

**ECE 350**  
**Real-time**  
**Operating**  
**Systems**



# Lecture 7: Address Translation

---

Prof. Seyed Majid Zahedi

<https://ece.uwaterloo.ca/~smzahedi>

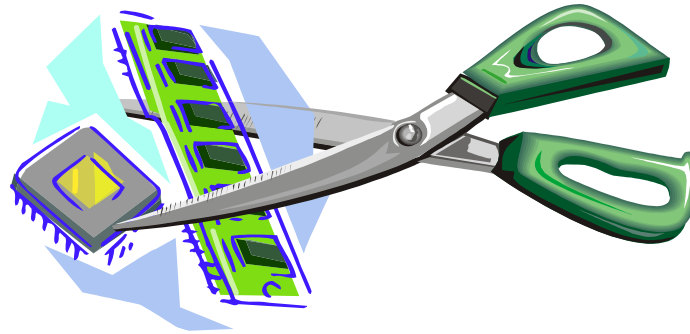
# Outline

---

- Virtual to physical address translation
  - Base and bound
  - Segmentation
  - Page table
  - Multi-level table
  - Inverted page table

# Recall: OS as Illusionist and Referee

---



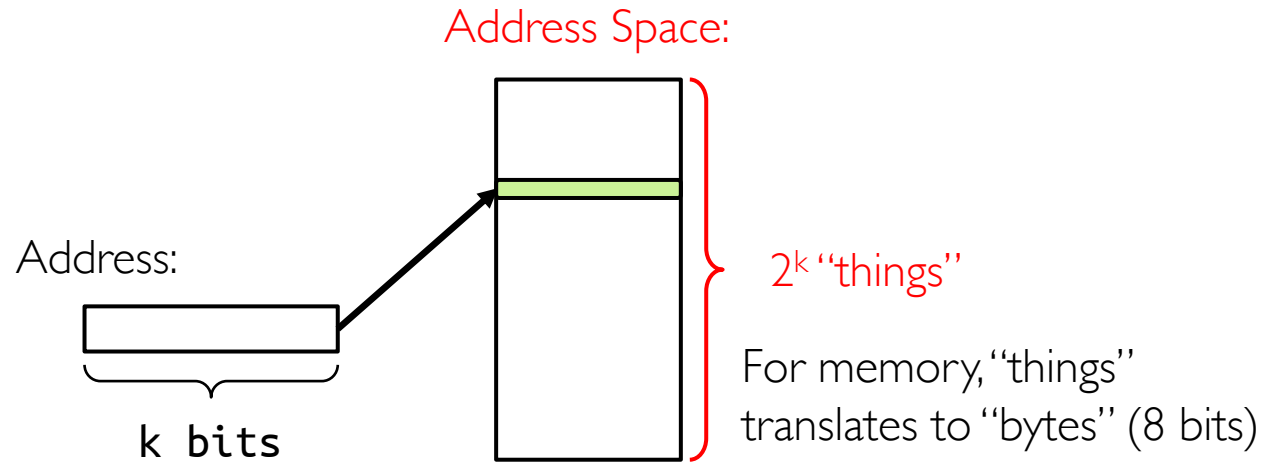
- **Illusion**: each process has its own processor with (almost) infinite memory capacity
- **Physical reality**: there are only few processes, memory capacity is limited
- **Scheduling**: need to multiplex processors (done)
- **Memory management**: need to multiplex memory (now!)

# Memory Management Goals

---

- **Protection**: prevent processes/threads from accessing others' private data
  - Protect kernel data from user programs
  - Protect programs from themselves
  - Give special access permissions to different data
  - Allow processes to share data (**controlled overlap**)
    - E.g., Shared binary file between multiple processes (e.g., `fork()`)
    - E.g., Shared memory used for inter-process communication
    - E.g., Memory-mapped file shared by multiple processes
    - E.g., User-level system libraries
- **Allocation**: divide available physical memory among processes/threads
  - Manage memory capacity efficiently
  - Avoid memory fragmentation
  - Evict memory blocks to persistent storage if needed

# Background: Some Basics



- What is  $2^{10}$  bytes (where one byte is abbreviated as “B”)?
  - $2^{10} \text{ B} = 1024\text{B} = 1\text{KiB}$  (for memory,  $1\text{KiB} = 1024\text{B}$ , not  $1000\text{B}$ )
- How many bits to address each byte of  $4\text{KiB}$  memory?
  - $4\text{KiB} = 4 \times 1\text{KiB} = 4 \times 2^{10} = 2^{12} \Rightarrow 12 \text{ bits}$
- How much memory can be addressed with 20 bits? 32 bits? 64 bits?
  - $2^{20}\text{B} = 2^{10}\text{KiB} = 1\text{MiB}$  (mebibyte)
  - $2^{32}\text{B} = 2^{12}\text{MiB} = 2^2\text{GiB}$  (gibibyte)
  - $2^{64}\text{B} = 2^{34}\text{GiB} = 2^{24}\text{TiB}$  (tebibyte) =  $2^{14}\text{PiB}$  (pebibyte) =  $2^4\text{EiB}$  (exbibyte)

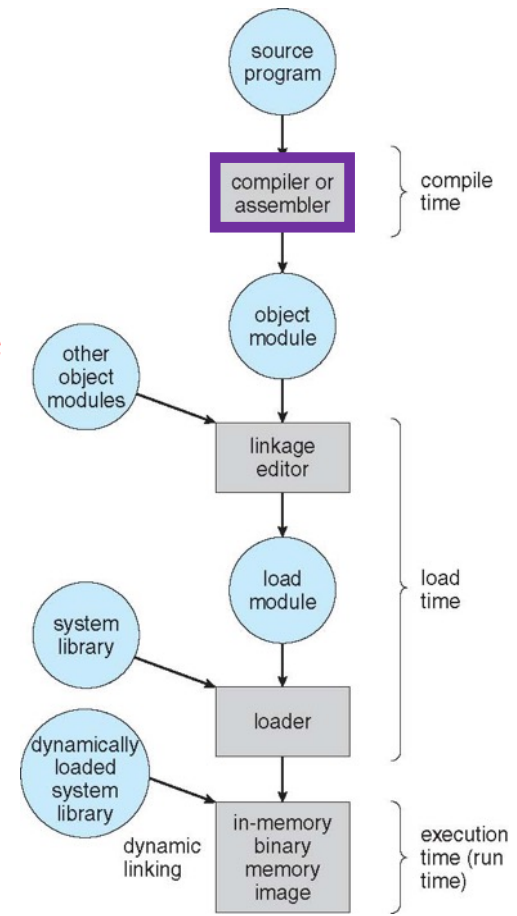
# Recall: Some Terminologies

---

- **Address space**: set of accessible addresses and their state
- **Physical memory**: data storage medium
- **Physical addresses**: addresses available on physical memory
  - For 4GiB of memory:  $2^{32}\text{B} \sim 4$  billion addresses
- **Virtual addresses**: addresses generated by program
  - For 64-bit processor:  $2^{64} > 18$  quintillion ( $10^{18}$ ) addresses

# Multi-step Processing of Programs

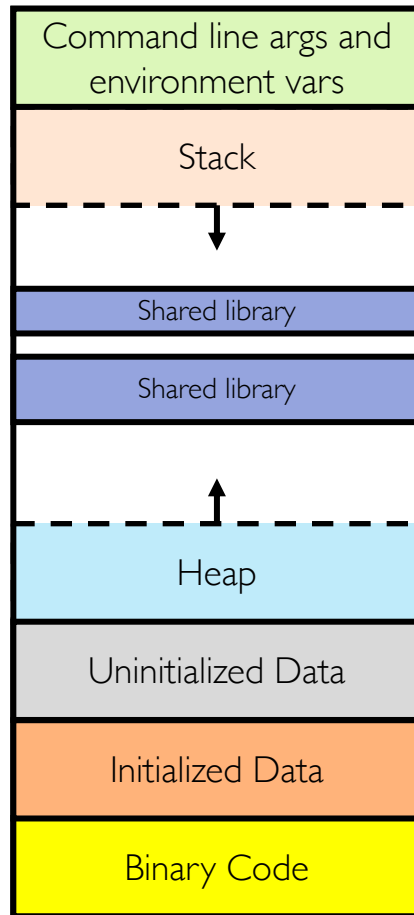
- Compiler: generate **object file** for each source code
  - Has incomplete information when compiling each source code
  - Doesn't know addresses of external objects (e.g., **printf** routine)
  - Doesn't know where in memory compiled code will go
- Linkage editor: combines objects to **single relocatable, executable image**
  - Arranges objects in program's virtual address space
  - Reorganizes code and data by changing **addresses**
- Loader: loads image from disk into memory for execution
  - Allocates memory space to executable image
  - Transfers control to the beginning instruction of the program
- Dynamic linker: defers linkage of shared libraries until run time
  - Brings shared libraries if it's not already in memory,
  - Binds regions of program's virtual address to shared library





# Recall: Virtual Address Space Layout of C Programs

0xFFFF...F



0x0000...0

```
#include <stdio.h>
#include <stdlib.h>
```

```
int x;
int y = 15;
```

```
int main(int argc, char *argv[]) {
```

```
    int *values;
    int I;
```

```
    values = (int *)malloc(sizeof(int)*5);
```

```
    for (i = 0; i < 5; i++)
        values[i] = i;
```

```
    return 0;
```

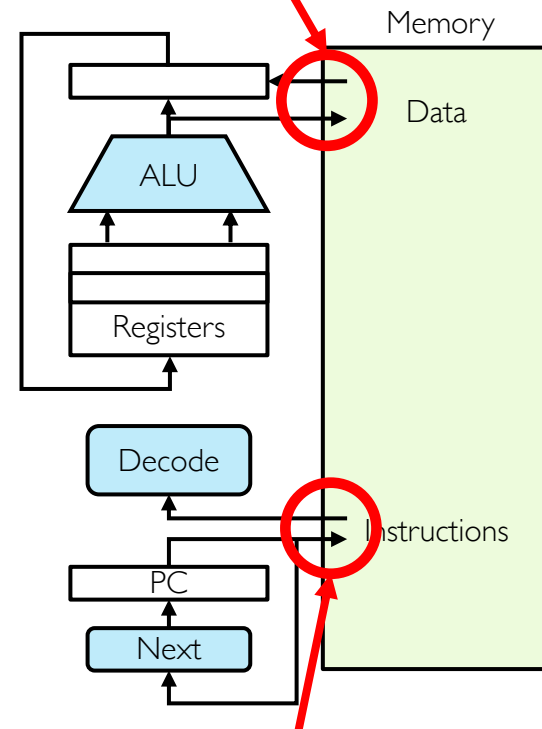
```
}
```

# Recall: What Happens During Program Execution?

- Execution sequence
  - Fetch instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/memory
  - $PC \leftarrow \text{Next}(PC)$
  - Repeat

E.g., function calls, return, branches, etc.

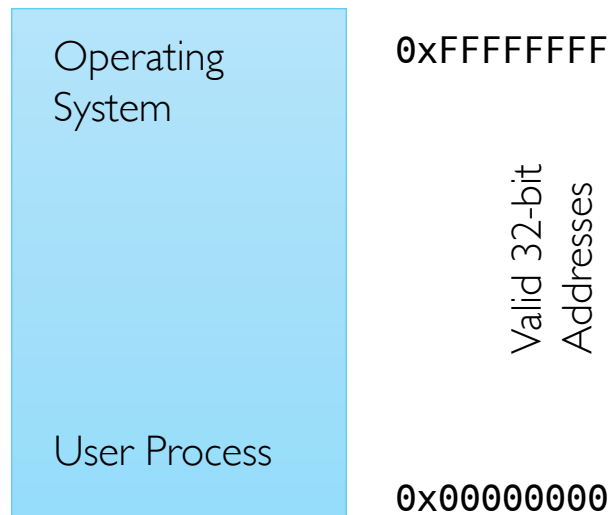
Data references:  
Memory access on load/store instructions



Instruction references:  
Memory access on every instruction

# Uni-programming Without Protection and Translation

- There is always **only one** program running at a time
- Program **always** runs at same place in physical memory
  - Virtual address space = physical address space
- Program can access any physical address

