CS111, Lecture 22
Dynamic Address Translation

Topic 4: 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?

CS111 Topic 4: Virtual Memory

Lecture 21 **Today** **Lecture 23** **Lecture 24** **Lecture 25**

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 #1: Base and Bound
- Approach #2: Multiple Segments
- Approach #3: Paging

Plan For Today

- **Recap: virtual memory and dynamic address translation**
- Approach #1: Base and Bound
- Approach #2: Multiple Segments
- Approach #3: Paging

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

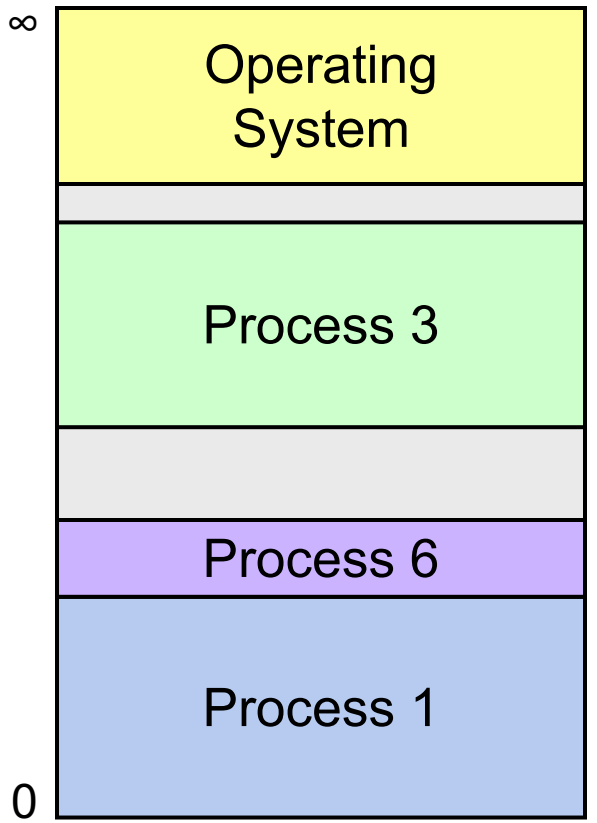
Sharing Memory

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



Idea: What if, instead of translating addresses when a program is loaded, the OS intercepted every memory reference and translated it?

Dynamic Address Translation

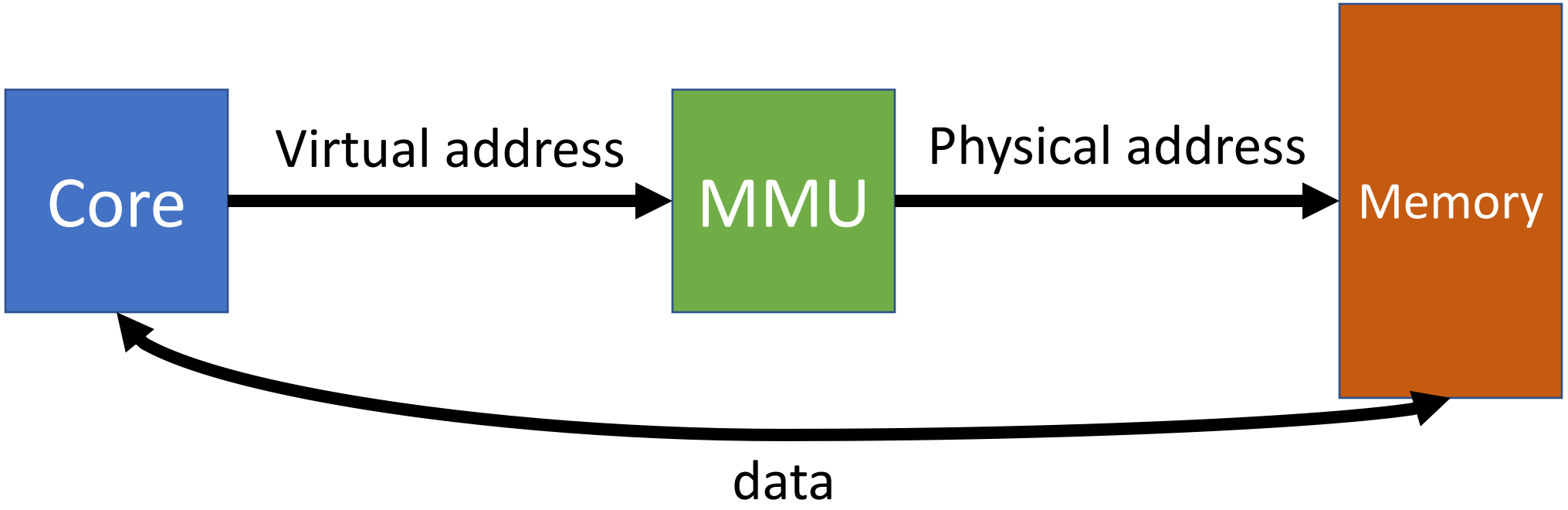
Let's have the OS intercept every memory reference a process makes.

