A Formal Specification of x86 Memory Management

Shilpi Goel
shigoel@cs.utexas.edu

Warren A. Hunt, Jr.
hunt@cs.utexas.edu

The University of Texas at Austin

6th May, 2015

HCSS 2015

Outline

- Motivation
- Project Overview
- IA-32e Paging
  - Specification
  - Verification
- Future Work
- Conclusion

Motivation

- Cost of incorrect software is extremely high.
- *Formal verification* can increase software quality.
- Programs never run in isolation.
- Program analysis should not be done in isolation.

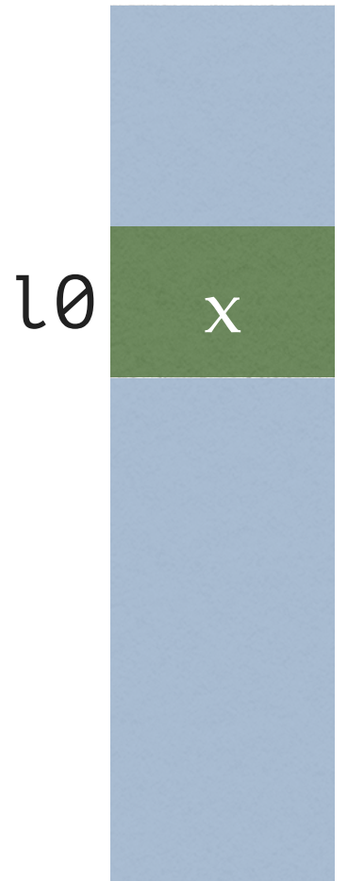
Motivation

- Cost of incorrect software is extremely high.
- *Formal verification* can increase software quality.
- Programs never run in isolation.
- Program analysis should not be done in isolation.
- We need to analyze:
  - *Libraries included by the program*
    - E.g., `stdlib.h`
  - *Low-level operating system routines*
    - E.g., system call services
  - *Hardware protection mechanisms*
    - E.g., memory protection via segmentation and paging

Example: Analysis of a Data-Copy Program

Specification:

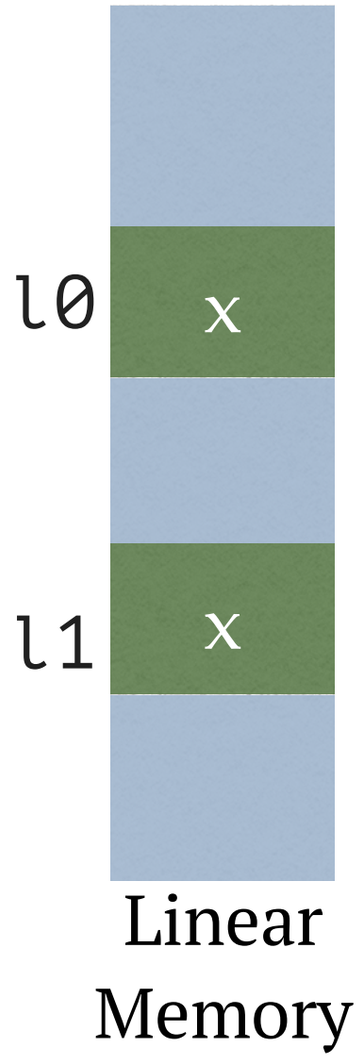
Copy data `x` from linear (virtual) memory location `l0` to disjoint linear memory location `l1`.



Example: Analysis of a Data-Copy Program

Specification:

Copy data `x` from linear (virtual) memory location `l0` to disjoint linear memory location `l1`.



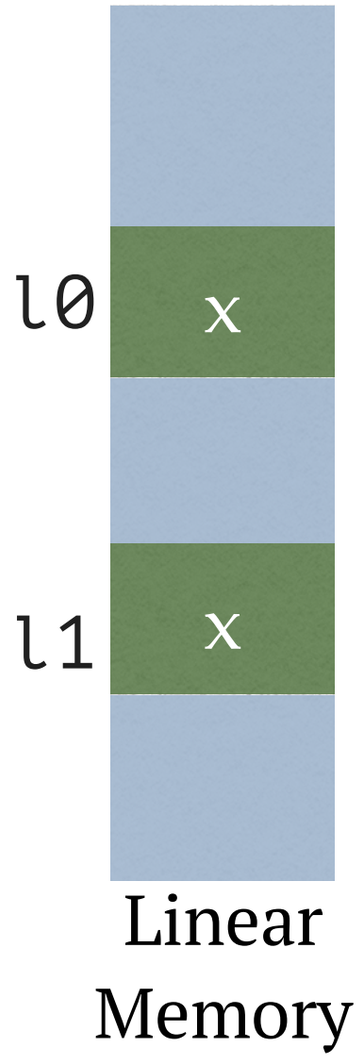
Example: Analysis of a Data-Copy Program

Specification:

Copy data `x` from linear (virtual) memory location `l0` to disjoint linear memory location `l1`.

Verification Objective:

After a successful copy, `l0` and `l1` contain `x`.



Example: Analysis of a Data-Copy Program

Specification:

Copy data `x` from linear (virtual) memory location `l0` to disjoint linear memory location `l1`.

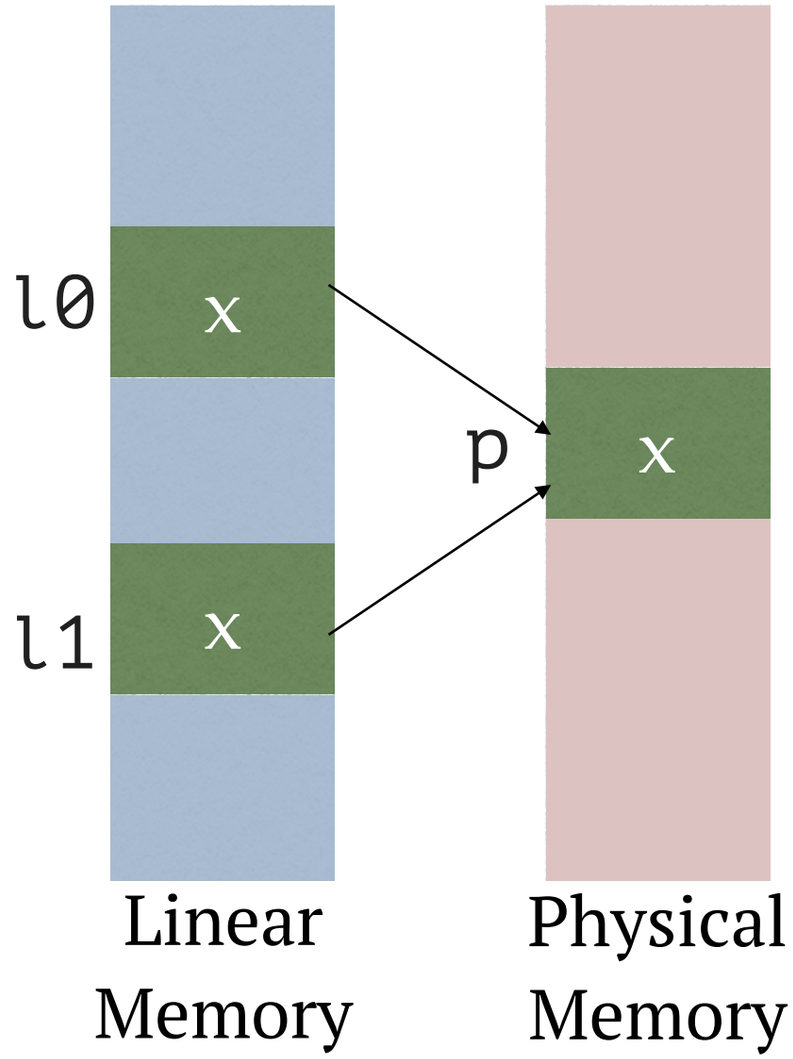
Verification Objective:

After a successful copy, `l0` and `l1` contain `x`.

Implementation:

Include the *copy-on-write* technique: `l0` and `l1` can be mapped to the same physical memory location `p`.

- System calls
- Modifications to address mapping
- Access control management



Our Goal

Goal: Build robust tools to increase software reliability

- Verify critical properties of application and system programs
- Correctness with respect to behavior, security, & resource usage

Approach: Machine-code verification for x86 platforms

Plan of Action:

1. Build a **formal, executable x86 ISA model** using ACL2
   - Includes a specification of segmentation and paging *(new this year!)*
2. Develop a **machine-code analysis framework** based on this model
3. Employ this framework to **verify application and system programs**

Our Goal

Goal: Build robust tools to increase software reliability

- Verify critical properties of application and system programs
- Correctness with respect to behavior, security, & resource usage

Approach: Machine-code verification for x86 platforms

Plan of Action:

1. Build a **formal, executable x86 ISA model** using ACL2
   - Includes a specification of segmentation and paging *(new this year!)*
2. Develop a **machine-code analysis framework** based on this model
3. Employ this framework to **verify application and system programs**

Focus of this Talk

- **System program verification** differs from application program verification.
  - Access to a larger machine state
  - Based on physical memory
  - Many data structures to maintain simple interfaces to applications

Focus of this Talk

- **System program verification** differs from application program verification.
  - Access to a larger machine state
  - Based on physical memory
  - Many data structures to maintain simple interfaces to applications
- Linear memory is an **abstraction** provided by **paging data structures**.

Focus of this Talk

- **System program verification** differs from application program verification.
  - Access to a larger machine state
  - Based on physical memory
  - Many data structures to maintain simple interfaces to applications

- Linear memory is an **abstraction** provided by **paging data structures**.

- Paging data structures control:
  - **virtualization**: translation from linear to physical address
  - **memory protection**: access rights (r/w/x)
  - **memory typing**

Focus of this Talk

- **System program verification** differs from application program verification.
  - Access to a larger machine state
  - Based on physical memory
  - Many data structures to maintain simple interfaces to applications
- Linear memory is an **abstraction** provided by **paging data structures**.
- Paging data structures control:
  - **virtualization**: translation from linear to physical address
  - **memory protection**: access rights (r/w/x)
  - **memory typing**

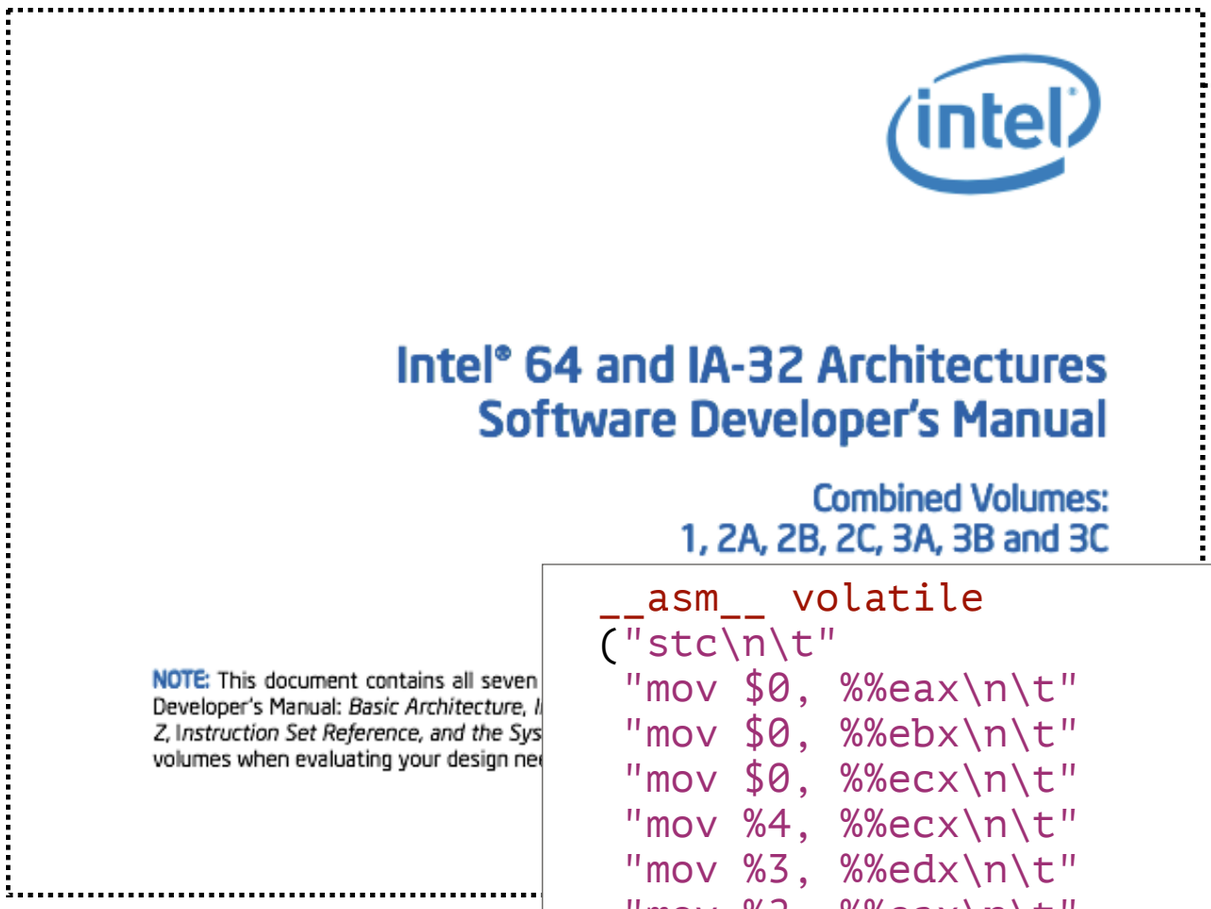
Focus: **Specifying and Reasoning about IA-32e Paging**

Outline

- Motivation
- **Project Overview**
- IA-32e Paging
  - Specification
  - Verification
- Future Work
- Conclusion

x86 ISA Model Development

Obtaining the x86 ISA Specification



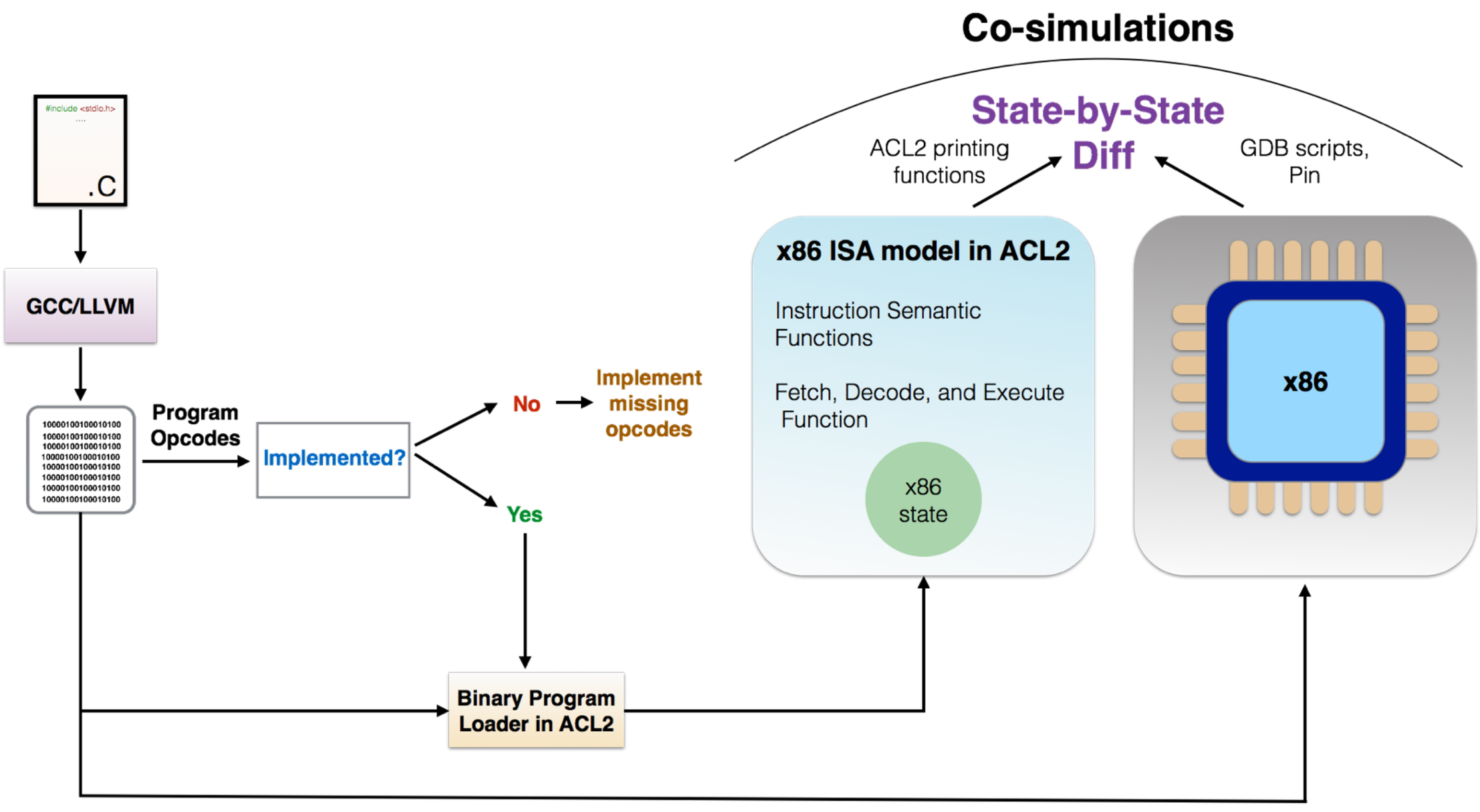

```
__asm__ volatile
("stc\n\t"                          // Set CF.
 "mov $0, %%eax\n\t"                // Set EAX = 0.
 "mov $0, %%ebx\n\t"                // Set EBX = 0.
 "mov $0, %%ecx\n\t"                // Set ECX = 0.
 "mov %4, %%ecx\n\t"                // Set CL = rotate_by.
 "mov %3, %%edx\n\t"                // Set EDX = old_cf = 1.
 "mov %2, %%eax\n\t"                // Set EAX = num.
 "rcl %%cl, %%al\n\t"               // Rotate AL by CL.
 "cmovb %%edx, %%ebx\n\t"           // Set EBX = old_cf if CF = 1.
                                    // Otherwise, EBX = 0.
 "mov %%eax, %0\n\t"                // Set res = EAX.
 "mov %%ebx, %1\n\t"                // Set cf  = EBX.

 : "=g"(res), "=g"(cf)
 : "g"(num), "g"(old_cf), "g"(rotate_by)
 : "rax", "rbx", "rcx", "rdx");
```

Running tests on x86 machines

x86 ISA Model Validation

How can we know that our model faithfully represents the x86 ISA?

Validate the model to increase trust in the applicability of formal analysis.



x86 ISA Model: Current Status

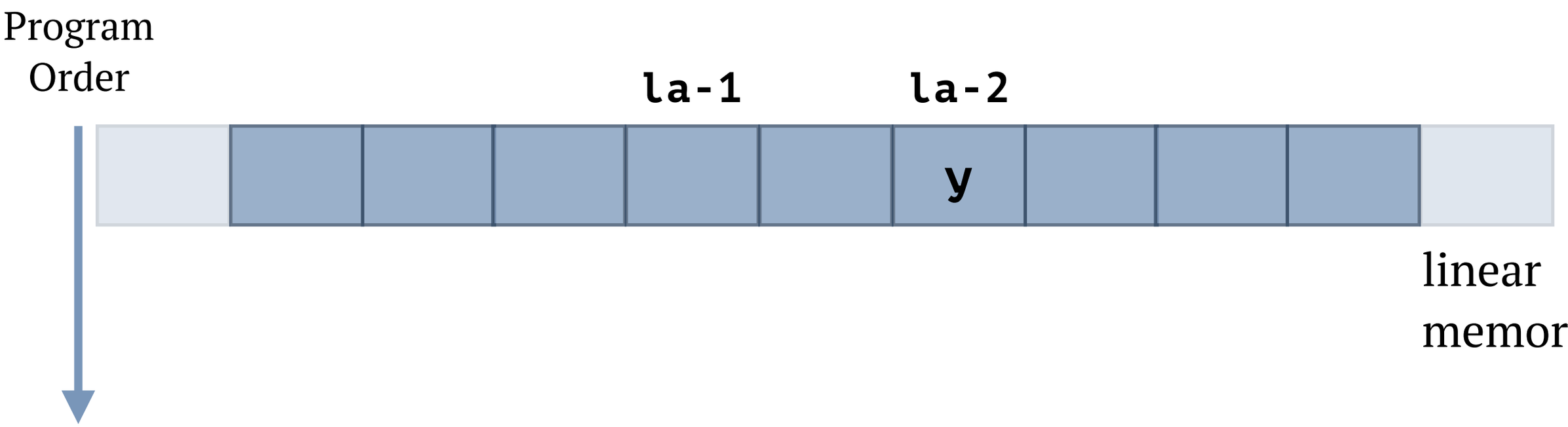
- The x86 ISA model supports 100+ instructions (**~220 opcodes**)
  - Exceptions: some FP and SIMD instructions
  - Can execute "real" programs emitted by GCC/LLVM
  - Successfully co-simulated a contemporary SAT solver on our model
- **IA-32e paging** for all page configurations (4K, 2M, 1G)
- **Segment-based addressing**
- **Simulation speed**[*]:
  - ~3.3 million instructions/second (paging disabled)
  - ~330,000 instructions/second (with 1G pages)
- **Verification** of several x86 **application programs**

[*] Simulation speed measured on an Intel Xeon E31280 CPU @ 3.50GHz

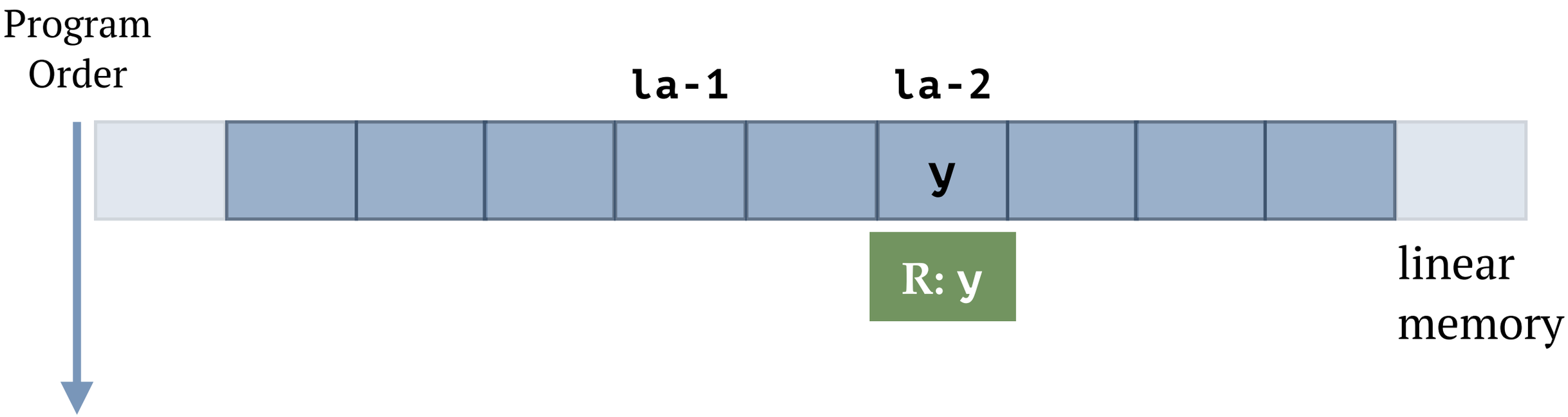
Outline

- Motivation
- Project Overview
- **IA-32e Paging**
  - **Specification**
  - **Verification**
- Future Work
- Conclusion

