

**Operating Systems:**  
**Chapter 3**  
**Memory-spaces**  
**CS400**

Week 5: 11<sup>th</sup> Feb

Spring 2020

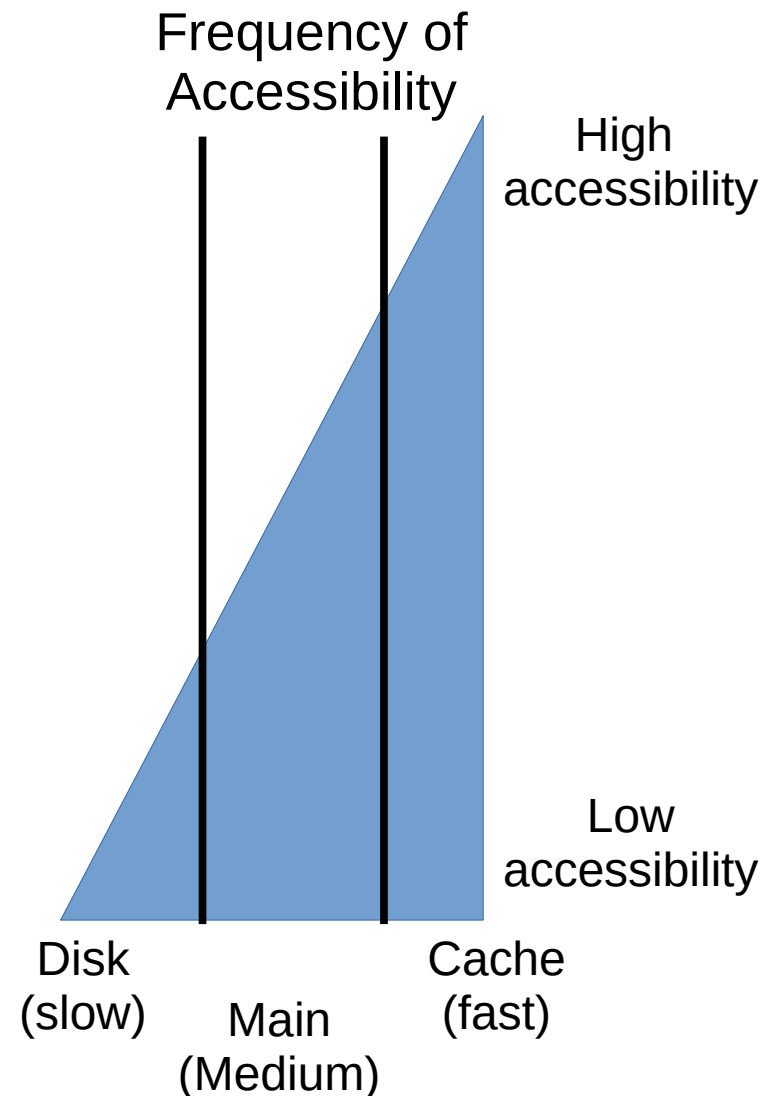
Oliver BONHAM-CARTER

# Exam 1

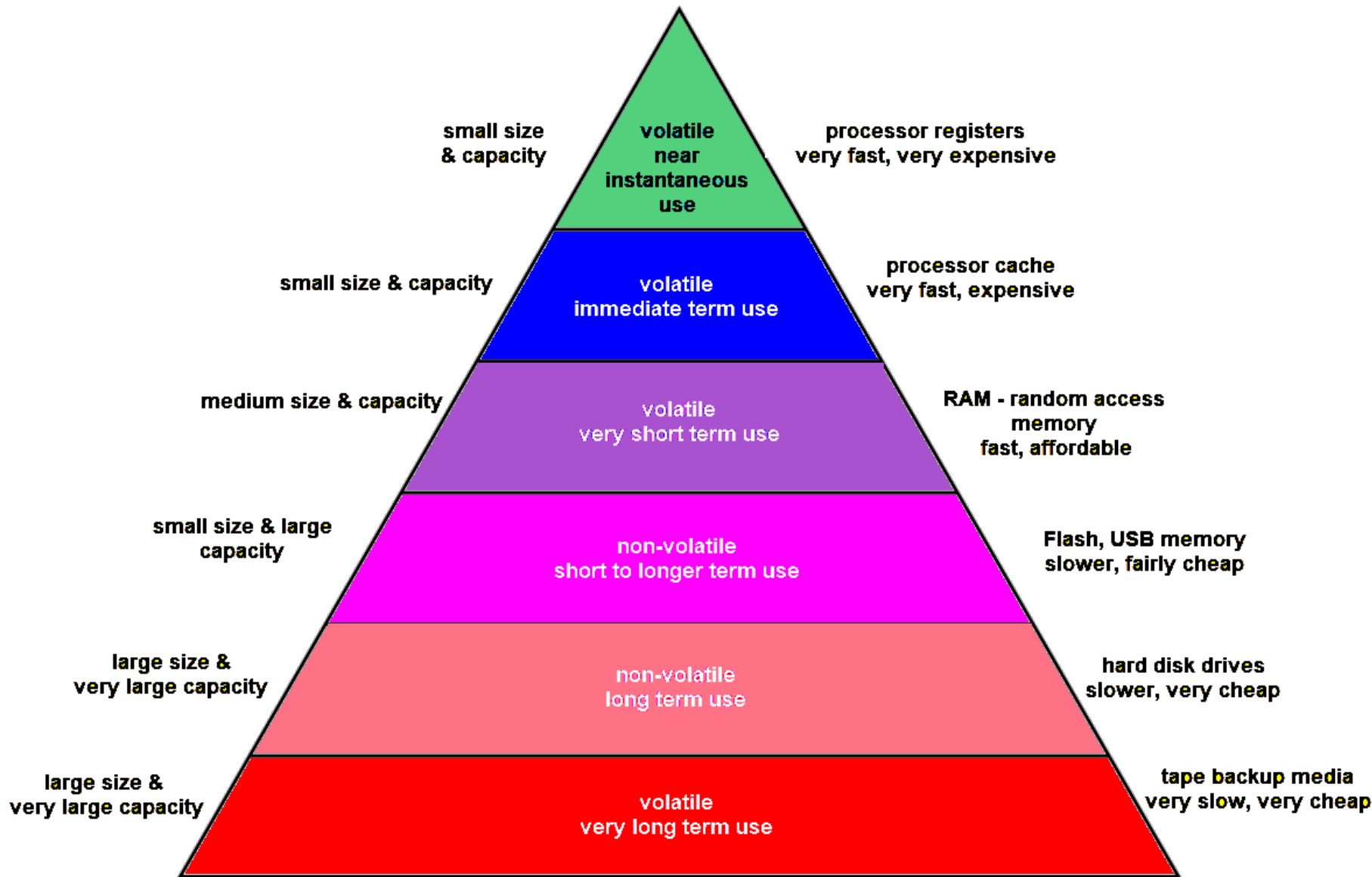
- 18<sup>th</sup> Feb (Tuesday) on Sakai during lab
- Closed note, closed book
- Multiple choice; short essay, code analysis and function
- Types of Questions:
  - Abstraction
  - Processes
  - Threads
  - User and Kernel spaces
  - Memory
- Know your slides, inspired by our textbook by Tanenbaum

