CISC 7310X

C06a Main Memory: Overview and Essential Scheme

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Acknowledgement

- These slides are a revision of the slides provided by the authors of the textbook

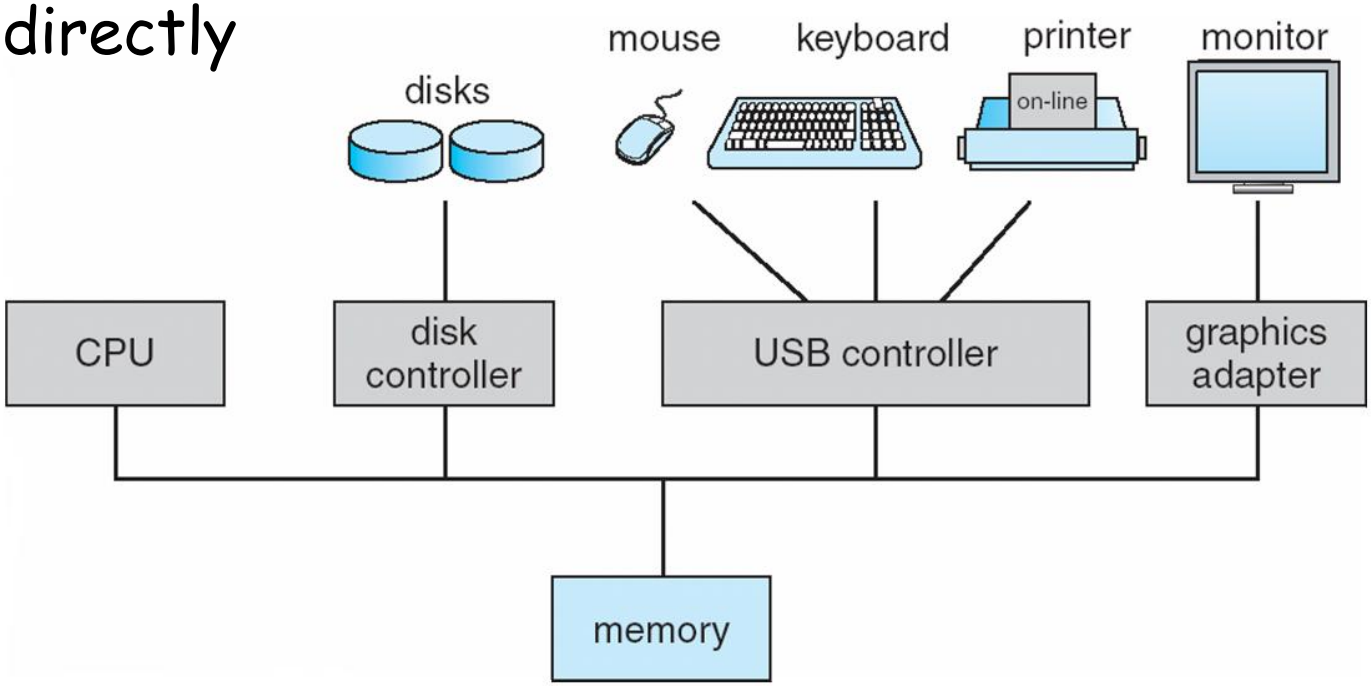
Outline

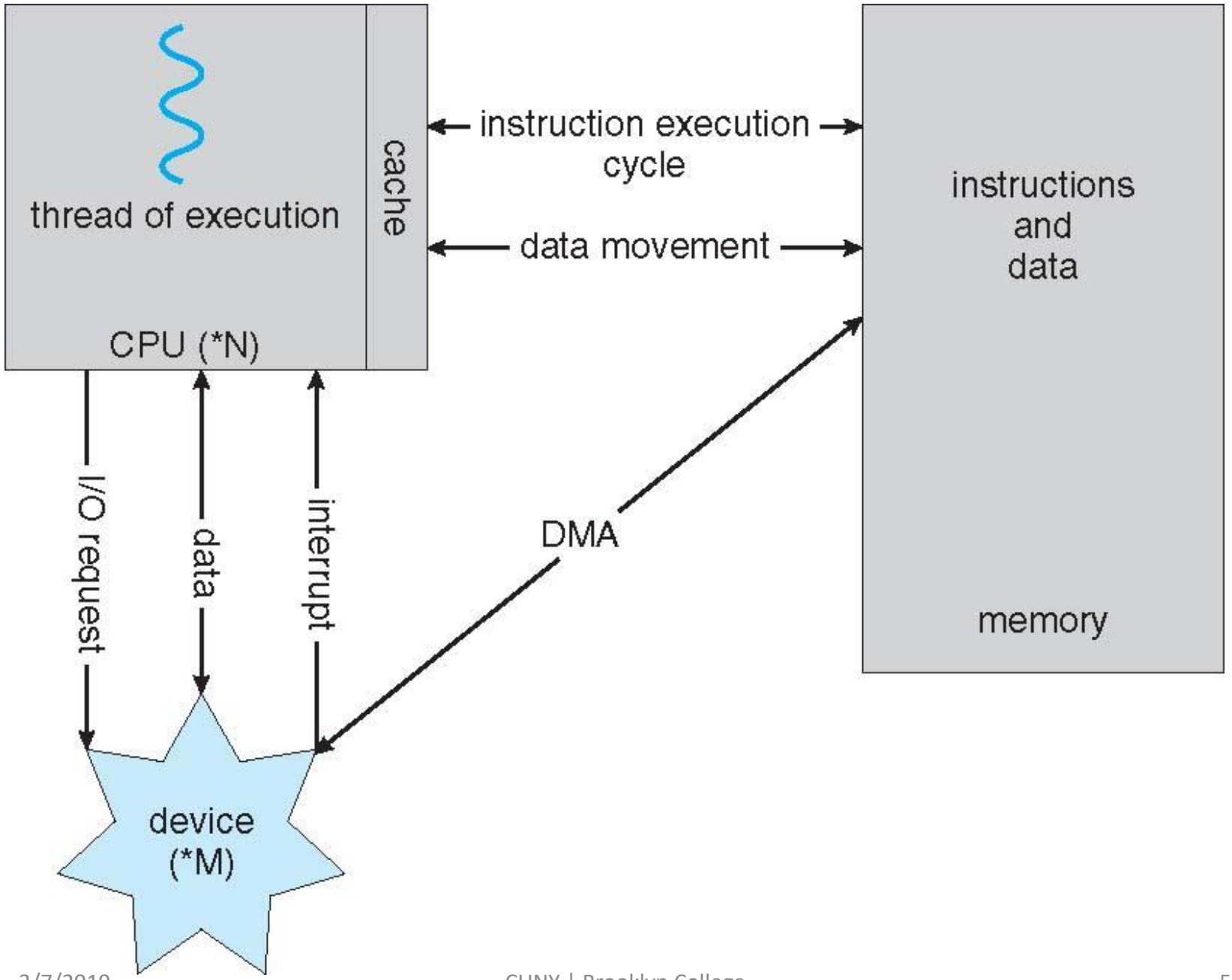
- Background
- Contiguous Memory Allocation

- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architectures
- Example: ARMv8 Architecture

Recap: An Organization of a Computer System

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly




thread of execution
cache
CPU (*N)
instruction execution cycle
data movement
instructions and data
memory
I/O request
data
interrupt
DMA
device (*M)

Recap: Addressing Memory

- Memory consists of a large array of bytes, each with its own address.
- Machine instructions that take memory addresses as arguments, but none that take disk addresses
- At the address, the memory contains
  - Instruction (whose address in program counter), or
  - data
- Memory unit only sees a stream of:
  - (when reading) address + read request, or
  - (when writing) address + data and write request

Recap: Memory Access Latency

- Registers are fast while memory slow
  - Register access is done in one CPU clock (or less)
  - Main memory can take many cycles, causing a stall
  - e.g., mov -0x8(%rbp),%rax
- Tacking memory stall:
  - Adding cache, fast memory sits between main memory and CPU registers
  - Implementing multithreaded processing core

Recap: the Key Takeaway

- Memory unit only sees a stream of:
  - (when reading) address + read request, or
  - (when writing) address + data and write request
- But multiple processes are running and accessing the memory. How do we ensure correct operation?

