ECE437: Introduction to Digital Computer Design

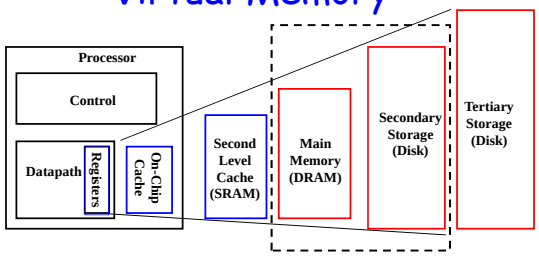
Chapter 5c (virtual memory)

Fall 2016

Virtual memory : The real reason

- Application's view of memory
  - Large: ~4GB in Pentium (32b address)
  - Exclusive: Only program in memory
- System view
  - Smaller: 512MB-2GB
  - Multiple programs share memory
  - Run in a protected manner (memory is private **)
  - Address range is fixed (starts at 0x0)
  - Do not bring in entire program
    - Bring in relevant parts as needed
- VM reconciles these conflicting "views"
  - Not just the classical benefits of hierarchy
    - Illusion of
      - speed of expensive level
      - capacity and cost of cheaper level
  - Sharing, Protection, memory mapping

Virtual Memory



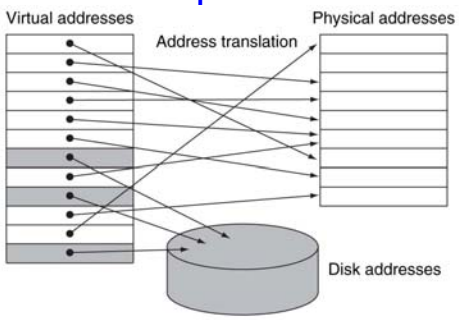
- Data movement between Disk and Main memory
- We know how layers of hierarchy interact
  - Cache and main memory
- Can we apply all the same techniques?
  - Similarities and differences

Virtual Memory

- Similarities:
  - Many-to-few mapping
    - larger address space to smaller memory
  - Used for data movement between layers of the memory hierarchy
- Differences:
  - Miss-penalty : 10-100 cycles vs. ~1 million cycle
  - Block size : 16-64 bytes vs. 4 KB – 8 KB
  - Full associativity (But no associative search!)
  - Software handling so more sophisticated schemes than hardware

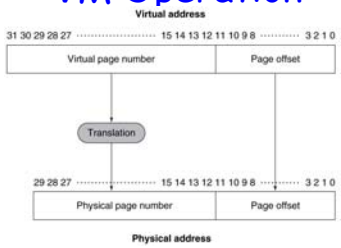
Fundamental

VM Operation



- Programs use virtual address
- The data/code resides elsewhere (physical address)

VM Operation



- Assume 4KB pages
- 32-bit VA and 30-bit PA
- System responsible for translation
- E.g.
  - lw $r2, 0xffffc004
  - 0xffffc -> 0x20000 # VPN-> PPN translation
  - PA = 0x20000004

VM Terminology

- Blocksize ~ 4KB - 8KB+
  - Block called as Page
- Miss : Page-fault
  - Fetch from disk
- Derivative properties:
  - Fundamental constraint: high access latency

Back to the 4 questions

- Recall the 4 questions in caches
- Q1. Where does a block go?
  - Fully associative
    - Block offset and tag
    - NO INDEX
  - Why?
    - AMAT = hit time + miss-rate*miss-penalty
    - Miss-penalty = ~1 million cycles
    - Have to minimize miss-rate
    - full- associativity minimizes miss rate

Q2. Block Identification

- Q2. Block identification
  - Fully associative search??
    - 30-bit physical address (1GB)
    - 4 KB pages
    - Number of frames = $2^{30}/2^{12} = 2^{18} = 256K$
    - Compare 256 K frames in parallel? Whoa!
  - Reframe question:
    - Cache (previously): Is this VA in any given frame? => Parallel search
    - VM (now): Where is this VA? => Table lookup
      - Page table
      - Fully associativity via table lookup
        - » each table entry can point to ANY frame

Replacement

- Q3. Block replacement
  - LRU and/or LRU approximation (NRU with reference/use bits)
  - Sophisticated mechanisms possible (handling in OS software)
- Page-fault : Exception
  - Save instruction that causes fault
  - OS service the fault, i.e., brings in the relevant page from disk (VA-> disk address??)
  - OS knows service is slow; schedule other program
  - When disk access is complete
  - Restart at offending instruction

Write handling

- Q4. What happens on a write
  - Write-through or write-back?

