CS3.301 Operating Systems and Networks

Memory Virtualization - Dynamic relocation and Segmentation

Karthik Vaidhyanathan

https://karthikvaidhyanathan.com

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

SERC
Software Engineering Research Centre

Acknowledgement

The materials used in this presentation have been gathered/adapted/generate from various sources as well as based on my own experiences and knowledge -- Karthik Vaidhyanathan

Sources:

- Operating Systems: In three easy pieces, by Remzi et al.

Goals of Virtualization

- **Transparency**
    - Illusion that physical memory is not visible to any processes
    - Take away worry from the user program about what happens behind scenes
- **Efficiency**
    - Minimize overhead in terms of space and access time
- **Protection**
    - Protect process from one another even OS itself
    - Each process must be running its own isolated cocoon safe from malicious process

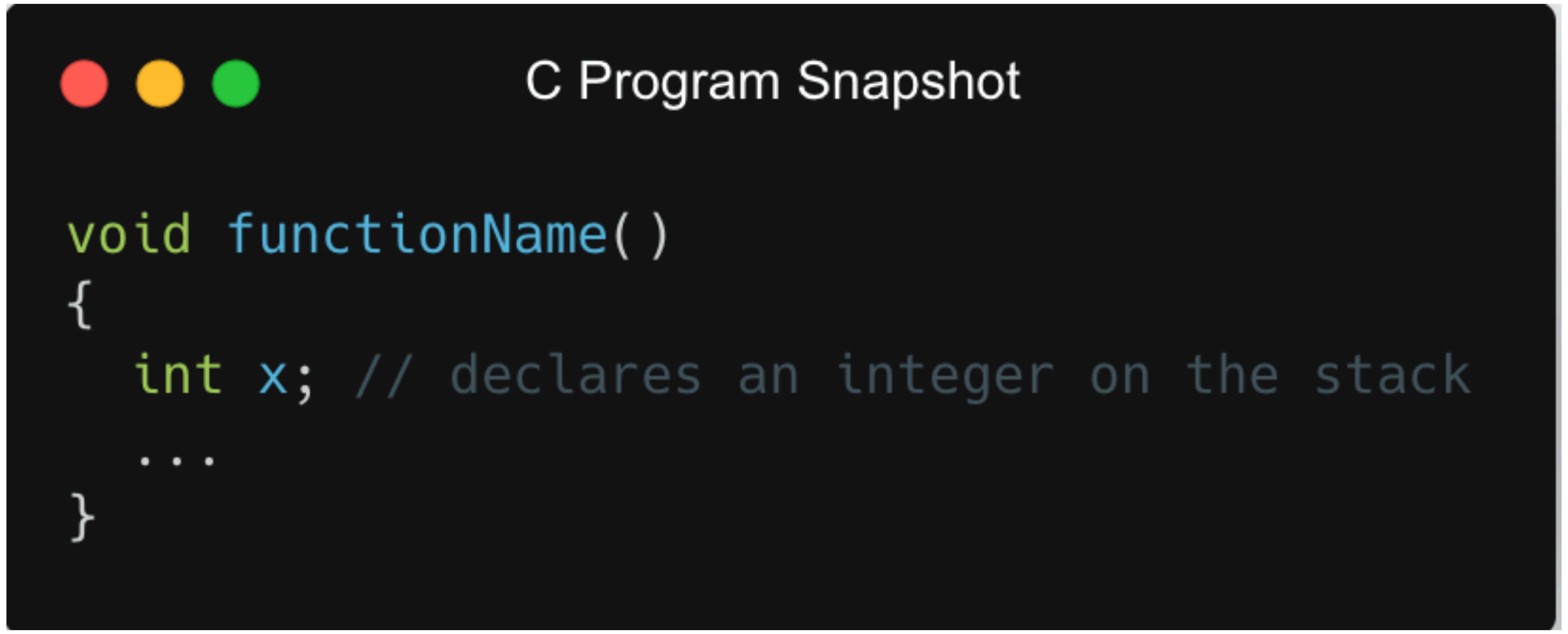
Memory API

- For process virtualization, we learned about APIs to create, destroy, duplicate processes
    - What about memory?
    - Can we think of some ways to do it?
    - What are the interfaces for it?
    - What are some common pitfalls that needs to be avoided?

Memory Allocations and Deallocations

First Type of Memory Allocation

- In C program, two types of memory allocation happens
    - Stack Memory
        - Allocations and deallocations are managed implicitly by compilers
        - Called **Automatic memory**
        - Once execution is done, compiler deallocates memory
- Static/global variables are allocated in executable

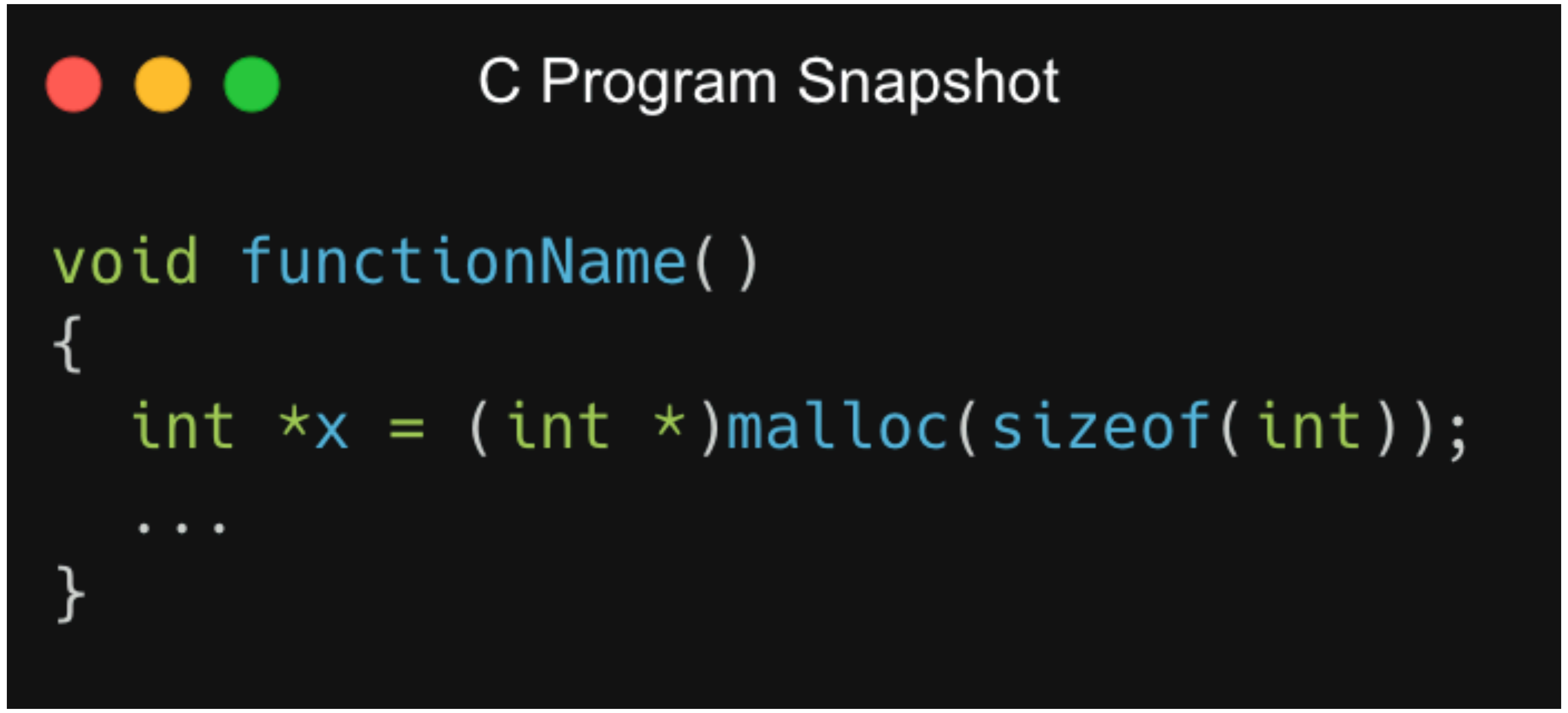


```c
void functionName( )
{
  int x; // declares an integer on the stack
  ...
}
```

Memory Allocations and Deallocations

Second type of Memory Allocation

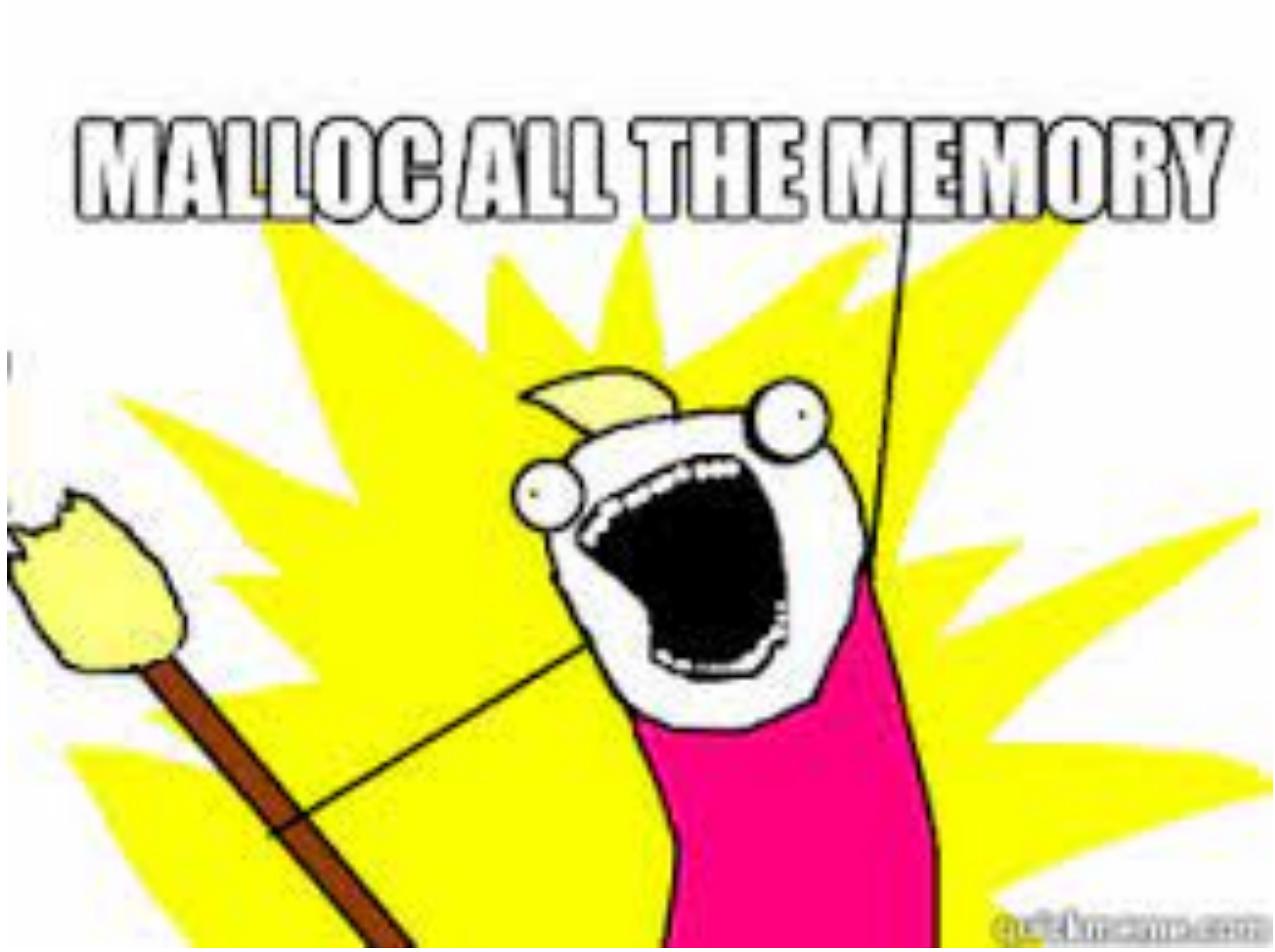
- Heap memory
    - Allocations and deallocations are handled explicitly by the programmer
    - **malloc()** requests for space of integer on the heap
    - The routine returns the address of the integer
    - Heap memory is more challenging to play with



C Program Snapshot

```
void functionName()
{
   int *x = (int *)malloc(sizeof(int));
   ...
}
```

The malloc() call



The malloc() call

- Quite a simple call
- Just pass as parameter, the size required in the heap (**size_t**) - Number of bytes
- The call will return pointer to new space
  - Returns NULL on failure
  - Under library stdlib.h
  - For allocating double precision value:

```
double *d  = (double *)malloc(sizeof (double));
```

