CS111, Lecture 22
Dynamic Address Translation

CS111 Topic 4: Virtual Memory

Virtual Memory *- How can one set of memory be shared among several processes? How can the operating system manage access to a limited amount of system memory?*

Virtual Memory Introduction	→	Dynamic Address Translation	→	Demand Paging	→	The Clock Algorithm
Lecture 21		**Today**		**Lecture 23**		**Lecture 24**

assign6: implement *demand paging* system to translate addresses and load/store memory contents for programs as needed.

Learning Goals

- Understand the benefits of dynamic address translation
- Reason about the tradeoffs in different ways to implement dynamic address translation

Plan For Today

- **Recap:** virtual memory and dynamic address translation
- Approach #2: Multiple Segments
- Approach #3: Paging

Plan For Today

- **Recap: virtual memory and dynamic address translation**
- Approach #2: Multiple Segments
- Approach #3: Paging

Virtual memory is a mechanism for multiple processes to simultaneously use system memory.

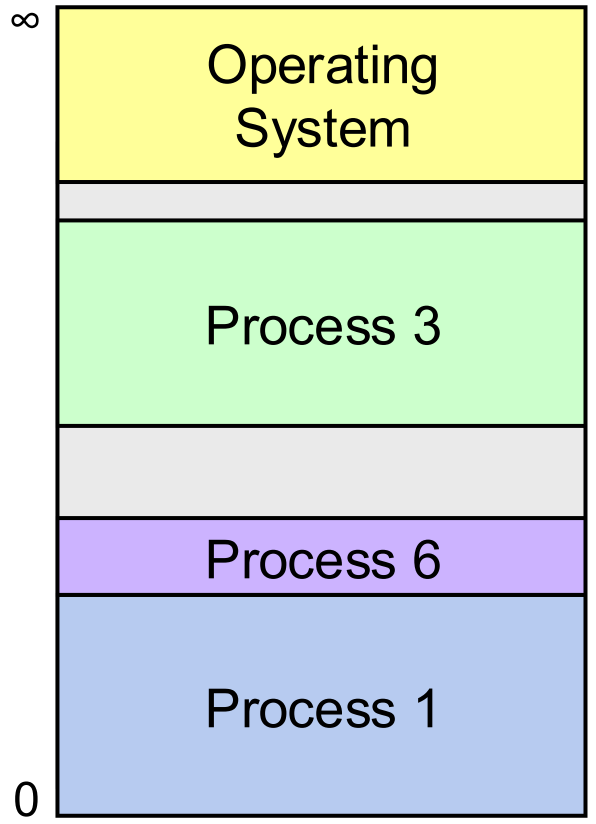
Sharing Memory

We want to allow multiple processes to simultaneously use system memory. Our goals are:

- **Multitasking** – allow multiple processes to be memory-resident at once
- **Transparency** – no process should need to know memory is shared. Each must run regardless of the number and/or locations of processes in memory.
- **Isolation** – processes must not be able to corrupt each other
- **Efficiency** (both of CPU and memory) – shouldn't be degraded badly by sharing

Load-Time Relocation

- When a process is loaded to run, place it in a designated memory space.
- That memory space is for everything for that process – stack/data/code
- Interesting fact – when a program is compiled, it is compiled assuming its memory starts at address 0. Therefore, we must update its addresses when we load it to match its real starting address.
- Use first-fit or best-fit allocation to manage available memory.
- **Problems:** isolation, deciding memory sizes in advance, fragmentation, updating addresses when loading



Dynamic Address Translation

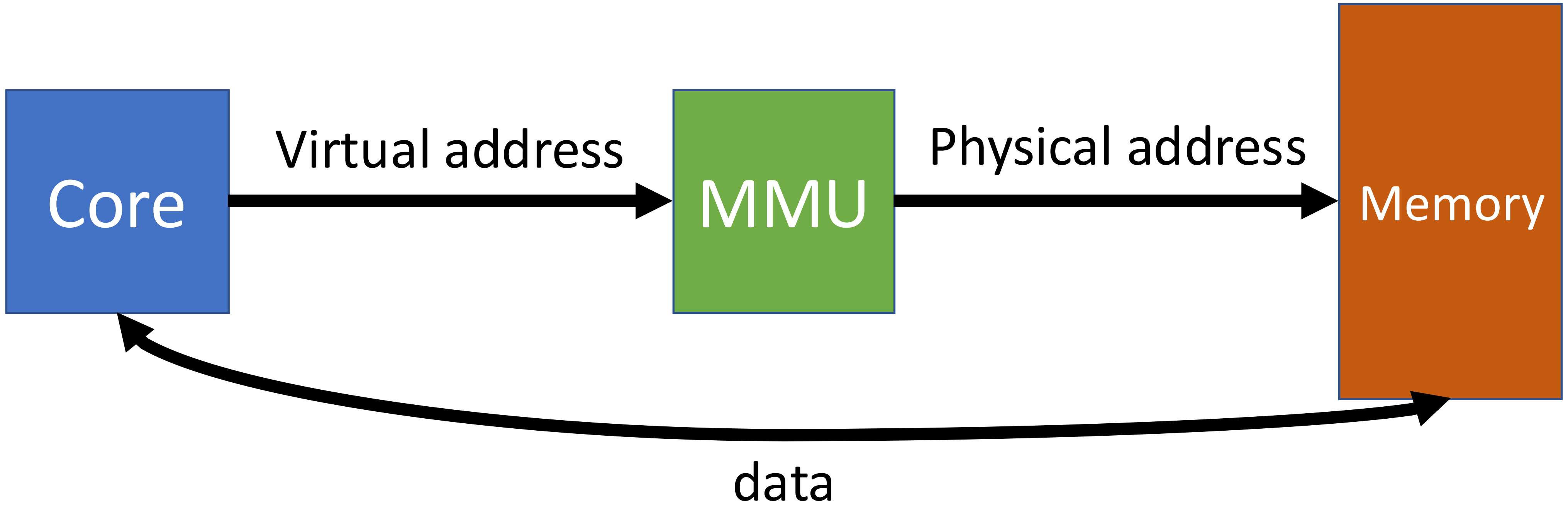
Idea: What if, instead of letting programs use the real physical addresses, we had them use "imaginary" addresses within their own private "virtual world", and have the OS translate virtual addresses to physical addresses on the fly?

- The OS can prohibit processes from accessing certain addresses (e.g. OS memory or another process's memory)
- Gives the OS lots of flexibility in managing memory
- Every process can now think that it is located starting at address 0 and is the only process in memory
- The OS will translate each process's address to the real one it's mapped to
- As a result, a process's virtual address space may look very different from how the memory is really laid out in the physical address space.

Dynamic Address Translation

We will add a *memory management unit* (MMU) in hardware that changes addresses dynamically during every memory reference.

- *Virtual address* is what the program sees
- *Physical address* is the actual location in memory



Dynamic Address Translation

Key question: how do the MMU / OS translate from virtual addresses to physical ones? Three designs we'll consider:

1. **Base and bound**
2. **Multiple Segments**
3. **Paging**

Approach #1: Base and Bound

- "base" is physical address starting point – corresponds to virtual address 0
- "bound" is one greater than highest allowable **virtual** memory address
- Each process has own base/bound. Stored in PCB and loaded into two registers when running.

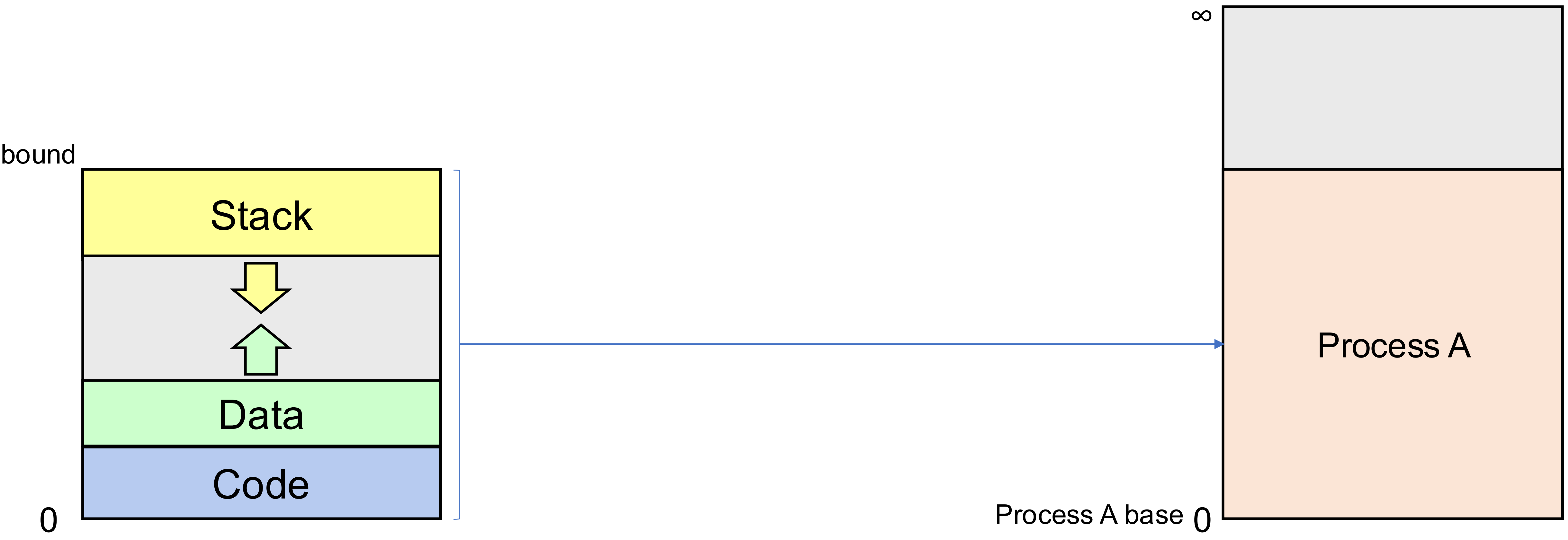
On each memory reference:

- Compare virtual address to bound, trap if >= (invalid memory reference)
- Otherwise, add base to virtual address to produce physical address

Approach #1: Base and Bound

- Key idea: each process appears to have a completely private memory whose size is determined by the bound register.
- The only physical address is in the base register, controlled by the OS. Process sees only virtual addresses!
- OS can update a process's base/bound if needed! E.g. it could move physical memory to a new location or increase bound.
- **Benefits**: inexpensive, little space needed, separation between virtual and physical addresses.
- **Drawbacks**: physical space must be contiguous, fragmentation, growth only upwards, no read-only region support

