
EEC 485 High Performance Computer Architecture (Fall 2007)

Chapter 7.4 Virtual Memory

Chansu Yu

Cleveland State University

1

Summary: Cache

- **Cache basics**
 - **Memory-processor speed gap** motivates “cache”
 - **A subset of memory contents that are frequently referenced is stored in cache in “blocks”**
 - **Why does it work?**
 - If size ratio is 5%, hit ratio is 5% in random order of references
 - In fact, it is about 95% due to “locality”
- **Three cache policies**
 - **Where to put a memory block in cache: “Placement policy”**
 - Direct-mapped: only one position
 - N-way set associative: N positions
 - Fully associative: any position
 - **Which block to kill: “Replacement policy”**
 - Direct: not necessary
 - N-way/Fully associate: “LRU (least recently used)” block among the set
 - **“Write policy”: Write back or write through**
- **Performance improvement**
 - **Miss ratio: Fully associative, 2-level cache, Big cache size, Big block size**
 - **Hit time: Direct, Small cache size**
 - **Miss penalty: Small block size**

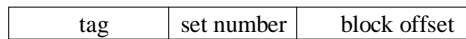
2

Summary: Cache

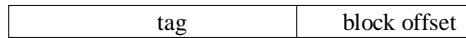
- Memory address decomposition



Direct-mapped



N-way set associative



Fully associative

Cache	Tag	Valid
	0	1
	1	0
	3	0
	2	1
	1	1
	2	0
	0	1
	3	1
	2	1
	1	0

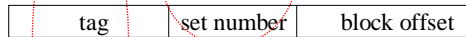
3

Summary: Cache

- Memory address decomposition



Direct-mapped



N-way set associative



Fully associative
(need to match with all tags
and many comparators!)

Cache	Tag	Valid
	0	1
	1	0
	3	0
	2	1
	1	1
	2	0
	0	1
	3	1
	2	1
	1	0

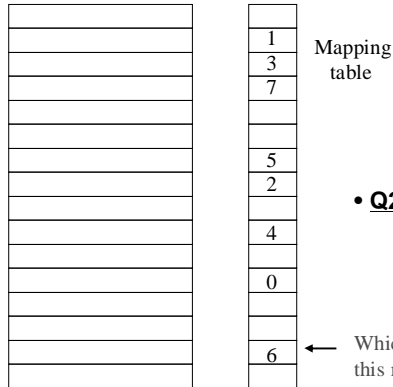
4

Summary: Cache

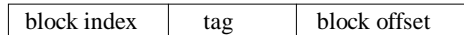
- 2 Questions



- **Q1: How about a mapping table from the memory side (one entry per memory block)?**



- **Q2: How about this address decomposition?**



Cache		Tag	Valid
		0	1
		1	0
		3	0
		2	1
		1	1
		2	0
		0	1
		3	1
		2	1
		1	0

Which memory block is this mapped from?

Which cache block is this mapped to?

5

Contents

- Virtual memory
- Page table and TLB
- Interaction of virtual memory with cache

6

Virtual Memory: Motivation

- Historically, motivations for VM are
 - Allow efficient and safe sharing of memory among multiple programs
 - Remove the programming burdens of a small, limited amount of main memory
 - Programmers make overlays and load/unload manually
 - Not that important now
- A number of programs are running concurrently
 - The total memory required to run all the programs exceeds the main memory available
 - But since only a small fraction of this memory is actively being used at any point of time, we can “cache” them in main memory and put others in disk
 - I.e., use disk to simulate more memory!
- Multiple users on same computer
 - Want object code to have fixed addresses.
 - How to relocate multiple users' programs into same memory?
 - How to protect one user program from the other?

 - Can automatically relocate code at run-time – separate address space per program!

7

Virtual Memory: Motivation

Most common solution is virtual memory.

Can address more virtual memory than physical memory.

So, think of disk as the "usual" place to store VM.

Then, physical memory is mainly a cache for the VM on the disk.

So, VM very similar to caches.

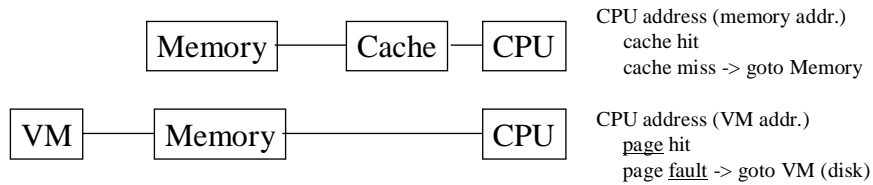
Different motivation.

Different evolution → different terminology.

VM page = cache block.
VM page fault = cache miss.

8

Virtual Memory: Motivation



9

Virtual Memory \approx Cache

Willing to do a lot to minimize access to the VERY SLOW disk.
Disk access: \approx 100K-1M cycles!

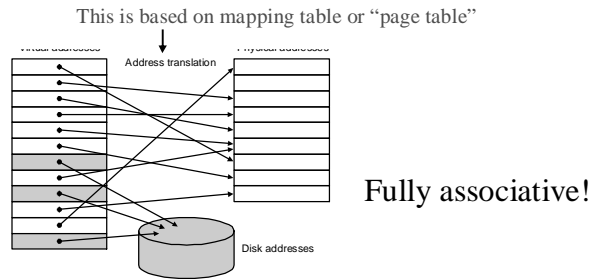
Choose caching strategies which minimize disk access, & thus misses:

- Fully associative
- Write-back
Reduce disk access on writes by grouping them.
- Approximate LRU
Exact LRU too expensive.
- Fairly large pages (typically 4-64KB)
Amortize high access time. Not too large to lose good spatial locality.

