15-213

Virtual Memory
March 23, 2000

Topics

- Motivations for VM
- Address translation
- Accelerating address translation with TLBs
- Pentium II/III memory system

Motivation #1: DRAM a "Cache" for Disk

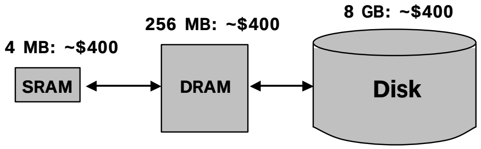
The full address space is quite large:

- 32-bit addresses: ~4,000,000,000 (4 billion) bytes
- 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes

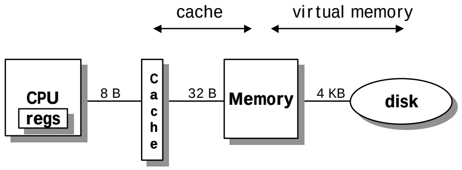
Disk storage is ~30X cheaper than DRAM storage

- 8 GB of DRAM: ~$12,000
- 8 GB of disk: ~$400

To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk



Levels in Memory Hierarchy



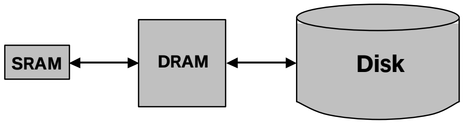
	Register	Cache	Memory	Disk Memory
size:	32 B	32 KB-4MB	128 MB	20 GB
speed:	2 ns	4 ns	60 ns	8 ms
$/Mbyte:		$100/MB	$1.50/MB	$0.05/MB
line size:	8 B	32 B	4 KB	

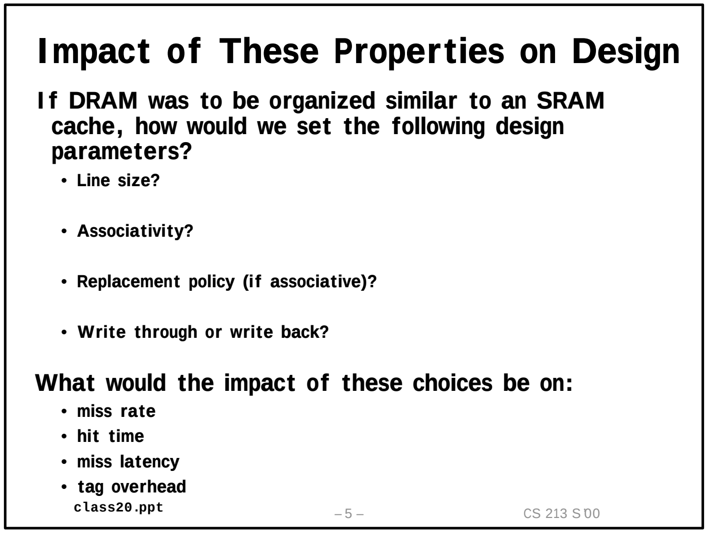
larger, slower, cheaper →

DRAM vs. SRAM as a "Cache"

DRAM vs. disk is more extreme than SRAM vs. DRAM

- **access latencies:**
  - DRAM is ~10X slower than SRAM
  - disk is ~**100,000X** slower than DRAM
- **importance of exploiting spatial locality:**
  - first byte is ~**100,000X** slower than successive bytes on disk
    - » vs. ~4X improvement for page-mode vs. regular accesses to DRAM
- **Bottom line:**
  - design of DRAM caches driven by enormous cost of misses