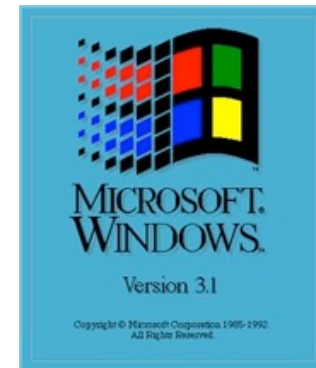
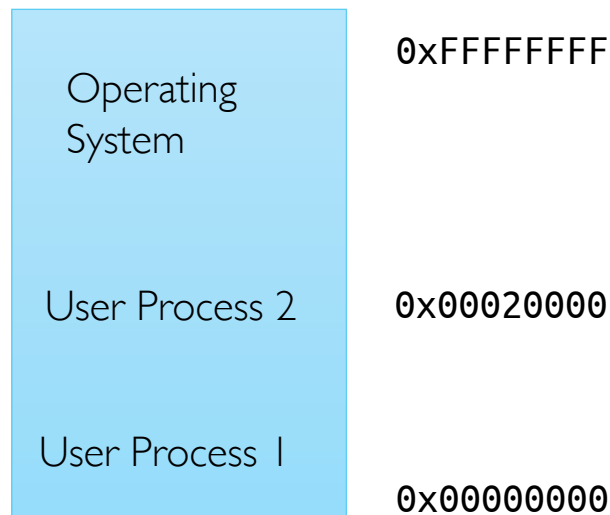


- Program is given illusion of dedicated machine by literally giving it one

# Multi-programming Without Protection and Translation

---

- To prevent address overlap between processes, loader/linker adjust addresses while programs are loaded into memory (loads, stores, jumps)
  - Virtual address = physical address

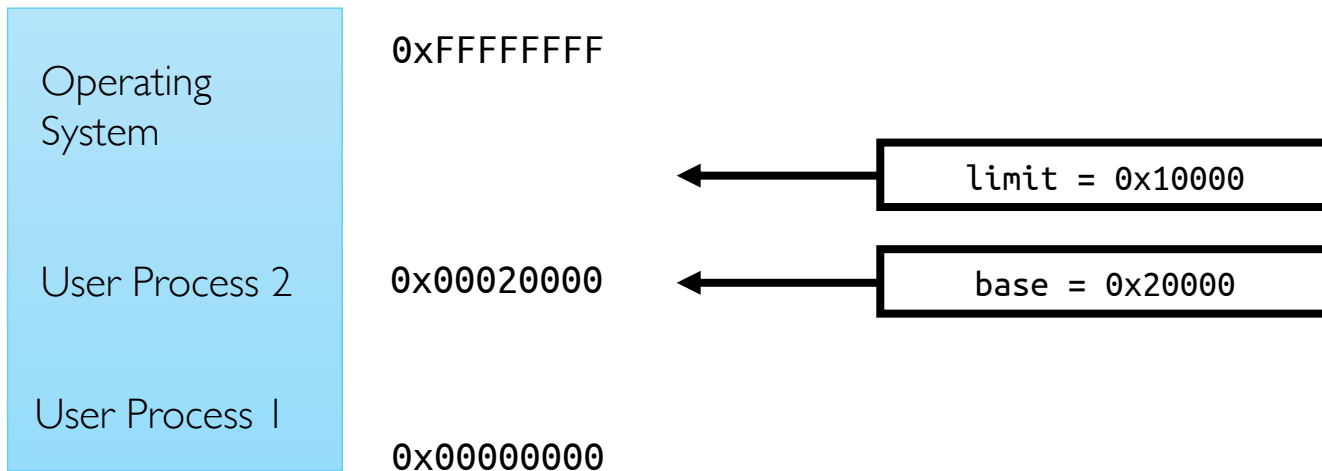


- Bugs in any program can cause other programs (including OS) to crash

# Multiprogramming With Protection but Without Translation

---

- Can we protect programs from each other without translation?
  - Yes: use two special registers **base** and **limit**
    - Prevent application from straying outside designated area
    - If application tries to access an illegal address, raise exception

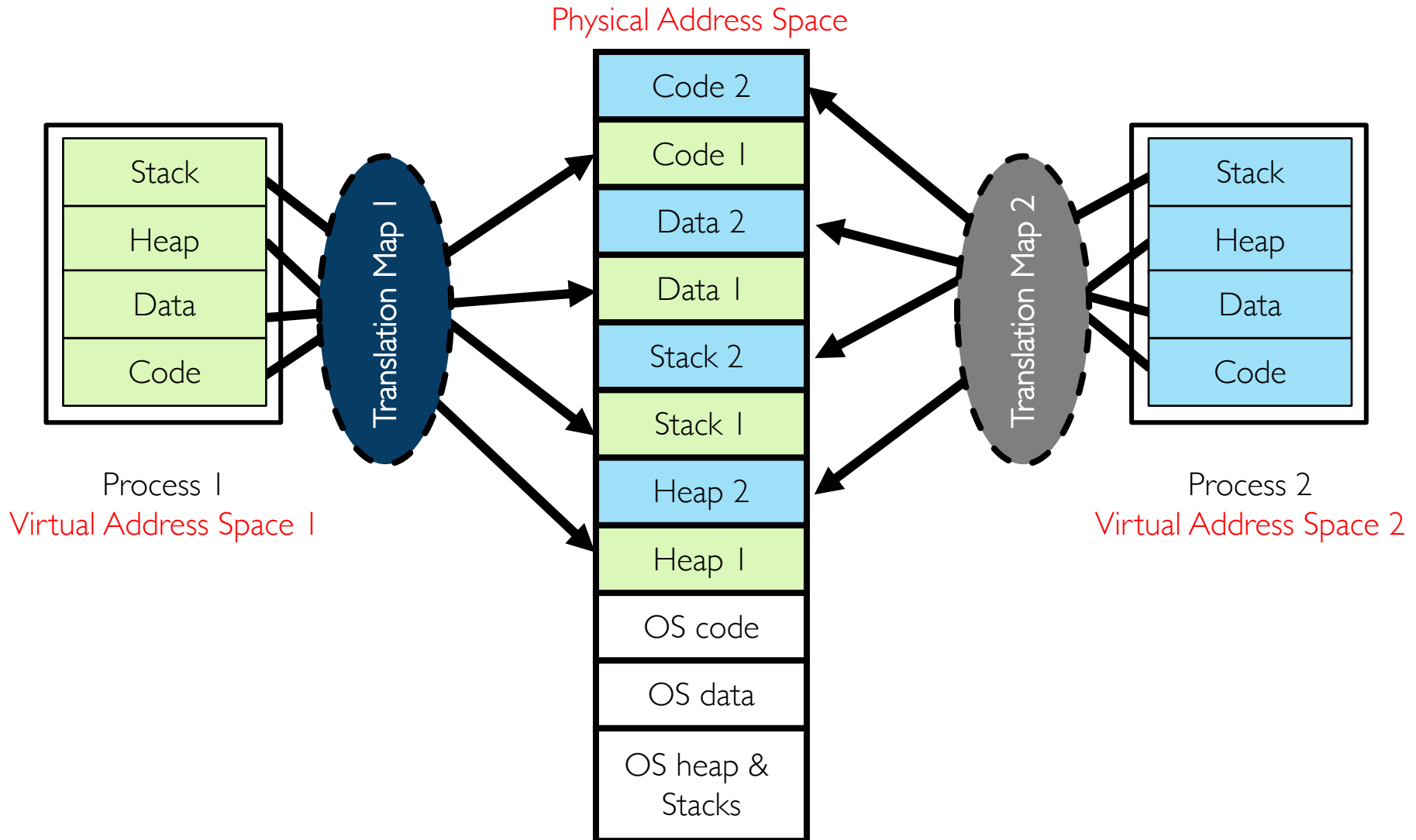


- During switch, kernel loads new base/limit from PCB
  - User is not allowed to change base/limit registers

# Recall:

## Protection With Address Translation

---

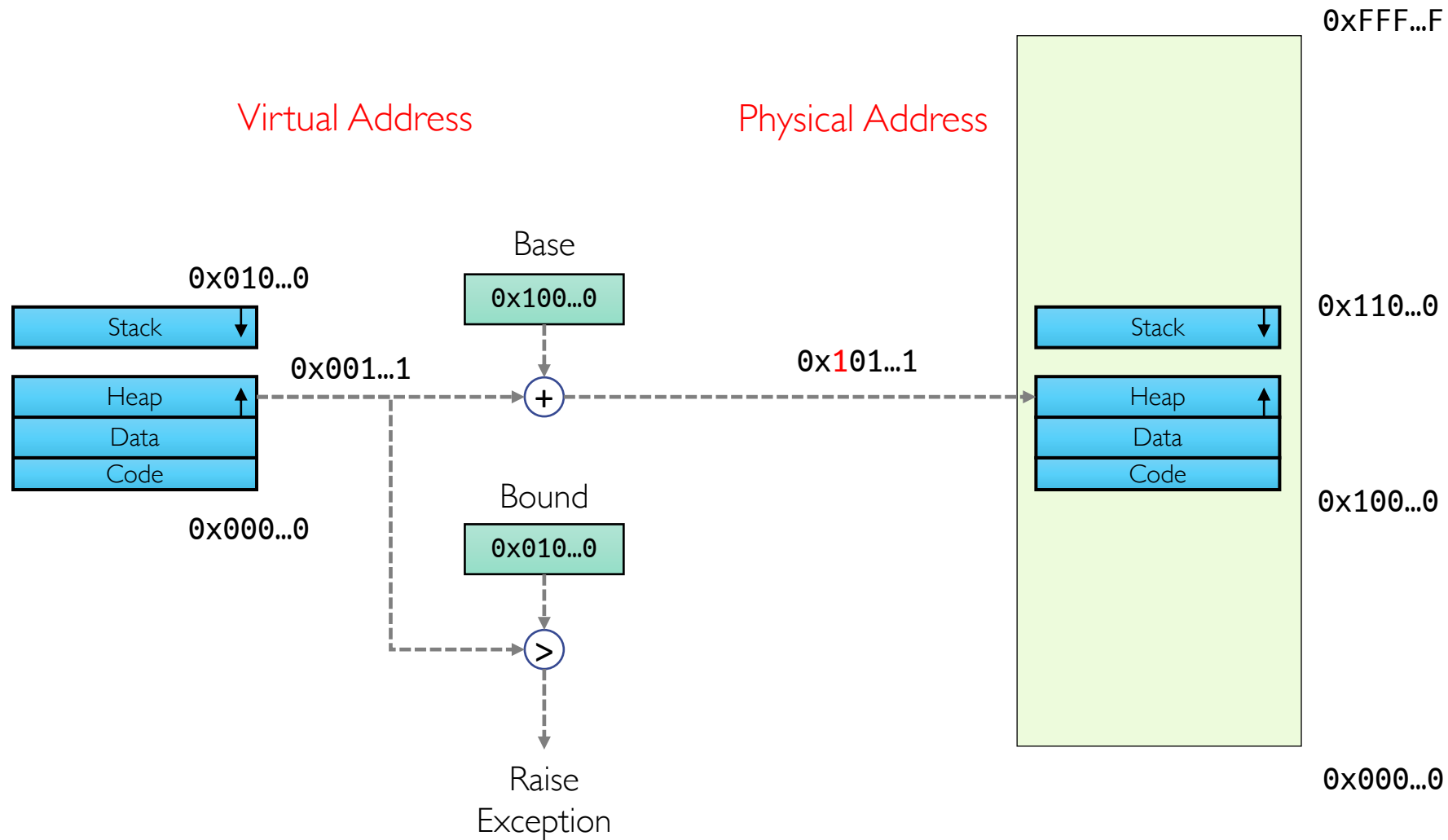


# Protection With Address Translation: Discussion

---

- Upsides
  - Code can be written, compiled, linked, and loaded **independently**
    - Threads think they have unrestricted access to their entire virtual memory range
    - Threads do not need to worry about memory usage of others
  - OS can provide **protection**
    - Threads cannot affect each other if they cannot see each other's memory
  - OS can allow **memory sharing**
    - Threads' virtual memory regions can be mapped to same physical regions
- Downsides
  - Address translation adds **performance overhead**
  - Address translation needs **extra hardware support**
    - Extra hardware consumes area and power

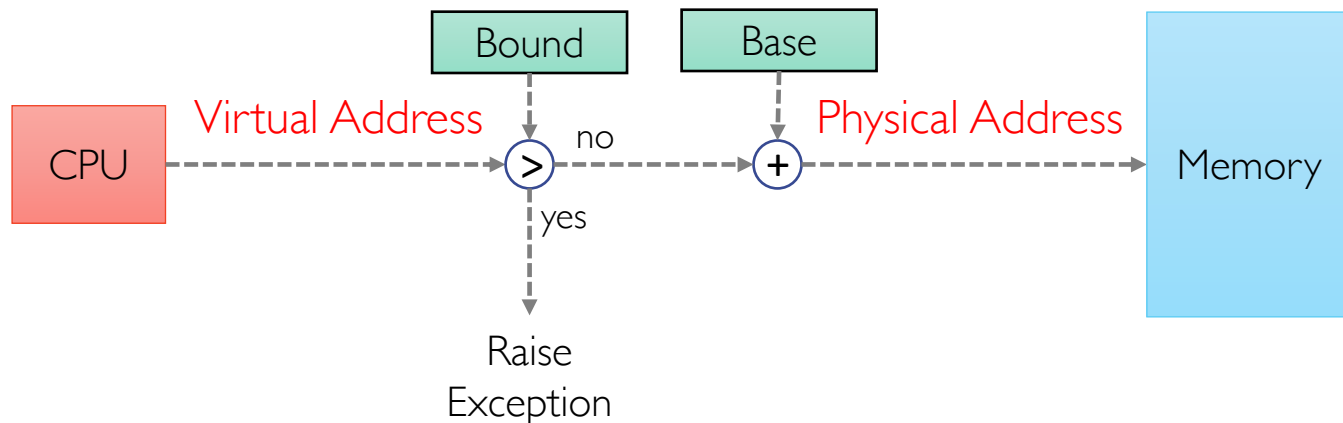
# Base and Bound (B&B) Address Translation





