# CS3.301 Operating Systems and Networks

## Memory Virtualization

**Karthik Vaidhyanathan**

https://karthikvaidhyanathan.com

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

SERC
Software Engineering Research Centre

# Acknowledgement

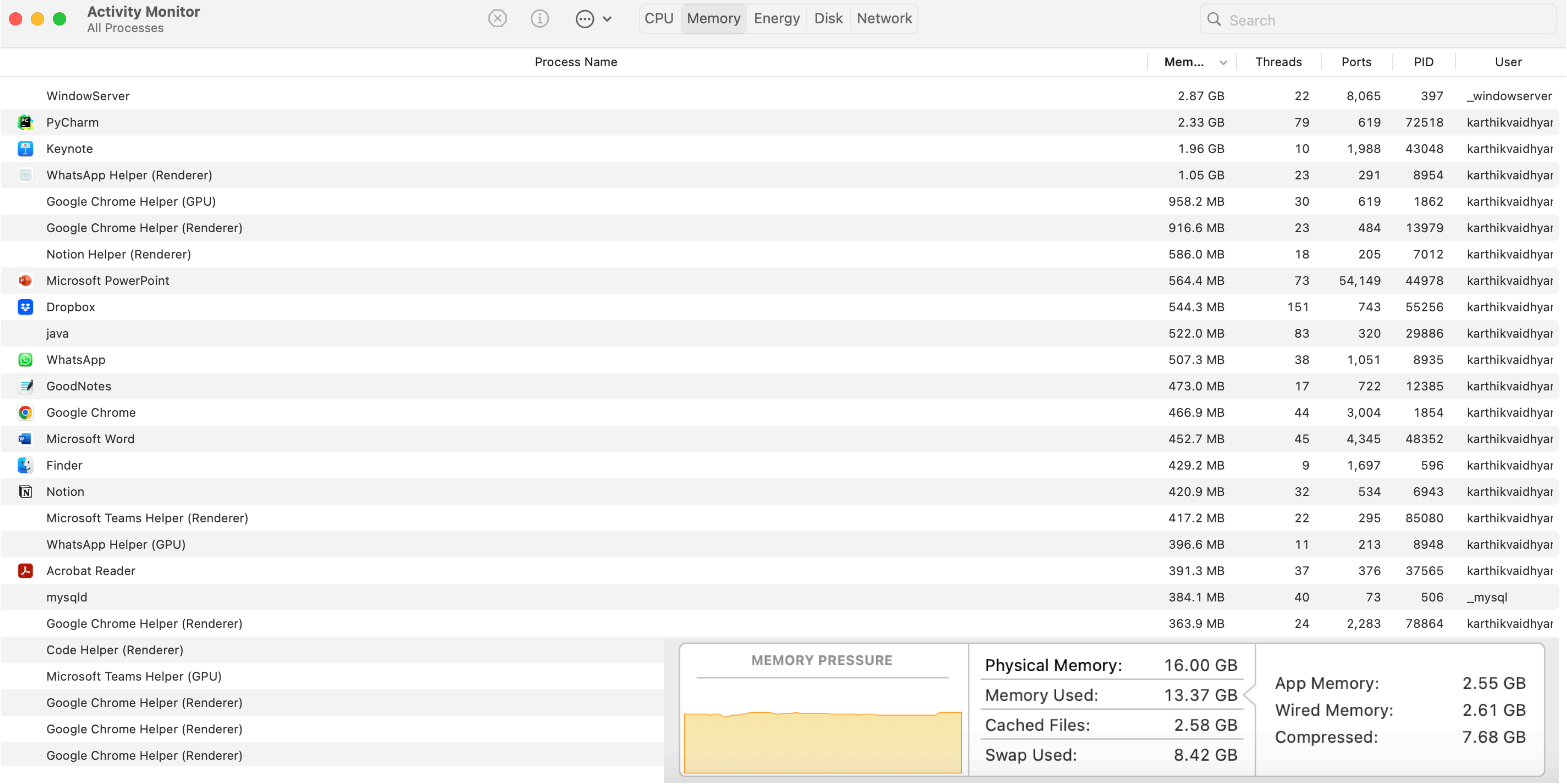
The materials used in this presentation have been gathered/adapted/generate from various sources as well as based on my own experiences and knowledge -- Karthik Vaidhyanathan

Sources:

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

# Many processes run at the same time!

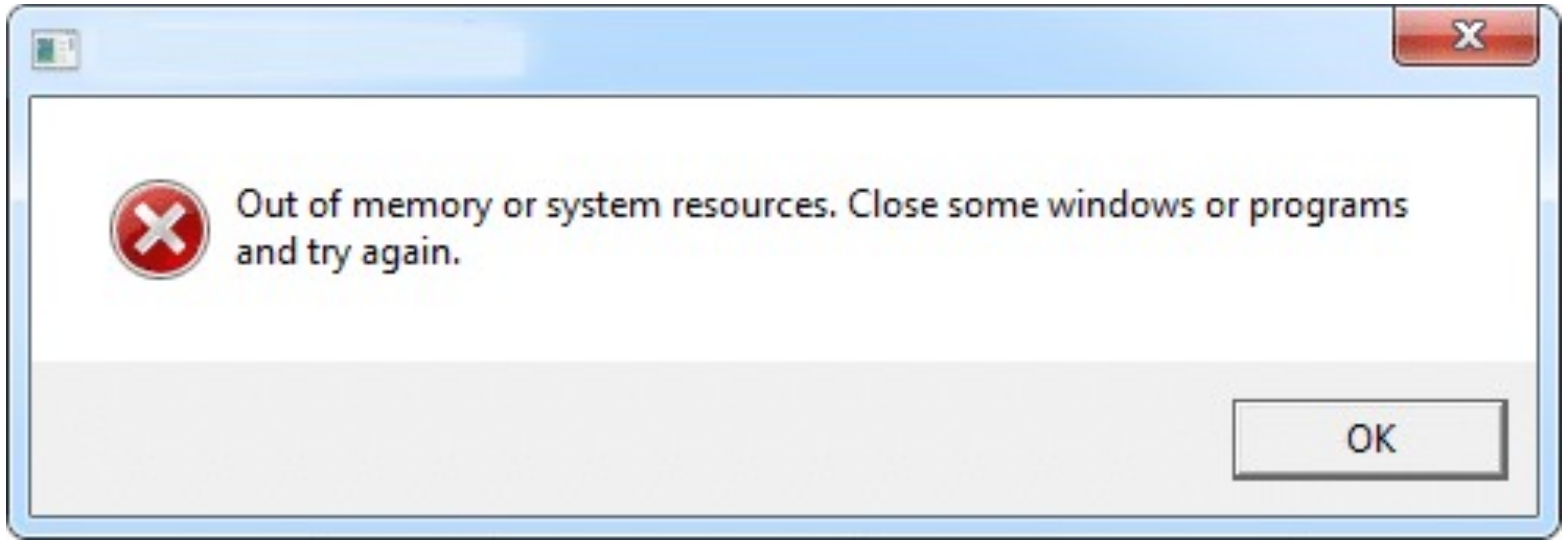
- What about Memory? Do we have enough Memory?

