CMPT 300

Introduction to Operating Systems

Virtual Memory

Agenda

- Virtual Memory Intro
- Page Tables
- Translation Lookaside Buffer
- Demand Paging
- System Calls
- Summary

Overarching Theme for Today

"Any problem in computer science can be solved by an extra level of indirection."

- Often attributed to Butler Lampson (Berkeley PhD and Professor, Turing Award Winner), who in turn, attributed it to David Wheeler, a British computer scientist, who also said "*… except for the problem of too many layers of indirection!*"

Butler Lampson

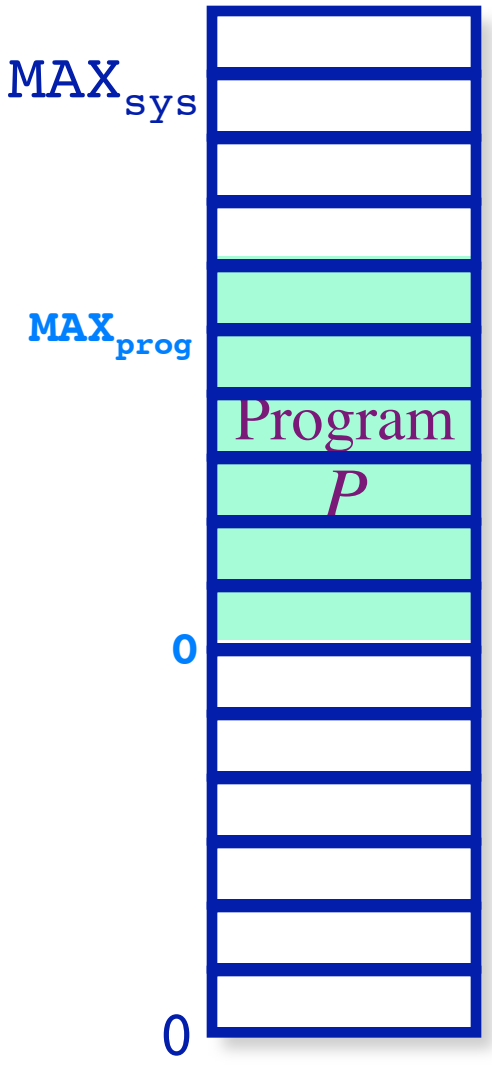
Virtualizing Resources

- Different Processes share the same HW
  - multiplex CPU (finished: scheduling)
  - multiplex and share Memory (Today)
  - multiplex disk and devices (later in term)
- Why worry about memory sharing?
  - complete working state of a process and/or kernel is defined by its data in memory (and registers)
  - Don't want different processes to have access to each other's memory (protection)

Basic Memory Management Concepts

Address spaces

- *Physical address space* — The address space supported by the hardware
  - Starting at address 0, going to address $\mathtt{MAX}_{\mathtt{sys}}$
- *Logical/virtual address space* — A process's view of its own memory
  - Starting at address 0, going to address $\mathtt{MAX}_{\mathtt{prog}}$



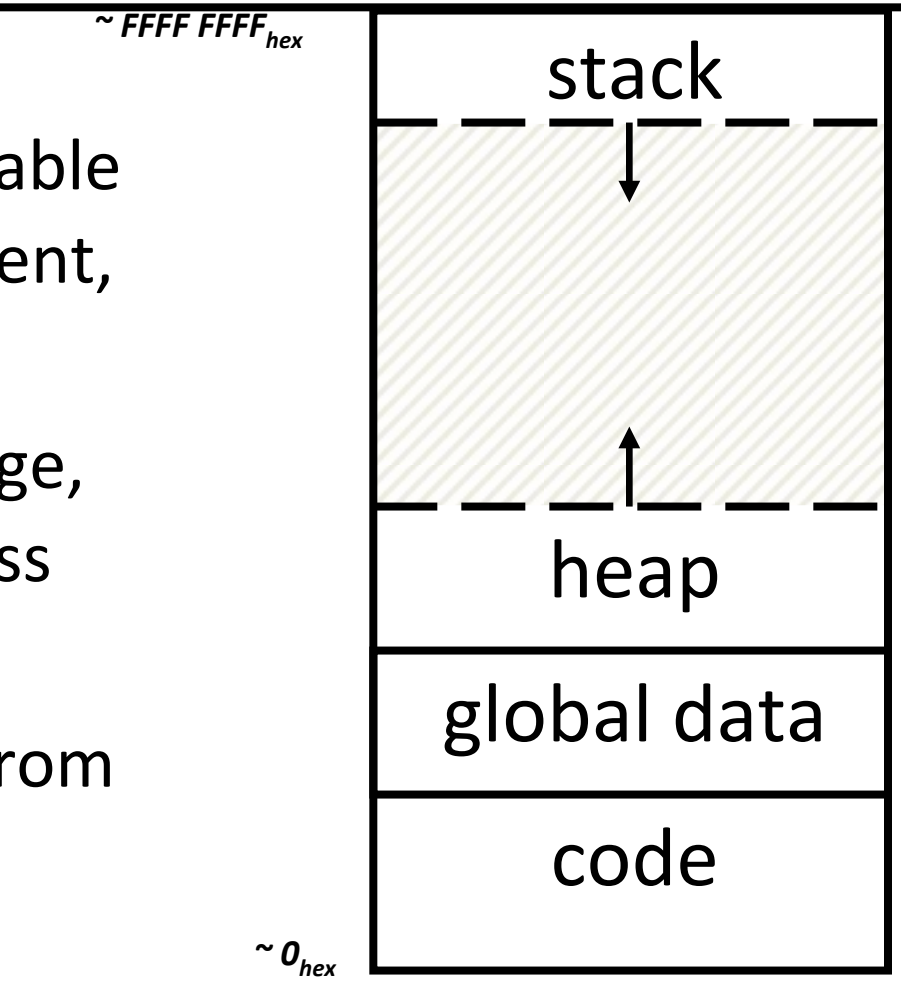
But where do addresses come from?

```
MOV r0, @0xfffa620e
```

- Which is bigger, physical or virtual address space?
  - A. Physical address space
  - B. Virtual address space
  - C. It depends on the system.

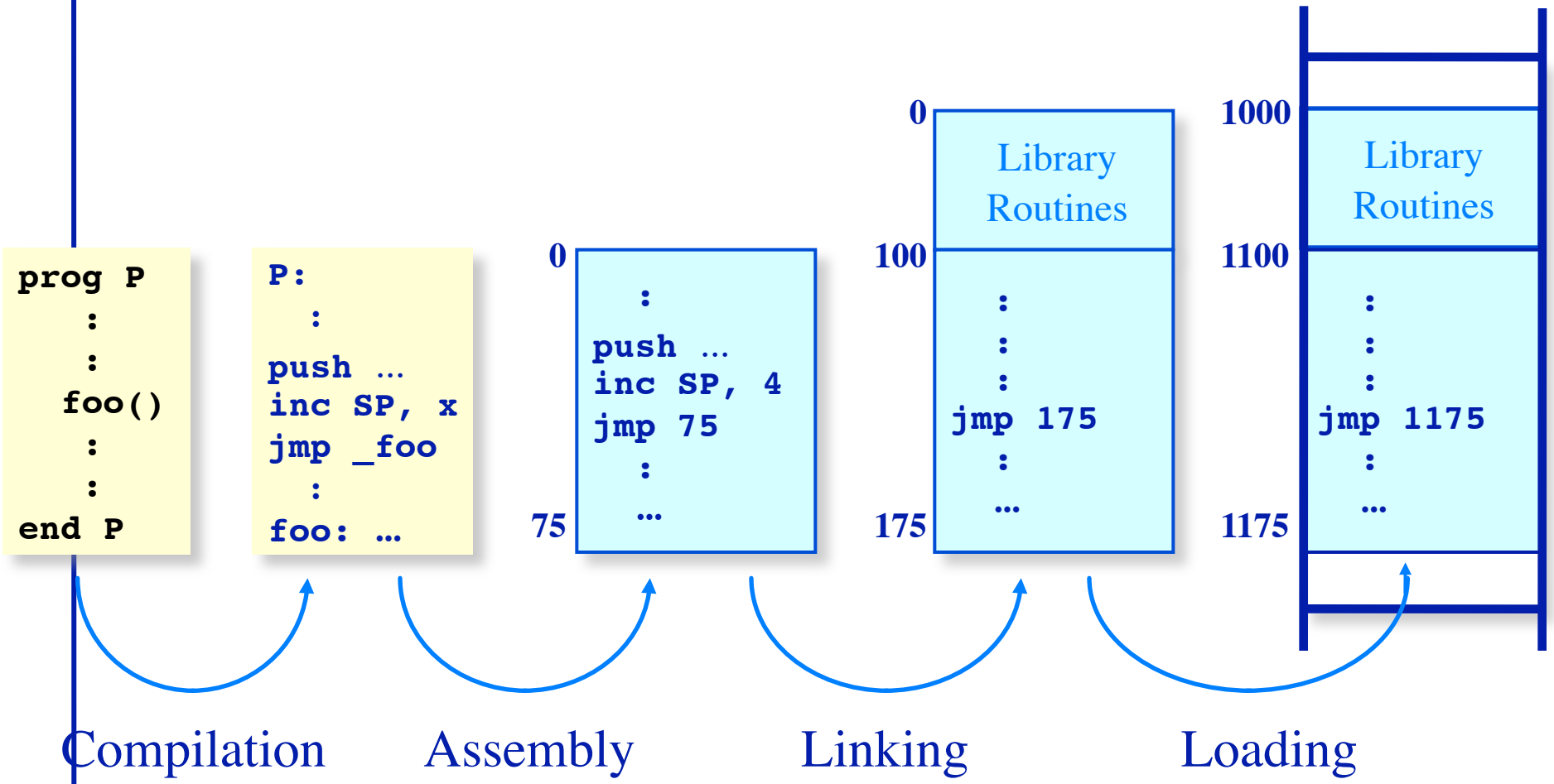
Review: Memory Management

- **Static storage**: global variable storage, basically permanent, entire program run
- **Stack**: local variable storage, parameters, return address
- **Heap** (dynamic storage): `malloc()` grabs space from here, `free()` returns it



Basic Concepts
Address generation

- The compilation pipeline

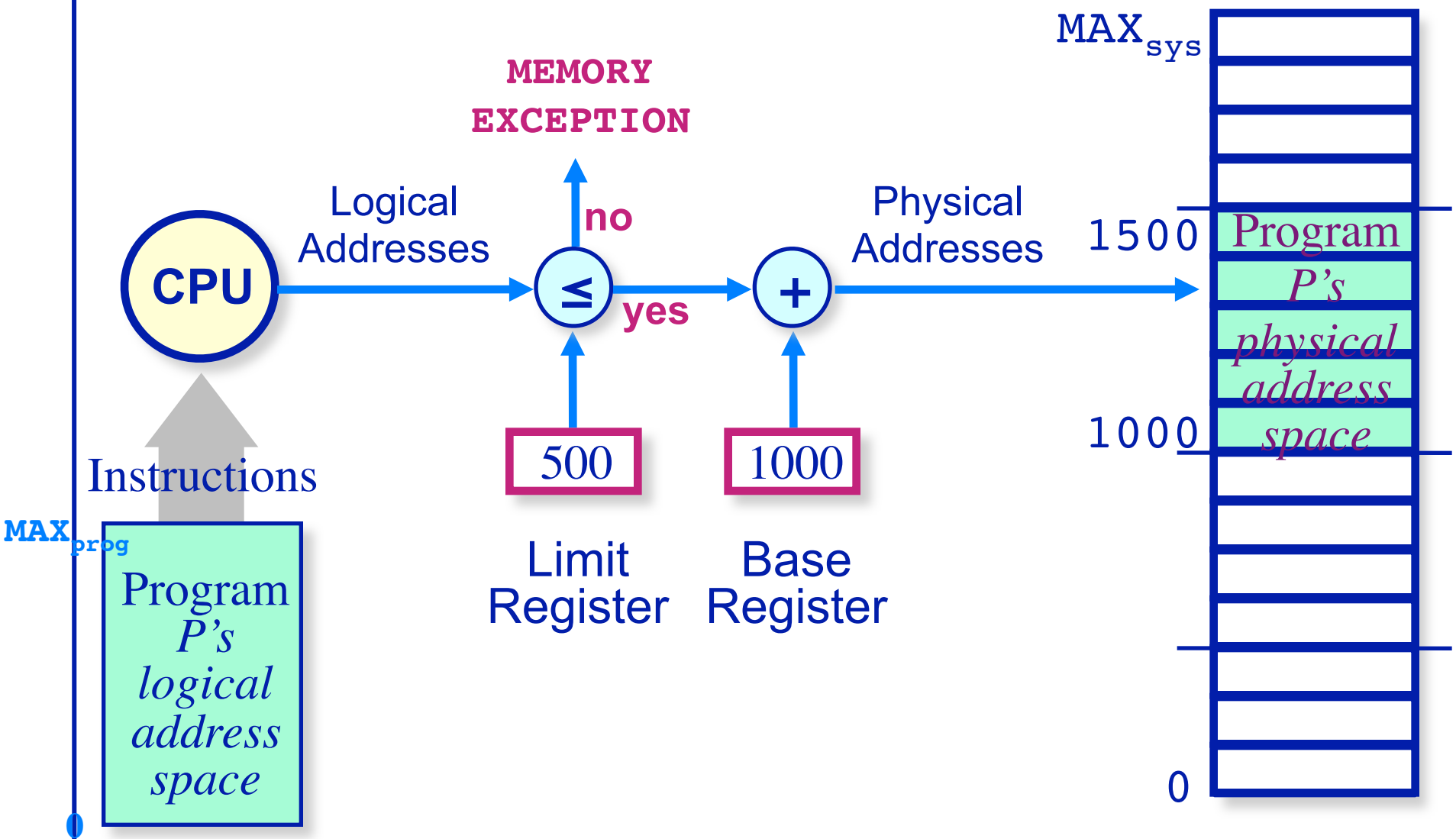


Program Relocation

- Program issues virtual addresses
- Machine has physical addresses.
- If virtual == physical, then how can we have multiple programs resident concurrently?
- Instead, relocate virtual addresses to physical at run time.
  - While we are relocating, also bounds check addresses for safety.
- I can relocate that program (safely) in two registers…

Basic Concepts (Cont'd.)

Address Translation

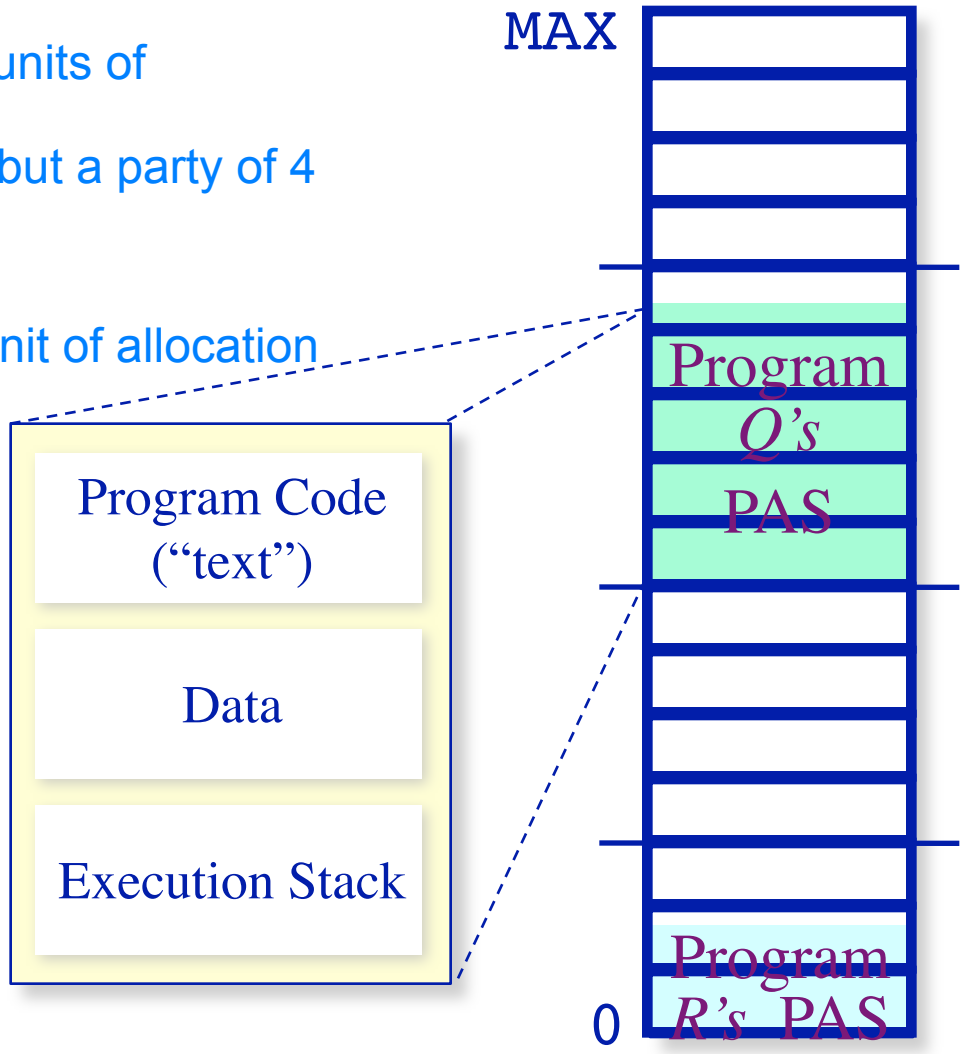


- With base and bounds registers, the OS needs a hole in physical memory at least as big as the process.
  - A. True
  - B. False

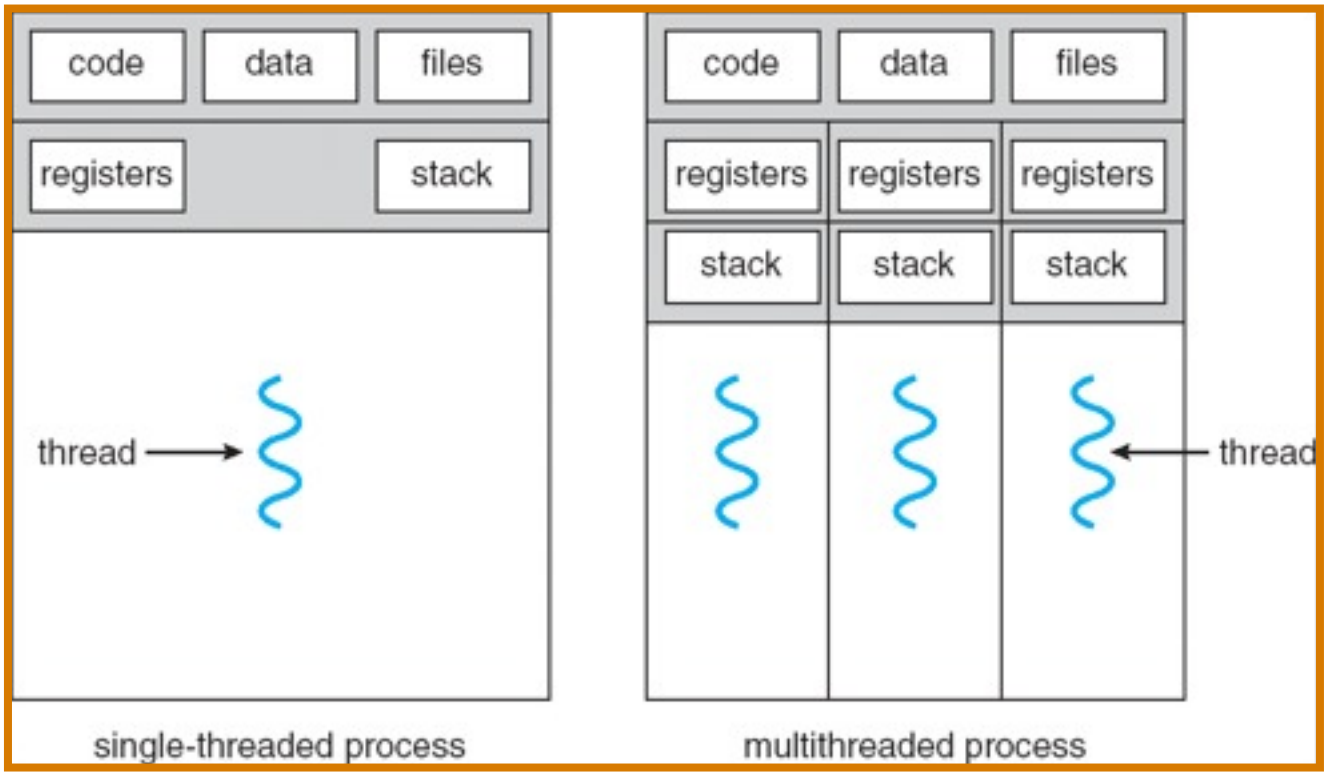
Evaluating Dynamic Allocation Techniques

The fragmentation problem

- External fragmentation
  - Unused memory between units of allocation
  - E.g, two fixed tables for 2, but a party of 4
- Internal fragmentation
  - Unused memory within a unit of allocation
  - E.g., a party of 3 at a table for 4



Recall: Single and Multithreaded Processes



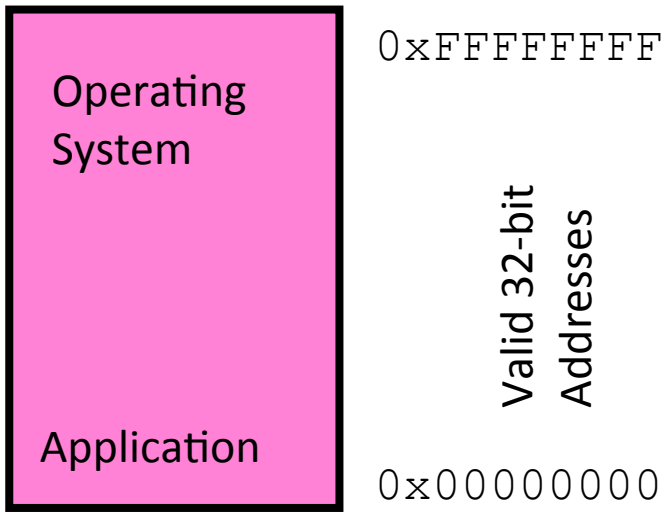
- Threads encapsulate concurrency
  - "Active" component of a process
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system