# B&B Address Translation: Discussion

---



- Process is given illusion of running on its own dedicated memory starting at `0x00000000`
- Program are mapped to continuous region of memory
- Virtual addresses do not change if program is relocated to different physical memory region

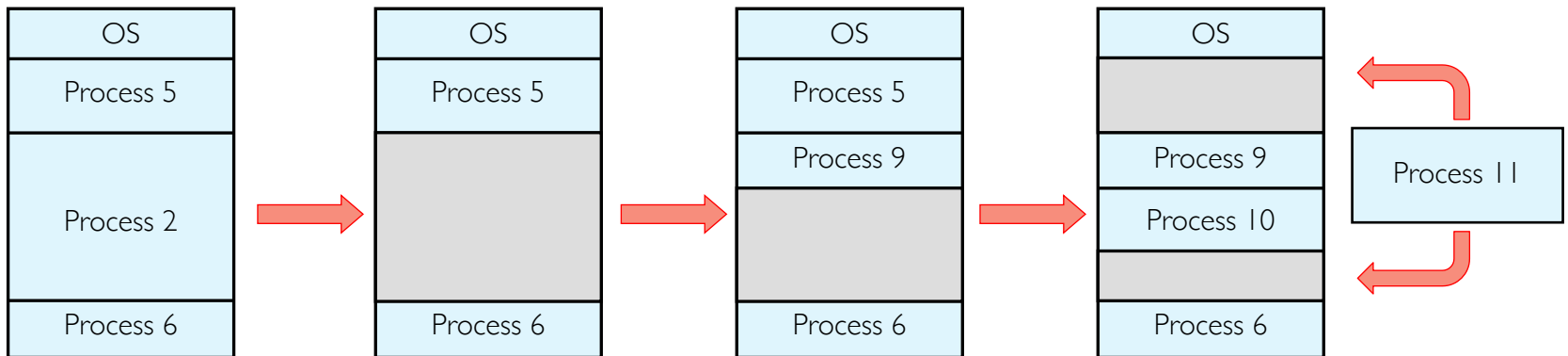
# B&B Address Translation: Discussion (cont.)

---

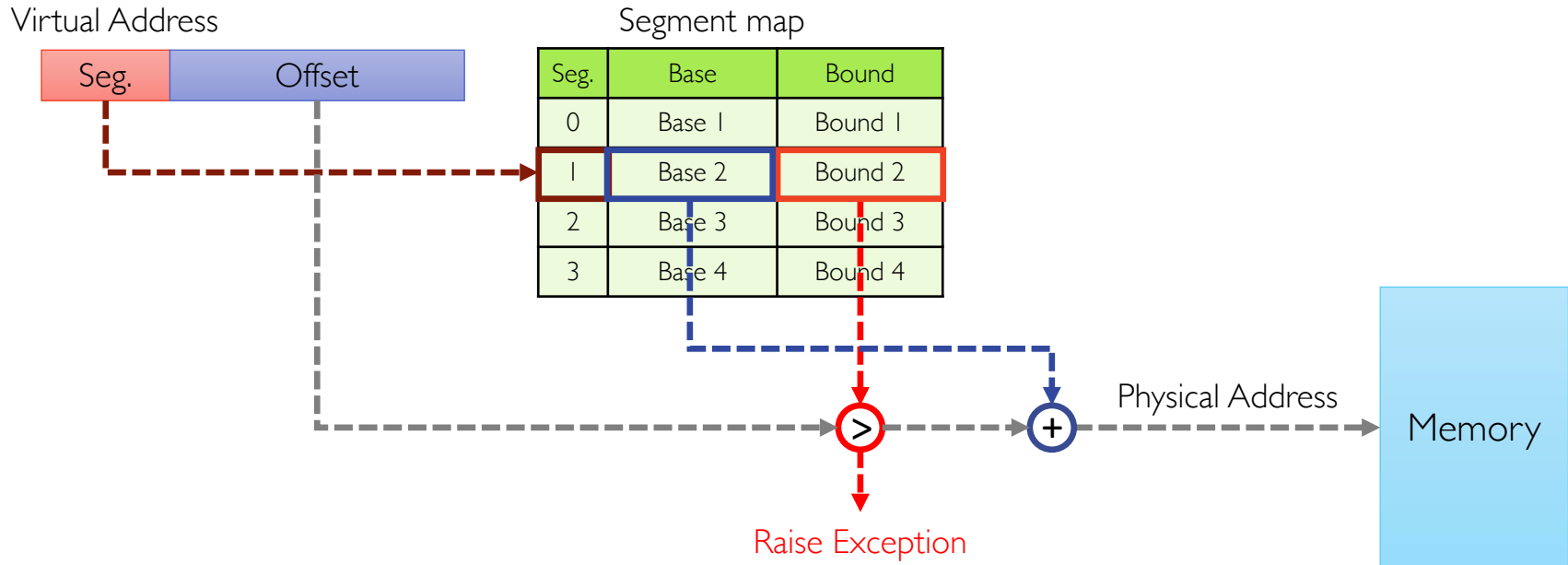
- Upsides
  - OS protection and program isolation
  - Low overhead address translation
- Downsides
  - Expandable heap?
  - Expandable stack?
  - Memory sharing between processes?
  - Non-relative addresses – hard to move memory around
  - Memory fragmentation

# Issues with B&B Address Translation

- Missing support for **inter-process memory sharing**
  - E.g., it's not possible to share code segments in two processes
- **Fragmentation**: wasted space
  - **External**: free gaps between allocated chunks
  - **Internal**: don't need all memory within allocated chunks

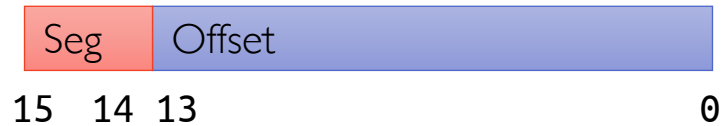


# Multi-segment Address Translation



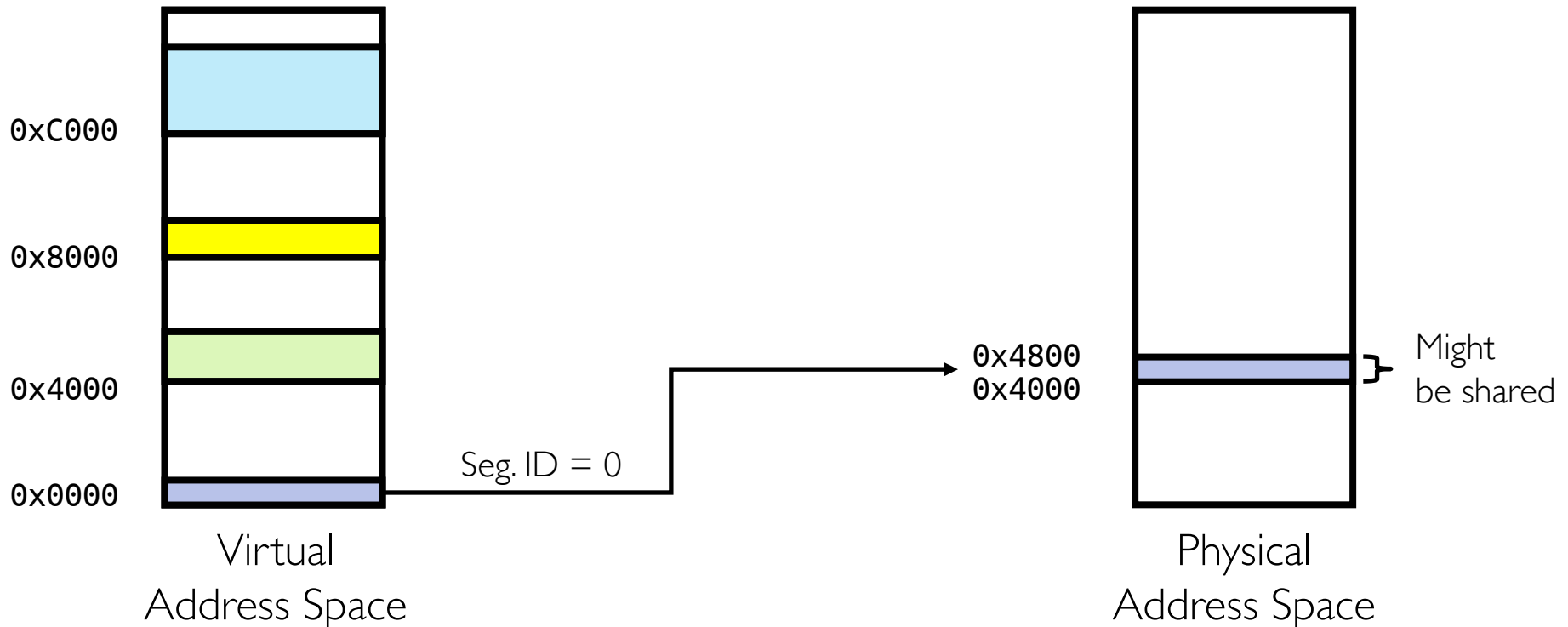
- Segment map resides in processor
  - Base is added to offset to generate physical address
- For each contiguous segment of physical memory there is one entry
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead
    - E.g., `mov ax, es:[bx]`

