CS111, Lecture 23
Demand Paging

Optional reading:

Operating Systems: Principles and Practice (2nd Edition): Chapter 9

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		**Lecture 22**		**Today**		**Lecture 24**

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

Learning Goals

- Learn about page maps and how they help translate virtual addresses to physical addresses
- Understand how paging allows us to swap memory contents to disk when we need more physical pages.
- Learn about the benefits of demand paging in making memory look larger than it really is

Plan For Today

- **Recap:** Base and bound, multiple segments, and paging
- Page Map Size
- Demand Paging

Plan For Today

- **Recap: Base and bound, multiple segments, and paging**
- Page Map Size
- Demand Paging

Dynamic Address Translation

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

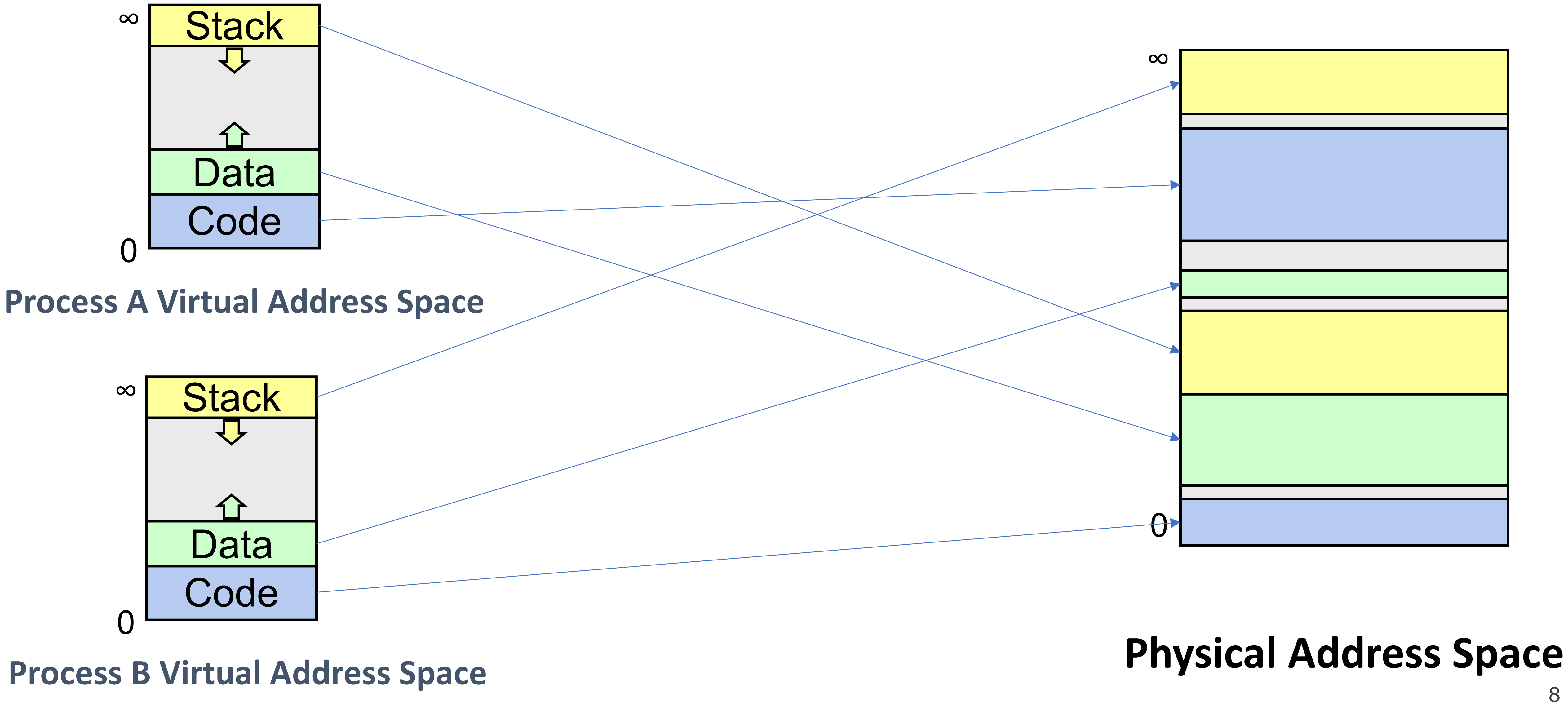
1. **Base and bound**
2. **Multiple Segments**
3. **Paging**

Approach #2: Multiple Segments

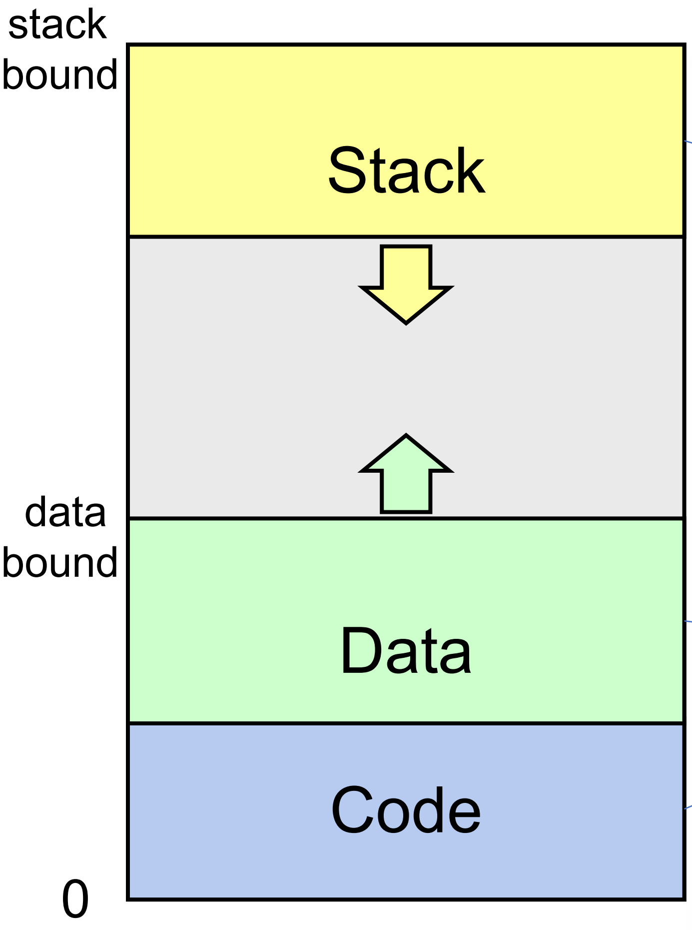
Key Idea: Each process is split among several variable-size areas of memory, called segments.