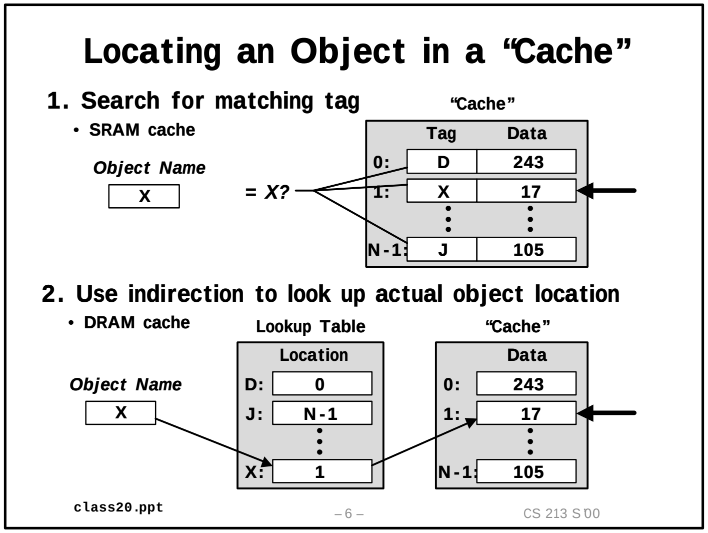
Impact of These Properties on Design

If DRAM was to be organized similar to an SRAM cache, how would we set the following design parameters?

- Line size?
- Associativity?
- Replacement policy (if associative)?
- Write through or write back?

What would the impact of these choices be on:

- miss rate
- hit time
- miss latency
- tag overhead



Locating an Object in a "Cache"

1. Search for matching tag

- SRAM cache

Object Name: X = *X?*

"Cache"

	Tag	Data
0:	D	243
1:	X	17
N-1:	J	105

2. Use indirection to look up actual object location

- DRAM cache

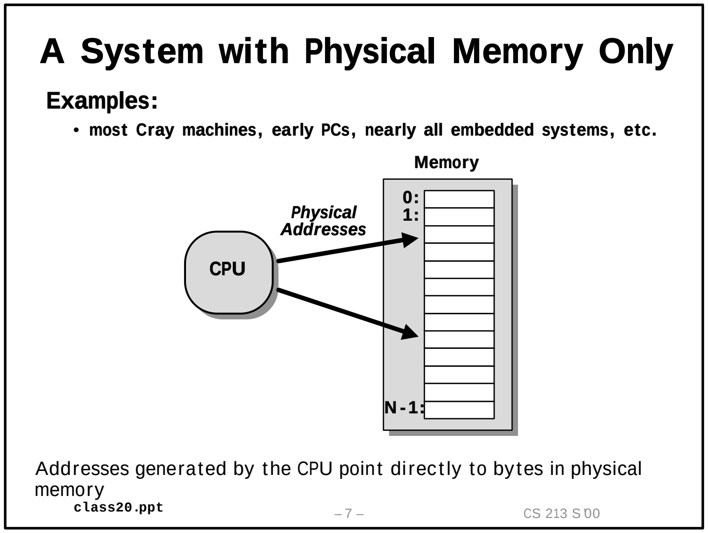
Object Name: X

Lookup Table

	Location
D:	0
J:	N-1
X:	1

"Cache"

	Data
0:	243
1:	17
N-1:	105

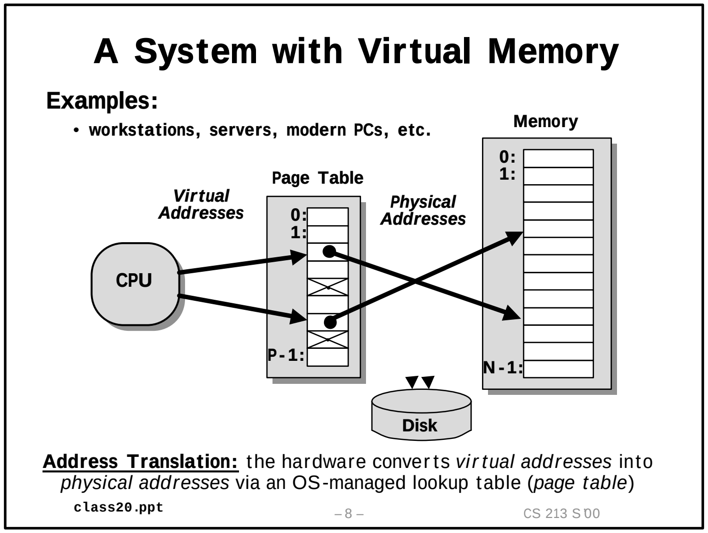


A System with Physical Memory Only

Examples:

- most Cray machines, early PCs, nearly all embedded systems, etc.

