Operating Systems
(Honor Track)

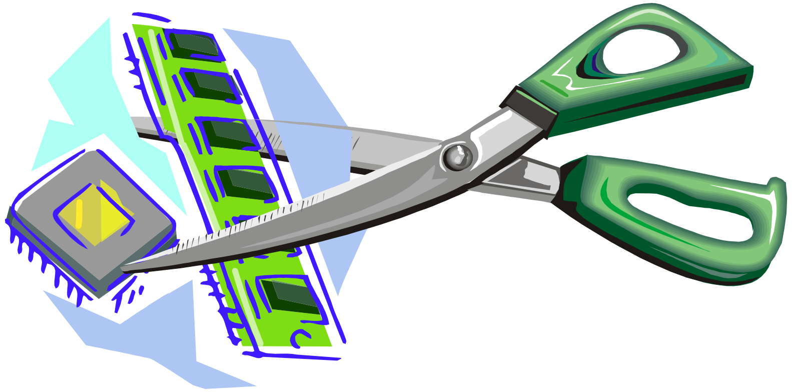
Memory 1: Address Translation and Virtual Memory

Xin Jin

Spring 2022

Acknowledgments: Ion Stoica, Berkeley CS 162

Virtualizing Resources

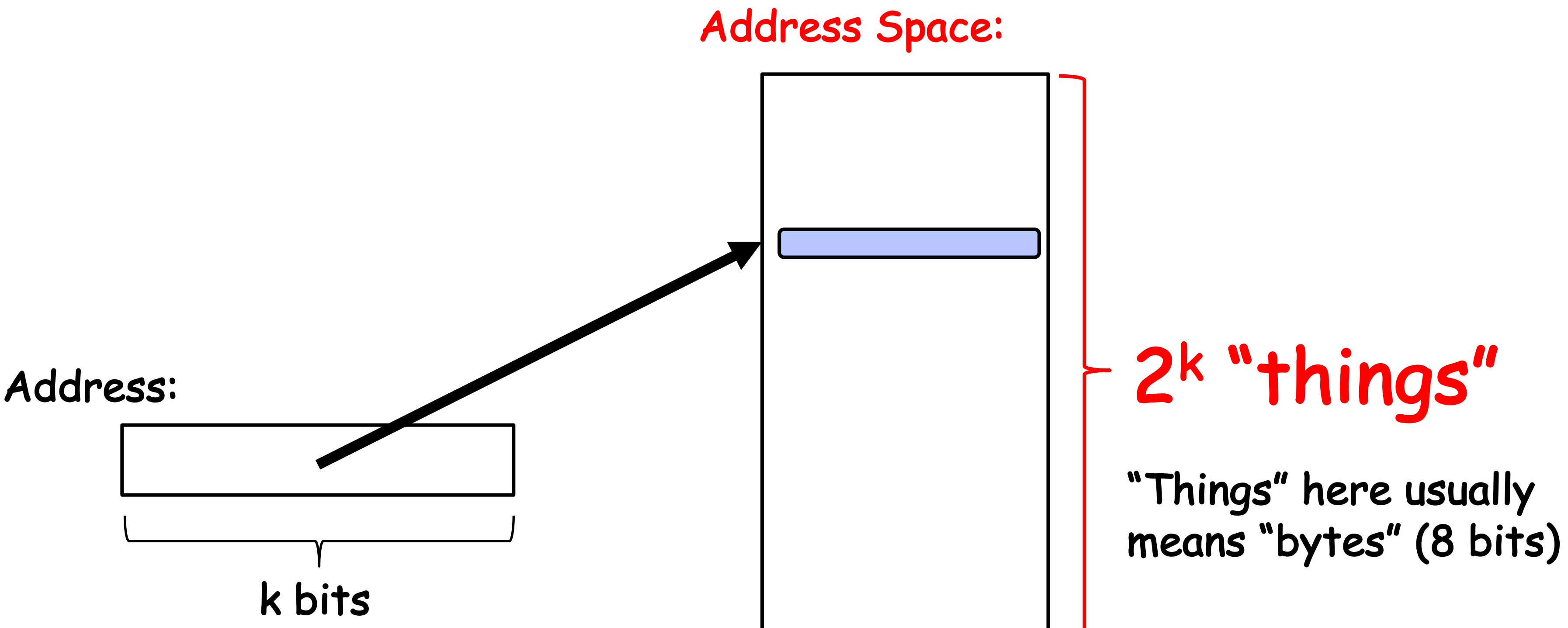


- Physical Reality:
  Different Processes/Threads share the same hardware
  - Need to multiplex CPU (Just finished: scheduling)
  - Need to multiplex use of Memory (starting today)
  - Need to multiplex disk and devices (later in term)
- Why worry about memory sharing?
  - The complete working state 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

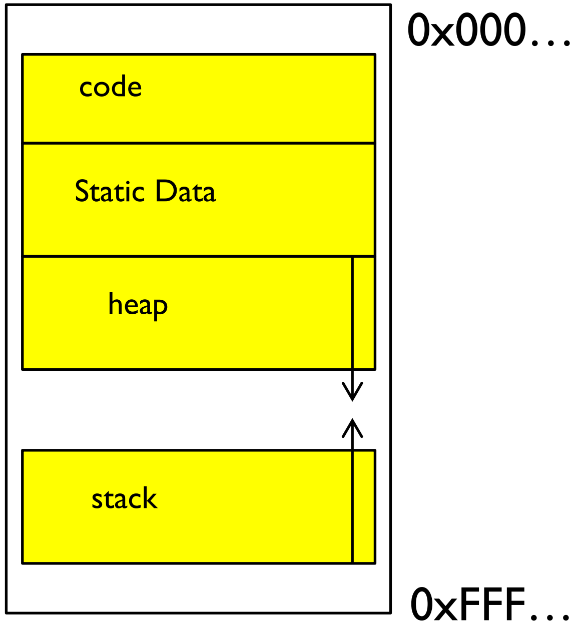
THE BASICS: Address/Address Space



- What is 2^{10} bytes (where a byte is abbreviated as "B")?
  - 2^{10} B = 1024B = 1 KB (for memory, 1K = 1024, *not* 1000)
- How many bits to address each byte of 4KB page?
  - 4KB = 4×1KB = $4 \times 2^{10} = 2^{12} \Rightarrow$ 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} = ~4 billion ***bytes*** on a 32-bit machine
- How many 32-bit numbers fit in this address space?
  - 32-bits = 4 bytes, so $2^{32}/4 = 2^{30}$=~1billion
- 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

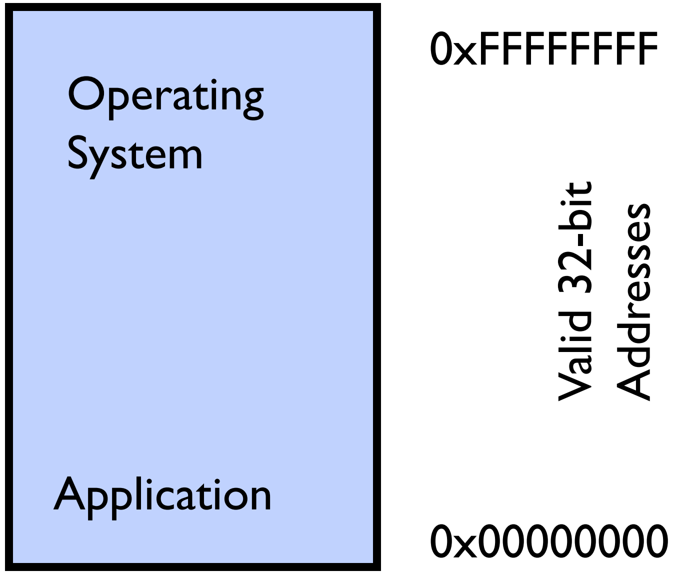
- OS interposes on process’ I/O operations
  - How? All I/O happens via syscalls.

- 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

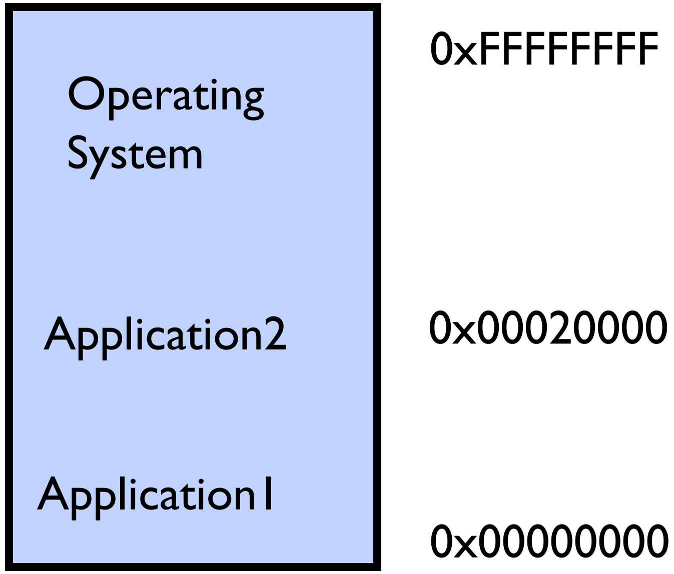
- Uniprogramming (no Translation or Protection)
  - Application always runs at same place in physical memory since only one application at a time
  - Application can access any physical address

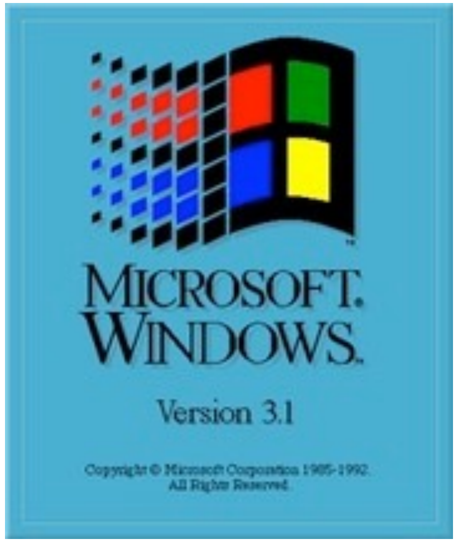


  - Application given illusion of dedicated machine by giving it reality of a dedicated machine

