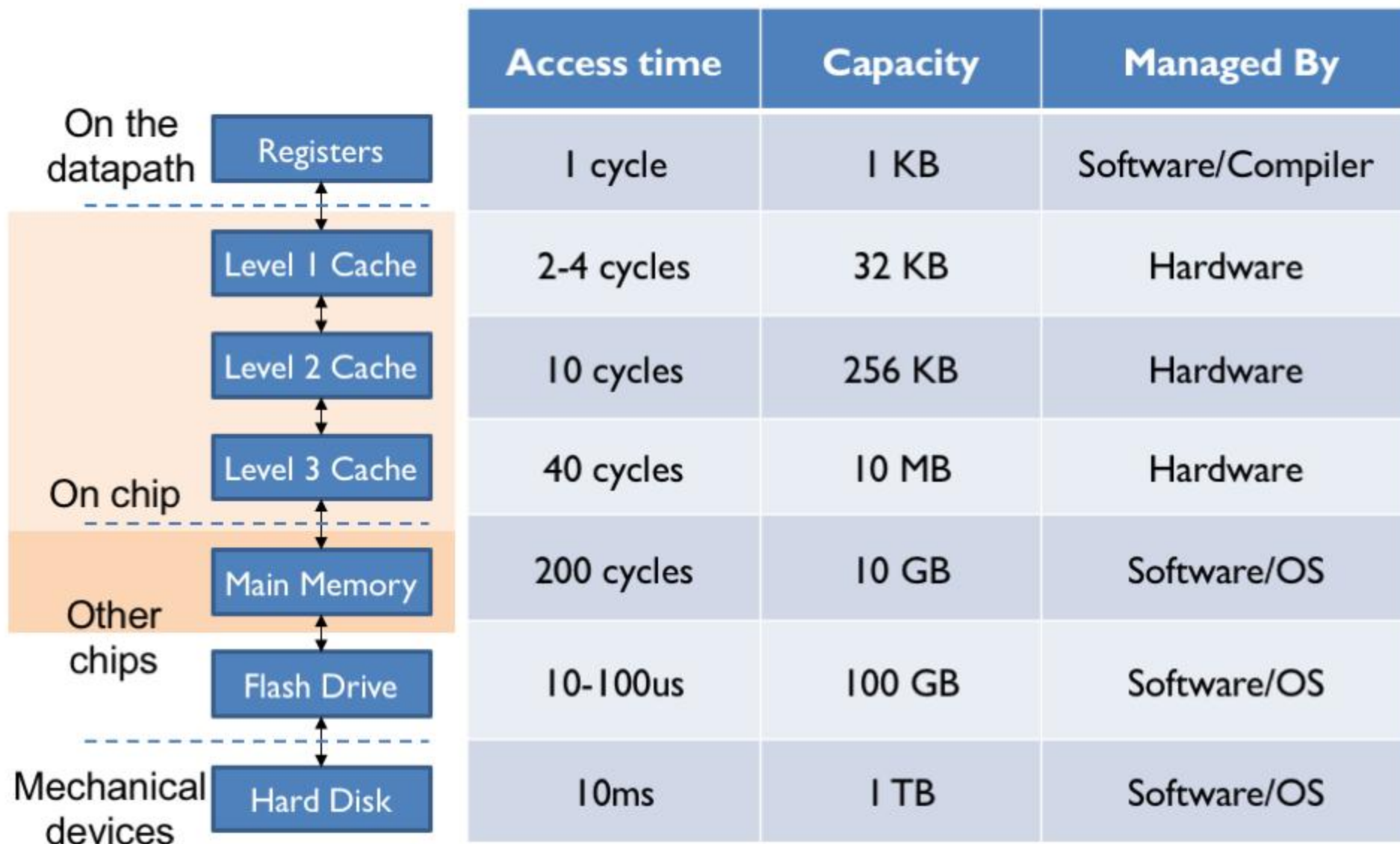


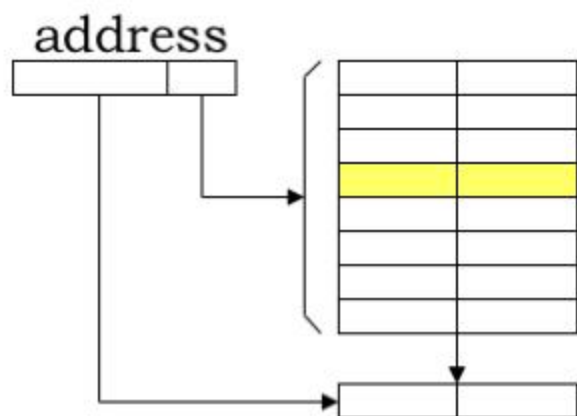
Reminder: A Typical Memory Hierarchy

- Everything is a cache for something else



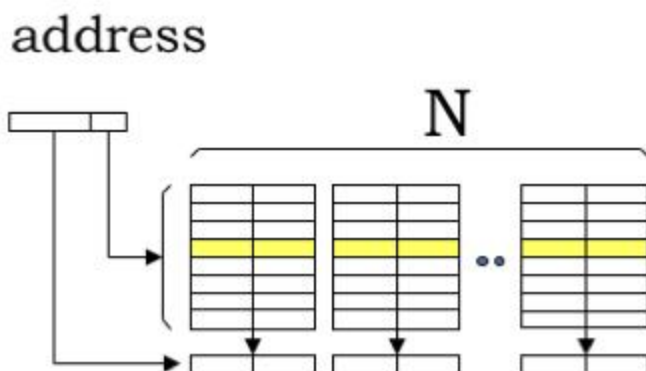
Reminder: Hardware Caches

Direct-mapped



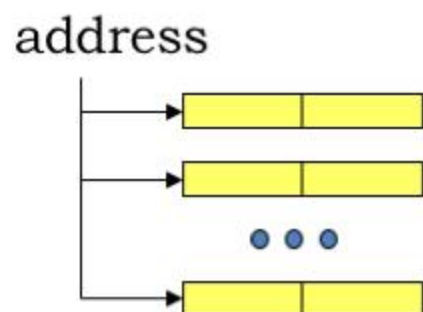
Each address maps to a single location in the cache, given by index bits

N-way set-associative



Each address can map to any of N locations (ways) of a single set, given by its index bits

