ECE 531 – Advanced Operating Systems
Lecture 20

Vince Weaver

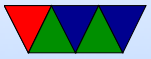
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

24 October 2025

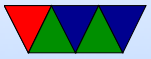
Announcements

- HW#5 still being graded
- Midterms still being graded
- HW#6 will be posted



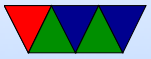
Virtual Memory

- In the early days of computing RAM was an extremely limited resource
- Could you give the illusion of more RAM by swapping RAM to/from disk?
- Can manually do this in software (overlays) but could HW/OS do this transparently for you?
- Despite the complexity, was implemented in the 1960s
- Pretty much all modern CPUs and OSes support this
- It enables many other features too



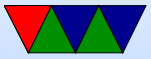
Virtual Memory in ECE Classes

- ECE471 (embedded) – mention it, to differentiate bare metal vs OS layout
- ECE574 (cluster computing) – mention it, has some performance implications
- ECE571 (advanced processors) – look at it in detail as modern memory / cache behavior tightly tied to it
- ECE531 (operating systems) – we are the OS, our job to run it all so we actually have to understand it

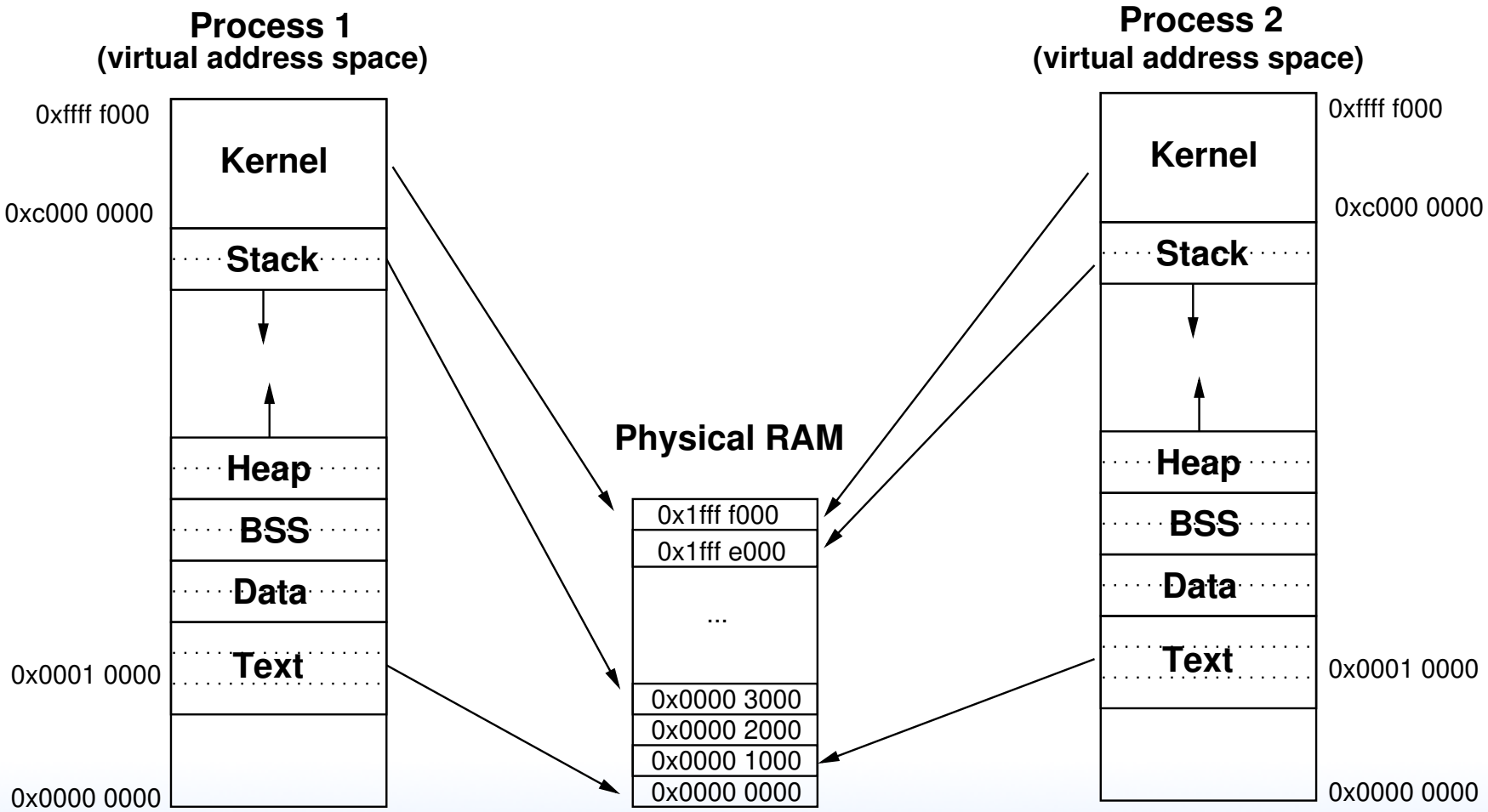


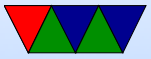
Virtual Memory – High Level

- Computer programs see a fake/virtual view of memory
- This memory is split up into chunks called pages
- A translation layer is set up that translates virtual pages to the actual physical pages of RAM
- Every single load and store from your program passes through this translation layer



