Analysis of x86 Application and System Programs via Machine-Code Verification

Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann

Department of Computer Science
The University of Texas at Austin
1 University Way, M/S C0500
Austin, TX 78712-0233

`hunt@cs.utexas.edu`
TEL: +1 512 471 9748
FAX: +1 512 471 8885

April, 2016

Introduction

Motivation: Increase the **reliability** of programs used in the industry

Approach: **Machine-code verification for x86 platforms**

- We are developing a **formal x86 model** in **ACL2** for code analysis.
- We are vetting our tools on **commercial-sized problems**.

Today, we talk about our ongoing work in formal verification of software, and present our plans to verify supervisor-level code in the immediate future.

Objective: Emulate an operating system, like **FreeBSD**, along with the programs running on it, and **prove properties about kernel code**

Ecosystem

Our group has significant collaboration with the government and industry.

Our own research includes:

- Development of **core technologies**
- **Application** of these technologies in different domains
- Validation of **commercial** processor designs at Centaur and Oracle (10+ developers, 30+ users)

Project Overview

Goal: Build robust tools to increase software reliability

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

Plan of Action:

1. Build a **formal, executable x86 ISA model** using ACL2
2. Develop a **machine-code analysis framework** based on this model
3. Employ this framework to **verify application and system programs**

Contributions

A new tool: General-purpose analysis framework for x86 machine-code

- Accurate x86 ISA reference

Program verification taking memory management into account:

- Properties of x86 memory-management data structures
- Analysis of programs, including low-level system & ISA features

Reasoning strategies: Insight into low-level code verification in general

- Build effective lemma libraries

Foundation for future research:

- Target for verified/verifying compilers
- Resource usage guarantees
- Information-flow analysis
- Ensuring process isolation

Outline

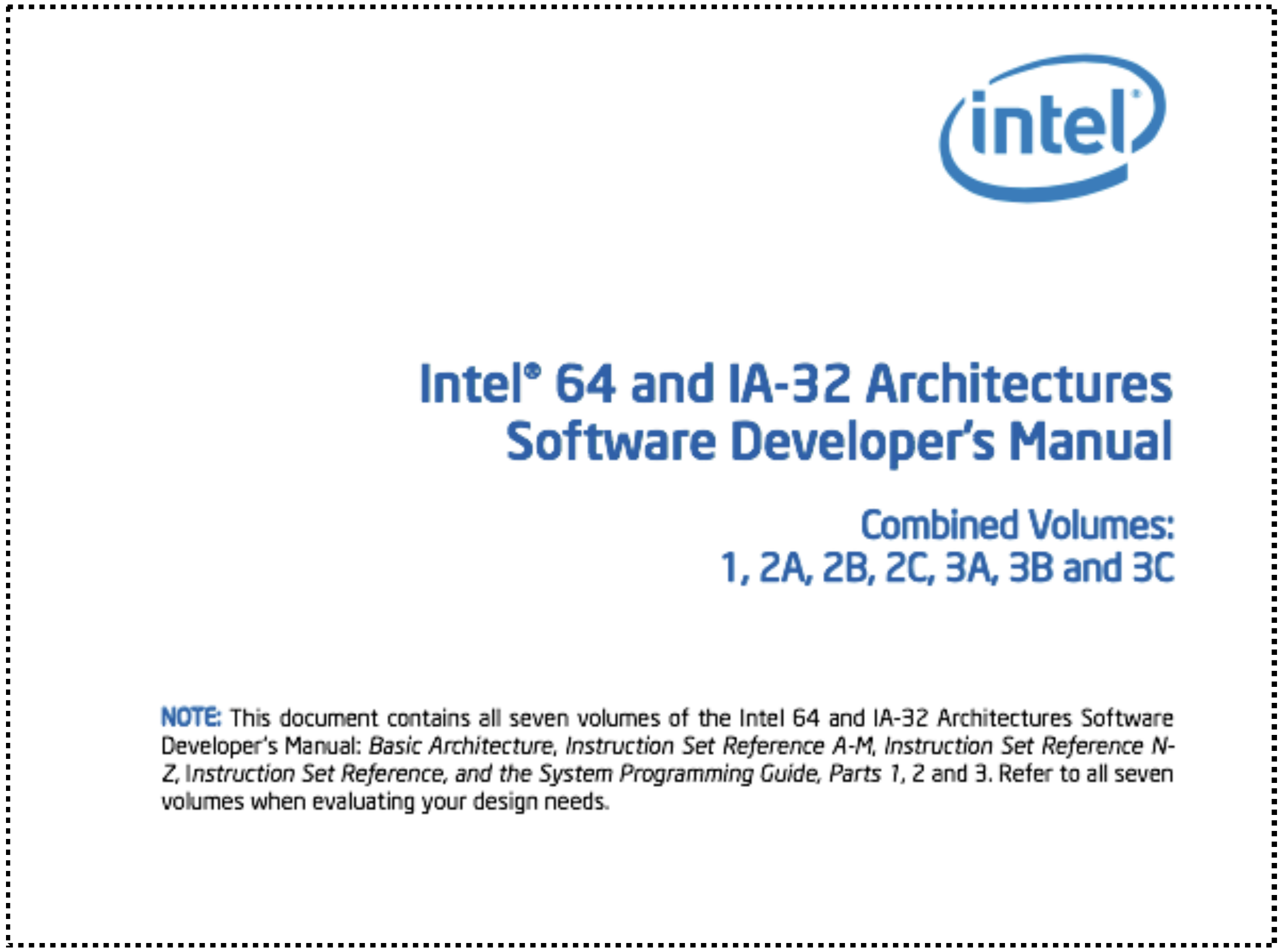
- Motivation
- Project Description
  - [1] **Developing an x86 ISA Model**
  - [2] Building a Machine-Code Analysis Framework
  - [3] Verifying Application and System Programs
- Future Work & Conclusion
- Accessing Source Code + Documentation

Model Development

Obtaining the x86 ISA Specification

Model Development

Obtaining the x86 ISA Specification



Intel® 64 and IA-32 Architectures Software Developer's Manual

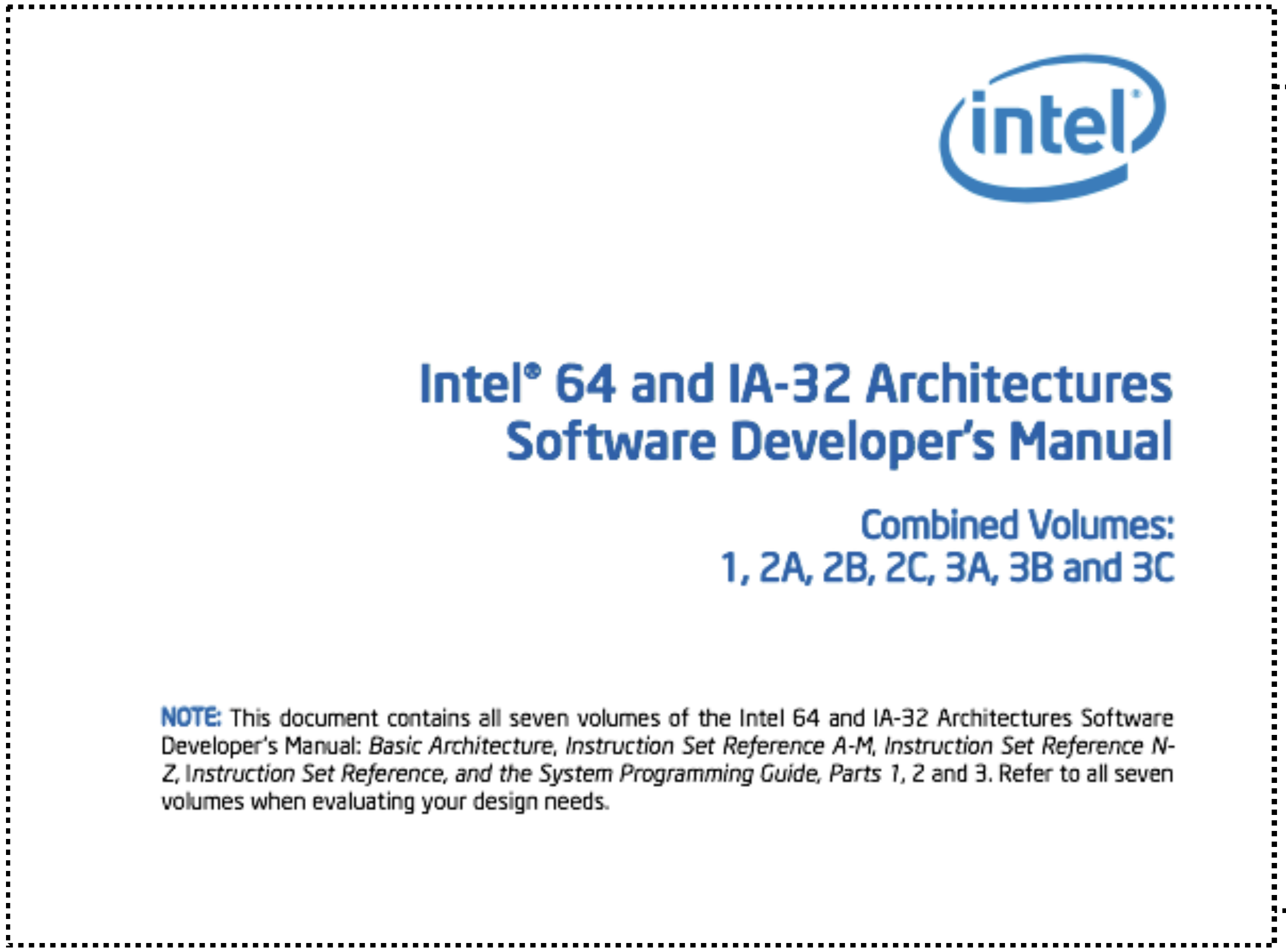
Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C

NOTE: This document contains all seven volumes of the Intel 64 and IA-32 Architectures Software Developer's Manual: *Basic Architecture, Instruction Set Reference A-M, Instruction Set Reference N-Z, Instruction Set Reference, and the System Programming Guide, Parts 1, 2 and 3*. Refer to all seven volumes when evaluating your design needs.

~3400 pages

Model Development

Obtaining the x86 ISA Specification



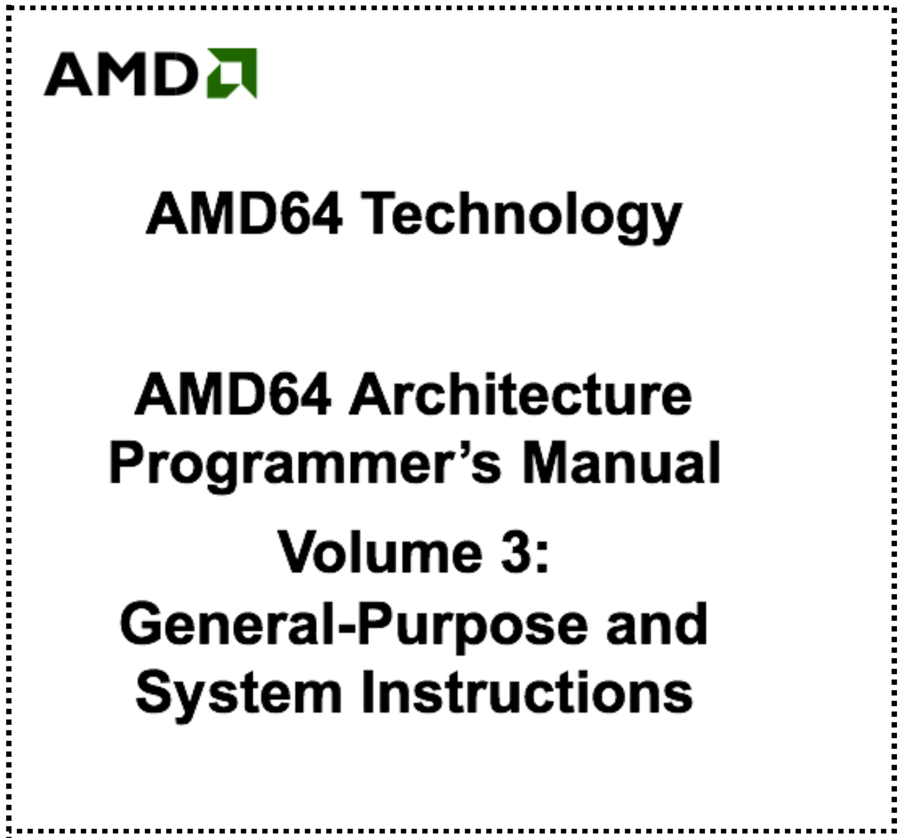
intel

Intel® 64 and IA-32 Architectures
Software Developer's Manual

Combined Volumes:
1, 2A, 2B, 2C, 3A, 3B and 3C

NOTE: This document contains all seven volumes of the Intel 64 and IA-32 Architectures Software Developer's Manual: *Basic Architecture, Instruction Set Reference A-M, Instruction Set Reference N-Z, Instruction Set Reference, and the System Programming Guide, Parts 1, 2 and 3*. Refer to all seven volumes when evaluating your design needs.

~3400 pages



AMD

AMD64 Technology

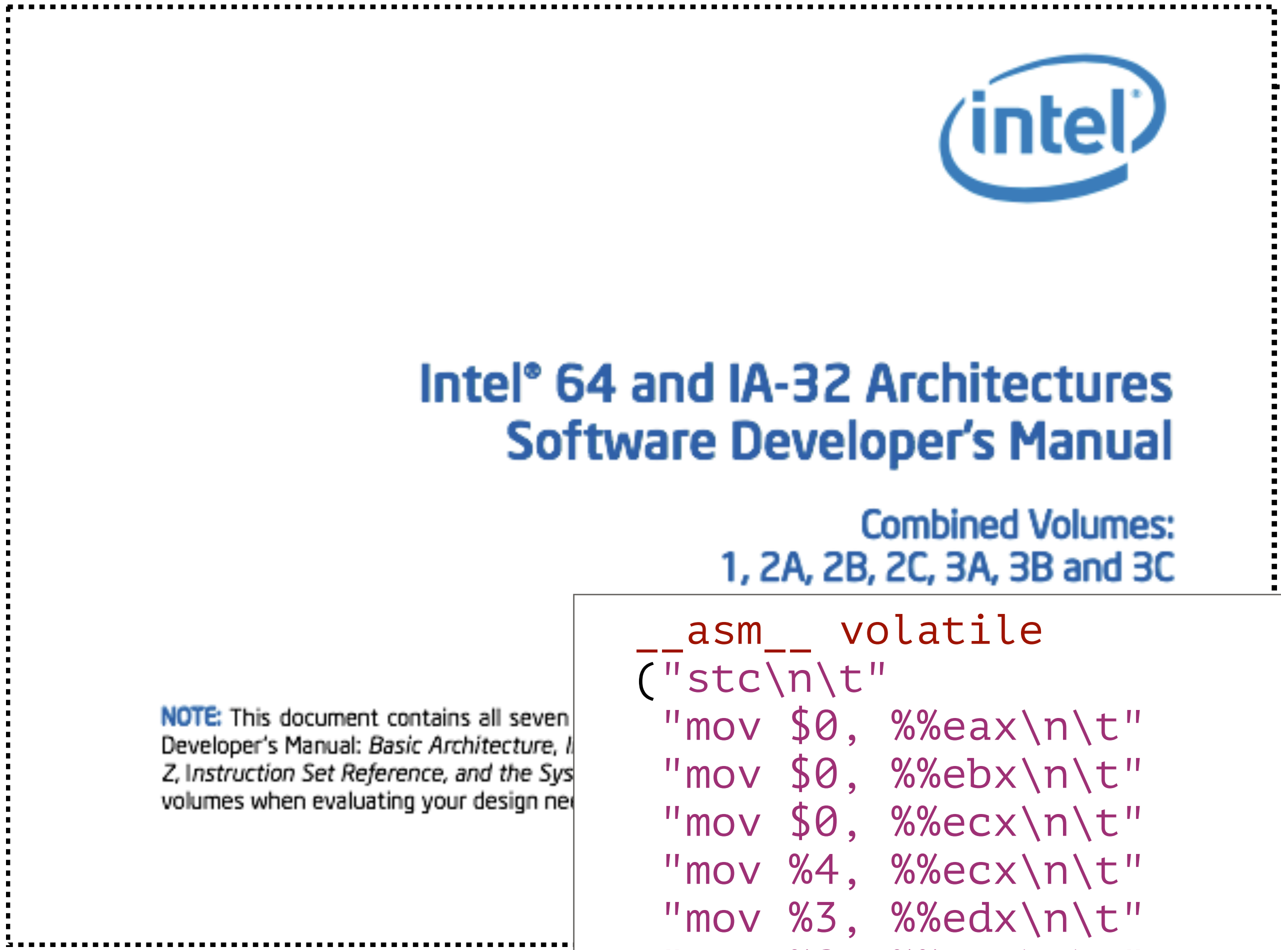
AMD64 Architecture
Programmer's Manual

Volume 3:
General-Purpose and
System Instructions

All AMD manuals: ~3000 pages

Model Development

Obtaining the x86 ISA Specification



Intel® 64 and IA-32 Architectures
Software Developer's Manual

Combined Volumes:
1, 2A, 2B, 2C, 3A, 3B and 3C

NOTE: This document contains all seven
Developer's Manual: *Basic Architecture,*
Z, *Instruction Set Reference, and the Sys*
volumes when evaluating your design ne

