

Systems and Networking – Unit I

B.Sc. in Applied Computer Science and Artificial Intelligence
2021-2022



SAPIENZA
UNIVERSITÀ DI ROMA

Gabriele Tolomei

Department of Computer Science

Sapienza Università di Roma

tolomei@di.uniroma1.it

Where Are We?

- So far, we have addressed:
 - Processes and Threads
 - CPU Scheduling
 - Synchronization and Deadlock

Where Are We?

- So far, we have addressed:
 - Processes and Threads
 - CPU Scheduling
 - Synchronization and Deadlock
- Today, we will be talking about:
 - Memory Management

Where Are We?

- So far, we have addressed:
 - Processes and Threads
 - CPU Scheduling
 - Synchronization and Deadlock
- Today, we will be talking about:
 - Memory Management
- ... Later on:
 - File Systems and I/O Storage
 - Advanced Topics (?)

Part IV: Memory Management

Goals of Memory Management

- Allocate memory resources among multiple competing processes
 - maximizing memory utilization and system throughput

Goals of Memory Management

- Allocate memory resources among multiple competing processes
 - maximizing memory utilization and system throughput
- Guarantee isolation between processes
 - addressability and protection

Goals of Memory Management

- Allocate memory resources among multiple competing processes
 - maximizing memory utilization and system throughput
- Guarantee isolation between processes
 - addressability and protection
- Provide a convenient abstraction to the programmer
 - illusion of unlimited amount of memory

From Source Code To Binary Executable

- User programs typically refer to instructions and data with **symbolic names**, such as "+", "&&", "x", "count", etc.

From Source Code To Binary Executable

- User programs typically refer to instructions and data with **symbolic names**, such as "+", "&&", "x", "count", etc.
- For a user program to be executed it first needs to be:
 - 1) Translated from source code to binary executable and stored on disk
 - 2) Brought from disk into main memory (RAM)

From Source Code To Binary Executable

- User programs typically refer to instructions and data with **symbolic names**, such as "+", "&&", "x", "count", etc.
- For a user program to be executed it first needs to be:
 - 1) Translated from source code to binary executable and stored on disk
 - 2) Brought from disk into main memory (RAM)
- Translation is done via **Compiler/Assembler/Linker**

From Source Code To Binary Executable

- User programs typically refer to instructions and data with **symbolic names**, such as "+", "&&", "x", "count", etc.
- For a user program to be executed it first needs to be:
 - 1) Translated from source code to binary executable and stored on disk
 - 2) Brought from disk into main memory (RAM)
- Translation is done via **Compiler/Assembler/Linker**
- Loading is done by the (OS) **Loader**

From Source Code To Binary Executable

- User programs typically refer to instructions and data with **symbolic names**, such as "+", "&&", "x", "count", etc.
- For a user program to be executed it first needs to be:
 - 1) Translated from source code to binary executable and stored on disk
 - 2) Brought from disk into main memory (RAM)
- Translation is done via **Compiler/Assembler/Linker**
- Loading is done by the (OS) **Loader**

NOTE: In case of purely-interpreted language implementations, translation from source code to executable is done "on-the-fly" by the loaded interpreter

CPU Must Refer To Memory Addresses

- CPU repeatedly **fetches**, **decodes**, and **executes** instructions from memory while executing a program

CPU Must Refer To Memory Addresses

- CPU repeatedly **fetches**, **decodes**, and **executes** instructions from memory while executing a program
- Most instructions involve retrieving data from memory or storing data in memory, or both

CPU Must Refer To Memory Addresses

- CPU repeatedly **fetches**, **decodes**, and **executes** instructions from memory while executing a program
- Most instructions involve retrieving data from memory or storing data in memory, or both
- Memory chips only respond to actual physical addresses

CPU Must Refer To Memory Addresses

- CPU repeatedly **fetches**, **decodes**, and **executes** instructions from memory while executing a program
- Most instructions involve retrieving data from memory or storing data in memory, or both
- Memory chips only respond to actual physical addresses
- It turns out that symbolic names used by user programs must be eventually bound to actual **physical memory addresses**

CPU Must Refer To Memory Addresses

- CPU repeatedly **fetches**, **decodes**, and **executes** instructions from memory while executing a program
- Most instructions involve retrieving data from memory or storing data in memory, or both
- Memory chips only respond to actual physical addresses
- It turns out that symbolic names used by user programs must be eventually bound to actual **physical memory addresses**

How?

Generating Memory Addresses: Example

```
void foo() {  
    int x = 42;  
    ...  
    x = x + 5;  
    ...  
}
```

Generating Memory Addresses: Example

```
void foo() {  
    int x = 42;  
    ...  
    x = x + 5;  
    ...  
}
```

x is a symbolic name

Generating Memory Addresses: Example

```
void foo() {  
    int x = 42;  
    ...  
    x = x + 5;  
    ...  
}
```

x is a symbolic name



```
128: MOV %R1, [%R2] // assuming R2 contains the  
                      // address of x in memory  
136: ADD 5, %R1  
144: MOV [%R2], %R1
```

Generating Memory Addresses: Example

```
void foo() {  
    int x = 42;  
    ...  
    x = x + 5;  
    ...  
}
```

x is a symbolic name



```
128: MOV %R1, [%R2] // assuming R2 contains the  
                        // address of x in memory  
136: ADD 5, %R1  
144: MOV [%R2], %R1
```

references to **logical addresses** containing
both **instructions** and **data**

Generating Memory Addresses: Example

```
void foo() {  
    int x = 42;  
    ...  
    x = x + 5;  
    ...  
}
```

x is a symbolic name



```
128: MOV %R1, [%R2] // assuming R2 contains the  
                        // address of x in memory  
136: ADD 5, %R1  
144: MOV [%R2], %R1
```

references to **logical addresses** containing
both **instructions** and **data**

1. Fetch instruction at address 128

Generating Memory Addresses: Example

```
void foo() {  
    int x = 42;  
    ...  
    x = x + 5;  
    ...  
}
```

x is a symbolic name



```
128: MOV %R1, [%R2] // assuming R2 contains the  
                        // address of x in memory  
136: ADD 5, %R1  
144: MOV [%R2], %R1
```

references to **logical addresses** containing
both **instructions** and **data**

1. Fetch instruction at address 128
2. Execute instruction: load from address [%R2] (e.g., 1234)

Generating Memory Addresses: Example

```
void foo() {  
    int x = 42;  
    ...  
    x = x + 5;  
    ...  
}
```

x is a symbolic name



```
128: MOV %R1, [%R2] // assuming R2 contains the  
                      // address of x in memory  
136: ADD 5, %R1  
144: MOV [%R2], %R1
```

references to **logical addresses** containing
both **instructions** and **data**

1. Fetch instruction at address 128
2. Execute instruction: load from address [%R2] (e.g., 1234)
3. Fetch instruction at address 136
4. Execute instruction: addition (no memory reference)
5. Fetch instruction at address 144
6. Execute instruction: store to address [%R2] (1234)

Symbolic Name vs. Logical vs. Physical Address

symbolic name: symbolic memory reference used by user programs

Symbolic Name vs. Logical vs. Physical Address

symbolic name: symbolic memory reference used by user programs



logical address: memory address generated by user programs via the CPU

Symbolic Name vs. Logical vs. Physical Address

symbolic name: symbolic memory reference used by user programs



logical address: memory address generated by user programs via the CPU



physical address: actual memory address which memory chip operates on

Symbolic Name vs. Logical vs. Physical Address

symbolic name: symbolic memory reference used by user programs



logical address: memory address generated by user programs via the CPU



physical address: actual memory address which memory chip operates on

address binding: mapping from logical to physical address

Symbolic Name vs. Logical vs. Physical Address

symbolic name: symbolic memory reference used by user programs



logical address: memory address generated by user programs via the CPU



physical address: actual memory address which memory chip operates on

address binding: mapping from logical to physical address

A green rectangular box with the text "Compile time" in white. A red arrow points from the box to the "address binding" text above it.

Compile time

Symbolic Name vs. Logical vs. Physical Address

symbolic name: symbolic memory reference used by user programs



logical address: memory address generated by user programs via the CPU



physical address: actual memory address which memory chip operates on

address binding: mapping from logical to physical address

Compile time

Load time

Symbolic Name vs. Logical vs. Physical Address

symbolic name: symbolic memory reference used by user programs



logical address: memory address generated by user programs via the CPU



physical address: actual memory address which memory chip operates on

address binding: mapping from logical to physical address

Compile time

Load time

Execution time

Address Binding: Compile Time

- If the starting physical location k where a program will reside in memory is known at compile time (e.g., $k = 0$)

Address Binding: Compile Time

- If the starting physical location k where a program will reside in memory is known at compile time (e.g., $k = 0$)
- The compiler generates so-called **absolute code**

Address Binding: Compile Time

- If the starting physical location k where a program will reside in memory is known at compile time (e.g., $k = 0$)
- The compiler generates so-called **absolute code**
- No intervention by the OS

Address Binding: Compile Time

- If the starting physical location k where a program will reside in memory is known at compile time (e.g., $k = 0$)
- The compiler generates so-called **absolute code**
- No intervention by the OS
- **physical address == logical address**

Address Binding: Compile Time

- If the starting physical location k where a program will reside in memory is known at compile time (e.g., $k = 0$)
- The compiler generates so-called **absolute code**
- No intervention by the OS
- **physical address == logical address**

What if the starting physical memory address k where the program is loaded changes into k' at some point later?

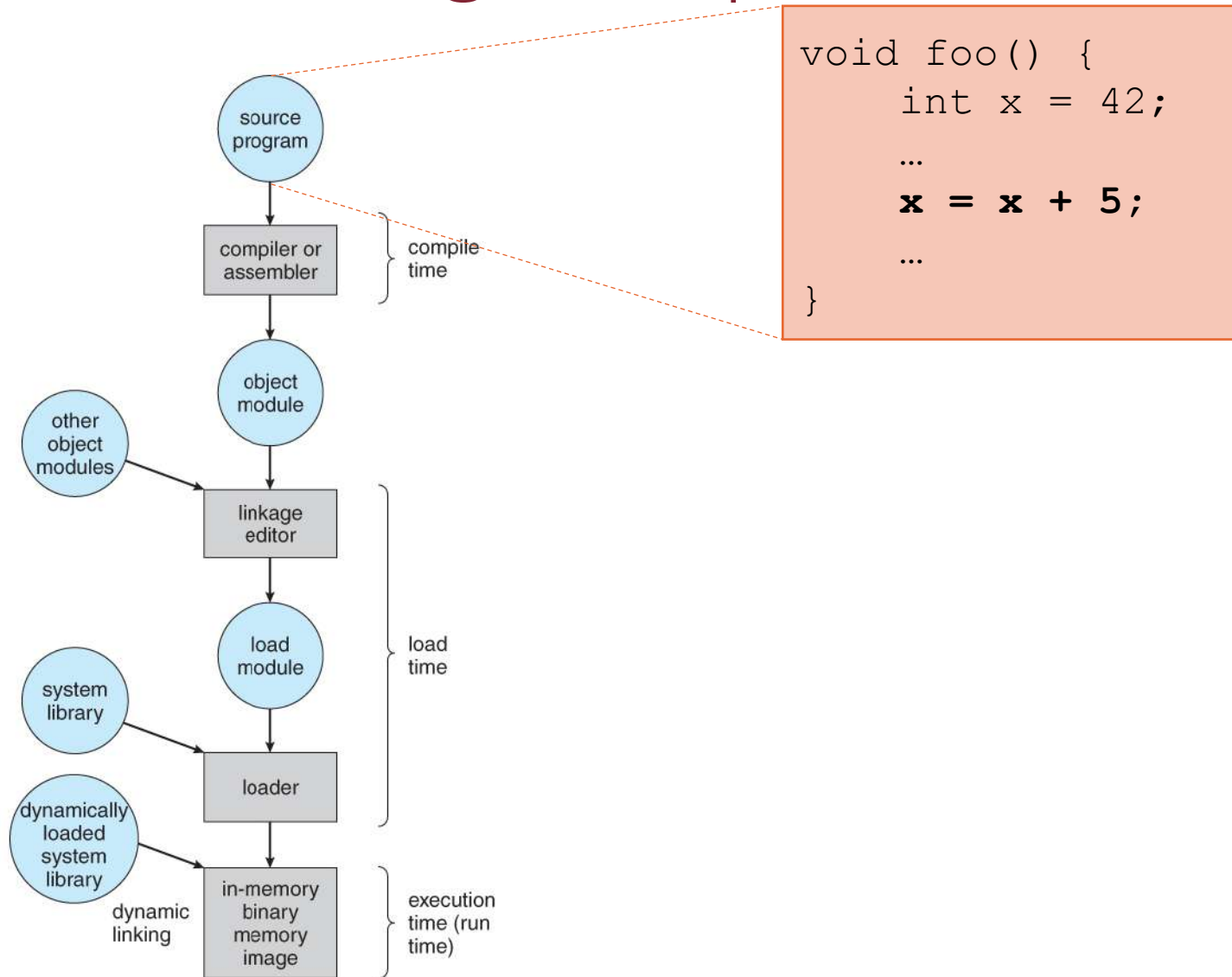
Address Binding: Compile Time

- If the starting physical location k where a program will reside in memory is known at compile time (e.g., $k = 0$)
- The compiler generates so-called **absolute code**
- No intervention by the OS
- **physical address == logical address**

What if the starting physical memory address k where the program is loaded changes into k' at some point later?

The program must be recompiled!

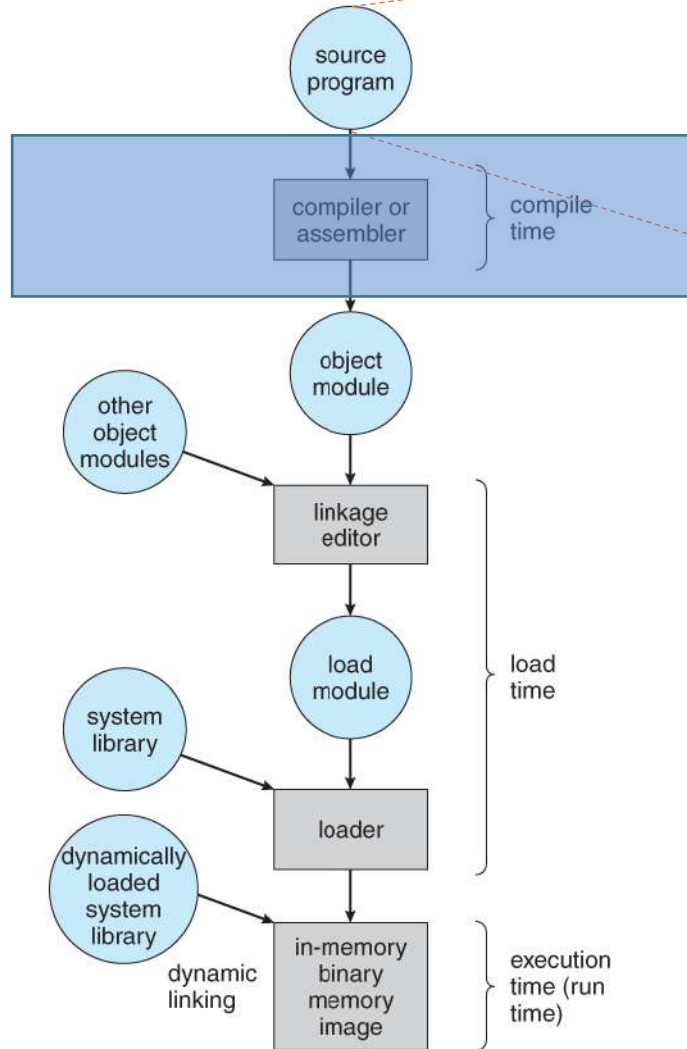
Address Binding: Compile Time



Address Binding: Compile Time

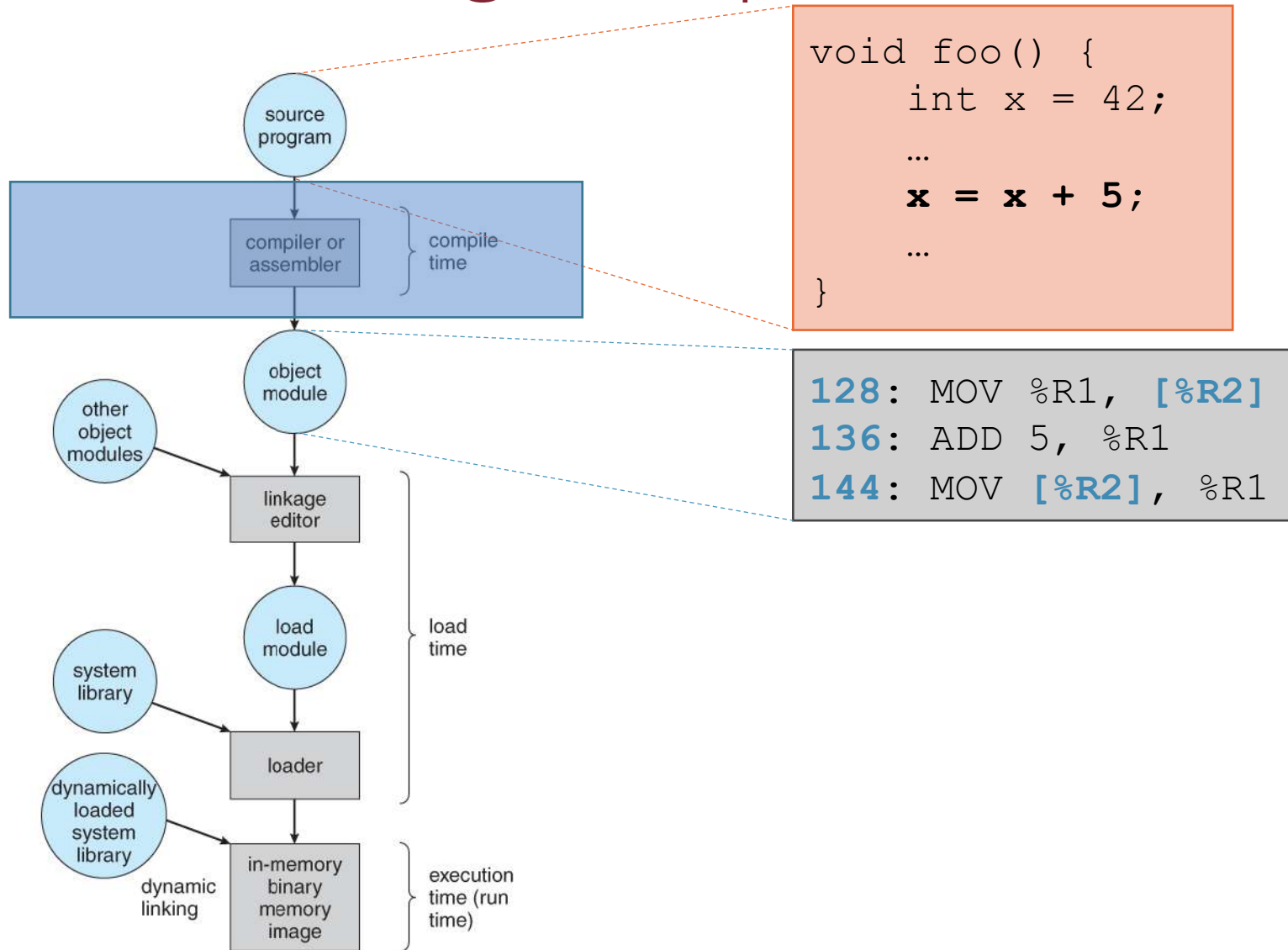
```
void foo() {  
    int x = 42;  
    ...  
    x = x + 5;  
    ...  
}
```

starting physical
address known
by the **compiler**
(e.g., $k = 0$)



