CS162
Operating Systems and Systems Programming
Lecture 14

Virtual Memory

Professor Natacha Crooks & Matei Zaharia

https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi

Topic Breakdown

Virtualised!

Virtualizing Memory

- Virtual Memory
- Paging

Persistence

- IO devices
- File Systems

Distributed Systems

- Challenges with distribution
- Data Processing & Storage

Recall: A process

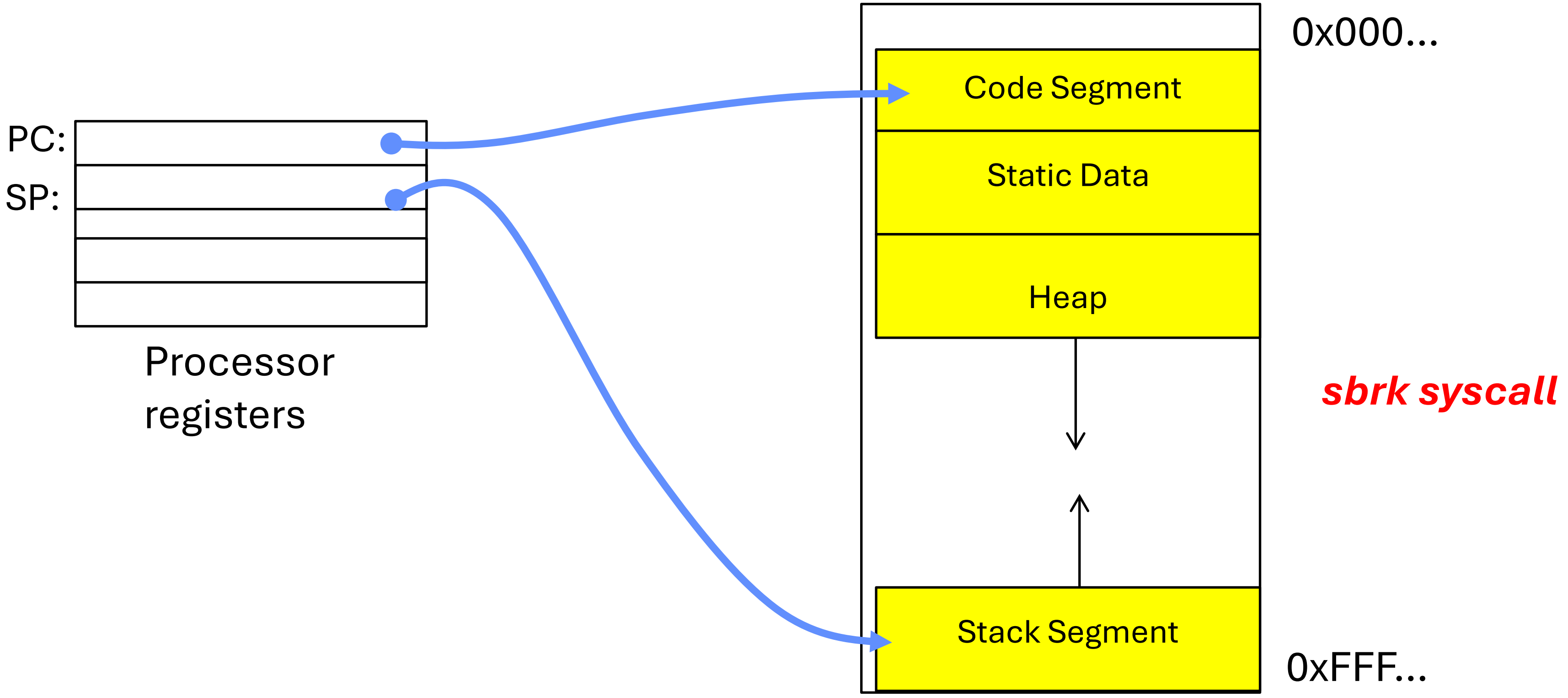
A process is an instance of a running program

CPU	Memory (address space)	Registers	IO information
	Store code, data, stack, heap	Program Counter, Stack Pointer Regular registers	Open files (and others)

Recall: Address Space

Set of memory addresses accessible to program

(for read or write)



sbrk syscall

Memory Virtualization Objectives

Isolation

Flexibility

Infinite Resources

How can we do so efficiently?

Interposing on Process Behaviours

OS interposes on process’s IO operations

via Syscalls

OS interposes on process’s CPU usage

Via Preemption

How can OS interpose on process’s memory access?

Too slow for the OS to interpose every memory access.

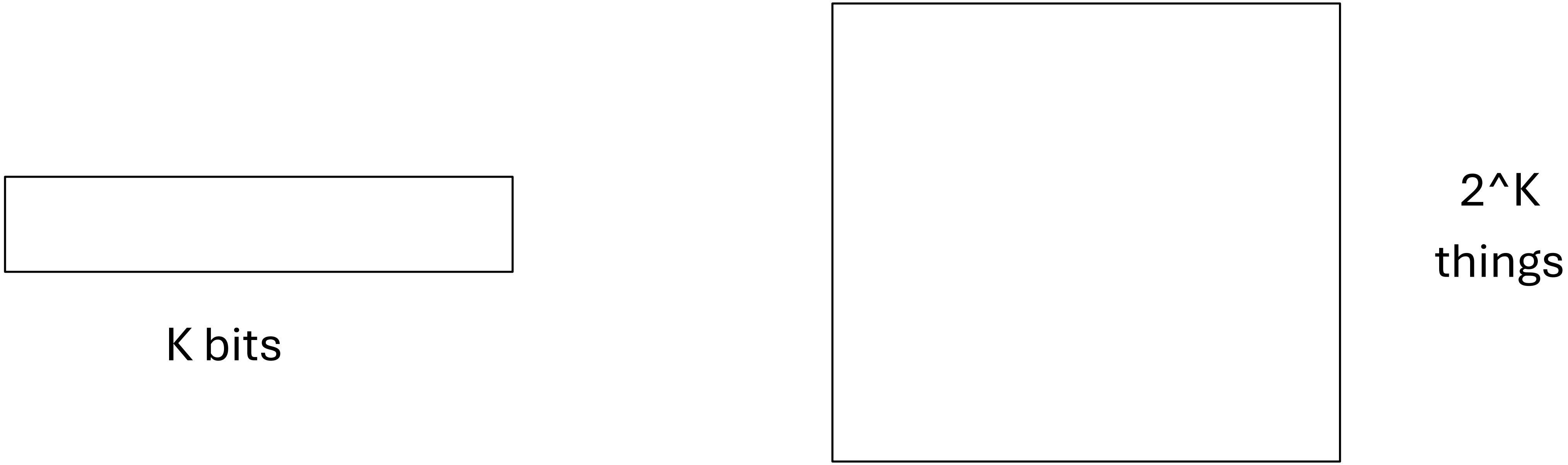
Translation: hardware support to accelerate common case.

Uncommon cases “trap” into the OS to handle

An Address

A memory address refers to the location of a byte in memory.

Most machines are byte-addressable



K bits

2^K things

Bits & Addresses

If an address space has 32 bits,

how many unique addresses do I have?