Primitive Multiprogramming

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

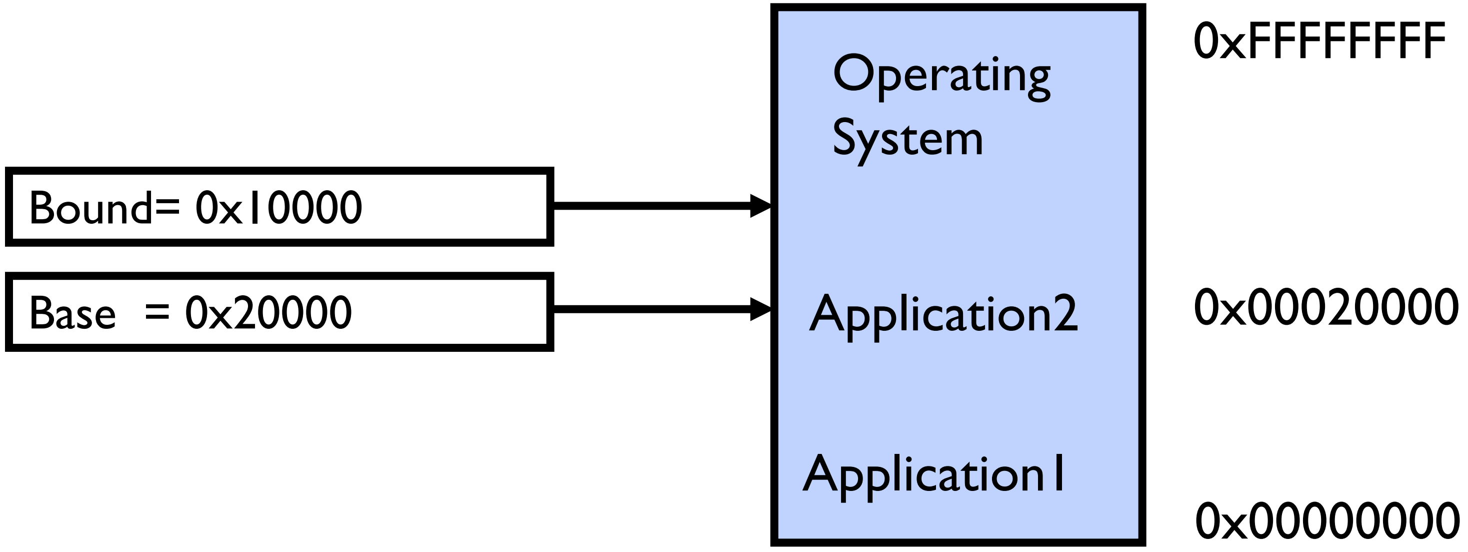




  - Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
    - » Everything adjusted to memory location of program
    - » Translation done by a linker-loader (relocation)
    - » Common in early days (… till Windows 3.x, 95?)
- With this solution, no protection: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

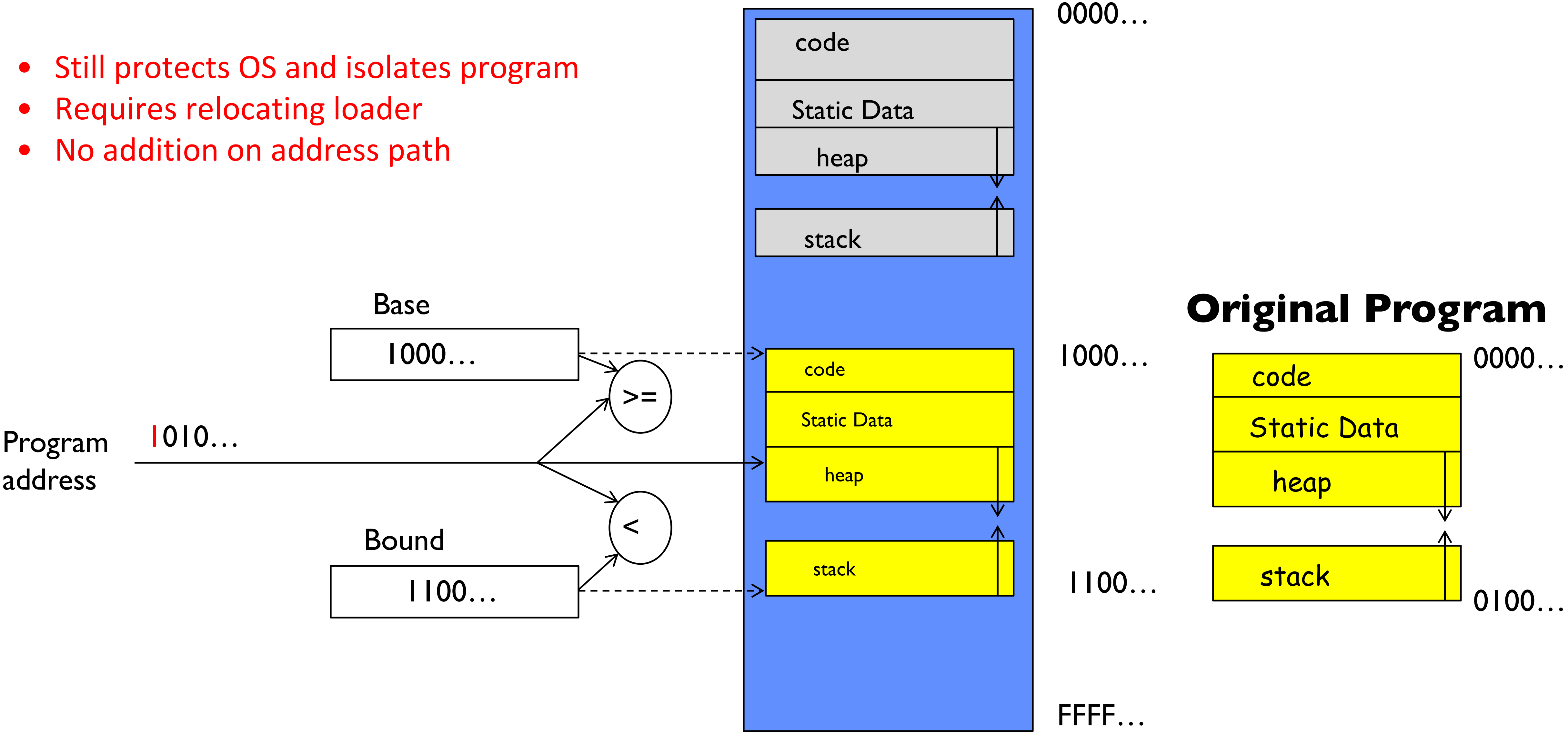
Multiprogramming with Protection

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

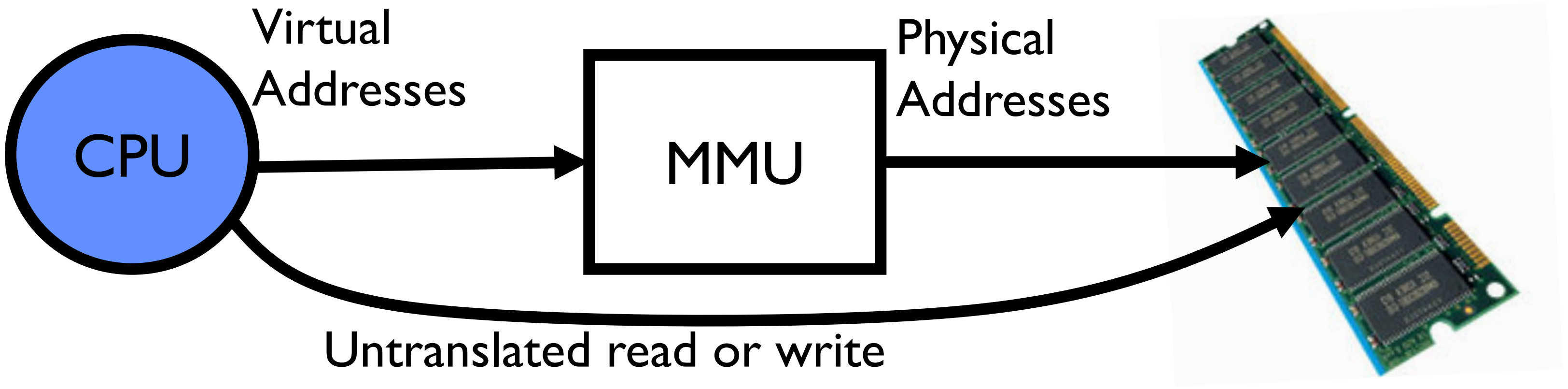


Recall: Base and Bound (No Translation)