Aspects of Memory Multiplexing

- Controlled overlap:
  - State of threads should not collide in physical memory. Obviously, unexpected overlap causes chaos!
  - Converse : ability to overlap when desired (for communication)
- Translation:
  - When translation exists, processor uses virtual addresses, physical memory uses physical addresses. (allows relocation)
- Protection:
  - Prevent access to private memory of other processes
    - Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
    - Kernel data protected from User programs. programs from each other

Uniprogramming

- Uniprogramming (no Translation or Protection)
  - Application always runs at same place in physical memory since only one application at a time
  - Application can access any physical address



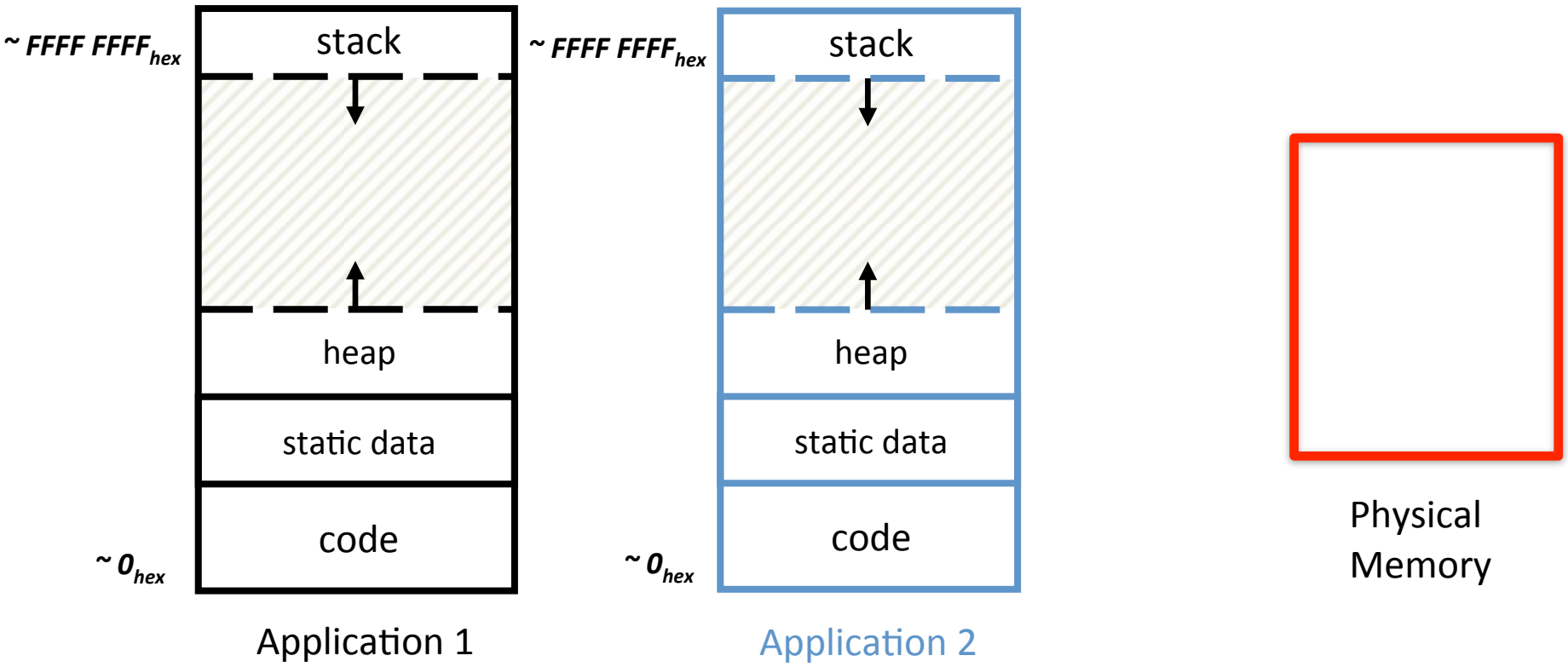
  - Application given illusion of dedicated machine by giving it reality of a dedicated machine

The Problem

- What if less physical memory than full address space?
  - 32 bit addresses => 4 GB address space, RAM hasn't always been larger than 4 GB.
  - 64 bit addresses => 16 exibytes.
- What if we want to run multiple programs at the same time?

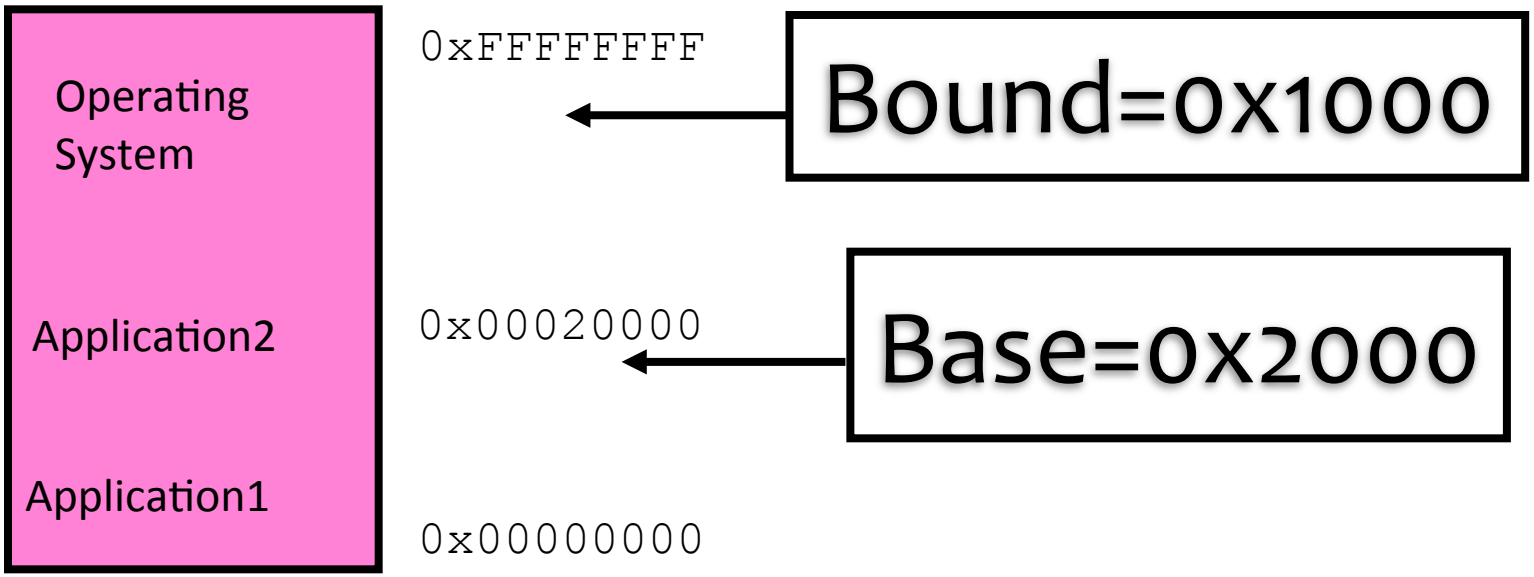
The Problem

- Limited physical memory, one or more programs each with their own address



Multiprogramming (Version with

- Can we protect programs from each other without translation?



  - Yes: use two special registers *base* and *bound* to prevent user from straying outside designated area
    - If user tries to access an illegal address, cause an error

Idea #1: Segmentation

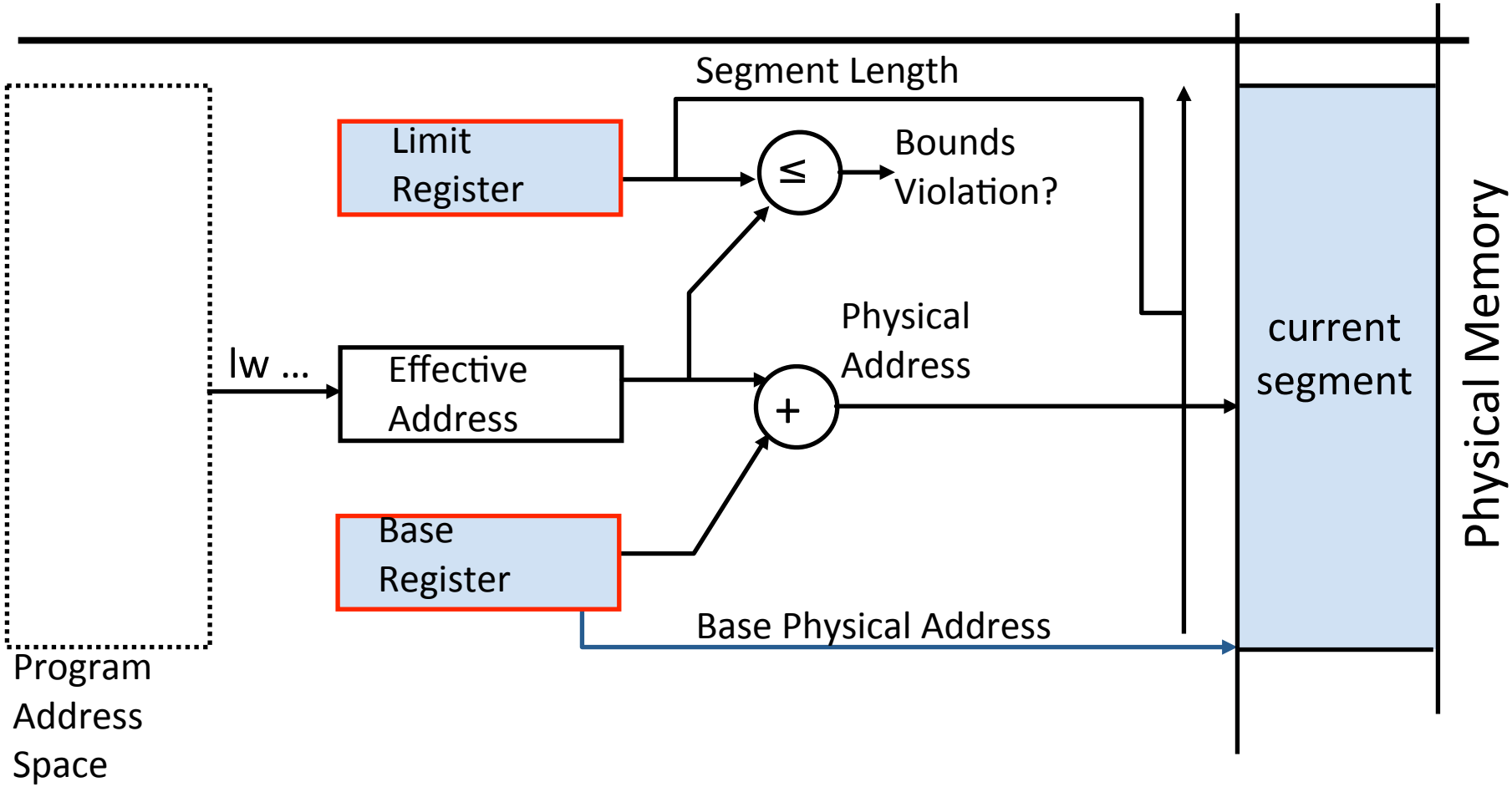
Location-independent programs

- Program doesn't need to specify its absolute memory addresses; need for a *base register*

Protection

- Independent programs should not affect each other inadvertently: need for a *limit (or bound) register*

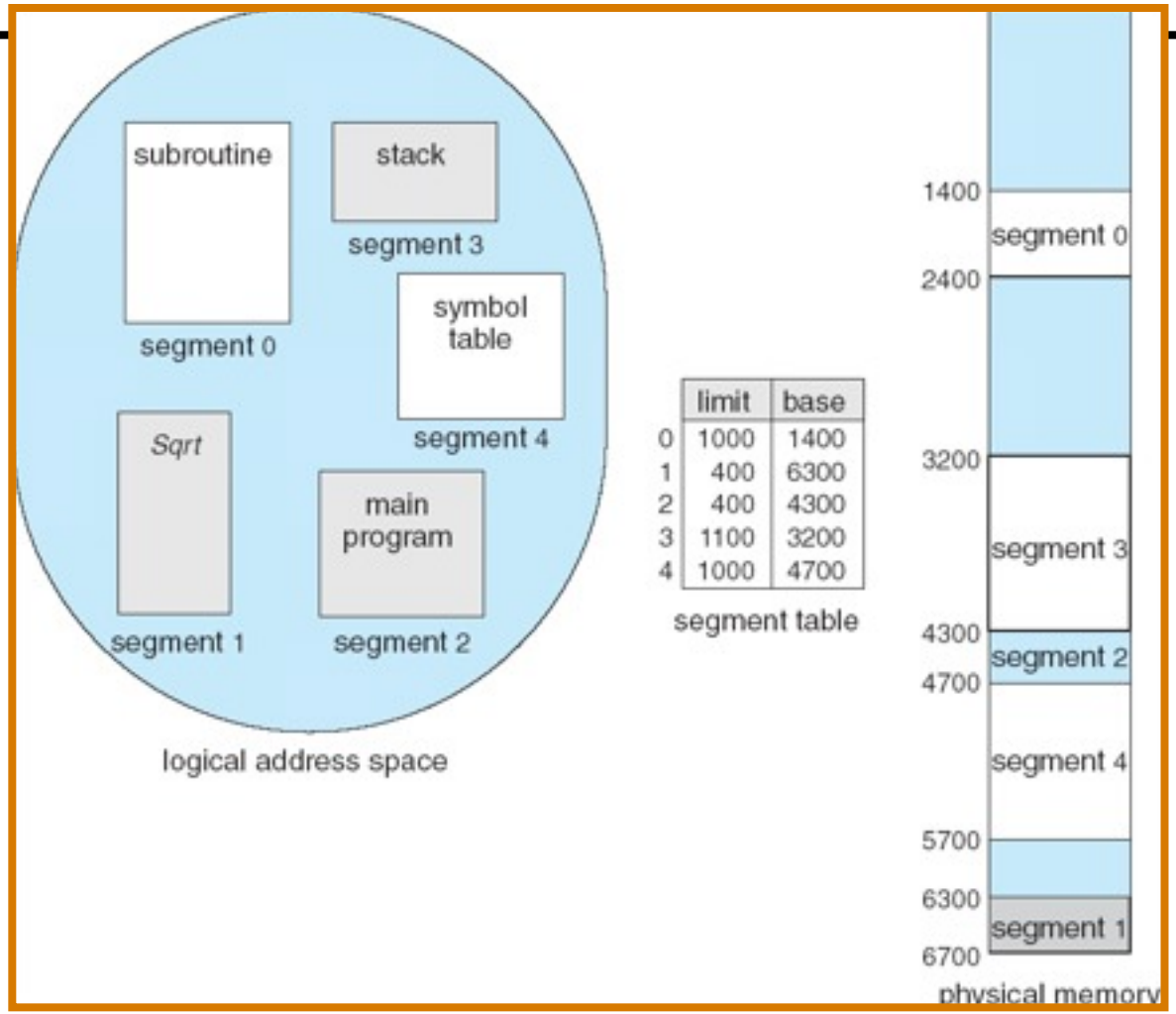
Segmentation Hardware



Use base and limit registers to perform address translation.

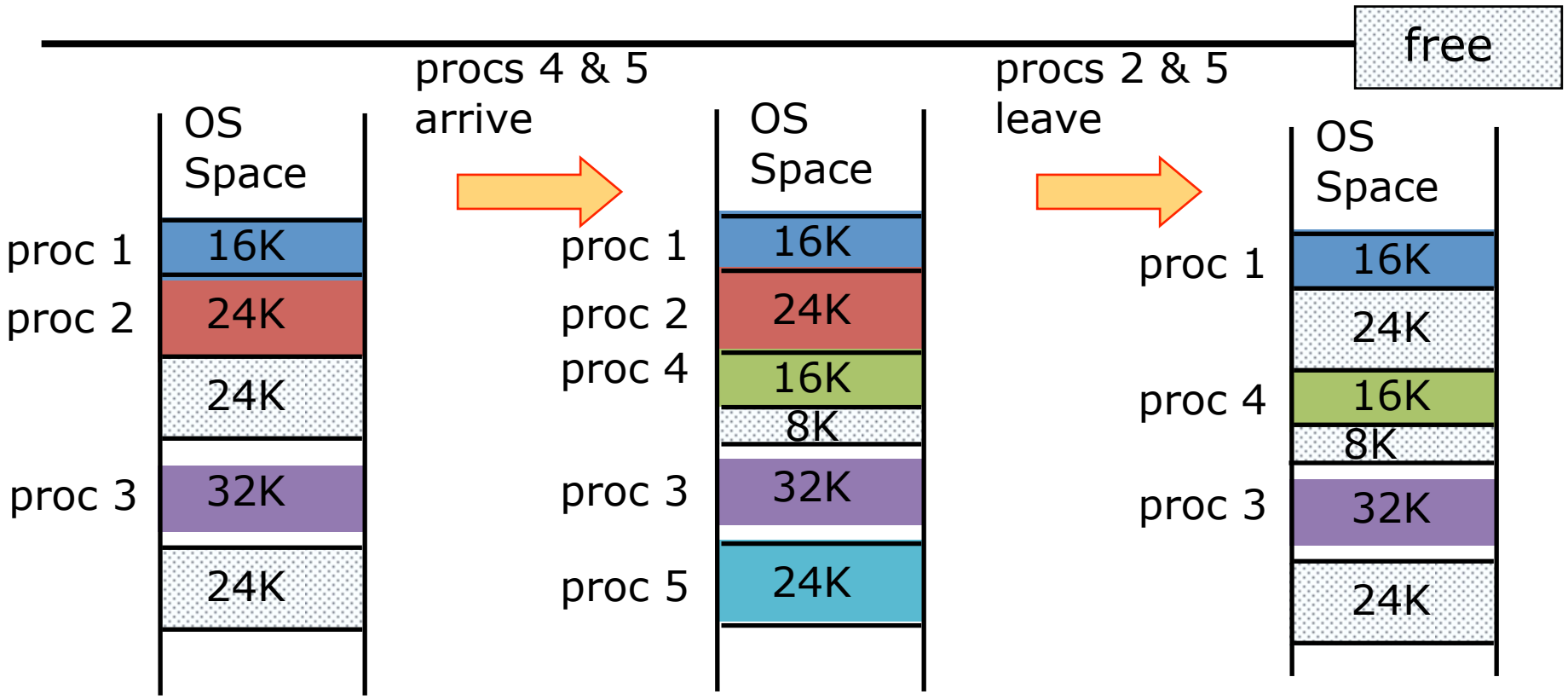
Trap to OS if bounds violation detected ("seg fault"/"core dump")

Segmentation Example



- Animation: http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/segmentation.htm

Processes Sharing Physical



As processes come and go, the storage is "fragmented". Therefore, at some stage processes have to be moved around to compact the storage.
Animation: http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/multiplepartcma.htm

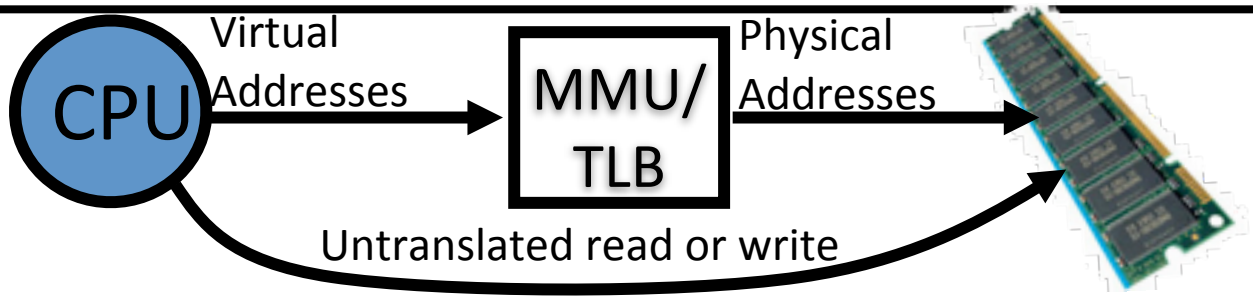
Agenda

- Virtual Memory Intro
- Page Tables
- Translation Lookaside Buffer
- Demand Paging
- System Calls
- Summary

Idea #2: Page Tables to avoid Fragmentation

- Divide memory address space into equal sized blocks, called pages
  - Page: a unit of memory translatable by memory management unit (MMU)
  - Traditionally 4 KB or 8 KB
- Use a level of indirection to map program addresses into physical memory addresses
  - One indirection mapping per address space page
- This table of mappings is called a Page Table
- Address Space (Process) switch: change

Two Views of Memory



- Address Space:
  - All the addresses and state a process can touch
  - Each process and kernel has different address space
- 2 views of memory:
  - CPU (what program sees, virtual memory)
  - Actual memory (physical memory)
  - Translation box converts between the two views

- Translation helps to implement
  - Portability: program can be linked separately
  - Protection: programs don't overlap with each other

Processes and Virtual Memory

- Allows multiple processes to simultaneously occupy memory
  - provides protection –
  - process cannot Rd/Wr memory from another
- Address space – give each program the illusion of own private memory
  - Suppose code starts at address 0x00400000. But different processes have different code, both residing at the same (virtual) address. So each program has a different view of memory.

Paging Terminology

- Program addresses called virtual addresses
  - Space of all virtual addresses called *virtual memory*
  - Divided into fixed-granularity (typically 4KB) pages indexed by Virtual Page Number (VPN).
- Memory addr. called phys. addresses
  - Space of all phys. addr. called *physical memory*
  - Divided into pages indexed by Physical Page Number (PPN).