# Example: Multi-segment Address Translation

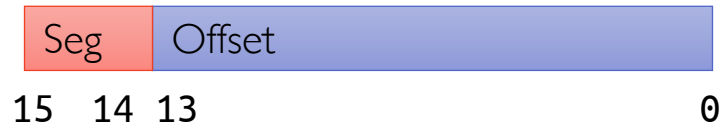


Virtual Address Format

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

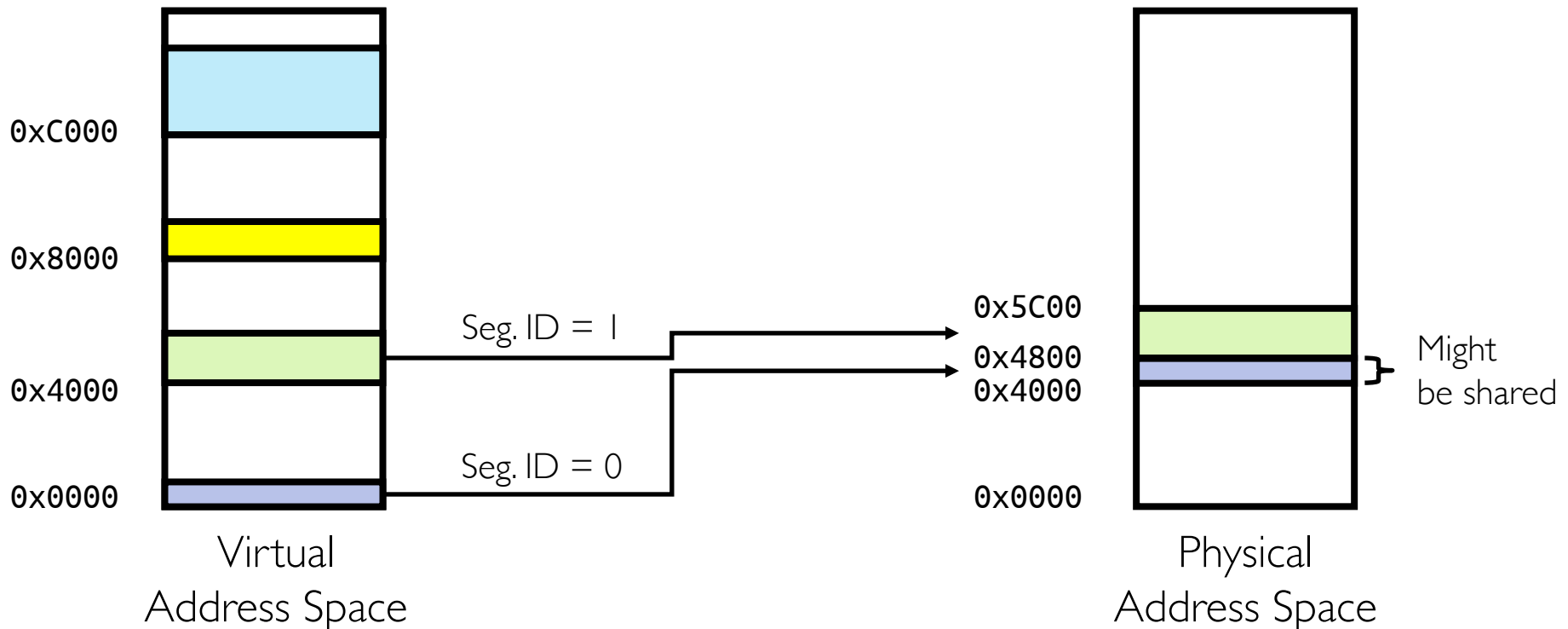


# Example: Multi-segment Address Translation (cont.)

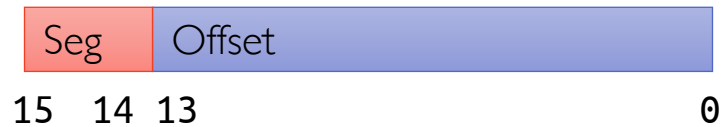


Virtual Address Format

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

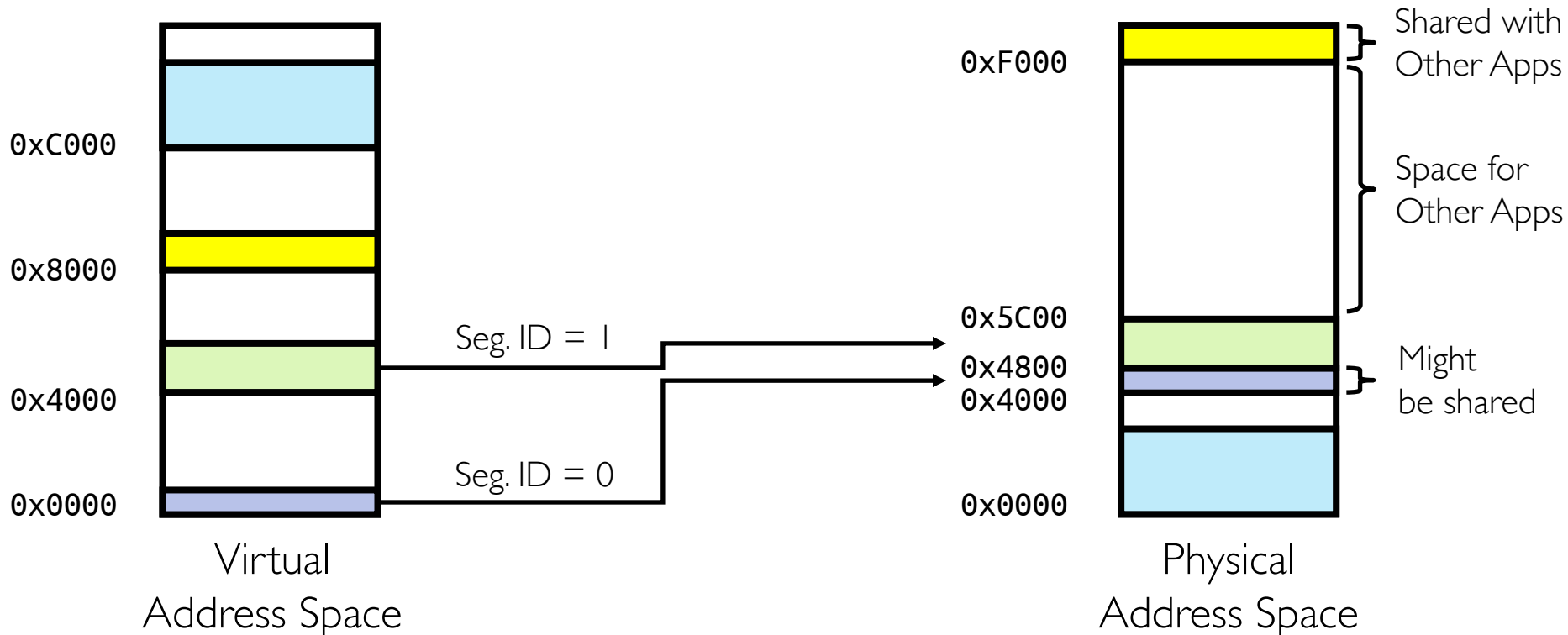


# Example: Multi-segment Address Translation (cont.)



Virtual Address Format

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



# Example: Multi-segment Address Translation (cont.)

0x0240	main:	la	\$a0, varx
0x0244		jal	strlen
...			
0x0360	strlen:	li	\$v0, 0 ;count
0x0364	loop:	lb	\$t0, (\$a0)
0x0368		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

- Fetch 0x0240
  - Virtual segment number? 0, offset? 0x240
  - Physical address? Base: 0x4000, so physical address: 0x4240
  - Fetch instruction at 0x4240, get “la \$a0, varx”
  - Move 0x4050 to \$a0, move PC+4 to PC
- Fetch 0x244, translated to physical address: 0x4244, get “jal strlen”
  - Move 0x0248 to \$ra (return address!), move 0x0360 to PC
- Fetch 0x360, translated to physical address: 0x4360, get “li \$v0, 0”
  - Move 0x0000 to \$v0, move PC+4 to PC
- Fetch 0x0364, translated to physical address 0x4364, get “lb \$t0, (\$a0)”
  - Since \$a0 is 0x4050, try to load byte from 0x4050
  - Translate 0x4050 (0100 0000 0101 0000): virtual segment #? 1, offset? 0x50
  - Physical address? Base: 0x4800, physical address; 0x4850
  - Load byte from 0x4850 to \$t0, move PC+4 to PC



# Multi-segment Address Translation: Discussion

---

- Virtual address space has holes
  - It's efficient for sparse address spaces (avoids internal fragmentation)
  - If program tries to access gaps, trap to kernel (*segmentation fault*)
- When is it OK to address outside valid range?
  - This is how stack and heap grow
  - E.g., stack takes segmentation fault, kernel automatically increases size of stack
- What must be saved/restored on context switch?
  - Segment table stored in CPU, not in memory (small)
  - Might store all of processes memory in disk when switched (called *swapping*)
- What are downsides?
  - Must fit variable-sized chunks into physical memory (external fragmentation)
  - Limited options for swapping to disk

# Paged Memory

---

- Allocate physical memory in fixed-size chunks called **pages**

- Can use simple **bit map** to handle allocation

00110001110001101 ... 110010

- Each bit represents page of physical memory

1  $\Rightarrow$  **allocated**, 0  $\Rightarrow$  **free**

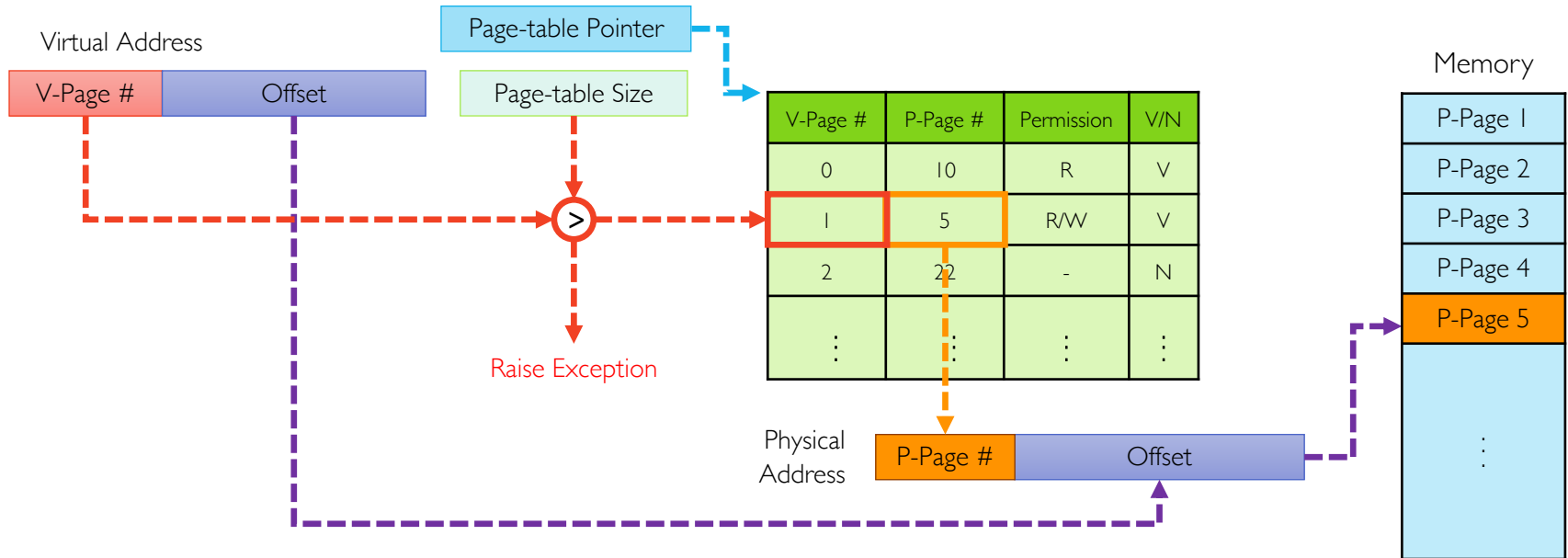
- Should pages be as big as our previous segments?

- No, big pages could lead to internal fragmentation

- Typically, pages are small (**1-16Kib**)

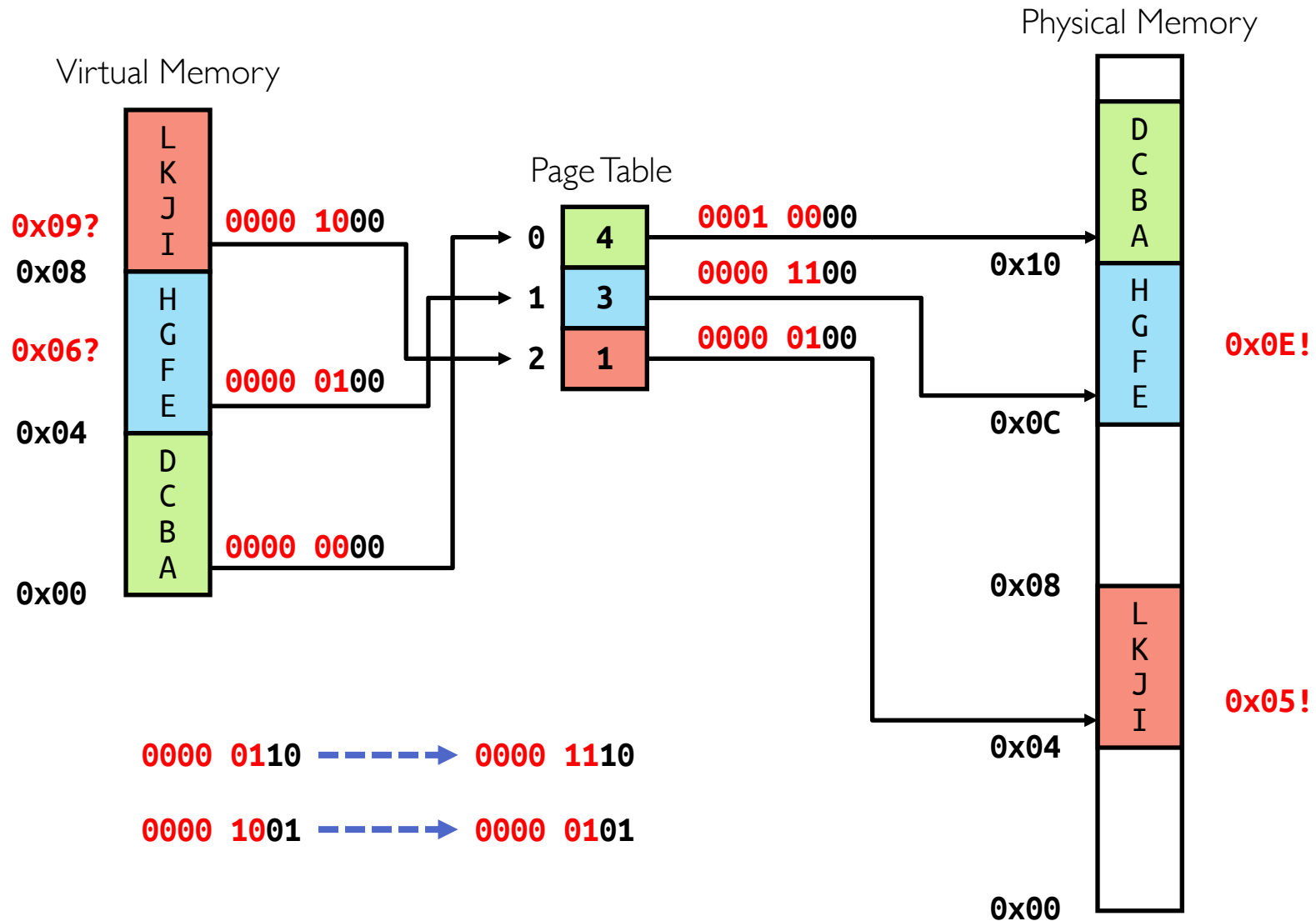
- Consequently, each segment needs multiple pages

