ECE 485/585
Microprocessor System Design

Prof. Mark G. Faust

Maseeh College of Engineering
and Computer Science

PORTLAND STATE UNIVERSITY

Virtual Memory

ECE 485/585

Mark G. Faust

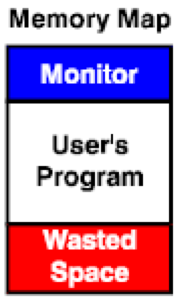
What is Virtual Memory

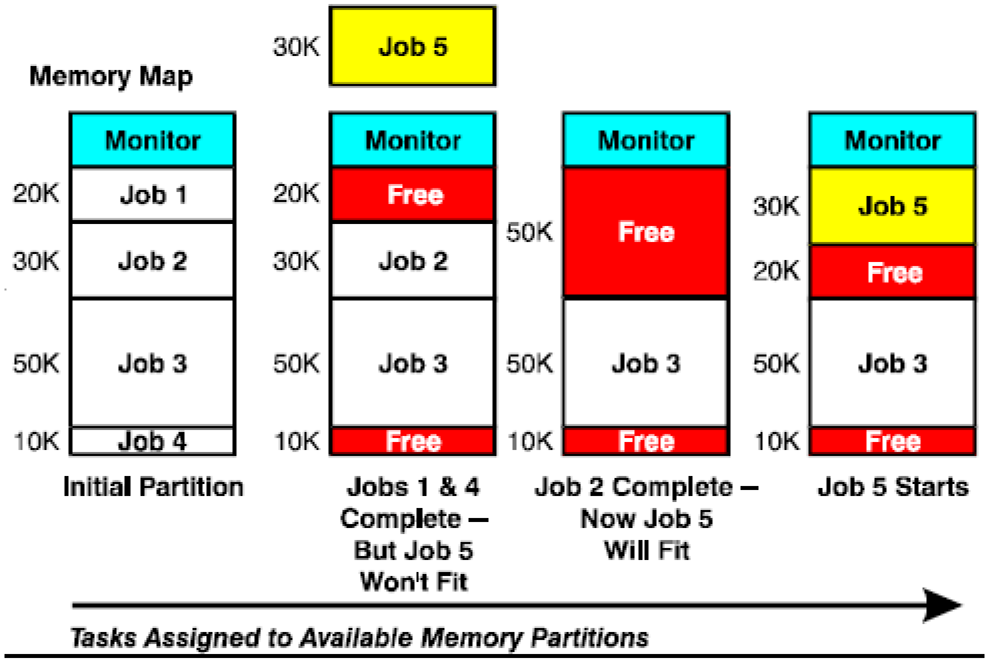
- Automatic address translation
  - Decouples program's addresses from physical location
  - Provides for program address space bigger than physical memory
  - Allows expanding program's address space without re-allocating existing memory
  - Provides protection from other tasks/processes
- Components of virtual memory
  - Physical memory partitioned into pages
  - Swap device (typically disk) holds pages not resident in physical memory
  - Address translation from program (virtual) address to physical address
    - Page tables
      - Provide translation from virtual to physical address (if resident)
      - Indicate where page can be found on swap device (backing store) if not resident
    - Translation lookaside buffer (TLB) is a cache of translations
  - Hardware/Software cooperation
    - TLB in hardware
    - Page table look up in H/W or S/W
    - Page table management in S/W

Origins of Virtual Memory

Early days of computing (pre-1960)
Single task batch systems: dedicated to one user
Unused memory space is wasted
Time spent waiting for I/O is wasted (CPU is idle)

Multi-task batch systems:
Run another task while one is waiting for I/O
Multiple programs resident in memory
New program needs to await free space
- must be contiguous
- can result in waits even when space available

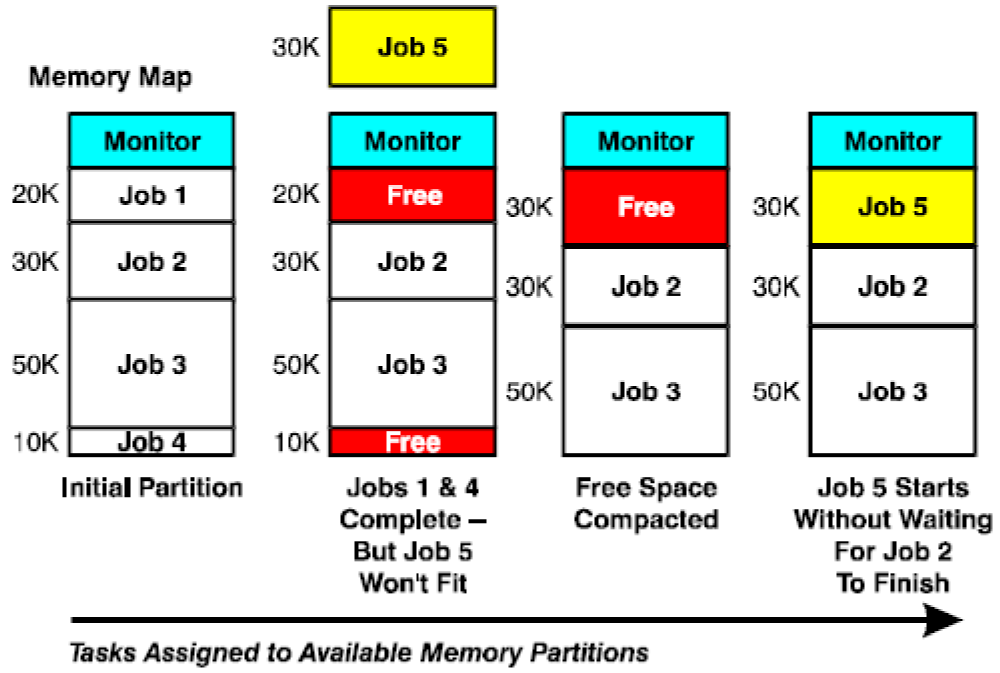



Memory Map
30K
Job 5
Monitor
20K
Job 1
30K
Job 2
50K
Job 3
10K
Job 4
Initial Partition
Free
Jobs 1 & 4 Complete – But Job 5 Won't Fit
50K
Job 2 Complete – Now Job 5 Will Fit
Job 5 Starts
Tasks Assigned to Available Memory Partitions

Origins of Virtual Memory

Relocation registers (IBM S/360 c. 1964)
Base register is added to all memory references
Rudimentary protection
  No addresses less than base register
  Exception if Base + Address exceeds Bound
Programs can be moved in physical memory
  - if we adjust base register

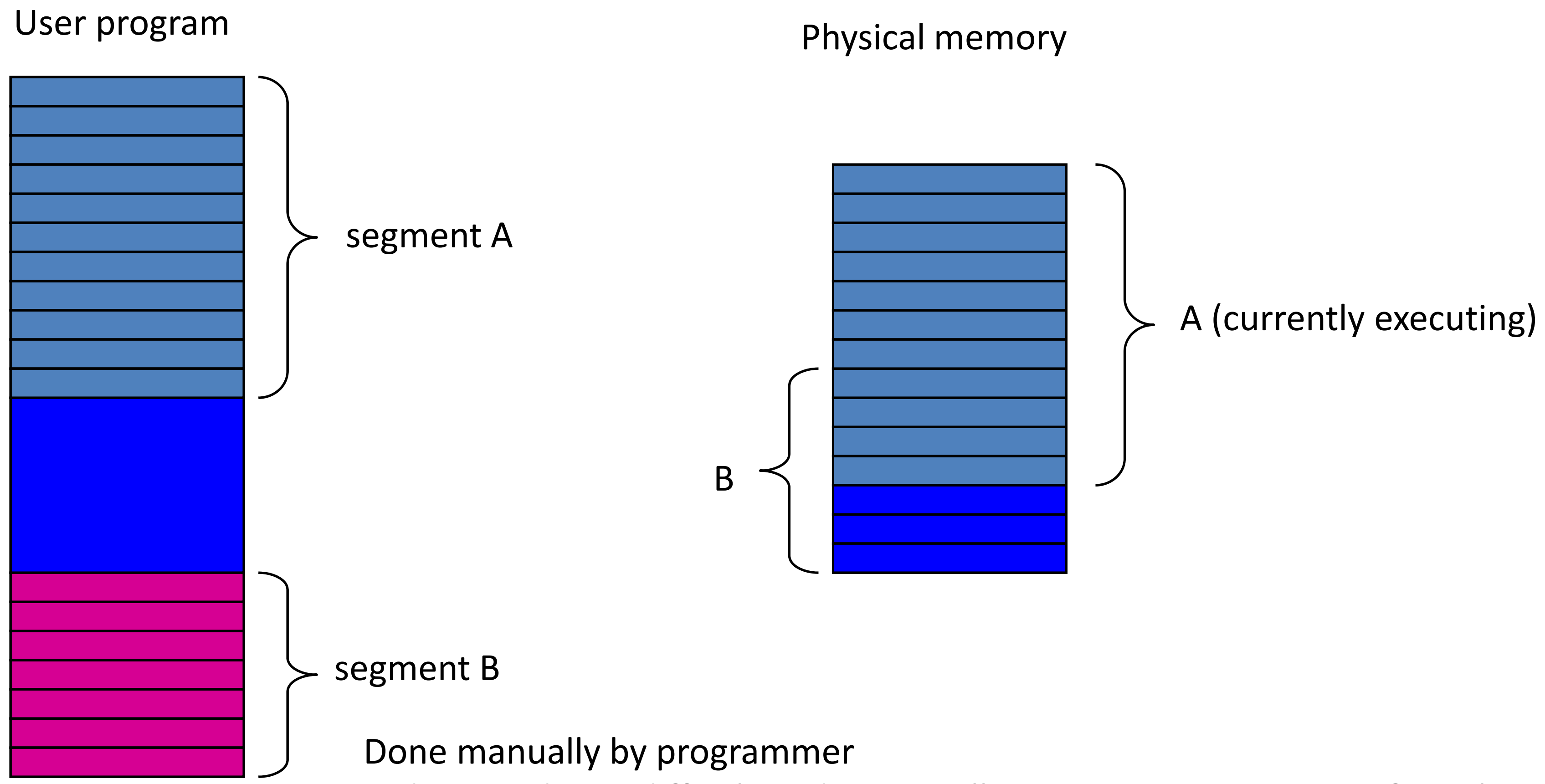
Still consume time by copying memory image
Doesn't allow programs > than physical memory



Origins of Virtual Memory

Use of overlays

Programs partitioned into segments which were overlayed in physical memory (to/from disk) as needed (entire program didn't need to reside in physical memory at the same time)



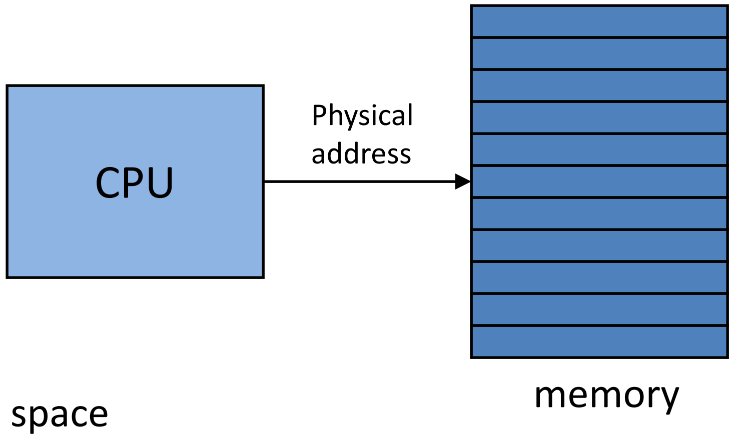
Done manually by programmer

Tedious and very difficult to do optimally to minimize swapping of overlays

Virtual Addressing

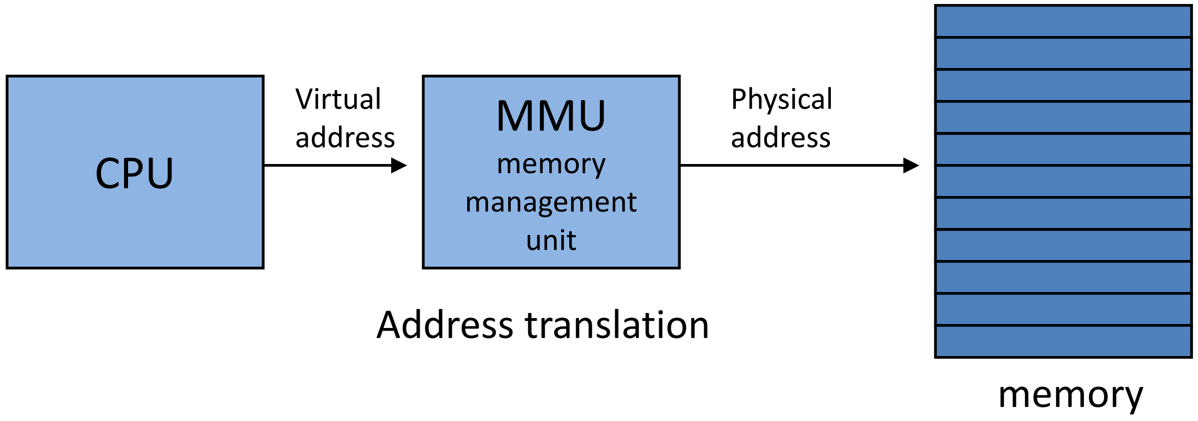
Early 1960s…

Physical addressing

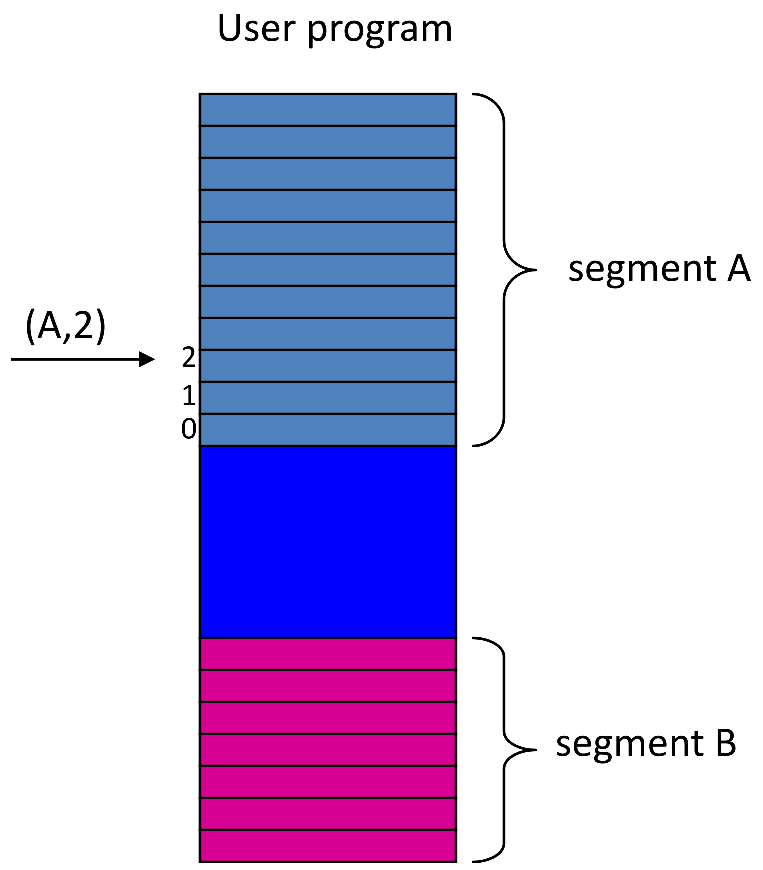


Mapping from virtual to physical addresses
Virtual address space could be larger than physical address space
Each process could have its own virtual address space

