

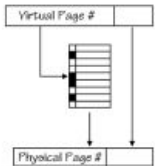


VIRTUAL MEMORY

My memory is full! In the last 3 weeks, we've built a CPU, pipelined it, added a cache, and now page faults.

I've been lost since NAND gates...

Problem: Translate
VIRTUAL ADDRESS
to PHYSICAL ADDRESS

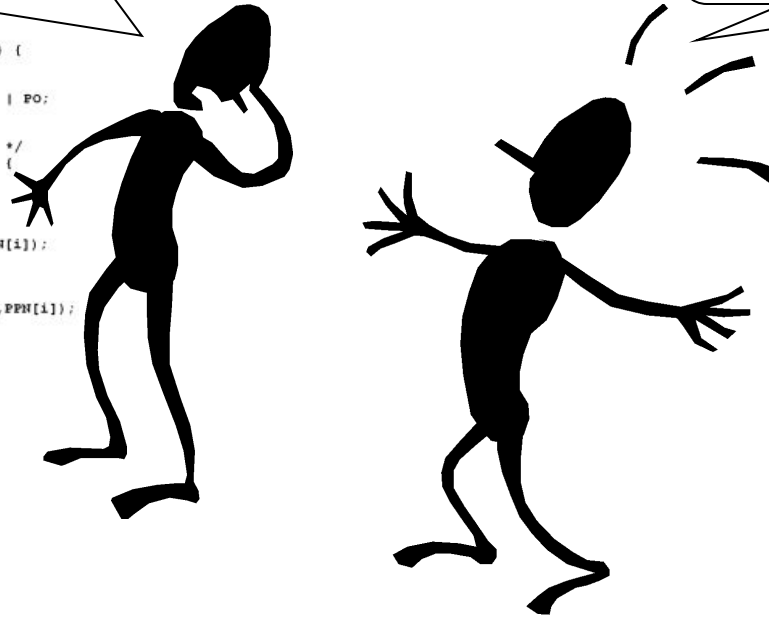


```
int VtoP(int VPageNo, int PO) {
    if (R[VPageNo] == 0)
        PageFault(VPageNo);
    return (PPN[VPageNo] << p) | PO;
}

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

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

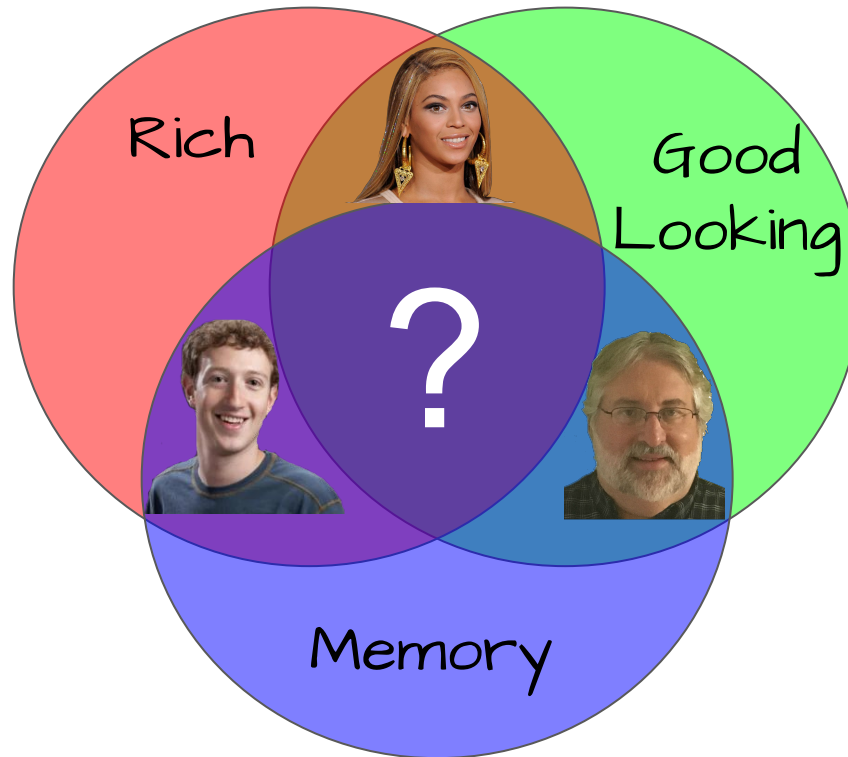
    PPN[VPageNo] = PPN[i];
    ReadPage(DiskAdr[VPageNo], PPN[i]);
    R[VPageNo] = 1;
    D[VPageNo] = 0;
}
```



