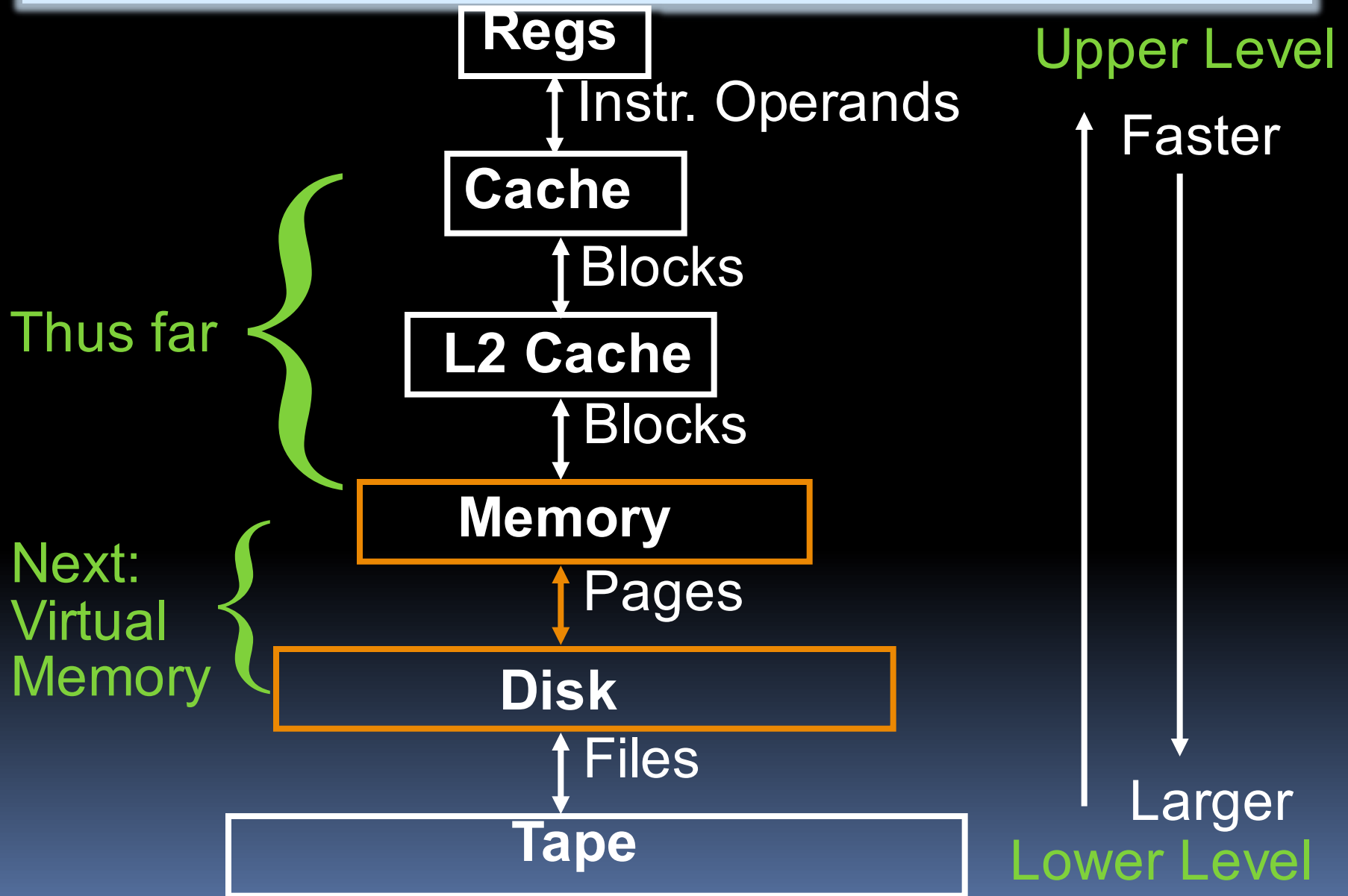


Review

- Cache design choices:
 - Size of cache: speed v. capacity
 - Block size (i.e., cache aspect ratio)
 - Write Policy (Write through v. write back)
 - Associativity choice of N (direct-mapped v. set v. fully associative)
 - Block replacement policy
 - 2nd level cache?
 - 3rd level cache?
- Use performance model to pick between choices, depending on programs, technology, budget, ...

