Sistemi Operativi

Corso di Laurea in Informatica

a.a. 2020-2021

Gabriele Tolomei

Dipartimento di Informatica

Sapienza Università di Roma

tolomei@di.uniroma1.it

SAPIENZA
UNIVERSITÀ DI ROMA

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
  - Distributed Systems (?)

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 the program

CPU Must Refer To Memory Addresses

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

CPU Must Refer To Memory Addresses

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

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?

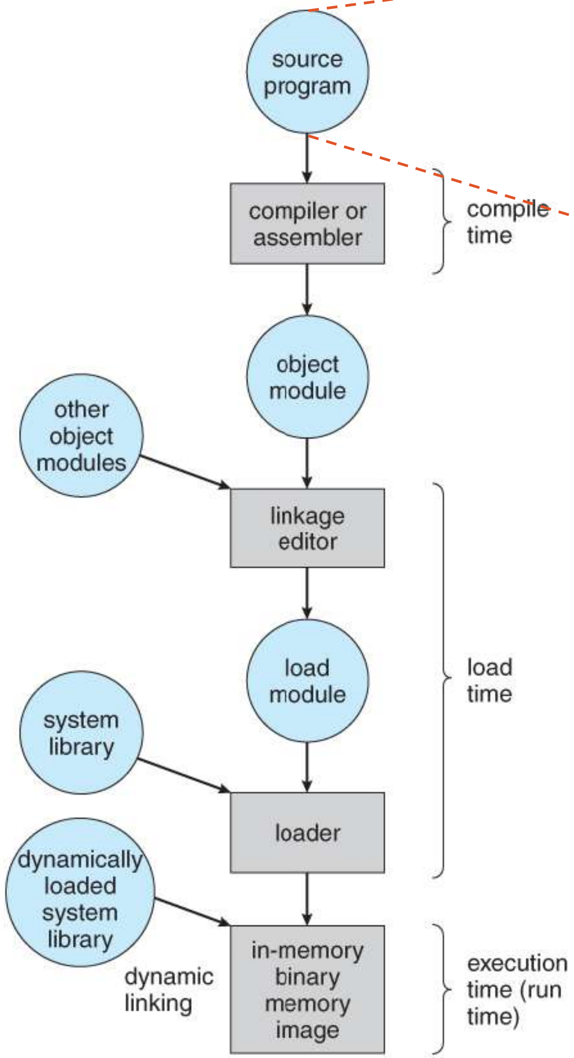
Address Binding: Compile Time

- 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!

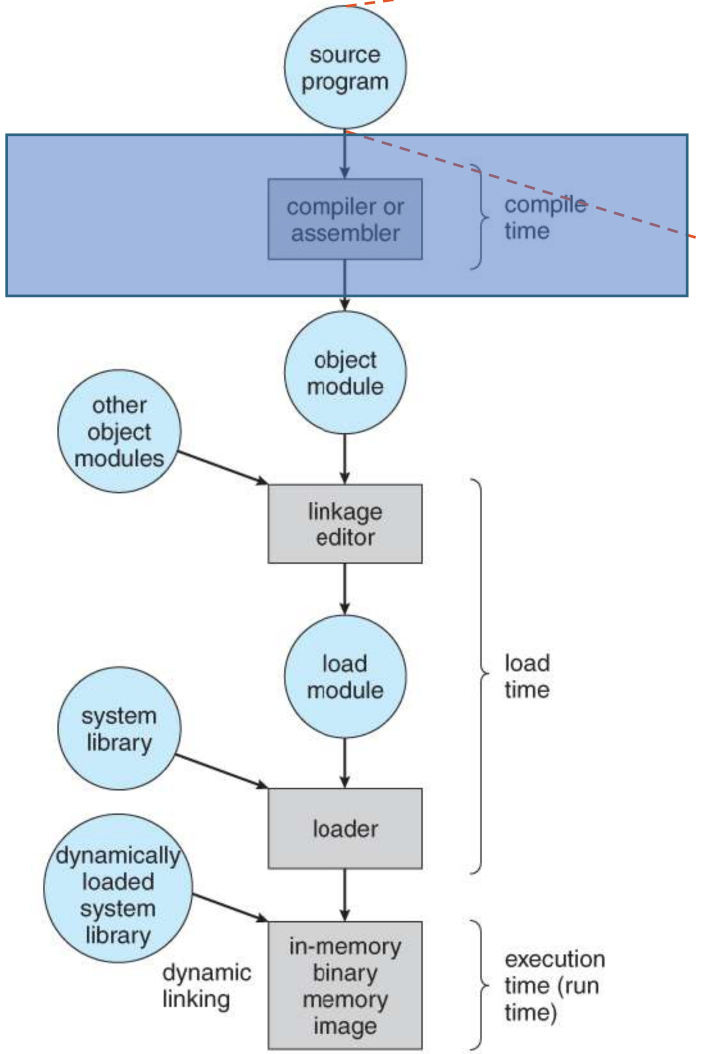
Address Binding: Compile Time




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

Address Binding: Compile Time

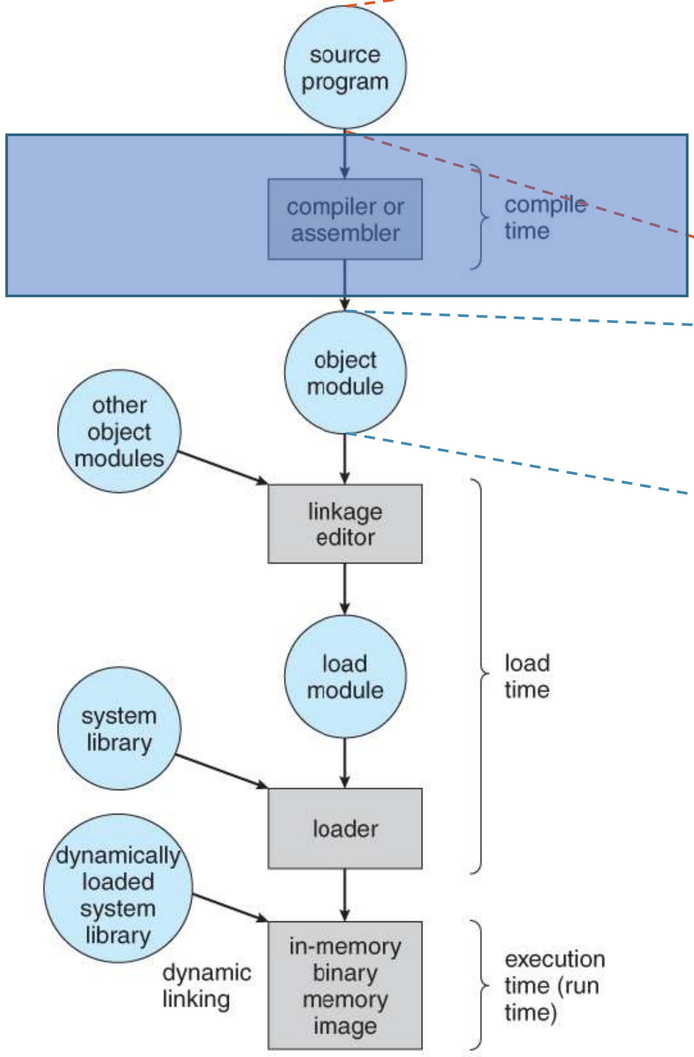
starting physical address known by the compiler (e.g., `k = 0`)




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

Address Binding: Compile Time

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

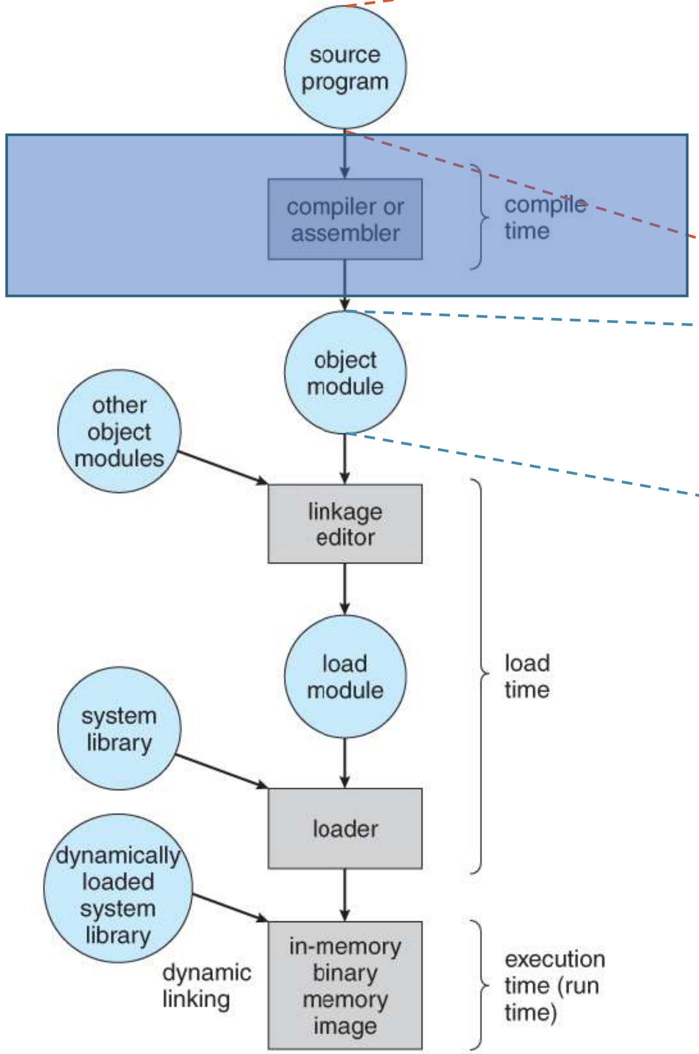



```
void foo() {
     int x = 42;
     …
     x = x + 5;
     …
}
```

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

Address Binding: Compile Time

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



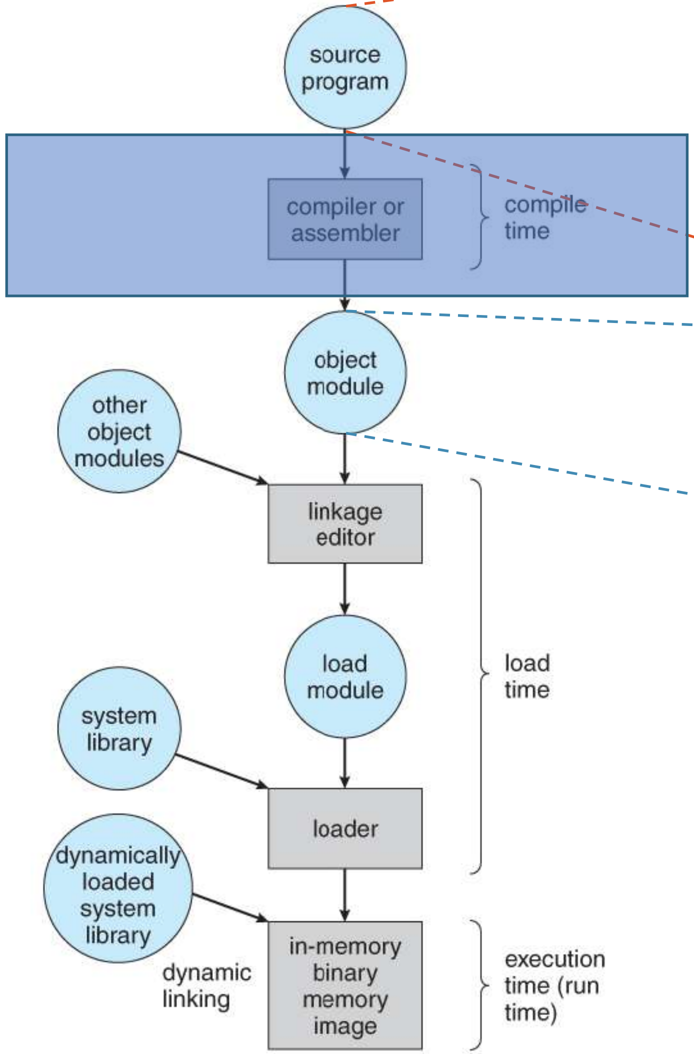

```
void foo() {
     int x = 42;
     …
     x = x + 5;
     …
}
```

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

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`)




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

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

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'`?

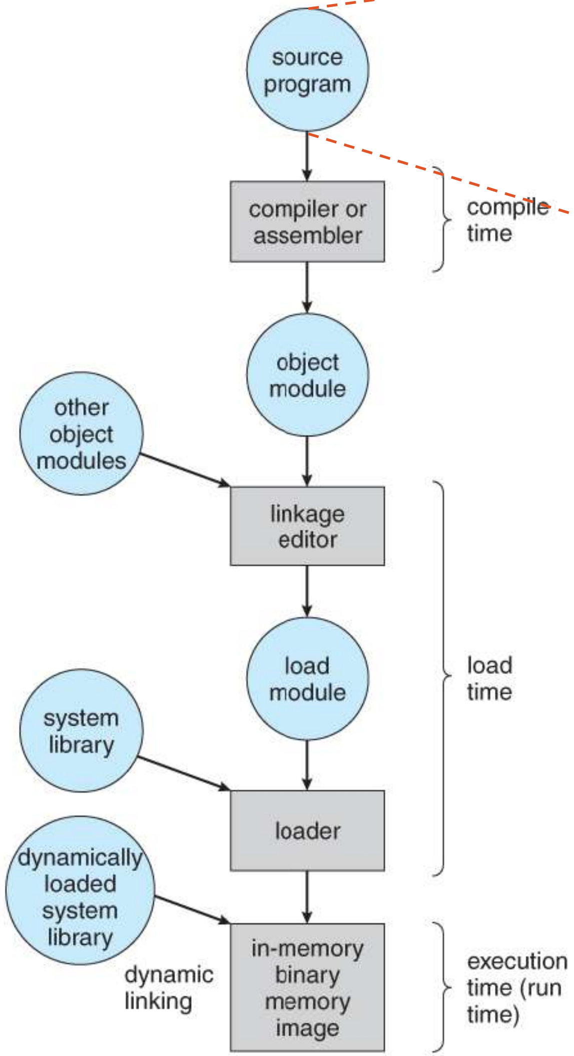
Address Binding: Load Time

- 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!

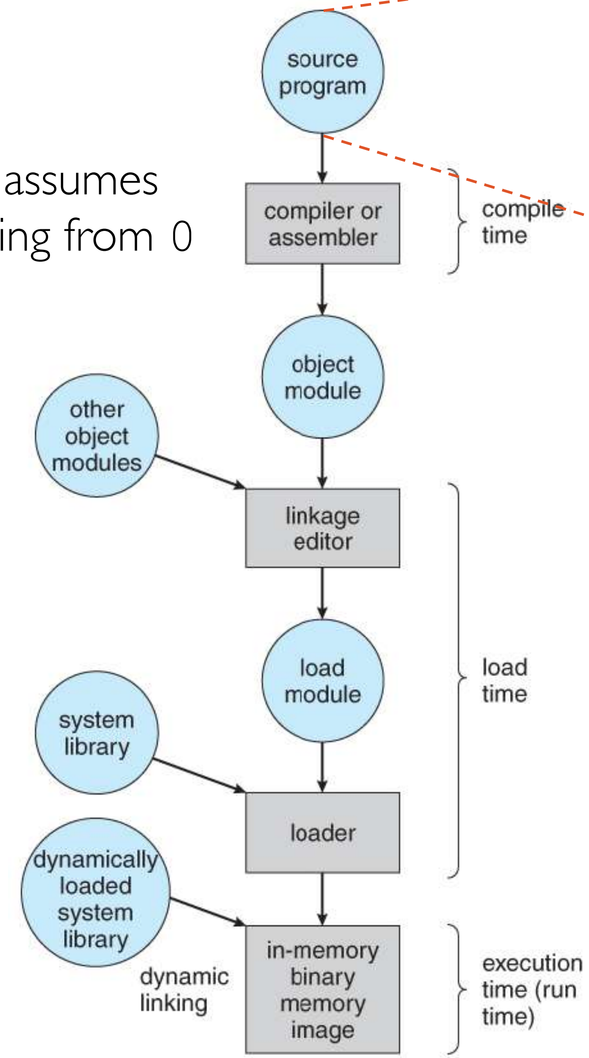
Address Binding: Load Time




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

Address Binding: Load Time

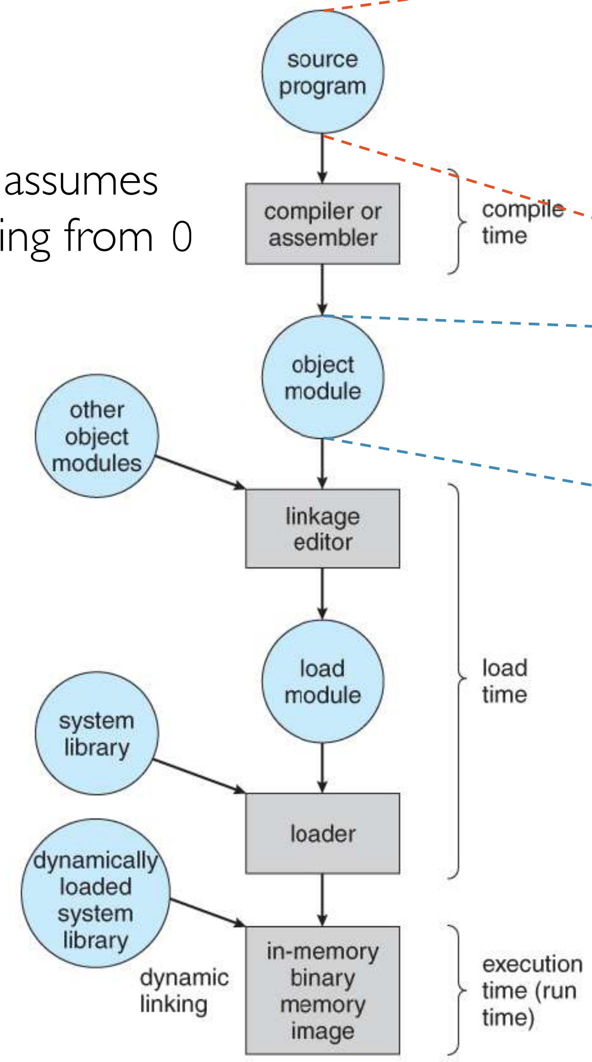
compiler still assumes addresses starting from 0




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

Address Binding: Load Time

compiler still assumes addresses starting from 0

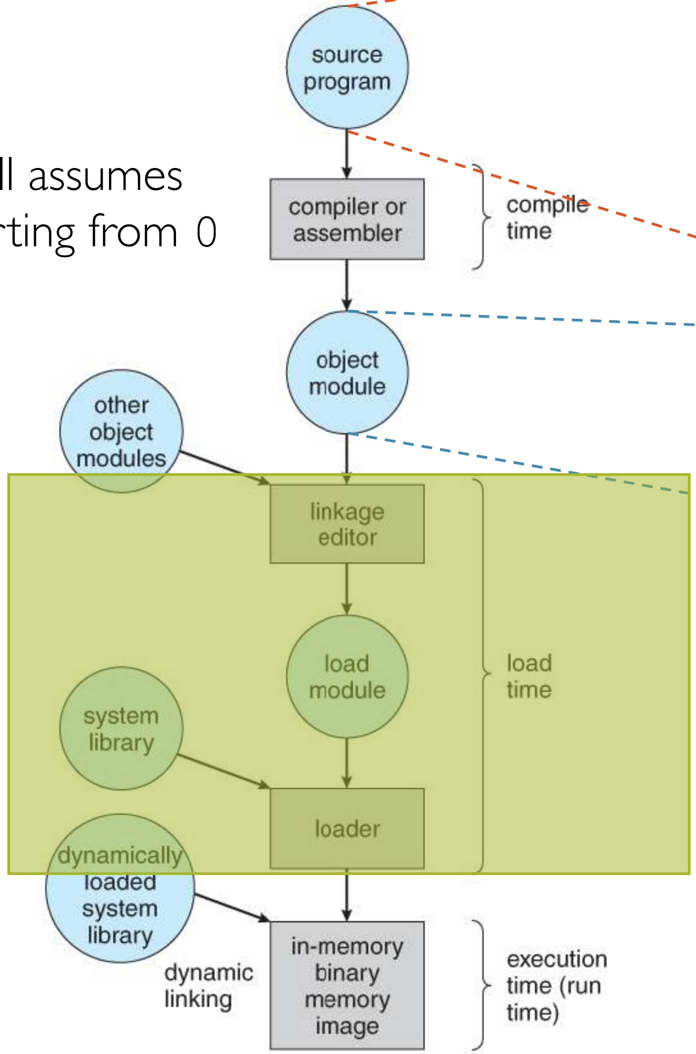



```
void foo() {
     int x = 42;
     …
     x = x + 5;
     …
}
```

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

Address Binding: Load Time

compiler still assumes addresses starting from 0



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

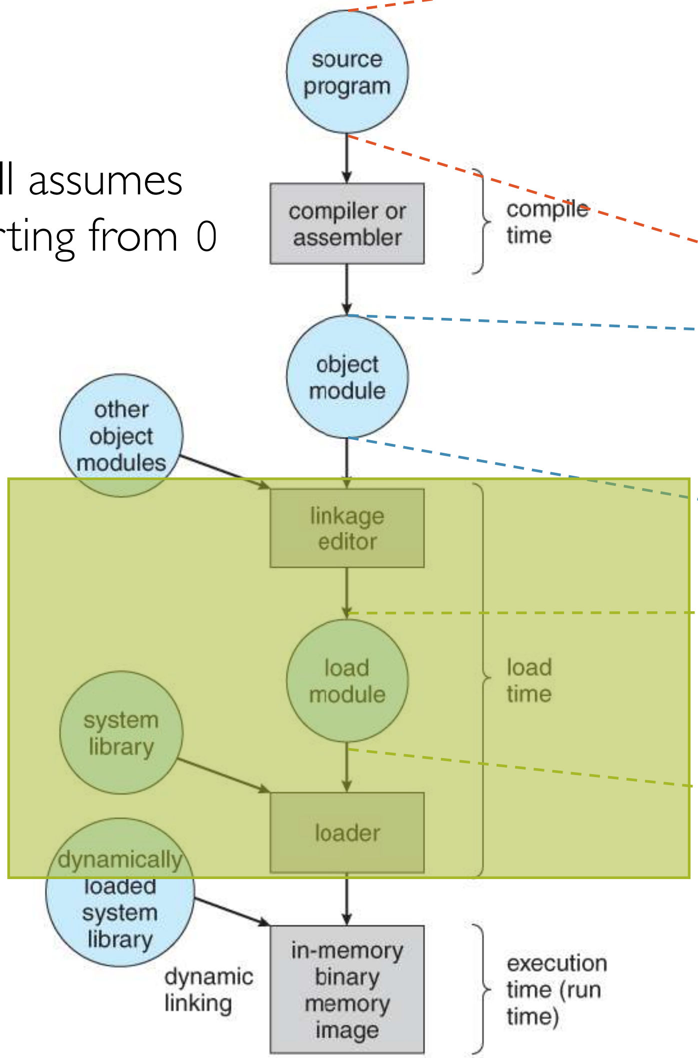
```
void foo() {
     int x = 42;
     …
     x = x + 5;
     …
}
```