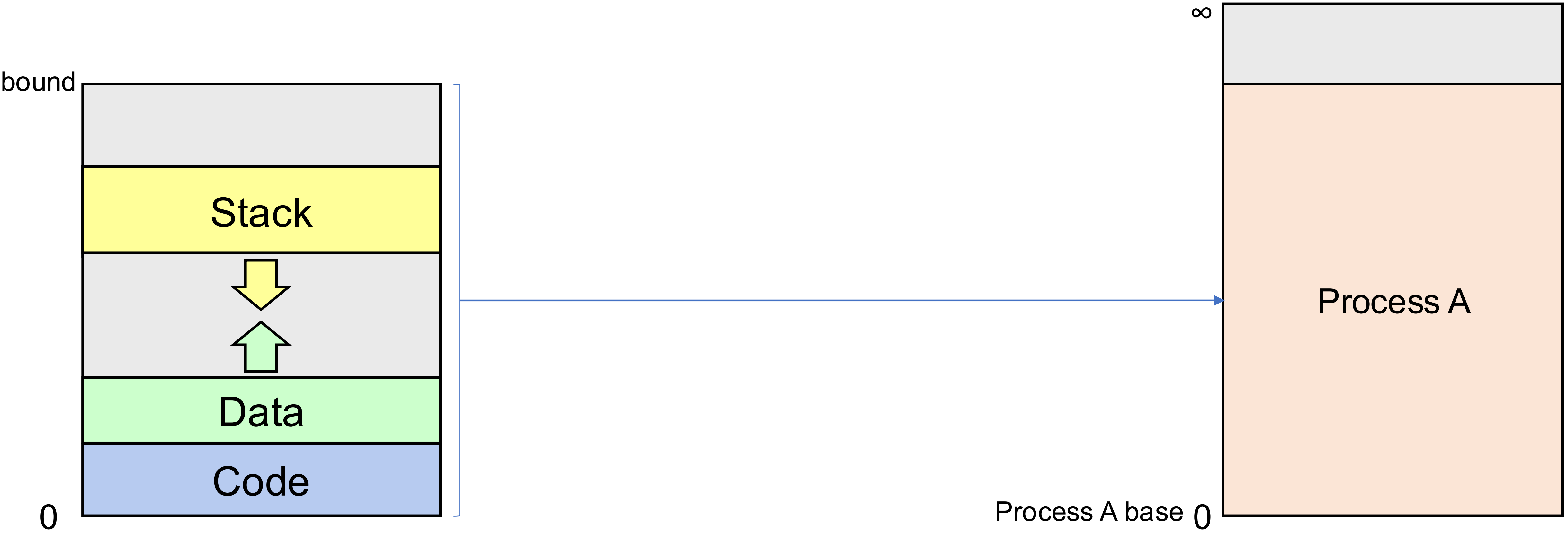
Base and Bound – Changing Bound



Process A Virtual Address Space

Physical Address Space

Base and Bound – Changing Bound

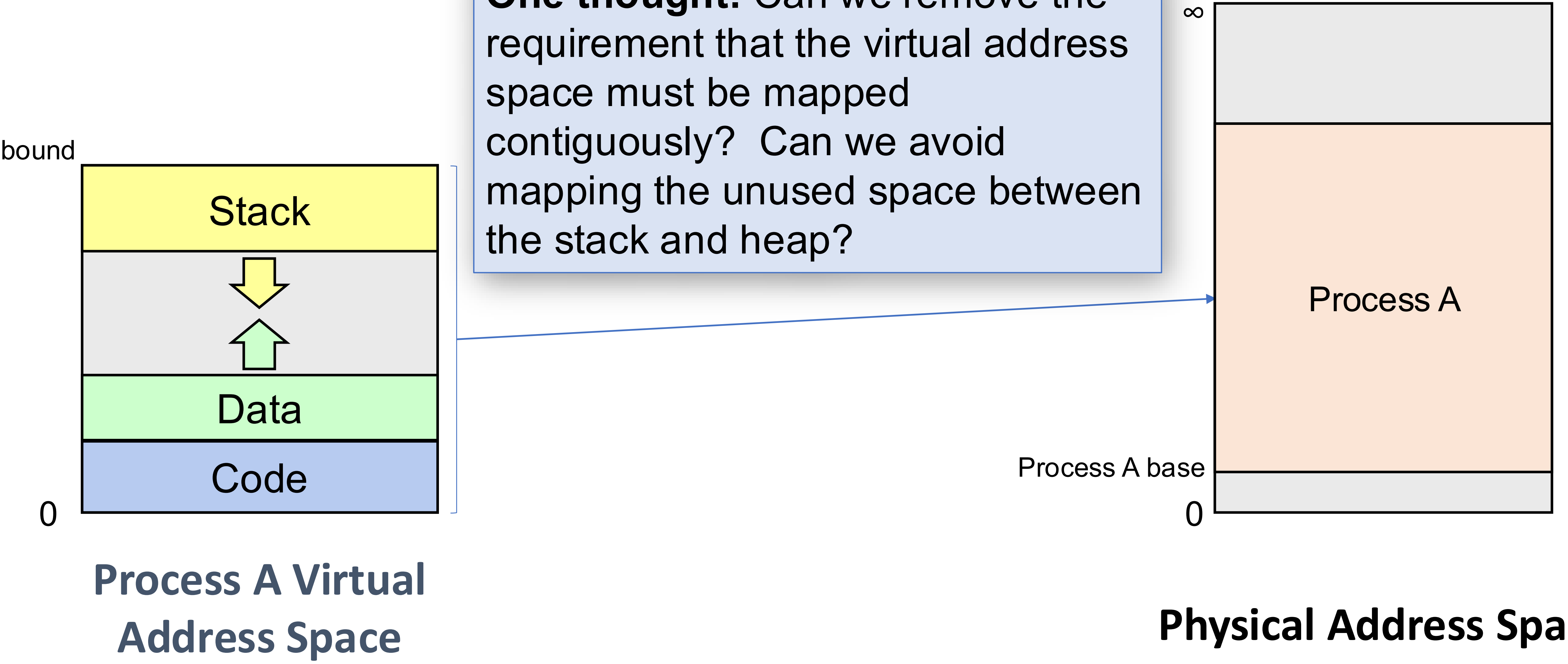


Process A Virtual Address Space

Physical Address Space

Base and Bound

One thought: Can we remove the requirement that the virtual address space must be mapped contiguously? Can we avoid mapping the unused space between the stack and heap?



Plan For Today

- **Recap:** virtual memory and dynamic address translation
- **Approach #2: Multiple Segments**
- Approach #3: Paging

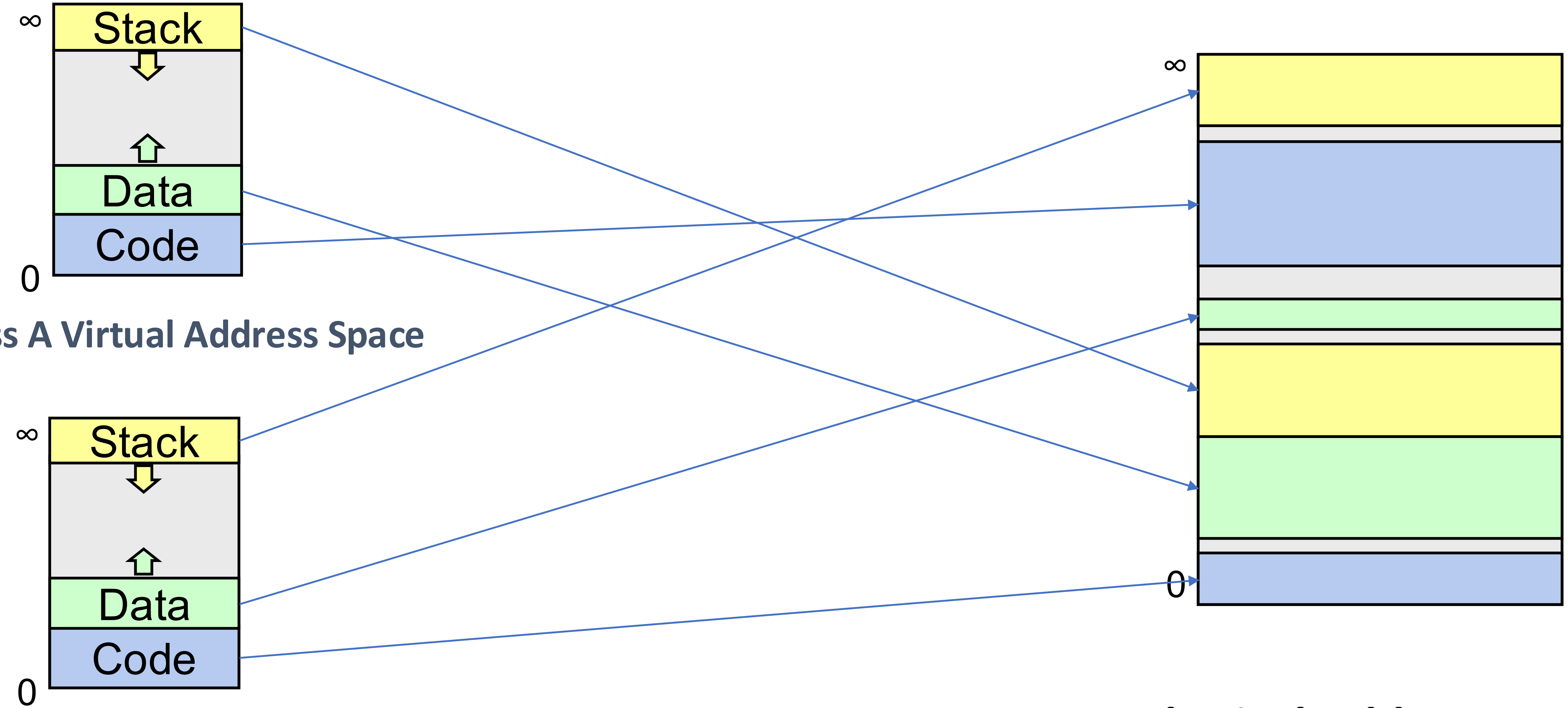
Idea: what if we broke up the virtual address space into segments and mapped each segment independently?

Approach #2: Multiple Segments

Key Idea: Each process is split among several variable-size areas of memory, called segments.

- E.g. one segment for code, one segment for data/heap, one segment for stack.
- The OS maps each segment individually – each segment would have its own base and bound, and these are stored in a *segment map* for that process
- Start of each segment is fixed in the virtual address space
- We can also store a *protection* bit for each segment; whether the process is allowed to write to it or not in addition to reading
- Now each segment can have its own permissions, grow/shrink independently, be swapped to disk independently, be moved independently, and even be shared between processes (e.g. shared code).

Multiple Segments



Process A Virtual Address Space

Process B Virtual Address Space

Physical Address Space

Approach #2: Multiple Segments

On each memory reference:

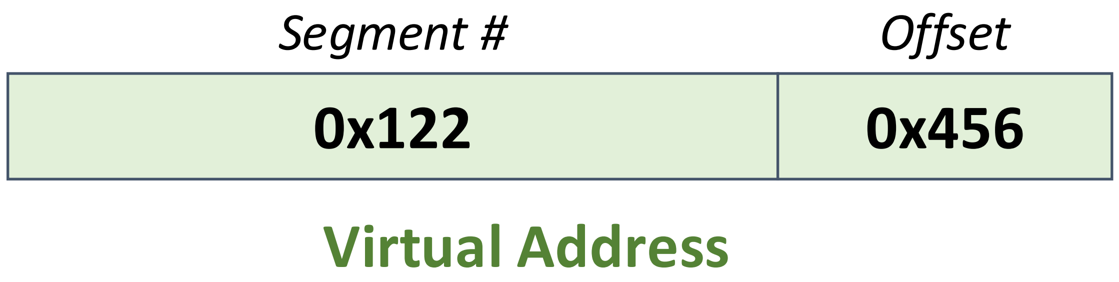
- Look up info for the segment that address is in
- Compare virtual address to that segment’s bound, trap if >= (invalid memory reference)
- Add segment’s base to virtual address to produce physical address

Problem: how do we know which segment a virtual address is in?

Approach #2: Multiple Segments

Problem: how do we know which segment a virtual address is in?

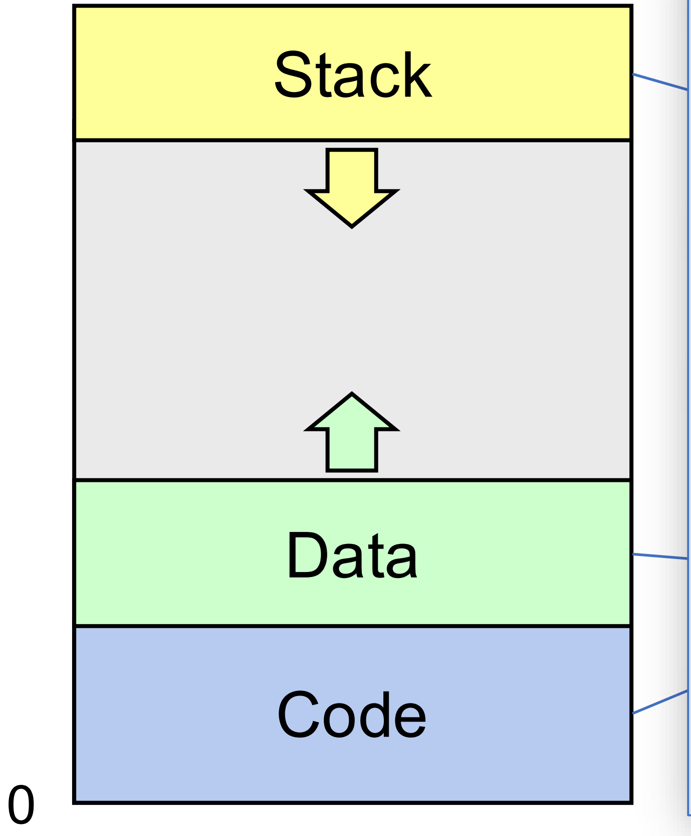
One Idea: make virtual addresses such that the top bits of the address specify its segment, and the low bits of the address specify the offset in that segment.



Example: PDP-10 computer had design with 2 segments, and the most-significant bit in addresses encoded which one was being referenced.

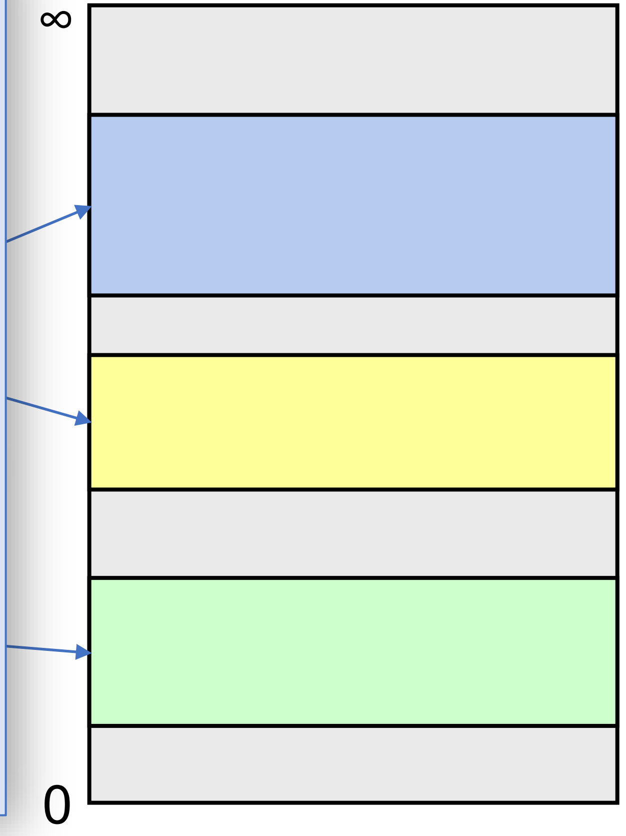
Another possibility: deduce from machine code instruction executing

Multiple Segments



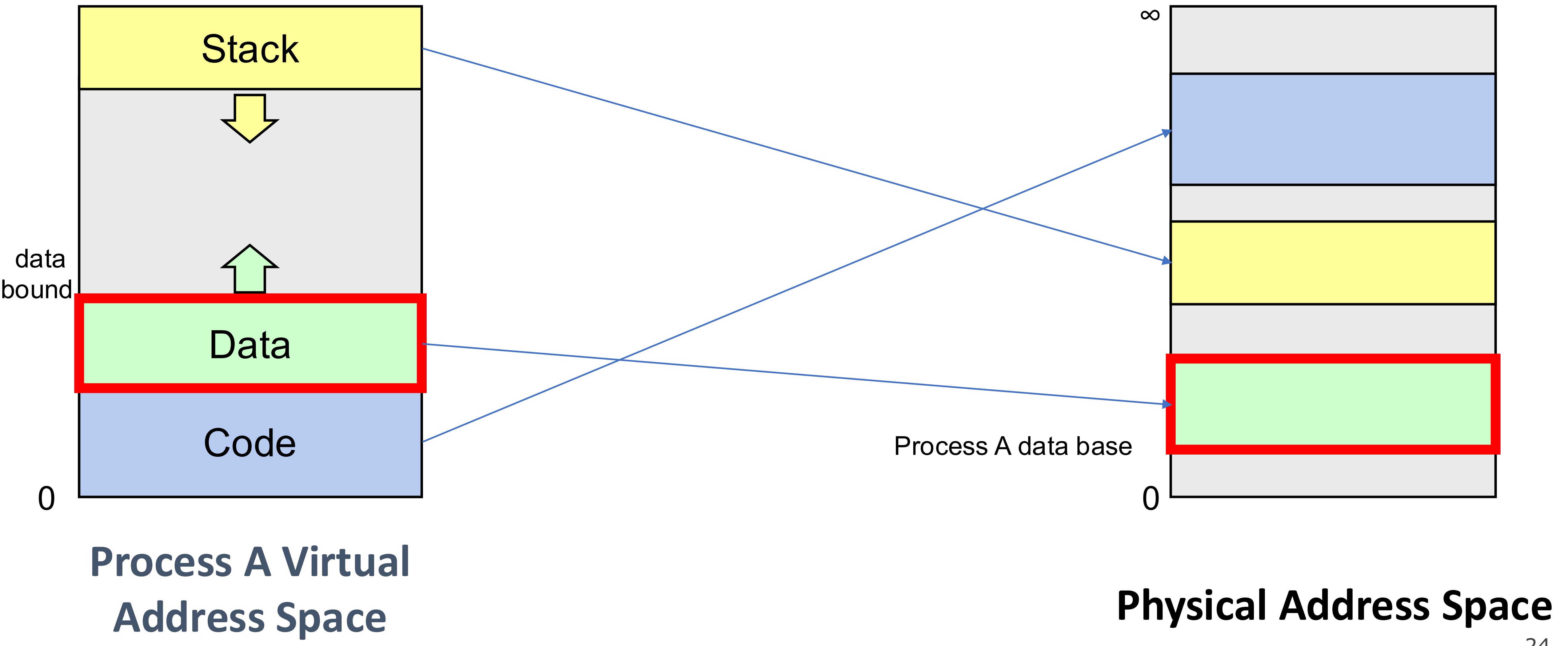
Process A Virtual Address Space

- Do not need to initially map full virtual address space, nor map it contiguously.
- Instead, individually/contiguously map each segment.
- Move an individual segment in physical memory by modifying its base (pinned to that segment's offset 0)
- Expand an individual segment's size by adjusting its bound.

