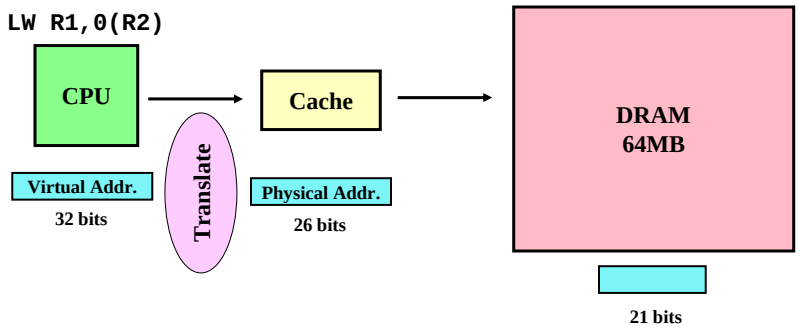
Lecture 21: Virtual Memory II

- Last Lecture:
  - Introduction to virtual memory
- Today
  - Review and continue virtual memory discussion
- Next time
  - Guest lecture, Ron Kalla, IBM

Goals of virtual memory

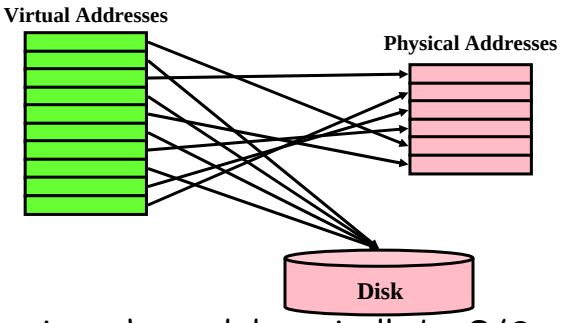
- Make it appear as if each process has:
  - Its own private memory
  - Nearly infinite-sized memory

A Load to Virtual Memory



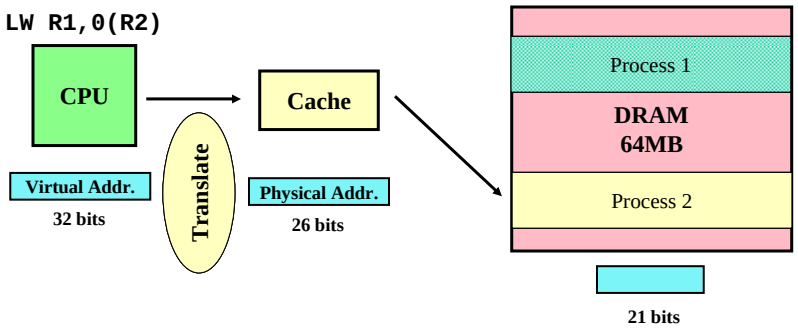
- Translate from virtual space to physical space
  - VA ⇒ PA
  - May need to go to disk

Virtual Addresses Span Memory+Disk



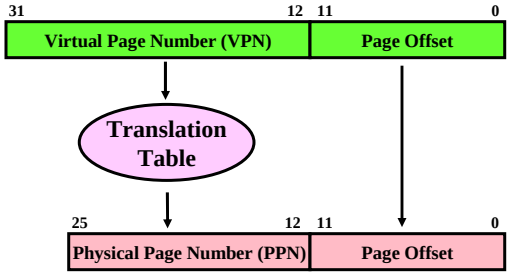
- Mappings changed dynamically by O/S
  - In response to users data accesses
  - OS triggered by hardware

A Load to Virtual Memory



- Both programs can use the same set of addresses!
  - Change translation tables to point same VA to different PA for different programs

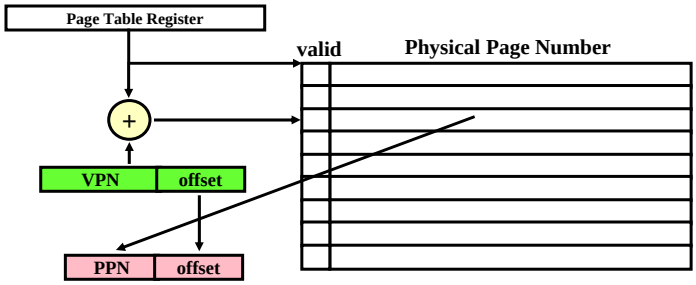
Virtual Address Translation



- Main Memory = 64MB
- Page Size = 4KB
- VPN = 20 bits
- PPN = 14 bits

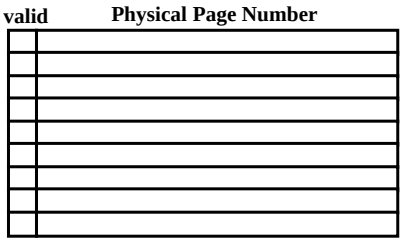
- Translation table
  - aka "Page Table"
  - An "array of pointers"

Page Table Construction



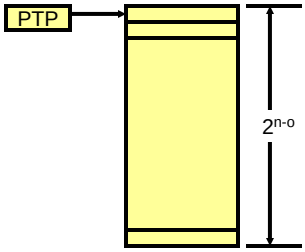
- Page table size
  - (14 + 1) * 2^{20} = 4MB
- Where to put the page table?

