THE UNIVERSITY *of* EDINBURGH
informatics

Operating Systems (INFR10079)

2023/2024 Semester 2

Memory Management
(Physical Memory)

abarbala@inf.ed.ac.uk

Chapter 9.1, 9.2, 9.5

Overview

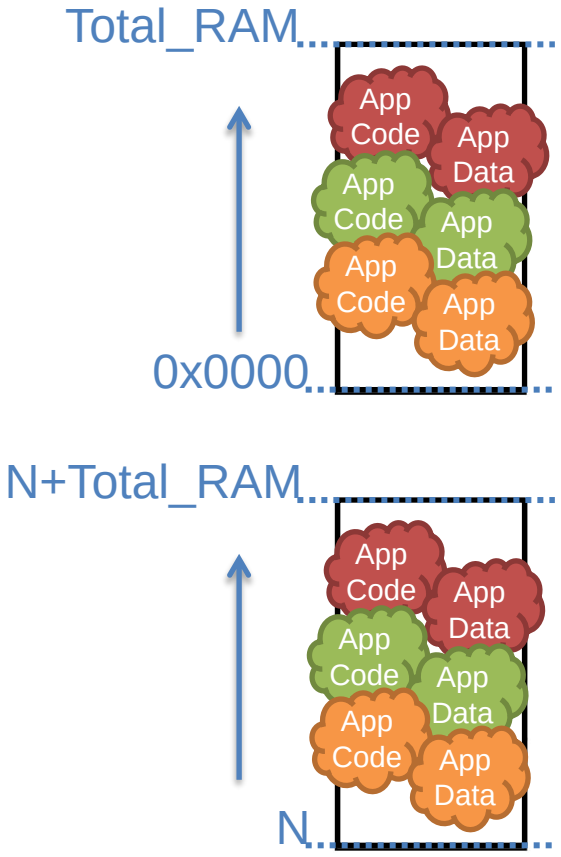
- *Background: basics*
- Running without any support (and abstraction)
- *Background: binding*
- Running with compiler support (no abstraction)
- Entering Hardware support
  - Base + Limit Registers
- *Address Space and Logical Memory Address*
- More Hardware support
  - MMU, Relocation + Limit Registers

- Contiguous Memory Allocation
- Swapping
- Growing Programs
- Fragmentation

Memory

- CPU (directly) connected (volatile) RAM

- A (large) byte addressable array
  - **Starts** with address N
    - N is 0x0000 in certain archs (x86)
  - **Ends** with address N + Total_RAM
    - Total_RAM is the amount of RAM
    - But there are exceptions
  - **Contains**
    - **Data, and**
    - **Code**

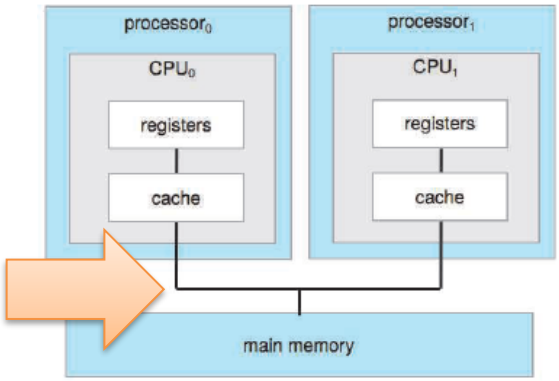


Goals of Memory Management

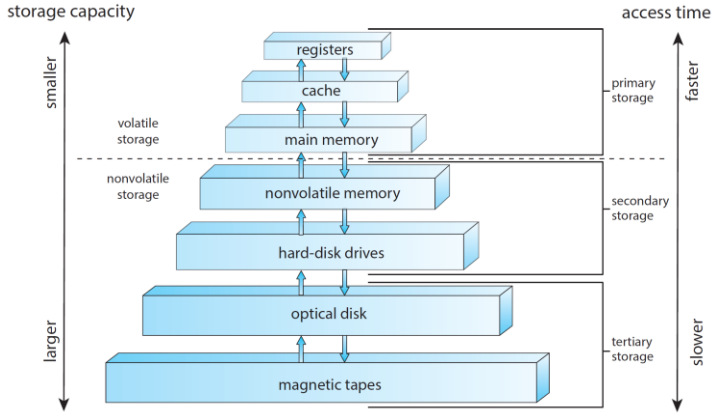
- **Allocate** memory resources among **processes and OS**
  - Maximizing memory utilization and system throughput
- Provide **convenient abstraction** for **processes** (applications) **and OS** programmers
  - Simplify memory utilization and addressability
- Provide **isolation** between **processes and OS**
  - Addressability and protection orthogonal problems

Background: Basics

- Program must be brought (from persistent storage) **into memory** and **placed within a process context** to be run
- *Main memory* and *CPU registers* are the only storage **CPU can access directly** (e.g., with a CPU instruction)
  - Memory unit only sees
    - **Physical** memory address + read request, or
    - **Physical** memory address + data and write request
  - Register access in one CPU clock
  - Main memory takes multiple CPU cycles, causing **CPU to stall**
  - **Cache** sits between main memory and CPU registers
    - Reduces CPU cycles to access memory
    - Transparent to the (assembly) programmer