# Page-table Address Translation



- Page resides in physical memory
- Contains physical page and permission for each virtual page
- Offset from virtual address gets copied to physical address
  - E.g., **10-bit** offset  $\Rightarrow$  **1024-byte** = **4KiB** pages
- Virtual page number is all remaining bits
- Physical page number is copied from table into physical address

# Example: Page-table Address Translation with 4-byte Pages



# Page-table Entry

---

- What is in each page-table entry (or PTE)?
  - Pointer to actual page
  - Permission bits: valid, read-only, read-write, write-only

Read	Write	Execute	Use Case
X	X	X	Code or data; was common, but now generally deprecated/discouraged due to security risks
X	X	-	Read-write data; very common
X	-	X	Executable code; very common
X	-	-	Read-only data; very common
-	X	X	N/A
-	X	-	Interaction with devices
-	-	X	To protect code from inspection; uncommon
-	-	-	Guard; security feature used to trap buffer overflows or other illegal accesses

# Permissions in Action

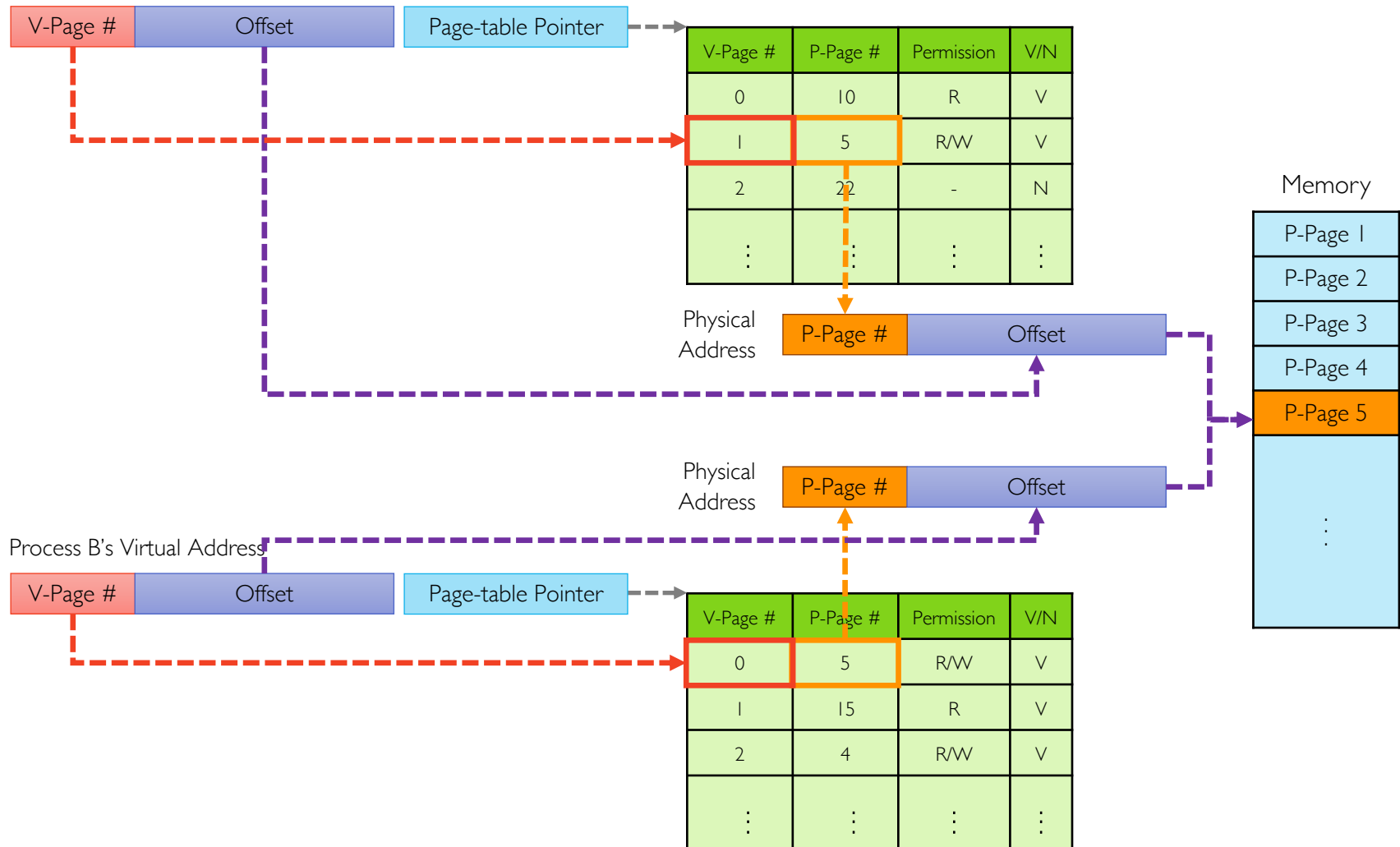
---

- **Demand paging** (more on this later)
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid
- **Copy-on-write**
  - UNIX fork gives copy of parent address space to child
  - How to do this cheaply?
    - Make copy of parent's page tables
    - Mark entries in both sets of page tables as read-only
    - On write, page fault happens, OS creates two copies
- **Zero-fill-on-demand**
  - New data pages must carry no information (say be zeroed)
  - Mark PTEs as invalid; page fault on use gets zeroed page
  - Often, OS creates zeroed pages in background

