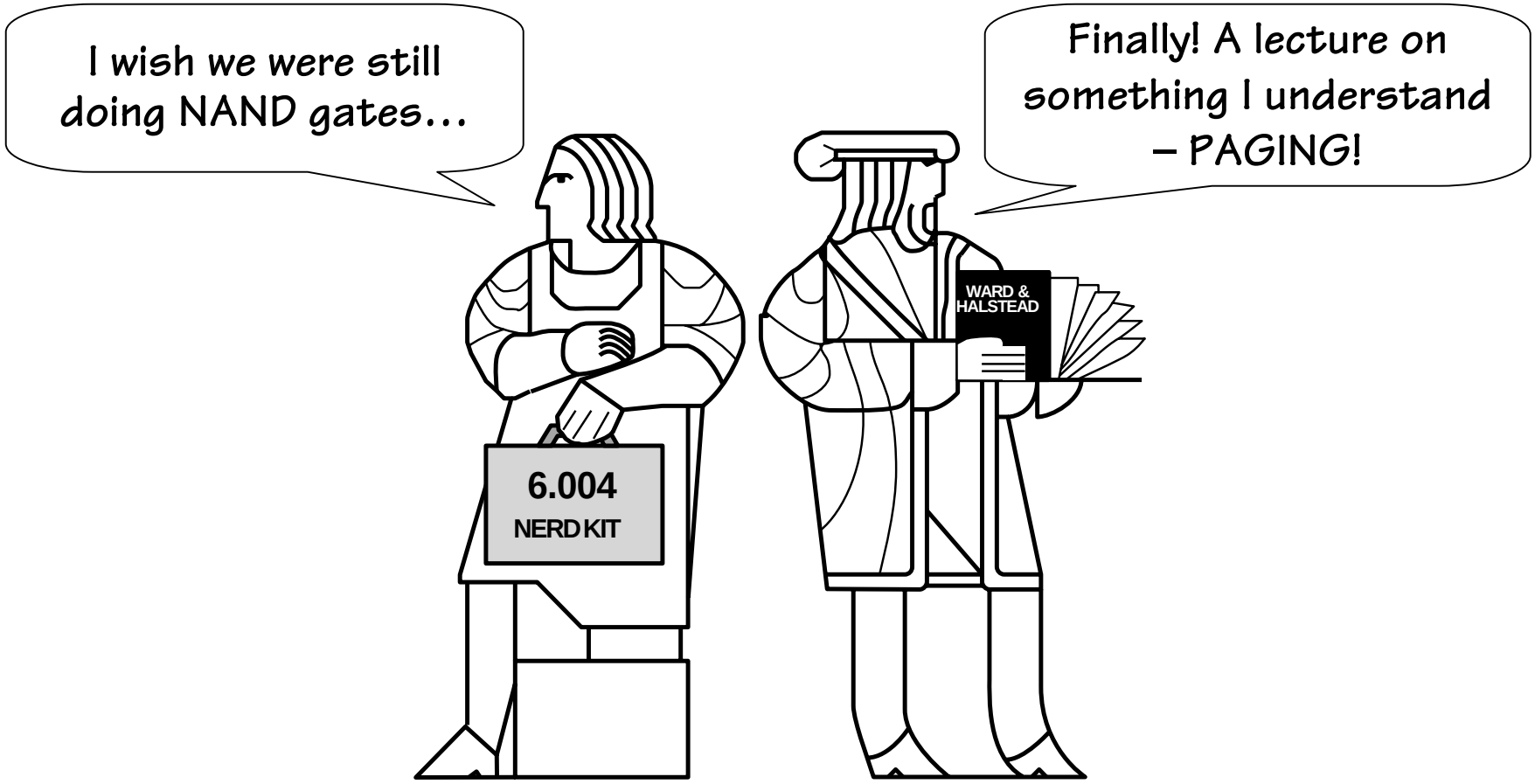
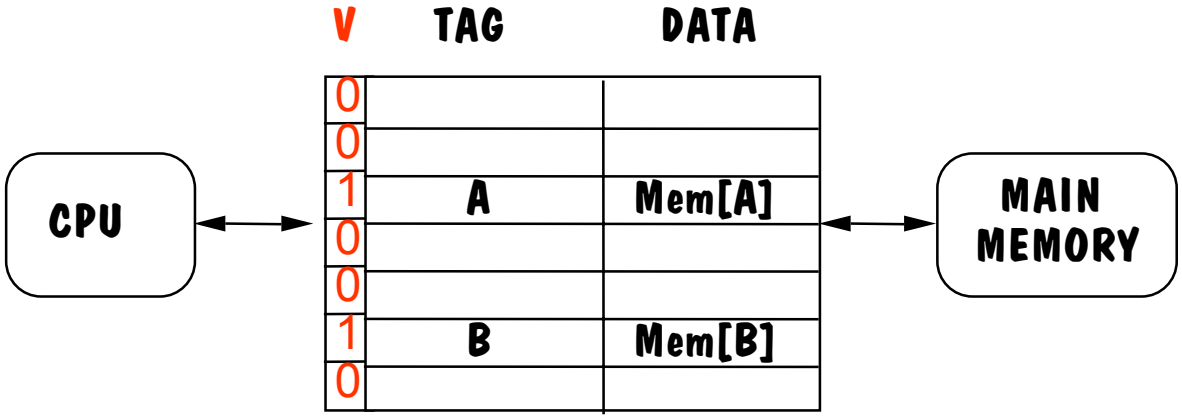
Virtual Memory



Handouts: Lecture Slides

From Last Time: Valid Bits



Problem:

Ignoring cache lines that don't contain anything of value... e.g., on

- start-up
- "Back door" changes to memory (eg loading program from disk)

Solution:

Extend each TAG with VALID bit.

- Valid bit must be set for cache line to HIT.
- At power-up / reset : clear all valid bits
- Set valid bit when cache line is first replaced.
- Cache Control Feature: Flush cache by clearing all valid bits, Under program/external control.

Handling of WRITES

Observation: Most (90+%) of memory accesses are *READs*. How should we handle writes? Issues:

Write-through: CPU writes are cached, but also written to main memory (stalling the CPU until write is completed). Memory always holds "the truth".

Write-behind: CPU writes are cached; writes to main memory may be *buffered*, perhaps pipelined. CPU keeps executing while writes are completed (in order) in the background.

Write-back: CPU writes are cached, but not immediately written to main memory. Memory contents can be "stale".

Our caches thus far have used write-through.

Can we improve write performance?