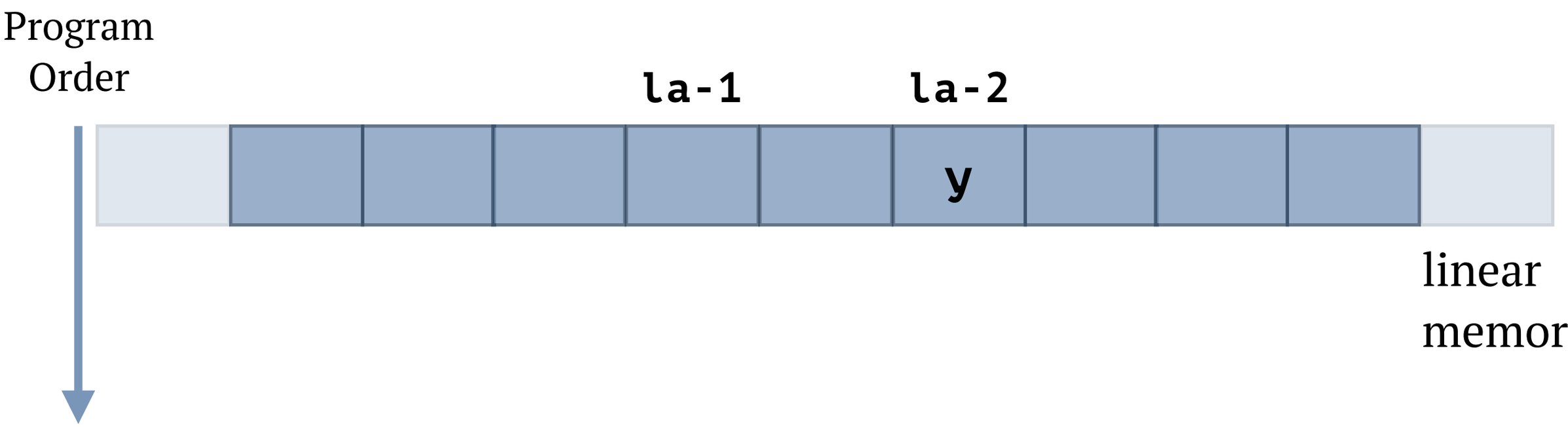
Linear Memory Non-Interference Theorem



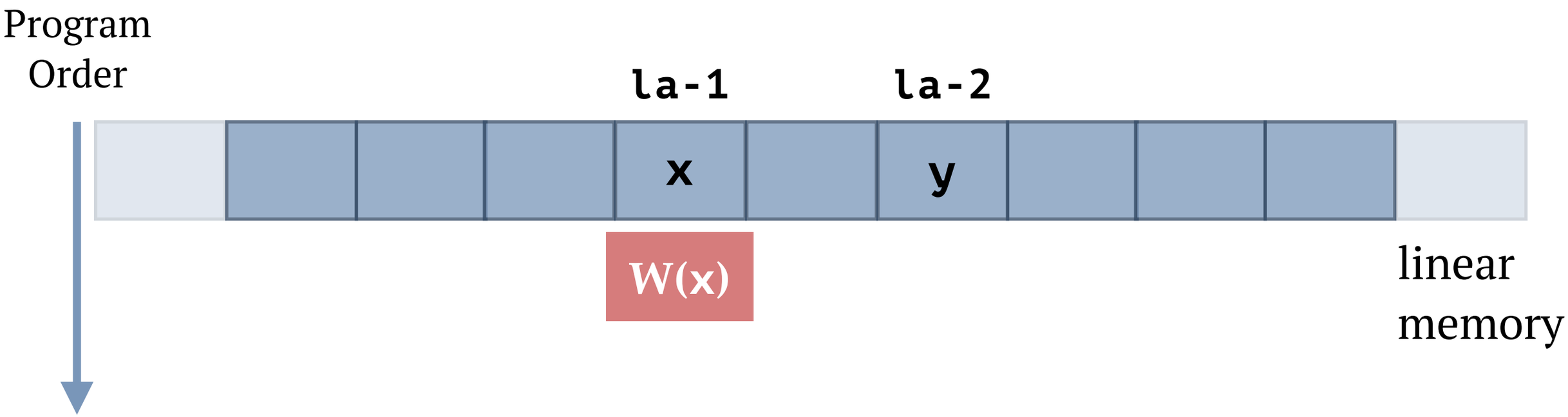
Linear Memory Non-Interference Theorem



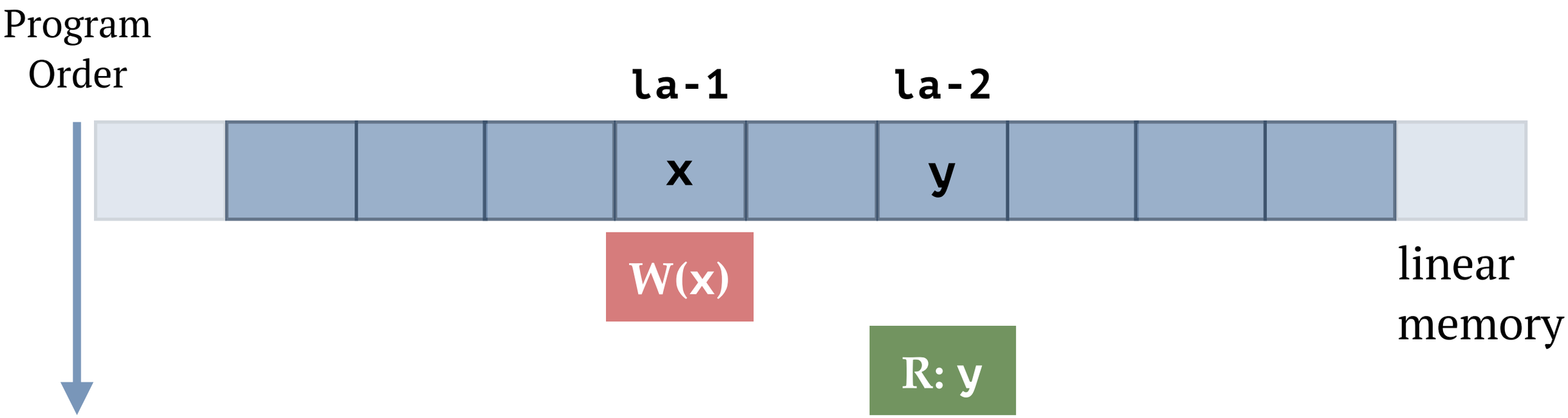
Linear Memory Non-Interference Theorem



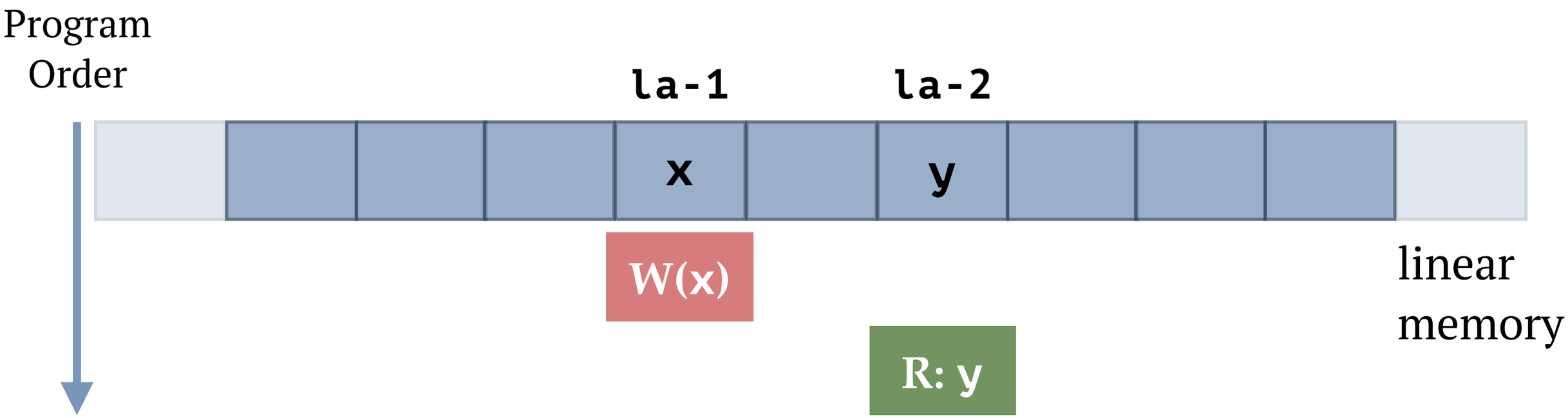
Linear Memory Non-Interference Theorem



Linear Memory Non-Interference Theorem



Linear Memory Non-Interference Theorem



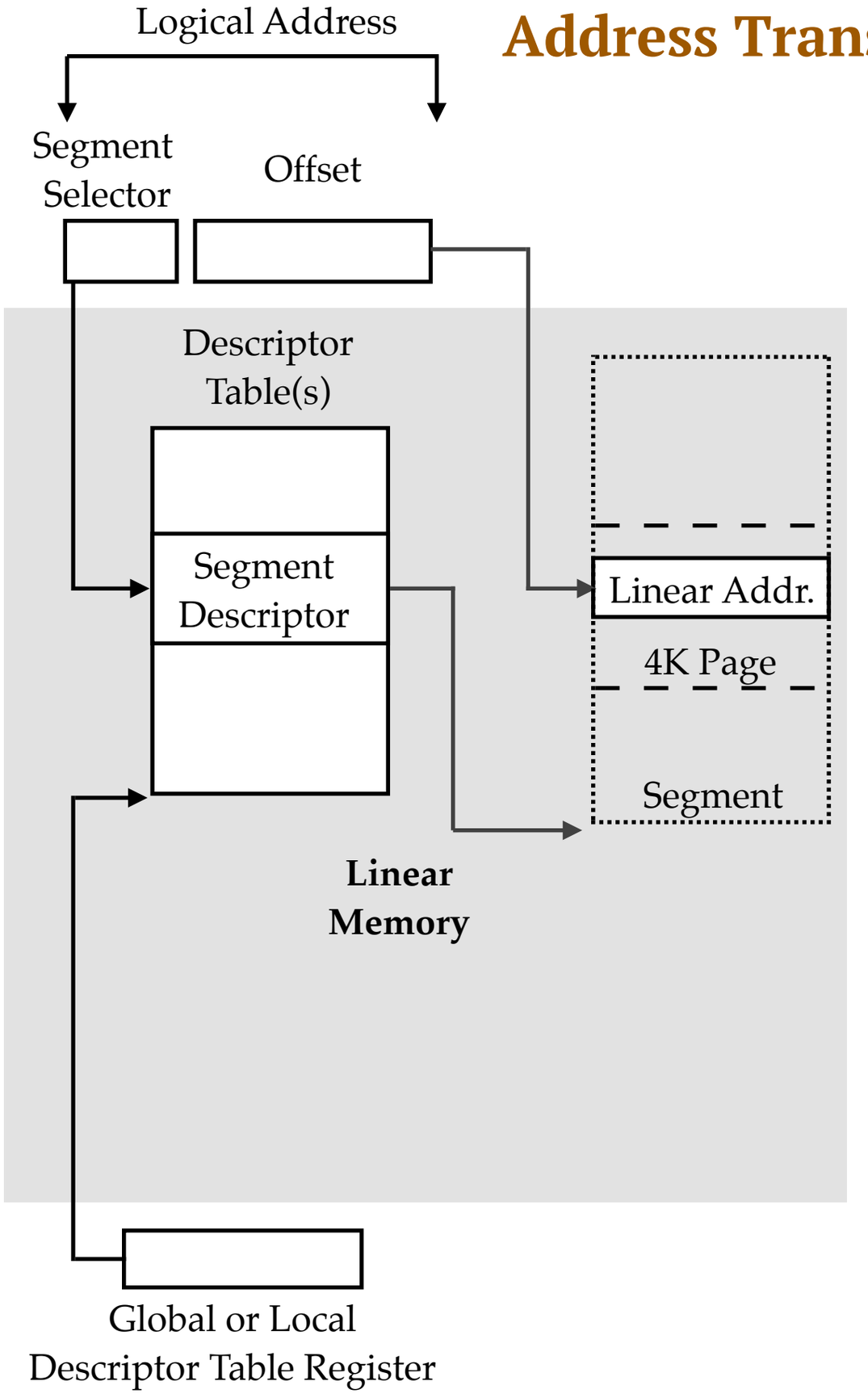
But, linear memory is an abstraction!

Does paging (h/w + s/w) provide this non-interference property?

Paging: A Brief Introduction

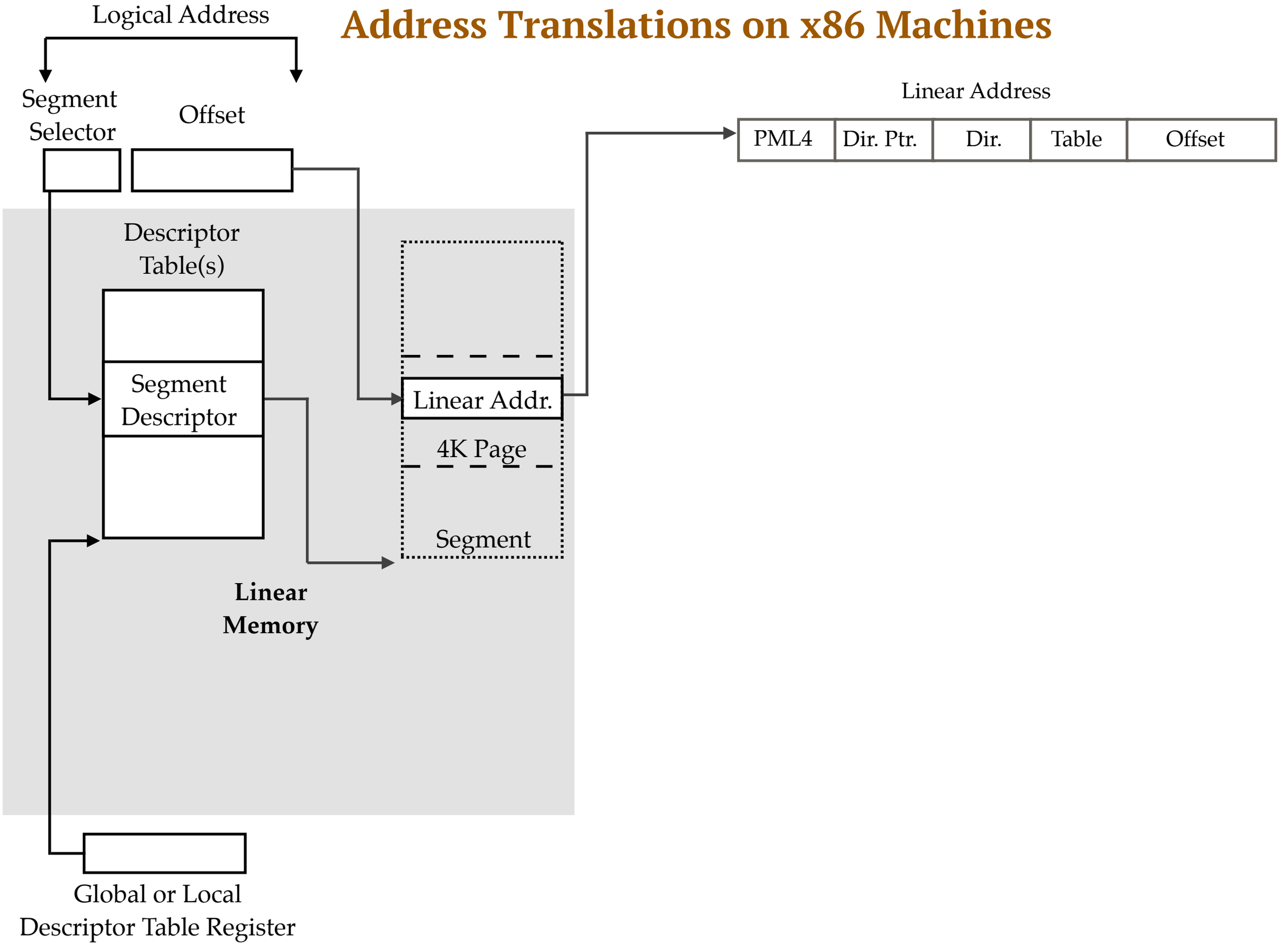
- Linear address space is divided into pages; an OS tracks these pages via **hierarchical data structures**.
- For every linear memory access, these structures are "walked" to obtain:
  - corresponding physical address
  - access rights
  - memory type
- A **page-fault exception** is generated if:
  - the required page is located in secondary storage
  - the access rights do not permit the memory access

Address Translations on x86 Machines



SEGMENTATION

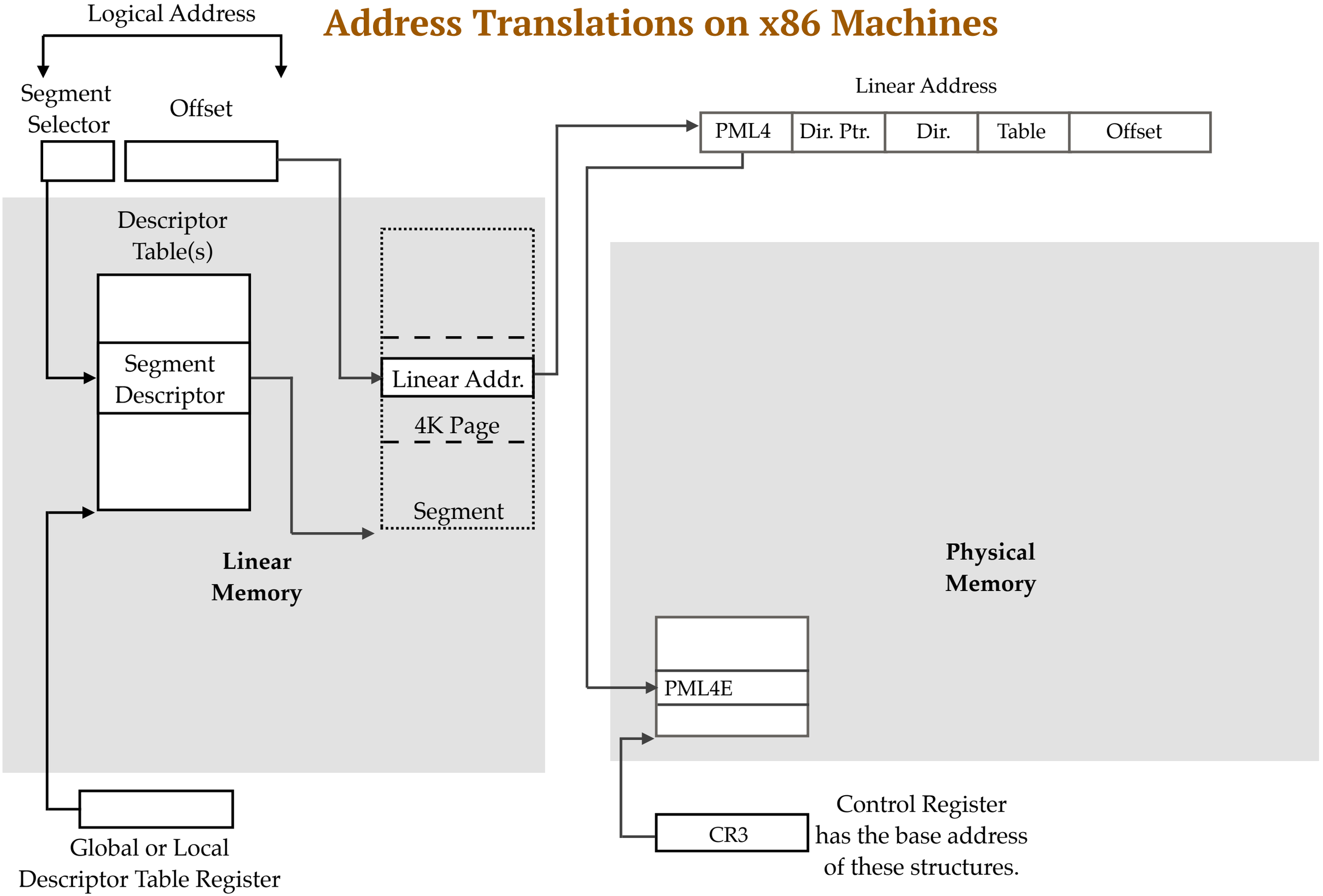
Address Translations on x86 Machines



SEGMENTATION

IA-32e PAGING (4K page)

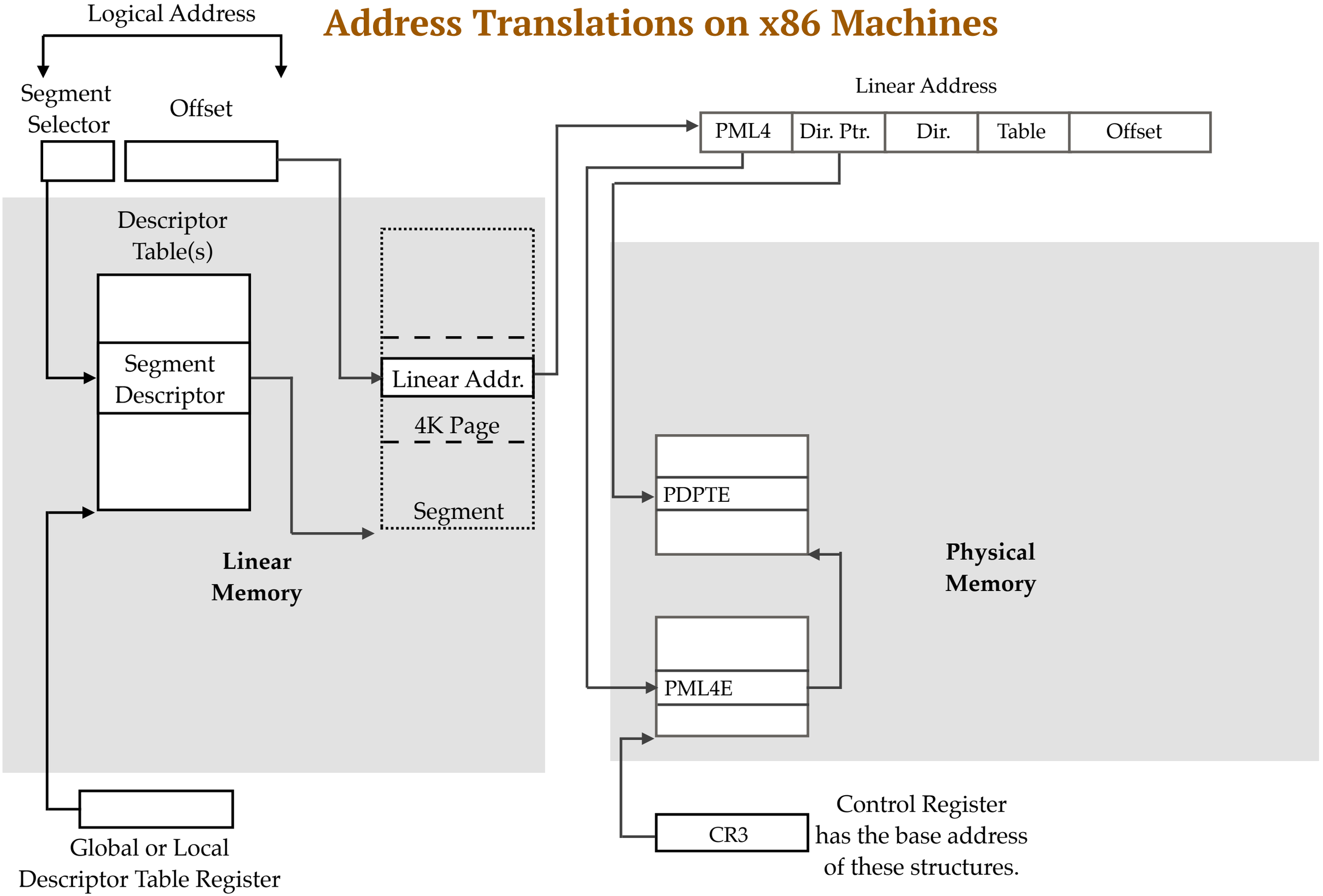
Address Translations on x86 Machines



SEGMENTATION

IA-32e PAGING (4K page)

