ECE 571 – Advanced Microprocessor-Based Design Lecture 13

Vince Weaver

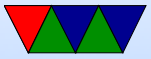
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

21 March 2017

Announcements

- More on HW#6
  - When ask for reasons why cache better than other, give details. Saying Jetson cache is better... not clear. Is it bigger? Faster? Have more ways? Write through?
  - Memory behavior mostly tied to size of data being processed and memory access patterns. This might be tangentially related to branch or integer/floating point, but you need to make an argument why (say FP might be accessing arrays sequentially or integer might

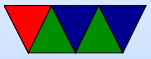


be using less regular data structures).

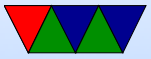
Memory footprint (i.e. fitting in memory) is the most important thing for cache performance, all the other things happen once you start getting misses.

- Grad Class. Not necessarily a right answer, but need to explain.

  Gathering data is often tedious but straightforward. Analyzing/making guesses about the behavior you see is the interesting part. Want to see you are applying what you've learned to the real world results. Not necessarily a "right" answer.



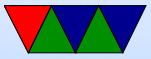
- HW#7 will be after the midterm
- Project topic reminder. Topic due (an e-mail from group) on the 30th, which is next Thursday.



Midterm Review

Closed book/laptop/phone but can have front of one 8.5x11 piece of paper worth of notes if you want.

1. Performance/Benchmarking
   - Be familiar with the general idea of performance counters and interpreting perf results.
   - Benchmark choice: it should match what you plan to do with the computer.
   - Know a little about the difference between integer benchmarks and floating point (integer have

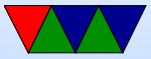


more random/ unpredictable behavior with lots of conditionals; floating point are often regular looped strides over large arrays or data sets)

- Be familiar with concept of skid.

2. Power
   - Know the CMOS Power equation
   - Energy, Energy Delay, Energy Delay Squared
   - Idle Power Question

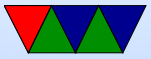
3. Branch Prediction
   - Static vs Dynamic



- 2-bit up/down counter
- Looking at some simple C constructs say expected branch predict rate

4. Cache

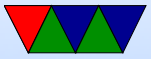
- Given some parameters (size, way, blocksize, addr space) be able to calculate number of bits in tag, index, and offset.
- Know why caches are used, that they exploit temporal and spatial locality, and know the tradeoffs (speed vs nondeterminism)



- Be at least familiar with the types of cache misses (cold, conflict, capacity)
- Know difference between writeback and write-through
- Be able to work a few simple steps in a cache example (like in HW#5)

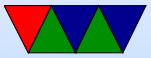
5. Prefetch

- Why have prefetchers?
- Common prefetch patterns?

