ECE 571 – Advanced Microprocessor-Based Design Lecture 13

Vince Weaver

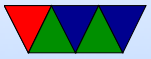
http://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

1 March 2016

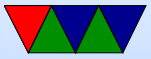
Announcements

- Notes on campus closing
- HW6 posted
- HW4 redux – branch prediction with random on haswell
  It turns out that it can't predict random
  It just turns out there were so many other branches going on that our results lost in noise.



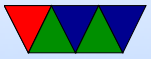
Project

- Document posted to the website.
- Topic selection not required until end of March, but posting it in case anyone wants to get an early start.



HW#5 Review

1. Cache Paramaters: 44-bit, 32kB, 8-way (haswell), 64-Byte
   (a) Offset $= log_2 64 = 6bits$
   (b) Lines $= \frac{\frac{2^{15}}{2^3}}{2^6} = 2^6 = 6bits$
   (c) Tag $=$ 44 bits - 6 bits - 6 bits $=$ 32 bits
   (d) Note: the offset+lines $=$ 12 is probably not a coincidence, as we'll learn about with Virtual Memory. $2^{12} = 4096$ which is a typical page size, which makes PIVT caches much easier

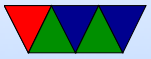


2. Cache Example

(a) ld 0000 080f = line 0, tag 8 = hit
(b) ld ffff ffff = line f, tag ffffff = miss (cold)
(c) ld 0000 0810 = line 1 tag 8 = miss (unknown type), and LRU says we throw out tag a, which is dirty, so writeback

3. Bzip2 on Haswell

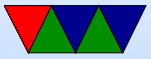
(a) L1-icache = 13k/19B = 0% miss rate
(b) L1-dcache = 311M/6.2B = 5% miss rate
(c) L2 (didn't ask for this) 208M/411M = 50% miss rate



(d) LLC 601k/139M = 0.4% miss rate

(e) Note, l1-dcache is loads. Issue with l1d-stores, in Linux 4.1 (17 Feb 2015) Kleen posted patch to separate SNB evens from HSW in Linux kernel. So your results will change based on kernel version. Annoying. Before there was a l1d-store events

(f) Why do the results not match up? Shouldn't L1-misses be same as L2-accceses? Why would they not match up? Bug in counters, bug in counter selection, other things going on in system, shared resources, chip errata, prefetching, etc. LLC actually uses offcore-



response events

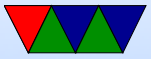
(g) What can we tell about bzip2 behavior? Fits well in icache. Why is L2 so bad?

4. equake_l on Haswell

(a) L1-icache = 14M/1.4T = 0%
(b) L1-dcache = 31B/526B = 5.8%
(c) L2 = 22B/52B = 42%
(d) LLC = 8B/13B=58%

5. bzip2 on Jetson

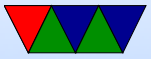
(a) L1-icache = 184k/10B = 0%



(b) L1-dcache-load = 254M/6B = 4%
(c) L1-dcache-store = 56M/2.2B = 2.5%
(d) L2-dcache-load = 28M/330M = 8.5%
(e) L2-dcache-store = 21M/308M = 6.8%

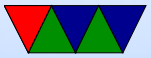
6. Analysis

(a) Bzip2: haswell vs Jetson
HSW: 32kBx8x64B L1, L2=256B, L3=8MB
Jetson: 32kb L1D, 48kBL1I, L2=2MB
l1-dcache-load actually slightly better jetson?
(b) Bzip2 vs equake