- 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
- The OS will translate each process's address to the real one it's mapped to

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

- Every process can think it starts at address 0 and is the only process in memory
- Behind the scenes, the OS can choose how it maps each process's virtual addresses to real ("physical") addresses
- 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.

Key Question: How do the MMU/OS translate from virtual addresses to physical ones?

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**

Plan For Today

- **Recap:** virtual memory and dynamic address translation
- **Approach #1: Base and Bound**
- Approach #2: Multiple Segments
- Approach #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.

Approach #1: Base and Bound

What are some benefits of this approach?

- Inexpensive translation – just doing addition
- Doesn't require much additional space – just per-process base + bound
- The separation between virtual and physical addresses means we can move the physical memory location and simply update the base, or we could even *swap* memory to disk and copy it back later when it's actually needed.

What are some drawbacks of this approach?

- One contiguous region per program
- Fragmentation
- Growing can only happen upwards with the bound
- Doesn't support read-only regions of memory within a process

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

Plan For Today

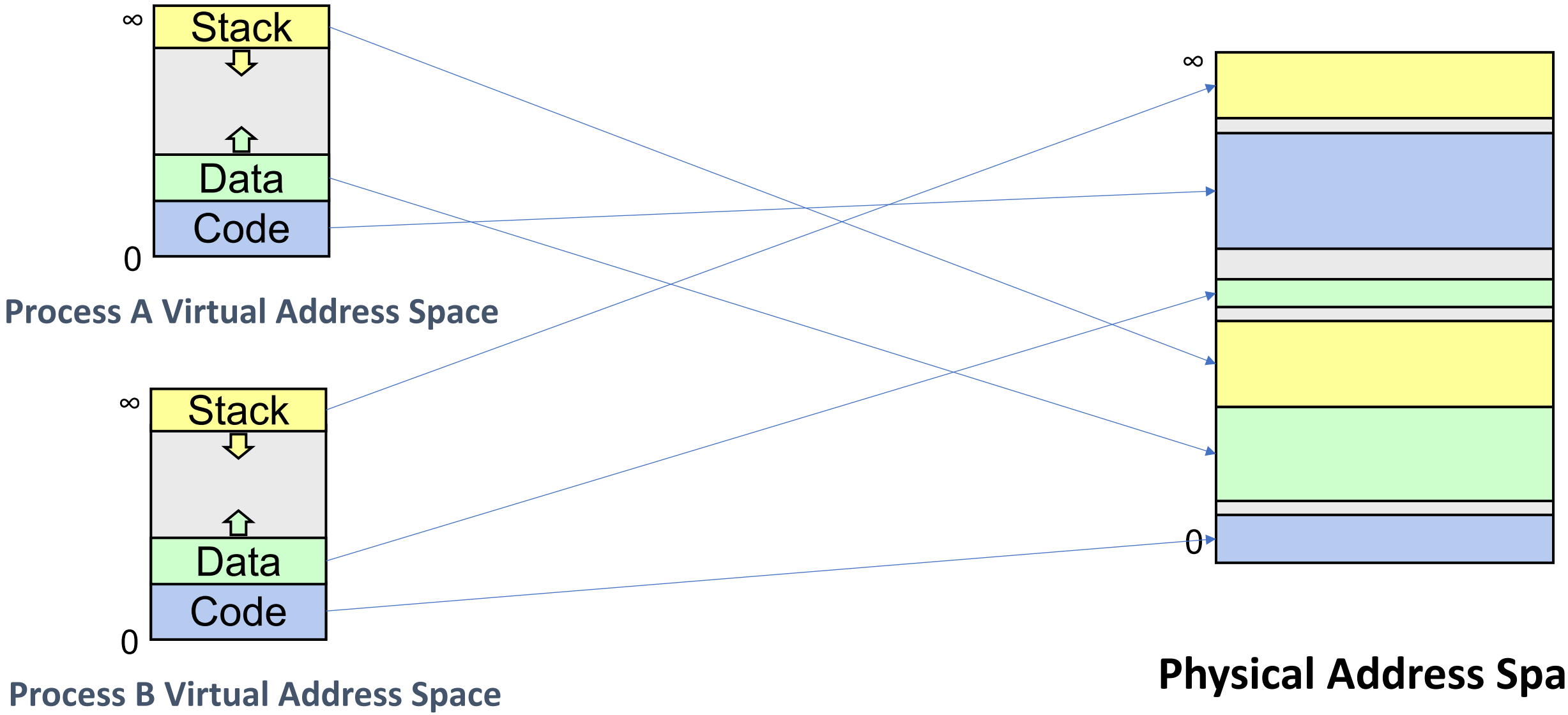
- **Recap:** virtual memory and dynamic address translation
- Approach #1: Base and Bound
- **Approach #2: Multiple Segments**
- Approach #3: Paging

