CISC 3320

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 via the publisher of the textbook

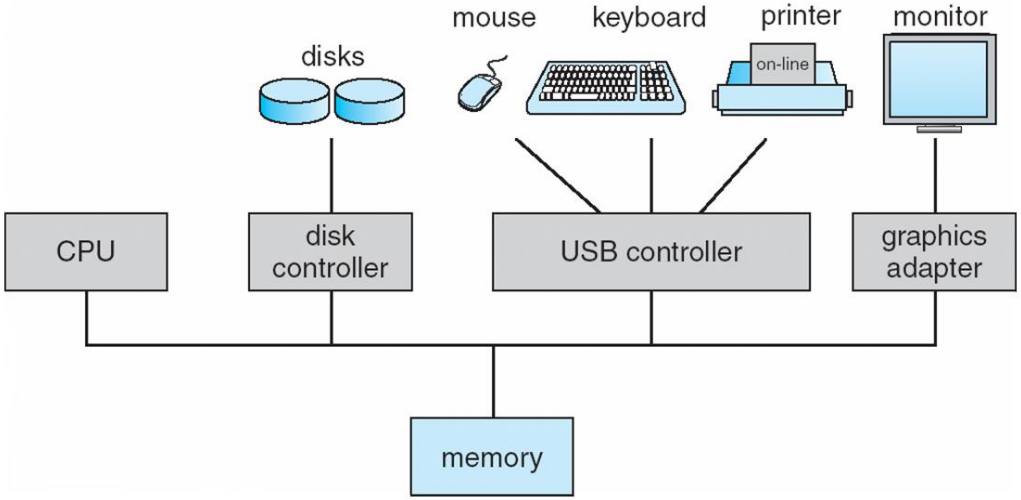
Outline

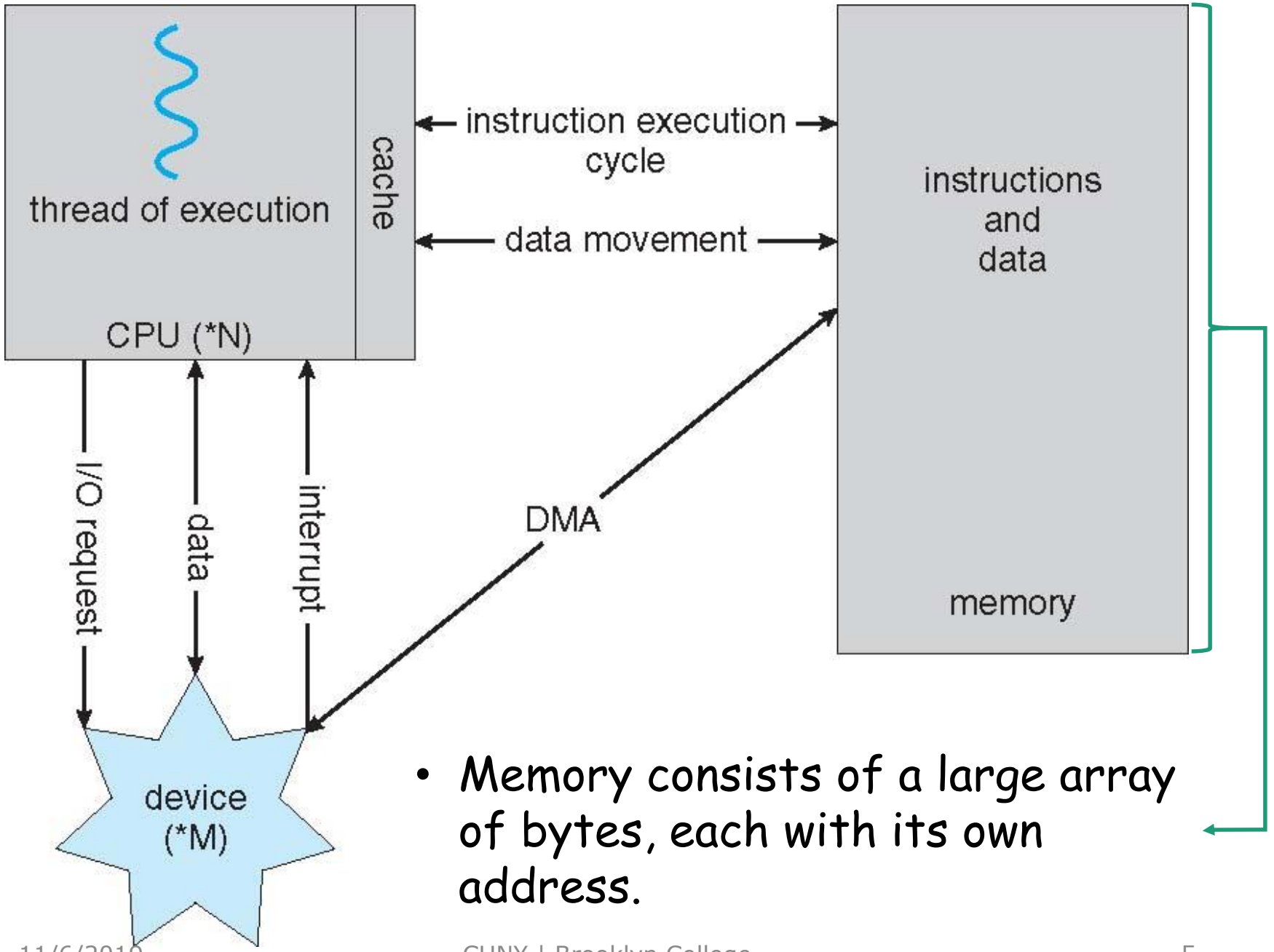
- Recap and background
- Contiguous Memory Allocation