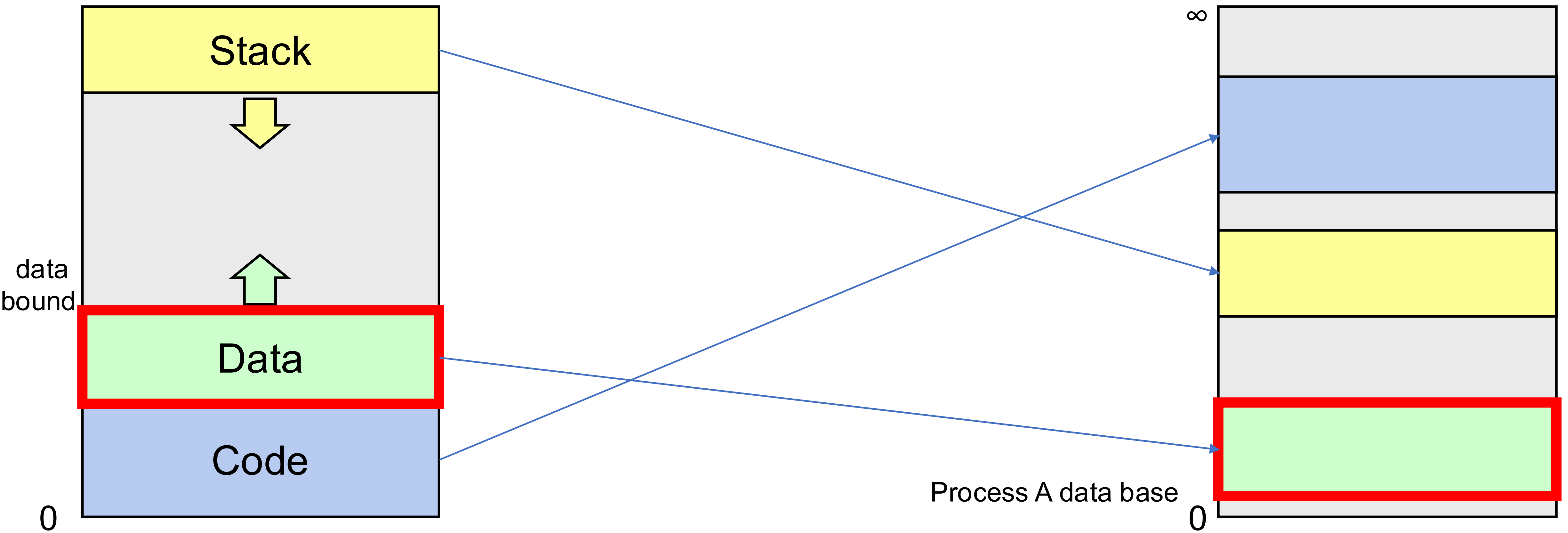


Physical Address Space

Multiple Segments – Changing A Base



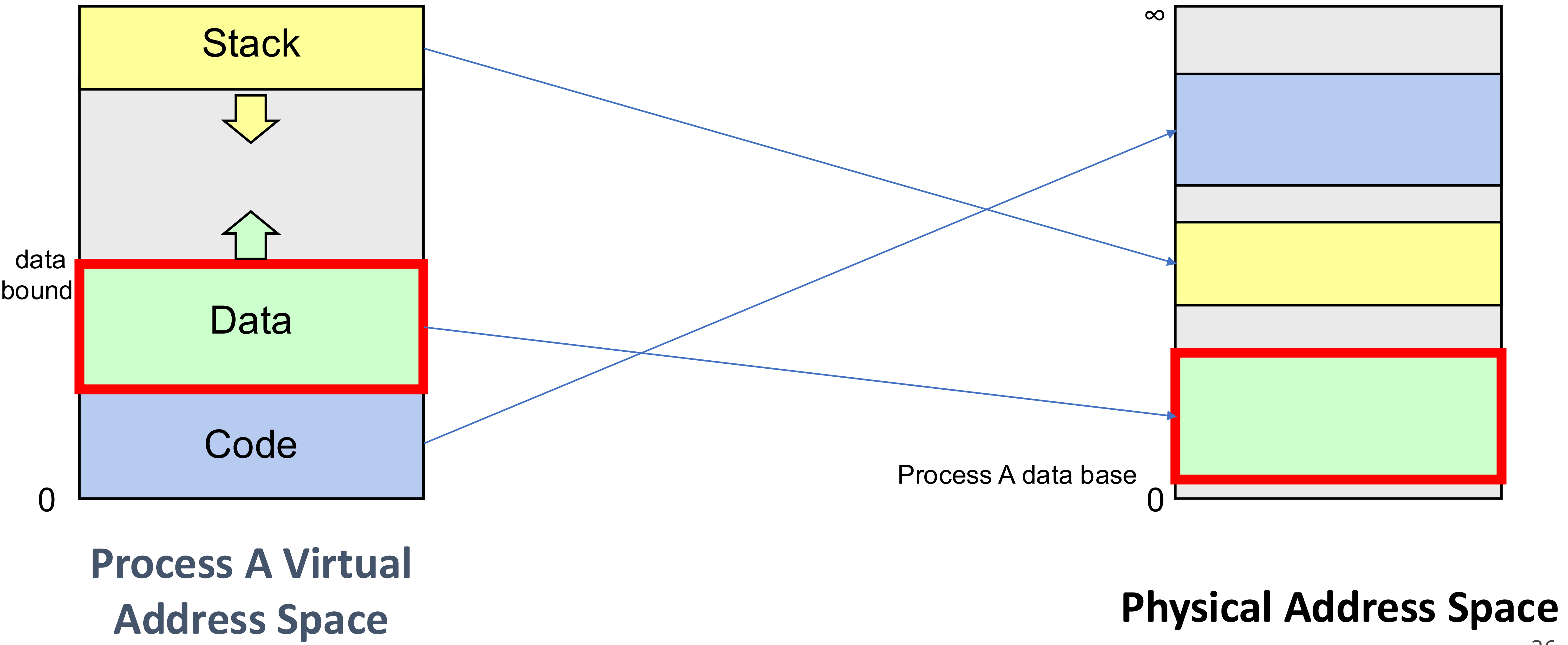
Multiple Segments – Changing A Base



Process A Virtual Address Space

Physical Address Space

Multiple Segments – Changing A Bound



Approach #2: Multiple Segments

What are some benefits of this approach?

- Can move segments to compact memory and eliminate fragmentation
- Flexibility – can manage each segment independently
- Can share segments between processes

What are some drawbacks of this approach?

- Variable-length segments result in memory fragmentation – can move, but creates friction
- Typically small number of segments
- Encoding segment + offset rigidly divides virtual addresses (how many bits for segment vs. how many for offset?)
- Can only grow segments upwards with the bound

Approach #2: Multiple Segments

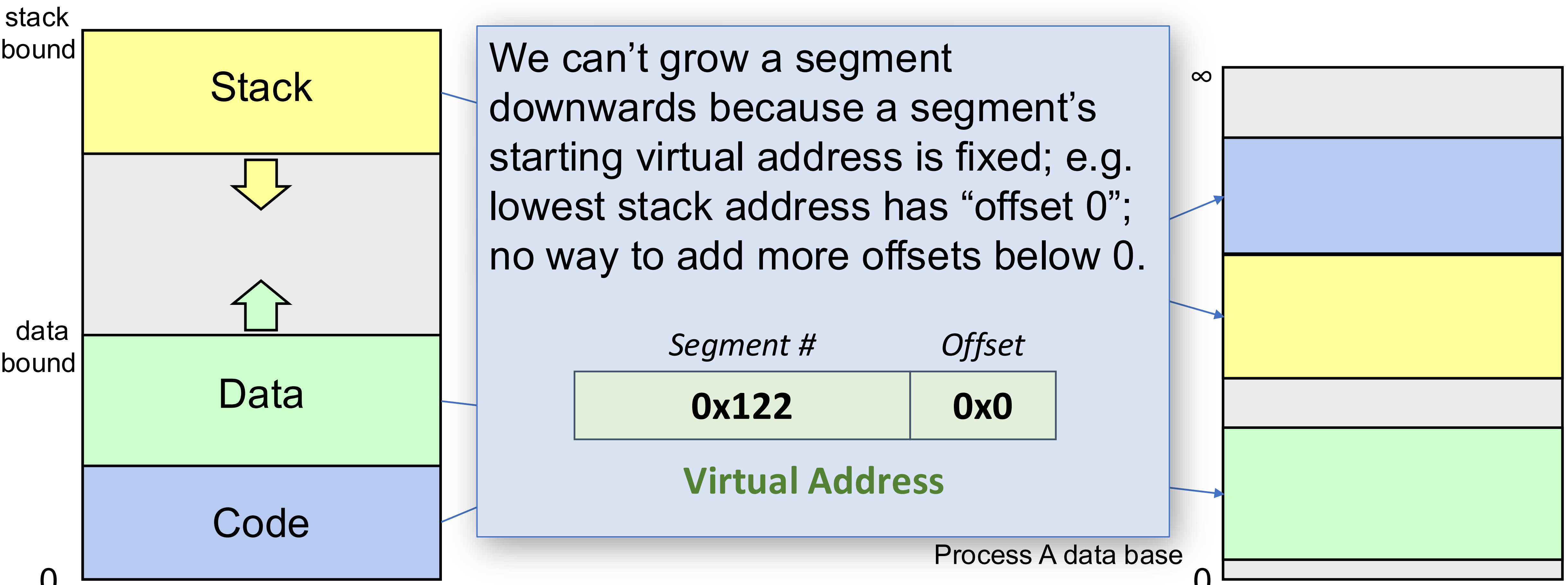
What are some benefits of this approach?

- Can move segments to compact memory and eliminate fragmentation
- Flexibility – can manage each segment independently
- Can share segments between processes

What are some drawbacks of this approach?

- Variable-length segments result in memory fragmentation – can move, but creates friction
- Typically small number of segments
- Encoding segment + offset rigidly divides virtual addresses (how many bits for segment vs. how many for offset?)
- **Can only grow segments upwards with the bound**

Multiple Segments – Changing A Bound



We can't grow a segment downwards because a segment's starting virtual address is fixed; e.g. lowest stack address has "offset 0"; no way to add more offsets below 0.