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
- 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



Approach #2: Multiple Segments

On each memory reference:

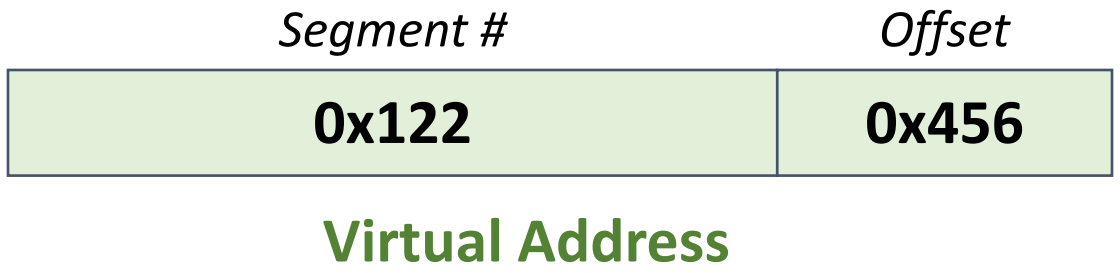
- Look up info for the segment that address is in
- Compare virtual address to that segment's bound, trap if >= (invalid memory reference)
- Add segment's base to virtual address to produce physical address

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.

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.

Another possibility: deduce from machine code instruction executing

Approach #2: Multiple Segments

What are some benefits of this approach?

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

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?)

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

Plan For Today

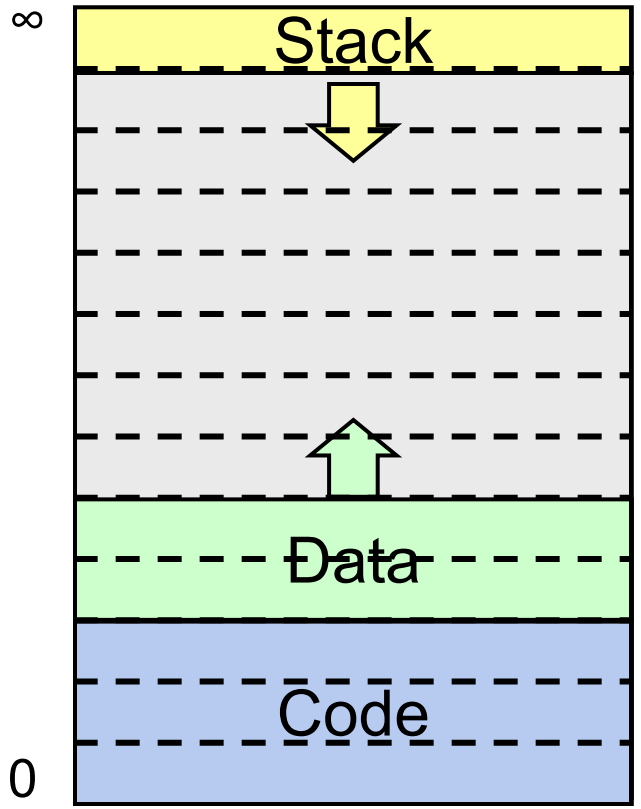
- **Recap:** virtual memory and dynamic address translation
- Approach #1: Base and Bound
- Approach #2: Multiple Segments
- **Approach #3: Paging**