Fully associative



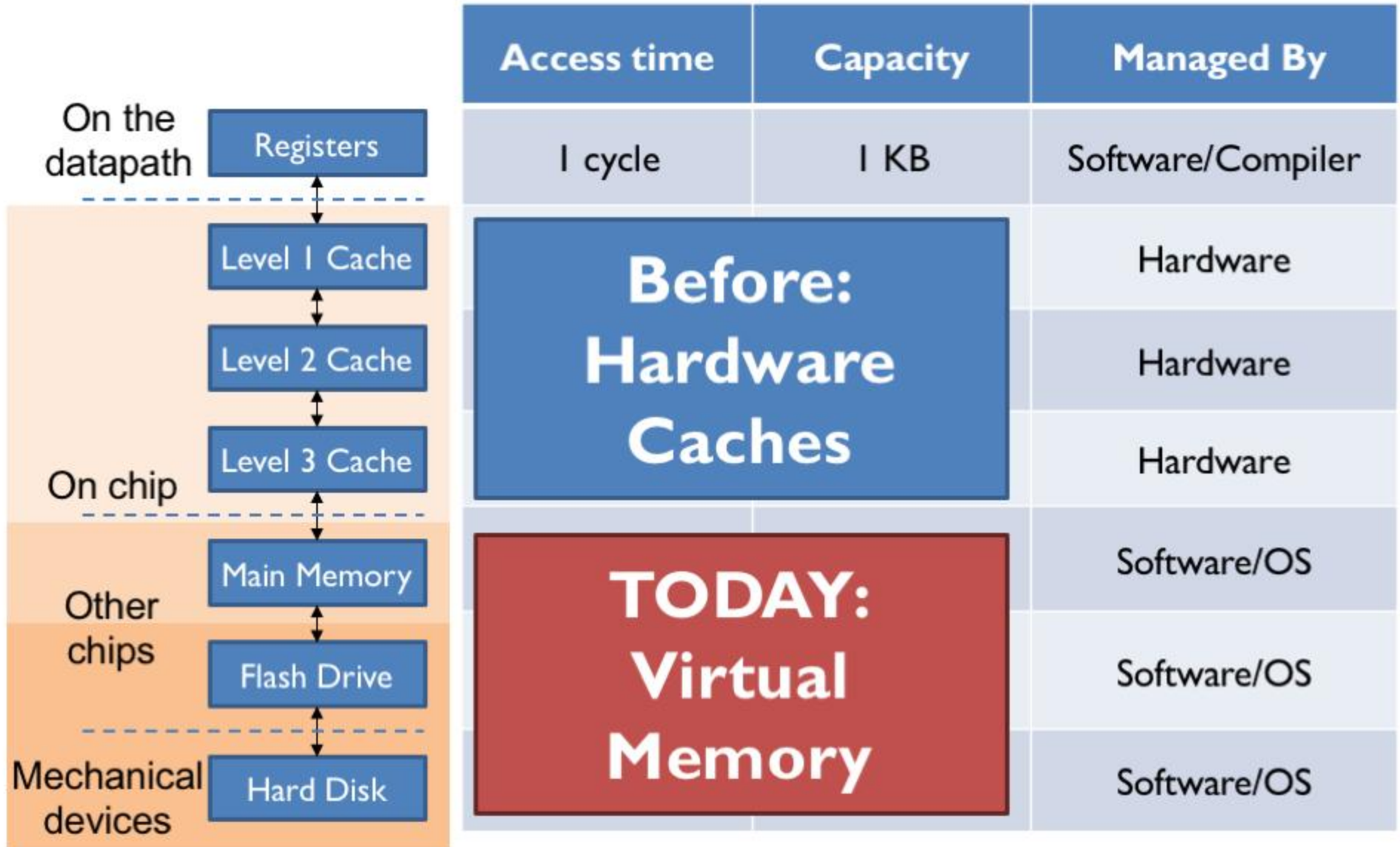
Any location can map to any address

Other implementation choices:

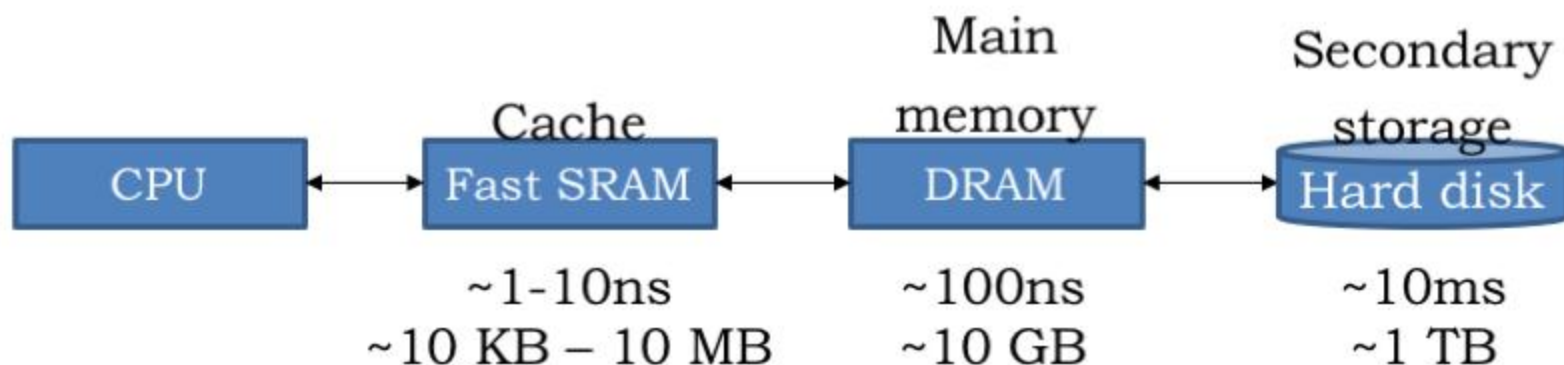
- Block size
- Replacement policy (LRU, Random, ...)
- Write policy (write-through, write-behind, write-back)

Reminder: A Typical Memory Hierarchy

- Everything is a cache for something else



Extending the Memory Hierarchy



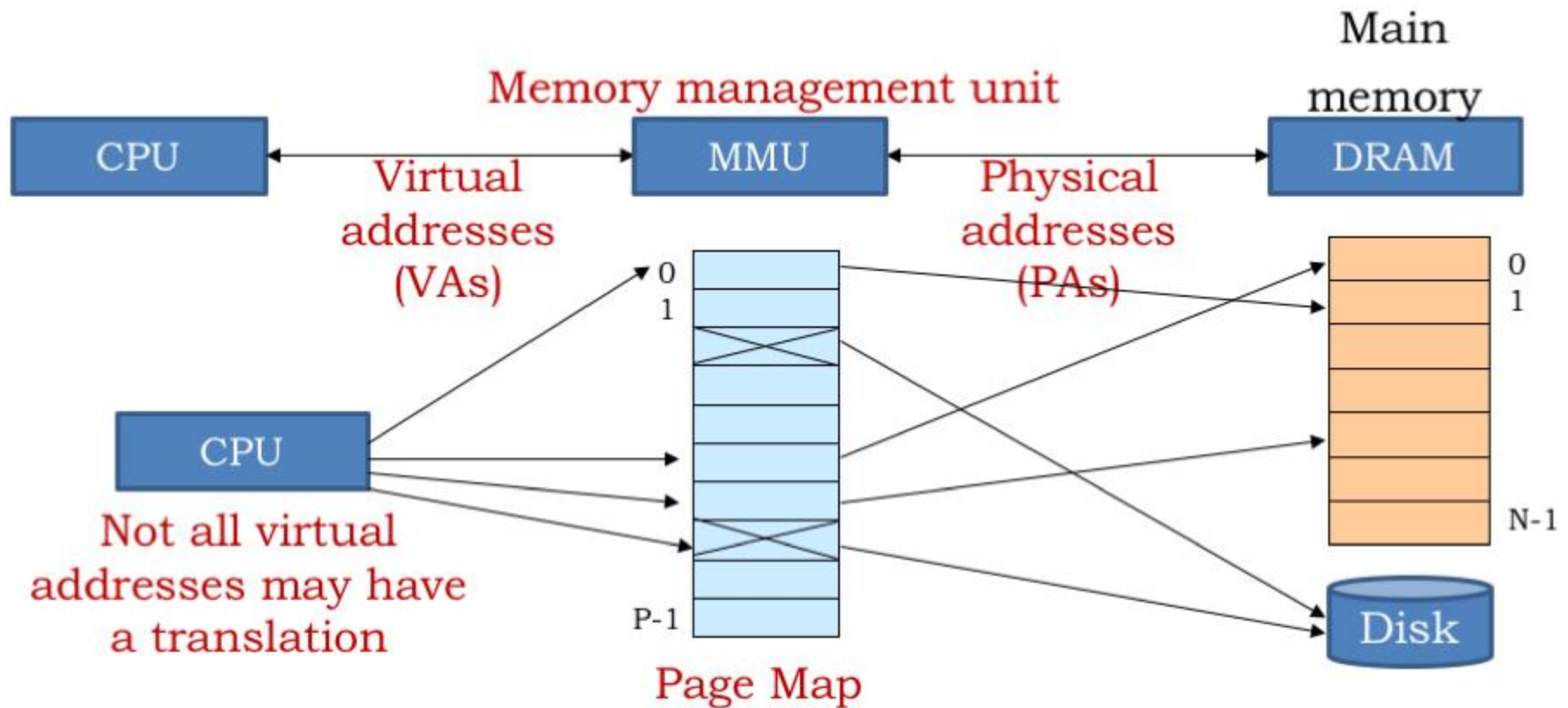
- Problem: DRAM vs disk has much more extreme differences than SRAM vs DRAM
 - Access latencies:
 - DRAM $\sim 10-100\times$ slower than SRAM
 - Disk $\sim 100,000\times$ slower than DRAM
 - Importance of sequential accesses:
 - DRAM: Fetching successive words $\sim 5\times$ faster than first word
 - Disk: Fetching successive words $\sim 100,000\times$ faster than first word
- Result: Design decisions driven by **enormous cost of misses**

Impact of Enormous Miss Penalty

- If DRAM was to be organized like an SRAM cache, how should we design it?
 - Associativity: High, minimize miss ratio
 - Block size: Large, amortize cost of a miss over multiple words (locality)
 - Write policy: Write back, minimize number of writes
- Is there anything good about misses being so expensive?
 - We can handle them in software! What's 1000 extra instructions ($\sim 1\mu\text{s}$) vs 10ms?
 - Approach: Handle hits in hardware, misses in software
 - Simpler implementation, more flexibility

