

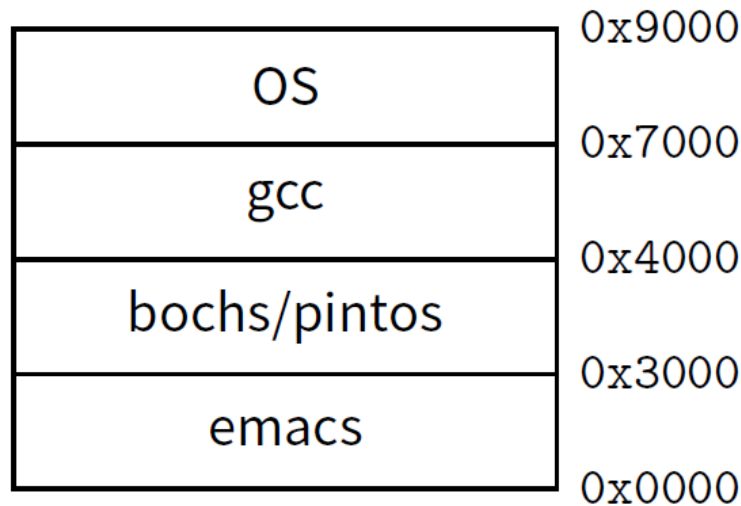
# Lecture 8:

# Address Translation

---

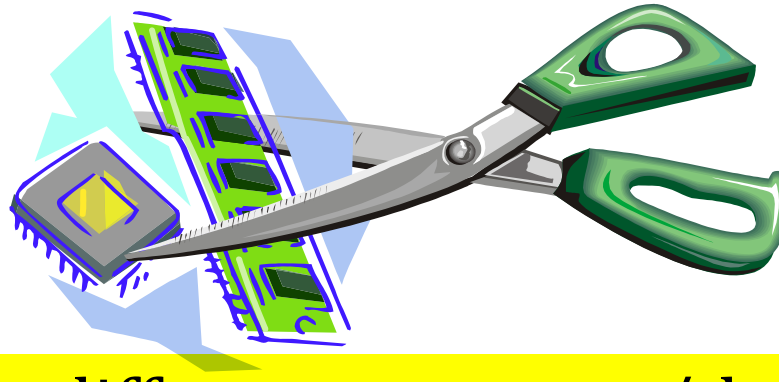
Bo Tang @ 2021, Spring

# Want Processes to Co-exist



- ◆ Consider multiprogramming on physical memory
  - ◆ What happens when pintos needs to expand?
  - ◆ If emacs needs more memory than is on the machine?
  - ◆ If pintos has an error and writes to address 0x7100?
  - ◆ When does gcc have to know it will run at 0x4000?
  - ◆ What is emacs is not using its memory?

# Virtualizing Resources



◆ **Physical Reality:** different processes/threads share the same hardware

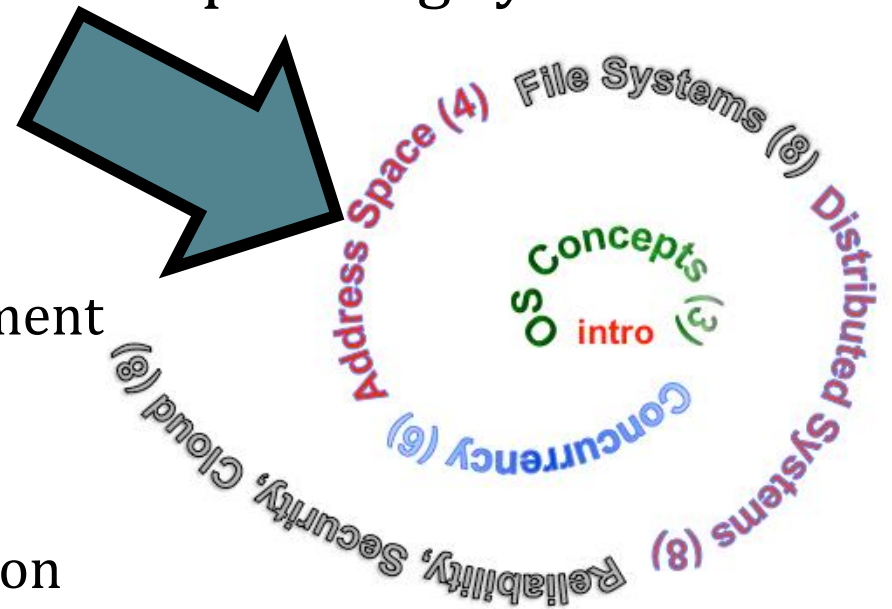
- ◆ Need to multiplex CPU (done)
- ◆ Need to multiplex use of Memory (today)
- ◆ Need to multiplex disk and devices (later in term)

◆ **Why worry about memory sharing?**

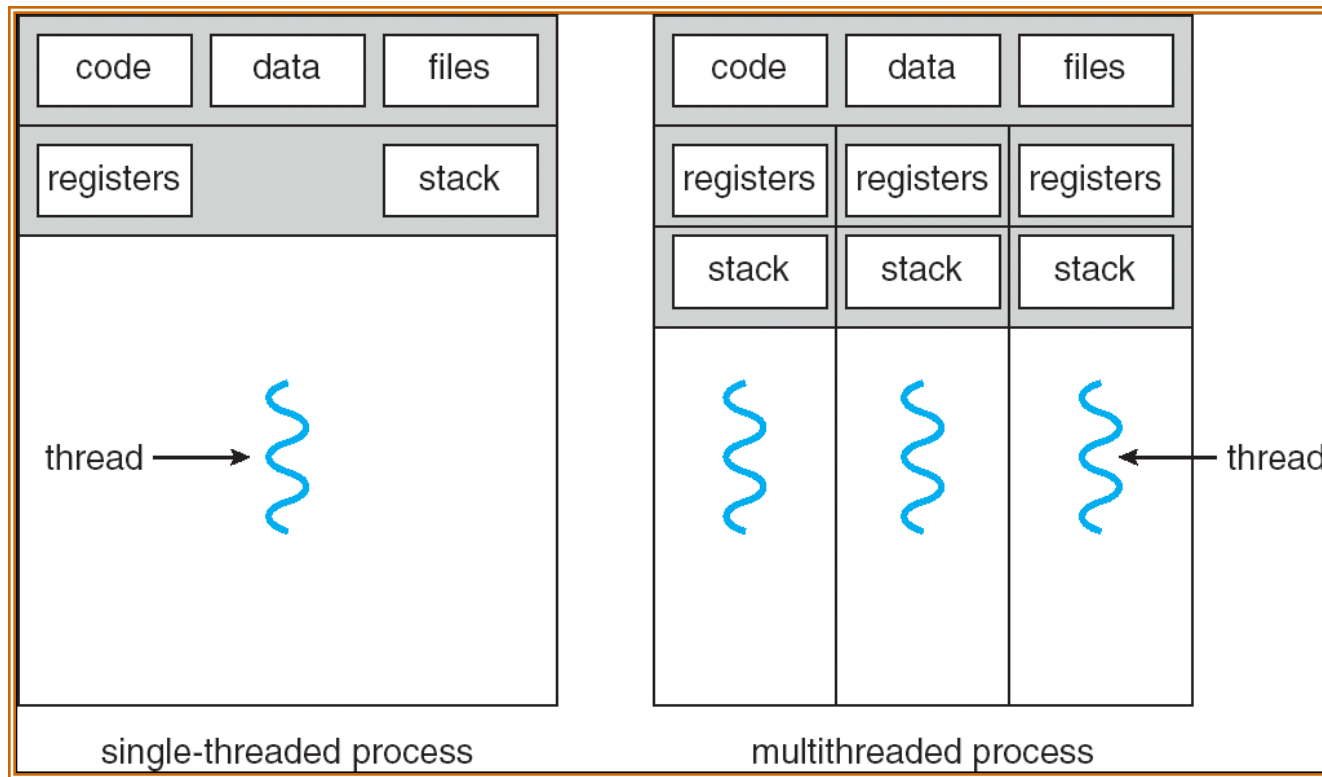
- ◆ The complete working state of a process is defined by its data in memory (and registers)
- ◆ Consequently, two different processes cannot use the same memory
  - ◆ Physics: two different data cannot occupy same locations in memory
- ◆ May not want different threads to have access to each other's memory

# Next Objective

- ◆ Dive deeper into the concepts and mechanisms of memory sharing and address translation
- ◆ Enabler of many key aspects of operating systems
  - ◆ Protection
  - ◆ Multi-programming
  - ◆ Isolation
  - ◆ Memory resource management
  - ◆ I/O efficiency
  - ◆ Sharing
  - ◆ Inter-process communication
  - ◆ Demand paging
- ◆ Today: Linking, Segmentation



# Recall: Single and Multithreaded Processes



- ◆ Threads encapsulate concurrency
  - ◆ “Active” component of a process
- ◆ Address spaces encapsulate protection
  - ◆ Keeps buggy program from trashing the system
  - ◆ “Passive” component of a process

# Important Aspects of Memory Multiplexing

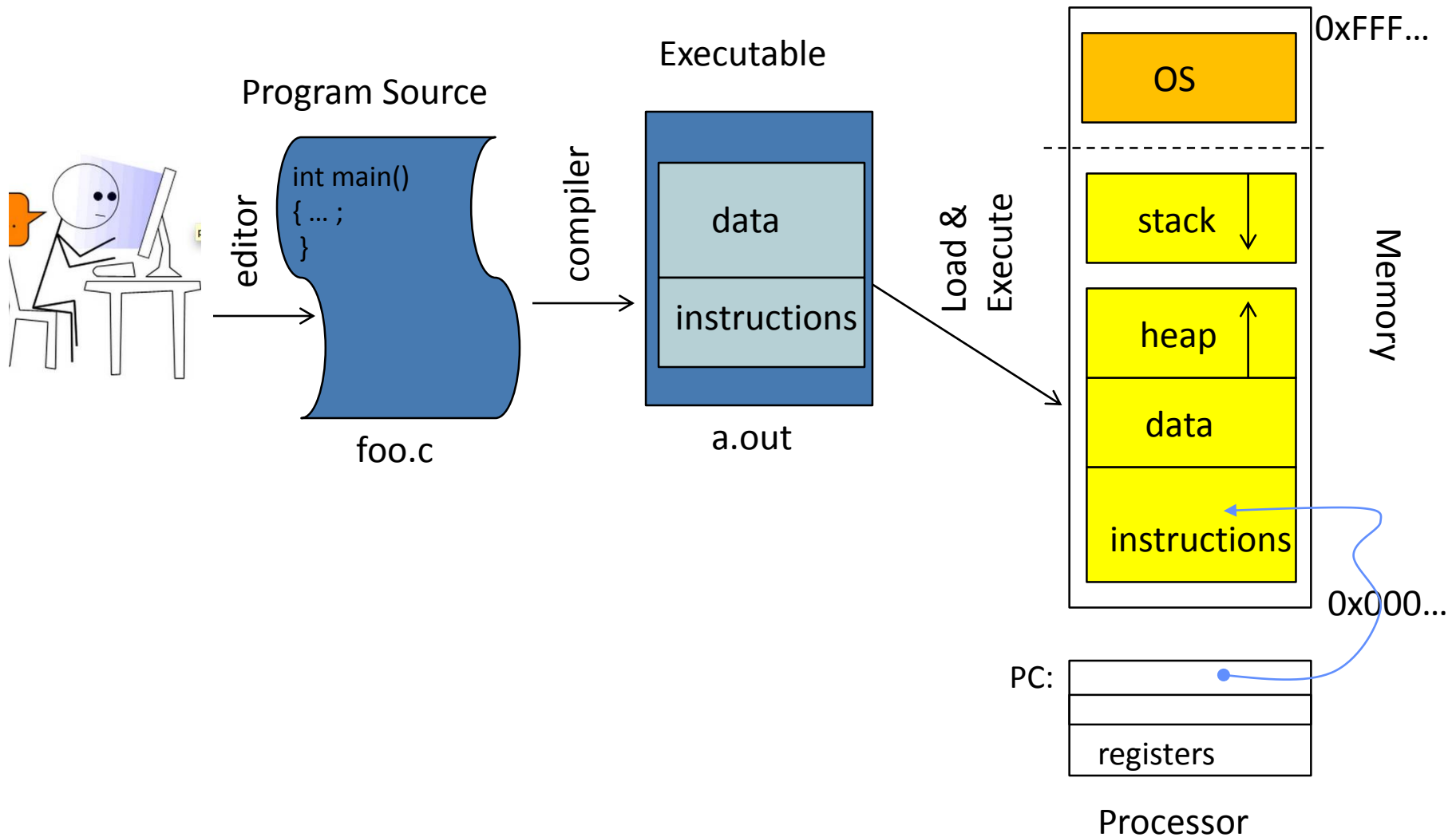
- ◆ **Protection:** prevent access to private memory of other processes
  - ◆ Kernel data protected from User programs
  - ◆ Programs protected from themselves
  - ◆ May want to give special behavior to different memory regions (Read Only, Invisible to user programs, etc)
- ◆ **Controlled overlap:** sometimes we want to share memory across processes.
  - ◆ E.g., communication across processes, share code
  - ◆ Need to control such overlap

# Important Aspects of Memory Multiplexing

## ◆ Translation:

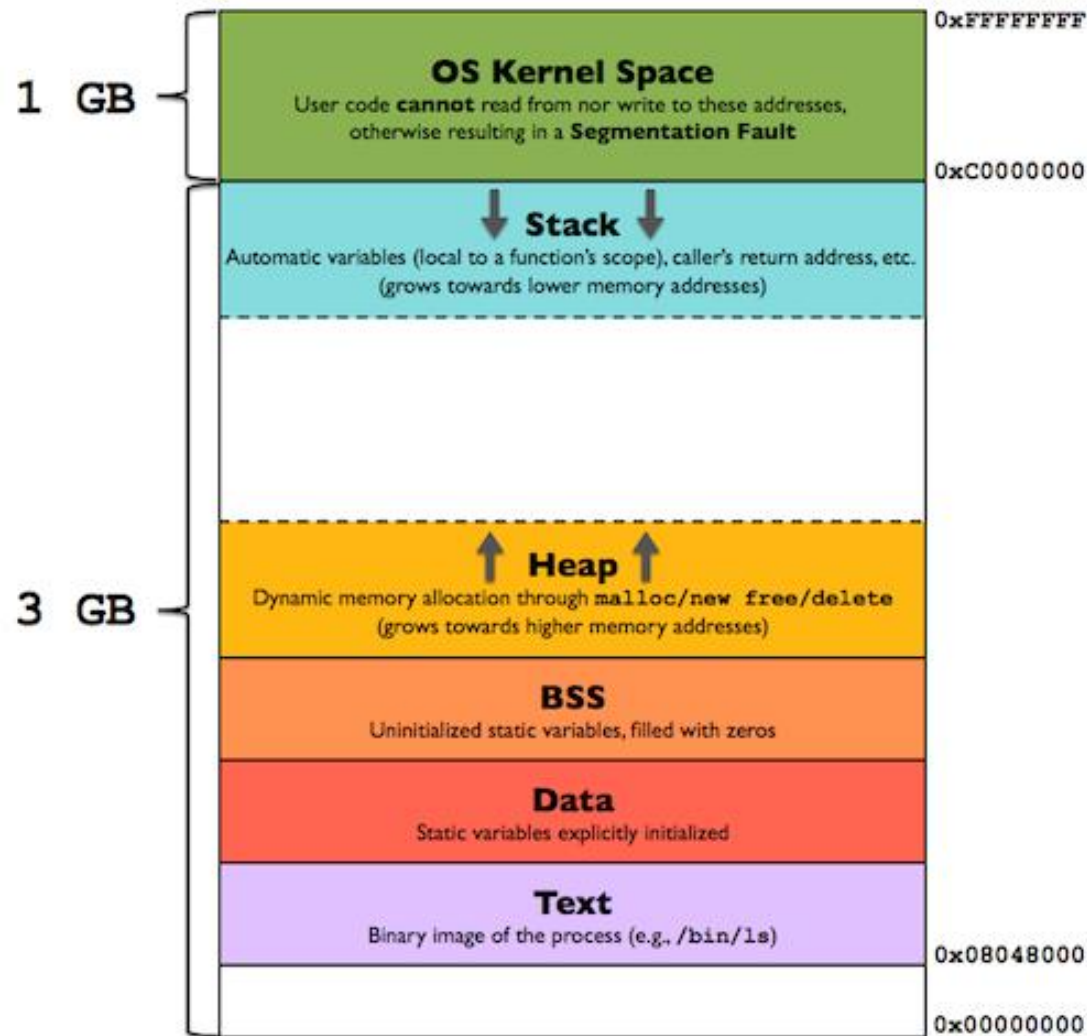
- ◆ Ability to translate accesses from one address space (virtual) to a different one (physical)
- ◆ When translation exists, processor uses virtual addresses, physical memory uses physical addresses
- ◆ Side effects:
  - ◆ Can be used to give uniform view of memory to programs
  - ◆ Can be used to provide protection (e.g., avoid overlap)
  - ◆ Can be used to control overlap