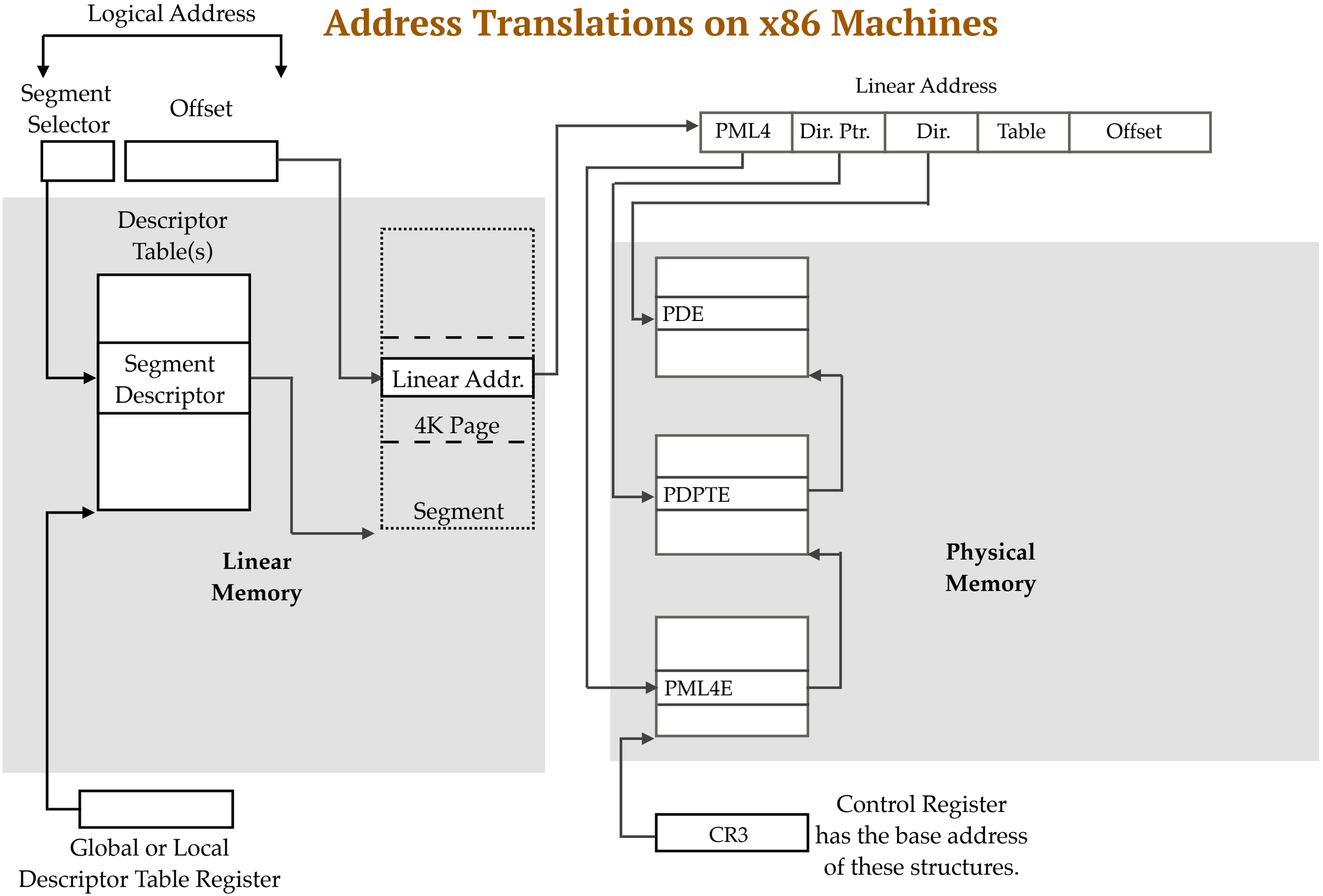
Address Translations on x86 Machines



SEGMENTATION

IA-32e PAGING (4K page)

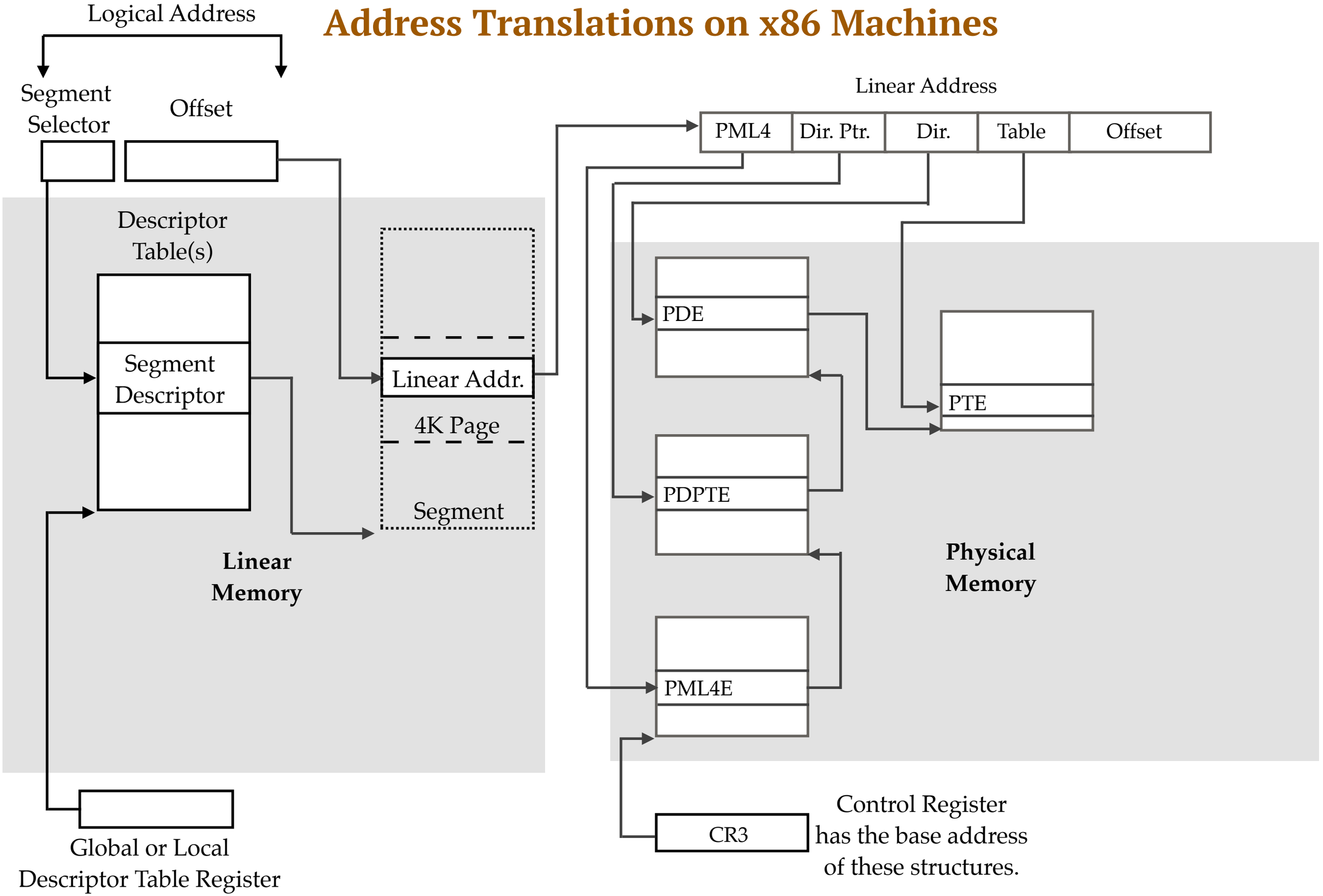
Address Translations on x86 Machines



SEGMENTATION

IA-32e PAGING (4K page)

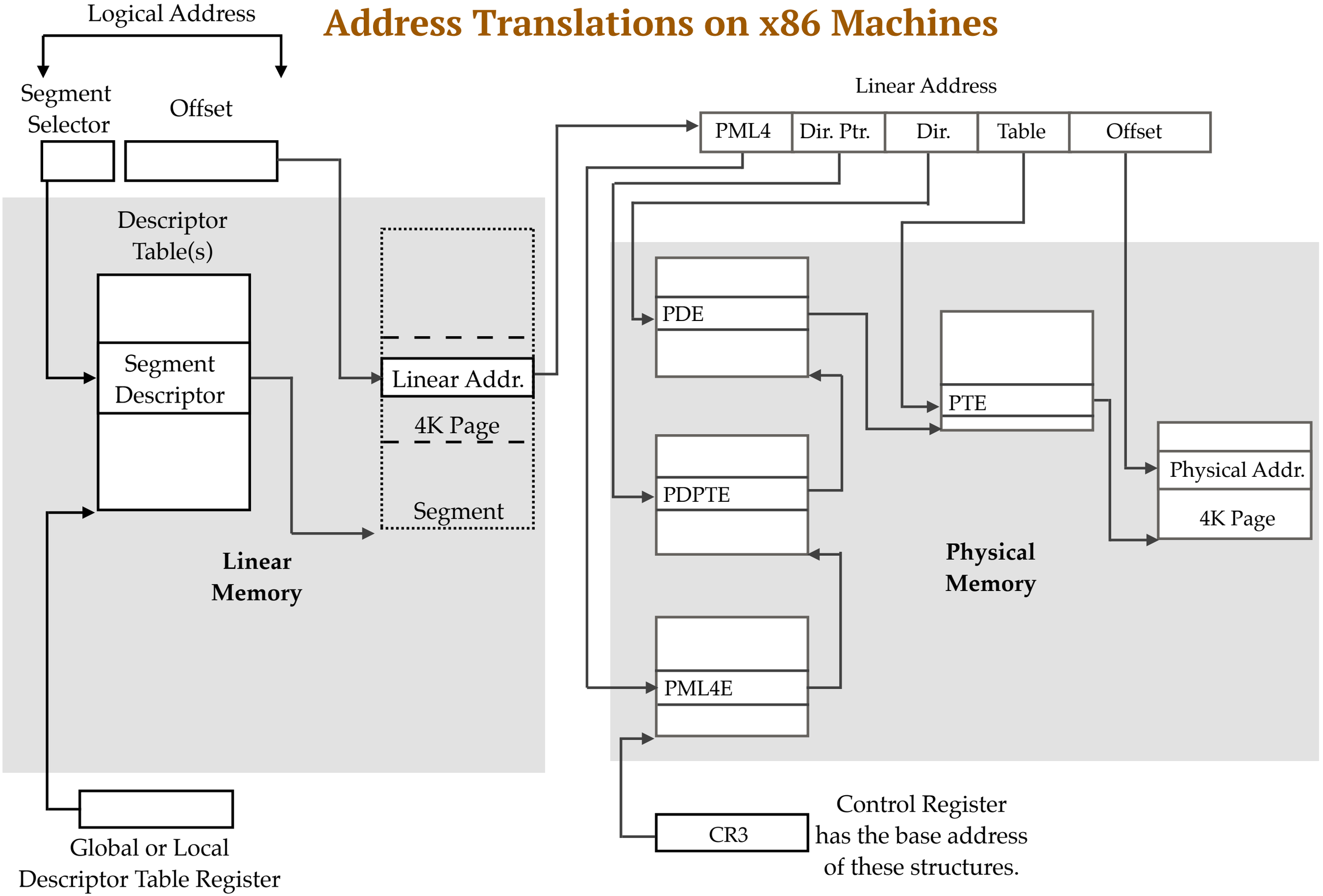
Address Translations on x86 Machines



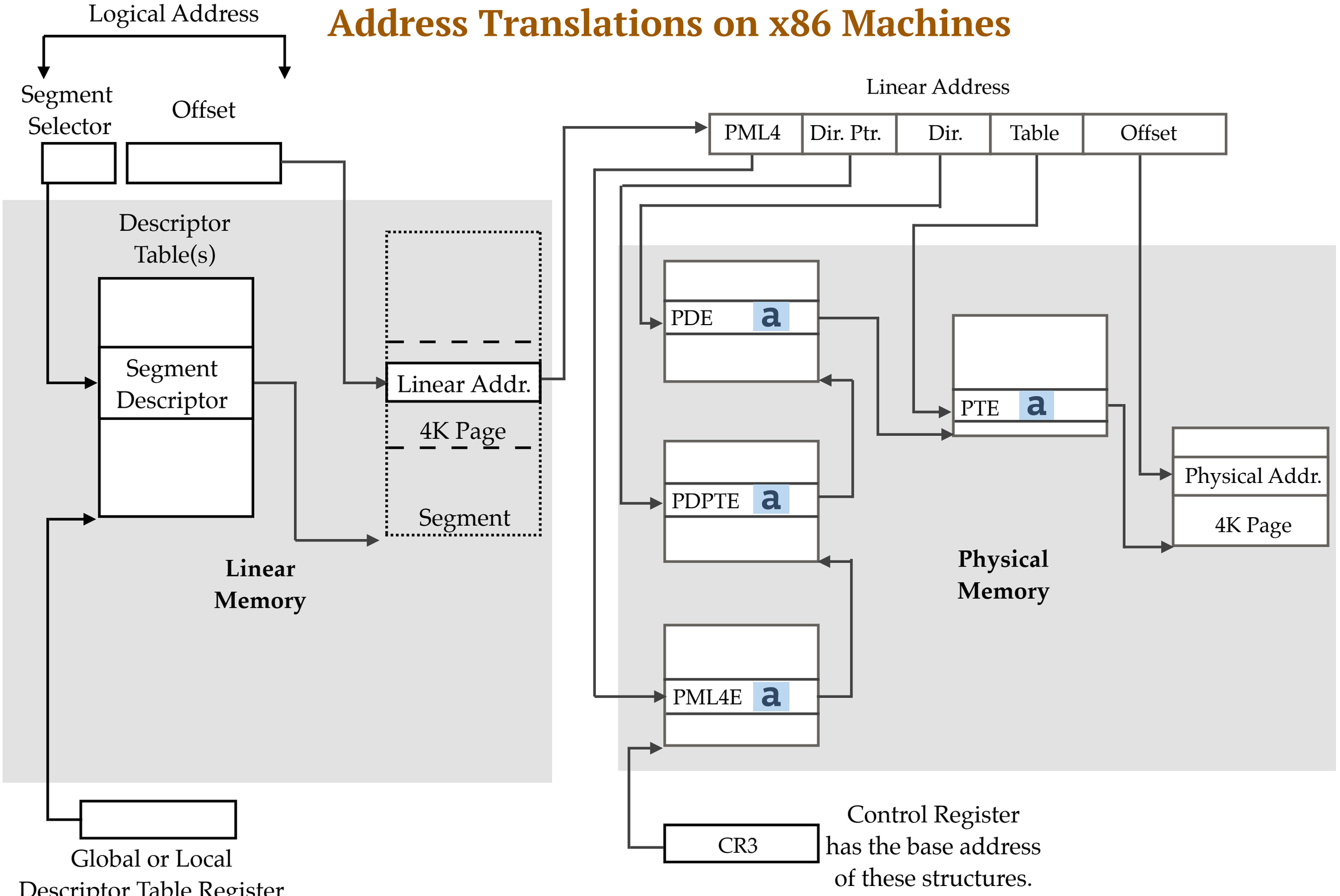
SEGMENTATION

IA-32e PAGING (4K page)

Address Translations on x86 Machines



Address Translations on x86 Machines

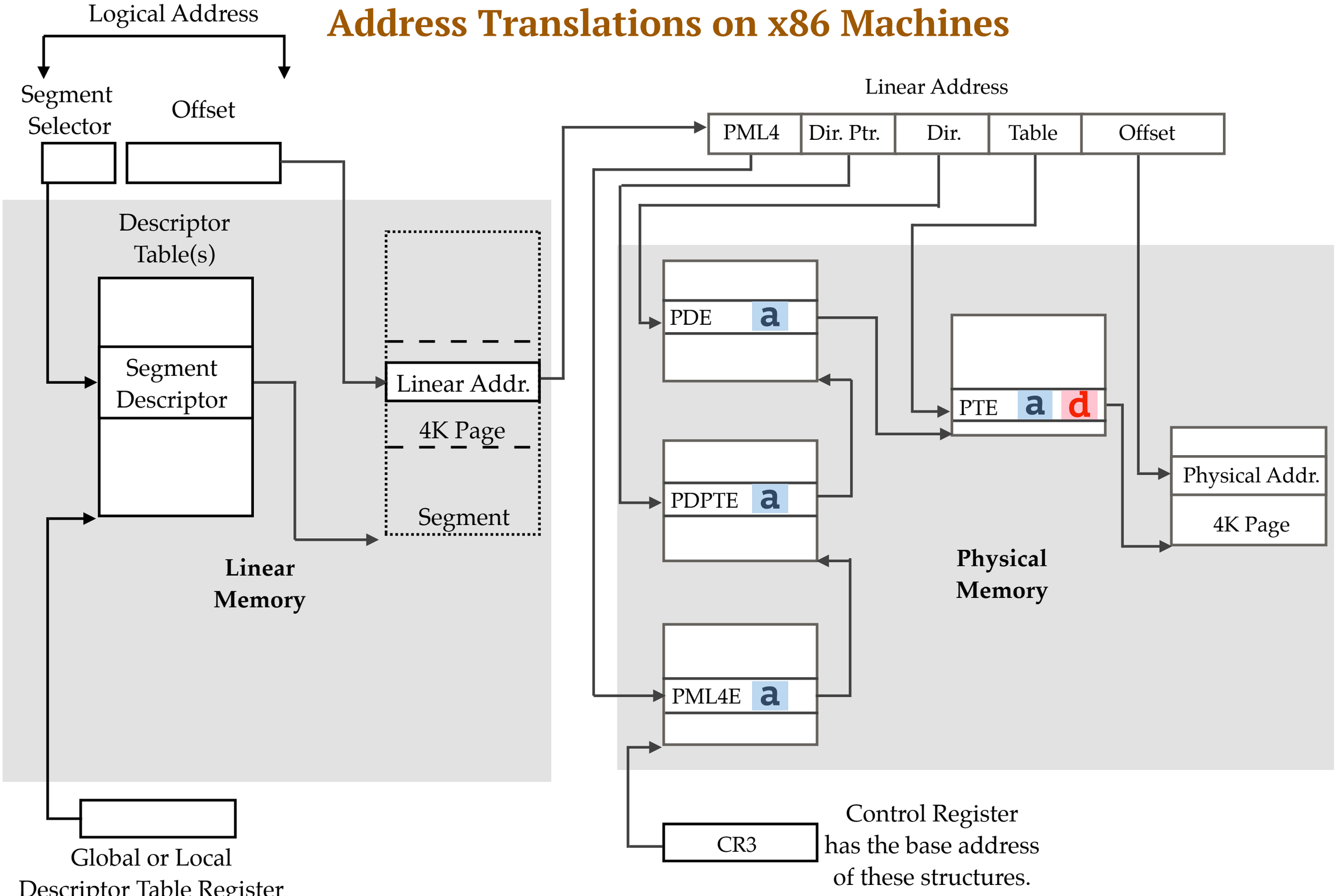


SEGMENTATION

a *accessed flag*

IA-32e PAGING (4K page)

Address Translations on x86 Machines



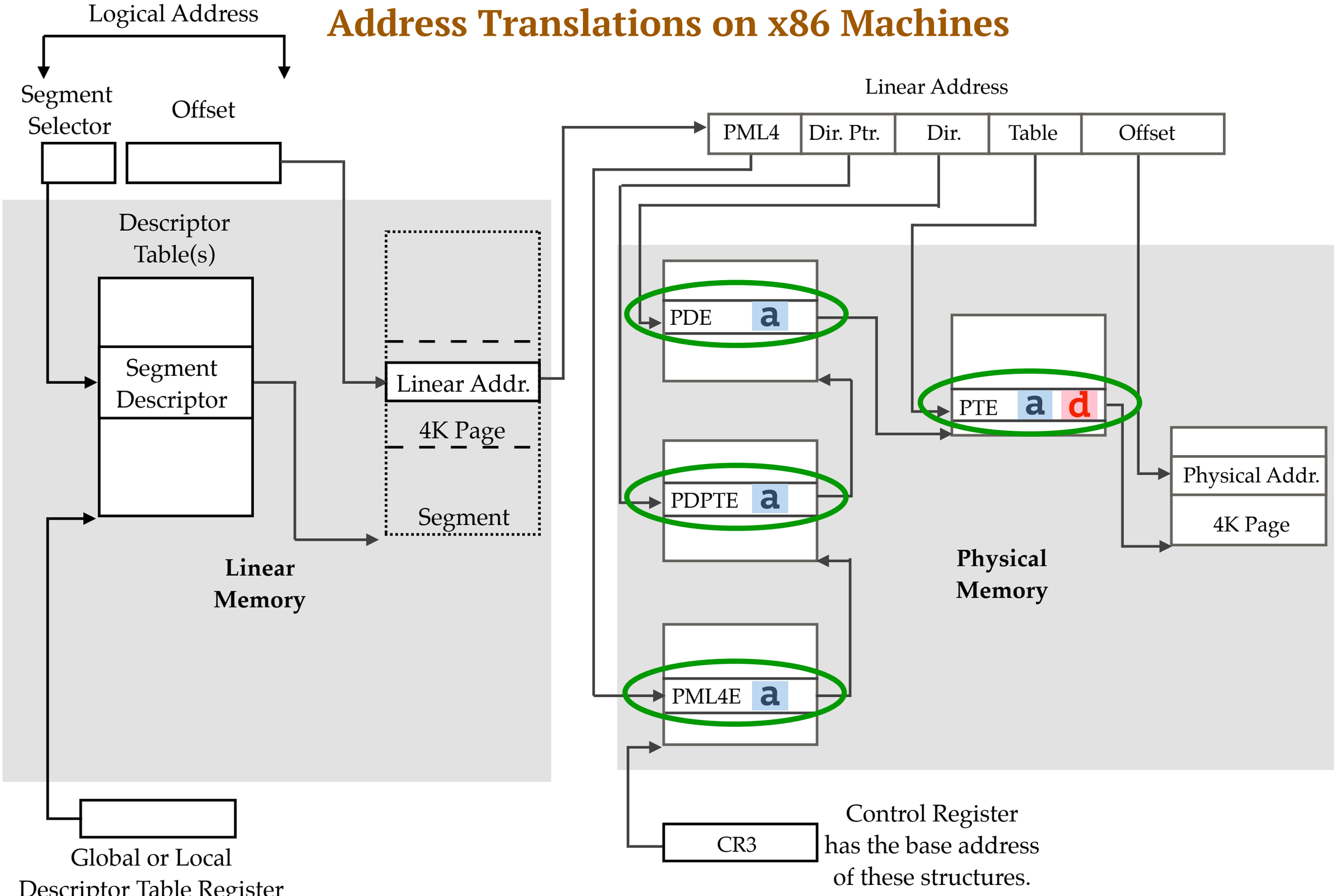
SEGMENTATION

a *accessed flag*

d *dirty flag*

IA-32e PAGING (4K page)

Address Translations on x86 Machines



SEGMENTATION

a *accessed flag*

d *dirty flag*

IA-32e PAGING (4K page)

63	62	61	60	59	58	57	56	55	54	53	52	51		M[1]	M-1			32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved[2]															Address of PML4 table																												Ignored							PCD	PWT	Ign.			CR3
XD	Ignored													Rsvd.	Address of page-directory-pointer table																												Ign.			Rsvd	Ign	A	PCD	PWT	U/S	R/W	1	PML4E: present
Ignored																																																				0	PML4E: not present
XD	Ignored													Rsvd.	Address of 1GB page frame																					Reserved							PAT	Ign.		G	1	D	A	PCD	PWT	U/S	R/W	1	PDPTE: 1GB page
XD	Ignored													Rsvd.	Address of page directory																												Ign.			0	Ign	A	PCD	PWT	U/S	R/W	1	PDPTE: page directory
Ignored																																																			0	PDTPE: not present
XD	Ignored													Rsvd.	Address of 2MB page frame																					Reserved							PAT	Ign.		G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 2MB page
XD	Ignored													Rsvd.	Address of page table																												Ign.			0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																																																			0	PDE: not present
XD	Ignored													Rsvd.	Address of 4KB page frame																												Ign.			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																																																					0	PTE: not present

Figure 4-11. Formats of CR3 and Paging-Structure Entries with IA-32e Paging

Accessed and Dirty Flags

Source: Intel Manuals, Vol. 3

Formal Specification of x86 Paging

Formal model of components in the x86 state:

- Physical memory (2^{52} bytes)
- Registers
  - Control Registers: `cr0`, `cr3`, `cr4`
  - Model-Specific Register: `ia32_efer`

Formal Specification of x86 Paging

Formal model of components in the x86 state:

- Physical memory (2^{52} bytes)
- Registers
  - Control Registers: `cr0`, `cr3`, `cr4`
  - Model-Specific Register: `ia32_efer`

Specification functions that access and update paging entries:

- Base address of the next data structure or page frame
- Fields related to page protection
  - User/supervisor, read/write, execute, etc.