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

Address Binding: Load Time

compiler still assumes addresses starting from 0

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




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

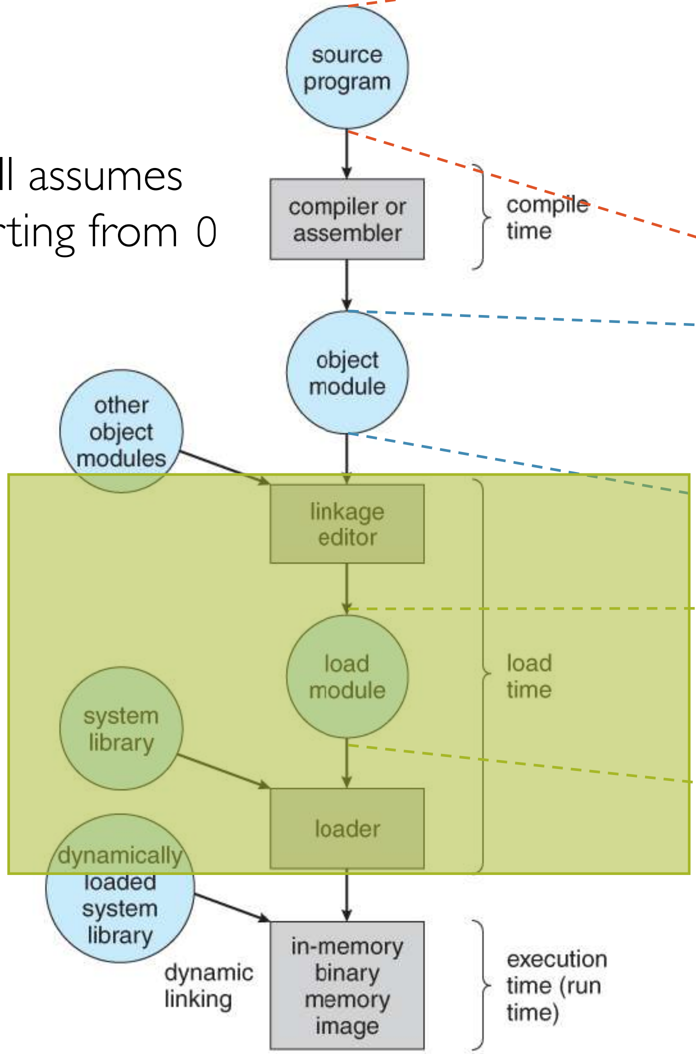
```
128: MOV %R1, [%R2]
136: ADD 5, %R1
144: MOV [%R2], %R1
```

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

Address Binding: Load Time

compiler still assumes addresses starting from 0

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




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

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

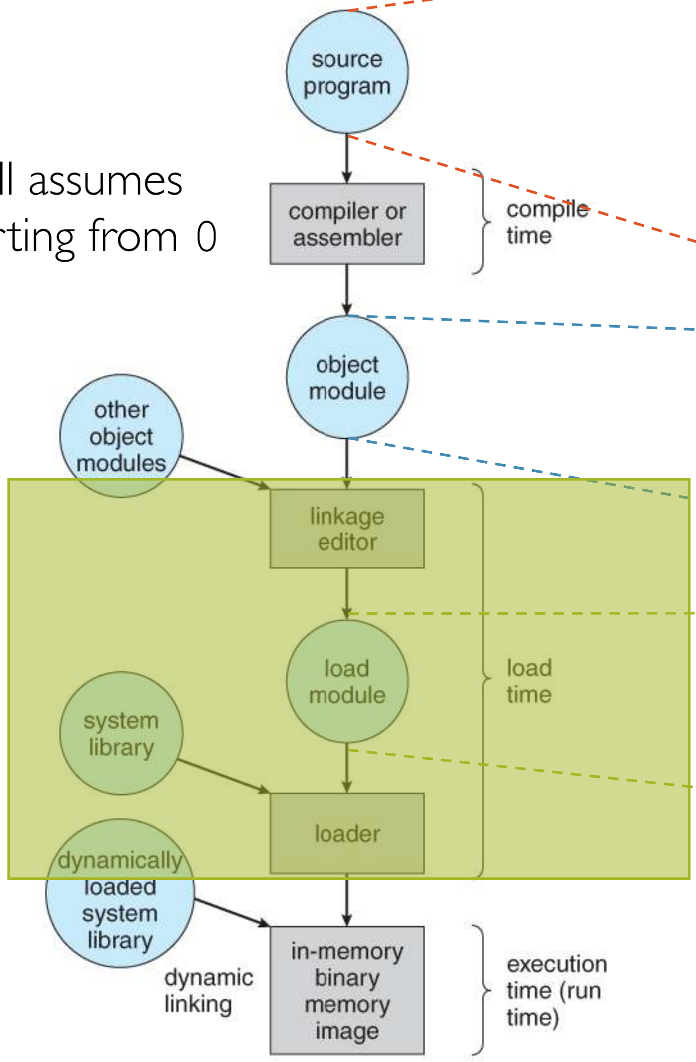
```
228: MOV %R1, [%R2+100]
236: ADD 5, %R1
244: MOV [%R2+100], %R1
```

The OS loader updates logical addresses *before* loading the executable

Address Binding: Load Time

compiler still assumes addresses starting from 0

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




```
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
```

The OS loader updates logical addresses *before* loading the executable

physical addresses == logical addresses

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

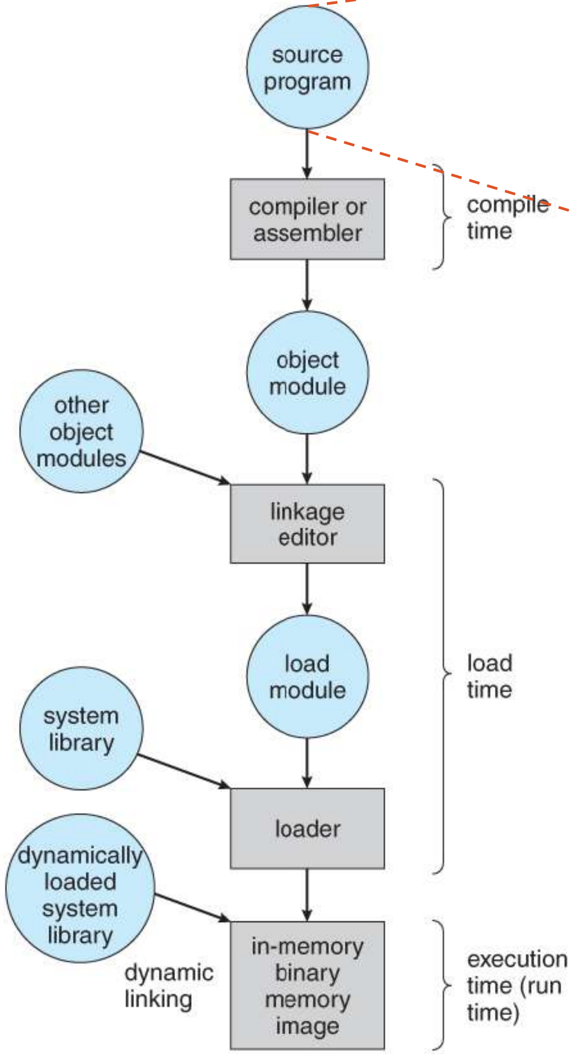
- If the program can be moved around in main memory during its execution
- The compiler generates so-called so-called (dynamically) relocatable code or virtual addresses
- The OS maps virtual addresses to physical memory locations using special HW support (MMU)
- physical address != logical/virtual address

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 != logical/virtual address

Most flexible solution implemented by the majority of modern OSs

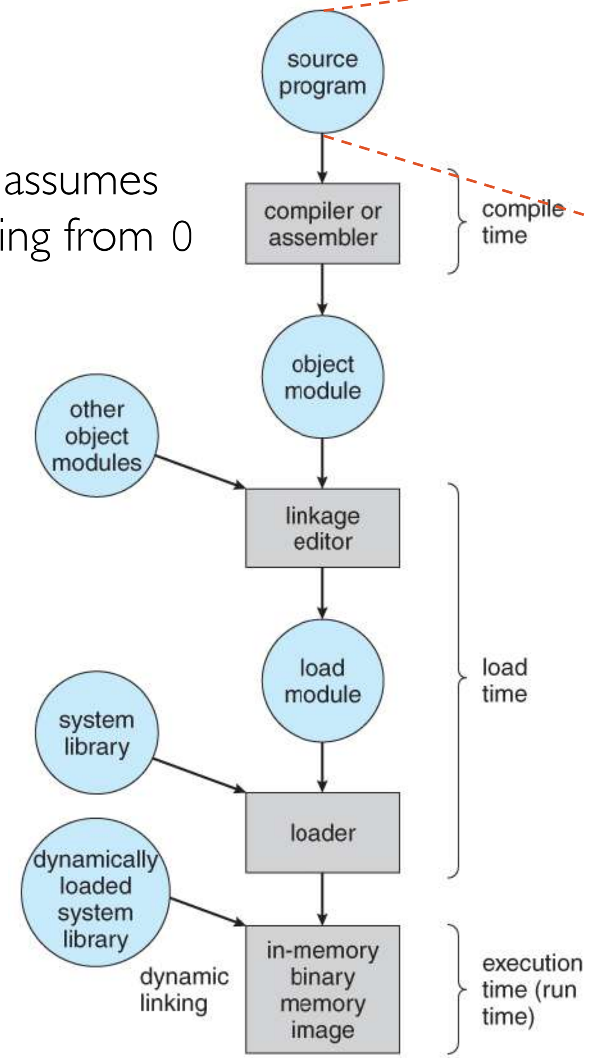
Address Binding: Execution Time




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

Address Binding: Execution Time

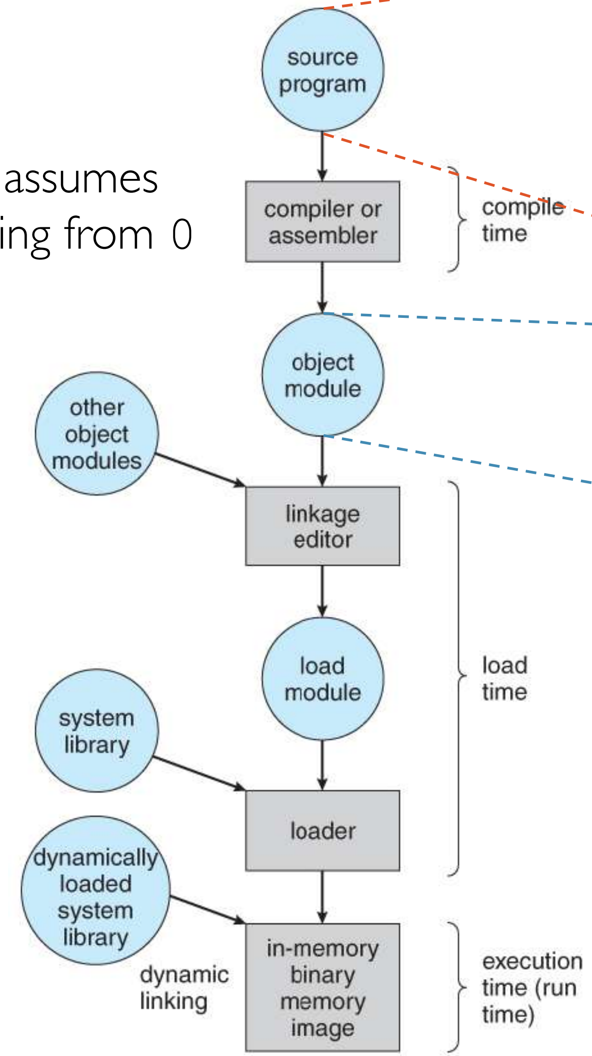
compiler still assumes addresses starting from 0




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

Address Binding: Execution Time

compiler still assumes addresses starting from 0

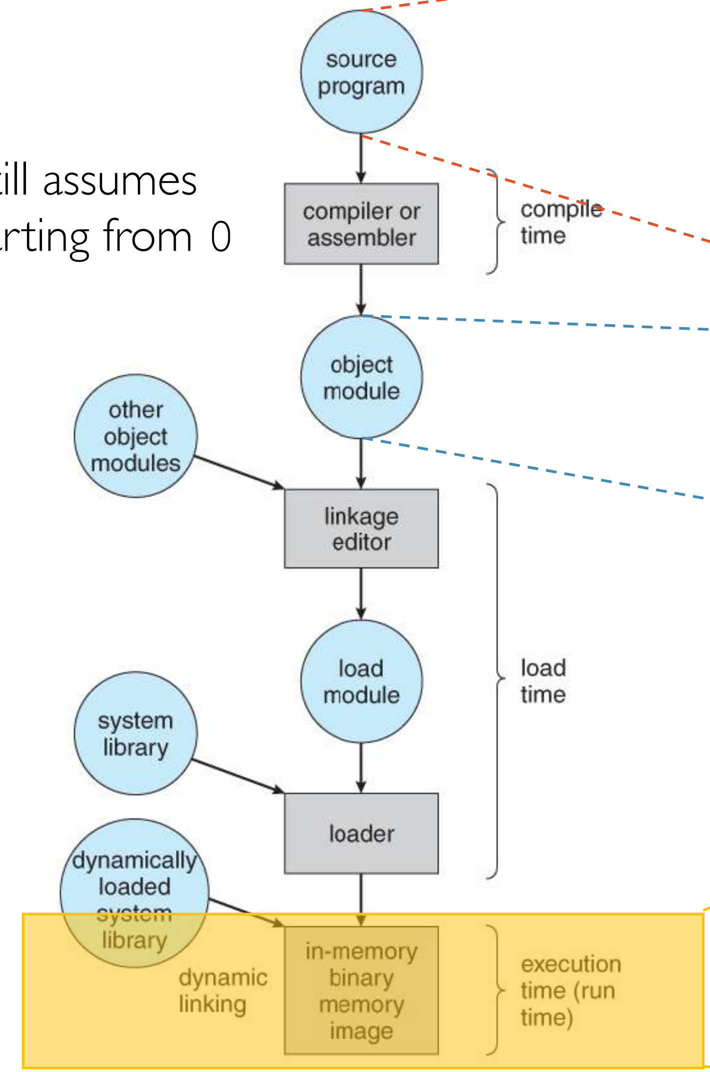



```
void foo() {
    int x = 42;
    …
    x = x + 5;
    …
}
```

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

Address Binding: Execution Time

compiler still assumes addresses starting from 0



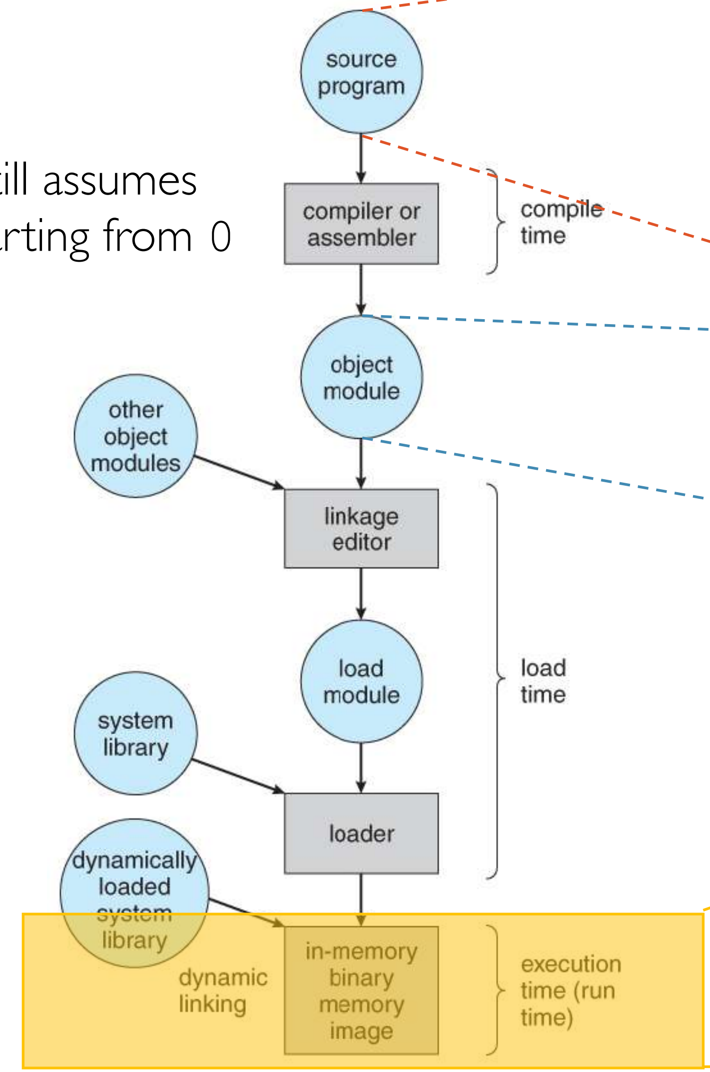

```
void foo() {
    int x = 42;
    …
    x = x + 5;
    …
}
```

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

MMU

Address Binding: Execution 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
```

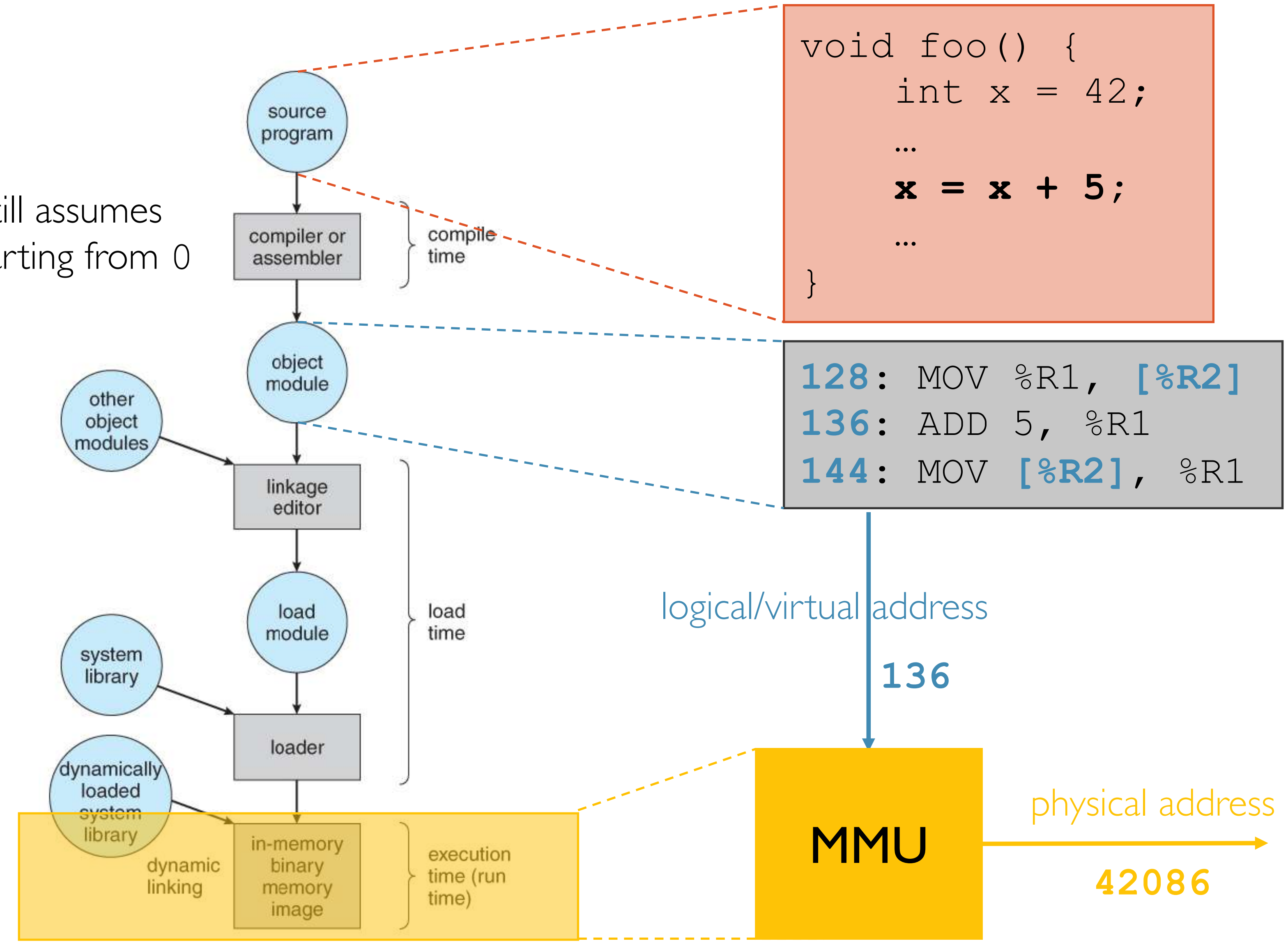
logical/virtual address

136

MMU

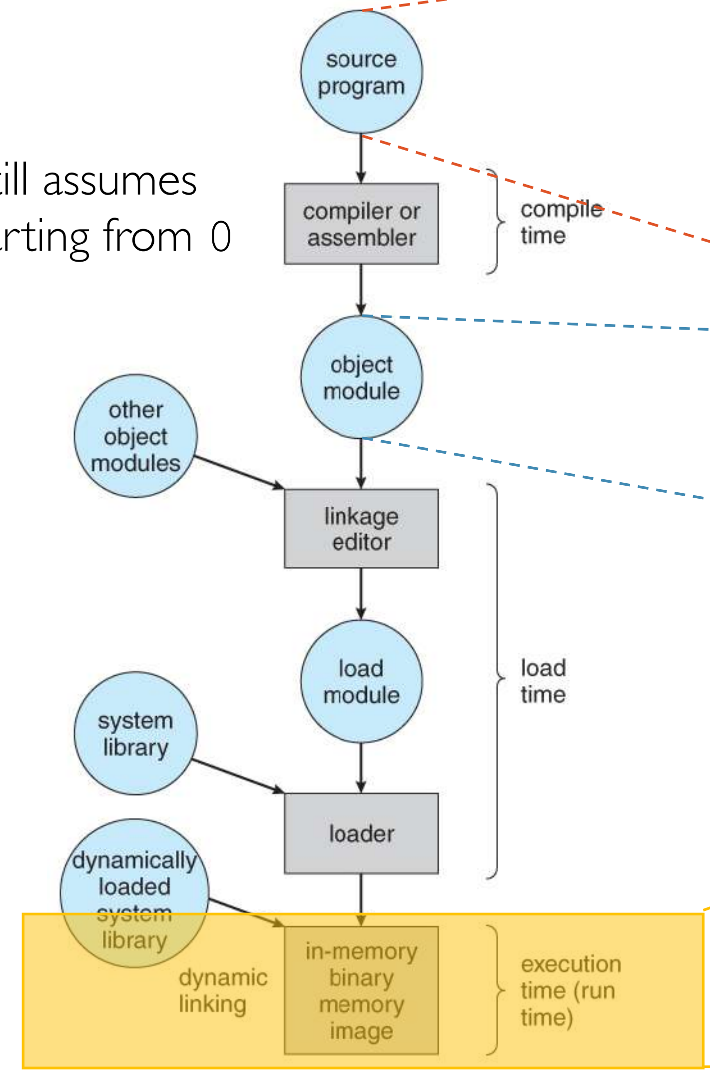
Address Binding: Execution Time

compiler still assumes addresses starting from 0



Address Binding: Execution 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
```

logical/virtual address

136

MMU

physical address

42086

Every logical address referenced by the CPU is translated by HW at run time

physical addresses != logical addresses

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

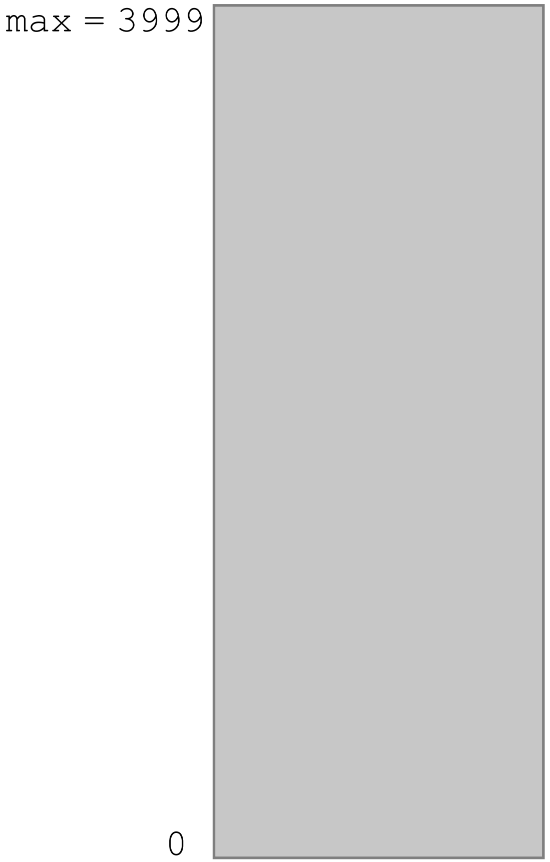
- Easiest memory management requirements
- The OS gets a fixed part of physical memory (e.g., highest memory addresses in MS-DOS)
- Run one process at a time, always loaded starting at physical address `k = 0` up to maximum address (`memory_size - os_size - 1`)
- Each process executes in a contiguous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