# Memory Management Basics

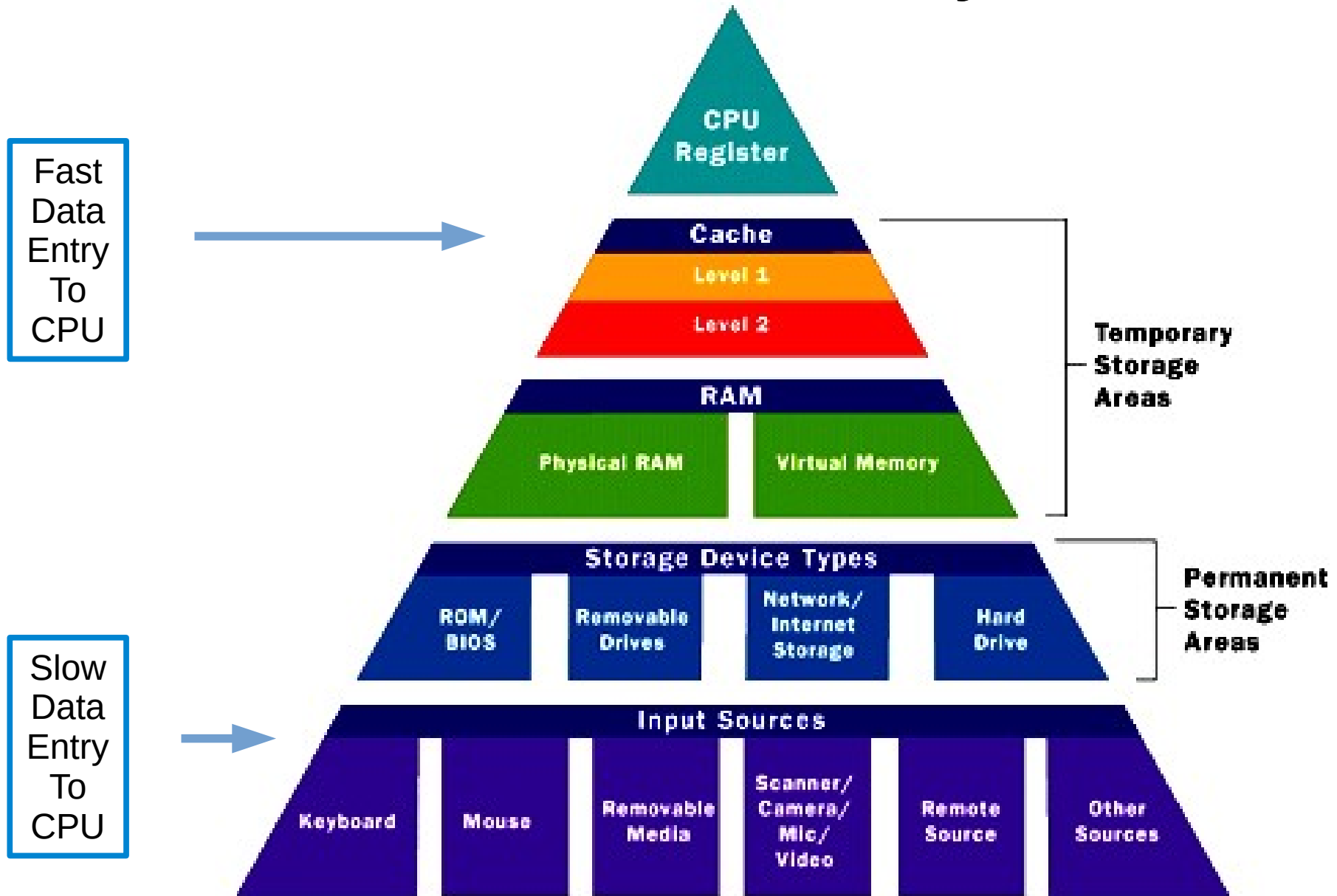
- Fast access to instructions in memory increases the overall speed of the software program.
- Limitations: Computers do not have unlimited RAM (read access memory).
  - Must make due with what we have!
- A computer's *memory hierarchy (memory spaces)*
  - **Cache memory** → very fast
  - **Main** → quick
  - **Disk** → slowest



# Levels of Memory Hierarchy



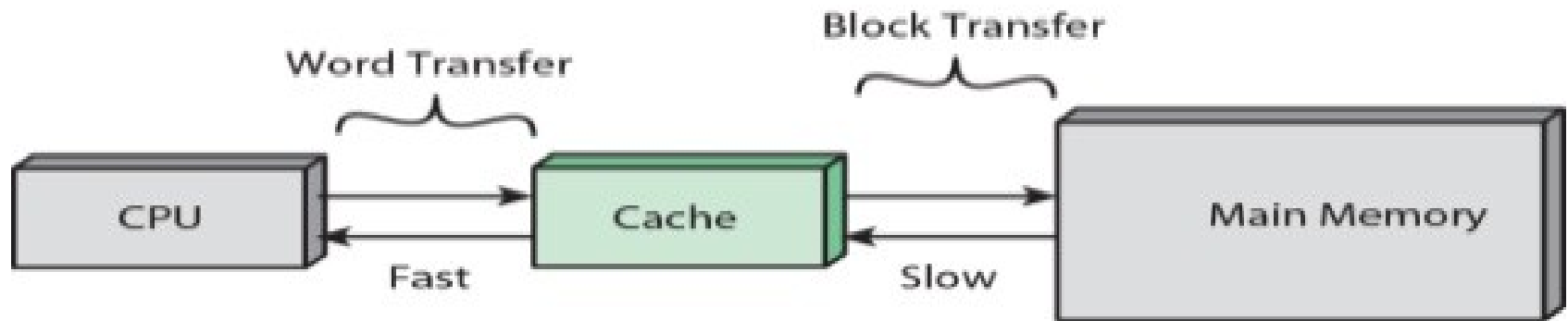
# Levels of Memory



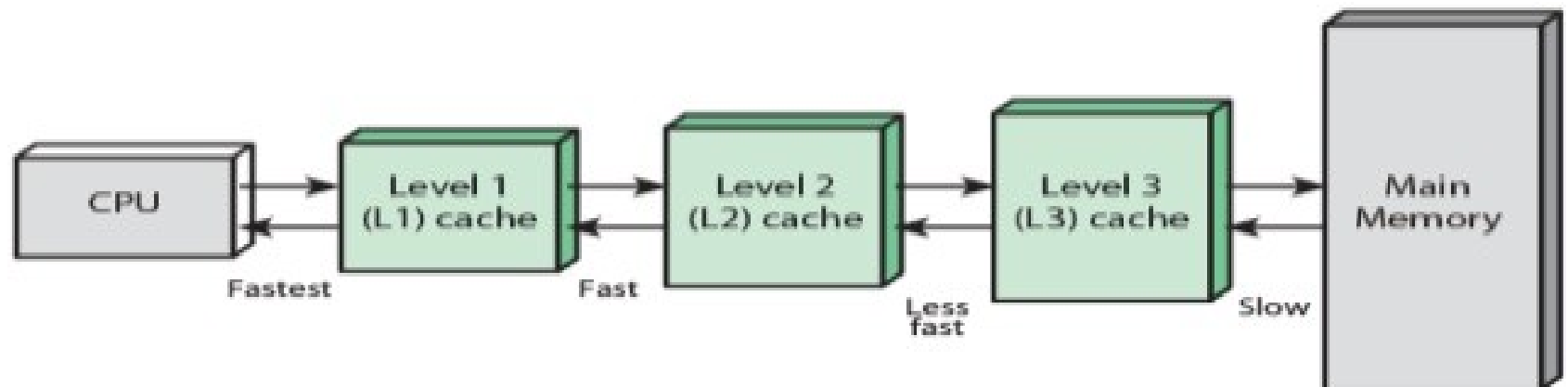
# Cache Memory

- Also called CPU memory, is RAM that a computer microprocessor can access more quickly than it can access regular RAM.
- Frequently re-referenced instructions
- Typically integrated directly with the CPU chip
- Or placed on a separate chip that has a separate bus interconnect with the CPU.
- First checked by CPU when using memory

# Cache and Main Memory



(a) Single cache



(b) Three-level cache organization

# Levels of Cache (Part 1)

- **Level 1 (L1)** cache: very fast but relatively small, usually embedded in the processor chip (CPU)
  - Size: 8 Km to 128 Km.
- **Level 2 (L2)** cache not as fast as L1 but has more storage. It may be located on the CPU or on a separate chip or coprocessor with a high-speed alternative system bus interconnecting the cache to the CPU, so as not to be slowed by traffic on the main system bus.
  - Size: 512kb to 8Mb



# Levels of Cache (Part 2)

- **Level 3 (L3)** cache is typically specialized memory that works to improve the performance of L1 and L2.
  - May be much slower than L1 or L2, but has a larger size: to to 8Mb.
- In the case of multicore processors, each core may have its own dedicated L1 and L2 cache, but share a common L3 cache.
  - When an instruction is referenced in the L3 cache, it is typically elevated to a higher tier cache.

