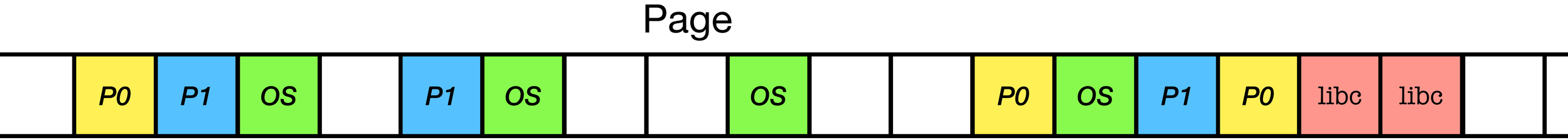
Memory Management Overview

File Systems (programming)
due tonight!

Outline

- Considerations of memory management
- Revisiting address translation
- Memory management strategies

The State of Physical Memory



Concern 1: Memory Protection

Concern 2: Memory Sharing

Concern 3: Flexible Memory Placement

Concern 4: Sparse Addresses

Concern 5: Runtime Lookup Efficiency

Concern 6: Translation Table Efficiency

Concern 7: Portability

The State of Physical Memory

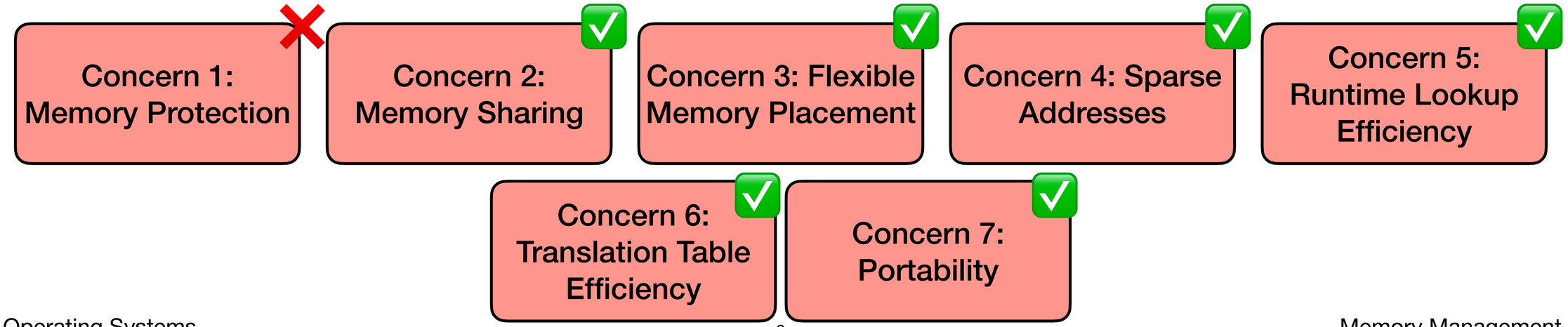
- From CS105, we know that data is stored in memory at a *page*-granularity (e.g., 4kB chunks of data) ➡️ data may be program data, program instructions, shared libraries, memory-mapped files, etc…
- To ensure that applications cannot reach data that is not mapped to its process, the operating system and hardware jointly need to implement *address translation* so that only certain addresses are reachable by certain applications
- To ensure that memory addresses are all reachable and used efficiently, the operating system needs to implement efficient *memory allocation* so that memory remains efficiently reachable with minimal fragmentation