Virtual addressing



Segmentation

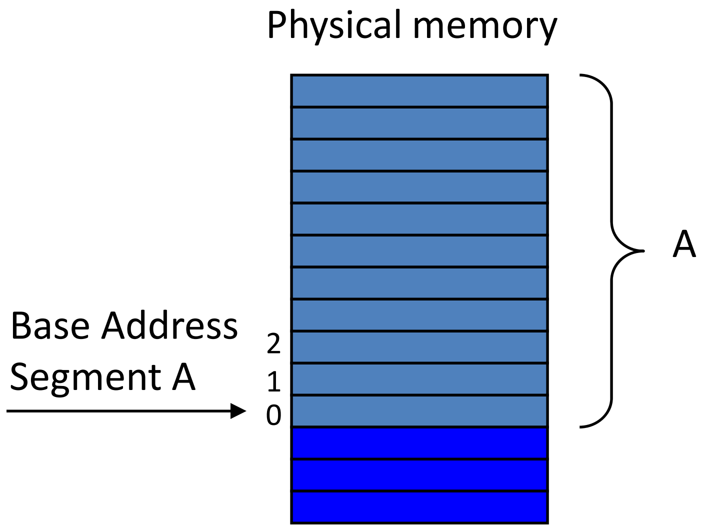


Address is (Segment, Offset) pair

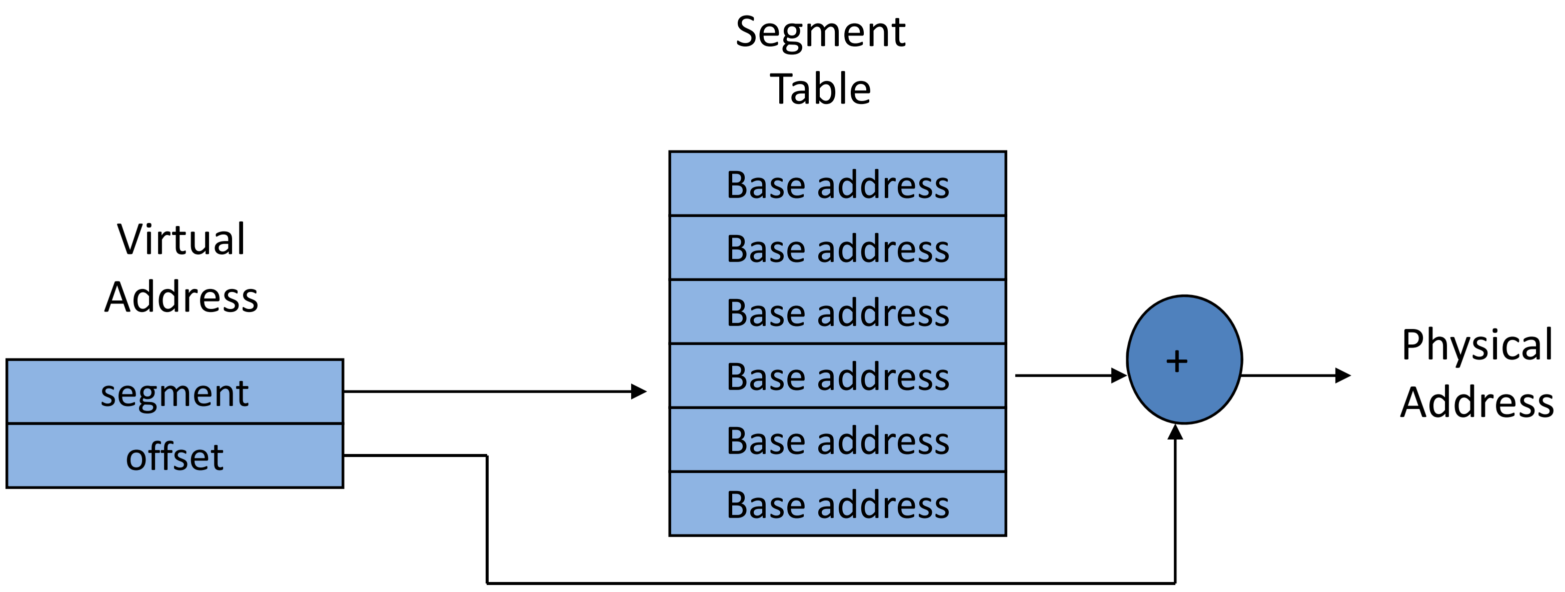
Each segment has own contiguous address space

Facilitates relocating code

- Segment can be relocated anywhere in physical memory
  - position independent code!
- Offsets in segment maintained
- Physical address is Base Address of Segment + Offset

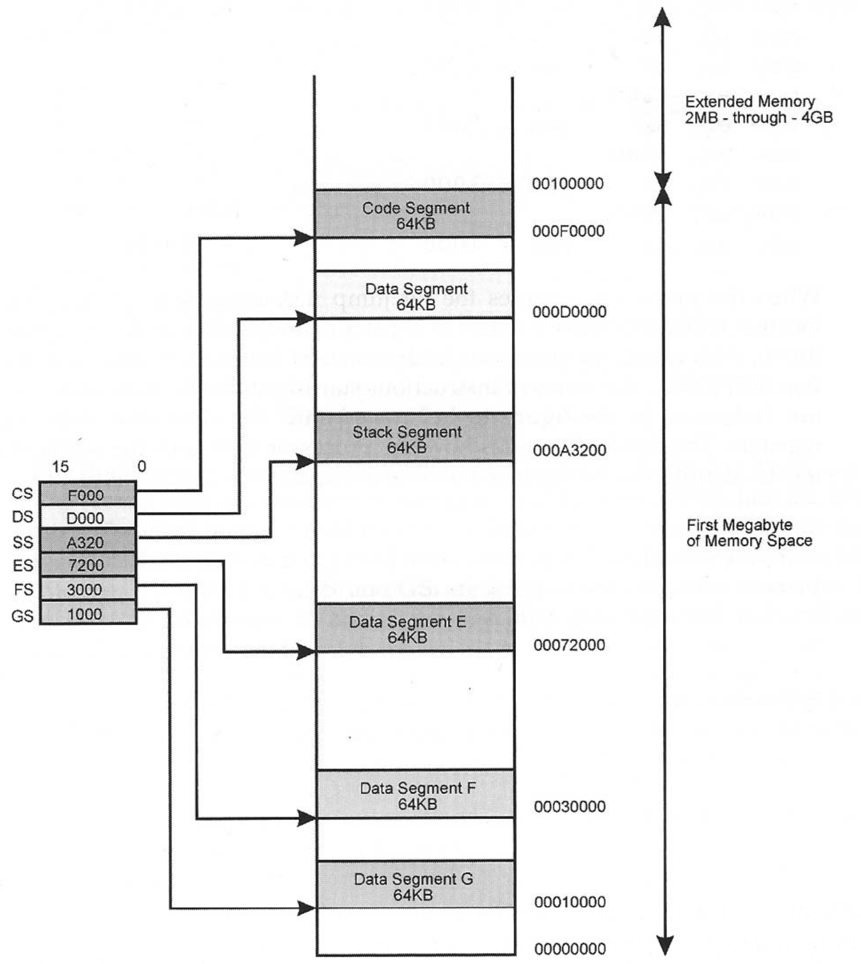


Segmentation



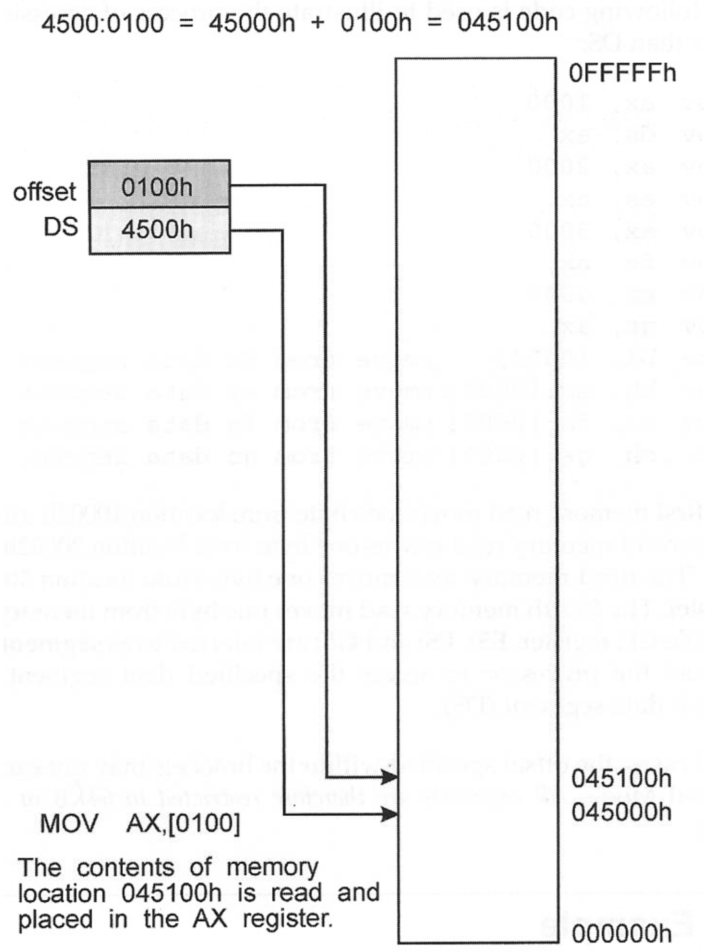
Segmentation: Intel

- Code and data references
  - Segment:Offset e.g. CS:IP
- Segment registers
  - Contain base address of segment
    - Code Segment (CS)
    - Default Data Segment (DS)
    - Stack Segment (SS)
    - Additional Data Segments
- Simplest case (e.g. Real Mode)
  - Linear Address =
    - Segment Base (shifted) + Offset



Real Mode Segmentation

- Default Data Segment Register (DS)
  - Contents shifted left 4 to form base address
  - Creates effective 20 bit base address
- Offset (actual operand)
  - Added to shifted base address to form linear address
- Same process occurs for
  - Normal instruction execution
    - Code Segment (CS)
  - Stack operations
    - Stack segment (SS)
- Explicit segment references allowed
  - "Segment overrides"
  - MOV BH, ES:[0002]


4500:0100 = 45000h + 0100h = 045100h
0FFFFFh
offset
0100h
DS
4500h
045100h
045000h
MOV AX,[0100]
The contents of memory location 045100h is read and placed in the AX register.
000000h

Segment-Level Protection

- Segmentation evolved to provide additional support
  - Protect OS and other processes from each other
    - Malicious process
    - Buggy process
  - Aid to debugging
    - Process memory image not corrupted
    - Debugger resident and not corrupted
- Every memory reference is checked
  - Type Check (e.g. don’t write to code segment)
  - Limit Check (e.g. offset within segment’s range)
  - Restriction of addressable domain (through segments)
  - Restriction of procedure entry points (can’t jump/branch to random places in other segments – e.g. system code!)
  - Restriction of instruction set (only OS can manipulate segment registers, etc)

Segment-Level Protection

- Restriction of instruction set
  - Reserved to Operating System (Level 0)

HLT	Halt
INVD	Invalidate the cache
INVLPG	Invalidate TLB Entry
LGDT	Load GDT register
LIDT	Load IDT register
LLDT	Load LDT register
LMSW	Load Machine Status Word
LTR	Load Task Register
MOV <CRn>	Move to Control Register
MOV <DRn>	Move to Debug Register
WBINVD	Write Back and Invalidate Cache

Segment-Level Protection

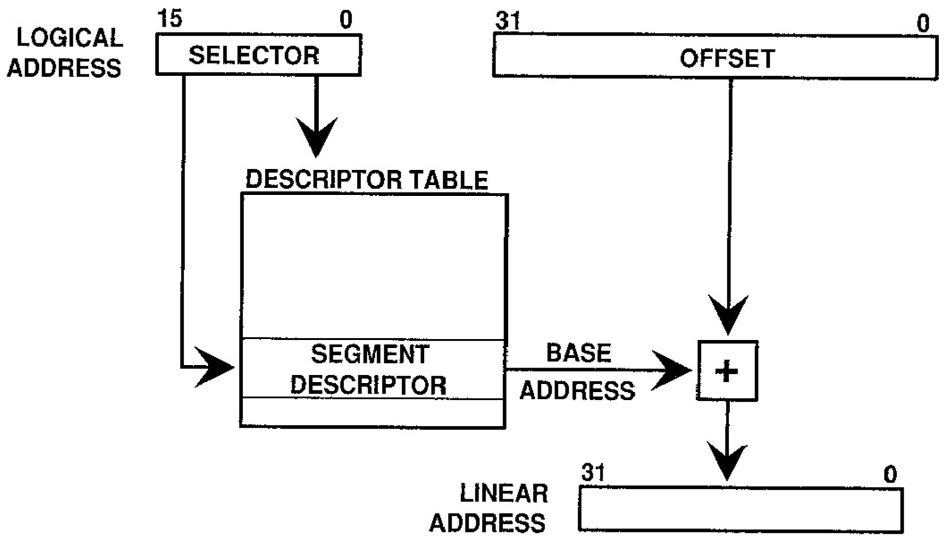
- Restriction of instruction set
  - IOPL (I/O protection level)
  - "Sensitive" instructions reserved to device drivers

IN	Input
INS	Input String
OUT	Output
OUTS	Output String
CLI	Clear Interrupt-Enable Flag
STI	Set Interrupt-Enable Flag

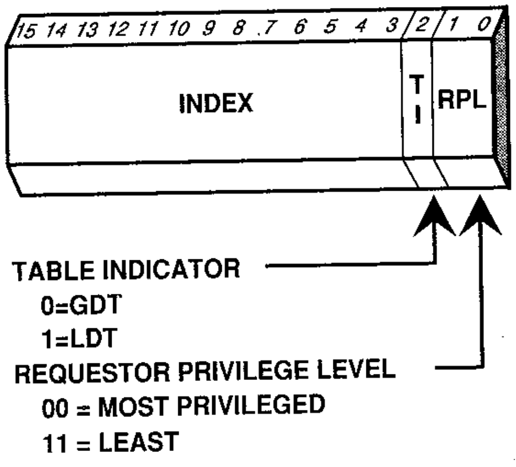
- Other checks
  - Stack on IRET and RET
    - Don't allow user process to set-up bogus return IP on stack

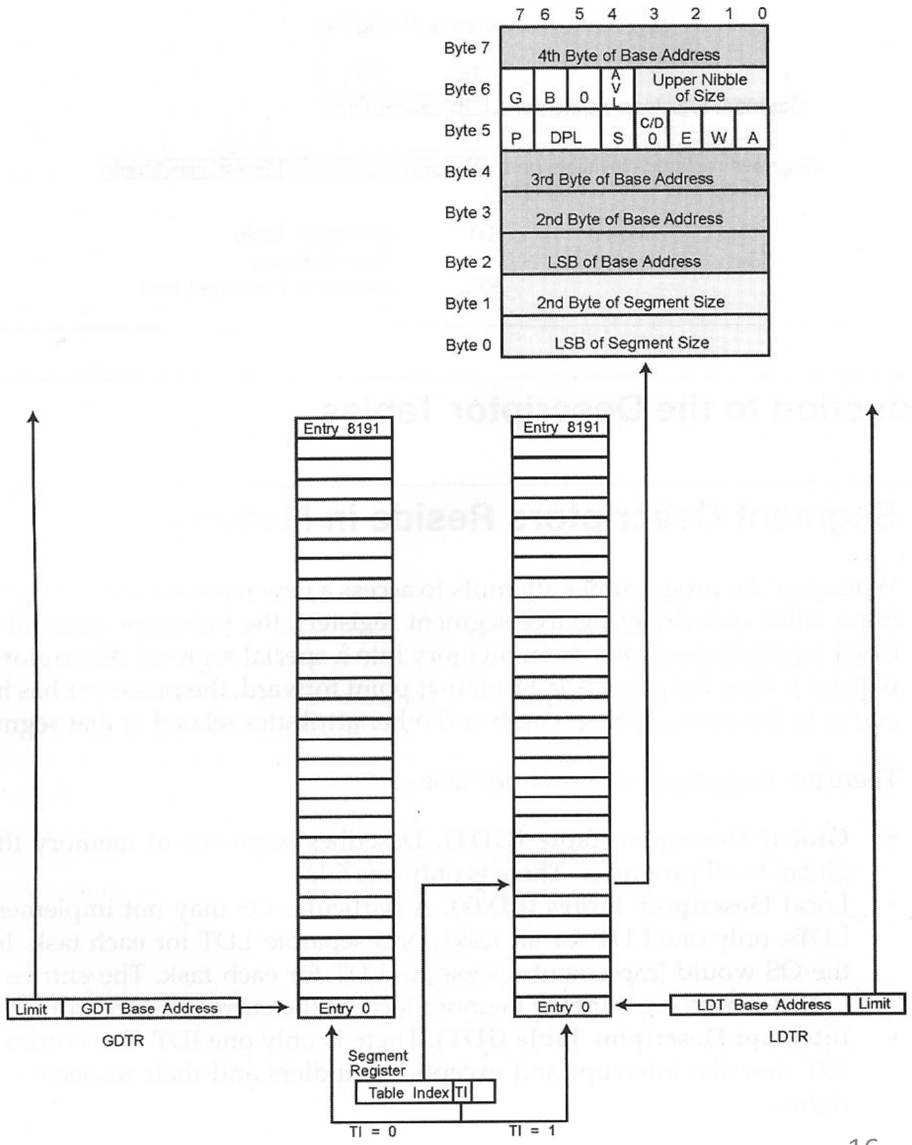
Protected Mode Segmentation

- Segment registers are used as selectors instead of actual base address
  - Index into either GDT or LDT
    - Global or Local Descriptor Table
    - Single GDT
    - 0, 1, or many LDTs
- Descriptors contain segment info
  - Base address
  - Limit
  - Privileges
- Hardware performs bounds checking
  - Data and Code Segments
- Hardware performs access checking
  - Code Segments