# Main Memory: RAM

- Silicon-based transistors and integrated circuits
- **Volatile:** able to rapidly change
- Primary storage, fast CPU memory
  - DRAM (dynamic random-access memory)
    - Main computer memory type
    - Memory eventually fades unless refreshed
    - Loss of power → erasure.
  - SRAM (static random-access memory)
    - Bistable latching circuitry (flip-flop)
    - Faster, expensive
    - Used for buffers (storage, CPU caches, etc)

# Main Memory: RAM

- **Non-volatile**

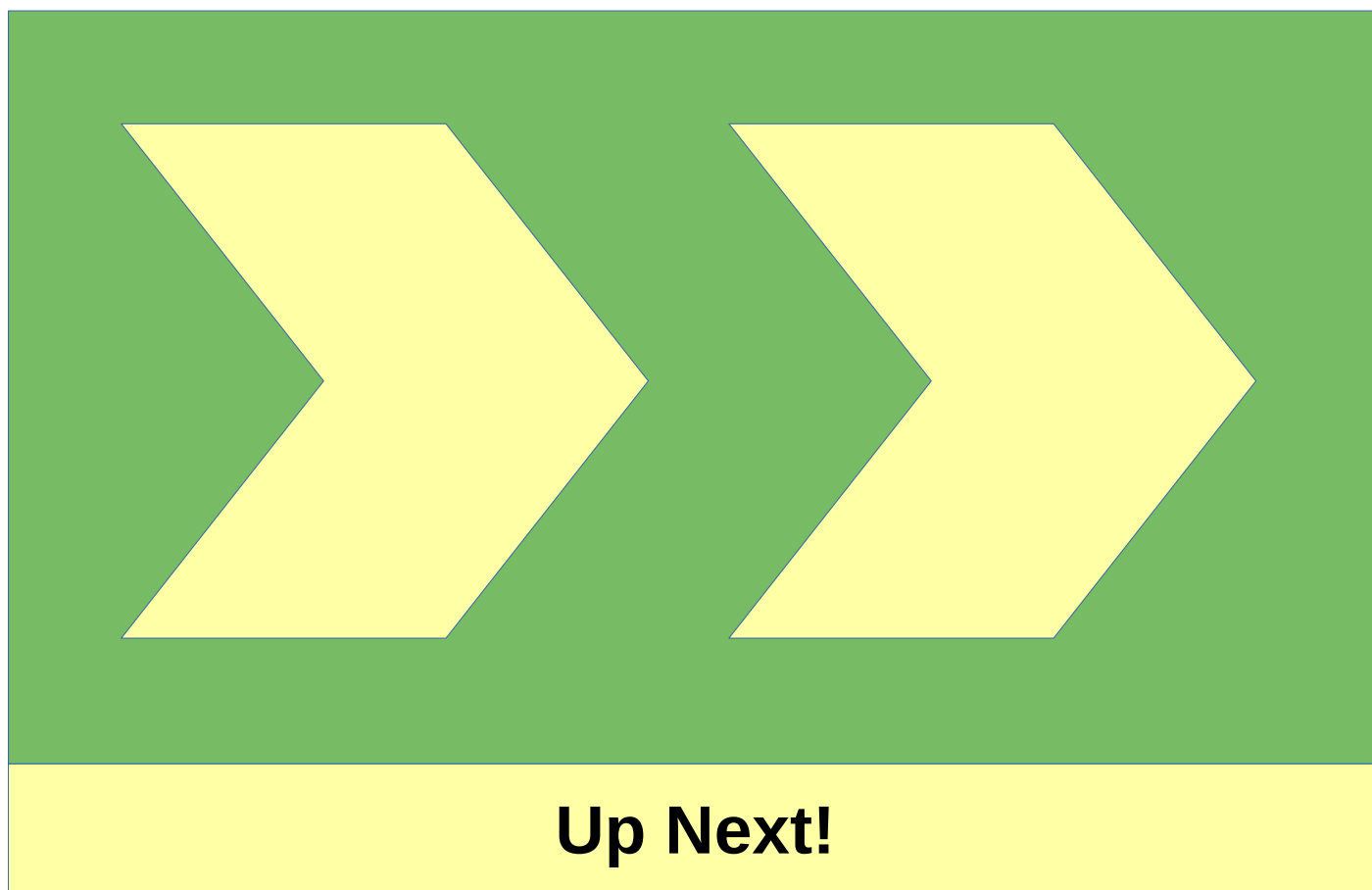
- Long-term, persistent storage
- Power-free; memory retained in absence of power.
- Flash and storage memory
- Read-only memory (ROM)
- PROM (programmable read-only memory)
- EPROM (erasable programmable read-only memory)
- Chip firmware

# Fun Fact!

- Systems having slower processors but larger caches tend to perform better than systems having faster processors but more limited cache space.

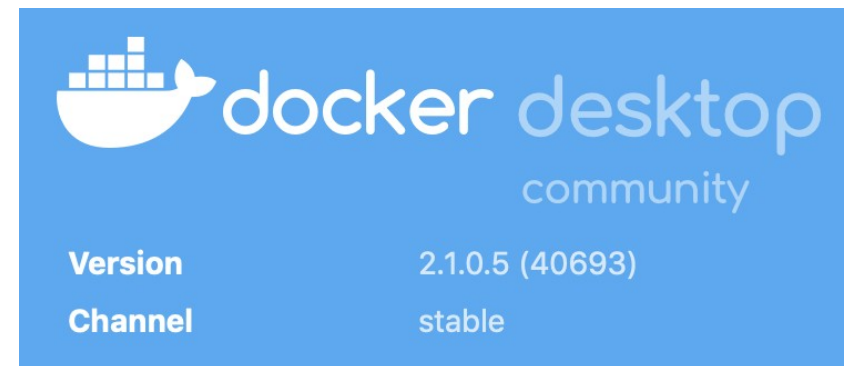


# Let's Code!



# Commands to Run From (Linux) Bash

- Build the container :
  - `docker build -t gccdev .`
- Run the container :
  - `docker run -it gccdev`
- Mount local drive and run container :
  - `docker run -it --mount type=bind,source=$PWD,target=/home/gccdev gccdev`