10

Virtual Memory: Mapping (= Block Placement)

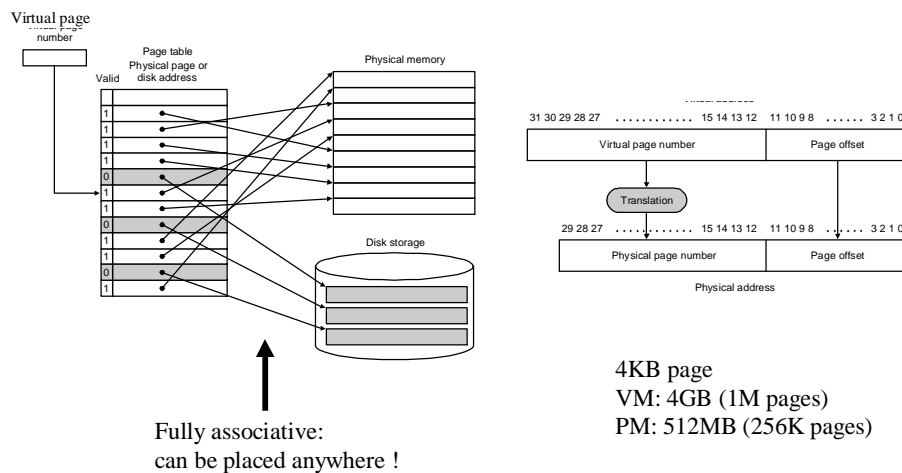
- Main memory can act as a cache for the secondary storage (disk)



- Why fully associate?
 - Huge miss penalty (100k-1M cycles) : the data is not in memory, retrieve it from disk; it is called Page faults
 - Pages should be fairly large (e.g., 4KB)
 - LRU is worth the price
 - Handle the faults in software instead of hardware
 - Writeback

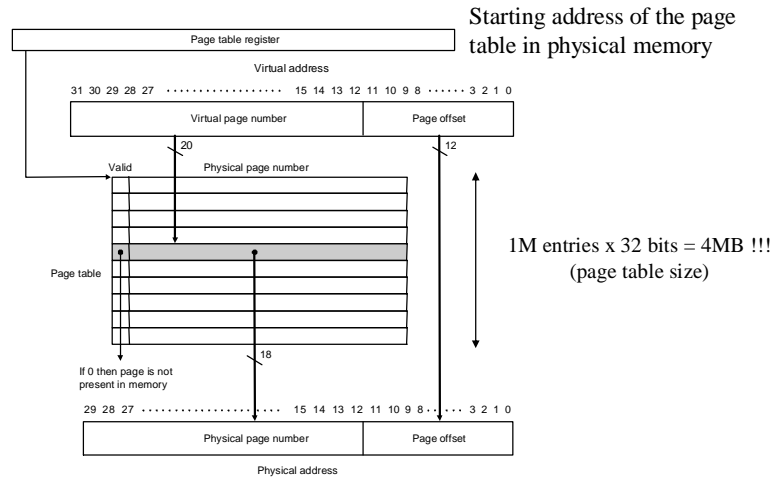
11

Page Tables for Address Translation (VA to PA)



12

Page Tables for Address Translation (VA to PA)



13

A Performance Problem

Every VM access uses multiple memory accesses:

1 for page table (or more if multi-level).

1 for data.

What's our general method for improving memory access performance?

?

?

Caches. Let's add a cache for the page table!

14

Translation Look-aside Buffer (TLB)

TLB = Hardware cache for page table access.

Only one TLB, not one per process.

Either caches for all processes or only for current process.

Will see issues...

Only cache page table entries which map to physical memory.

Speed up the common case.

Disk access is VERY SLOW – an additional page table access is negligible.

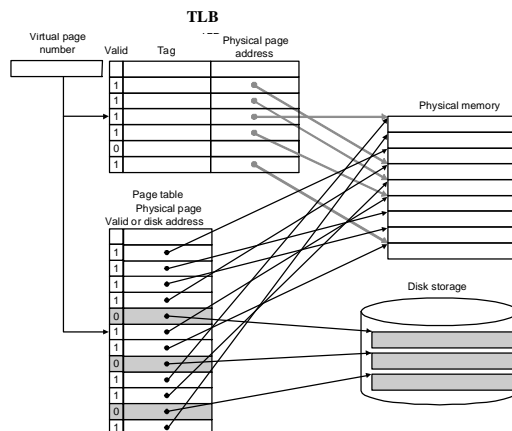
Low miss rate because of locality & large page size.

Usually write-back to minimize memory access.

15

Making Address Translation Fast

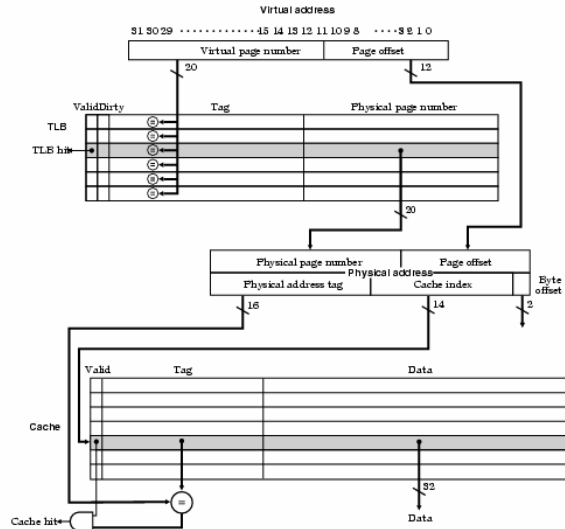
- A cache for address translations: translation lookaside buffer



TLB size = 32~4096 entries
Tag size = 20-bit (V.PN) if
fully associative

16

Example: DECStation 3100 (MIPS R2000)



