20. Virtual memory – Organization and Operation

EECS 370 – Introduction to Computer Organization - Winter 2016

Profs. Valeria Bertacco & Reetu Das

EECS Department
University of Michigan in Ann Arbor, USA

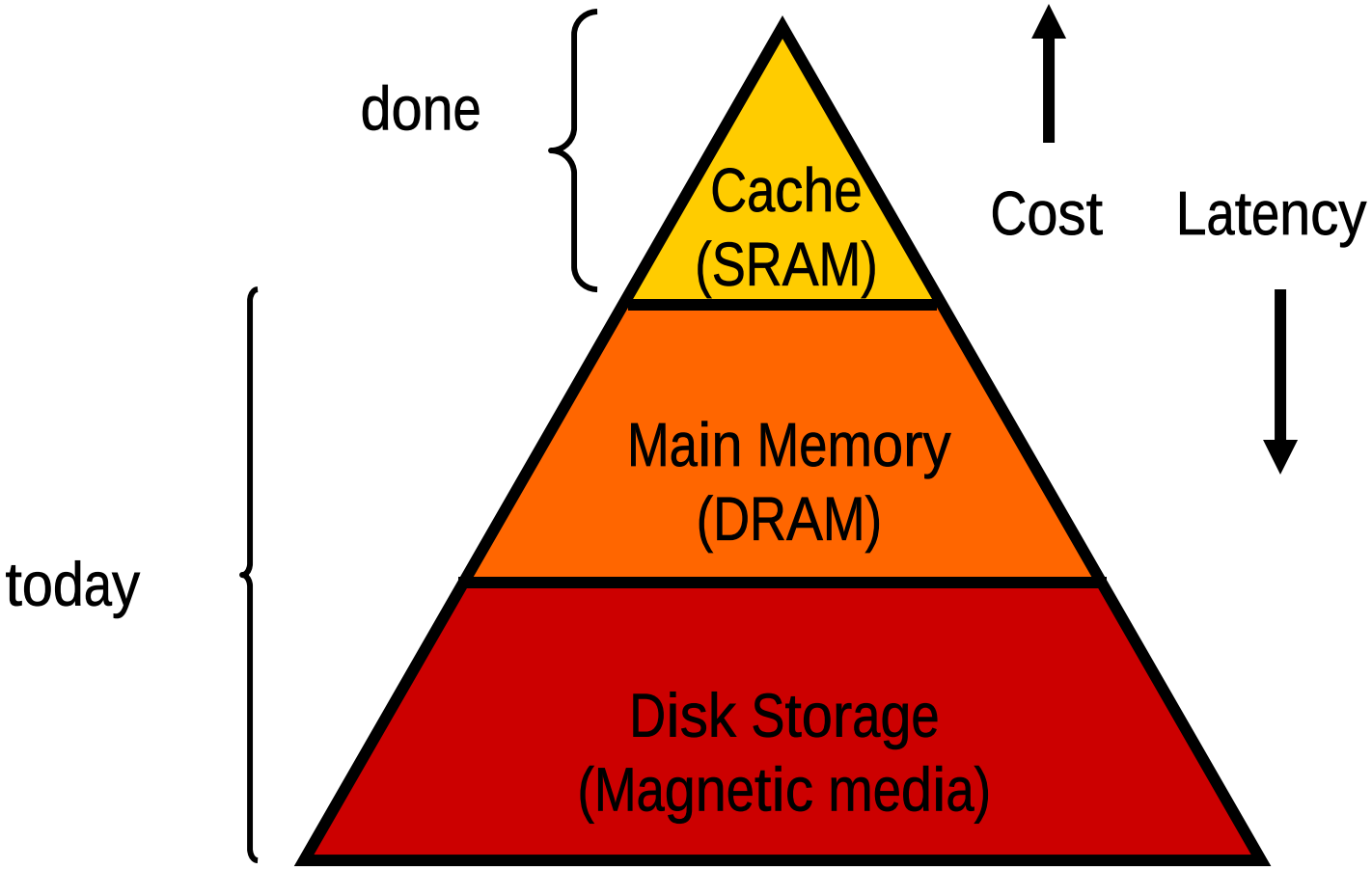
Announcements

- Project 4 – due Thursday
  - And no more projects after that!
- Homework 7 a week from today

- FINAL EXAM preview:
  - Cumulative
  - Cheat sheet – 3 sides (2 sheets)
  - 120 minutes long (longer than midterms!)
  - Make sure to not get sick

So glad to be back!

Storage Hierarchy



Memory: the issues(s)

- We run many programs on a same machine
  - Each of them may require GBs of storage
  - Unrelated programs should not have access to each other's storage
- DRAM is too expensive to buy 100s GB, but disk space is not…
  - We want our system to work even if it requires more DRAM than we bought.
  - We also don't want a program that works on a machine with 2048 MB of DRAM to stop working if we try to run it on a machine with only 512 MB of main memory.
- And, it would be nice to be able to enforce different policies on different portions of the memory (e.g.: read-only, etc)

Solution 1: User control

- Leave the problem to the programmer
  - Assume the programmer knows the exact configuration of the machine.
    - Programmer must either make sure the program fits in memory, or break the program up into pieces that do fit and load each other off the disk when necessary

- Not a bad solution in some domains
  - The hardware design is simple
  - Very simple embedded controllers
  - Playstation 2, arduino