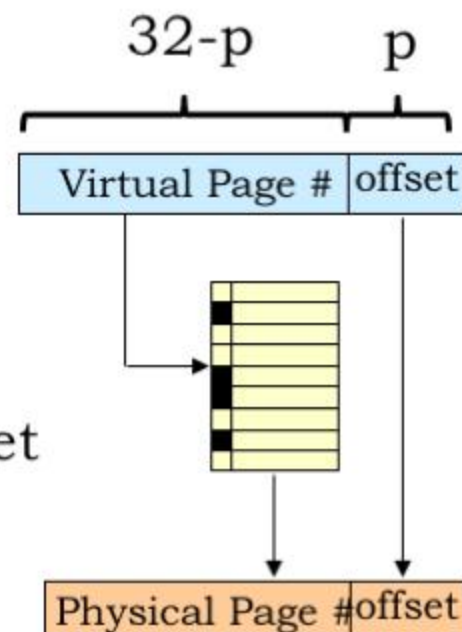
Virtual Memory

- Two kinds of addresses:
 - CPU uses **virtual addresses**
 - Main memory uses **physical addresses**
- Hardware translates virtual addresses to physical addresses via an operating system (OS)-managed table, the **page map**



Virtual Memory Implementation: Paging

- Divide physical memory in fixed-size blocks, called **pages**
 - Typical page size (2^p): 4KB -16 KB
 - Virtual address: Virtual page number + offset bits
 - Physical address: Physical page number + offset bits
 - Why use lower bits as offset?
- MMU maps virtual pages to physical pages
 - Use page map to perform translation
 - Cause a **page fault** (a miss) if virtual page is not resident in physical memory.



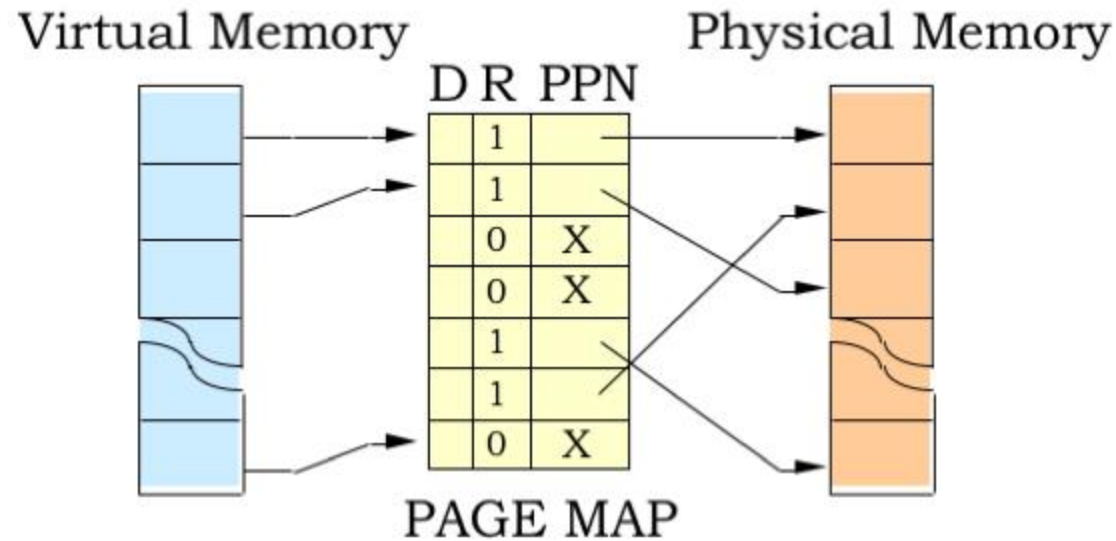
Using main memory as a page cache = *paging* or *demand paging*

Demand Paging

Basic idea:

- Start with all virtual pages in secondary storage, MMU “empty”, ie, there are no pages resident in physical memory
- Begin running program... each VA “mapped” to a PA
 - Reference to RAM-resident page: RAM accessed by hardware
 - Reference to a non-resident page: page fault, which traps to software handler, which
 - Fetches missing page from DISK into RAM
 - Adjusts MMU to map newly-loaded virtual page directly in RAM
 - If RAM is full, may have to replace (“swap out”) some little-used page to free up RAM for the new page.
- Working set incrementally loaded via page faults, gradually evolves as pages are replaced...

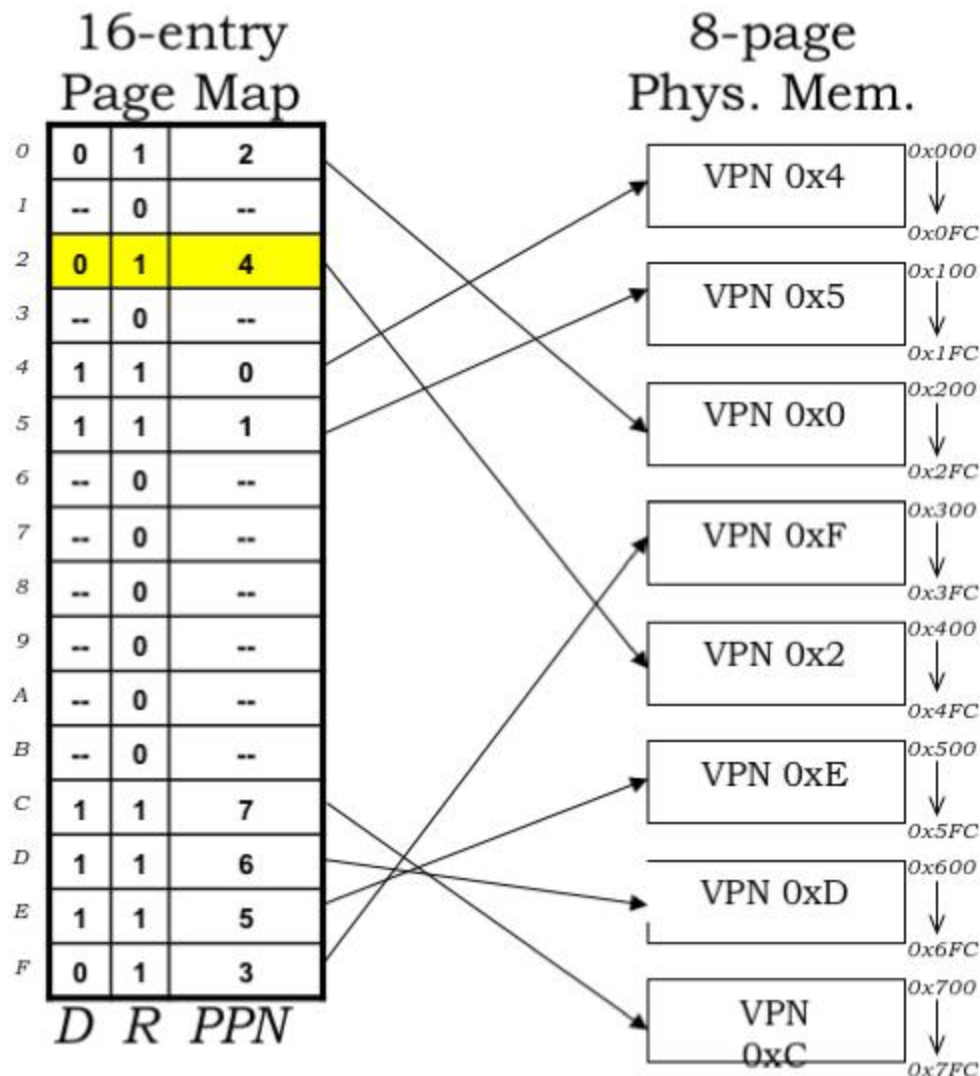
Simple Page Map Design



One entry per virtual page

- **Resident bit** R = 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** D = 1 if we've changed this page since loading it from disk (and therefore need to write it to disk when it's replaced)

Example: Virtual → Physical Translation



VPN	offset	
4	8	VA
3	8	PA
PPN		

Setup:

256 bytes/page (2^8)

16 virtual pages (2^4)

8 physical pages (2^3)

12-bit VA (4 vpn, 8 offset)

11-bit PA (3 ppn, 8 offset)

LRU page: VPN = 0xE

LD(R31,0x2C8,R0):