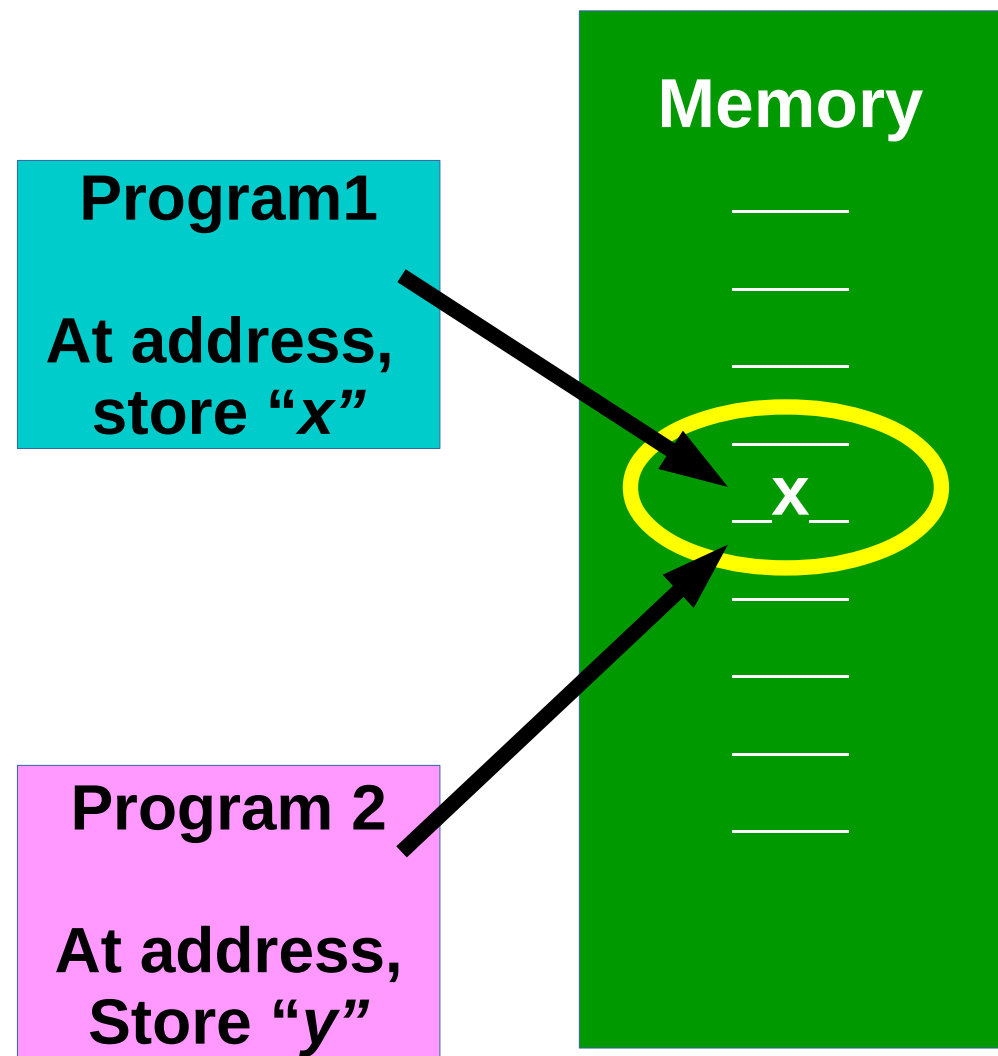
Note: the directory where you run this becomes your local directory in the container.

# No Abstraction

- No abstraction: having ***no** layer* to separate memory usage between processes
- Simple OSs: all running processes see the memory directly. Not a great solution for concurrent processes...

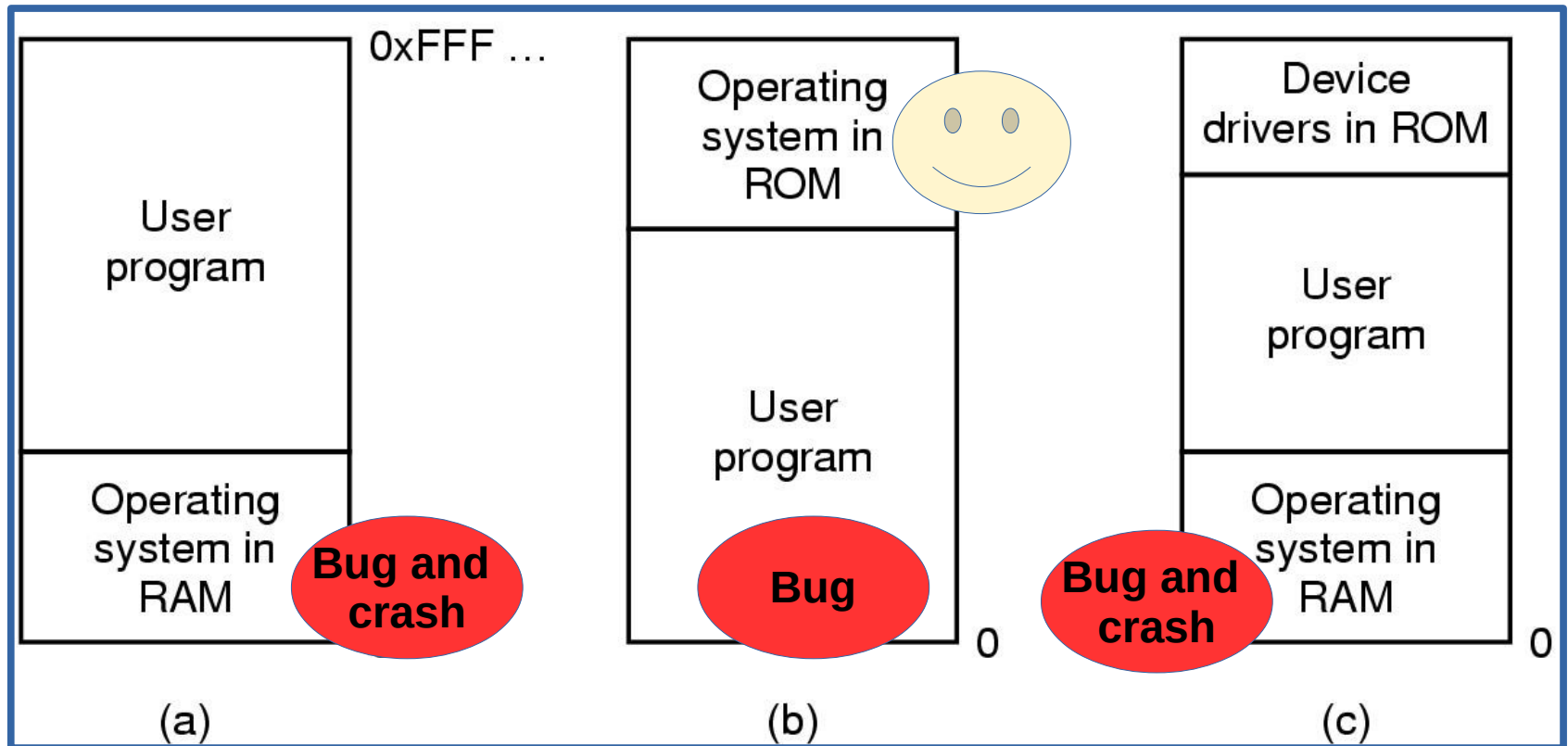
# No Memory Abstraction

- Two programs using the same memory address for recording data
- Is this good OS practice?
- Why/ why not?