- 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).
- Top bit(s) of virtual address encode segment number, rest encode offset

Multiple Segments

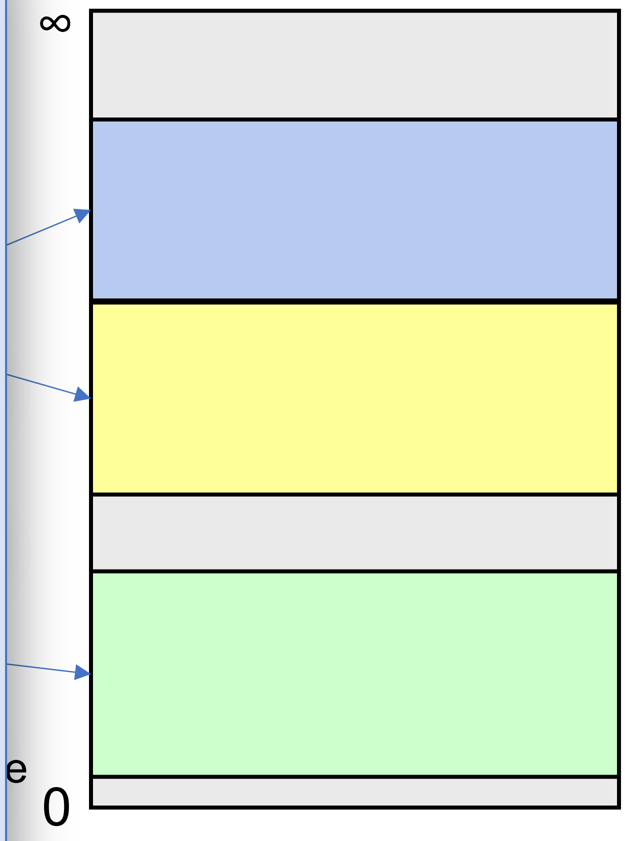


Multiple Segments – Changing A Bound



Process A Virtual Address Space

- Buffer space between stack + heap doesn't need to be initially mapped.
- Growing a segment upwards 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.
- Still fragmentation problem, plus need to decide # bits for segment number vs. offset.



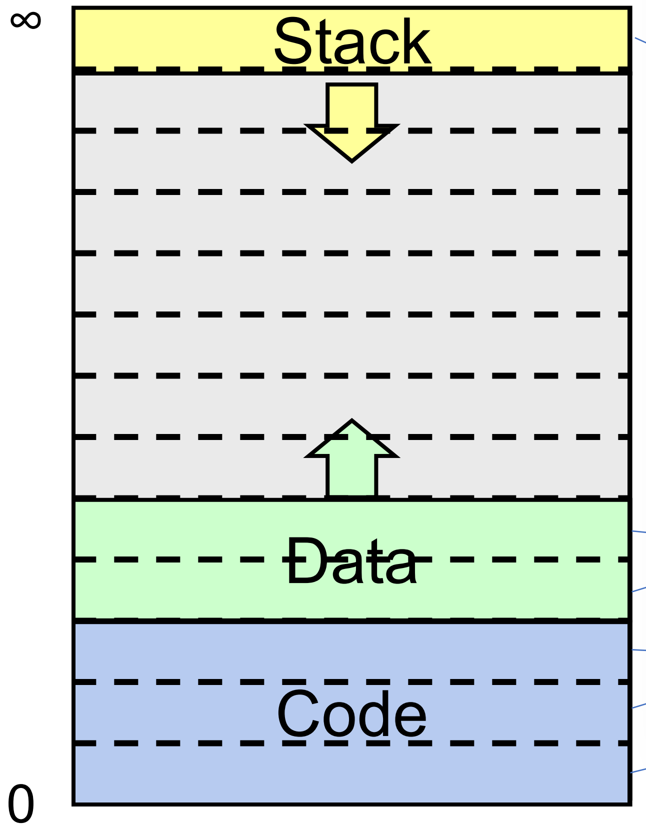
Physical Address Space

Paging

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

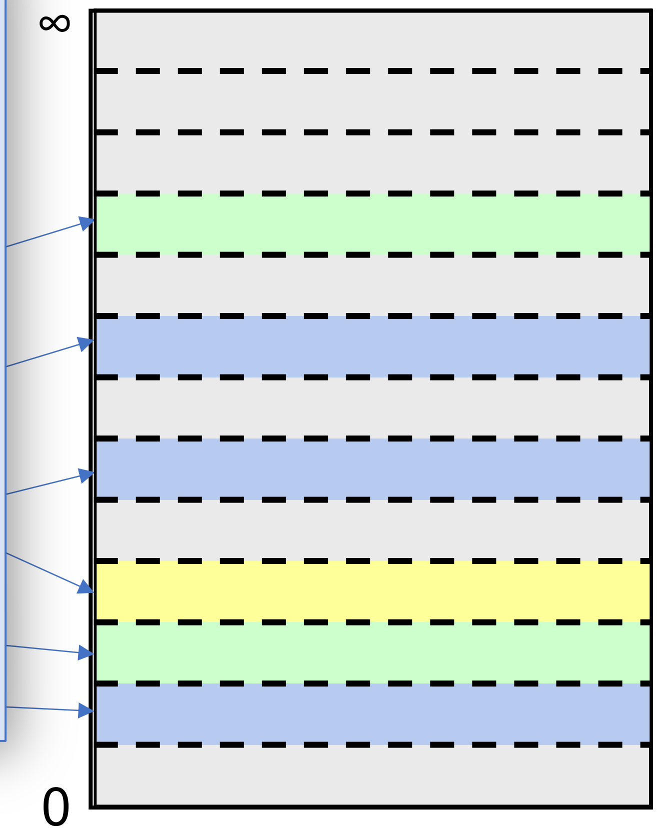
- A "page" of virtual memory maps to a "page" of physical memory. No partial pages. No more external fragmentation! (but some internal fragmentation if not all of a page is used).
- The **page number** is a numerical ID for a page. We have virtual page numbers and physical page numbers.
- Each process 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.