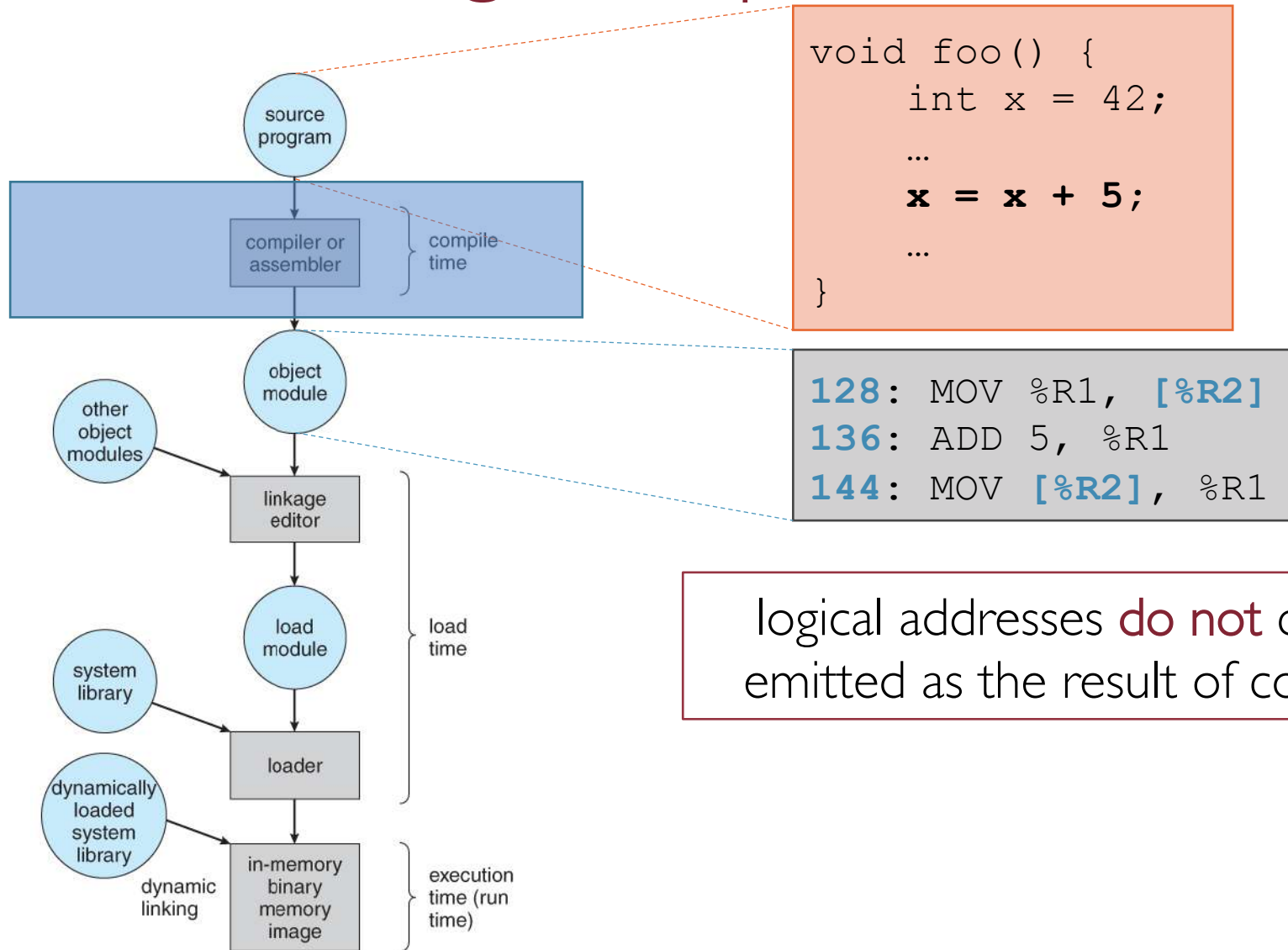
Address Binding: Compile Time

starting physical
address known
by the **compiler**
(e.g., $k = 0$)



Address Binding: Compile Time

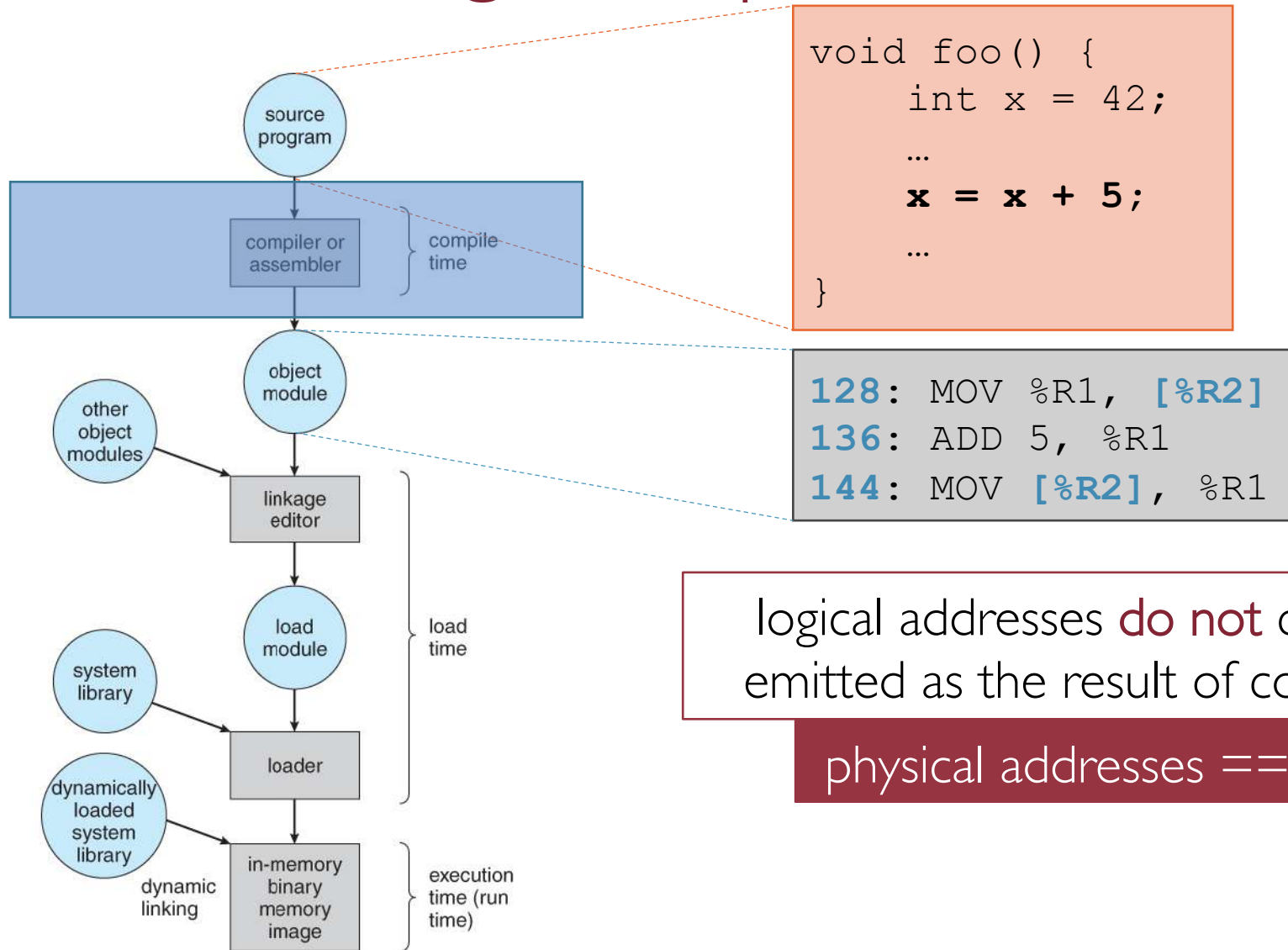
starting physical
address known
by the **compiler**
(e.g., $k = 0$)



logical addresses **do not** change after they are emitted as the result of compilation/assembling

Address Binding: Compile Time

starting physical
address known
by the **compiler**
(e.g., $k = 0$)



logical addresses **do not** change after they are emitted as the result of compilation/assembling

physical addresses == logical addresses

Address Binding: Load Time

- If the starting physical location k where a program will reside in memory is not known at compile time

Address Binding: Load Time

- If the starting physical location k where a program will reside in memory is not known at compile time
- The compiler generates so-called (**statically**) **relocatable code**, which references relative addresses to k

Address Binding: Load Time

- If the starting physical location k where a program will reside in memory is not known at compile time
- The compiler generates so-called (**statically**) **relocatable code**, which references relative addresses to k
- The OS loader determines each process' starting physical location k

Address Binding: Load Time

- If the starting physical location k where a program will reside in memory is not known at compile time
- The compiler generates so-called (**statically**) **relocatable code**, which references relative addresses to k
- The OS loader determines each process' starting physical location k
- **physical address == logical address**

Address Binding: Load Time

- If the starting physical location k where a program will reside in memory is not known at compile time
- The compiler generates so-called (**statically**) **relocatable code**, which references relative addresses to k
- The OS loader determines each process' starting physical location k
- **physical address == logical address**

What if the OS decides to use a different starting physical memory address k '?

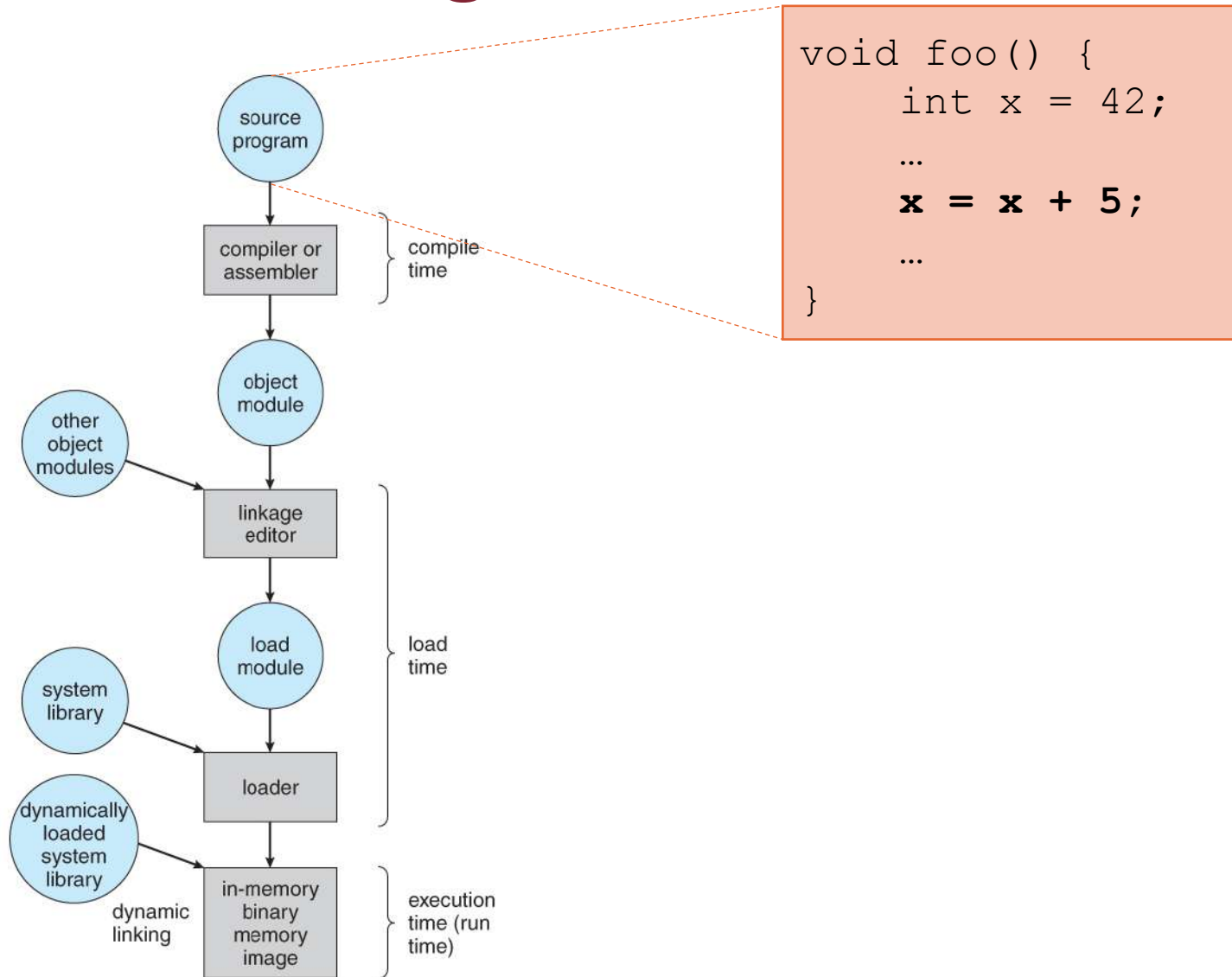
Address Binding: Load Time

- If the starting physical location k where a program will reside in memory is not known at compile time
- The compiler generates so-called (**statically**) **relocatable code**, which references relative addresses to k
- The OS loader determines each process' starting physical location k
- **physical address == logical address**

What if the OS decides to use a different starting physical memory address k '?

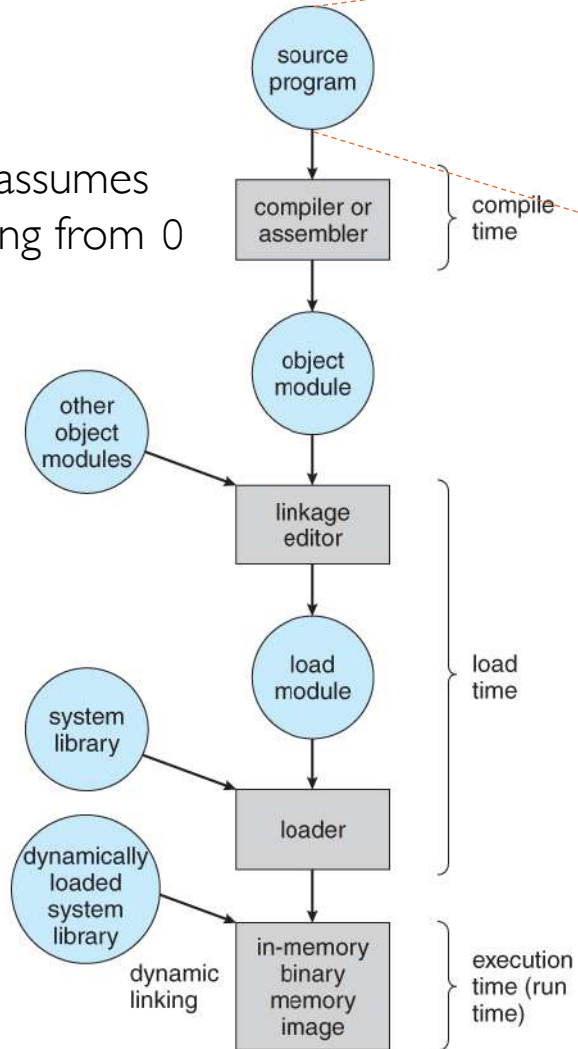
The program must be reloaded but not recompiled!

