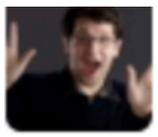
# Operating Systems
## Lecture 10: Memory Virtualisation
## Base & Bounds + Segmentation

Nipun Batra
Aug 24, 2018

# CS stories



**Dion Almaer**
@dalmaer

Following

Hey @duhroach, would be a fun show.... where do all the bytes come from! (And increase in pixels / density etc) :)
m.imgur.com/3TtTwrU

The original NES console was only designed to output images that were 256 wide by 240 high; meaning that the final image that needed to be displayed to the screen was 180kb in size.

Besides that, the NES only had 2kb of RAM. A cartridge itself could could hold 8k to 1mb of game data. So, there was no way to fit the entire game's contents into main memory. Basically, a subset of the 1MB cartridge data must be loaded into the 2kb RAM and used to render the 180kb screen. How does one achieve that?

SpriteSheets.

Sprite sheets contain small tiles of graphics, that are re-used over and over again. For example, below is a remake of the original super mario sprite sheet:

# Administrative

1. 2 - 3 mins CS story at start of lecture - given by you!
    1. Technically interesting tidbit
    2. CS hero - Grace Hopper, Kerninghan …
2. +1 marks
3. Order : Script run by instructor
4. Lecture tomorrow, 25th August, 4 PM, 7/102

# Revision

# Revision

1. Space v/s Time Multiplexing

# Revision

1. Space v/s Time Multiplexing
2. Direct Physical Memory Multiplexing :

# Revision

1. Space v/s Time Multiplexing
2. Direct Physical Memory Multiplexing :
    1. Limited to physical memory

# Revision

1. Space v/s Time Multiplexing
2. Direct Physical Memory Multiplexing :
    1. Limited to physical memory
    2. Fragmentation :

# Revision

1. Space v/s Time Multiplexing
2. Direct Physical Memory Multiplexing :
    1. Limited to physical memory
    2. Fragmentation :
        1. Internal

# Revision

1. Space v/s Time Multiplexing
2. Direct Physical Memory Multiplexing :
    1. Limited to physical memory
    2. Fragmentation :
        1. Internal
        2. External

# Revision

1. Space v/s Time Multiplexing
2. Direct Physical Memory Multiplexing :
    1. Limited to physical memory
    2. Fragmentation :
        1. Internal
        2. External
3. Goals of Memory Management:

# Revision

1. Space v/s Time Multiplexing
2. Direct Physical Memory Multiplexing :
    1. Limited to physical memory
    2. Fragmentation :
        1. Internal
        2. External
3. Goals of Memory Management:
    1. Transparency

# Revision

1. Space v/s Time Multiplexing
2. Direct Physical Memory Multiplexing :
    1. Limited to physical memory
    2. Fragmentation :
        1. Internal
        2. External
3. Goals of Memory Management:
    1. Transparency
    2. Efficiency

# Revision

1. Space v/s Time Multiplexing
2. Direct Physical Memory Multiplexing :
    1. Limited to physical memory
    2. Fragmentation :
        1. Internal
        2. External
3. Goals of Memory Management:
    1. Transparency
    2. Efficiency
    3. Isolation

# Revision

1. Space v/s Time Multiplexing
2. Direct Physical Memory Multiplexing :
    1. Limited to physical memory
    2. Fragmentation :
        1. Internal
        2. External
3. Goals of Memory Management:
    1. Transparency
    2. Efficiency
    3. Isolation
4. Memory Interface : Load and Store

# Revision

1. Space v/s Time Multiplexing
2. Direct Physical Memory Multiplexing :
    1. Limited to physical memory
    2. Fragmentation :
        1. Internal
        2. External
3. Goals of Memory Management:
    1. Transparency
    2. Efficiency
    3. Isolation
4. Memory Interface : Load and Store
5. Abstraction

# Revision

1. Space v/s Time Multiplexing
2. Direct Physical Memory Multiplexing :
    1. Limited to physical memory
    2. Fragmentation :
        1. Internal
        2. External
3. Goals of Memory Management:
    1. Transparency
    2. Efficiency
    3. Isolation
4. Memory Interface : Load and Store
5. Abstraction
    1. Decouple physical memory and address

# Hello Assembly

```
void func() {
    int x = 3000; //
    x = x + 3;    //
    ...
```

# Hello Assembly

```
void func() {
    int x = 3000; //
    x = x + 3;    //
    ...
```

Compiler →

# Hello Assembly

```
void func() {
    int x = 3000; //
    x = x + 3;    //
    ...
```


```
128: movl 0x0(%ebx), %eax
132: addl $0x03, %eax
135: movl %eax, 0x0(%ebx)
```

# Hello Assembly

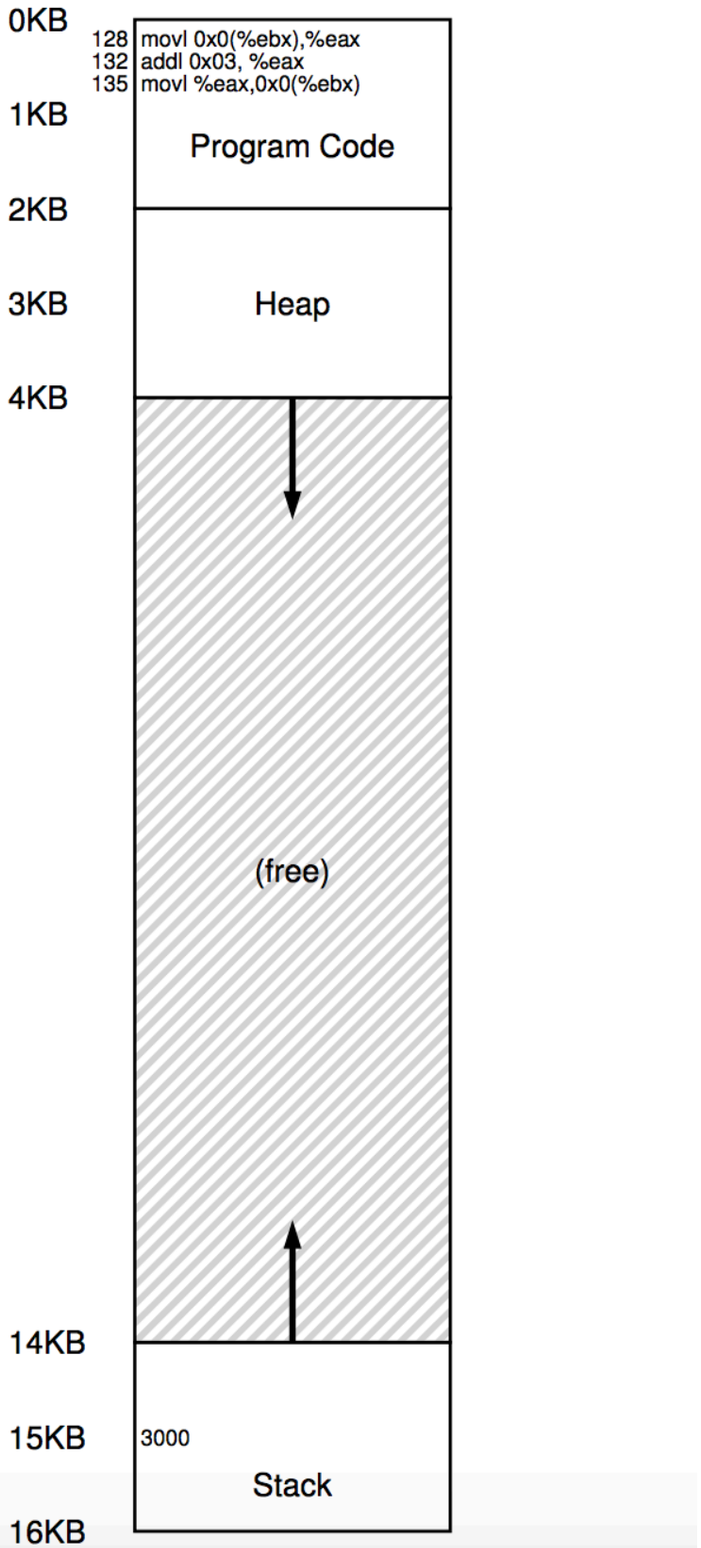
Compiler →

```
128: movl 0x0(%ebx), %eax
132: addl $0x03, %eax
135: movl %eax, 0x0(%ebx)
```

# Hello Assembly

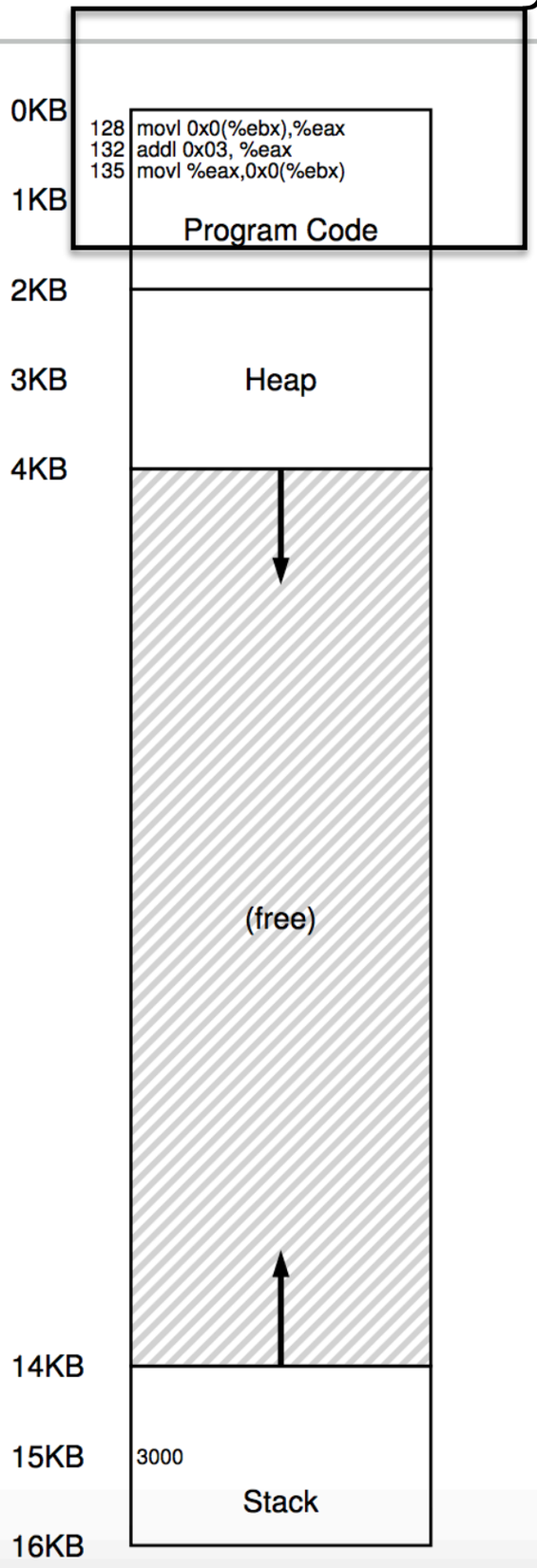
```
128: movl 0x0(%ebx), %eax
132: addl $0x03, %eax
135: movl %eax, 0x0(%ebx)
```

# Hello Assembly



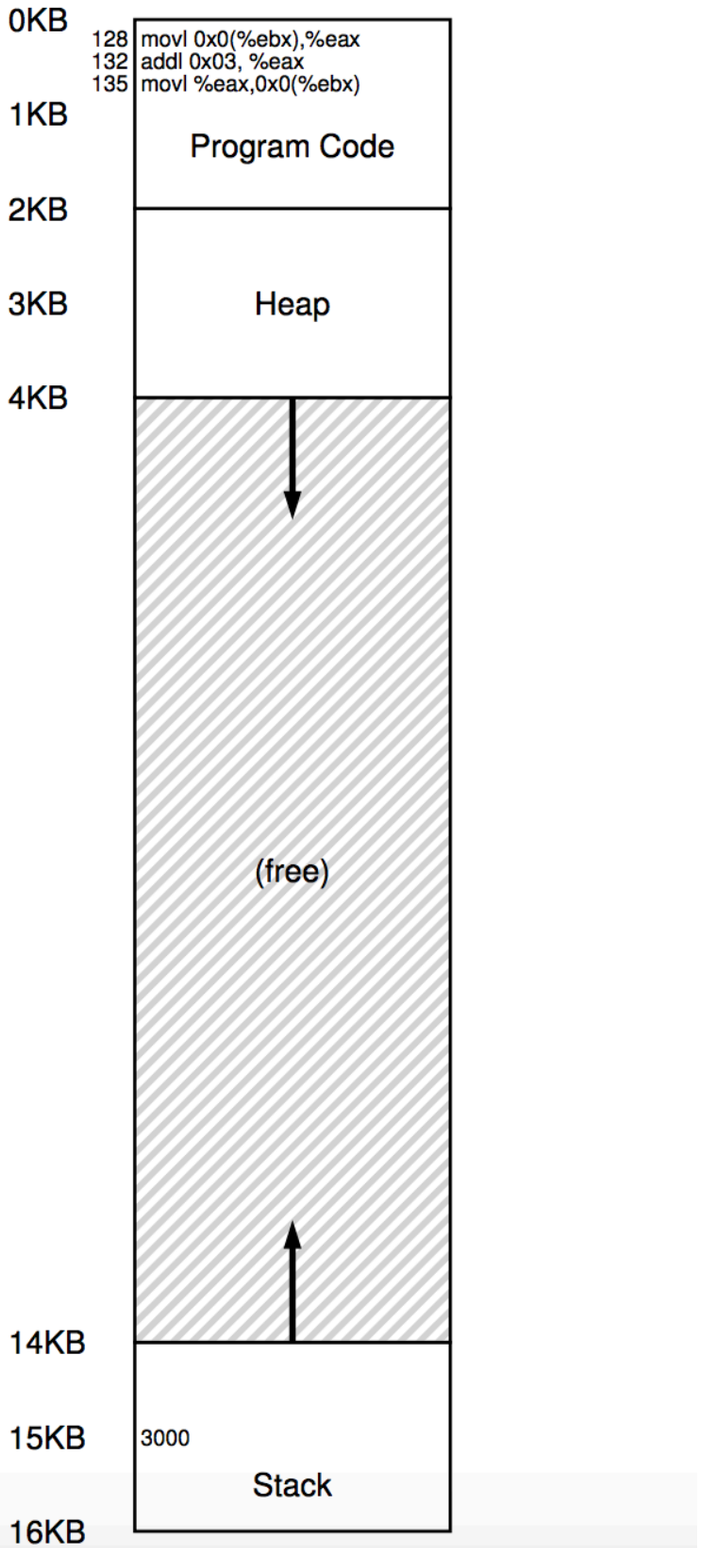

```
128: movl 0x0(%ebx), %eax
132: addl $0x03, %eax
135: movl %eax, 0x0(%ebx)
```

# Hello Assembly




```
128: movl 0x0(%ebx), %eax
132: addl $0x03, %eax
135: movl %eax, 0x0(%ebx)
```

# Hello Assembly




```
128: movl 0x0(%ebx), %eax
132: addl $0x03, %eax
135: movl %eax, 0x0(%ebx)
```

# Hello Assembly

# Hello Assembly

```
▶ cat hello_world.c
#include<stdio.h>

int main()
{
int x = 10;
x = x + 4;
x = x*4;
return 0;
}
```

# Hello Assembly

```
▶ cat hello_world.c
#include<stdio.h>

int main()
{
int x = 10;
x = x + 4;
x = x*4;
return 0;
}
```

```
▶ otool -tv hello_world
hello_world:
(__TEXT,__text) section
_main:
0000000100000f90        pushq   %rbp
0000000100000f91        movq    %rsp, %rbp
0000000100000f94        xorl    %eax, %eax
0000000100000f96        movl    $0x0, -0x4(%rbp)
0000000100000f9d        movl    $0xa, -0x8(%rbp)
0000000100000fa4        movl    -0x8(%rbp), %ecx
0000000100000fa7        addl    $0x4, %ecx
0000000100000faa        movl    %ecx, -0x8(%rbp)
0000000100000fad        movl    -0x8(%rbp), %ecx
0000000100000fb0        shll    $0x2, %ecx
0000000100000fb3        movl    %ecx, -0x8(%rbp)
0000000100000fb6        popq    %rbp
0000000100000fb7        retq
```

# Hello Assembly

| Suffix | Name | Size |
|---|---|---|
| B | BYTE | 1 byte (8 bits) |
| W | WORD | 2 bytes (16 bits) |
| L | LONG | 4 bytes (32 bits) |
| Q | QUADWORD | 8 bytes (64 bits) |

```
▶ cat hello_world.c
#include<stdio.h>

int main()
{
int x = 10;
x = x + 4;
x = x*4;
return 0;
}
```

```
▶ otool -tv hello_world
hello_world:
(__TEXT,__text) section
_main:
0000000100000f90        pushq   %rbp
0000000100000f91        movq    %rsp, %rbp
0000000100000f94        xorl    %eax, %eax
0000000100000f96        movl    $0x0, -0x4(%rbp)
0000000100000f9d        movl    $0xa, -0x8(%rbp)
0000000100000fa4        movl    -0x8(%rbp), %ecx
0000000100000fa7        addl    $0x4, %ecx
0000000100000faa        movl    %ecx, -0x8(%rbp)
0000000100000fad        movl    -0x8(%rbp), %ecx
0000000100000fb0        shll    $0x2, %ecx
0000000100000fb3        movl    %ecx, -0x8(%rbp)
0000000100000fb6        popq    %rbp
0000000100000fb7        retq
```

# General Address Translation

Kernel

CPU

MMU

Physical Memory

# General Address Translation

Kernel

CPU

MMU

Physical Memory

Virtual Address

# General Address Translation

Kernel

CPU

MMU

Physical Memory

Virtual Address
0x10102030

# General Address Translation

Kernel

CPU

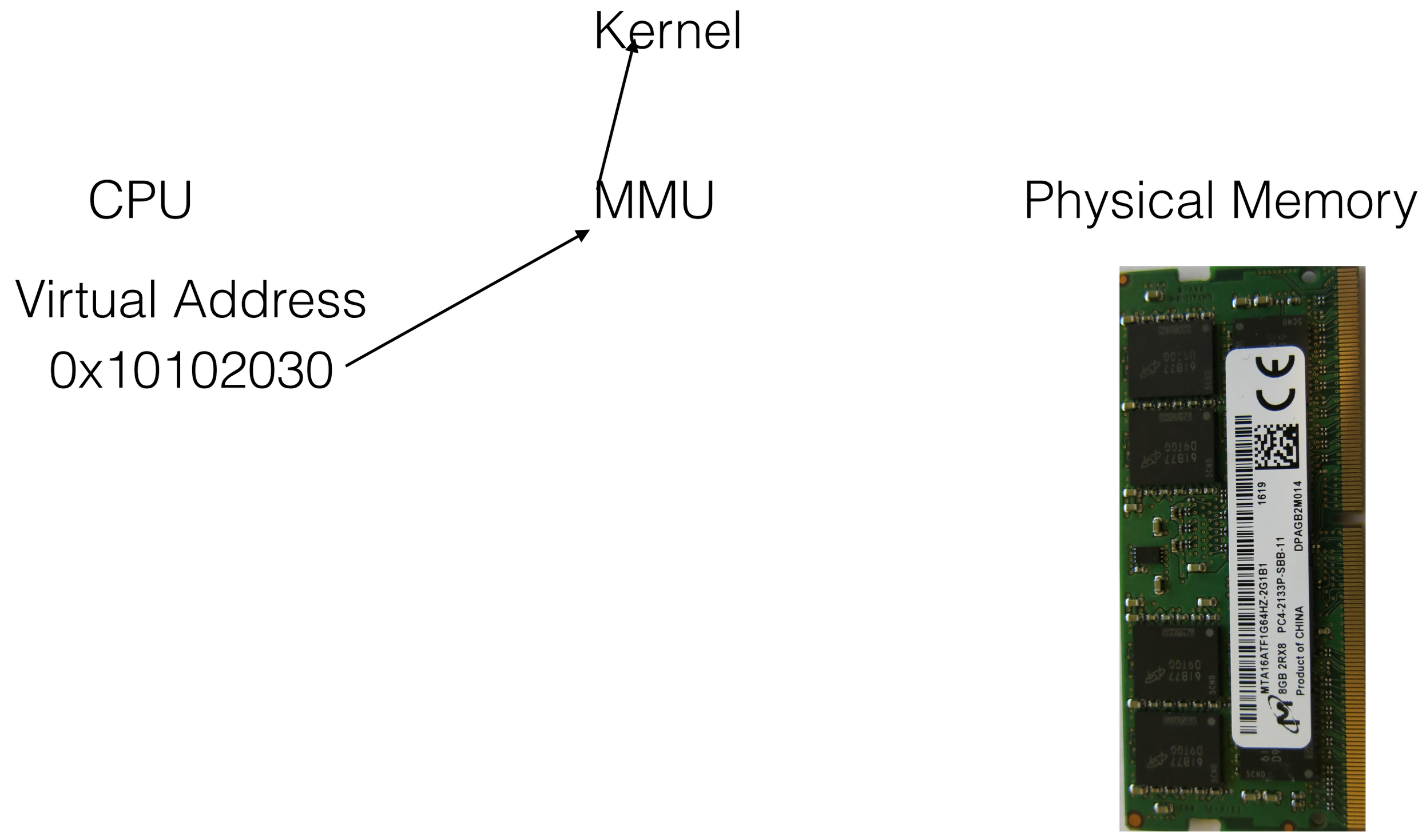
MMU

Physical Memory

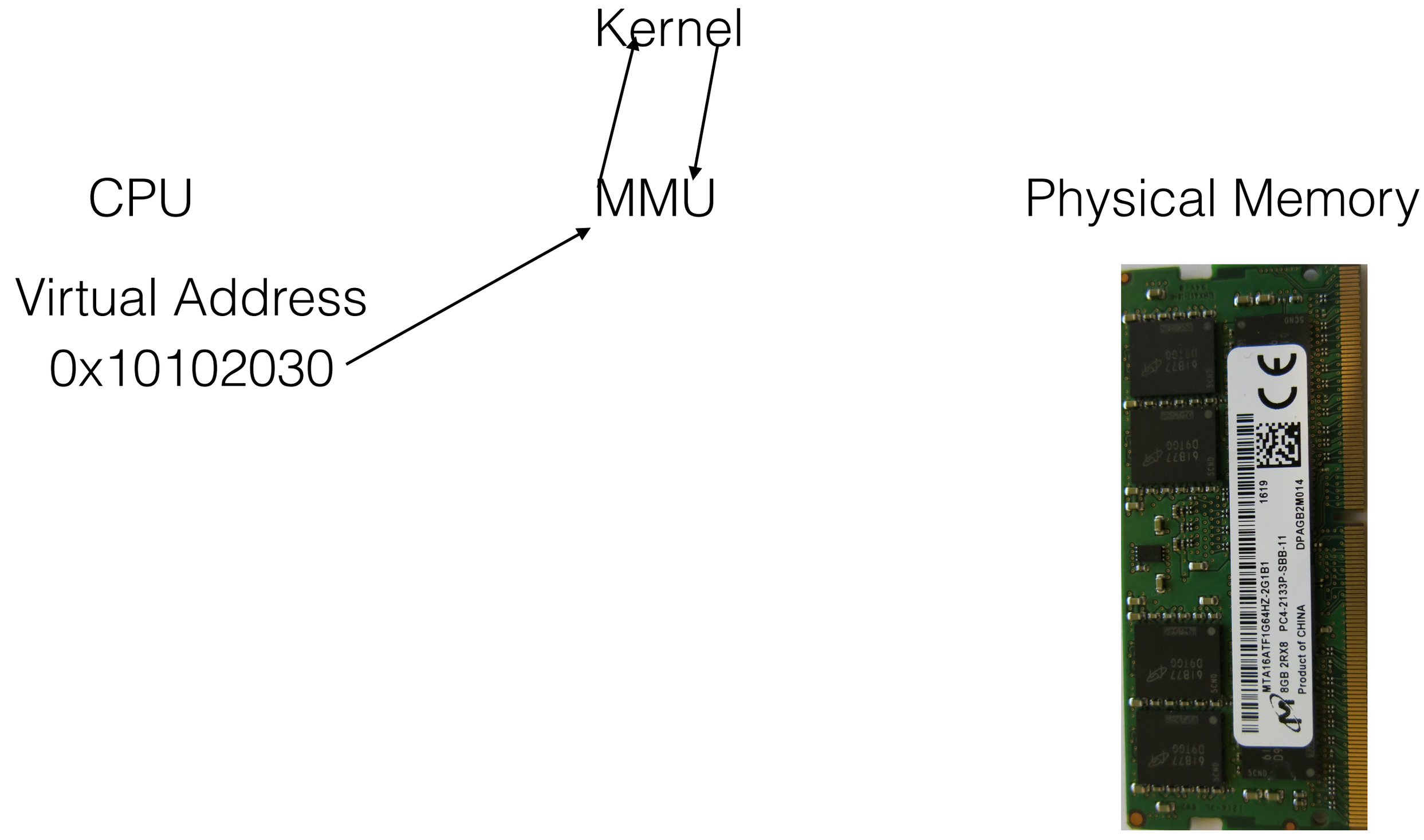
Virtual Address

0x10102030

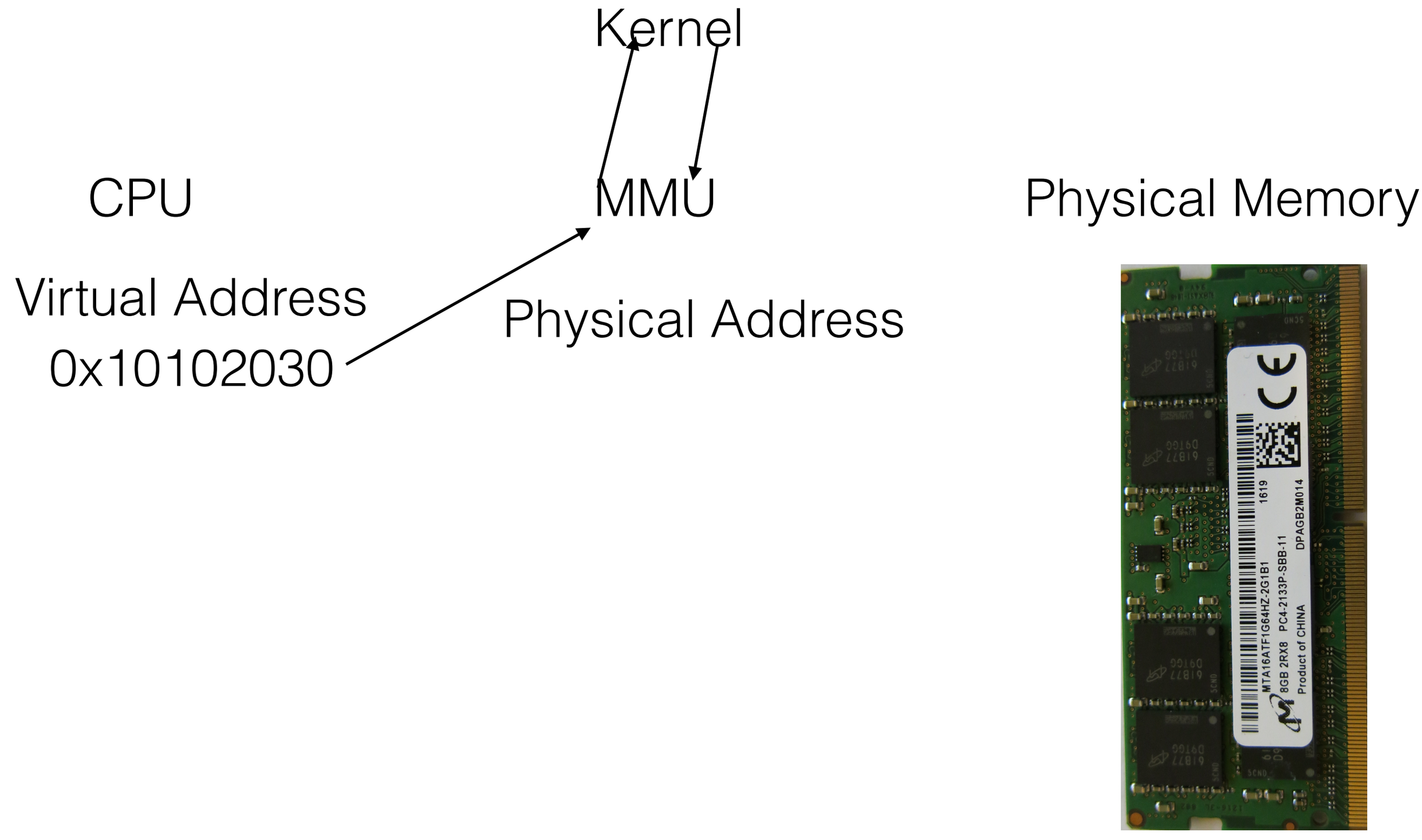
# General Address Translation



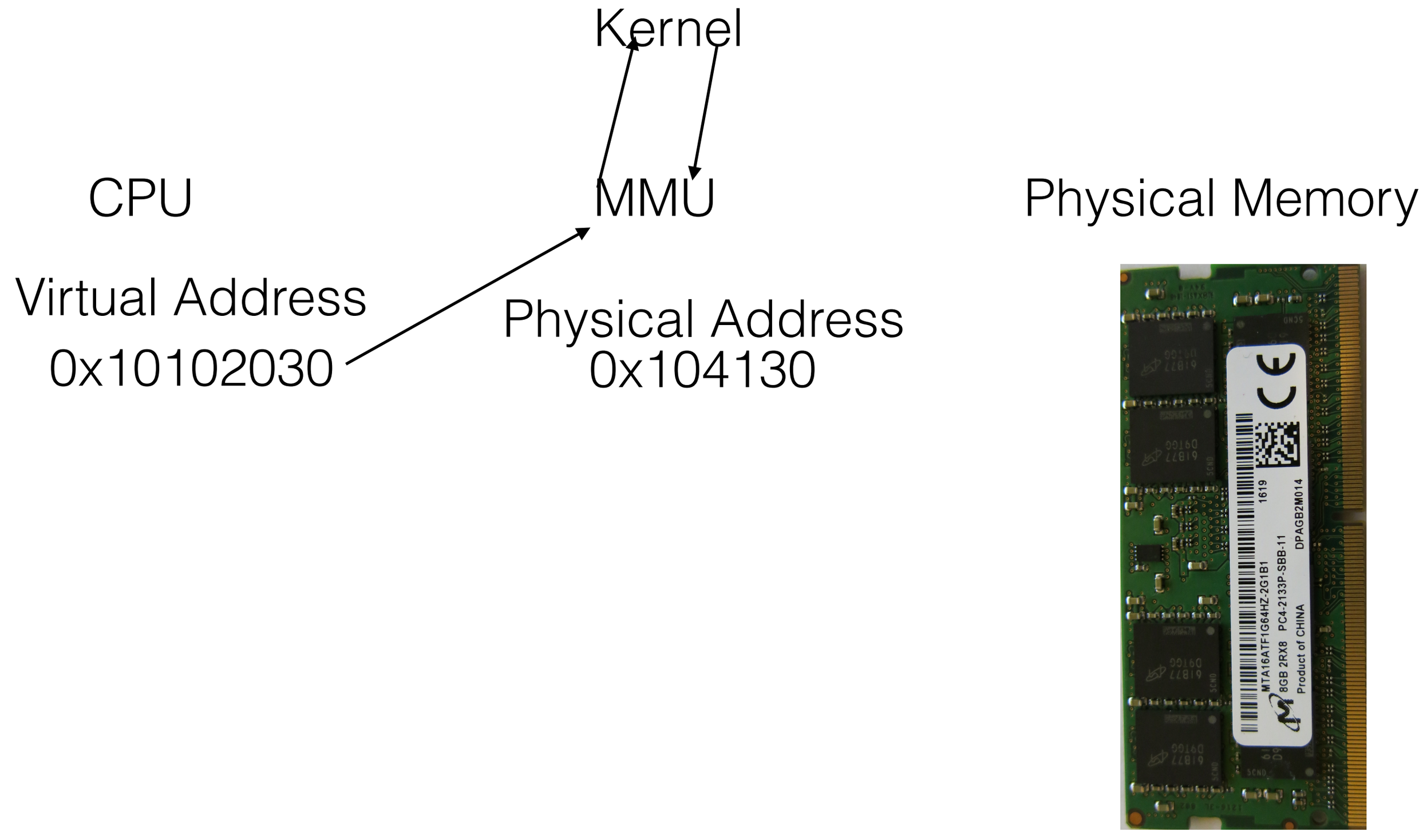
# General Address Translation



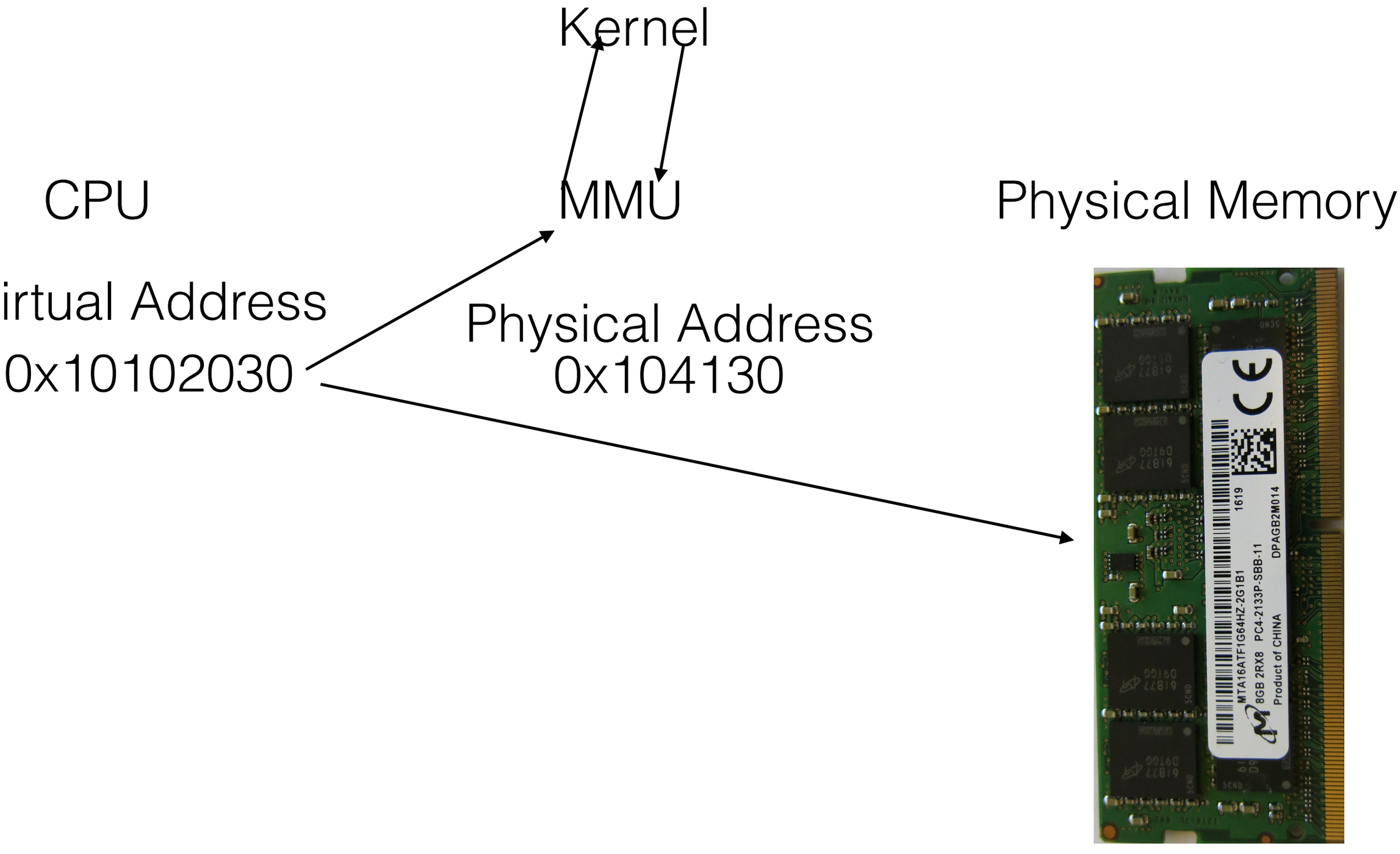
# General Address Translation



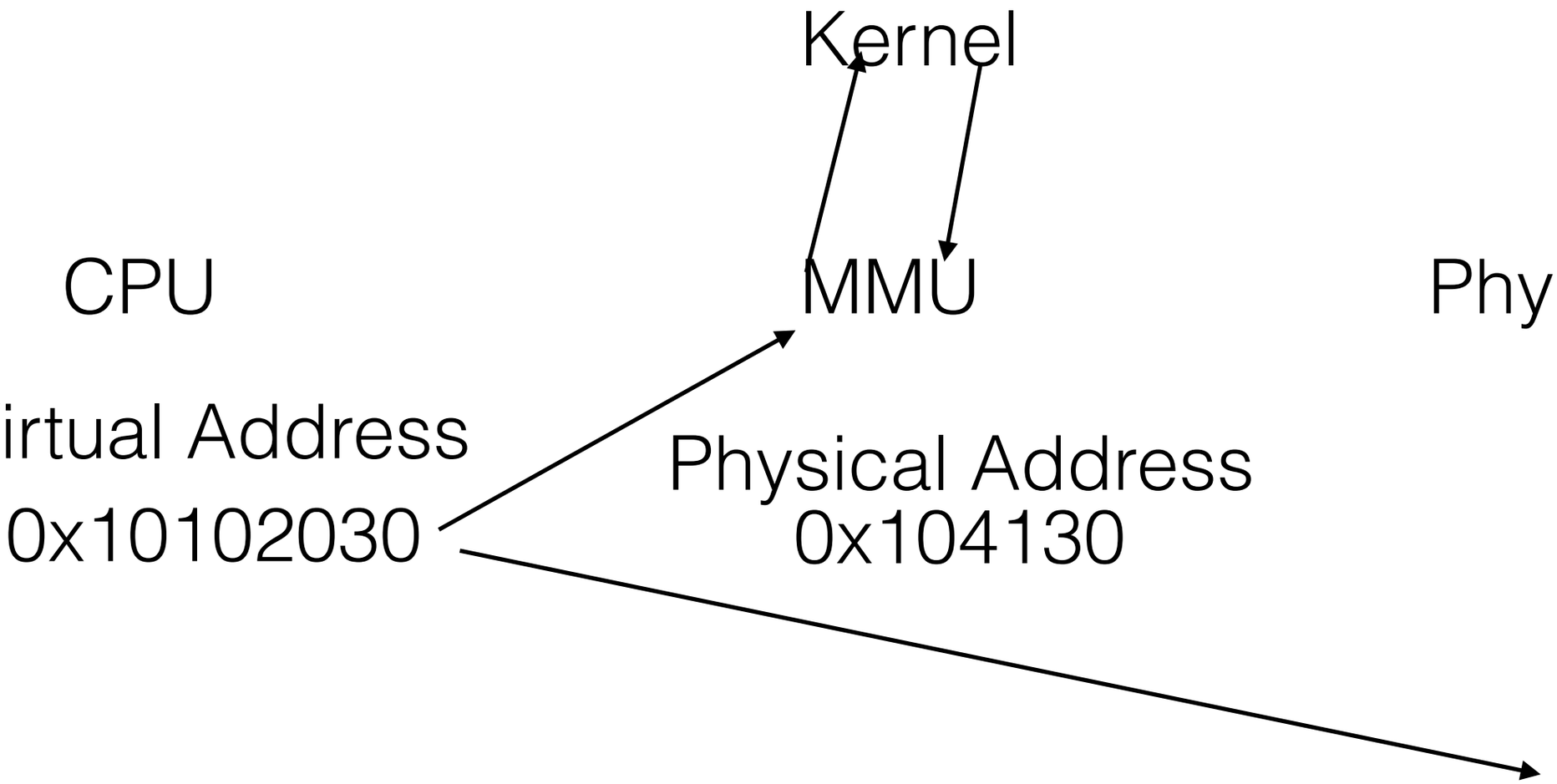
# General Address Translation



# General Address Translation



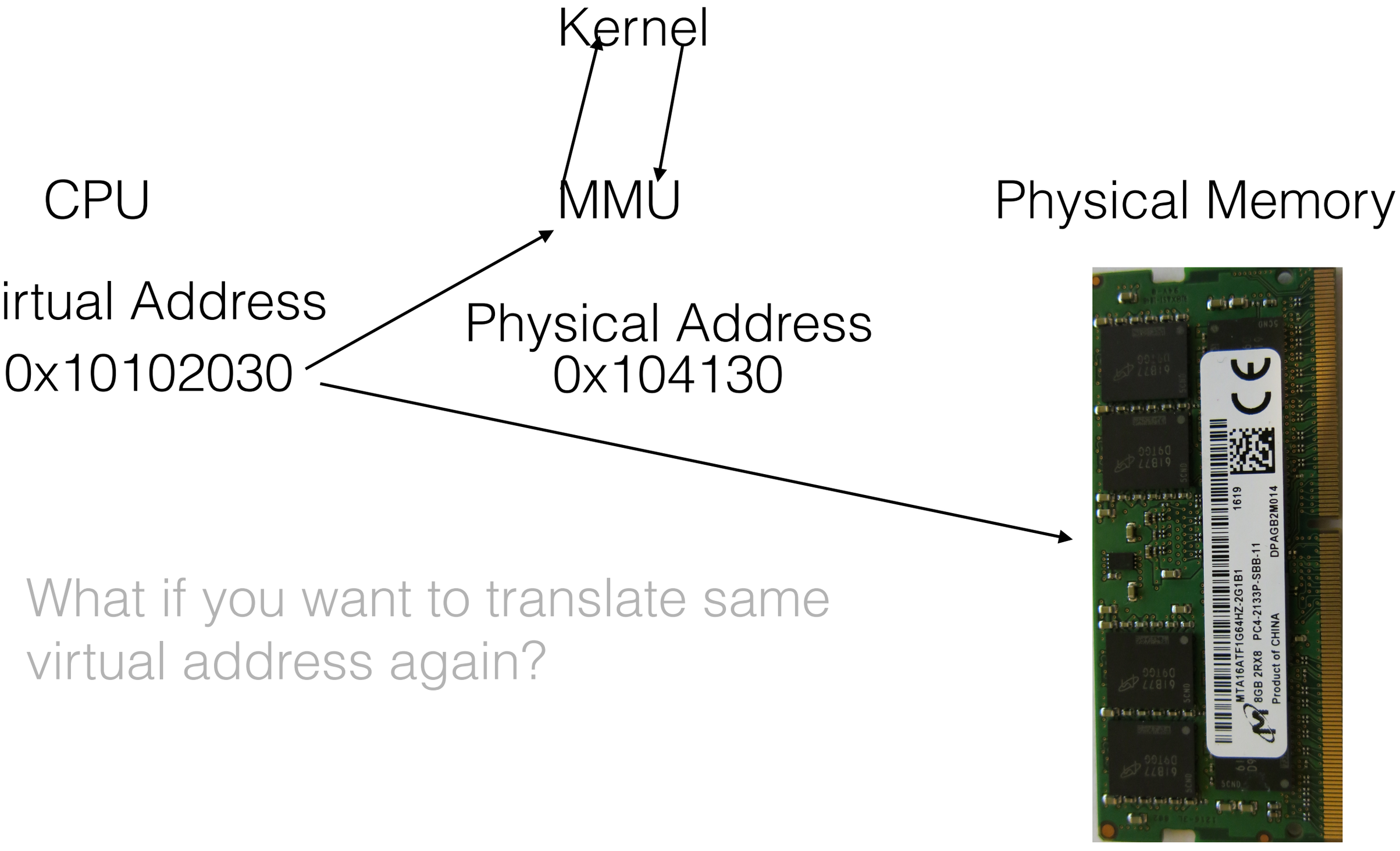
# General Address Translation





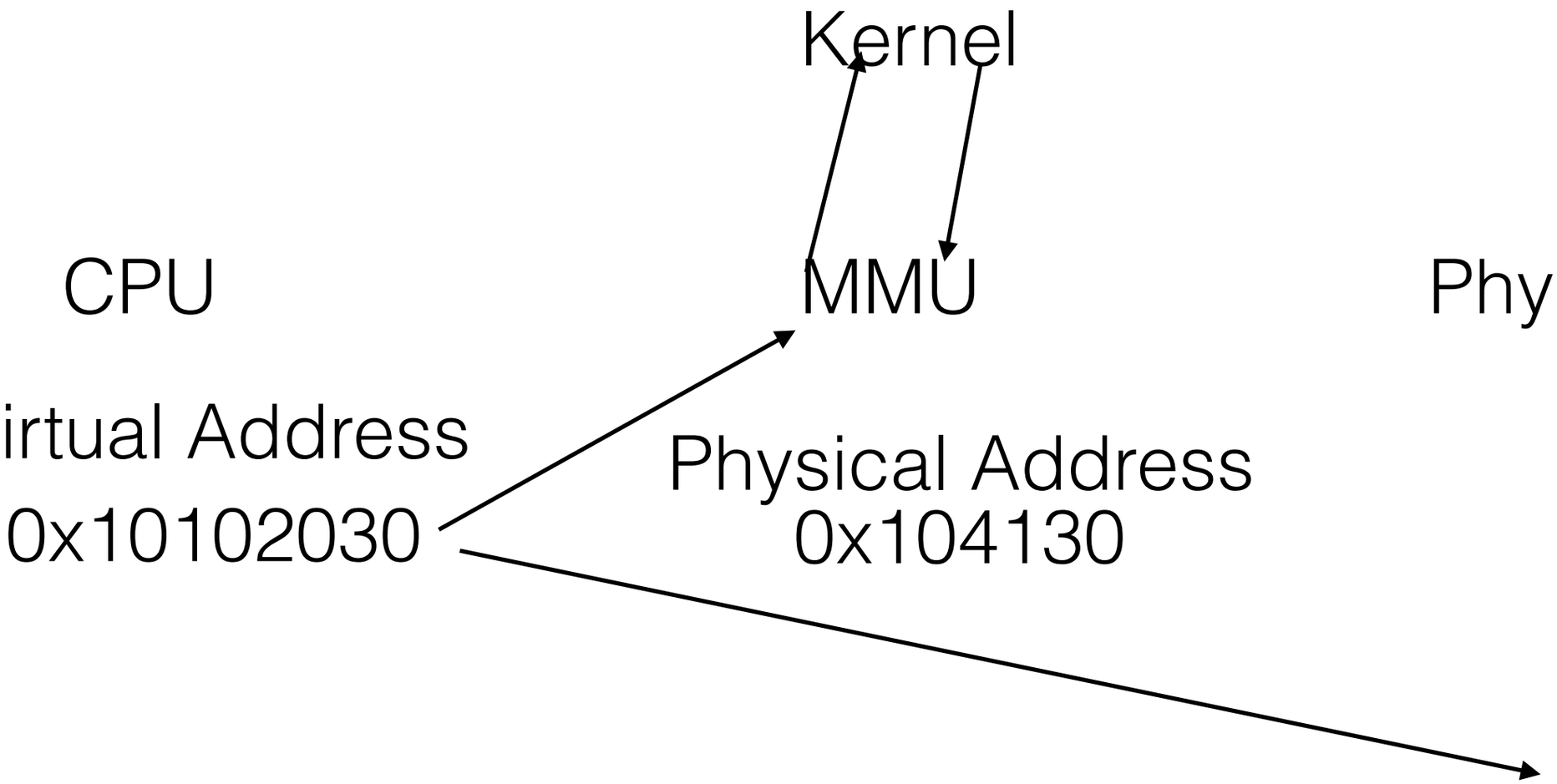
What if you want to translate same virtual address again?