Concerns to Address When Implementing Efficient Memory Management

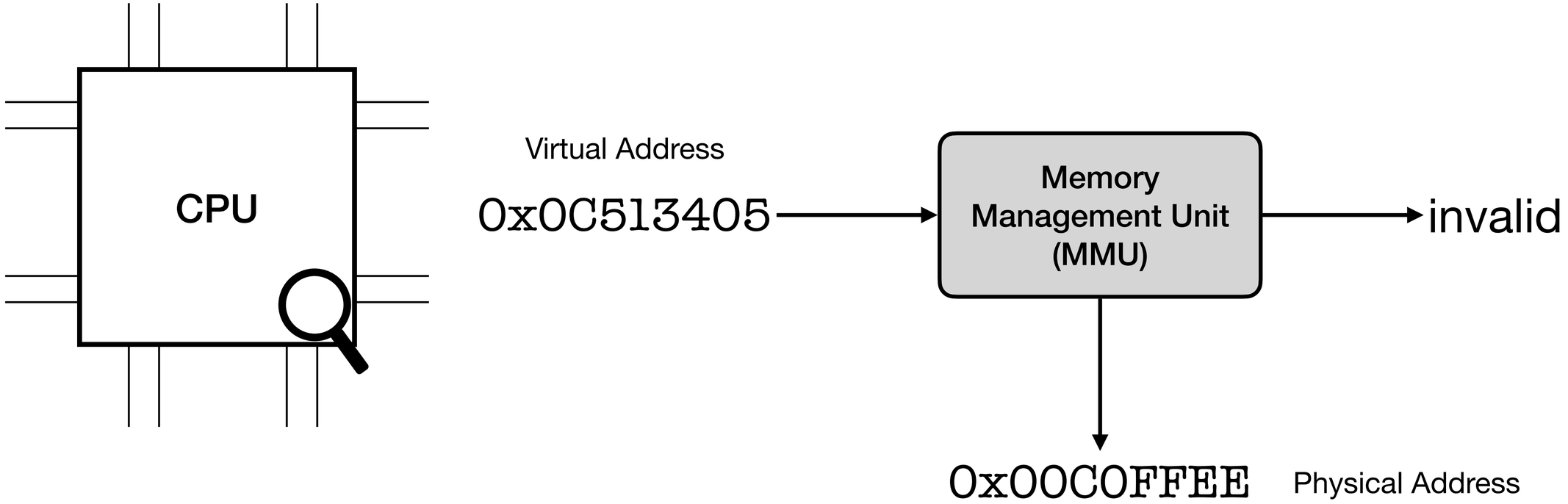
- Memory Protection: processes should be prevented from accessing data associated with other processes
- Memory Sharing: certain data should be accessible to several applications (e.g., shared library program instruction data, pages from forked parent, etc…)
- Flexible Memory Placement: to avoid fragmentation, a program should be able to be placed anywhere in physical memory
- Sparse Addresses: the stack and heap should be able to grow/shrink dynamically over the course of the process' execution, so the whole address space of a process may or may not need to be allocated in advance
- Lookup Efficiency: if we perform address translation, (1) getting the physical address associated with a virtual address should be fast and (2) the translation data structure should be small
- Portability: if we perform address translation, the same mechanism should generally work across different hardwares

Chat with your neighbor(s)!

Consider an operating system implementation in which all applications strictly deal with physical addresses. How would you score such a system on the various concerns of our memory management system?