Addresses generated by the CPU point directly to bytes in physical memory



A System with Virtual Memory

Examples:

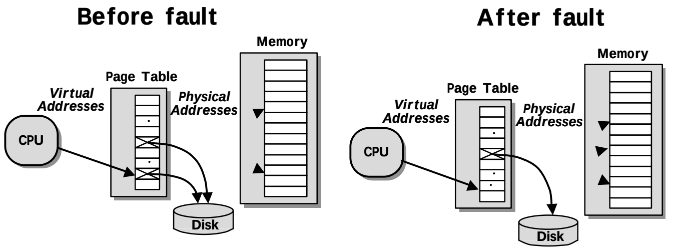
- workstations, servers, modern PCs, etc.

Address Translation: the hardware converts *virtual addresses* into *physical addresses* via an OS-managed lookup table (*page table*)

Page Faults (Similar to "Cache Misses")

What if an object is on disk rather than in memory?

- Page table entry indicates that the virtual address is not in memory
- An OS exception handler is invoked, moving data from disk into memory
  - current process suspends, others can resume
  - OS has full control over placement, etc.



Servicing a Page Fault

Processor Signals Controller

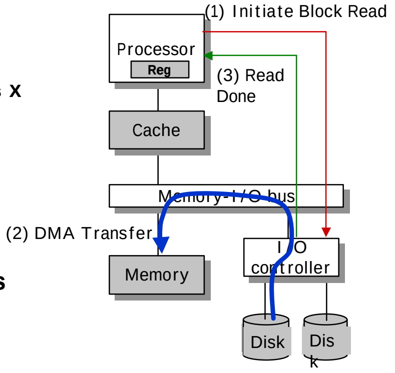
- Read block of length P starting at disk address X and store starting at memory address Y

Read Occurs

- Direct Memory Access (DMA)
- Under control of I/O controller

I/O Controller Signals Completion

- Interrupt processor
- OS resumes suspended process



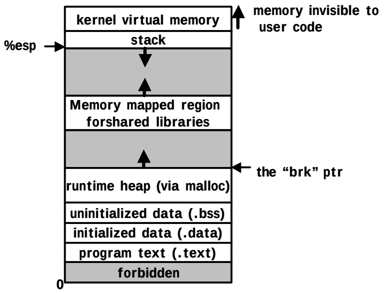
Motivation #2: Memory Management

Multiple processes can reside in physical memory.

How do we resolve address conflicts?

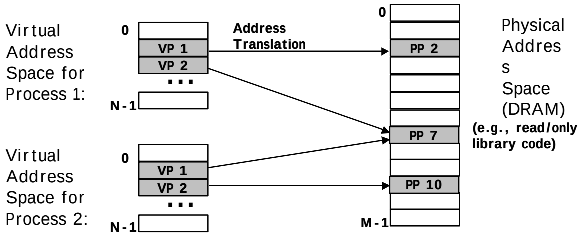
- what if two processes access something at the same address?