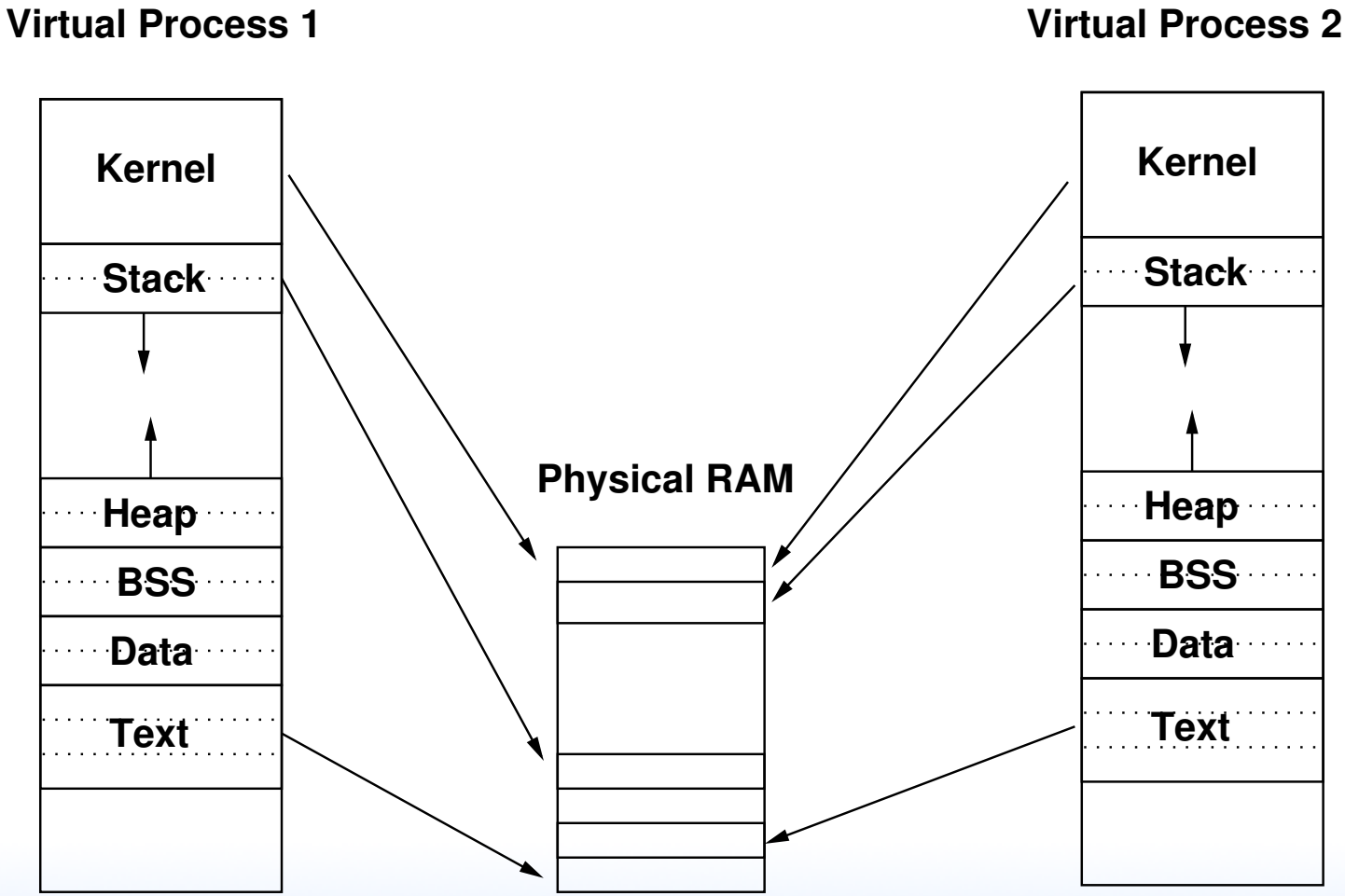


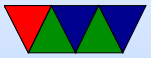
Virtual Memory

- 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

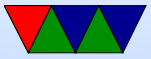

Virtual Process 1
Virtual Process 2
Kernel
Stack
Heap
BSS
Data
Text
Physical RAM
Kernel
Stack
Heap
BSS
Data
Text



Memory Management Unit

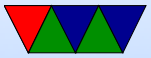
Can run without MMU. There's even MMU-less Linux.

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



Page Table

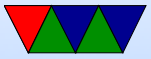
- Collection of Page Table Entries (PTE)
- Some common components:
  - ID of owner
  - Virtual Page Number
  - valid bit,
  - location of page (memory, disk, etc)
  - protection info (read only, etc)
  - page is dirty, age (how recent updated, for LRU)