Formal Specification of x86 Paging

Formal model of components in the x86 state:

- Physical memory (2^{52} bytes)
- Registers
  - Control Registers: `cr0`, `cr3`, `cr4`
  - Model-Specific Register: `ia32_efer`

Specification functions that access and update paging entries:

- Base address of the next data structure or page frame
- Fields related to page protection
  - User/supervisor, read/write, execute, etc.

Specification functions that recognize well-formed paging entries and structures

Formal Specification of a Linear Memory Read

```
lin-mem-read(l-addr, x86):
[ err?, p-addr, x86] := la-to-pa(l-addr, x86)
if (err?) then
  go to exception handling routine
else
  val := read-mem(p-addr, x86)
  return(val, x86)
end if
```

Formal Specification of a Linear Memory Write

```
lin-mem-write(l-addr, val, x86):
[ err?, p-addr, x86] := la-to-pa(l-addr, x86)
if (err?) then
  go to exception handling routine
else
  x86 := write-mem(p-addr, val, x86)
  return(x86)
end if
```

The Reality: Walking the "Lowest" Structure

```
(define la-to-pa-page-table
  ((lin-addr  :type (signed-byte   #.*max-linear-address-size*))
   (base-addr :type (unsigned-byte #.*physical-address-size*))
   (u-s-acc   :type (unsigned-byte  1))
   (wp        :type (unsigned-byte  1))
   (smep      :type (unsigned-byte  1))
   (nxe       :type (unsigned-byte  1))
   (r-w-x     :type (member  :r :w :x))
   (cpl       :type (unsigned-byte  2))
   (x86))

  (b* ((p-entry-addr
        (the (unsigned-byte #.*physical-address-size*)
          (page-table-entry-addr lin-addr base-addr)))
       (entry (the (unsigned-byte 64) (rm-low-64 p-entry-addr x86)))

       (page-present (page-tables-slice :p entry))
       ((when (equal page-present 0))
        (let ((err-no (page-fault-err-no
                       page-present r-w-x cpl
                       0      ;; rsvd
                       smep 1 ;; pae
                       nxe)))
          (page-fault-exception lin-addr err-no x86)))
       (read-write      (page-tables-slice :r/w entry))
       (user-supervisor (page-tables-slice :u/s entry))
       (execute-disable (page-tables-slice :xd  entry))

       (rsvd
        (mbe
         :logic
         (if (or
              (not (equal
                    (part-select entry :low
                                 *physical-address-size* :high 62)
                    0))
              (and (equal nxe 0)
                   (not (equal (pte-4K-page-slice :pte-xd entry)
                               0))))
             1 0)
         :exec
         (if (or
              (not (equal (logand (+ -1 (ash 1 (+ 63 (- #.*physical-address-size*))))
                                  (the (unsigned-byte 28)
                                    (ash entry (- #.*physical-address-size*))))
                          0))
              (and (equal nxe 0)
                   (not (equal
                         (the (unsigned-byte 1)
                           (logand 1
                                   (the (unsigned-byte 1)
                                     (ash entry (- 63)))))
                         0))))
             1 0)))

       ((when (equal rsvd 1))
        (let ((err-no (page-fault-err-no page-present
                                         r-w-x
                                         cpl
                                         rsvd
                                         smep
                                         1 ;; pae
                                         nxe)))
          (page-fault-exception lin-addr err-no x86)))

       ((when (or (and (equal r-w-x :r)
                       (if (< cpl 3)
                           nil
                         (equal user-supervisor 0)))
                  (and (equal r-w-x :w)
                       (if (< cpl 3)
                           (and (equal wp 1)
                                (equal read-write 0))
                         (or (equal user-supervisor 0)
                             (equal read-write 0))))
                  (and (equal r-w-x :x)
                       (if (< cpl 3)
                           (if (equal nxe 0)
                               (and (equal smep 1)
                                    (equal u-s-acc 1)
                                    (equal user-supervisor 1))
                             (if (equal smep 0)
                                 (equal execute-disable 1)
                               (or (equal execute-disable 1)
                                   (and (equal u-s-acc 1)
                                        (equal user-supervisor 1)))))
                         (or (equal user-supervisor 0)
                             (and (equal nxe 1)
                                  (equal execute-disable 1)))))))
        (let ((err-no (page-fault-err-no page-present
                                         r-w-x
                                         cpl
                                         rsvd
                                         smep
                                         1 ;; pae
                                         nxe)))
          (page-fault-exception lin-addr err-no x86)))

       ;; No errors, so we proceed with the address translation.
```

The Reality: Walking the “Lowest” Structure

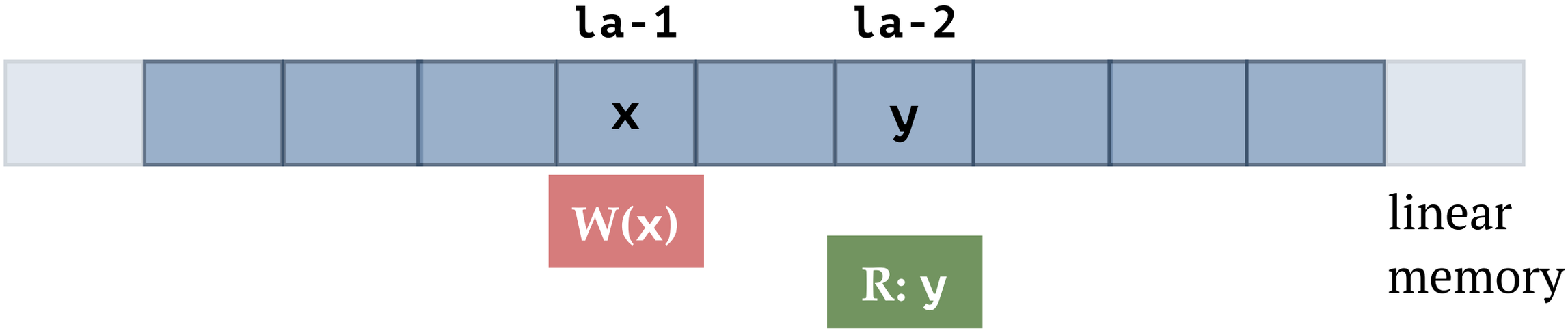
```
 ;; Get accessed and dirty bits:
 (accessed       (page-tables-slice :a entry))
 (dirty          (page-tables-slice :d entry))
 ;; Compute accessed and dirty bits:
 (entry (if (equal accessed 0)
            (!page-tables-slice :a 1 entry)
          entry))
 (entry (if (and (equal dirty 0)
                 (equal r-w-x :w))
            (!page-tables-slice :d 1 entry)
          entry))
 ;; Update x86 (to reflect accessed and dirty bits change), if needed:
 (x86 (if (or (equal accessed 0)
              (and (equal dirty 0)
                   (equal r-w-x :w)))
          (wm-low-64 p-entry-addr entry x86)
        x86)))

;; Return address of 4KB page frame and the modified x86 state.
(mv nil
    (mbe

     :logic
     (part-install
      (part-select lin-addr :low 0 :high 11)
      (ash (pte-4K-page-slice :pte-page entry) 12)
      :low 0 :high 11)

     :exec
     (the (unsigned-byte #.*physical-address-size*)
       (logior
        (the (unsigned-byte #.*physical-address-size*)
          (logand
           (the (unsigned-byte #.*physical-address-size*)
             (ash
              (the (unsigned-byte 40)
                (logand (the (unsigned-byte 40) 1099511627775)
                        (the (unsigned-byte 52)
                          (ash (the (unsigned-byte 64) entry)
                               (- 12)))))
              12))
           -4096))
        (the (unsigned-byte 12)
          (logand 4095 lin-addr)))))
    x86)))
```

The Reality: Walking the "Lowest" Structure

```
 ;; Get accessed and dirty bits:
 (accessed       (page-tables-slice :a entry))
 (dirty          (page-tables-slice :d entry))
 ;; Compute accessed and dirty bits:
 (entry (if (equal accessed 0)
            (!page-tables-slice :a 1 entry)
          entry))
 (entry (if (and (equal dirty 0)
                 (equal r-w-x :w))
            (!page-tables-slice :d 1 entry)
          entry))
 ;; Update x86 (to reflect accessed and dirty bits change), if needed:
 (x86 (if (or (equal accessed 0)
              (and (equal dirty 0)
                   (equal r-w-x :w)))
          (wm-low-64 p-entry-addr entry x86)
        x86)))

;; Return address of 4KB page frame and the modified x86 state.
(mv nil
    (mbe

     :logic
     (part-install
      (part-select lin-addr :low 0 :high 11)
      (ash (pte-4K-page-slice :pte-page entry) 12)
      :low 0 :high 11)

     :exec
     (the (unsigned-byte #.*physical-address-size*)
       (logior
        (the (unsigned-byte #.*physical-address-size*)
          (logand
           (the (unsigned-byte #.*physical-address-size*)
             (ash
              (the (unsigned-byte 40)
                (logand (the (unsigned-byte 40) 1099511627775)
                        (the (unsigned-byte 52)
                          (ash (the (unsigned-byte 64) entry)
                               (- 12)))))
              12))
           -4096))
        (the (unsigned-byte 12)
          (logand 4095 lin-addr)))))
    x86)))
```

There are FOUR more specification functions that are used to specify `la-to-pa`.

Linear Memory Non-Interference Theorem




```
let

 [y, x86₁]    := lin-mem-read(la-1, x86)

 x86₂         := lin-mem-write(la-2, x, x86)
 [y′, x86₃]   := lin-mem-read(la-1, x86₂)

then

    y == y′
```

Non-Interference: Paging Data Structure Entries

[Theorem]

Irrespective of the state of the accessed and dirty flags, walking a paging data structure entry will not affect any operation that occurs at another entry.

63	62	61	60	59	58	57	56	55	54	53	52	51		M[1]	M-1			32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved[2]															Address of PML4 table																												Ignored							PCD	PWT	Ign.			CR3
XD	Ignored													Rsvd.	Address of page-directory-pointer table																												Ign.			Rsvd	Ign	A	PCD	PWT	U/S	R/W	1	PML4E: present
Ignored																																																				0	PML4E: not present
XD	Ignored													Rsvd.	Address of 1GB page frame																					Reserved							PAT	Ign.		G	1	D	A	PCD	PWT	U/S	R/W	1	PDPTE: 1GB page
XD	Ignored													Rsvd.	Address of page directory																												Ign.			0	Ign	A	PCD	PWT	U/S	R/W	1	PDPTE: page directory
Ignored																																																			0	PDTPE: not present
XD	Ignored													Rsvd.	Address of 2MB page frame																					Reserved							PAT	Ign.		G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 2MB page
XD	Ignored													Rsvd.	Address of page table																												Ign.			0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																																																			0	PDE: not present
XD	Ignored													Rsvd.	Address of 4KB page frame																												Ign.			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																																																					0	PTE: not present

Figure 4-11. Formats of CR3 and Paging-Structure Entries with IA-32e Paging

Accessed and Dirty Flags

Source: Intel Manuals, Vol. 3

Non-Interference: Paging Data Structure Entries

[Theorem]

Irrespective of the state of the accessed and dirty flags, walking a paging data structure entry will not affect any operation that occurs at another entry.

Non-Interference: Paging Data Structure Entries

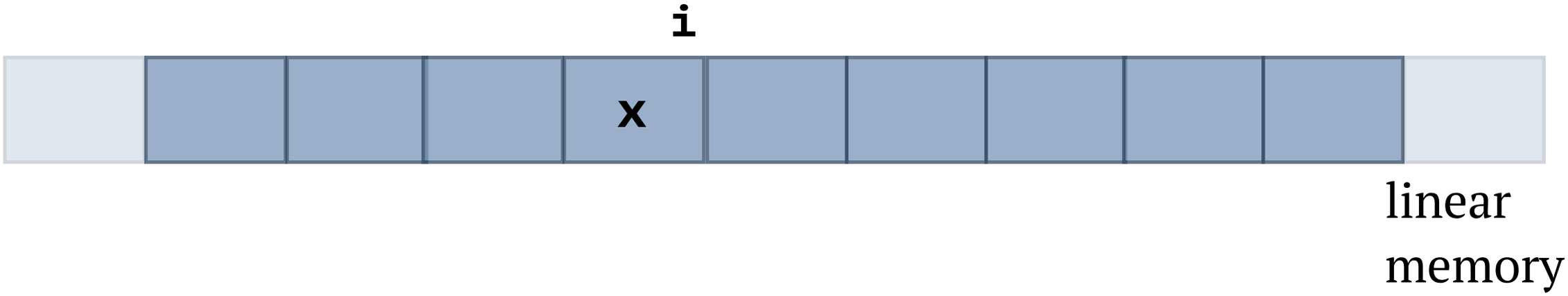
[Theorem]

Irrespective of the state of the accessed and dirty flags, walking a paging data structure entry will not affect any operation that occurs at another entry.

Involves:

1. Formulating predicates that recognize **valid paging entries and walks**
2. Reasoning about **non-interference** of **paging data structures**
3. Reasoning about **non-interference** of **sub-fields of each paging entry**

Linear Memory Preservation Theorems



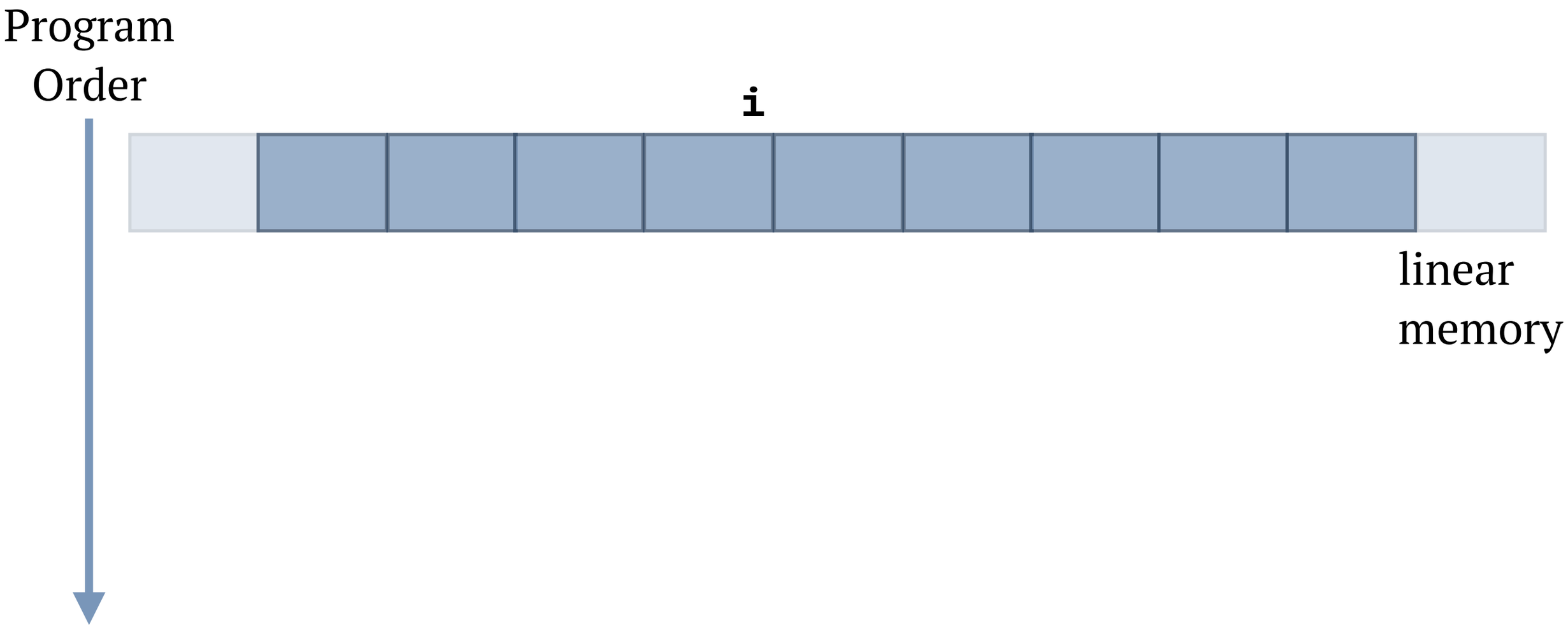
reading from a valid x86 state

```
valid-address-p(i) ∧
valid-x86-p(x86)
⇒
valid-value-p( Rᵢ: x ) ∧
valid-x86-p(x86)
```

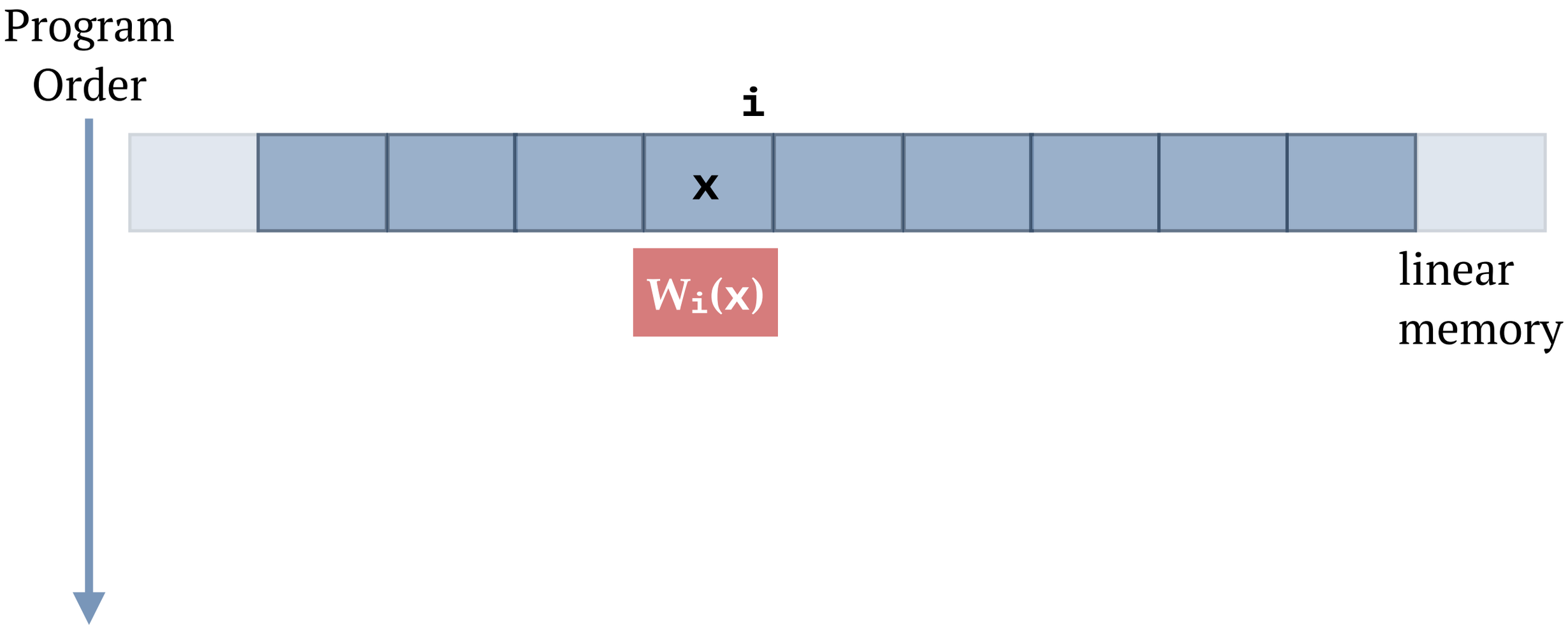
writing to a valid x86 state

```
valid-address-p(i) ∧
valid-value-p(x) ∧
valid-x86-p(x86)
⇒
valid-x86-p( Wᵢ(x) )
```