Address Binding: Load Time



Address Binding: Load Time

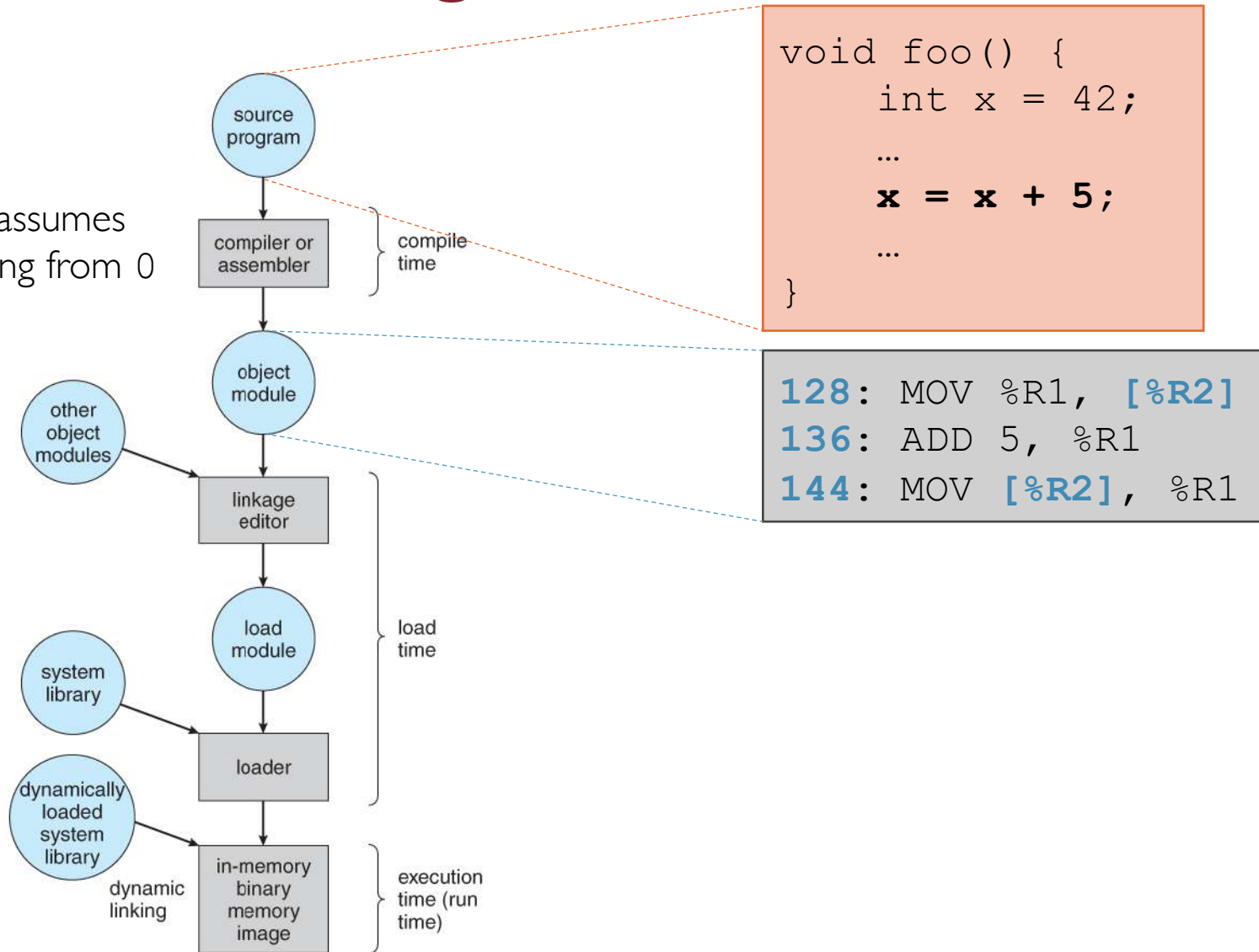
compiler still assumes
addresses starting from 0



```
void foo() {  
    int x = 42;  
    ...  
    x = x + 5;  
    ...  
}
```

Address Binding: Load Time

compiler still assumes
addresses starting from 0



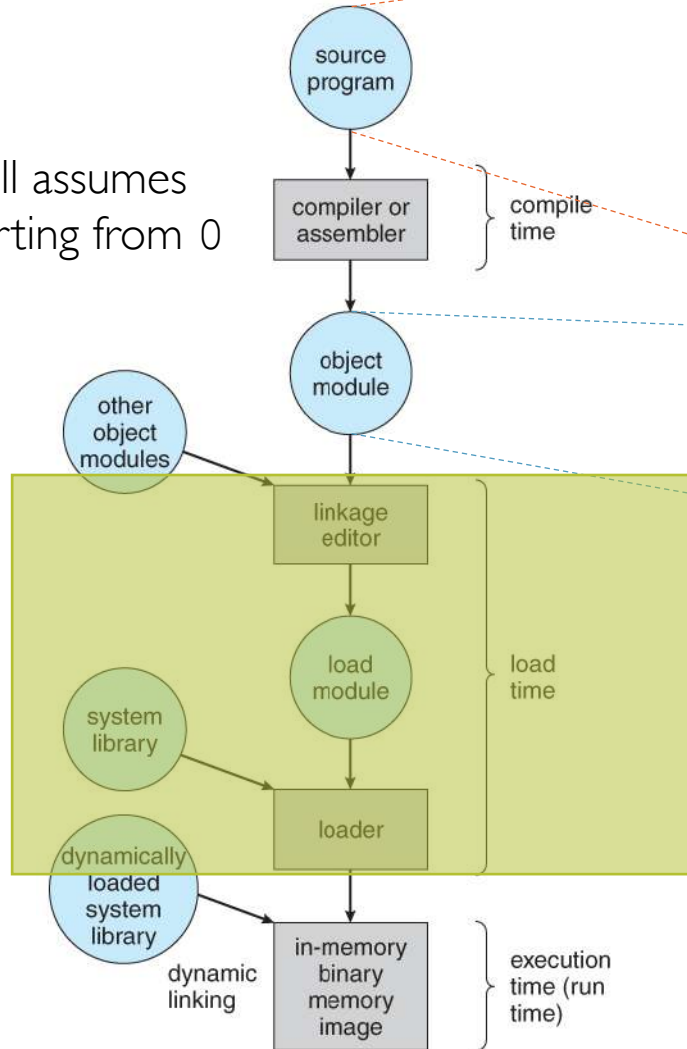
Address Binding: Load Time

compiler still assumes
addresses starting from 0

```
void foo() {  
    int x = 42;  
    ...  
    x = x + 5;  
    ...  
}
```

```
128: MOV %R1, [%R2]  
136: ADD 5, %R1  
144: MOV [%R2], %R1
```

starting physical
address known
by the **loader**
(e.g., k = 100)



Address Binding: Load Time

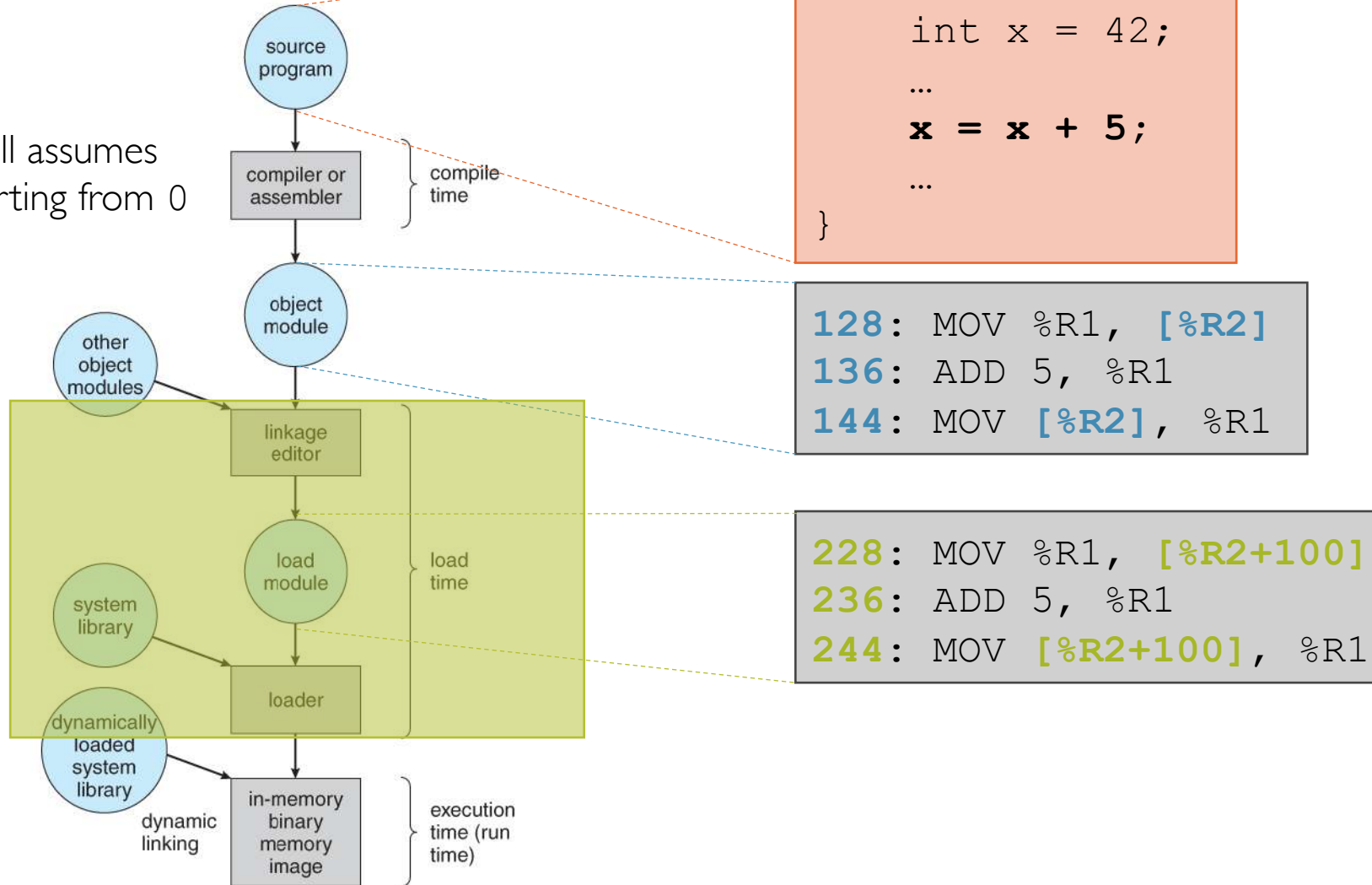
compiler still assumes
addresses starting from 0

```
void foo() {  
    int x = 42;  
    ...  
    x = x + 5;  
    ...  
}
```

```
128: MOV %R1, [%R2]  
136: ADD 5, %R1  
144: MOV [%R2], %R1
```

```
228: MOV %R1, [%R2+100]  
236: ADD 5, %R1  
244: MOV [%R2+100], %R1
```

starting physical
address known
by the **loader**
(e.g., k = 100)



Address Binding: Load Time

compiler still assumes
addresses starting from 0

```
void foo() {  
    int x = 42;  
    ...  
    x = x + 5;  
    ...  
}
```

compile
time

source
program

compiler or
assembler

object
module

other
object
modules

linkage
editor

load
module

loader

system
library

dynamically
loaded
system
library

dynamic
linking

in-memory
binary
memory
image

execution
time (run
time)

load
time

```
128: MOV %R1, [%R2]  
136: ADD 5, %R1  
144: MOV [%R2], %R1
```

```
228: MOV %R1, [%R2+100]  
236: ADD 5, %R1  
244: MOV [%R2+100], %R1
```

The OS loader updates logical addresses
before loading the executable

starting physical
address known
by the **loader**
(e.g., k = 100)

Address Binding: Load Time

compiler still assumes
addresses starting from 0

```
void foo() {  
    int x = 42;  
    ...  
    x = x + 5;  
    ...  
}
```

compile
time

source
program

compiler or
assembler

object
module

other
object
modules

linkage
editor

load
module

loader

system
library

dynamically
loaded
system
library

dynamic
linking

in-memory
binary
memory
image

execution
time (run
time)

load
time

```
128: MOV %R1, [%R2]  
136: ADD 5, %R1  
144: MOV [%R2], %R1
```

```
228: MOV %R1, [%R2+100]  
236: ADD 5, %R1  
244: MOV [%R2+100], %R1
```

The OS loader updates logical addresses
before loading the executable

physical addresses == logical addresses

starting physical
address known
by the **loader**
(e.g., k = 100)

Address Binding: Execution Time

- If the program can be moved around in main memory during its execution

Address Binding: Execution Time

- If the program can be moved around in main memory during its execution
- The compiler generates so-called so-called (**dynamically**) **relocatable code** or **virtual addresses**

Address Binding: Execution Time

- If the program can be moved around in main memory during its execution
- The compiler generates so-called so-called (**dynamically**) **relocatable code** or **virtual addresses**
- The OS maps virtual addresses to physical memory locations using special HW support (**MMU**)

Address Binding: Execution Time

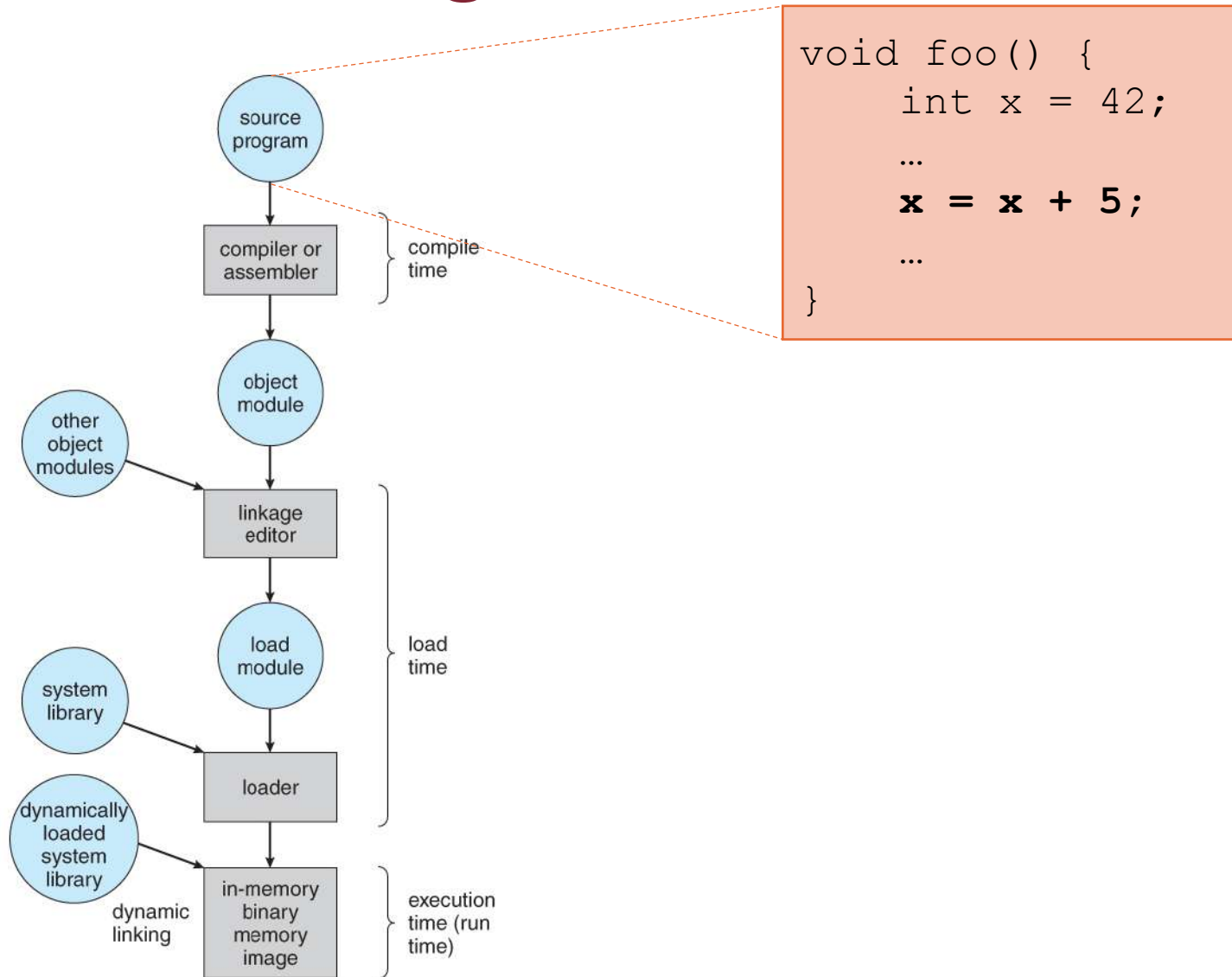
- If the program can be moved around in main memory during its execution
- The compiler generates so-called so-called (**dynamically**) **relocatable code** or **virtual addresses**
- The OS maps virtual addresses to physical memory locations using special HW support (**MMU**)
- **physical address \neq logical/virtual address**

Address Binding: Execution Time

- If the program can be moved around in main memory during its execution
- The compiler generates so-called so-called (**dynamically**) **relocatable code** or **virtual addresses**
- The OS maps virtual addresses to physical memory locations using special HW support (**MMU**)
- **physical address \neq logical/virtual address**

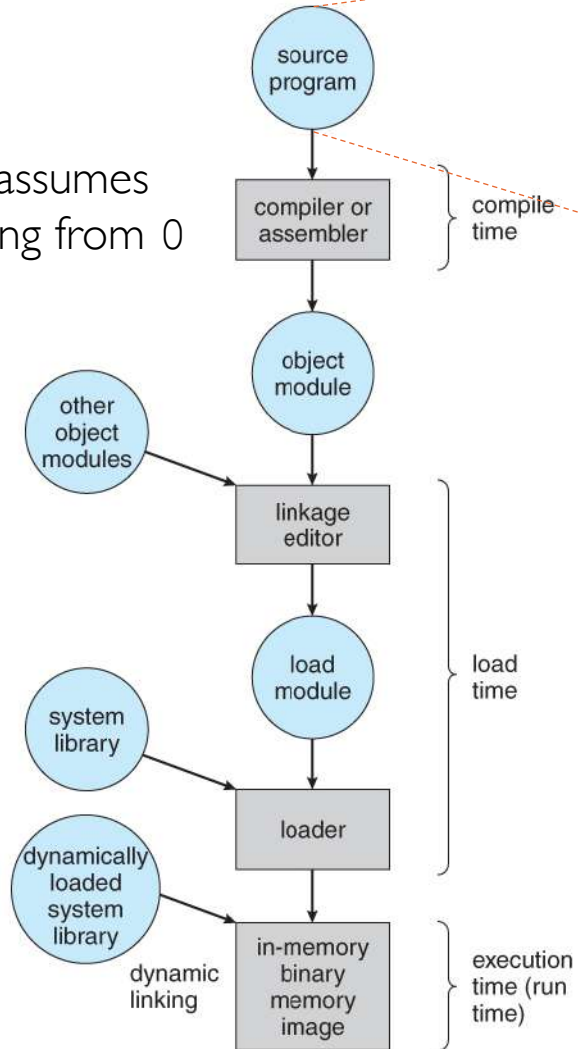
Most flexible solution implemented by the majority of modern OSs

