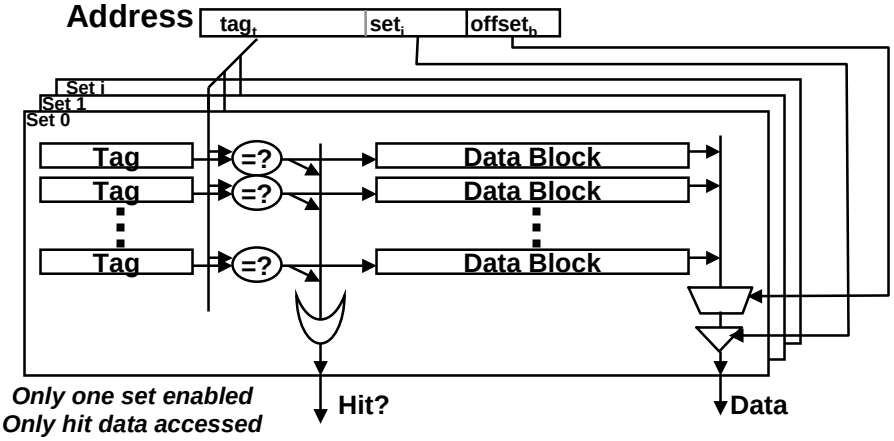
Improving Cache Performance and Memory Management: *From Absolute Addresses to Demand Paging*

Krste Asanovic
Laboratory for Computer Science
M.I.T.

http://www.csg.lcs.mit.edu/6.823

Highly-Associative Caches

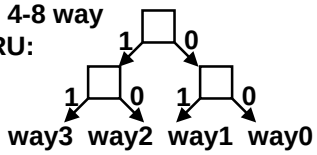
- For high associativity, use content-addressable memory (CAM) for tags
- Used in low-power microprocessors, e.g. StrongARM is 32-way set-associative. *(Higher hit rates at lower energy than 2-4 way set-ass. RAM tags)*
- Comparator per tag requires more transistors (~double area per tag bit)



Replacement Policy

In an associative cache, which block from a set should be evicted when the set becomes full?

- ***Random***
- ***Least Recently Used (LRU)***
  - **LRU cache state must be updated on every access**
  - **true implementation only feasible for small sets (2-way easy)**
  - **pseudo-LRU binary tree often used for 4-8 way**
    e.g. 3 state bits for 4-way pseudo-LRU:

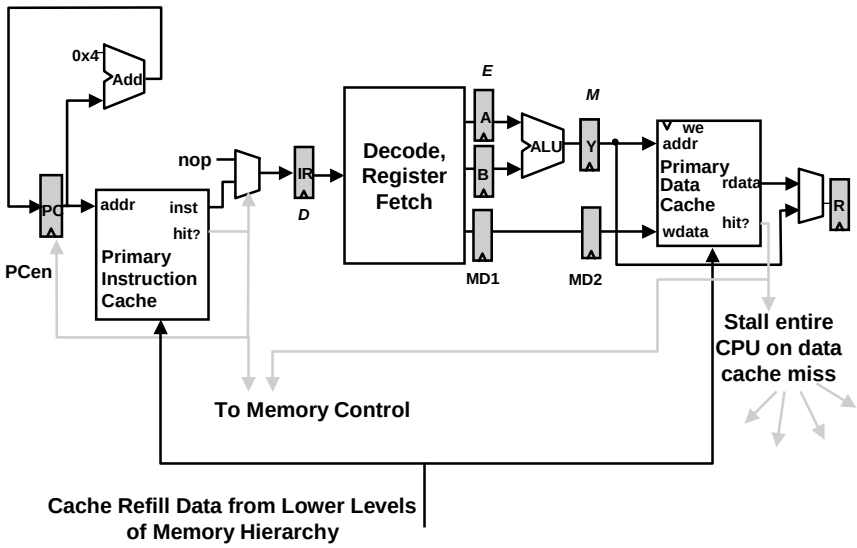


- ***First In, First Out (FIFO)* aka. *Round-Robin***
  - **used in highly associative caches**

This is a second-order effect. How often does replacement happen?