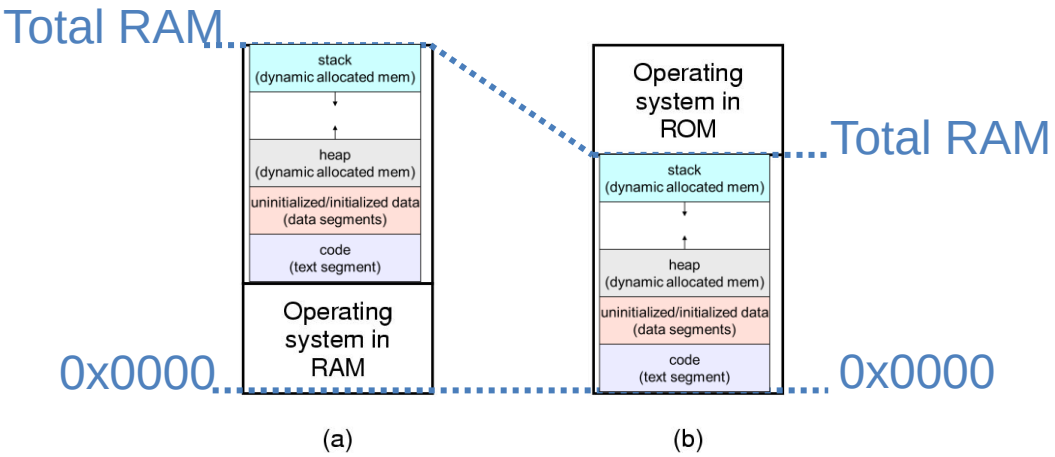


Physical Addr +
Request type +
Data (if W)



One Program with No Memory Abstraction

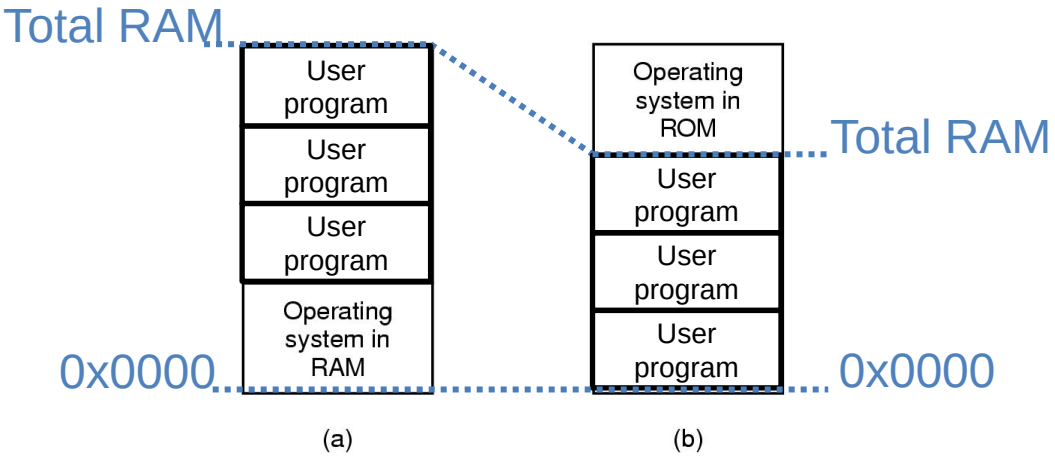
- Program sees (access) the physical memory
- Sharing physical memory with the OS (and even BIOS)
  - **Program can mess up with OS (and BIOS)**
  - Example, MSDOS



Two simple examples of organizing memory with an operating system and one user process

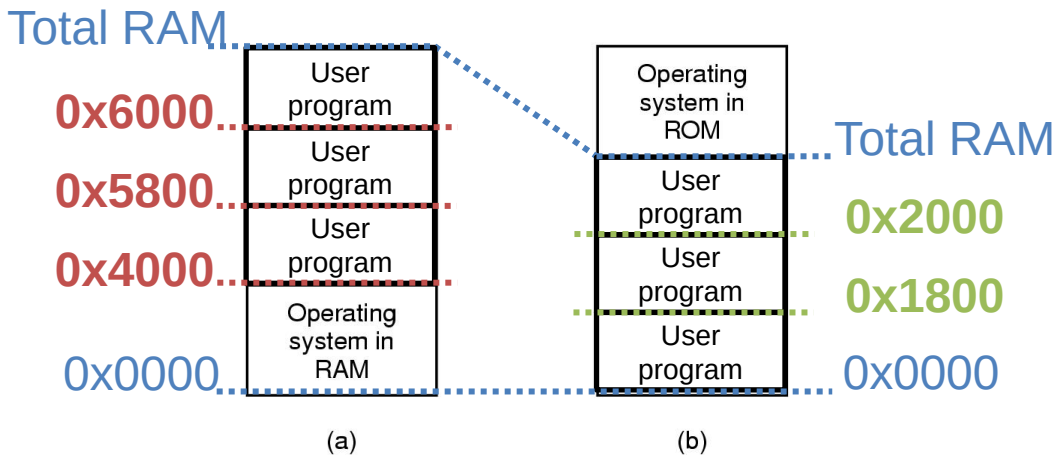
Multiple Programs with No Memory Abstraction

- **Every** program sees physical memory
  - Can access each other memory
  - Total # of programs depends on their size and memory size
- Sharing physical memory with the OS and even BIOS
  - **Programs can mess up with OS and BIOS**



Two simple examples of organizing memory with an operating system and **multiple** user processes