# Recall: OS Bottom Line: Run Programs





# Recall: Address Space

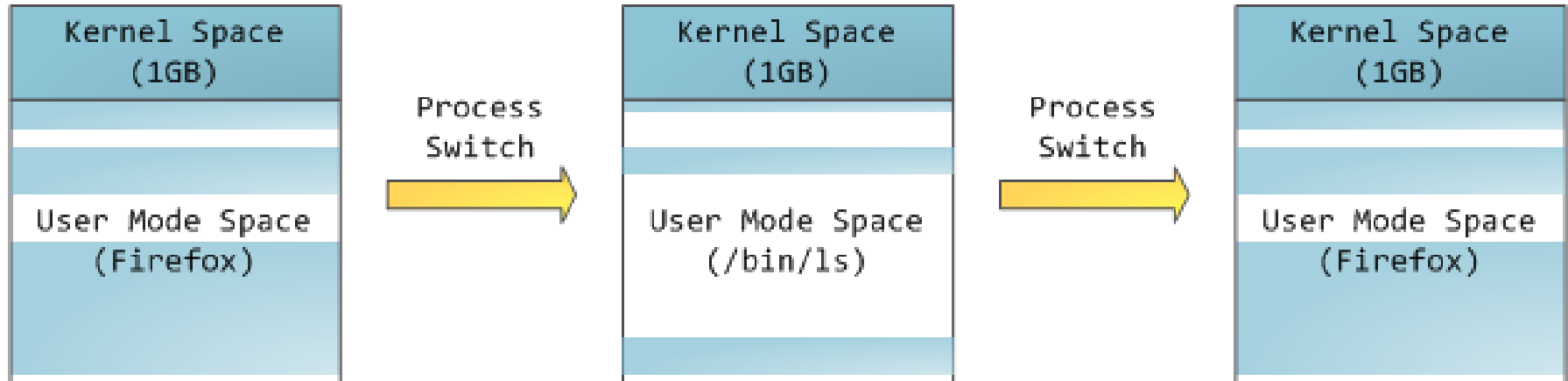


Each process has  
its own user  
(address) space

Each process  
thinks it has  $2^{32}$   
( $2^{64}$ ) byte  
memory

<https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/> (text → code segment, read only / execute), BSS: block started by symbol, “static int i” (data segment: read-write )

# Recall: Context Switch



One address maps to one byte.

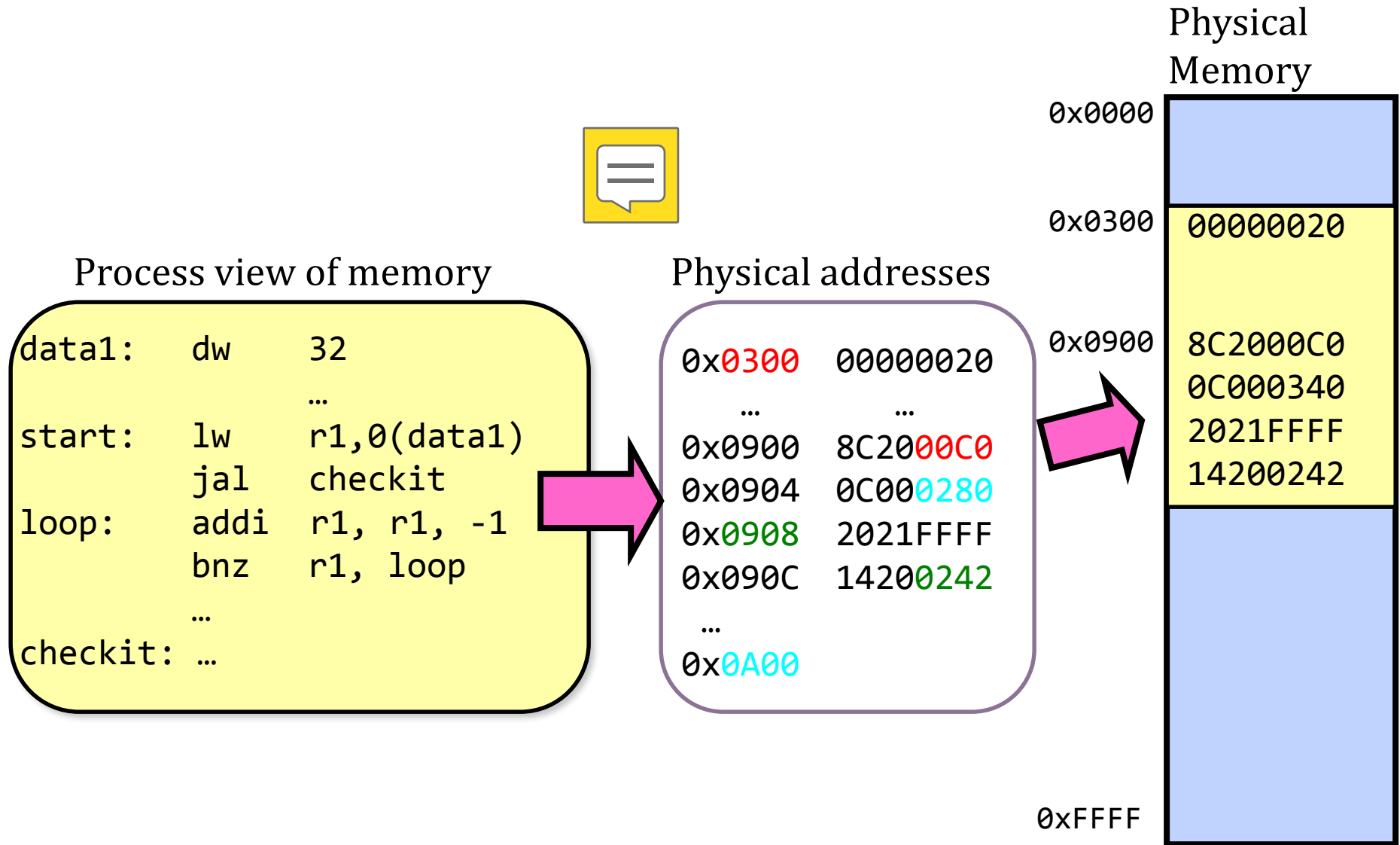
On a 32-bit system,

- The maximum amount of memory in a process is  $2^{32}$  bytes = **4GB**.

Then, how about a 64-bit system? = **16EB**

<http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/>

# Binding of Instructions and Data to Memory



# 2nd copy of program from previous example

- Call `fork()`

Process view of memory

```
data1:    dw    32
          ...
start:    lw     r1,0(data1)
          jal    checkit
loop:     addi   r1, r1, -1
          bnz    r1, r0, loop
          ...
checkit:  ...
```

Physical addresses

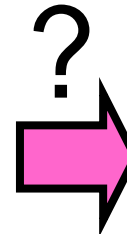
```
0x0300    00000020
          ...
0x0900    8C2000C0
0x0904    0C000280
0x0908    2021FFFF
0x090C    14200242
          ...
0x0A00
```

Physical  
Memory

0x0000

0x0300

0x0900

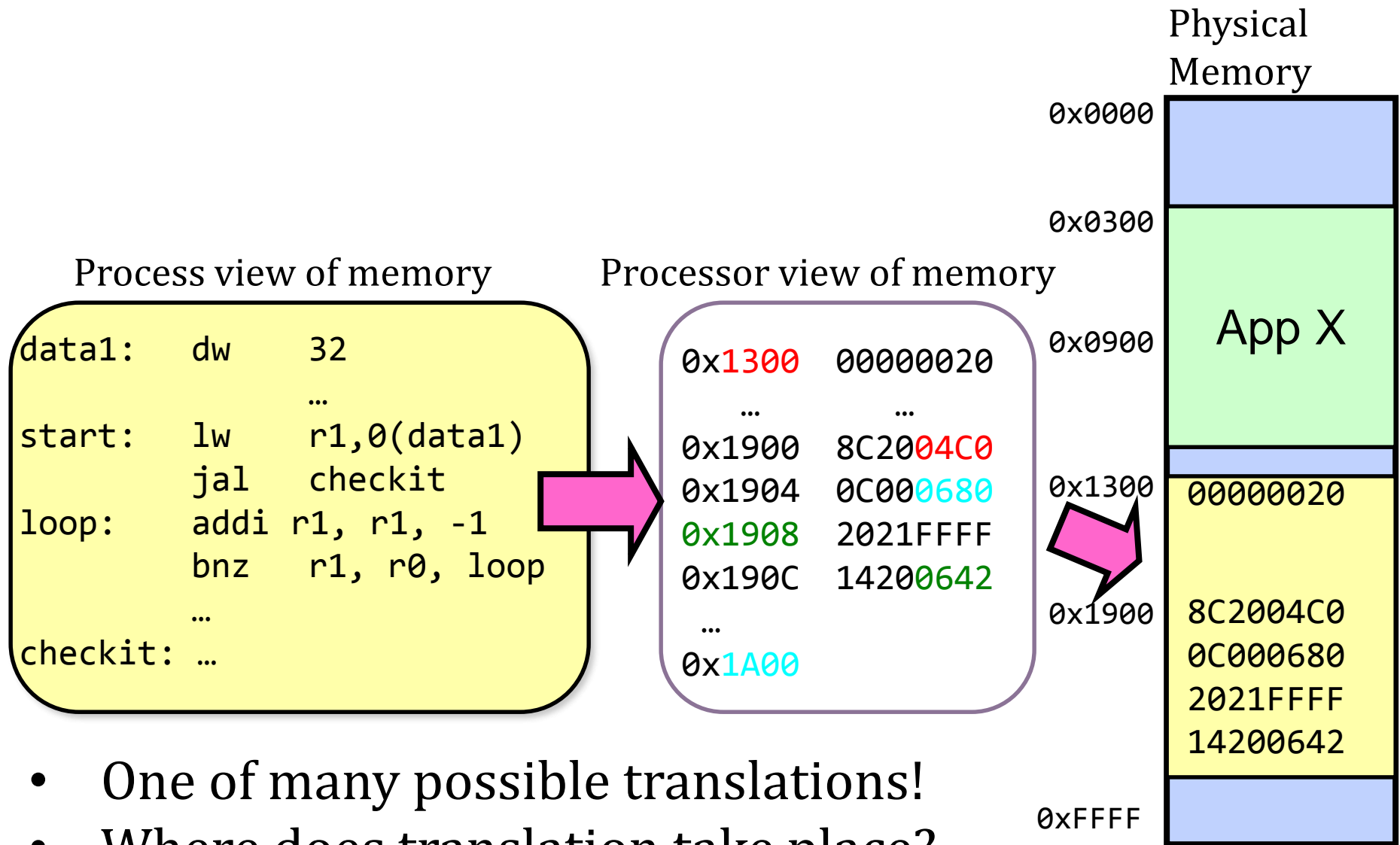


App X

0xFFFF

Need address translation!

# 2nd copy of program from previous example

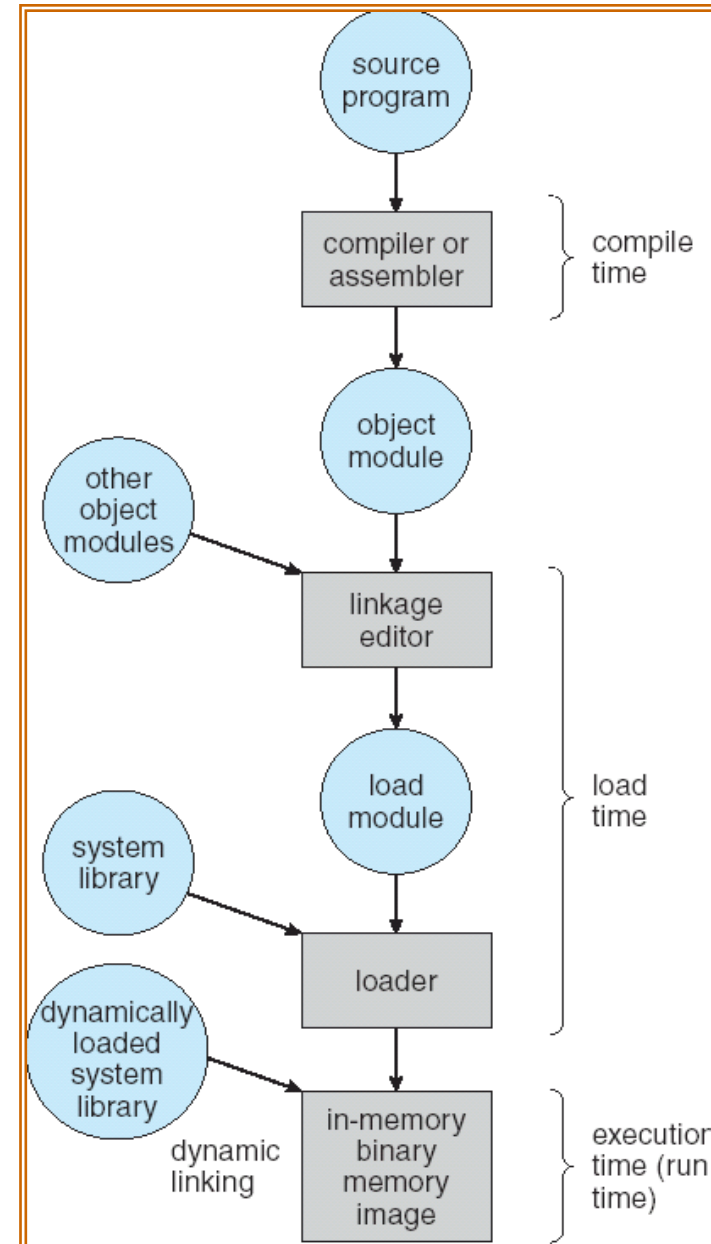


- One of many possible translations!
- Where does translation take place?

Compile time, Link/Load time, or Execution time?

