Review (1/2)

- Caches are NOT mandatory:
  - Processor performs arithmetic
  - Memory stores data
  - Caches simply make data transfers go *faster*
- Each level of memory hierarchy is just a subset of next higher level
- Caches speed up due to temporal locality: store data used recently
- Block size > 1 word speeds up due to spatial locality: store words adjacent to the ones used recently

Review (2/2)

- Cache design choices:
  - size of cache: speed v. capacity
  - direct-mapped v. associative
  - for N-way set assoc: choice of N
  - block replacement policy
  - 2nd level cache?
  - Write through v. write back?
- Use performance model to pick between choices, depending on programs, technology, budget, ...

Overview

- Generalized Caching
- Address Translation
- Virtual Memory / Page Tables
- TLBs
- Page Replacement Policy
- Multi-level Page Tables

Generalized Caching

- We've discussed memory caching in detail. Caching in general shows up over and over in computer systems
  - File system cache
  - Web page cache
  - Software memorization
  - Others?
- General idea: if something is expensive but we want to do it repeatedly, just do it once and *cache* the result.

Generalized Caching

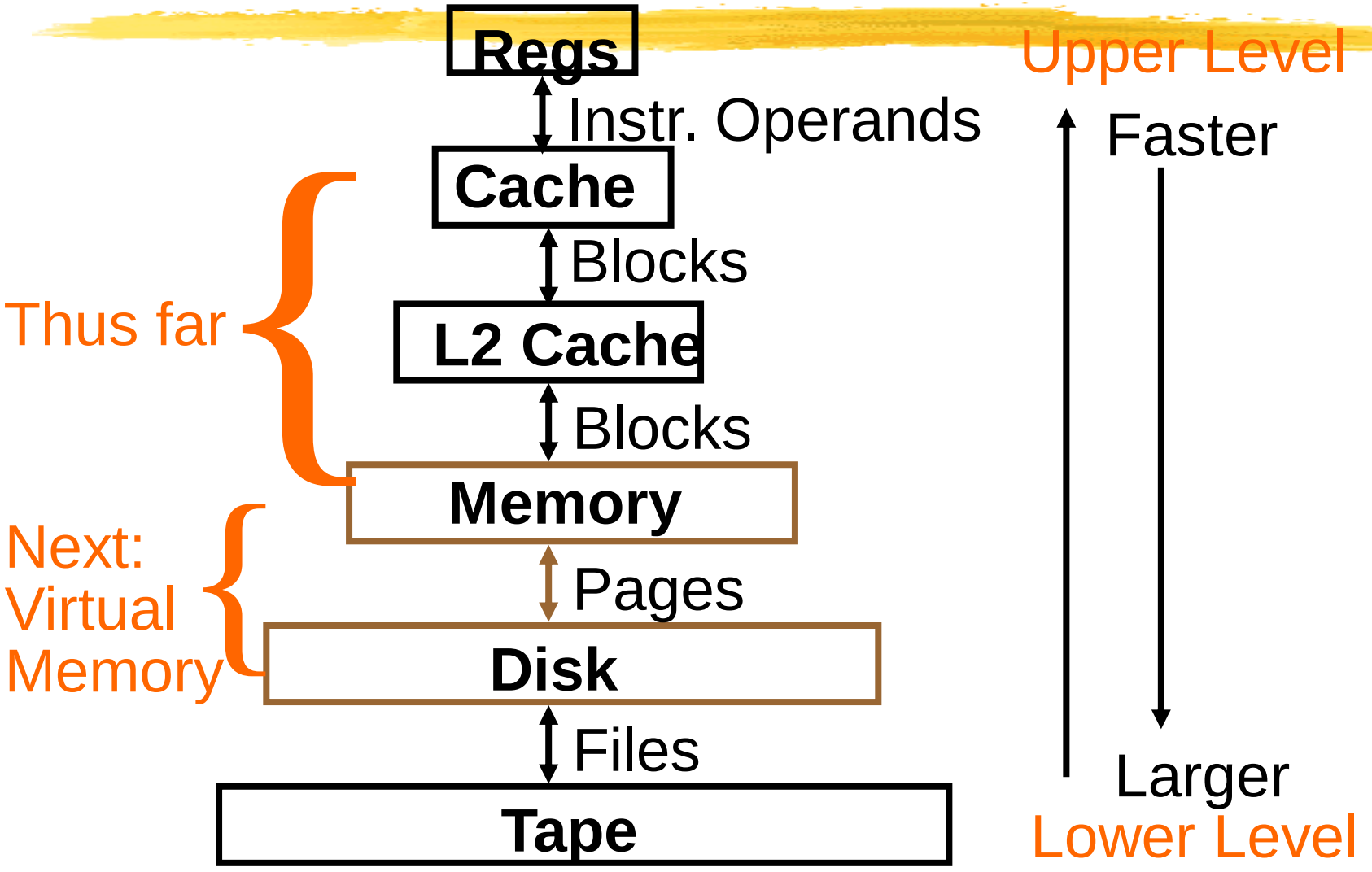
- For any caching system:

  average access time =

  hit rate * hit time + miss rate * miss time

- In general, hit time << miss time so we seek to improve performance by reducing the miss rate. Miss rate is influenced by:
  - Cache size
  - Layout policy (e.g., associativity)
  - Replacement policy

Another View of the Memory Hierarchy



Memory Hierarchy Requirements

- If Principle of Locality allows caches to offer (close to) speed of cache memory with size of DRAM memory, then recursively why not use at next level to give speed of DRAM memory, size of Disk memory?
- In order to have the appearance of a large memory, we explore the relationship between main memory and hard disk.

Virtual Memory

With respect to large memory, not only do we want the entire virtual address space to appear to be main memory, but in most cases we would also like this complete space to appear to be available to each program that is executing.

The job of the software and hardware that implement the virtual memory is to map each ***virtual address*** for each program into a ***physical address*** in the main memory.

It is possible for a virtual address from one program and a virtual address from another program to map the same physical address.

Basic Issues in VM System Design

- size of information blocks that are transferred from secondary to main storage (M)

- block of information brought into M, and M is full, then some region of M must be released to make room for the new block --> *replacement policy*

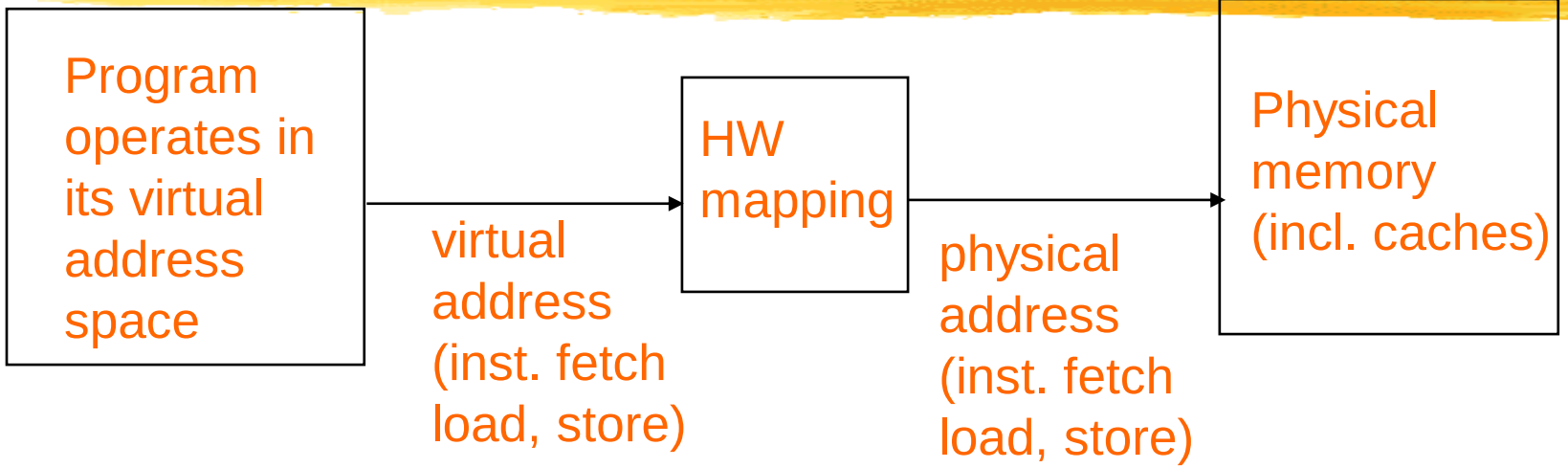
- which region of M is to hold the new block --> *placement policy*

- missing item fetched from secondary memory only on the occurrence of a fault --> *demand load policy*