Write-Through

ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: *X == TAG(i) , for some cache line i*

READ: return DATA[I]
WRITE: change DATA[I]; Start Write to Mem[X]

MISS: *X not found in TAG of any cache line*

REPLACEMENT SELECTION:
Select some line k to hold Mem[X]

READ: Read Mem[X]
Set TAG[k] = X, DATA[k] = Mem[X]

WRITE: Start Write to Mem[X]
Set TAG[k] = X, DATA[k] = new Mem[X]

Write-Back

ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: *X = TAG(i) , for some cache line I*

READ: return DATA(i)
WRITE: change DATA(i); ~~Start Write to Mem[X]~~

MISS: *X not found in TAG of any cache line*

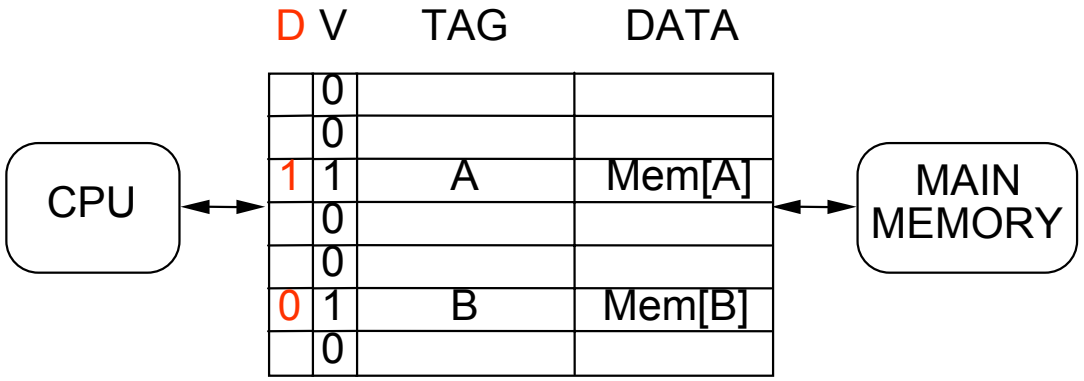
REPLACEMENT SELECTION:
Select some line k to hold Mem[X]
Write Back: Write Data(k) to Mem[Tag[k]]

READ: Read Mem[X]
Set TAG[k] = X, DATA[k] = Mem[X]

WRITE: ~~Start Write to Mem[X]~~
Set TAG[k] = X, DATA[k] = new Mem[X]

Is write-back worth the trouble? Depends on (1) cost of write; (2) consistency issues.

Write-Back w/ "Dirty" Bits



ON REFERENCE TO Mem[X]: Look for X among tags...

HIT: *X = TAG(i) , for some cache line I*

READ: return DATA(i)

WRITE: change DATA(i); ~~Start Write to Mem[X]~~ D[i]=1

MISS: *X not found in TAG of any cache line*

REPLACEMENT SELECTION:

Select some line k to hold Mem[X]

If D[k] == 1 (Write Back) Write Data(k) to Mem[Tag[k]]

READ: Read Mem[X]; Set TAG[k] = X, DATA[k] = Mem[X], D[k]=0

WRITE: ~~Start Write to Mem[X]~~ D[k]=1

Set TAG[k] = X, DATA[k] = new Mem[X]

Cache Trends & Observations

Associativity:

- Less important as size increases
- 2-way or 4-way usually plenty for typical program clustering; BUT additional associativity smoothes performance curve and reduces number of select bits (we'll see shortly how this helps)
- TREND: Invest in RAM, not comparators.

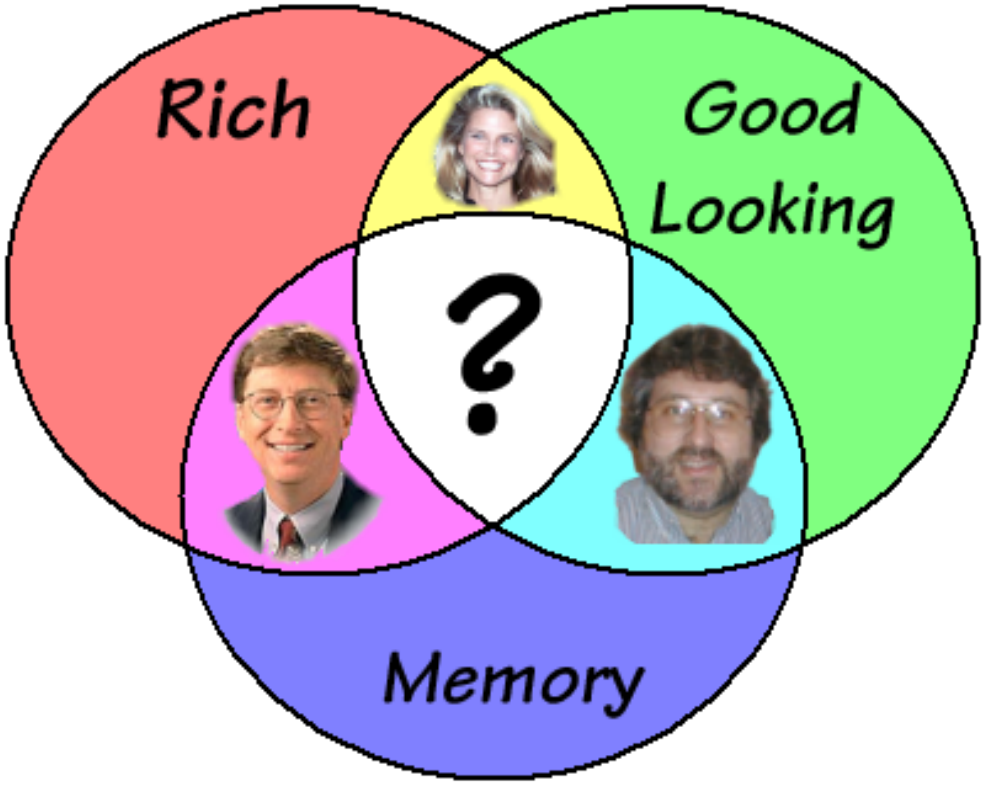
Replacement Strategy:

- BIG caches: any sane approach works well
- REAL randomness assuages paranoia!

Performance Analysis:

- Tedious hand synthesis may build intuition from simple examples, BUT
- Computer simulation of cache behavior on REAL programs (or using REAL trace data) is the basis for most real-world cache design decisions. YOU'LL DO THIS IN LAB 9!

You can never be too rich, too good looking, or have too much memory!



Now that we've learned how to FAKE a FAST memory, we'll turn our attention to FAKING a large memory.

Top 10 Reasons for a BIG Address Space

10. Keeping TI's memory division in business.

9. Unique addresses within every internet host.

8. Generating good 6.004 Final problems.

7. Performing ADD via table lookup

6. Support for meaningless advertising hype

5. Emulation of a Turning Machine's tape.

4. Supporting lazy programmers.

3. Isolating ISA from IMPLEMENTATION
   - details of HW configuration shouldn't enter into SW design

2. Usage UNCERTAINTY
   - provide for run-time expansion of stack and heap

1. Programming CONVENIENCE
   - create regions of memory with different semantics: read-only, shared, etc.
   - avoid annoying bookkeeping

Lessons from History…

There is only one mistake that can be made in computer design that is difficult to recover from—not having enough address bits for memory addressing and memory management.

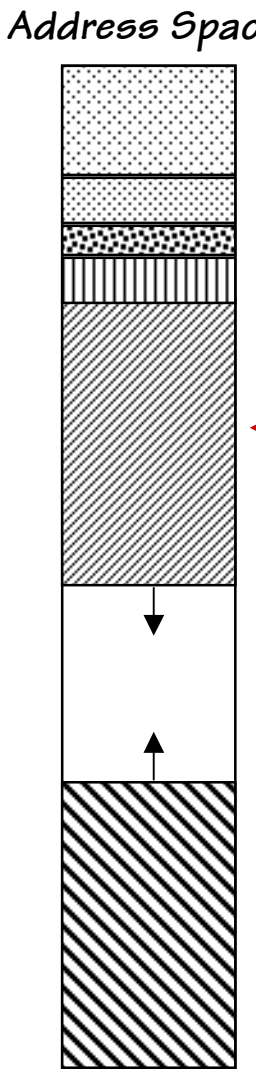
Gordon Bell and Bill Strecker
speaking about the PDP-11 in 1976

A partial list of successful machines that eventually starved to death for lack of address bits includes the PDP 8, PDP 10, PDP 11, Intel 8080, Intel 8086, Intel 80186, Intel 80286, Motorola 6800, AMI 6502, Zilog Z80, Cray-1, and Cray X-MP.