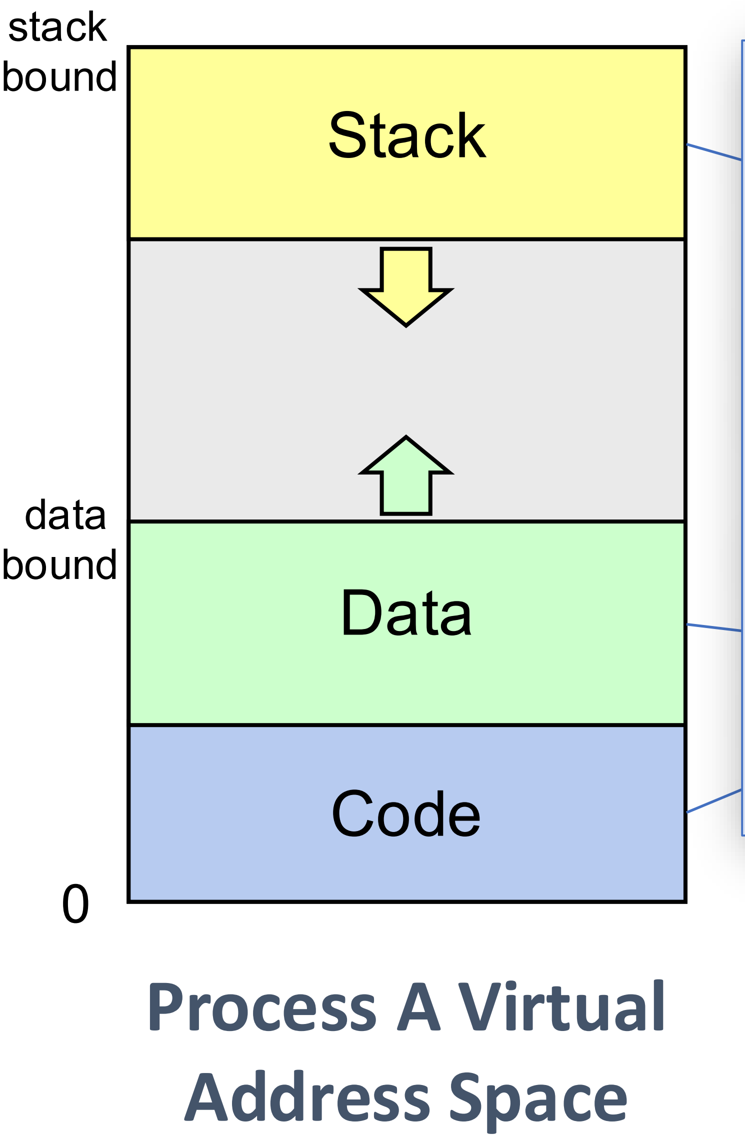
Segment #	*Offset*
0x122	**0x0**

Virtual Address

Process A Virtual Address Space

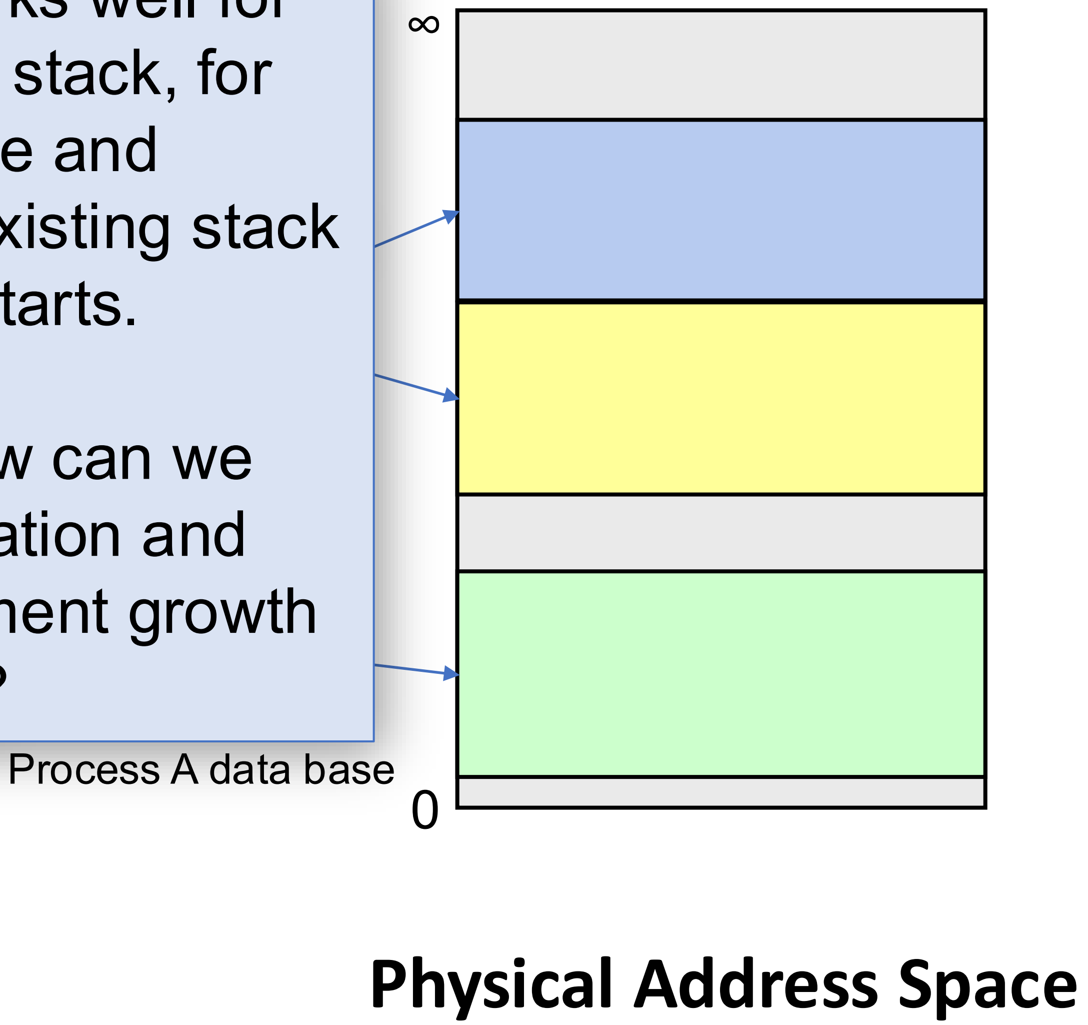
Physical Address Space

Multiple Segments – Changing A Bound



Upward-only growth works well for the heap, but not for the stack, for the same reason as base and bound: we can't move existing stack data after the program starts.

Can we do better? How can we avoid external fragmentation and also easily support segment growth upwards *or* downwards?



Idea: what if we broke up the virtual address space not into variable-length segments, but into fixed-size chunks?

Plan For Today

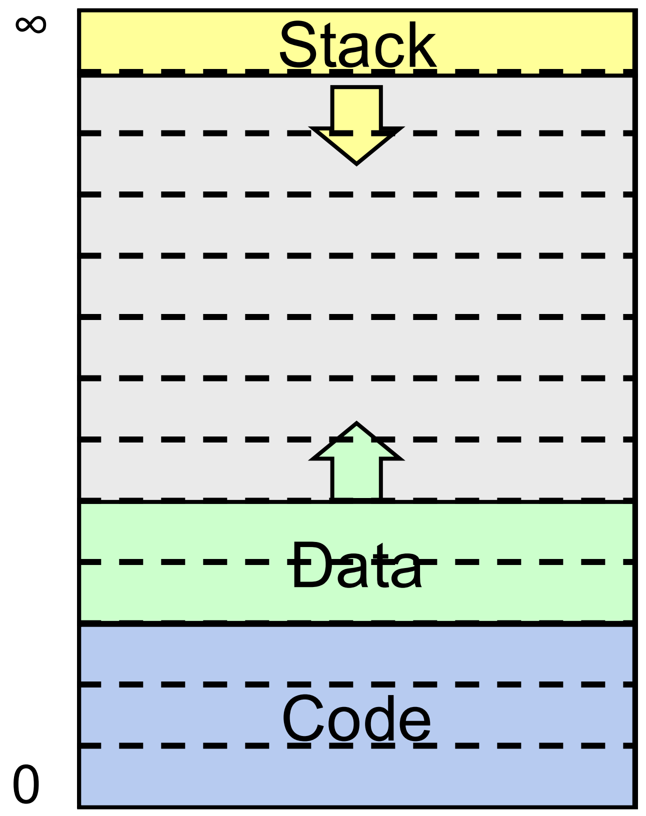
- **Recap:** virtual memory and dynamic address translation
- Approach #2: Multiple Segments
- **Approach #3: Paging**

Approach #3: Paging

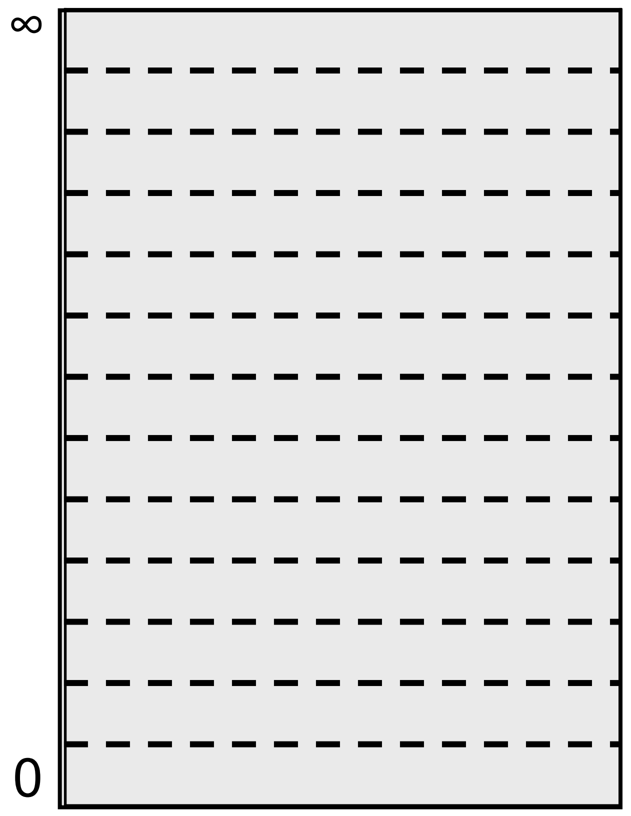
Key Idea: Each process's virtual (and physical) memory is divided into fixed-size chunks called *pages*. (Common size is 4KB pages).

- A "page" of virtual memory maps to a "page" of physical memory. No partial pages