The code of each **user program,** or **operating system,** must have references (pointers) that fall into its program memory area only



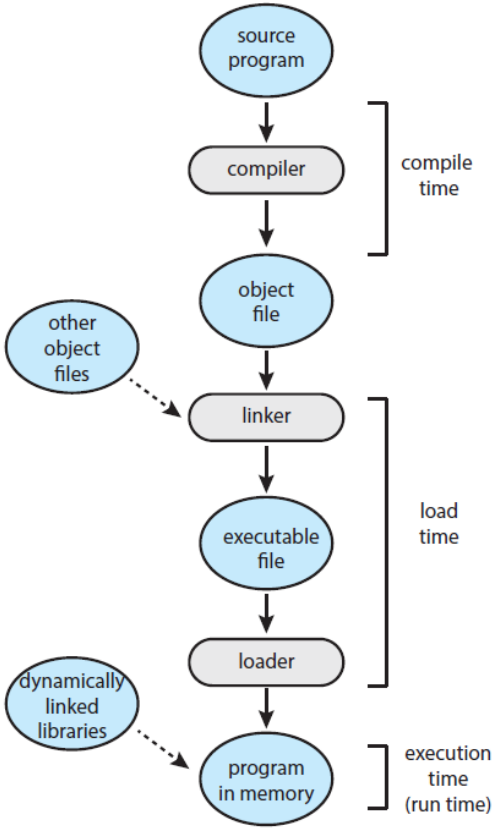
Two simple examples of organizing memory with an operating system and **multiple** user processes

Background: Binding

It is all about (memory) addresses

- Addresses in a program are usually symbolic (e.g., variable *count*)
  - **Binding** is the process of mapping

- Compiler (From source to object file)
  - **Binds** symbolic addresses to *relocatable addresses*
    - **E.g., base + offset**
- Linker (From object file to executable file)
  - **Binds** relocatable to ***absolute*** *addresses*
    - If the **final** memory location is known
    - Cannot be moved without hardware support
- Loader (Executable file image moved to memory)
  - **Binds** relocatable to ***absolute*** *addresses*
    - After the **final** memory location is decided
    - Cannot be moved without hardware support
- Execution



Multiple Programs: Understanding Relocation Problem

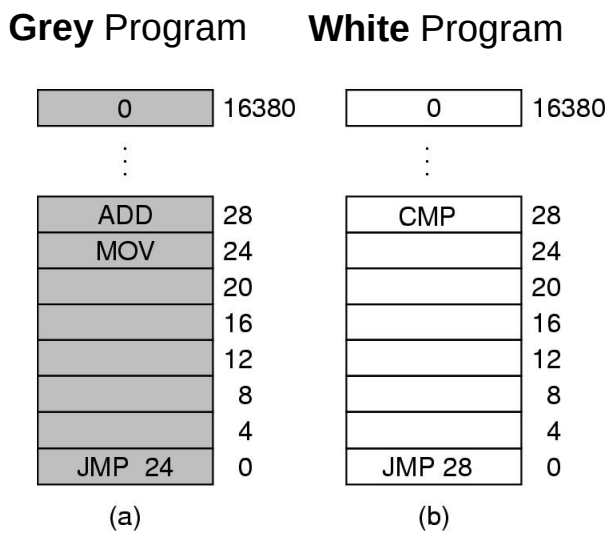


Illustration of the relocation problem. (a) A **non-relocatable (absolute addresses)** 16-KB program, the last address points to the first instruction. (b) Another **non-relocatable (absolute addresses)** 16-KB program, the last address points to the first instruction. (c) The two programs loaded consecutively into memory for execution.