Hennessy & Patterson

Why? Address size determines minimum width of anything that can hold an address: PC, registers, memory words, HW for address arithmetic (BR/JMP, LD/ST). When you run out of address space it's time for a new ISA!

Squandering Address Space



CODE, large monolithic programs (eg, Office, Netscape)....

- only small portions might be used
- add-ins and plug-ins
- shared libraries/DLLs
- ...

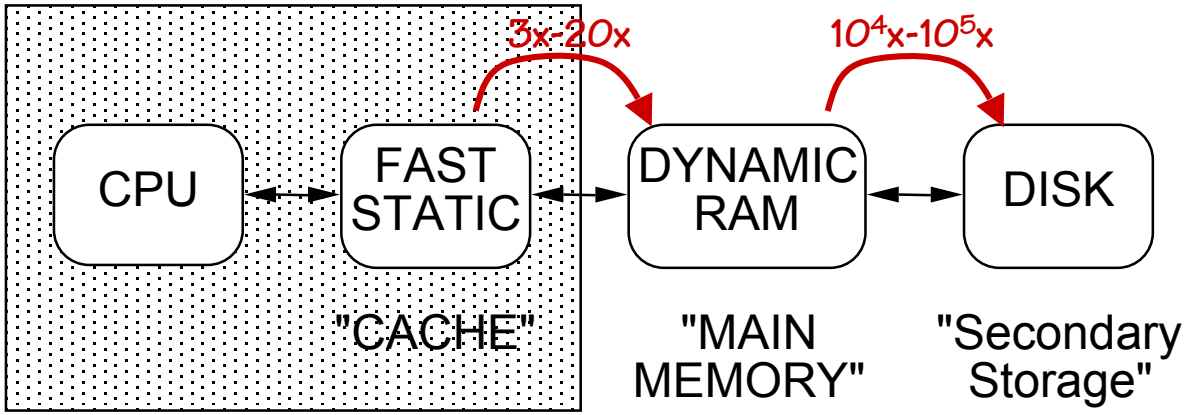
STACK: How much to reserve? (consider RECURSION!)

OBSERVATIONS:

- Can't BOUND each usage... without compromising use.
- Actual use is SPARSE
- Working set even MORE sparse

HEAP: N variable-size data records...
Bound N? Bound Size?

Extending the Memory Hierarchy



So, we've used SMALL fast memory + BIG slow memory to fake BIG FAST memory.

Can we combine RAM and DISK to fake DISK size at RAM speeds?

VIRTUAL MEMORY

- use of RAM as cache to much larger storage pool, on slower devices
- TRANSPARENCY - VM locations "look" the same to program whether on DISK or in RAM.
- ISOLATION of RAM size from software.

Virtual Memory

ILLUSION: Huge memory
(2^{32} bytes? 2^{64}bytes?)

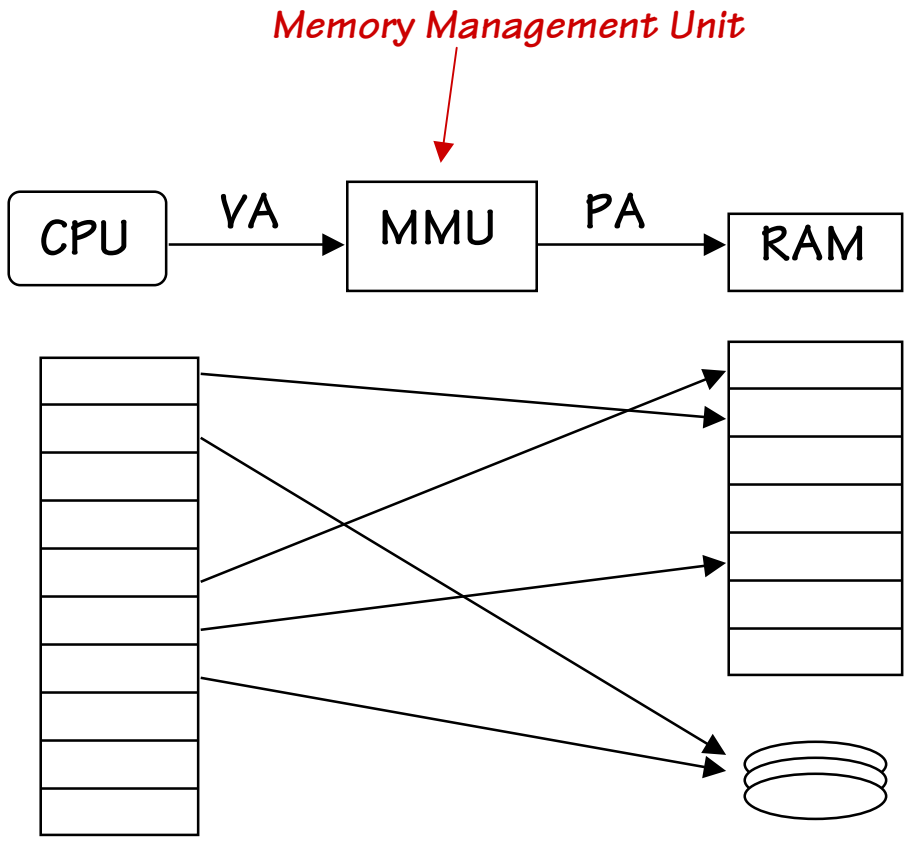
ACTIVE USAGE: small fraction
(2^{24} bytes?)

HARDWARE:

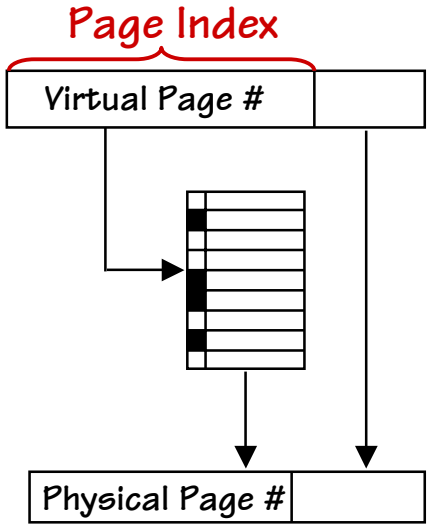
- 2^{26} (64M) bytes of RAM
- 2^{32} (4 G) bytes of DISK...
  ... maybe more, maybe less!

ELEMENTS OF DECEIT:

- Partition memory into "Pages" (2K-4K-8K)
- MAP a few to RAM, others to DISK
- Keep "HOT" pages in RAM.



