

Operating Systems

Memory Virtualisation

Lecture Overview

- Address Space of a Process
- Memory Virtualisation
 - Time sharing, static relocation, dynamic relocation
 - Base, Base & Bounds, Segmentation, Page Tables
- Hardware Considerations

Last Week

Scheduling

- FCFS, SJF, MLFQ, RR, Lottery Scheduling, CFS

Address Space and Translation

Where?

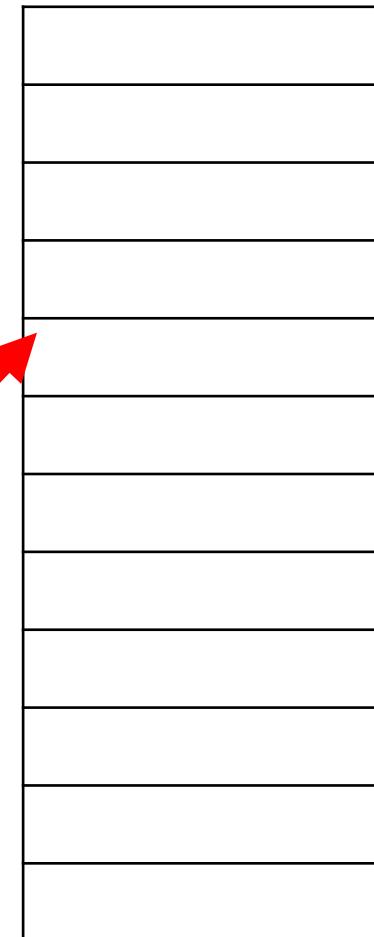
Basic Idea

Address

x0000FFFF

 bits

Memory



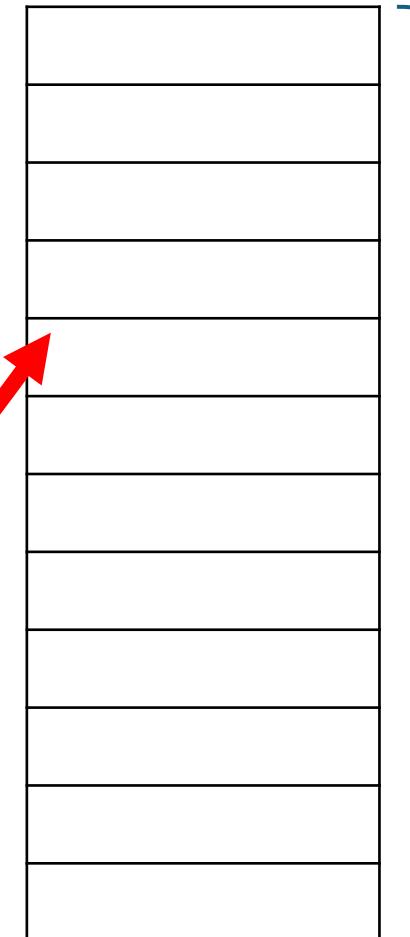
Basic Idea

Address

x0000FFFF

k bits

Memory



2^k spots
(bytes)

Basic Idea

Address

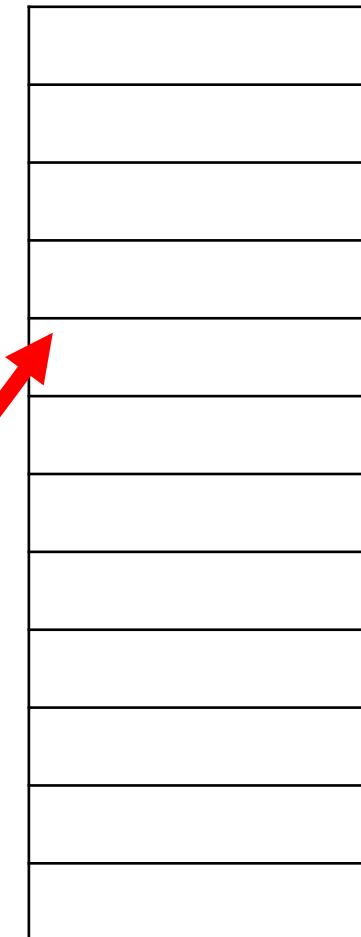
x0000FFFF

k bits

$$2^{12} = 4\text{KB}$$

$$2^{10} = 1024$$

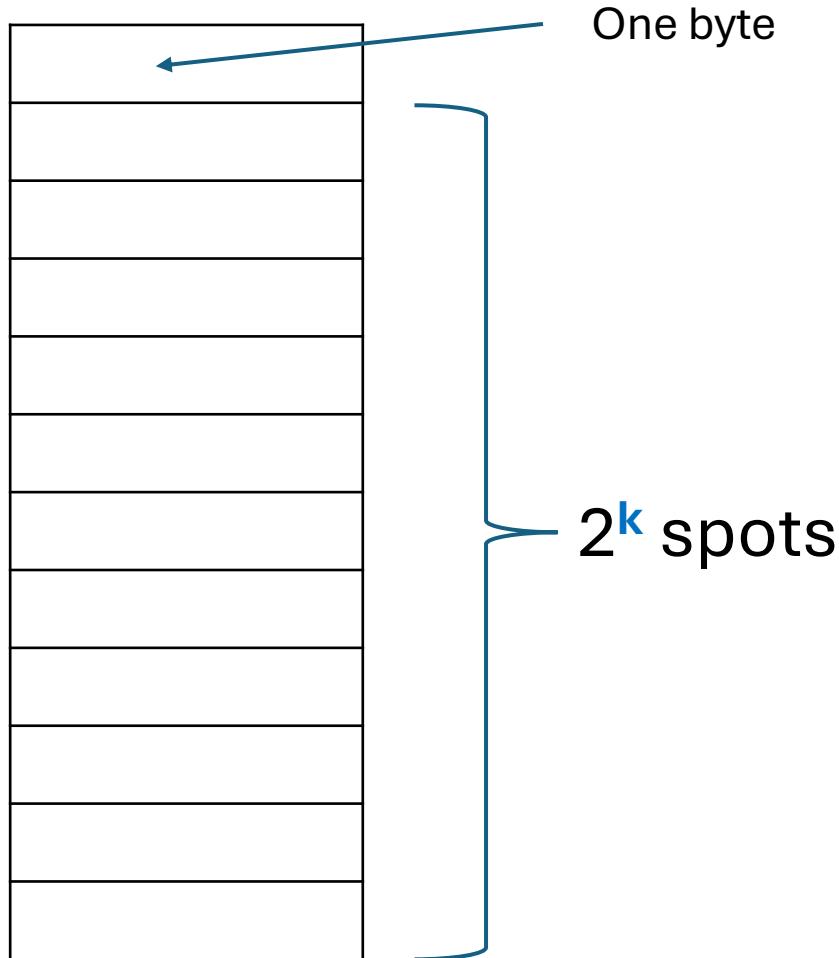
Memory



2^k spots
(bytes)

4 MB

Memory Thinking

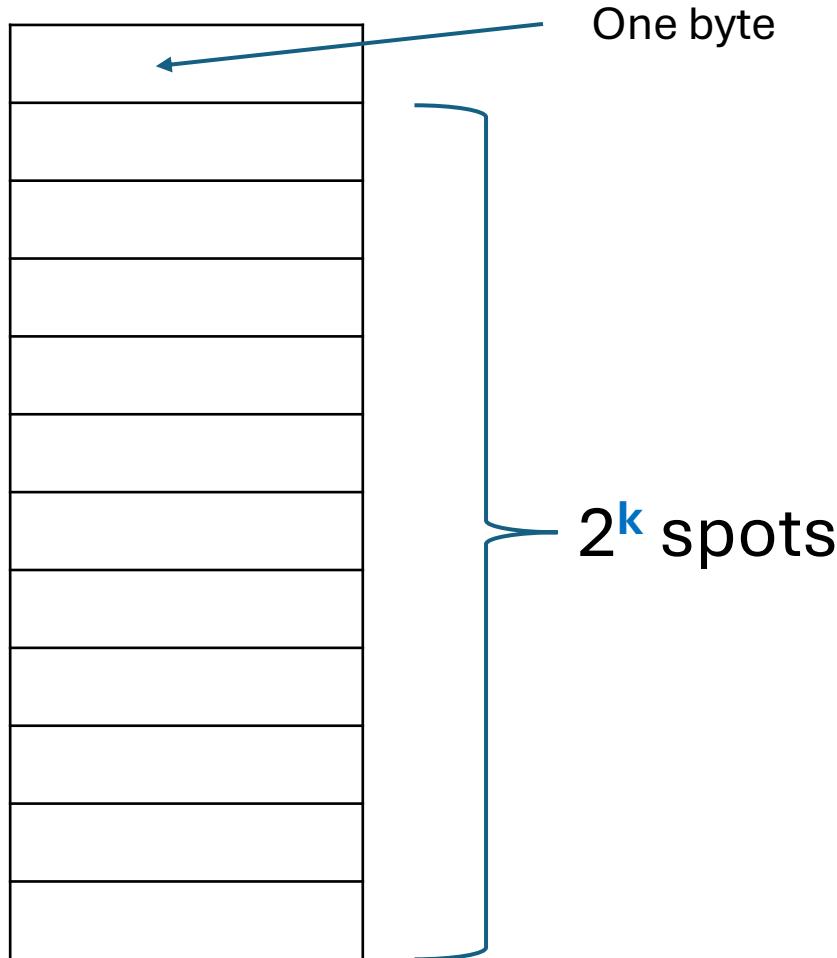


$32 \text{ bit} \Rightarrow 2^{32} \text{ bytes} \Rightarrow 2^2 * 2^{10} * 2^{10} * 2^{10}$

1024

~4 billion **bytes** +
rounding

Memory Thinking



64 bit => 2^{64} bytes => $2^4 * 2^{10} * 2^{10} * 2^{10} * \dots$

~16 billion, billion **bytes** +
rounding

Motivation for Virtualisation

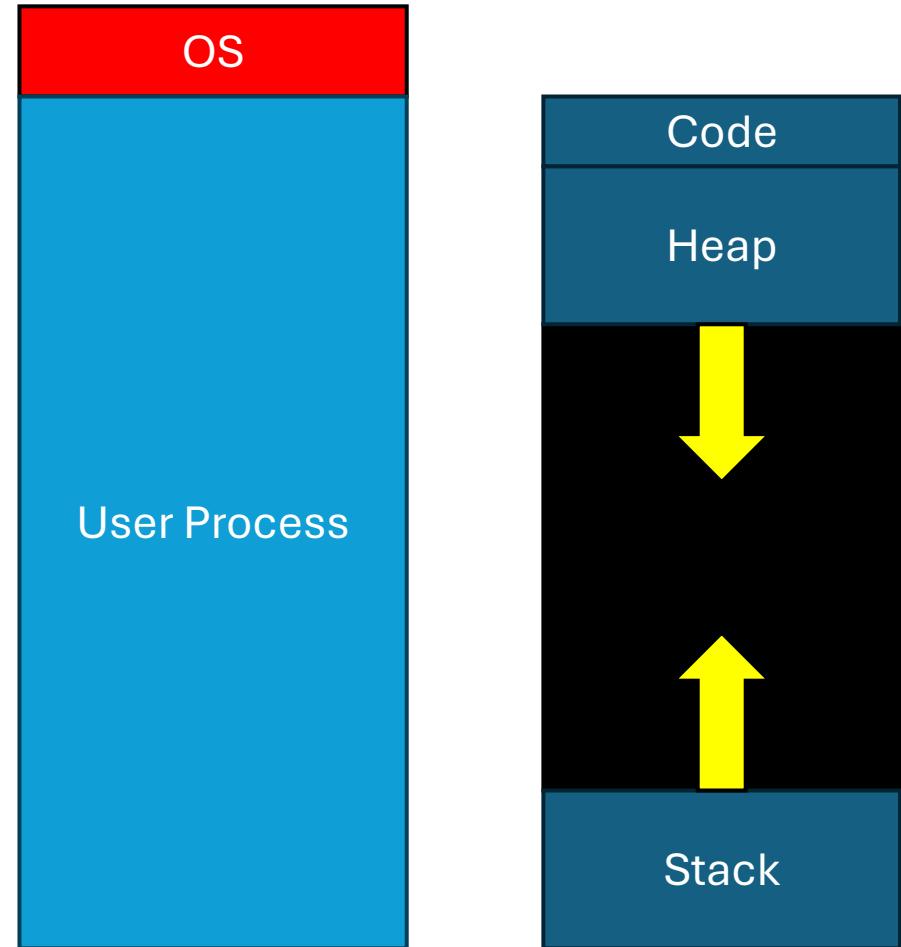
Uniprogramming: One at a time

Advantages:

- Easy!

Disadvantages:

- Process can destroy OS
- Only one process at a time...



Motivation for Virtualisation

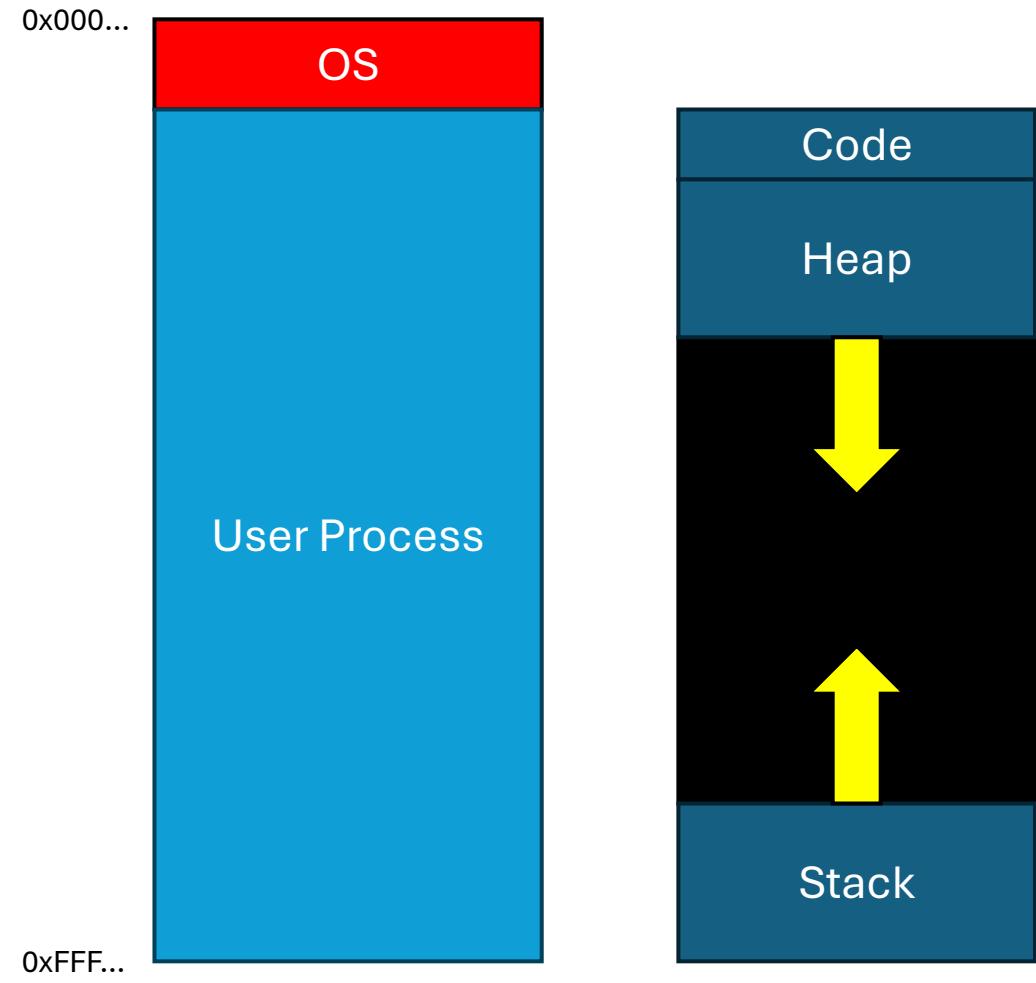
Uniprogramming: One at a time

Advantages:

- Easy!

Disadvantages:

- Process can destroy OS
- Only one process at a time...



Stacks usually count backwards...?

Stacks and Heaps

Some Revision

Process Memory Revision

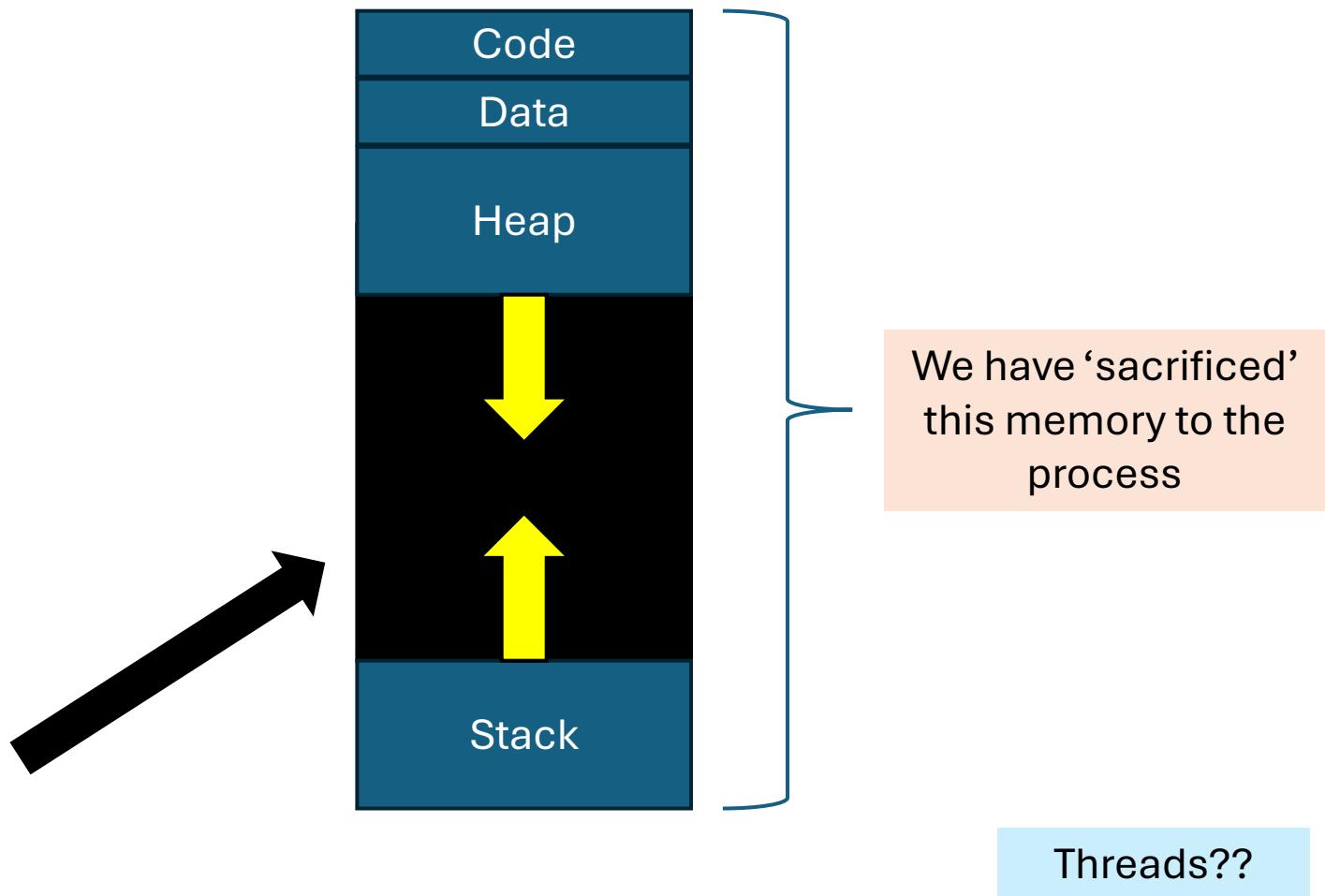
Why **Heap** must grow?

- We don't know how much memory we will need?
- Linked lists, trees, etc...

Why **Stack** must grow?

- Mostly **recursion**...

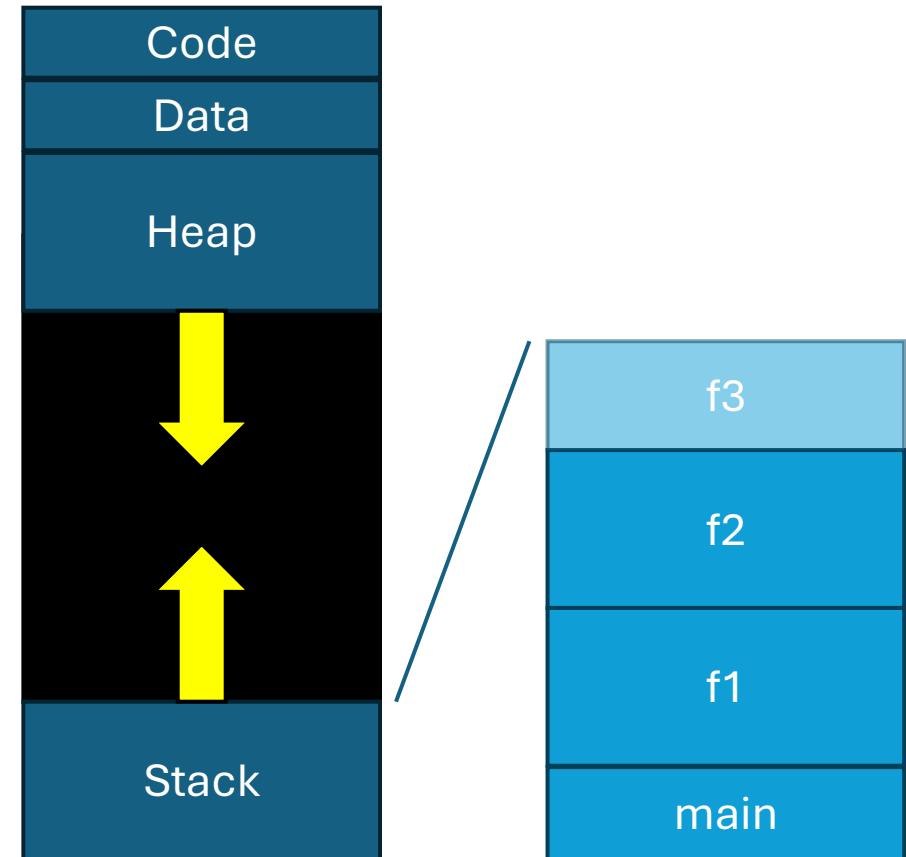
Growing like this makes it about as easy as it will get...



Process Memory Revision: Stack

Stacks...

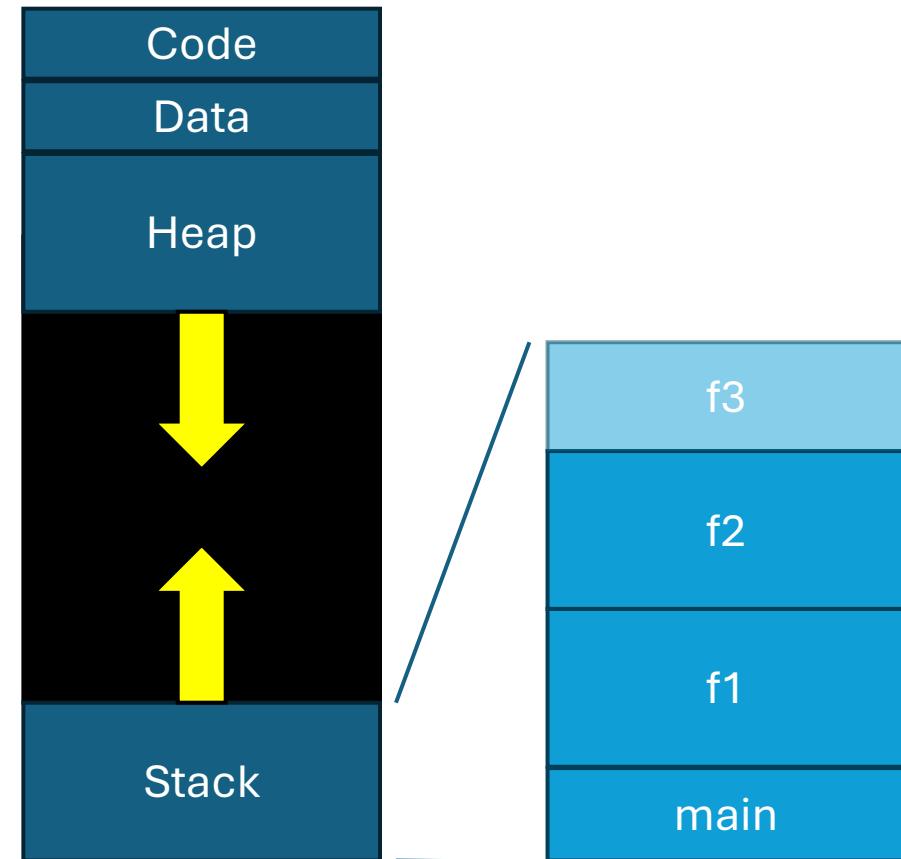
- Are easy to manage
 - Stack Pointer
 - Push/Pop mechanic
- Each frame can have size determined
- Each frame aligns perfectly with the next
 - => No fragmentation



Process Memory Revision: Stack

...