Manage Uniprogramming Memory

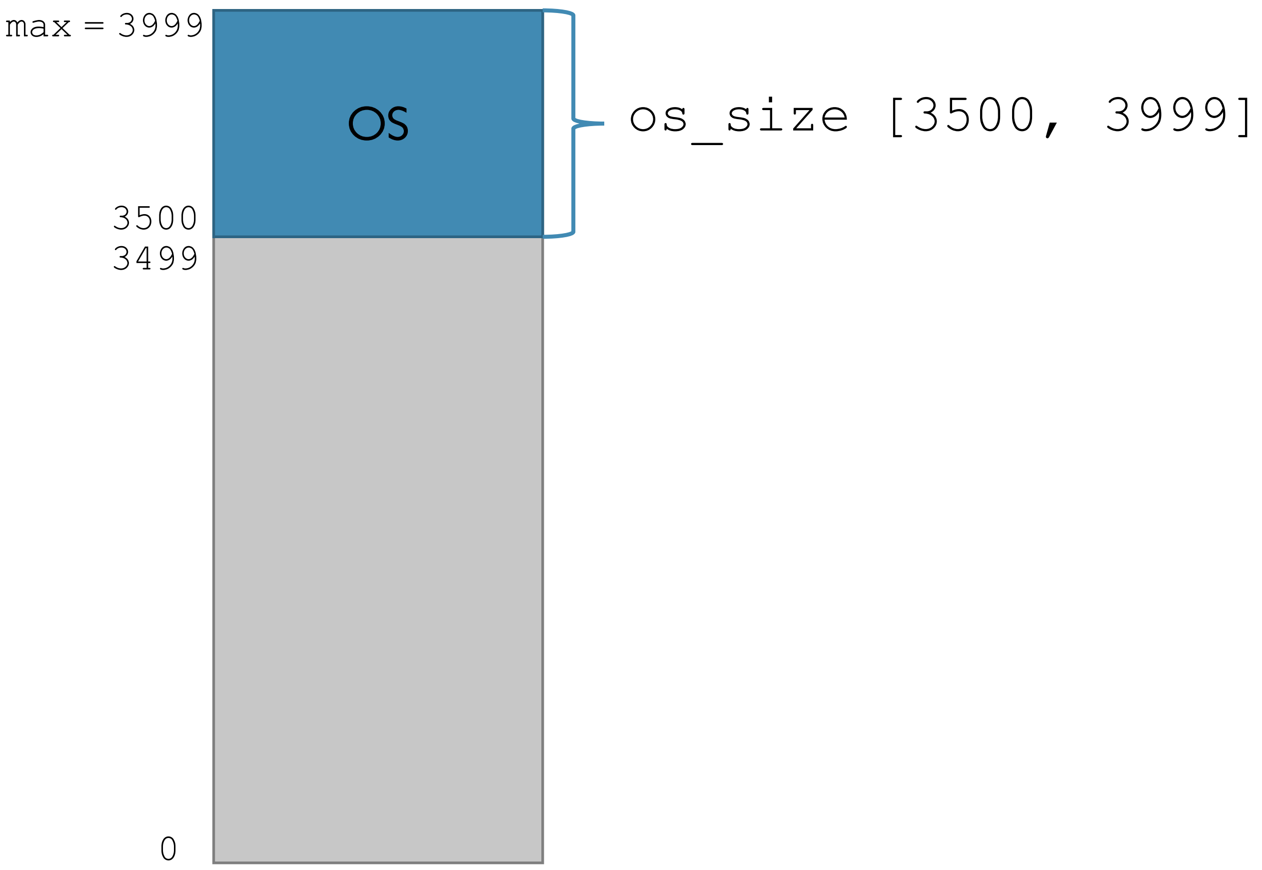
`memory_size = 4000B = 4kB`



Manage Uniprogramming Memory

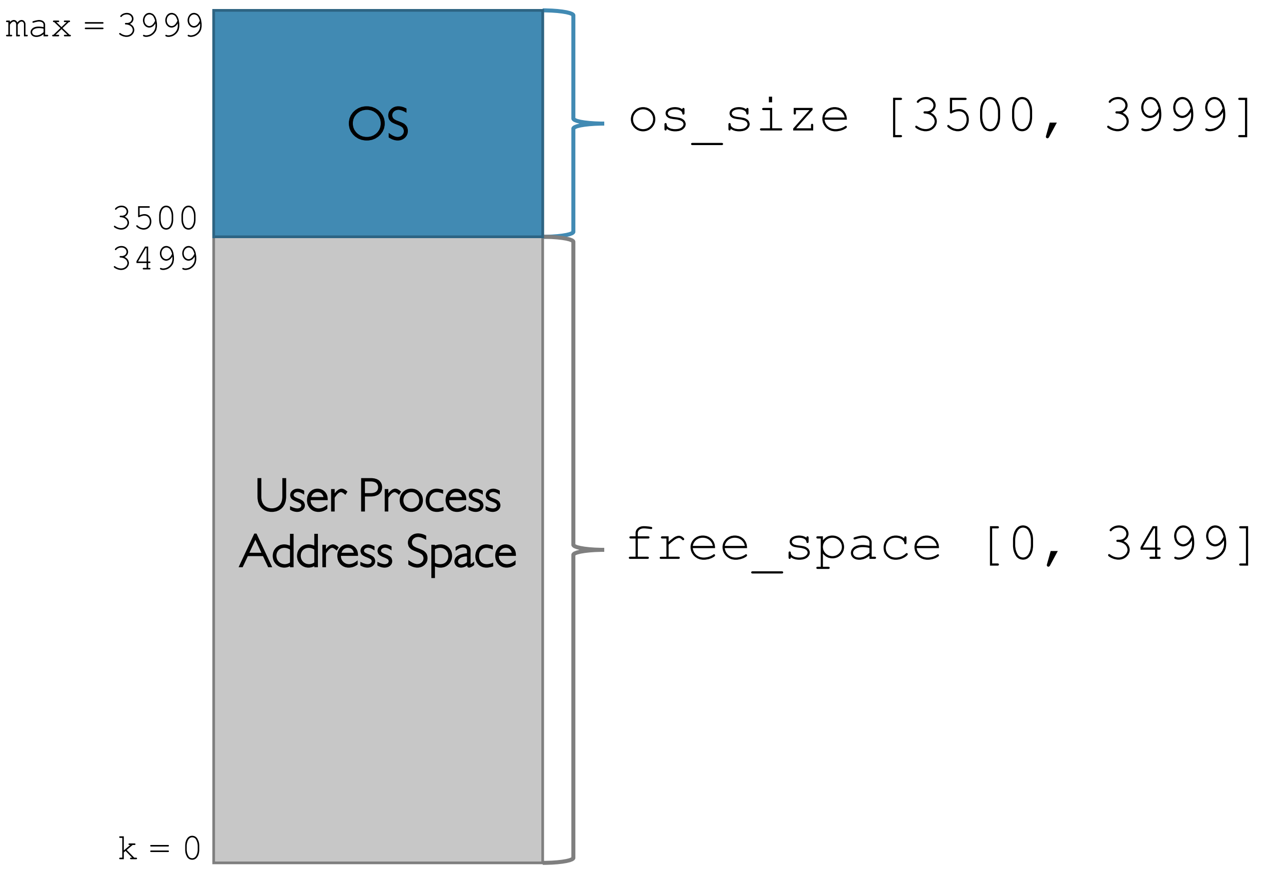
`memory_size = 4000B = 4kB`

`os_size = 500B`



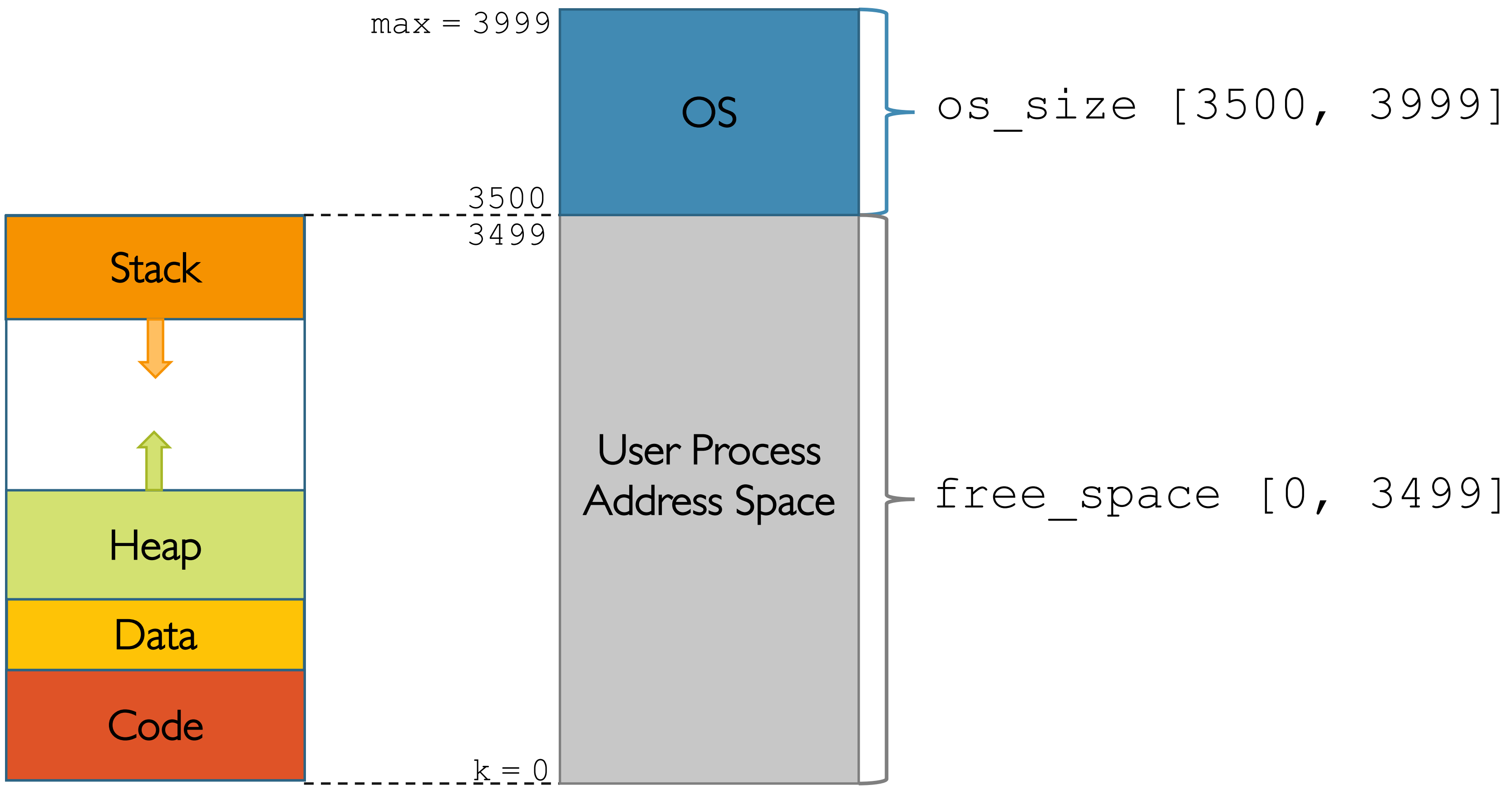
Manage Uniprogramming Memory

`memory_size = 4000B = 4kB` | `os_size = 500B` | `k = 0` | `free_space = 3.5kB`

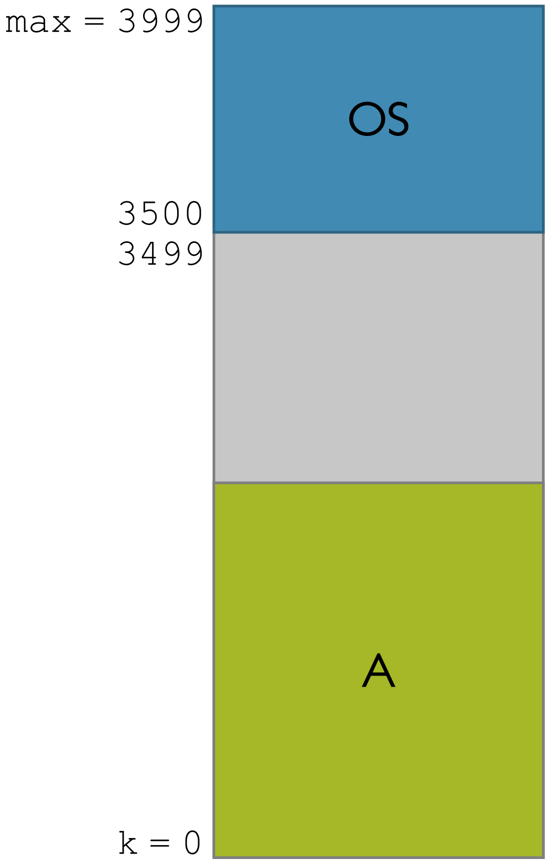


Manage Uniprogramming Memory

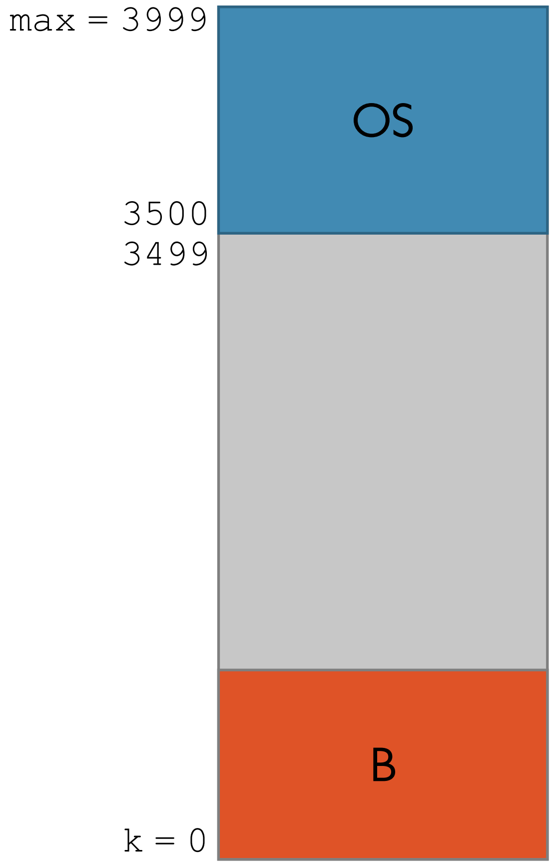
memory_size = 4000B = 4kB	os_size = 500B	k = 0	free_space = 3.5kB



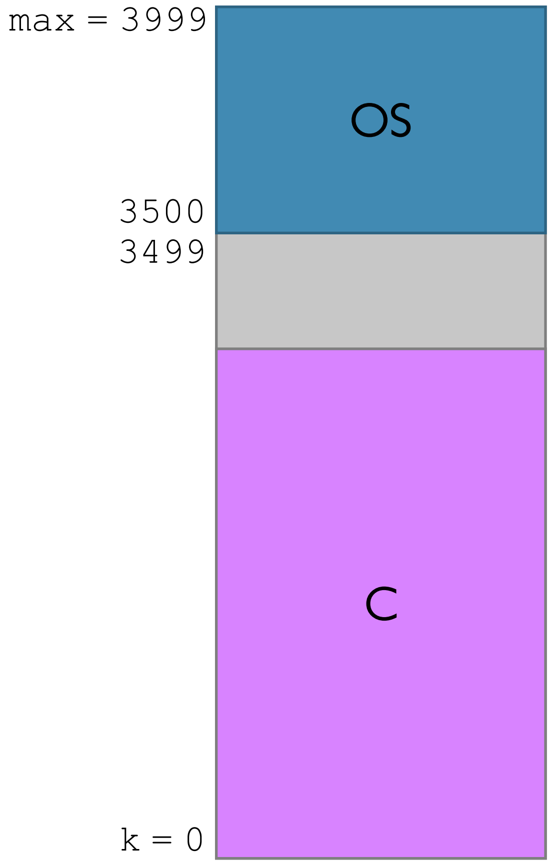
Manage Uniprogramming Memory



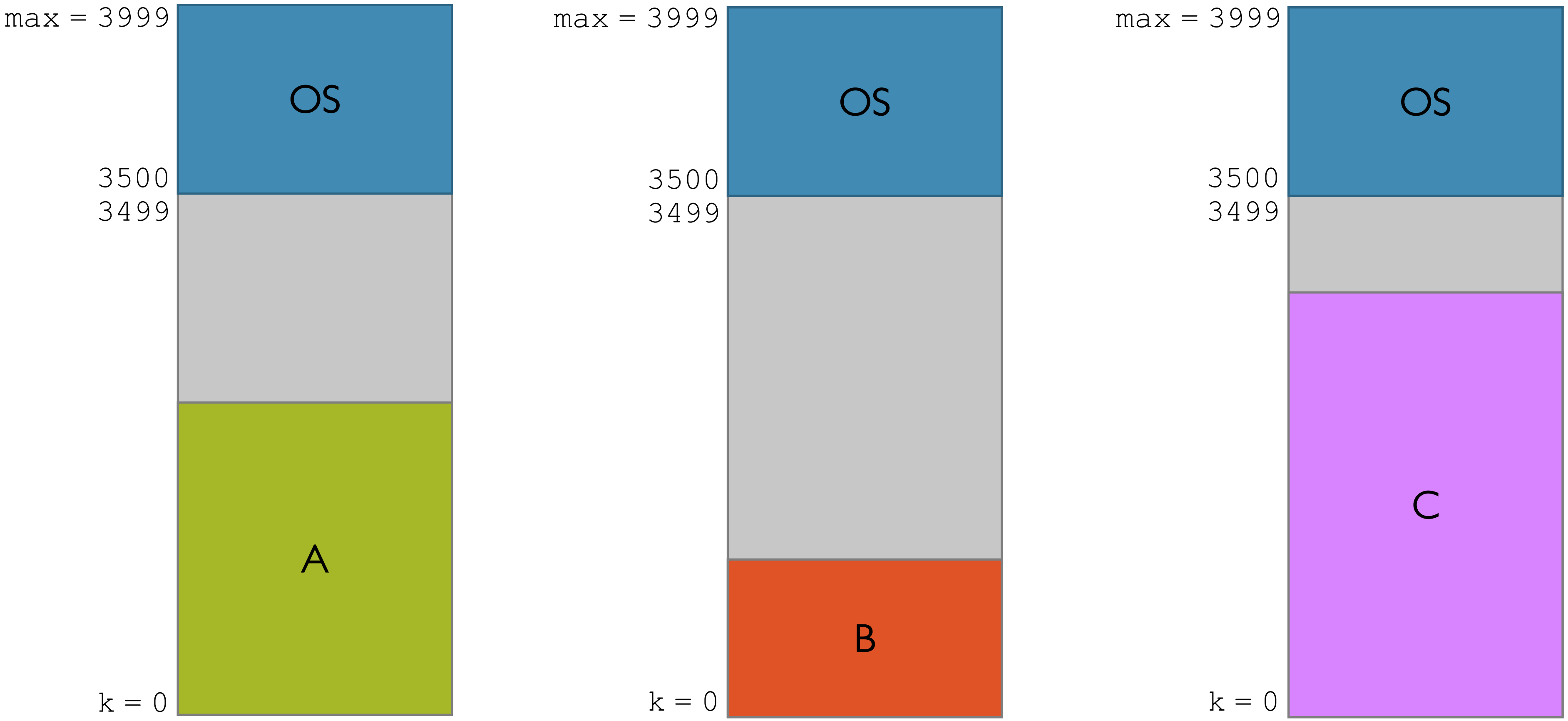
Manage Uniprogramming Memory



Manage Uniprogramming Memory



Manage Uniprogramming Memory



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

Manage Multiprogramming Memory: Goals (1)

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

Manage Multiprogramming Memory: Goals (1)

- 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)