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?
- While we're at it, what other things do we need from our memory system?

Memory Hierarchy Requirements

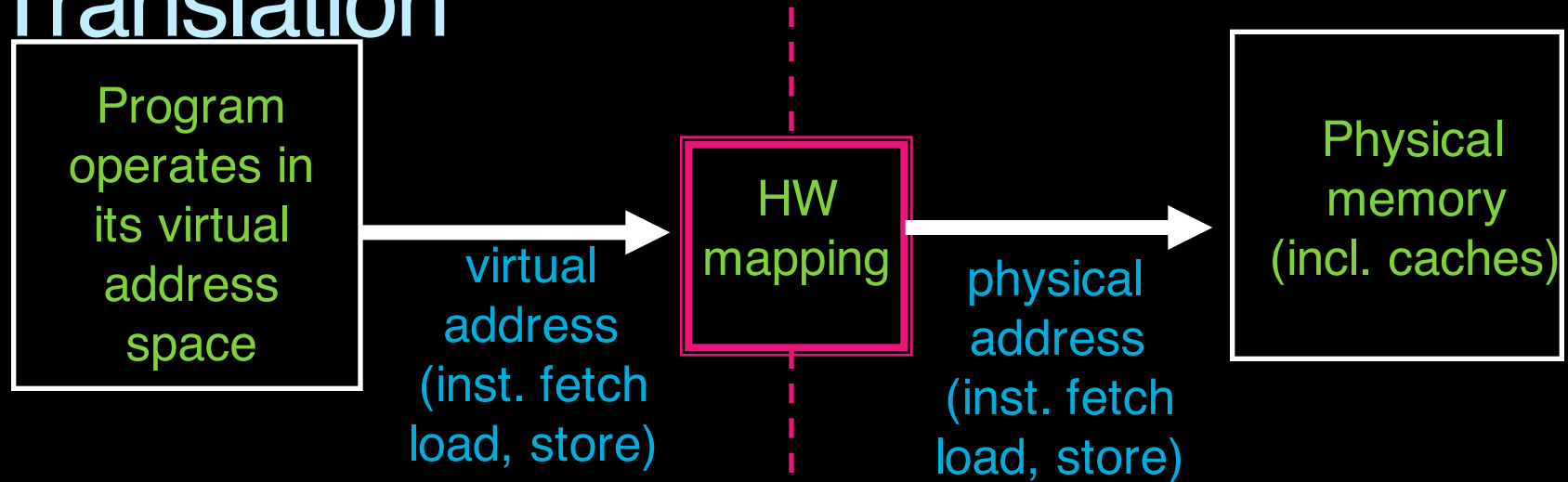
- Allow multiple **processes** to simultaneously occupy memory and provide protection – don't let one program read/write memory from another
- Address space – give each program the **illusion** that it has its own private memory
 - Suppose code starts at address 0x40000000. But different processes have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


Virtual Memory

- Next level in the memory hierarchy:
 - Provides program with illusion of a very large main memory:
 - Working set of “pages” reside in main memory - others reside on disk.
- Also allows OS to share memory, protect programs from each other
- Today, more important for **protection** vs. just another level of memory hierarchy
- Each process thinks it has all the memory to itself
- (Historically, it predates caches)