Paging



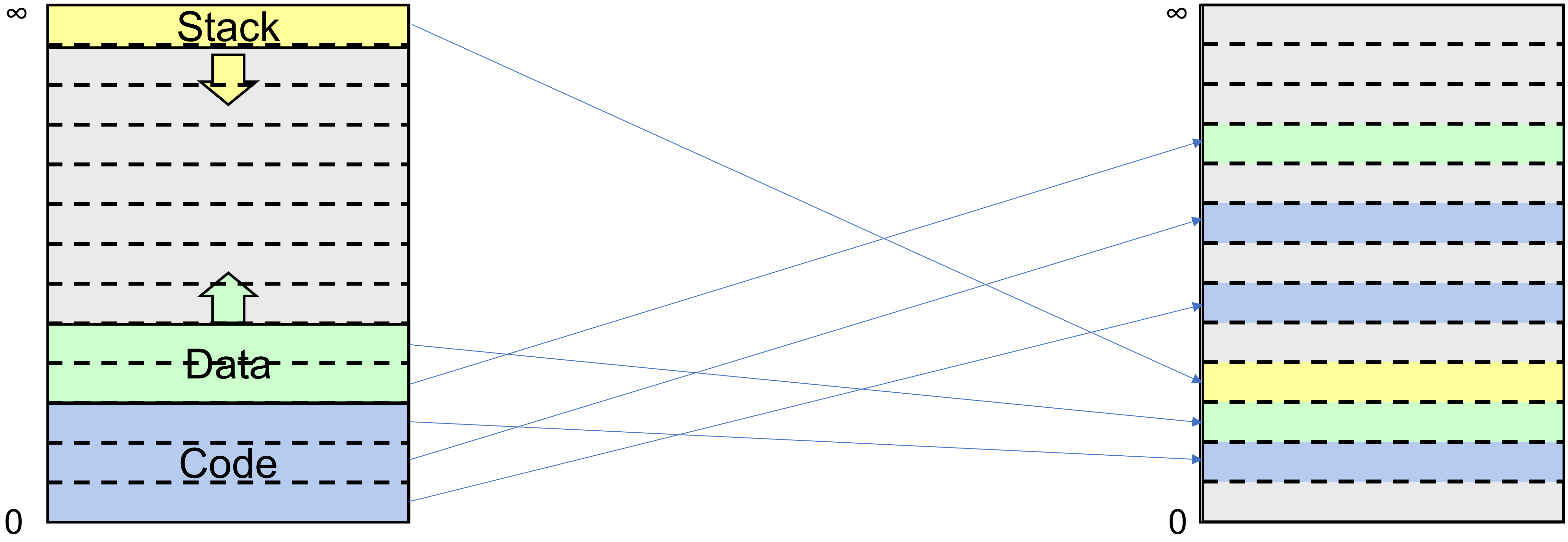
Process A Virtual Address Space

- 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 physical address**.
- We can move each page separately in physical memory by modifying the page map.



Physical Address Space

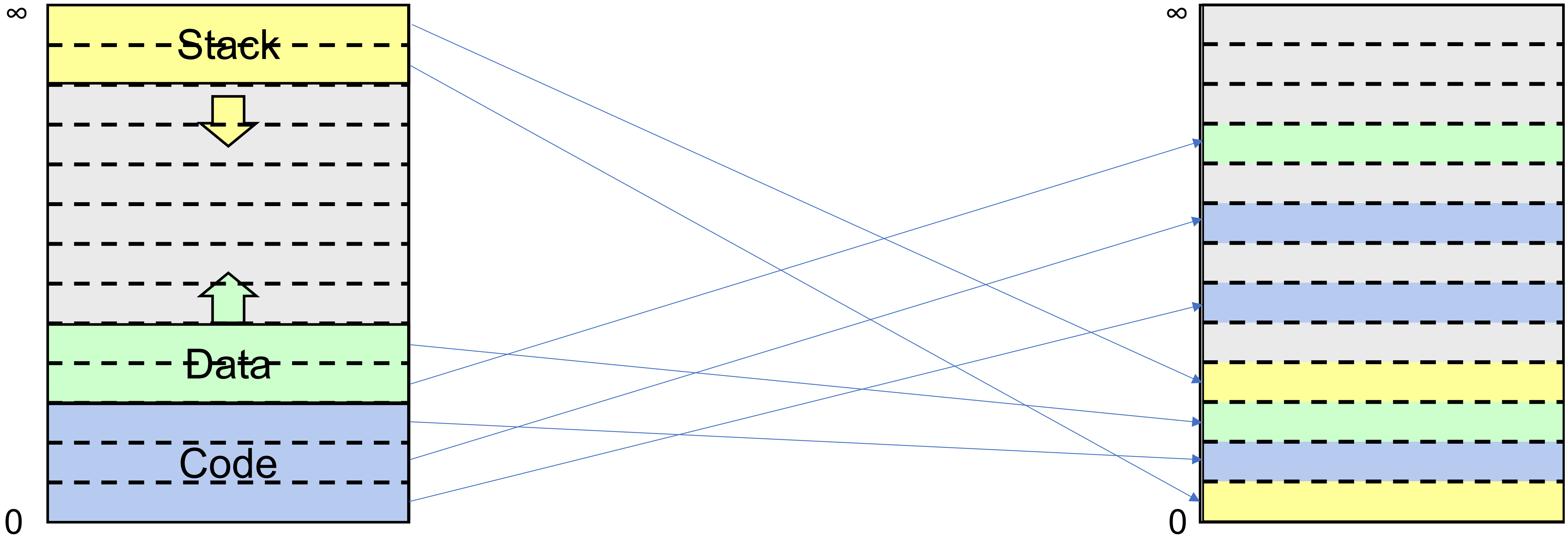
Paging



Process A Virtual Address Space

Physical Address Space

Paging



Process A Virtual Address Space

Physical Address Space

Page Map

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

Virtual page #	*Offset*
0x3	**0x400**

Virtual Address

0x3400

→

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

Physical Address

???

Page Map

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

Virtual Address

Virtual page #	*Offset*
0x3	**0x400**

0x3400

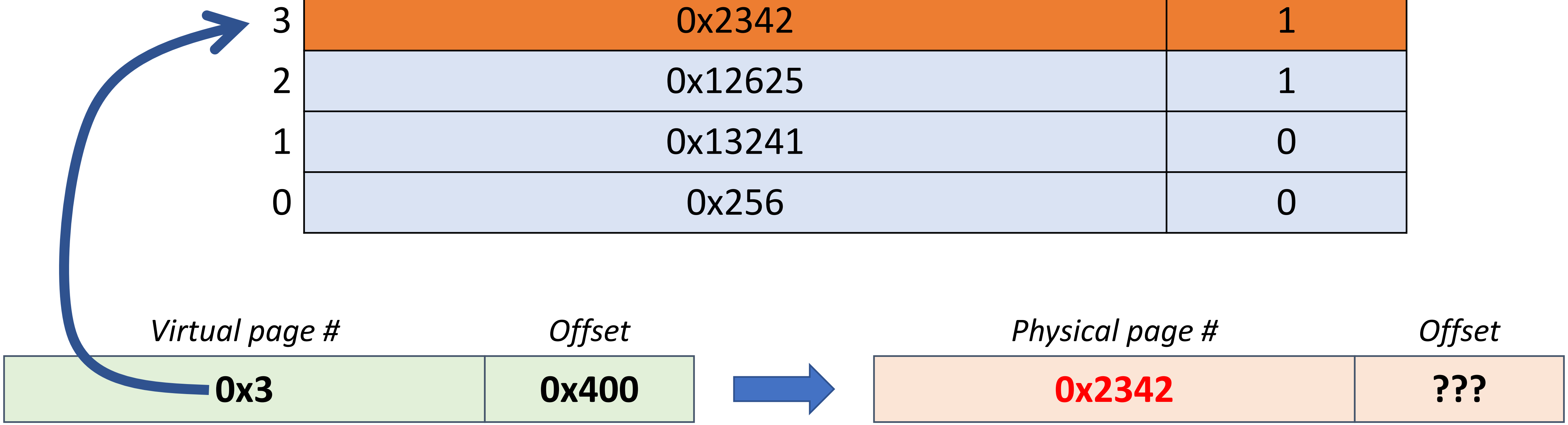
Physical Address

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

???