AMD64 Technology

AMD64 Architecture
Programmer's Manual

```
__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

Model Development

Focus: 64-bit sub-mode of Intel’s IA-32e mode

Model Development

Focus: 64-bit sub-mode of Intel’s IA-32e mode

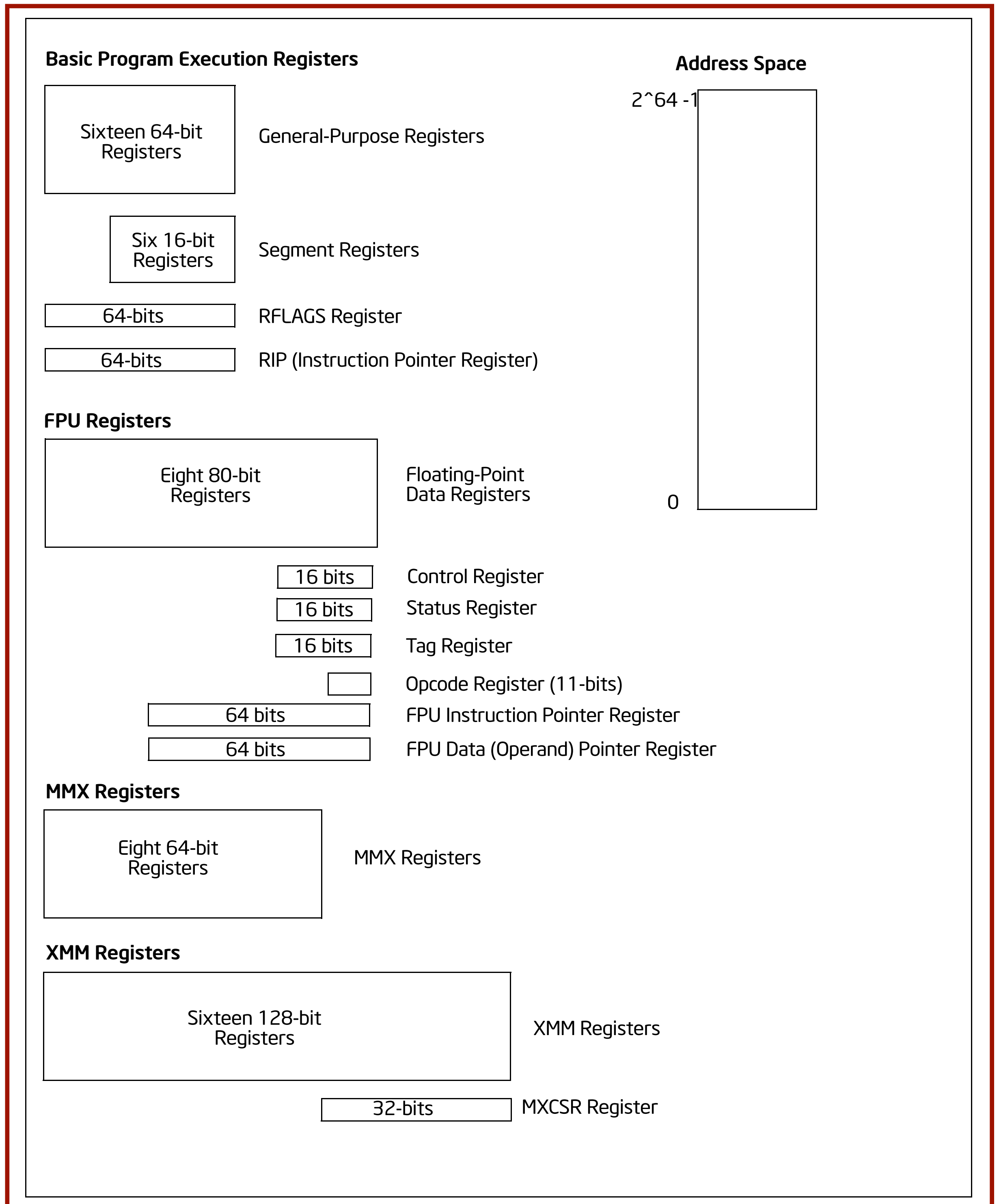


Figure 3-2. 64-Bit Mode Execution Environment

Source: Intel Manuals

Model Development

Focus: 64-bit sub-mode

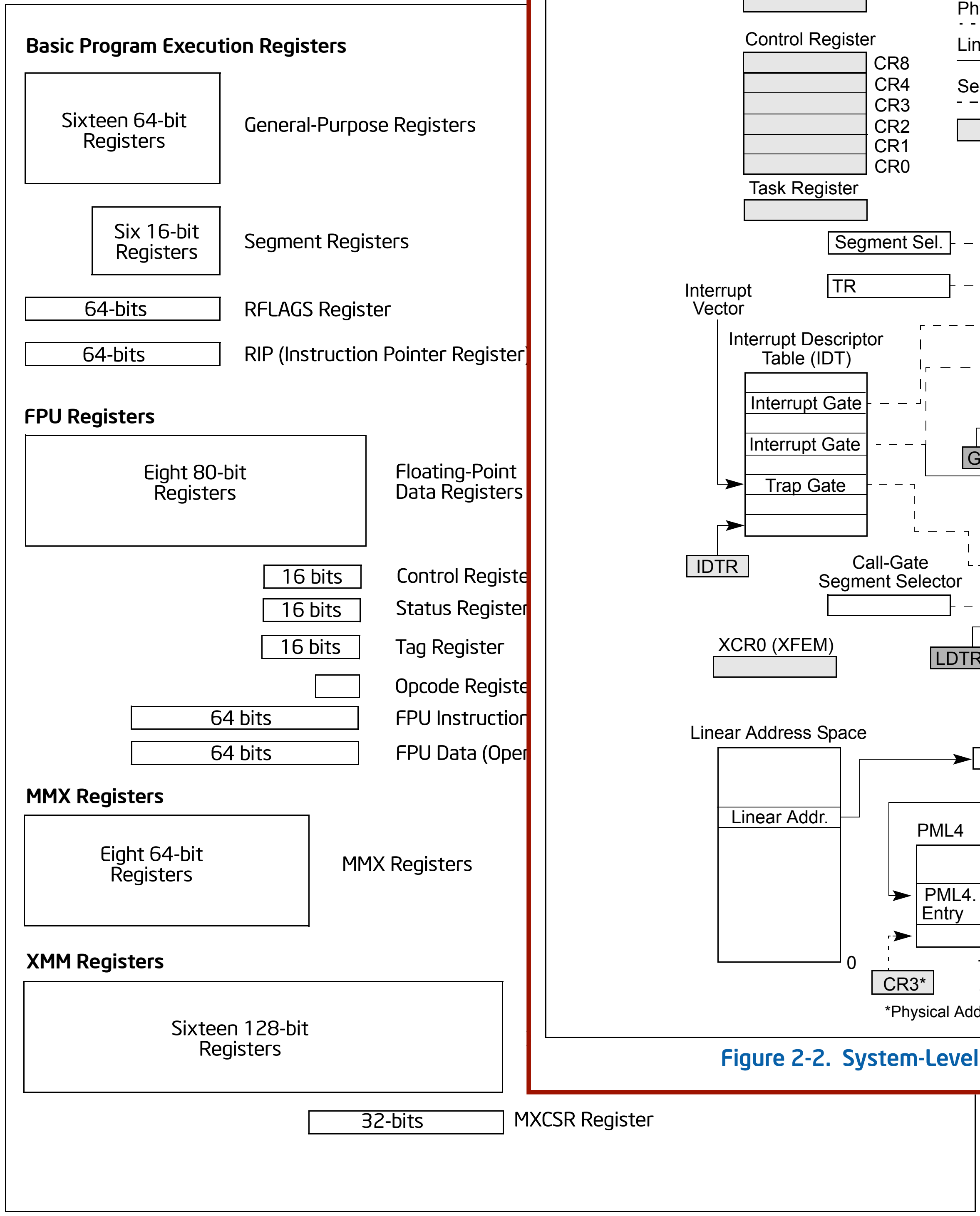


Figure 3-2. 64-Bit Mode Execution Environment

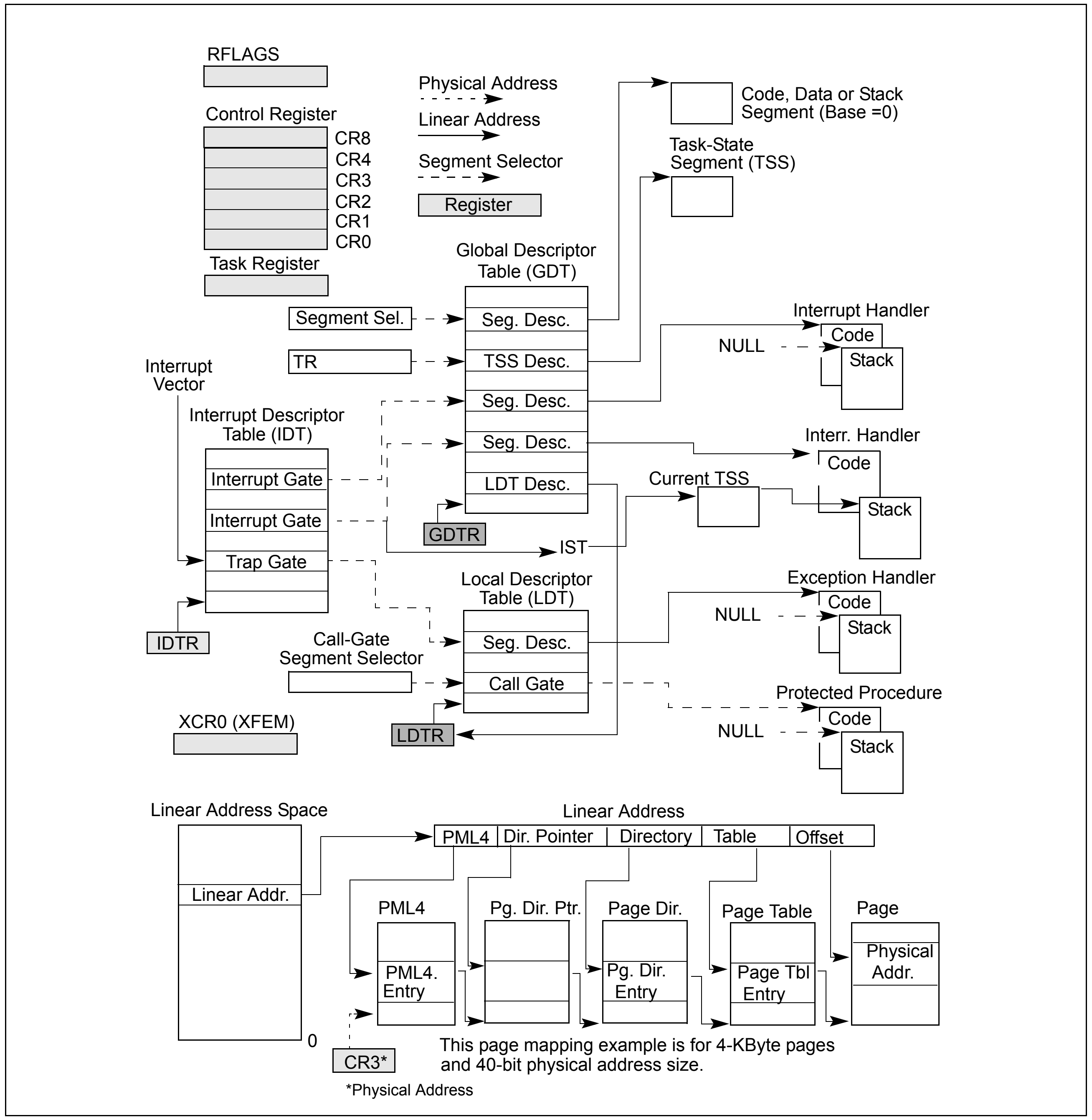


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

Source: Intel Manuals

Model Development

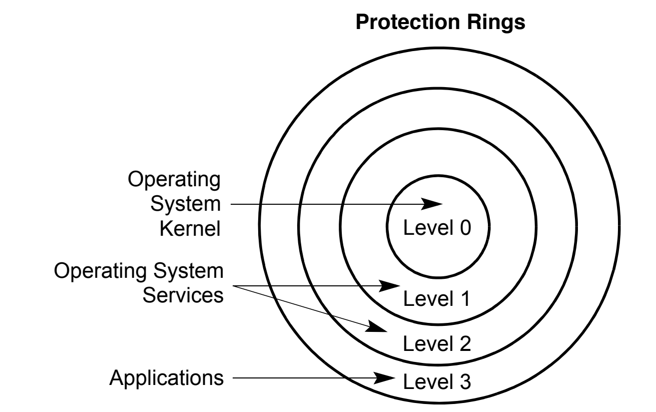


Figure 5-3. Protection Rings

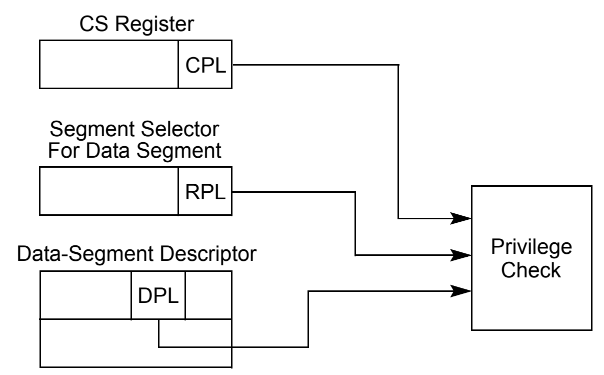


Figure 5-4. Privilege Check for Data Access

64-bit sub-mode

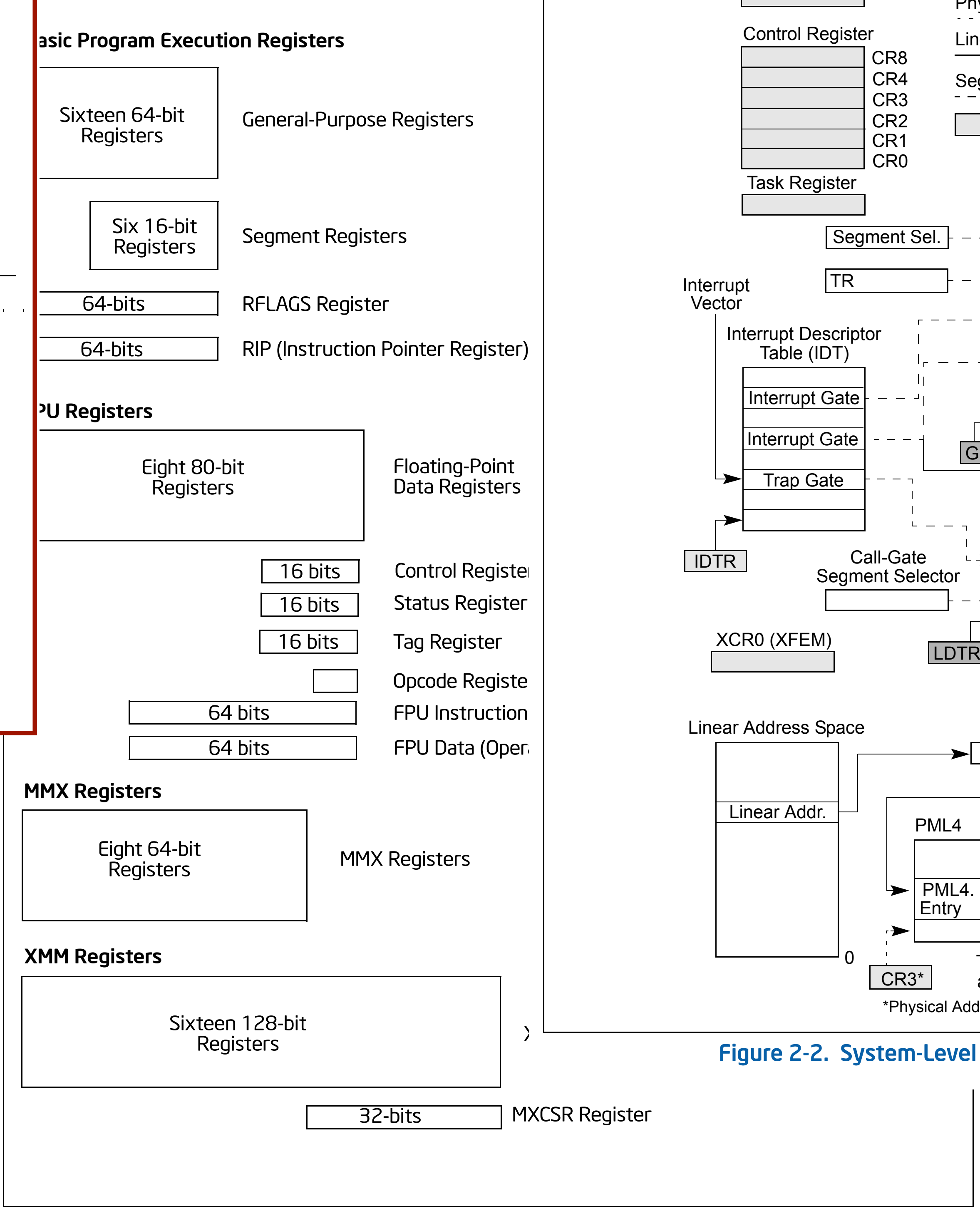


Figure 3-2. 64-Bit Mode Execution Environment

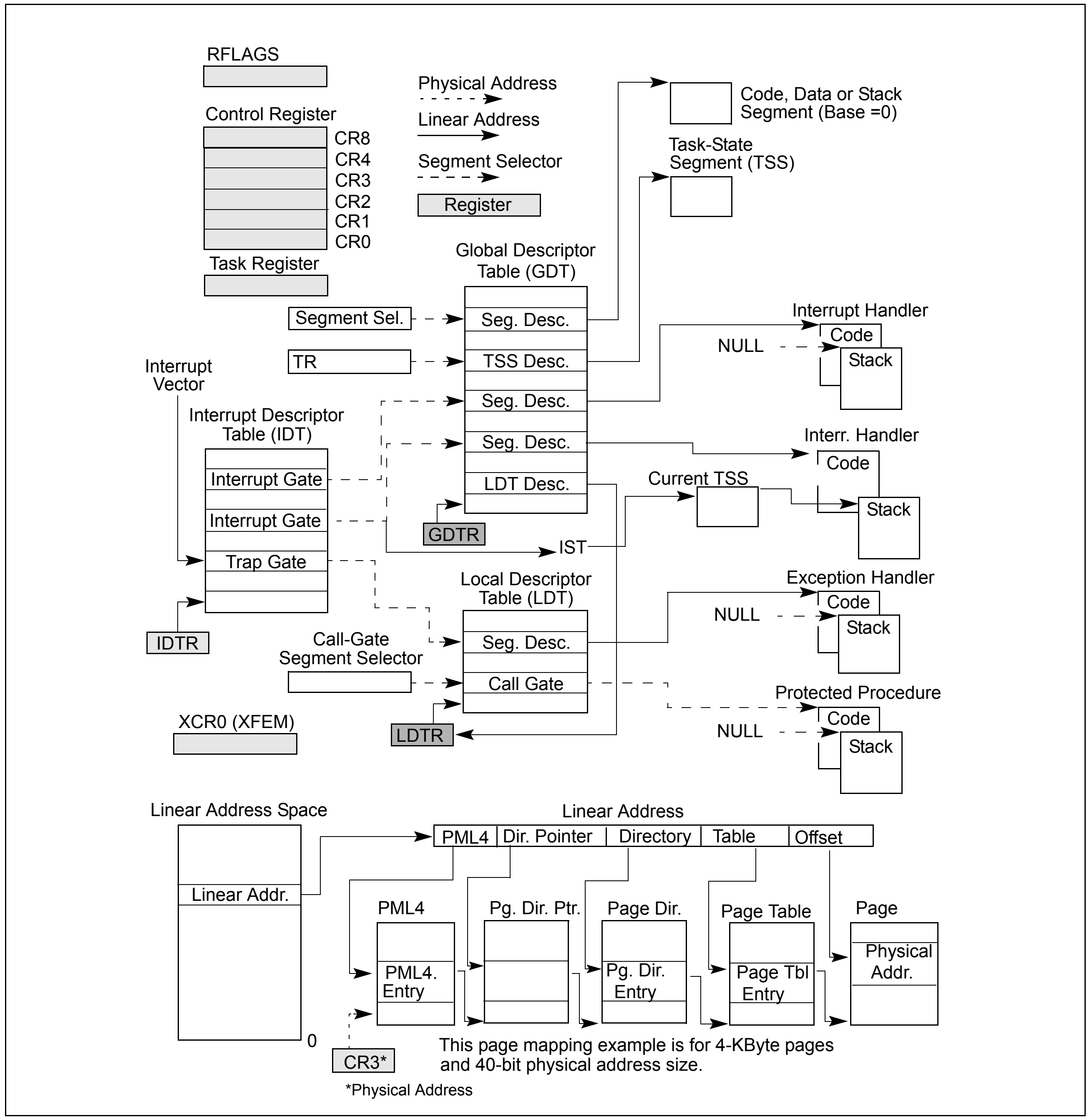


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

Source: Intel Manuals

Model Development

64-bit sub-mode

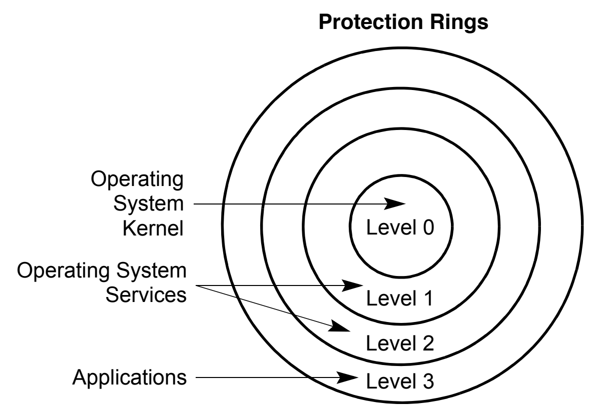
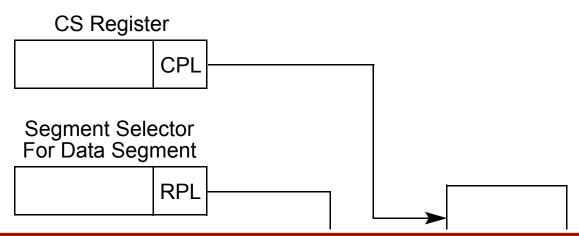


Figure 5-3. Protection Rings



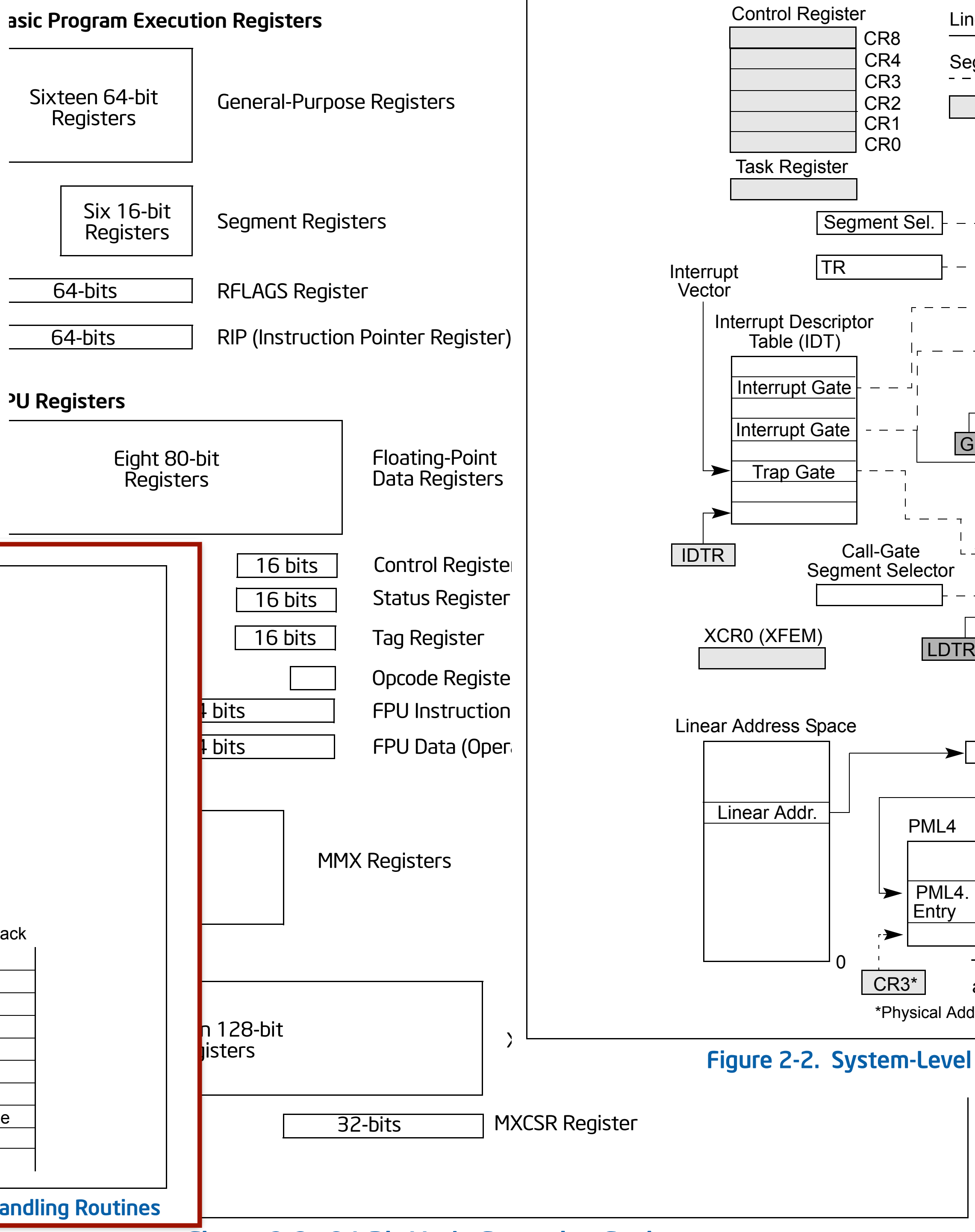


Figure 3-2. 64-Bit Mode Execution Environment

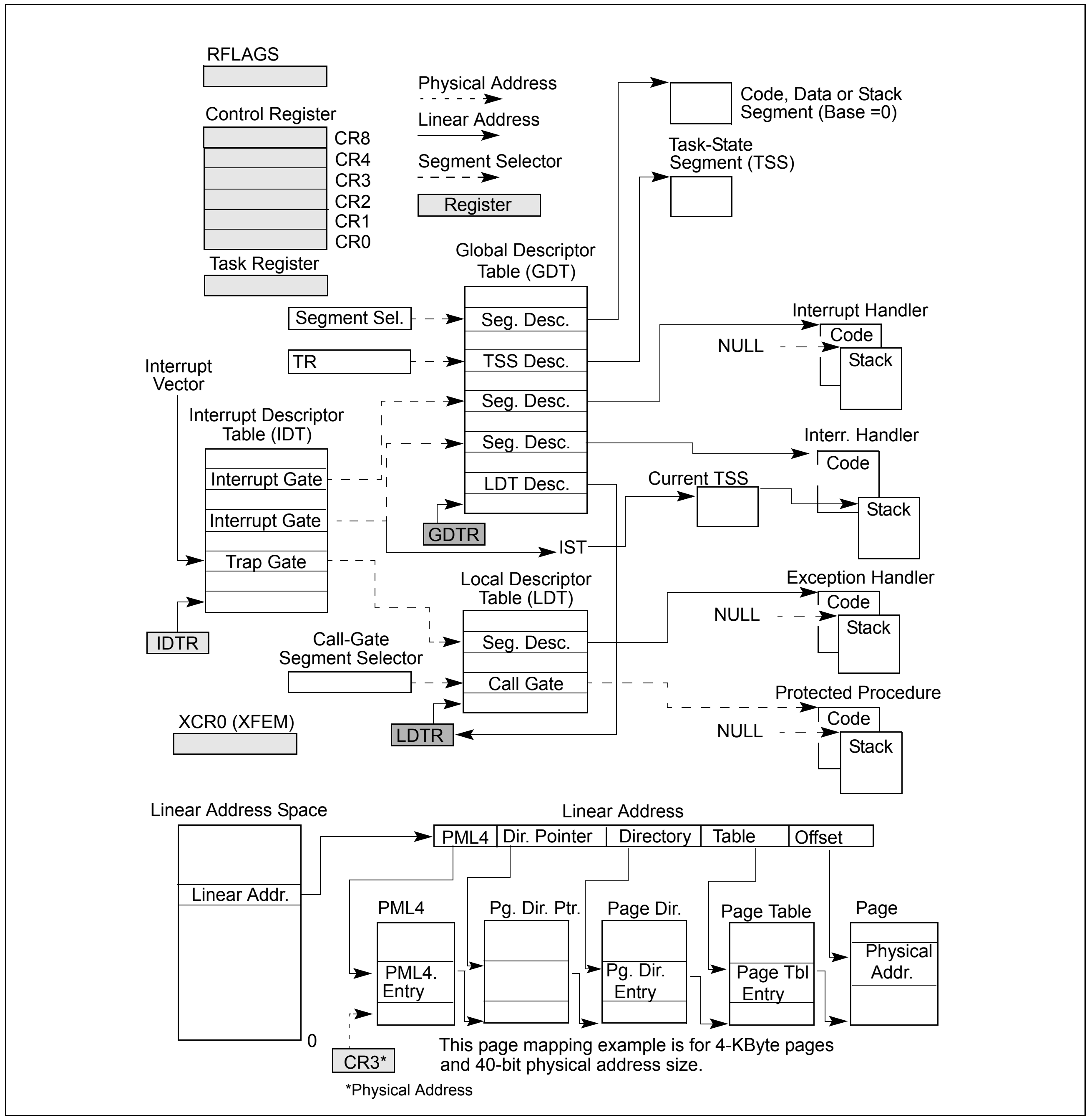
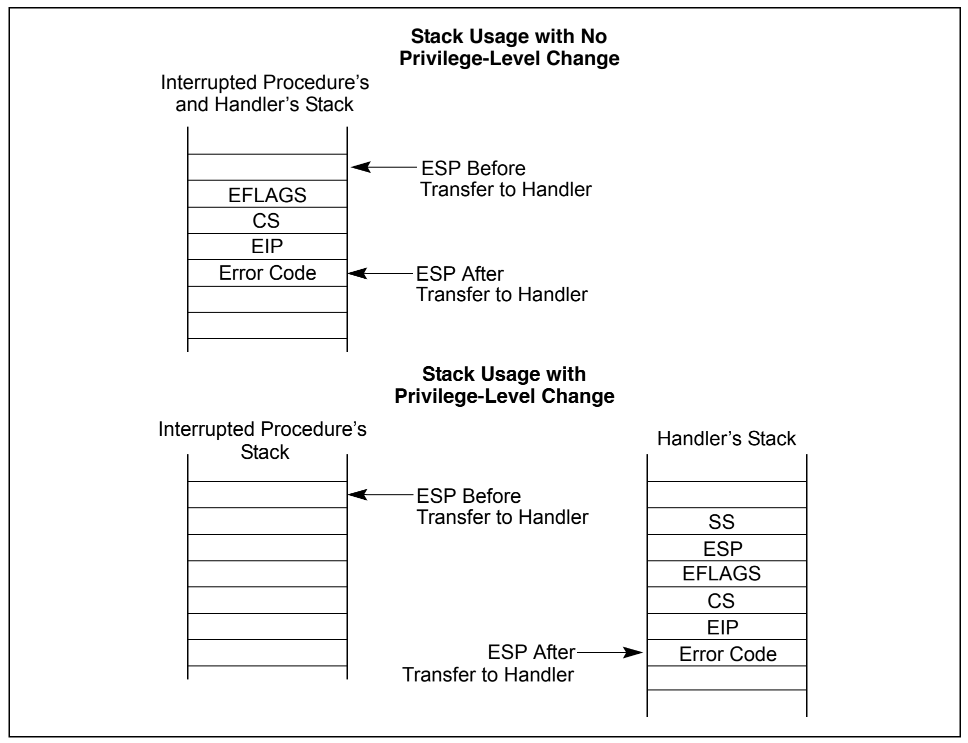


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

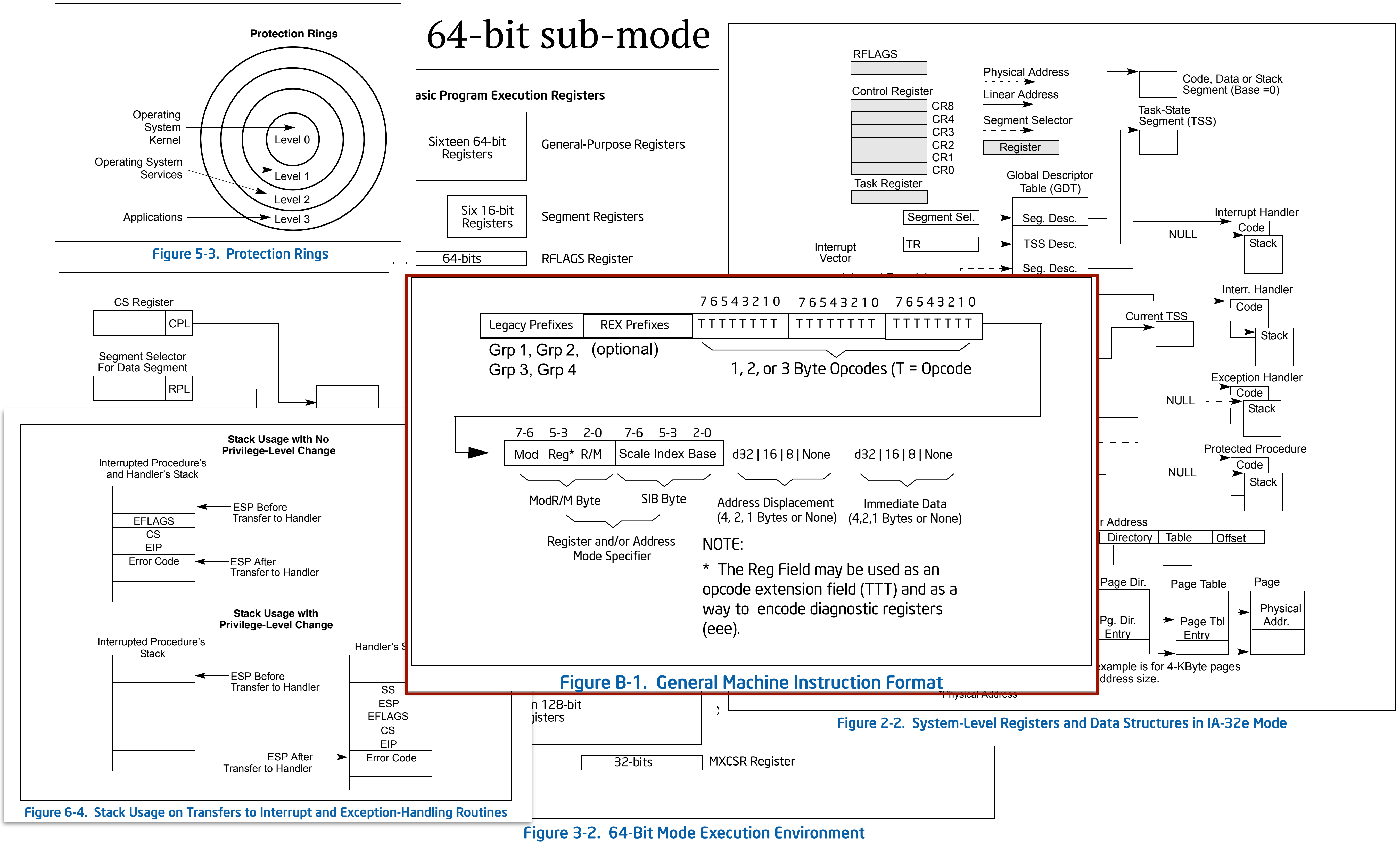


Stack Usage with No Privilege-Level Change
Interrupted Procedure's and Handler's Stack
EFLAGS
CS
EIP
Error Code
ESP Before Transfer to Handler
ESP After Transfer to Handler
Stack Usage with Privilege-Level Change
Interrupted Procedure's Stack