Page size =
 VM space =
 PM space =
 No. entries in TLB =
 TLB access method =

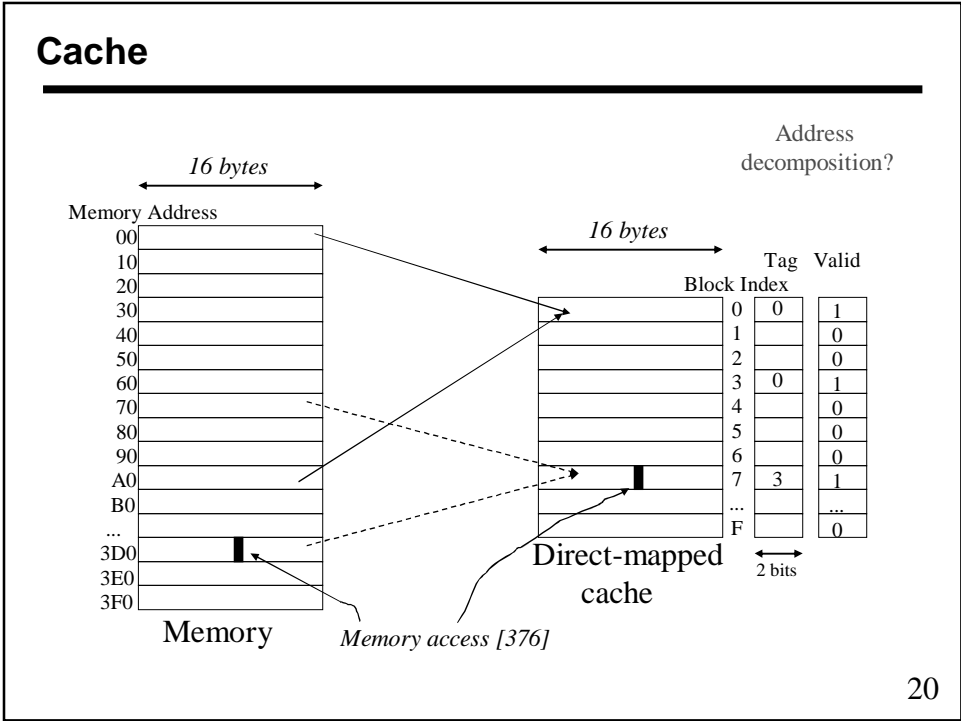
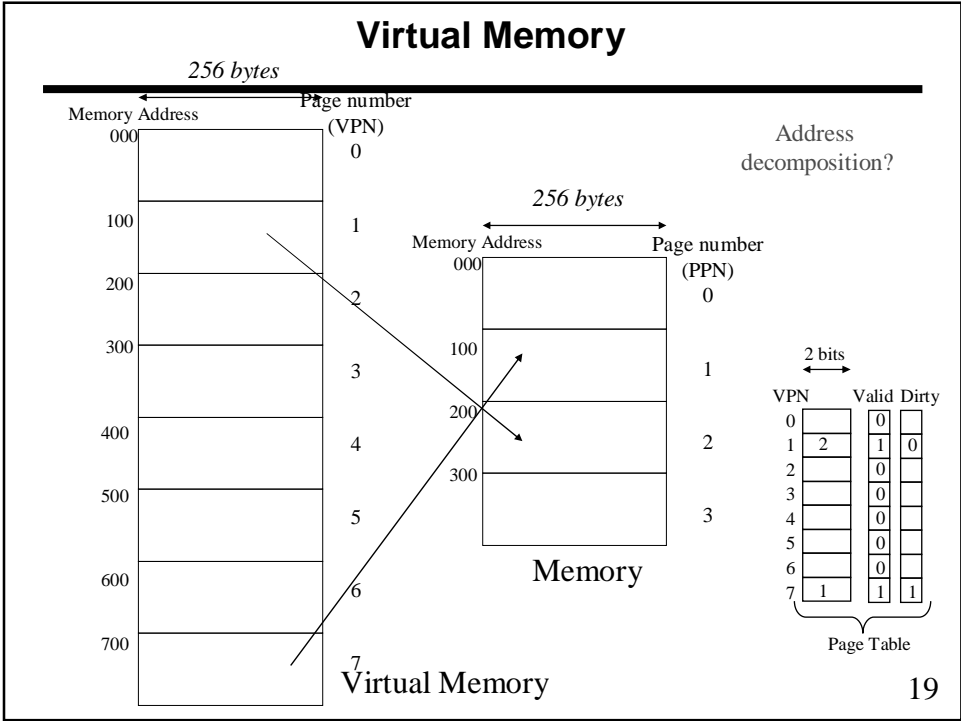
Block size =
 Tag size =
 No. entries in cache =
 Cache access method =

17

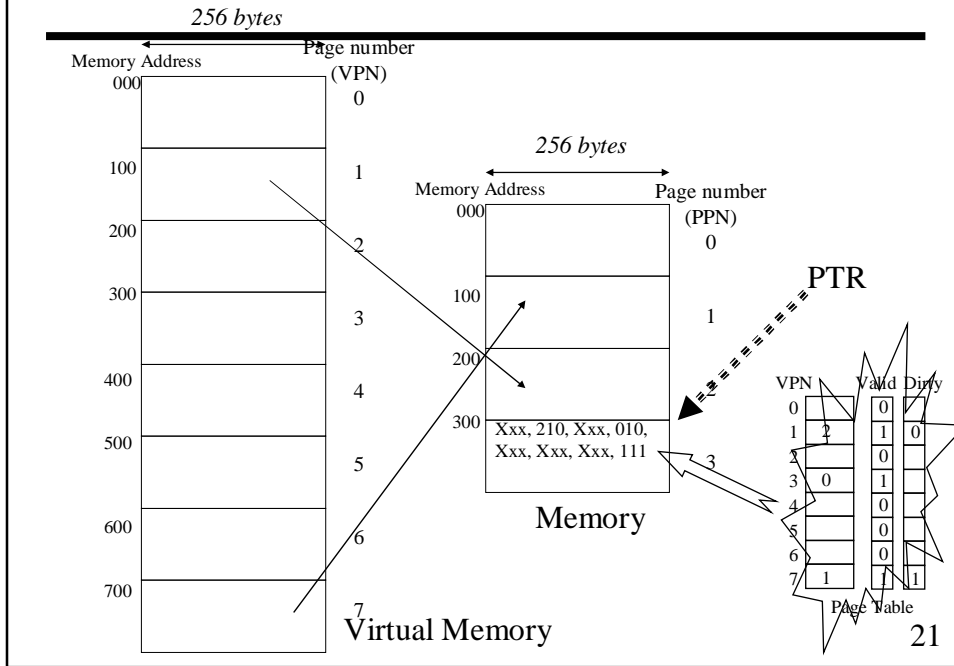
Example

- Virtual Memory
 - Page size is 256 bytes (2^8)
 - Virtual memory is 2048 bytes (2^{11} , 0~0x7ff) -> $2^3 = 8$ pages
 - (Physical) Memory has 1024 bytes (2^{10} , 0~0x3ff) -> $2^2 = 4$ pages
- TLB
 - TLB has 2 entries
- Cache
 - Block size is 16 bytes of data (2^4 , 0~0xf)
 - Memory has 1024 bytes (2^{10} , 0~0x3ff) -> $2^6 = 64$ blocks
 - Cache has 256 bytes (2^8 , 0~0xff) -> $2^4 = 16$ blocks
 - "Direct mapped"

