Lecture 8: Memory Management

CSE 120: Principles of Operating Systems

UC San Diego: Summer Session I, 2009
Frank Uyeda

Announcements

- PeerWise questions due tomorrow.
- Project 2 is due on Friday.
  - Milestone on ~~Tuesday night~~. Tonight.
- Homework 3 is due next Monday.

PeerWise

- A base register contains:
  - The first available physical memory address for the system
  - The beginning physical memory address of a process's address space
  - The base (i.e. base 2 for binary, etc) that is used for determining page sizes
  - The minimum size of physical memory that will be assigned to each process

PeerWise

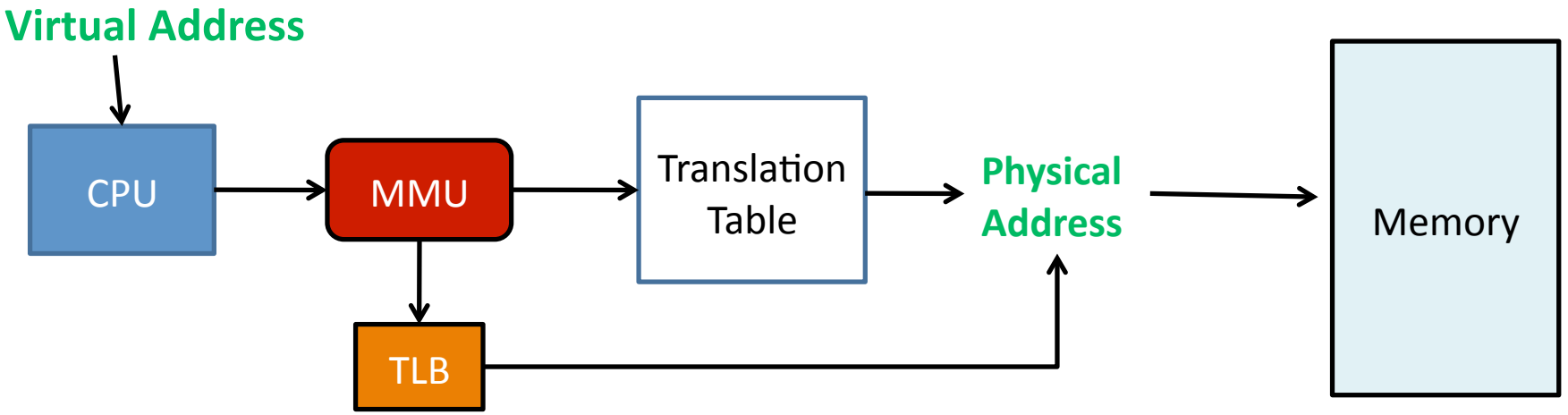
- If the base register holds 640000 and the limit register is 200000, then what range of addresses can the program legally access?
  - 440000 through 640000
  - 200000 through 640000
  - 0 through 200000
  - 640000 through 840000

Goals for Today

- Understand Paging
  - Address translation
  - Page Tables
- Understand Segmentation
  - How it combines features of other techniques

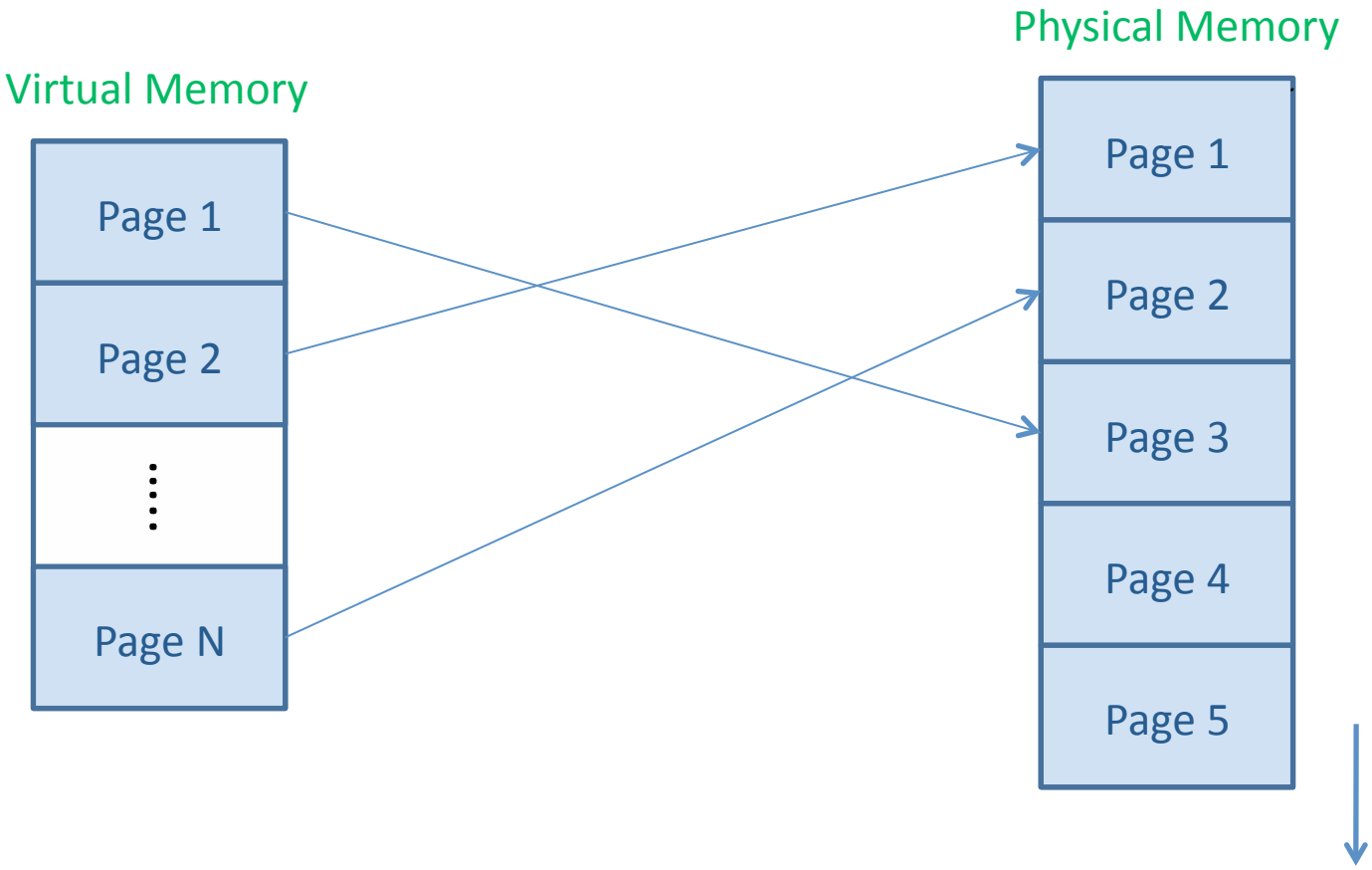
Recap: Virtual Memory

- Memory Management Unit (MMU)
  - Hardware unit that translates a virtual address to a physical address
  - Each memory reference is passed through the MMU
  - Translate a virtual address to a physical address
- Translation Lookaside Buffer (TLB)
  - Essentially a cache for the MMU's virtual-to-physical translations table
  - Not needed for correctness but source of *significant* performance gain

