Protecting Kernel Memory

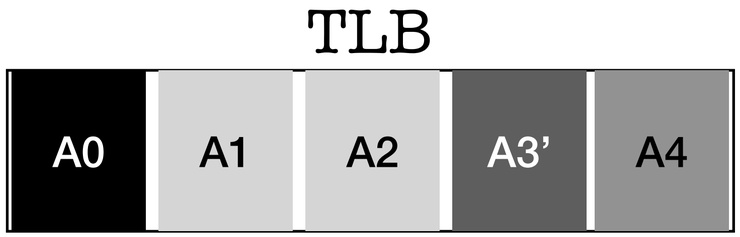
FS Resubmission due Friday;
MM Assignment due Nov. 13
(Friday next week);
Philippe's Boba talk tomorrow!

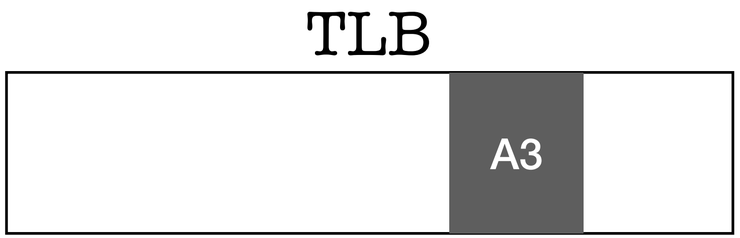
Outline

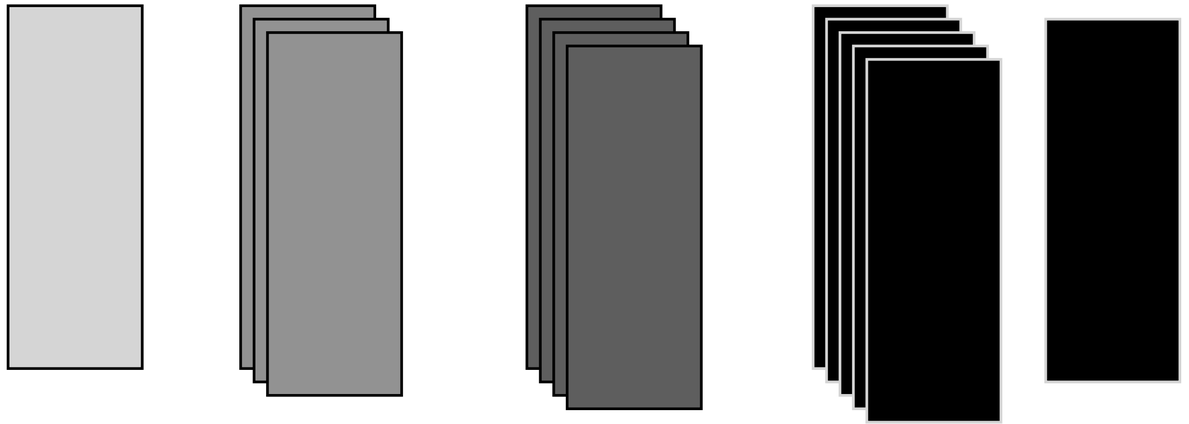
- Final word on TLB shootdowns
- How to protect kernel memory
- Implications of mapping kernel memory

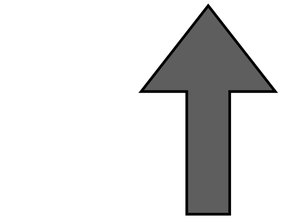
Review: TLB Synchronization

We have a synchronization issue between cached versions of the TLBs!







