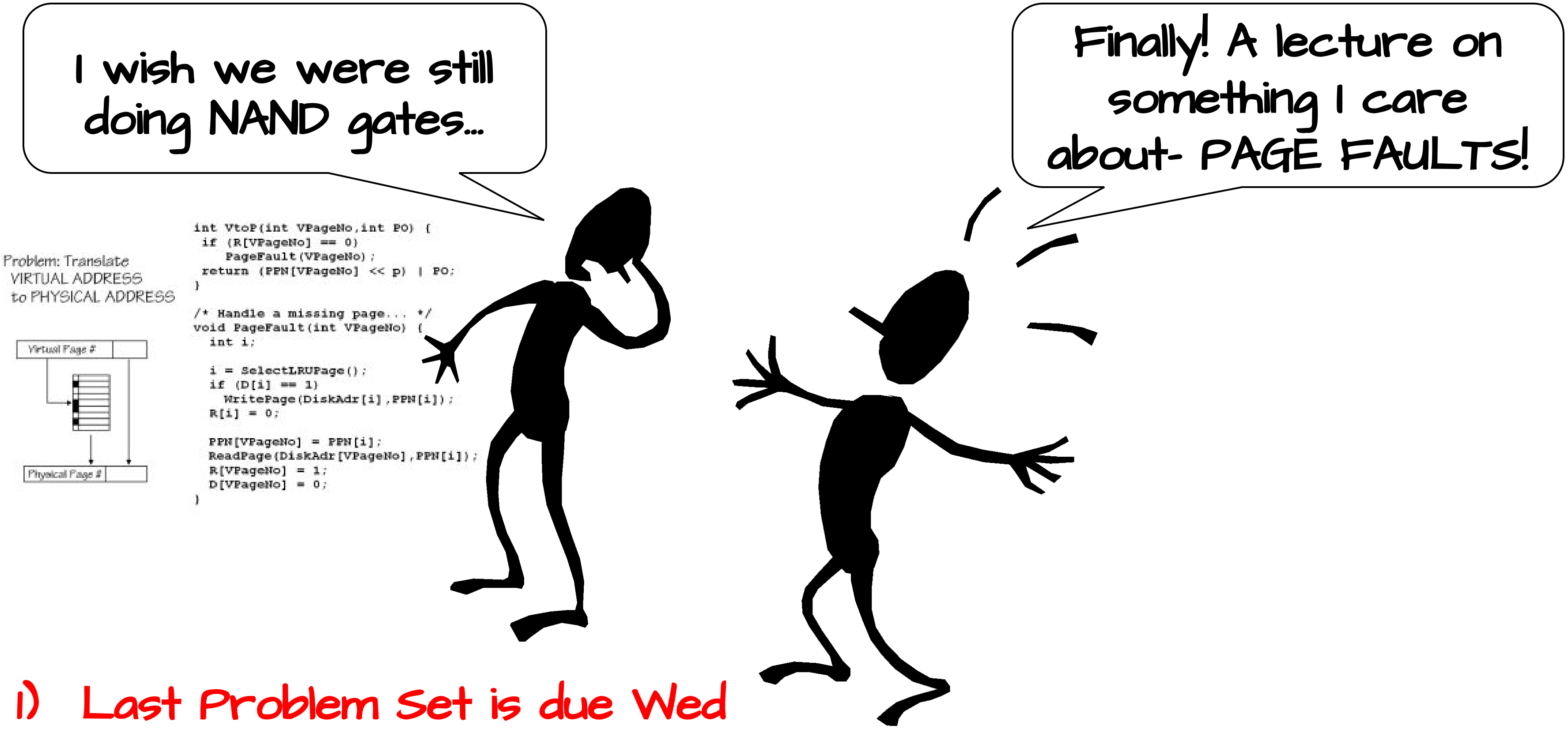
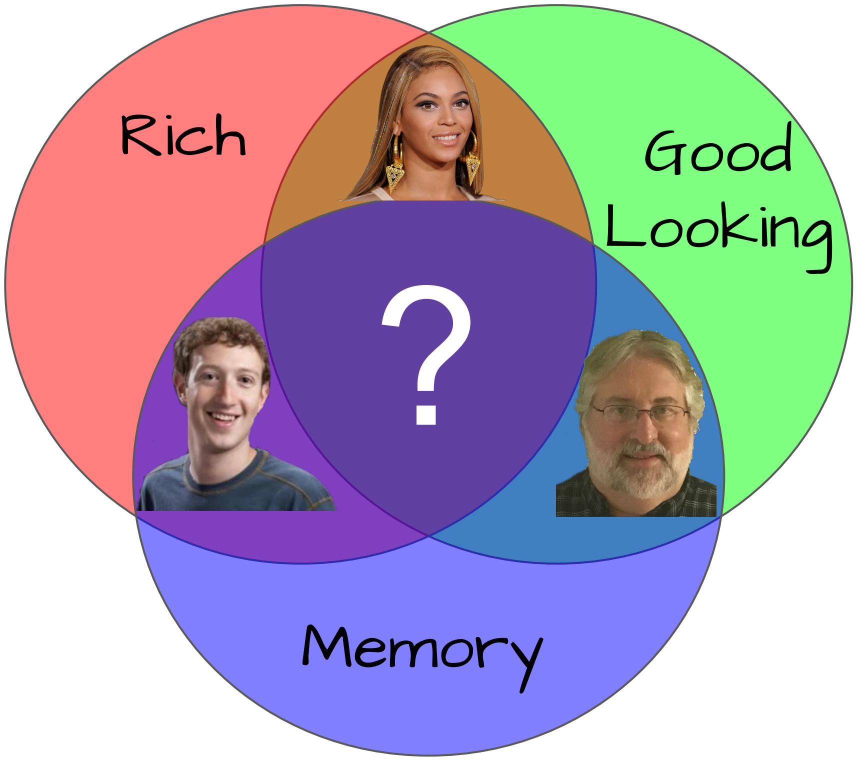
Virtual Memory




```
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;
}
```