# Multi-step Processing of a Program for Execution

- ◆ Preparation of a program for execution involves components at:
  - (1) Compile time (i.e., “gcc”)
  - (2) Link/Load time (UNIX “ld” does link)
  - (3) Execution time (e.g., dynamic libs)
- ◆ Addresses can be bound to final values anywhere in this path
  - ◆ Depends on hardware support
  - ◆ Also depends on operating system
- ◆ Dynamic Libraries
  - ◆ Linking postponed until execution
  - ◆ Small piece of code, *stub*, used to locate appropriate memory-resident library routine
  - ◆ Stub replaces itself with the address of the routine, and executes routine

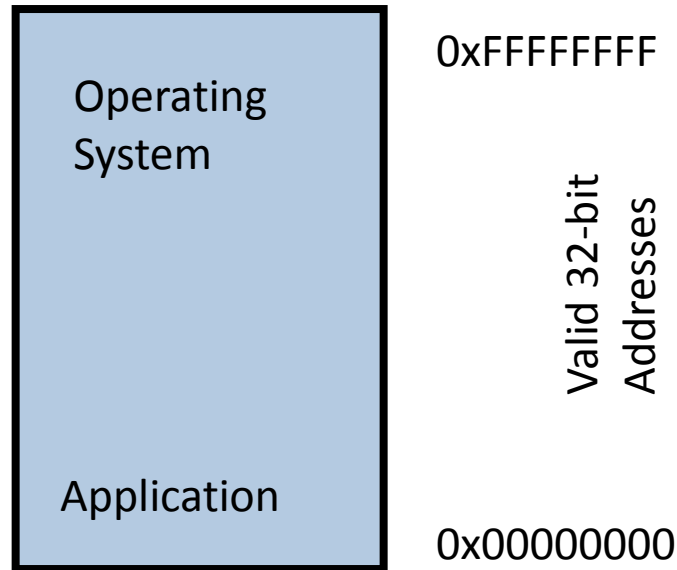


# Multiplexing Memory Approaches

- ◆ Uniprogramming
- ◆ Multiprogramming
  - ◆ Without protection
  - ◆ With protection (base+bound)
- ◆ Virtual memory
  - ◆ Base & Bound
  - ◆ Segmentation
  - ◆ Paging
  - ◆ Paging + Segmentation

# 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

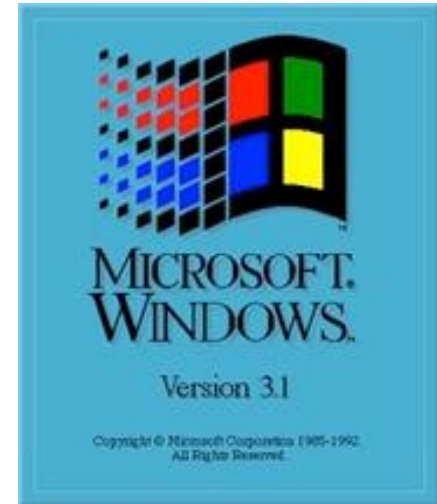
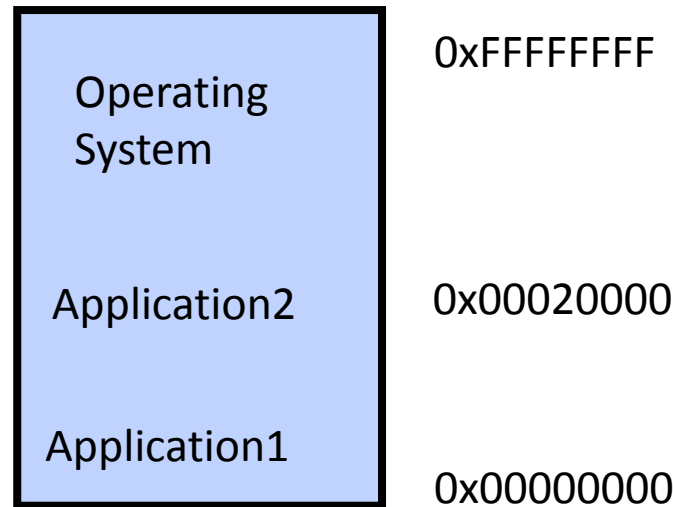


- ◆ Application given illusion of dedicated machine by giving it reality of a dedicated machine



# Multiprogramming (primitive stage)

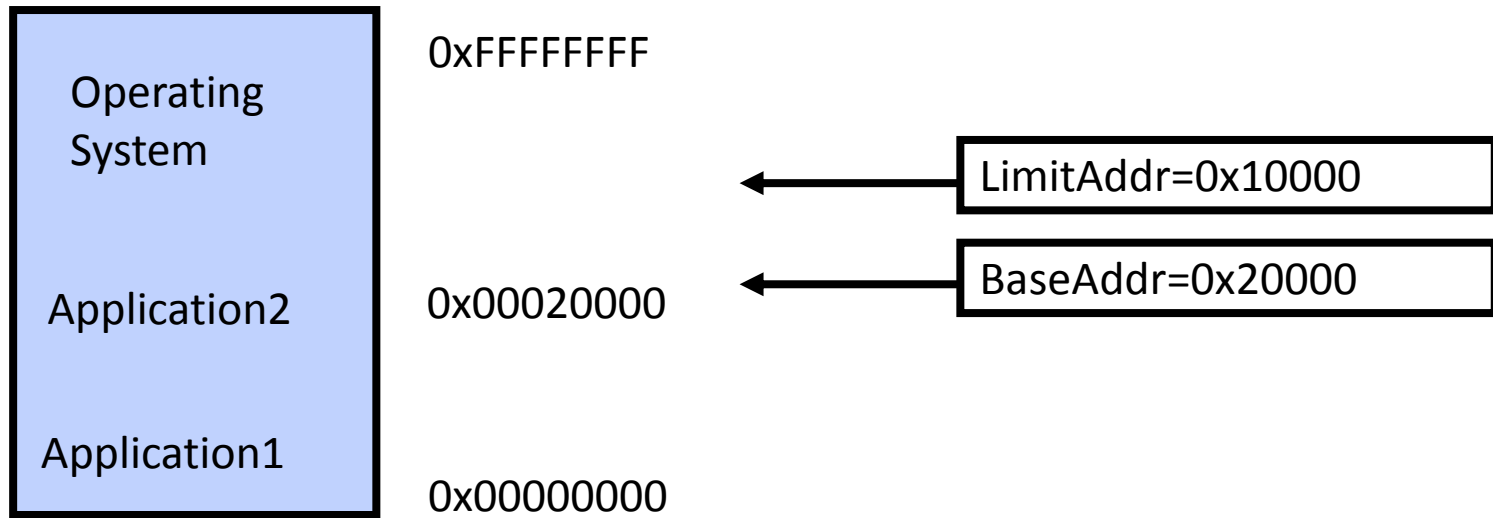
- ◆ Multiprogramming with Translation without Protection
  - ◆ Must somehow prevent address overlap between threads



- ◆ 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

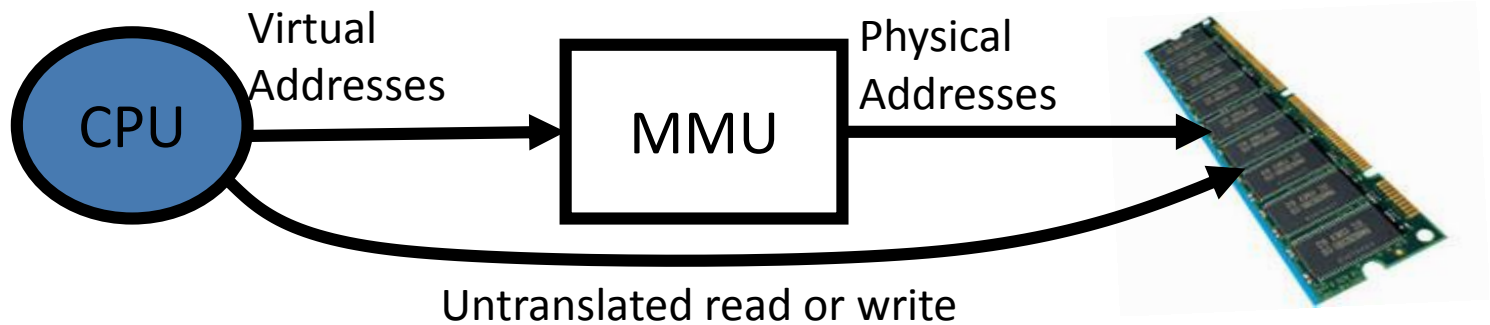
# Multiprogramming (Version with Protection)

- ◆ Can we protect programs from each other without translation?



- ◆ Yes: use two special registers *BaseAddr* and *LimitAddr* to prevent user from straying outside designated area
  - ◆ If user tries to access an illegal address, cause an error
- ◆ During switch, kernel loads new base/limit from PCB (Process Control Block)
  - ◆ User not allowed to change base/limit registers

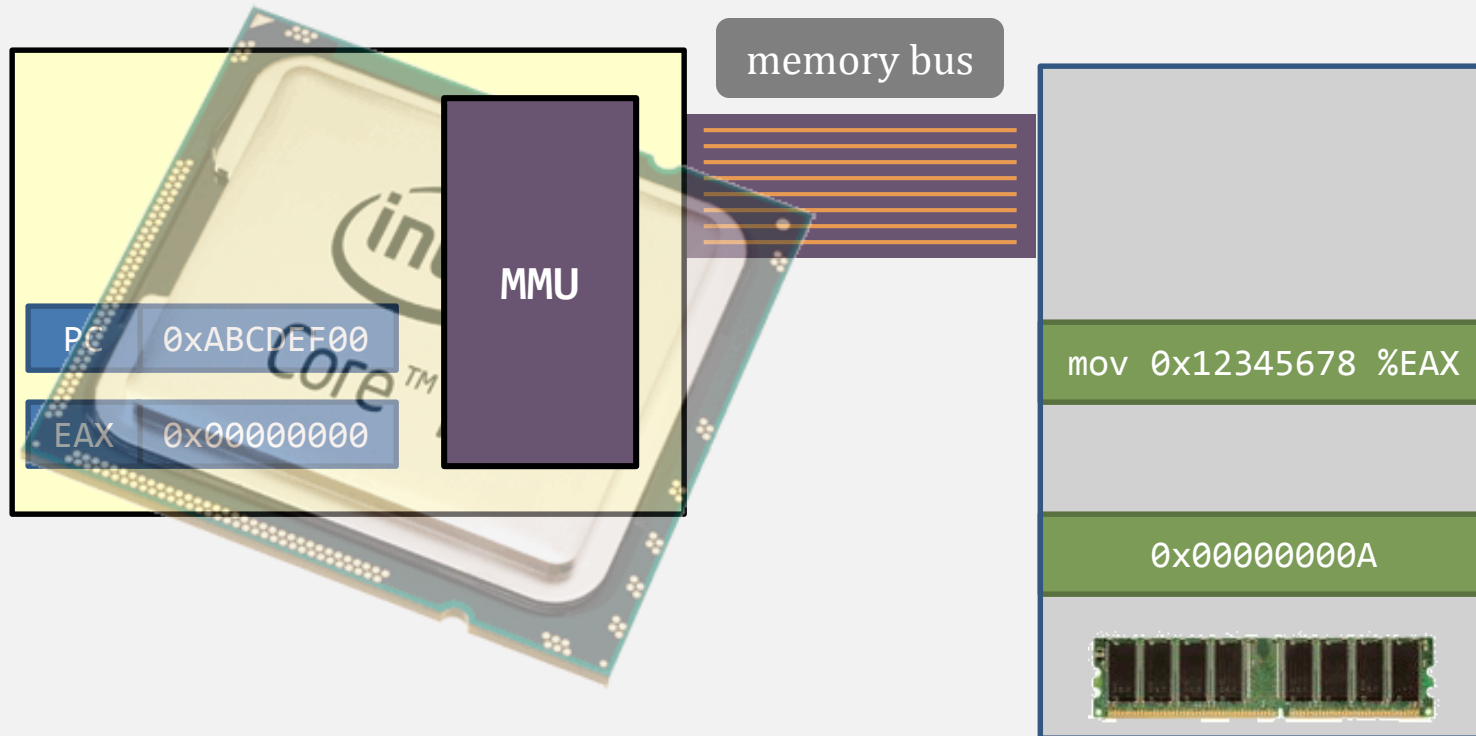
# General Address translation



- ◆ Recall: Address Space:
  - ◆ All the addresses and state a process can touch
  - ◆ Each process has different address space
- ◆ Consequently, two views of memory:
  - ◆ View from the CPU (what program sees, virtual memory)
  - ◆ View from memory (physical memory)
  - ◆ Translation box (MMU) converts between the two views
- ◆ Translation makes it 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

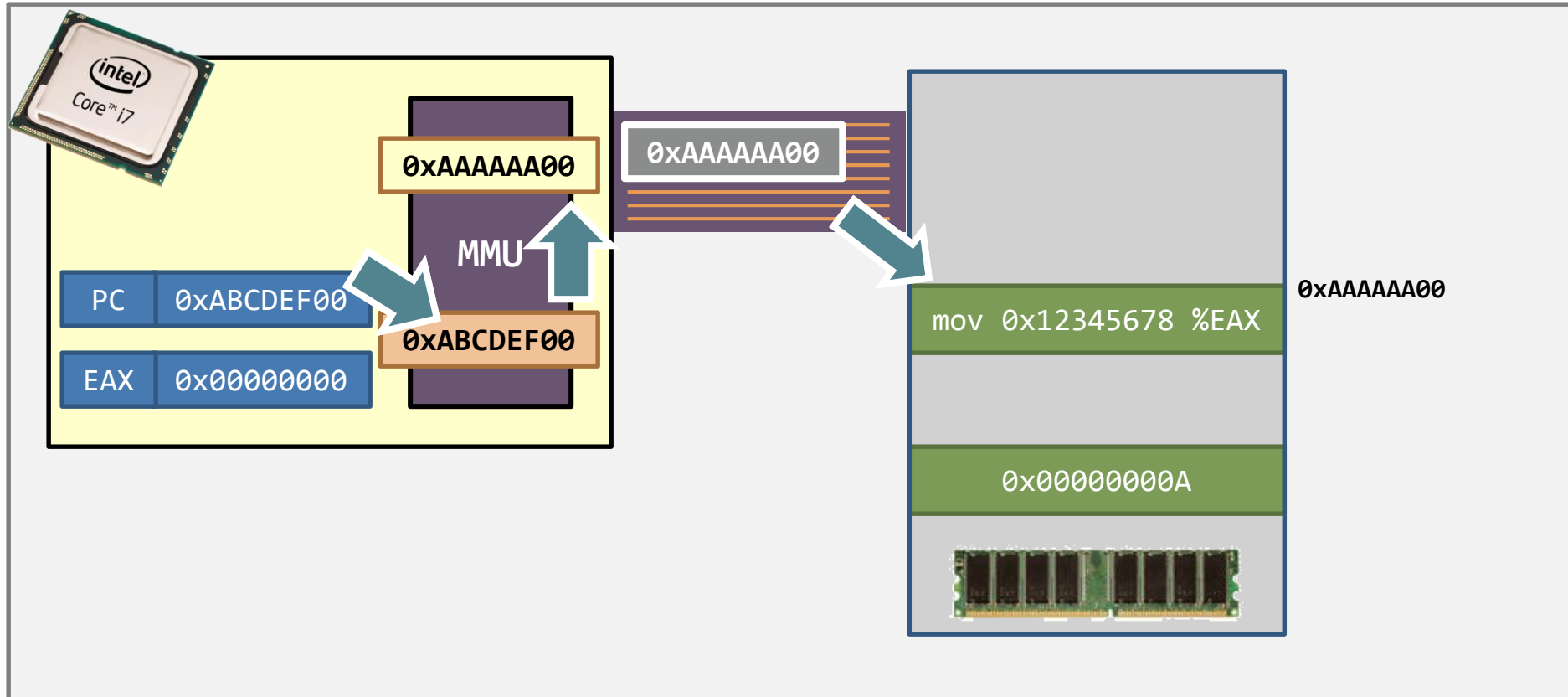
# Virtual memory support in modern CPUs

- ◆ The **MMU – memory management unit**
  - ◆ Usually on-chip (but some architecture may off-chip)



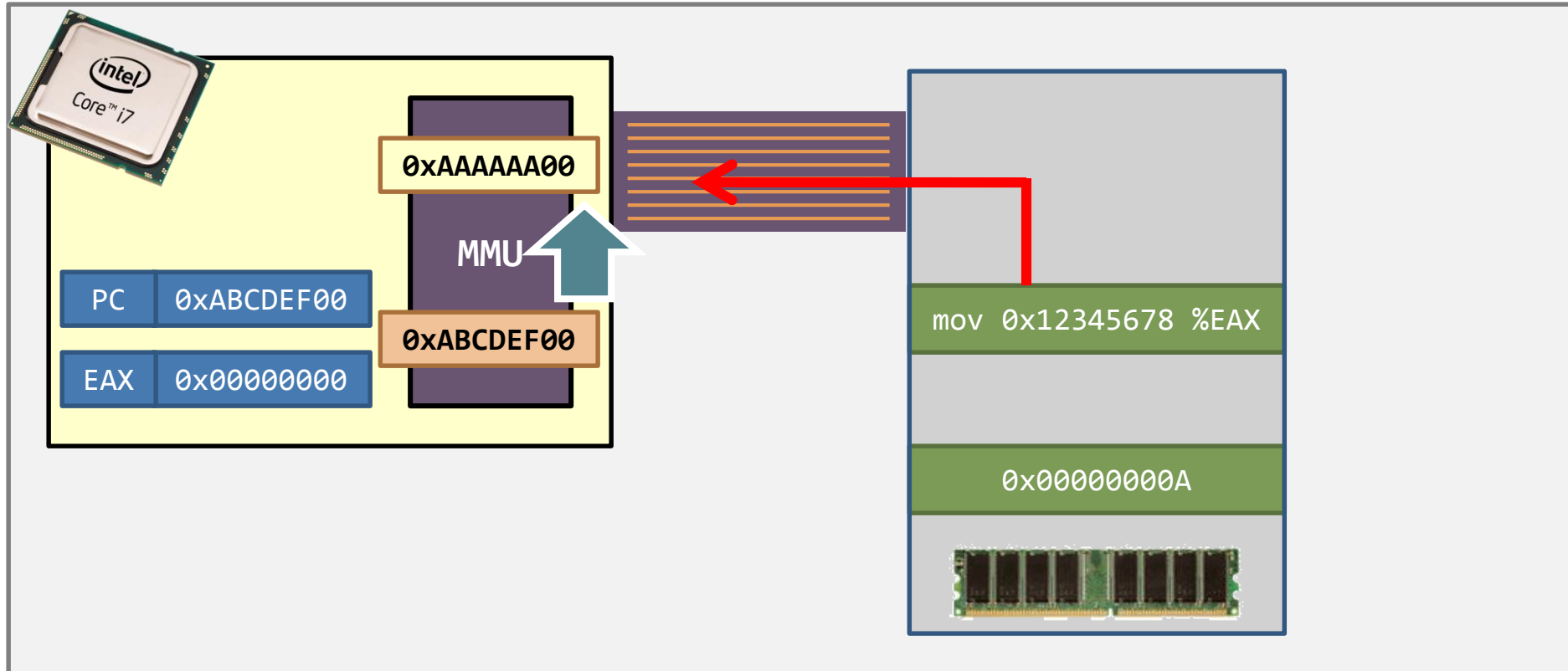
# Virtual memory – how does it work?