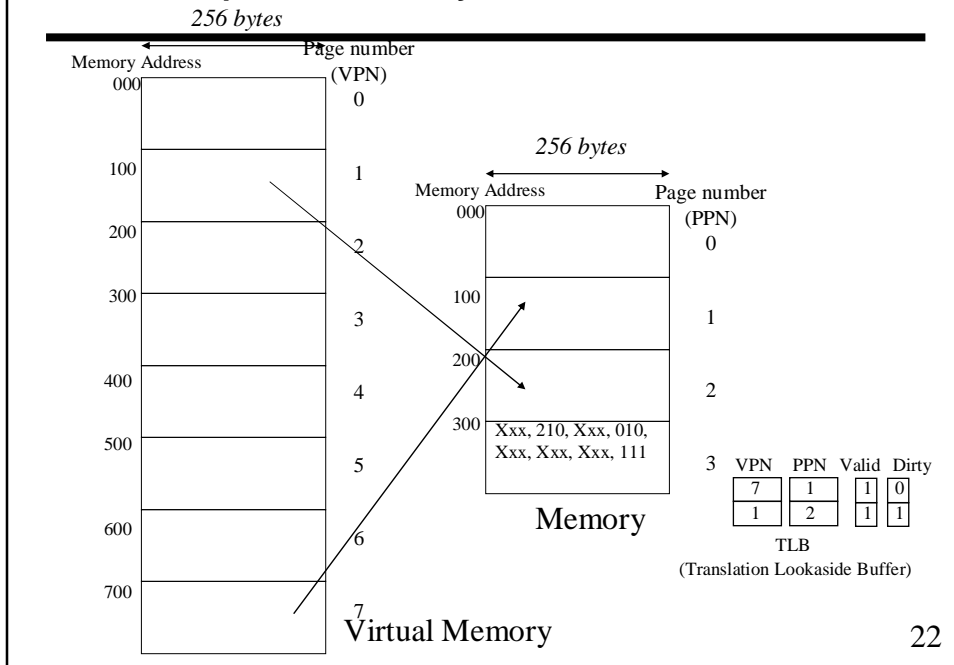
18



Where to put Page Table?



Expensive memory access to access PT => TLB



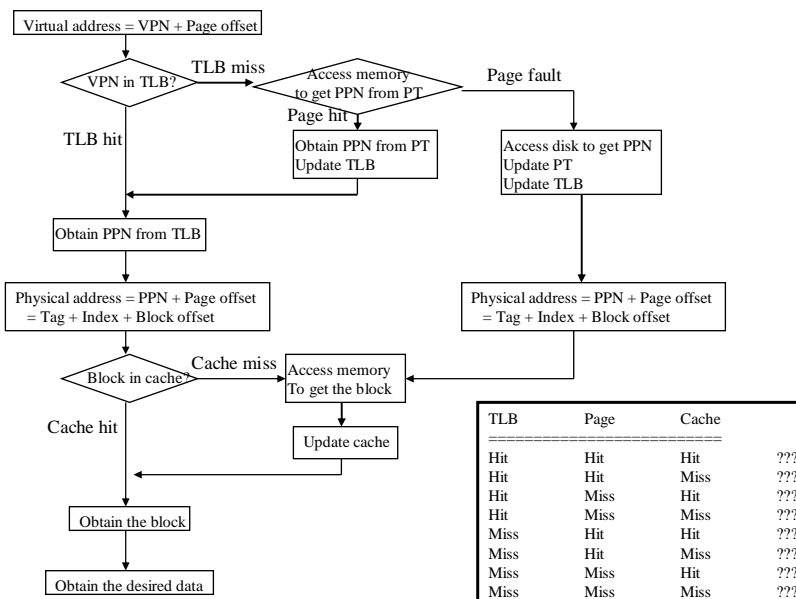
TLB (Translation Lookaside Buffer)

Now, the questions...

- Virtual address (VA) [776]
 - TLB hit, translated to [176]
 - Cache hit (block index=7, tag=1)
- VA [133]
 - TLB hit, translated [233]
 - Cache miss (block index=3, tag=2)
- VA [309]
 - TLB miss, PT (page table) lookup, translated to [009]
 - Cache hit (block index=0, tag=0)
- VA [452]
 - TLB miss, PT lookup, Page fault

23

Cache & VM Interaction



24

Cache & VM Interaction: Details of a Single Access

Access TLB.

Hit? →

TLB gives physical address of data.

No need to look at page table.

Access physical memory cache. (Assume one level cache for brevity.)

Hit? →

Cache gives data value.

Best case. No memory or disk access at all!

Miss? →

Access memory for data value.

Update cache.

25

Cache & VM Interaction: Details of a Single Access

...

TLB Miss? →

Access page table in physical memory.

(Assume single page table for brevity.)

(Page table entries could be in cache. This possibility omitted for brevity.)

No page fault? →

Page table gives physical memory address for data value.

Update TLB.

Access physical memory cache.

Hit? →

Cache gives data value.

Miss? →

Access memory for data value.

Update cache.

26

Cache & VM Interaction: Details of a Single Access

...

TLB Miss? →

...

Page fault? →

Page table gives disk location for data value.

Access disk for data value.