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

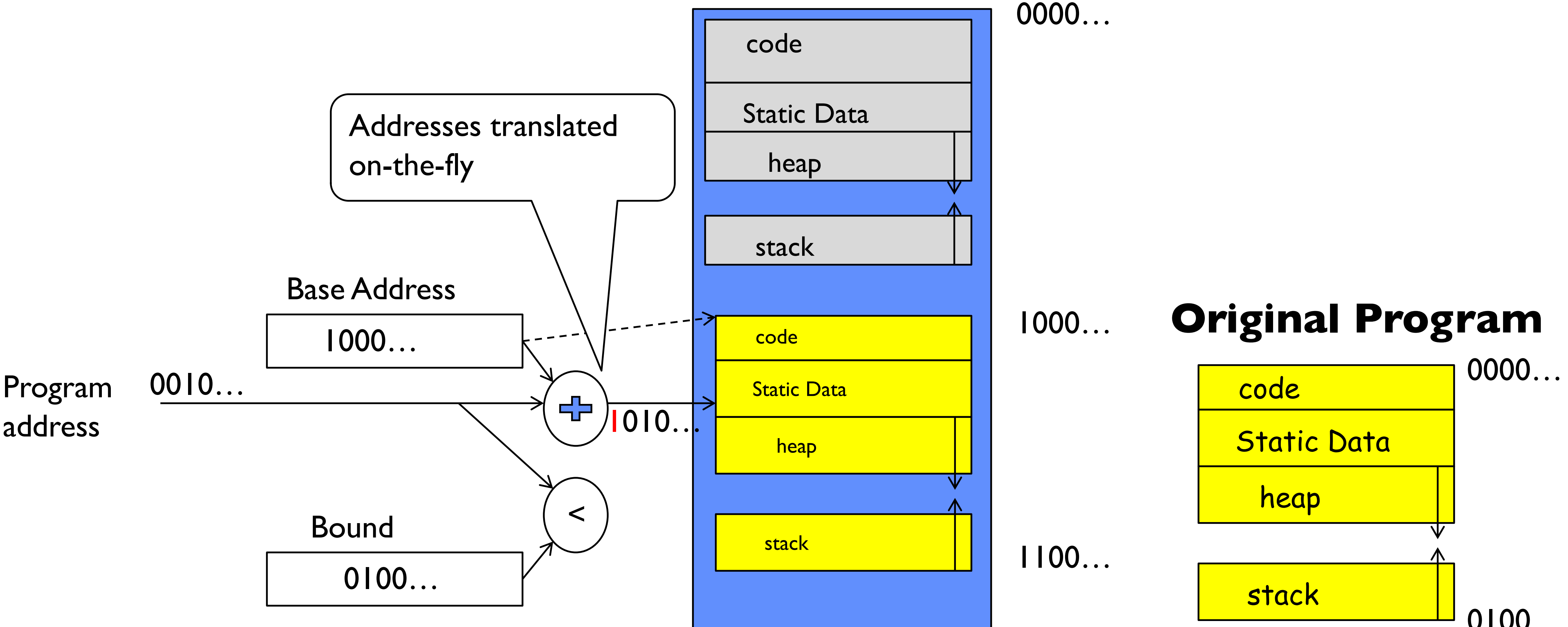


Recall: General Address translation



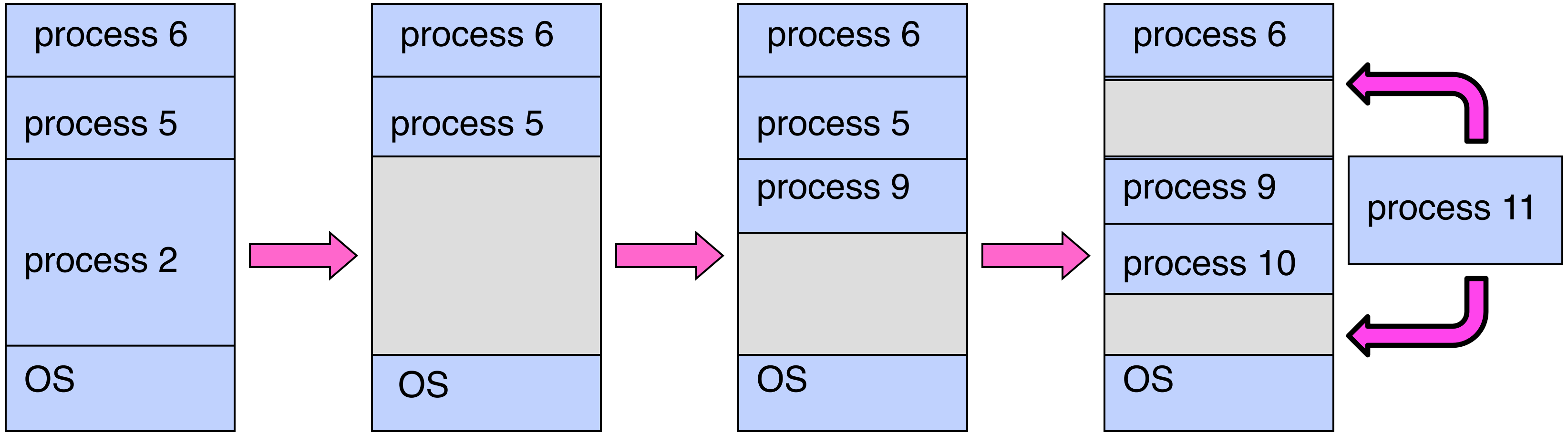
- Consequently, two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)
  - Translation box (Memory Management Unit or MMU) converts between the two views
- Translation ⇒ much easier to implement protection!
  - If task A cannot even gain 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)



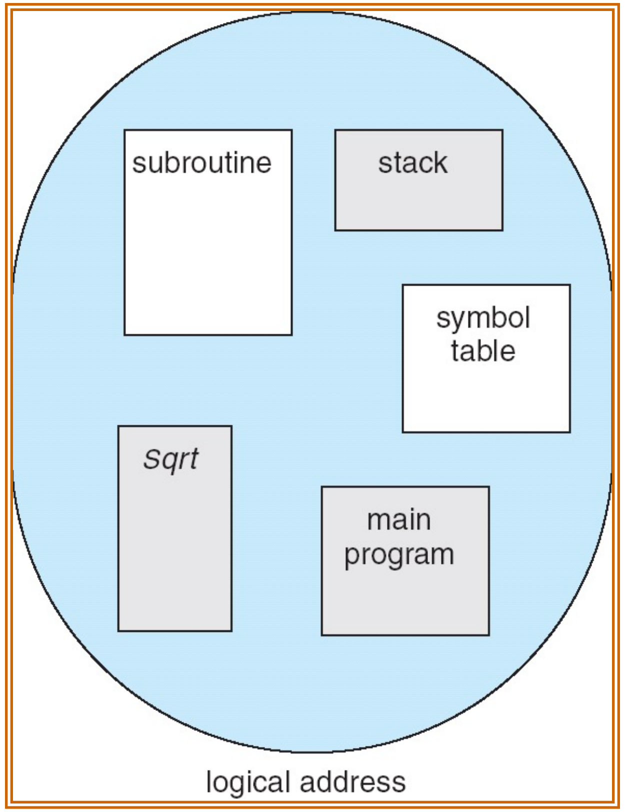
- Hardware relocation
- Can the program touch OS?
- Can it touch other programs?

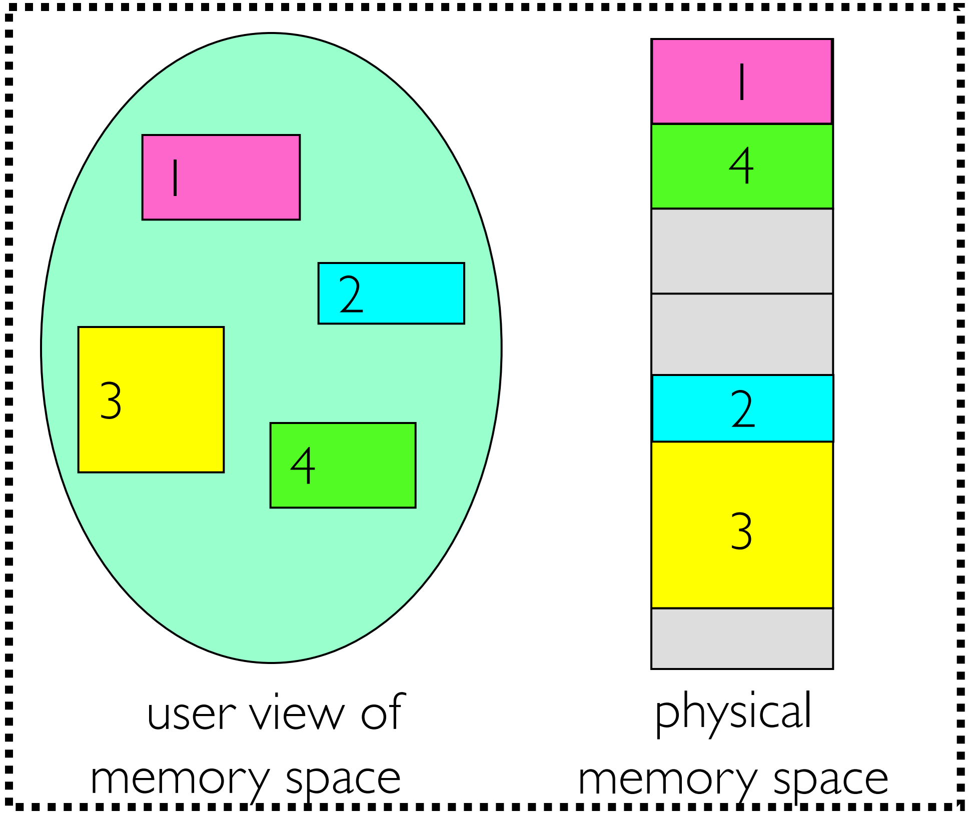
Issues with Simple B&B Method



- Fragmentation problem over time
  - Not every process is same size ⇒ 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

