

Lecture 09: Virtual Memory

CS343 – Operating Systems
Branden Ghena – Fall 2025

Some slides borrowed from:

Stephen Tarzia (Northwestern), Shivaram Venkataraman (Wisconsin), and UC Berkeley CS61C and CS162

Administrivia

- Grades are posted
 - PC Lab
 - Midterm Exam
- Drop deadline is this week on Friday
 - I'm happy to meet with you if you're concerned
- Scheduling Lab due Thursday!
 - Warning: way more C code than PC Lab

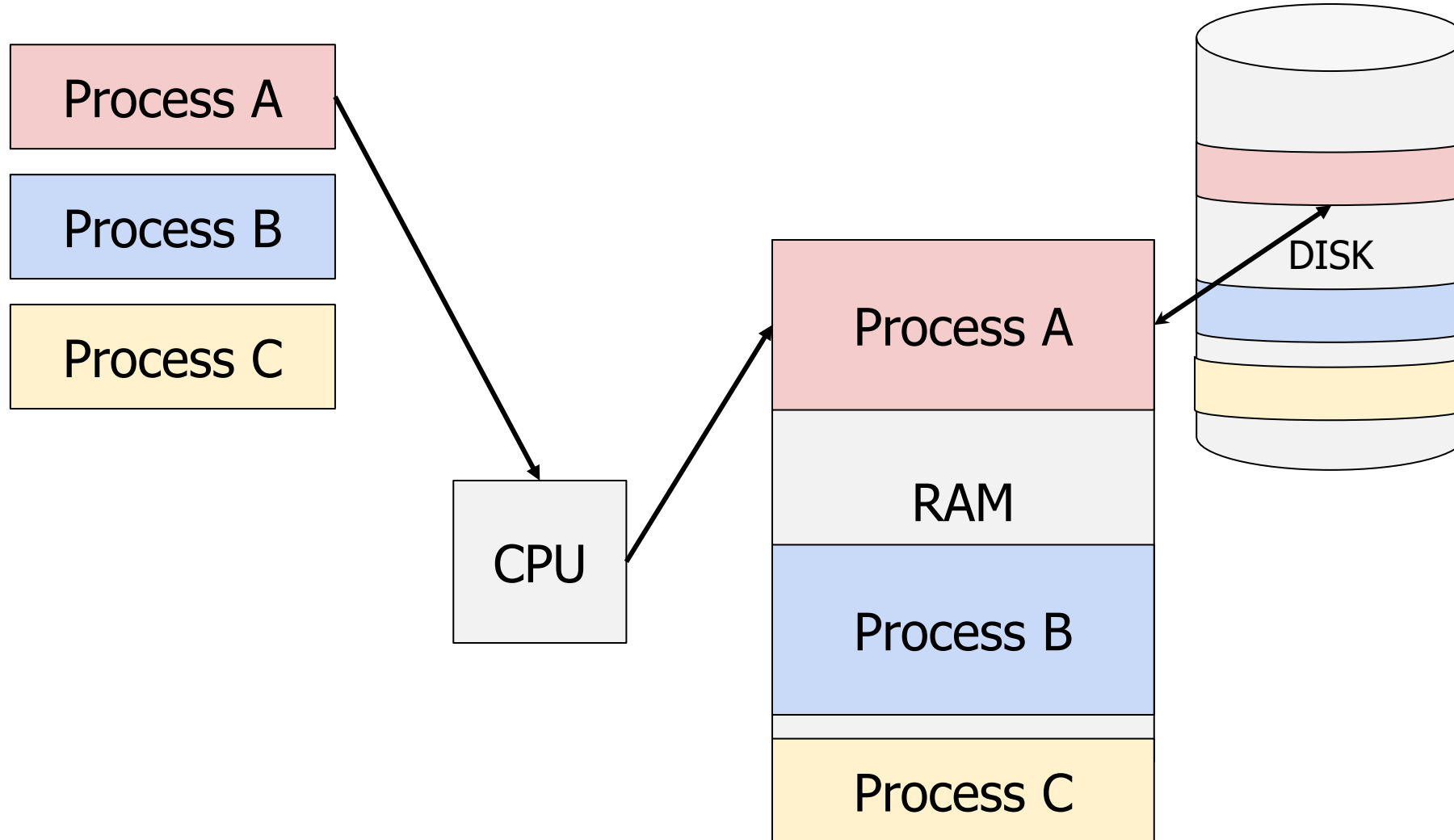
Today's Goals

- Discuss OS management of process memory with virtual memory
- Understand two virtual memory mechanisms: segmentation and paging
- Explore optimizations to memory paging

Outline

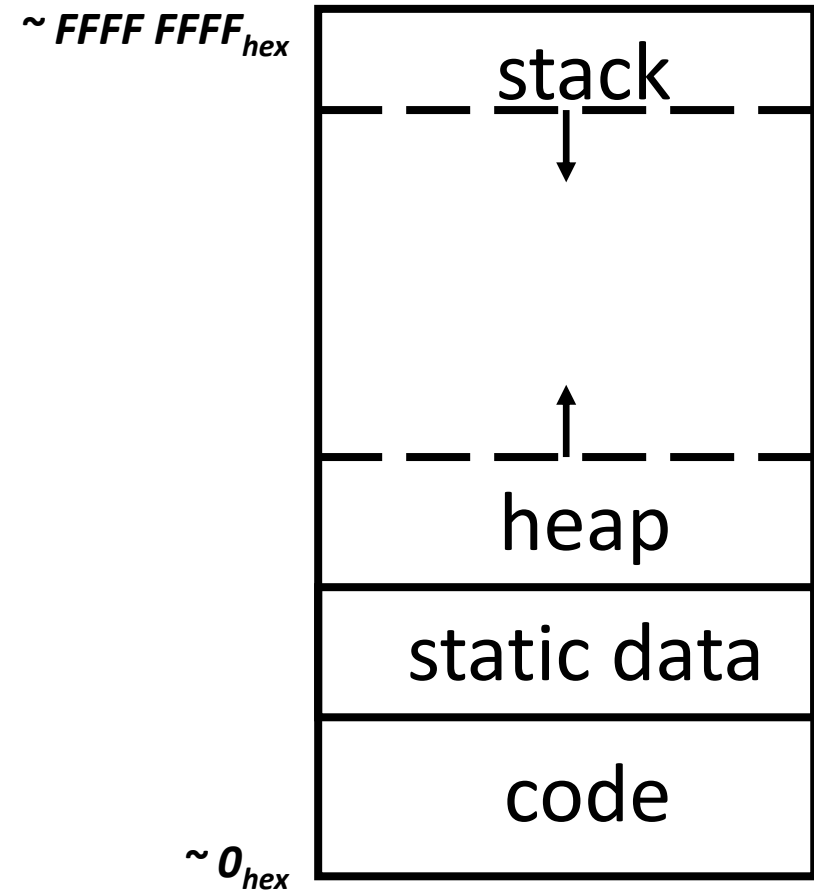
- **Address Spaces**
- Methods of address translation
 - Segmentation
 - Paging

The reality of memory in a computer



A process's view of the memory

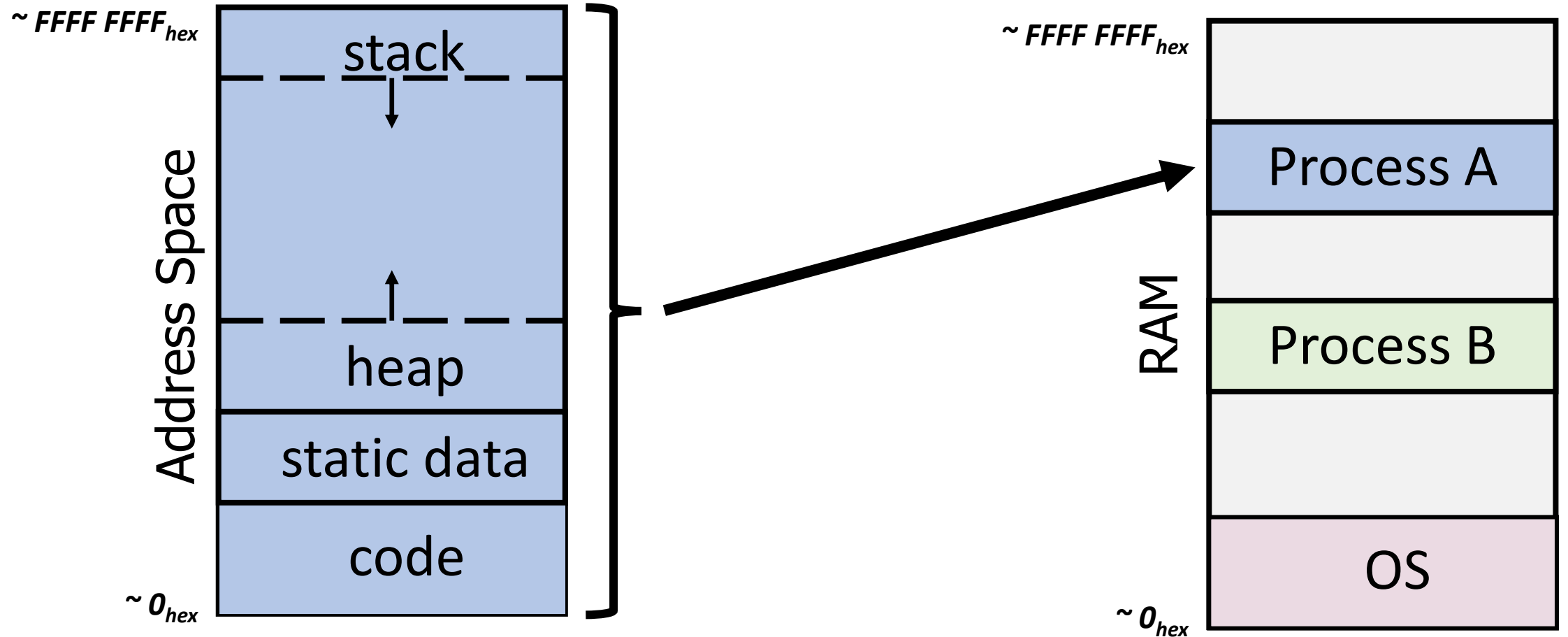
- The **Address Space** of the process
- The illusion:
 - Processes run alone on the computer
 - They have full access to every memory address
 - 2^{64} bytes of memory available to them
- The reality:
 - There are many processes
 - There is only so much RAM available



Virtual memory enables this illusion

Virtual Addresses

Physical Addresses



Why is this illusion important?