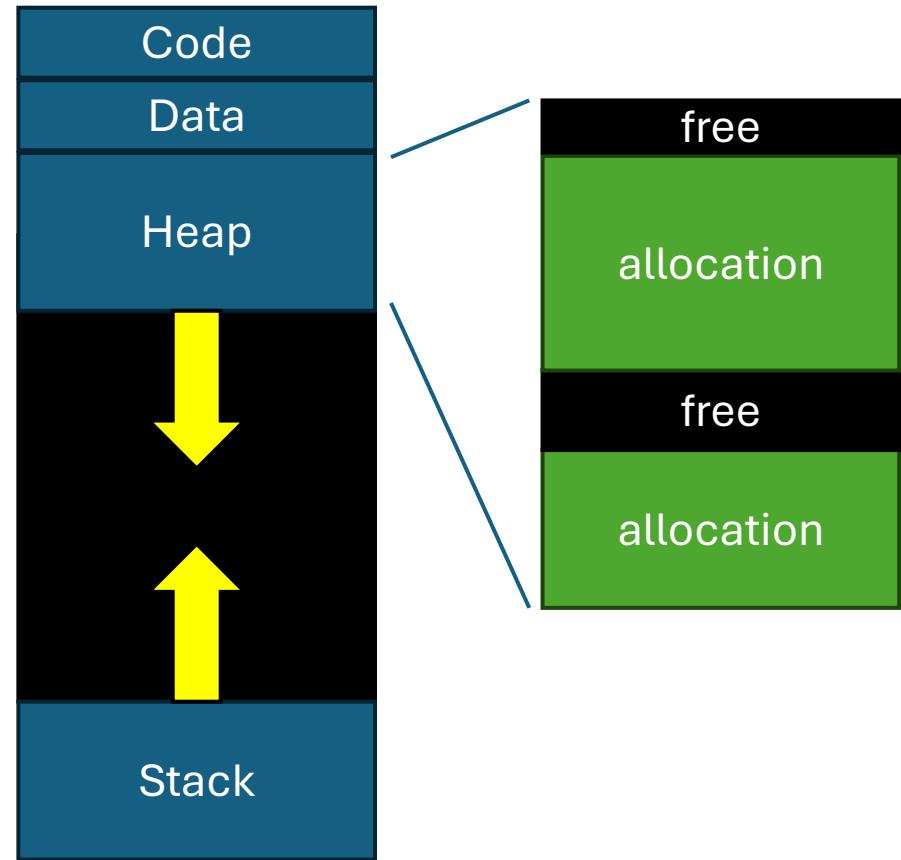
```
f2 () {  
    f3 ();  
}  
  
f1 () {  
    f2 ();  
}  
  
main () {  
    f1 ();  
}
```



Heap

Heaps:

- Are harder to manage
 - Free/allocations
- Structure is more haphazard
 - => fragmentation



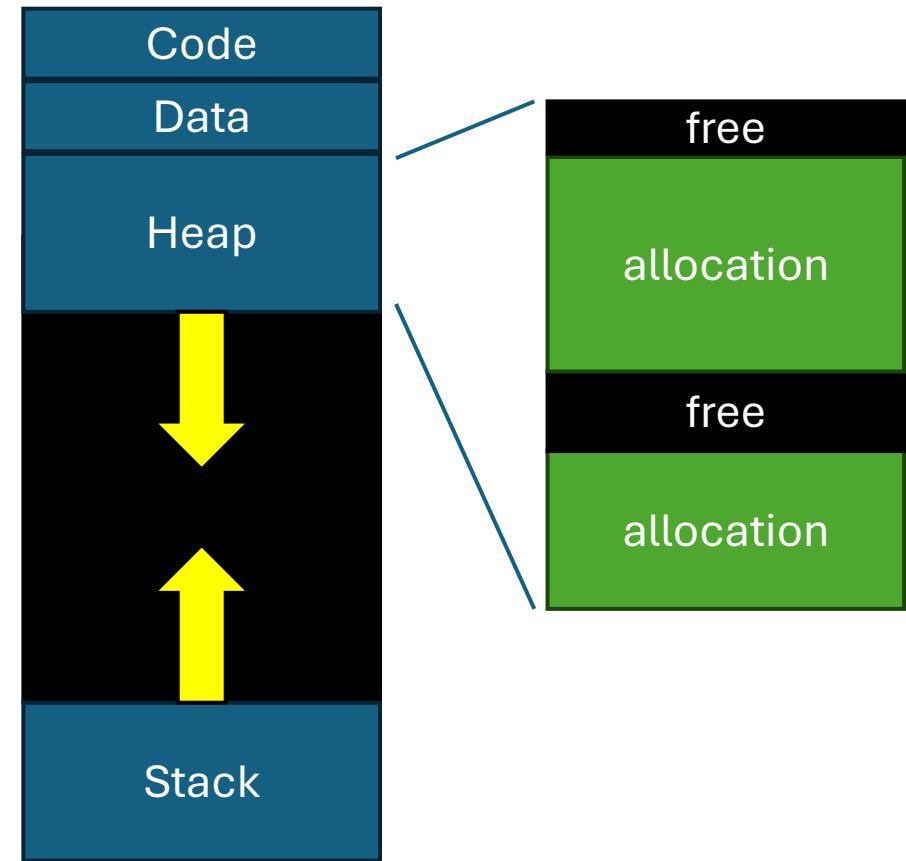
Heap

C

```
int *p = (int*) malloc(sizeof(int));  
  
free(p);
```

C++

```
int *p = new int;  
  
delete p;
```



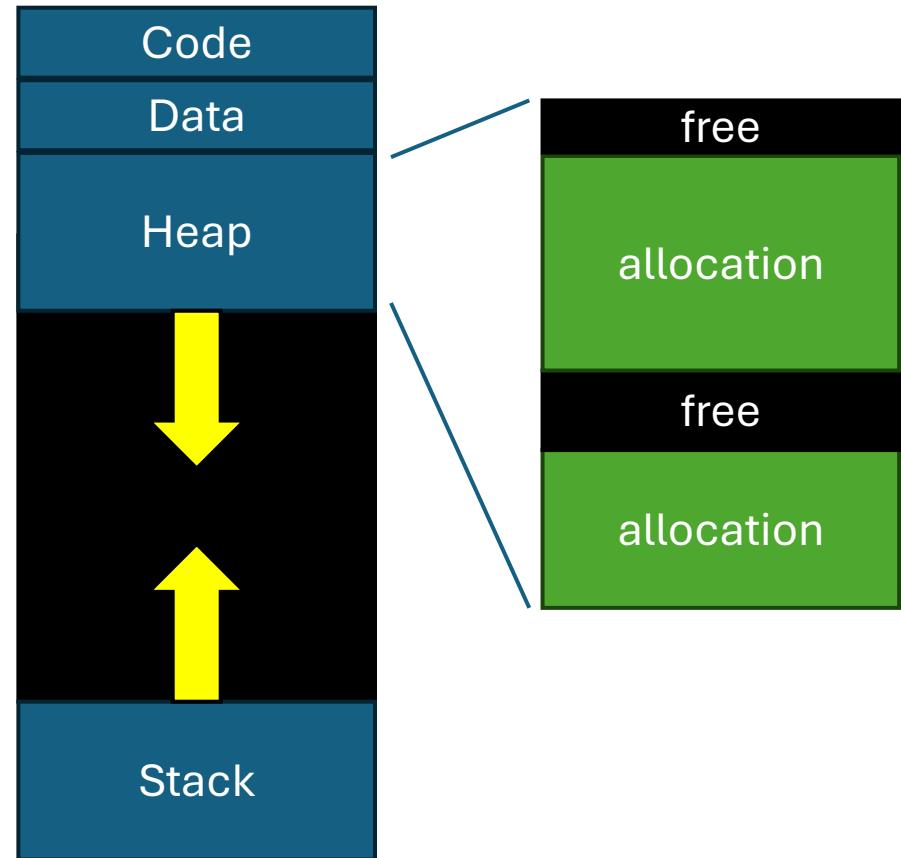
Heap

Advantage

- Works for all data structures
- Size – (sbrk, mmap)

Disadvantage:

- Speed
- Fragmentation
 - How do we find gaps?



A visual analogy

One is ordered..

One is chaotic...

And AI thought to label the stack
one with **heap labels**... not
confusing at all.



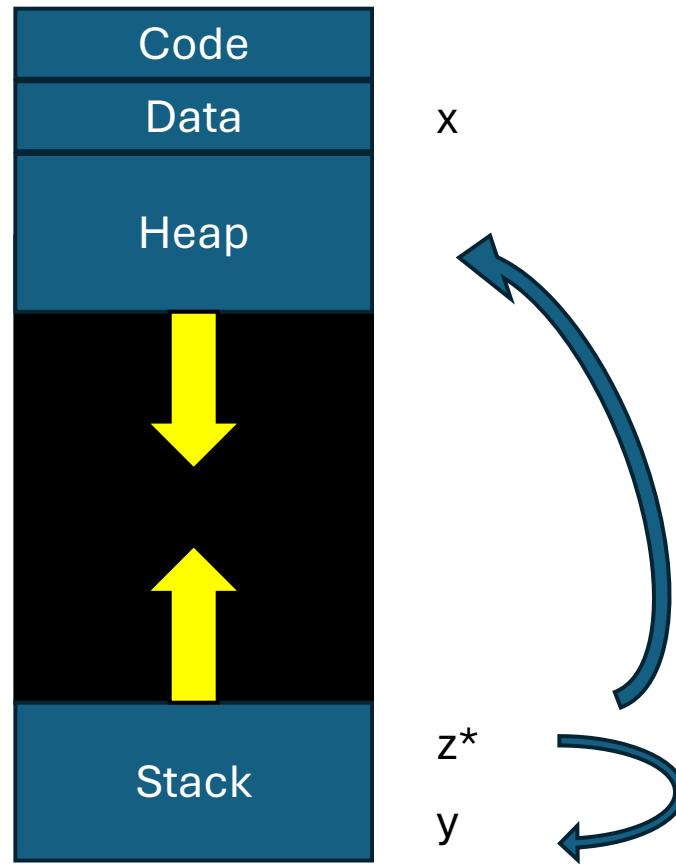
Heap on **left**, Stack on **right**
(NightCafeStudio)

Data

Data:

- Global Variables

```
int x = 5;  
int main()  
{  
    int y = 5;  
    int * z = &y;  
    z = (int*)malloc(sizeof(int));  
}
```



Virtualisation - Again

...

Motivation for Virtualisation

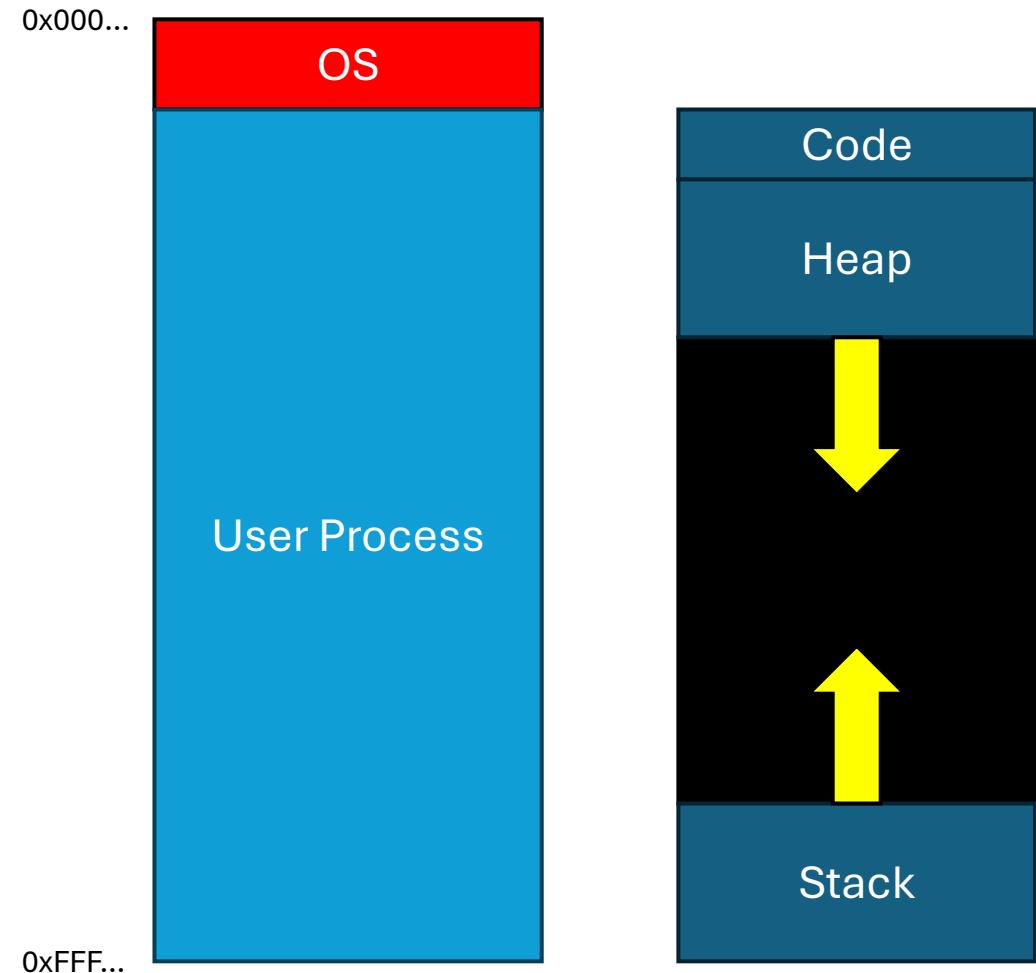
Multiprogramming: Many at a time

Advantages:

- Cool!

Disadvantages:

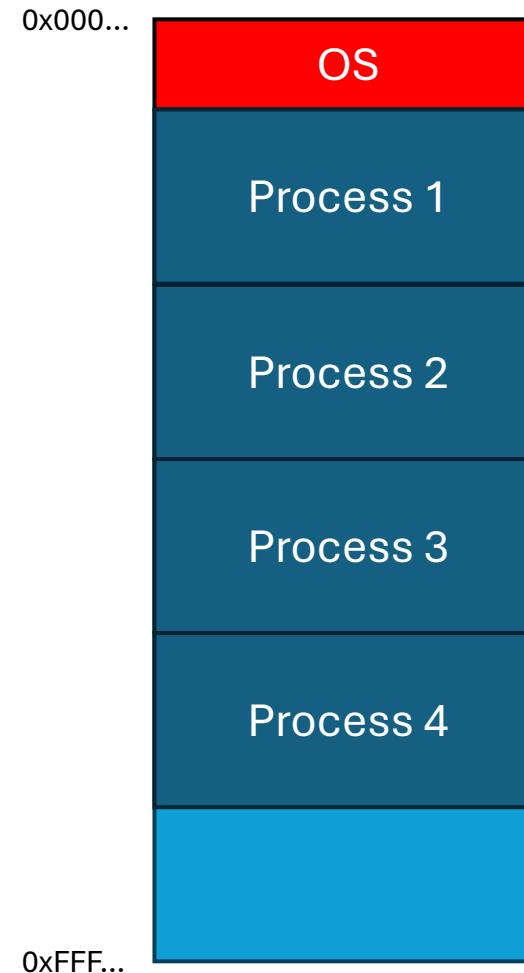
- How?
 - Collisions?
 - Missing data?



Virtualisation

Problem #1:

What is problem #1?

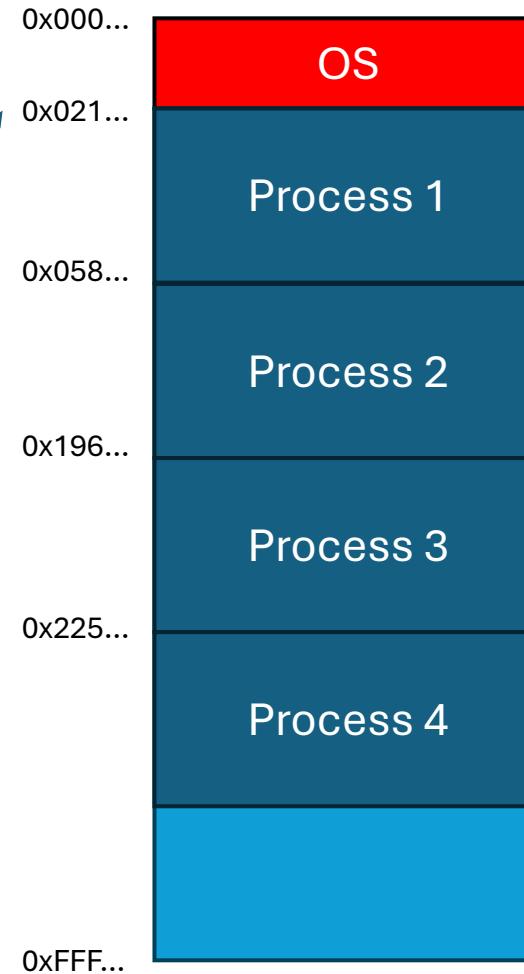


Virtualisation

Problem #1:

What is problem #1?

Where does the
program think it is
writing to?



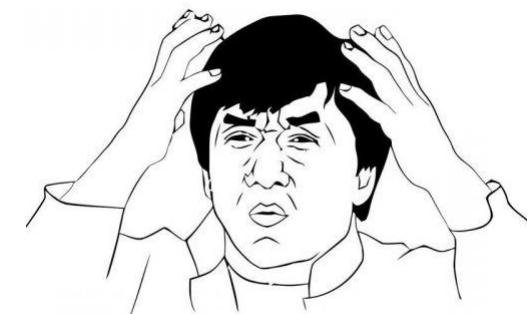
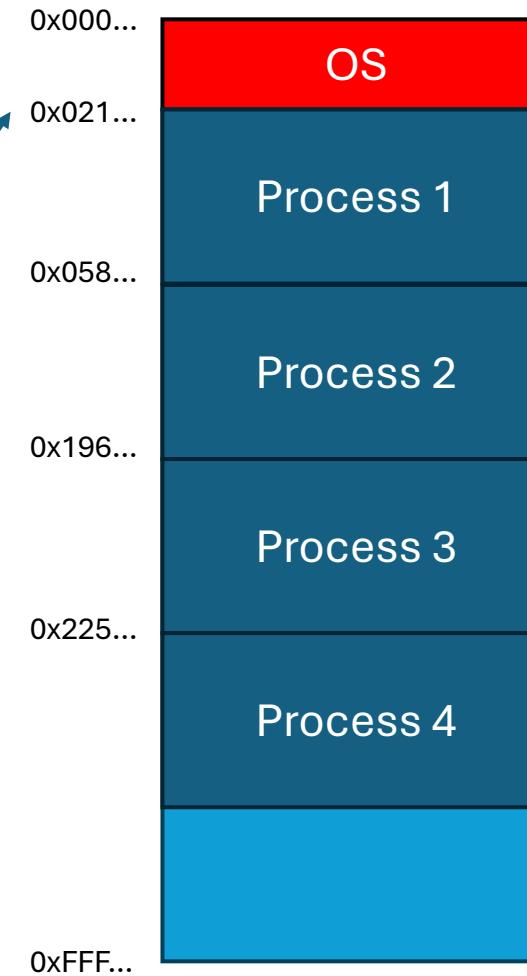
Virtualisation: Transparency

Problem #1:

What is problem #1?

Where does the
program think it is
writing to?

The process shouldn't know that it
is sharing memory...



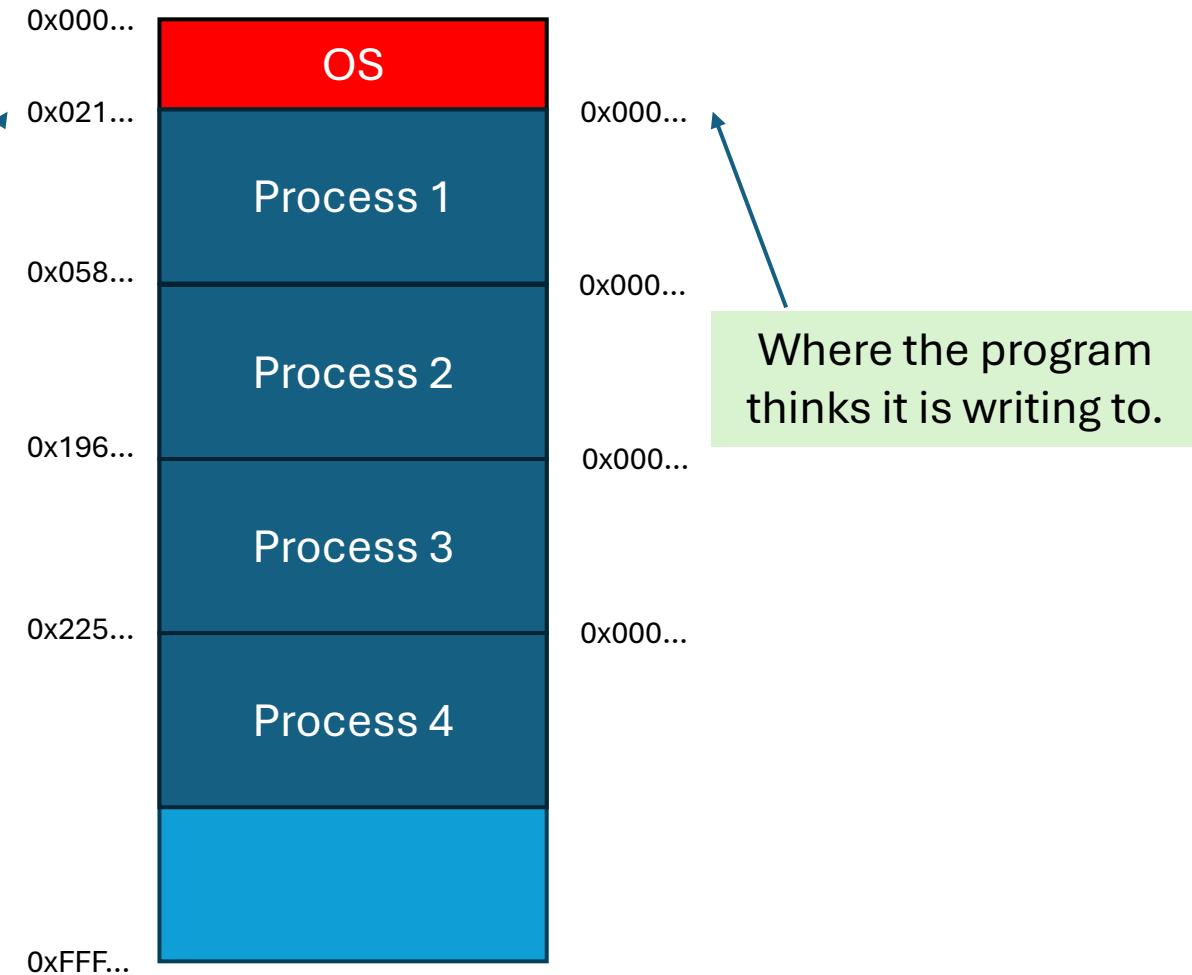
How is that
“transparent”??

Virtualisation: Transparency

Problem #1:

What is problem #1?

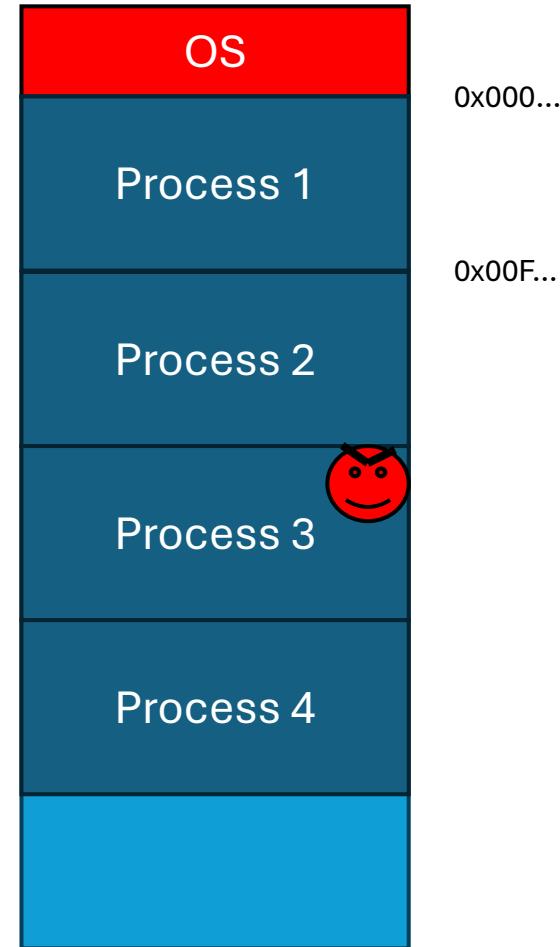
Where the program is writing to.