Protected Mode Segmentation





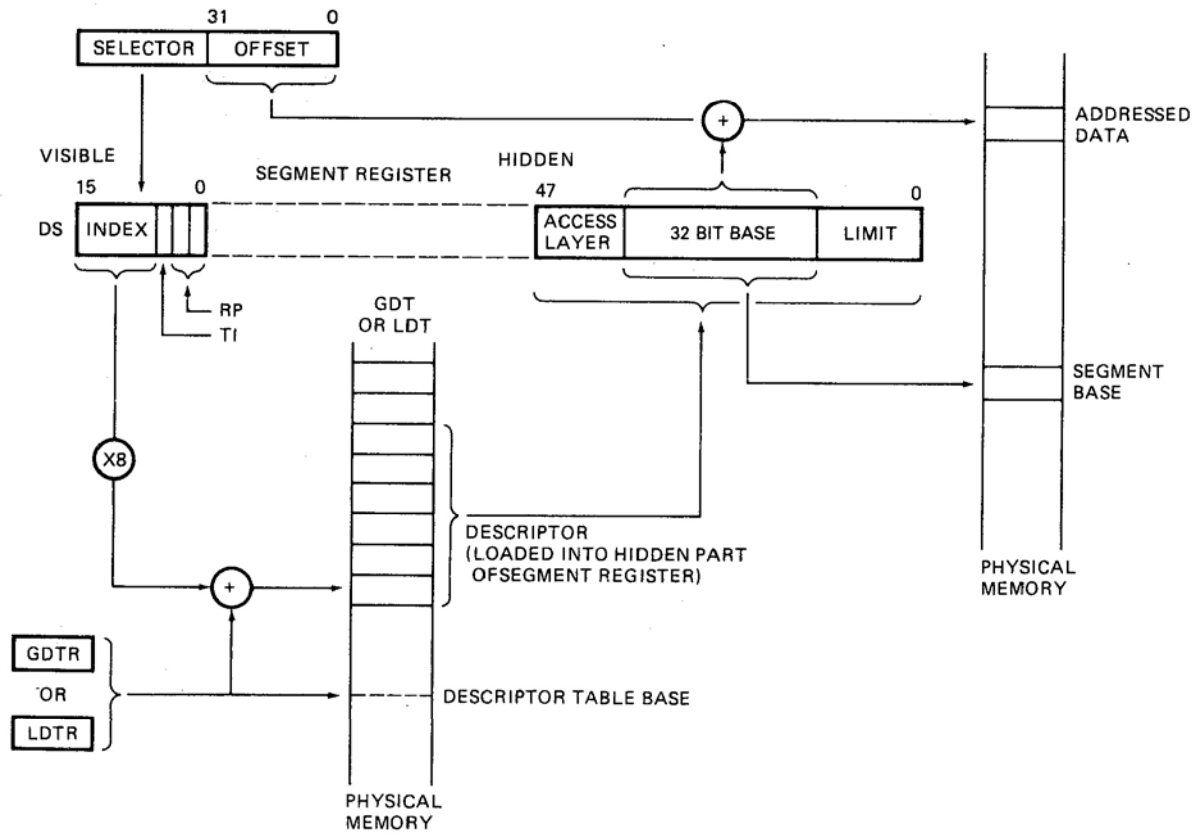
Entry 0 of GDT unused? Why?

New CS register ("selector") loaded upon CALL or long JMP

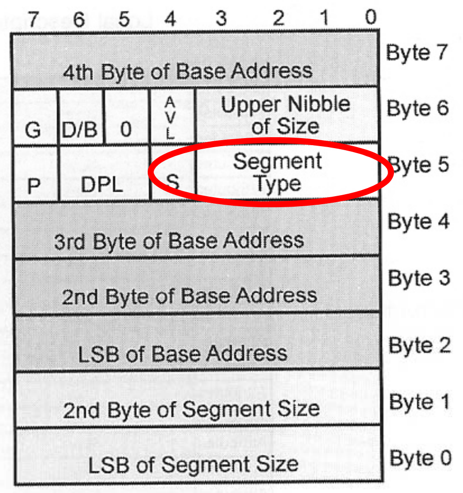
For efficiency, contents of descriptors for current segments as cached in hidden parts of registers

Descriptors are cached

- Descriptors are cached in hidden portions of the segment registers
- Saves memory accesses for each reference!



Segment Descriptors

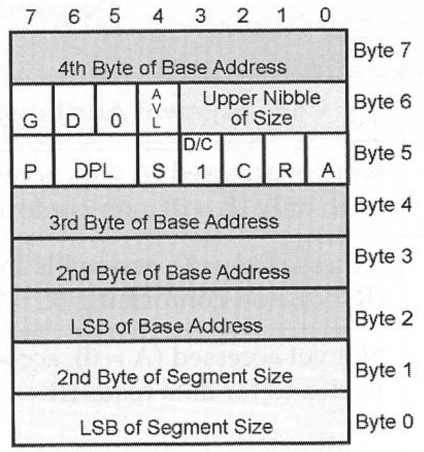


G Bit	Granularity bit defines meaning of limit value: 0 = length of segment in bytes. 1 = length of segment in pages.
Type Field	Defines a segment as data or code, read-only or read/write, expansion direction and whether it is has been accessed or not.
D/B Bit	In a code segment, D bit indicates default operand and address size. In data segment, B bit defines SP size and upper bound of expand-down stack.
AVL Bit	Available for use by system software.
P Bit	Segment Present bit.
DPL Field	Descriptor Privilege Level.
S Bit	System bit. When 0, indicates system segment.
Size Field	Specifies the size of the segment (also see the G bit).
Base address	Specifies the 32-bit base linear address of the segment in the 4GB linear address space.

Code
Data
Call Gate

Protected Mode Segmentation

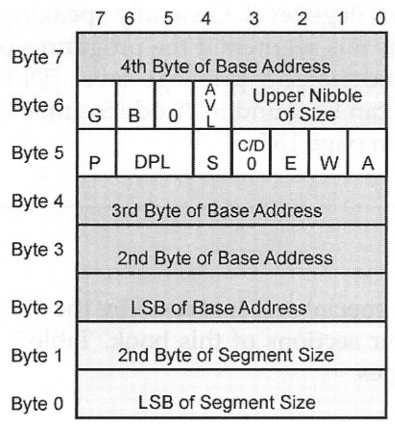
- Code Segment Descriptors
  - 32-bit base address
  - 20-bit segment size (limit)
    - G bit (granularity) indicates whether limit is
      - Bytes
      - 4K pages (4K = 2^{12}) therefore 2^{32} = 4GB
- Types of code segments
  - Conforming
    - Used for shared libraries
    - Segment conforms to privilege of caller
  - Non-Conforming



G Bit	Granularity bit defines meaning of limit value. 0 = length of segment in bytes. 1 = length of segment in 4KB pages.
D Bit	In code segment, Default bit defines default size of operands and effective addresses. 0 = 16-bit, 1 = 32-bit.
AVL Bit	Available for use by system software
P Bit	Segment Present bit (must be 1 if the code segment is present in memory).
DPL Field	Descriptor Privilege Level (0-3)
S Bit	System bit. When 0, indicates system segment. Must be 1 in a code segment descriptor.
D/C	This could be called the Data/Code bit. A 0 indicates a data segment and a 1 indicates a code segment.
C Bit	Conforming bit. Set to 1 if code segment is conforming. See text for a detailed description.
R Bit	Readable bit. A 0 indicates an execute-only segment, while a 1 indicates the segment may be read from by both the prefetcher and for data accesses.
A Bit	Accessed bit. Set to 1 by the processor when a code segment is accessed.

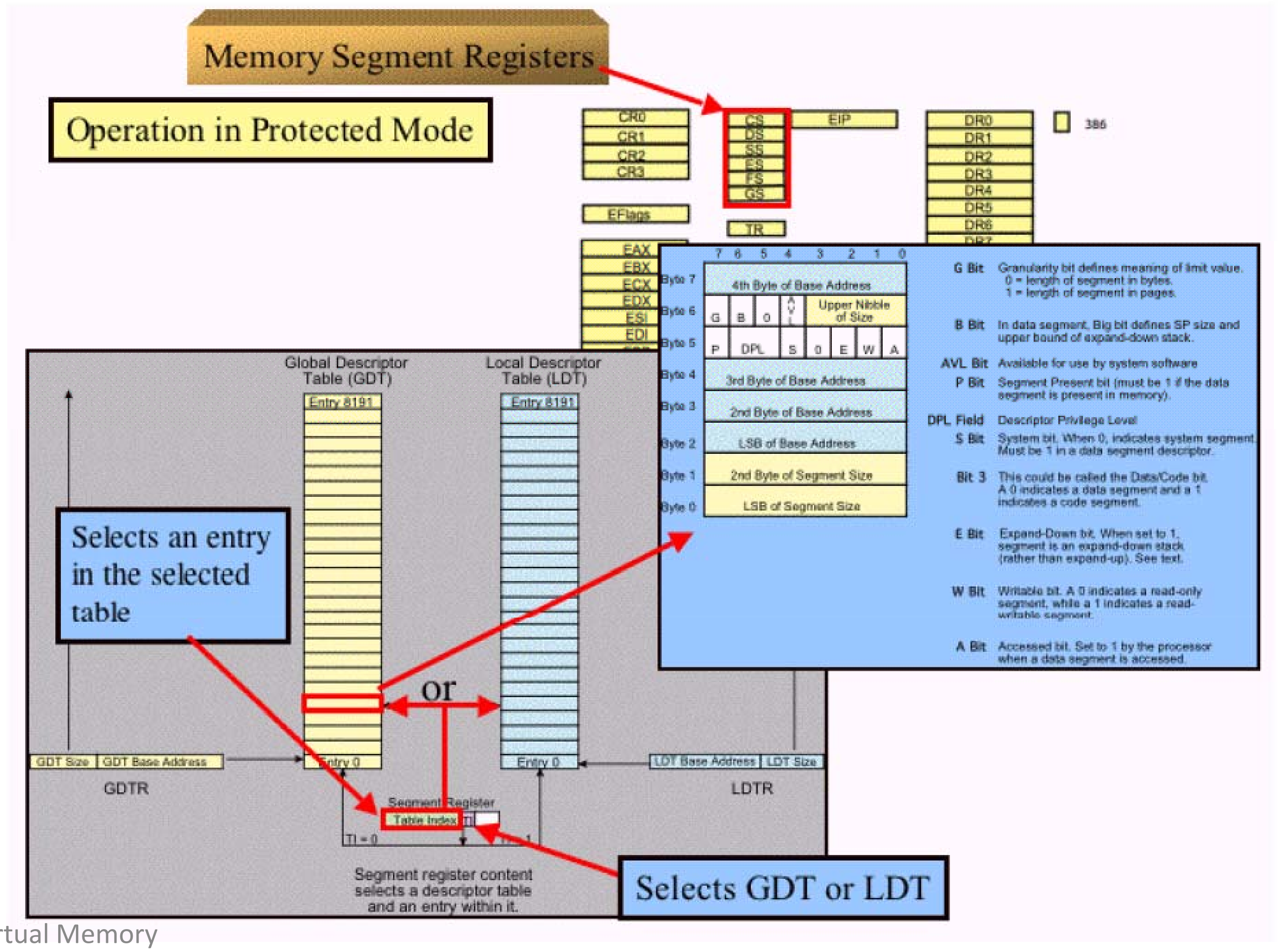
Protected Mode Segmentation

- Data Segment Descriptors



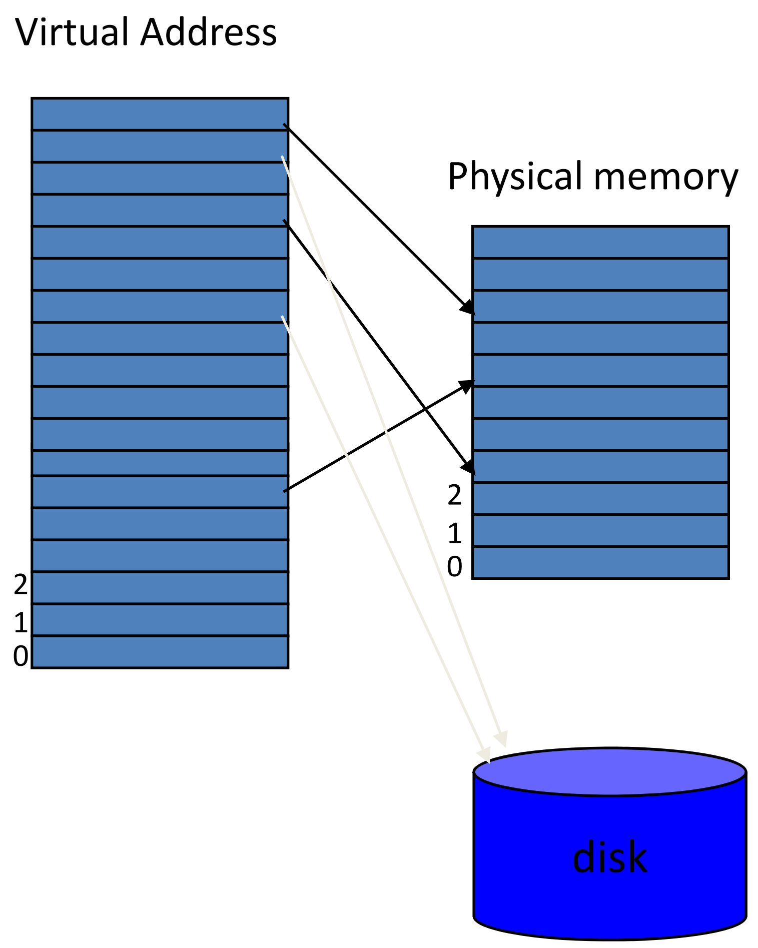
G Bit	Granularity bit defines meaning of limit value: 0 = length of segment in bytes. 1 = length of segment in pages.
B Bit	In data segment, Big bit defines SP size and upper bound of expand-down stack.
AVL Bit	Available for use by system software
P Bit	Segment Present bit (must be 1 if the data segment is present in memory).
DPL Field	Descriptor Privilege Level
S Bit	System bit. When 0, indicates system segment. Must be 1 in a data segment descriptor.
Bit 3	This could be called the Data/Code bit. A 0 indicates a data segment and a 1 indicates a code segment.
E Bit	Expand-Down bit. When set to 1, segment is an expand-down stack (rather than expand-up). See text.
W Bit	Writable bit. A 0 indicates a read-only segment, while a 1 indicates a read-writable segment.
A Bit	Accessed bit. Set to 1 by the processor when a data segment is accessed.

Protected Mode Segmentation



G Bit	Granularity bit defines meaning of limit value. 0 = length of segment in bytes. 1 = length of segment in pages.
B Bit	In data segment, Big bit defines SP size and upper bound of expand-down stack.
AVL Bit	Available for use by system software
P Bit	Segment Present bit (must be 1 if the data segment is present in memory).
DPL Field	Descriptor Privilege Level
S Bit	System bit. When 0, indicates system segment. Must be 1 in a data segment descriptor.
Bit 3	This could be called the Data/Code bit. A 0 indicates a data segment and a 1 indicates a code segment.
E Bit	Expand-Down bit. When set to 1, segment is an expand-down stack (rather than expand-up). See text.
W Bit	Writable bit. A 0 indicates a read-only segment, while a 1 indicates a read-writable segment.
A Bit	Accessed bit. Set to 1 by the processor when a data segment is accessed.

Paging



Segments too large to provide sufficient granularity
- entire segment had to be in memory
- variable size segments made allocation of physical memory difficult

So… paging was created.

Basic idea:
Virtual addresses mapped to physical addresses
Data corresponding to a virtual address may be
- in physical memory
- on disk
- unallocated

But…managing <u>individual</u> addresses too cumbersome and uses too much memory

Use fixed-length <u>pages</u> – an entire page is mapped and either resident in memory or on disk

Worked so well, that over time, features of segments incorporated into paging (protection, sharing)

Benefits of Virtual Memory

1. Provides every process with large (linear) virtual address space (may be larger than physical memory)

2. Facilitates code sharing (different virtual addresses can map to same physical address, eliminating need to have multiple copies of shared libraries – how many copies of `printf()` in memory?)

