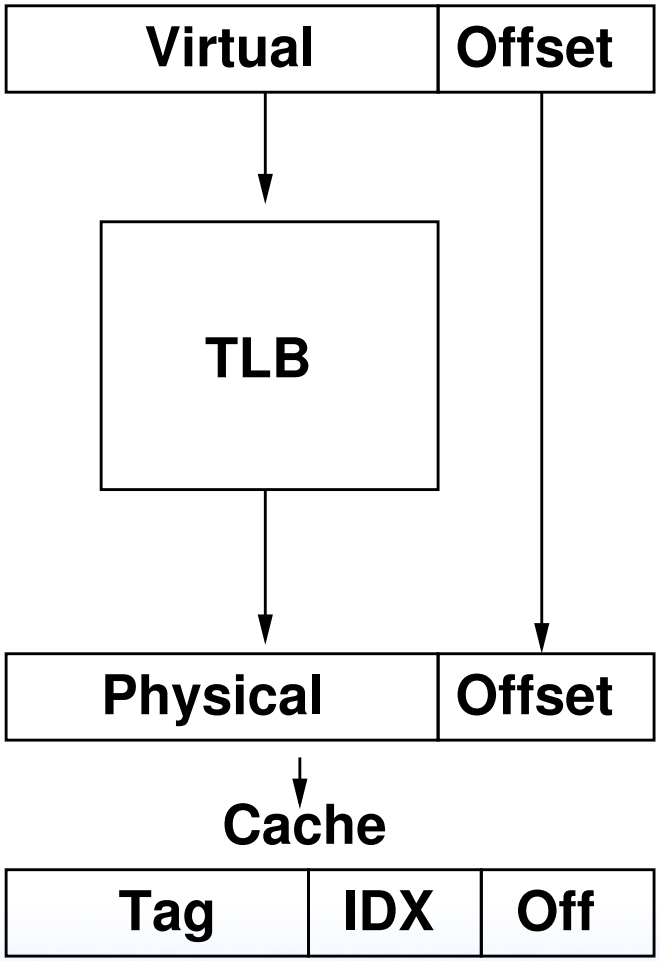
- Page table Entries are cached too
- What happens if more memory can fit in the cache than can be covered by the TLB?
- If you have 128 TLB entries * 4kB you can cover 512kB
- If your cache is larger (say 1MB) then a simple walk through the cache will run out of TLB entries, so page lookups will happen (bringing page table data into cache) and so you do not get maximal usefulness from the cache

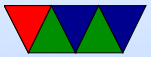


- This has happened in various chips over the years



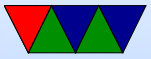
Physical Caches



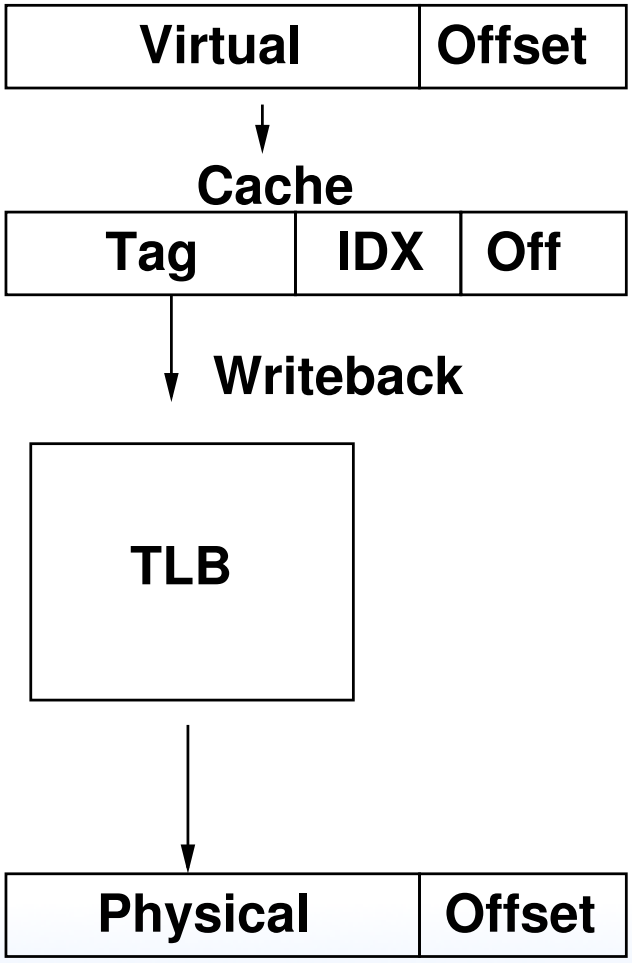


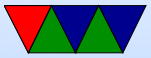
Physical Caches, PIPT

- Location in cache based on physical address
- Can be slower, as need TLB lookup for each cache access
- No need to flush cache on context switch (or ever, really)
- No need to do TLB lookup on writeback



