

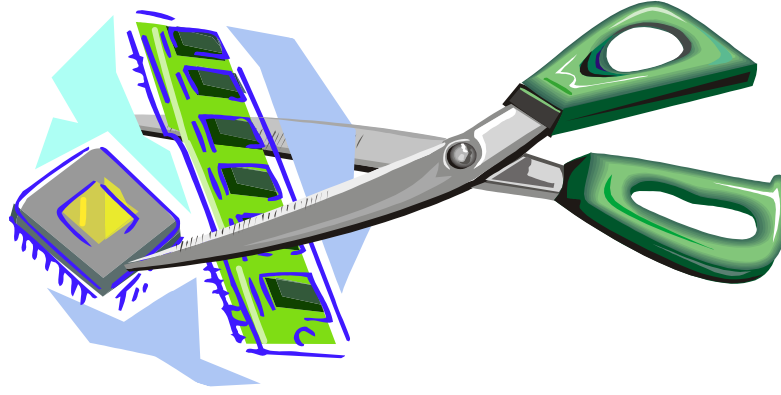
Operating Systems (Honor Track)

Memory 1: Address Translation and Virtual Memory

Xin Jin

Spring 2026

Virtualizing Resources

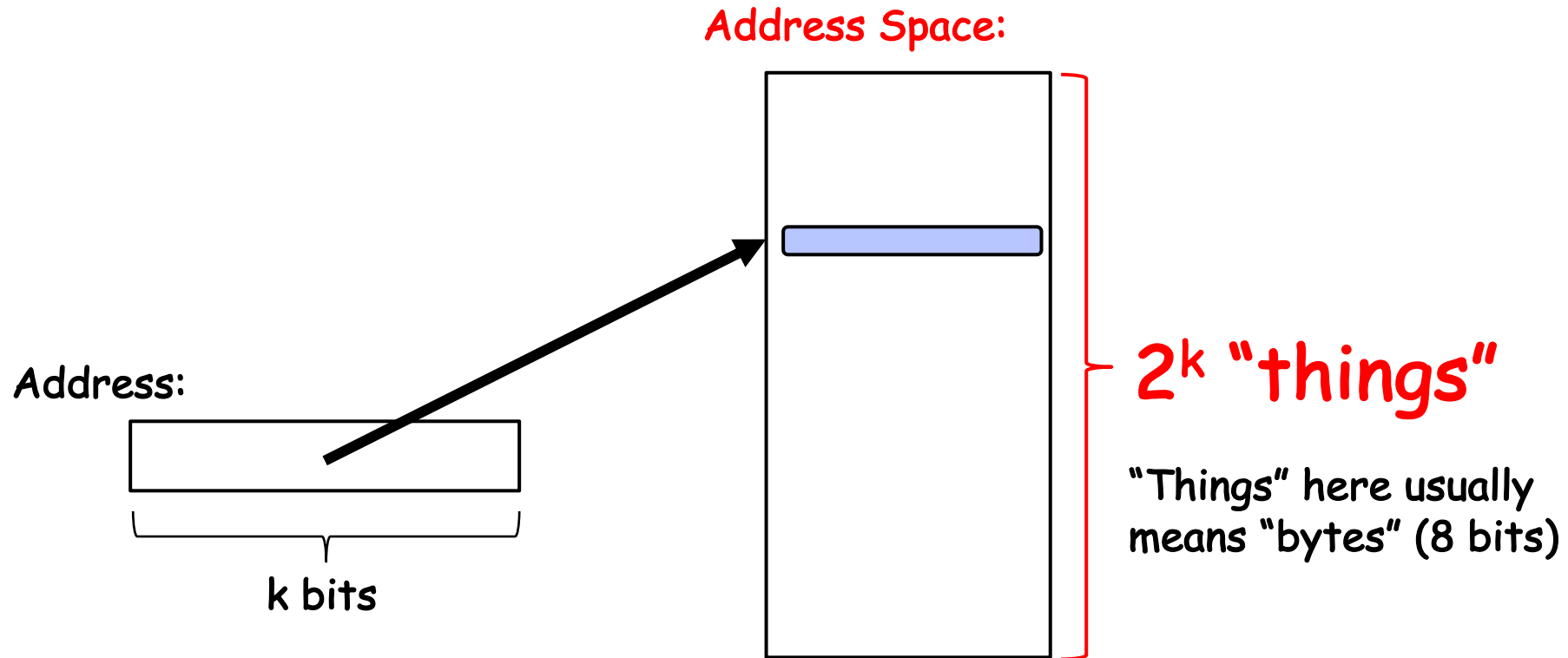


- Physical Reality:
Different Processes/Threads share the same hardware
 - Need to multiplex CPU (just finished: scheduling)
 - Need to multiplex use of memory (starting today)
 - Need to multiplex disk and devices (later in term)
- Why worry about memory sharing?
 - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
 - Consequently, cannot just let different threads of control use the same memory
 - » Physics: two different pieces of data cannot occupy the same locations in memory
 - Probably don't want different threads to even have access to each other's memory if in different processes (protection)

Recall: Four Fundamental OS Concepts

- **Thread: Execution Context**
 - Fully describes program state
 - Program Counter, Registers, Execution Flags, Stack
- **Address space (with or w/o translation)**
 - Set of memory addresses accessible to program (for read or write)
 - May be distinct from memory space of the physical machine (in which case programs operate in a virtual address space)
- **Process: an instance of a running program**
 - Protected Address Space + One or more Threads
- **Dual mode operation / Protection**
 - Only the “system” has the ability to access certain resources
 - Combined with translation, isolates programs from each other and the OS from programs

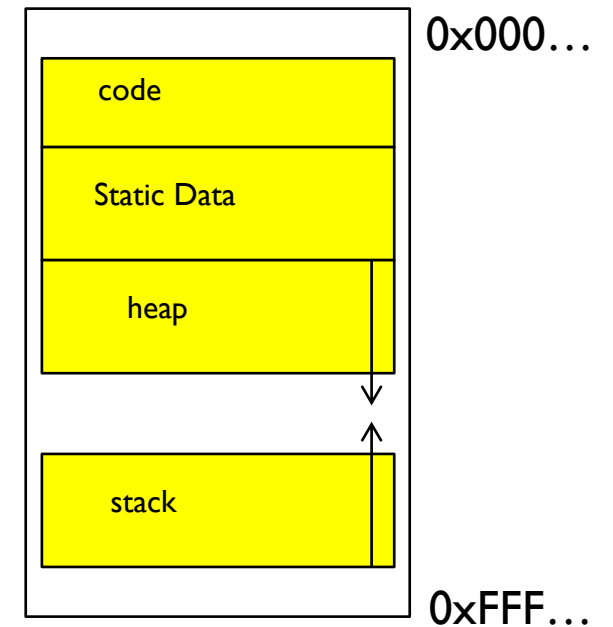
THE BASICS: Address/Address Space



- What is 2^{10} bytes (where a byte is abbreviated as "B")?
 - $2^{10} \text{ B} = 1024\text{B} = 1 \text{ KB}$ (for memory, $1\text{K} = 1024$, *not* 1000)
- How many bits to address each byte of 4KB page?
 - $4\text{KB} = 4 \times 1\text{KB} = 4 \times 2^{10} = 2^{12} \Rightarrow 12$ bits
- How much memory can be addressed with 20 bits? 32 bits? 64 bits?
 - Use 2^k

Process Virtual Address Space

- Definition: **Set of accessible addresses and the state associated with them**
 - $2^{32} = \sim 4$ billion **bytes** on a 32-bit machine
- How many 32-bit numbers fit in this address space?
 - 32-bits = 4 bytes, so $2^{32}/4 = 2^{30} = \sim 1$ billion
- What happens when processor reads or writes to an address?
 - Perhaps acts like regular memory
 - Perhaps causes I/O operation
 - » (Memory-mapped I/O)
 - Causes program to abort (segfault)?
 - Communicate with another program
 - ...



Important Aspects of Memory Multiplexing

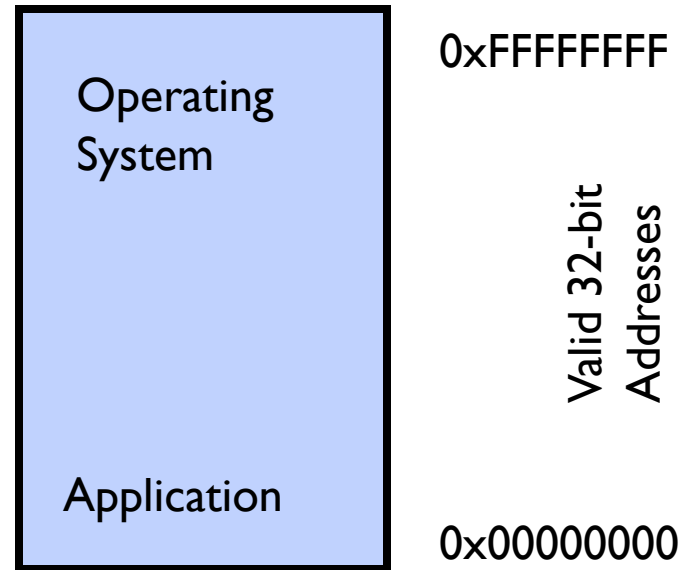
- **Protection:**
 - Prevent access to private memory of other processes
 - » Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc.)
 - » Kernel data protected from User programs
 - » Programs protected from themselves
- **Translation:**
 - Ability to translate accesses from one address space (virtual) to a different one (physical)
 - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
 - Side effects:
 - » Can be used to avoid overlap
 - » Can be used to give uniform view of memory to programs
- **Controlled overlap:**
 - Separate state of threads should not collide in physical memory. Obviously, unexpected overlap causes chaos!
 - Conversely, would like the ability to overlap when desired (for communication)

Alternative View: Interposing on Process Behavior

- OS interposes on process' I/O operations
 - How? All I/O happens via syscalls.
- OS interposes on process' CPU usage
 - How? Interrupt lets OS preempt current thread
- **Question: How can the OS interpose on process' memory accesses?**
 - Too slow for the OS to interpose *every* memory access
 - Translation: hardware support to accelerate the common case
 - Page fault: uncommon cases trap to the OS to handle