Virtual to Physical Address

Translation

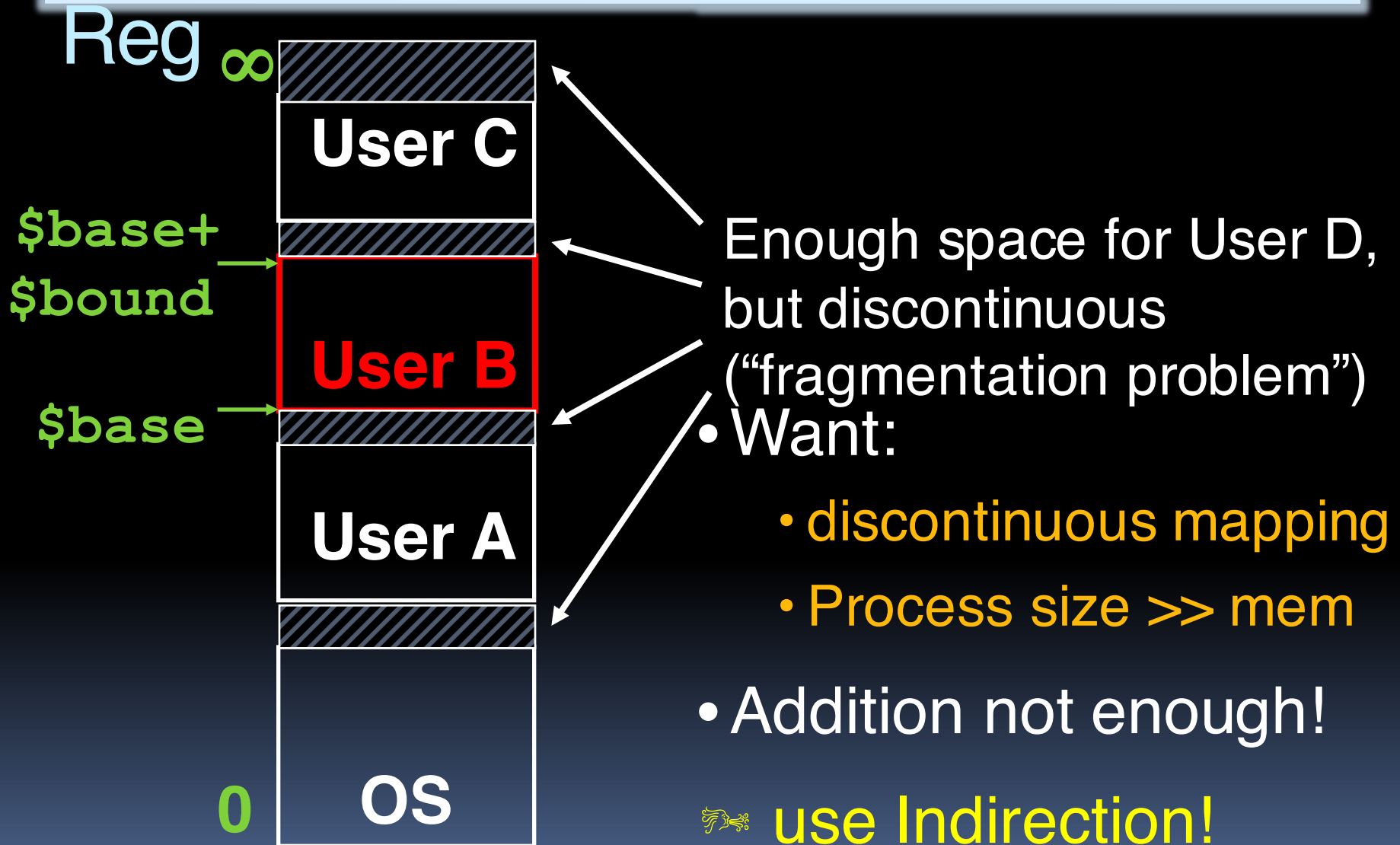


- Each program operates in its own virtual address space; ~only program running
- Each is protected from the other
- OS can decide where each goes in memory
- Hardware gives virtual  physical mapping

Analogy

- Book title like **virtual address**
- Library of Congress call number like **physical address**
- Card catalogue like **page table**, mapping from book title to call #
- On card for book, in local library vs. in another branch like **valid bit** indicating in main memory vs. on disk
- On card, available for 2-hour in library use (vs. 2-week checkout) like **access rights**