# General Address Translation




What if you want to translate same virtual address again?

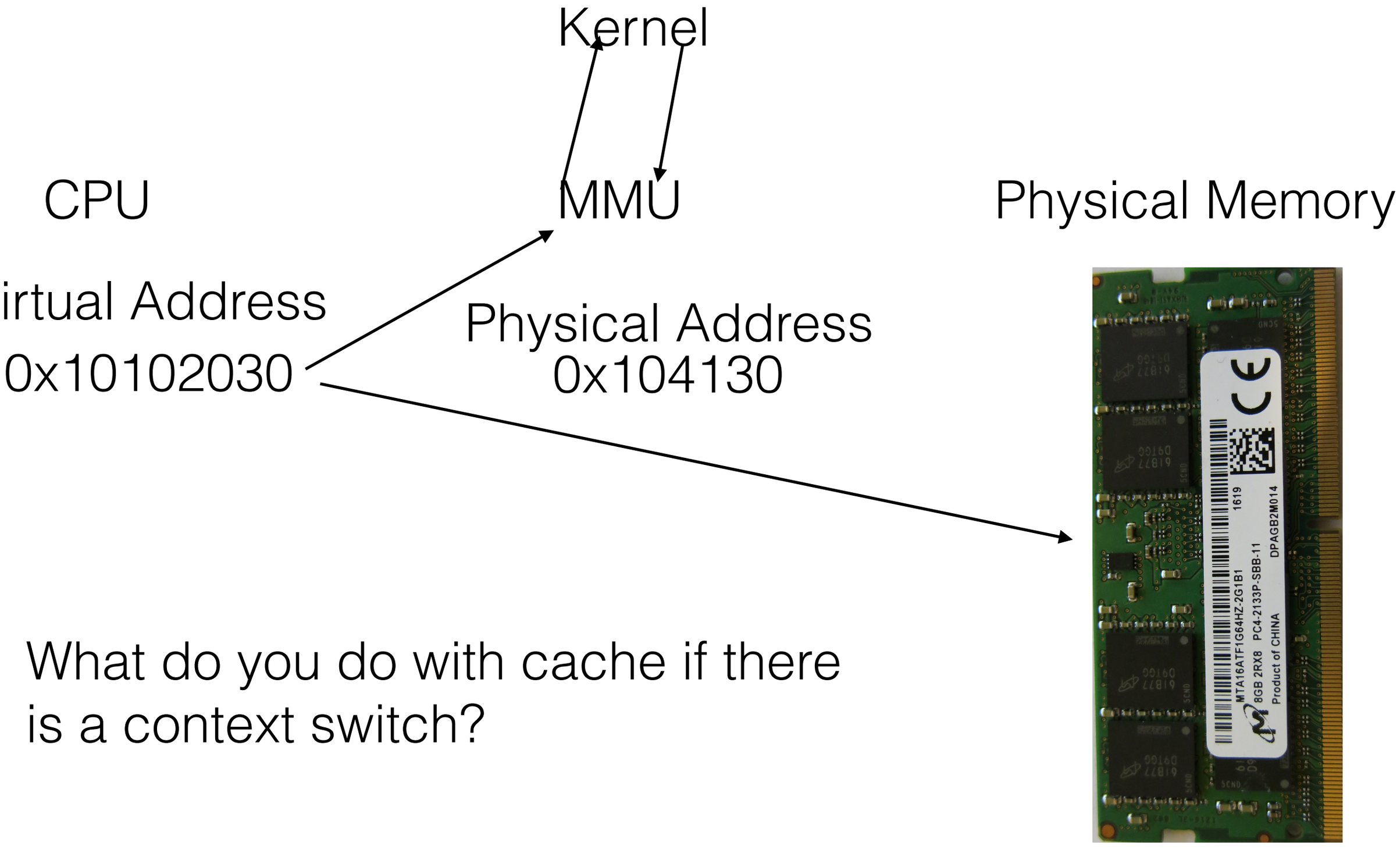
# General Address Translation




What if you want to translate same virtual address again?

Cache!!

# General Address Translation




What do you do with cache if there is a context switch?

# Dynamic (Hardware-based) Relocation
# Base & Bounds

Kernel

CPU

MMU

Physical Memory

# Dynamic (Hardware-based) Relocation
# Base & Bounds

Kernel

CPU

MMU

Physical Memory

Virtual Address

# Dynamic (Hardware-based) Relocation
# Base & Bounds

Kernel

CPU

MMU

Physical Memory

Virtual Address

0x100

# Dynamic (Hardware-based) Relocation
# Base & Bounds

Kernel

CPU

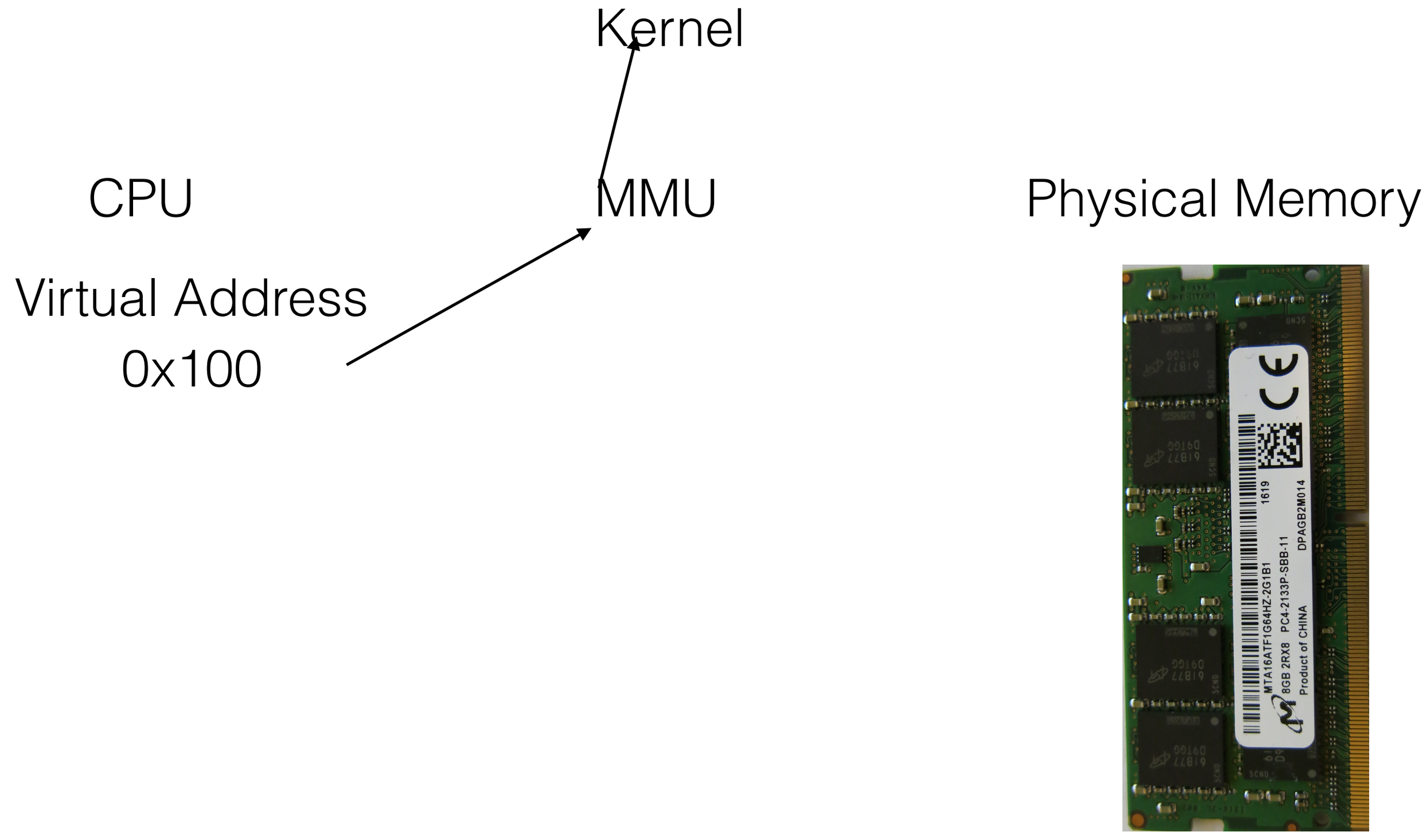
MMU

Physical Memory

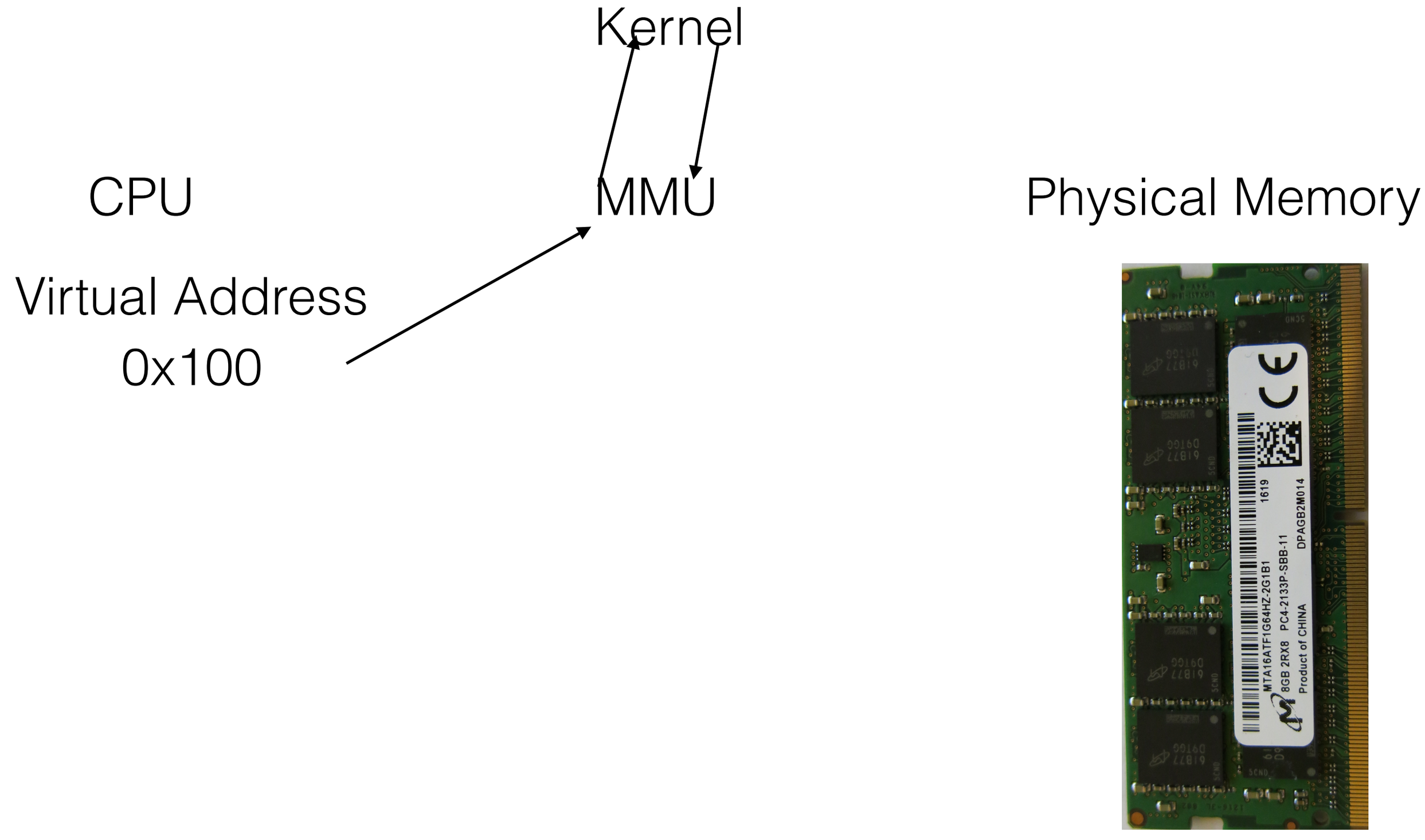
Virtual Address

0x100

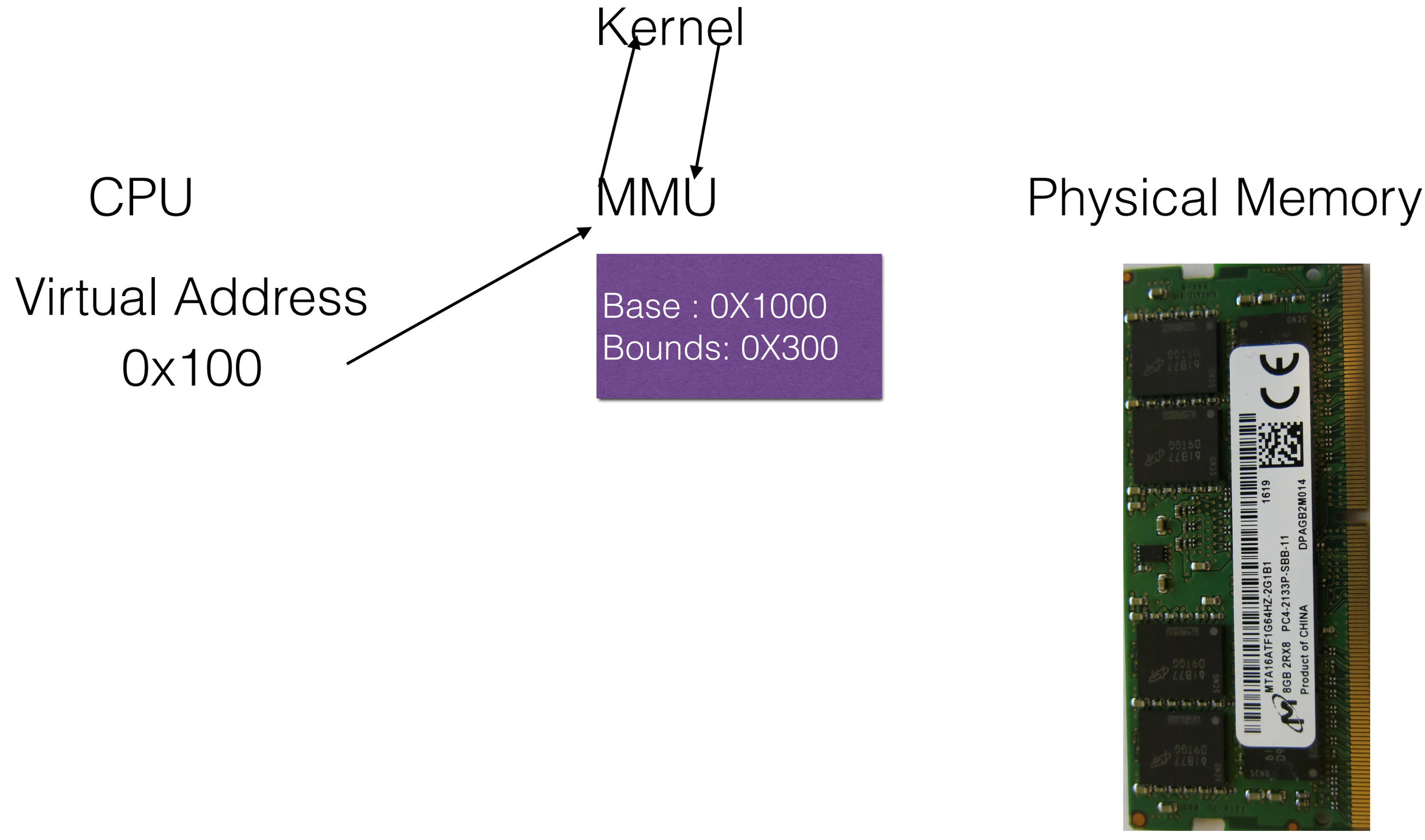
# Dynamic (Hardware-based) Relocation
# Base & Bounds



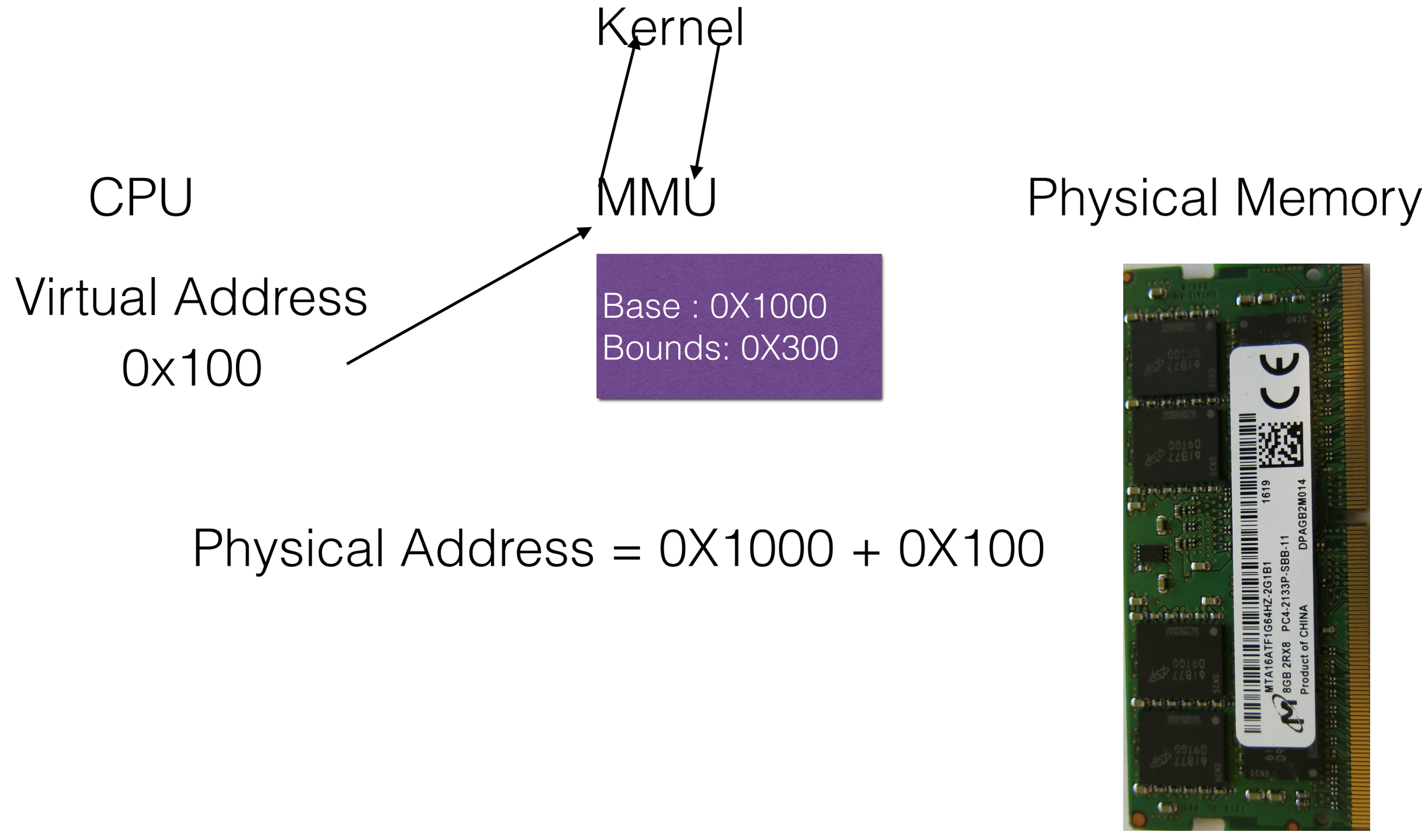
# Dynamic (Hardware-based) Relocation
# Base & Bounds



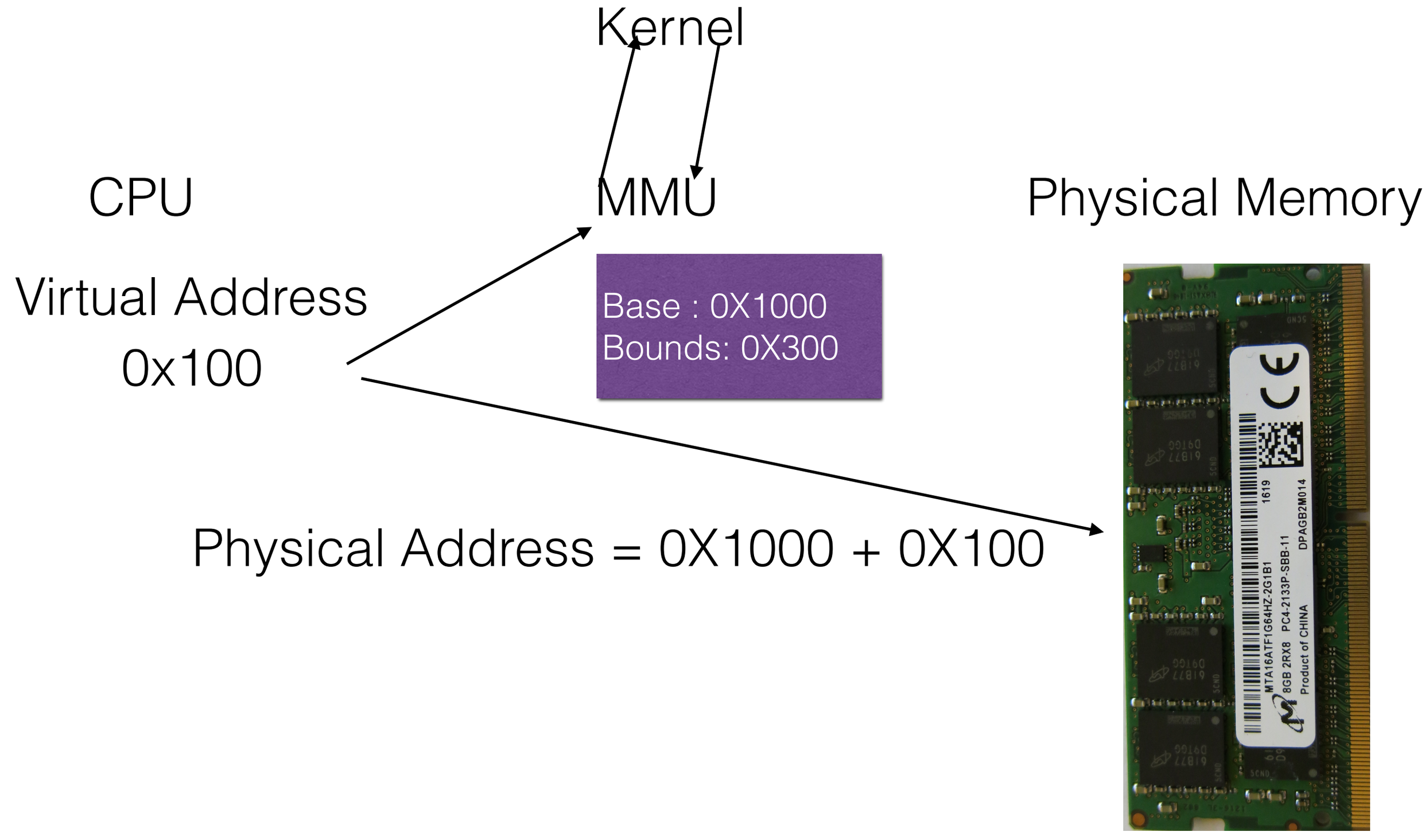
# Dynamic (Hardware-based) Relocation
# Base & Bounds



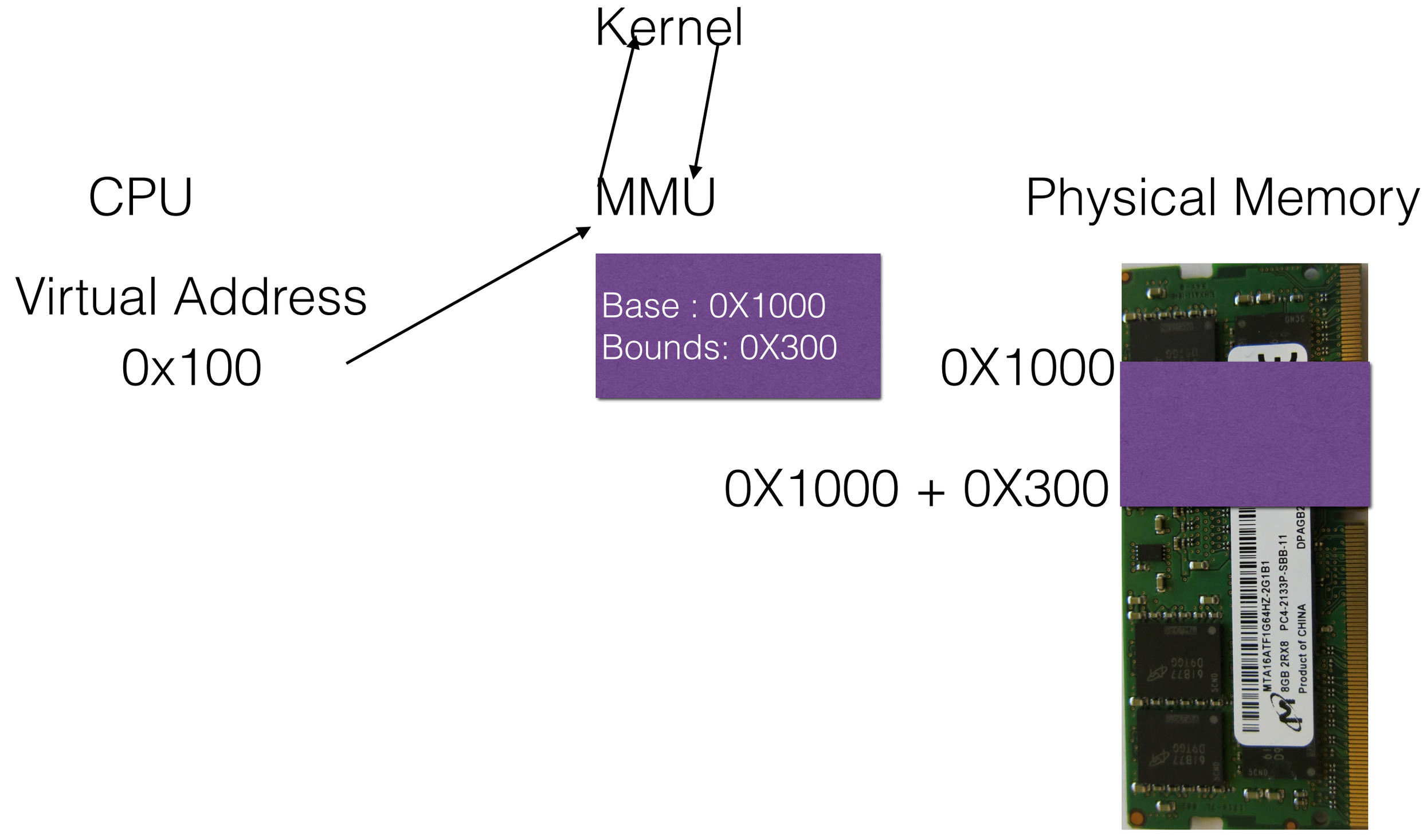
# Dynamic (Hardware-based) Relocation
# Base & Bounds



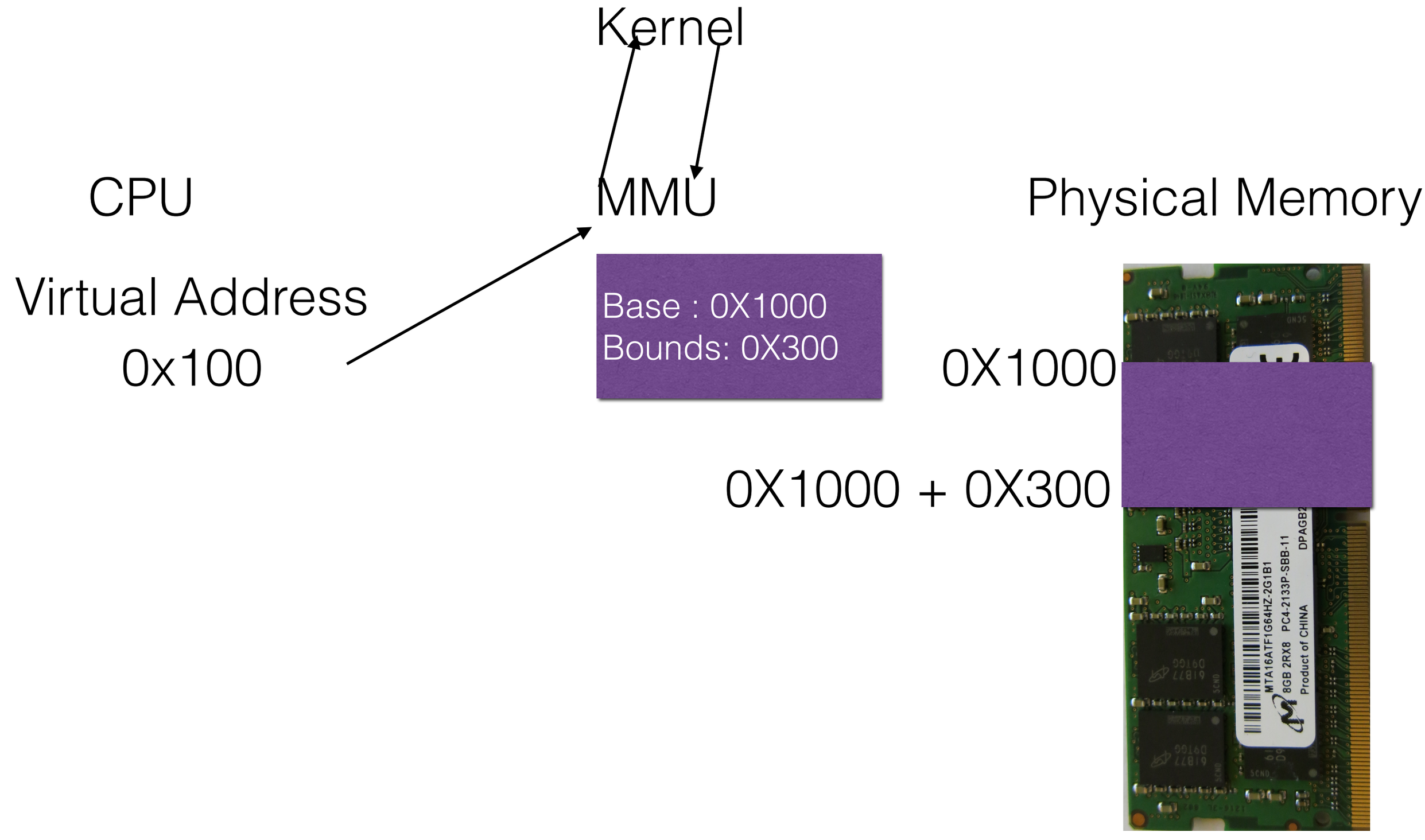
# Dynamic (Hardware-based) Relocation
# Base & Bounds



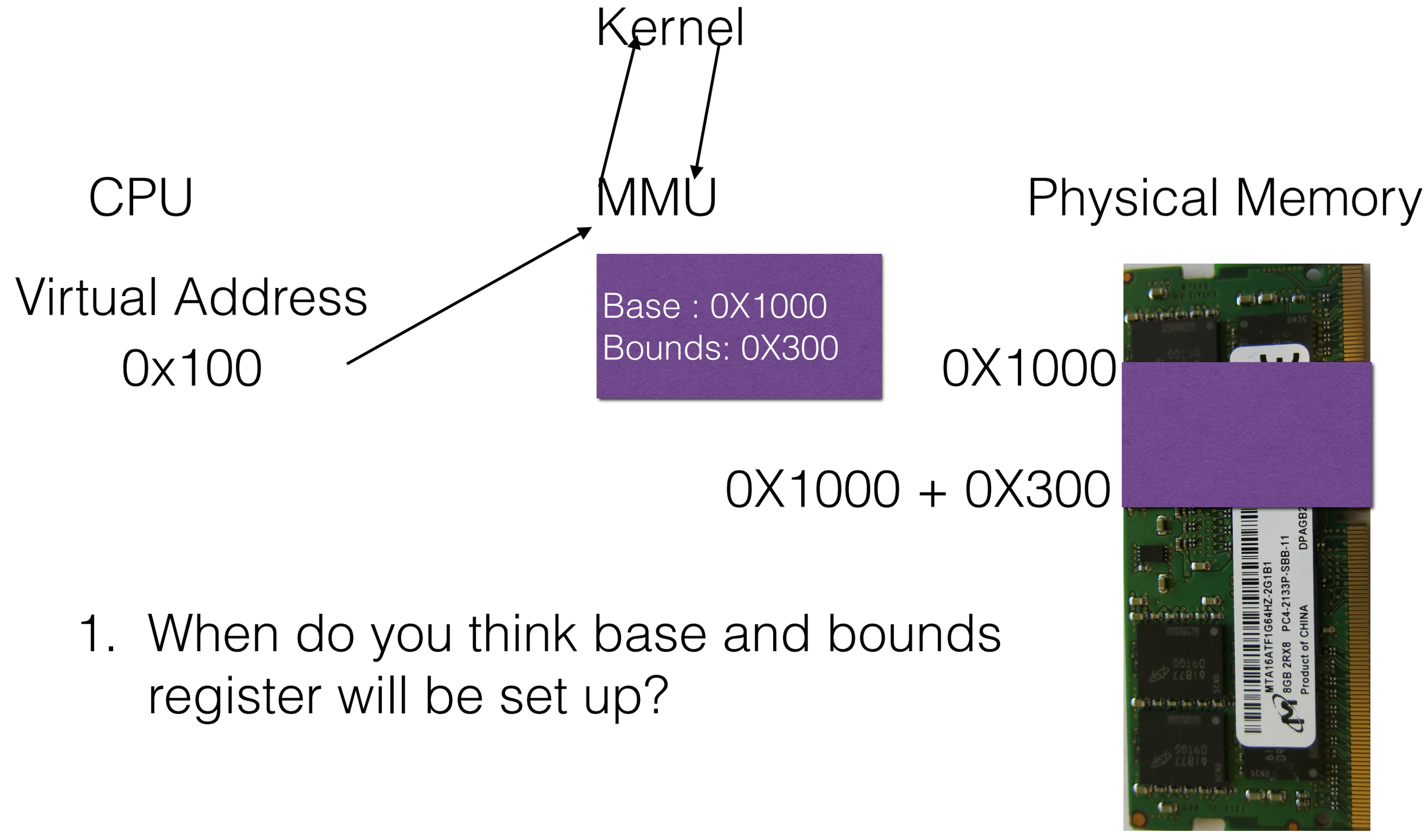
# Dynamic (Hardware-based) Relocation
# Base & Bounds