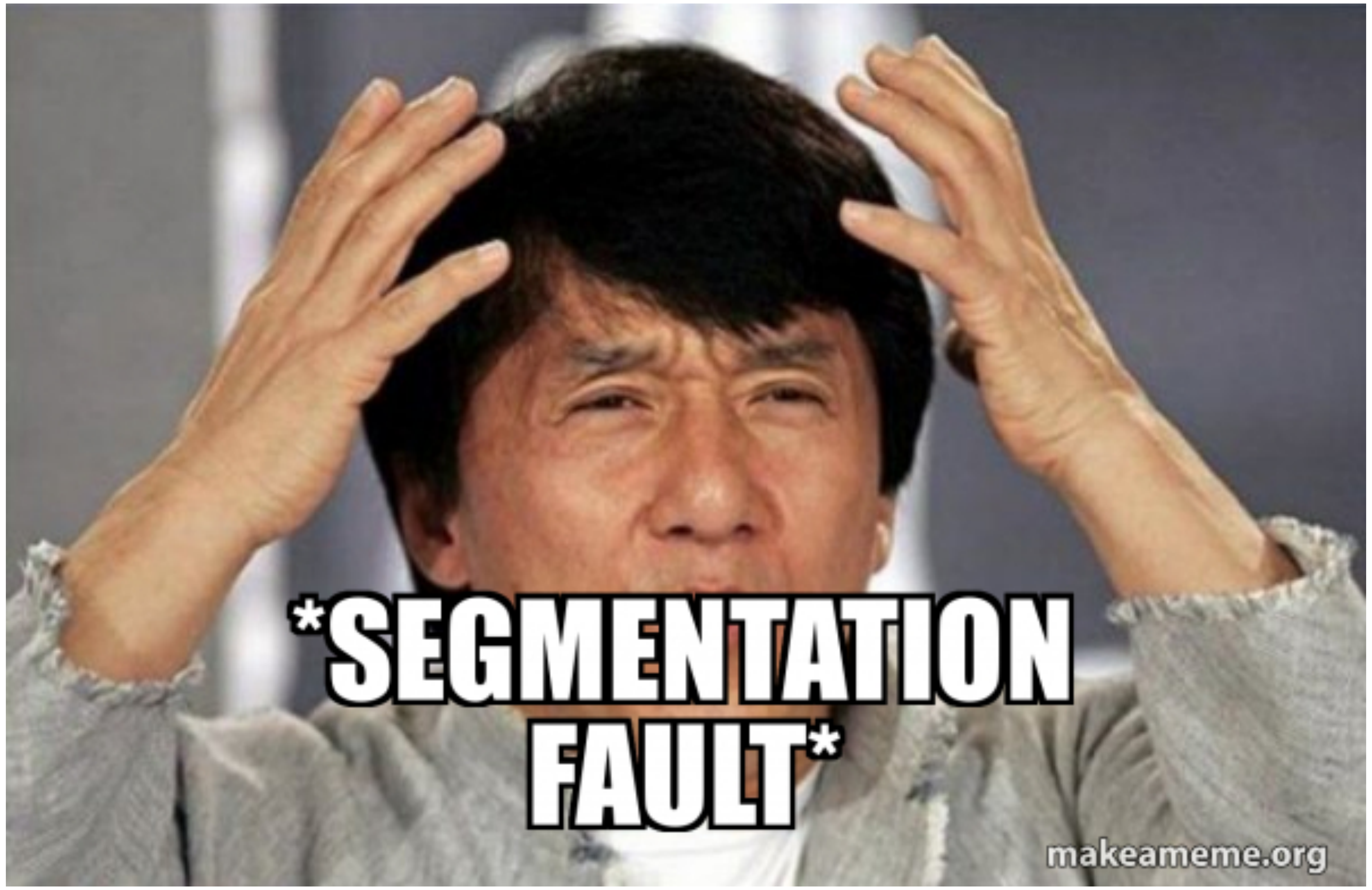
Free() call

- Free the heap memory
- Takes as argument the pointer returned by malloc.
    - The size of allocated region is not passed by user
    - Tracked by the memory allocation library itself
- Not enough we do malloc
    - Its very important to free it - why?

Common Errors made by Programmers

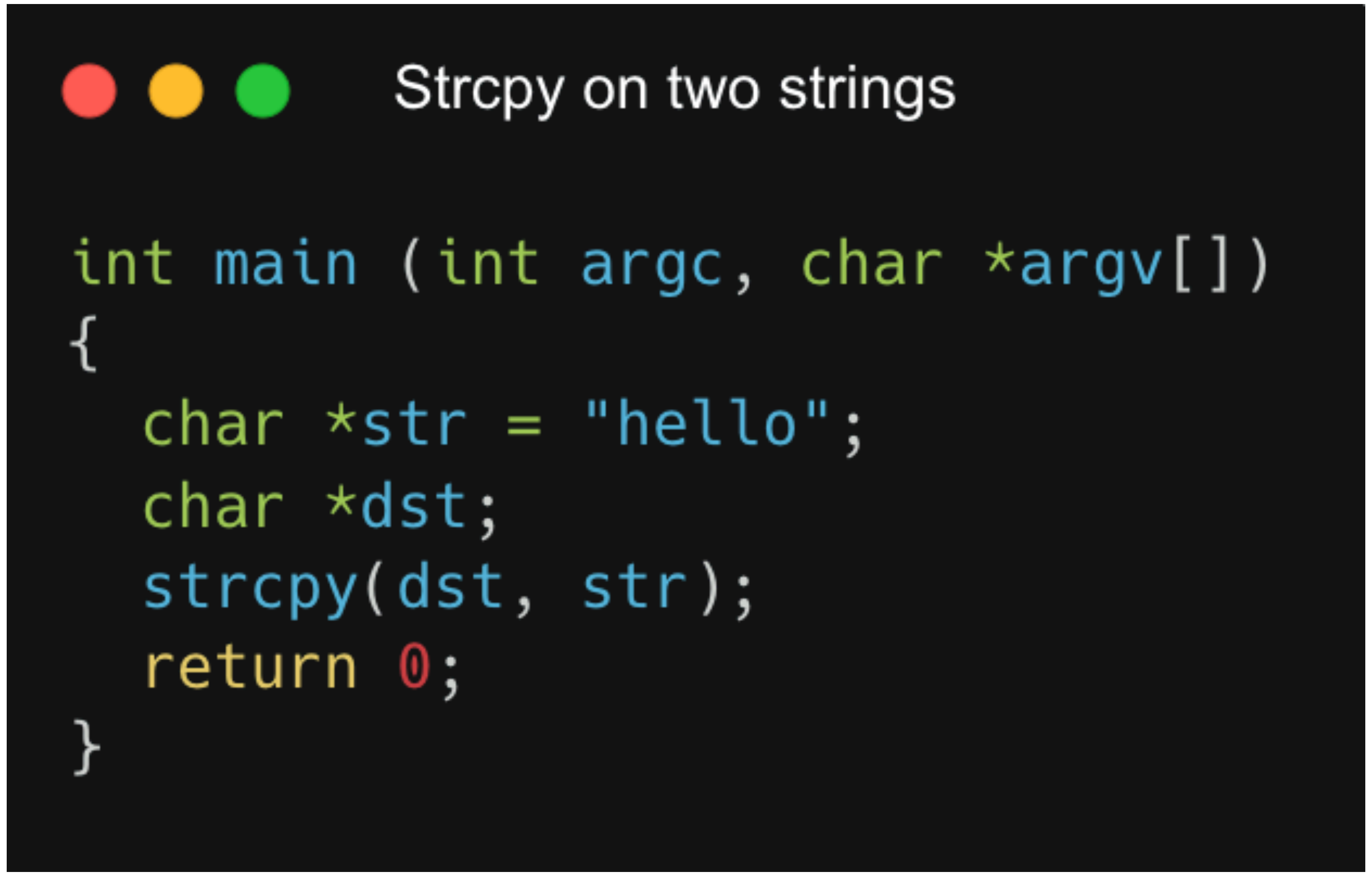
- Lot of errors arise in the usage of malloc() and free()
- Error free memory management has always been a problem
  - Modern programming languages support it implicitly
  - Most of the times we may call something similar to malloc()
  - Free is not called in most languages by programmers
    - Garbage collectors in Java

Ever Come Across This?



Error 1: Forgetting to allocate memory

- Many routines expect memory to be allocated before invoked



Strcpy on two strings

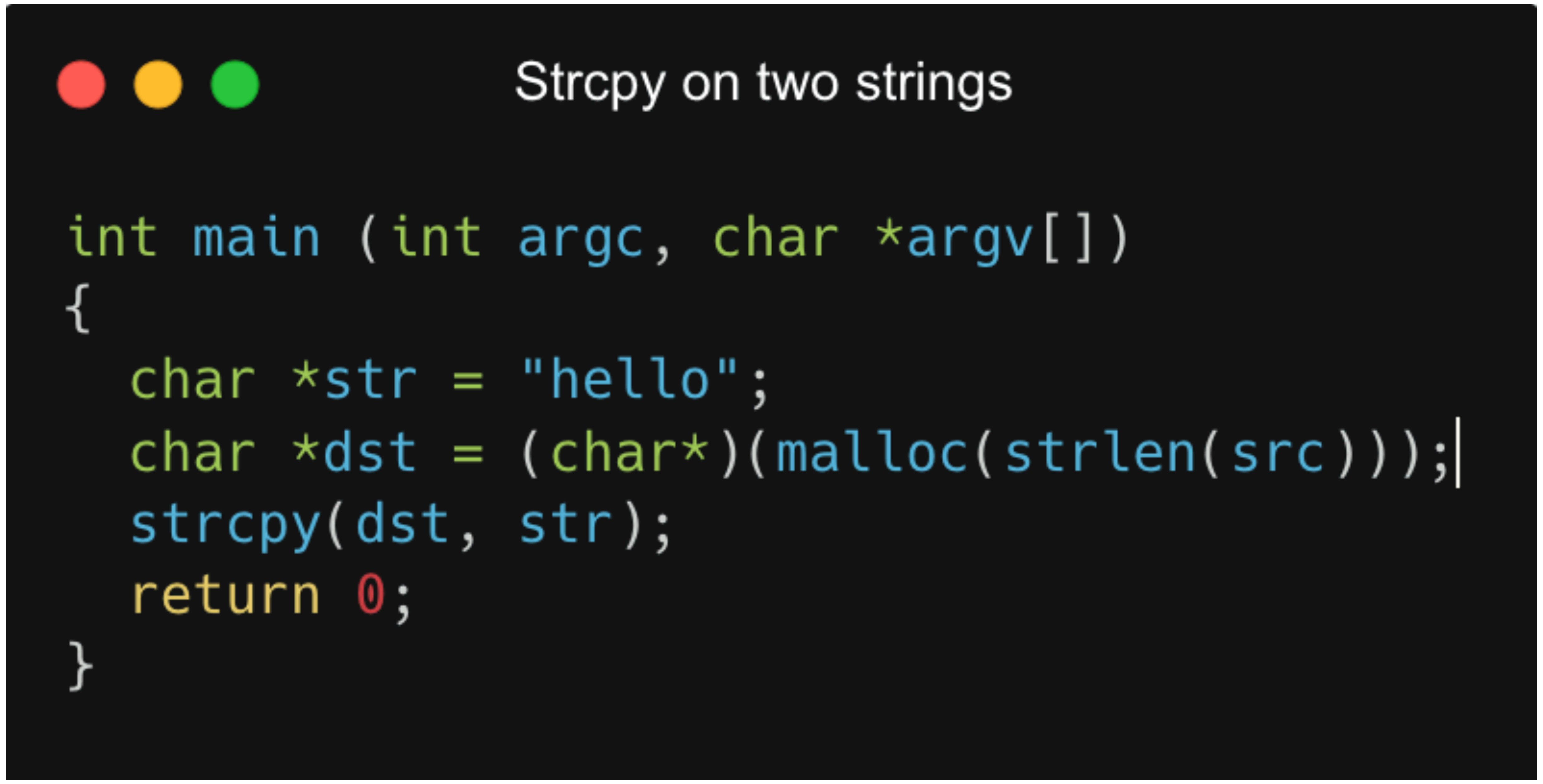
```
int main (int argc, char *argv[])
{
  char *str = "hello";
  char *dst;
  strcpy(dst, str);
  return 0;
}
```

Is there some issue?

Segmentation Fault!!

Not allocating enough Memory

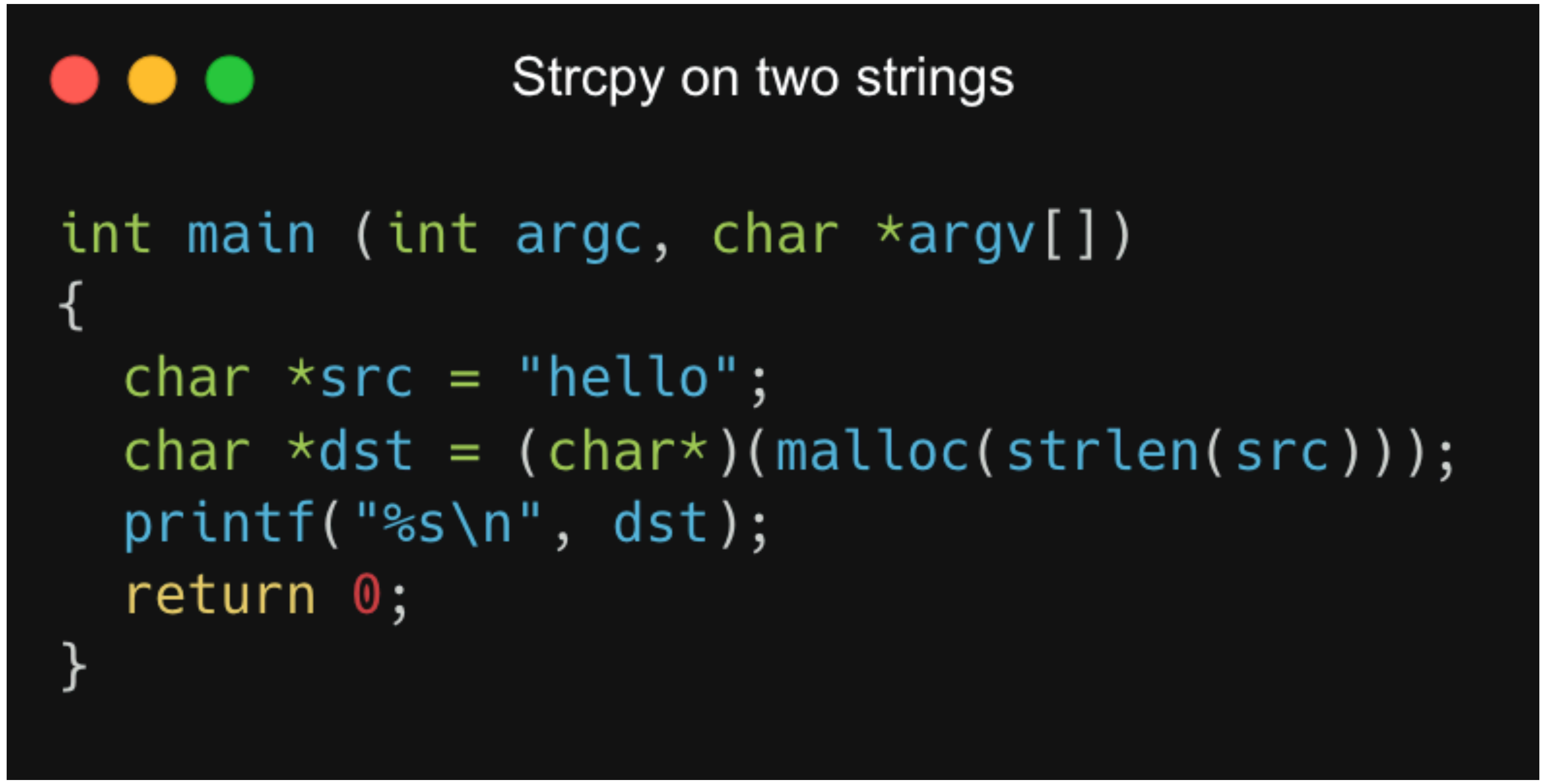
Yes, this can also be a problem



```
int main (int argc, char *argv[])
{
  char *str = "hello";
  char *dst = (char*)(malloc(strlen(src)));
  strcpy(dst, str);
  return 0;
}
```