Solution 2: Virtual memory

- Build new hardware and software that automatically translates each memory reference from a

virtual address

(which the programmer sees as an array of bytes)

to a

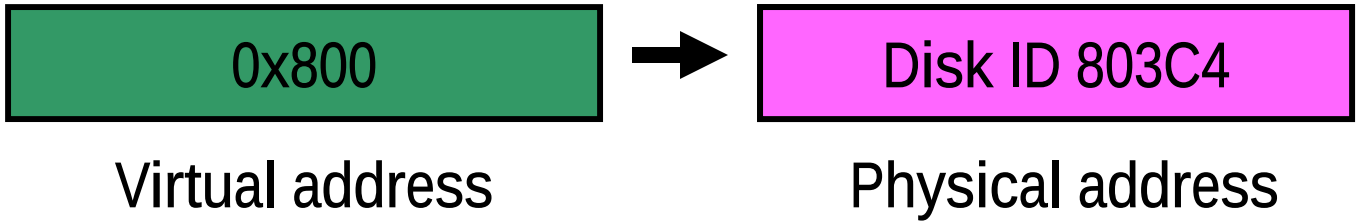
physical address

(which the hardware uses to either index DRAM or identify where the storage resides on disk)

Basics of Virtual Memory

- Any time you see the word virtual in computer science and architecture it means "using a level of indirection"

- Virtual memory hardware changes the virtual address the programmer sees into the physical one the memory chips see.



Virtual Memory View

- Virtual memory lets the programmer "see" a memory array larger than the DRAM available on a particular computer system.

- Virtual memory enables multiple programs to share the physical memory without:
  - Knowing other programs exist (transparency).
  - Worrying about one program modifying the data contents of another (protection).

Managing virtual memory

- Managed by hardware logic *and* operating system software.
  - Hardware for speed
  - Software for flexibility and because disk storage is controlled by the operating system.

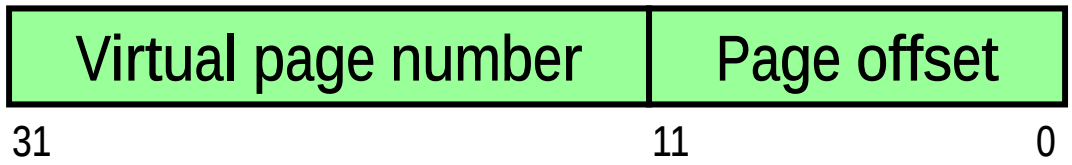
- The hardware must be designed to support VM

Virtual Memory

- Treat main memory like a cache
  - Misses go to the disk
- How do we minimize disk accesses?
  - Buy lots of memory.
  - Exploit temporal locality
    - Fully associative? Set associative? Direct mapped?
  - Exploit spatial locality
    - How big should a block be?
  - Write-back or write-through?