- ◆ Step 1. When CPU wants to fetch an instruction
  - ◆ the **virtual address** is sent to MMU and
  - ◆ is translated into a **physical address**.



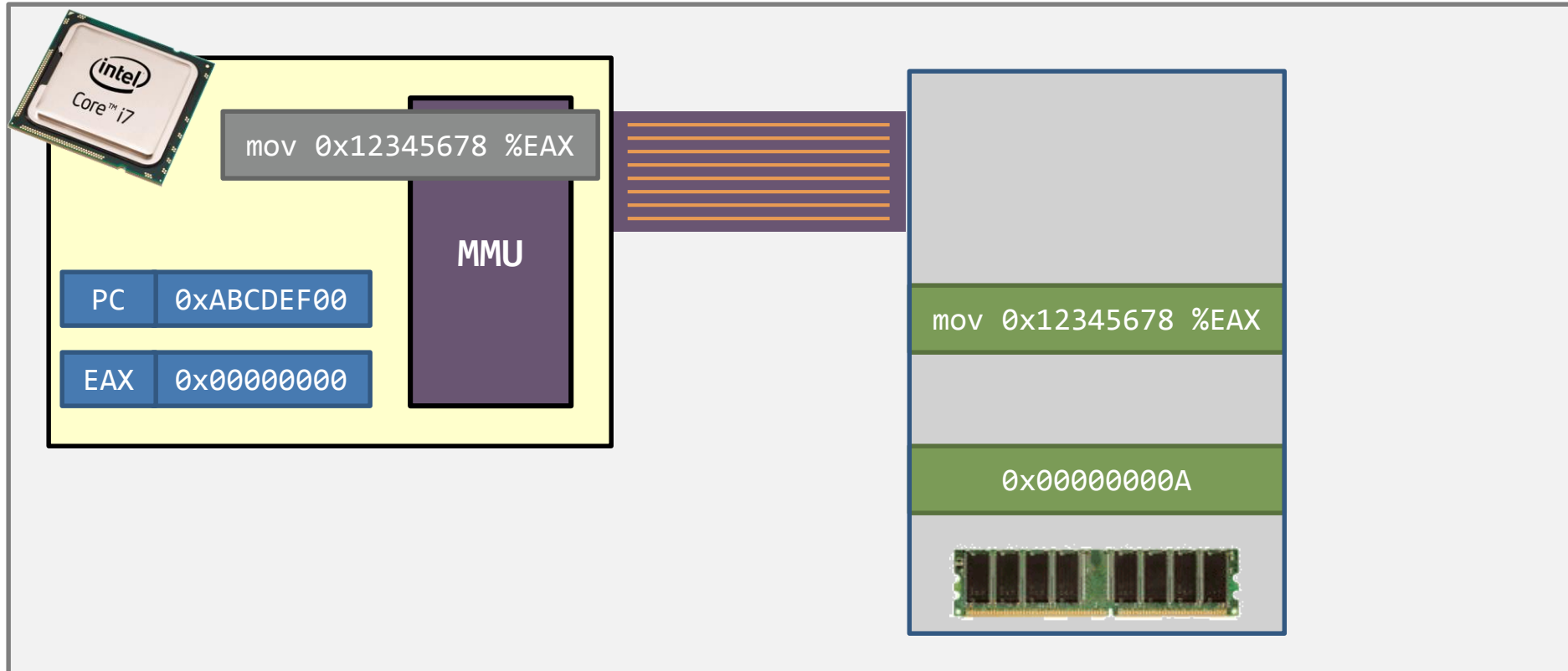
# Virtual memory – how does it work?

- ◆ Step 2. The memory returns the instruction



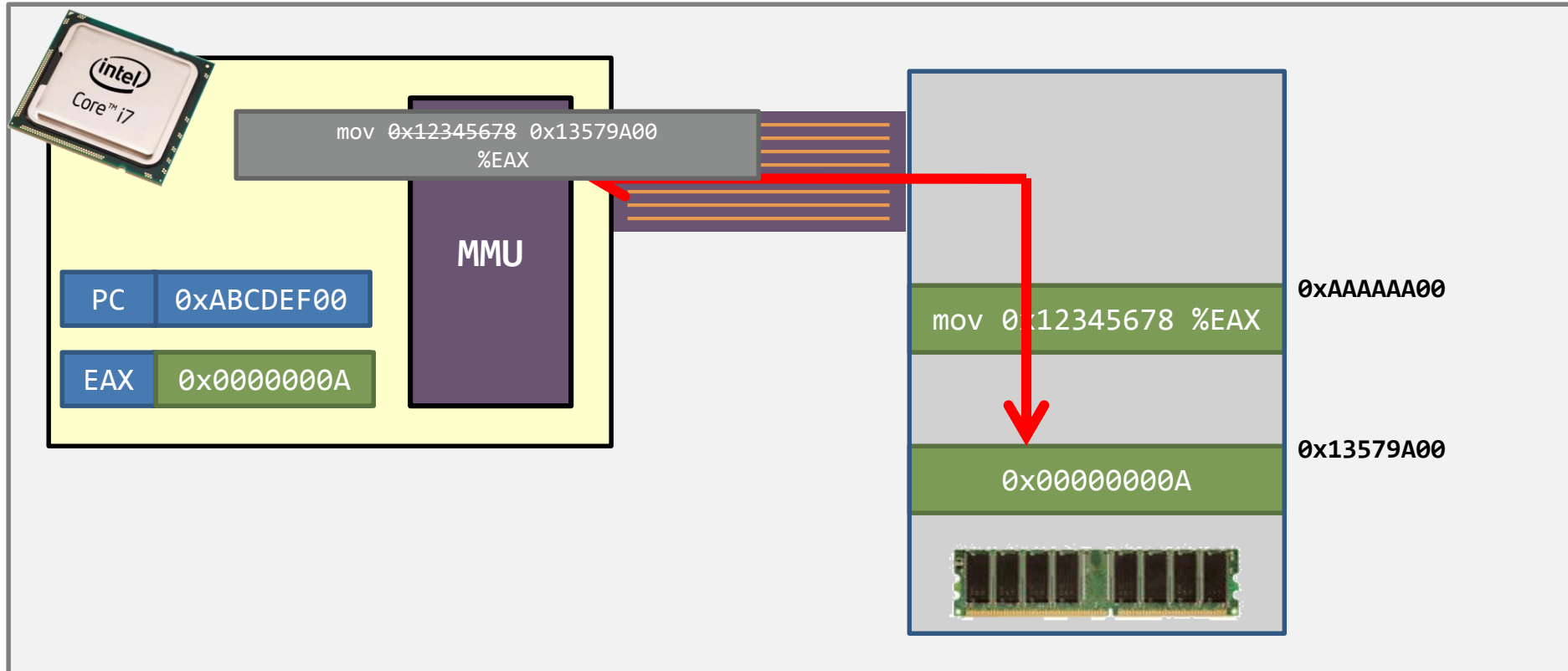
# Virtual memory – how does it work?

- ◆ Step 3. The CPU decodes the instruction.
  - ◆ An instruction **uses virtual addresses**
    - ◆ but not physical addresses.



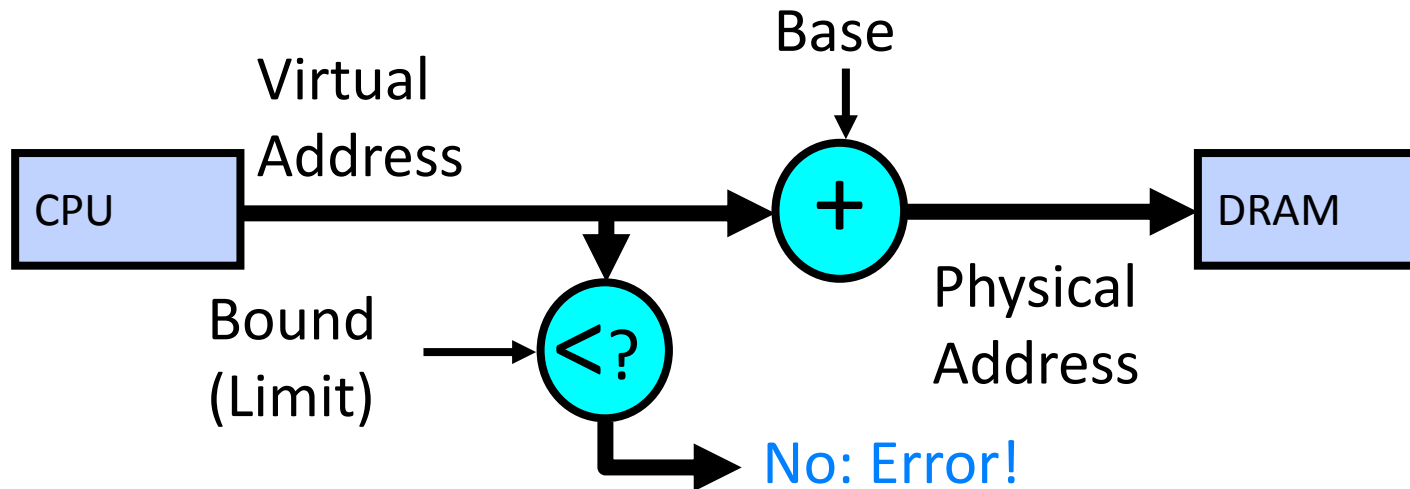
# Virtual memory – how does it work?

- Step 4. With the help of the MMU, the target memory is retrieved.



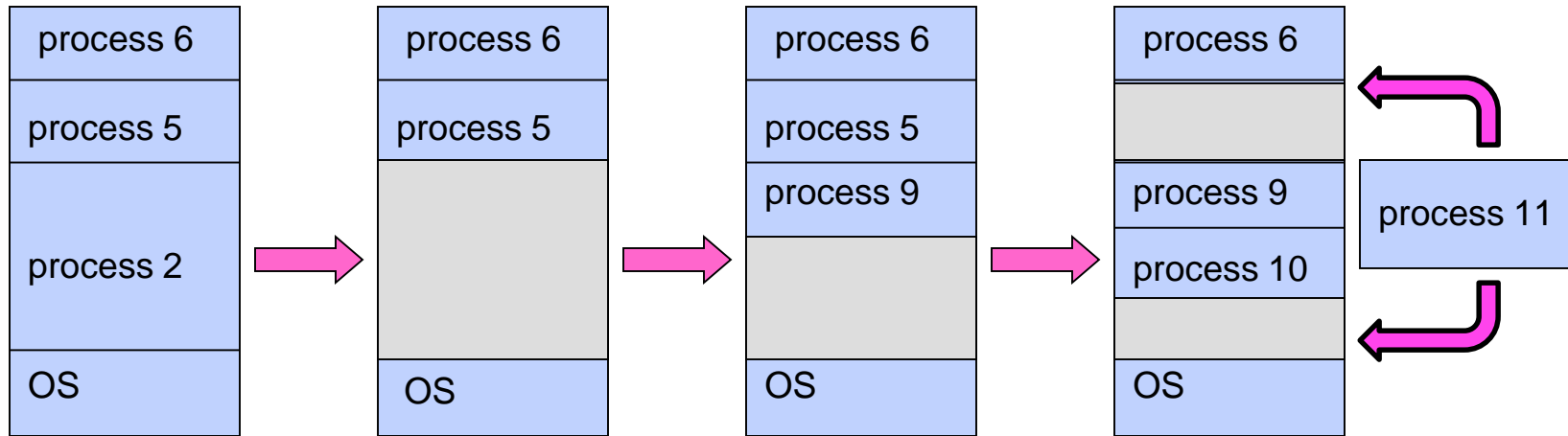


# Simple Example: Base and Bounds



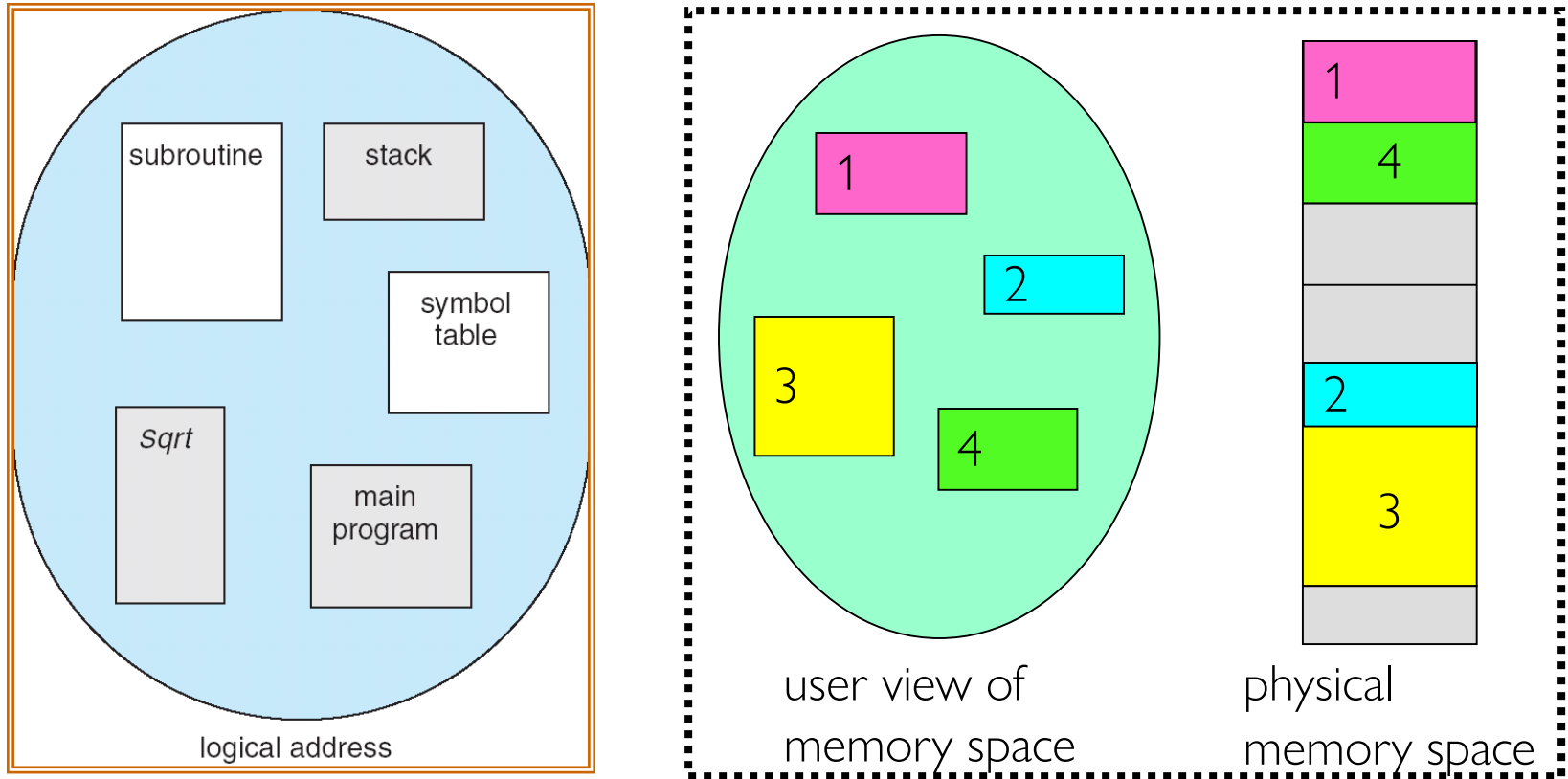
- ❖ Could use base/bounds for **dynamic address translation** – translation happens at execution:
  - ❖ Alter address of every load/store by adding “base”
  - ❖ Generate error if address bigger than limit
- ❖ This gives program the illusion that it is running on its own dedicated machine, with memory starting at 0
  - ❖ Program gets continuous region of memory
  - ❖ Addresses within program do not have to be relocated when program placed in different region of DRAM