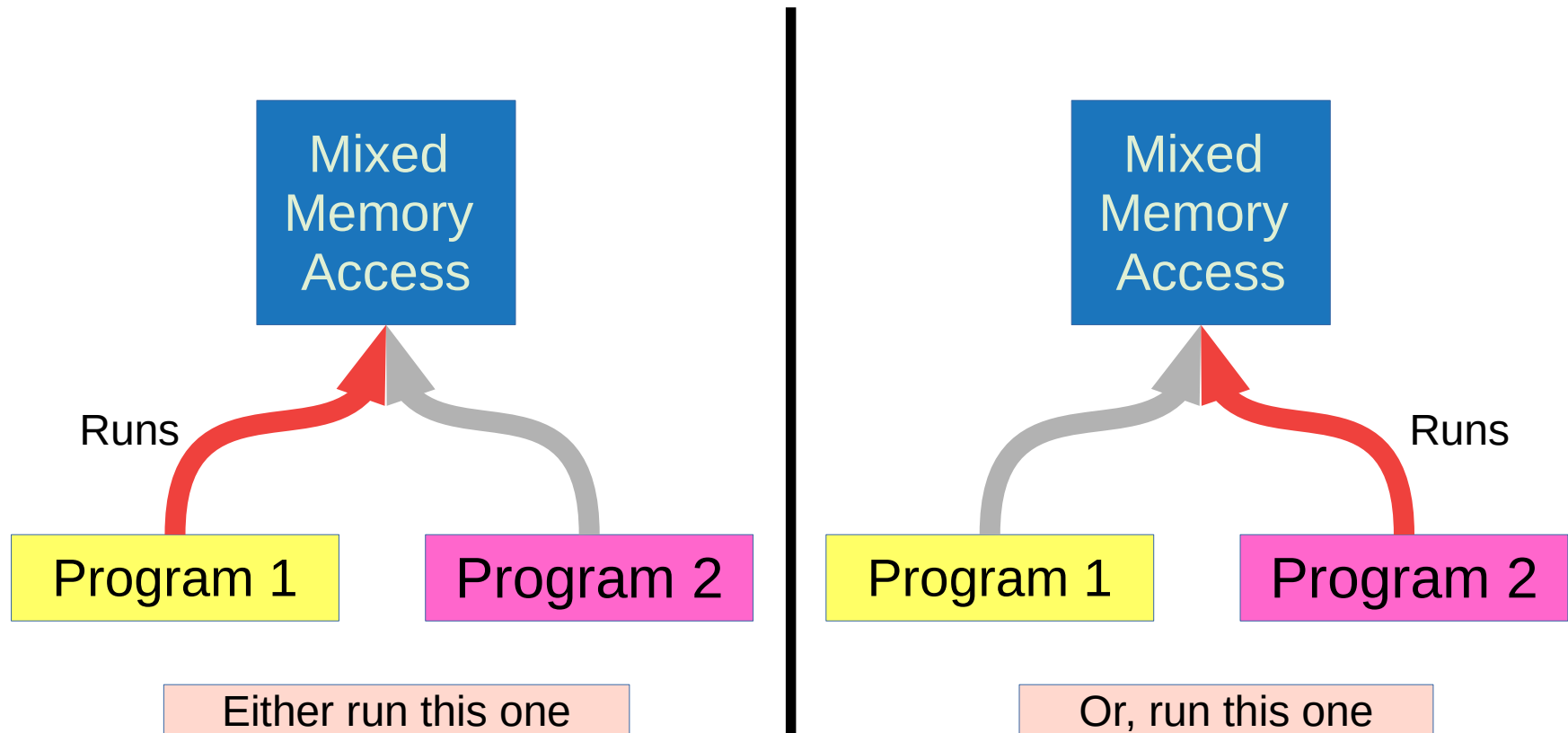
# Physical Memory Usage Options



- A) OS in bottom of memory in RAM (mainframes)
- B) OS in ROM in top of memory (handheld devices)
- C) Device drivers in ROM and rest of system in RAM (early PCs)
- **A and C could crash with bugs introduced in user-programs.**

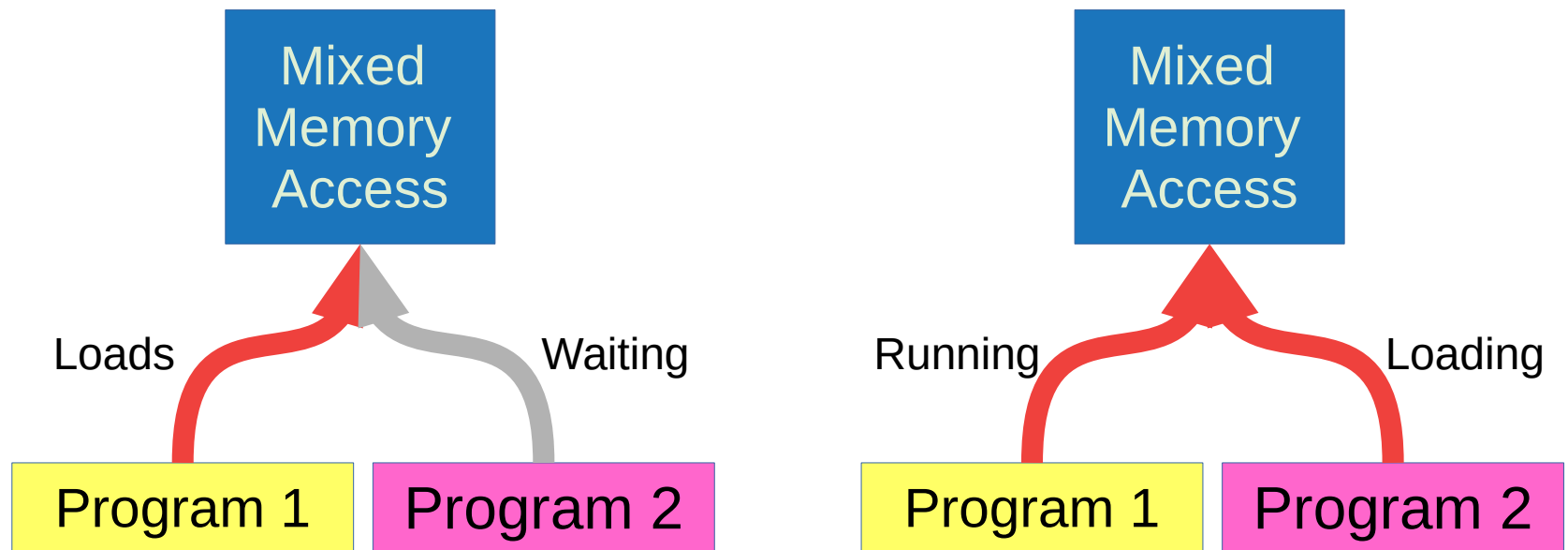
# Running Safely Without Memory Abstraction: **Swapping**

Only one of the programs is running at a time and there is no memory sharing.