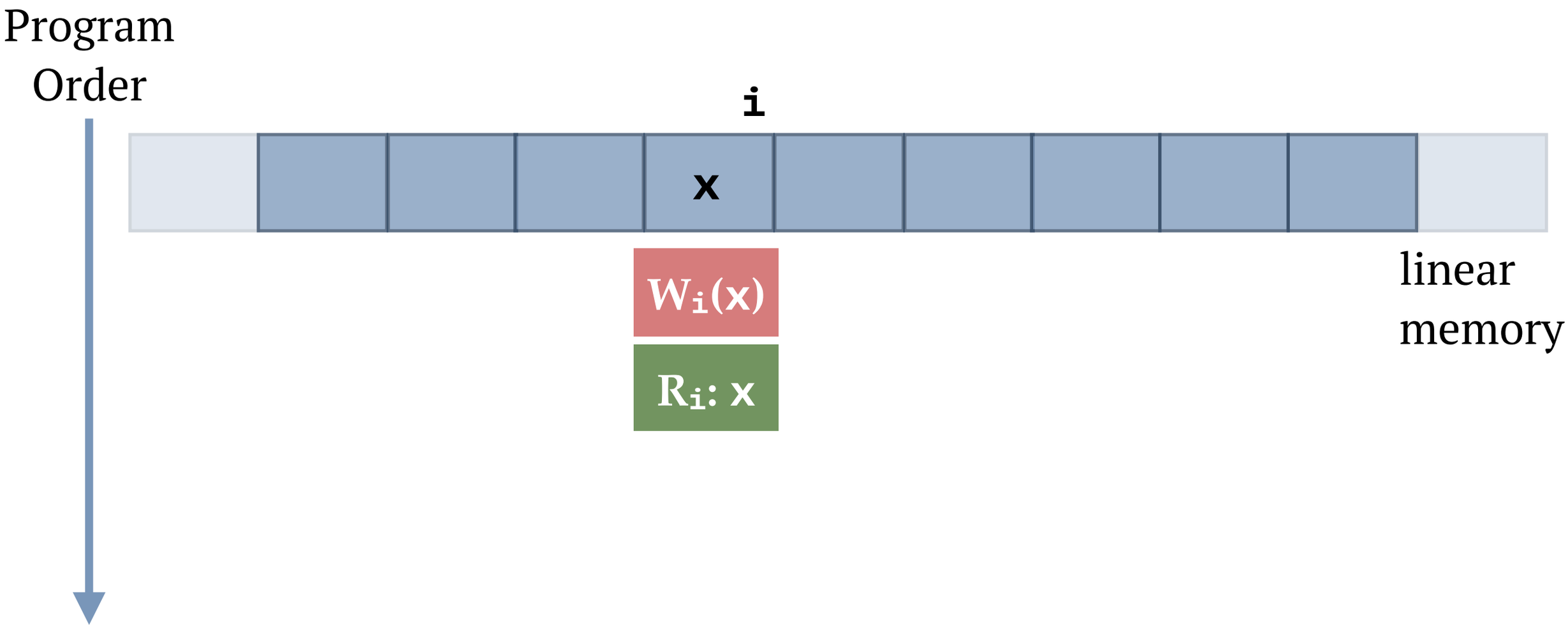
Linear Memory Interference Theorem



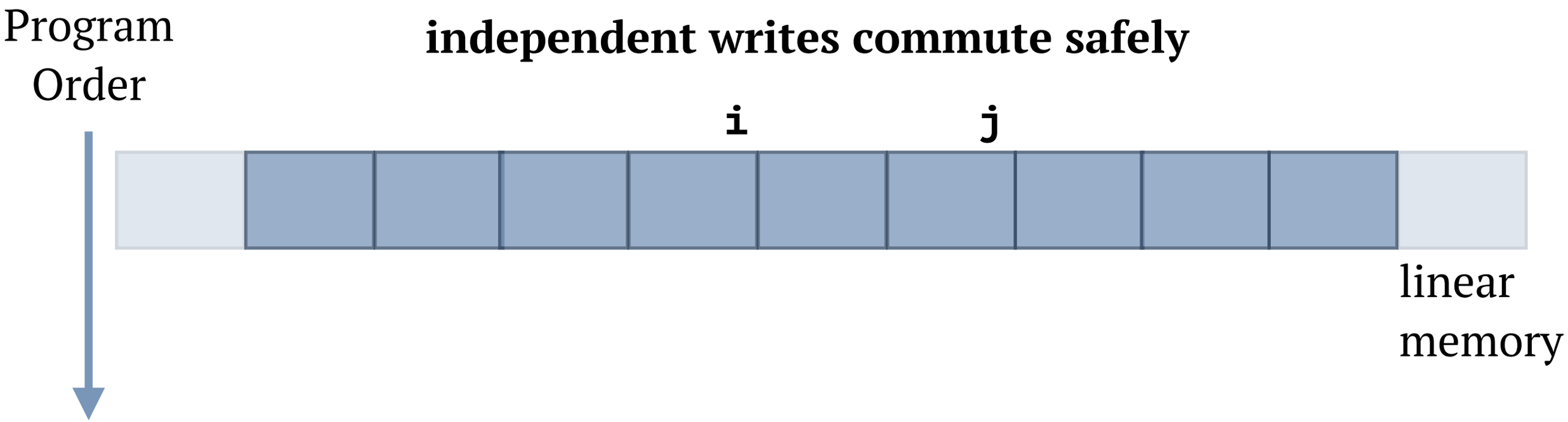
Linear Memory Interference Theorem



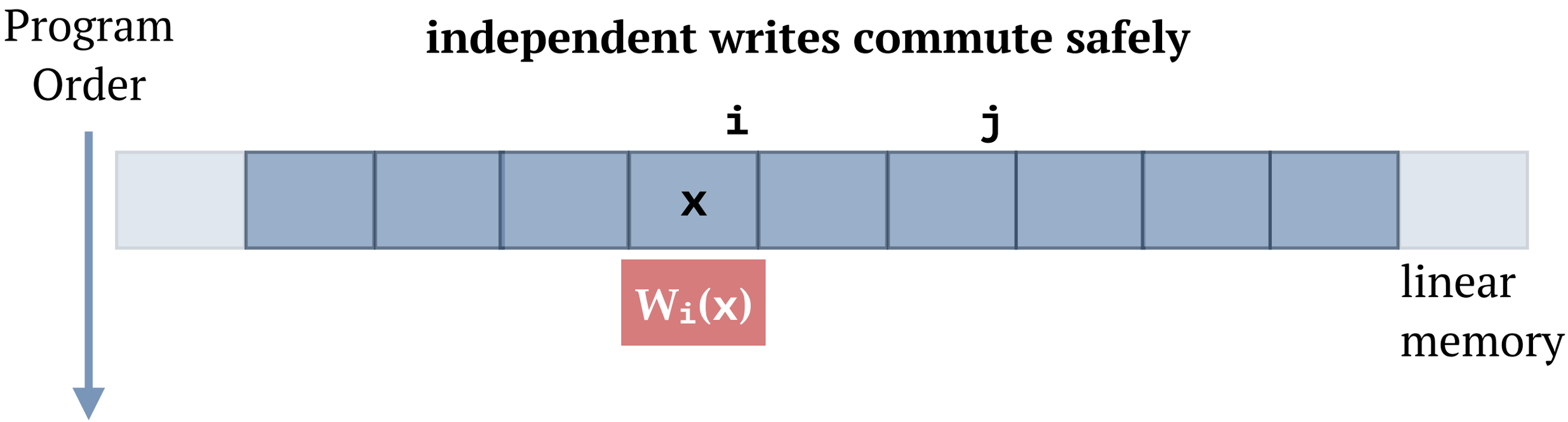
Linear Memory Interference Theorem



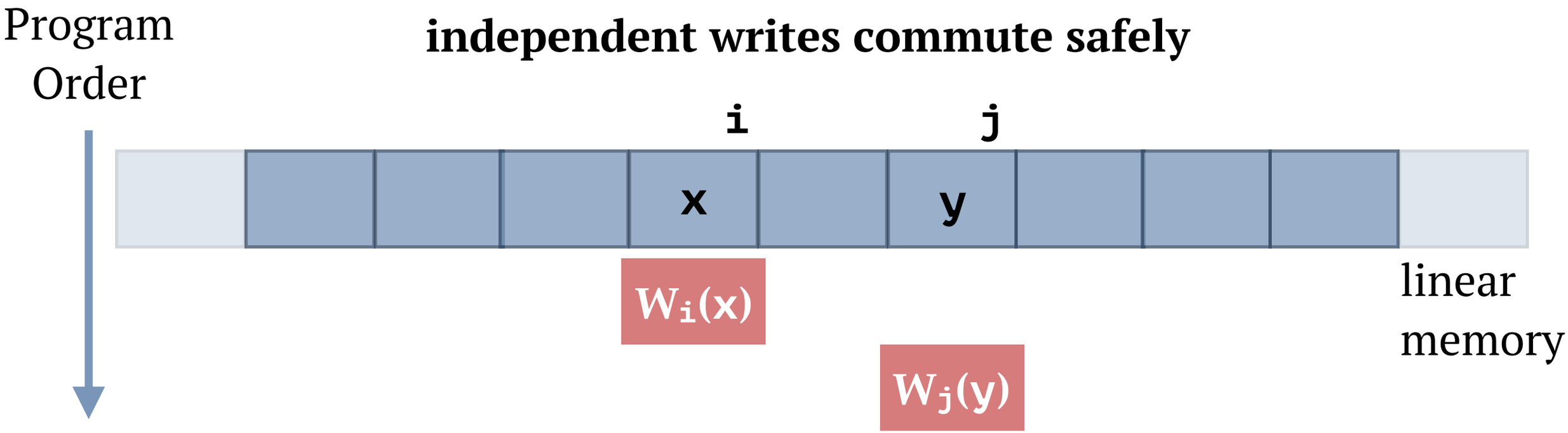
Linear Memory Write-over-Write Theorem: #1



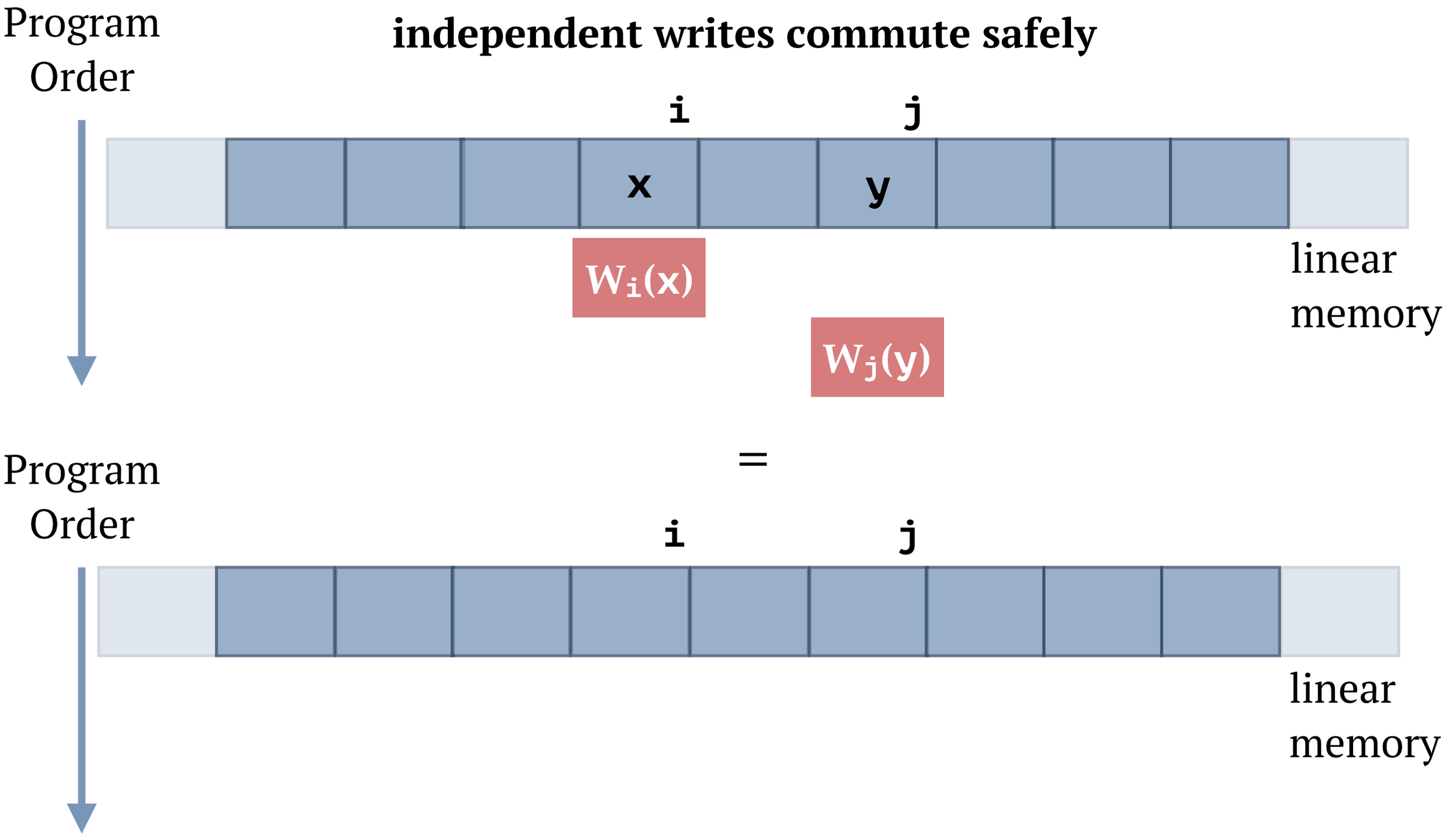
Linear Memory Write-over-Write Theorem: #1



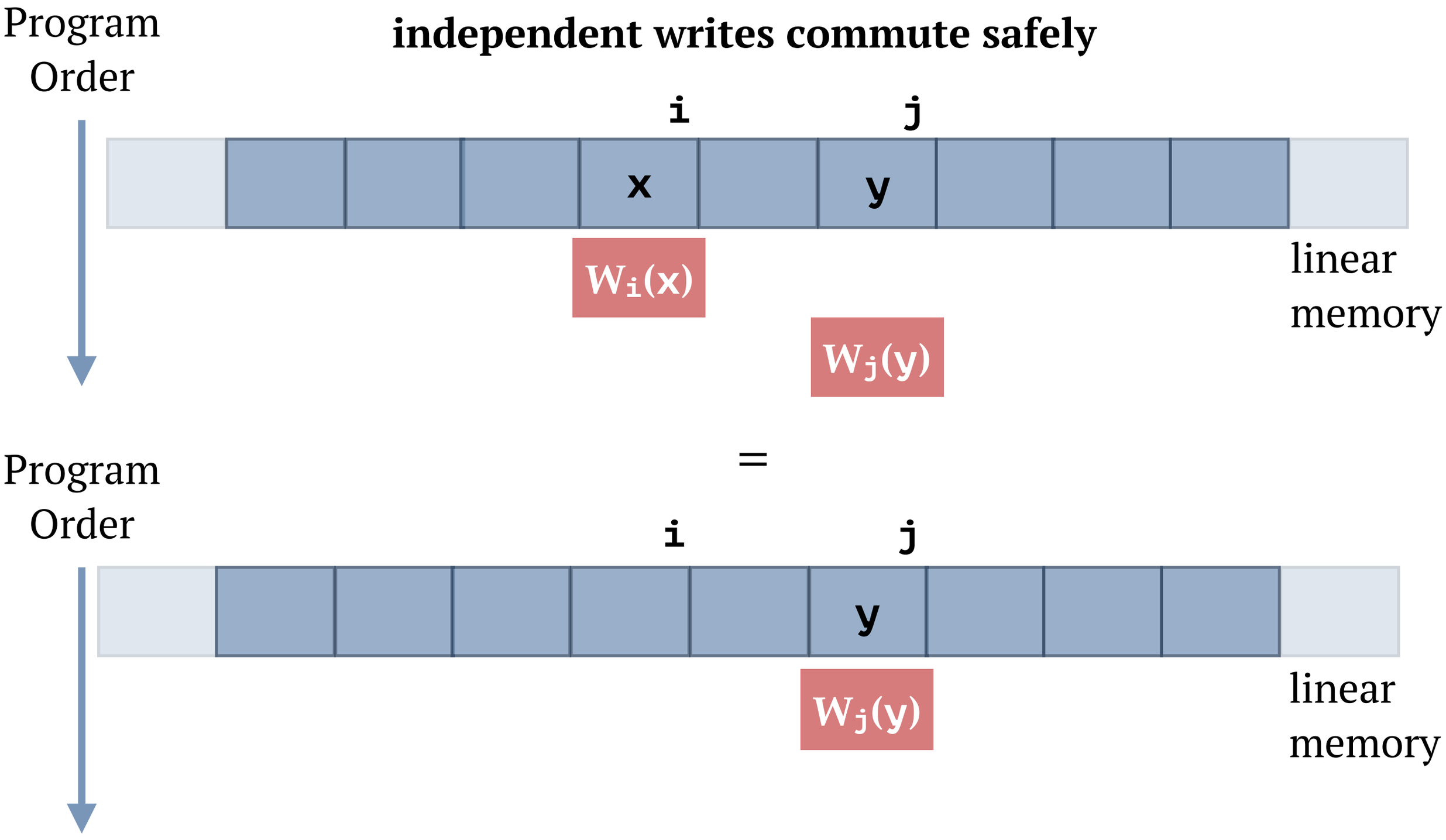
Linear Memory Write-over-Write Theorem: #1



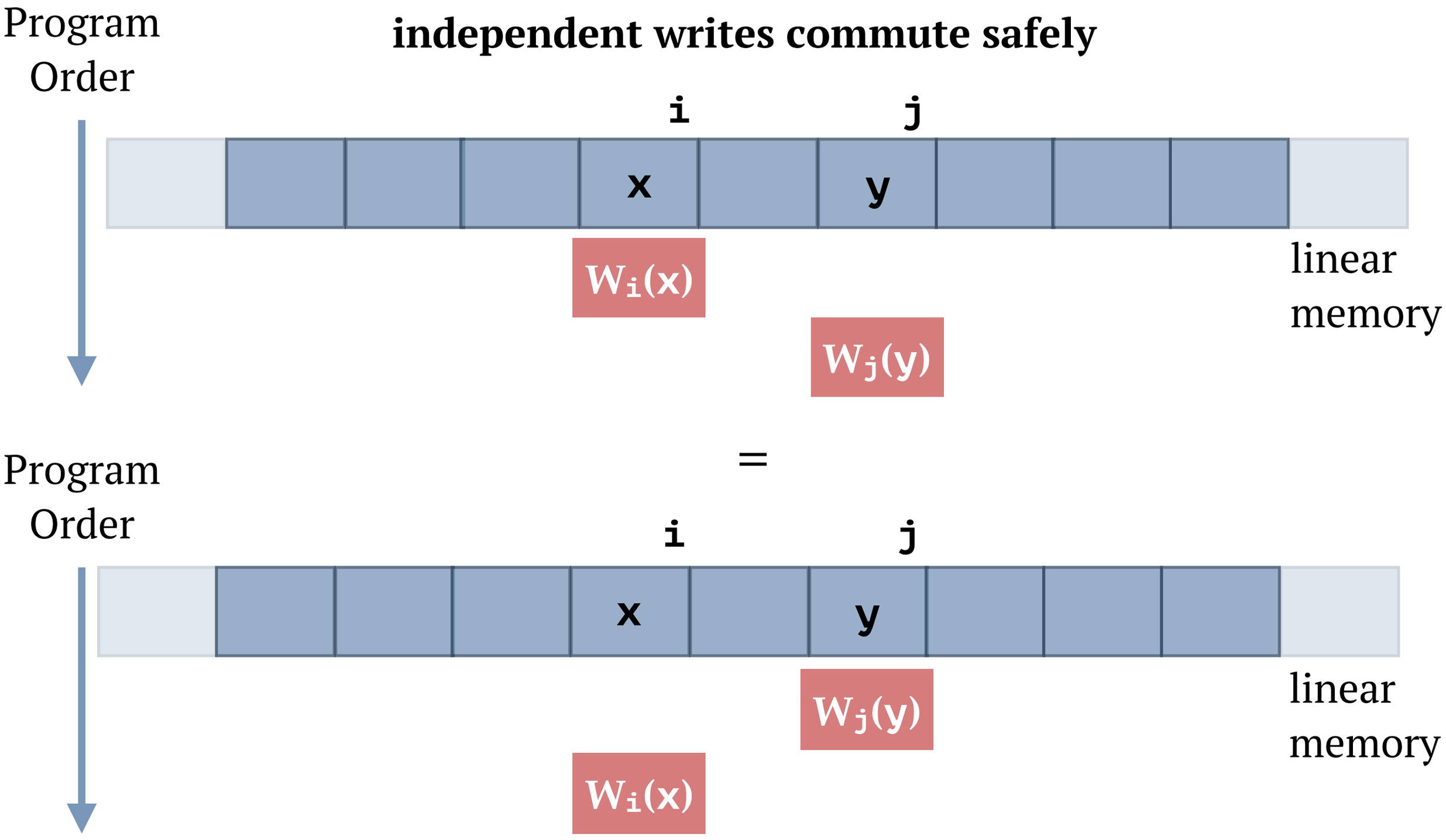
Linear Memory Write-over-Write Theorem: #1



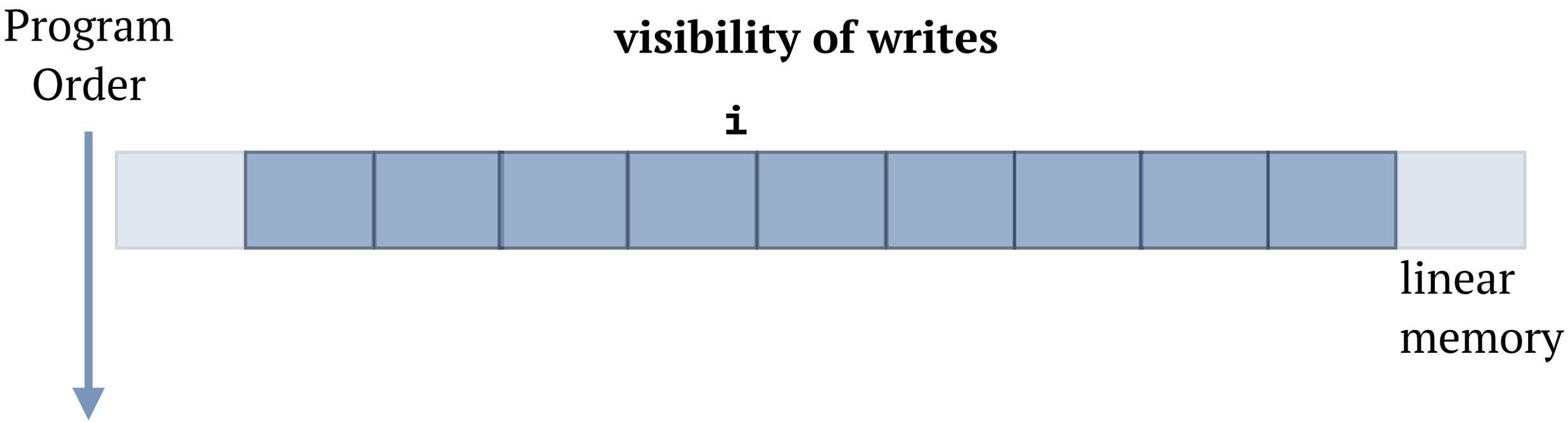
Linear Memory Write-over-Write Theorem: #1



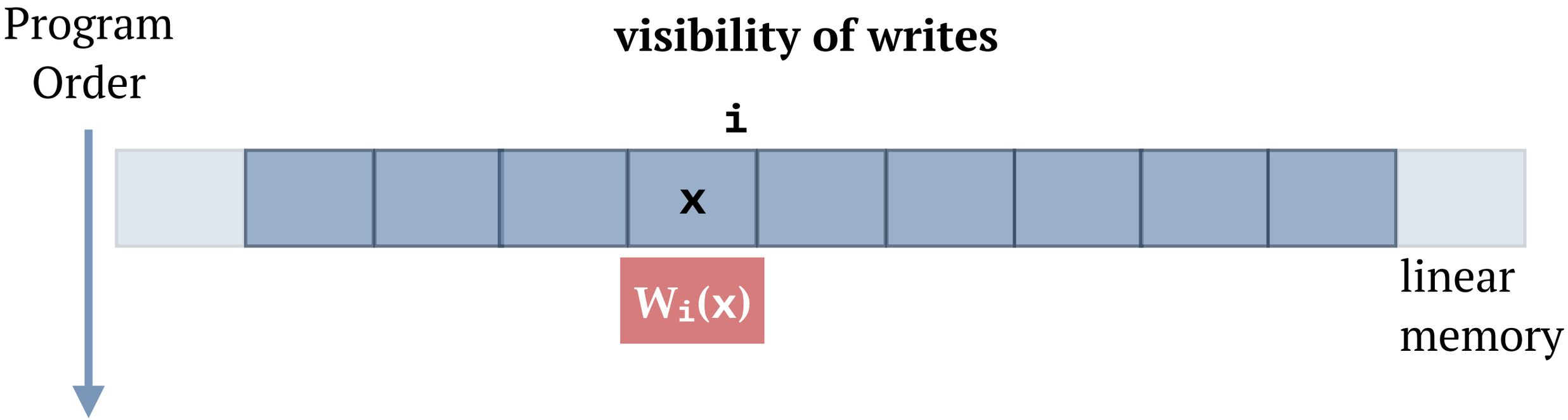
Linear Memory Write-over-Write Theorem: #1



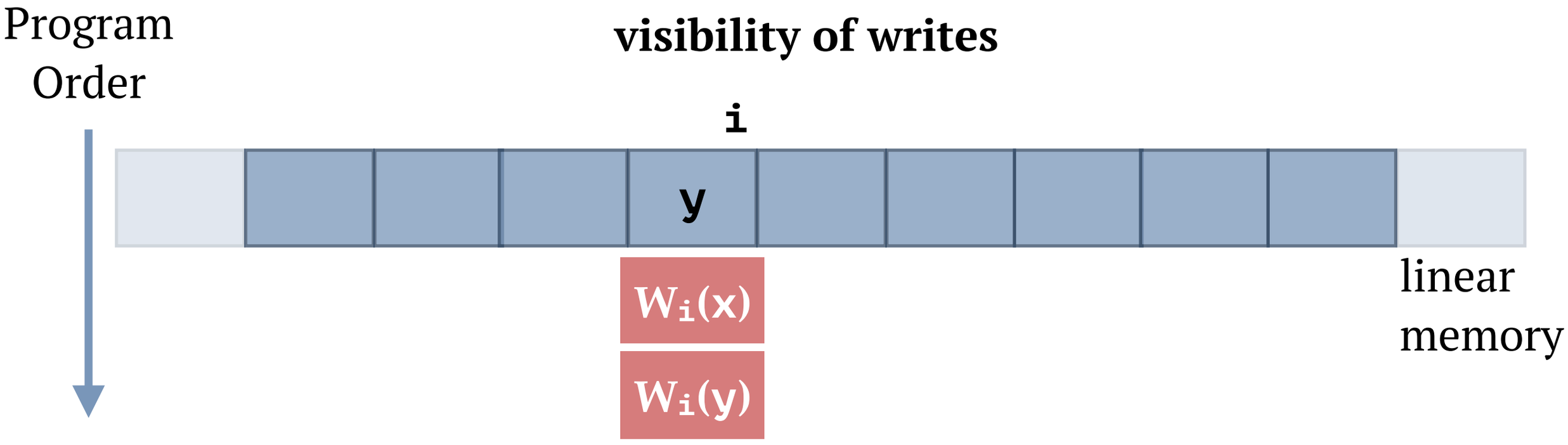
Linear Memory Write-over-Write Theorem: #2