CPU-Cache Interaction

(Simple 5-stage pipeline)



Write Policy

Cache hit:

write through: **write both cache & memory**

- generally higher traffic but simplifies cache coherence

write back: **write cache only (memory is written only when the entry is evicted)**

- a dirty bit per block can further reduce the traffic

Cache miss:

no write allocate: **only write to main memory**

write allocate: (aka fetch on write)

fetch block into cache

Common combinations:

write through and no write allocate

write back with write allocate

Managing Cache Writes

In a direct-mapped cache, can we write cache data RAM in same cycle as cache tag RAM read?

In a highly-associative cache with CAM tags, can we write cache data in the same cycle as tag CAM search?

Pipelining Cache Writes

Possible solutions:

- **Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit**
- **Design data RAM that can perform read *and* write in one cycle, restore old value after tag miss**
- **Hold write data for store in single buffer ahead of cache, write cache data during *next* store's tag check**
  - **Need to bypass from write buffer if read matches write buffer tag**
- **Use CAM tags --- data write only enabled if hit**

Cache Performance

Average memory access time =
Hit time + Miss rate x Miss penalty

To improve performance:
- **reduce the hit time**
- **reduce the miss rate (e.g., larger cache)**
- **reduce the miss penalty (e.g., L2 cache)**

First order effect: *the size and the hit time*
⇒ design the largest primary cache without slowing down the clock or adding pipeline stages

Causes for Cache Misses

- ***Compulsory:*** first-reference *aka* ***cold start misses***
  - misses that would occur even with infinite cache
- ***Capacity:*** cache is too small to hold all data needed by the program
  - misses that would occur even under perfect placement & replacement policy
- ***Conflict:*** misses that occur because of collisions due to block-placement strategy
  - misses that would not occur with full associativity

Determining the type of a miss requires running program traces on a cache simulator

Effect of Cache Parameters on Performance

- **Larger cache size**
  - **+ reduces capacity and conflict misses**
  - **- hit time may increase**

- **Larger block size**
  - **+ spatial locality reduces compulsory misses and capacity reload misses**
  - **- fewer blocks may increase conflict miss rate**
  - **- larger blocks may increase miss penalty**

- **Higher associativity**
  - **+ reduces conflict misses (up to around 4-8 way)**
  - **- hit time may increase**

Reducing Hit Time

- On-chip versus off-chip caches
- Pipelining write tag check and data update for single-cycle write-hits
- Sum-Addressed Caches (UltraSPARC-III)
  - Can evaluate A+B=C equality faster than adding A and B!
  - In register+offset addressing mode, no need to perform addition before cache access
  - Tag compare unit performs A+B=C operation
- Pseudo-associative caches (way-predicting caches)
  - Guess which way will have hit, only look there first
  - Check other ways sequentially on miss
  - Combine direct-mapped hit time, associative miss rate
  - Larger hit time if way prediction poor

Techniques to Further Reduce Cache Miss Rate

- Hardware prefetching of instructions and data
  - Can remove compulsory misses
  - Stream buffers predict sequential or strided accesses
  - Speculative fetches can add useless memory traffic
- Software prefetching
  - Software prefetch requires extending the ISA with register and cache prefetch instructions
  - Takes extra instruction issue slots
  - Software tuned to one hardware implementation
- Other software techniques
  - placement (array padding) to reduce cache conflicts
  - blocking transformations (reuse data once in cache) to reduce capacity misses

Reducing Miss Penalty

- **Give priority to read-misses over writes and write-backs**
  - **queue writes in write buffer, let read overtake writes to memory**
  - **must check write buffer for match on read address**
- **Multi-level caches**
  - **reduced latency on a primary cache miss**
- **Fetch critical word in block first** ***(*****aka** ***wrap-around refill)***
  - **restart the processor as soon as needed word arrives**
- **Sub-block placement, fetch only part of a block on miss**
  - **small tag overhead from large blocks, but low miss penalty from small sub-blocks**
  - **need an extra valid bit per sub-block**