What else can we put in this table?



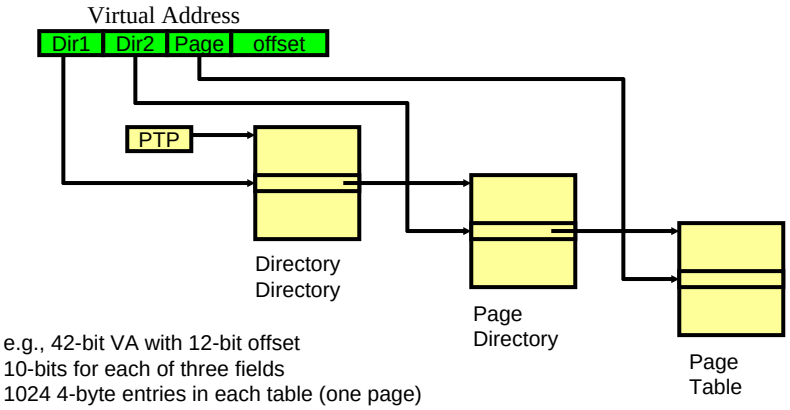
- What if we want a page to be read-only?
- What if we want a page to only hold instructions?
- What if we want to keep track of "dirty" pages?

Page Table Organization



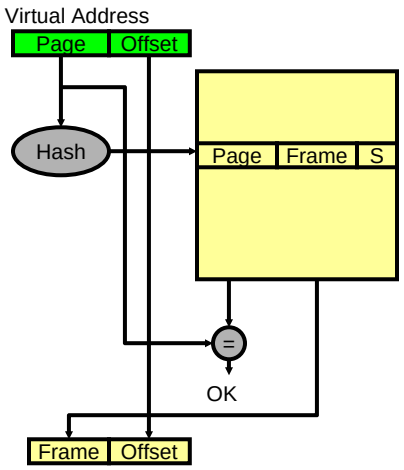
- Flat page table has size proportional to size of *virtual* address space
  - can be very large for a machine with 64-bit addresses and several processes
- Three solutions
  - page the page table (fixed mapping)
    - what really needs to be *locked down?*
  - multi-level page table (lower levels paged - Tree)
  - inverted page table (hash table)

Multi-Level Page Table



e.g., 42-bit VA with 12-bit offset
10-bits for each of three fields
1024 4-byte entries in each table (one page)

Inverted Page Tables

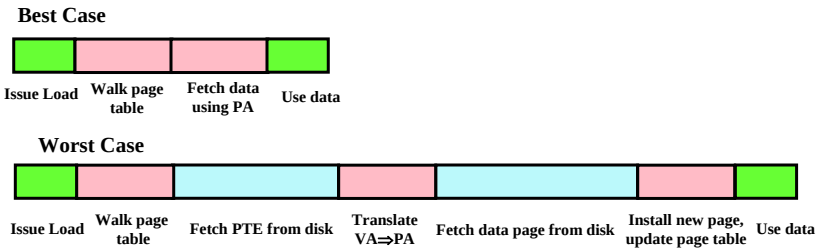


- Only store table entries for pages in physical memory
- Miss in page table implies page is on disk
- Usual hash-table rules apply:
  - Hash table should not be full
  - Rule of thumb: at least twice as many hash table entries as there are pages in memory.

Summary so far

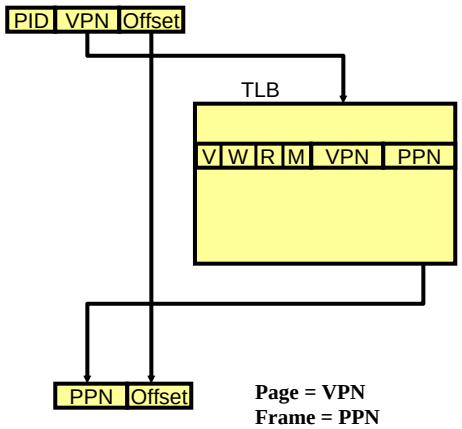
- Virtual memory provides
  - Illusion of private memory system for each process
  - Protection
  - Relocation in memory system
  - Demand paging
- But – page tables can be large
  - Motivates: paging page tables, multi-level tables, inverted page tables
- Next:
  - How can we improve performance of page tables?

How Long does it Take to Access VM?



- Problems
  - Multiple memory and potentially disk accesses
  - Can we use cache for the page-table access? How?

Translation Lookaside Buffers



- Store most frequently used translations in small, fast memory (cache for page table entries)
- Valid, Writeable, Referenced, Modified
  - Access protection
  - Replacement strategies
- Size – often 128 entries
- Highly associative (sometimes fully assoc.)