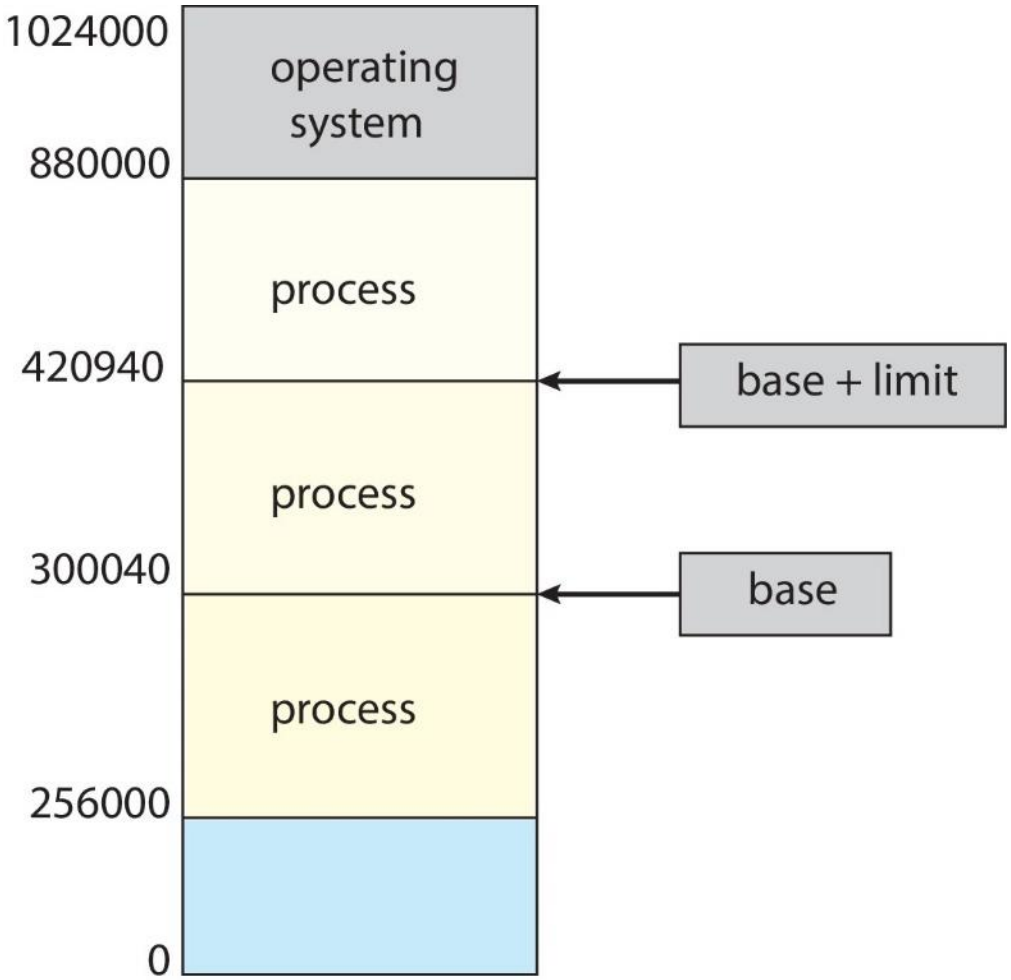
Protection

- Need to ensure that a process can *access and only access* those addresses in its address space.
  - Protect the operating system (including code and data) loaded in the memory from access by user processes
  - Protect one user process (including code and data) from access by another
- This protection is generally provided by hardware

Protection: Essential Approach

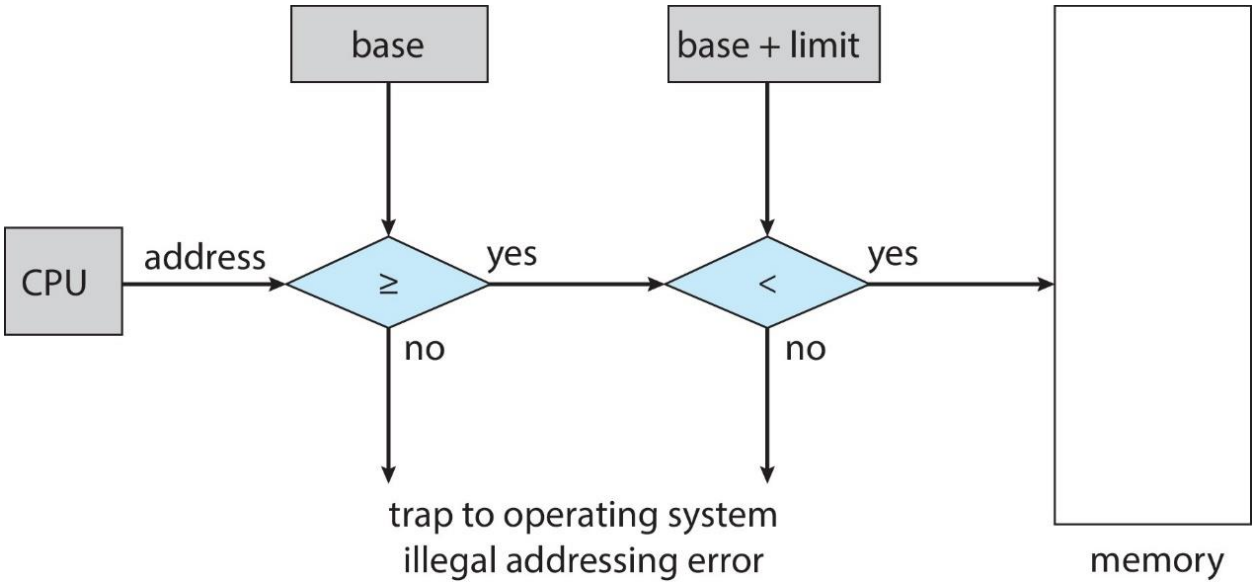
- Each process is given separate memory space.
- Using a pair of base and limit registers to define the logical address space of a process.