Multiple Programs: Understanding Relocation Problem

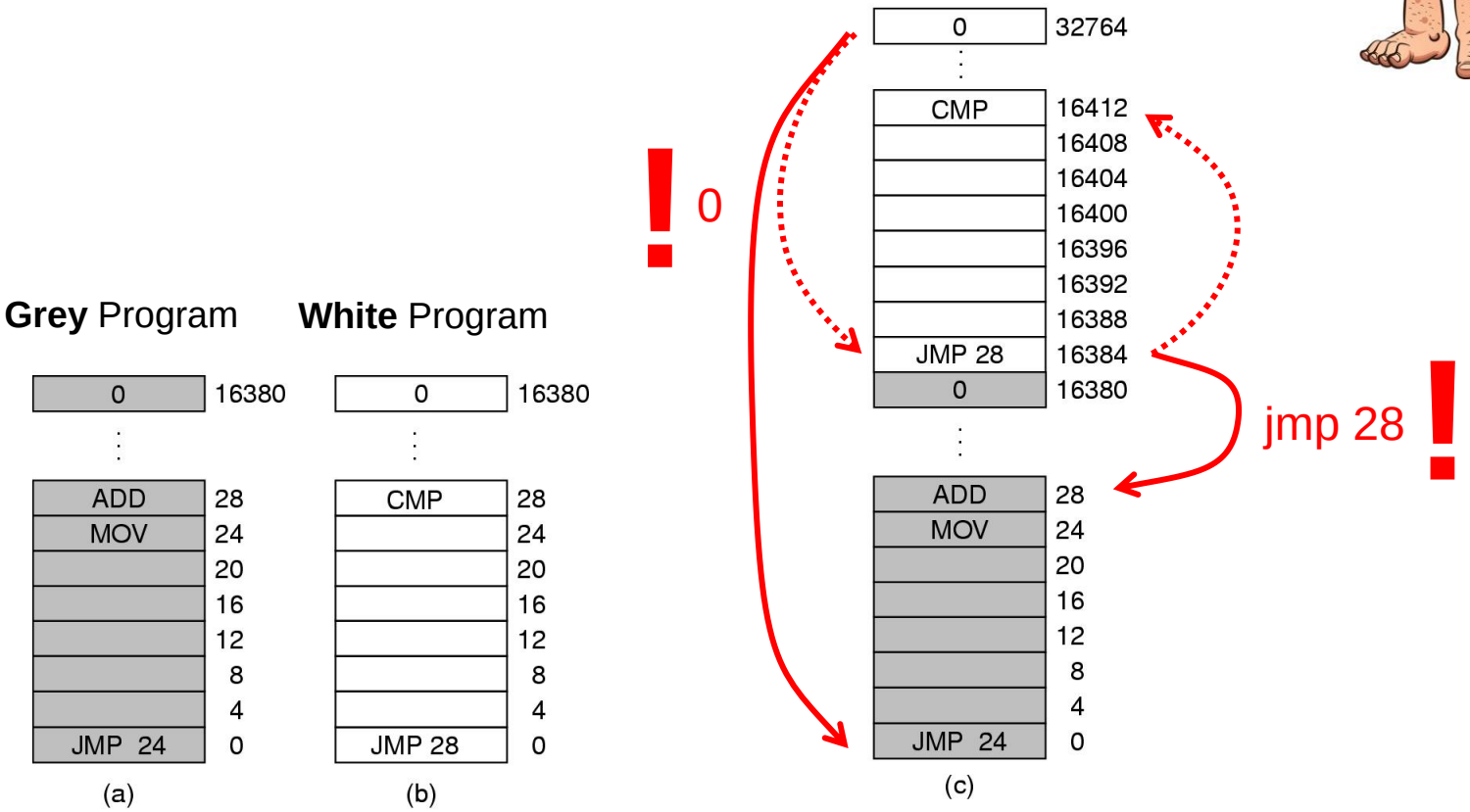
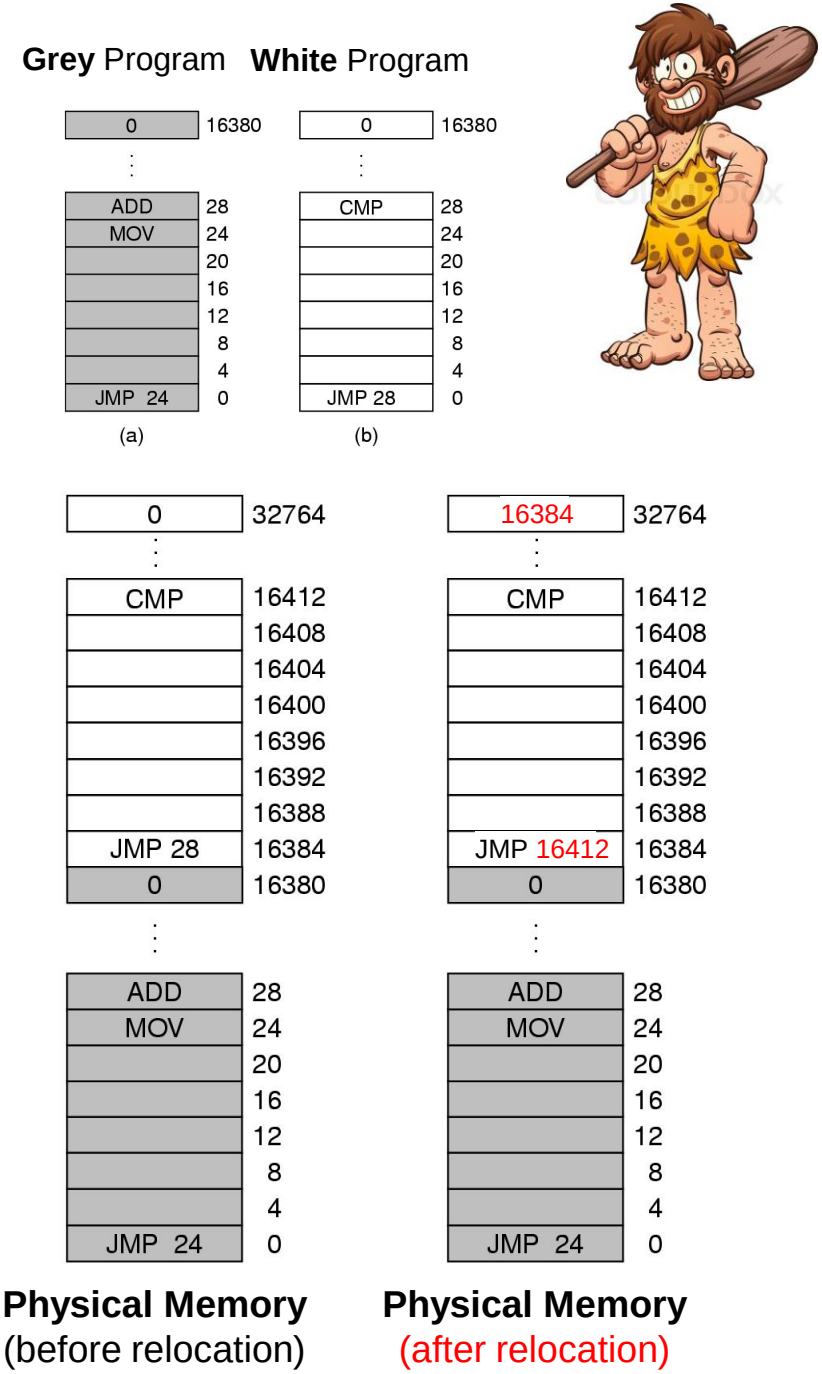