Virtual Memory

- CPU and memory are shared by processes and VM ensures protection but the sharing is slightly different
- CPU is time shared – processes run for a while before being switched out by OS
  - OS saves their registers and PC in its data structures
- Memory is space shared – processes occupy pages and their page tables sit in memory
  - This occurs even when a process is switched out of the CPU

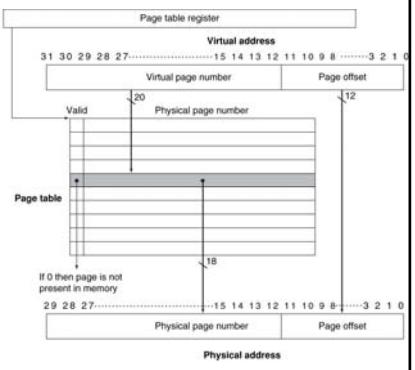
Exercise: Virtual Memory Design

- Page size = 8KB
- Virtual Memory
  - 64b → 2^{64} = 2^{34} GB!
- Physical memory = 8GB
- What is the size of the per-process page table?

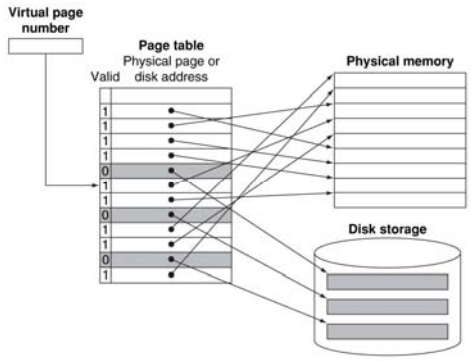
- What happens on a context switch?

Page Table

- Virtual page number
  - Tag in caches
- Physical address
  - Frame-number in caches
- PTBR (ONE per CPU)
  - Change value on context switch
  - Per-process page table



Page Table



Page Table Entries

- What does a page table entry contain
  - Physical page number (18 bits)
  - Access control (read/write permissions)
  - Valid bit (1 bit)
  - Misc
    - Use bit for replacement
    - Dirty bit for write-back
  - ~ 4 bytes (1 word)

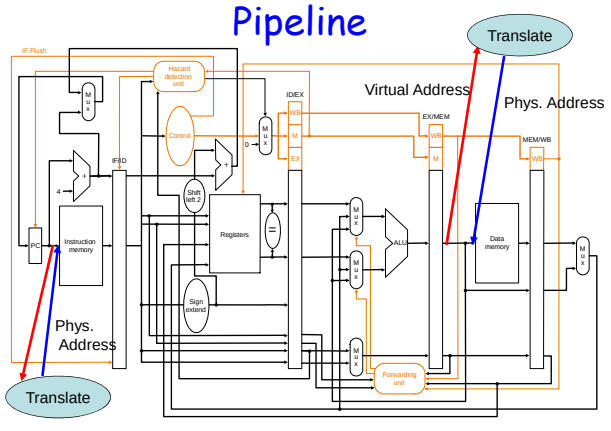
Size of Page Table

- What is the size of the page table for a system with
  - 32 bit addresses
  - 1 GB physical memory
  - 4KB pagesize
  - Virtual pages = $2^{32}/2^{12} = 2^{20}$
  - Physical pages = $2^{30}/2^{12} = 2^{18}$
  - PT size (per process) = (Entry size) * (# entries)
    = 4 bytes * 2^{20} = 2^{22} bytes = 4 MB
- # of processes? ~60 on my WinXP Pro machine
  - 240 MB for page tables?
  - "Big government" : 25% consumed for administration!!