Approach #3: Paging

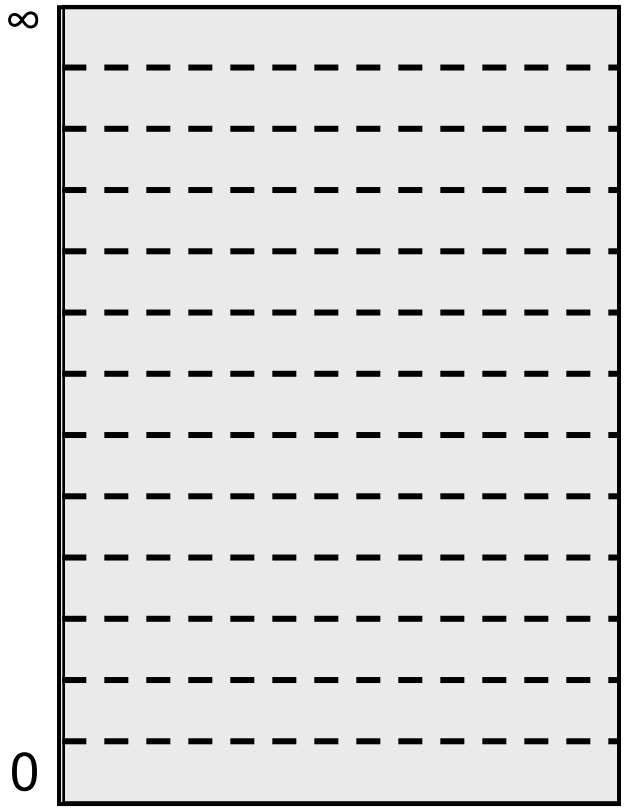
Key Idea: Each process's virtual (and physical) memory is divided into fixed-size chunks called *pages*. (Common size is 4KB pages).