VA = 0x2C8, PA = 0x4C8

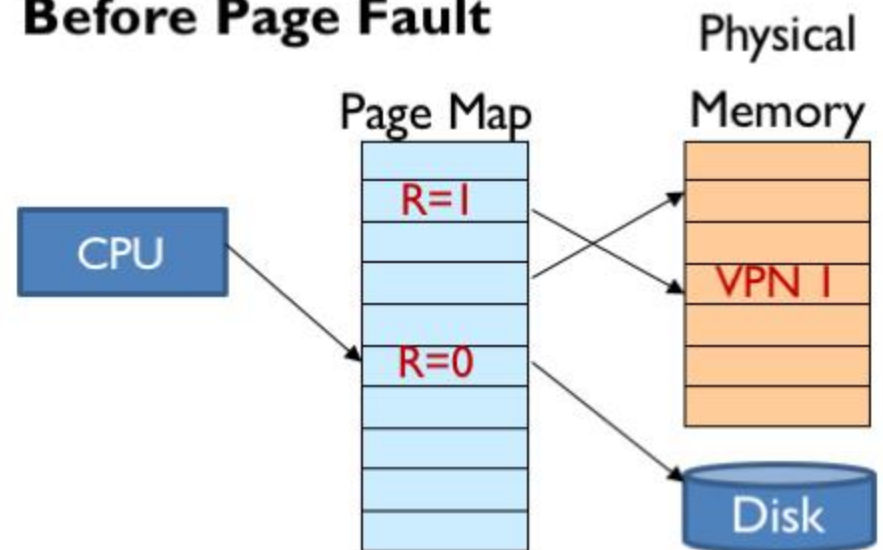
VPN = 0x2
→ PPN = 0x4

Page Faults

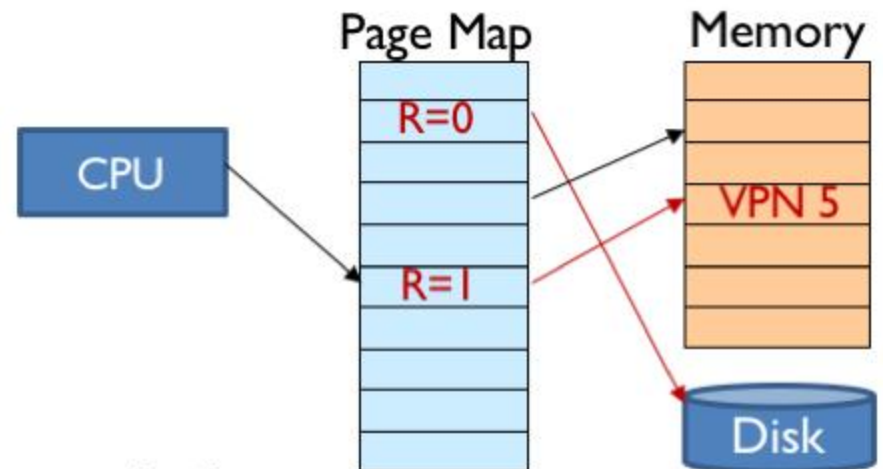
If a page does not have a valid translation, MMU causes a **page fault**. OS page fault handler is invoked, handles miss:

- Choose a page to replace, write it back if dirty. Mark page as no longer resident
 - Are there any restrictions on which page we can we select? **
- Read page from secondary storage into available physical page
- Update page map to show new page is resident
- Return control to program, which re-executes memory access

Before Page Fault



After Page Fault



** https://en.wikipedia.org/wiki/Page_replacement_algorithm#Page_replacement_algorithms

Example: Page Fault

VPN	offset	VA
4	8	VA

PPN	offset	PA
3	8	PA

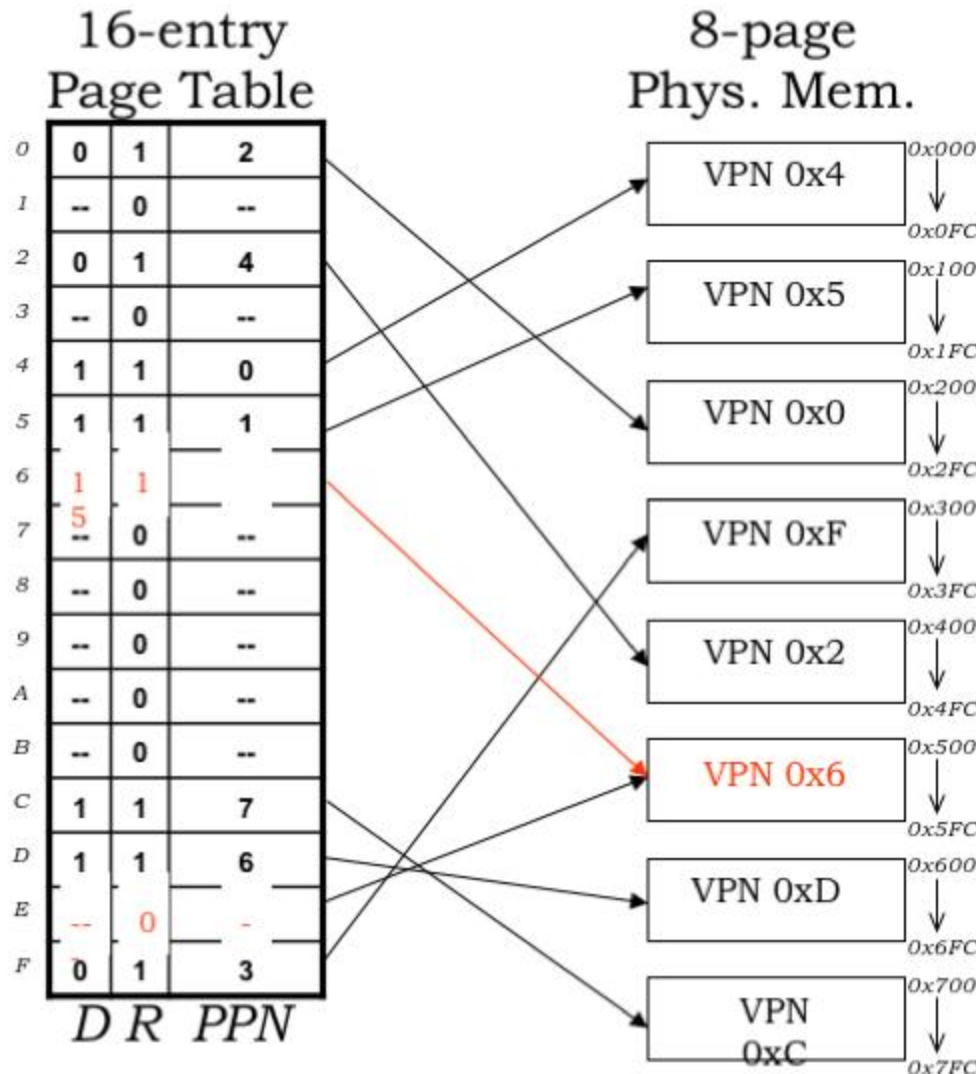
Setup:

256 bytes/page (2^8)
 16 virtual pages (2^4)
 8 physical pages (2^3)
 12-bit VA (4 vpn, 8 offset)
 11-bit PA (3 ppn, 8 offset)
 LRU page: VPN = 0xE

ST(BP, -4, SP), SP = 0x604
 VA = 0x600, PA = 0x500

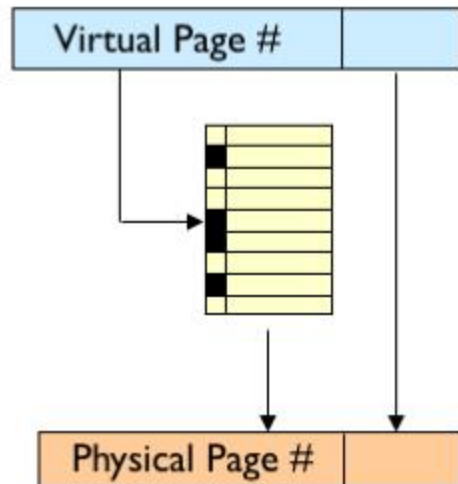
VPN = 0x6

- ⇒ Not resident, it's on disk
- ⇒ Choose page to replace (LRU = 0xE)
- ⇒ D[0xE] = 1, so write 0x500-0x5FC to disk
- ⇒ Mark VPN 0xE as no longer resident
- ⇒ Read in page VPN 0x6 from disk into 0x500-0x5FC
- ⇒ Set up page map for VPN 0x6 = PPN 0x5
- ⇒ PA = 0x500
- ⇒ This is a write so set D[0x6] = 1



Virtual Memory: the CS View

Problem: Translate
VIRTUAL ADDRESS
to PHYSICAL ADDRESS



```
int VtoP(int Vaddr) {  
    int VPageNo = Vaddr >> p;  
    int P0 = Vaddr & ((1 << p) - 1);  
    if (R[VPageNo] == 0)  
        PageFault(VPageNo);  
    return (PPN[VPageNo] << p) | P0;  
}
```