Recall: Uniprogramming

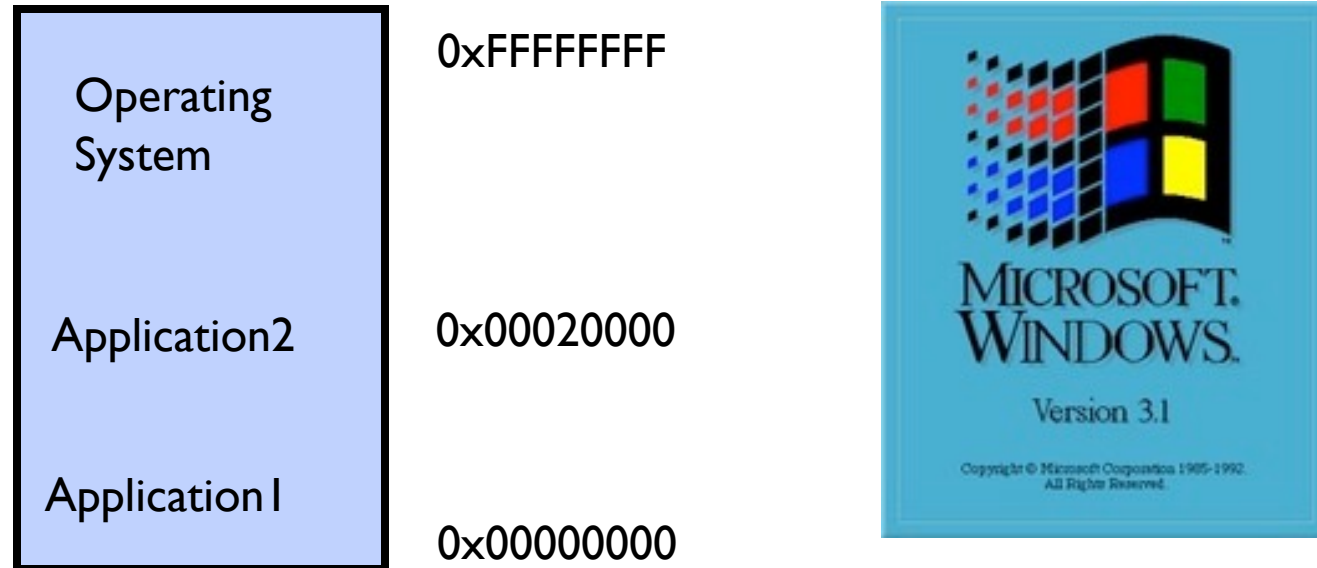
- Uniprogramming (no Translation or Protection)
 - Application always runs at same place in physical memory since only one application at a time
 - Application can access any physical address



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

Primitive Multiprogramming

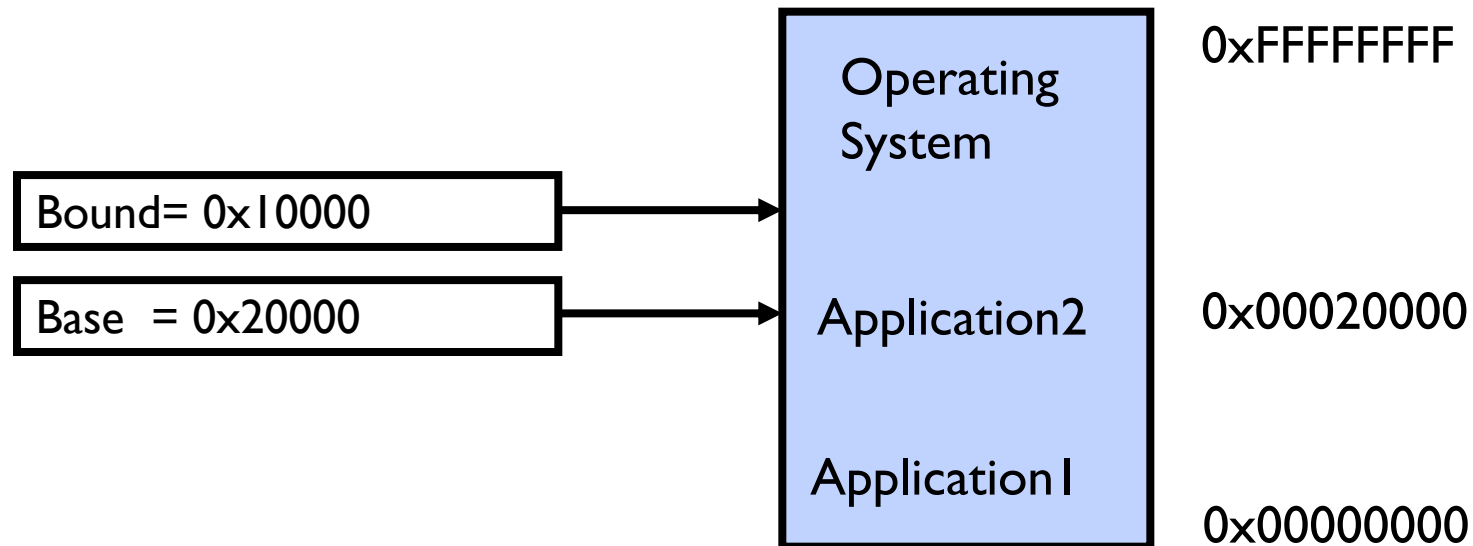
- Multiprogramming without Translation or Protection
 - Must somehow prevent address overlap between threads



- Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
 - » Everything adjusted to memory location of program
 - » Translation done by a linker-loader (relocation)
 - » Common in early days (... till Windows 3.x, 95?)
- With this solution, no protection: bugs in any program can cause other programs to crash or even the OS

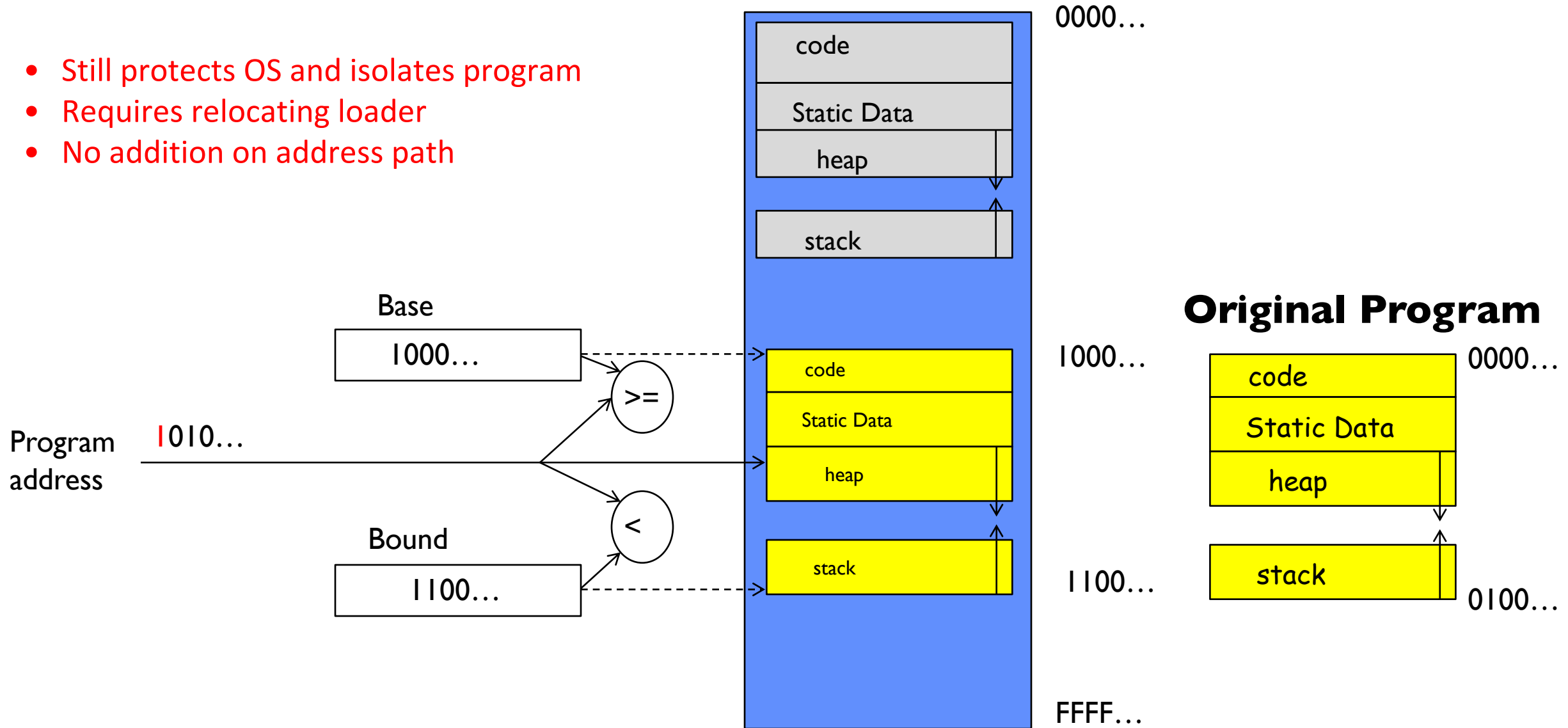
Multiprogramming with Protection

- Can we protect programs from each other without translation?
 - **Yes: Base and Bound!**
 - **Used by, e.g., Cray-1 supercomputer**

