Address Translation

Introduction

Hammurabi Mendes

Fall 18

Goals

Goals

- Sharing

Goals

- Sharing
  - Two processes with the same program

Goals

- Sharing
  - Two processes with the same program
  - We do not want to load the *same code section* twice

Goals

- Sharing
  - Two processes with the same program
  - We do not want to load the *same code section* twice
- Memory Protection

Goals

- Sharing
  - Two processes with the same program
  - We do not want to load the *same code section* twice
- Memory Protection
  - Processes should not mutate their code, *only their data*

Goals

- Sharing
  - Two processes with the same program
  - We do not want to load the *same code section* twice
- Memory Protection
  - Processes should not mutate their code, *only their data*
  - Processes cannot see each other's data

Goals

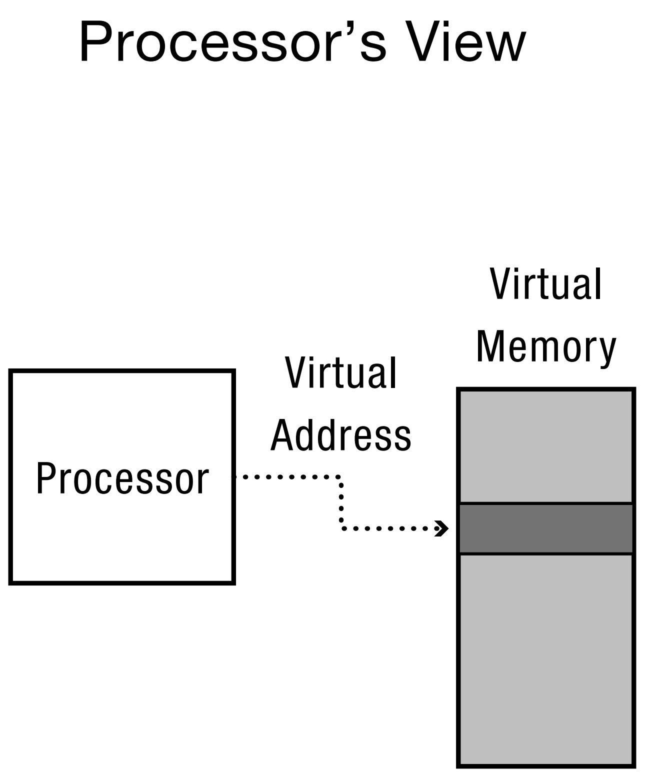
- Sharing
  - Two processes with the same program
  - We do not want to load the *same code section* twice
- Memory Protection
  - Processes should not mutate their code, *only their data*
  - Processes cannot see each other's data
- Relocation Features

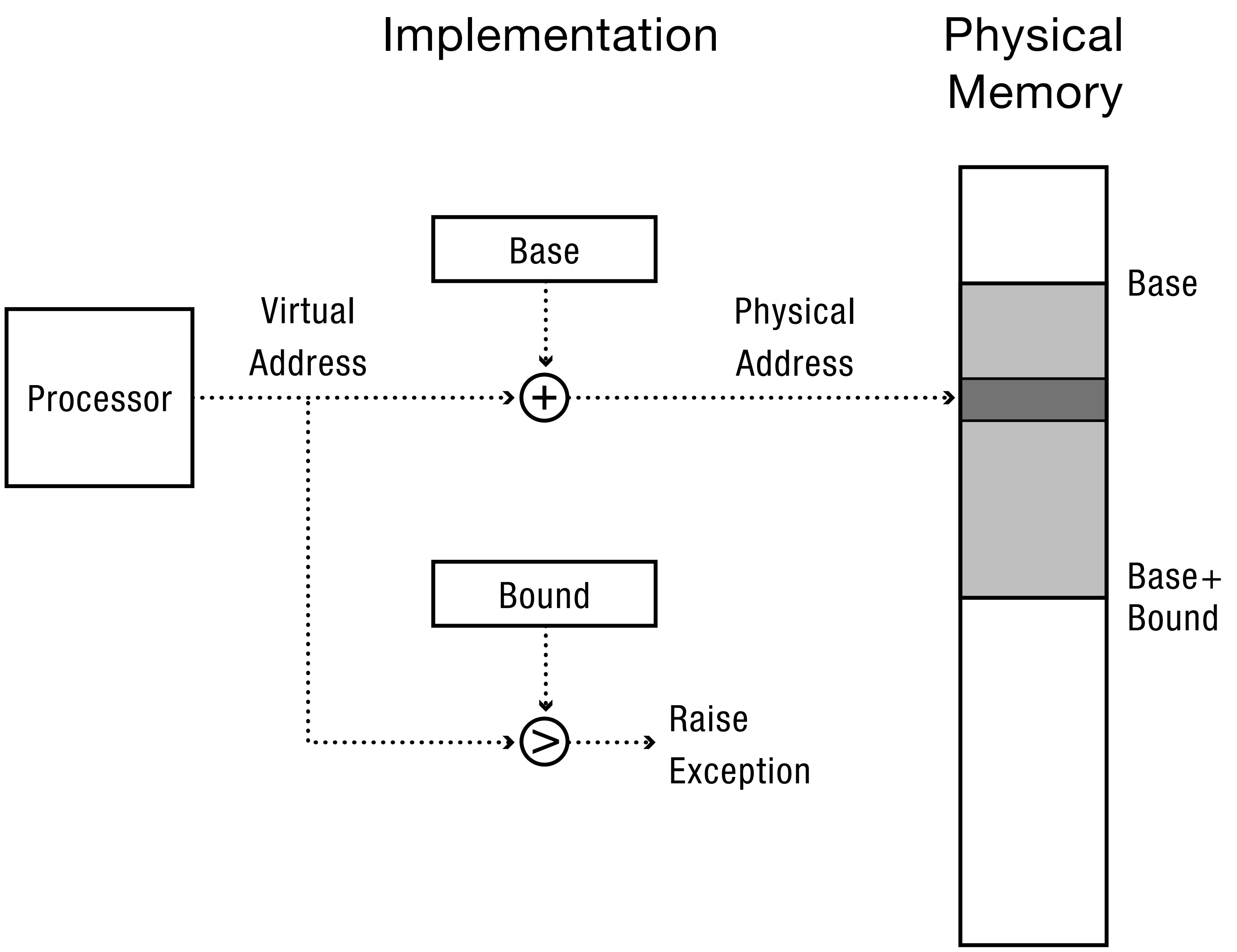
Goals

- Sharing
  - Two processes with the same program
  - We do not want to load the *same code section* twice
- Memory Protection
  - Processes should not mutate their code, *only their data*
  - Processes cannot see each other's data
- Relocation Features
  - We want programs to be allocated in memory *fast*

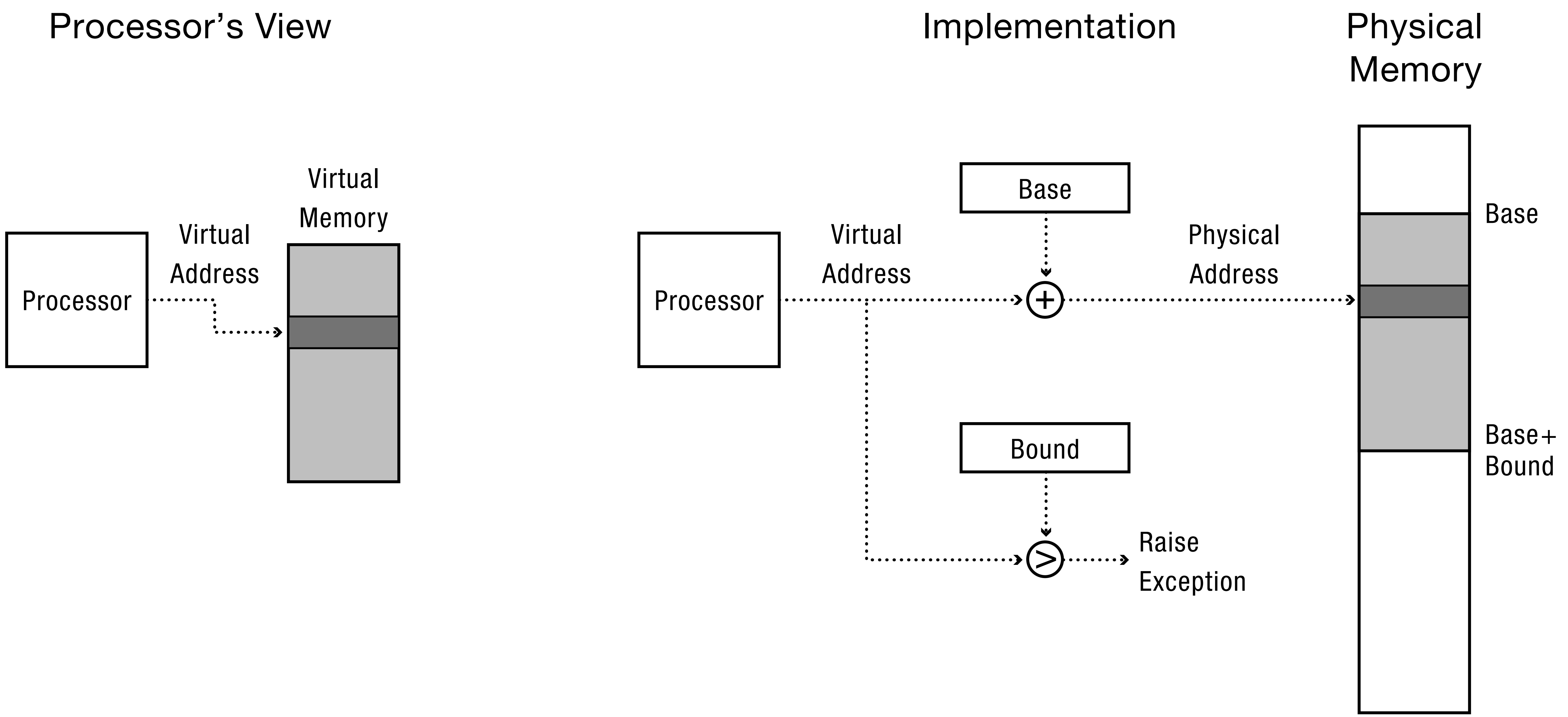
Approach 1: Base & Bound

Base & Bound


Processor's View
Virtual Memory
Processor
Virtual Address

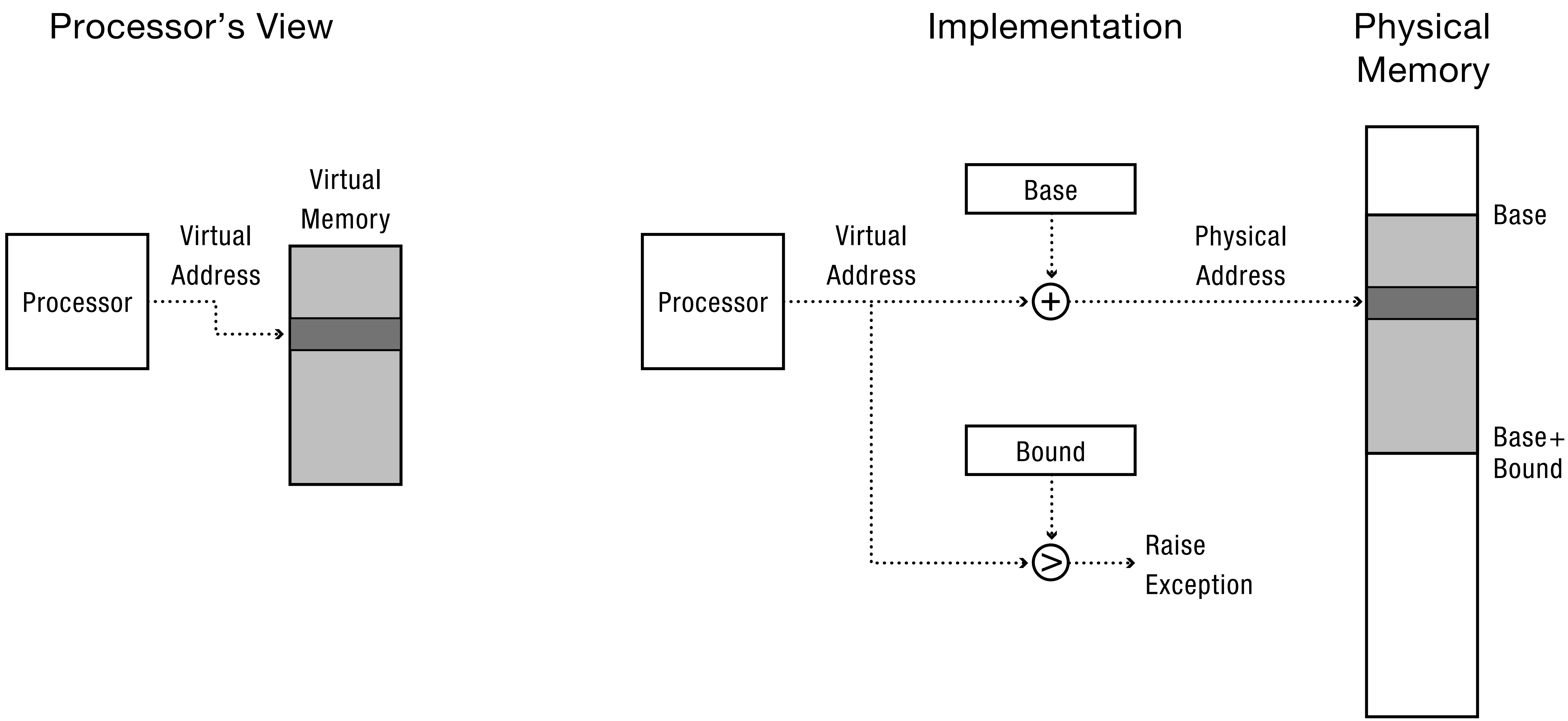

Implementation
Physical Memory
Processor
Virtual Address
Base
Physical Address
Bound
Raise Exception
Base
Base+ Bound

Base & Bound



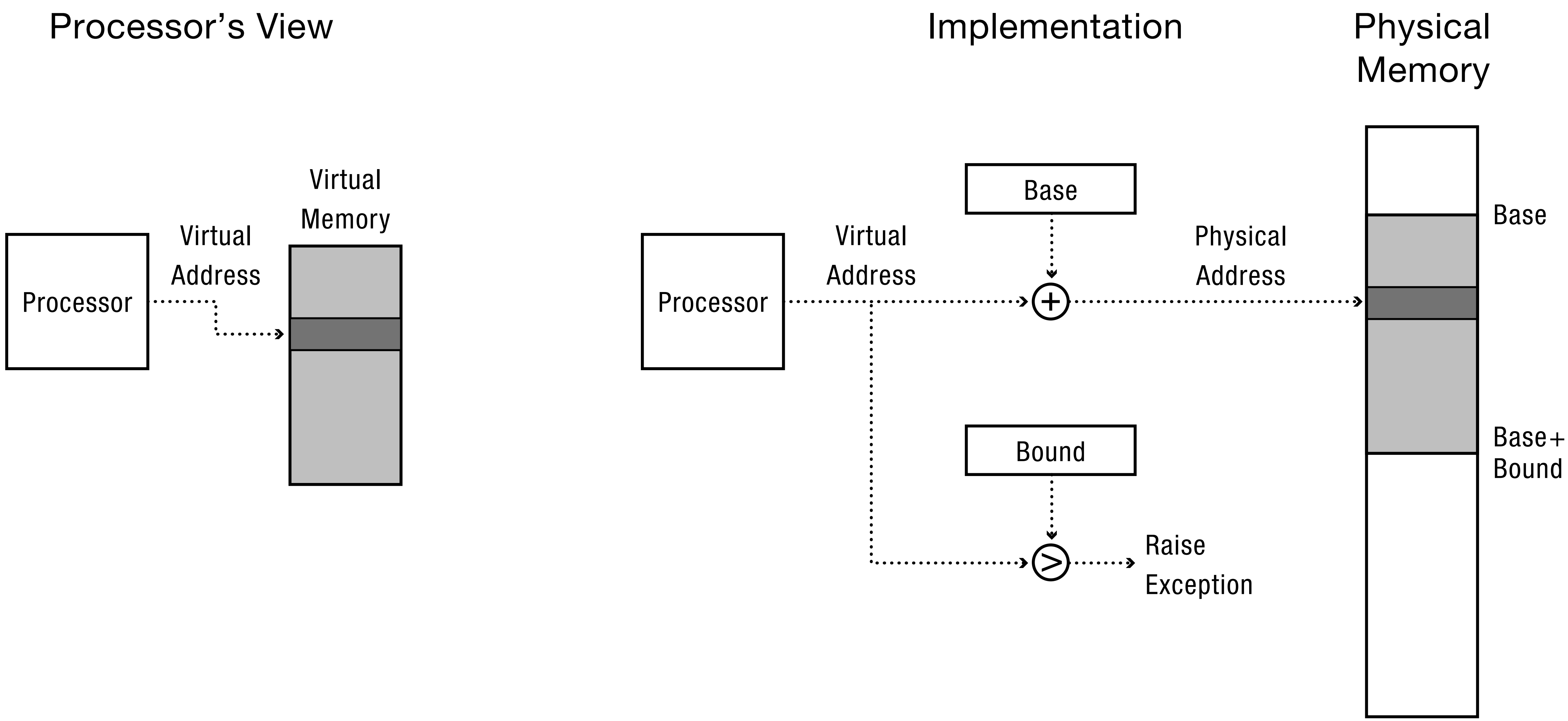
- A program cannot see past its own memory!

Base & Bound



- A program cannot see past its own memory!
- No more need for relocating loader!

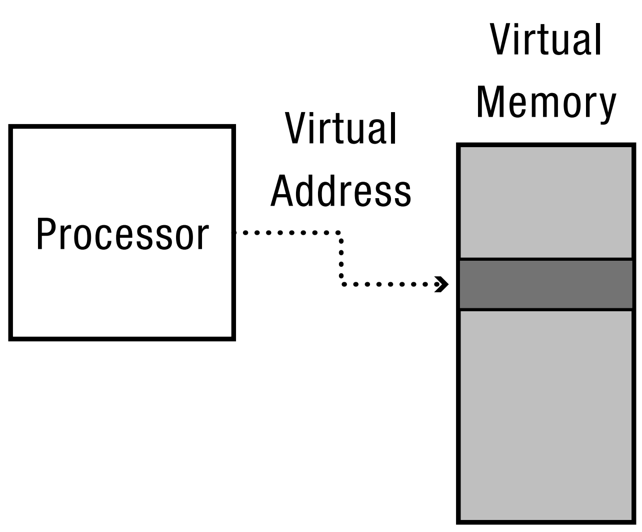
Base & Bound



- A program cannot see past its own memory!
- No more need for relocating loader!
- Context switch: just change base & bounds