Handler's Stack
ESP Before Transfer to Handler
SS
ESP
EFLAGS
CS
EIP
Error Code
ESP After Transfer to Handler

Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

Source: Intel Manuals

Model Development



NOTE:

* The Reg Field may be used as an opcode extension field (TTT) and as a way to encode diagnostic registers (eee).

Figure 5-3. Protection Rings

Figure B-1. General Machine Instruction Format

Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

Figure 3-2. 64-Bit Mode Execution Environment

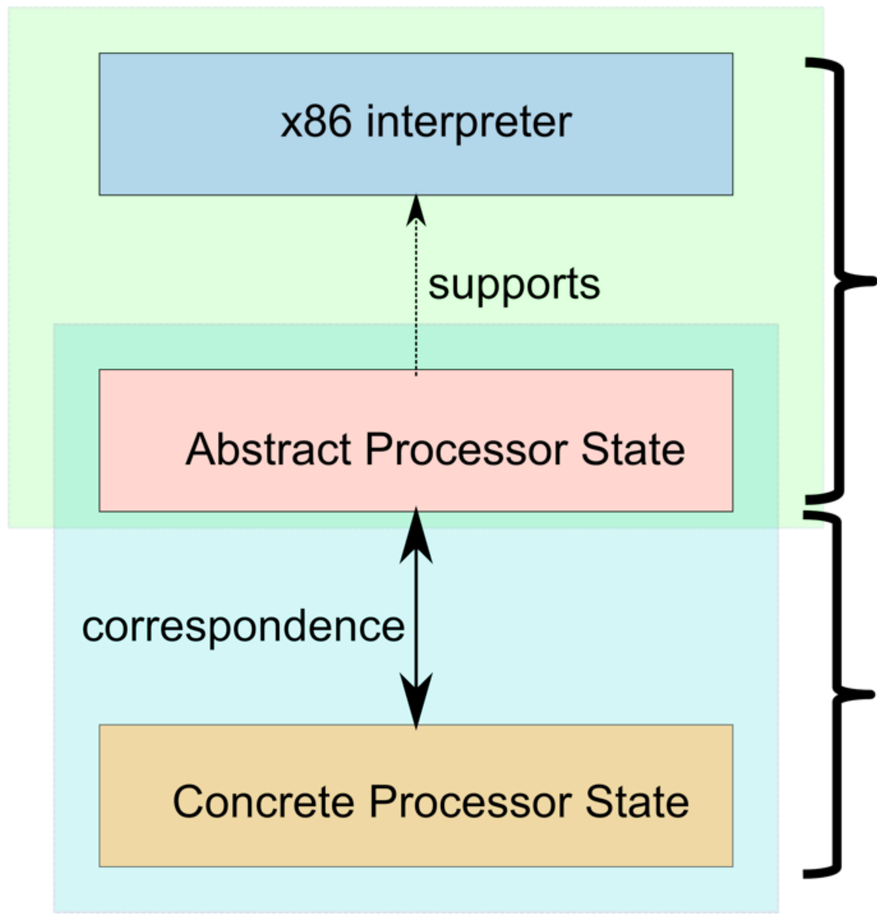
Source: Intel Manuals

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]



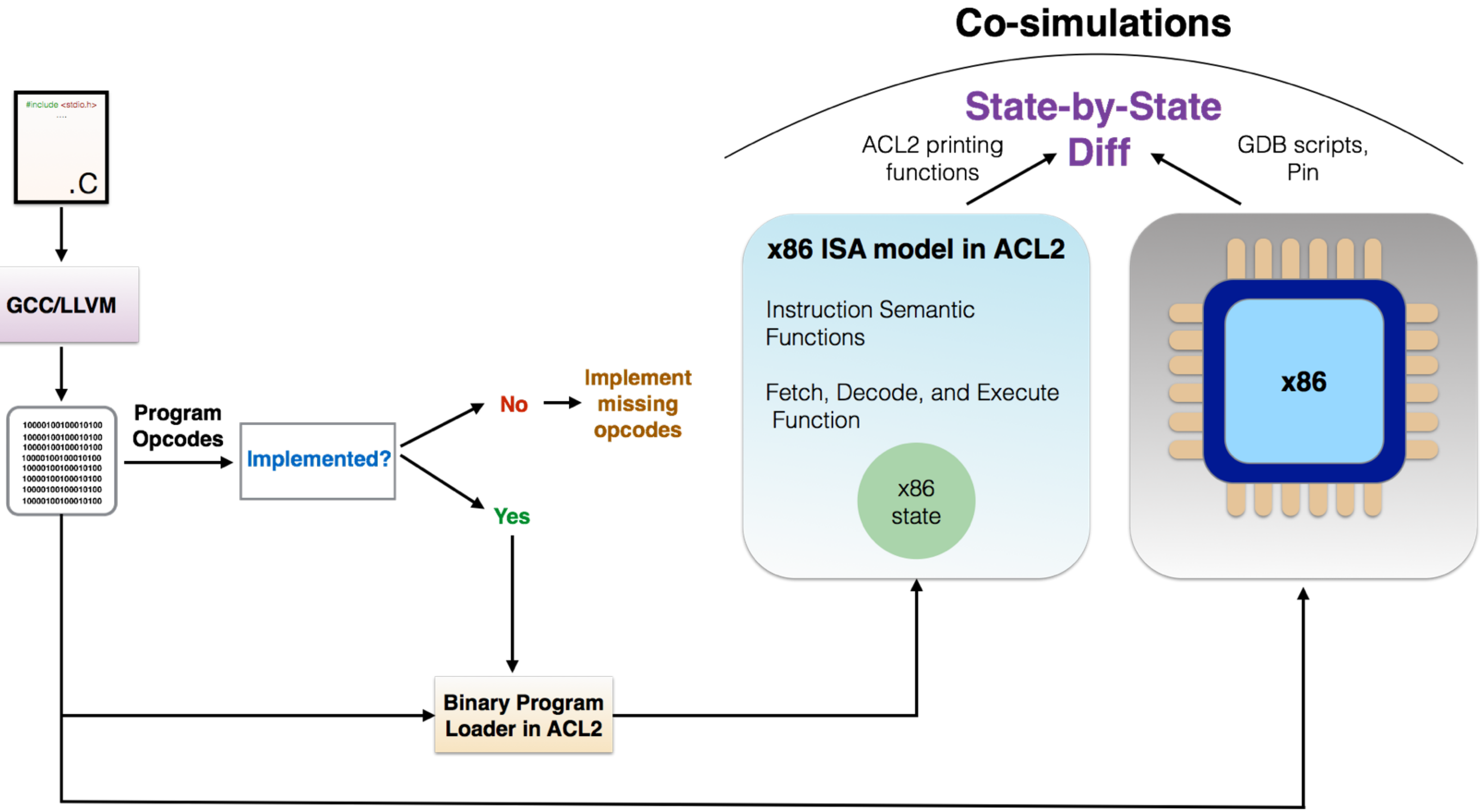
Optimized for reasoning efficiency

Optimized for execution efficiency

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.



Current Status: x86 ISA Model

- The x86 ISA model supports **400+ instructions**, including some floating-point and supervisor-mode instructions
  - Can execute almost all user-level programs emitted by GCC/LLVM
  - Successfully co-simulated a contemporary SAT solver on our model
  - Successfully simulated a supervisor-mode zero-copy program
- **IA-32e paging** for all page configurations (4K, 2M, 1G)
- **Segment-based addressing**
- Lines of Code: ~85,000 (not including blank lines)
- Simulation speed*:
  - ~3.3 million instructions/second (paging disabled)
  - ~330,000 instructions/second (with 1G pages)

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

Outline

Building a 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

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

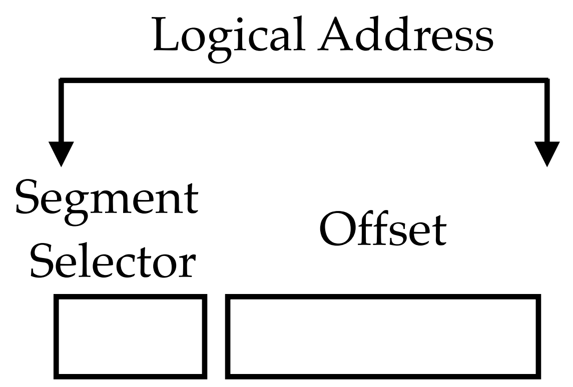