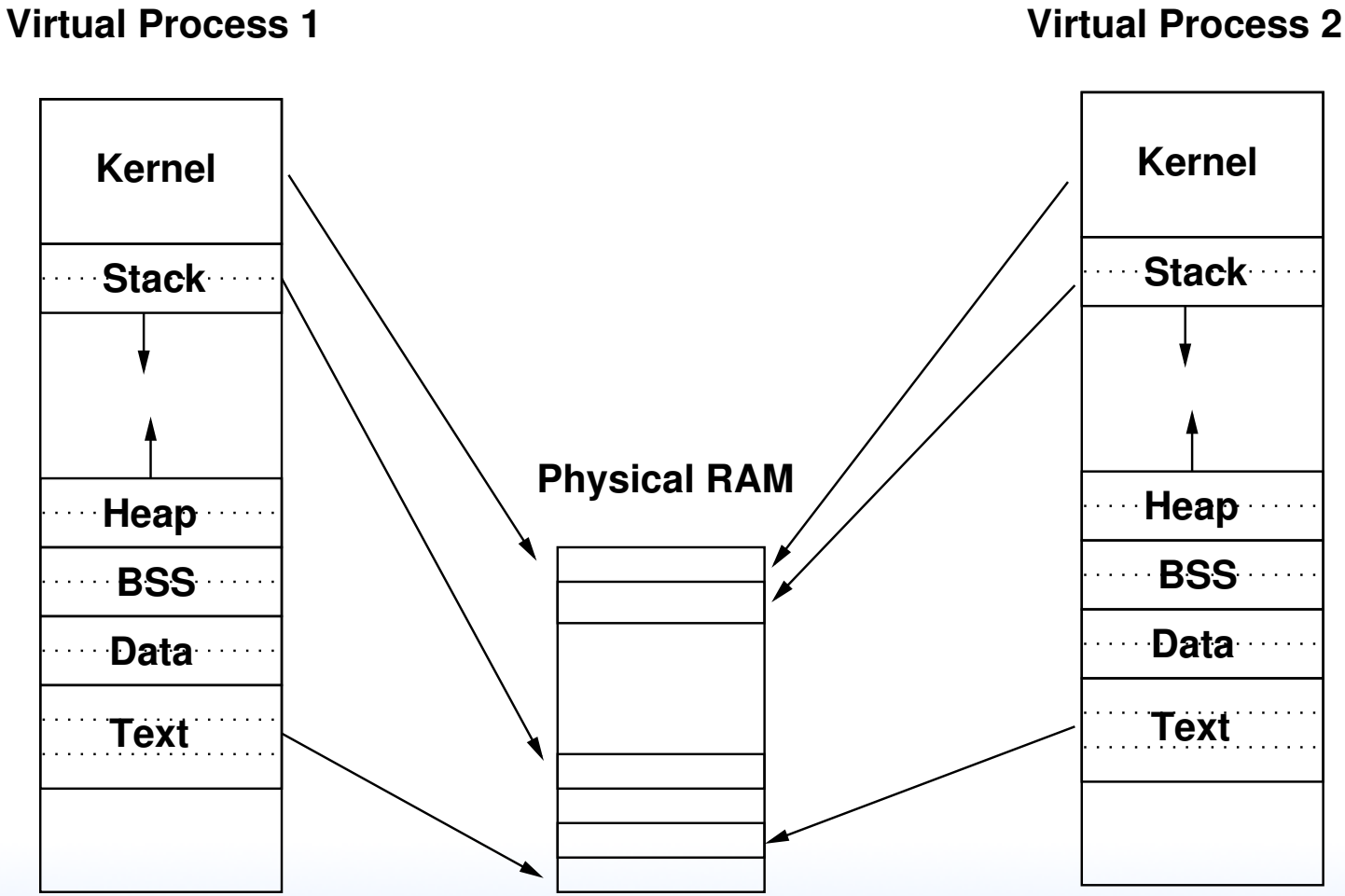


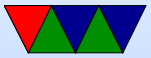
Virtual Memory

- 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



Diagram

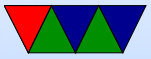

Virtual Process 1
Virtual Process 2
Kernel
Stack
Heap
BSS
Data
Text
Physical RAM
Kernel
Stack
Heap
BSS
Data
Text



Memory Management Unit

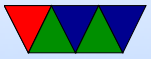
Can run without MMU. There's even MMU-less Linux.