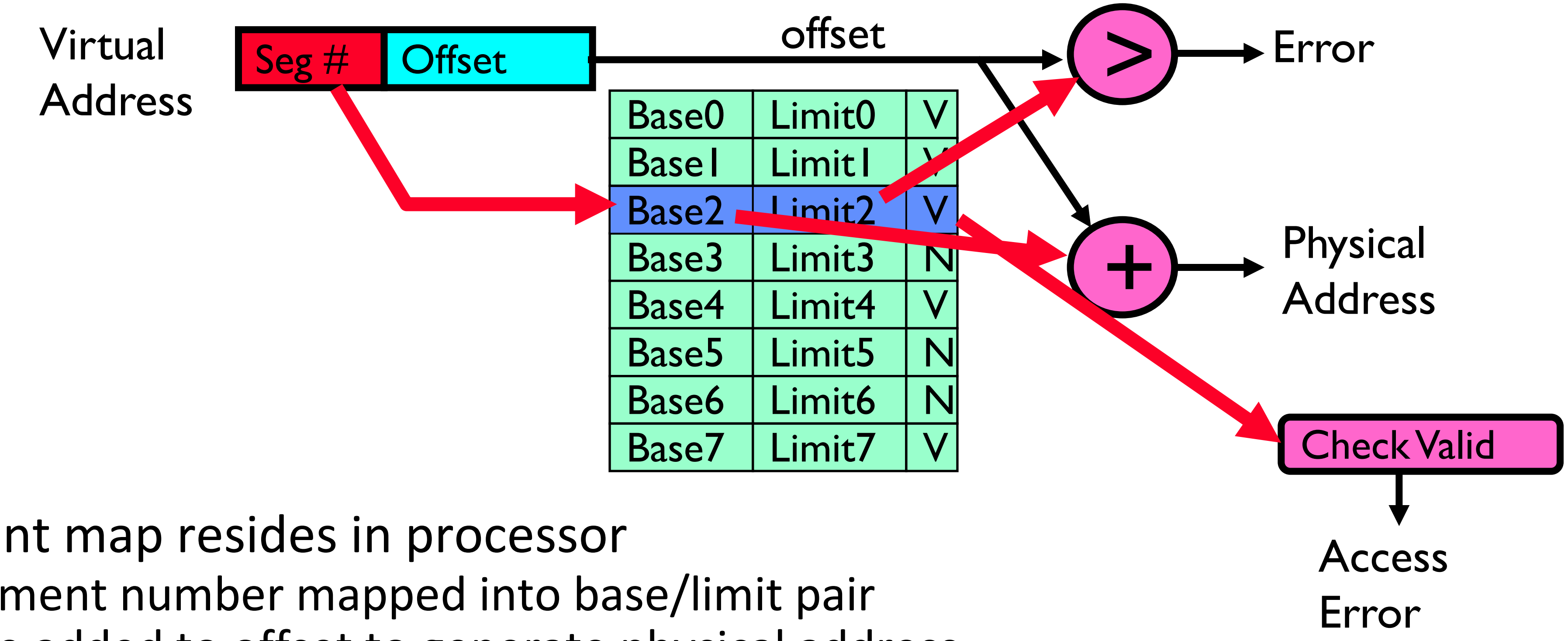
More Flexible Segmentation





- 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



Virtual Address

Seg # | Offset

offset

>

Error

+

Physical Address

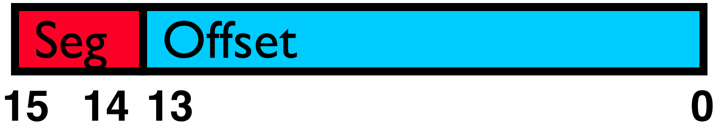
Check Valid

Access Error

Base0	Limit0	V
Base1	Limit1	V
Base2	Limit2	V
Base3	Limit3	N
Base4	Limit4	V
Base5	Limit5	N
Base6	Limit6	N
Base7	Limit7	V

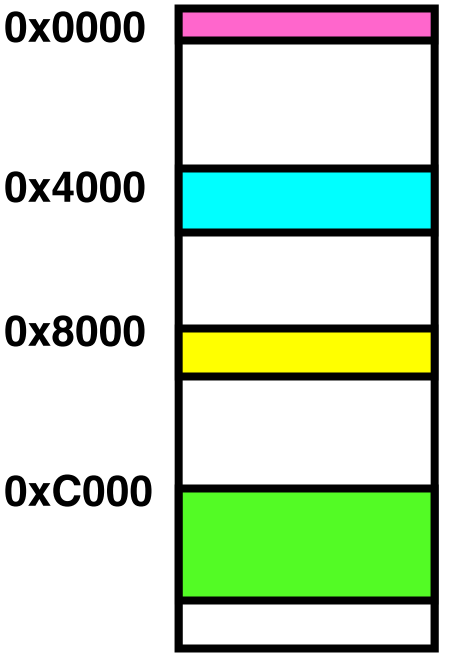
- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be 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)

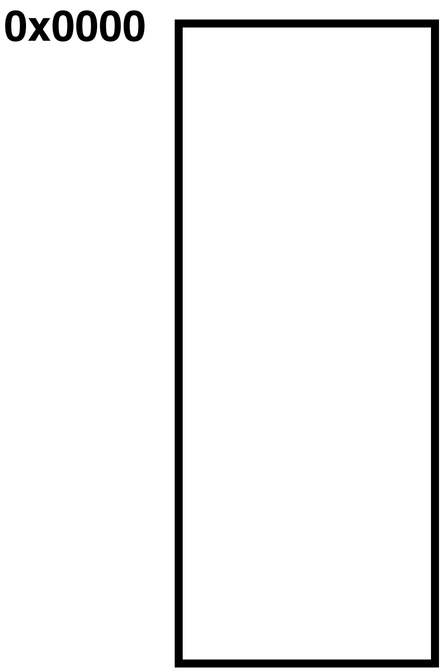


Virtual Address Format

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

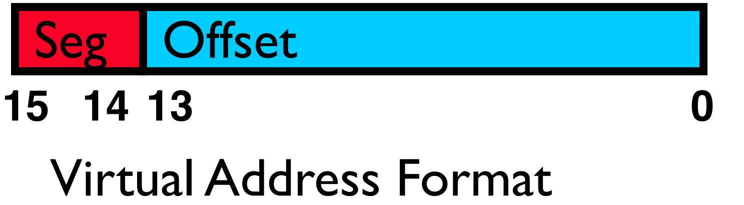


Virtual
Address Space

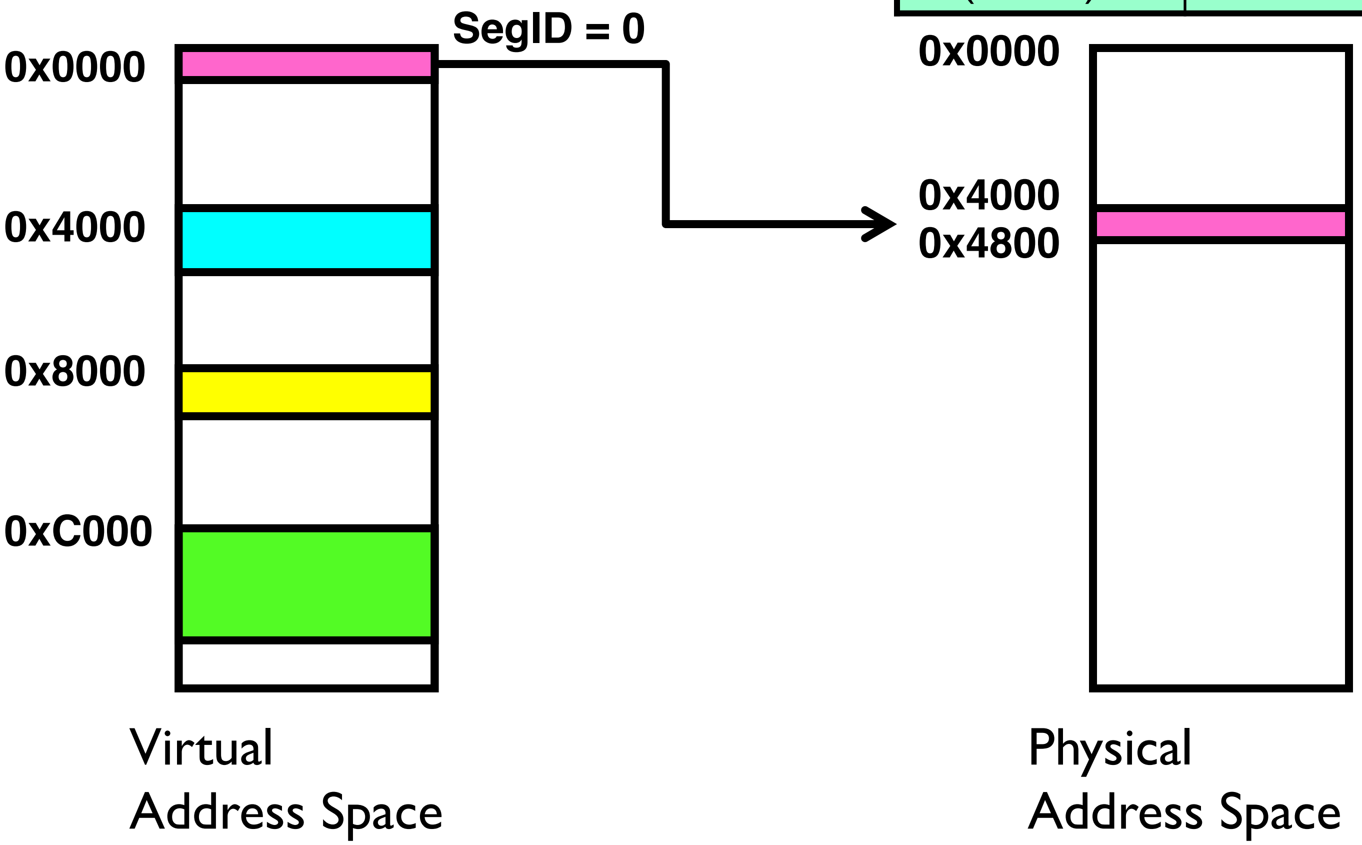


Physical
Address Space

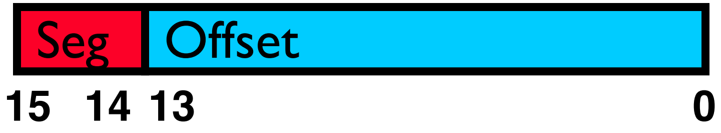
Example: Four Segments (16 bit addresses)