Memory Hierarchy Requirements

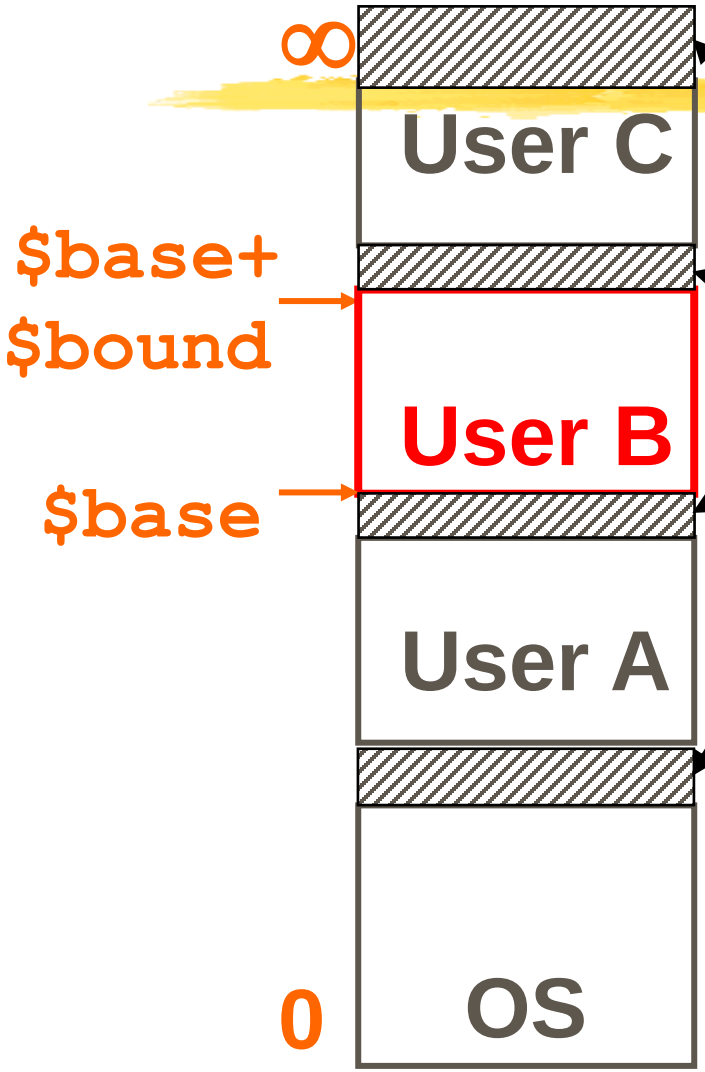
- Share memory between multiple processes but still provide protection – don't let one program read/write memory from another
- Address space – give each program the illusion that it has its own private memory
  - Suppose code starts at address 0x40000000. But different processes have different code, both residing at the same address. So each program has a different view of memory.

Virtual to Physical Addr. Translation



- Each program operates in its own virtual address space; ~only program running
- Each is protected from the other
- OS can decide where each goes in memory
- Hardware (HW) provides virtual -> physical mapping