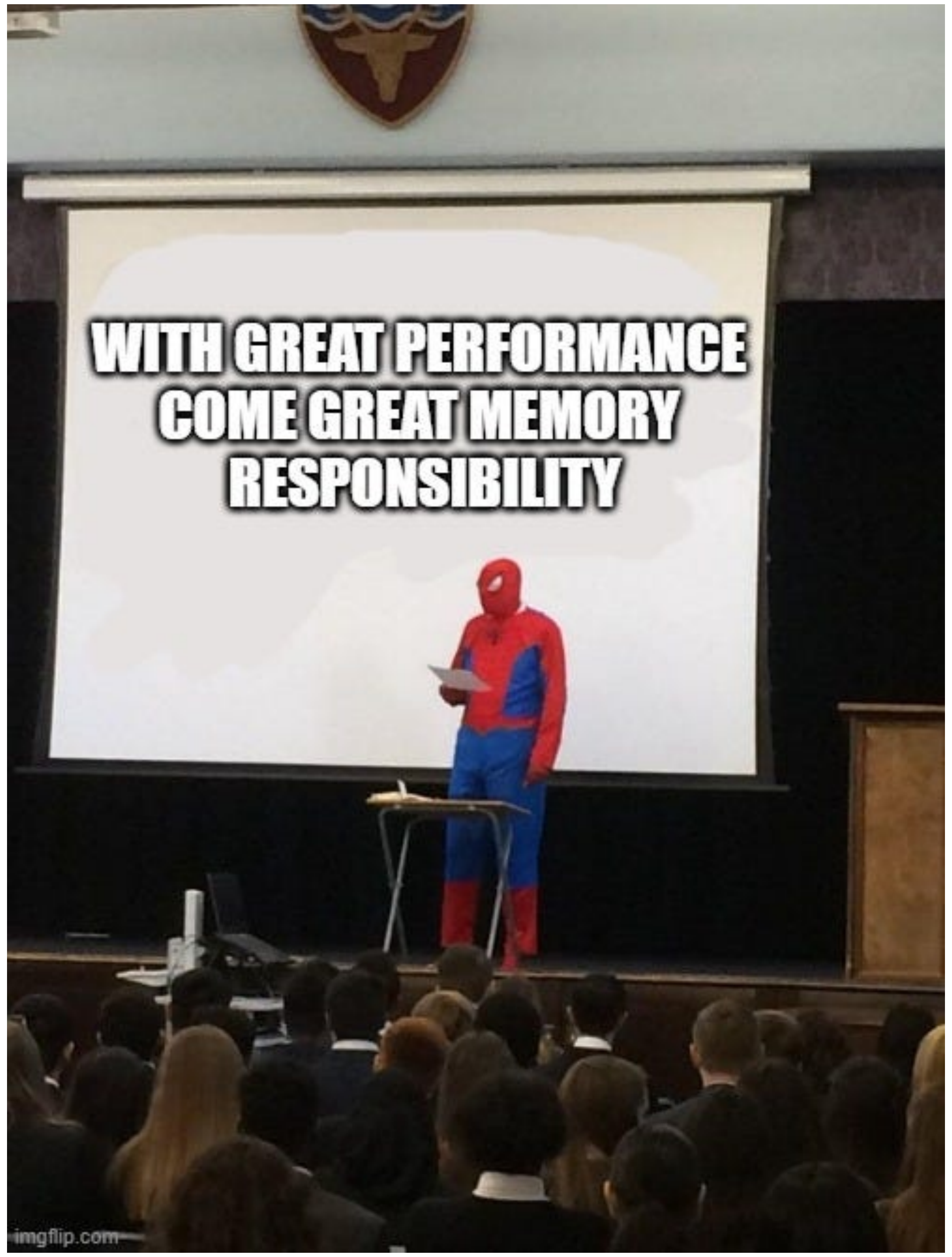



| Process Name | Mem... | Threads | Ports | PID | User |
|---|---|---|---|---|---|
| WindowServer | 2.87 GB | 22 | 8,065 | 397 | _windowserver |
| PyCharm | 2.33 GB | 79 | 619 | 72518 | karthikvaidhyar |
| Keynote | 1.96 GB | 10 | 1,988 | 43048 | karthikvaidhyar |
| WhatsApp Helper (Renderer) | 1.05 GB | 23 | 291 | 8954 | karthikvaidhyar |
| Google Chrome Helper (GPU) | 958.2 MB | 30 | 619 | 1862 | karthikvaidhyar |
| Google Chrome Helper (Renderer) | 916.6 MB | 23 | 484 | 13979 | karthikvaidhyar |
| Notion Helper (Renderer) | 586.0 MB | 18 | 205 | 7012 | karthikvaidhyar |
| Microsoft PowerPoint | 564.4 MB | 73 | 54,149 | 44978 | karthikvaidhyar |
| Dropbox | 544.3 MB | 151 | 743 | 55256 | karthikvaidhyar |
| java | 522.0 MB | 83 | 320 | 29886 | karthikvaidhyar |
| WhatsApp | 507.3 MB | 38 | 1,051 | 8935 | karthikvaidhyar |
| GoodNotes | 473.0 MB | 17 | 722 | 12385 | karthikvaidhyar |
| Google Chrome | 466.9 MB | 44 | 3,004 | 1854 | karthikvaidhyar |
| Microsoft Word | 452.7 MB | 45 | 4,345 | 48352 | karthikvaidhyar |
| Finder | 429.2 MB | 9 | 1,697 | 596 | karthikvaidhyar |
| Notion | 420.9 MB | 32 | 534 | 6943 | karthikvaidhyar |
| Microsoft Teams Helper (Renderer) | 417.2 MB | 22 | 295 | 85080 | karthikvaidhyar |
| WhatsApp Helper (GPU) | 396.6 MB | 11 | 213 | 8948 | karthikvaidhyar |
| Acrobat Reader | 391.3 MB | 37 | 376 | 37565 | karthikvaidhyar |
| mysqld | 384.1 MB | 40 | 73 | 506 | _mysql |
| Google Chrome Helper (Renderer) | 363.9 MB | 24 | 2,283 | 78864 | karthikvaidhyar |
| Code Helper (Renderer) | | | | | |
| Microsoft Teams Helper (GPU) | | | | | |
| Google Chrome Helper (Renderer) | | | | | |
| Google Chrome Helper (Renderer) | | | | | |
| Google Chrome Helper (Renderer) | | | | | |