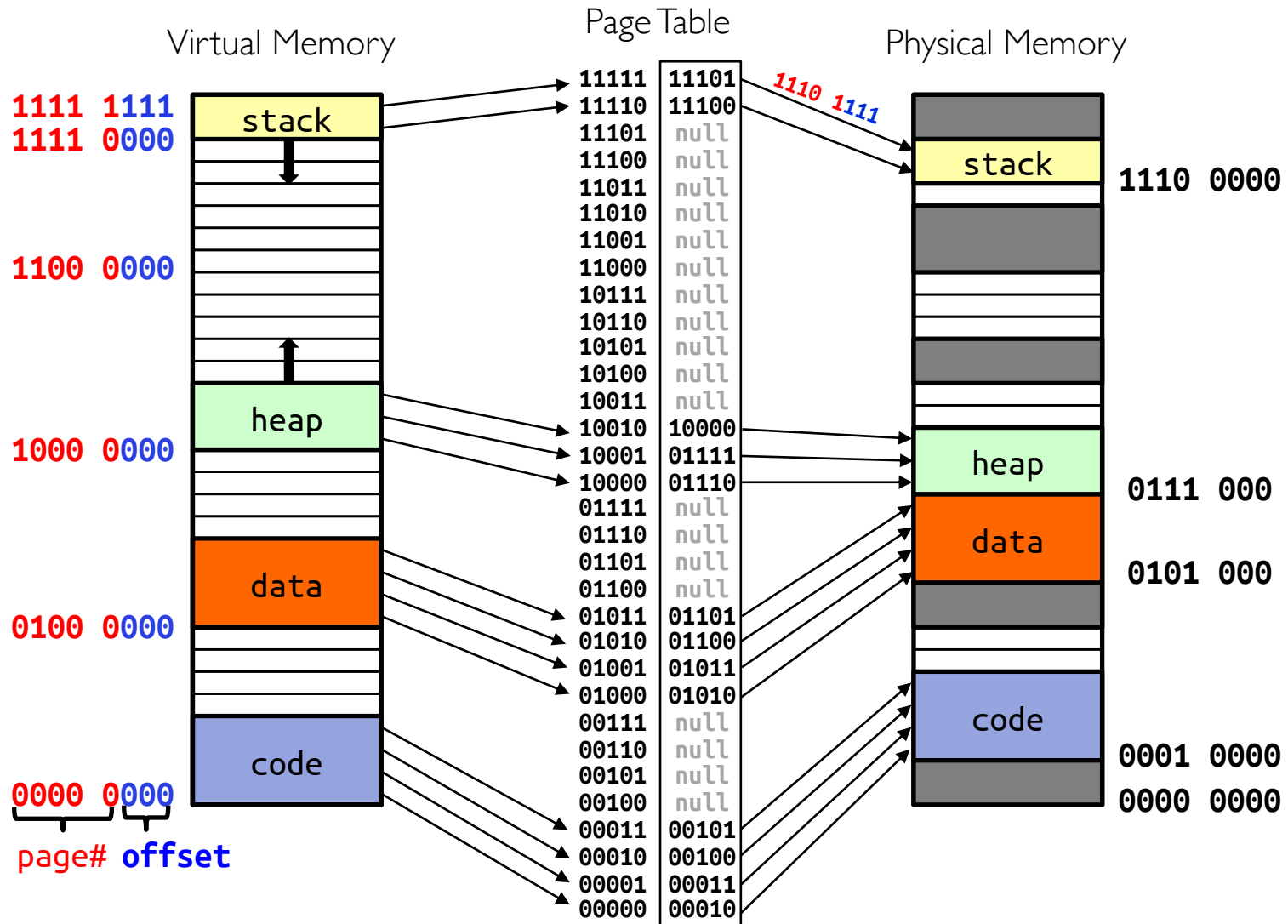


# Memory Sharing

Process A's Virtual Address

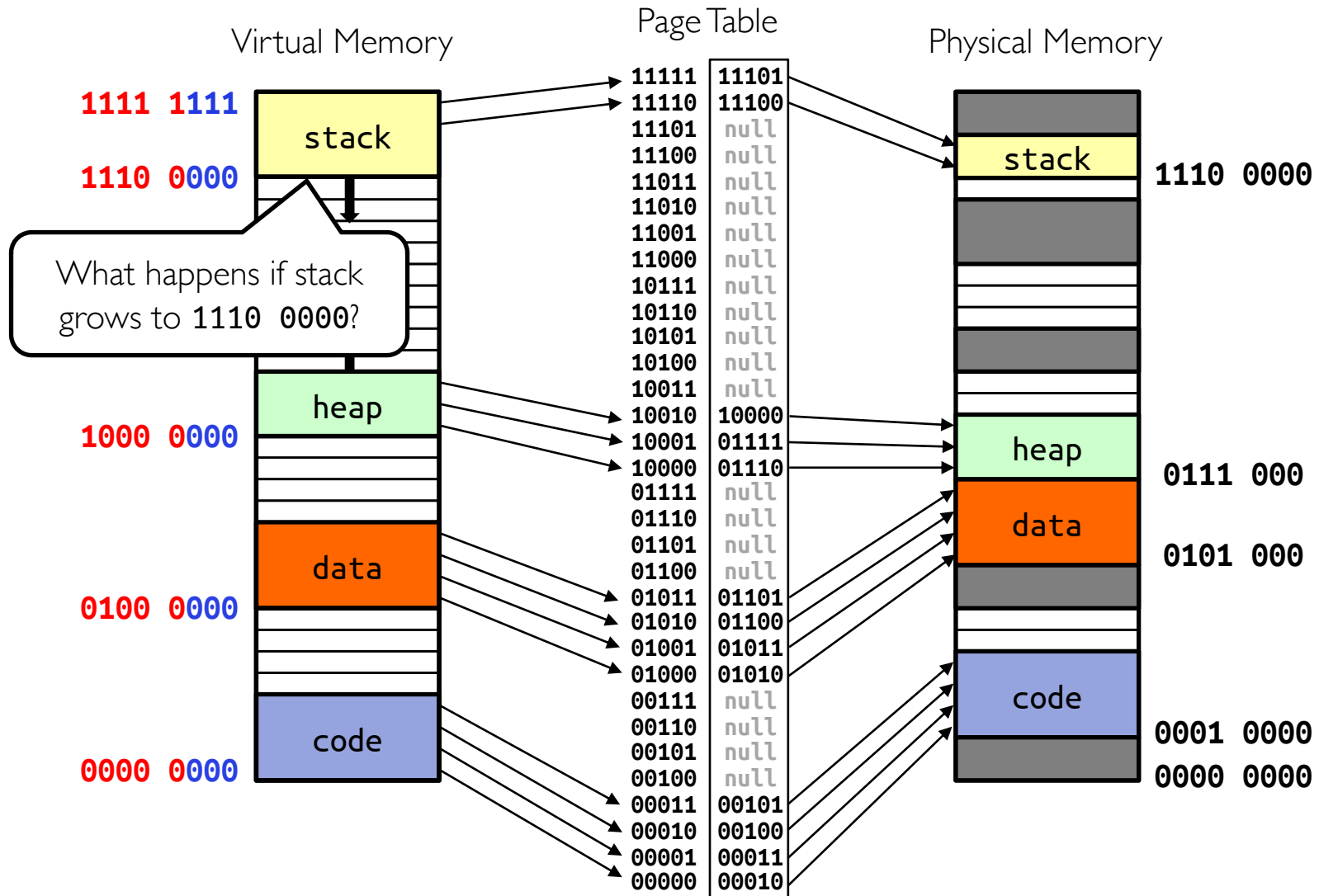


# Example: Updating Page Table

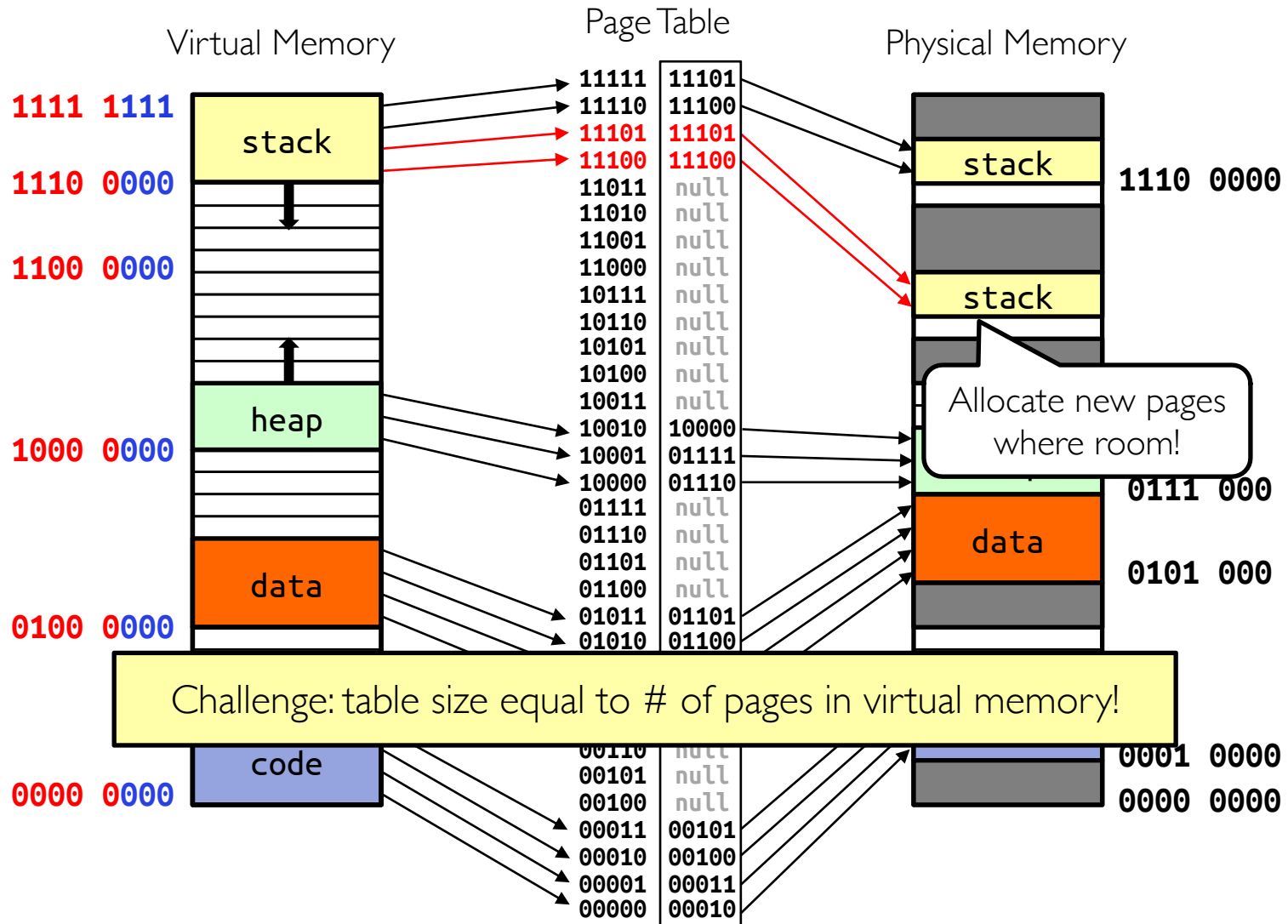




# Example: Updating Page Table (cont.)



# Example: Updating Page Table (cont.)

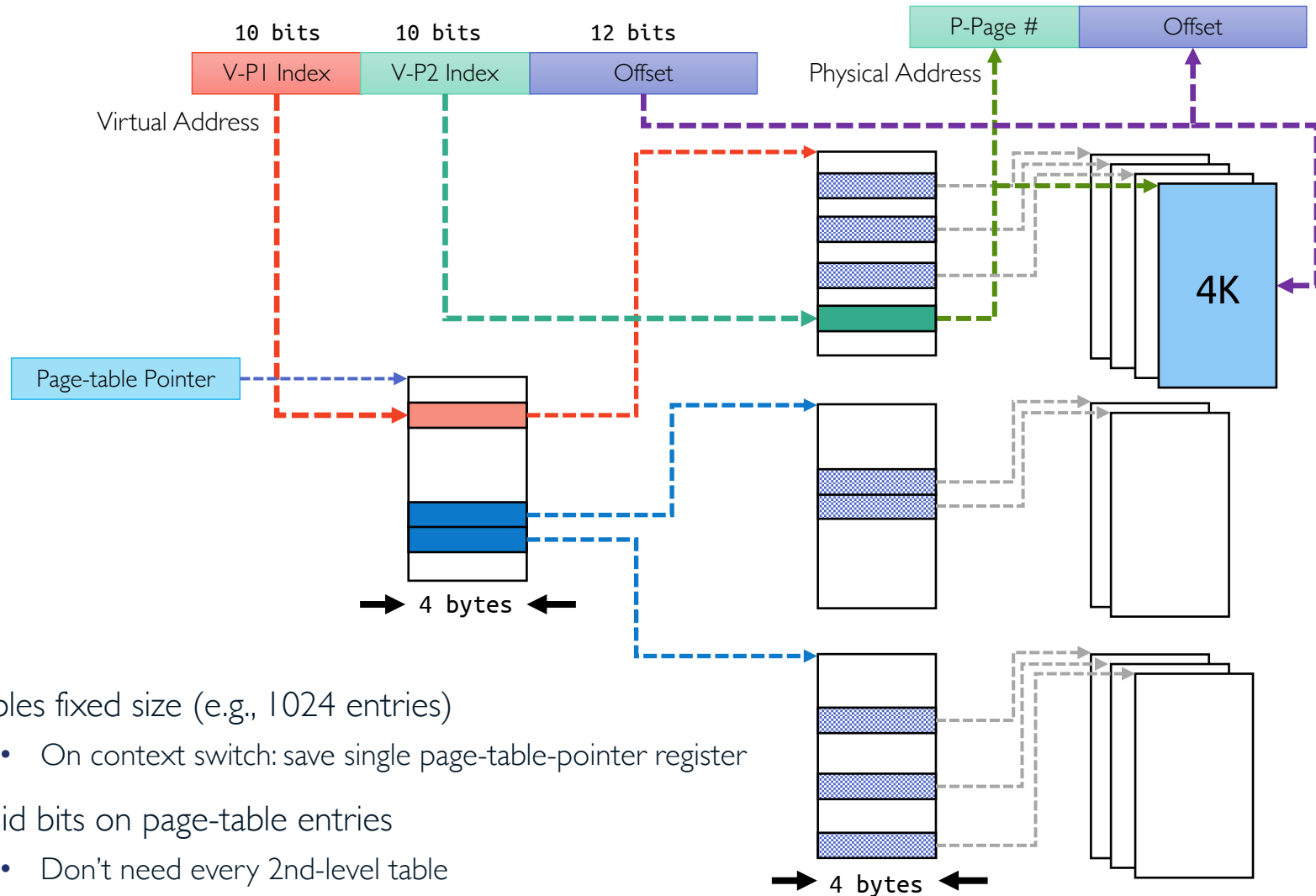


# Page-table Address Translation: Discussion

---

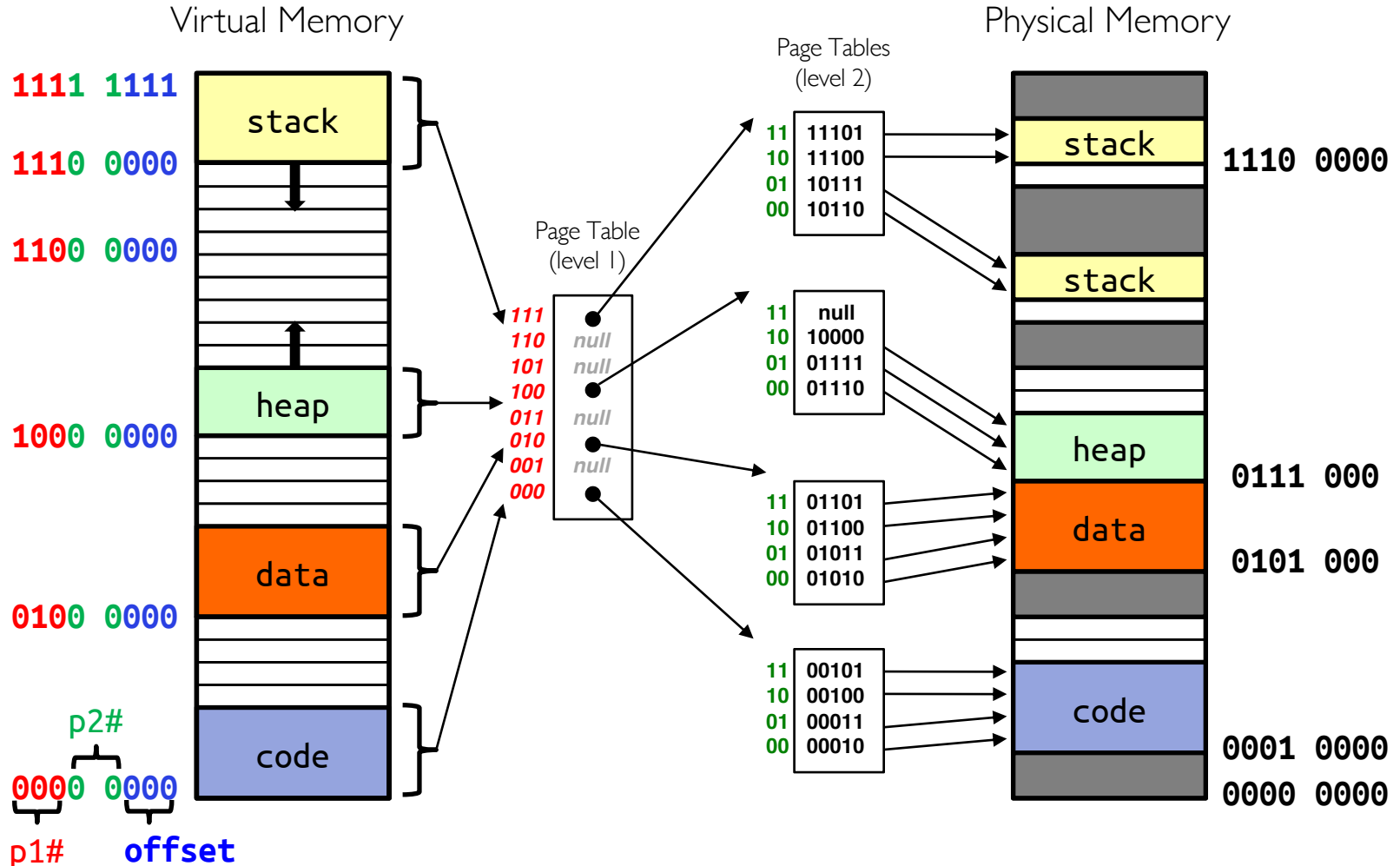
- What needs to be switched on context switch?
  - Page-table pointer and page-table size
- How big is page table?
  - **32-bits** and **4KiB** pages  $\Rightarrow 2^{20}$  entries  $\times$  **4B** each  $\Rightarrow$  **4MiB**
  - **64-bits** and **4KiB** pages  $\Rightarrow 2^{52}$  entries  $\times$  **8B** each  $\Rightarrow$  **32PiB**
- Upsides
  - + Simple memory allocation
  - + Easy to share
- Downsides
  - – Inefficient for sparse address spaces
  - There are too many unused page-table entries
  - What if page size is very small?
    - With **1KiB** pages, we need  $2^{22}$  (~4 million) table entries!
  - What if page size is too big?
    - Wastes space inside of page (internal fragmentation)