Simple Page Map Design



FUNCTION: Given Virtual Address,

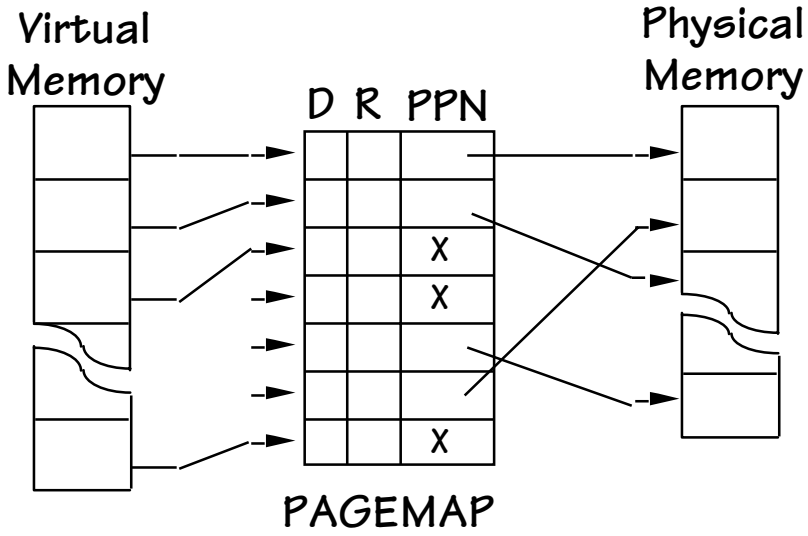
- Map to PHYSICAL address

OR

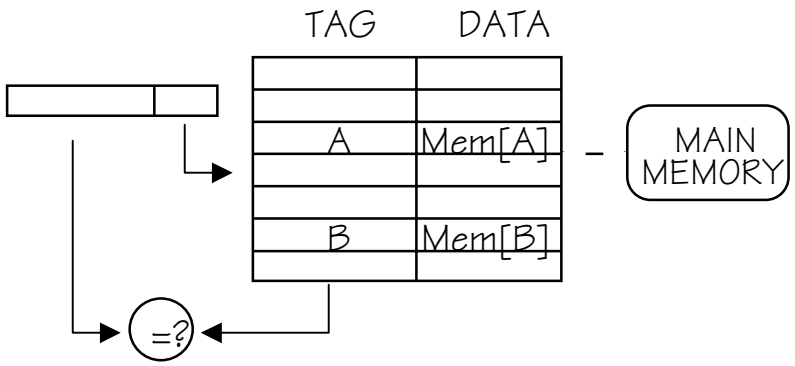
- Cause PAGE FAULT allowing page replacement

Why use HIGH address bits to select page?
... LOCALITY.
Keeps related data on same page.

Why use LOW address bits to select cache line?
... LOCALITY.
Keeps related data from competing for same cache lines.

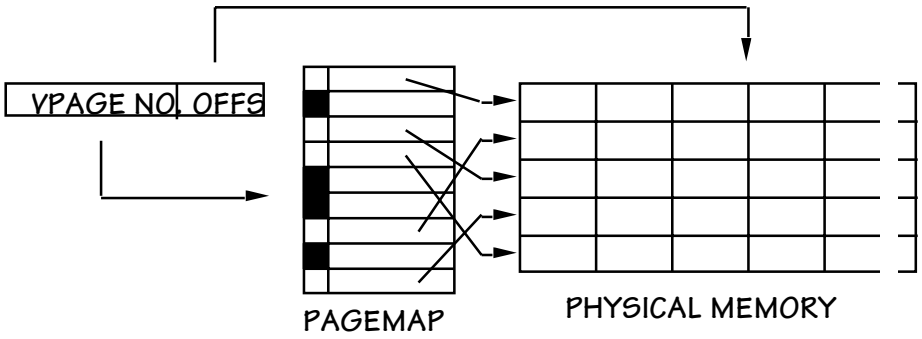


Virtual Memory vs. Cache



Cache:

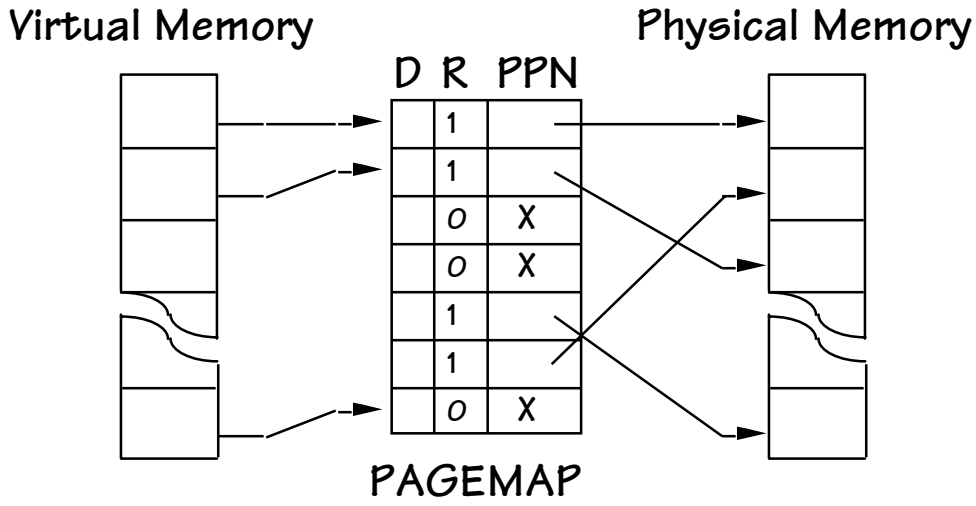
- Relatively short blocks
- Few lines: scarce resource
- miss time: 3x-20x hit times



Virtual memory:

- disk: long latency, fast xfer
  - → miss time: ~10^5 x hit time
  - → write-back essential!
  - → large pages in RAM
- lots of lines: one for each page
- tags in page map, data in physical memory

Virtual Memory: the VI-1 view

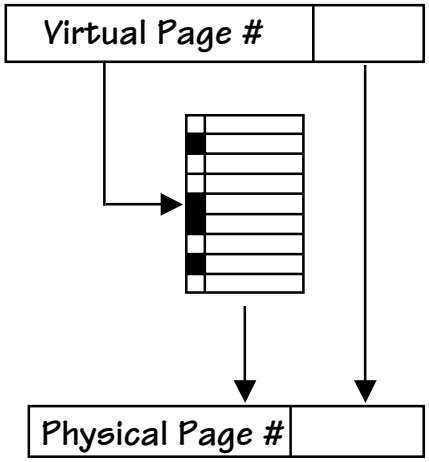


Pagemap Characteristics:

- One entry per virtual page!
- RESIDENT bit = 1 for pages stored in RAM, or 0 for non-resident (disk or unallocated). Page fault when R = 0.
- Contains PHYSICAL page number (PPN) of each resident page
- DIRTY bit says we've changed this page since loading it from disk (and therefore need to write it to disk when it's replaced)

Virtual Memory: the VI-3 view

Problem: Translate
VIRTUAL ADDRESS
to PHYSICAL ADDRESS




```
int VtoP(int VPageNo,int PO) {
 if (R[VPageNo] == 0)
    PageFault(VPageNo);
 return (PPN[VPageNo] << p) | PO;
}

/* Handle a missing page... */
void PageFault(int VPageNo) {
  int i;

  i = SelectLRUPage();
  if (D[i] == 1)
    WritePage(DiskAdr[i],PPN[i]);
  R[i] = 0;

  PPN[VPageNo] = PPN[i];
  ReadPage(DiskAdr[VPageNo],PPN[i]);
  R[VPageNo] = 1;
  D[VPageNo] = 0;
}
```