# Issues with Simple B&B Method



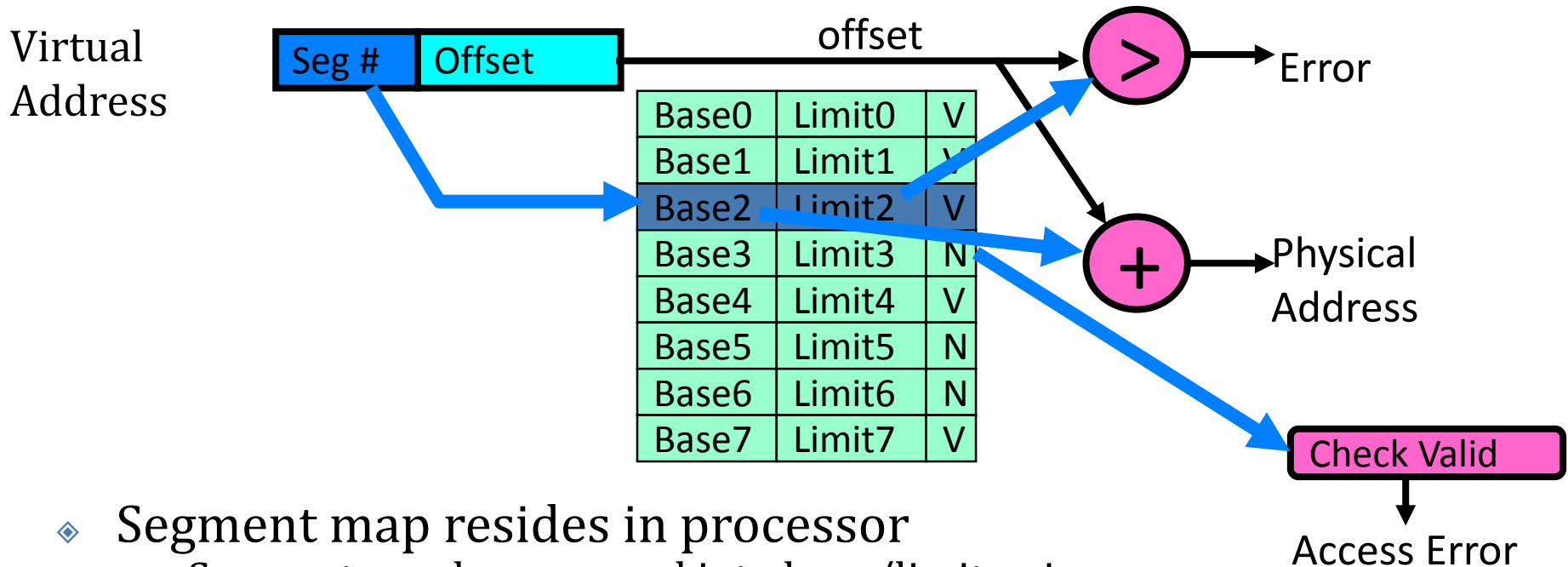
- ❖ Fragmentation problem over time
  - ❖ Not every process is same size → memory becomes fragmented
- ❖ Missing support for sparse address space
  - ❖ Would like to have multiple chunks/program (Code, Data, Stack)
- ❖ Hard to do inter-process sharing
  - ❖ Want to share code segments when possible
  - ❖ 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



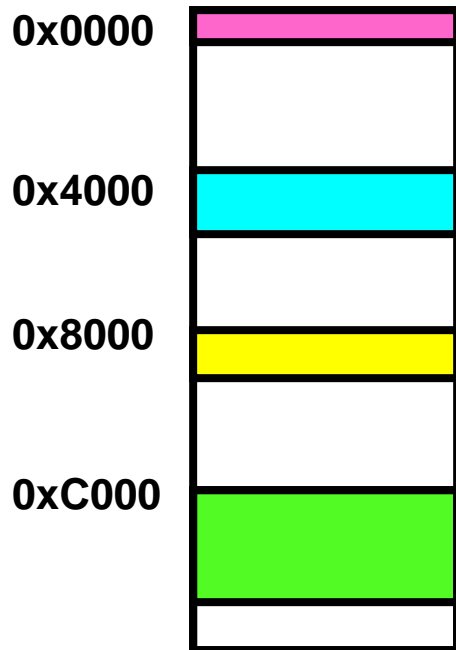
- ◆ Segment map resides in processor
  - ◆ Segment number mapped into base/limit pair
  - ◆ Base added to offset to generate physical address
  - ◆ Error check catches offset out of range
- ◆ As many chunks of physical memory as entries
  - ◆ Segment addressed by portion of virtual address
  - ◆ However,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

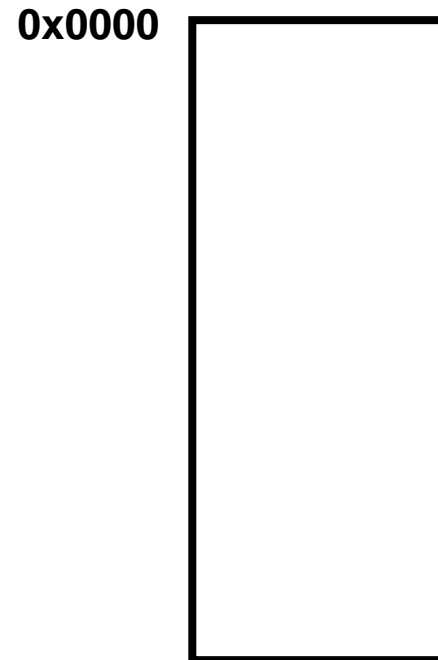


Virtual Address Format

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



Virtual  
Address Space



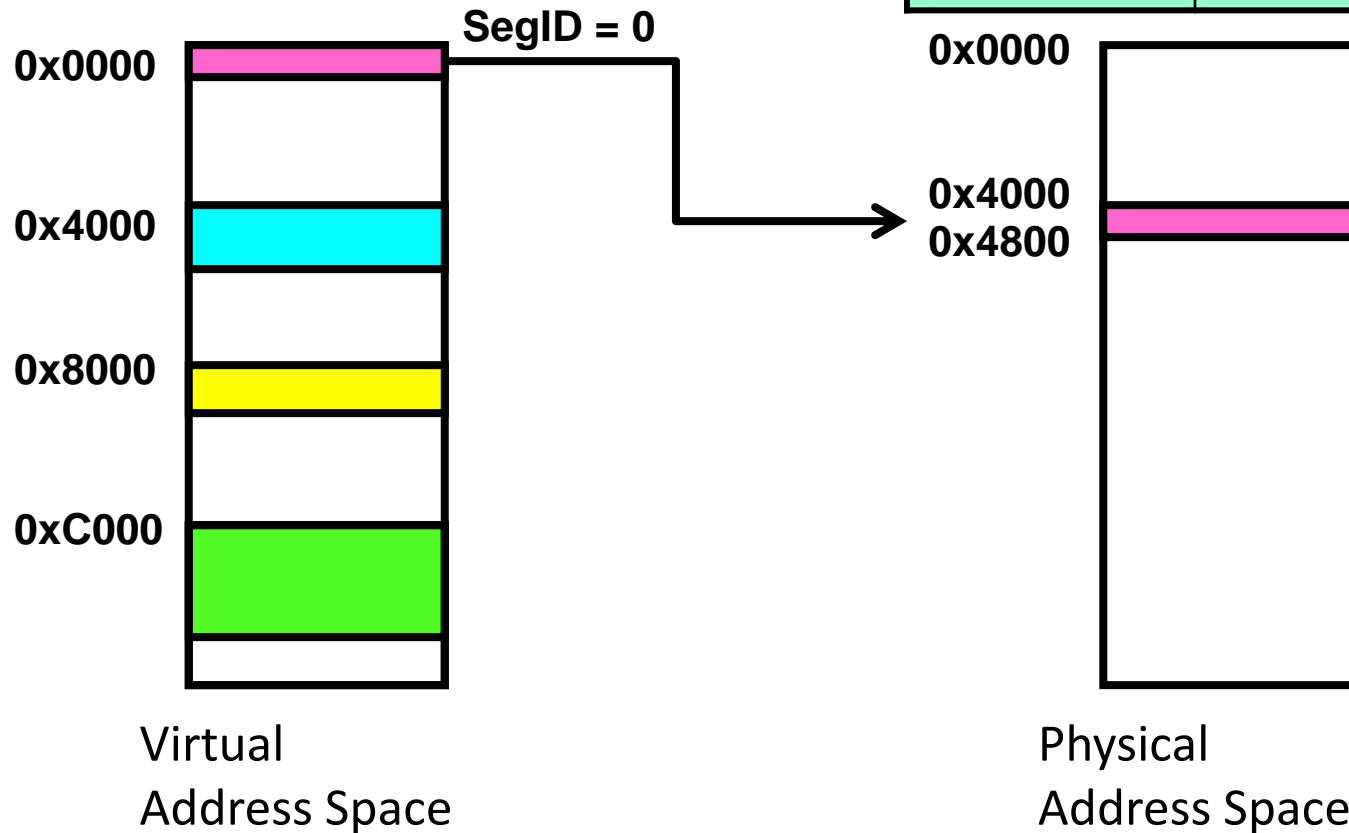
Physical  
Address Space

# 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

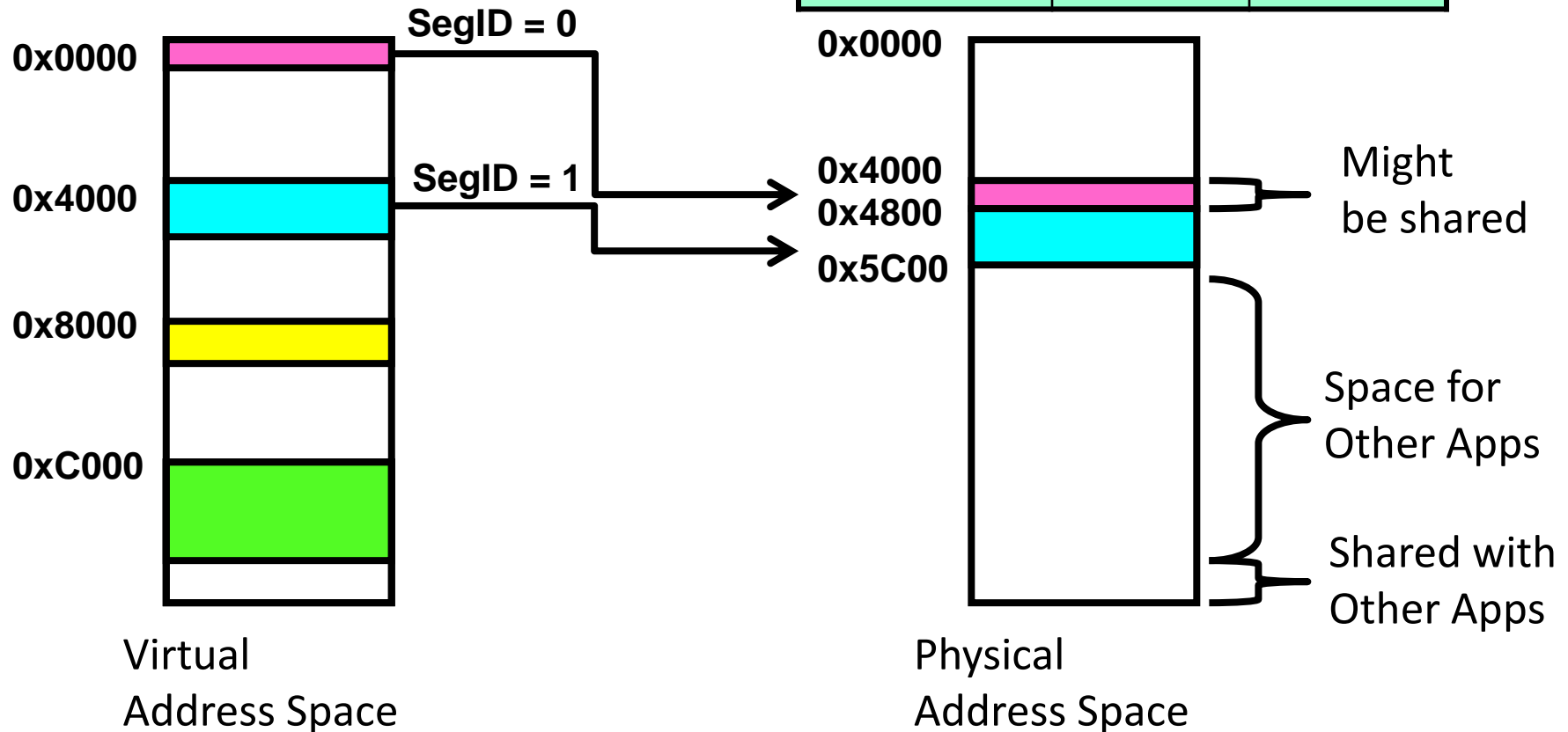


# Example: Four Segments (16 bit addresses)



Virtual Address Format

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

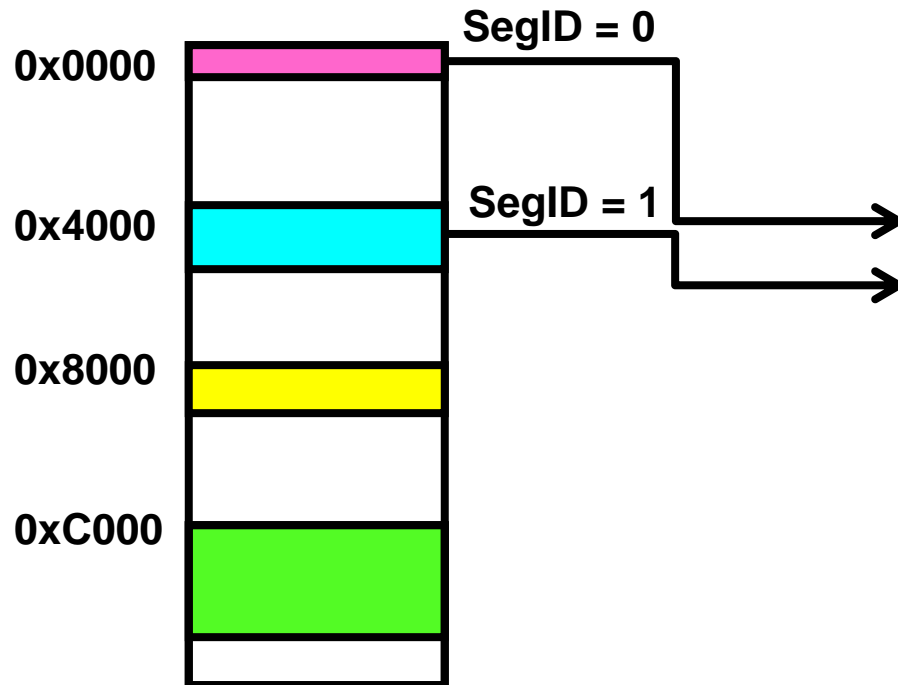


# Example: Four Segments (16 bit addresses)

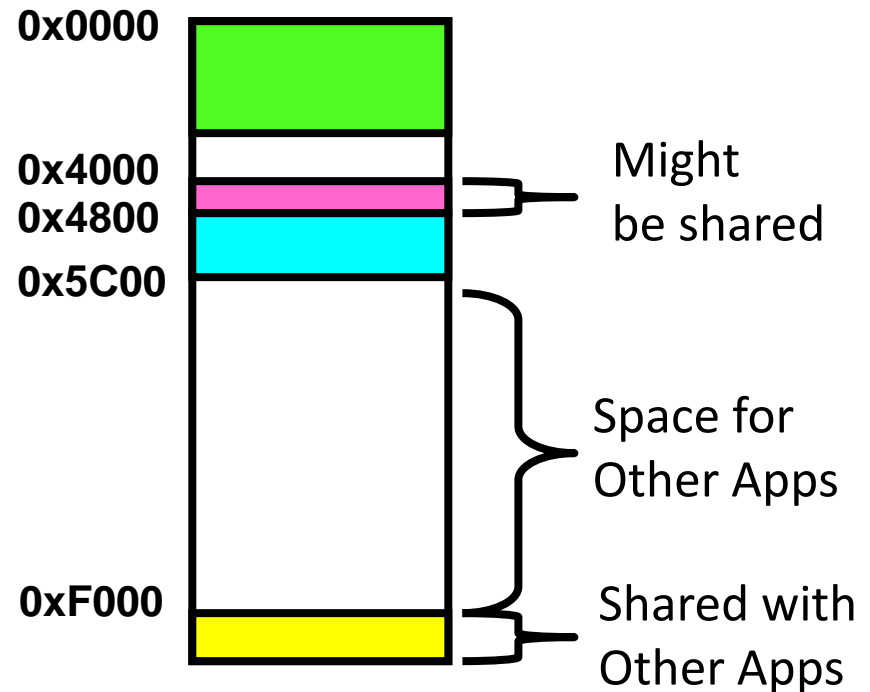


Virtual Address Format

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



Virtual Address Space



Physical Address Space



# Example of Segment Translation (16b address)

0x240	main:	la \$a0, varx
0x244		jal strlen
...		...
0x360	strlen:	li \$v0, 0 ;count
0x364	loop:	lb \$t0, (\$a0)
0x368		beq \$r0,\$t0, done
...		...
0x4050	varx	dw 0x314159