Page Map

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



Virtual Address

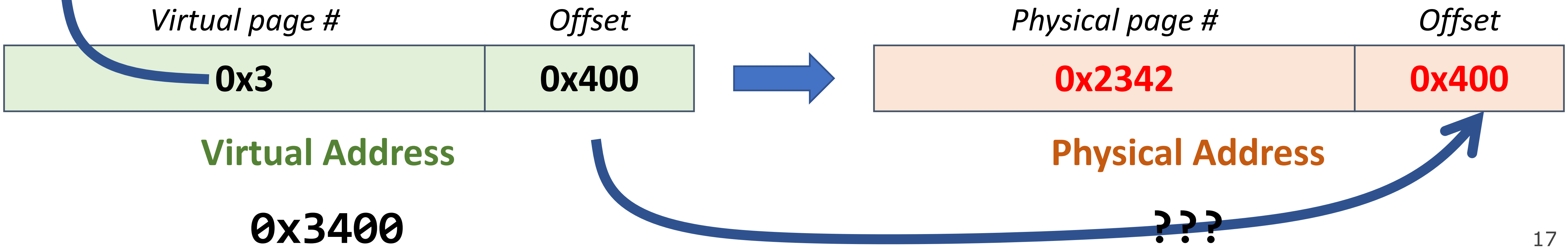
Physical Address

0x3400

???

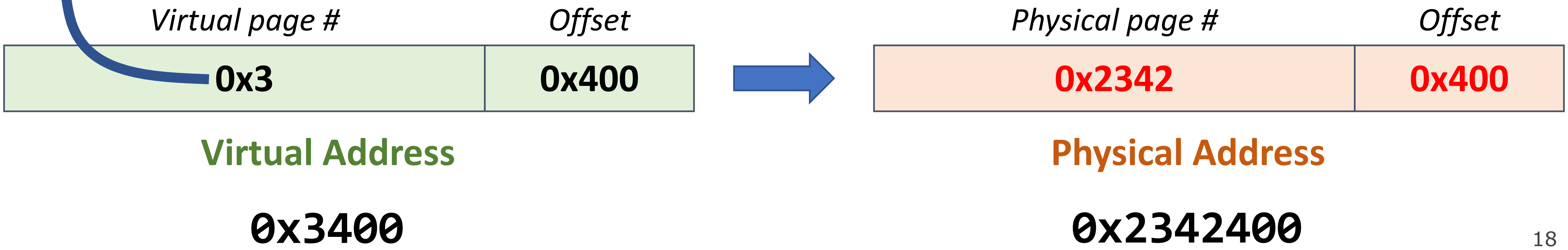
Page Map

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



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?

Plan For Today

- **Recap:** Base and bound, multiple segments, and paging
- **Page Map Size**
- Demand Paging

Paging

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.

Page Map Size

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

Example with x86-64: 36-bit virtual page numbers, 8-byte map entries

How many possible virtual page #s? 2^{36}

2^{36} virtual pages x 8 bytes per page entry = ???

Page Map Size

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

Example with x86-64: 36-bit virtual page numbers, 8-byte map entries

How many possible virtual page #s? 2^{36}

2^{36} virtual pages x 8 bytes per page entry = **512GB!!** (2^{39} bytes)

Plus, most processes are small, so most pages will be "not present". And even large processes use their address space sparsely (e.g. code at bottom, stack at top).

Page Map Size

x86-64 solution: represent the page map as a multi-level *tree.*

- Top level of page map has entries for *ranges of virtual pages* (0 to 2^{27}-1), 2^{27} to 2^{54} – 1, etc.). **Only if** any pages in that range are present does that entry point to a lower level in the tree (saves space).
- Lower levels follow a similar structure – entry for ranges of pages, and they only map to something if at least one of the pages in that range is present.
- The lowest level of the tree contains actual physical page numbers.

assign6

On assign6, you'll implement your own virtual memory system using paging:

- You'll intercept memory requests
- You'll maintain a page map mapping virtual addresses to physical ones
- For our purposes, we won't worry about page map size (will store it without using tree structure)

Plan For Today

- **Recap:** Base and bound, multiple segments, and paging
- Page Map Size
- **Demand Paging**

What should we do if we run out of physical memory?

Running Out Of Memory

If memory is in high demand, we could fill up all of memory, since a process needs all its pages in memory to run. What should we do in that case?

- Prohibit further program memory requests until some is freed? Not ideal.
- Another idea – what if we kicked out a page and used that page? We could save a page to disk, use the page for new data, and load the old data back in to a physical page later if it's still needed.

We can make physical memory look larger than it is!

Demand Paging

Overall goal: allow programs to run without all their information in memory.

- Keep in memory the information that is being used.
- Keep unused information on disk in *paging file* (also called backing store, or swap space)
- Move information back and forth as needed.
- Locality – most programs spend most of their time using a small fraction of their code and data

Ideally: we have a memory system with the performance of main memory and the cost/capacity of disk!