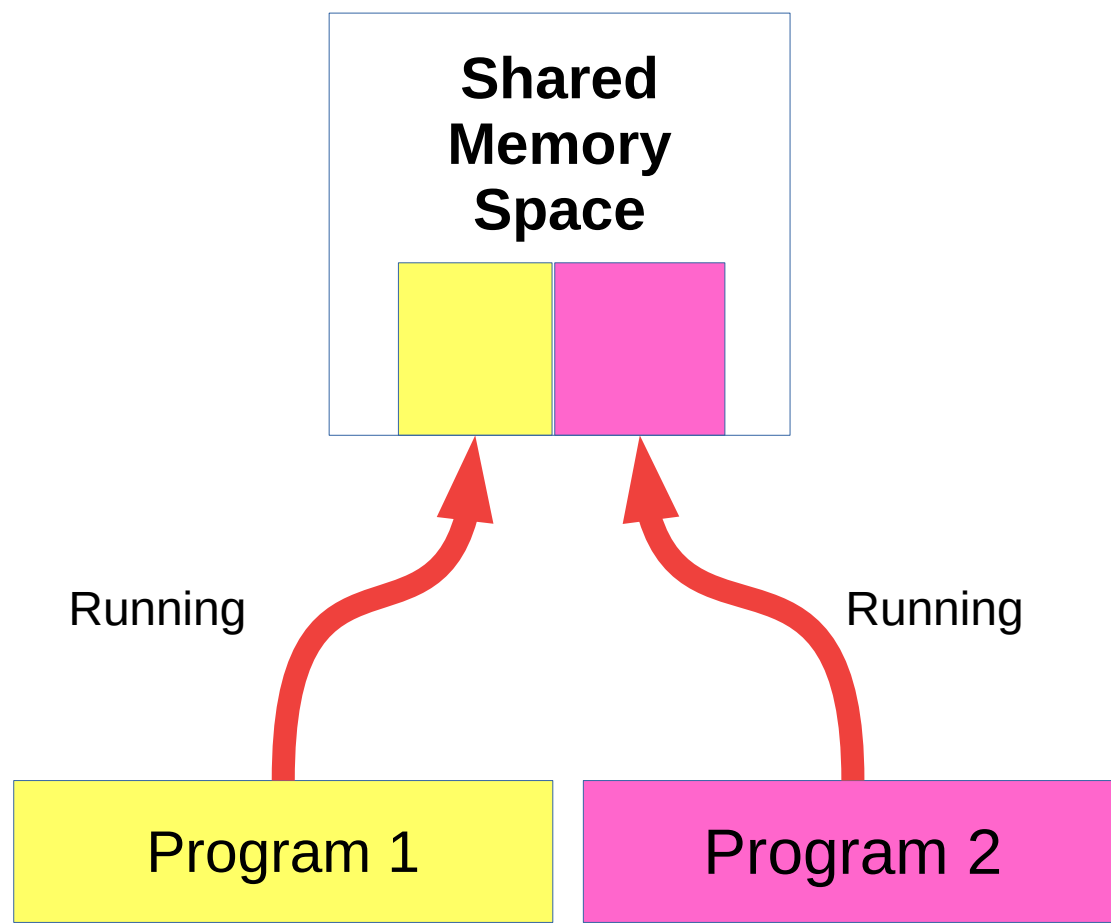


# Running *UNSaferly* Without Memory Abstraction

We will try to run two programs  
together where memory will be shared

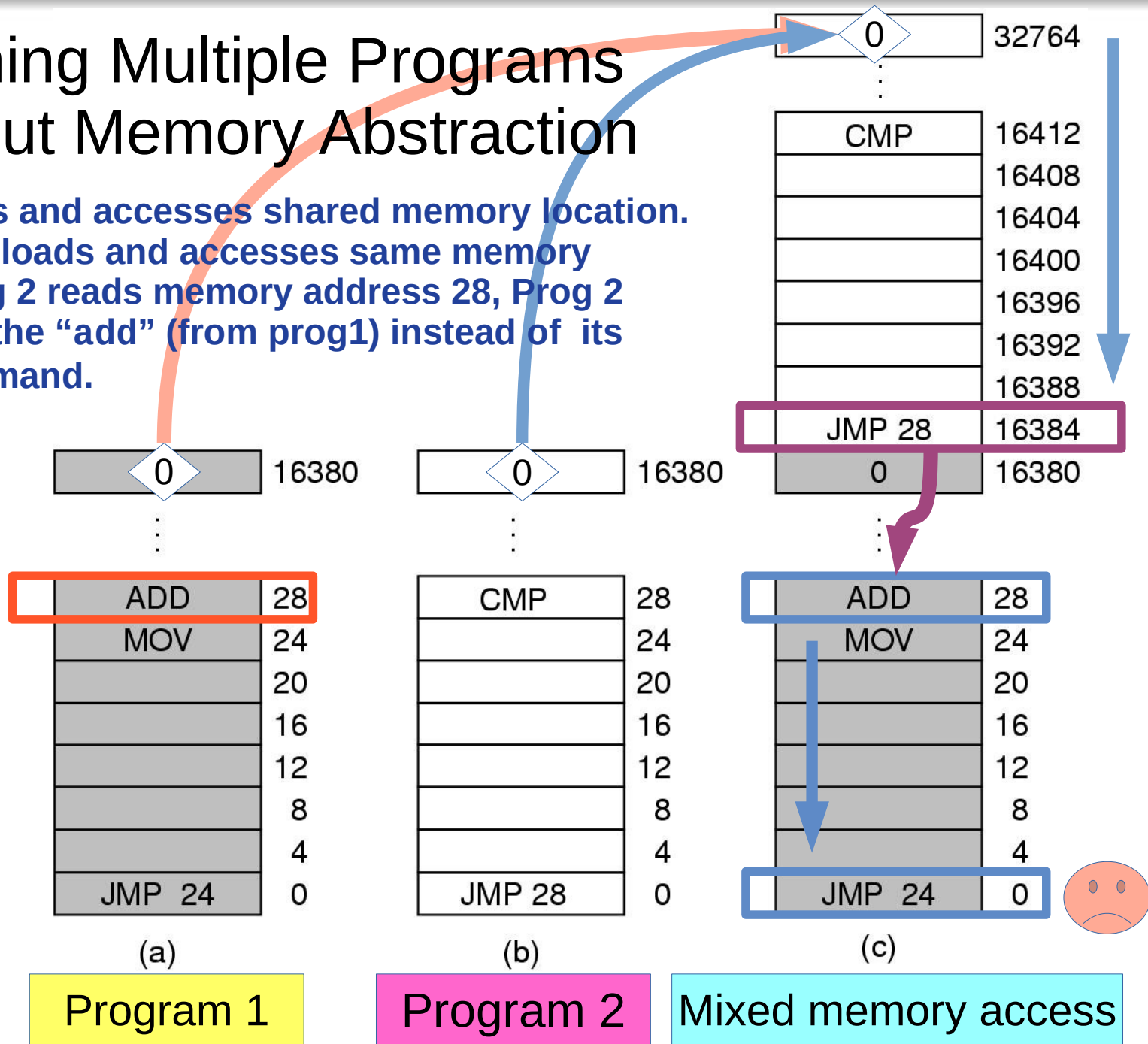


# Both programs are Running in the Same Memory Space



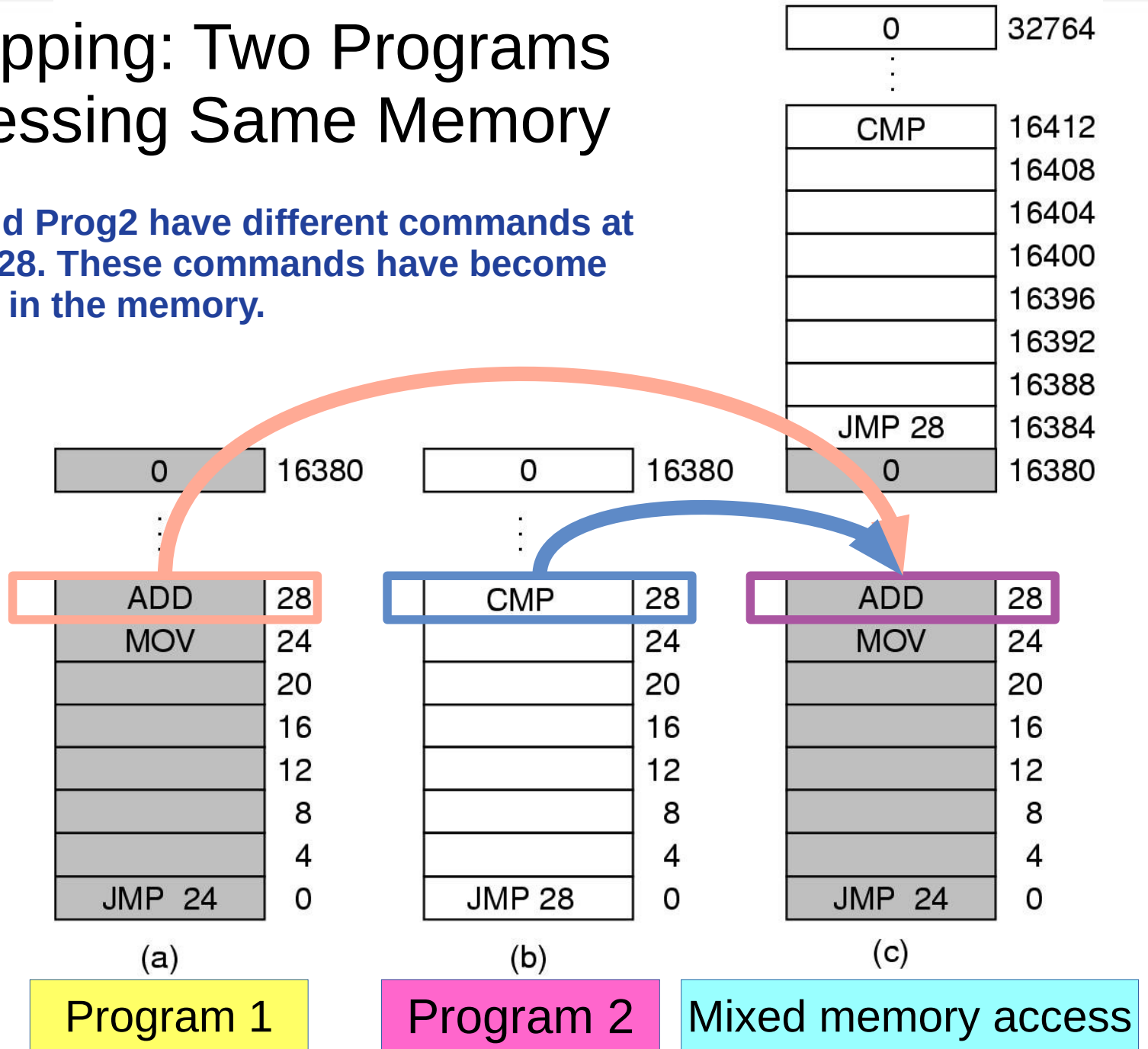
# Running Multiple Programs Without Memory Abstraction

Prog1 loads and accesses shared memory location. Prog2 then loads and accesses same memory space. Prog 2 reads memory address 28, Prog 2 processes the "add" (from prog1) instead of its "cmp" command.



