CS184b: Computer Architecture [Single Threaded Architecture: abstractions, quantification, and optimizations]

Day14: February 22, 2000

Virtual Memory

Today

- Problems
  - memory size
  - multitasking
- Different from caching?
- TLB
- co-existing with caching

Problem 1:

- Real memory is finite
- Problems we want to run are bigger than the real memory we may be able to afford…
  - larger set of instructions / potential operations
  - larger set of data
- Given a solution that runs on a big machine
  - would like to have it run on smaller machines, too
    - but maybe slower / less efficiently

Opportunity 1:

- Instructions touched < Total Instructions
- Data touched
  - not uniformly accessed
  - working set < total data
  - locality
    - temporal
    - spatial

Problem 2:

- Convenient to run more than one program at a time on a computer
- Convenient/Necessary to isolate programs from each other
  - shouldn't have to worry about another program writing over your data
  - shouldn't have to know about what other programs might be running
  - don't want other programs to be able to see your data

Problem 2:

- If share same address space
  - where program is loaded (puts its data) depends on other programs (running? Loaded?) on the system
- Want abstraction
  - every program sees same machine abstraction independent of other running programs

One Solution

- Support large address space
- Use cheaper/larger media to hold complete data
- Manage physical memory “like a cache”
- Translate large address space to smaller physical memory
- Once do translation
  - translate multiple address spaces onto real memory
  - use translation to define/limit what can touch

Conventionally

- Use magnetic disk for secondary storage
- Access time in ms
  - e.g. 9ms
  - 9 million cycles latency
- bandwidth ~100Mb/s
  - vs. read 64b data item at GHz clock rate
    - 64Gb/s

Like Caching?

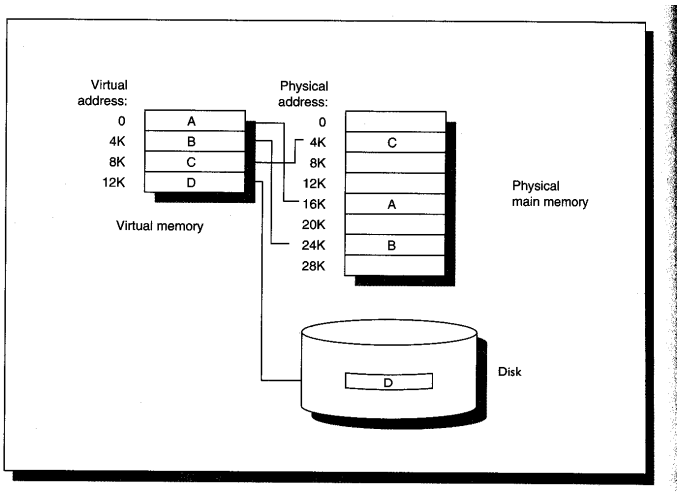
- Cache tags on all of Main memory?

- Disk Access Time >> Main Memory time
- Disk/DRAM >> DRAM/L1 cache
  - bigger penalty for being wrong
    - conflict, compulsory
- …also historical
  - solution developed before widespread caching...

Mapping

- Basic idea
  - map data in large blocks (pages)
  - use memory table
  - to record physical memory location for each, mapped memory block

Address Mapping



[Hennessy and Patterson 5.36]

Mapping

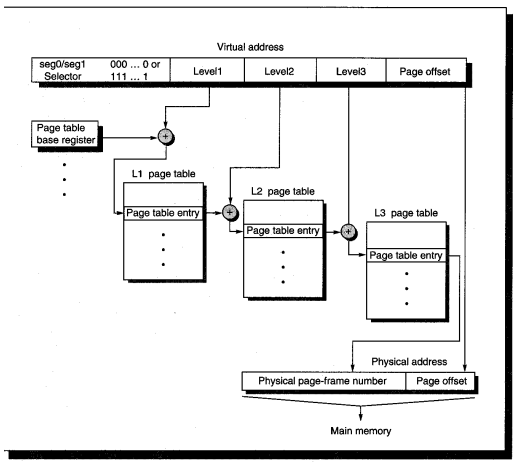
- 32b address space
- 4Kb pages
- $2^{32}/2^{12}=2^{20}$=1M address mappings