2^32 = (4294967296)

2^64 = more than the atoms of the universe

How many bits necessary to exclusively enumerate 4 elements?

2 bits => 2^2 = 4. => log2(4)

How many 32 bit numbers fit in a 2^32 address space?

32 bits -> 4 bytes -> 2^2.

2^32/2^2 = 2^30, 1 billion

Increasingly powerful mechanisms

No protection. Living life on the edge

Base & Bound

Base & Bound with Relocation

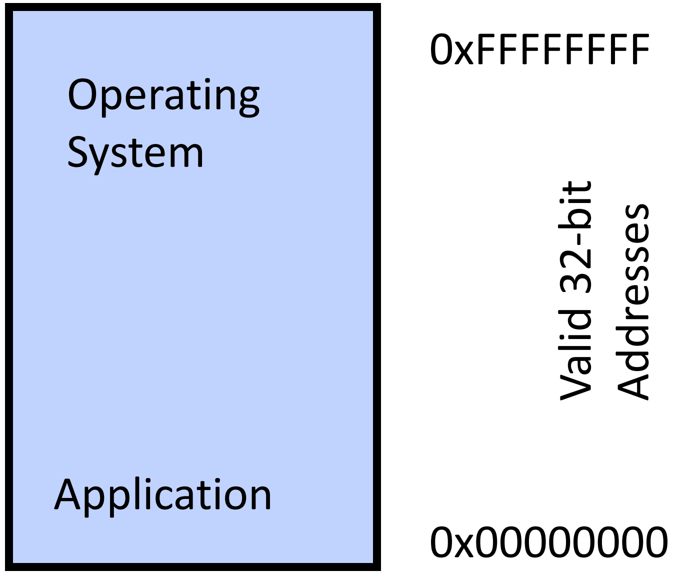
Segmentation

Paging

Uniprogramming: I’m all alone

Application always runs at same place in physical memory since only one application at a time

Application can access any physical address



Operating System

Application

0xFFFFFFFF

Valid 32-bit Addresses

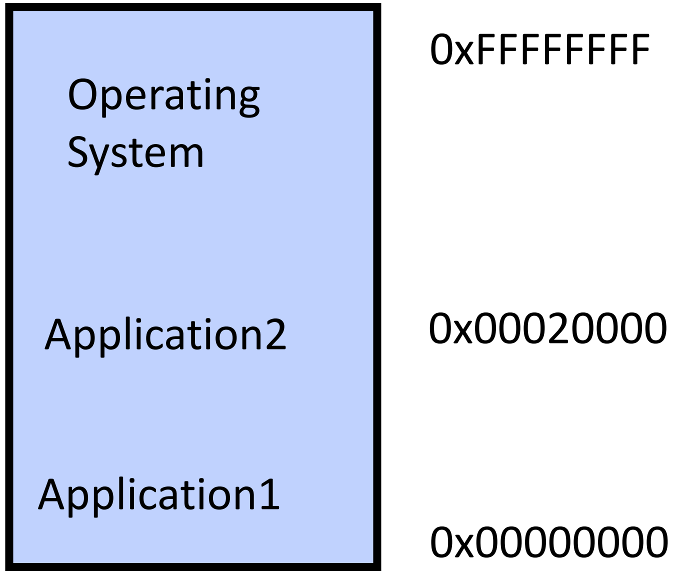
0x00000000

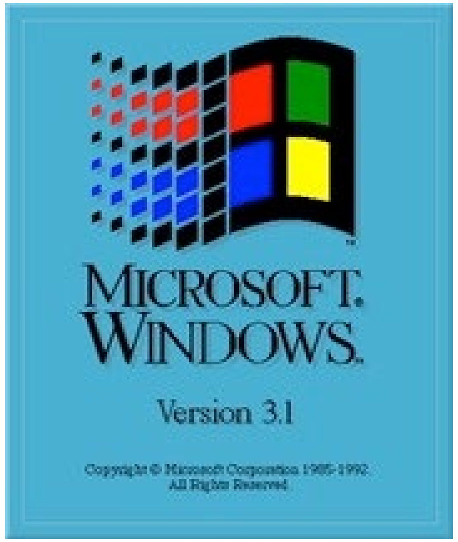
Application given illusion of dedicated machine by giving it reality of a dedicated machine

Memory Translation Through Relocation

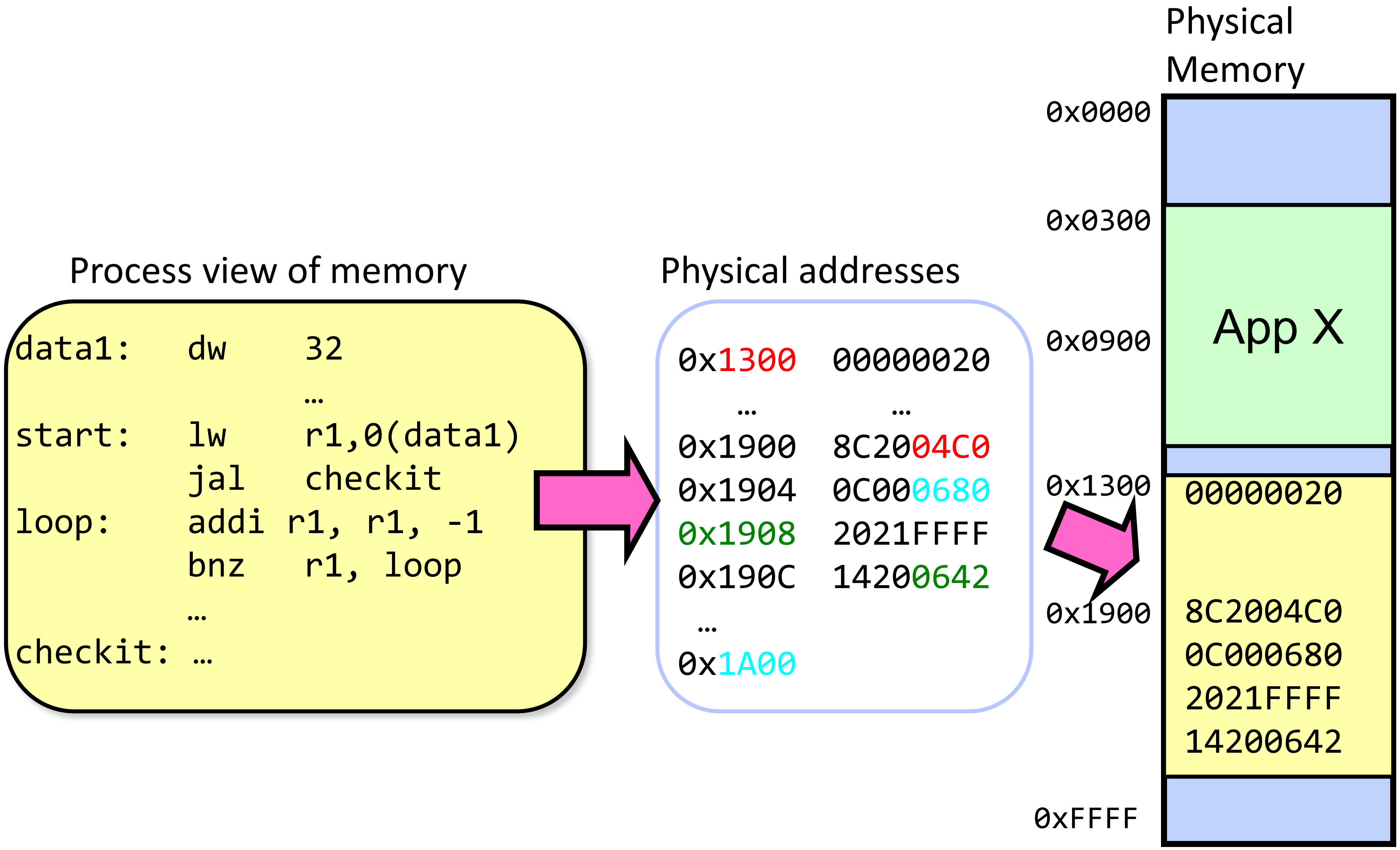
Use loader/linker to adjust addresses when program loaded into memory.