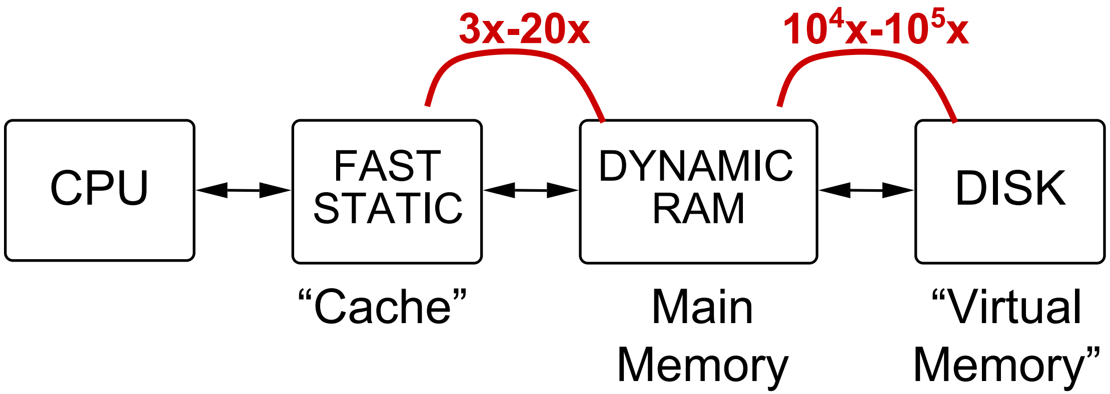
1) Last Problem Set is due Wed
2) Final Exam on Saturday at 8am
3) Second Midterm is graded.
   - We'll go over it on our next and last class meeting.
   - For the remainder, we'll do a final-exam study session.

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



Last time we discussed how to FAKE a FAST memory, this time we'll turn our attention to FAKING a LARGE memory.

Extending the Memory Hierarchy



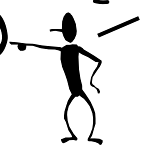
- So far, we've used SMALL fast memory + BIG slow memory to fake a BIG FAST memory (caching).
- Can we combine RAM and DISK to fake DISK sized at near RAM speeds?

<u>VIRTUAL MEMORY</u>

- Use RAM as cache to a much larger storage pool, on slower devices
- TRANSPARENCY - VM locations "look" the same to program whether on DISK or in RAM.
- ISOLATION of actual RAM size from software.
- Support for MULTIPLE, SIMULTANEOUS ADDRESS SPACES

Virtual Memory

ILLUSION: Huge memory
(2^{32} (4G) bytes? 2^{64} (18E) bytes?)

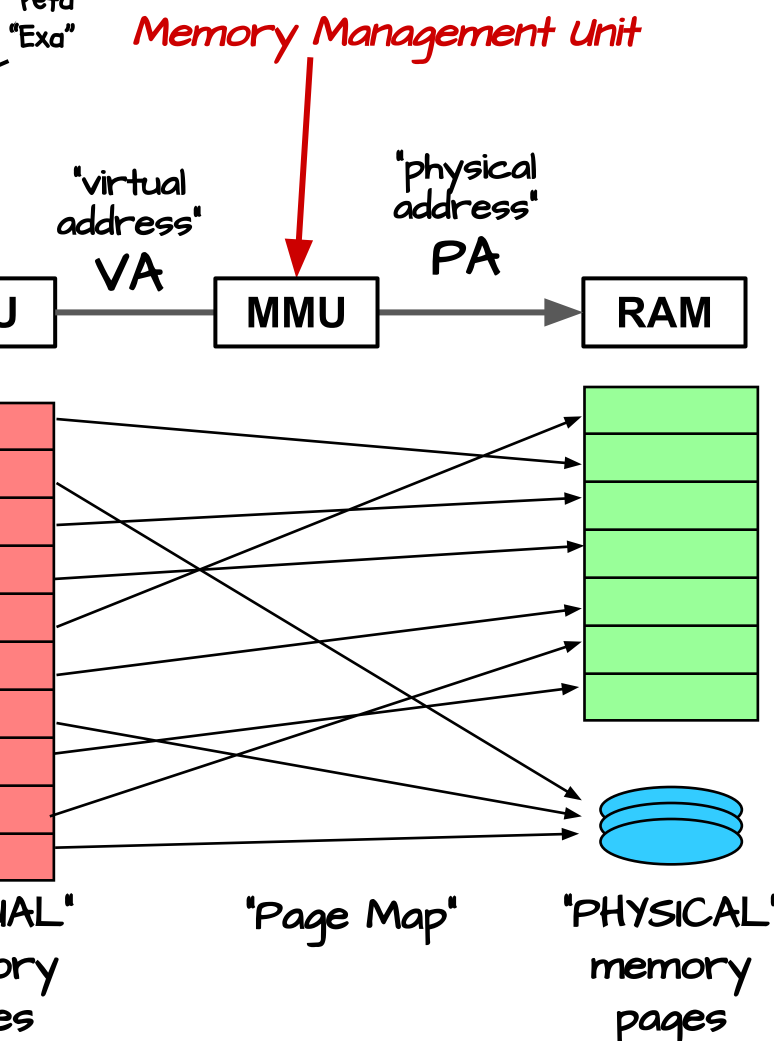


2^{30} "Giga"
2^{40} "Tera"
2^{50} "Peta"
2^{60} "Exa"