Base and Limit Registers



Hardware Address Protection

- CPU must check every memory access generated in *user mode* to be sure it is between base and limit for that process



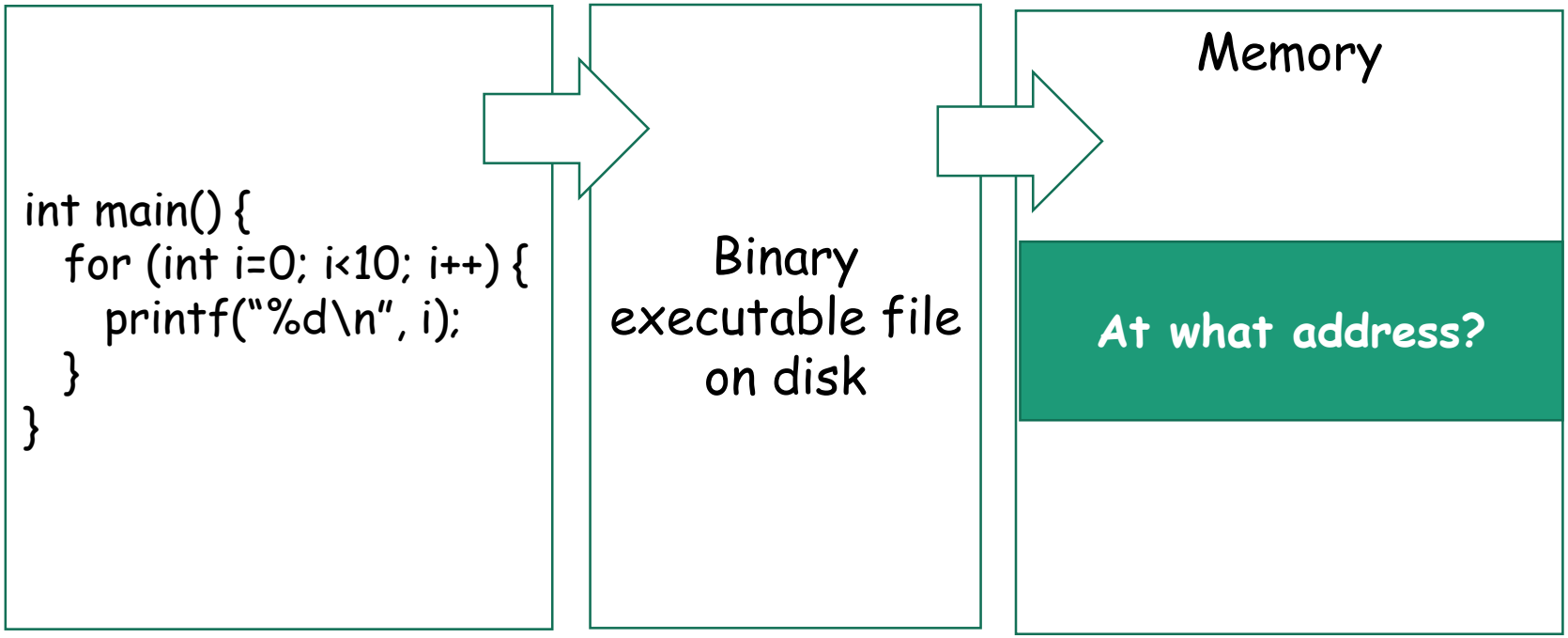
Loading Base and Limit Registers

- The instructions to loading the base and limit registers are privileged

Questions?

- Logical address space
- Memory protection
- Essential memory protection approach

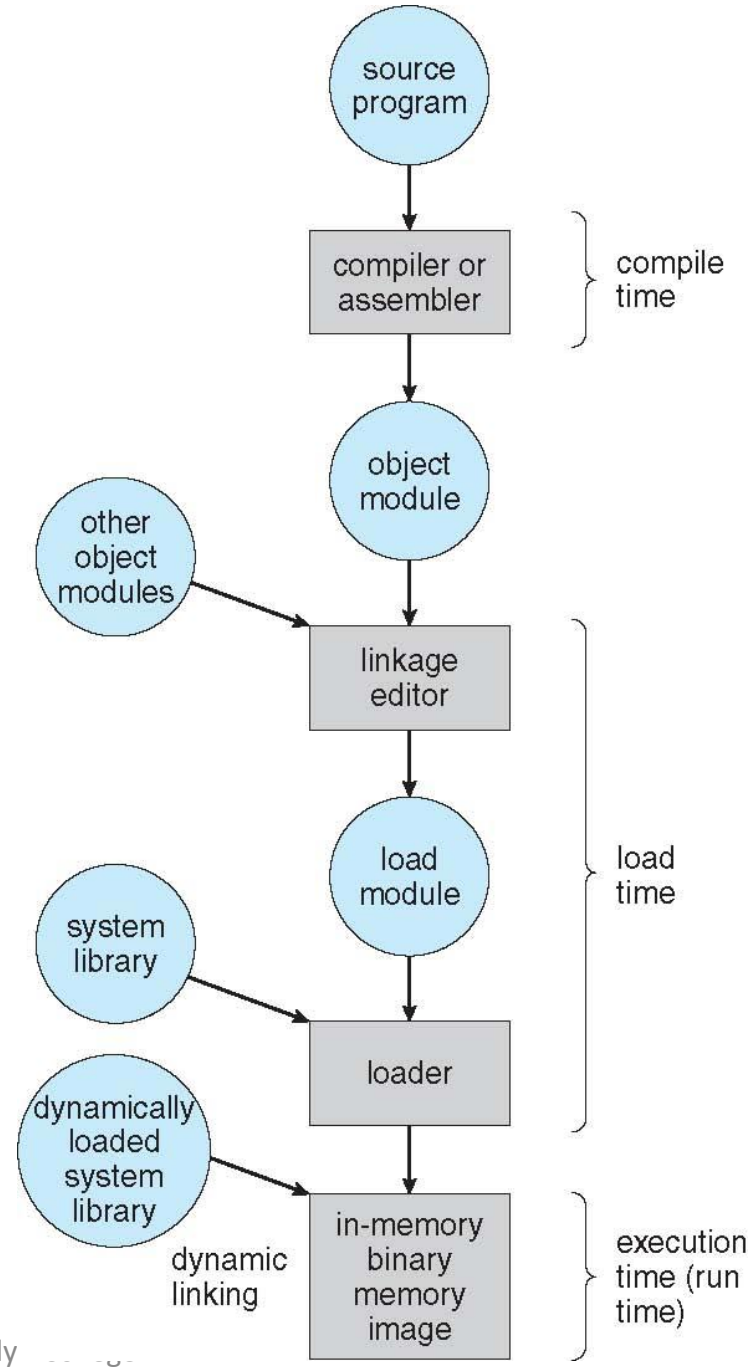
The Address Binding Problem



Fixed Address?

- Loaded into address 0000 (or other fixed address)
  - Inconvenient to have first user process physical address always at 0000
- How can it not be?
  - A number of address binding approaches will be discussed.
  - But, it can happen at many stages, meaning…

- A user program goes through multiple step of processing and transformation.
- The binding of instructions and data to memory addresses can be done at any step along the way