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

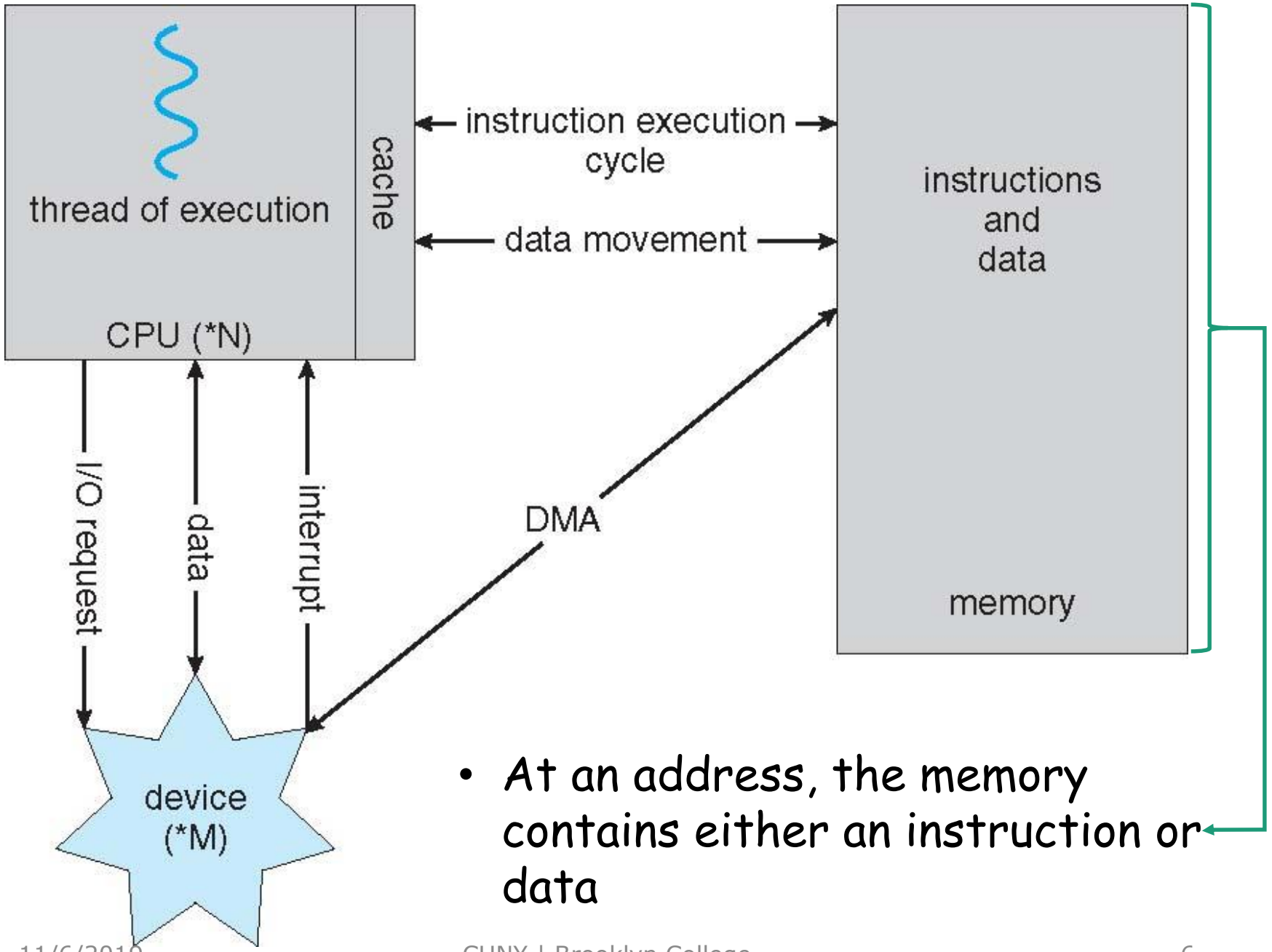
Recap: An Organization of a Computer System

- 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

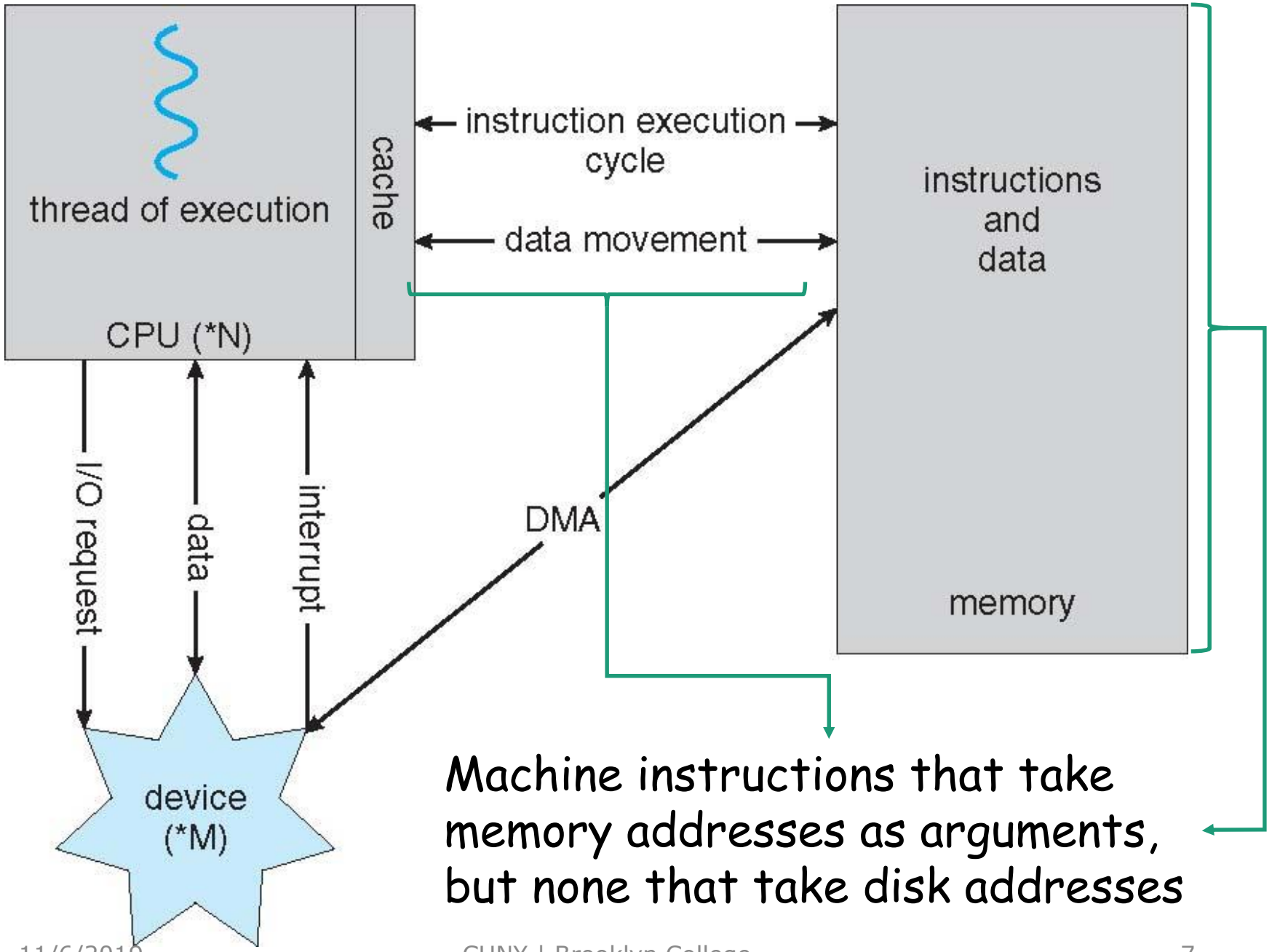




- Memory consists of a large array of bytes, each with its own address.