Paged Memory Systems

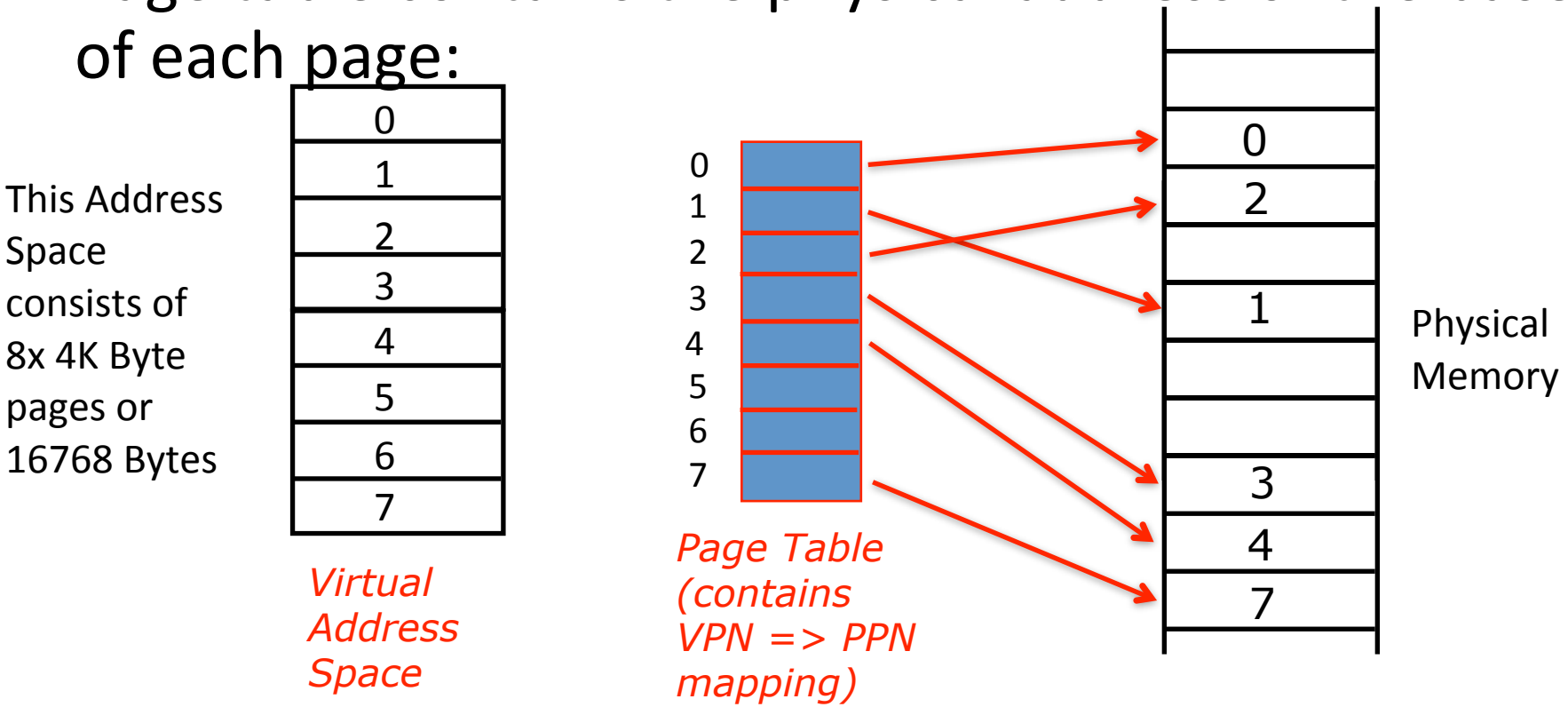
- Virtual memory address is split into:

32-bit memory address
4KB pages

- Offset refers to which byte in the page.
- Page# refers to page in address space.
- PageTable is a “MAP” : stores VPN to PPN, a page in physical memory.

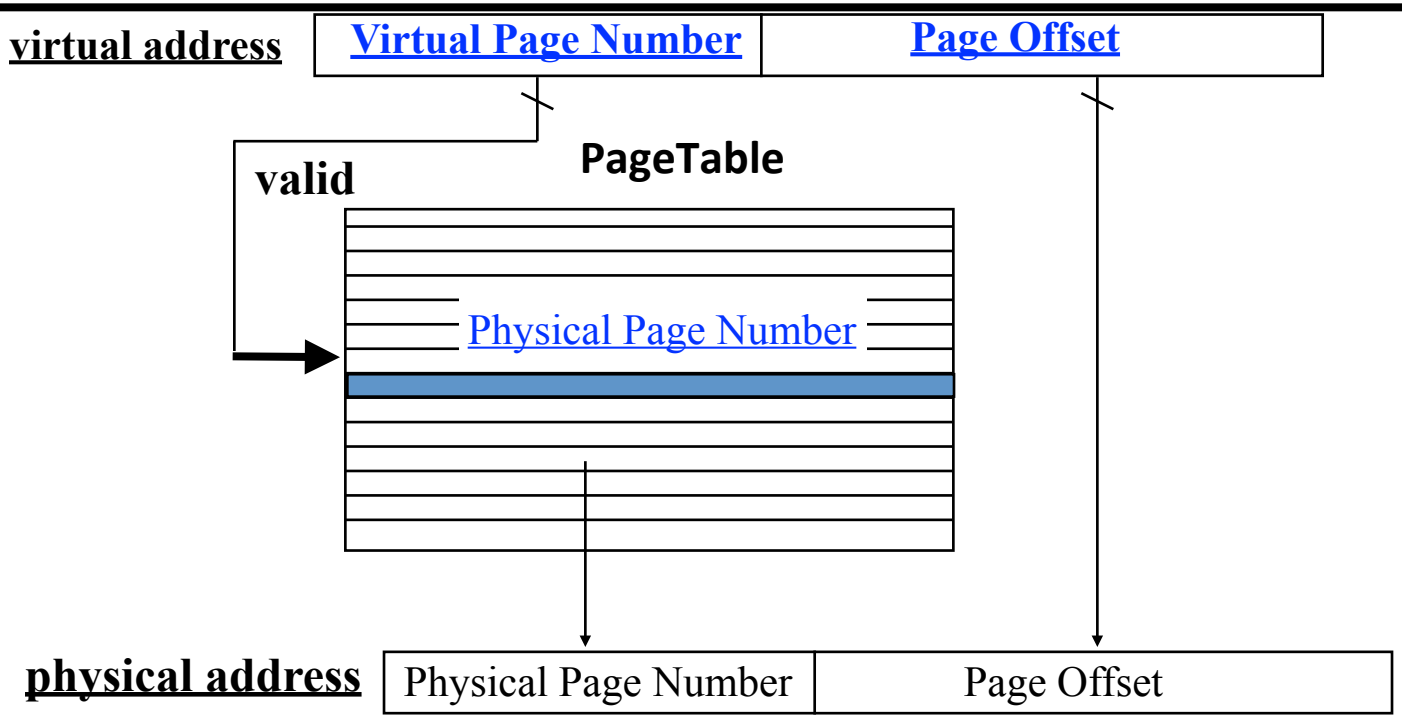
Paged Memory Systems

- Page table contains the physical address of the base of each page:



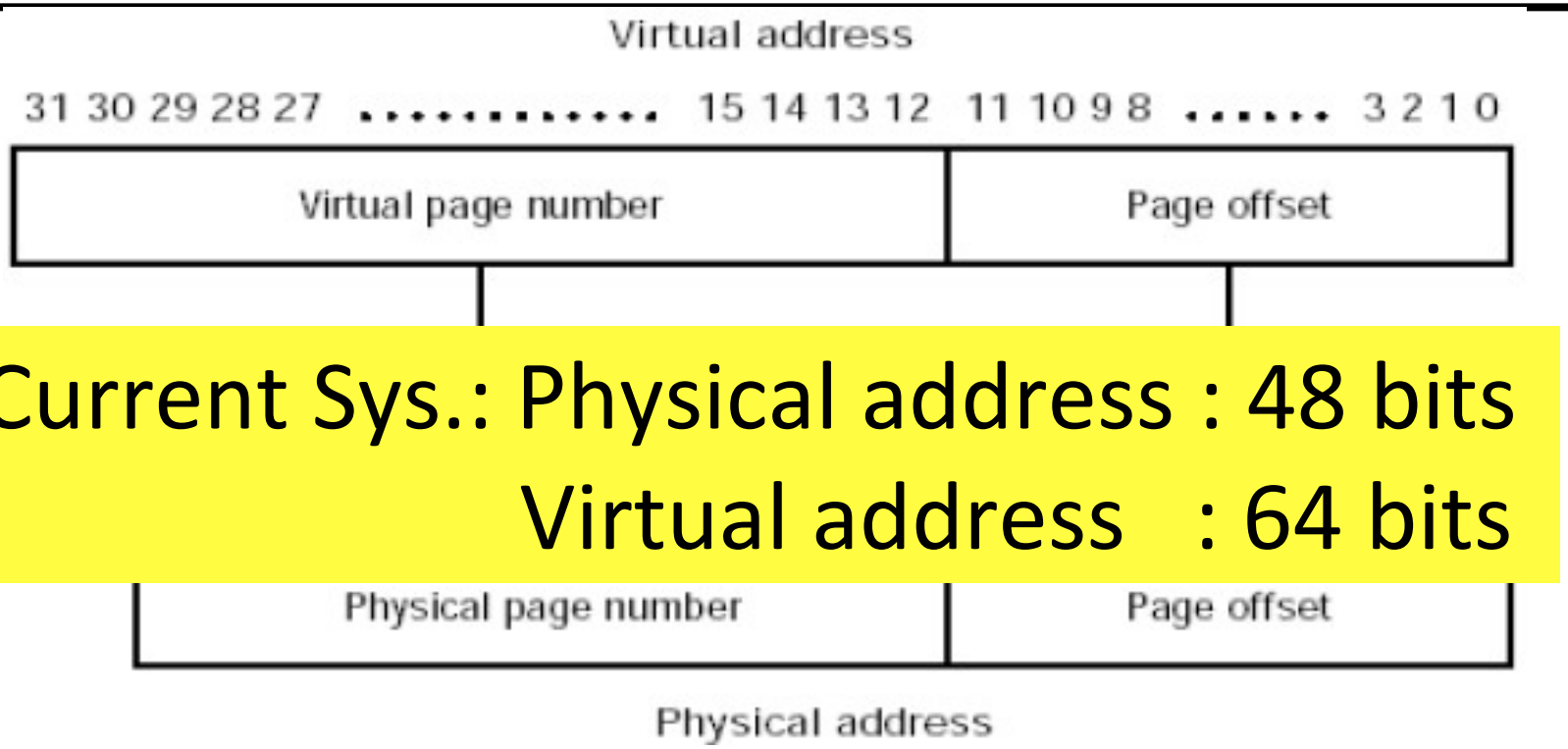
Page tables make it possible to store the pages of a program non-contiguously.

Address Translation via Page Table



- Generally, VPN has more bits than PPN, since phys. mem. is smaller (# virtual pages ≥ # physical page)
- Page offset determines page size, which is the same for both virt. and phys. memory

Addr Translation Example

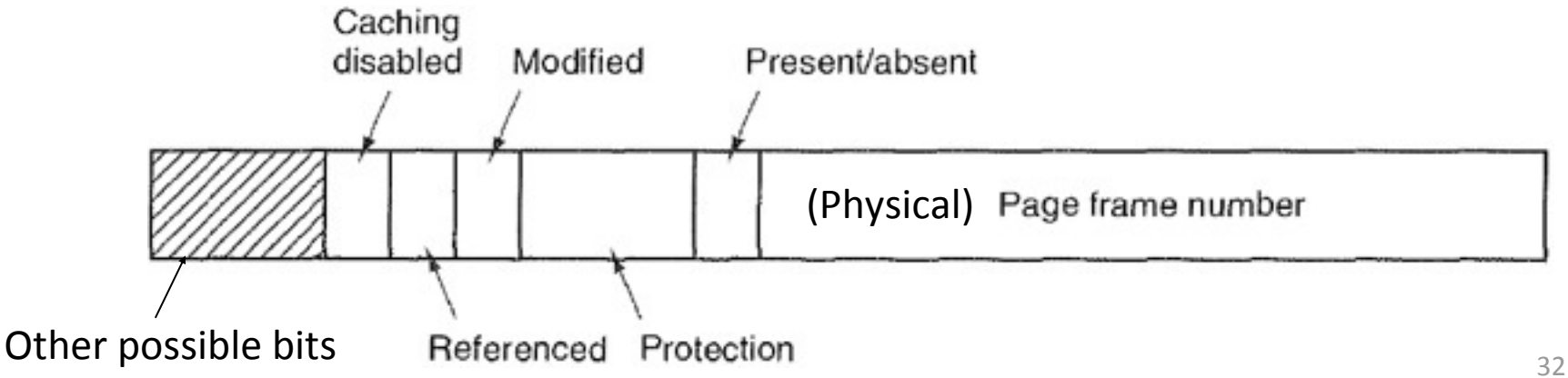


Current Sys.: Physical address : 48 bits
Virtual address : 64 bits

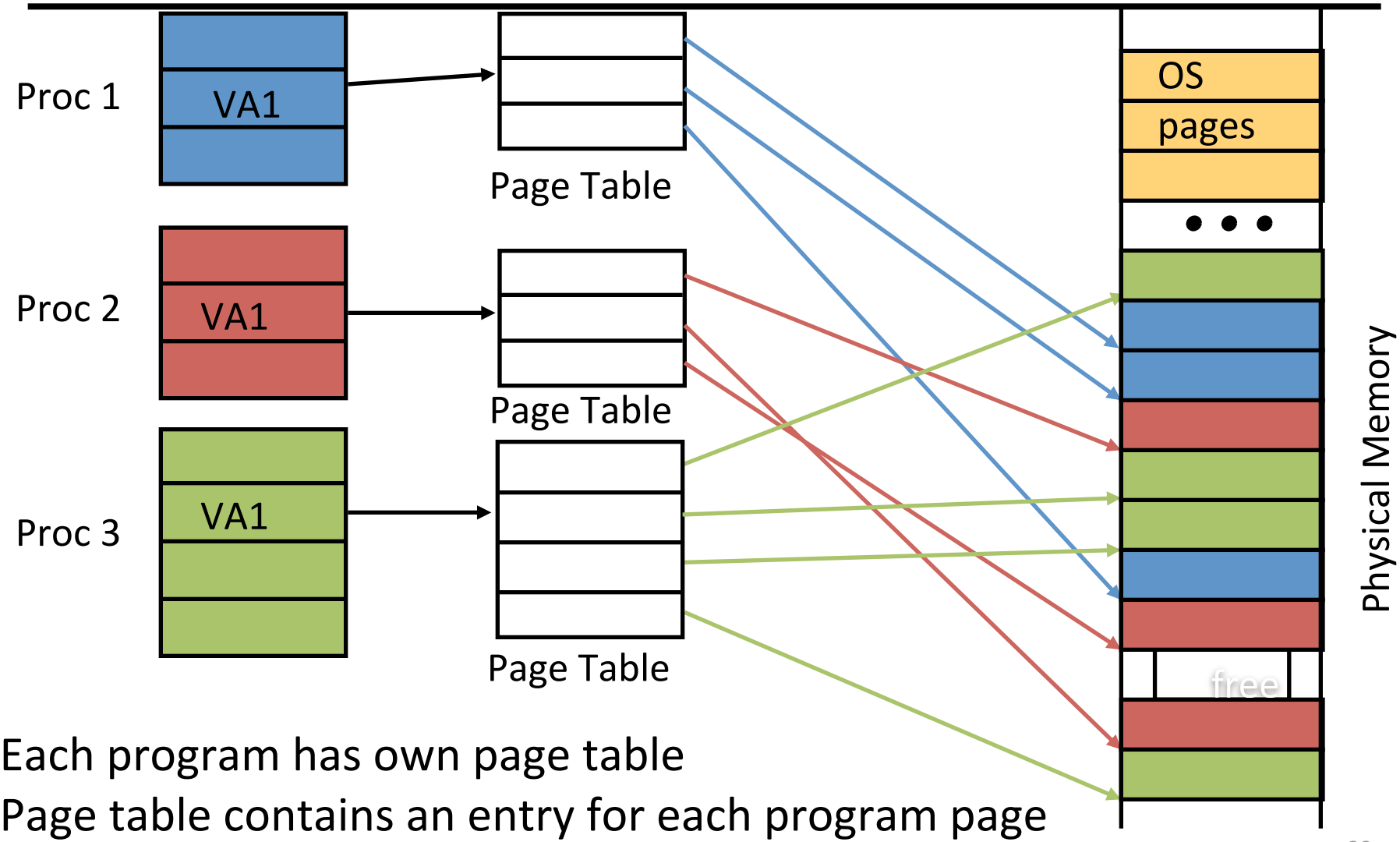
- Virtual Address: 32 bits
- Physical Address: 29 bits
- Page size: 2^12=4KB
- Note: Page offset here refers to byte address within a page (e.g., 4KB);

What is in a Page Table Entry (PTE)?

- Figure on Slide 24 is simplified. Several Additional bits in PTE
  - Physical Page Number (PPN), also called Page frame number
  - Present/absent bit, 1, in memory and 0 not in memory. Accessing a page table entry with this bit set to 0 causes a page fault to get page from disk.
  - Protection bits tell what kinds of access are permitted on the page. 3 bits, one bit each for enabling read, write, and execute.
  - Modified (M) bit, also called dirty bit, is set to 1 when a page is written to
  - Referenced (R) bit, is set whenever a page is referenced, either for reading or writing.
    - M and R bits are very useful to page replacement algorithms
  - Caching disabled bit, important for pages that map onto device registers rather than memory

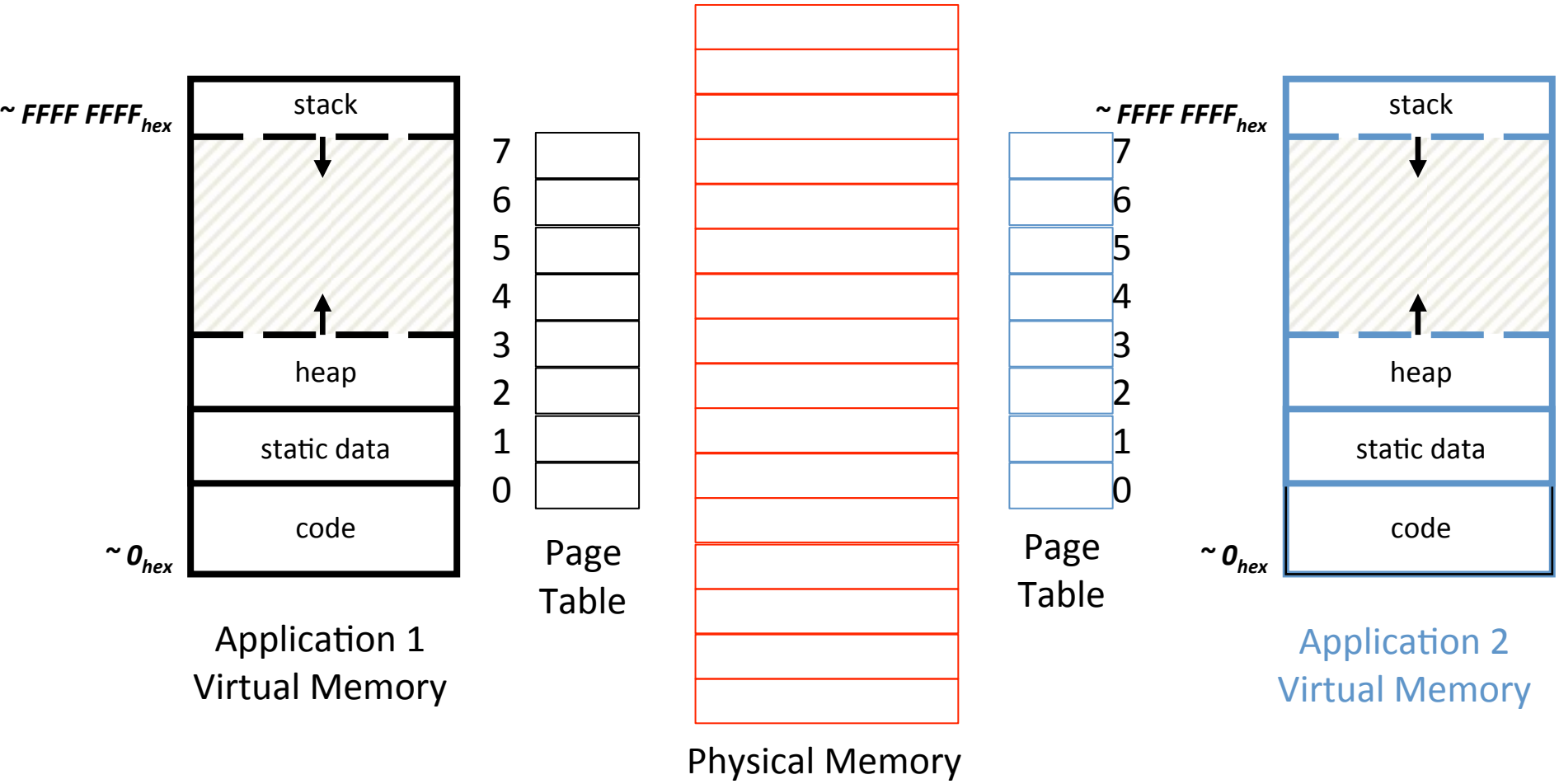


Separate Address Space per

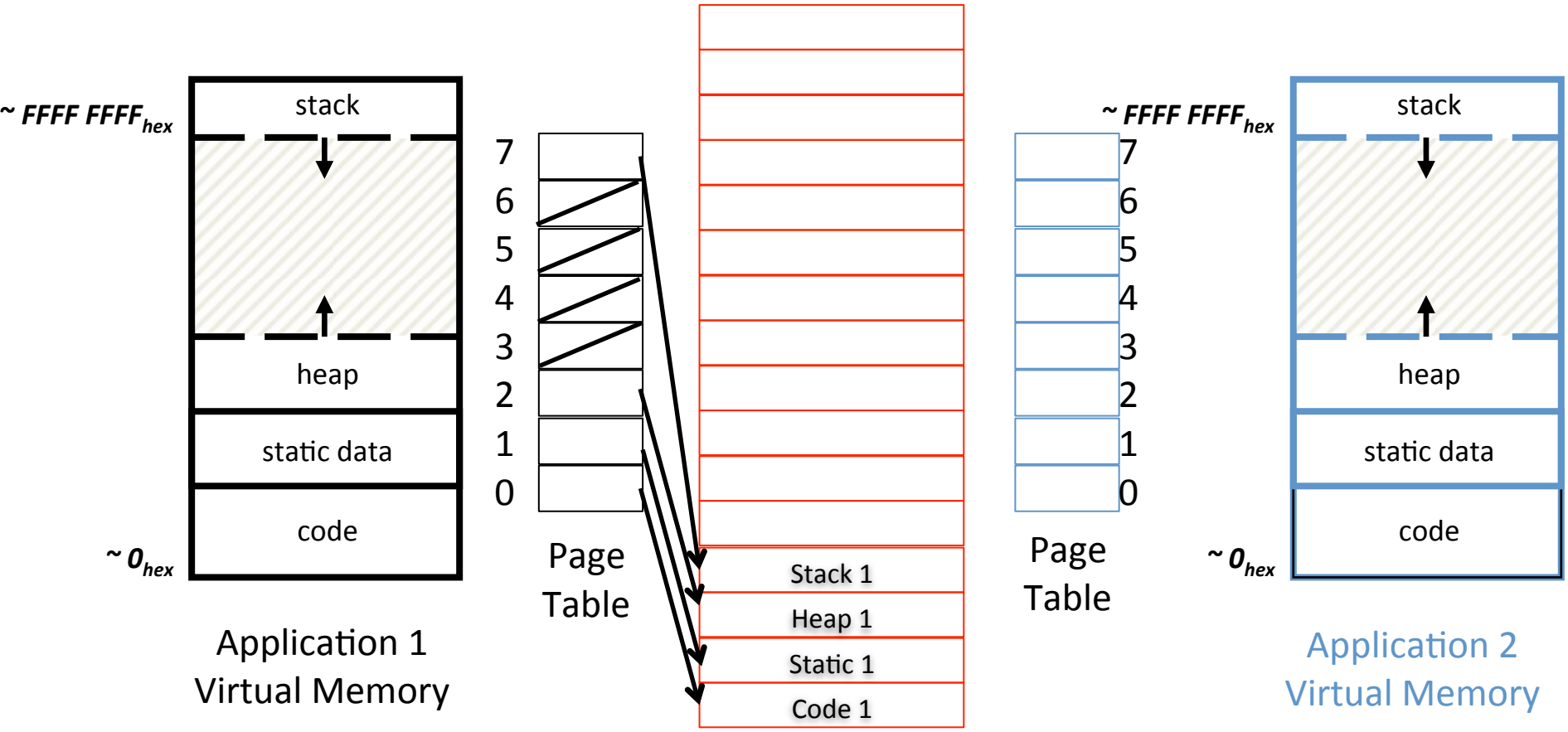


- Each program has own page table
- Page table contains an entry for each program page