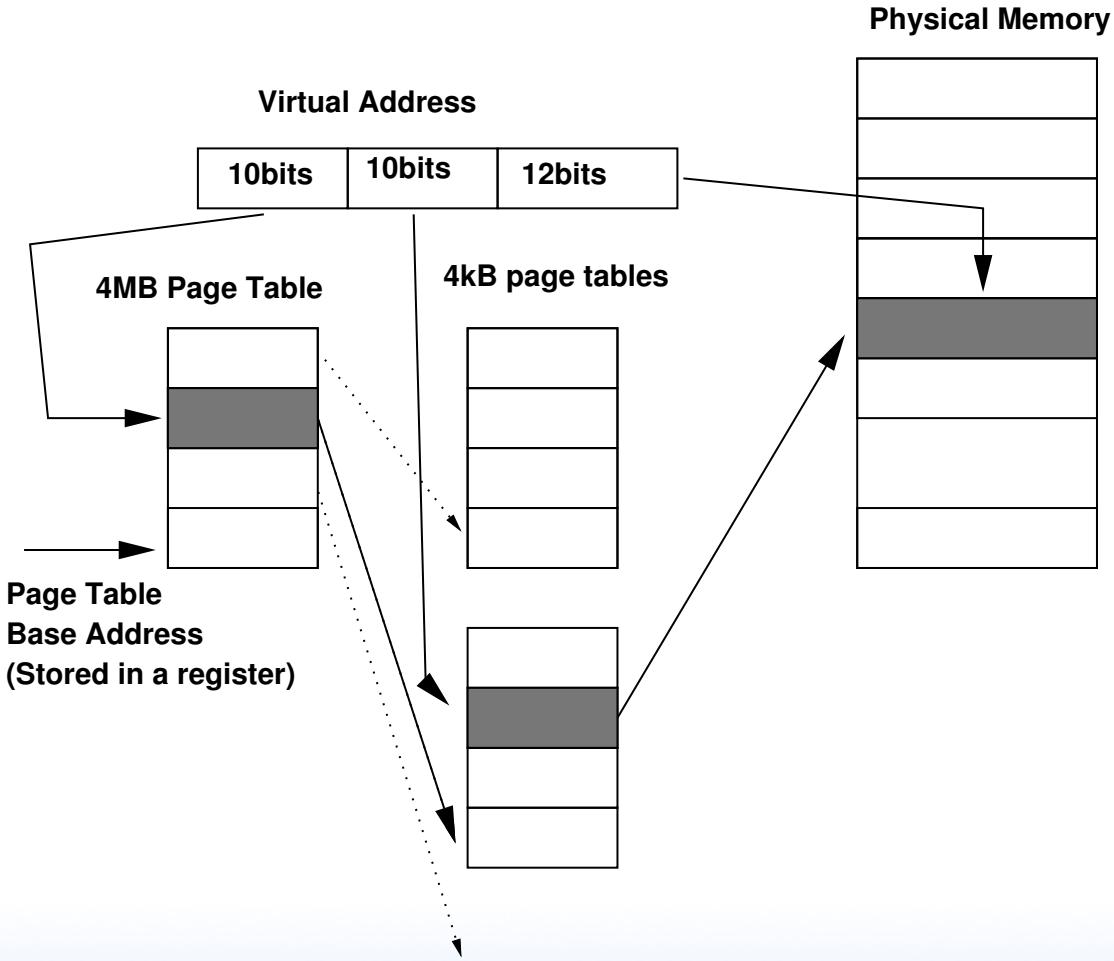
Hierarchical Page Tables

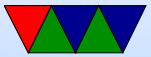
- With 4GB memory and 4kb pages, you have 1 Million pages per process. If each has 4-byte PTE then 4MB of page tables per-process. Too big.

- 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.



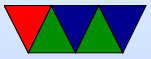
Hierarchical Page Table Diagram



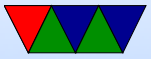


Hierarchical Page Table Diagram

- 32-bit x86 chips have hardware 2-level page tables
- ARM 2-level page tables
- 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 happens when you have unused bits? People try to use them, causes problems later. AMD64 canonical

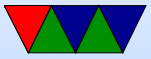


addresses.