Illustration of the relocation problem. (a) A **non-relocatable (absolute addresses)** 16-KB program, the last address points to the first instruction. (b) Another **non-relocatable (absolute addresses)** 16-KB program, the last address points to the first instruction. (c) The two programs loaded consecutively into memory for execution.

Multiple Programs: Using Relocation

- Programs must be **relocatable**
  - Written to be **placed and run** at any memory address
  - Extra information in executable
- **Loader** decides where to place them
  - Based on the available physical memory
  - It does relocation (increases load time)



Multiple Programs on Physical Memory

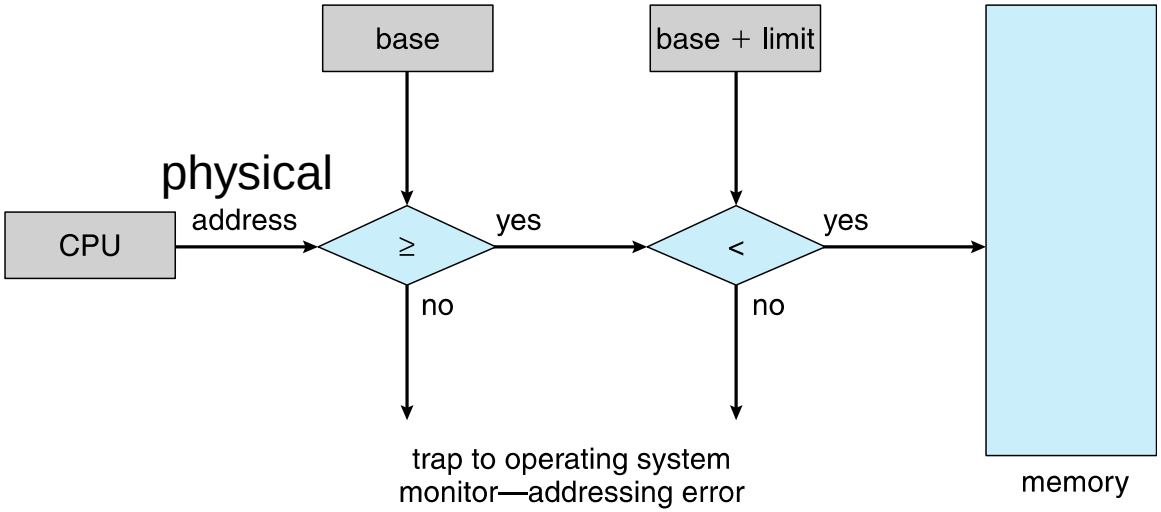
- All physical memory exposed to processes
  - User processes may **interfere** with OS and each other
  - Processes may **access** OS's and each other **secrets**
  - Programs must be relocated when loaded

- **Problems**
  - **No protection**
  - **Expensive relocation**

- *How to solve?*

Protection: Base and Limit Registers

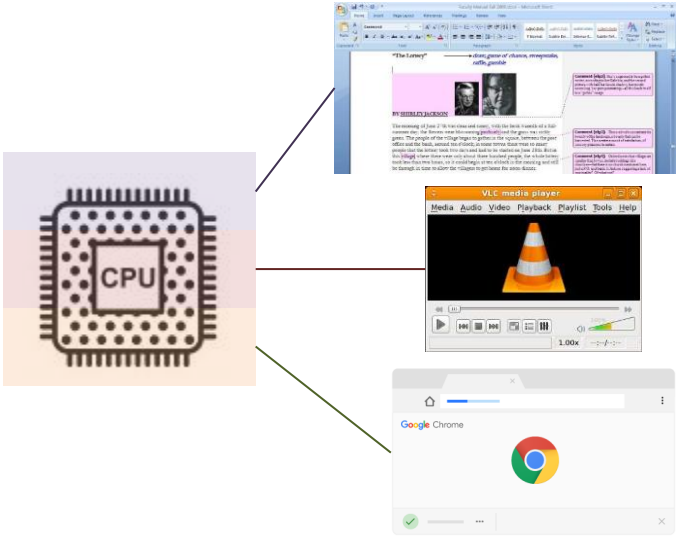
- **Base** and **limit registers** define the **valid** addresses for a process
- CPU checks every memory access generated by the process
  - If the address is **not valid (outside the range)** traps to OS



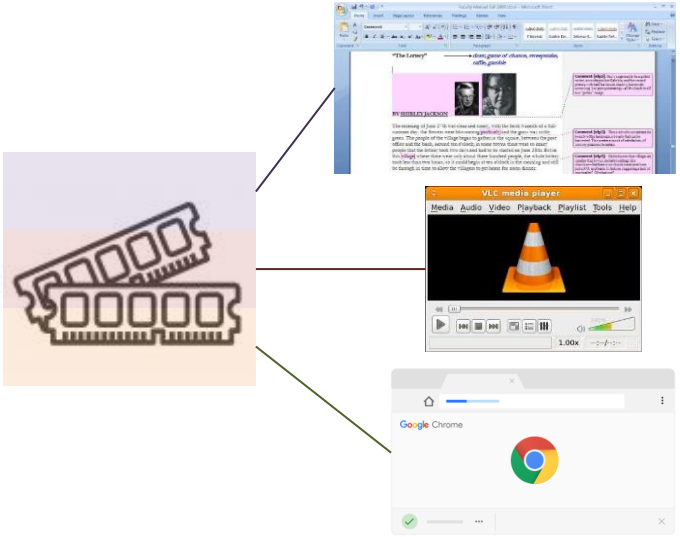
Memory Abstraction: Address Space

NOT just physical memory addresses!