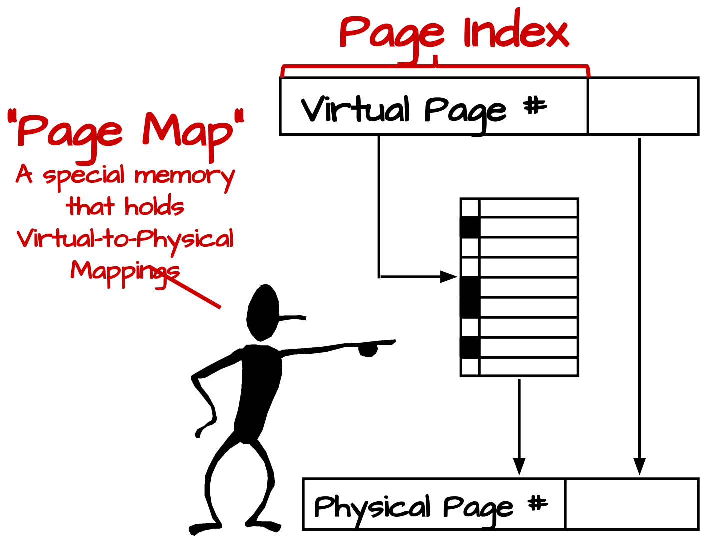
ACTIVE USAGE: small fraction
(2^{28} bytes?)

Actual HARDWARE:
- 2^{31} (2G) bytes of RAM
- 2^{39} (500G) bytes of DISK…
  … maybe more, maybe less!

ELEMENTS OF DECEIT:
- Partition memory into manageable chunks-- "Pages" (4K-8K-16K-64K)
- MAP a few to RAM, assign others to DISK
- Keep "HOT" pages in RAM.



Simple Page Map Design

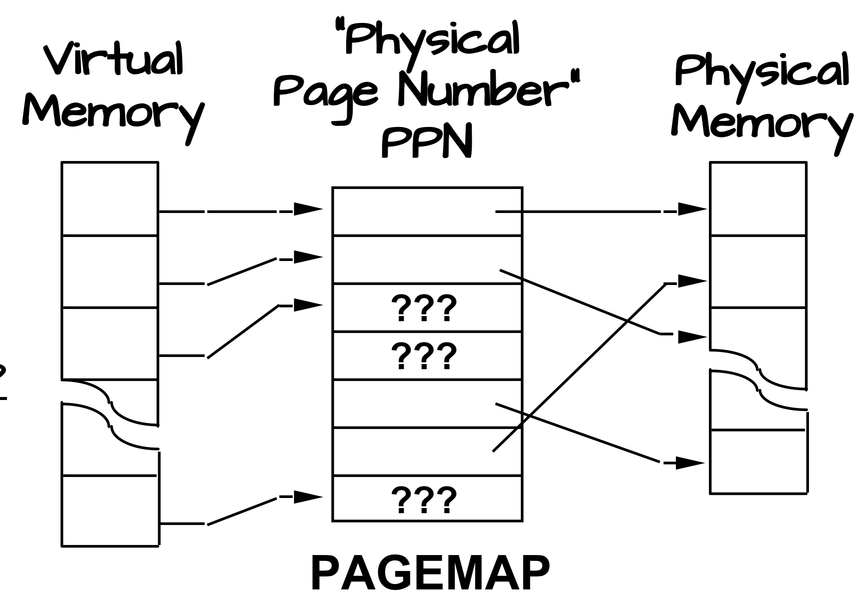


FUNCTION: Given Virtual Address,

- Map to PHYSICAL address

OR

- Cause PAGE FAULT allowing page replacement



Why use HIGH address bits to index pages?

… LOCALITY.

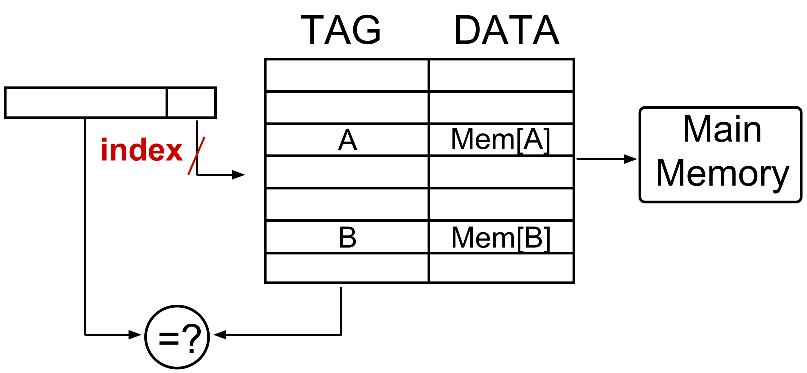
Keeps related data on same page.