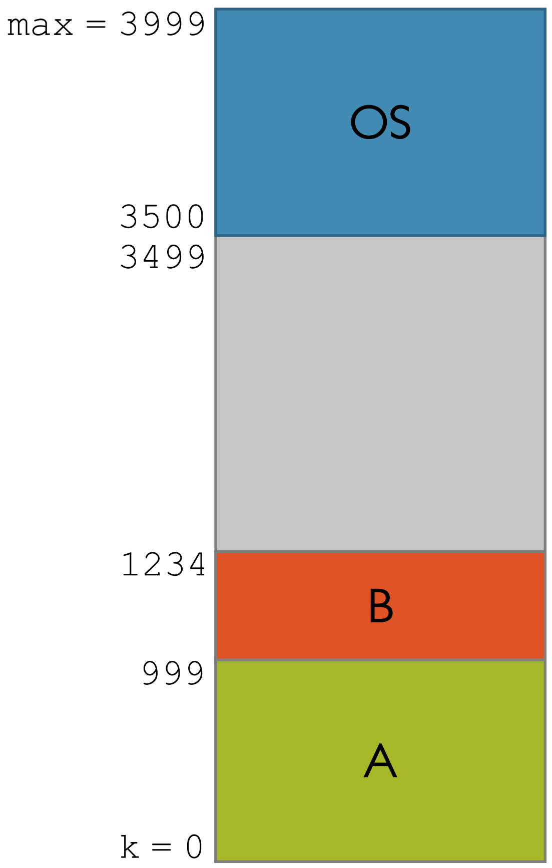
Static Relocation

- 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

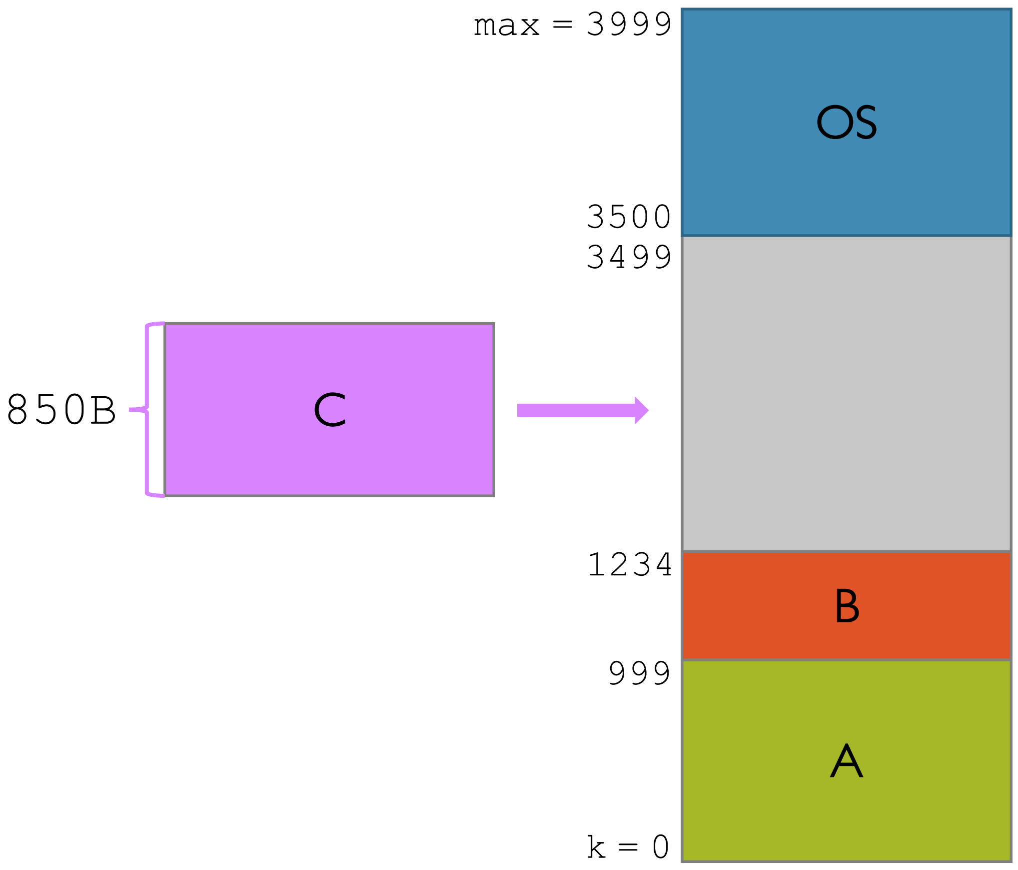
Static Relocation

- 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

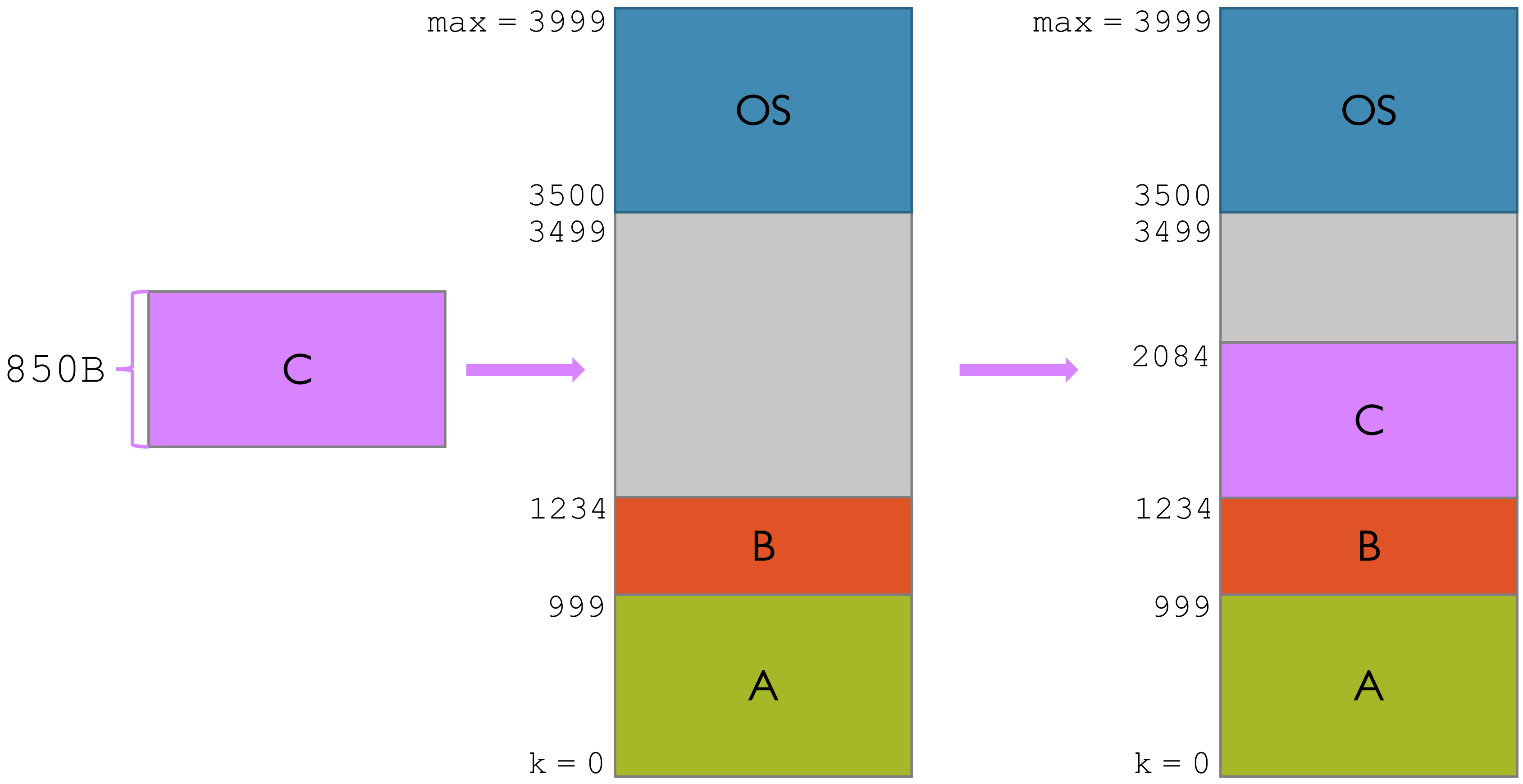
Static Relocation



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

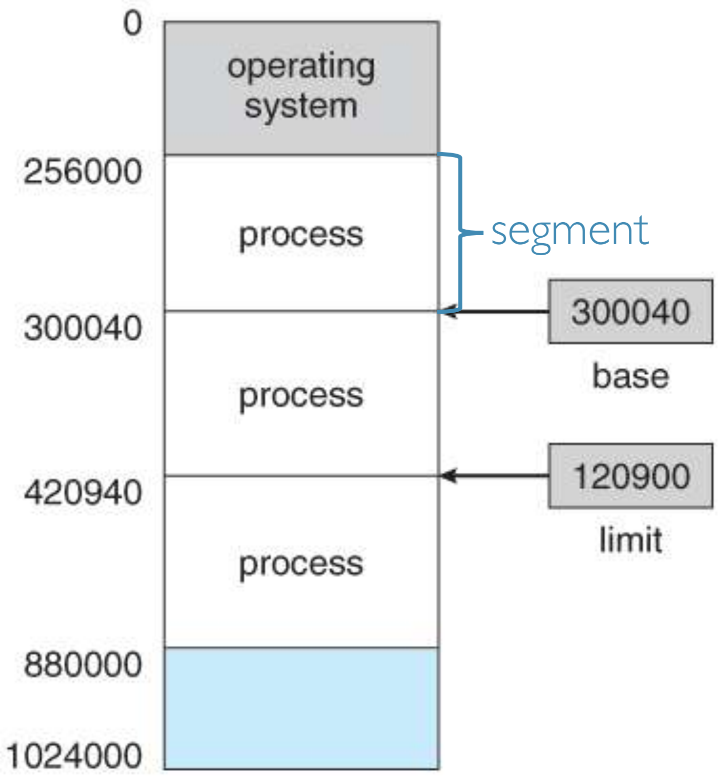
HW Support for Dynamic Relocation

- 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

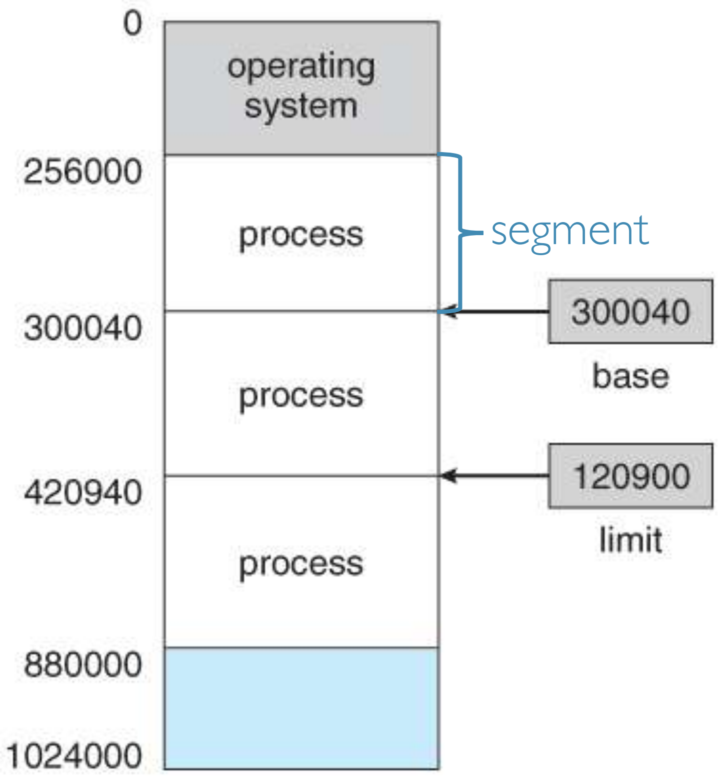
- MMU contains at least 2 registers:
  - base → start physical memory location of address space
  - limit → size limit of address space
- CPU supports at least 2 operating modes:
  - privileged (kernel) mode when the OS is running
    - after issuing any trap (system call, interruption, or exception)
    - 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

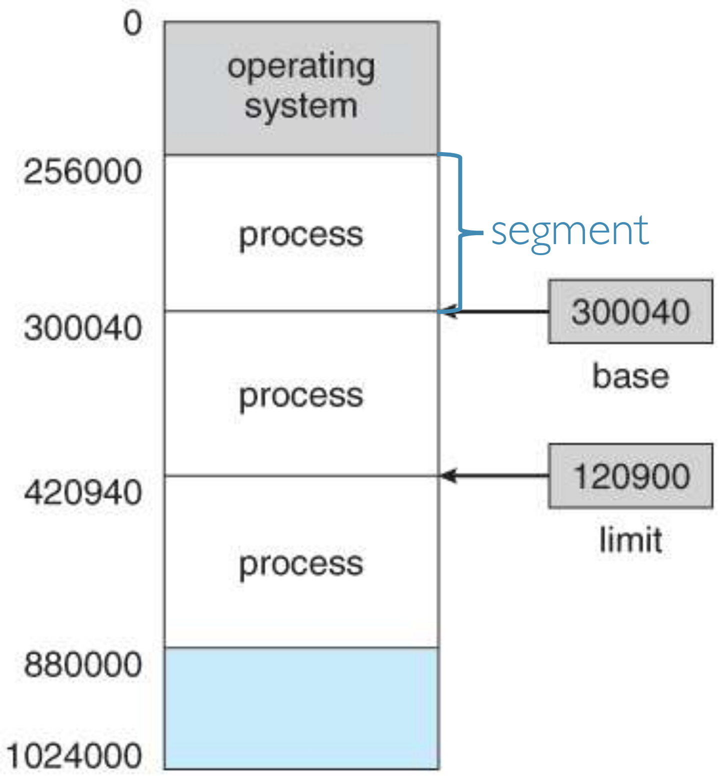
Base and Limit Registers: Idea



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

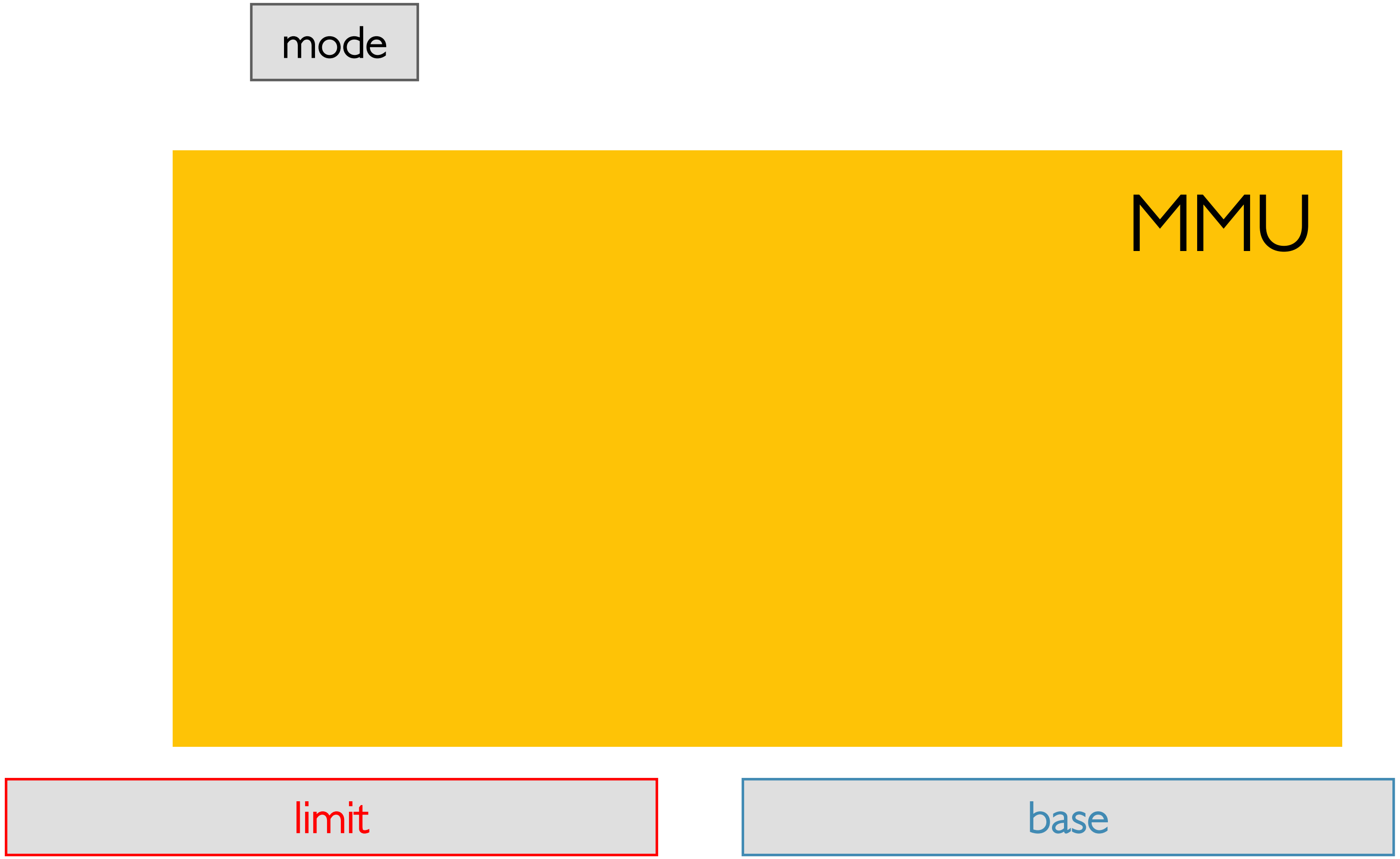
Protection implemented using two MMU registers: base and limit