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

Let us simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x240. Virtual segment #? 0; Offset? 0x240

Physical address? Base=0x4000, so physical addr=0x4240

Fetch instruction at 0x4240. Get "la \$a0, varx"

Move 0x4050 → \$a0, Move PC+4→PC

# Example of Segment Translation (16b address)

0x240	main:	la	\$a0, varx
0x244		jal	strlen
...			...
0x360	strlen:	li	\$v0, 0 ;count
0x364	loop:	lb	\$t0, (\$a0)
0x368		beq	\$r0,\$t0, done
...			...
0x4050	varx	dw	0x314159

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

Let us simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x240. Virtual segment #? 0; Offset? 0x240

Physical address? Base=0x4000, so physical addr=0x4240

Fetch instruction at 0x4240. Get "la \$a0, varx"

Move 0x4050 → \$a0, Move PC+4 → PC

2. Fetch 0x244. Translated to Physical=0x4244. Get "jal strlen"

Move 0x0248 → \$ra (return address!), Move 0x0360 → PC

# Example of Segment Translation (16b address)

0x240	main:	la \$a0, varx
0x244		jal strlen
...		...
0x360	strlen:	li \$v0, 0 ;count
0x364	loop:	lb \$t0, (\$a0)
0x368		beq \$r0,\$t0, done
...		...
0x4050	varx	dw 0x314159

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

Let us simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x240. Virtual segment #? 0; Offset? 0x240  
Physical address? Base=0x4000, so physical addr=0x4240  
Fetch instruction at 0x4240. Get "la \$a0, varx"  
**Move 0x4050 → \$a0, Move PC+4→PC**
2. Fetch 0x244. Translated to Physical=0x4244. Get "jal strlen"  
**Move 0x0248 → \$ra (return address!), Move 0x0360 → PC**
3. Fetch 0x360. Translated to Physical=0x4360. Get "li \$v0, 0"  
**Move 0x0000 → \$v0, Move PC+4→PC**

# Example of Segment Translation (16b address)

0x240	main:	la \$a0, varx
0x244		jal strlen
...		...
0x360	strlen:	li \$v0, 0 ;count
0x364	loop:	lb \$t0, (\$a0)
0x368		beq \$r0,\$t0, done
...		...
0x4050	varx	dw 0x314159

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

Let us simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x240. Virtual segment #? 0; Offset? 0x240  
Physical address? Base=0x4000, so physical addr=0x4240  
Fetch instruction at 0x4240. Get “la \$a0, varx”  
Move 0x4050 → \$a0, Move PC+4→PC
2. Fetch 0x244. Translated to Physical=0x4244. Get “jal strlen”  
Move 0x0248 → \$ra (return address!), Move 0x0360 → PC
3. Fetch 0x360. Translated to Physical=0x4360. Get “li \$v0, 0”  
Move 0x0000 → \$v0, Move PC+4→PC
4. Fetch 0x364. Translated to Physical=0x4364. Get “lb \$t0, (\$a0)” Since \$a0 is 0x4050, try to load byte from 0x4050, Translate 0x4050 (0100 0000 0101 000). Virtual segment #? 1; Offset? 0x50 Physical address? Base=0x4800, Physical addr = 0x4850, Load Byte from 0x4850→\$t0, Move PC+4→PC

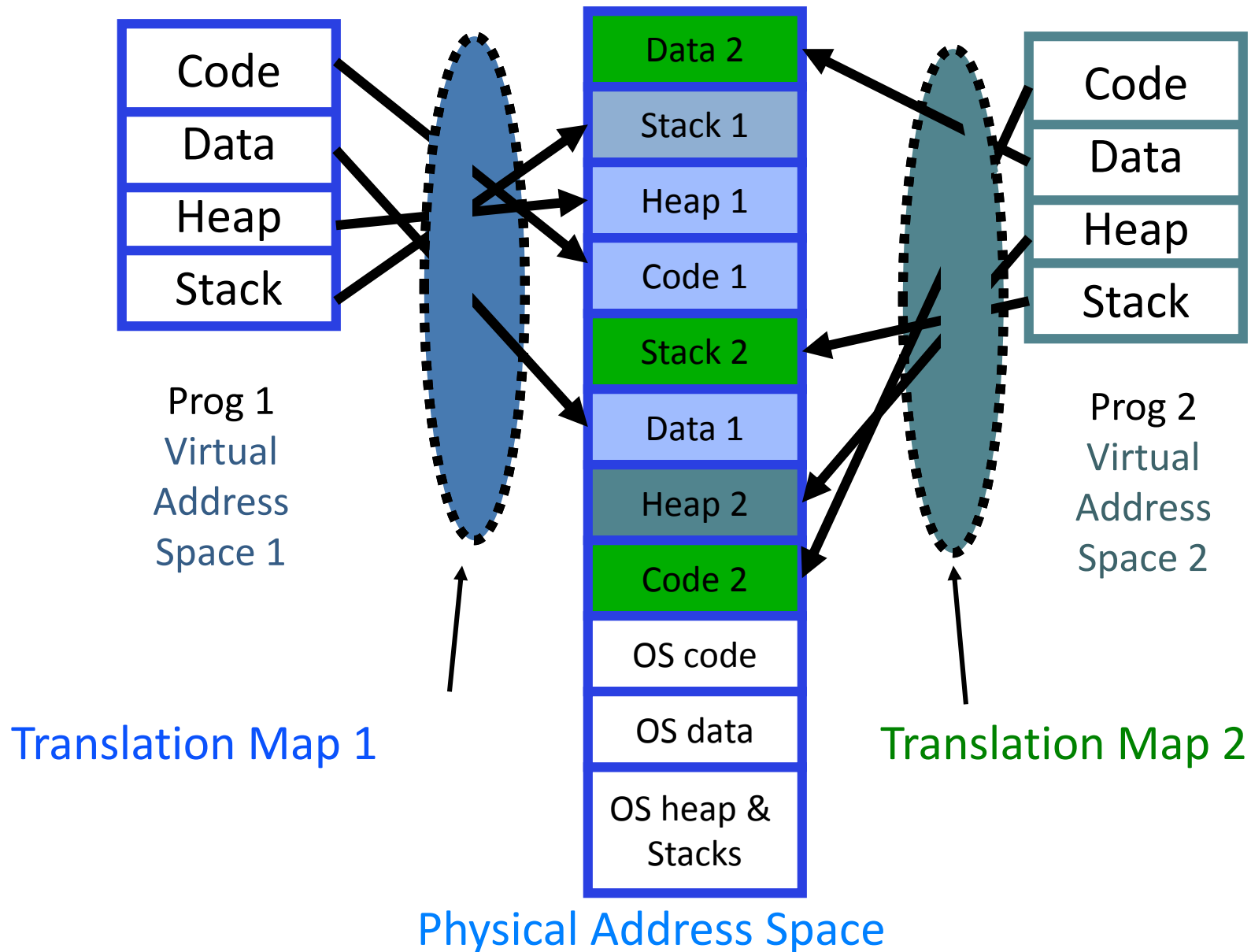
# Observations about Segmentation

- ◆ Virtual address space has holes
  - ◆ Segmentation efficient for sparse address spaces
  - ◆ A correct program should never address gaps
    - ◆ If it does, trap to kernel and dump core
- ◆ When it is OK to address outside valid range?
  - ◆ This is how the stack and heap are 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)
  - ◆ Shared segment could be read-only or read-write
- ◆ 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”)

# 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**: do not need all memory within allocated chunks

# General Address Translation

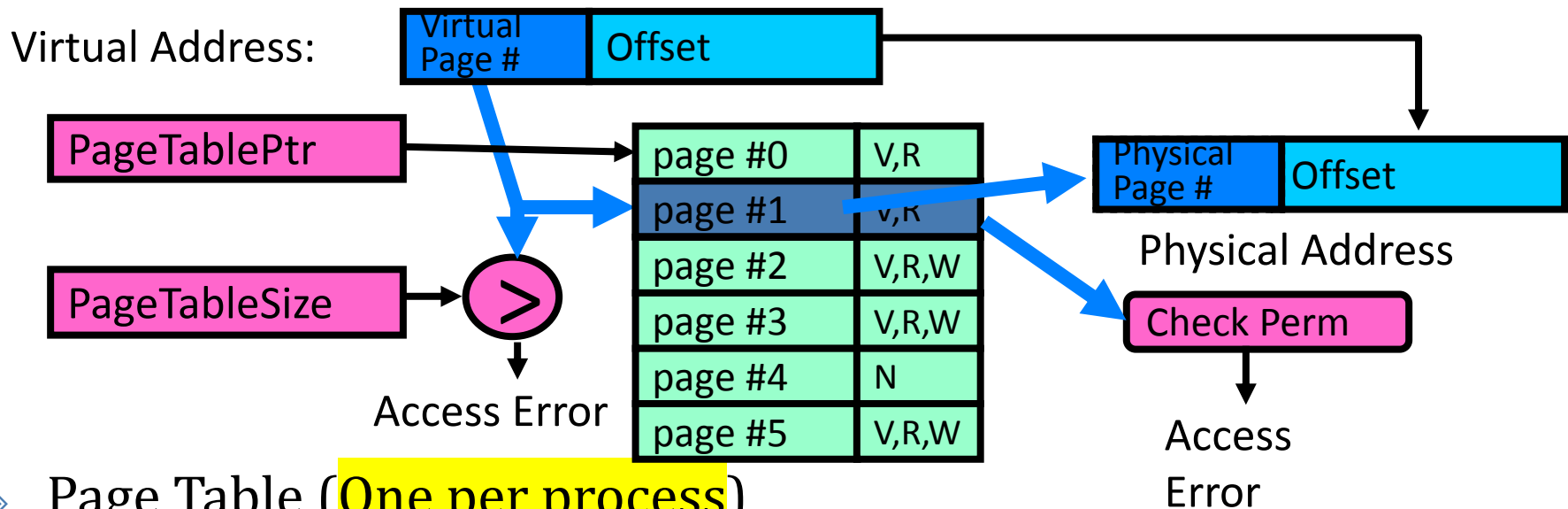


# Paging: Physical Memory in Fixed Size Chunks

- ◆ 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



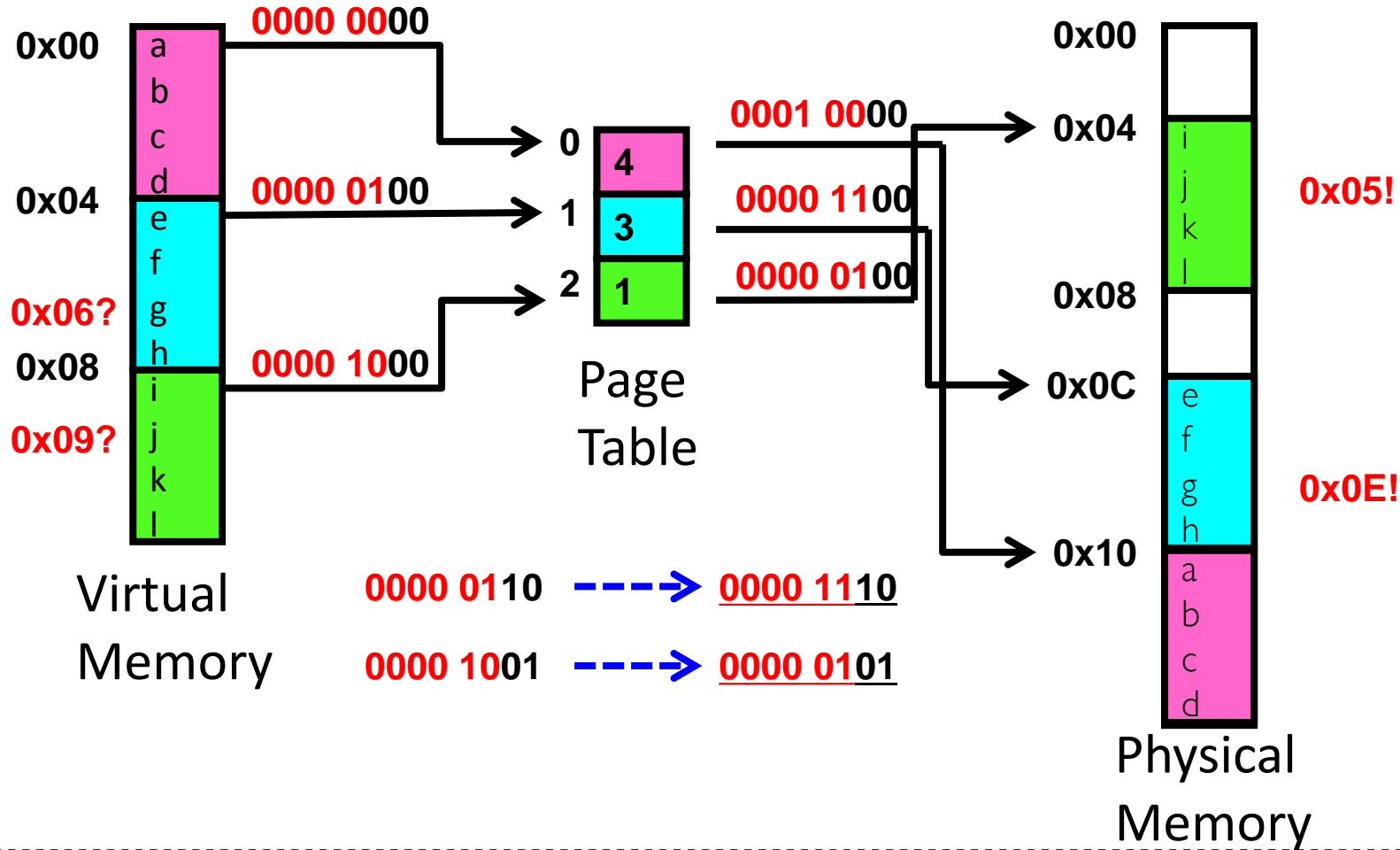
# How to Implement Paging?