Address Binding: Execution Time



Address Binding: Execution Time

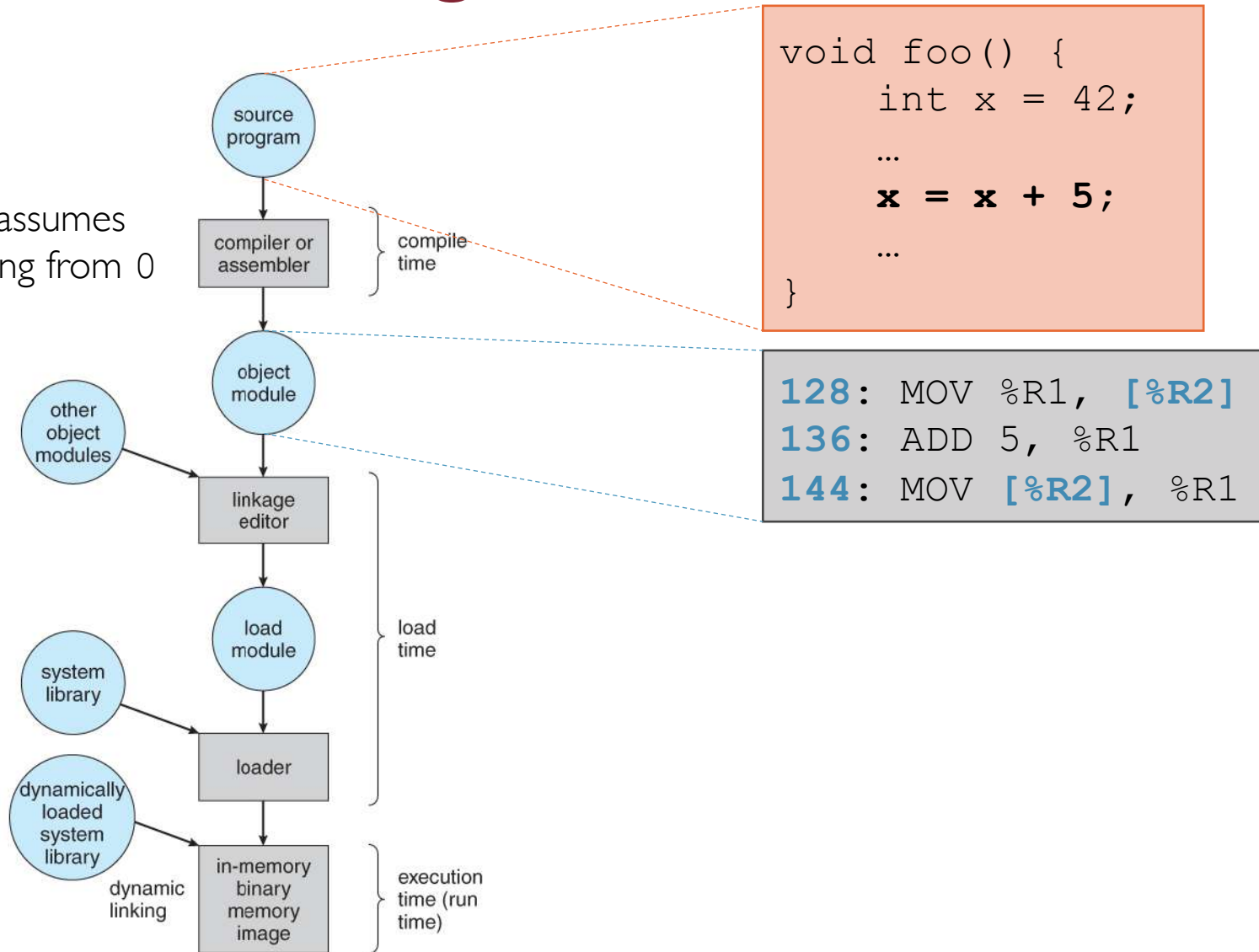
compiler still assumes
addresses starting from 0



```
void foo() {  
    int x = 42;  
    ...  
    x = x + 5;  
    ...  
}
```

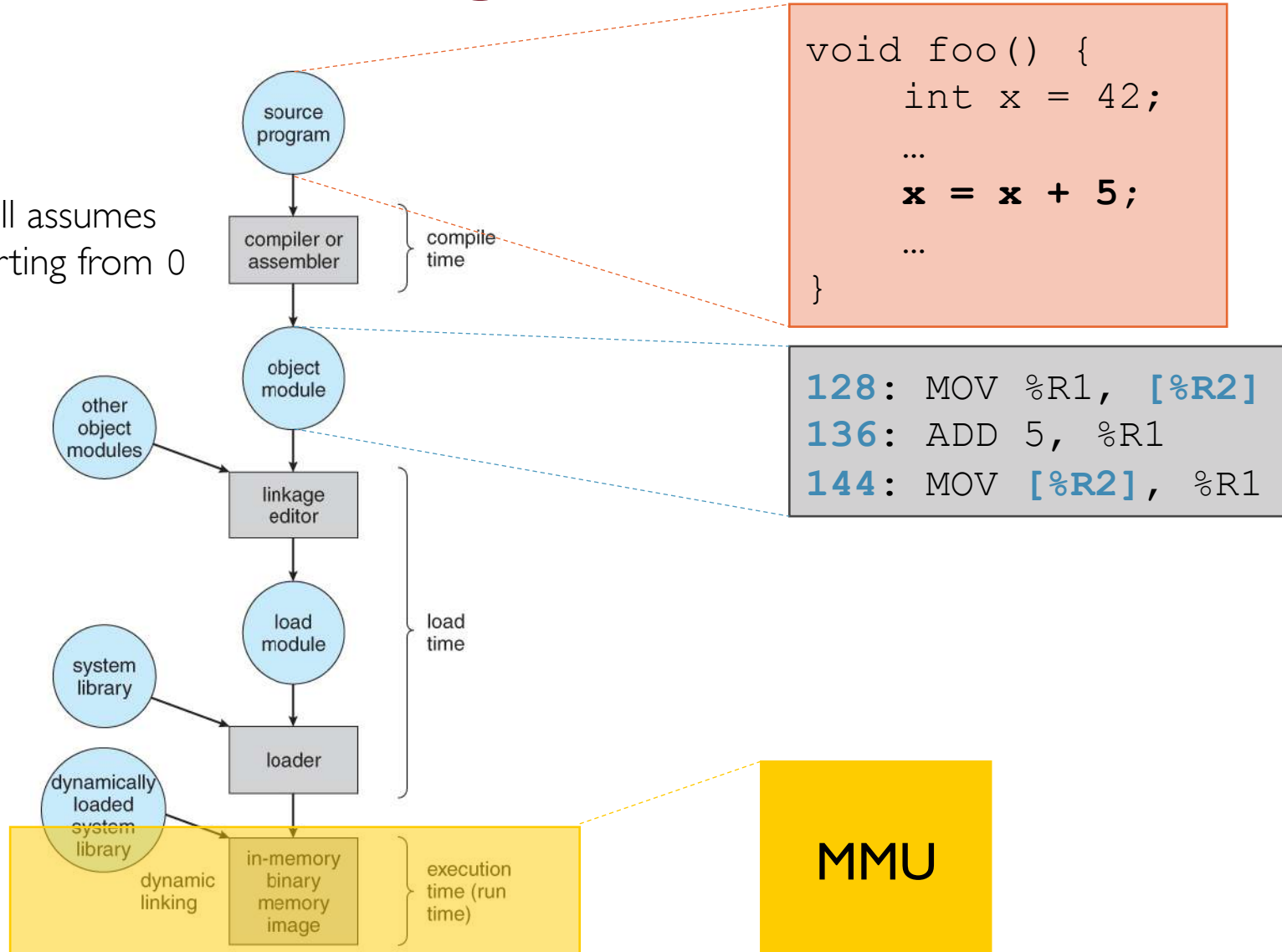
Address Binding: Execution Time

compiler still assumes
addresses starting from 0



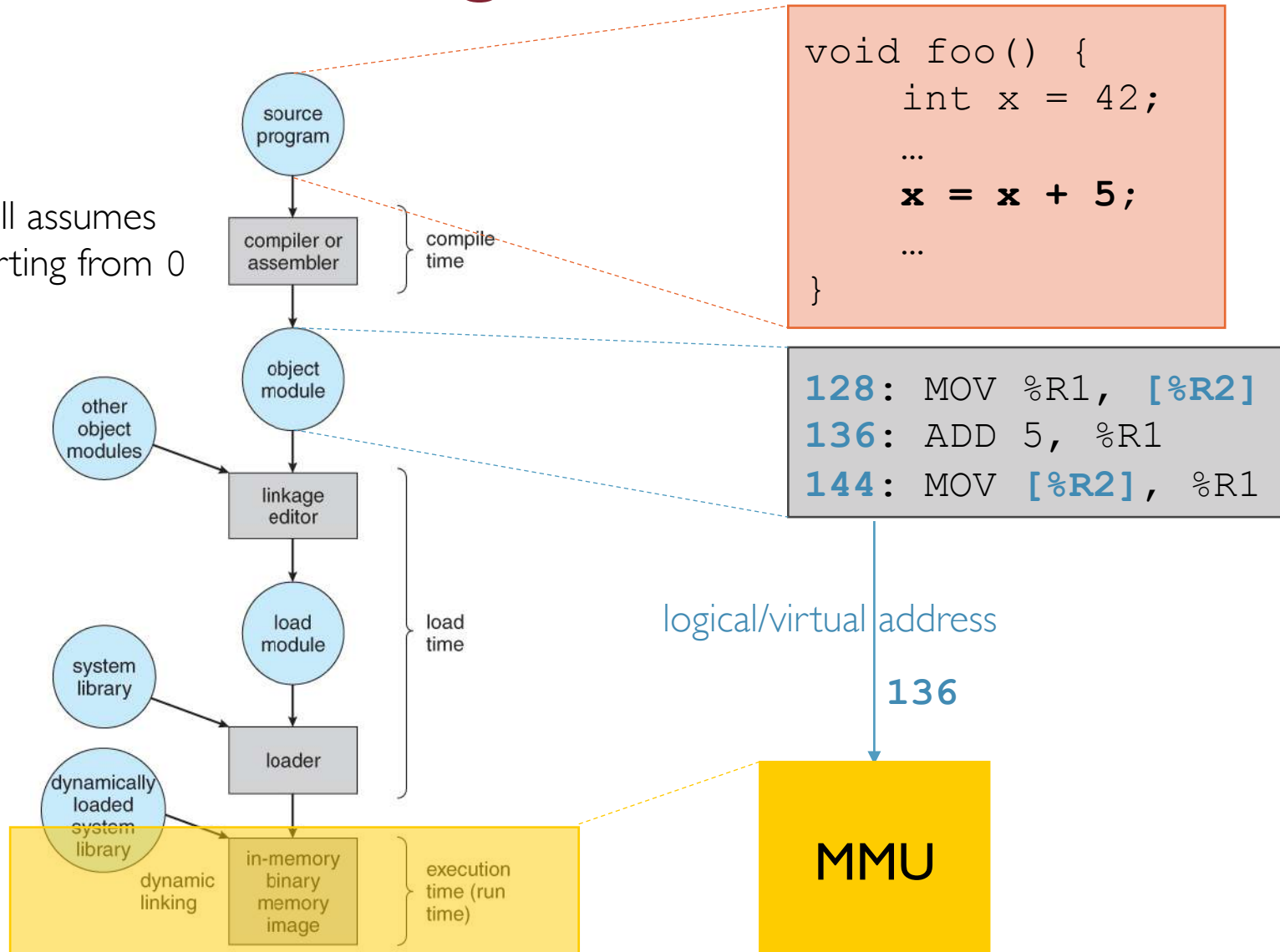
Address Binding: Execution Time

compiler still assumes
addresses starting from 0



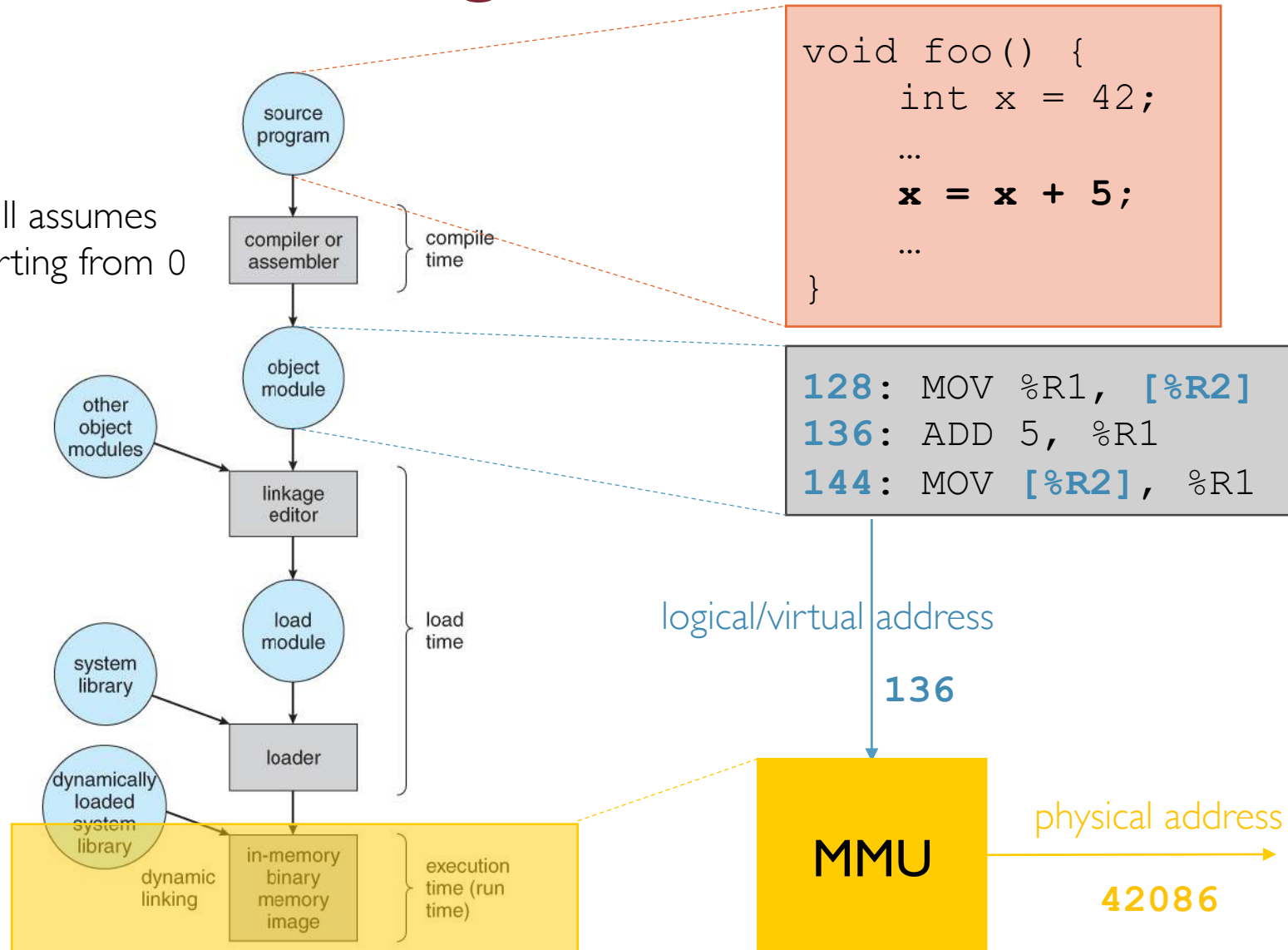
Address Binding: Execution Time

compiler still assumes
addresses starting from 0



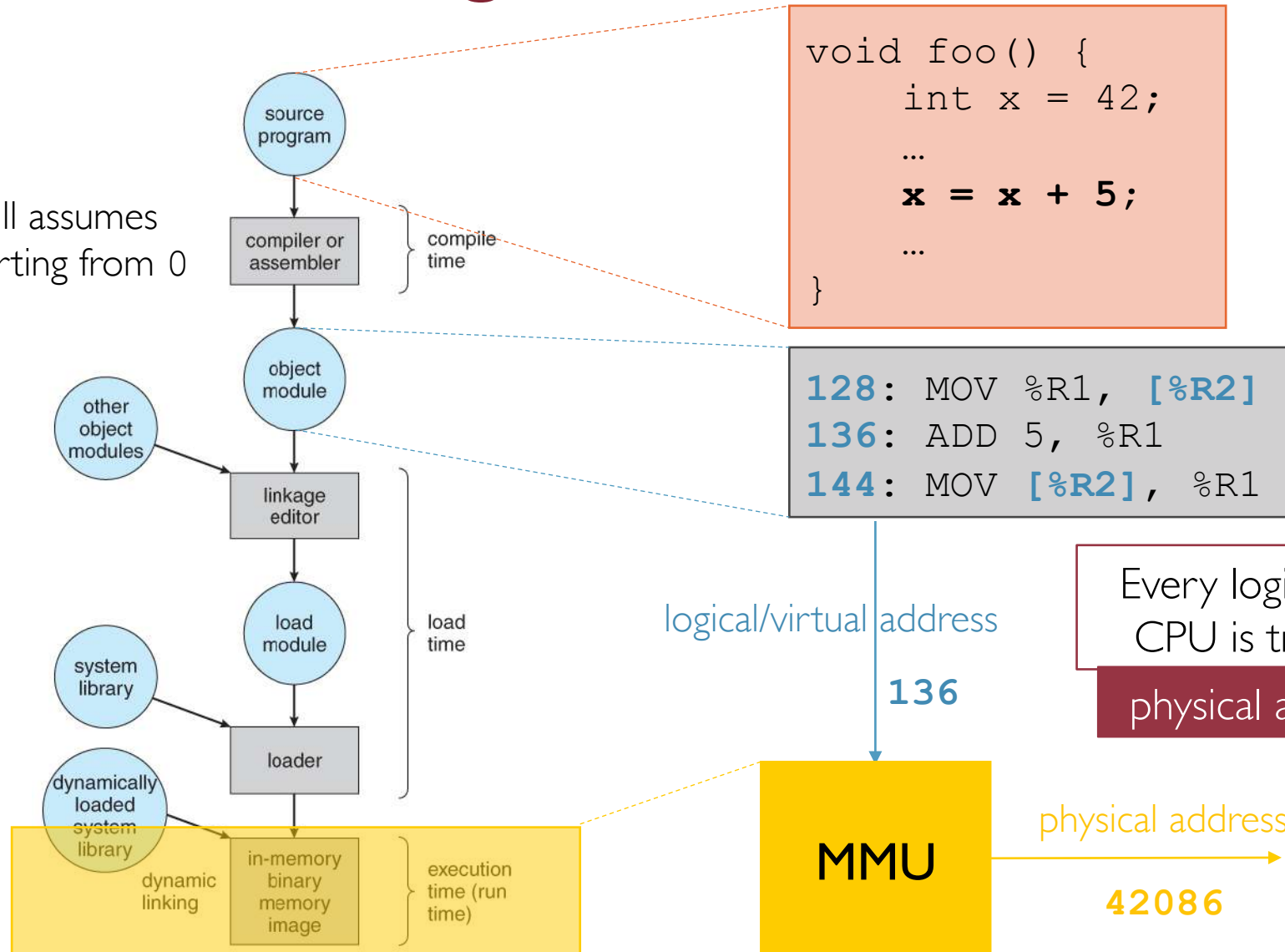
Address Binding: Execution Time

compiler still assumes
addresses starting from 0



Address Binding: Execution Time

compiler still assumes
addresses starting from 0



Manage Uniprogramming Memory

- Easiest memory management requirements

Manage Uniprogramming Memory

- Easiest memory management requirements
- The OS gets a fixed part of physical memory (e.g., highest memory addresses in MS-DOS)