Multiply by 2^p , the page size

```
/* 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:

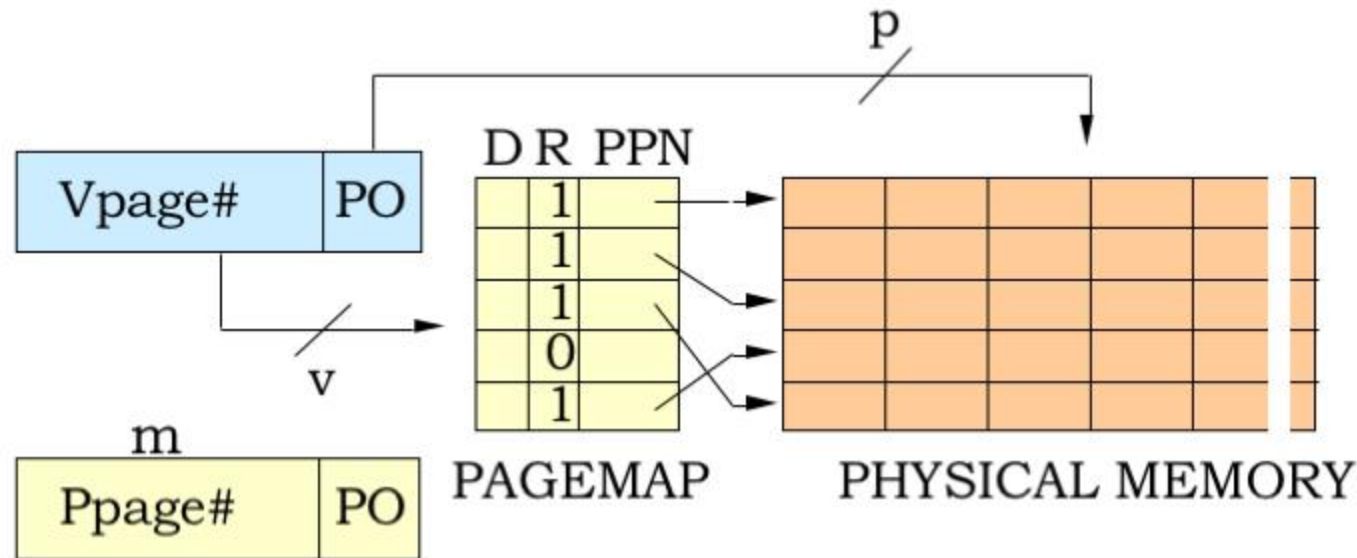
- devote HARDWARE to high-traffic, performance-critical path
- use (slow, cheap) SOFTWARE to handle exceptional cases

hardware	{	<pre>int VtoP(int VPageNo,int P0) { if (R[VPageNo] == 0)PageFault(VPageNo); return (PPN[VPageNo] << p) P0; }</pre>
software	{	<pre>/* 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; }</pre>

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: 4KB -16 KB

Typical $(v+p)$: 32-64 bits

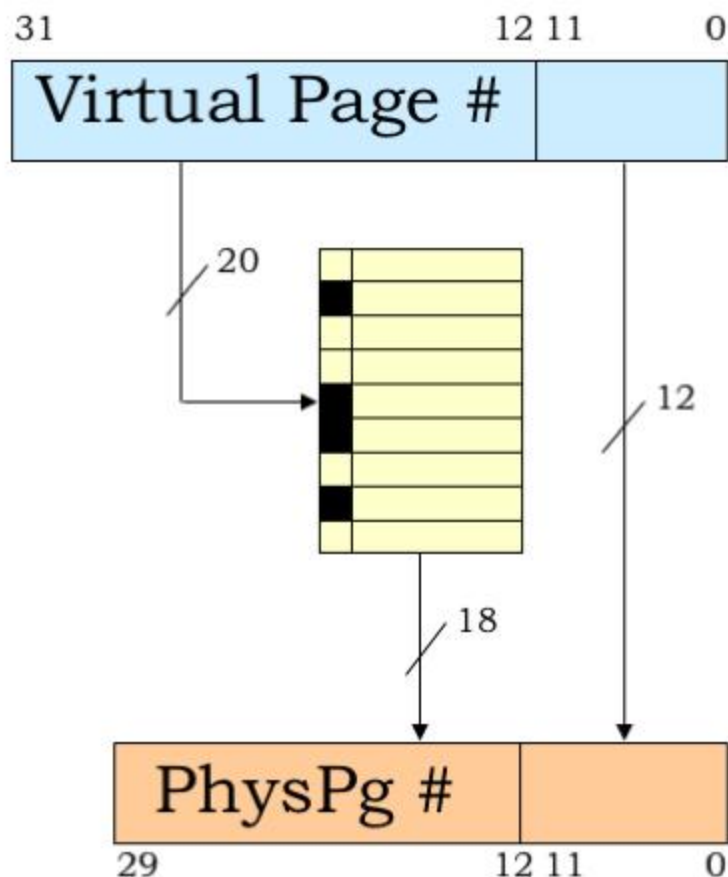
(4GB-16EB)

Typical $(m+p)$: 30-40+ bits

(1GB-1TB)

Long virtual addresses allow
ISAs to support larger
memories → ISA longevity

Example: Page Map Arithmetic



SUPPOSE...

32-bit Virtual address ($v+p$)

30-bit physical address ($m+p$)

4 KB page size ($p = 12$)

THEN:

Physical Pages = $2^{18} = 256K$

Virtual Pages = 2^{20}

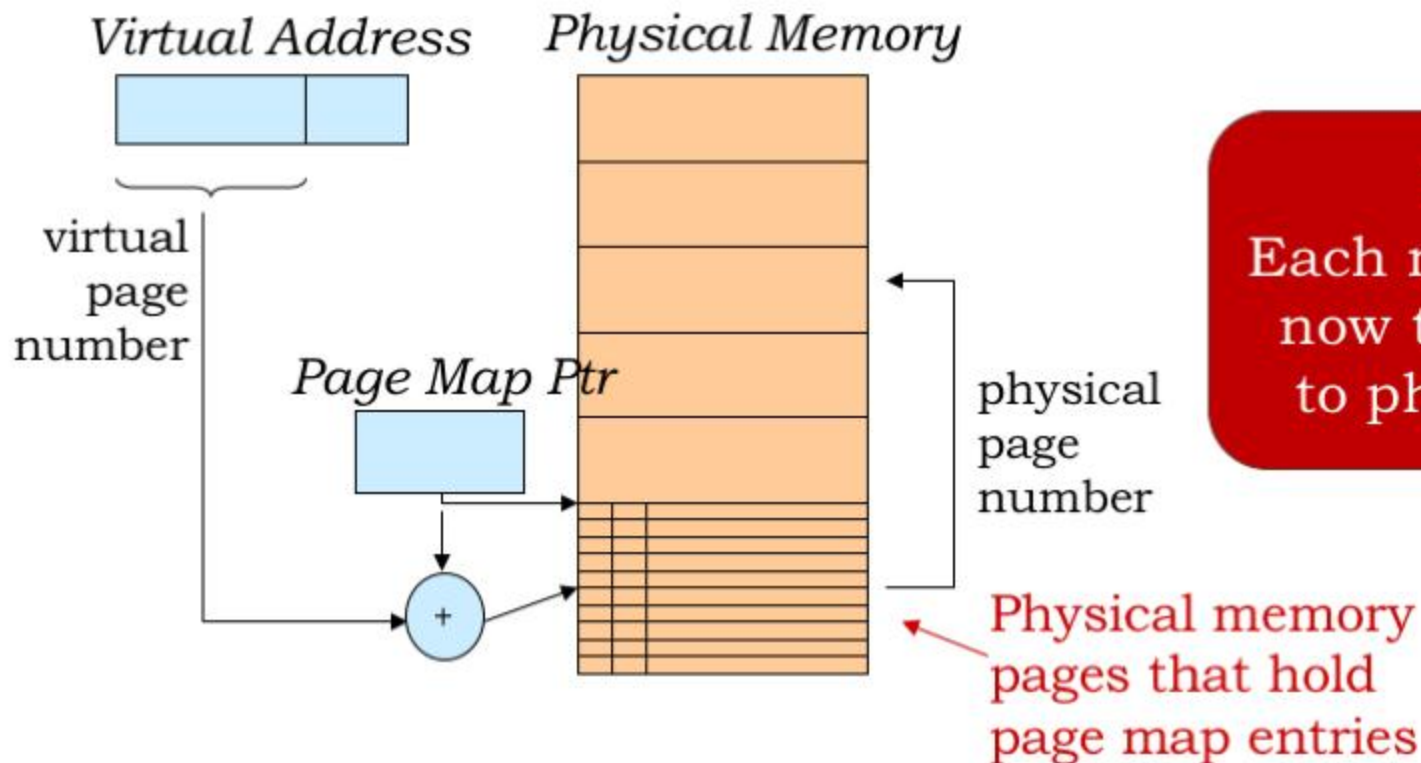
Page Map Entries = $2^{20} = 1M$

Bits In pagemap = $20 * 2^{20} \sim 20M$

Use fast SRAM for page map??? **OUCH!**

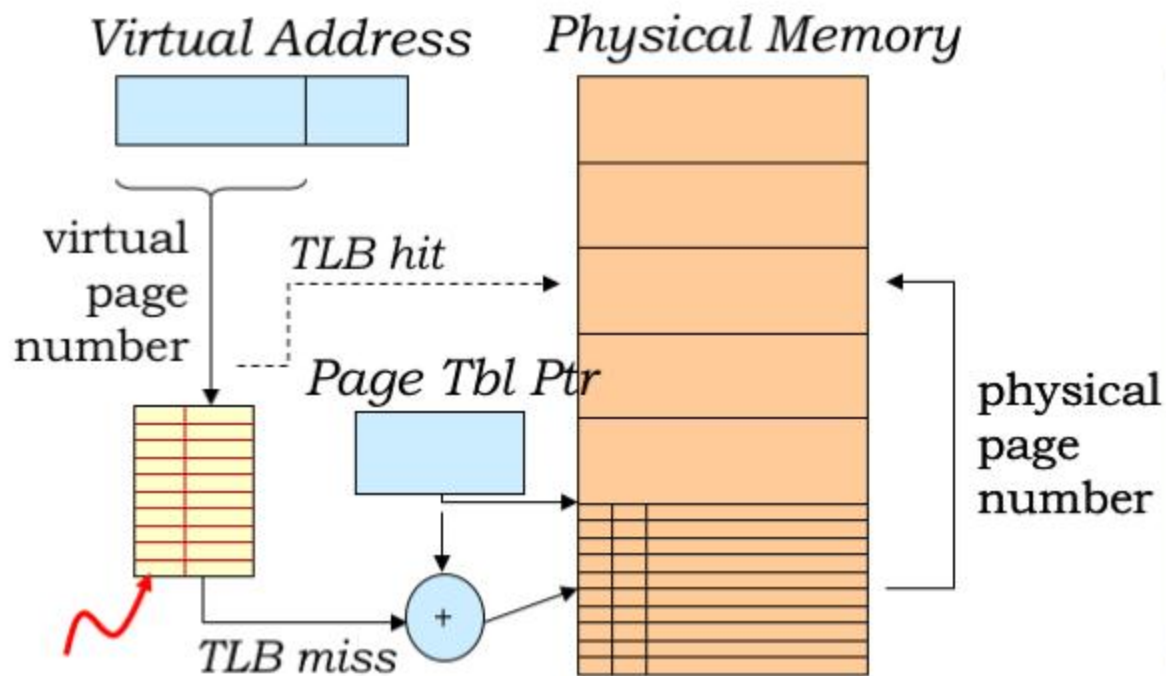
RAM-Resident Page Maps

- **Small** page maps can use dedicated SRAM... gets expensive for big ones!
- Solution: Move page map to **main memory**:



Translation Look-aside Buffer (TLB)

- Problem: 2x performance hit... each memory reference now takes 2 accesses!
- Solution: **Cache the page map entries**



IDEA:

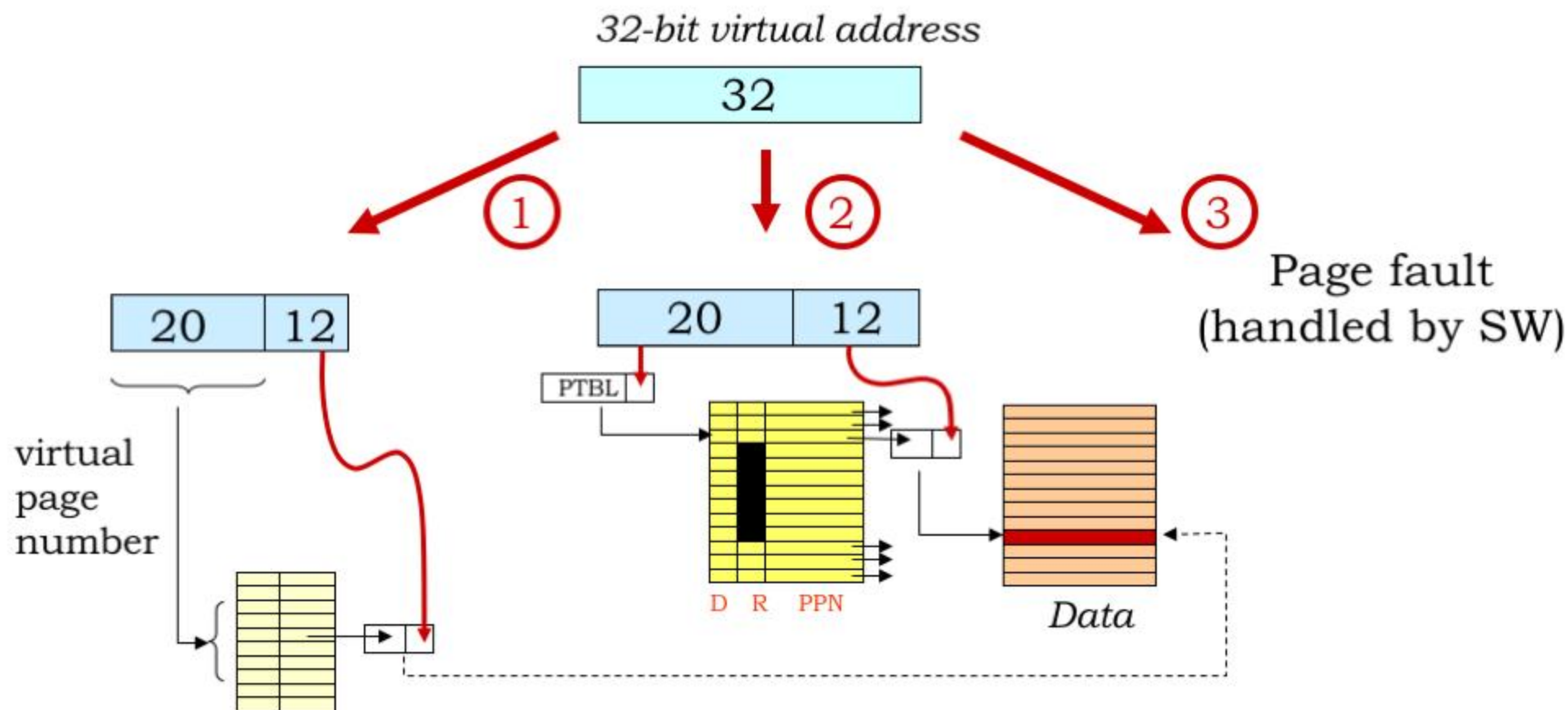
LOCALITY in memory
reference patterns →
SUPER locality in
references to page
map

VARIATIONS:

- multi-level page map
- paging the page map!

TLB: small cache of page table entries
Associative lookup by VPN

MMU Address Translation



Look in TLB: VPN→PPN cache
Usually implemented as a small
fully-associative cache

Putting it All Together: MMU with TLB

Suppose

- virtual memory of 2^{32} bytes
- physical memory of 2^{24} bytes
- page size is 2^{10} (1 K) bytes
- 4-entry fully associative TLB
- $[p = 10, v = 22, m = 14]$

Page Map

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
...			

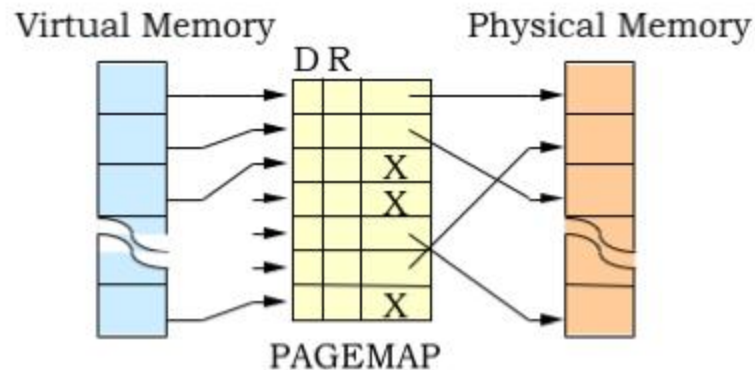
TLB

Tag	Data
VPN	R D PPN
0	0 0 3
6	1 1 2
1	1 1 9
3	0 0 5

1. How many pages can reside in physical memory at one time? 2^{14}
2. How many entries are there in the page table? 2^{22}
3. How many bits per entry in the page table? (Assume each entry has PPN, resident bit, dirty bit) 16
4. How many pages does the page table occupy? 2^{23} bytes = 2^{13} pages
5. What fraction of virtual memory can be resident? $1/2^8$
6. What is the physical address for virtual address 0x1804? What components are involved in the translation? [VPN=6] 0x804
7. Same for 0x1080 [VPN=4] 0x1480
8. Same for 0x0FC [VPN=0] page fault

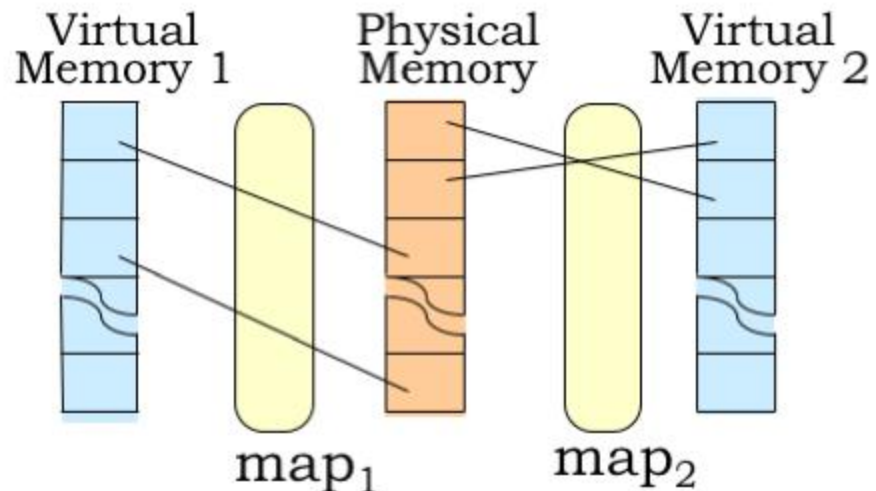
Contexts

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

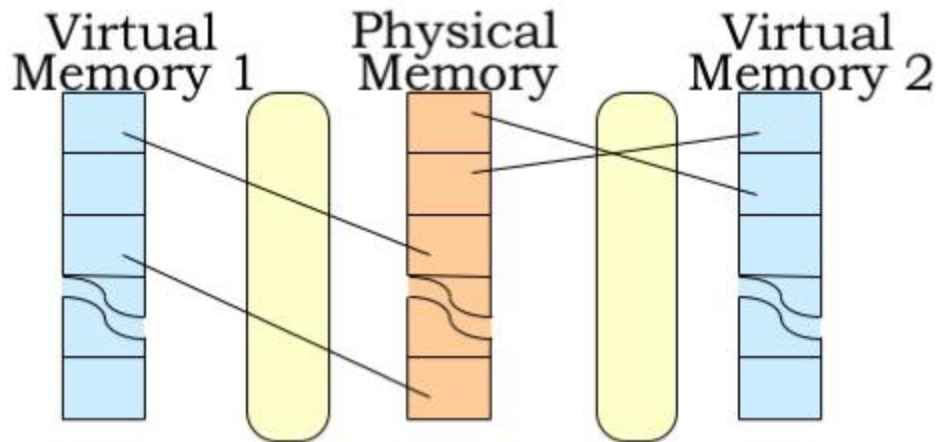


