

ECE 571 – Advanced Microprocessor-Based Design Lecture 18

Vince Weaver

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

vincent.weaver@maine.edu

18 October 2024

Announcements

- HW#6 was posted
- Project was posted
- Midterm on the 30th?



Virtual Memory

- In the early days of computing RAM was an extremely limited resource
- Could you give the illusion of more RAM by transparently swapping data to/from disk?
- Despite the complexity, this was implemented in the 1960s
- Pretty much all modern CPUs and operating systems support this
- It enables many other features too



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



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



Where does your program end up in Memory Anyway?

```
int a[100]; // global uninitialized, BSS
int y=3;    // global initialized, Data

int main(int argc, char **argv) {
    // trivia, argv lives above stack
    int x=5; // local initialized, Stack

    char *c=malloc(100); // malloc in Heap

    printf("%d %p\n",x,&x); // code in Text

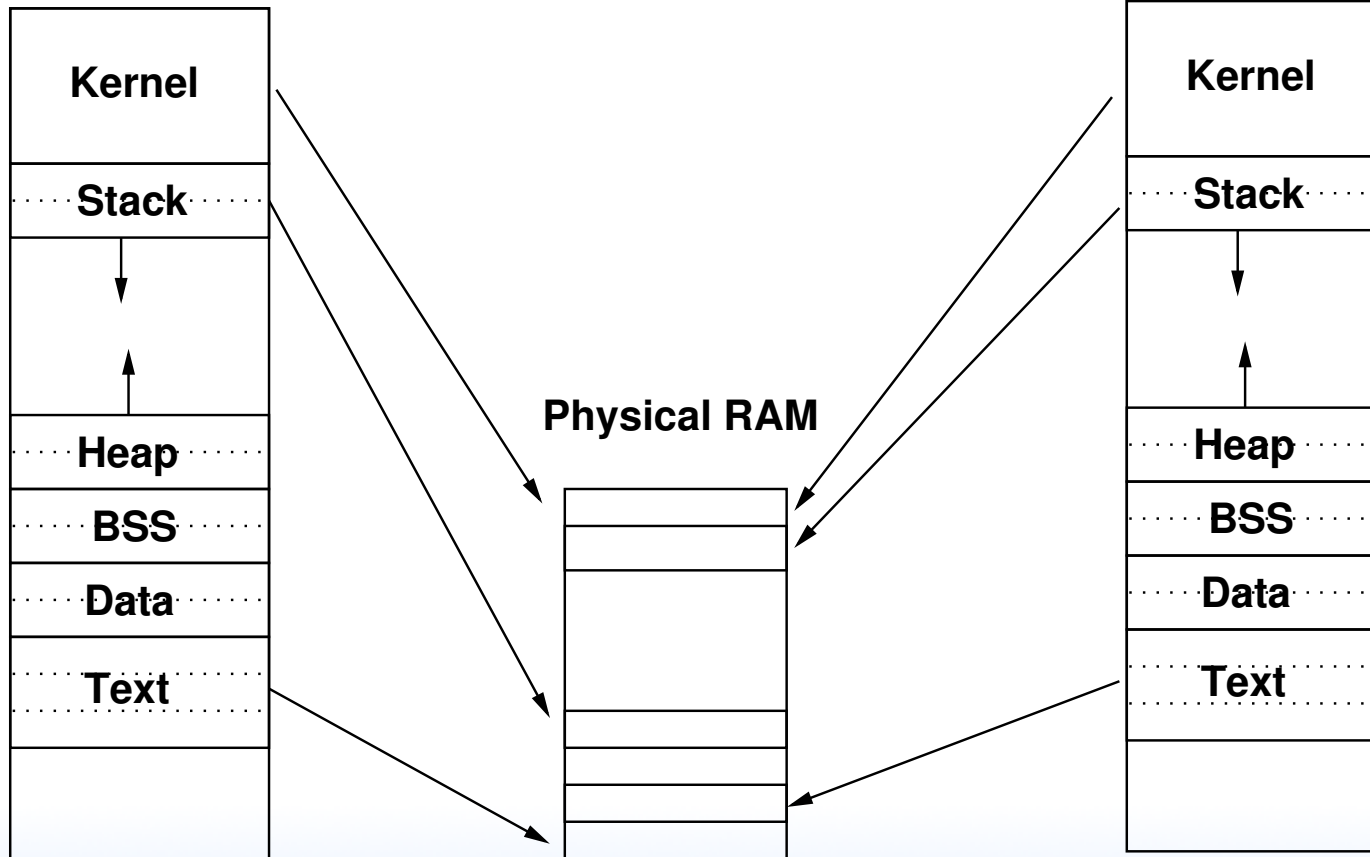
    return 0;
}
```



Diagram

Virtual Process 1

Virtual Process 2



Memory Management Unit

- In very old days was a separate (optional!) chip
- Can you run OS without an MMU?
 - 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.