- At an address, the memory contains either an instruction or data



Machine instructions that take memory addresses as arguments, but none that take disk addresses

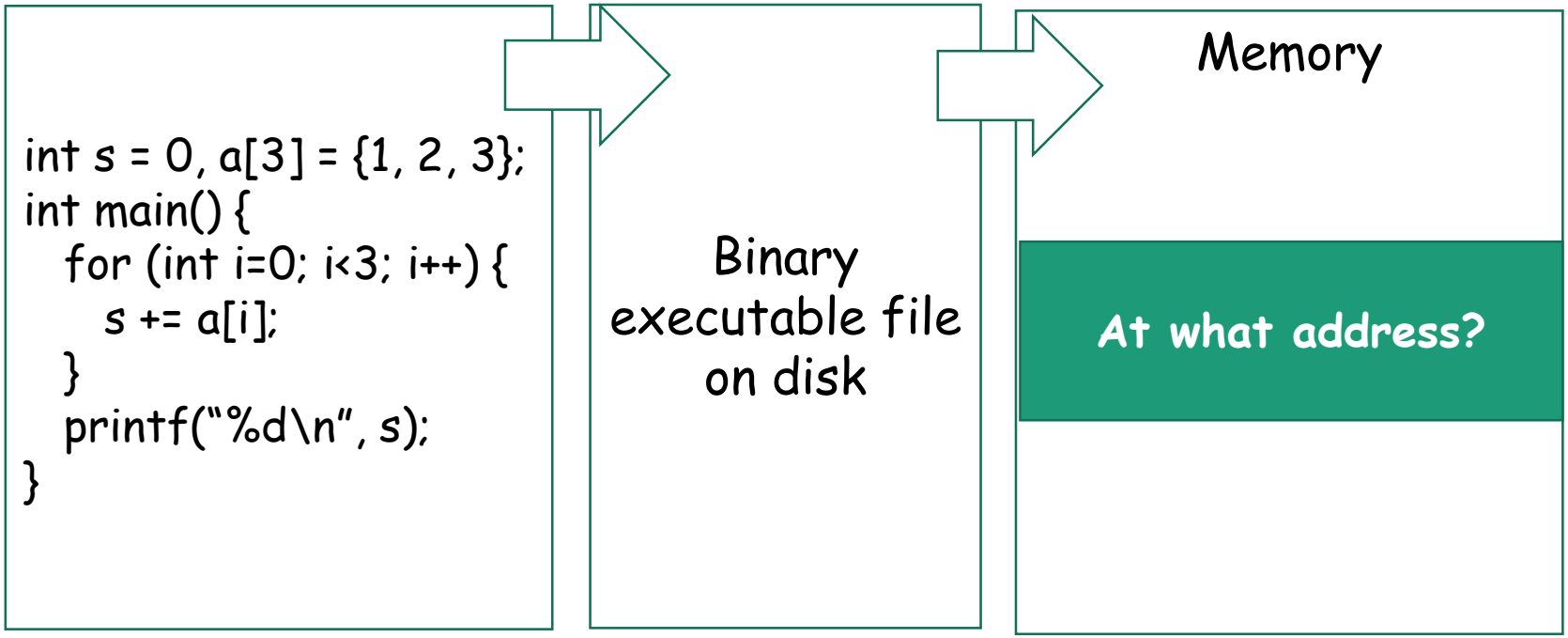
Recap: Addressing Memory

- Memory unit only sees a stream of:
  - (when reading) address + read request, or
  - (when writing) address + data + write request