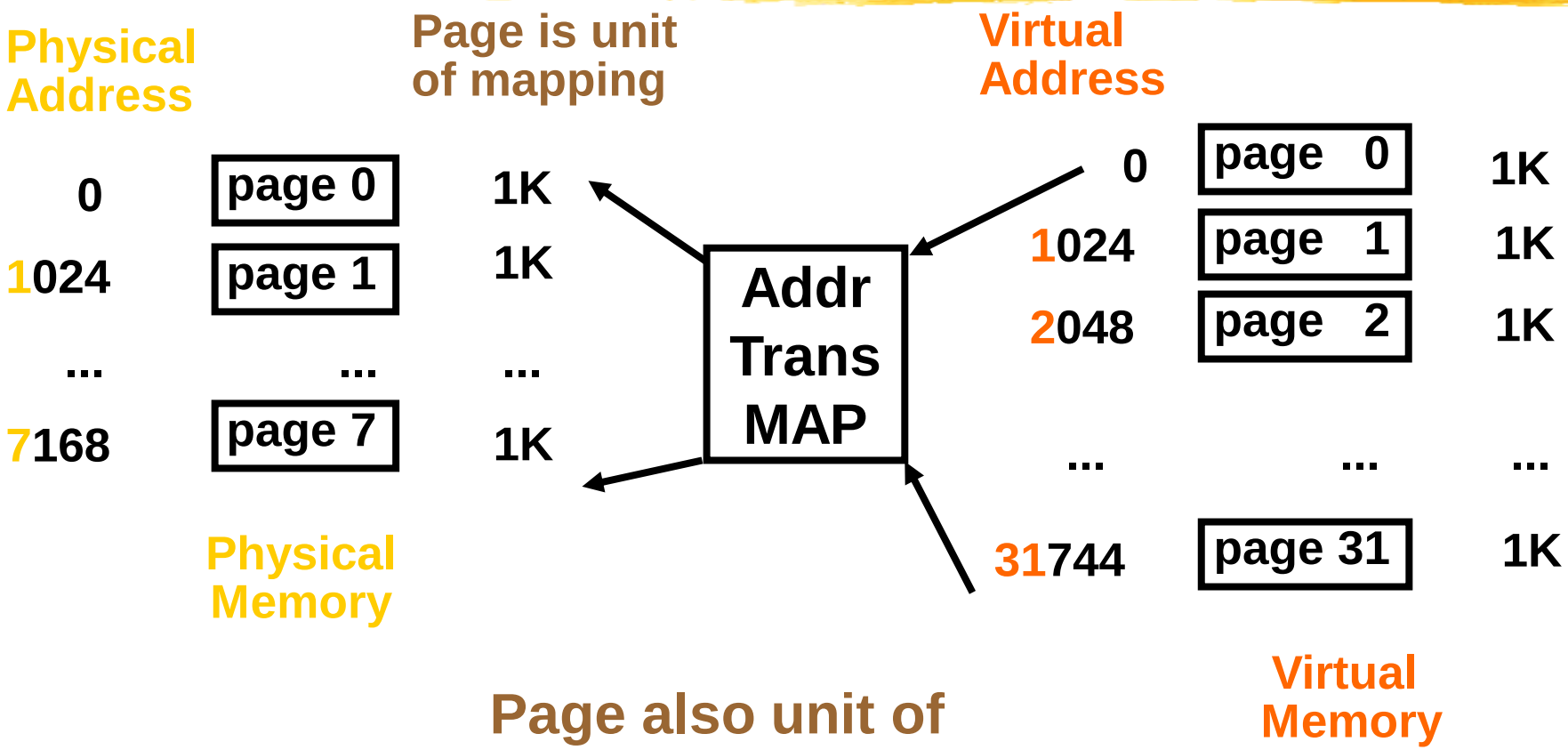
Simple Example: Base and Bound Reg



Enough space for User D, but discontinuous ("fragmentation problem")

Want discontinuous mapping
Process size >> mem
Addition not enough!
=> use Indirection!

Paging Organization (assume 1 KB pages)



Page also unit of transfer from disk to physical memory

Virtual Memory Mapping Function

- Cannot have simple function to predict arbitrary mapping
- Use table lookup of mappings