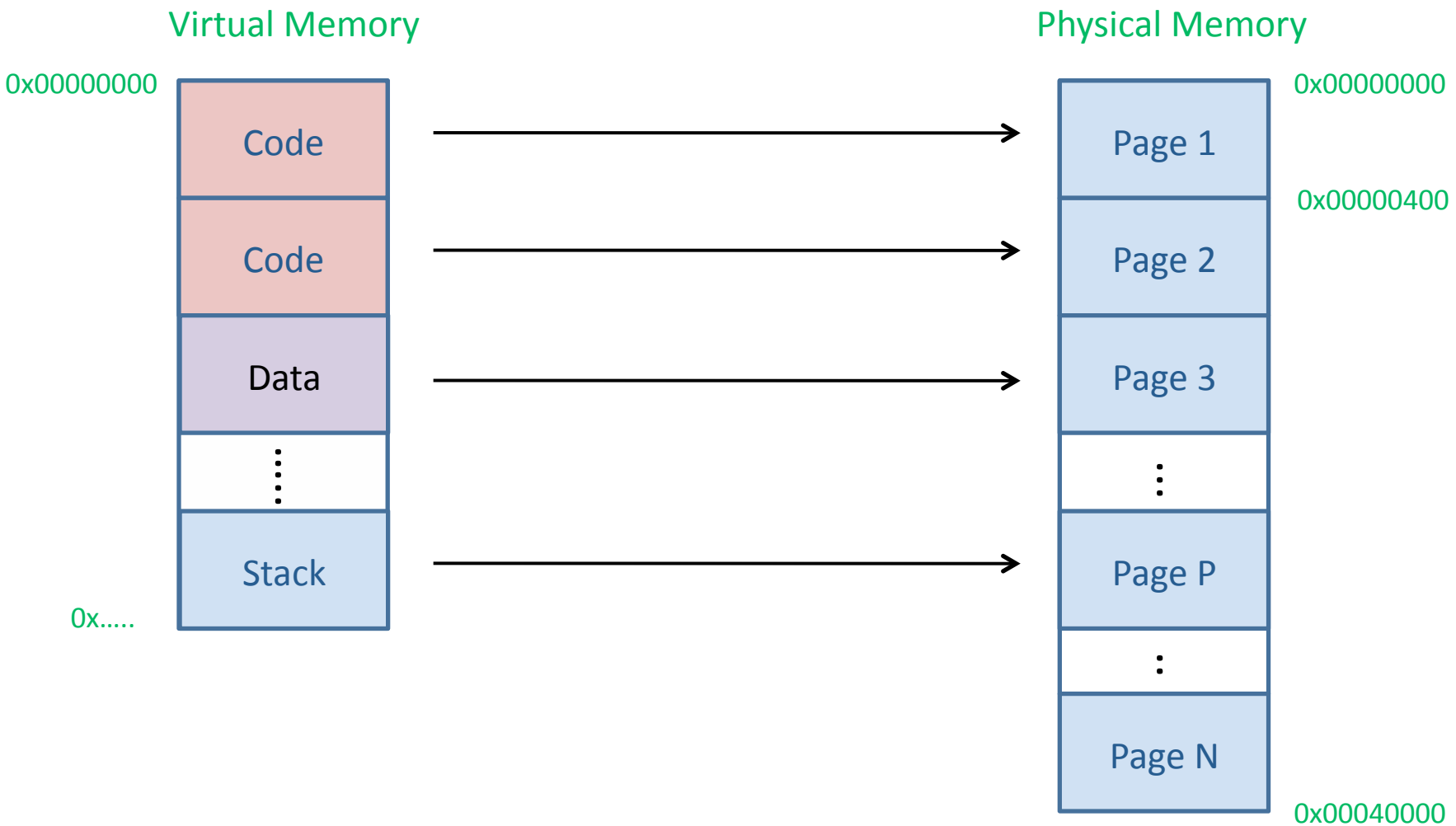


Recap: Paging

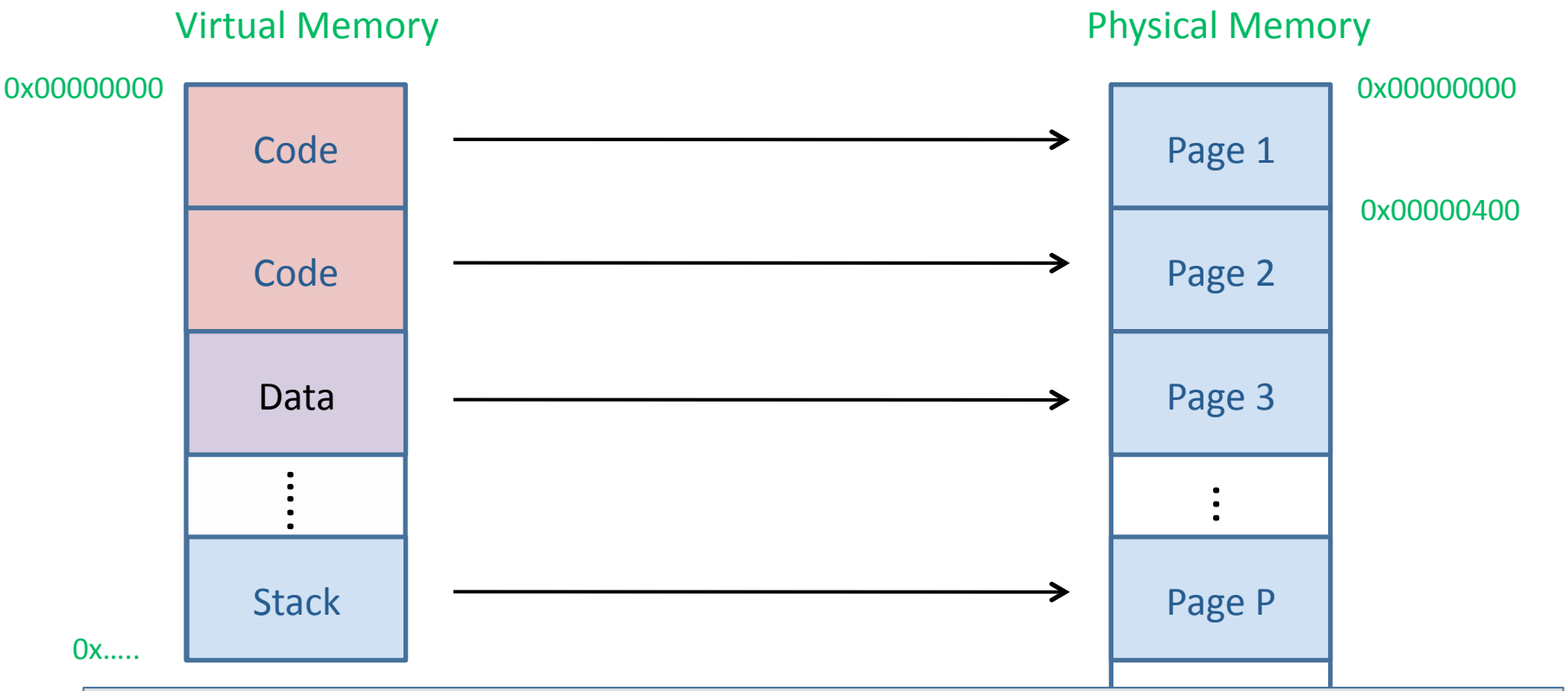
- Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory



Memory Management in Nachos



Memory Management in Nachos



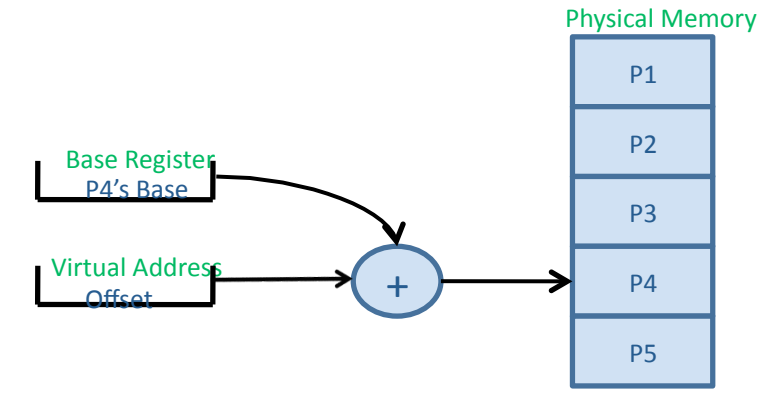
- What if code section isn't a multiple of page size?
- What if 0x..... is larger than physical memory?
- What if 0x..... is smaller than physical memory?

Memory Management in Nachos

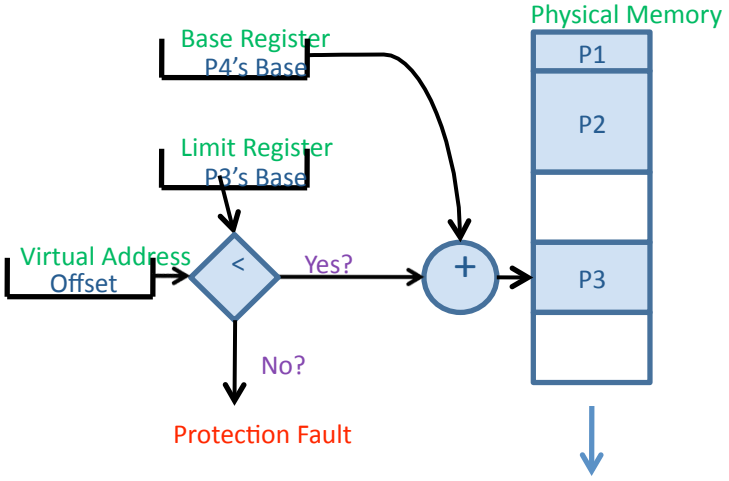
- How do we know how large a program is?
  - A lot is done for you in userprog/UserProcess.java
    - Your applications will be written in “C”
    - Your “C” programs compile into .coff files
    - Coff.java and CoffSection.java are provided to partition the coff file
  - If not enough physical memory, exec() returns error
    - Nachos exec() is different from Unix exec()!!!!!
- Virtual Memory maps exactly to Physical Memory?
  - Who needs a page table, TLB or MMU?
  - What are the implications for this in terms of multiprogramming?