- The database should include lemmas about **reads from** and **writes to the machine state**, along with the **interactions** between these operations.

Building a Lemma Database

- System data structures, like the paging structures, are extremely complicated.
- Correct operation of a system heavily depends upon such structures.
- We need to prove lemmas that can aid in proving the following kinds of critical properties:
  - Processes are isolated from each other.
  - Page tables, including access rights, are set up correctly.

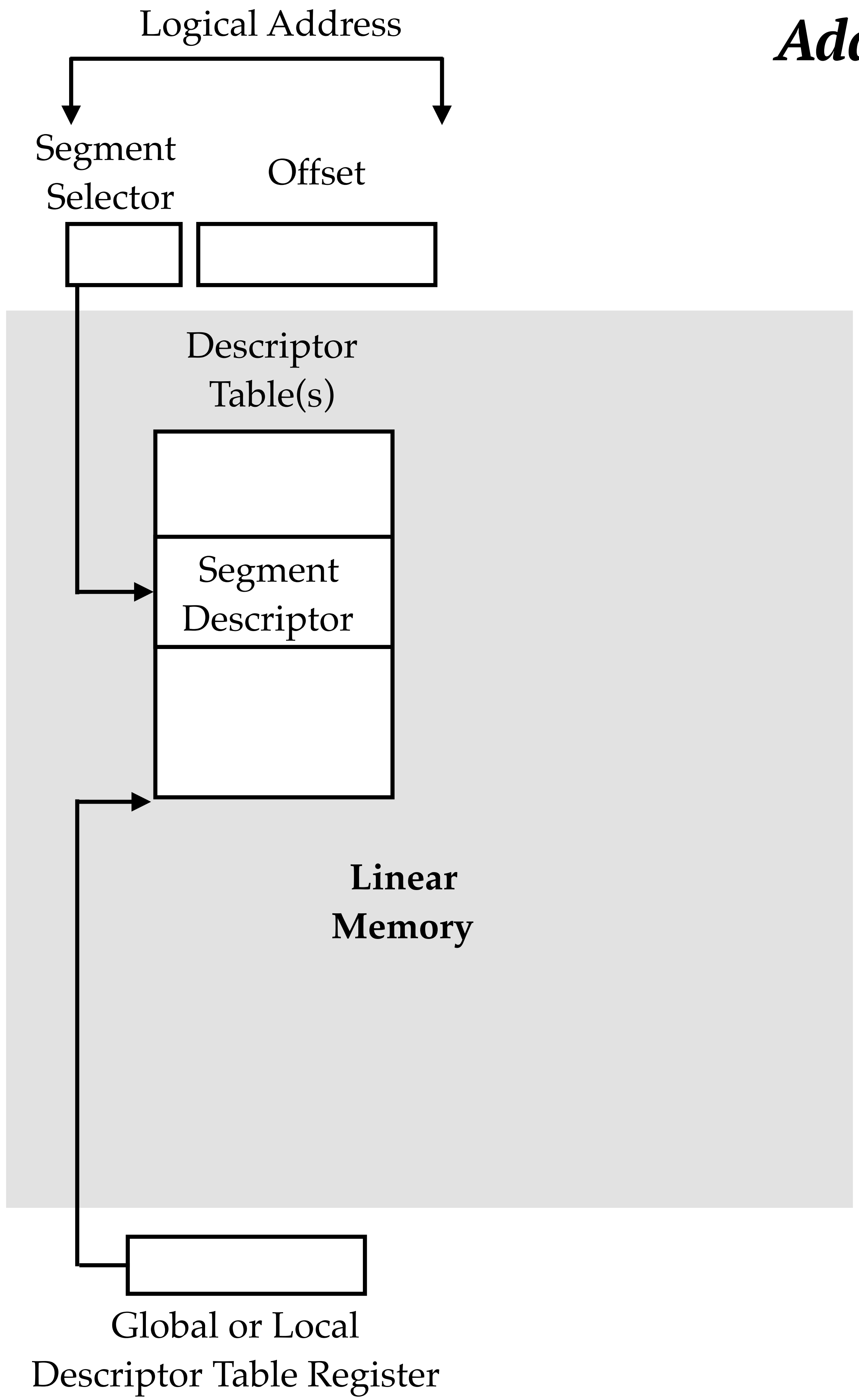
Address Translations

Address Translations



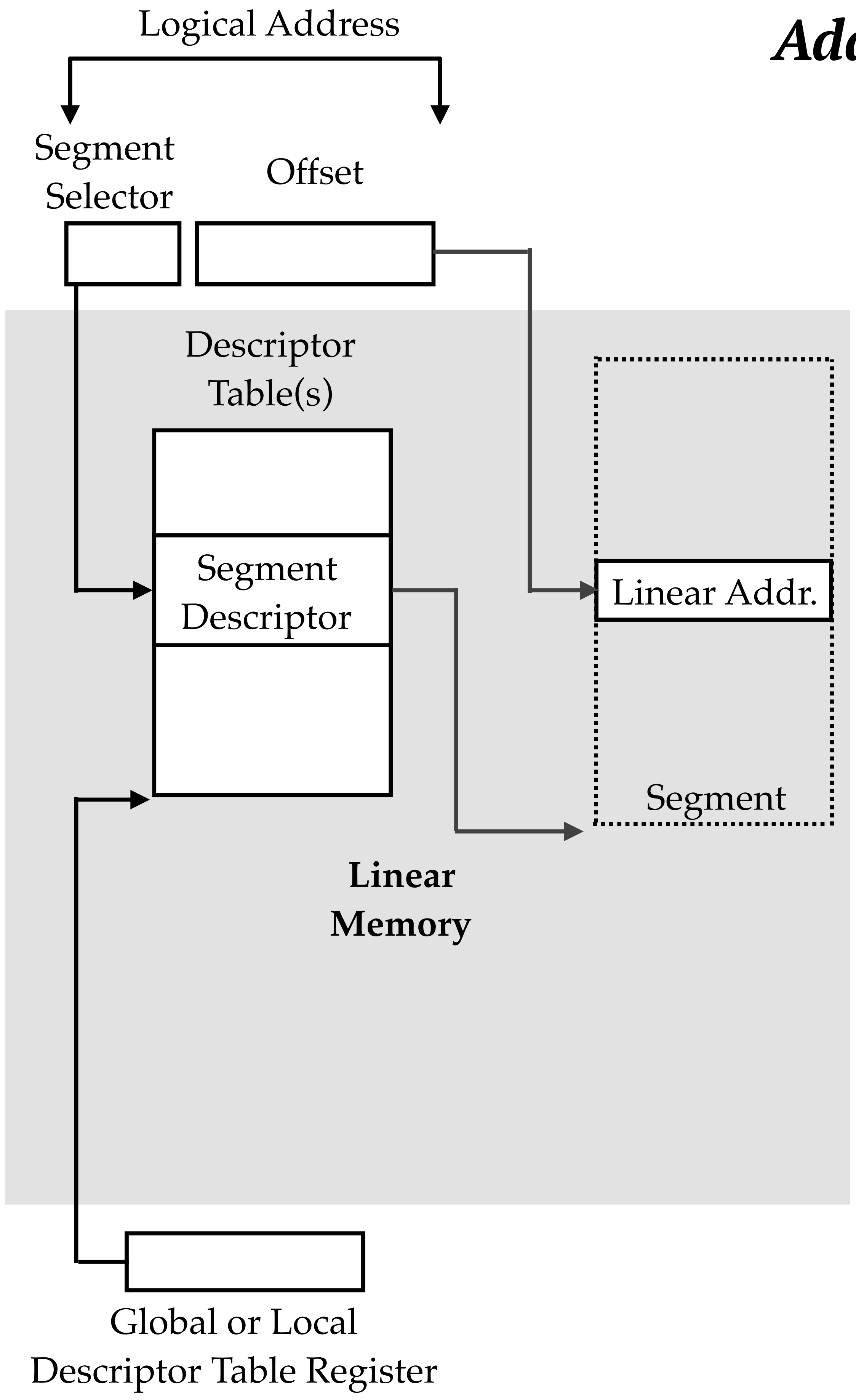
SEGMENTATION

Address Translations



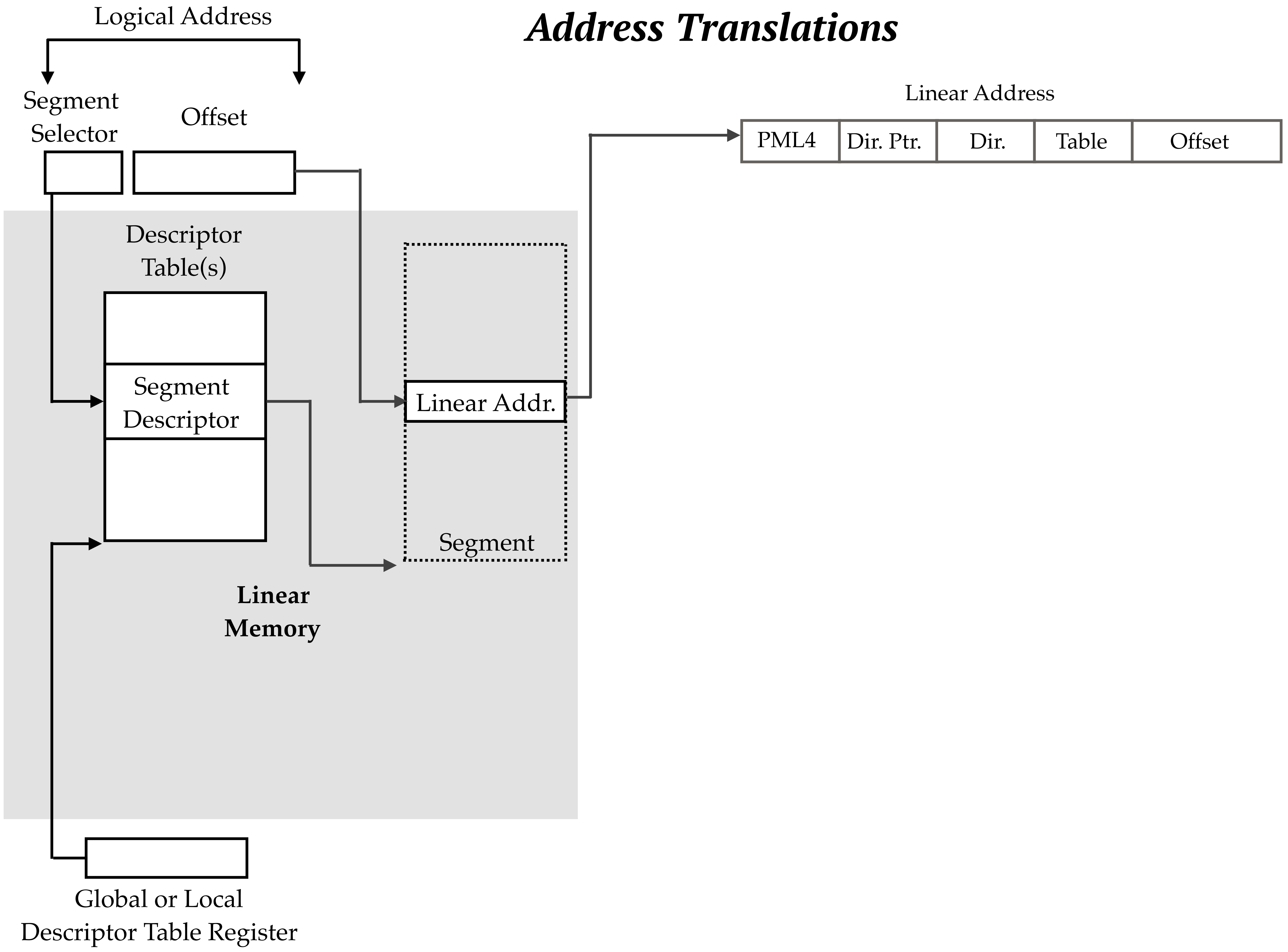
SEGMENTATION

Address Translations



SEGMENTATION

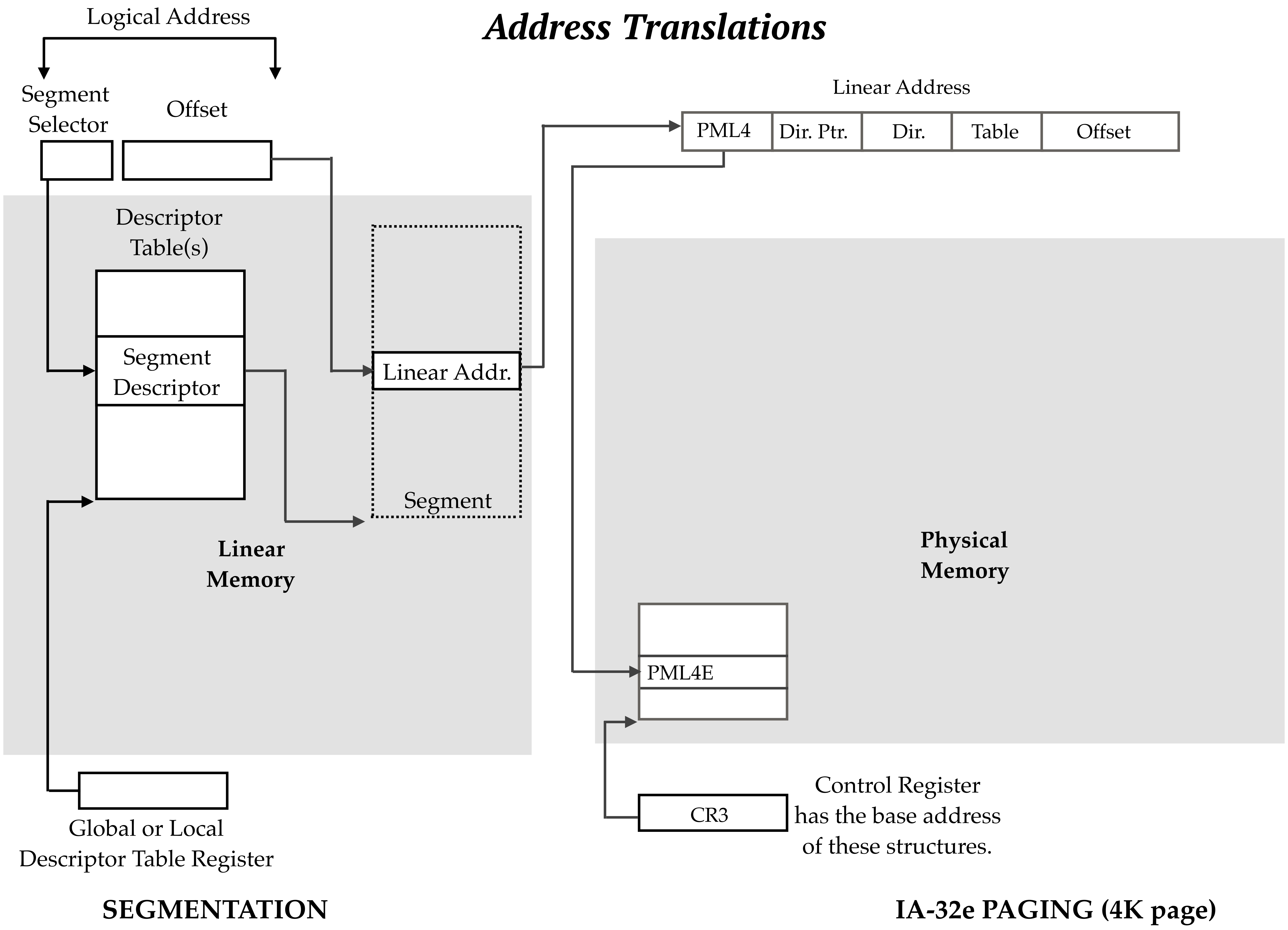
Address Translations



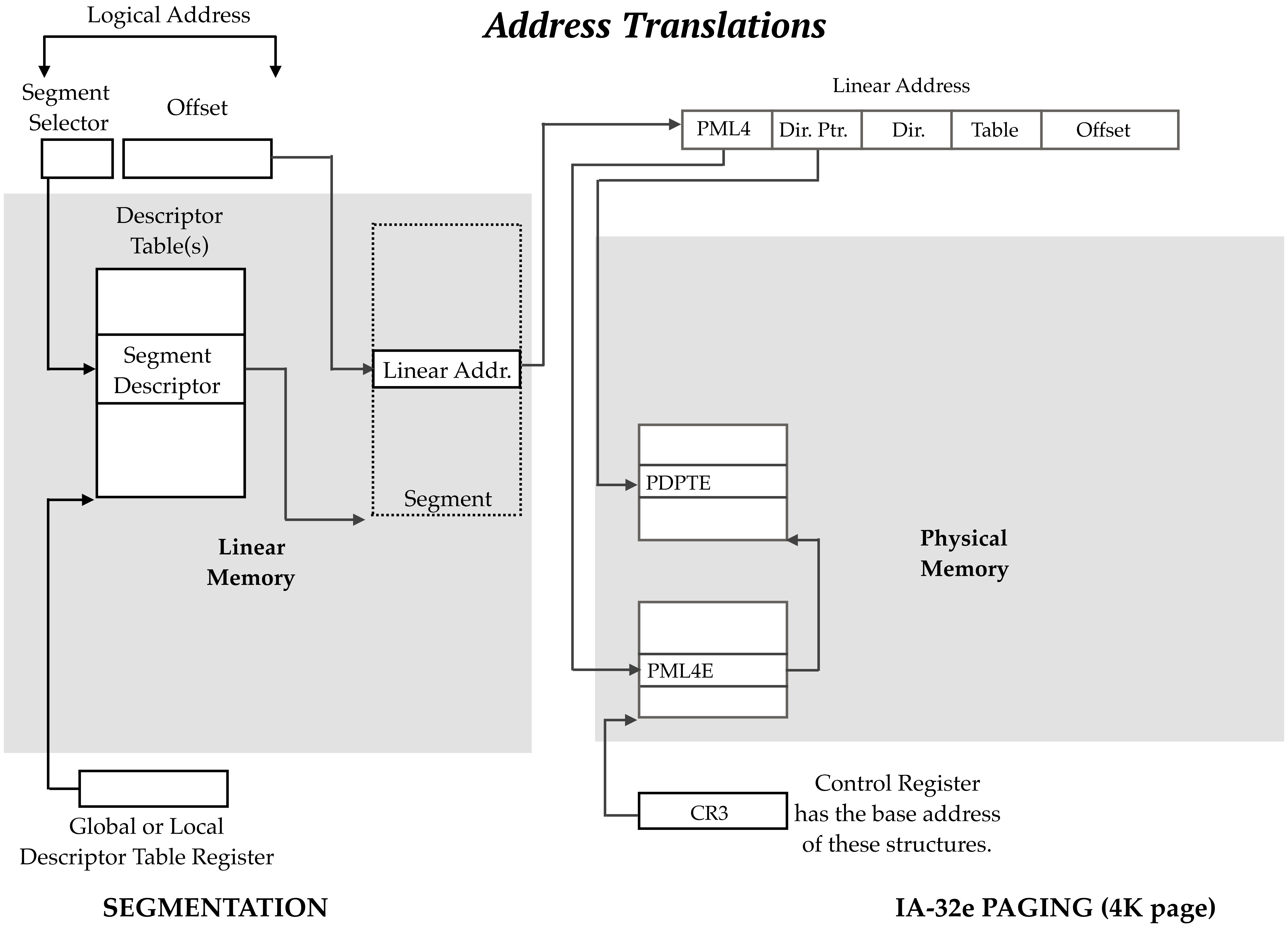
SEGMENTATION

IA-32e PAGING (4K page)

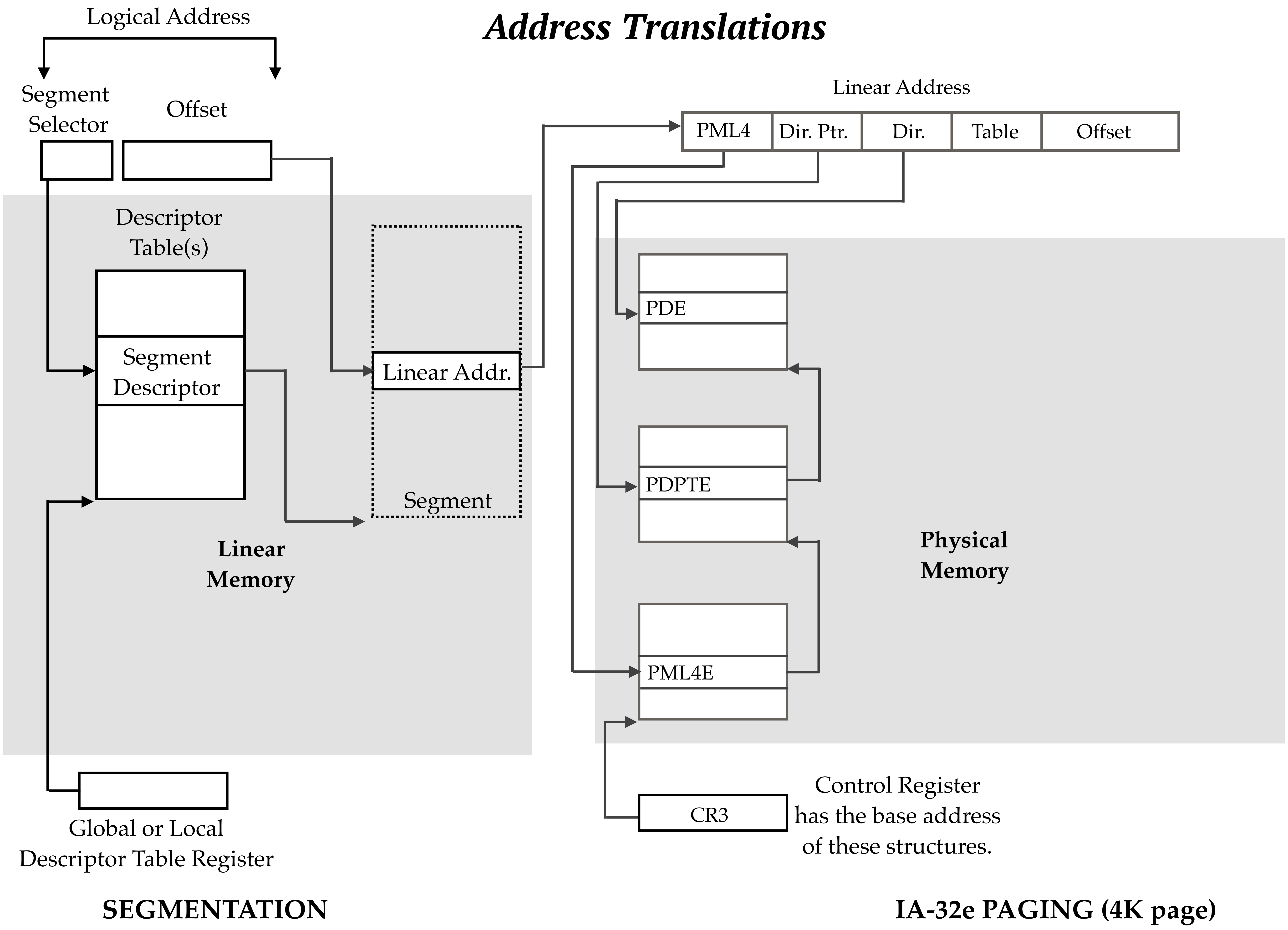
Address Translations



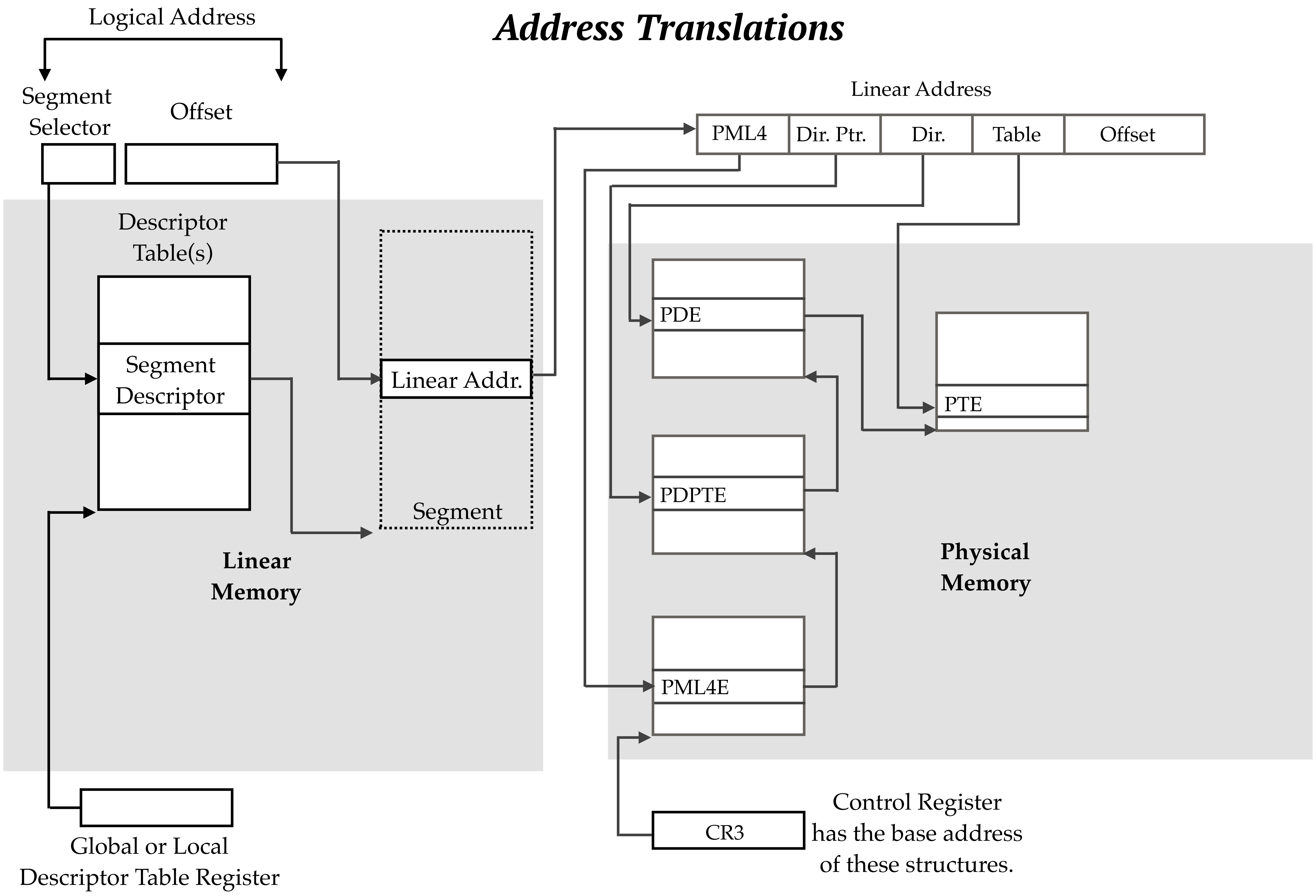
Address Translations



Address Translations



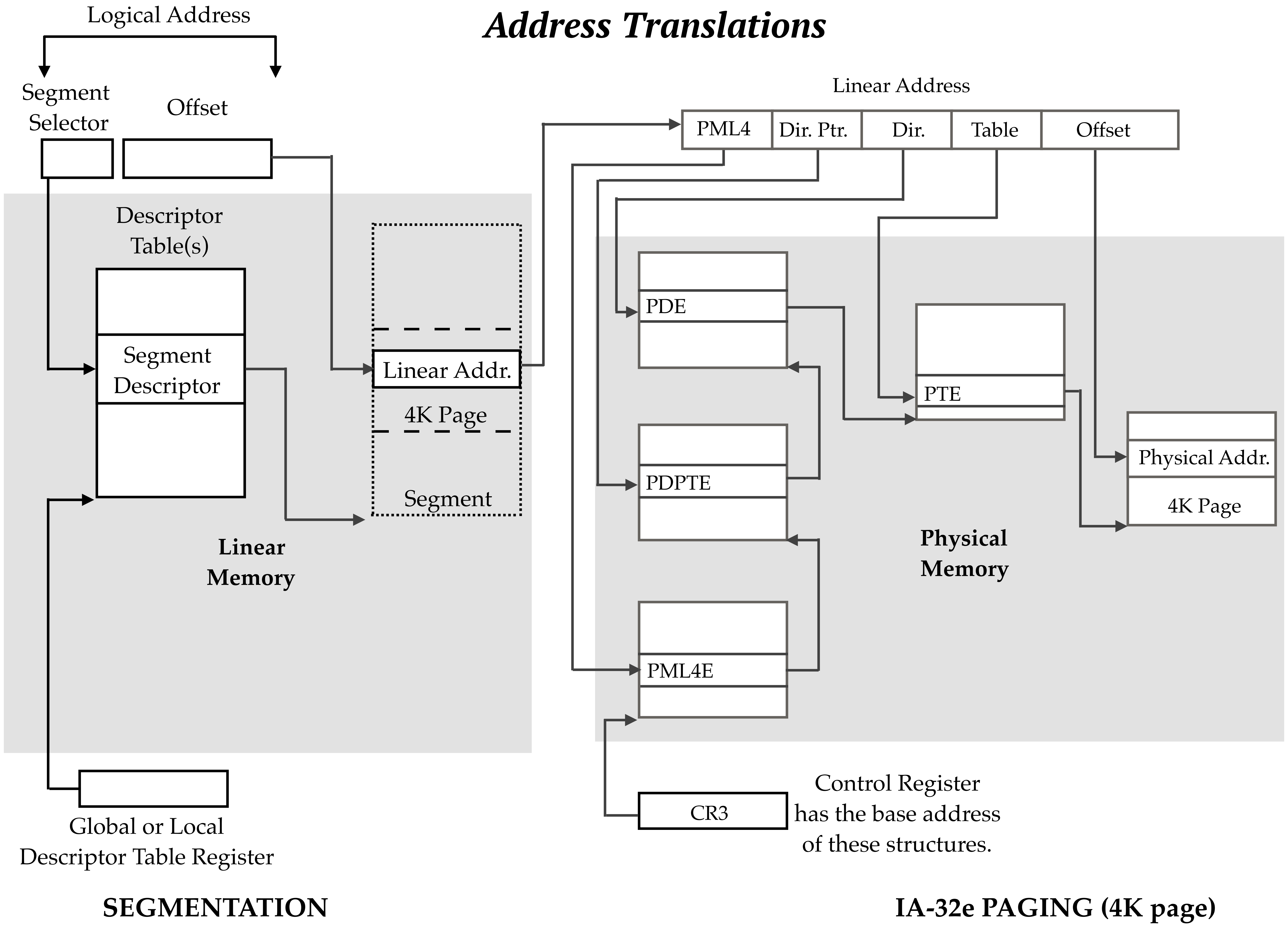
Address Translations



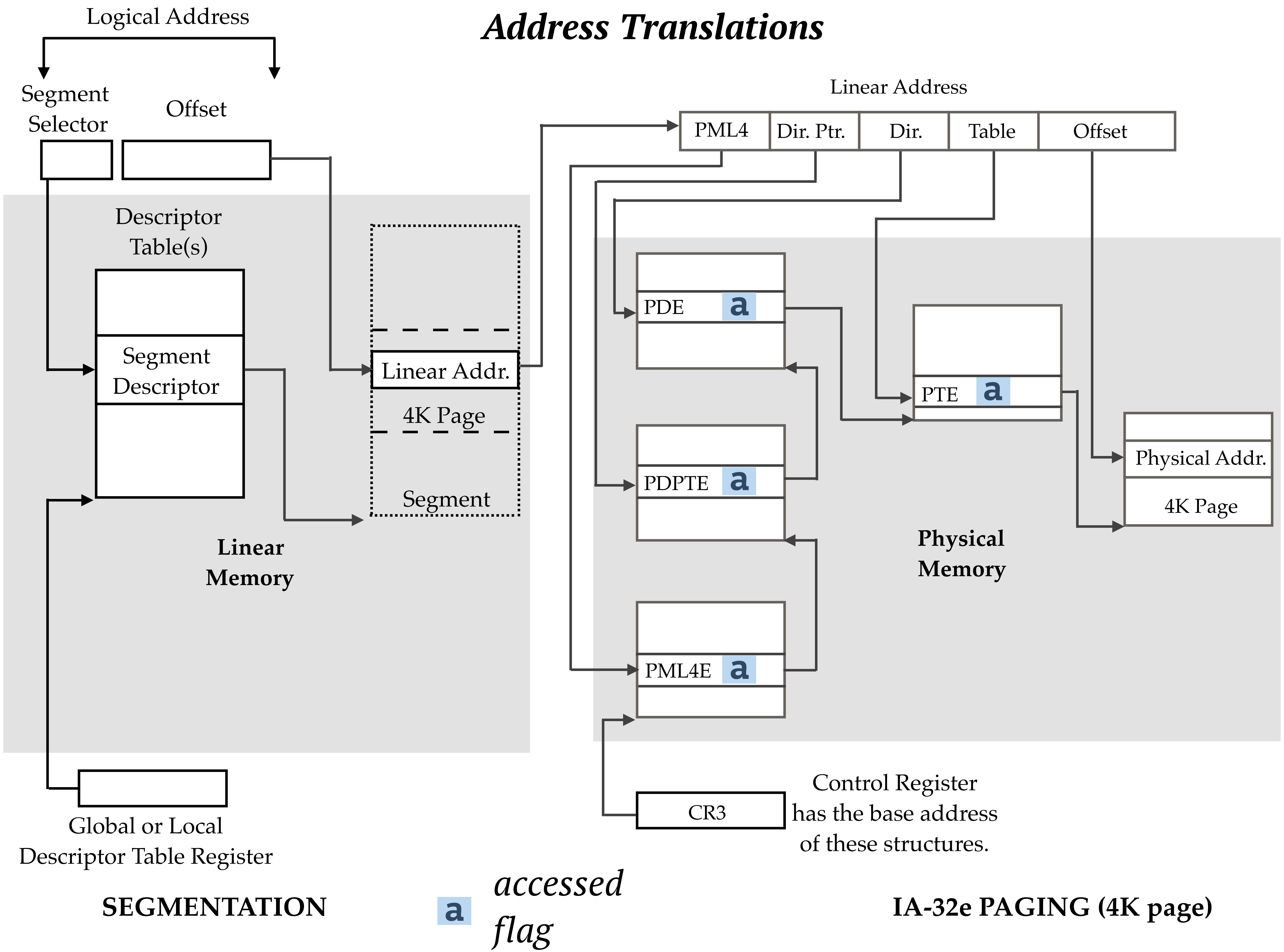
SEGMENTATION

IA-32e PAGING (4K page)

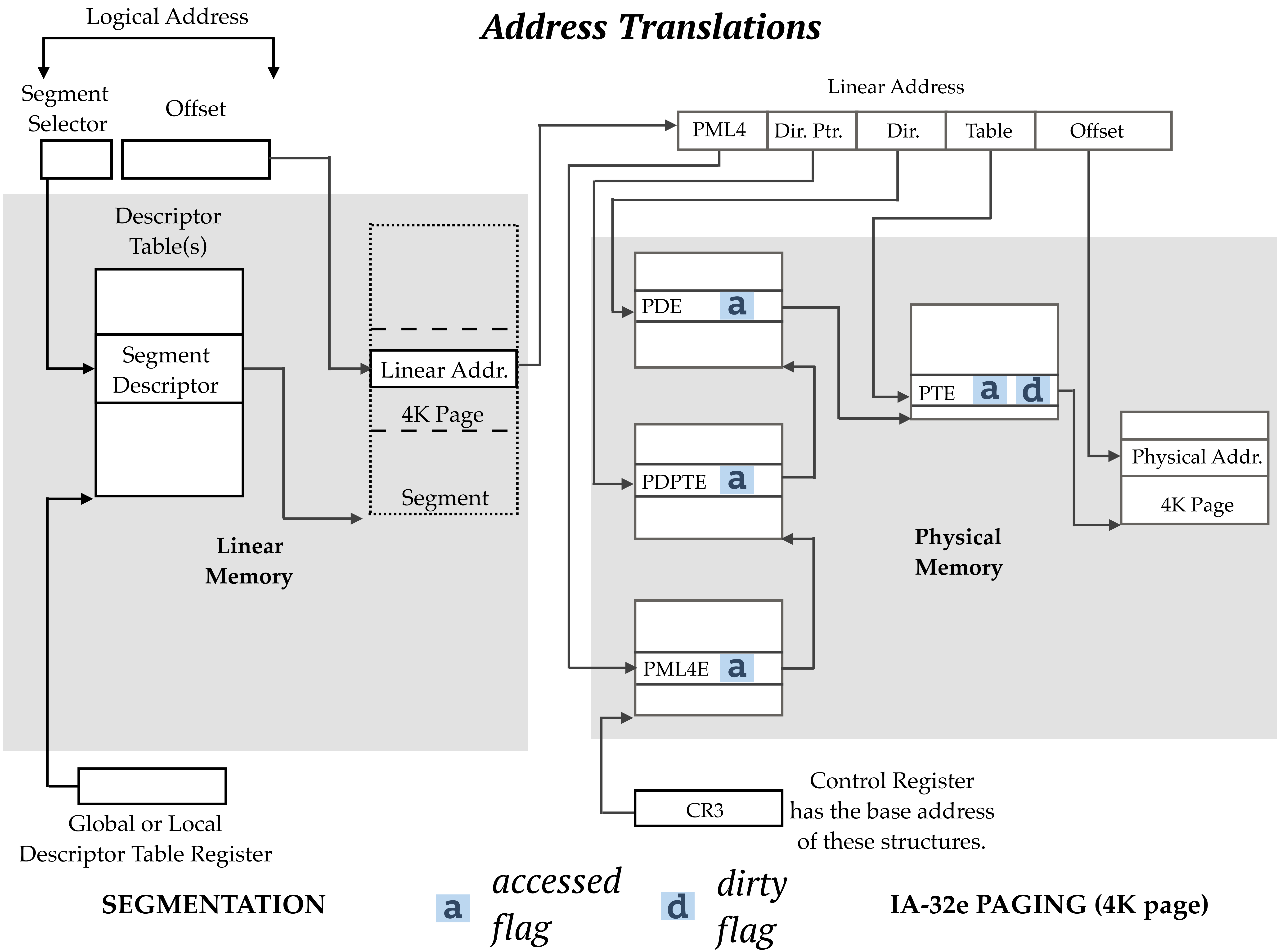
Address Translations



Address Translations



Address Translations



Current Status: Analysis Framework

- Automatically generate and prove many lemmas about reads and writes
- Libraries to reason about (non-)interference of memory regions
- Proved general lemmas about paging data structure traversals

Outline

Application Program #1: *popcount*

Automatically verify snippets of straight-line machine code using **bit-blasting** [VSTTE'13]