Review: Address Translation

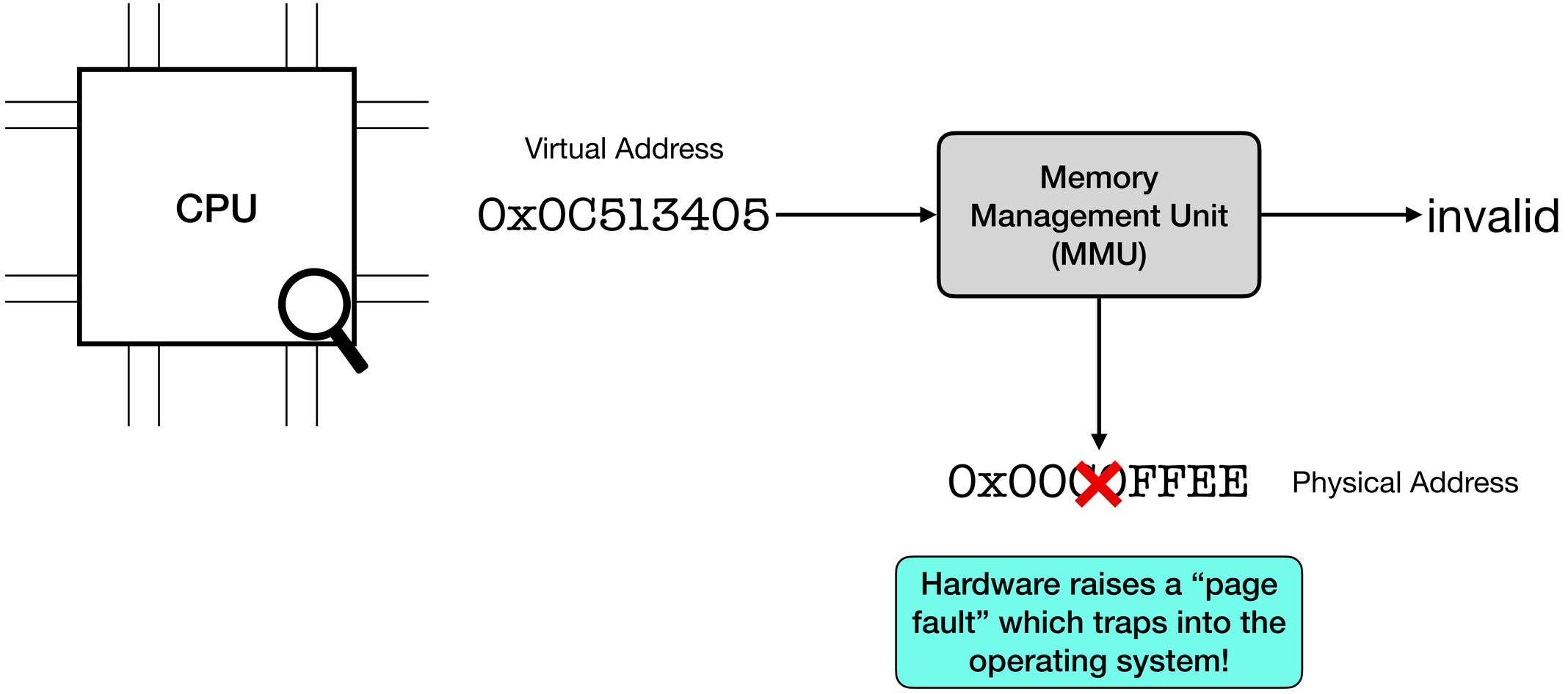


Review: Address Translation

- Whenever we access an address (data or instruction), the process describes those addresses in terms of virtual addresses ➡️ virtual addresses may be used to ensure that applications can only access locations in physical memory that are mapped to its virtual address space
- The processor consults the *memory management unit* (MMU) to produce a physical address associated with the virtual address ➡️ the mechanism for performing the translation is independent from the hardware mechanism
- Nowhere in this picture is the software involved ➡️ accessing an address automatically results in consulting the MMU hardware

Review: Address Translation

A page fault may occur when the process page is not currently mapped into memory

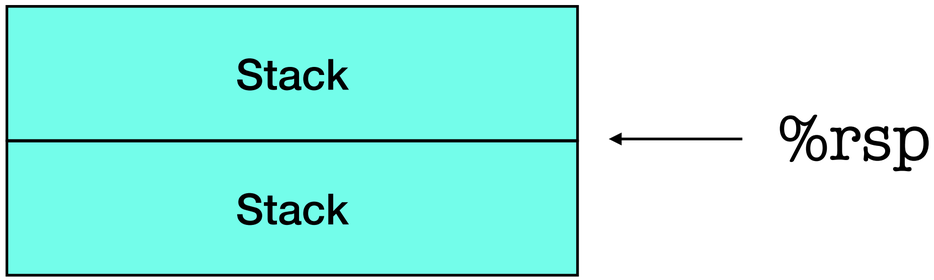


Hardware raises a “page fault” which traps into the operating system!