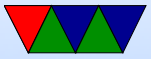
Virtual to Physical Mapping Diagram





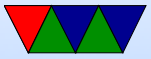
Notes on Previous Diagram

- Physical RAM can be much smaller than Virtual (in this example, 512MB vs 4GB)
- Everything lives in Physical RAM, it's only given the illusion that virtual memory exists because the CPU translates the virtual address pointers to physical on-the-fly
- Each process can use the same identical virtual addresses, they don't conflict because behind the scenes they map to separate physical addresses



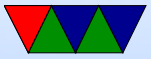
Physical Memory Downsides

- Never enough memory
- No memory protection between programs
- Memory fragmentation
- Programs need to be PIC (position independent code)
- Programs need to be totally loaded into memory before execution, stack fixed size



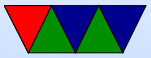
Virtual Memory Upsides

- Give the illusion of more memory than available, with disk as backing store.
- Memory protection
- Give illusion of contiguous memory to avoid fragmentation
- Demand paging (no swapping out whole processes), only load parts of programs as needed
- Give each process own linear view of memory.



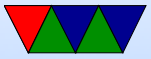
Virtual Memory Downsides

- Complicated hardware/software
- Potentially slower, lots of indirection on every memory access
- If run out of physical memory can end up swap storm, machine unusable



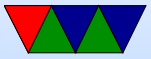
Memory Management Unit

- In very old days was a separate (optional!) chip
- Can run run OS without an MMU?
  - There's MMU-less Linux (uclinux)
  - How do you keep processes separate? Very carefully...



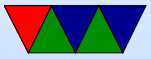
Page Lookup

- Simplest would just be a table, with virtual page as index and physical page as value
- Ends up being more complex than this



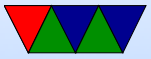
Page Tables – Hold Virt/Phys Mappings

- Collection of Page Table Entries (PTE)
- Some common components: (TODO: look up actual values on ARM/Linux)
  - 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)



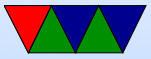
Page Table Encoding

- If 4k pages, bottom 12 bits of mappings unused
- Could you squeeze all the PTE info in those bits?
- Can be complex, ARM32 and ARM64 have really complicated page table setups



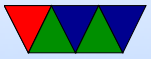
Page Table Issues – Size

- With 4GB memory and 4kB pages, you have 1 Million pages per process.
- With 4-byte PTE then 4MB of page tables per-process. Too big.
  (or it was in the 1990s when your computer maybe only had 4MB RAM total)