Use table lookup ("Page Table") for mappings: Page number is index

Virtual Memory Mapping Function

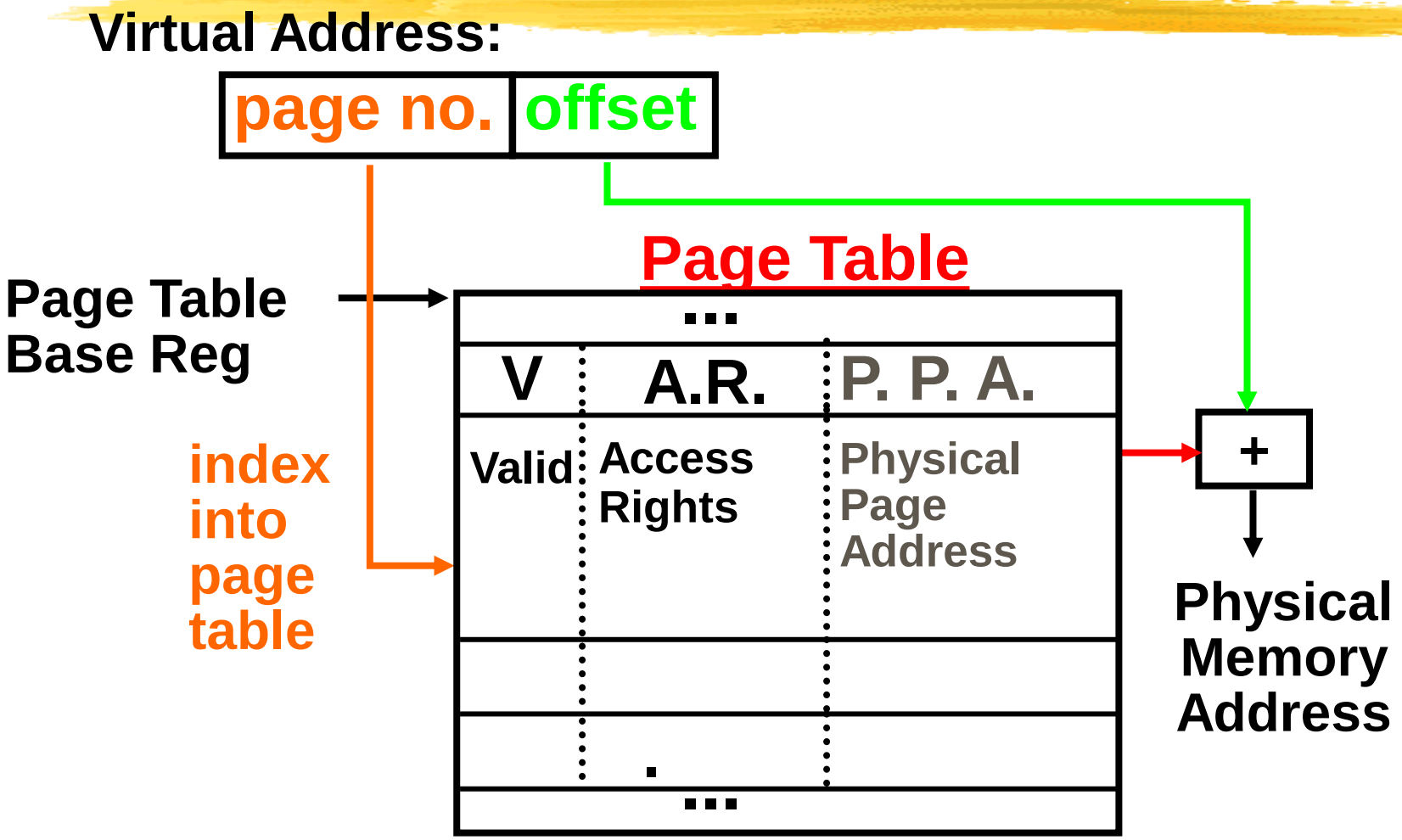
Physical Offset = Virtual Offset

Physical Page Number

= Page Table [Virtual Page Number]

(P.P.N. also called "Page Frame")

Address Mapping: Page Table



Page Table located in physical memory

Page Table

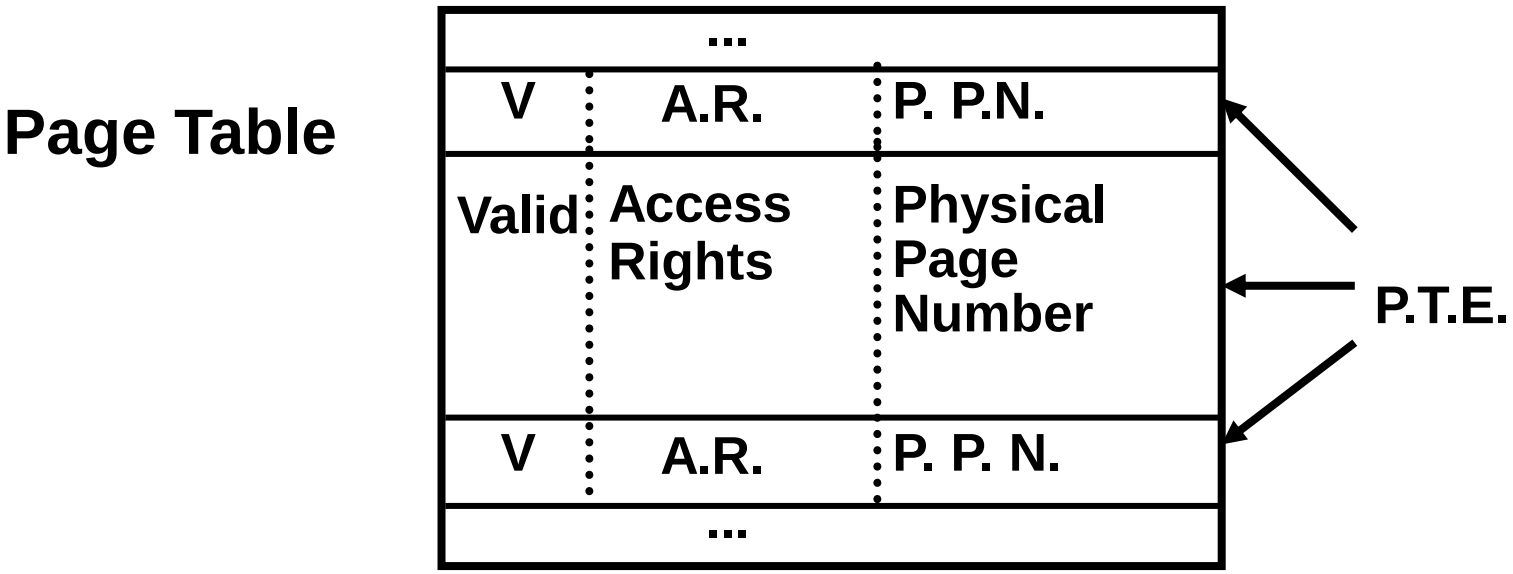
- A page table is an operating system structure which contains the mapping of virtual addresses to physical locations
  - There are several different ways, all up to the operating system, to keep this data around
- Each process running in the operating system has its own page table
  - "State" of process is PC, all registers, plus page table
  - OS changes page tables by changing contents of Page Table Base Register