Memory Translation Through Relocation





Memory Translation Through Relocation



Process view of memory

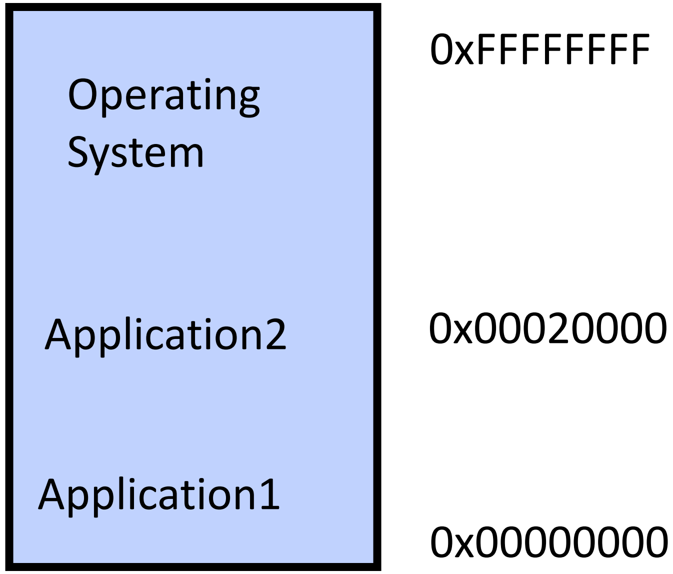
```
data1:   dw    32
               …
start:   lw    r1,0(data1)
         jal   checkit
loop:    addi r1, r1, -1
         bnz   r1, loop
         …
checkit: …
```

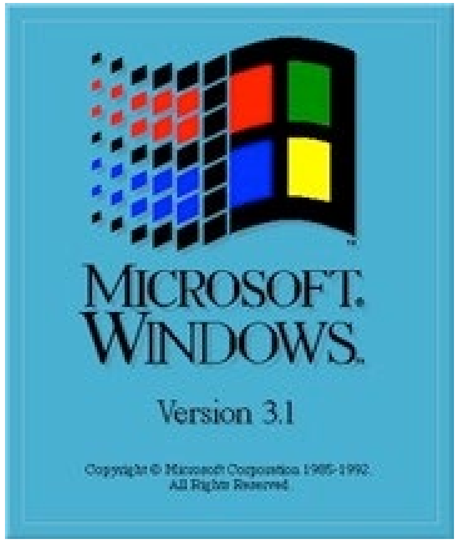
Physical addresses

```
0x1300  00000020
   …        …
0x1900  8C2004C0
0x1904  0C000680
0x1908  2021FFFF
0x190C  14200642
 …
0x1A00
```

Memory Translation Through Relocation

With this solution, no protection: bugs in any program can cause other programs to crash or even the OS





Recall: A Bug’s Tail

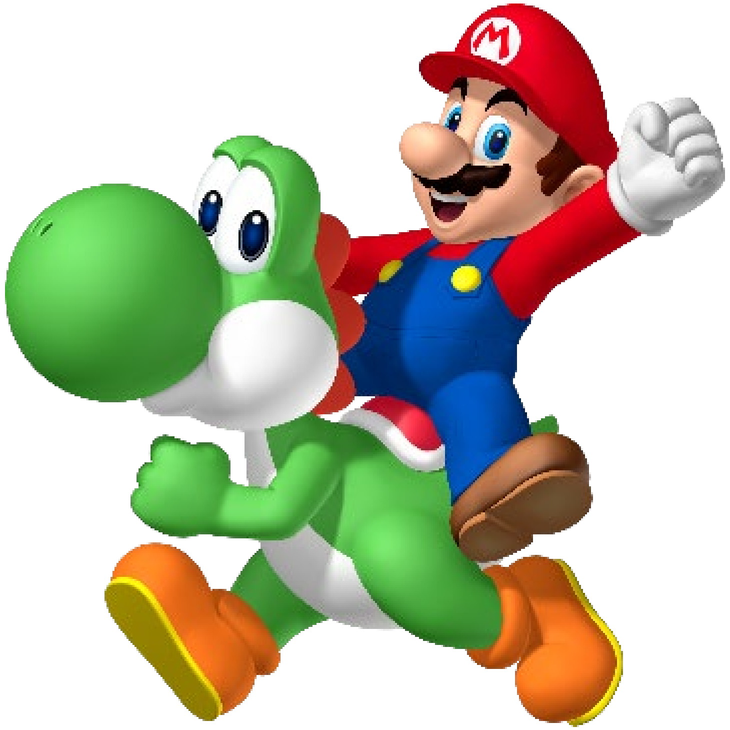
The character could leave the game area and start overwriting other running programs and kernel memory.

One of the worst bugs I ever had to deal with was in this game. Once the game player made it to the Colony, every so often the system would crash and burn at totally random times. You might be playing for ten minutes when it happened or ten hours, but it would just die in a totally random way

There was a slow-moving slug like creature that knew how to follow the game player’s trail. When it came across another creature, rather than bouncing off and risk losing the trail, I made it so that it would destroy the other creature and stay on target to find you. This worked great, except that on some rare occasions, this slug could do to a wall what it did to the other creatures. That is, it could delete it. This meant that the virtual door was now open for this creature to explore the rest of the RAM on the Macintosh, deleting and modifying it as it went along. Of course, it was just a matter of time before it found some juicy code. In other words, the bug was a REAL bug.