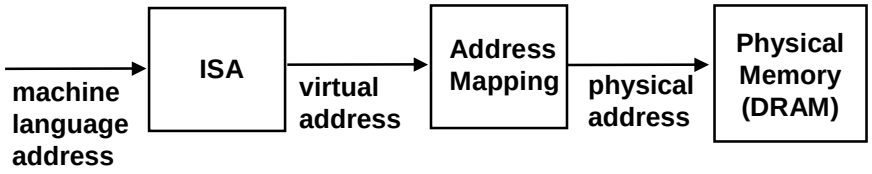
V	Tag	V	sub-blk. 0	V	sub-blk. 1	V	sub-blk. 2	V	sub-blk. 3

- **Victim caches**
  - **hold recently evicted blocks nearby to reduces conflict miss penalty for low-associativity caches**
- **Non-blocking caches to reduce stalls on cache misses**

Memory Management

- **The Fifties:**
  - **Absolute Addresses**
  - **Dynamic address translation**
- **The Sixties:**
  - **Paged memory systems and TLBs**
  - **Atlas' Demand paging**
- **Modern Virtual Memory Systems (next lecture)**

Types of Names for Memory Locations



- **Machine language address**
  - ⇒ **as specified in machine code**
- **Virtual address**
  - ⇒ **ISA specifies translation of machine code address into virtual address of program variable (may involve segment registers etc.)**
- **Physical address**
  - ⇒ **operating system specifies mapping of virtual address into name for a physical memory location (i.e., actual address signals going to DRAM chips)**

Absolute Addresses

EDSAC, early 50's

effective address = physical memory address

- **Only one program ran at a time, with unrestricted access to entire machine (RAM + I/O devices)**
- **Addresses in a program depended upon where the program was to be loaded in memory**
- **But it was more convenient for programmers to write location-independent subroutines**

  ⇒ **Lead to the development of *loaders* & *linkers* to statically relocate and link programs**

Dynamic Address Translation

Motivation:

In the early machines, I/O operations were slow and each word transferred involved the CPU

Higher throughput if CPU and I/O of two or more programs were overlapped

⇒ *multiprogramming*

Location independent programs:

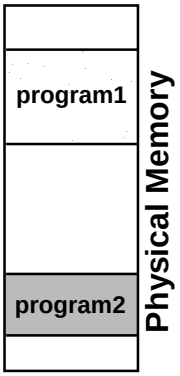
Programming and storage management ease

⇒ need for a *base register*

Protection:

Independent programs should not affect each other inadvertently

⇒ need for a *bound register*

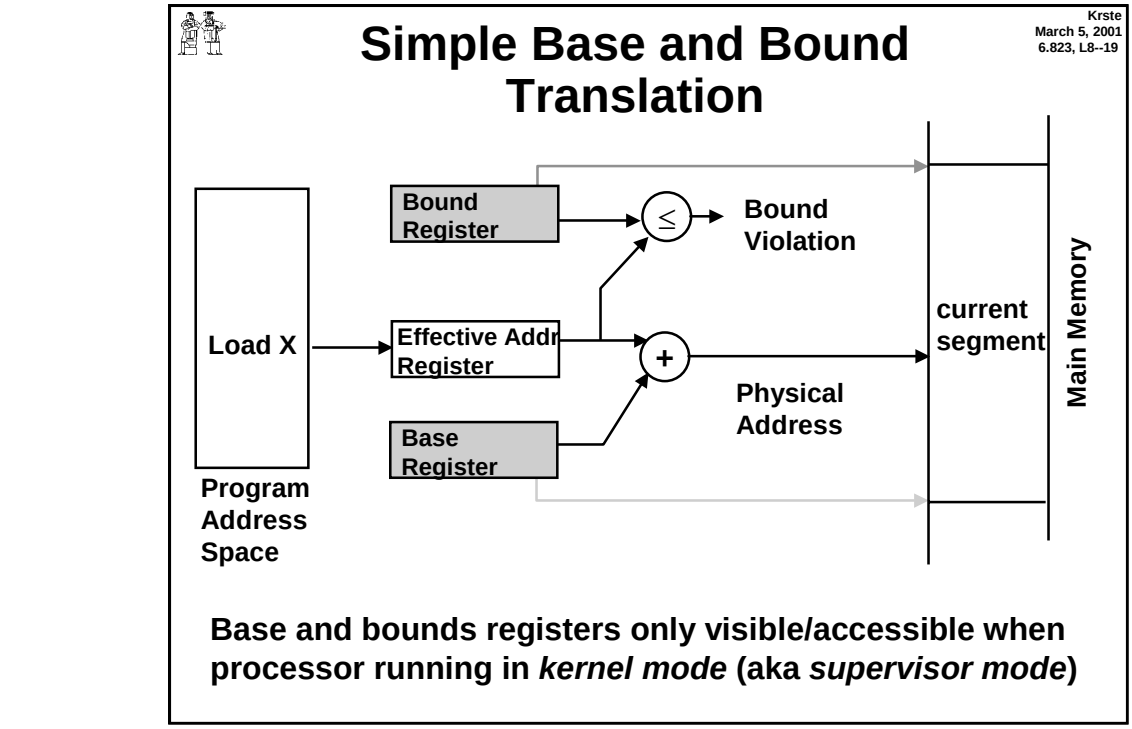


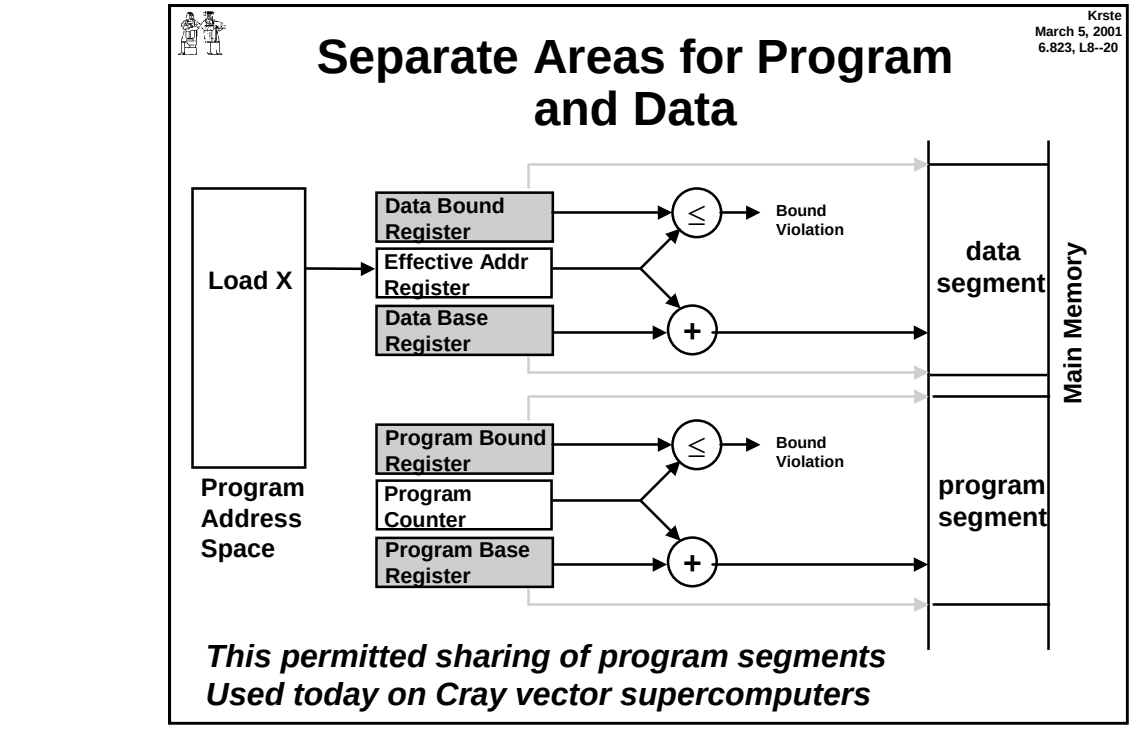
User versus Kernel

- **With multiprogramming came move away from programming bare machine**
  - **users should be protected from each other (and program bugs)**
  - **users need to share resources (e.g., CPU time, memory, disk)**
- **Hardware support evolves to support OS**
  - **device interrupts to support multiprogramming (can't enforce that users' software polls for each others devices)**
  - **protected state and privileged execution modes to run OS**