```
55                   push   %rbp
48 89 e5             mov    %rsp,%rbp
89 7d fc             mov    %edi,-0x4(%rbp)
8b 7d fc             mov    -0x4(%rbp),%edi
8b 45 fc             mov    -0x4(%rbp),%eax
c1 e8 01             shr    $0x1,%eax
25 55 55 55 55       and    $0x55555555,%eax
29 c7                sub    %eax,%edi
89 7d fc             mov    %edi,-0x4(%rbp)
8b 45 fc             mov    -0x4(%rbp),%eax
25 33 33 33 33       and    $0x33333333,%eax
8b 7d fc             mov    -0x4(%rbp),%edi
c1 ef 02             shr    $0x2,%edi
81 e7 33 33 33 33    and    $0x33333333,%edi
01 f8                add    %edi,%eax
89 45 fc             mov    %eax,-0x4(%rbp)
8b 45 fc             mov    -0x4(%rbp),%eax
8b 7d fc             mov    -0x4(%rbp),%edi
c1 ef 04             shr    $0x4,%edi
01 f8                add    %edi,%eax
25 0f 0f 0f 0f       and    $0xf0f0f0f,%eax
69 c0 01 01 01 01    imul   $0x1010101,%eax,%eax
c1 e8 18             shr    $0x18,%eax
89 45 fc             mov    %eax,-0x4(%rbp)
8b 45 fc             mov    -0x4(%rbp),%eax
5d                   pop    %rbp
c3                   retq
```

Application Program #1: *popcount*

Automatically verify snippets of straight-line machine code using **bit-blasting** [VSTTE'13]