- 1) I will release and go over the 2nd midterm when we get back
- 2) Problem Set #6 is due next Tuesday
- 3) Extra Office hours Monday 11/28, 2-4pm
- 4) Final Exam time is fixed, Thursday 12/8, noon-3pm



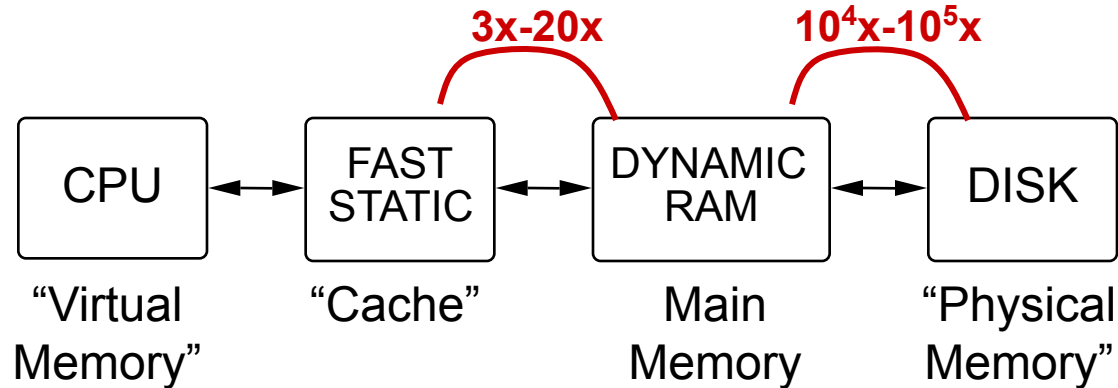
**YOU CAN NEVER BE TOO RICH, TOO GOOD
LOOKING, OR HAVE TOO MUCH MEMORY!**



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 "size" at near RAM speeds?

VIRTUAL MEMORY

- 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} (1BE) bytes?)

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

Actual HARDWARE:

- 2^{33} (8G) bytes of RAM
- 2^{39} (500G) bytes of DISK...
... maybe more, maybe less!

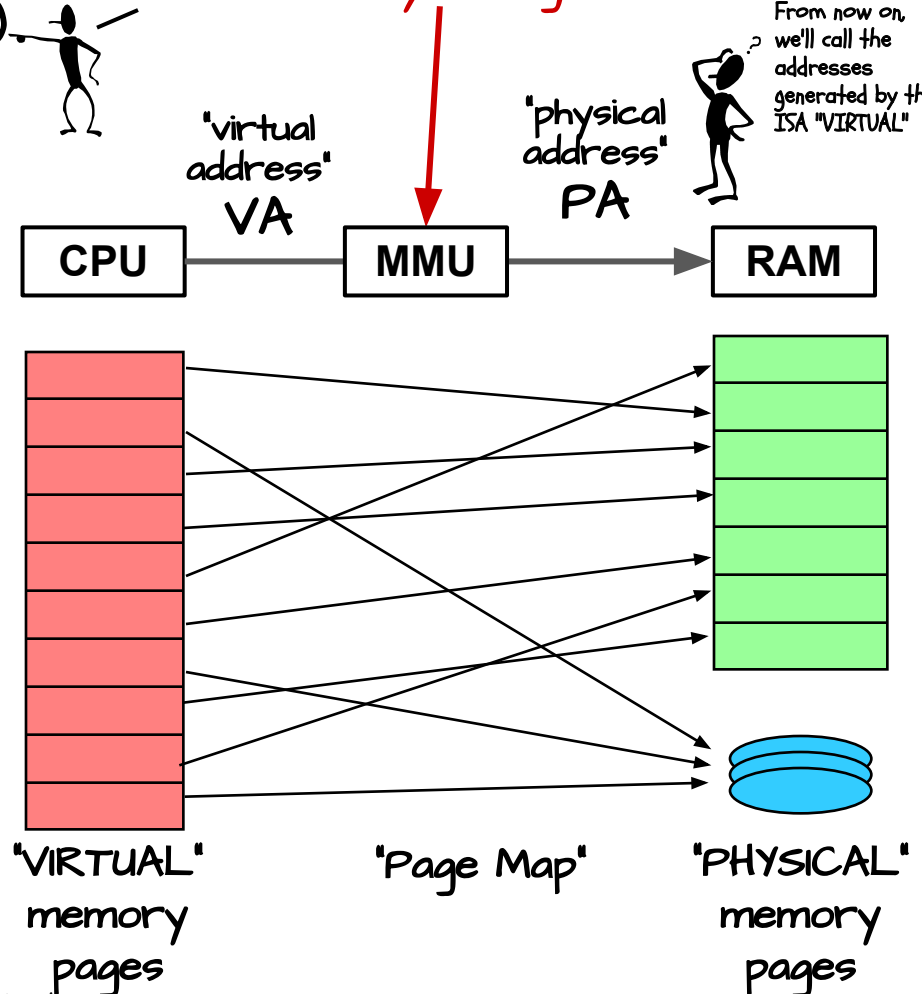
ELEMENTS OF DECEIT:

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

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

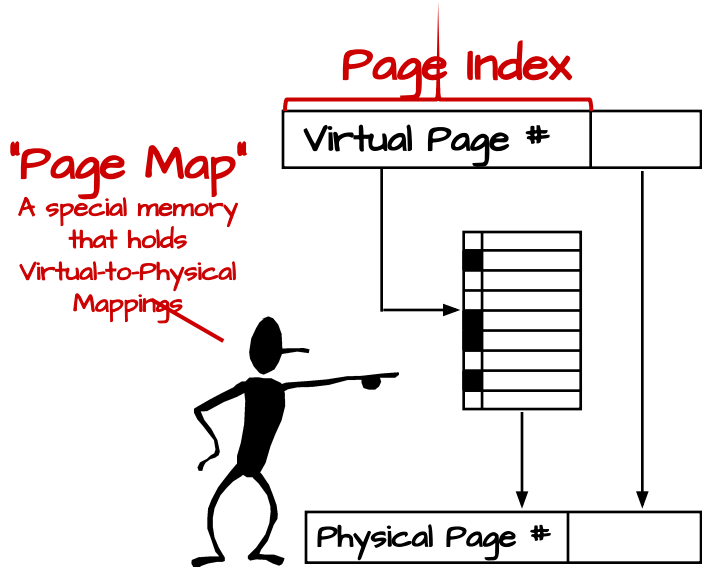
Memory Management Unit

From now on, we'll call the addresses generated by the ISA "VIRTUAL"





SIMPLE PAGE MAP DESIGN

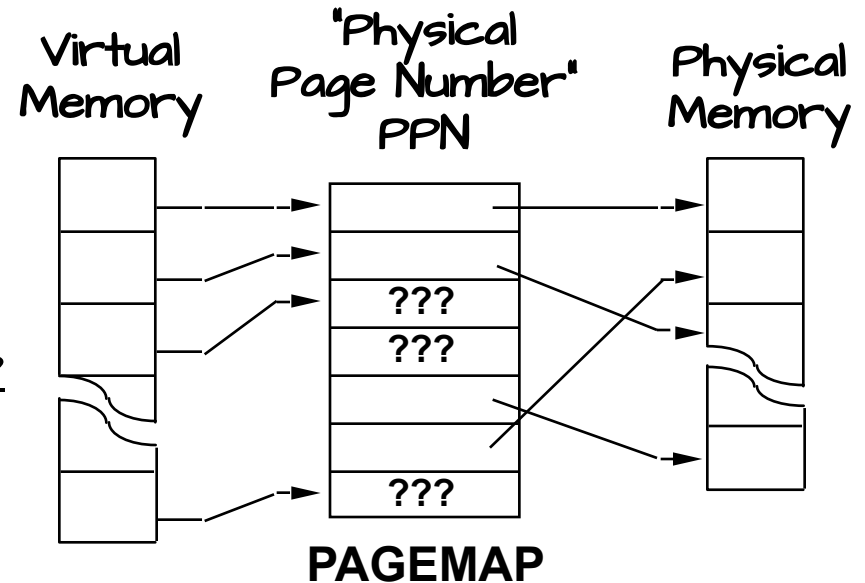


FUNCTION: Given Virtual Address,

- 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.