Project 2: Multiprogramming

- Need to support for multiprogamming
  - Programs coexist on physical memory
  - How do we do this?

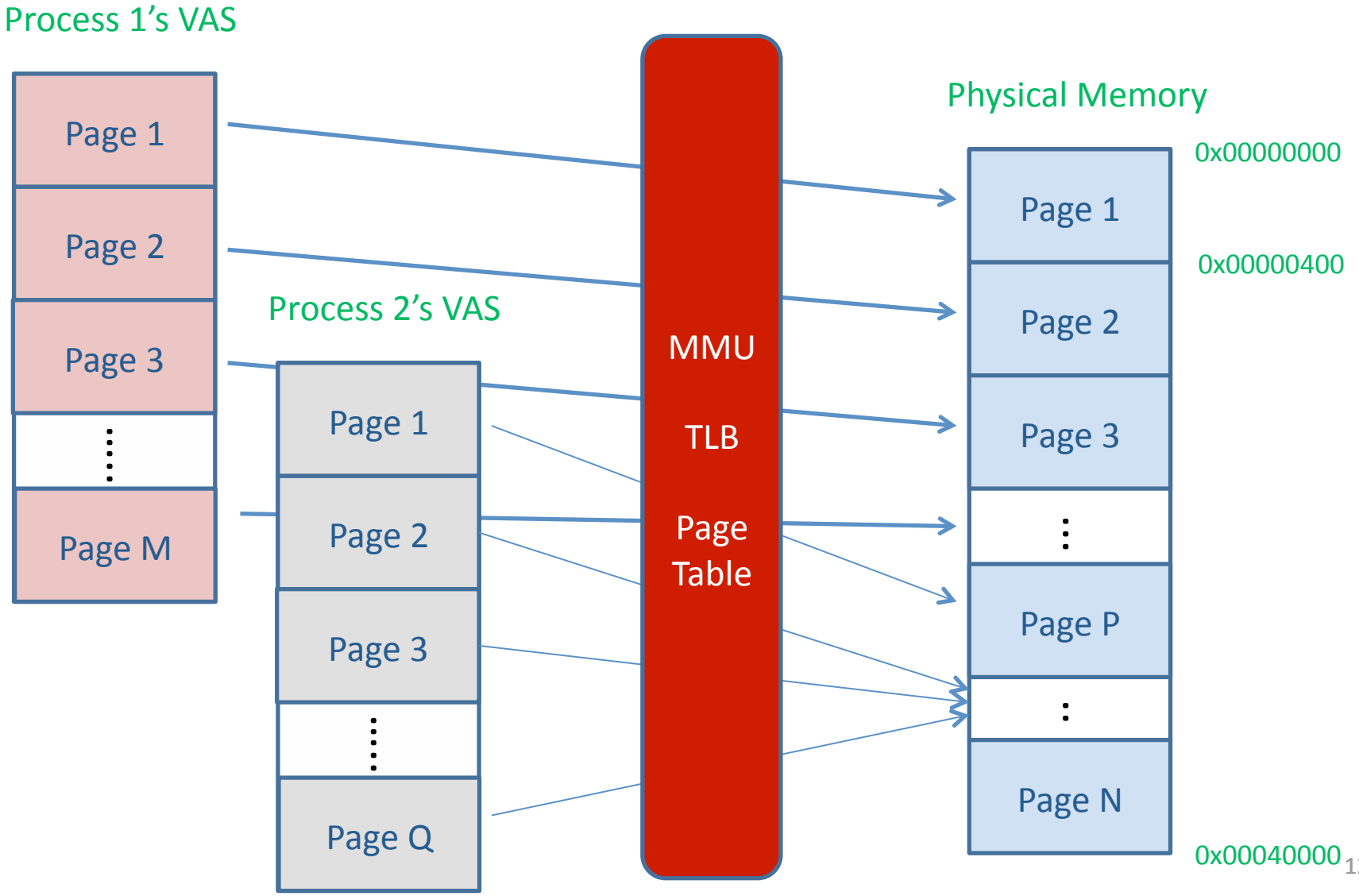


Would Fixed Partitions work?



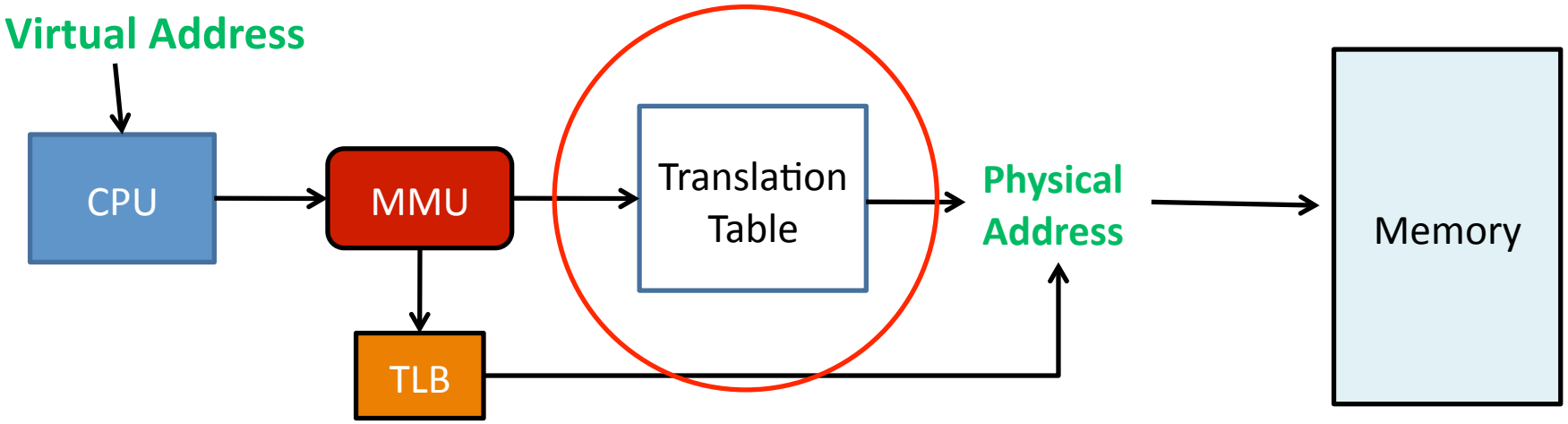
Would Variable Partitions work?