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

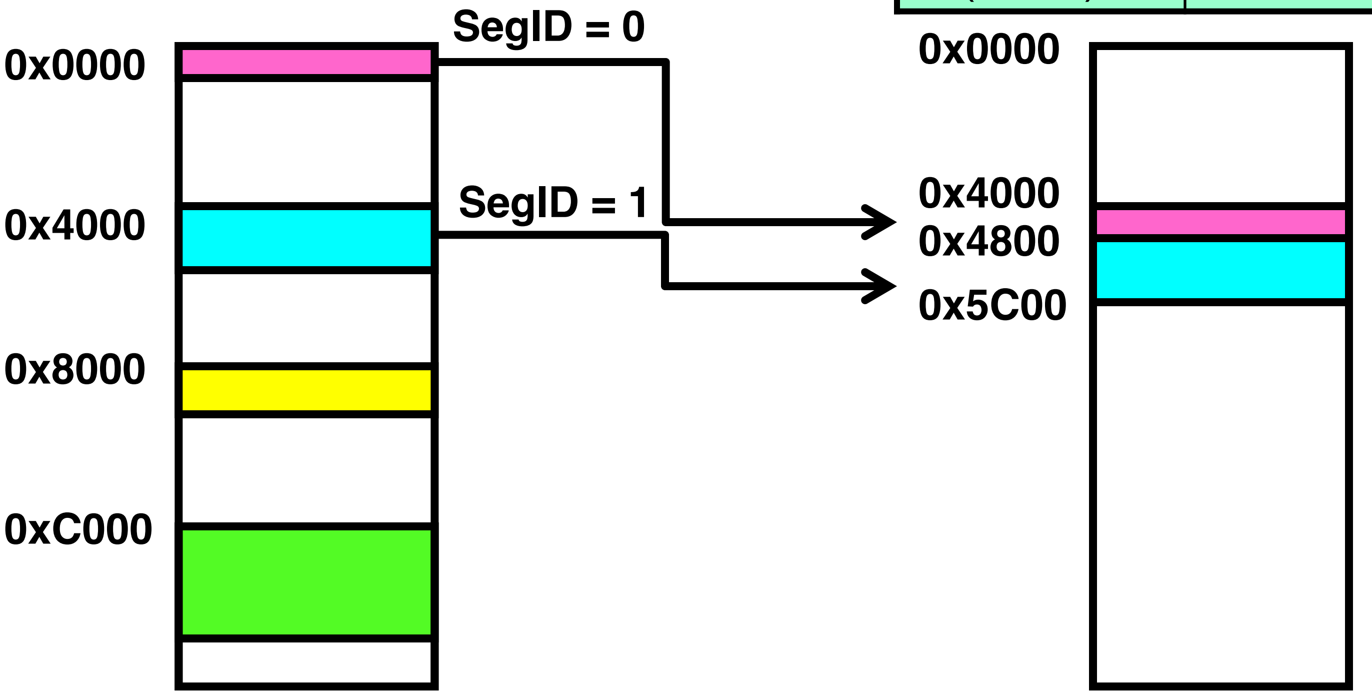


Example: Four Segments (16 bit addresses)



Virtual Address Format

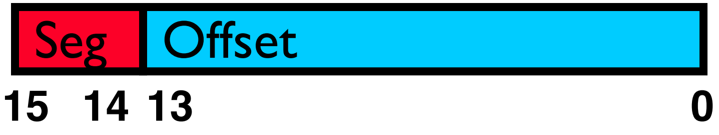
Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



Virtual
Address Space

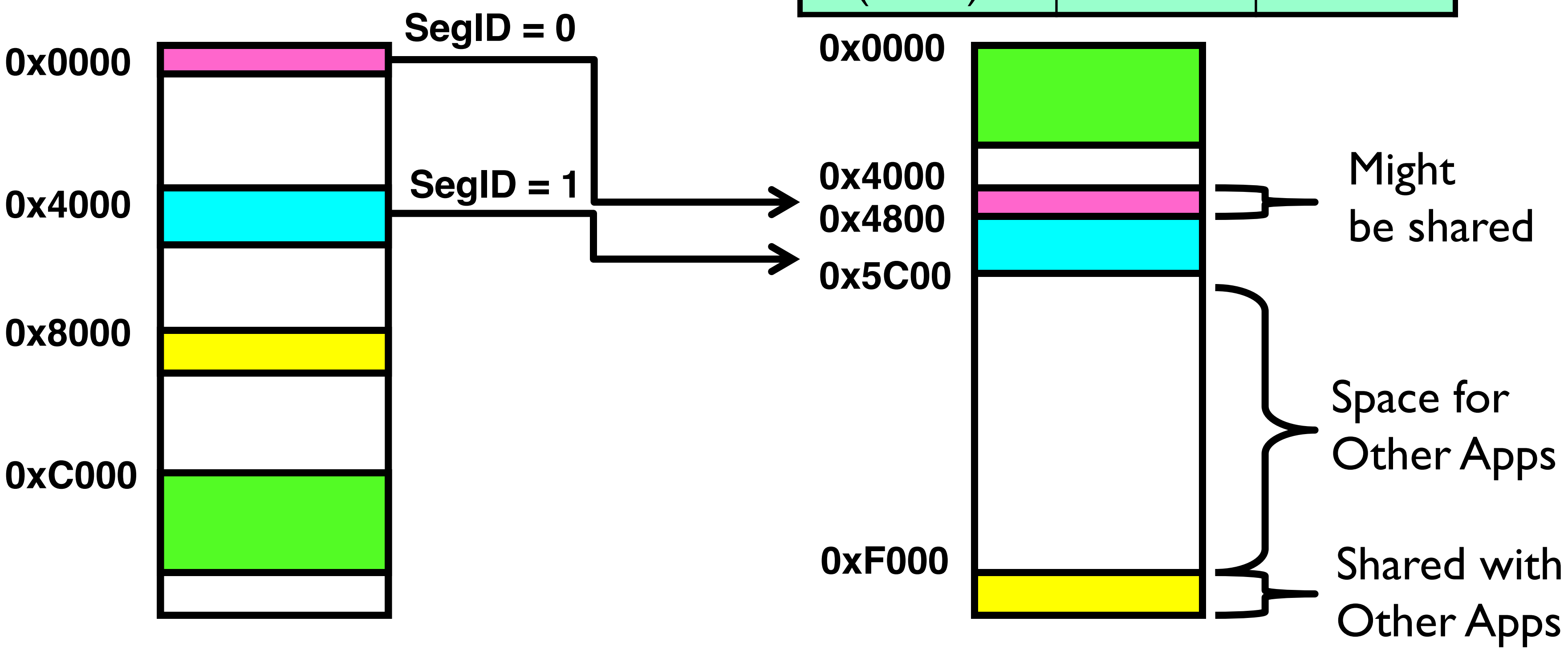
Physical
Address Space

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



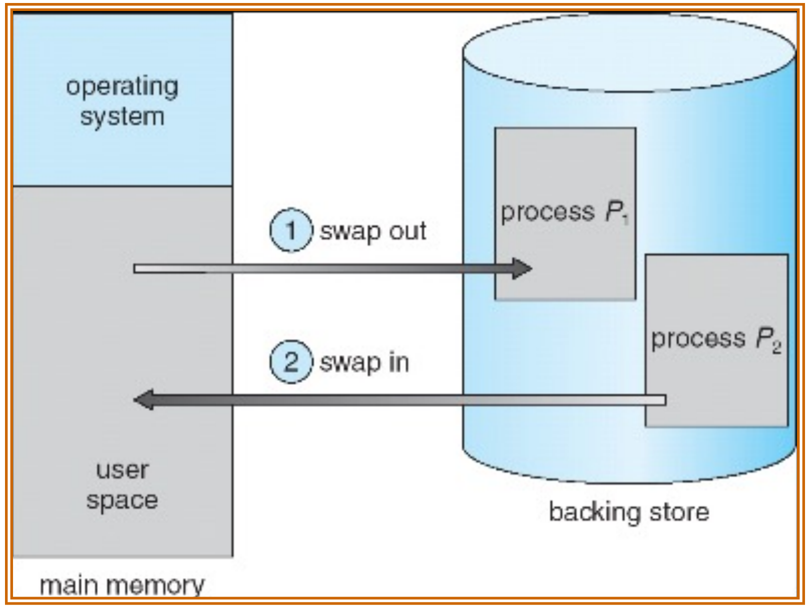
Virtual Address Space

Physical Address Space

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 processes 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