Simple Example: Base and Bound

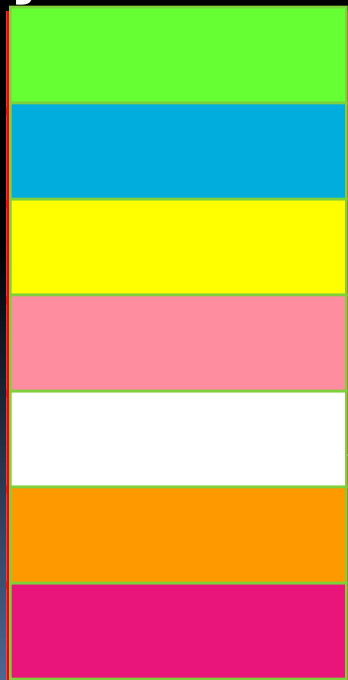


Mapping Virtual Memory to Physical Memory

- Divide into equal sized chunks (about 4 KB - 8 KB)
- Any chunk of Virtual Memory assigned to any chunk of Physical Memory (“page”)

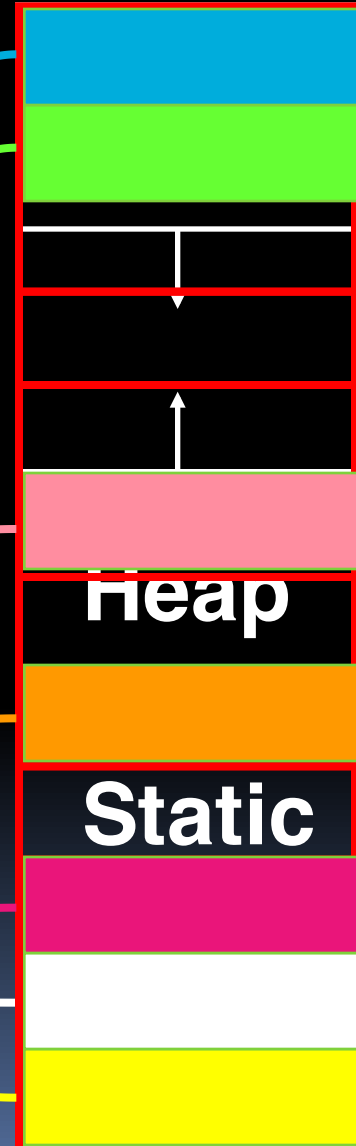
64 MB **Physical Memory**

0



Virtual Memory

∞



0

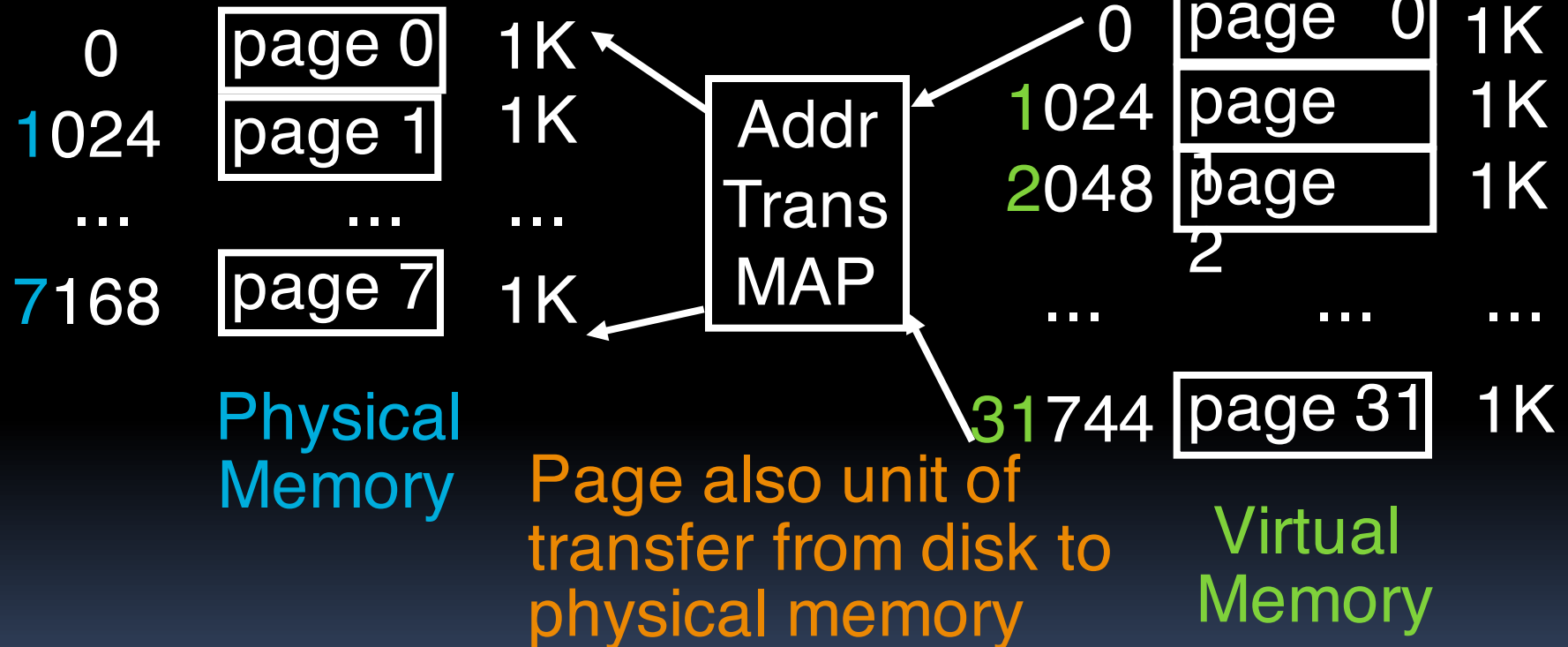
Paging Organization (assume 1 KB

pages)

Physical
Address

Page is unit
of mapping

Virtual
Address



Virtual Memory Mapping Function

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

Page Number	Offset
-------------	--------

- Use table lookup (“Page Table”) for mappings: Page number is index
- Virtual Memory Mapping Function
 - $\text{Physical Offset} = \text{Virtual Offset}$
 - $\text{Physical Page Number} = \text{PageTable}[\text{Virtual Page Number}]$
(P.P.N. also called “Page Frame”)

Address Mapping: Page Table

Virtual Address:

page no. offset

Page Table

Page Table
Base Reg

index
into
page
table

...		
V	A.R.	P. P. A.
Val	Access	Physical
-id	Rights	Page
		Address
...		

+

Physical
Memory
Address

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 ($V = 0$)

Page Table

...		
V	A.R.	P. P.N.
Val	Access	Physical
-id	Rights	Page
		Number
V	A.R.	P. P. N.
...		