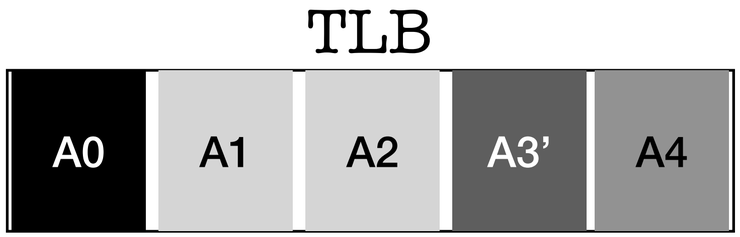


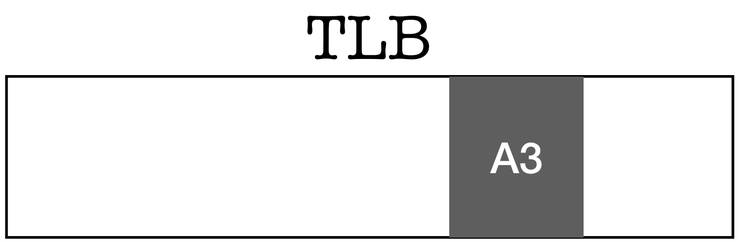
Update PTE

Page fault! Map a new frame to the page table

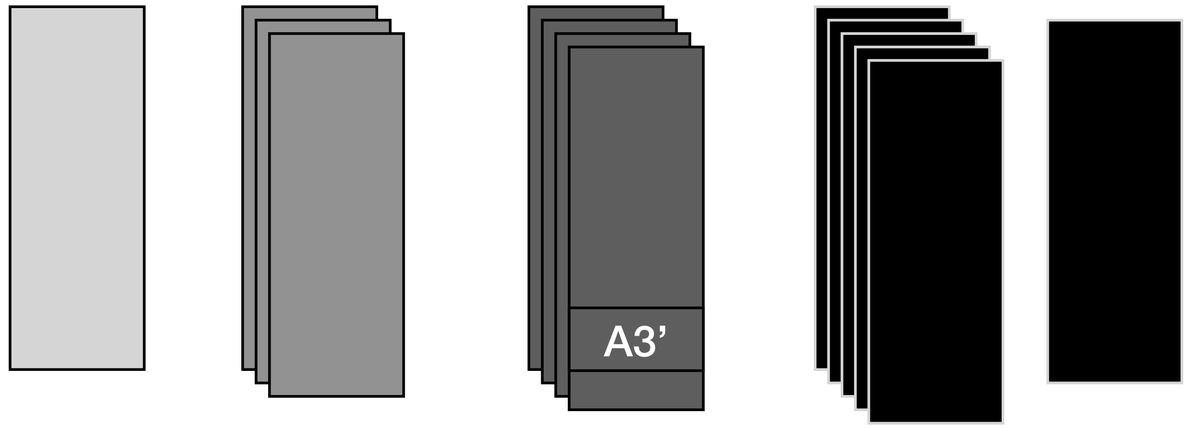
To handle this case, we need a *TLB shootdown* operation to invalidate cached copies of PTEs in other TLBs… updates need to be written through to copy of PTE in memory as well!

Review: Implementing TLB Shootdowns





Page fault! Map a new frame to the page table



Copy of the page table in memory serves as the true state of the data structure

Cached copy can never be out of sync with the version in memory!

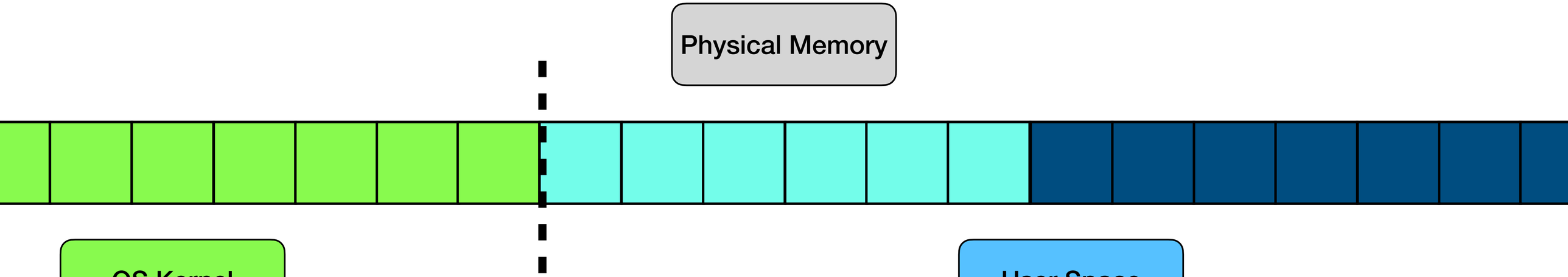
The implementation of this coordination is described as a *TLB shutdown algorithm* (e.g., using the TLB as a write-through cache)

Chat with your neighbor(s)!