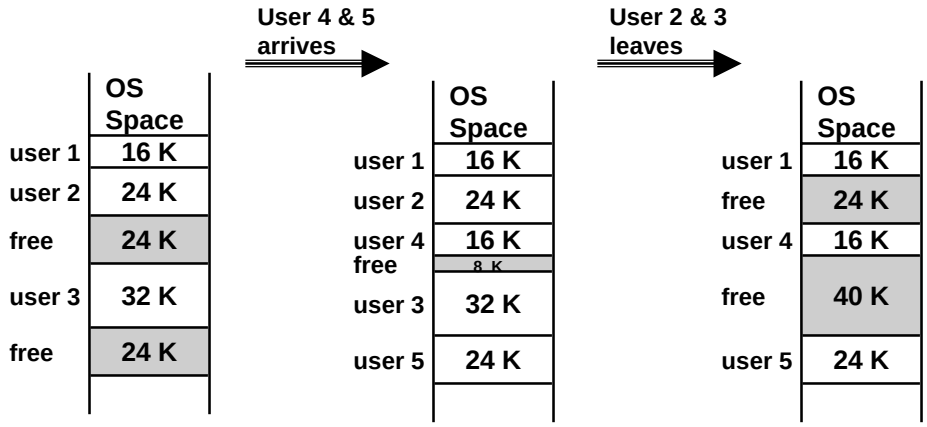
    ⇒ *exceptions to catch protection violations*
- **Purely software schemes to manage protection**
  - **high-level language programming only (no assembly code)**
  - **trusted compiler**

  ⇒ *software bugs cause security loopholes and system crashes*


Krste
March 5, 2001
6.823, L8--19
Simple Base and Bound Translation
Load X
Program Address Space
Bound Register
Effective Addr Register
Base Register
≤
+
Bound Violation
Physical Address
current segment
Main Memory
Base and bounds registers only visible/accessible when processor running in kernel mode (aka supervisor mode)


Krste
March 5, 2001
6.823, L8--20
Separate Areas for Program and Data
Load X
Program Address Space
Data Bound Register
Effective Addr Register
Data Base Register
Program Bound Register
Program Counter
Program Base Register
≤
+
Bound Violation
Bound Violation
data segment
program segment
Main Memory
This permitted sharing of program segments
Used today on Cray vector supercomputers

Memory Fragmentation



As users come and go, the storage is "fragmented". Therefore, at some stage programs have to be moved around to compact the storage *(burping the memory)*.

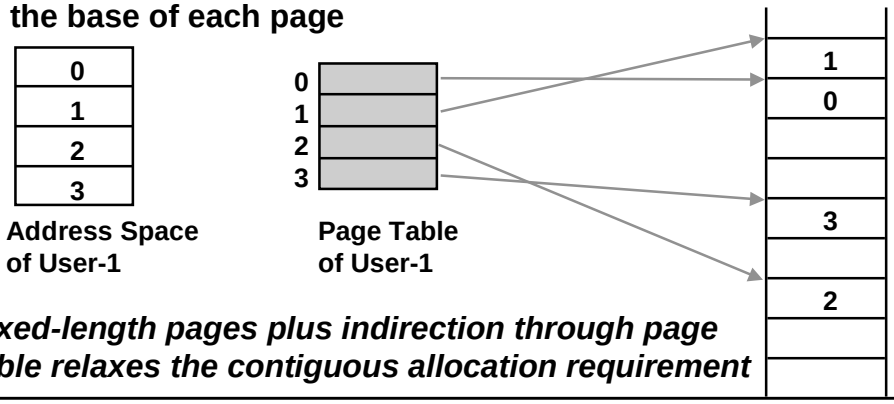
Paged Memory Systems:

To reduce fragmentation

Processor generated address can be interpreted as a pair <page number,offset>

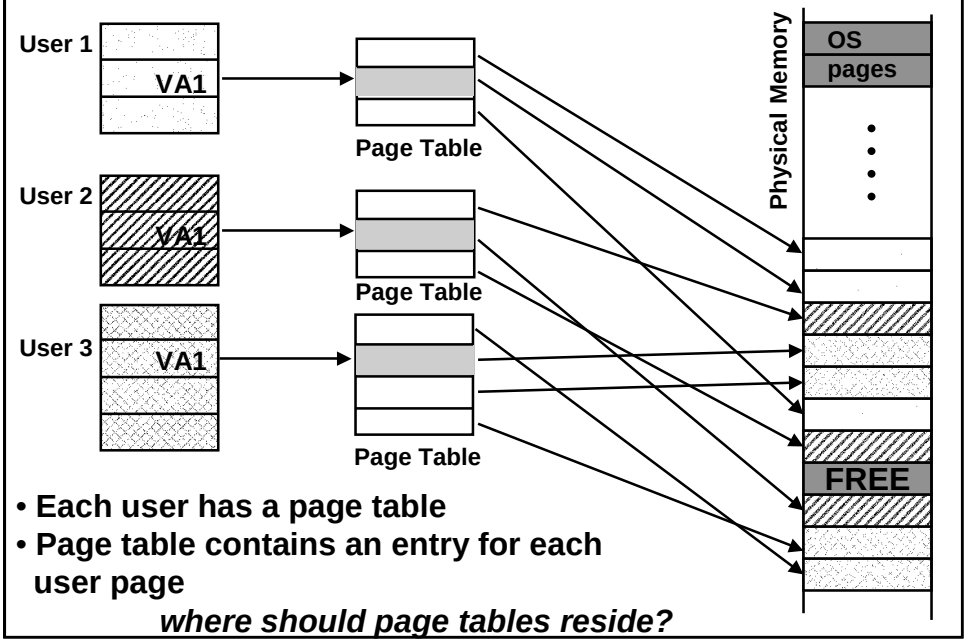
page number	offset

A page table contains the physical address of the base of each page



Fixed-length pages plus indirection through page table relaxes the contiguous allocation requirement

Private Address Space per User



- Each user has a page table
- Page table contains an entry for each user page

where should page tables reside?

Where Should Page Tables Reside?

Space required by the page tables is proportional to the page size, number of users, ...

⇒ Space requirement is large

too expensive to keep in registers

Special registers just for the current user:

- need new management instructions

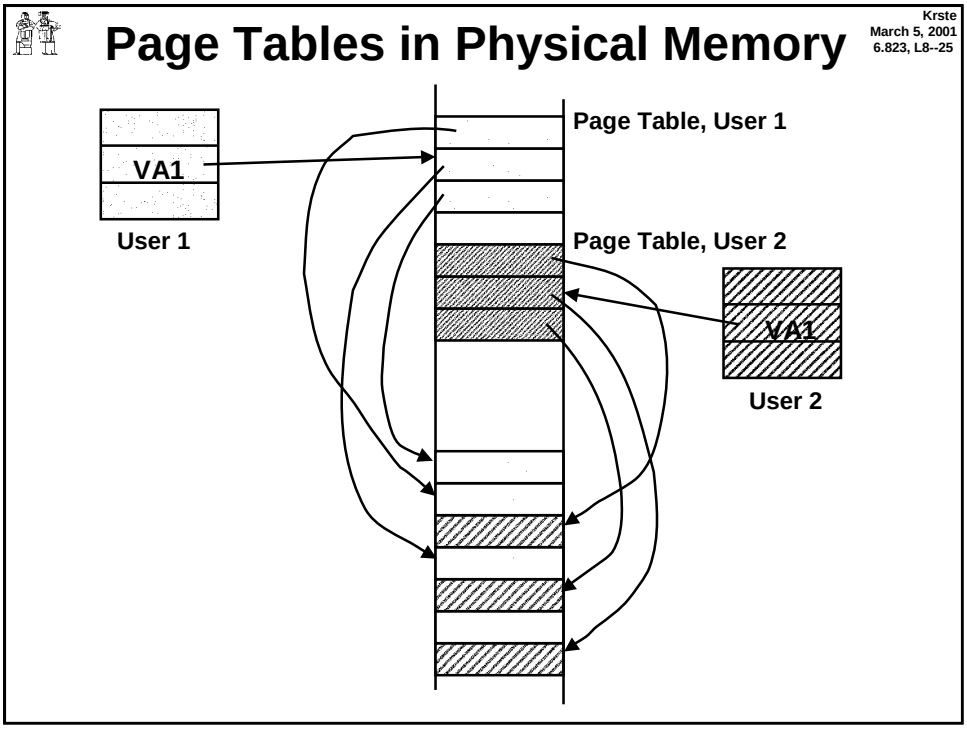
- affects the context-switching time

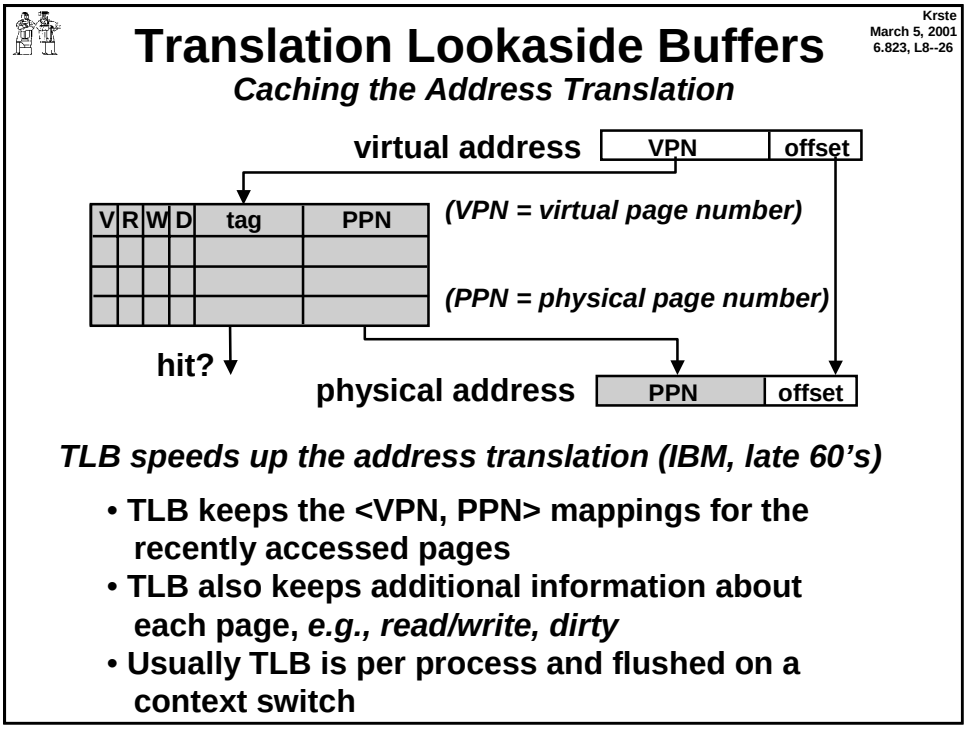
may not be feasible for large page tables

Main memory:

- needs one reference to retrieve the page base address and another to access the data word

⇒ *doubles number of memory references!*


Krste
March 5, 2001
6.823, L8--25
Page Tables in Physical Memory
VA1
User 1
Page Table, User 1
Page Table, User 2
VA1
User 2


Krste
March 5, 2001
6.823, L8--26
Translation Lookaside Buffers
Caching the Address Translation
virtual address
VPN
offset
V
R
W
D
tag
PPN
(VPN = virtual page number)
(PPN = physical page number)
hit?
physical address
PPN
offset
TLB speeds up the address translation (IBM, late 60's)
• TLB keeps the <VPN, PPN> mappings for the recently accessed pages
• TLB also keeps additional information about each page, e.g., read/write, dirty
• Usually TLB is per process and flushed on a context switch

A Problem in Early Sixties

There were many applications whose data could not fit in the main memory, e.g., Payroll

Paged memory system reduced fragmentation but still required the whole program to be resident in the main memory

Programmers moved the data back and forth from the secondary store by *overlaying* it repeatedly on the primary store

tricky programming!