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

Paging

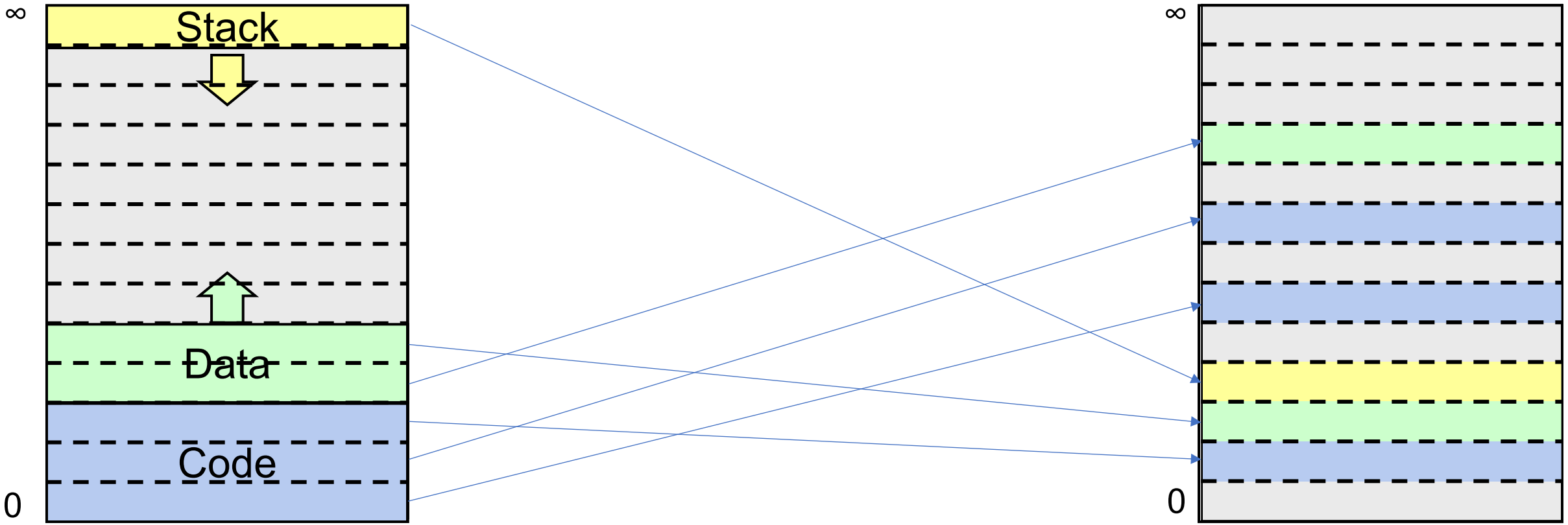


Process A Virtual Address Space



Physical Address Space

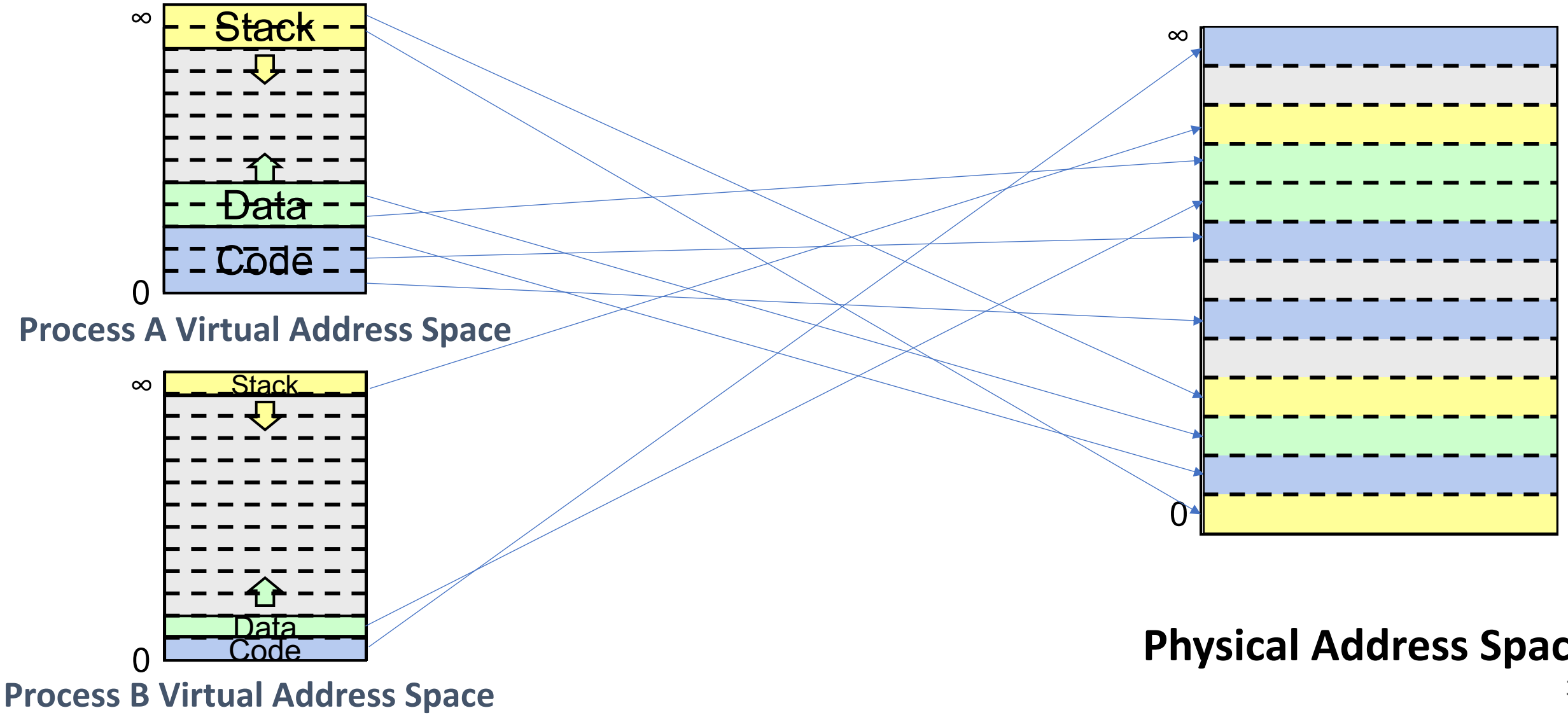
Paging



Process A Virtual Address Space

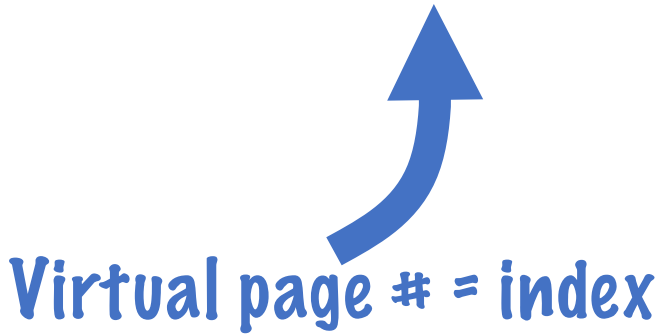
Physical Address Space

Paging


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

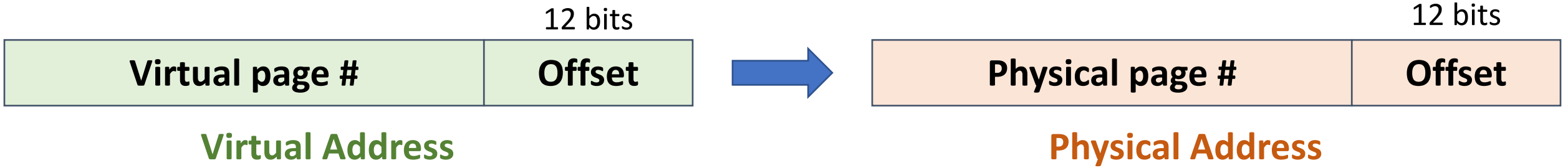
Page Map

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



Page Map