Manage Uniprogramming Memory

- Easiest memory management requirements
- The OS gets a fixed part of physical memory (e.g., highest memory addresses in MS-DOS)
- Run one process at a time, always loaded starting at physical address $k = 0$ up to maximum address (`memory_size - os_size - 1`)

Manage Uniprogramming Memory

- Easiest memory management requirements
- The OS gets a fixed part of physical memory (e.g., highest memory addresses in MS-DOS)
- Run one process at a time, always loaded starting at physical address $k = 0$ up to maximum address (`memory_size - os_size - 1`)
- Each process executes in a contiguous segment of memory

Manage Uniprogramming Memory

- Easiest memory management requirements
- The OS gets a fixed part of physical memory (e.g., highest memory addresses in MS-DOS)
- Run one process at a time, always loaded starting at physical address $k = 0$ up to maximum address $(\text{memory_size} - \text{os_size} - 1)$
- Each process executes in a contiguous segment of memory
- Address binding can occur at compile time

Manage Uniprogramming Memory

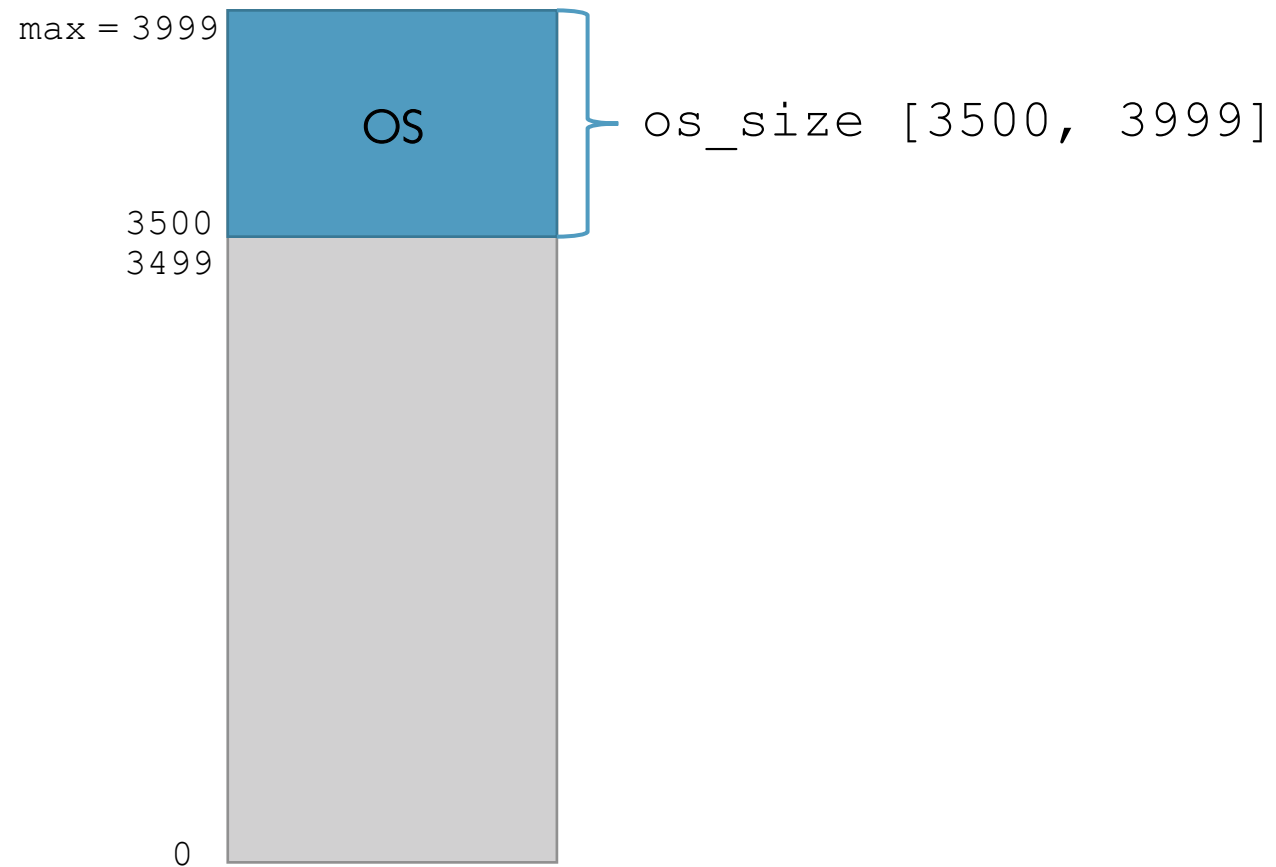
```
memory_size = 4000B = 4kB
```



Manage Uniprogramming Memory

`memory_size = 4000B = 4kB`

`os_size = 500B`



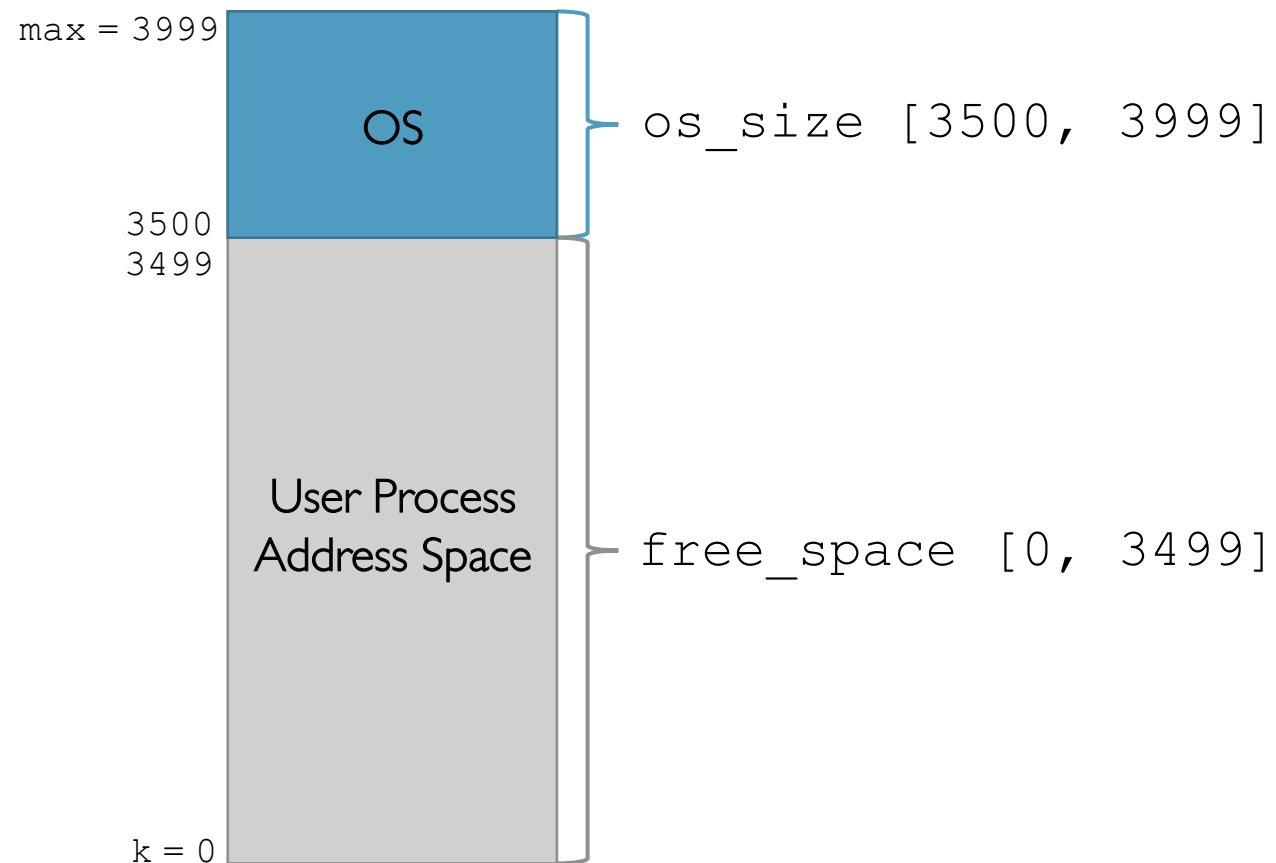
Manage Uniprogramming Memory

`memory_size = 4000B = 4kB`

`os_size = 500B`

`k = 0`

`free_space = 3.5kB`



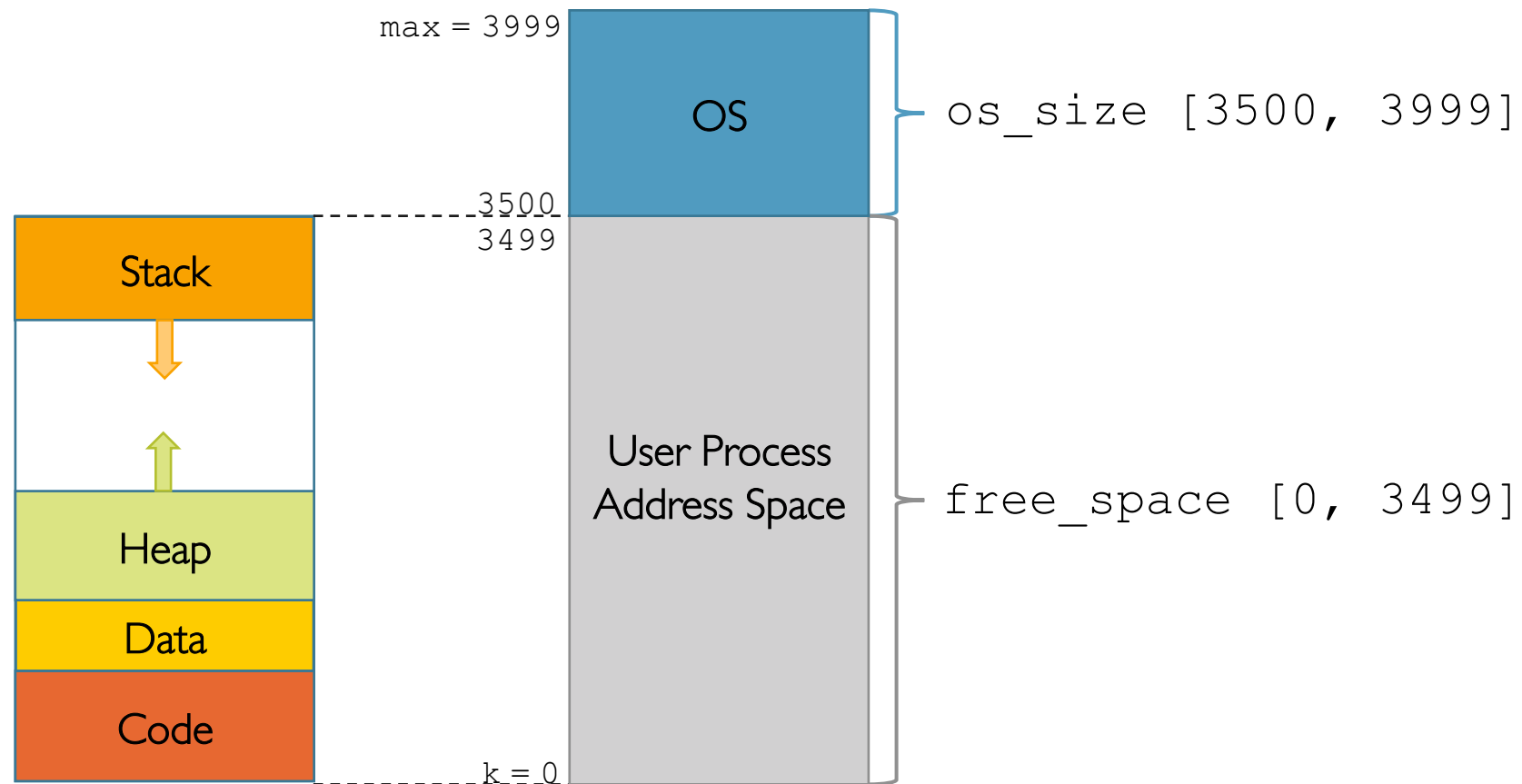
Manage Uniprogramming Memory

`memory_size = 4000B = 4kB`

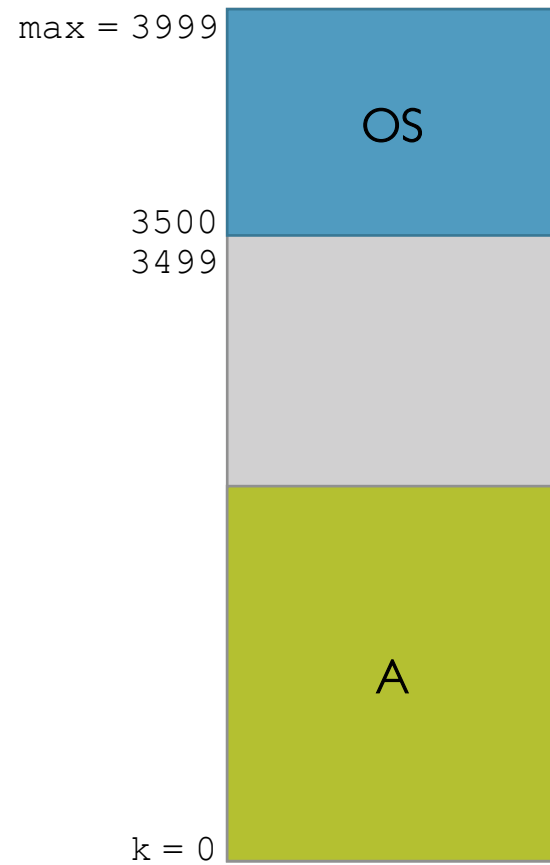
`os_size = 500B`

`k = 0`

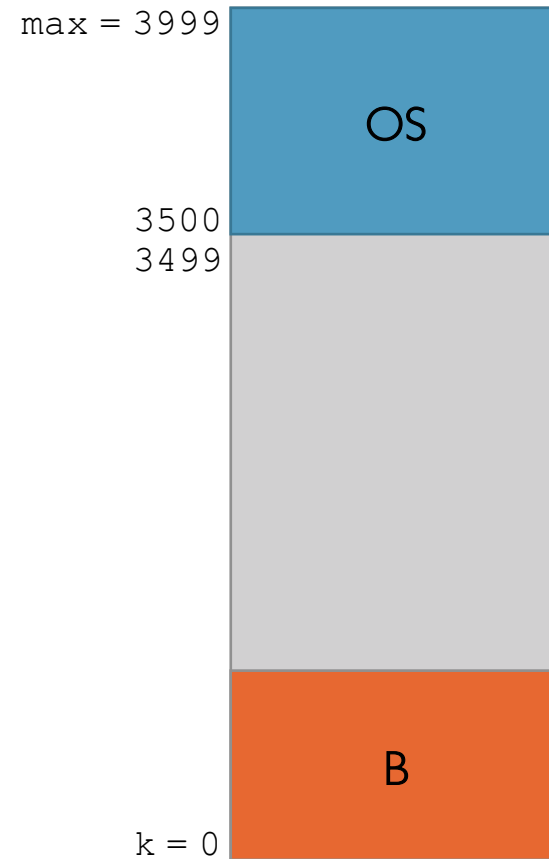
`free_space = 3.5kB`



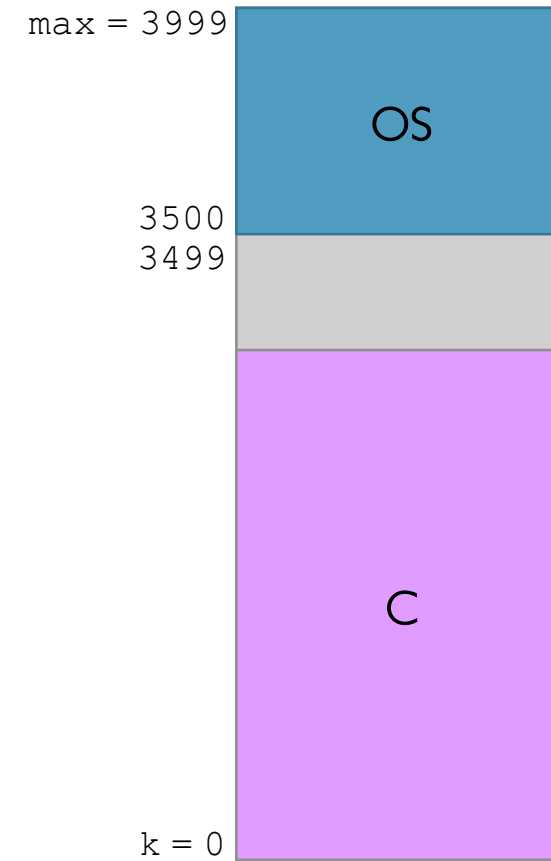
Manage Uniprogramming Memory



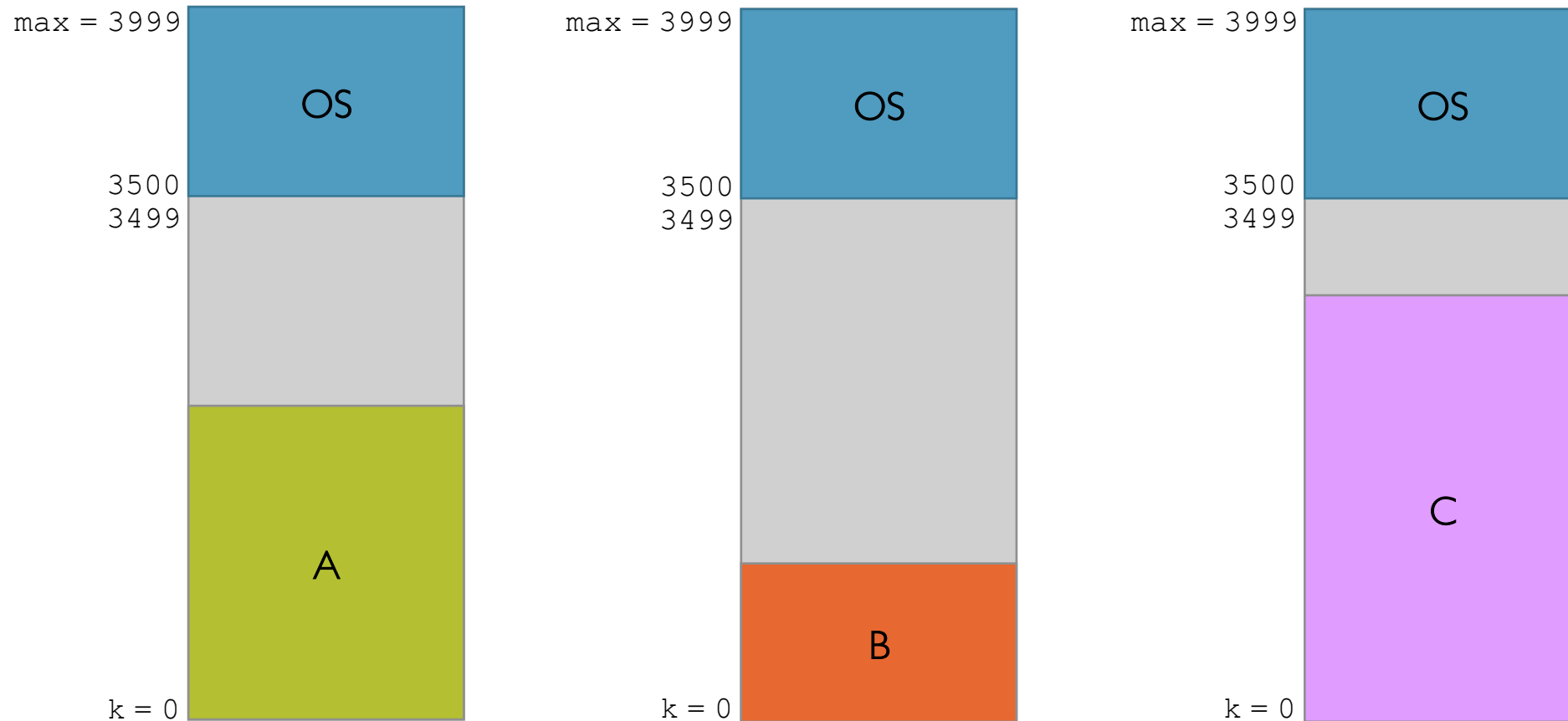
Manage Uniprogramming Memory



Manage Uniprogramming Memory



Manage Uniprogramming Memory



Very simple! But only one process executes at a time and no OS protection

Manage Multiprogramming Memory: Goals (I)

- Sharing
 - Several processes coexist in main memory at the same time
 - Cooperating processes can share portions of address space

Manage Multiprogramming Memory: Goals (I)

- Sharing

- Several processes coexist in main memory at the same time
- Cooperating processes can share portions of address space

- Transparency

- Processes should not be aware that memory is shared
- Processes should not be aware of which portions of physical memory they are assigned to

Manage Multiprogramming Memory: Goals (II)

- Protection/Security
 - Processes must not be able to corrupt each other or the OS
 - Processes must not be able to read data of other processes

Manage Multiprogramming Memory: Goals (II)

- Protection/Security

- Processes must not be able to corrupt each other or the OS
- Processes must not be able to read data of other processes

- Efficiency

- CPU and memory performance should not degrade badly due to sharing
- Keep memory fragmentation low

Relocation: Initial Idea

- Assume the OS is allocated to the highest memory addresses

Relocation: Initial Idea

- Assume the OS is allocated to the highest memory addresses
- Assume logical addresses generated by each user process starts at 0 up to maximum address (`memory_size - os_size - 1`)

Relocation: Initial Idea

- Assume the OS is allocated to the highest memory addresses