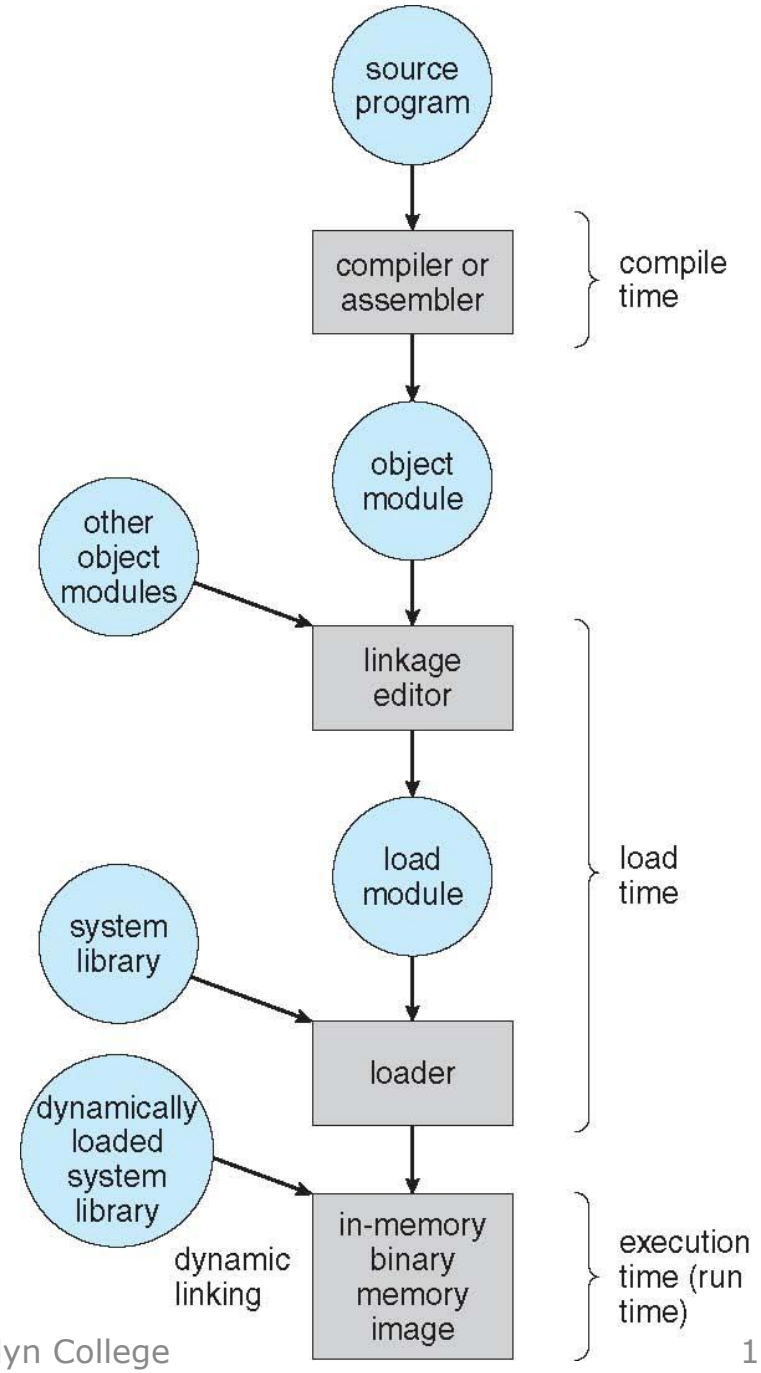
Recap: Key Takeaway

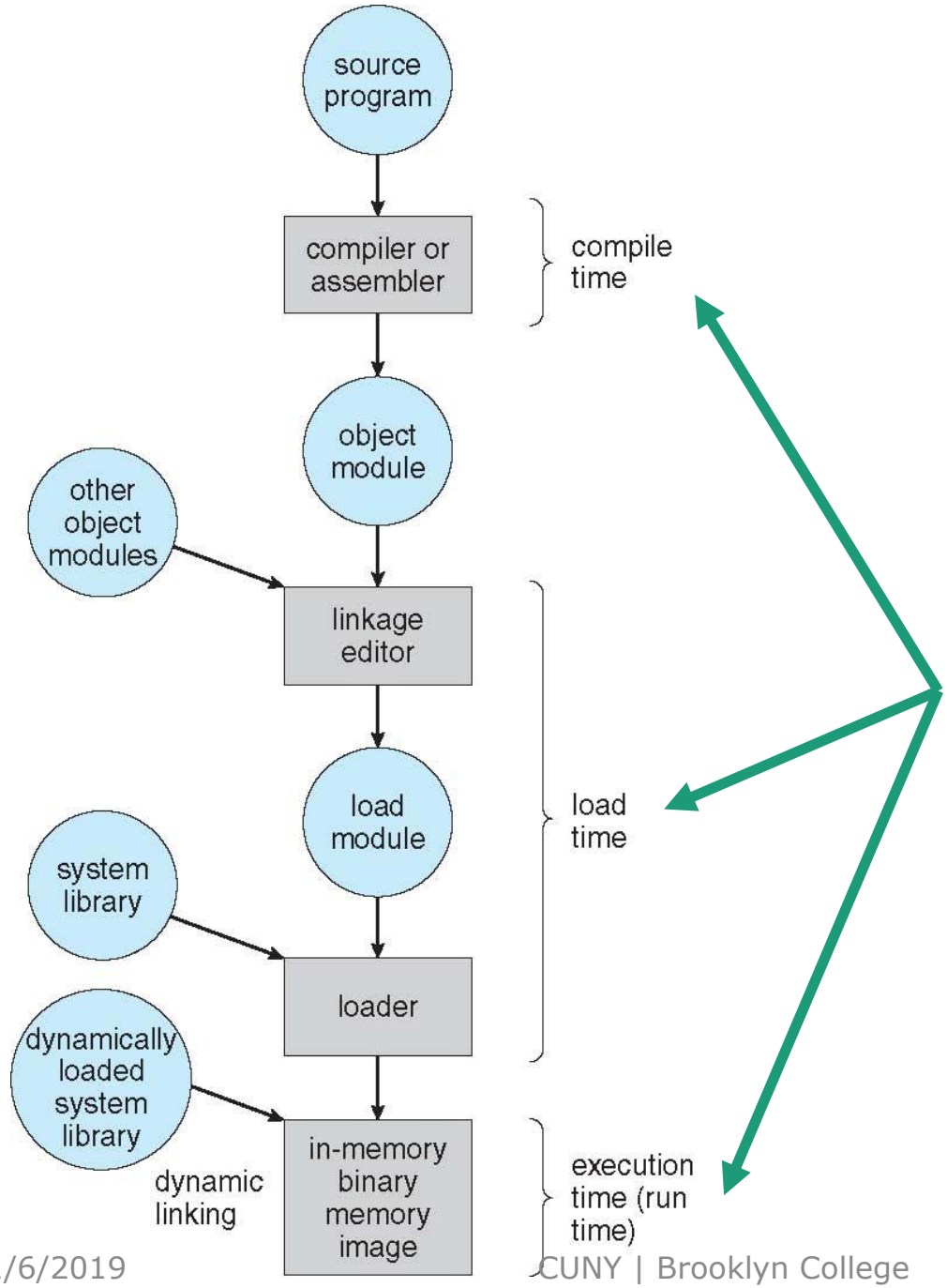
- Memory unit only sees a stream of:
  - (when reading) address + read request, or
  - (when writing) address + data + write request
- But *multiple processes* are running and accessing the memory.
  - How do we allocate memory to processes?
  - How do we ensure correct operation?
- Problem to be solved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

The Address Binding Problem



- A user program goes through multiple step of processing and transformation.





Address binding

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: Compilation Time

- If memory location known *a priori*, *absolute code* can be generated
  - e.g., 1st Process loaded into address 0000 (or other fixed address)
  - Inconvenient to have first user process physical address always at 0000
- Must recompile code if starting location changes

Address Binding: Load Time

- Must generate *relocatable code* if memory location is not known at compile time
  - i.e. gpa is at "14 bytes from beginning of this module" (offset).
  - The address of the beginning of "this module" (starting address or base address) is determined by the loader
  - If the base address changes, we need only reload the user code to incorporate this changed value.
    - Address = Start Address + Offset
    - e.g.,
      - base address 1000 + offset 14 = 1014
      - base address 9000 + offset 14 = 9014

Address Binding: Execution Time

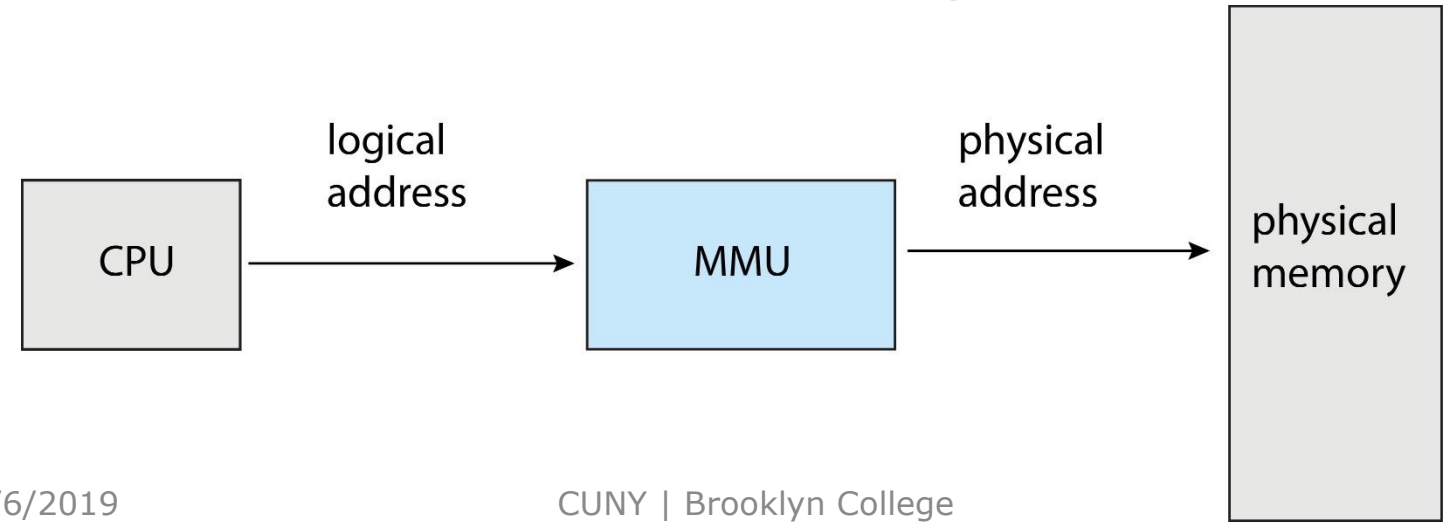
- Binding delayed until run time if the process can be moved during its execution from one memory segment to another
- Most operating systems use this method
- But how?
  - Need hardware support for address maps
  - Most of our discussion is to explain how

Questions?

- Binding instruction and data to memory addresses
  - What? (Meaning)?
  - When?
  - Have we discussed “how”, in particular, address binding at execution time?