- Address Space
  - Abstraction from physical memory space
  - Set of memory addresses that a process can use
    - Independently from other processes



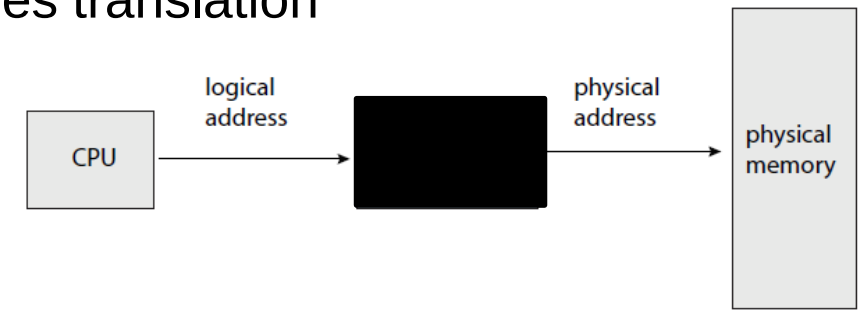
Process, abstracts physical CPU



Address space, abstracts physical memory

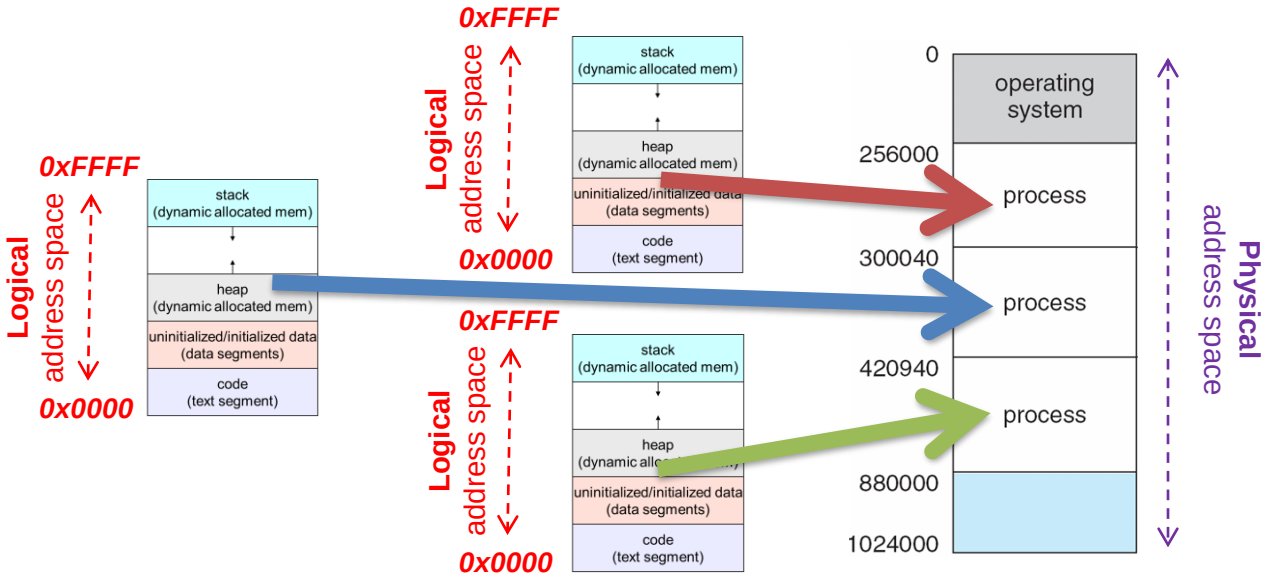
Logical Addresses #1

- To make it easier to manage memory of multiple processes
- Make processes use logical addresses
  - Logical addresses are independent of physical addresses
    - Data lives in physical addresses
    - OS manages physical memory
- Instructions issued by CPU are logical addresses
  - Example: pointers, arguments to load/store instructions, PC, etc.
- Logical addresses are translated by hardware into physical addresses
  - OS configures translation



Logical Addresses #2

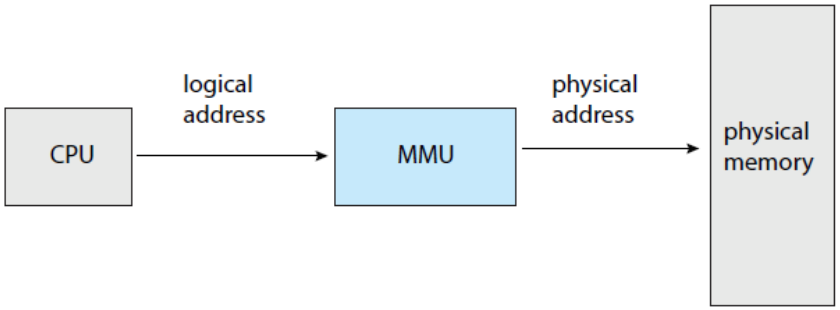
- Set of logical addresses a process can reference is its **address space**
- Program issues addresses in a logical address space
  - Must be **translated** to physical address space
  - Think of the program as having
    - A contiguous **logical address space** that starts at 0
    - A contiguous **physical address space** that starts somewhere