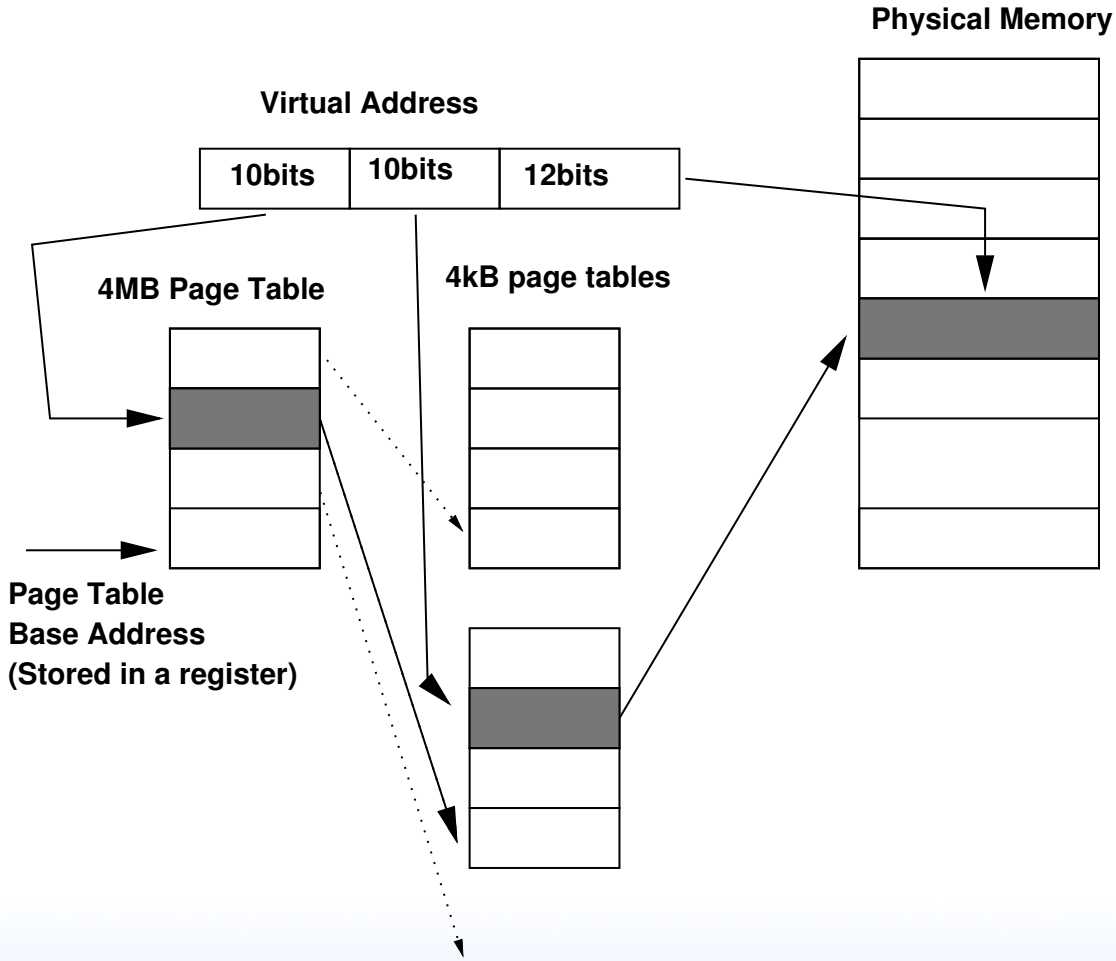


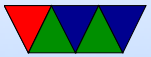
Hierarchical Page Tables

- It is likely each process does not use all 4GB at once (sparse)
- Put page tables in swappable virtual memory
- 4MB page table is 1024 entries which can be mapped by one 4kB page
- In this particular example, the smallest possible process uses only 8k (2 pages) of page tables, rather than 4MB



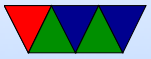
Hierarchical Page Table Diagram (32-bit)





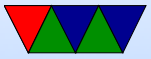
32-bit Hierarchical Page Tables

- 32-bit x86 chips have hardware 2-level page tables
- ARM32 2-level page tables (3-level if use PAE), but more complicated than the example we are using



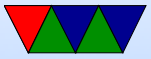
64-bit System Page Tables

- Virtual address space much bigger, how to handle?
- Physical memory usually not 64-bit yet, often from 40-48 bits
- Can we just add more levels of page tables?
  - 64 bit x86 has 4-level page tables (256TBv/64TBp) 44/40 bits
  - Push by Intel for 5-level tables (128PBv/4PBp) 57 bits
- (Aside) Linux assumes 5 levels of page tables and collapses down unused ones



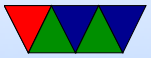
Another approach (Historical) Inverted Page Table

- IBM Power, Ultrasparc, ia64
- 4/5 level tables can be slow
- Have one single mapping, page mapping for each physical to virtual page
- Almost like having a large software TLB
- Note: Linus Torvalds wasn't a fan
- A linear search to find a mapping is slow, so can use hash to find page. Better best case performance, can