Size of Page Table

- Techniques to reduce Page Table Size
  - Gradual growing –
    - Not all processes use all of their VA space
    - only small fraction of all virtual pages actually used , especially in 64-bit machines
    - so VA space is sparse - not contiguous but has holes in it
    - Previous slide calculation assumes flat table for page table
    - what data structure would you use for a sparse space?
  - Inverted PT : entries per physical page
    - Hash table with as many buckets as physical pages
- Know that these options exist – explored in later courses

Pipeline



Translation overhead

- Where does table reside?
  - Main memory
  - Whoa! Wait a minute – this is 100% overhead!
    - Each memory reference now generates two memory references
    - lw $r2,0xffff0004
      - Access page table entry for 0xffff0, get PPN
      - Access PPN:004
  - But we busted our heads to put caches and 100 other tricks to avoid main memory access and now VM adds one WHOLE extra main memory access?

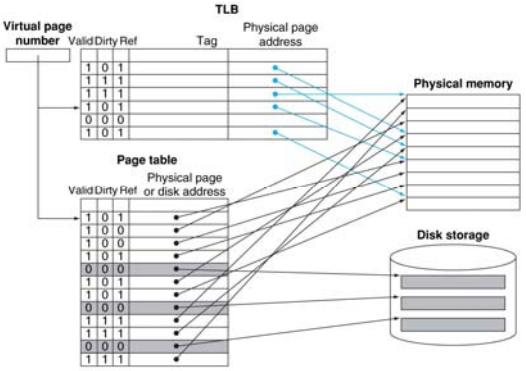
Translation overhead

- We want to minimize this overhead!
  - Page table entries can be cached like ordinary data
  - Why would it work?????
  - Tricky ***!!
  - Special cache for Page table entries

Faster Translation

- 100% overhead with VM system
- Eliminate memory access for translation
  - Caching
  - Translation lookaside buffer (TLB)
    - Also DTB in some literature
    - A cache of translations
  - 64-128 entries
    - Covers 256 KB ~512 KB
  - Organization
    - 64 entry fully associative
    - 256 entry, 16-way set associative
    - May be multi-level
- TLB is a cache for page table

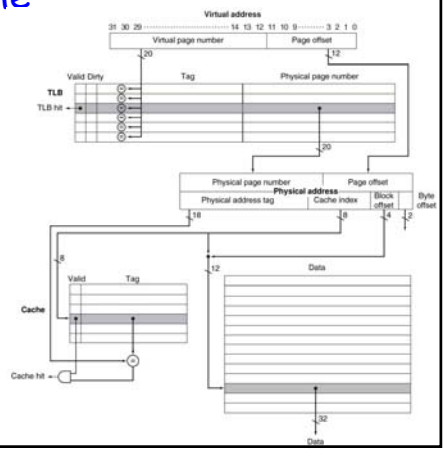
Translation Lookaside Buffer



•TLB is a cache for page table

TLB+Cache

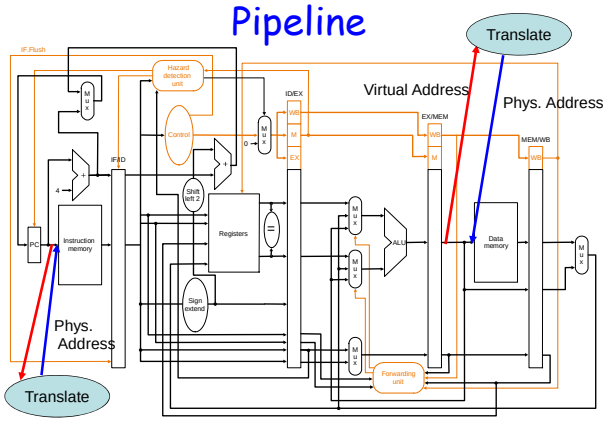
- Cache operation
  - With physical addresses
  - Translation on critical path



TLB design

- TLB's input address is virtual page number
- TLB is a cache for page table entries (PTEs) which already exploits spatial locality thru pages so TLB's block size = 1 PTE → TLB block offset is 0 bits
- TLB could be set assoc. or fully assoc.
  - VM is fully assoc. (VM miss penalty is huge) but TLB miss penalty is memory access (not huge) so TLB need not be fully assoc.
- So depending on #TLB entries and associativity, compute index & tag

Pipeline



Single TLB is a structural hazard too !