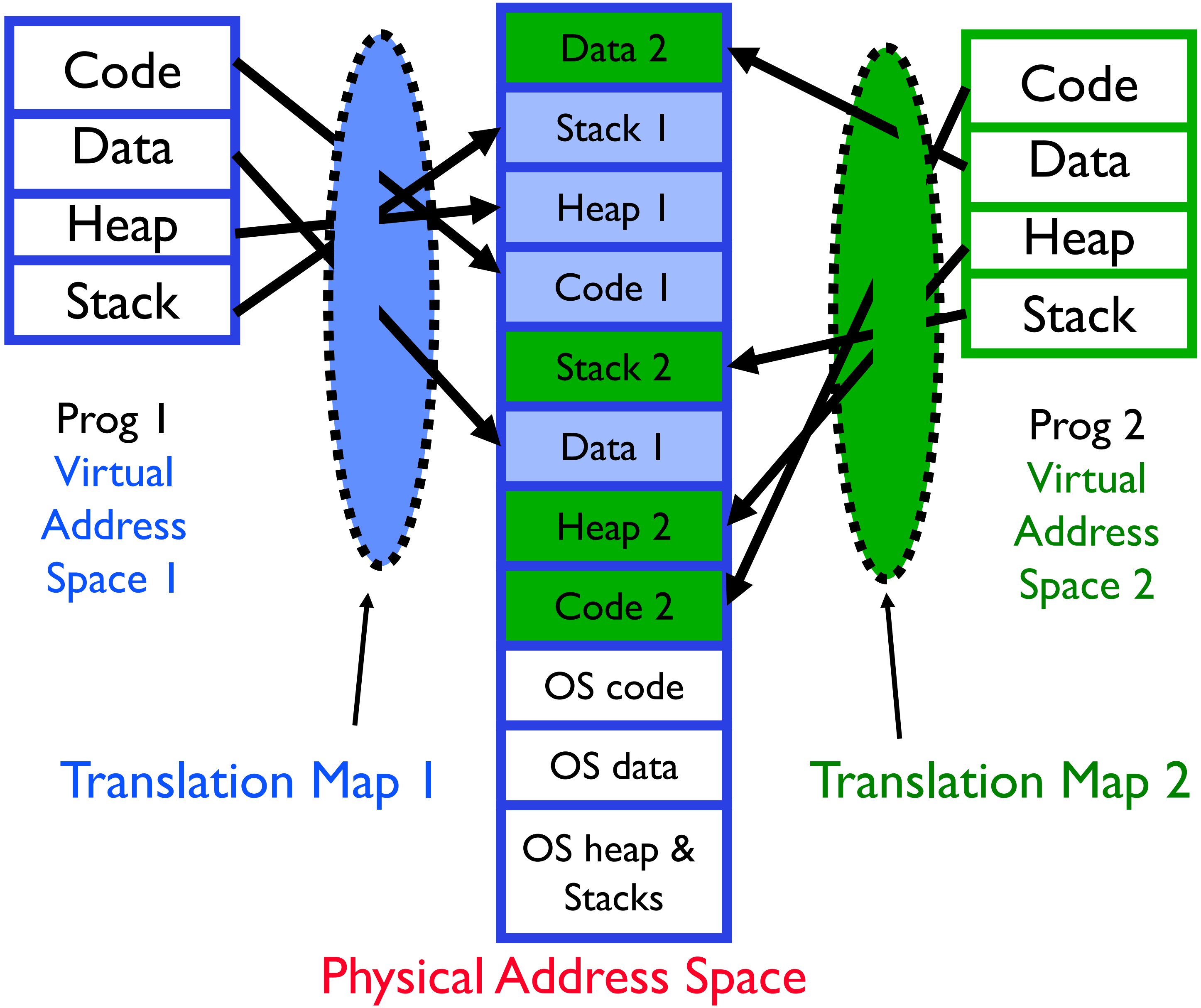
- Must fit variable-sized chunks into physical memory

- 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

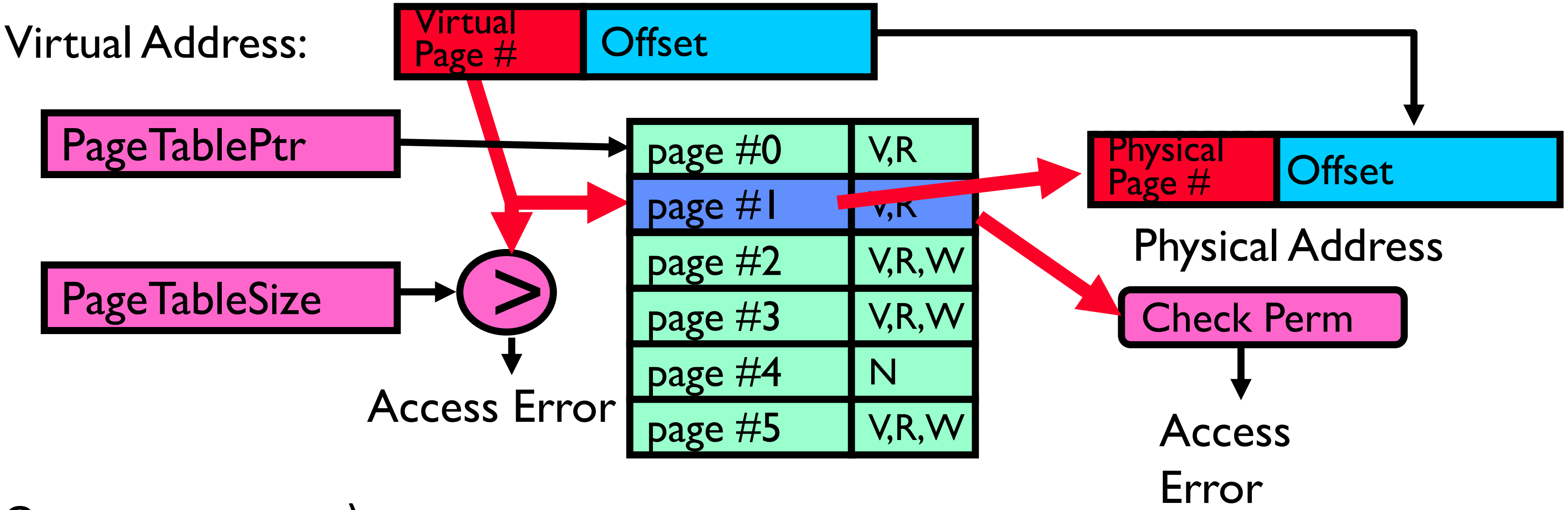


Paging: Physical Memory in Fixed Size Chunks

- 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` ⇒ allocated, `0` ⇒ 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?