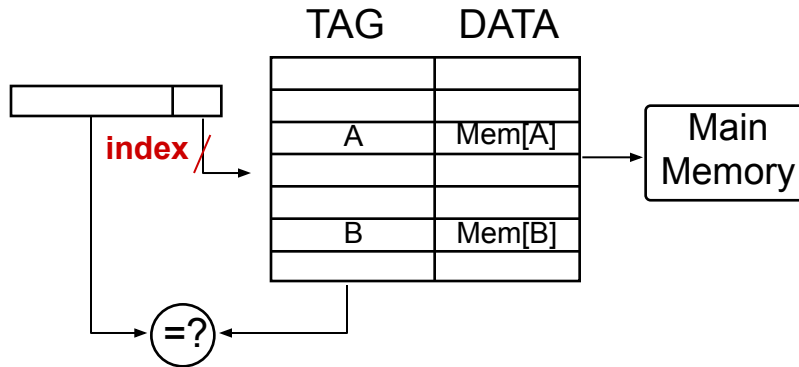
Why use LOW address bits to index cache lines?

... LOCALITY.

Keeps related data from competing for same cache lines.



VIRTUAL MEMORY VS. CACHE

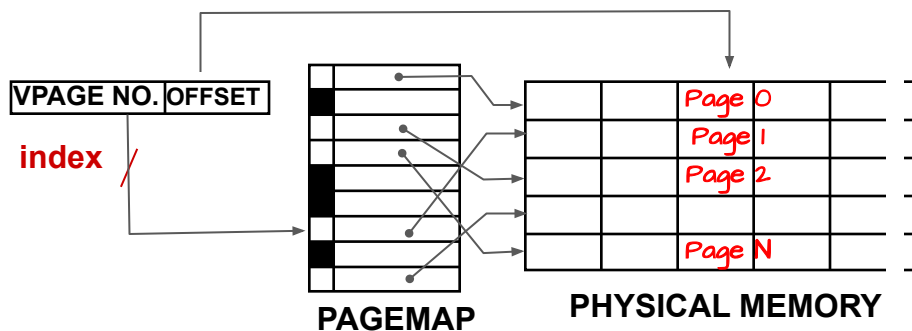


CACHE:

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

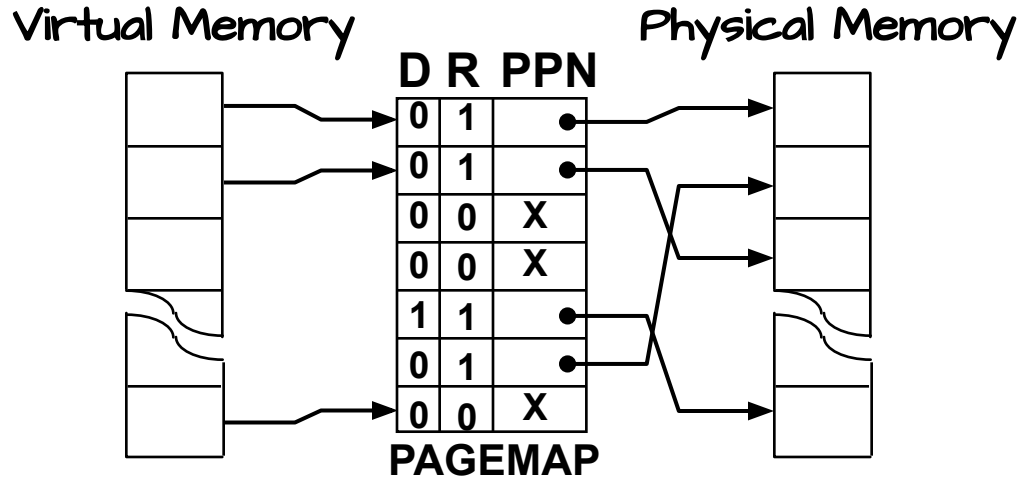
VIRTUAL MEMORY:

- Disk: long latency, fast xfer
 - miss time: $\sim 10^5$ 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



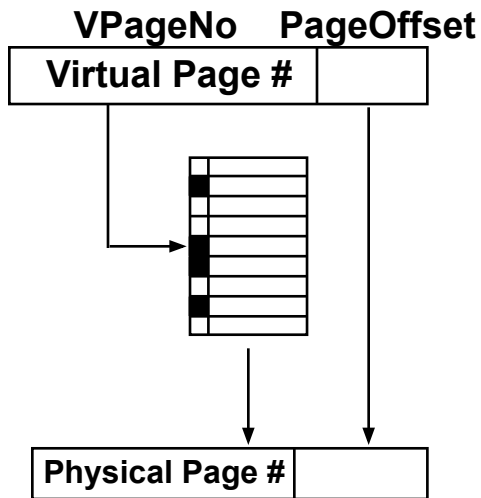
Pagemap Characteristics:

- 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

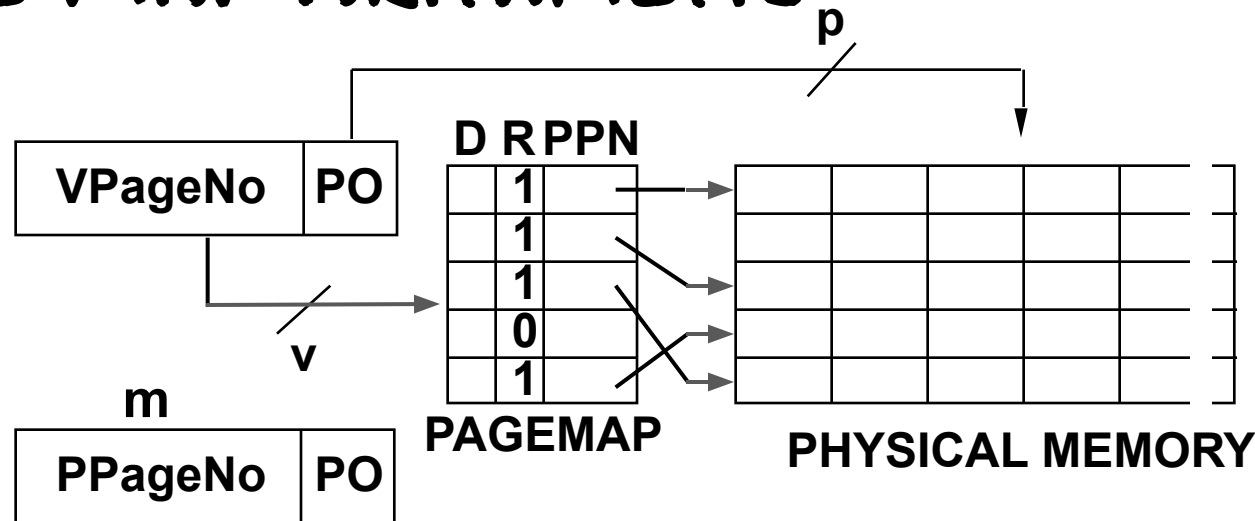
```
/* Handle a missing page... */
void PageFault(int VPageNo) {
    int i = SelectLRUPage();
    if (D[i] == 1) WritePage(DiskAdr(i),PPN[i]);
    R[i] = 0;
    PA[VPageNo] = PPN[i];
    ReadPage(DiskAdr(VPageNo),PPN[i]);
    R[VPageNo] = 1;
    D[VPageNo] = 0;
}
```

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

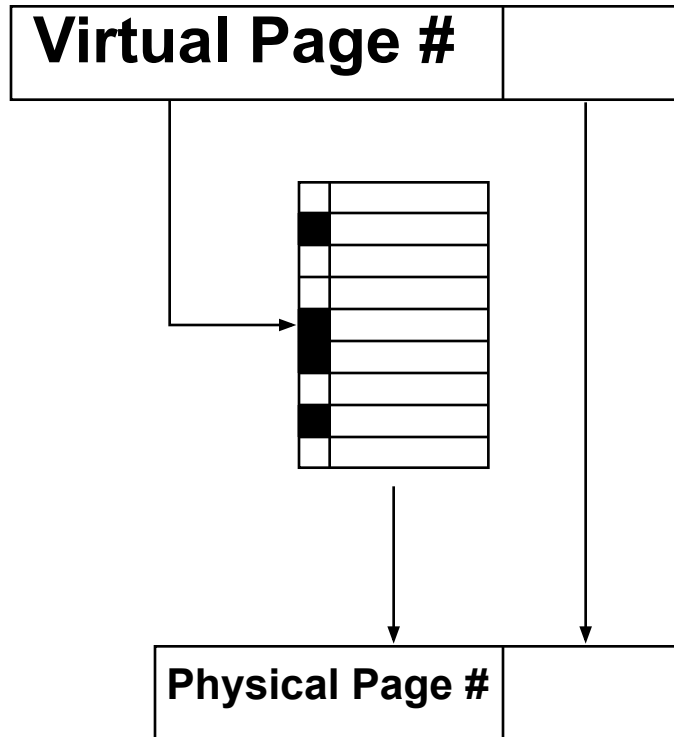


$(v + p)$ bits in virtual address
 $(m + p)$ bits in physical address
 2^v number of VIRTUAL pages
 2^m number of PHYSICAL pages
 2^p bytes per physical page
 2^{v+p} bytes in virtual memory
 2^{m+p} bytes in physical memory
 $(m+2)2^v$ bits in the page map

Typical page size: 4K – 128K bytes
Typical $(v+p)$: 32 or 48 or 64 bits
Typical $(m+p)$: 30 – 39 bits
(1GB – 512 GB)



EXAMPLE: PAGE MAP ARITHMETIC



SUPPOSE...