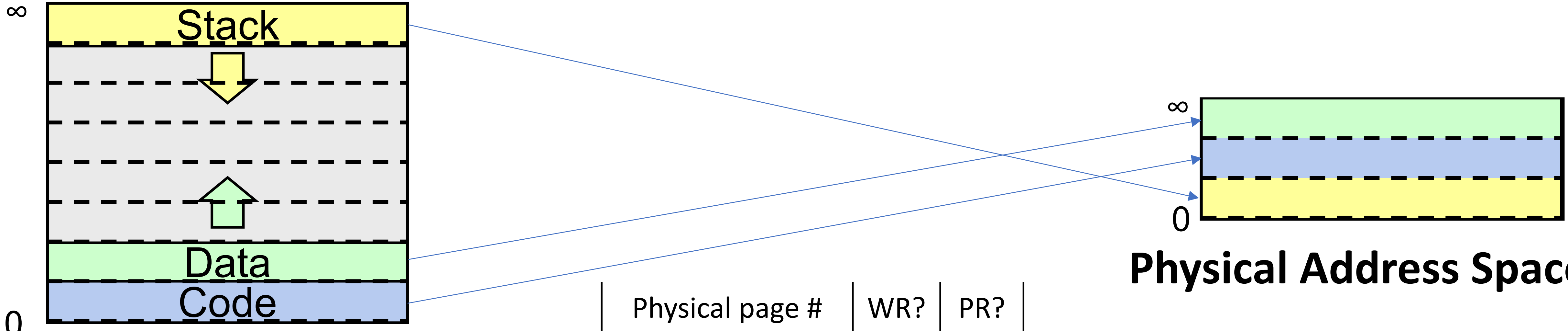
Demand Paging – 2 Key Questions

1. What is the process for kicking a page out to disk?
2. How do we choose which page to kick out? (next time!)

Demand Paging – 2 Key Questions

1. **What is the process for kicking a page out to disk?**
2. How do we choose which page to kick out? (next time!)

Demand Paging

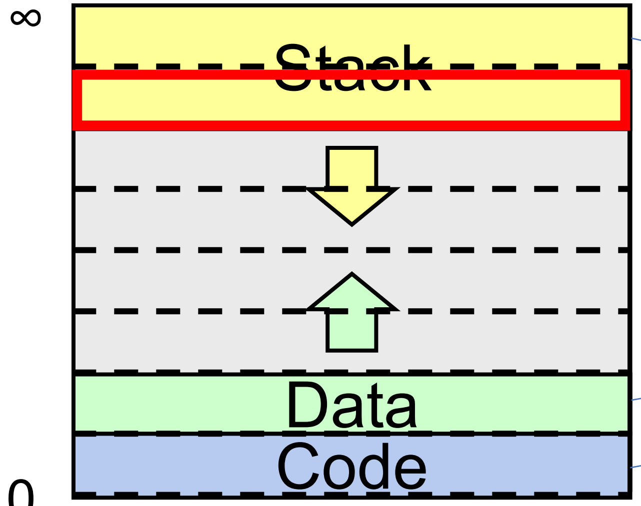


Process A Virtual Address Space

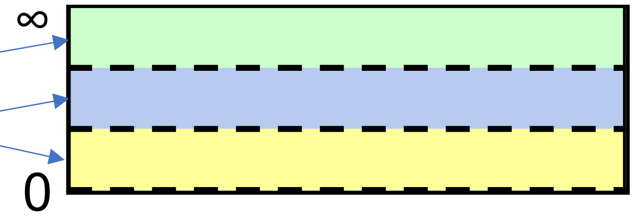
Physical Address Space

	Physical page #	WR?	PR?
7	0	1	1
6	X	X	0
5	X	X	0
4	X	X	0
3	X	X	0
2	X	X	0
1	2	0	1
0	1	0	1

Demand Paging



Process A Virtual Address Space

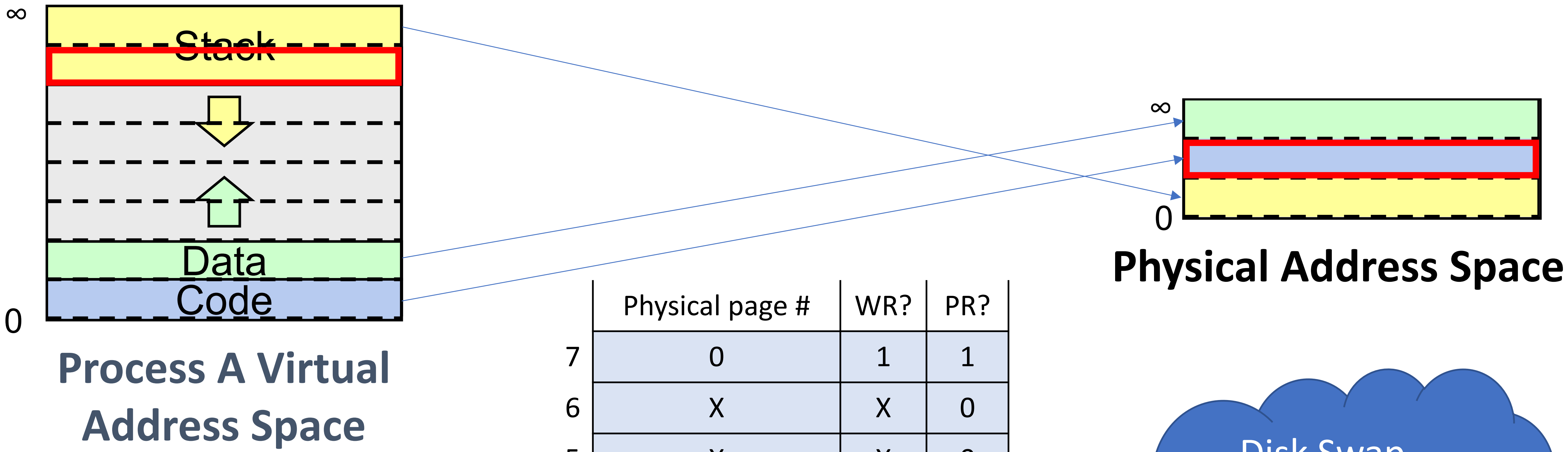


Physical Address Space

	Physical page #	WR?	PR?
7	0	1	1
6	X	X	0
5	X	X	0
4	X	X	0
3	X	X	0
2	X	X	0
1	2	0	1
0	1	0	1

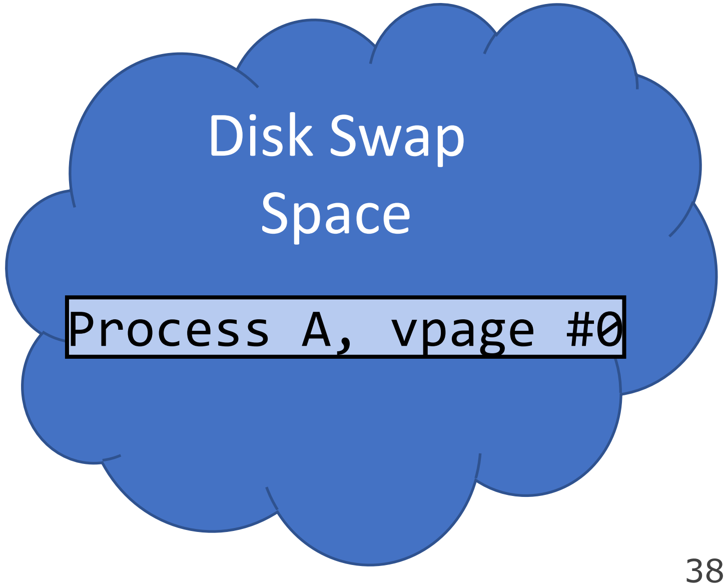
1. Pick an existing physical page and swap it to disk.