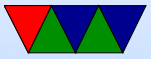
Virtual Caches





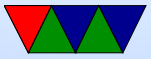
Virtual Caches

- Location in cache based on virtual address
- Faster, as no need to do TLB lookup before access
- Will have to use TLB on miss (for fill) or when writing back dirty addresses
- Cache might have extra bits to indicate permissions so TLB doesn't have to be checked on write
- Aliasing: Homonyms: Same virtual address (in multiple processes) map to different physical page
  - Must flush cache on context switch?

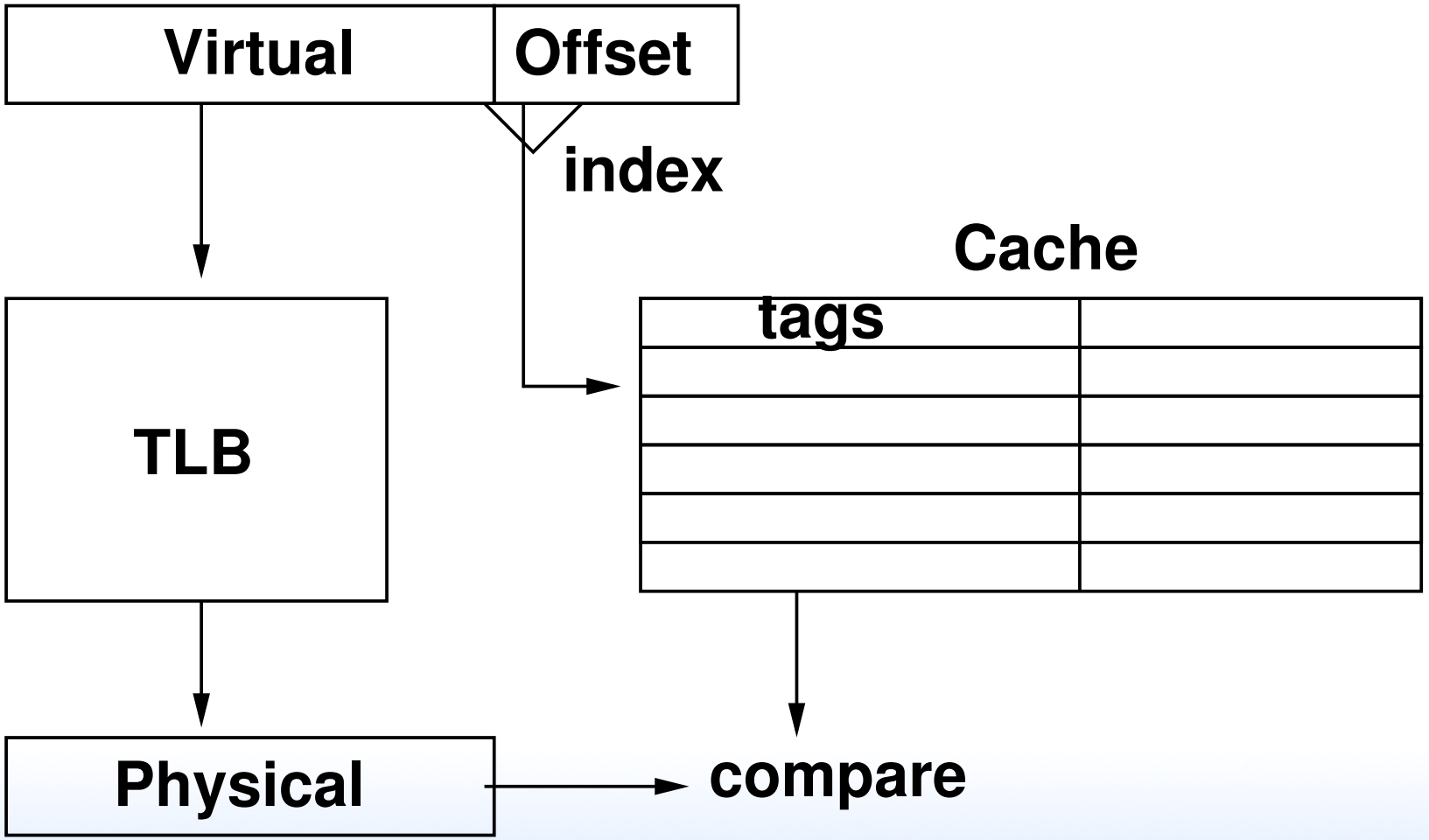


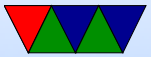
- How to avoid flushing? Have a process-id (ASID). Can also implement sharing this way, by both processes mapping to same virt address.
- Having kernel addresses high also avoids aliasing

- Aliasing: Synonyms: Phys address has two virtual mappings
  - Operating system might use page or cache coloring
- Operating system has to do more work.

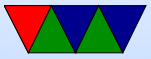


VIPT



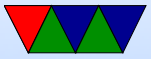


- Cache lookup and TLB lookup in parallel. Cache size + associativity must be less than page size.
- If properly sized (so that the page offset fits completely in the index) then index bits are the same for virt and physical.
- If not sized, the extra index bits need to be stored in the cache so they can be passed along with the tag when doing a lookup
- No need to flush or track ASID on context switch

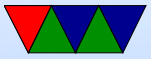


Combinations

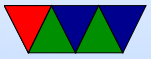
- PIPT – older systems. Slow, as must be translated (go through TLB) for every cache access (don't know index or tag until after lookup)
- VIVT – fast. Do not need to consult TLB to find data in cache.
- VIPT – ARM L1/L2. Faster, cache line can be looked up in parallel with TLB. Needs more tag bits.
- PIVT – theoretically possible, but useless. As slow as



PIPT but aliasing like VIVT.

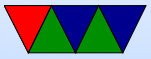


Dealing with Limitations?

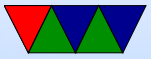


Large Pages

- Another way to avoid problems with 64-bit address space
- Larger page size (64kB? 1MB? 2MB? 2GB?)