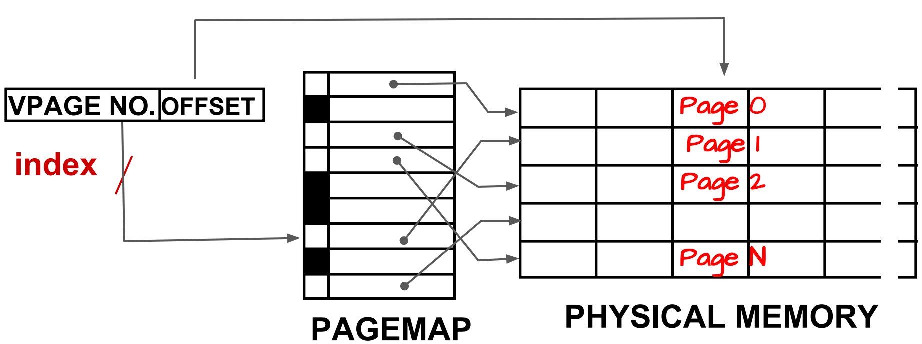
Why use LOW address bits to index cache lines?

… LOCALITY.

Keeps related data from competing for same cache lines.

Virtual Memory vs. Cache





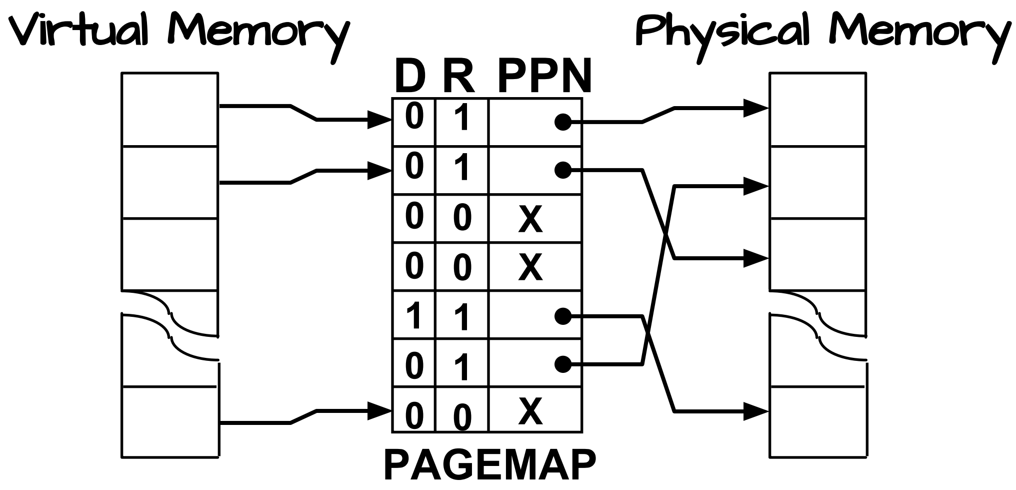
CACHE:

Relatively short blocks (16-64 bytes)
Few lines: scarce resource
miss time: 3x-20x hit time

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
- Vpage mapping is determined by an index (i.e. "direct-mapped" w/o tag) data in physical memory

Virtual Memory: A H/W view

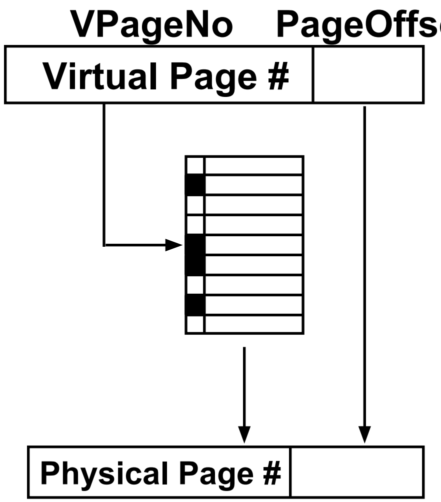


Pagemap Characteristics:

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

Virtual Memory: A S/W view

Problem: Translate
VIRTUAL ADDRESS
to PHYSICAL ADDRESS




```
int VtoP(unsigned int address) {
   unsigned int VPageNo = address>>p;
   unsigned int PageOffset = address & ((1<<p)-1);
   if (R[VPageNo] == 0)
      PageFault(VPageNo);
   return (PPN[VPageNo]<<p)|PageOffset;
}

/* 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 H/W – S/W Balance

IDEA:

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

hardware