# Two-level Page-table Address Translation

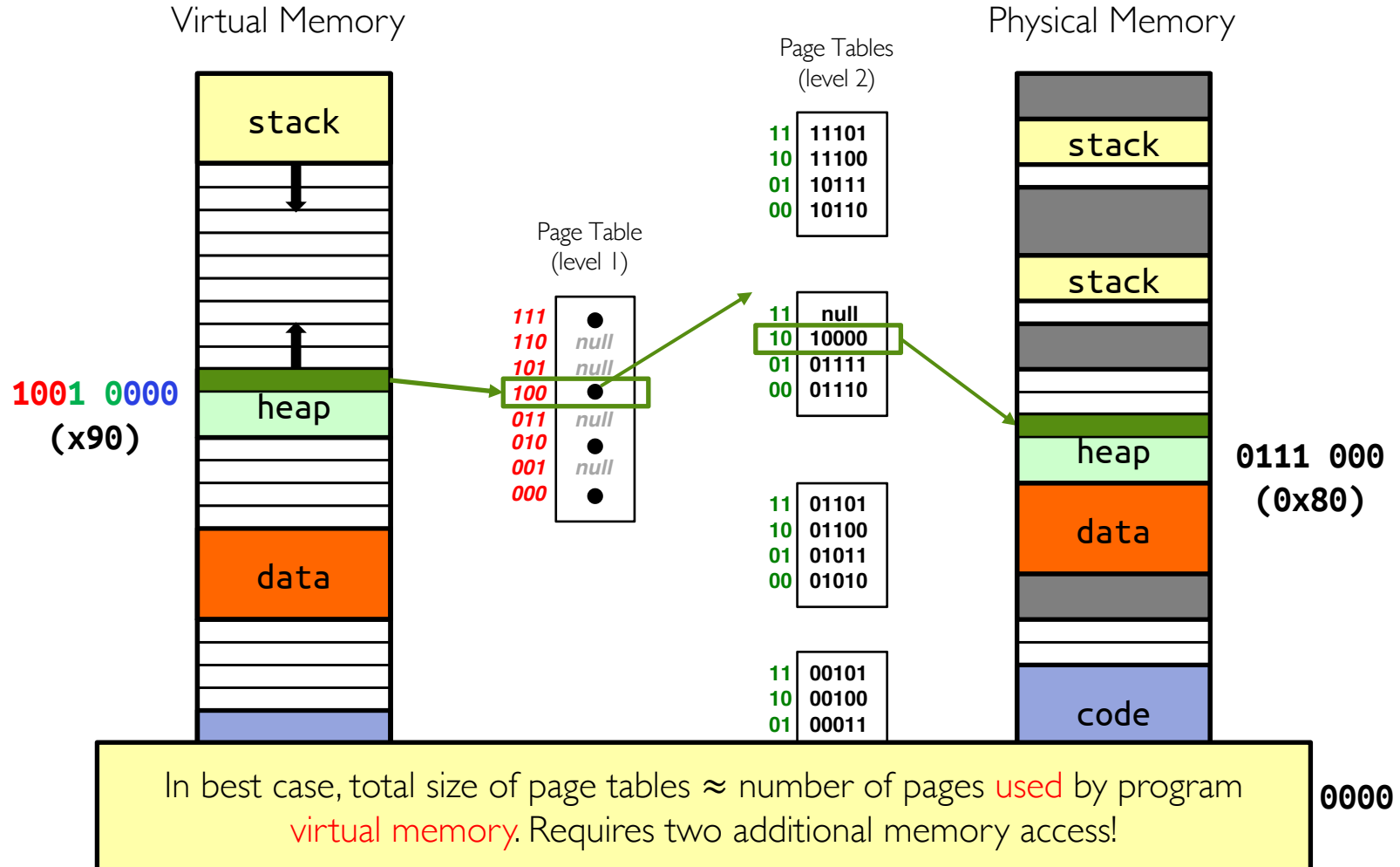


- Tables fixed size (e.g., 1024 entries)
  - On context switch: save single page-table-pointer register
- Valid bits on page-table entries
  - Don't need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

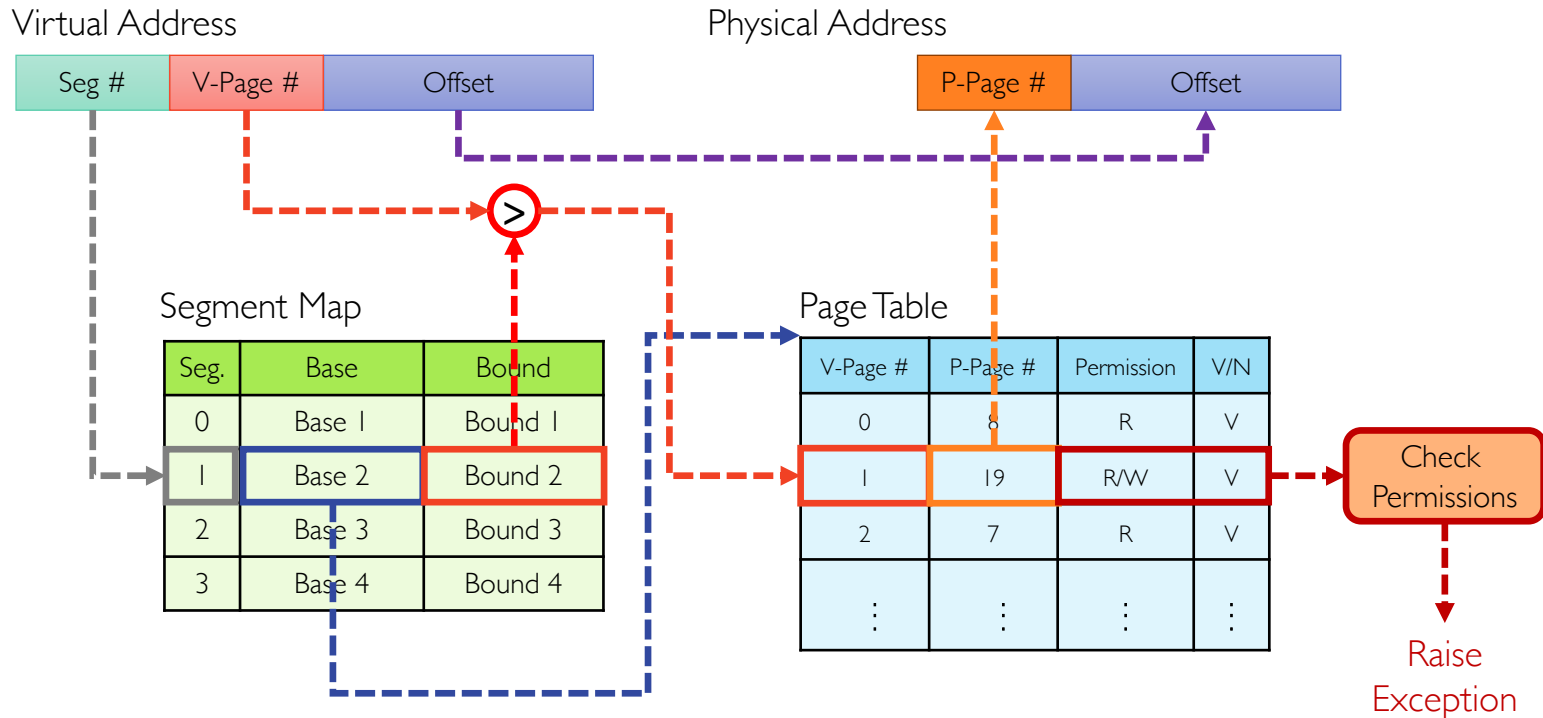
# Example: Two-level Page-table Address Translation



# Example: Two-level Page-table Address Translation (cont.)



# Multi-level Address Translation: Segments and Pages



- What must be saved/restored on context switch?
  - Contents of top-level segment registers

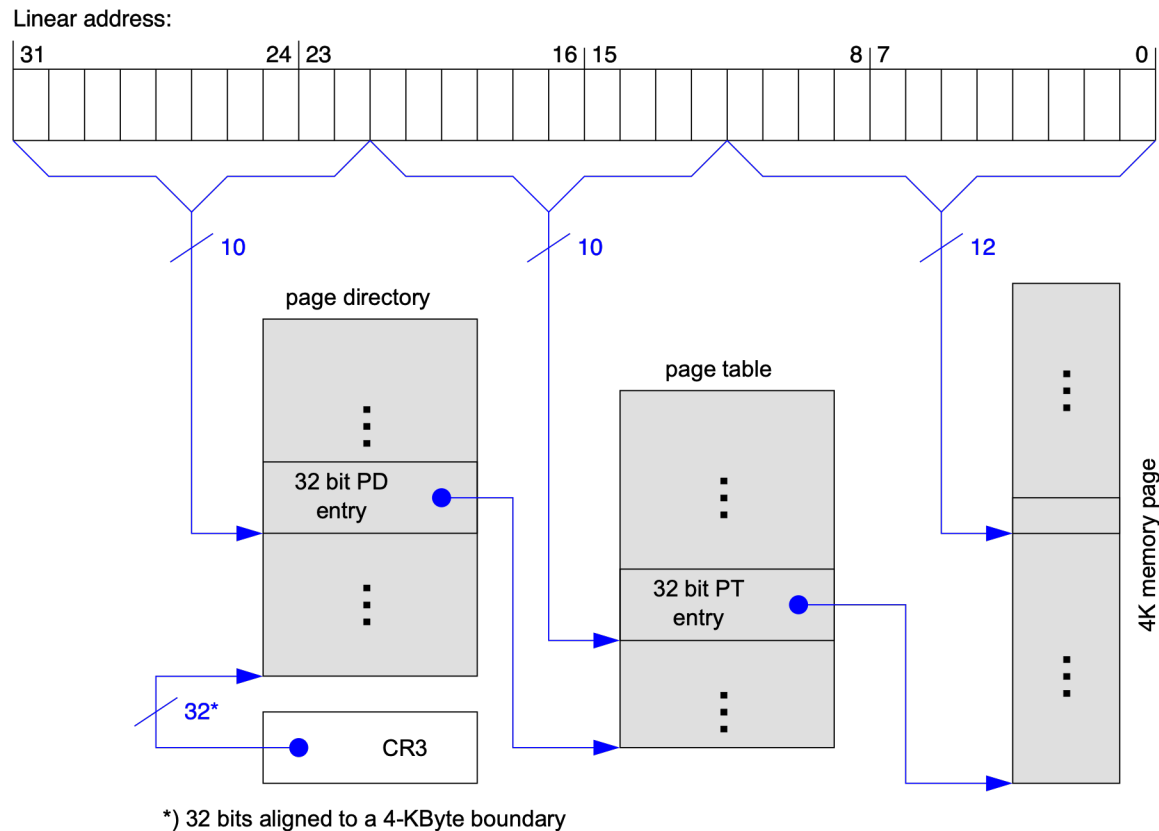
# Example: Multi-level Paged Segmentation (x86)

---

- Global descriptor table (segment table)
  - Pointer to page table for each segment
  - Segment length
  - Segment access permissions
- What should be saved on context switch?
  - Change global descriptor table register (GDTR, pointer to global descriptor table)
- Multi-level page table
  - 32-bit: two-level page table (per segment)
  - 64-bit: four-level page table (per segment)

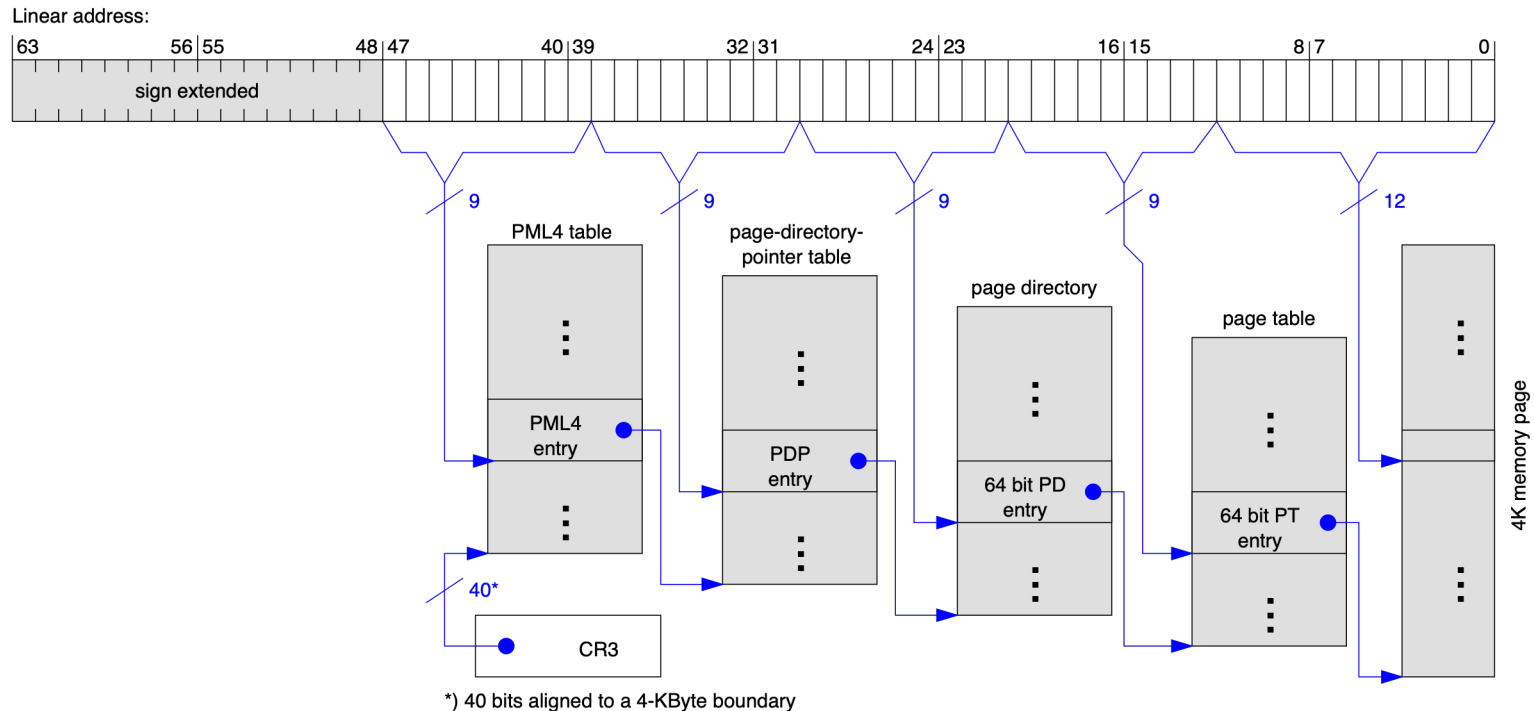


# x86 32-bit Virtual Address



- 4KiB pages; each level of page table fits in one page

# x86 64-bit Virtual Address



- Fourth-level table maps 2MiB, and third level table maps 1 GiB of data
- If physical memory covered by fourth level table is contiguous, then one third-level entry can directly point to this region instead of pointing to fourth-level page table