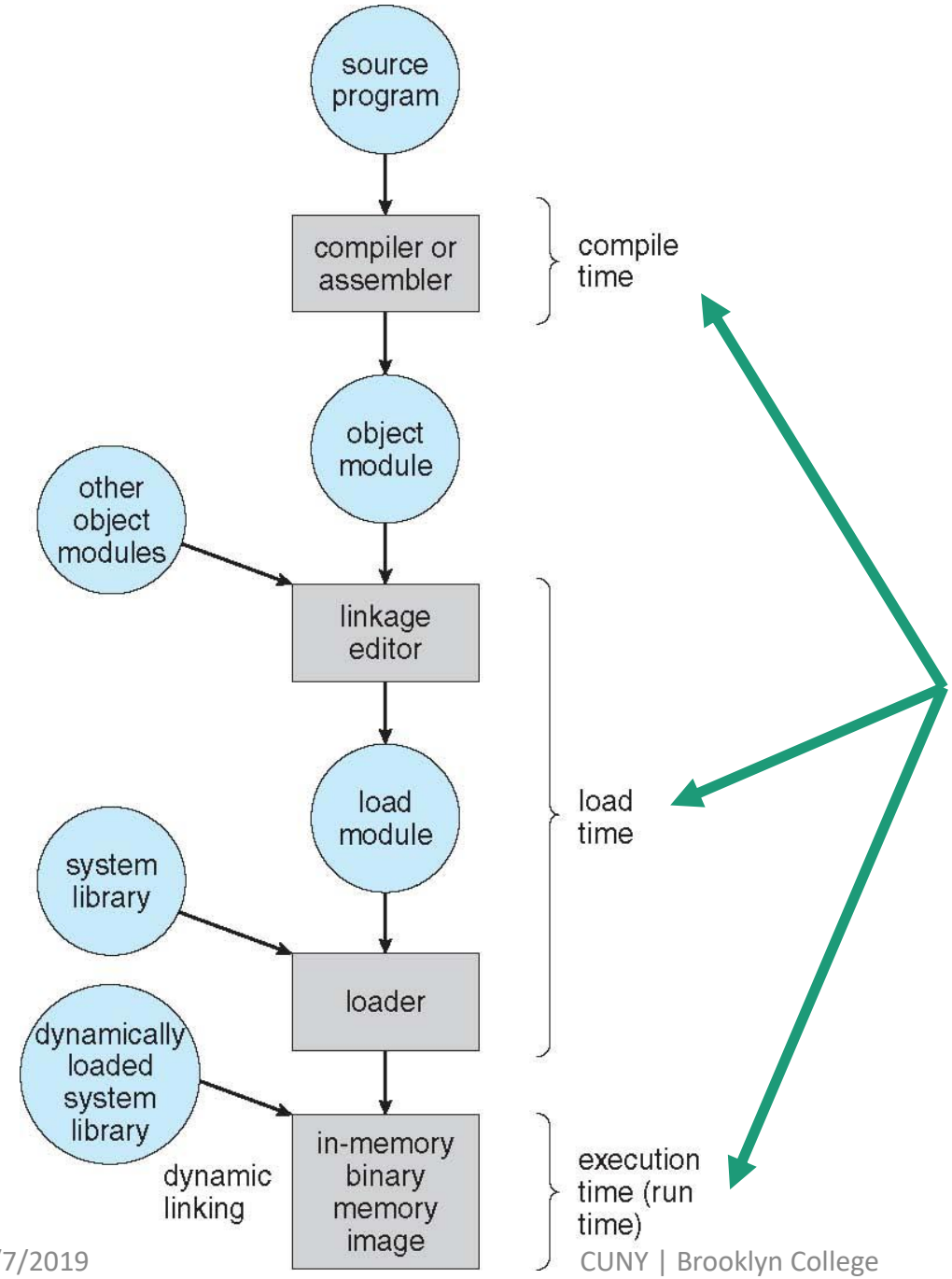
Address Representation and Binding

- Addresses are represented in different ways at different steps

1. Source code addresses usually symbolic
   - i.e., gpa = grade_points/credits; print_gpa(sid, gpa);
2. Compiled code addresses bind to relocatable addresses
   - i.e. "14 bytes from beginning of this module"
3. Linker or loader will bind relocatable addresses to absolute addresses
   - i.e. 74014
4. Each binding maps one address space to another

Address Binding

- Address binding of instructions and data to memory addresses can happen at,
  - Compile time.
    - If memory location known a priori, absolute code can be generated; must recompile code if starting location changes
  - Load time.
    - Must generate relocatable code if memory location is not known at compile time
  - Execution time.
    - Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)



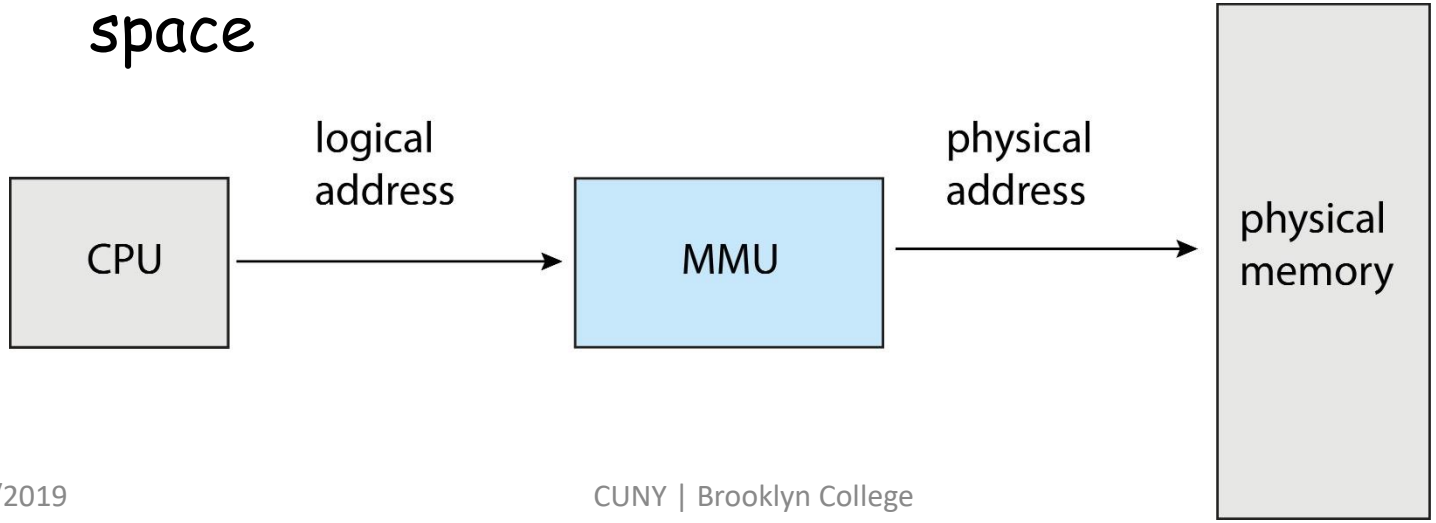
Address binding

Questions?

- Binding instruction and data to memory addresses
  - What? (Meaning)?
  - When?

: CPU and MMU

- Hardware component
  - CPU: generates logical addresses forming logical/virtual address space
  - MMU: memory management unit generates physical address becoming physical address space