(Disks usually have caches too! This possibility omitted for brevity.)

Update memory with this page.

Update page table & TLB.

Worst case. Both memory & disk access!

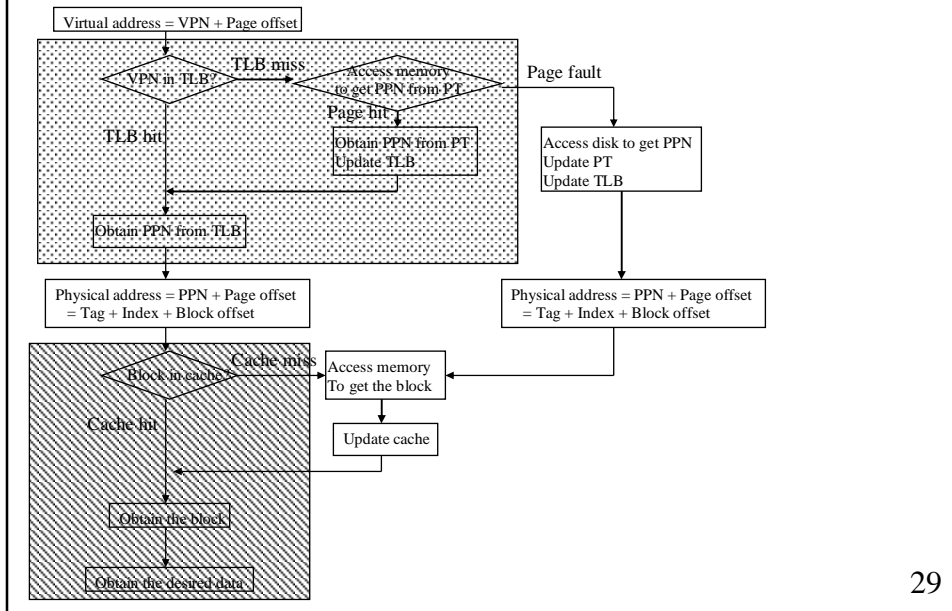
27

Memory Performance

- Most memory accesses require a sequence of two cache accesses
 - TLB cache access
 - Cache access
- How can it be faster?
 - Overlapping TLB & Cache access

28

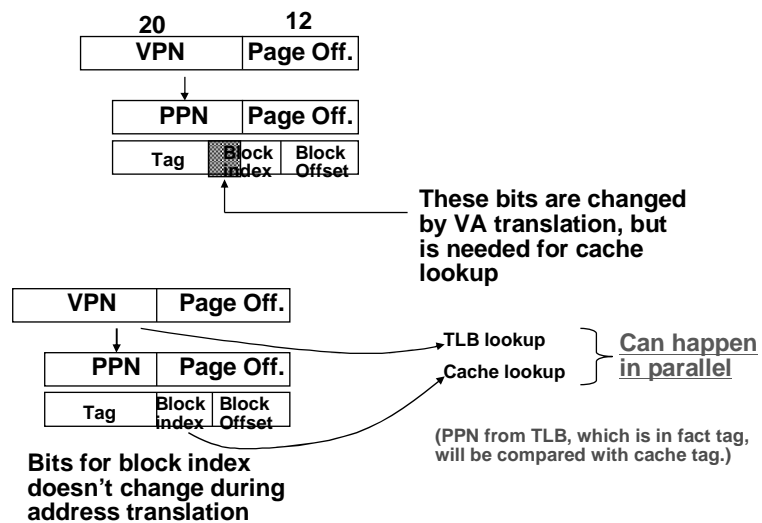
Cache & VM Interaction (revisited)



29

Problems With Overlapped TLB Access

Overlapped access only works as long as the address bits used to index into the cache do not change as the result of VA translation



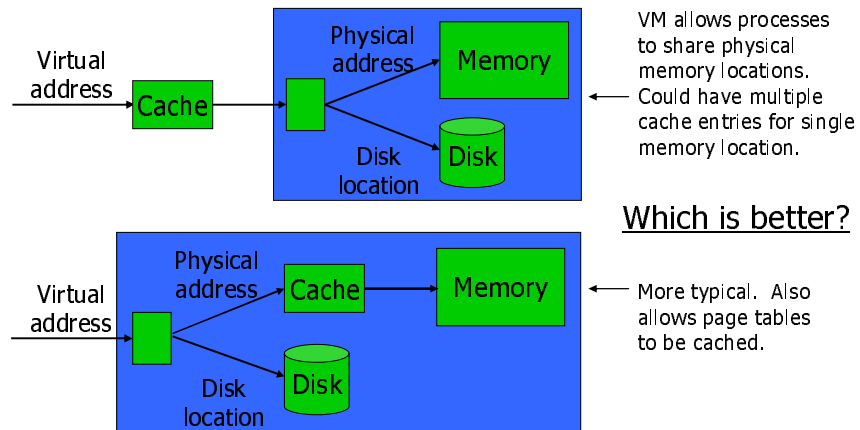
30

Questions

- Q1: How to expedite cache access?
 - Virtual addressed cache
- Q2: Page table per process? - Yes
- Q2: TLB per process? – No
 - Which means that we need to flush TLB upon a context switch
 - How to avoid that? : ASID (Address space identifier), ASN (AS number), PID (Process ID)
- Q3: Large PT size
 - 4GB VM (2^{32}) with 4KB page (2^{12}) gives 2^{20} PTEs (page table entries)
 - 4 bytes per PTE means 4MB per PT
 - 100 processes means 400MB for 100 PTs
- Q4: Page size
- Q5: Page fault (page replacement)
- Q6: Unit of data transfer

31

Q1: Virtual Cache



32

Virtual Cache

TLB access is required only on cache miss !!
(most memory access requires one cache access)

synonym problem: two different virtual addresses map to same physical address => two different cache entries holding data for the same physical address!

nightmare for update: must update all cache entries with same physical address or memory becomes inconsistent

determining this requires significant hardware, essentially an associative lookup on the physical address tags to see if you have multiple hits. (usually disallowed by fiat)