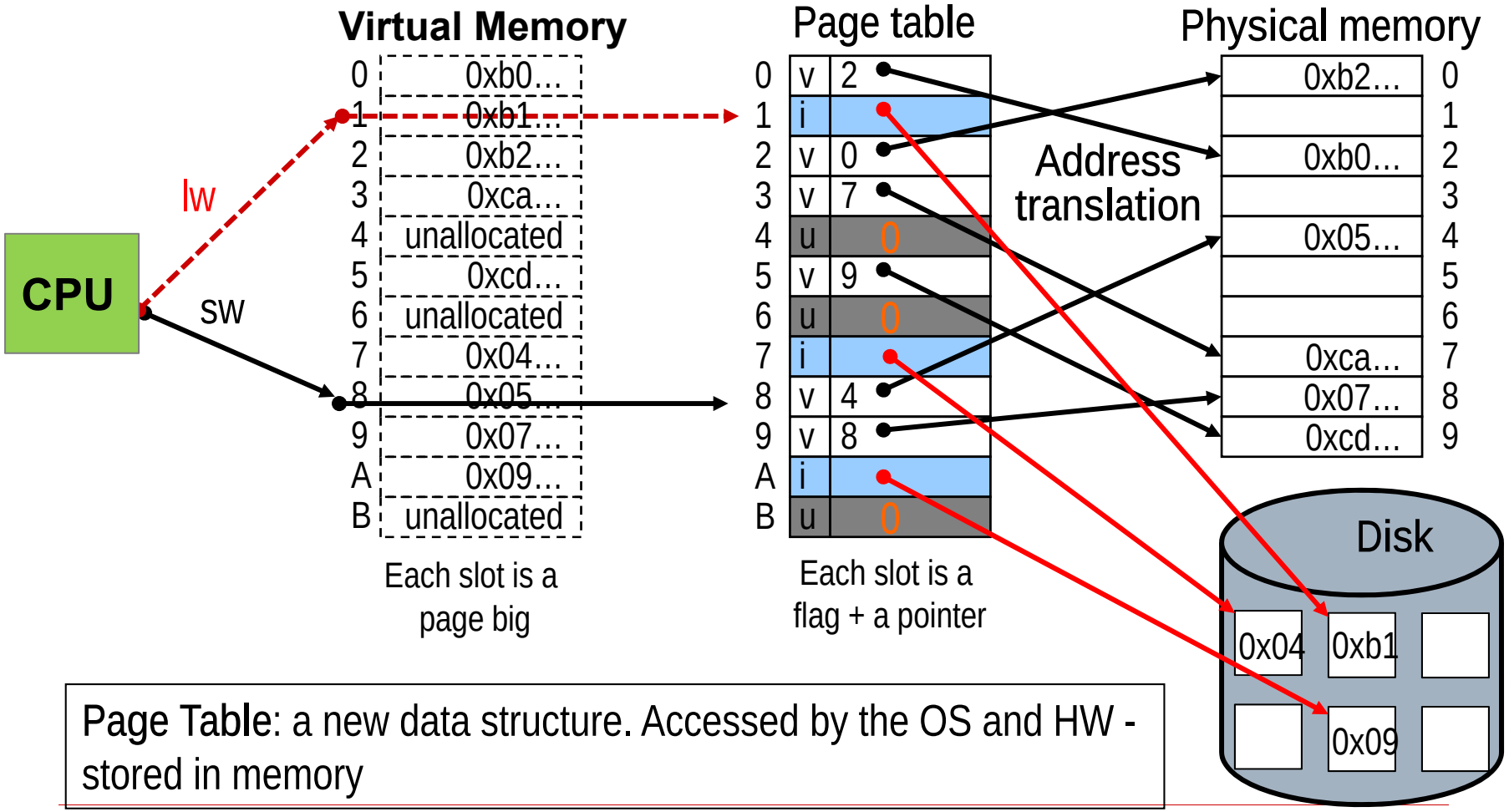
Virtual memory terminology

- Blocks are called Pages
  - A virtual address consists of
    - A virtual page number
    - A page offset field (low order bits of the address)
- Misses are call Page faults
  - and they are generally handled as an exception



Address Translation

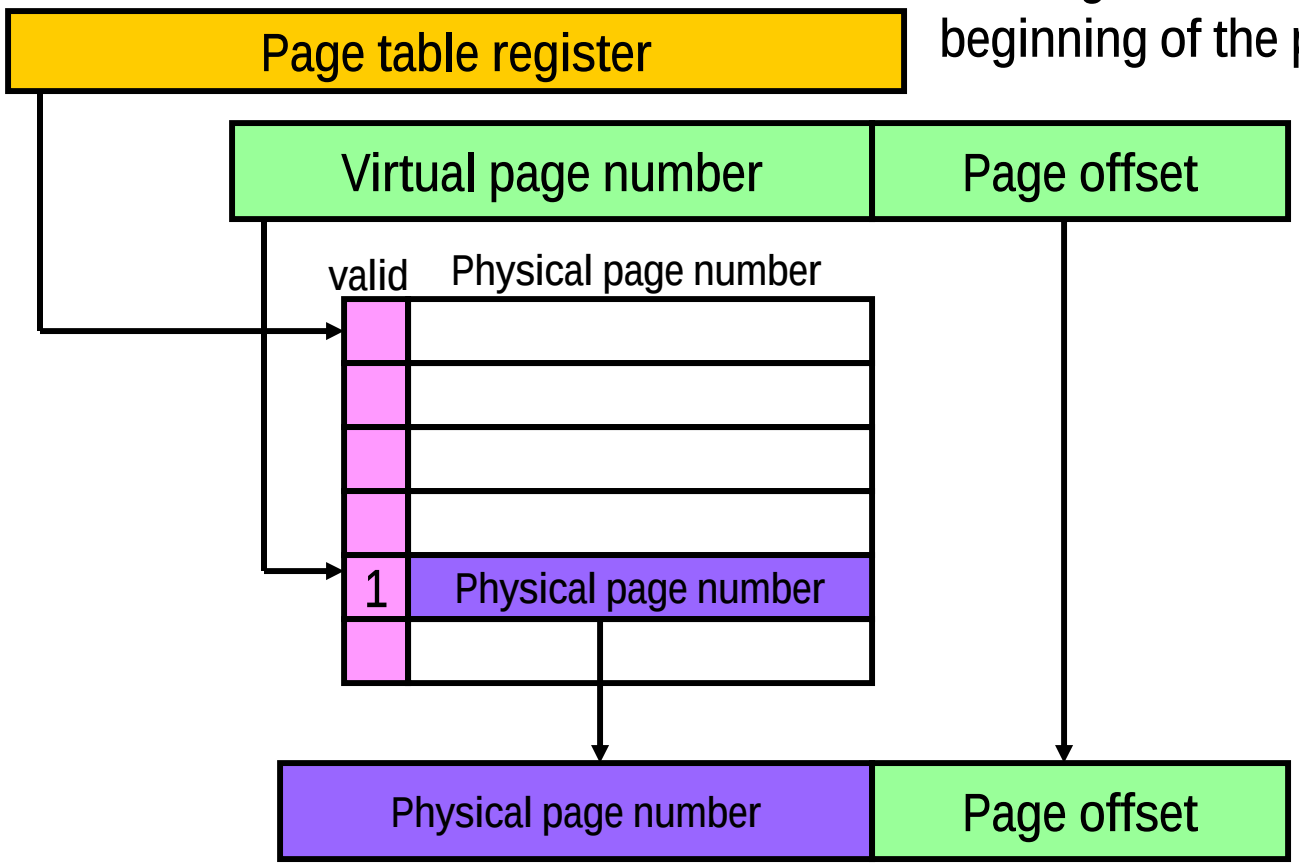
The address translation information of the program is contained in the *Page Table*