# Dynamic (Hardware-based) Relocation
# Base & Bounds

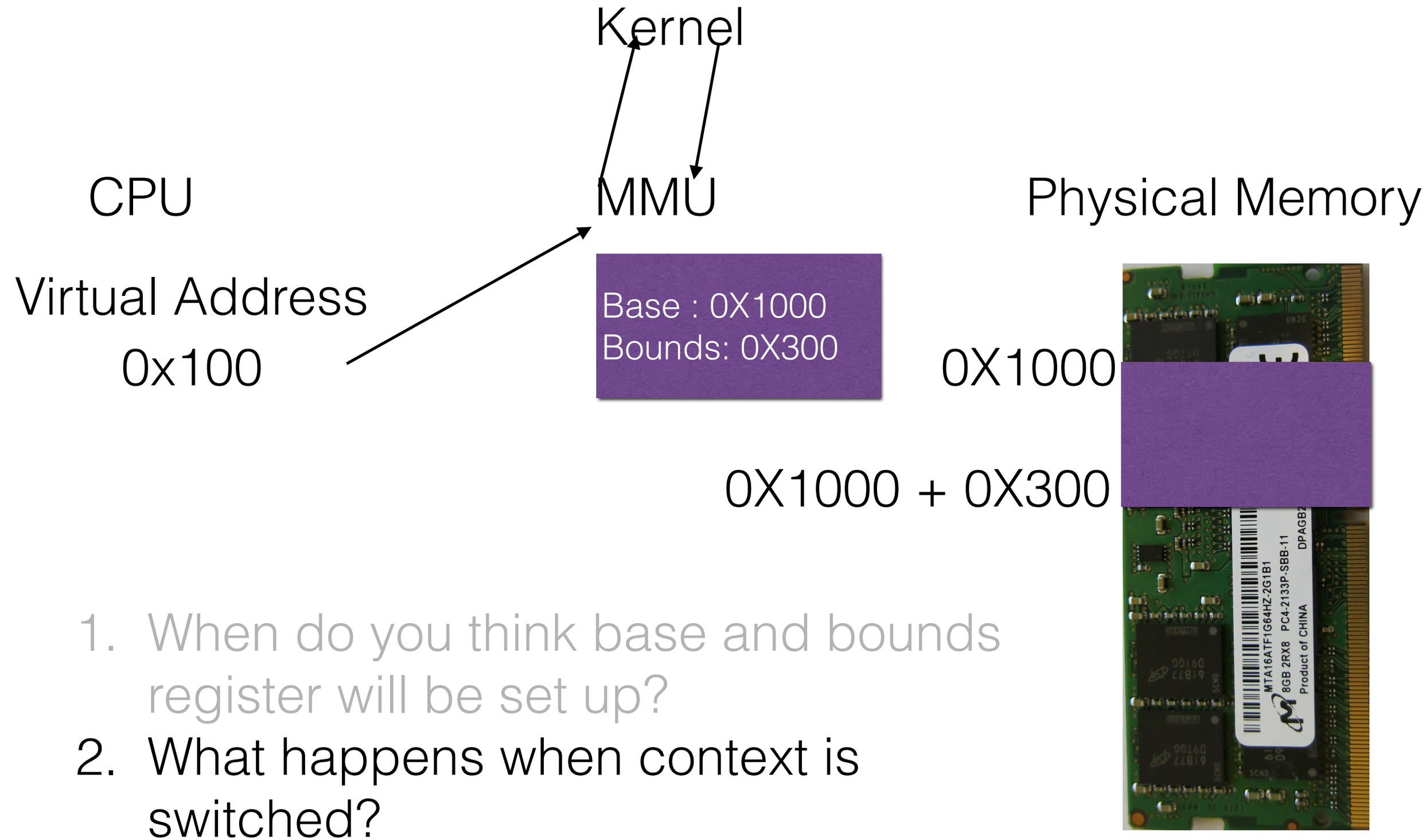


# Dynamic (Hardware-based) Relocation
# Base & Bounds




1. When do you think base and bounds register will be set up?

# Dynamic (Hardware-based) Relocation
# Base & Bounds




1. When do you think base and bounds register will be set up?
2. What happens when context is switched?

# Dynamic (Hardware-based) Relocation

# Dynamic (Hardware-based) Relocation

1. Base and Bound registers

# Dynamic (Hardware-based) Relocation

1. Base and Bound registers
2. Allows:

# Dynamic (Hardware-based) Relocation

1. Base and Bound registers
2. Allows:
    1. Place address space anywhere in memory (not just at location 0)

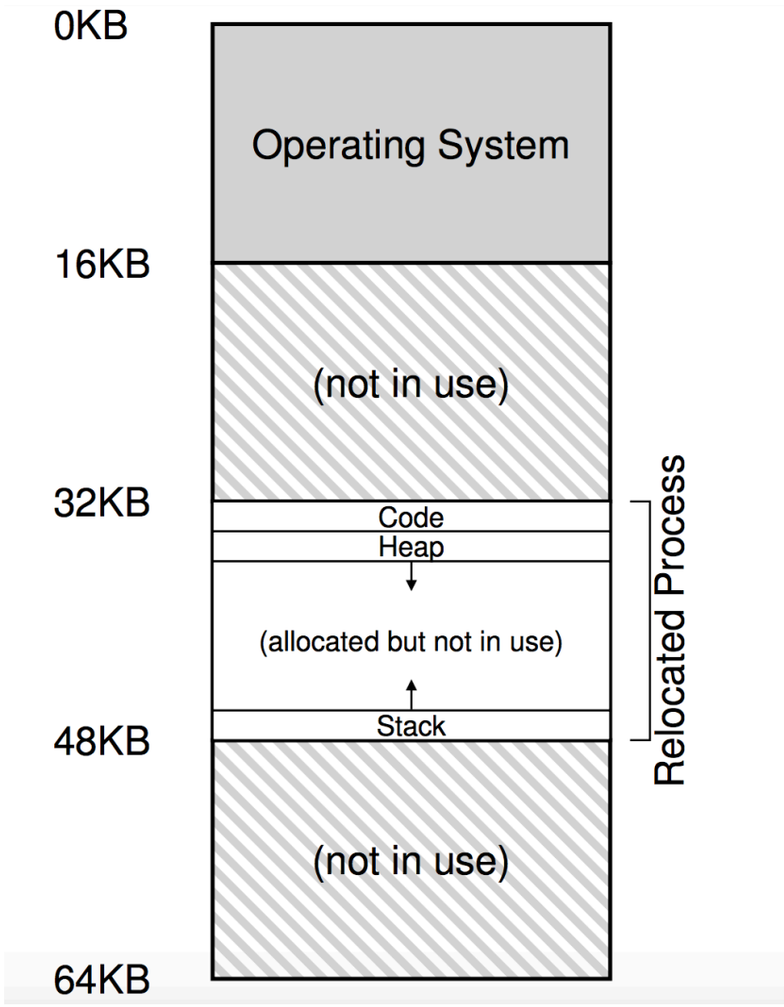
# Dynamic (Hardware-based) Relocation

1. Base and Bound registers
2. Allows:
    1. Place address space anywhere in memory (not just at location 0)
    2. Ensures process only accesses its own space

# Dynamic (Hardware-based) Relocation

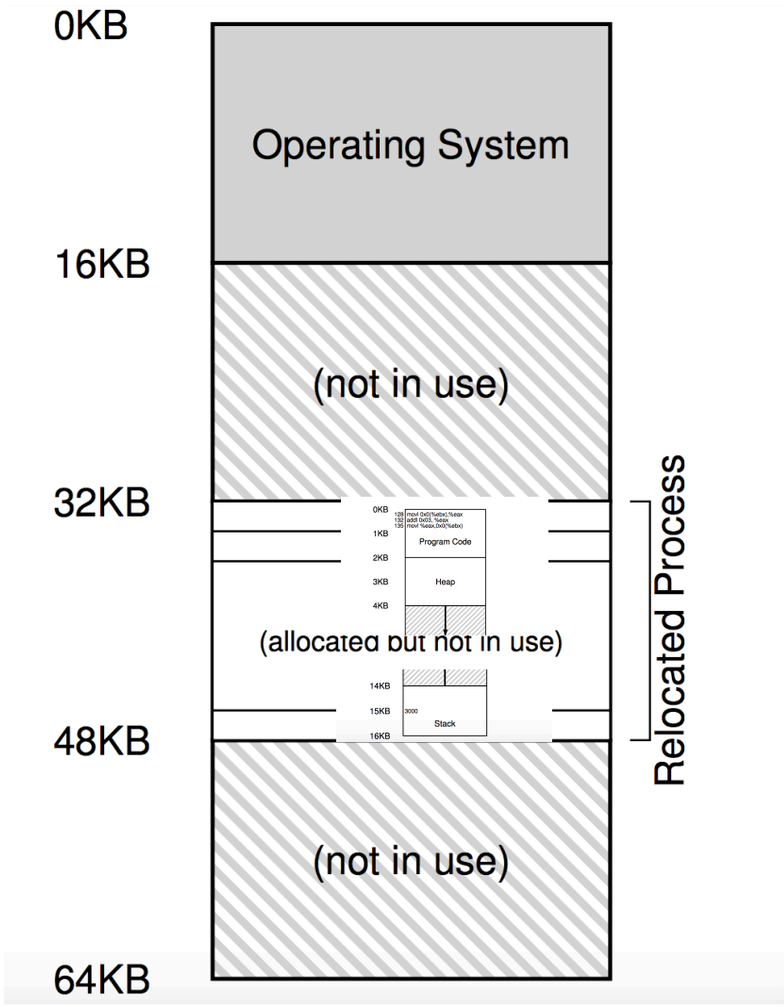
Physical Address = Virtual Address + Base

# Dynamic (Hardware-based) Relocation




Physical Address = Virtual Address + Base

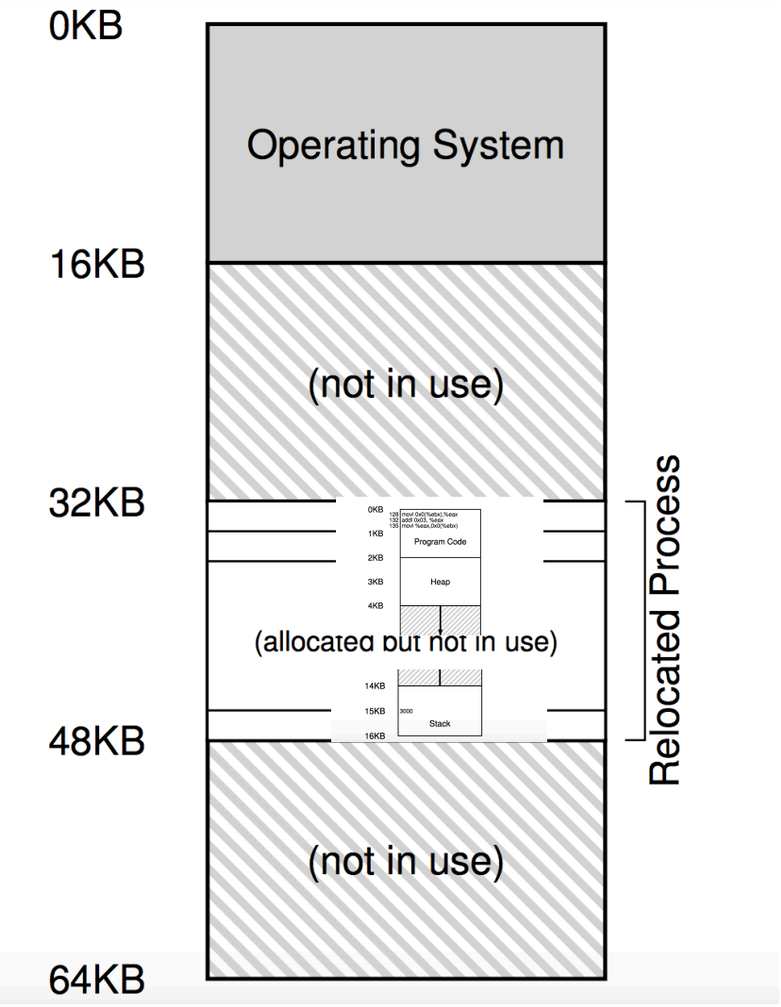
# Dynamic (Hardware-based) Relocation




Physical Address = Virtual Address + Base

# Dynamic (Hardware-based) Relocation

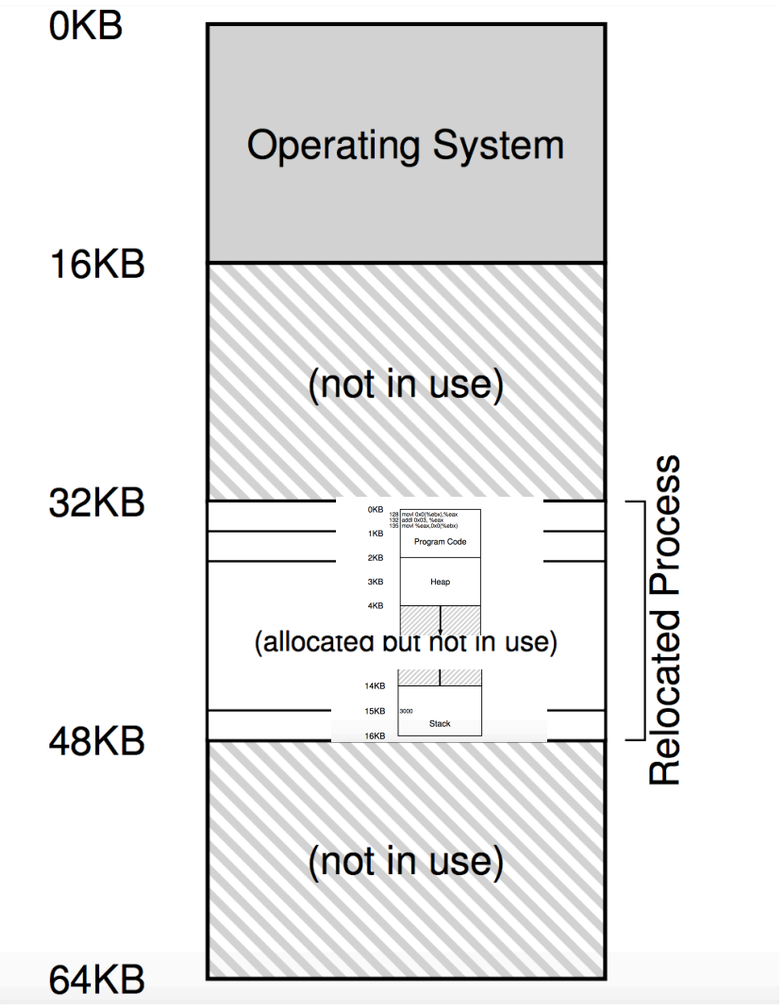
1. Base




Physical Address = Virtual Address + Base

# Dynamic (Hardware-based) Relocation

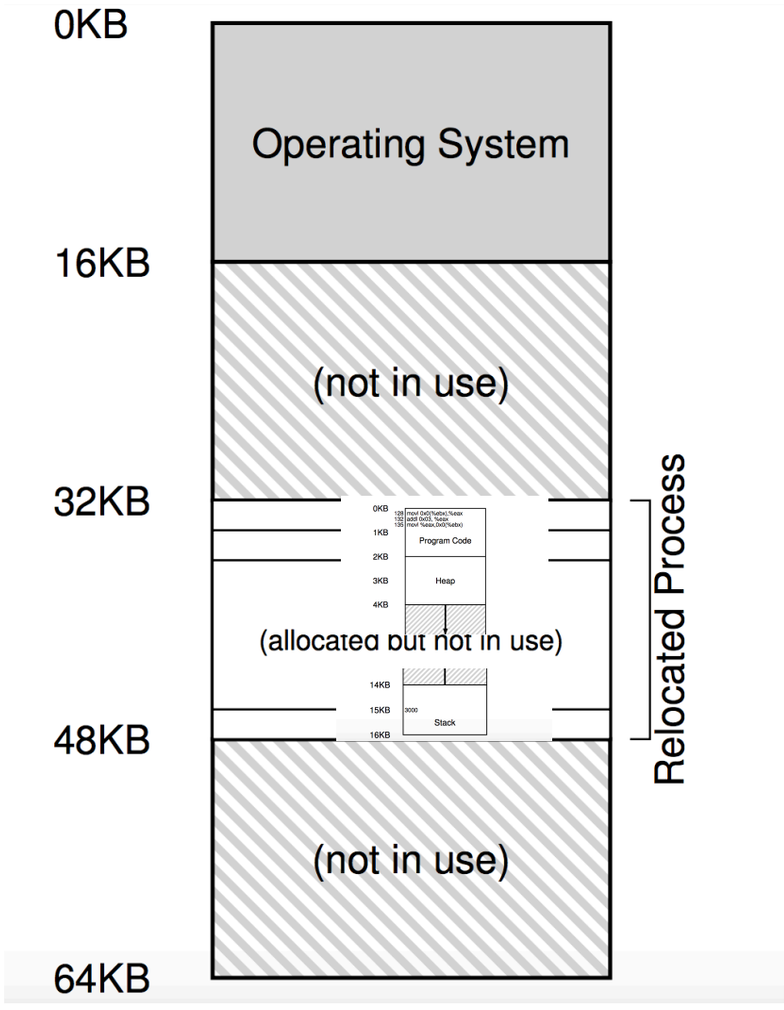
1. Base
    1. OS decides where in physical address to load the address space




**Physical Address = Virtual Address + Base**

# Dynamic (Hardware-based) Relocation

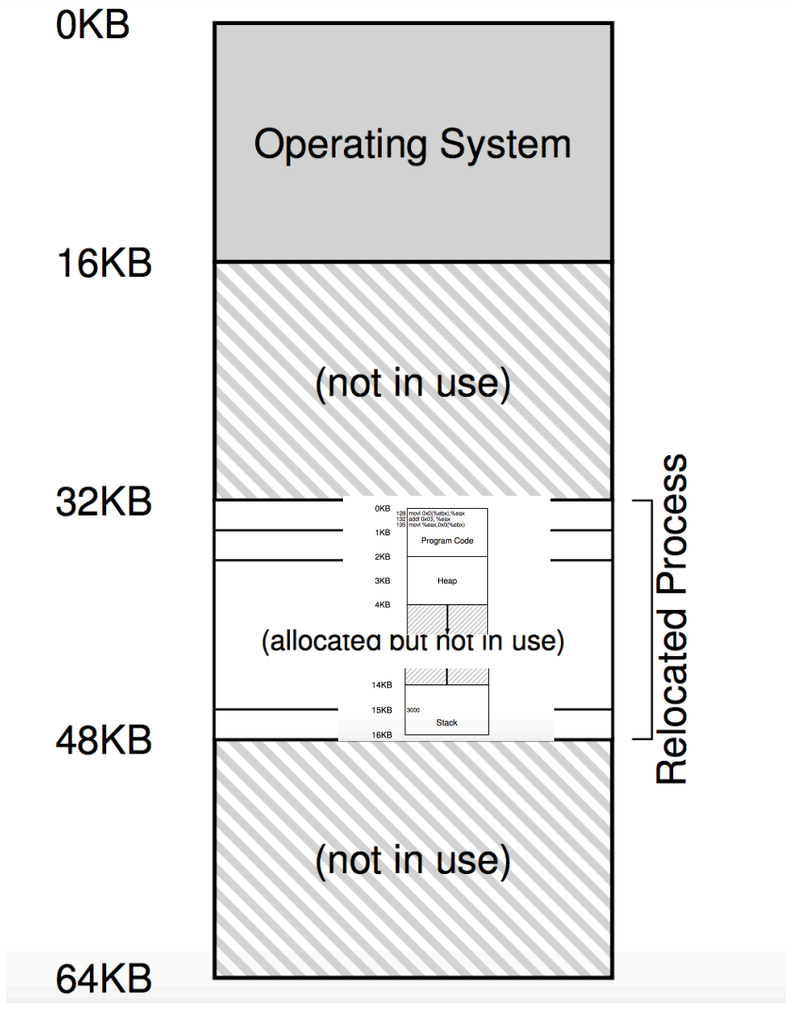
1. Base
    1. OS decides where in physical address to load the address space
    2. In previous example, base is …?




Physical Address = Virtual Address + Base

# Dynamic (Hardware-based) Relocation

1. Base
    1. OS decides where in physical address to load the address space
    2. In previous example, base is …?
    3. Ans: 32 KB




Physical Address = Virtual Address + Base

# Example Revisited

```
128: movl 0x0(%ebx), %eax
```

# Example Revisited

```
128: movl 0x0(%ebx), %eax
```

1. PC points to 128

# Example Revisited

```
128: movl 0x0(%ebx), %eax
```

1. PC points to 128
2. Fetch instruction :

# Example Revisited

128: movl 0x0(%ebx), %eax

1. PC points to 128
2. Fetch instruction :
    1. Physical Address = Base + Virtual Address = 32K + 128 = 32896

# Example Revisited

128: movl 0x0(%ebx), %eax

1. PC points to 128
2. Fetch instruction :
    1. Physical Address = Base + Virtual Address = 32K + 128 = 32896
3. Execute load:

# Example Revisited

128: movl 0x0(%ebx), %eax

1. PC points to 128
2. Fetch instruction :
    1. Physical Address = Base + Virtual Address = 32K + 128 = 32896
3. Execute load:
    1. Data resides in Virtual Address 15 KB

# Example Revisited

128: movl 0x0(%ebx), %eax

1. PC points to 128
2. Fetch instruction :
    1. Physical Address = Base + Virtual Address = 32K + 128 = 32896
3. Execute load:
    1. Data resides in Virtual Address 15 KB
    2. Fetch data from 32 KB + 15 KB = 47 KB

# Example Revisited

128: movl 0x0(%ebx), %eax

1. PC points to 128
2. Fetch instruction :
    1. Physical Address = Base + Virtual Address = 32K + 128 = 32896
3. Execute load:
    1. Data resides in Virtual Address 15 KB
    2. Fetch data from 32 KB + 15 KB = 47 KB
4. Pop quiz - why is it called dynamic relocation?

# Example Revisited

128: movl 0x0(%ebx), %eax

1. PC points to 128
2. Fetch instruction :
    1. Physical Address = Base + Virtual Address = 32K + 128 = 32896
3. Execute load:
    1. Data resides in Virtual Address 15 KB
    2. Fetch data from 32 KB + 15 KB = 47 KB
4. Pop quiz - why is it called dynamic relocation?
    1. Relocation happens at runtime

# Example Revisited

128: movl 0x0(%ebx), %eax

1. PC points to 128
2. Fetch instruction :
    1. Physical Address = Base + Virtual Address = 32K + 128 = 32896
3. Execute load:
    1. Data resides in Virtual Address 15 KB
    2. Fetch data from 32 KB + 15 KB = 47 KB
4. Pop quiz - why is it called dynamic relocation?
    1. Relocation happens at runtime
    2. Can change the address even after creation

# Example Revisited

128: movl 0x0(%ebx), %eax

1. PC points to 128
2. Fetch instruction :
    1. Physical Address = Base + Virtual Address = 32K + 128 = 32896
3. Execute load:
    1. Data resides in Virtual Address 15 KB
    2. Fetch data from 32 KB + 15 KB = 47 KB
4. Pop quiz - why is it called dynamic relocation?
    1. Relocation happens at runtime
    2. Can change the address even after creation
    3. Why would you do that? How would you do that?

# Example Revisited

128: movl 0x0(%ebx), %eax

1. PC points to 128
2. Fetch instruction :
    1. Physical Address = Base + Virtual Address = 32K + 128 = 32896
3. Execute load:
    1. Data resides in Virtual Address 15 KB
    2. Fetch data from 32 KB + 15 KB = 47 KB
4. Pop quiz - why is it called dynamic relocation?
    1. Relocation happens at runtime
    2. Can change the address even after creation
    3. Why would you do that? How would you do that?

# Example Revisited

128: movl 0x0(%ebx), %eax

1. PC points to 128
2. Fetch instruction :
    1. Physical Address = Base + Virtual Address = 32K + 128 = 32896
3. Execute load:
    1. Data resides in Virtual Address 15 KB
    2. Fetch data from 32 KB + 15 KB = 47 KB
4. Pop quiz - why is it called dynamic relocation?
    1. Relocation happens at runtime
    2. Can change the address even after creation
    3. Why would you do that? How would you do that?

# Bounds register

## Goals of OS for Memory Virtualisation

1. Transparency
    1. Virtual memory is invisible to user program
    2. Program thinks it has own private large memory
2. Efficiency
    1. Not taking very long
    2. Not taking too much space
3. Protection/Isolation
    1. Protect processes from each other

# Bounds register

## Goals of OS for Memory Virtualisation

1. Transparency
    1. Virtual memory is invisible to user program
    2. Program thinks it has own private large memory
2. Efficiency
    1. Not taking very long
    2. Not taking too much space
3. Protection/Isolation
    1. Protect processes from each other

# Bounds register

# Bounds register

1. Checks if memory reference is within bounds

# Bounds register

1. Checks if memory reference is within bounds
2. Pop Quiz: What's the bound in our example?

# Bounds register

1. Checks if memory reference is within bounds
2. Pop Quiz: What's the bound in our example?
    1. Ans : 16 KB

# Bounds register

1. Checks if memory reference is within bounds
2. Pop Quiz: What's the bound in our example?
    1. Ans : 16 KB
3. Incorrect virtual address : Terminate!

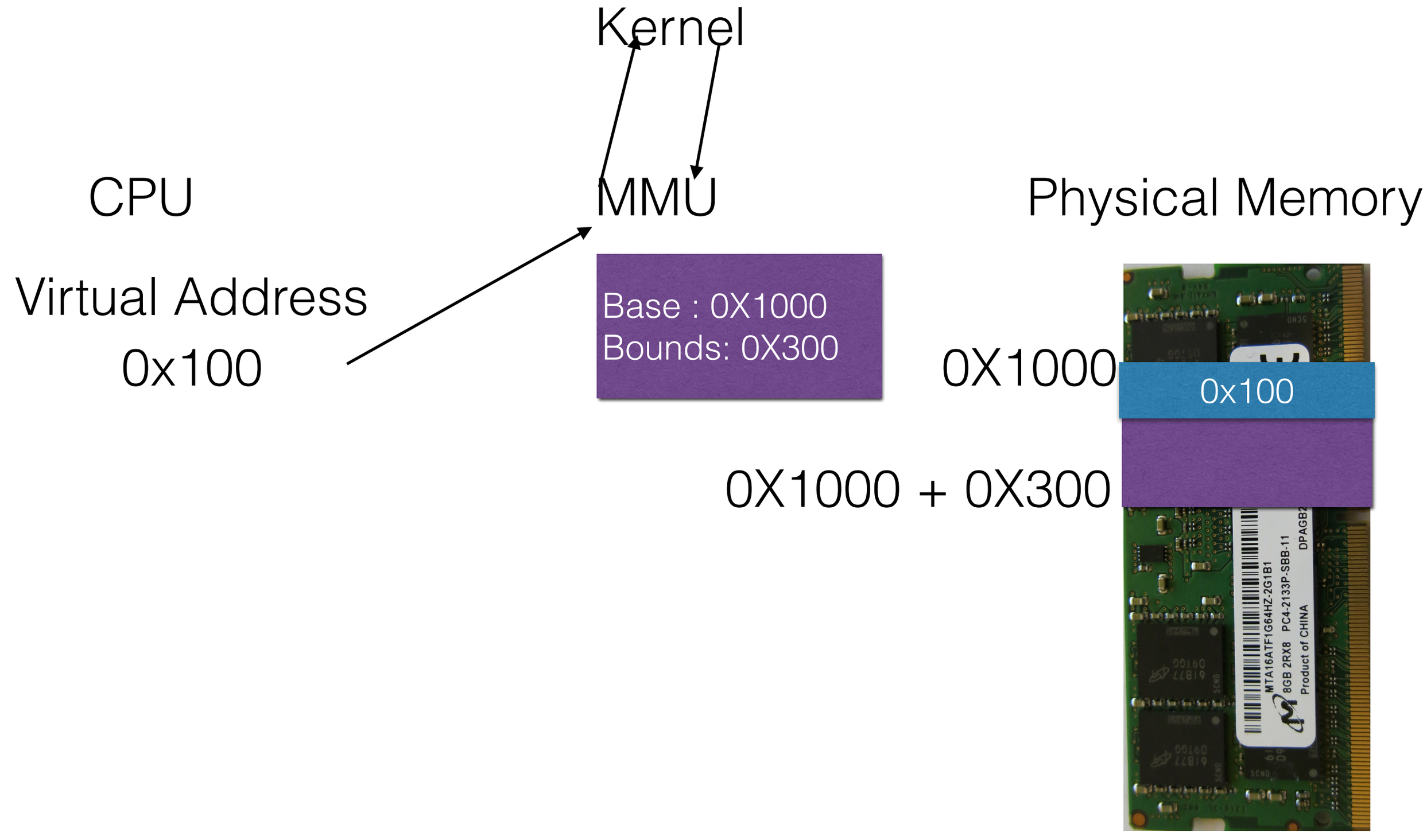
# Bounds register

1. Checks if memory reference is within bounds
2. Pop Quiz: What's the bound in our example?
    1. Ans : 16 KB
3. Incorrect virtual address : Terminate!

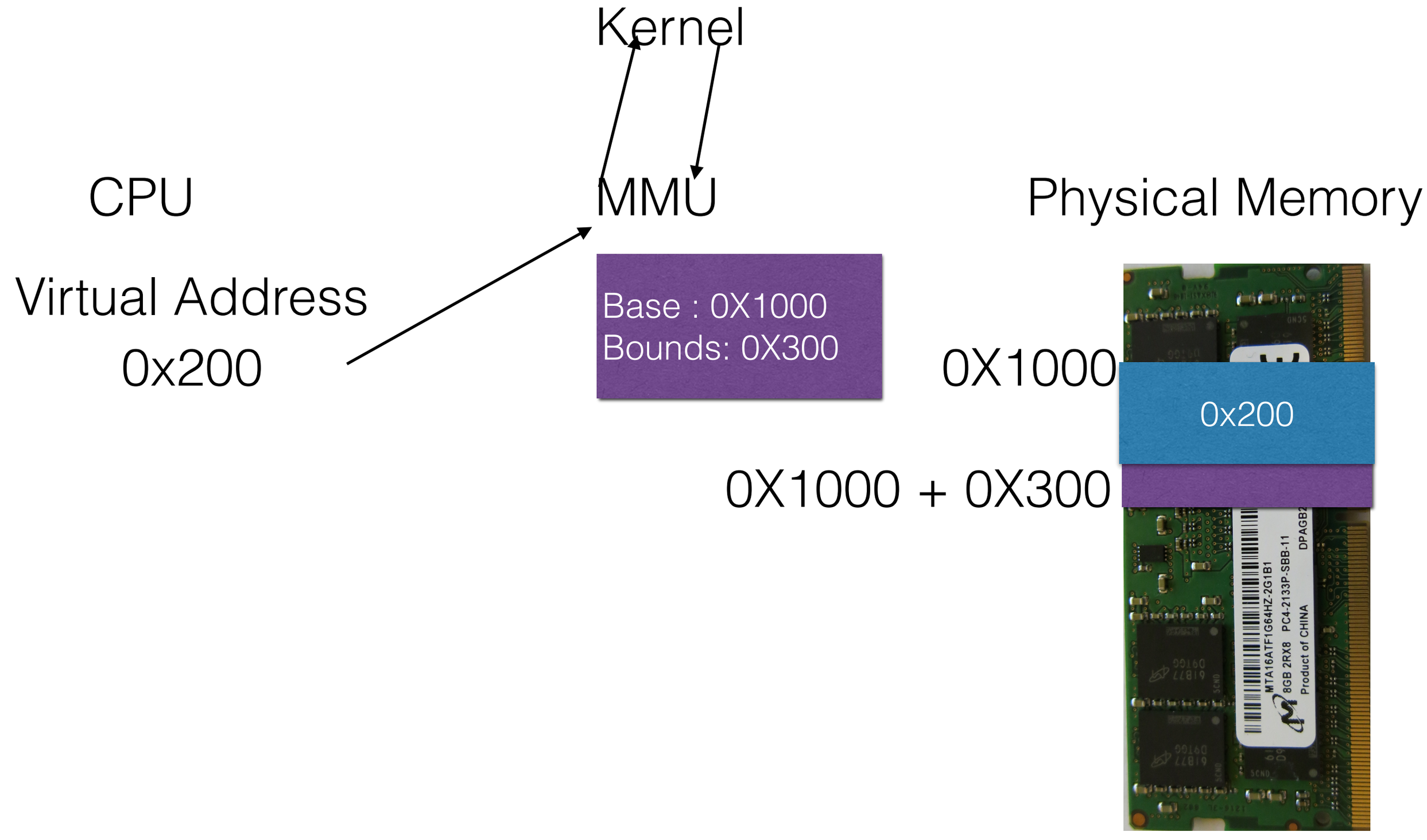
# Bounds register

1. Checks if memory reference is within bounds
2. Pop Quiz: What's the bound in our example?
    1. Ans : 16 KB
3. Incorrect virtual address : Terminate!

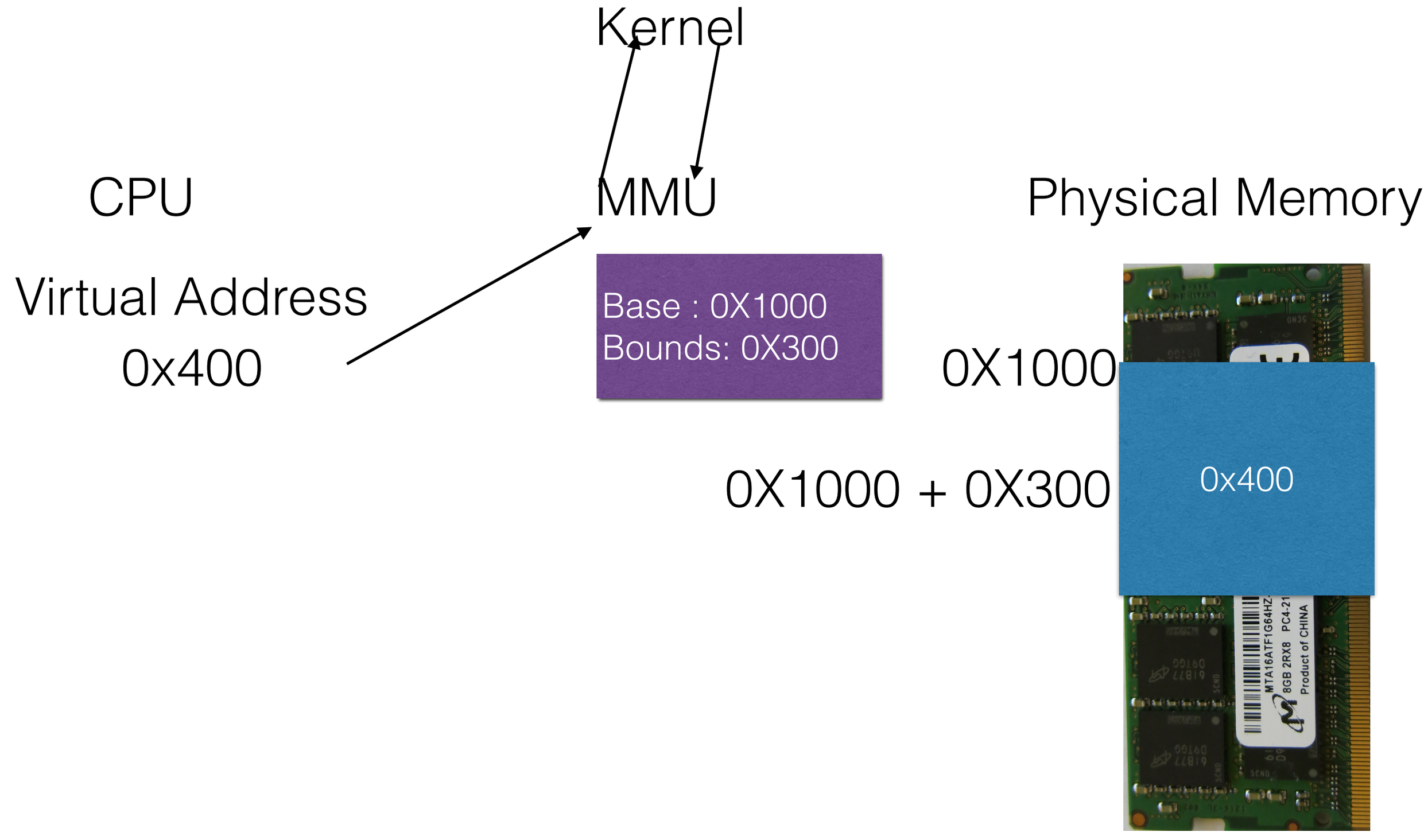
# Dynamic (Hardware-based) Relocation
# Base & Bounds



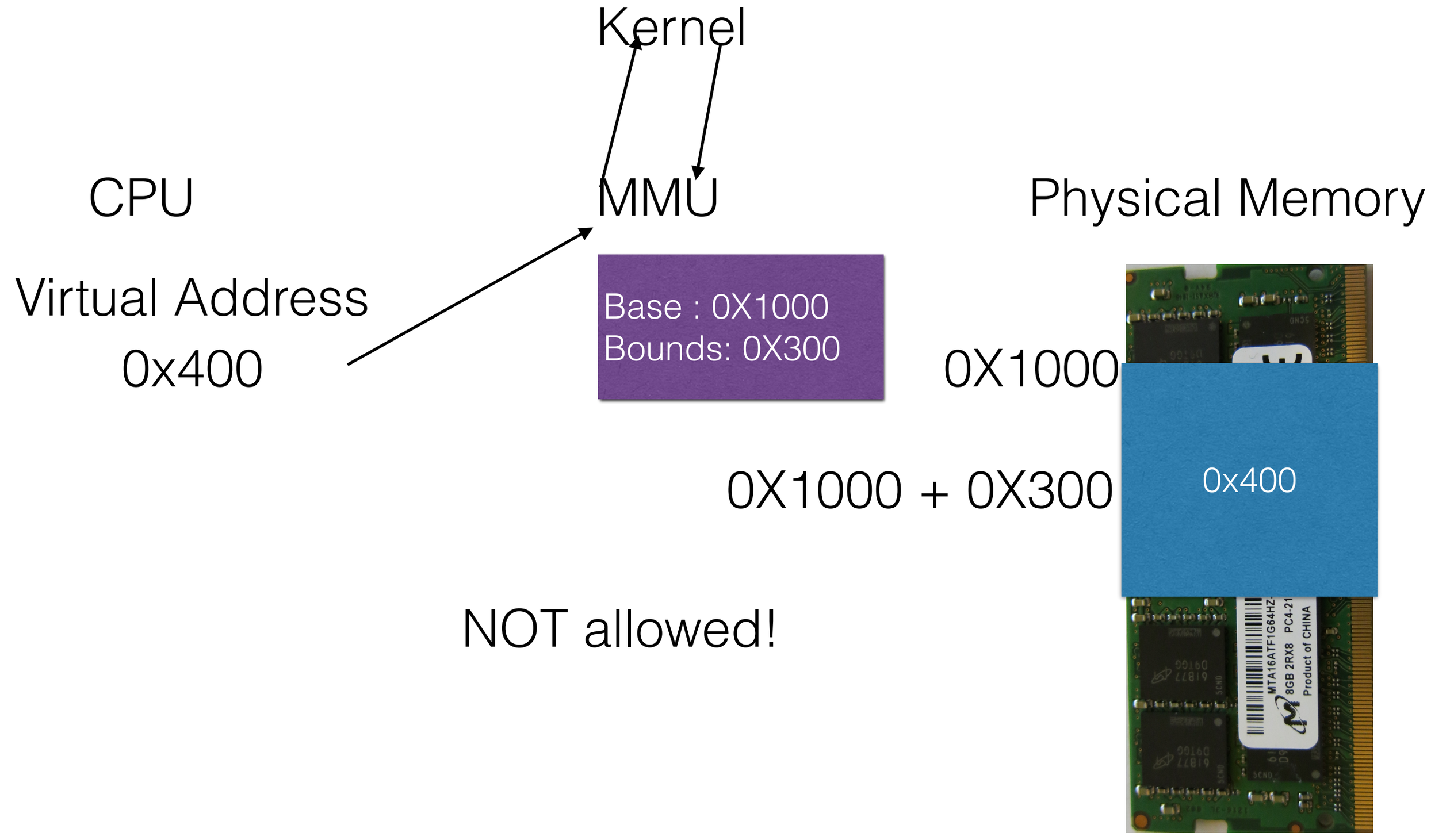
# Dynamic (Hardware-based) Relocation
# Base & Bounds



# Dynamic (Hardware-based) Relocation
# Base & Bounds



# Dynamic (Hardware-based) Relocation
# Base & Bounds



# Hardware Requirement for Dynamic Relocation

| Hardware Requirements | Notes |
|---|---|
| Privileged mode | |
| Base/bounds registers | |
| Ability to translate virtual addresses and check if within bounds | |
| Privileged instruction(s) to update base/bounds | |
| Privileged instruction(s) to register exception handlers | |
| Ability to raise exceptions | |

# Hardware Requirement for Dynamic Relocation

| **Hardware Requirements** | **Notes** |
|---|---|
| Privileged mode | *Needed to prevent user-mode processes from executing privileged operations* |
| Base/bounds registers | |
| Ability to translate virtual addresses and check if within bounds | |
| Privileged instruction(s) to update base/bounds | |
| Privileged instruction(s) to register exception handlers | |
| Ability to raise exceptions | |

# Hardware Requirement for Dynamic Relocation

| **Hardware Requirements** | **Notes** |
|---|---|
| Privileged mode | *Needed to prevent user-mode processes from executing privileged operations* |
| Base/bounds registers | *Need pair of registers per CPU to support address translation and bounds checks* |
| Ability to translate virtual addresses and check if within bounds | |
| Privileged instruction(s) to update base/bounds | |
| Privileged instruction(s) to register exception handlers | |
| Ability to raise exceptions | |

# Hardware Requirement for Dynamic Relocation

| **Hardware Requirements** | **Notes** |
|---|---|
| Privileged mode | *Needed to prevent user-mode processes from executing privileged operations* |
| Base/bounds registers | *Need pair of registers per CPU to support address translation and bounds checks* |
| Ability to translate virtual addresses and check if within bounds | *Circuitry to do translations and check limits; in this case, quite simple* |
| Privileged instruction(s) to update base/bounds | |
| Privileged instruction(s) to register exception handlers | |
| Ability to raise exceptions | |

# Hardware Requirement for Dynamic Relocation

| **Hardware Requirements** | **Notes** |
|---|---|
| Privileged mode | *Needed to prevent user-mode processes from executing privileged operations* |
| Base/bounds registers | *Need pair of registers per CPU to support address translation and bounds checks* |
| Ability to translate virtual addresses and check if within bounds | *Circuitry to do translations and check limits; in this case, quite simple* |
| Privileged instruction(s) to update base/bounds | *OS must be able to set these values before letting a user program run* |
| Privileged instruction(s) to register exception handlers | |
| Ability to raise exceptions | |

# Hardware Requirement for Dynamic Relocation

| **Hardware Requirements** | **Notes** |
|---|---|
| Privileged mode | *Needed to prevent user-mode processes from executing privileged operations* |
| Base/bounds registers | *Need pair of registers per CPU to support address translation and bounds checks* |
| Ability to translate virtual addresses and check if within bounds | *Circuitry to do translations and check limits; in this case, quite simple* |
| Privileged instruction(s) to update base/bounds | *OS must be able to set these values before letting a user program run* |
| Privileged instruction(s) to register exception handlers | *OS must be able to tell hardware what code to run if exception occurs* |
| Ability to raise exceptions | |

# Hardware Requirement for Dynamic Relocation

| **Hardware Requirements** | **Notes** |
|---|---|
| Privileged mode | *Needed to prevent user-mode processes from executing privileged operations* |
| Base/bounds registers | *Need pair of registers per CPU to support address translation and bounds checks* |
| Ability to translate virtual addresses and check if within bounds | *Circuitry to do translations and check limits; in this case, quite simple* |
| Privileged instruction(s) to update base/bounds | *OS must be able to set these values before letting a user program run* |
| Privileged instruction(s) to register exception handlers | *OS must be able to tell hardware what code to run if exception occurs* |
| Ability to raise exceptions | *When processes try to access privileged instructions or out-of-bounds memory* |

# OS Responsibilities for Dynamic Relocation

# OS Responsibilities for Dynamic Relocation

1. Memory Management :

# OS Responsibilities for Dynamic Relocation

1. Memory Management :
    1. Allocate memory for new processes

# OS Responsibilities for Dynamic Relocation

1. Memory Management :
    1. Allocate memory for new processes
    2. Reclaim memory from terminated process

# OS Responsibilities for Dynamic Relocation

1. Memory Management :
    1. Allocate memory for new processes
    2. Reclaim memory from terminated process
    3. Manage memory via free list

# OS Responsibilities for Dynamic Relocation

1. Memory Management :
    1. Allocate memory for new processes
    2. Reclaim memory from terminated process
    3. Manage memory via free list
2. Base/Bound management :

# OS Responsibilities for Dynamic Relocation

1. Memory Management :
    1. Allocate memory for new processes
    2. Reclaim memory from terminated process
    3. Manage memory via free list
2. Base/Bound management :
    1. Set base/bound upon context switch

# OS Responsibilities for Dynamic Relocation

1. Memory Management :
    1. Allocate memory for new processes
    2. Reclaim memory from terminated process
    3. Manage memory via free list
2. Base/Bound management :
    1. Set base/bound upon context switch
3. Exception handling :

# OS Responsibilities for Dynamic Relocation

1. Memory Management :
    1. Allocate memory for new processes
    2. Reclaim memory from terminated process
    3. Manage memory via free list
2. Base/Bound management :
    1. Set base/bound upon context switch
3. Exception handling :
    1. Terminate offending process