- Assume logical addresses generated by each user process starts at 0 up to maximum address (`memory_size - os_size - 1`)
- Load a process by allocating the first **contiguous segment** of memory in which the process fits

Relocation: Initial Idea

- Assume the OS is allocated to the highest memory addresses
- Assume logical addresses generated by each user process starts at 0 up to maximum address (`memory_size - os_size - 1`)
- Load a process by allocating the first **contiguous segment** of memory in which the process fits
- Allow transparent sharing of memory: each process' address space may be placed anywhere in memory

Static Relocation

- The OS loader rewrites the addresses generated by a process, so as to reflect its position in main memory (**load time binding**)

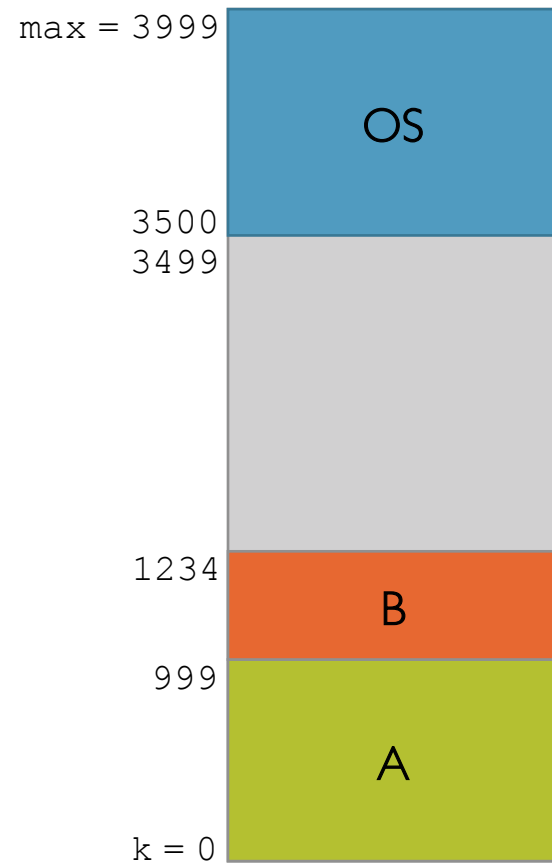
Static Relocation

- The OS loader rewrites the addresses generated by a process, so as to reflect its position in main memory (**load time binding**)
- PRO:
 - No HW support is needed

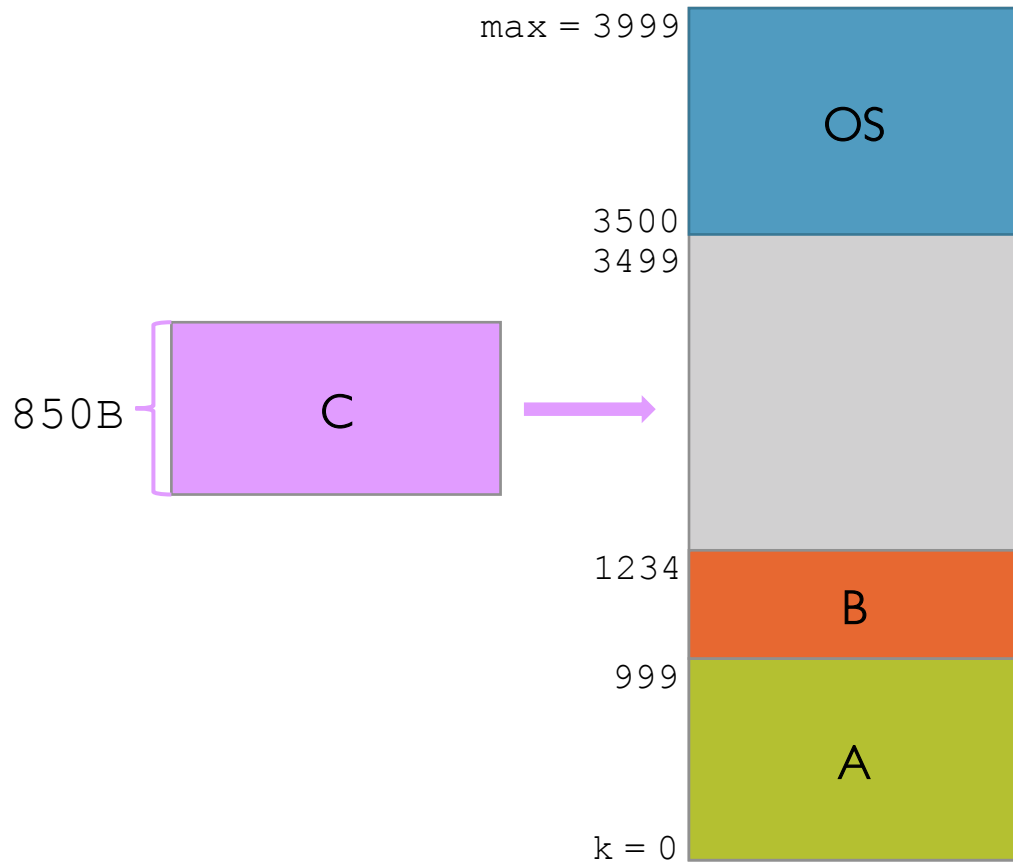
Static Relocation

- The OS loader rewrites the addresses generated by a process, so as to reflect its position in main memory (**load time binding**)
- **PRO:**
 - No HW support is needed
- **CONS:**
 - No protection/privacy → processes can corrupt the OS or other processes
 - Address space must be allocated contiguously → assuming worst-case stack and heap request
 - The OS cannot move a process (address space) once allocated in memory

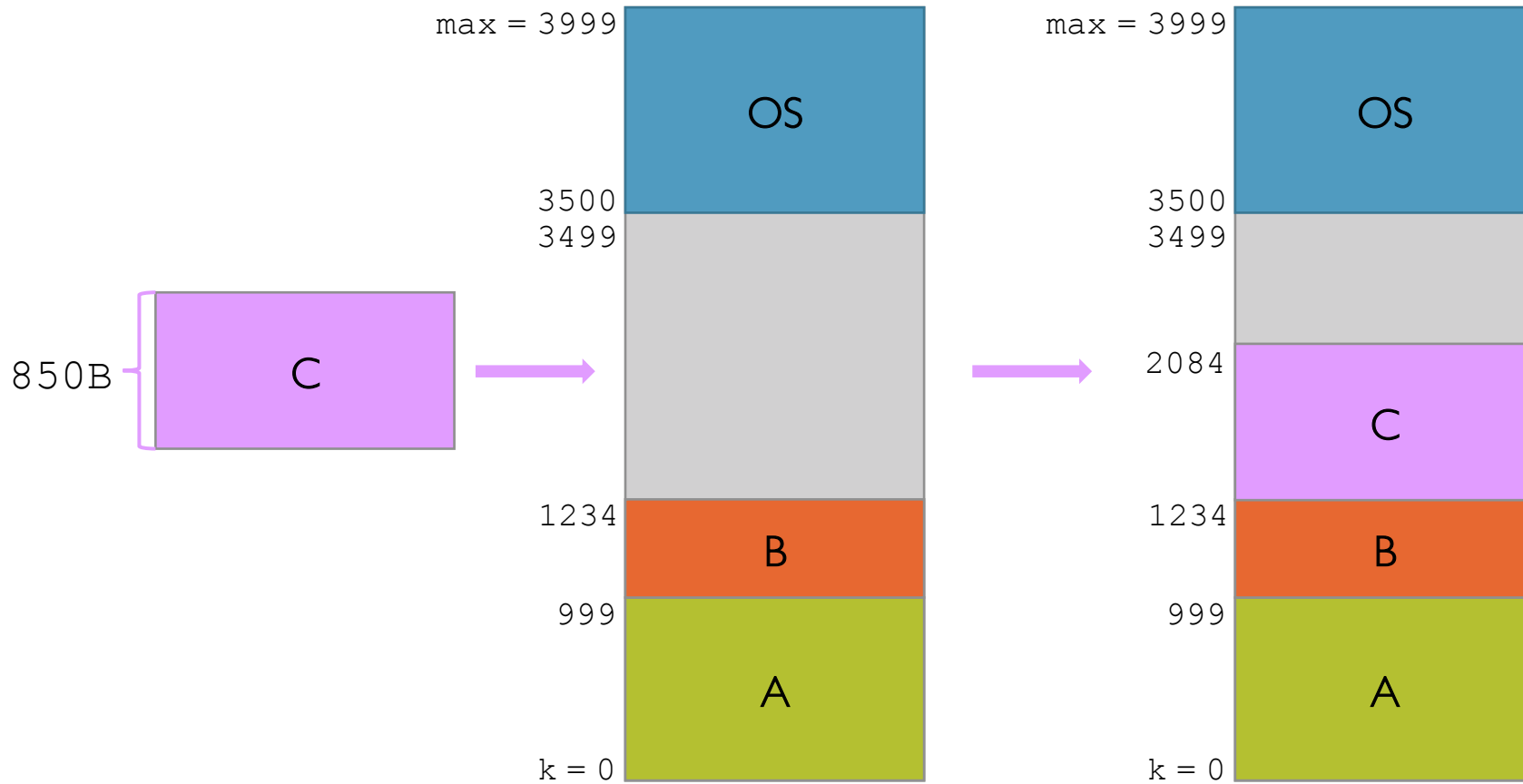
Static Relocation



Static Relocation



Static Relocation



Dynamic Relocation

- Protect OS and processes from one another

Dynamic Relocation

- Protect OS and processes from one another
- Requires hardware support (Memory Management Unit or MMU)

Dynamic Relocation

- Protect OS and processes from one another
- Requires hardware support (Memory Management Unit or MMU)
- Address binding at execution/run time

Dynamic Relocation

- Protect OS and processes from one another
- Requires hardware support (Memory Management Unit or MMU)
- Address binding at execution/run time
- MMU dynamically translates each logical/virtual address generated by a process into a corresponding physical address

Dynamic Relocation

- Protect OS and processes from one another
- Requires hardware support (Memory Management Unit or MMU)
- Address binding at execution/run time
- MMU dynamically translates each logical/virtual address generated by a process into a corresponding physical address



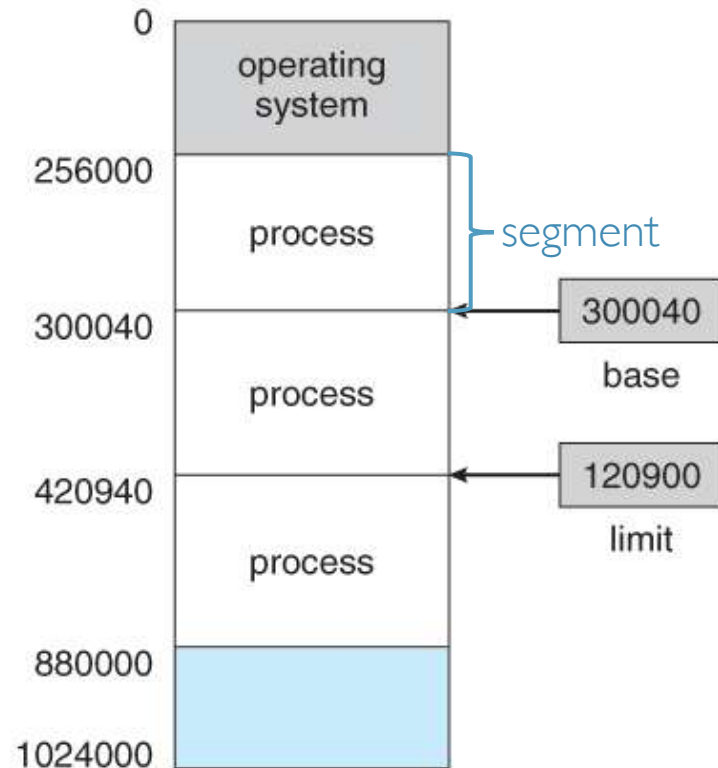
HW Support for Dynamic Relocation

- MMU contains at least 2 registers:
 - **base** → start physical memory location of address space
 - **limit** → size limit of address space

HW Support for Dynamic Relocation

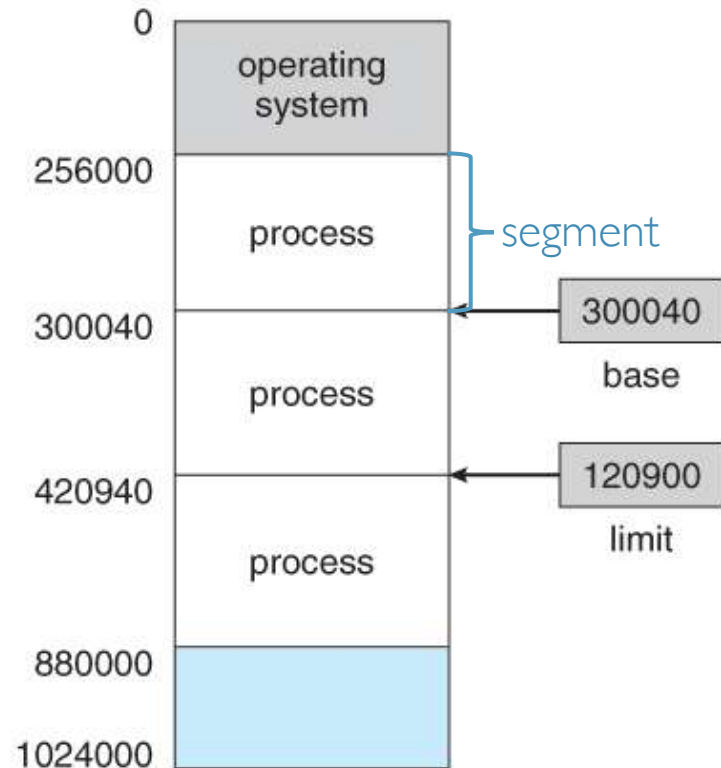
- MMU contains at least 2 registers:
 - **base** → start physical memory location of address space
 - **limit** → size limit of address space
- CPU supports at least 2 operating modes:
 - **privileged (kernel) mode** when the OS is running
 - after issuing any trap (system call, interruption, or exception)
 - when manipulating sensitive resources (e.g., the content of MMU registers)
 - **user mode** when user process is running
 - while executing process instructions on the CPU