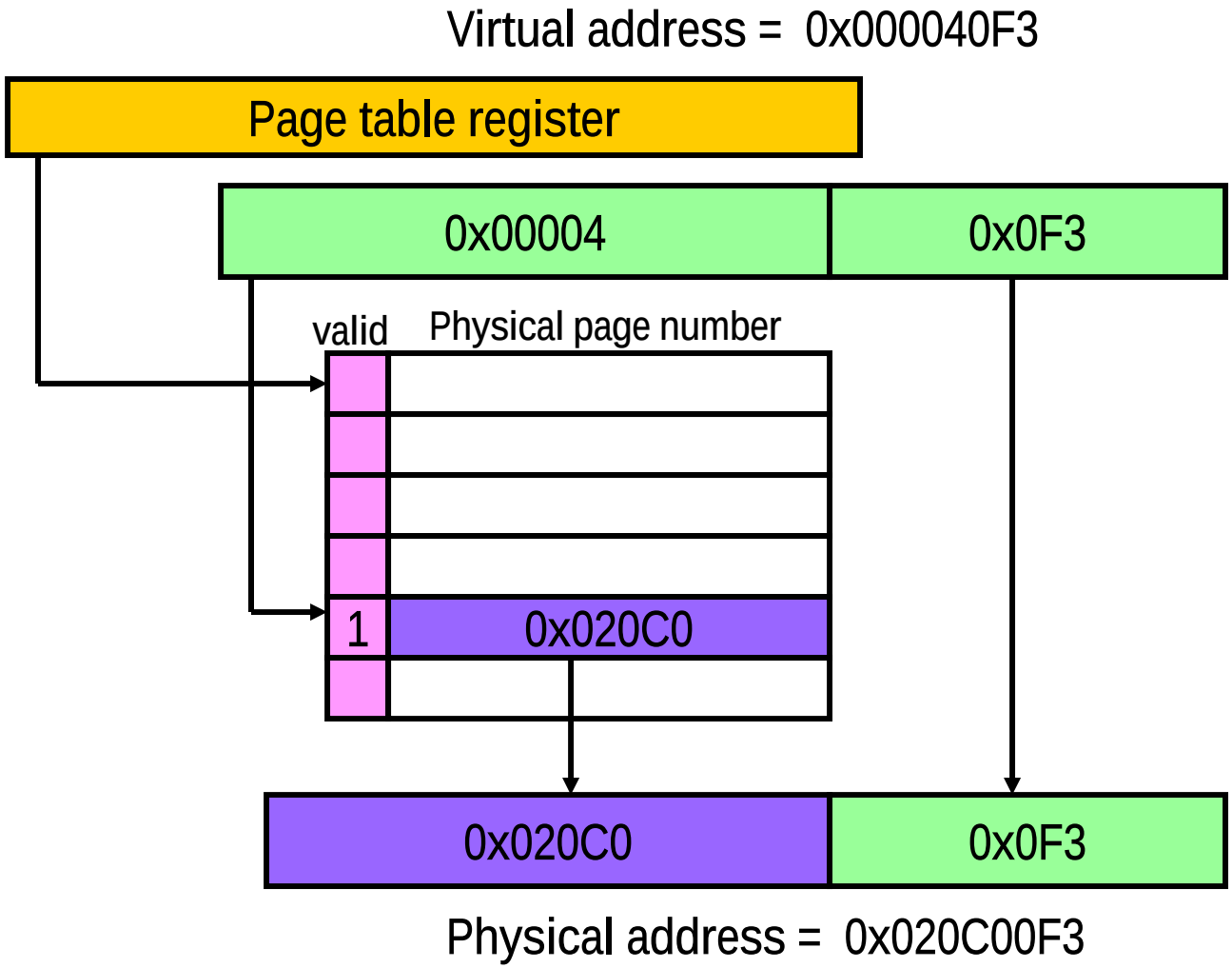
Page Table: a new data structure. Accessed by the OS and HW - stored in memory