- We want to compile our programs at set addresses
 - There are alternatives to this, such as Position Independent code, but those alternatives often have performance costs
- But we can't know which addresses will be available
 - How would developers know which addresses Chrome could use safely or which addresses Powerpoint intended to use?
- Plus, the amount of RAM on systems varies widely
 - Old laptop with 512 MB, Desktop with 16 GB, Server with 256 GB
 - If they run x86-64 Linux, the same program will work on all of them
 - Specialized systems, like embedded, might not need this requirement

Goals of virtual memory

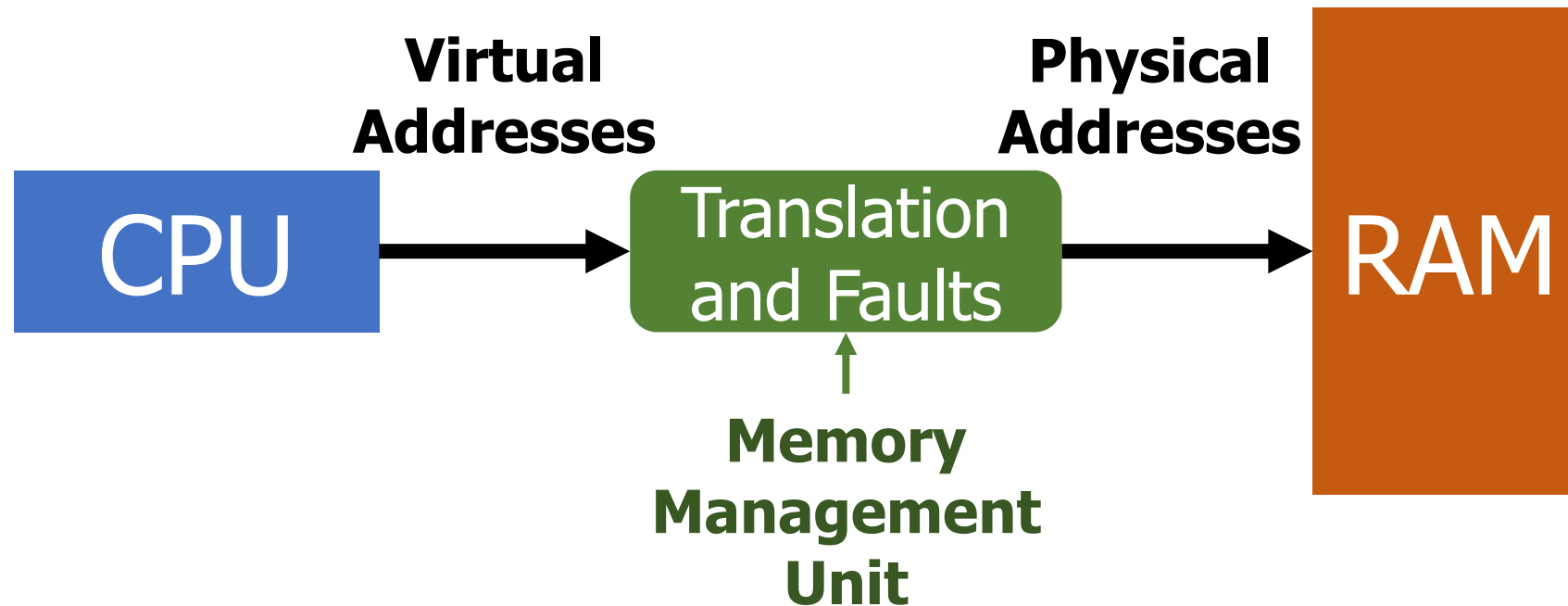
1. Independence from other programs running
2. Independence from machine hardware
3. Security
 - Applications shouldn't be able to even *read* other memory much less write
4. Efficiency
 - Allow reuse of some parts of memory
 - Code sections for threads, duplicate processes, or shared libraries
 - Don't slow down the system too much by enabling the above

Virtual memory is how the OS controls memory accesses

- I/O operations are controlled by system calls
- CPU usage is controlled by the scheduler (and interrupts)
- How can the OS control memory accesses?
 - Context switch for each memory read/write is too high of a cost
 - Hardware needs to automatically handle *most* requests

Memory Management Unit (MMU) supports virtual memory

1. Translation: hardware support for common case reads/writes
 - Configured by the OS
2. Faults: trap to OS to handle uncommon errors



Short Break + Question

- Which is bigger in practice: virtual memory or physical memory?

Short Break + Question

- Which is bigger in practice: virtual memory or physical memory?
 - 2^{64} bytes worth of addresses in both
 - Both could hold up to 16 Exabytes (~ 16000 Petabytes, ~ 16000000 Terabytes)
 - (Actually modern x86-64 CPUs limit physical memory to 2^{57} bytes = 128 PB)
 - Virtual memory: practically there isn't a limit
 - Physical memory: practically limited to amount of RAM installed
 - So, likely measured in Gigabytes
 - On almost any real system: Virtual Memory is **MUCH** larger

Outline

- Address Spaces
- **Methods of address translation**
 - **Segmentation**
 - Paging

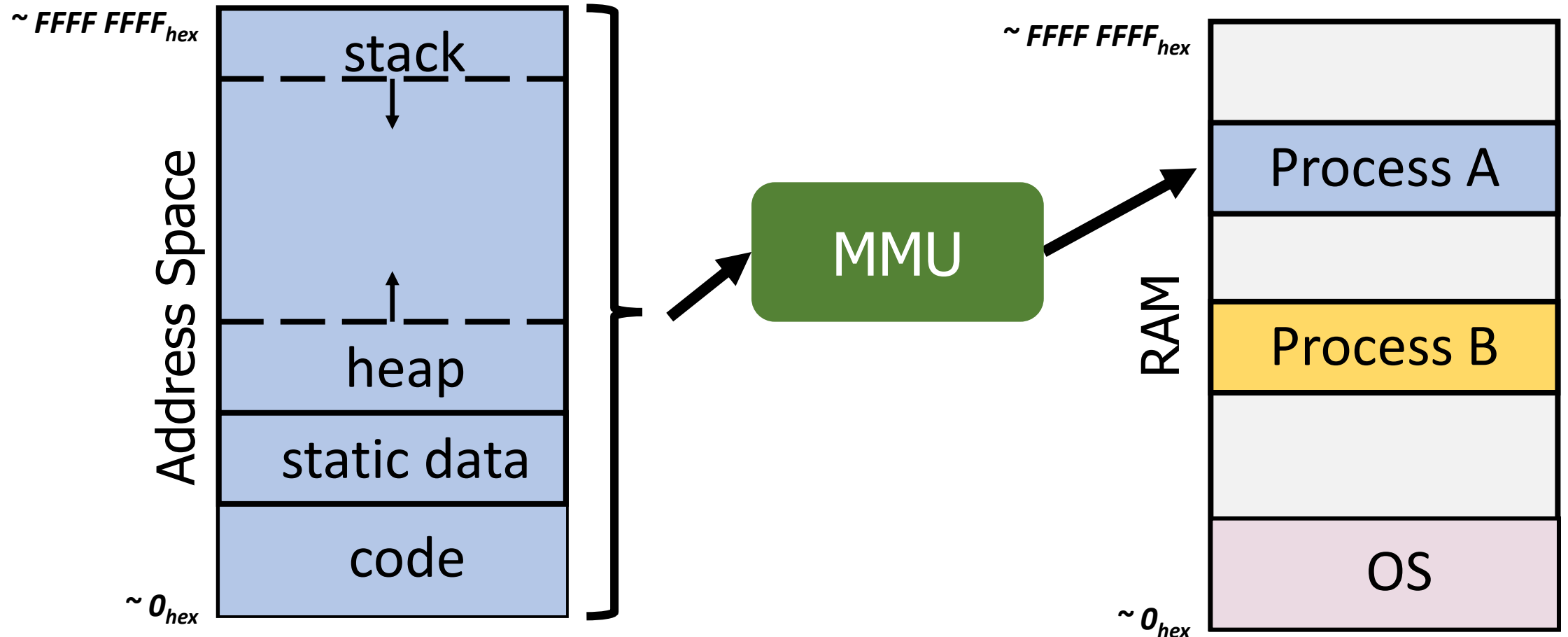
How do you implement virtual memory?

- Paging is going to be the modern computer answer
 - Combination of hardware and OS support
- But let's first do a design exploration of how you *might* do it and what the downsides are of simpler methods

Share memory by splitting between whole processes

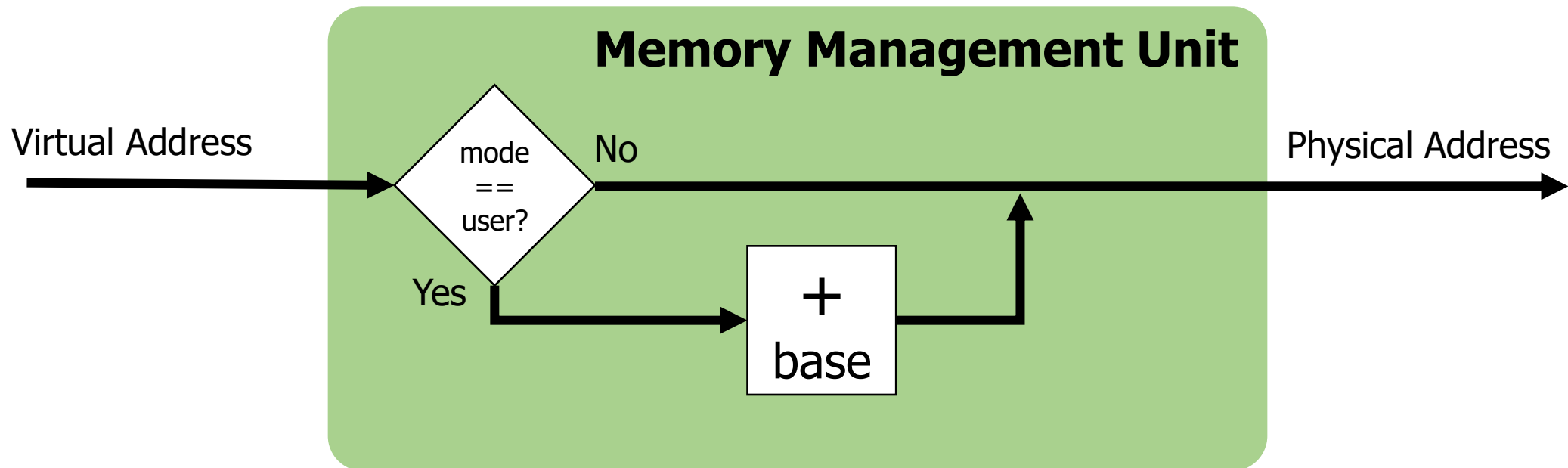
Virtual Addresses

Physical Addresses



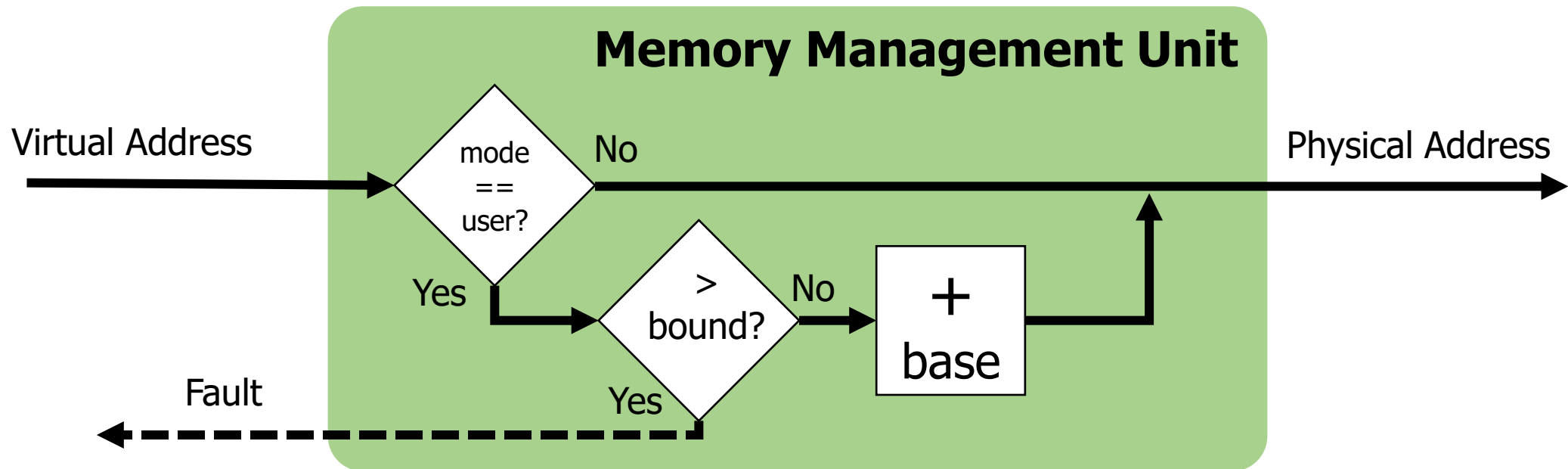
Address translation with a base register