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

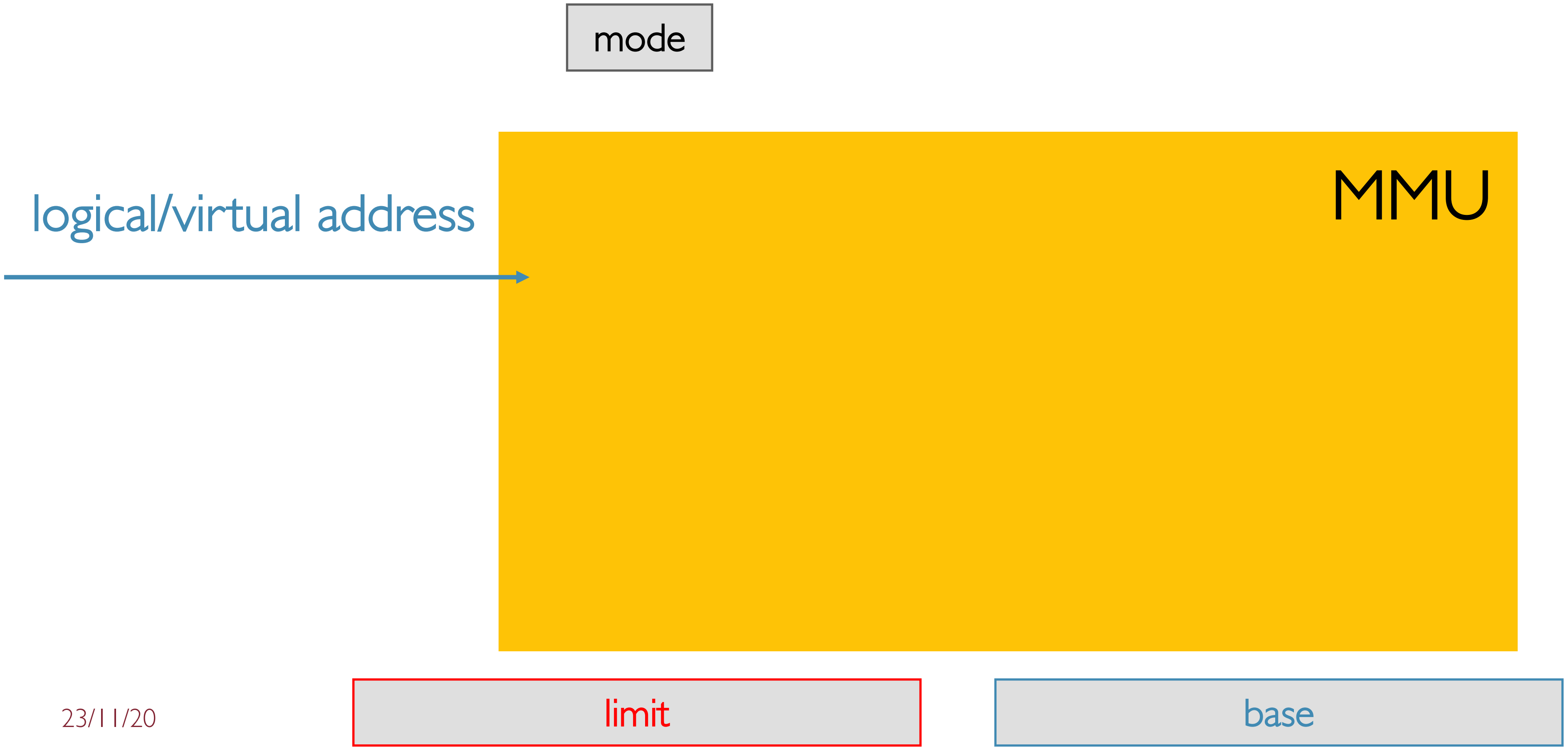
Implementing Dynamic Relocation

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



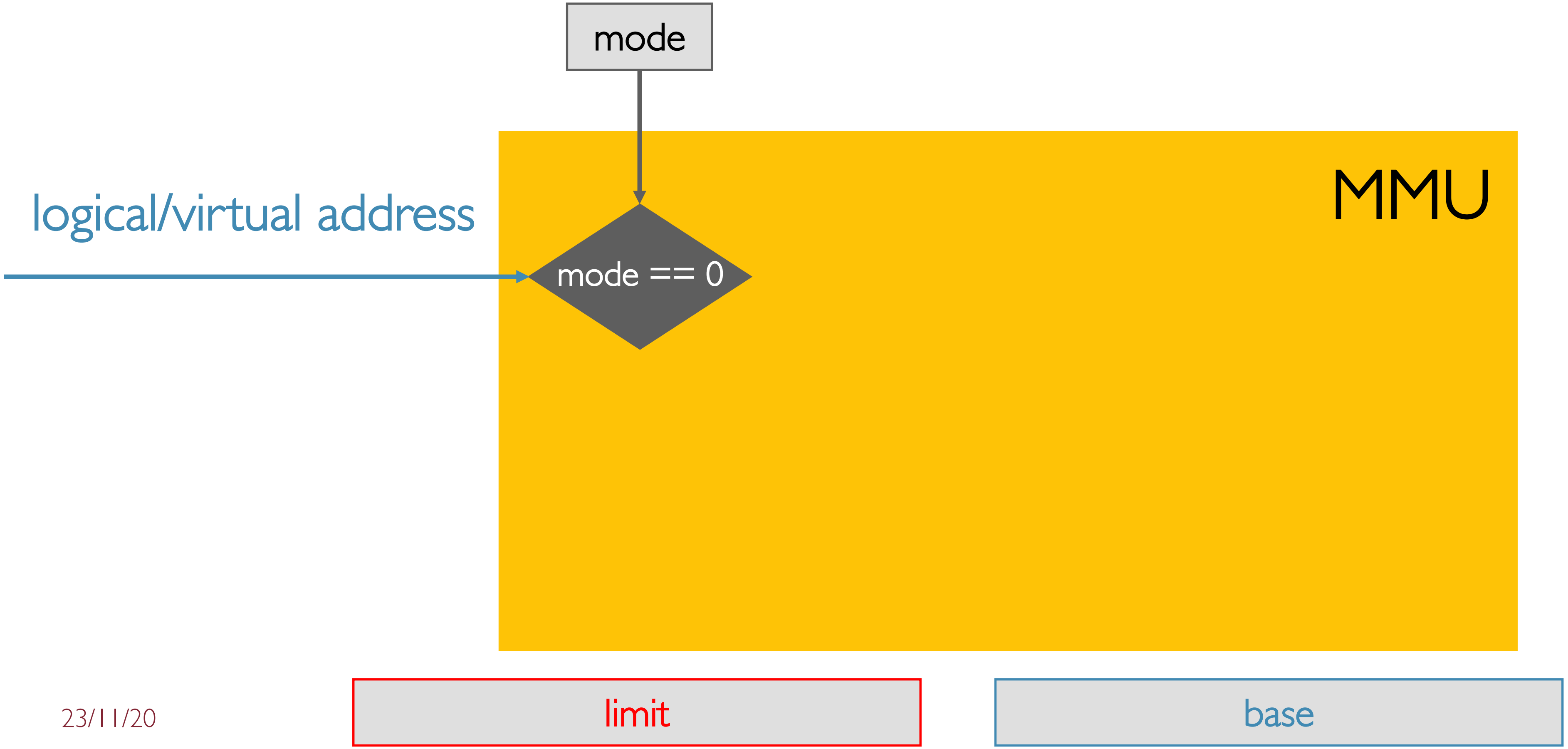
Implementing Dynamic Relocation

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



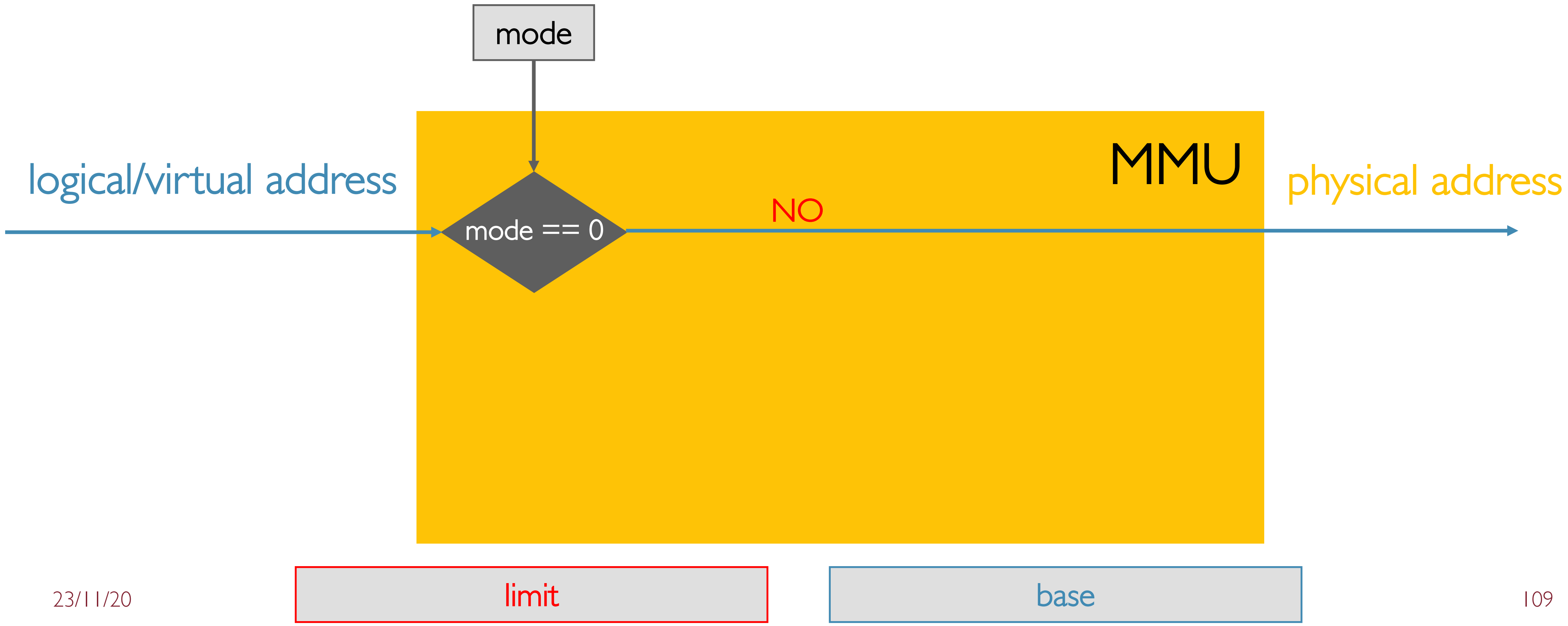
Implementing Dynamic Relocation

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



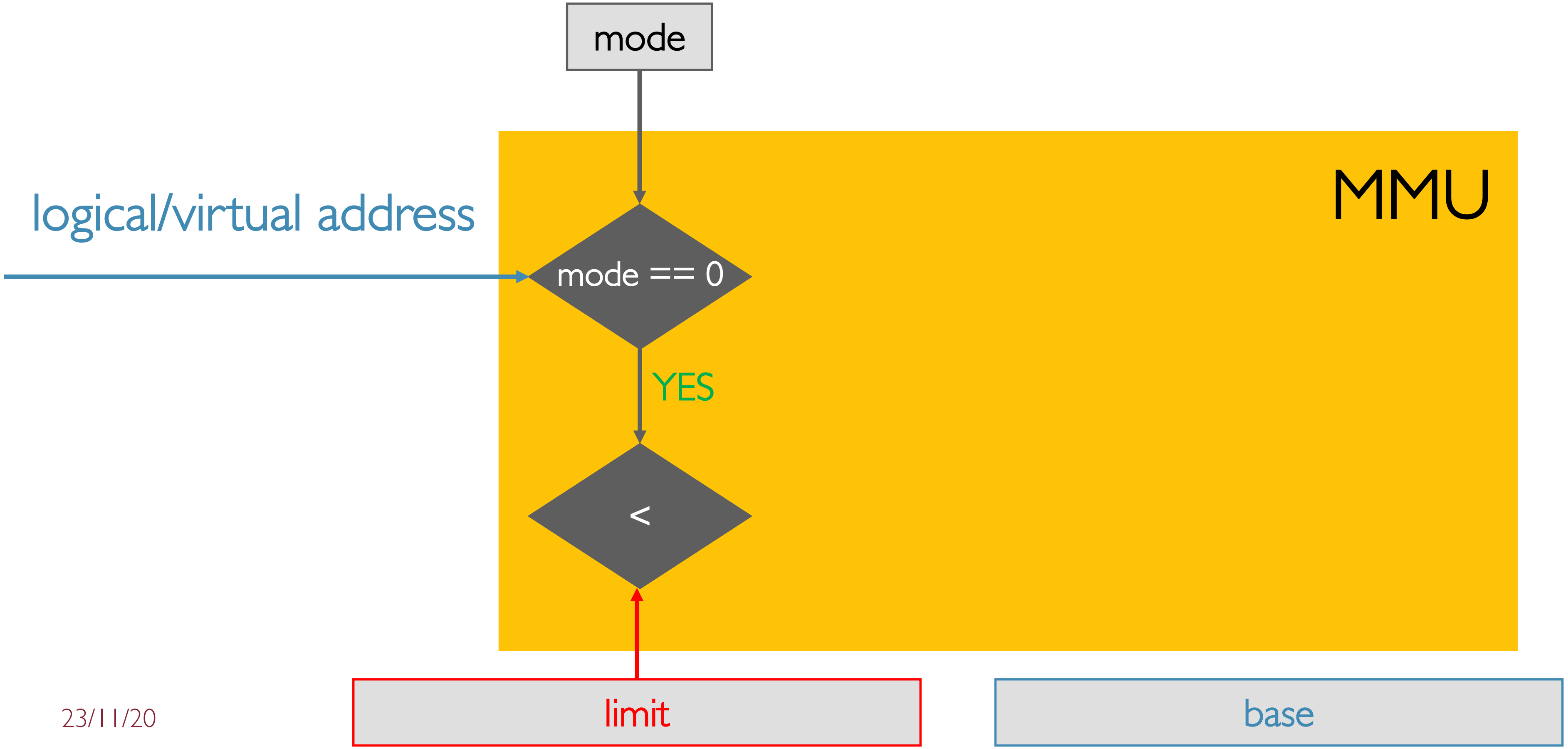
Implementing Dynamic Relocation

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



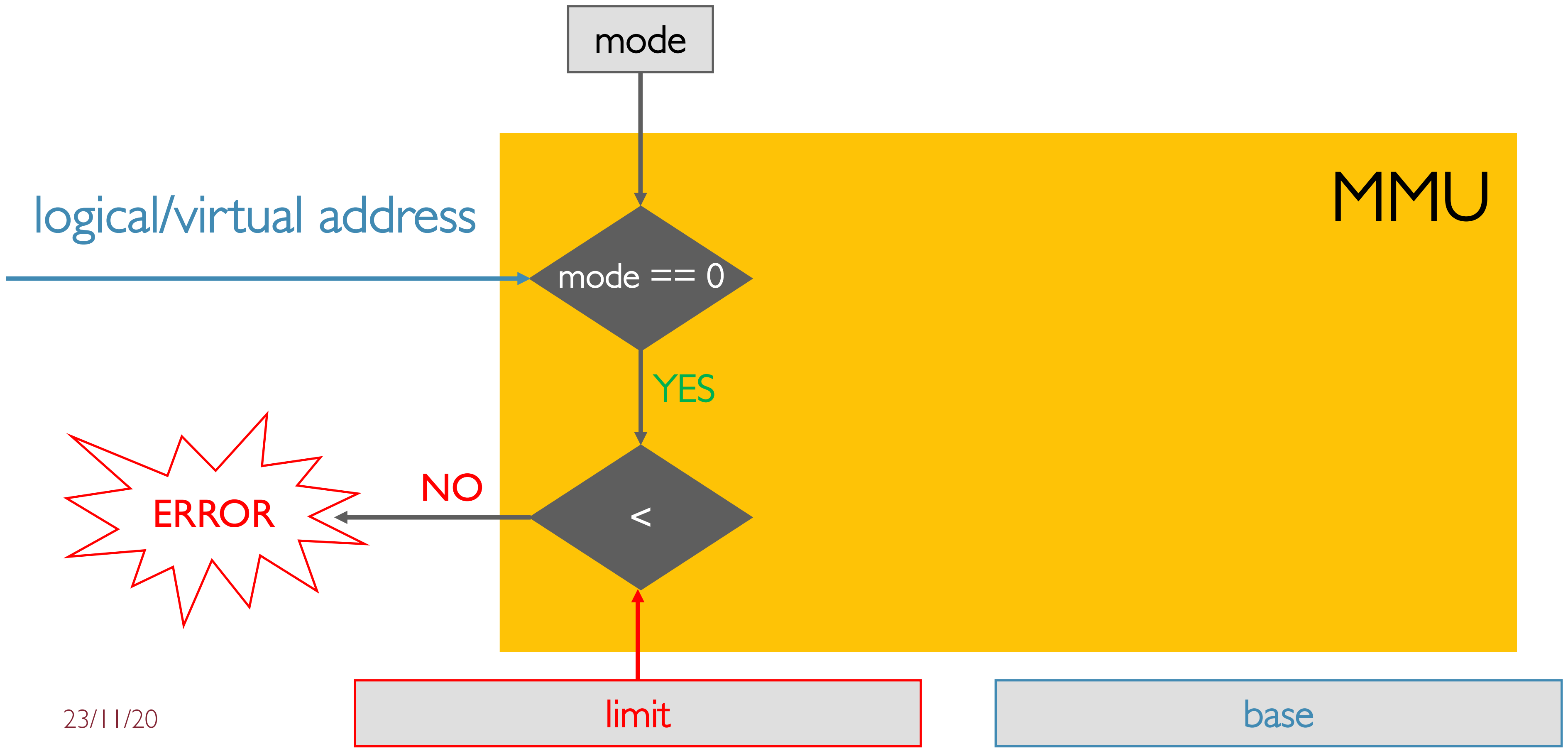
Implementing Dynamic Relocation

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



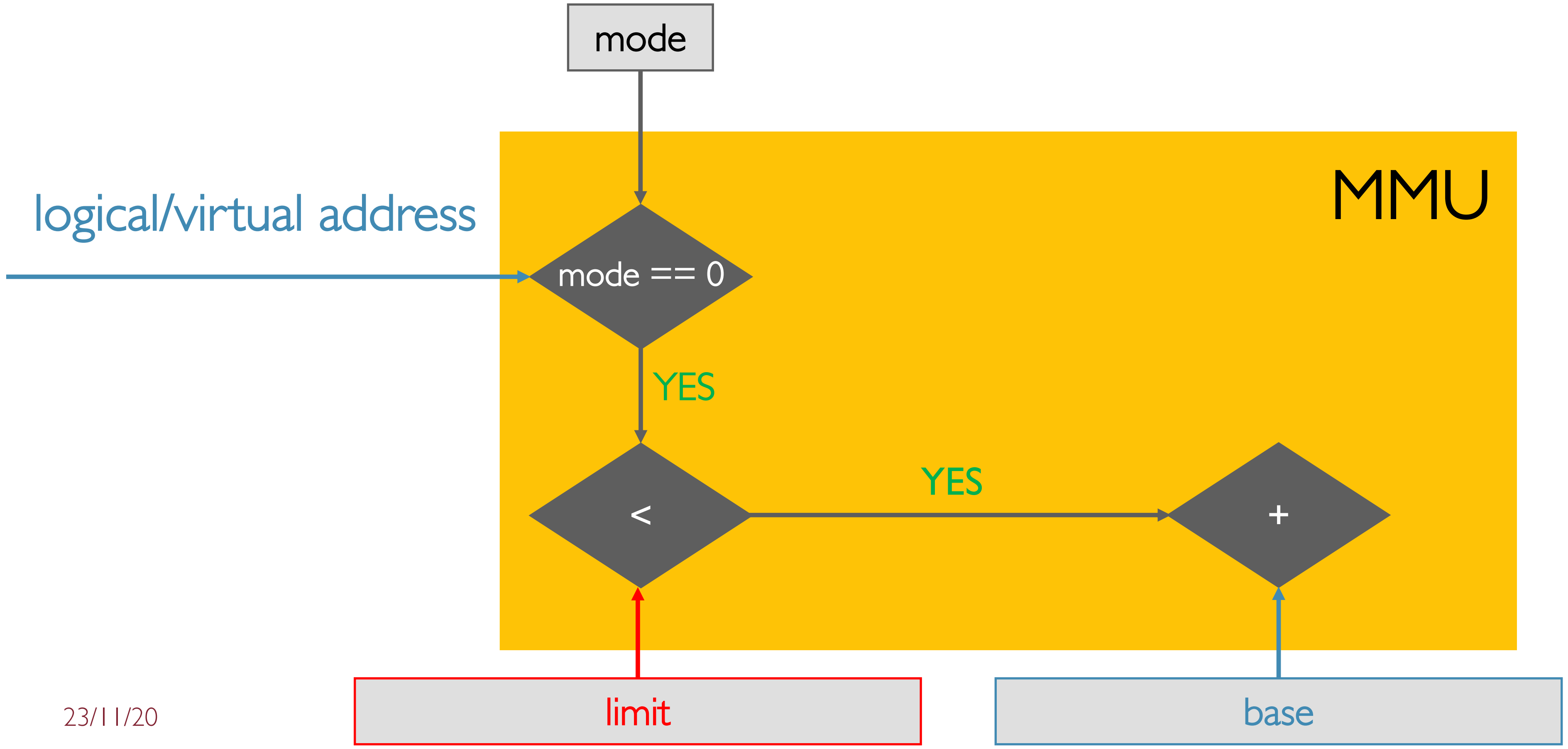
Implementing Dynamic Relocation

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



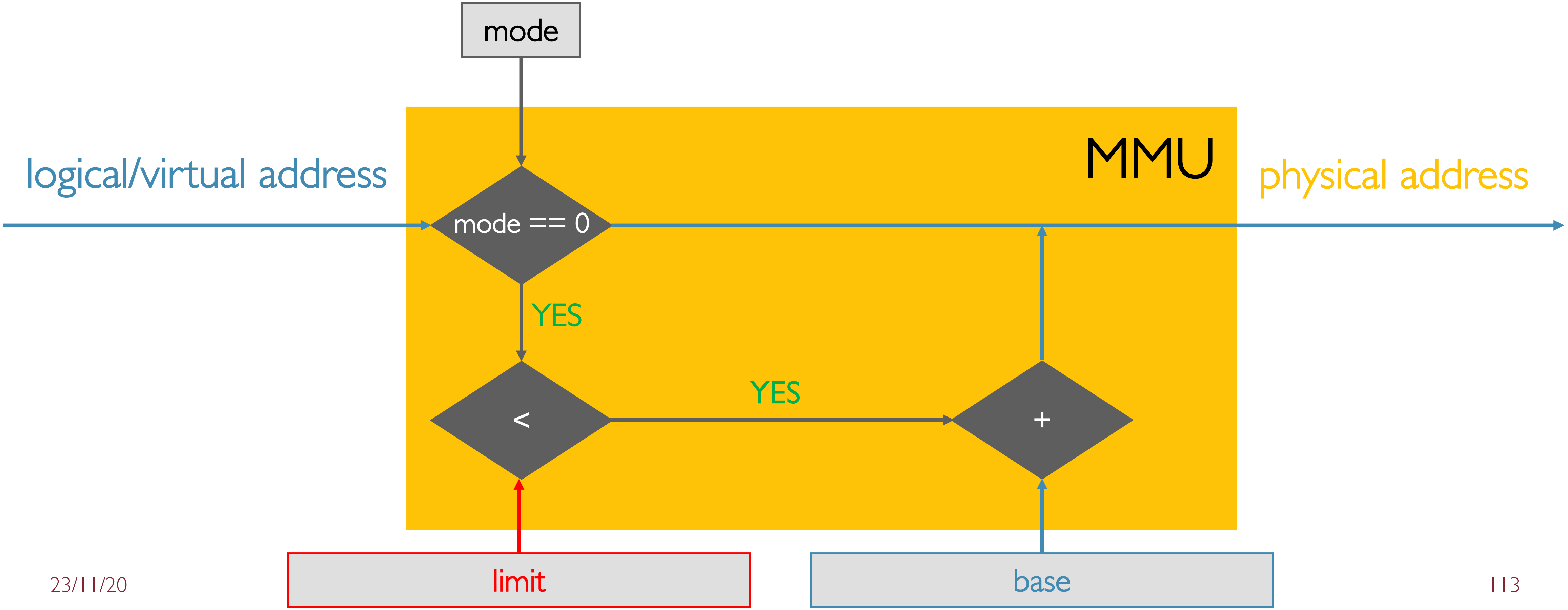
Implementing Dynamic Relocation

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



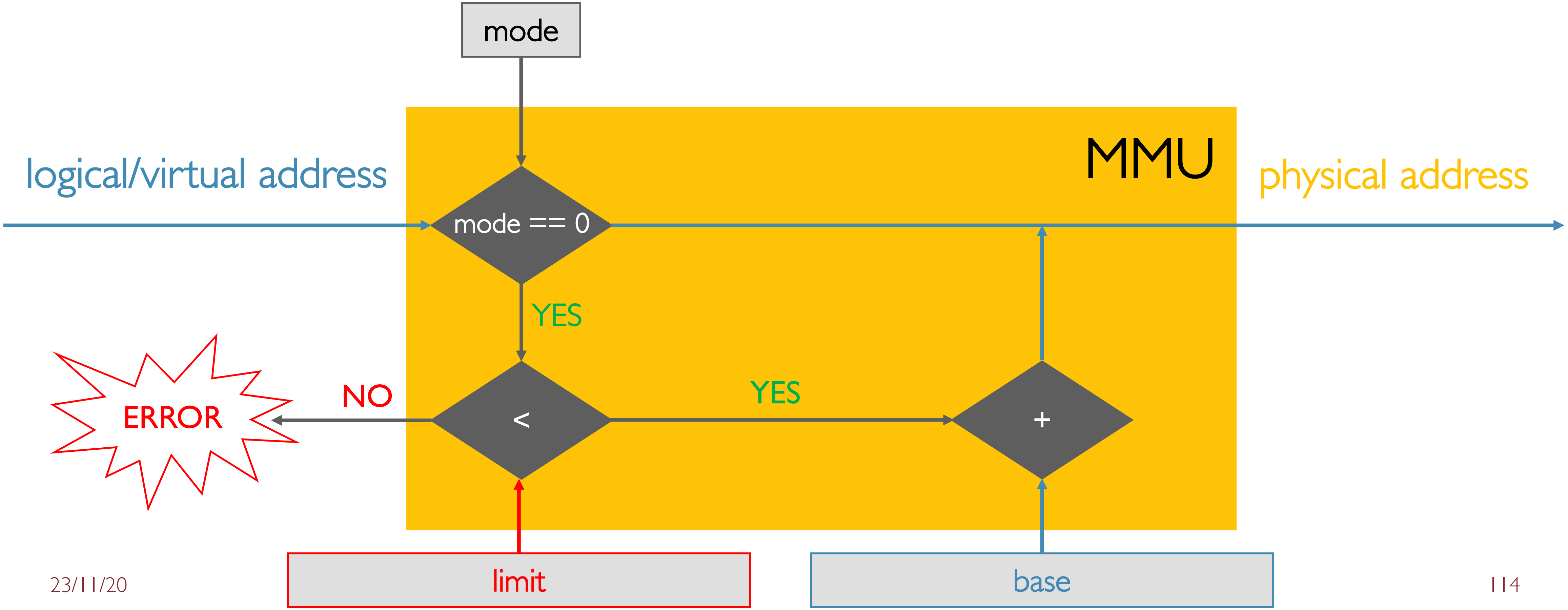
Implementing Dynamic Relocation

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

Contiguous Memory Allocation

- So far, we have assumed each process is allocated into a contiguous space of physical memory

Contiguous Memory Allocation

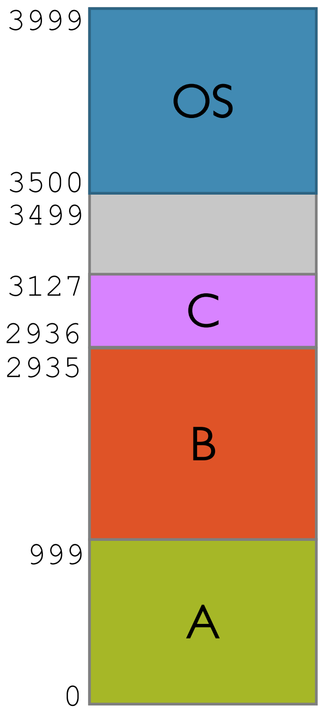
- So far, we have assumed each process is allocated into a contiguous space of physical memory
- One simple method is to divide upfront all available memory dedicated to user processes into equally-sized segments/partitions
  - Assign each process to a segment
  - Implicitly restricts the grade of multiprogramming (i.e., the number of simultaneous processes) and their size
  - No longer used!