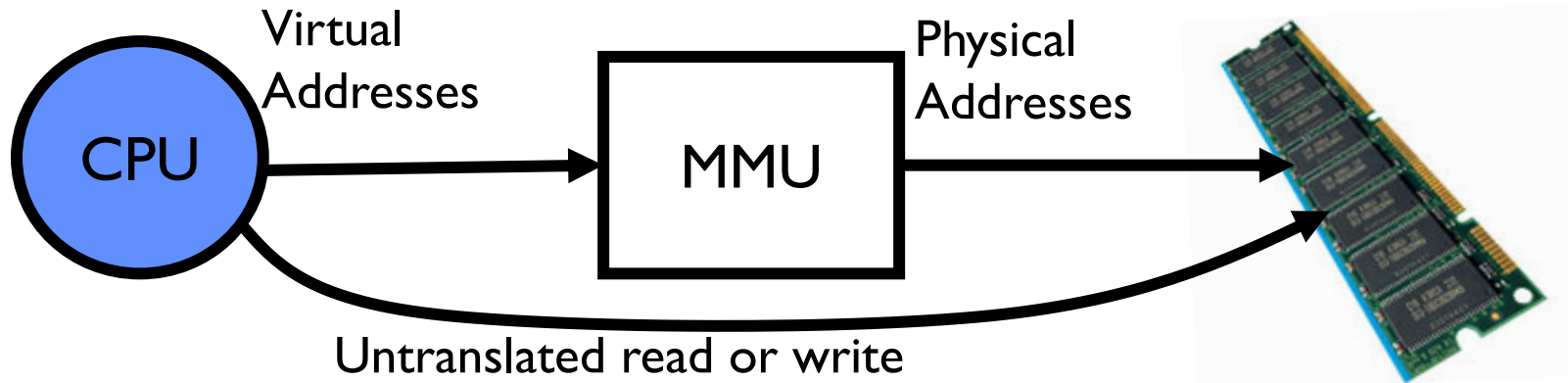


Recall: Base and Bound (No Translation)

- Still protects OS and isolates program
- Requires relocating loader
- No addition on address path

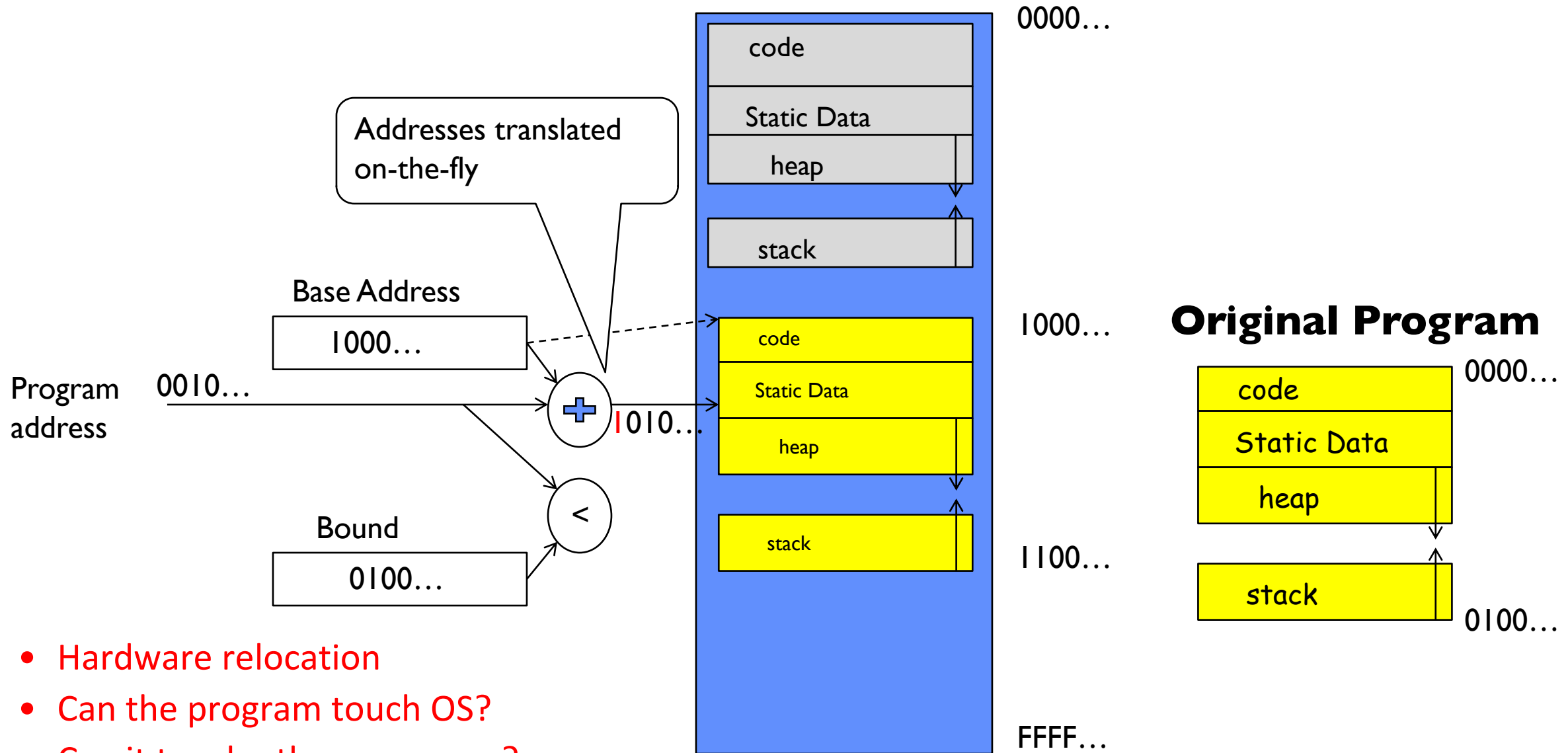


Recall: General Address translation



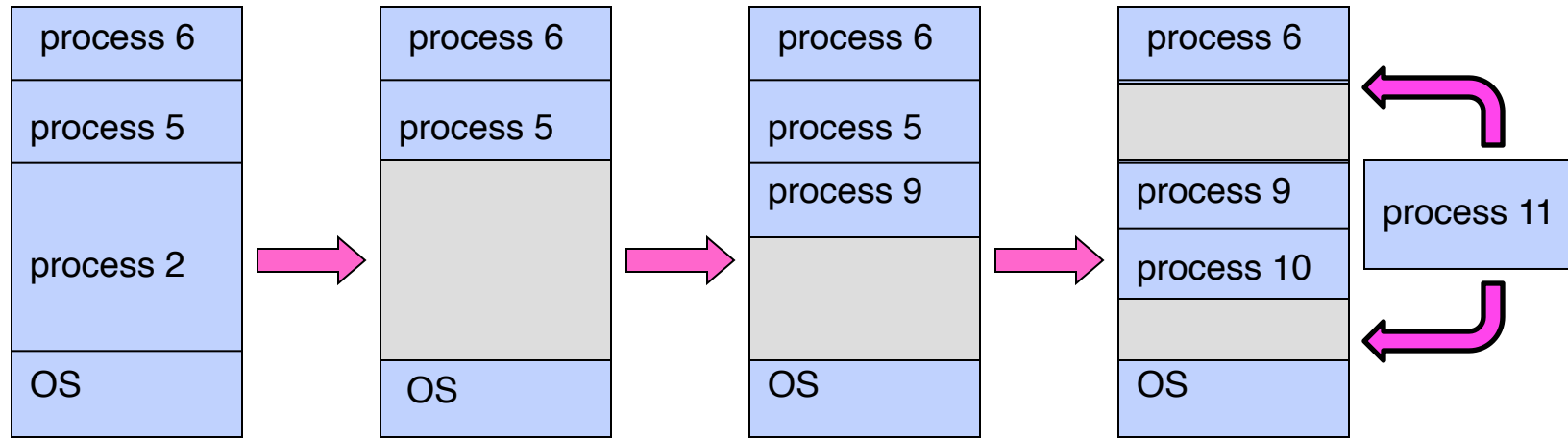
- Consequently, two views of memory:
 - View from the CPU (what program sees, virtual memory)
 - View from memory (physical memory)
 - **Translation box** (Memory Management Unit or MMU) converts between the two views
- **Translation \Rightarrow much easier to implement protection!**
 - If task A cannot even gain access to task B's data, no way for A to adversely affect B
- With translation, every program can be linked/loaded into same region of user address space

Recall: Base and Bound (with Translation)



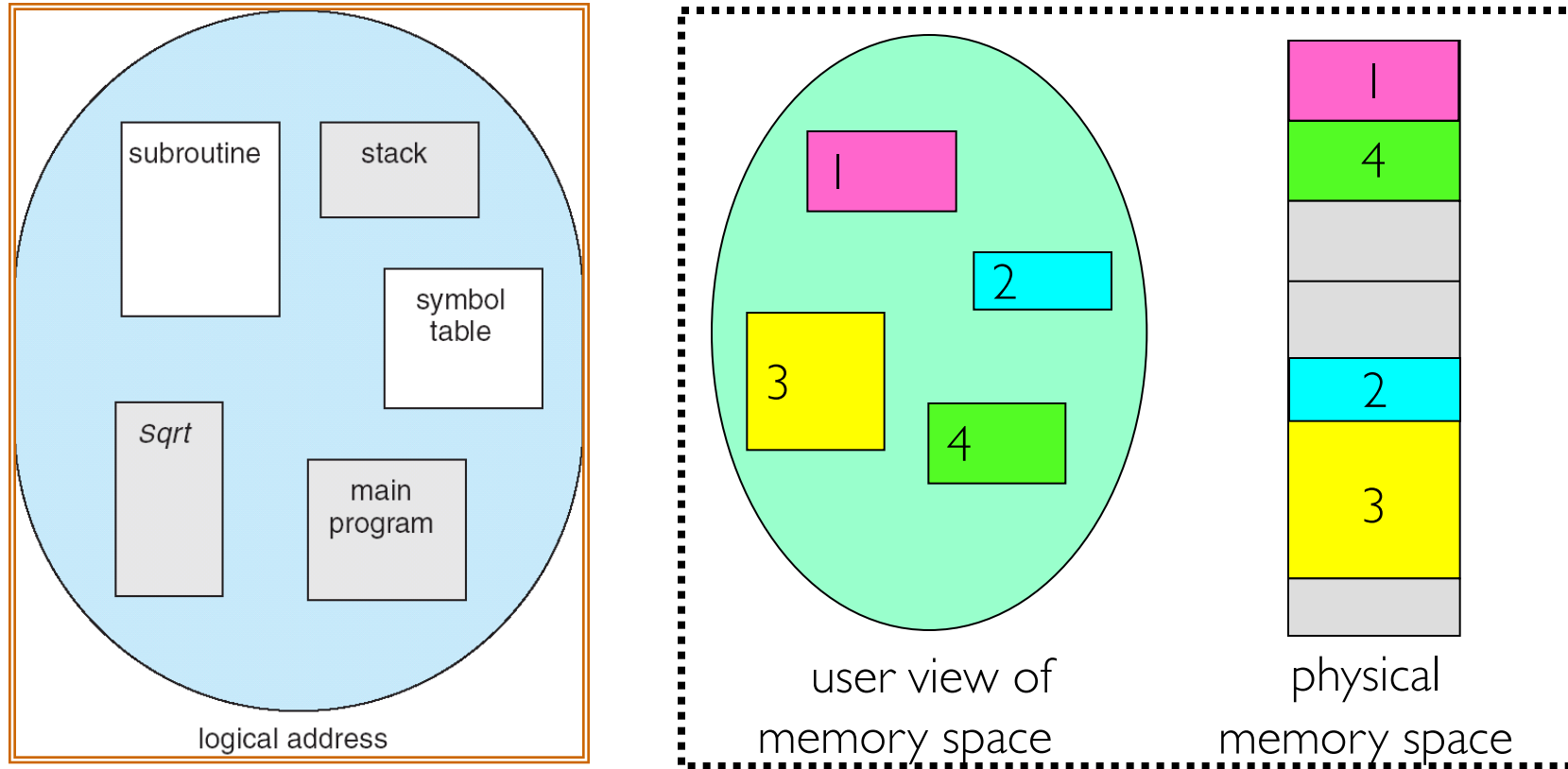
- Hardware relocation
- Can the program touch OS?
- Can it touch other programs?