- Depending on how malloc is implemented, this may work more often
- **strcpy** may write one byte past the allocated space
- This may result in **overflow** - **It ran correctly doesn't mean its correct!**

Forgetting to Initialize Allotted Memory



Strcpy on two strings

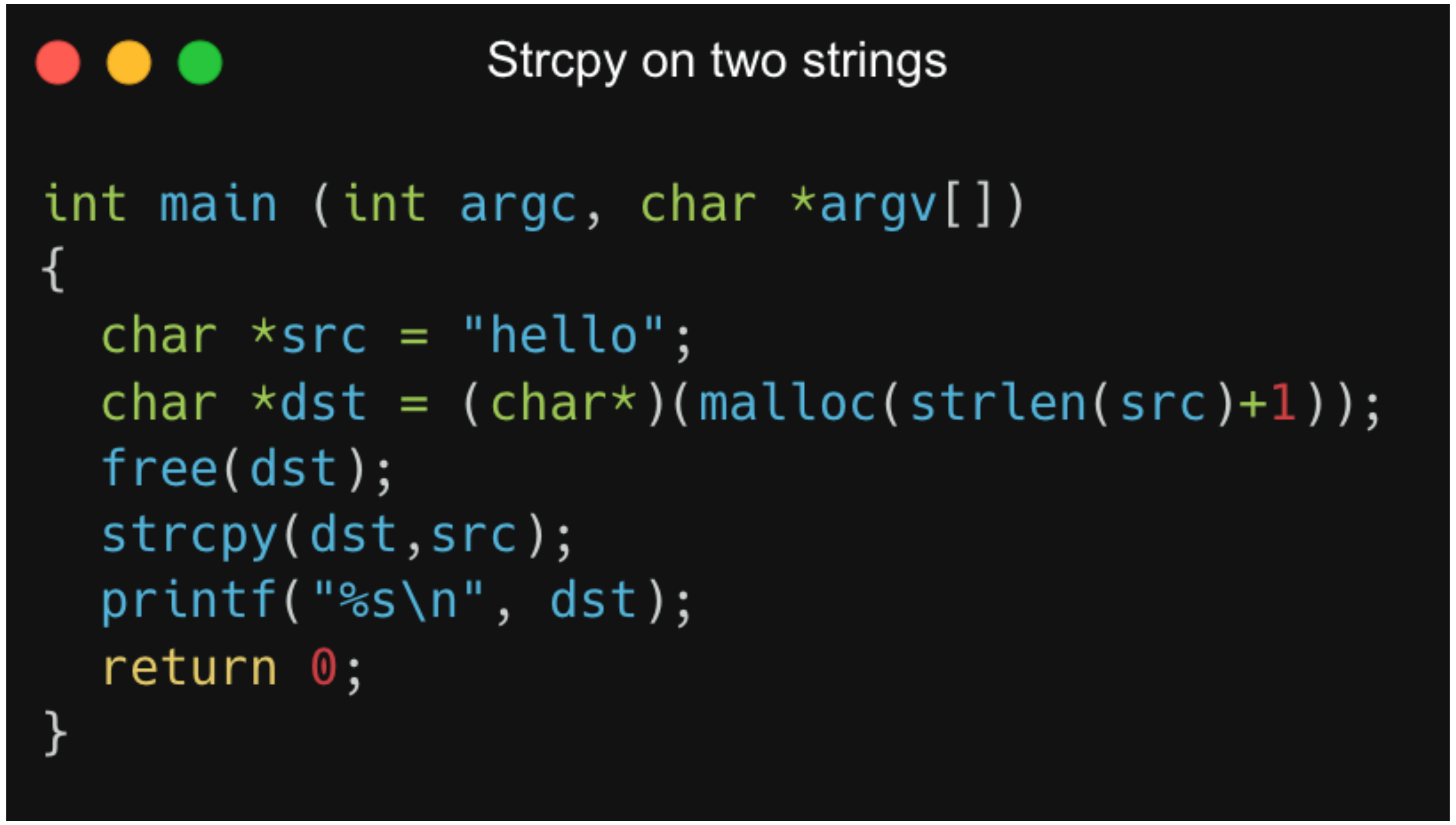
```c
int main (int argc, char *argv[])
{
  char *src = "hello";
  char *dst = (char*)(malloc(strlen(src)));
  printf("%s\n", dst);
  return 0;
}
```

- malloc() is called properly but no value assigned
- May result in an error -> **Uninitialized read**
- It may read some data of unknown value from the heap => program will be affected!

Forgetting to Free Memory

- Results in **Memory Leak**
- Occurs when one forgets to free memory after use
- Slowly leaking memory => system runs out of memory => **System restart!!**
- When done with chunk of memory - free it off
- Best solution: Ensure program exits! OS will clean up everything

Freeing Memory before the completing the use

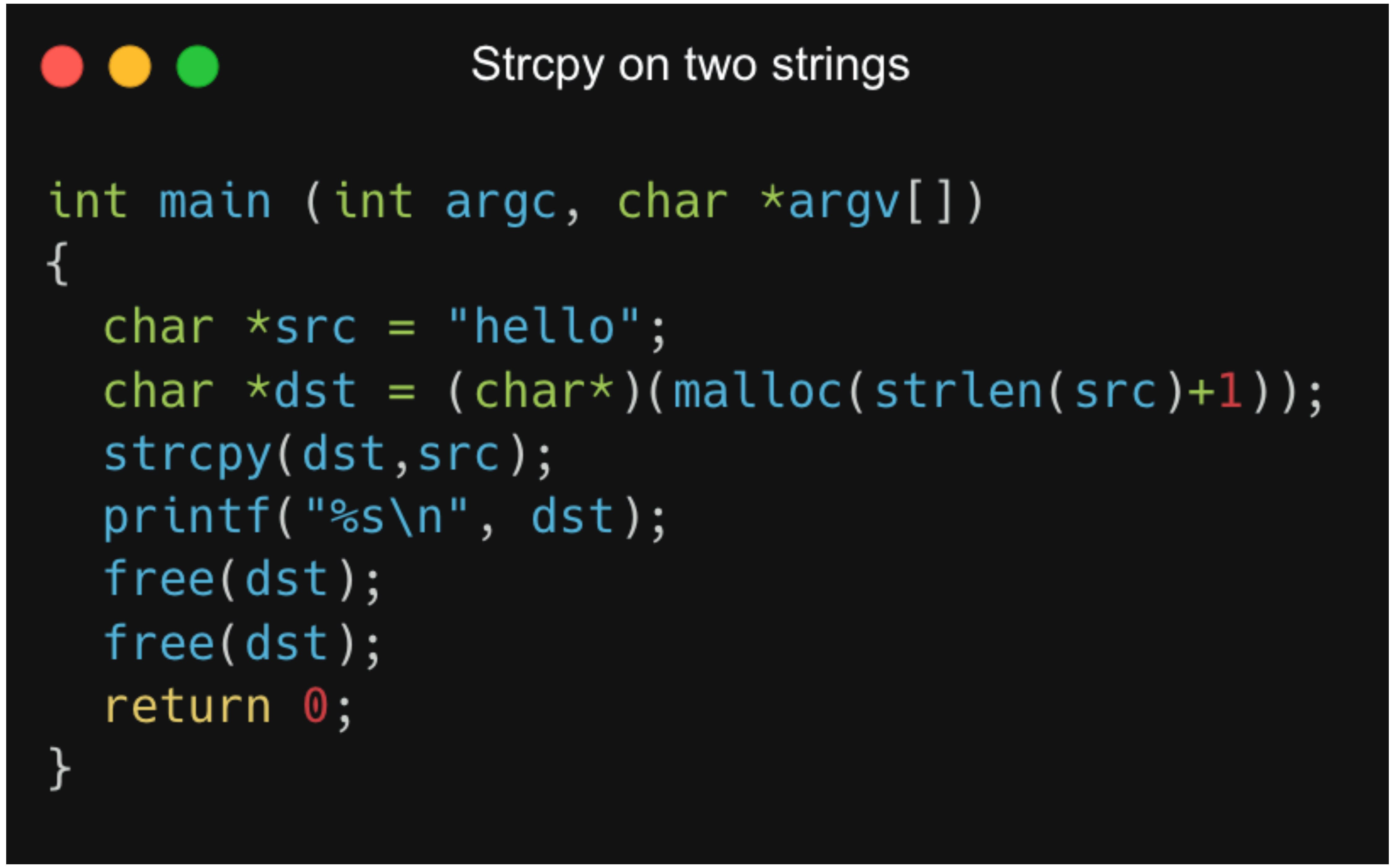


```
int main (int argc, char *argv[])
{
  char *src = "hello";
  char *dst = (char*)(malloc(strlen(src)+1));
  free(dst);
  strcpy(dst,src);
  printf("%s\n", dst);
  return 0;
}
```

- Calling free before using it
  - Subsequent call of the pointer can crash the program or overload memory
- Results in a **potential error** due to **Dangling Pointer**

Freeing More than once

Too much of anything is dangerous!!

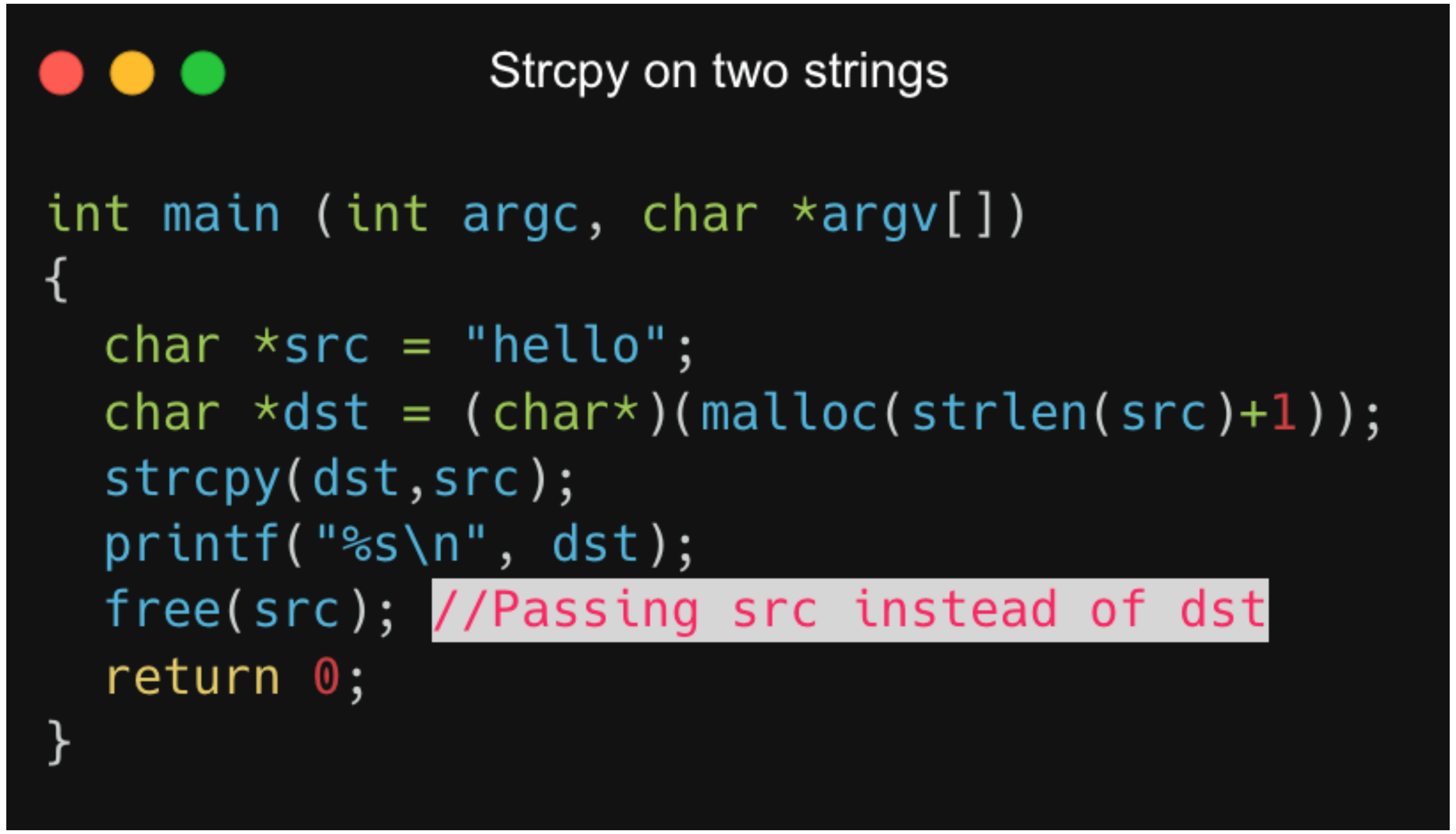


Strcpy on two strings

```
int main (int argc, char *argv[])
{
  char *src = "hello";
  char *dst = (char*)(malloc(strlen(src)+1));
  strcpy(dst,src);
  printf("%s\n", dst);
  free(dst);
  free(dst);
  return 0;
}
```

- Free memory more than once
  - **Double free** error
- May result in undefined issues - Memory allocation library may get confused

Calling free incorrectly



Strcpy on two strings

```
int main (int argc, char *argv[])
{
  char *src = "hello";
  char *dst = (char*)(malloc(strlen(src)+1));
  strcpy(dst,src);
  printf("%s\n", dst);
  free(src); //Passing src instead of dst
  return 0;
}
```

- free() expects to get the pointer returned from malloc() as input
- When another value is passed, bad things happen
- **Invalid free** needs to be avoided

Common Issues with Memory

- Lots of issues with memory exist and abusing of memory happens
  - Lots of tools exist to solve issues - valgrind, purify, etc.
- malloc() and free() are not system calls rather just library calls
  - stdlib.h - library in C that provides functions malloc and free
  - Built on top of system calls - brk or sbrk
  - Brk or sbrk increases or decreases the size of heap based on value
  - Not advised to call them directly

More on Memory related APIs

- Another system call that can be used is mmap()
    - Creates anonymous memory region within the program
- Variations of malloc() exist
    - calloc() -> allocates memory and initialises with 0's.
    - realloc() -> add something more to the existing space allocated with malloc()

The Big Question: How to Virtualise

- Each process requires memory
- OS performs context switch between processes
- Process should not overwrite each others memory
- Users should not worry about memory allocations and where to store
- OS needs to virtualise memory
    - Can we do something similar to process virtualisation?
    - What are the two key aspects that enabled process virtualisation?