Virtualisation

Problem #2:

What is problem #2?

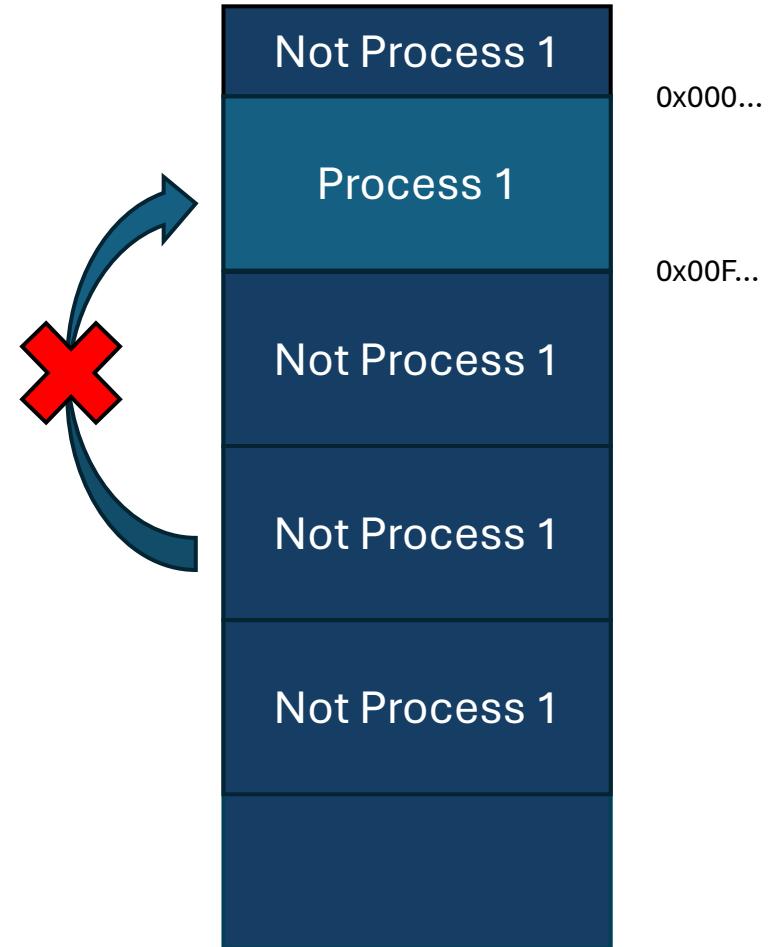


Virtualisation: Protection

Problem #2:

What is problem #2?

Processes are limited
to their own address
space

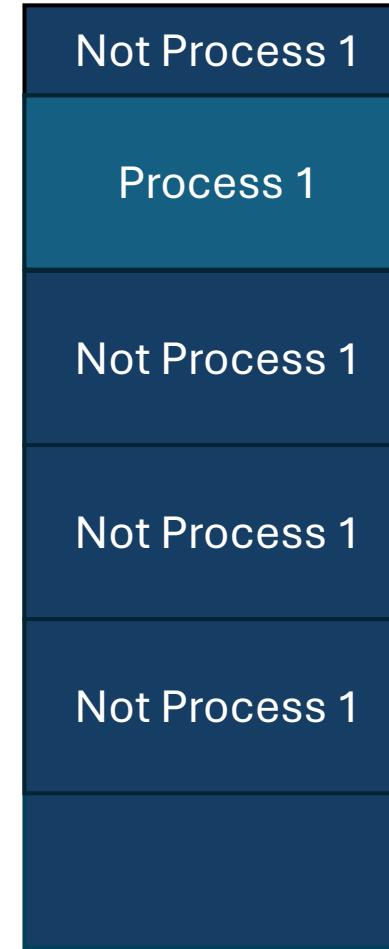
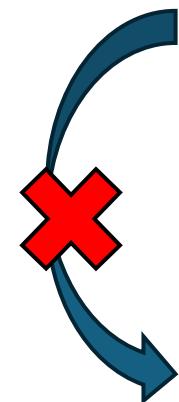


Virtualisation: Privacy

Problem #3:

What is problem #3?

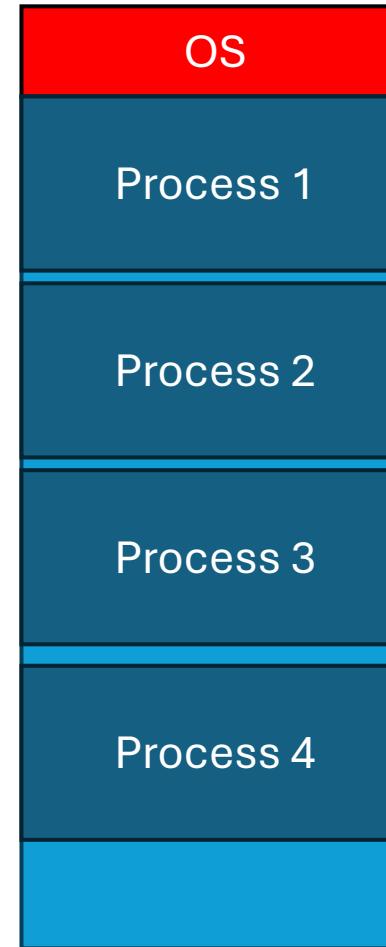
Processes are limited
to their own address
space



Virtualisation

Problem #4:

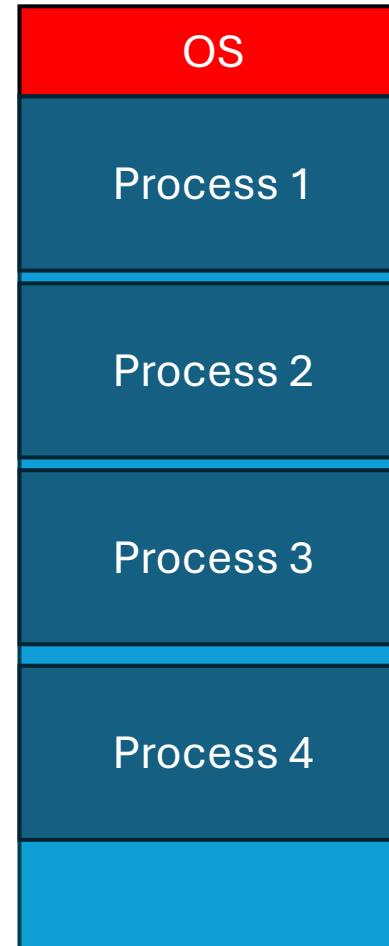
What is problem #4?



Virtualisation

Problem #4:

What is problem #4?

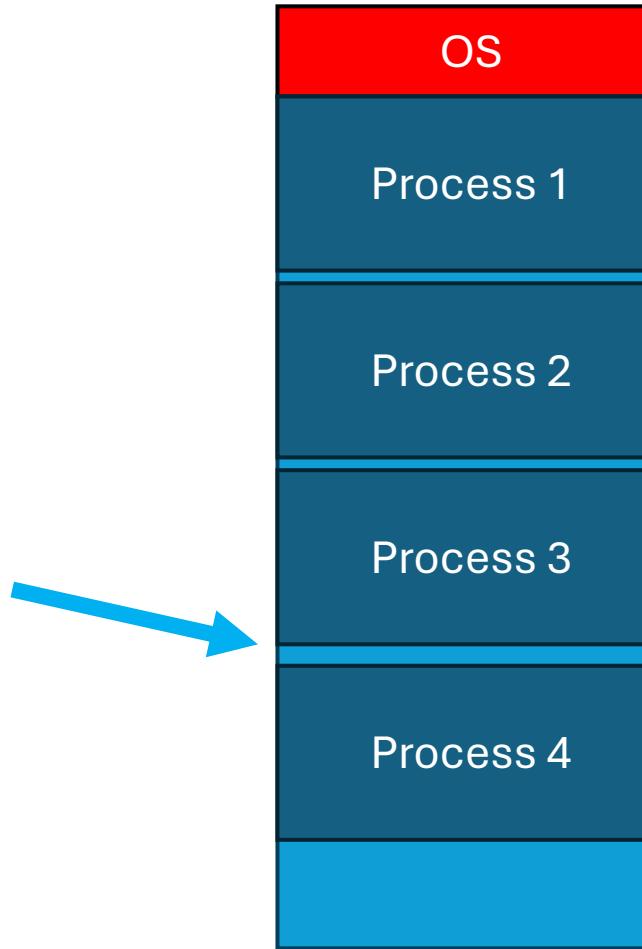


Virtualisation: Fragmentation

Problem #4:

What is problem #4?

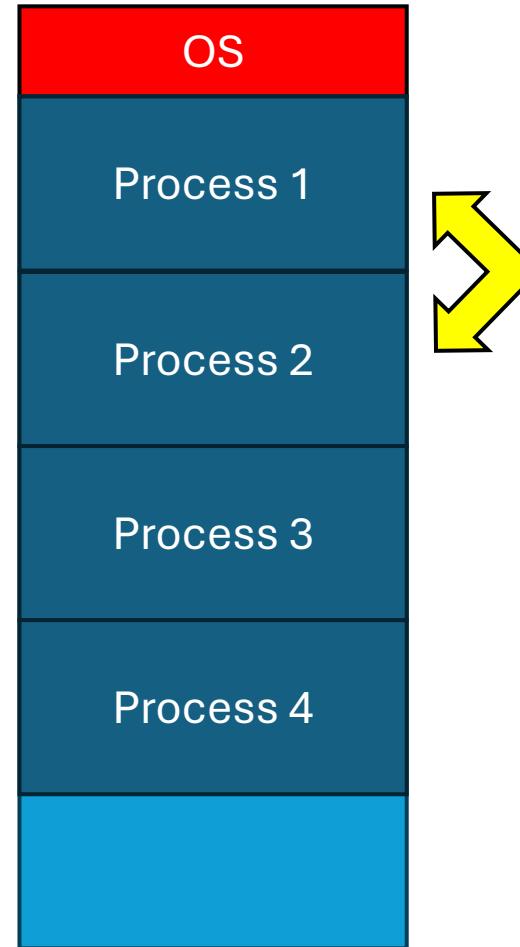
Don't waste memory



Virtualisation

Problem #5:

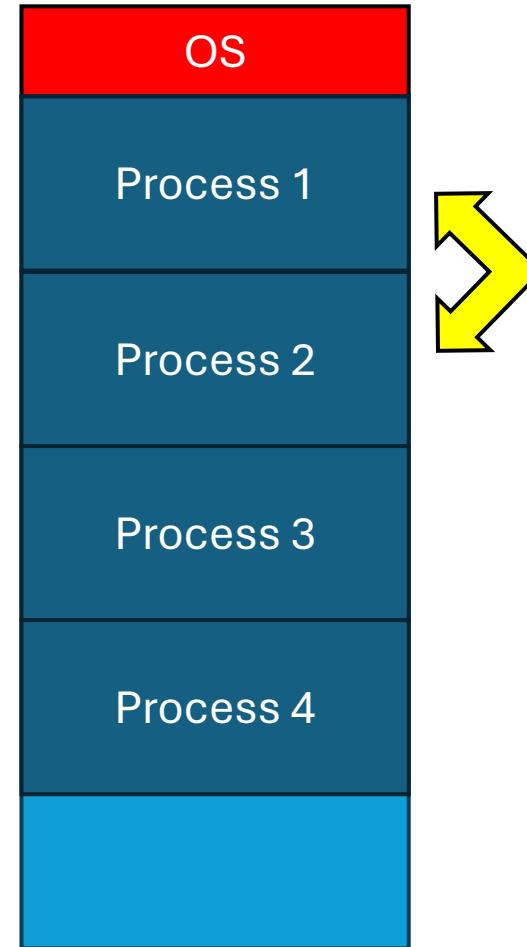
What is problem #5?



Virtualisation: Sharing

Problem #5:

What is problem #5?

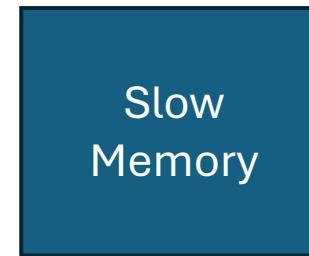
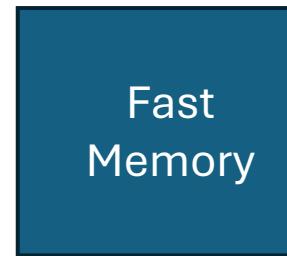


Multi-programming

...

Bad Solution #1: Time Sharing

Time Sharing



code, data,
heap, stack

code, data,
heap, stack

Bad Solution #1

Why is this bad?

Better Solution #1: Space Sharing

Space Sharing

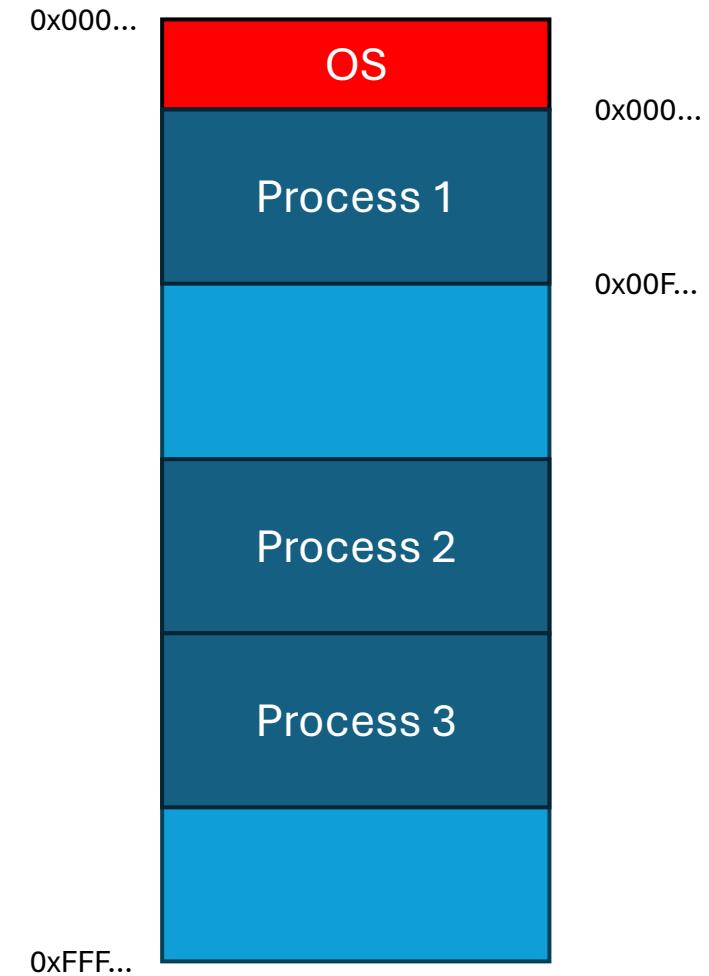
C

```
int main() {  
    int x;  
    x = x + 3;  
}
```

Assembly

```
0x10: movl 0x8(%rbp), %edi  
0x13: addl $0x3, %edi  
0x19: movl %edi, 0x8(%rbp)
```

Bottom of stack



We need some way of converting
addresses efficiently

Static Relocation

Hacky Solution:

- OS program loader re-writes the program!!

0x0010 => 0x**3**010

OR

0x0010 => 0x**5**010

**Why could this
be bad?**

Dynamic Relocation

Goal:

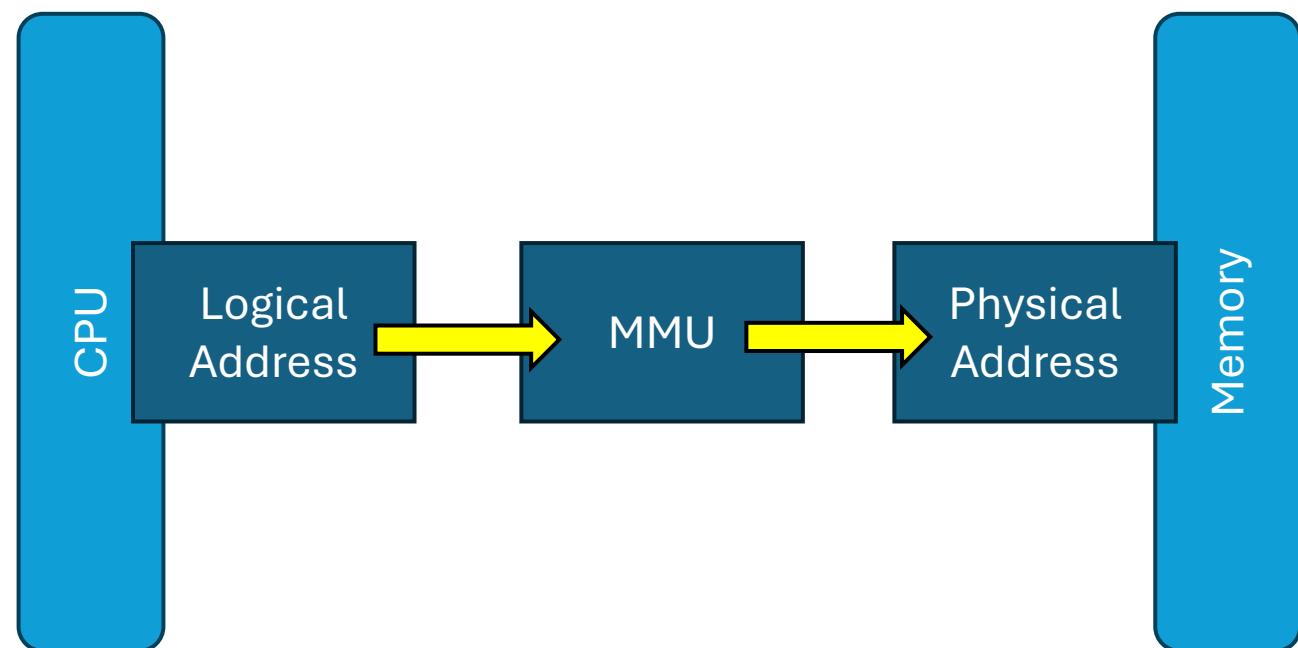
- Memory but with protection...

Requirement:

- Hardware Support

MMU

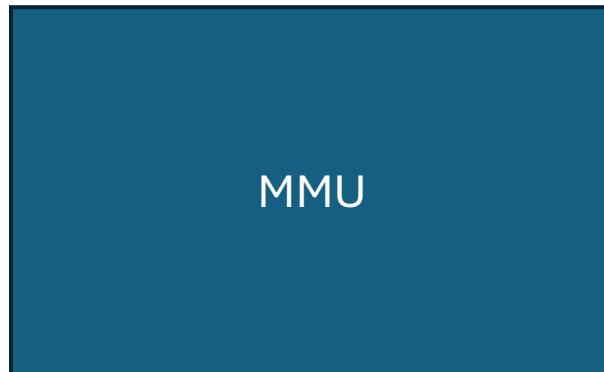
Memory Management Unit



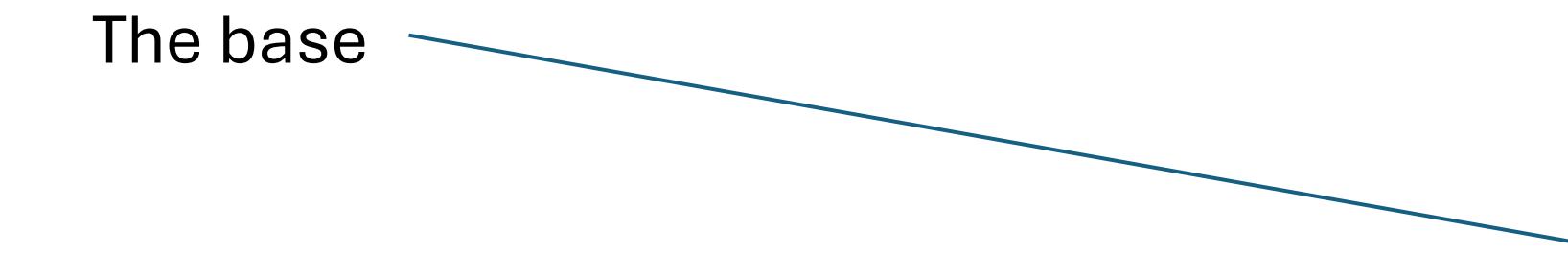
Dynamic Relocation

Two operating modes:

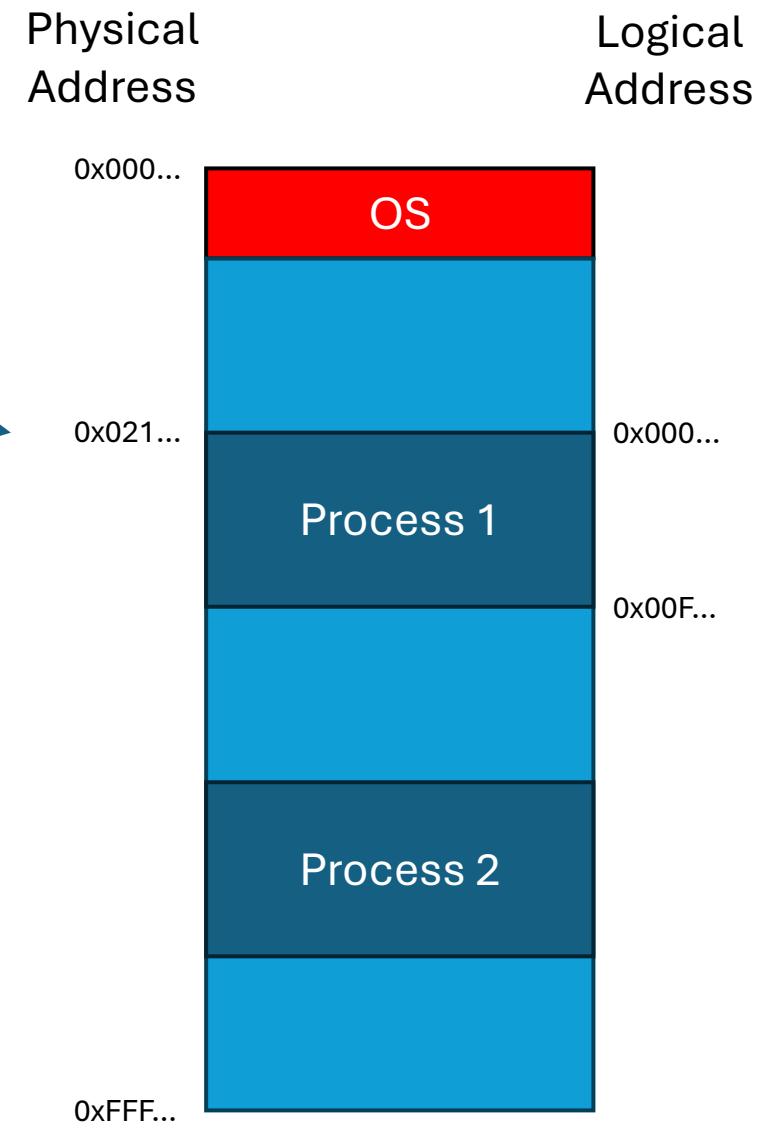
- **Privileged** (Kernel/Protected)
 - When enter OS (trap, system call, interrupts, exceptions)
 - Allows certain instructions
 - Allows OS to **ALL** of physical memory
- **User**
 - Uses ‘logical addresses’