Linux/x86 process memory image


kernel virtual memory
memory invisible to user code
stack
%esp
Memory mapped region forshared libraries
the "brk" ptr
runtime heap (via malloc)
uninitialized data (.bss)
initialized data (.data)
program text (.text)
forbidden
0

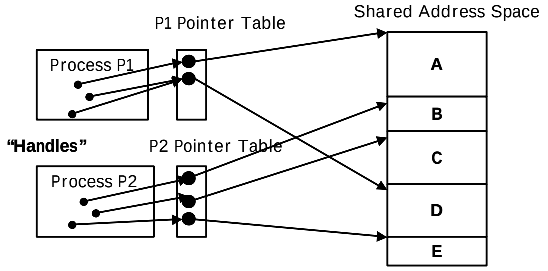
Solution: Separate Virtual Addr. Spaces

- Virtual and physical address spaces divided into equal-sized blocks
  - blocks are called "pages" (both virtual and physical)
- Each process has its own virtual address space
  - operating system controls how virtual pages as assigned to physical memory


Virtual Address Space for Process 1:
0
VP 1
VP 2
...
N-1
Address Translation
0
PP 2
PP 7
PP 10
M-1
Physical Addres s Space (DRAM)
(e.g., read/only library code)
Virtual Address Space for Process 2:
0
VP 1
VP 2
...
N-1

Contrast: (Old) Macintosh Memory Model

Does not use traditional virtual memory



All program objects accessed through "handles"

- indirect reference through pointer table
- objects stored in shared global address space

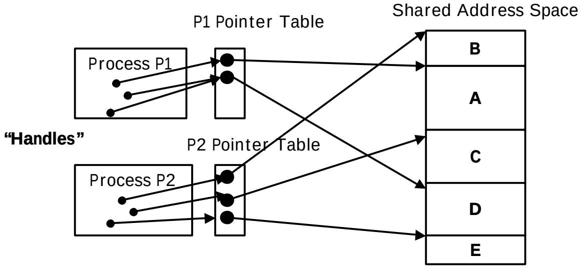
(Old) Macintosh Memory Management

Allocation / Deallocation

- Similar to free-list management of malloc/free

Compaction

- Can move any object and just update the (unique) pointer in pointer table



(Old) Mac vs. VM-Based Memory Mgmt

Allocating, deallocating, and moving memory:

- can be accomplished by both techniques

Block sizes:

- Mac: variable-sized
  - may be very small or very large
- VM: fixed-size
  - size is equal to *one page* (4KB on x86 Linux systems)

Allocating contiguous chunks of memory:

- Mac: contiguous allocation is ***required***
- VM: can map contiguous range of virtual addresses to disjoint ranges of physical addresses

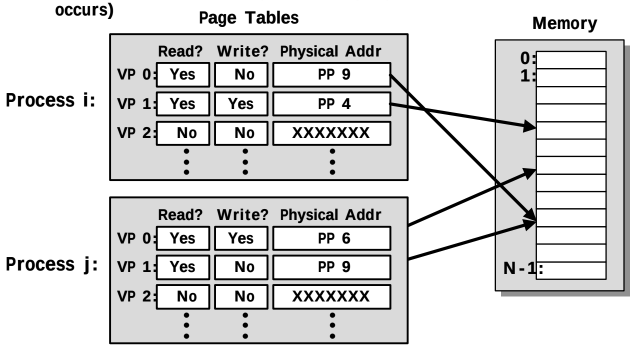
Protection?

- Mac: "wild write" by one process can corrupt another's data

Motivation #3: Protection

Page table entry contains access rights information

- hardware enforces this protection (trap into OS if violation occurs)



Process i:

	Read?	Write?	Physical Addr
VP 0:	Yes	No	PP 9
VP 1:	Yes	Yes	PP 4
VP 2:	No	No	XXXXXXX
	•	•	•

Process j:

	Read?	Write?	Physical Addr
VP 0:	Yes	Yes	PP 6
VP 1:	Yes	No	PP 9
VP 2:	No	No	XXXXXXX
	•	•	•