# Swapping: Two Programs Accessing Same Memory

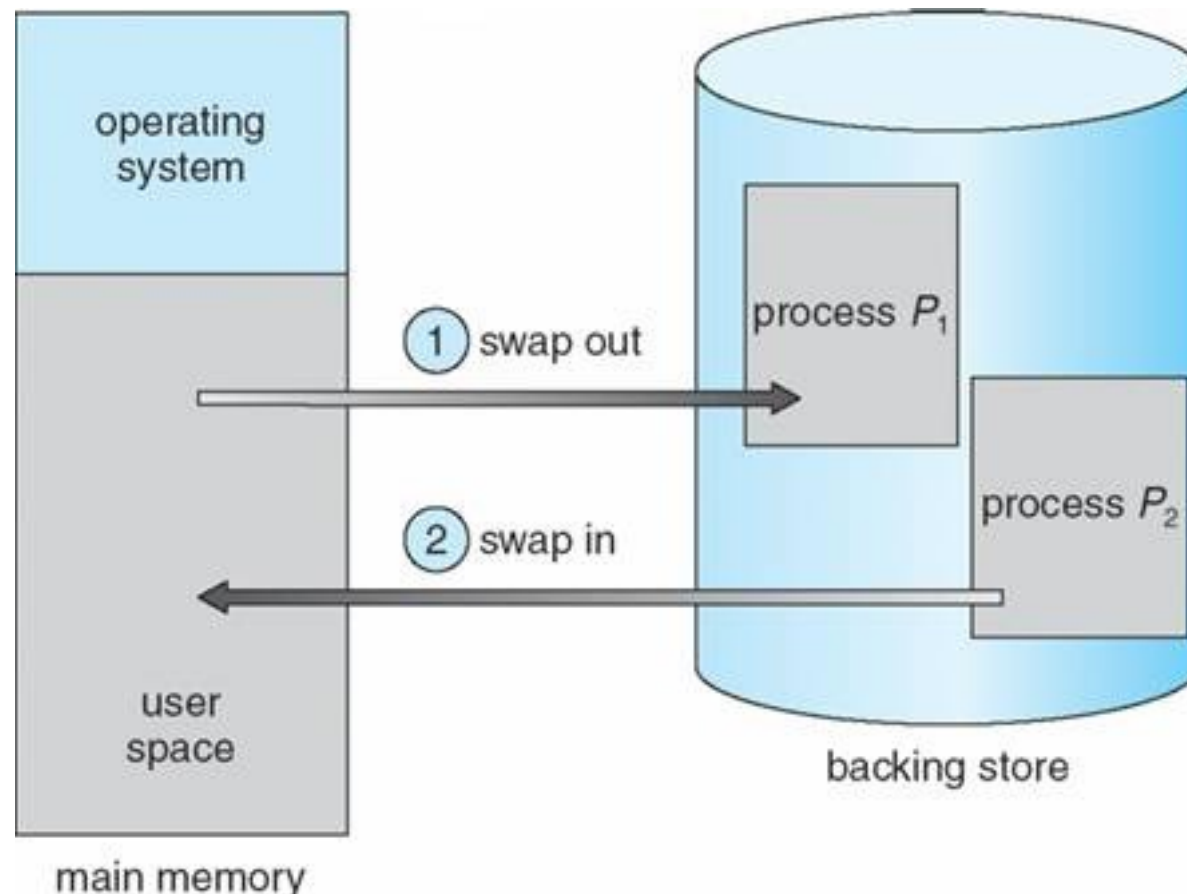
Prog1 and Prog2 have different commands at address 28. These commands have become muddled in the memory.



## Simple Solution For No Abstraction

- If one running program changes the memory cells of another then a system crash
- How to avoid overwriting when no abstraction available?
- Solution: *Swapping* enables abstraction: running several programs concurrently
  - Save all data of a program in file
  - Flush out memory cells, load file each program into memory cells one at a time, run separately.
  - Idle processes stored on disk