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



Page Table

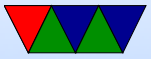
- Collection of Page Table Entries (PTE)
- Some common components: ID of owner, Virtual Page Number, valid bit, location of page (memory, disk, etc), protection info (read only, etc), page is dirty, age (how recent updated, for LRU)



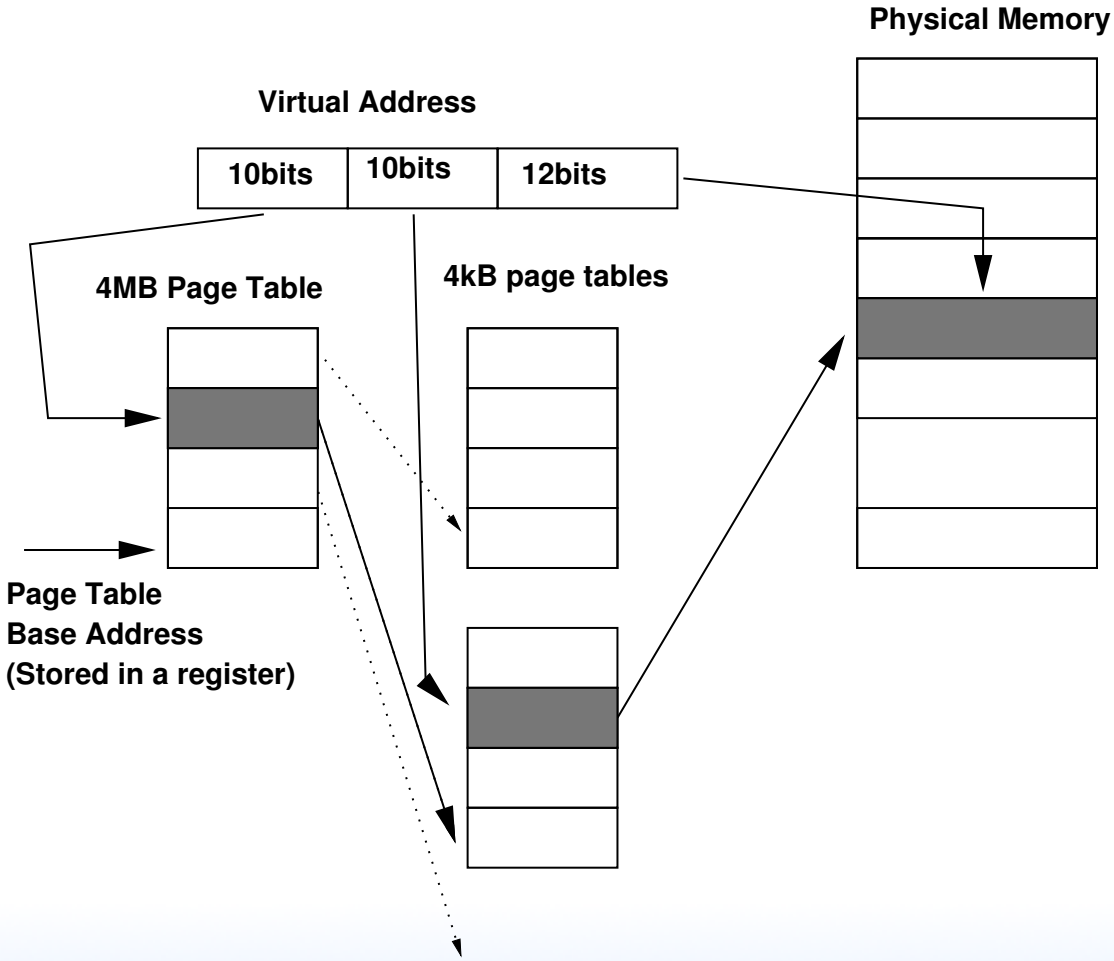
Hierarchical Page Tables

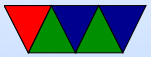
- With 4GB memory and 4kb pages, you have 1 Million pages per process. If each has 4-byte PTE then 4MB of page tables per-process. Too big.

- 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.