Memory Virtualisation: Key Requirements

- Bring hardware into the picture (similar to LDE)
  - Use some hardware support for memory management - **efficiency**
- OS can play its role when it comes to controlling
  - Ensuring that no application has direct access to memory by its own
  - Keep track of which locations are free and which are in use - **control**
- There should also be flexibility
  - Allow programs to use address space in the ways they like

The Overall Goal

- **Goal:** Create an illusion that each process has its own private memory where the code and data reside
  - Reality: Many processes are actually sharing memory at the same time!
- How to make this happen? - Three Key assumptions:
  - User address space must be placed contiguously in physical memory
  - Size of address space is not too big; less than size of physical memory
  - Each address space is of exactly the same size

Memory Virtualization: An Analogy

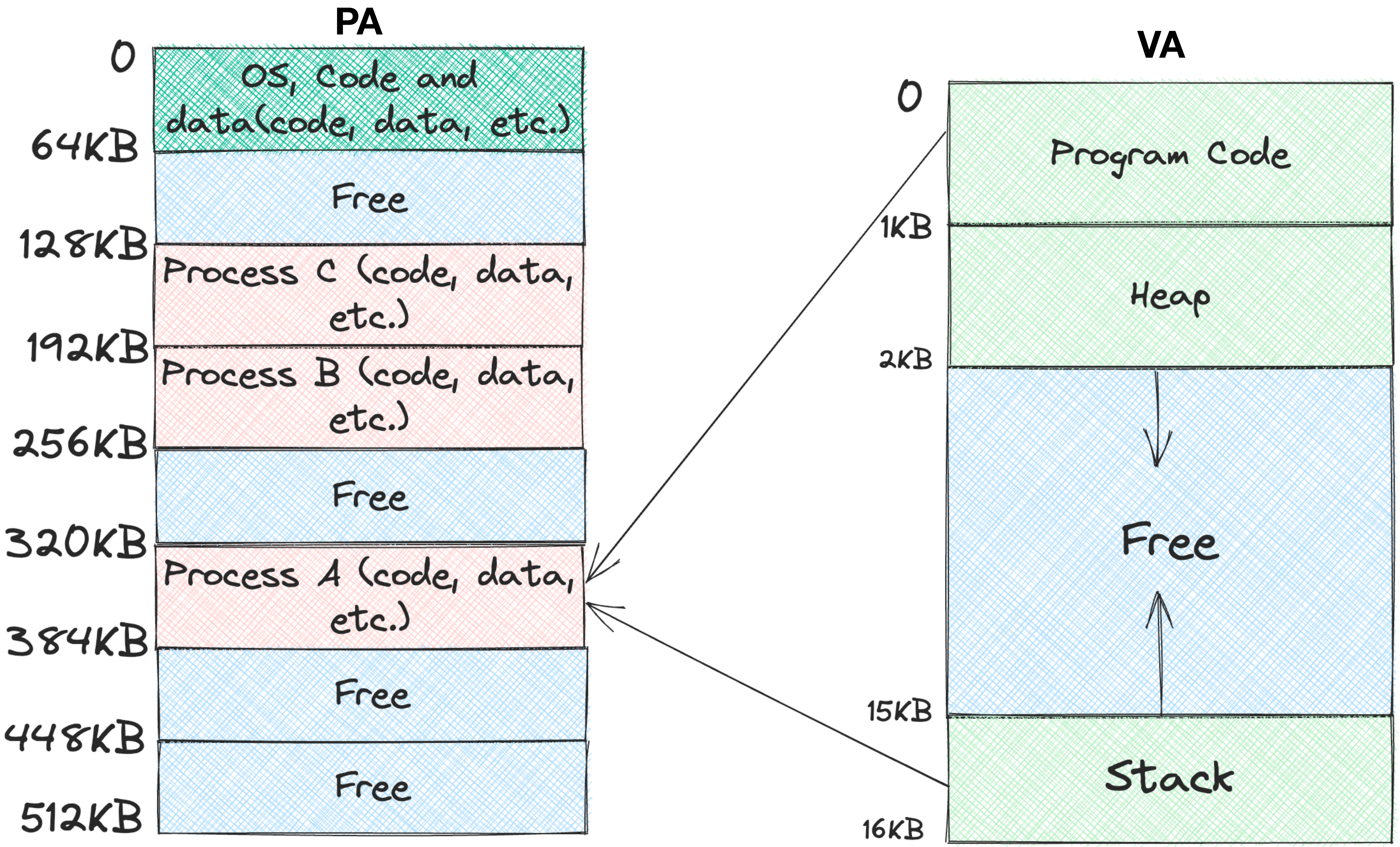
Onsite Shopping

Every users have access to different items but to a limited set

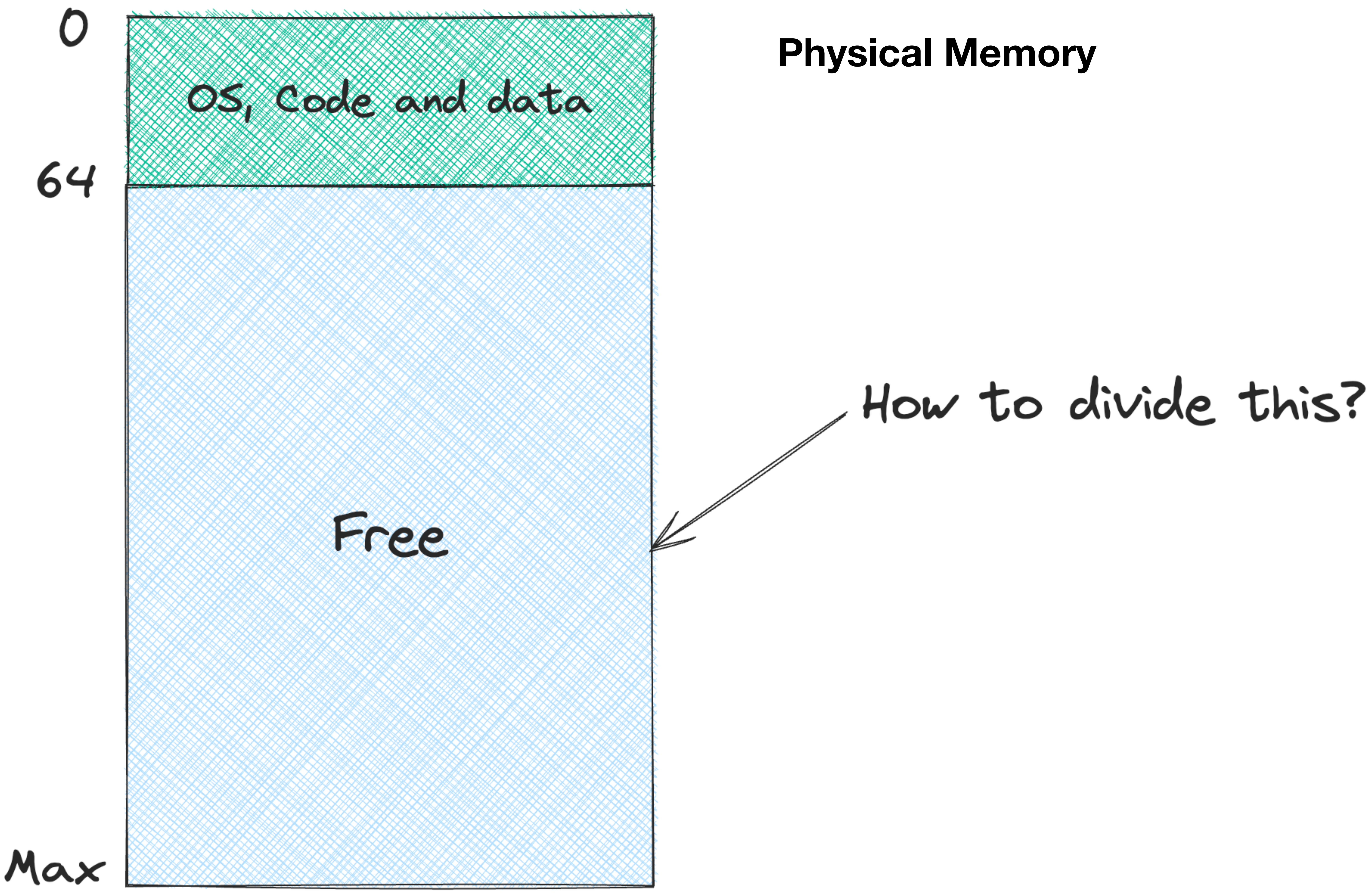
Online Shopping

Every Users feel that they have access to infinite items

Address Translation — Recap

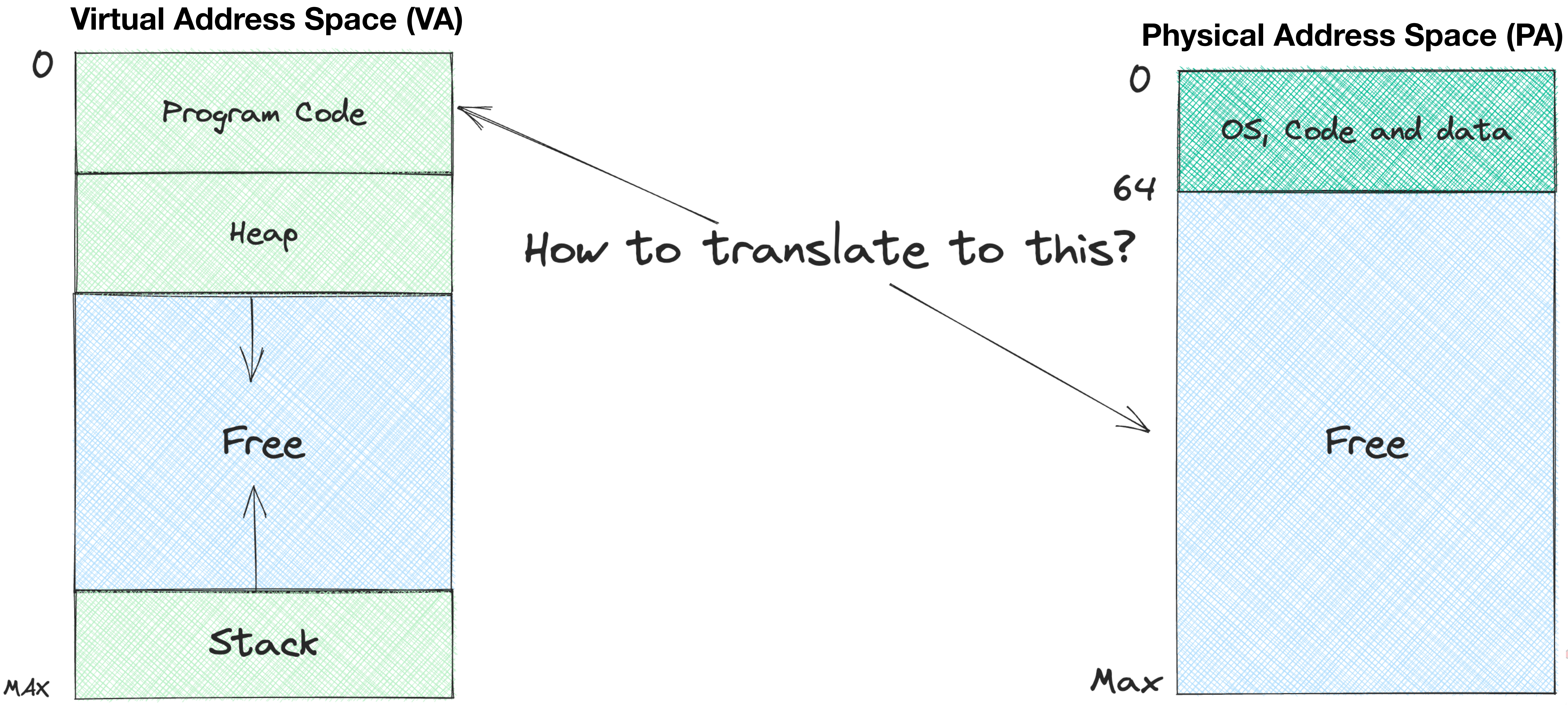

PA
0
64KB
128KB
192KB
256KB
320KB
384KB
448KB
512KB
OS, Code and data(code, data, etc.)
Free
Process C (code, data, etc.)
Process B (code, data, etc.)
Free
Process A (code, data, etc.)
Free
Free
VA
0
1KB
2KB
15KB
16KB
Program Code
Heap
Free
Stack

Essentially its about two things!