CPU and MMU

- To support *execution time address binding*
  - Make a distinction between logical address and physical address
  - Introduce a hardware component: MMU



Logical vs. Physical Address Space

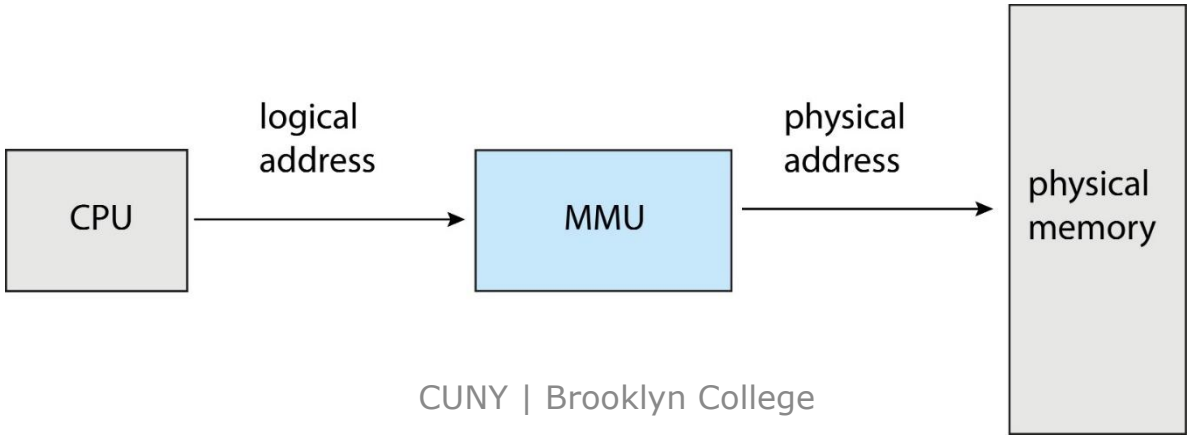
- Logical address
  - generated by the CPU; also referred to as virtual address
- Physical address
  - address seen by the memory management unit (MMU)
- Logical address space
  - the set of all logical addresses generated by a *program*
- Physical address space
  - the set of all physical addresses generated by a *program*

Execution-Time Address Binding

- 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
- 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

Execution-Time Address Binding via MMU

- MMU is a hardware device that at run time maps virtual to physical address (address binding)
- 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 addresses are bound to physical addresses