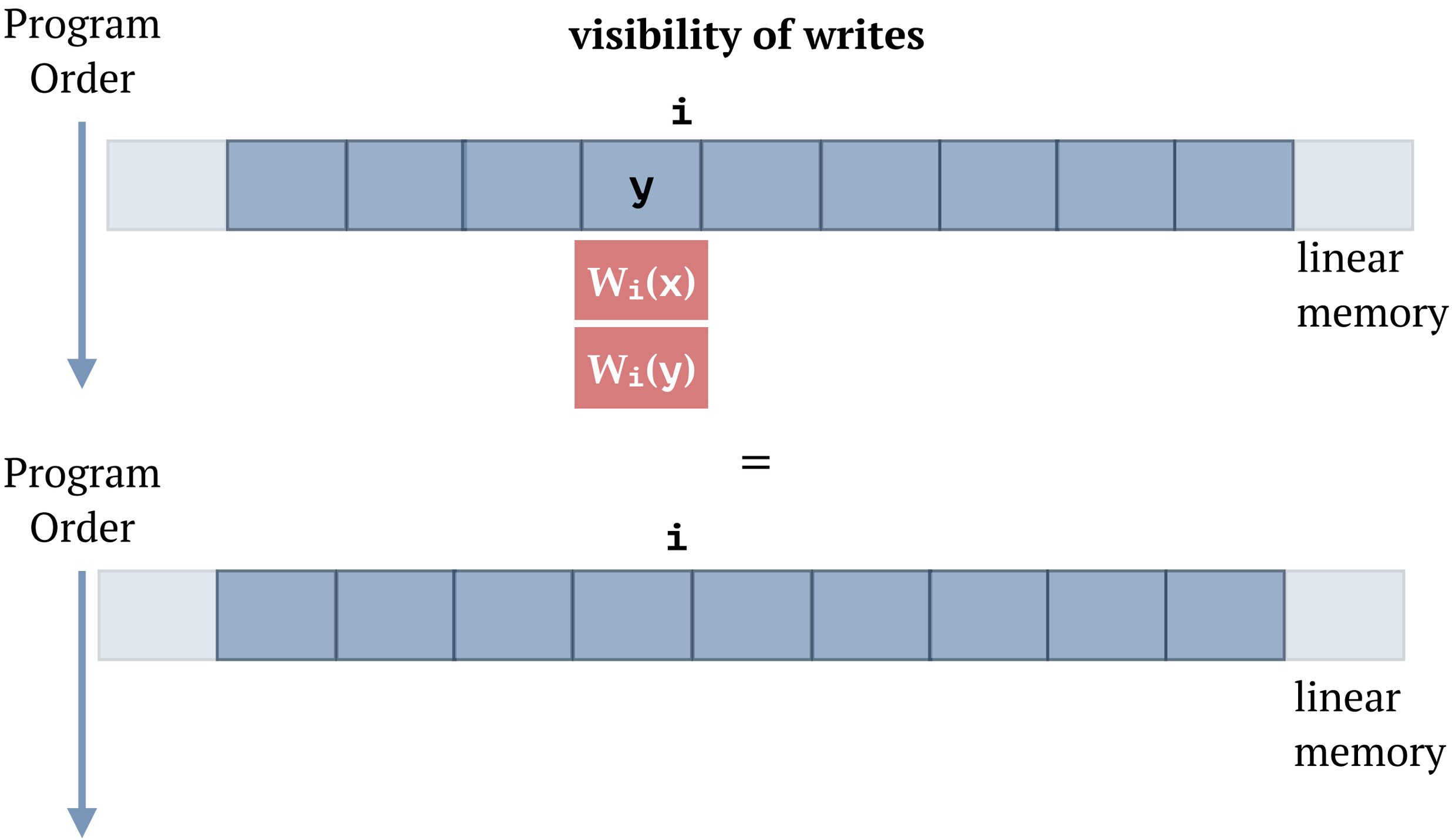
Linear Memory Write-over-Write Theorem: #2



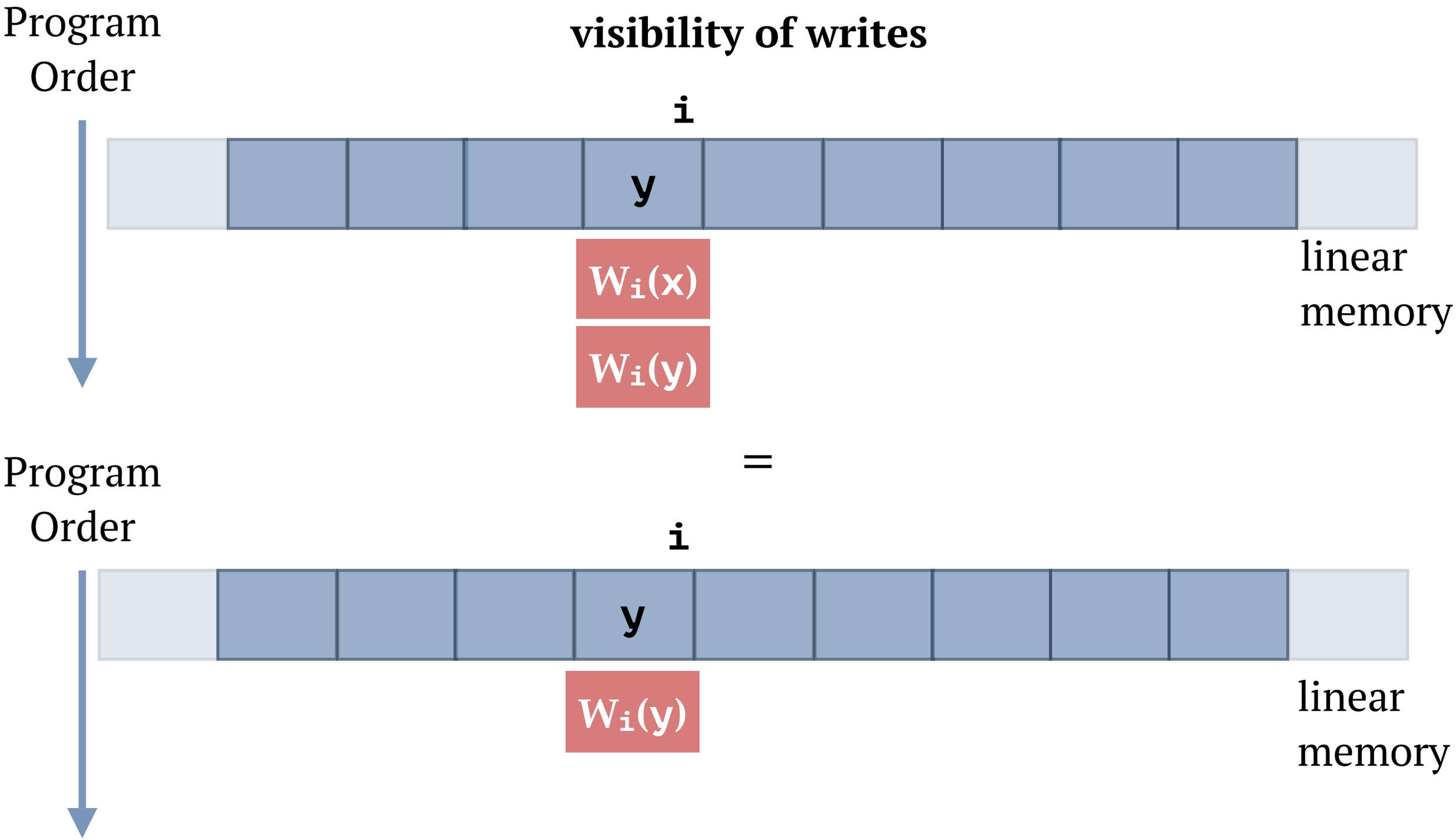
Linear Memory Write-over-Write Theorem: #2



Linear Memory Write-over-Write Theorem: #2



Linear Memory Write-over-Write Theorem: #2



Properties of Paging Data Structures and Entries

We have proved ~400 general theorems about paging data structures and their entries.

Two main lessons:

1. Separate on-the-fly updates from traversals
2. Find patterns and stick to them — helps with automation!

Future Work

Short-term Goals:

- Formulate and prove other critical **properties of paging structures**
- **Verify system programs** that access and modify paging structures
  - E.g., optimized data-copy program

Future Work

Short-term Goals:

- Formulate and prove other critical **properties of paging structures**
- **Verify system programs** that access and modify paging structures
  - E.g., optimized data-copy program

Long-term Goals:

- **Simulate a mainstream system**, i.e., FreeBSD, on our x86 ISA model
  - Support I/O devices
- **Verify OS routines**
  - Functional behavior
  - Security
  - Resource Usage

Conclusion

Verification of programs should take low-level “details” into account.

Unvalidated abstractions == dangerously **inaccurate assumptions**

A Formal Specification of x86 Memory Management

Shilpi Goel
shigoel@cs.utexas.edu

Warren A. Hunt, Jr.
hunt@cs.utexas.edu

The University of Texas at Austin

Thanks!
Questions/Comments?

Extra Slides

Our Approach

Machine-code verification for x86 platforms

Our Approach

Machine-code verification for x86 platforms

Why not high-level code verification?

- x High-level verification frameworks do not address compiler bugs
  - ✓ Verified/verifying compilers can help
- x Need to build verification frameworks for many high-level languages
- x Sometimes, high-level code is unavailable

Our Approach

Machine-code verification for x86 platforms

Why not high-level code verification?

- x High-level verification frameworks do not address compiler bugs
  - ✓ Verified/verifying compilers can help
- x Need to build verification frameworks for many high-level languages
- x Sometimes, high-level code is unavailable

Why x86?

- ✓ x86 is in widespread use — our approach will have immediate practical application

Model Development

Under active development: an x86 ISA model in ACL2

- ***x86 State:*** specifies the components of the ISA (registers, flags, memory)

- ***Instruction Semantic Functions:*** specify the effect of each instruction

- ***Step Function:*** fetches, decodes, and executes one instruction

Model Development

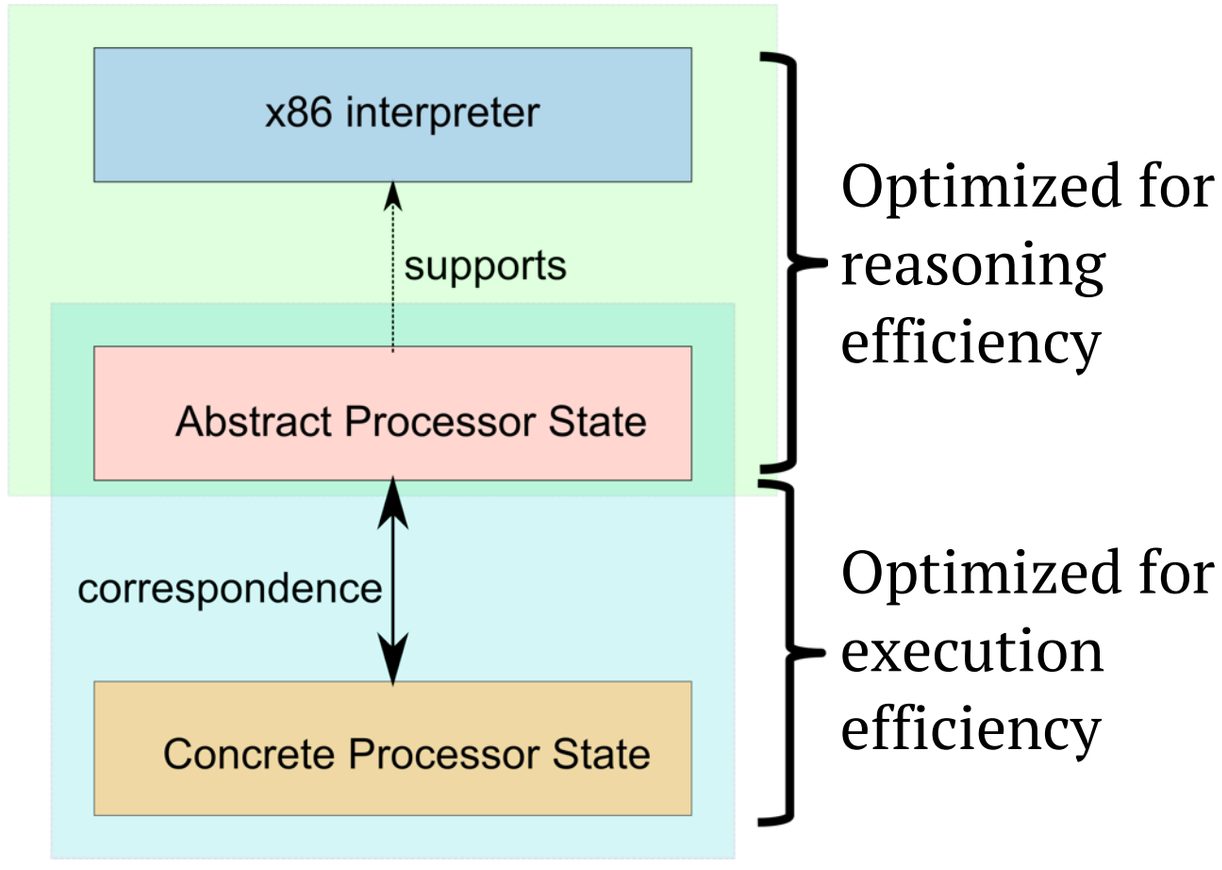
Under active development: an x86 ISA model in ACL2

- ***x86 State:*** specifies the components of the ISA (registers, flags, memory)

- ***Instruction Semantic Functions:*** specify the effect of each instruction

- ***Step Function:*** fetches, decodes, and executes one instruction

Layered modeling approach mitigates the trade-off between reasoning and execution efficiency [ACL2'13]



Verification Effort vs. Verification Utility

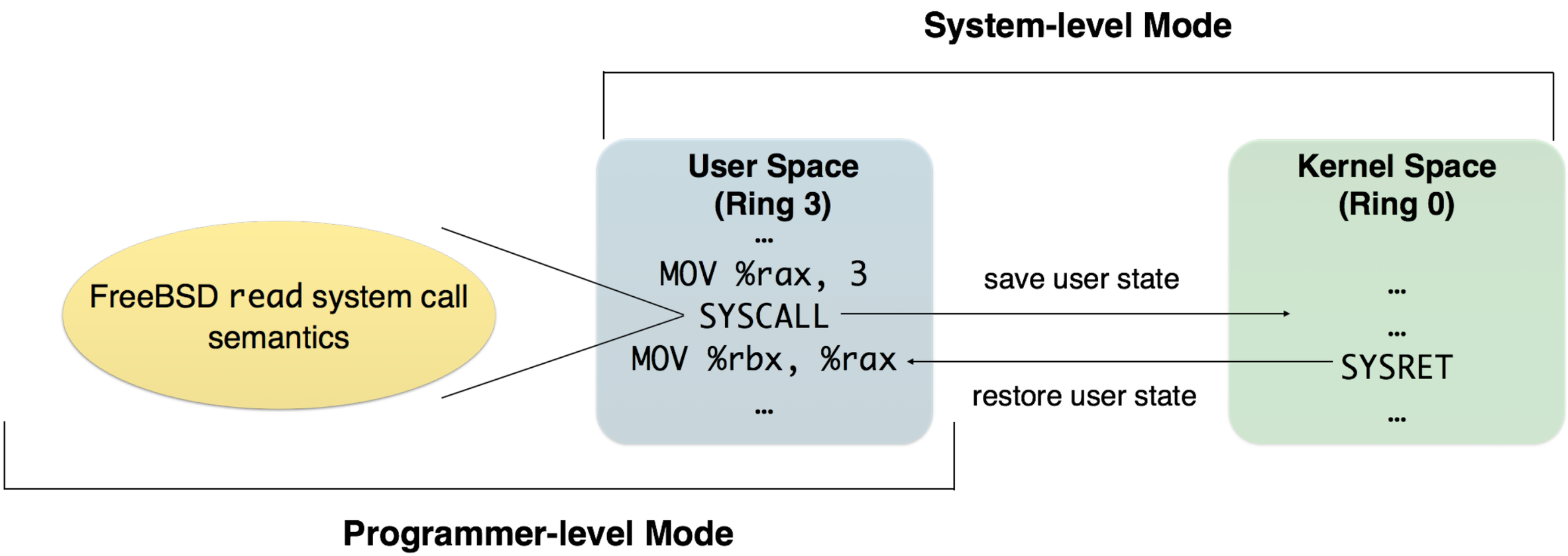
Programmer-level Mode

- Verification of *application* programs
- *Linear* memory address space (2^{64} bytes)
- *Assumptions* about correctness of OS operations

System-level Mode

- Verification of *system* programs
- *Physical* memory address space (2^{52} bytes)
- *No assumptions* about OS operations

Verification Effort vs. Verification Utility



Lemma Database

- Semantics of the program is given by the effect it has on the machine state.

Lemma Database

- Semantics of the program is given by the effect it has on the machine state.

```
add %edi, %eax
je  0x400304
```

Lemma Database

- Semantics of the program is given by the effect it has on the machine state.

```
add %edi, %eax
je  0x400304
```

1. read instruction from `mem`
2. read operands
3. write sum to `eax`
4. write new value to `flags`
5. write new value to `pc`

Lemma Database

- Semantics of the program is given by the effect it has on the machine state.

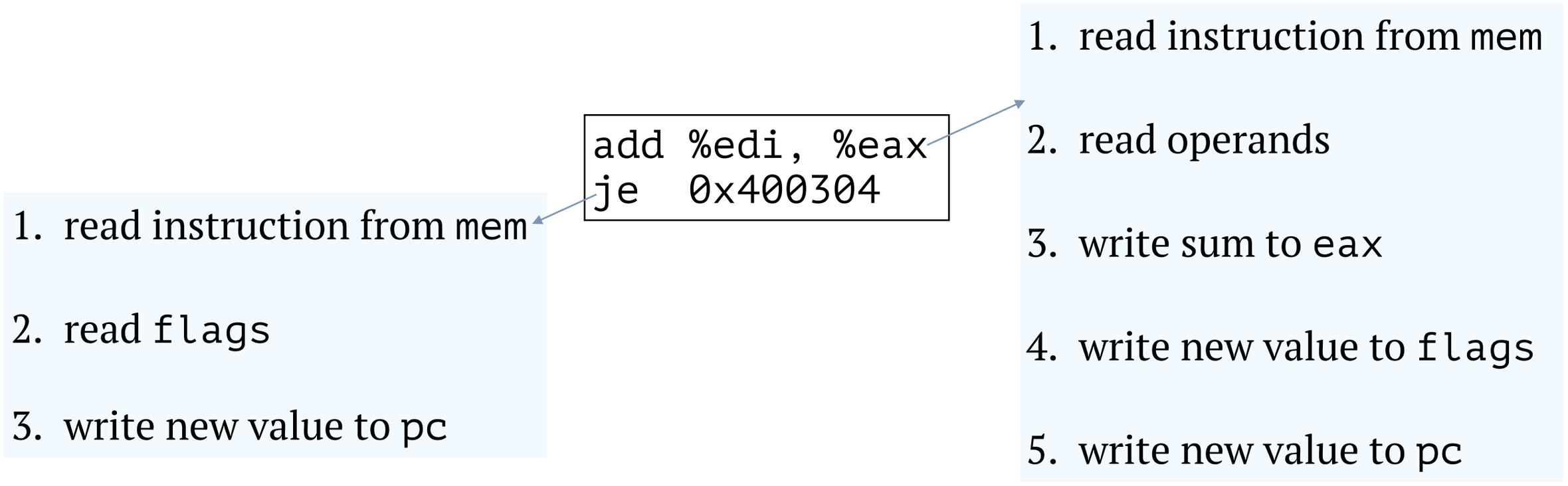
1. read instruction from `mem`
2. read `flags`
3. write new value to `pc`

```
add %edi, %eax
je  0x400304
```

1. read instruction from `mem`
2. read operands
3. write sum to `eax`
4. write new value to `flags`
5. write new value to `pc`

Lemma Database

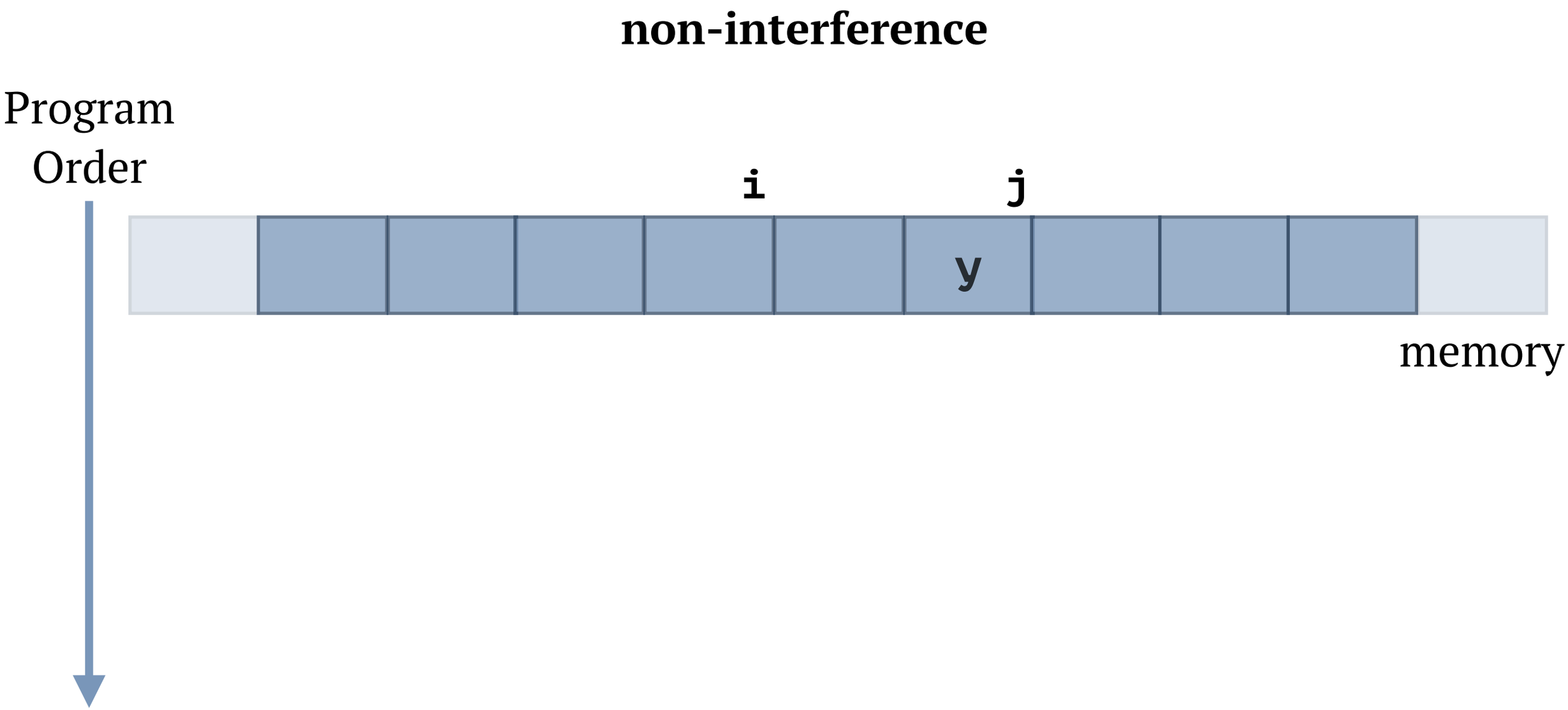
- Semantics of the program is given by the effect it has on the machine state.