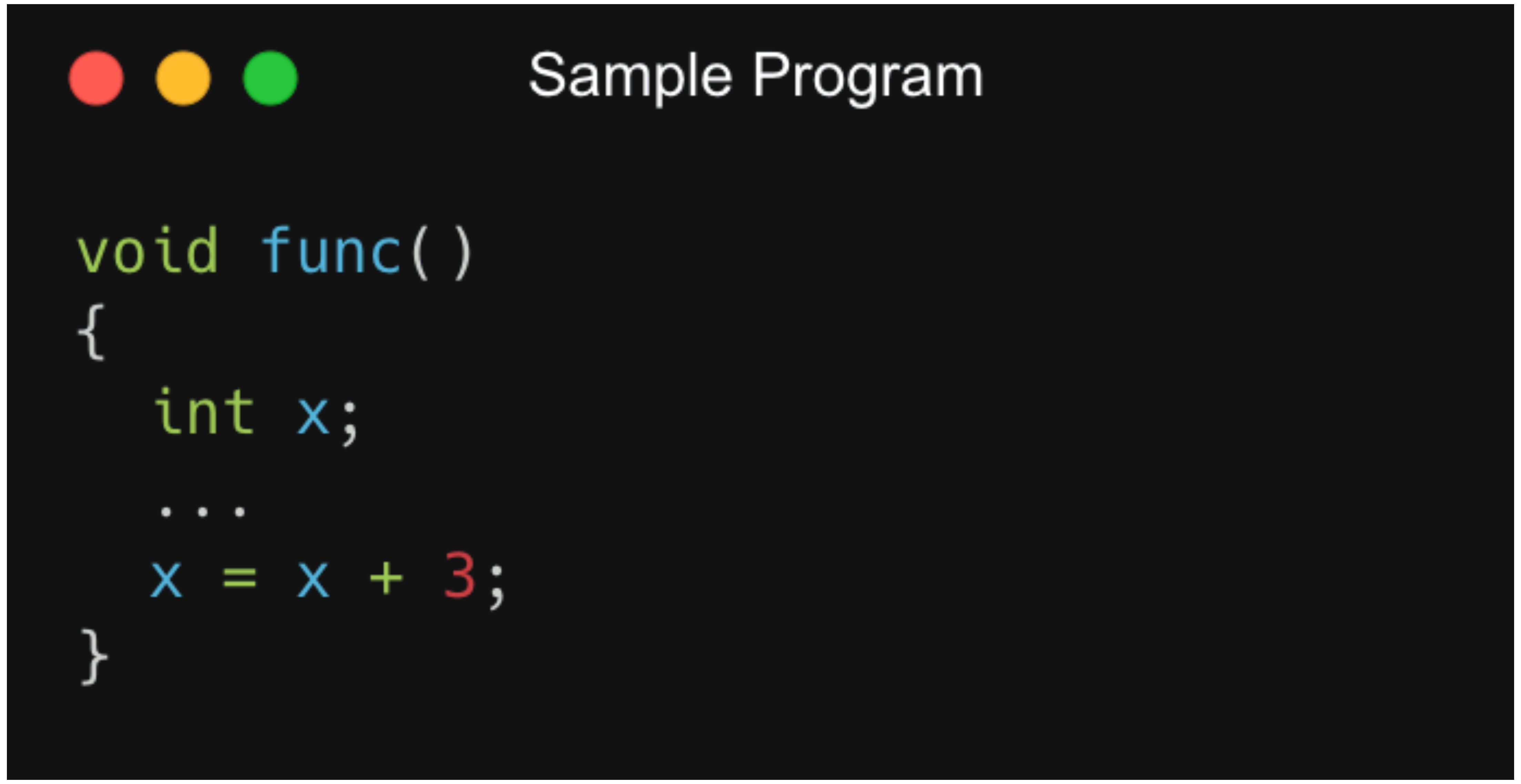
Physical Memory

Essentially its about two things!

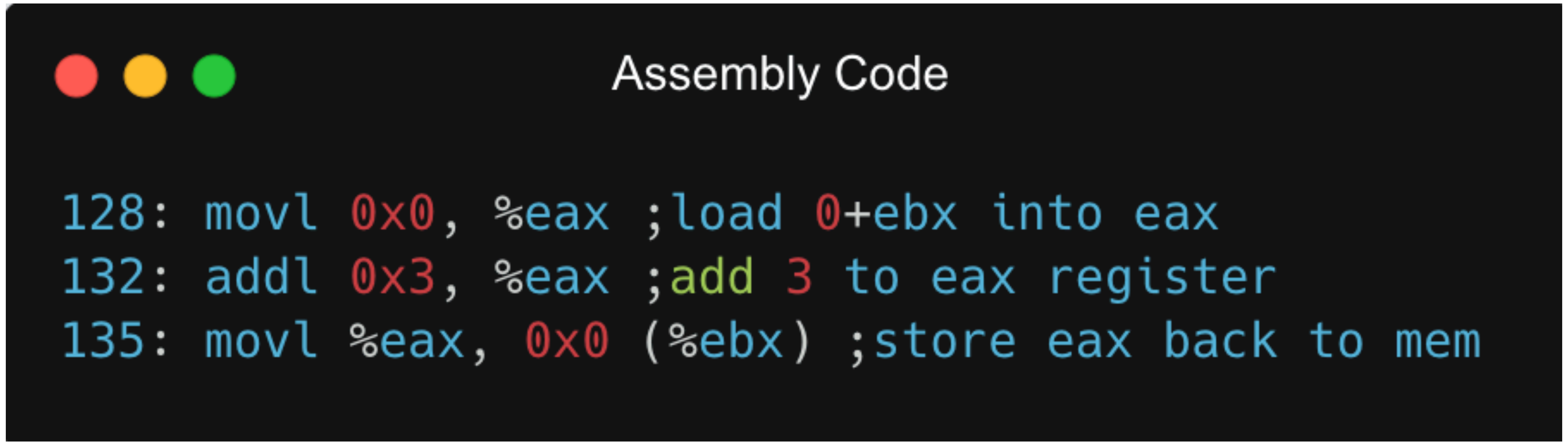

Virtual Address Space (VA)
0
Program Code
Heap
Free
Stack
MAX
How to translate to this?
Physical Address Space (PA)
0
OS, Code and data
64
Free
Max

Simple Program

C Program to Assembly



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

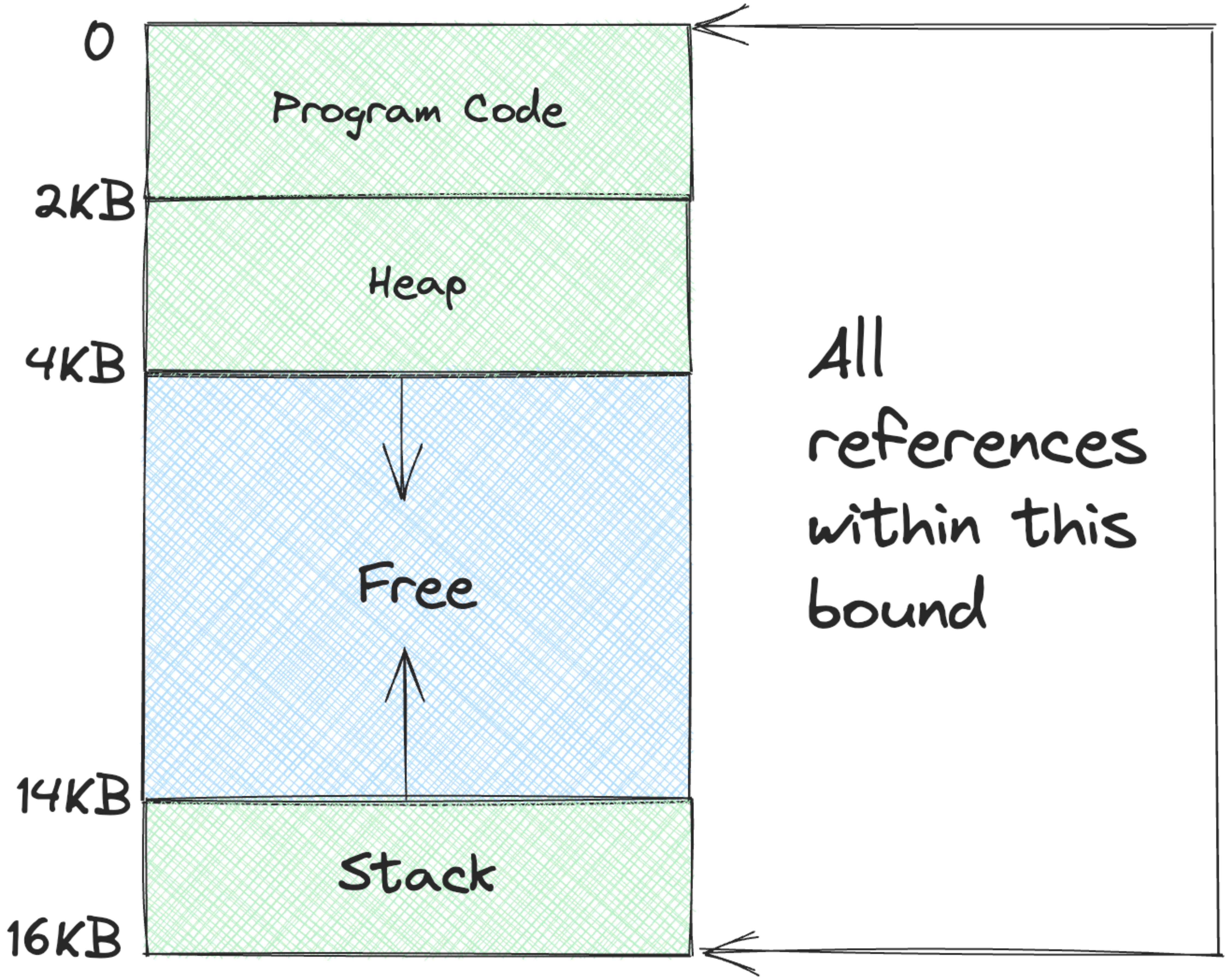


```
128: movl 0x0, %eax ;load 0+ebx into eax
132: addl 0x3, %eax ;add 3 to eax register
135: movl %eax, 0x0 (%ebx) ;store eax back to mem
```

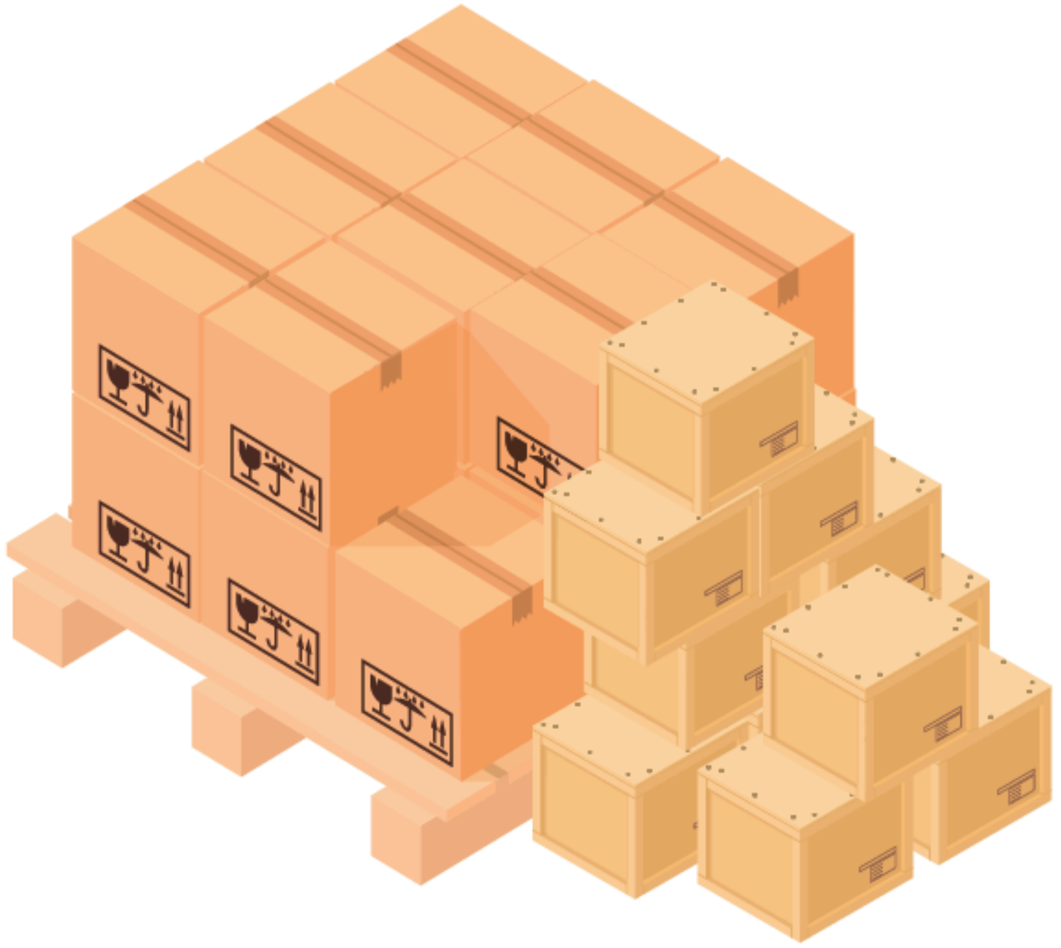
int x = 3000;