Several programs may be simultaneously loaded into main memory, each in its separate context:

“Context switch”:
reload the page map?



Contexts: A Sneak Preview



First Glimpse of a
VIRTUAL MACHINE

1. TIMESHARING among several programs
 - Separate context for each program
 - OS loads appropriate context into page map when switching among programs
2. Separate context for OS “Kernel” (e.g., interrupt handlers)...
 - “Kernel” vs “User” contexts
 - Switch to Kernel context on interrupt;
 - Switch back on interrupt return.

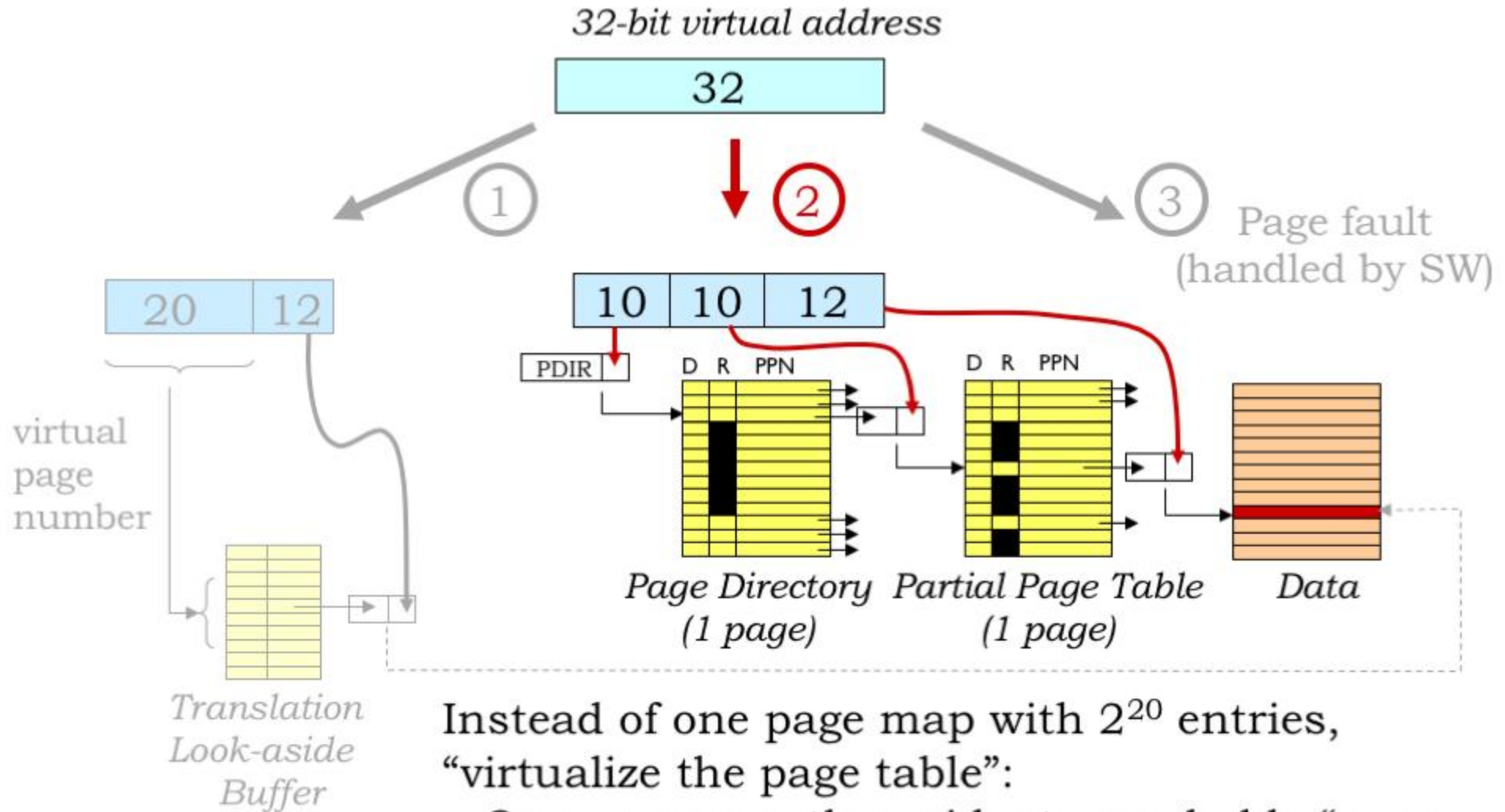
Memory Management & Protection

- Applications are written as if they have access to the entire virtual address space, without considering where other applications reside
 - Enables fixed conventions (e.g., program starts at 0x1000, stack is contiguous and grows up, ...) without worrying about conflicts
- OS Kernel controls all contexts, prevents programs from reading and writing into each other's memory

Address Space



Multi-level Page Maps

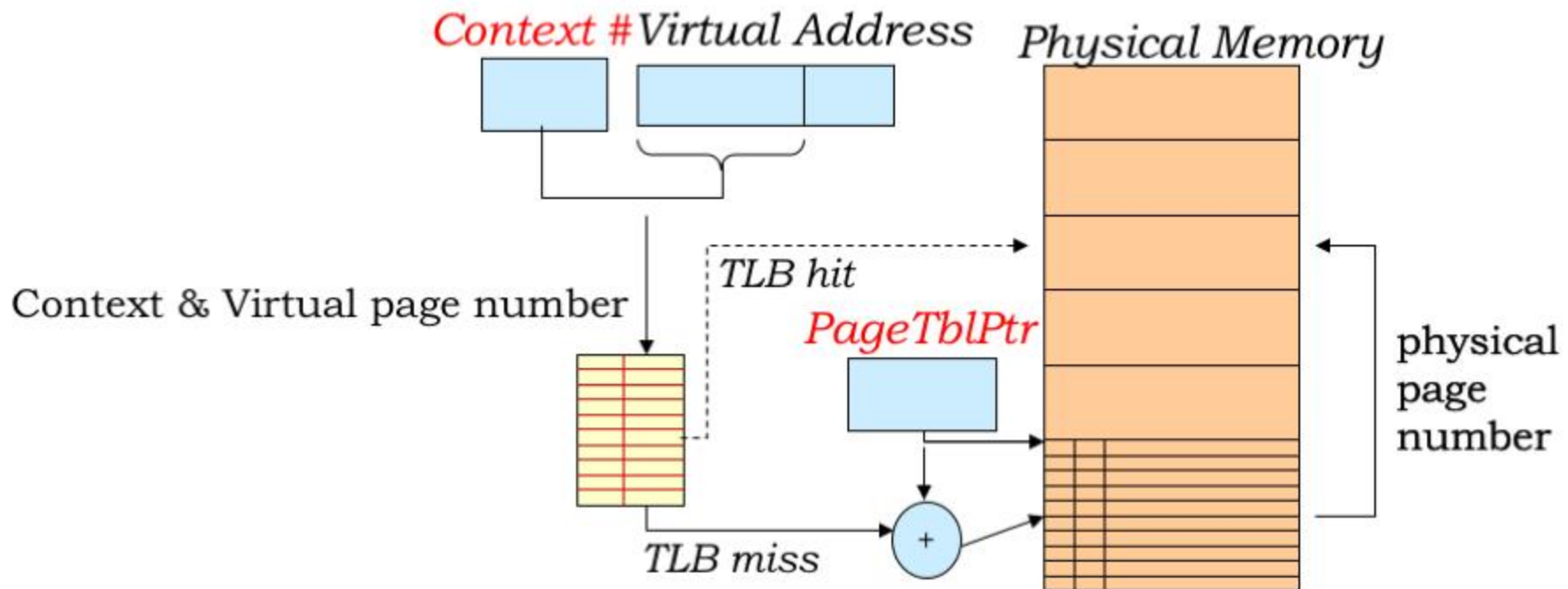


Instead of one page map with 2^{20} entries, “virtualize the page table”:

One permanently-resident page holds “page directory” which has 1024 entries pointing to 1024-entry partial page tables in *virtual* memory!

Rapid Context -Switching

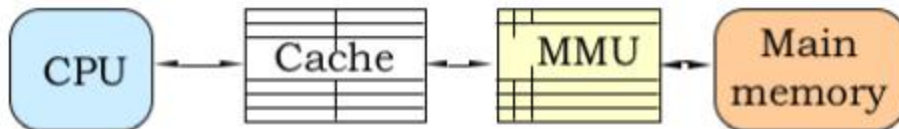
Add a register to hold index of current context. To switch contexts: update Context # and PageTblPtr registers. Don't have to flush TLB since each entry's tag includes context # in addition to virtual page number