```
int popcount_32 (unsigned int v)
{
  // From Sean Anderson's Bit-Twiddling Hacks
  v = v - ((v >> 1) & 0x55555555);
  v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
  v = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
  return(v);
}
```

```
8b 45 fc            mov    -0x4(%rbp),%eax
25 33 33 33 33      and    $0x33333333,%eax
8b 7d fc            mov    -0x4(%rbp),%edi
c1 ef 02            shr    $0x2,%edi
81 e7 33 33 33 33   and    $0x33333333,%edi
01 f8               add    %edi,%eax
89 45 fc            mov    %eax,-0x4(%rbp)
8b 45 fc            mov    -0x4(%rbp),%eax
8b 7d fc            mov    -0x4(%rbp),%edi
c1 ef 04            shr    $0x4,%edi
01 f8               add    %edi,%eax
25 0f 0f 0f 0f      and    $0xf0f0f0f,%eax
69 c0 01 01 01 01   imul   $0x1010101,%eax,%eax
c1 e8 18            shr    $0x18,%eax
89 45 fc            mov    %eax,-0x4(%rbp)
8b 45 fc            mov    -0x4(%rbp),%eax
5d                  pop    %rbp
c3                  retq
```

Application Program #1: *popcount*

Automatically verify snippets of straight-line machine code using **bit-blasting** [VSTTE'13]

```
int popcount_32 (unsigned int v)
{
  // From Sean Anderson's Bit-Twiddling Hacks
  v = v - ((v >> 1) & 0x55555555);
  v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
  v = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
  return(v);
}
```

```
8b 45 fc            mov     -0x4(%rbp),%eax
25 33 33 33 33      and     $0x33333333,%eax
8b 7d fc            mov     -0x4(%rbp),%edi
c1 ef 02            shr     $0x2,%edi
81 e7 33 33 33 33   and     $0x33333333,%edi
01 f8               add     %edi,%eax
89 45 fc            mov     %eax,-0x4(%rbp)
8b 45 fc            mov     -0x4(%rbp),%eax
8b 7d fc            mov     -0x4(%rbp),%edi
c1 ef 04            shr     $0x4,%edi
01 f8               add     %edi,%eax
25 0f 0f 0f 0f      and     $0xf0f0f0f,%eax
69 c0 01 01 01 01   imul    $0x1010101,%eax,%eax
c1 e8 18            shr     $0x18,%eax
89 45 fc            mov     %eax,-0x4(%rbp)
8b 45 fc            mov     -0x4(%rbp),%eax
5d                  pop     %rbp
c3                  retq
```

unsigned int

```
RAX = popcount(input)
```

specification function

```
popcount(x):

if (x <= 0) then
    return 0
else
    lsb := x & 1
    x   := x >> 1
    return (lsb + popcount(x))
endif
```

Application Program #2: *word-count*

- Proved the functional correctness of a word-count program that reads input from the user using `read` system calls. System calls are ***non-deterministic*** for application programs. [FMCAD'14]

```
55                      push   %rbp
48 89 e5                mov    %rsp,%rbp
48 83 ec 20             sub    $0x20,%rsp
c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
c7 45 e8 00 00 00 00    movl   $0x0,-0x18(%rbp)
c7 45 ec 00 00 00 00    movl   $0x0,-0x14(%rbp)
c7 45 f0 00 00 00 00    movl   $0x0,-0x10(%rbp)
c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%rbp)
e8 90 ff ff ff          callq  <_gc>
…
05 01 00 00 00          add    $0x1,%eax
89 45 f0                mov    %eax,-0x10(%rbp)
e9 00 00 00 00          jmpq   <_main+0xb8>
e9 6e ff ff ff          jmpq   <_main+0x2b>
31 c0                   xor    %eax,%eax
48 83 c4 20             add    $0x20,%rsp
5d                      pop    %rbp
c3                      retq
```

Application Program #2: *word-count*

- Proved the functional correctness of a word-count program that reads input from the user using `read` system calls. System calls are ***non-deterministic*** for application programs. [FMCAD'14]

Specification for counting the # of characters in `str`:

```
ncSpec(offset, str, count):

 if (well-formed(str) && offset < len(str)) then
    c := str[offset]
    if (c == EOF) then
       return count
    else
       count := (count + 1) mod 2^32
       ncSpec(1 + offset, str, count)
    endif
 endif
```

```
55                          push    %rbp
48 89 e5                    mov     %rsp,%rbp
48 83 ec 20                 sub     $0x20,%rsp
                00 00       movl    $0x0,-0x4(%rbp)
                00 00       movl    $0x0,-0x18(%rbp)
                00 00       movl    $0x0,-0x14(%rbp)
                00 00       movl    $0x0,-0x10(%rbp)
                00 00       movl    $0x0,-0xc(%rbp)
                            callq   <_gc>