MMU: Base

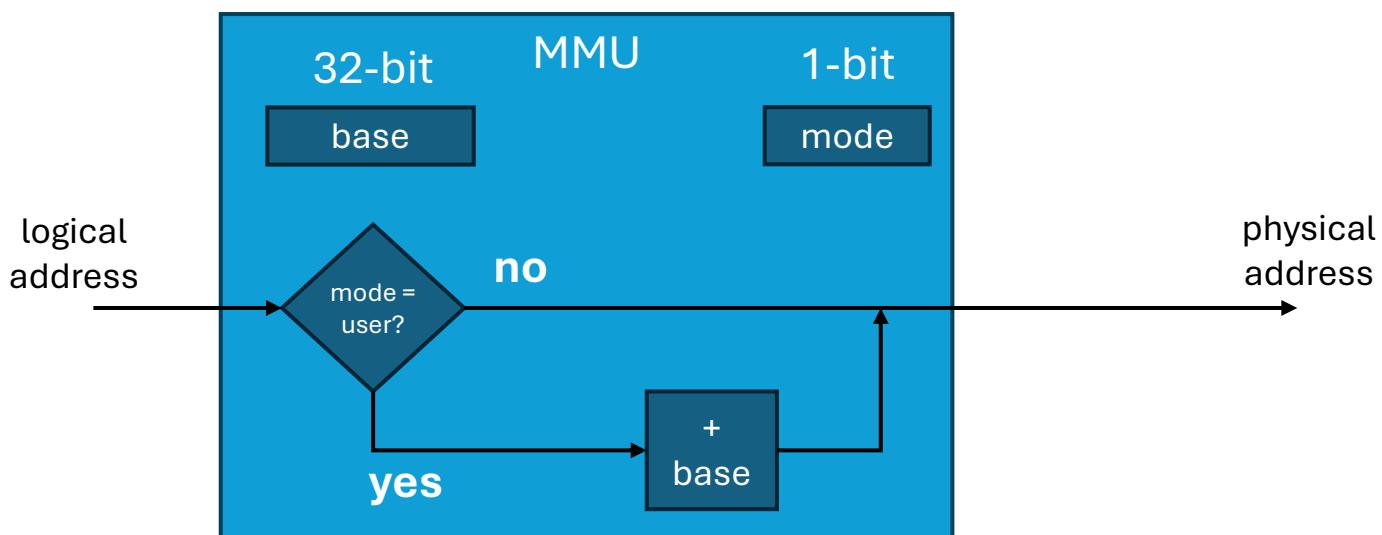


The MMU can check if the requested address (using the OS stored offset) is OK

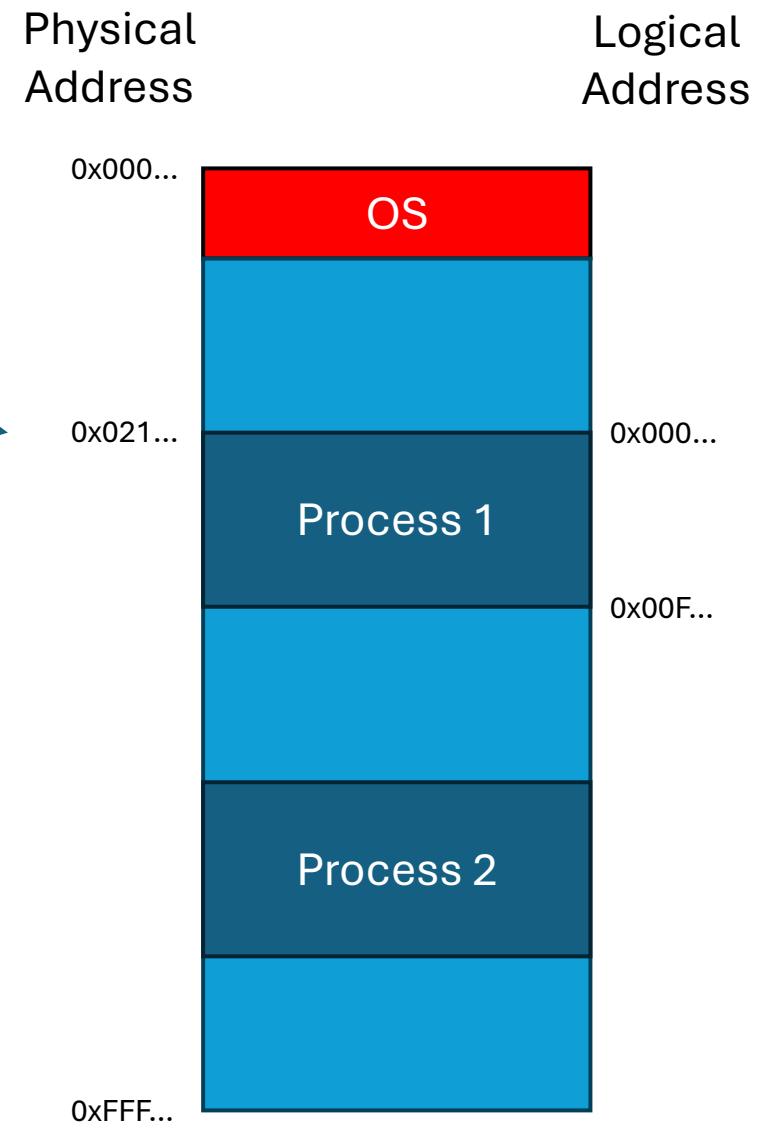


MMU: Base

The base



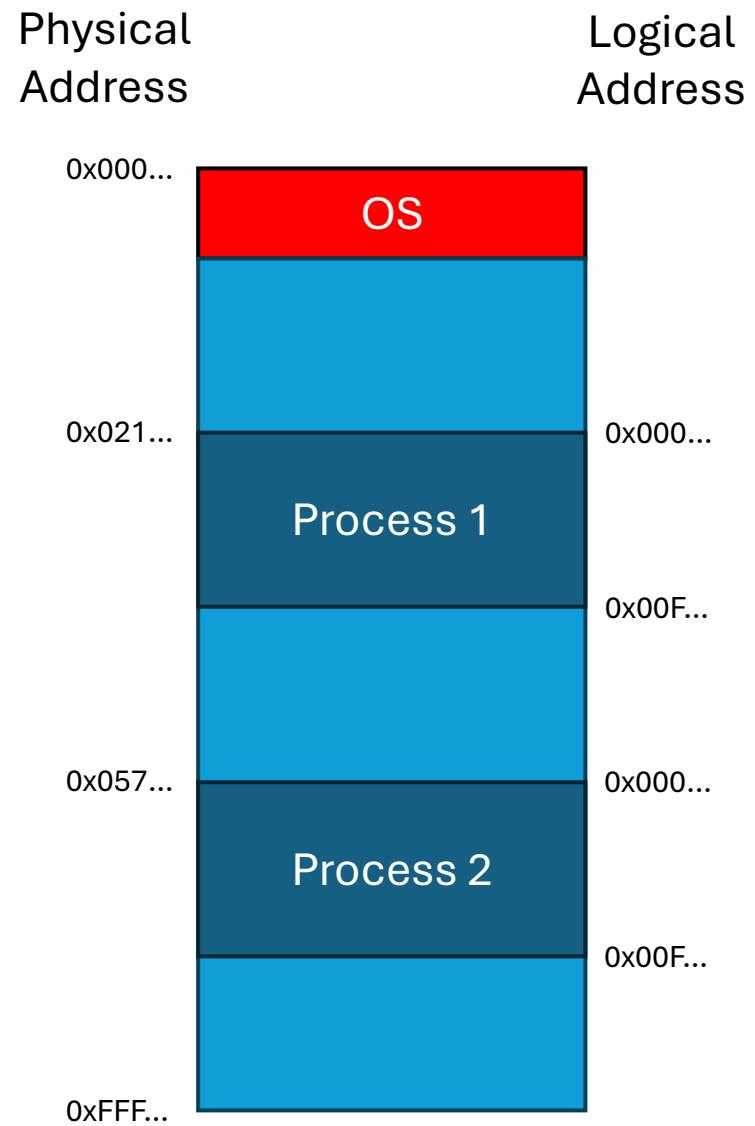
The MMU can check if the requested address (using the OS stored offset) is OK



Dynamic Relocation: Base

Summary:

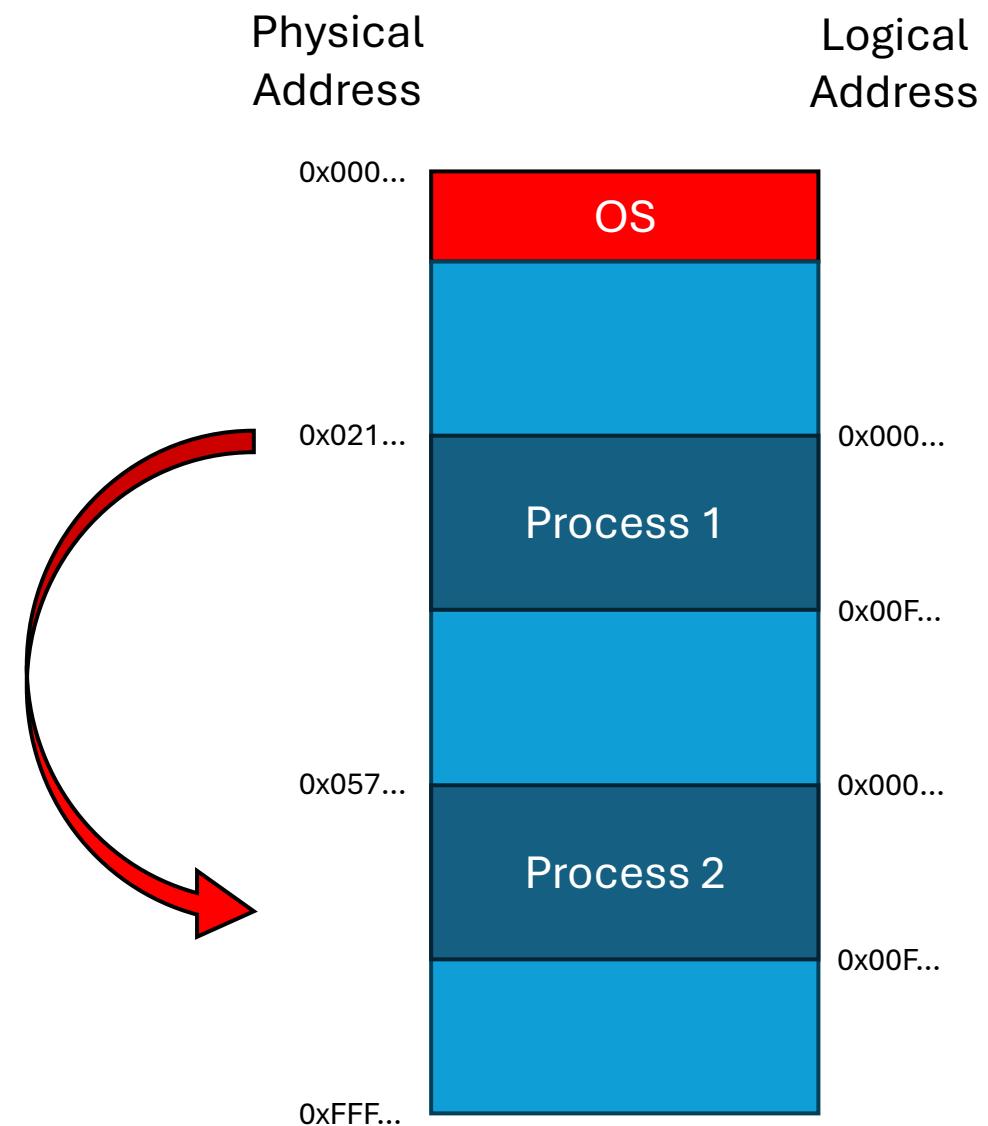
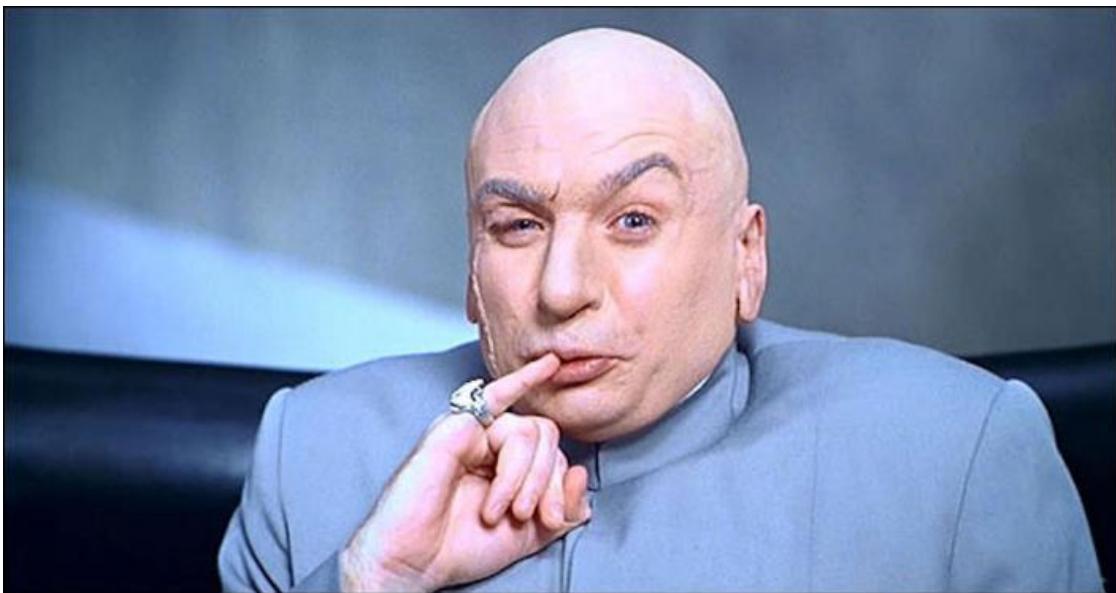
- Store a ‘base register’ value
- Each process has a unique base
- ‘Translate’ means add an offset



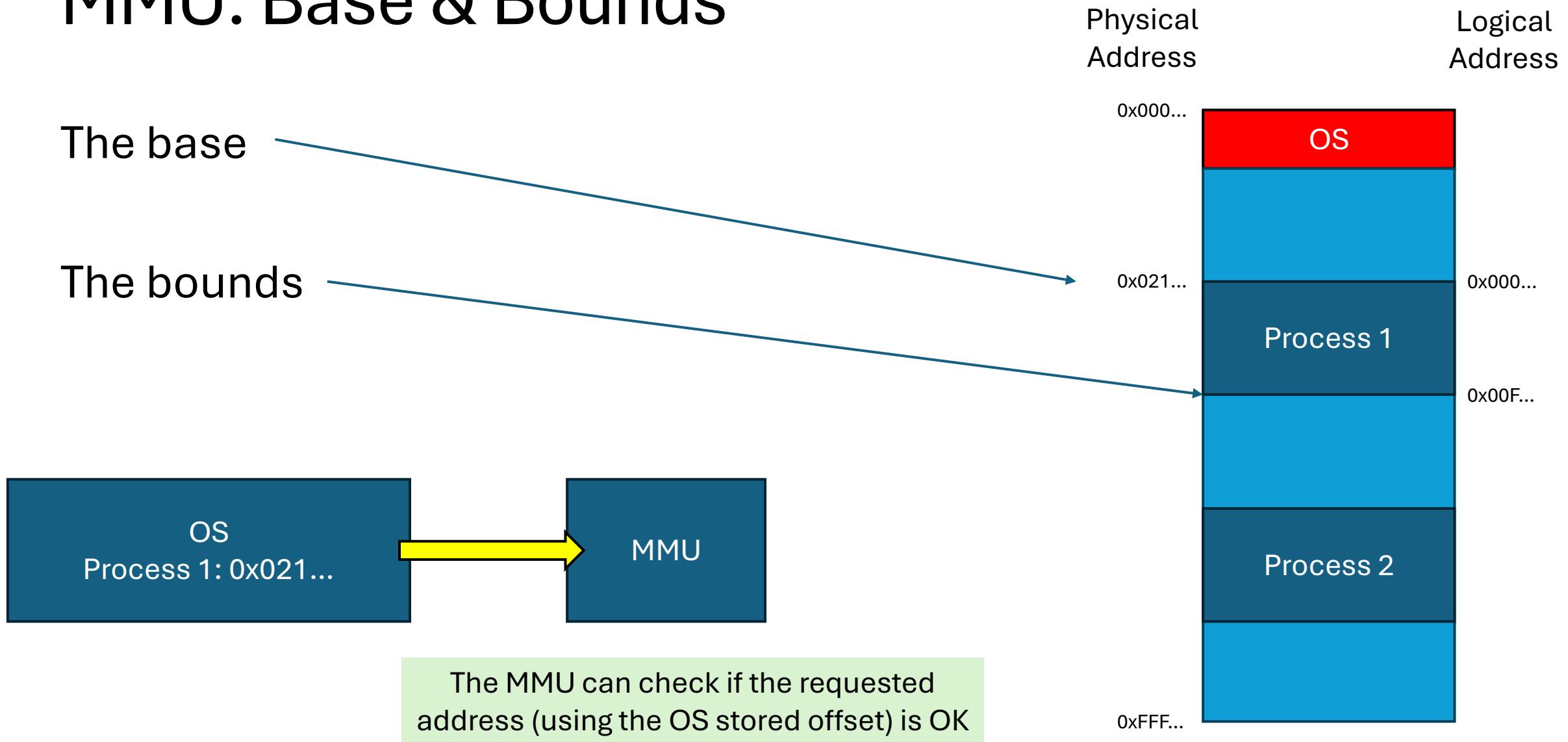
Dynamic Relocation: Base

Protection:

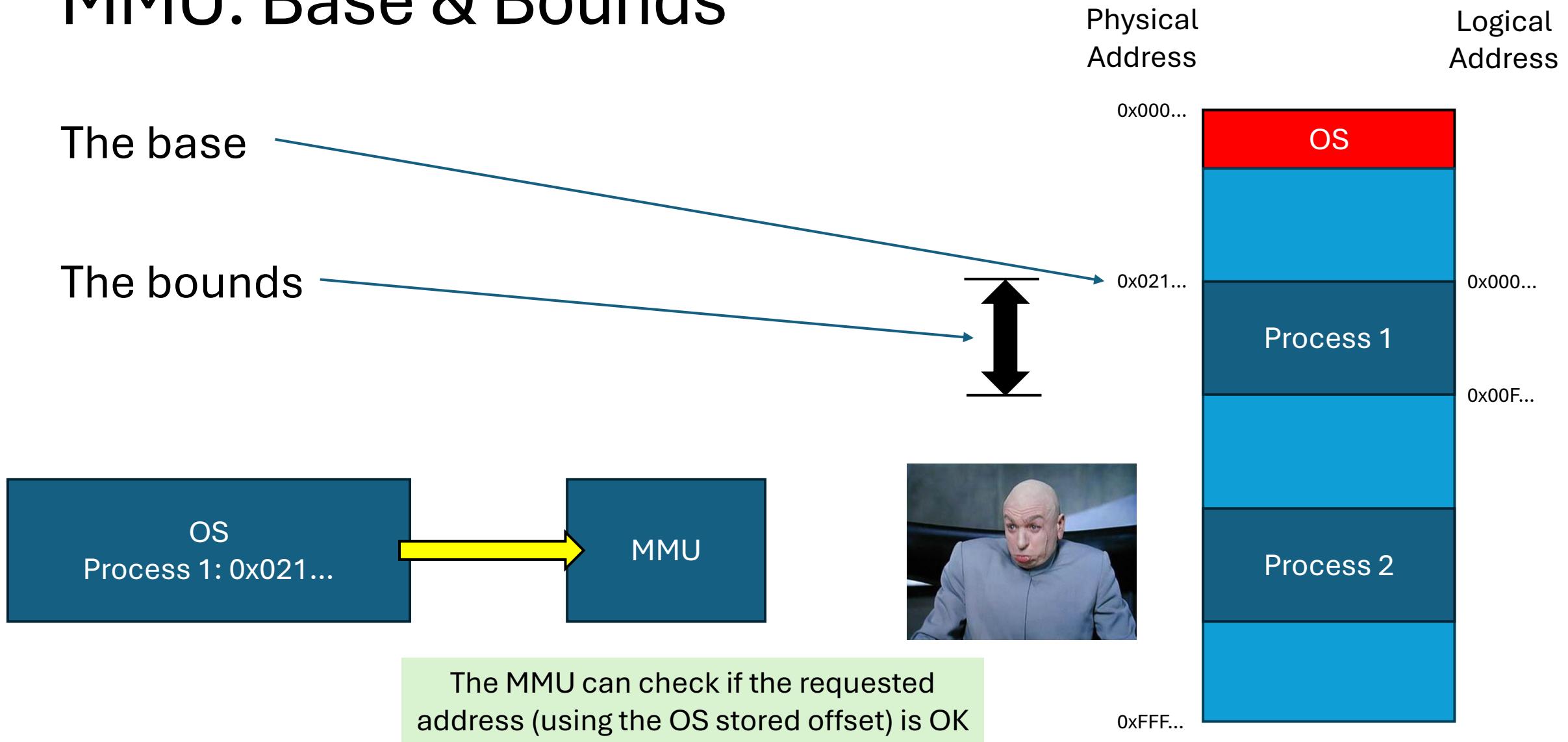
- I would like register... 1 billion



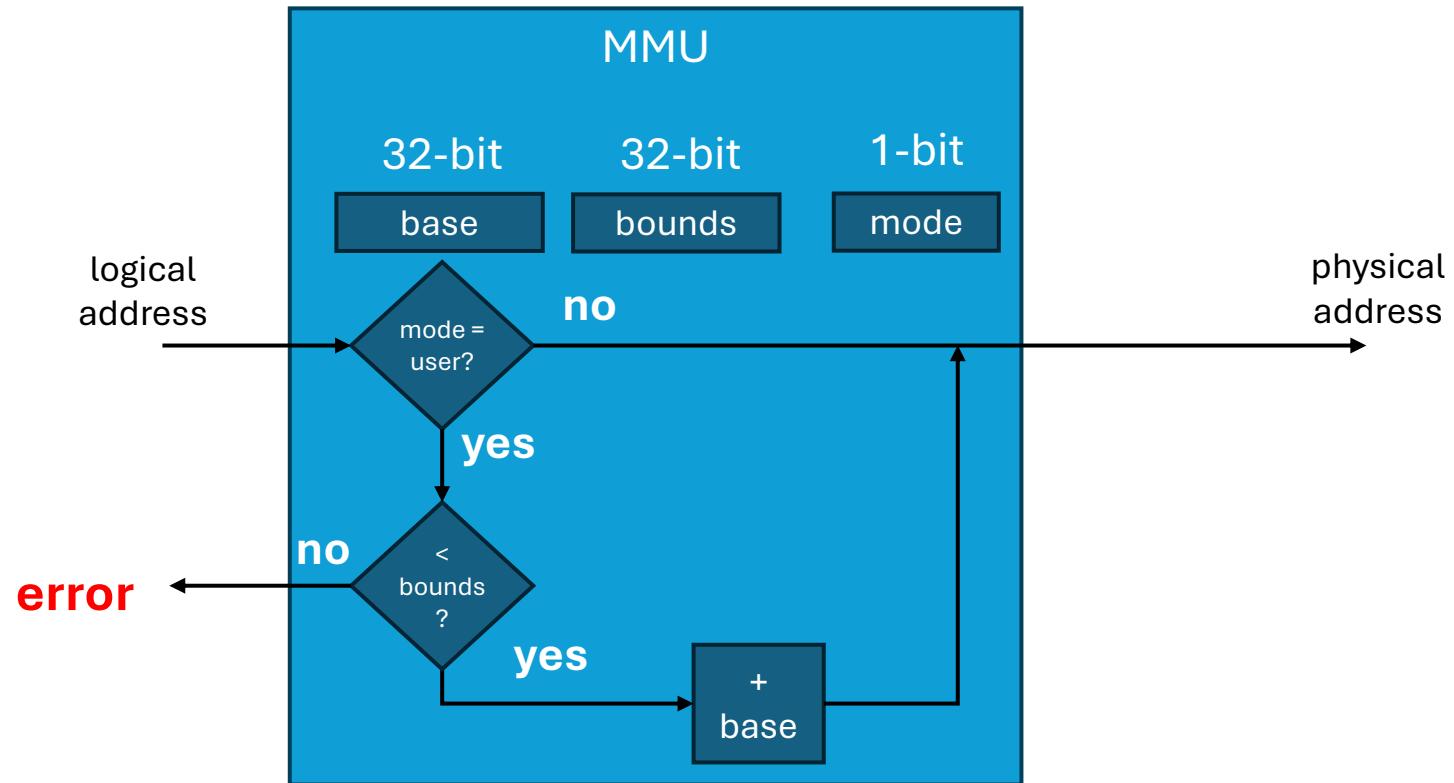
MMU: Base & Bounds



MMU: Base & Bounds



Base & Bounds



We are safe!

The MMU can check if the requested address (using the OS stored offset) is OK

Base and Bounds

Context Switch

- Two new registers (B & B)
- Change to **privileged mode**
- Save B & B (old process)
- Load B & B (new process)
- Change to **user mode**

Some thoughts:

- Can **user** change B&B?
- Can **user** activate **privileged**?
- What about threads?