Contiguous Memory Allocation

An alternative approach is for the OS to keep track of free (unused) memory segments, as processes enter the system, grow, and terminate

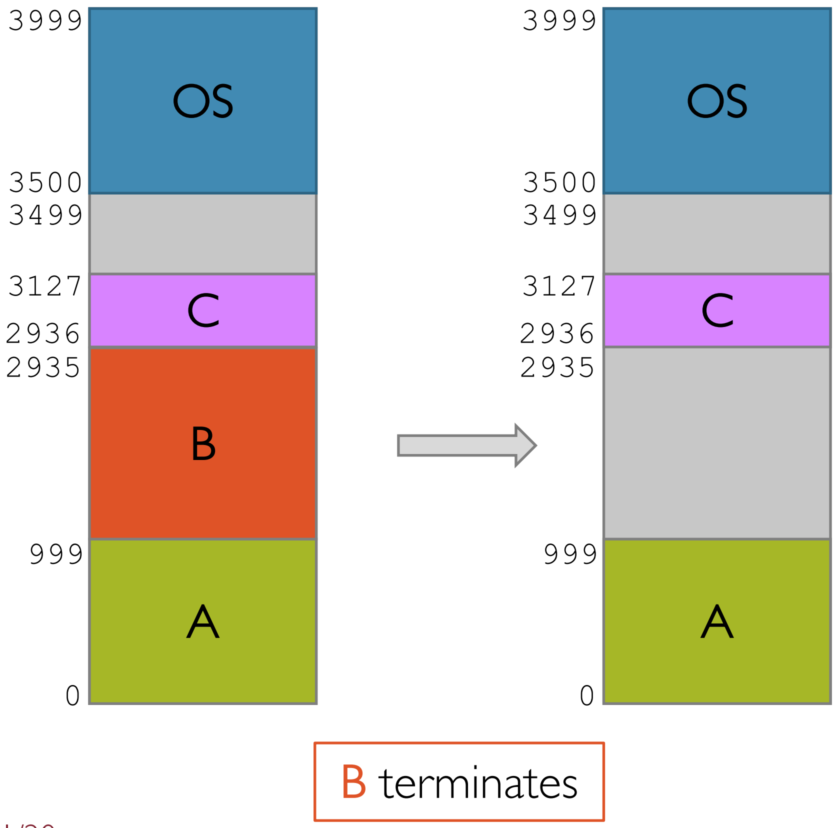
Contiguous Memory Allocation

An alternative approach is for the OS to keep track of **free** (unused) memory segments, as processes enter the system, grow, and terminate



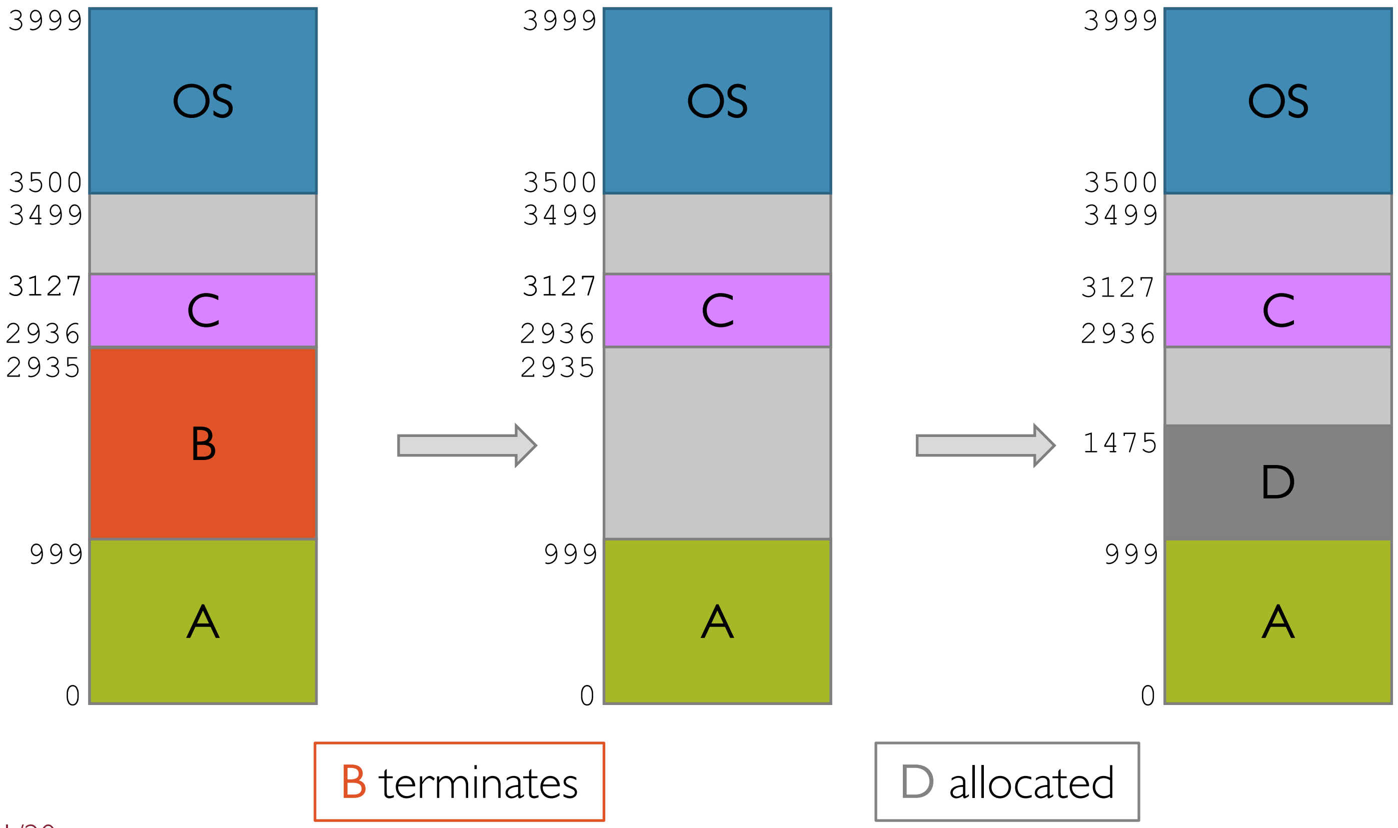
Contiguous Memory Allocation

An alternative approach is for the OS to keep track of free (unused) memory segments, as processes enter the system, grow, and terminate



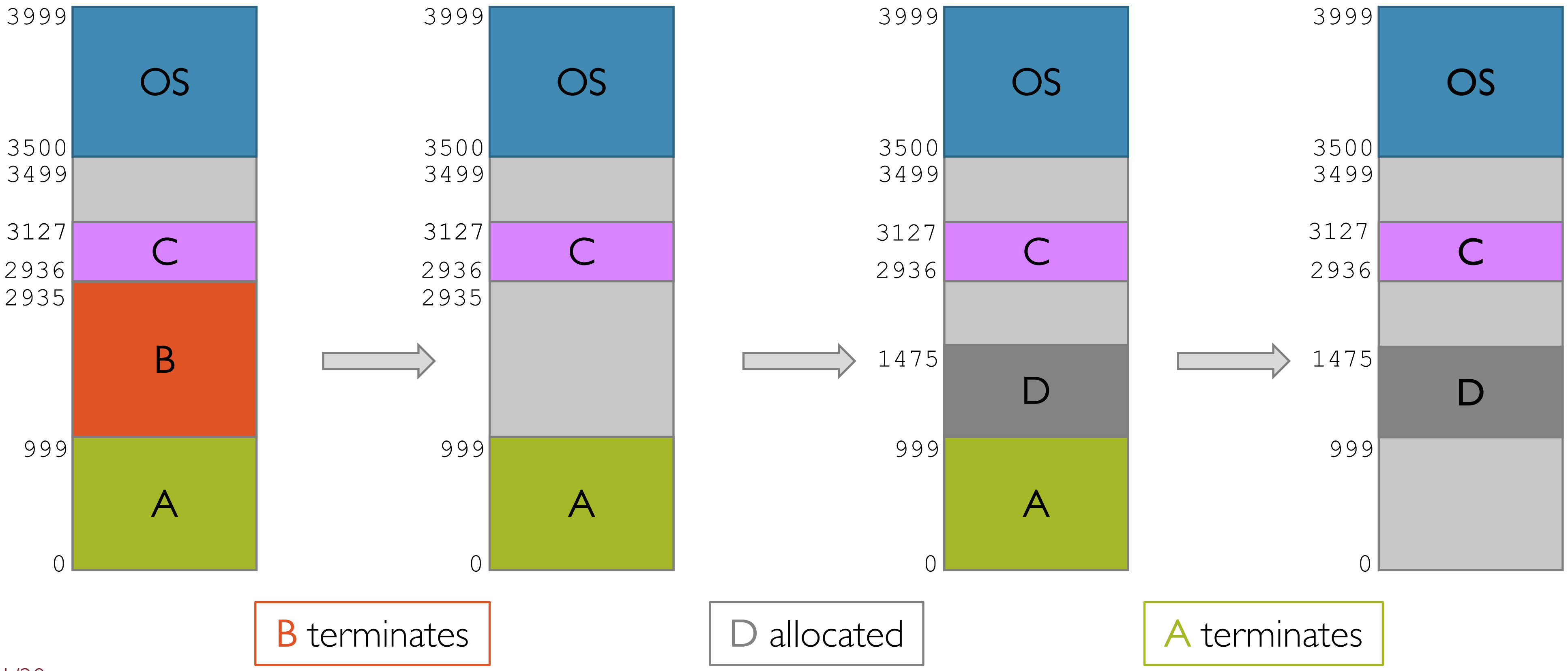
Contiguous Memory Allocation

An alternative approach is for the OS to keep track of **free** (unused) memory segments, as processes enter the system, grow, and terminate

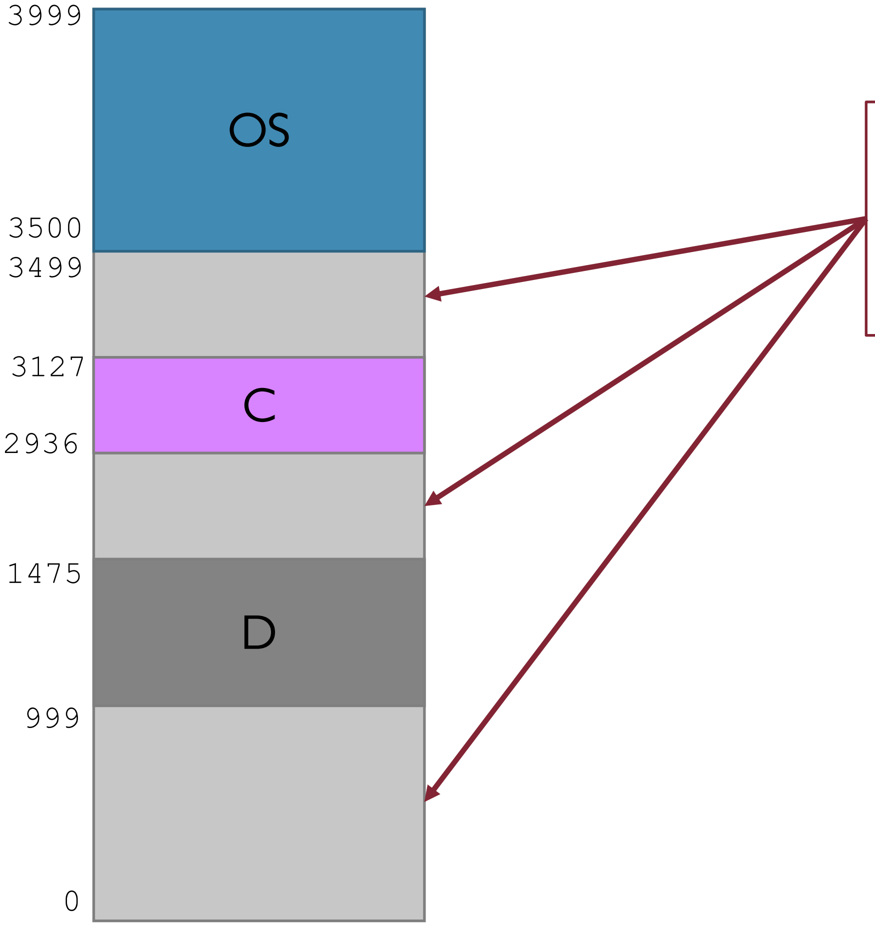


Contiguous Memory Allocation

An alternative approach is for the OS to keep track of free (unused) memory segments, as processes enter the system, grow, and terminate

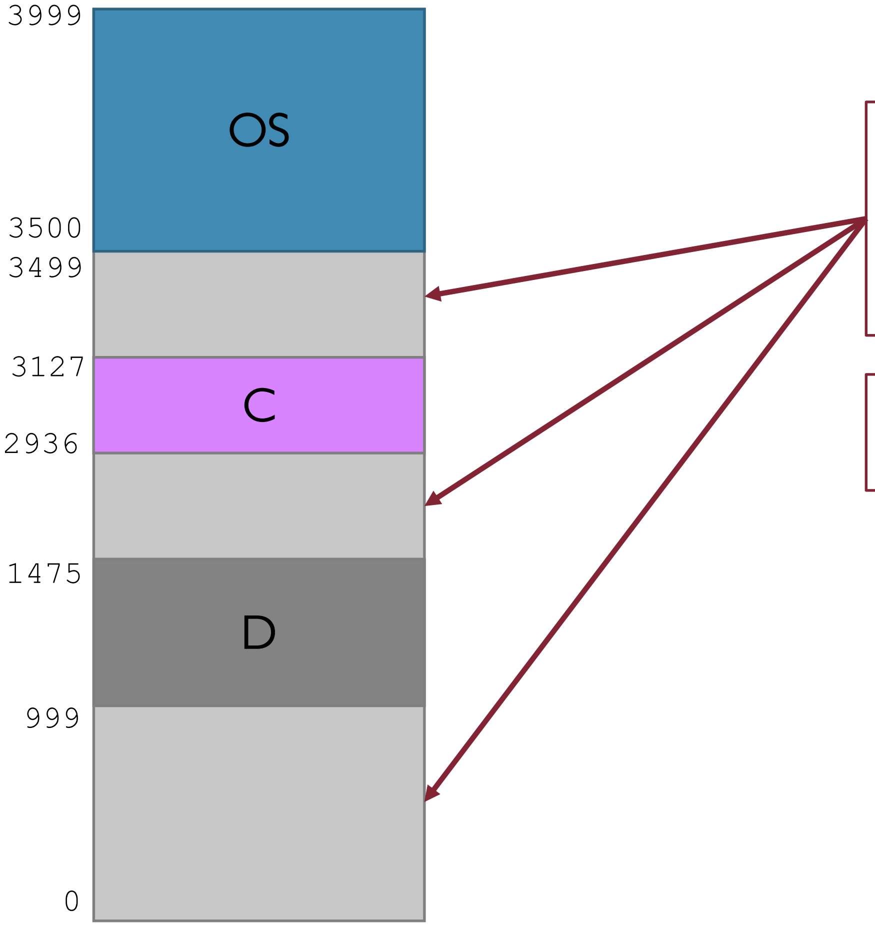


Contiguous Memory Allocation



Unused segments of memory are called **holes**

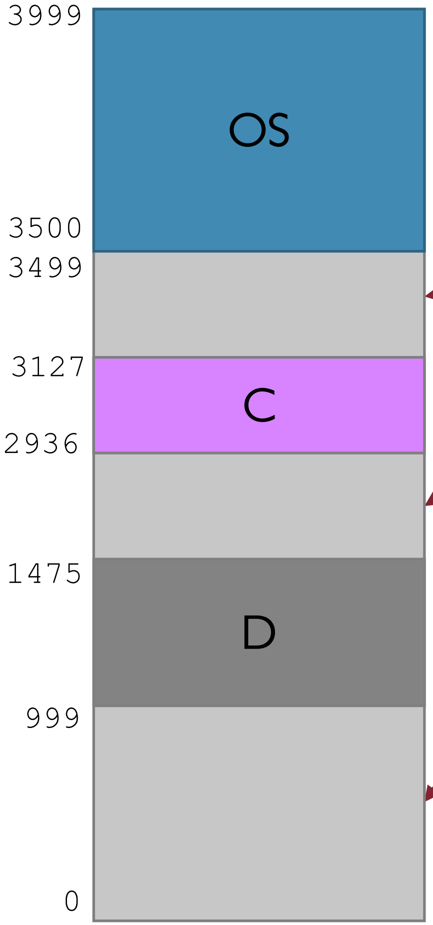
Contiguous Memory Allocation



Unused segments of memory are called **holes**

The OS keeps a list of holes

Contiguous Memory Allocation

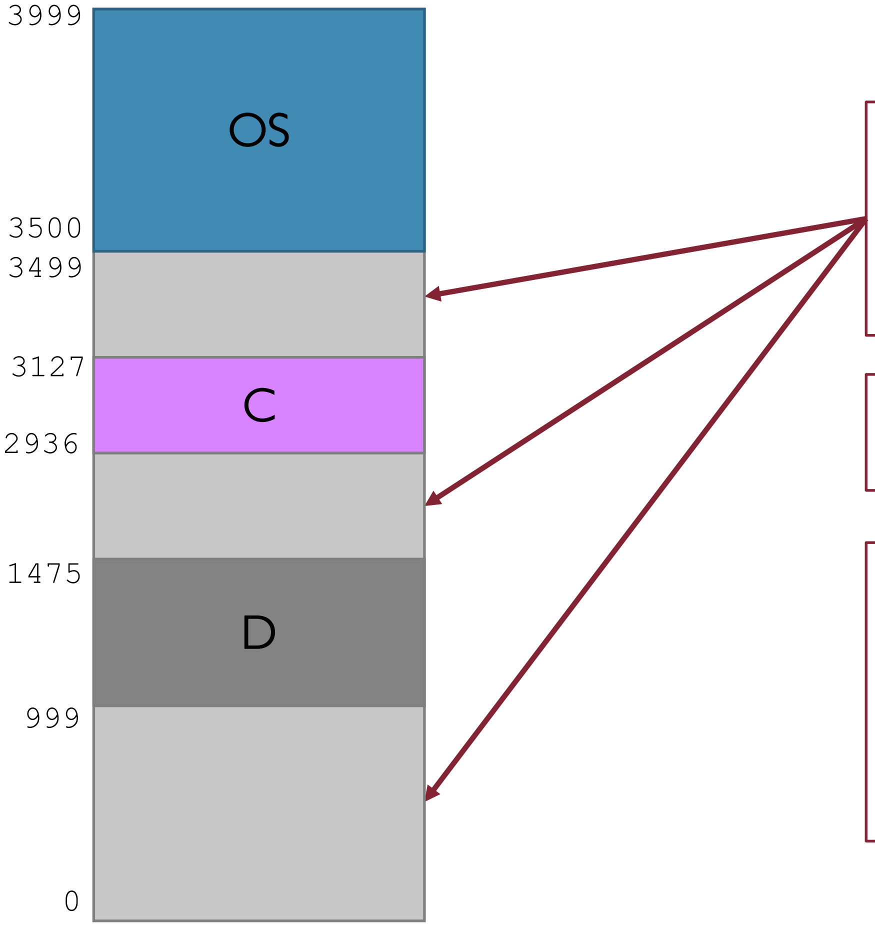


Unused segments of memory are called **holes**

The OS keeps a list of holes

Whenever a process has to be loaded, the OS must select a hole of suitable size

Contiguous Memory Allocation



Unused segments of memory are called **holes**

The OS keeps a list of holes

Whenever a process has to be loaded, the OS must select a hole of suitable size

How?

Memory Allocation Policies: First-Fit

- Linearly scan the list of holes until one is found that is big enough to satisfy the request

Memory Allocation Policies: First-Fit

- Linearly scan the list of holes until one is found that is big enough to satisfy the request
- Subsequent requests may either start from the beginning of the list or from the end of previous search

Memory Allocation Policies: First-Fit

- Linearly scan the list of holes until one is found that is big enough to satisfy the request
- Subsequent requests may either start from the beginning of the list or from the end of previous search
- Complexity: O(n), where n is the number of holes

Memory Allocation Policies: First-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:

Memory Allocation Policies: First-Fit

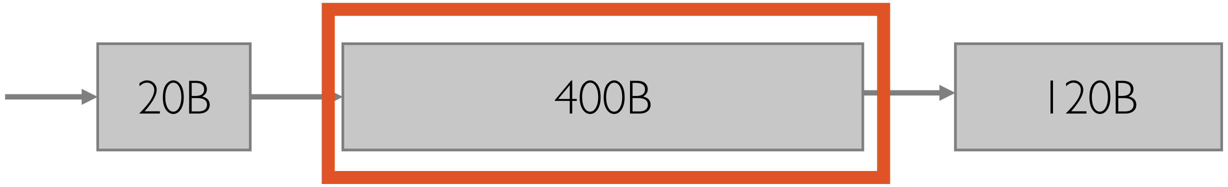
Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:

→ 20B → 400B → 120B

Which segment will be picked by the OS using first-fit?

Memory Allocation Policies: First-Fit

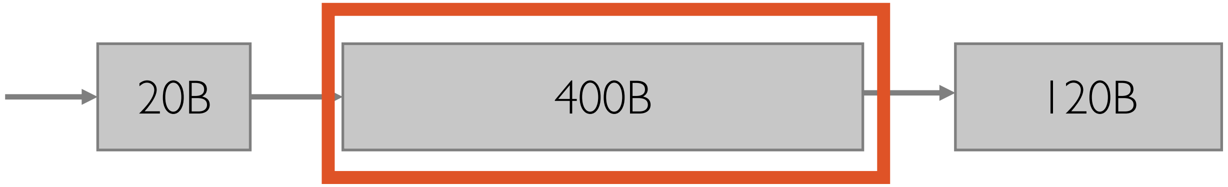
Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:



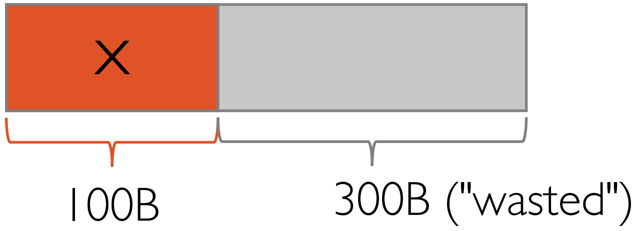
Which segment will be picked by the OS using first-fit?