The HW/SW Balance

IDEA:

- devote HARDWARE to high-traffic, performance-critical path
- use (slow, cheap) SOFTWARE to handle exceptional cases

hardware

```
int VtoP(int VPageNo,int PO) {
 if (R[VPageNo] == 0)PageFault(VPageNo);
 return (PPN[VPageNo] << p) | PO;
}
```

software

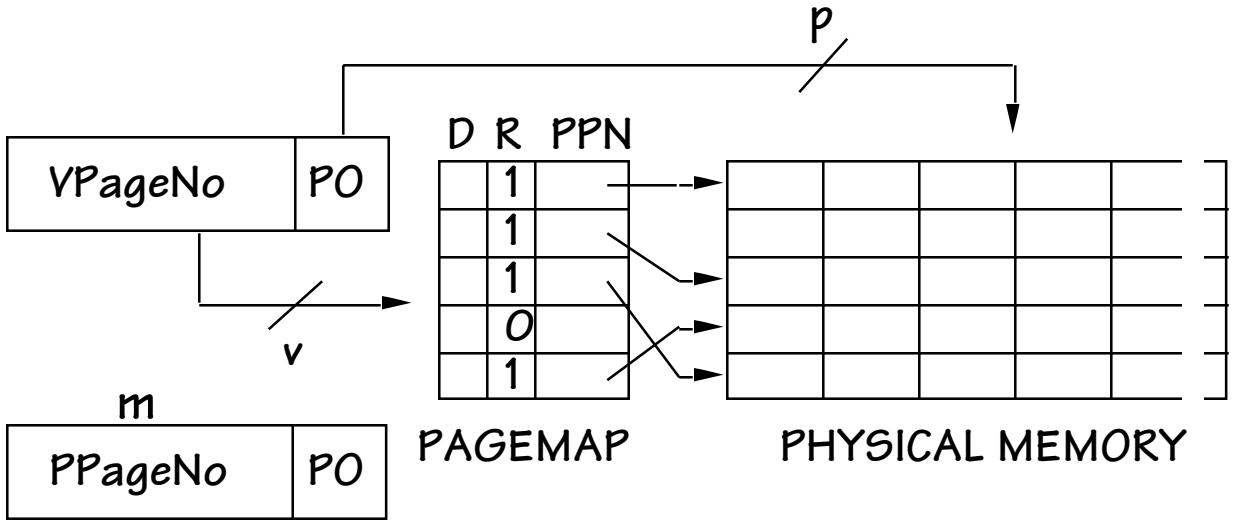
```
/* Handle a missing page... */
void PageFault(int VPageNo) {
  int i = SelectLRUPage();
  if (D[i] == 1) WritePage(DiskAdr[i],PPN[i]);
  R[i] = 0;

  PA[VPageNo] = PPN[i];
  ReadPage(DiskAdr[VPageNo],PPN[i]);
  R[VPageNo] = 1;
  D[VPageNo] = 0;
}
```

HARDWARE performs address translation, detects page faults:

- running program interrupted ("suspended");
- PageFault(...) is forced;
- On return from PageFault; running program continues

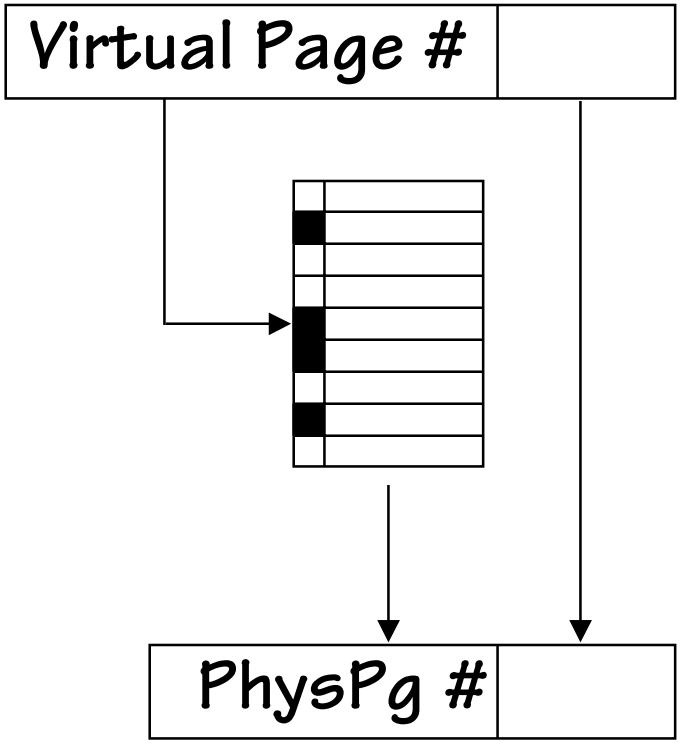
Page Map Arithmetic



$(v + p)$	bits in virtual address
$(m + p)$	bits in physical address
2^v	number of VIRTUAL pages
2^m	number of PHYSICAL pages
2^p	bytes per physical page
2^{v+p}	bytes in virtual memory
2^{m+p}	bytes in physical memory
$(m+2)2^v$	bits in the page map

Typical page size: 1K – 8K bytes
Typical (v+p): 32 (or more) bits
Typical (m+p): 26 – 30 bits
(64 – 1024 MB)

Example: Page Map Arithmetic



SUPPOSE...

32-bit Virtual address

2^{13} page size (8 KB)

2^{26} RAM (64 MB)

THEN:

Physical Pages = _____

Virtual Pages = _____

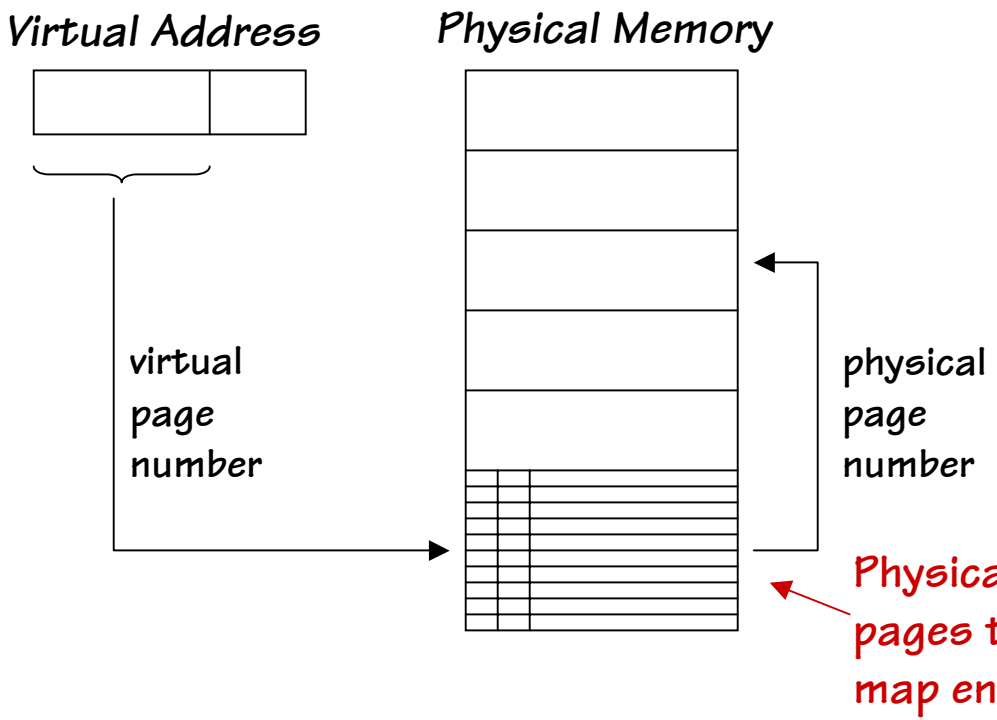
Page Map Entries = _____

Use SRAM for page map??? OUCH!