# Real View of Memory can be Messy!

## Managing it can be even further difficult

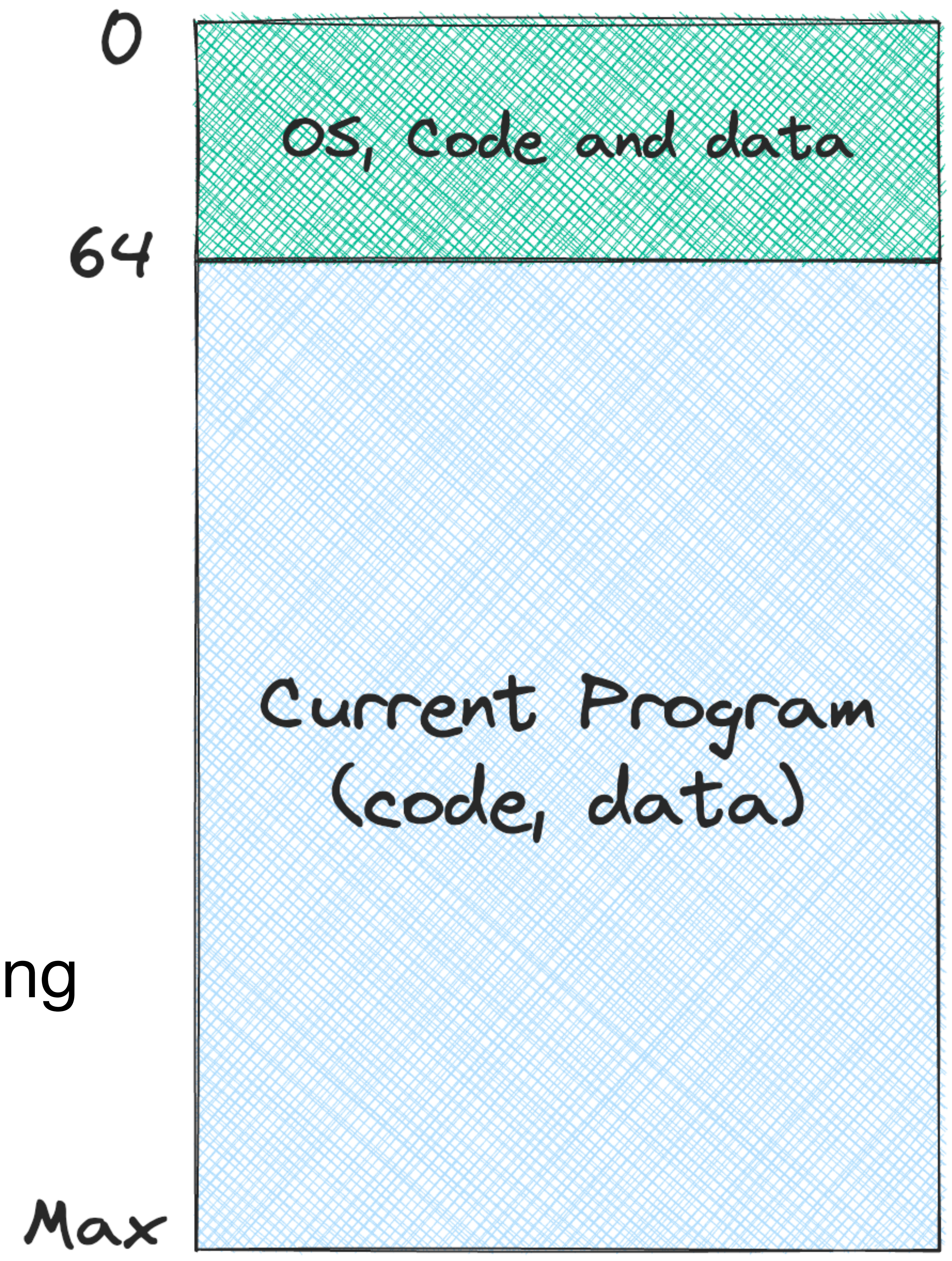



**Source:** Google Images

# Memory Virtualization

- Early days OS had just one program
- OS, its code and data resides in one part
- The running program, its code and data resides in one part
- Does it work today?
  - Today its about multiple processes
  - Run process for sometime save everything to disk, run next - **Problems?**
  - OS provides process virtualisation




**Only One running Program**

# Memory Virtualization: Why?

- We need to think about multiple processes
- Need to increase utilisation and efficiency
- Particularly useful in olden times when it costed millions of dollars for machines
- Soon came era of time sharing
- Batch computing was not anymore appreciated
- Instead of saving in the disk, can we keep the process on disk itself?