- Divide RAM into segments, each with a separate “**base**” address
 - Processes each get their own individual segment
 - Takes advantage of processes usually being smaller than RAM
- To get a physical address from a virtual one, add to base value



Adding protection creates “Base and Bound” translation

- Add a “bound” register with maximum value of the segment
 - Memory accesses greater than bound trigger a fault
 - No need to worry about lower bound, since minimum address is $0 + \text{base}$



Base and bounds evaluation

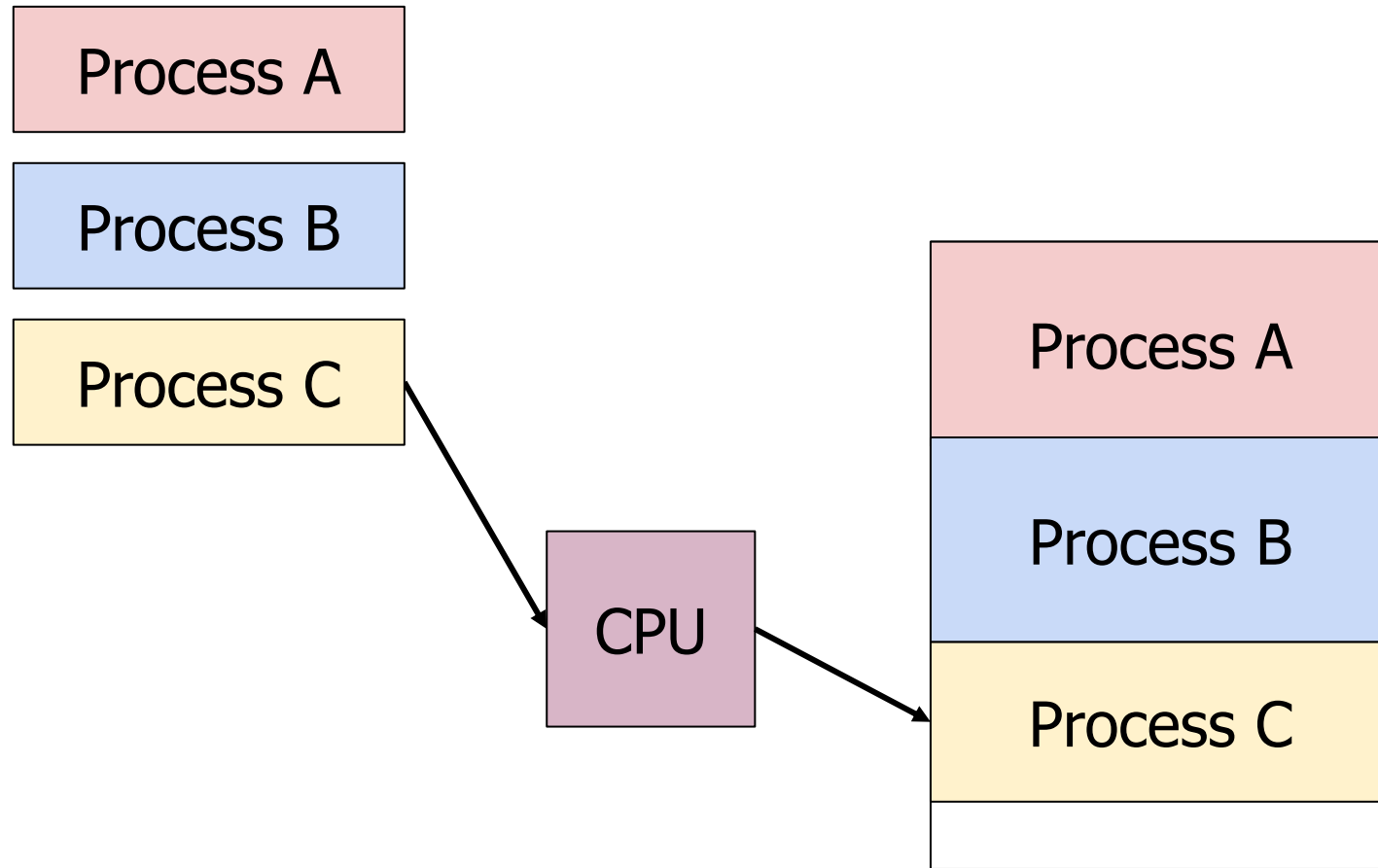
- Advantages

- Provides protection between address spaces
- Supports dynamic relocation of processes (even at runtime)
- Simple, inexpensive hardware implementation

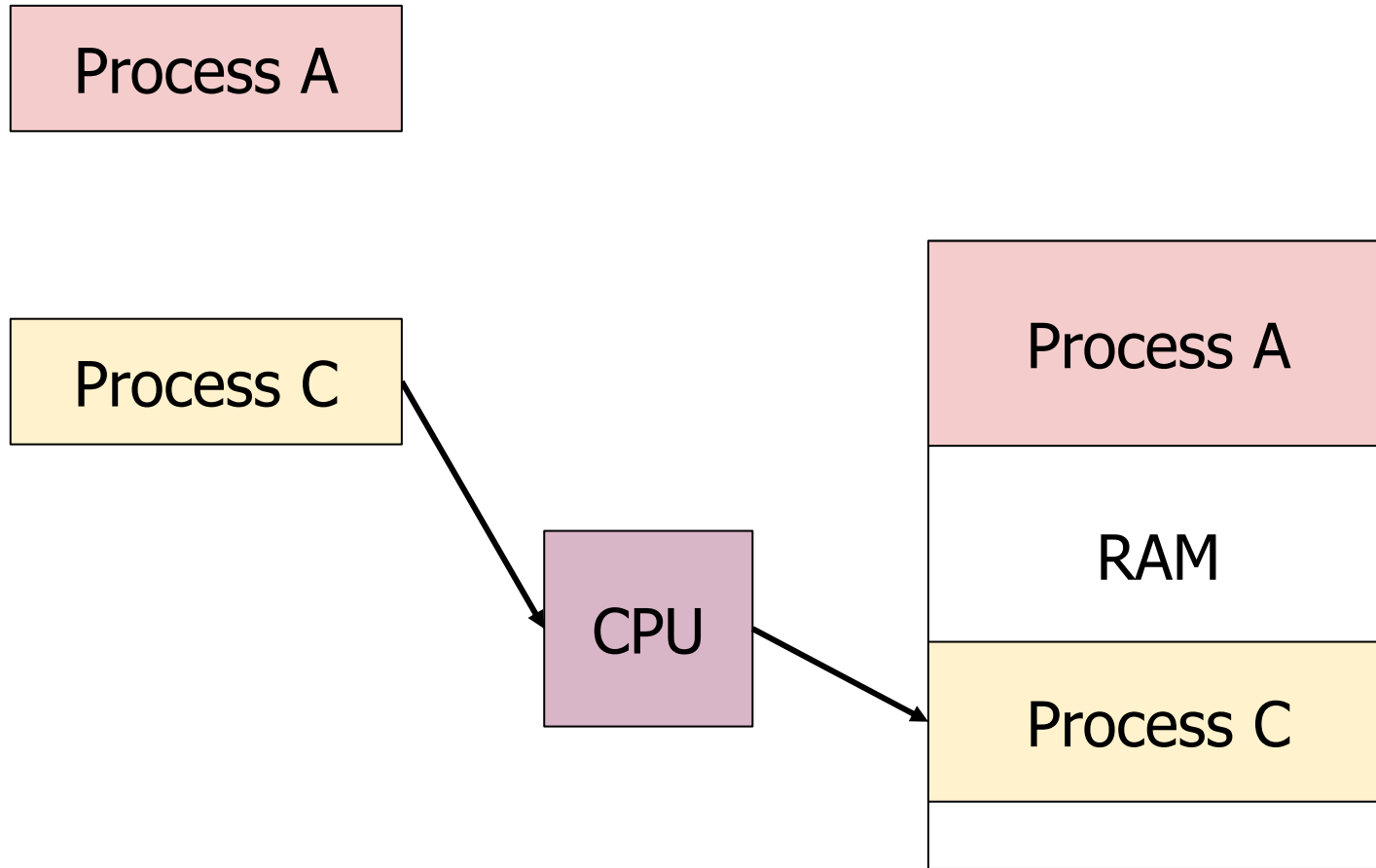
- Disadvantages

- Process must be allocated contiguous physical memory
 - Including memory between sections that might never be used
 - Large allocations end up wasting a lot of space through fragmentation
- No partial sharing of memory