Page Tables – Hold Virt/Phys Mappings

- Collection of Page Table Entries (PTE)

notes for next time, look up format amd64 they are 64-bits can contain write-back vs write-through info (For mmio areas?)

- Some common components:
 - ID of owner
 - Virtual Page Number

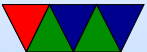


- valid bit,
- location of page (memory, disk, etc)
- protection info (read only, etc)
- page is dirty, age (how recent updated, for LRU)



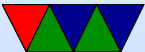
Page Table Issues – Size

- 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.

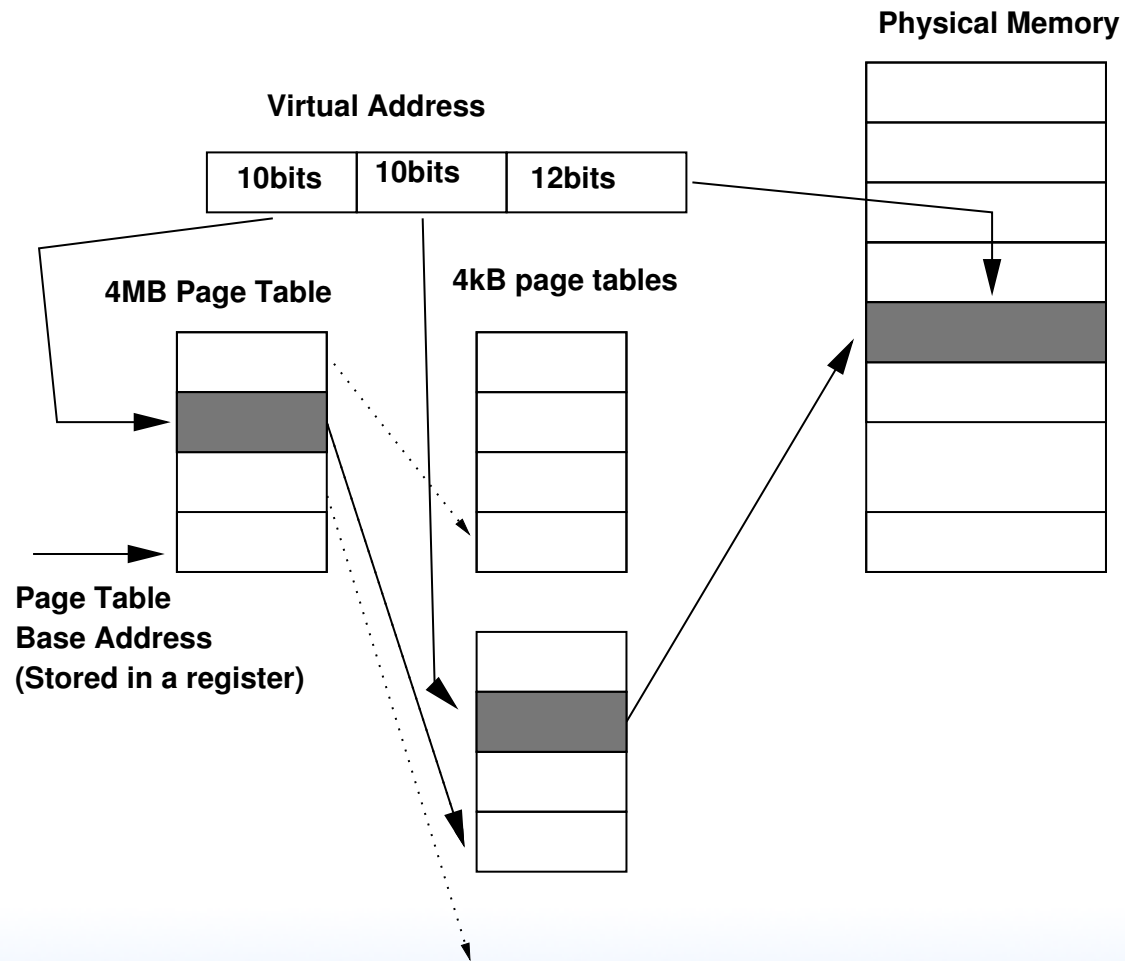


Hierarchical Page Tables

- It is likely each process does not use all 4GB at once.
(sparse)
- 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



64-bit Systems

- 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 (256TB_v/64TB_p)
44/40-bits?
 - Push by Intel for 5-level tables (128PB_v/4PB_p)
57 bits?