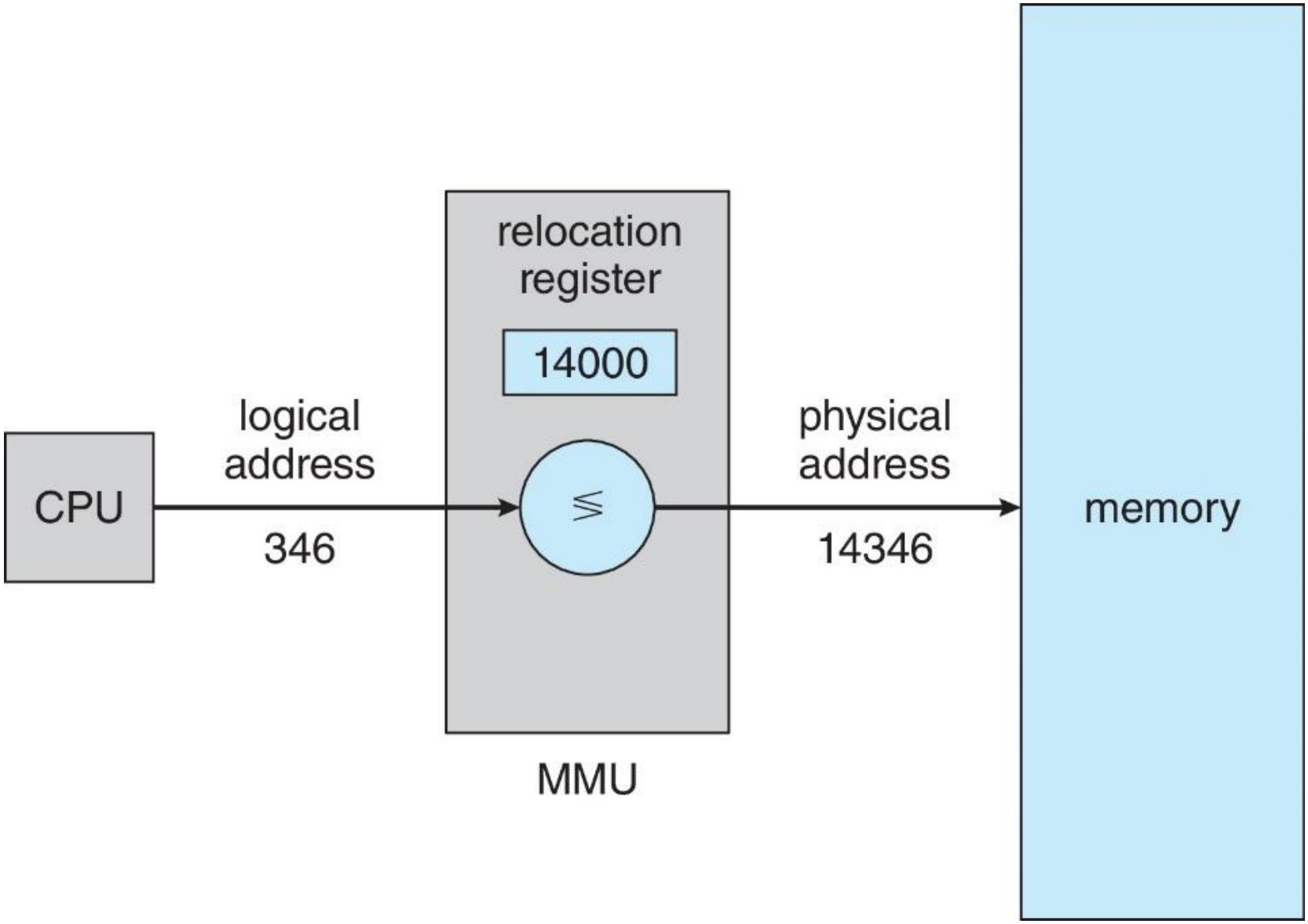


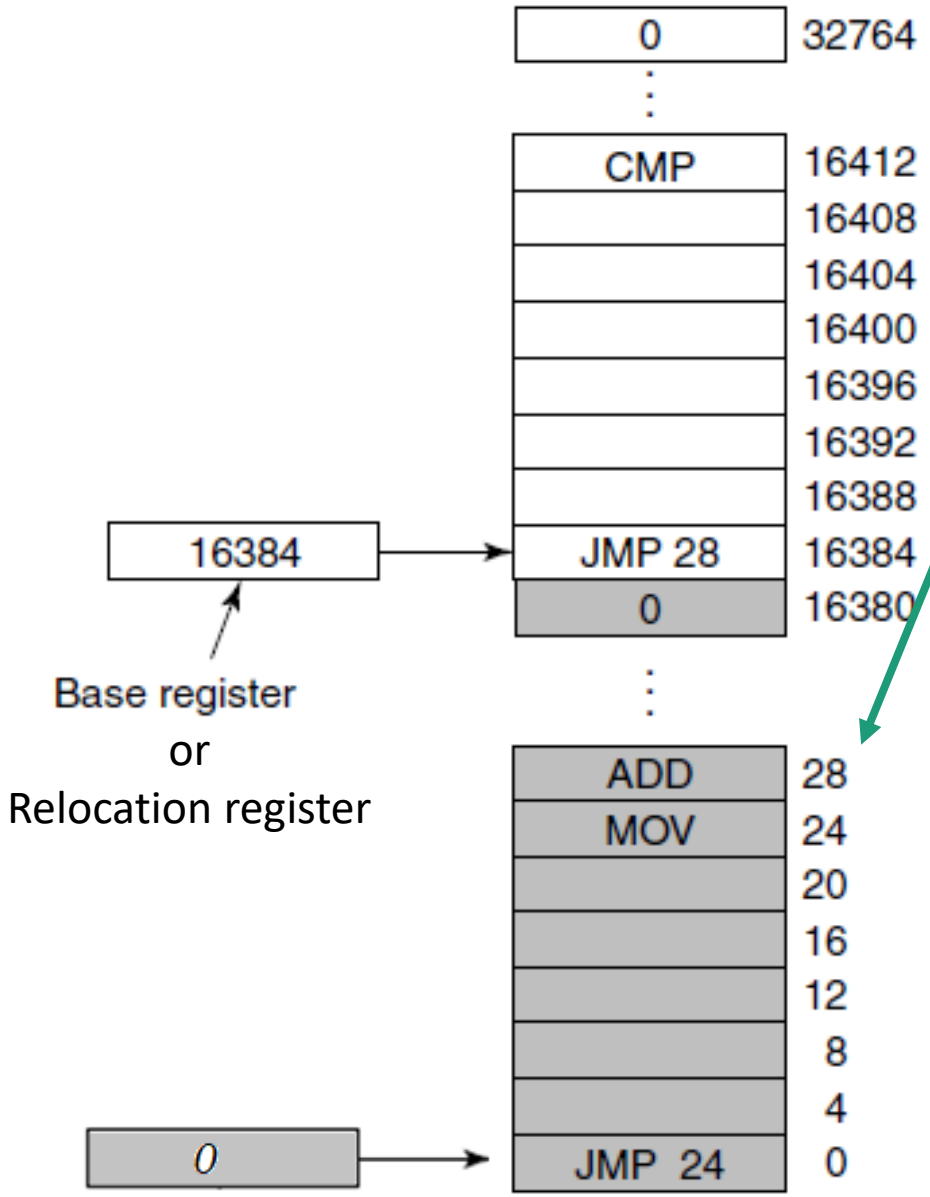
Questions?

- Concept of logical and physical addresses and address spaces
- Concept of address binding
- Concept of exestuation-time address binding and MMU
- How do we bind logical address to physical address?
  - Many methods were developed

Base-Limit Register Scheme

- Consider a relocation register or base register scheme
  - The base register is also called the relocation register, or vice versa
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - relocation register + offset (in this case, logical address) → physical address


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?

- Relocation register scheme
  - What is it?
  - How do we bind logical address to physical address?
  - What does the operating system need to do during a context switch?

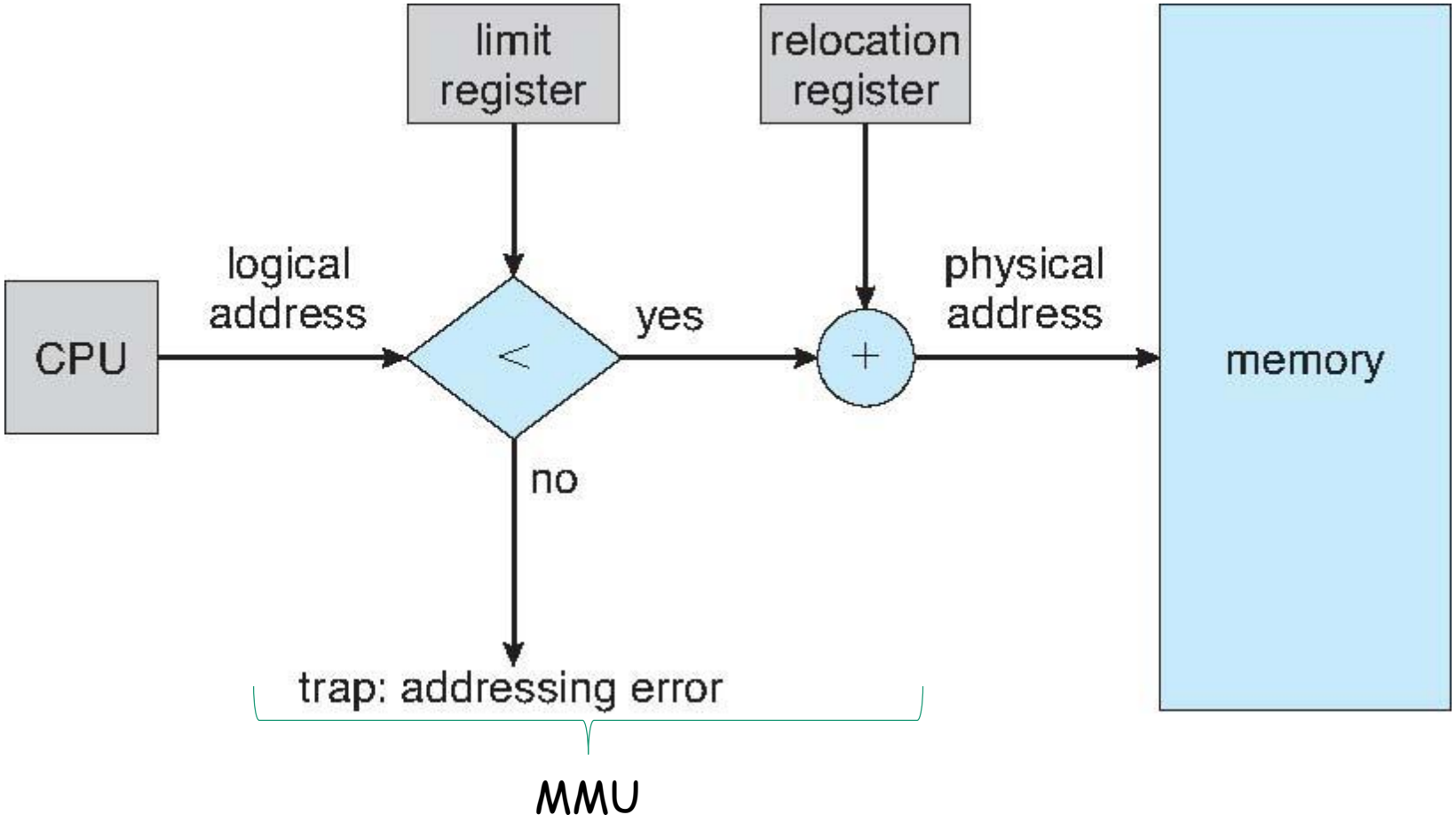
Memory Allocation

- 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 protection?
  - Limit register method
- Memory allocation?
  - Variable partition method