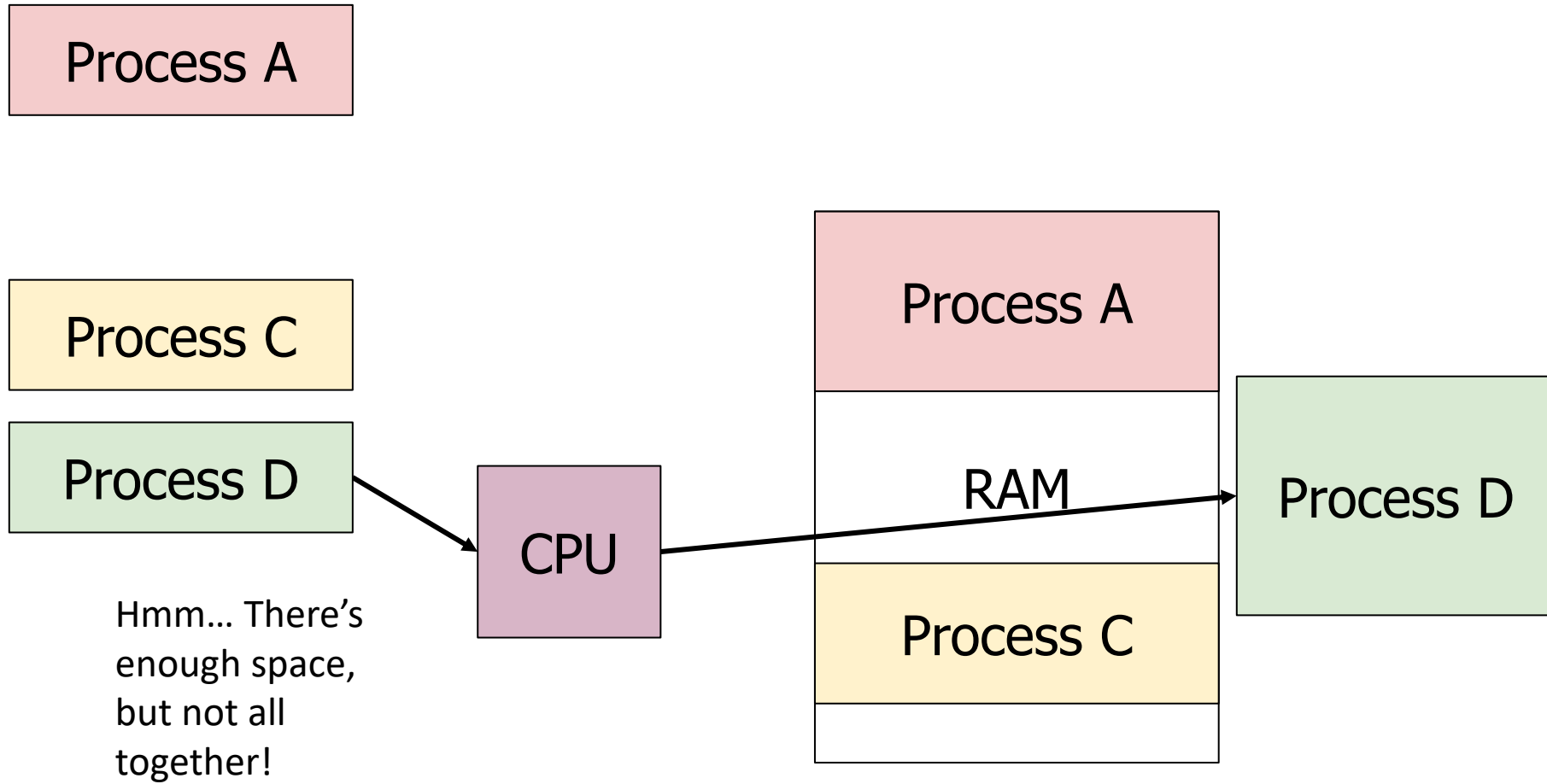
Memory fragmentation example



Memory fragmentation example



Memory fragmentation example



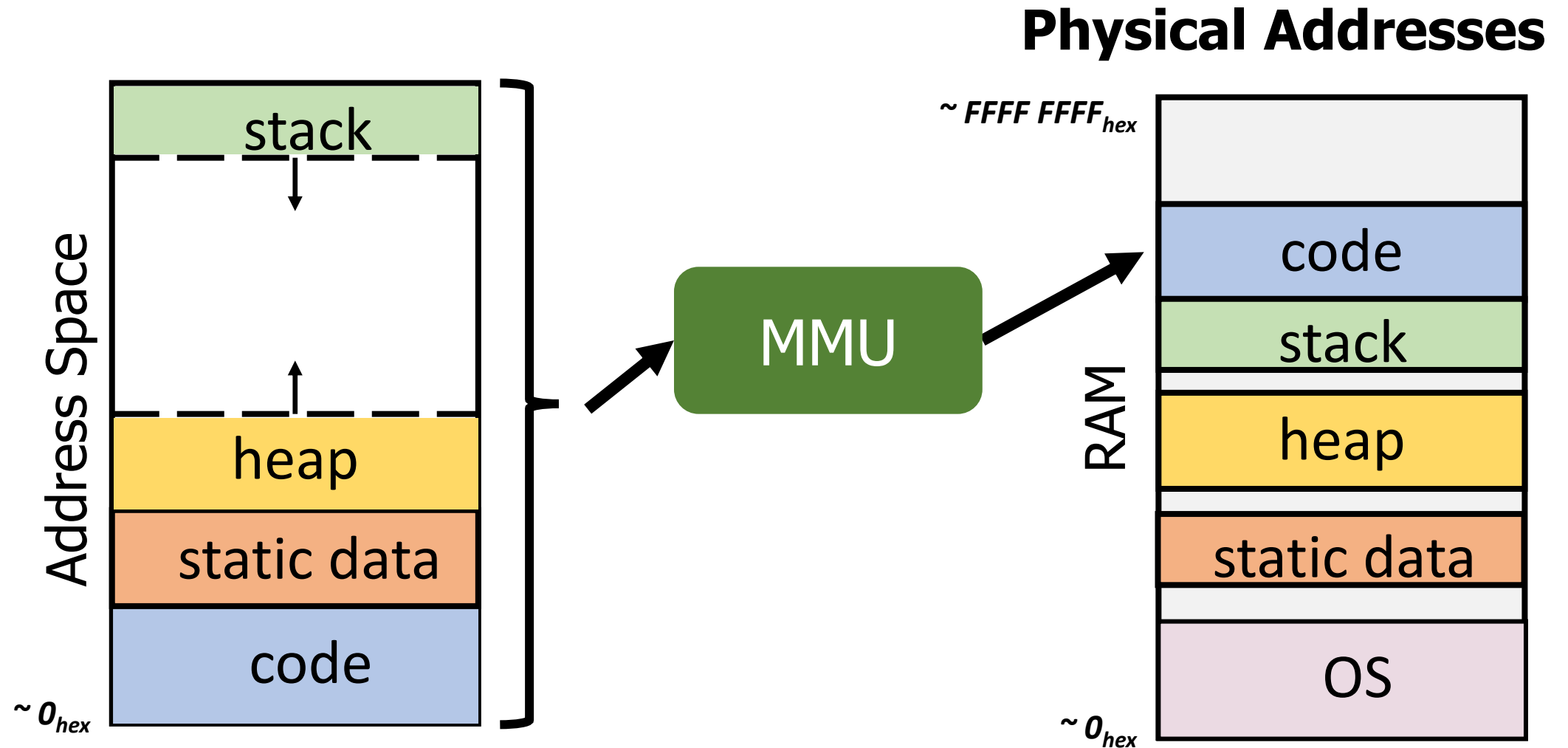
Check your understanding – base and bound

- What are the results of the following memory reads? (16-bit)
 - Base: 0xC000 Bound: 0x1FFF
 - Read 0x0010
 - Read 0x1400
 - Read 0xD000

Check your understanding – base and bound

- What are the results of the following memory reads? (16-bit)
 - Base: 0xC000 Bound: 0x1FFF
 - Read 0x0010 -> **0xC010**
 - Read 0x1400 -> **0xD400**
 - Read 0xD000 -> **Fault (greater than 0x1FFF bound)**

What if we split the code into multiple base/bound segments?



Segmentation design

- Select some number of “segments” that processes may have
 - Separate base and bound register for each one
- Need to distinguish which accesses correspond to which segment
 - Solution: use top few bits of the virtual address, $\log_2(\text{number of segments})$
 - 00 -> segment 0
 - 01 -> segment 1
 - etc.
 - Only add remaining lower bits to the base register
- Look at most significant bits of an address to determine segment
 - 0x1234 -> 0b**00**01_0010_0011_0100 -> Segment 0 (assuming 4 segments)

Memory Management Unit for segmentation

- Similar comparison and addition hardware as before
- New **segment table** to select correct base and bounds
 - Bits from virtual address decide on the correct segment
 - Segment decides the proper base and bound selection
 - Can also apply permissions to individual segments

Segment	Base	Bound	Permissions
0	0x2000	0x06FF	Read/Execute
1	0x0000	0x04FF	Read/Write
2	0x3000	0x0FFF	Read/Write
3	0x0000	0x0000	None

Example

← Code

← Stack

← Data

← Unused

OS management of processes with segmentation

- On context switch
 - Hardware changes to kernel mode and deactivates the MMU
 - Save process's segment table with the rest of the process data
 - Load new process's segment table into the MMU
 - Change to user mode and jump to new process
- x86 example
 - No table, but rather registers for each segment
 - Stack Segment, Code Segment, Data Segment
 - Extra Segment, F Segment, G Segment

Segmentation evaluation

- Advantages

- Sparse allocation of address space (most of it goes in no segment at all)
- Stack and heap segments can grow
- Different protection for different segments
 - Only execute or write where it makes sense to
- Still possible to do dynamic relocation and hardware still relatively simple

- Disadvantages

- Still results in fragmentation of memory
 - Entire section must fit
 - But sections are irregularly sized

Quick question – segmentation (16-bit address space)

- How many bits are used for the segment?

Segment	Base	Bound	Permissions
0	0x0000	0x06FF	Read/Execute
1	0x0700	0x02FF	Read/Write
2	0x3C00	0x01FF	Read/Write
3	0x1800	0x01FF	Read/Execute
4	0x4200	0x0400	Read/Execute
5	0x0000	0x0000	None
6	0x0000	0x0000	None
7	0x0000	0x0000	None

Quick question – segmentation (16-bit address space)

- How many bits are used for the segment?
- Three bits (8 choices)
- Placed as most significant bits
- Lower 13 bits are added to base

Segment	Base	Bound	Permissions
0	0x0000	0x06FF	Read/Execute
1	0x0700	0x02FF	Read/Write
2	0x3C00	0x01FF	Read/Write
3	0x1800	0x01FF	Read/Execute
4	0x4200	0x0400	Read/Execute
5	0x0000	0x0000	None
6	0x0000	0x0000	None
7	0x0000	0x0000	None

Break + Practice – segmentation (16-bit address space)

- Which segment is each?
- Read 0x0200
- Read 0x0500
- Write 0x0410
- Read 0x4004
- Write 0x5004

Segment	Base	Bound	Permissions
0	0x0000	0x06FF	Read/Execute
1	0x0700	0x02FF	Read/Write
2	0x3C00	0x01FF	Read/Write
3	0x1800	0x01FF	Read/Execute
4	0x4200	0x0400	Read/Execute
5	0x0000	0x0000	None
6	0x0000	0x0000	None
7	0x0000	0x0000	None

Upper 3 bits of address are the segment
Lower 13 bits of address are appended to Base

Break + Practice – segmentation (16-bit address space)

- Which segment is each?