32-bit virtual address

2^{14} page size (16 KB)

2^{28} RAM (256 MB)

THEN:

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

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

$$\# \text{ Page Map Entries} = \underline{262,144}$$

Use SRAM for page map???

OUCH!

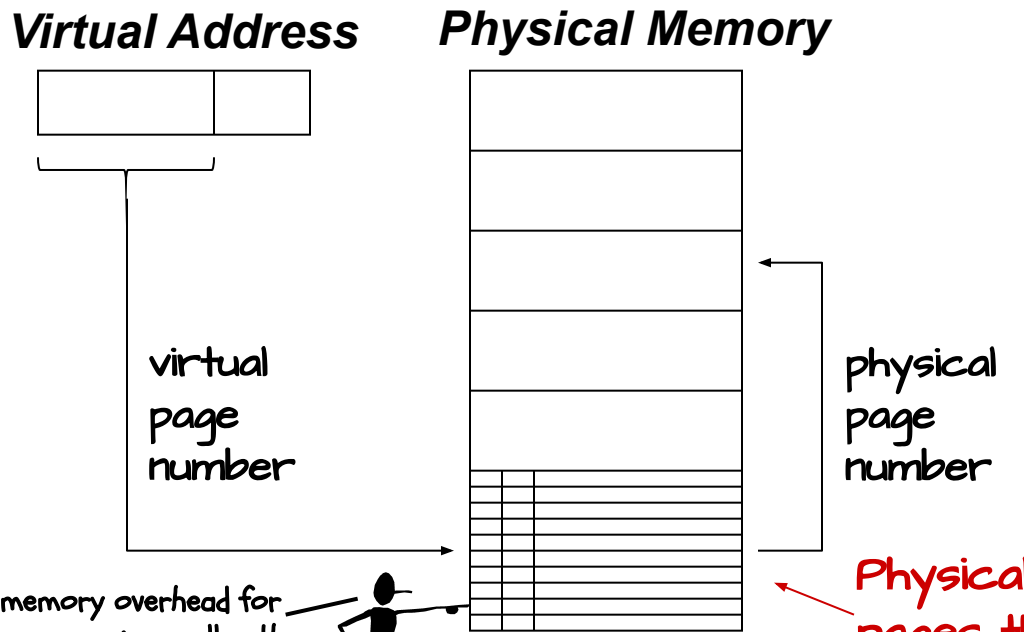


RAM-RESIDENT PAGE MAPS

SMALL page maps can use dedicated RAM...

but, getting this approach gets expensive for big ones!

SOLUTION: Move page map into MAIN MEMORY:



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:

$$4 \cdot 2^{18} / 2^{28} = 0.4\%$$

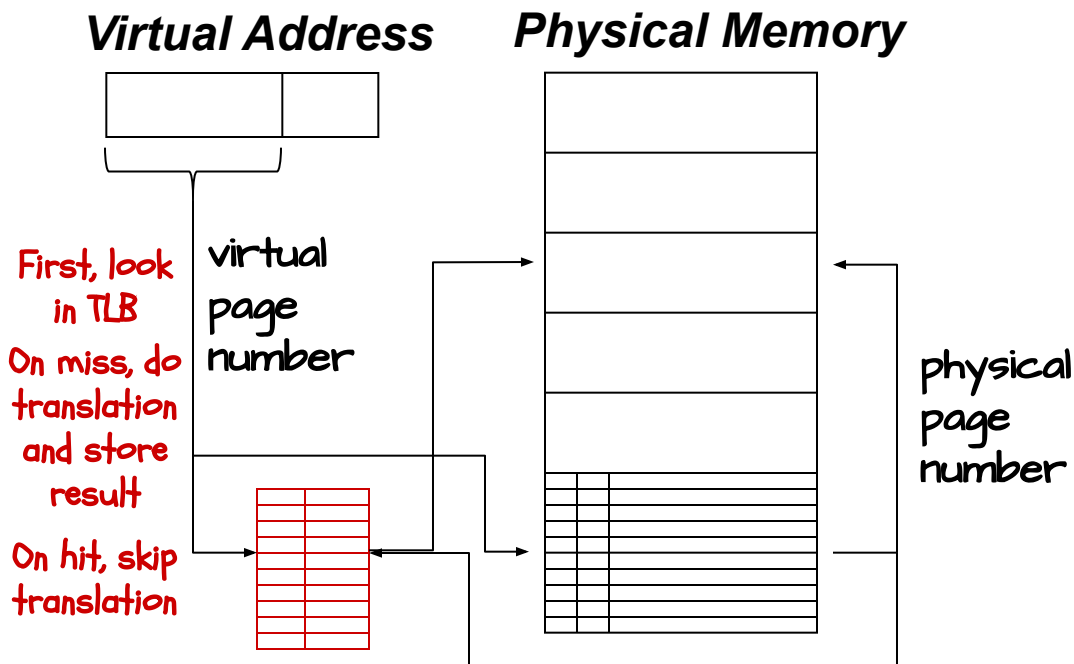
TRANSLATION LOOK-ASIDE BUFFER (TLB)