RAM-Resident Page Maps

SMALL page maps can use dedicated RAM… gets expensive for big ones!

SOLUTION: Move page map to MAIN MEMORY:

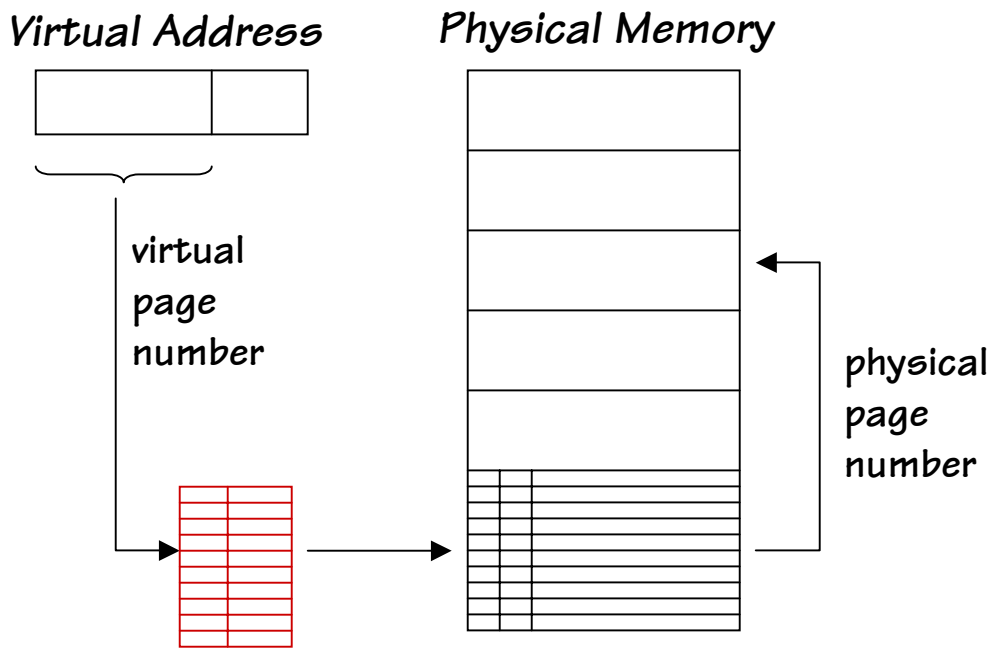


PROBLEM:
Each memory references now takes 2 accesses to physical memory!

Translation Look-aside Buffer (TLB)

PROBLEM: 2x performance hit... each memory reference now takes 2 accesses!

SOLUTION: CACHE the page map entries

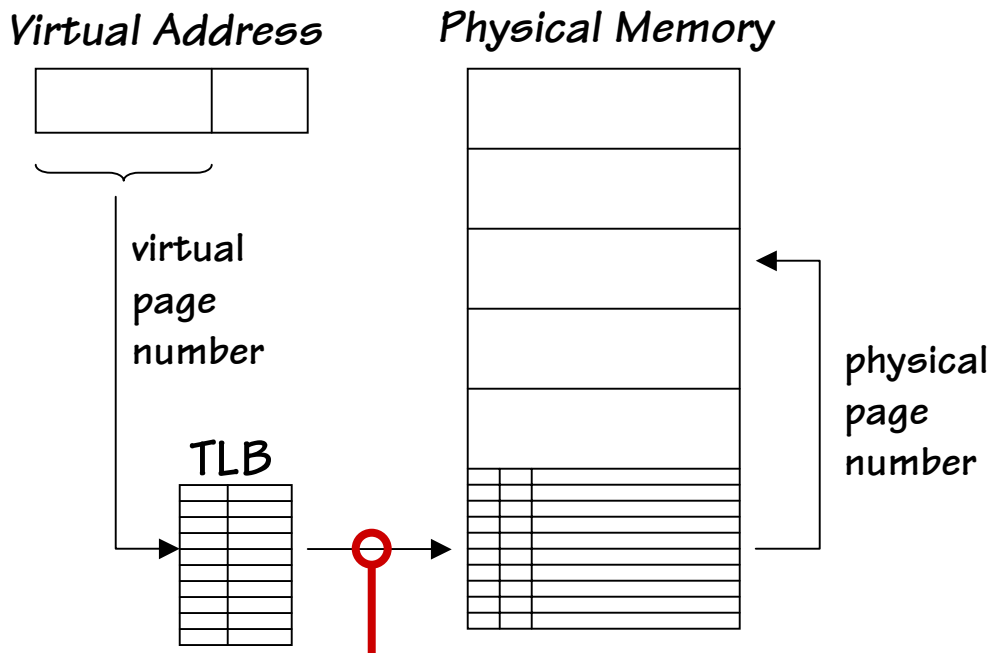


TLB: small, usually fully-associative cache for mapping VPN→PPN

IDEA:
LOCALITY in memory reference patterns → SUPER locality in references to page map

VARIATIONS:
- sparse page map storage
- paging the page map

Optimizing Sparse Page Maps



On TLB miss:

- look up VPN in "sparse" data structure (e.g., a list of VPN-PPN pairs)
- use hash coding to speed search
- only have entries for ALLOCATED pages
- allocate new entries "on demand"
- time penalty? LOW if TLB hit rate is high…

Should we do this in HW or SW?