# Base & Bounds : Pros

# Base & Bounds : Pros

1. Simple : Hardware only needs to know base & bounds

# Base & Bounds : Pros

1. Simple : Hardware only needs to know base & bounds
2. Relatively fast :

# Base & Bounds : Pros

1. Simple : Hardware only needs to know base & bounds
2. Relatively fast :
    1. Protection : 1 comparison (bound)

# Base & Bounds : Pros

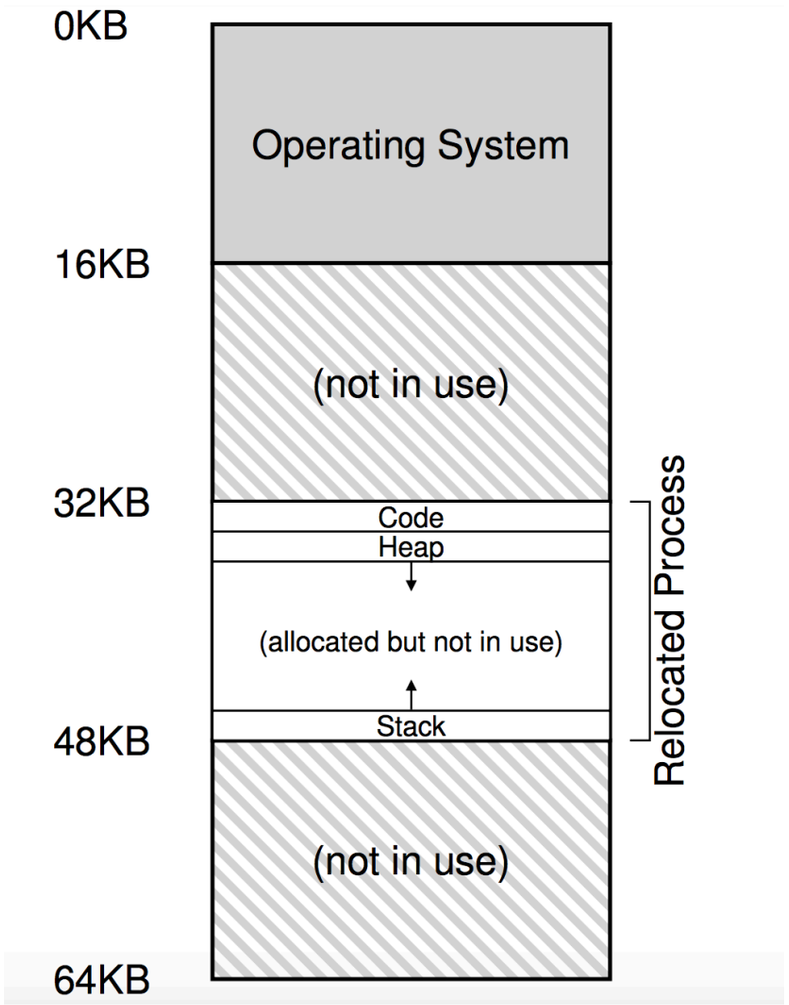
1. Simple : Hardware only needs to know base & bounds
2. Relatively fast :
    1. Protection : 1 comparison (bound)
    2. Translation : 1 addition

# Base & Bounds : Cons

# Base & Bounds : Cons

Base

Base + Bound

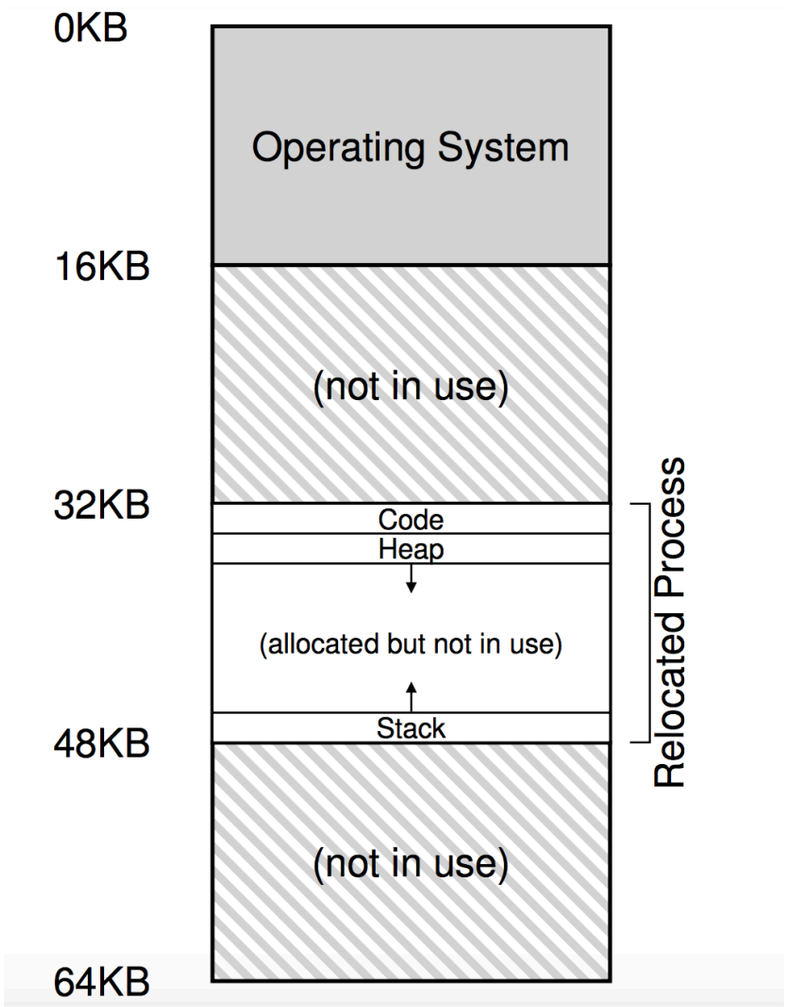


# Base & Bounds : Cons

1. Contiguous block of memory needed in physical memory

Base

Base + Bound

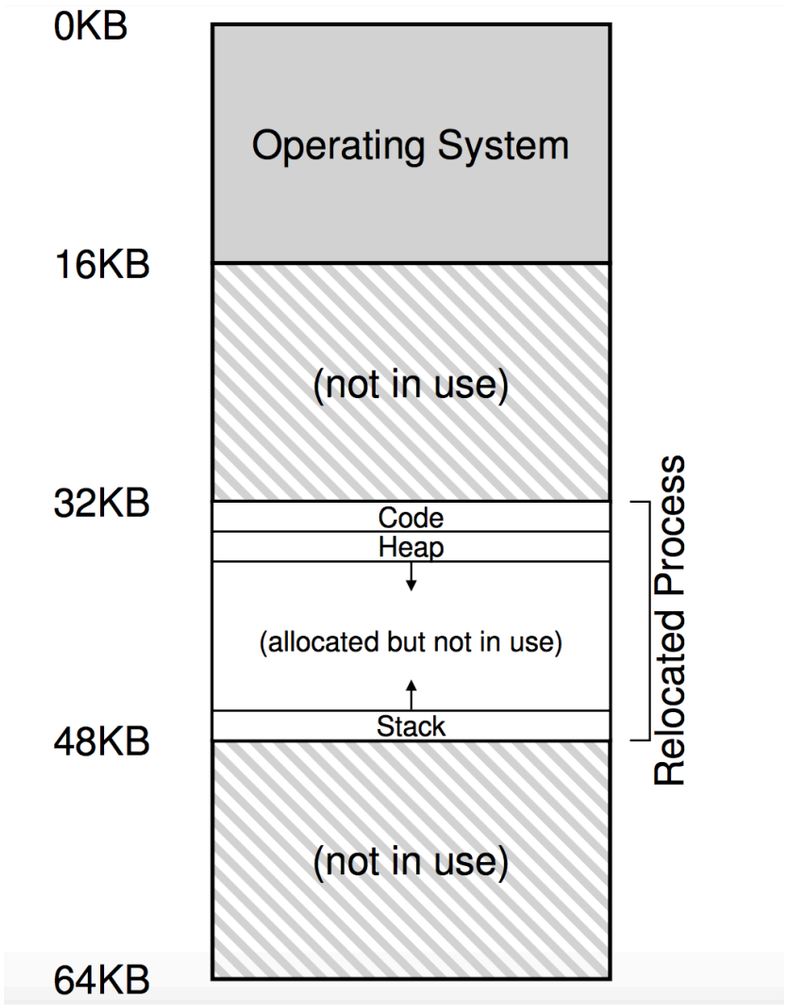


# Base & Bounds : Cons

1. Contiguous block of memory needed in physical memory
    1. Internal fragmentation

Base

Base + Bound

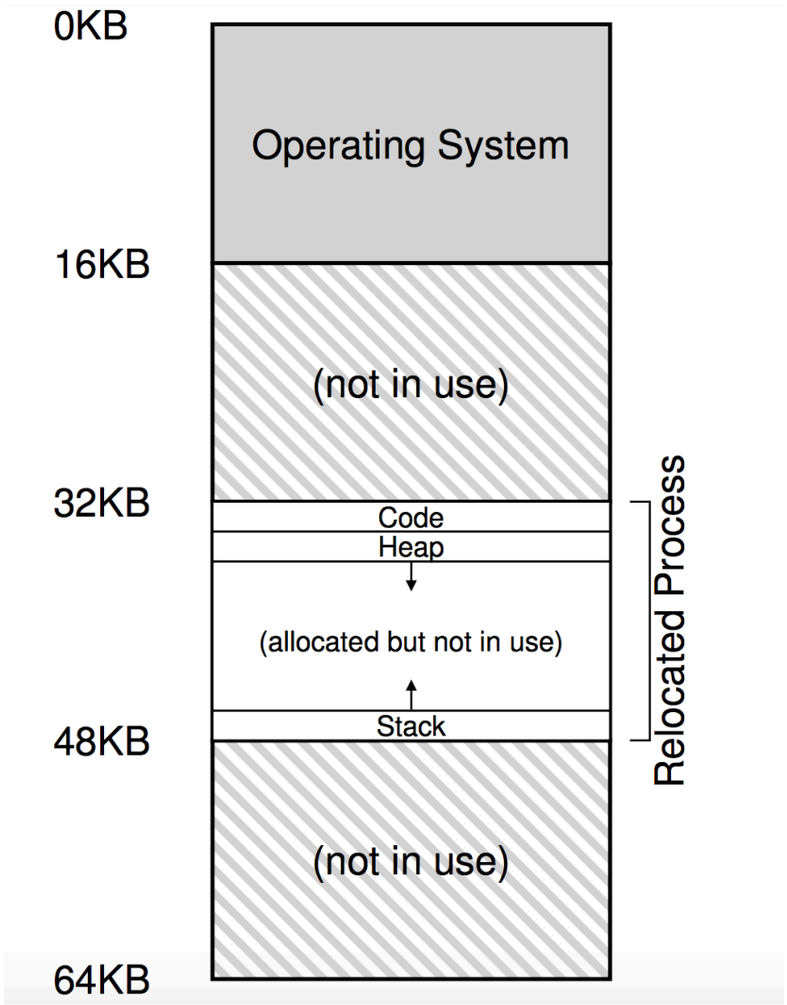


# Base & Bounds : Cons

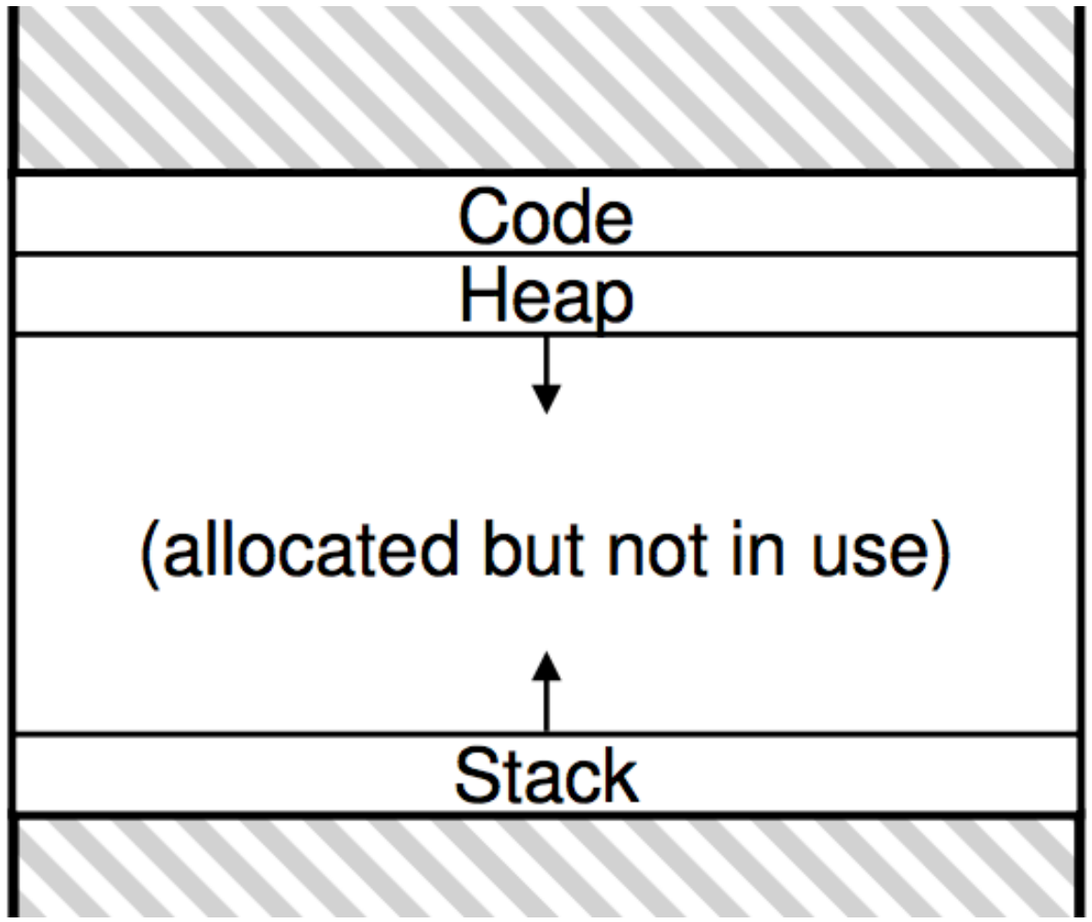
1. Contiguous block of memory needed in physical memory
   1. Internal fragmentation
   2. External fragmentation