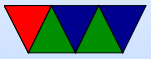
Hierarchical Page Table Diagram





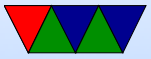
Hierarchical Page Table Diagram

- 32-bit x86 chips have hardware 2-level page tables
- ARM 2-level page tables

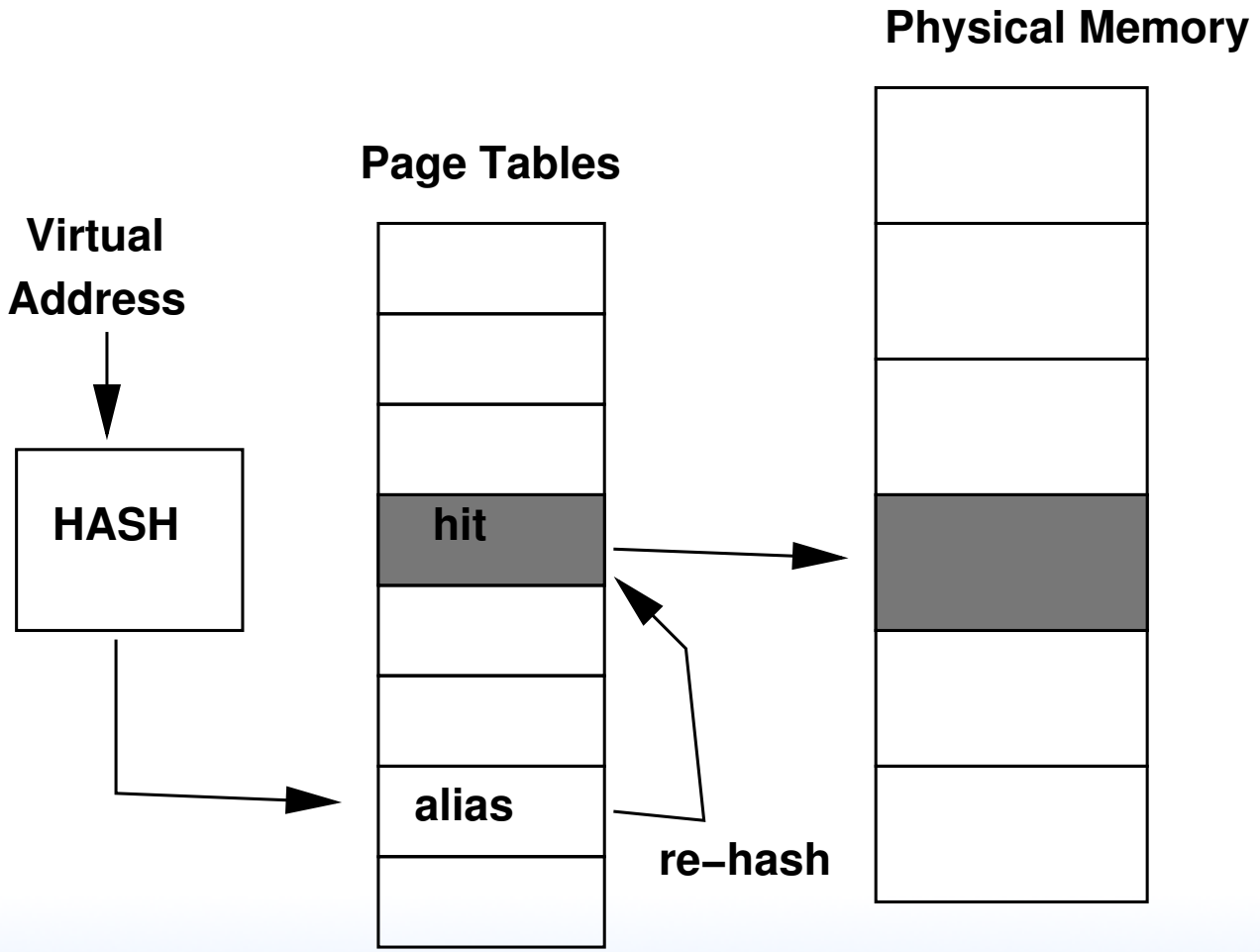


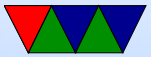
Inverted Page Table

- 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.



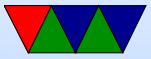
Inverted Page Table Diagram





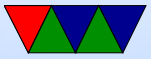
Walking the Page Table

- 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?

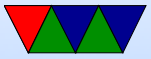


TLB

- 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

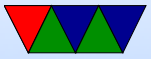


and TLB invalidated on all other processors



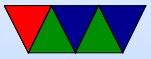
Flushing the TLB

- May need to do this on context switch if doesn't store ASID or ASIDs run out.
- 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

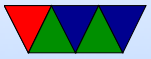


What happens on a memory access

- 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

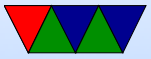


- It no physical mapping in page table, then page fault happens



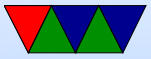
What happens on a page fault

- Walk the page table and see if the page is valid and there
- "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