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:

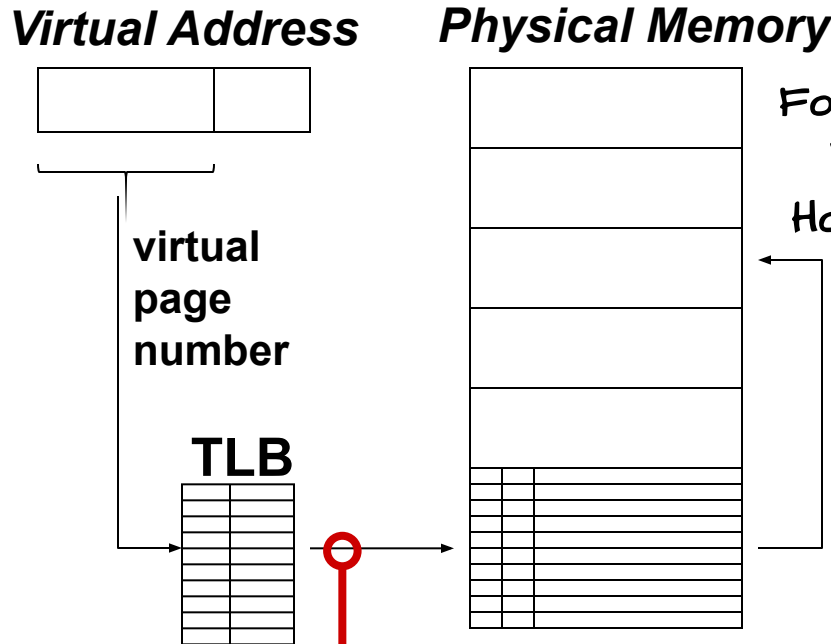
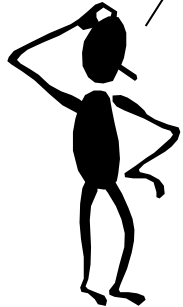
- sparse page map storage
- paging the page map

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?