Page table components

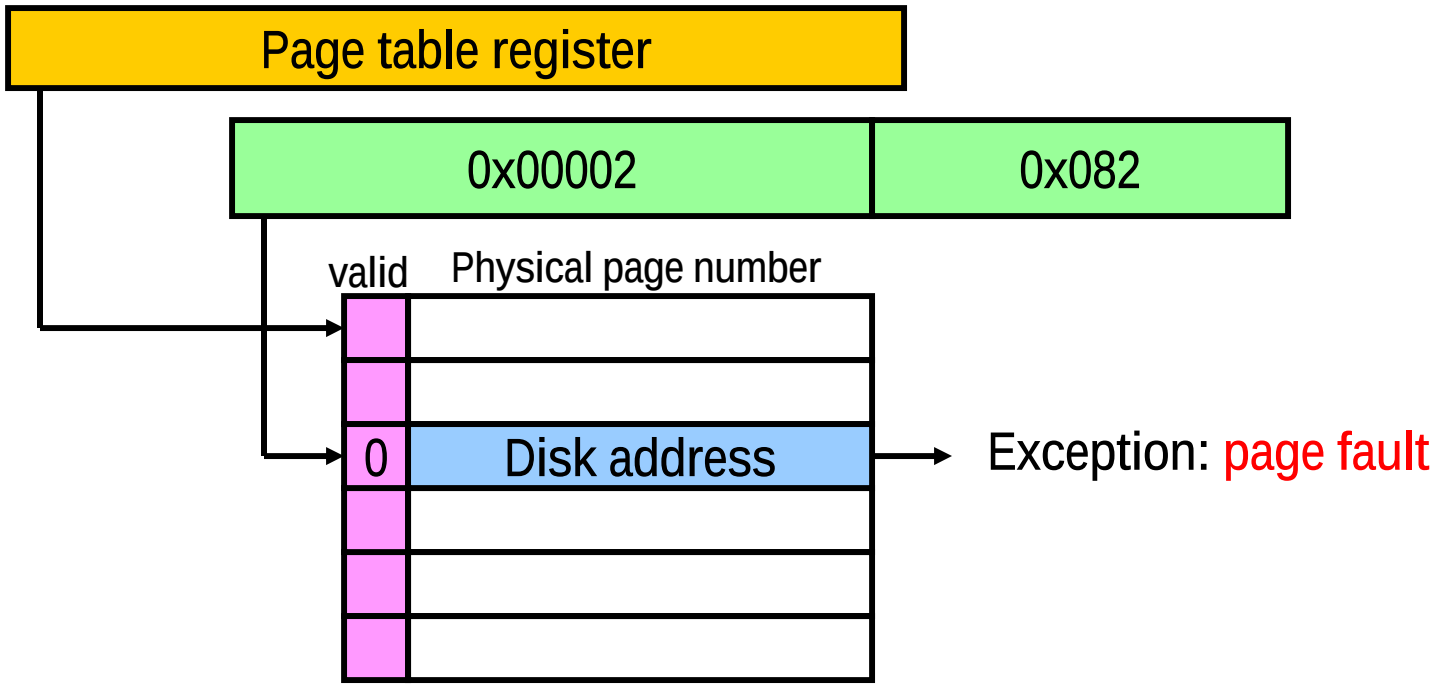
The *Page table register* points to the beginning of the page table



Page table components - Example

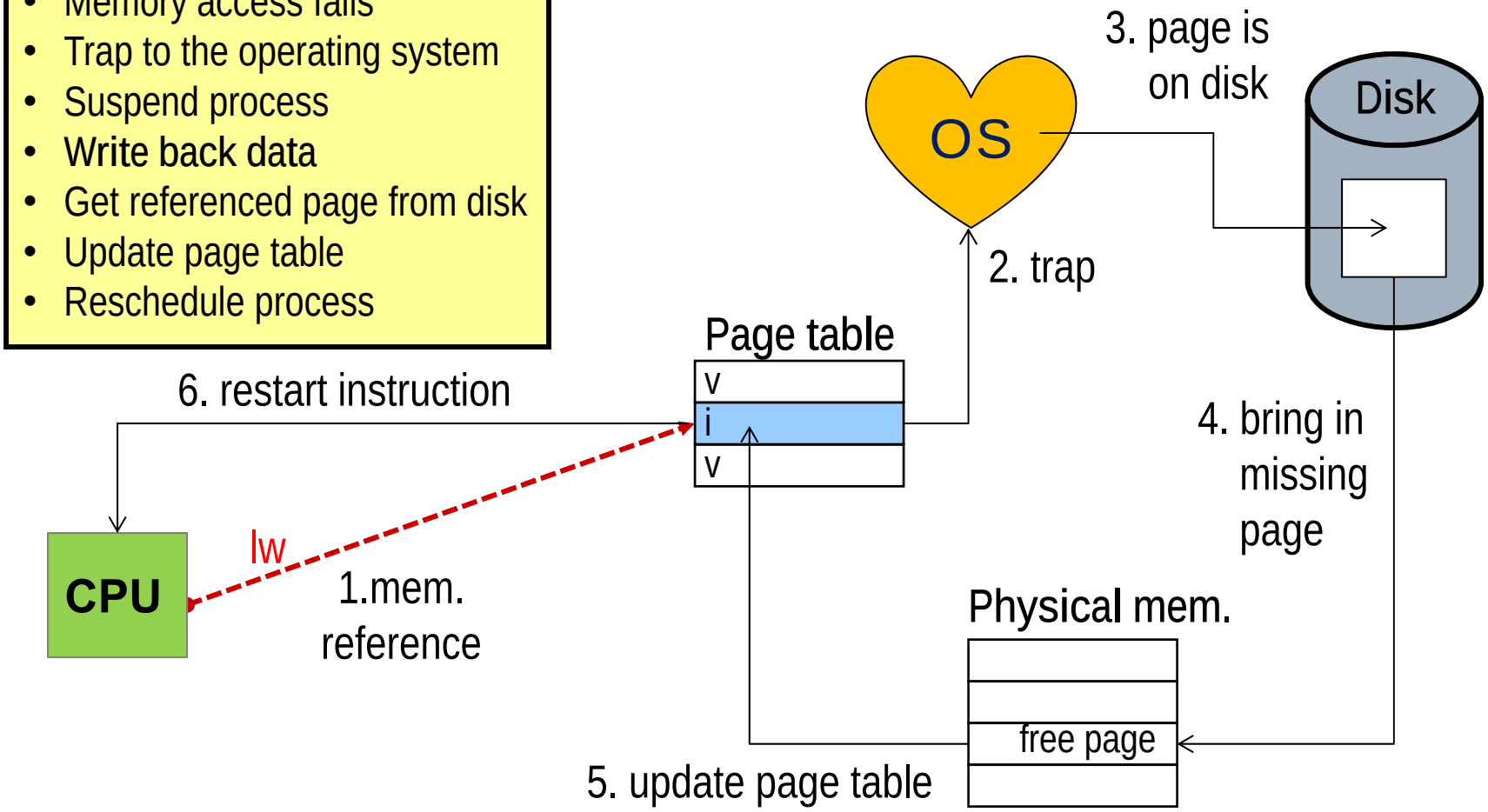


Page faults


Page table register
0x00002
0x082
valid
Physical page number
0
Disk address
Exception: page fault

Page faults (cont.)