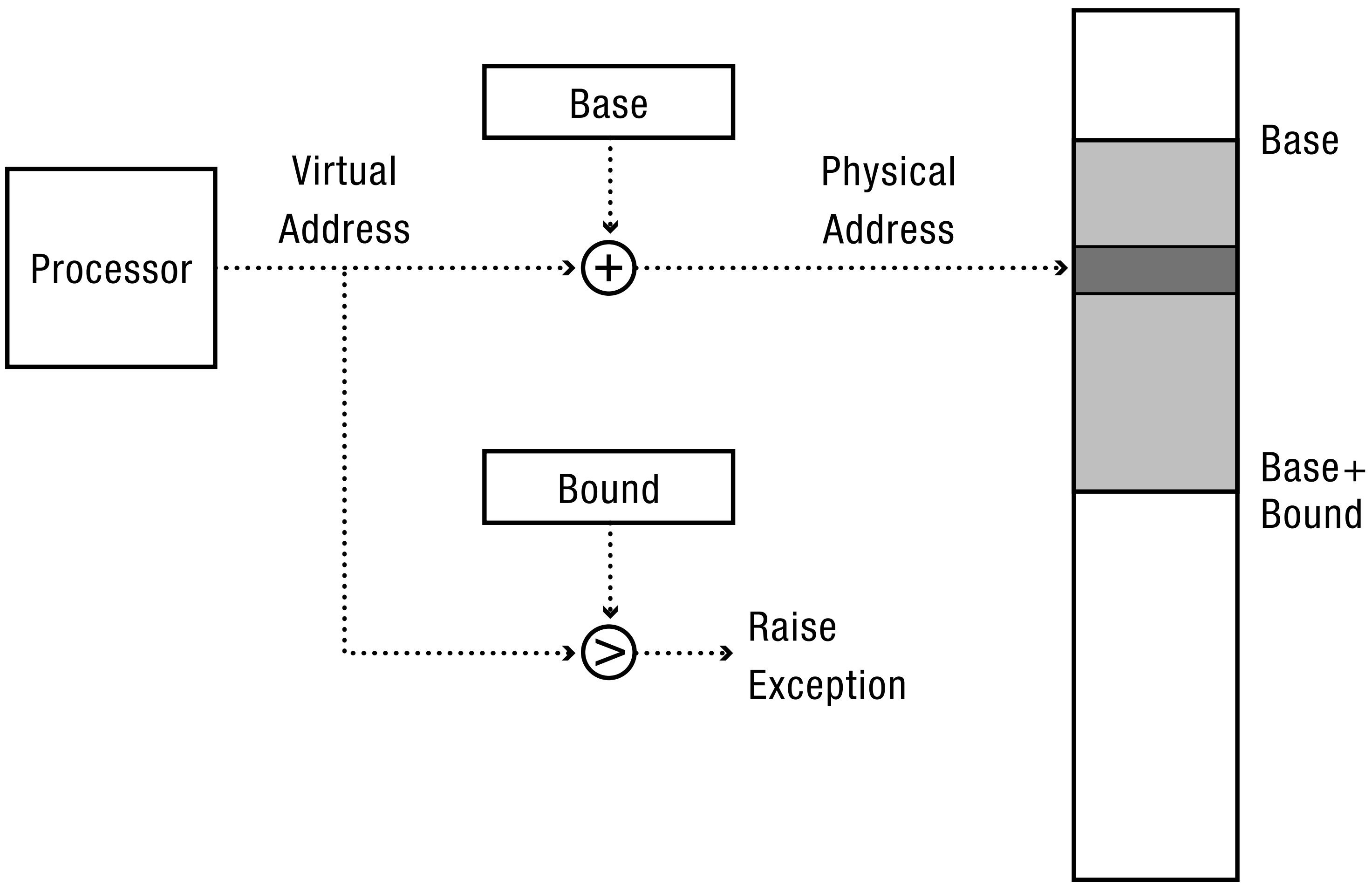
Base & Bound

Processor's View

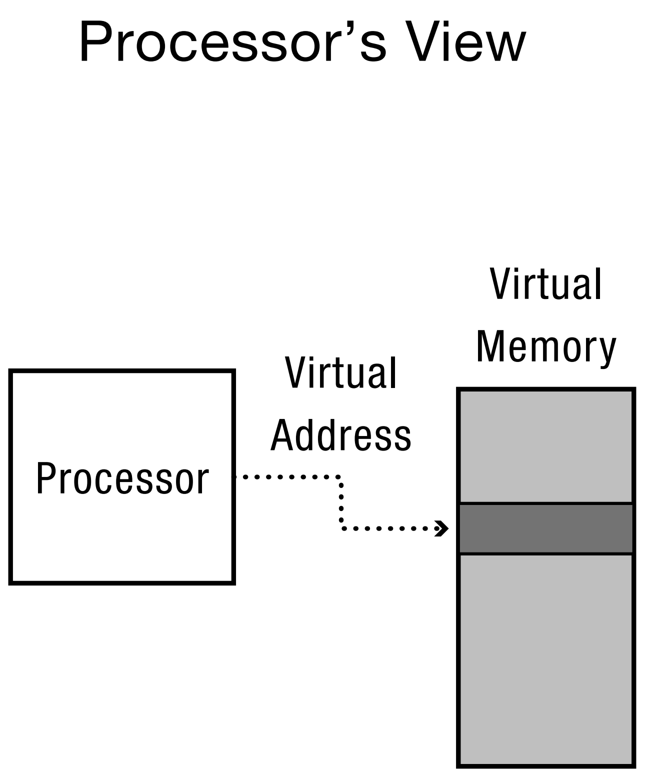


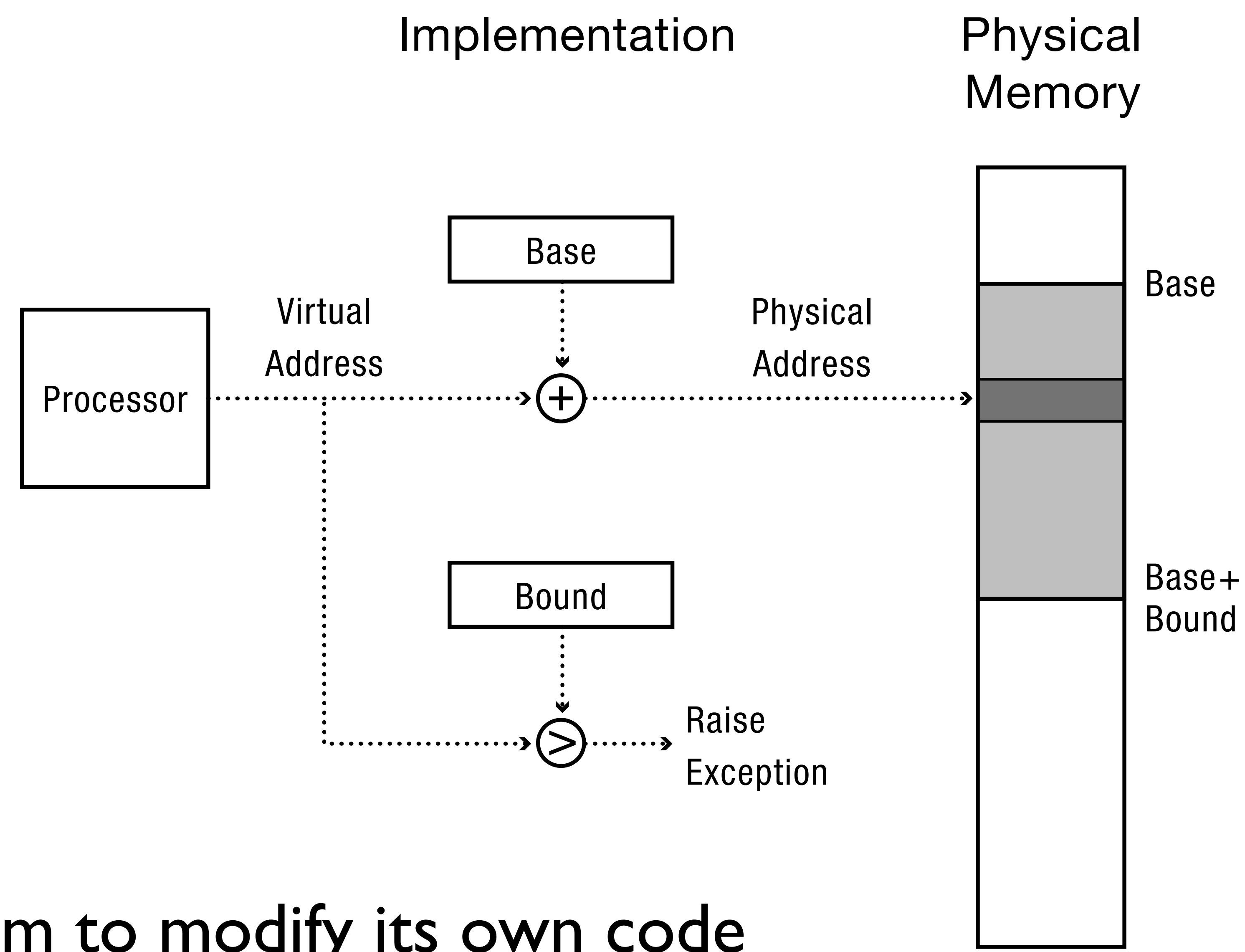
Implementation

Physical Memory



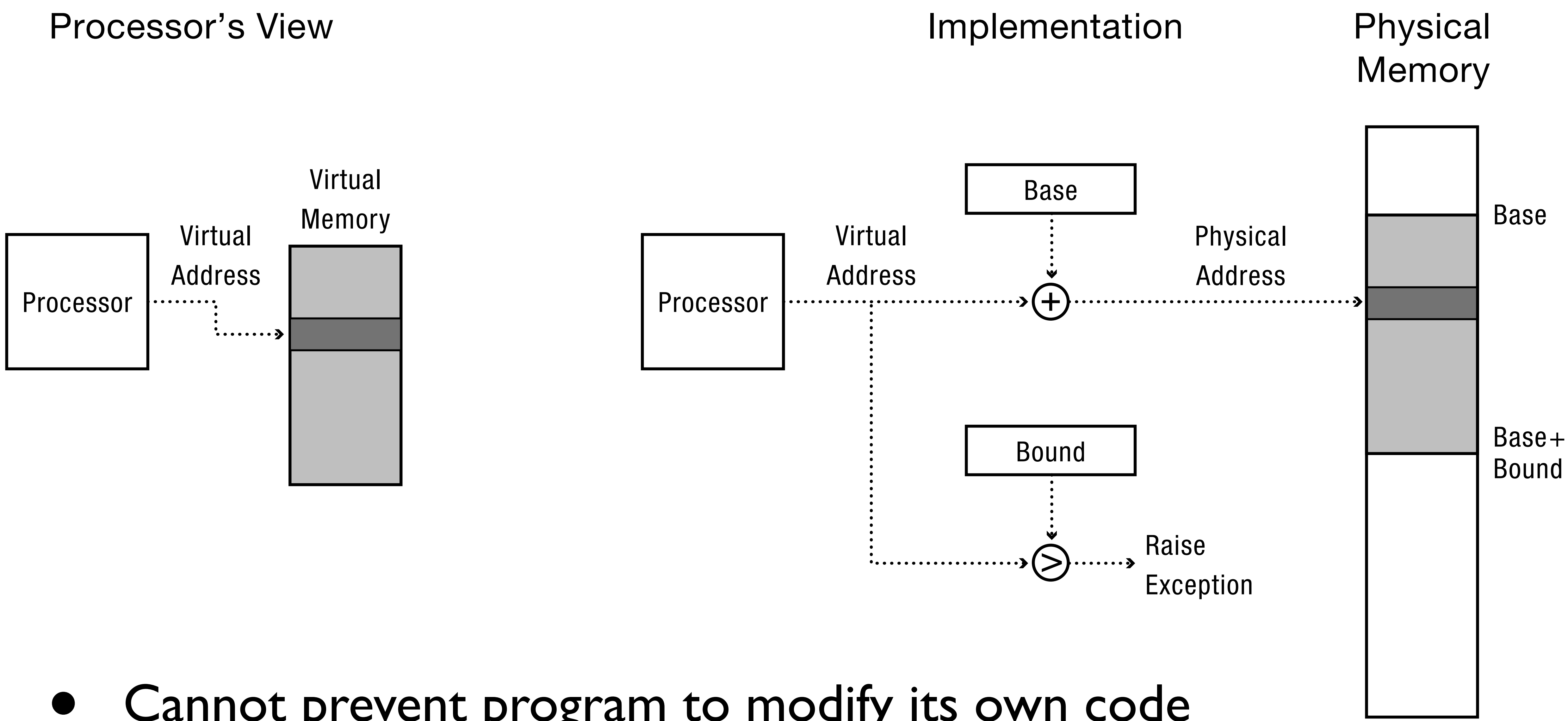
Base & Bound





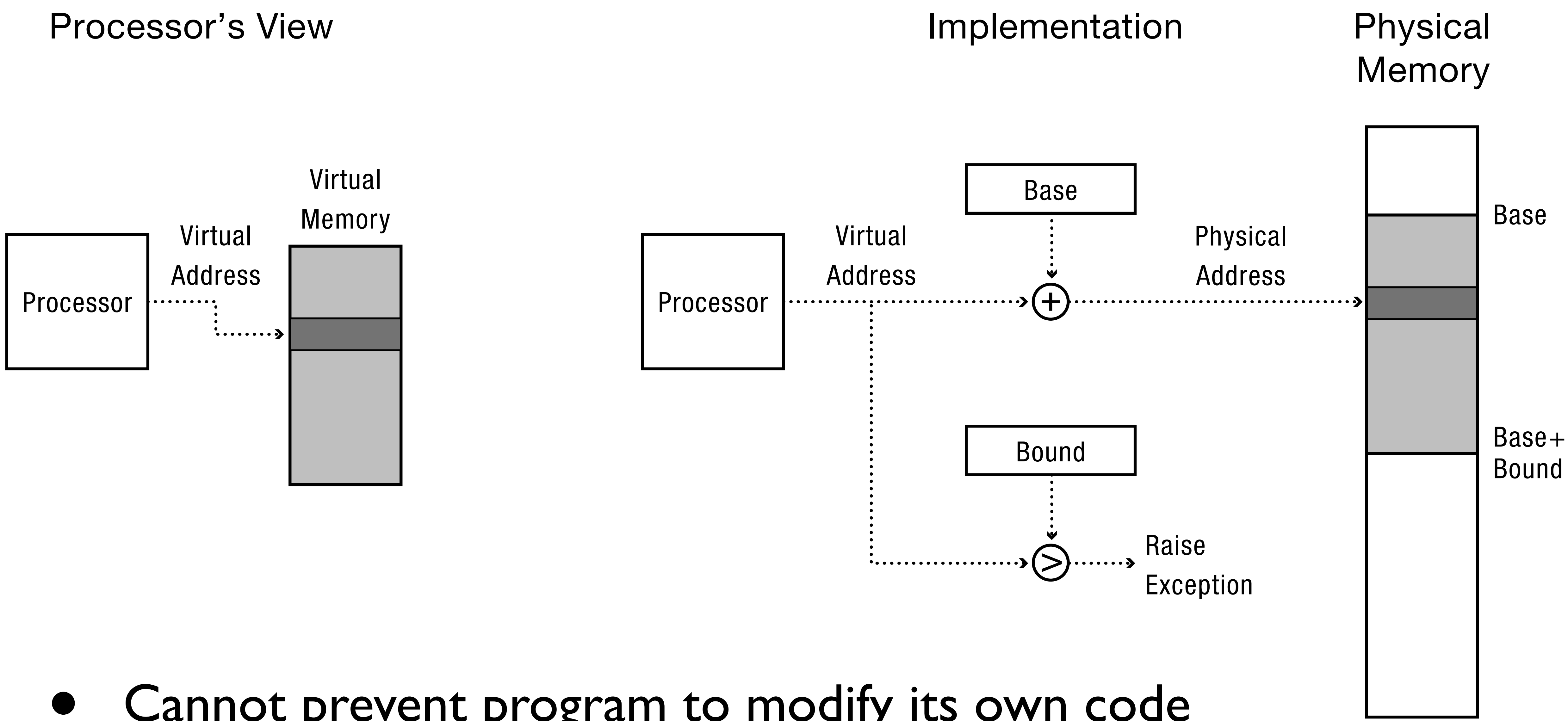
- Cannot prevent program to modify its own code

Base & Bound



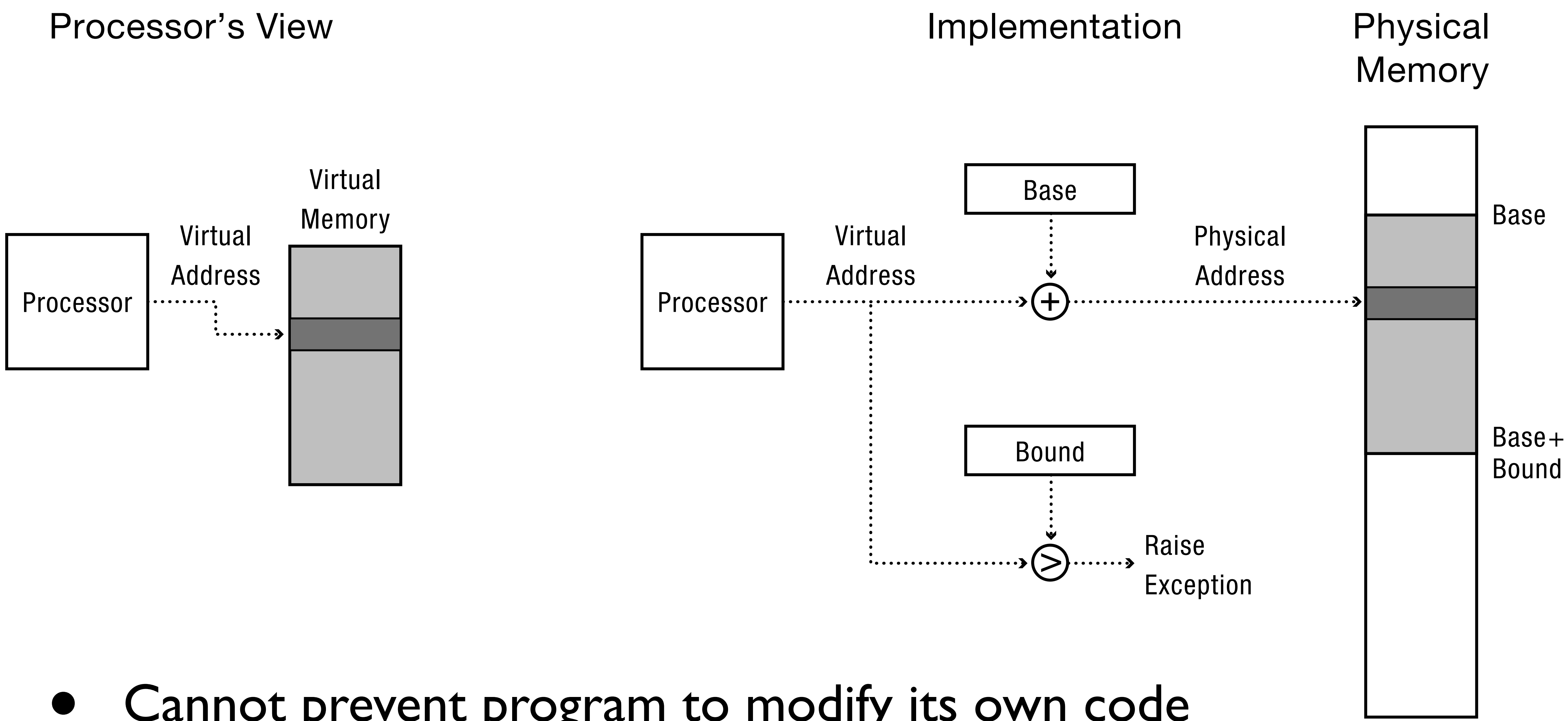
- Cannot prevent program to modify its own code
- No sharing!

Base & Bound



- Cannot prevent program to modify its own code
- No sharing!
- No expandable heap/stack

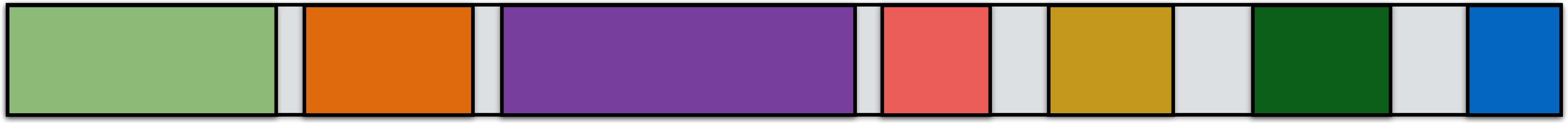
Base & Bound



- Cannot prevent program to modify its own code
- No sharing!
- No expandable heap/stack
- External fragmentation