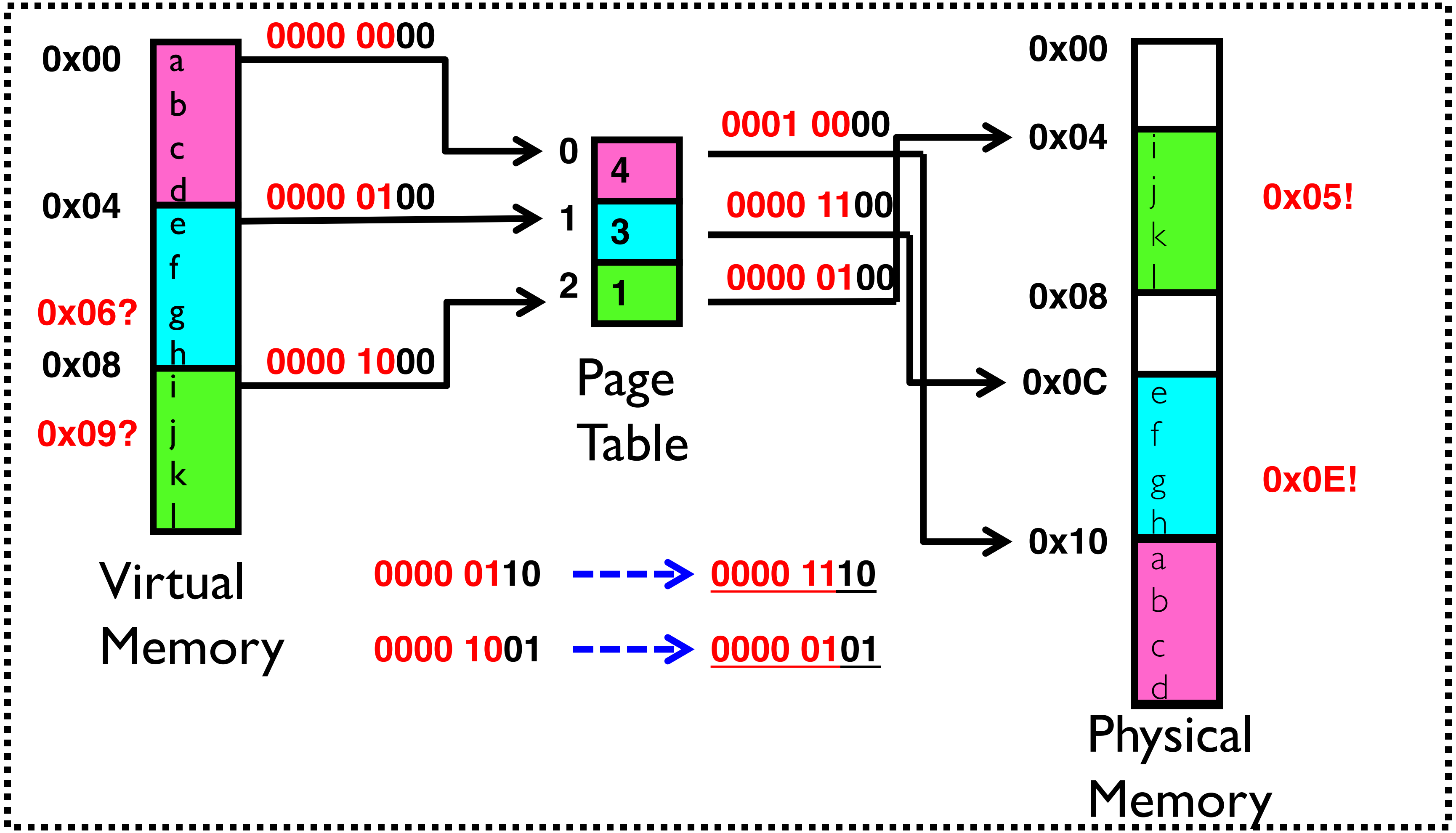


page #0	V,R
page #1	V,R
page #2	V,R,W
page #3	V,R,W
page #4	N
page #5	V,R,W

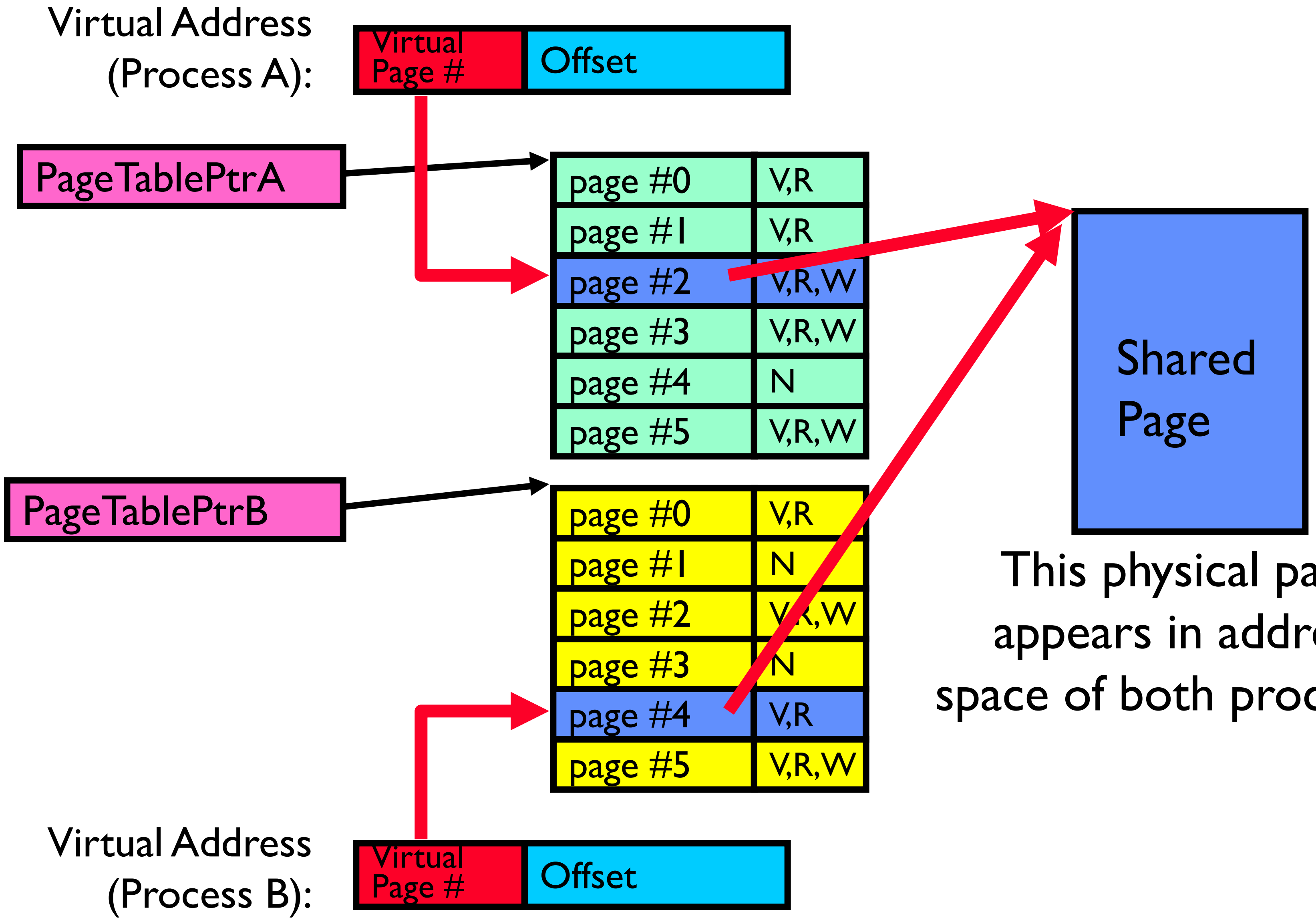
- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page (e.g. Valid bits, Read, Write, etc.)
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - » Example: 10 bit offset ⇒ 1024-byte pages
  - 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?



page #0	V,R
page #1	V,R
page #2	V,R,W
page #3	V,R,W
page #4	N
page #5	V,R,W

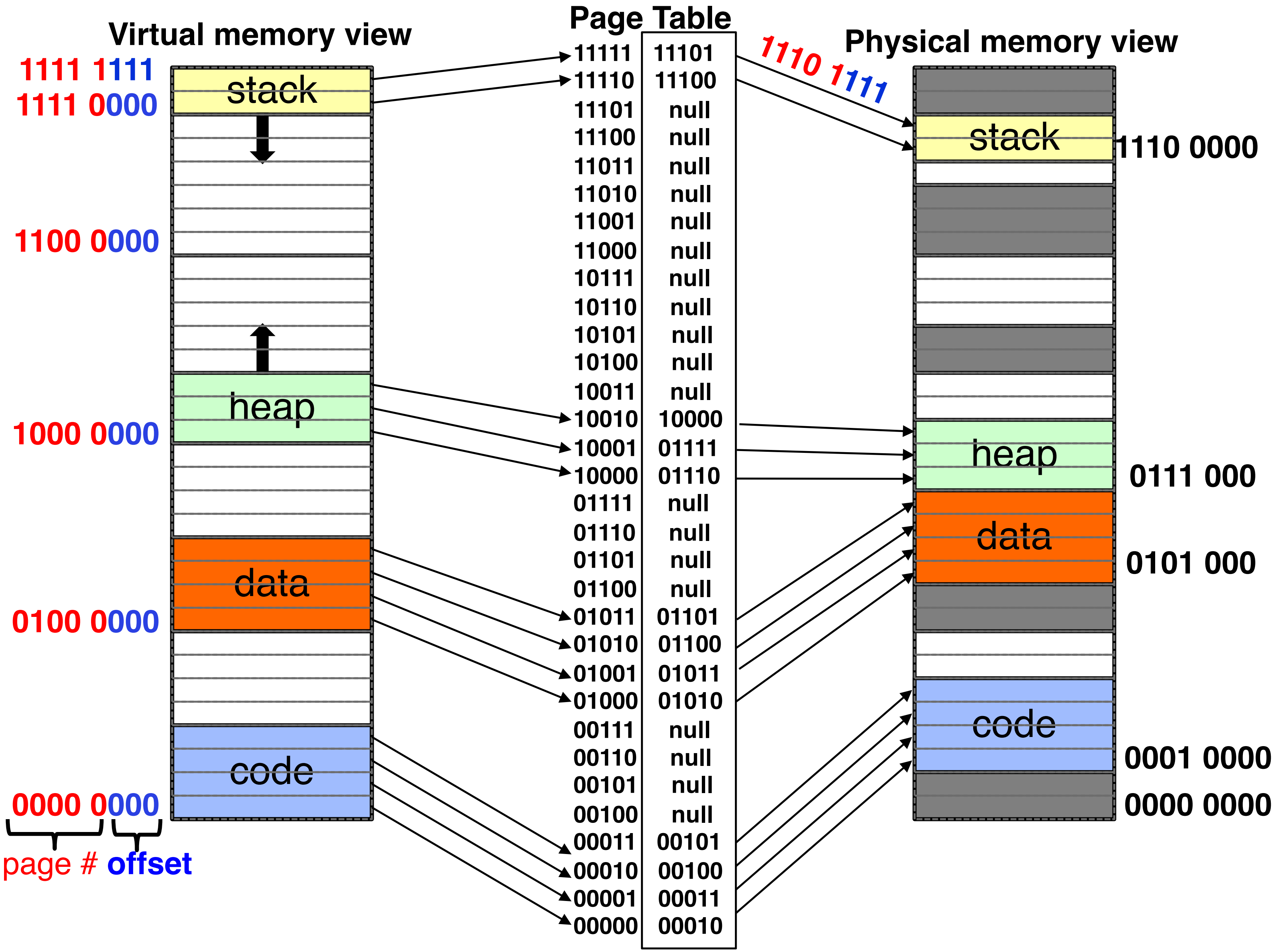
page #0	V,R
page #1	N
page #2	V,R,W
page #3	N
page #4	V,R
page #5	V,R,W