Requirements revisited

- Remember the motivation for VM:
- Sharing memory with protection
  - Different physical pages can be allocated to different processes (sharing)
  - A process can only touch pages in its own page table (protection)
- Separate address spaces
  - Since programs work only with virtual addresses, different programs can have different data/code at the same address!
- What about the memory hierarchy?

Page Table Entry (PTE) Format

- Contains either Physical Page Number or indication not in Main Memory
- OS maps to disk if Not Valid (V = 0)

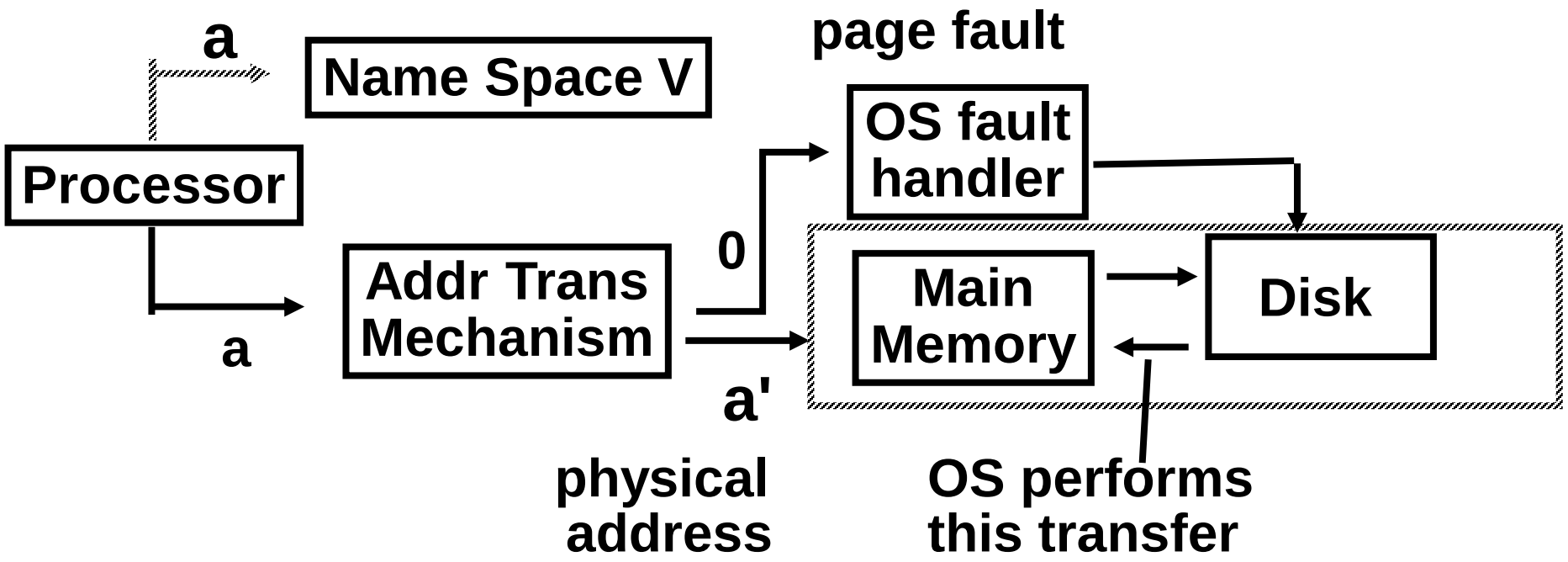


If valid, also check if have permission to use page: Access Rights (A.R.) may be Read Only, Read/Write, Executable