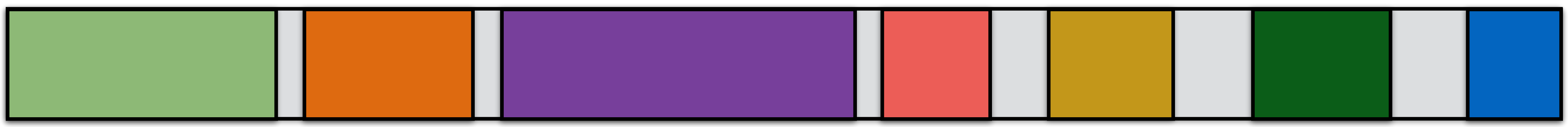
External Fragmentation

Suppose your memory looks like this

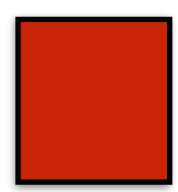


External Fragmentation

Suppose your memory looks like this

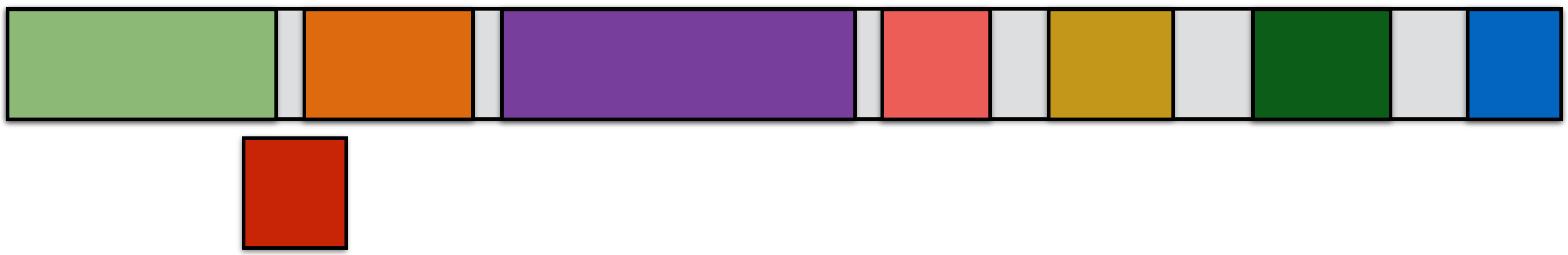


Allocate this



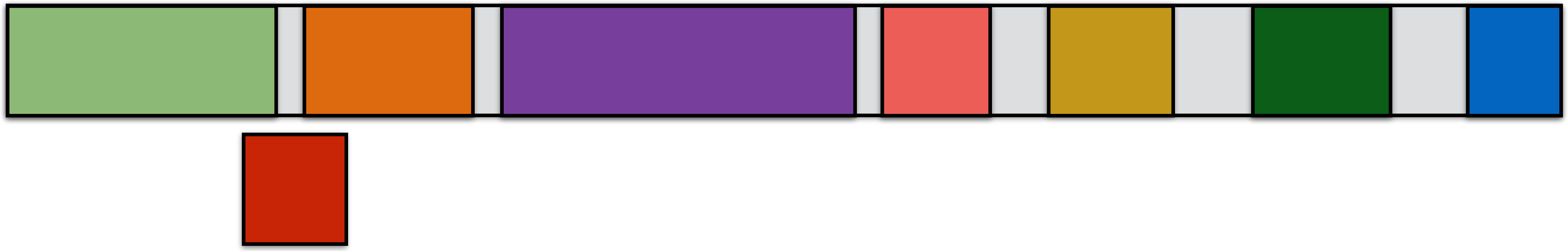
External Fragmentation

Suppose your memory looks like this



External Fragmentation

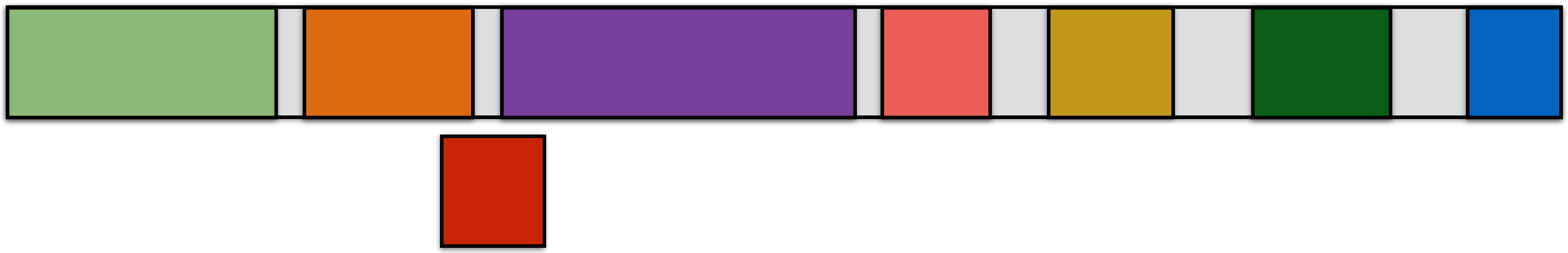
Suppose your memory looks like this



Nope

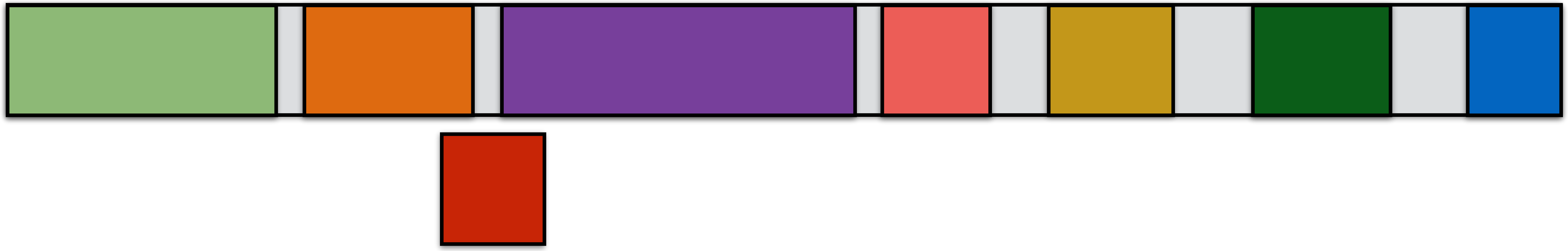
External Fragmentation

Suppose your memory looks like this



External Fragmentation

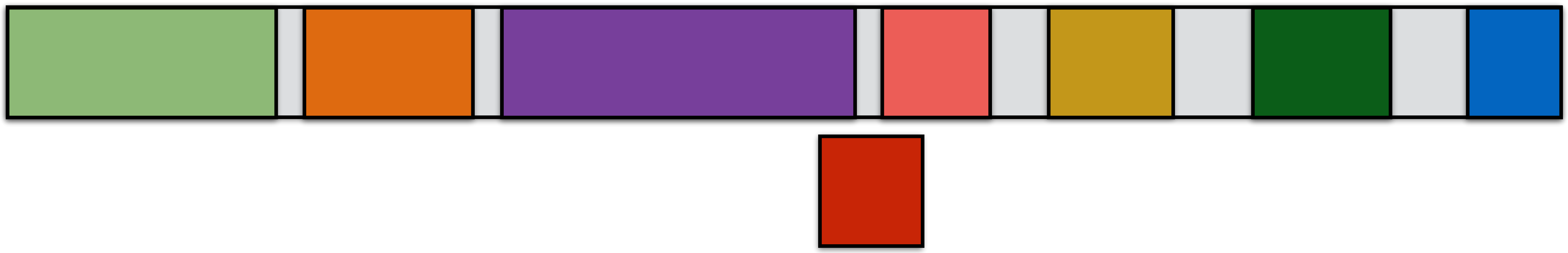
Suppose your memory looks like this



Nope

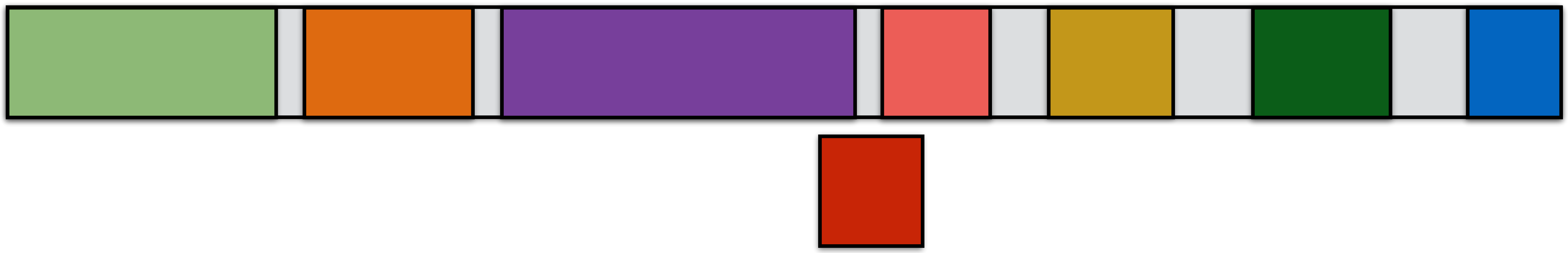
External Fragmentation

Suppose your memory looks like this



External Fragmentation

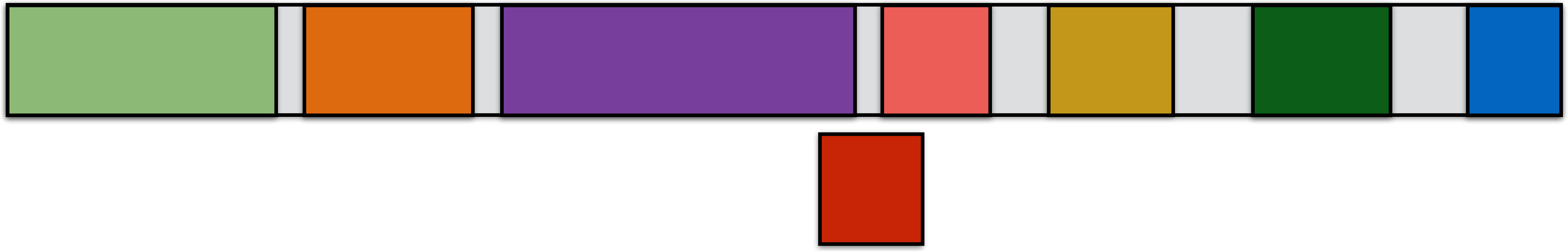
Suppose your memory looks like this



Nope… you get the idea

External Fragmentation

Suppose your memory looks like this

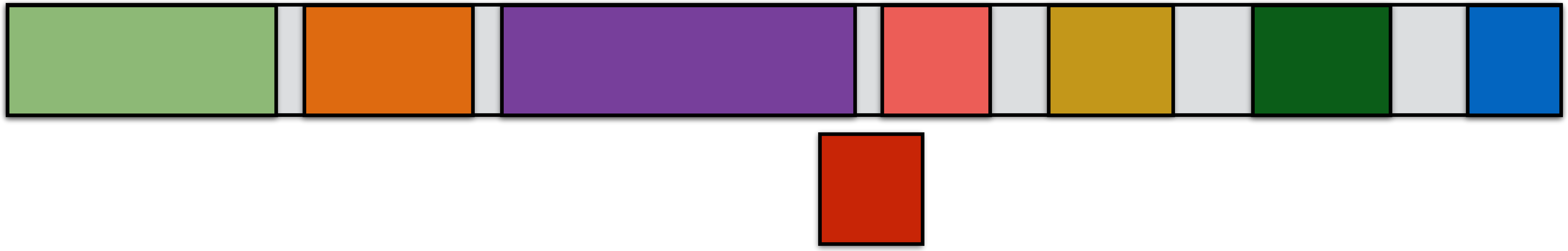


Nope… you get the idea

You suffer from *external fragmentation:*

External Fragmentation

Suppose your memory looks like this



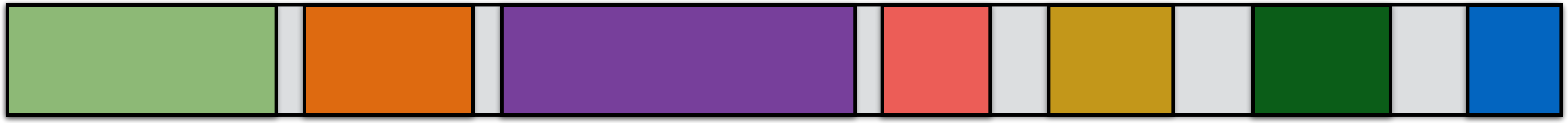
Nope… you get the idea

You suffer from *external fragmentation:*

You have enough space available, but not contiguously!

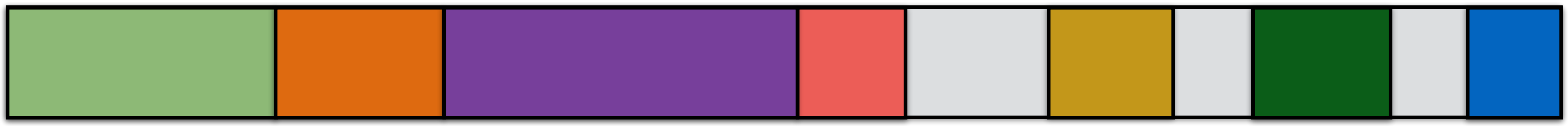
External Fragmentation

Defragment memory is costly…



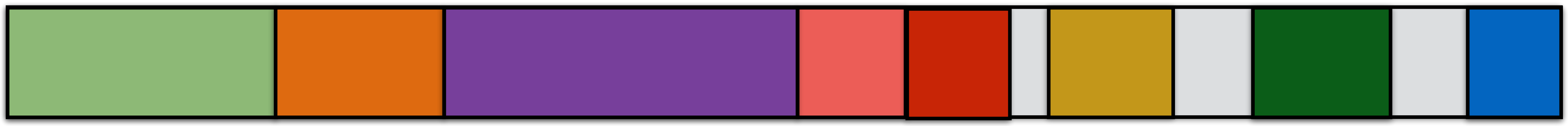
External Fragmentation

Defragment memory is costly…



External Fragmentation

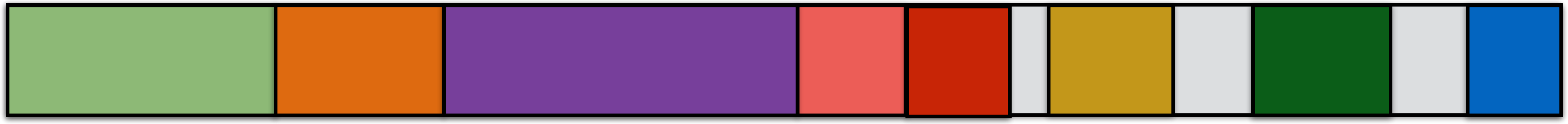
Defragment memory is costly…



Aha!

External Fragmentation

Defragment memory is costly…



Aha!

Much better to *avoid* ever having to do that!

Approach II: Segmentation

Segmentation

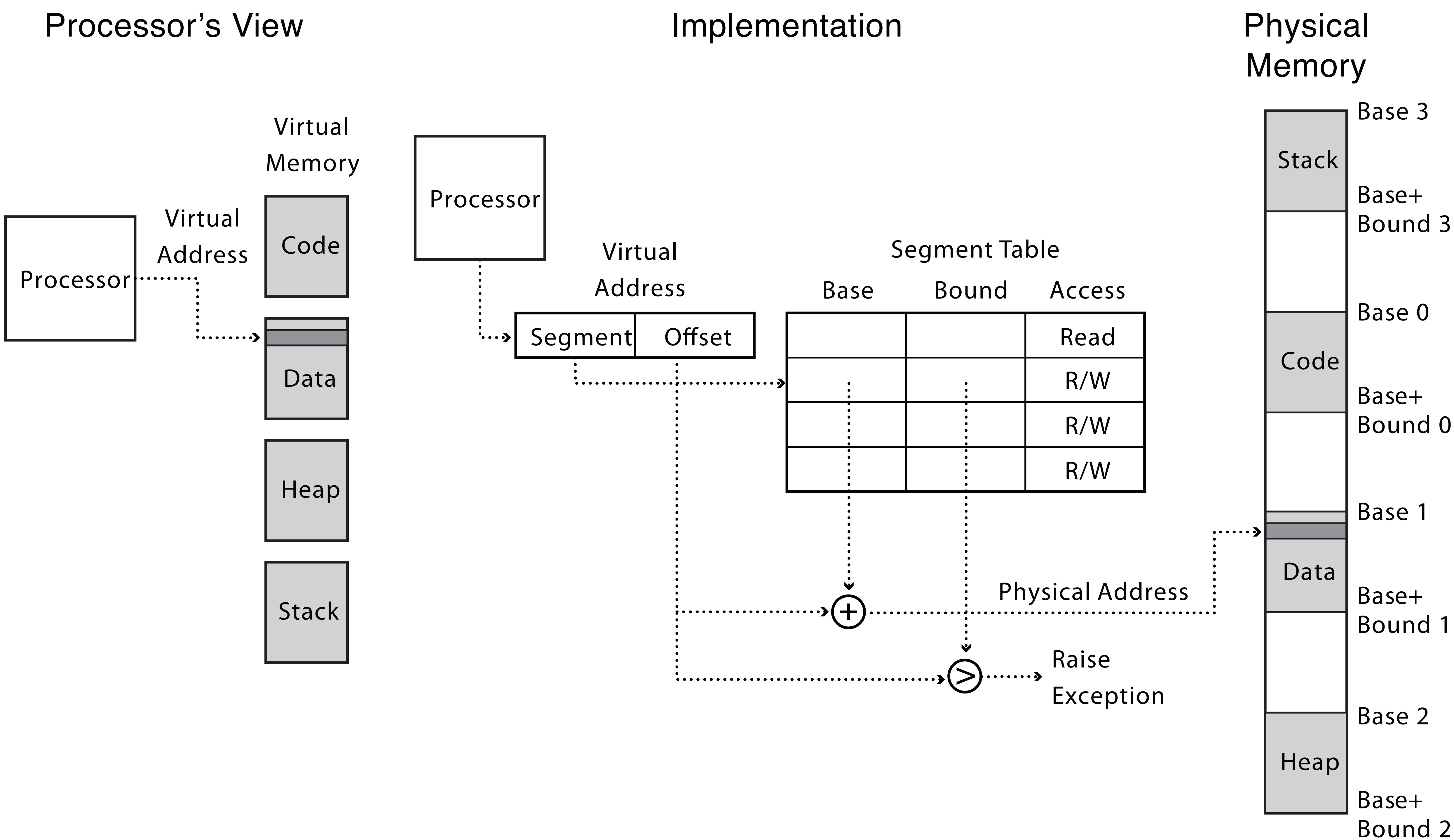
Segmentation

- Multiple base & bound registers

Segmentation

- Multiple base & bound registers
- On Intel x86:
  - Code (CS)
  - Data (DS)
  - Stack (SS)
  - Extra segment (ES)
  - Two general-purpose segments (FS, GS)

Segmentation



Segmentation

Segmentation

- If your program accesses a wild pointer

Segmentation

- If your program accesses a wild pointer
  - *Segmentation fault*

Segmentation

- If your program accesses a wild pointer
  - *Segmentation fault*
- Context Switch:

Segmentation

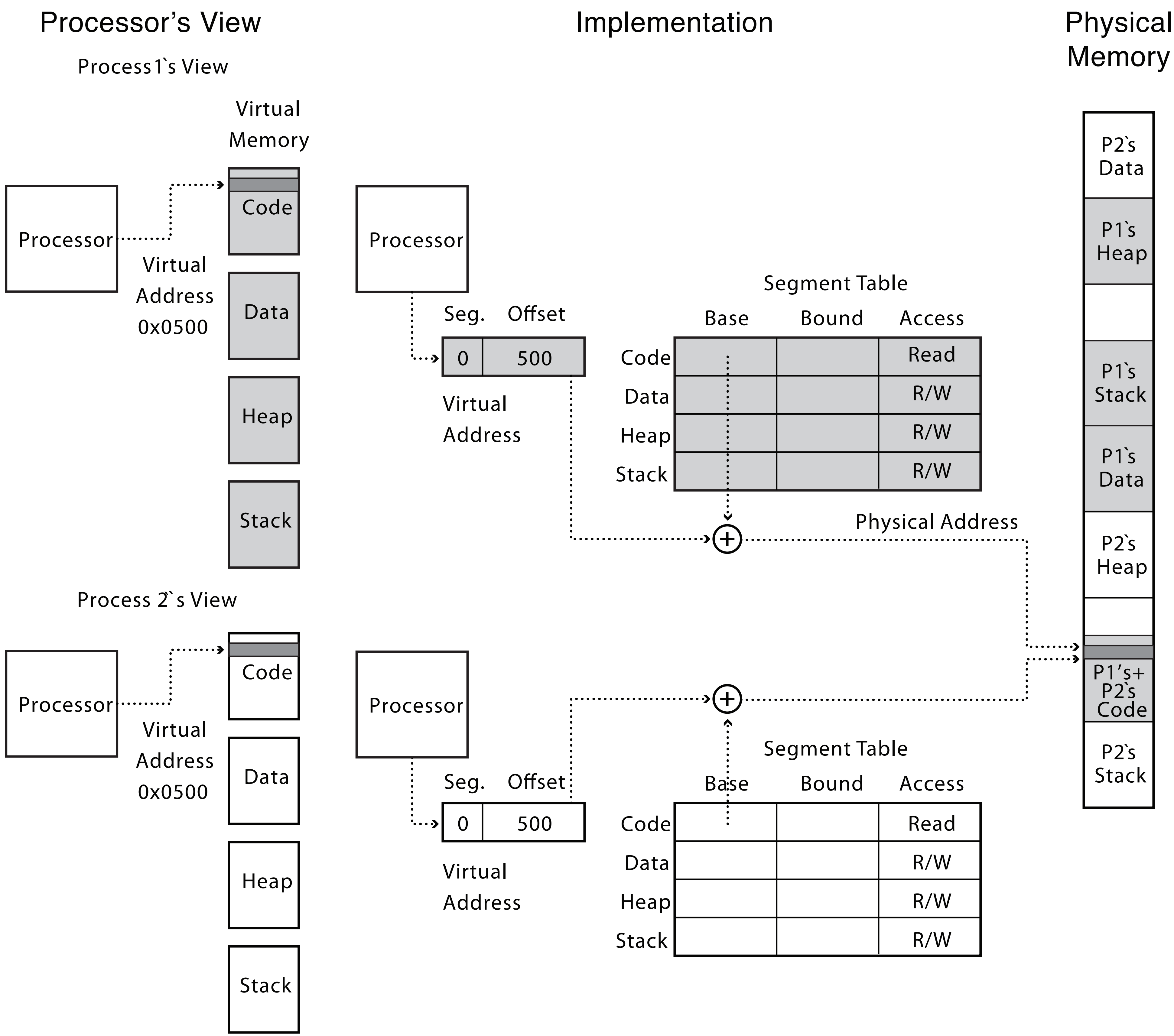
- If your program accesses a wild pointer
  - *Segmentation fault*
- Context Switch:
  - Save the segment table contents

Segmentation

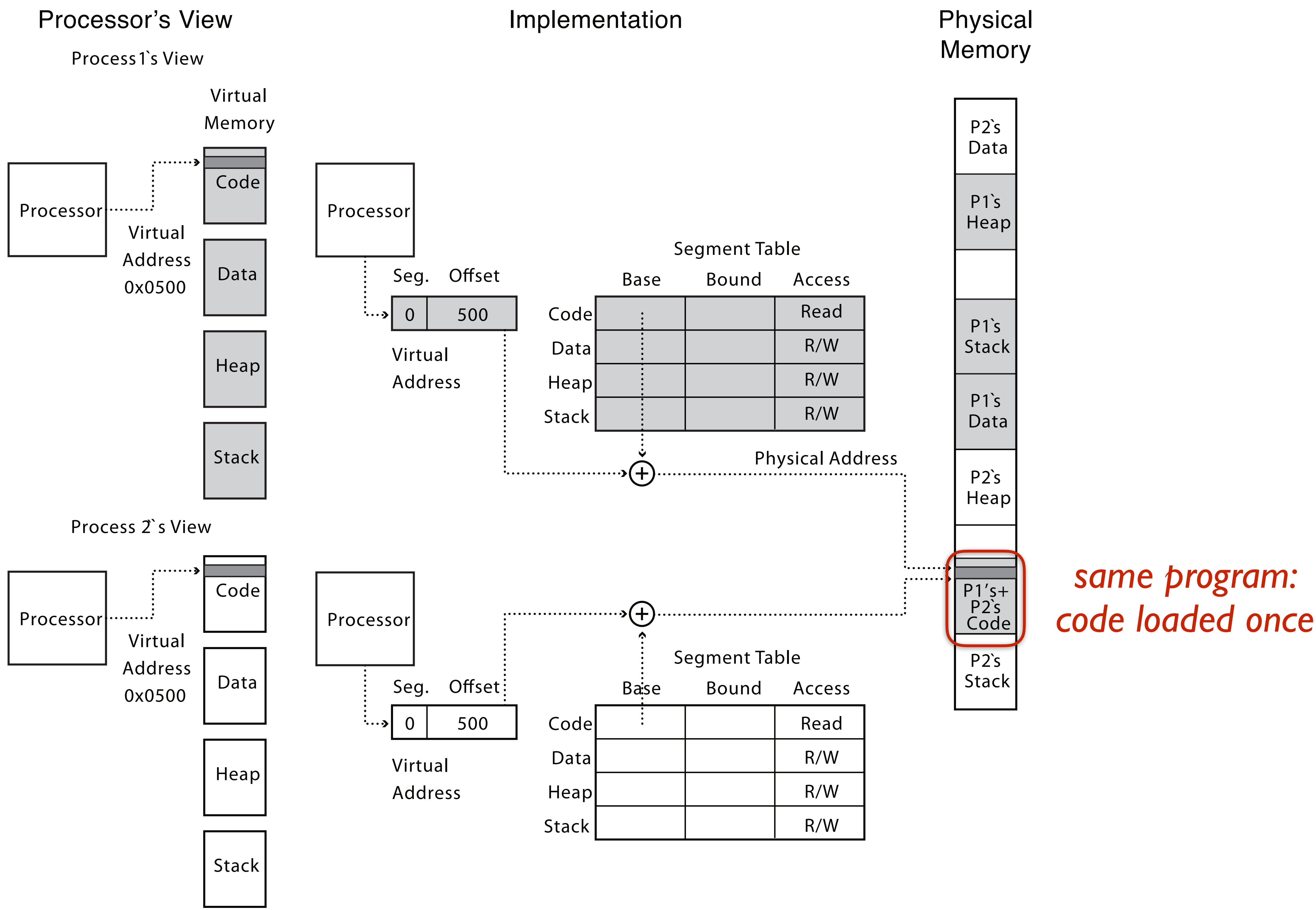
- If your program accesses a wild pointer
  - *Segmentation fault*
- Context Switch:
  - Save the segment table contents
- Sharing is allowed!

* source: Wikipedia user Sander van der Wel

Segment Sharing



Segment Sharing



Segment Sharing

Segment Sharing

- `fork()`: share all segments with the new process
  - Data section in read-only mode, please
  - That costs almost nothing, but…

Segment Sharing

- `fork()`: share all segments with the new process
  - Data section in read-only mode, please
  - That costs almost nothing, but…
- As soon as the new process changes the data
  - A fresh copy of parent's data segment is made: *copy on write*

Segment Sharing

- `fork()`: share all segments with the new process
  - Data section in read-only mode, please
  - That costs almost nothing, but…
- As soon as the new process changes the data
  - A fresh copy of parent's data segment is made: *copy on write*
- As soon as new process calls `exec()`
  - Newly initialized code and data sections are made

Segment Sharing

- `fork()`: share all segments with the new process
  - Data section in read-only mode, please
  - That costs almost nothing, but…
- As soon as the new process changes the data
  - A fresh copy of parent's data segment is made: *copy on write*
- As soon as new process calls `exec()`
  - Newly initialized code and data sections are made
- This approach is called lazy*, opportunistic**

Segment Sharing

- `fork()`: share all segments with the new process
  - Data section in read-only mode, please
  - That costs almost nothing, but…
- As soon as the new process changes the data
  - A fresh copy of parent's data segment is made: *copy on write*
- As soon as new process calls `exec()`
  - Newly initialized code and data sections are made
- This approach is called lazy*, opportunistic**

* laziness paying off in computing

Segment Sharing

- `fork()`: share all segments with the new process
  - Data section in read-only mode, please
  - That costs almost nothing, but…
- As soon as the new process changes the data
  - A fresh copy of parent's data segment is made: *copy on write*
- As soon as new process calls `exec()`
  - Newly initialized code and data sections are made
- This approach is called lazy*, opportunistic**

* laziness paying off in computing

** opportunistic understood in a good sense.

Pros and Cons

Pros and Cons

- Solves most problems from Base & Bound:

Pros and Cons

- Solves most problems from Base & Bound:
  - Read-only code sections (no program mutation)

Pros and Cons

- Solves most problems from Base & Bound:
  - Read-only code sections (no program mutation)
  - Can share information across processes
    - Code section of the same programs
    - Libraries loaded in special segments

Pros and Cons

- Solves most problems from Base & Bound:
  - Read-only code sections (no program mutation)
  - Can share information across processes
    - Code section of the same programs
    - Libraries loaded in special segments
  - Has expandable heap/stack

Pros and Cons

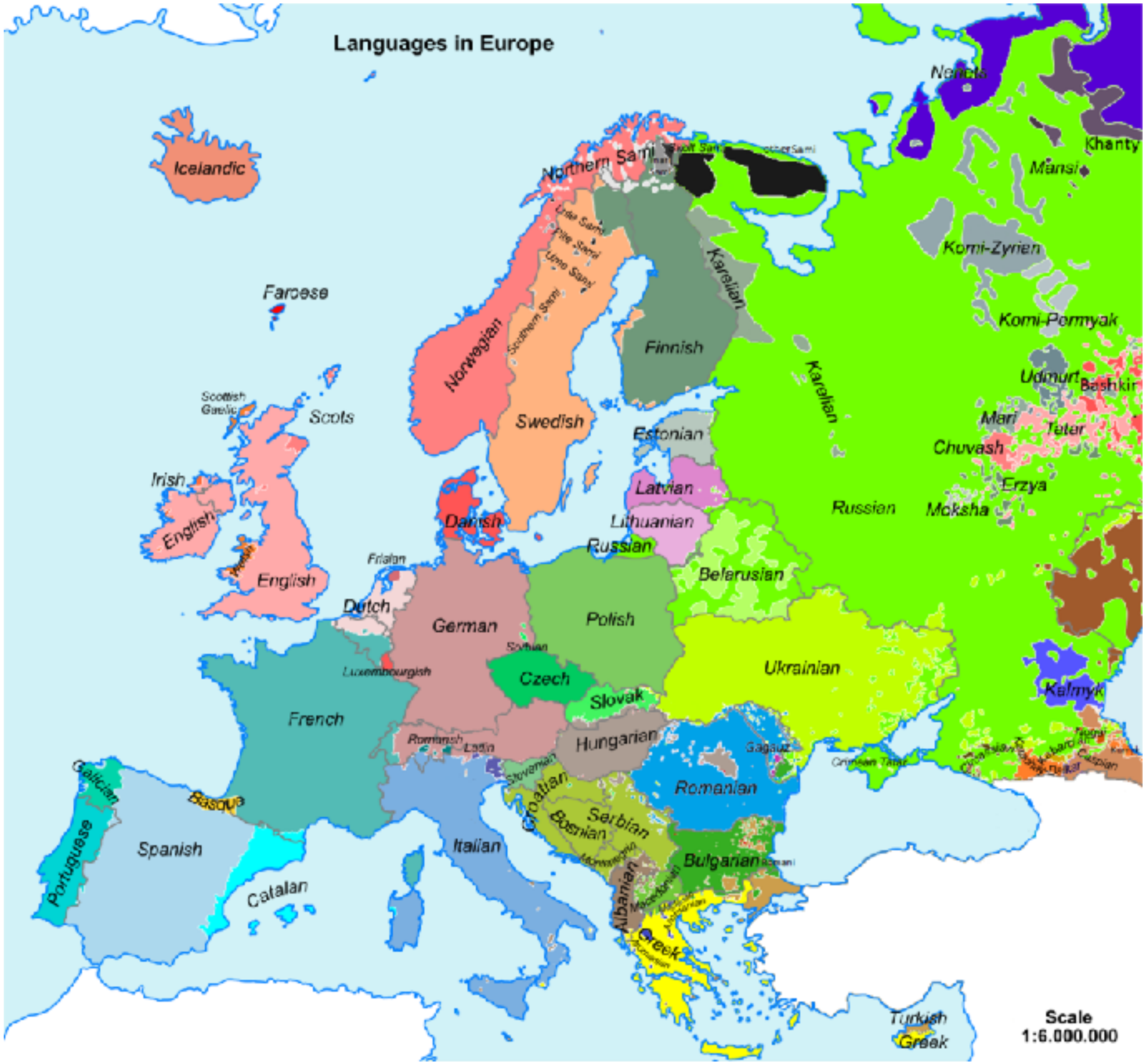
- Solves most problems from Base & Bound:
  - Read-only code sections (no program mutation)
  - Can share information across processes
    - Code section of the same programs
    - Libraries loaded in special segments
  - Has expandable heap/stack
    - Just leave gaps in the virtual segmented space