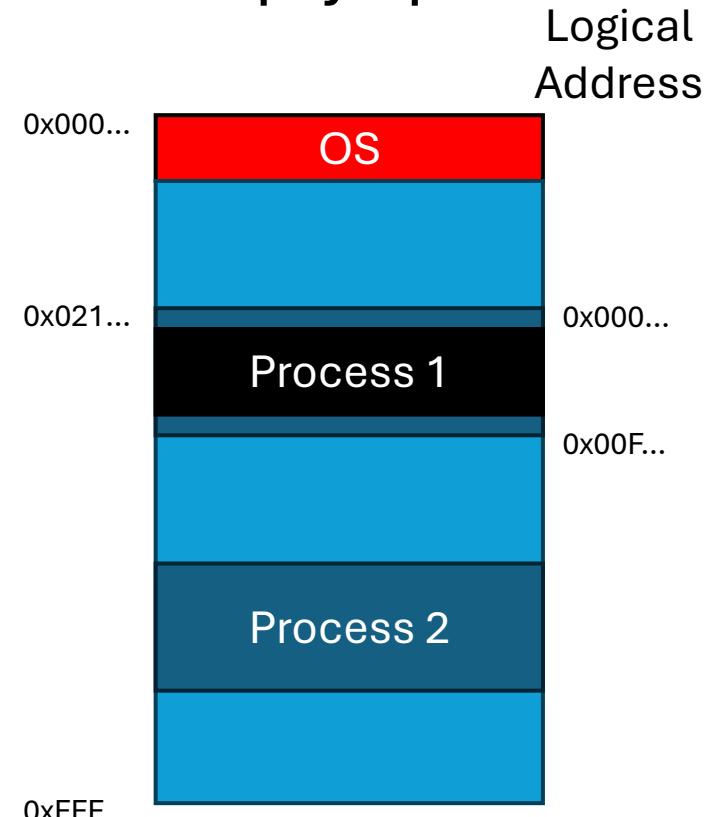
Base and Bounds

Advantages

- Protection/Privacy!
- Dynamic Relocation
- Fast (hardware)
- Small (two registers)

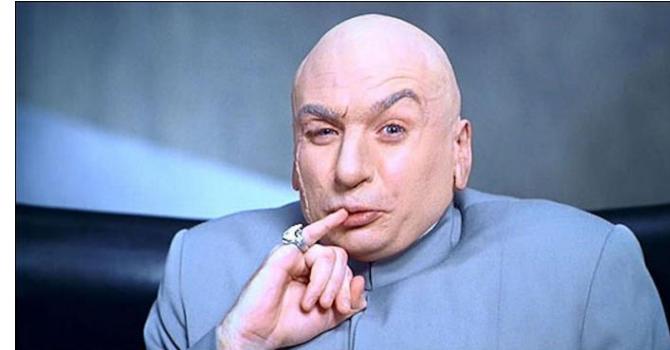
Disadvantages

- Contiguous empty space...



Revision: C-Strings

In C, strings end in `\0`



H | e | l | l | o | \0

But what if... there
were... null nulls?

H | e | l | l | o | **X** | - | - | N | o | t | - | m | i | n | e

Revision: C-Strings

Buffer overflow attacks!

```
int main() {  
    int array[5] = {1, 2, 3, 4, 5};  
    int i;  
    for (i = 0; i < 255; ++i) {  
        array[i] = 41;  
    }  
}
```



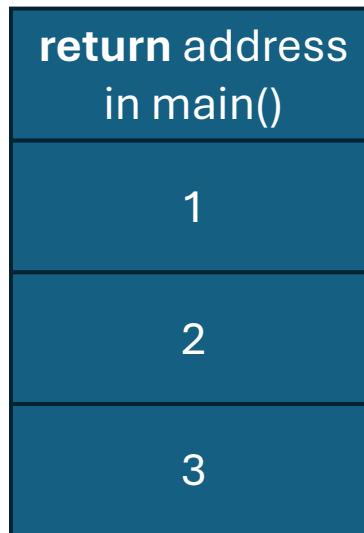
Buffer overflow attacks!

```
int main() {  
    int array[5] = {1, 2, 3, 4, 5};  
    int i;  
    for (i = 0; i < 255; ++i) {  
        array[i] = 41;  
    }  
}
```



Stack Smashing

```
int main() {  
    f(1, 2, 3);  
  
}  
  
pushl $3; constant 3  
pushl $2; (revers order)  
pushl $1  
call f
```



So far... so good?

Stack Smashing

```
void f(int a, int b, int c) {  
    char buffer1[4];  
    char buffer2[12];  
    strcpy(buffer2, "This is a very long string!!!!");  
}
```

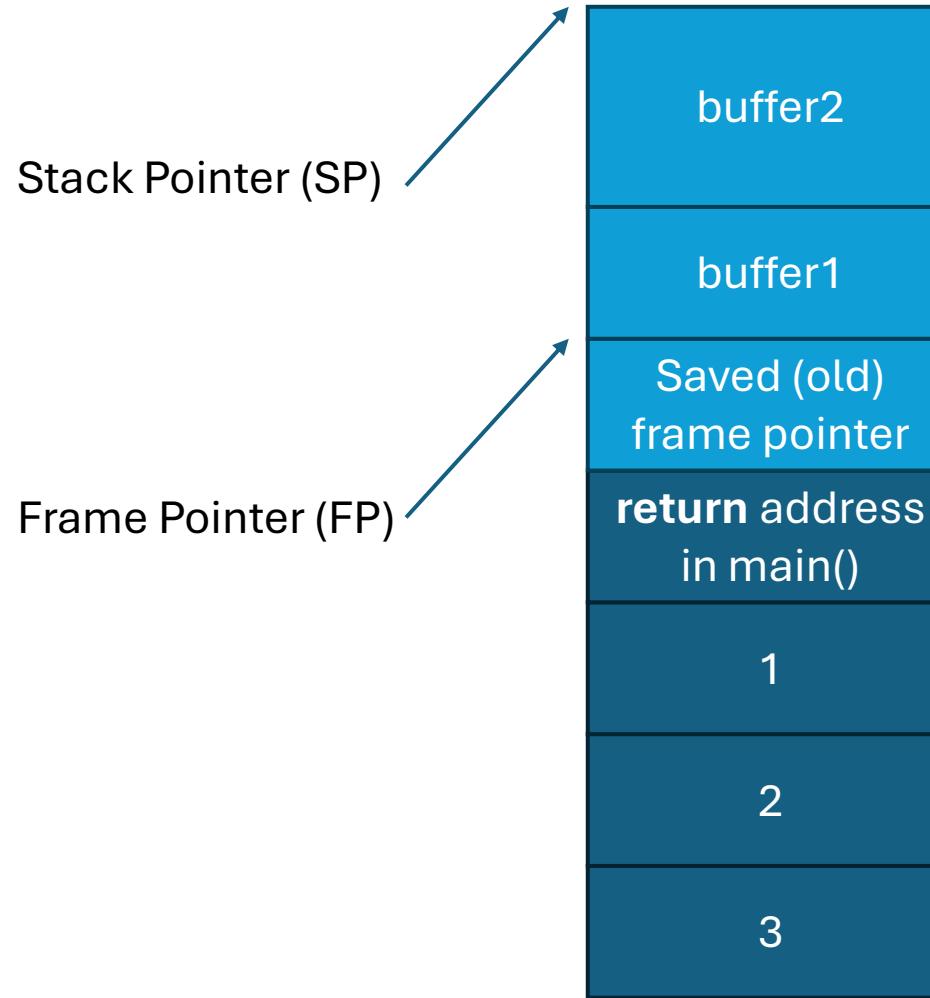
How many characters?

```
pushl %ebp          ; Push old frame pointer (FP)  
movl $esp, %ebp    ; New FP is old SP  
subl $10, %esp     ; New SP is after local vars  
                   ; "$10" is calculated to be >= local var space
```

Stack Smashing

```
void f(int a, int b, int c) {  
    char buffer1[4];  
    char buffer2[12];  
    strcpy(buffer2, "This is a very long string!!!!");  
}
```

SEGFAULT!!!!

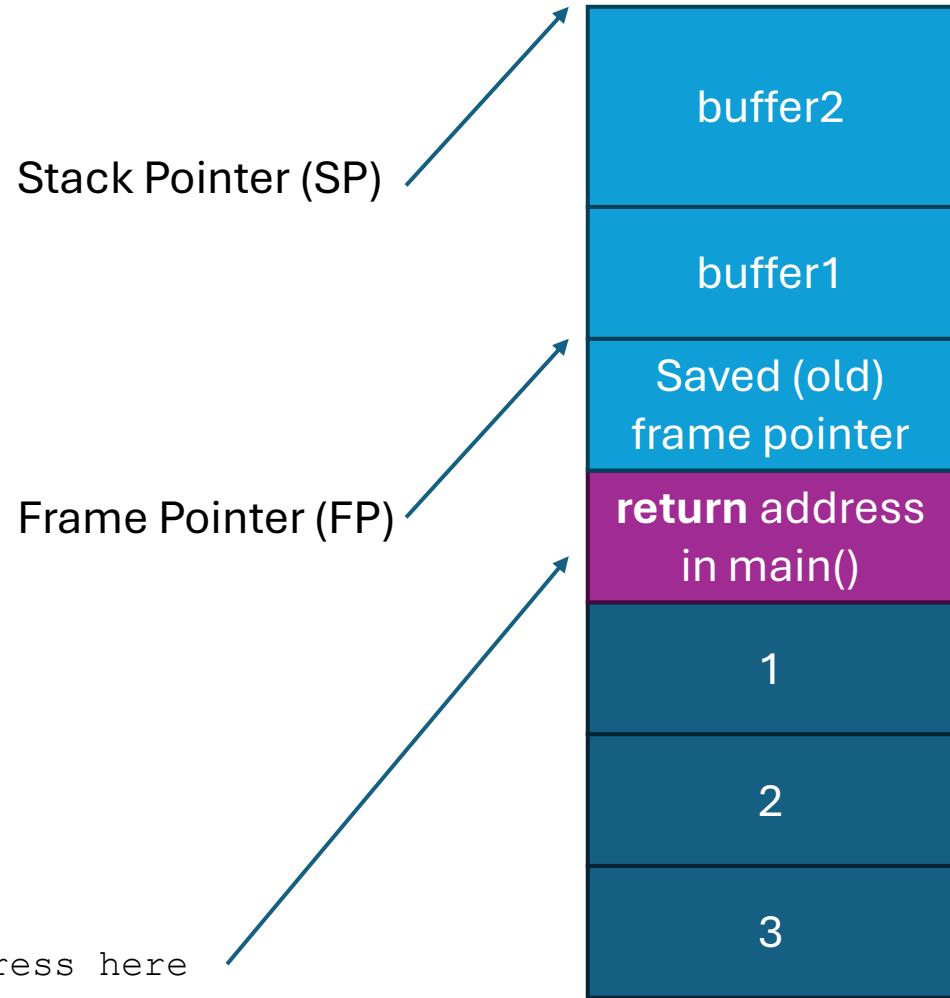


This is a very long string!!!!

Stack Smashing

```
void f(int a, int b, int c) {  
    char buffer1[4];  
    char buffer2[12];  
    strcpy(buffer2, "This is a very long string!!!!");  
}
```

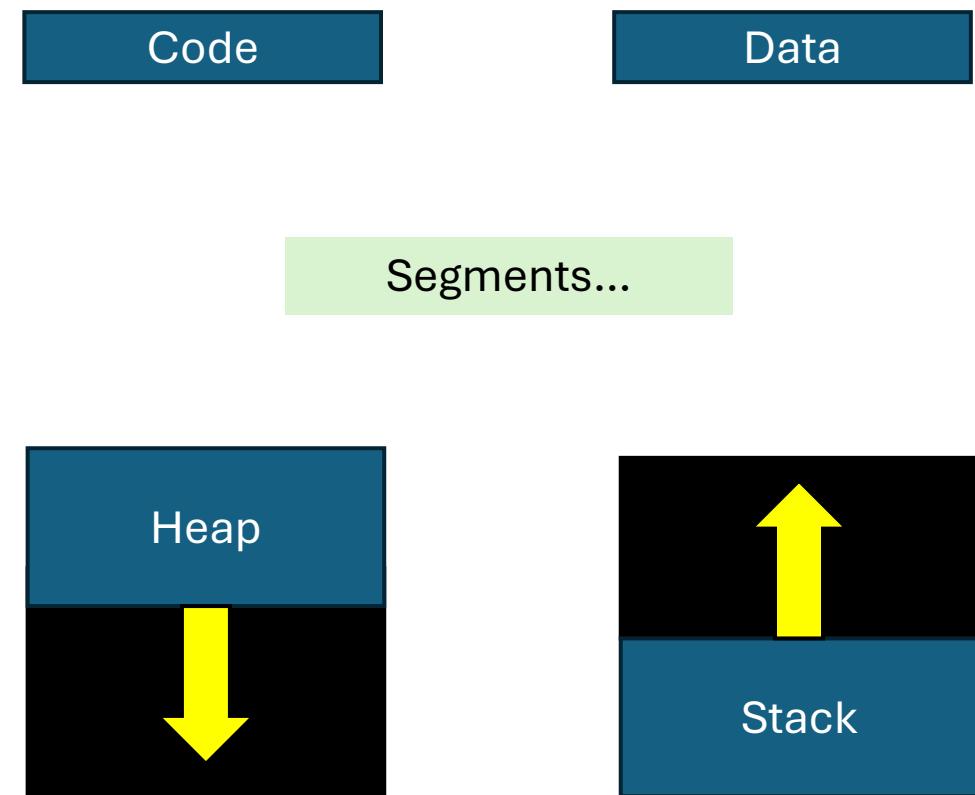
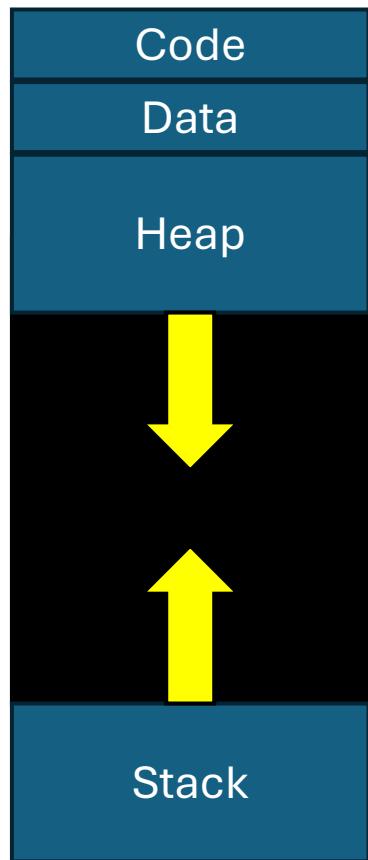
Insert something that looks like an address here



Segmentation

How to break things into bits.

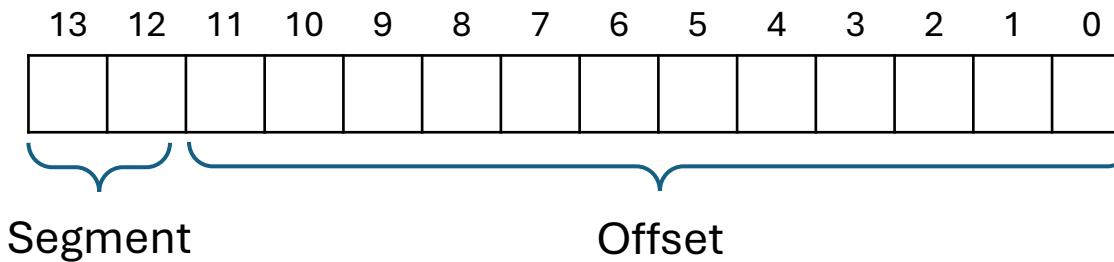
Segmentation: The Idea



Segmentation

The Practicities

- Where are my three segments?
 - Base & Bounds x3



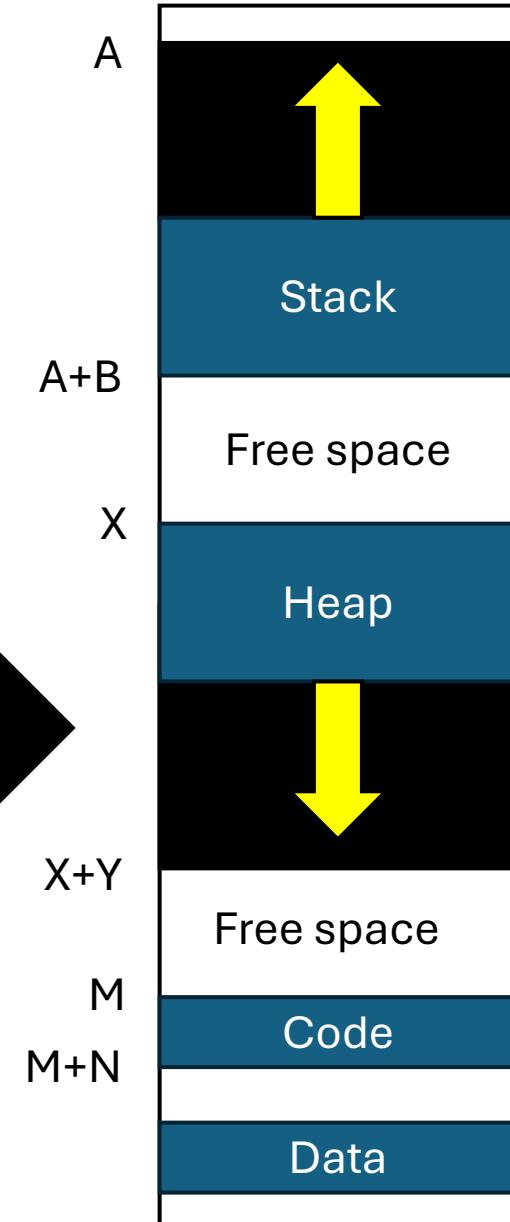
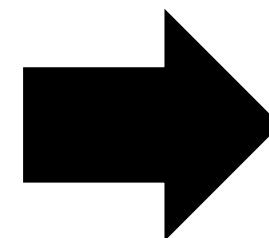
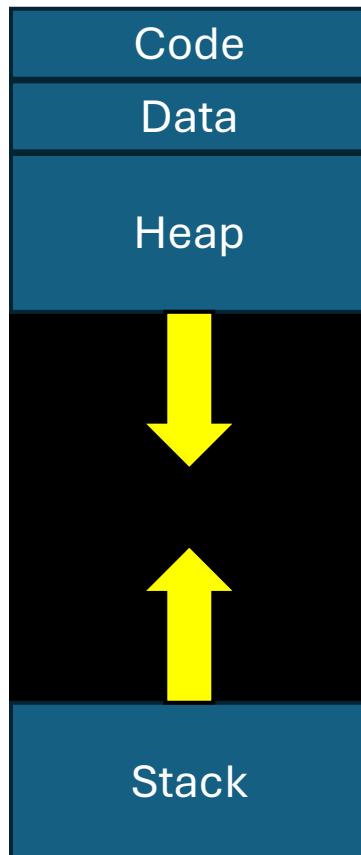
00: Code
01: Heap
10: Stack
11: ???

Segmentation

Segment	Stack
Base	A
Bounds	B

Segment	Heap
Base	X
Bounds	Y

Segment	Code
Base	M
Bounds	N

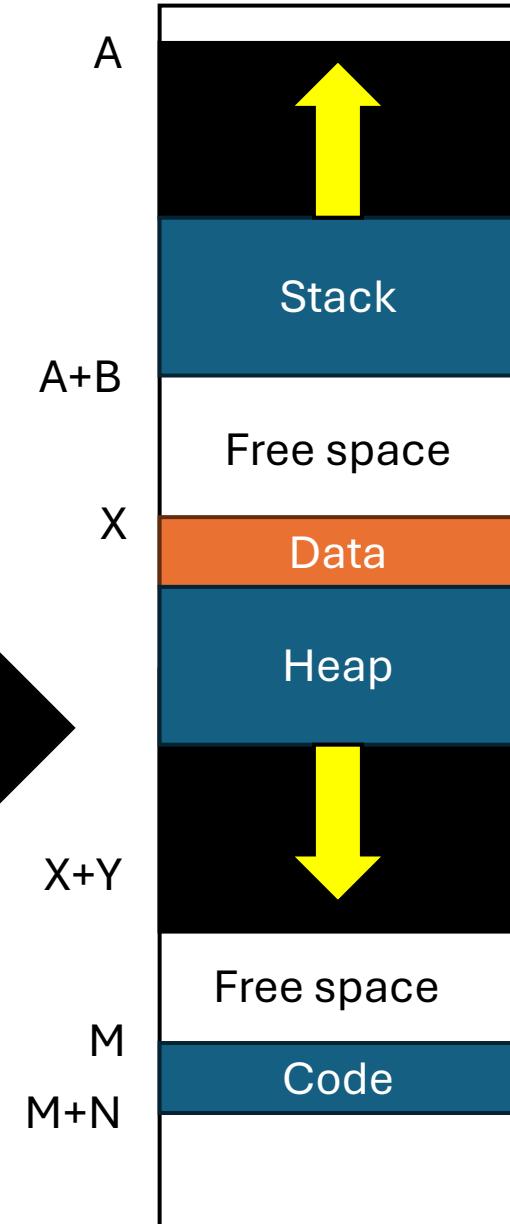
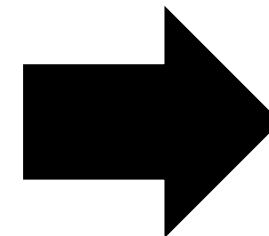
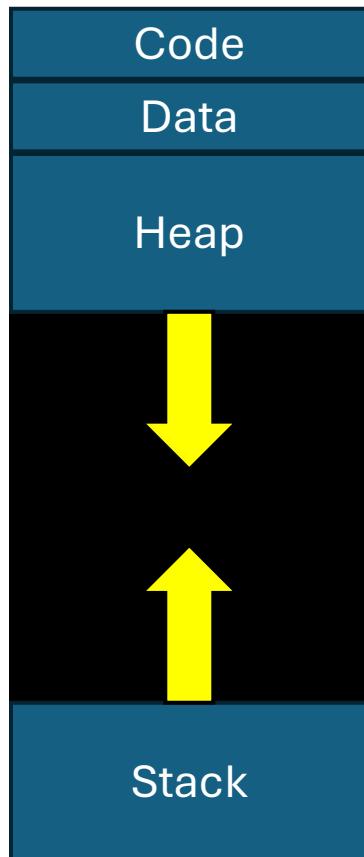


Segmentation

Segment	Stack
Base	A
Bounds	B

Segment	Data & Heap
Base	X
Bounds	Y

Segment	Code
Base	M
Bounds	N

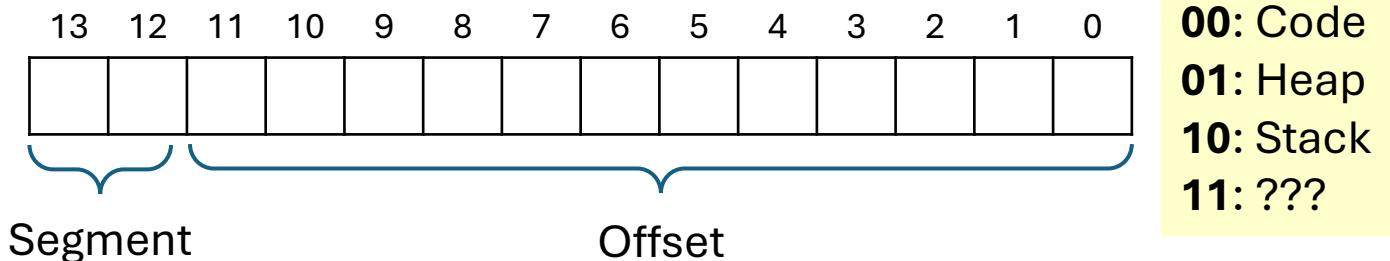
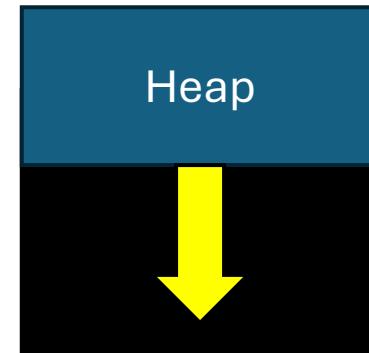
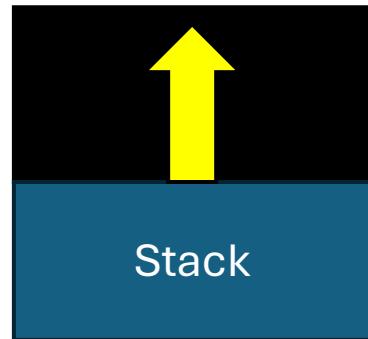


Segmentation: Up vs Down

How do we determine up vs down?