Summary: Motivations for VM

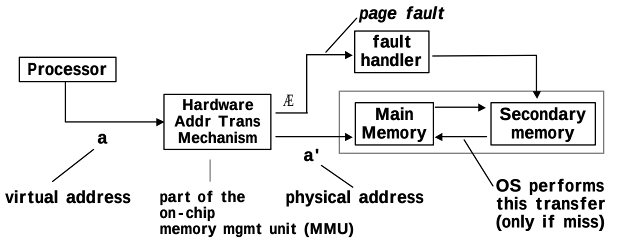
- Uses physical DRAM memory as a cache for the disk
  - address space of a process can exceed physical memory size
  - sum of address spaces of multiple processes can exceed physical memory
- Simplifies memory management
  - Can have multiple processes resident in main memory.
  - Each process has its own address space (0, 1, 2, 3, …, n-1)
  - Only "active" code and data is actually in memory
  - Can easily allocate more memory to process as needed.
    - – external fragmentation problem nonexistent
- Provides protection
  - One process can't interfere with another.
    - – because they operate in different address spaces.
  - User process cannot access privileged information
    - – different sections of address spaces have different permissions.

VM Address Translation

V = {0, 1, . . . , N–1} virtual address space
P = {0, 1, . . . , M–1} physical address space N > M

MAP: V ® P U {Æ} address mapping function

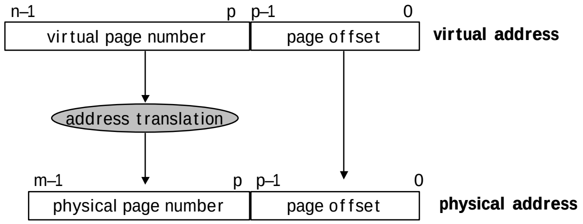
MAP(a) = a' if data at virtual address <u>a</u> is present at physical address <u>a'</u> in P
= Æ if data at virtual address a is not present in P


page fault
fault handler
Processor
Hardware Addr Trans Mechanism
Æ
Main Memory
Secondary memory
a
a'
virtual address
part of the on-chip memory mgmt unit (MMU)
physical address
OS performs this transfer (only if miss)

VM Address Translation

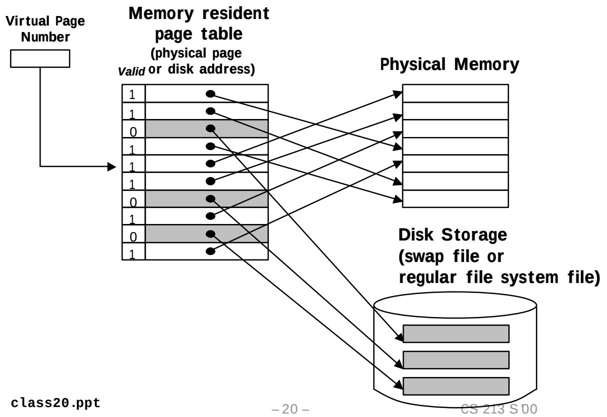
Parameters

- P = 2^p = page size (bytes).
- N = 2^n = Virtual address limit
- M = 2^m = Physical address limit

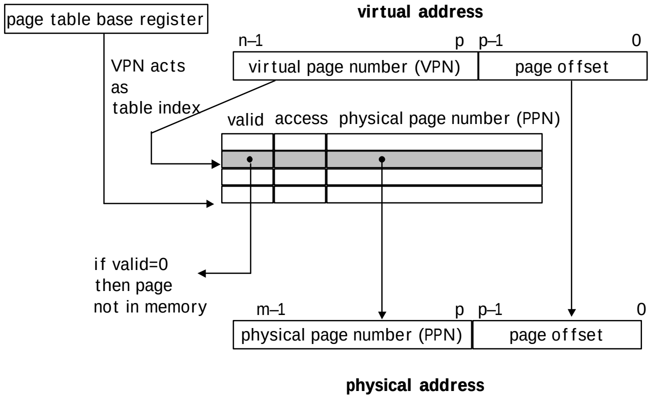


Notice that the page offset bits don't change as a result of translation

Page Tables


Virtual Page Number
Memory resident page table
(physical page or disk address)
Valid
1 1 0 1 1 1 0 1 0 1
Physical Memory
Disk Storage
(swap file or regular file system file)