Logical vs. Physical Address Space

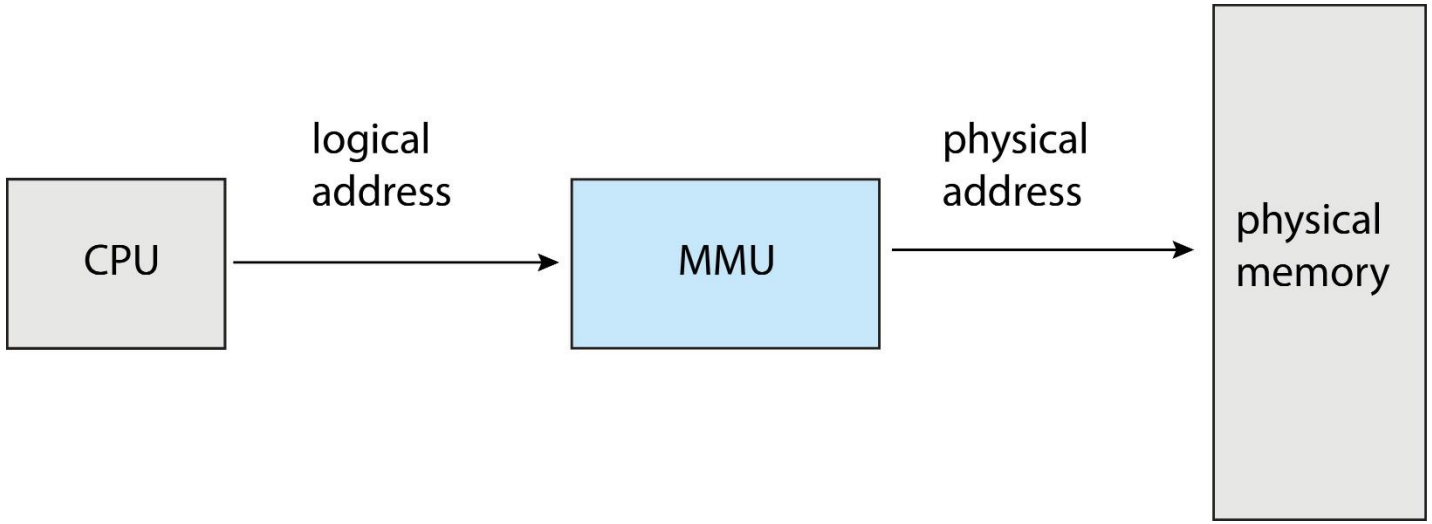
- Logical address: generated by the CPU; also referred to as virtual address
- Physical address: address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes
- logical (virtual) and physical addresses differ in execution-time address-binding scheme
- Logical address space is the set of all logical addresses generated by a program
- Physical address space is the set of all physical addresses generated by a program

Execution-Time Address Binding

- A logical address space is bound to a *separate* physical address space at execution time
- How this execution-time address binding takes spaces is central to memory management

MMU

- Hardware device that at run time maps virtual to physical address



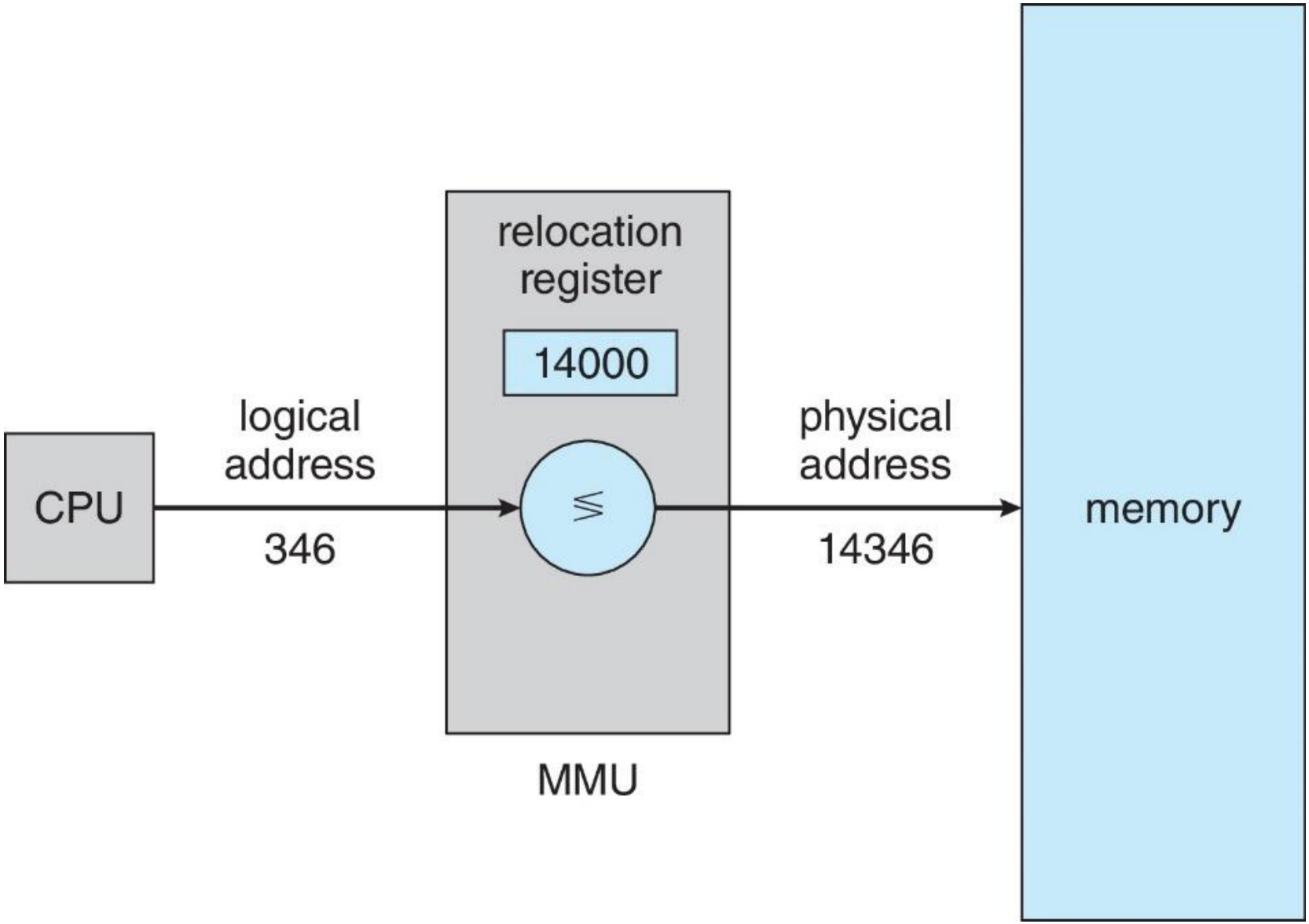
- Many methods possible, covered in the rest of this chapter