Relocation and Limit Registers



Memory Allocation

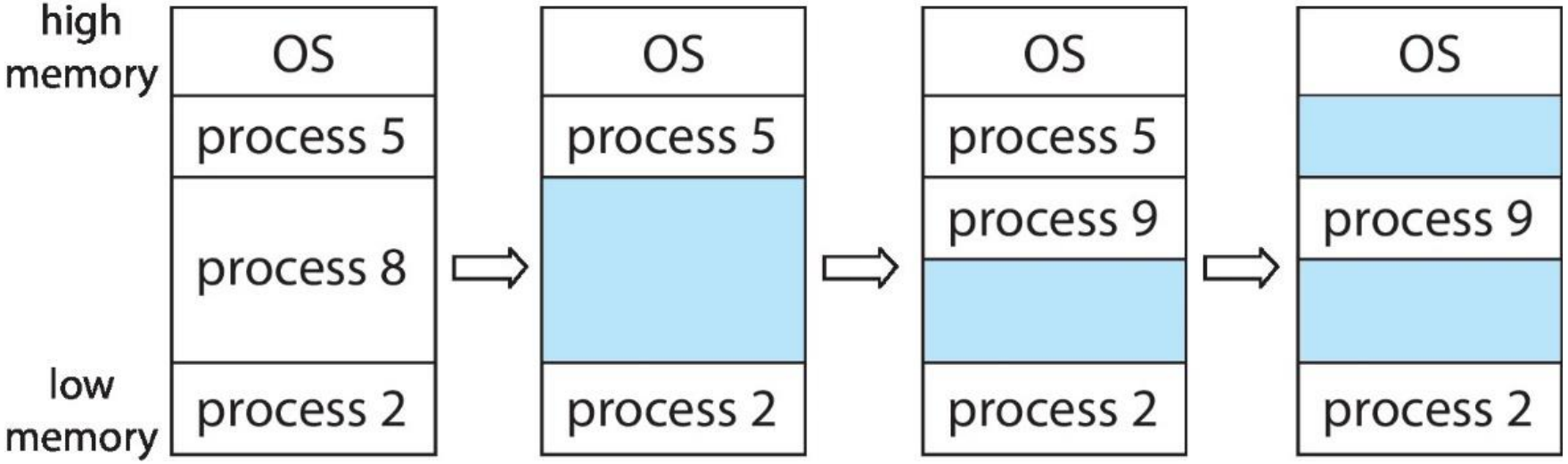
- Assign processes to *variably* sized partitions in memory, where each partition may contain exactly one process
- Partition can begin at any address
- A variable partition scheme
- Continuous allocation
  - A process is given a partition of physical addresses
  - Physical address space of a process is then continuously allocated

Variable Partition

- Operating system maintains information about:
  - allocated partitions
  - free partitions (hole)
- 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

Memory Holes

- block of available memory
- holes of various size are scattered throughout memory