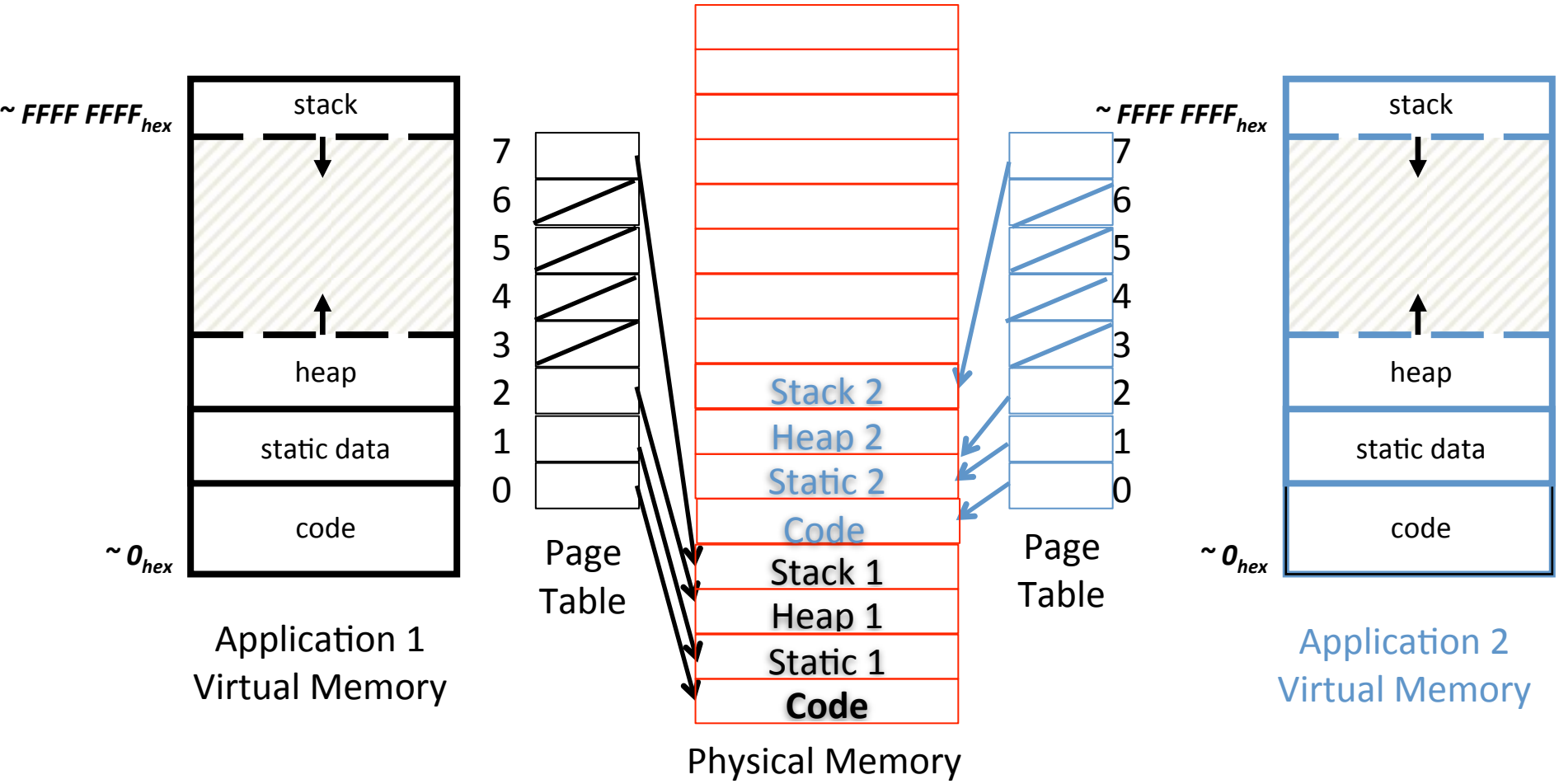
Protection + Indirection = Virtual Address Space



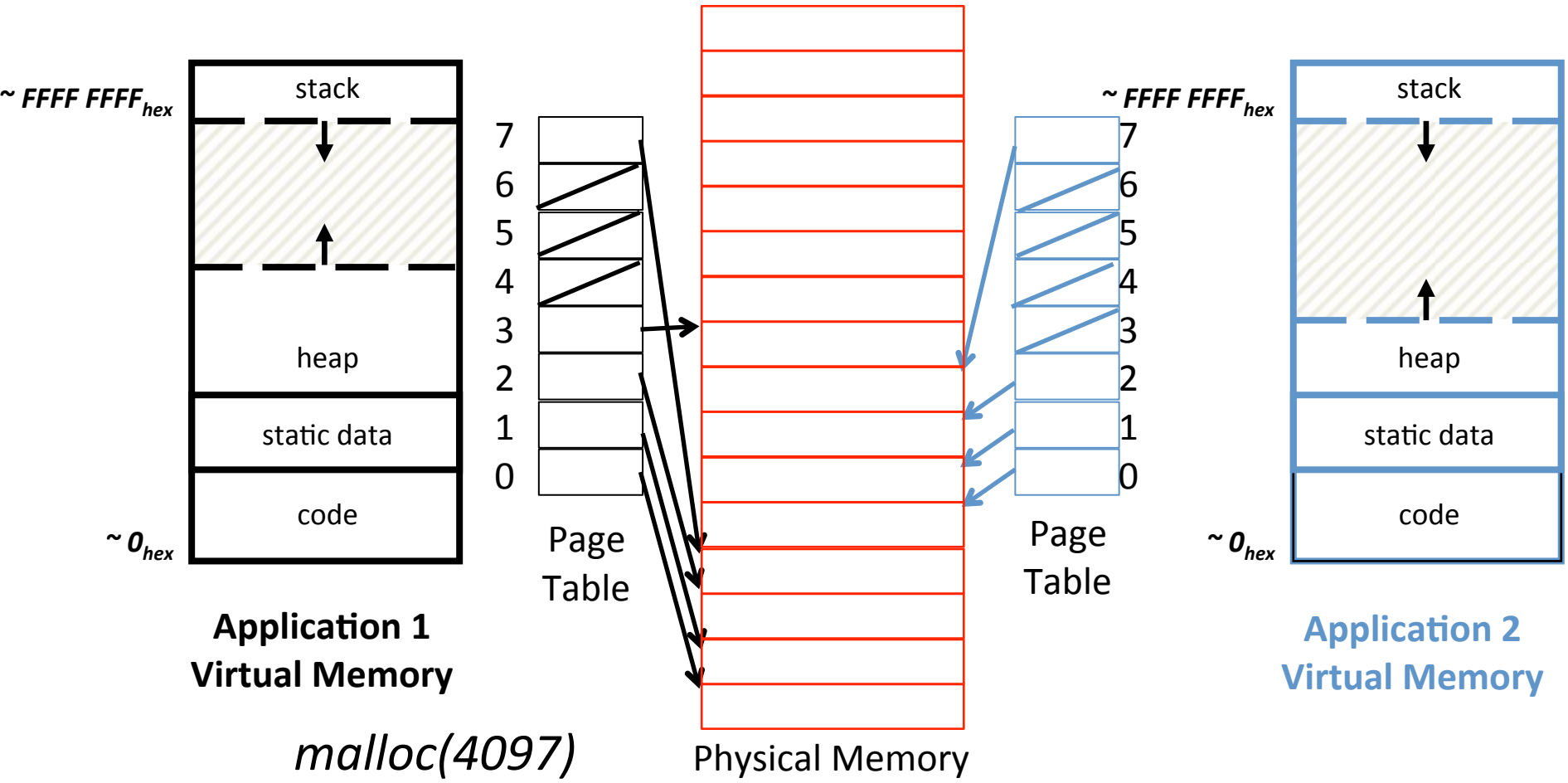
Protection + Indirection = Virtual Address Space



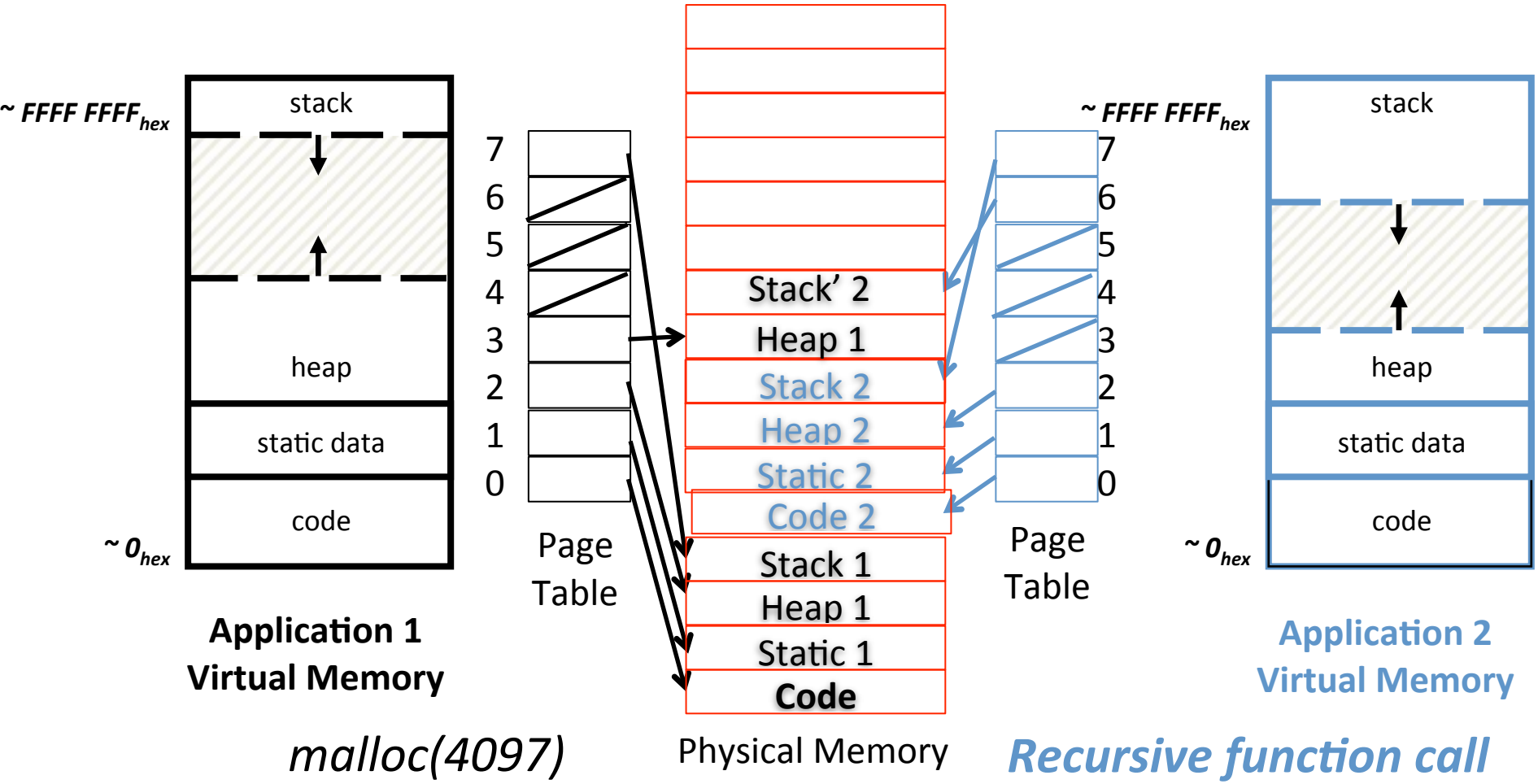
Protection + Indirection = Virtual Address Space



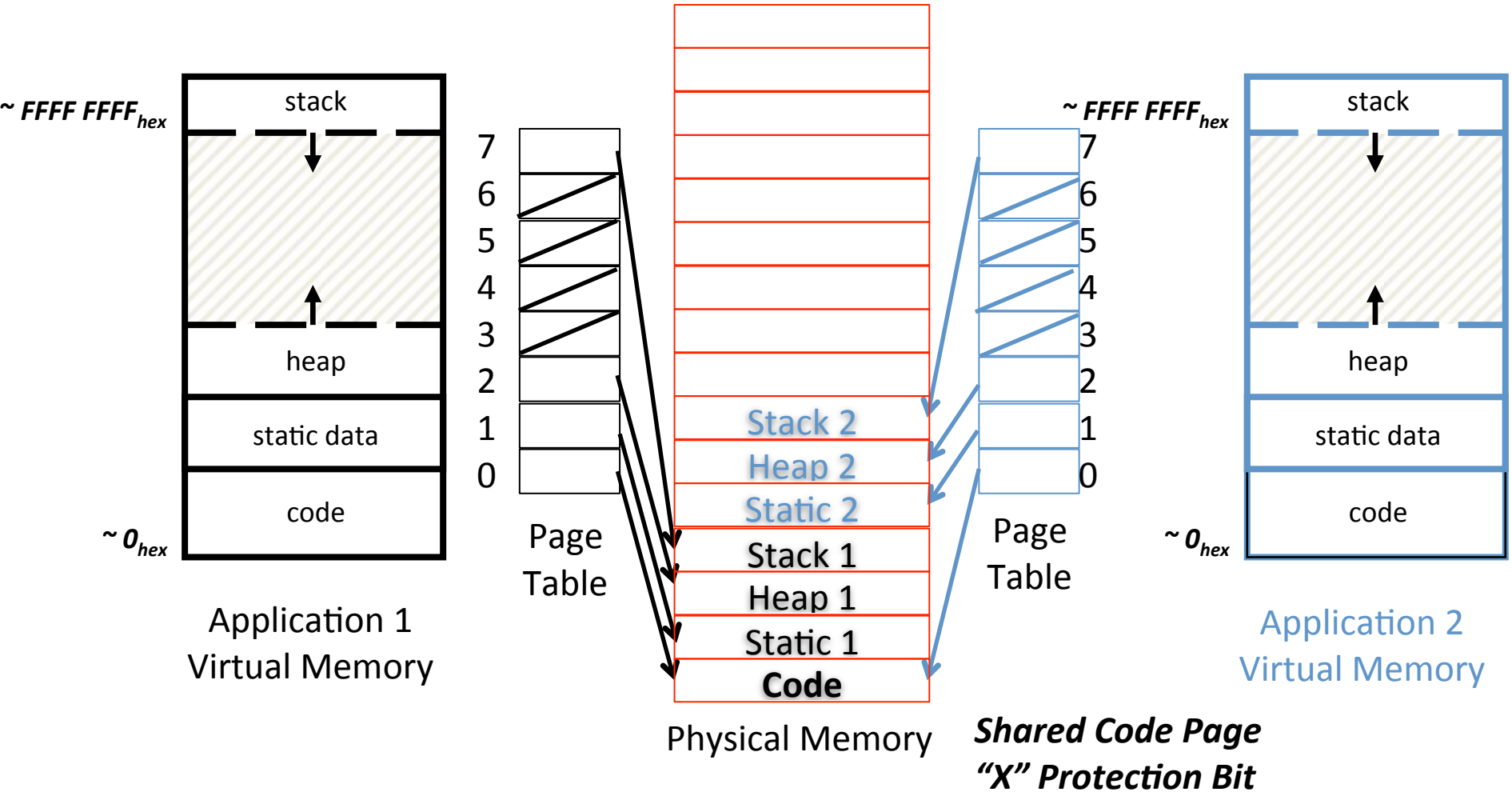
Dynamic Memory Allocation



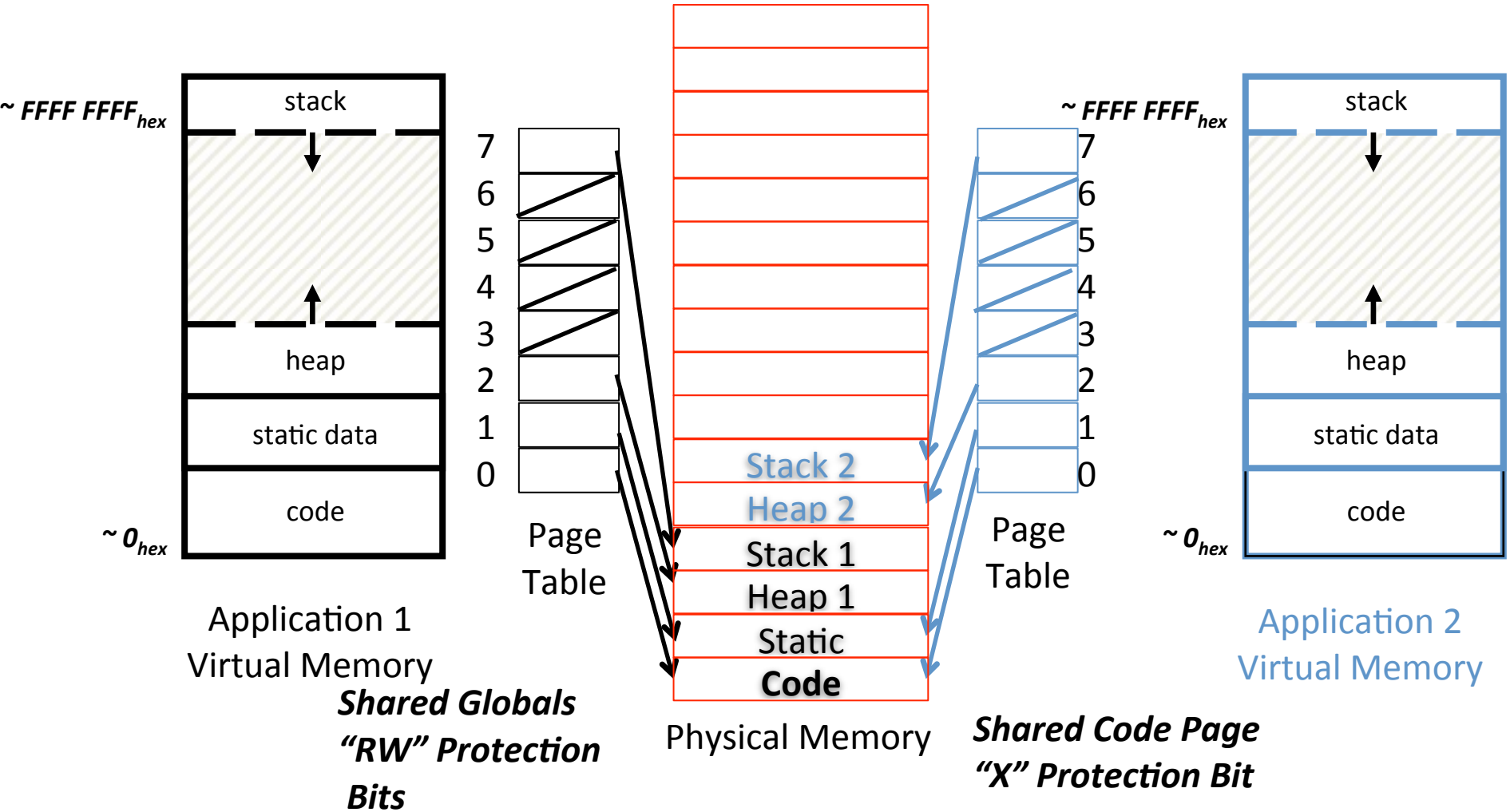
Dynamic Memory Allocation



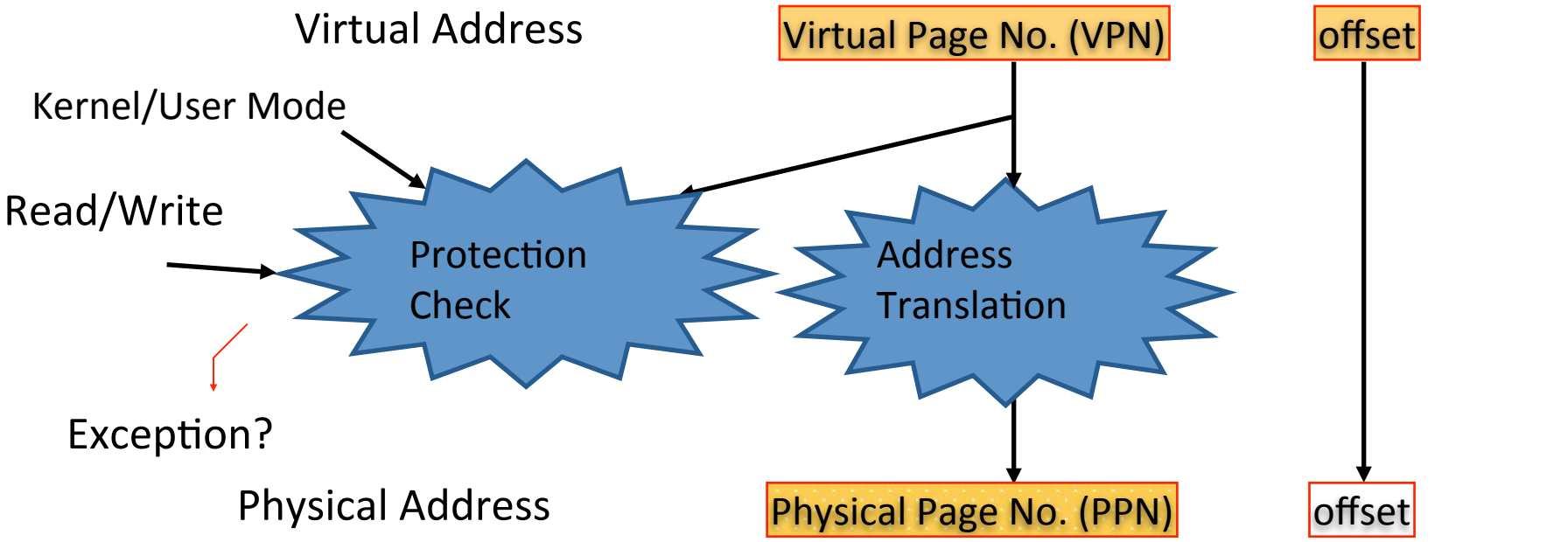
Controlled Sharing



Controlled Sharing



Address Translation & Protection



Every instruction and data access needs address translation and protection checks
Animation: http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/paginghardware.htm
http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/pagingmodel.htm
http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/pagingexample.htm

Where Should Page Tables Reside?