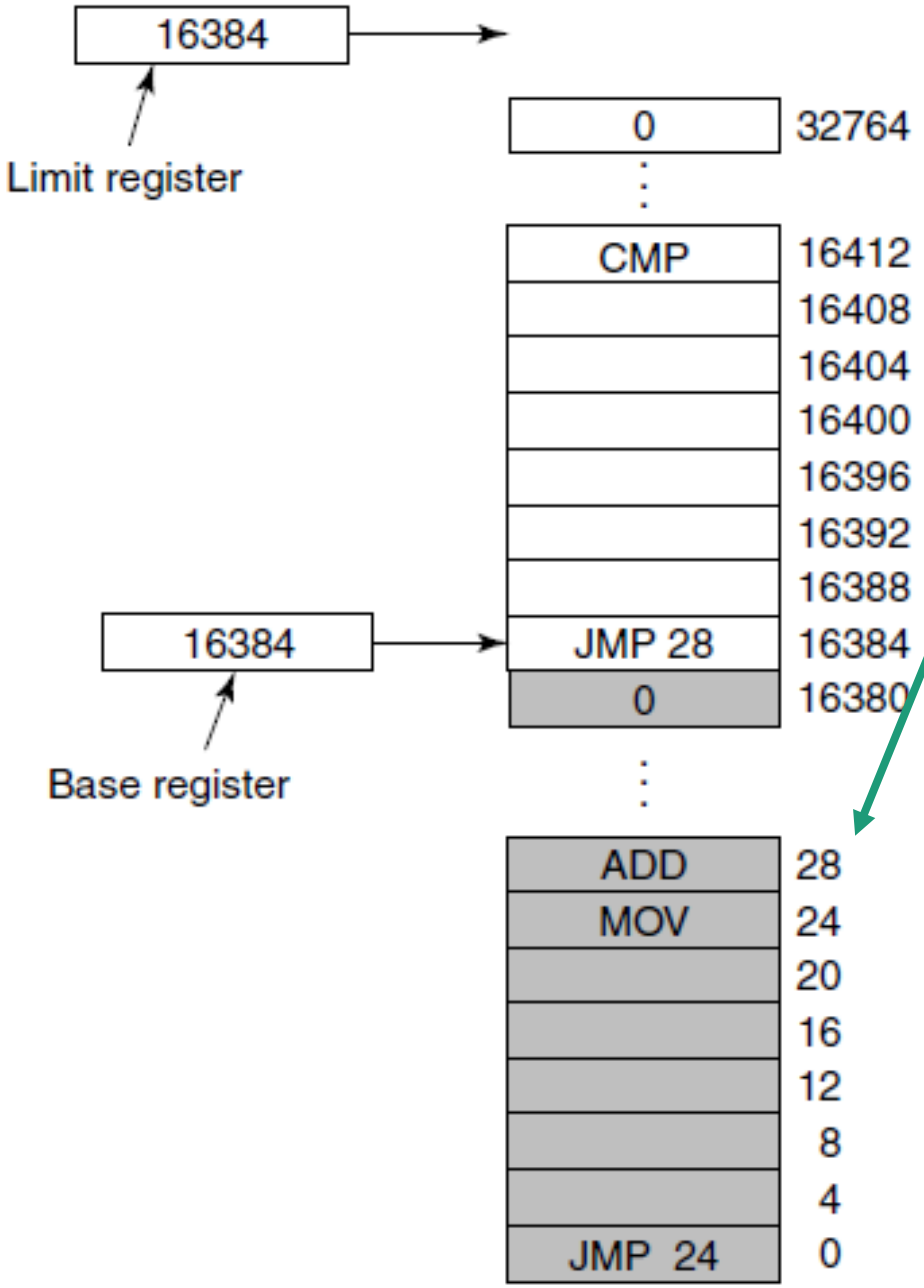
Questions?

- Concept of logical and physical addresses and address spaces
- CPU and MMU

MMU: Base-Limit Register Scheme

- Consider a base register scheme (base-limit register scheme)
  - The base register now called relocation register
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with logical addresses; it never sees the real physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses


relocation
register
14000
logical
address
346
CPU
≤
physical
address
14346
memory
MMU



- Two processes, each has its own logical address spaces starting at 0, which are mapped to separate physical address spaces starting at the addresses in their respective relocation (or base registers)

- Base register ≡ Relocation register
- Dynamic relocation via base and limit registers [Figure 3-3 in Tanenbaum & Bos, 2014]

Questions?

- Logical and physical addresses
- Logical and physical address spaces
- Hardware support

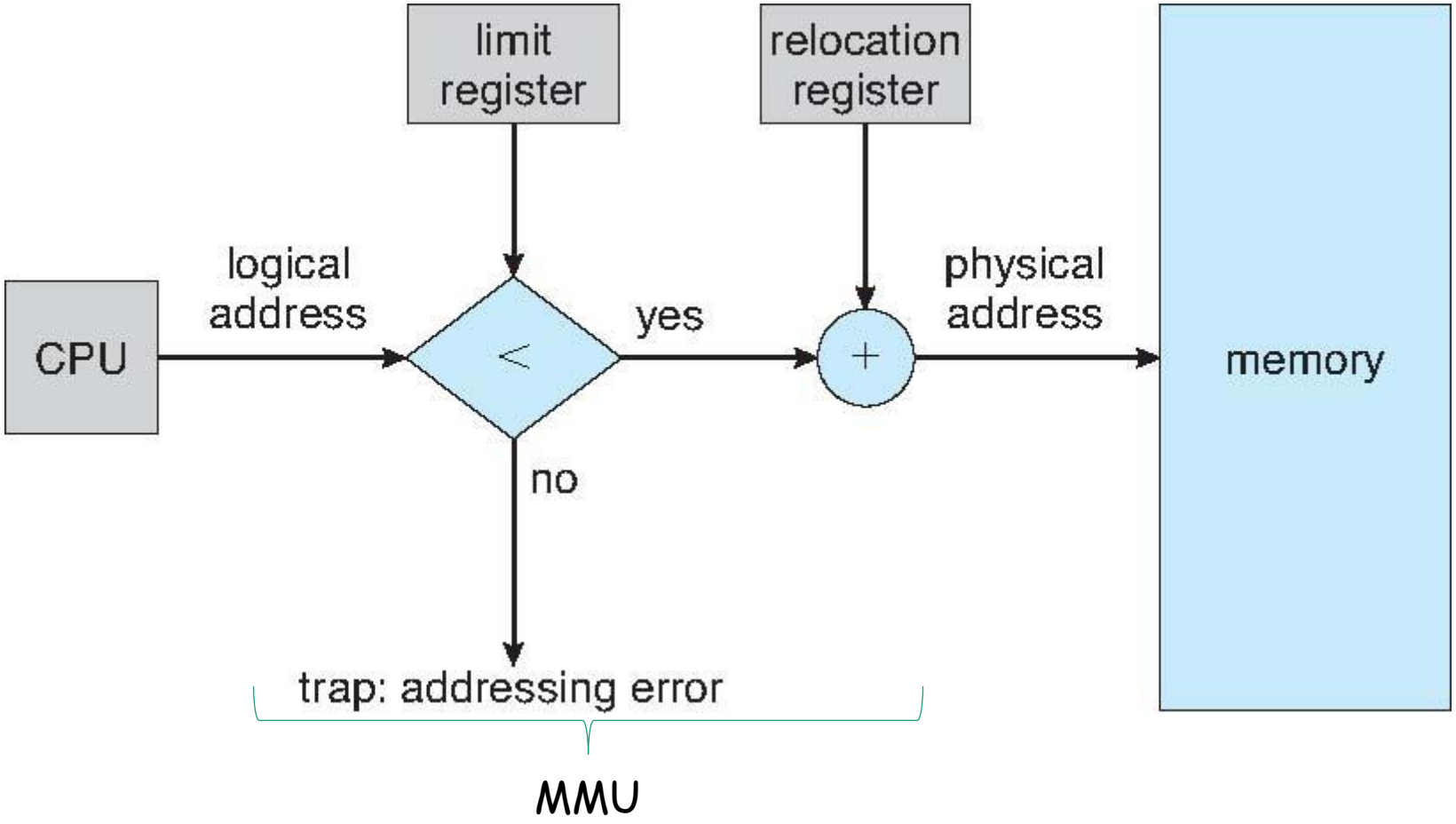
Memory Allocation

- The main memory must accommodate both the operating system and the various user processes.
- We ought to allocate main memory in the most efficient way possible.
- Example:
  - contiguous memory allocation, an early method

Continuous Memory Allocation

- Each process is contained in a single section of memory that is contiguous to the section containing the next process.
- Memory allocation?
- Memory protection?