Base

Base + Bound



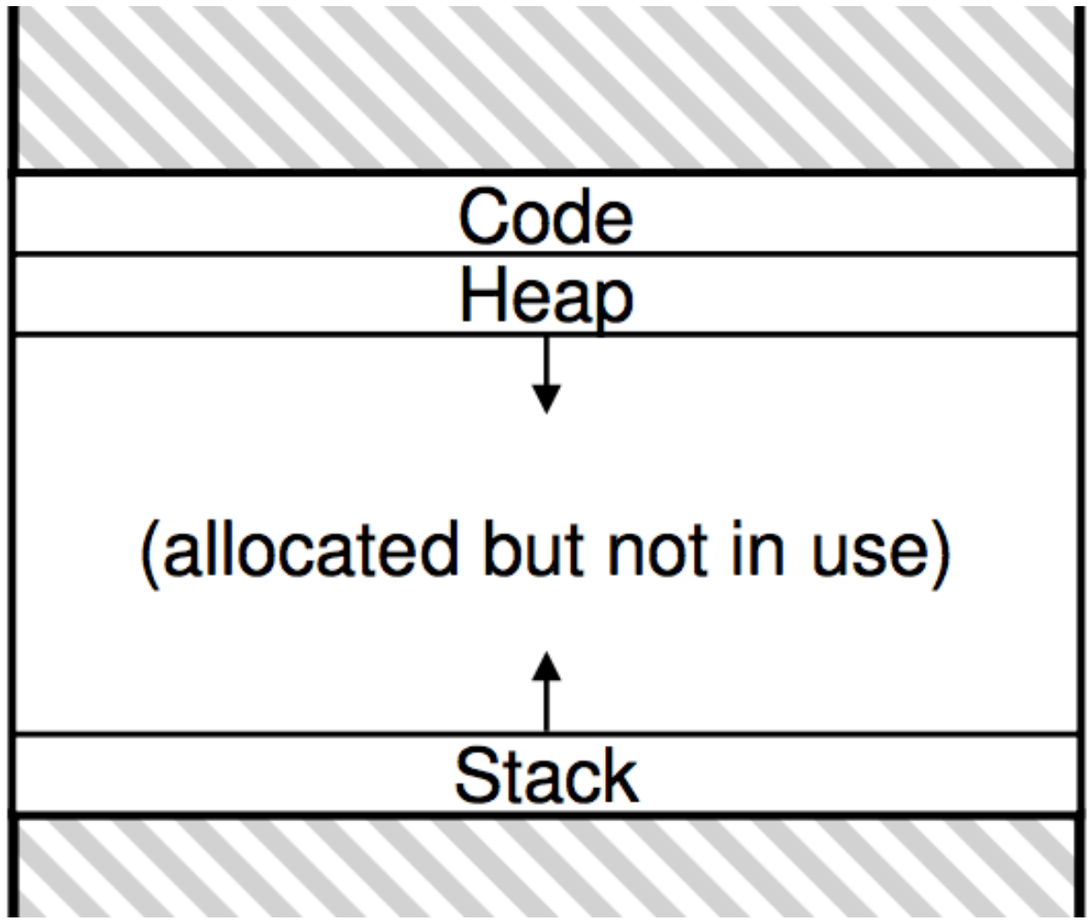
# Segmentation



# Segmentation

Base for Code

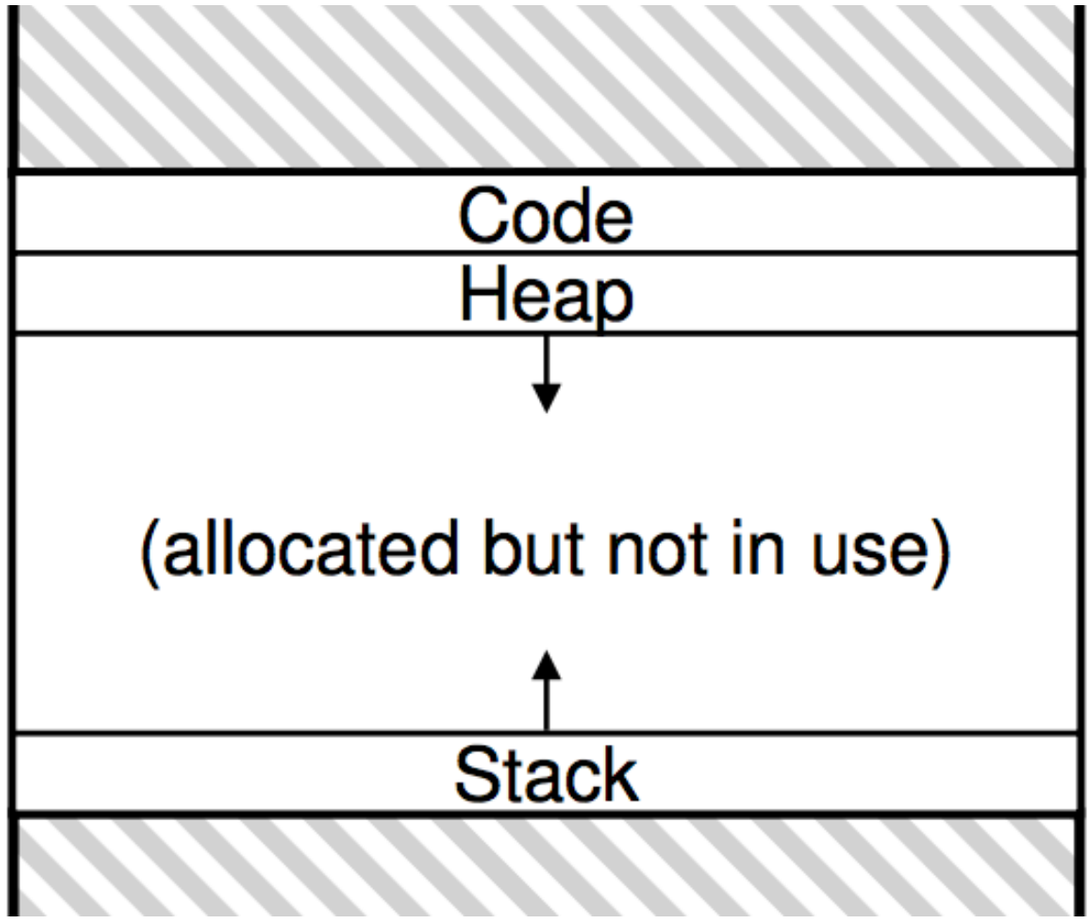
Base + Bound for Code



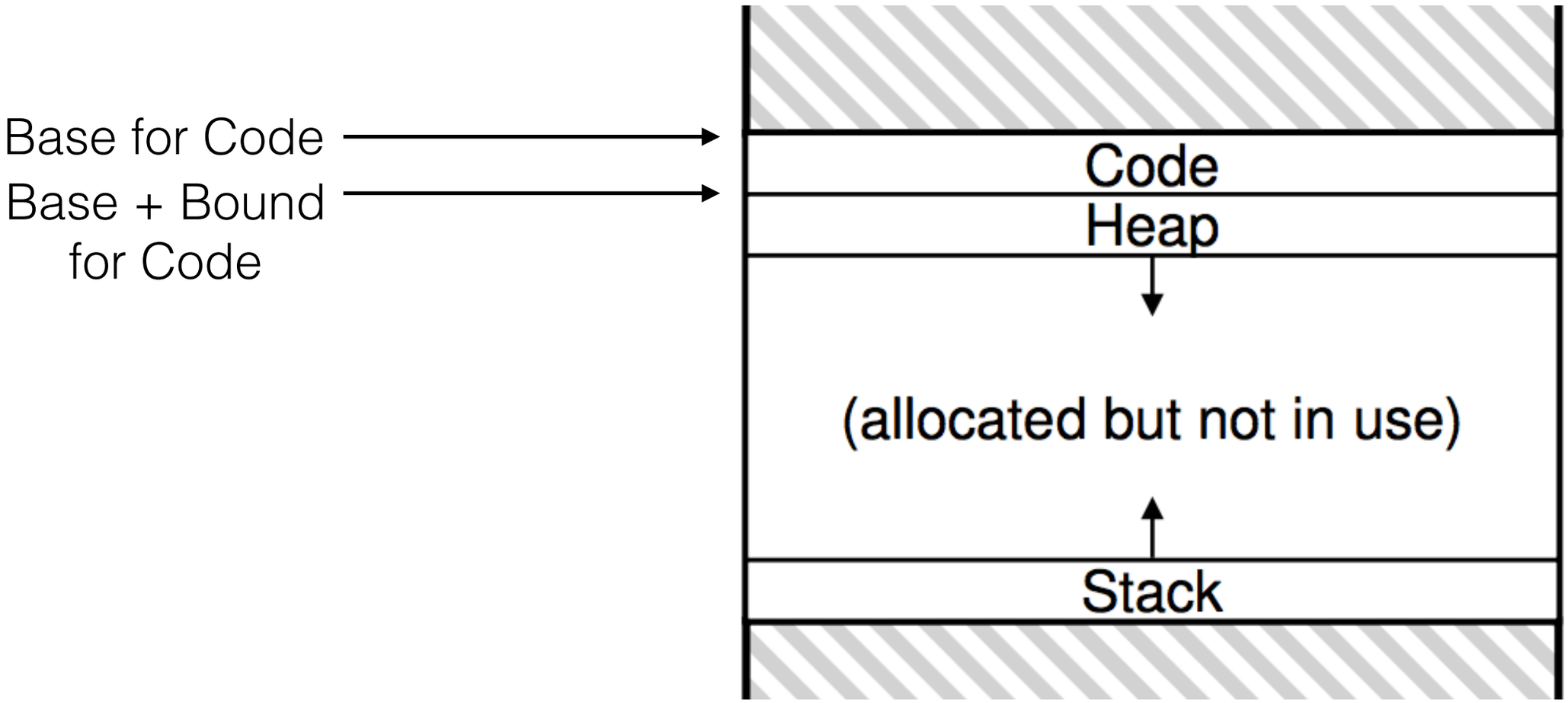
# Segmentation

Base for Code

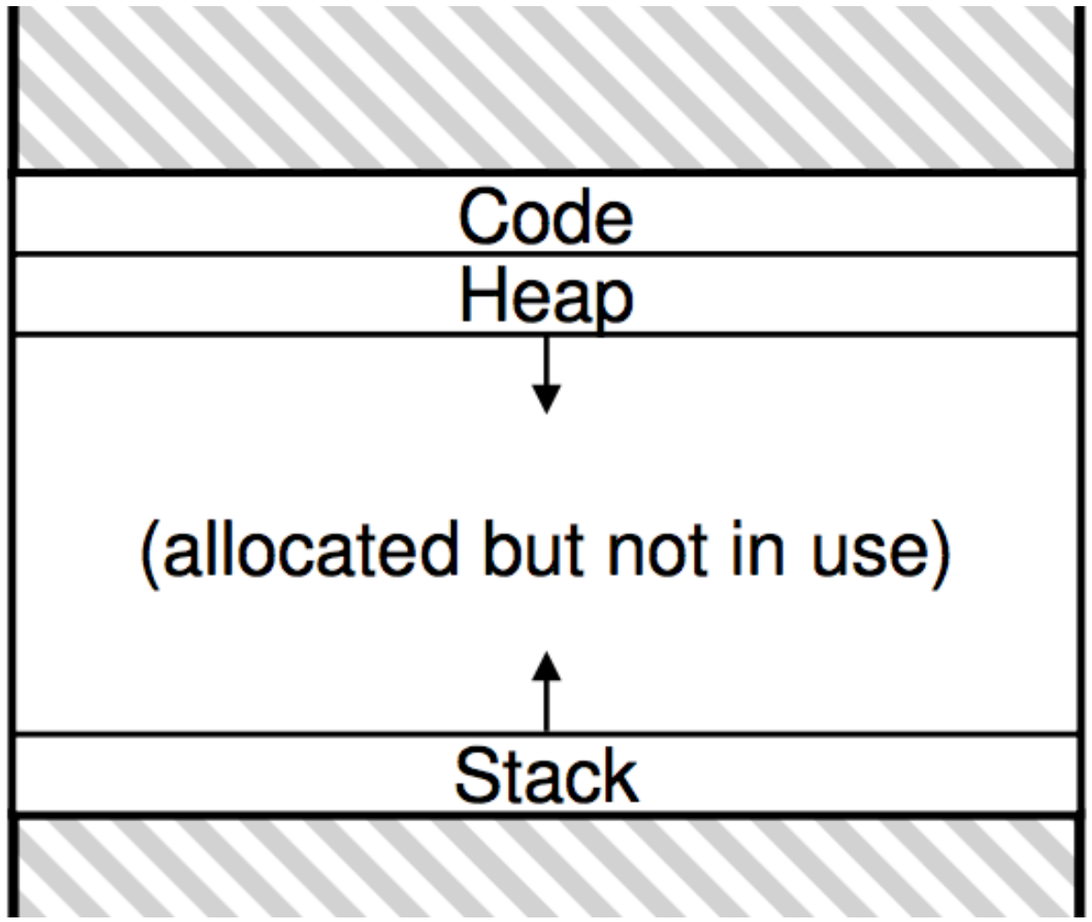
Base + Bound
for Code



# Segmentation

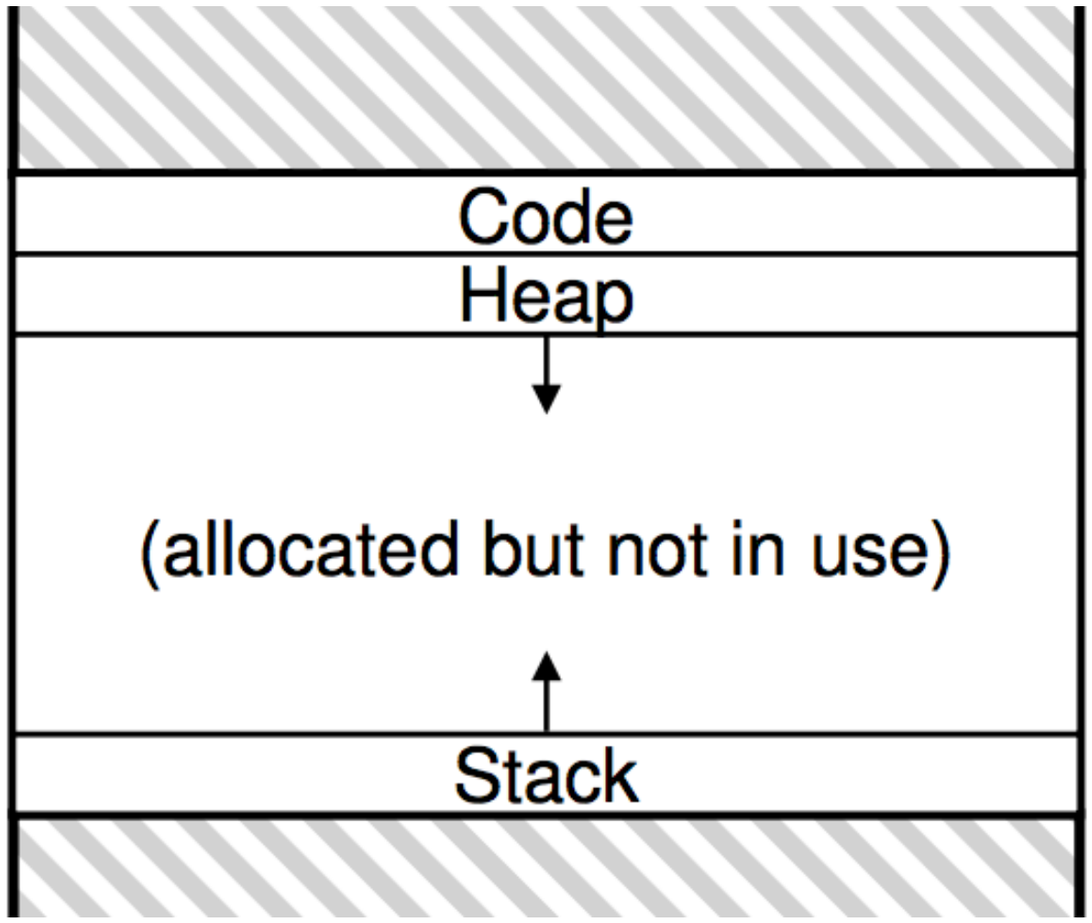


# Segmentation



# Segmentation

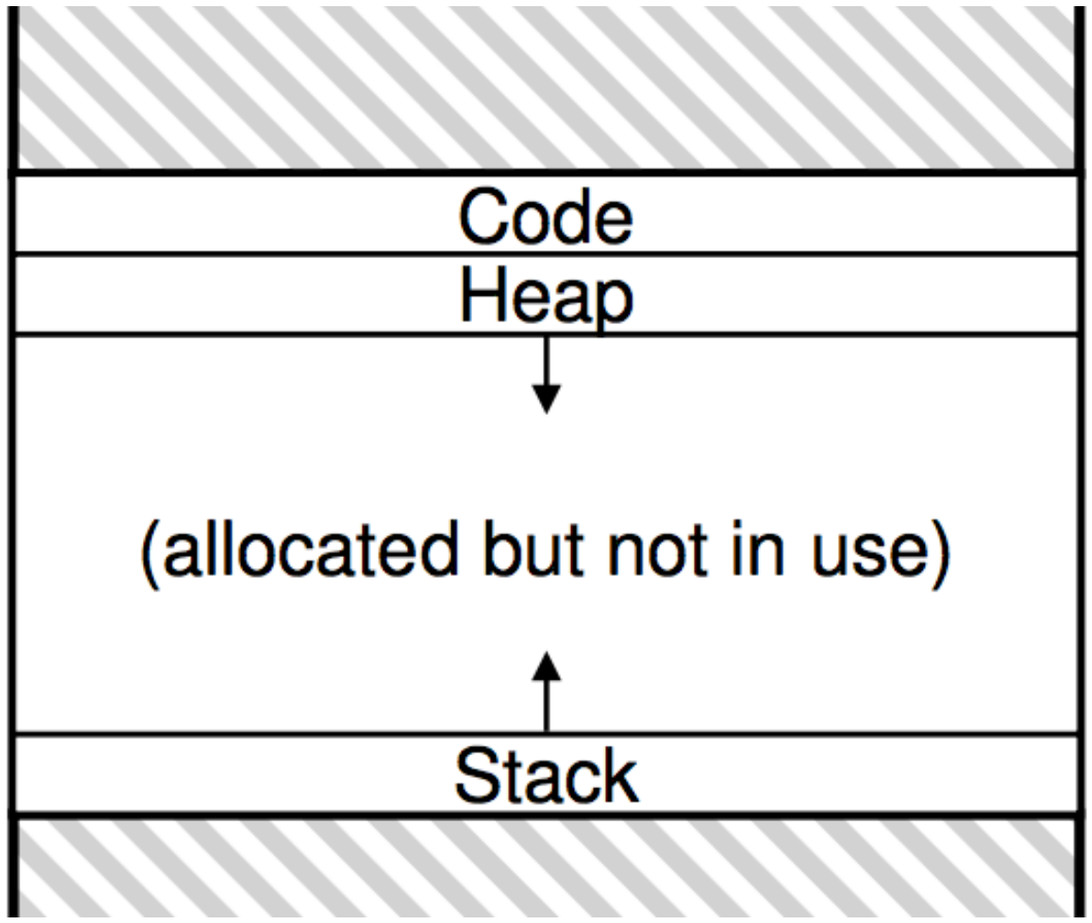
Base for Heap
Base + Bound
for Heap



# Segmentation

Base for Heap

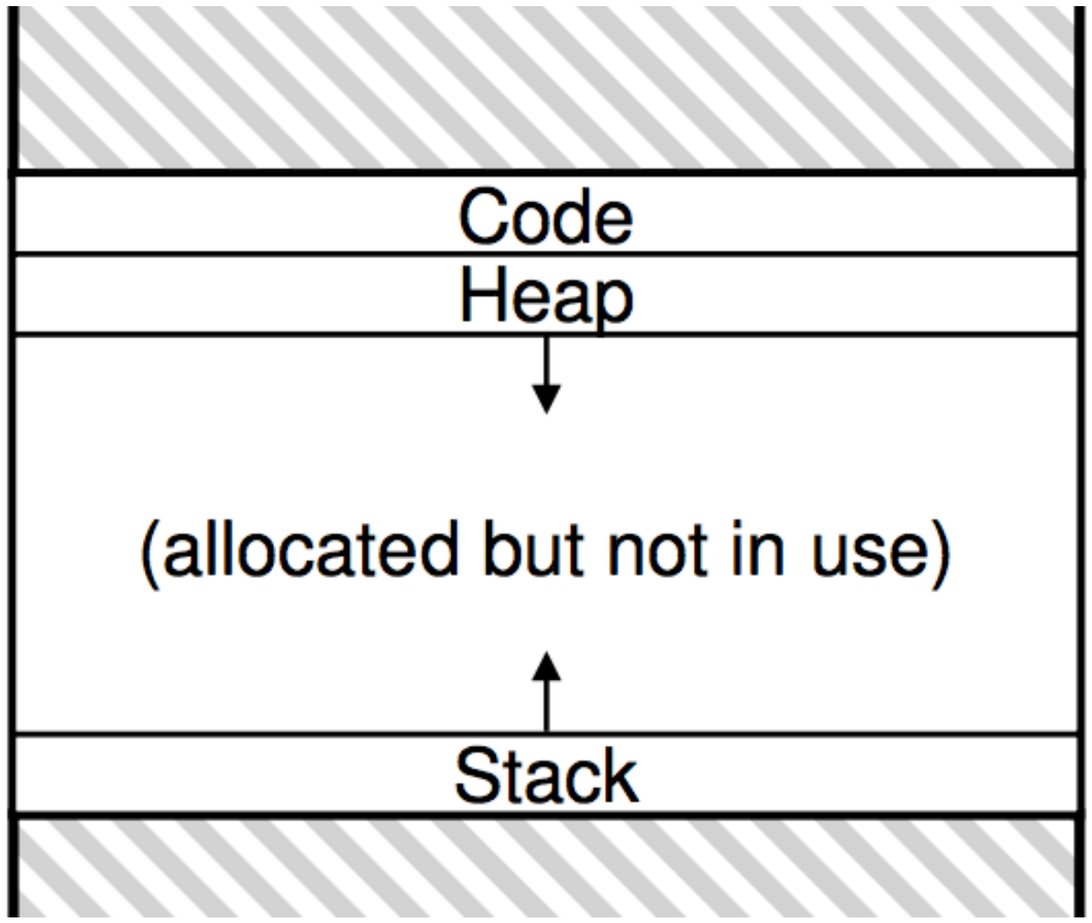
Base + Bound for Heap



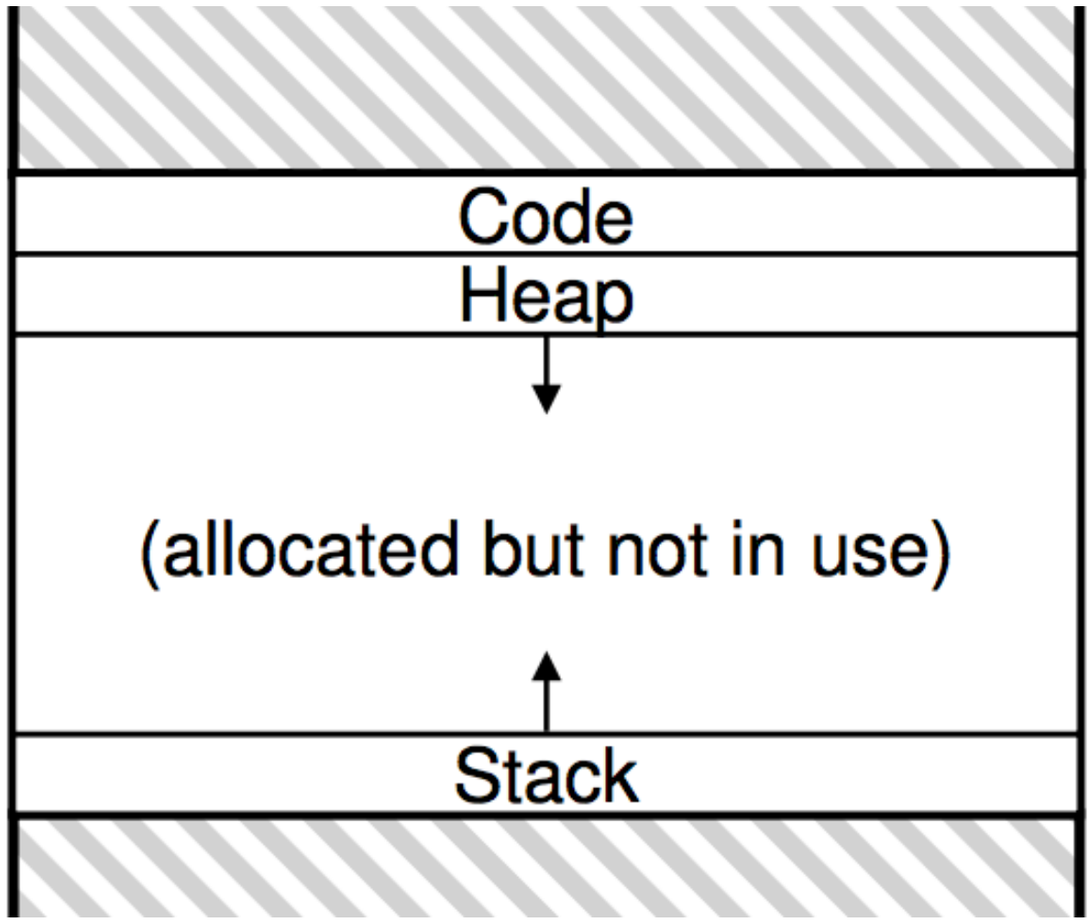
# Segmentation

Base for Heap

Base + Bound for Heap



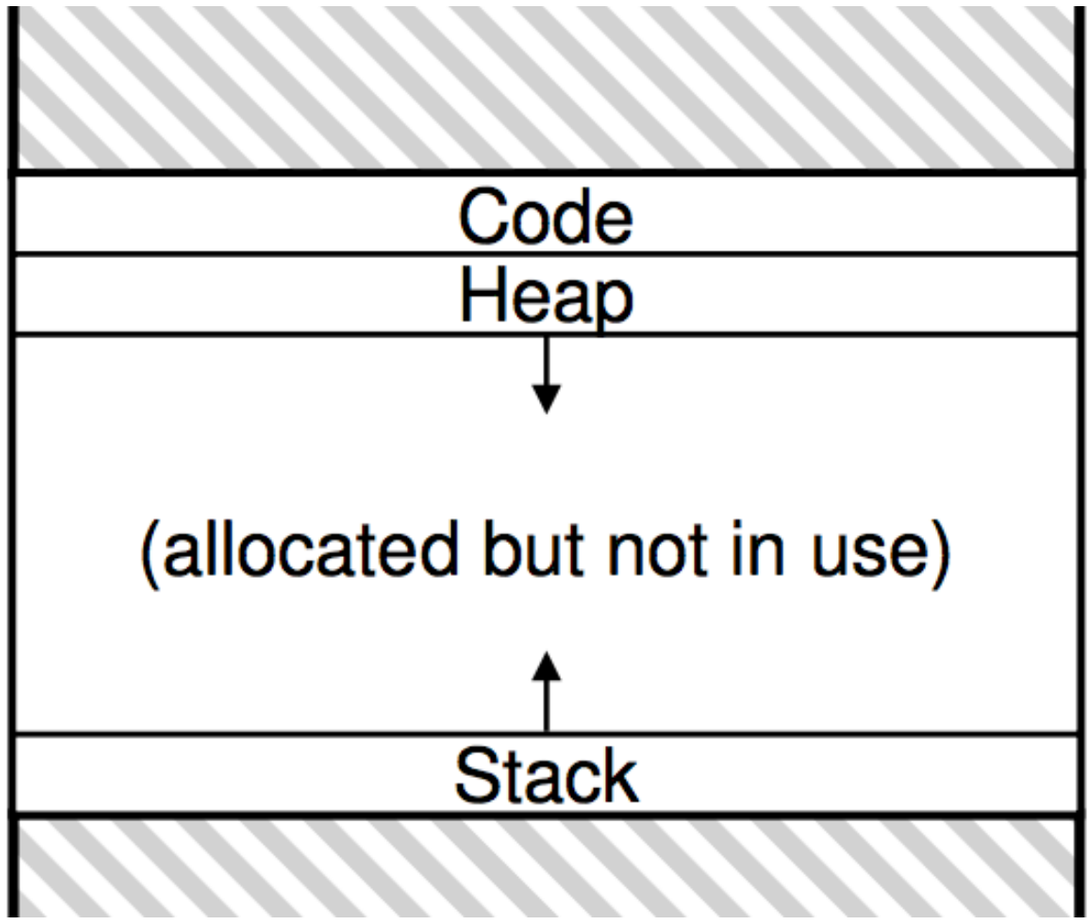
# Segmentation



# Segmentation

Base + Bound for Stack

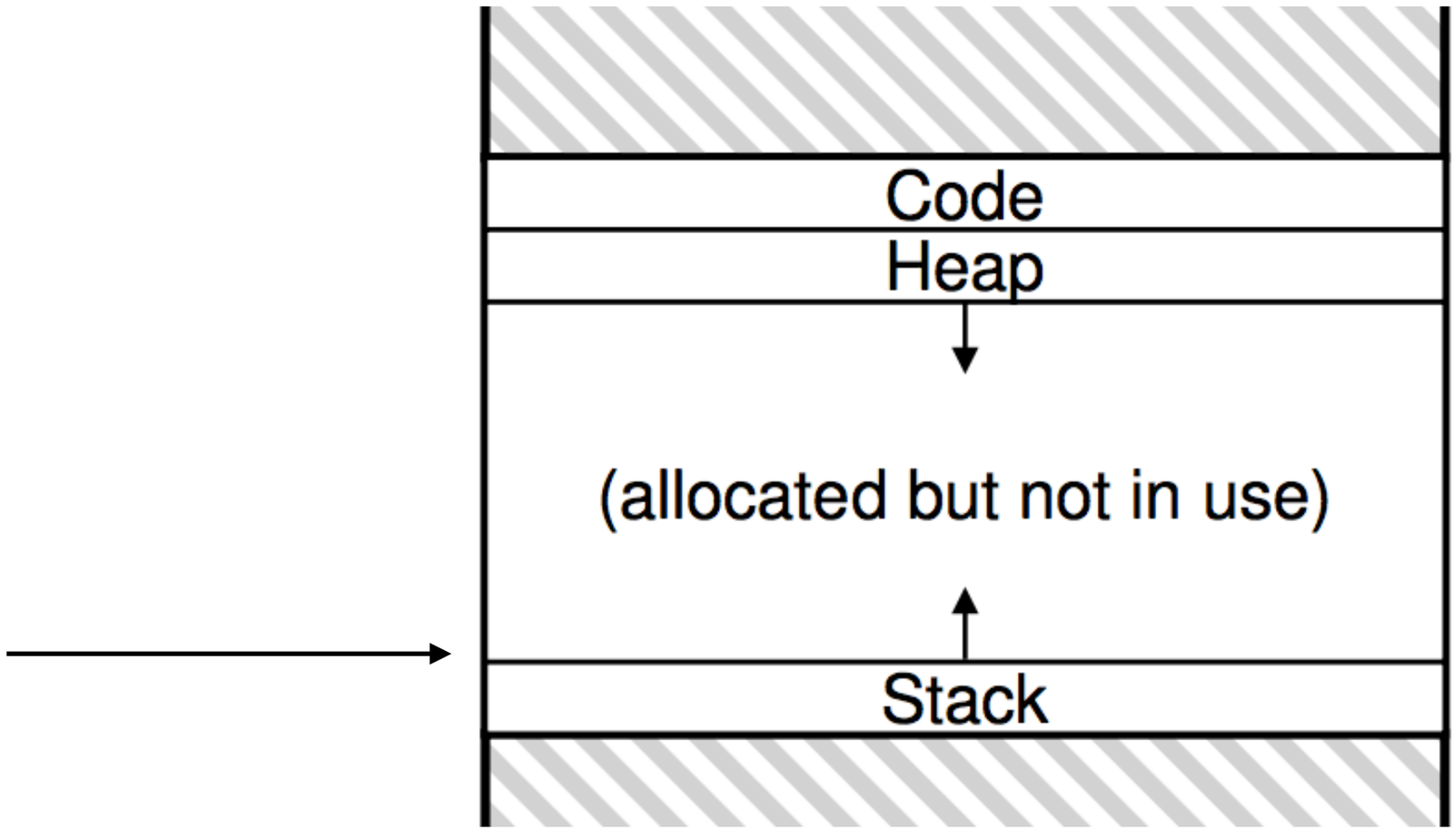
Base for Stack



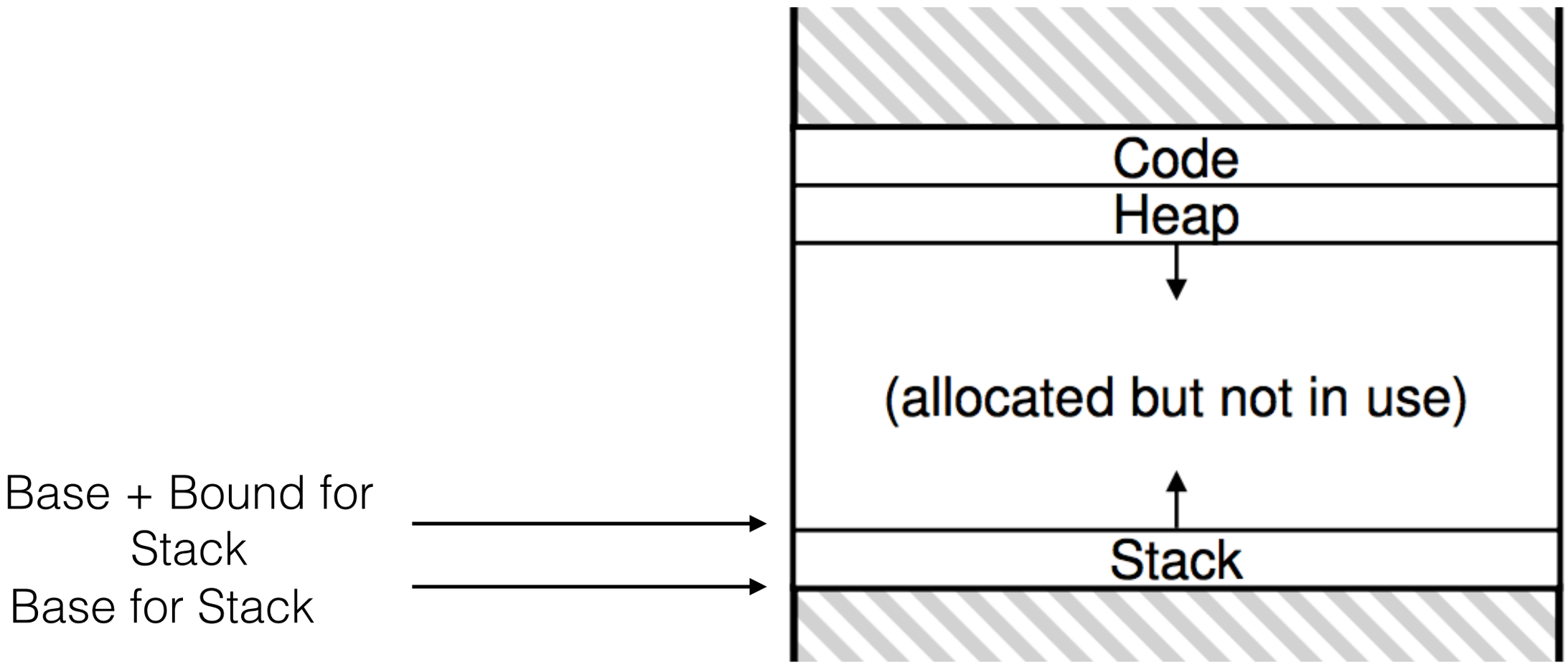
# Segmentation

Base + Bound for Stack

Base for Stack



# Segmentation



# Segmentation

# Segmentation

1. Multiple bases and bounds per process. Each of them called a segment.

# Segmentation

1. Multiple bases and bounds per process. Each of them called a segment.
    1. Each segment can have own size

# Segmentation

1. Multiple bases and bounds per process. Each of them called a segment.
    1. Each segment can have own size
    2. Each segment can have own permissions:

# Segmentation

1. Multiple bases and bounds per process. Each of them called a segment.
    1. Each segment can have own size
    2. Each segment can have own permissions:
        1. Code would be?

# Segmentation

1. Multiple bases and bounds per process. Each of them called a segment.
    1. Each segment can have own size
    2. Each segment can have own permissions:
        1. Code would be?
3. Each segment can have direction of growth!

# Segmentation : Mini Guide (TLDR version)

# Segmentation : Mini Guide (TLDR version)

1. Each segment has:

# Segmentation : Mini Guide (TLDR version)

1. Each segment has:
    1. Start virtual address (VA)

# Segmentation : Mini Guide (TLDR version)

1. Each segment has:
    1. Start virtual address (VA)
    2. Base physical address

# Segmentation : Mini Guide (TLDR version)

1. Each segment has:
    1. Start virtual address (VA)
    2. Base physical address
    3. Bound

# Segmentation : Mini Guide (TLDR version)

1. Each segment has:
    1. Start virtual address (VA)
    2. Base physical address
    3. Bound
2. **Check:** Virtual Address is OK if it inside some segment, or for some segment:

# Segmentation : Mini Guide (TLDR version)

1. Each segment has:
    1. Start virtual address (VA)
    2. Base physical address
    3. Bound
2. **Check:** Virtual Address is OK if it inside some segment, or for some segment:
    1. Segment Start < V.A. < Segment Start + Segment Bound.

# Segmentation : Mini Guide (TLDR version)

1. Each segment has:
    1. Start virtual address (VA)
    2. Base physical address
    3. Bound
2. **Check:** Virtual Address is OK if it inside some segment, or for some segment:
    1. Segment Start < V.A. < Segment Start + Segment Bound.
3. **Translate**: For the segment that contains this virtual address:

# Segmentation : Mini Guide (TLDR version)

1. Each segment has:
    1. Start virtual address (VA)
    2. Base physical address
    3. Bound
2. **Check:** Virtual Address is OK if it inside some segment, or for some segment:
    1. Segment Start < V.A. < Segment Start + Segment Bound.
3. **Translate**: For the segment that contains this virtual address:
    1. Physical Address = (V.A. - Segment Start) + Segment Base

# Segmentation Example

Kernel

CPU

MMU

Physical Memory

Virtual Address

0x100

# Segmentation Example

Kernel

CPU

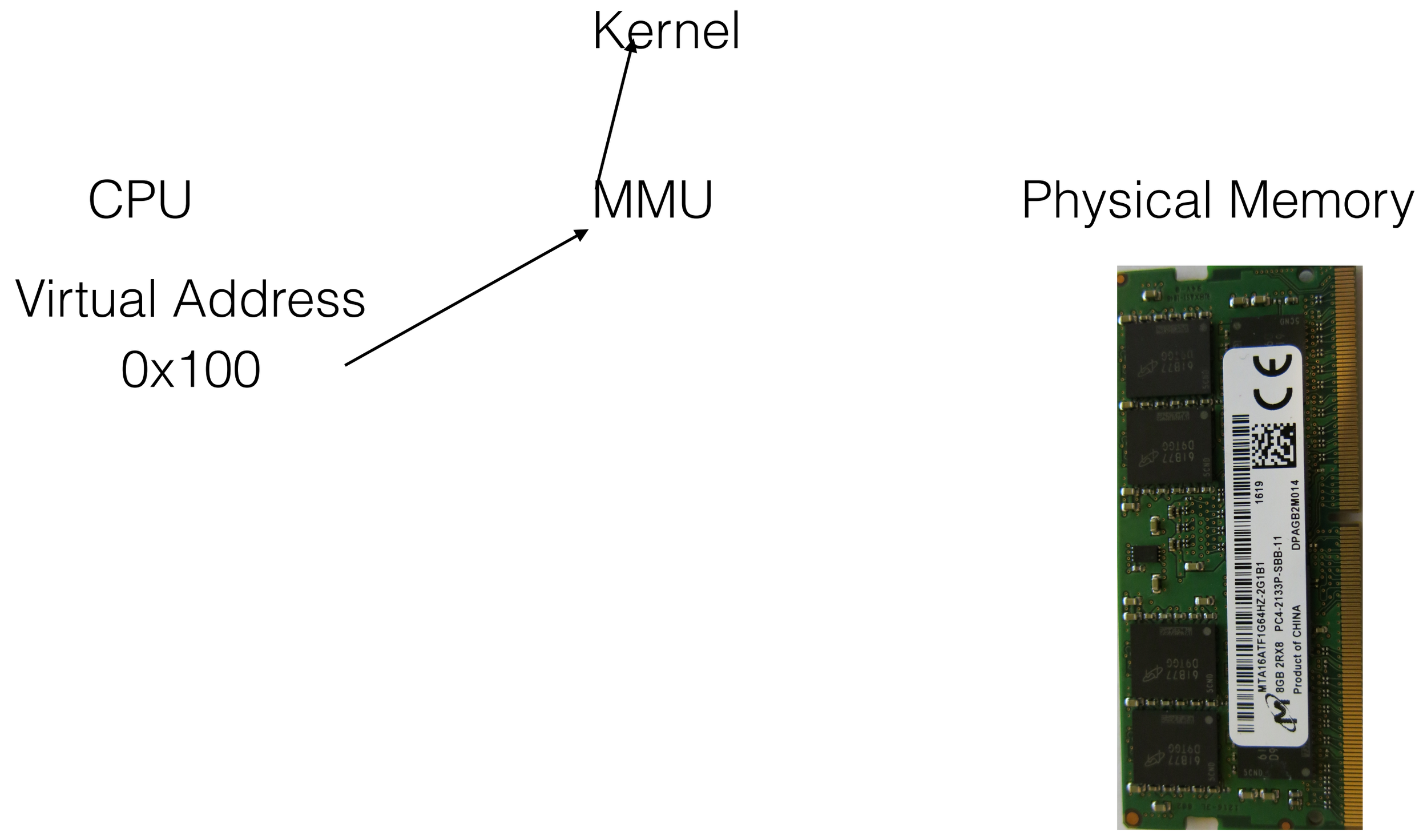
MMU

Physical Memory

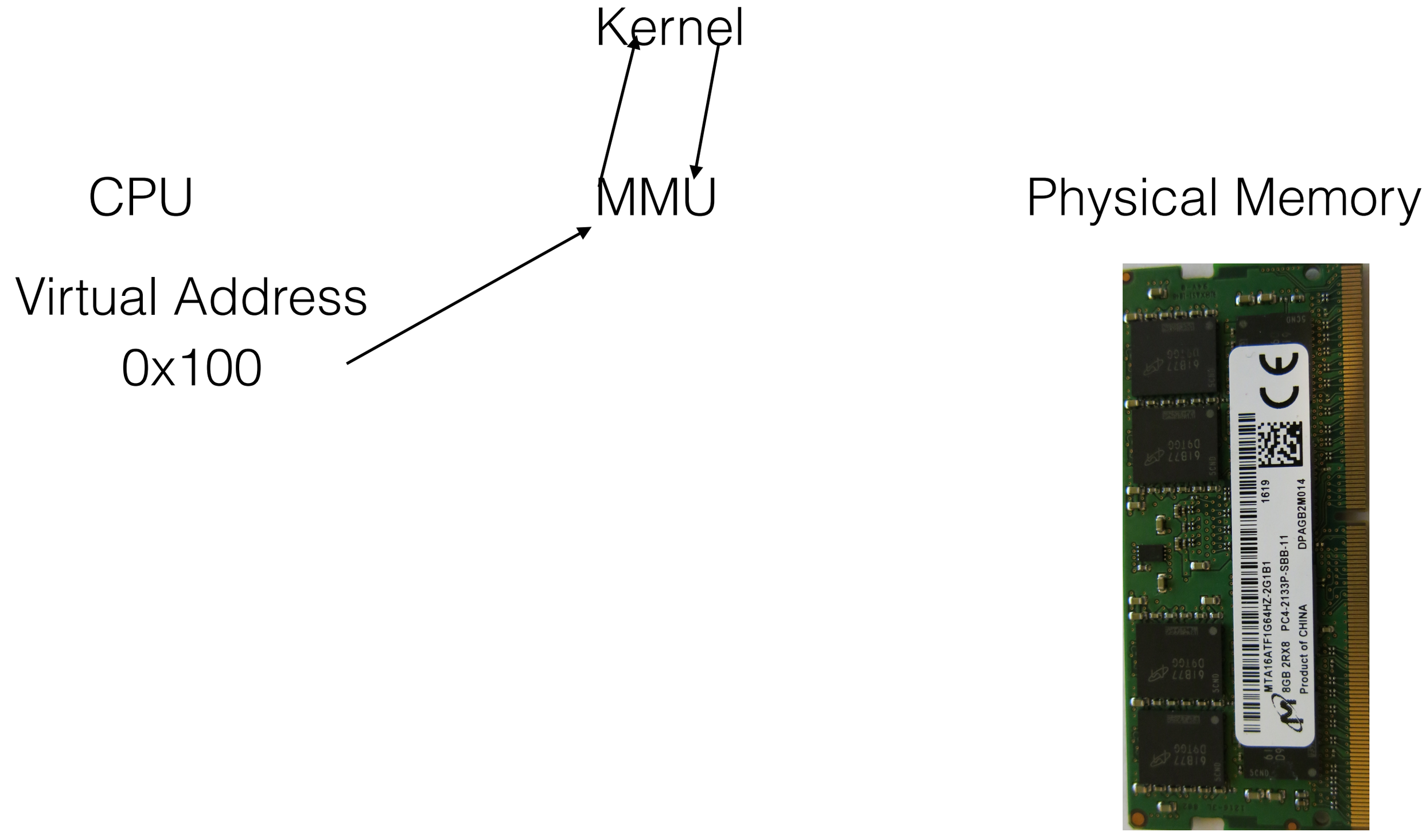
Virtual Address

0x100

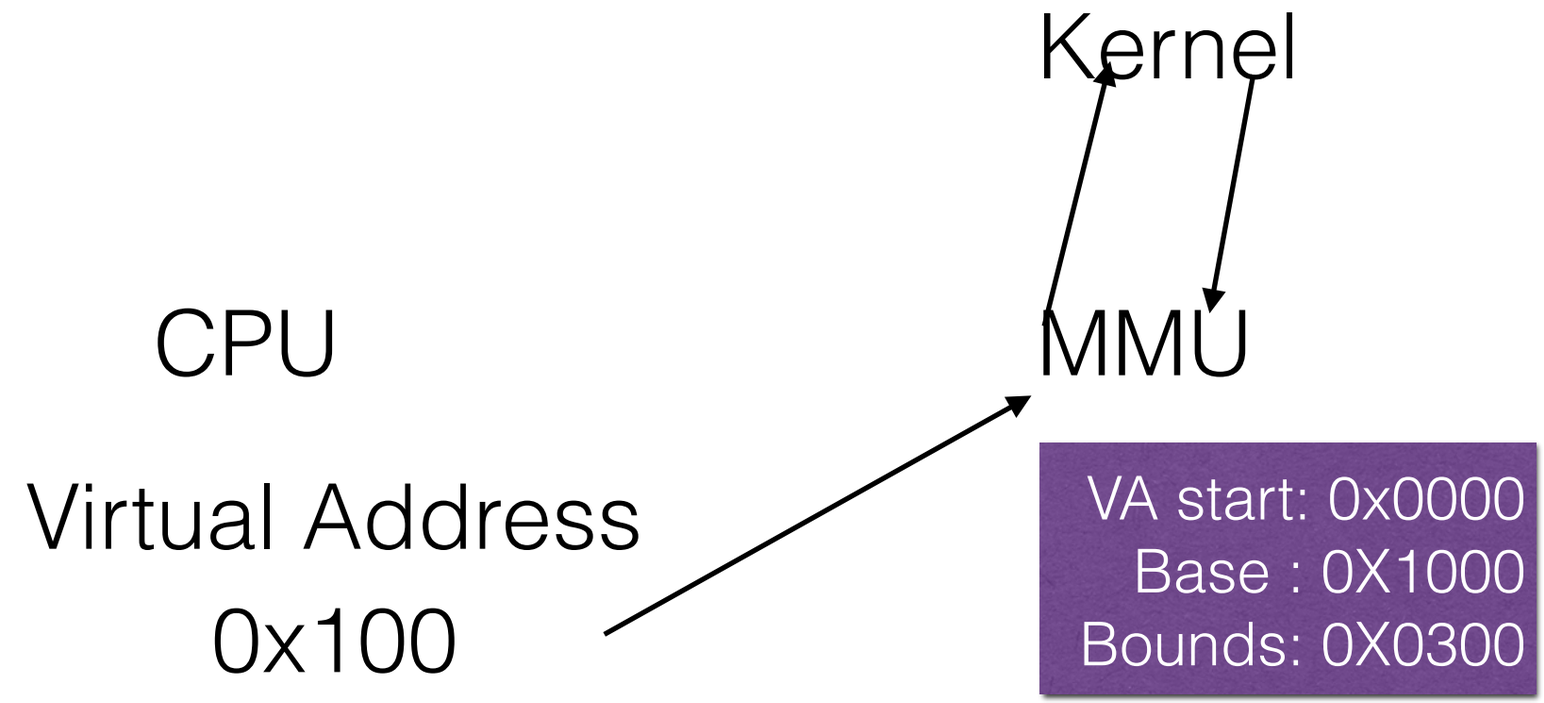
# Segmentation Example



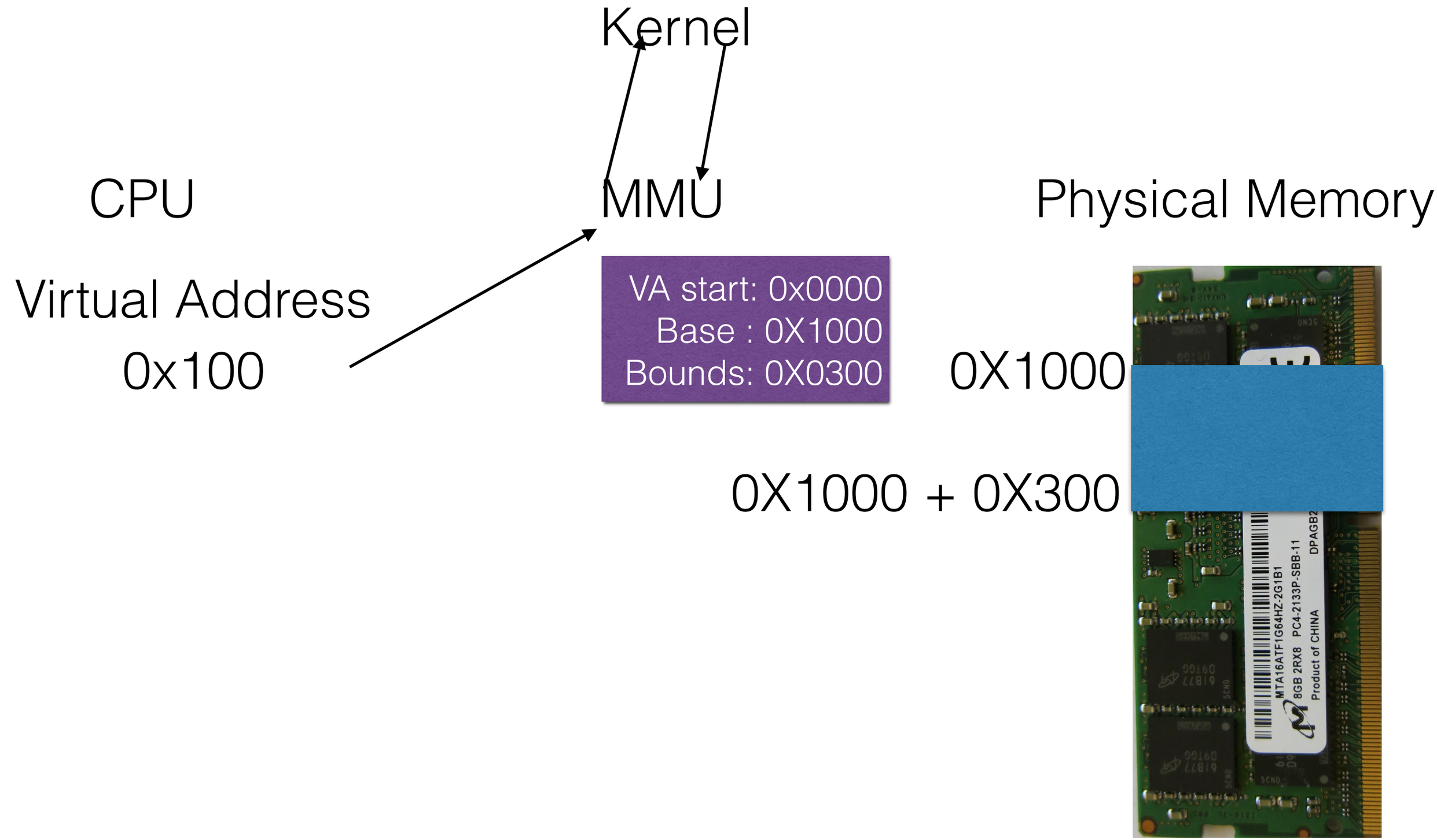
# Segmentation Example



# Segmentation Example




# Segmentation Example



# Segmentation Example

Kernel

CPU

MMU

Physical Memory

Virtual Address

0x100

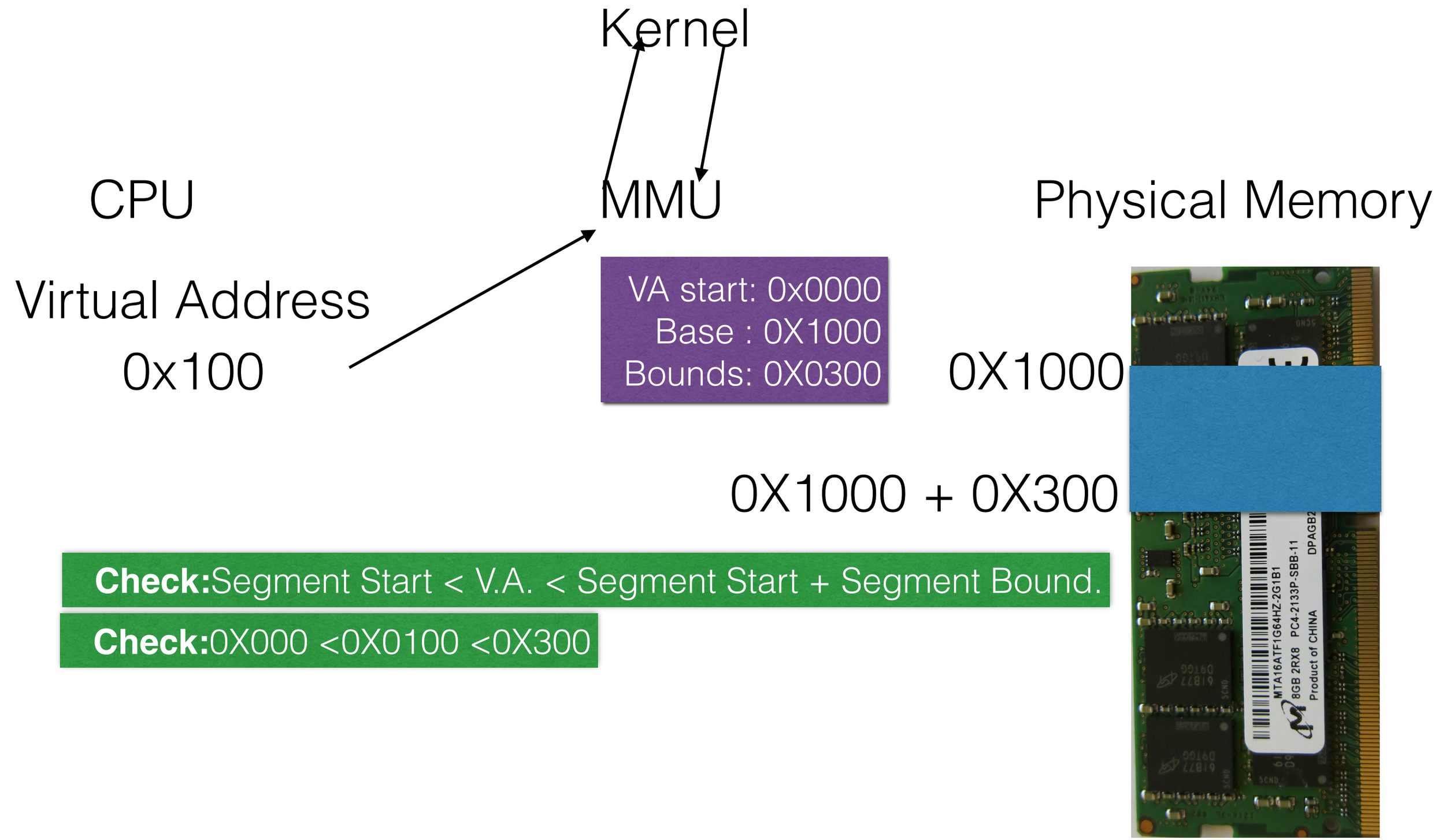
VA start: 0x0000
Base : 0X1000
Bounds: 0X0300