Paging


Process 1's VAS
Page 1
Page 2
Page 3
Page M
Process 2's VAS
Page 1
Page 2
Page 3
Page Q
MMU
TLB
Page Table
Physical Memory
0x00000000
Page 1
0x00000400
Page 2
Page 3
Page P
Page N
0x00040000

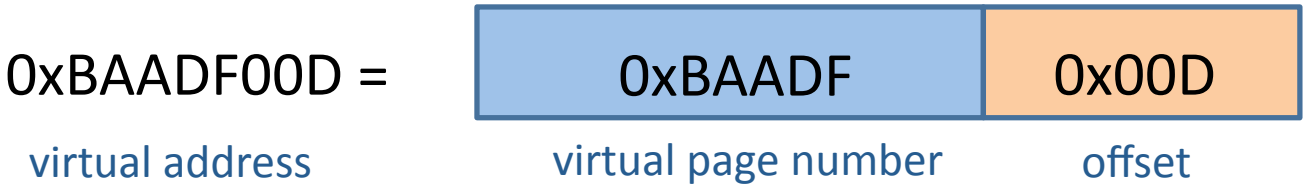
MMU and TLB

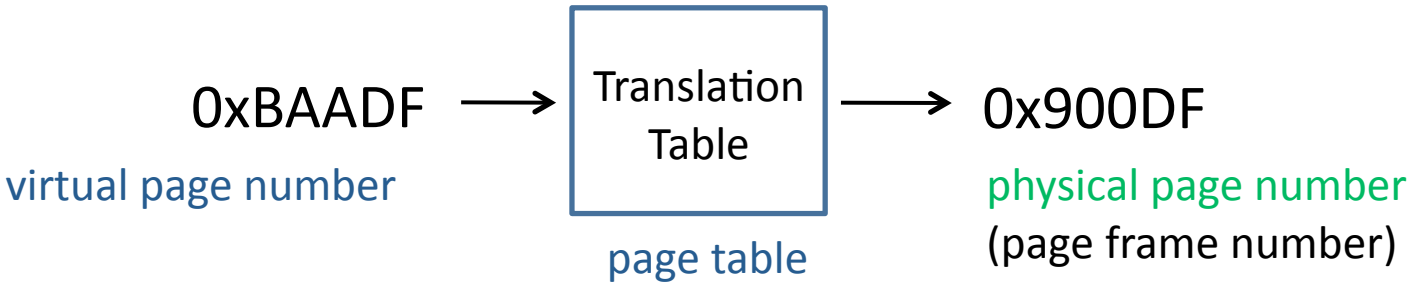
- Memory Management Unit (MMU)
  - Hardware unit that translates a virtual address to a physical address
  - Each memory reference is passed through the MMU
  - Translate a virtual address to a physical address
- Translation Lookaside Buffer (TLB)
  - Essentially a cache for the MMU's virtual-to-physical translations table
  - Not needed for correctness but source of *significant* performance gain



Paging: Translations

- Translating addresses
  - Virtual address has two parts: virtual page number and offset
  - Virtual page number (VPN) is an index into a page table
  - Page table determines page frame number (PFN)
  - Physical address is PFN::offset

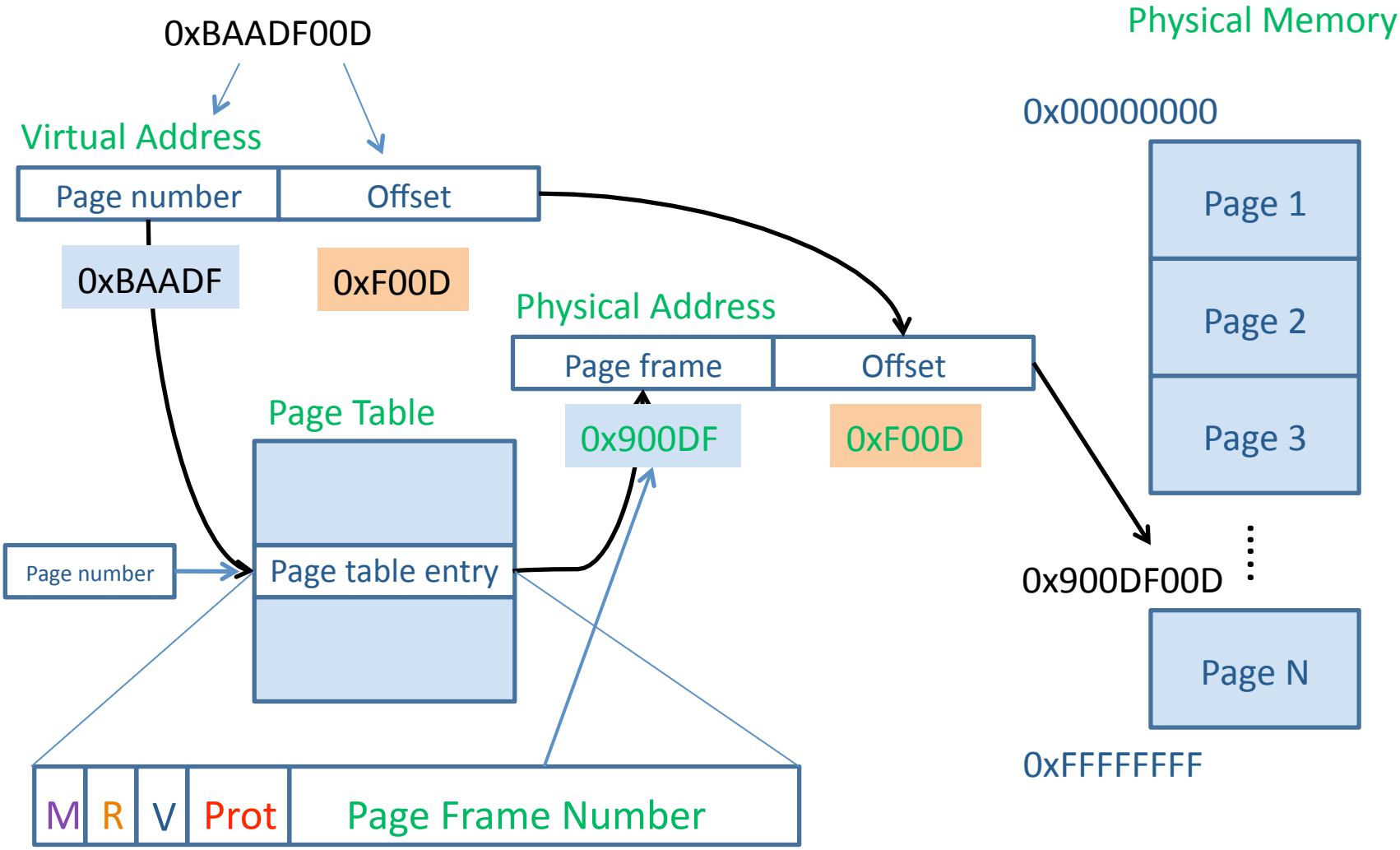




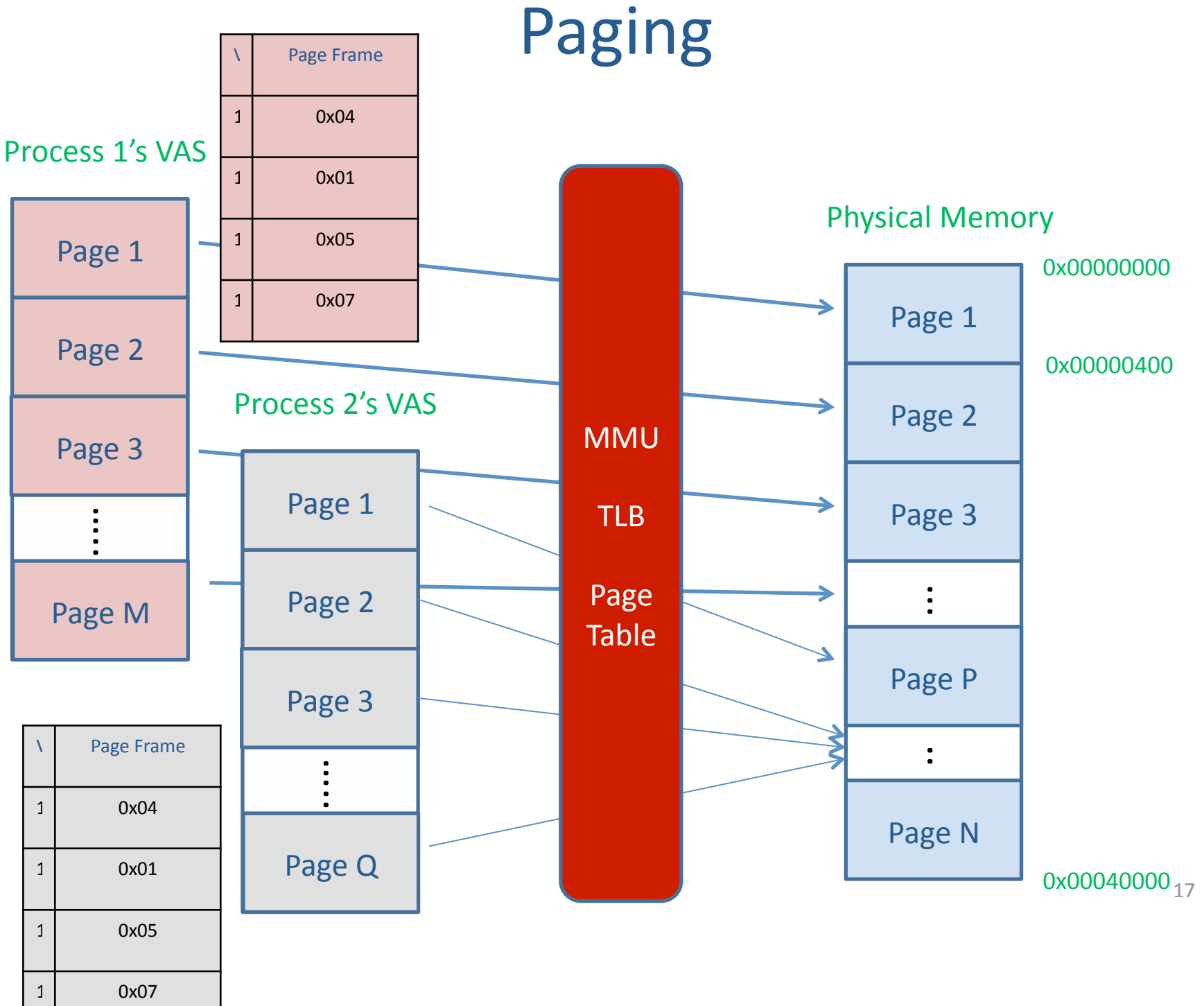
Page Table Entries (PTEs)