Recall: Super Mario Land 2

Mario could exit a level and explore the entire memory of the system



Increasingly powerful mechanisms

No protection. Living life on the edge

Base & Bound

Base & Bound with Relocation

Segmentation

Paging

Recall: Memory Protection

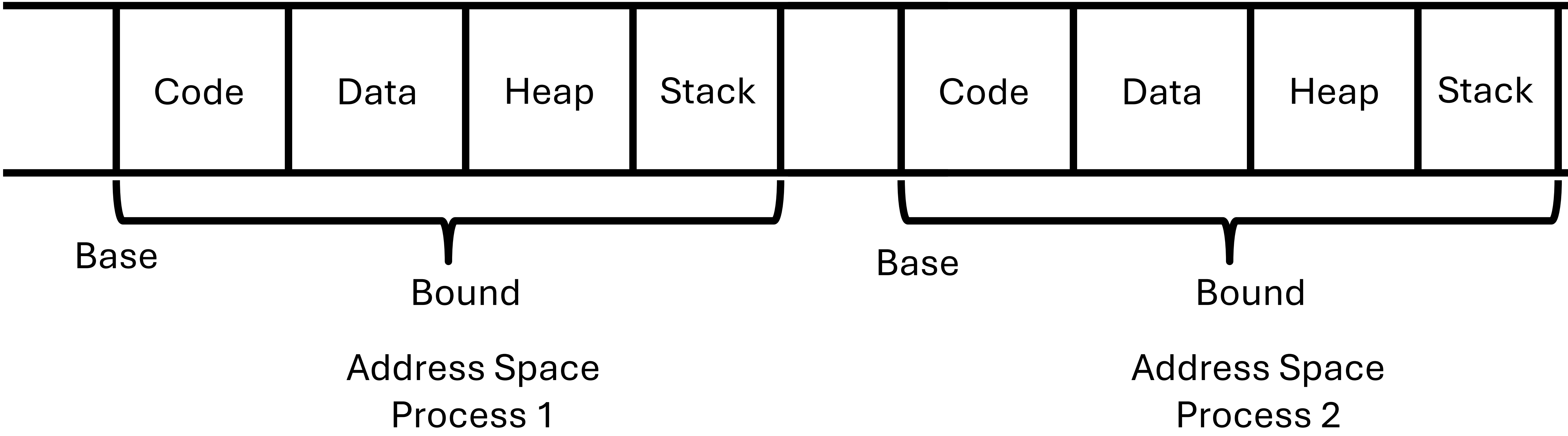
OS and applications both resident in memory

Application should not read/write kernel memory
(or other apps memory)

Base And Bound

Hardware to the rescue!

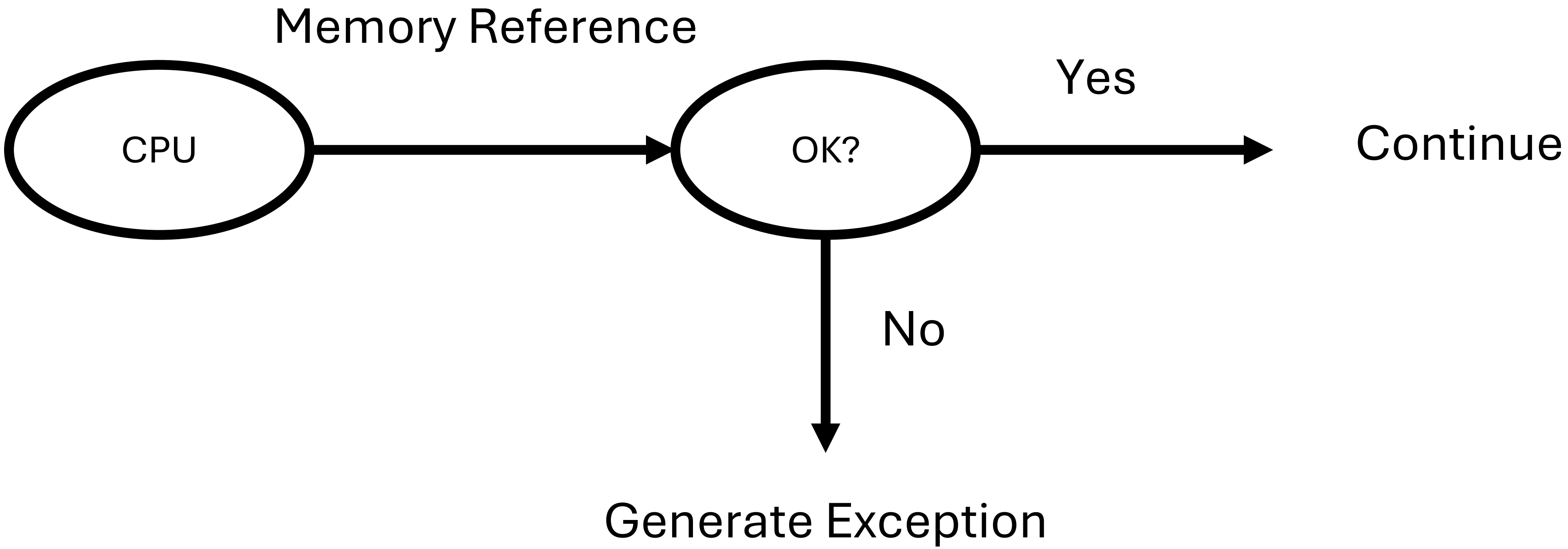
Base and Bound registers



Base & Bound

Hardware to the rescue!

Base and Bound registers



Base & Bound

Kernel Mode executes without
Base and Bound registers

Loader rewrites address to the desired offset in physical memory.