# Swapping

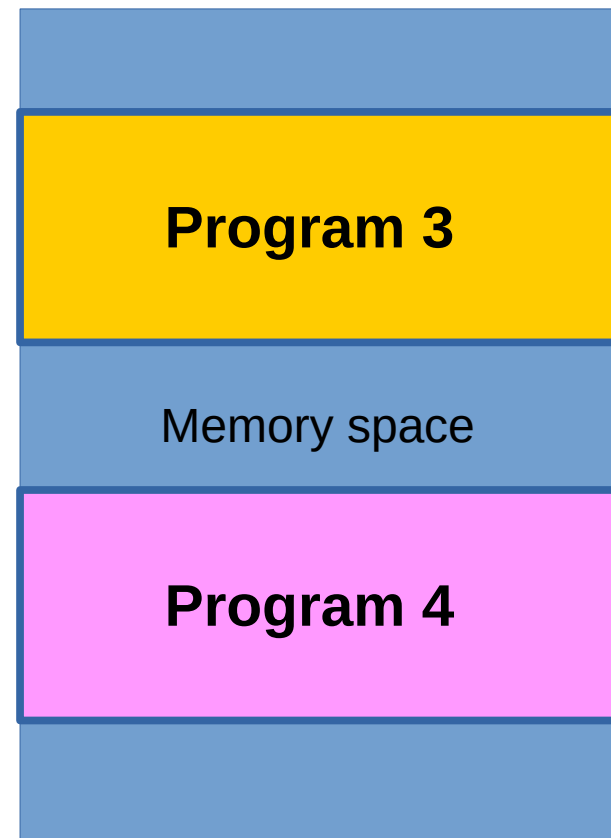
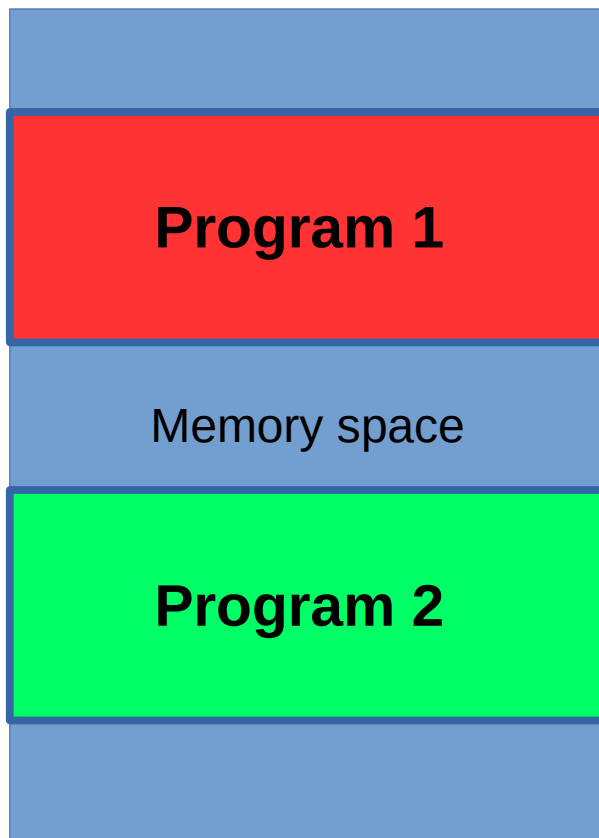


- Major part of swap time is transfer time; directly proportional to the amount of memory swapped



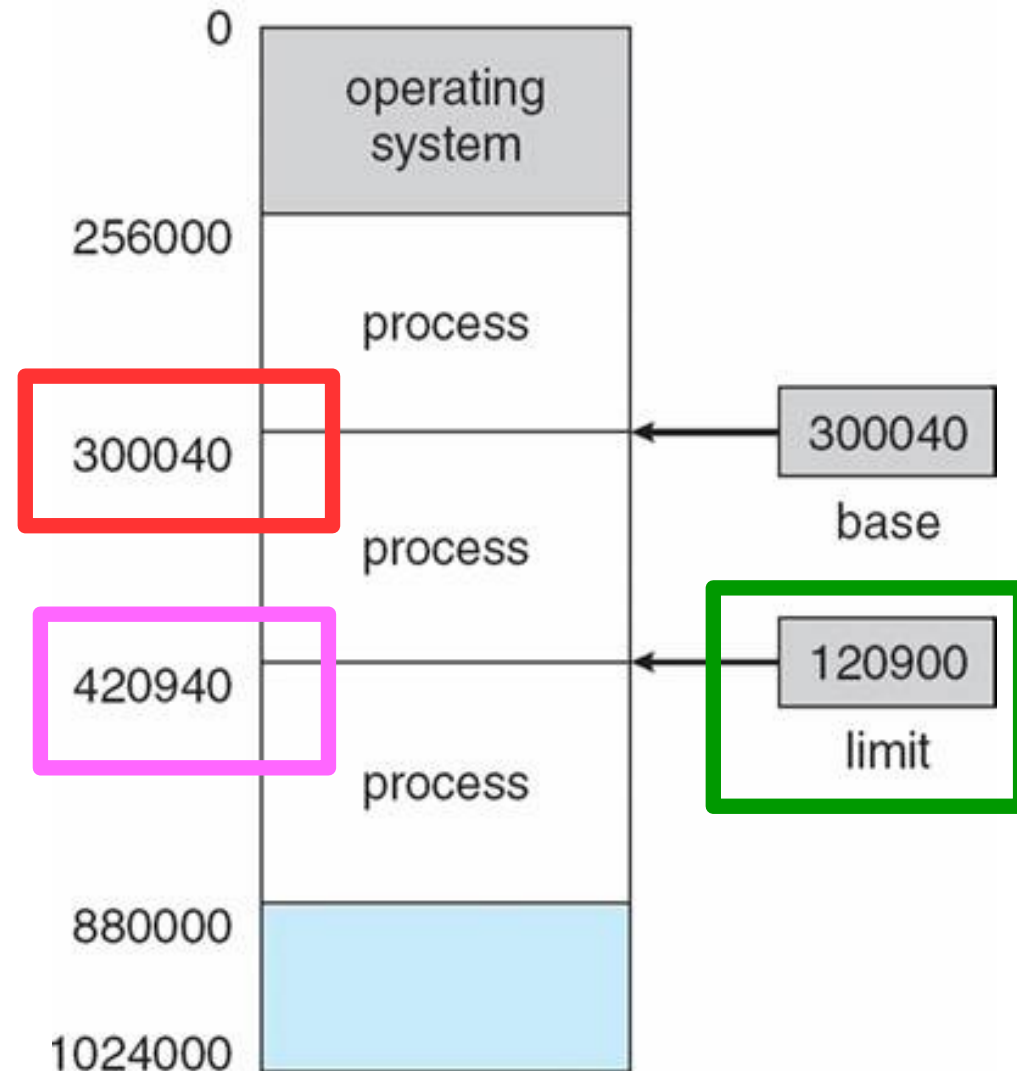
## Or Avoid Crashes By Imposing Limits

- To avoid programs invading each other's memory space, impose limits on which memory cells they can access.



# Private Spaces for Processes

- Programs are loaded into consecutive memory locations
- **Base register**
  - starting point
- **Limit register**
  - Amount of allowed memory from base point.
- Allocated memory:  
**Base** + **Limit** =  
**300040** + **120900** =  
**420940**



# Consider This!

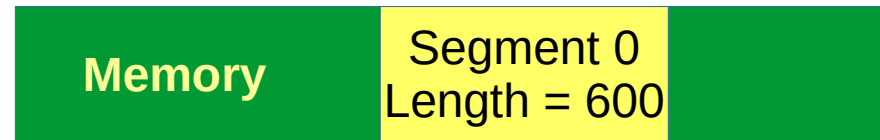
- Consider the memory segment table. What are the physical spans for the following logical addresses :
- a. (segment, limit)  $\rightarrow$  (0,430)
- b. (1,10)
- c. (1,11)
- d. (2,500)
- e. (7, 112)

<i>Segment</i>	<i>Base</i>	<i>Length</i>
<b>0</b>	<b>219</b>	<b>600</b>
<b>1</b>	<b>2300</b>	<b>14</b>
<b>2</b>	<b>90</b>	<b>100</b>
<b>3</b>	<b>1327</b>	<b>580</b>
<b>4</b>	<b>1952</b>	<b>96</b>

**THINK**

# Solutions

Problem 1  
explanation



Begin location  
Base = 219

End location:  
Base + length =  $219 + 600$   
= 812

- a.  $\text{base} + \text{limit} = 219 + 430 = 649$ .
  - Since the base =  $219 < 600 = \text{segment length}$ , good reference; segment fits here
- b.  $2300 + 10 = 2310$ .
  - Since the base =  $10 < 14 = \text{segment length}$ , good reference; segment fits here
- c.  $2300 + 11 = 2311$ 
  - Since the base =  $11 < 14 = \text{segment length}$ , good reference; segment fits here
- d. Illegal address since size of segment 2 is 100 and the offset in logical address is 500.
  - Since the base =  $500 \nless 100 = \text{segment length}$ , NOT good reference.
  - Segment cannot fit here.
- e. Illegal address; there is no segment '7'