Pros and Cons

- Solves most problems from Base & Bound:
  - Read-only code sections (no program mutation)
  - Can share information across processes
    - Code section of the same programs
    - Libraries loaded in special segments
  - Has expandable heap/stack
    - Just leave gaps in the virtual segmented space

But…

External Fragmentation

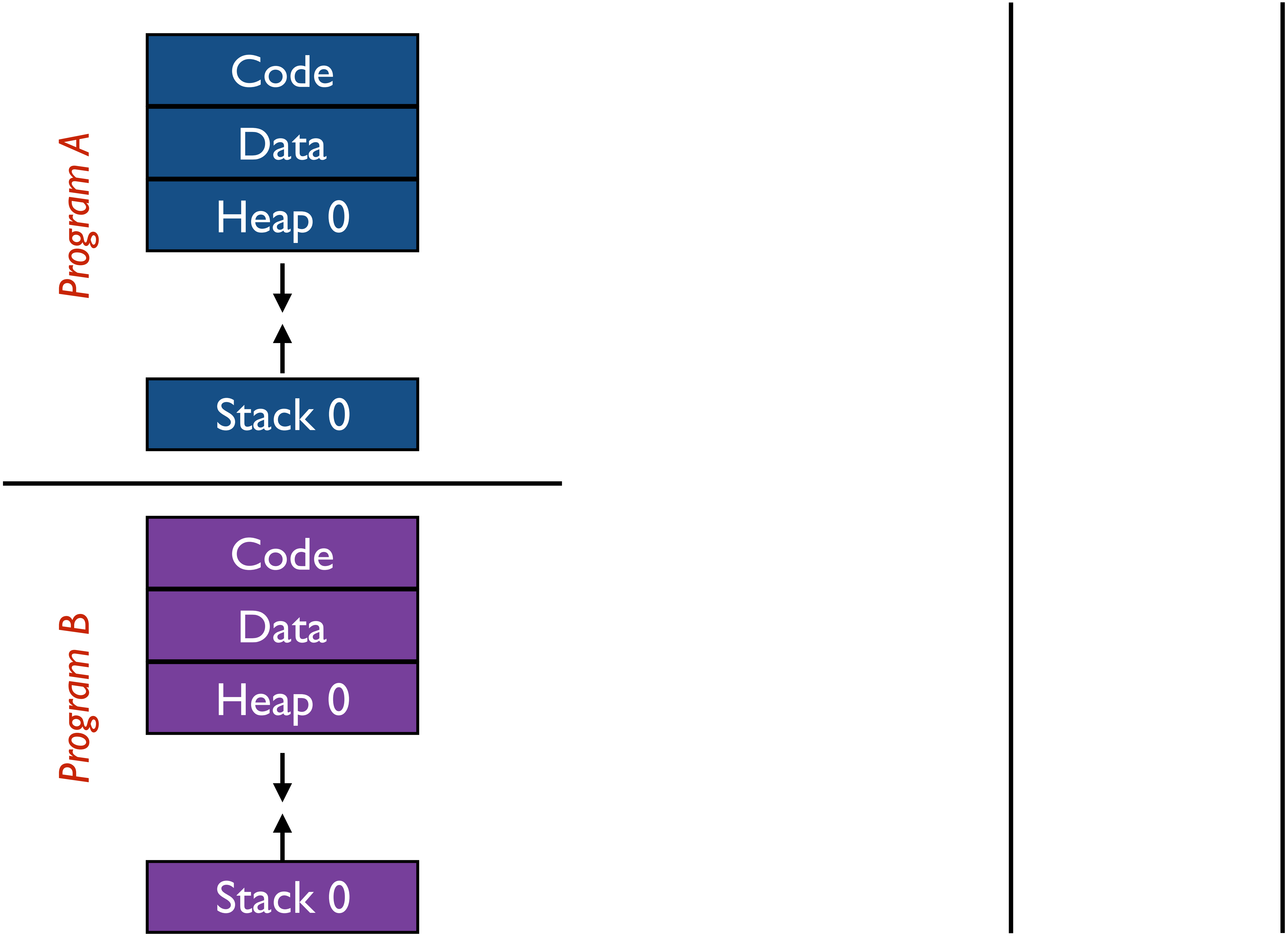


Approach III: Paging

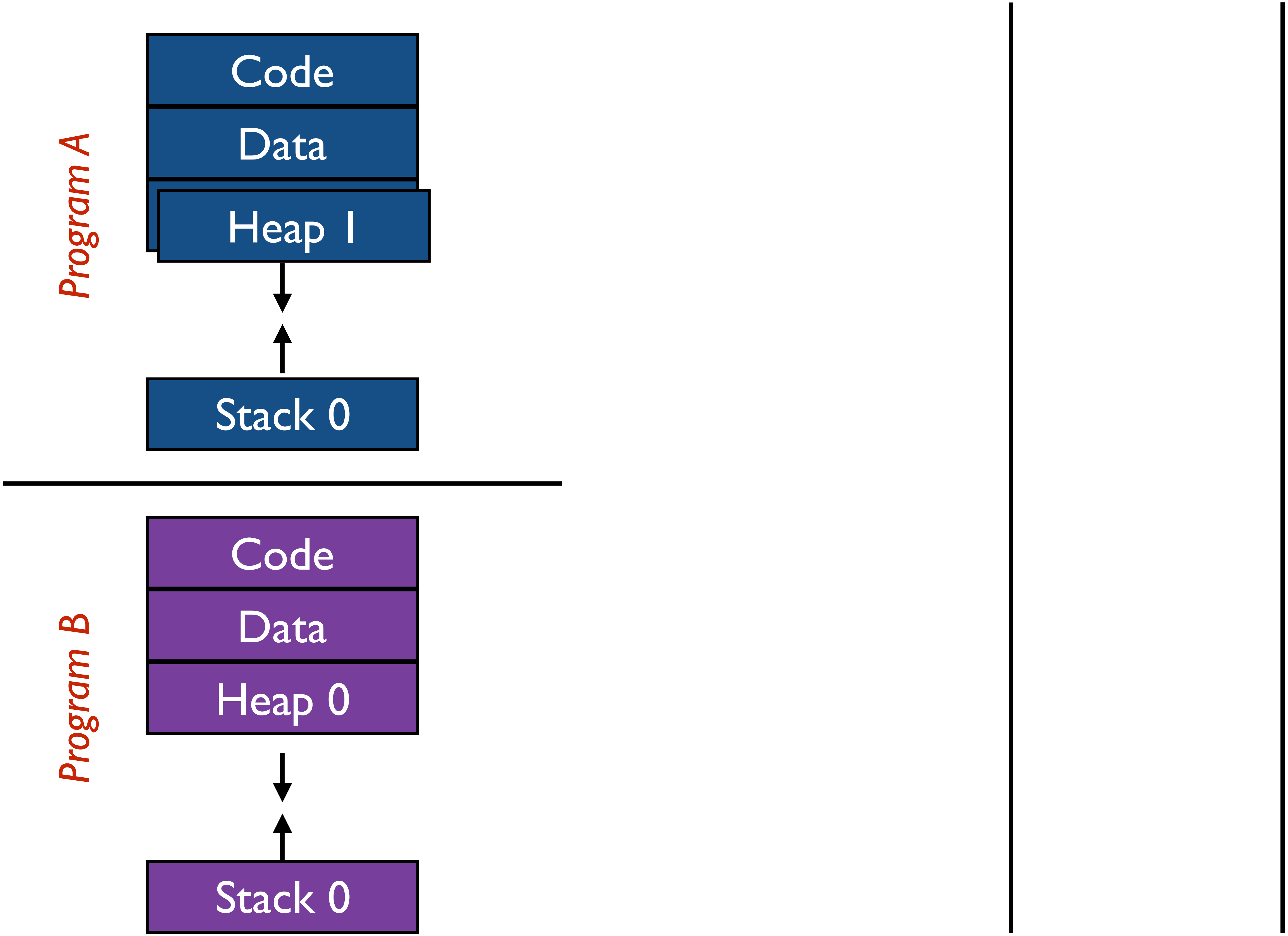
Approach III: Paging

deja vu

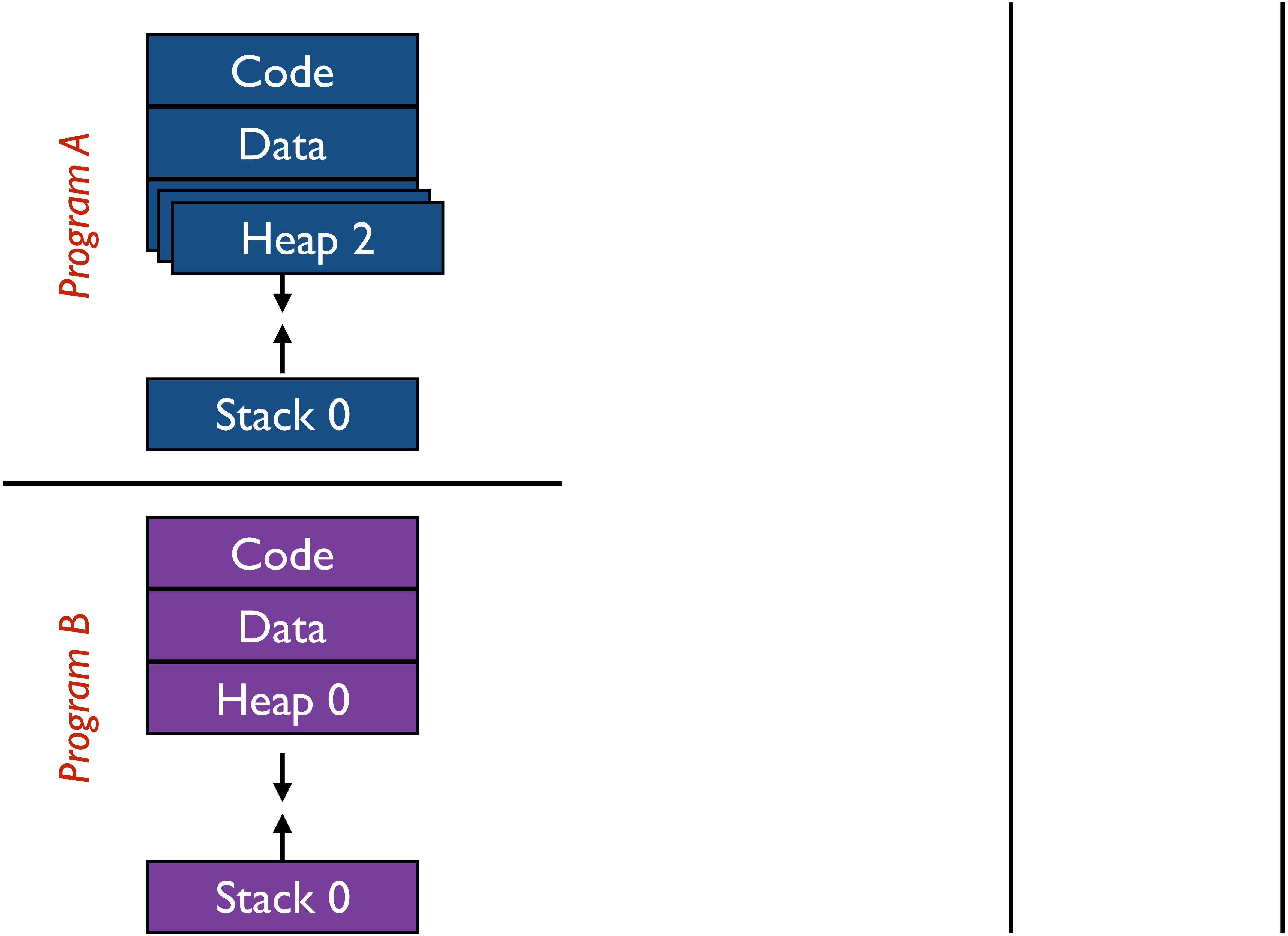
Paging


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0

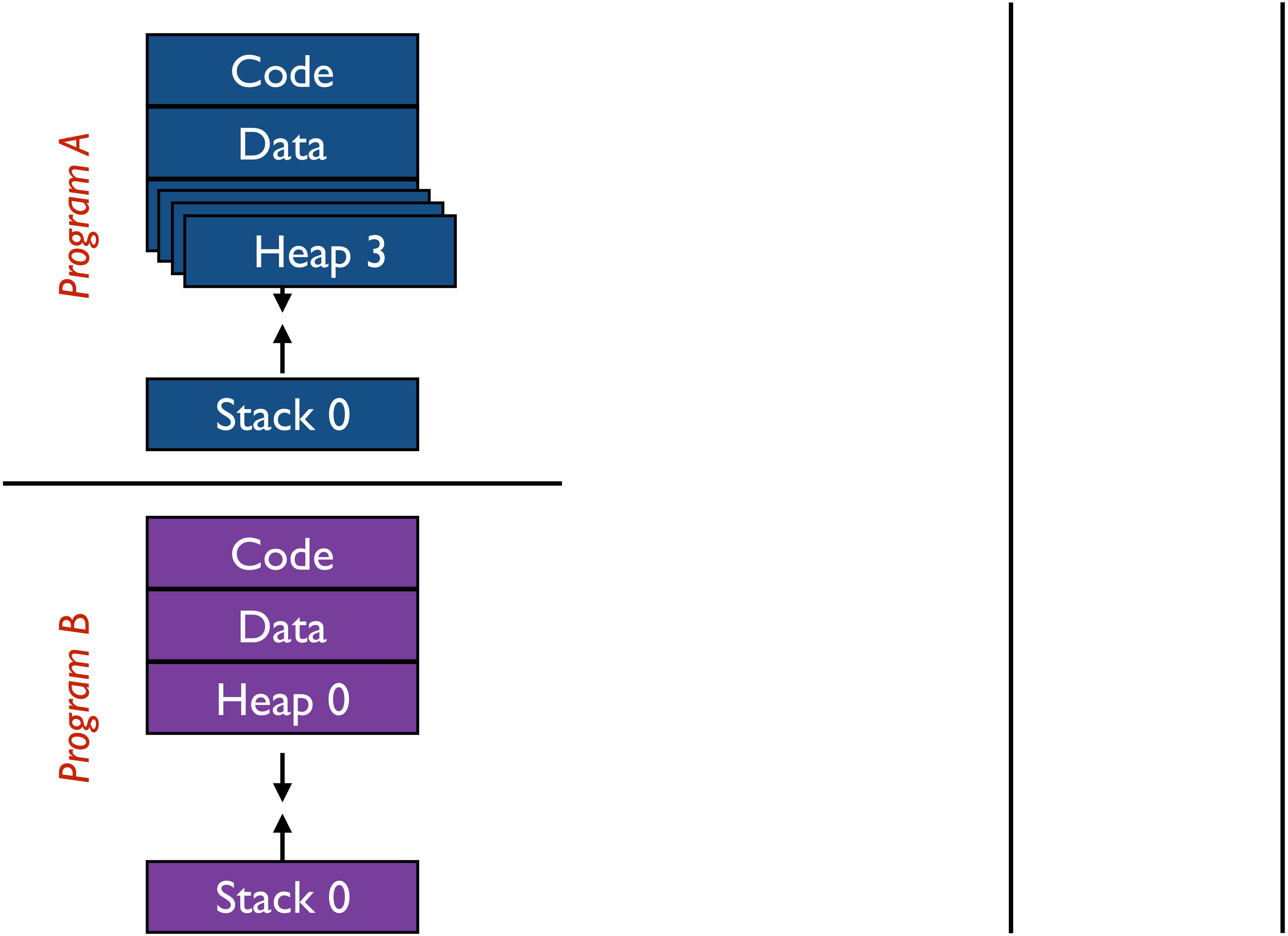
Paging


Program A
Code
Data
Heap 1
Stack 0
Program B
Code
Data
Heap 0
Stack 0