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

physical page number

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

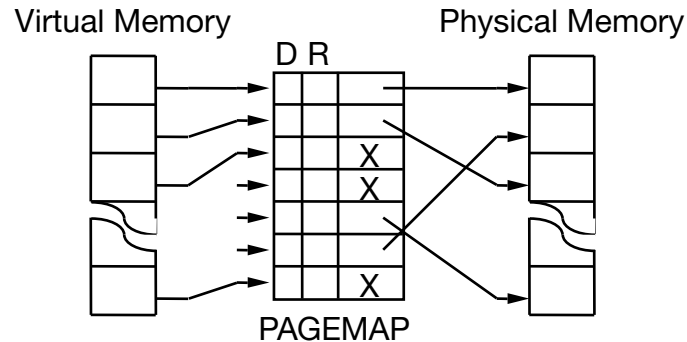


CONTEXTS

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



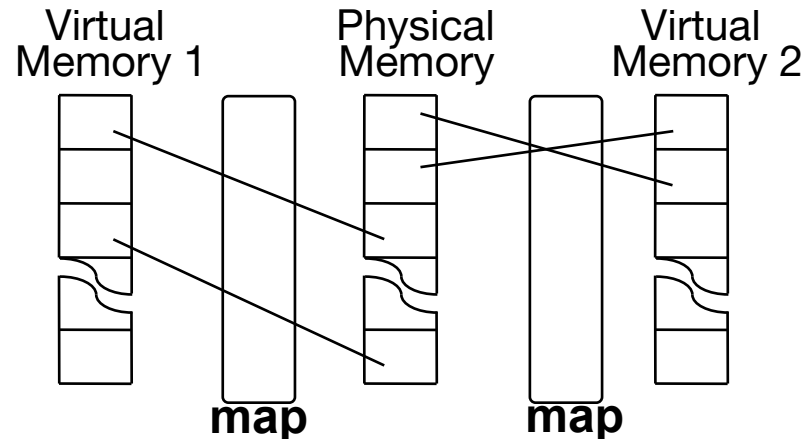
We might like to support multiple VIRTUAL to PHYSICAL Mappings and, thus, multiple Contexts.



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

"Context Switch":
Reload the page map!

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

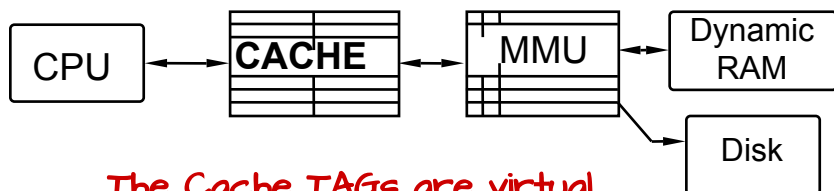


USING CACHES WITH VIRTUAL MEMORY



Virtual Cache

Tags match virtual addresses



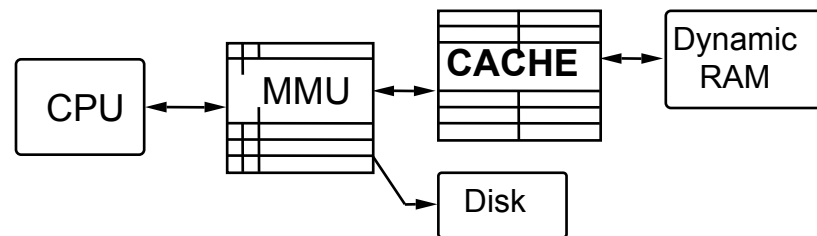
The Cache TAGs are virtual, they represent addresses before translation.

- 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.

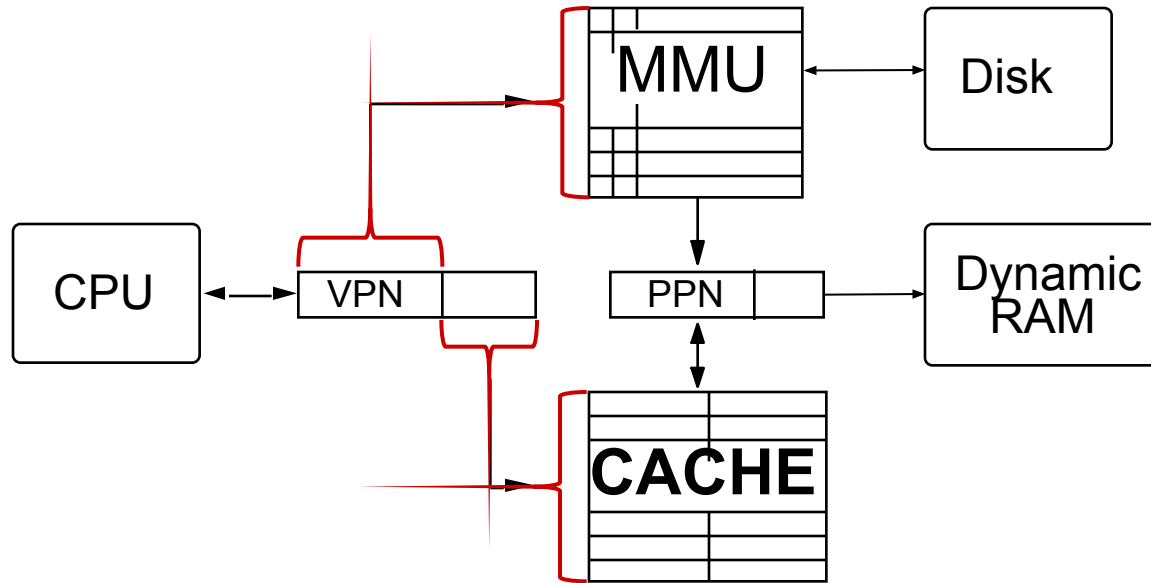


- Avoids stale cache data after context switch
- SLOW: MMU time on HIT

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)