Using Caches with Virtual Memory

Virtually-Addressed Cache

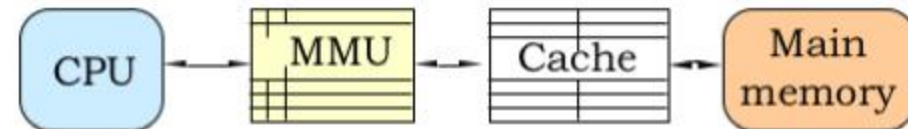
Tags from virtual addresses



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

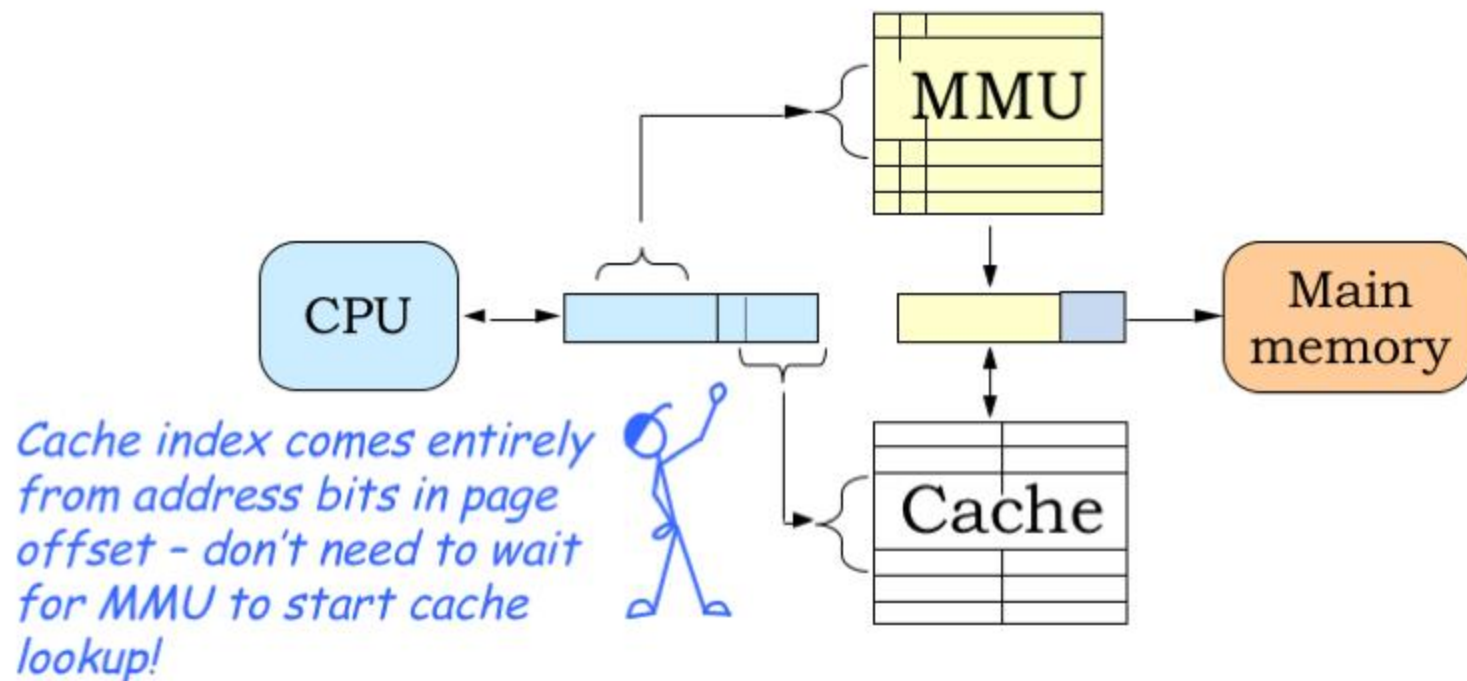
Physically-Addressed Cache

Tags from physical addresses



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

Best of Both Worlds: Overlapped Operation



OBSERVATION: If cache index bits are a subset of page offset bits, tag access in a physical cache can *overlap* page map access. Tag from cache is compared with physical page address from MMU to determine hit/miss.

Problem: Limits # of bits of cache index → increase cache capacity by increasing associativity

Summary: Virtual Memory

- Goal 1: Exploit locality on a large scale
 - Programmers want a large, flat address space, but use a small portion!
 - Solution: Cache working set into RAM from disk
 - Basic implementation: MMU with single-level page map
 - Access loaded pages via fast hardware path
 - Load virtual memory on demand: page faults
 - Several optimizations:
 - Moving page map to RAM, for cost reasons
 - Translation Lookaside Buffer (TLB) to regain performance
 - Cache/VM interactions: Can cache physical or virtual locations
- Goals 2 & 3: Ease memory management, protect multiple contexts from each other
 - We'll see these in detail on the next lecture!