Paging

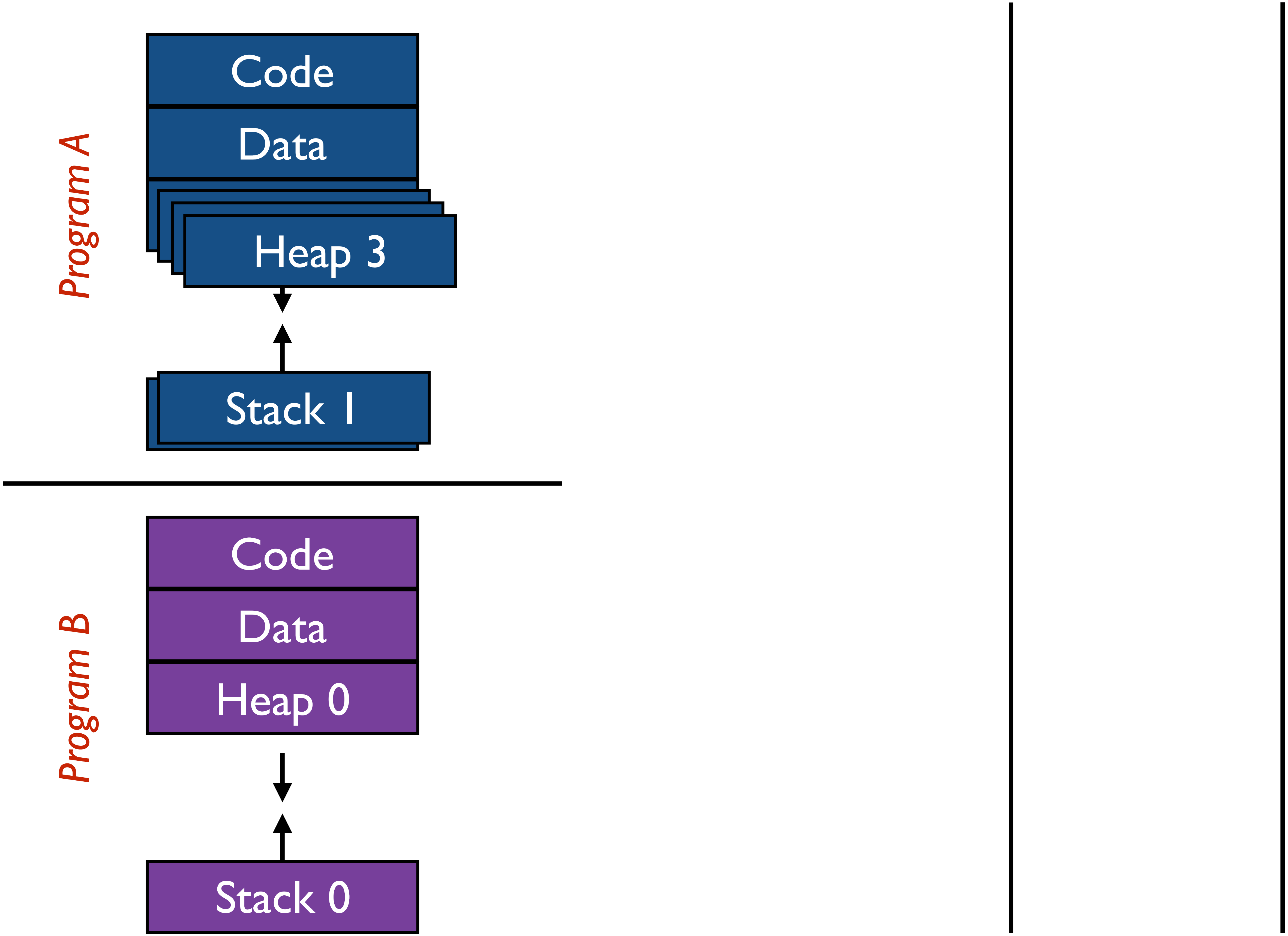

Program A
Code
Data
Heap 2
Stack 0
Program B
Code
Data
Heap 0
Stack 0

Paging

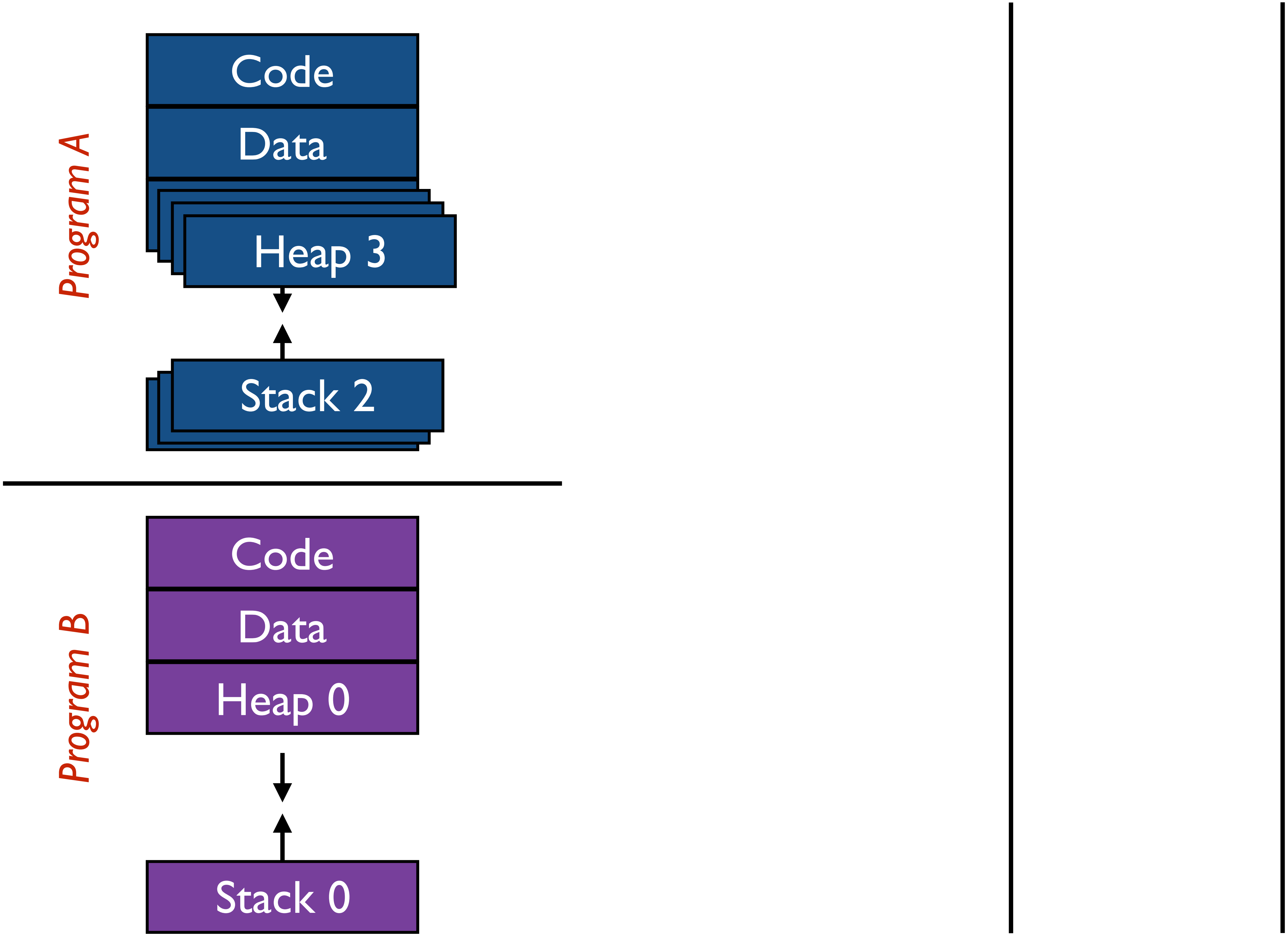


Program A
Code
Data
Heap 3
Stack 0
Program B
Code
Data
Heap 0
Stack 0

Paging

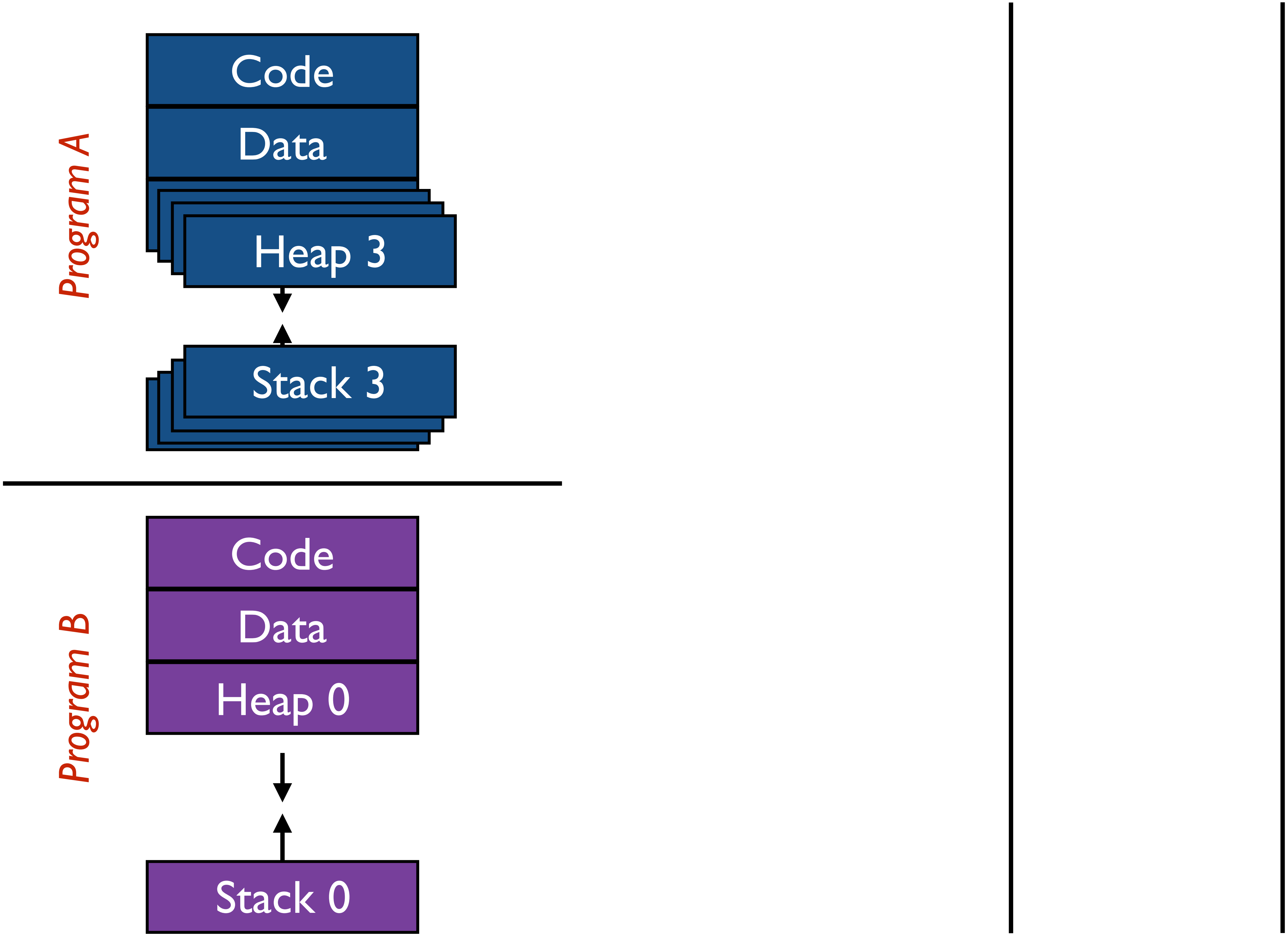

Program A
Code
Data
Heap 3
Stack 1
Program B
Code
Data
Heap 0
Stack 0

Paging

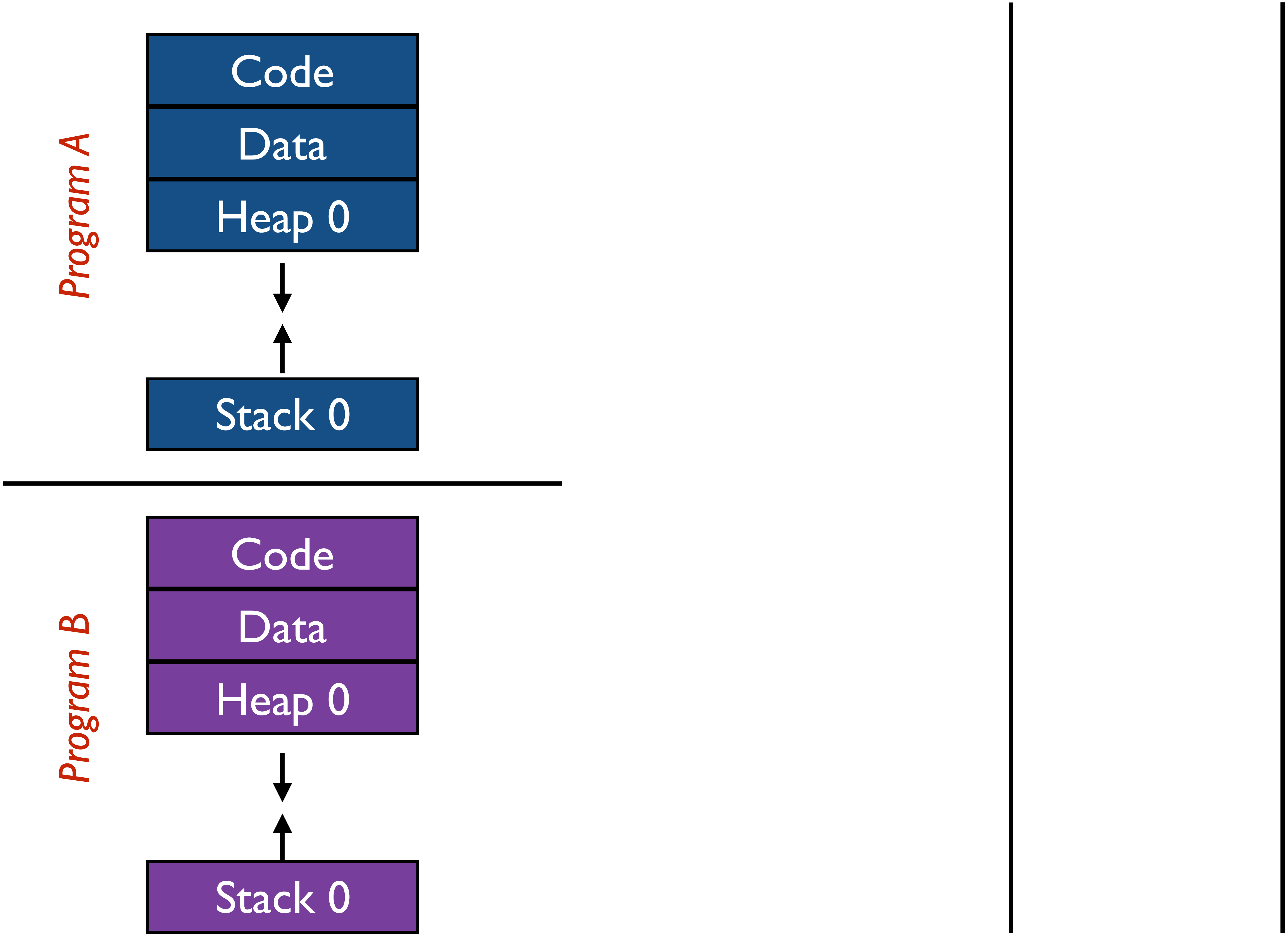


Program A
Code
Data
Heap 3
Stack 2
Program B
Code
Data
Heap 0
Stack 0

Paging

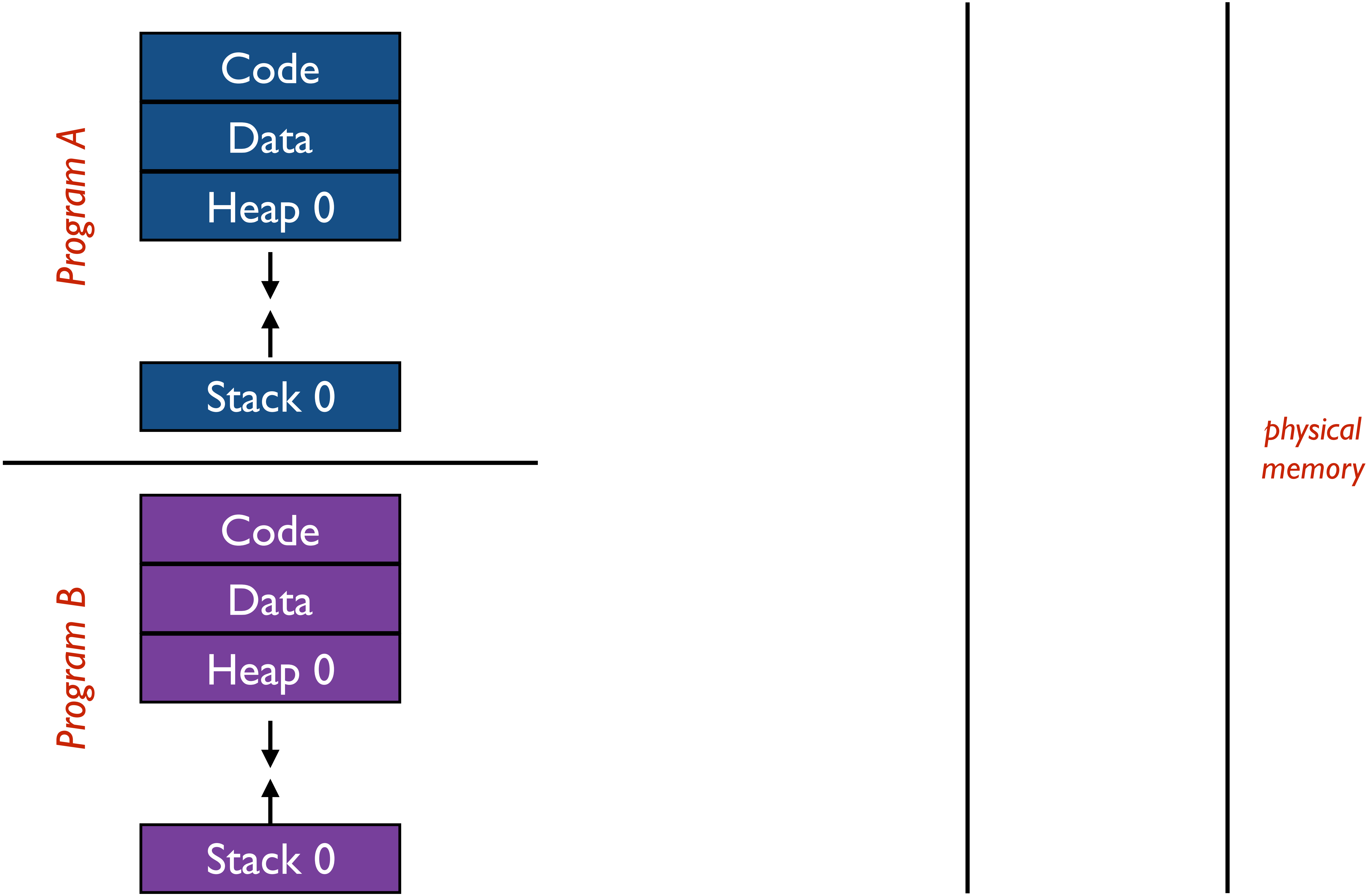

Program A
Code
Data
Heap 3
Stack 3
Program B
Code
Data
Heap 0
Stack 0

Paging



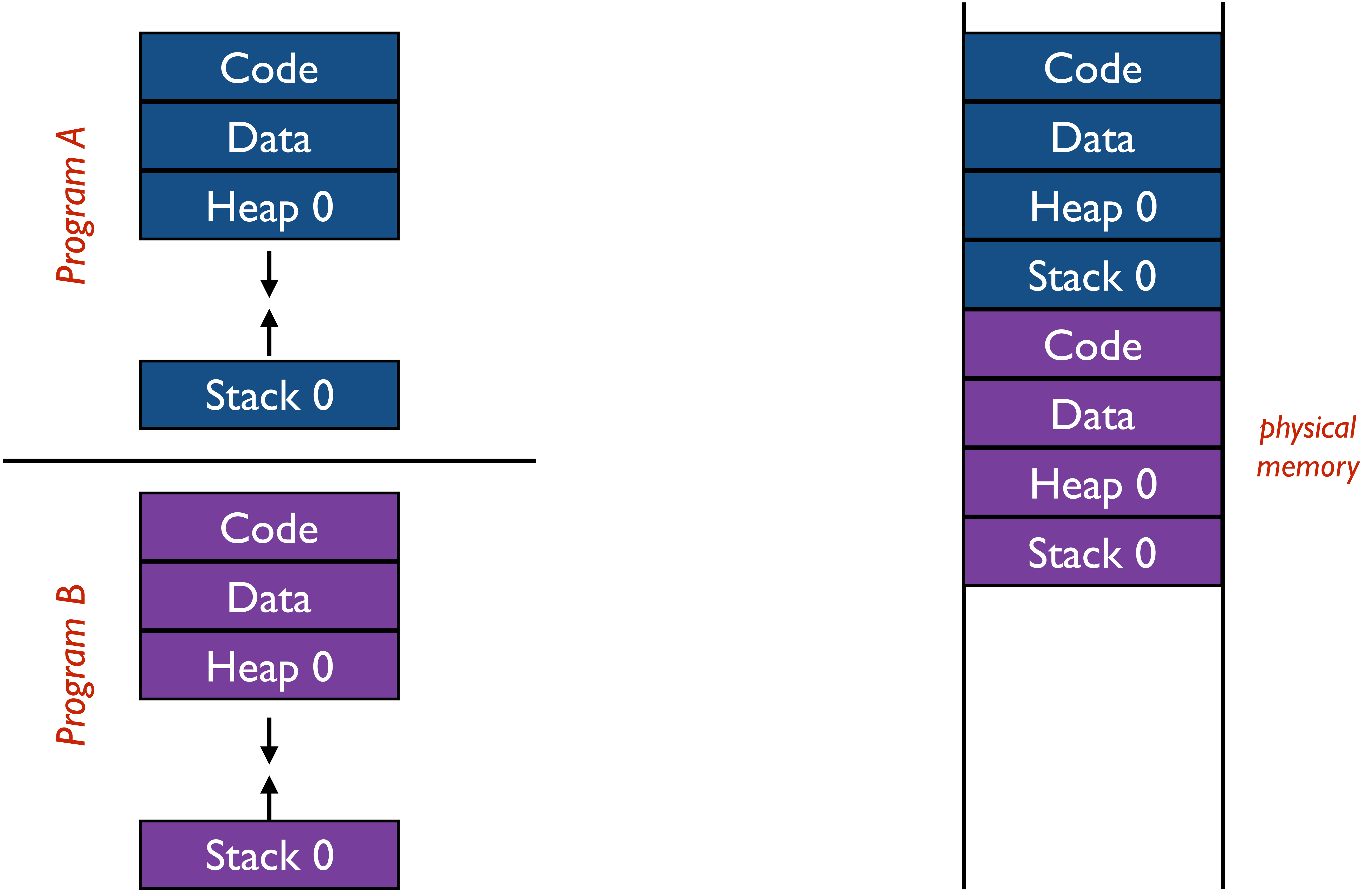
Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0