If it's the case that our OS kernel is not protected from our application, what are some of the types of malicious behaviors that a user can achieve?

- Modify page access permissions
- Access to another process' address spaces
- Ability to modify the file system state
- And more!

Kernel Protection Without Virtual Memory



To protect the kernel address space from users, the MMU utilizes a *fence* to protect the "kernel addresses" in the physical address space

When creating a process, the OS kernel needs to allocate its address space in physical memory

Kernel Protection Without Virtual Memory

- To ensure that users could not access code in the OS kernel, the MMU hardware historically enforced address space protection by ensuring that user space memory could not be allocated on the other side of a *memory fence*
- The memory fence describes a preset boundary in hardware that describes the maximum possible size that the OS kernel can be ➡️ this allows the MMU to detect illegal access and throw an interrupt before an access could be made to one of these addresses
- In order to invoke the OS kernel, the interrupt handler needed to perform a context switch into the OS kernel handling code (e.g., saving the register/ stack state of the user space program) before executing the interrupt handler

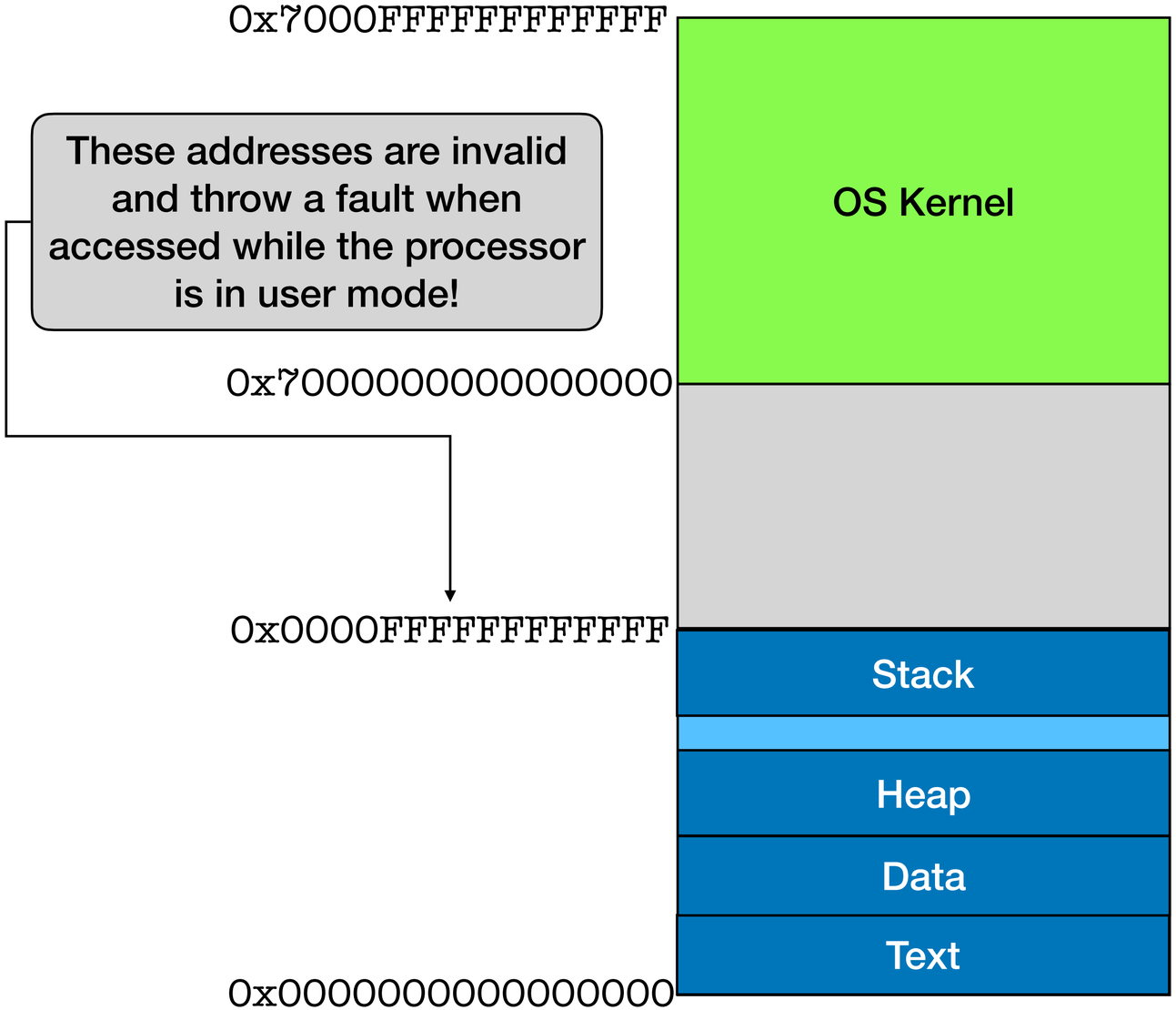
Chat with your neighbor(s)!