Software Relocation

```
movl 1000, %eax
```

```
movl 4000, %eax
```

Limitations of Base & Bound

1) No expandable memory

Static memory allocation

2) No memory Sharing

Cannot share memory between processes

3) Non-Relative Memory Addresses

Location of code & data determined at runtime

4) External Fragmentation

Cannot relocate/move programs. Leads to fragmentation

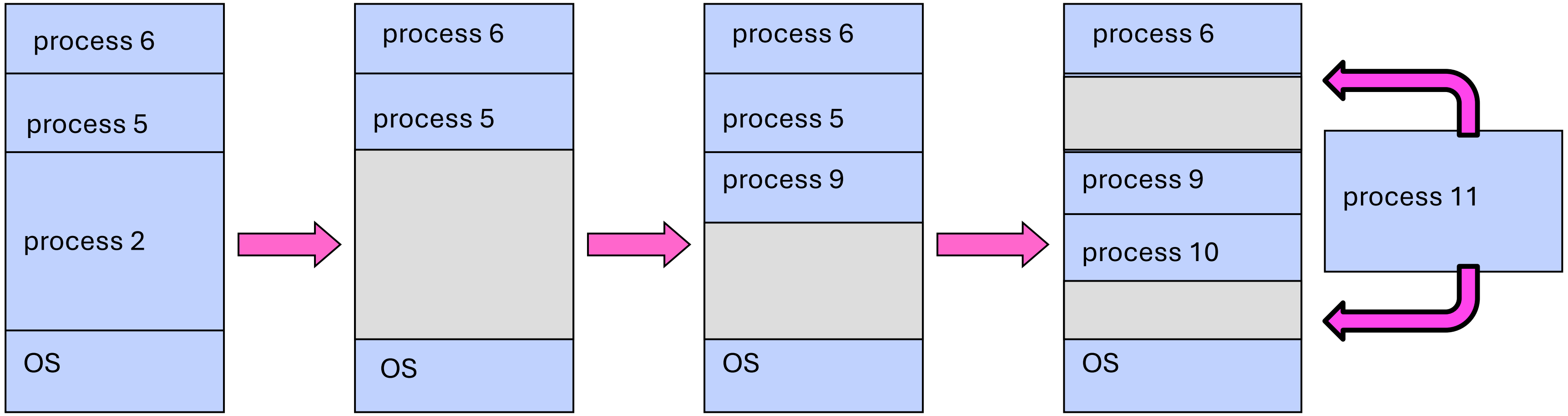
5) Internal Fragmentation

Address Space must be contiguous

Fragmentation in More Detail

External Fragmentation

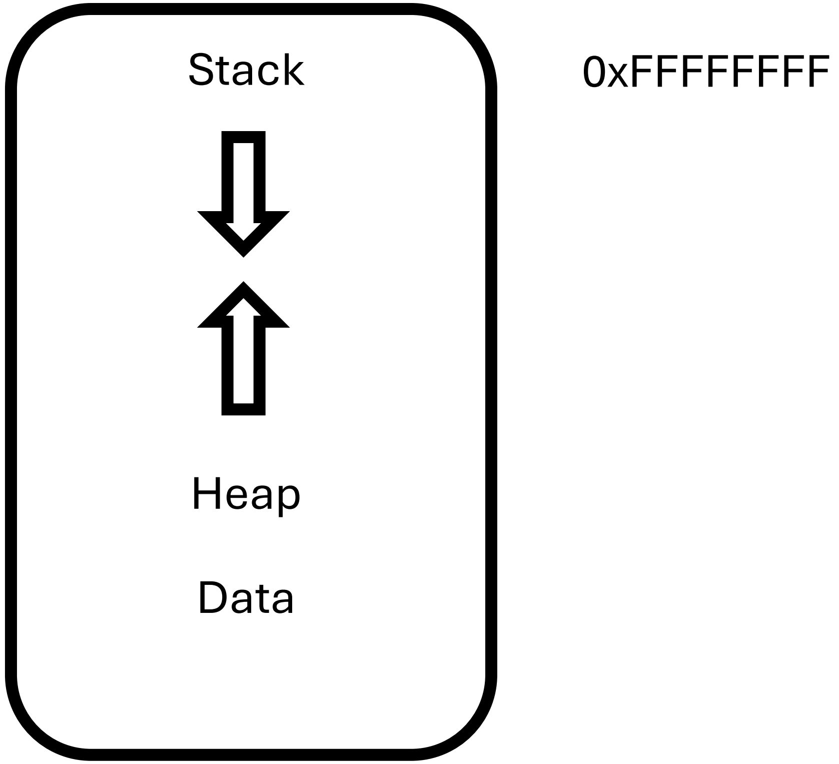
Free chunks between allocated regions



Fragmentation in More Detail

Internal Fragmentation

Space inside allocated address space may not be fully used.



Limitations of Base & Bound

1) No expandable memory

Static memory allocation

2) No memory Sharing

Cannot share memory between processes

3) Non-Relative Memory Addresses

Location of code & data determined at runtime

4) External Fragmentation

Cannot relocate/move programs. Leads to fragmentation

5) Internal Fragmentation

Address Space must be contiguous

Increasingly powerful mechanisms

No protection. Living life on the edge

Base & Bound

Base & Bound with Relocation

Segmentation

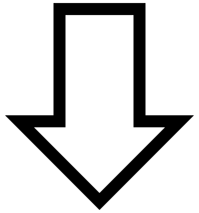
Paging

Base & Bound With Hardware Relocation

Address Translation

Virtual address space

Set of memory addresses that process can “touch”



Physical address space

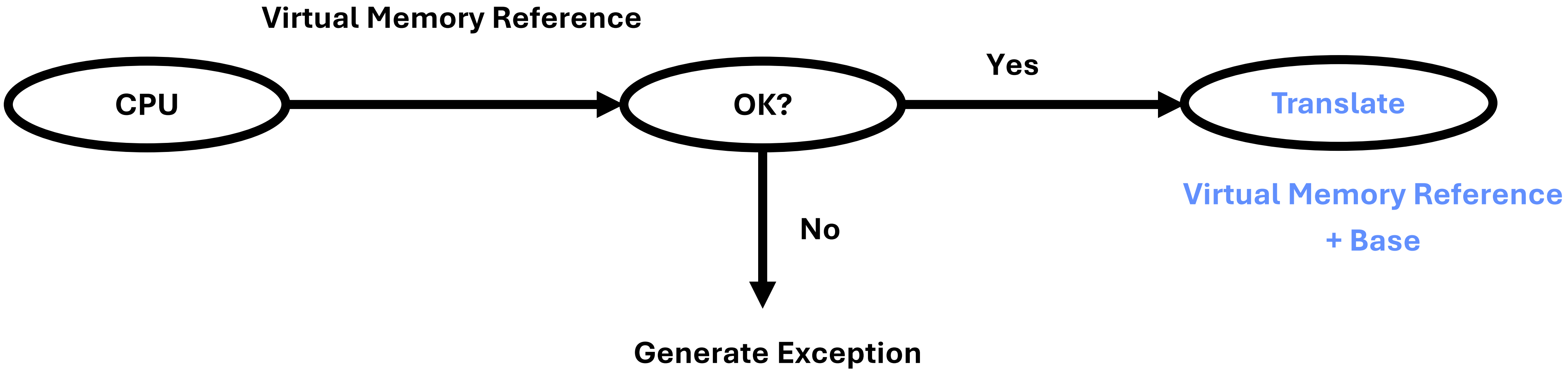
Set of memory addresses supported by hardware

Base And Bound With Relocation

Each program is written and compiled as
if it is loaded at address zero

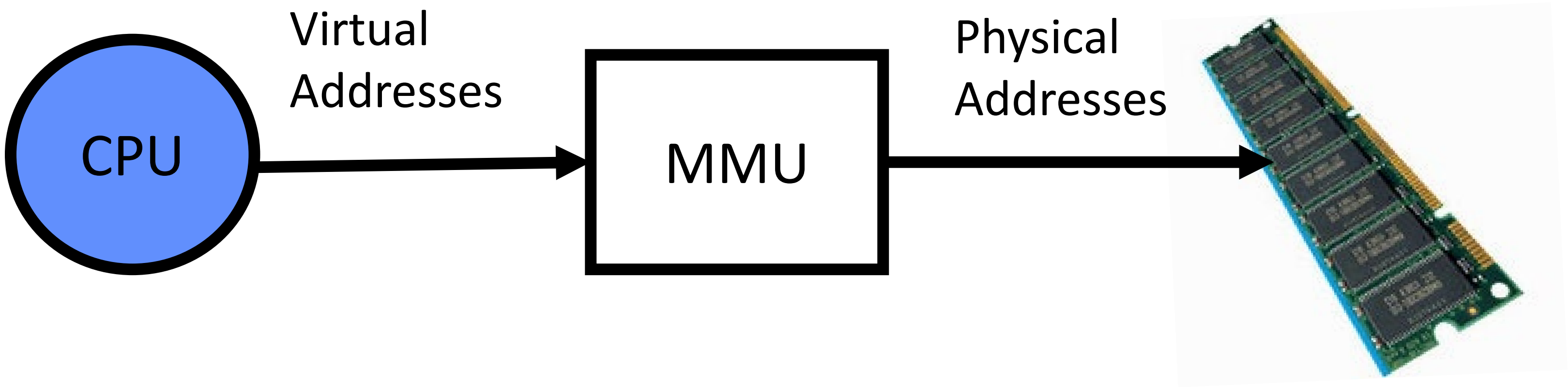
Memory references are **translated** by the processor