Relocation and Limit Registers



Memory Allocation

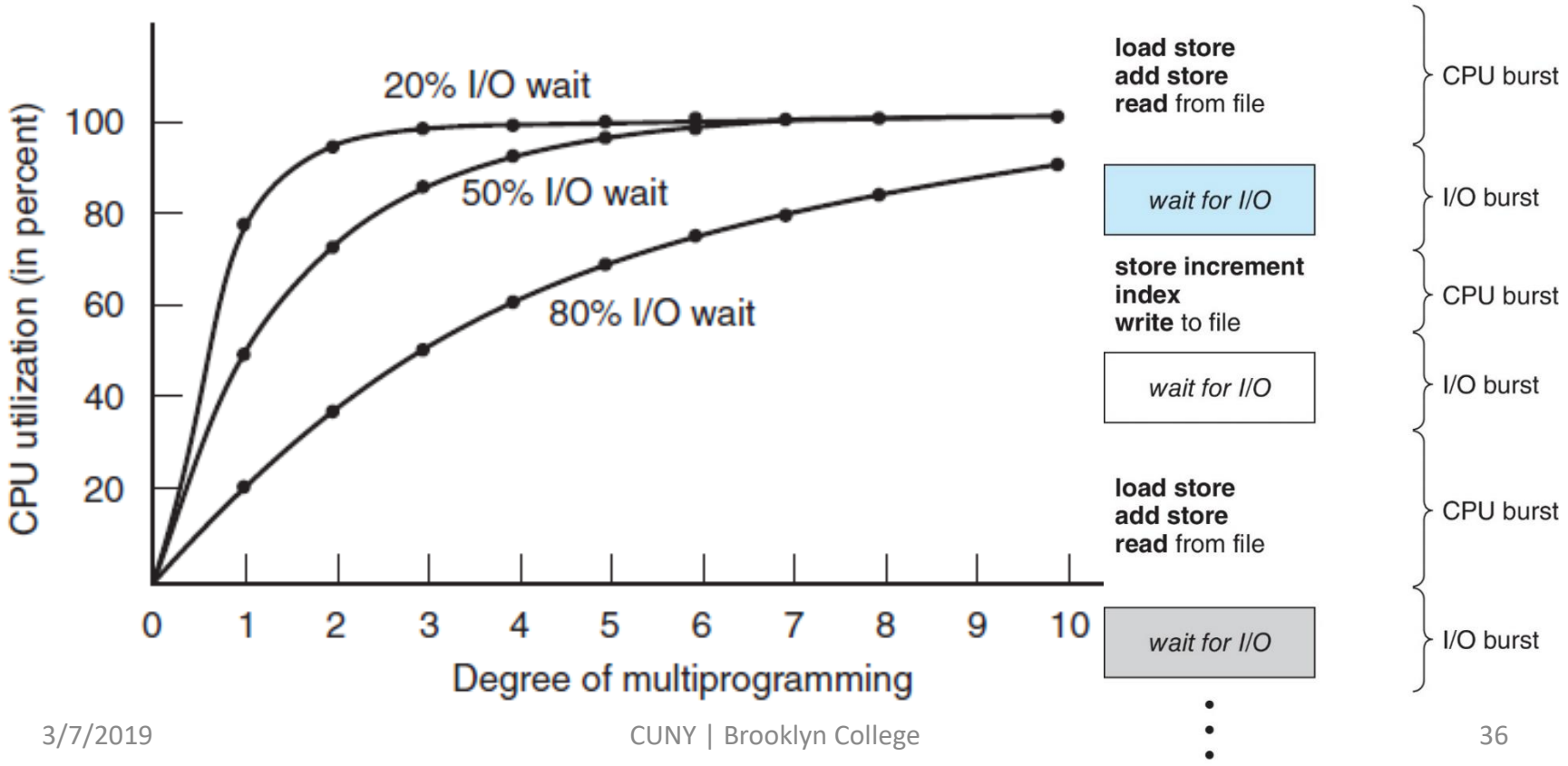
- Assign processes to *variably* sized partitions in memory, where each partition may contain exactly one process
- A variable partition scheme

Variable Partition

- Variable-partition sizes for efficiency (sized to a given process' needs)
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about: a) allocated partitions b) free partitions (hole)

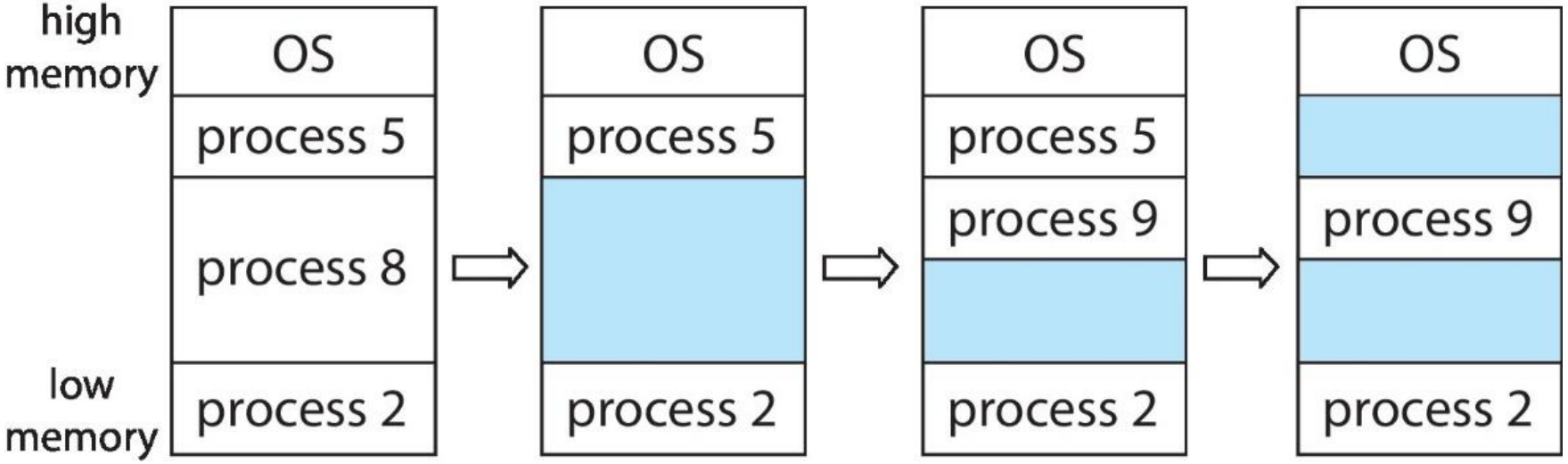
Degree of Multiprogramming

- Degree of multiprogramming limited by number of partitions



Memory Holes

- Hole – block of available memory; holes of various size are scattered throughout memory