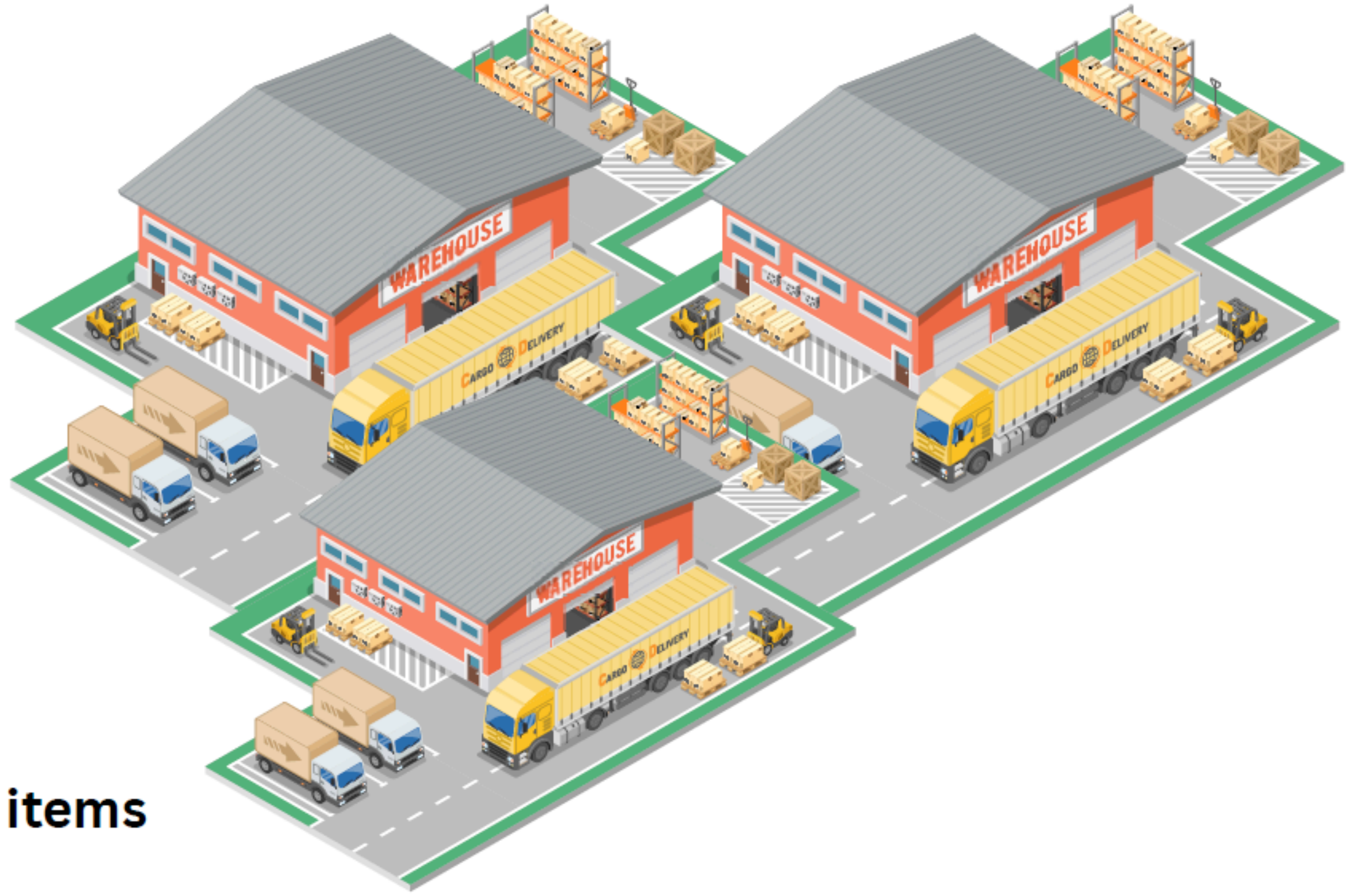
# An Analogy

## Onsite Shopping

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

## Online Shopping

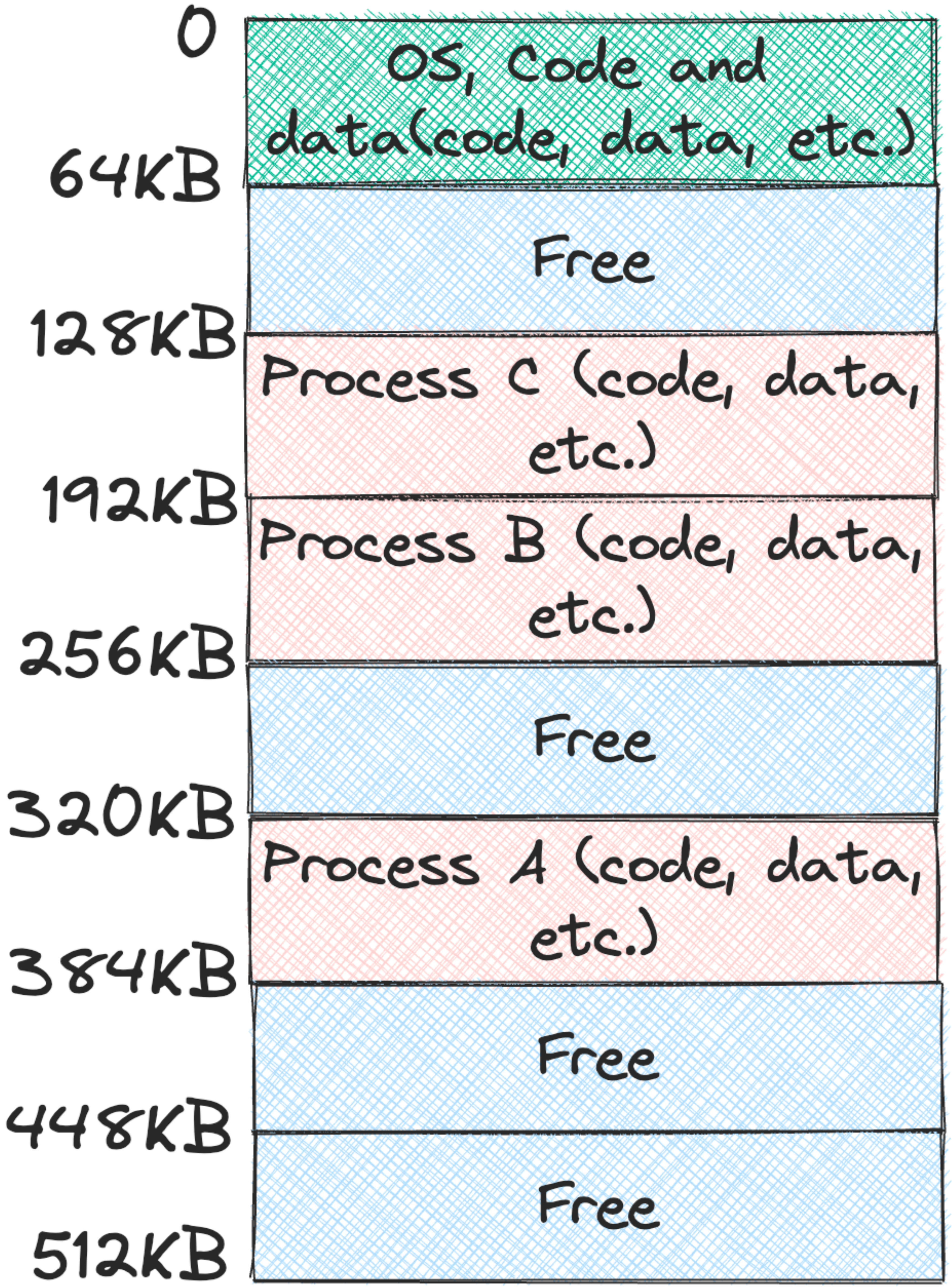





Every Users feel that they have access to infinite items

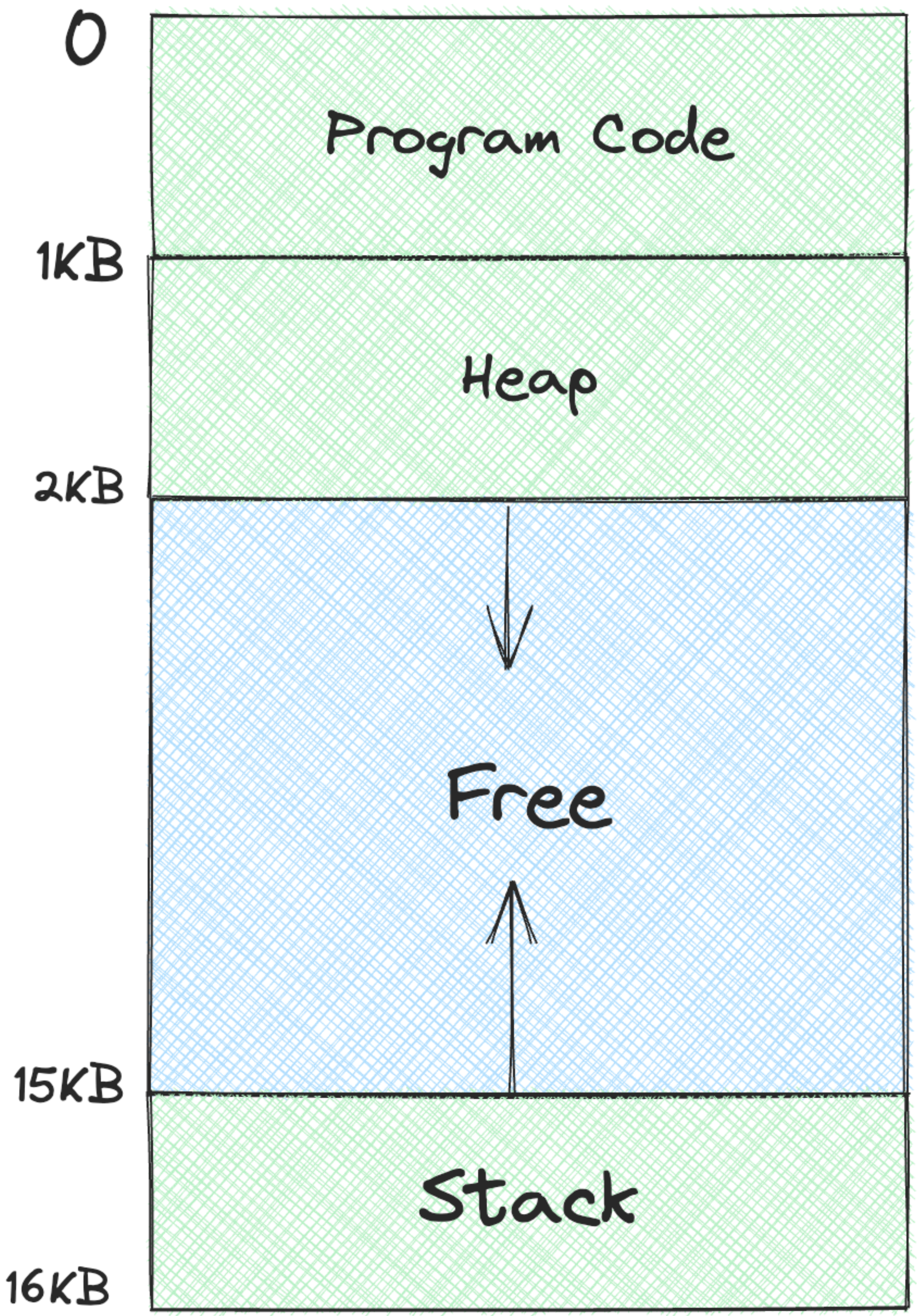
# Keep Process in the Memory

- Each process is given a dedicated location
- There are multiple free spaces where process can be added
    - Main challenge: We don't want any process to read any other process data
    - Real life OS has 100s of process that will be running
    - Giving control to user may make it hard



# Abstraction: Virtual Address Space

- OS creates easy to use abstraction of the physical space
- Address space (Memory image of process)
  - Program Code (and static data)
  - Heap - Dynamic memory allocations (malloc)
  - Stack - Function calls during runtime
  - The stack and heap grow during runtime
- Every process assumes that it has access to large block of memory from 0 to MAX
- CPU issues loads and stores to virtual addresses