- ◆ Page Table (**One per process**)
  - ◆ Resides in physical memory
  - ◆ Contains physical page and permission for each virtual page
    - ◆ Permissions include: 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 (write pages)



# What about Sharing?

Virtual Address  
(Process A):

PageTablePtrA

Virtual Page #	Offset
----------------	--------

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

PageTablePtrB

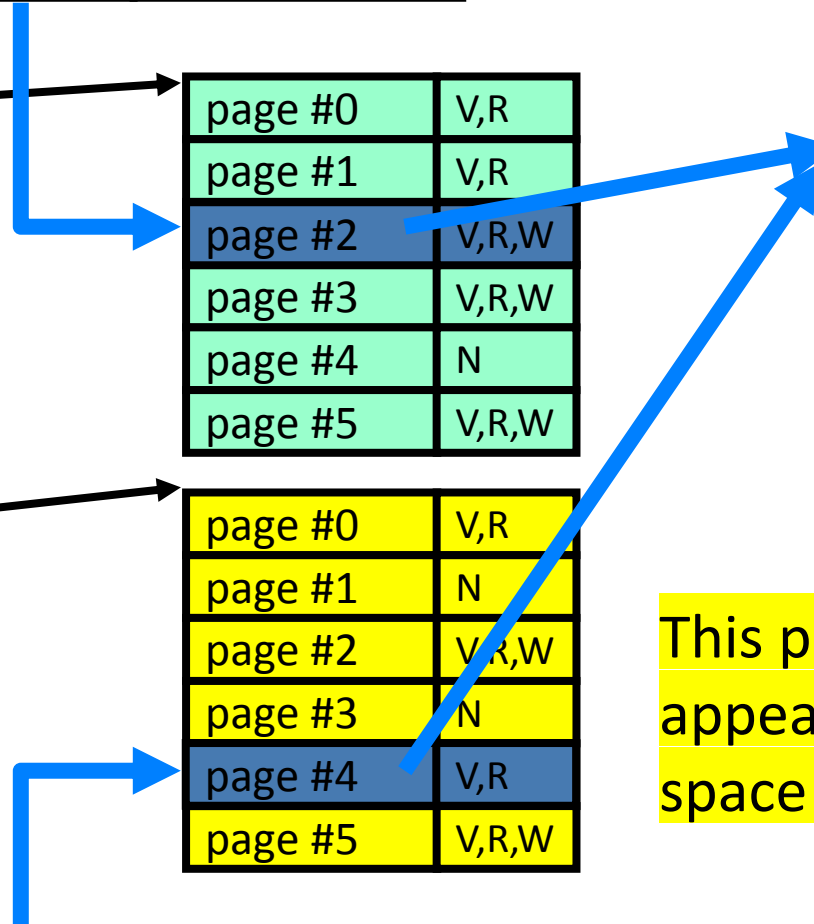
page #0	V,R
page #1	N
page #2	V,R,W
page #3	N
page #4	V,R
page #5	V,R,W

Virtual Address  
(Process B):

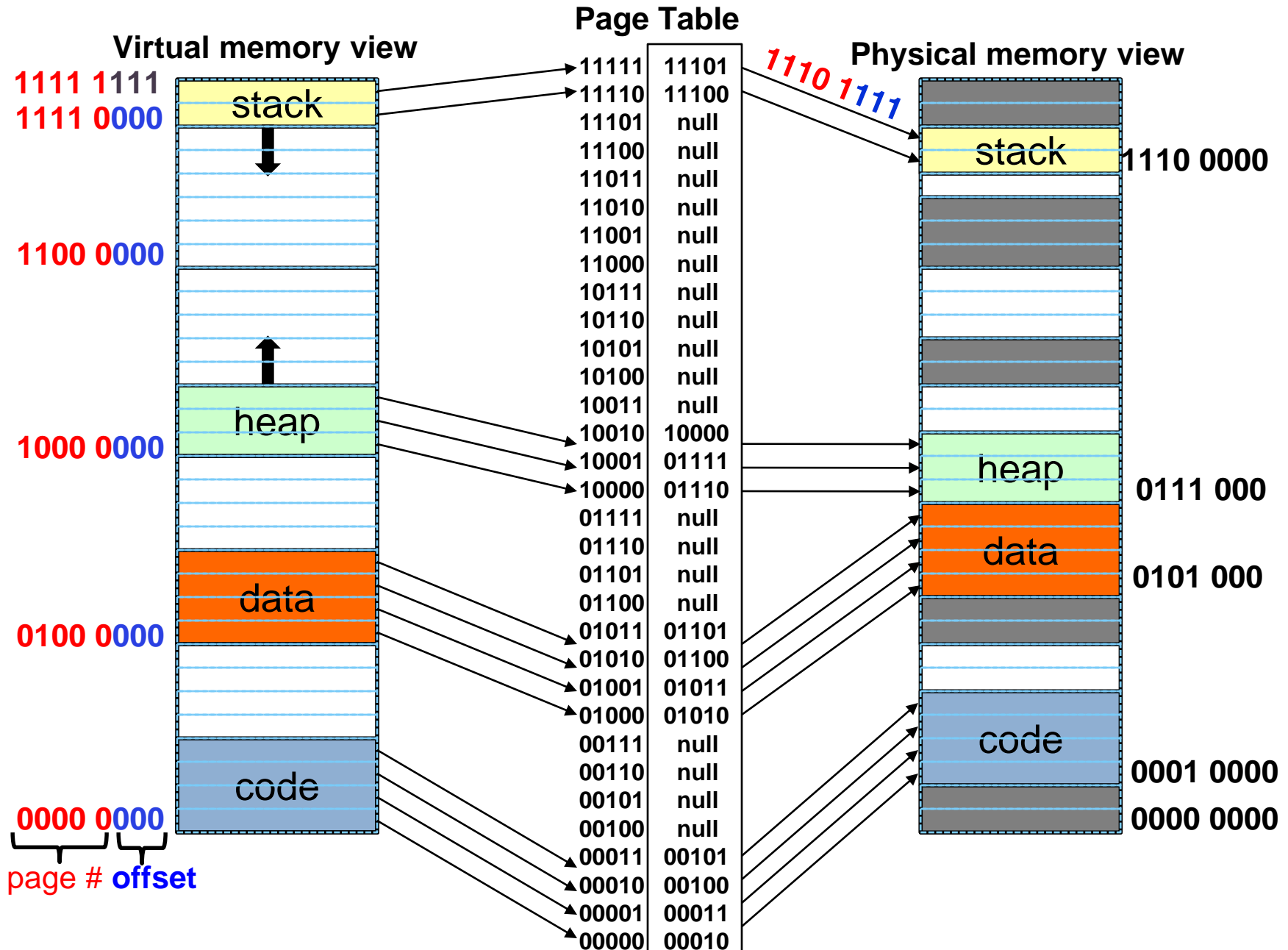
Virtual Page #	Offset
----------------	--------

Shared Page

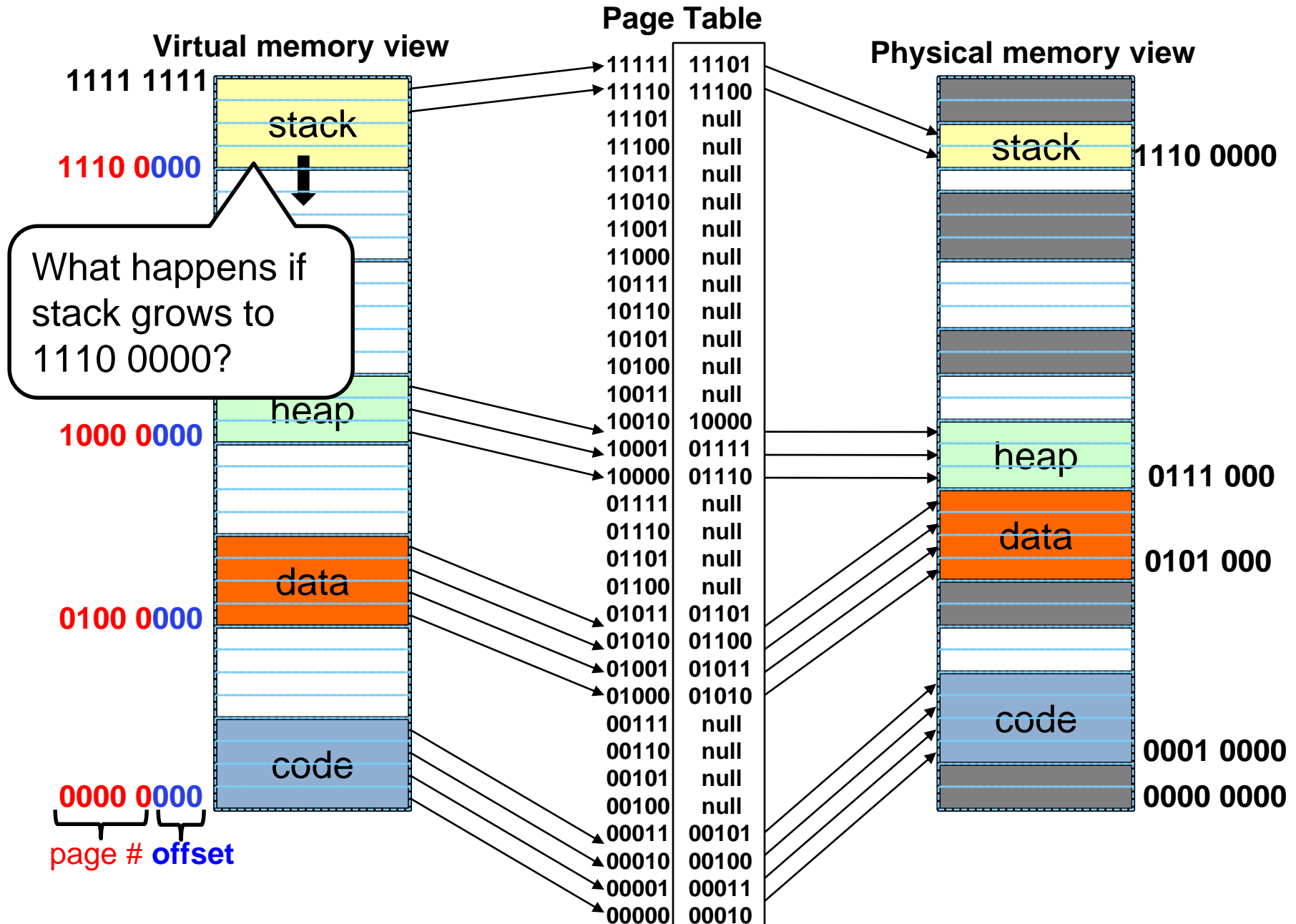
This physical page appears in address space of both processes