- Less granularity. Potentially waste space
- Fewer TLB entries needed to map large data structures
- Compromise: multiple page sizes.
  Complicate O/S and hardware. OS have to find free blocks of contiguous memory when allocating large page.

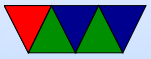


- Transparent usage? Transparent Huge Pages? Alternative to making people using special interfaces to allocate.

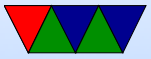


Having Larger Physical than Virtual Address Space

- 32-bit processors cannot address more than 4GB
  x86 hit this problem a while ago, ARM just now
- Real solution is to move to 64-bit
- As a hack, can include extra bits in page tables, address more memory (though still limited to 4GB per-process)
- Linus Torvalds hates this.

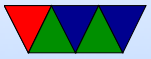


- Hit an upper limit around 16-32GB because entire low 4GB of kernel addressable memory fills with page tables



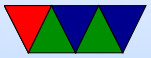
Haswell Virtual Memory

- L1 TLB (4-way associative)
  - 64 4kB
  - 32 2MB
  - 4 1GB
- L2 TLB (1024 entry 8-way associative, combined 4kB and 2M)
- DCache – 32kB/8-way/64B so VIPT possible
  offset=6 bits, lines=64 (6-bits) so offset+index=12 (4096)

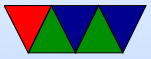


Cortex A9 MMU

- Virtual Memory System Architecture version 7 (VMSAv7)
- page table entries that support 4KB, 64KB, 1MB, and 16MB
- global and address space ID (no more TLB flush on context switch)
- instruction micro-TLB (32 or 64 fully associative)

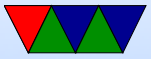


- data micro-TLB (32 fully associative)
- Unified main TLB, 2-way, 2x64 (128 total) on pandaboard
- 4 lockable entries (why want to do that?)
- Supports hardware page table walks



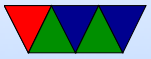
Cortex A9 MMU

- Virtual Memory System Architecture version 7 (VMSAv7)
- Addresses can be 40bits virt / 32 physical
- First check FCSE – linear translation of bottom 32MB to arbitrary block in physical memory (optional with VMSAv7)



Cortex A9 TLB

- micro-TLB. 1 cycle access. needs to be flushed if ASID changes
- fully-associative lockable 4 elements plus 2-way larger. varying cycles access



Cortex A9 TLB Measurement