Another approach (Historical) – Inverted Page Table

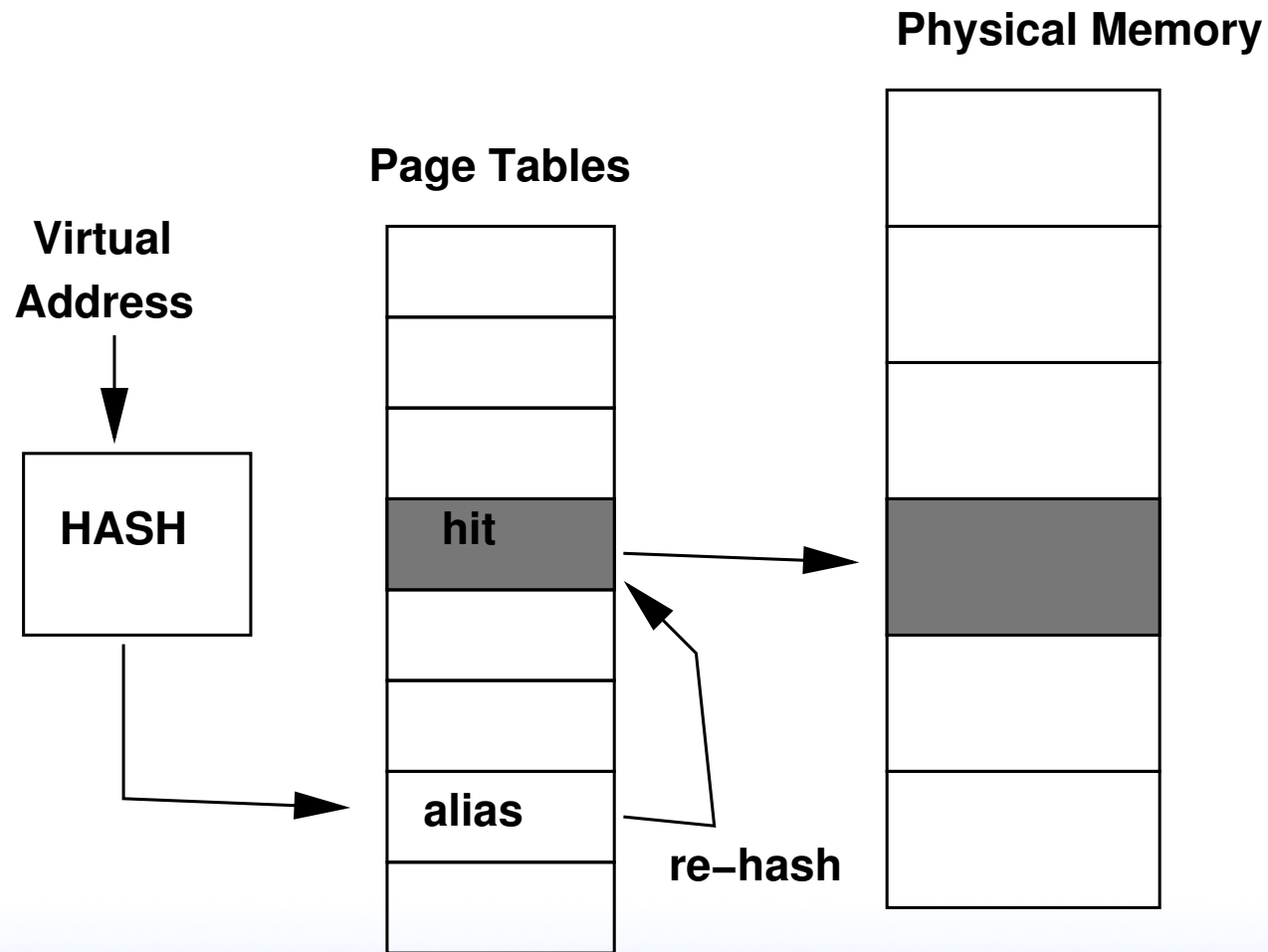
- IBM Power, Ultrasparc, ia64
- 4/5 level tables can be slow
- Have one single mapping, page mapping for each physical to virtual page
- Almost like having a large software TLB
- Note: Linus Torvalds wasn't a fan



- 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 to hash



Inverted Page Table Diagram



Walking the Page Table

- Can be walked in Hardware or Software
- Hardware is more common
 - Generally have a register pointing to the main page table (CR3 on x86?)
- 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



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 / OS

- 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 pages pre-zeroed by OS
 - heap/stack might be auto-allocated zeroed pages)
- If address in a region unknown by OS, segfault



major/minor page faults

- “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.



Bringing a Page into Physical Memory

- Need 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)