- Very large translation table

Translation Table

- Traditional solution
  - from when 1M words >= real memory
  - break down page table hierarchically
  - divide 1M entries into 4*1M/4K=1K pages
  - use another translation table to give location of those 1K pages
  - …multi-level page table

Page Mapping



[Hennessy and Patterson 5.43]

Page Mapping Semantics

- Program wants value contained at A
- pte1=top_pte[A[32:24]]
- if pte1.present
  - ploc=pte1[A[23:12]]
  - if ploc.present
    - Aphys=ploc<<12 + (A [11:0])
    - Give program value at Aphys
  - else … load page
- else … load pte

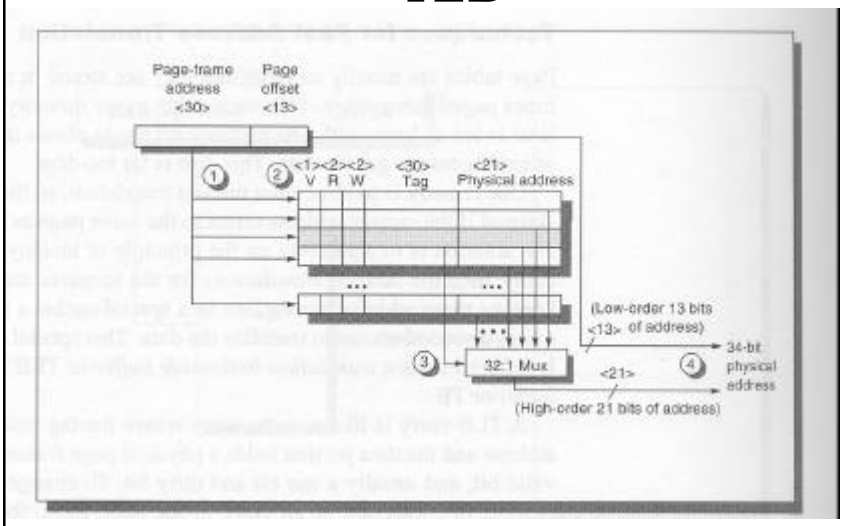
Early VM Machine

- Did something close to this...

Modern Machines

- Keep hierarchical page table
- Optimize with lightweight hardware assist
- Translation Lookaside Buffer (TLB)
  - Small associative memory
  - maps physical address to virtual
  - in series/parallel with every access
  - faults to software on miss
  - software uses page tables to service fault

TLB



[Hennessy and Patterson 5.43]

VM Page Replacement

- Like cache capacity problem
- Much more expensive to evict wrong thing
- Tend to use LRU replacement
  - touched bit on pages (cheap in TLB)
  - periodically (TLB miss? Timer interrupt) use to update touched epoch
- Writeback (not write through)
- Dirty bit on pages, so don't have to write back unchanged page (also in TLB)

VM (block) Page Size

- Larger than cache blocks
  - reduce compulsory misses
  - full mapping
    - not increase conflict misses
    - could increase capacity misses
  - reduce size of page tables, TLB required to maintain working set

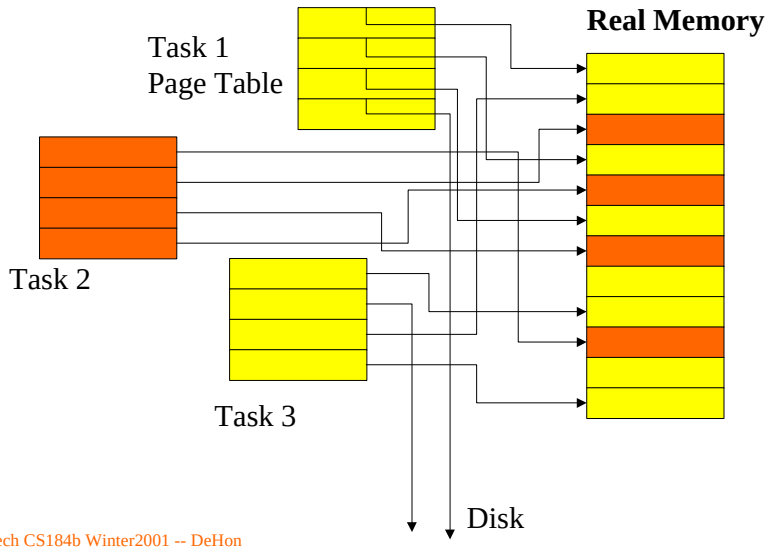
VM Page Size

- Modern idea: allow variety of page sizes
  - "super" pages
  - save space in TLBs where large pages viable
    - instruction pages
  - decrease compulsory misses where large amount of data located together
  - decrease fragmentation and capacity costs when not have locality