Paging

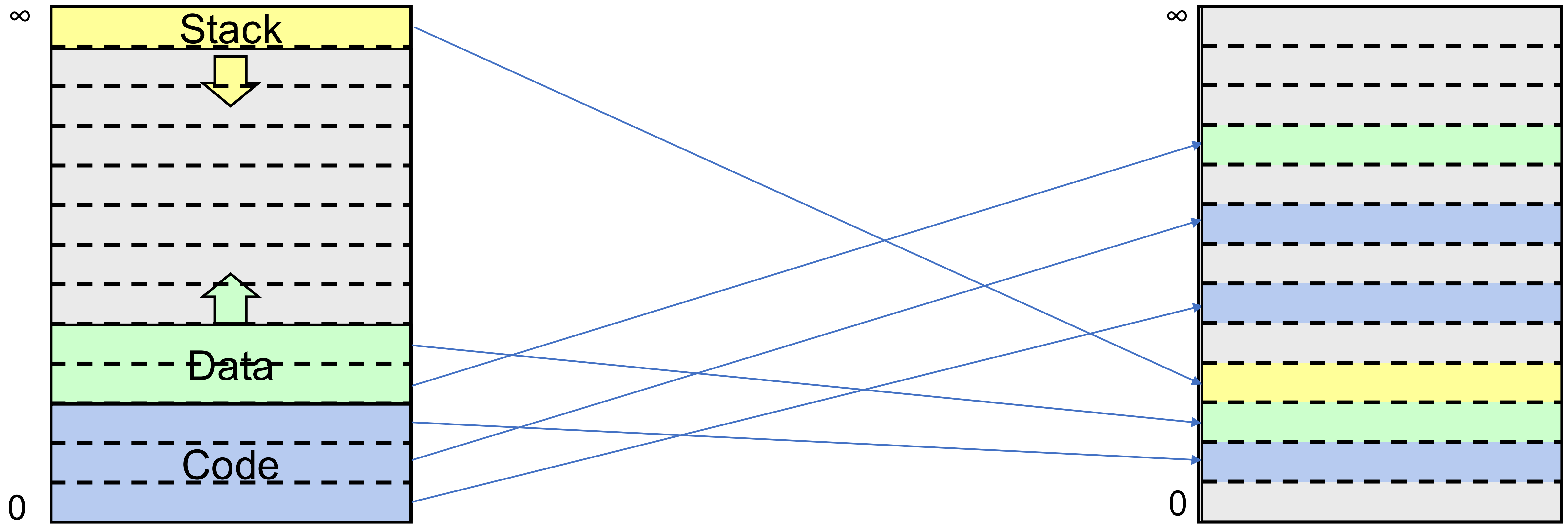


Process A Virtual Address Space



Physical Address Space

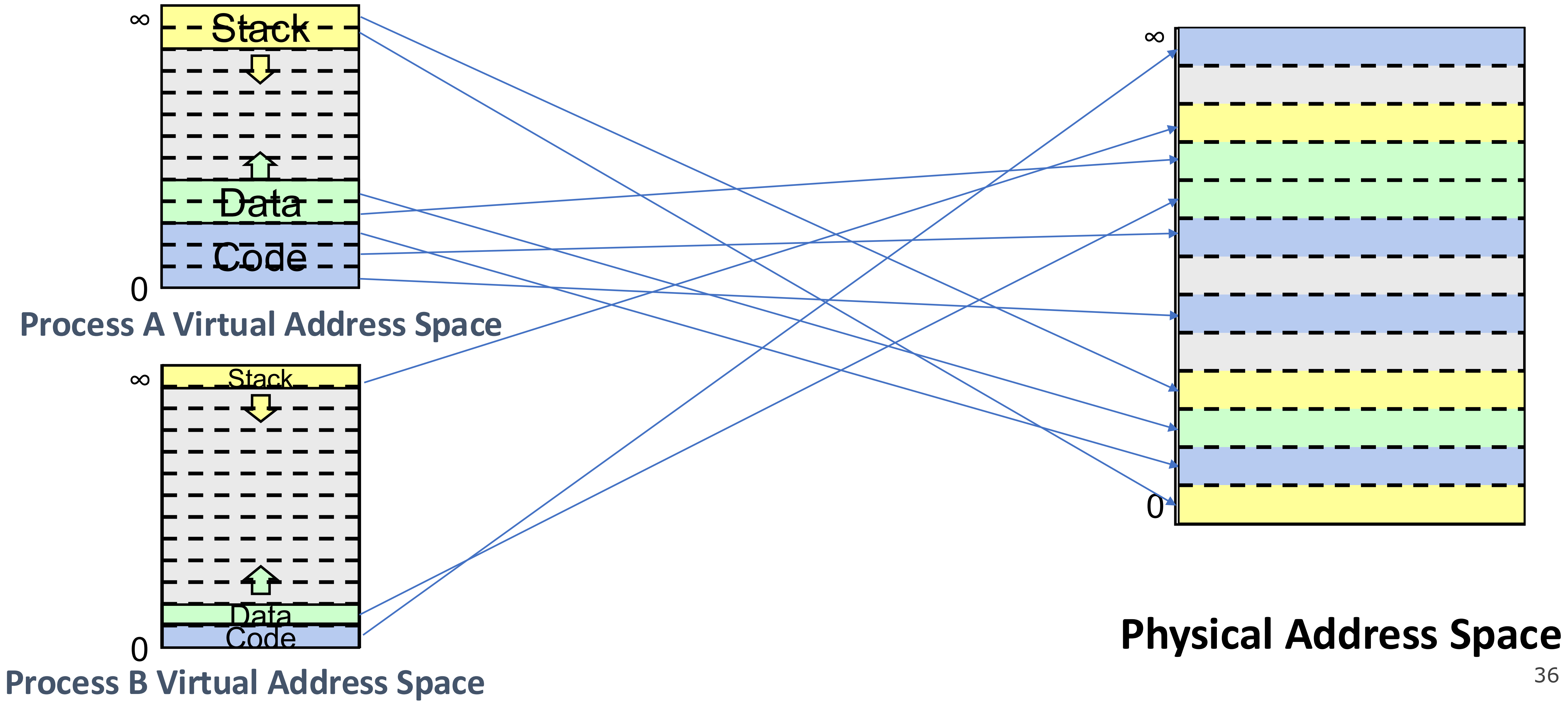
Paging



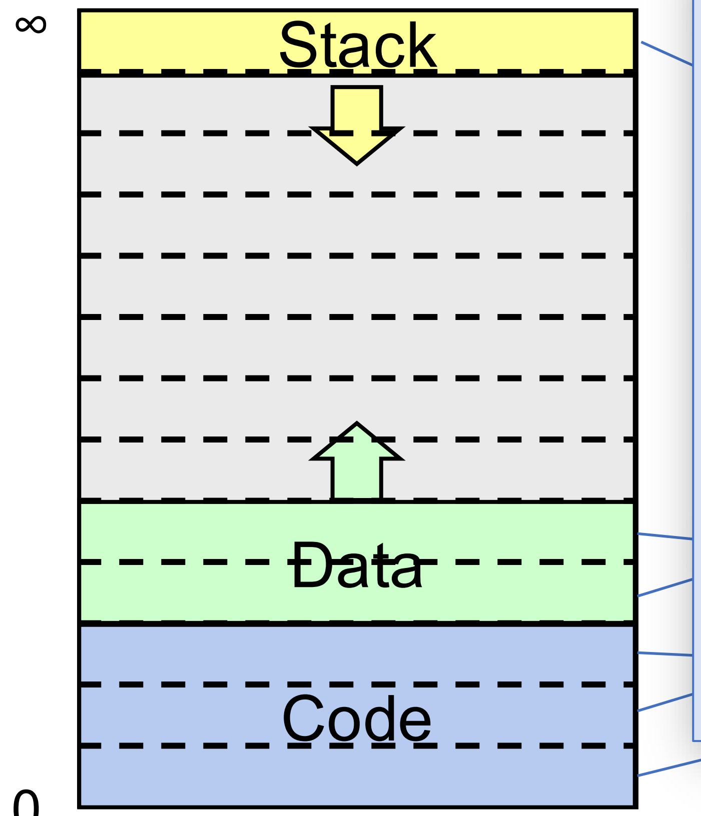
Process A Virtual Address Space

Physical Address Space

Paging

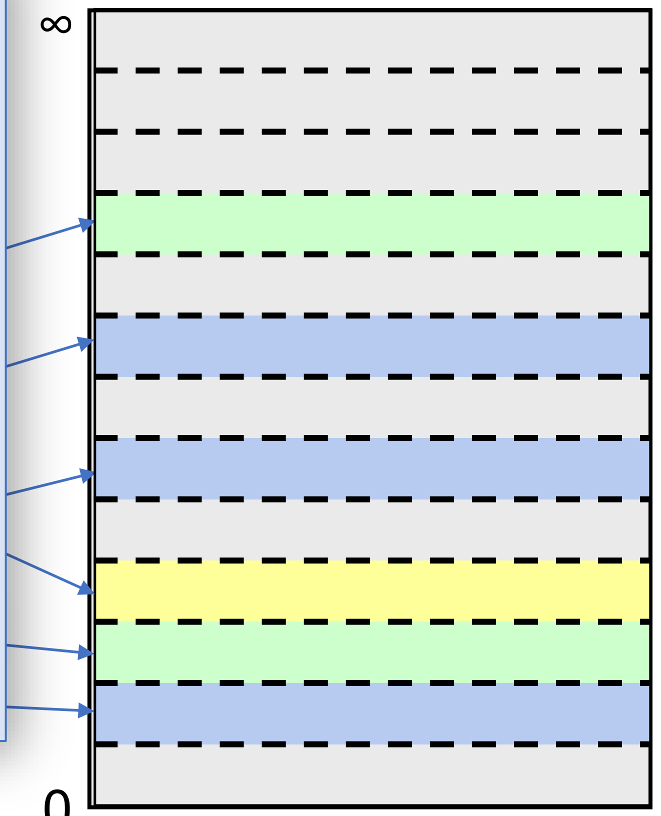

∞
Stack
Data
Code
0
Process A Virtual Address Space
∞
Stack
Data
Code
0
Process B Virtual Address Space
∞
0
Physical Address Space

Paging



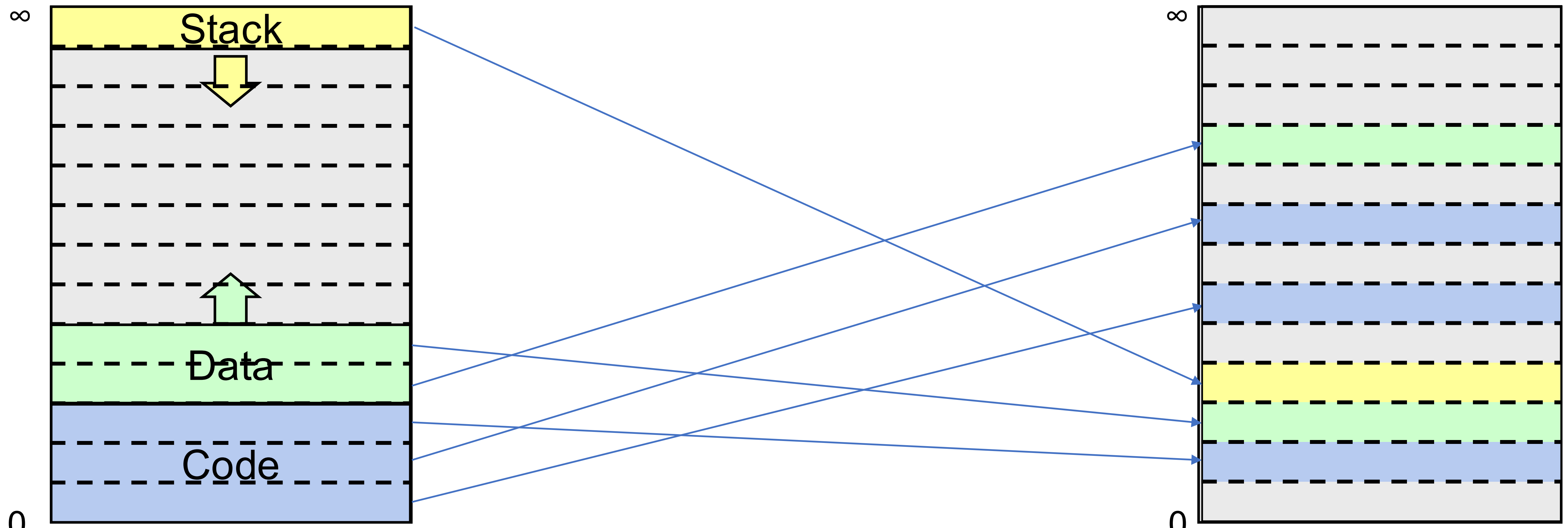
Process A Virtual Address Space

- Do not need to map each segment contiguously. Instead, we map just one page at a time.
- We can later map more pages either up or down, **because the start of the segment is not pinned to a specific address**.
- We can move each page separately in physical memory as well.



Physical Address Space

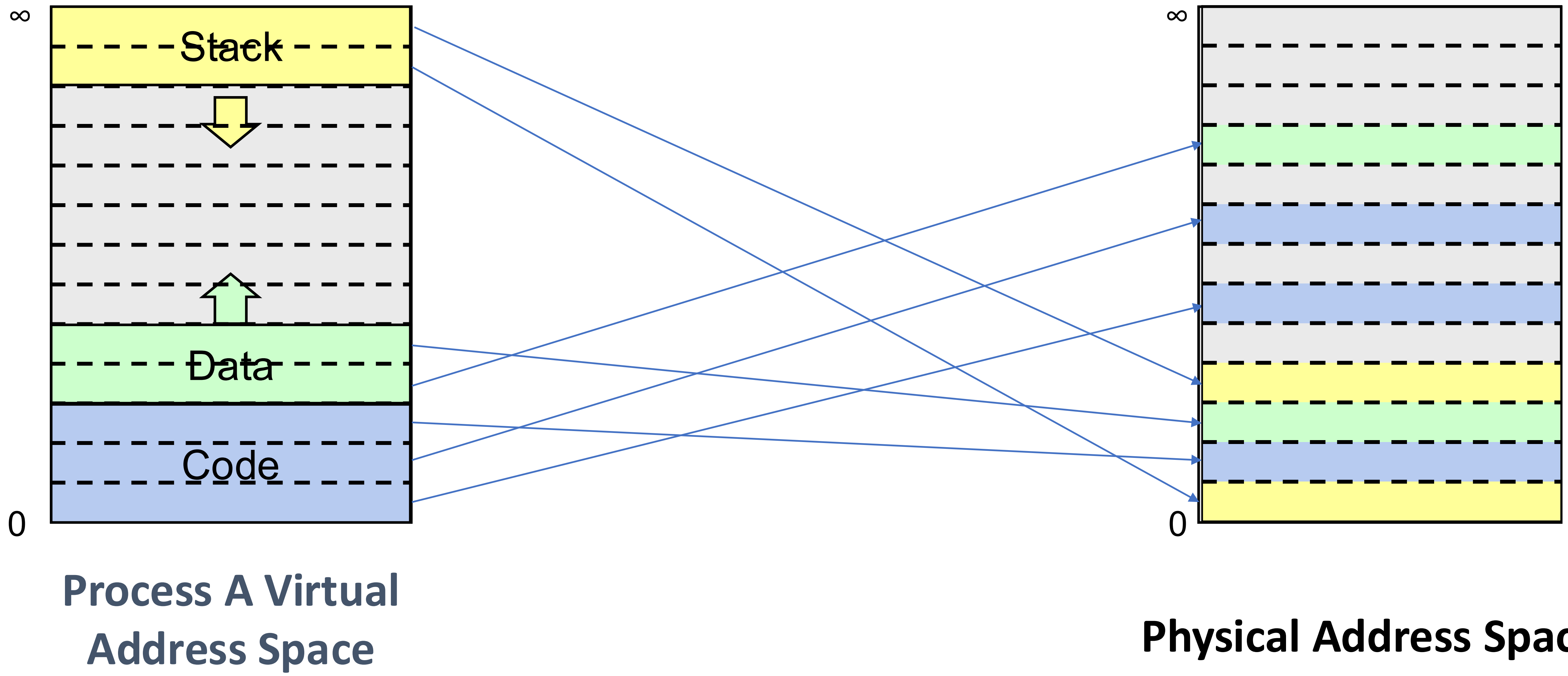
Paging



Process A Virtual Address Space

Physical Address Space

Paging


∞
Stack
Data
Code
0
Process A Virtual Address Space
∞
0
Physical Address Space

Approach #3: Paging

Key Idea: Each process's virtual (and physical) memory is divided into fixed-size chunks called *pages*. (Common size is 4KB pages).

- A "page" of virtual memory maps to a "page" of physical memory. No partial pages
- The **page number** is a numerical ID for a page. We have virtual page numbers and physical page numbers.
- A virtual address is comprised of the virtual page # and offset in that page.
- A physical address is comprised of the physical page # and offset in that page.

Page Maps

How do we track, for a given process, which virtual page maps to which physical page?

Each process has a *page map ("page table")* with an entry for each virtual page, mapping it to a physical page number and other info such as a protection bit (read-only or read-write).

The page map is stored in contiguous memory.

Page Map

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0

Virtual page # = index

Key Idea: to enable quick lookup, the page table is an *array* where the indexes are the virtual page numbers. E.g. index 5 corresponds to virtual page 5.

<u>This means we must have an entry for every virtual page, even invalid ones</u> – because missing entries would mean indexes wouldn't line up anymore.

Page Map

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0

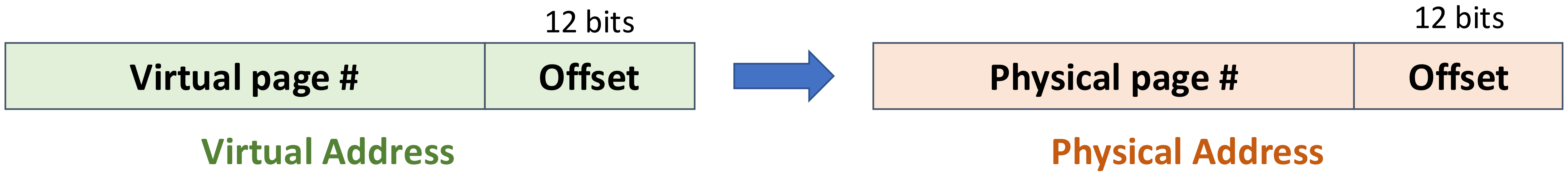
Virtual page # = index

Key Idea: to enable quick lookup, the page table is an *array* where the indexes are the virtual page numbers. E.g. index 5 corresponds to virtual page 5.