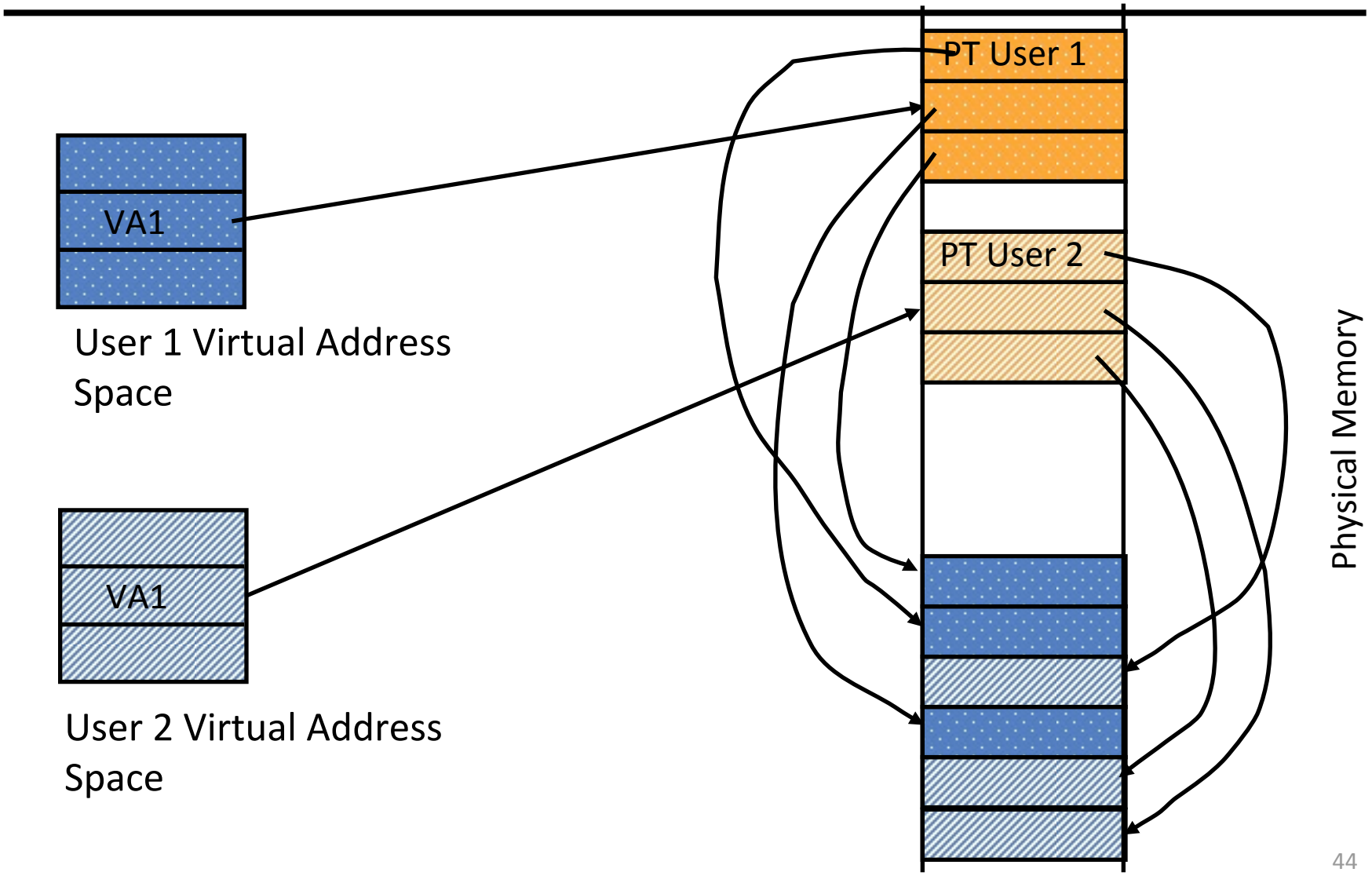
- Space required prop to the address space, …
  - Space requirement is large:
    e.g., 2^{32} byte virtual address space, 4K(2^{12})byte page size
    = 2^{20} PTEs (per process)
  - If each PTE is 4 bytes, then total bytes 4 * 2^{20} = 4MB

- Each process has its own page table. Suppose 50 processes running on a system, then 200MB of memory used for page tables!

Where Should Page Tables Reside?

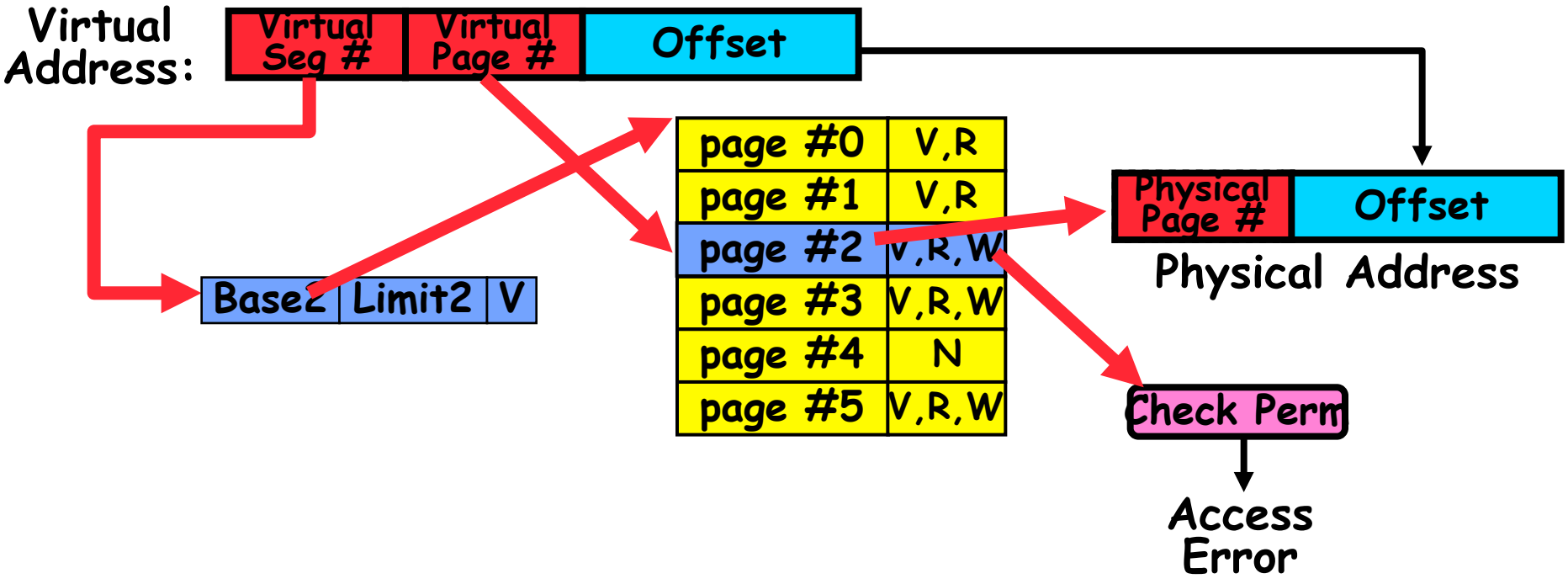
- Too large to keep in cache. Keep in main memory
  - Keep physical address of page table in P-Table Base Reg..
  - One access to retrieve the physical page address.
  - Second memory access to retrieve the data word
  - *Doubles* the number of memory references!
    - Use TLB to avoid the double memory access (later)
- What if Page Table doesn't fit in memory?
  - Multiple levels of page tables, (x86 (32bit) 3 levels x86 (64 bit) 4 levels or
  - segmentation + paging (Discussed later)

Page Tables In Physical Memory

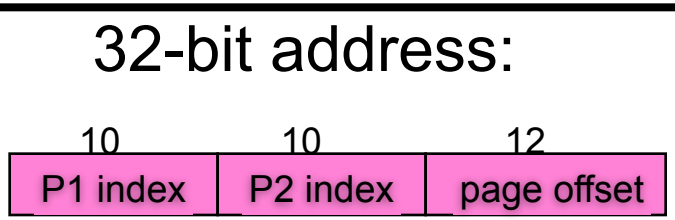


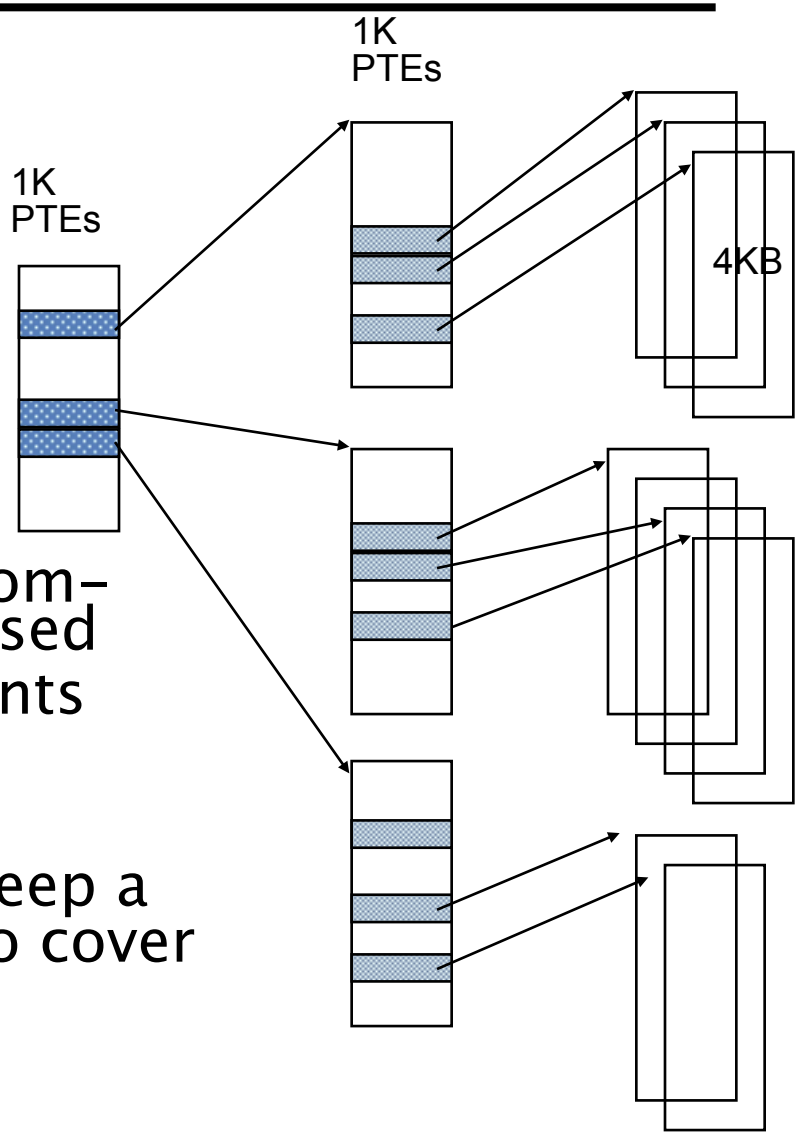
Segmentation + Paging (Multi-level Translation)

- What about a tree of tables?
  - Lowest level page table⇒memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):



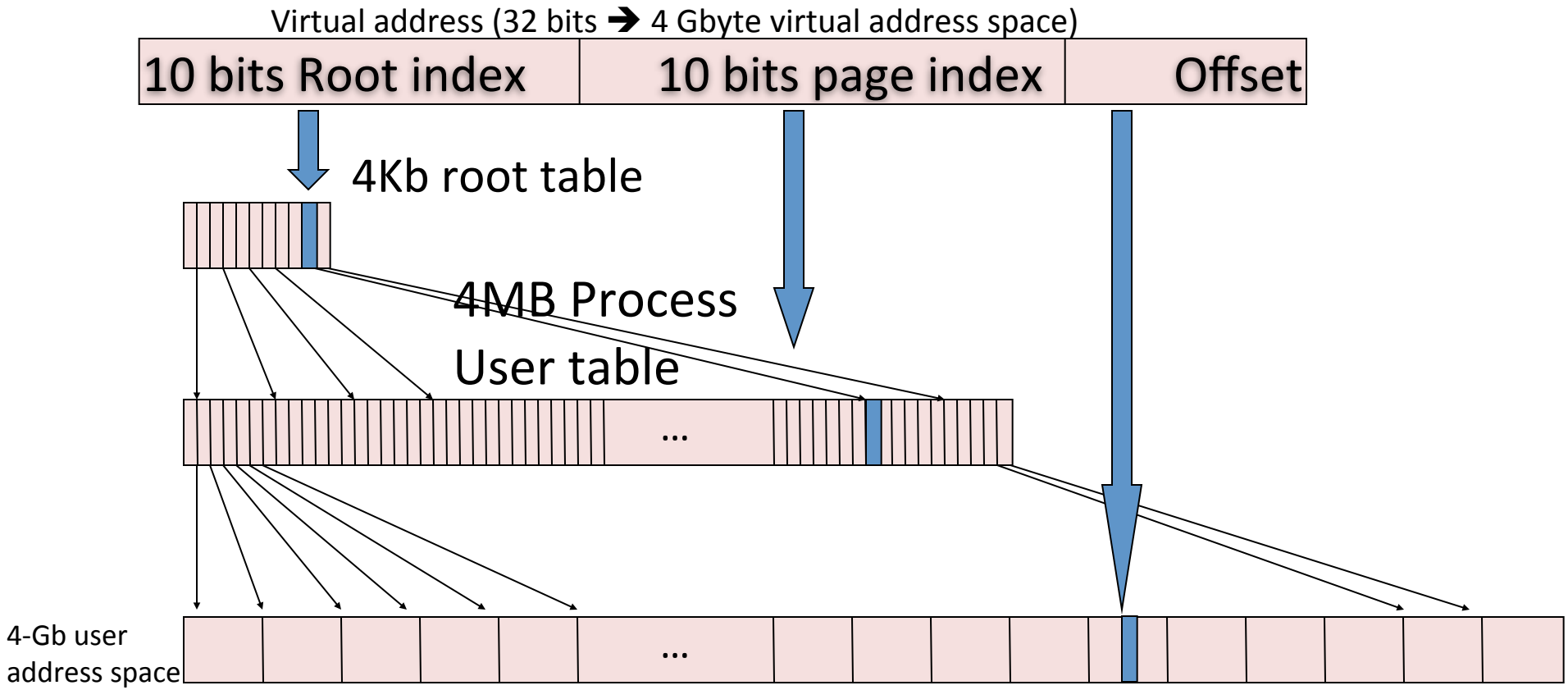
Two-level Page Tables



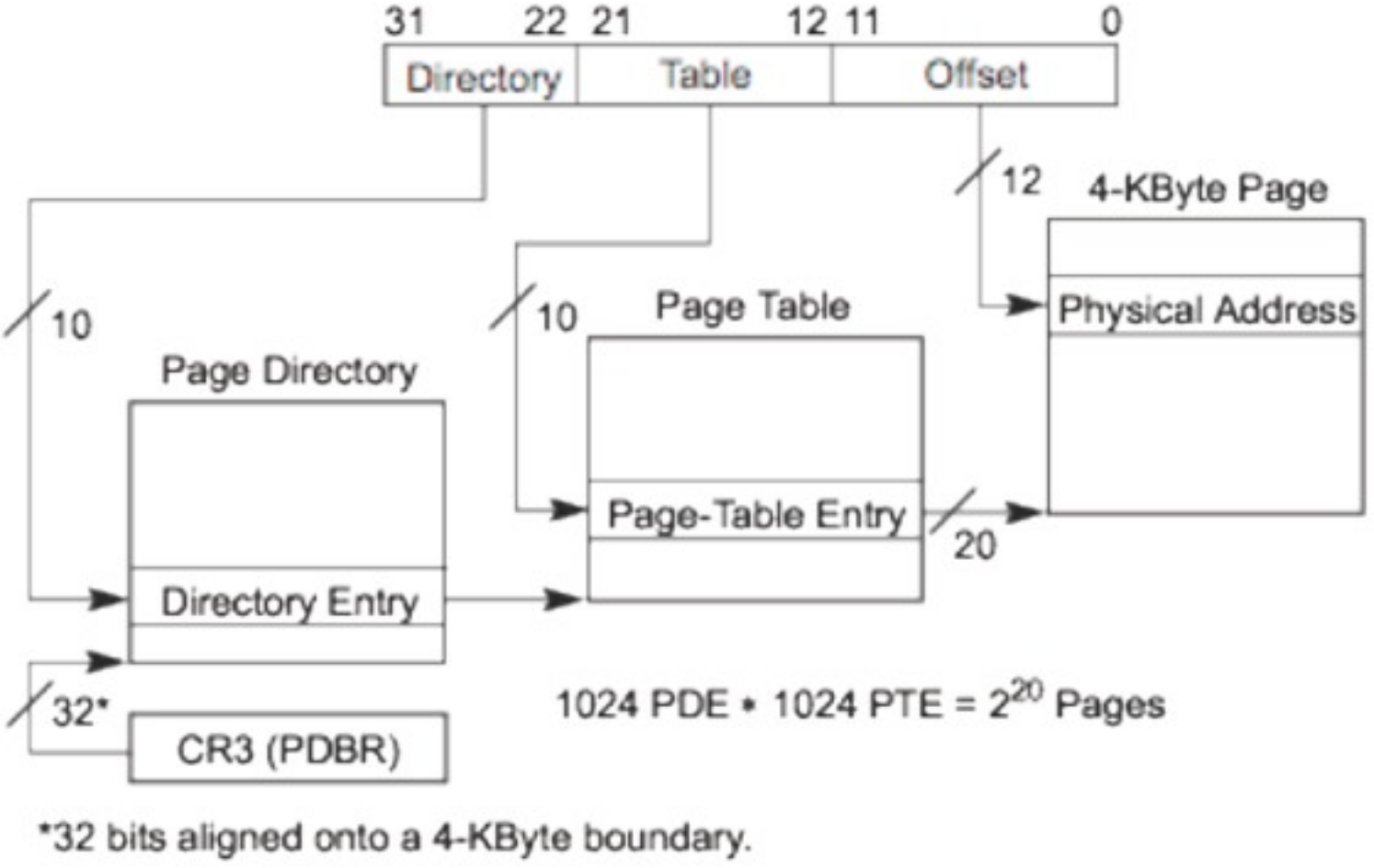


- Often the top-most parts and bottom-most parts of virtual memory are used
  - bottom for text and data segments
  - top for stack,
  - free memory in between.
  - The multilevel page table may keep a few of the smaller page tables to cover the required pages.