- Page table entries control mapping
  - The Modify bit says whether or not the page has been written
    - It is set when a write to the page occurs
  - The Reference bit says whether the page has been accessed
    - It is set when a read or write to the page occurs
  - The Valid bit says whether or not the PTE can be used
    - It is checked each time the virtual address is used
  - The Protection bits say what operations are allowed on page
    - Read, write, execute
  - The page frame number (PFN) determines physical page
- Note: when you do exercises in exams or hw, we'll often tell you to ignore counting M,R,V,Prot bits when calculating size of structures

Paging Example Revisited

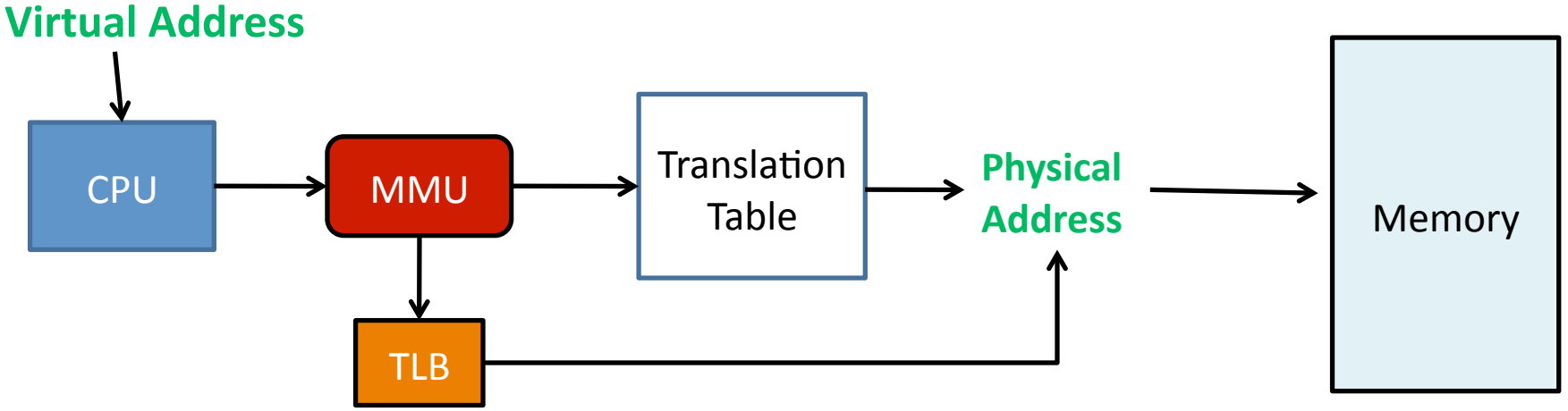


Paging


Process 1's VAS
Page 1
Page 2
Page 3
Page M
\ Page Frame
1 0x04
1 0x01
1 0x05
1 0x07
Process 2's VAS
Page 1
Page 2
Page 3
Page Q
\ Page Frame
1 0x04
1 0x01
1 0x05
1 0x07
MMU
TLB
Page Table
Physical Memory
0x00000000
0x00000400
0x00040000
Page 1
Page 2
Page 3
Page P
Page N

Example

- Assume we are using Paging and:
  - Memory access = 5us
  - TLB search = 500ns
- What is the avg. memory access time without the TLB?
- What is the avg. memory access time with 50% TLB hit rate?
- What is the avg. memory access time with 90% TLB hit rate?



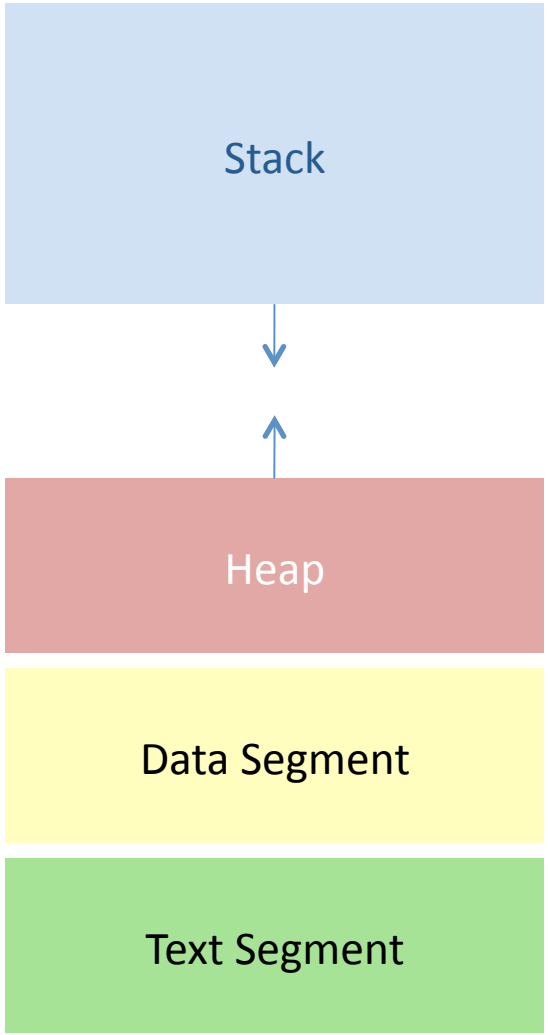
Paging Advantages

- Easy to allocate memory
  - Memory comes from a free list of fixed-size chunks
  - Allocating a page is just removing it from the list
  - External fragmentation is not a problem
- Easy to swap out chunks of a program
  - All chunks are the same size
  - Pages are a convenient multiple of the disk block size
  - How do we know if a page is in memory or not?

Paging Limitations

- Can still have internal fragmentation
  - Process may not use memory in multiples of pages
- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
  - Solution – use a hardware cache of lookups (TLB)
- Memory required to hold page table can be significant
  - Need one PTE per page
  - 32-bit address space w/4KB pages = up to ____ PTEs
  - 4 bytes/PTE = _____ MB page table
  - 25 processes = _____ MB just for page tables!
  - Solution: page the page tables (more later)