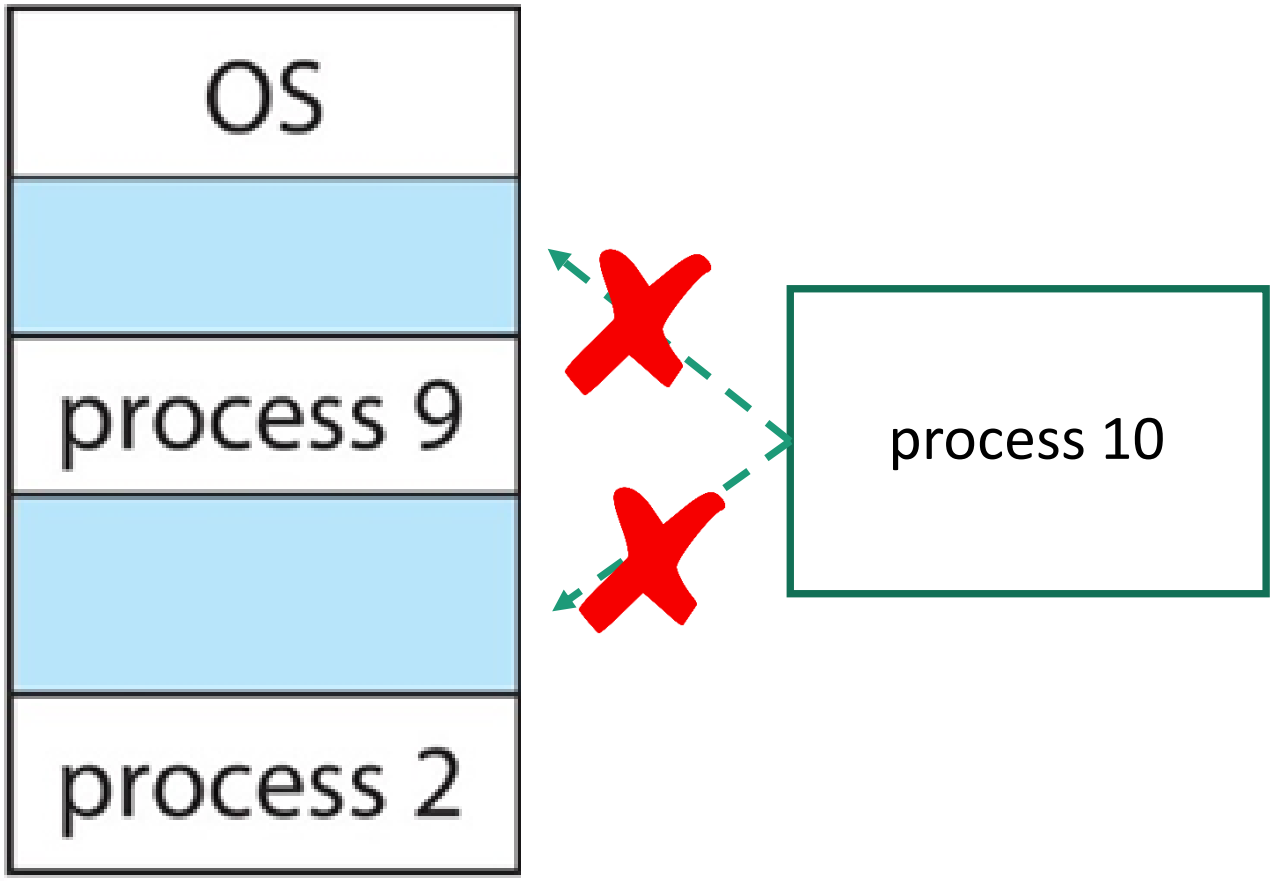
Dynamic Storage-Allocation Problem

- How to satisfy a request of size n from a list of free holes?
  - First-fit: Allocate the first hole that is big enough
  - Best-fit: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
  - Worst-fit: Allocate the largest hole; must also search entire list
  - Produces the largest leftover hole
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

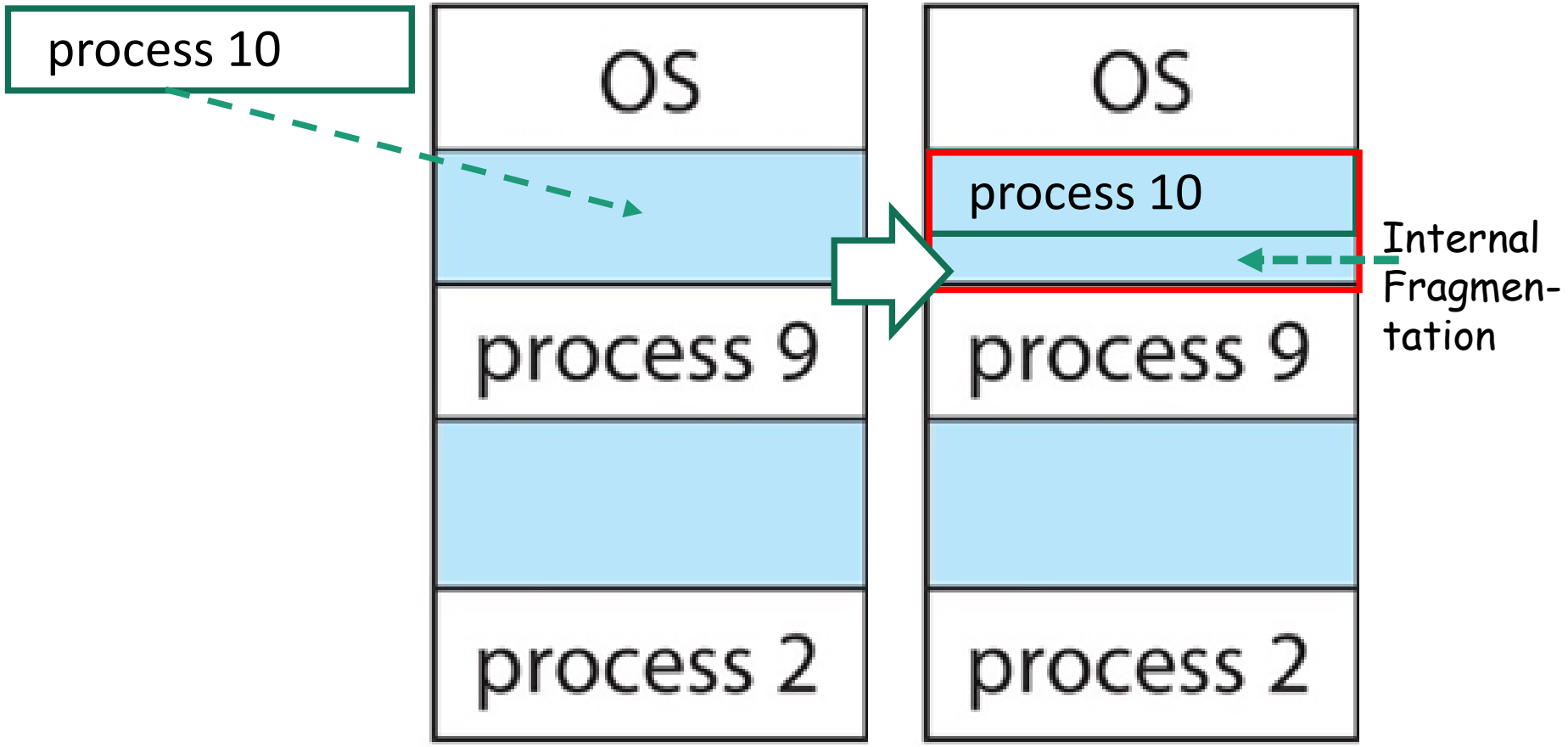
Fragmentation

- External Fragmentation
  - Total memory space exists to satisfy a request, but it is not contiguous
- Internal Fragmentation
  - Allocated memory may be slightly larger than requested memory
  - This size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given N blocks allocated, another 0.5 N blocks lost to fragmentation
  - 1/3 may be unusable -> 50-percent rule

External Fragmentation



Internal Fragmentation



Combating Fragmentation

- Reduce external fragmentation by compaction
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

Questions

- Continuous Memory Allocation
  - Memory protection mechanism and hardware
  - Memory allocation
    - Variable partition allocation
      - Degree of multiprogramming
      - Memory hole
      - Memory fragmentation
      - Compaction
- Any other methods to solve fragmentation problem?