Base and Limit Registers: Idea



Each process is given a **contiguous segment** of main memory when loaded

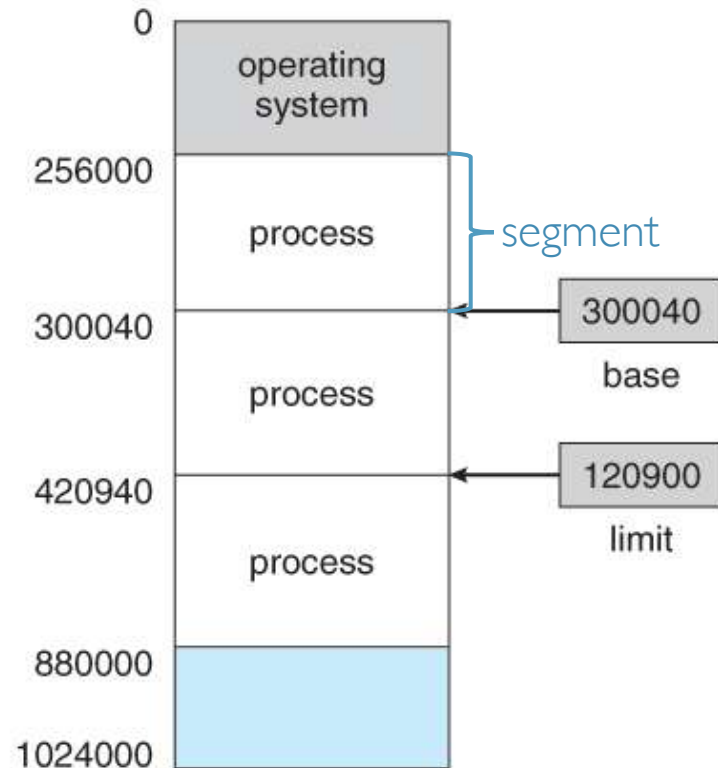
Base and Limit Registers: Idea



Each process is given a **contiguous segment** of main memory when loaded

As such, each process can only access to memory locations belonging to the segment it is assigned to

Base and Limit Registers: Idea



Each process is given a **contiguous segment** of main memory when loaded

As such, each process can only access to memory locations belonging to the segment it is assigned to

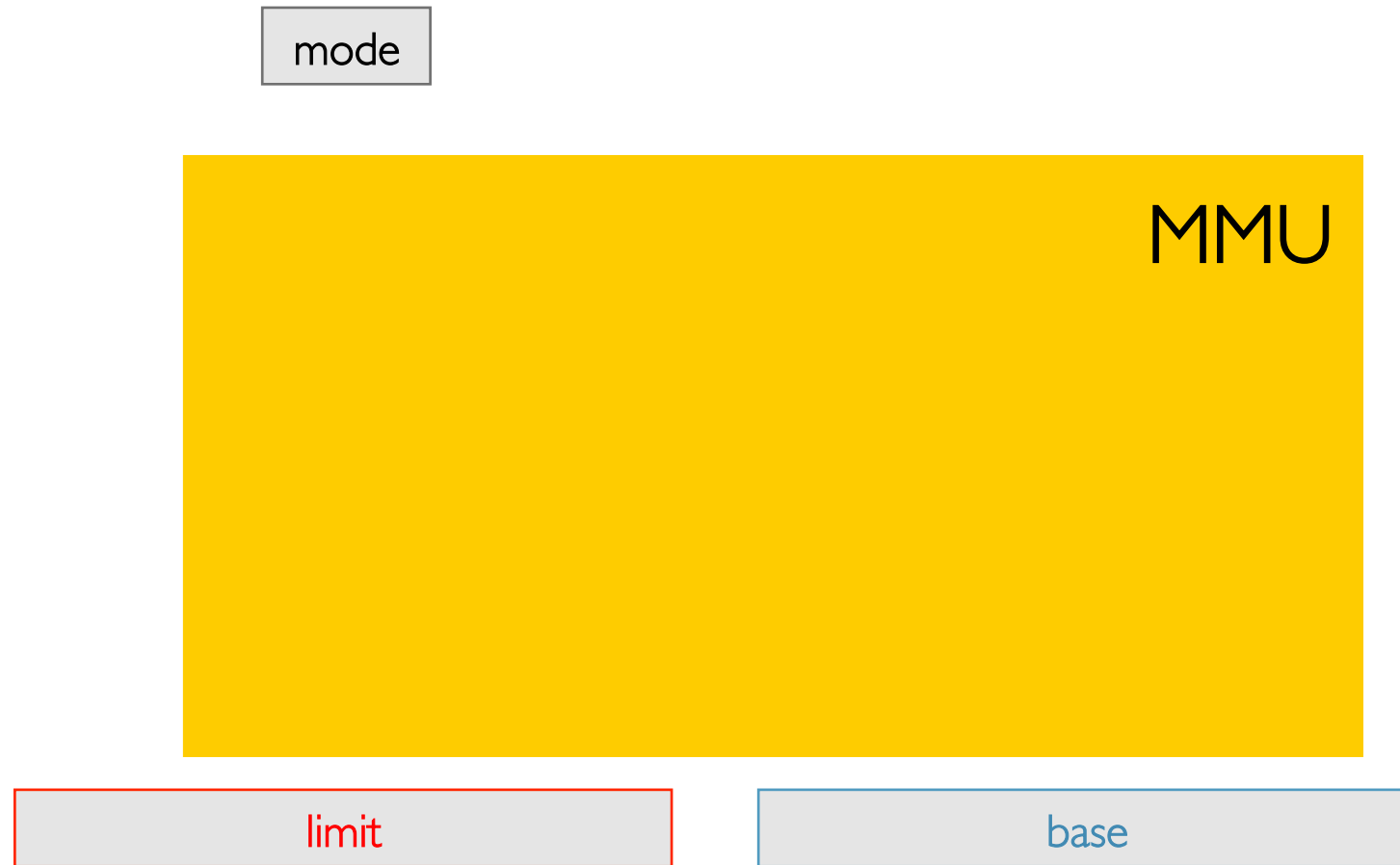
Protection implemented using two MMU registers: **base** and **limit**

Implementing Dynamic Relocation

CPU must check every memory access generated in user mode (i.e., by a user process) is within the correct [`base`, `base` + `limit`) range for *that* process

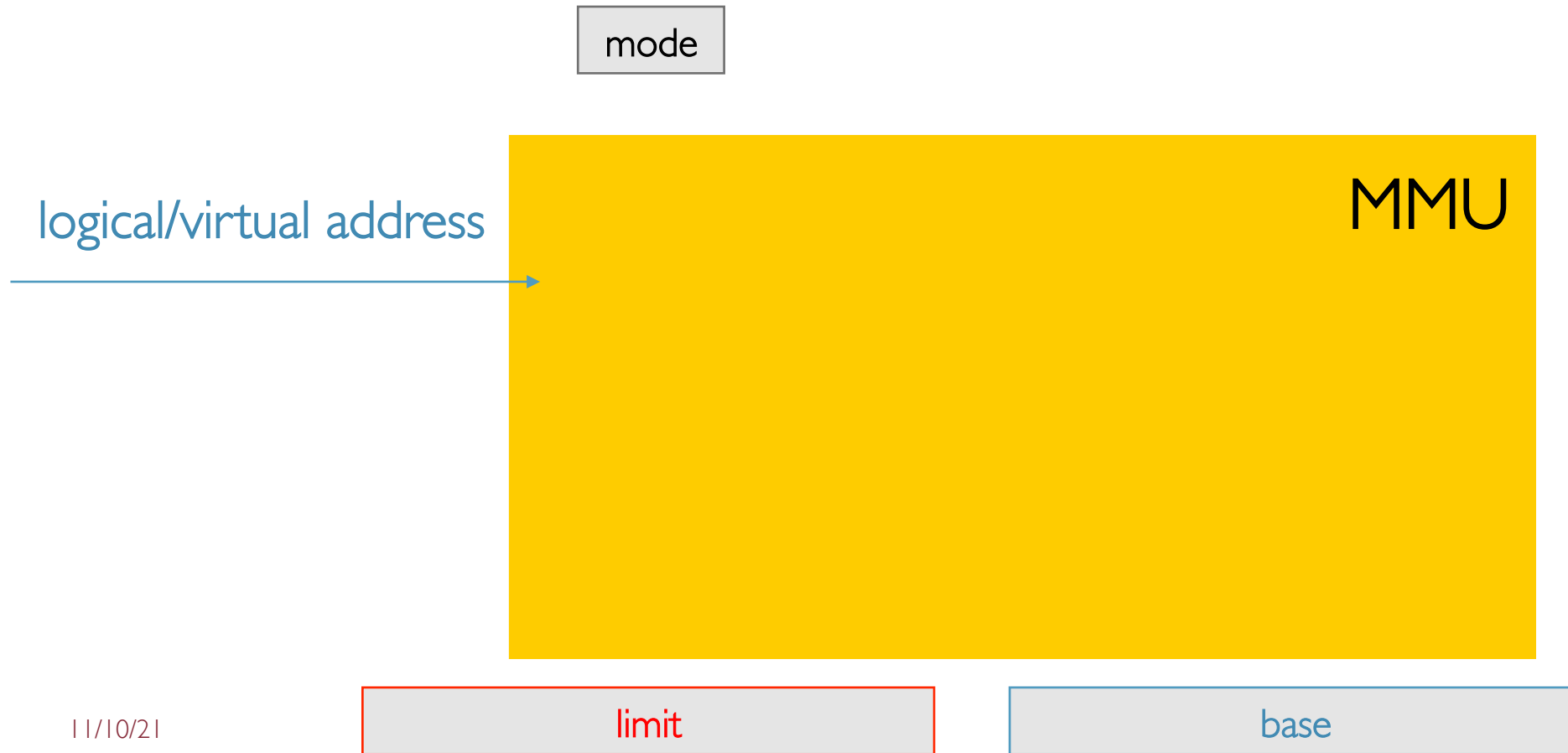
Implementing Dynamic Relocation

CPU must check every memory access generated in user mode (i.e., by a user process) is within the correct [base , $\text{base} + \text{limit}$) range for *that* process



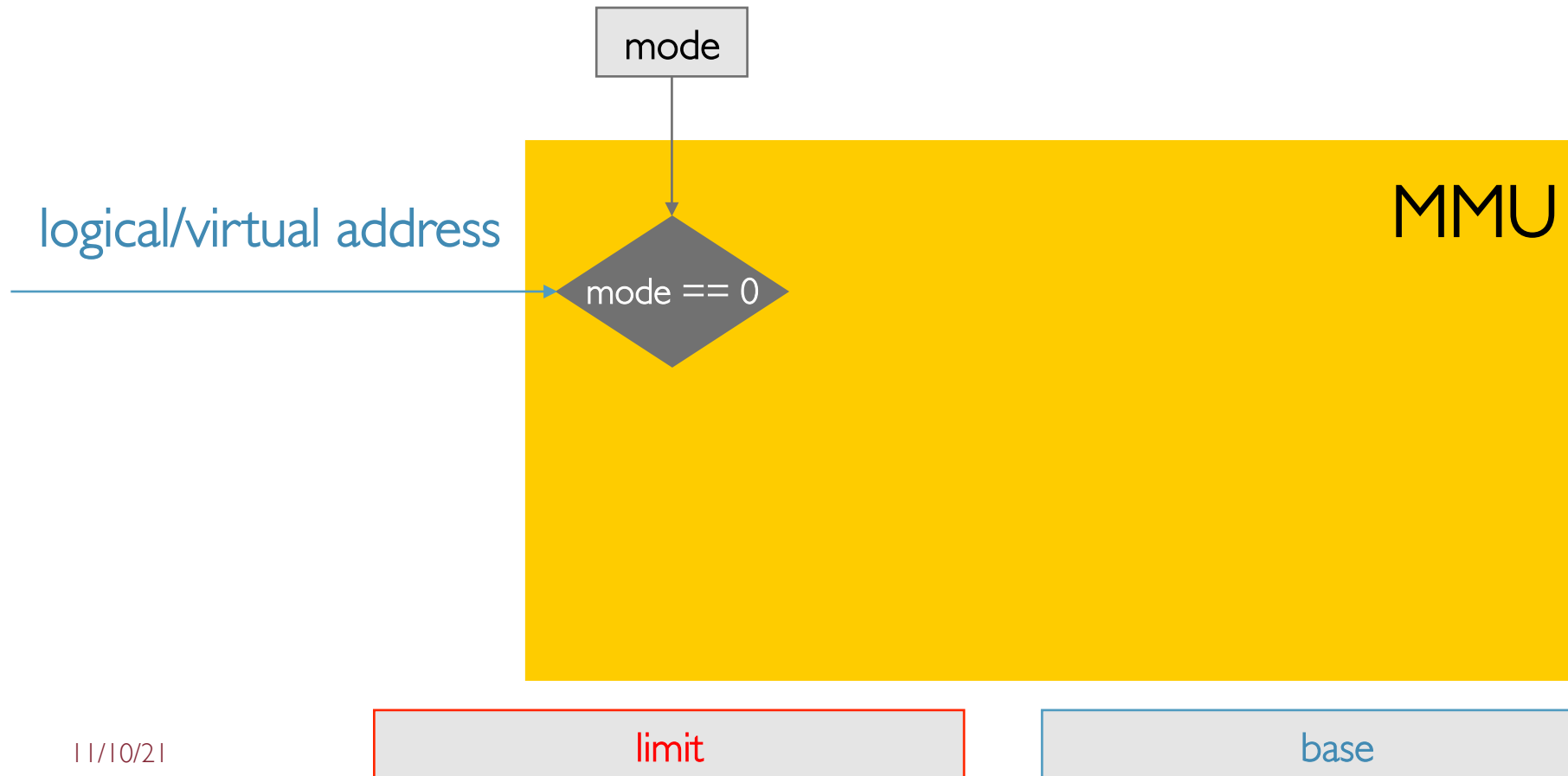
Implementing Dynamic Relocation

CPU must check every memory access generated in user mode (i.e., by a user process) is within the correct [base , $\text{base} + \text{limit}$) range for *that* process



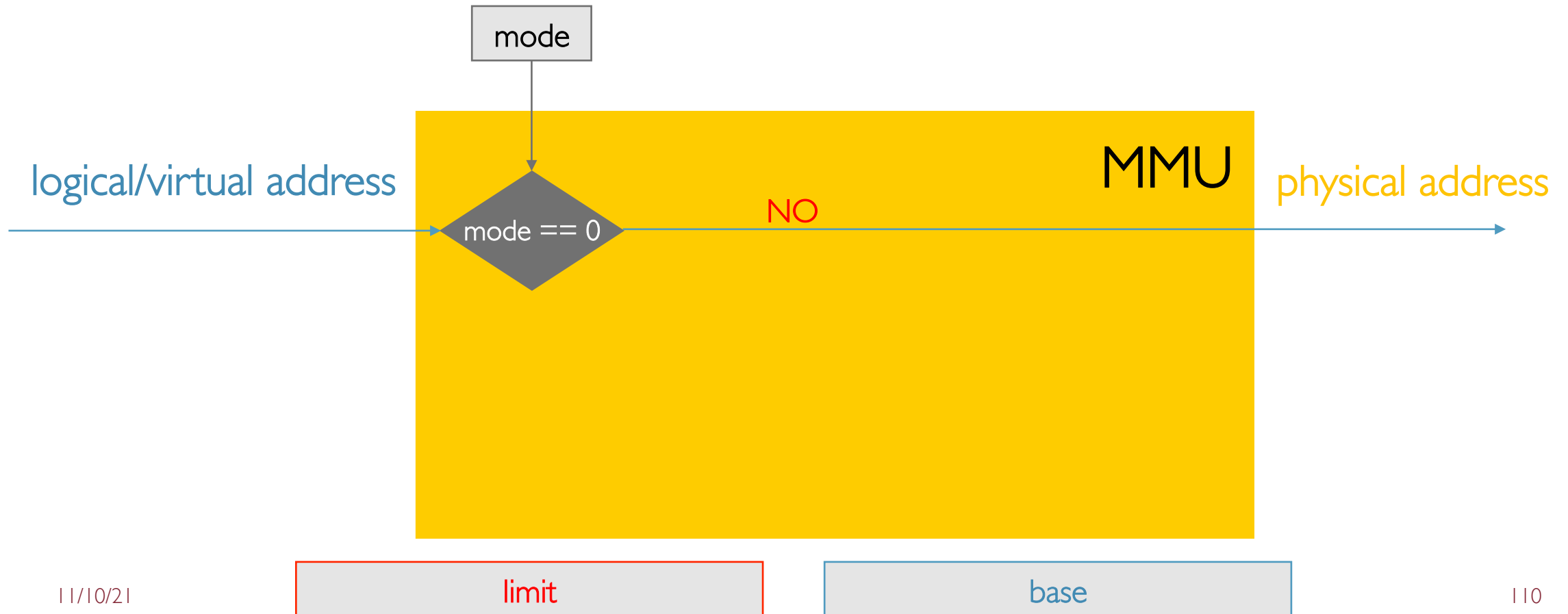
Implementing Dynamic Relocation

CPU must check every memory access generated in user mode (i.e., by a user process) is within the correct [base , $\text{base} + \text{limit}$) range for *that* process