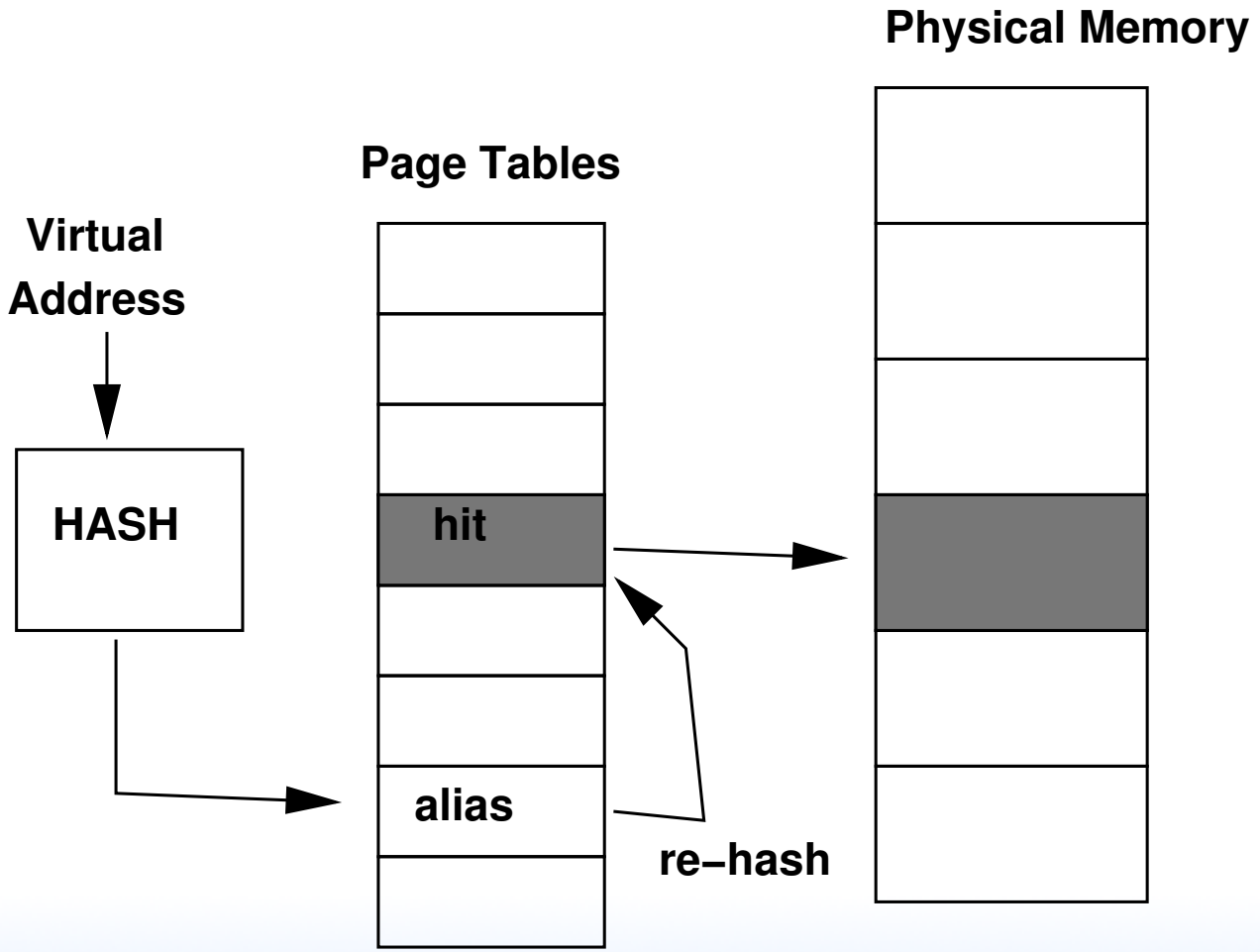


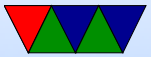
Inverted Page Table

- 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.



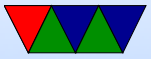
Inverted Page Table Diagram





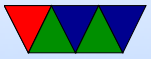
Walking the Page Table

- 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?

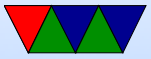


TLB

- 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

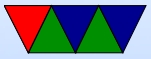


and TLB invalidated on all other processors



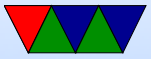
Flushing the TLB

- 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

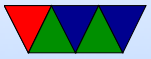


What happens on a memory access

- 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

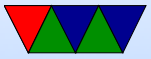


- It no physical mapping in page table, then page fault happens



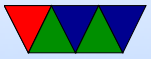
What happens on a page fault

- 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