VM for Multitasking

- Once we're translating addresses
  - easy step to have more than one page table
  - separate page table (address space) for each process
  - code/data can be live anywhere in real memory and have consistent virtual memory address
  - multiple live tasks may map data to to same VM address and not conflict
    - independent mappings

Multitasking Page Tables



VM Protection/Isolation

- If a process cannot map an address
  - real memory
  - memory stored on disk
- and a process cannot change it page-table
  - and cannot bypass memory system to access physical memory...
- the process has no way of getting access to a memory location

Elements of Protection

- Processor runs in (at least) two modes of operation
  - user
  - privileged / kernel
- Bit in processor status indicates mode
- Certain operations only available in privileged mode
  - e.g. updating TLB, PTEs, accessing certain devices

System Services

- Provided by privileged software
  - e.g. page fault handler, TLB miss handler, memory allocation, io, program loading
- System calls/traps from user mode to privileged mode
  - …already seen trap handling requirements...
- Attempts to use privileged instructions (operations) in user mode generate faults

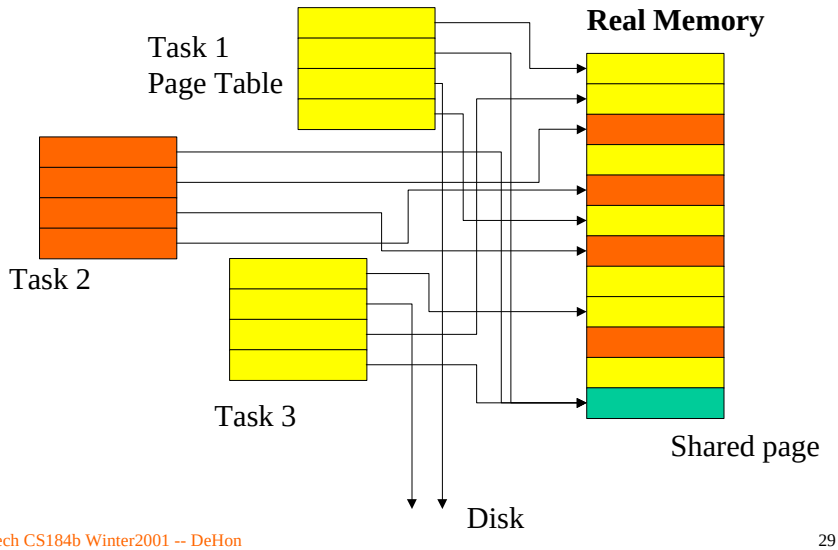
System Services

- Allows us to contain behavior of program
  - limit what it can do
  - isolate tasks from each other
- Provide more powerful operations in a carefully controlled way
  - including operations for bootstrapping, shared resource usage

Also allow controlled sharing

- When want to share between applications
  - read only shared code
    - e.g. executables, common libraries
  - shared memory regions
    - when programs want to communicate
    - (do know about each other)