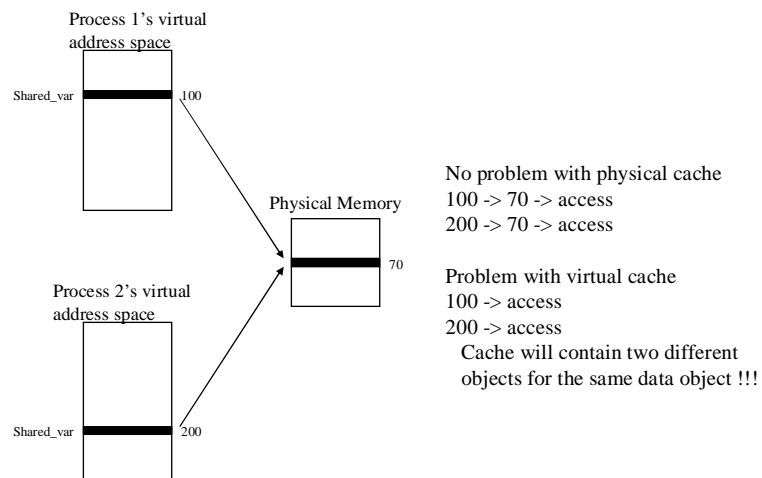
or

software enforced alias boundary: same lsb of VA & PA > cache size

Possible but simple solution:
clear cache when context switching

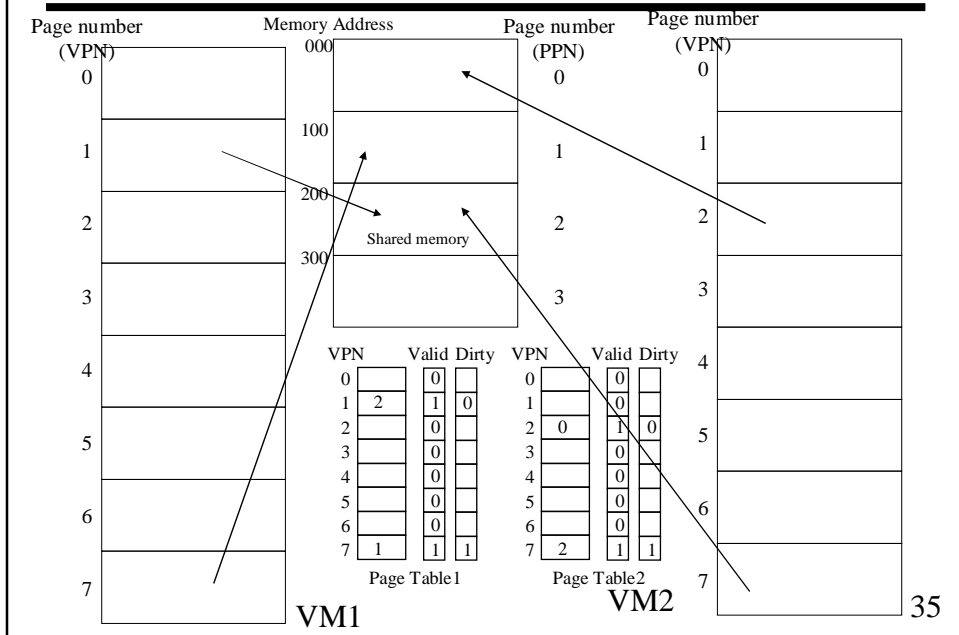
33

Virtual Cache



34

Q2: PT per Process



Virtual Memory & Context Switch

Why context switching?

Upon an I/O operation, CPU cannot proceed further until I/O completes (stalls)
 Allow CPU to execute other processes while waiting

When multiple users are supported

CPU allocates its time to multiple users in a time-shared fashion

Context switch → Save current process' state. Restore next process' state.

What are the state – PC, Registers, Memory (Page Table), and TLB

Saving & restoring registers:

Use memory to store **all** registers, including pointer to page table.
 Very expensive!

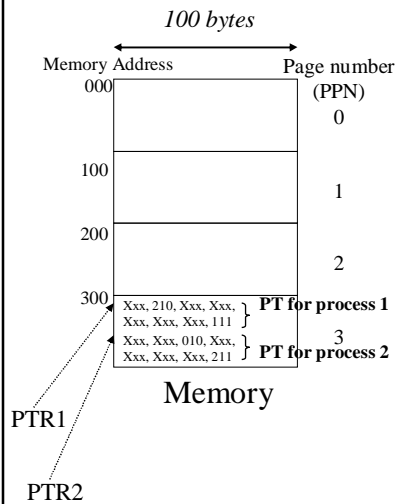
Memory (Page Table):

Just save/restore PTR rather than save/restore the entire PT

TLB caches for only one process' page table? → Clear TLB. Increases compulsory misses.

TLB caches for all processes' page tables? → Process id (or its page table pointer) must be provided to TLB.

How to deal with more than one Page Tables?



- Two PTs reside in memory
- When context switching to process 2
 - PT must be replaced
 - Just change PTR value
 - I.e., PTR = PTR2
- How about TLB?
 - Only one TLB for all processes
 - Is the first entry for the 1st process or the second process?
 - Need to flush TLB when context switching

VPN	PPN	Valid	Dirty
7	1	1	0
1	2	1	1

TLB

37

Q2+: How to avoid flushing TLB upon a context switch ?