Segment 0

- Read 0x0200
- Read 0x0500
- Write 0x0410

Segment 2

- Read 0x4004
- Write 0x5004

Segment	Base	Bound	Permissions
0	0x0000	0x06FF	Read/Execute
1	0x0700	0x02FF	Read/Write
2	0x3C00	0x01FF	Read/Write
3	0x1800	0x01FF	Read/Execute
4	0x4200	0x0400	Read/Execute
5	0x0000	0x0000	None
6	0x0000	0x0000	None
7	0x0000	0x0000	None

Upper 3 bits of address are the segment
Lower 13 bits of address are appended to Base

Break + Practice – segmentation (16-bit address space)

- Do full translation

Segment 0

- Read 0x0200
- Read 0x0500
- Write 0x0410

Segment 2

- Read 0x4004
- Write 0x5004

Segment	Base	Bound	Permissions
0	0x0000	0x06FF	Read/Execute
1	0x0700	0x02FF	Read/Write
2	0x3C00	0x01FF	Read/Write
3	0x1800	0x01FF	Read/Execute
4	0x4200	0x0400	Read/Execute
5	0x0000	0x0000	None
6	0x0000	0x0000	None
7	0x0000	0x0000	None

Upper 3 bits of address are the segment
Lower 13 bits of address are appended to Base

Break + Practice – segmentation (16-bit address space)

- Do full translation

Segment 0

- Read 0x0200 -> 0x0200
- Read 0x0500 -> 0x0500
- Write 0x0410 -> Fault
(Permission)

Segment 2

- Read 0x4004
- Write 0x5004

Segment	Base	Bound	Permissions
0	0x0000	0x06FF	Read/Execute
1	0x0700	0x02FF	Read/Write
2	0x3C00	0x01FF	Read/Write
3	0x1800	0x01FF	Read/Execute
4	0x4200	0x0400	Read/Execute
5	0x0000	0x0000	None
6	0x0000	0x0000	None
7	0x0000	0x0000	None

Upper 3 bits of address are the segment
Lower 13 bits of address are appended to Base

Break + Practice – segmentation (16-bit address space)

- Do full translation

Segment 0

- Read 0x0200 -> 0x0200
- Read 0x0500 -> 0x0500
- Write 0x0410 -> Fault
(Permission)

Segment 2

- Read 0x4004 -> 0x3C04
- Write 0x5004 -> Fault (Bound) [0x1004 > 0x01FF]

Segment	Base	Bound	Permissions
0	0x0000	0x06FF	Read/Execute
1	0x0700	0x02FF	Read/Write
2	0x3C00	0x01FF	Read/Write
3	0x1800	0x01FF	Read/Execute
4	0x4200	0x0400	Read/Execute
5	0x0000	0x0000	None
6	0x0000	0x0000	None
7	0x0000	0x0000	None

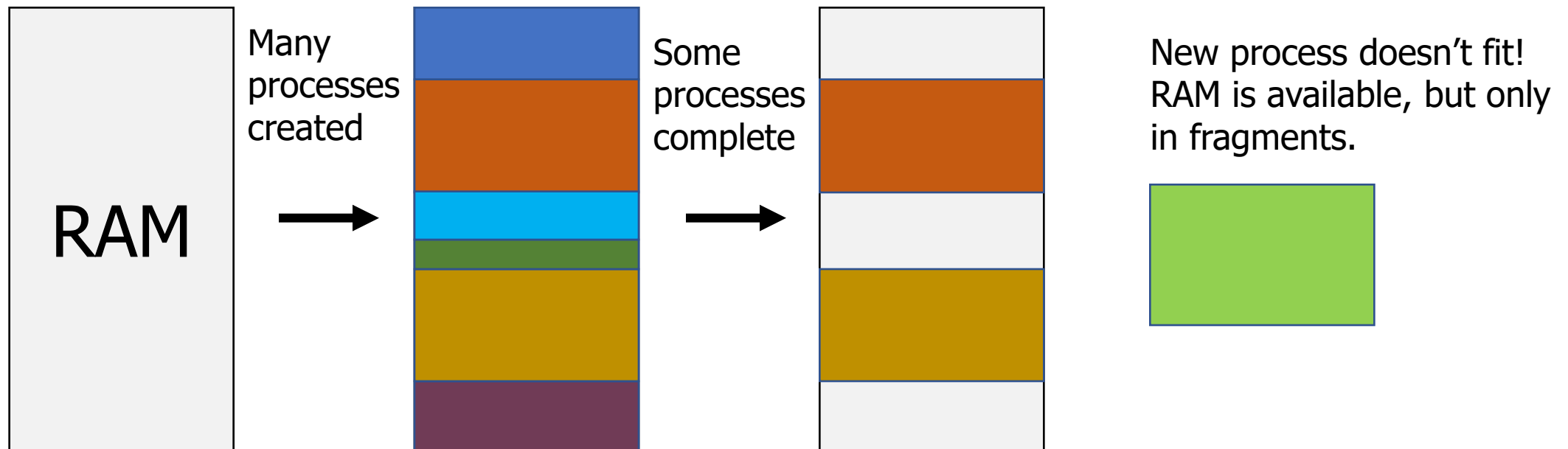
Upper 3 bits of address are the segment
Lower 13 bits of address are appended to Base

Outline

- Address Spaces
- **Methods of address translation**
 - Segmentation
 - **Paging**

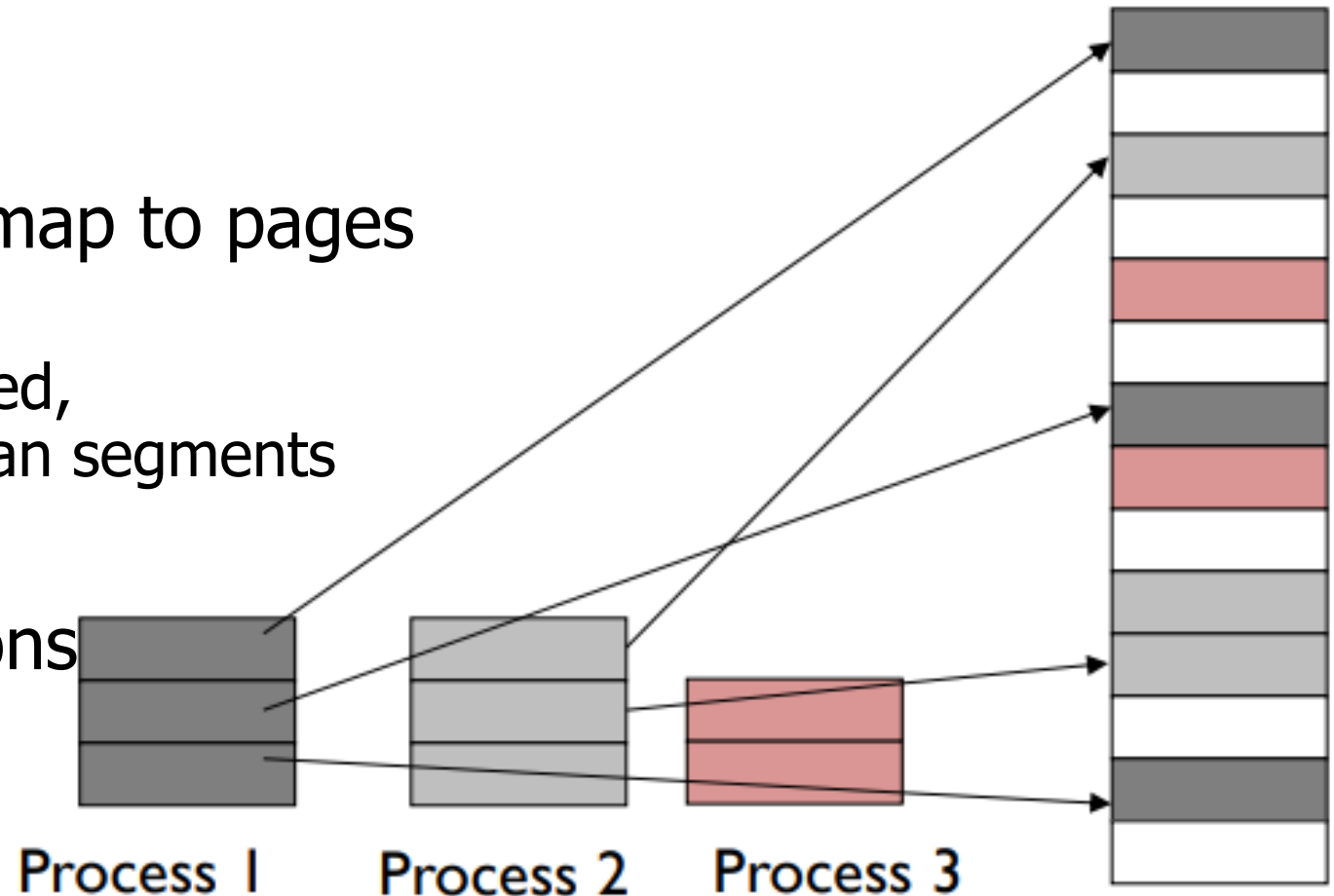
Improving upon segmentation

- Segmentation had some good features
 - Address space does not need to be contiguous
 - Segments can grow when needed
- But irregularly-sized segments lead to fragmentation



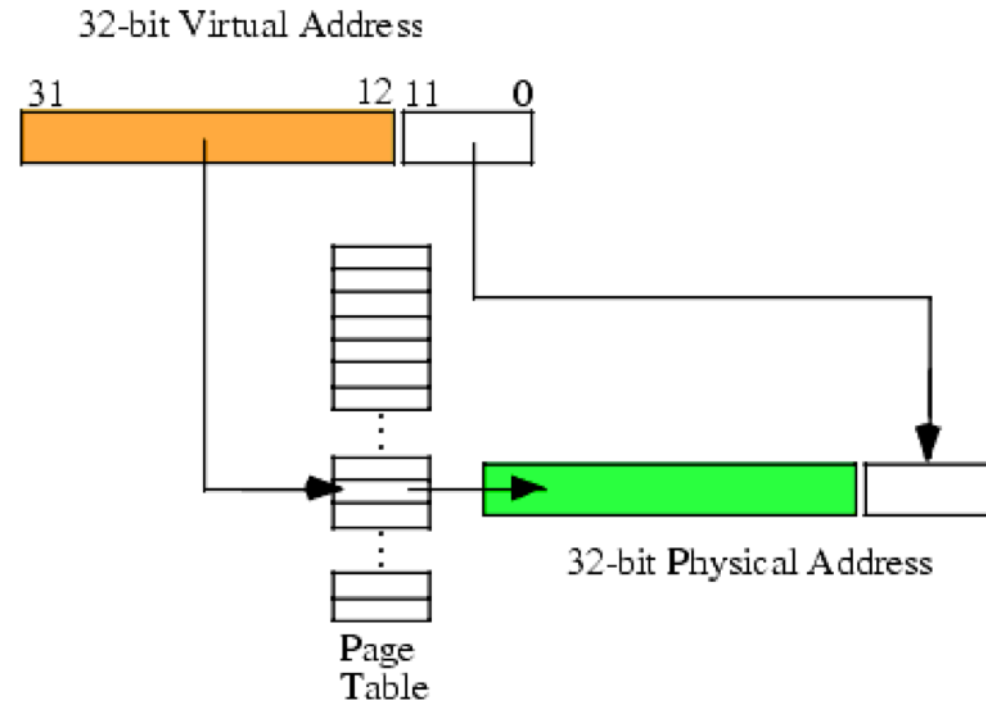
Solution to fragmentation: *pages* of memory