This physical page appears in address space of both processes

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

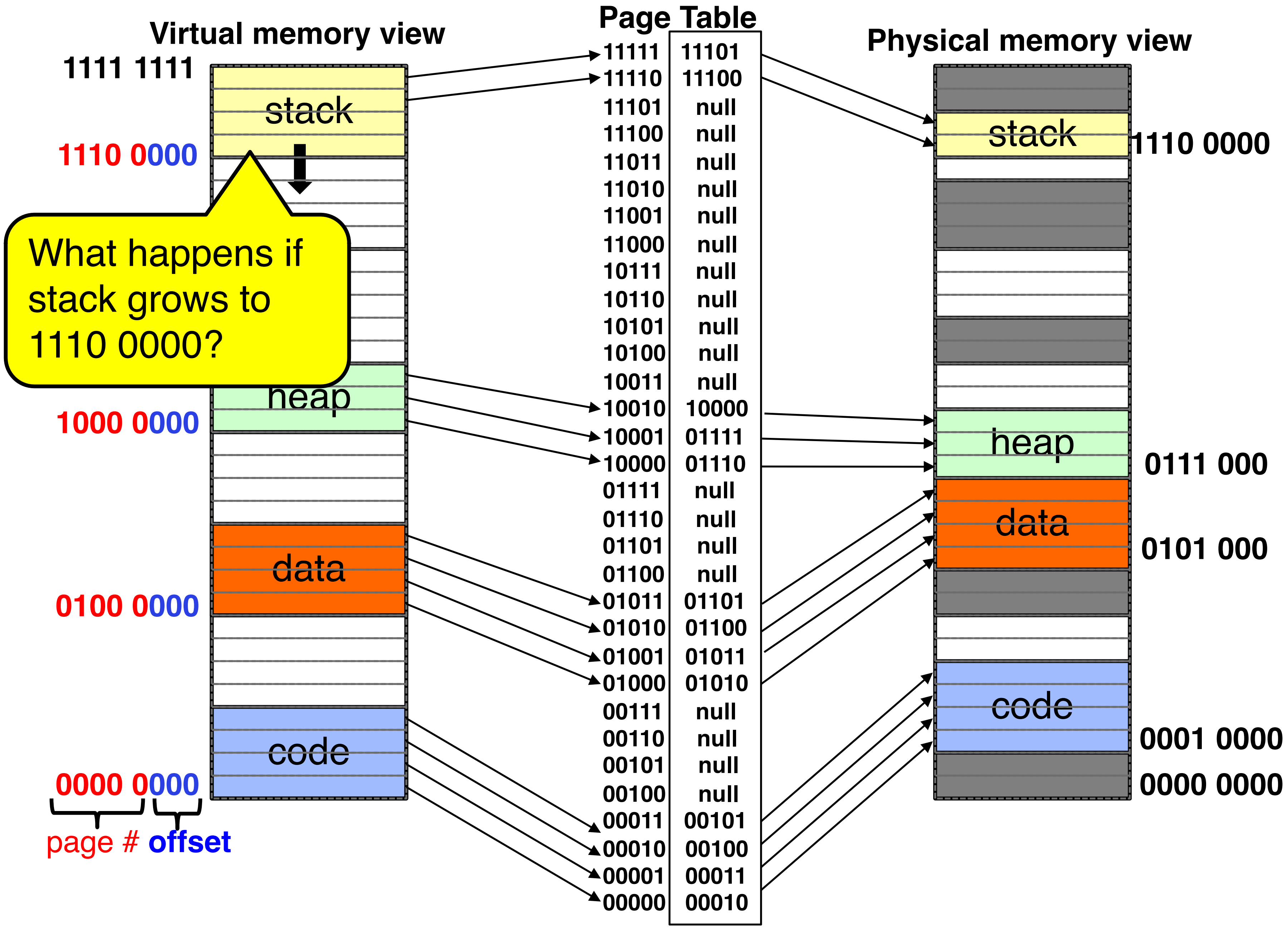
Summary: Paging



Page Table

11111	11101
11110	11100
11101	null
11100	null
11011	null
11010	null
11001	null
11000	null
10111	null
10110	null
10101	null
10100	null
10011	null
10010	10000
10001	01111
10000	01110
01111	null
01110	null
01101	null
01100	null
01011	01101
01010	01100
01001	01011
01000	01010
00111	null
00110	null
00101	null
00100	null
00011	00101
00010	00100
00001	00011
00000	00010

Summary: Paging

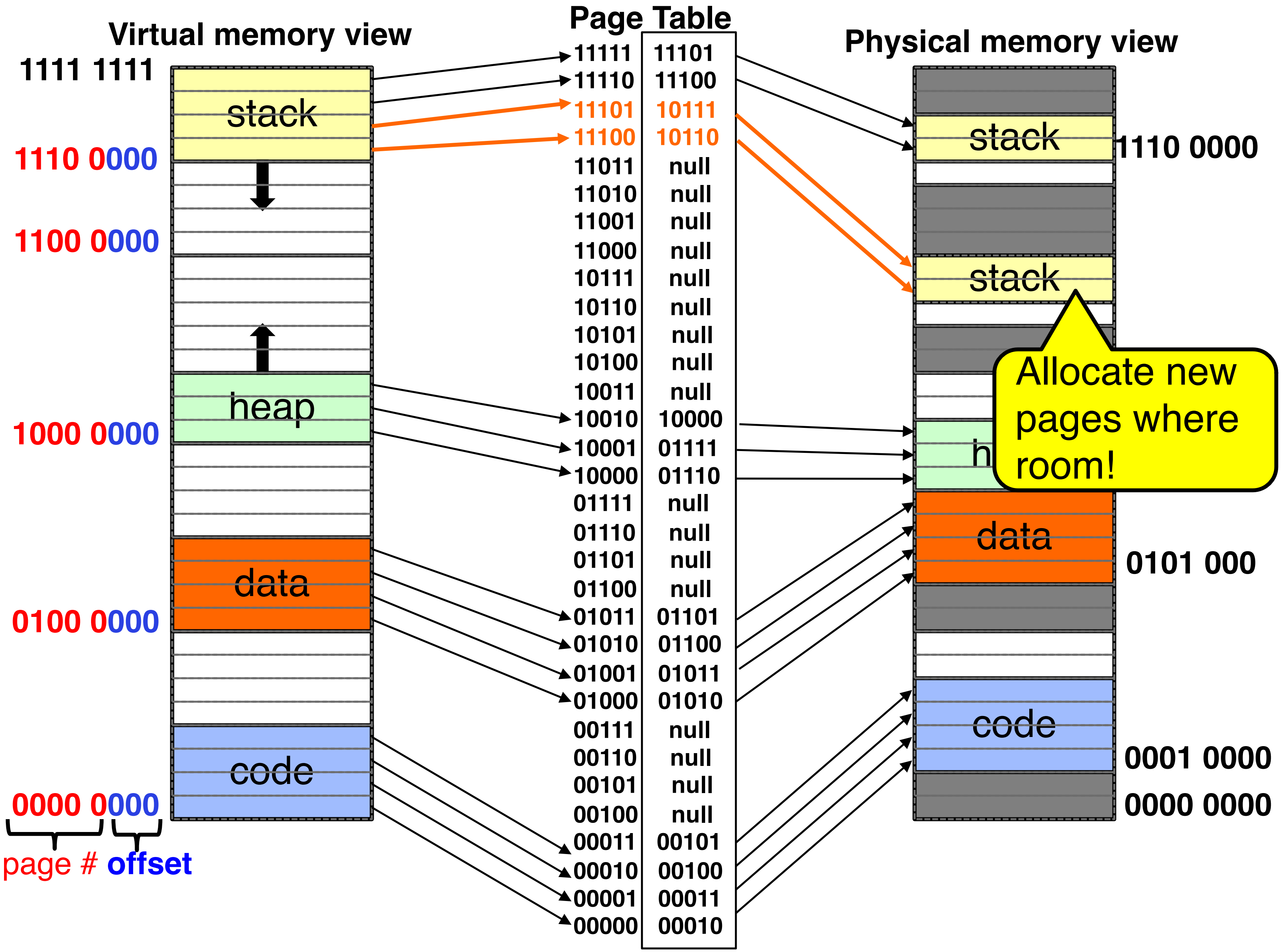


What happens if stack grows to 1110 0000?

Page Table

11111	11101
11110	11100
11101	null
11100	null
11011	null
11010	null
11001	null
11000	null
10111	null
10110	null
10101	null
10100	null
10011	null
10010	10000
10001	01111
10000	01110
01111	null
01110	null
01101	null
01100	null
01011	01101
01010	01100
01001	01011
01000	01010
00111	null
00110	null
00101	null
00100	null
00011	00101
00010	00100
00001	00011
00000	00010

Summary: Paging



Page Table

11111	11101
11110	11100
11101	10111
11100	10110
11011	null
11010	null
11001	null
11000	null
10111	null
10110	null
10101	null
10100	null
10011	null
10010	10000
10001	01111
10000	01110
01111	null
01110	null
01101	null
01100	null
01011	01101
01010	01100
01001	01011
01000	01010
00111	null
00110	null
00101	null
00100	null
00011	00101
00010	00100
00001	00011
00000	00010

Allocate new pages where room!

How big do things get?

- 32-bit address space => 2^{32} bytes (4 GB)
  - Note: “b” = bit, and “B” = byte
  - And *for memory*:
    - » “K”(kilo) = 2^{10} = 1024 ≈ 10^3 (But not quite!): Sometimes called “Ki” (Kibi)
    - » “M”(mega) = 2^{20} = $(1024)^2$ = 1,048,576 ≈ 10^6 (But not quite!): Sometimes called “Mi” (Mibi)
    - » “G”(giga) = 2^{30} = $(1024)^3$ = 1,073,741,824 ≈ 10^9 (But not quite!): Sometimes called “Gi” (Gibi)
- Typical page size: 4 KB
  - how many bits of the address is that ? (remember 2^{10} = 1024)
  - Ans – 4KB = $4\times 2^{10} = 2^{12} \Rightarrow$ 12 bits of the address
- So how big is the simple page table for *each* process?
  - $2^{32}/2^{12} = 2^{20}$ (that’s about a million entries) x 4 bytes each => 4 MB
  - When 32-bit machines got started (vax 11/780, intel 80386), 16 MB was a LOT of memory
- How big is a simple page table on a 64-bit processor (x86_64)?
  - $2^{64}/2^{12} = 2^{52}$(that’s 4.5×10^{15} or 4.5 exa-entries)×8 bytes each = 36×10^{15} bytes or 36 exa-bytes!!!! This is a ridiculous amount of memory!
  - This is really a lot of space – for only the page table!!!
- The address space is *sparse*, i.e. has holes that are not mapped to physical memory
  - So, most of this space is taken up by page tables mapped to nothing

Group Discussion

- Topic: segmentation vs. paging
  - Can you compare segmentation and paging?
  - What are the pros and cons of each approach?

- 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

Page Table Discussion

- What needs to be switched on a context switch?
  - Page table pointer and limit
- What provides protection here?
  - Translation (per process) *and* dual-mode!
  - Can't let process alter its own page table!
- Analysis
  - Pros
    - » Simple memory allocation
    - » Easy to share
  - Con: What if address space is sparse?
    - » E.g., on UNIX, code starts at 0, stack starts at (2^{31}-1)
    - » With 4KB pages, need 1 million page table entries!
  - Con: What if table really big?
    - » Not all pages used all the time ⇒ would be nice to have working set of page table in memory
- Simple Page table is way too big!
  - Does it all need to be in memory?
  - How about multi-level paging?
  - or combining paging and segmentation

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 tables can be placed into virtual memory
- Next Time: Multi-Level Tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space