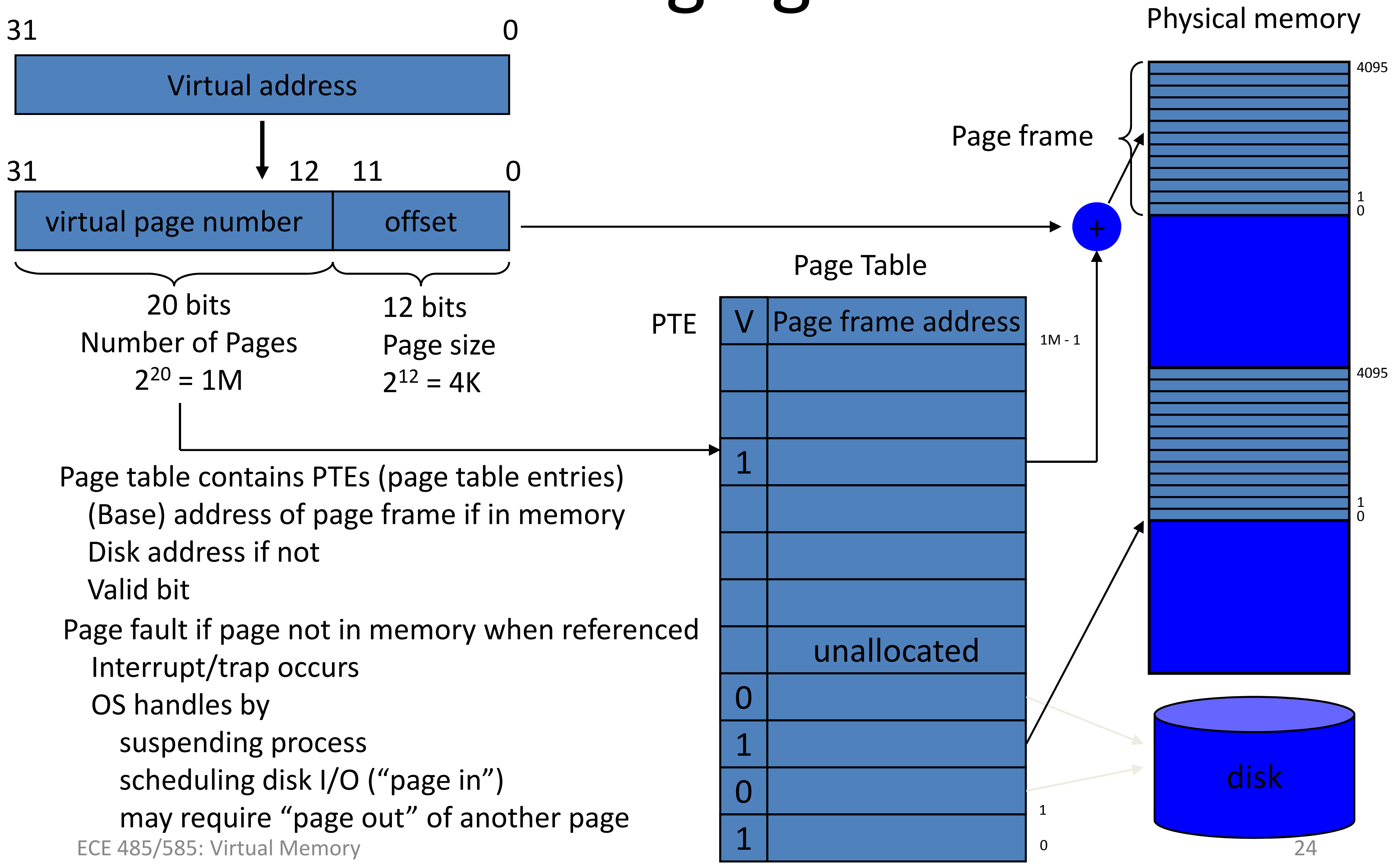
3. Facilitates position independent code (data/code can be relocated anywhere in physical memory)

4. Facilitates debugging by providing mechanisms for checking references (e.g. attempts to access unallocated memory, null pointers)

5. Provides protection from intentional or accidental attempts to access or write memory belonging to another user process (or the OS)

6. No need to have entire process memory image loaded to begin execution ("demand paging")

Paging

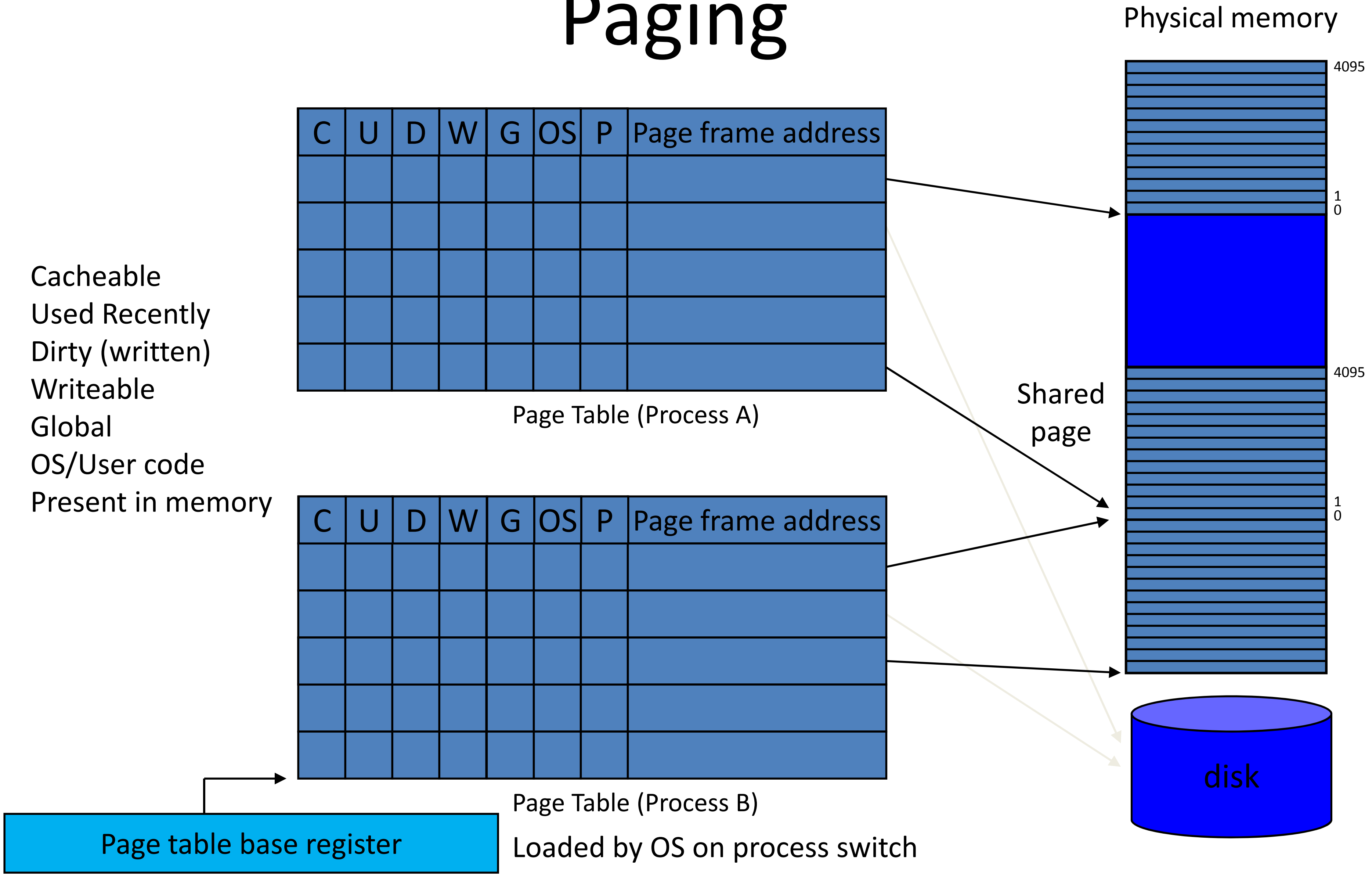


Page table contains PTEs (page table entries)
- (Base) address of page frame if in memory
- Disk address if not
- Valid bit

Page fault if page not in memory when referenced
- Interrupt/trap occurs
- OS handles by
  - suspending process
  - scheduling disk I/O (“page in”)
  - may require “page out” of another page

Paging

Cacheable
Used Recently
Dirty (written)
Writeable
Global
OS/User code
Present in memory



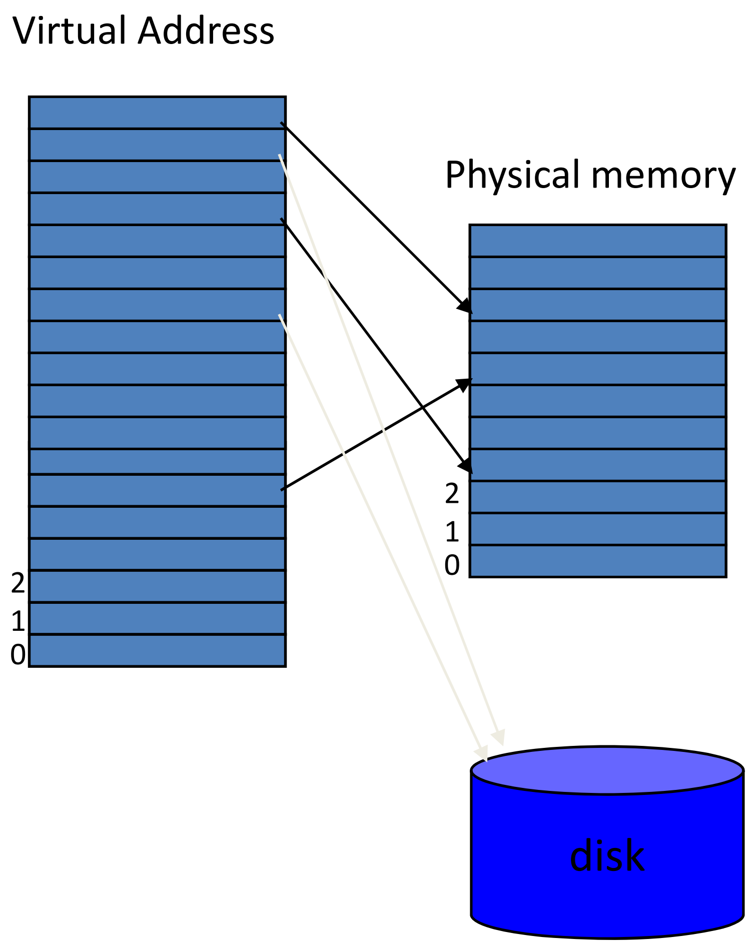
Loaded by OS on process switch

Similarities to Caching

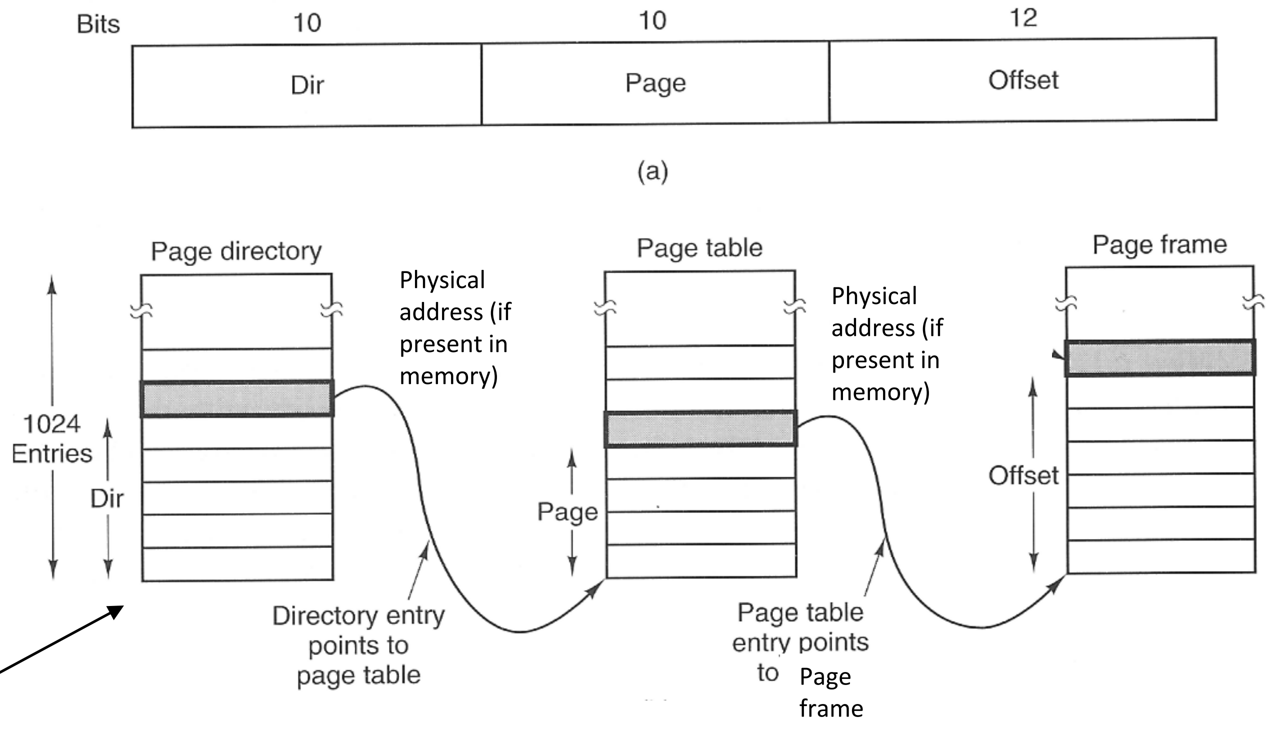
- “Present” bit → “Valid” bit
  - Indicates whether page is present in memory
  - Indicates whether line in cache is valid
- Page Fault → Cache Miss
  - Page not resident in memory, page in from disk
  - Line not in cache, cache line fill from next level memory
- Insufficient memory for new page → Cache collision
  - Insufficient physical memory for page from disk
  - All “ways” in cache occupied for this index
- Need to identify page to be over-written → Victim
  - Page replacement algorithms
  - Cache line replacement algorithms (LRU)
- Need to “page out” pages to make room → Write Back line
- “Dirty” bit to indicate page has been written to → Modified bit
  - No need to write out “clean” pages
  - No need to write back unmodified lines
- Locality and Working Set
  - Avoid “thrashing”

Types of Page Tables

- Flat
  - Large!
  - Proportional to VA space
  - Too big for physical memory
- Hierarchical
- Inverted (hashed)



Intel Pentium: Linear (Virtual) to Physical Address Mapping



Bits
10
10
12
Dir
Page
Offset
(a)
Page directory
Page table
Page frame
Physical address (if present in memory)
Physical address (if present in memory)
1024 Entries
Dir
Page
Offset
Directory entry points to page table
Page table entry points to Page frame

Root level page table “locked down” in memory

Page Table Lookup

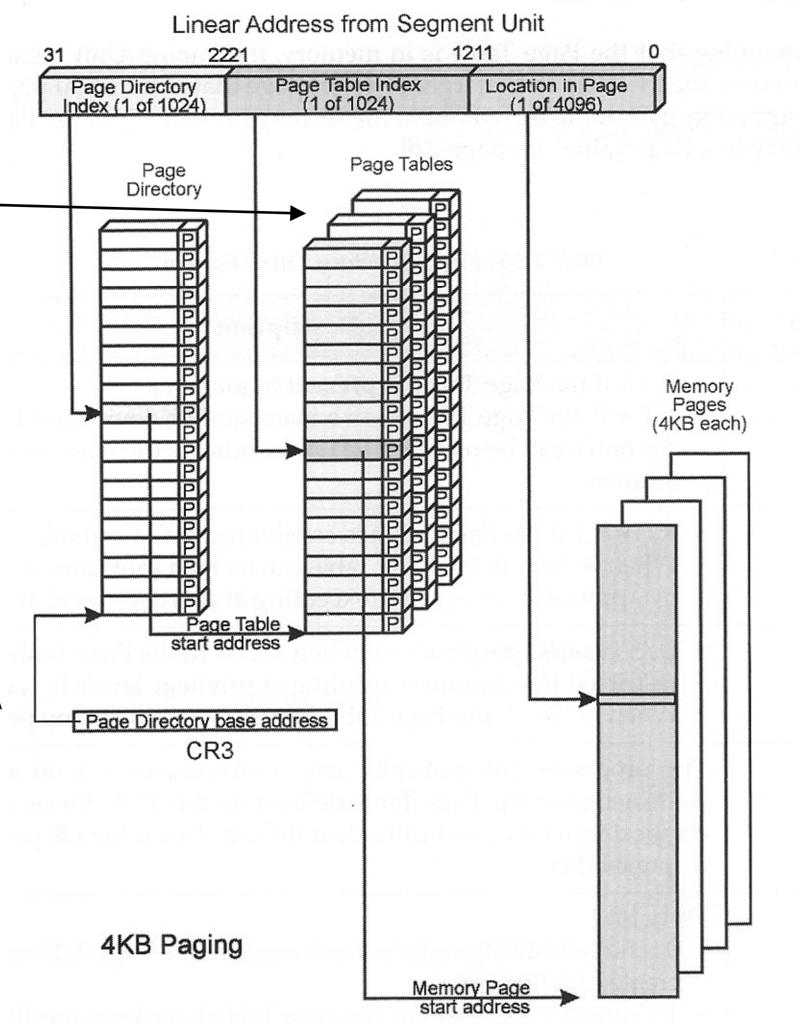
Page Tables can be paged!
Not first-level "root" or "directory" tables

Each process requires its own page tables. Base address loaded on process switch.