- Memory access fails
- Trap to the operating system
- Suspend process
- **Write back data**
- Get referenced page from disk
- Update page table
- Reschedule process



How do we find it on disk?

- That is not a hardware problem! ☺
- Most operating systems partition the disk into logical devices (C: , D: , /home, etc.)
- They also have a hidden partition to support the disk portion of virtual memory
  - Swap partition on UNIX machines
  - You then index into the correct page in the swap partition.

Class Problem 1

- Given the following
  - 4KB page size, physical memory of 16KB, page table stored in physical page 0 and can never be evicted, 20 bit, byte-addressable virtual address space.
  - The page table initially has virtual page 0 in physical page 1, virtual page 1 in physical page 2 and no valid data in other physical pages.
- Fill in the table on the next slide for each reference

Class Problem (continued)

Virtual addr	Virtual page	Page fault?	Phys addr
0x00F0C			
0x01F0C			
0x20F0C			
0x00100			
0x00200			
0x30000			
0x01FFF			
0x00200			

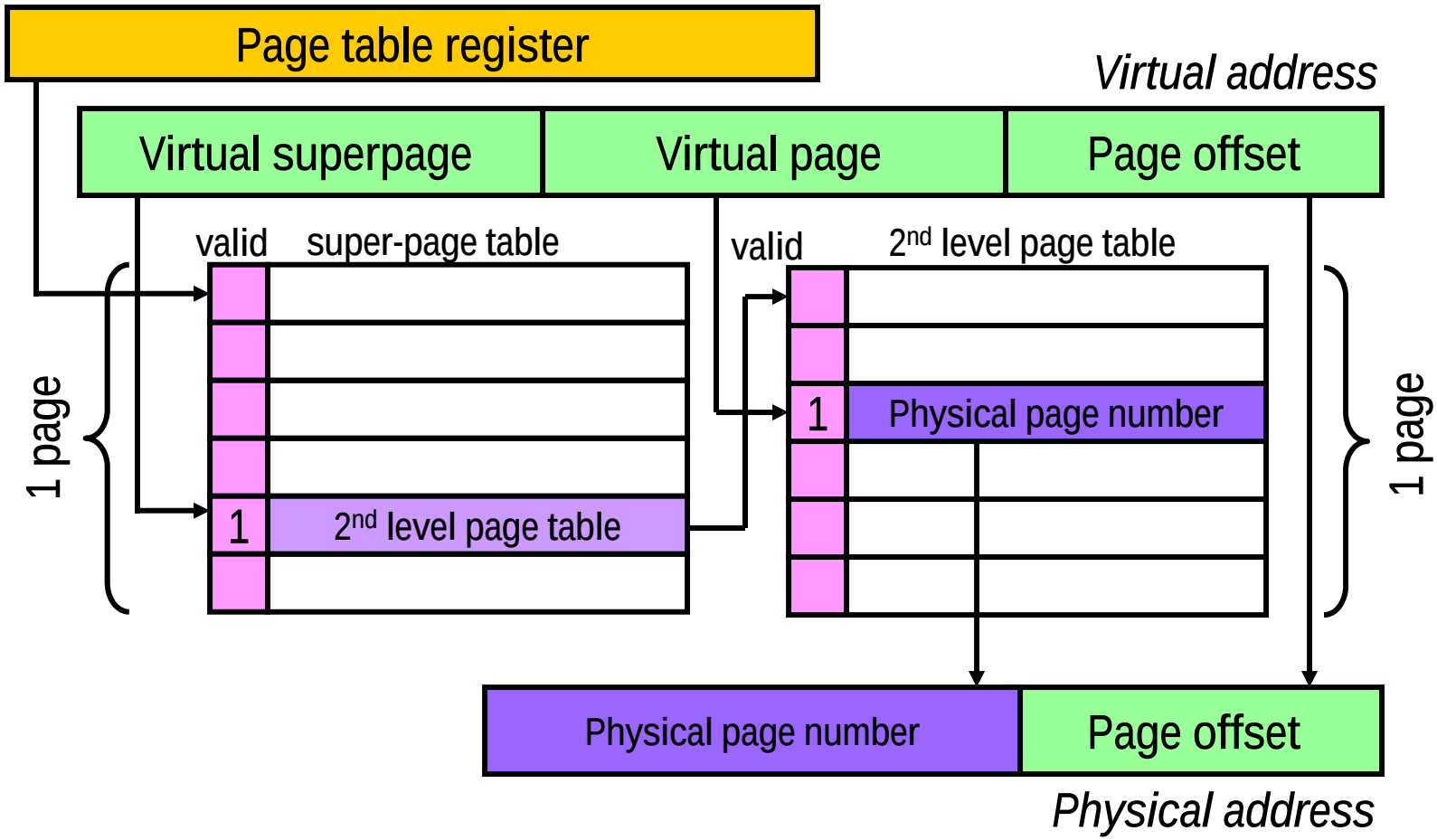
Size of the page table

- How big is a page table entry?
  - For ARM the virtual address is 32 bits
    - If the machine can support 1GB = 2^{30} bytes of physical memory and we use pages of size 4KB = 2^{12}, then the physical page number is 30-12 = 18 bits.
      Plus another valid bit + other useful stuff (read only, dirty, etc.)
    - Let say about 3 bytes.
- How many entries in the page table?
  - ARM virtual address is 32 bits – 12 bit page offset = 2^{20} or ~1,000,000 entries
- Total size of page table: 2^{20} x 3 bytes ~ 3 MB

How can you organize the page table?

1. Continuous 3MB region of physical memory
   - Cons: Always takes 3MB regardless of how much physical memory is used
2. Use inverted page table
   - Slower, but less memory
3. Use a hash function instead of an array
   - Slower, but less memory
4. Use a multi-level page table! (Build a hierarchical page table, keep in physical memory only the translations used)
   - Super page table in physical memory
   - Second (and maybe third) level page tables only if needed
   - Size is proportional to the amount of memory used

Hierarchical page table – Possible structure



Hierarchical page table – Possible structure (cont'd)

	Example
How many bits in the virtual superpage field?	*10*
How many bits in the virtual page field?	*10*
How many bits in the page offset?	*12*
How many pages in the super page table?	*1024*
How many bytes for each entry in the super page table?	*4*
How many entries in the 2nd level of the page table?	*1024*
How many bytes for each VPN in a 2nd level table?	*4*
What is the total size of the page table?	*4K+n*4K*