"Rare" Behavior in VM system

- TLB Miss
  - Translation is not in TLB – but everything could be in memory
  - Two approaches
    - Hardware state machine *walks* the page table
      - fast but inflexible
    - Exception raised and software walks the page table
- Page Fault
  - Entry not in TLB and target page not in main memory
- Worst case
  - Page fault and page table and target page

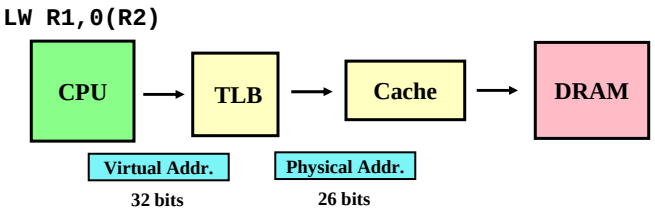
Reducing TLB misses

- Same type of optimizations as for cache
  - Associativity (many TLBs are fully associative)
  - Capacity – TLBs tend to be 32-128 entries
- Adjust page size
  - Small pages
    - Reduces internal fragmentation
    - Speeds page movement to/from disk
  - Large pages
    - Can cover more physical memory with same number of TLB entries
  - Solution – typically have a variable page size
    - Select by OS, 4KB-256KB (superpages)

Virtual Memory + Caching

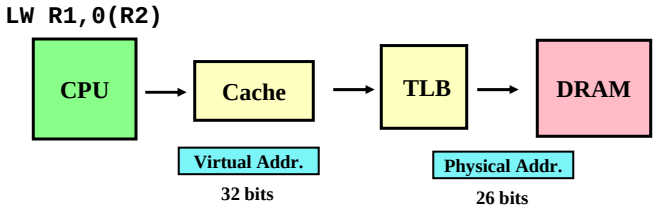
- Conflicting demands:
  - Convenience of flexible memory management (translation)
  - Performance of memory hierarchy (caching)
- Requires cooperation of O/S
  - Data in cache implies that data is in main memory
- Combine VM and Caching
  - Where do we put the Cache and the TLB????

Physically Addressed Cache



- Translate first from VA ⇒ PA
- Access cache with PA
- Problems?

Virtually Addressed Cache



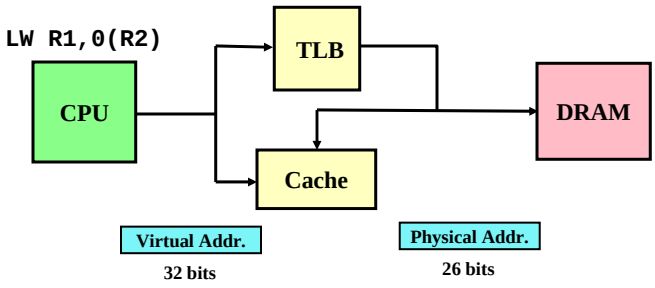
- Access cache first
- Only translate if going to main memory

- Problems?

Virtually addressed caches give aliasing problems

- Can occur when switching among multiple address spaces
- Synonym aliasing
  - Different VAs point to the same PA
  - Occurs when data shared among multiple address spaces
  - One solution – always translate before going to the cache
- Homonym aliasing
  - Same VA point to different PAs
  - Occurs on context switching
  - Two solutions:
    - Flush TLB on process switch/system call
    - TLB includes process ID

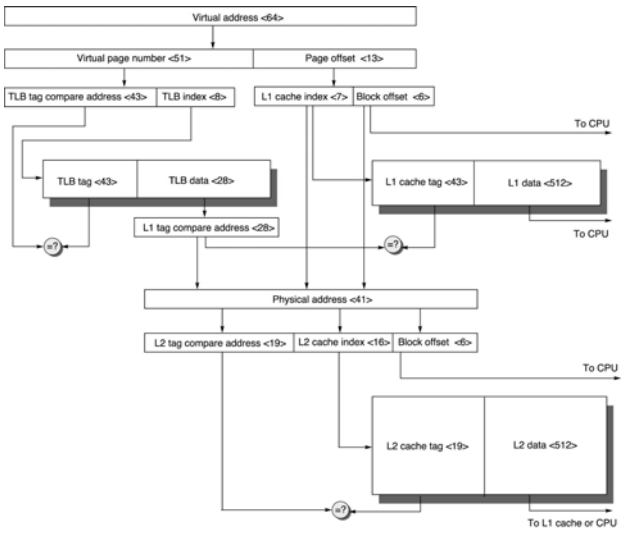
Best of Both Worlds:



Virtually addressed, Physically Tagged

- **Parallel Access**
- **Eliminate synonym problem**

Virtual Index, Physical Tag



Other Aliasing Solutions

- Note virtually indexed/physically tagged put constraints on cache capacity, page size, etc.
- Other solutions:
  - 21264: 8KB pages, 64KB i-cache, 2-way set associative
    - Aliases could reside in 8 different places in cache
    - On cache miss, invalidate any possible aliases in cache
  - Intel Pentium 4
    - Virtually indexed/virtually tagged cache
    - Check for TLB misses off line (roll back if necessary)

Virtual Memory Summary

- Relocation, Protection
- Apparent memory size >> DRAM capacity

- Translation
  - From large VA space to smaller PA space
  - Page tables hold translations

- Provides
  - Separation of memory management from user programs
  - Ability to use DRAM as cache for disk
  - Fast translation using Translation Lookaside Buffer (TLB)
    - Cache for page table
  - Speed - translate in parallel with cache lookup