Following Process happens

1. Fetch instruction at 128
2. Execute the instruction (load address)
3. Fetch instruction at 132
4. Execute the instruction (No memory reference)
5. Fetch instruction at 135
6. Execute the instruction (Store to 15 KB)



Warehouse Scenario



Warehouse with lots of new packages/shipments

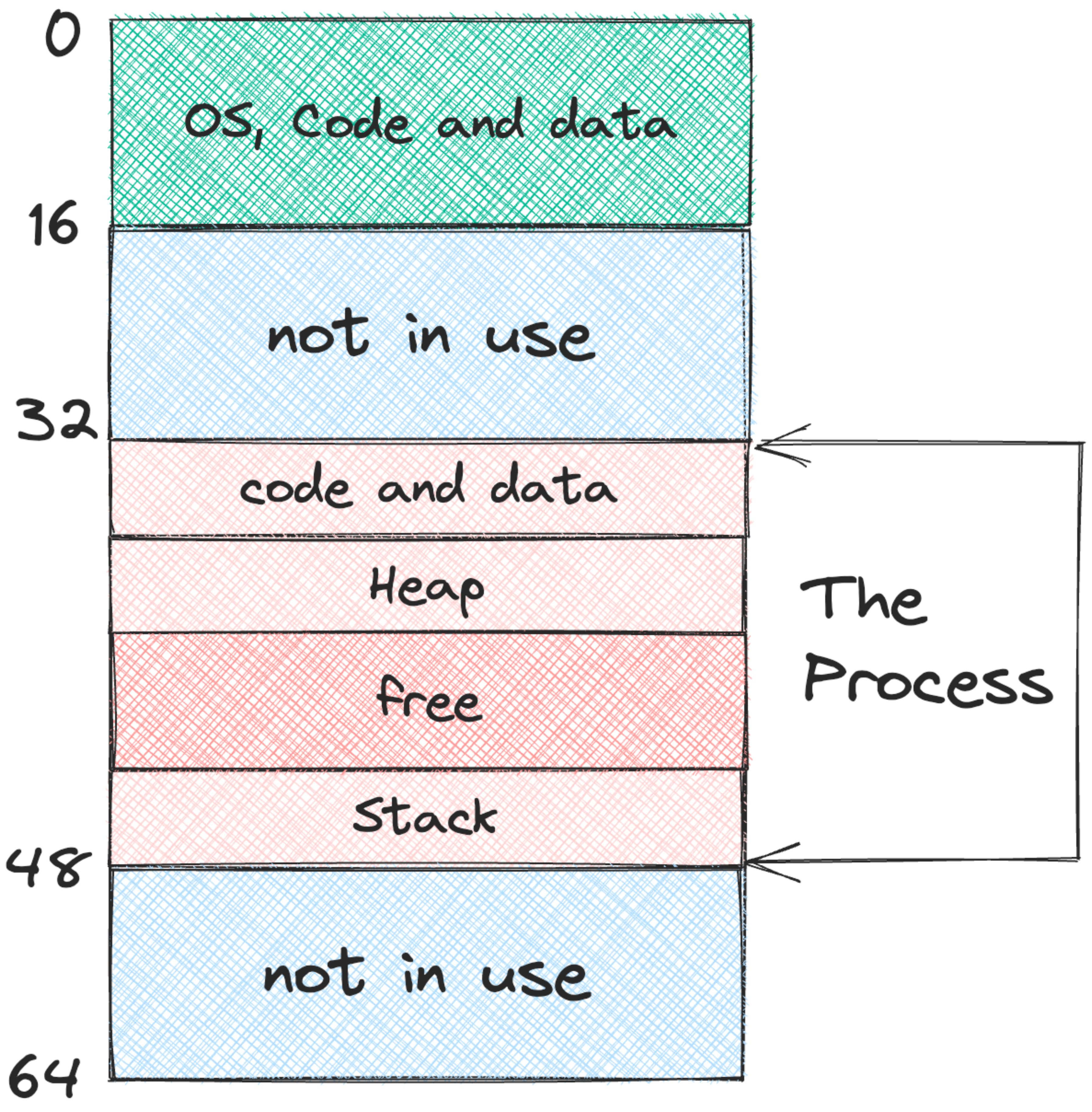
Based on a Category: Range can be decided
Category like Electronics, Clothing, etc

They can be grouped - Each type of shipment can be grouped in a range of locations (0 - 200: Electronics)

Manager/other staff: Simply go to the corresponding range to find the product - There is a starting and ending value

Can we not do this at Physical Memory Level?

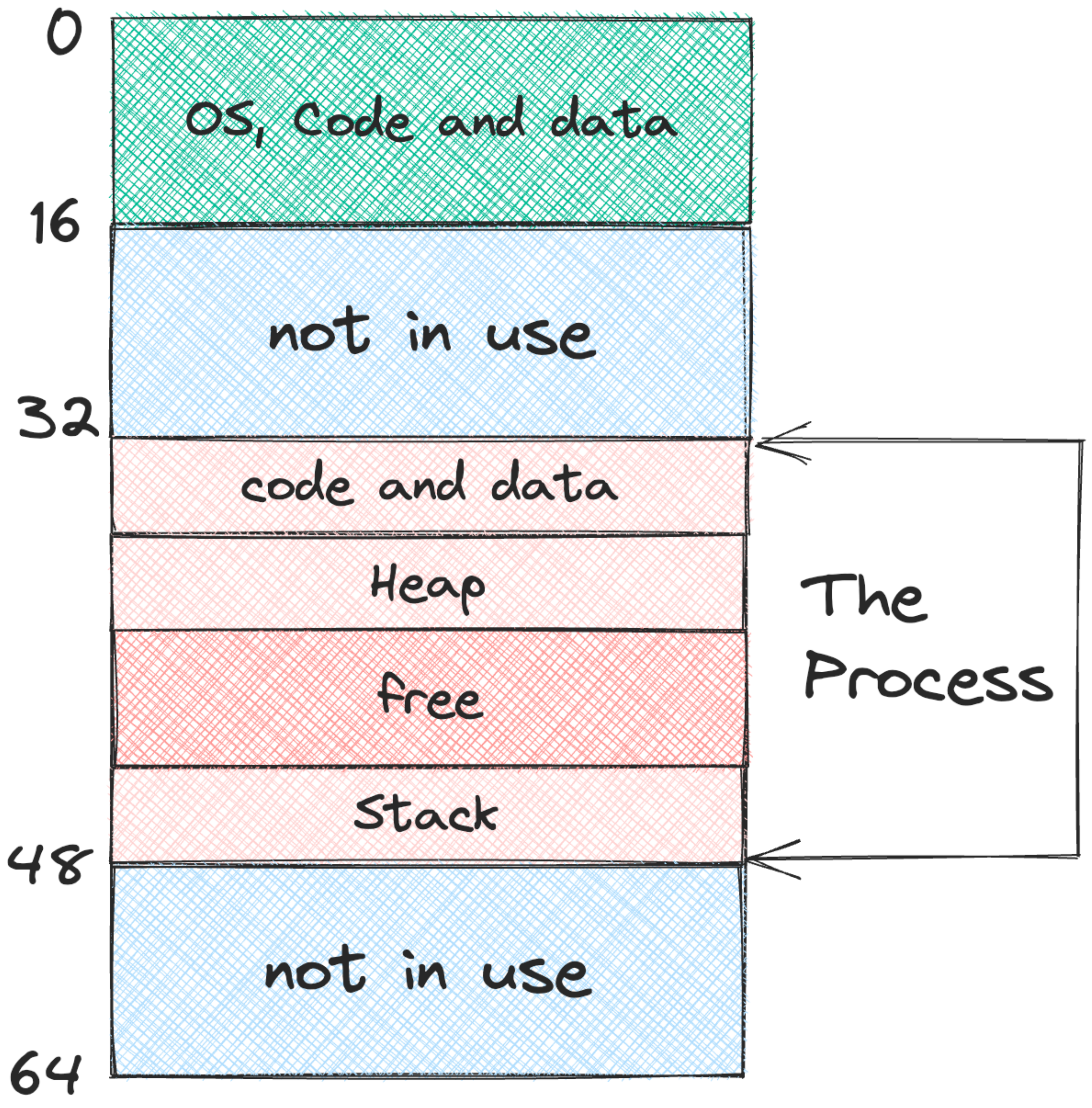
- To virtualize, OS cannot place the process starting from 0.
- The process requires same amount of space as in Virtual address space but somewhere else.
- The reality of physical memory is different from what the process sees!
- The process of translation just needs to map the two
- Can you think of a simple approach?



Dynamic Relocation

The Base and Bounds approach

- Each process allocated **contiguous memory (Segment)**
- Two hardware registers in the CPU (MMU)
  - Base register
  - Bounds register (limits register)
- Each program is written and compiled as if it is loaded at 0
  - However, when the program needs to be run, OS decides the location in physical memory
  - Sets base register to that value
  - Here 32 KB becomes the value in base register



Dynamic Relocation

The Base and Bounds Approach

Physical address = Virtual address + base

- Every memory reference generate by process is virtual address
- Hardware just adds the base value to generate the actual physical address
- This process of transforming VA to PA => (hardware-based) **Address translation**
- Since this happens at runtime => **Dynamic relocation**
- There is only one pair of base and bounds register in the MMU
- OS can make use of simple data structure to keep track of available memory (free list)

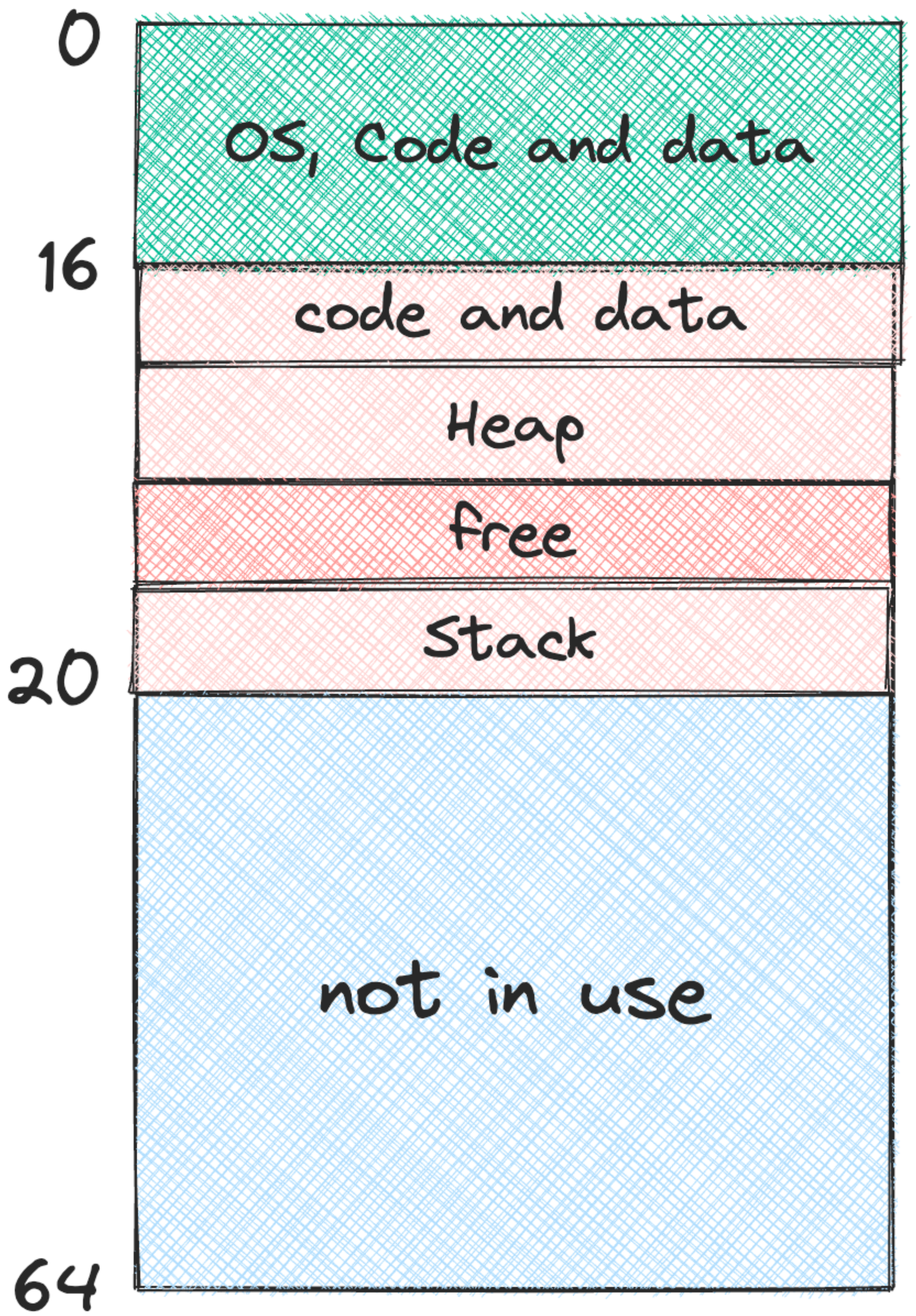
Dynamic Relocation

The Base and Bounds Approach

- **Bounds** register ensures that any memory reference is within bounds
  - Everything has to be a legal access
  - If process generates address > bounds (Either relative to VA or PA)
    - CPU raises an exception (Interrupt raised)
    - Process is terminated
- The base and bounds are registers part of hardware (Kept on chip)
- These registers will be inside **Memory Management Unit (MMU)**

Illustration of Base and Bounds Approach

- Process A has an address space of 4 KB, assume that the base is 16 KB
  - Lets say there is an access to VA 0 - PA?
    - PA: 16KB
  - Access to VA 3000 - PA?
    - PA: 16384 + 3000 = 19384
  - Access to VA 4400 - PA?
    - PA: 16384 + 4400 = 20784! **Fault! Why?**



There are some issues!