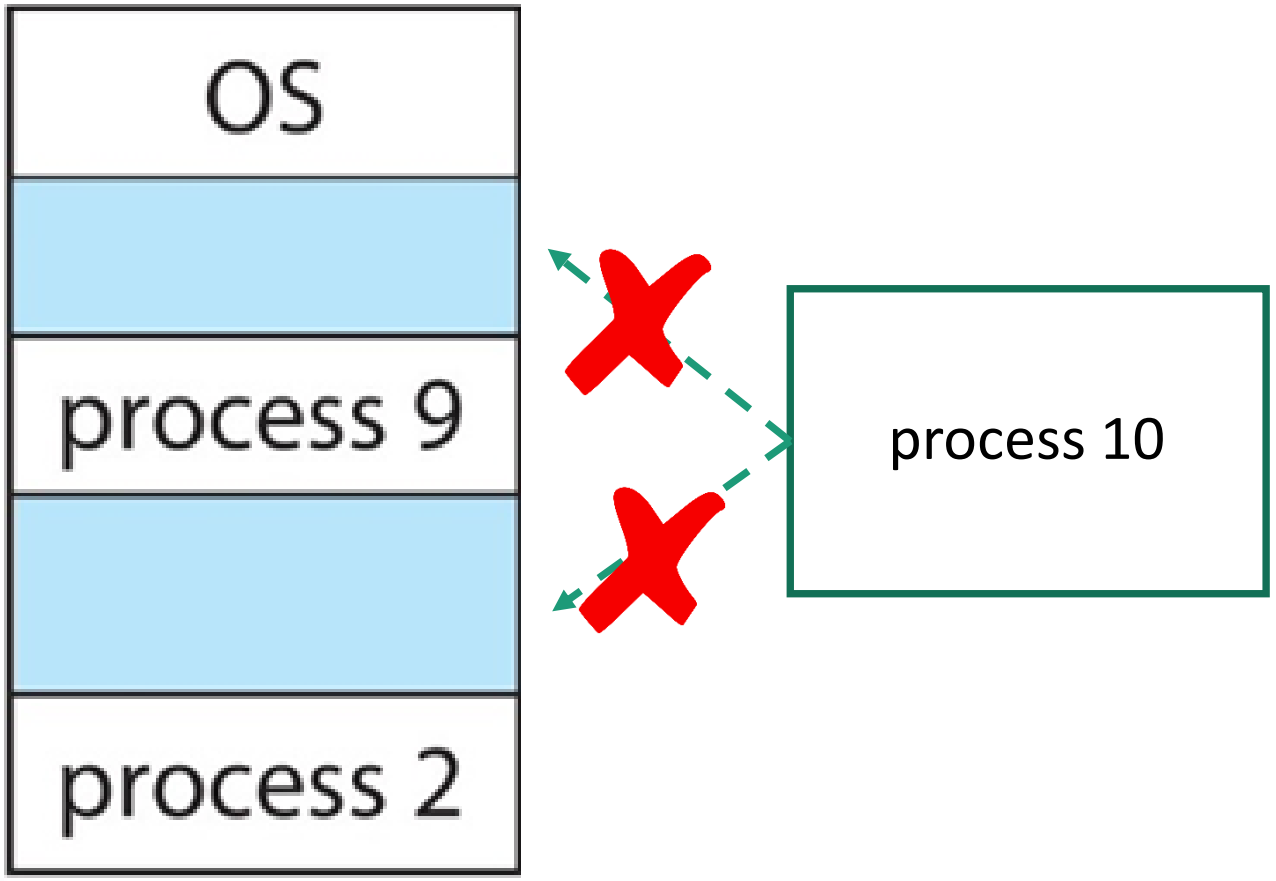
Dynamic Storage-Allocation Problem

- 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

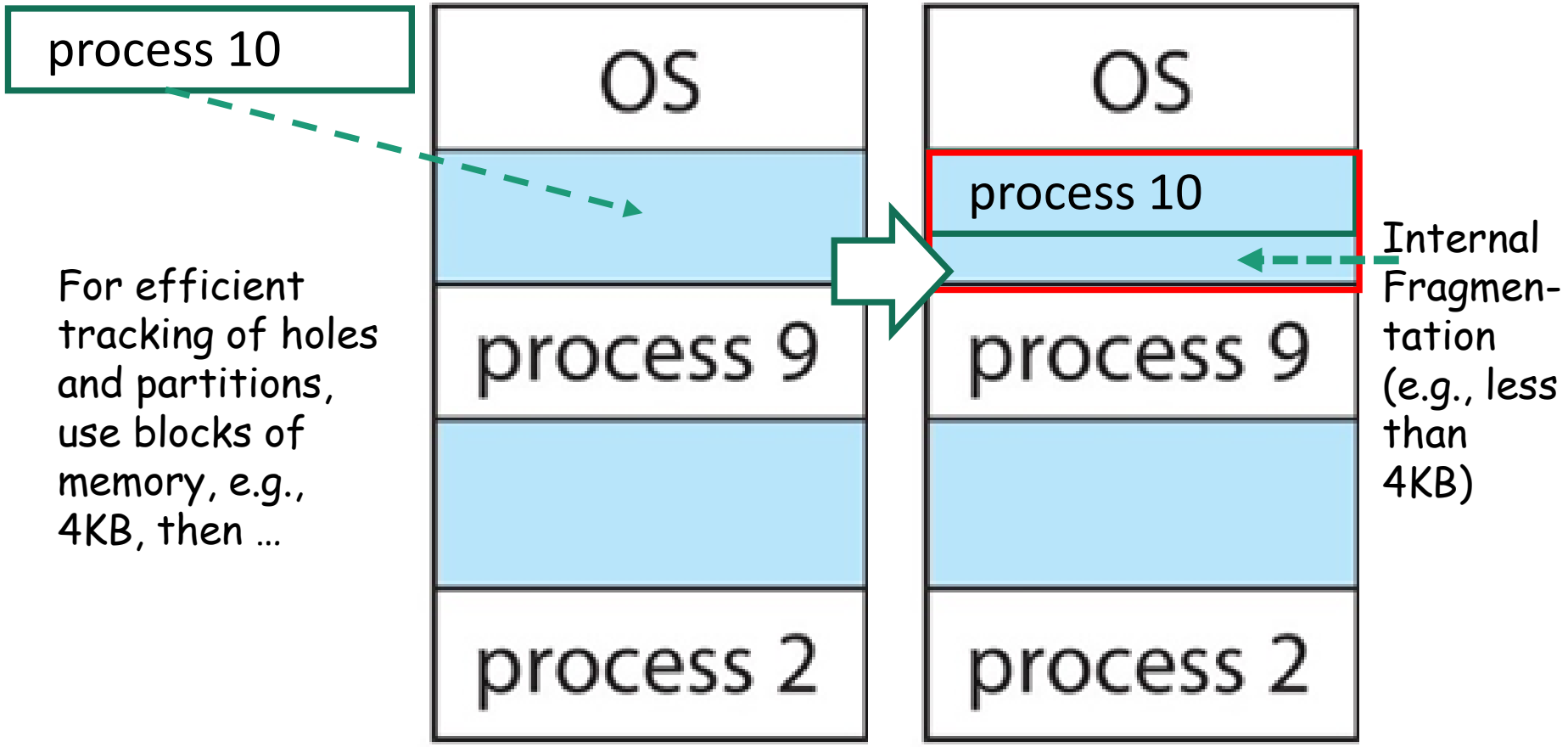
Fragmentation

- 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

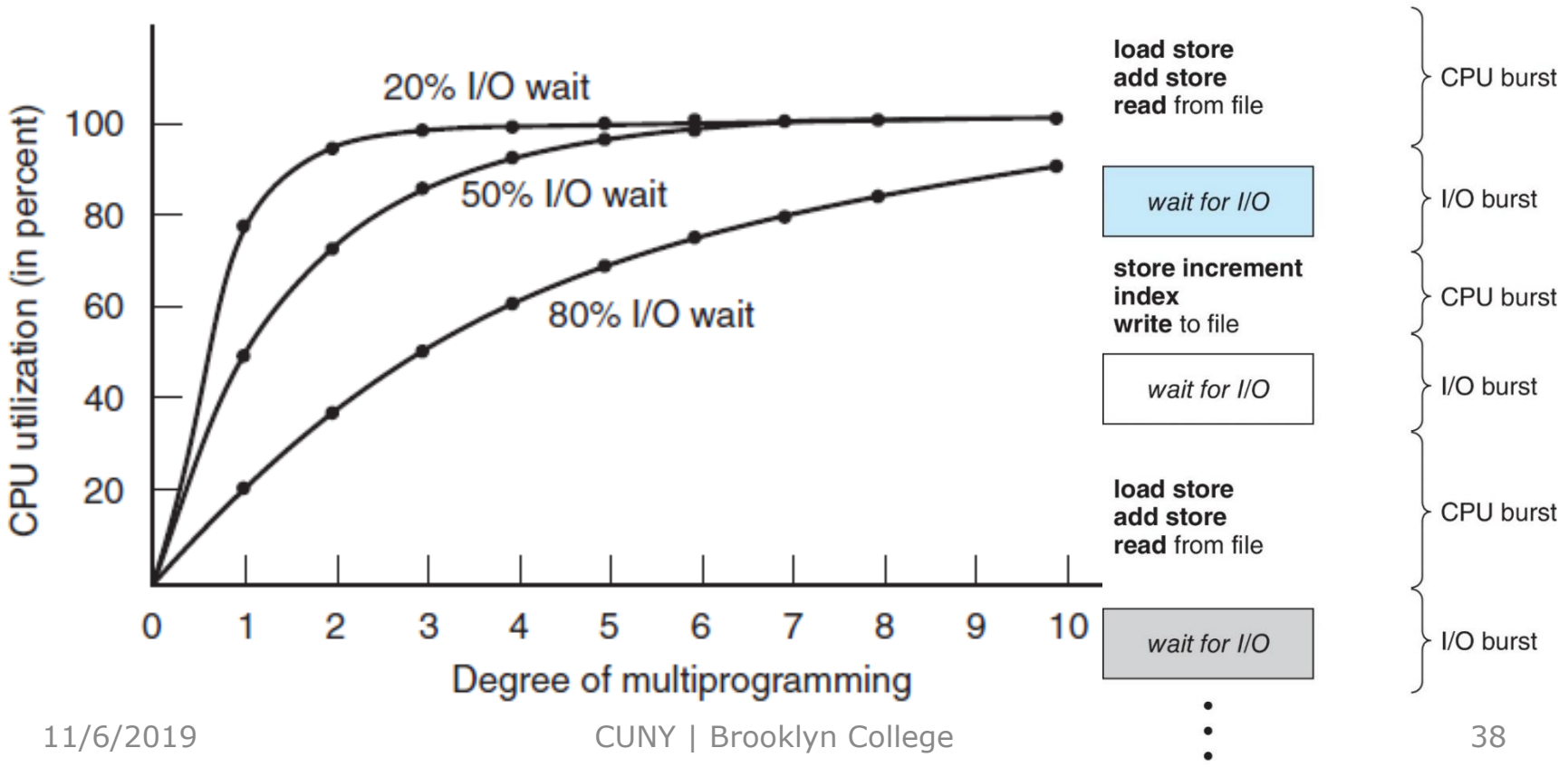


Combating Fragmentation

- 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

Degree of Multiprogramming

- Degree of multiprogramming limited by number of partitions



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?