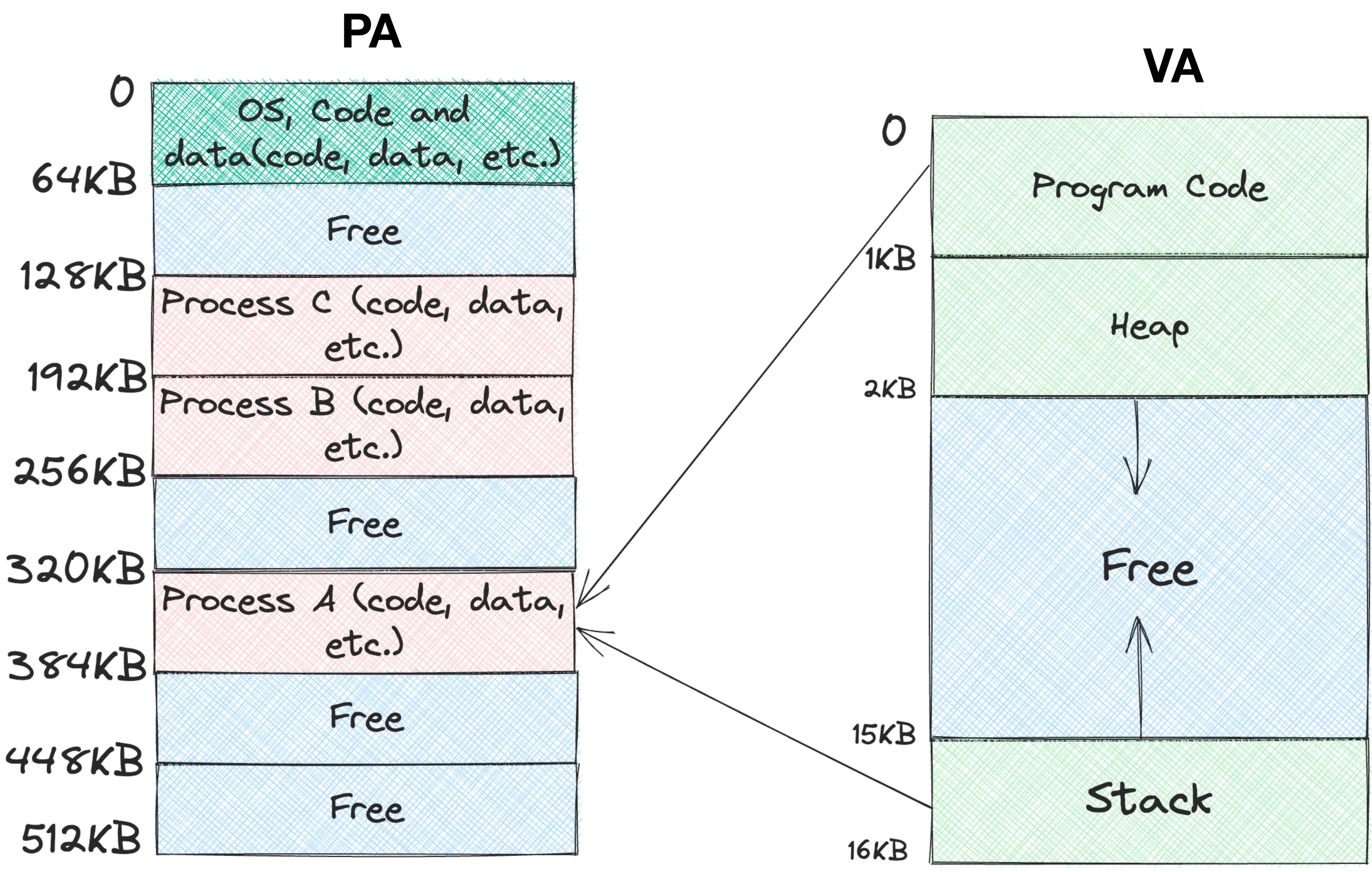


# There is only one physical memory

- How can OS build the abstraction of a private large address space on top of single physical memory?
  - There is only one physical memory, process feels has it has its own starting at 0
  - When a process tries to load from a particular location, **K** (0)
    - OS with some hardware support ensures that the load doesn’t go to actual location
    - Rather to the physical address **Z** (320) - **Virtualization**

# How actual memory is reached?

- Address translation from virtual address (VA) to physical address (PA)
  - CPU loads/stores to VA but memory hardware access PA
- OS allocates memory and tracks the location of the process
- Translation is done by Memory Management Unit (MMU)
  - OS makes necessary information available



# 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 **Autonomic memory**
    - Once execution is done, compiler deallocates memory
- Static/global variables are allocated in executable