Paging

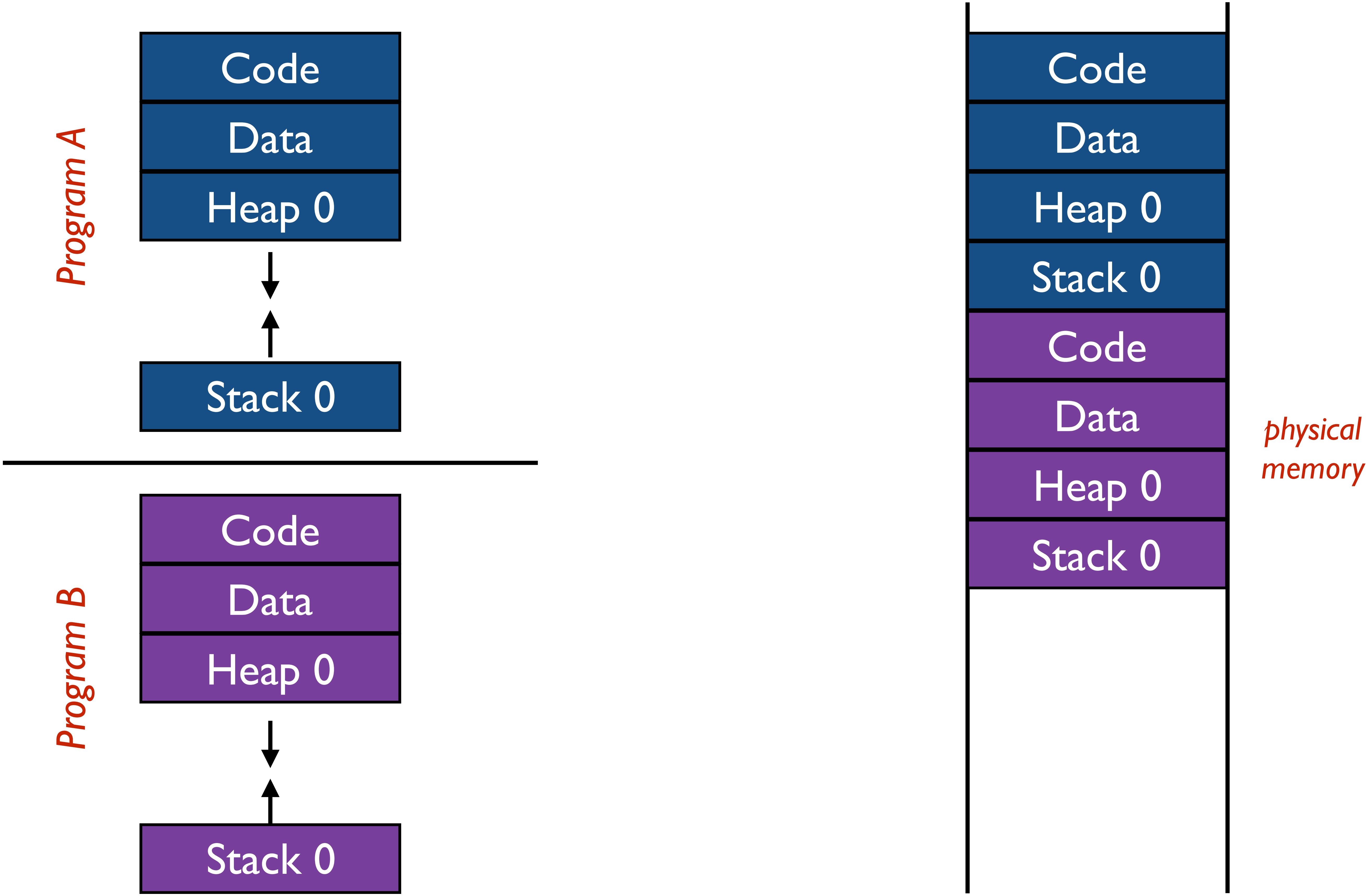


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
physical memory

Paging

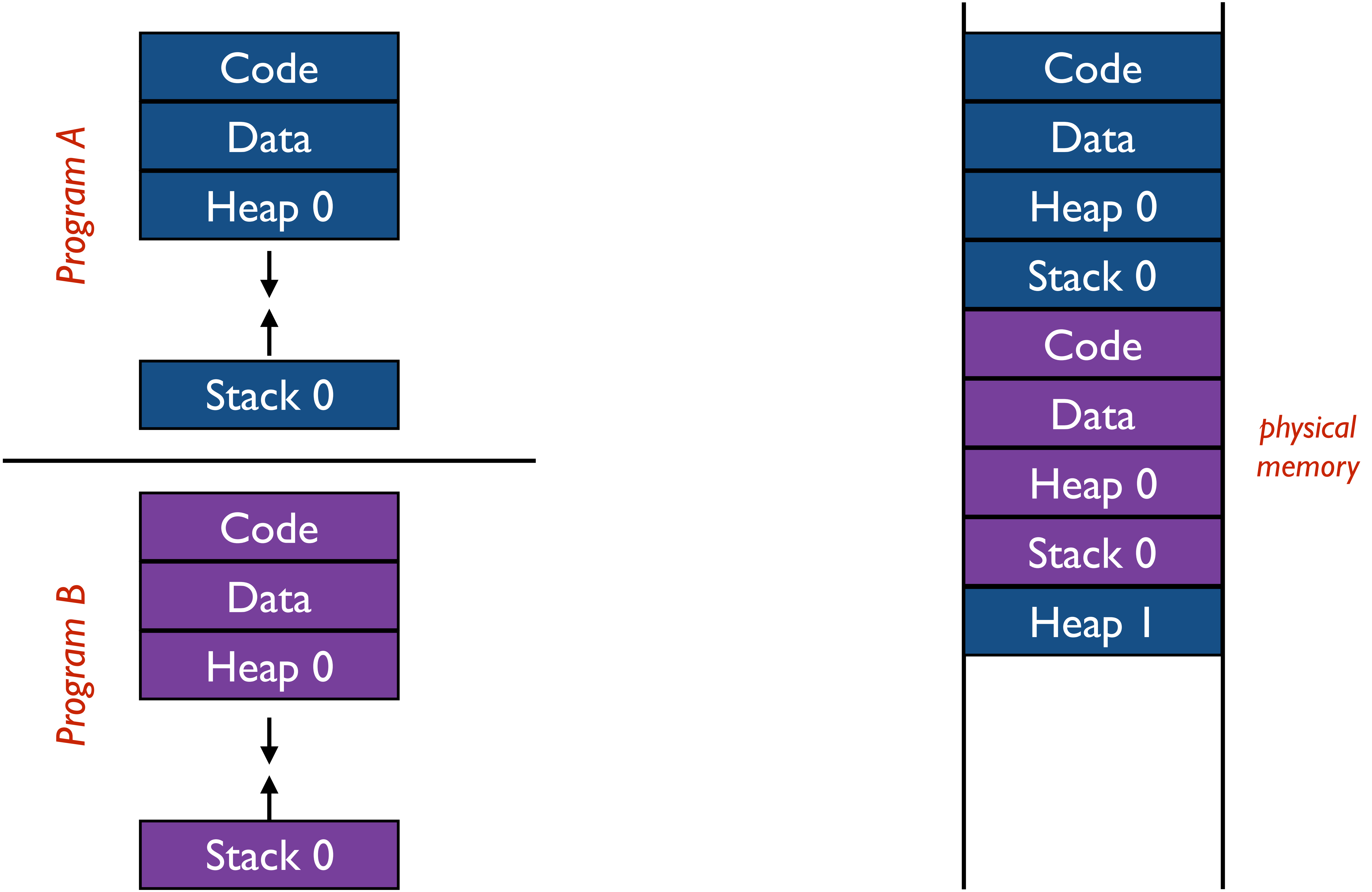

Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
physical memory

Paging

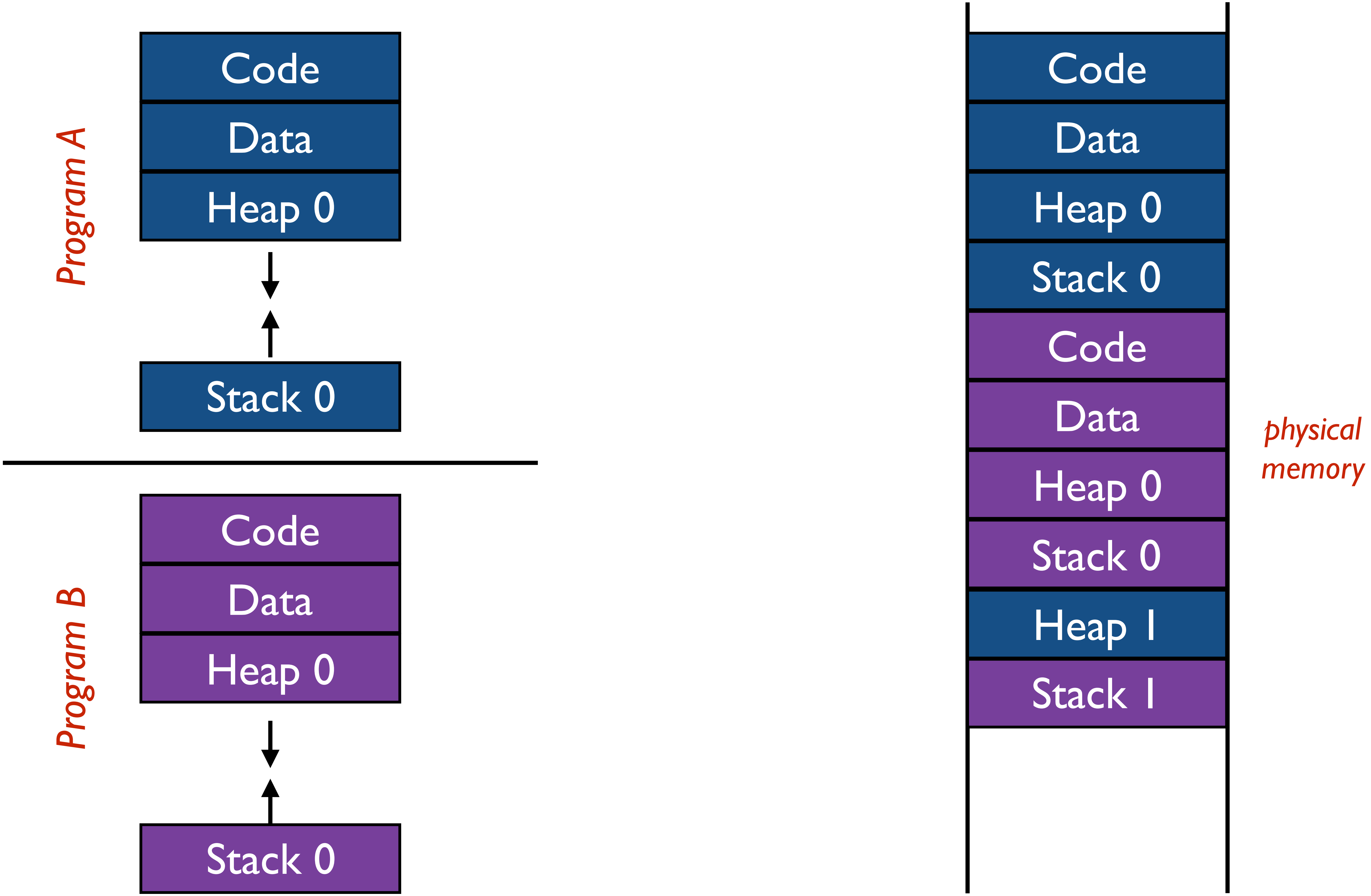


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
physical memory

Paging

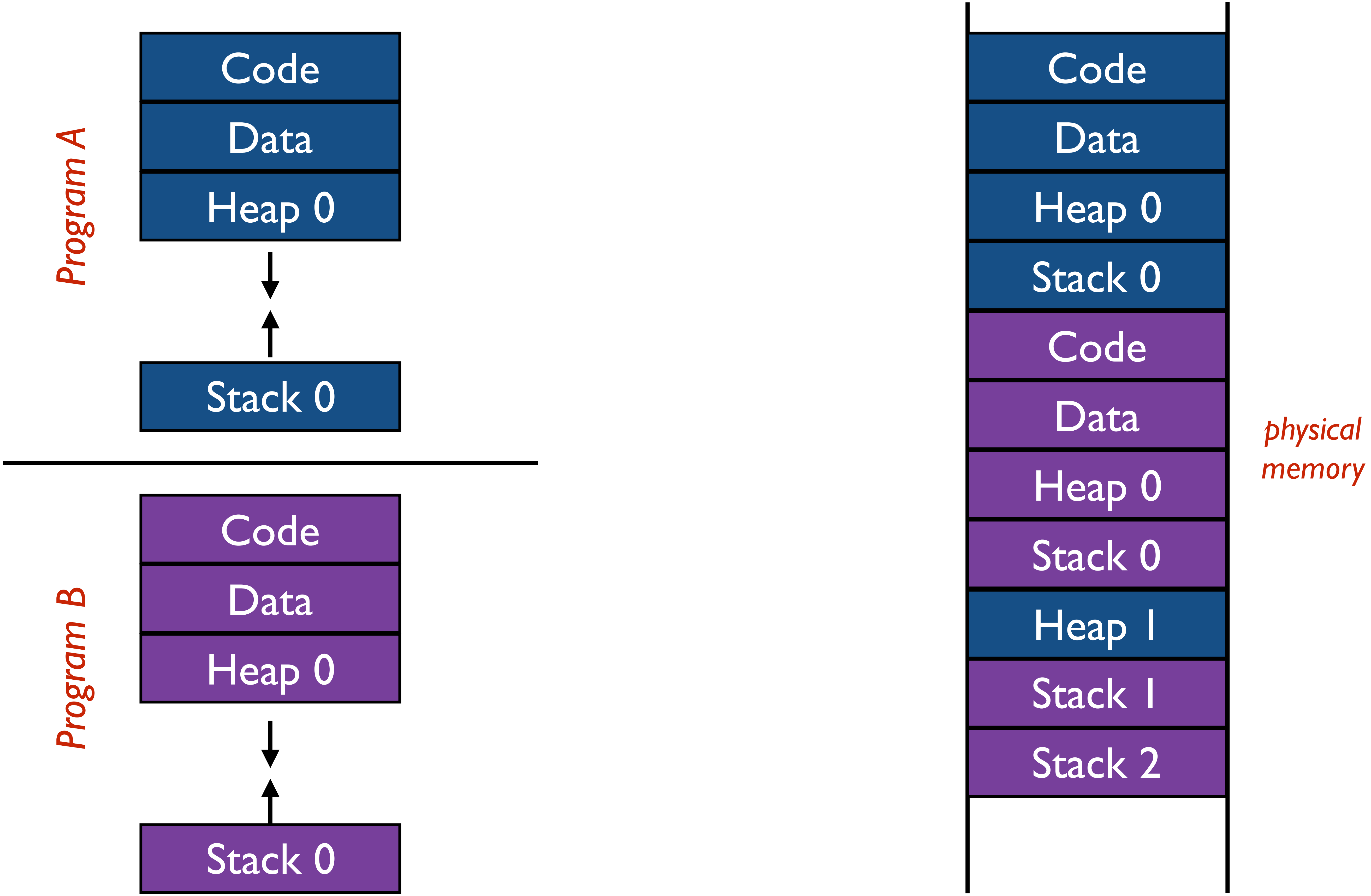

Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Heap 1
physical memory

Paging



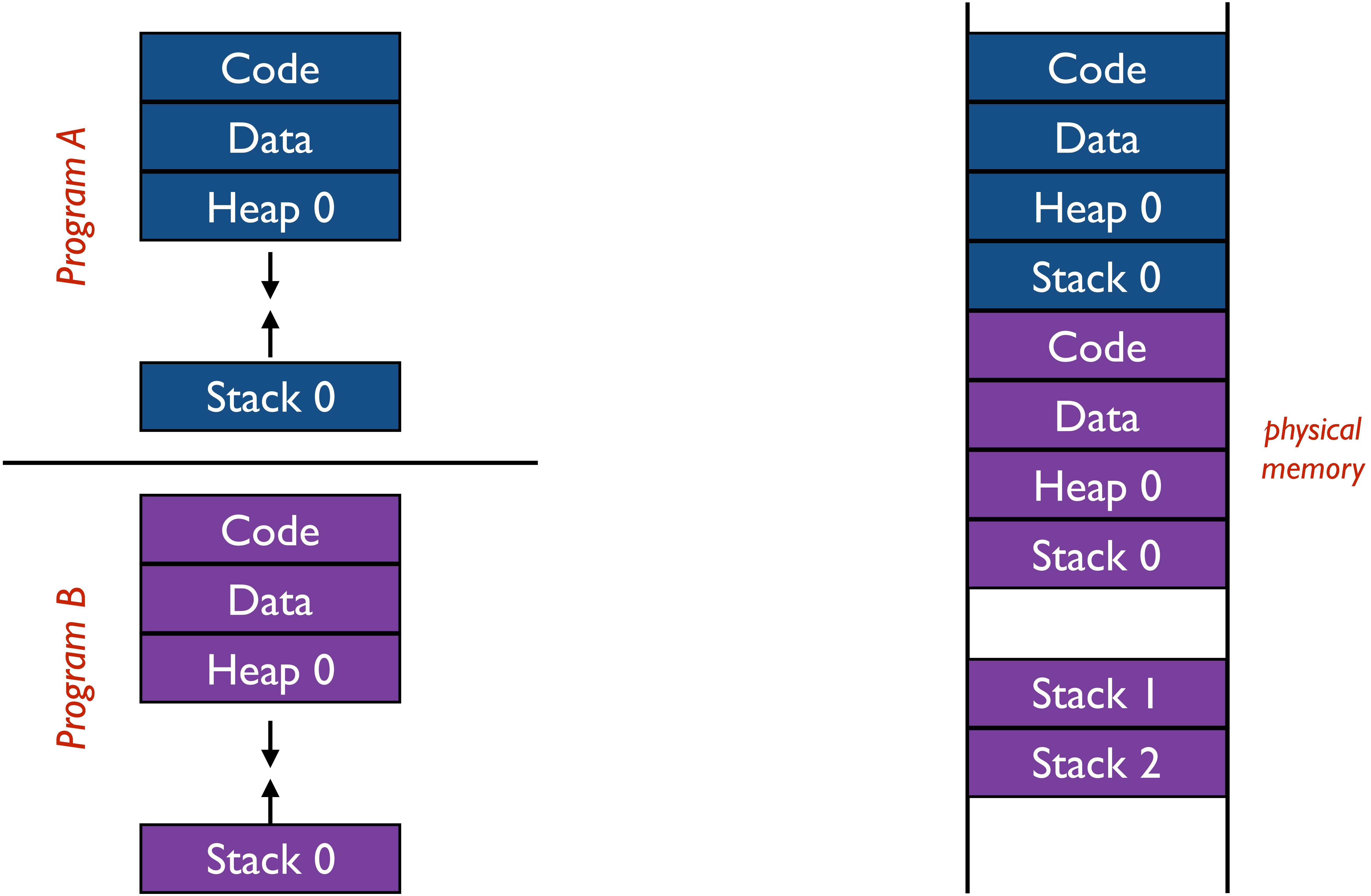
Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Heap 1
Stack 1
physical memory

Paging

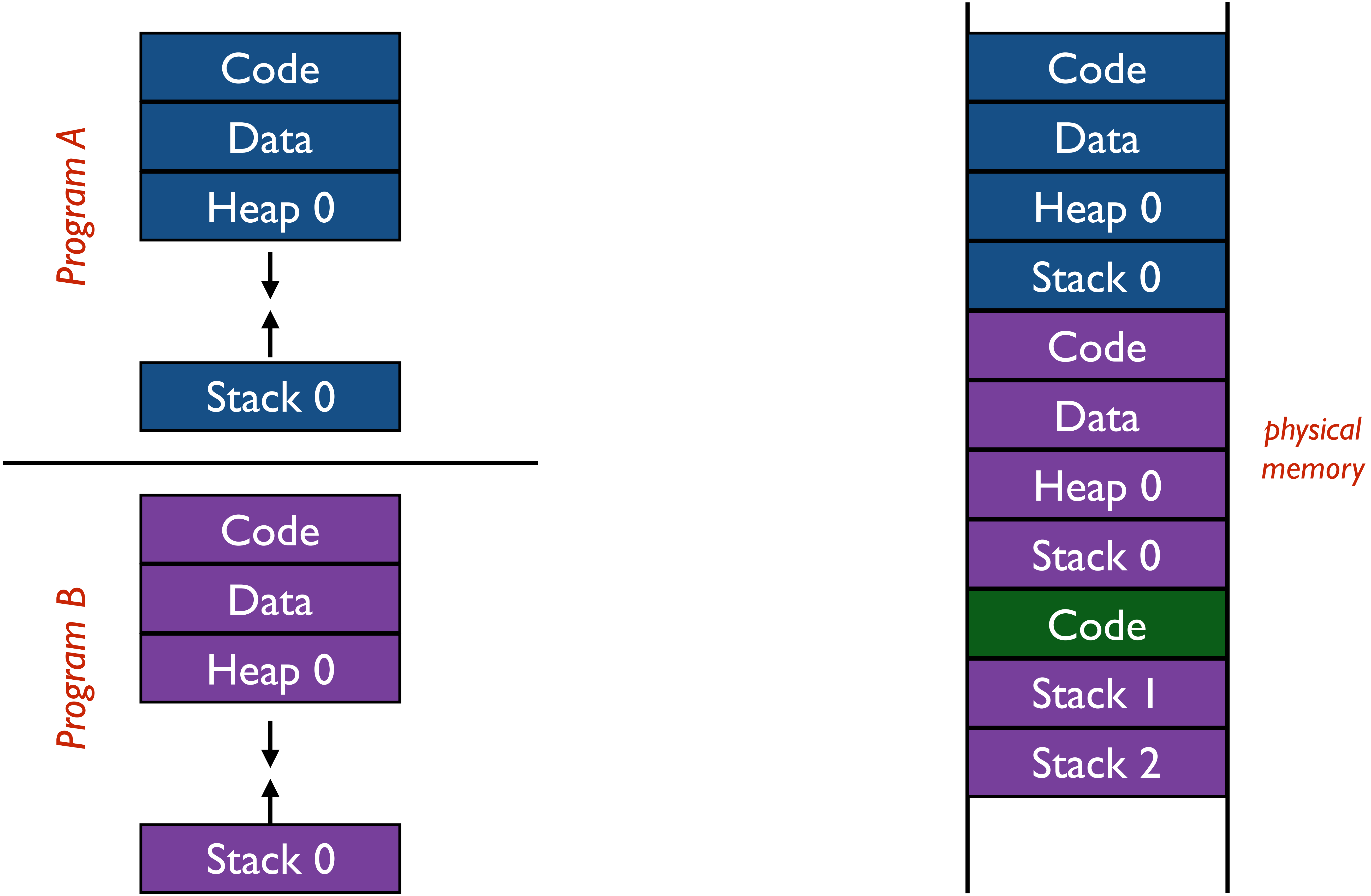


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Heap 1
Stack 1
Stack 2
physical memory

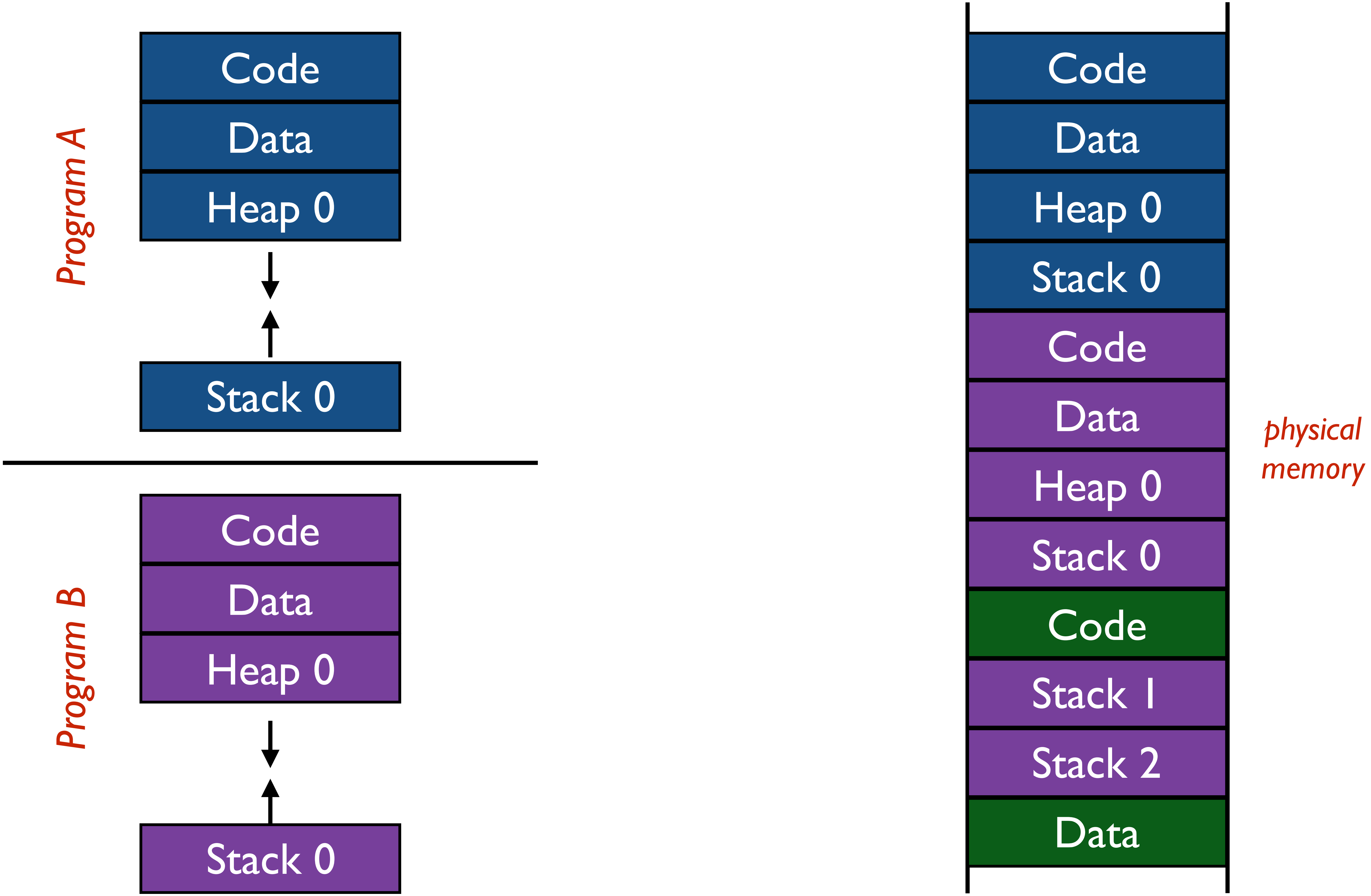
Paging


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Stack 1
Stack 2
physical memory

Paging

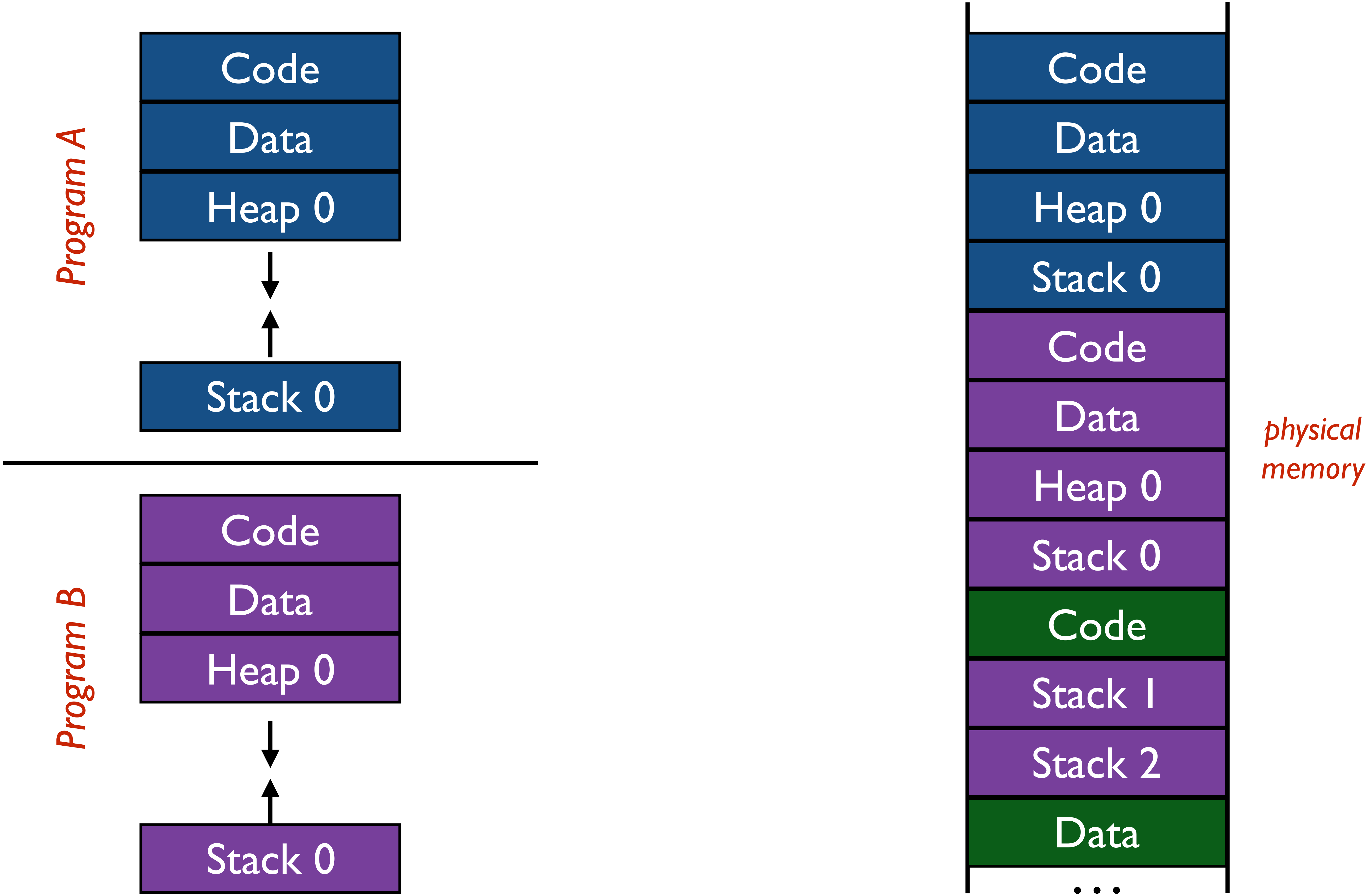

Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Code
Stack 1
Stack 2
physical memory