Address Translation via Page Table



Page Table Operation

Translation

- Separate (set of) page table(s) per process
- VPN forms index into page table (points to a page table entry)

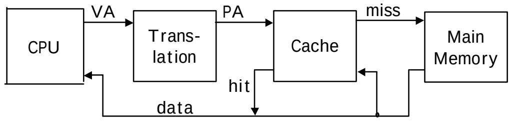
Computing Physical Address

- Page Table Entry (PTE) provides information about page
  - if (valid bit = 1) then the page is in memory.
    - » Use physical page number (PPN) to construct address
  - if (valid bit = 0) then the page is on disk
    - » Page fault
    - » Must load page from disk into main memory before continuing

Checking Protection

- Access rights field indicate allowable access
  - e.g., read-only, read-write, execute-only
  - typically support multiple protection modes (e.g., kernel vs. user)
- Protection violation fault if user doesn't have necessary permission

Integrating VM and Cache



Most Caches "Physically Addressed"

- Accessed by physical addresses
- Allows multiple processes to have blocks in cache at same time
- Allows multiple processes to share pages
- Cache doesn't need to be concerned with protection issues
  - Access rights checked as part of address translation

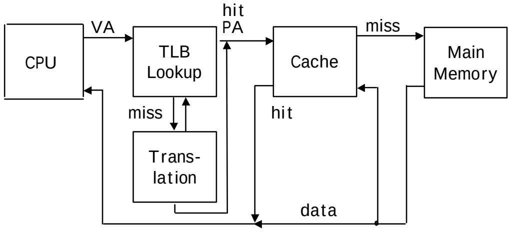
Perform Address Translation Before Cache Lookup

- But this could involve a memory access itself (of the PTE)
- Of course, page table entries can also become cached

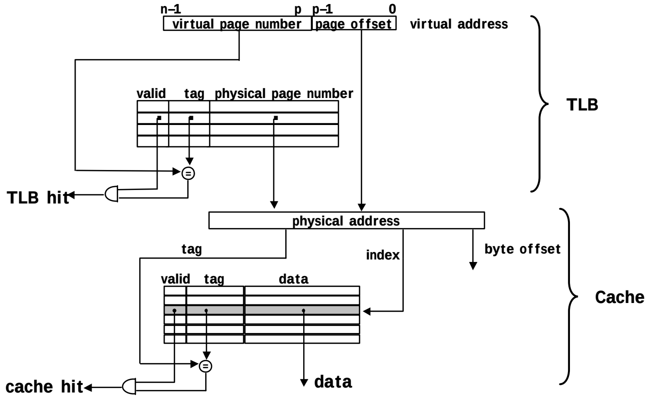
Speeding up Translation with a TLB

"Translation Lookaside Buffer" (TLB)

- Small hardware cache in MMU
- Maps virtual page numbers to physical page numbers
- Contains complete page table entries for small number of pages



Address Translation with a TLB



Address translation summary

Symbols:

- **Components of the virtual address (VA)**
  - TLBI: TLB index
  - TLBT: TLB tag
  - VPO: virtual page offset
  - VPN: virtual page number
- **Components of the physical address (PA)**
  - PPO: physical page offset (same as VPO)
  - PPN: physical page number
  - CO: byte offset within cache line
  - CI: cache index
  - CT: cache tag

Address translation summary (cont)

Processor:

- **execute an instruction to read the word at address VA into a register.**
- **send VA to MMU**

- **MMU:**
  - **receive VA from MMU**
  - **extract TLBI, TLBT, and VPO from VA.**
  - **if TLB[TLBI].valid and TLB[TLBI].tag = TLBT, then TLB hit.**
  - ***note: requires no off-chip memory references.***
  - **if TLB hit:**
    - read PPN from TLB line.
    - construct PA = PPN+VPO (+ is bit concatenation operator)
    - send PA to cache
    - *note: requires no off-chip memory references*

Address translation summary (cont)

MMU (cont)

- **if TLB miss:**
  - if PTE[VPN].valid, then page table hit.
  - if page table hit:
    - » PPN = PTE[VPN].ppn
    - » PA = PPN+VPO (+ is bit concatenation operator)
    - » send PA to cache
    - » *note: requires an off-chip memory reference to the page table.*
  - if page table miss:
    - » transfer control to OS via page fault exception.
    - » OS will load missing page and restart instruction.