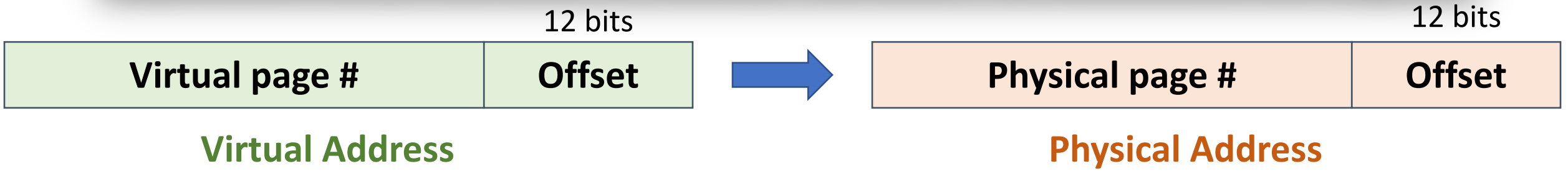
Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1
1	0x13241	0
0	0x256	0



Page Map

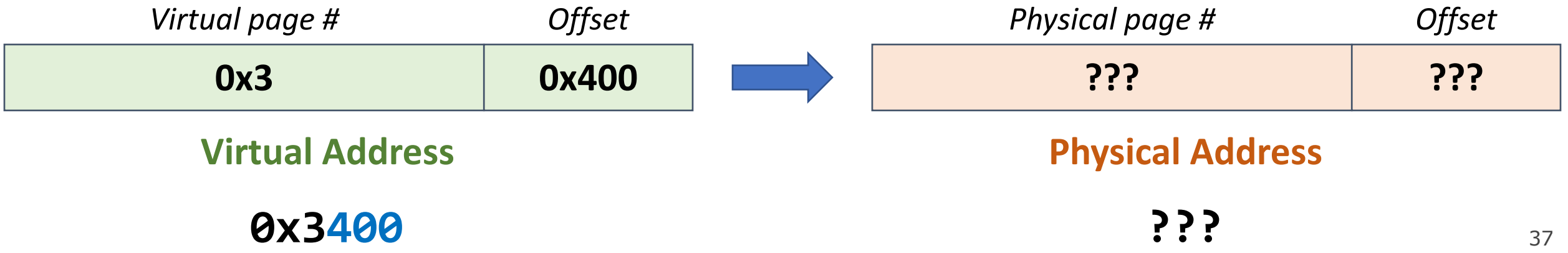
Index	Physical page #	Writeable?
…	…	…
3	0x2342	1
2	0x12625	1

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

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

???

Page Map

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

Virtual Address

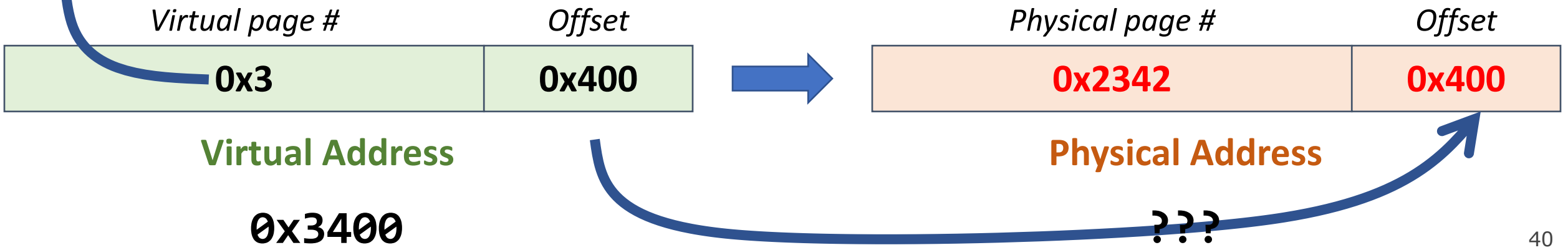
0x3400

Physical Address

???