`physical address = virtual address + base`



Memory Management Unit

Hardware that performs translation of virtual to physical addresses



CPU

Virtual Addresses

MMU

Physical Addresses

Limitations of Base & Bound with Relocation

1) No expandable memory

Static memory allocation ∼

2) No memory Sharing

Cannot share memory between processes

3) Non-Relative Memory Addresses ✓

Location of code & data determined at runtime

4) External Fragmentation

Cannot relocate/move programs.
Leads to fragmentation ∼

5) Internal Fragmentation

Address Space must be contiguous

Increasingly powerful mechanisms

No protection. Living life on the edge

Base & Bound

Base & Bound with Relocation

Segmentation

Paging

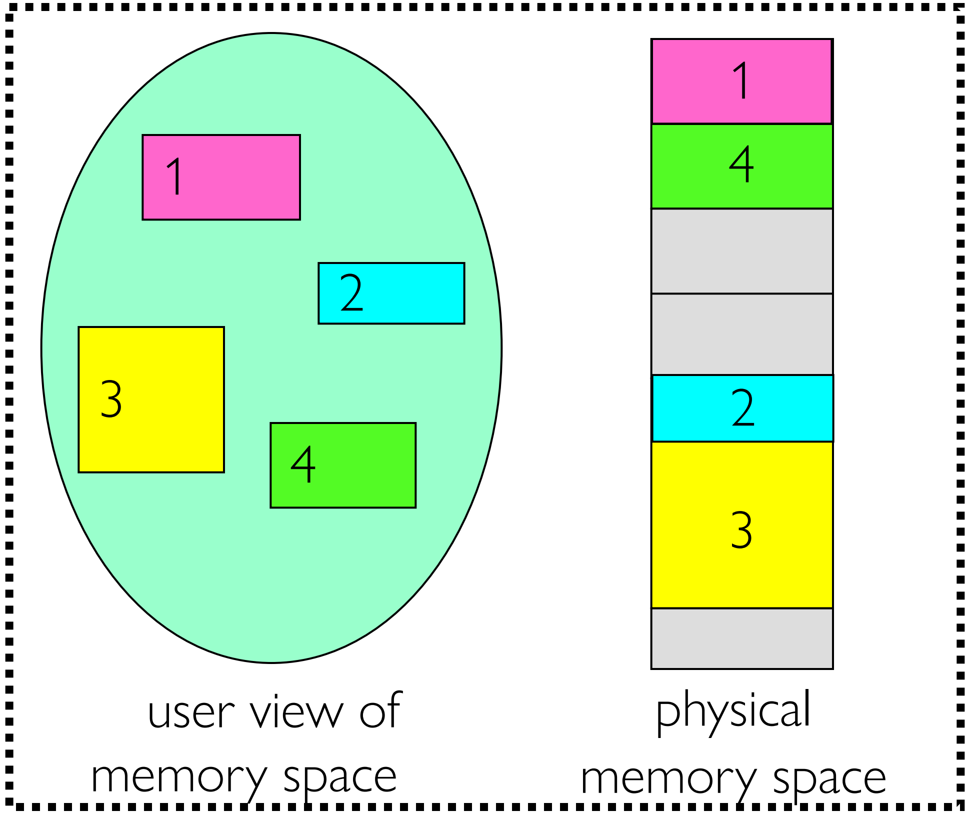
Segmentation

Create a base and bounds pair per logical segment of the address space

A segment is a contiguous portion of the address space of a particular length

Can place each segment independently at different locations in memory

Segmentation



Minimises internal fragmentation

(code, data, heap, stack segments placed independently)

Implementation of a multi-segment model

Segment map resides in processor

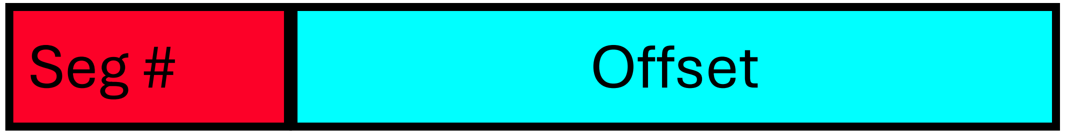
Segment number mapped into base/limit pair

Base added to offset to generate physical address

0:	Base0	Limit0
1:	Base1	Limit1
2:	Base2	Limit2
3:	Base3	Limit3
4:	Base4	Limit4
5:	Base5	Limit5
6:	Base6	Limit6
7:	Base7	Limit7
8:		

Address Translation

A logical address consists of two parts: a segment identifier (top bits) and an offset that specifies the relative address within the segment (bottom bits)



Seg #

Offset

Address Translation

Assume we have 16 bit addresses

Question: if I have 4 segments (code, data, stack, heap), how many segment bits do I need?

Log(4) = 2

Segment 0: 00
Segment 1: 01
Segment 2: 10
Segment 3: 11

Address Translation

Assume we have 16 bit addresses

Question: if I have 4 segments (code, data, stack, heap), how many segment bits do I need?

Log(4) = 2

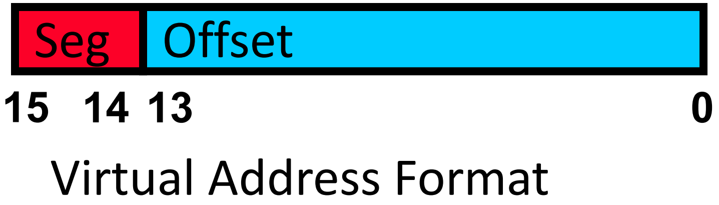
Question: what is the maximum size of each segment?

16-2 = 14 bits left. => 2^14 bytes

Question: if I have 7 segments and an address size of 32 bits, what is the maximum size of a segment?

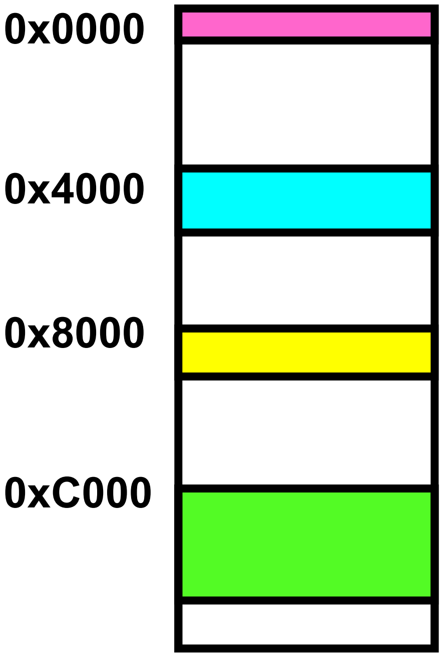
Log2(7) = 2.8 => 3 bits. 2^(32-3)=2^29

Example: Four Segments (16 bit addresses)