When we perform an interrupt, the operating system is invoked to implement some application-tasked behavior. Would implementing a debugging breakpoint using an interrupt require more or less state than a standard system call?

Our operating system needs to have knowledge of the state of data in user space!

Using a memory fence to protect the OS kernel limits the ability to implement other useful OS functionality

(Re-)Interpreting the Virtual Address Space



These addresses are invalid and throw a fault when accessed while the processor is in user mode!

To allow the OS kernel to access user space data, the address space of every process has a copy of the OS Kernel mapped into it!

Maximum Virtual Address

0x0000FFFFFFFFFFFF

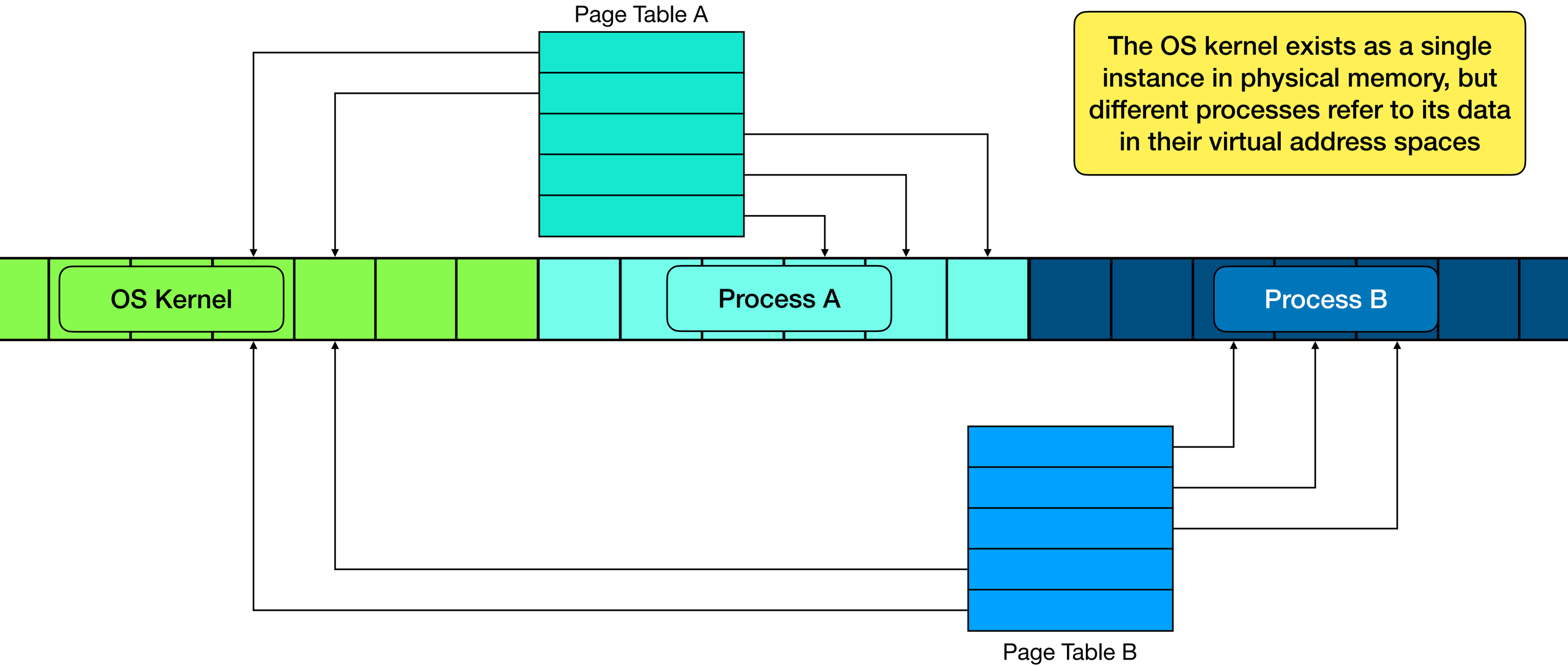
Virtual address is 48 bits, and the upper 12 bits are unused