- Divide memory into small, **fixed-sized** chunks
 - These are called “pages”
- Pages of virtual memory map to pages of physical memory
 - Like segments were mapped, but ***many more*** pages than segments
- Processes and their sections can be mapped to any place in memory



Page table translates virtual addresses to physical addresses

- Use topmost bits of virtual address to select page table entry
 - One page table entry per each virtual page
- Combine address at page table entry with bottommost bits
 - Actually just concatenate the two
- Just like segment tables, there will be a different page table for each process



Paging versus segmentation

- Every page of virtual memory maps to a page of physical memory
 - No need for a bound anymore
 - Above a bound would just be within the bounds of some other page
- We don't pick the number of pages, we pick page size
 - $\text{Number of pages} = \text{Size of memory} / \text{Size of Page}$
- Result: **Way** more pages than there were segments
 - 4 kB pages with 4 GB of RAM -> ~1 million pages
 - Need to keep page table in memory rather than hardware registers
 - Hardware register points at the base of the page table

Process A

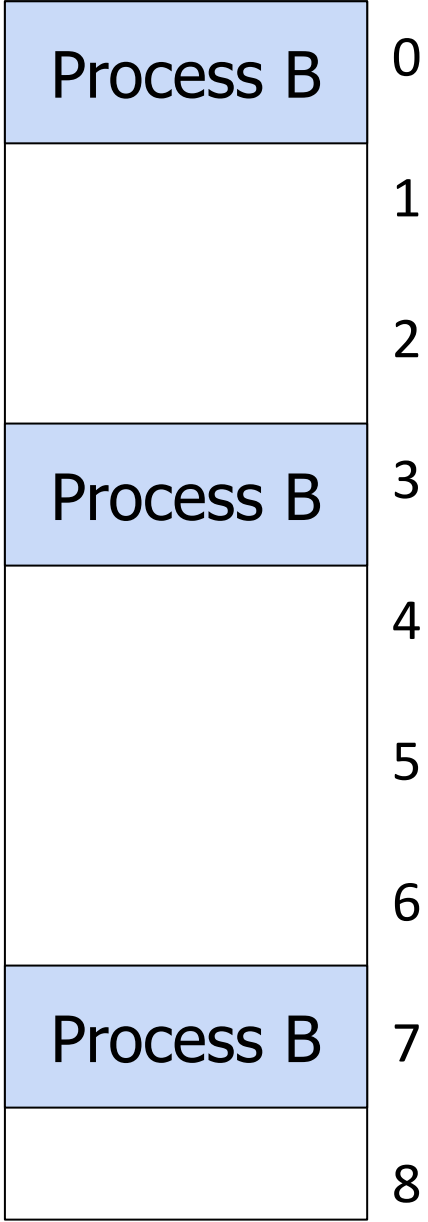
Process B

Process B Page Table

VPN	PPN	Valid?
0		
1		
2		
3		
4		
5		
6		
7		
8		

CPU

Virtual Memory
(Process B Only!)



Process A

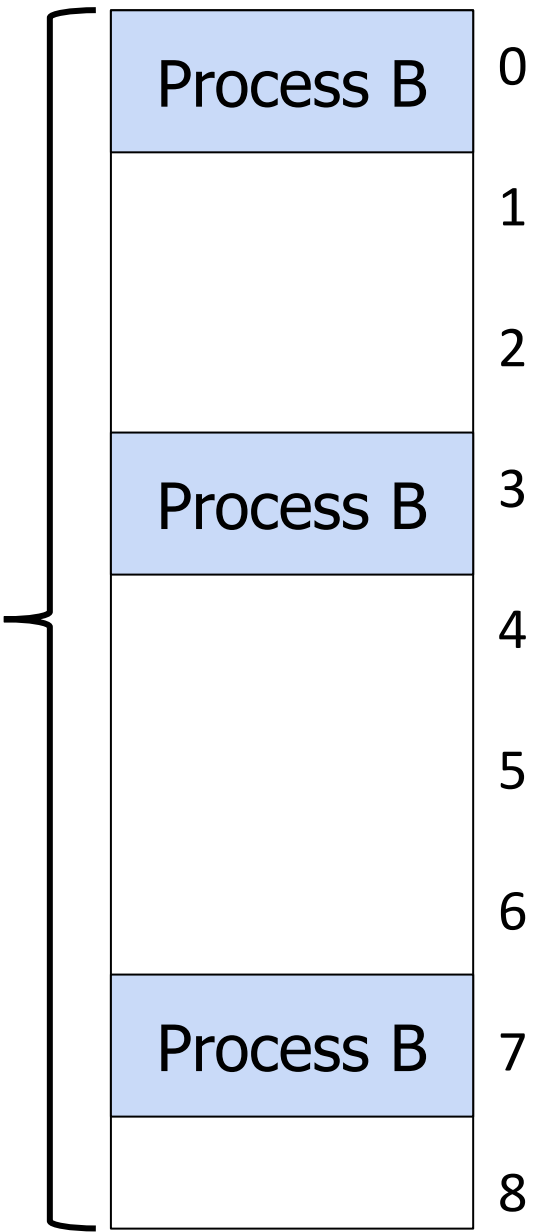
Process B

Process B Page Table

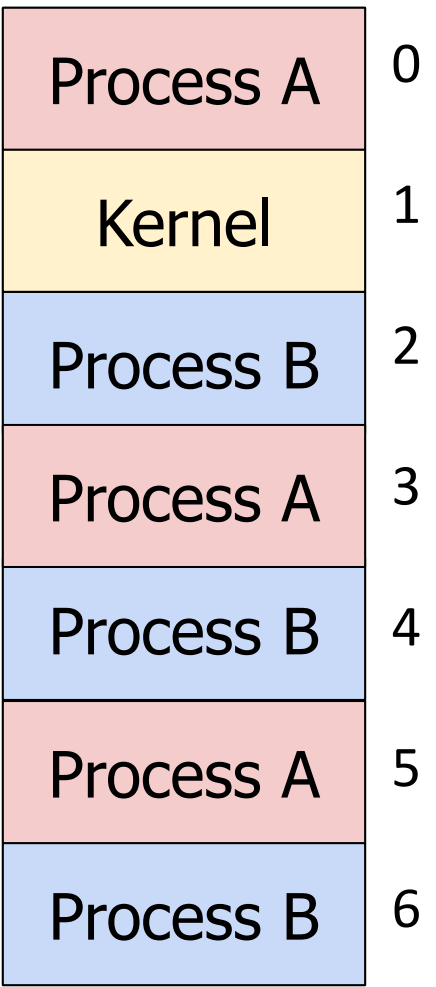
VPN	PPN	Valid?
0		
1		
2		
3		
4		
5		
6		
7		
8		

CPU

Virtual Memory
(Process B Only!)



Physical Memory (RAM)
Shared



Process A

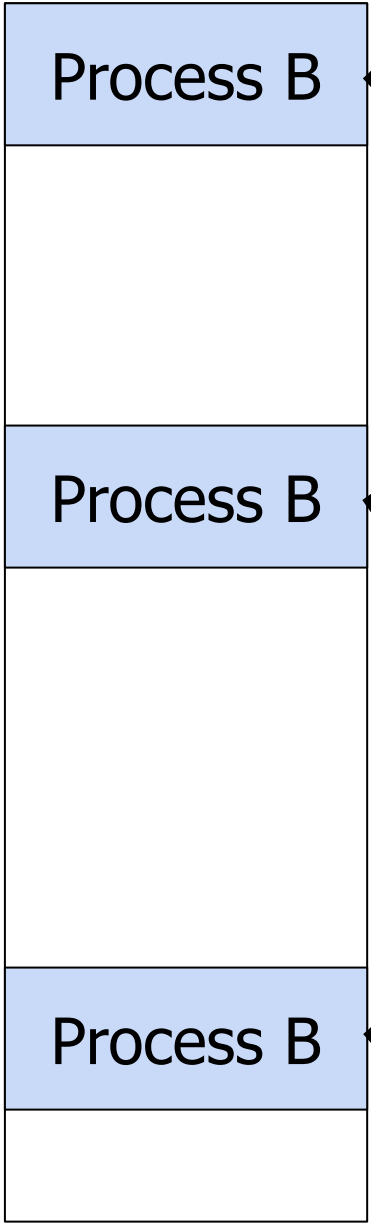
Process B

Process B Page Table

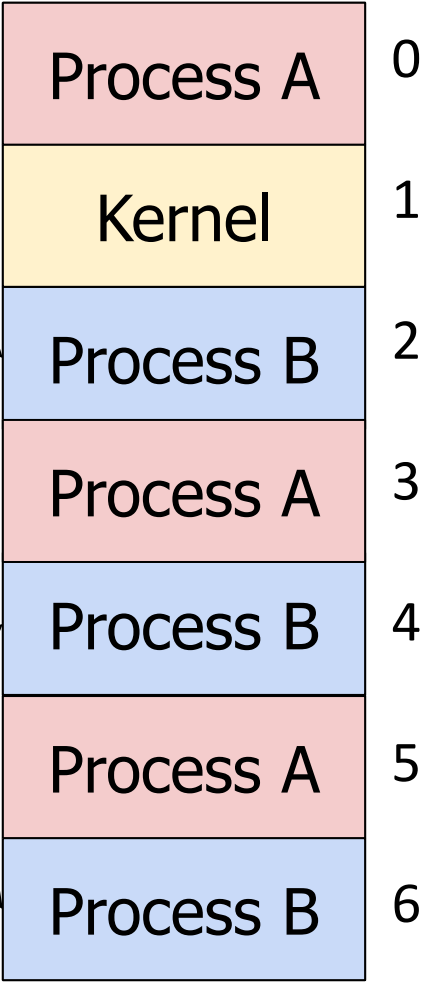
VPN	PPN	Valid?
0	2	1
1		
2		
3	6	1
4		
5		
6		
7	4	1
8		