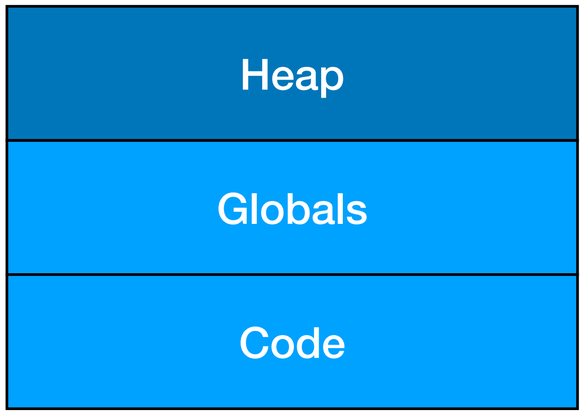
Page Fault Handler

- When a page fault occurs, is generally because the page is not mapped into memory ➡️ the OS kernel code that gets executed by this event is the *page fault handler*
- Demand paging: the first access to a newly allocated page
- Swapping: accessing a page that has been evicted to secondary storage
- Copy-on-Write: writing to a read-only page in the address space of a forked child process
- Invalid Access: the address to access is out-of-bounds, has invalid permissions, etc… (e.g., a segmentation fault)

Efficient Demand Page Faults







On an access to data in the new address, the page fault handler is invoked

How does the OS kernel determine where to map the physical page?

Efficient Demand Page Faults

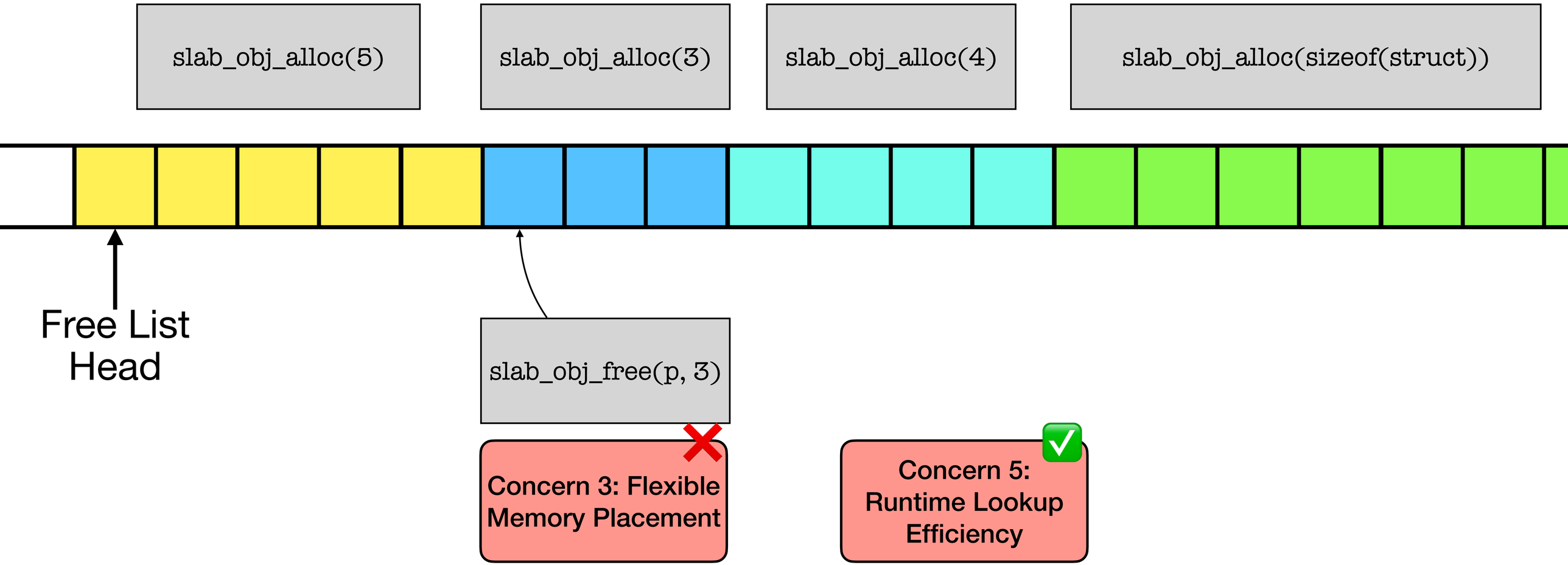
- Address space may be increased by growing/shrinking the stack pointer or the `brk` pointer (top of the heap)
- When the page fault handler is invoked for a demand page fault, it is because there exists an address in the application address space that has not been created in physical memory yet (e.g., the OS kernel allocates the address space lazily)
- When allocating the space for the unmapped address, the OS kernel needs to determine both the size and location to where the address should be mapped ➡️ both of these considerations affect the potential *fragmentation* of the memory space
- To make efficient memory allocation, we need to know where the next location to allocate will be as quickly as possible ➡️ this requires knowing the size of the allocation and potentially performing multiple allocations of small chunks as needed