Multilevel Page Maps

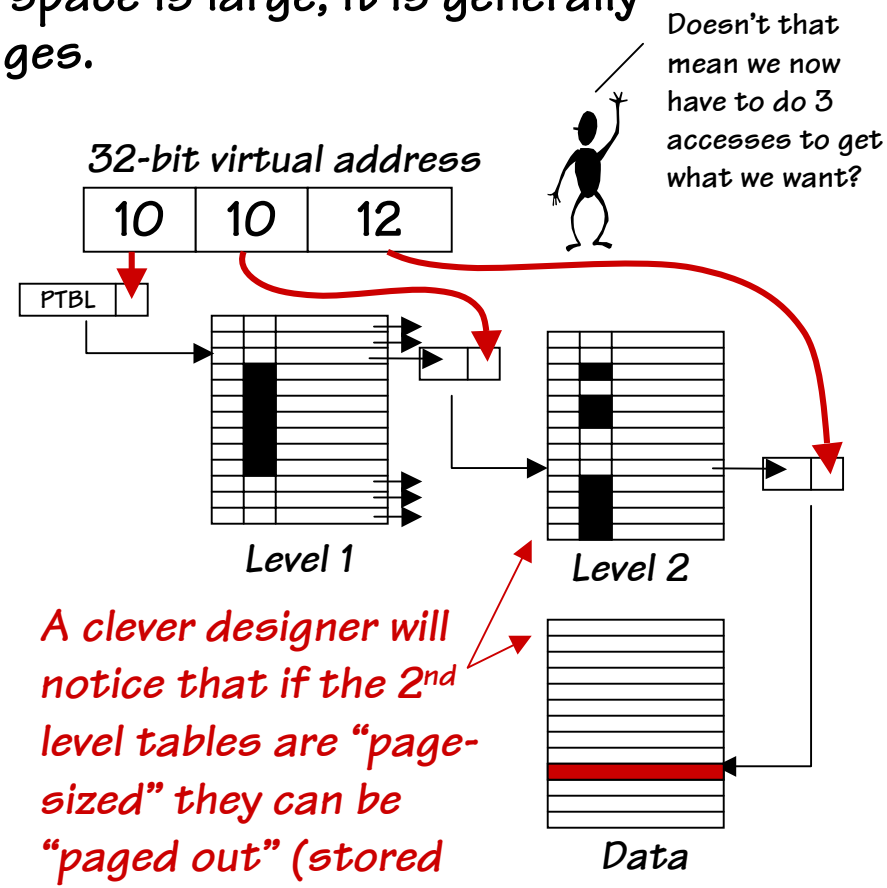
Given a HUGE virtual memory, the cost of storing all of the page map entries in RAM may still be too expensive…

SOLUTION: A hierarchical page map… take advantage of the observation that while the virtual memory address space is large, it is generally sparsely populated with clusters of pages.

Consider a machine with a 32-bit virtual address space and 64 MB (26-bit) of physical memory that uses 4 KB pages.

Assuming 4 byte page-table entries, a single-level page map requires 4MB (>6% of the available memory). Of these, more than 98% will reference non-resident pages (Why?).

A 2-level look-up increases the size of the worse-case page table slightly. However, if a first level entry is non-resident bit it saves large amounts of memory.



Example: mapping VAs to PAs

Suppose

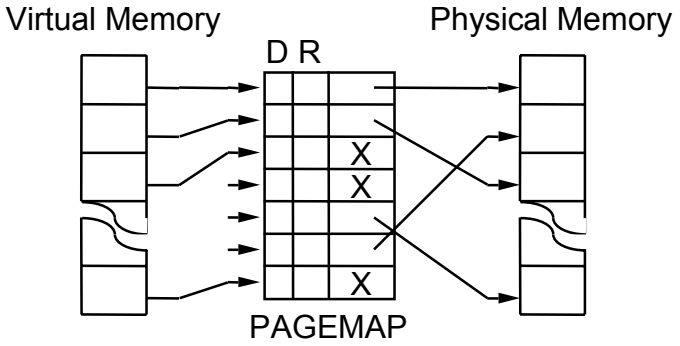
- virtual memory of 2^{32} bytes
- physical memory of 2^{26} bytes
- page size is 2^{12} (4 K) bytes

VPN	R	D	PPN
0	0	0	7
1	1	1	9
2	1	0	0
3	0	0	5
4	1	0	5
5	0	0	3
6	1	1	2
7	1	0	4
8	1	0	1
...			

1. How many pages can be stored in physical memory at once?
2. How many entries are there in the page table?
3. How many bits are necessary per entry in the page table? (Assume each entry has PPN, resident bit, dirty bit)
4. How many pages does the page table require?
5. A portion of the page table is given to the left. What is the physical address for virtual address 0x6004?

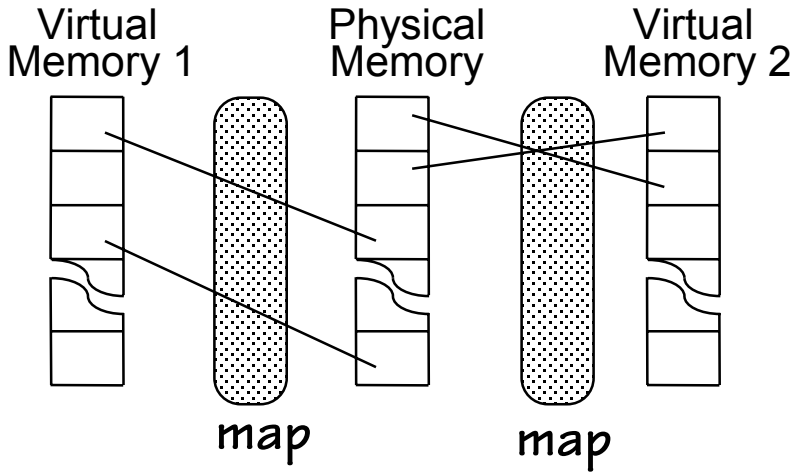
Contexts

A *context* is a mapping of VIRTUAL to PHYSICAL locations, as dictated by contents of the page map:

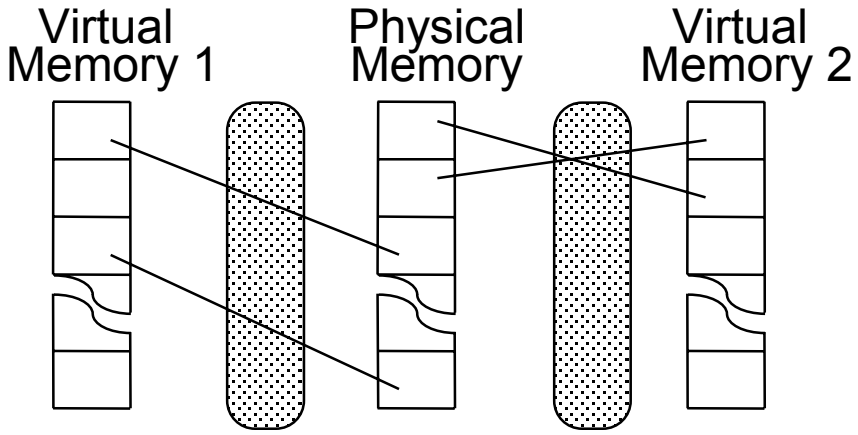


Several programs may be simultaneously loaded into main memory, each in its separate context:

"Context switch":
reload the page map!