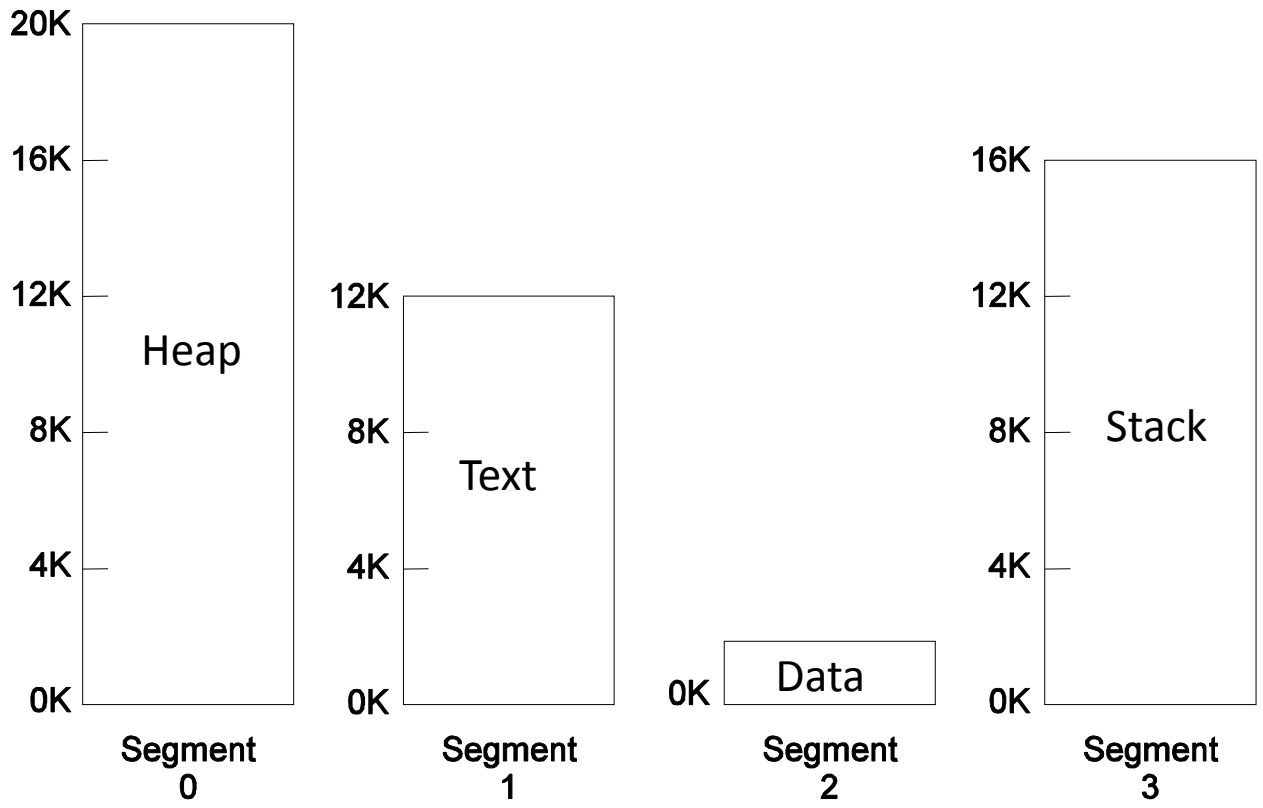
Paging: Linear Address Space



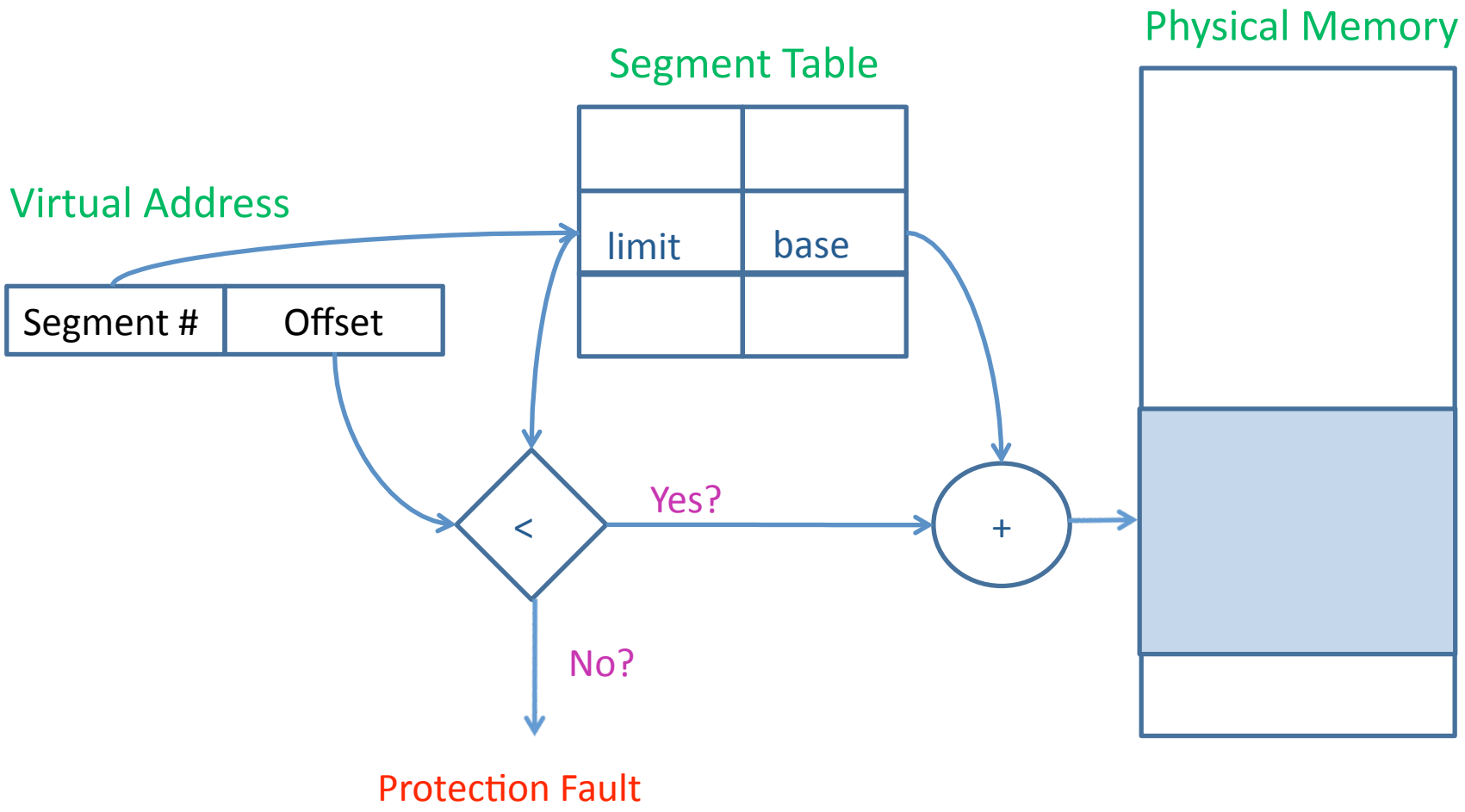
0xFFF….. (Ending Address)

0x00……. (Starting Address)

Segmentation: Multiple Address Spaces



Segmentation



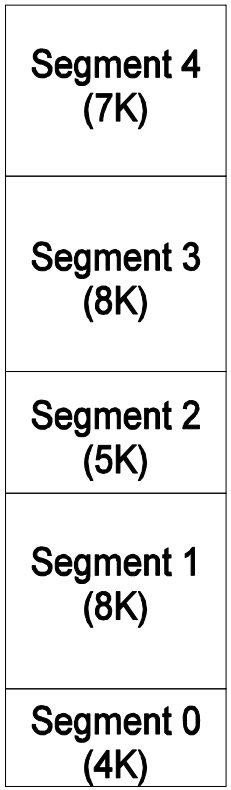
Segmentation

- Segmentation is a technique that partitions memory into logically related data units
  - Module, procedure, stack, data, file, etc.
  - Virtual addresses become <segment #, offset>
  - Units of memory from user's perspective
- Natural extension of variable-sized partitions
  - Variable-sized partitions = 1 segment/process
  - Segmentation = many segments/process
- Hardware support
  - Multiple base/limit pairs, one per segment (segment table)
  - Segments named by #, used to index into table

Segment Table

- Extensions
  - Can have one segment table per process
    - Segment #s are then process-relative (why do this?)
  - Can easily share memory
    - Put same translation into base/limit pair
    - Can share with different protections (same base/limit, diff prot)
    - Why is this different from pure paging?
  - Can define protection by segment
- Problems
  - Cross-segment addresses
    - Segments need to have same #s for pointers to them to be shared among processes
  - Large segment tables
    - Keep in main memory, use hardware cache for speed

Segmentation Example



(a)

External Fragmentation in Segmentation

Note: Image courtesy of Tanenbaum, MOS 3/e

Paging vs Segmentation

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?		
How many linear address spaces are there?		
Can the total address space exceed the size of physical memory?		
Can procedures and data be distinguished and separately protected?		
Is sharing of procedures between users facilitated?		