Chat with your neighbor(s)!

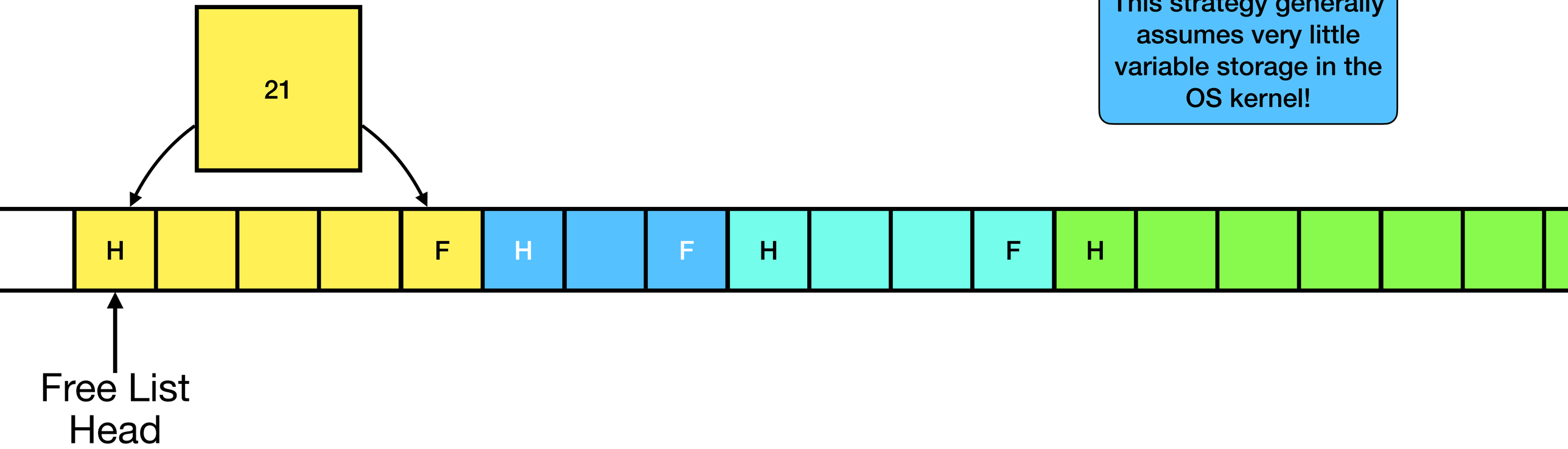
Suppose our OS kernel needs to handle a demand page fault. What is a strategy that the OS kernel could implement to know where a chunk that is both free and big enough could live in physical memory?

Our design space needs to consider both the organization of data in free memory and the data structures that make that memory accessible from OS kernel data structures!