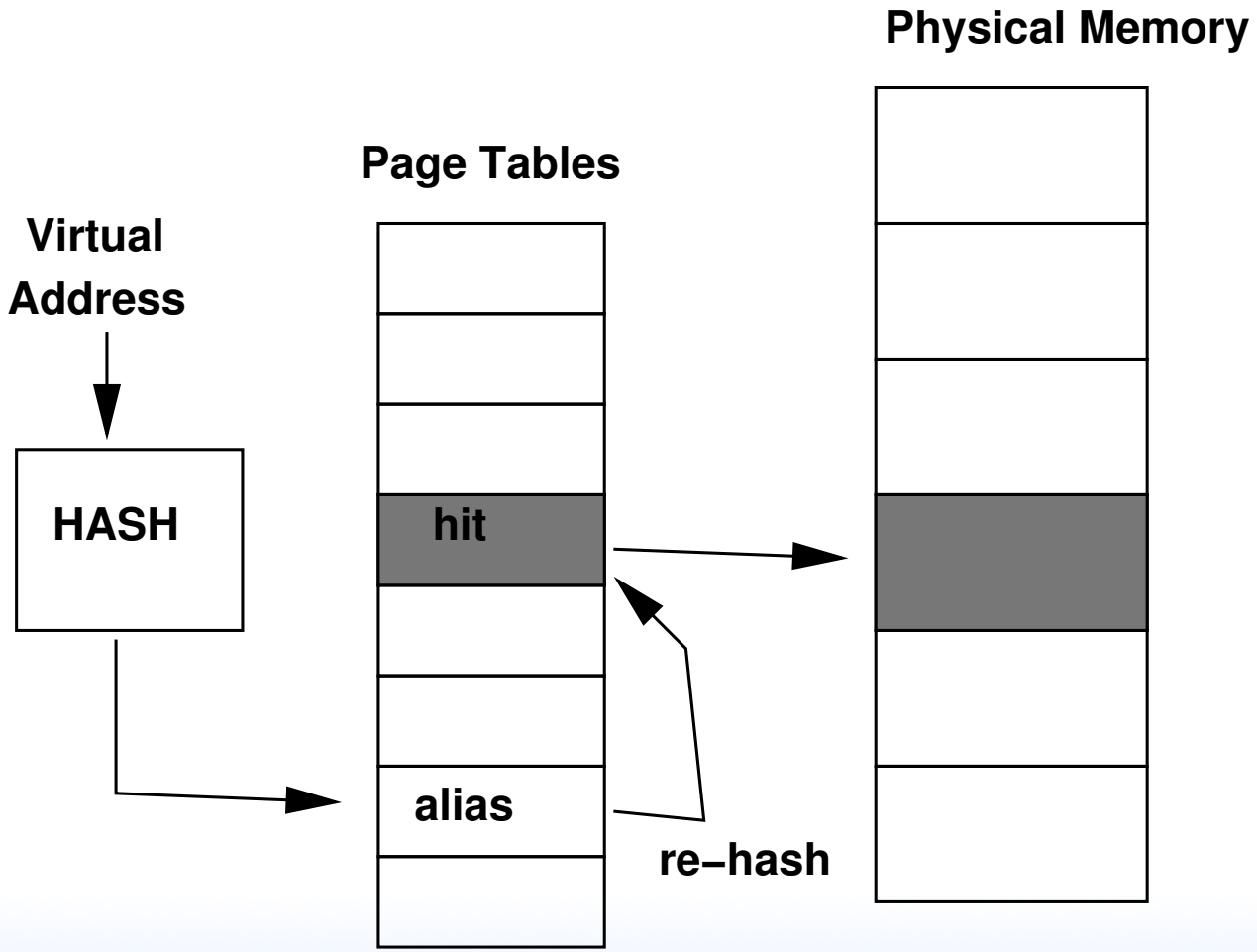


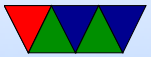
perform poorly if hash algorithm has lots of aliasing.

- Also has poor cache performance due



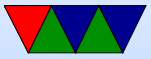
Inverted Page Table Diagram





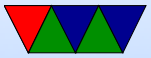
Walking the Page Tables

- Can be walked in Hardware or Software
- Hardware is more common
  - Generally have a register pointing to the current process page table (CR3 ox x86)? CR4? TTBR0?
  - Saved/restored on context switch
- Early RISC machines would do it in Software.
  - Can be slow
  - Has complications: what if the page-walking code was swapped out?



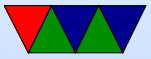
Translation Lookaside Buffer (TLB)

- How can you avoid multiple extra memory lookups on every memory access?
- Like all other CPU memory solutions, a cache
- (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/way



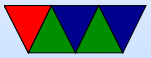
Page Tables in Cache?

- Page tables live in memory like any other data structure
- Then can end up in the CPU cache as well, which can help performance
- Some chips were designed where TLB wasn't big enough to cover all of the cache so a walk through the cache would end up causing a lot of TLB misses



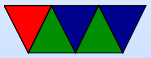
Page Table Caches

- Why walk the whole page table if likely you've walked similar before
- Many processors have page table caches
  - AMD Page Walking Caches (unified page table)
  - Intel Page-structure caches (split translation cache)
  - ARM unified translation cache
- Translation Caching: Skip, Don't Walk (the Page Table) (ISCA'10)



Flushing the TLB

- If page tables no longer accurate may need to clear out (flush) the TLB
- Sometimes called a "TLB Shootdown"
- Hurts performance as the TLB gradually refills



When do we flush TLB?

- May need to do this on context switch if doesn't store ASID or ASIDs run out (ASID=Address Space ID, intel only added support recently)
- Avoiding this is why the top part is mapped to kernel under Linux (security issue with Meltdown bug!)