Issues with Simple B&B Method



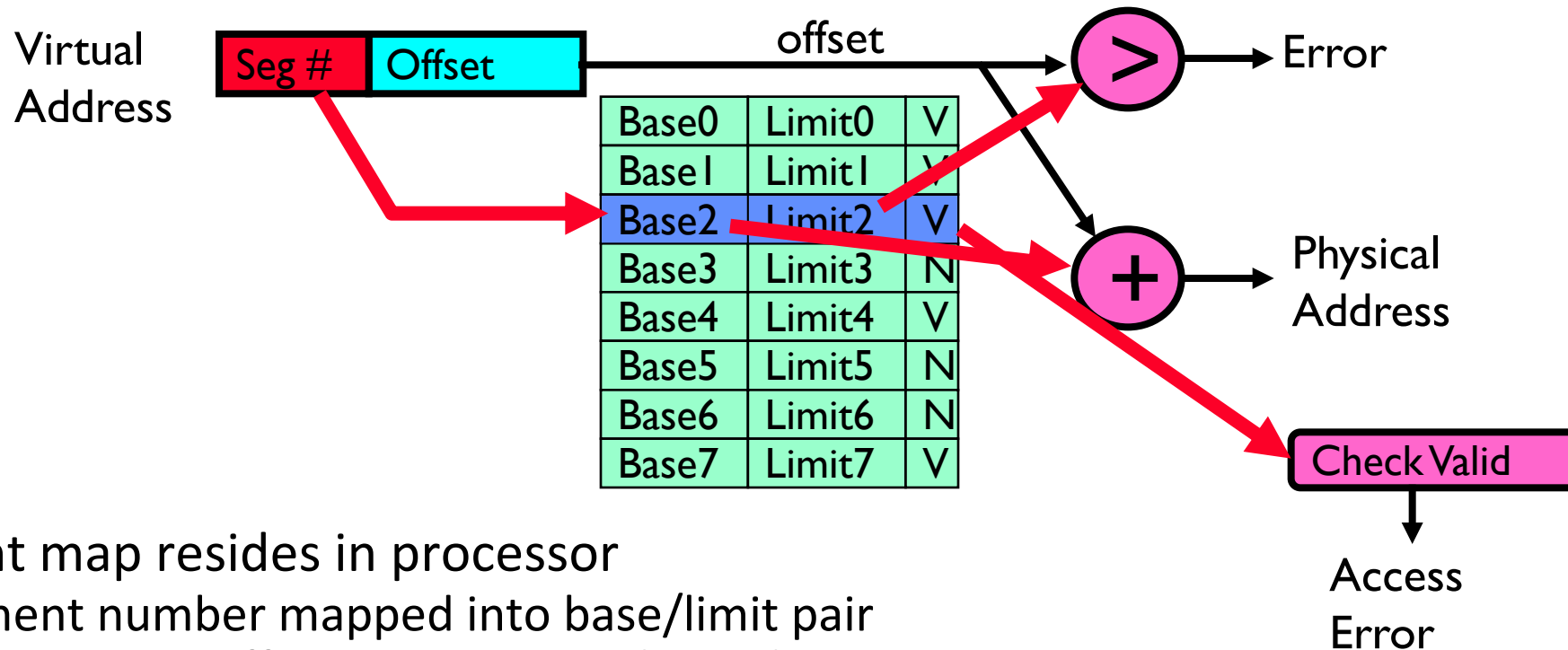
- Fragmentation problem over time
 - Not every process is same size \Rightarrow memory becomes fragmented over time
- Missing support for sparse address space
 - Would like to have multiple chunks/program (Code, Data, Stack, Heap, etc)
- Hard to do inter-process sharing
 - Want to share code segments when possible
 - Want to share memory between processes
 - Helped by providing multiple segments per process

More Flexible Segmentation



- Logical View: multiple separate segments
 - Typical: Code, Data, Stack
 - Others: memory sharing, etc
- Each segment is given region of contiguous memory
 - Has a base and limit
 - Can reside anywhere in physical memory

Implementation of Multi-Segment Model



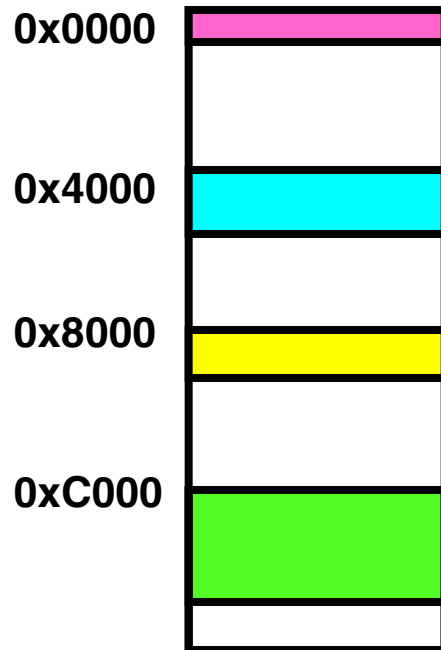
- Segment map resides in processor
 - Segment number mapped into base/limit pair
 - Base added to offset to generate physical address
 - Error check catches offset out of range
- As many chunks of physical memory as entries
 - Segment addressed by portion of virtual address
 - However, could be included in instruction instead:
 - » x86 Example: `mov [es:bx],ax.`
- What is “V/N” (valid / not valid)?
 - Can mark segments as invalid; requires check as well

Example: Four Segments (16 bit addresses)