“Upper addresses” in the virtual address space maintain a copy of the operating system!

(Re-)Interpreting the Virtual Address Space

- In order to protect the OS kernel from the user space, the OS kernel tends to be written to a protected part of the *physical* address space behind a “memory fence”
- We often want to allow the OS kernel to have access to the data and address space associated with a process to perform OS-assisted tasks ➡️ to achieve this, we run the OS from the virtual address space *of the process*
- The OS kernel code needs to be mapped to the same virtual address space, meaning that OS kernel can continue to read user space variables during its execution as it operates with the *same page table* as the rest of the process

Sharing the Operating System



Chat with your neighbor(s)!

Consider the case in which many long running processes are created by applications using the OS. What is the implication of this behavior on OS memory? How will this affect our processes?

Creating lots of processes will grow the space to maintain these structures in OS memory

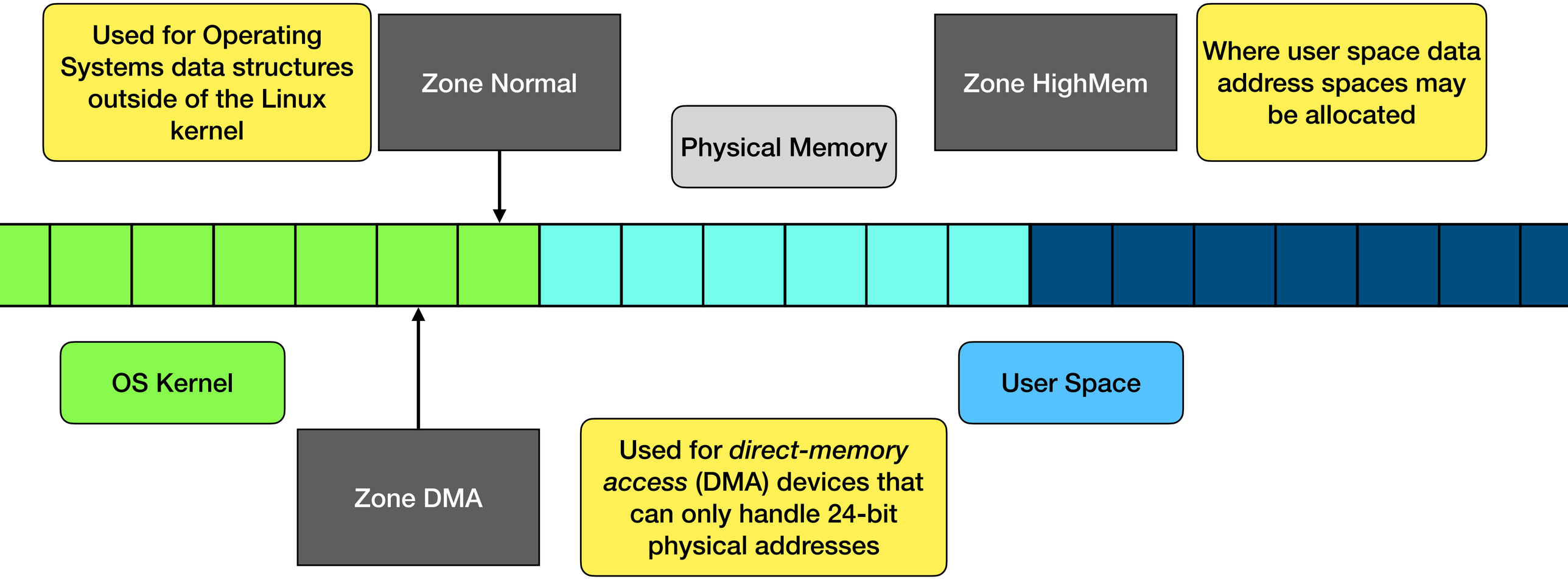
We don't want to constantly update the virtual address space of every process with the changes to the OS state!