Contexts: A Sneak Preview

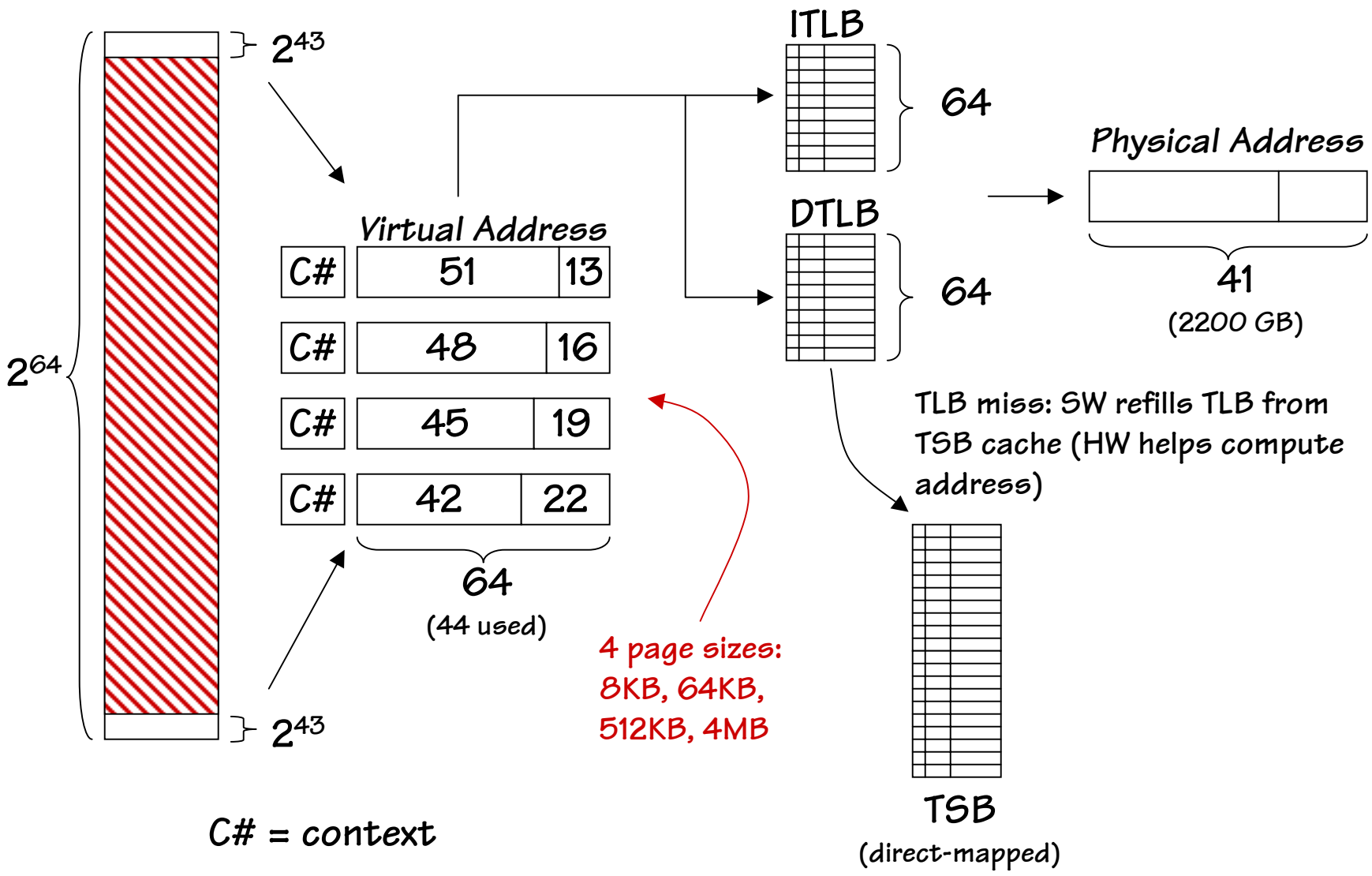


1. TIMESHARING among several programs --
   - Separate context for each program
   - OS loads appropriate context into pagemap when switching among pgms

2. Separate context for OS "Kernel" (eg, interrupt handlers)...
   - "Kernel" vs "User" contexts
   - Switch to Kernel context on interrupt;
   - Switch back on interrupt return.

   HARDWARE SUPPORT: 2 HW pagemaps

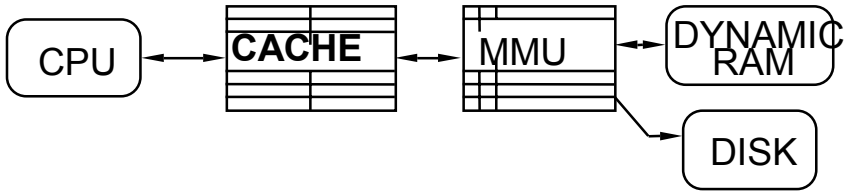
Example: UltraSPARC II MMU



Using Caches with Virtual Memory

Virtual Cache

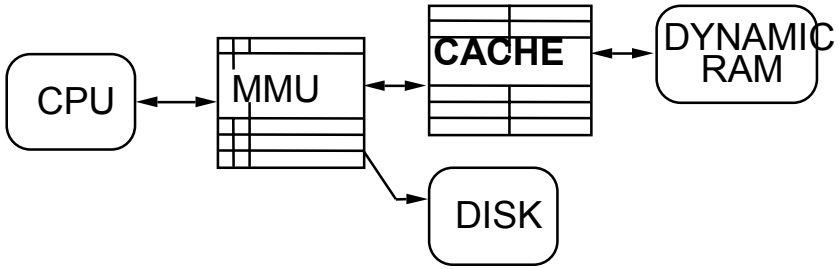
Tags match virtual addresses



- Problem: cache invalid after context switch
- FAST: No MMU time on HIT

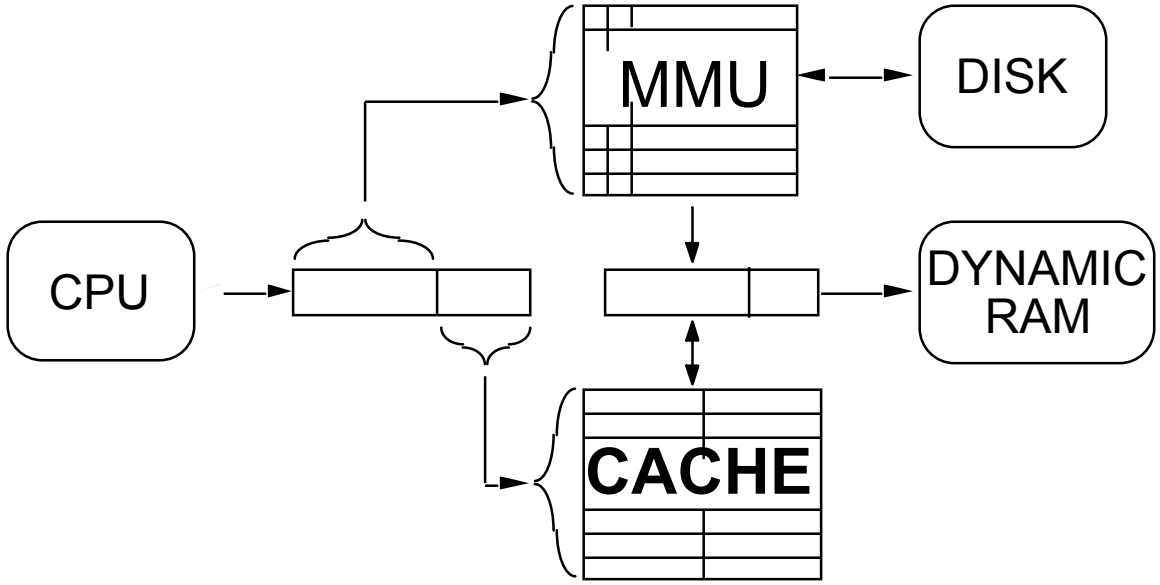
Physical Cache

Tags match physical addresses



- Avoids stale cache data after context switch
- SLOW: MMU time on HIT

Best of both worlds



OBSERVATION: If cache line selection is based on *unmapped* page offset bits, RAM access in a physical cache can *overlap* page map access. Tag from cache is compared with physical page number from MMU.

Want "small" cache index → go with more associativity