0X1000

0X1000 + 0X300


**Check:**Segment Start < V.A. < Segment Start + Segment Bound.

# Segmentation Example



# Segmentation Example

Kernel

CPU

MMU

Physical Memory

Virtual Address

0x100

VA start: 0x0000
Base : 0X1000
Bounds: 0X0300

0X1000

0X1000 + 0X300


**Check:**Segment Start < V.A. < Segment Start + Segment Bound.

**Check:**0X000 <0X0100 <0X300

# Segmentation Example

Kernel

CPU

MMU

Physical Memory

Virtual Address

0x100

VA start: 0x0000
Base : 0X1000
Bounds: 0X0300

0X1000


0X1000 + 0X300

**Check:**Segment Start < V.A. < Segment Start + Segment Bound.

**Check:**0X000 <0X0100 <0X300

Physical Address = (V.A. - Segment Start) + Segment Base

# Segmentation Example

Kernel

CPU

MMU

Physical Memory

Virtual Address

0x100

VA start: 0x0000
Base : 0X1000
Bounds: 0X0300

0X1000

0X1000 + 0X300

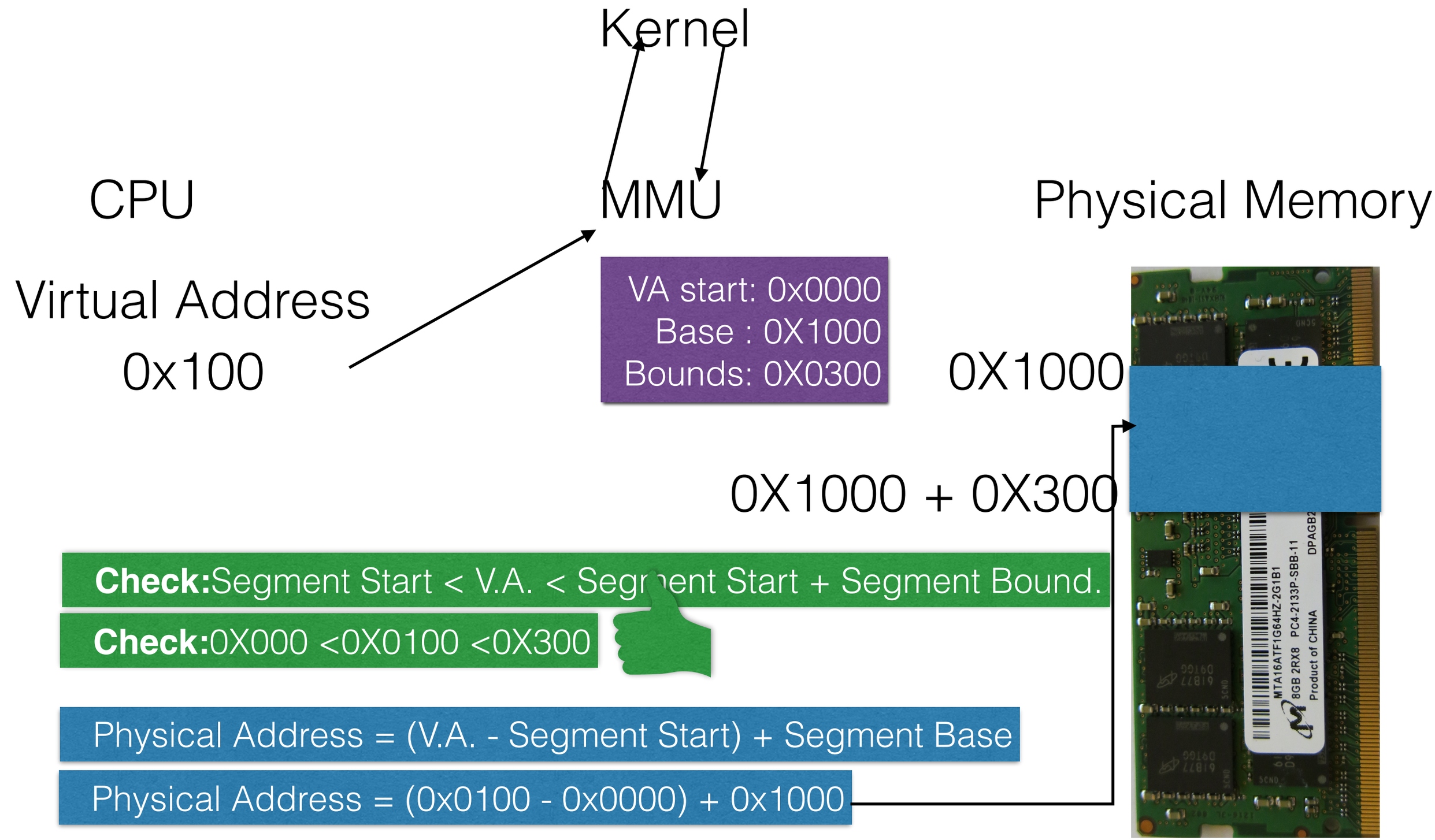

**Check:**Segment Start < V.A. < Segment Start + Segment Bound.

**Check:**0X000 <0X0100 <0X300

Physical Address = (V.A. - Segment Start) + Segment Base

Physical Address = (0x0100 - 0x0000) + 0x1000

# Segmentation Example



# Segmentation Example

Kernel

CPU

MMU

Physical Memory

Virtual Address

0x2400

# Segmentation Example

Kernel

CPU

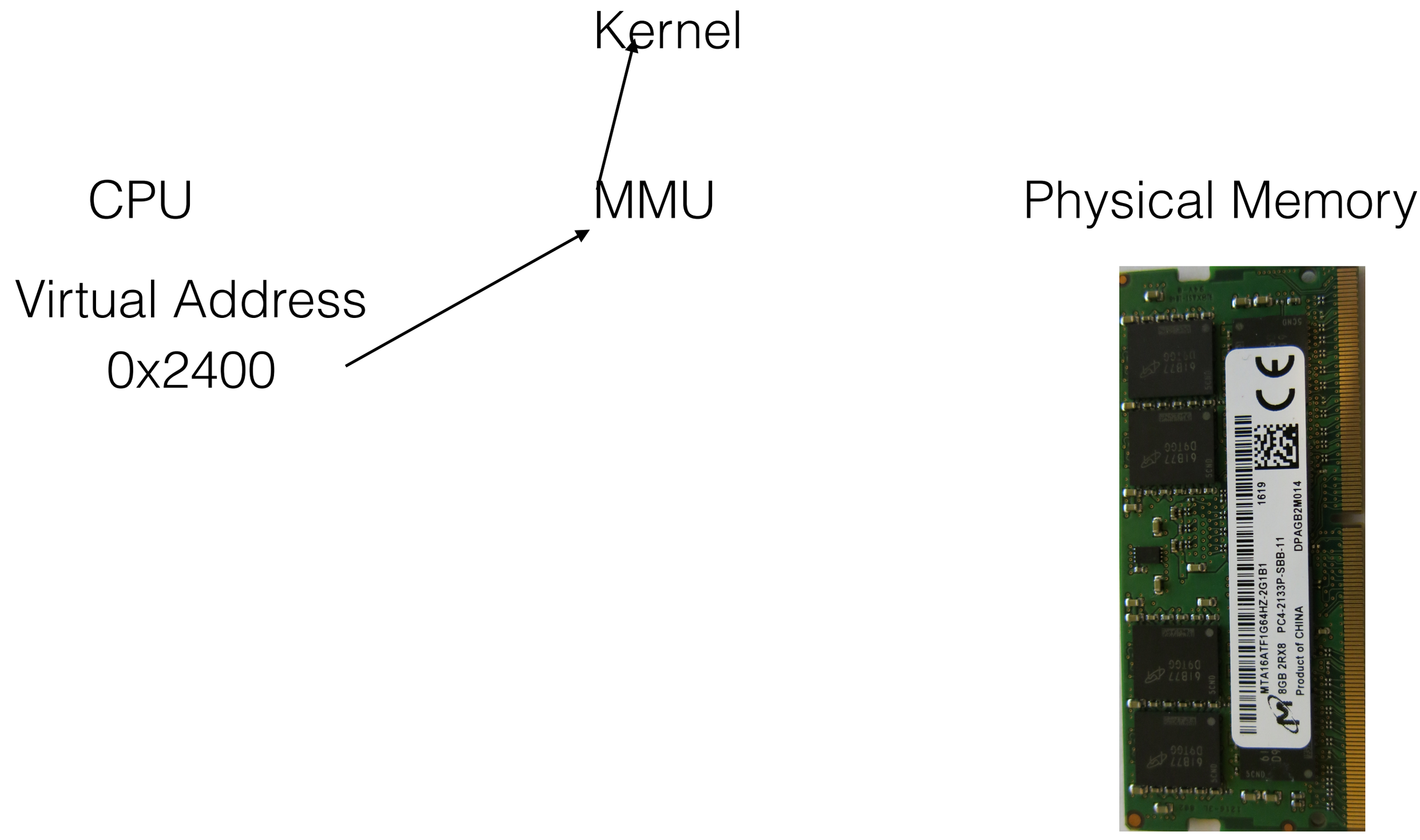
MMU

Physical Memory

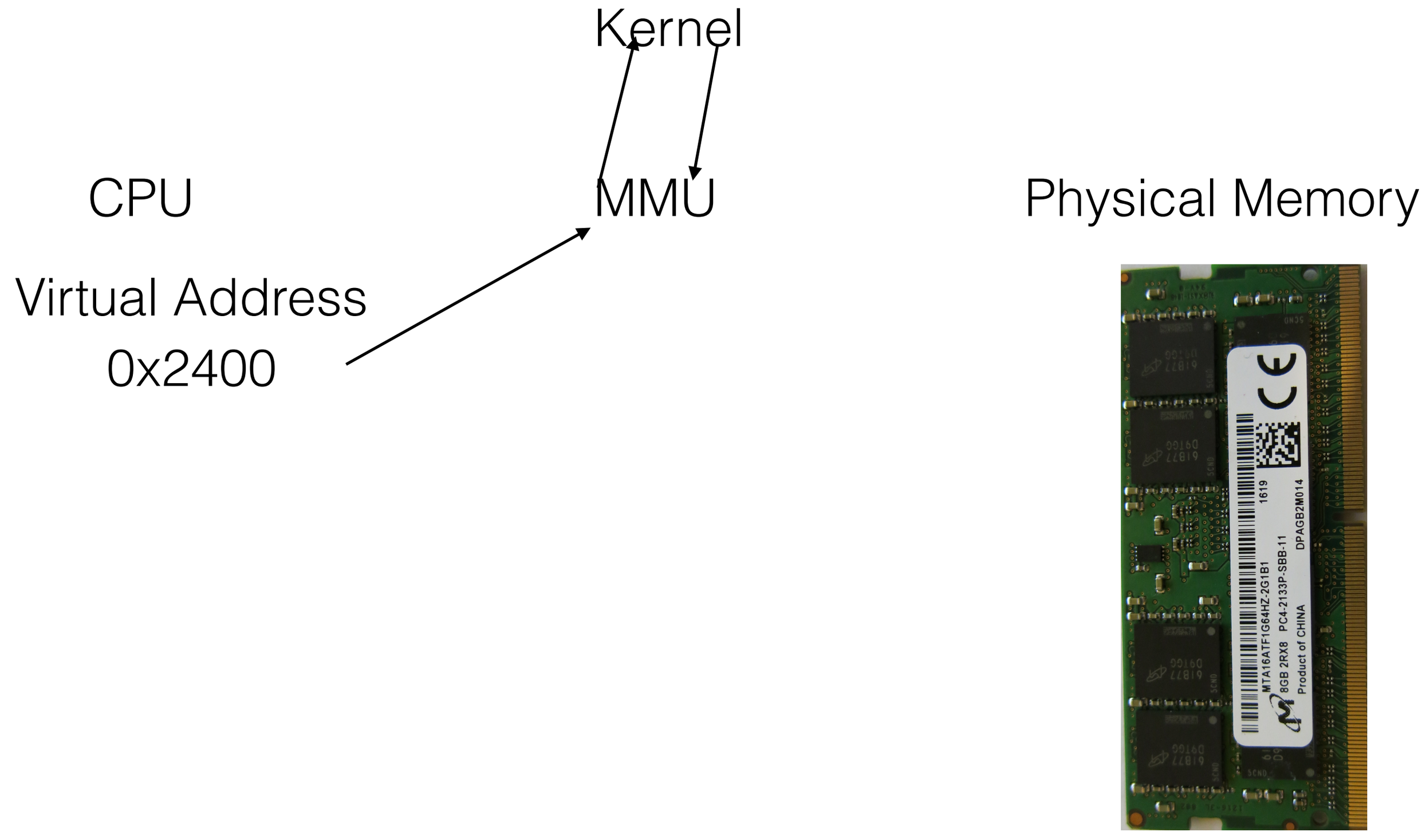
Virtual Address

0x2400

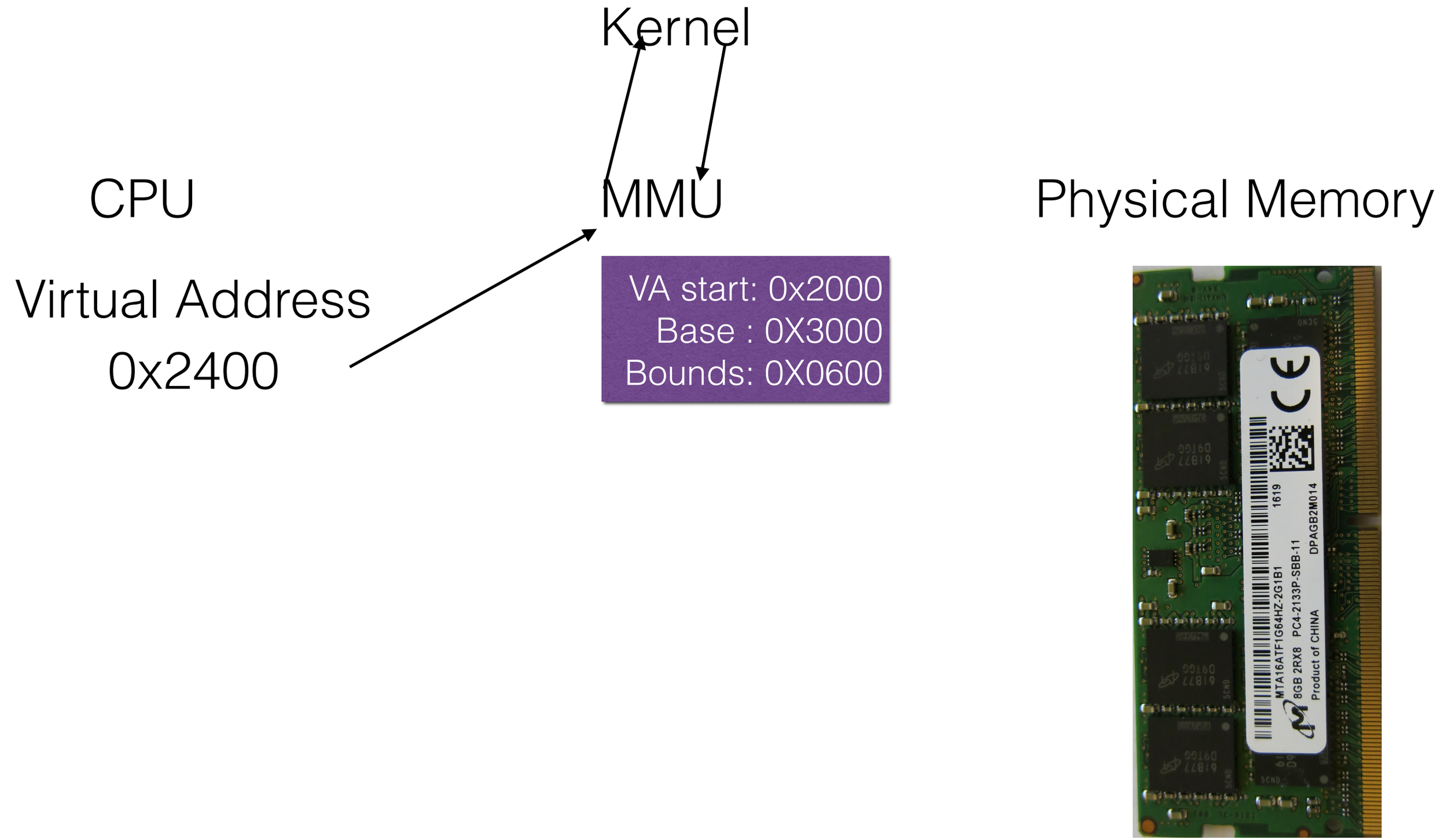
# Segmentation Example



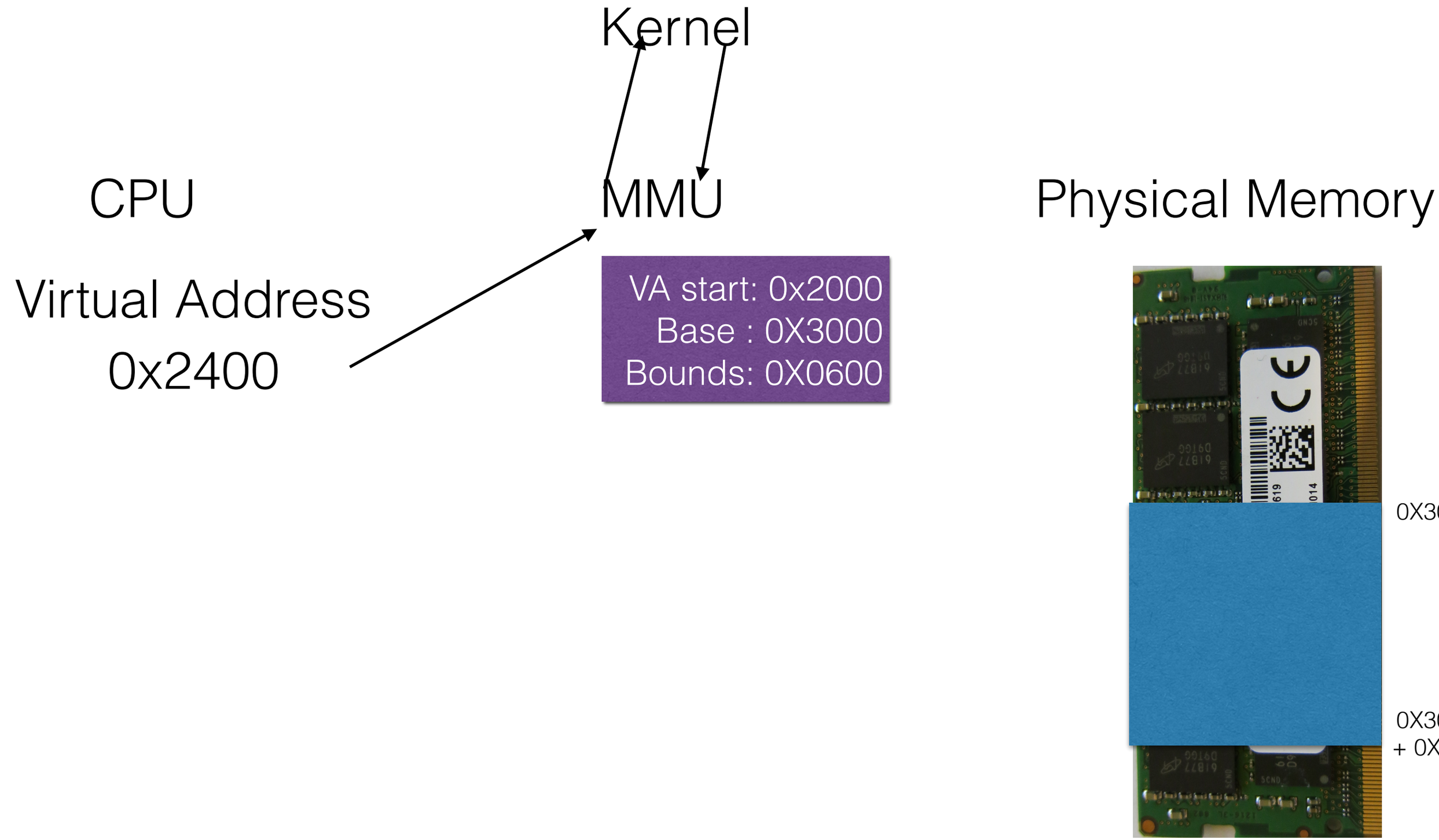
# Segmentation Example



# Segmentation Example



# Segmentation Example



# Segmentation Example

Kernel

CPU

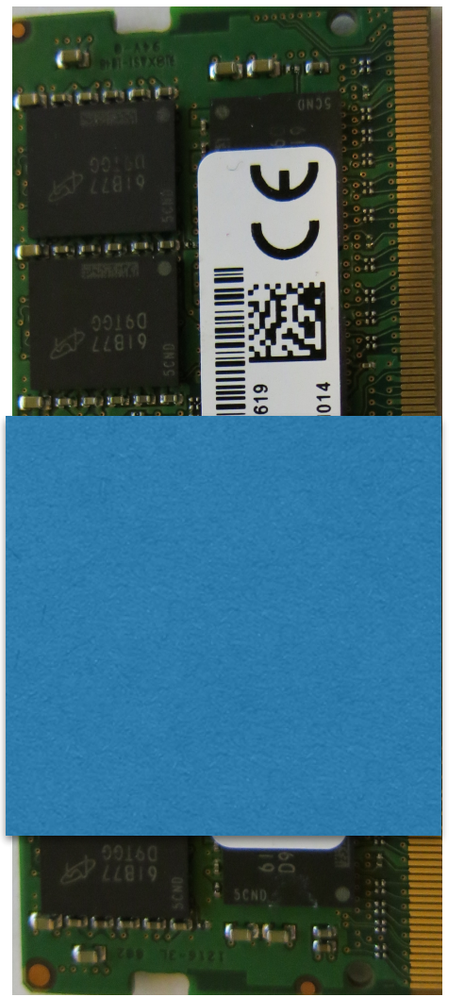
MMU

Physical Memory

Virtual Address

0x2400

VA start: 0x2000
Base : 0X3000
Bounds: 0X0600



0X3000

0X3000
+ 0X600

**Check:**Segment Start < V.A. < Segment Start + Segment Bound.

# Segmentation Example

Kernel

CPU

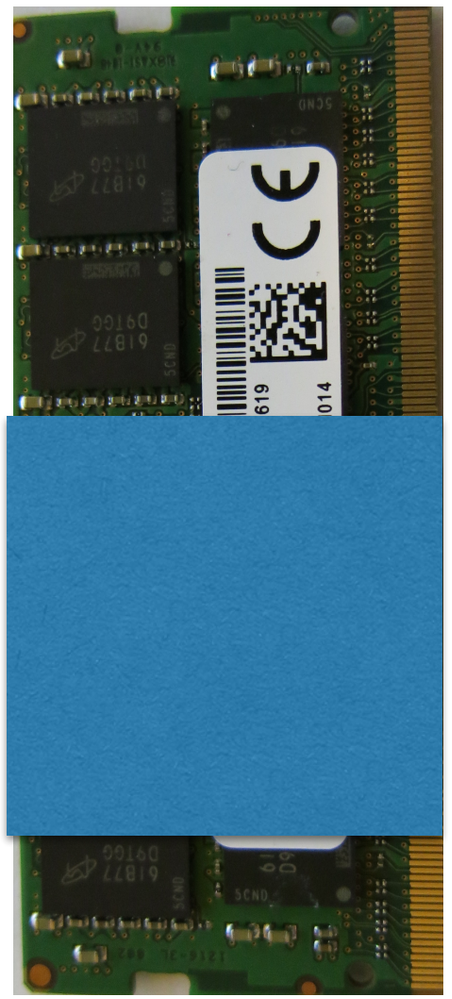
MMU

Physical Memory

Virtual Address

0x2400

VA start: 0x2000
Base : 0X3000
Bounds: 0X0600



0X3000

0X3000
+ 0X600

**Check:**Segment Start < V.A. < Segment Start + Segment Bound.

**Check:**0X2000 <0X2400 <0X2400+0X600

# Segmentation Example

Kernel

CPU

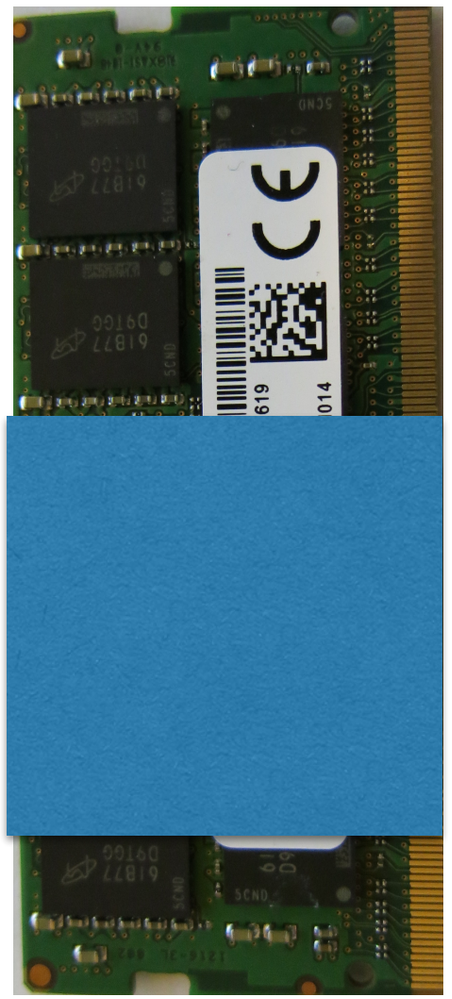
MMU

Physical Memory

Virtual Address

0x2400

VA start: 0x2000
Base : 0X3000
Bounds: 0X0600



0X3000

0X3000
+ 0X600

**Check:**Segment Start < V.A. < Segment Start + Segment Bound.

**Check:**0X2000 <0X2400 <0X2400+0X600

# Segmentation Example

Kernel

CPU

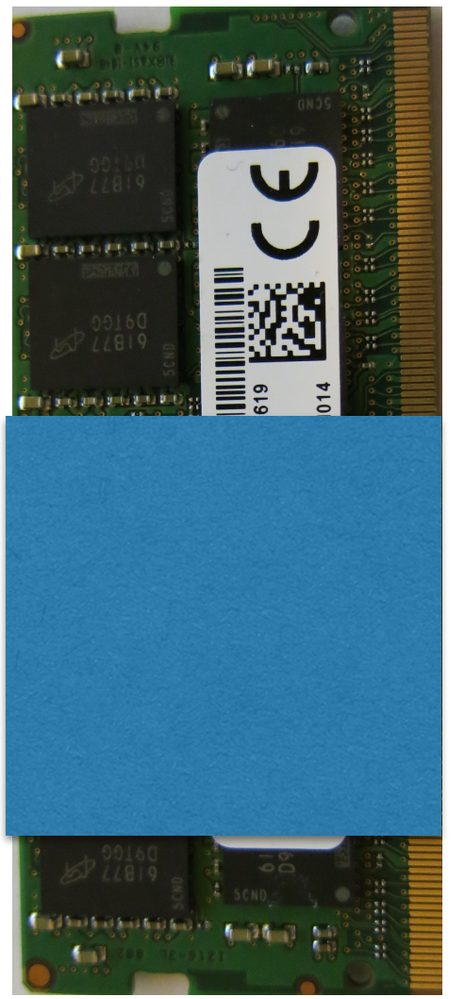
MMU

Physical Memory

Virtual Address

0x2400

VA start: 0x2000
Base : 0X3000
Bounds: 0X0600



0X3000

0X3000
+ 0X600

**Check:**Segment Start < V.A. < Segment Start + Segment Bound.

**Check:**0X2000 <0X2400 <0X2400+0X600

Physical Address = (V.A. - Segment Start) + Segment Base

# Segmentation Example

Kernel

CPU

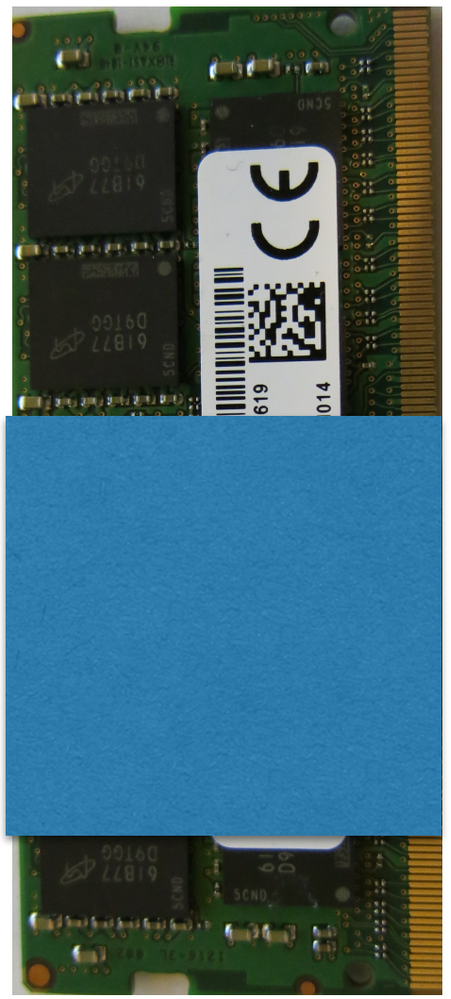
MMU

Physical Memory

Virtual Address

0x2400

VA start: 0x2000
Base : 0X3000
Bounds: 0X0600



0X3000

0X3000
+ 0X600

**Check:**Segment Start < V.A. < Segment Start + Segment Bound.

**Check:**0X2000 <0X2400 <0X2400+0X600

Physical Address = (V.A. - Segment Start) + Segment Base

Physical Address = (0x2400 - 0x2000) + 0x3000

# Segmentation Example

Kernel

CPU

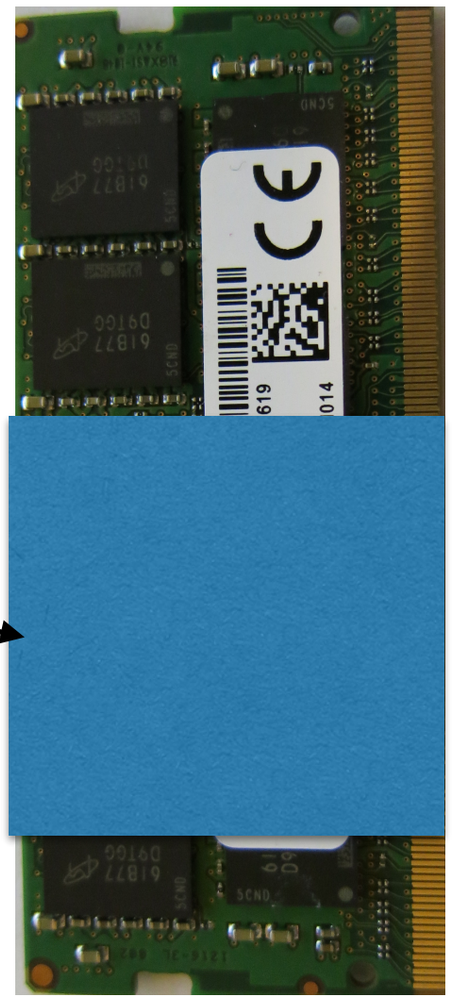
MMU

Physical Memory

Virtual Address

0x2400

VA start: 0x2000
Base : 0X3000
Bounds: 0X0600



0X3000

0X3000
+ 0X600

**Check:**Segment Start < V.A. < Segment Start + Segment Bound.

**Check:**0X2000 <0X2400 <0X2400+0X600

Physical Address = (V.A. - Segment Start) + Segment Base

Physical Address = (0x2400 - 0x2000) + 0x3000

# Segmentation Example

Kernel

CPU

MMU

Physical Memory

Virtual Address

0x2700

# Segmentation Example

Kernel

CPU

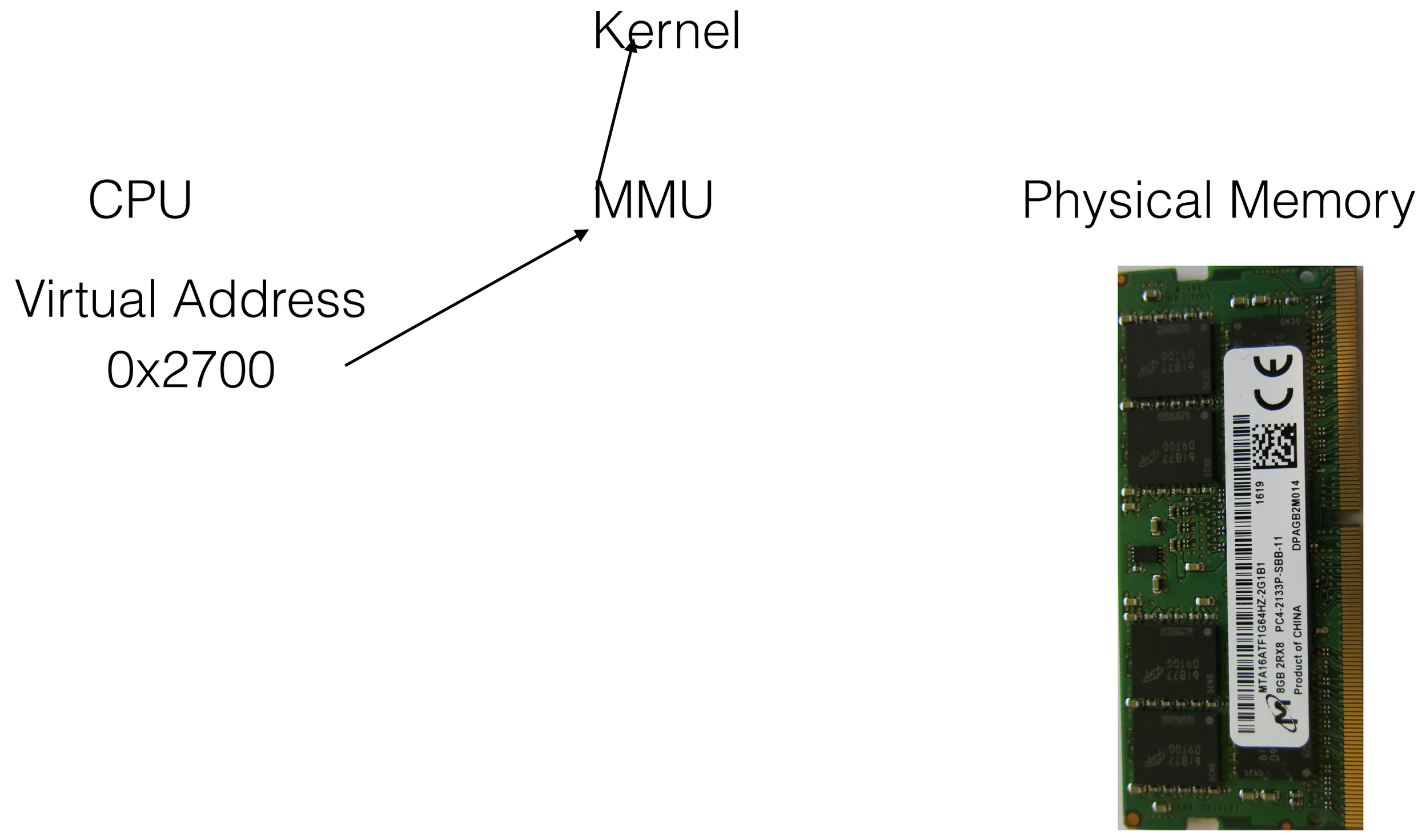
MMU

Physical Memory

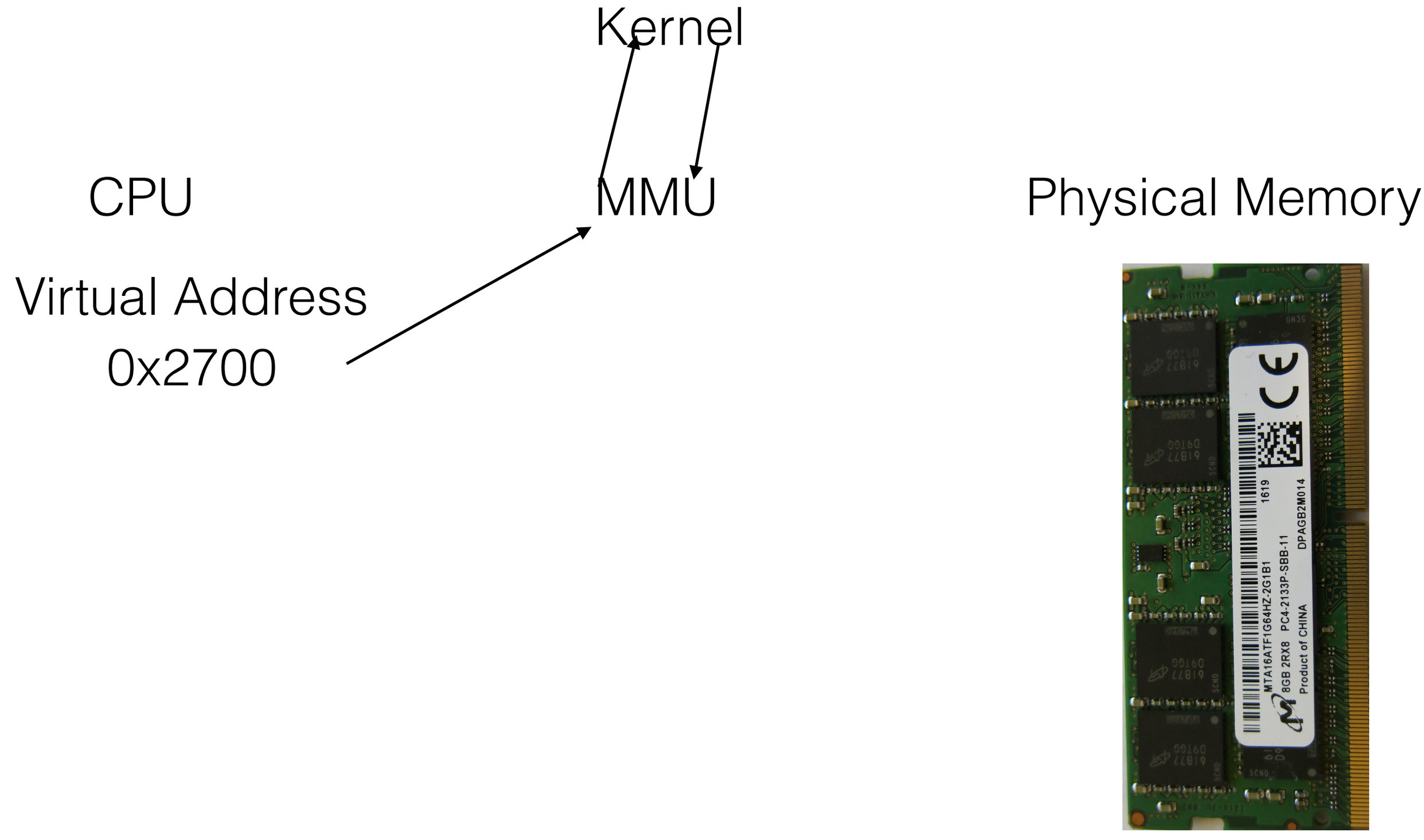
Virtual Address

0x2700

# Segmentation Example



# Segmentation Example



# Segmentation Example

Kernel

VA start: 0x0000
Base : 0X1000
Bounds: 0X0300

CPU

MMU

Physical Memory

Virtual Address

0x2700

# Segmentation Example

Kernel

VA start: 0x0000
Base : 0X1000
Bounds: 0X0300

VA start: 0x2000
Base : 0X3000
Bounds: 0X0600

CPU

MMU

Physical Memory

Virtual Address
0x2700

# Segmentation Example

Kernel

VA start: 0x0000
Base : 0X1000
Bounds: 0X0300

VA start: 0x2000
Base : 0X3000
Bounds: 0X0600

CPU

MMU

Physical Memory

Virtual Address
0x2700


**Check:**Segment Start < V.A. < Segment Start + Segment Bound.

# Segmentation Example

Kernel

VA start: 0x0000
Base : 0X1000
Bounds: 0X0300

VA start: 0x2000
Base : 0X3000
Bounds: 0X0600

CPU

MMU

Physical Memory

Virtual Address
0x2700


**Check:**Segment Start < V.A. < Segment Start + Segment Bound.

# Segmentation Example

Kernel

VA start: 0x0000
Base : 0X1000
Bounds: 0X0300

VA start: 0x2000
Base : 0X3000
Bounds: 0X0600

CPU

MMU

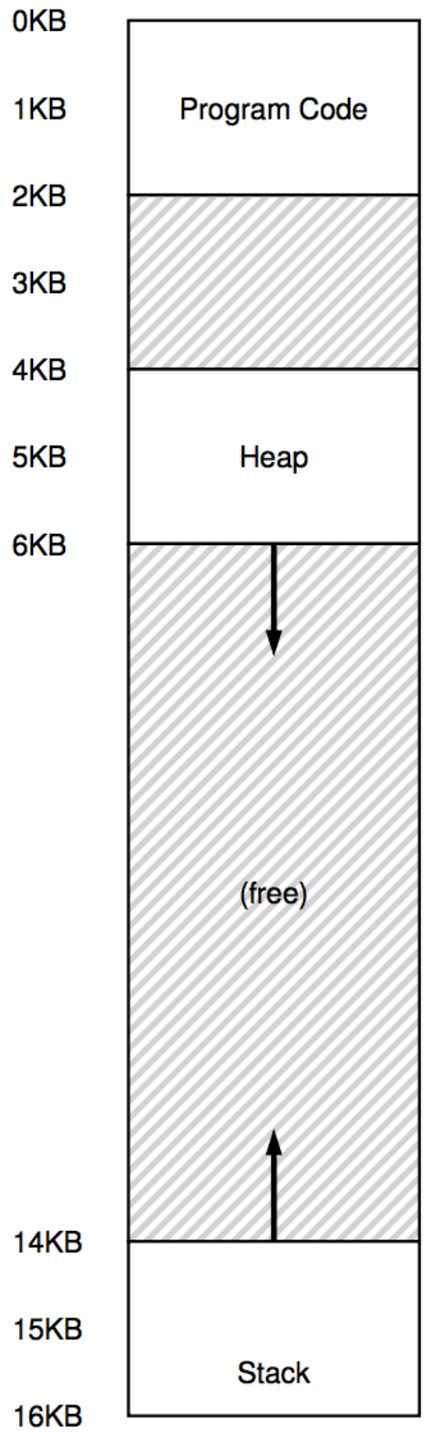
Physical Memory

Virtual Address
0x2700


**Check:**Segment Start < V.A. < Segment Start + Segment Bound.

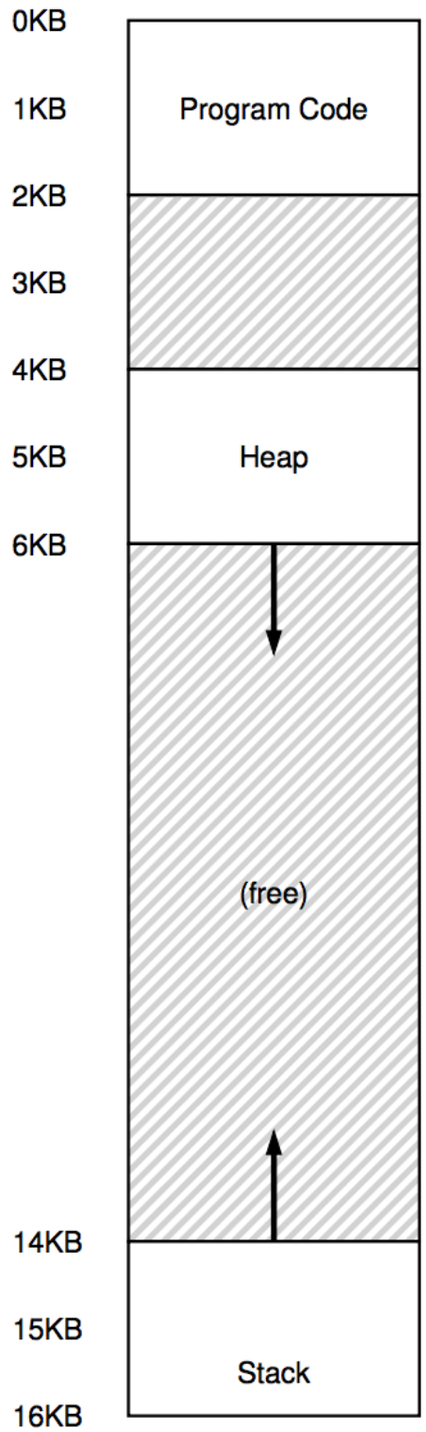
# Segment Reference

Virtual Address Space



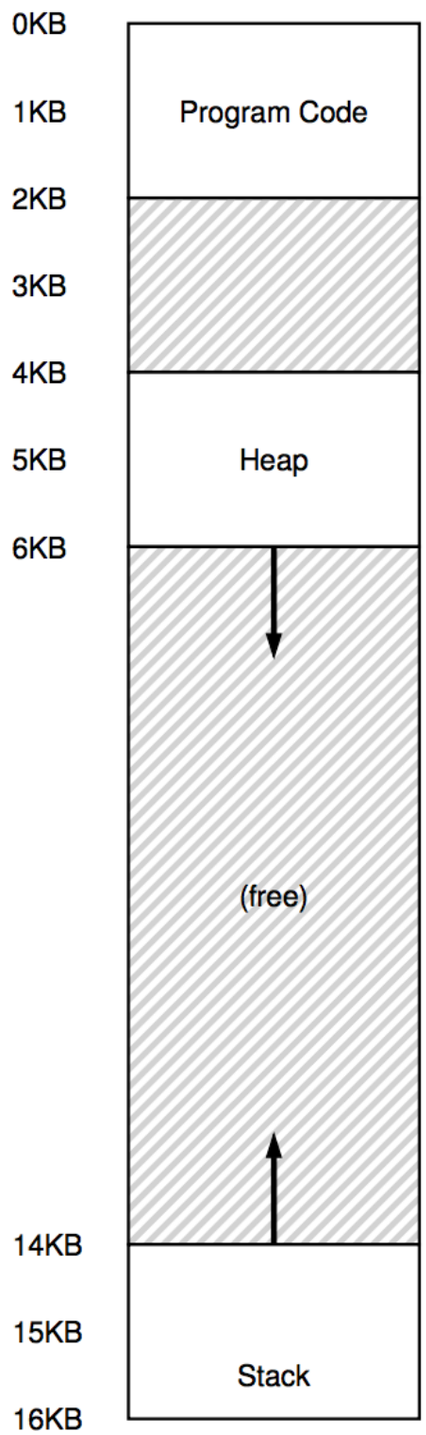
# Segment Reference

Virtual Address Space




# Segment Reference

Virtual Address Space

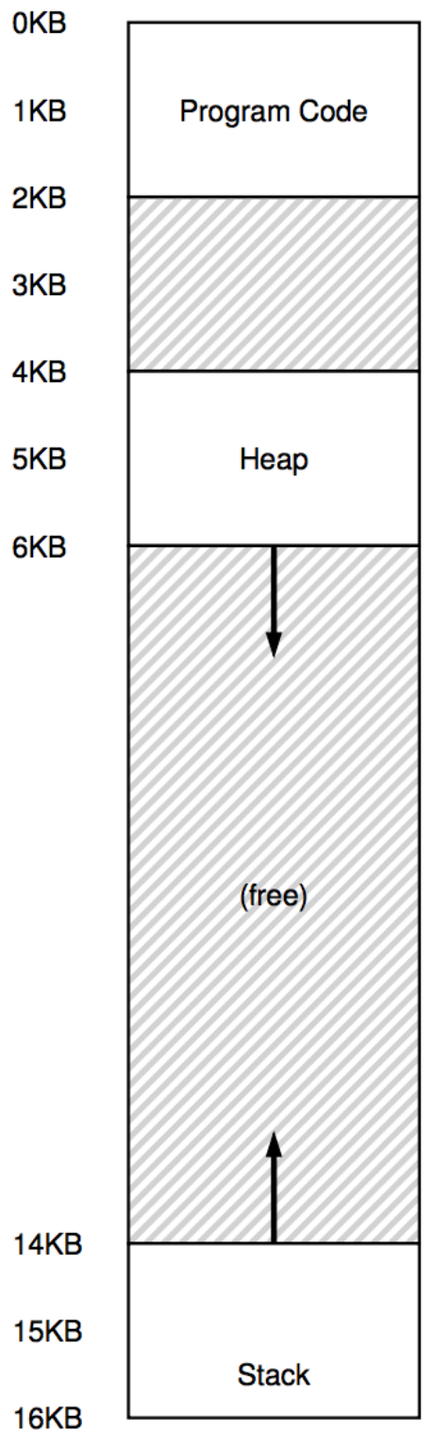



00 | 0000 | 0000 | 0000

11 | 1111 | 1111 | 1111

# Segment Reference

Virtual Address Space

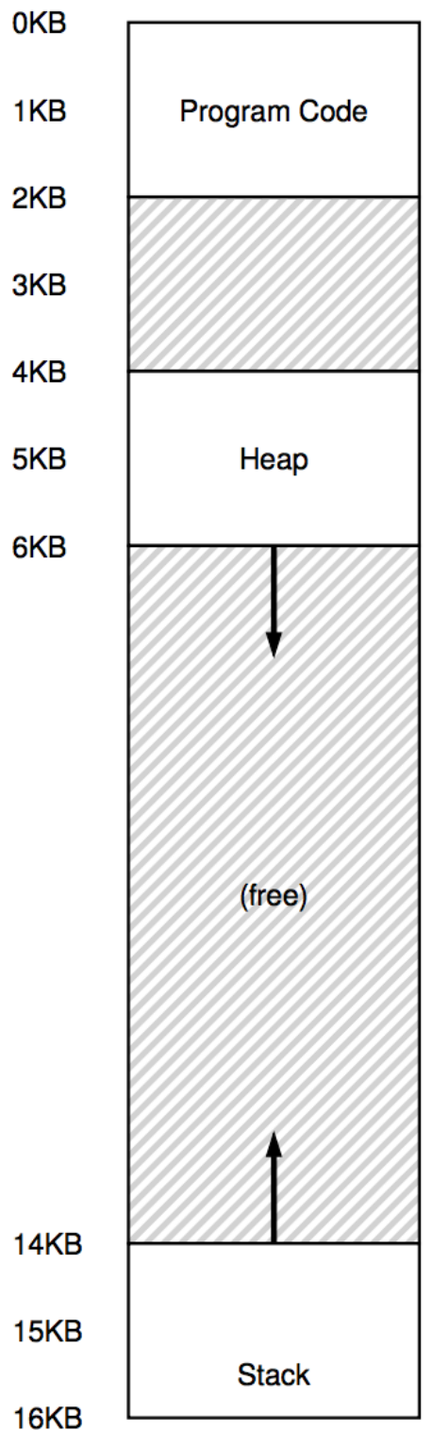



| 00 | 0000 | 0000 | 0000 |
|---|---|---|---|
| 00 | 0111 | 1111 | 1111 |

| 11 | 1111 | 1111 | 1111 |
|---|---|---|---|

# Segment Reference

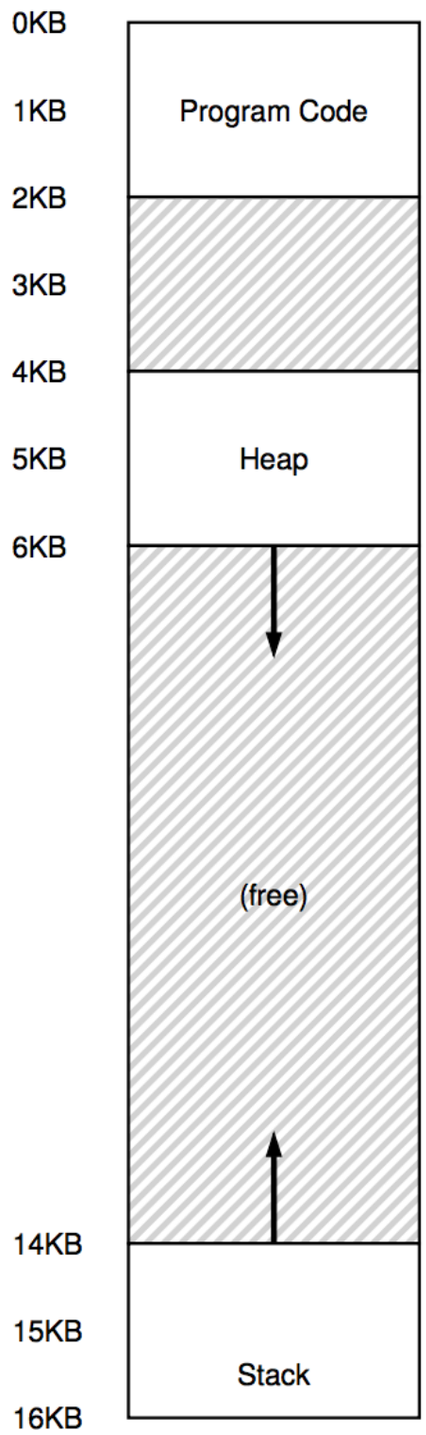
Virtual Address Space




| | | | |
|---|---|---|---|
| 00 | 0000 | 0000 | 0000 |
| 00 | 0111 | 1111 | 1111 |
| 01 | 0000 | 0000 | 0000 |
| 11 | 1111 | 1111 | 1111 |

# Segment Reference

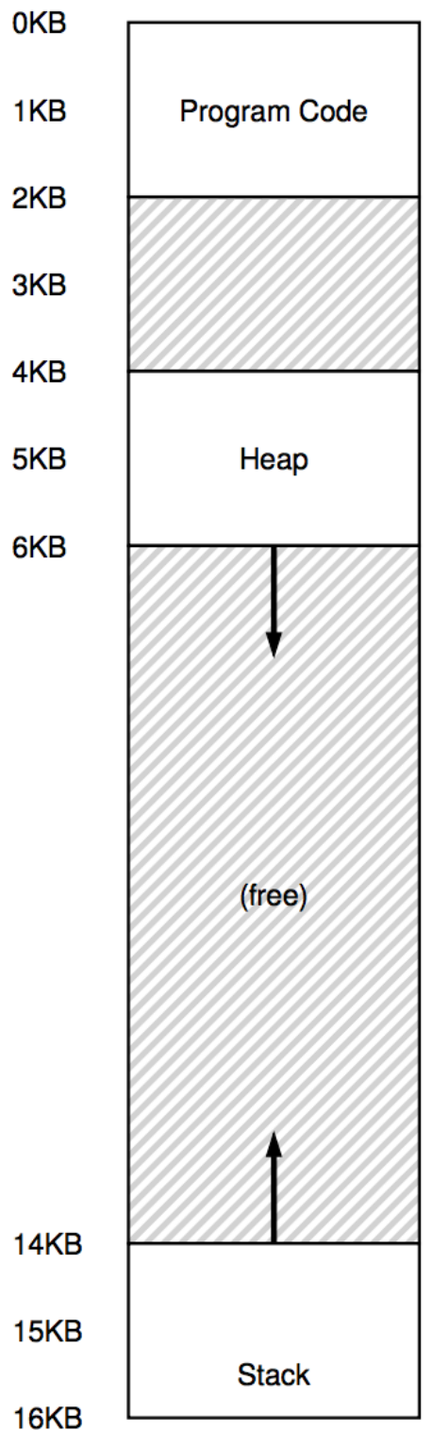
Virtual Address Space




| | | | |
|---|---|---|---|
| 00 | 0000 | 0000 | 0000 |
| 00 | 0111 | 1111 | 1111 |
| 01 | 0000 | 0000 | 0000 |
| 01 | 0111 | 1111 | 1111 |
| 11 | 1111 | 1111 | 1111 |