```
int VtoP(unsigned int address) {
 unsigned int VPageNo = address>>p;
 unsigned int PageOffset = address&((1<<p)-1);
 if (R[VPageNo] == 0)PageFault(VPageNo);
 return (PPN[VPageNo]<<p)|PageOffset;
}
```

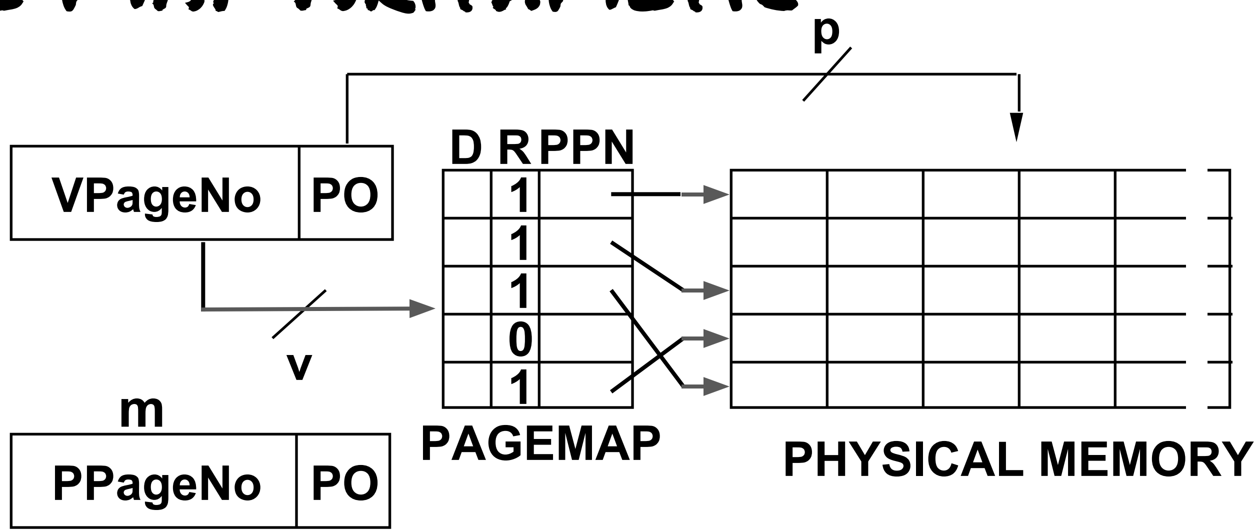
software

```
/* 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 is interrupted ("suspended");
- PageFault(…) is called;
- On return from PageFault; running program can continue

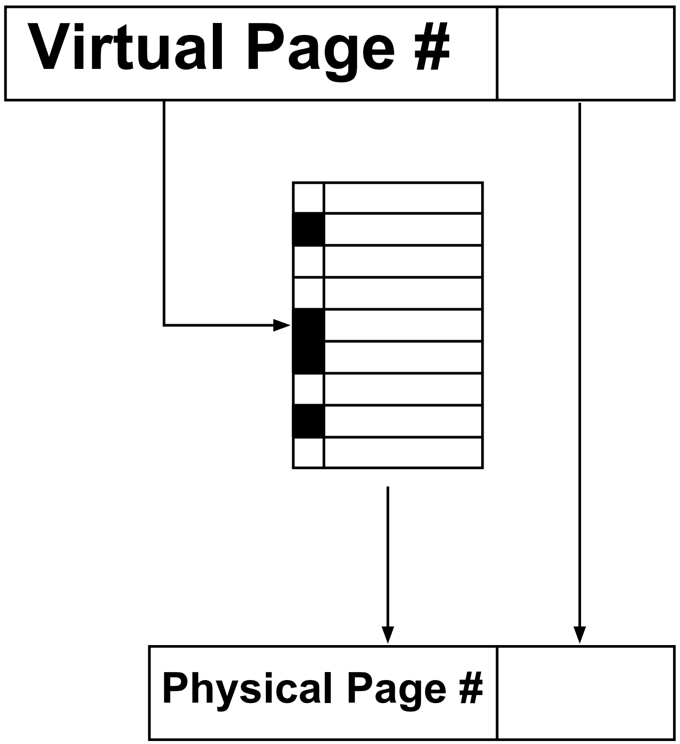
Page Map Arithmetic



(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: 4K – 128K bytes
Typical (v+p): 32 or 64 bits
Typical (m+p): 28 – 34 bits
(256 MB – 16 GB)

Example: Page Map Arithmetic



SUPPOSE...

32-bit Virtual address

2^{14} page size (16 KB)

2^{28} RAM (256 MB)

THEN:

\# Physical Pages = $2^{28}/2^{14} = 16384$

\# Virtual Pages = $2^{32}/2^{14} = 2^{18}$

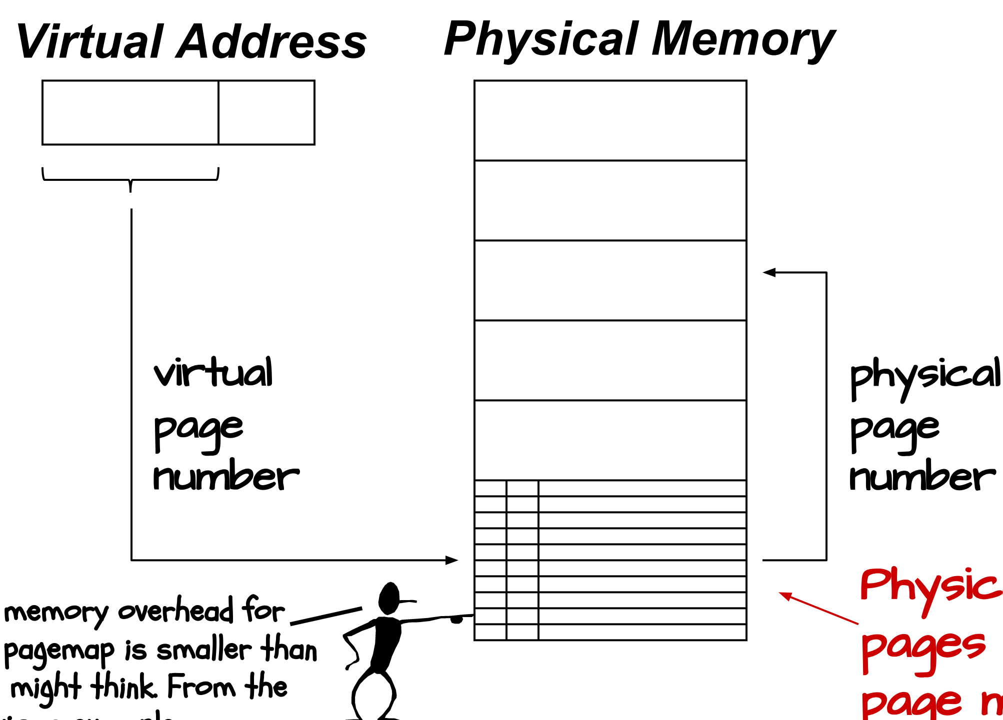
\# Page Map Entries = 262,144

Use SRAM for page map??? OUCH!