Address Map, Mathematically Speaking

V = {0, 1, . . . , n - 1} virtual address space (n > m)
M = {0, 1, . . . , m - 1} physical address space
MAP: V --> M U {θ} address mapping function

MAP(a) = a' if data at virtual address a is present in
physical address a' and a' in M
= θ if data at virtual address a is not present in M



Comparing the 2 levels of hierarchy

Cache Version	Virtual Memory version
Block or Line	Page
Miss	Page Fault
Block Size: 32-64B	Page Size: 4K-8KB
Placement: Direct Mapped, N-way Set Associative	Fully Associative
Replacement: LRU or Random	Least Recently Used (LRU)
Write Thru or Back	Write Back

Notes on Page Table

- Solves Fragmentation problem: all chunks same size, so all holes can be used
- OS must reserve "Swap Space" on disk for each process
- To grow a process, ask Operating System
  - If unused pages, OS uses them first
  - If not, OS swaps some old pages to disk
  - (Least Recently Used to pick pages to swap)
- Each process has own Page Table
- Will add details, but Page Table is essence of Virtual Memory

Translation Lookaside Buffer (TLBs)

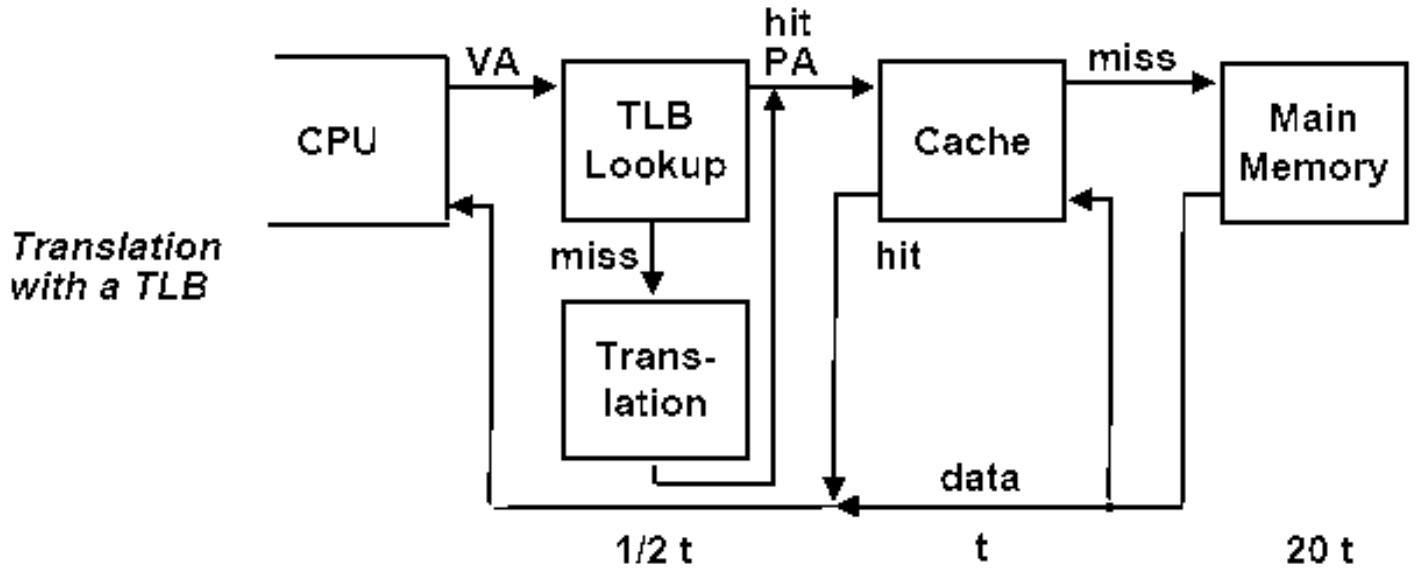
A way to speed up VM translation is to use a special cache of recently used page table entries -- this has many names, but the most frequently used is *Translation Lookaside Buffer* or *TLB*

Virtual Address	Physical Address	Dirty	Ref	Valid	Access

Really just a cache on the page table mappings TLB access time comparable to cache access time (much less than main memory access time)

Translation Lookaside Buffer (TLBs)

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped. TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.