Two-level Page Tables Example



Two-level Page Tables Example



- PDBR: Page Directory Base Register
- PDE: Page Directory Entry

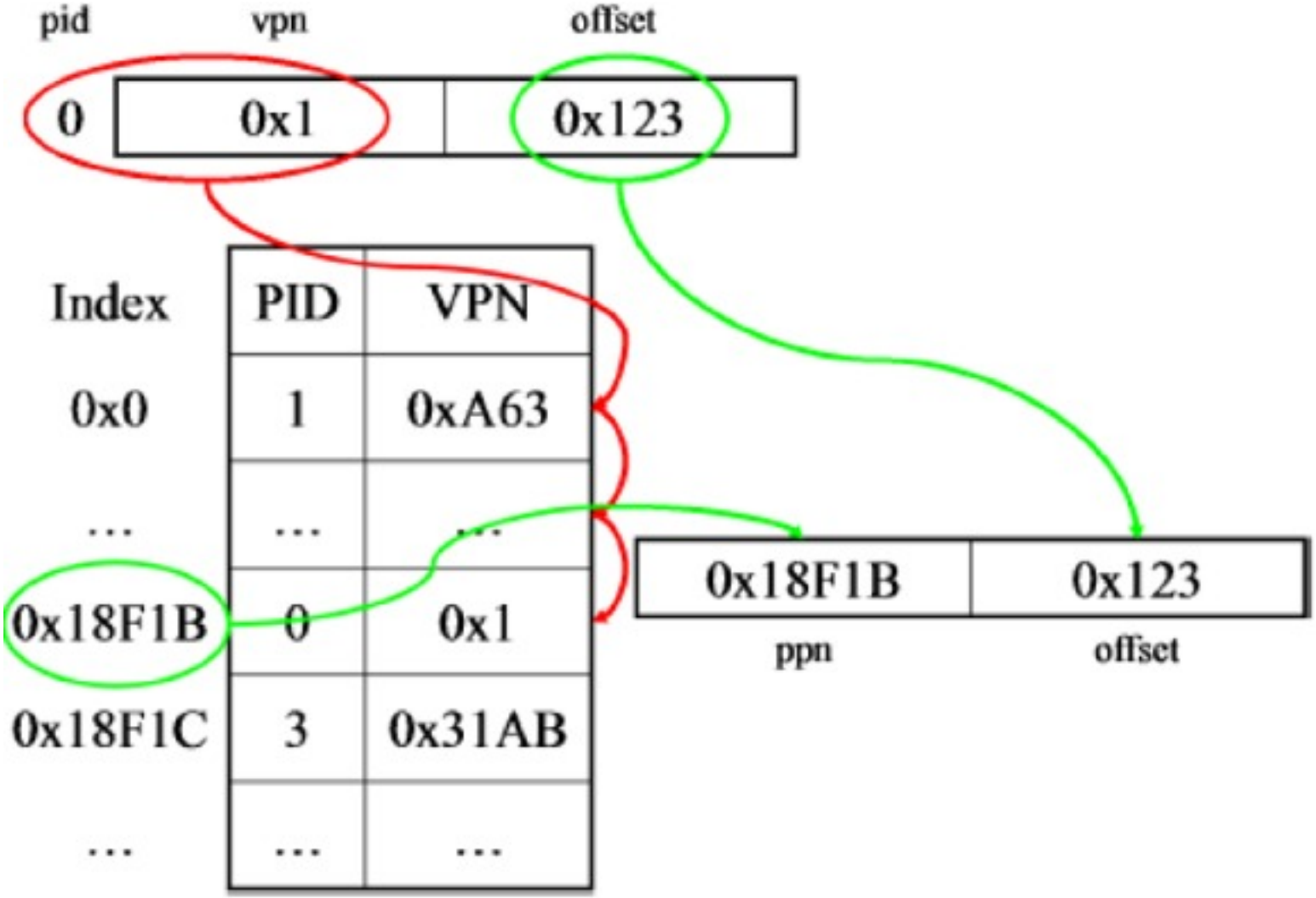
Inverted Page Tables

- As the size of virtual memory address space grows, additional levels must be added to multilevel page tables to avoid that the root page table becomes too large

- Assuming 64-bits address space, 4-Kb page size (12 bits for page offset), each page table can store 1024 entries, or 10 bits of address space. Thus $\lceil (64-12)/10 \rceil = 6$ levels are required → 6 memory accesses for each address translation

- **Inverted page tables:**
  - Indexed by PPN instead of VPN → # entries is equal to # PPNs, which is generally much smaller than #VPNs

Inverted Page Tables Lookup Example



Inverted Page Tables

- Consider a simple inverted page table
  - One entry per Physical Page.
  - The table is now shared by the processes, so each PTE must contain the pair <process ID, virtual page #>
  - Translation : Virtual Page # and Process ID # are compared against each entry,
  - If a match is found, its **index** in the inverted page table is the PPN
  -

Inverted Page Tables

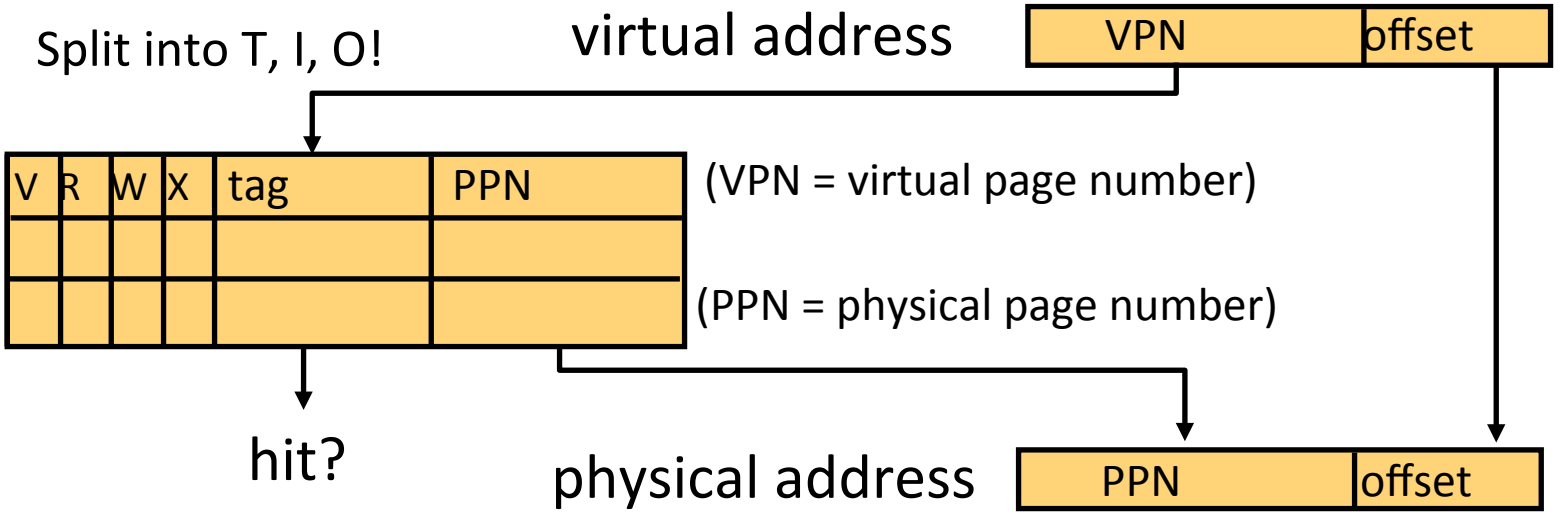
- The search can be very inefficient.
- Solution: Hashed IPT to reduce # memory accesses (Worst case: all entries)
- Q: Is PPN stored in the table?
- A: No, since the table index is PPN. Remember that we are doing a reverse lookup.

Agenda

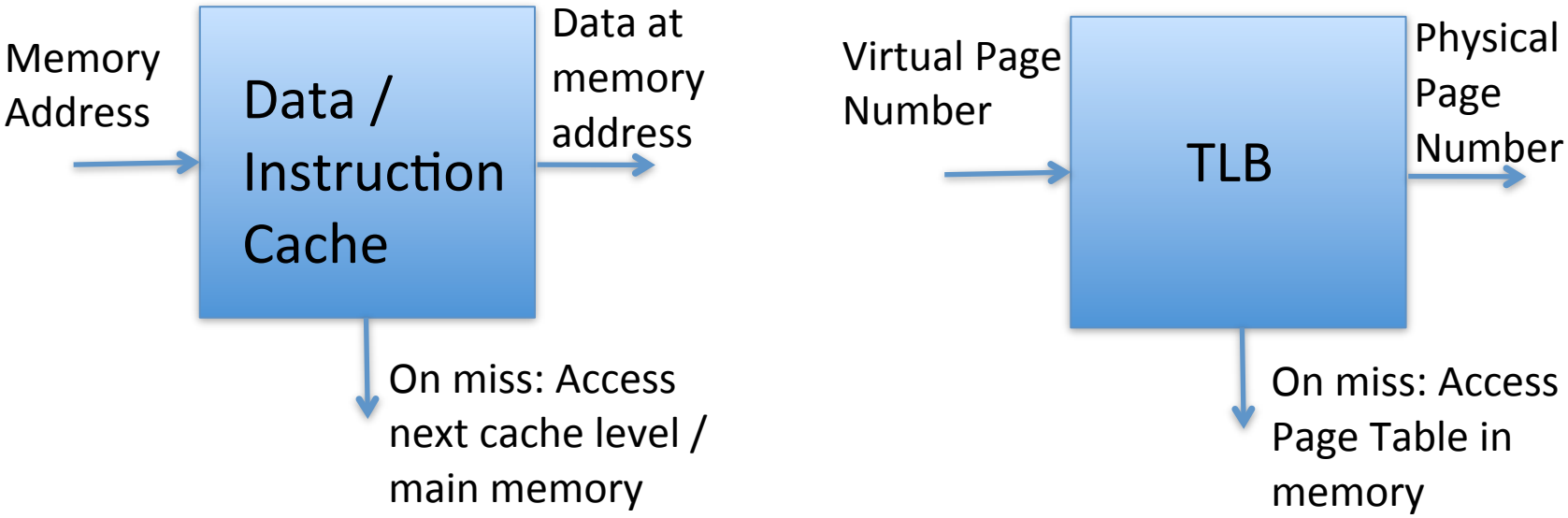
- Virtual Memory Intro
- Page Tables
- Translation Lookaside Buffer
- Demand Paging
- System Calls
- Summary

Translation Lookaside Buffer

- PageTable entry Cache (128-256 entries).
- Note that is caches the final translation (no intermediate points on the tree)
  - Looks up Virtual Address; returns Physical Address



TLB is Cache


Memory Address
Data / Instruction Cache
Data at memory address
On miss: Access next cache level / main memory
Virtual Page Number
TLB
Physical Page Number
On miss: Access Page Table in memory

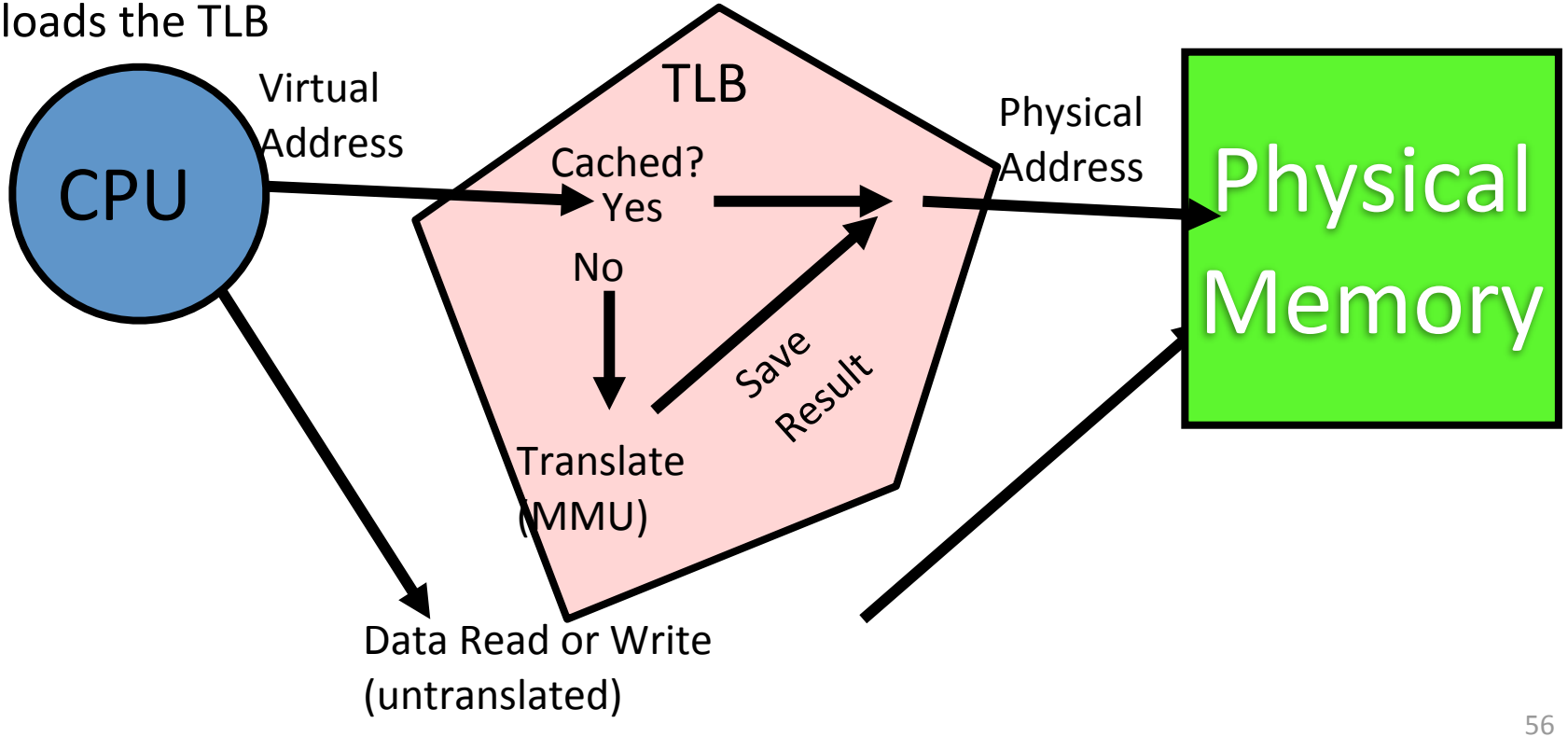
TLB: More Details

Cache Page Table Entries in TLB

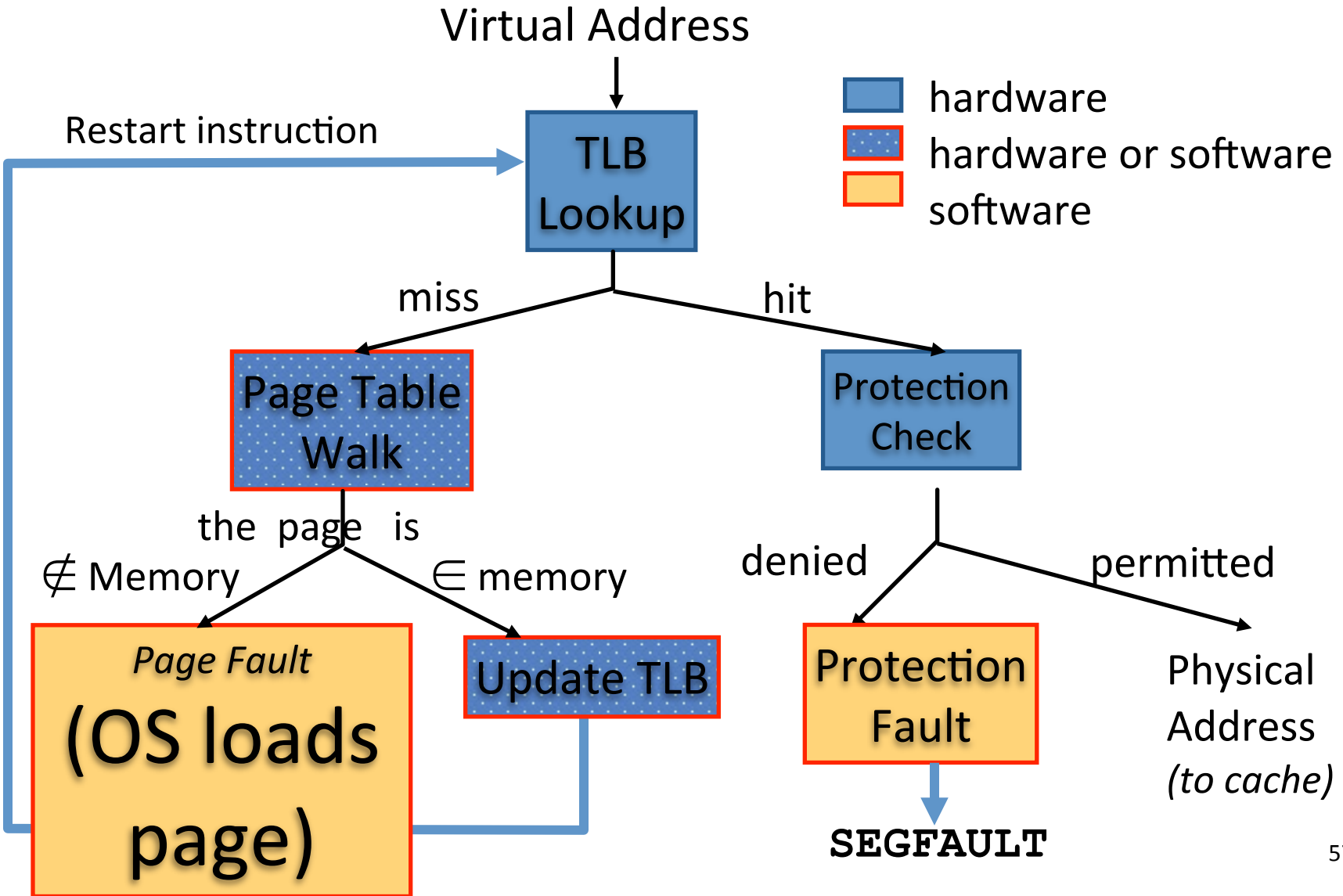
TLB hit => *Single Cycle Translation*

TLB miss => *Access Page-Table to fill*

A *memory management unit* (MMU) is hardware that walks the page tables and reloads the TLB



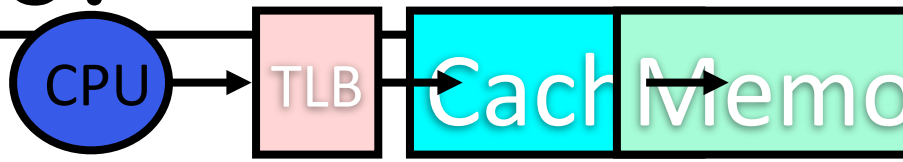
TLB Lookup Sequence



What Happens on a Context Switch?

- Recall each process has its own page table and virtual address space; But there is only a single TLB in the system
  - TLB entries no longer valid upon process context-switch
- Options:
  - Invalidate TLB: set valid bits of all TLB entries to 0
    - Simple but might be expensive
    - What if switching frequently between processes?
  - Include ProcessID in TLB
    - This is an architectural solution: needs hardware support

What TLB organization makes sense?



- Needs to be really fast
  - Critical path of memory access
    - Thus, this adds to access time (reducing cache speed)

- However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely high!
  - This argues that cost of Conflict (Miss Time) is much higher than slightly increased cost of access (Hit Time)

TLB organization

- How big does TLB actually have to be?
  - Usually small: 128-512 entries
  - Not very big, can support higher associativity without much performance degradation
- TLB is usually fully-associative, but can also be set-associative
- Q: Is TLB write-through or write-back?
- A: write-through → always keep TLB and page table consistent

What Actually Happens on a TLB Miss?

- HW/SW looks at current page table to fill TLB (may walk multiple levels).
  - x86 (hardware), SPARC (hardware and software)
  - If PTE valid (page present in memory), refill
  - If PTE marked as invalid (page on disk), causes Page Fault, then kernel gets the page from disk
- Example: http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/pagingtlb.htm

Effective Access Time with TLB

- TLB lookup time = σ time unit
- Memory access time = m time unit
  - Assume: Page table needs single access (no multilevel page tables)
  - There is no cache
- TLB Hit ratio = η
- Effective access time:
  - $EAT = (m + \sigma)\,\eta + (2m + \sigma)(1 - \eta) = 2m + \sigma - m\,\eta$

Valid & Dirty Bits

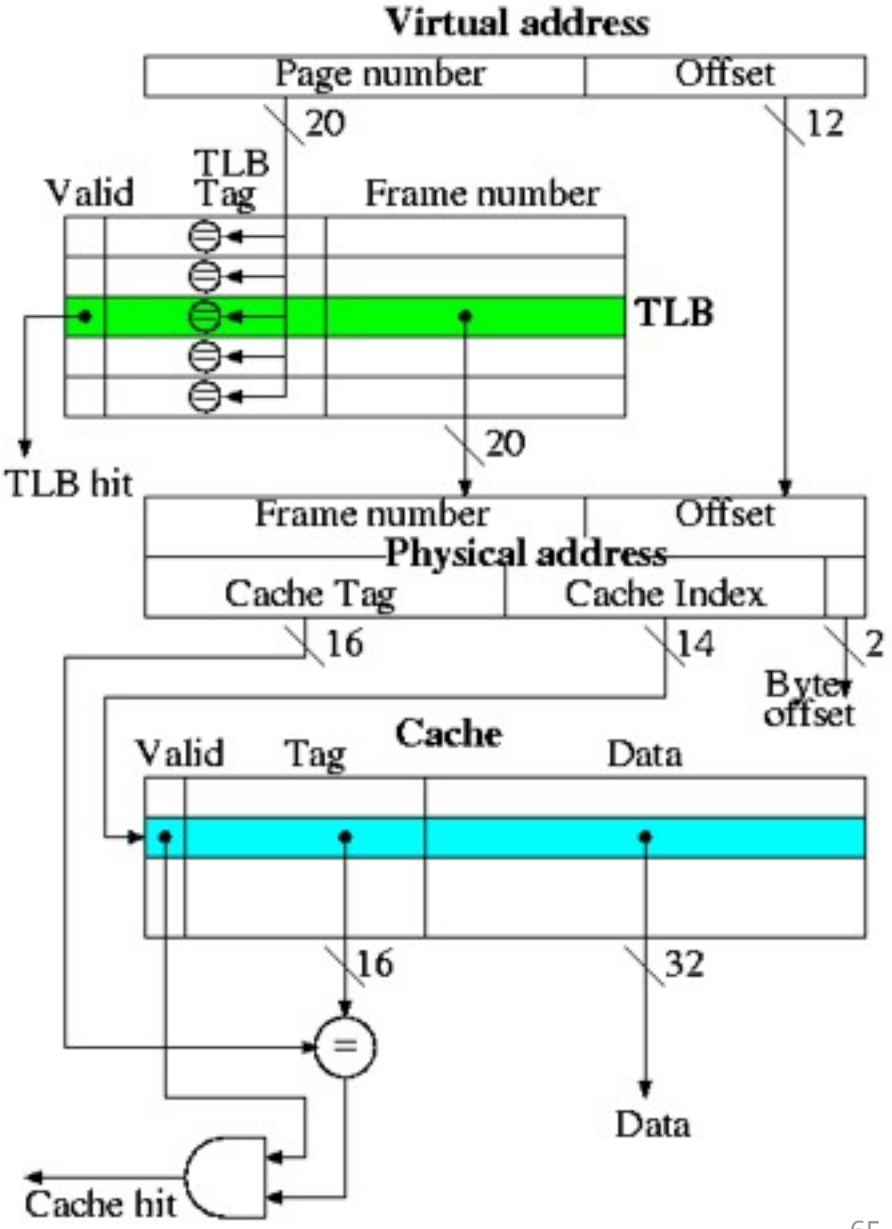
- TLB entries have valid bits and dirty bits.
  - valid bit: 0 means TLB miss
  - The dirty bit has different meanings. Page corresponding to this TLB entry has been changed.