RAM-Resident Page Maps

SMALL page maps can use dedicated RAM...
but, gets this approach gets expensive for big ones!

SOLUTION: Move page map into MAIN MEMORY:



Physical memory pages that hold page map entries

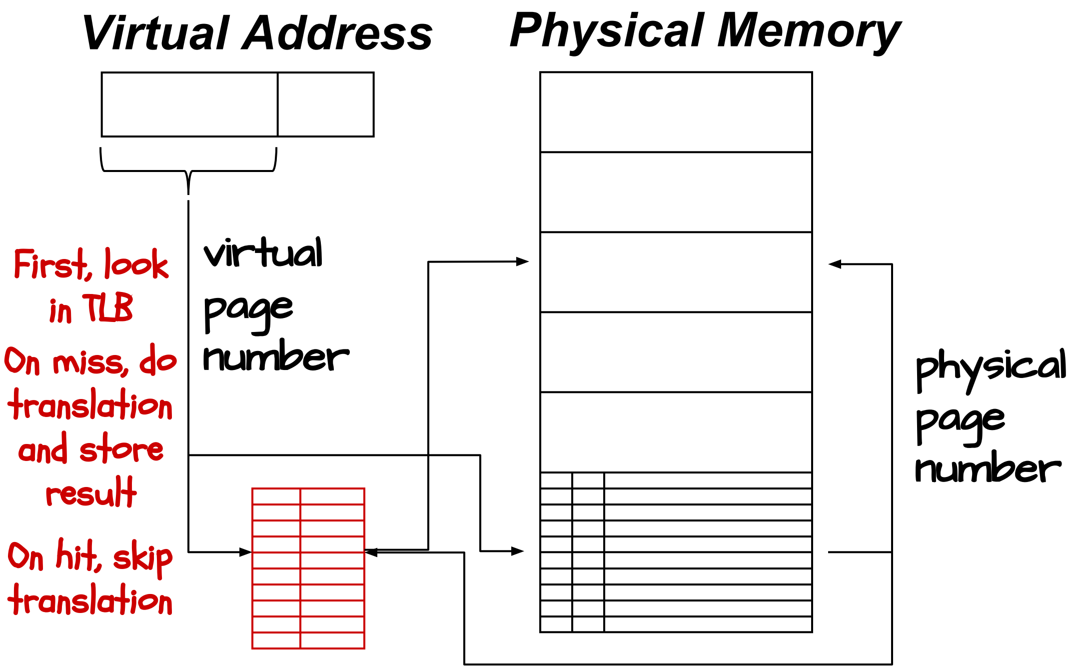
PROBLEM:
Each memory reference now takes 2 accesses to physical memory!
1) Load VPN → PPN
2) Load Mem[PPN | PO]

The memory overhead for the pagemap is smaller than you might think. From the previous example:
$4*2^{18}/2^{28} = 0.4$ %

Translation Look-aside Buffer (TLB)

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

SOLUTION: a special CACHE of recently used page map entries



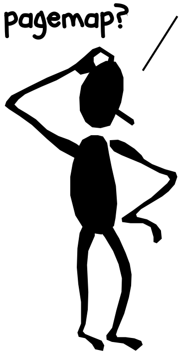
IDEA:
LOCALITY in memory reference patterns → SUPER locality in References to page map

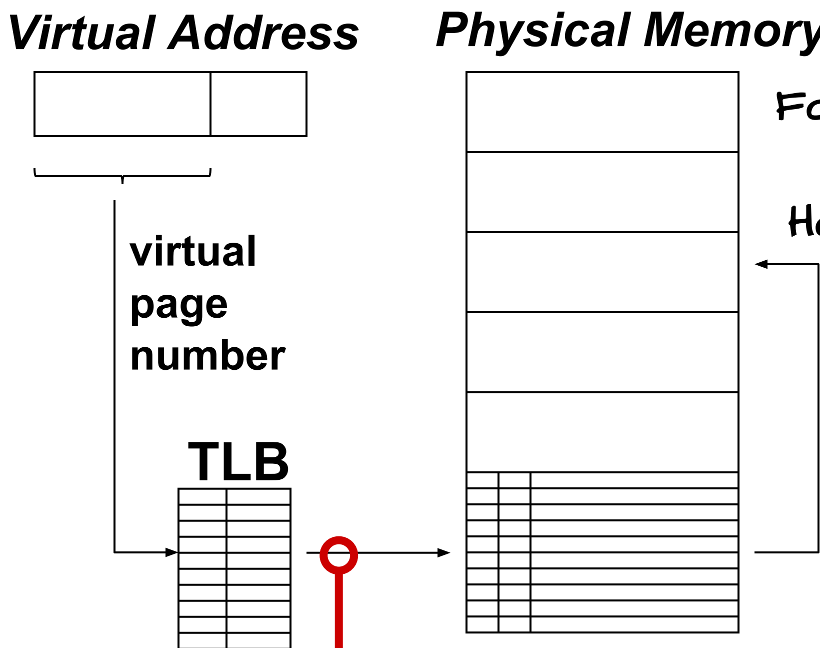
VARIATIONS:
- sparse page map storage
- paging the page map

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

Optimizing Sparse Page Maps

For large Virtual Address spaces only a small percentage of page table entries contain "Mappings". This is because some address ranges are never used by the application. How can we save space in the pagemap?