Options:

- Use the header
- Have a more complex structure (x86)
 - Many bits: privilege etc.
 - R/W/E?



Memory Accesses

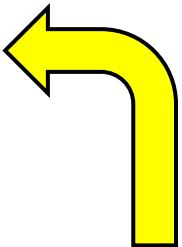
Segment	Base	Bounds
Code	0x0000	0x3FFF
Data	0x2000	0x3FFF
Stack	0x3000	0x3FFF

mov ax, 0x4002 → 0100 0000 0000 0010

Memory

Memory Accesses

Segment	Base	Bounds
Code	0x0000	0x3FFF
Data	0x2000	0x3FFF
Stack	0x3000	0x3FFF

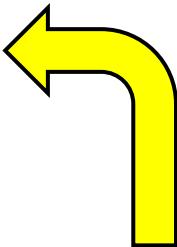


mov ax, 0x4002 → 0100 0000 0000 0010

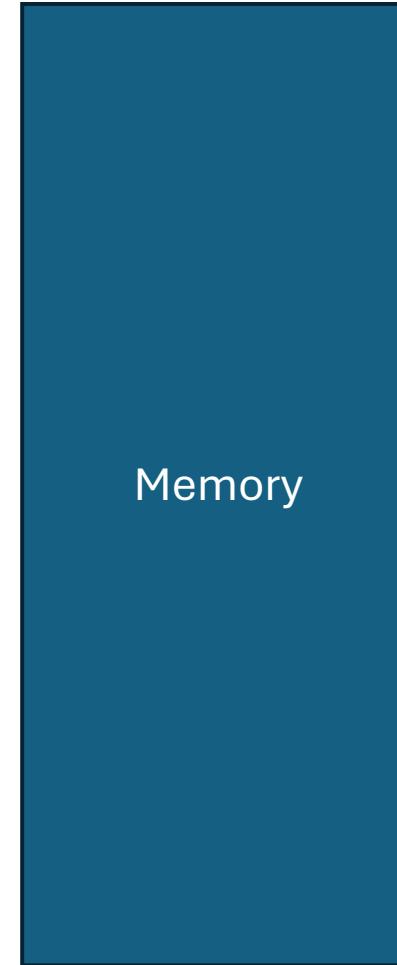


Memory Accesses

Segment	Base	Bounds
Code	0x0000	0x3FFF
Data	0x2000	0x3FFF
Stack	0x3000	0x3FFF

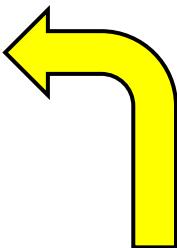


mov ax, 0x4002 → 0100 0000 0000 0010
 00 0000 0000 0010
 +
 10 0000 0000 0000
 =
 10 0000 0000 0010
 (0x2002)



Memory Accesses

Segment	Base	Bounds
Code	0x0000	0x3FFF
Data	0x2000	0x3FFF
Stack	0x3000	0x3FFF



mov ax, 0x4002



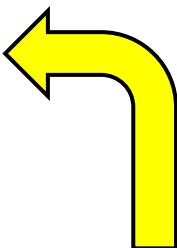
$$\begin{array}{r} \textcolor{blue}{01}00\ 0000\ 0000\ 0010 \\ 00\ 0000\ 0000\ 0010 \\ + \\ 10\ 0000\ 0000\ 0000 \\ = \\ 10\ 0000\ 0000\ 0010 \\ (0x2002) \end{array}$$

0x2002

Memory

Memory Accesses

Segment	Base	Bounds
Code	0x0000	0x3FFF
Data	0x2400	0x3FFF
Stack	0x3000	0x3FFF



mov ax, 0x4002



0100 0000 0000 0010

00 0000 0000 0010

+

10 0100 0000 0000

=

10 0100 0000 0010

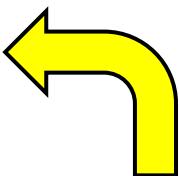
(0x2402)

0x2402

Memory

Memory Accesses

Segment	Base	Bounds
Code	0x0000	0x3FFF
Data	0x2400	0x3FFF
Stack	0x3000	0x3FFF



mov ax, 0x8002



1000 0000 0000 0010

00 0000 0000 0010

+

11 0000 0000 0000

=

11 0000 0000 0010

(0x3002)

0x3002

Memory

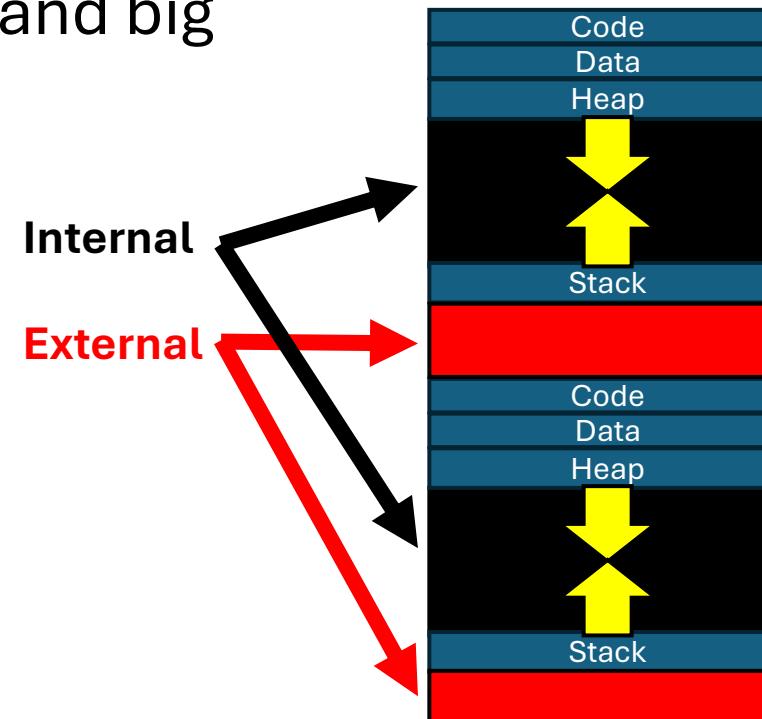
Segmentation Summary

Benefits:

- We can reduce internal fragmentation (i.e., save the space between the stack and the heap which may not be used).
- Stack and heap can grow independently (we can extend the stack as needed)
- Segments have specified protection (rwe)
- Could do sharing?
- Can move segments around

Drawbacks:

- Trading internal fragmentation for external fragmentation woes...
- Segments are still contiguous and big



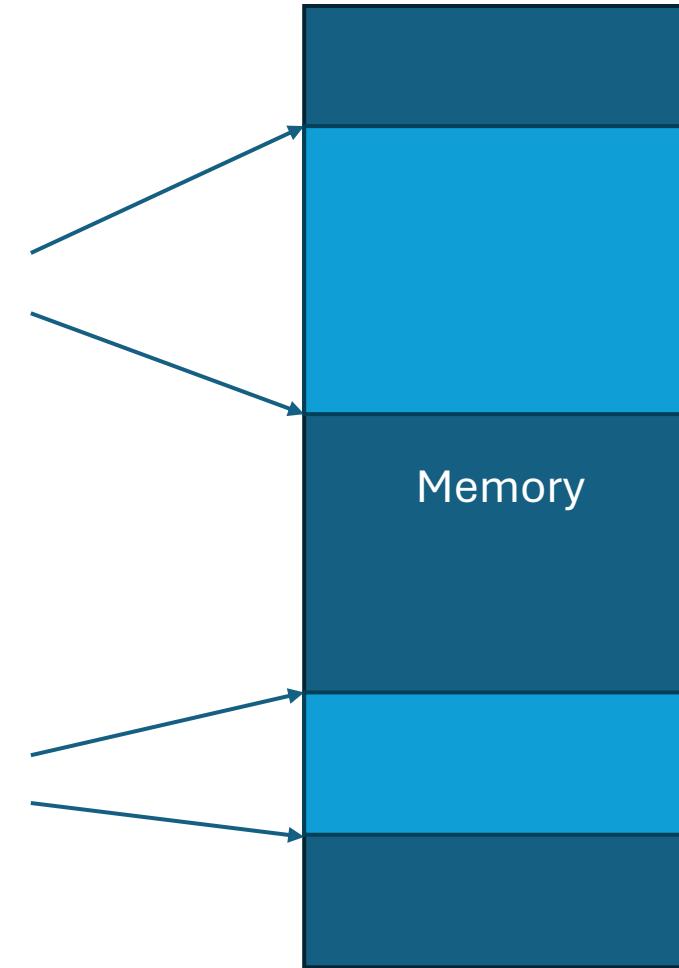
Paging

How to ‘lookupify’ OS memory

Paging

Memory not virtualised

- Address translation (B&B)



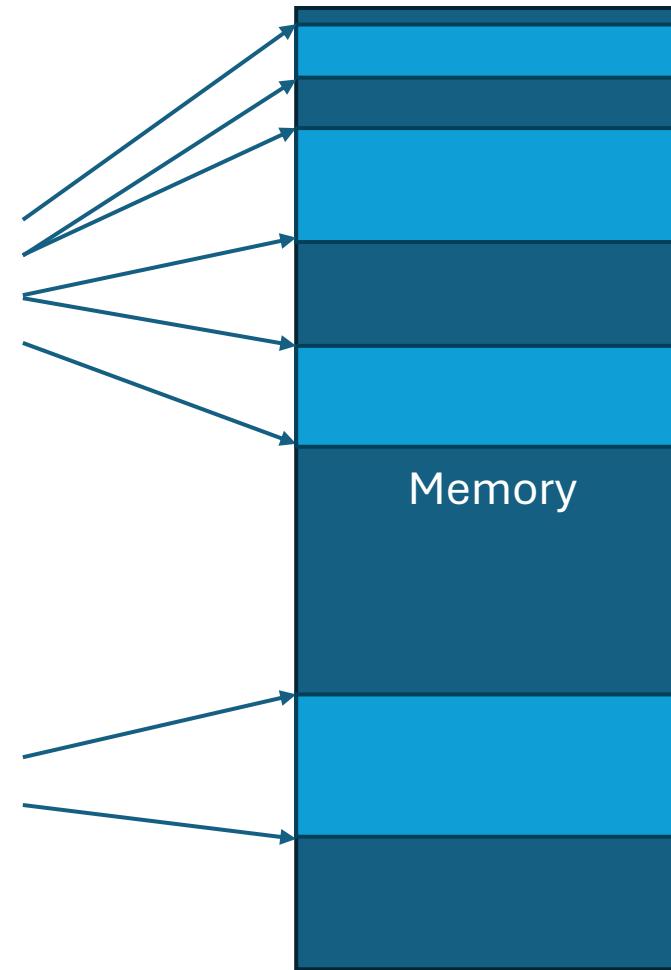
Paging

Memory not virtualised

- Address translation (B&B)

Internal Fragmentation

- Segmentation



Paging

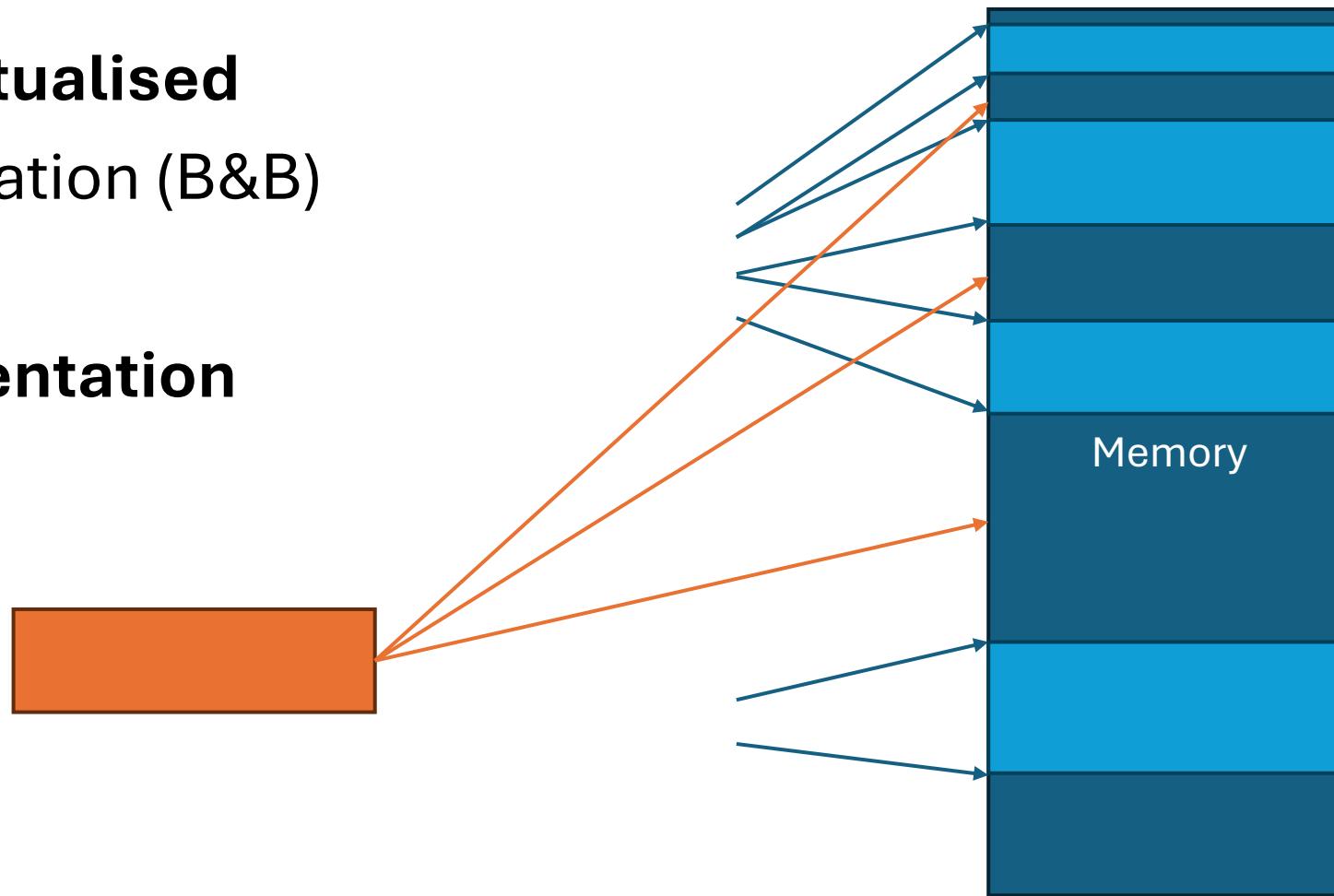
Memory not virtualised

- Address translation (B&B)

Internal Fragmentation

- Segmentation

Does it fit?



Paging

Memory not virtualised

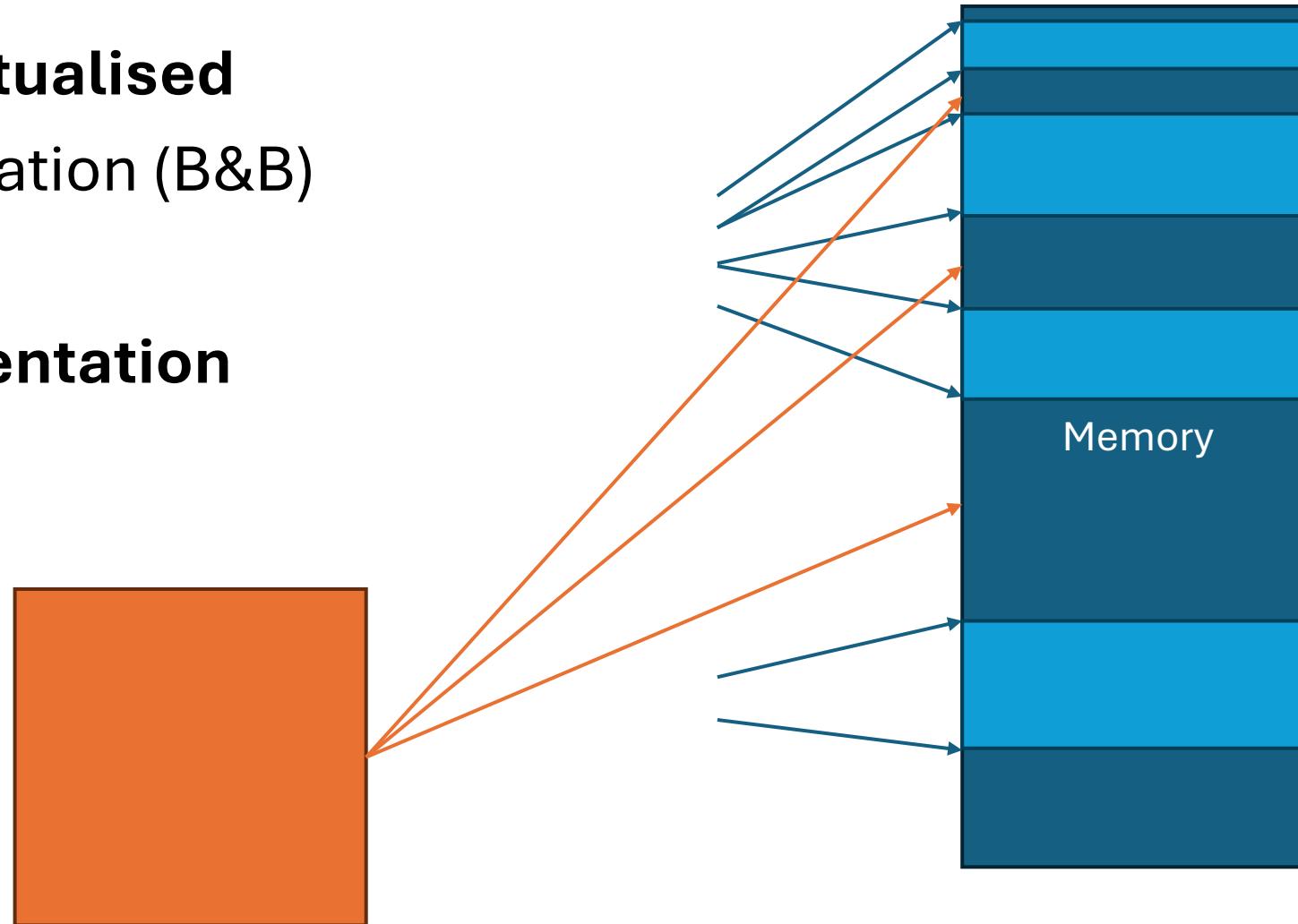
- Address translation (B&B)

Internal Fragmentation

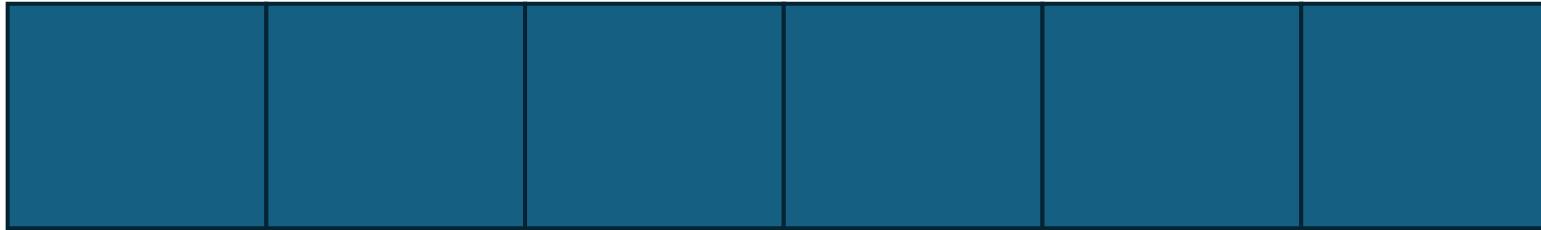
- Segmentation

Does it fit?

Too big?

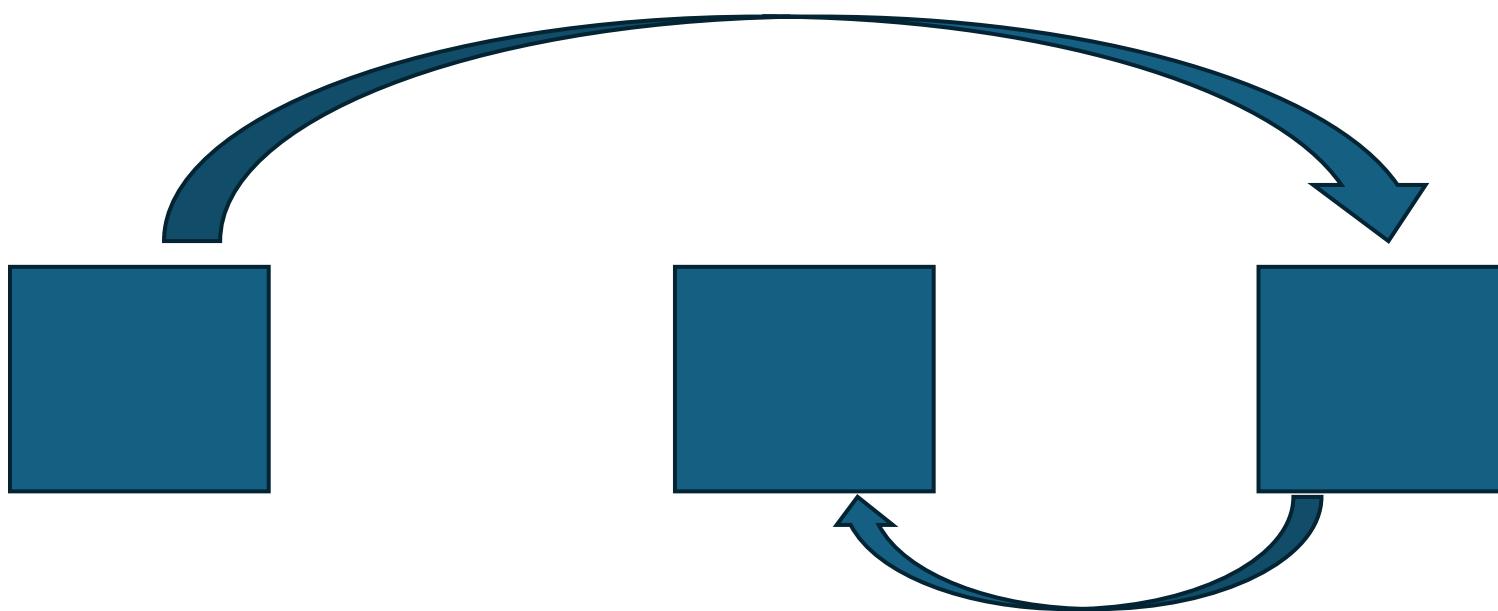


Paging



Array

**Which is better?
Why?**

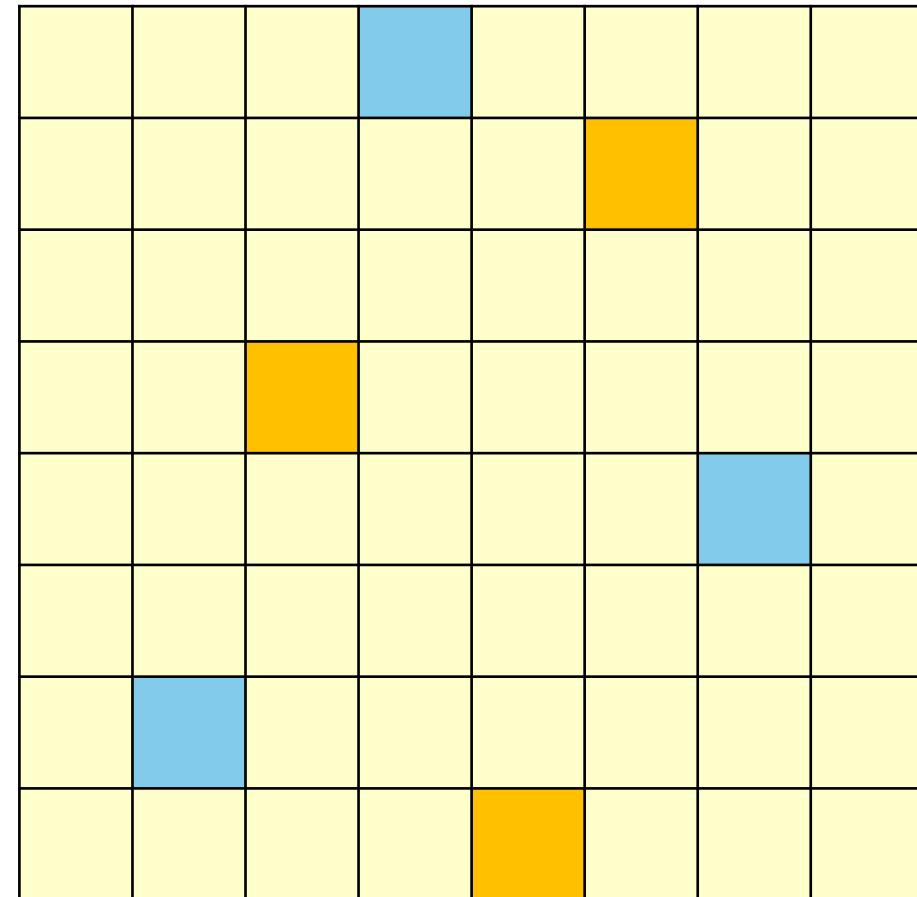


Linked List

Paging

Idea:

- Turn memory into a big array of fixed-sized pages.
- All processes are given a number of pages



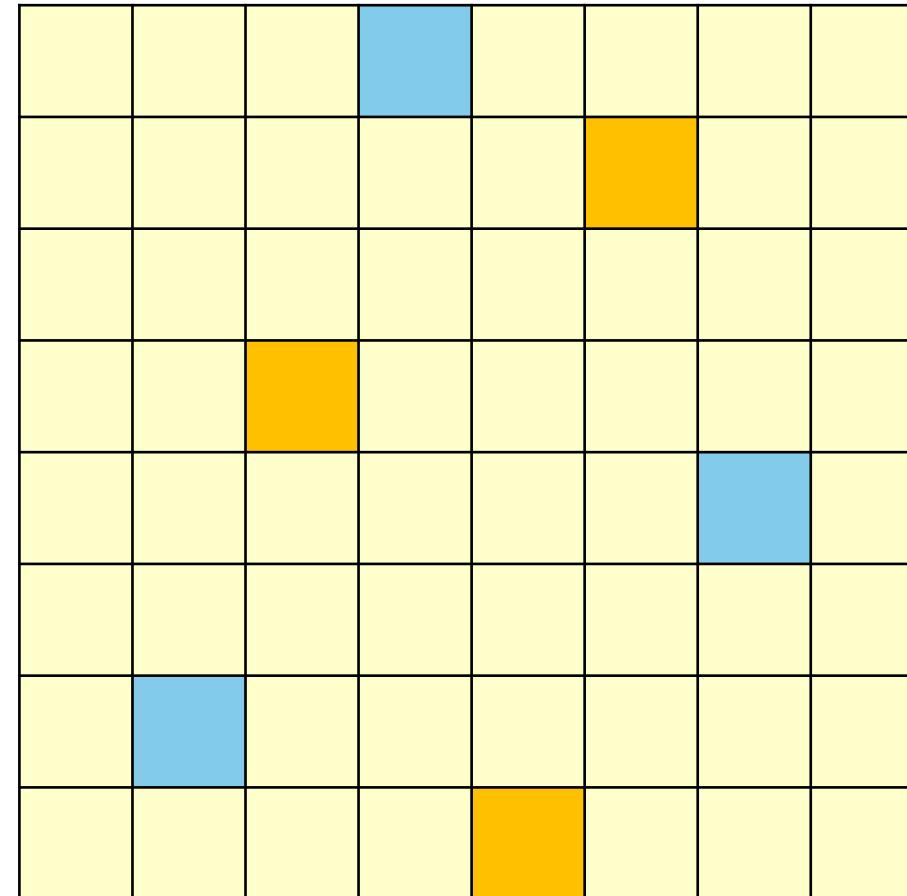
Paging

Idea:

- Turn memory into a big array of fixed-sized pages.
 - All processes are given a number of pages

Result:

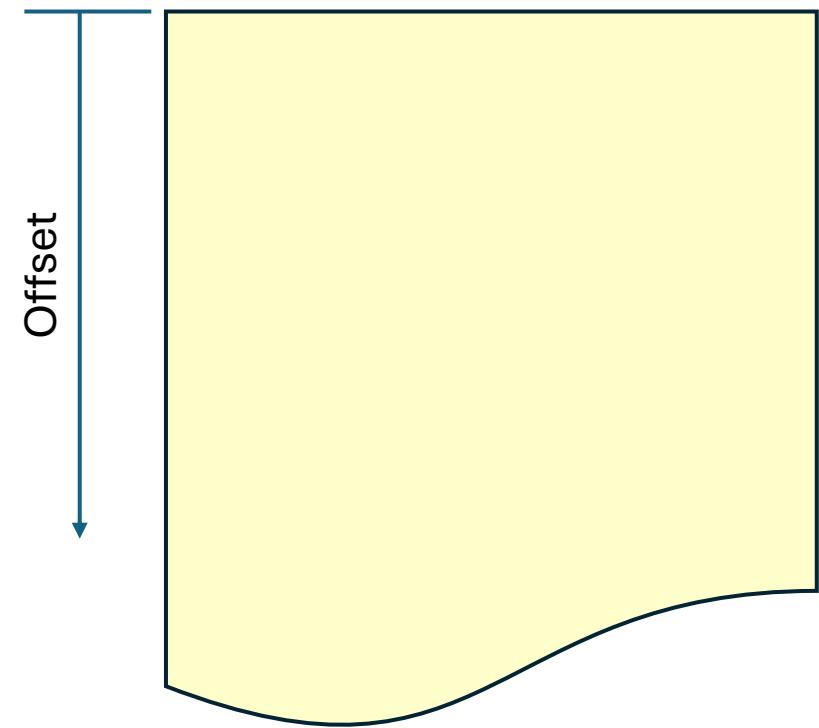
- No external fragmentation
(only internal fragmentation)



Paging

Design Considerations

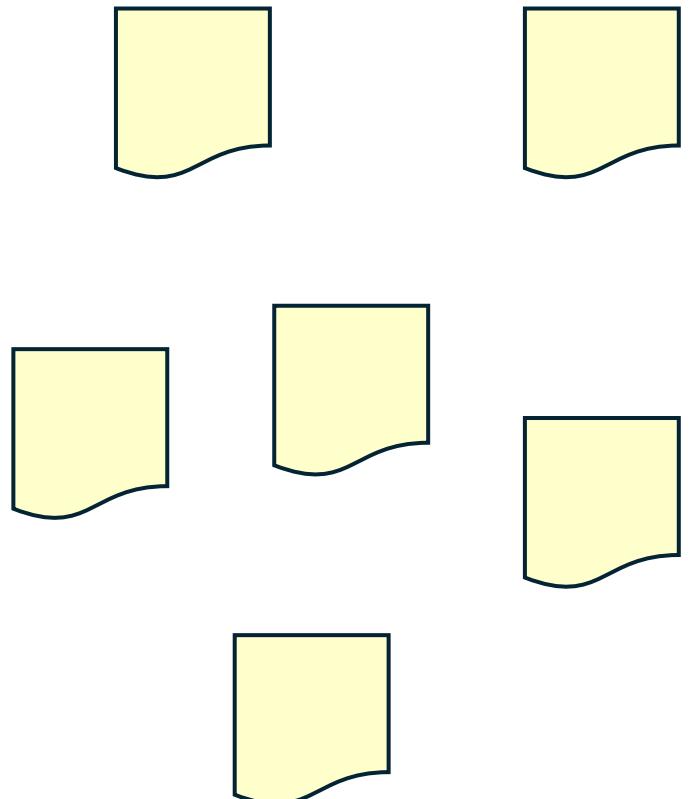
- How big is a page?
 - How many can you have?
 - How much can you address into a page?
 - How much space is wasted?



Paging

Design Considerations

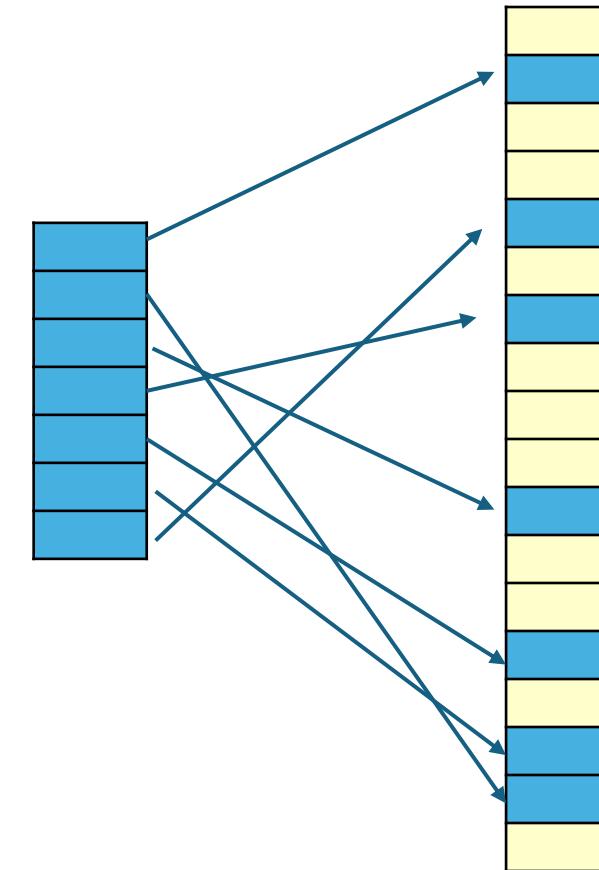
- How big is a page?
 - How many can you have?
 - How much can you address into a page?
 - How much space is wasted?
- How do we ‘get to’ a page?
 - How many pages can you have?



Paging: What does it look like?

Paging:

- Each process is broken up into page-size chunks
- Each chunk is stored in a page of memory



Paging: Addressing

How to address an address within a page



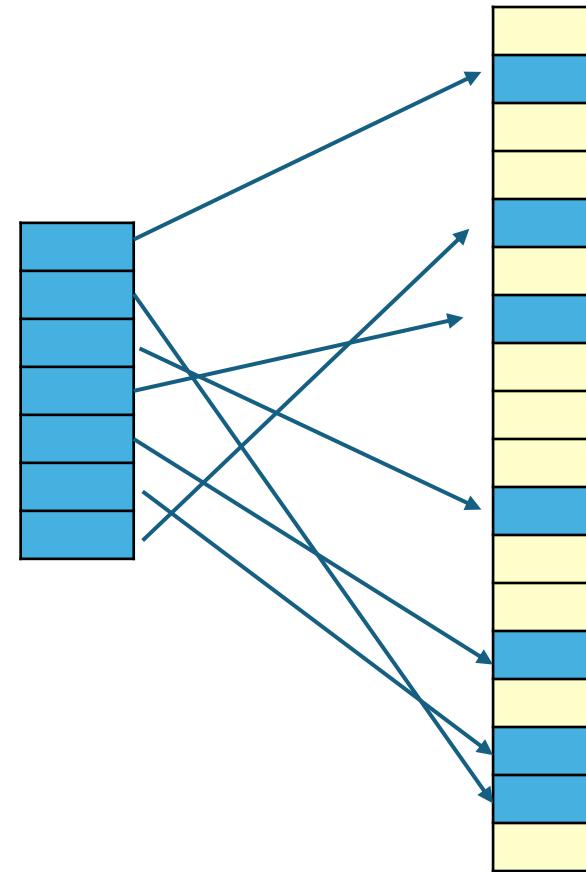
Constrains the
number of pages

Constrains the
size of the page



Software
(Logical)

Hardware
(Physical)



Paging: Page Sizes

Page Size	Low Bits
16 bytes	4
512 bytes	9
1 KB	10
4 KB	12
1 MB	20

Page Sizes

- They don't actually have to be the same size
 - You could have small and big pages in some contexts

Paging: Page Table Size

How big is a page?

- Usually 4 KB

Well...

$$2^{32} / 2^{12} = 2^{20} = 1 \text{ million-ish}$$

How big is a page table?

- One entry... 32 bits/64 bits?

Each entry is 32 bits... (4 bytes)

How many entries?

=> 4 MB page table...

Paging Page: Table Size

Why 4 MB?

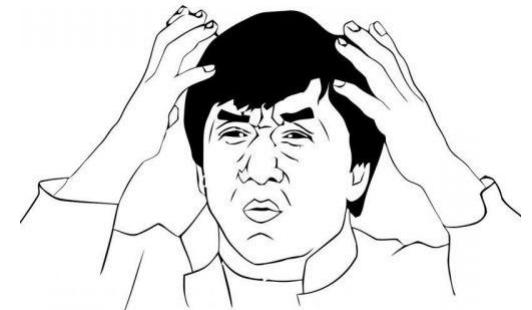
- What if you give it the wrong address?
- How does the hardware know its wrong?

OS:

4 MB per process

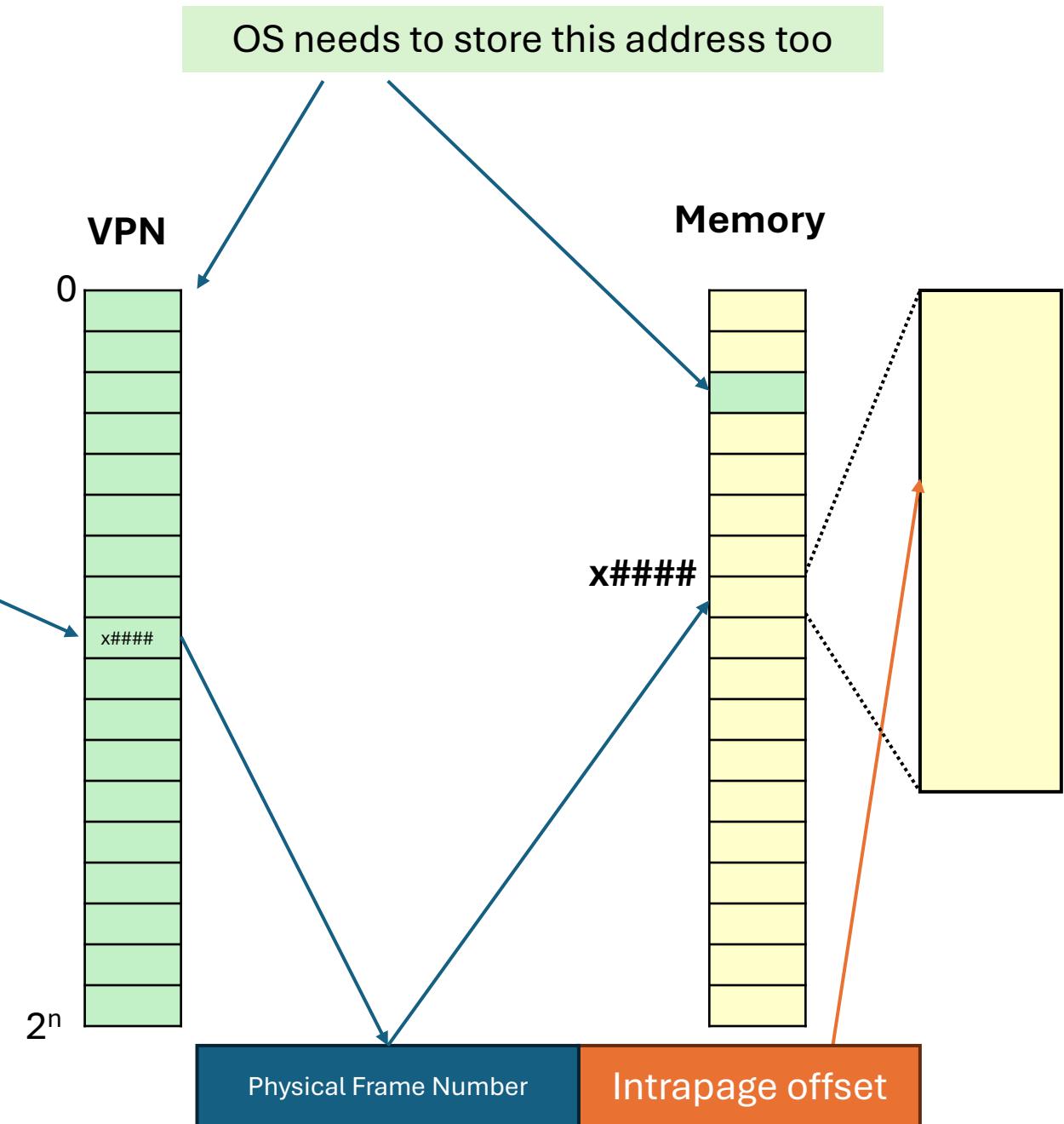
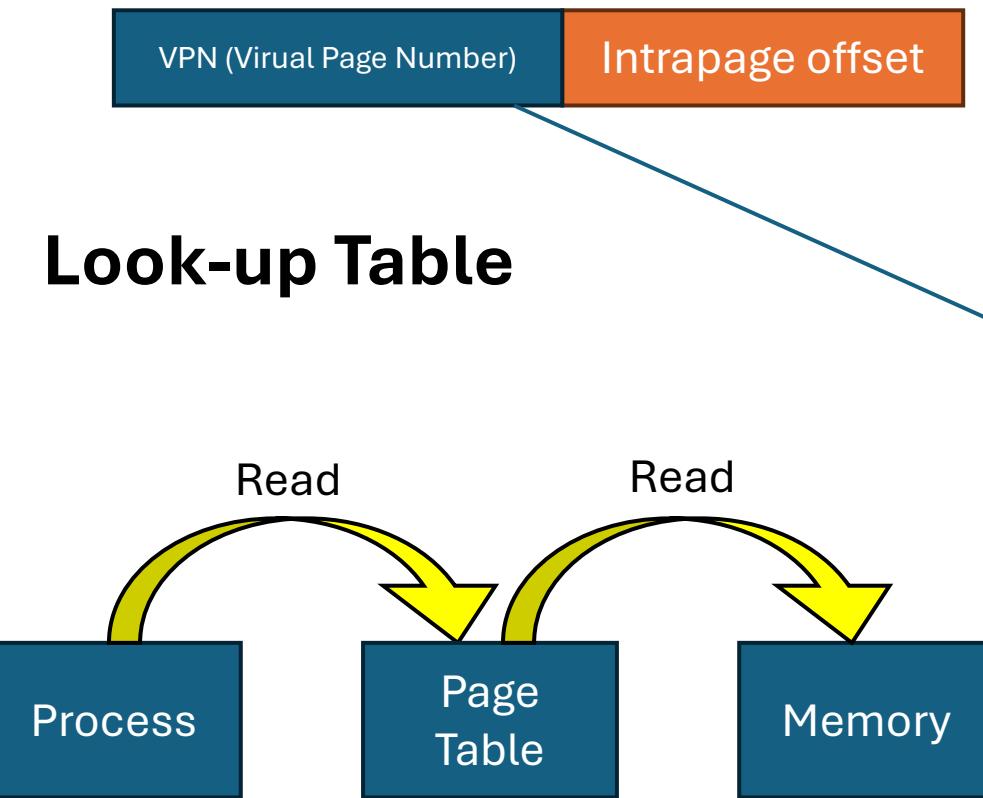
100 processes

400 MB... for address mapping?



Using 400 MB to
'save' memory?

Paging

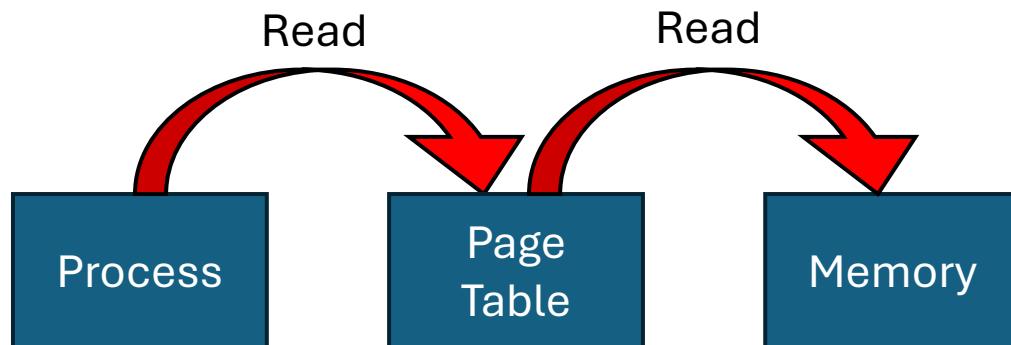


Paging: In Steps

- 1: Get VPN (from a register)
- 2: Calculate address of Page Table Entry
- 3: Read Page Table Entry from memory
- 4: Extract Page Frame Number
- 5: Build the Physical Address
- 6: Read contents of Physical Address

Paging: In Steps

- | | |
|--|-------------|
| 1: Get VPN (from a register) | Fast |
| 2: Calculate address of Page Table Entry | Fast |
| 3: Read Page Table Entry from memory | Slow |
| 4: Extract Page Frame Number | Fast |
| 5: Build the Physical Address | Fast |
| 6: Read contents of Physical Address | Slow |



Paging: Summary

Advantages:

- Removes external fragmentation
- All free pages the same size (easy to manage)
- Fast to allocate and free!
- Easy to swap out! (present bit)

Disadvantages:

- Page tables are really big
- Accessing page tables is slow
- Some internal fragmentation
- MMU only stores base address of page table

Paging: Page Table Extras

What else is in the page table?

- Page validity?
- Page protection values?
- Present bit?
- Reference bit?
- Dirty bit?

} Discuss
Later

Inverted Page Tables

Thinking about it backwards?

“Inverted”

Page Table

- For each virtual address,
where is the physical address?

Inverted Table

- For each physical address,
where is the virtual address?

“Inverted”

Page Table

- For each virtual address, where is the physical address?
- We are mapping virtual addresses we aren't using.

Inverted Table

- For each physical address, where is the virtual address?
- We only care about pages which are physically mapped.
- Implemented via hash-tables

Why this is a non-solution?

What do I have?

- A virtual address

What do I need?

- A physical address

P Address	
0	
1	
2	VPN1
3	
4	
5	
6	VPN0
7	
...	