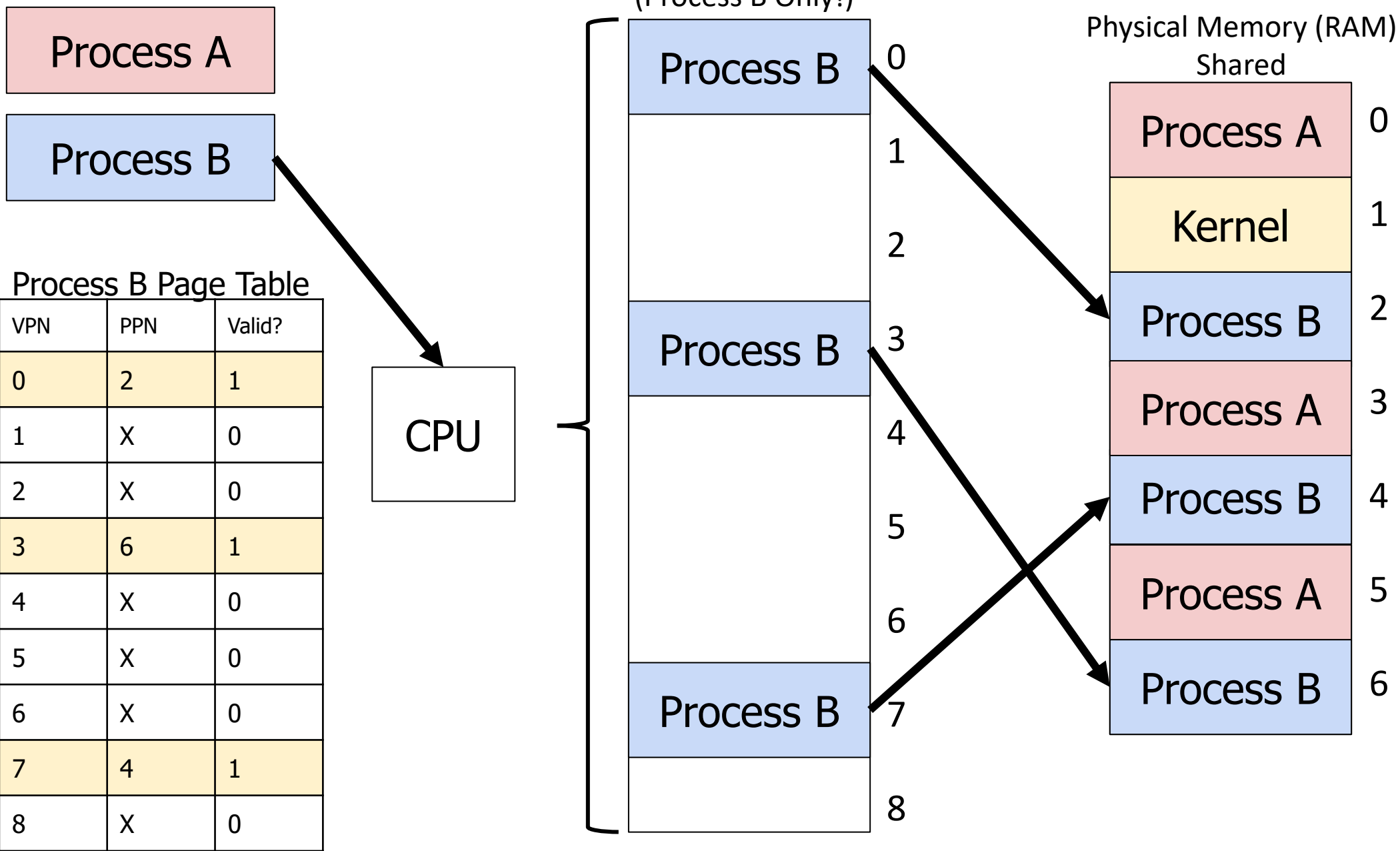
CPU

Virtual Memory
(Process B Only!)

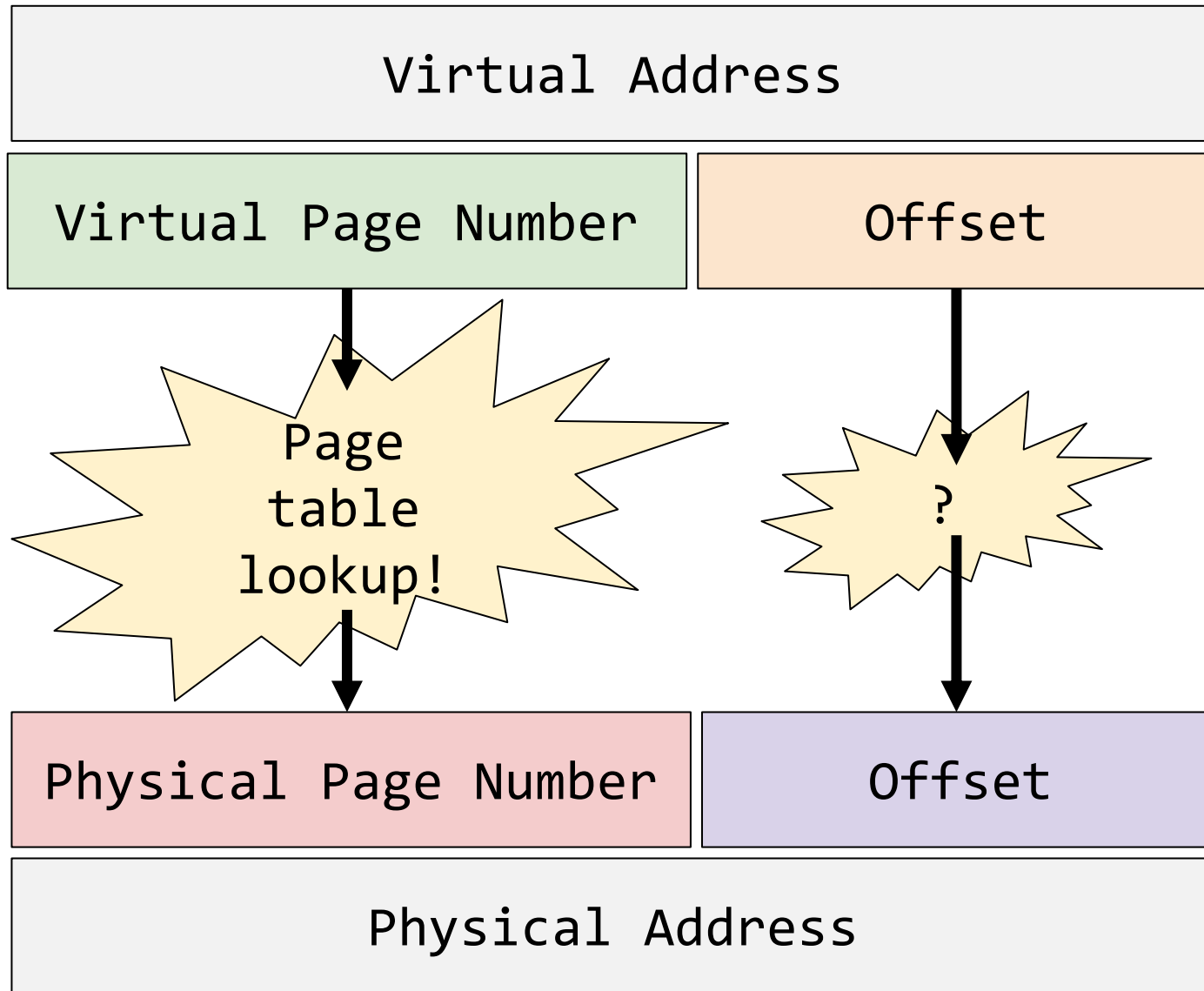


Physical Memory (RAM)
Shared



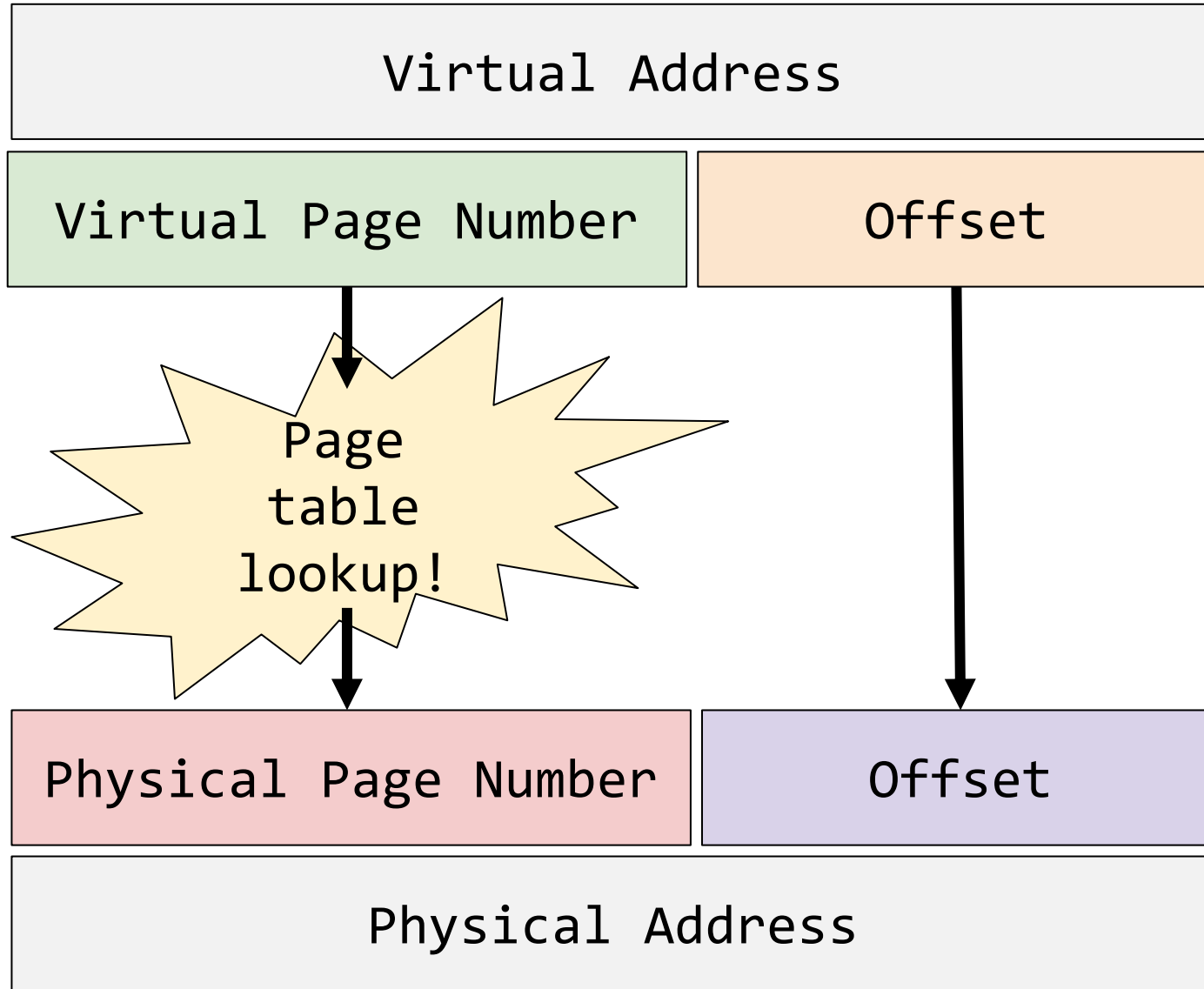


Check your understanding – virtual address translation



Do we need to translate the lower bits of a virtual address?