For Example:

VA 2^{64}, 8Kb pages, PA 2^{36}

How large of a page table?

$2^{64-13} = 4 \times 2^{51} = 2^{53}$ bytes

At most, how many could have a resident mapping?

$2^{36-13} = 2^{23}$

$2^{23}/2^{51} = 3.7 \times 10^{-9}$

On TLB miss:

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

Another good reason to handle page misses in SW

Multilevel Page Maps

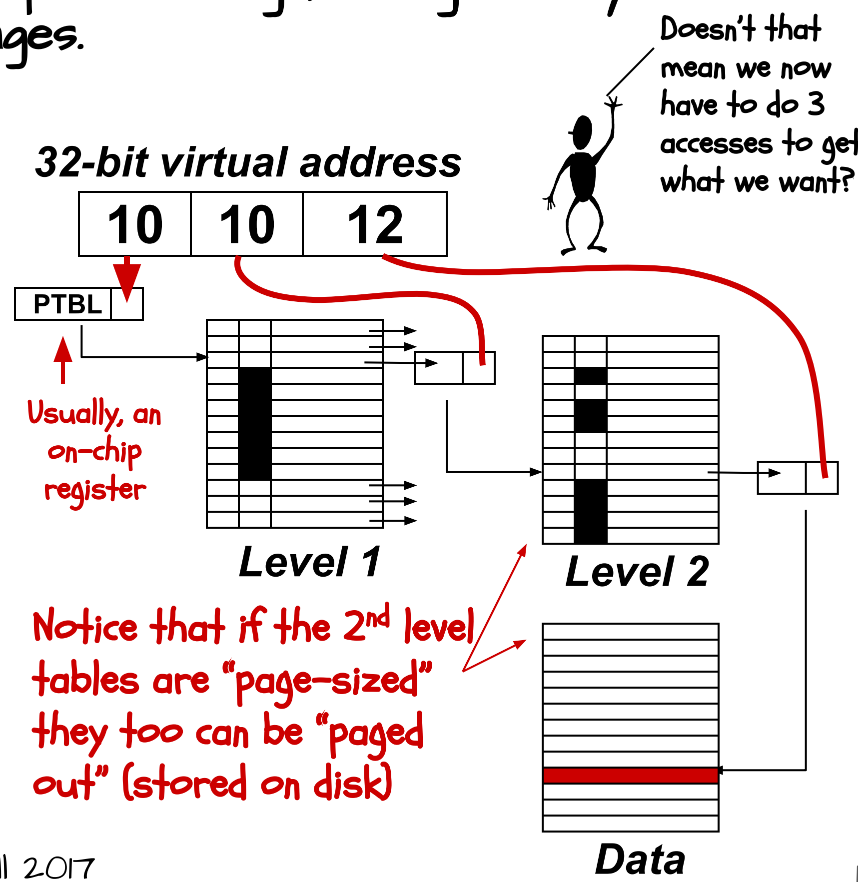
Given a HUGE virtual memory, the cost of storing all of the page map entries in RAM may STILL be too expensive…

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?).

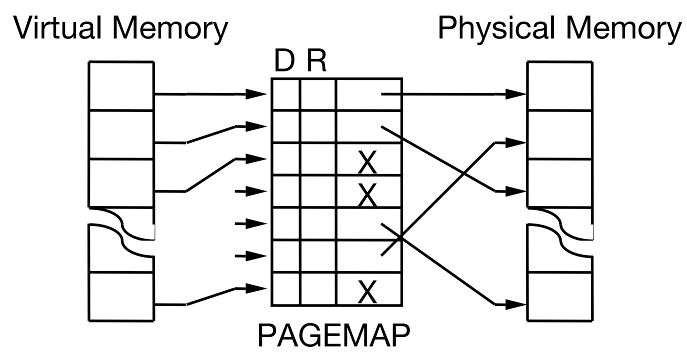
A 2-level look-up increases the size of the worse-case page table slightly. However, if a first level entry has its non-resident bit set it saves large amounts of memory.