- How to avoid flushing TLB upon a context switch ?
 - ASID (Address space identifier)
 - ASN (AS number)
 - PID (Process ID)

38

MIPS R3000

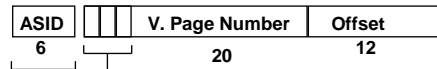
Pipeline

Inst Fetch		Dcd/ Reg		ALU / E.A		Memory		Write Reg	
TLB	I-Cache	RF	Operation				WB		
			E.A.	TLB	D-Cache				

TLB

64 entry, on-chip, fully associative, software TLB fault handler

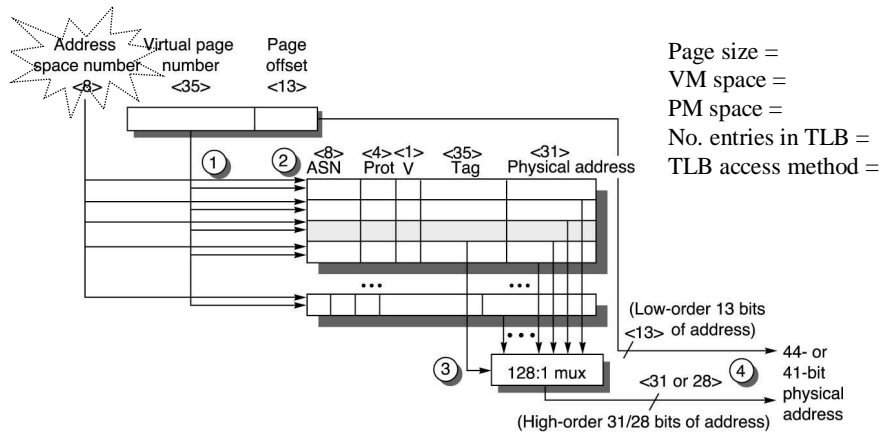
Virtual Address Space



0xx User segment (caching based on PT/TLB entry)
 100 Kernel physical space, cached
 101 Kernel physical space, uncached
 11x Kernel virtual space

Allows context switching among
 64 user processes without TLB flush

Alpha 21164



Q3: Page Table Overhead

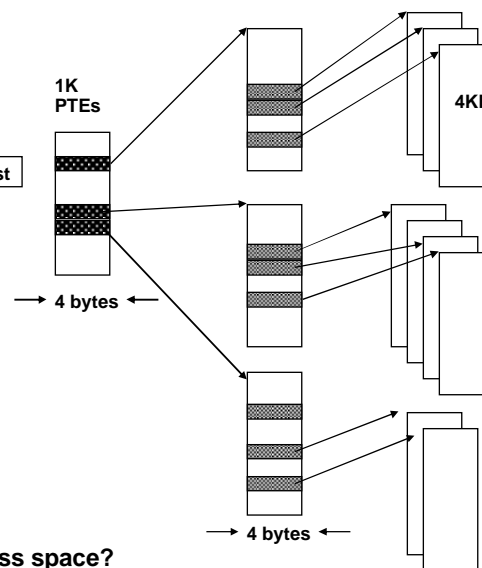
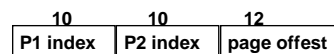
- 4MB page table size per each process => 400MB for 100 processes
- Bound register
 - That limits the size of the page table for a given process
 - If VPN becomes larger than the bound register, entries must be added to the PT
- Hashing function with inverted page table => next slide
- Multiple levels of page tables => next slide
- Allow page tables to be paged (PT also in virtual address space)

41

Large Address Spaces

Two-level Page Tables

32-bit address:



° 2 GB virtual address space

° 4 MB of PTE2

– paged, holes

° 4 KB of PTE1

What about a 48-64 bit address space?

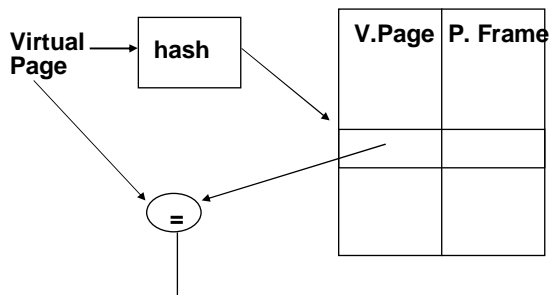
42

Inverted Page Tables

IBM System 38 (AS400) implements 64-bit addresses.

48 bits translated

start of object contains a 12-bit tag



=> TLBs or virtually addressed caches are critical

43

Q4: Optimal Page Size

°Mimimize wasted storage

- small page minimizes internal fragmentation
- small page increase size of page table

°Minimize transfer time

- large pages (multiple disk sectors) amortize access cost
- sometimes transfer unnecessary info
- sometimes prefetch useful data
- sometimes discards useless data early

General trend toward larger pages because

- big cheap RAM
- increasing mem / disk performance gap
- larger address spaces

44

Q5: Page Fault: What happens when you miss?

- Not talking about TLB miss
 - TLB is HWs attempt to make page table lookup fast (on average)
- Page fault means that page is not resident in memory
- Hardware must detect situation
- Hardware cannot remedy the situation
- Therefore, hardware must trap to the operating system so that it can remedy the situation
 - pick a page to discard (possibly writing it to disk)
 - load the page in from disk
 - update the page table
 - resume to program so HW will retry and succeed!
- What is in the page fault handler?
- What can HW do to help it do a good job?

- => “Context switching” during disk read

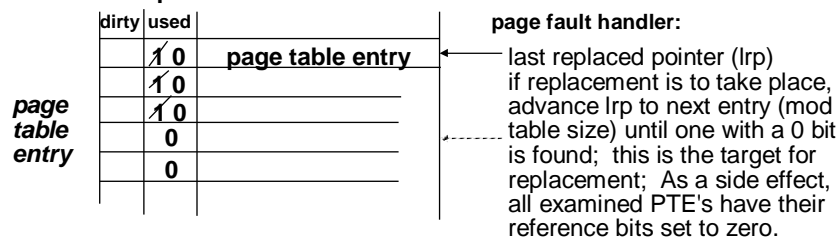
45

Page Replacement: “Not” Recently Used (1-bit LRU, Clock)

Associated with each page is a reference flag such that
 ref flag = 1 if the page has been referenced in recent past
 = 0 otherwise

CPU clears the flags periodically

-- if replacement is necessary, choose any page frame such that its reference bit is 0. This is a page that has not been referenced in the recent past



Or search for the a page that is both not recently referenced AND not dirty.

Architecture part: support dirty and used bits in the page table
 => may need to update PTE on any instruction fetch, load, store
 How does TLB affect this design problem? Software TLB miss?