WHAT ARE YOU WORKING ON?
TRYING TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN...

Source: xkcd

Some Possible Issues

- Simple base and bounds approach is very limiting
  - Memory is contiguous
  - One base and bounds pair per process in the MMU
  - How to support large address space?
- Lot of free space between stack and heap may go unused
  - A typical program would use only certain amount of memory
  - But may demand more! - How to address this?

Segmentation

Generalized Base and Bounds approach

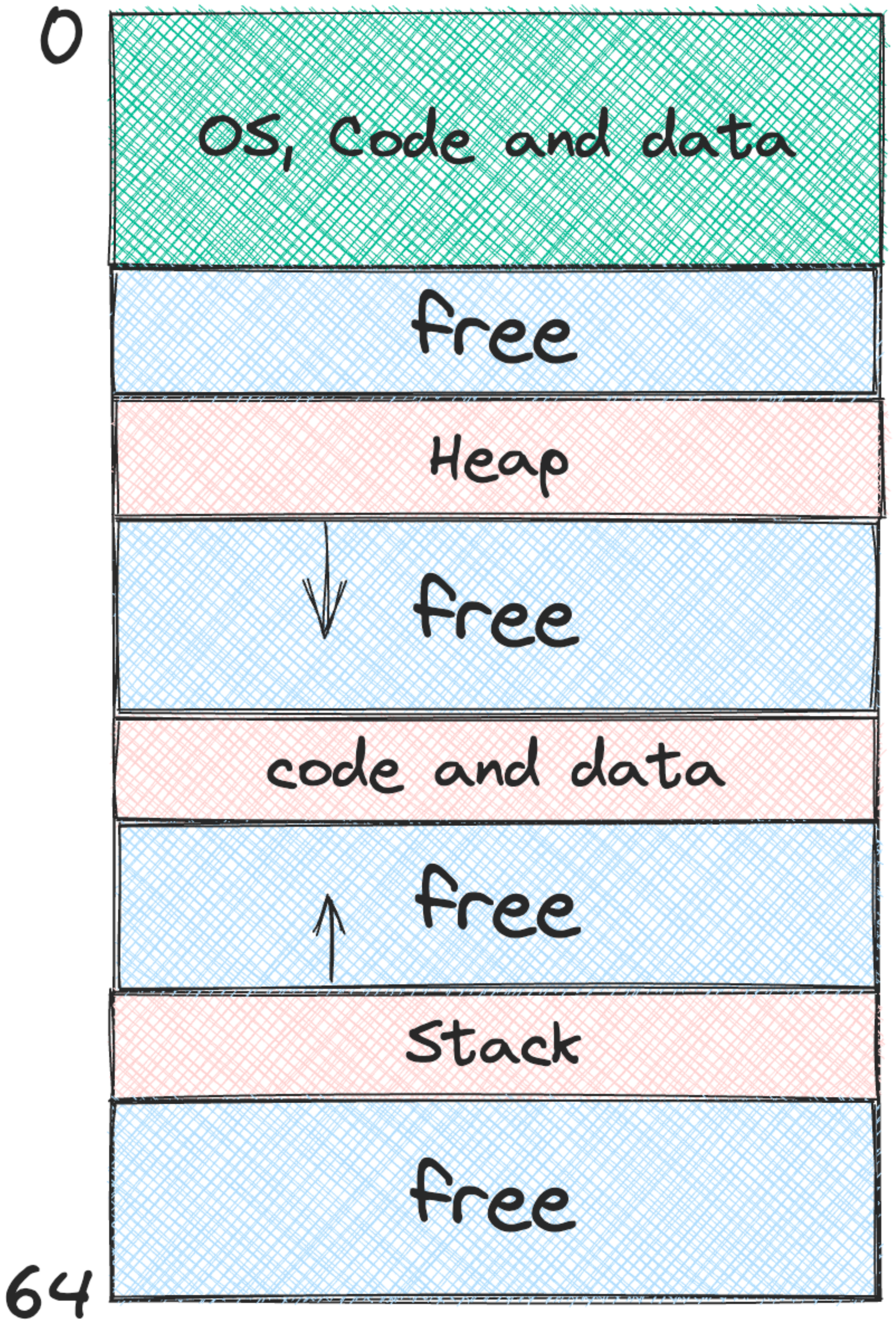
- Instead of having one base and bounds per process
  - Why not have it per logical segment of the address space?
- Segment: Contiguous portion of the address space of a particular length
  - In canonical address space - Three segments
    - Code, Stack and Heap
  - Segmentation basically allows each segment to placed in different parts of memory

Segmentation

Generalized Base and Bounds

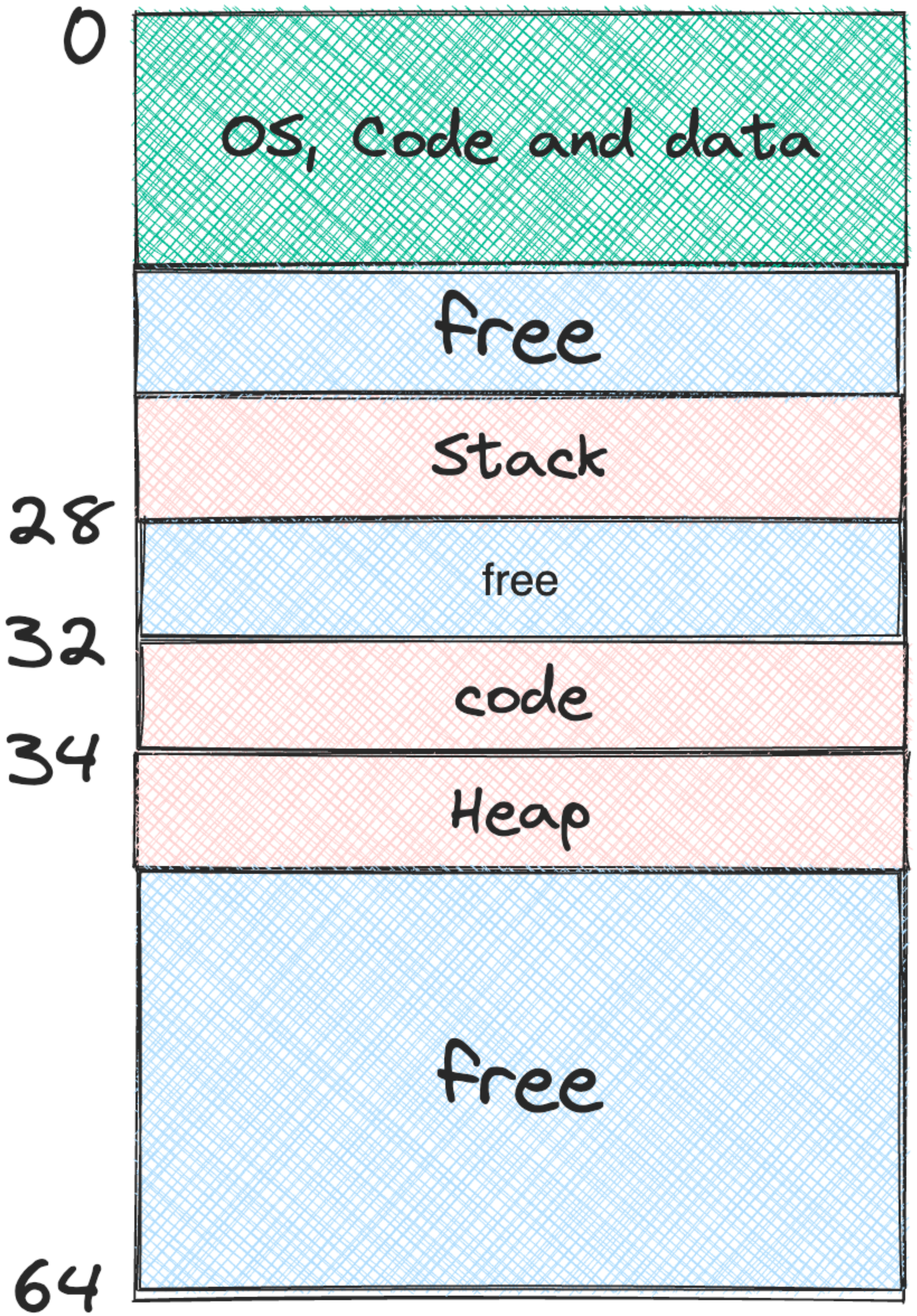
- Only used memory is allocated in physical memory
  - Allows allocating large address space
  - Sparse address space

- Note: Different segments can be placed in different parts of the memory - **How does mapping work?**



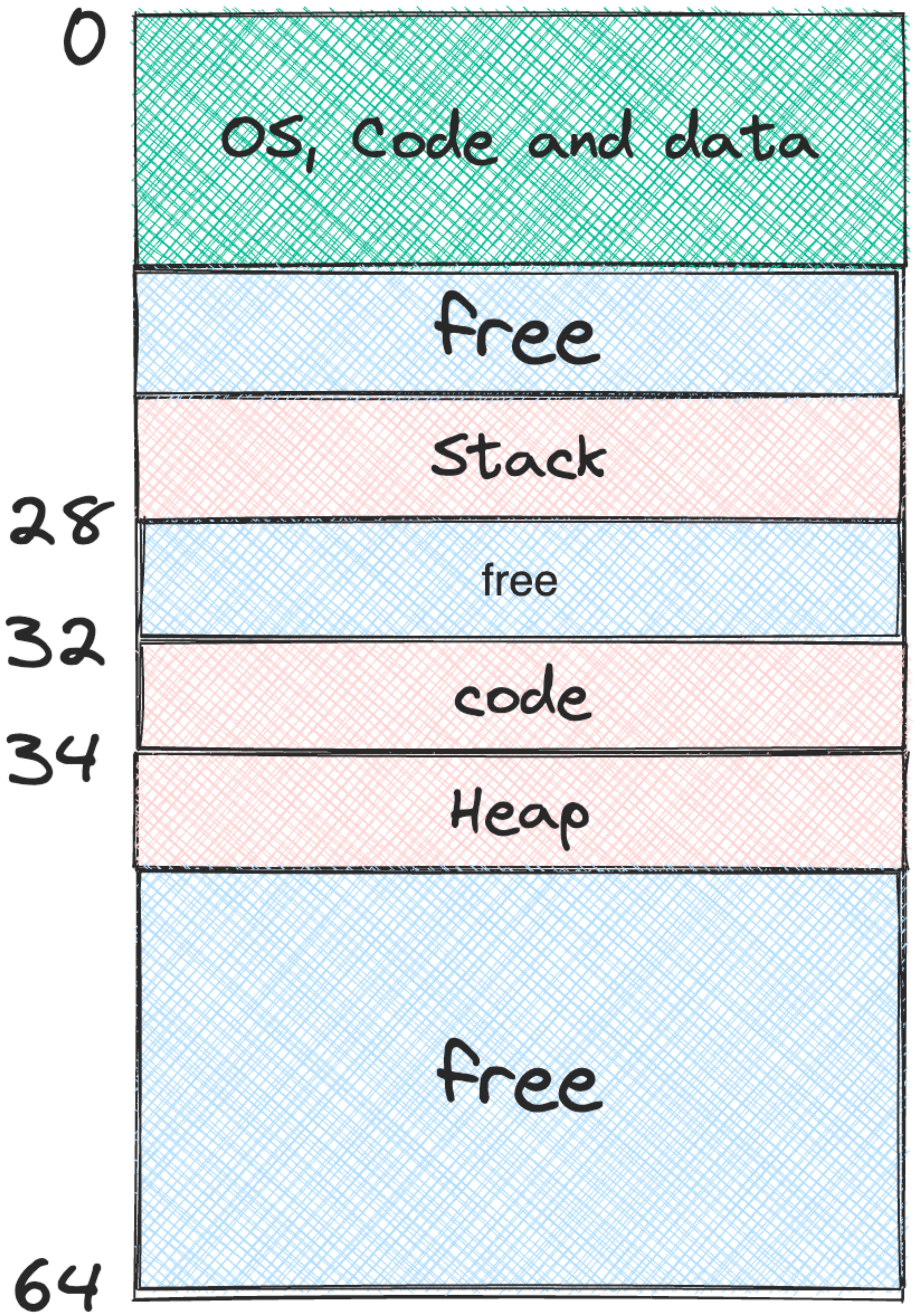
Hardware support (Registers)