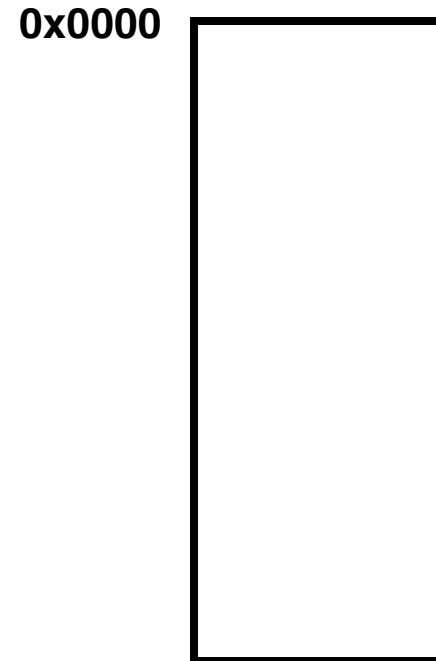


Virtual Address Format

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



Virtual Address Space



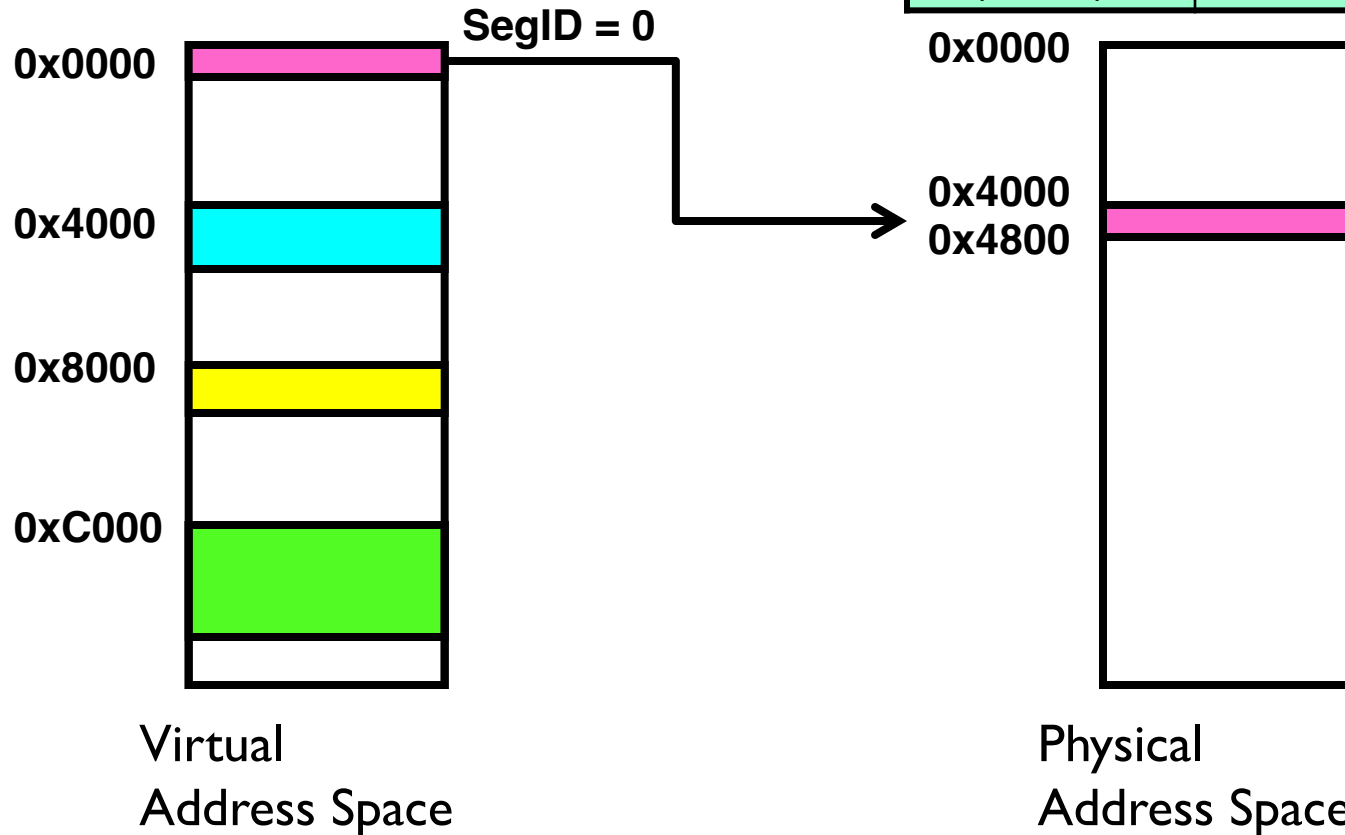
Physical Address Space

Example: Four Segments (16 bit addresses)



Virtual Address Format

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

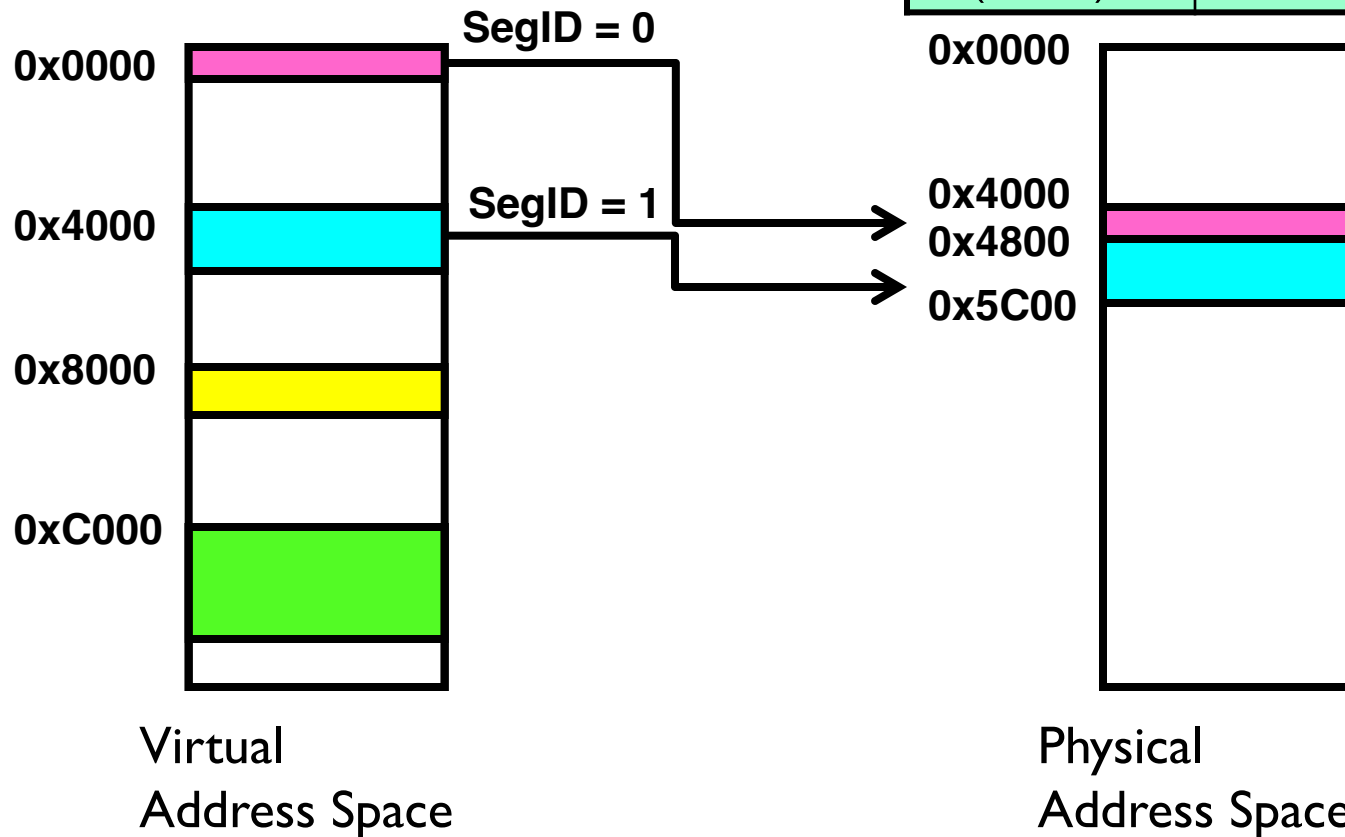


Example: Four Segments (16 bit addresses)



Virtual Address Format

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

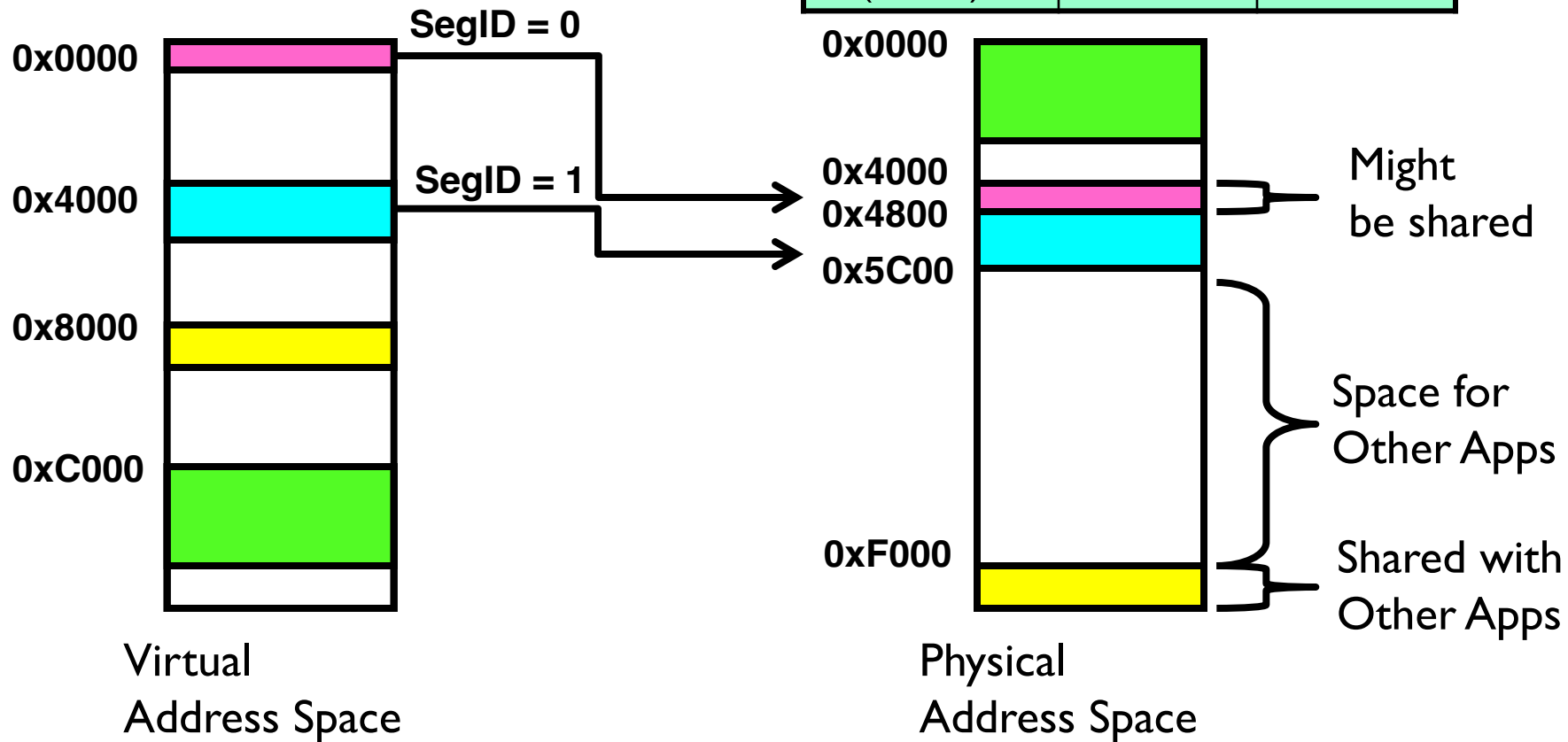


Example: Four Segments (16 bit addresses)



Virtual Address Format

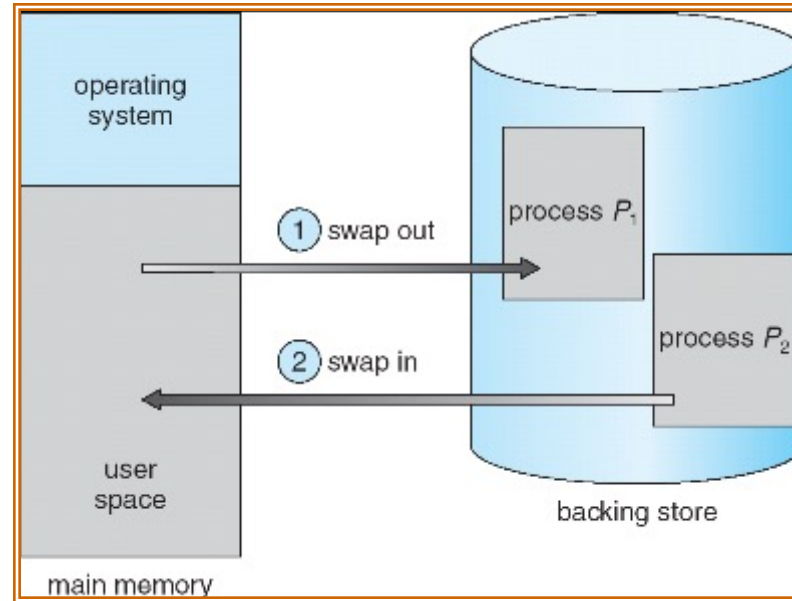
Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



Observations about Segmentation

- Translation on every instruction fetch, load or store
- Virtual address space has holes
 - Segmentation efficient for sparse address spaces
- When it is OK to address outside valid range?
 - This is how the stack (and heap?) allowed to grow
 - For instance, stack takes fault, system automatically increases size of stack
- Need protection mode in segment table
 - For example, code segment would be read-only
 - Data and stack would be read-write (stores allowed)
- What must be saved/restored on context switch?
 - Segment table stored in CPU, not in memory (small)
 - Might store all of process memory onto disk when switched (called “swapping”)

What if not all segments fit in memory?