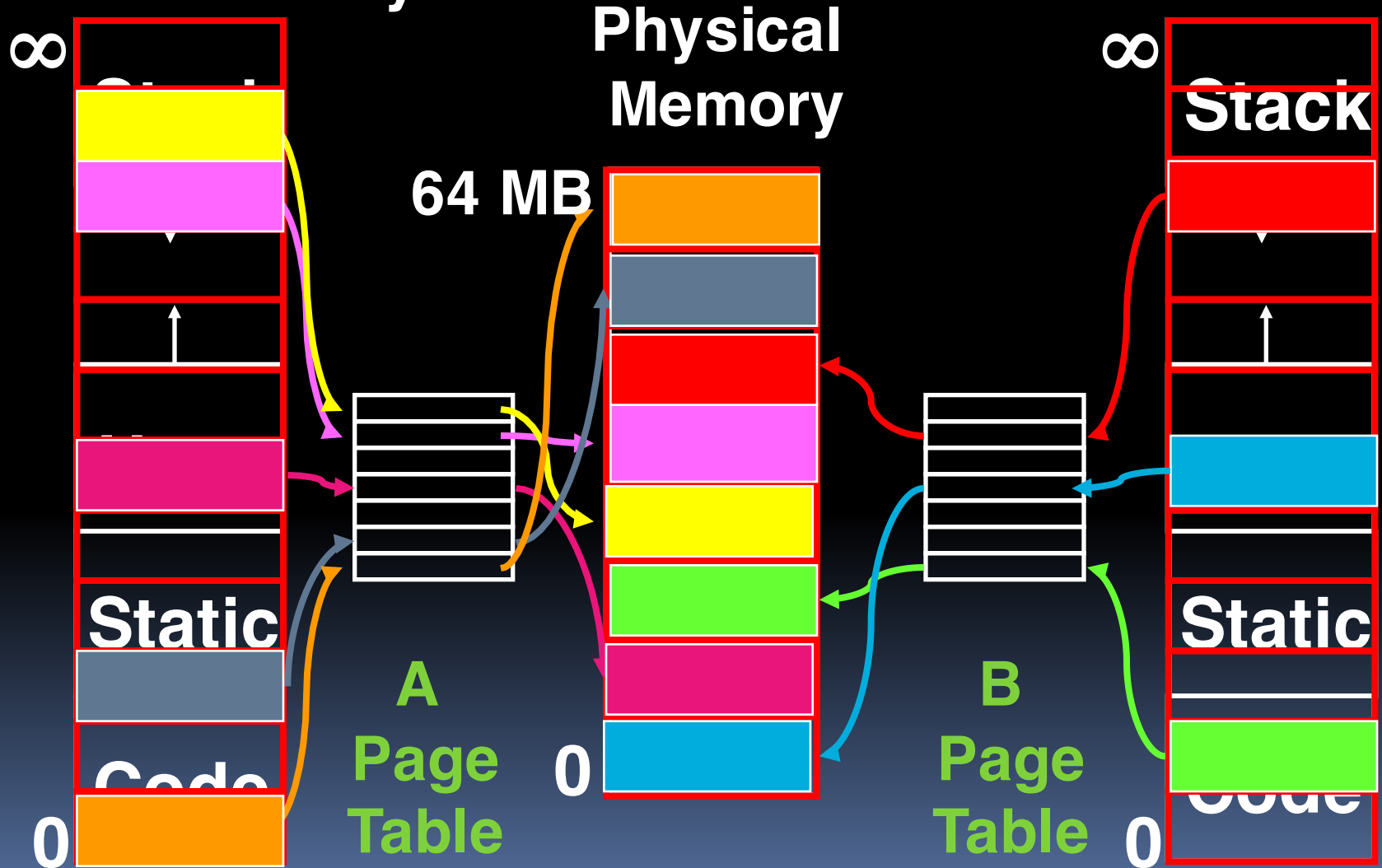
P.T.E.

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

Paging/Virtual Memory Multiple

User A:
Processes
Virtual Memory

User B:
Processes
Virtual Memory



Comparing the 2 levels of hierarchy

Cache version

Block or Line

Miss

Block Size: 32-64B

Placement:

Direct Mapped,
N-way Set Associative

Replacement:

LRU or Random

Write Thru or Back

Virtual Memory vers.

Page

Page Fault

Page Size: 4K-8KB

Fully Associative

Least Recently Used
(LRU)

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

Why would a process need to “grow”?

- A program's *address space* contains 4 regions:

- *stack*: local variables, grows downward
- *heap*: space requested for pointers via `malloc()`; resizes dynamically, grows upward
- *static data*: variables declared outside main, does not grow or shrink
- *code*: loaded when program starts, does not change