TLB access latency

- TLB reduces access latency FROM cache (1-2 cycles) + memory for page table (300 cycles) TO cache (1-2 cycles) + TLB (~1-2 cycles) but still 100% overhead over cache
- One suggestion: use VA for L1, translate on L1 miss to use PA for L2
- Makes sense BUT has 2 problems due to modern OS

TLB access latency

- Problem1: two processes' VAs look alike – so context switch from process1 to process2 and process2 will get cache hits on process1's data!
  - Flush cache at context switch – many cache misses
  - Any other solution?
  - Even if we solve this problem2 is harder

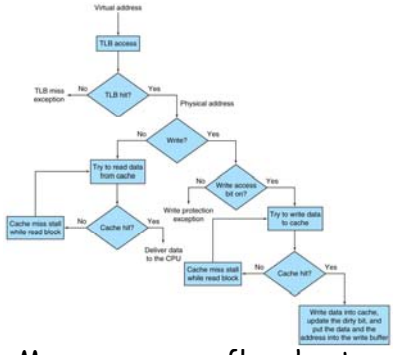
TLB access latency

- Problem2: Modern OS allow a strange thing: the SAME data may use DIFFERENT virtual addresses in different parts of SAME process but accesses to these VAs must access the same data
  - Such VAs are called aliases/synonyms
  - Provide key flexibility for applications
  - Key for DLLs, memory-map I/O,

TLB access latency

- Using VA into L1 in the presence of synonyms cause this problem
  - Assume two aliases VA1 and VA2 where a store to VA2 and load from VA1, then VA1 and VA2 will go to different blocks in the cache so the load will not load the stored value!
- Real systems solve this problem so that TLB and L1 are accessed in parallel (come to 565)

Putting it all together



- Memory access flowchart

See Fig 5.25 for full chart

Full Hierarchy design

- 32b VA, 30 bit PA (i.e. 1GB Phys. Mem)
  - 4 KB pages
  - TLB
    - 256 entry, 4-way associative
  - Cache
    - 32KB, direct mapped, 32B blocks

Summary

- 4Q on VM
  - Placement : fully associative
  - Identification : Page Table lookup
  - Replacement : LRU / LRU-approx
  - Writes : Writeback
- 4Q on TLB
  - Placement: small and fully associative, larger and set-associative
  - Identification: Associative search (CAM)
  - Replacement : NRU, random**
  - Writes : ?? Writes to TLB??

VM Miscellanea

- On a context switch:
  - Change contents of PTBR for appropriate page table
  - What do we do with TLB contents?
    - What's the problem? All VAs look alike!
      - How will TLB distinguish between VAs of previous and new process
    - Flush all entries
      - Simple, but inefficient
    - Associate Process ID with address
      - Flush required only when process IDs are reused

VM Miscellanea

- Memory efficiency of page tables
  - Limit register
    - Only region between PTBR and Limit is valid
    - Grow as needed
  - Segmented
    - Two page tables and two limit registers (Stack and Heap)
  - Inverted Page table
    - Hashing to map VA to a number within PA range
      - Hash 32-bit VA to 28 bit PA
      - Lookup complications
      - Collision
  - Multilevel page tables
  - Paging page tables

Real stuff

Characteristic	ARM Cortex-A8	Intel Core i7
Virtual address	32 bits	48 bits
Physical address	32 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 16 MiB	Variable: 4 KiB, 2/4 MiB
TLB organization	1 TLB for instructions and 1 TLB for data Both TLBs are fully associative, with 32 entries, round robin replacement TLB misses handled in hardware	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, 7 per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB is four-way set associative, LRU replacement The L2 TLB has 512 entries TLB misses handled in hardware

- Ch 5 done!