# Segment Reference

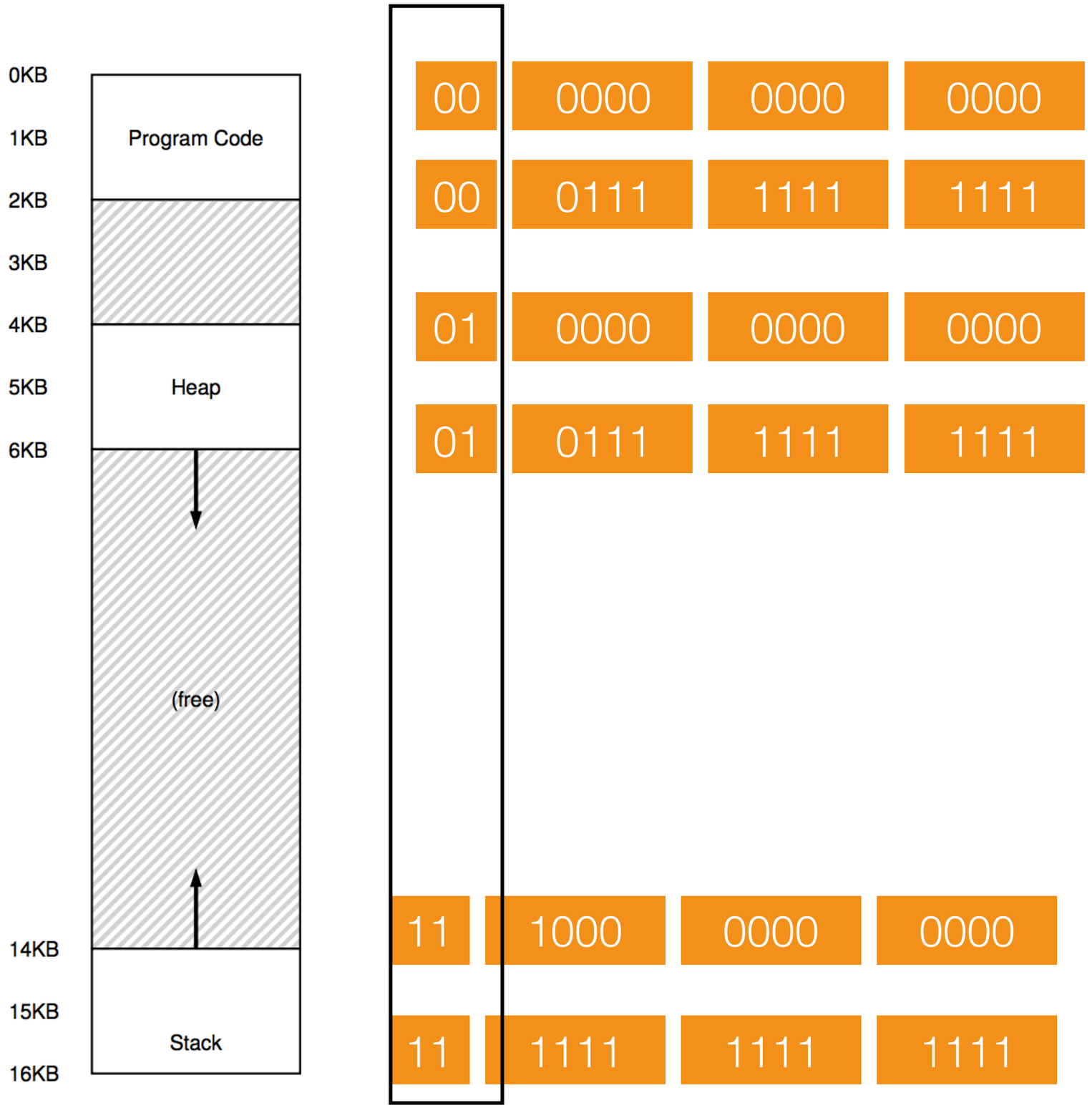
Virtual Address Space




| | | | |
|---|---|---|---|
| 00 | 0000 | 0000 | 0000 |
| 00 | 0111 | 1111 | 1111 |
| 01 | 0000 | 0000 | 0000 |
| 01 | 0111 | 1111 | 1111 |
| 11 | 1000 | 0000 | 0000 |
| 11 | 1111 | 1111 | 1111 |

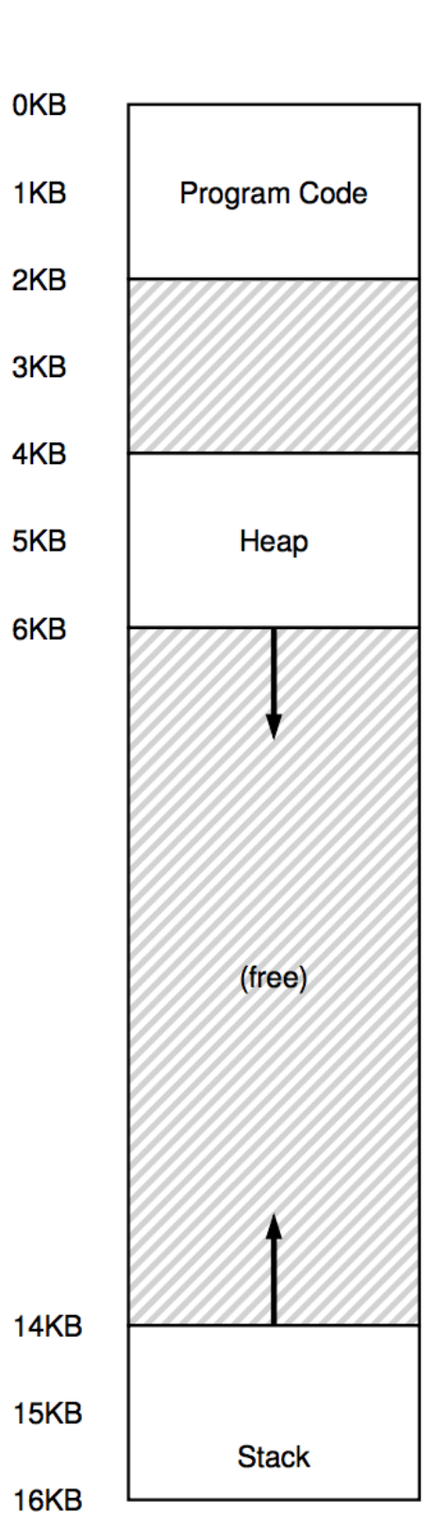
# Segment Reference

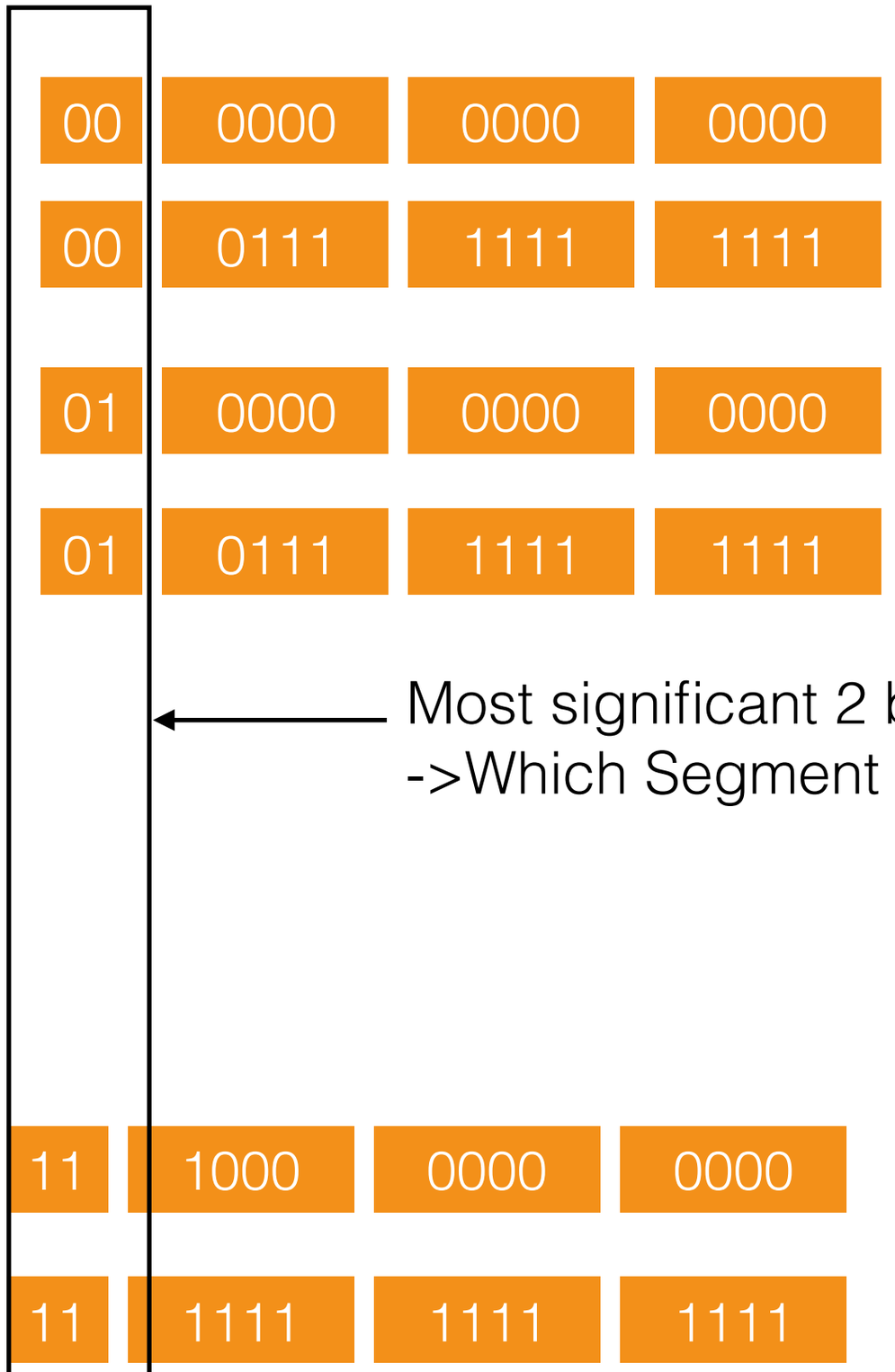
Virtual Address Space



# Segment Reference

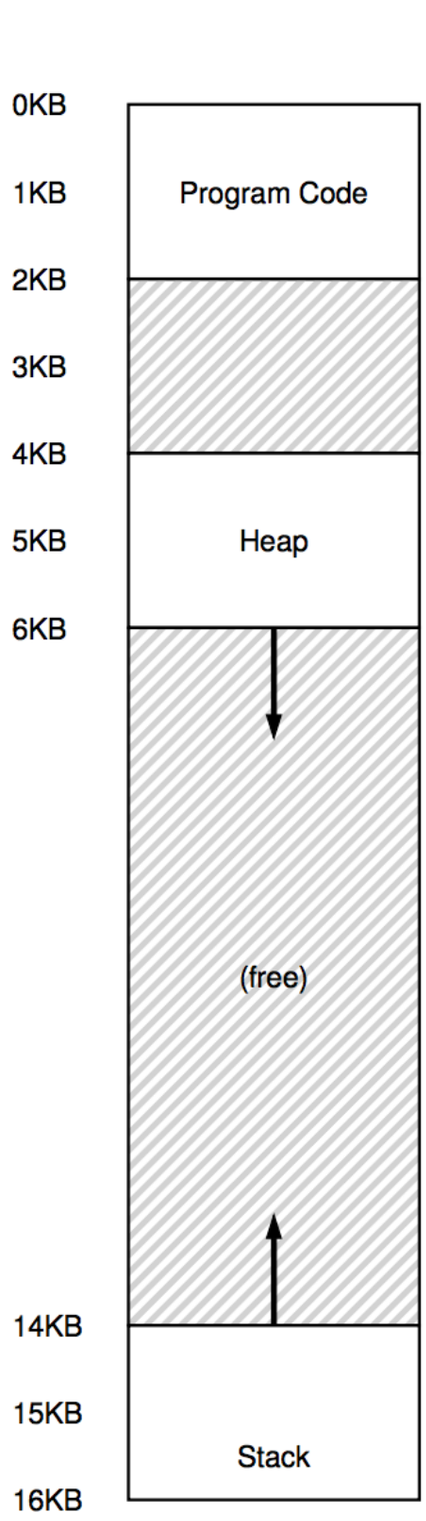
Virtual Address Space

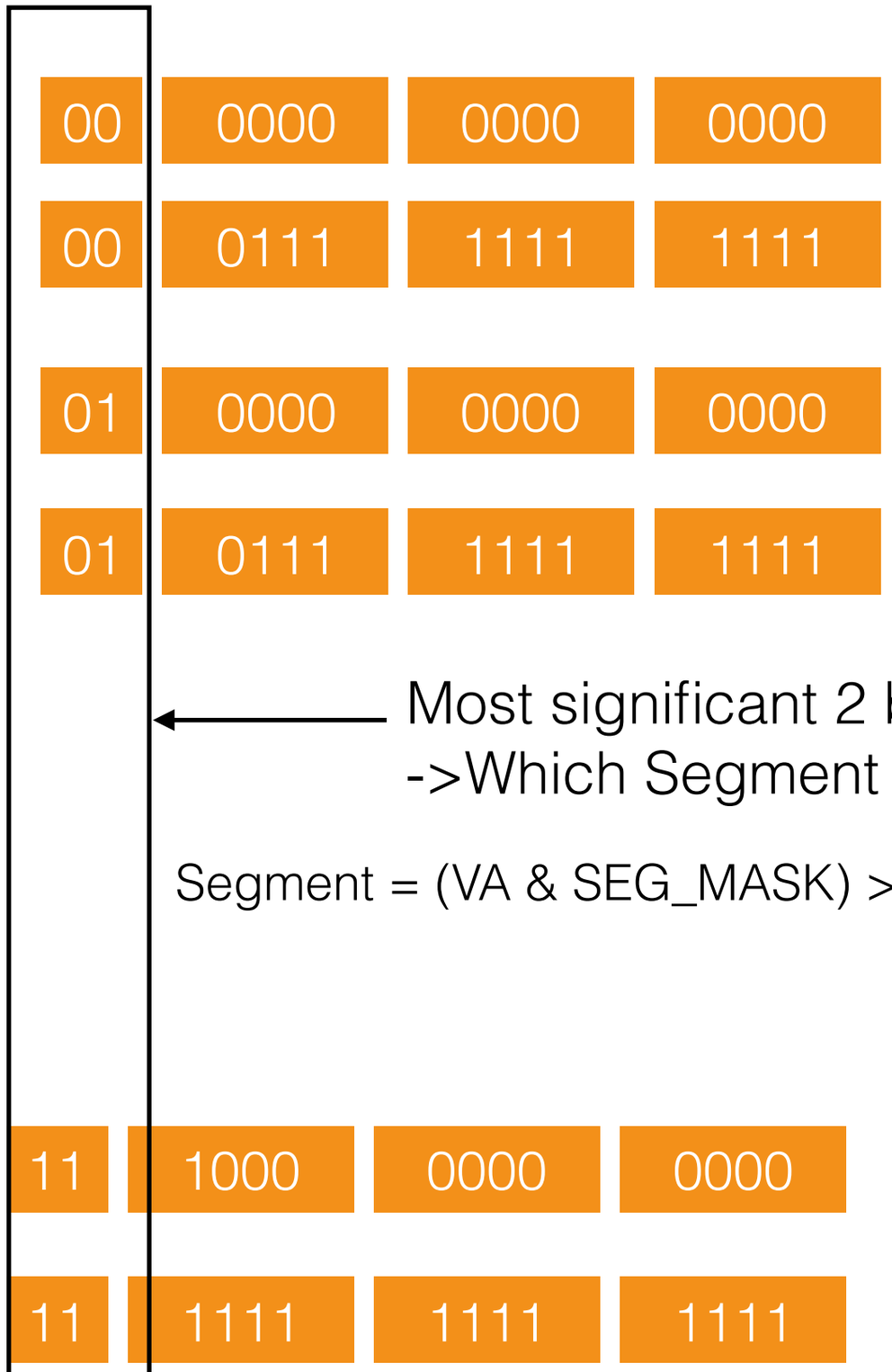


# Segment Reference

Virtual Address Space



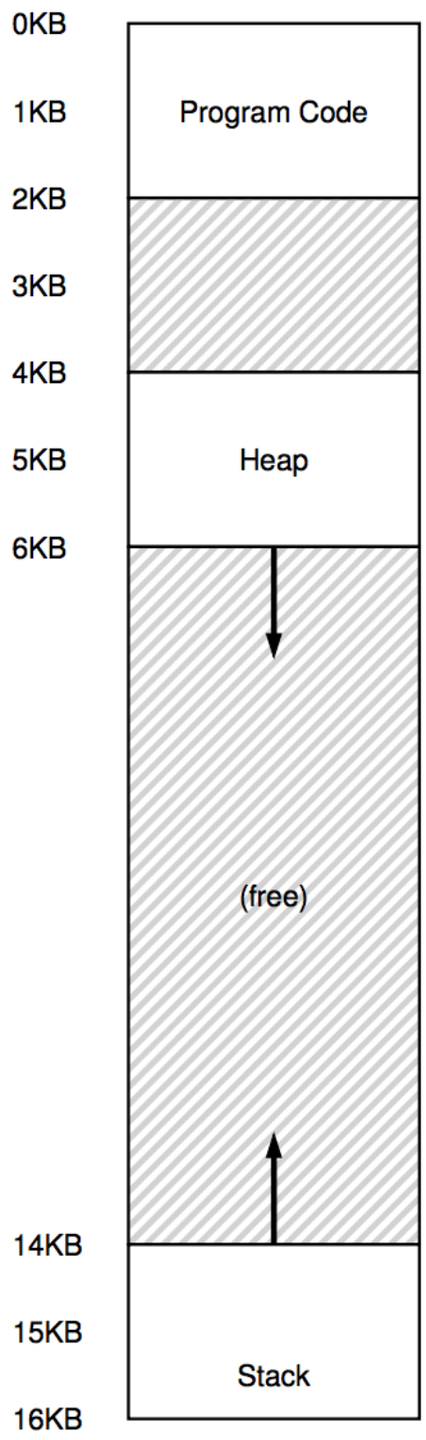



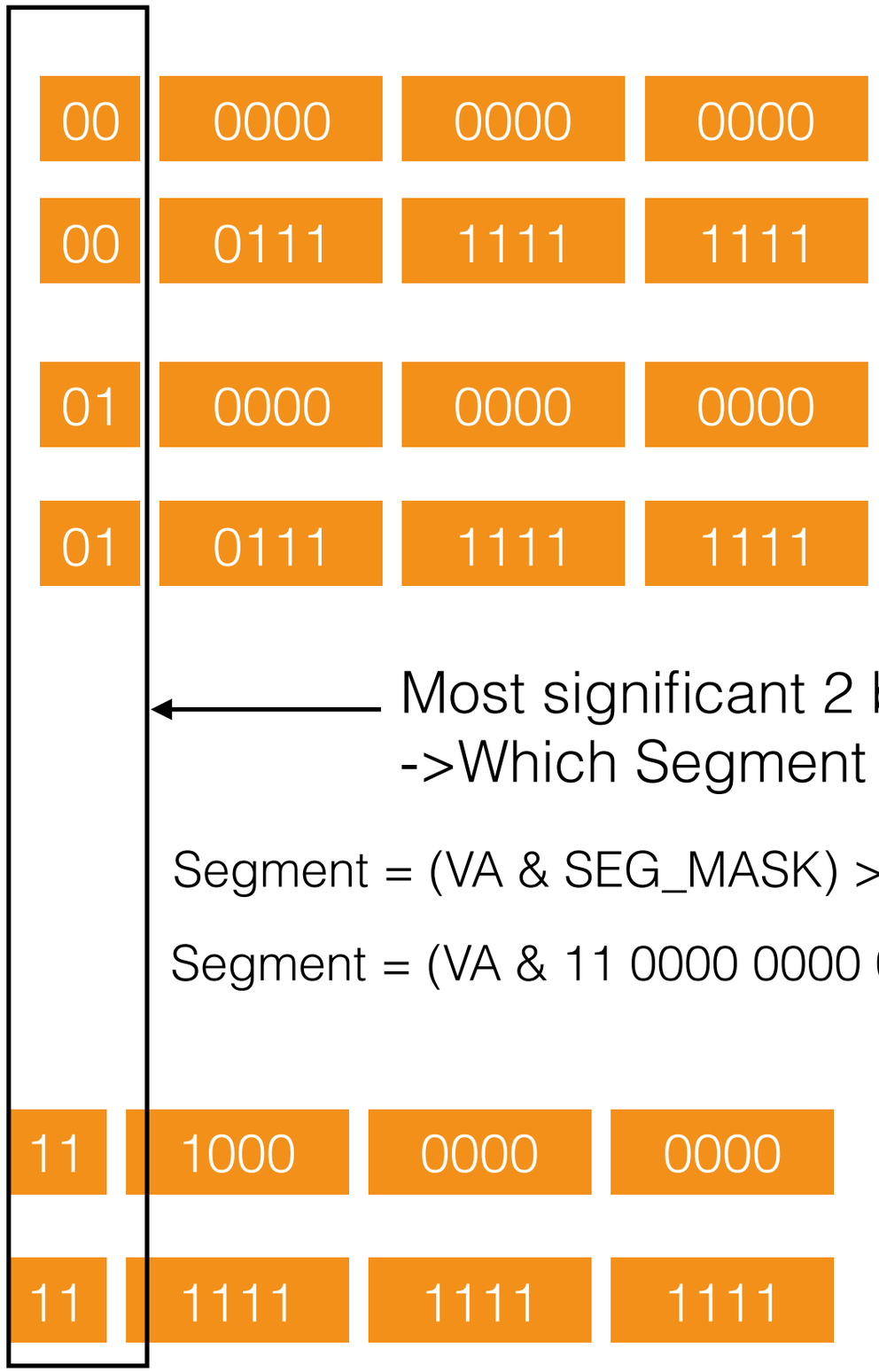

Most significant 2 bits
->Which Segment

Segment = (VA & SEG_MASK) >>SEG_SHIFT

# Segment Reference

Virtual Address Space



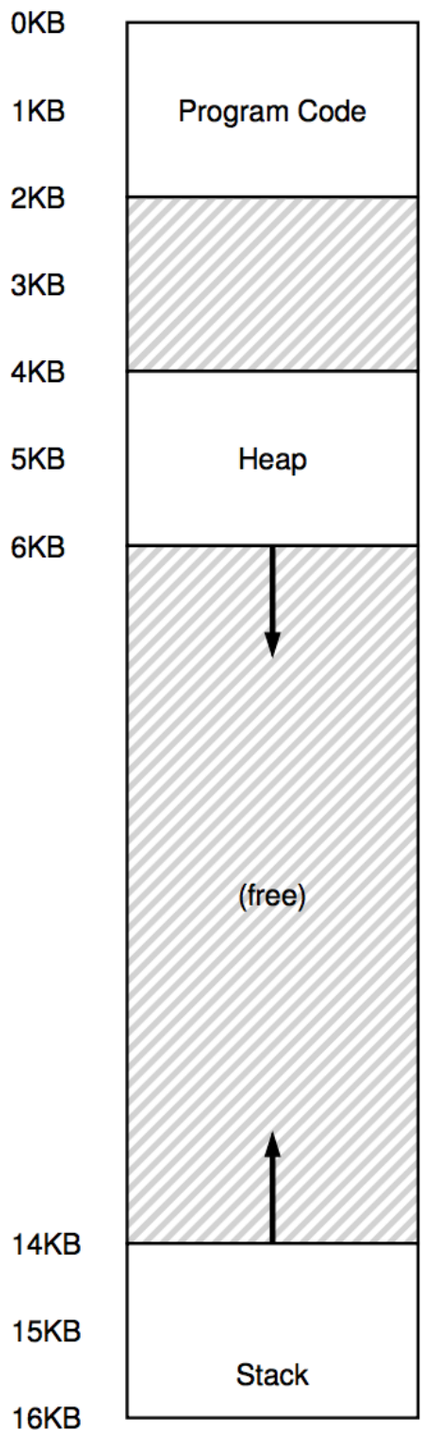




Most significant 2 bits
->Which Segment

Segment = (VA & SEG_MASK) >>SEG_SHIFT

Segment = (VA & 11 0000 0000 0000) >>12

# Segment Reference




00 0000 0000 0000

00 0111 1111 1111

01 0000 0000 0000

01 0111 1111 1111

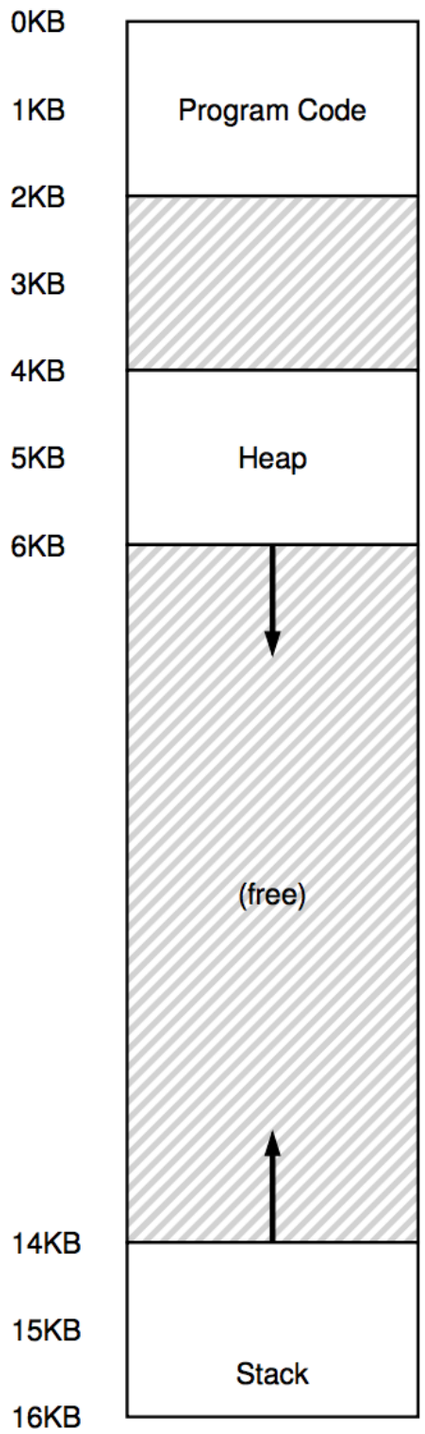
Segment = (VA & 11 0000 0000 0000) >>12

11 1000 0000 0000

11 1111 1111 1111

# Segment Reference

00 | 0000 | 0000 | 1010




00 | 0000 | 0000 | 0000

00 | 0111 | 1111 | 1111

01 | 0000 | 0000 | 0000

01 | 0111 | 1111 | 1111

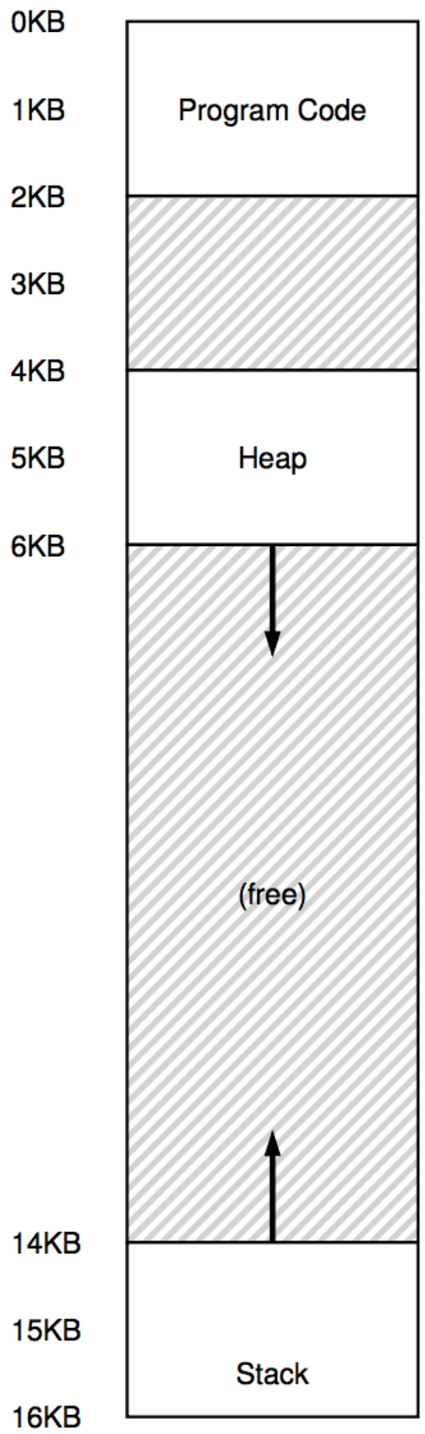
Segment = (VA & 11 0000 0000 0000) >>12

11 | 1000 | 0000 | 0000

11 | 1111 | 1111 | 1111

# Segment Reference

| 00 | 0000 | 0000 | 1010 |
|---|---|---|---|
| 11 | 0000 | 0000 | 0000 |

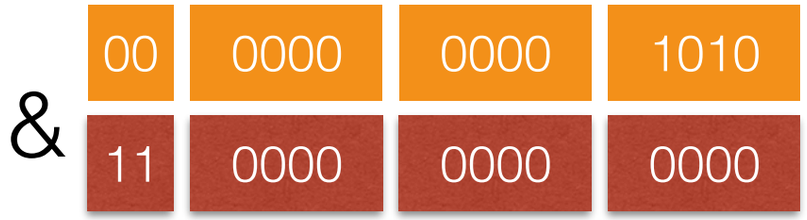


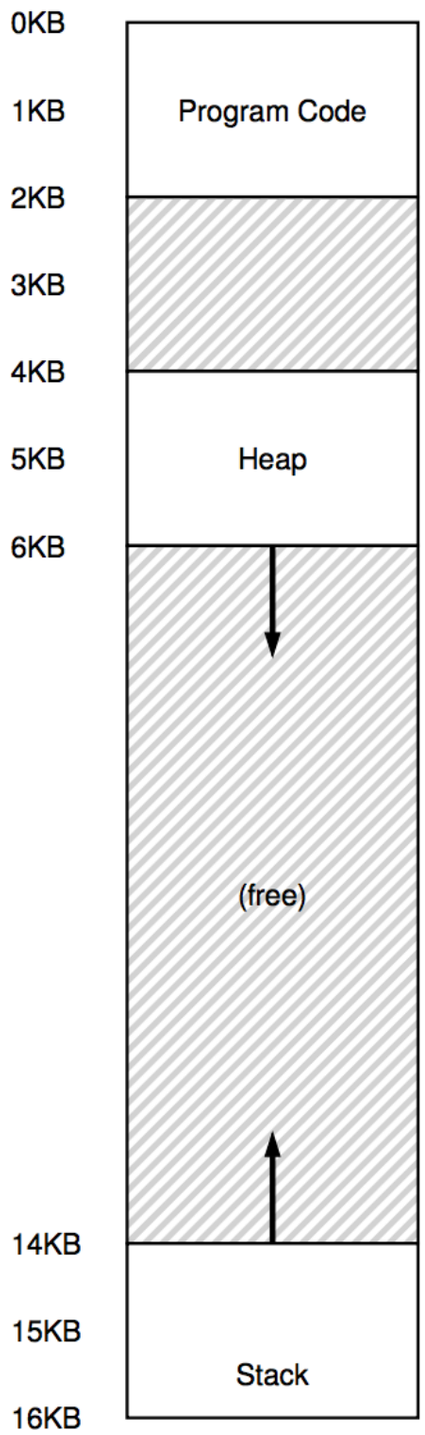

| 00 | 0000 | 0000 | 0000 |
|---|---|---|---|
| 00 | 0111 | 1111 | 1111 |
| 01 | 0000 | 0000 | 0000 |
| 01 | 0111 | 1111 | 1111 |