Memory-Management Unit (MMU)

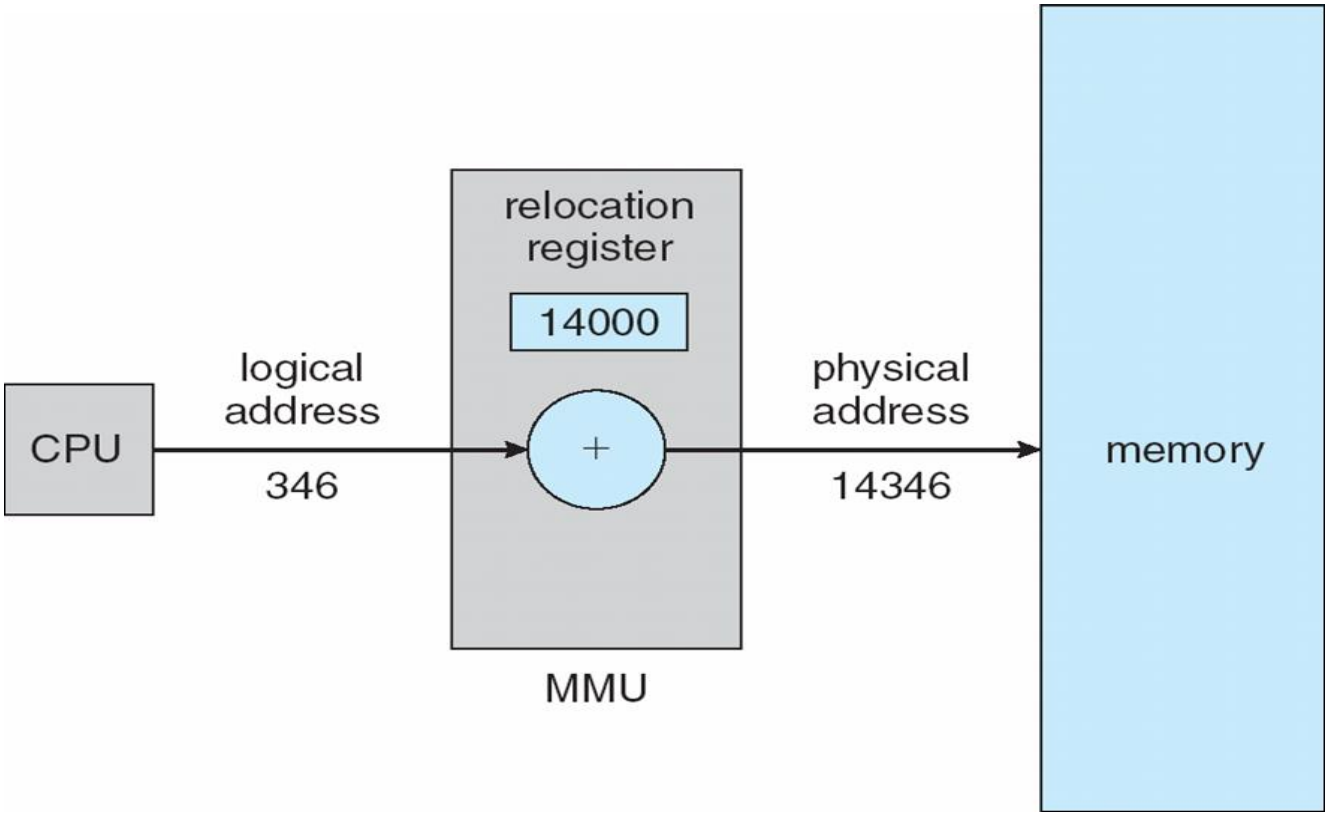
- Hardware component
  - **Translates** CPU generated addresses to physical addresses
- Programs deal with *logical addresses*; never see physical *addresses*
  - Logical address *bound to* physical addresses

- Many implementations
  - Relocation+limit registers, segmentation, paging, etc.
- Many names, based on features
  - MMU, MTU, MPU, etc.