- Extreme form of Context Switch: **Swapping**
 - To make room for next process, some or all of the previous process is moved to disk
 - » Likely need to send out complete segments
 - This greatly increases the cost of context-switching
- What might be a desirable alternative?
 - **Some way to keep only active portions of a process in memory at any one time**
 - Need finer granularity control over physical memory

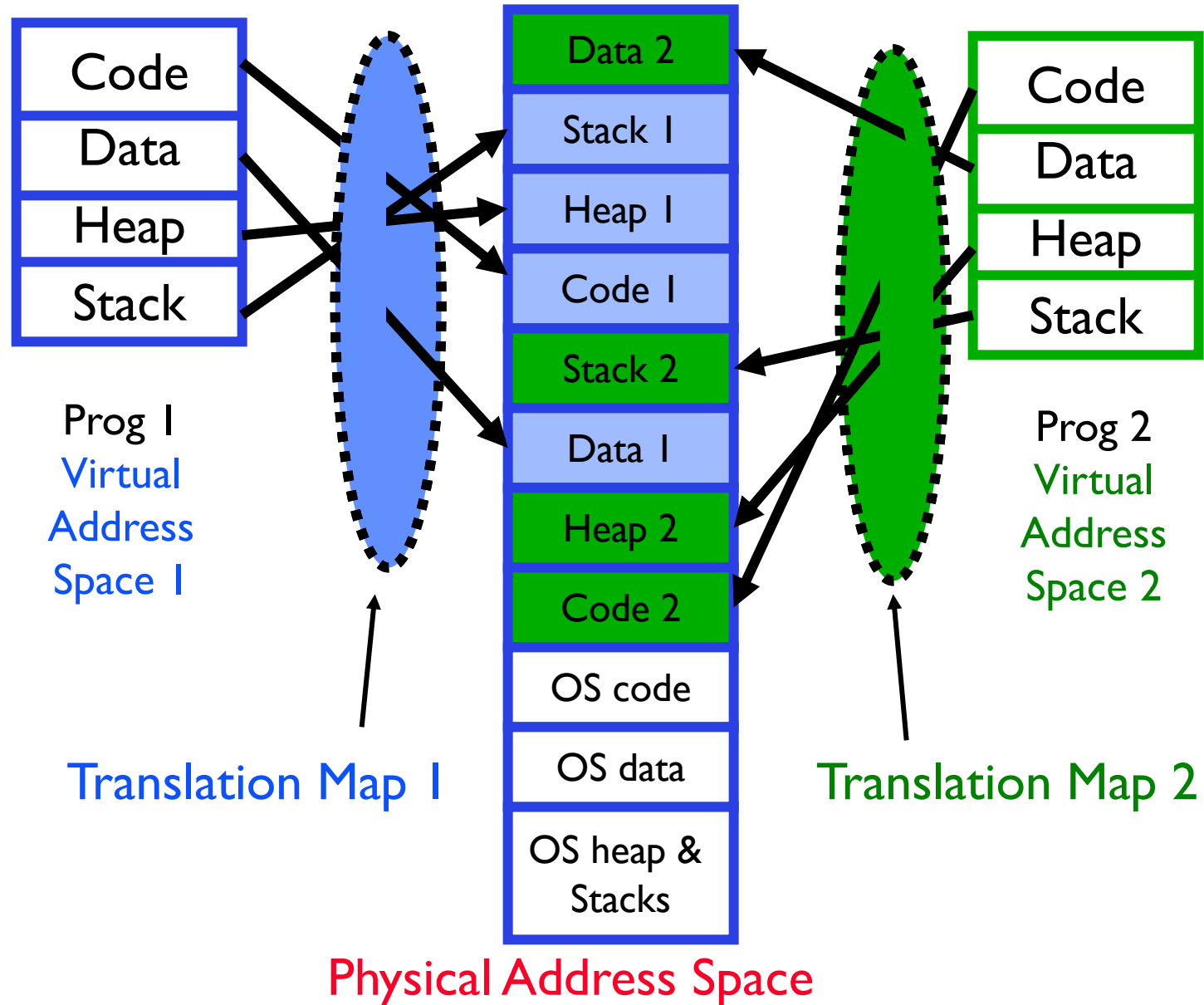
Group Discussion

- Topic: segmentation
 - What are the problems of segmentation?
- Discuss in groups of two to three students
 - Each group chooses a leader to summarize the discussion
 - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

Problems with Segmentation

- Must fit variable-sized chunks into physical memory
- May move processes multiple times to fit everything
- Limited options for swapping to disk
- **Fragmentation**: wasted space
 - **External**: free gaps between allocated chunks
 - **Internal**: don't need all memory within allocated chunks

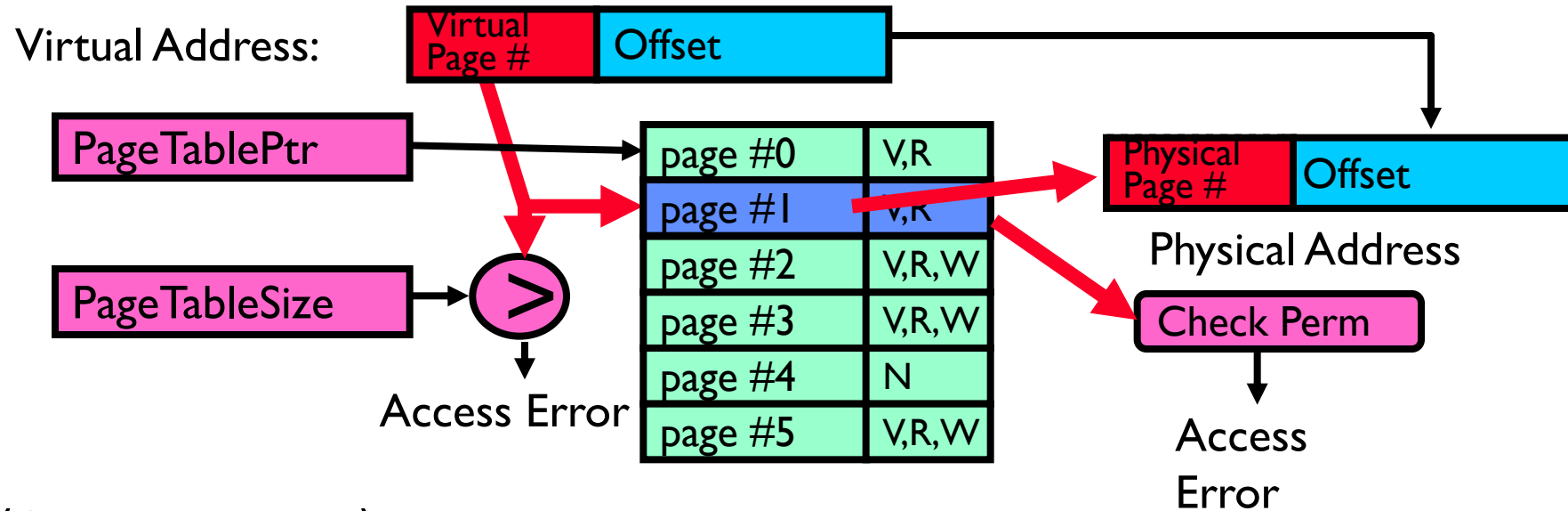
General Address Translation



Paging: Physical Memory in Fixed Size Chunks

- Solution to fragmentation from segments?
 - Allocate physical memory in **fixed size** chunks (“pages”)
 - Every chunk of physical memory is equivalent
 - » Can use simple vector of bits to handle allocation:
00110001110001101 ... 110010
 - » Each bit represents page of physical memory
1 \Rightarrow allocated, 0 \Rightarrow free
- Should pages be as big as our previous segments?
 - No: Can lead to lots of internal fragmentation
 - » Typically have small pages (1K-16K)
 - Consequently: need multiple pages/segment