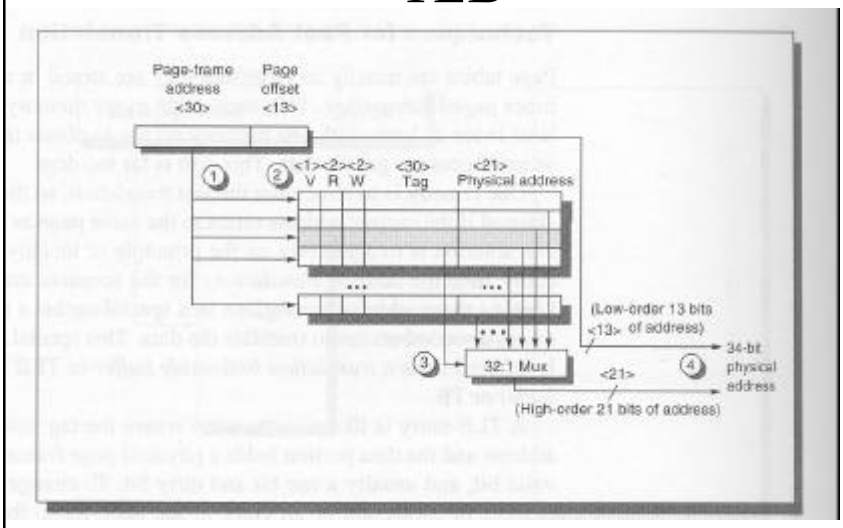
Multitasking Page Tables



Page Permissions

- Also track permission to a page in PTE and TLB
  - read
  - write
    - support read-only pages
    - pages read by some tasks, written by one

TLB



[Hennessy and Patterson 5.43]

Page Mapping Semantics

- Program wants value contained at A
- pte1=top_pte[A[32:24]]
- if pte1.present
  - ploc=pte1[A[23:12]]
  - if ploc.present and ploc.read
    - Aphys=ploc<<12 + (A [11:0])
    - Give program value at Aphys
  - else … load page
- else … load pte

VM and Caching?

- Should cache be virtually or physically tagged?
  - Tasks speaks virtual addresses
  - virtual addresses only meaningful to a single process

Virtually Mapped Cache

- L1 cache access directly uses address
  - don't add latency translating before check hit
- Must flush cache between processes?

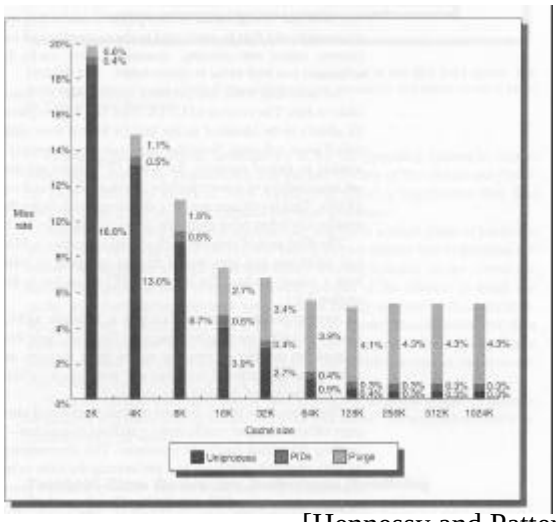
Physically Mapped Cache

- Must translate address before can check tags
  - TLB translation can occur in parallel with cache read
    - (if direct mapped part is within page offset)
  - contender for critical path?
- No need to flush between tasks
- Shared code/data not require flush/reload between tasks
- Caches big enough, keep state in cache between tasks

Virtually Mapped

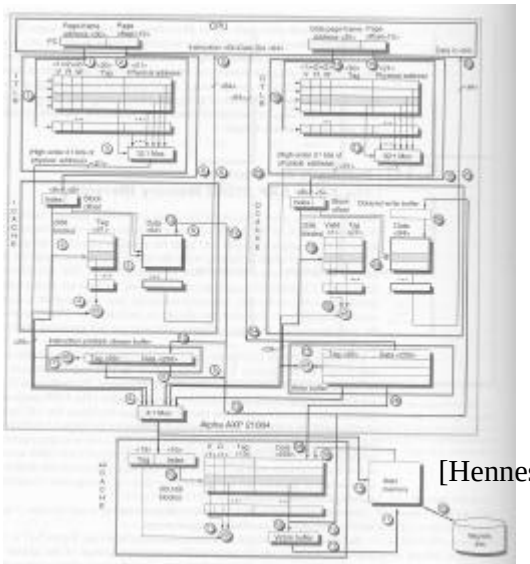
- Mitigate against flushing
  - also tagging with process id
  - processor (system?) must keep track of process id requesting memory access
- Still not able to share data if mapped differently
  - may result in aliasing problems
    - (same physical address, different virtual addresses in different processes)

Virtually Addressed Caches



[Hennessy and Patterson 5.26]

Processor Memory Systems



[Hennessy and Patterson 5.47]

Administrative

- No class next Thursday (3/1)

Big Ideas

- Virtualization
  - share scarce resource among many consumers
  - provide “abstraction” that own resource
    - not sharing
  - make small resource look like bigger resource
    - as long as backed by (cheaper) memory to manage state and abstraction
- Common Case
- Add a level of Translation