Manual Overlays

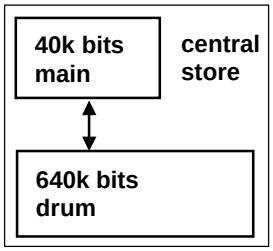
Assuming an instruction can address all the storage on the drum

***method1* - programmer keeps track of addresses in the main memory and initiates an I/O transfer when required**

***method2* - automatic initiation of I/O transfers by software address translation**

Brookner's interpretive coding, 1960

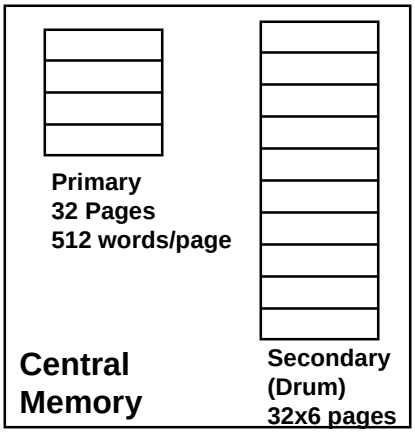
method 1 proved too difficult for users and method 2 too slow!



Ferranti Mercury 1956

Demand Paging

Atlas, 1962



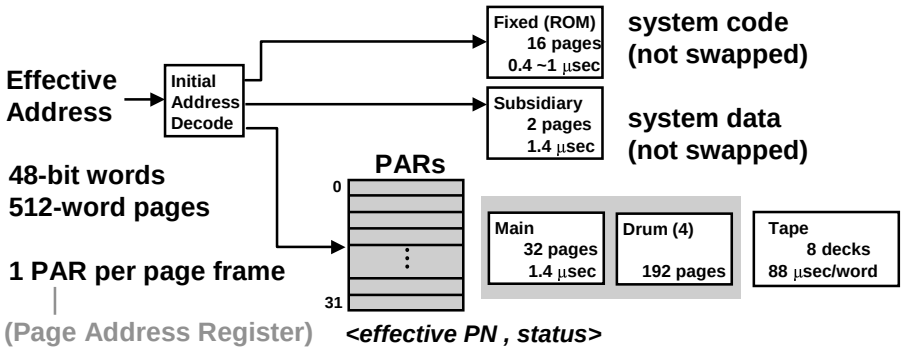
"A page from secondary storage is brought into the primary storage whenever it is (implicitly) demanded by the processor."

Tom Kilburn

Primary memory as a *cache* for secondary memory

User sees 32 x 6 x 512 words of storage

Hardware Organization of Atlas



Compare the effective page address against all 32 PARs

match ⇒ normal access

no match ⇒ *page fault*

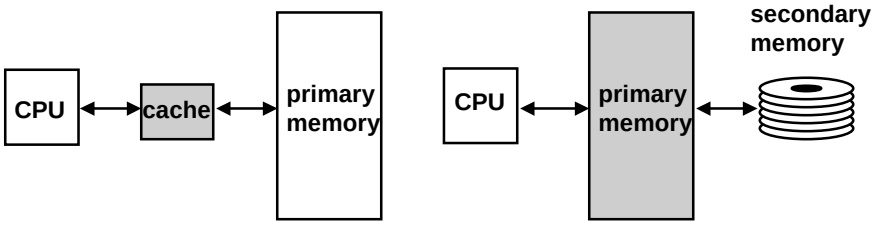
the state of the partially executed instruction was saved

Atlas Demand Paging Scheme

On a page fault:

- **input transfer into a free page is initiated**
- **the PAR is updated**
- **if no free page is left, a *page is selected to be replaced* (based on usage)**
- **the replaced page is written on the drum**
  - **to minimize drum latency effect, the first empty page on the drum was selected**
- **the *page table* is updated to point to the new location of the page on the drum**

Caching vs Demand Paging



Caching

- **cache entry**
- **cache block (~32 bytes)**
- **cache miss (1% to 20%)**
- **cache hit (~1 cycle)**
- **cache miss (~10 cycles)**
- **a miss is handled in *hardware***

Demand paging

- **page-frame**
- **page (~4K bytes)**
- **page miss (<0.001%)**
- **page hit (~100 cycles)**
- **page miss(~5M cycles)**
- **a miss is handled mostly in *software***