C Program Snapshot

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

# Memory Allocations and Deallocations

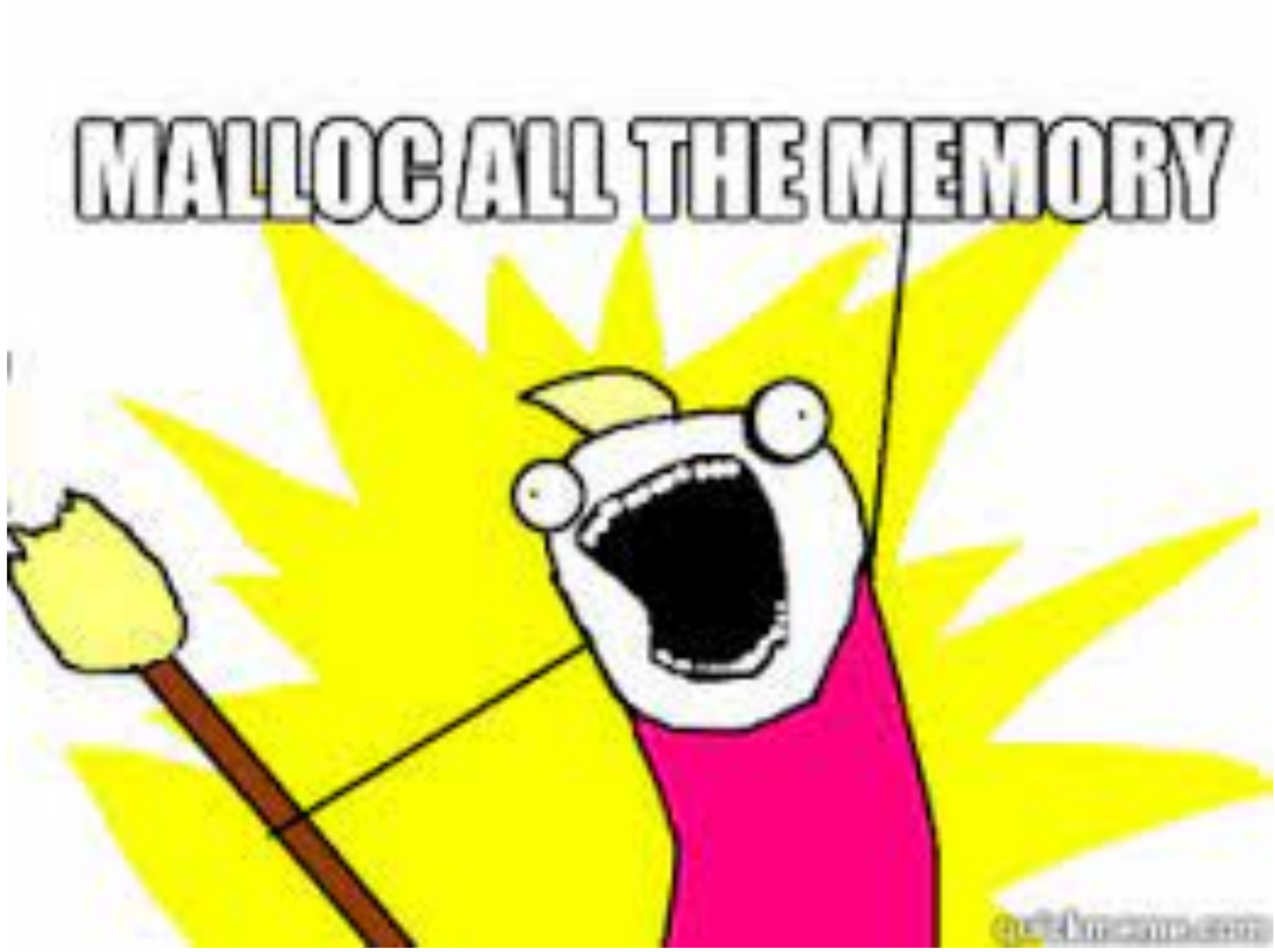
## Second type of Memory Allocation

- 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));
```

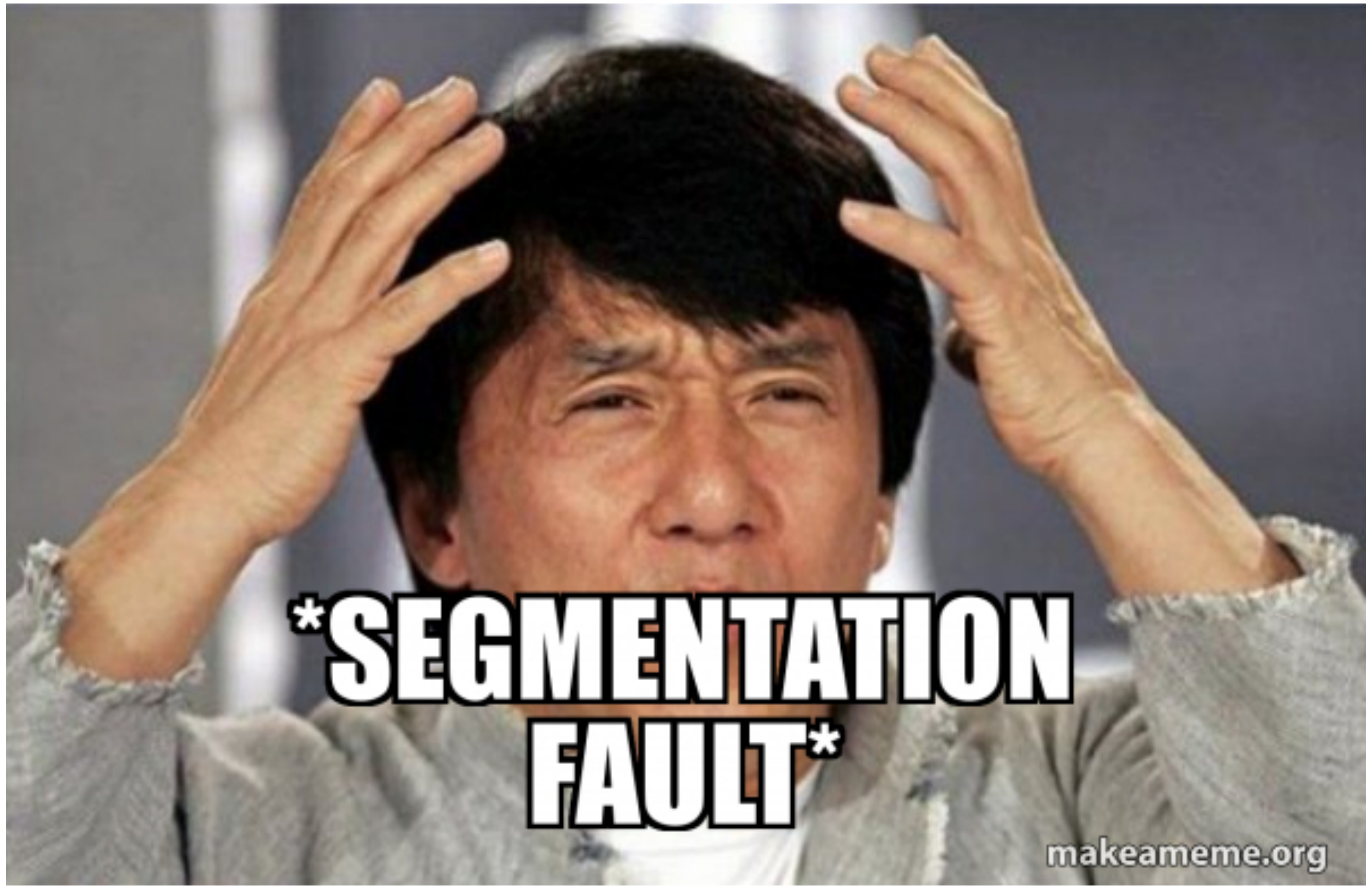
# Free() call

- 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

Strcpy on two strings

```
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

```
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

Strcpy on two strings

```
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

SCAN ME

# Thank you

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