(Why not e.g. a hashmap? Not feasible at the time with hardware, plus turns out array ends up working pretty well with optimizations)

Page Map

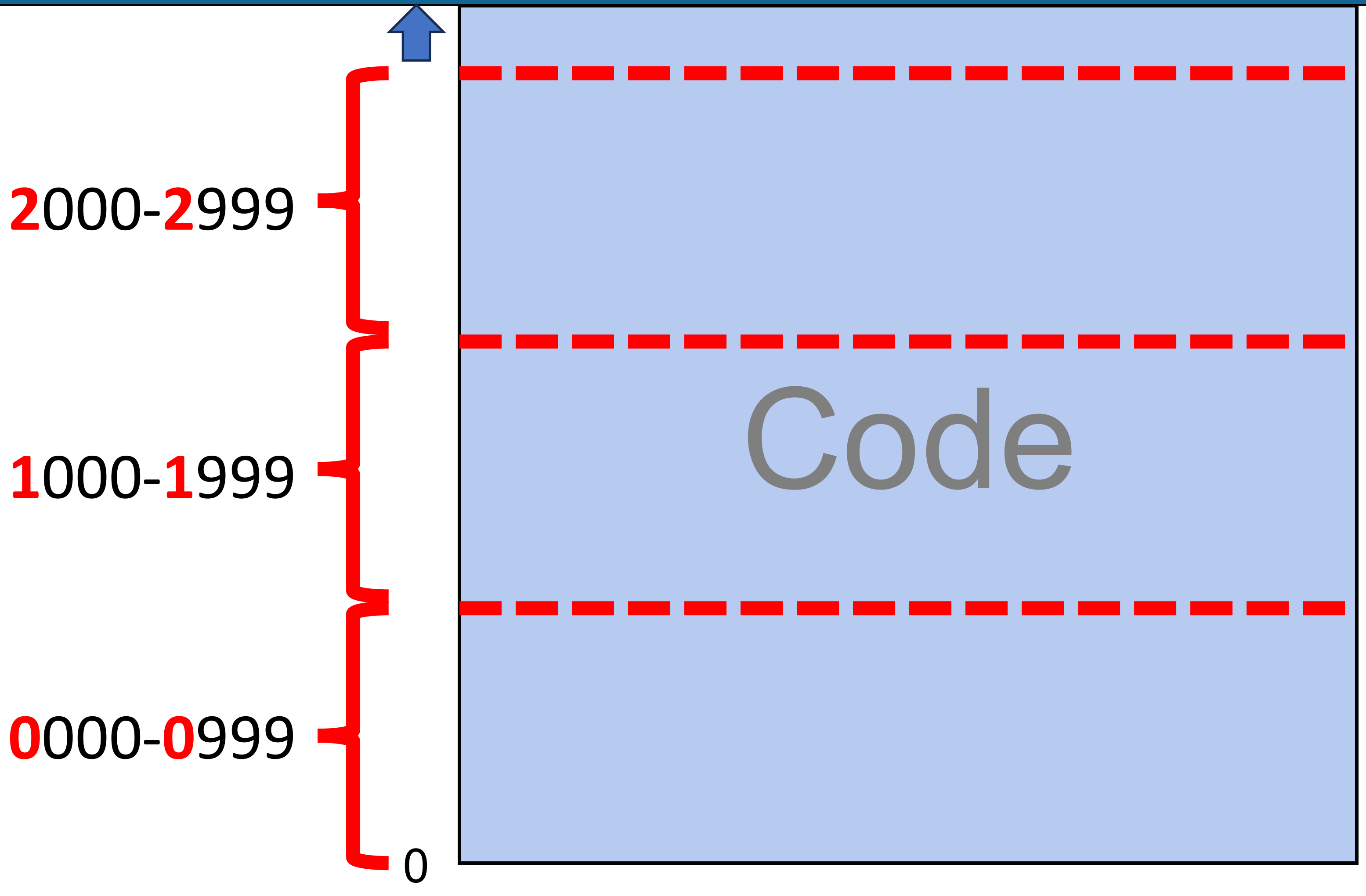
Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0



Virtual Address Encodes Page + Offset

Key idea: if you pick a page size that is a power of the base, the upper digits identify the page #.

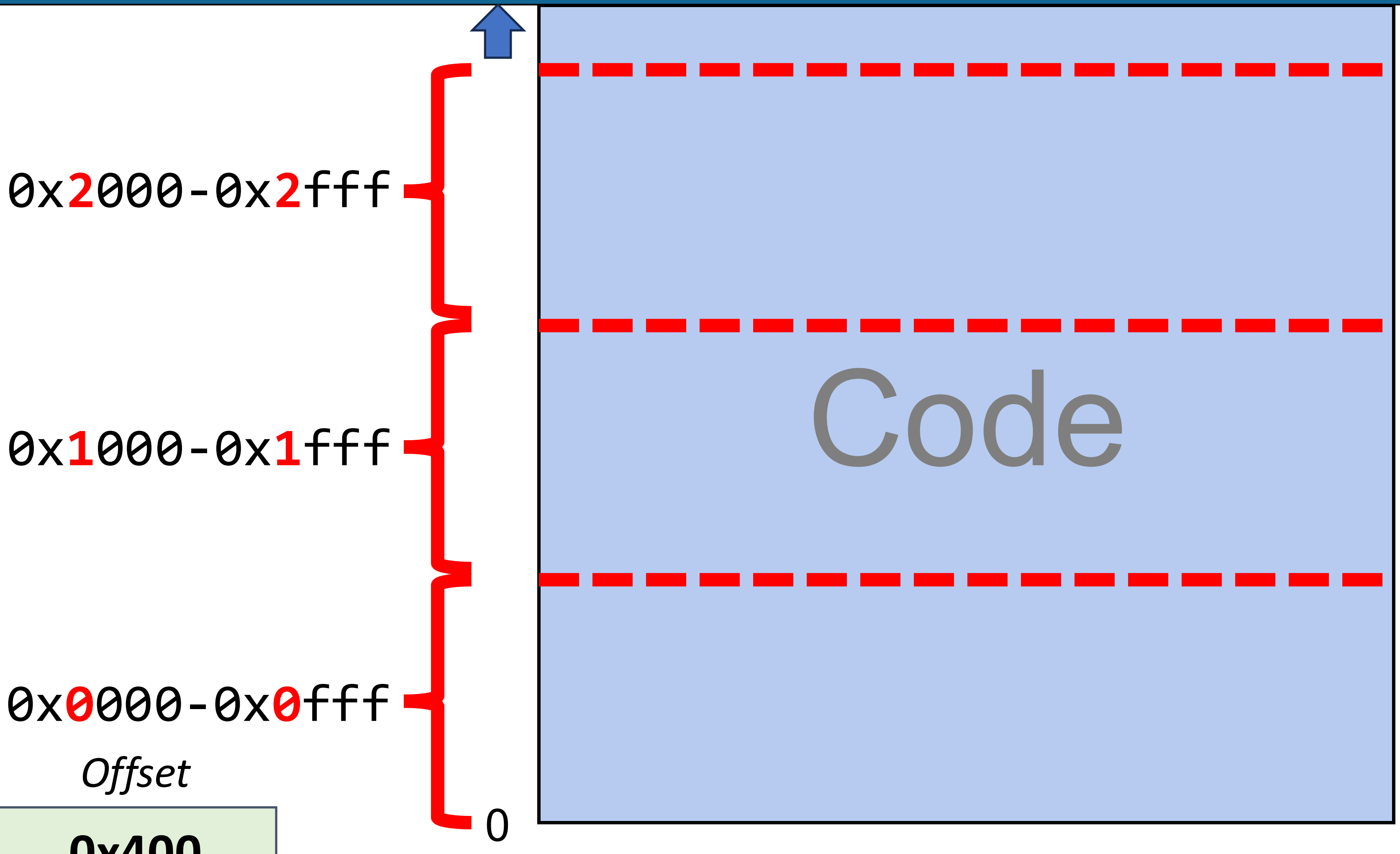
E.g. base 10, say page size = 10^3 = 1000:



Virtual Address Encodes Page + Offset

Key idea: if you pick a page size that is a power of the base, the upper digits identify the page #.

E.g. base 16, say page size = 16^3 = 4096:



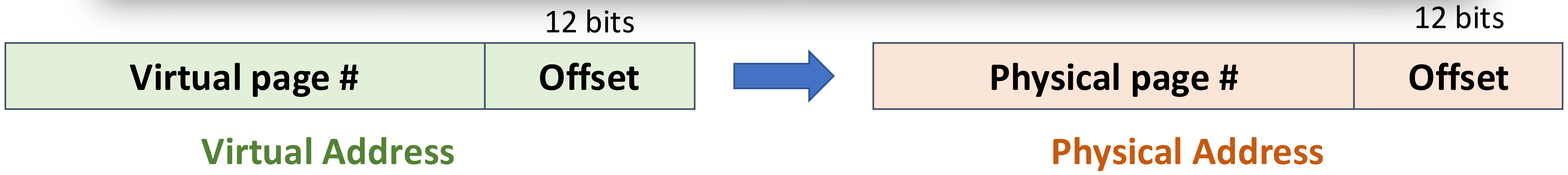
Virtual page #	*Offset*
0x323	**0x400**

Virtual Address `0x323400`

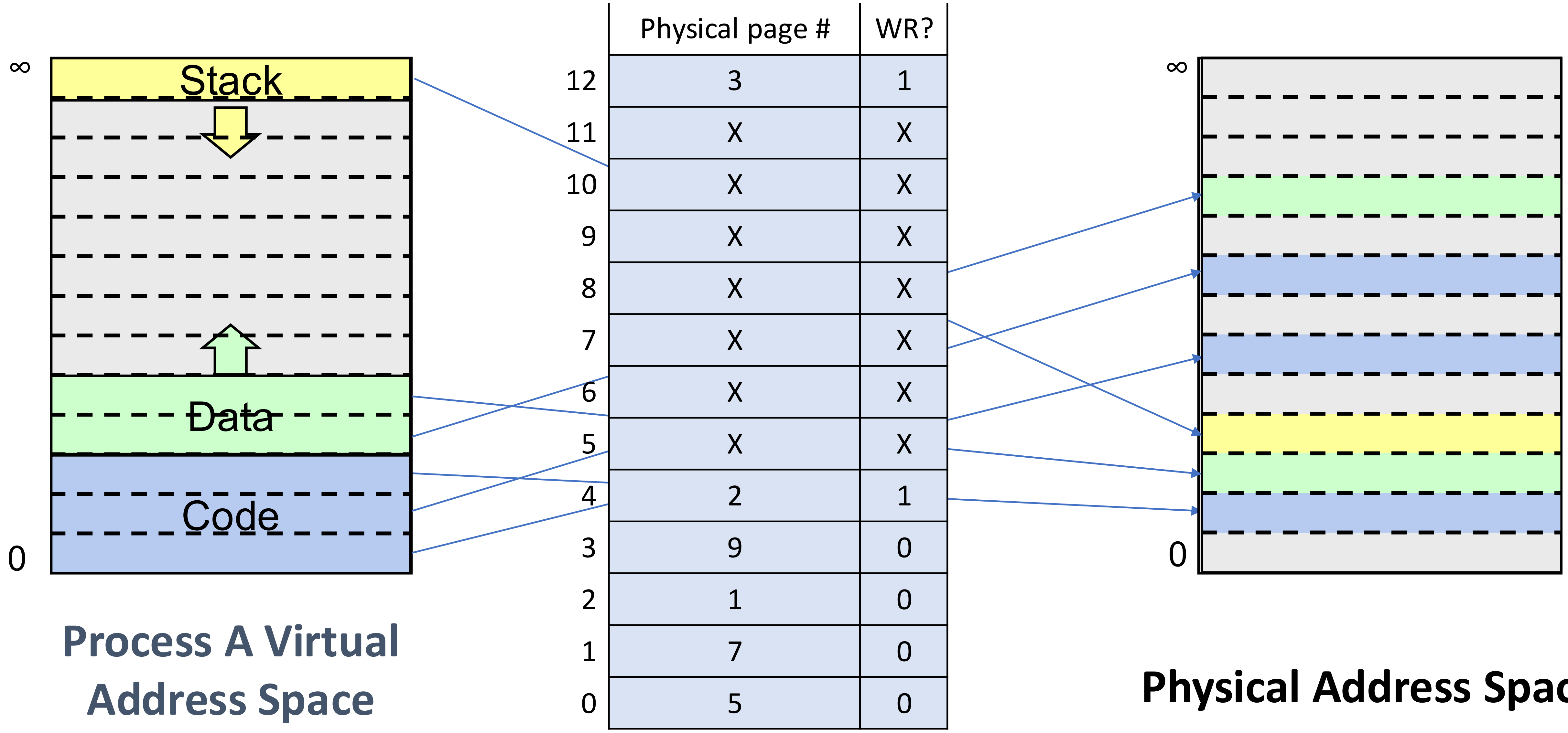
Page Map

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1

For 4KB pages (4096 bytes), the offset can be 0-4095. Thus, we can store the offset in 12 bits (the amount needed to represent any number 0-4095). 12 bits = 3 hexadecimal digits.



Page Map

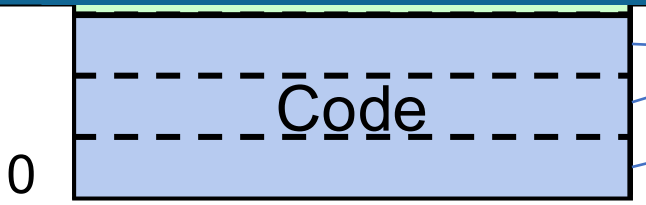


	Physical page #	WR?
12	3	1
11	X	X
10	X	X
9	X	X
8	X	X
7	X	X
6	X	X
5	X	X
4	2	1
3	9	0
2	1	0
1	7	0
0	5	0

Process A Virtual Address Space

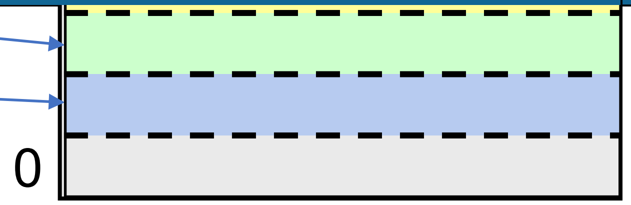
Physical Address Space

Page Map



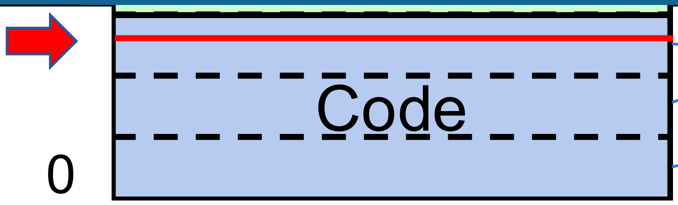
Process A Virtual Address Space

	Physical page #	WR?
12	3	1
11	X	X
10	X	X
9	X	X
8	X	X
7	X	X
6	X	X
5	X	X
4	2	1
3	9	0
2	1	0
1	7	0
0	5	0



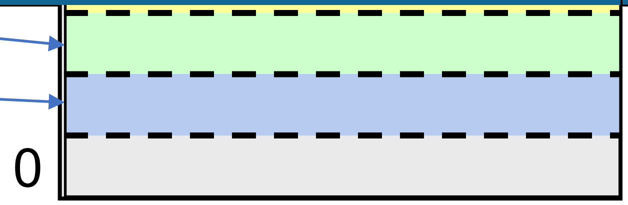
Physical Address Space

Page Map



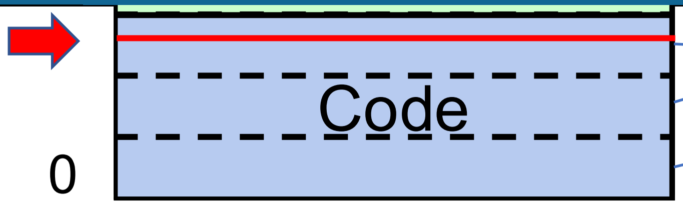
Process A Virtual Address Space

	Physical page #	WR?
12	3	1
11	X	X
10	X	X
9	X	X
8	X	X
7	X	X
6	X	X
5	X	X
4	2	1
3	9	0
2	1	0
1	7	0
0	5	0