Demand Paging

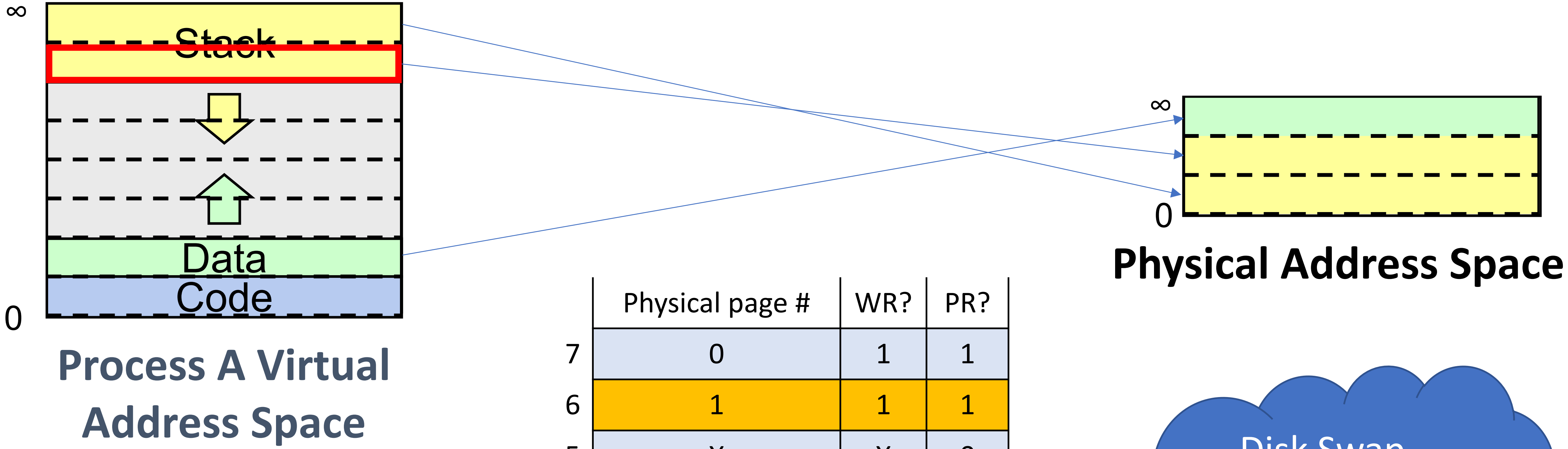


	Physical page #	WR?	PR?
7	0	1	1
6	X	X	0
5	X	X	0
4	X	X	0
3	X	X	0
2	X	X	0
1	2	0	1
0	1	0	0

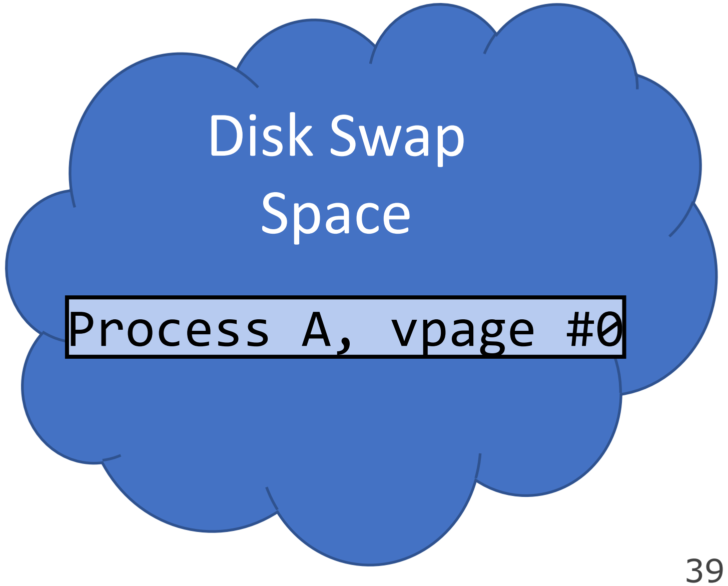
1. Pick an existing physical page and swap it to disk, mark not present.



Demand Paging

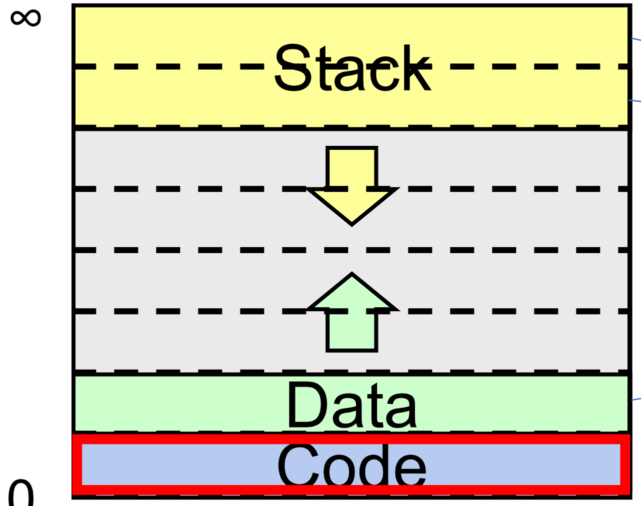


	Physical page #	WR?	PR?
7	0	1	1
6	1	1	1
5	X	X	0
4	X	X	0
3	X	X	0
2	X	X	0
1	2	0	1
0	1	0	0