1K x 4K = 4M -- page tables
4K -- page directory

Page directory can be "sparse" – so don't require all page tables to be allocated. For example, using 4M virtual memory space requires only 4K (page directory) + 4K (page tables for 1K pages of 4KB each) = 8K.

Multiple memory references/page table lookups for each address (instruction fetch or data reference)


Linear Address from Segment Unit
31
22
21
12
11
0
Page Directory Index (1 of 1024)
Page Table Index (1 of 1024)
Location in Page (1 of 4096)
Page Directory
Page Tables
Memory Pages (4KB each)
P
Page Table start address
Page Directory base address
CR3
4KB Paging
Memory Page start address

Translation Lookaside Buffer (TLB)

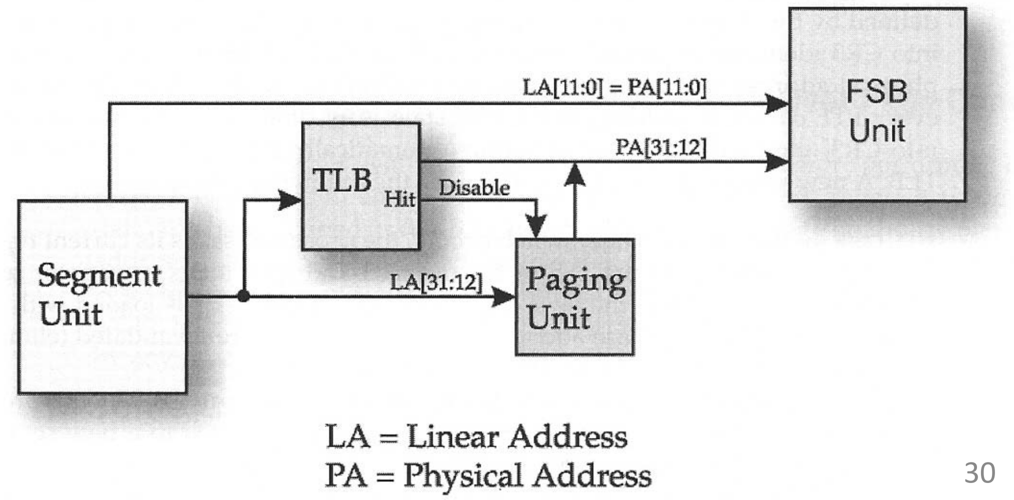
Walking the page table would require multiple memory accesses for each memory reference.

TLB is a cache of most recently used virtual → physical address mappings. Done in parallel with access via paging unit. Similar to… look aside cache!

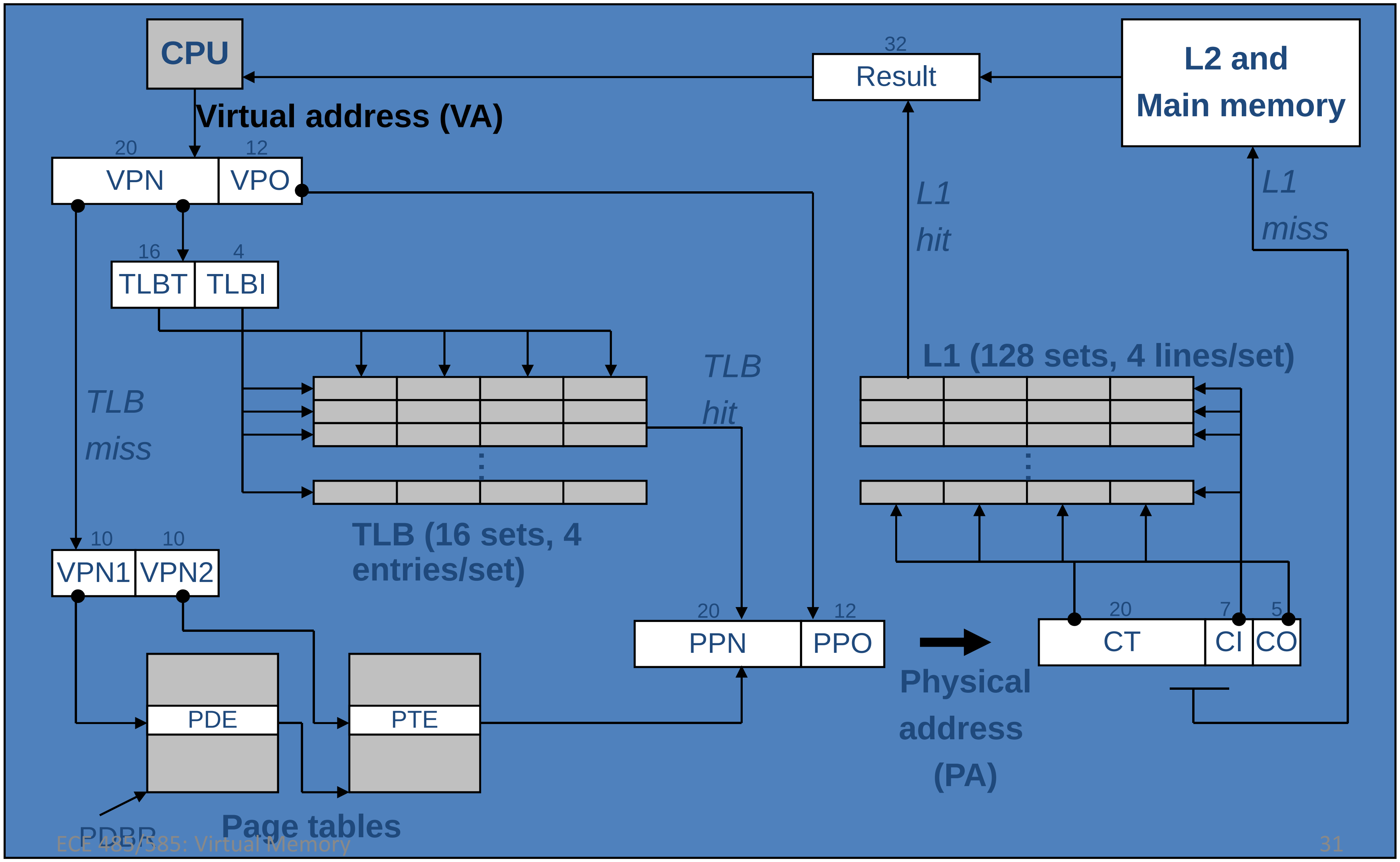
Single TLB shared by all processes, so must flush the TLB on a context switch (each process has own linear → physical address mapping!) [Intel x86: write to CR3 register flushes TLB] or tag entries with process ID (ASID).

Lookup is in hardware, but maintenance (flush on context swap, replacement algorithm on miss, page table walking) can be handled in hardware (e.g. PowerPC, Intel x86) or software (Sparc, MIPS, ARM)

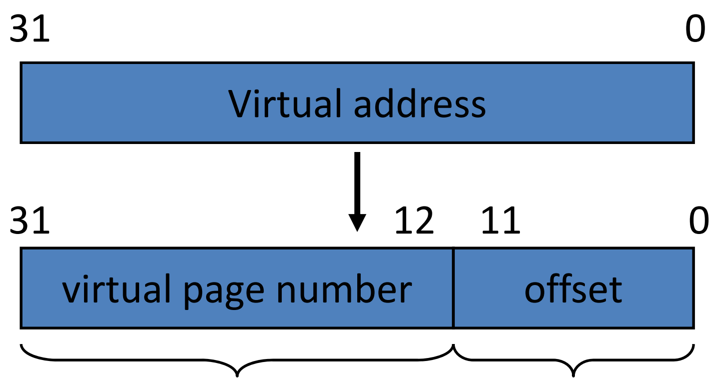
IA-32 terminology:
Linear Address is Virtual Address after translation by segmentation hardware



Pentium Address Translation



TLBs



TLB (cache) lookup [often fully associative]

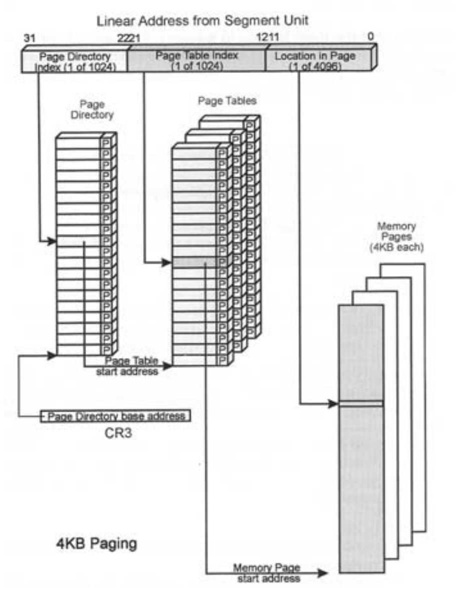
Virtual page number	C	U	D	W	G	OS	P	Page frame address

If entry present in TLB, don't need to walk the page table

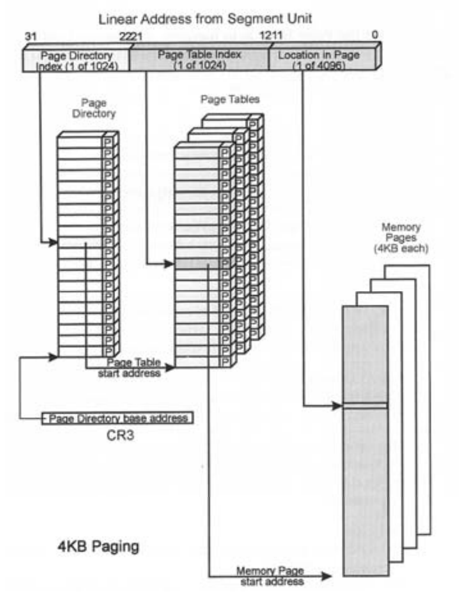
If not present

- Walk the page table
- Evict an entry from TLB
- Insert new mapping

Flush TLB (except for Global pages) whenever process switch!

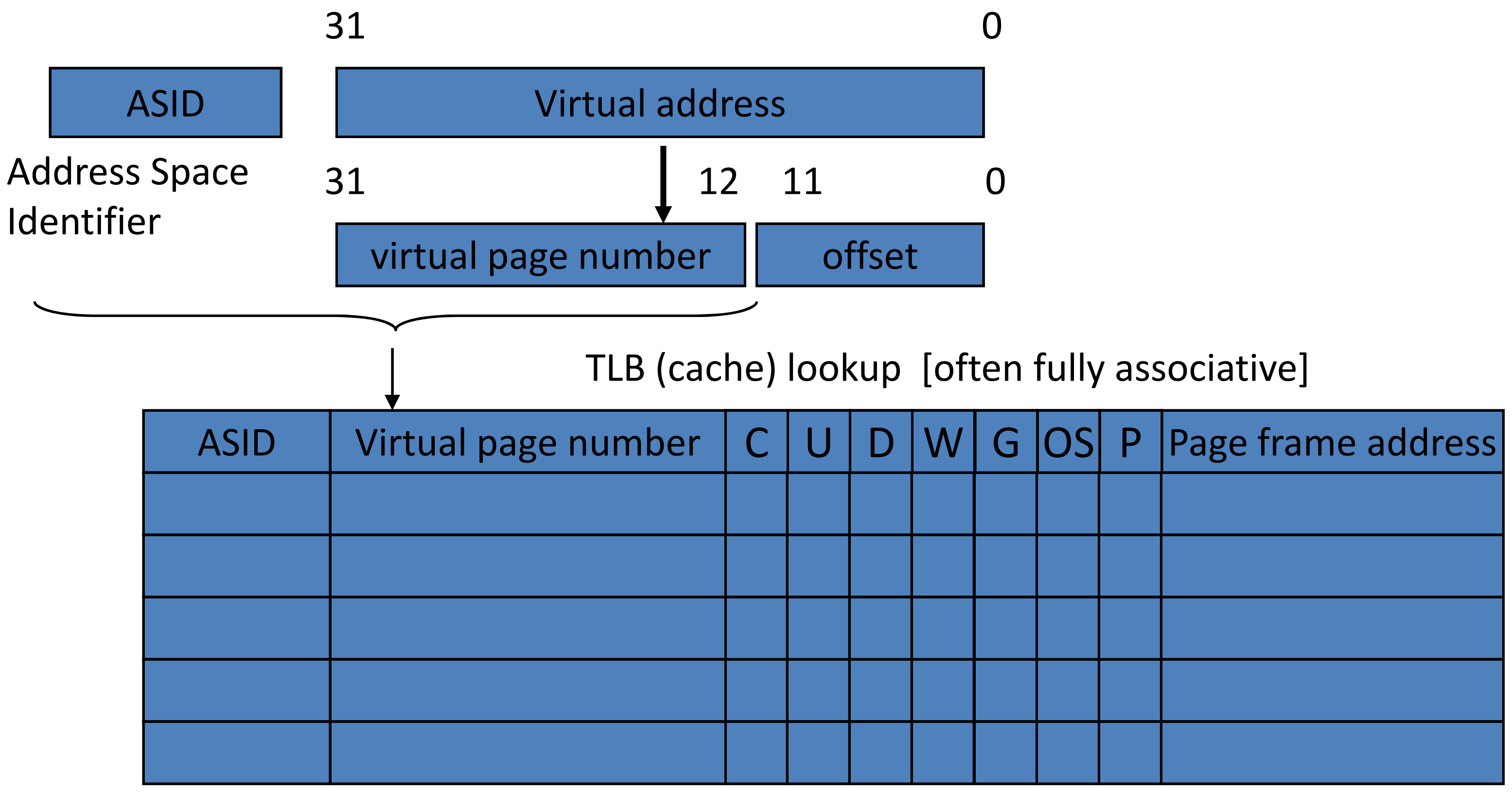

Linear Address from Segment Unit
31
22 21
12 11
0
Page Directory Index (1 of 1024)
Page Table Index (1 of 1024)
Location in Page (1 of 4096)
Page Directory
Page Tables
Memory Pages (4KB each)
Page Table start address
Page Directory base address
CR3
4KB Paging
Memory Page start address

Page Table (Process A)


Linear Address from Segment Unit
31
22 21
12 11
0
Page Directory Index (1 of 1024)
Page Table Index (1 of 1024)
Location in Page (1 of 4096)
Page Directory
Page Tables
Memory Pages (4KB each)
Page Table start address
Page Directory base address
CR3
4KB Paging
Memory Page start address

Page Table (Process B)