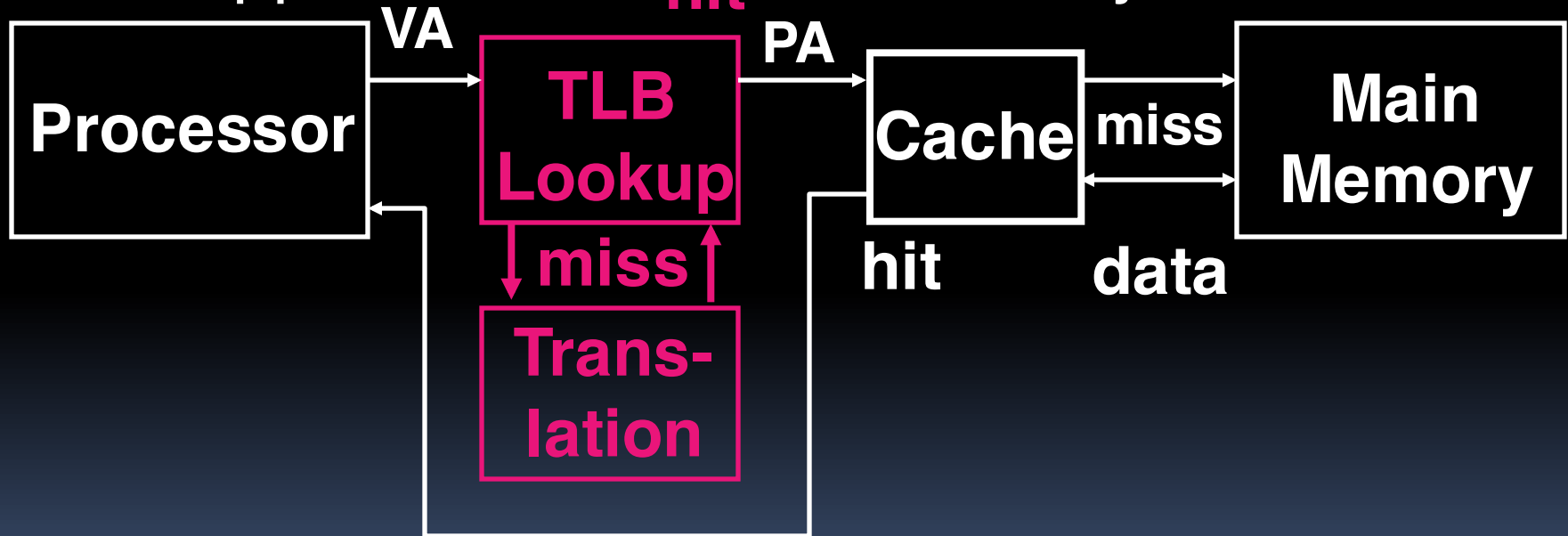
For now, OS somehow prevents accesses between stack and heap (gray hash lines)

Virtual Memory Problem #1

- Map every address \Rightarrow 1 indirection via Page Table in memory per virtual address \Rightarrow 1 virtual memory accesses = 2 physical memory accesses \Rightarrow SLOW!
- Observation: since locality in pages of data, there must be locality in **virtual address translations** of those pages
- Since small is fast, why not use a small cache of virtual to physical address translations to make translation fast?
- For historical reasons, cache is called a **Translation Lookaside Buffer**, or **TLB**

Translation Look-Aside Buffers

- TLBs usually small, typically 128 - 256 entries
- Like any other cache, the TLB can be direct mapped, set associative, or fully associative



On TLB miss, get page table entry from main memory

Peer Instruction

- 1) Locality is important yet different for cache and virtual memory (VM): temporal locality for caches but spatial locality for VM
- 2) VM helps both with security and cost

	12
a)	FF
b)	FT
c)	TF
d)	TT

Peer Instruction Answer

- 1) Locality is important yet different for cache and virtual memory (VM): temporal locality for caches but spatial locality for VM

FALSE

1. No. Both for VM and cache


- 2) VM helps both with security and cost

TRUE

2. Yes. Protection and
a bit smaller memory

	12
a)	FF
b)	FT
c)	TF
d)	TT

And in conclusion...

- Manage memory to disk? Treat as cache
 - Included protection as bonus, now critical
 - Use Page Table of mappings for each user vs. tag/data in cache
 - TLB is cache of Virtual  Physical addr trans
- Virtual Memory allows protected sharing of memory between processes
- Spatial Locality means Working Set of Pages is all that must be in memory for process to run fairly well