                            add     $0x1,%eax
                            mov     %eax,-0x10(%rbp)
                            jmpq    <_main+0xb8>
                            jmpq    <_main+0x2b>
                            xor     %eax,%eax
                            add     $0x20,%rsp
```

Functional Correctness Theorem: Values computed by specification functions on standard input are found in the expected memory locations of the final x86 state.

Application Program #2: *word-count*

Other properties verified using our machine-code framework:

- **Resource Usage:**
  - Program and its stack are disjoint for all inputs.
  - Irrespective of the input, program uses a fixed amount of memory.
- **Security:**
  - Program does not modify unintended regions of memory.

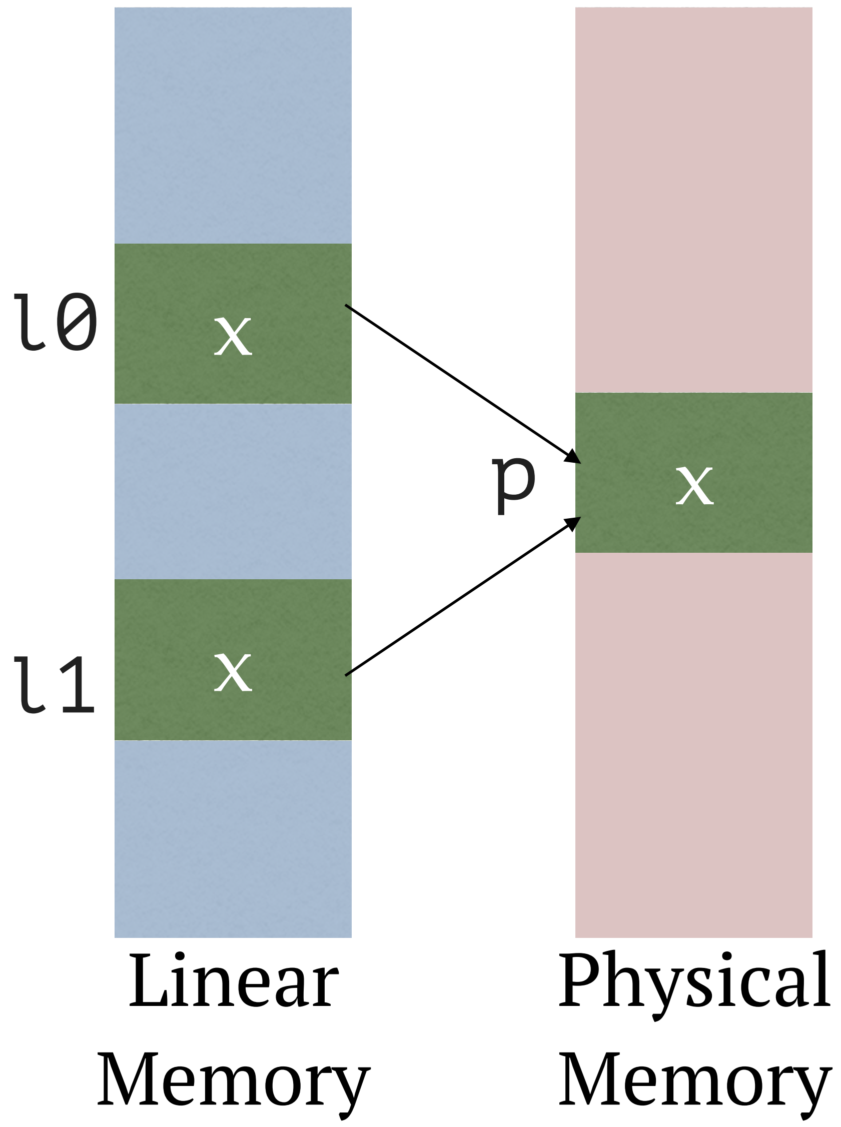
System Program: *zero-copy*

Specification:
Copy data `x` from 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`.

- Modifications to address mapping
- Access control management

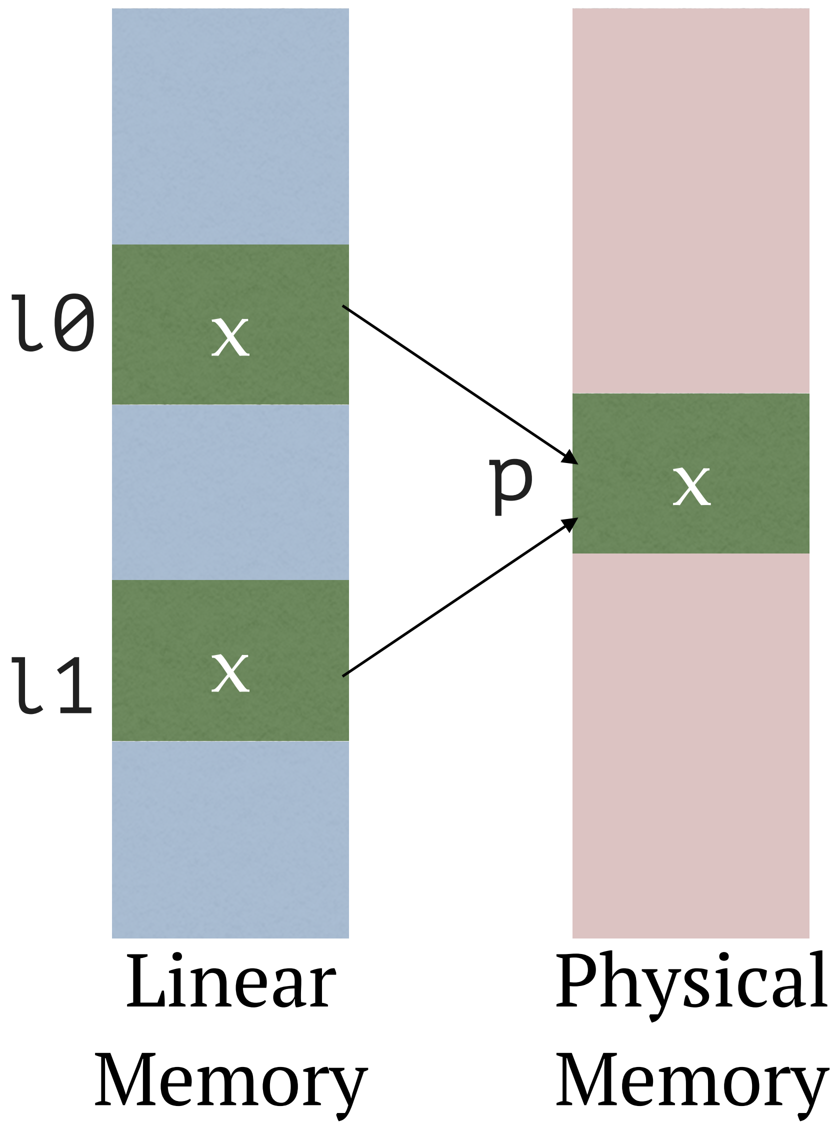


System Program: *zero-copy*

Proved that the **implementation of a zero-copy program** meets the **specification of a simple copy operation**.

For simplicity, marking of paging structures during their traversal was turned off, i.e., no accessed and dirty bit updates were allowed for this proof.

We are currently porting this proof over to a more accurate x86 model, which characterizes updates to accessed and dirty bits as well.



Outline

- Motivation
- Project Description
  - [1] Developing an x86 ISA Model
  - [2] Building a Machine-Code Analysis Framework
  - [3] Verifying Application and System Programs
- **Future Work & Conclusion**
- Accessing Source Code + Documentation

Future Work

- **Run a 64-bit FreeBSD kernel on our x86 ISA model**
  - This involves identifying and implementing relevant instructions, call gates, supporting task management, etc.
- Develop **lemma libraries to reason about kernel code**
  - This involves developing automated reasoning infrastructure for page table walks, access rights, etc.
- **Identify and prove critical invariants** in kernel code
  - This includes proving the correctness of context switches, privilege escalations, etc.

We look forward to collaborating with the FreeBSD® community!

Conclusion

It is essential to state and prove properties related to behavior, security, and resource usage; bug-hunting can only take us so far. This task is **within the scope of mechanized theorem proving**, as is evidenced by its use by our collaborators in the government and the industry to prove complex properties about complex systems.

Although full verification of all software is the ultimate goal, the focus for the coming years is to create ***islands of trust***, i.e., parts of the system for which complex properties have been formally verified.

Accessing Source Code + Documentation

The `x86isa` project is available under **BSD 3-Clause license** as a part of the **ACL2 Community Books** project.

Go to https://github.com/acl2/acl2/
and see `books/projects/x86isa/README` for details.

Also, documentation and user's manual is available online at
www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____X86ISA

Some Publications

- *Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann.*
  Abstract Stobjs and Their Application to ISA Modeling
  In ACL2 Workshop, 2013
- *Shilpi Goel and Warren A. Hunt, Jr.*
  Automated Code Proofs on a Formal Model of the x86
  In Verified Software: Theories, Tools, Experiments (VSTTE), 2013
- *Shilpi Goel, Warren A. Hunt, Jr., Matt Kaufmann, and Soumava Ghosh.*