Cache Addressing

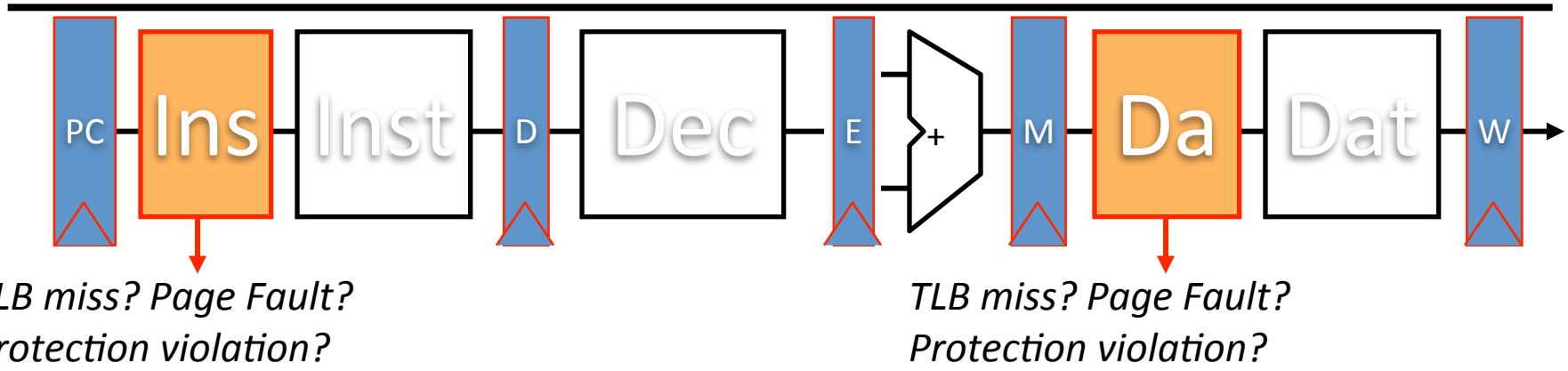
- Cache can be virtually addressed ...
  - the virtual address is broken into tag-index-offset to look up data in cache
  - Must either clear cache on context-switch or store Process ID with the Tag.
  - Address translation only needed upon cache miss
- ... or physically addressed
  - the virtual address is first converted to a physical address (using page table)
  - the physical address is used to find data in cache
- Virtually addressed caches are faster, but make sharing data between processes complicated.
  - Next examples assume physically-addressed cache

Example

- Consider the TLB +Physically-Addressed Cache:
  - Virtual address = 32 bits
  - Physical address = 32 bits
  - Fully associative TLB
  - Direct mapped cache
  - Cache blocksize = one word (4 bytes)
  - Pagesize = 4KB = 2^12 bytes
  - Cache size = 16K entries = 64KB

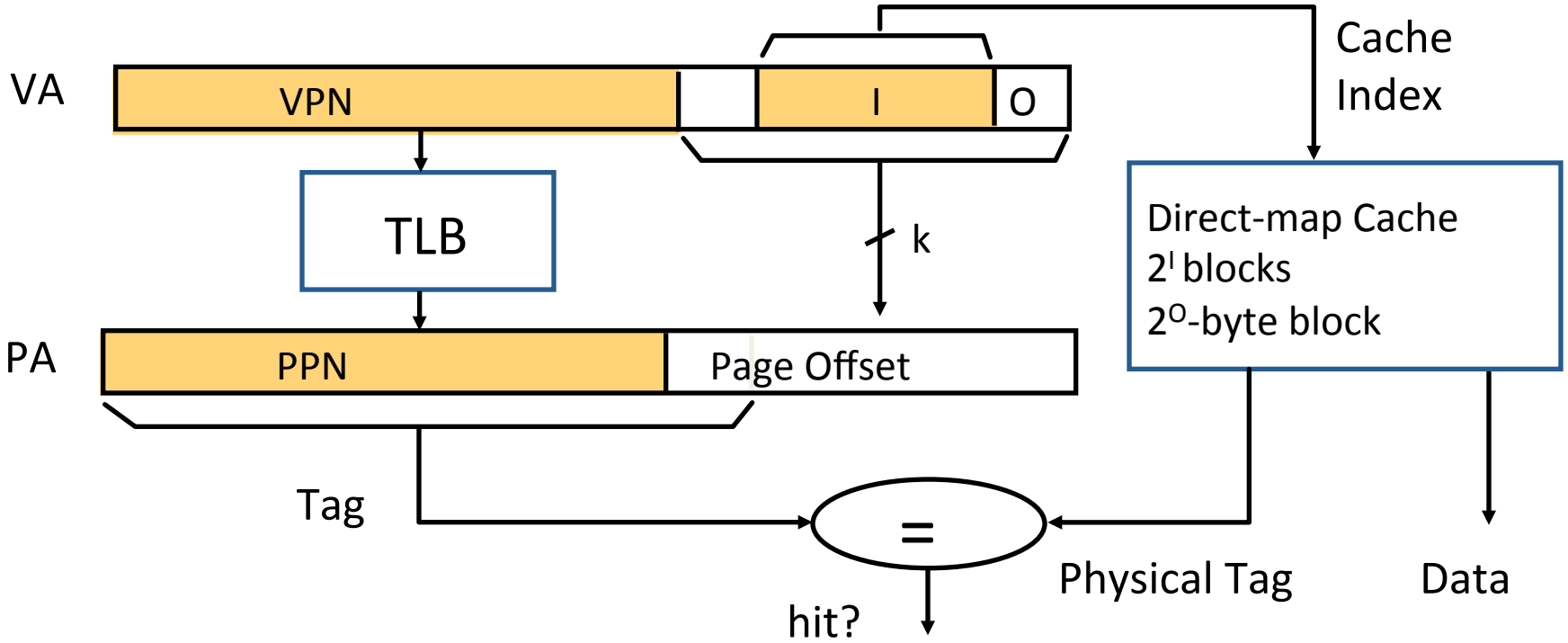


Address Translation in CPU



- Need mechanisms to cope with the additional latency of a TLB:
  - Slow down the clock
  - Pipeline the TLB and cache access (make it span multiple page stages)
  - Virtually-addressed cache

Parallel TLB &Cache Access



Index I is available without consulting the TLB

⇒ cache access and TLB access can begin simultaneously

Tag comparison is made after both accesses are completed to determine cache hit or miss

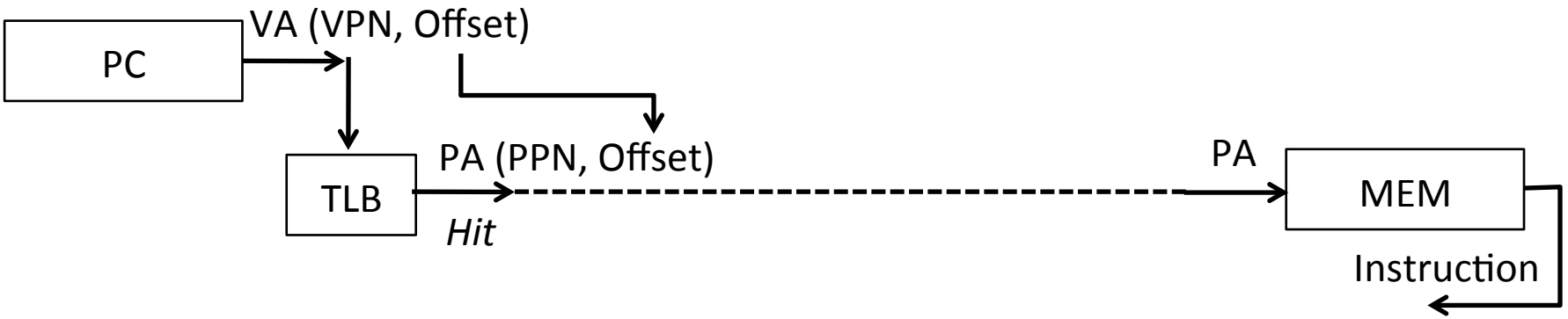
Only works if I + O ≤ k (Page Offset has same or more number of bits as Cache Index+Offset)

e.g., won't work for example on Slide 58 with cache T:I:O=16:14:2

Day in the Life of an (Instruction) Address

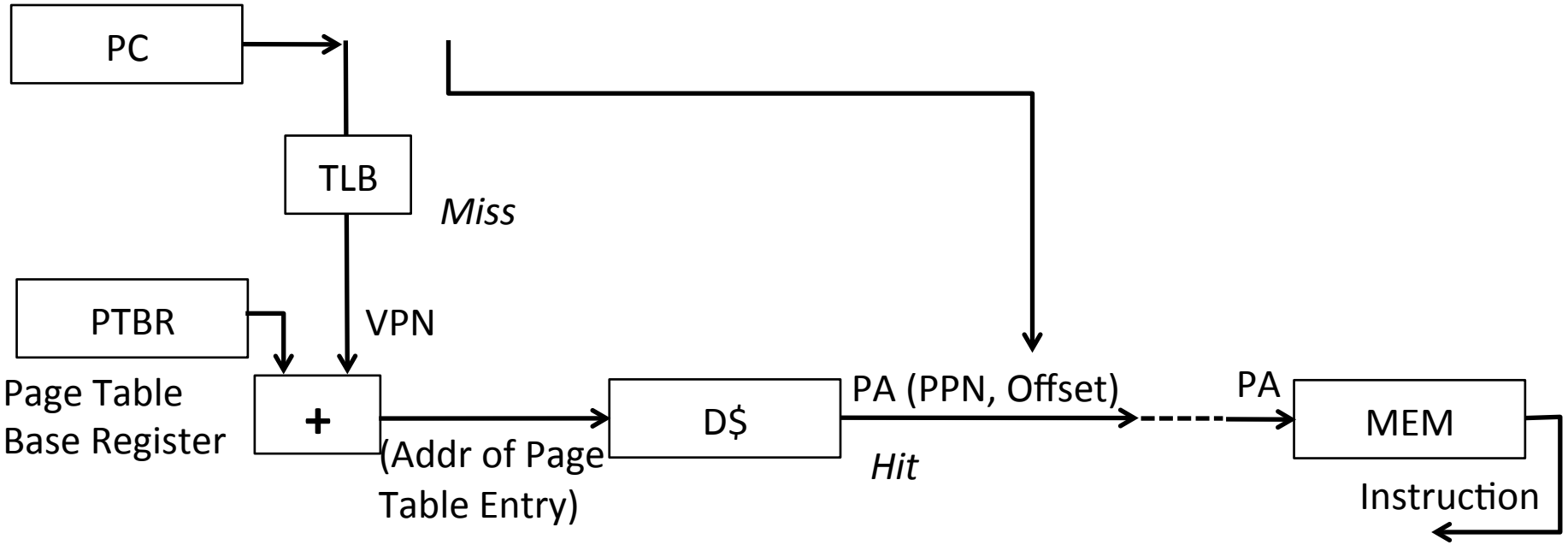
No Cache, No Virtual Memory
(Note: PA - Physical Address, VA - Virtual Address)

Day in the Life of an (Instruction) Address



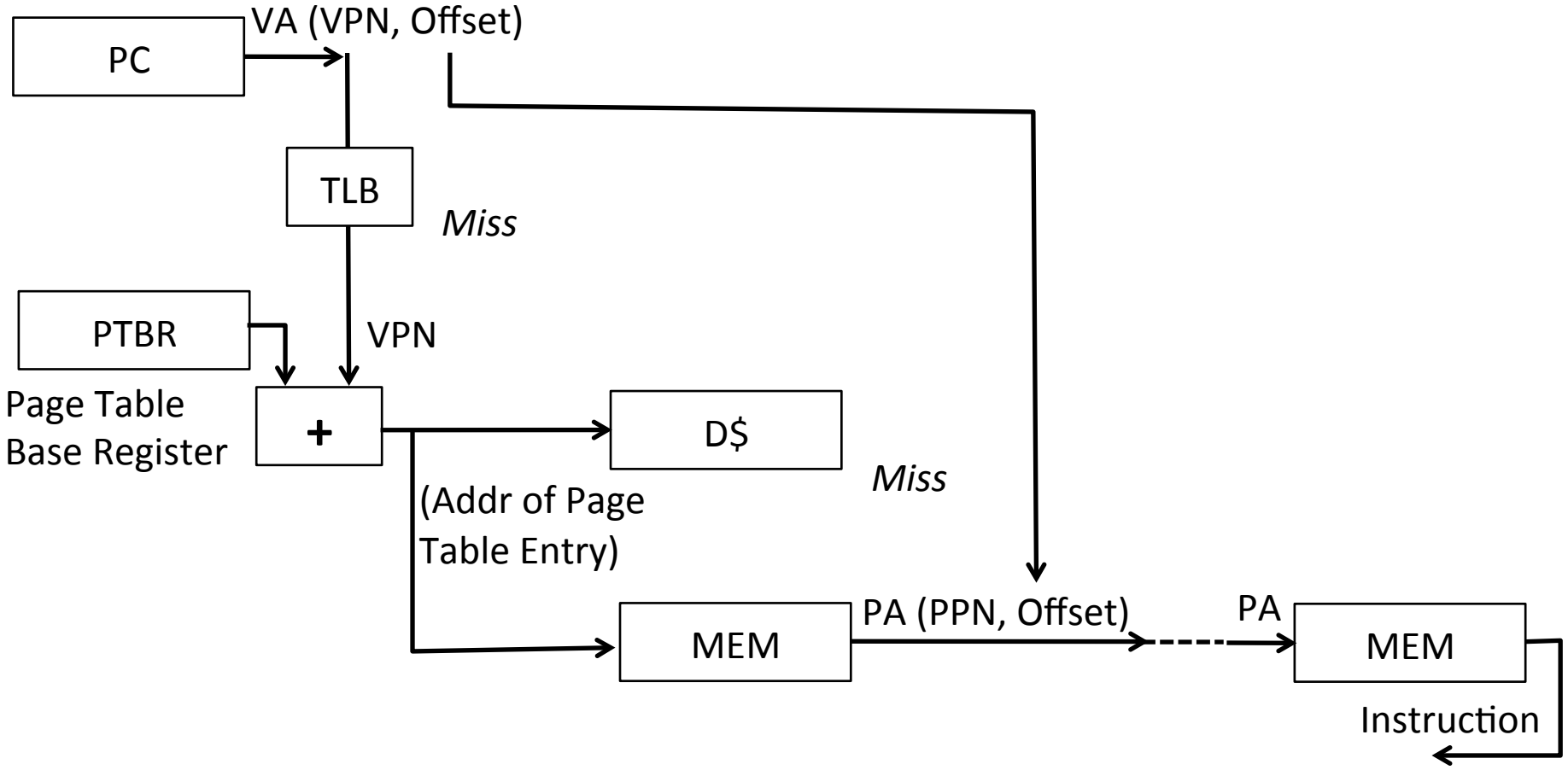
No Cache, Virtual Memory, TLB Hit

Day in the Life of an (Instruction) Address



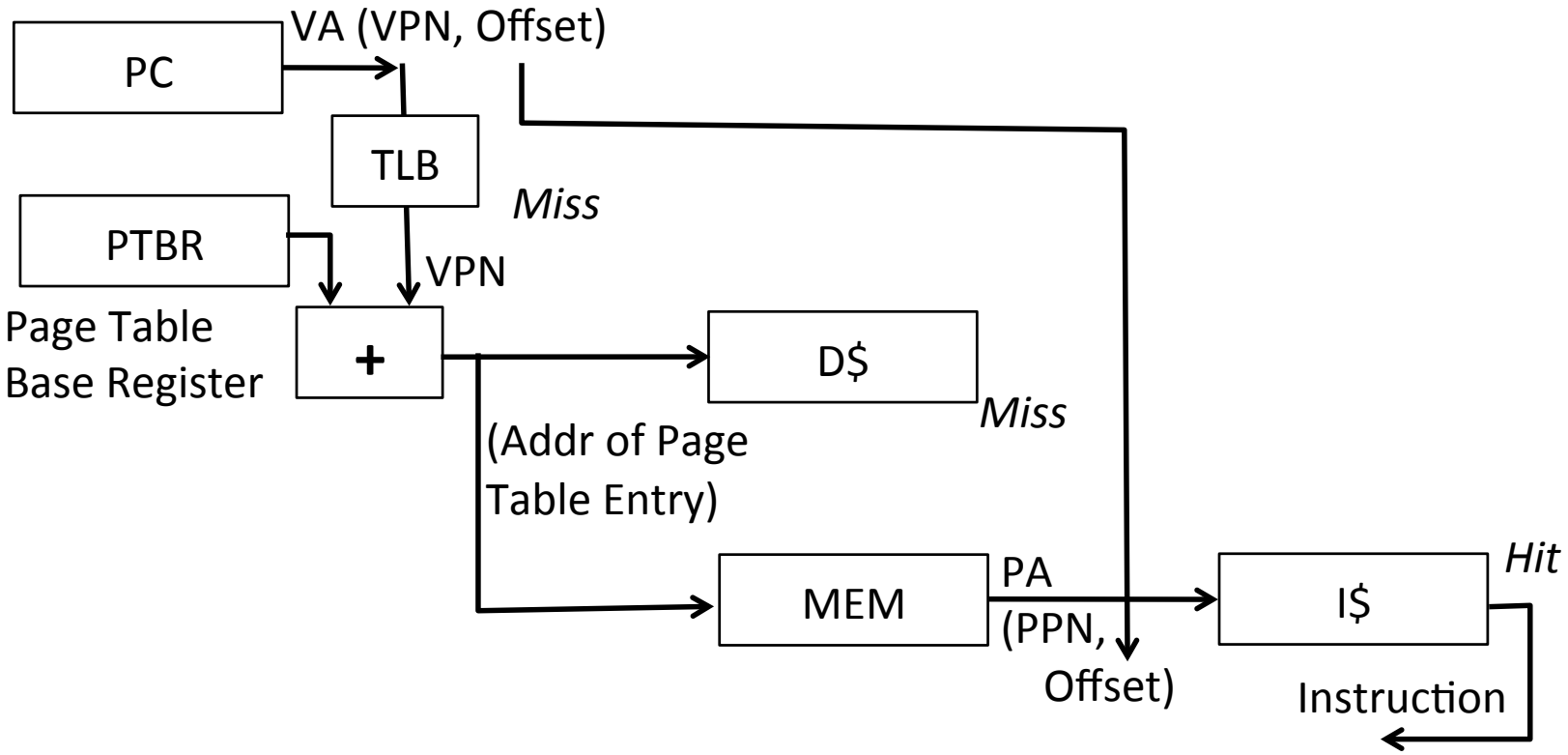
PA Data Cache, Virtual Memory, TLB Miss, Page Table in D$
(NOTE: PA cache means addresses translation before caching)