(this example is how 32bit x86 works)

Class Problem 2 – Multi-level VM

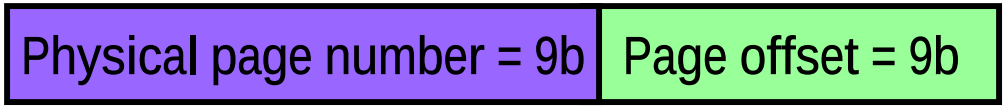
- Design the virtual memory of a byte addressable processor with 24-bits long addresses. No cache in the system. 256Kbytes of memory installed, and no additional memory can be added.

- Virtual memory page: 512 Bytes. Each page table entry must be an integer number of bytes, and must be the smallest size required to fit the physical page number + 1 bit to mark valid-entry

- Design a two-level virtual memory system. We want each second-level page table to fit exactly in one memory page, and superpage table entries are 3 bytes each (a memory address pointer to a 2L page table). Compute: Number of entries in each second-level page table; Number of virtual address bits used to index the second-level page table; Number of virtual address bits used to index the superpage table; Size of the superpage table.

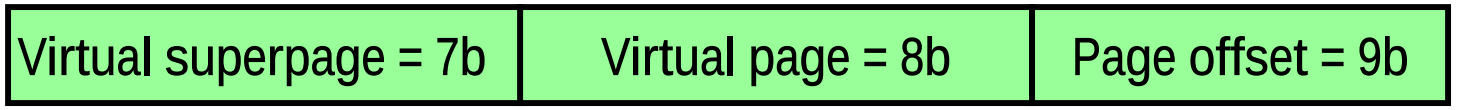
Class Problem – Multi-level VM

Page Offset:
9 bits (512B page size)



Physical address = 18b (256KB Mem size)

2nd level page table entry size: 9b (physical page number) + 1b ~ 2 bytes
2nd level page table **fits exactly in 1 page**
#entries in 2nd level page table is 512 bytes / 2 bytes = 256
#entries in 2nd level page table = 256 ➜Virtual page bits = 8b
Virtual super page bits = 24 – 8 – 9 = 7b
Super page table size = 2^7 * 3 bytes = 384

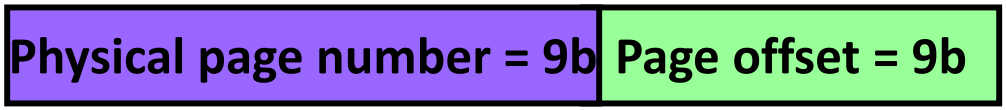


Virtual address = 24b

Exercise using previous multi-level VM

Virtual Address	Virtual Super Page	Virtual Page	Page offset	Page fault?	Physical page num.	Physical Address
0x000F0C						
0x001F0C						
0x020F0C						

Virtual address = 24b



Physical address = 18b

Assume memory for page tables is “somewhere else” in memory

Page replacement strategies

- Page table indirection enables a fully associative mapping between virtual and physical pages.

- How do we implement LRU?
  - True LRU is expensive, but LRU is a heuristic anyway, so approximating LRU is fine
  - Reference bit on page, cleared occasionally by operating system. Then pick any “unreferenced” page to evict.

Other VM translation functions

- Page data location
  - Physical memory, disk, uninitialized data
- Access permissions
  - Read only pages for instructions
- Gathering access information
  - Identifying dirty pages by tracking stores
  - Identifying accesses to help determine LRU candidate

OS support for virtual memory

- It must be able to modify the page table register, update page table values, etc.
- To enable the OS to do this, BUT not the user program, we have different execution modes for a process.
  - Executive (or supervisor or kernel level) permissions and
  - User level permissions.