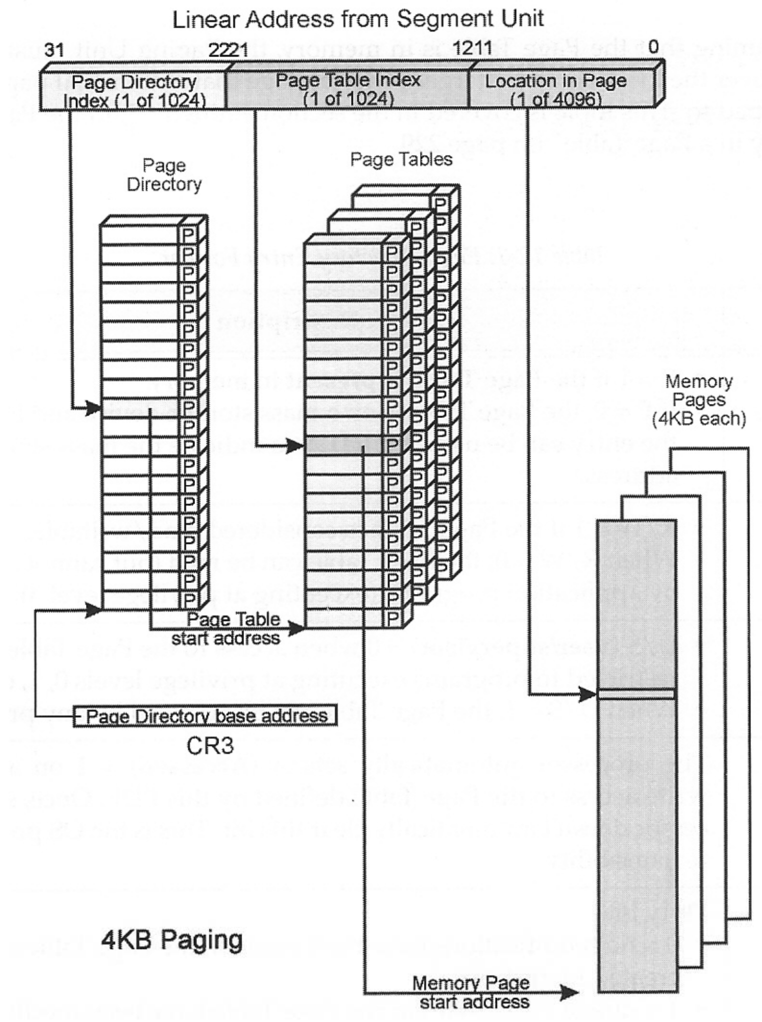
TLBs



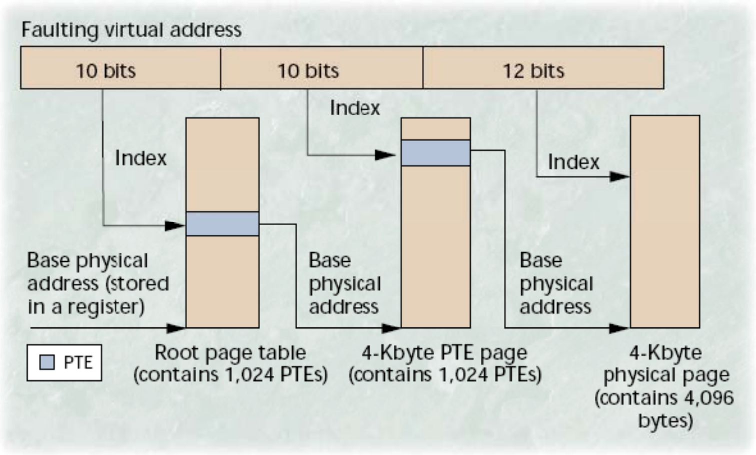
By including ASID in tag we can allow PTEs from different processes to remain in the TLB to improve performance. No longer need to flush TLB on context switch. [Used in MIPS, Alpha, Sparc]. Global bit (G) avoids ASID comparison (OS, shared libraries)

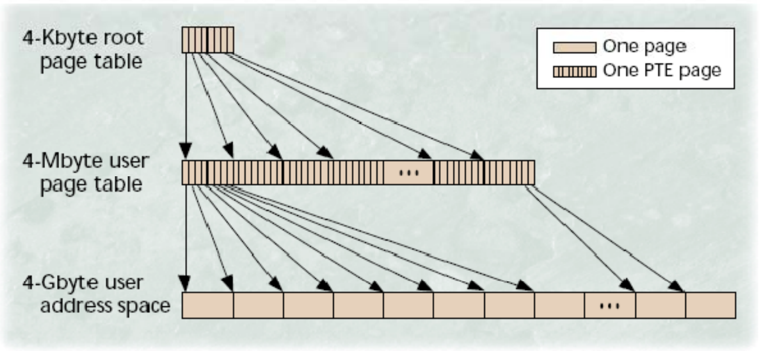
New problem: Number of ASIDs < number of processes…

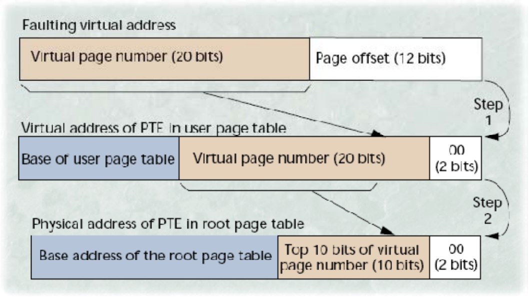
Page Table Walking


Linear Address from Segment Unit
31
22
21
12
11
0
Page Directory Index (1 of 1024)
Page Table Index (1 of 1024)
Location in Page (1 of 4096)
Page Directory
Page Tables
P
Memory Pages (4KB each)
Page Table start address
Page Directory base address
CR3
4KB Paging
Memory Page start address

Top Down vs. Bottom Up Traversal


Faulting virtual address
10 bits
10 bits
12 bits
Index
Index
Index
Base physical address (stored in a register)
Base physical address
Base physical address
PTE
Root page table (contains 1,024 PTEs)
4-Kbyte PTE page (contains 1,024 PTEs)
4-Kbyte physical page (contains 4,096 bytes)




Faulting virtual address
Virtual page number (20 bits)
Page offset (12 bits)
Step 1
Virtual address of PTE in user page table
Base of user page table
Virtual page number (20 bits)
00 (2 bits)
Step 2
Physical address of PTE in root page table
Base address of the root page table
Top 10 bits of virtual page number (10 bits)
00 (2 bits)

Top down traversal requires multiple memory references to obtain physical address.

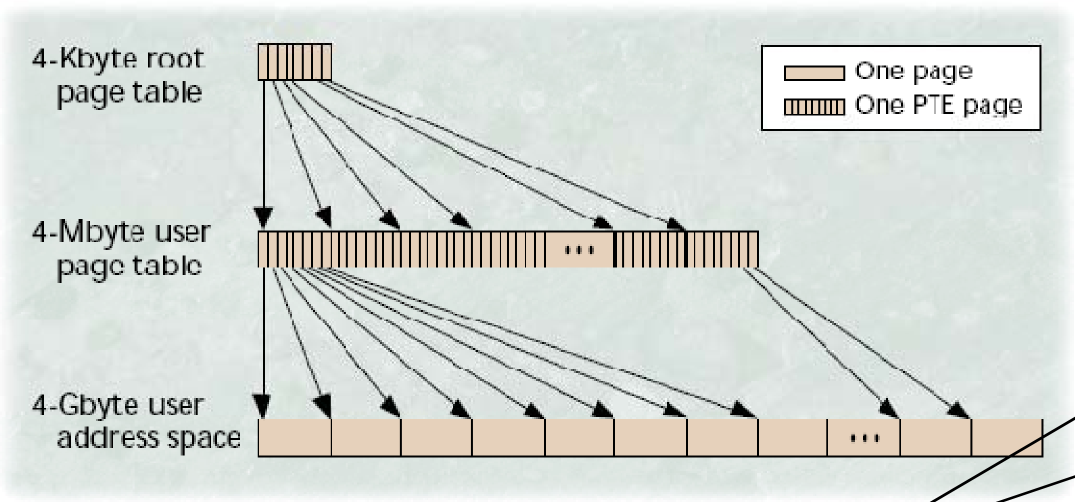
If second level page table is stored in contiguous virtual memory addresses, a short cut is possible…

The index into the second level page table is the same as the virtual page number of the page it maps. We can create address from base of user page table and virtual page number and check TLB for this translation.

If found, load the PTE into the TLB. If not, obtain the physical address of the PTE entry in the first level page table for the second level page table's location

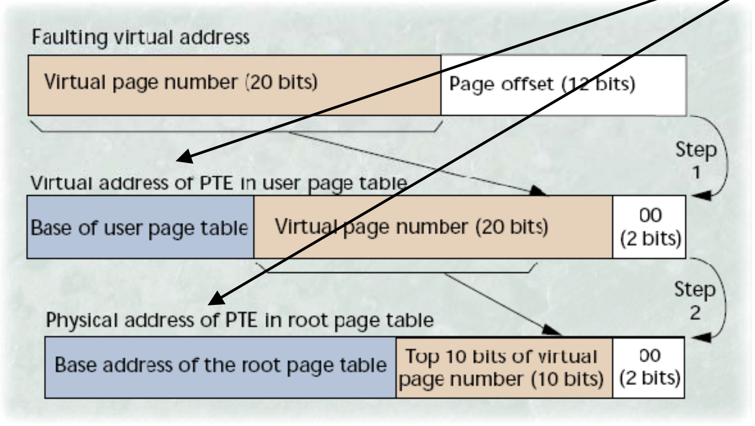
First used commercially in MIPS, later in Alpha

Top Down vs. Bottom Up Traversal



Physical address is $0x\underbrace{nnnnn}_{\text{20 bits}}000$ (4K)

Virtual address is $0x\underbrace{nnC}_{\text{10 bits}}00000$ (4M)


Faulting virtual address
Virtual page number (20 bits)
Page offset (12 bits)
Step 1
Virtual address of PTE in user page table
Base of user page table
Virtual page number (20 bits)
00 (2 bits)
Step 2
Physical address of PTE in root page table
Base address of the root page table
Top 10 bits of virtual page number (10 bits)
00 (2 bits)

(VPN, PPN) not in TLB

See if VPN of PTE is in TLB (if yes, use it to get physical address of page frame). LSBs = 00 because PTE is 4 bytes long.

No, get physical address of PTE from root page table

Note: page containing PTE and user page can both still result in page faults if not resident in memory. This process just loads the TLB for the VPN to PPN translation. The page may be non-resident in memory.

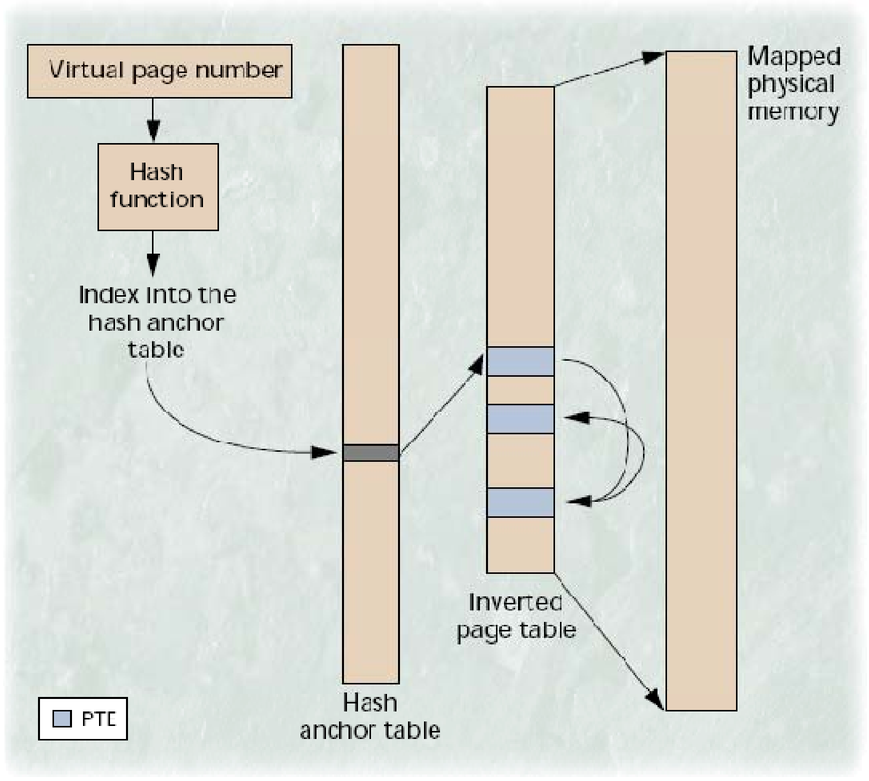
Inverted Page Tables

Alternative: inverted page tables proportional to size of physical memory instead of virtual address space. [PowerPC, UltraSparc]

Hash anchor table not always used (omitted in PowerPC, PA-RISC) because it necessitates additional memory reference.

Downside: only maps pages which are currently resident in memory. Need another structure to map pages which are on disk so they can be paged in.

Can't accommodate different virtual addresses mapping to same physical address at same time. If memory is used for inter-process communication, may incur two page faults for each message.

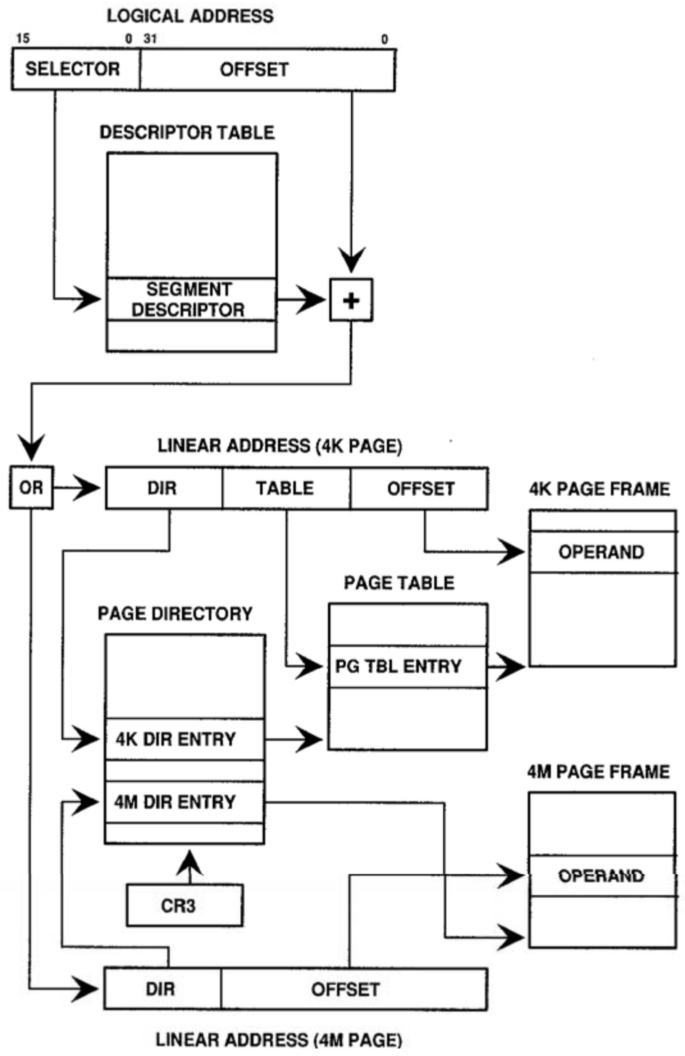


Segmentation and Paging

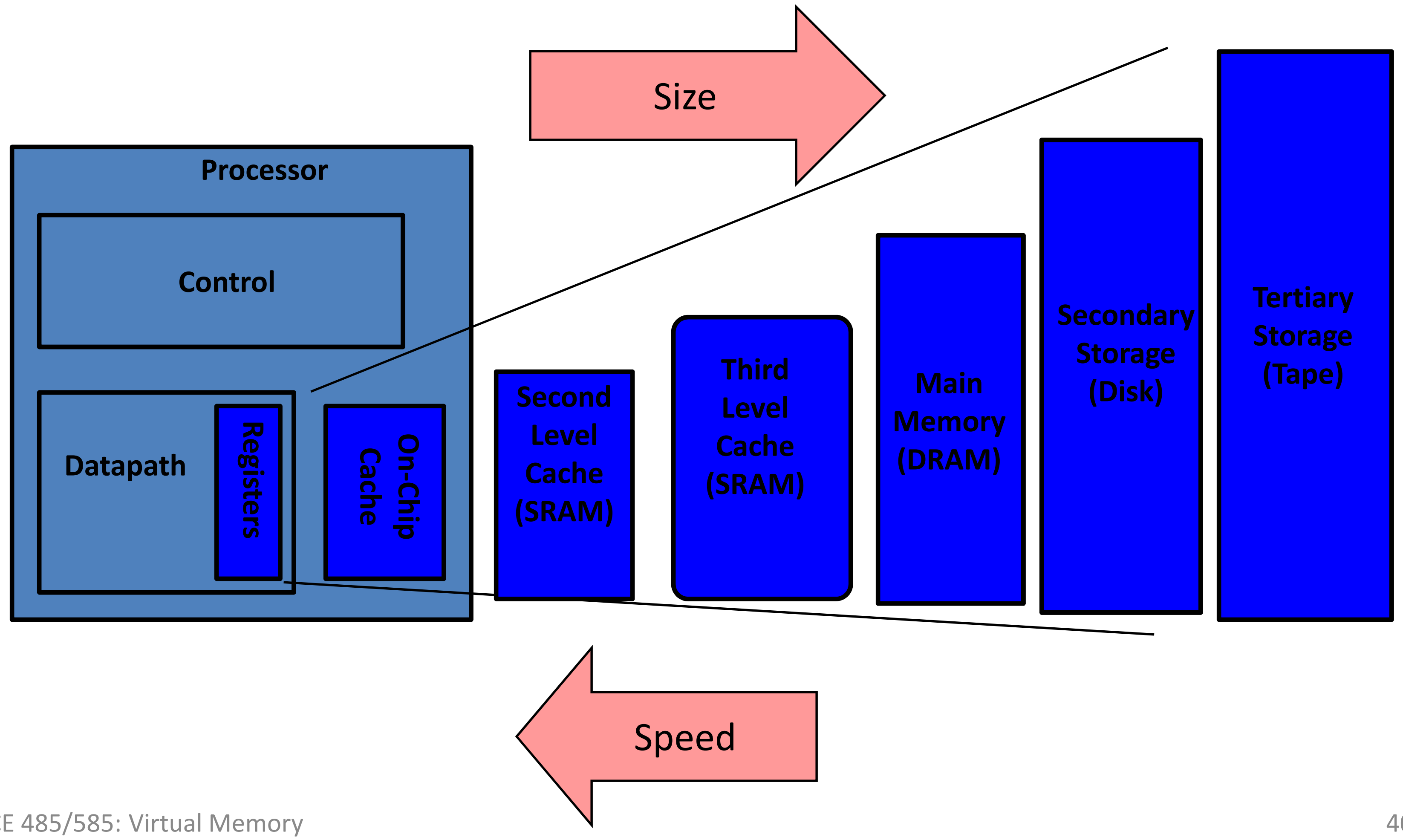
- Segmentation designed to provide very large virtual address space and protection
- Paging was designed to support efficient swapping between physical memory and disk
- However, paging provides adequate protection so virtually all modern operating systems (e.g. Windows NT, Unix) put all 32-bit code in single 4 GB segment. This is called the “flat address” model.
- Cannot disable segmentation on Pentium
- Effectively disable it by having single segment pointed to by all segment registers
- Use paging for everything else