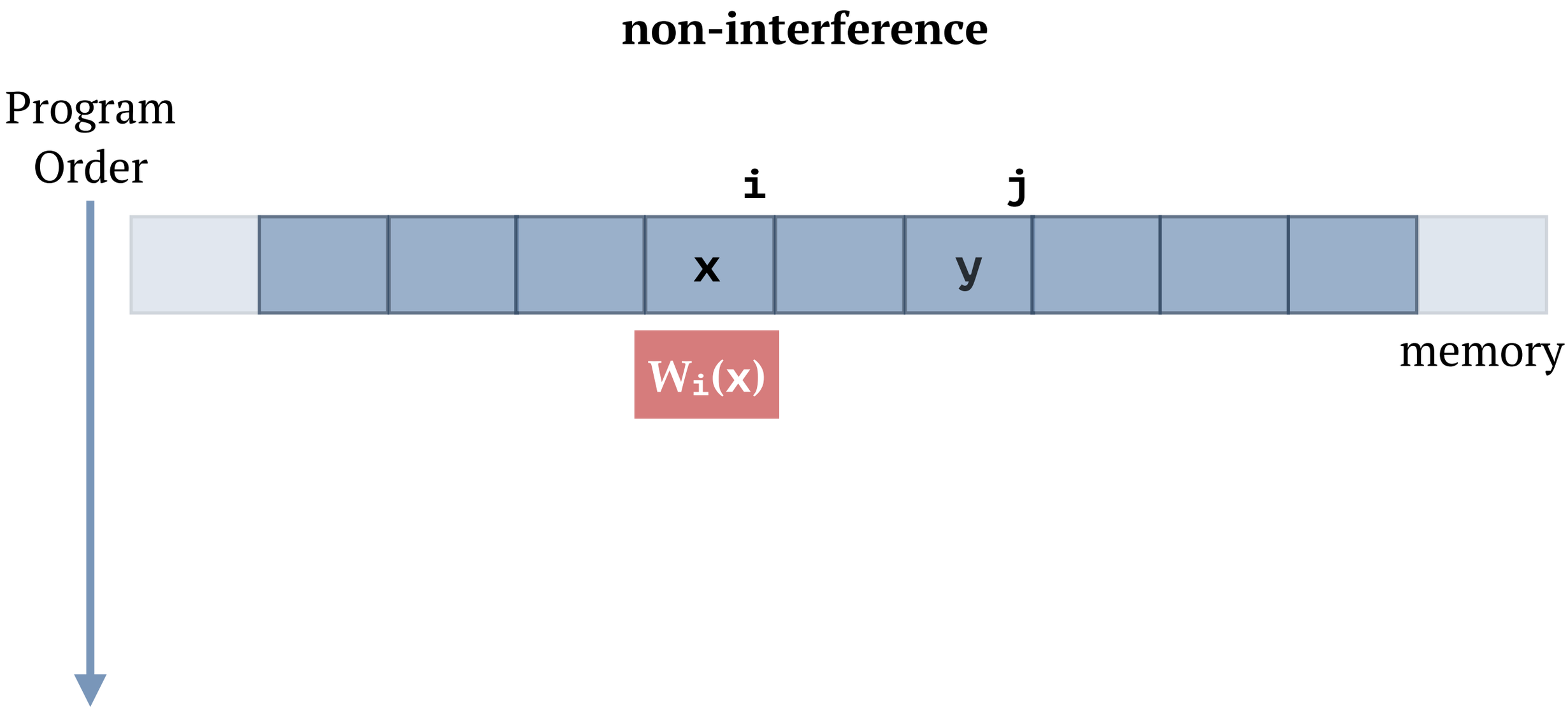
- Need to reason about:
  - Reads from machine state
  - Writes to machine state

- Three kinds of theorems:
  - Read-over-Write Theorems
  - Write-over-Write Theorems
  - Preservation Theorems