Note: Image adapted from Tanenbaum, MOS 3/e

Segmentation and Paging

- Can combine segmentation and paging
  - The x86 supports segments and paging
- Use segments to manage logically related units
  - Module, procedure, stack, file, data, etc.
  - Segments vary in size, but usually large (multiple pages)
- Use pages to partition segments into fixed sized chunks
  - Makes segments easier to manage within physical memory
    - Segments become "pageable" – rather than moving segments into and out of memory, just move page portions of segment
  - Need to allocate page table entries only for those pieces of the segments that have themselves been allocated
- Tends to be complex…

Virtual Memory Summary

- Virtual memory
  - Processes use virtual addresses
  - OS + hardware translates virtual addresses into physical addresses
- Various techniques
  - Fixed partitions – easy to use, but internal fragmentation
  - Variable partitions – more efficient, but external fragmentation
  - Paging – use small, fixed size chunks, efficient for OS
  - Segmentation – manage in chunks from user's perspective
  - Combine paging and segmentation to get benefits of both

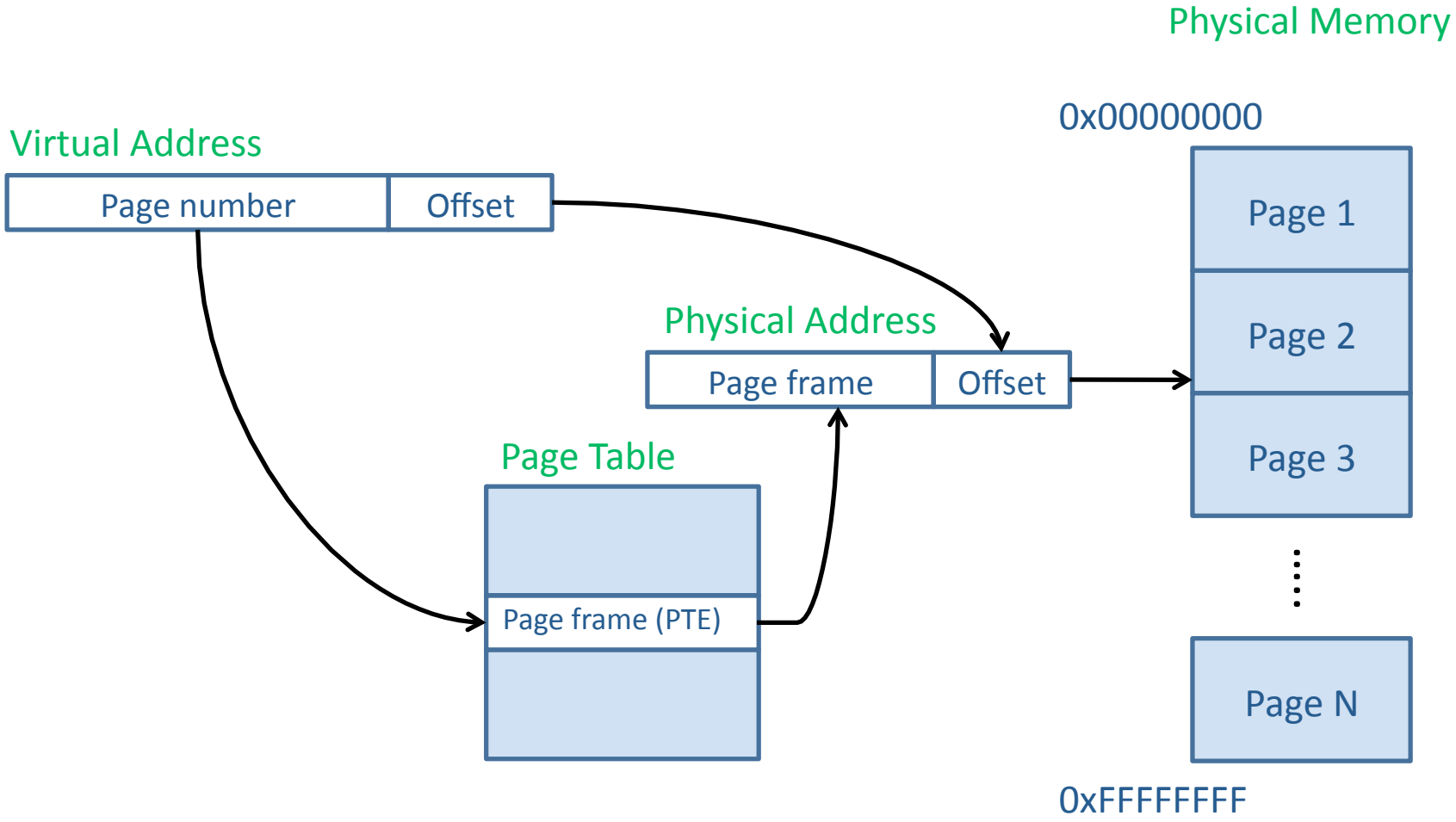
Managing Page Tables

- We computed the size of the page table for a 32-bit address space with 4K pages to be ___ MB
  - This is far too much overhead for each process
- How can we reduce this overhead?
  - Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire address space)
- How do we only map what is being used?
  - Can dynamically extend page table…
  - Does not work if address space is sparse (internal fragmentation)
- Use another level of indirection: multi-level page tables

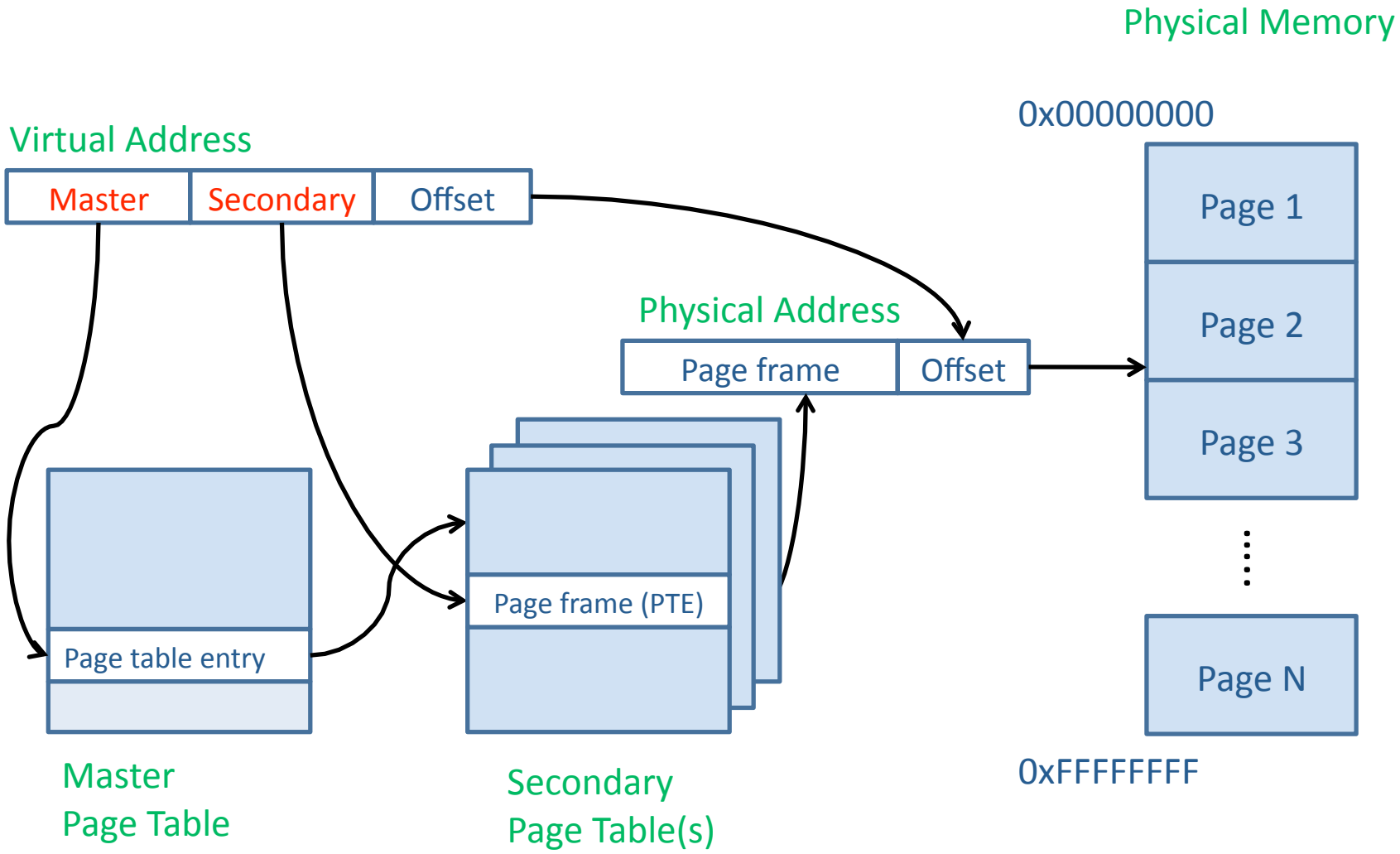
Managing Page Tables

- We computed the size of the page table for a 32-bit address space with 4K pages to be 100 MB
  - This is far too much overhead for each process
- How can we reduce this overhead?
  - Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire address space)
- How do we only map what is being used?
  - Can dynamically extend page table…
  - Does not work if address space is sparse (internal fragmentation)
- Use another level of indirection: multi-level page tables