46

Q6: Unit of data transfer

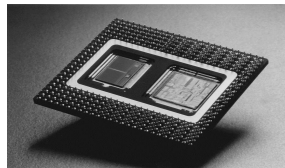
- CPU – Cache : word
- Cache – Memory : block
- Memory – Disk : page

47

Modern Systems

- Very complicated memory systems:

Characteristic	Intel Pentium Pro	PowerPC 604
Virtual address	32 bits	62 bits
Physical address	32 bits	32 bits
Page size	4 KB, 4 MB	4 KB, selectable, and 256 MB
TLB organization	A TLB for instructions and a TLB for data Both four-way set associative Pseudo-LRU replacement Instruction TLB: 32 entries Data TLB: 64 entries TLB misses handled in hardware	A TLB for instructions and a TLB for data Both two-way set associative LRU replacement Instruction TLB: 128 entries Data TLB: 128 entries TLB misses handled in hardware

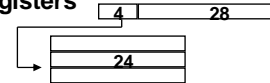


Characteristic	Intel Pentium Pro	PowerPC 604
Cache organization	Split instruction and data caches	Split instruction and data caches
Cache size	8 KB each for instructions/data	16 KB each for instructions/data
Cache associativity	Four-way set associative	Four-way set associative
Replacement	Approximated LRU replacement	LRU replacement
Block size	32 bytes	32 bytes
Write policy	Write-back	Write-back or write-through

48

Survey

- R4000
 - 32 bit virtual, 36 bit physical
 - variable page size (4KB to 16 MB)
 - 48 entries mapping page pairs (128 bit)
- MPC601 (32 bit implementation of 64 bit PowerPC arch)
 - 52 bit virtual, 32 bit physical, 16 segment registers
 - 4KB page, 256MB segment
 - 4 entry instruction TLB
 - 256 entry, 2-way TLB (and variable sized block xlate)
 - overlapped lookup into 8-way 32KB L1 cache
 - hardware table search through hashed page tables
- Alpha 21064
 - arch is 64 bit virtual, implementation subset: 43, 47,51,55 bit
 - 8,16,32, or 64KB pages (3 level page table)
 - 12 entry ITLB, 32 entry DTLB
 - 43 bit virtual, 28 bit physical octword address



49

Summary: Why virtual memory?

- *Generality*
 - ability to run programs larger than size of physical memory
- *Storage management*
 - allocation/deallocation of variable sized blocks is costly and leads to (external) fragmentation
- *Protection*
 - regions of the address space can be R/O, Ex, . . .
- *Flexibility*
 - portions of a program can be placed anywhere, without relocation
- *Storage efficiency*
 - retain only most important portions of the program in memory
- *Concurrent I/O*
 - execute other processes while loading/dumping page
- *Expandability*
 - can leave room in virtual address space for objects to grow.
- *Performance*

Observe: impact of multiprogramming, impact of higher level languages

50

Decoding Memory Address

- Memory address [776] (VA)
 - ↓ offset within the page
 - Virtual page number
- With the memory address
 - Extract the VPN (7)
 - Lookup the PT to see the corresponding PPN (Physical Page Number), 1
 - Address translation is 776 => 176
- Memory address [176] (PA)
 - ↓ offset within the block
 - ↓ cache block index
 - for identifying the original memory block
- With the memory address
 - Extract the cache block index (7)
 - Check if the cache block #7 corresponds to memory 170~179
 - For this, each cache entry remembers the “tag” data (e.g. “1”)
 - Extract the byte within the block with offset address (e.g. 6)

Address translation

Cache access

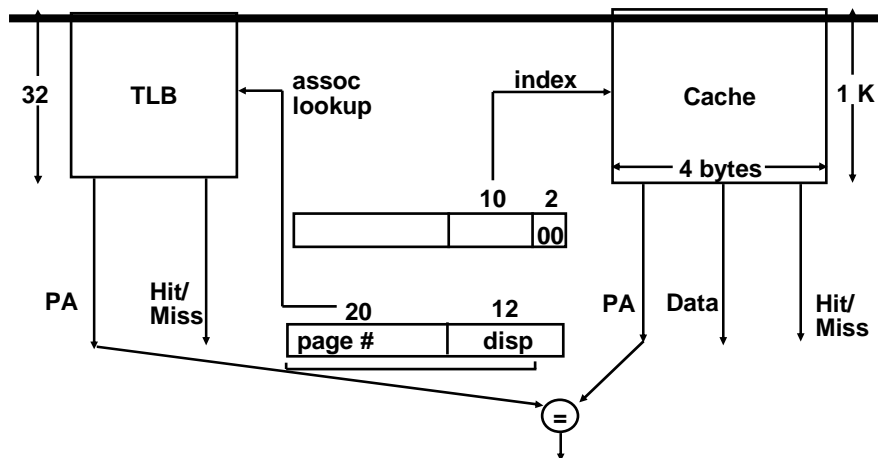
Decoding Memory Address

- Memory address [776] (VA)
 - ↓ offset within the page
 - Virtual page number
- With the memory address
 - Extract the VPN (7)
 - Lookup the PT to see the corresponding PPN (Physical Page Number), 1
 - Address translation is 776 => 176
- Memory address [176] (PA)
 - ↓ offset within the block
 - cache block index
 - for identifying the original memory block
- With the memory address
 - Extract the cache block index (7)
 - Check if the cache block #7 corresponds to memory 170~179
 - For this, each cache entry remembers the “tag” data (e.g. “1”)
 - Extract the byte within the block with offset address (e.g. 6)

- Access TLB with VPN (fully associative cache)
- If found, PPN (1) can be obtained without memory access (PT)

53

Overlapped Cache & TLB Access



IF cache hit AND (cache tag = PA) then deliver data to CPU
ELSE IF [cache miss OR (cache tag = PA)] and TLB hit THEN
access memory with the PA from the TLB
ELSE do standard VA translation

54

Hardware / Software Boundary

- **What aspects of the Virtual -> Physical Translation is determined in hardware?**
 - **TLB Format**
 - **Type of Page Table**
 - **Page Table Entry Format**
 - **Disk Placement**
 - **Paging Policy**