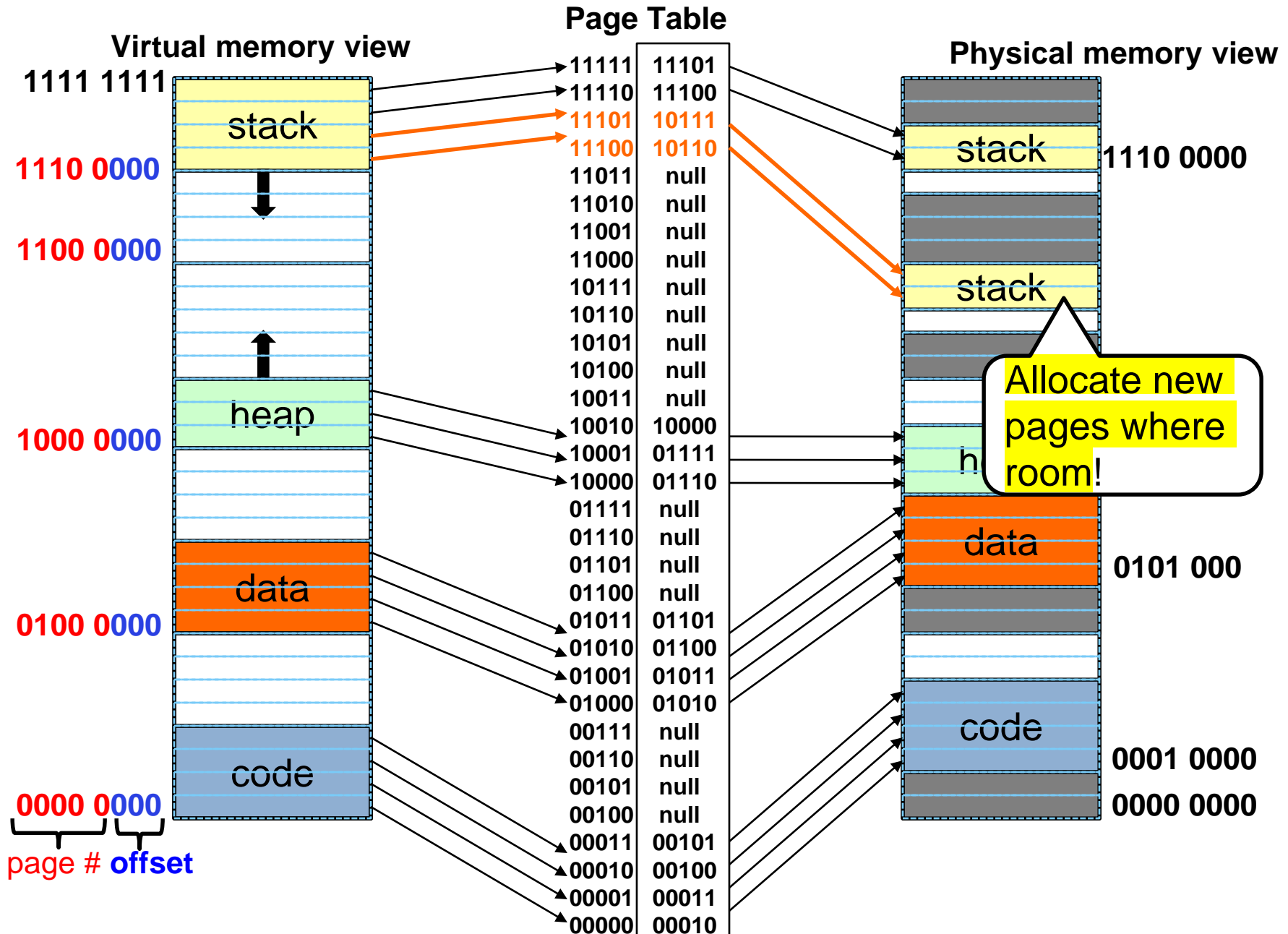
# Summary: Paging



# Summary: Paging



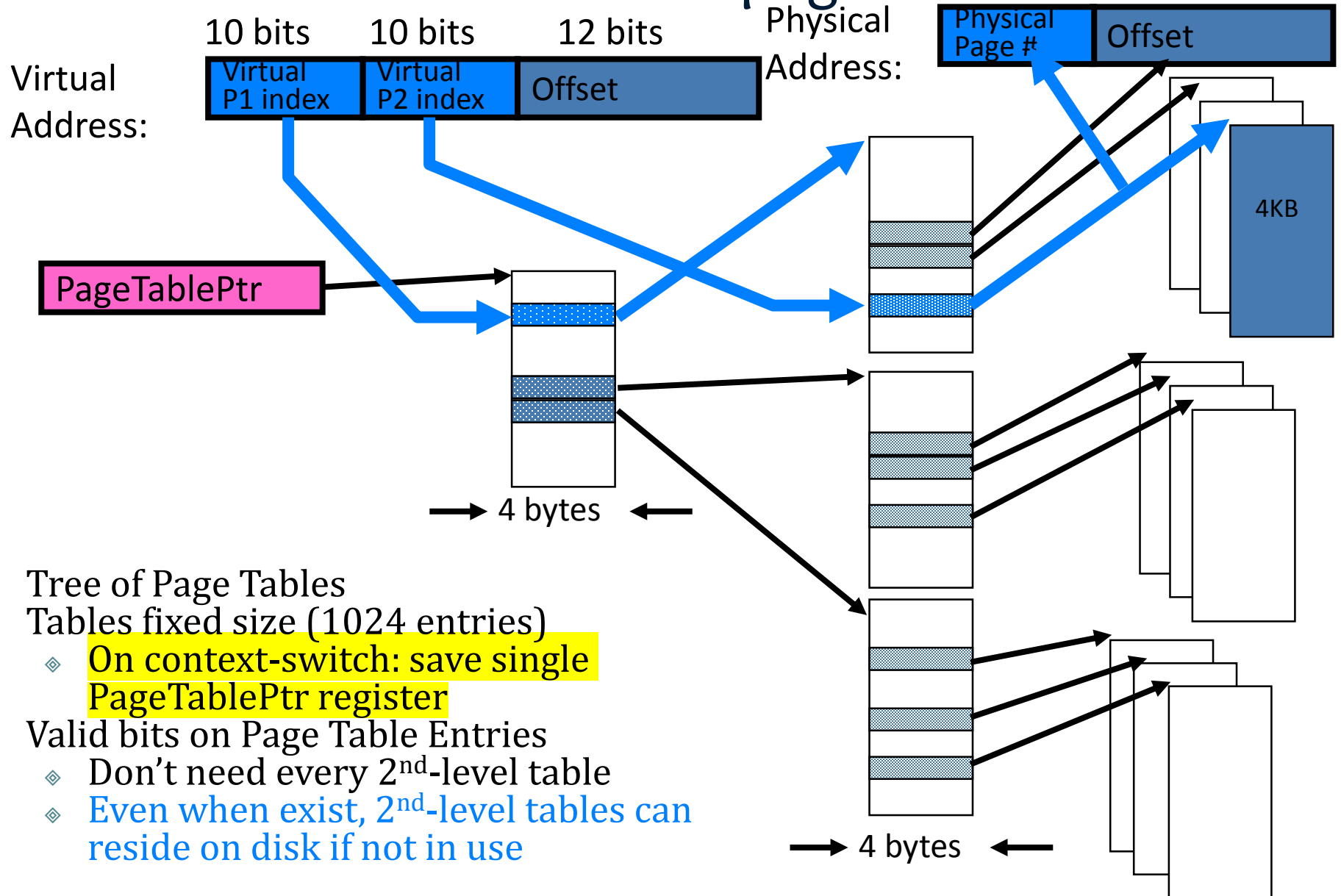
# Summary: Paging



# Page Table Discussion

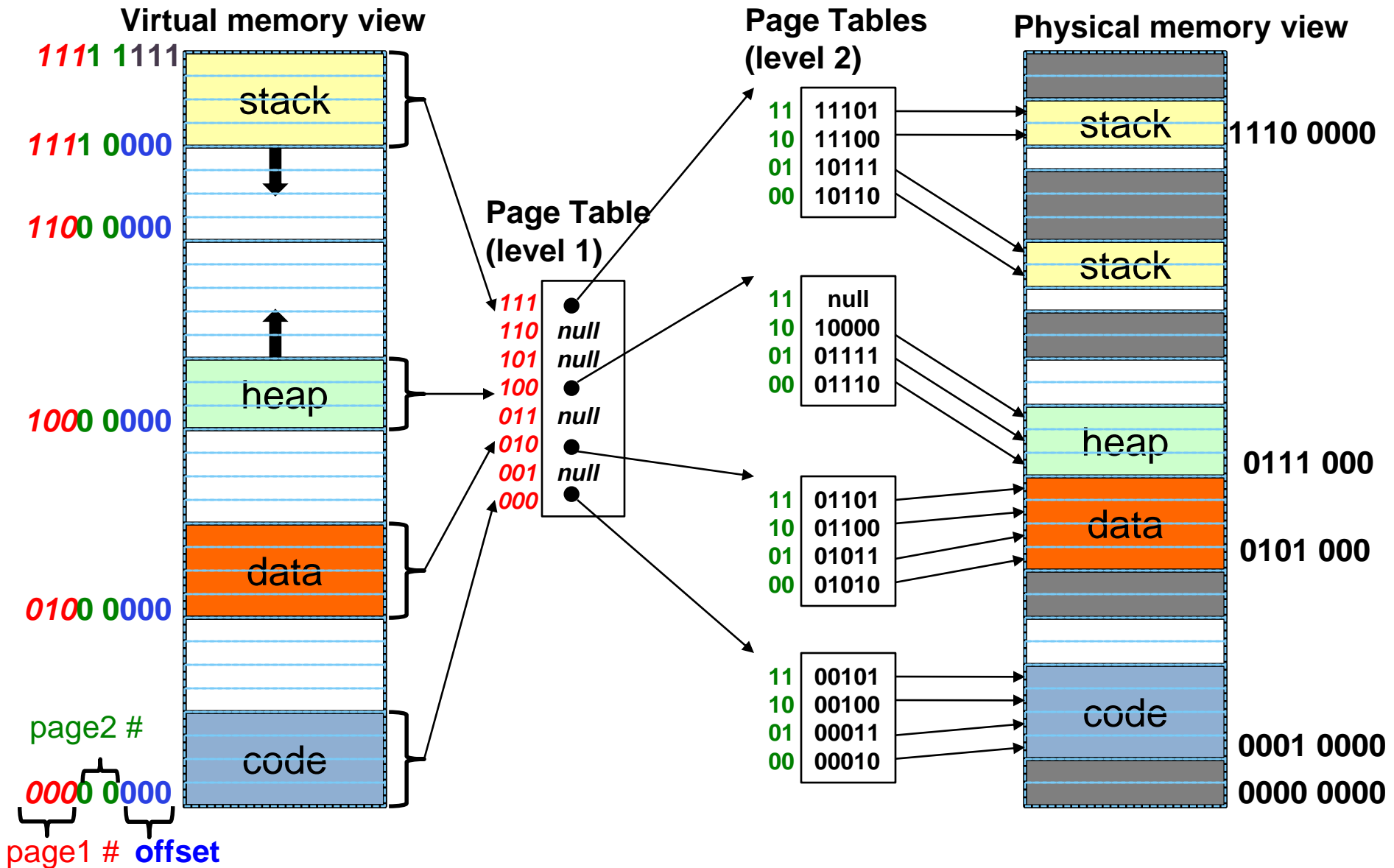
- ◆ What needs to be switched on a context switch?
  - ◆ Page table pointer and limit
- ◆ Analysis
  - ◆ Pros
    - ◆ Simple memory allocation
    - ◆ Easy to share
  - ◆ Con: What if address space is sparse?
    - ◆ E.g., on UNIX, code starts at 0, stack starts at  $(2^{31}-1)$
    - ◆ With 1K pages, need 2 million page table entries!
  - ◆ Con: What if table really big?
    - ◆ Not all pages used all the time  $\Rightarrow$  would be nice to have working set of page table in memory
- ◆ How about multi-level paging or combining paging and segmentation?

# Fix for sparse address space: The two-level page table

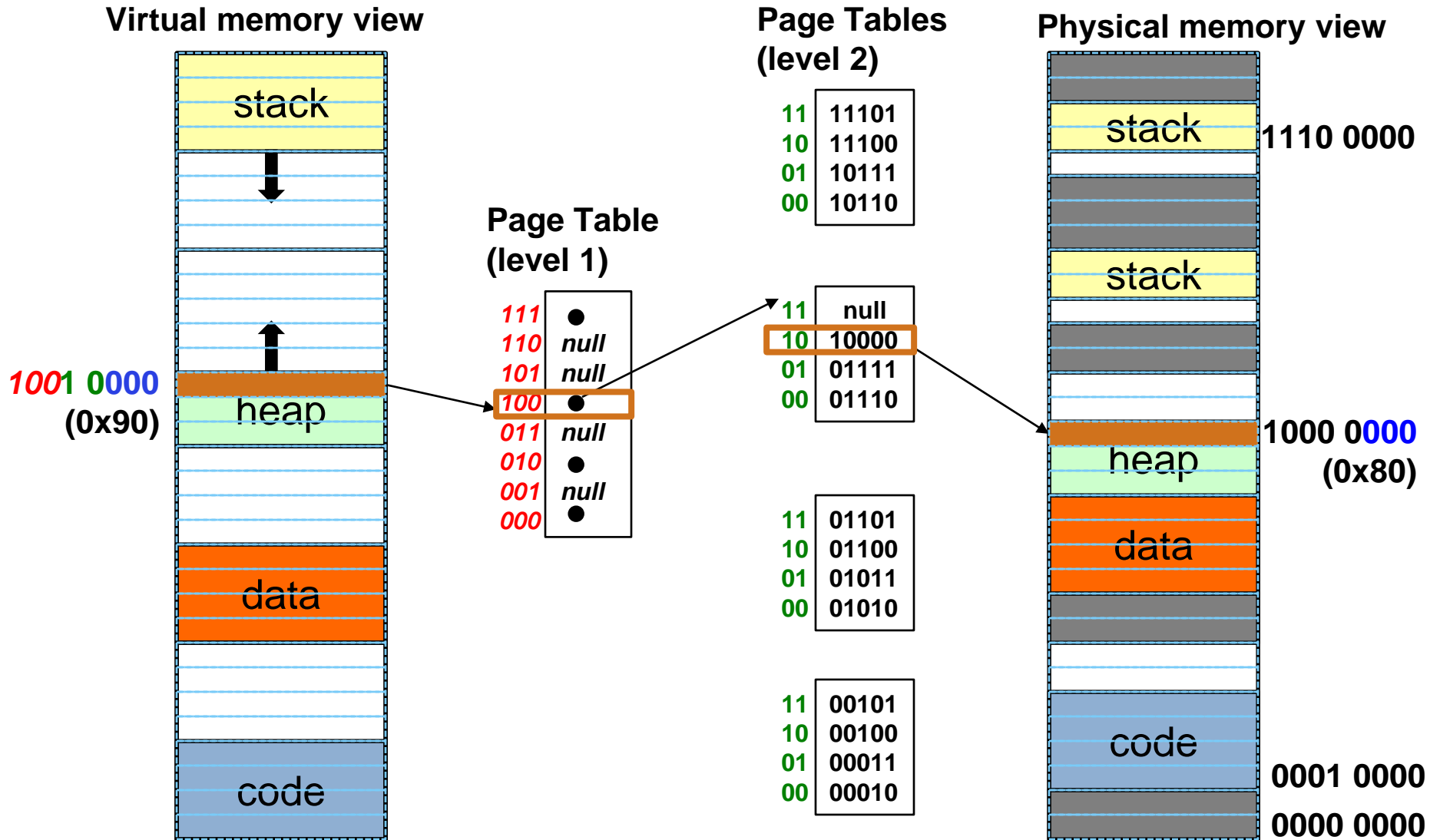




# Summary: Two-Level Paging

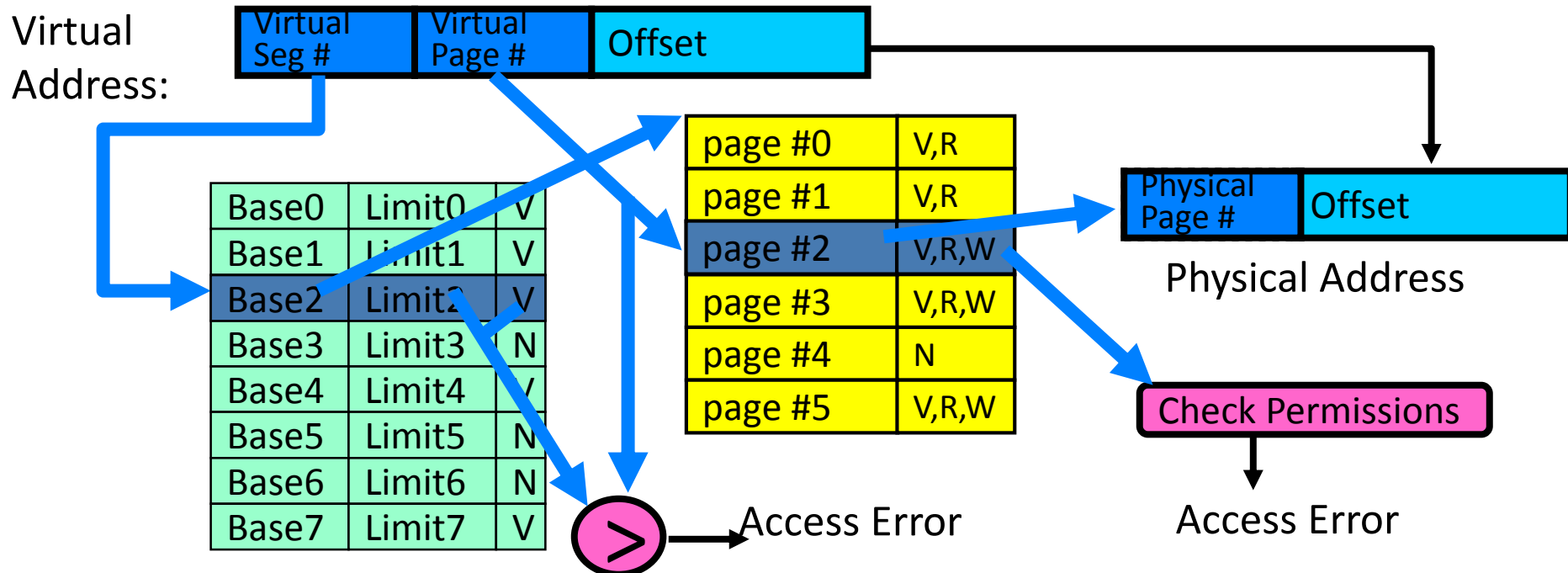


# Summary: Two-Level Paging



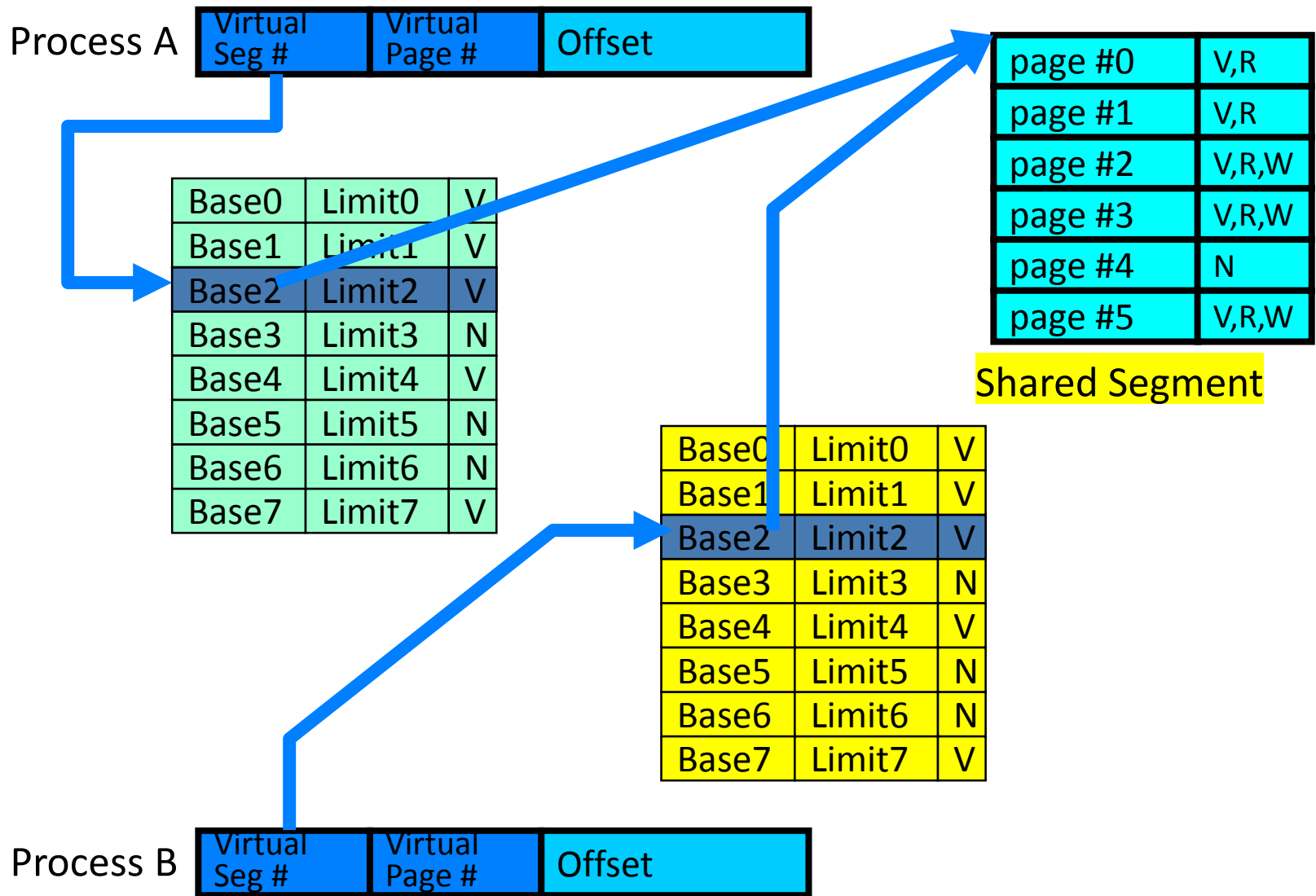
# Multi-level Translation: Segments + Pages

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



- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)

# What about Sharing (Complete Segment)?



Thank You!