Cache:

- **receive PA from MMU**
- **extract CO, CI, and CT from PA**
- **use CO, CI, and CT to access cache in the normal way.**

Multi-level Page Tables

Given:

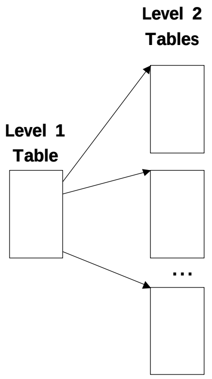
- 4KB (2^{12}) page size
- 32-bit address space
- 4-byte PTE

Problem:

- Would need a 4 MB page table (2^{20} *4 bytes) per process!

Common solution

- multi-level page tables
- e.g., 2-level table (Pentium II)
  - Level 1 table: 1024 entries, each which points to a Level 2 page table.
  - Level 2 table: 1024 entries, each of which points to a page



Pentium II Memory System

Virtual address space

- 32 bits (4 GB max)

Page size

- 4 KB (can also be configured for 4 MB)

Instruction TLB

- 32 entries, 4-way set associative.

Data TLB

- 64 entries, 4-way set associative.

L1 instruction cache

- 16 KB, 4-way set associative, 32 B linesize.

L1 data cache

- 16 KB, 4-way set associative, 32 B linesize.

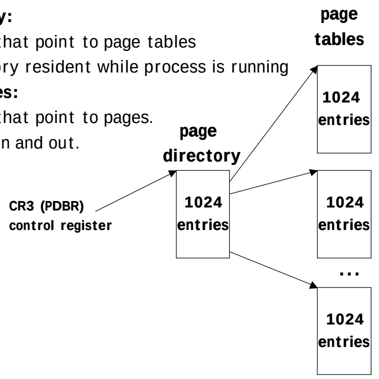
Unified L2 cache

- 512 KB (2 MB max), 4-way set associative, 32 B linesize

Pentium II Page Table Structure

2-level per-process page table:

- 1 Page directory:
  - 1024 entries that point to page tables
  - must be memory resident while process is running
- 1024 page tables:
  - 1024 entries that point to pages.
  - can be paged in and out.



Pentium II Page Directory Entry

31 … 12	11 … 9	8	7	6	5	4	3	2	1	0
page table base addr	Avail	G	PS	0	A	CD	WT	U/S	R/W	P

Avail: available for system programmers
G: global page (don't evict from TLB)
PS: page size (0 -> 4K)
A: accessed (set by MMU on reads and writes)
CD: cache disabled
WT: write-through
U/S: user/supervisor
R/W: read/write
P: present

Pentium II Page Table Entry

31 12	11 9	8	7	6	5	4	3	2	1	0
page base address	Avail	G	0	D	A	CD	WT	U/S	R/W	P=1

Avail: available for system programmers
G: global page (don't evict from TLB)
D: dirty (set by MMU on writes)
A: accessed (set by MMU on reads and writes)
CD: cache disabled
WT: write-through
U/S: user/supervisor
R/W: read/write
P: present

31 1	0
Available for OS	P=0

Main Themes

Programmer's View

- **Large "flat" address space**
  - Can allocate large blocks of contiguous addresses
- **Processor "owns" machine**
  - Has private address space
  - Unaffected by behavior of other processes

System View

- **User virtual address space created by mapping to set of pages**
  - Need not be contiguous
  - Allocated dynamically
  - Enforce protection during address translation
- **OS manages many processes simultaneously**
  - Continually switching among processes
  - Especially when one must wait for resource
    - » E.g., disk I/O to handle page fault