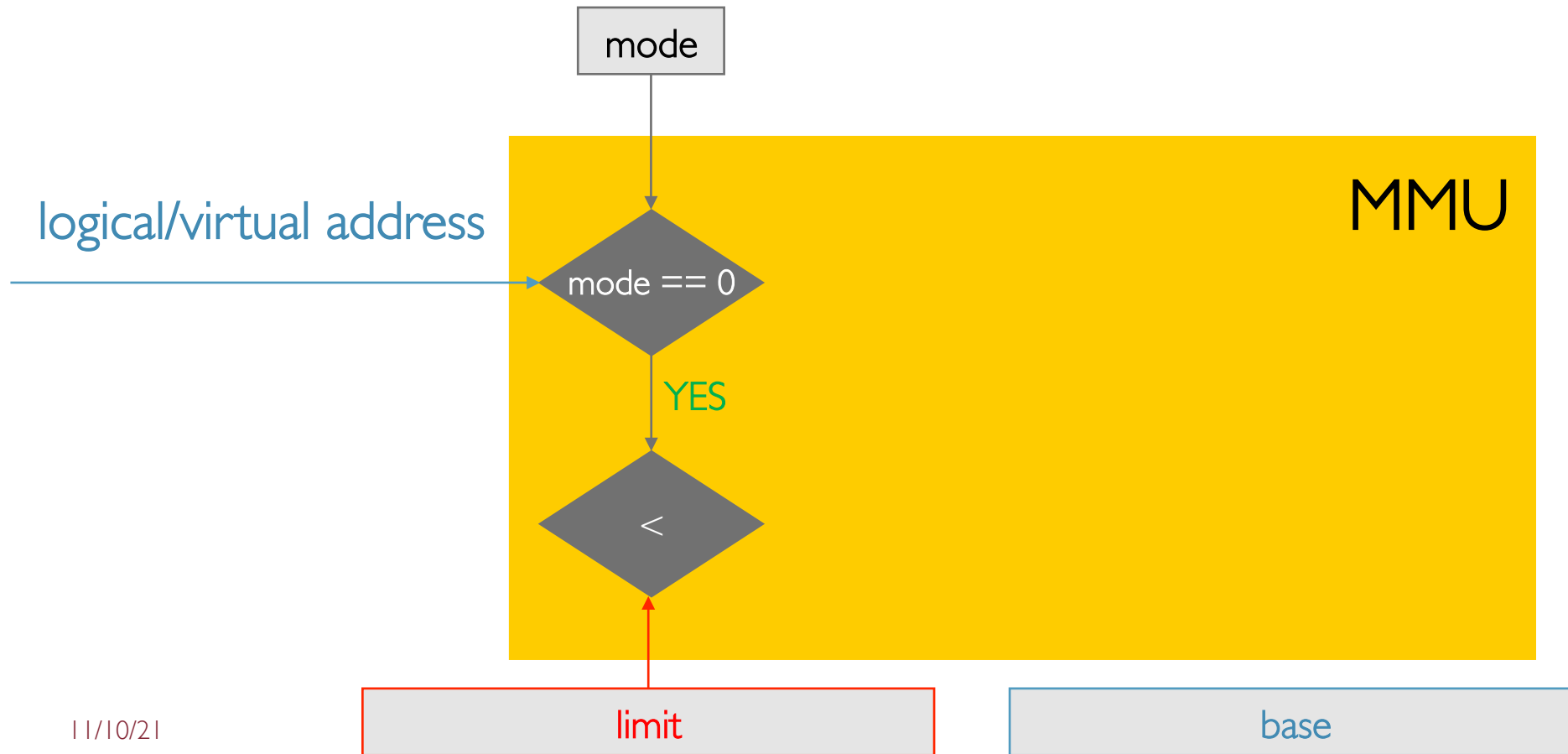
Implementing Dynamic Relocation

CPU must check every memory access generated in user mode (i.e., by a user process) is within the correct [base , $\text{base} + \text{limit}$) range for *that* process



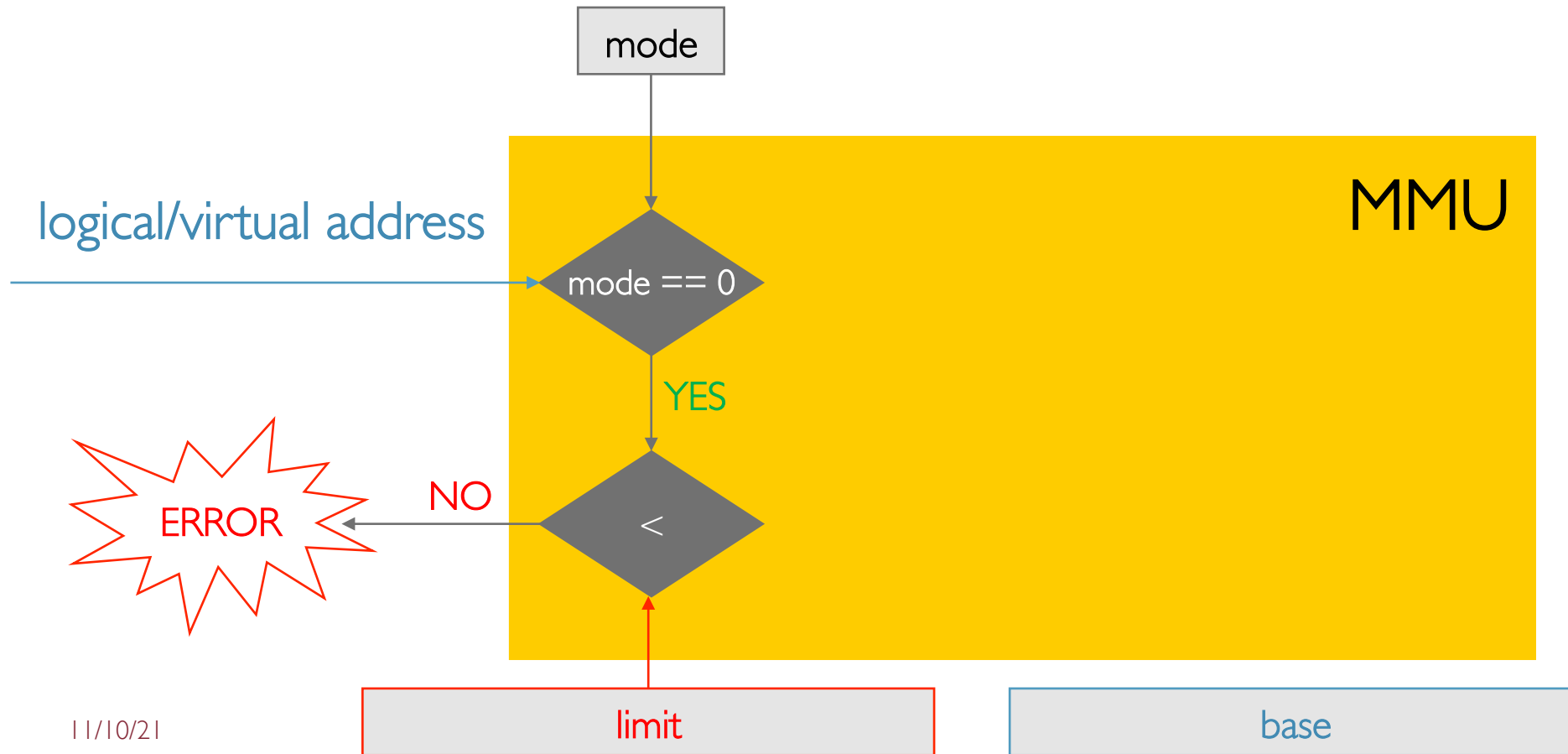
Implementing Dynamic Relocation

CPU must check every memory access generated in user mode (i.e., by a user process) is within the correct [base , $\text{base} + \text{limit}$) range for *that* process



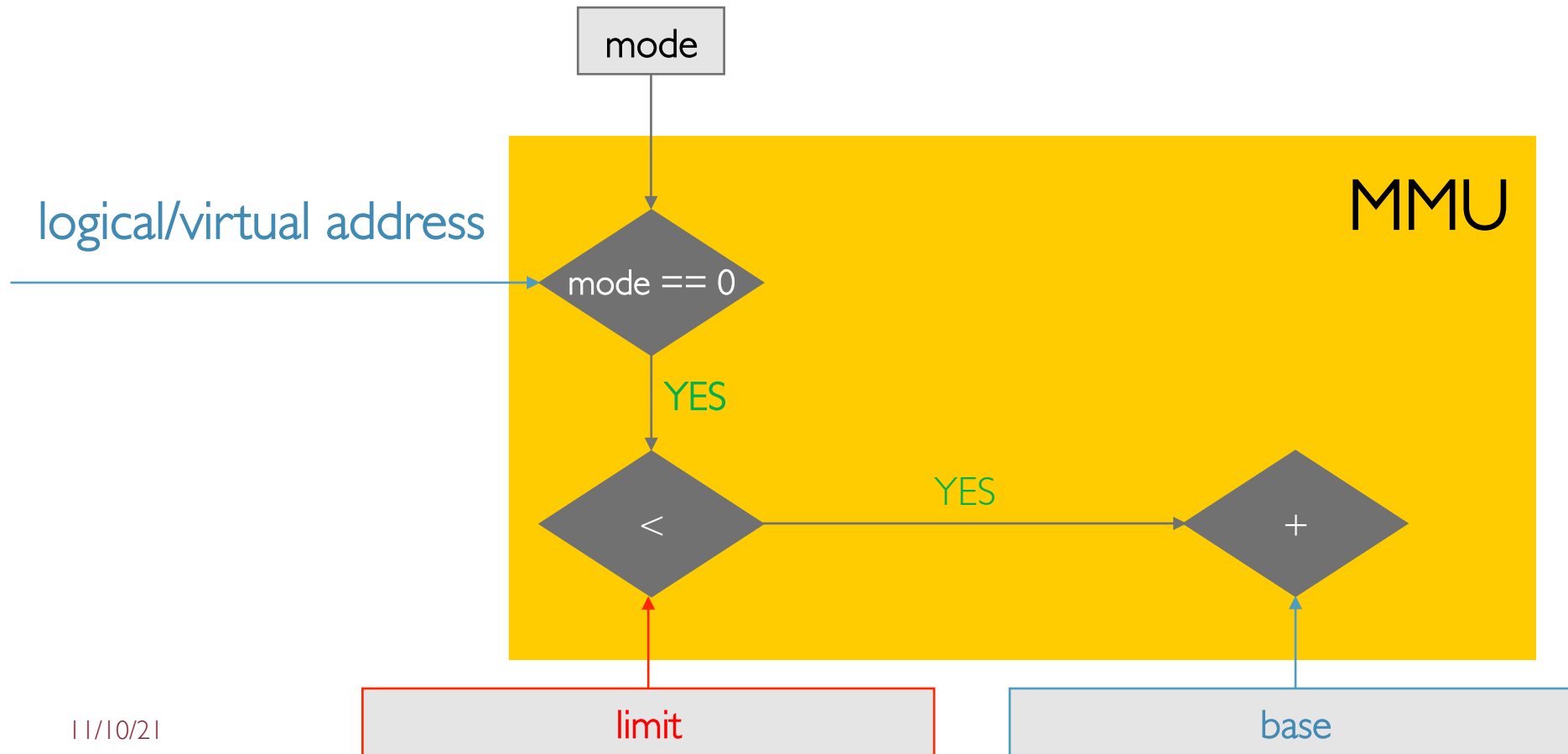
Implementing Dynamic Relocation

CPU must check every memory access generated in user mode (i.e., by a user process) is within the correct $[\text{base}, \text{base} + \text{limit})$ range for *that* process



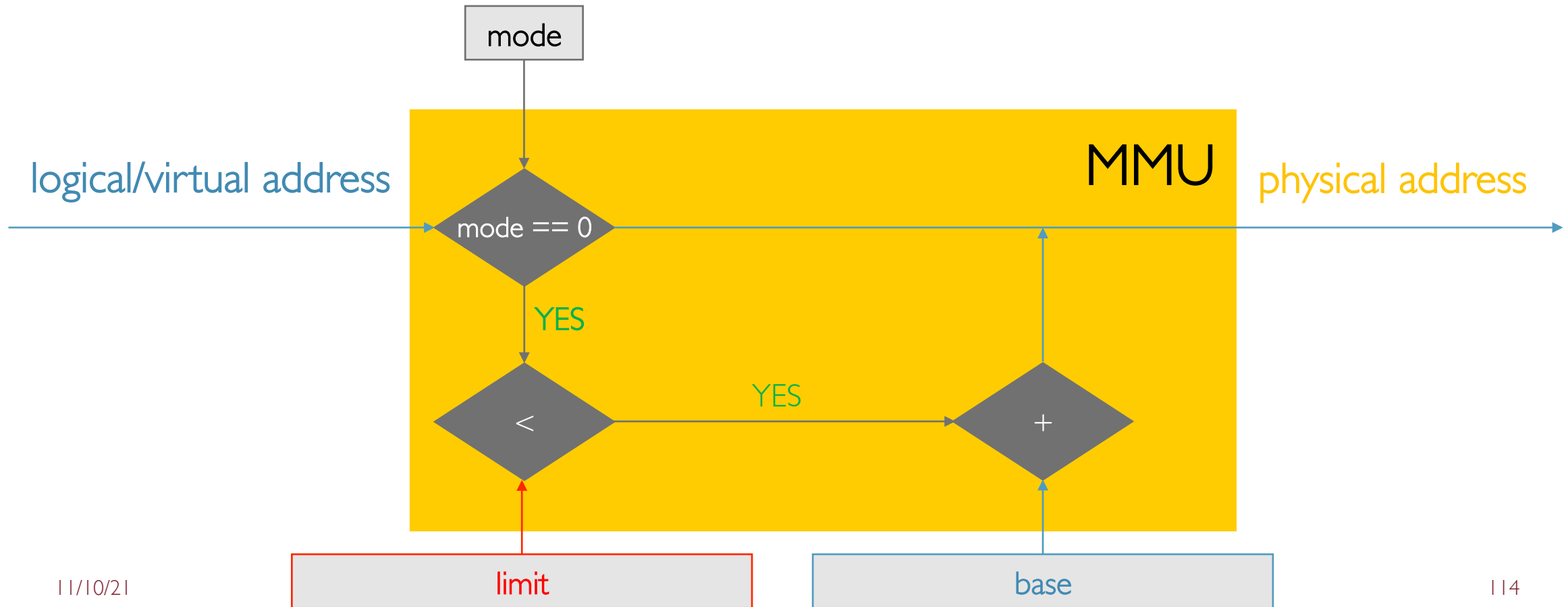
Implementing Dynamic Relocation

CPU must check every memory access generated in user mode (i.e., by a user process) is within the correct [base , $\text{base} + \text{limit}$) range for *that* process



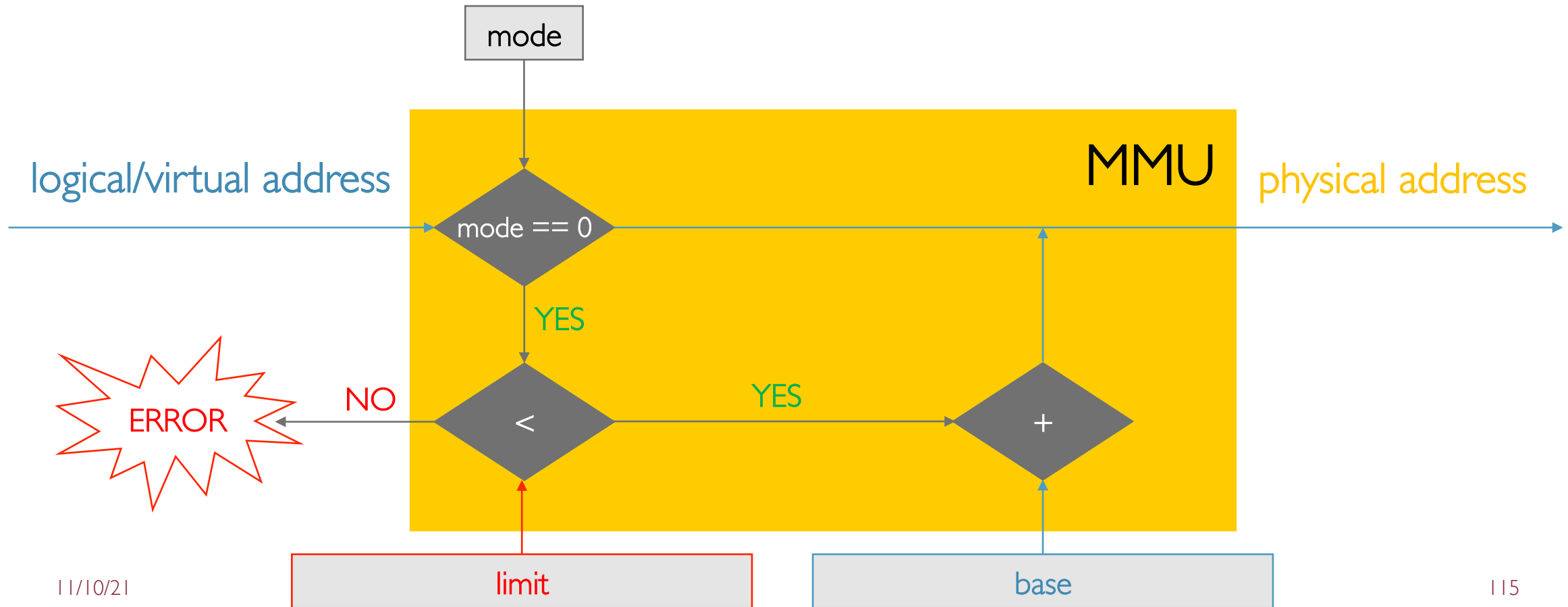
Implementing Dynamic Relocation

CPU must check every memory access generated in user mode (i.e., by a user process) is within the correct [base , $\text{base} + \text{limit}$) range for *that* process



Implementing Dynamic Relocation

CPU must check every memory access generated in user mode (i.e., by a user process) is within the correct [base , $\text{base} + \text{limit}$) range for *that* process



Dynamic Relocation

- PROs:
 - Provides protection (both read and write) across address spaces
 - OS can easily move a process during execution
 - OS can allow process to dynamically grow over time
 - Simple, fast hardware implementation (MMU):
 - 2 special registers, one add and one compare operation (can be done in parallel)

Dynamic Relocation

- **CONS:**
 - Little hardware overhead to pay at each memory reference
 - Each process must still be allocated contiguously in physical memory (possible memory waste)
 - Process is still limited to physical memory size
 - Degree of multiprogramming is bound since all memory of all active processes must fit in memory
 - No partial sharing of address space (e.g., processes can't share program's text)

Relocation: Properties

- Sharing/Transparency → processes are unaware of sharing memory
- Protection/Security → each memory reference is checked in HW
- Efficiency → somewhat achieved but if a process grows it may need to be moved to other location (very slow)

Relocation: Properties

Static Relocation

- Sharing/Transparency → processes are unaware of sharing memory
- Protection/Security → each memory reference is checked in HW
- Efficiency → somewhat achieved but if a process grows it may need to be moved to other location (very slow)

Relocation: Properties

Dynamic Relocation

- Sharing/Transparency → processes are unaware of sharing memory
- Protection/Security → each memory reference is checked in HW
- Efficiency → somewhat achieved but if a process grows it may need to be moved to other location (very slow)

Summary

- Effective memory management is crucial for system performance
- Very basic management doesn't even require OS intervention
- Modern OSs manage memory ensuring:
 - Transparency → logical/virtual vs. physical address space
 - Protection/Flexibility → dynamic relocation
 - Efficiency → hardware support
- We are still assuming the whole virtual address space of a process is fully and contiguously loaded in main memory → **serious limitation!**