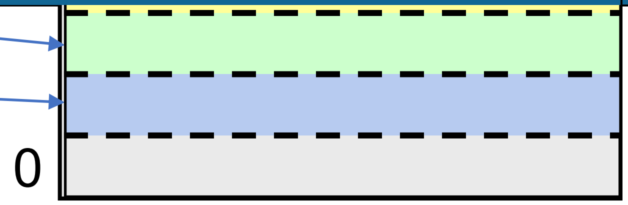
Physical Address Space

Page Map



Process A Virtual Address Space

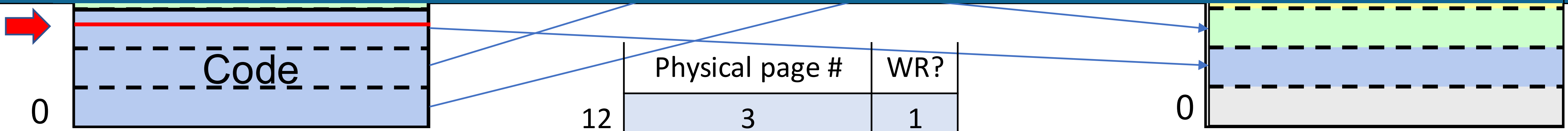
	Physical page #	WR?
12	3	1
11	X	X
10	X	X
9	X	X
8	X	X
7	X	X
6	X	X
5	X	X
4	2	1
3	9	0
2	1	0
1	7	0
0	5	0



Physical Address Space

Virtual Address
0x2223

Page Map



Process A Virtual Address Space

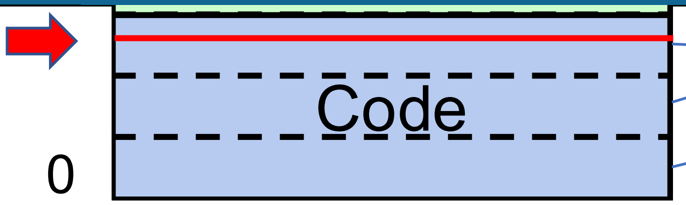
Physical Address Space

	Physical page #	WR?
12	3	1
11	X	X
10	X	X
9	X	X
8	X	X
7	X	X
6	X	X
5	X	X
4	2	1
3	9	0
2	1	0
1	7	0
0	5	0

Virtual page #	*Offset*
0x2	**0x223**

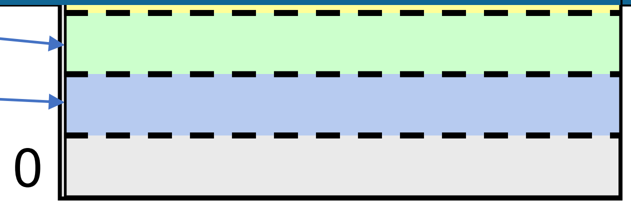
Virtual Address
`0x2223`

Page Map



Process A Virtual Address Space

	Physical page #	WR?
12	3	1
11	X	X
10	X	X
9	X	X
8	X	X
7	X	X
6	X	X
5	X	X
4	2	1
3	9	0
2	1	0
1	7	0
0	5	0



Physical Address Space

Virtual page #	*Offset*
0x2	**0x223**

Virtual Address

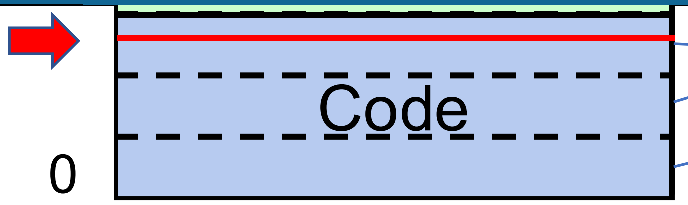
0x2223

Physical page #	*Offset*
???	**???**

Physical Address

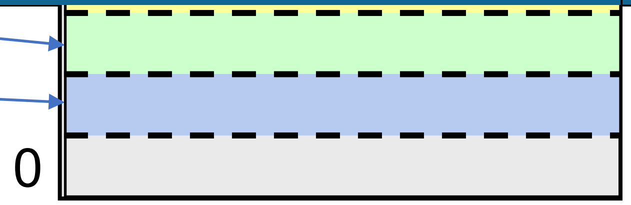
???

Page Map



Process A Virtual Address Space

	Physical page #	WR?
12	3	1
11	X	X
10	X	X
9	X	X
8	X	X
7	X	X
6	X	X
5	X	X
4	2	1
3	9	0
2	1	0
1	7	0
0	5	0



Physical Address Space

Virtual page #	*Offset*
0x2	**0x223**

Virtual Address

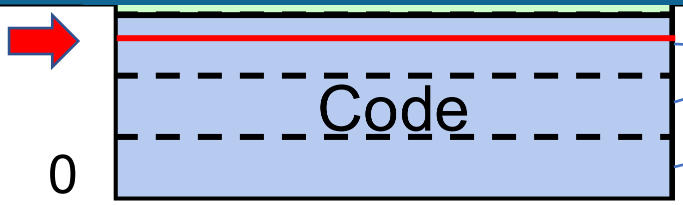
0x2223

Physical page #	*Offset*
0x1	**???**

Physical Address

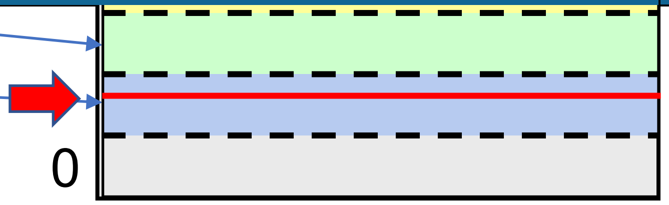
???

Page Map



Process A Virtual Address Space

	Physical page #	WR?
12	3	1
11	X	X
10	X	X
9	X	X
8	X	X
7	X	X
6	X	X
5	X	X
4	2	1
3	9	0
2	1	0
1	7	0
0	5	0



Physical Address Space

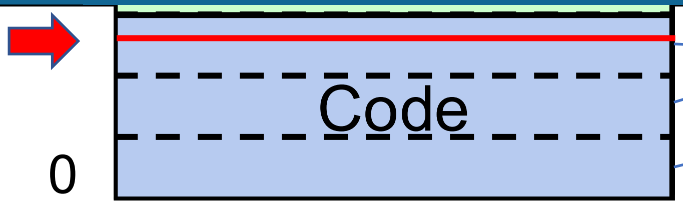
Virtual page #	*Offset*
0x2	**0x223**

Virtual Address
0x2223

Physical page #	*Offset*
0x1	**0x223**

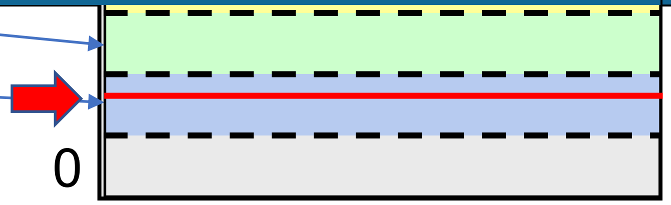
Physical Address
???

Page Map



Process A Virtual Address Space

	Physical page #	WR?
12	3	1
11	X	X
10	X	X
9	X	X
8	X	X
7	X	X
6	X	X
5	X	X
4	2	1
3	9	0
2	1	0
1	7	0
0	5	0



Physical Address Space

Virtual page #	*Offset*
0x2	**0x223**

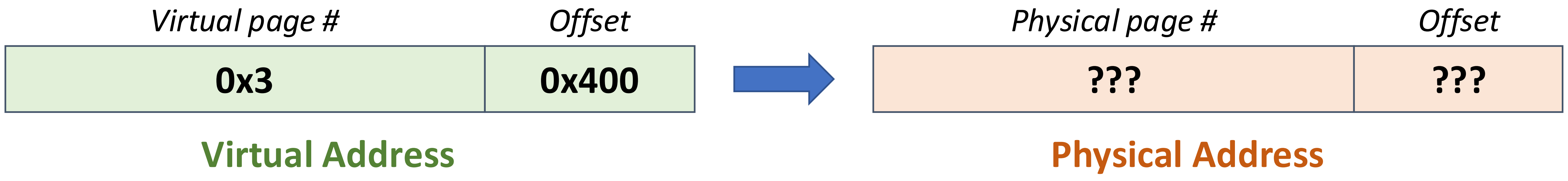
Virtual Address
`0x2223`

Physical page #	*Offset*
0x1	**0x223**

Physical Address
`0x1223`

Page Map

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0

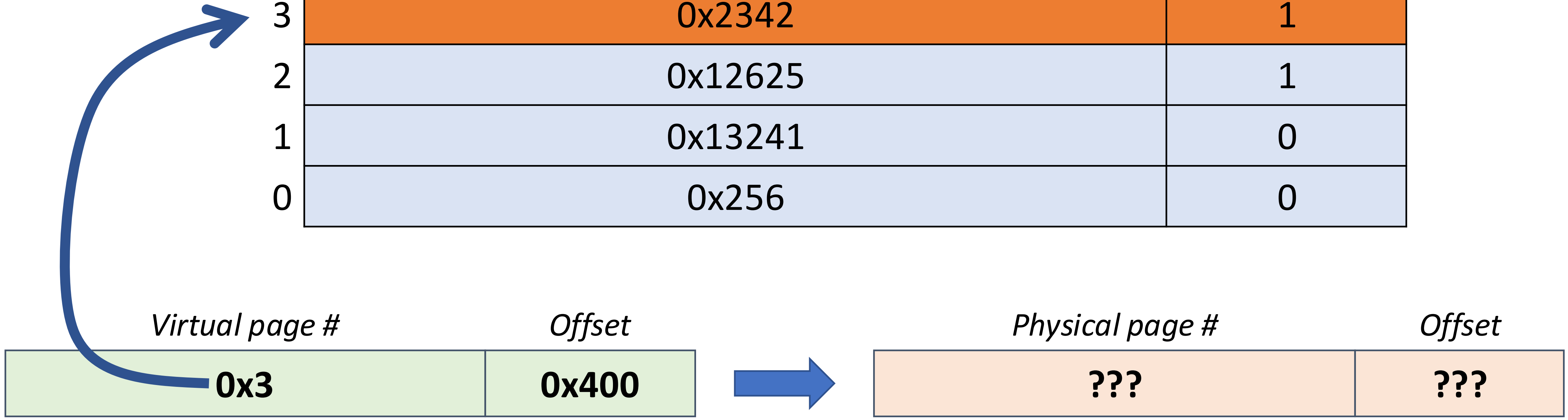


`0x3400`

???

Page Map

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0



Virtual Address

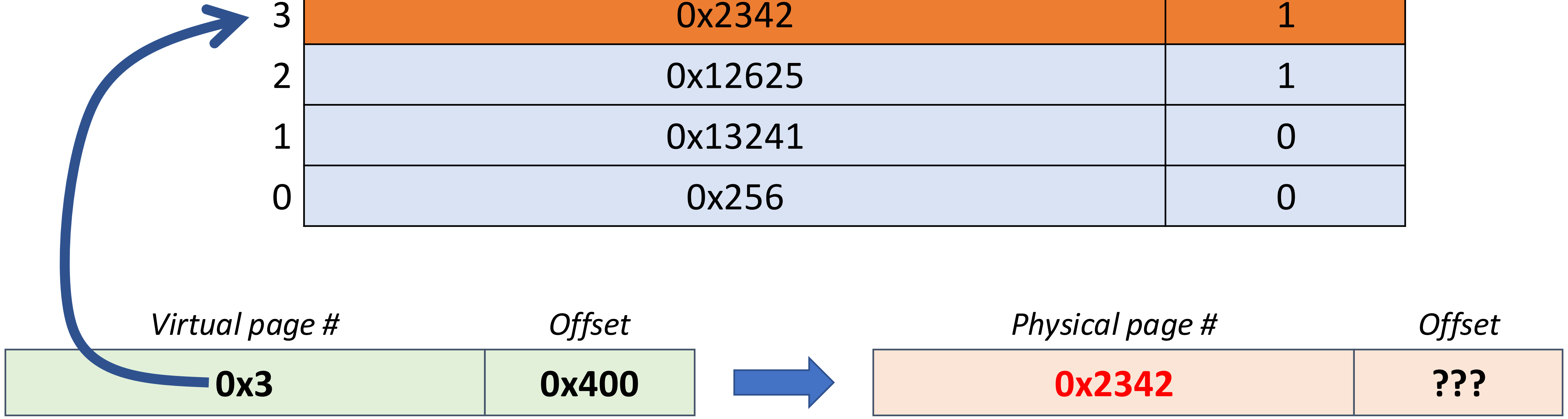
0x3400

Physical Address

???

Page Map

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0



Virtual Address

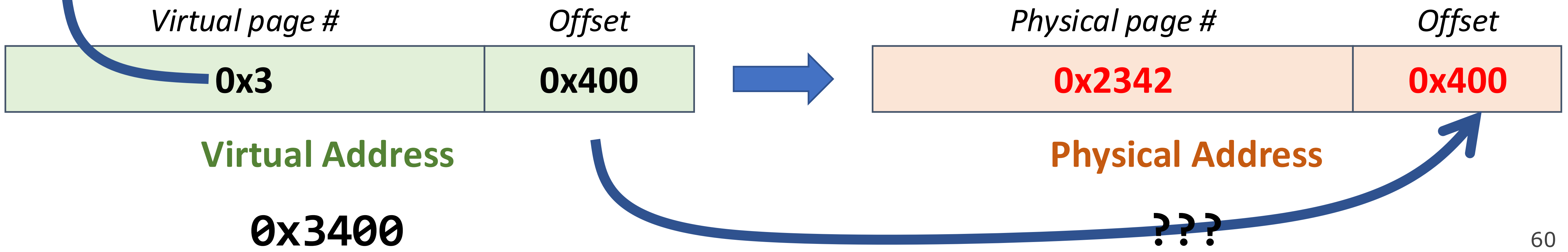
Physical Address

`0x3400`

???