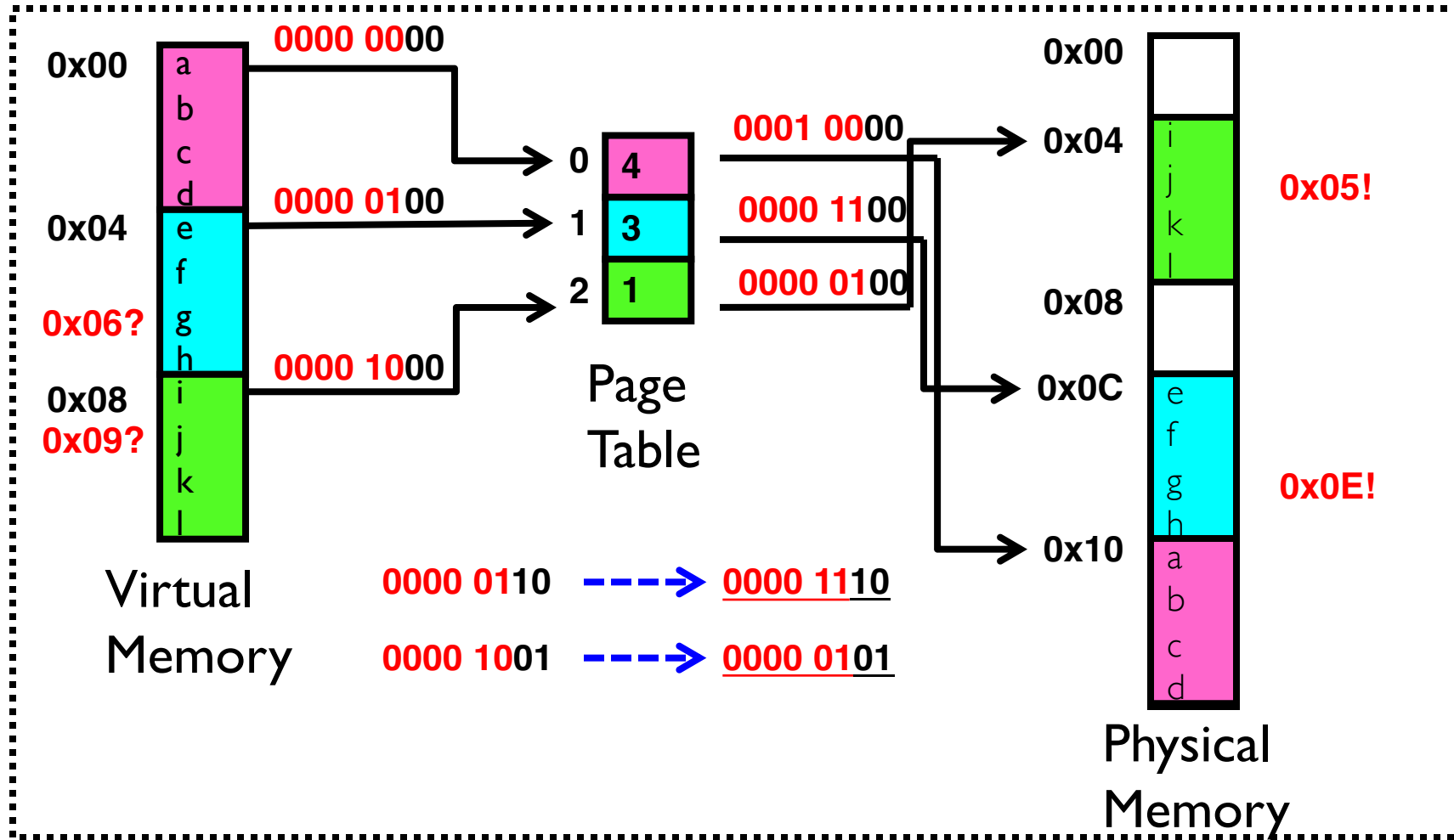
How to Implement Simple Paging?



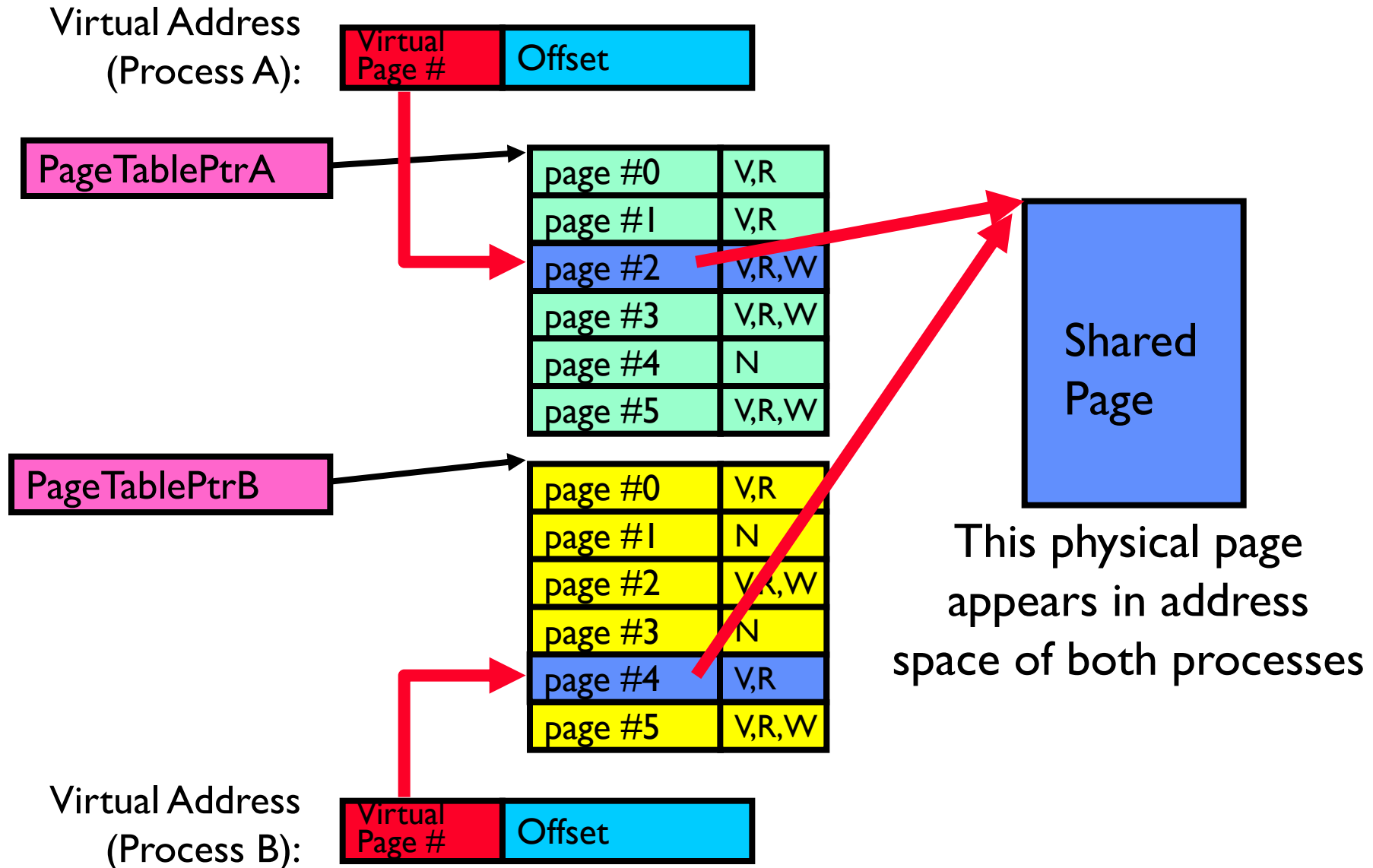
- Page Table (One per process)
 - Resides in physical memory
 - Contains physical page and permission for each virtual page (e.g. Valid bits, Read, Write, etc.)
- Virtual address mapping
 - Offset from Virtual address copied to Physical Address
 - » Example: 10 bit offset \Rightarrow 1024-byte pages
 - Virtual page # is all remaining bits
 - » Example for 32-bits: $32 - 10 = 22$ bits, i.e. 4 million entries
 - » Physical page # copied from table into physical address
 - Check Page Table bounds and permissions

Simple Page Table Example

Example (4 byte pages)



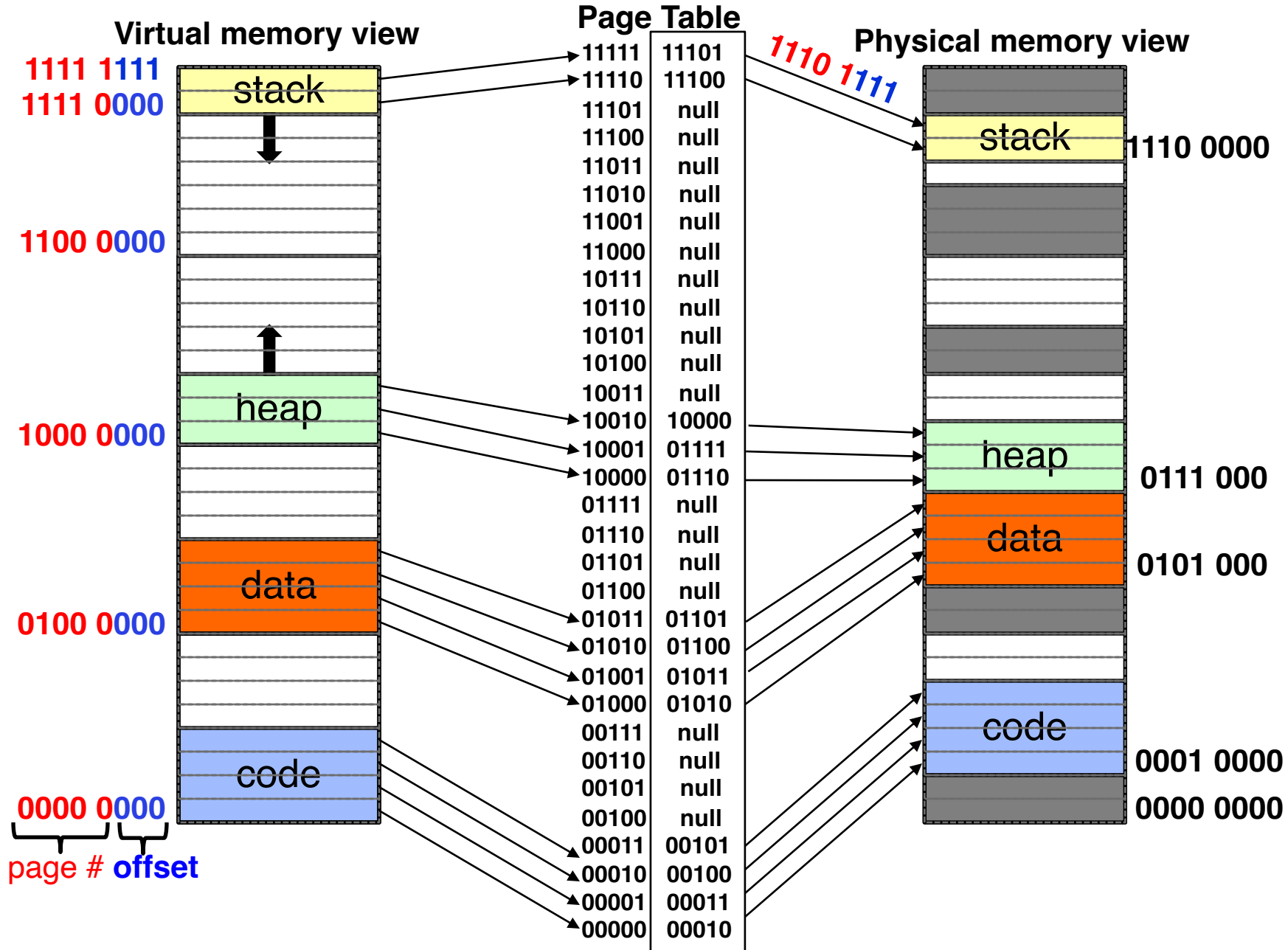
What about Sharing?