I need to **search** 😞

Segementation Paging

Breaking pages into chunks

Segmentation Paging

The Theory

- Divide address space into segments (code, heap, stack) of variable length
- Break each segment into pages (fixed size)
- Use segment bits

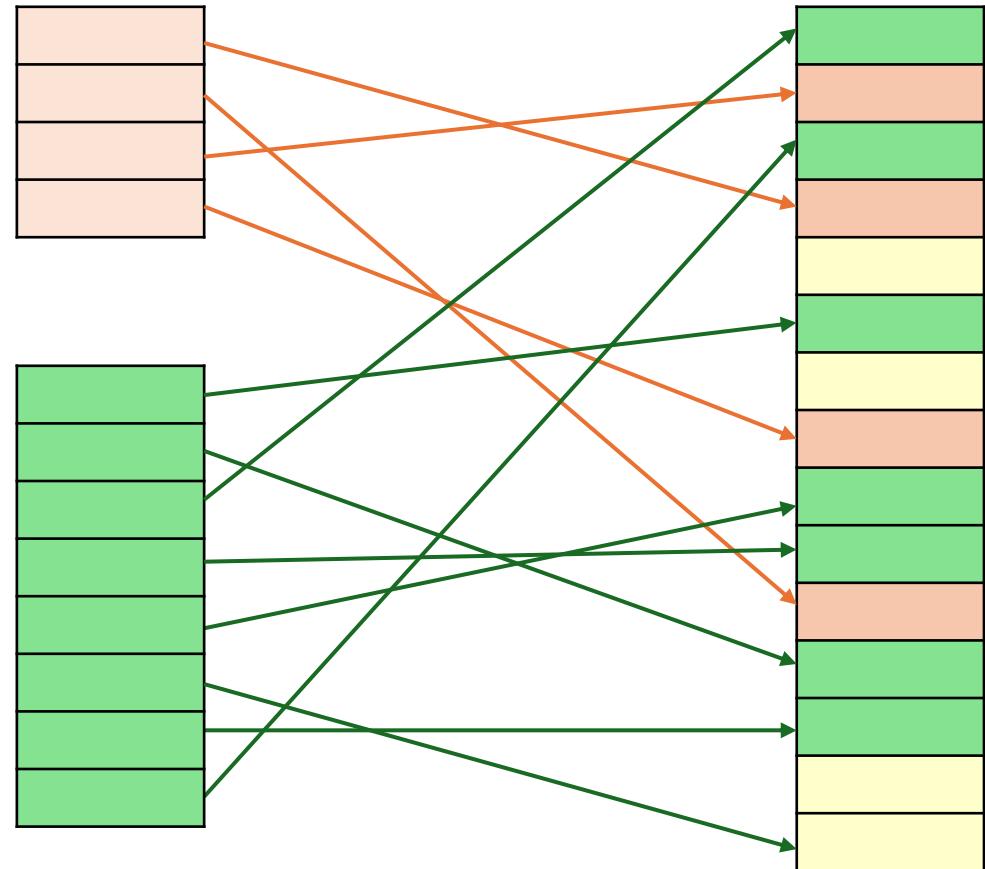


Implementation

- Each segment has a page table
- Track the base and bounds of page table for that segment

Segmentation Paging

Segment	Base	Bounds	R/W
0	0x0000	0x3FFF	1/0
1	0x2000	0x3FFF	0/0
2	0x3000	0x3FFF	1/1



Segmentation Paging

Strengths

- Segmentation
 - Sparse address spaces
 - Decreased page table size
 - If segment unused, no need for page table
- Pages
 - No external fragmentation
 - Segments can grow without shuffling memory
 - Can run process when some memory is swapped to disk

Segmentation Paging

Weaknesses

- Memory sharing is less straight-forward (need to match segments)
- We now have segmentation & paging (both require data structure support)
- Segments have size-limits (max size => more segments)
- Large page tables... (contiguous)

For segment (2 bits), page # (18 bits), offset (12 bits)

Each page table is:

$$= \# \text{entries} * \text{size}_{\text{entry}}$$

$$= \# \text{pages} * 4 \text{ bytes}$$

$$= 2^{18} * 4 \text{ bytes} = 1 \text{ MB!}$$

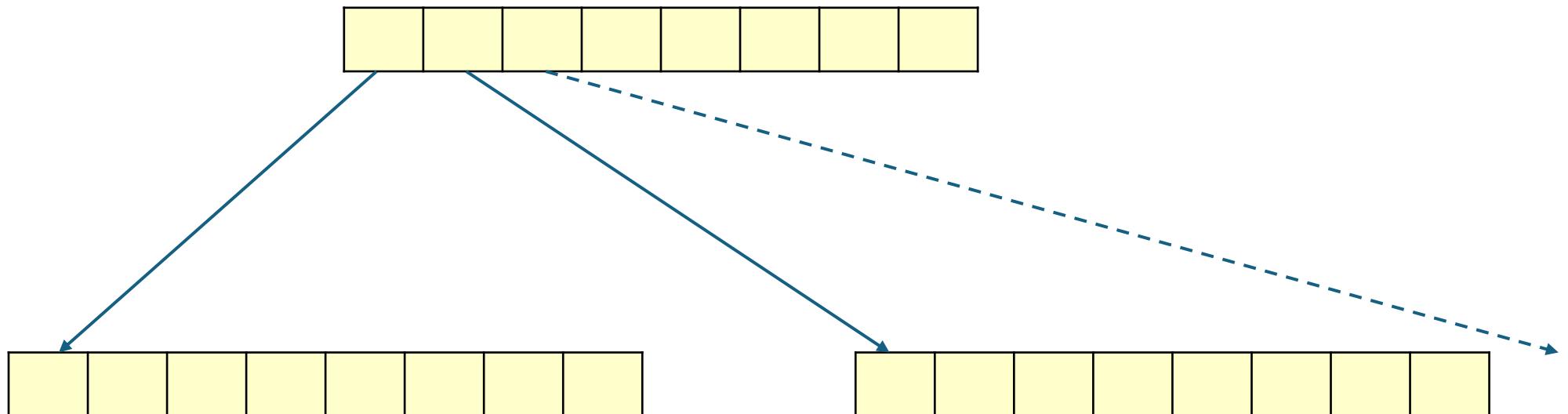
BIG

Multi-Level Paging

How to ‘tree-ify’ memory

A Page Table for a Page Table

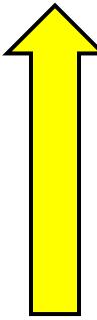
Turn our long list into a short list with another list in it...



Example: Multi-Level Page Table

Example: Multi-Level Page Table

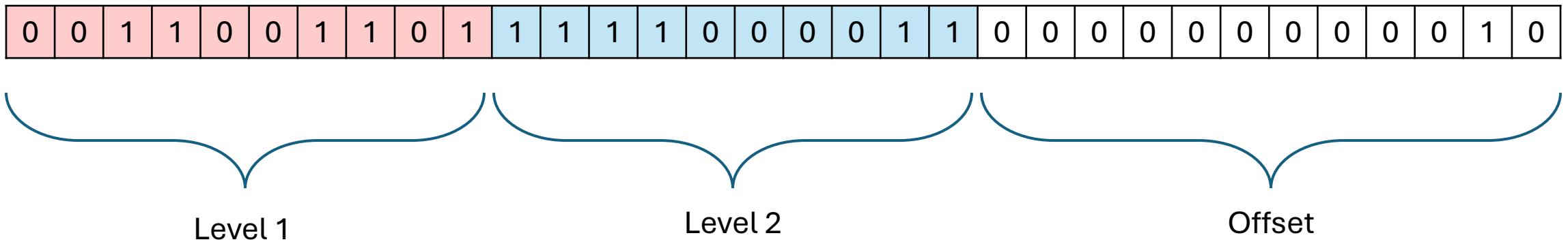
0	0	1	1	0	0	1	1	0	1	1	1	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



My process knows that 'x' is stored here...
a 'virtual address'

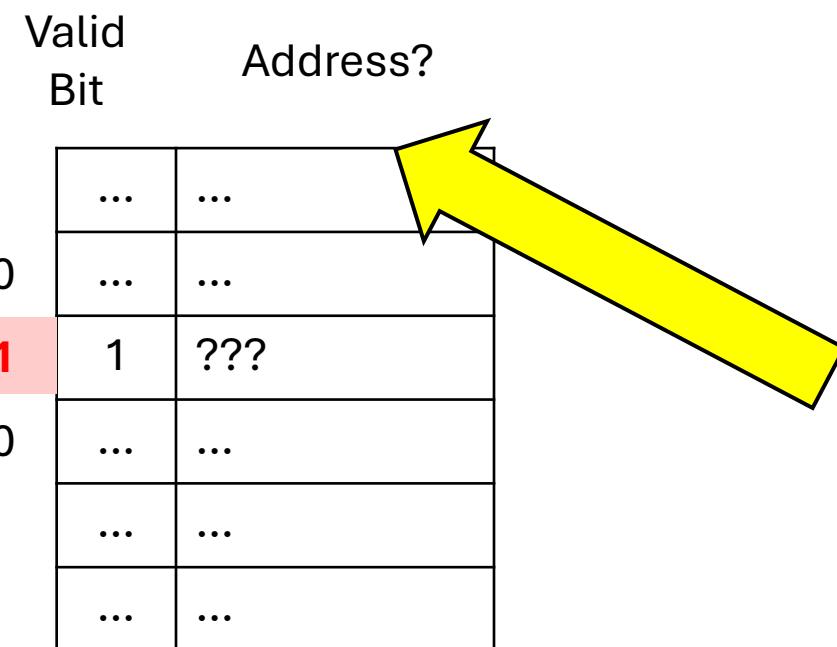
Example: Multi-Level Page Table

Example: Multi-Level Page Table



Example: Multi-Level Page Table

0	0	1	1	0	0	1	1	0	1	1	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



We know by default where this particular table is...

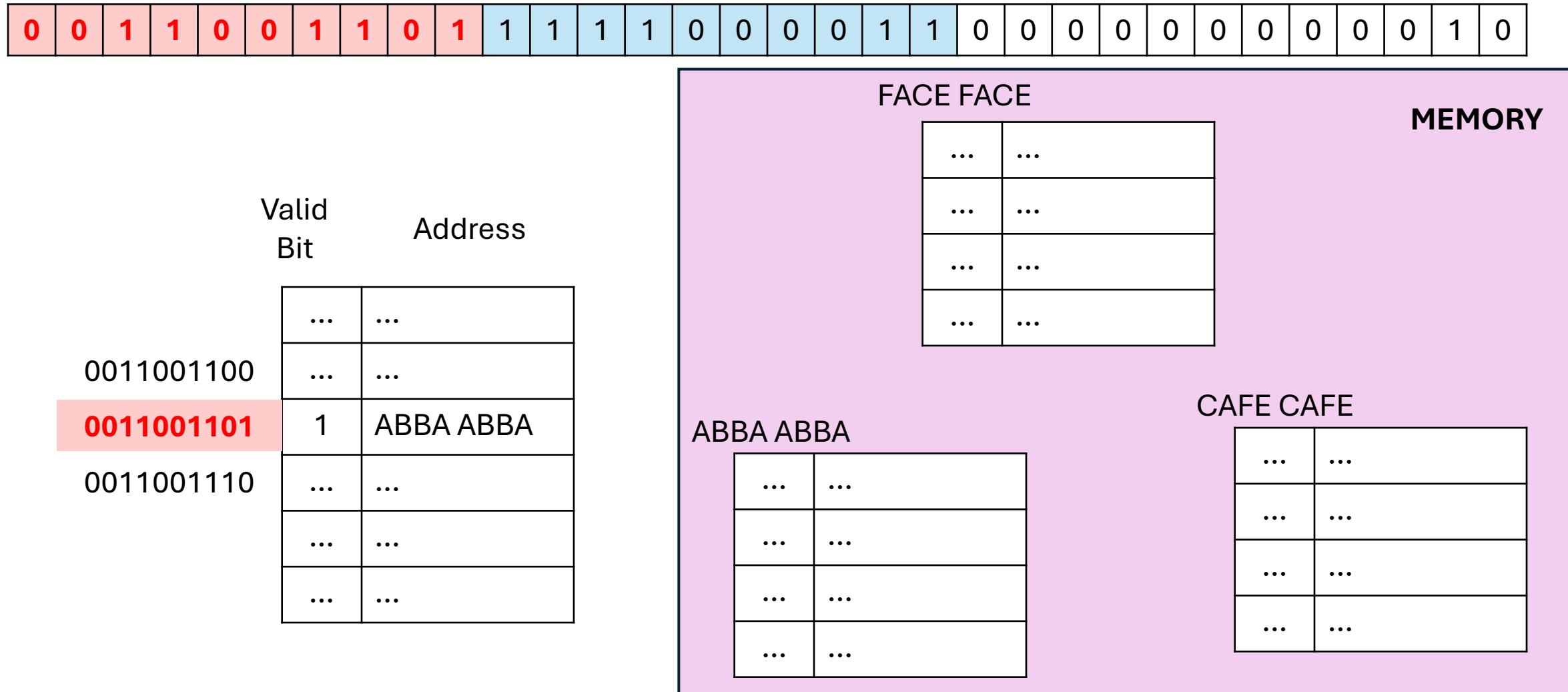
“page directory”

Example: Multi-Level Page Table

0	0	1	1	0	0	1	1	0	1	1	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

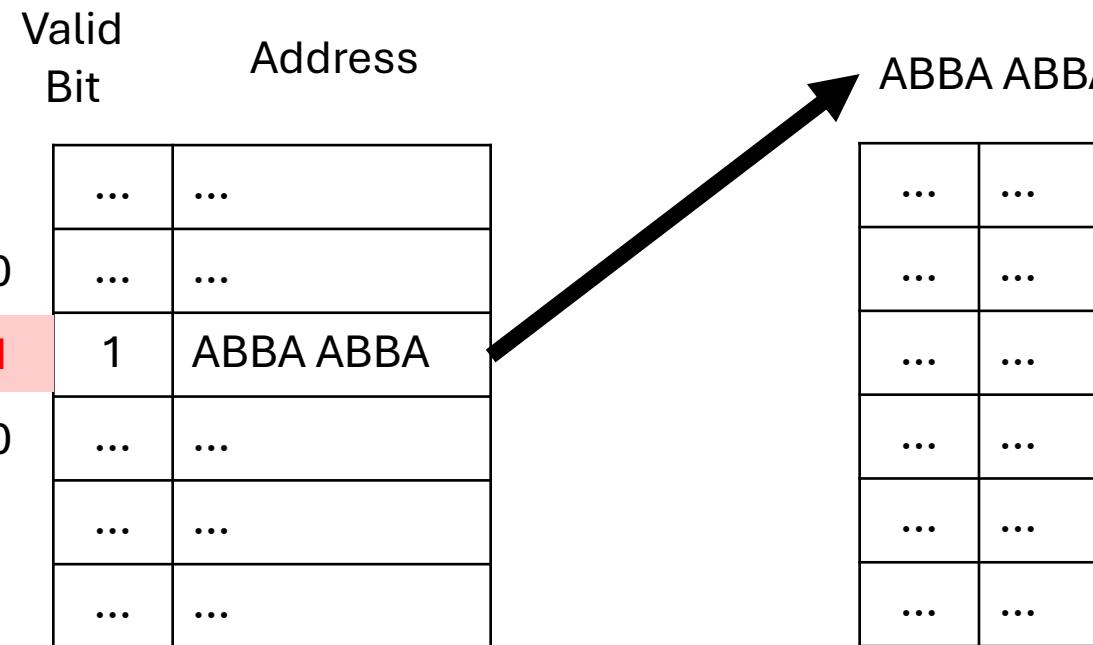
	Valid Bit	Address
0011001100
0011001101	1	ABBA ABBA
0011001110

Example: Multi-Level Page Table



Example: Multi-Level Page Table

0	0	1	1	0	0	1	1	0	1	1	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



Example: Multi-Level Page Table

0	0	1	1	0	0	1	1	0	1	1	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Valid Bit Address

0011001100
0011001101	1	ABBA ABBA
0011001110
...	...	
...	...	
...	...	

1111000010
1111000011	1	BEEF BEEF
1111000100
...	...	
...	...	
...	...	

ABBA ABBA

Example: Multi-Level Page Table



Valid
Bit

Address

0011001100

0011001101

0011001110

...	...
...	...
1	ABBA ABBA
...	...
...	...
...	...

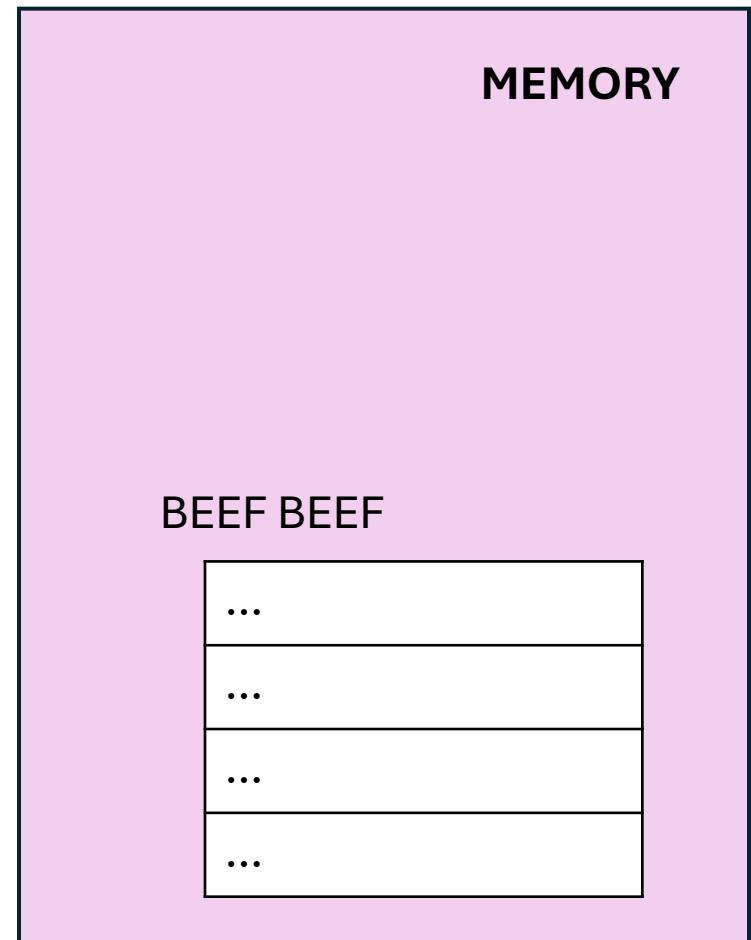
1111000010
1111000011

1111000100

...	...
...	...
1	BEEF BEEF
...	...
...	...

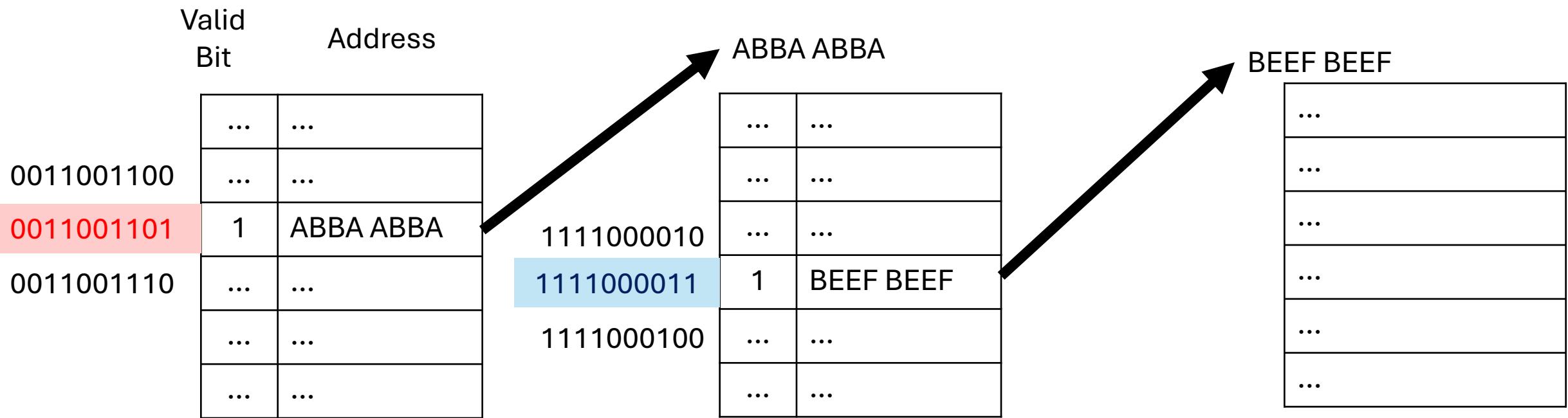
ABBA ABBA

BEEF BEEF

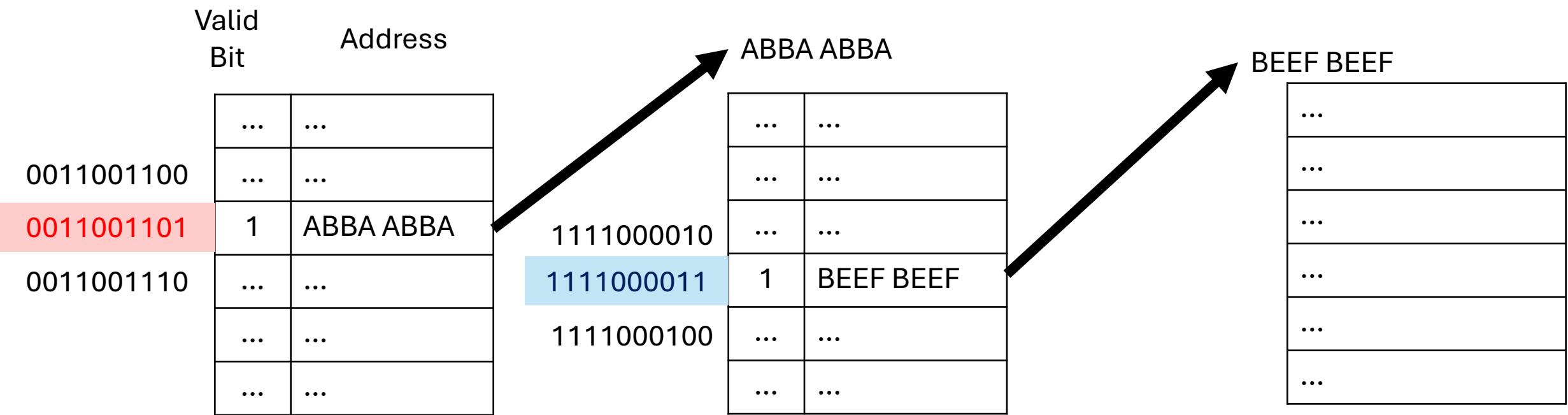


Example: Multi-Level Page Table

0	0	1	1	0	0	1	1	0	1	1	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



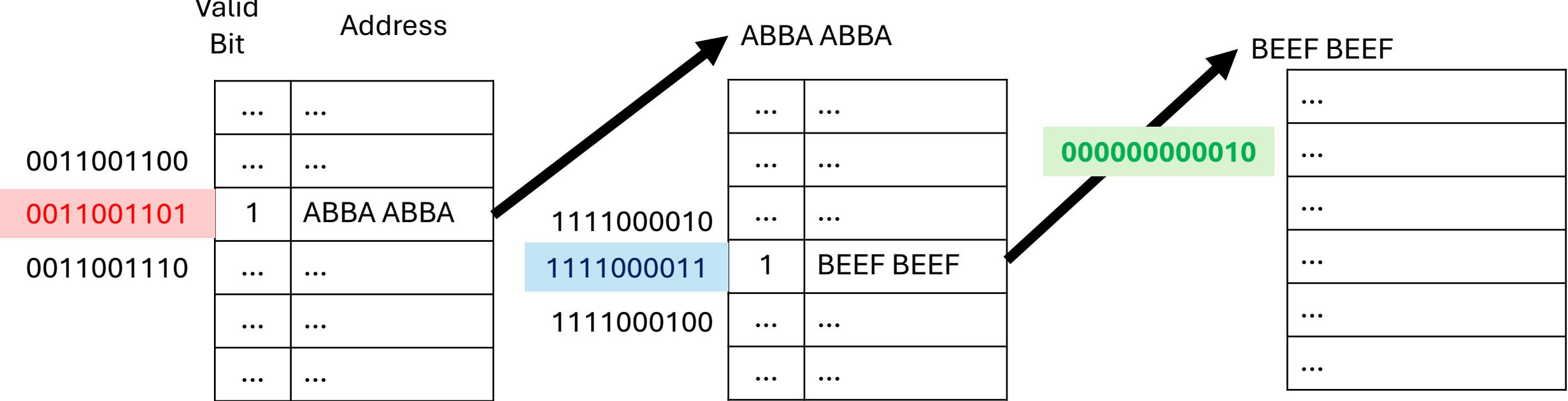
Example: Multi-Level Page Table



Example: Multi-Level Page Table

0	0	1	1	0	0	1	1	0	1	1	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

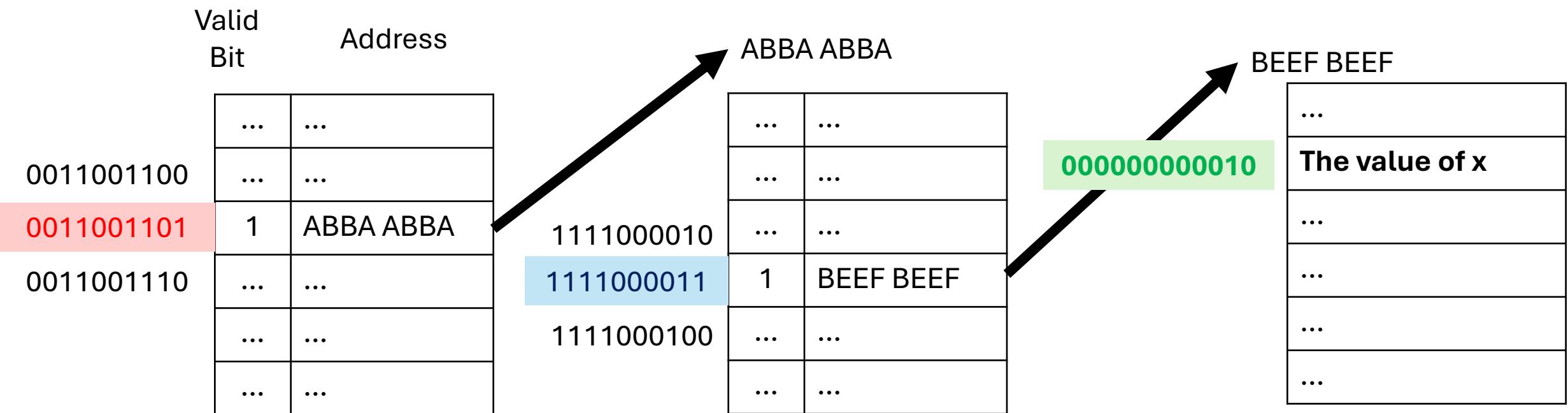
Valid
Bit Address



Example: Multi-Level Page Table

0	0	1	1	0	0	1	1	0	1	1	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

The virtual address of x



64-bit Implementation

4 KB pages... = 12 bits

52 bits??

Each page table needs to fit inside a page...

The page tables
may end up being
enormous anyway?

Or very deep...

Or we don't
actually use the
address space

Page Table Summary

Problems:

- Linear tables are too big
- OS-handling
 - Up to you
- Hardware-handling
 - Multi-level page tables (VAX, x86 etc.)
 - Each page table fits within a page

Summary

Virtualising Memory:

- Address Translation
- Base and Bounds
- Segmentation
- Page Tables

Questions?