Segment = (VA & 11 0000 0000 0000) >>12

| 11 | 1000 | 0000 | 0000 |
|---|---|---|---|
| 11 | 1111 | 1111 | 1111 |

# Segment Reference




00 0000 0000 0000

00 0111 1111 1111

01 0000 0000 0000

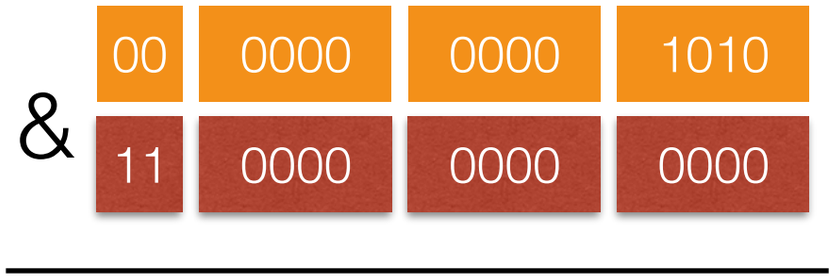
01 0111 1111 1111

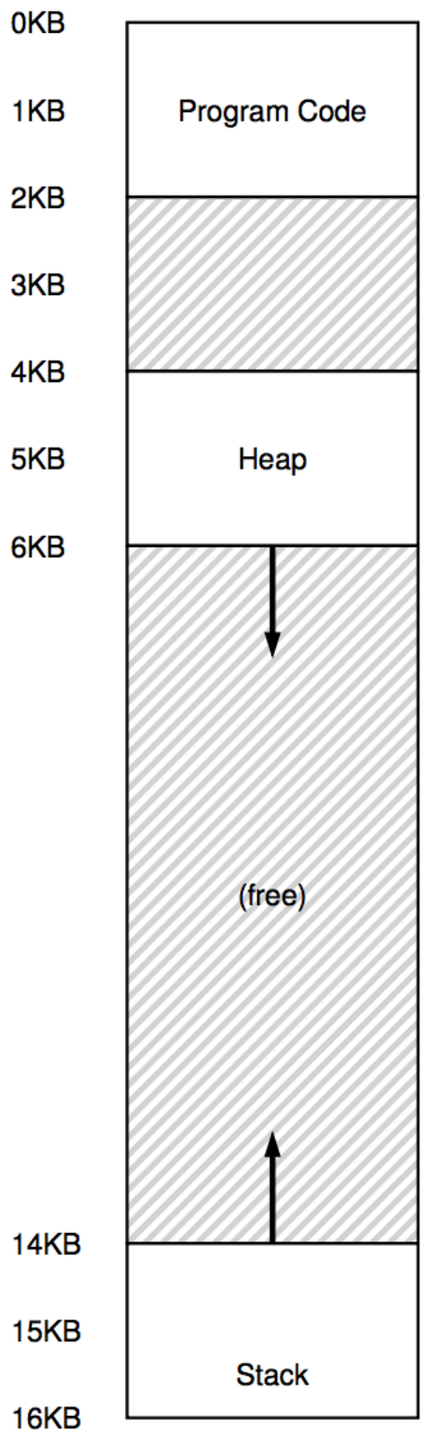
Segment = (VA & 11 0000 0000 0000) >>12

11 1000 0000 0000

11 1111 1111 1111

# Segment Reference




00 0000 0000 0000

00 0111 1111 1111

01 0000 0000 0000

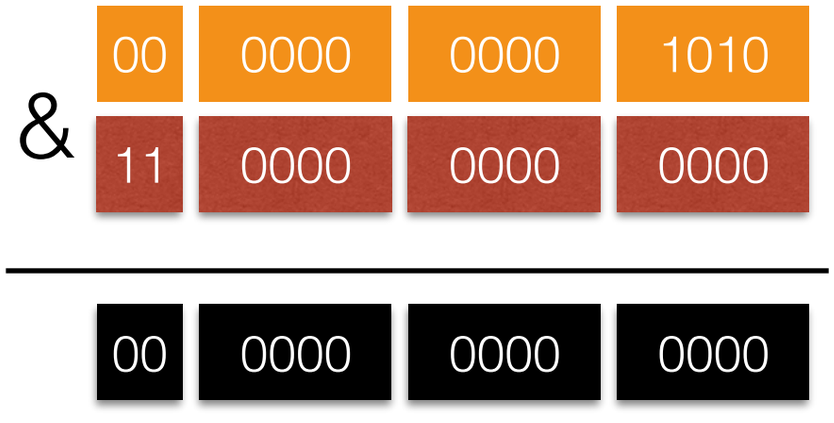
01 0111 1111 1111

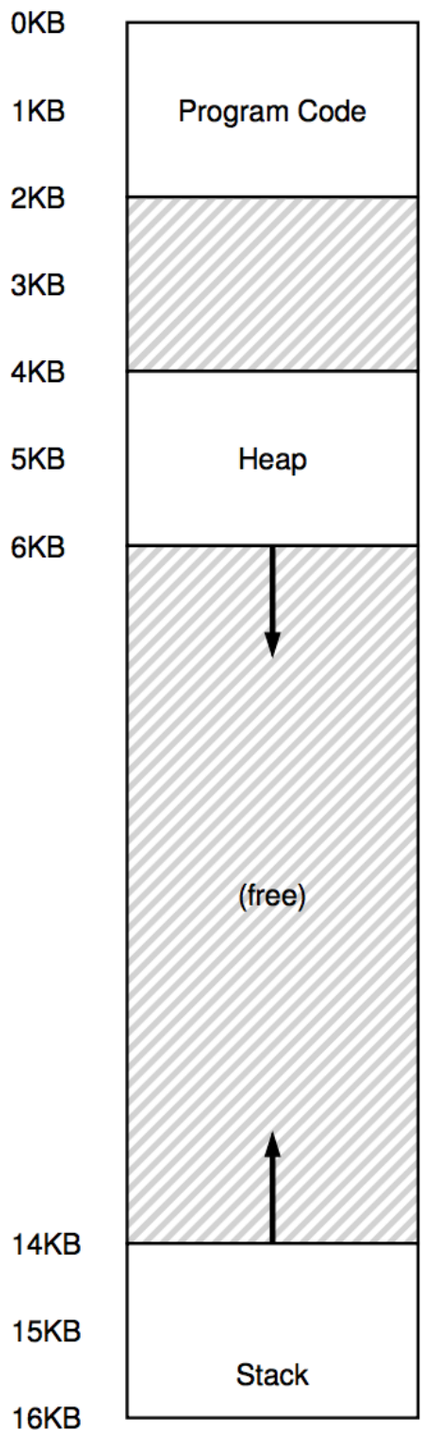
Segment = (VA & 11 0000 0000 0000) >>12

11 1000 0000 0000

11 1111 1111 1111

# Segment Reference




00 0000 0000 0000

00 0111 1111 1111

01 0000 0000 0000

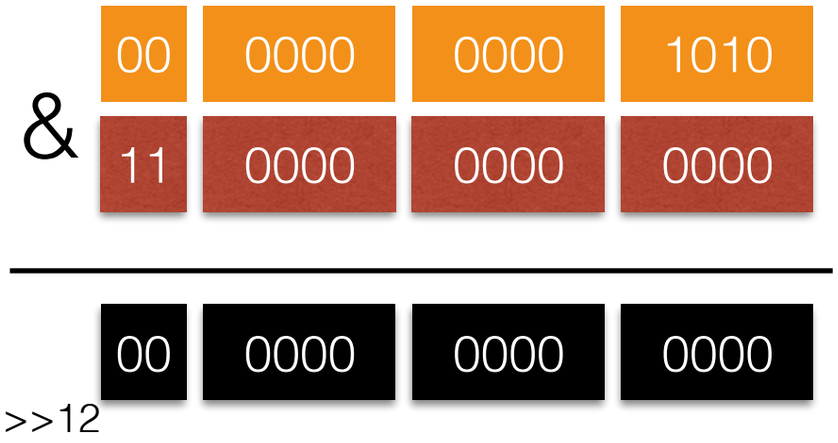
01 0111 1111 1111

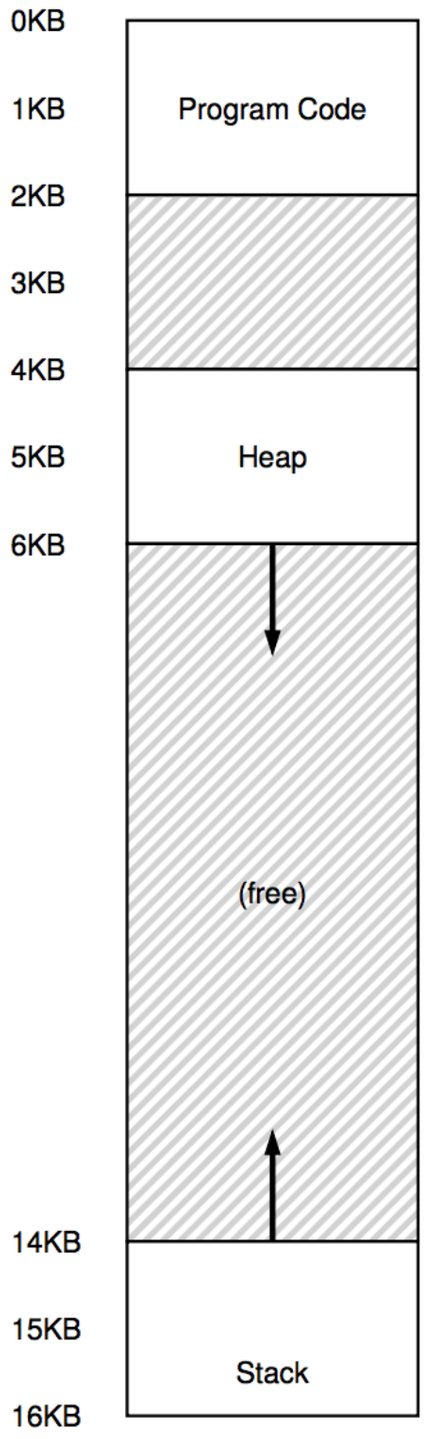
Segment = (VA & 11 0000 0000 0000) >>12

11 1000 0000 0000

11 1111 1111 1111

# Segment Reference




00 0000 0000 0000

00 0111 1111 1111

01 0000 0000 0000

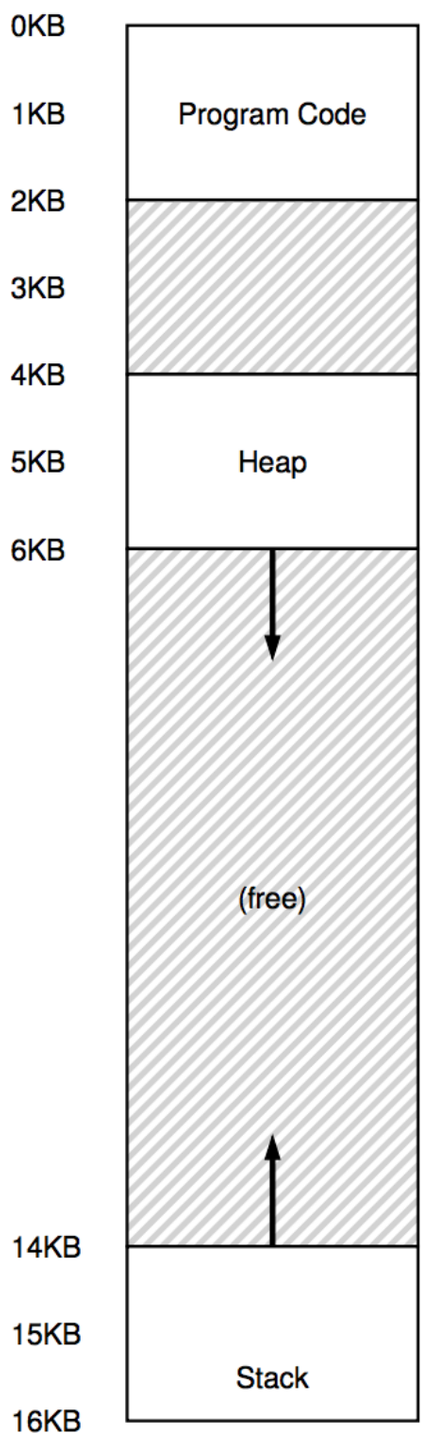
01 0111 1111 1111

Segment = (VA & 11 0000 0000 0000) >>12

11 1000 0000 0000

11 1111 1111 1111

# Segment Reference


00 0000 0000 0000

00 0111 1111 1111

01 0000 0000 0000

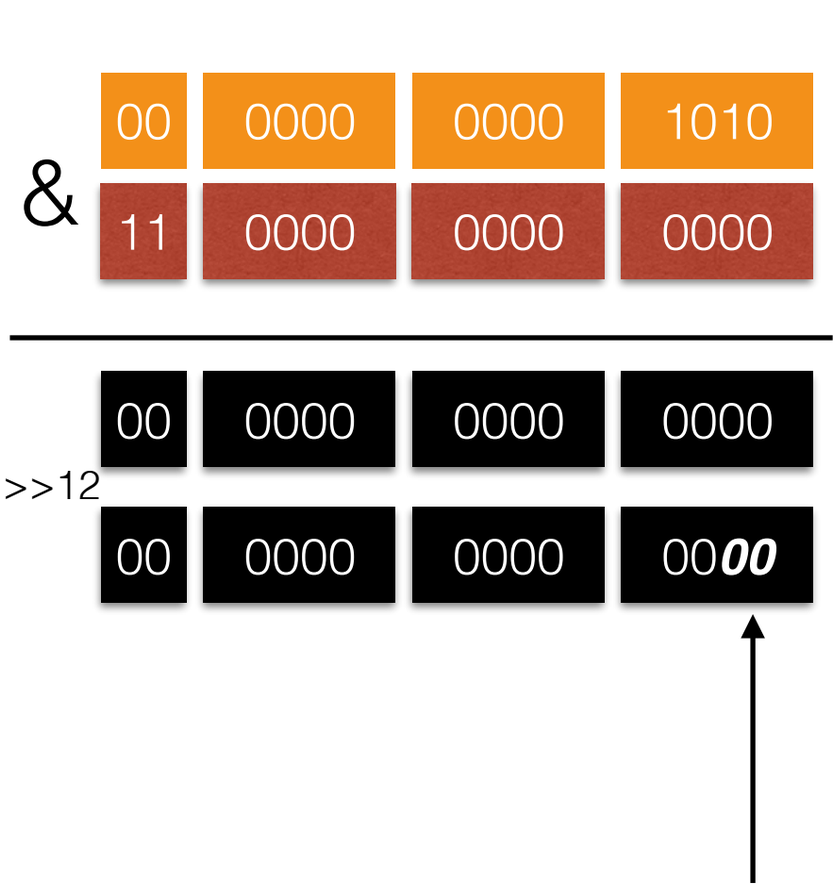
01 0111 1111 1111

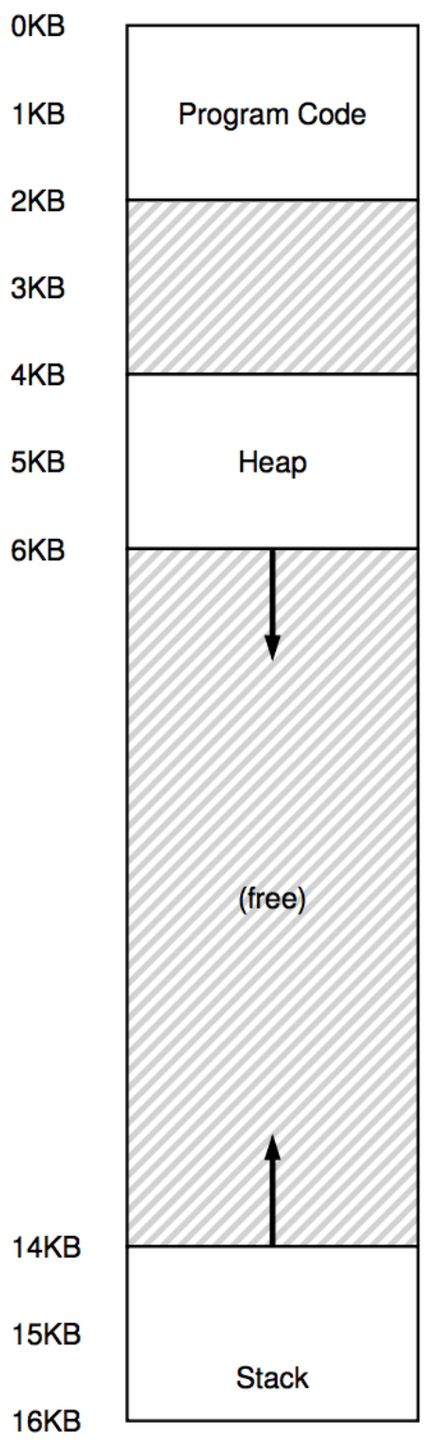
Segment = (VA & 11 0000 0000 0000) >>12

11 1000 0000 0000

11 1111 1111 1111

# Segment Reference




00 0000 0000 0000

00 0111 1111 1111

01 0000 0000 0000

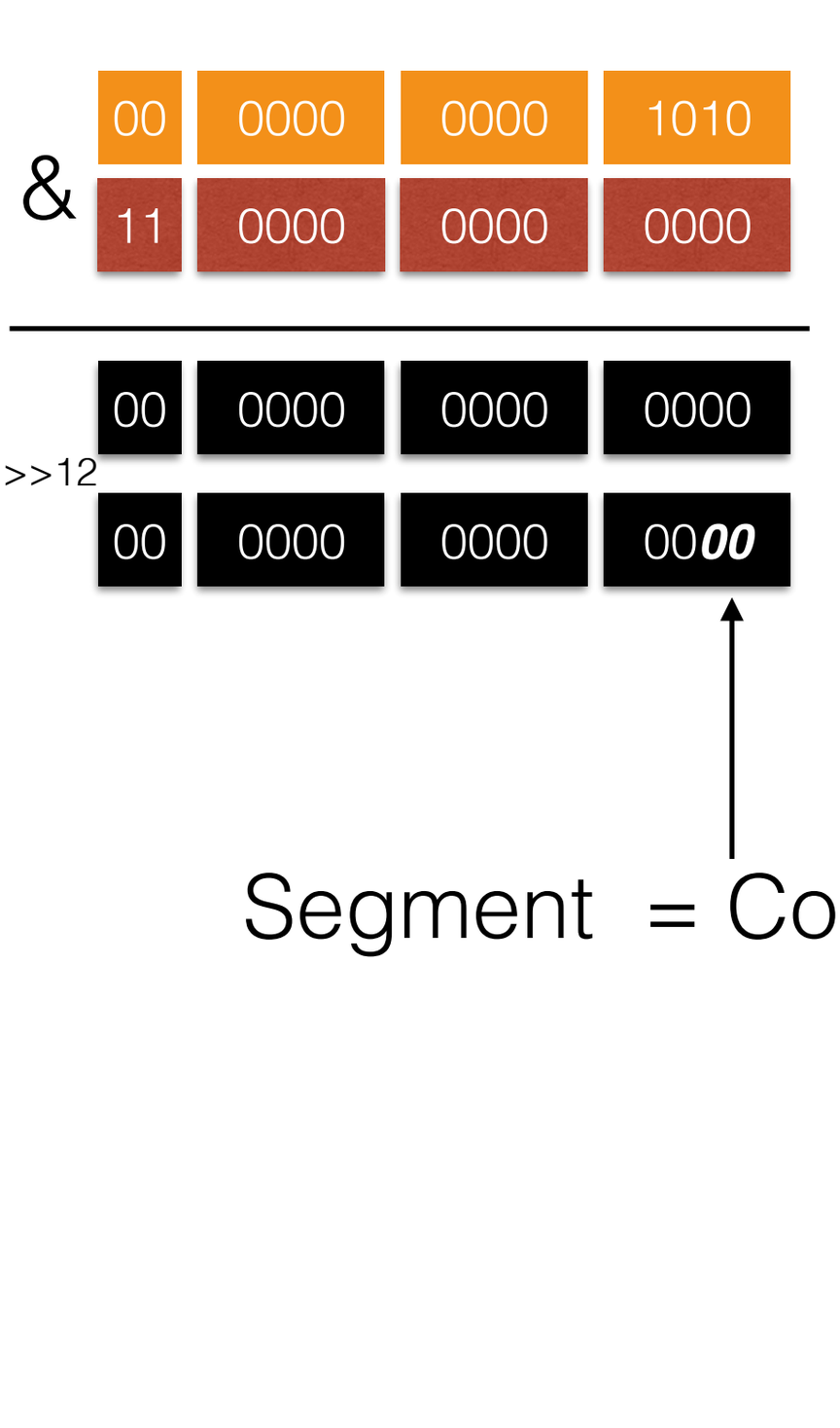
01 0111 1111 1111

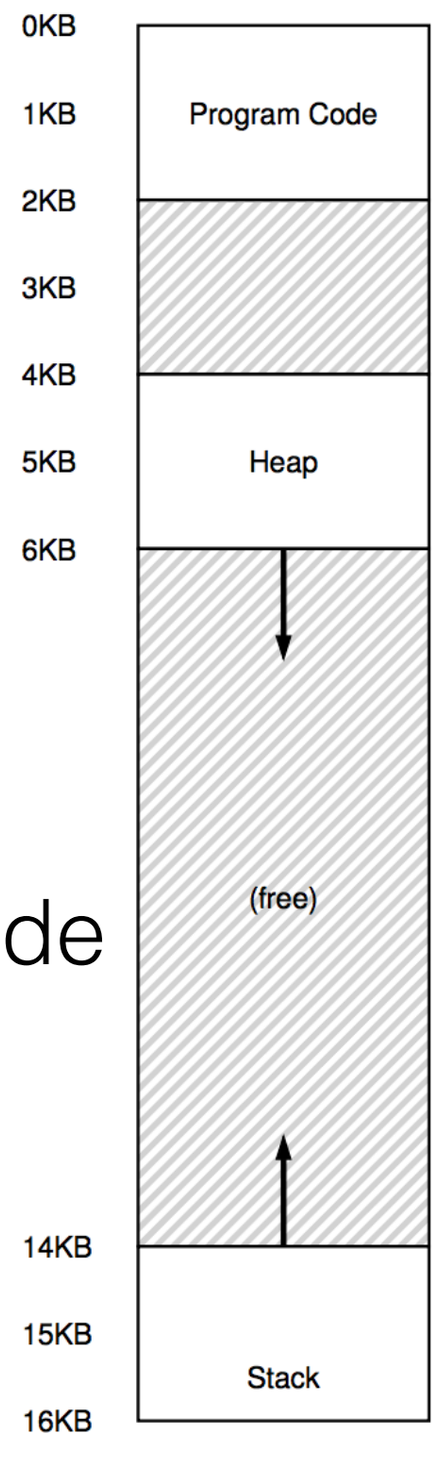
Segment = (VA & 11 0000 0000 0000) >>12

11 1000 0000 0000

11 1111 1111 1111

# Segment Reference




00 0000 0000 0000

00 0111 1111 1111

01 0000 0000 0000

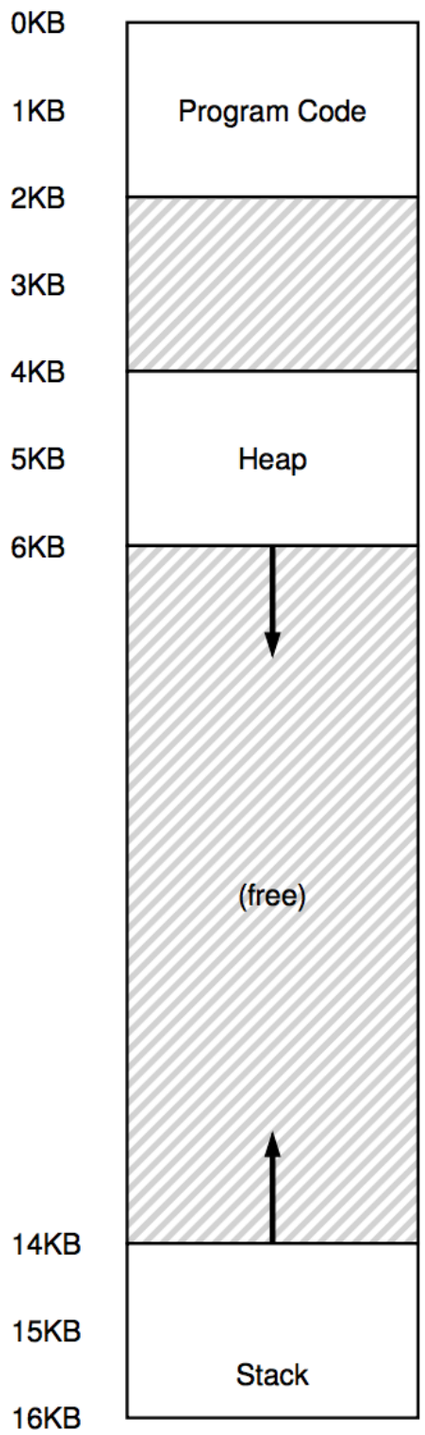
01 0111 1111 1111

Segment = (VA & 11 0000 0000 0000) >>12

11 1000 0000 0000

11 1111 1111 1111

# Segment Reference



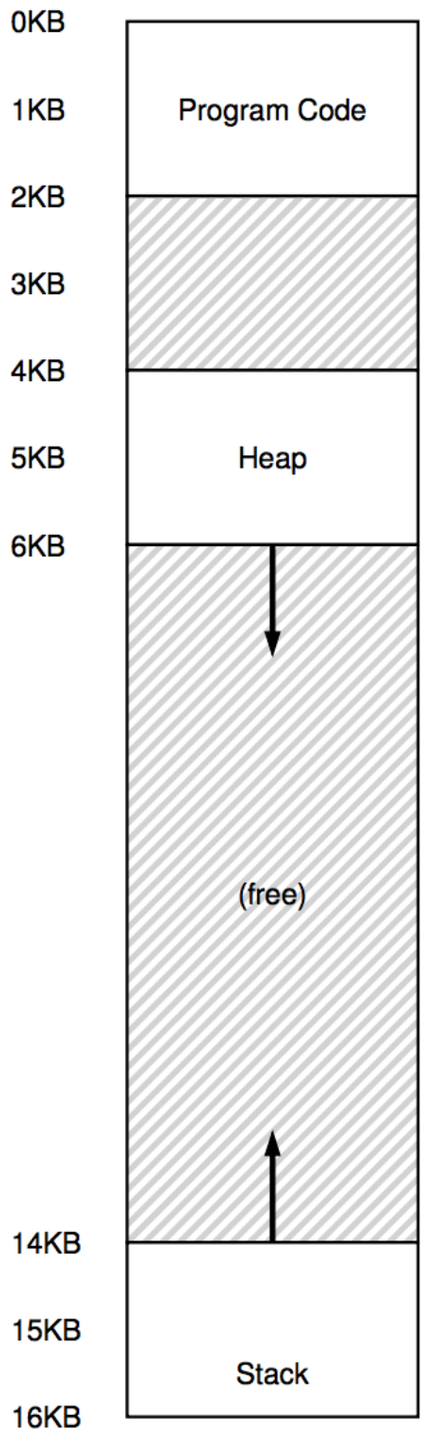

| | | | |
|---|---|---|---|
| 00 | 0000 | 0000 | 0000 |
| 00 | 0111 | 1111 | 1111 |
| 01 | 0000 | 0000 | 0000 |
| 01 | 0111 | 1111 | 1111 |

Segment = (VA & 11 0000 0000 0000) >>12

| | | | |
|---|---|---|---|
| 11 | 1000 | 0000 | 0000 |
| 11 | 1111 | 1111 | 1111 |

# Segment Reference

01 | 0000 | 1010 | 1010




00 | 0000 | 0000 | 0000

00 | 0111 | 1111 | 1111

01 | 0000 | 0000 | 0000

01 | 0111 | 1111 | 1111

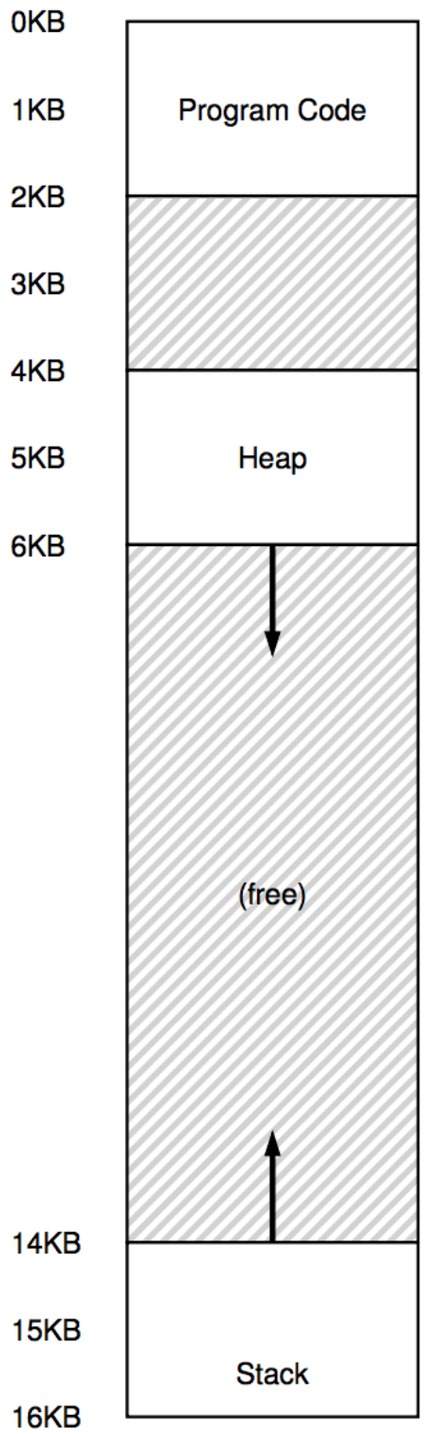
Segment = (VA & 11 0000 0000 0000) >>12

11 | 1000 | 0000 | 0000

11 | 1111 | 1111 | 1111

# Segment Reference

| | | | |
|---|---|---|---|
| 01 | 0000 | 1010 | 1010 |
| 11 | 0000 | 0000 | 0000 |

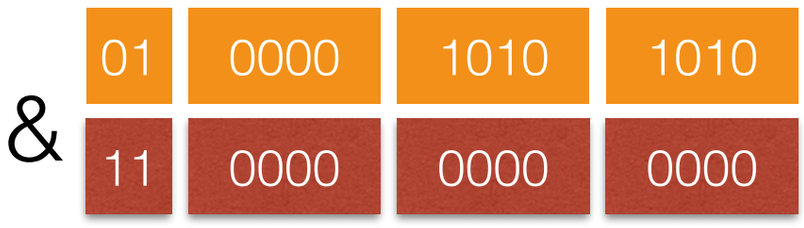


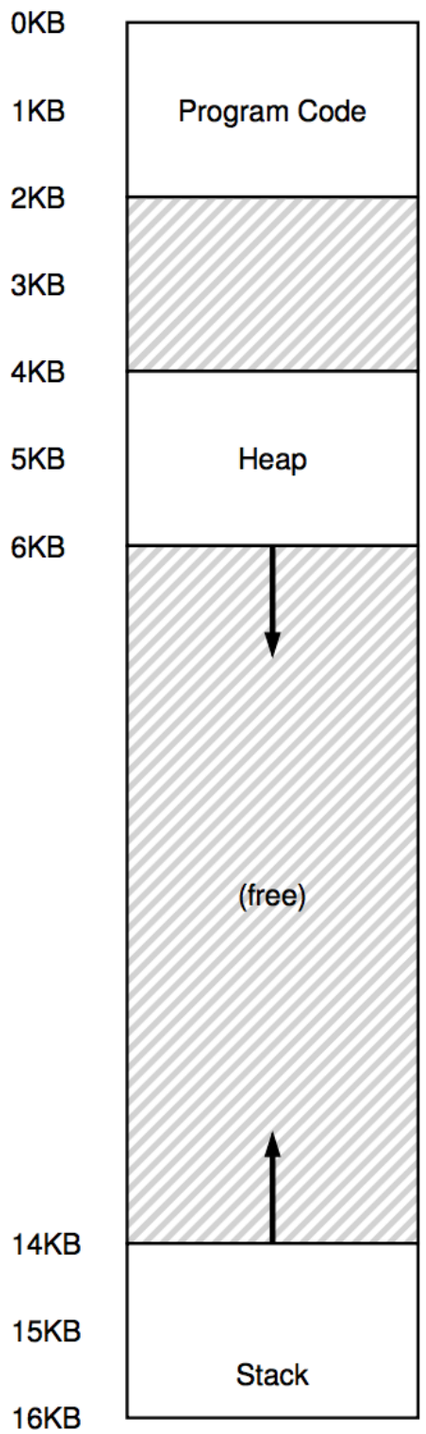

| | | | |
|---|---|---|---|
| 00 | 0000 | 0000 | 0000 |
| 00 | 0111 | 1111 | 1111 |
| 01 | 0000 | 0000 | 0000 |
| 01 | 0111 | 1111 | 1111 |

Segment = (VA & 11 0000 0000 0000) >>12

| | | | |
|---|---|---|---|
| 11 | 1000 | 0000 | 0000 |
| 11 | 1111 | 1111 | 1111 |

# Segment Reference




| | | | |
|---|---|---|---|
| 00 | 0000 | 0000 | 0000 |
| 00 | 0111 | 1111 | 1111 |
| 01 | 0000 | 0000 | 0000 |
| 01 | 0111 | 1111 | 1111 |

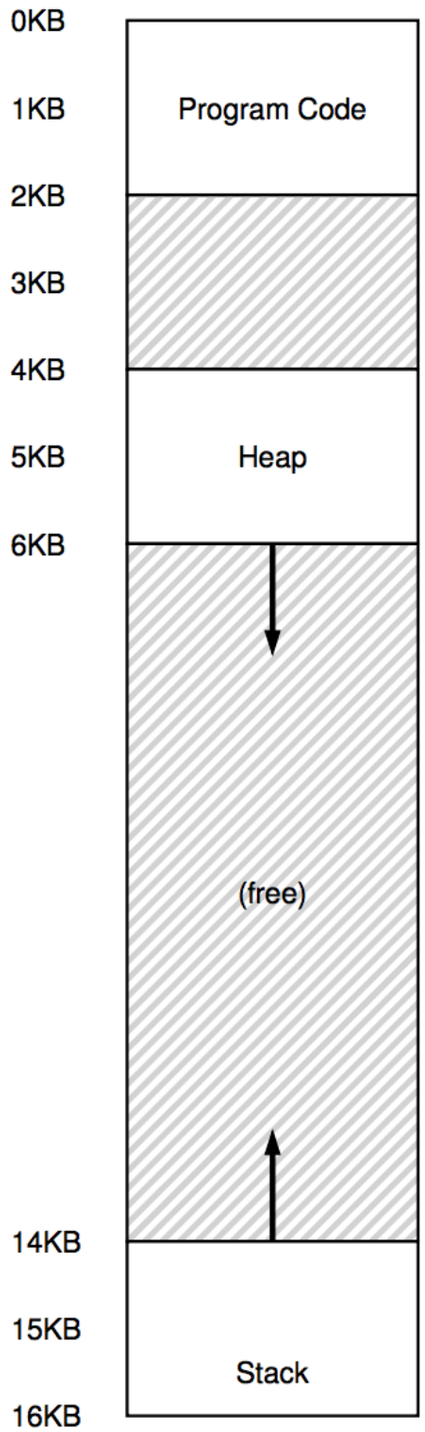
Segment = (VA & 11 0000 0000 0000) >>12

| | | | |
|---|---|---|---|
| 11 | 1000 | 0000 | 0000 |
| 11 | 1111 | 1111 | 1111 |

# Segment Reference

01 0000 1010 1010

& 11 0000 0000 0000




00 0000 0000 0000

00 0111 1111 1111

01 0000 0000 0000

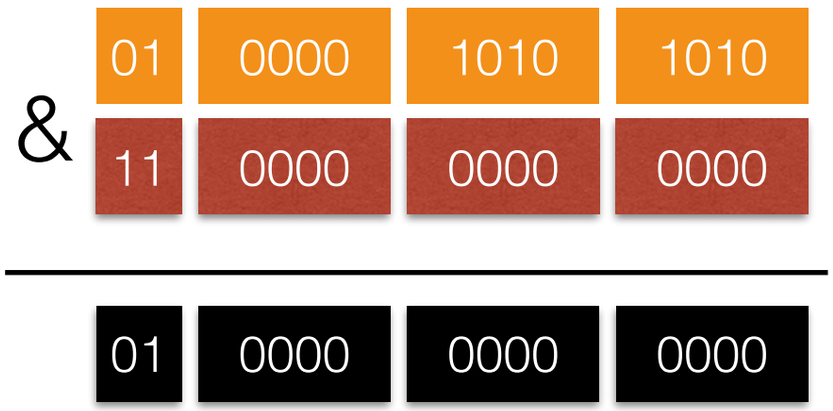
01 0111 1111 1111

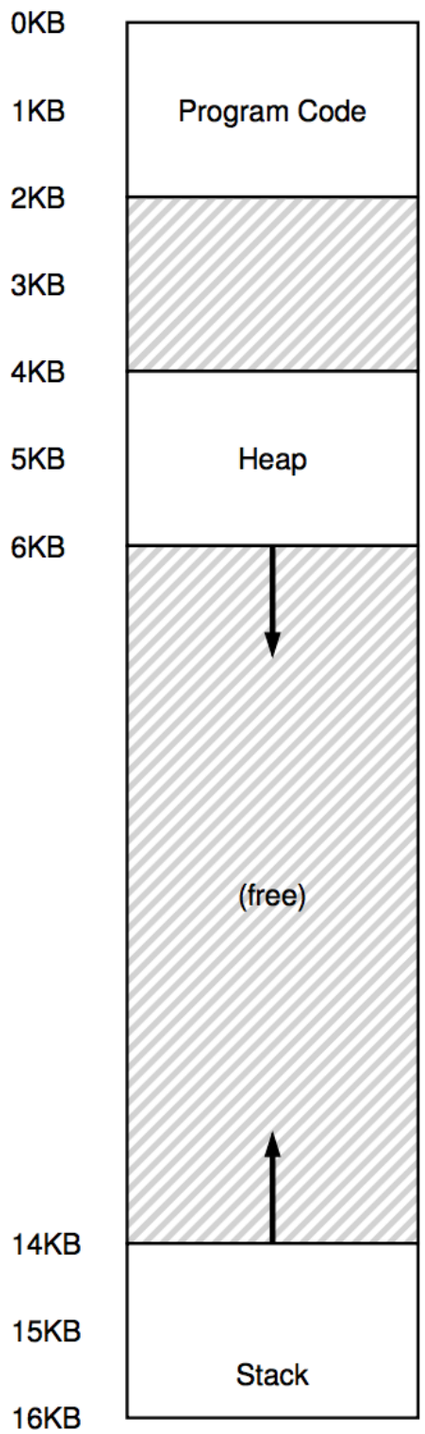
Segment = (VA & 11 0000 0000 0000) >>12

11 1000 0000 0000

11 1111 1111 1111

# Segment Reference




00 0000 0000 0000

00 0111 1111 1111

01 0000 0000 0000

01 0111 1111 1111

Segment = (VA & 11 0000 0000 0000) >>12

11 1000 0000 0000

11 1111 1111 1111

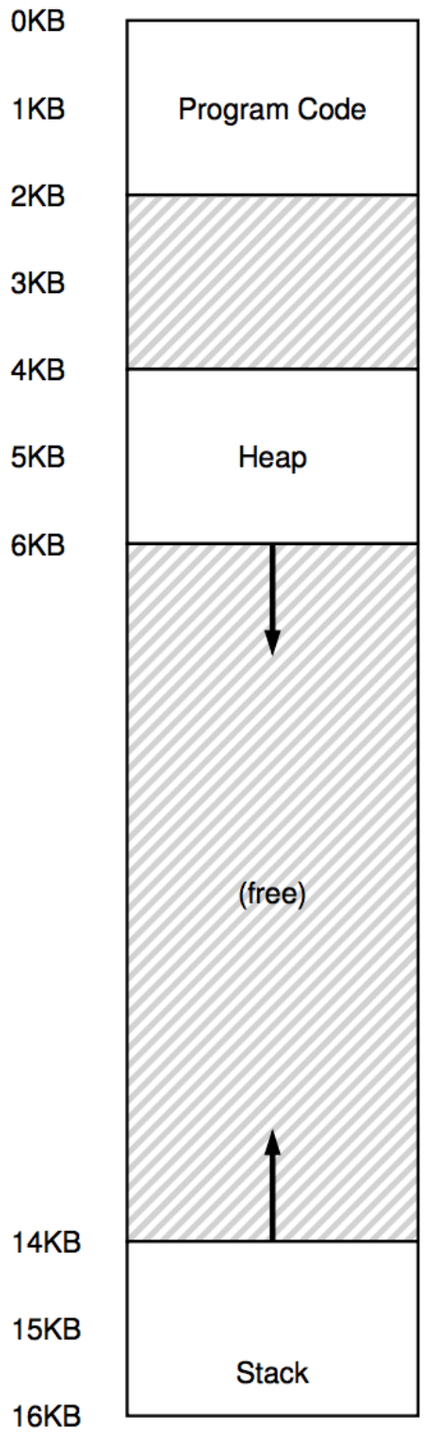
# Segment Reference

01 0000 1010 1010

& 11 0000 0000 0000

01 0000 0000 0000

>>12




00 0000 0000 0000

00 0111 1111 1111

01 0000 0000 0000

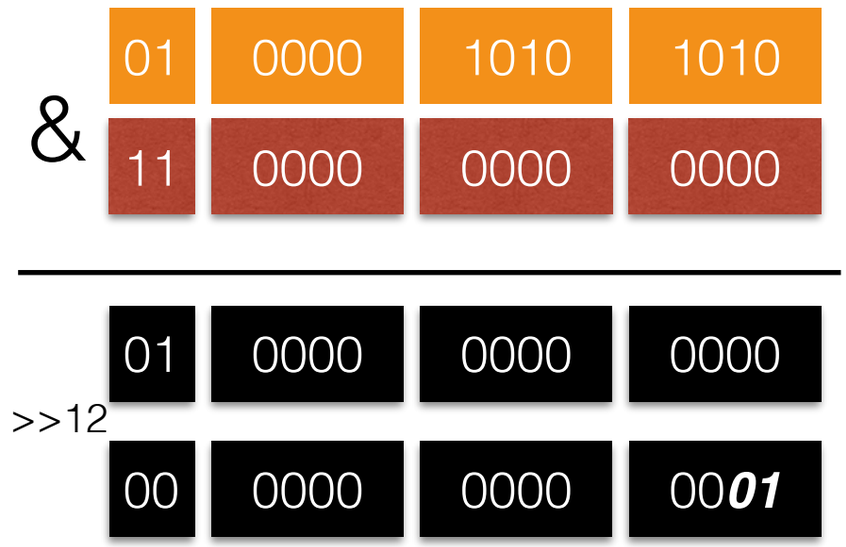
01 0111 1111 1111

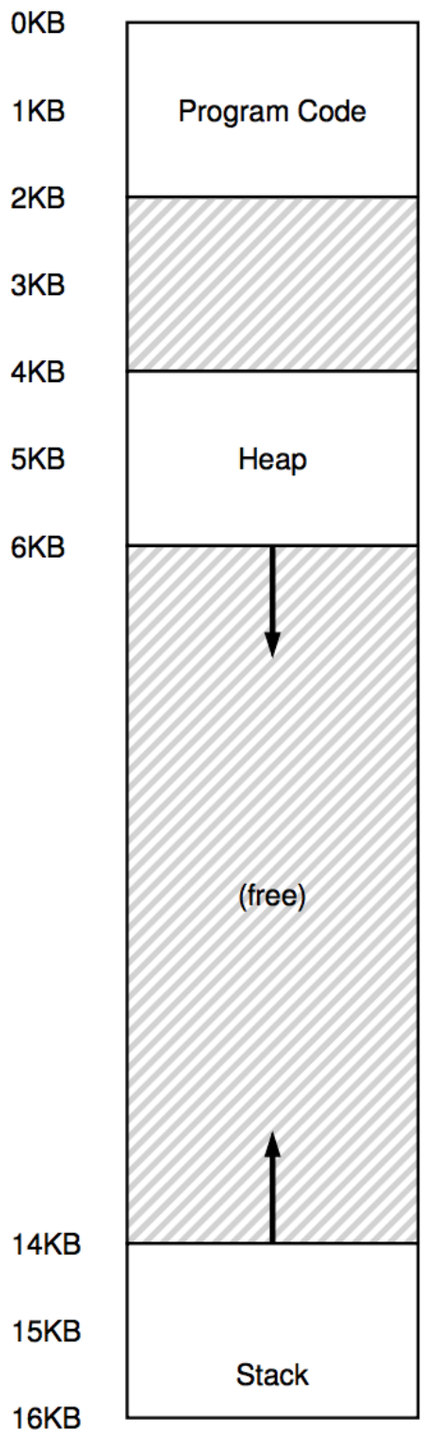
Segment = (VA & 11 0000 0000 0000) >>12

11 1000 0000 0000

11 1111 1111 1111

# Segment Reference




00 0000 0000 0000

00 0111 1111 1111

01 0000 0000 0000

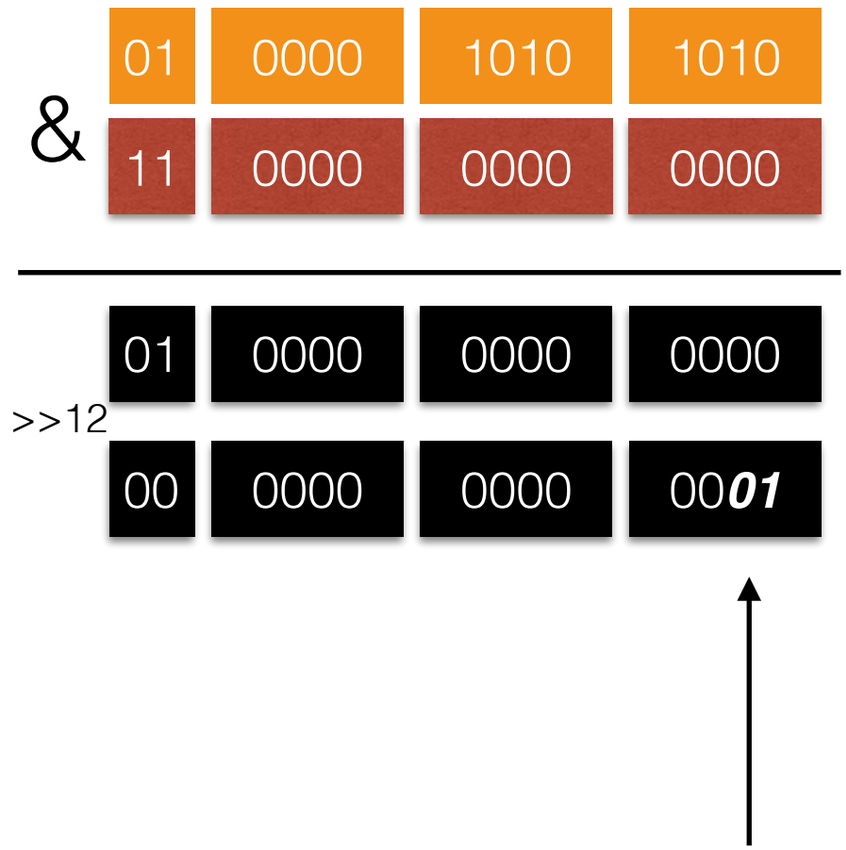
01 0111 1111 1111

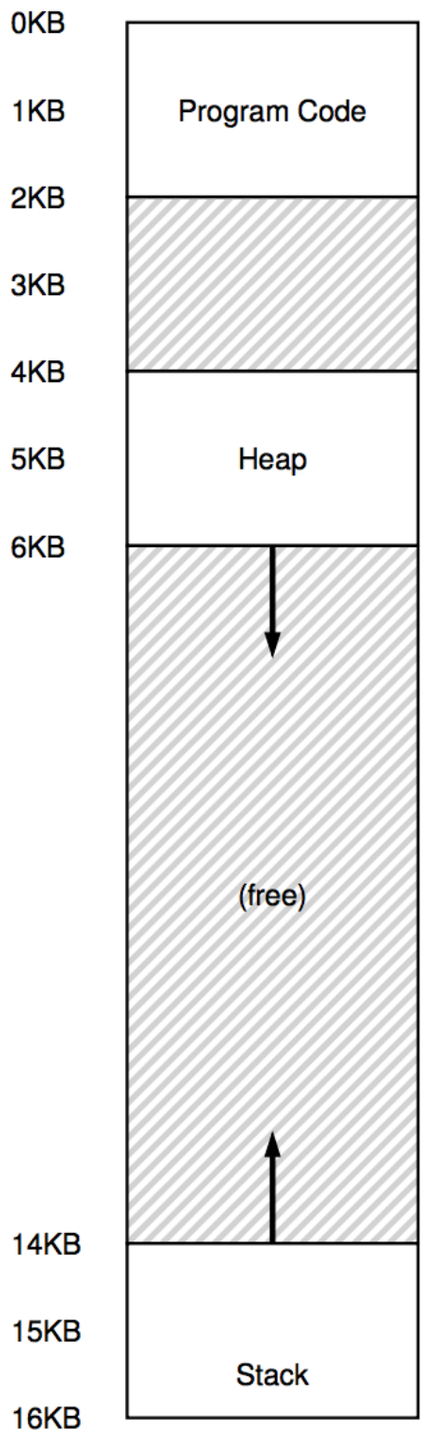
Segment = (VA & 11 0000 0000 0000) >>12

11 1000 0000 0000

11 1111 1111 1111

# Segment Reference




00 0000 0000 0000

00 0111 1111 1111

01 0000 0000 0000

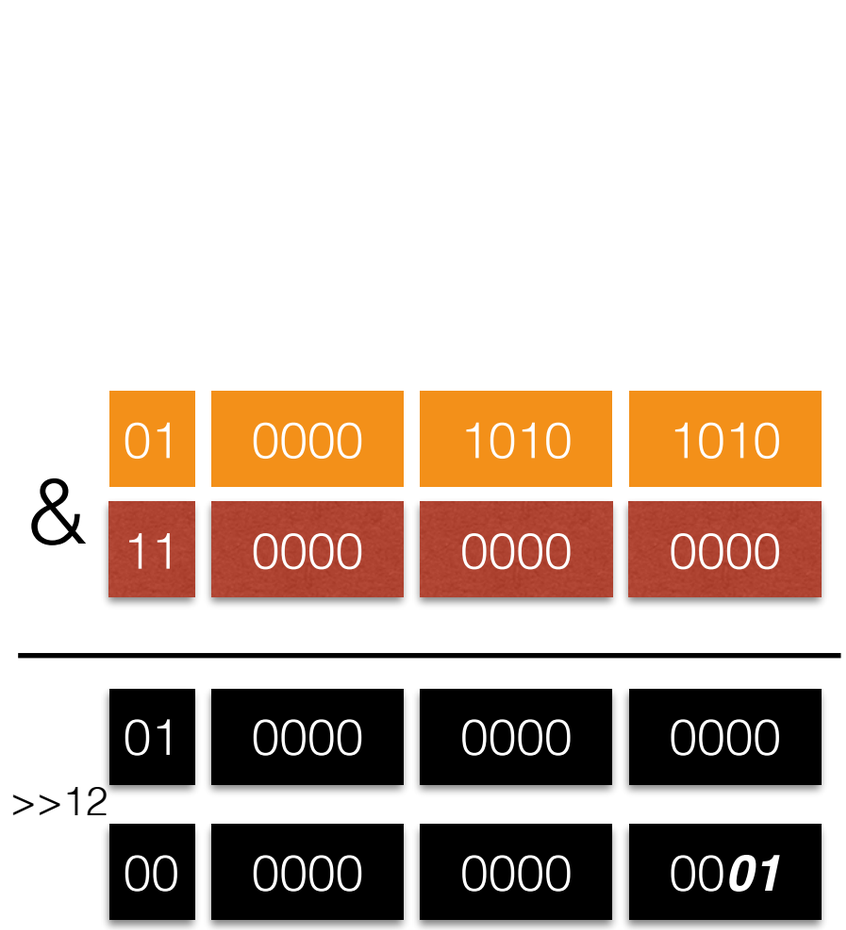
01 0111 1111 1111

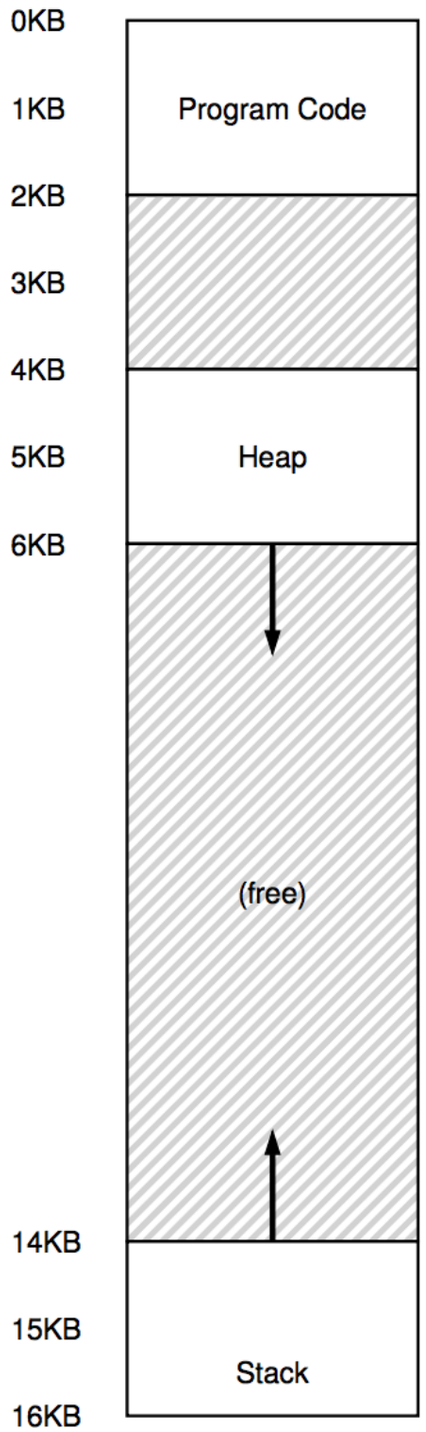
Segment = (VA & 11 0000 0000 0000) >>12

11 1000 0000 0000

11 1111 1111 1111

# Segment Reference




00 0000 0000 0000

00 0111 1111 1111

01 0000 0000 0000

01 0111 1111 1111

Segment = (VA & 11 0000 0000 0000) >>12

11 1000 0000 0000

11 1111 1111 1111