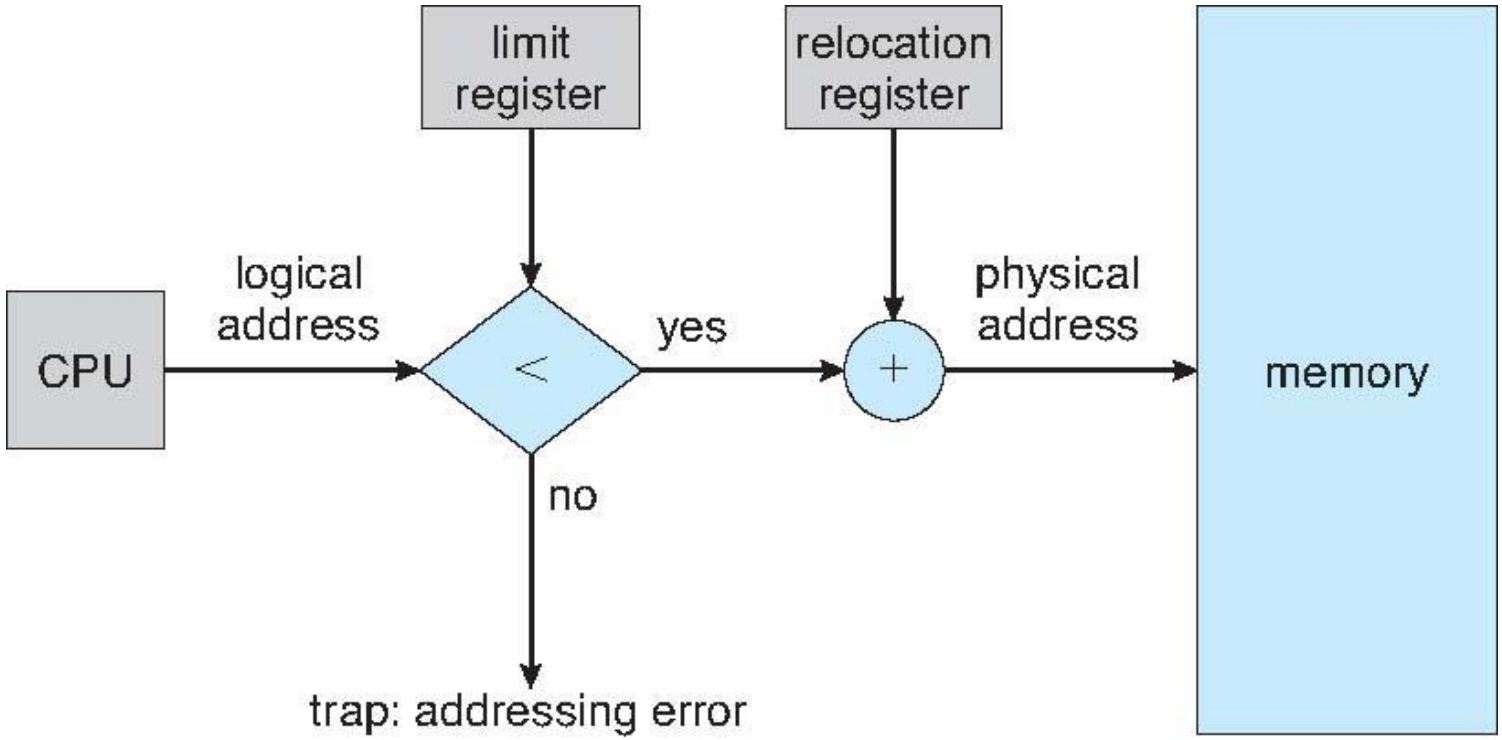


MMU as a Relocation Register

No protection
(all addresses are positive values)



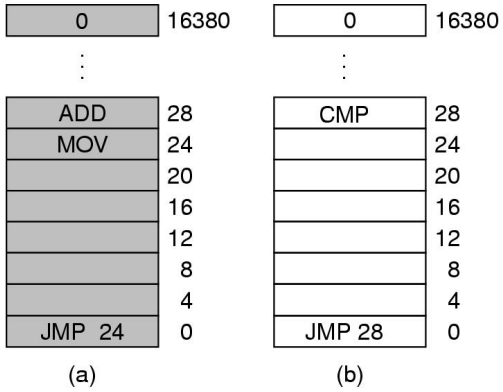
MMU as a Relocation and Limit Registers

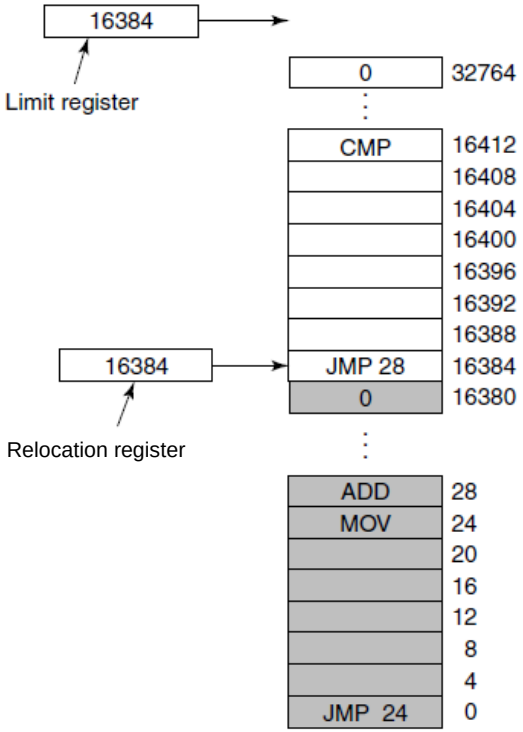


Multiple Programs: Relocation and Limit Registers #1

- **Hardware support** to ease relocation
  - Relocation register
  - Limit register
- **Program not relocatable**, simple loader
  - Faster load time
- Protection
  - Each process its own private address space

Base and limit registers can be used to give each process a separate address space.





Multiple Programs: Base and Limit Registers #2

- Variable size partitions (program size)
  - Relocation register and a limit register arithmetic
  - Arithmetic performed for each memory access