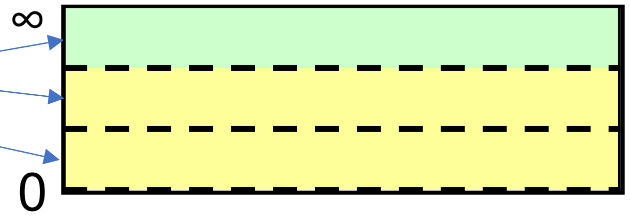


2. Map this physical page to the new virtual page.

Demand Paging



Process A Virtual Address Space



Physical Address Space

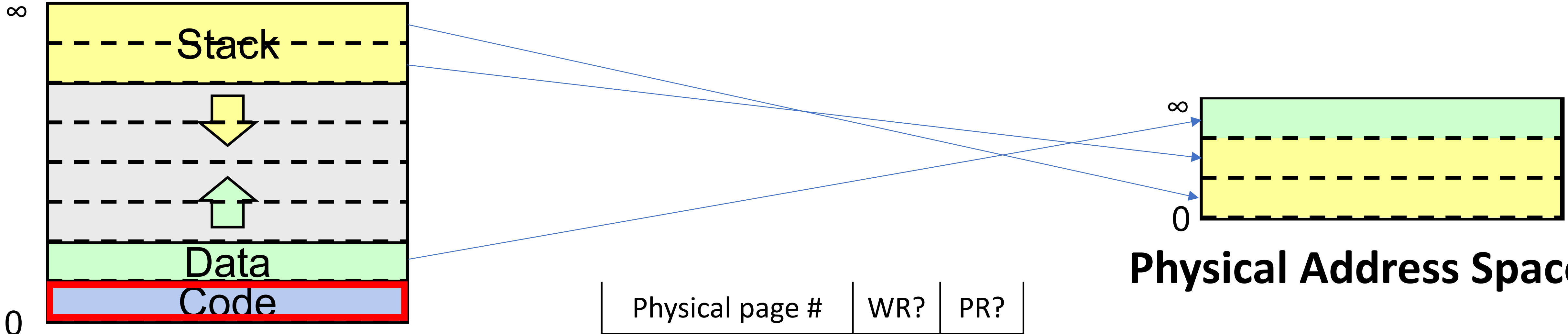
	Physical page #	WR?	PR?
7	0	1	1
6	1	1	1
5	X	X	0
4	X	X	0
3	X	X	0
2	X	X	0
1	2	0	1
0	1	0	0

Disk Swap Space

`Process A, vpage #0`

1. We look in the page map and see it's not present.

Demand Paging

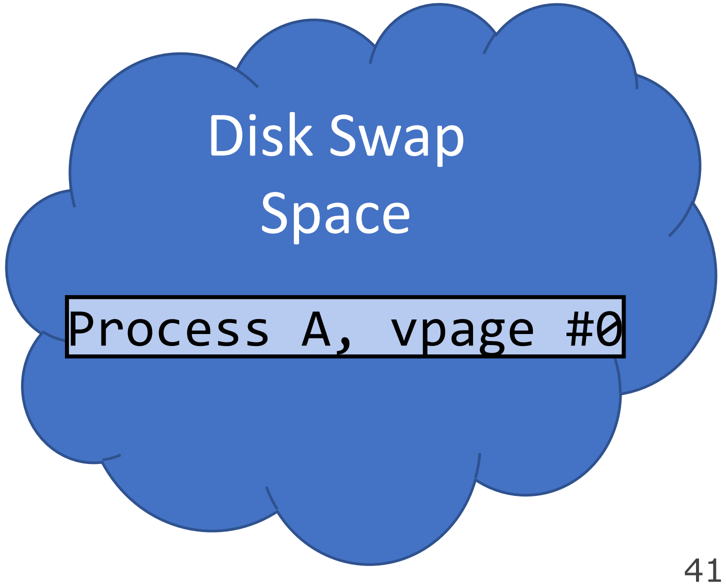


Physical Address Space

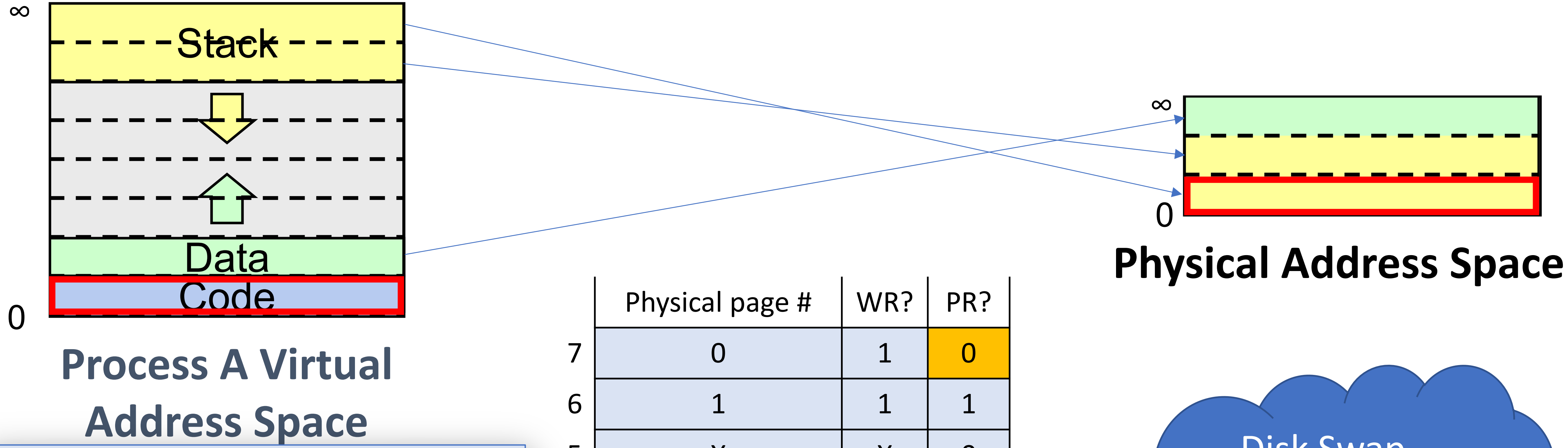
Process A Virtual Address Space

2. But it is stored in disk swap, so we load it back in (kicking another page if needed).

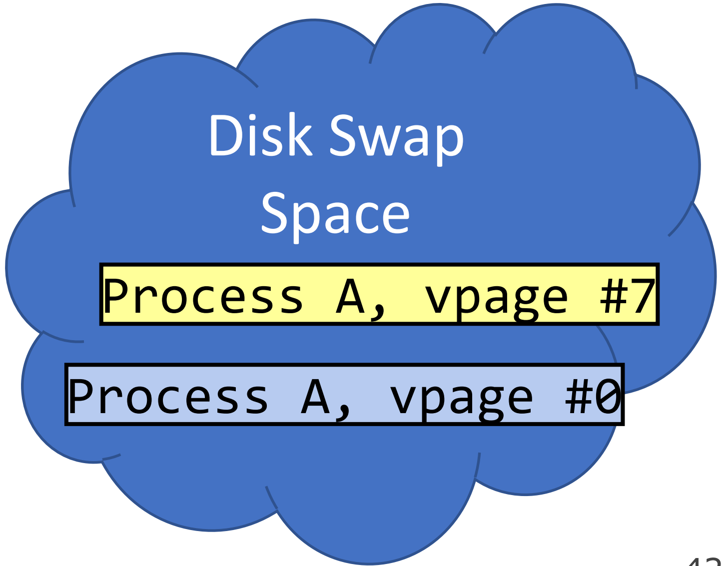
	Physical page #	WR?	PR?
7	0	1	1
6	1	1	1
5	X	X	0
4	X	X	0
3	X	X	0
2	X	X	0
1	2	0	1
0	1	0	0