Putting it all together

- Loading your program in memory
  - Ask operating system to create a new process
  - Construct a page table for this process
  - Mark all page table entries as invalid with a pointer to the disk image of the program
    - That is, point to the executable file containing the binary.
  - Run the program and get an immediate page fault on the first instruction.

Performance of virtual memory

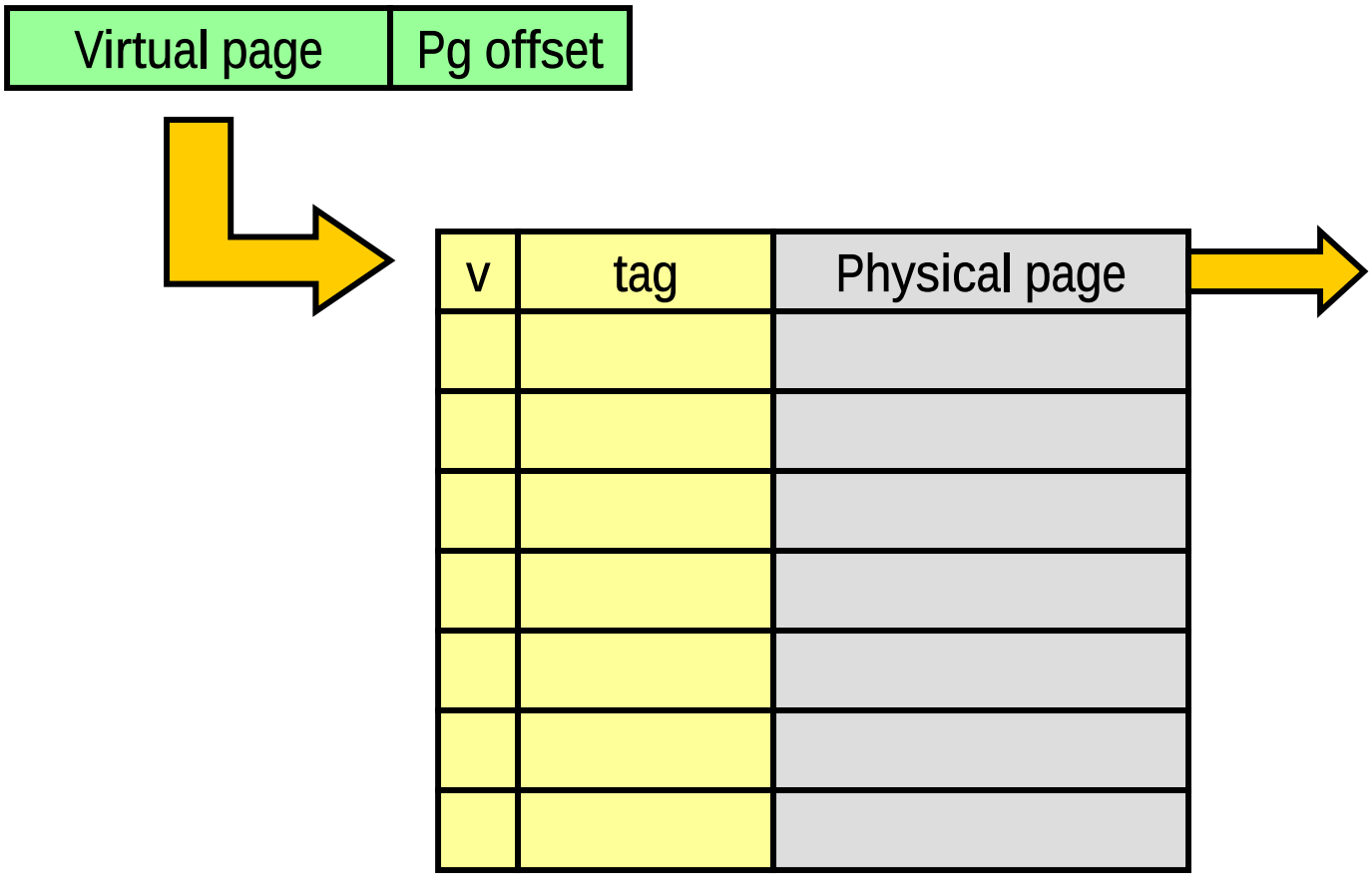
- To translate a virtual address into a physical address, we must first access the page table in physical memory.
- Then we access physical memory again to get (or store) the data
  - A load instruction performs at least 2 memory reads
  - A store instruction performs at least 1 read and then a write.
- Every memory instruction performs at least two accesses to main memory!

Translation look-aside buffer

- We fix this performance problem by avoiding main memory in the translation from virtual to physical pages.

- We buffer the common translations in a Translation Look-aside Buffer (TLB), a fast cache memory dedicated to storing a small subset of valid VtoP translations.

TLB



Where is the TLB lookup?

- We put the TLB lookup in the pipeline after the virtual address is calculated and before the memory reference is performed.
  - This may be before or during the data cache access.
  - Without a TLB hit we need to perform the translation during the memory stage of the pipeline.

Next Topic: Placing caches in a VM system

- VM systems give us two different addresses: virtual and physical

- Which address should we use to access the data cache?
  - Virtual address (before VM translation)
    - Faster access? More complex?
  - Physical address (after VM translations)
    - Delayed access?