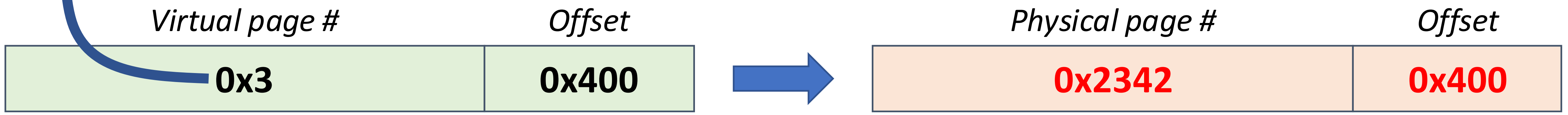
Page Map

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0



Page Map

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0



Virtual Address

0x3400

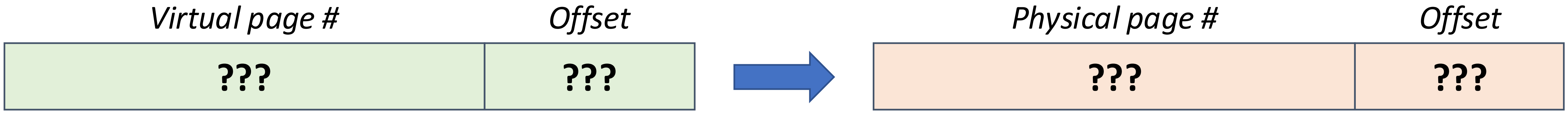
Physical Address

0x2342400

PollEV: What is the physical address?

Respond on PollEv:
pollev.com/cs111

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0



Virtual Address
0x1456

Physical Address
???

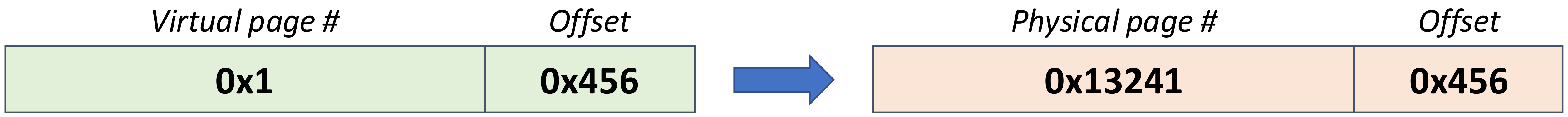
What physical address corresponds with virtual address 0x1456 in this example?

Nobody has responded yet.

Hang tight! Responses are coming in.

Practice: What is the physical address?

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0



Virtual Address
`0x1456`

Physical Address
`0x13241456`

Practice: What is the physical address?

Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0

unused (16 bits)	**Virtual page # (36 bits)**	**Offset (12 bits)**

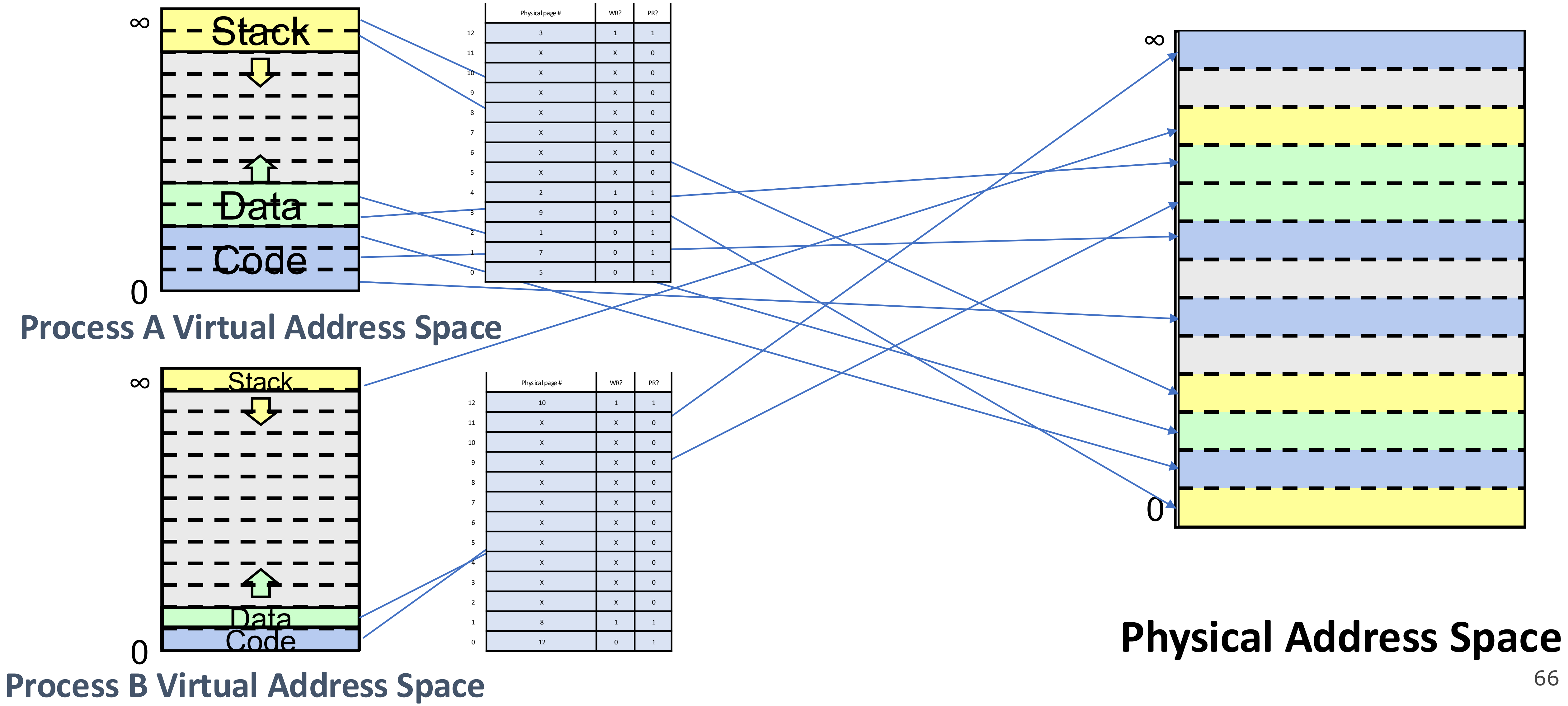
x86-64 64-bit Virtual Address

Physical page # (40 bits)	**Offset (12 bits)**

x86-64 52-bit Physical Address

x86-64 with 4KB pages has 36-bit virtual page numbers and 40-bit physical page numbers.

Each Process Has A Page Map


∞
Stack
Data
Code
0
Process A Virtual Address Space
∞
Stack
Data
Code
0
Process B Virtual Address Space
∞
0
Physical Address Space

	Physical page #	WR?	PR?
12	3	1	1
11	X	X	0
10	X	X	0
9	X	X	0
8	X	X	0
7	X	X	0
6	X	X	0
5	X	X	0
4	2	1	1
3	9	0	1
2	1	0	1
1	7	0	1
0	5	0	1

	Physical page #	WR?	PR?
12	10	1	1
11	X	X	0
10	X	X	0
9	X	X	0
8	X	X	0
7	X	X	0
6	X	X	0
5	X	X	0
4	X	X	0
3	X	X	0
2	X	X	0
1	8	1	1
0	12	0	1

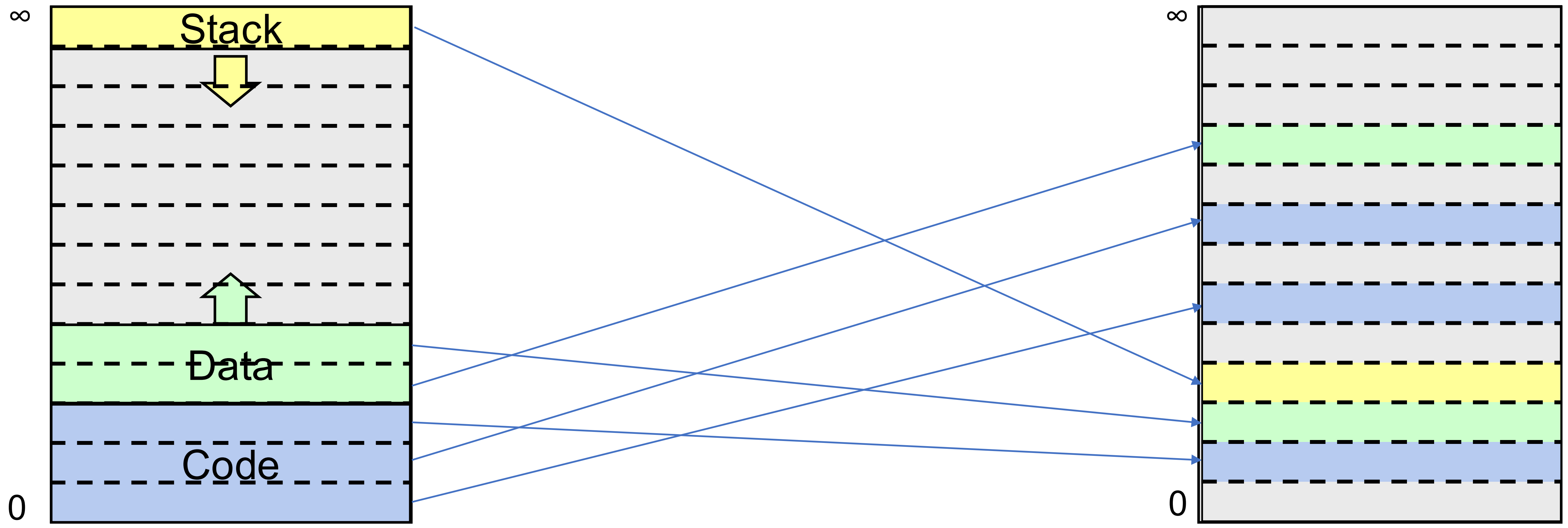
Paging

How do we provide memory to a process?

- Keep a global free list of physical pages – grab the first one when we need one
- Update process page table for a virtual page to map to this physical page

In this way, we can represent a process's segments (e.g. code, data) as a collection of 1 or more pages, starting on any page boundary.

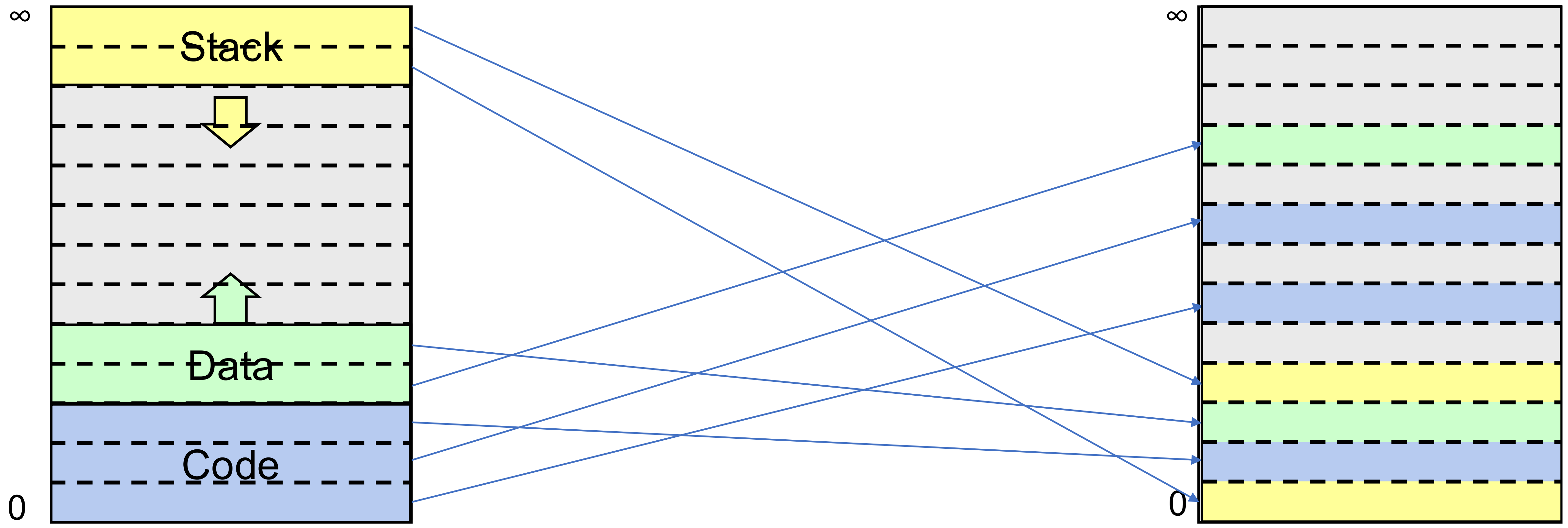
Requesting More Memory



Process A Virtual Address Space

Physical Address Space

Requesting More Memory



Process A Virtual Address Space

Physical Address Space

Paging

Key Idea: Each process's virtual (and physical) memory is divided into fixed-size chunks called *pages*. (Common size is 4KB pages).

- A "page" of virtual memory maps to a "page" of physical memory. No partial pages. No more external fragmentation! (but some internal fragmentation if not all of a page is used).
- The **page number** is a numerical ID for a page. We have virtual page numbers and physical page numbers.
- Each process has a *page map ("page table")* with an entry for each virtual page, mapping it to a physical page number and other info such as a protection bit (read-only or read-write).
- A memory address can tell us the page number and offset within that page.
- Not all pages are mapped – unmapped pages cannot be accessed

Paging

Problem: for the page table to be an array where the indexes are page numbers, we need to have an entry in the array for *every* page number, even if it hasn't been mapped. Missing entries would mean indexes wouldn't line up anymore (e.g. we want index 5 to correspond to virtual page 5).

- **That sounds like a ton of space!**
- **How do we track which pages are "valid"** (mapped + ok to use) **vs. "invalid"** (not mapped)? *We'll store this info in each entry - "present" bit.*

On each memory reference:

- Look up info for that virtual page in the page map
- If it's a valid virtual page number, get the physical page number it maps to, and combine it with the specified offset to produce the physical address.

Next Time

- How do we get enough space for such a big page map? And have one for *every* process?
- What do we do if we run out of physical pages?

Recap

- **Recap:** virtual memory and dynamic address translation
- Approach #2: Multiple Segments
- Approach #3: Paging

Next time: demand paging

Lecture 22 takeaway: Dynamic Address translation means that the OS intercepts and translates each memory access. Initial approaches to this include base+bound per process, or expanding that to be base+bound per variable-length segment, or instead dividing into fixed-size pages.