One-Level Page Table



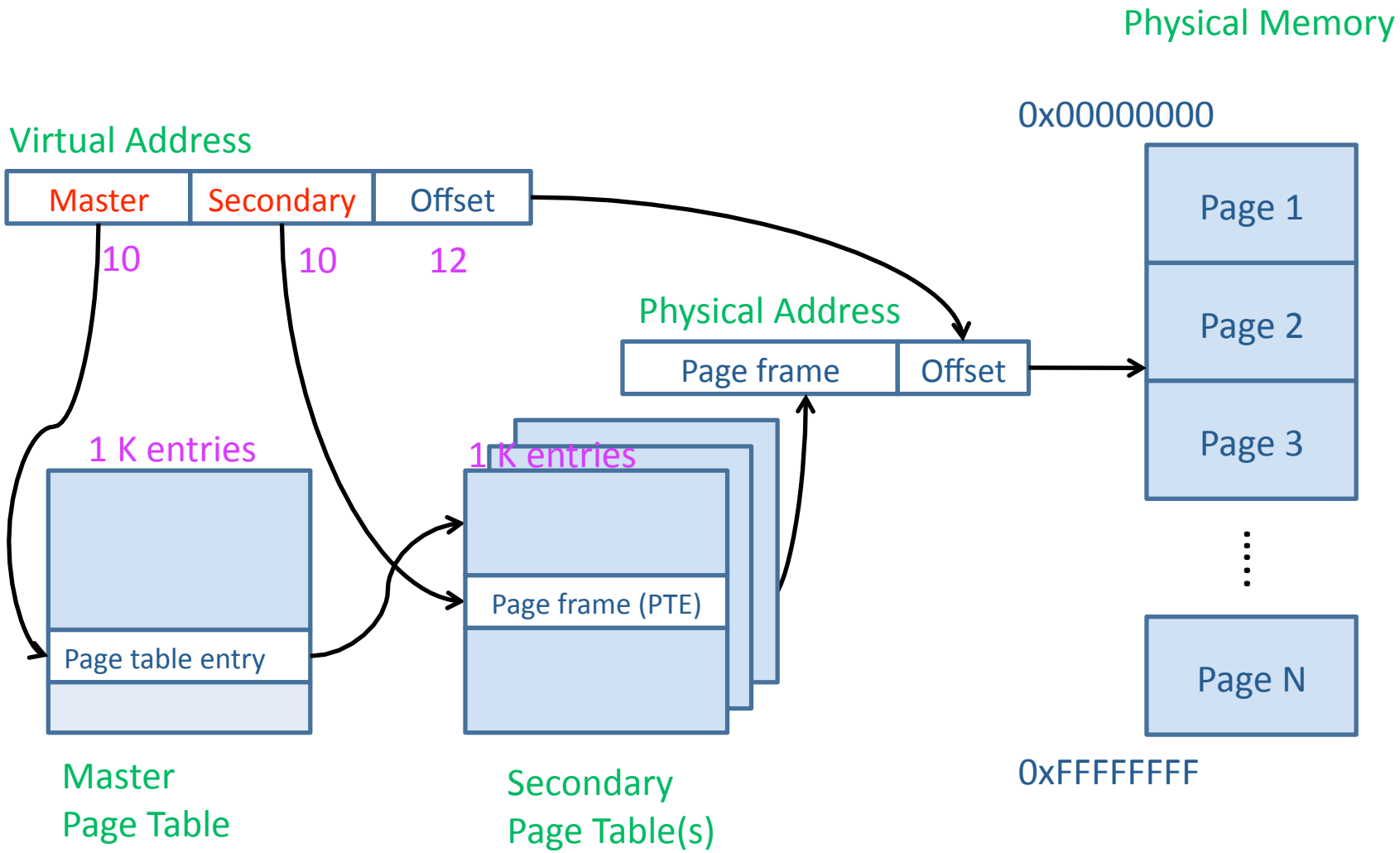
Two-Level Page Table



Two-Level Page Tables

- Originally, virtual addresses (VAs) had two parts
  - Page number (which mapped to frame) and an offset
- Now VAs have three parts:
  - Master page number, secondary page number, and offset
- Master page table maps VAs to secondary page table
  - We'd like a manageable master page size
- Secondary table maps page number to physical page
  - Determines which physical frame the address resides in
- Offset indicates which byte in physical page
  - Final system page/frame size is still the same, so offset length stays the same

Two-Level Page Table Example



Example: 4KB pages, 4B PTEs (32-bit RAM), and split remaining bits evenly among master and secondary

Where do Page Tables Live?

- Physical memory
  - Easy to address, no translation required
  - But, allocated page tables consume memory for lifetime of Vas
- Virtual memory (OS virtual address space)
  - Cold (unused) page table pages can be paged out to disk
  - But, addressing page tables requires translation
  - How do we stop recursion
  - Do not page the outer page table (an instance of page pinning or wiring)
- If we're going to page the page tables, might as well page the entire OS address space, too
  - Need to wire special code and data (fault, interrupt handlers)

Efficient Translations

- Our original page table scheme already doubled the cost of doing memory lookups
  - One lookup into the page table, another to fetch the data
- Now two-level page tables triple the cost!
  - Two lookups into the page tables, a third to fetch the data
  - And this assumes the page table is in memory
- How can we use paging but also have lookups cost about the same as fetching from memory?
  - Cache translations in hardware
  - Translation Lookaside Buffer (TLB)
  - TLB managed by Memory Management Unit (MMU)

TLBs

- Translation Lookaside Buffers
  - Translate virtual page #s into PTEs (not physical addresses)
  - Can be done in a single machine cycle
- TLBs implemented in hardware
  - Fully associative cache (all entries looked up in parallel)
  - Cache tags are virtual page numbers
  - Cache values are PTEs (entries from page tables)
  - With PTE + offset, can directly calculate physical address
- TLBs exploit locality
  - Processes only use a handful of pages at a time
    - 16-48 entries/pages (64-192K)
    - Only need those pages to be "mapped"
  - Hit rates are therefore very important

Loading TLBs

- Most address translations are handled using the TLB
  - > 99% of translations, but there are misses (TLB miss)…
- Who paces translations into the TLB (loads the TLB)?
  - Software loaded TLB (OS)
    - TLB faults to the OS, OS finds appropriate PTE, loads it into TLB
    - Must be fast (but still 20-200 cycles)
    - CPU ISA has instructions for manipulating TLB
    - Tables can be in any format convenient for OS (flexible)
  - Hardware (Memory Management Unit)
    - Must know where page tables are in main memory
    - OS maintains tables, HW accesses them directly
    - Tables have to be in HW-defined format (inflexible)

Managing TLBs

- OS ensures that TLB and page tables are consistent
  - When I changes the protection bits of a PTE, it needs to invalidate the PTE if it is in the TLB also
- Reload TLB on a process context switch
  - Invalidate all entries
  - Why? What is one way to fix it?
- When the TLB misses and a new PTE has to be loaded, a cached PTE must be evicted
  - Choosing a PTE to evict is called the TLB replacement policy
  - Implemented in hardware, often simple (e.g., Last-Not-Used)

Summary

- Segmentation
  - Observation: User applications group related data
  - Idea: Set virtual->physical mapping to mirror application organization. Assign privileges per segment.
- Page Table management
  - Observation: Per-process page tables can be very large
  - Idea: Page the page tables, use multi-level page tables
- Efficient translations
  - Observation: Multi-level page tables and other page faults can make memory lookups slow
  - Idea: Cache lookups in hardware TLB

Next Time

- Read Chapter 9
- Peerwise questions due tomorrow at midnight.
- Check Web site for course announcements
  - http://www.cs.ucsd.edu/classes/su09/cse120