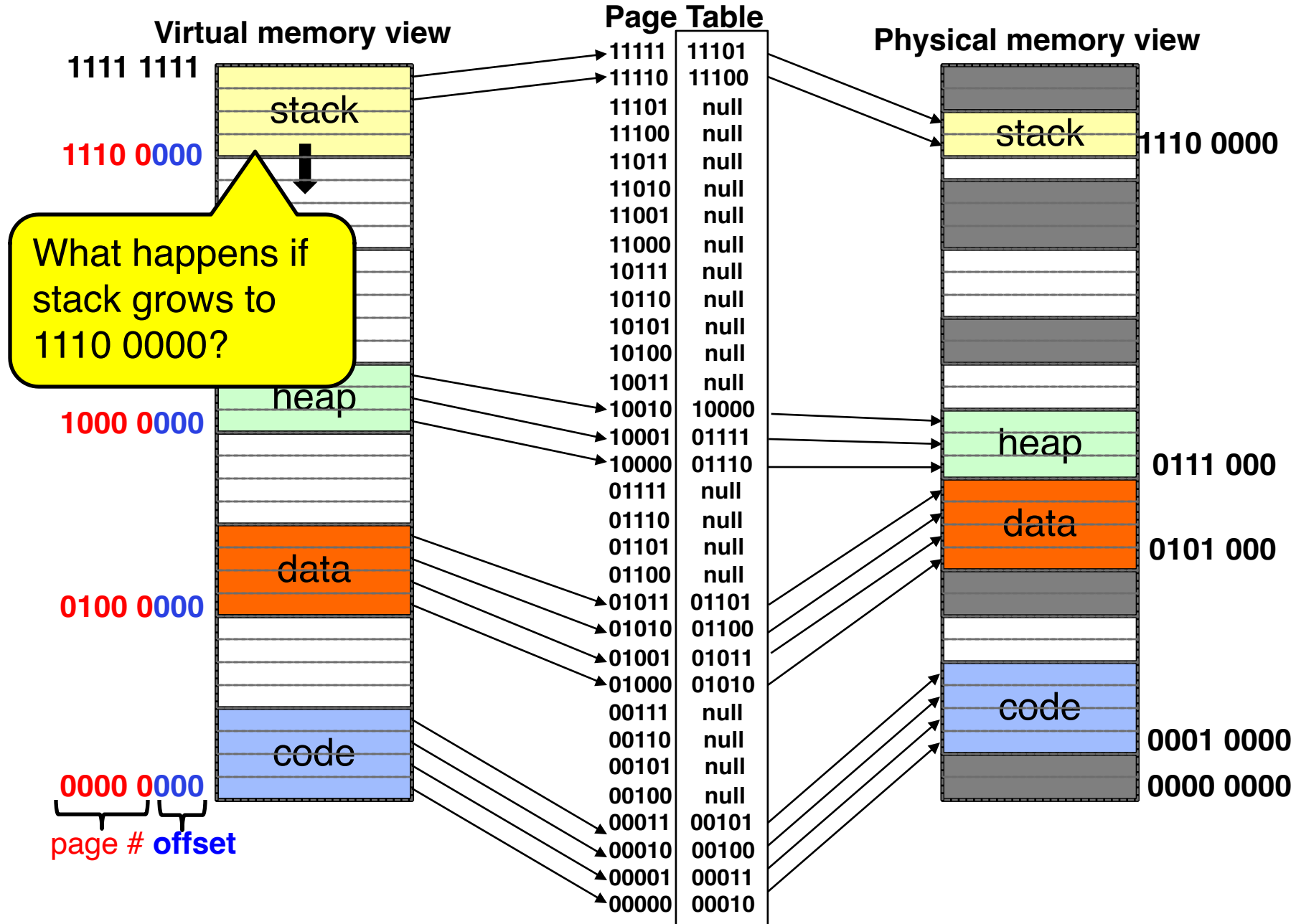
Where is page sharing used ?

- The “kernel region” of every process has the same page table entries
 - The process cannot access it at user level
 - But on U->K switch, kernel code can access it AS WELL AS the region for THIS user
- Different processes running same binary!
 - Execute-only, but do not need to duplicate code segments
- User-level system libraries (execute only)
- Shared-memory segments between different processes
 - Can actually share objects directly between processes
 - » Must map page into same place in address space!
 - This is a limited form of the sharing that threads have within a single process

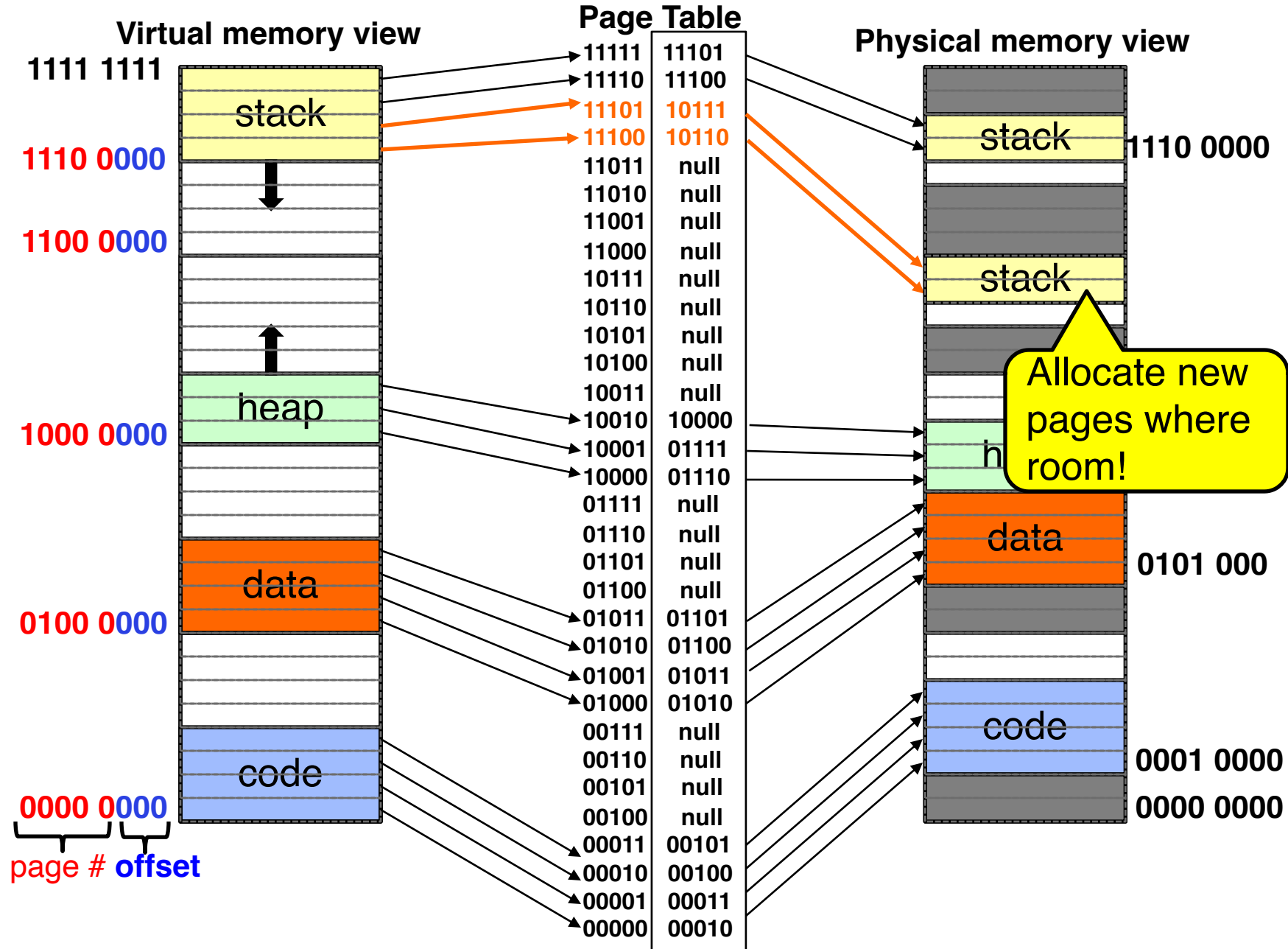
Summary: Paging



Summary: Paging



Summary: Paging



Summary

- Segment Mapping
 - Segment registers within processor
 - Segment ID associated with each access
 - » Often comes from portion of virtual address
 - » Can come from bits in instruction instead (x86)
 - Each segment contains base and limit information
 - » Offset (rest of address) adjusted by adding base
- Page Tables
 - Memory divided into fixed-sized chunks of memory
 - Virtual page number from virtual address mapped through page table to physical page number
 - Offset of virtual address same as physical address
 - Large page table
- Next Time: Multi-Level Tables
 - Virtual address mapped to series of tables
 - Permit sparse population of address space