Day in the Life of an (Instruction) Address



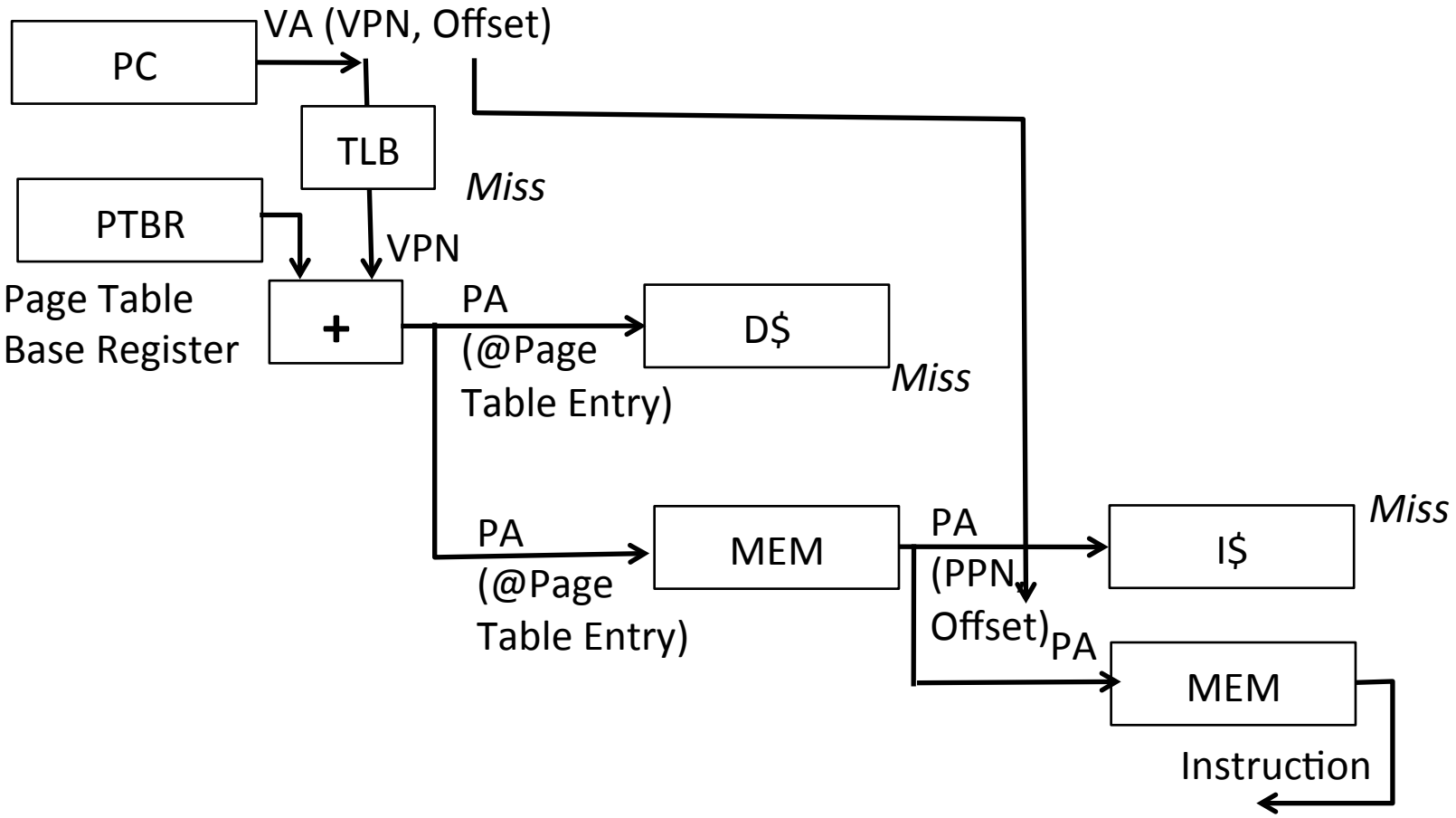
PA Data Cache, Virtual Memory, TLB Miss, Page Table not in D$, PT Access in Memory

Day in the Life of an (Instruction) Address



PA Data & Instruction Cache, Virtual Memory, TLB Miss, Page Table not in D$, PT Access in Memory, Instruction in I$

Day in the Life of an (Instruction) Address



PA Data & Instruction Cache, Virtual Memory, TLB Miss, Page Table Access, Instruction not in I$, Instruction access in Memory

Hit/Miss possibilities

TLB	Page	Cache	Remarks
hit	hit	hit	Possible, page table not checked on TLB hit, data from cache
hit	hit	miss	Possible, page table not checked, cache block loaded from memory
hit	miss	hit	XXXXXXXXX, TLB references in-memory pages
hit	miss	miss	XXXXXXXXX, TLB references in-memory pages
miss	hit	hit	Possible, TLB entry loaded from page table, data from cache
miss	hit	miss	Possible, TLB entry loaded from page table, cache block loaded from memory
miss	miss	hit	XXXXXXXXX, cache is a subset of memory
miss	miss	miss	Possible, page fault brings in page, TLB entry loaded, cache block loaded from memory

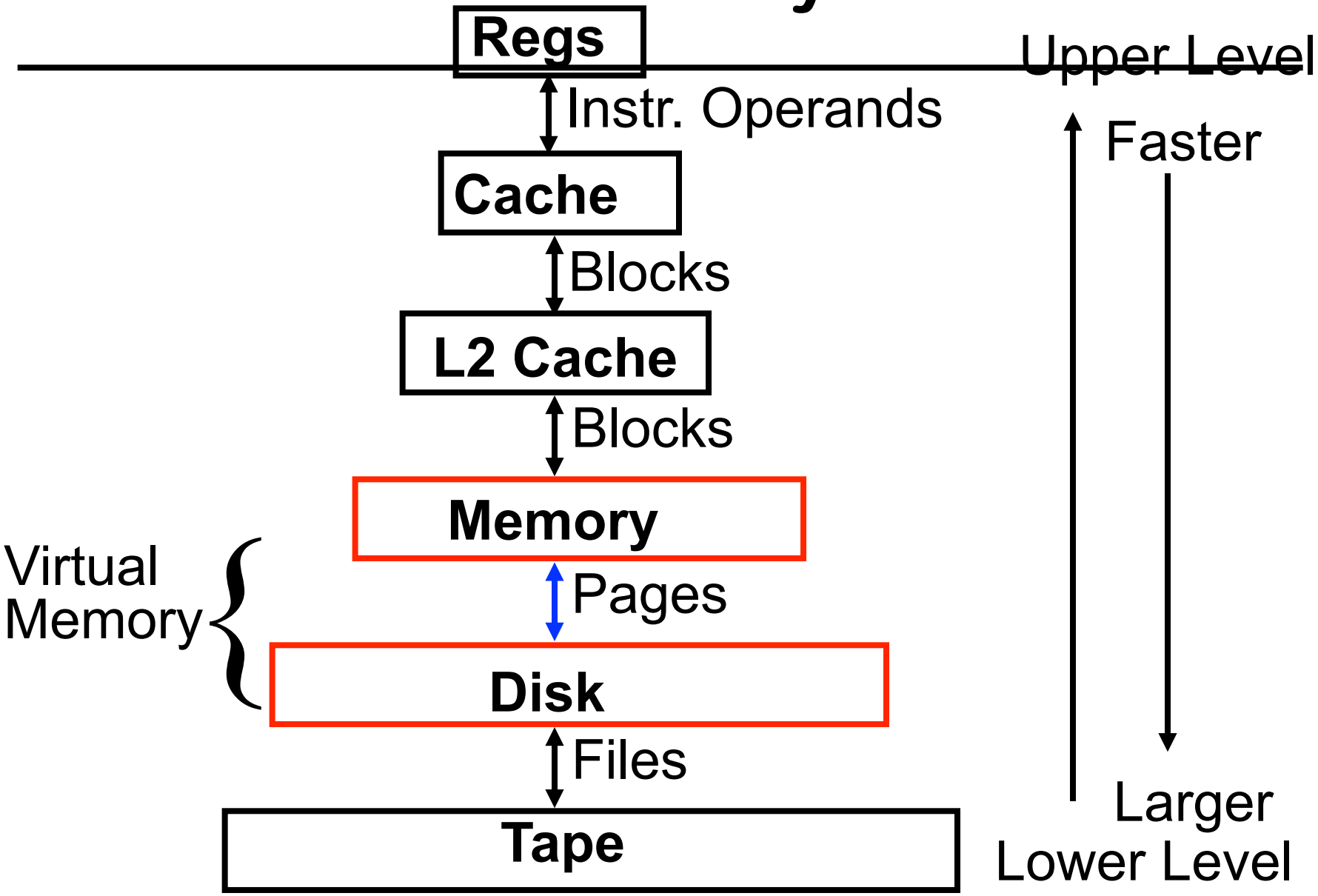
Agenda

- Virtual Memory Intro
- Page Tables
- Translation Lookaside Buffer
- Demand Paging
- System Calls
- Summary

Demand Paging

- What if required pages no longer fit into Physical Memory?
  - Think running out of RAM on a machine
- Physical memory becomes a cache for disk.
  - Page not found in memory => ***Page Fault***, must retrieve it from disk.
  - Memory full => Invoke replacement policy to swap pages back to disk.

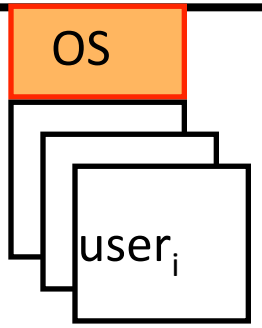
Just Another Level in the Memory Hierarchy



VM Provides
Illusion of a large, private, uniform store

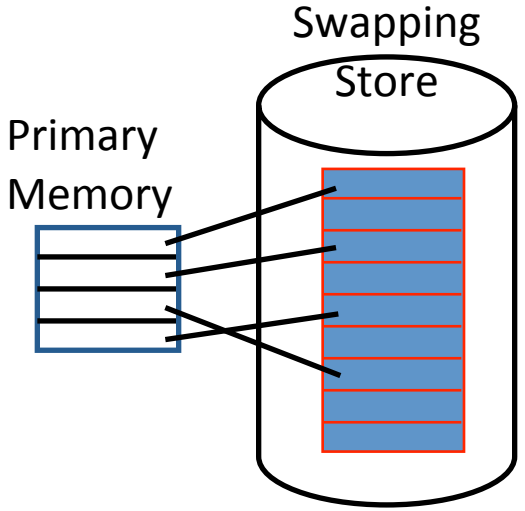
Protection & Privacy

- Several users, each with their private address Space and one or more shared address spaces

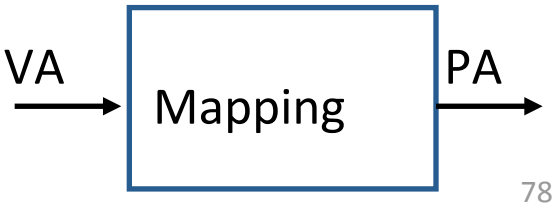


Demand Paging

- Provides ability to run programs larger than the primary memory



Price is address translation on each memory reference;
And disk so slow that performance suffers if going to disk all the time ("thrashing")



Historical Retrospective: 1960 versus 2010

- Memory expensive $$$; Now desktop memory is cheap: <$10 per GB
- New Limitation : Power. Apps intensive.

- Many apps' data could not fit
  - Big data and analytics is getting there)
- Programmers moved data memory and distk
  - OS takes care of it.

BLAST FROM THE PAST

- CELL PHONES -
- Maximum # of applications limited.
- If Total space required exceeds space; application shut down.
- Limited memory space.

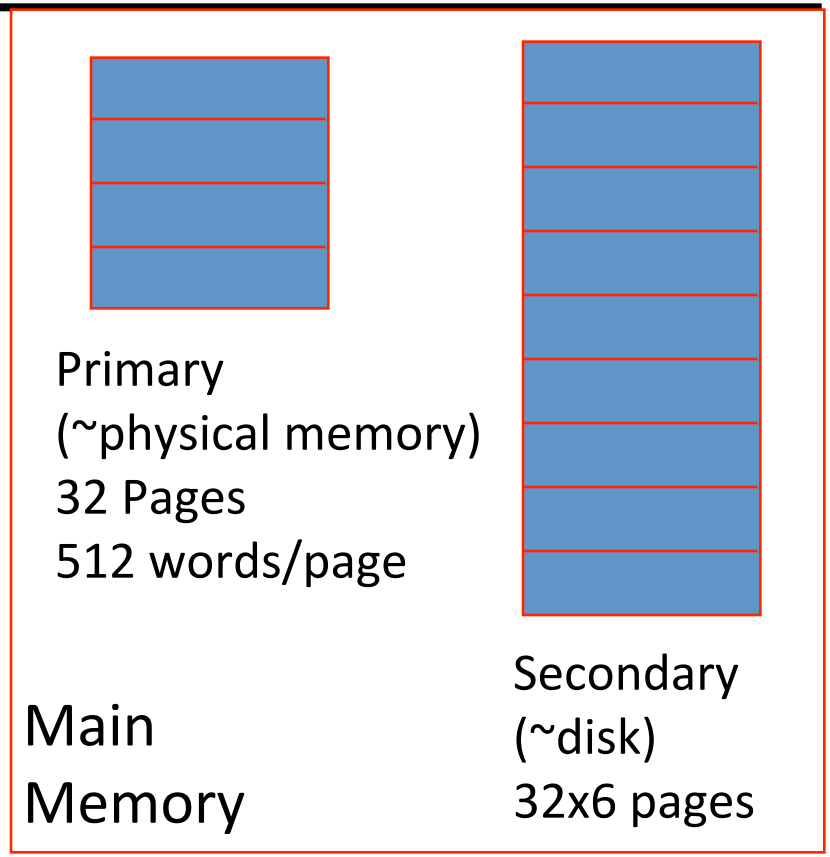
Demand Paging in Atlas (1962)

"A page from secondary storage is brought into the primary storage whenever it is (implicitly) demanded by the processor."

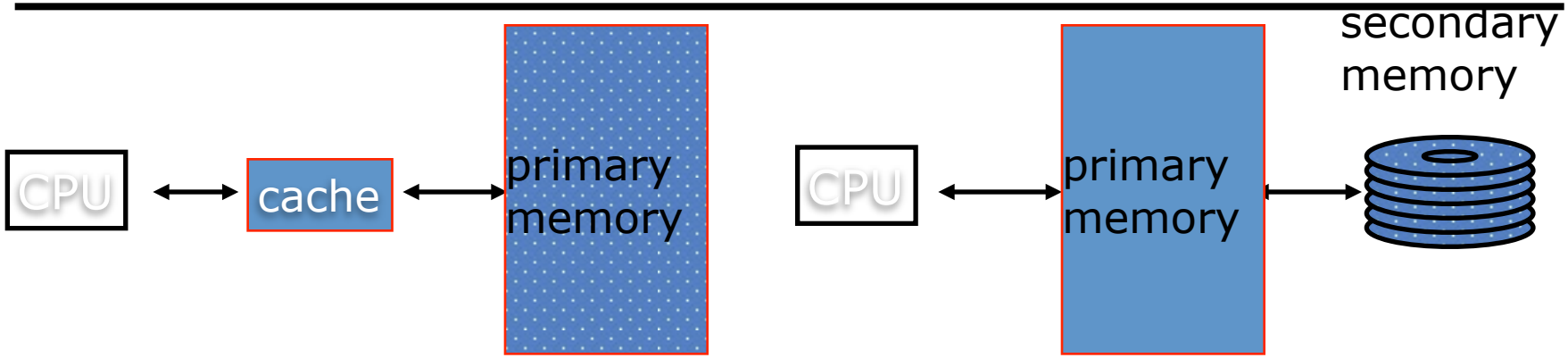
Tom Kilburn

Primary memory as a *cache* for secondary memory

User sees 32 x 6 x 512 words of storage



Caching vs. Demand Paging



Caching

- cache entry
- cache block (~32 bytes)
- cache miss rate (1% to 20%)
- cache hit (~1 cycle)
- cache miss (~100 cycles)
- a miss is handled
  - in *hardware*

Demand paging

- page frame
- page (~4K bytes)
- page miss rate (<0.001%)
- page hit (~100 cycles)
- page fault (~5M cycles)
- a miss is handled
  - mostly in *software*

Design Issues

- Design issues for VM are related to HUGE cost of a miss
  - Accessing disk may take MILLIONS of clock cycles
    - Cache (SRAM): 5-25ns
    - Memory (DRAM): 60-120ns
    - Disk: 10-20 million ns
- Page size should be large enough to cover page miss cost
  - transfer time is much less than access time
  - 4KB to 16KB common (newer systems: 32KB - 64KB)
- Reducing page fault rate has high priority
  - fully-associative page placement
  - write back + write allocate (instead of write though) to minimize writes to disk
- Page faults are handled in software by OS
  - overhead is small compared to cost of disk access
  - use clever algorithms to minimize page faults

Demand Paging Scheme

- On a page fault:
  - Allocate a free page in memory, if available.
  - If no free pages, invoke replacement policy to select page to swap out.
  - Replaced page is written to disk
  - *Page table is updated* - The entry for the replaced page is marked as invalid. The entry for the new page is filled in.

Demand Paging

- OS must reserve Swap Space on disk for each process
  - Place to put swapped out pages.
- To grow a process, ask OS
  - If unused pages available, OS uses them first
  - If not, OS swaps some old pages to disk
  - Many page replacement algorithms, discussed next lecture

Impact on TLB

- Keep track of whether written page
- Set “Page Dirty Bit” in TLB when any data in page is written
- TLB entry replaced, corresponding PageDirty Bit is set in PageTable Entry

Agenda

- Virtual Memory Intro
- Page Tables
- Translation Lookaside Buffer
- Demand Paging
- Summary

Summary #1

- Virtual Memory supplies two features:
  - *Translation* (mapping of virtual address to physical address)
  - *Protection* (permission to access word in memory)
- All desktops/servers have full demand-paged Virtual Memory
  - Portability between machines with different memory
  - Share small physical memory among active tasks
- HW protection: User/Kernel Mode (cannot touch PTE).

Summary #2

- PTE: Page Table Entries
  - Includes physical page number
  - Control info (valid bit, writeable, dirty, user, etc)
- "Translation Lookaside Buffer" (TLB)
  - Relatively small number of entries (< 512)
  - Fully Associative
  - TLB entries contain PTE and optional ASID

Summary #3

- On TLB miss, page table must be traversed
  - If located PTE is invalid, cause Page Fault
- On context switch/change in page table
  - TLB entries must be invalidated (If ProcessID is not contained)
- TLB is logically in front of cache
  - But can be overlapped with cache access in some cases

Quiz: Tag Bits

- Q: For a fully-associate cache, tag bits in each cache block are used to disambiguate among multiple candidate memory blocks that can map to the same cache block. For fully-associative page placement, and # virtual pages ≥ # physical page frames → VPN bits ≥ PPN bits. Hence it seems like a Tag field with VPN bits - PPN bits is needed. Why are there no tag bits in each page table entry?
- A: The page table performs a one-way translation: it always maps from virtual page to physical page, but not vice versa, hence we do not need to distinguish between multiple possible virtual pages mapped to the same physical page → no need for tags.

Quiz: Write-Through vs Write-Back

- Q: On a write, should we write the new value through to (memory/disk) or just keep it in the (cache/memory) and write it back to (memory/disk) when the (cache-block/page) is replaced?
- A:
- Write-back has fewer writes to (memory/disk) since multiple writes to the (cache-block/page) may occur before the (cache-block/page) is evicted.
- For caching, the cost of writing through to memory is less than 100 cycles, so the cost of write through is bearable
- For paging, the cost of writing through to disk is on the order of 1,000,000 cycles. Since write-back has fewer writes to disk, it is used.

Quiz: True or False

Each process must have its own Page Table.

True. Each VPN in each address space must have its own mapping.

Quiz I: True or False

A program tries to load a word at X that causes a TLB miss but not a page fault. Which are True or False:

1. A TLB miss means that the page table does not contain a valid mapping for virtual page corresponding to the address X
2. There is no need to look up in the page table because there is no page fault
3. The word that the program is trying to load is

Red) 1 F, 2 F, 3 F

Blue) 1 F, 2 F, 3 T

Green) 1 F, 2 T, 3 F

Yellow) 1 F, 2 T, 3 T

Purple) 1 T, 2 F, 3 F

White) 1 T, 2 F, 3 T

Quiz: Write-Allocate

- Recall “write allocate” in caching: fetch the block from memory to cache before writing to cache.
  - Consider a cache with 2-word (8-byte) cache block size. Assume we do a 4-byte write to memory location 0x000000 and causes a cache miss. We have to load the entire block (Bytes 0-7) from memory to cache before we write the cache tag and the 4-byte data into Bytes 0-3.
- Q: For paging, do we need to fetch the page from disk to memory before writing to a page?
- A: Yes, if a write causes a page fault; then we have to load the entire page from disk to memory before we write the word.

Quiz II

- Consider a processor with 32-bit virtual addresses, and 28-bit real addresses.
  - A 4-way set associative cache that can hold 512 cache blocks; cache block size is 128 Bytes
  - A 2-way set associative TLB with 64 entries; memory page size is 4096 Bytes.
- The cache is physically addressed and memory references use byte addressing. Describe in detail the steps involved in processing a load instruction which results in a TLB hit and a cache hit.

TLB and Cache Both Hit

- Cache Tag:Index:Offset is 14:7:7 (physical memory address is 28 bits)
  - Offset =7 bits since the cacheline size is 128=2^7 Bytes; Index=7 bits since there are 512/4 = 128 = 2^7 sets (each set has 4 cache blocks). The remaining higher-order bits are the Tag=28-(7+7)=14
- TLB Tag:Index:Offset is 15:5:12 (virtual memory address is 32 bits)
  - Offset=12 bits since the page size is 4096 = 2^12 Bytes; Index=5 bits since there are 64/2 = 32 = 2^5 sets in the TLB (each set has 2 entries). The remaining high-order bits are the Tag = 32-(5+12)=15. The physical page number in each TLB entry is 28-12=16 bits.
- We first look up the memory address in the TLB. We use the index field, bits 16–12 (numbering from 31 for the left-most bit and 0 for the right-most one), to index into the TLB. There are two entries at this index (since the TLB is 2-way set associative) – we check to see if either entry there has a tag field that matches the 15 bit tag of the virtual address (bits 31–17). We are told that this is a TLB hit, so one of the two entries in this set matches the tag (and the corresponding valid bit is "1"). So we take the 16 bit PPN in the TLB and prepend it to the 12 bit offset of the virtual address to form the 28 bit physical address.
- Now we need to use this physical address to look up the data in cache. We use the index (bits 7–13 of the physical address) to choose a set, and compare the tag (bits 27–14 of the physical address) to the 4 tags stored in the chosen set. Again, we are told it is a cache hit, so the corresponding line in cache has the desired data. We use the offset field (bits 0–6) of the physical address to tell us which byte in the cache block is the first byte of the data that we are looking for.

TLB and Cache Both Miss

- In the case of cache and TLB misses, we start out as before, but don't find the desired virtual address in the TLB. Thus, we must use the virtual page number (bits 31-12 of the virtual address) to index into the page table. There we find the physical page number. Let's assume the "valid" bit in the page table is set to "1", meaning the page actually is in main memory. (If not, we need to invoke the operating system to go off to disk and find it). So we form the physical address from the 16 bit physical page number (in the page table) concatenated with the 12 bit offset. We also bring this information into the TLB using the TLB index described earlier. Again, we break the physical page number into tag-index-offset as described earlier, look for the data in cache, and don't find it. Thus, we go to main memory and bring into cache the block that holds the desired data, kicking out the least recently block in the set at the referenced index. If the replaced block happens to be dirty, we must also write it back to memory.