Segment	Base	Size
Code (00)	32K	2K
Heap (01)	34K	2K
Stack (11)	28K	2K



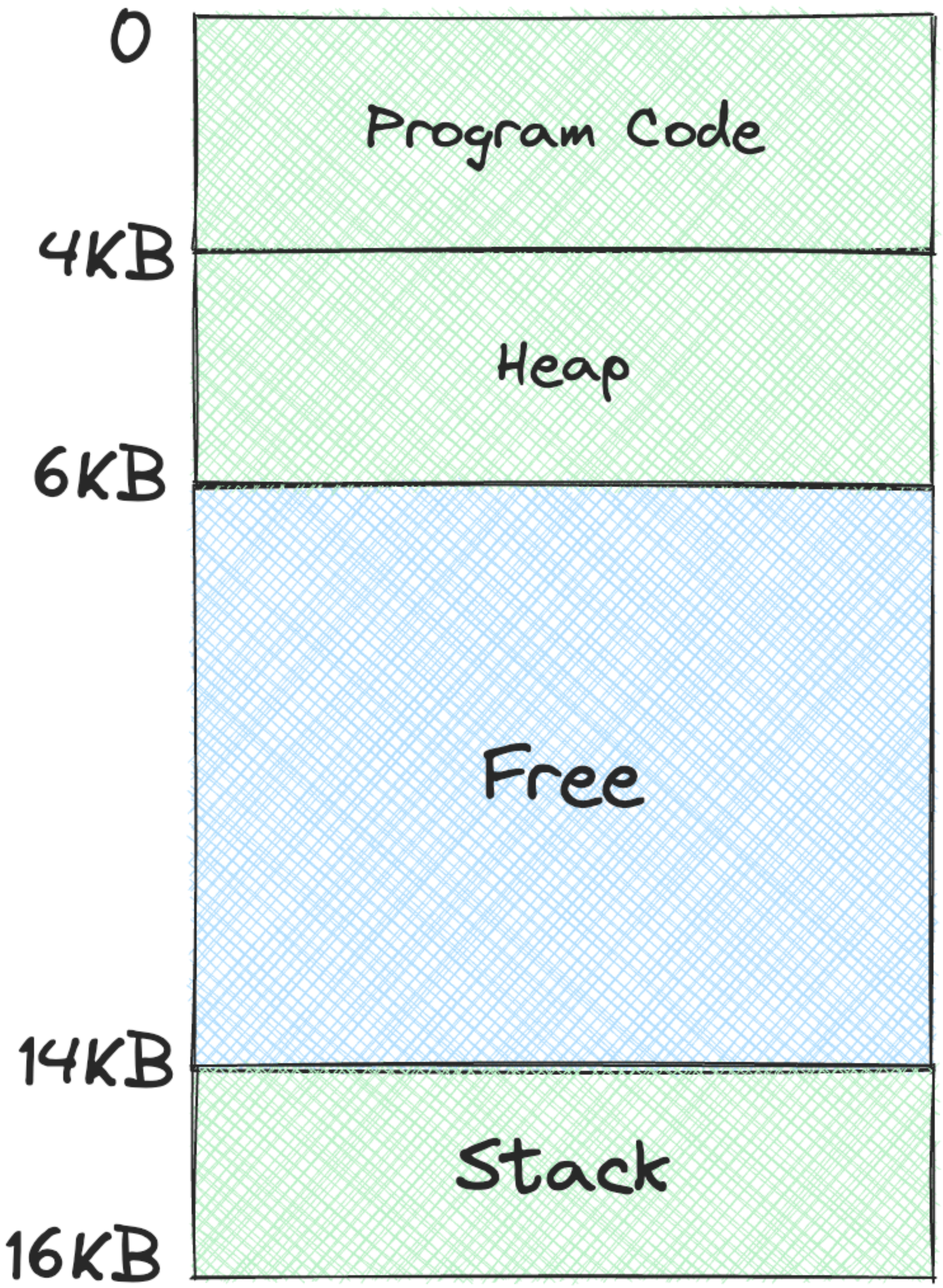
Simple Address Translation

- Reference is made to VA: 100 and code segment
  - MMU: Code starts at 32K
  - PA: 100 + 32868 (32 KB) (100 < 2K)
- Reference made to 4200 to heap segment
  - Can we just add 4200 to base of heap - 34816?
  - Code starts at 0 in virtual address space
  - Heap starts at different location - Get offset?



Simple Address Translation

- Heap starts at 4KB in VA:
  - Offset is 4200
  - Actual base value: 4200 - 4096 = 104: PA?
    - PA: 104 + 34816 (34 KB) = 34920
- How about VA of 7KB (beyond heap address)?
  - Address out of bounds - process termination
  - **Segmentation fault or violation!**



Wait! How to Identify the segments?

- Different segments per process - Code, stack and heap
- Two different approaches - Explicit and implicit
- Explicit approach
  - VA: 14 bit address
  - Use first two bits to identify segment and rest offset

Bits	Segment
00	Code
01	Heap
11	Stack
10	-

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	0	1	0	0	0

Segment (12-13)

Offset (0-11)

Wait! How to Identify the segments?

- With two bits - Code, heap and stack can be referred
  - Still pair of bits go unused
  - Some systems puts code and heap in segment and uses only one bit
- Implicit Approach
  - Based on how address was formed
  - If it was generated by programming counter during fetch => **Code**
  - Based on stack pointer => **Stack**; else -> **Heap!**

What about Stack?

- Stack grows backwards!
- Some support from hardware to understand which direction to go
  - It is not just about addition to base
  - One bit can be used to indicate direction
  - Each bit implies extra bit to represent the address

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K (max 4K)	1
Stack	28K	2K (max 4K)	0

Example of translation involving stack

- Reference VA: 15 KB - Physical?
  - Try to put 15 KB in binary

13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0

Segment: Stack (13 -12)

Offset: 3 KB (0 - 11)

  - Grows positive 0 (Going negative)
  - Maximum segment size in address space: 4 KB
  - Absolute value = 3 - 4 = -1 KB
  - PA: -1 + 28 (base) = **27 KB**

Bounds Check and Beyond

- For bounds check, ensure that absolute negative value of offset is less than segment size
- The different registers for storing these values are called **segment registers**
- **Can we make this more memory efficient?**
  - **Can we share some segments of the memory?**
  - Code sharing is still in use in many systems
  - Hardware introduce support in the form of protection bits
  - Code segment can be set to read only (Hardware can check if address is within bounds and permissible)

Coarse-grained vs Fine-grained

- **Coarse-grained**: Memory management which takes only few segments into consideration
    - Chops memory into large sized segments
- **Fine-grained:** Address space consisted of large number of smaller sized segments
    - This requires further hardware support
    - **Segment table** stored in-memory

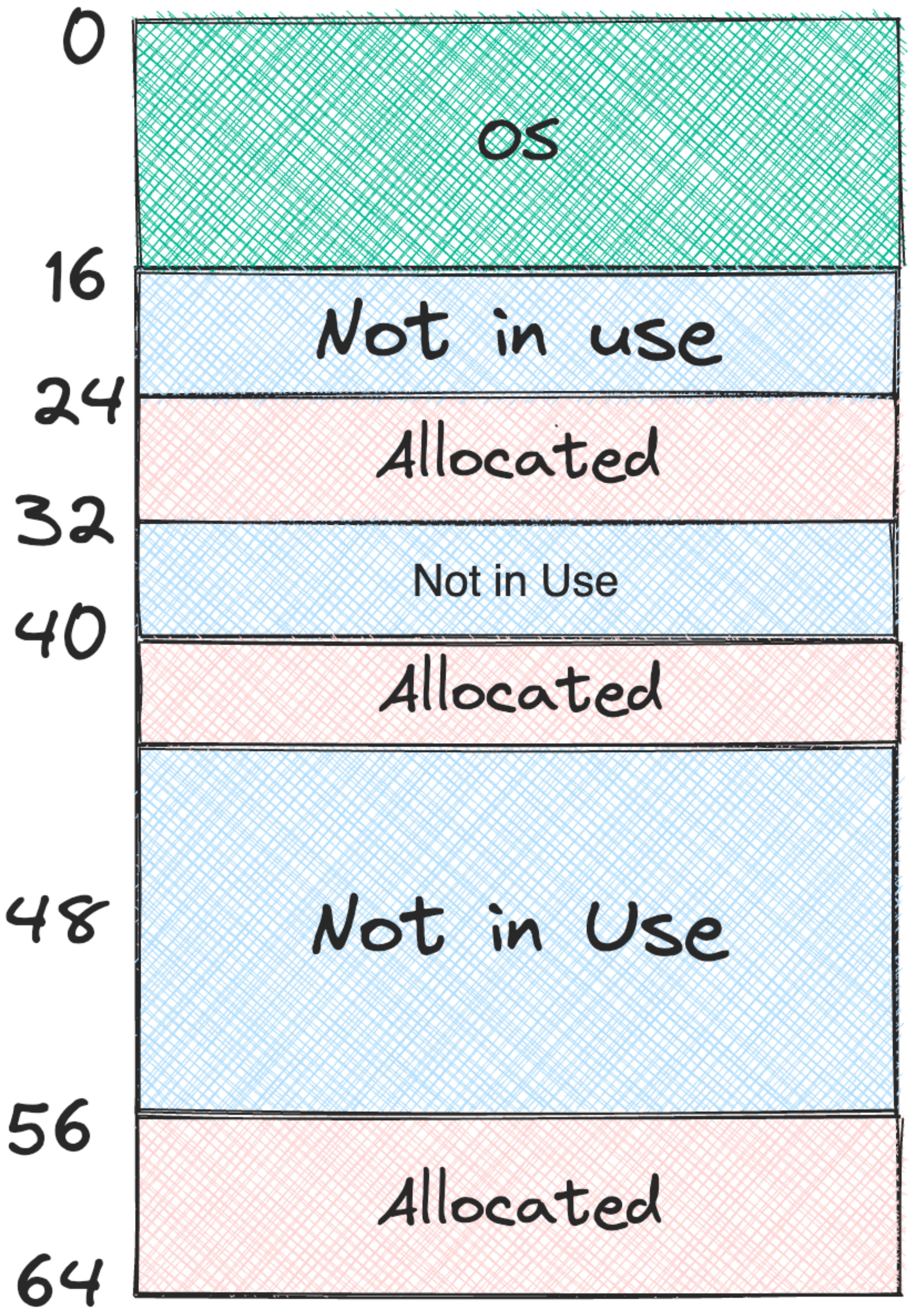
Some Challenges/Issues

- Context-switch:
  - OS must save segment registers and restore them - Each process has own VA
- Free space management:
  - OS should be able find physical memory for its segments
  - Each process has number of segments and each segment could be different size
  - Results in **External Fragmentation!**

Source: imageflip.com

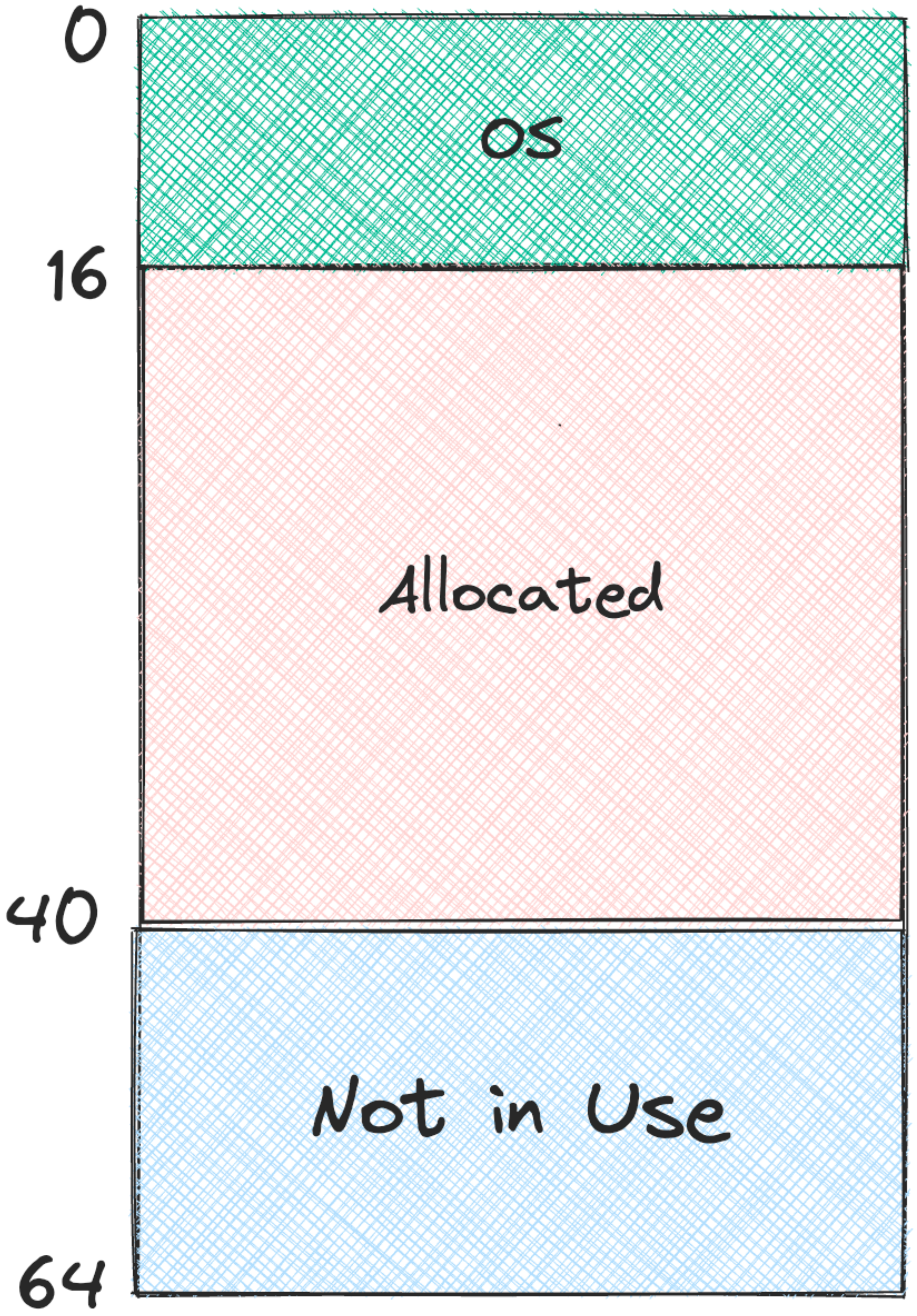
External Fragmentation

- Physical memory quickly becomes full of little holes
- Hard to allocate new segments
- Consider process wishes to allocate a 20 KB segment - 24 KB is free but not in a contiguous space!!
    - Can we come up with a compact version of this?



Compacted Version

- Seems like a more easy solution - OS could stop the running process
  - Copy data into a contiguous region
  - Change segment values to point to new region
  - Now there is larger memory
- Process is very **memory intensive!**



SCAN ME

Thank you

Course site: karthikv1392.github.io/cs3301_osn
Email: karthik.vaidhyanathan@iiit.ac.in
Twitter: @karthi_ishere