# Example: x86 64-bit PTE

---

$\overline{NX}$	SW	Reserved	P-Page Number	U	P	$\overline{C}$	CD	L	D	A	G	WT	O	W	V
63	62-52	51-40	39-12	11	10	9	8	7	6	5	4	3	2	1	0

- V: Valid
- W: Read/write
- O: Owner (user/kernel)
- WT: Write-through (more on this soon)
- CD: Cache-disabled (page cannot be cached)
- A: Accessed: page has been accessed recently
- D: Dirty bit (page has been modified recently)
- L: Large page
- G: Global
- CP: Copy-on-write
- P: Prototype PTE
- U: Reserved
- SW: Software (working set index)
- NX: No-execute

# Multi-level Address Translation: Sharing Entire Segment

Process A's Virtual Address

Seg #	Page #	Offset
-------	--------	--------

Segment Map

Seg.	Base	Bound
0	Base 1	Bound 1
1	Base 2	Bound 2
2	Base 3	Bound 3
3	Base 4	Bound 4

Segment Map

Seg.	Base	Bound
0	Base 1	Bound 1
1	Base 2	Bound 2
2	Base 3	Bound 3
3	Base 4	Bound 4

Seg #	Page #	Offset
-------	--------	--------

Process B's Virtual Address

V-Page #	P-Page #	Permission	V/N
0	8	R	V
1	19	R/W	V
2	7	R	V
⋮	⋮	⋮	⋮

# Aside: Shared Library Address Space

---

- Shared library's global and static variables are private to each process
  - Each process has **read and write** permissions on its own copy of variables
- Shared library's code is shared between different processes
  - Each process only has **read and execute** permissions on shared code
- Shared library code must be *position-independent code (PIC)*
  - Same library code could be mapped to different virtual address regions in different processes
  - Code must execute properly regardless of its absolute virtual address
  - **Code cannot contain absolute virtual addresses for data and instruction references**
- Data references are made indirectly through *global offset tables (GOT)*
  - GOT is located at fixed offset from code and can be accessed using PC-relative offset
  - GOT has one entry per variable which contains absolute address of that variable
  - **GOT is private to each process, and processes have read and write permissions to their GOT**
- Similarly, instruction references are made indirectly through *procedure linkage table (PLT)*

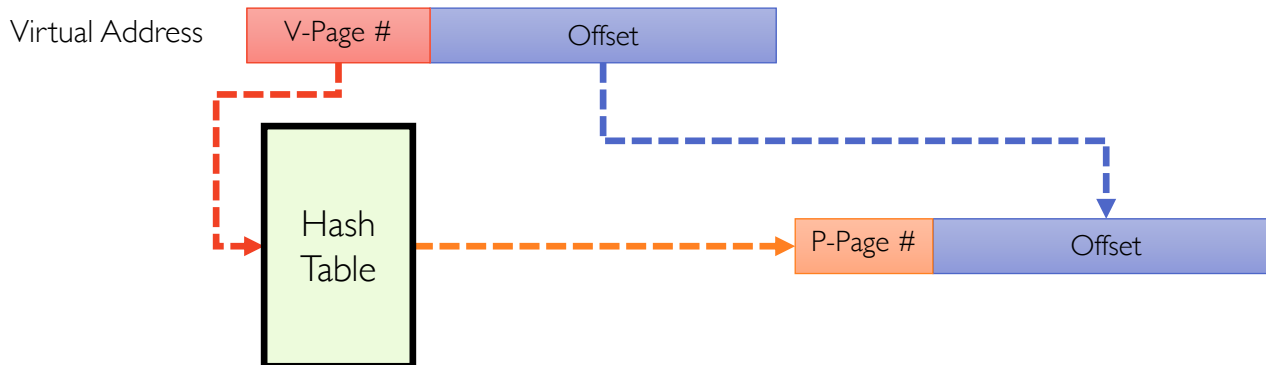
# Multi-level Address Translation: Discussion

---

- + Allocate only as many page-table entries as needed for application
  - In other words, sparse address spaces are easy
- + Easy memory allocation
  - Bit-map memory allocation
- + Easy sharing
  - Share at segment or page level (need additional reference counting)
- – One extra pointer per page
  - One pointer per 4 - 16KiB pages
- – Page tables need to be contiguous
  - However, we can make each table to fit exactly into one page
- – Two (or more, if  $> 2$  levels) lookups per reference
  - Seems very expensive!

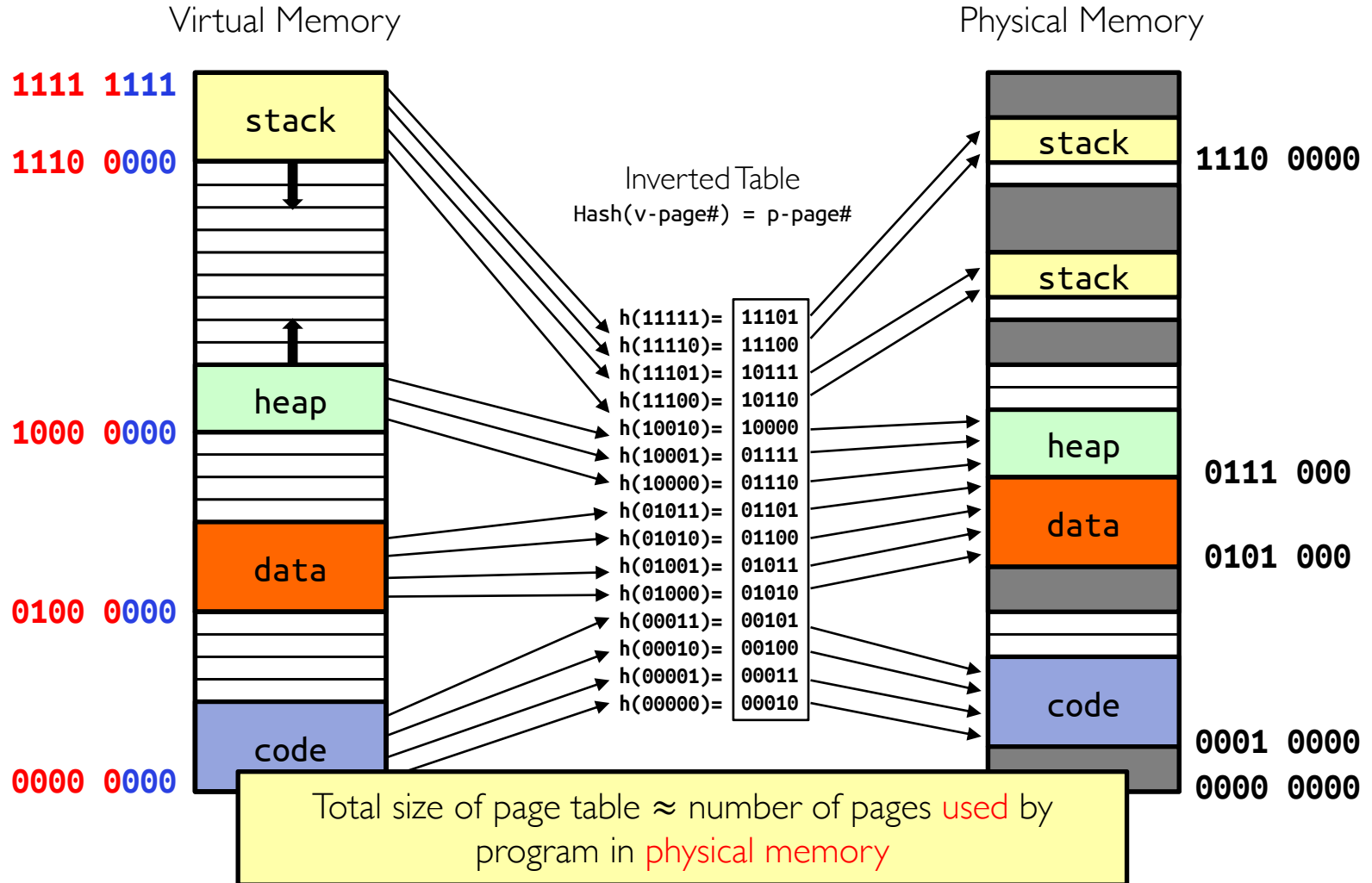
# Inverted Page Table

- In all previous methods (**forward page tables**), size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much smaller
- **Inverted page table** fixes this problem by using hash table
  - Size of hash table is related to size of physical memory not virtual address space
  - Very attractive option for 64-bit address spaces (e.g., PowerPC, UltraSPARC, IA64)



- Notice any downsides?
  - Complexity of managing hash chains: often in hardware!
  - Poor cache locality of page table

# Inverted Paging Example (cont.)





# HW vs. SW Address Translation

---

- Does kernel require HW support for translation?
  - No! Almost anything that can be done in HW can also be done in SW (might end up being too expensive, but possible!)
- Implement page tables in HW
  - All memory reference pass through **memory management unit (MMU)**
  - MMU generates **page fault** if it encounters invalid PTE
  - Fault handler will decide what to do (more on this later)
  - + Relatively fast (but still many memory accesses!)
  - – Inflexible, complex hardware
- Implement page tables in SW
  - + Very flexible
  - – Every translation must invoke fault!
- In fact, we need a way to cache translations for either case

# Address Translation Comparison

---

Method	Advantages	Disadvantages
Segmentation	Fast context switching: Segment mapping maintained by CPU	External fragmentation
Page table translation	No external fragmentation, fast easy allocation	Large table size $\sim$ virtual memory
Multi-level translation	Table size $\sim$ # of pages in virtual memory, fast easy allocation	Multiple memory references per page access
Inverted table	Table size $\sim$ # of pages in physical memory	Hash function more complex

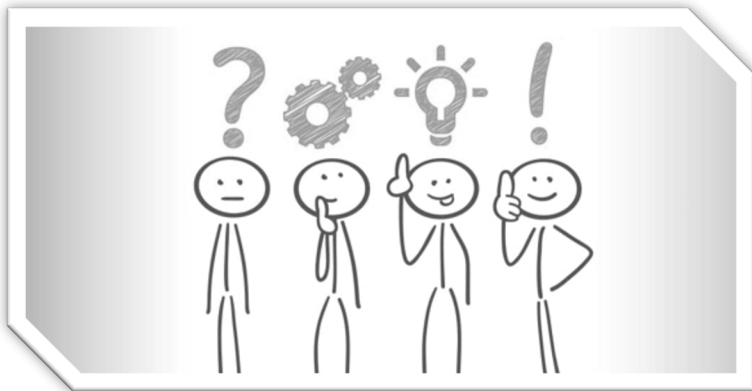
# Summary

---

- Segmentation
  - Segment ID associated with each access
  - Each segment contains base and limit information
- Page tables
  - Memory divided into fixed-sized chunks of memory
  - Virtual page # from virtual address mapped through page table to physical page #
- Multi-level tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
- Inverted page table
  - Use of hash-table to hold translation entries
  - Size of page table  $\sim$  size of physical memory rather than size of virtual memory

# Questions?

---



# Acknowledgment

---

- Slides by courtesy of Anderson, Ousterhout, Culler, Stoica, Silberschatz, Joseph, and Canny