Demand Paging

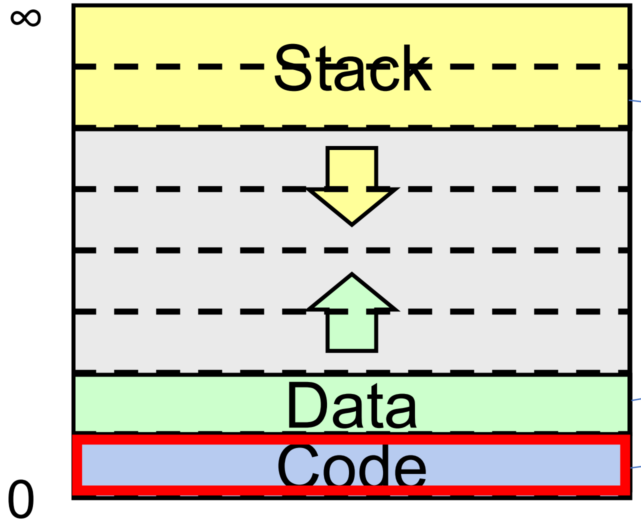


	Physical page #	WR?	PR?
7	0	1	0
6	1	1	1
5	X	X	0
4	X	X	0
3	X	X	0
2	X	X	0
1	2	0	1
0	1	0	0

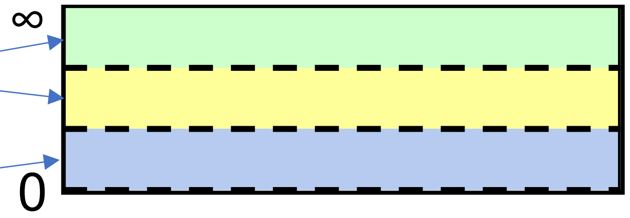


2. But it is stored in disk swap, so we load it back in (kicking another page if needed).

Demand Paging



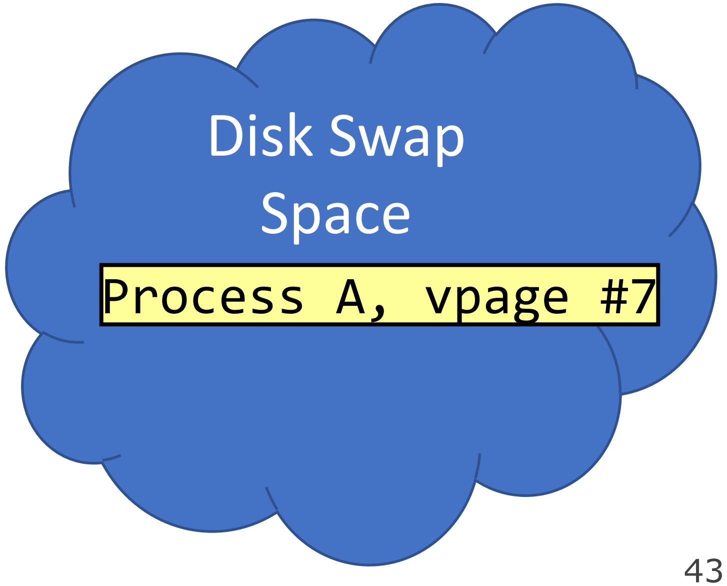
Process A Virtual Address Space



Physical Address Space

2. But it is stored in disk swap, so we load it back in (kicking another page if needed).

	Physical page #	WR?	PR?
7	0	1	0
6	1	1	1
5	X	X	0
4	X	X	0
3	X	X	0
2	X	X	0
1	2	0	1
0	0	0	1



Demand Paging

If we need another page but memory is full:

1. Pick a page to kick out
2. Write it to disk
3. Mark the old page map entry as not present
4. Update the new page map entry to be present and map to this physical page

Demand Paging

If the program accesses a page that was swapped to disk:

1. Triggers a page fault (not-present page accessed)
2. We see disk swap contains data for this page
3. Get a new physical page (perhaps kicking out another one)
4. Load the data from disk into that page
5. Update the page map with this new mapping

Thrashing

Demand paging can provide big benefits – but what potential scenario would lead demand paging to slow the system way down, where it is spending **all its time swapping pages and no time doing useful work?**

Respond on PollEv: pollev.com/cs111
or text CS111 to 22333 once to join.

What potential scenario would lead demand paging to slow the system way down?

0

Nobody has responded yet.

Hang tight! Responses are coming in.

Thrashing

Demand paging can provide big benefits – but what potential scenario would lead demand paging to slow the system way down?

If the pages being actively used don't all fit in memory, the system will spend all its time reading and writing pages to/from disk and won't get much work done.

- Called *thrashing*
- The page we kick to disk will be needed very soon, so we will bring it back and kick another page, which will be needed very soon, etc….
- Progress of the program will make it look like access time of memory is as slow as disk, rather than disk being as fast as memory. ☹
- With personal computers, users can notice thrashing and kill some processes

Page Fetching

Now we have a mechanism to allow programs to run without all their information in memory. But even if there is space, when should we bring pages into memory?

- Most modern OSes start with no pages loaded, load pages when referenced (“demand fetching”).
- Alternative: *prefetching* - try to predict when pages will be needed and load them ahead of time (requires predicting the future…)

Demand Paging Behaviors

- We don't *always* need to write a swapped-out page to disk (e.g., read-only code pages can always be loaded from executable)
- A page may have initial data even if it's never been accessed before (e.g., initialized global variables at program start.)

Kinds of Pages

The pages for a process divide into three groups:

1. **Read-only code pages**: program code, doesn’t change
   A. no need to store in swap when kicked out; can always read them from executable file
   B. on first access, the program expects them to contain data
2. **Initialized data pages:** program data with initial values (e.g., globals)
   A. save to swap since contents may have changed from initial values
   B. on first access, the program expects them to contain data
3. **Uninitialized data pages:** e.g., stack, heap
   A. save to swap as needed
   B. no set initial contents – on first access, just clear memory to all zeros

Assign6 Disk Swap

On assign6:

- You'll only write to disk if a page is "dirty" (modified). Page maps contain a dirty bit that is set whenever a page is modified.
- A page may have contents on disk from the executable or from a previous swap – you'll read into memory in both cases.

Page Replacement

If we need another physical page but all memory is used, **which page should we throw out?**

More next time…

Recap

- **Recap:** Base and bound, multiple segments, and paging
- Page Map Size
- Demand Paging

Next time: how to choose which pages to swap to disk (the clock algorithm).

Lecture 23 takeaway: We can make memory appear larger than it is by swapping pages to disk when we need more space and swapping them back later. But thrashing can occur when the system spends all its time doing disk operations and little time on actual work.