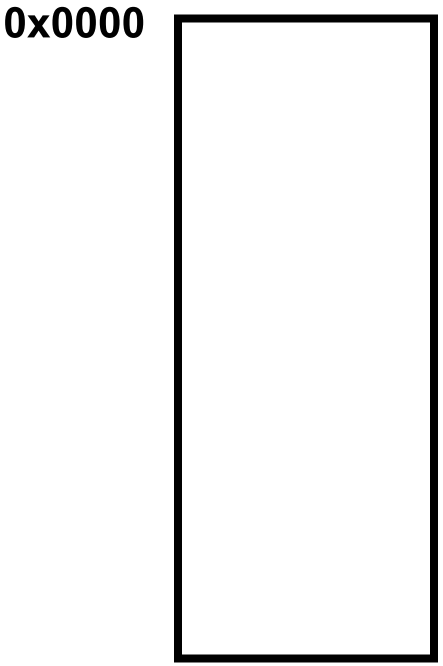


Virtual Address Format

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



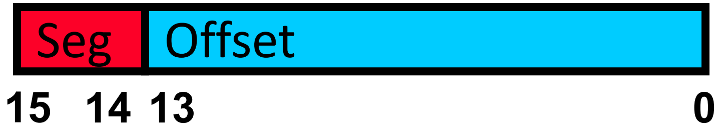
Virtual
Address Space



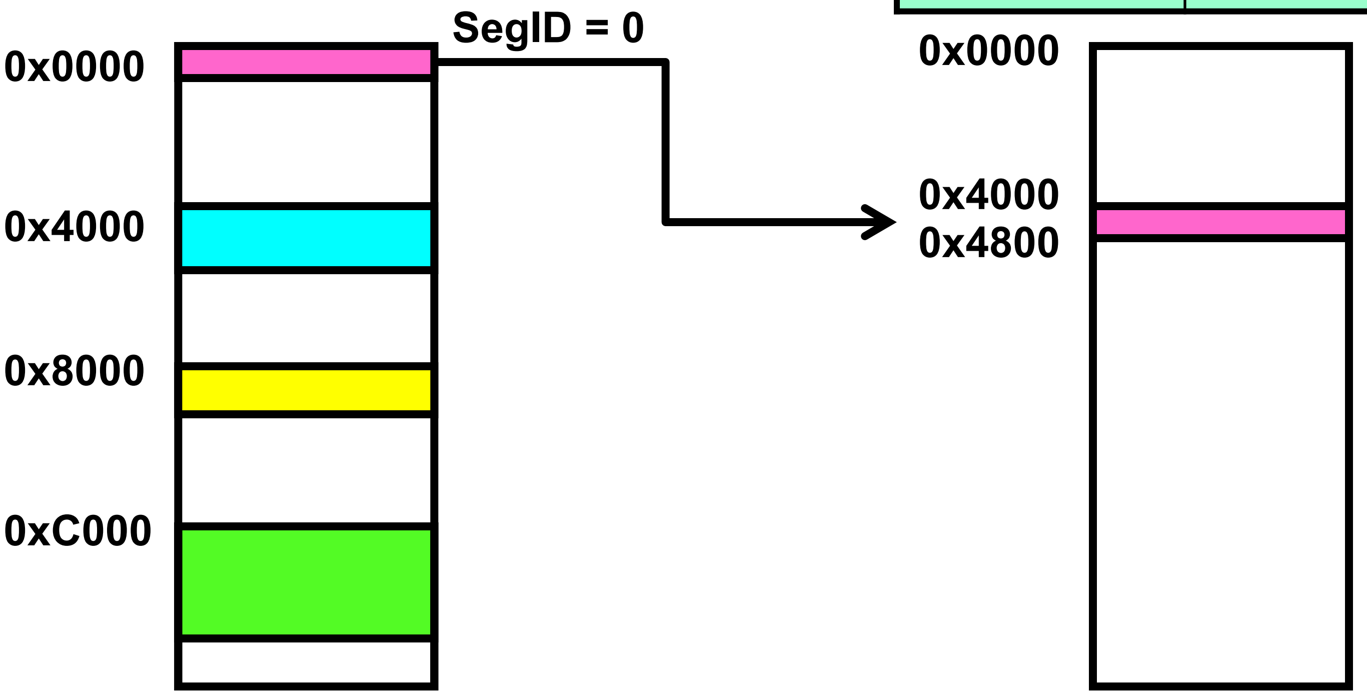
Physical
Address Space

Example: Four Segments (16 bit addresses)

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



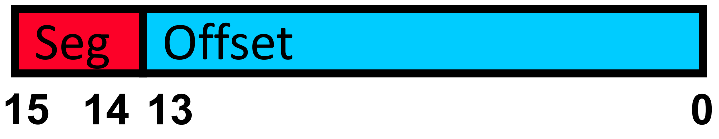
Virtual Address Format



Virtual Address Space

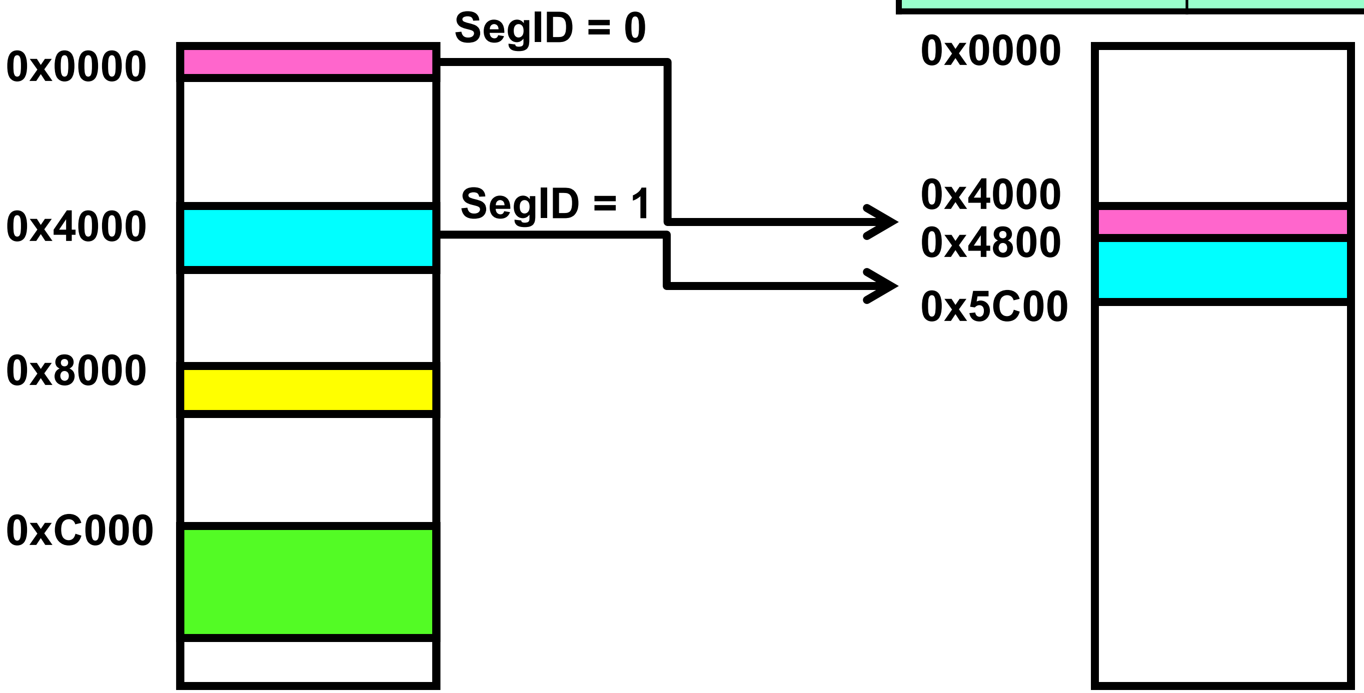
Physical Address Space

Example: Four Segments (16 bit addresses)