Combining Paging and Segmentation

Intel IA-32
(also PowerPC)



Memory Hierarchy of a Modern Computer System



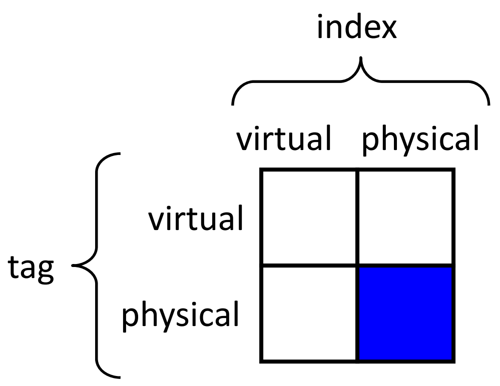
Intel Pentium 4 3.2 GHz Server

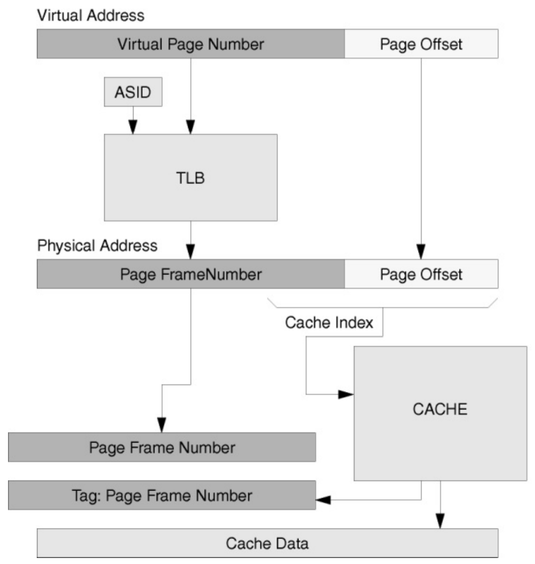
Component	Access Speed (Time for data to be returned)
Registers	1 cycle = 0.3 nanoseconds
L1 Cache	3 cycles = 1 nanoseconds
L2 Cache	20 cycles = 7 nanoseconds
L3 Cache	40 cycles = 13 nanoseconds
Memory	300 cycles = 100 nanoseconds
Disk	15,000,000 cycles = 5,000,000 nanoseconds

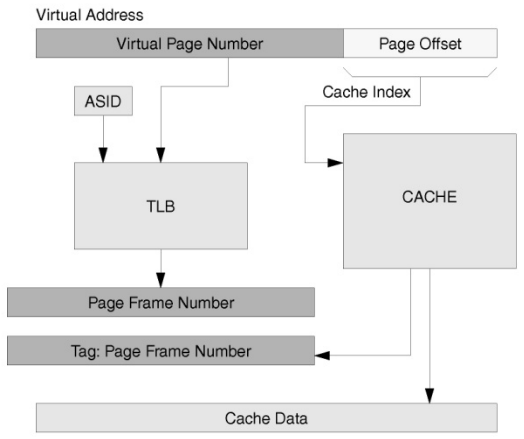
Caches and Virtual Memory

Parameter	L1 Cache	Virtual Memory
Block (page) size	16-128 bytes	4096-65,536 bytes
Hit time	1-3 clock cycles	50-150 clock cycles
Miss penalty	8-150 clock cycles	10^6-10^7 clock cycles
Access time	6-130 clock cycles	$8x10^5$-$8x10^6$ clock cycles
Transfer time	2-20 clock cycles	$2x10^5$-$2x10^6$ clock cycles
Miss rate	0.1-10%	0.00001 – 0.001%
Address mapping	24-25 bit physical to 14-20 bit cache (index)	32-64 bit virtual to 25-45 bit physical

Paging and Caching







Advantage

Physically tagged → no synonym problem

Disadvantage

Virtual address must be translated to physical

address first

Must wait for TLB translation

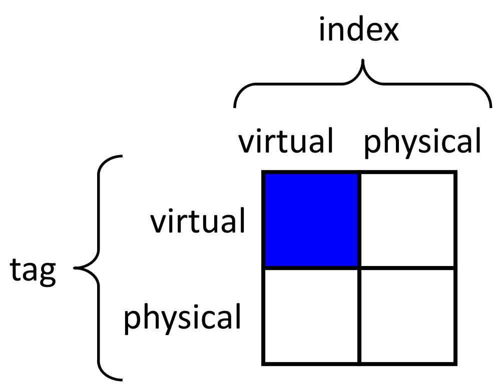
(or page table walk if miss)

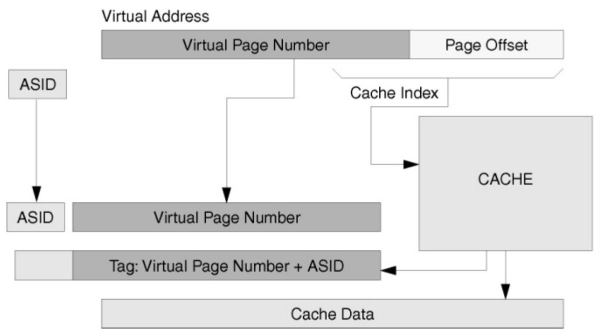
Solution

Size cache so that virtual page offset is same size as cache line index

Cache and TLB lookup in parallel

Paging and Caching





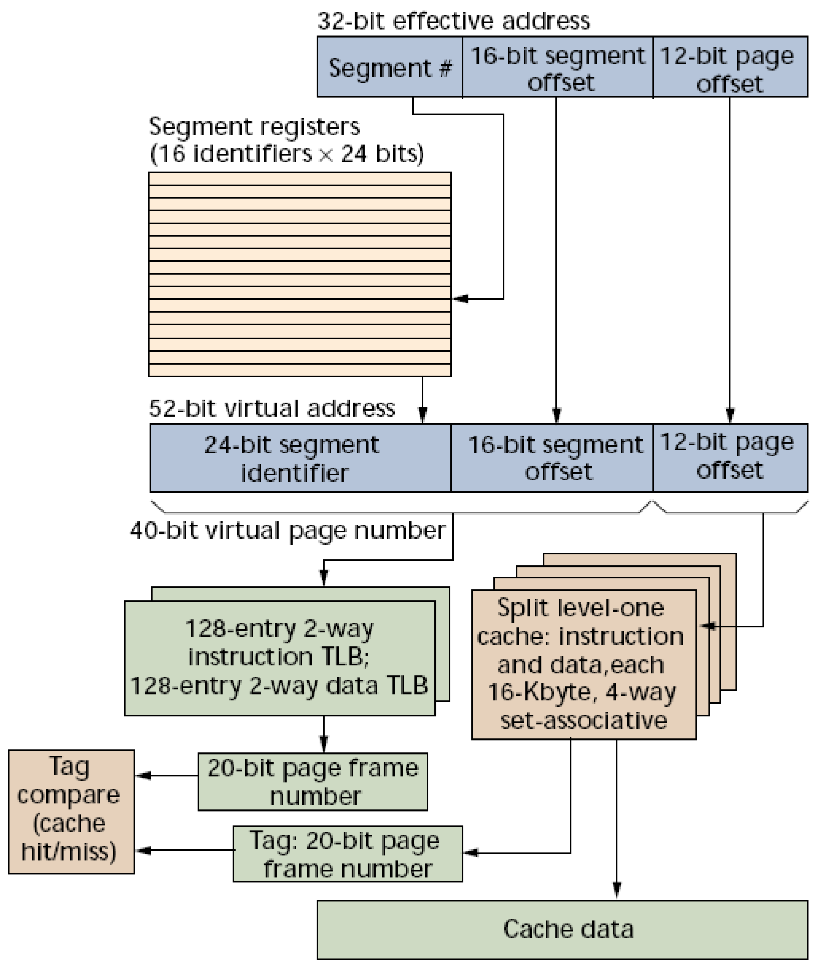
Advantage
- No TLB required
  - Can use on cache miss
  - Must store ASID in cache
    - Different processes can use same <u>virtual</u> address for different physical locations

Disadvantage
- Aliasing
  - Different physical addresses can map to same physical address at <u>different</u> cache locations!

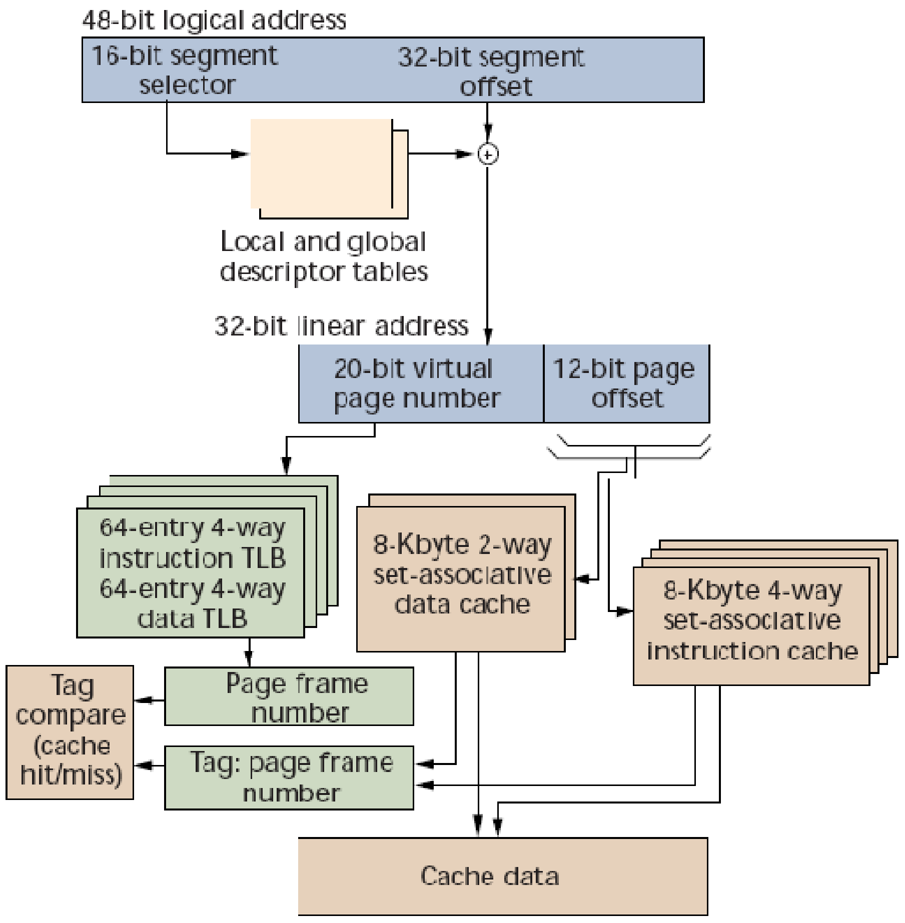
Solution
- Direct mapped cache only

Note: Still need some place to record physical address of cached on backing store for WB

Some Examples

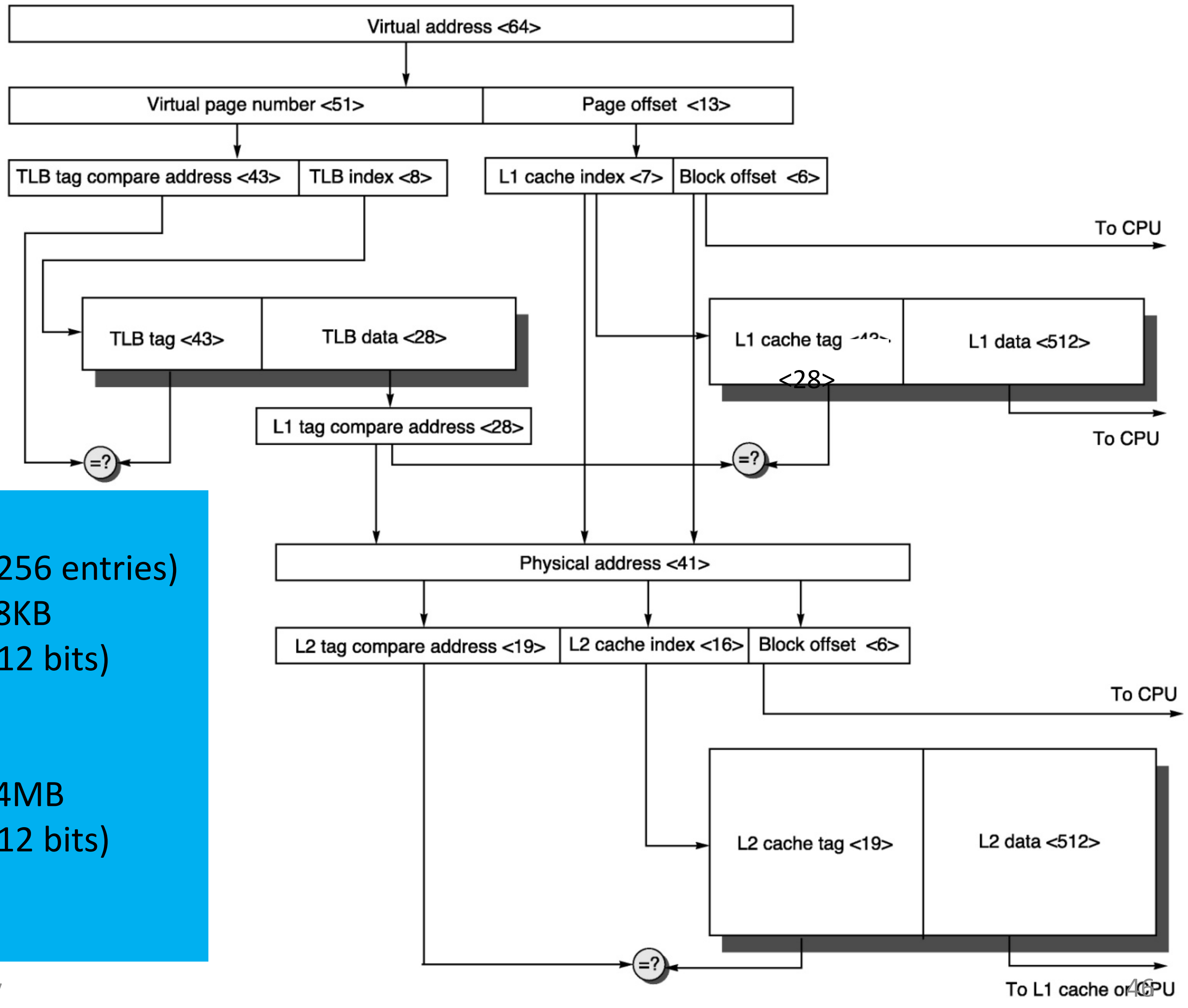

32-bit effective address
Segment #
16-bit segment offset
12-bit page offset
Segment registers (16 identifiers × 24 bits)
52-bit virtual address
24-bit segment identifier
16-bit segment offset
12-bit page offset
40-bit virtual page number
128-entry 2-way instruction TLB; 128-entry 2-way data TLB
Split level-one cache: instruction and data,each 16-Kbyte, 4-way set-associative
Tag compare (cache hit/miss)
20-bit page frame number
Tag: 20-bit page frame number
Cache data