Example-1

A virtual memory system uses 4KByte pages, 64-bit words, and a 48-bit virtual address. A particular program and its data require 4395 pages.

a- What is the minimum number of page tables required?

b-How many entries are there in the last page table.

Example-2

A small TLB has the following entries for a virtual page number of length 20 bits, a physical page number of 12 bits, and a page offset of 12 bits.

Valid-bit	Dirty-bit	Tag (Virtual Page Number)	Data (Physical Page Number)
1	1	01AF4	FFF
0	0	0E45F	E03
0	0	012FF	2F0
1	0	01A37	788
1	0	02BB4	45C
0	1	03CA0	657

The page number and offset are hexadecimal. For each of the virtual address listed, indicate a hit occurs and if it does, give the physical address : (a) 02BB4A65 (b)0E45FB32 (c)0D34E9DC (d)03CA0777

Page Selection Policy

- **Demand paging:**
  - Load page in response to access (page fault)
  - Predominant selection policy
- **Pre-paging (prefetching):**
  - Predict what pages will be accessed in near future
  - Prefetch pages in advance of access
  - Problems:
    - * Hard to predict accurately (trace cache)
    - * Mispredictions can cause useful pages to be replaced
- **Overlays**
  - Application controls when pages loaded/replaced
  - Only really relevant now for embedded/real-time systems

Page Replacement Policy

- **Random**

– Works surprisingly well

- **FIFO (first in, first out)**

– Throw out oldest pages

- **MIN (or OPTIMAL)**

– Throw out page used farthest in the future

- **LRU (least recently used)**

– Throw out page not used in longest time

- **NFU/Clock (not frequently used)**

– Approximation to LRU _ do not throw out recently used pages

FIFO Page Replacement

FIFO: replace oldest page (first loaded)

Example:

- **Memory system with three pages _ all initially free**
- **Reference string: A B C A B D A D B C B**

	A	B	C	A	B	D	A	D	B	C	B
Frame1	(A)					(D)				(C)	
Frame2		(B)					(A)				
Frame3			(C)						(B)		

Result: 7 page faults

Optimal Page Replacement

Optimal: replace page used farthest in future
– If several equal candidates, select one at random
Example:
– Memory system with three pages _ all initially free
– Reference string: A B C A B D A D B C B

	A	B	C	A	B	D	A	D	B	C	B
Frame1	(A)									(C)	
Frame2		(B)								↕	
Frame3			(C)			(D)				(C)	

Result: 5 page faults

LRU Page Replacement

LRU: replace page used farthest in past
Example:
– Memory system with three pages _ all initially free
– Reference string: A B C A B D A D B C B

	A	B	C	A	B	D	A	D	B	C	B
Frame1	(A)									(C)	
Frame2		(B)									
Frame3			(C)			(D)					

Result: 5 page faults

Example-1

In a paged system, suppose the following pages are requested in the order shown;

12, 14, 2, 34, 56, 23, 14, 56, 34, 12

and the main memory partition can only hold four pages at any instant (in practice usually many more pages can be held). List the pages in the main memory after each page is transferred using each of the following replacement algorithms:

(a) First-in-first-out replacement algorithm

(b) Least recently used replacement algorithm

(c) Clock replacement algorithm

Example-2

Using one use bit with each page entry, list the pages recorded in the following sequence and hence determine the pages removed using one use bit approximation to the least recently used algorithm:

13, 47, 13, 99, 47, 35, 13, 67, 47, 13, 34, 35, 99, 99, 14, 14, 47, 67

given that there are four pages in the memory partition. The use bits are only read at page fault time, and then scanned in the same order.

Suppose two use bits are provided. The first use bit is set when the page is first referenced. The second use bit is set when the page is referenced again. Deduce an algorithm to remove pages from the partition, and list the pages.

Things to Remember

- Apply Principle of Locality Recursively
- Manage memory to disk? Treat as cache
  - Included protection as bonus, now critical
  - Use Page Table of mappings vs. tag/data in cache
- Virtual Memory allows protected sharing of memory between processes with less swapping to disk, less fragmentation than always swap or base/bound