Virtual Address Format

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

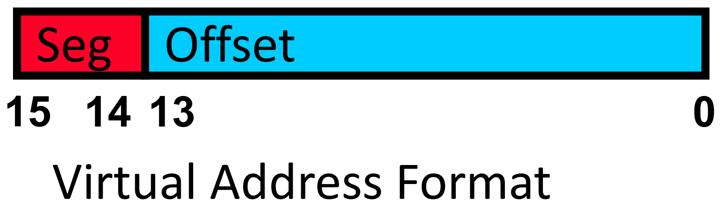


Virtual
Address Space

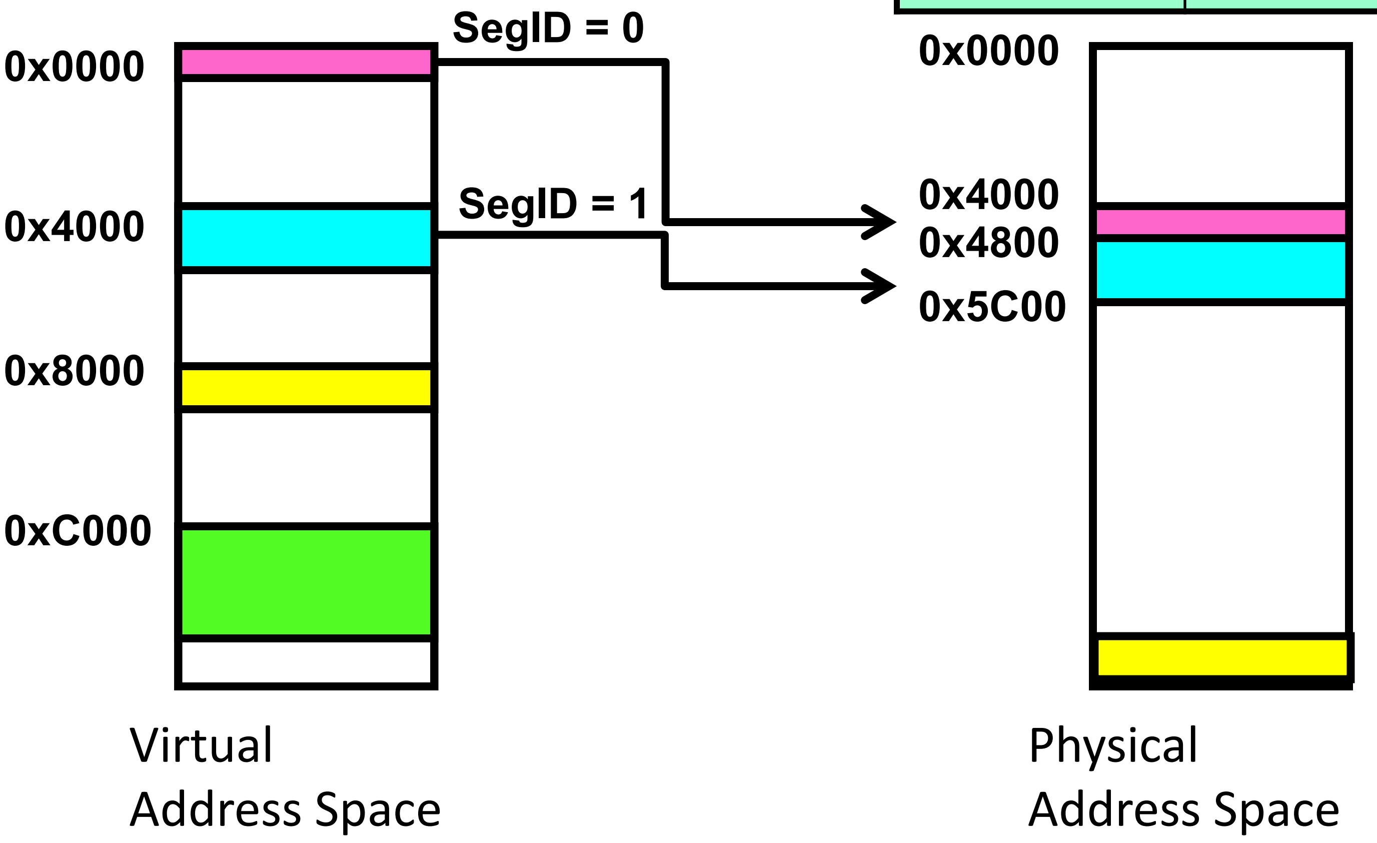
Physical
Address Space

Example: Four Segments (16 bit addresses)

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



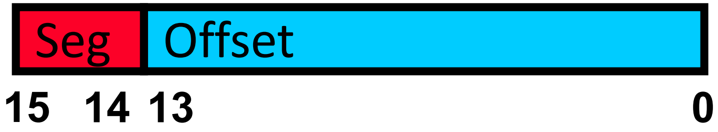
Virtual Address Format



Virtual Address Space

Physical Address Space

Example: Four Segments (16 bit addresses)



Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

0x4000	01 0	0000 0000 0000	Segment 1	Offset 0x0	0x4000
0x8020	10 0	0000 0010 0000	Segment 2	Offset 0x20	0xF020

Adding support for sharing

Useful to **share**
certain memory segments between address spaces.

Seg ID #	Base	Limit	Protection Bits
0 (code)	0x4000	0x0800	Read-Execute
1 (data)	0x4800	0x1400	Read-Write
2 (shared)	0xF000	0x1000	Read-Write
3 (stack)	0x0000	0x3000	Read-Write

Hardware must now check whether access is
1) within bounds 2) permissible

Segmentation Summary Pros

Minimal hardware requirements & efficient translation

Segmentation can better support sparse address spaces

Avoids internal fragmentation.

Minimises memory waste between logical segments of the address space

Limitations of Segmentation

1) No expandable memory

Static memory allocation ≈

2) No memory Sharing

Cannot share memory between processes ≈

3) Non-Relative Memory Address

Location of code & data determined at runtime ✓

4) External Fragmentation

Cannot relocate/move programs. Leads to fragmentation ≈

5) Internal Fragmentation

Address Space must be contiguous ≈

Segmentation Summary Cons

External fragmentation still a problem
Must fit variable-sized chunks into physical memory.

May move processes multiple times to fit everything