The "useful" parts of the OS to the application can be distinctly thought of as the "OS kernel" — this includes its code and essential data structures (e.g., the page table)

What to Share from the OS Kernel

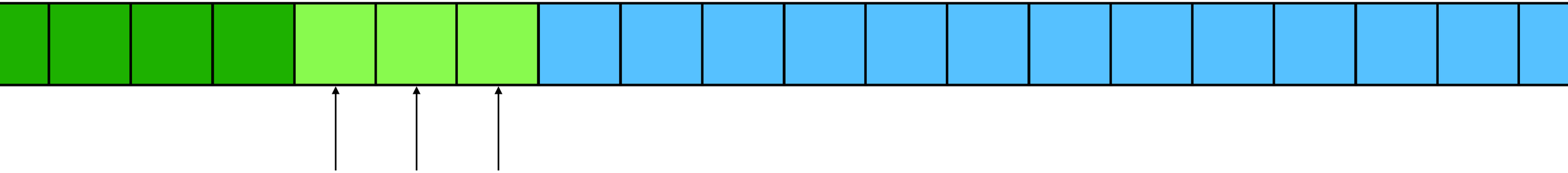
- Seeing as we maintain references of the OS into the virtual address space of the processes running in the system, we need to ensure that only the most essential pieces of the OS are mapped into this address space for the execution of the process ➡️ the data at these addresses are generally described as the *OS kernel*
- The OS kernel is comprised of the code and data structures associated with the subsystems for: (1) memory management and (2) system calls and interrupt handlers
- The rest of the OS data and data structures to implement the remaining subsystems remains mapped in a distinct subregion of kernel memory, which is not mapped to the address space of the process

“Outside” the Kernel in Linux



Chat with your neighbor(s)!

We define the maximum operating system size in hardware to avoid having user memory interfere with kernel memory. Suppose that our OS kernel does not consume enough memory to use the entirety of the pre-allocated space. What could be done with the extra space?



“Outside” the Kernel in Linux

- In Linux, the OS kernel lives in the *lowest* addresses of the physical address space (e.g., 0x0 to the maximum kernel size) to remain backwards compatible with older processors that implemented a memory fence
- Any space that isn’t used by the OS kernel to the end of the “memory fence” is used as a `kmap` region: a set of frames in physical memory that are used to copy user space frames into kernel space for a small subset of possible tasks to be done without explicitly interrupting user behavior
- Any space beyond the OS kernel memory is treated as members of distinct *zones* by the OS kernel: (1) Zone DMA for drivers, (2) Zone Normal for OS data structures, and (3) Zone HighMem for user memory

Windows OS in Memory

- Unlike Linux, the Windows OS does not rely on a predefined storage boundaries to track the OS kernel ➡️ the kernel code and data structures may grow to be very large and stretch beyond the boundary!
- If the operating system data/code lives in pageable memory, then the OS itself will be subject to page faults!
- Pros: the OS can grow to be very large, so the functionality of the OS can be robust ➡️ functionality that would otherwise require a daemon or driver can be implemented natively in the OS
- Cons: if the OS is subject to a paging, then OS code/data may be paged out to disk! This will make OS operations even slower than before

Summary

- In order to protect the address space where the kernel lived, historical devices protected it by giving it a fixed region to be stored behind a “memory fence”
- Nowadays, the kernel is protected by virtual memory and access permissions in the MMU
- Relatedly, to access the virtual address space from a kernel operation (e.g., an interrupt handler), we embed a copy of the OS kernel in the process’ virtual address space
- The OS interprets the non-OS code as being a member of a memory “zone”