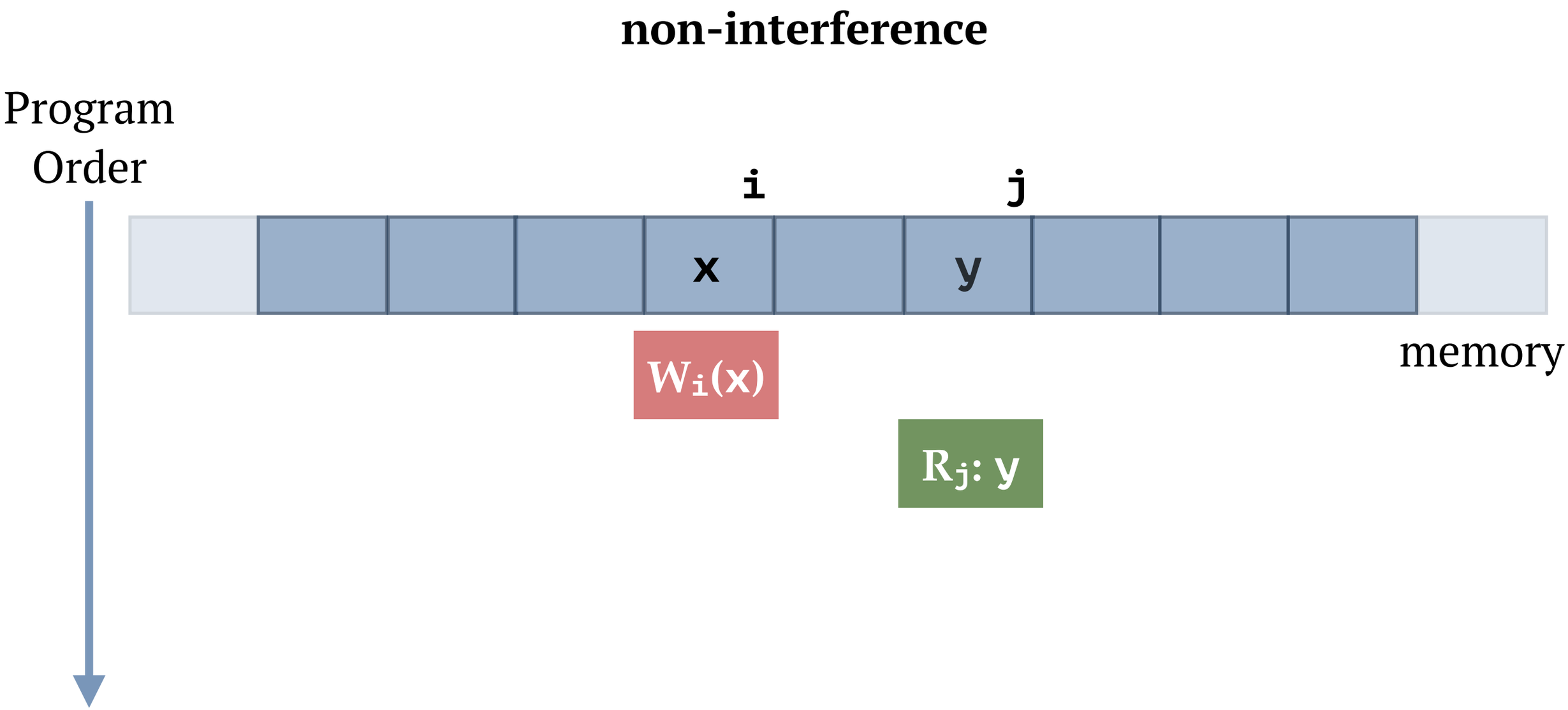
Read-over-Write Theorem: #1



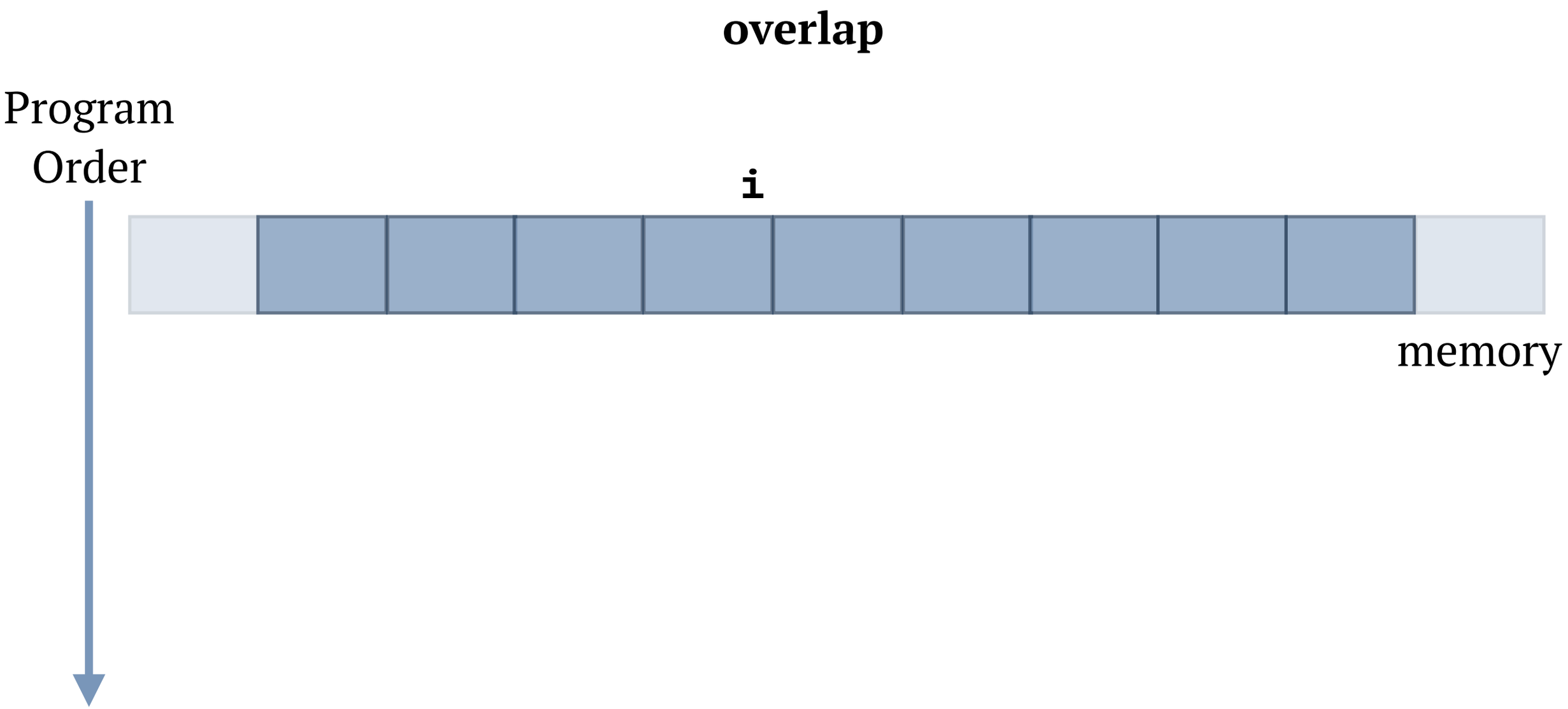
Read-over-Write Theorem: #1



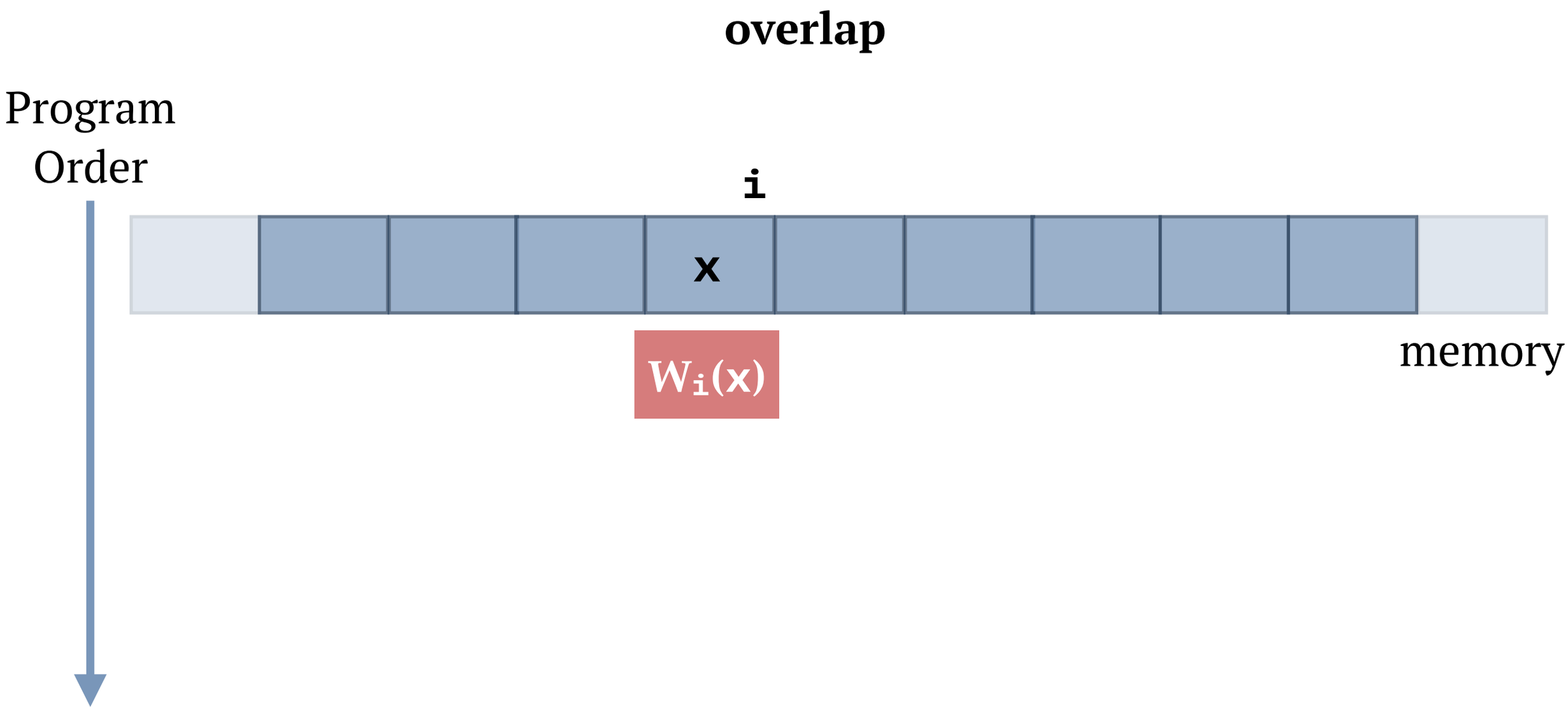
Read-over-Write Theorem: #1



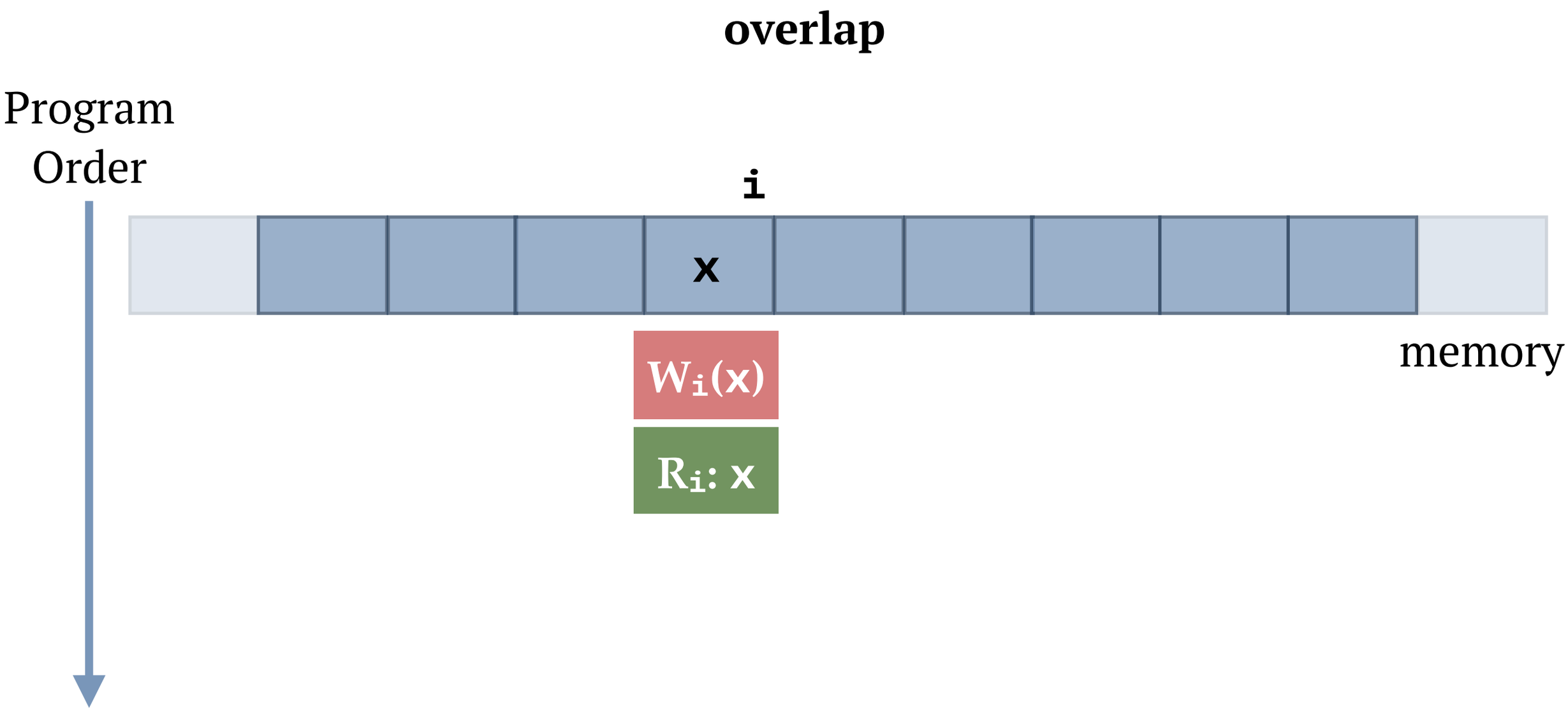
Read-over-Write Theorem: #2



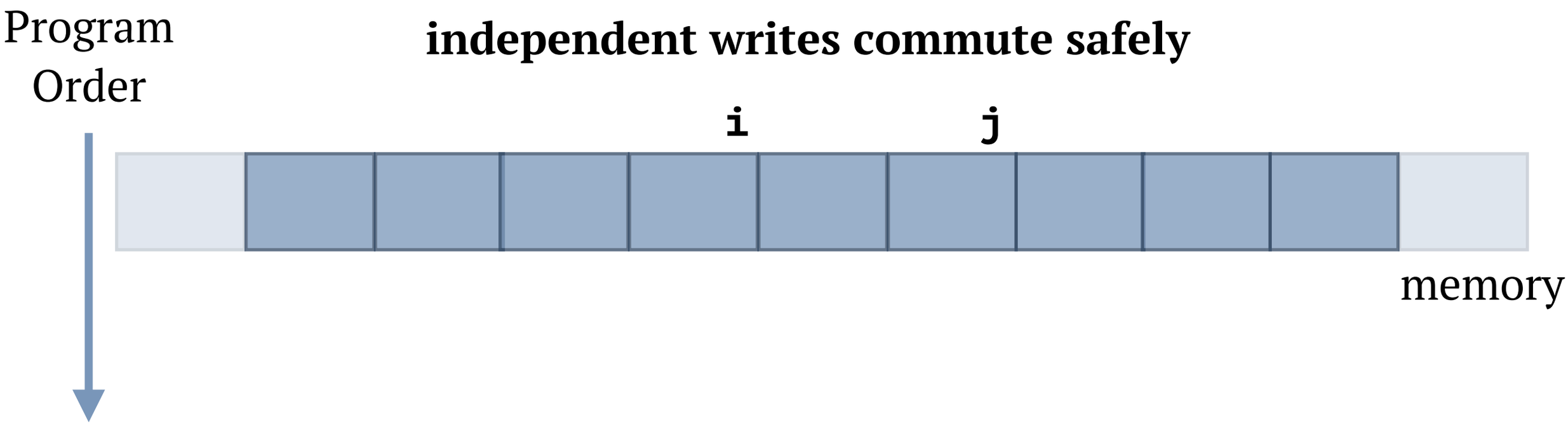
Read-over-Write Theorem: #2



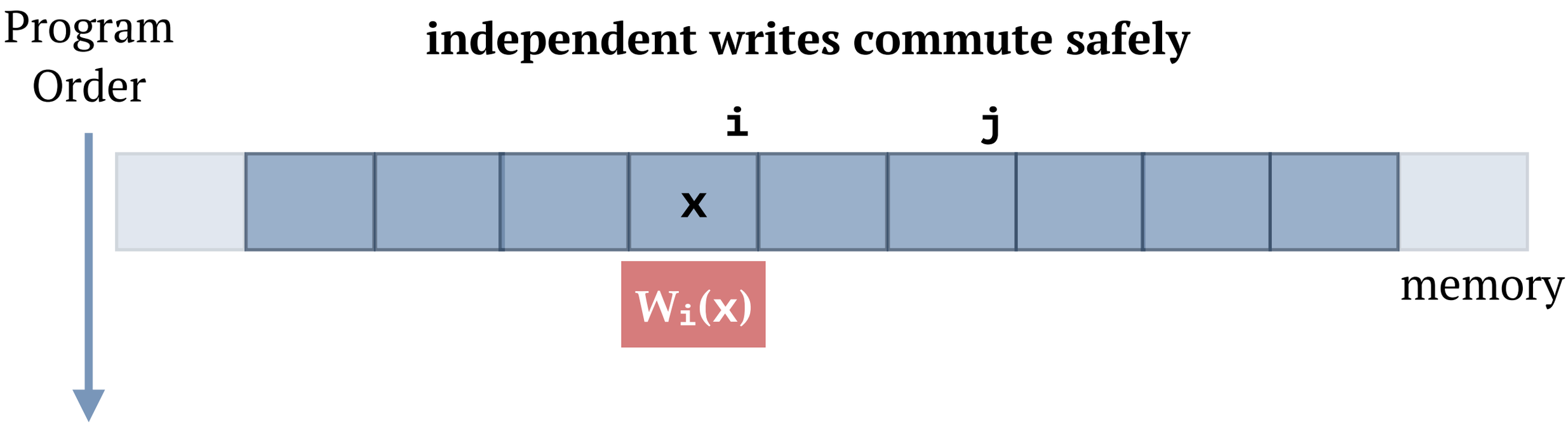
Read-over-Write Theorem: #2



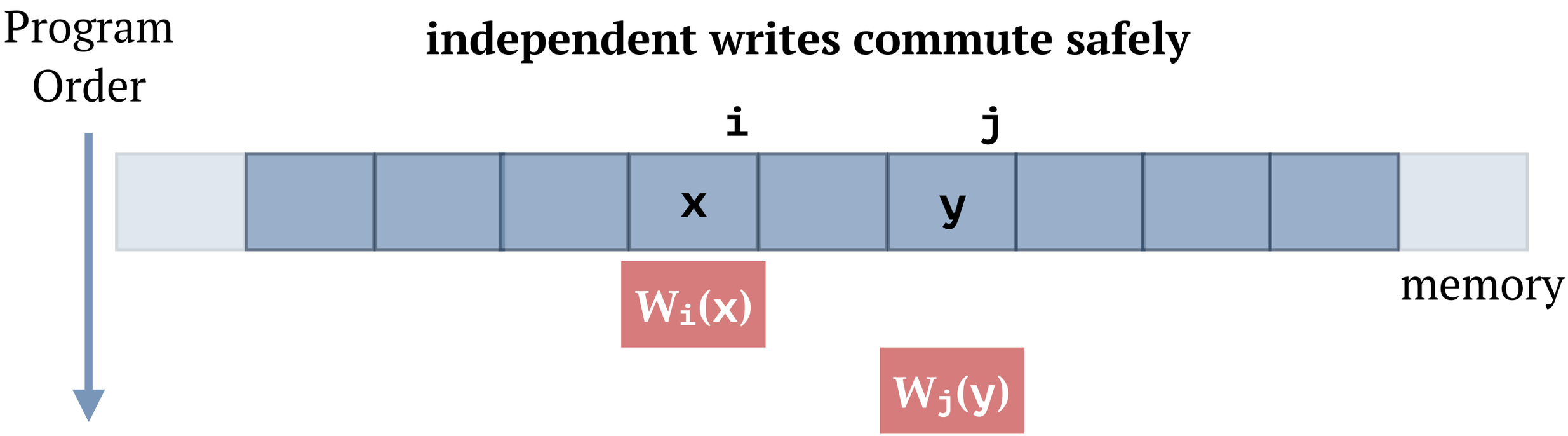
Write-over-Write Theorem: #1



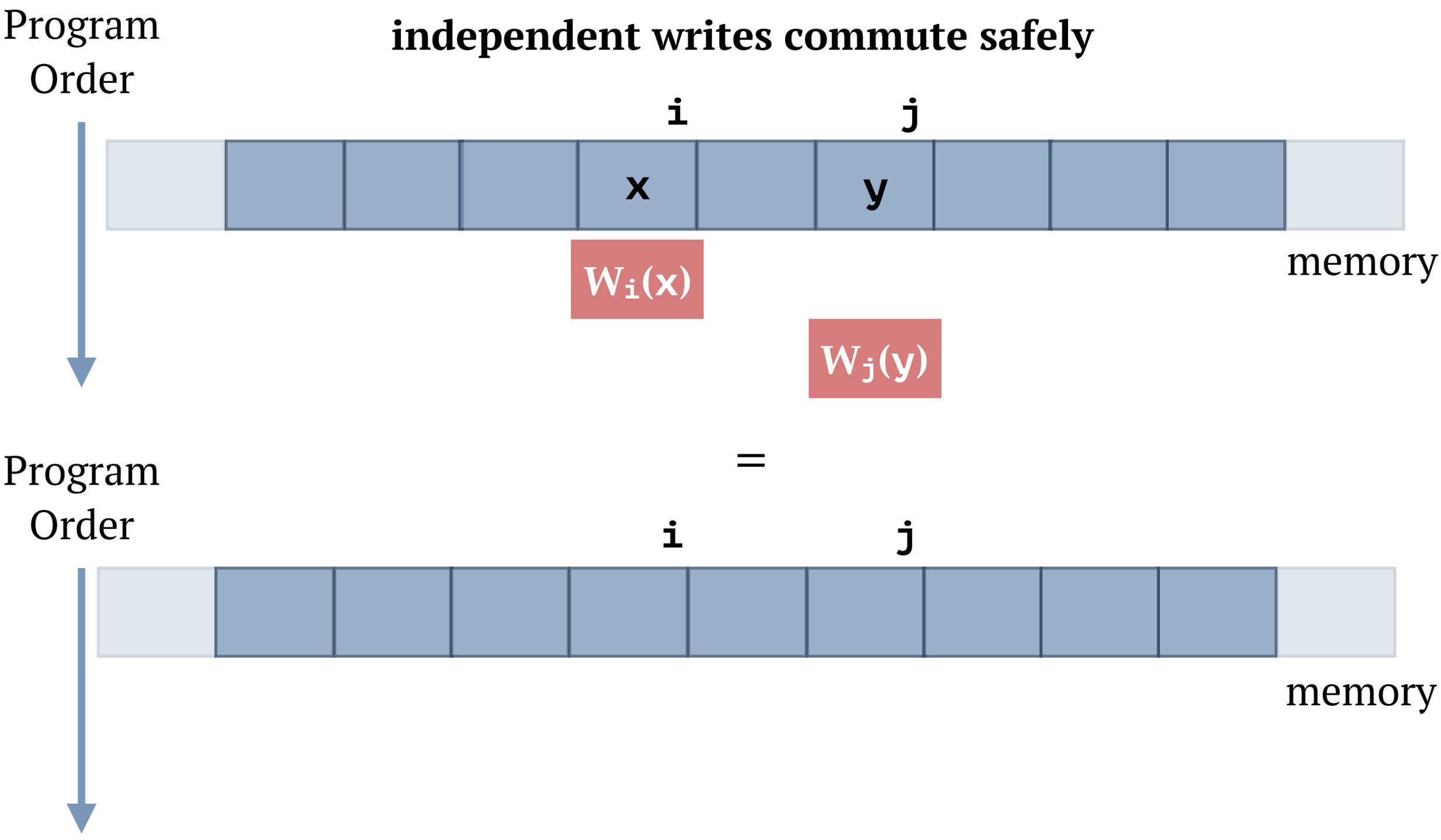
Write-over-Write Theorem: #1



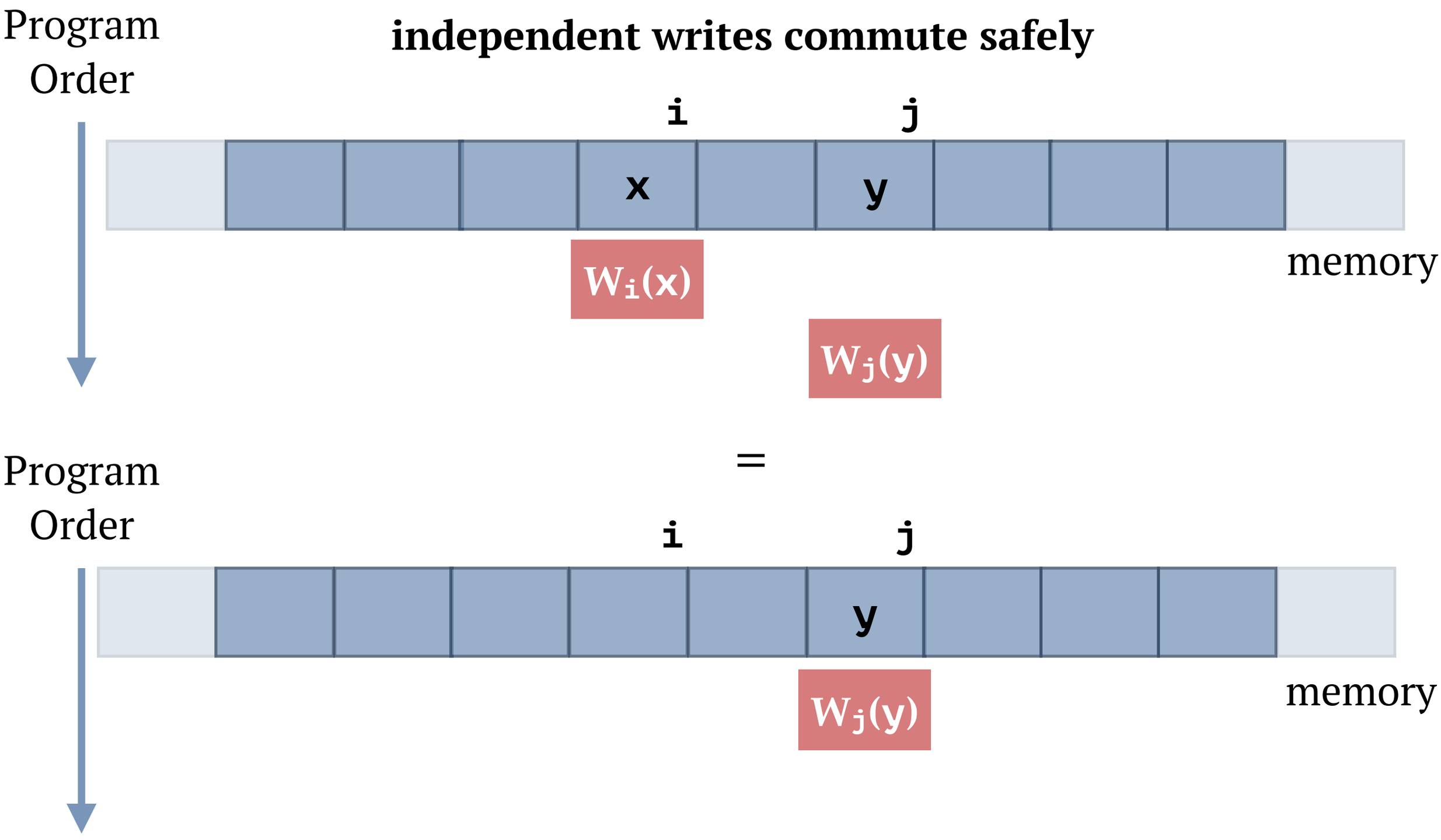
Write-over-Write Theorem: #1



Write-over-Write Theorem: #1



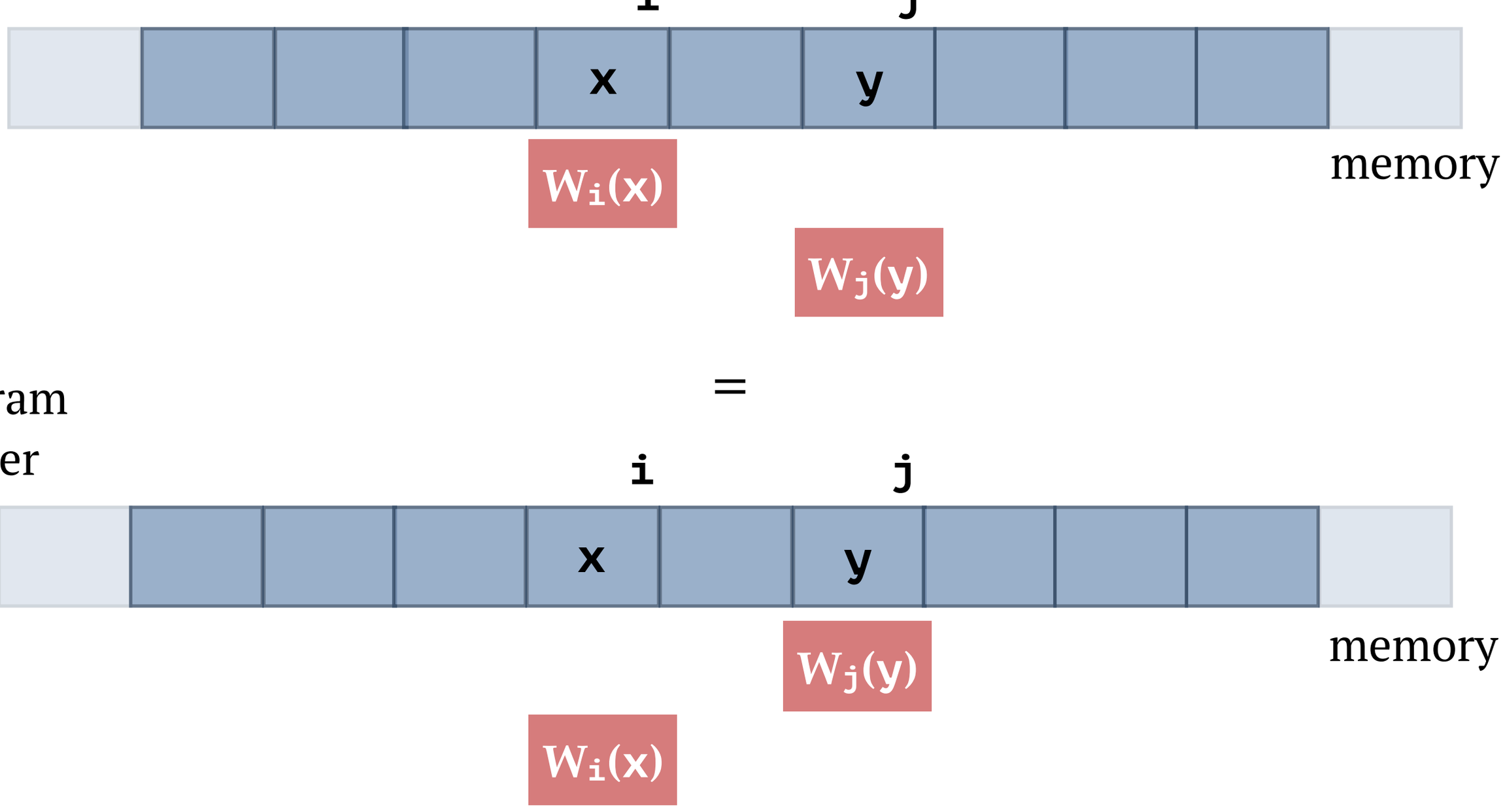
Write-over-Write Theorem: #1



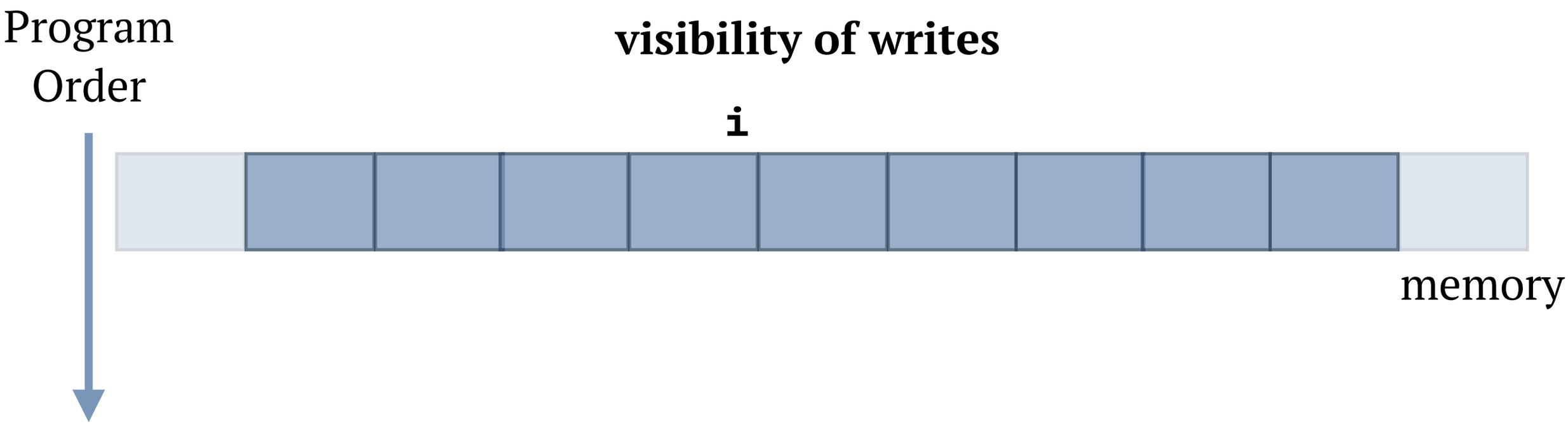
Write-over-Write Theorem: #1

Program Order

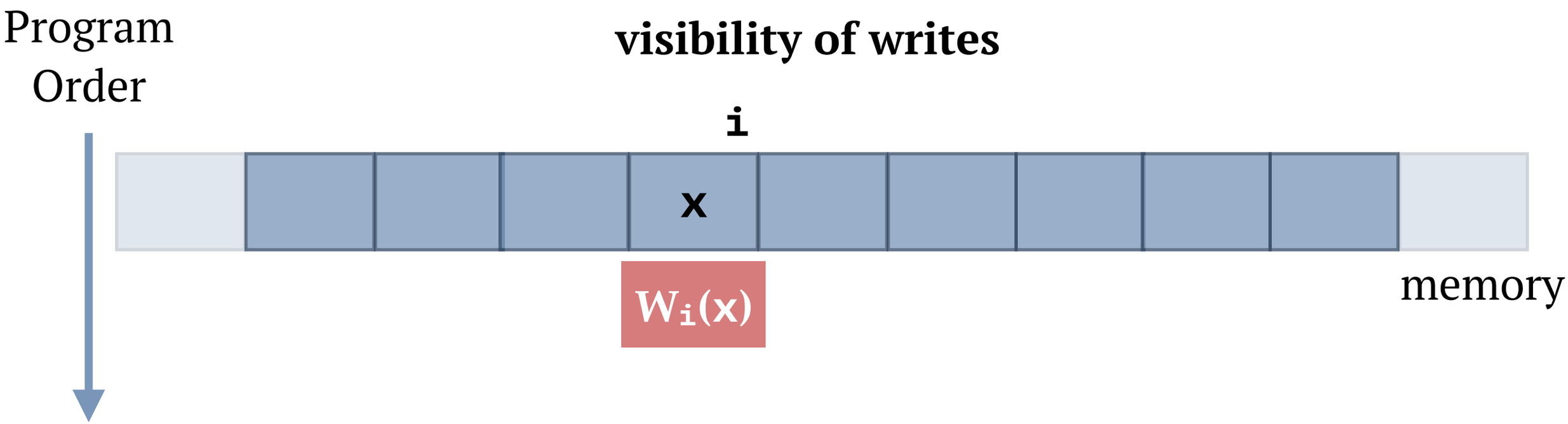
independent writes commute safely



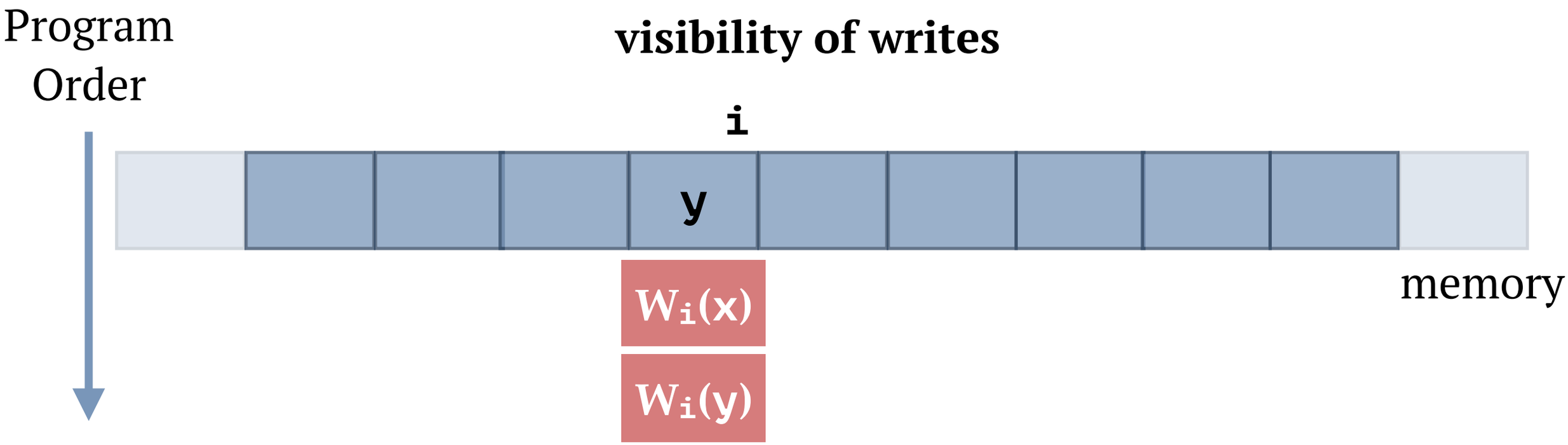
Write-over-Write Theorem: #2



Write-over-Write Theorem: #2



Write-over-Write Theorem: #2



Write-over-Write Theorem: #2

Program Order

visibility of writes

i

y

$W_i(x)$

$W_i(y)$

memory

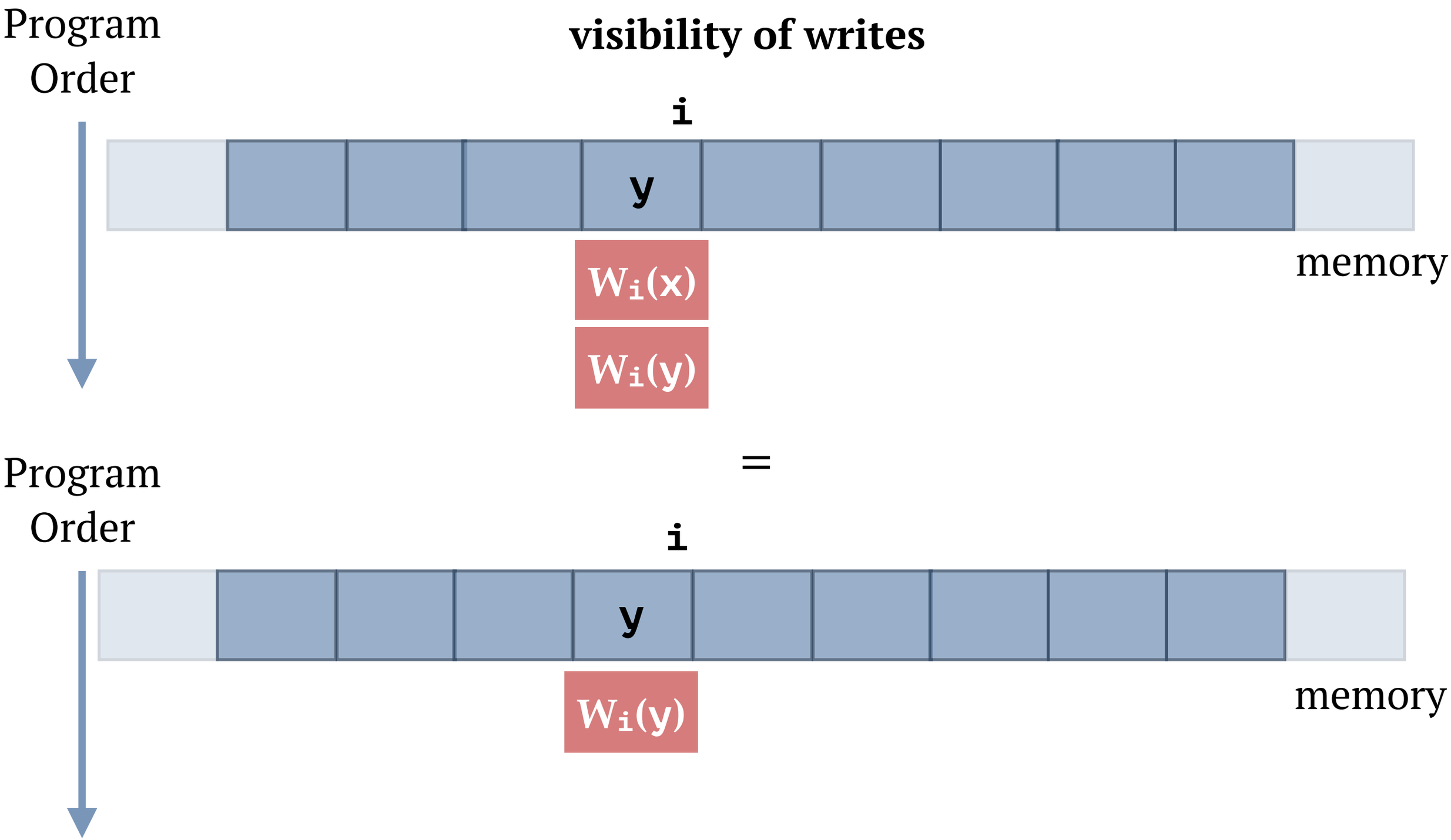
=

Program Order

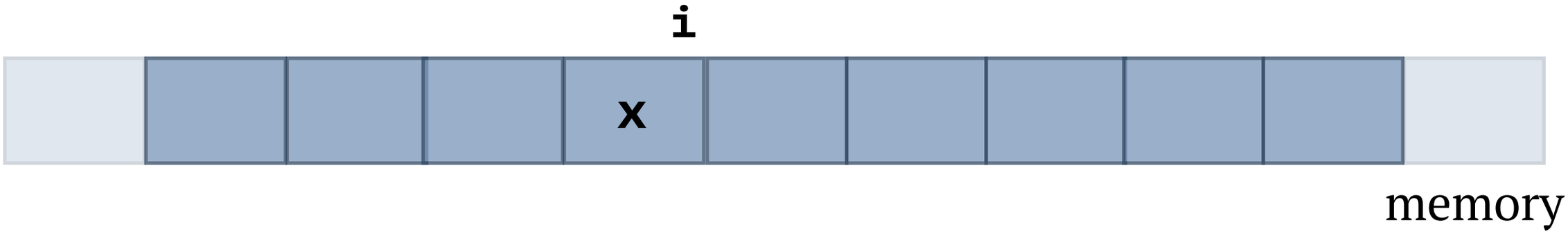
i

memory

Write-over-Write Theorem: #2



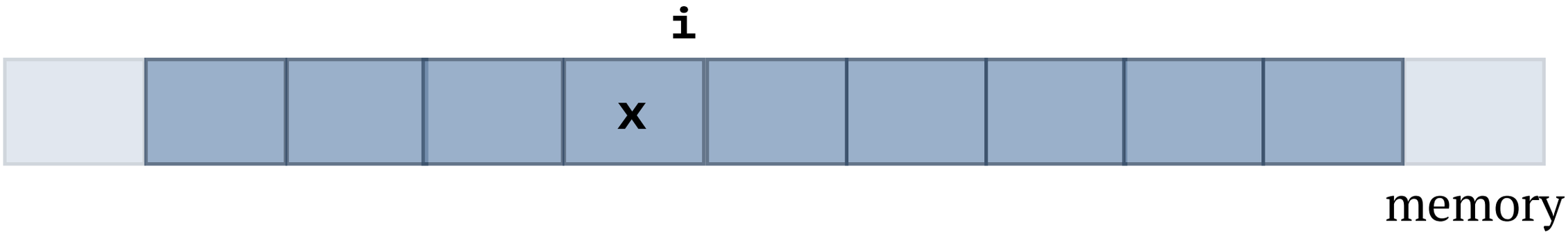
Preservation Theorems



reading from a valid x86 state

```
valid-address-p(i) ∧
valid-x86-p(x86)
⇒
valid-value-p( Ri: x ) ∧
valid-x86-p(x86)
```

Preservation Theorems



reading from a valid x86 state

```
valid-address-p(i) ∧
valid-x86-p(x86)
⇒
valid-value-p( Ri: x ) ∧
valid-x86-p(x86)
```

writing to a valid x86 state

```
valid-address-p(i) ∧
valid-value-p(x) ∧
valid-x86-p(x86)
⇒
valid-x86-p( Wi(x) )
```