Memory Allocation Policies: First-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:

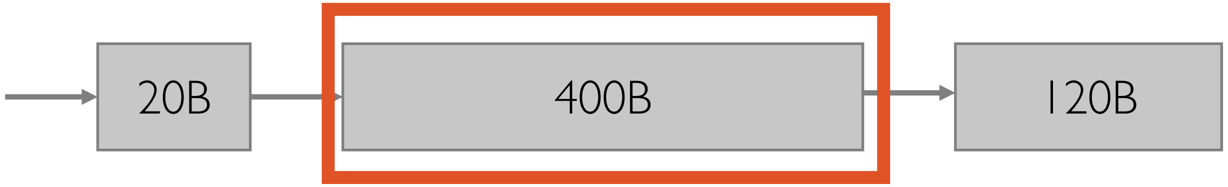


Which segment will be picked by the OS using first-fit?

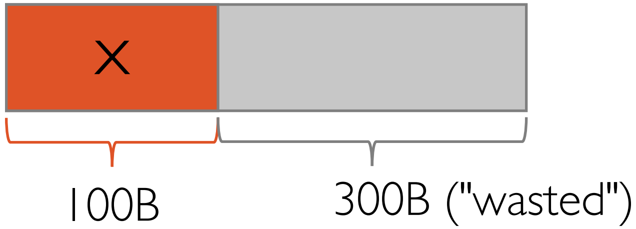


Memory Allocation Policies: First-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:



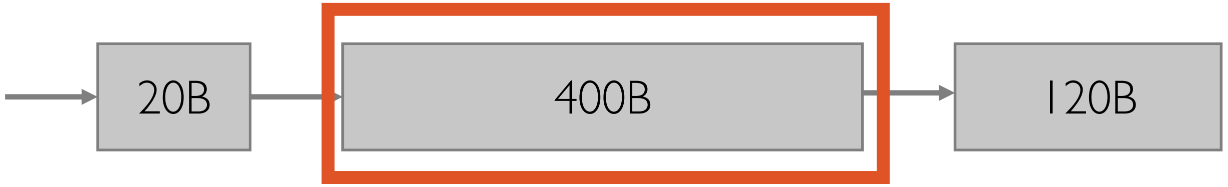
Which segment will be picked by the OS using first-fit?



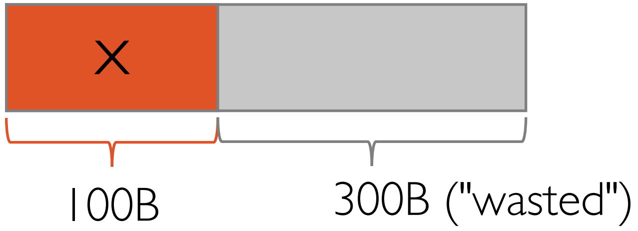
What if afterwards process Y requires 350B?

Memory Allocation Policies: First-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:



Which segment will be picked by the OS using first-fit?



What if afterwards process Y requires 350B?

We will not be able to satisfy this request even if theoretically we could

Memory Allocation Policies: Best-Fit

- Allocate the smallest hole that is big enough to satisfy the request

Memory Allocation Policies: Best-Fit

- Allocate the smallest hole that is big enough to satisfy the request
- This saves large holes for other process requests that may need them

Memory Allocation Policies: Best-Fit

- Allocate the smallest hole that is big enough to satisfy the request
- This saves large holes for other process requests that may need them
- However, the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted

Memory Allocation Policies: Best-Fit

- Allocate the smallest hole that is big enough to satisfy the request
- This saves large holes for other process requests that may need them
- However, the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted
- Complexity: still O(n) but can be O(log n) if the list of holes is kept sorted

Memory Allocation Policies: Best-Fit

- Allocate the smallest hole that is big enough to satisfy the request
- This saves large holes for other process requests that may need them
- However, the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted
- Complexity: still O(n) but can be O(log n) if the list of holes is kept sorted

Do you know how which data structure can be used to achieve this?

Memory Allocation Policies: Best-Fit

- Allocate the smallest hole that is big enough to satisfy the request
- This saves large holes for other process requests that may need them
- However, the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted
- Complexity: still O(n) but can be O(log n) if the list of holes is kept sorted

Do you know how which data structure can be used to achieve this?

Binary Search Tree (BST)

Memory Allocation Policies: Best-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:

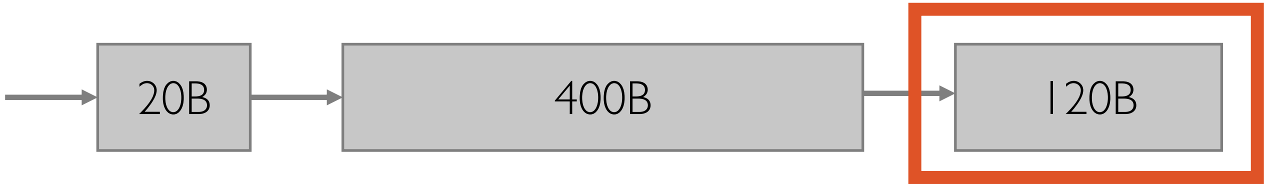
Memory Allocation Policies: Best-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:

Which segment will be picked by the OS using best-fit?

Memory Allocation Policies: Best-Fit

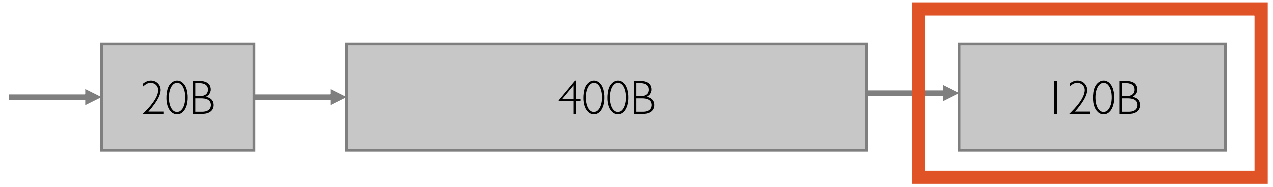
Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:



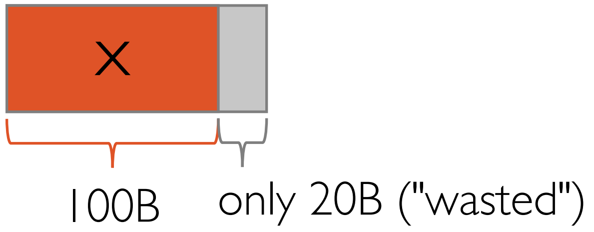
Which segment will be picked by the OS using best-fit?

Memory Allocation Policies: Best-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:

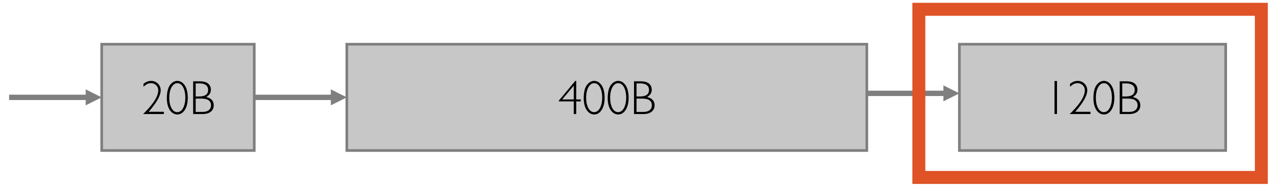


Which segment will be picked by the OS using best-fit?

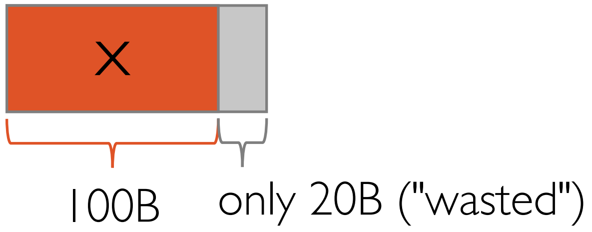


Memory Allocation Policies: Best-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:



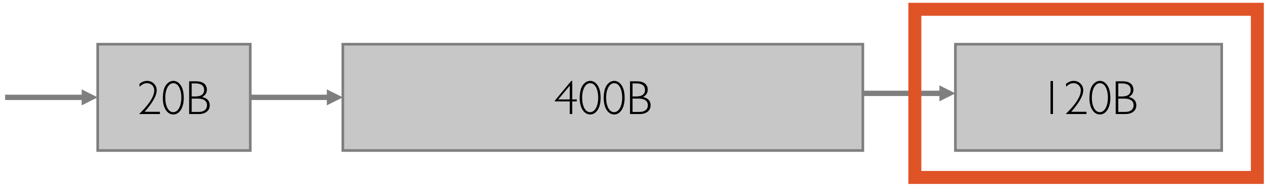
Which segment will be picked by the OS using best-fit?



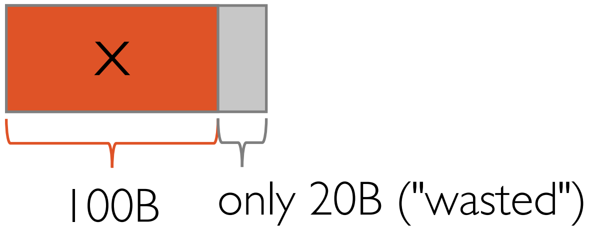
What if afterwards process Y requires 350B?

Memory Allocation Policies: Best-Fit

Suppose process X needs 100B of memory to be loaded, and the list of holes is as follows:



Which segment will be picked by the OS using best-fit?



What if afterwards process Y requires 350B?

We can now assign it the second available hole segment (400B)

Memory Allocation Policies: Worst-Fit

- Allocate the largest hole available

Memory Allocation Policies: Worst-Fit

- Allocate the largest hole available
- Might sound counterintuitive but this increases the likelihood that the remaining portion will be usable for satisfying future requests

Memory Allocation Policies: Worst-Fit

- Allocate the largest hole available
- Might sound counterintuitive but this increases the likelihood that the remaining portion will be usable for satisfying future requests
- Simulations show that First-Fit and Best-Fit usually work best

Memory Allocation Policies: Worst-Fit

- Allocate the largest hole available
- Might sound counterintuitive but this increases the likelihood that the remaining portion will be usable for satisfying future requests
- Simulations show that First-Fit and Best-Fit usually work best
- First-Fit is also generally faster than Best-Fit

Fragmentation

Problem

Individual holes may be too small to serve a process request but they can be large enough if combined together

Fragmentation

Problem

Individual holes may be too small to serve a process request but they can be large enough if combined together

External Fragmentation

Fragmentation

Problem

Individual holes may be too small to serve a process request but they can be large enough if combined together

External Fragmentation

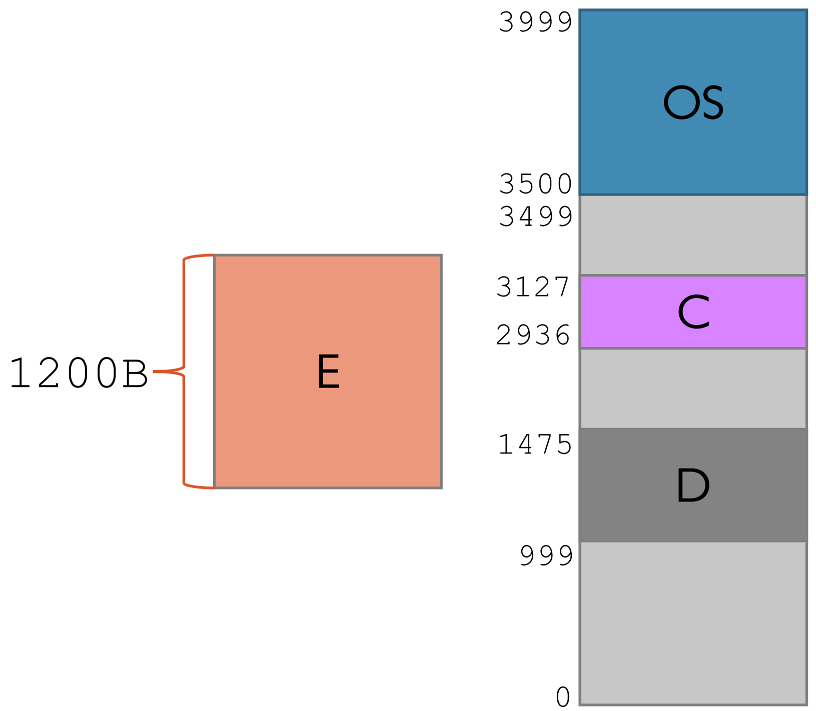
Internal Fragmentation

External Fragmentation

- Frequent loading and unloading processes causes holes to be broken into small (i.e., unusable) chunks

External Fragmentation

- Frequent loading and unloading processes causes holes to be broken into small (i.e., unusable) chunks
- It happens when there is enough memory to load a process in memory but space is not contiguous

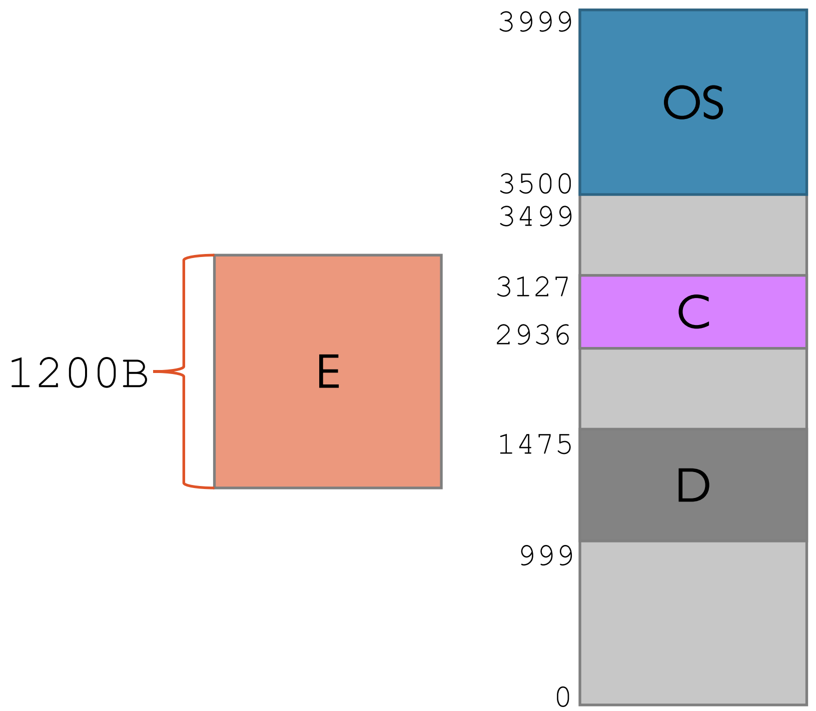


External Fragmentation

- Frequent loading and unloading processes causes holes to be broken into small (i.e., unusable) chunks
- It happens when there is enough memory to load a process in memory but space is not contiguous

Simulations show that for every 2N allocated blocks, N are lost due to external fragmentation

1/3 of memory space is wasted on average



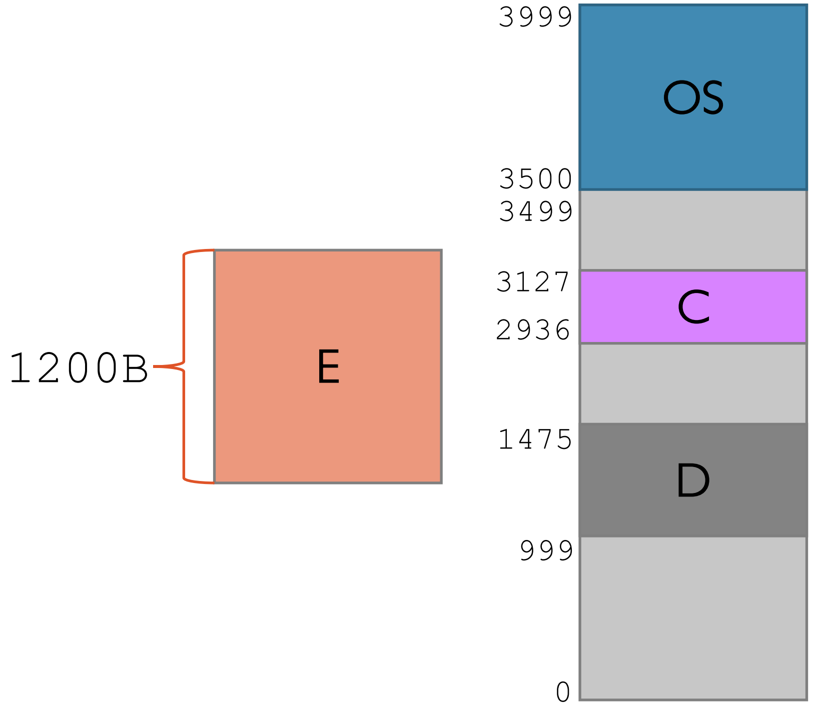
External Fragmentation

- Frequent loading and unloading processes causes holes to be broken into small (i.e., unusable) chunks
- It happens when there is enough memory to load a process in memory but space is not contiguous

Simulations show that for every 2N allocated blocks, N are lost due to external fragmentation

1/3 of memory space is wasted on average

<u>Goal:</u>
Allocation policy that minimizes wasted space!



Internal Fragmentation

- It happens when memory internal to a segment is wasted

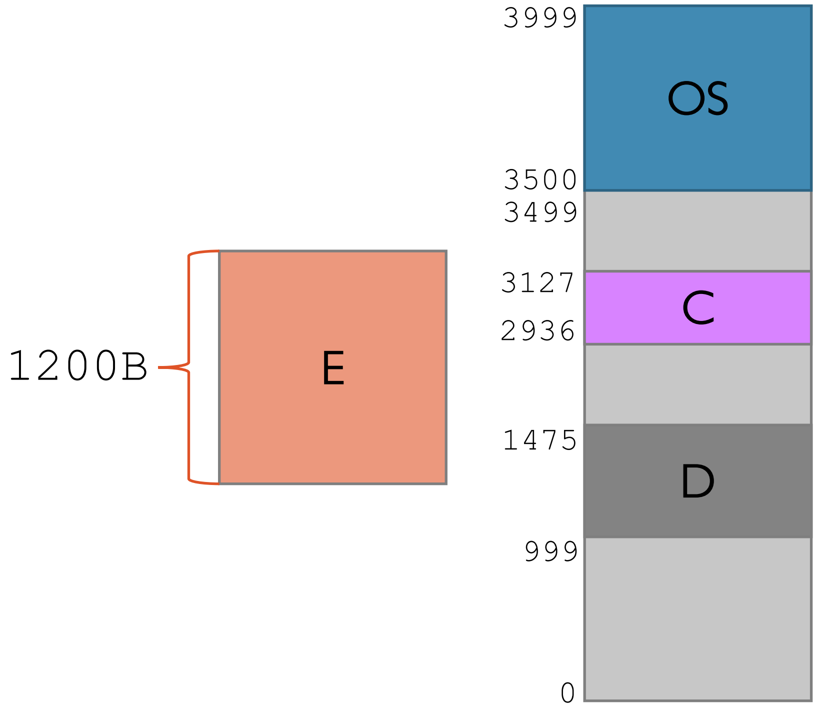
Internal Fragmentation

- It happens when memory internal to a segment is wasted
- For example, consider a process whose size is 8,846B and a hole of size 8,848B

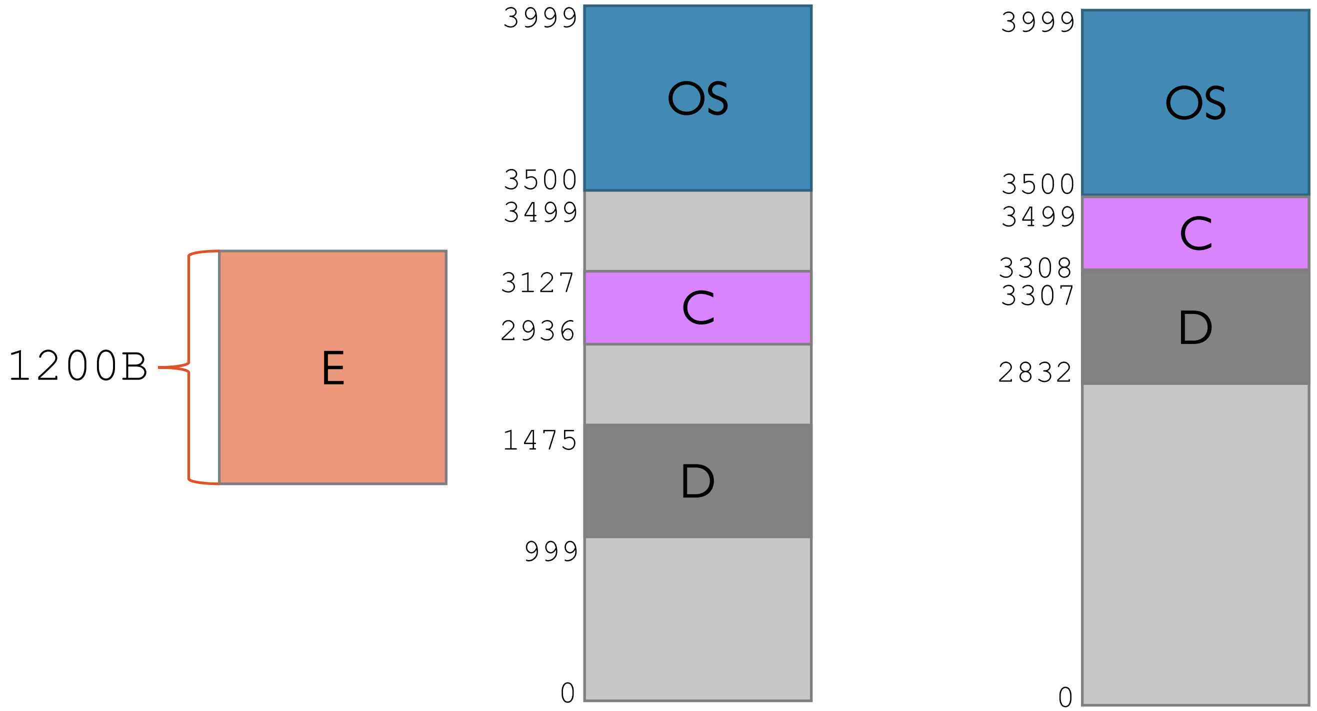
Internal Fragmentation

- It happens when memory internal to a segment is wasted
- For example, consider a process whose size is 8,846B and a hole of size 8,848B
- It may be much more efficient to allocate the process the whole block (and waste 2B) rather than keep track of a tiny 2B hole