Slab Allocation



Review: Free List Allocation



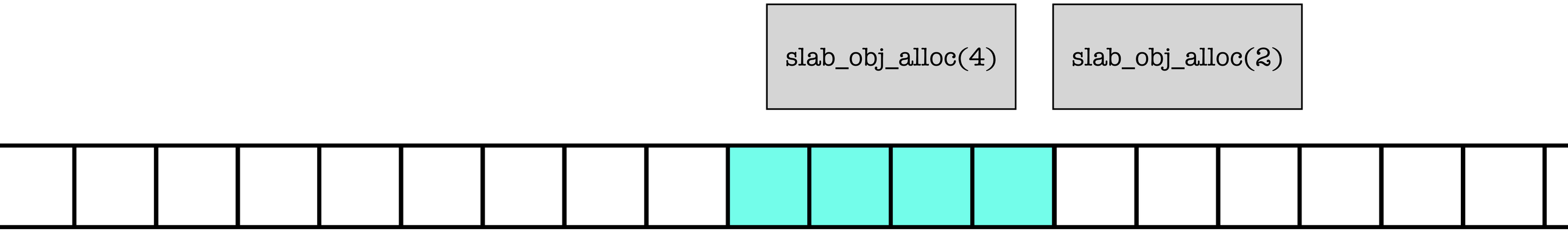
This strategy generally assumes very little variable storage in the OS kernel!

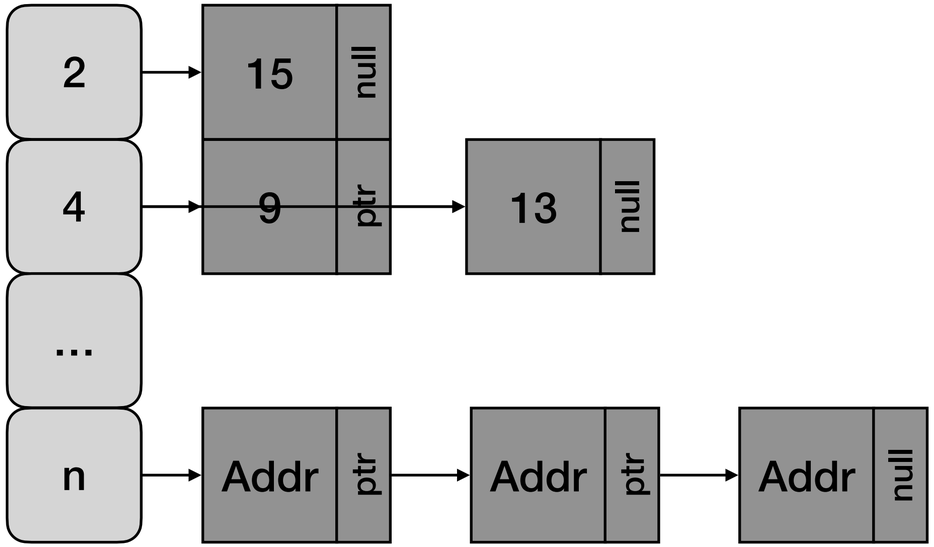
✅ Concern 3: Flexible Memory Placement

❌ Concern 5: Runtime Lookup Efficiency

❌ Additional Concern: Metadata Overhead!

Buddy Allocation





Concern 3: Flexible Memory Placement ✅

Concern 5: Runtime Lookup Efficiency ✅

Additional Concern: State in OS kernel! ❌

Efficient Demand Page Faults

- To handle a demand translation miss (e.g., a demand page fault), the OS kernel needs to efficiently manage the available space in memory ➡️ this requires balancing fragmentation with efficiency to achieve overall performant allocation
- When creating an efficient system, we need to consider the metadata that is tracked in memory versus in the OS kernel code (note: this metadata also lives in memory — just not on a per-block basis!)
- Modern OS kernels use “buddy allocation” to efficiently manage their space, which works by tracking all possible free sizes in the memory space and handling allocations on a per-size basis ➡️ if no chunk exists at the size available, borrow one from the next biggest chunk and “split” the space… when freeing, look to coalesce with a buddy!

Summary

- Virtual addresses and address translation are used to ensure isolation between processes while still allowing for some sharing between “chunks” of memory ➡️ also does not require application to manage/coordinate its physical addresses!
- Page faults are raised by the hardware whenever a translation fails (either legally or illegally) and triggers the OS kernel’s page fault handler
- To handle demand misses (e.g., the need for allocating a new chunk of physical memory), the OS kernel needs to implement a memory allocation strategy ➡️ modern OS kernels tend to use buddy allocators to manage their physical memory