Paging

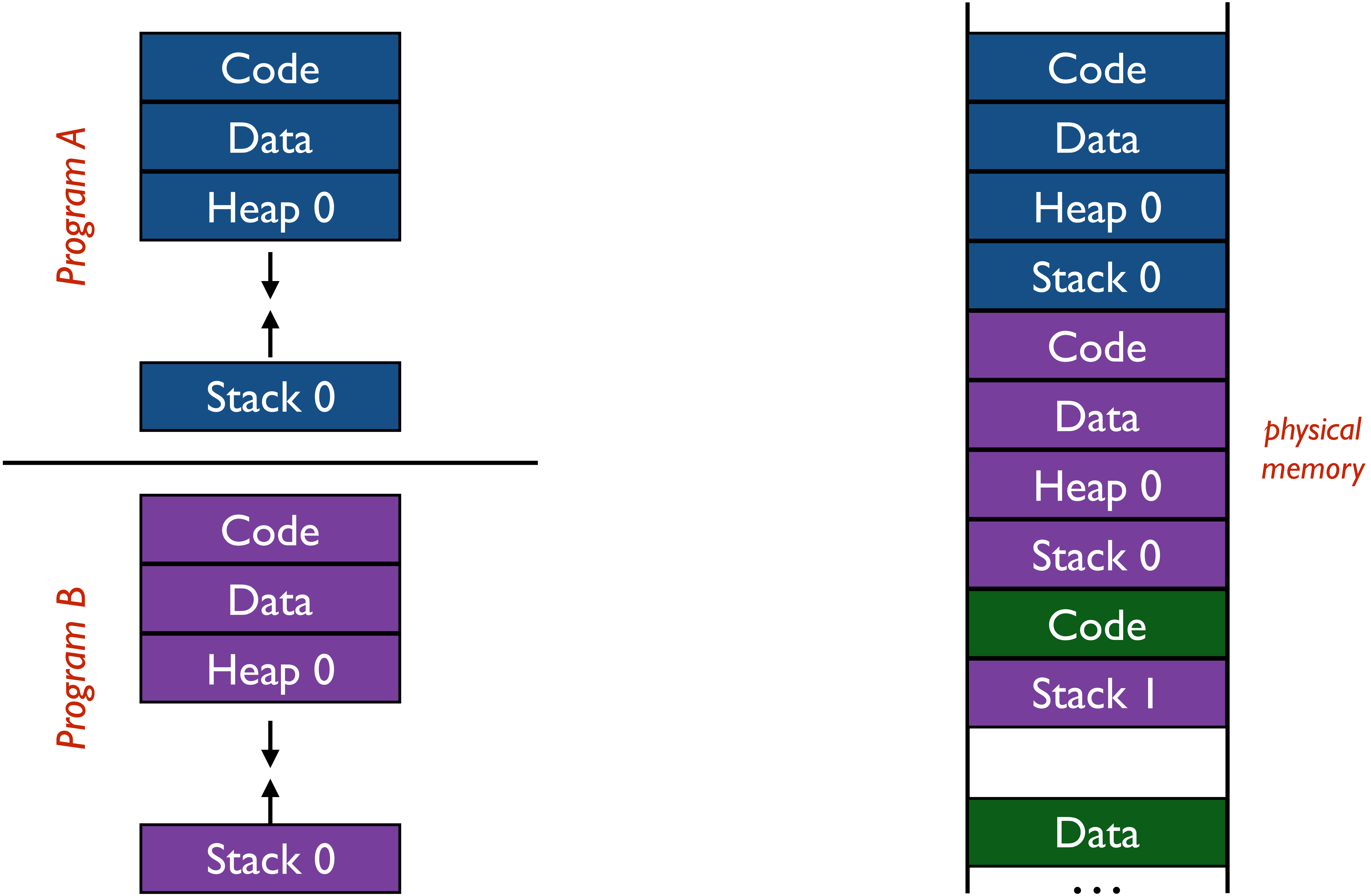


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Code
Stack 1
Stack 2
Data
physical memory

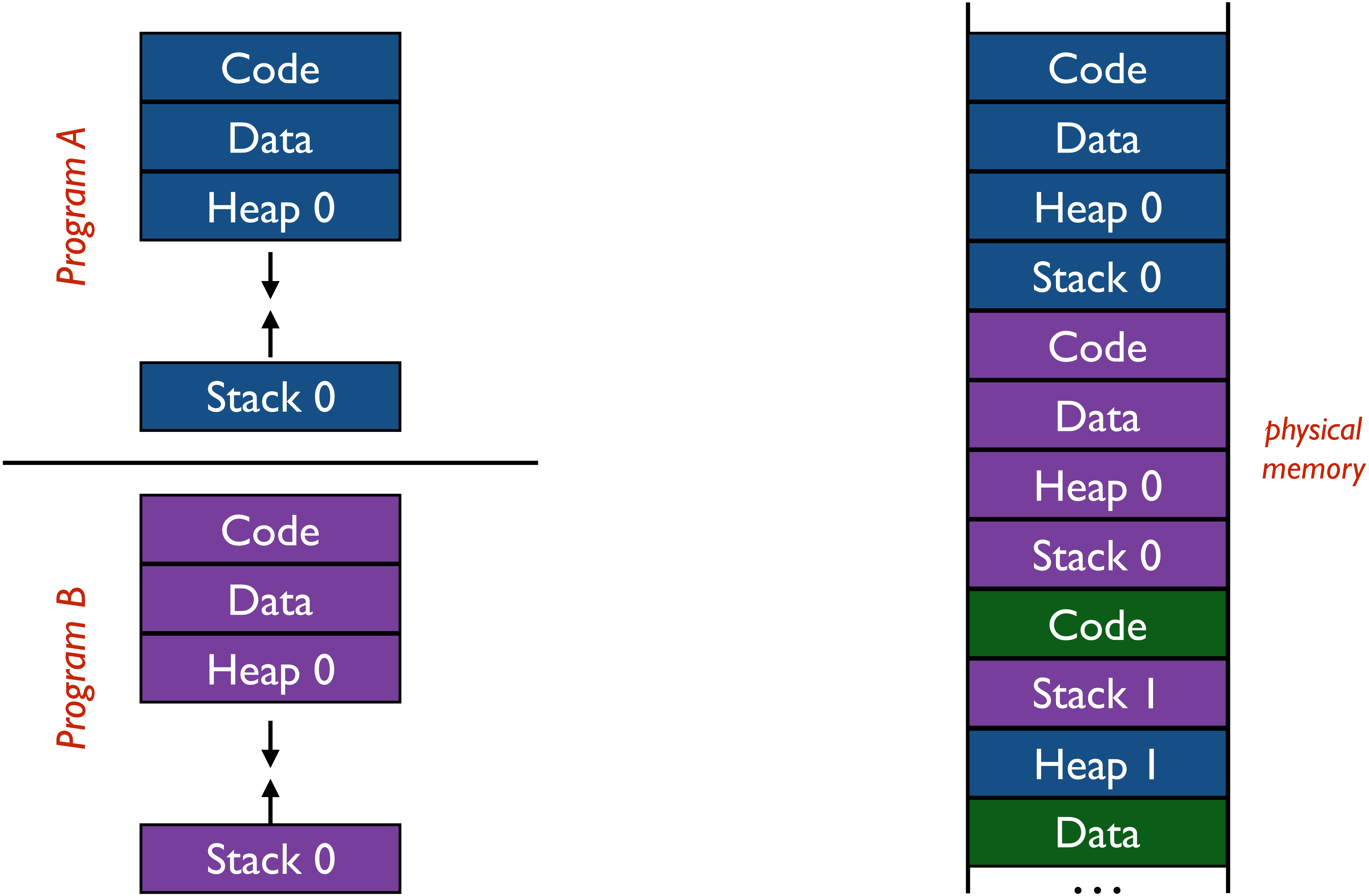
Paging


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Code
Stack 1
Stack 2
Data
...
physical memory

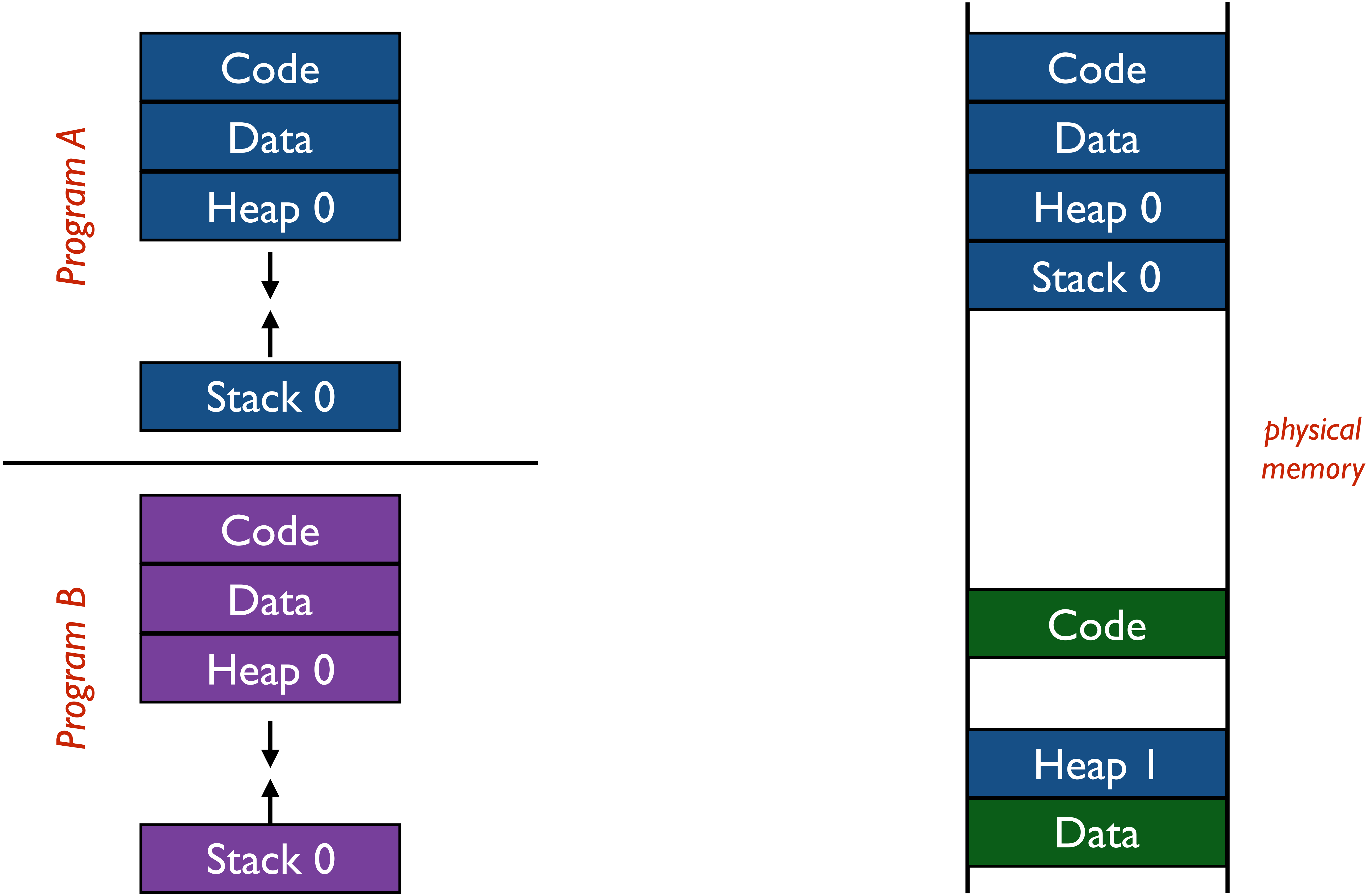
Paging


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Code
Stack 1
Data
. . .
physical memory

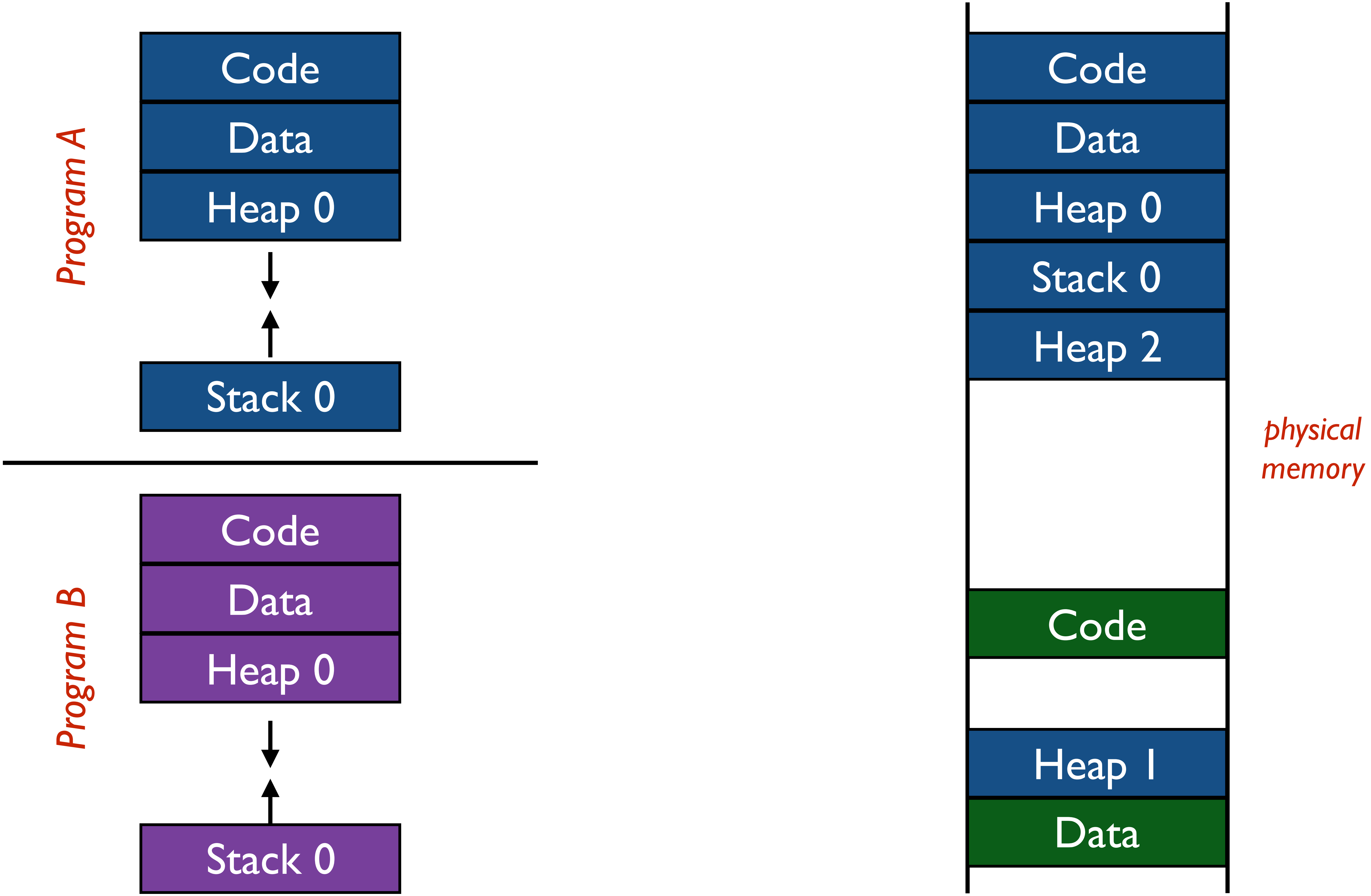
Paging


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Code
Stack 1
Heap 1
Data
. . .
physical memory

Paging

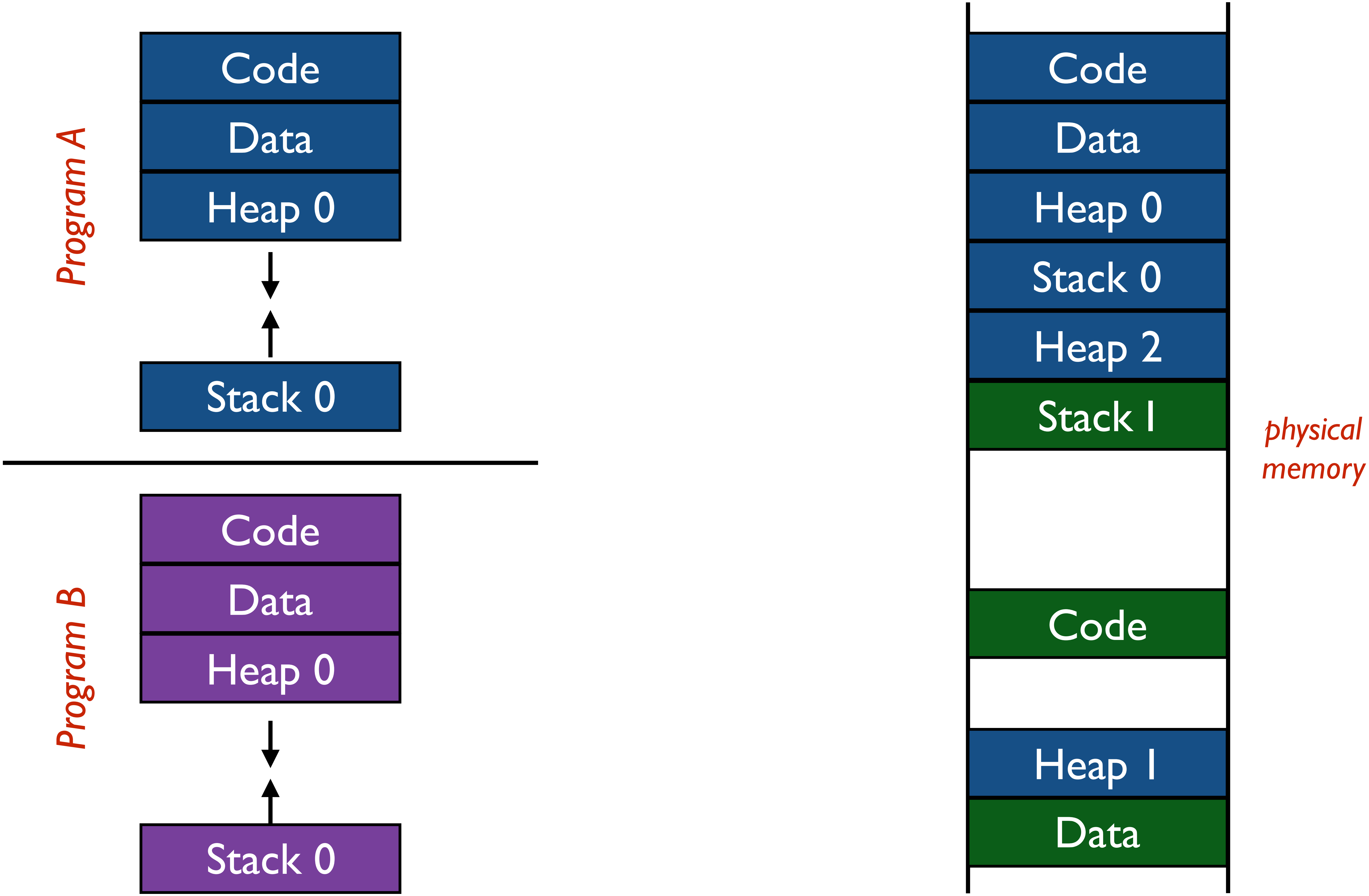

Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
physical memory
Code
Heap 1
Data

Paging

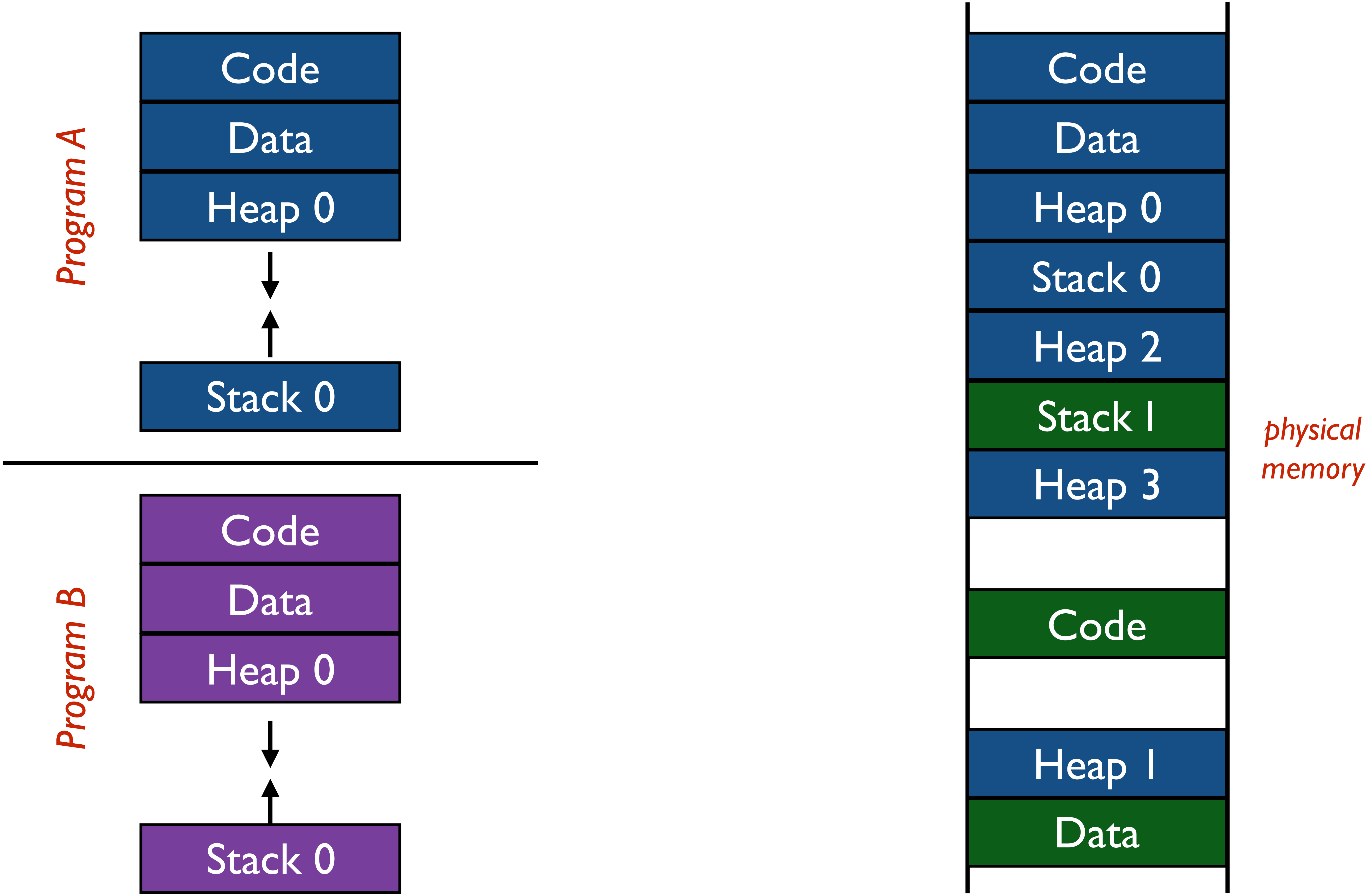


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Heap 2
physical memory
Code
Heap 1
Data