PowerPC 604


48-bit logical address
16-bit segment selector
32-bit segment offset
Local and global descriptor tables
32-bit linear address
20-bit virtual page number
12-bit page offset
64-entry 4-way instruction TLB 64-entry 4-way data TLB
8-Kbyte 2-way set-associative data cache
8-Kbyte 4-way set-associative instruction cache
Tag compare (cache hit/miss)
Page frame number
Tag: page frame number
Cache data

X86

Paging and Caching



8K page size

TLB direct-mapped (2^8 = 256 entries)

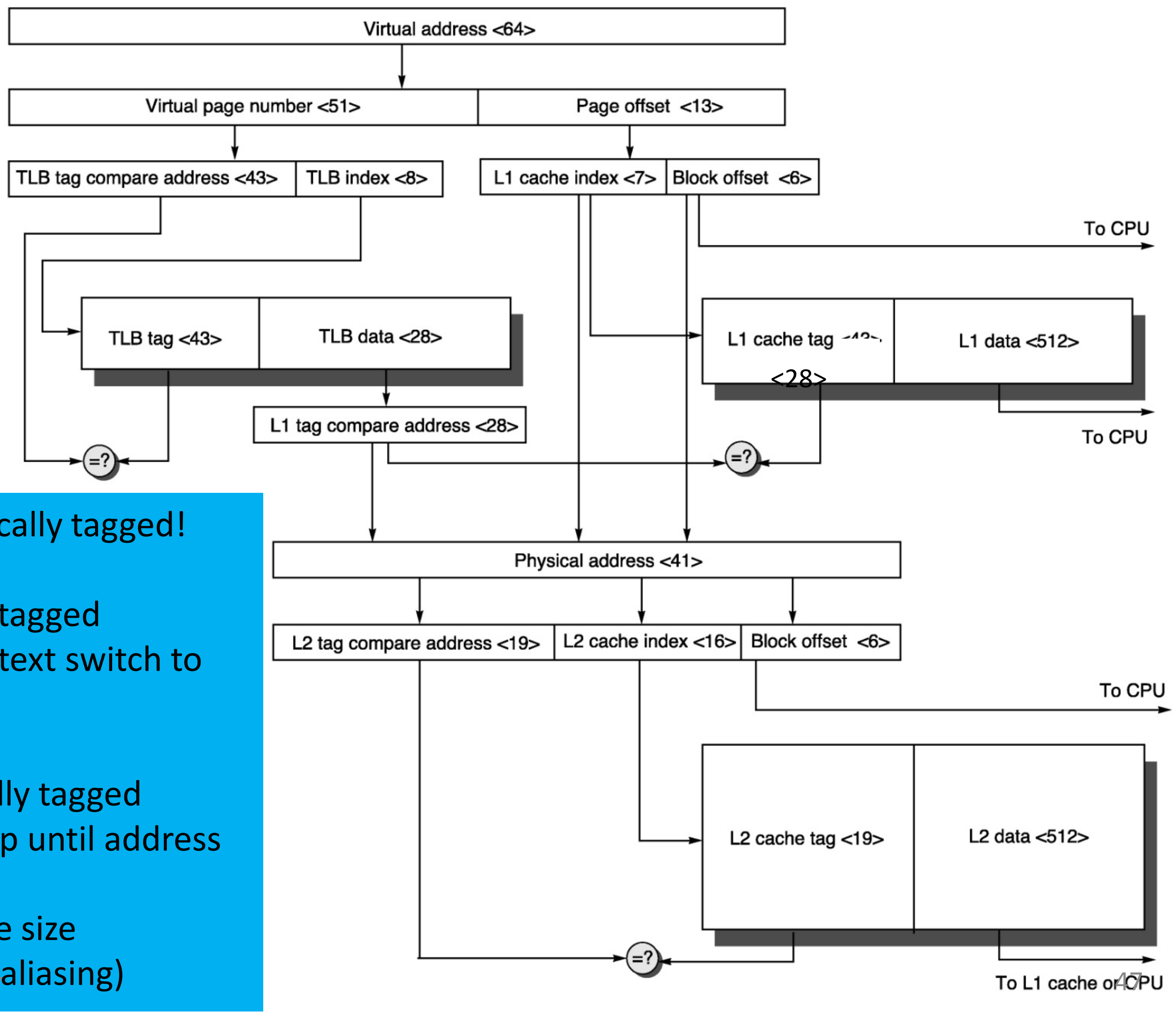
L1 cache direct-mapped 8KB

- line size 64 (2^6) bytes (512 bits)
- 7 bit index
- $2^7 \times 2^6 = 2^{13}$ = 8K

L2 cache direct-mapped 4MB

- line size 64 (2^6) bytes (512 bits)
- 16 bit index
- $2^{16} \times 2^6 = 2^{22}$ = 4M

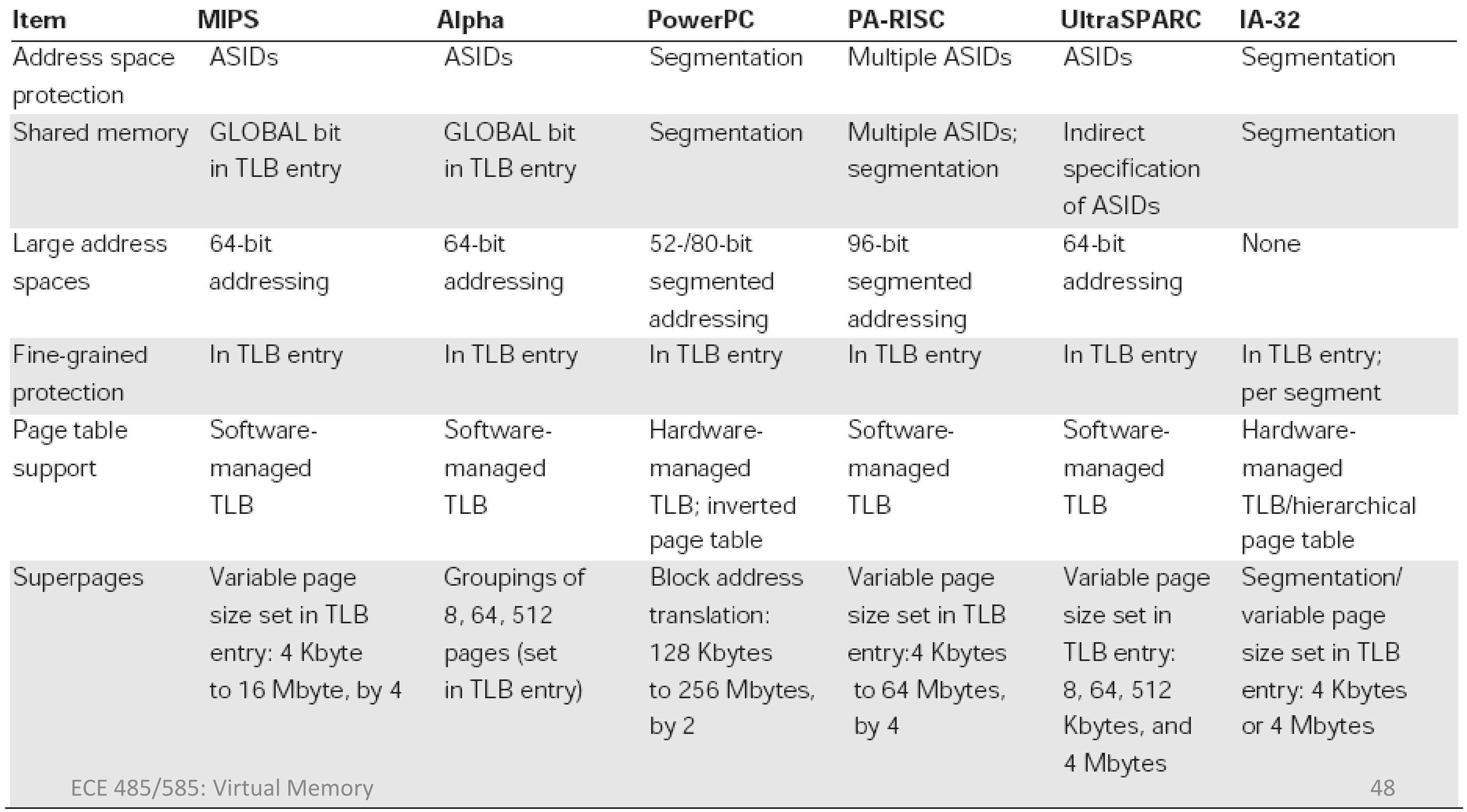
Paging and Caching



Virtually indexed but physically tagged!

Virtually indexed, virtually tagged
→Must flush cache on context switch to avoid aliasing!
→No need for TLB!
Physically indexed, physically tagged
→ Can't begin cache lookup until address translation complete
Simple if cache size == page size
(otherwise must deal with aliasing)

Architectural Support for VM



Item	MIPS	Alpha	PowerPC	PA-RISC	UltraSPARC	IA-32
Address space protection	ASIDs	ASIDs	Segmentation	Multiple ASIDs	ASIDs	Segmentation
Shared memory	GLOBAL bit in TLB entry	GLOBAL bit in TLB entry	Segmentation	Multiple ASIDs; segmentation	Indirect specification of ASIDs	Segmentation
Large address spaces	64-bit addressing	64-bit addressing	52-/80-bit segmented addressing	96-bit segmented addressing	64-bit addressing	None
Fine-grained protection	In TLB entry	In TLB entry	In TLB entry	In TLB entry	In TLB entry	In TLB entry; per segment
Page table support	Software-managed TLB	Software-managed TLB	Hardware-managed TLB; inverted page table	Software-managed TLB	Software-managed TLB	Hardware-managed TLB/hierarchical page table
Superpages	Variable page size set in TLB entry: 4 Kbyte to 16 Mbyte, by 4	Groupings of 8, 64, 512 pages (set in TLB entry)	Block address translation: 128 Kbytes to 256 Mbytes, by 2	Variable page size set in TLB entry:4 Kbytes to 64 Mbytes, by 4	Variable page size set in TLB entry: 8, 64, 512 Kbytes, and 4 Mbytes	Segmentation/ variable page size set in TLB entry: 4 Kbytes or 4 Mbytes

Intel Nehalem and AMD Opteron

Characteristic	Intel Nehalem	AMD Opteron X4 (Barcelona)
Virtual address	48 bits	48 bits
Physical address	44 bits	48 bits
Page size	4 KB, 2/4 MB	4 KB, 2/4 MB
TLB organization	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement The L2 TLB is 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 has 512 entries TLB misses handled in hardware	1 L1 TLB for instructions and 1 L1 TLB for data per core Both L1 TLBs fully associative, LRU replacement 1 L2 TLB for instructions and 1 L2 TLB for data per core Both L2 TLBs are four-way set associative, round-robin Both L1 TLBs have 48 entries Both L2 TLBs have 512 entries TLB misses handled in hardware

AMD Opteron
- 48-bit virtual address
- 40-bit physical address

- L1 TLB
  - 40 entries, fully associative
- L2 TLB
  - 4-way, 512 entries, $\log_2(512/4) = 7$

- L1 data, instruction caches (separate)
  - 64KB, 2-way, LRU replacement
  - 64-byte cache line
  - $\log_2(64KB/(64 \times 2)) = 9$

- L2 unified cache
  - 1M, 16-way, 64-byte cache line
  - pseudo-LRU replacement

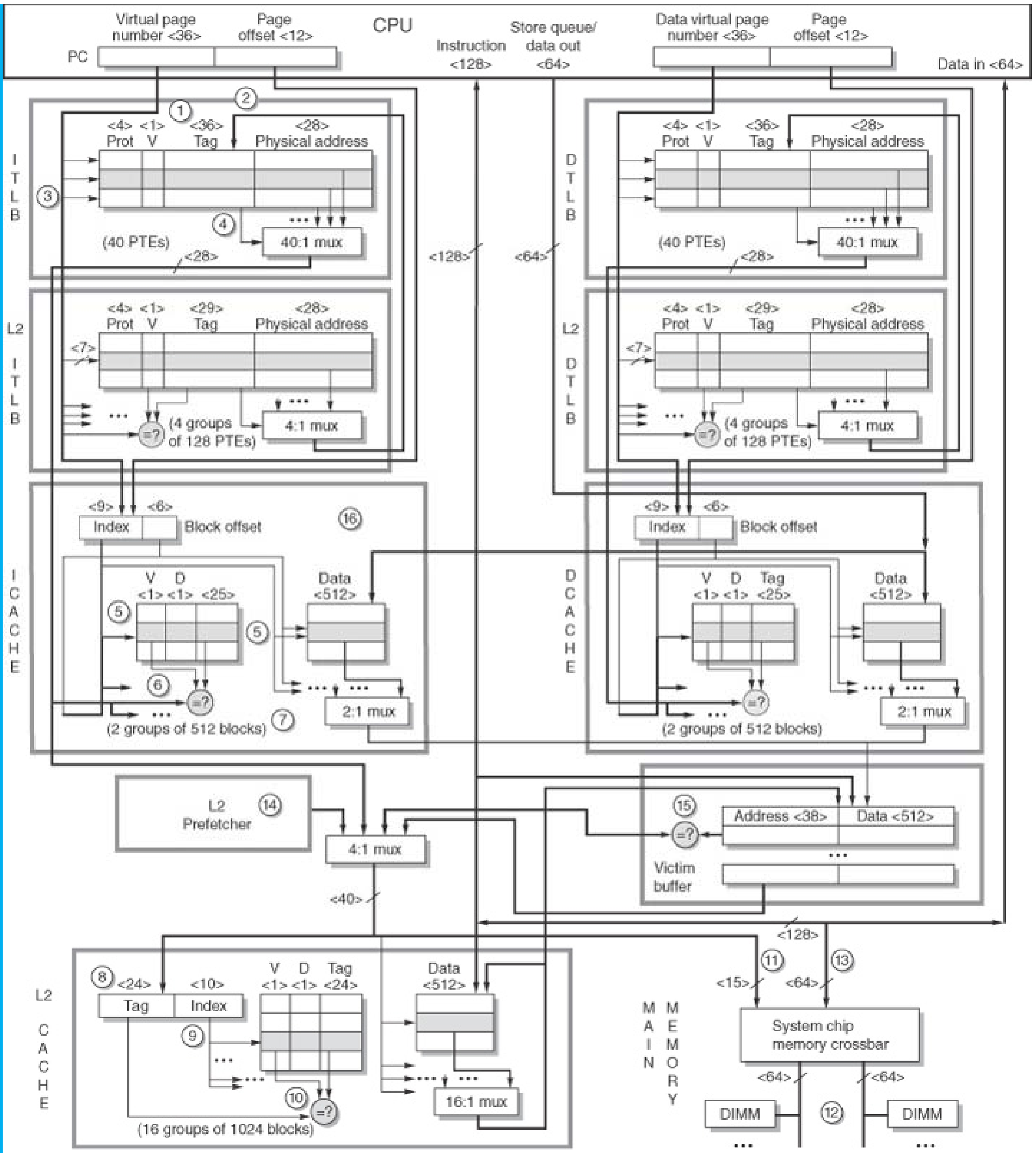
Victim buffer
- 8 entries

L2 prefetcher
- based on patterns of misses

Memory controller
- 8 simultaneous data cache misses
- 2 simultaneous instruction misses
- (hit under 10 miss)
- 2x 64-bit (+ECC) channels

Physically indexed, virtually tagged

Page size not equal to cache size!
- potential for aliasing – H/W checks!



Benefits of Virtual Memory

- Programs have illusion of entire potential address space
  - All programs can start at address (e.g. 0) without being remapped by loader
- Don’t need contiguous physical memory
  - Programs can co-exist, be swapped in/out with small granularity (no holes)
- Facilitates dynamic storage allocation (e.g. malloc)
  - Simply request more memory, pages allocated as needed
  - Physical memory required when used (“demand paged”)
  - No need to copy/relocate code or data
  - Don’t have issues of stacks growing into each other
- Protection
  - Hardware assisted range checking via page fault mechanism
    - Permissions on per page basis for sharing
    - Can detect referencing null pointer
    - Prevents accidental or malicious attempts to access memory not owned by process