Page Map

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



Page Map

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

Virtual Address

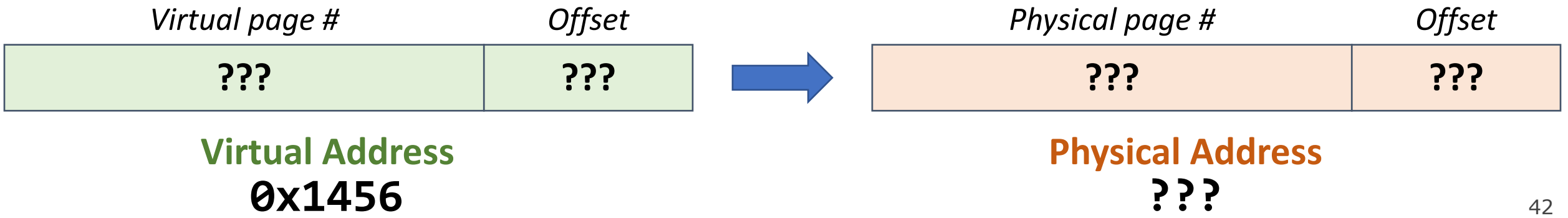
0x3400

Physical Address

0x2342400

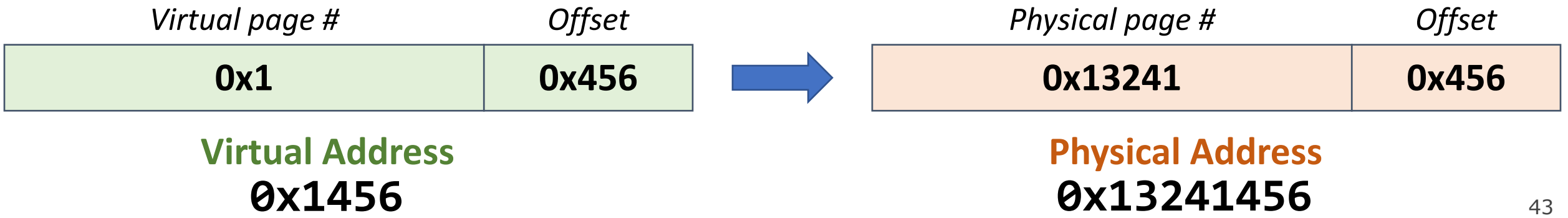
Practice: What is the physical address?

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



Practice: What is the physical address?

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



Practice: What is the physical address?

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

x86-64 64-bit Virtual Address

x86-64 52-bit Physical Address

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

Paging

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.

Problem: what about invalid page numbers? I.e. how do we know/represent which pages are valid or invalid?

Solution: have entries in the page map for *all* pages, including invalid ones. Add an additional field marking whether it's valid ("present").

Page Map

Index	Physical page #	Writeable?	Present?
…	…	…	…
3	0x2342	1	1
2	XXX	X	0
1	0x13241	0	1
0	XXX	X	0

Page Map

Index	Physical page #	Writeable?	Present?
…	…	…	…
3	0x2342	1	1
2	XXX	X	0
1	0x13241	0	1
0	XXX	X	0

If there is a memory access in virtual pages 0 or 2 here, it would trap due to an invalid memory reference.

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, and mark present / set permission bit

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.

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
- Each process has a *page map ("page table")* with an entry for every virtual page, mapping it to a physical page number and other info such as a protection bit (read-only or read-write) and whether it is present.
- The page map is stored in contiguous memory

Problem: how big is a single process's page map? You said an entry for *every* page?

Recap

- **Recap:** virtual memory and dynamic address translation
- Approach #1: Base and Bound
- Approach #2: Multiple Segments
- Approach #3: Paging

Next time: more about 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.