Paging

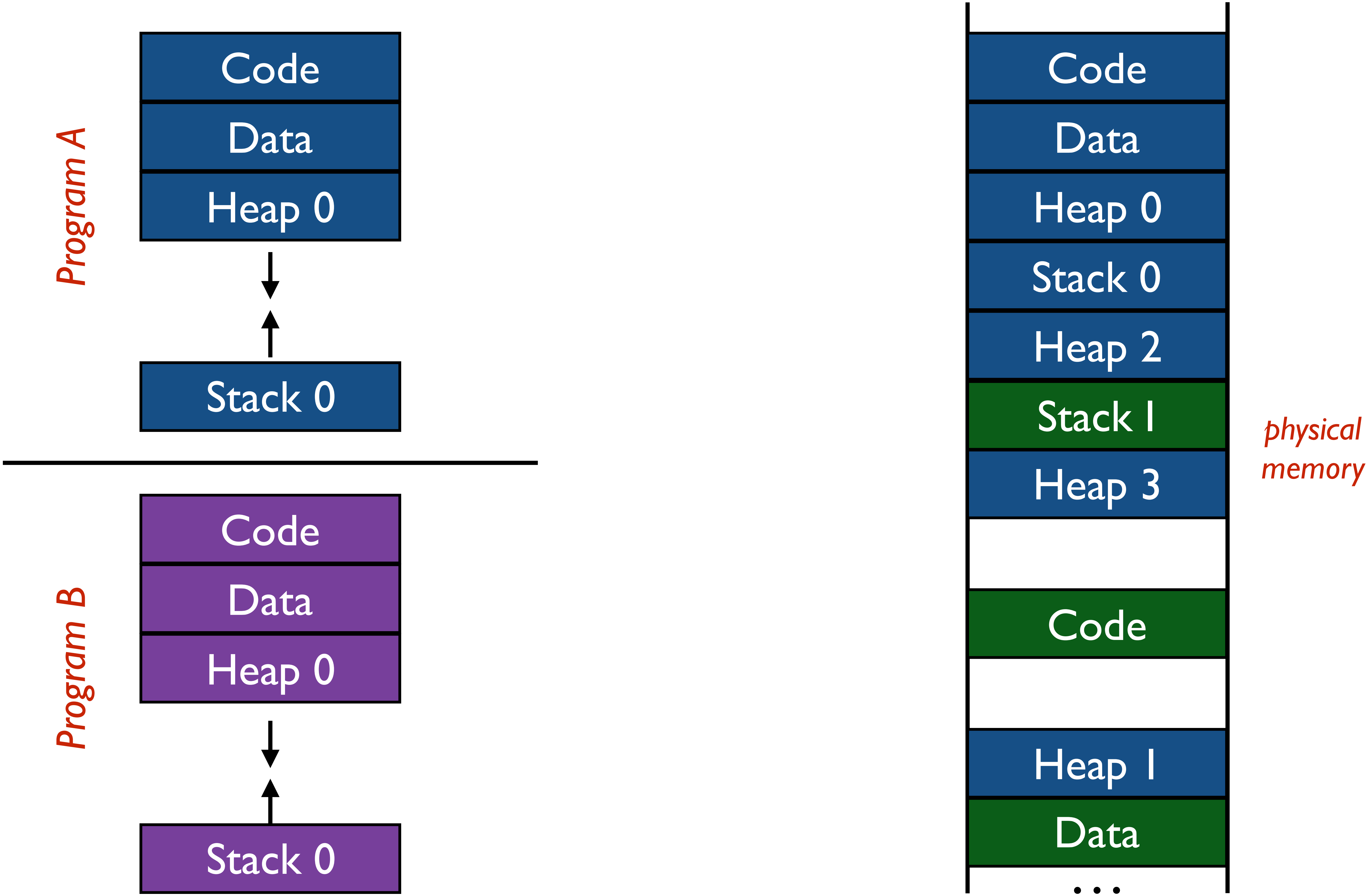

Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Heap 2
Stack 1
Code
Heap 1
Data
physical memory

Paging

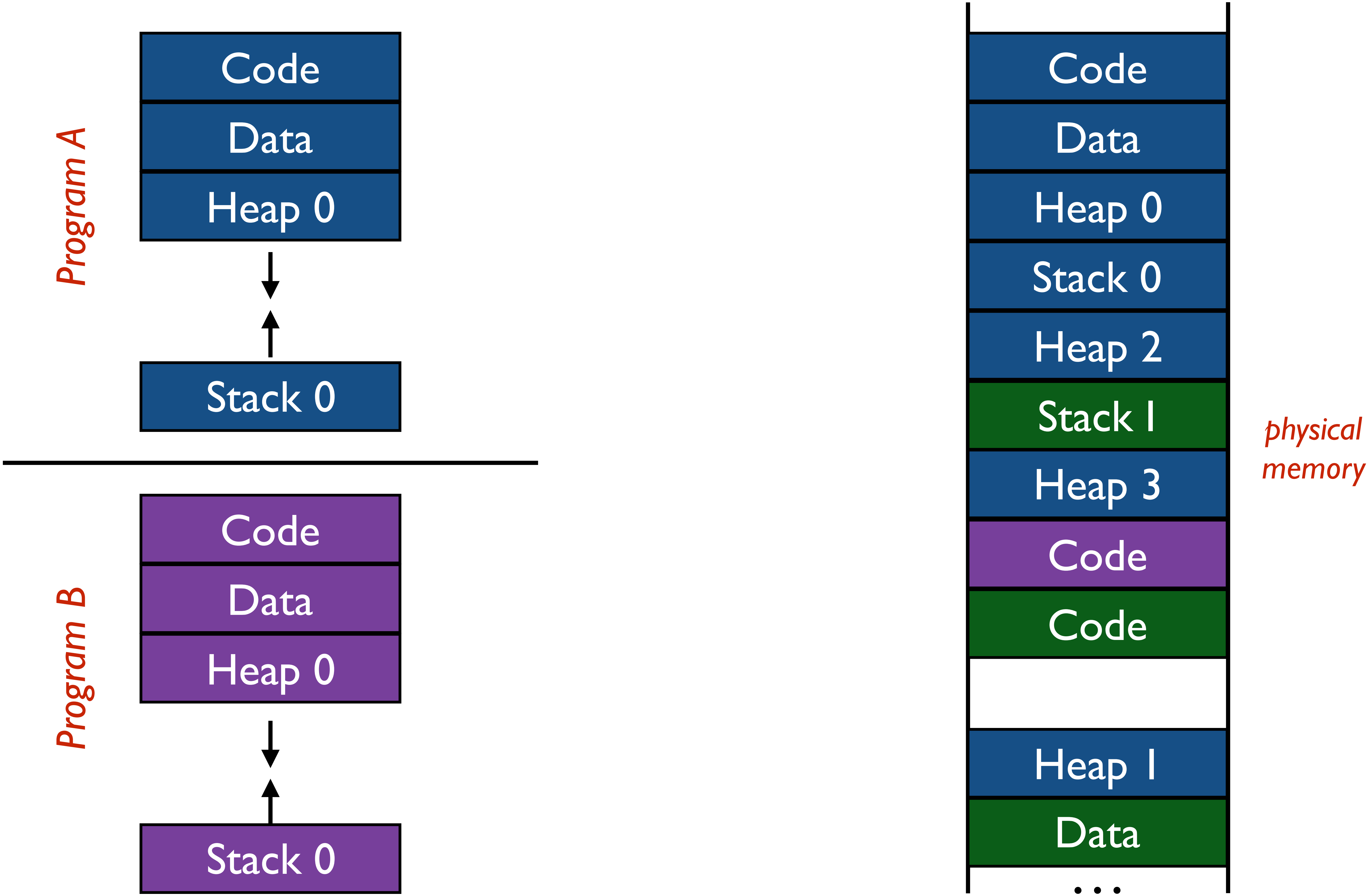


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Heap 2
Stack 1
Heap 3
Code
Heap 1
Data
physical memory

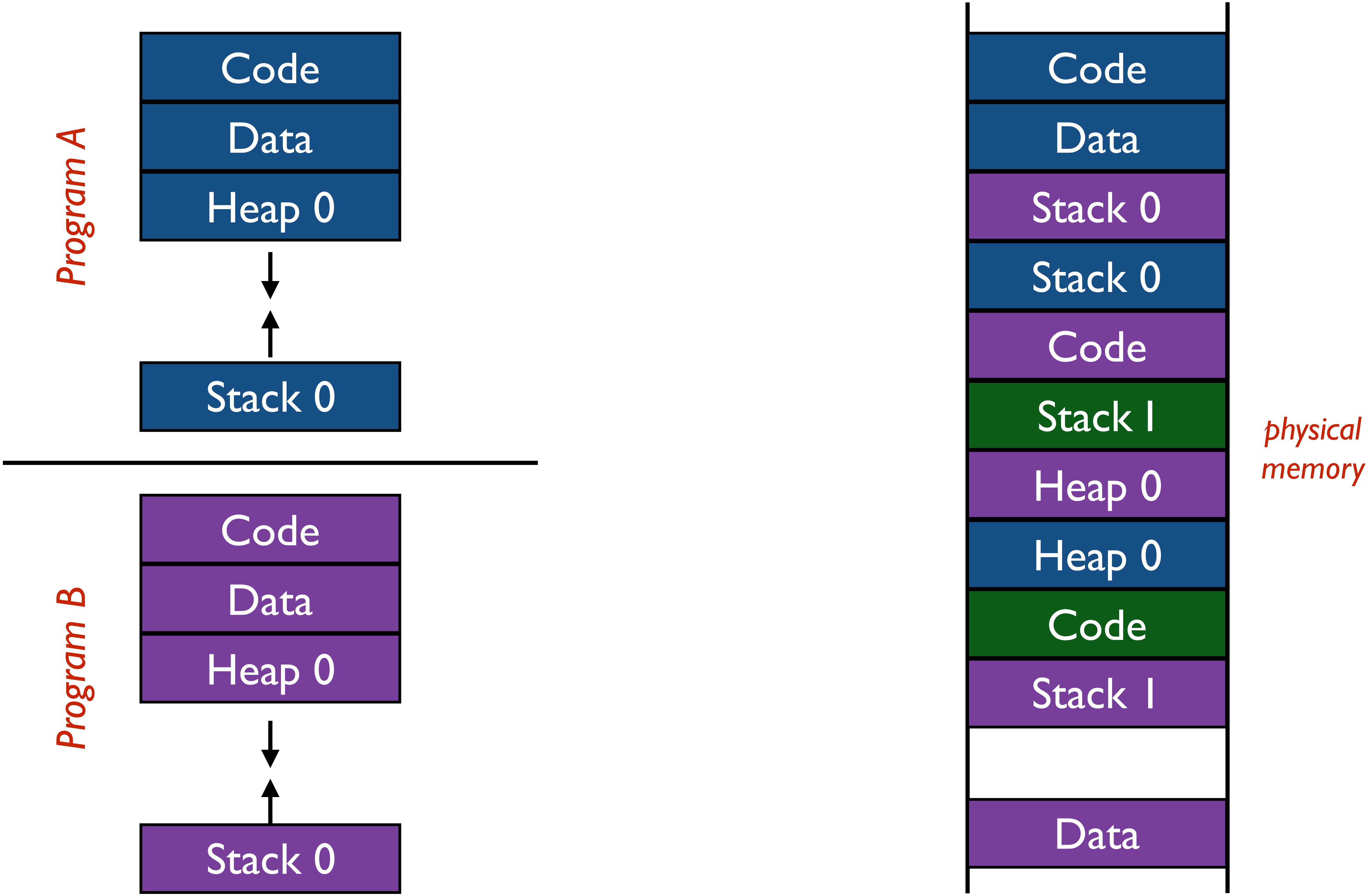
Paging


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Heap 2
Stack 1
Heap 3
Code
Heap 1
Data
...
physical memory

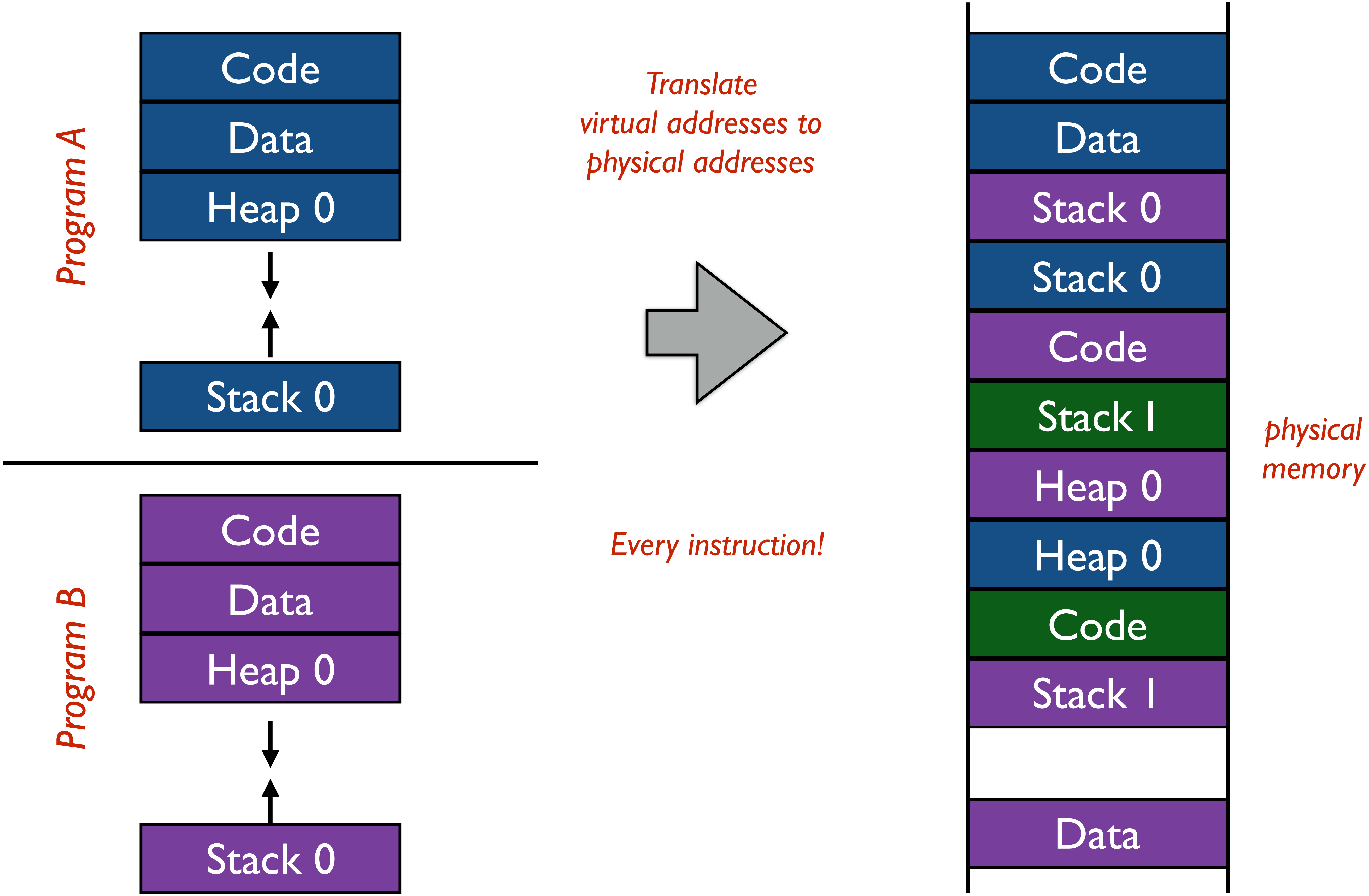
Paging


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Heap 0
Stack 0
Heap 2
Stack 1
Heap 3
Code
Code
Heap 1
Data
. . .
physical memory

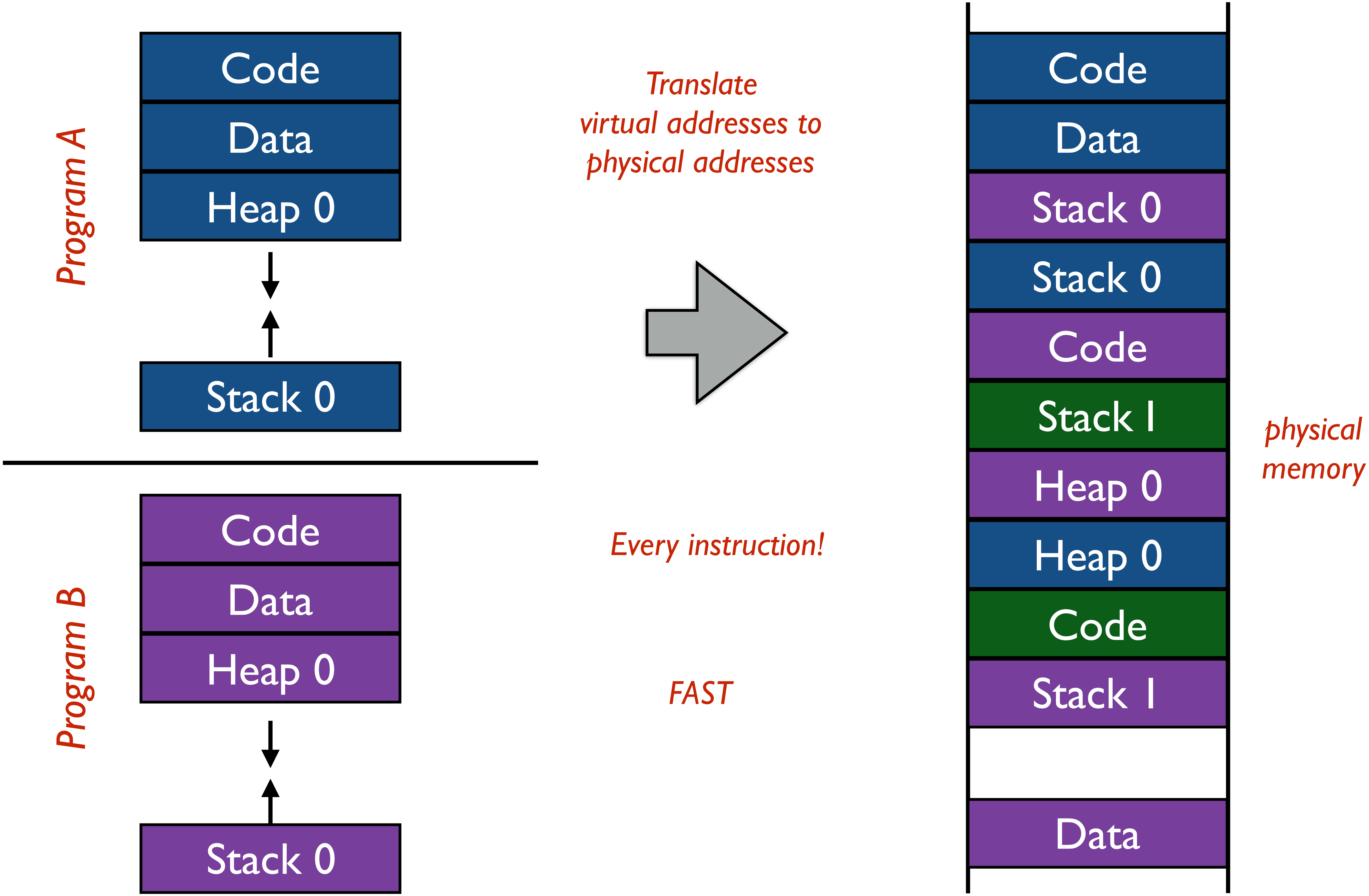
Paging


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Code
Data
Stack 0
Stack 0
Code
Stack 1
Heap 0
Heap 0
Code
Stack 1
Data
physical
memory

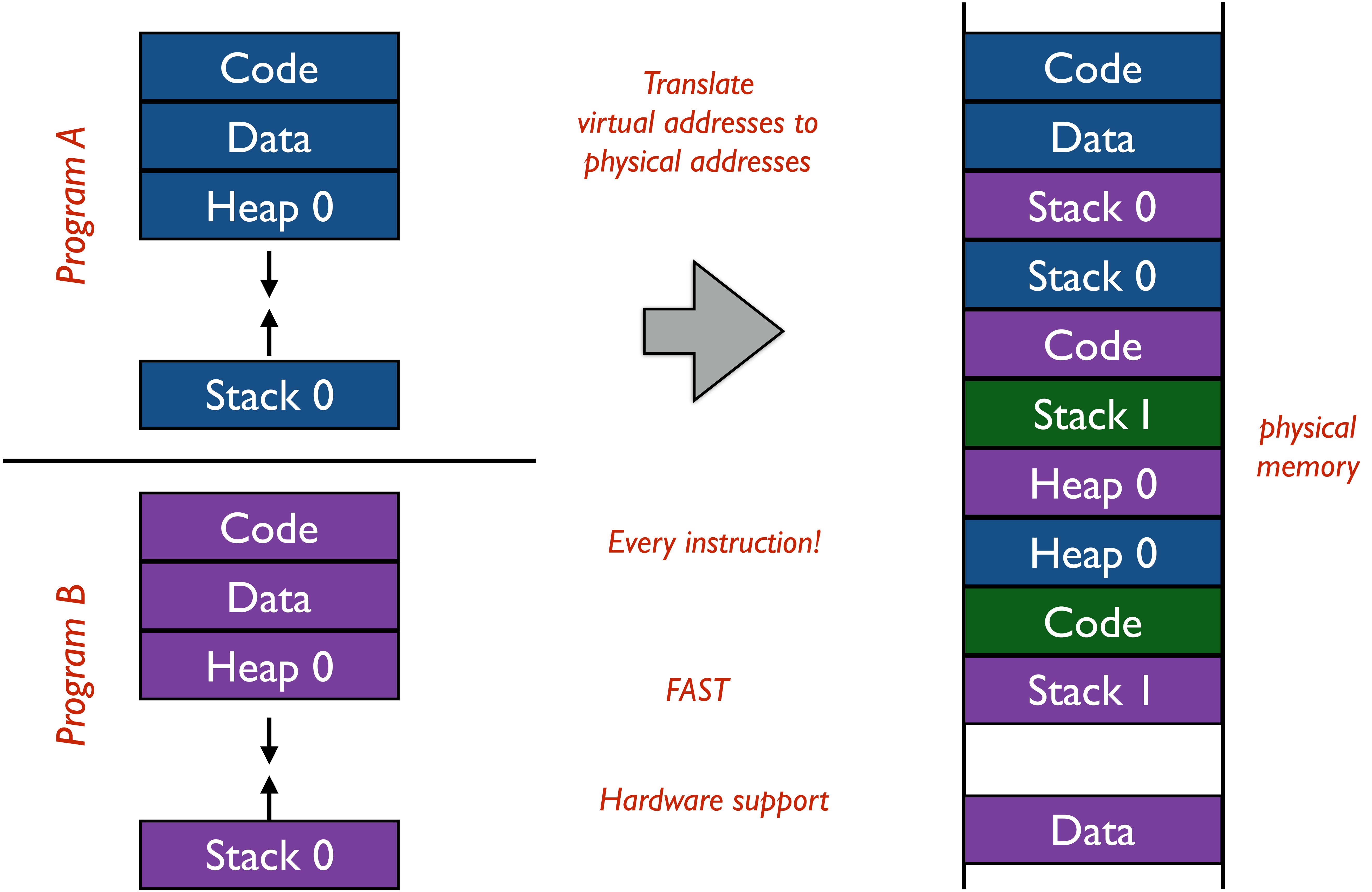
Paging


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Translate
virtual addresses to
physical addresses
Every instruction!
Code
Data
Stack 0
Stack 0
Code
Stack 1
Heap 0
Heap 0
Code
Stack 1
Data
physical
memory

Paging


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Translate
virtual addresses to
physical addresses
Every instruction!
FAST
Code
Data
Stack 0
Stack 0
Code
Stack 1
Heap 0
Heap 0
Code
Stack 1
Data
physical
memory

Paging


Program A
Code
Data
Heap 0
Stack 0
Program B
Code
Data
Heap 0
Stack 0
Translate
virtual addresses to
physical addresses
Every instruction!
FAST
Hardware support
Code
Data
Stack 0
Stack 0
Code
Stack 1
Heap 0
Heap 0
Code
Stack 1
Data
physical
memory

Fim