  Simulation and Formal Verification of x86 Machine-Code Programs That Make System Calls
  In Formal Methods in Computer-Aided Design (FMCAD), 2014
- *Shilpi Goel, Warren A. Hunt, Jr., and Matt Kaufmann.*
  Engineering a Formal, Executable x86 ISA Simulator for Software Verification
  To appear in Provably Correct Software (ProCoS), 2015

Extra Slides

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

Motivation: x86 Machine-Code Verification

- ***Why not high-level code verification?***
  - x High-level verification frameworks do not address compiler bugs
    - ✓ Verified/verifying compilers can help
    - x But these compilers typically generate inefficient code
  - 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

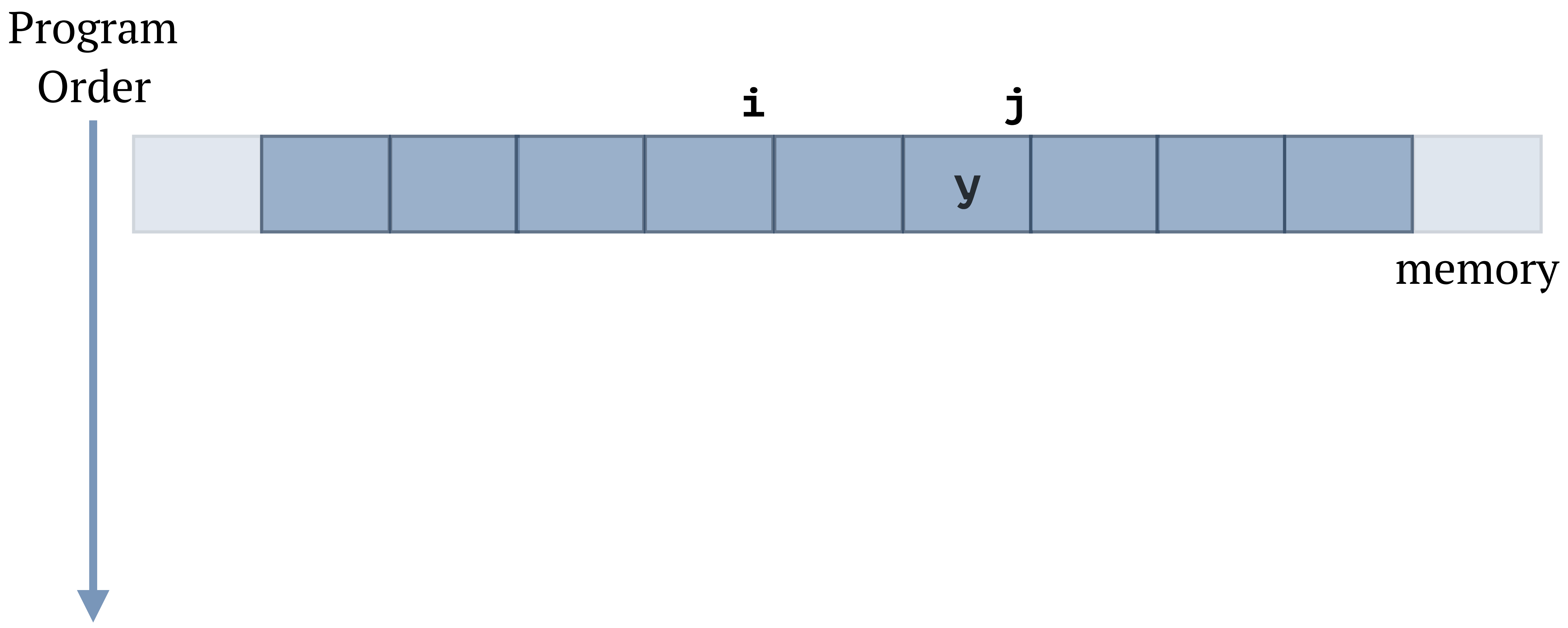
Building a Lemma Database

Three kinds of theorems:

- Read-over-Write Theorems
- Write-over-Write Theorems
- Preservation Theorems

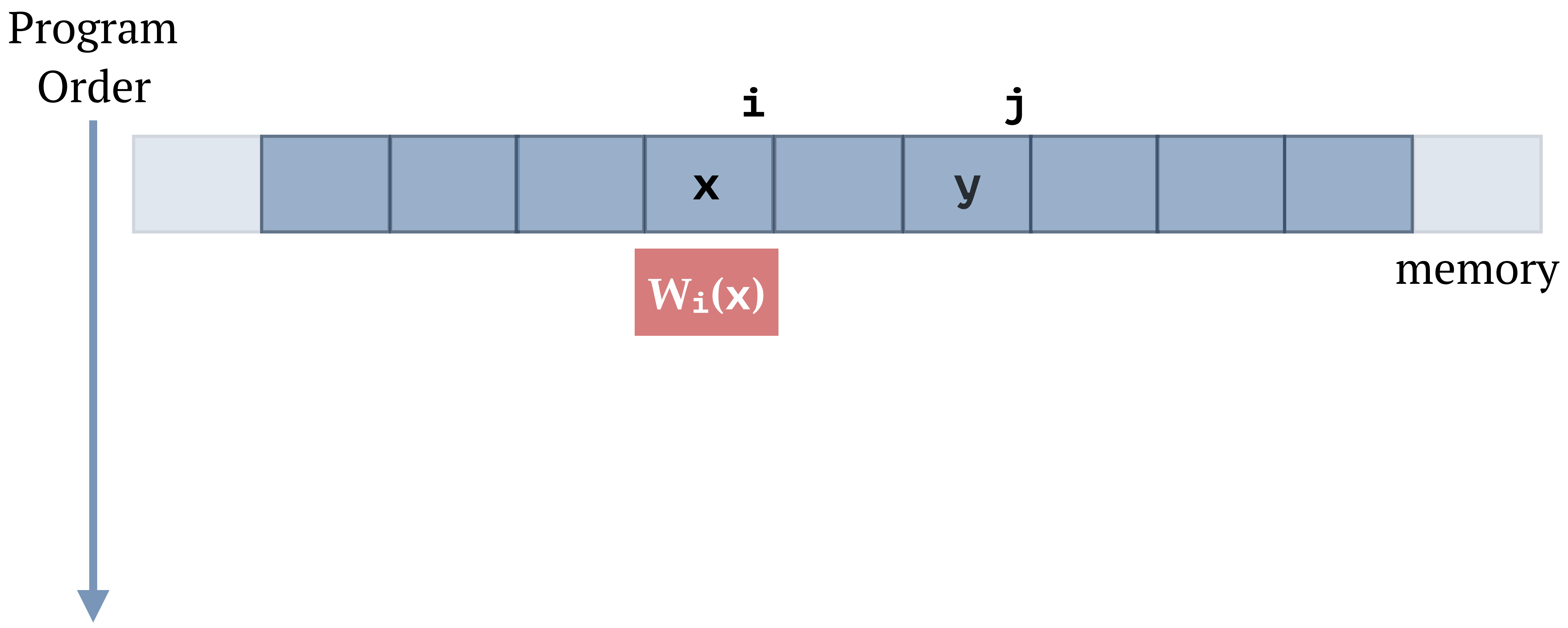
Read-over-Write Theorem: #1

non-interference

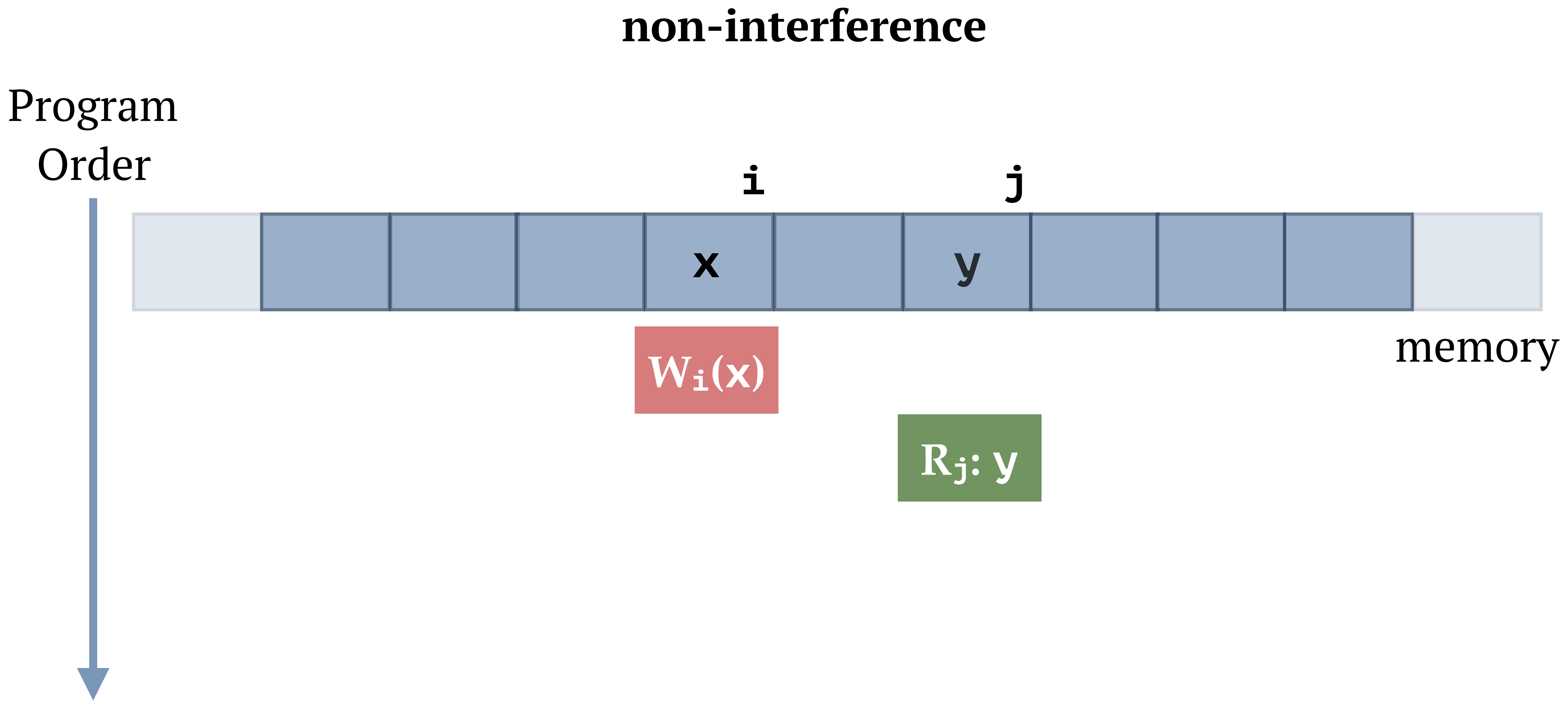


Read-over-Write Theorem: #1

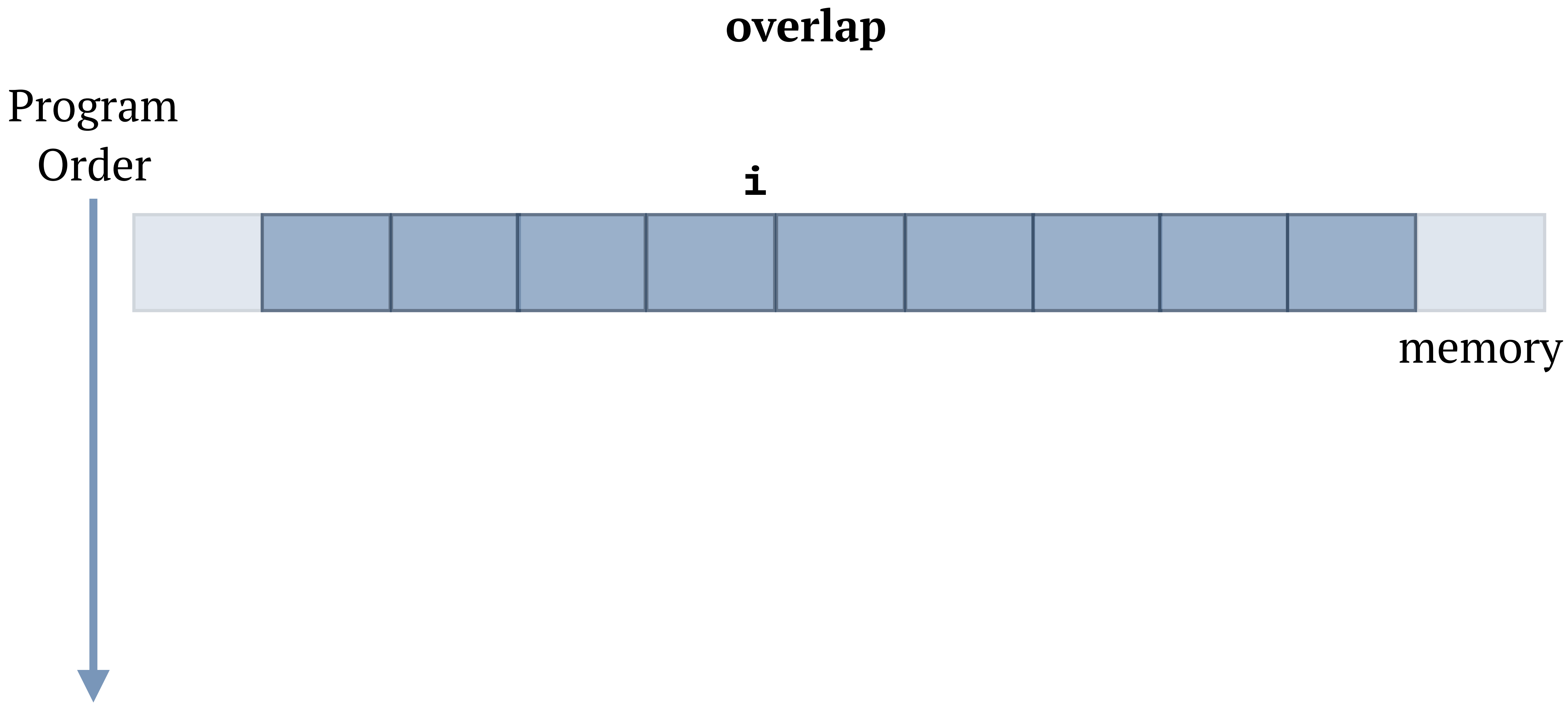
non-interference



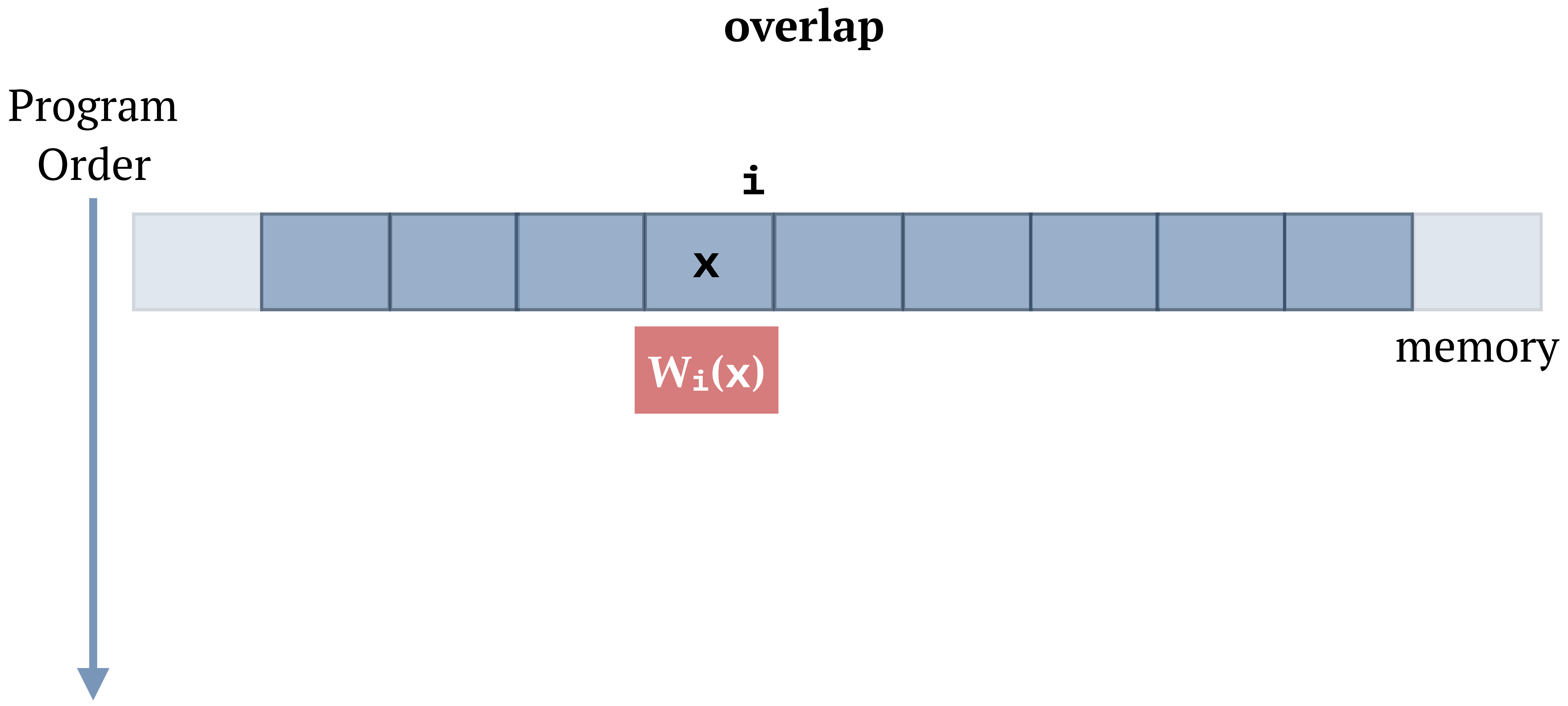
Read-over-Write Theorem: #1



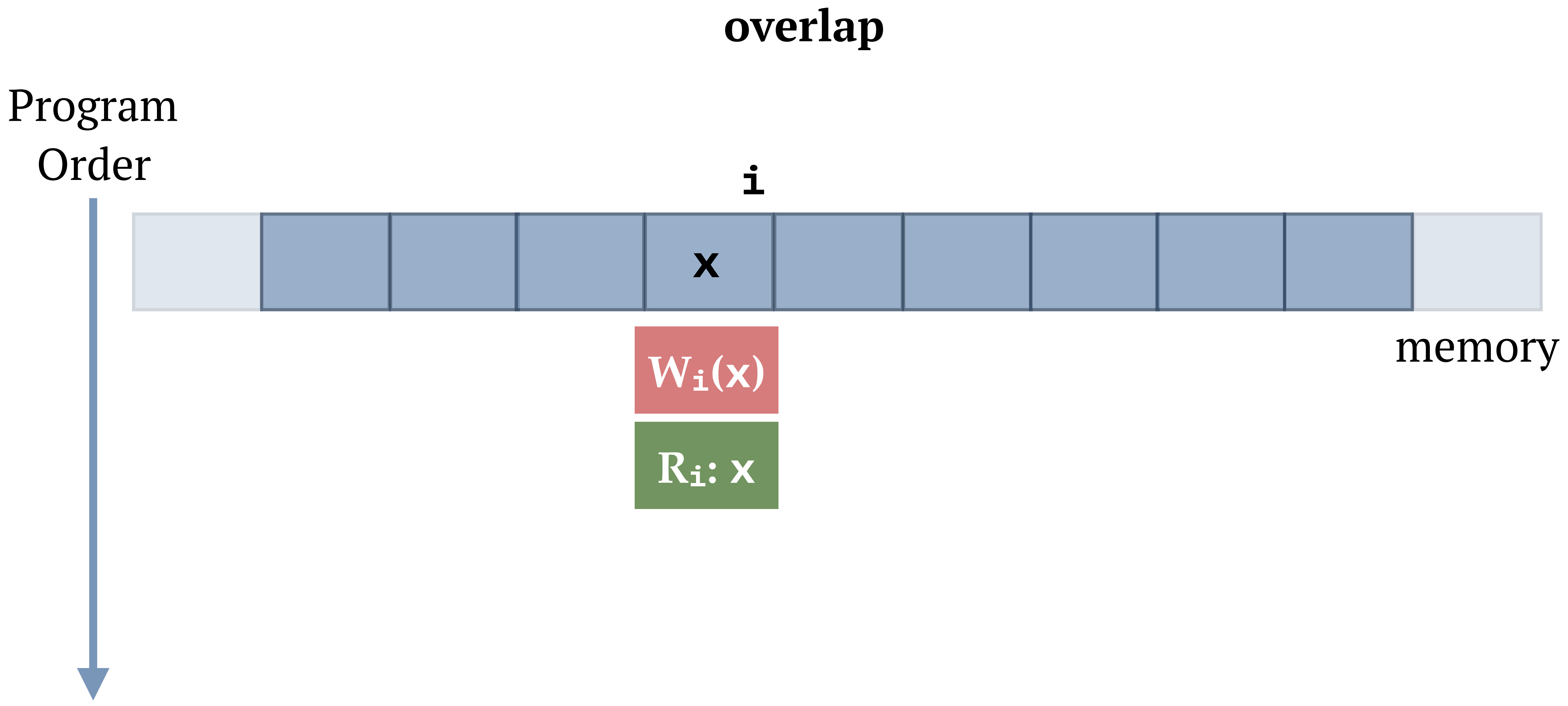
Read-over-Write Theorem: #2



Read-over-Write Theorem: #2

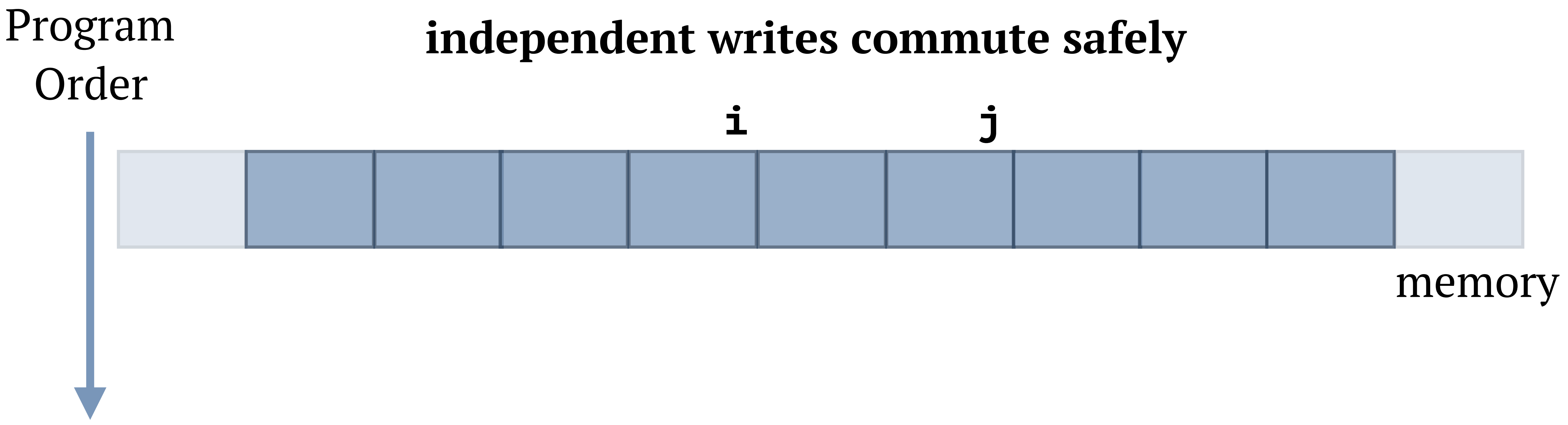


Read-over-Write Theorem: #2



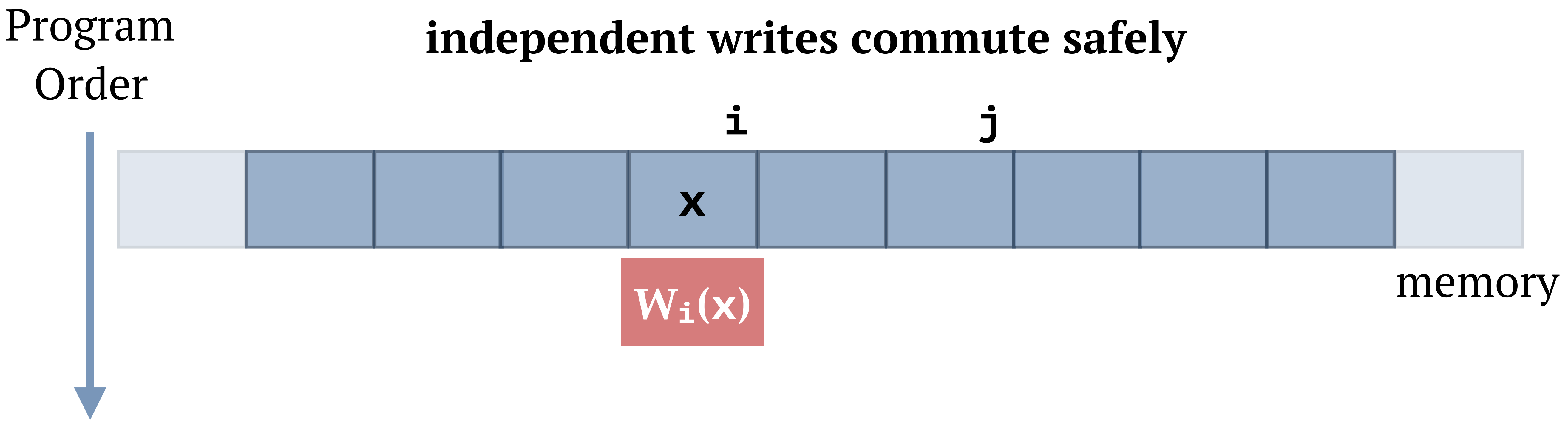
Write-over-Write Theorem: #1

independent writes commute safely



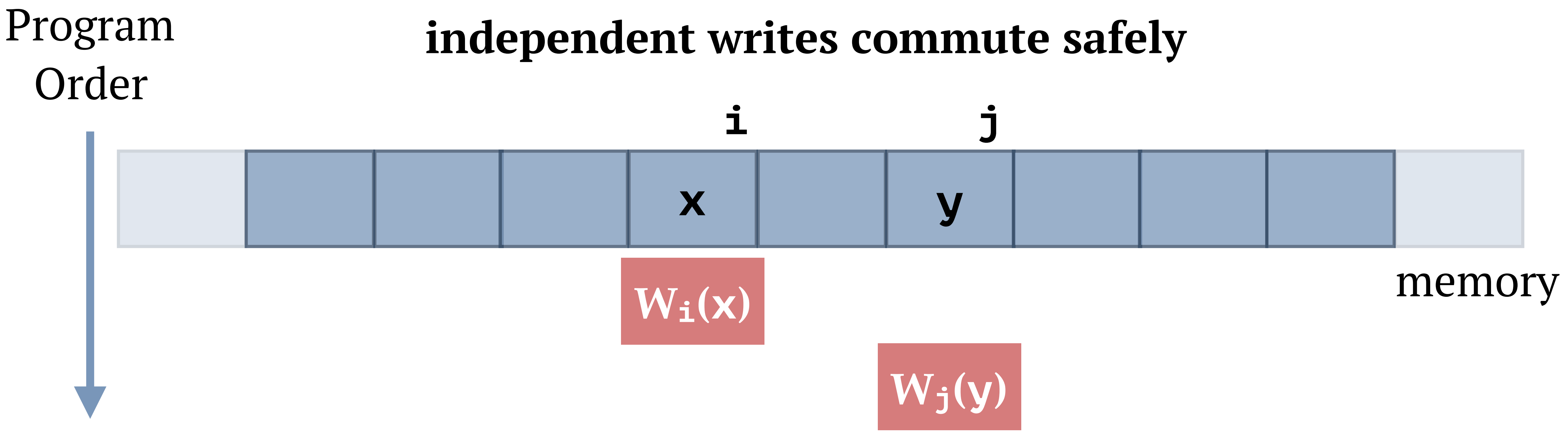
Write-over-Write Theorem: #1

independent writes commute safely



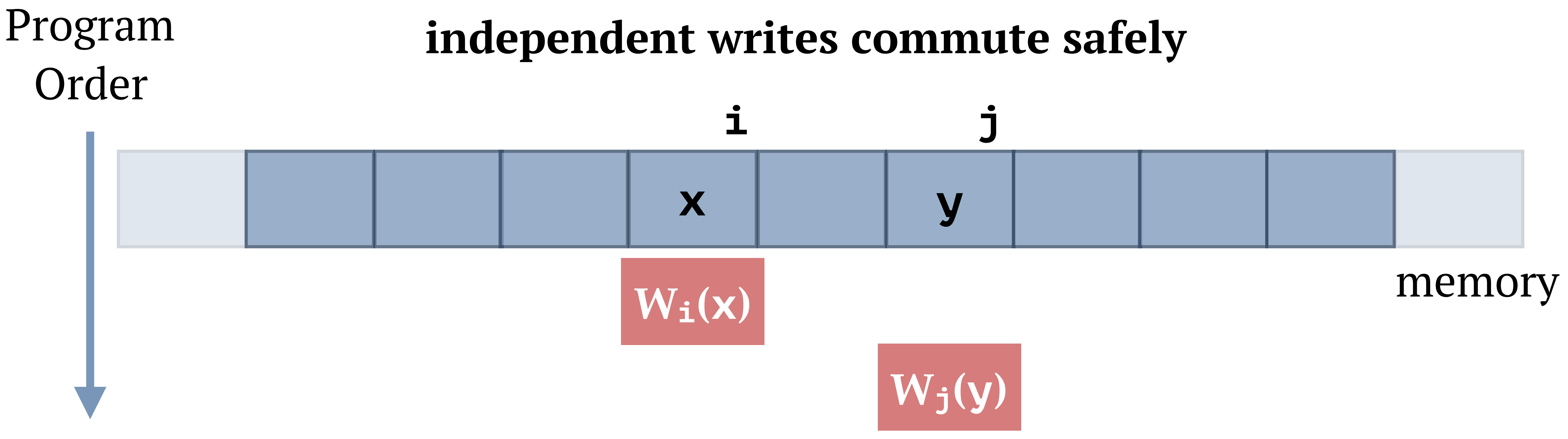
Write-over-Write Theorem: #1

independent writes commute safely

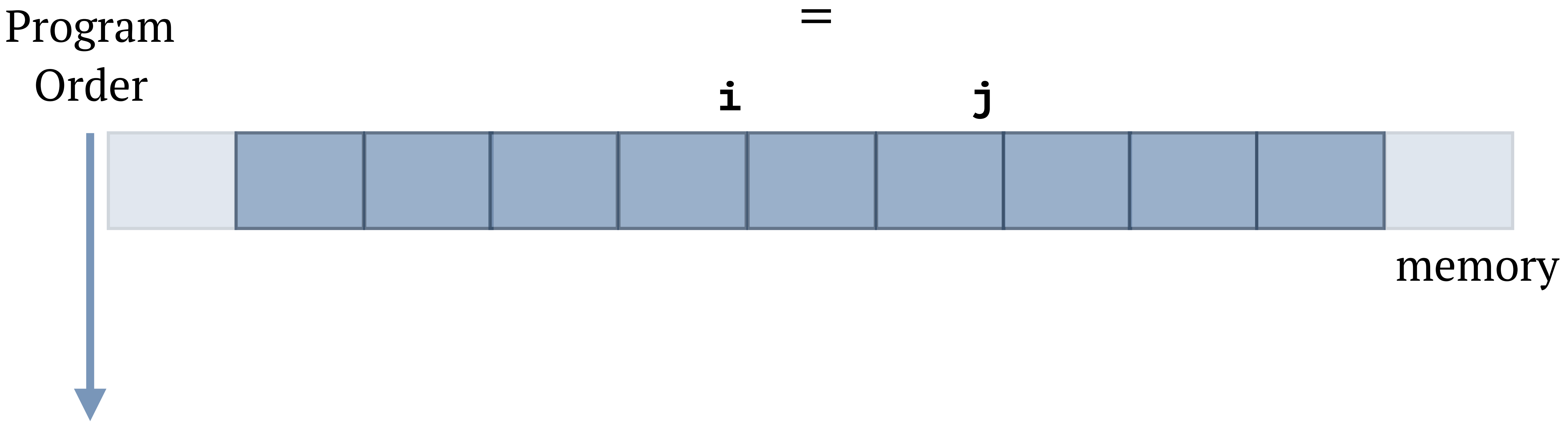


Write-over-Write Theorem: #1

independent writes commute safely

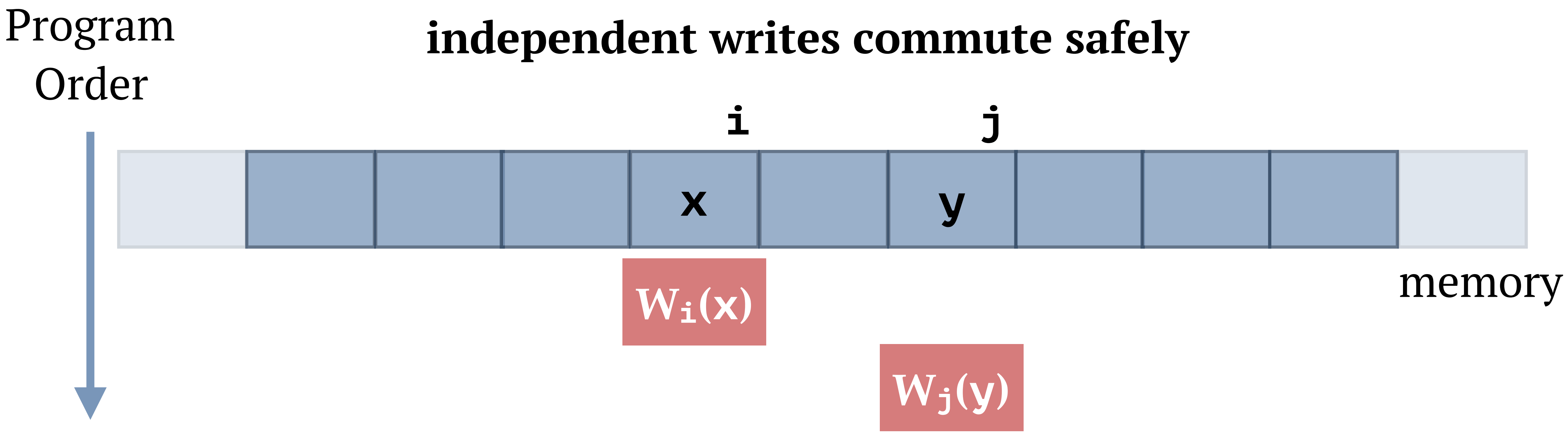


=

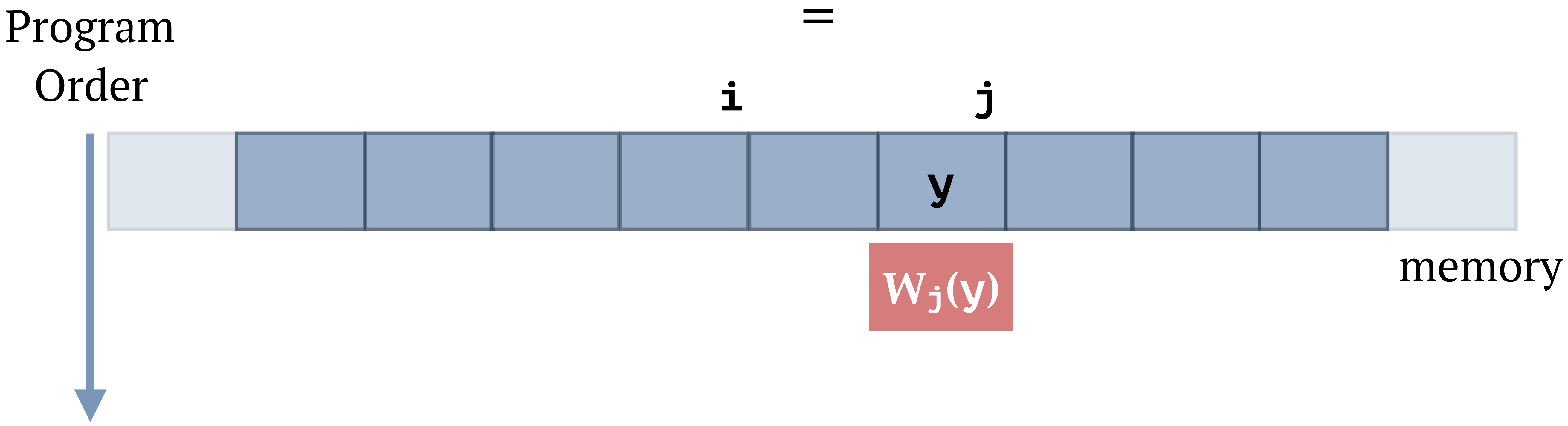


Write-over-Write Theorem: #1

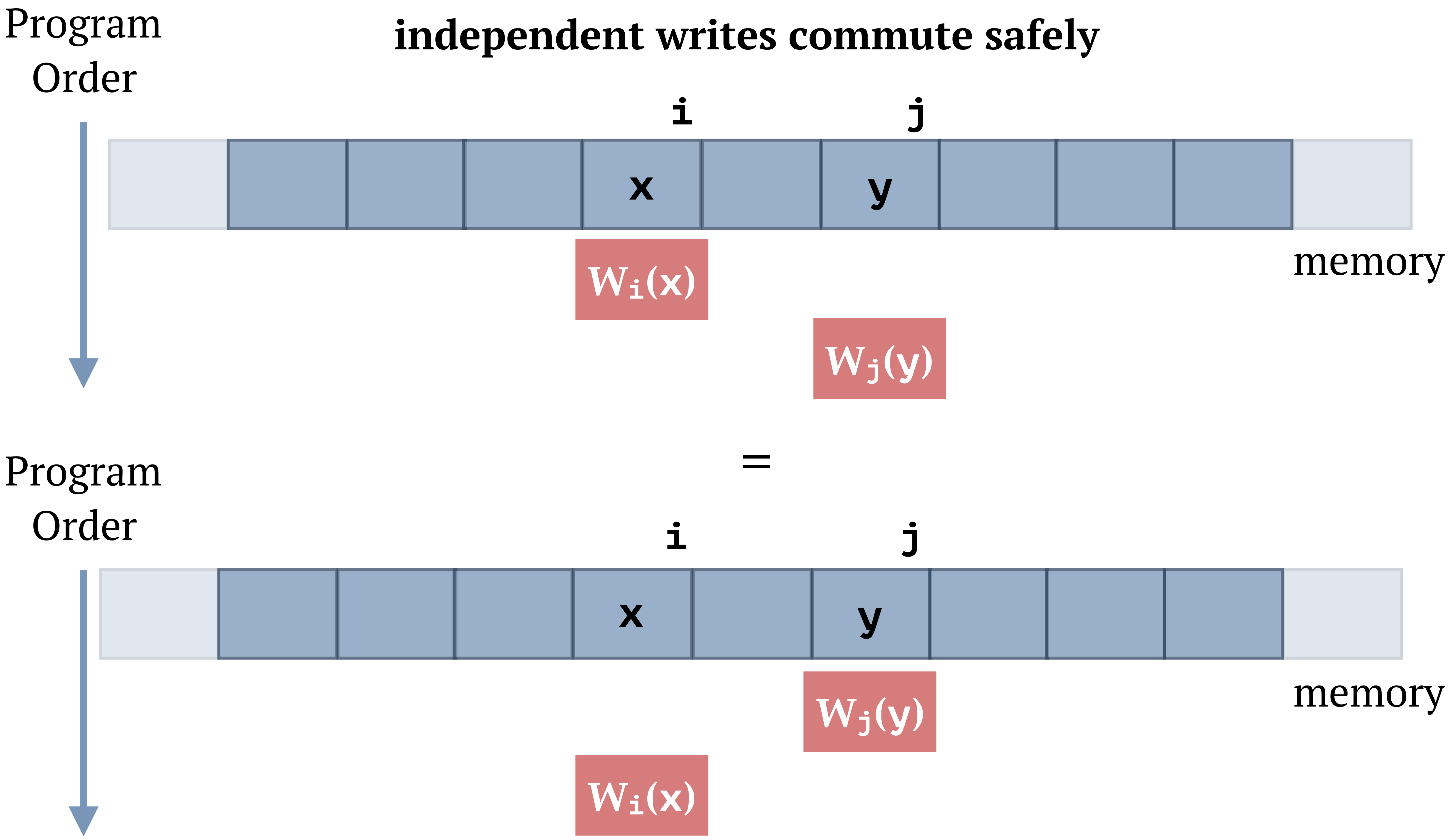
independent writes commute safely



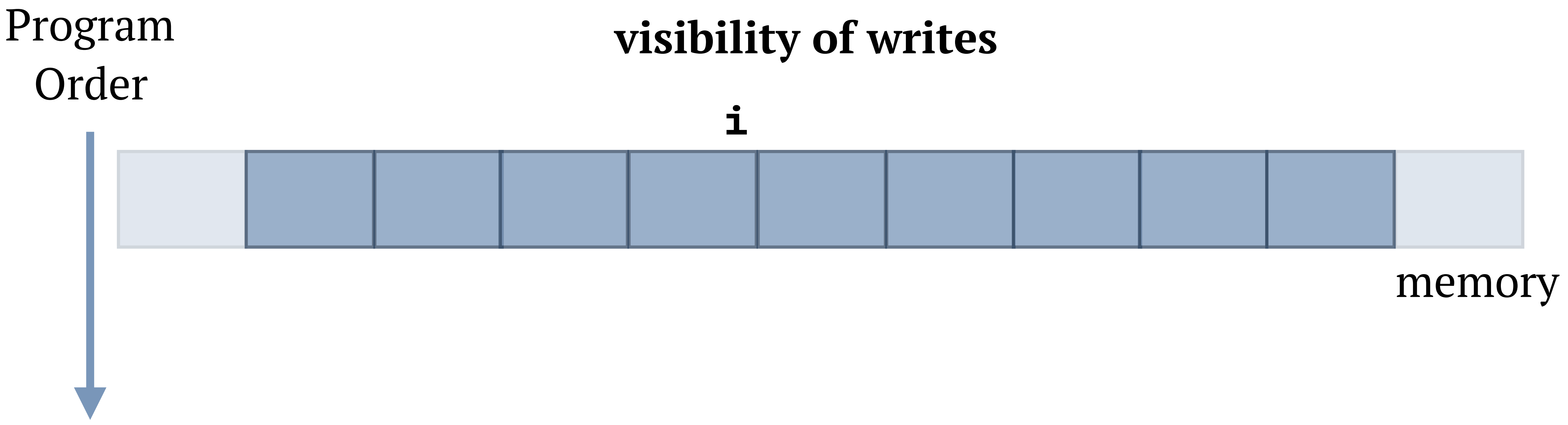
=



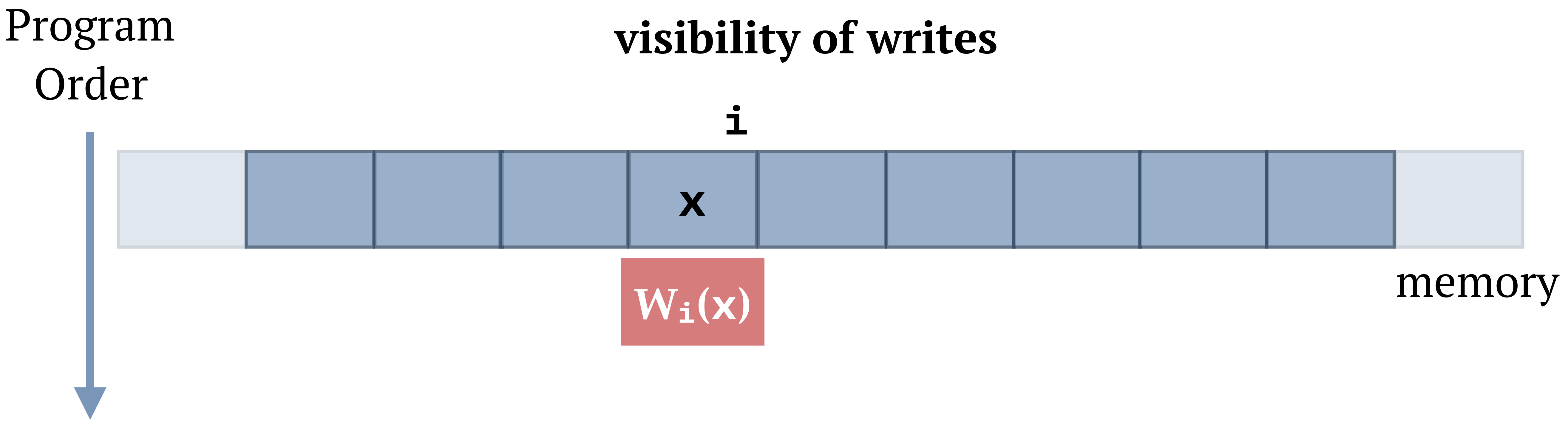
Write-over-Write Theorem: #1



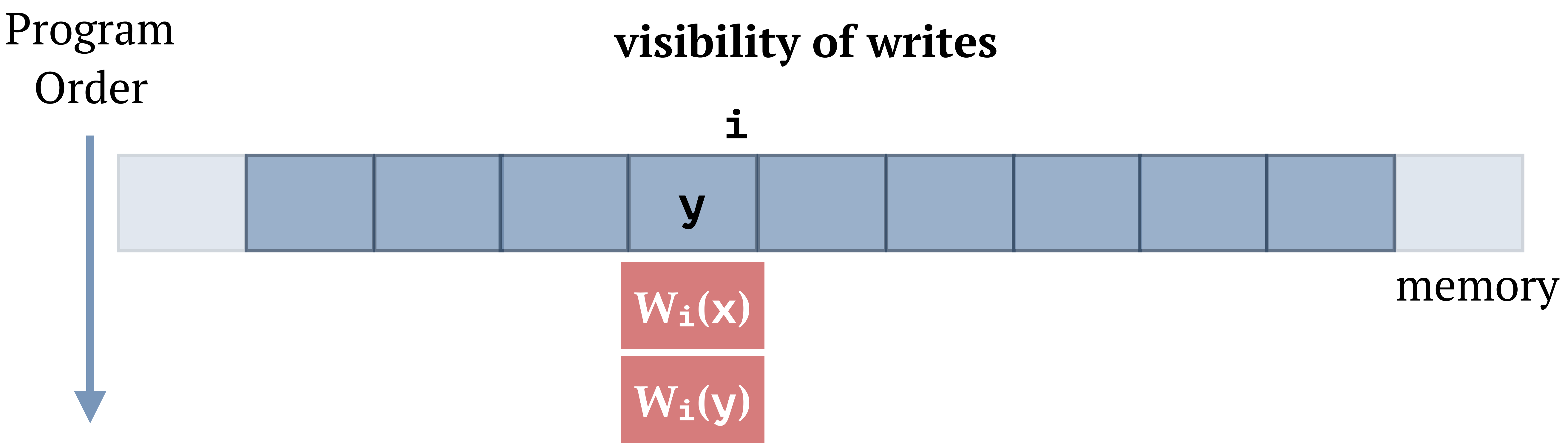
Write-over-Write Theorem: #2



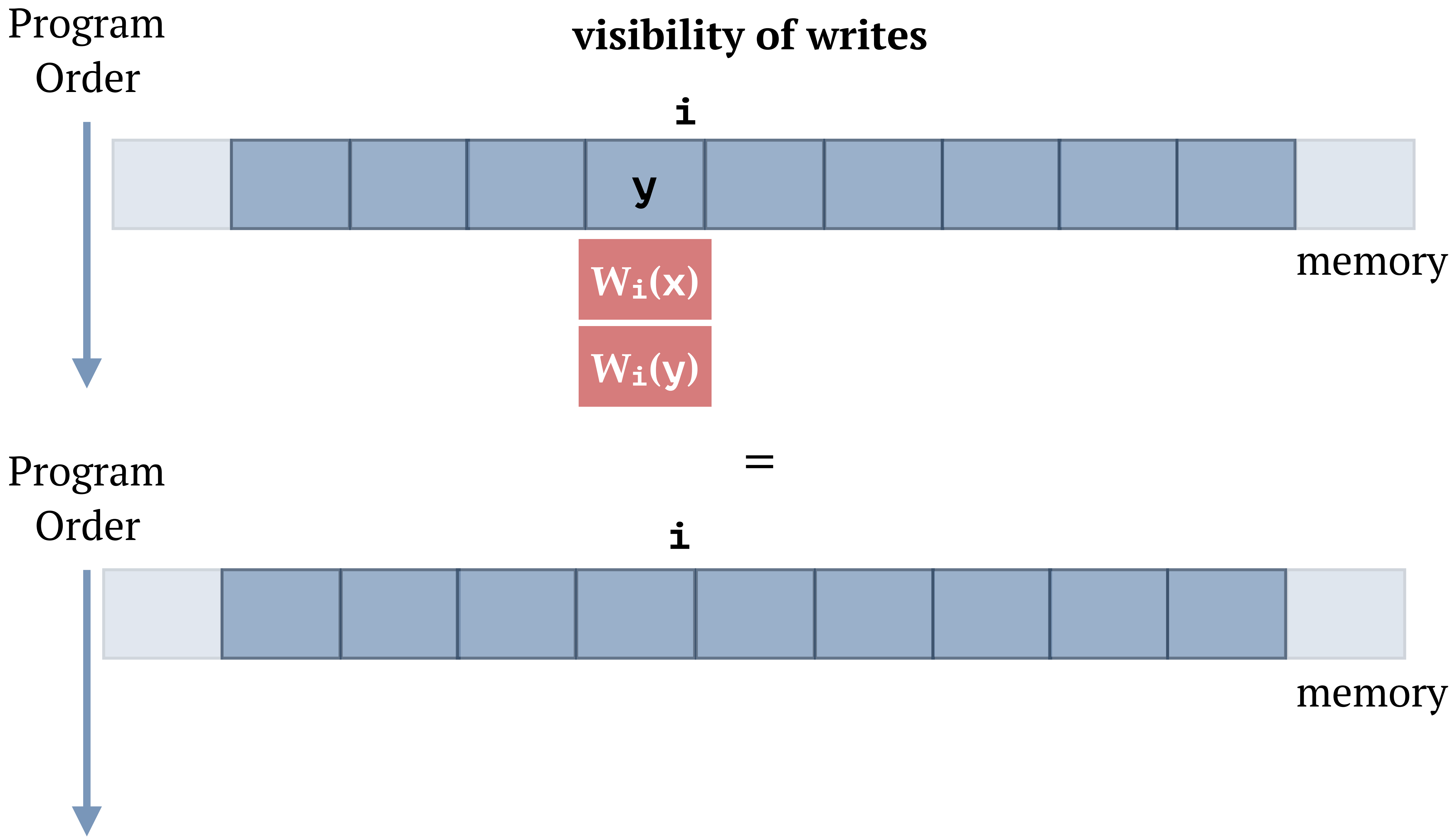
Write-over-Write Theorem: #2



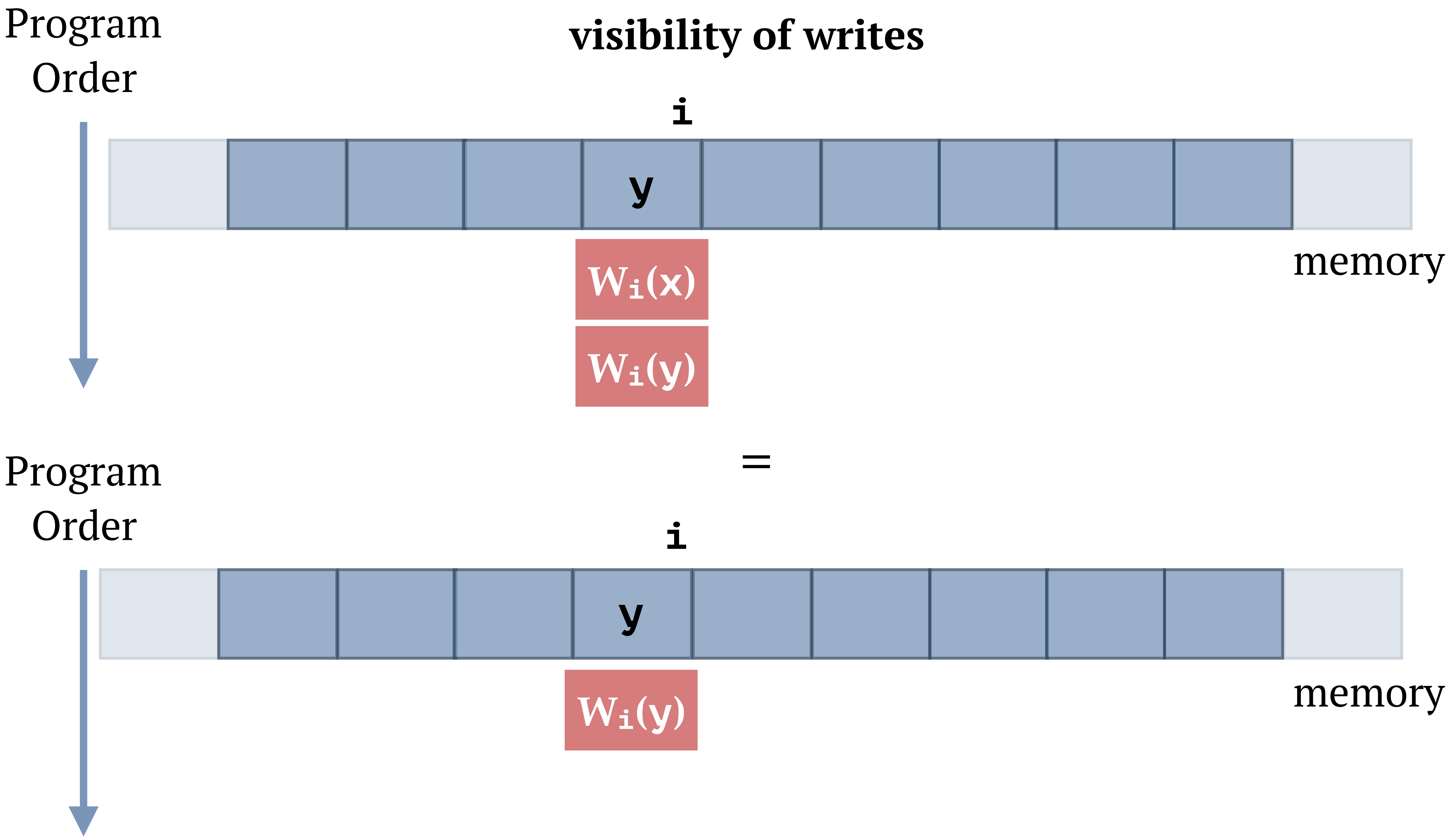
Write-over-Write Theorem: #2



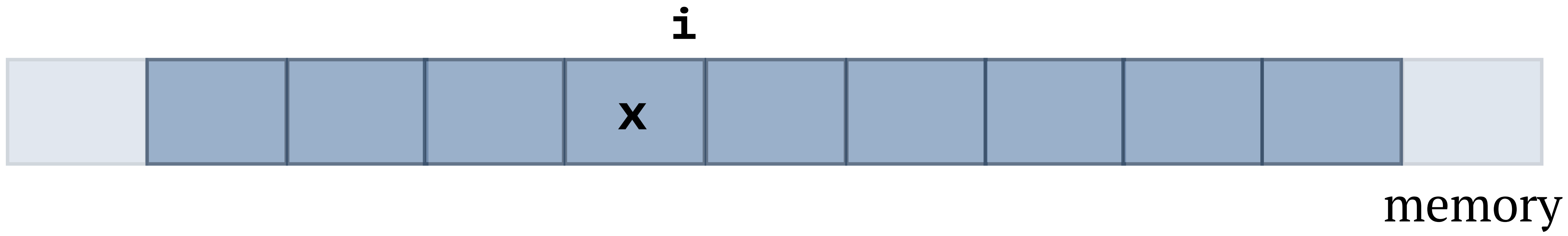
Write-over-Write Theorem: #2



Write-over-Write Theorem: #2



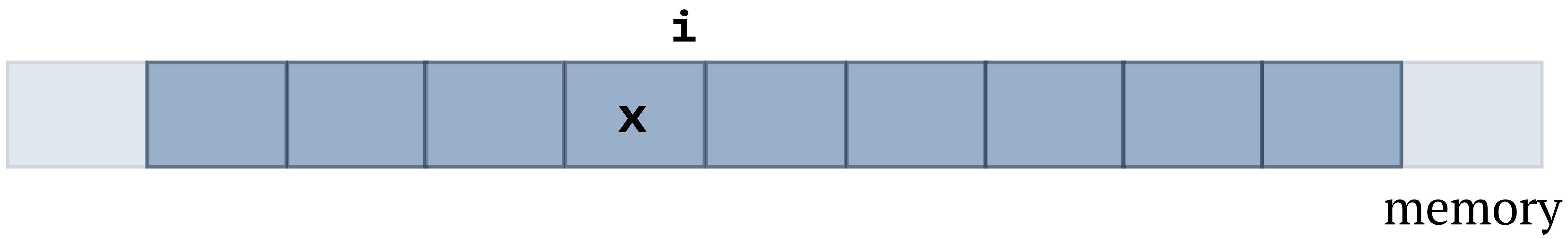
Preservation Theorems



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



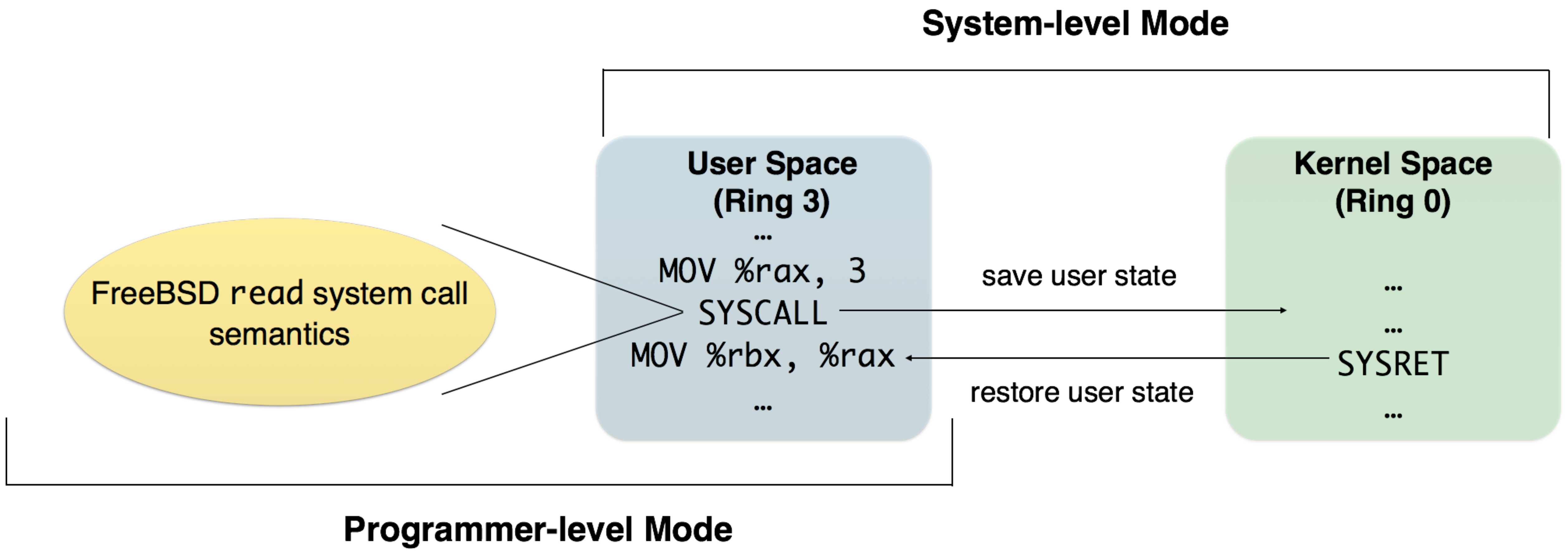
reading from a valid x86 state

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

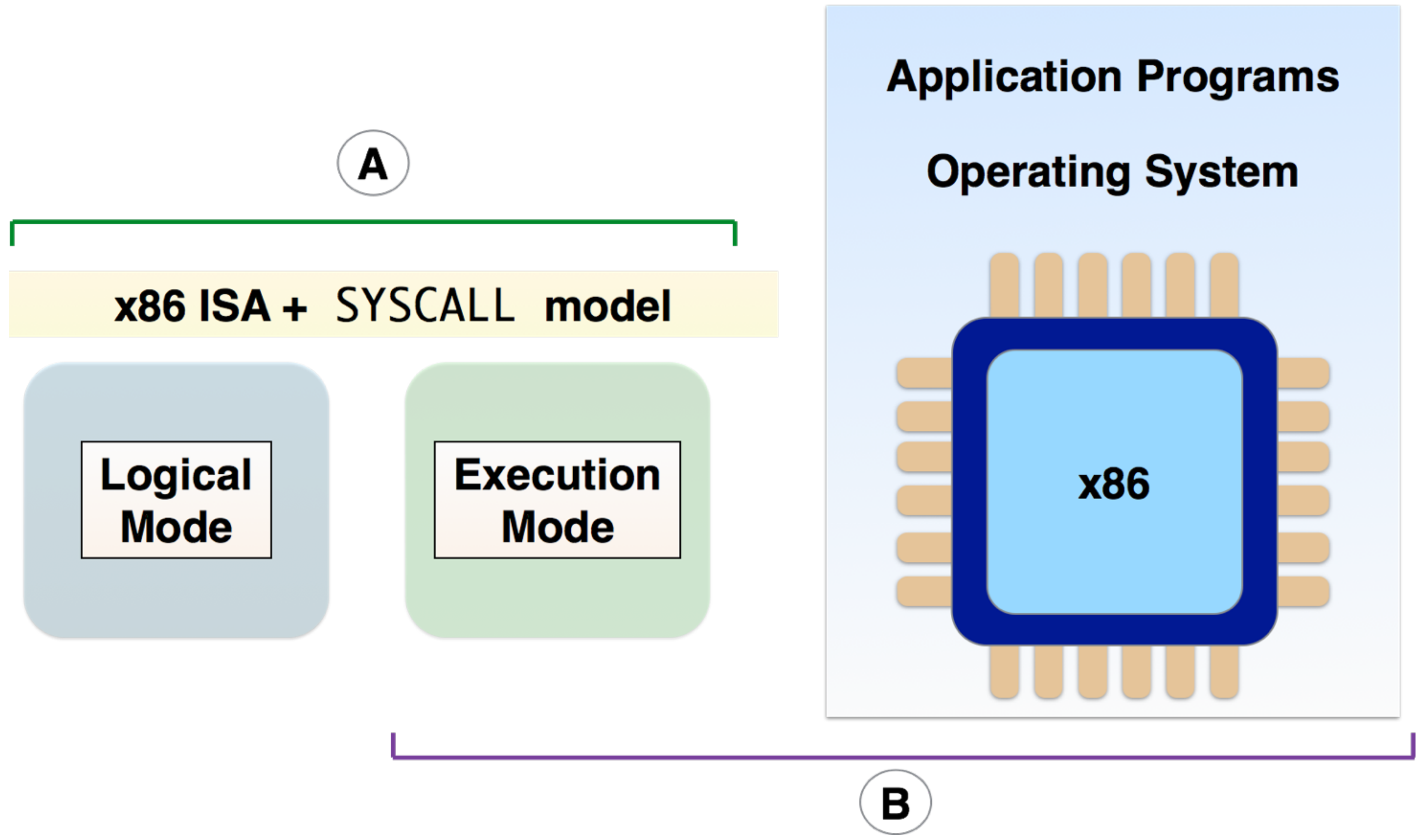
writing to a valid x86 state

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

Verification Effort vs. Verification Utility



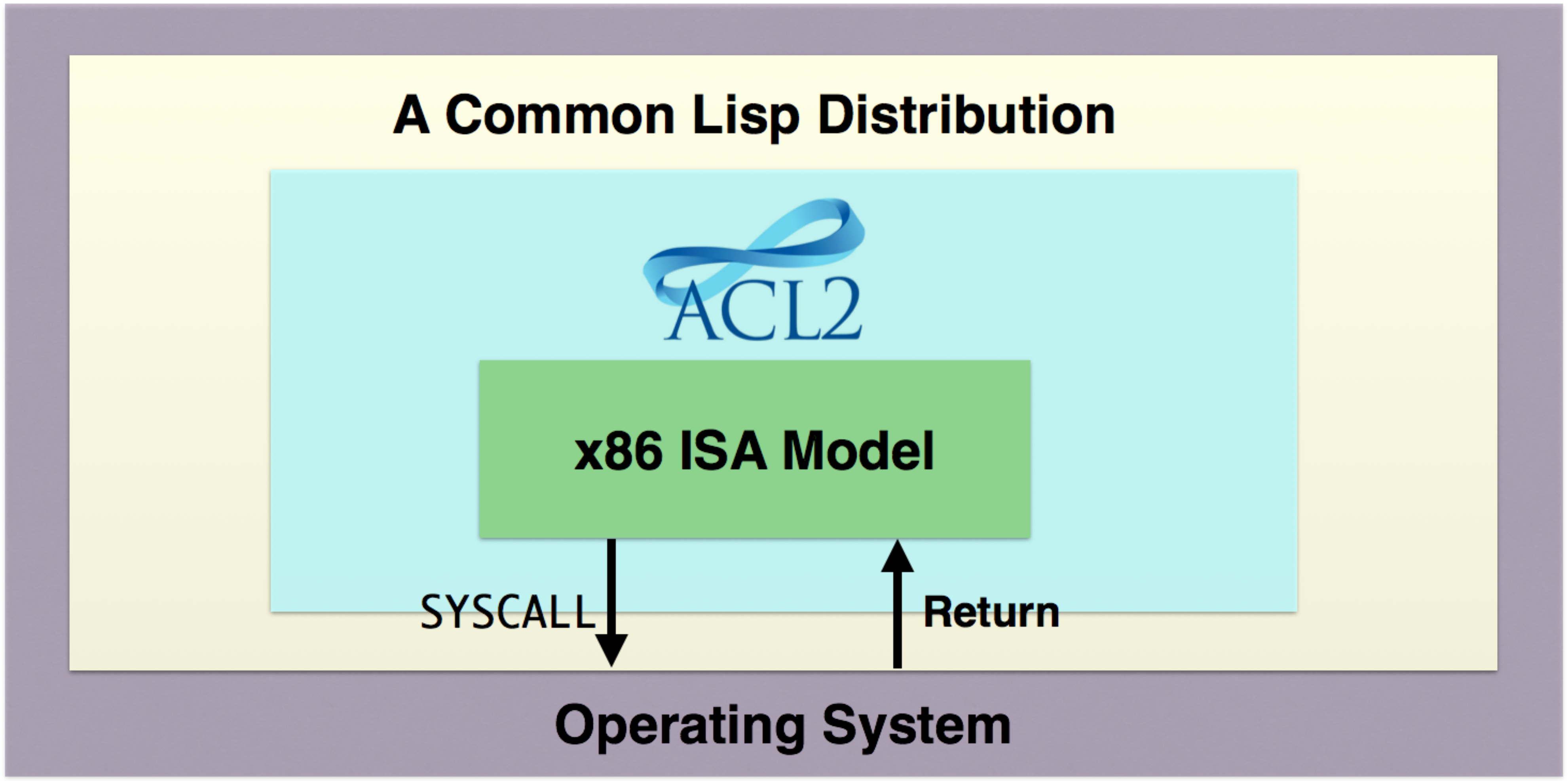
Programmer-level Mode: Model Validation



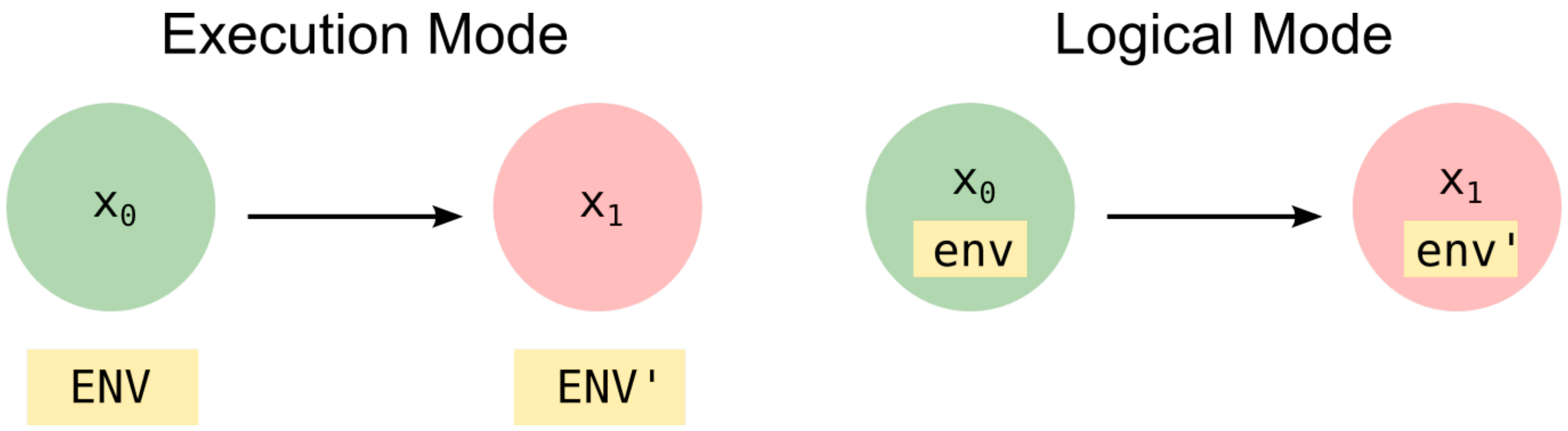
Task A: Validate the logical mode against the execution mode

Task B: Validate the execution mode against the processor + system call service provided by the OS

Programmer-level Mode: Execution Mode

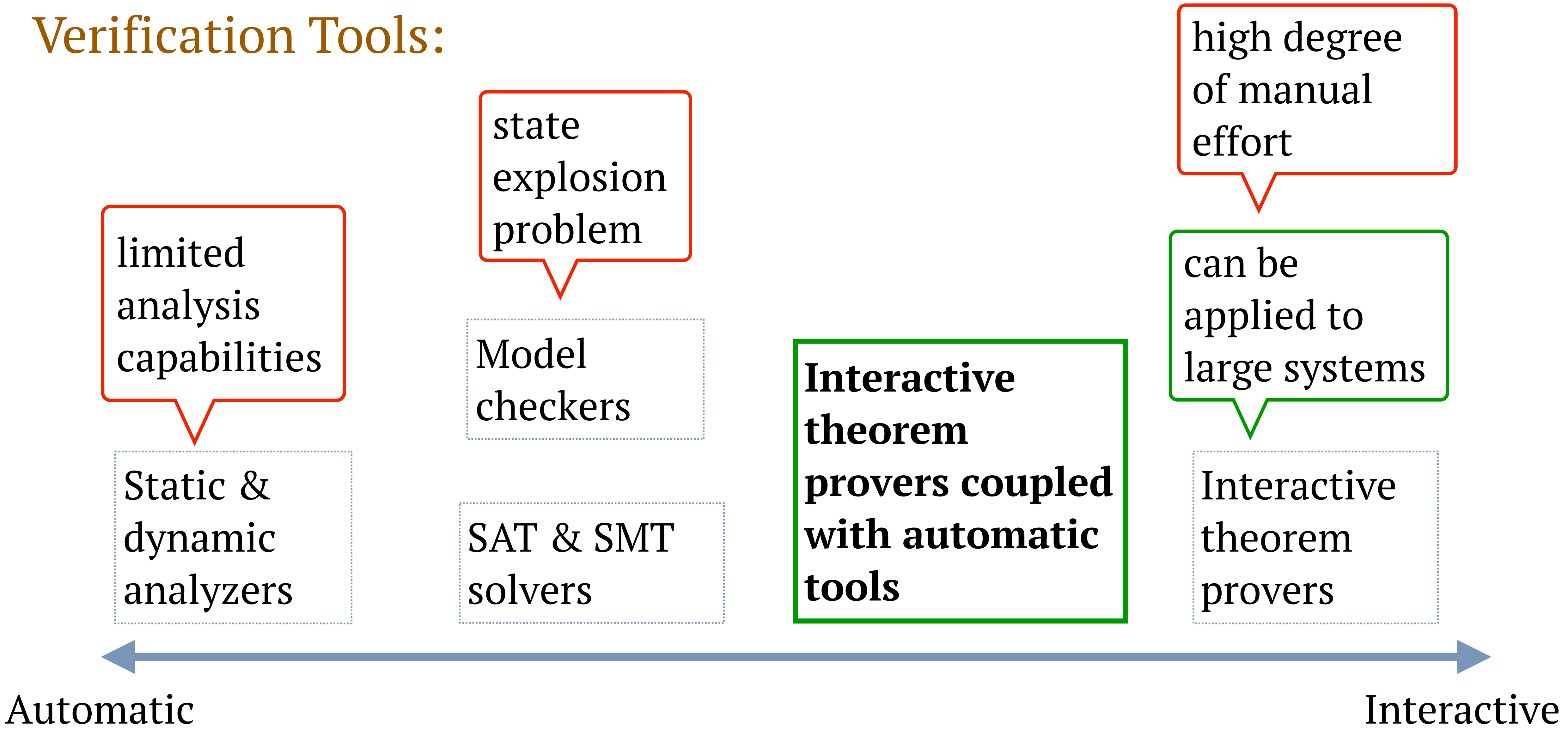


Programmer-level Mode: Execution and Reasoning



Verification Landscape

Verification Tools:


limited analysis capabilities
Static & dynamic analyzers
state explosion problem
Model checkers
SAT & SMT solvers
Interactive theorem provers coupled with automatic tools
high degree of manual effort
can be applied to large systems
Interactive theorem provers
Automatic
Interactive