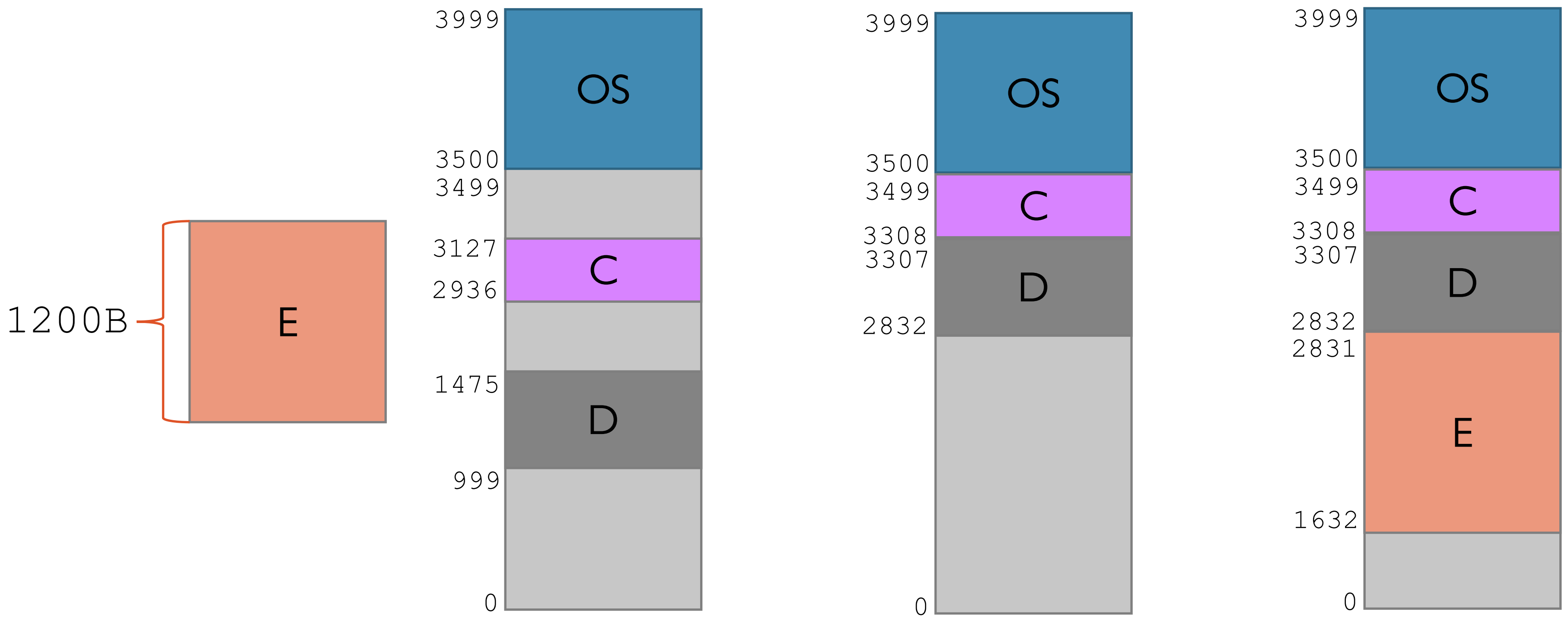
Solution to Fragmentation: Full Compaction



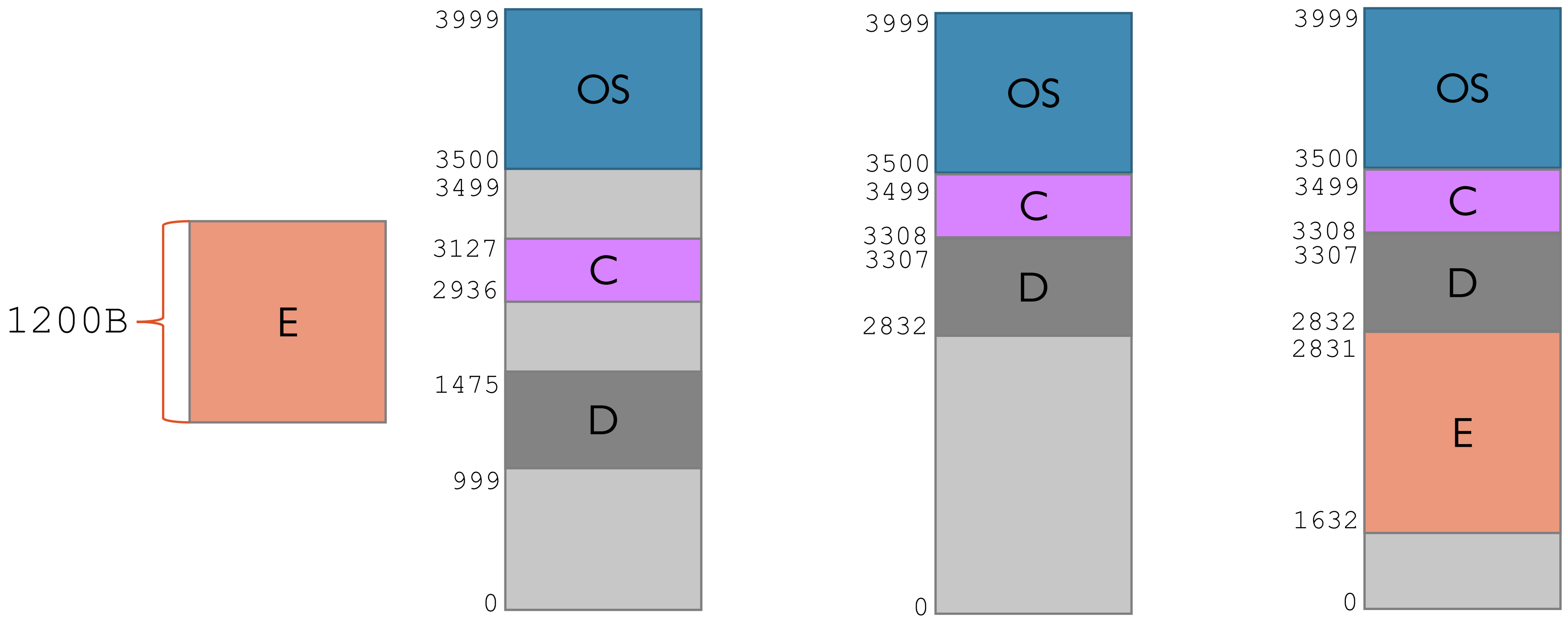
Solution to Fragmentation: Full Compaction



Solution to Fragmentation: Full Compaction

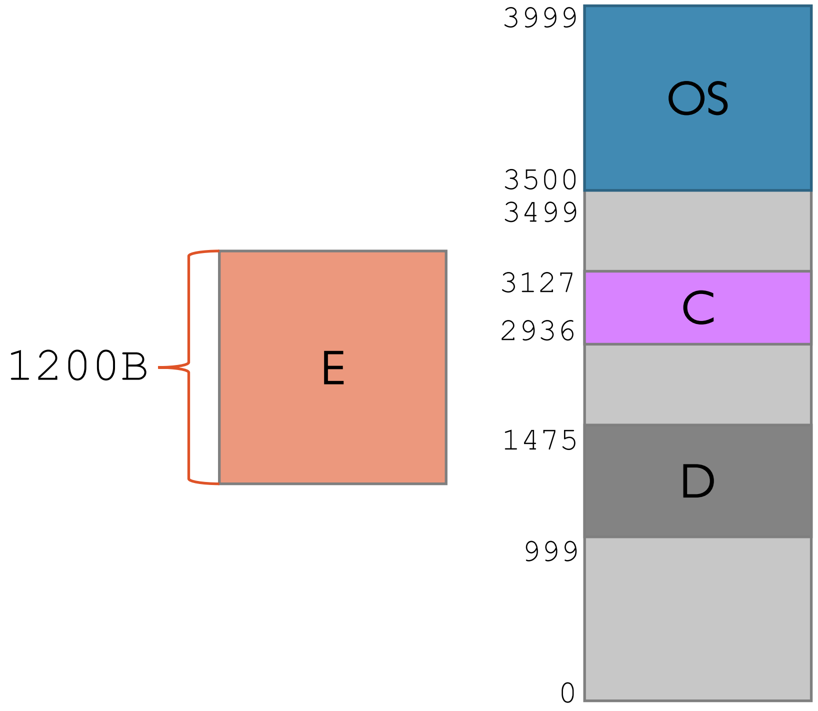


Solution to Fragmentation: Full Compaction

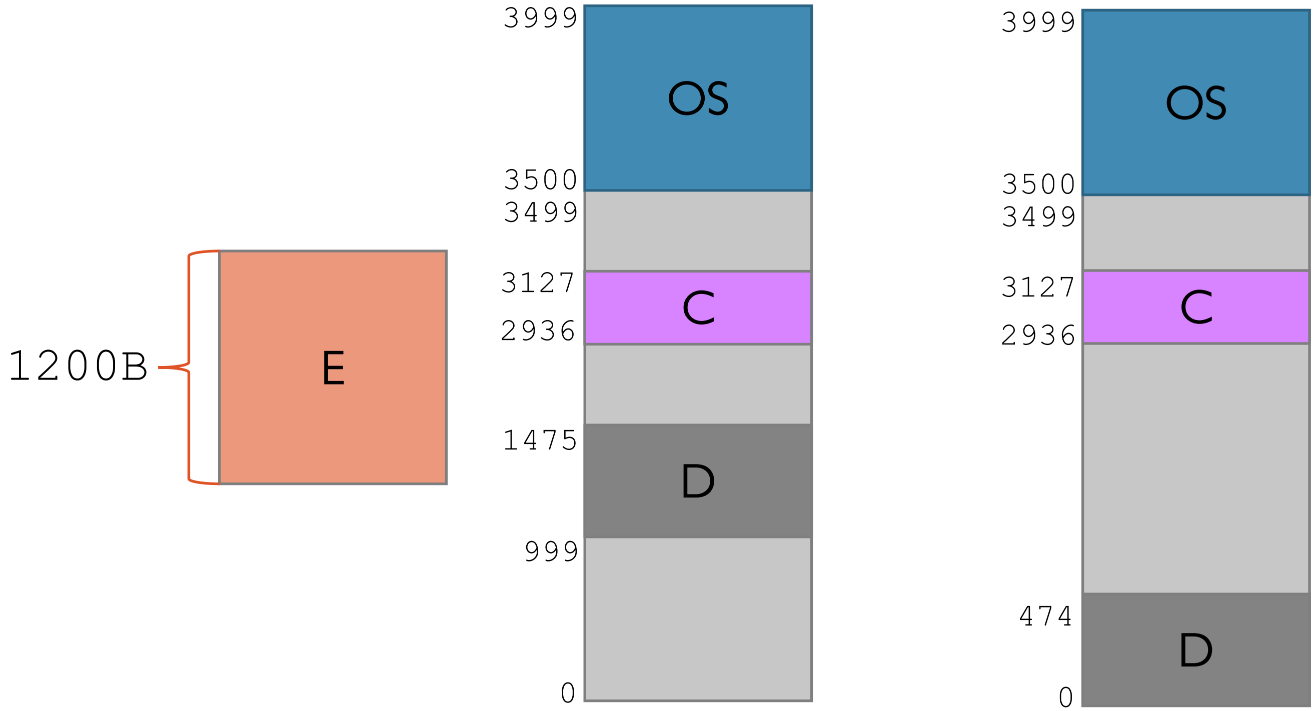


Only one hole is left but two processes need to be moved (C and D)

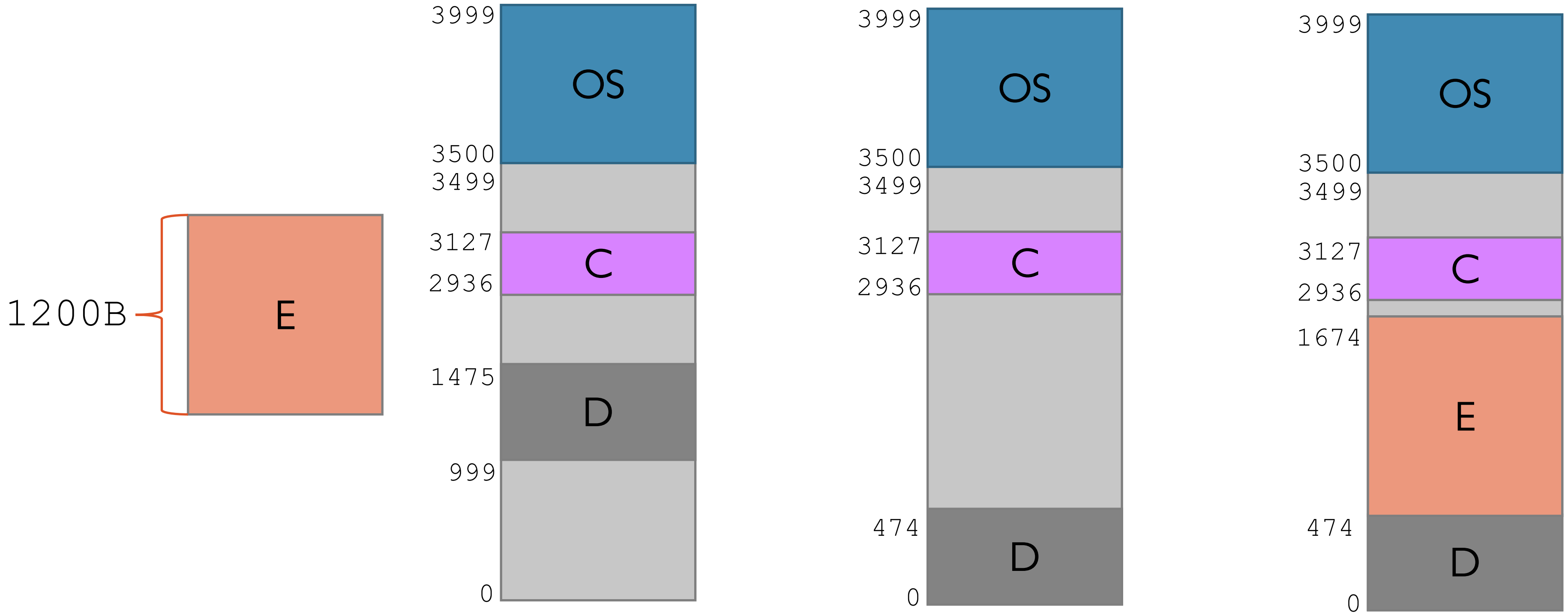
Solution to Fragmentation: Partial Compaction



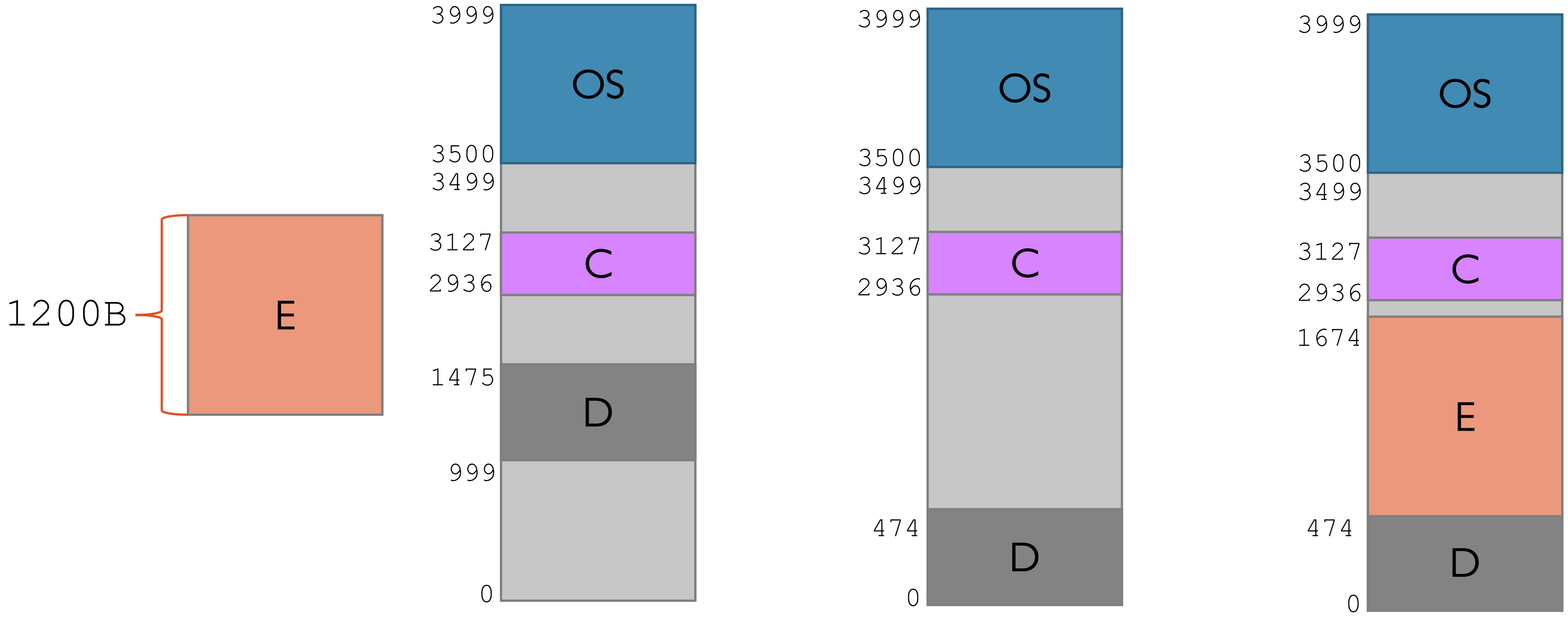
Solution to Fragmentation: Partial Compaction



Solution to Fragmentation: Partial Compaction



Solution to Fragmentation: Partial Compaction



Still some holes left but only one process is moved (D) rather than two

Swapping

- So far, we have assumed all processes are entirely loaded in memory (of course, when loaded!)

Swapping

- So far, we have assumed all processes are entirely loaded in memory (of course, when loaded!)
- Remember: A process needs to sit physically in main memory only if the CPU executes its instructions and accesses its data

Swapping

- So far, we have assumed all processes are entirely loaded in memory (of course, when loaded!)
- Remember: A process needs to sit physically in main memory only if the CPU executes its instructions and accesses its data
- If a process blocks (e.g., due to an I/O call) it doesn't need to be in memory while I/O is running

Swapping

- So far, we have assumed all processes are entirely loaded in memory (of course, when loaded!)
- Remember: A process needs to sit physically in main memory only if the CPU executes its instructions and accesses its data
- If a process blocks (e.g., due to an I/O call) it doesn't need to be in memory while I/O is running
- That process can be "swapped out" from memory to disk to make room for other processes

Swapping

- Once process becomes ready again, the OS must reload it in memory

Swapping

- Once process becomes ready again, the OS must reload it in memory
- Swap in depends on the address binding used:
  - compile- or load-time: must be swapped back into the same memory location from which they were swapped out

Swapping

- Once process becomes ready again, the OS must reload it in memory
- Swap in depends on the address binding used:
  - compile- or load-time: must be swapped back into the same memory location from which they were swapped out
  - execution-time: can be swapped back into any available location (updating base and limit registers)

Swapping

- Once process becomes ready again, the OS must reload it in memory
- Swap in depends on the address binding used:
  - compile- or load-time: must be swapped back into the same memory location from which they were swapped out
  - execution-time: can be swapped back into any available location (updating base and limit registers)
- Using swapping, fragmentation can be tackled easily
  - Just run compaction before swapping-in a process

Swapping: Example

- Swapping is a very slow process compared to other operations due to the interaction with hard disk (more on this later)

Swapping: Example

- Swapping is a very slow process compared to other operations due to the interaction with hard disk (more on this later)
- Example:
  - 10 MB user process
  - disk transfer rate = 40 MB/sec (250 msec just to do the data transfer)

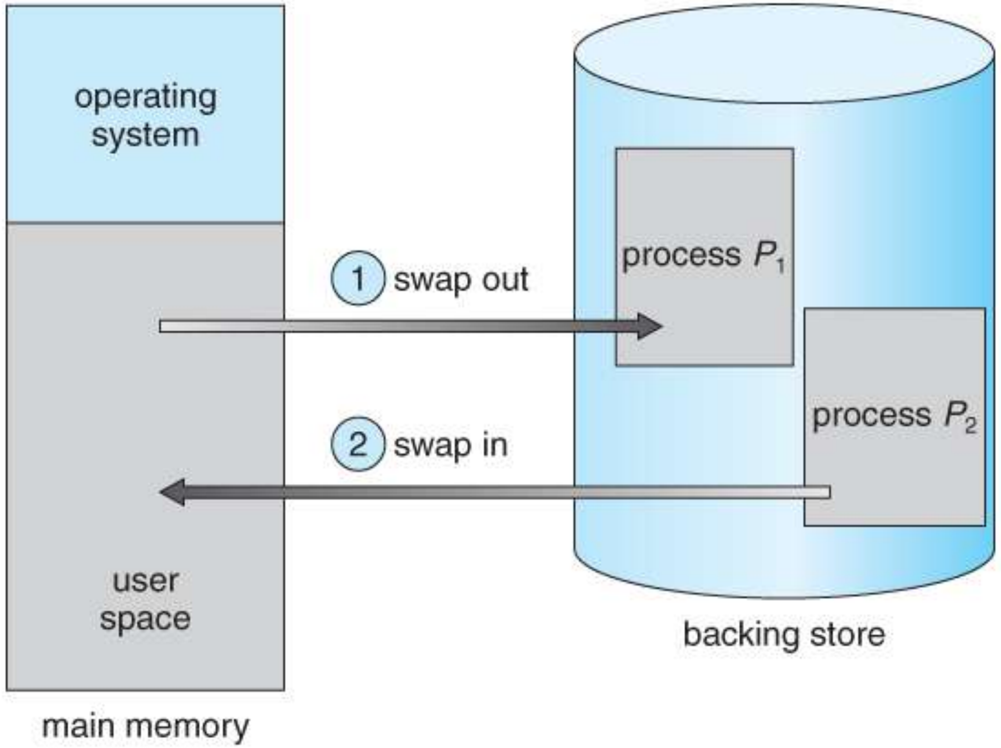
Swapping: Example

- Swapping is a very slow process compared to other operations due to the interaction with hard disk (more on this later)
- Example:
  - 10 MB user process
  - disk transfer rate = 40 MB/sec (250 msec just to do the data transfer)
- Since swap-in may involve swapping-out another process, the overall time required will be ~500 msec

Swapping: Example

- Swapping is a very slow process compared to other operations due to the interaction with hard disk (more on this later)
- Example:
    - 10 MB user process
    - disk transfer rate = 40 MB/sec (250 msec just to do the data transfer)
- Since swap-in may involve swapping-out another process, the overall time required will be ~500 msec
- Time slice is usually way smaller than that!

Swapping



Most modern OSs no longer use swapping, because it is too slow and there are faster alternatives available (e.g., paging)

Summary

- Effective memory management is crucial for system performance

Summary

- Effective memory management is crucial for system performance
- Very basic management doesn't even require OS intervention

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

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
- More advanced strategies are needed to overcome:
  - contiguous allocation, fragmentation, and entire loading