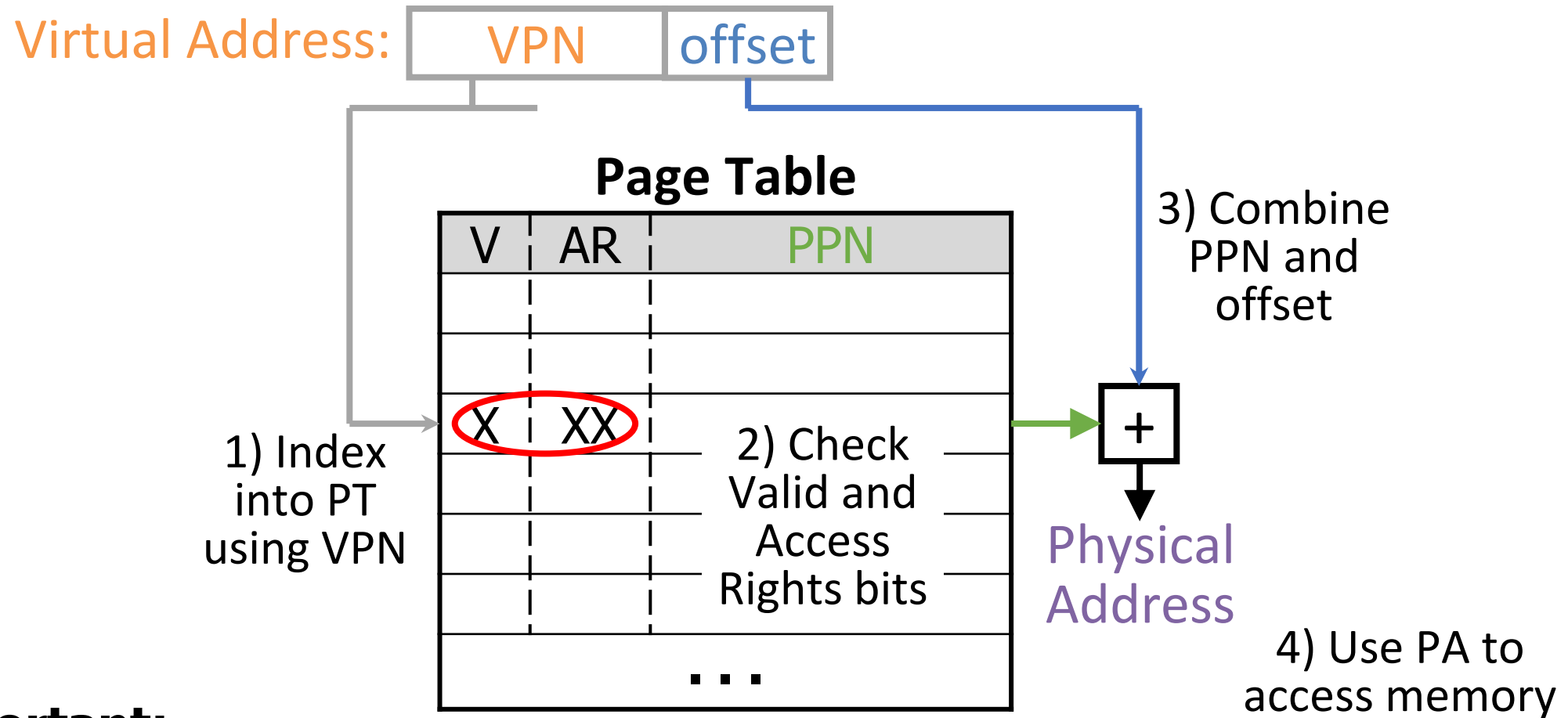
Check your understanding – virtual address translation



Do we need to translate the lower bits of a virtual address?

No. Those are used to determine word/byte within the page.

Steps to translating virtual addresses with paging



Important:

This is all done in hardware!! OS is not involved unless it faults

Break + Virtual Memory Practice

Assume `a` starts at 0x3000 (virtual)

Ignore instruction fetches and access to `i` and `sum` (they're in registers)

Code

```
int sum = 0;
for(int i=0; i<N; i++){
    sum += a[i];
}
```

Virtual Address Accesses

```
load 0x3000
load 0x3004
load 0x3008
load 0x300C
```

Break + Virtual Memory Practice

Assume `a` starts at 0x3000 (virtual)

Ignore instruction fetches and access to `i` and `sum` (they're in registers)

Code

```
int sum = 0;
for(int i=0; i<N; i++){
    sum += a[i];
}
```

Virtual Address Accesses

load 0x3000
load 0x3004
load 0x3008
load 0x300C

Physical Address Accesses

load 0x100C
load 0x7000
load 0x100C
load 0x7004
load 0x100C
load 0x7008
load 0x100C
load 0x700C

- Which physical address is within the page table?
- At what physical address does `a` start?

Break + Virtual Memory Practice

Assume `a` starts at 0x3000 (virtual)

Ignore instruction fetches and access to `i` and `sum` (they're in registers)

Code

```
int sum = 0;
for(int i=0; i<N; i++){
    sum += a[i];
}
```

Virtual Address Accesses

load 0x3000
load 0x3004
load 0x3008
load 0x300C

Physical Address Accesses

load 0x100C
load 0x7000
load 0x100C
load 0x7004
load 0x100C
load 0x7008
load 0x100C
load 0x700C

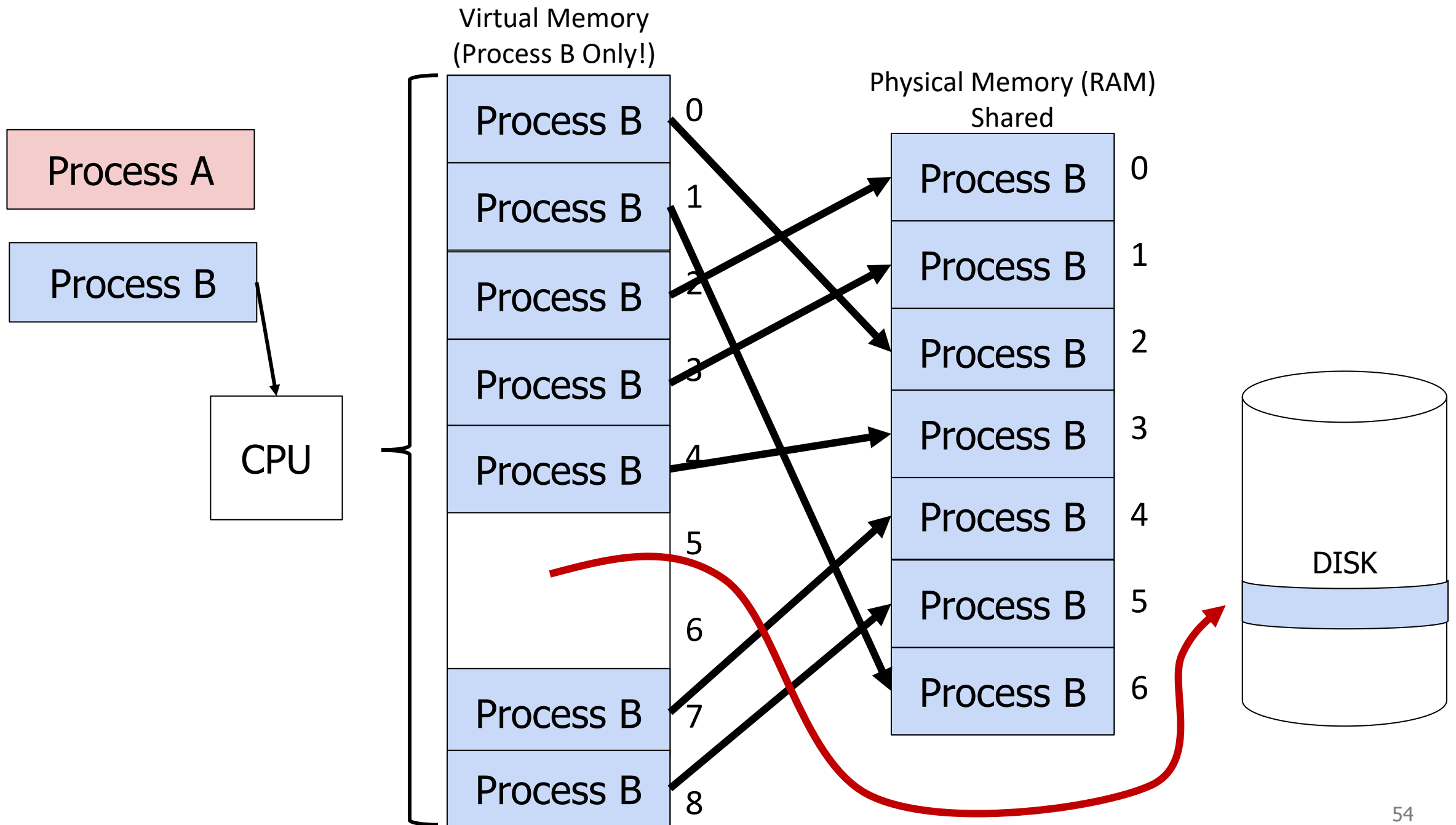
- Which physical address is within the page table? **0x100C**
 - All accesses are within the same Page
- At what physical address does `a` start? **0x7000**
 - Accesses to the array step by 4 byte increments

How the OS deals with memory in a paging system

1. How do the OS and program agree on addresses?
 - Each program can use any virtual addresses it wants
 - Some default for compiler/OS pairing
 - OS controls physical memory layout in RAM and maps the two
2. How does the OS move memory around without messing up programs?
 - Just update the record in the page table
 - Process doesn't know the difference
3. How to protect OS and process memory from other processes?
 - Ensure that virtual pages from a process never map to physical pages for another
 - But we can share physical pages for threads or shared libraries if we want!

Dealing with processes bigger than memory

- Paging allows the OS to support processes larger than RAM
 - Just leave the virtual pages unmapped
 - When a load occurs to the unmapped page, a fault triggers the OS
 - Which can then load the needed page into RAM from disk
 - (and push some other page onto disk)



OS management of processes with paging

- When loading a process
 - OS places actual memory into physical pages in RAM
 - OS creates page table for the process
 - OS decides access permissions to different pages
 - OS connects to shared libraries already in RAM
- When a context switch occurs
 - OS changes which page table is in use (%CR3 register in x86)
- When a fault occurs
 - OS decides how to handle it. (Invalid access or missing page?)

Paging evaluation

- Advantages
 - Still sparse allocation of address space and growing segments as needed
 - Still different protection for different segments
 - Only execute or write where it makes sense to
 - Still possible to do dynamic relocation and hardware still relatively simple
 - No fragmentation of main memory
 - Pages can fit anywhere they need to
 - Can load processes bigger than main memory!

Paging evaluation (continued)

- Disadvantages
 - More work on the part of the OS to set up a process
 - Only a problem if we create processes frequently
 - Page tables are slow to access
 - Page tables need to be stored in memory due to size
 - MMU only holds the base address of the page table and reads from it
 - Two memory loads per load!!!
 - Going to have to fix this...
 - Page tables require a lot of storage space
 - Mapping must exist for each virtual page, even if unused
 - Becomes a serious issue on 64-bit systems

Outline

- Address Spaces
- Methods of address translation
 - Segmentation
 - Paging