CONTEXTS

A CONTEXT is a complete set of mappings from VIRTUAL to PHYSICAL addresses, as dictated by the full contents of the page map:

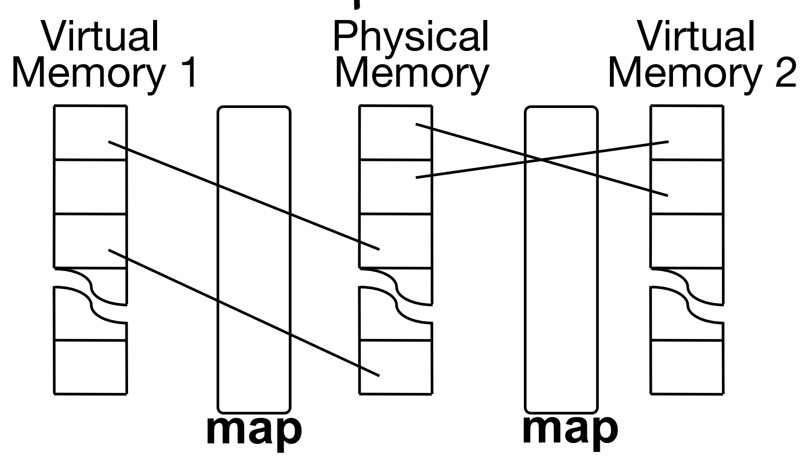
We might like to support multiple VIRTUAL to PHYSICAL Mappings and, thus, multiple Contexts.



This enables several programs to be simultaneously loaded into main memory, each with it's own "address space":

"Context Switch":
Reload the page map!

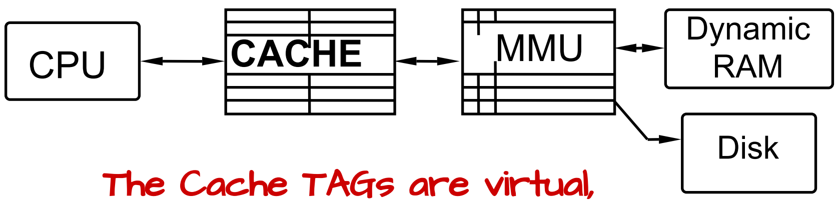
You end up with pages from different applications simultaneously in memory.



Using Caches with Virtual Memory

Virtual Cache

Tags match virtual addresses



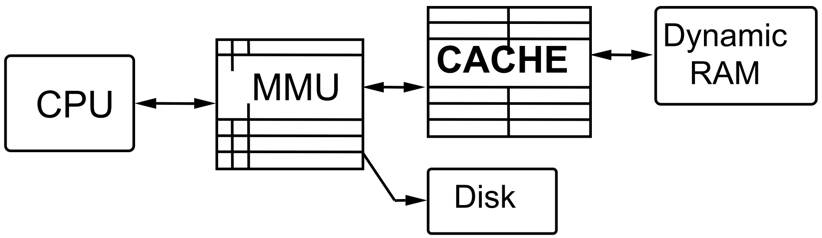
The Cache TAGs are virtual, they represent addresses before translation.

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

Physical Cache

Tags match physical addresses

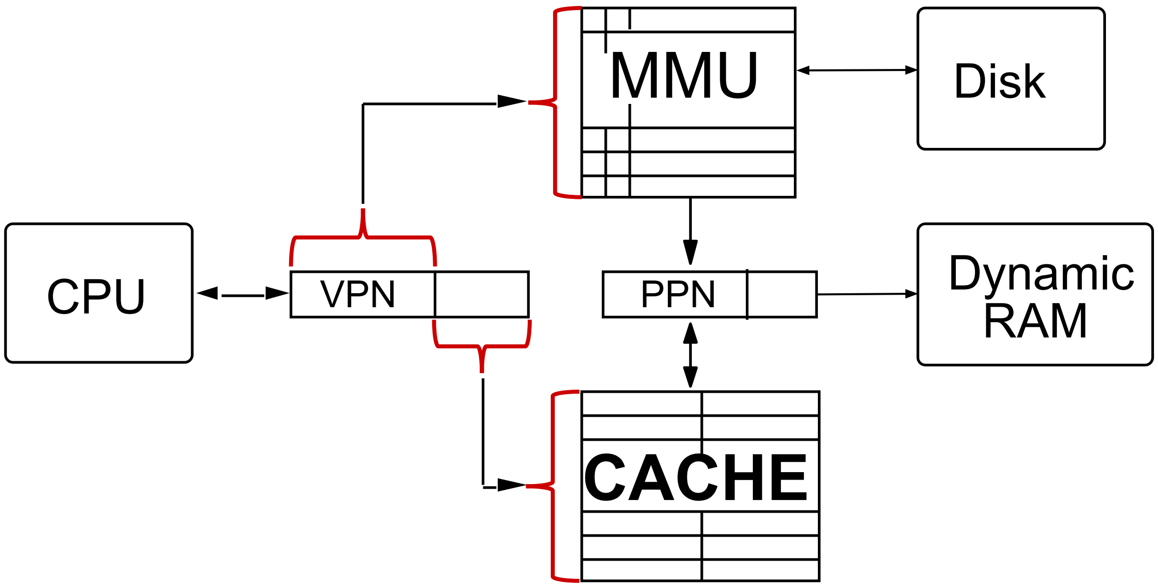
These TAGs are physical, they hold addresses after translation.



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

Physically addressed Caches are the trend, because they better support parallel processing

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 / small page size → go with more associativity

Summary

Virtual Memory:

Makes a small PHYSICAL memory appear to be a large VIRTUAL one

Break memory into manageable chunks called PAGES

Pagemap:

A table for mapping Virtual-to-Physical pages

Each entry has Resident, Dirty, and Physical Page Number

Can get large if virtual address space is large

Store in main memory

TLB - Translation Lookaside Buffer:

A pagemap "cache"

Contexts:

Sets of virtual-to-physical mapping that allow pages from multiple applications to be in physical memory simultaneously (even if they have the same virtual addresses)