

CS 3214

Computer Systems

Virtual Memory

Godmar Back

Brief Review from CompOrg

- Virtual address:
 - addresses used by user programs, linkers, etc. `printf("%p\n", ptr);`
 - Range: $0 \dots 2^{\text{addresswidth}}$
- Physical address:
 - address used internally to address memory; not visible to user
 - Range: $0 \dots X$ where X is memory in computer
- Page: contiguous range of addresses, typical sizes are 4K
 - Virtual page – contiguous range of virtual address
 - Physical page (frame) – contiguous range of physical addresses
- MMU: Memory management unit that maps virtual to physical pages based on information found in *page tables*
- TLB: Translation Lookaside Buffer:
 - Caches such mappings

Virtual Memory

- Is not a “kind” of memory
- Is a technique that combines one or more of the following concepts:
 - Address translation (always)
 - Paging from/to disk (usually)
 - Protection (usually)
- Can make storage that isn't physical DRAM appear as though it were

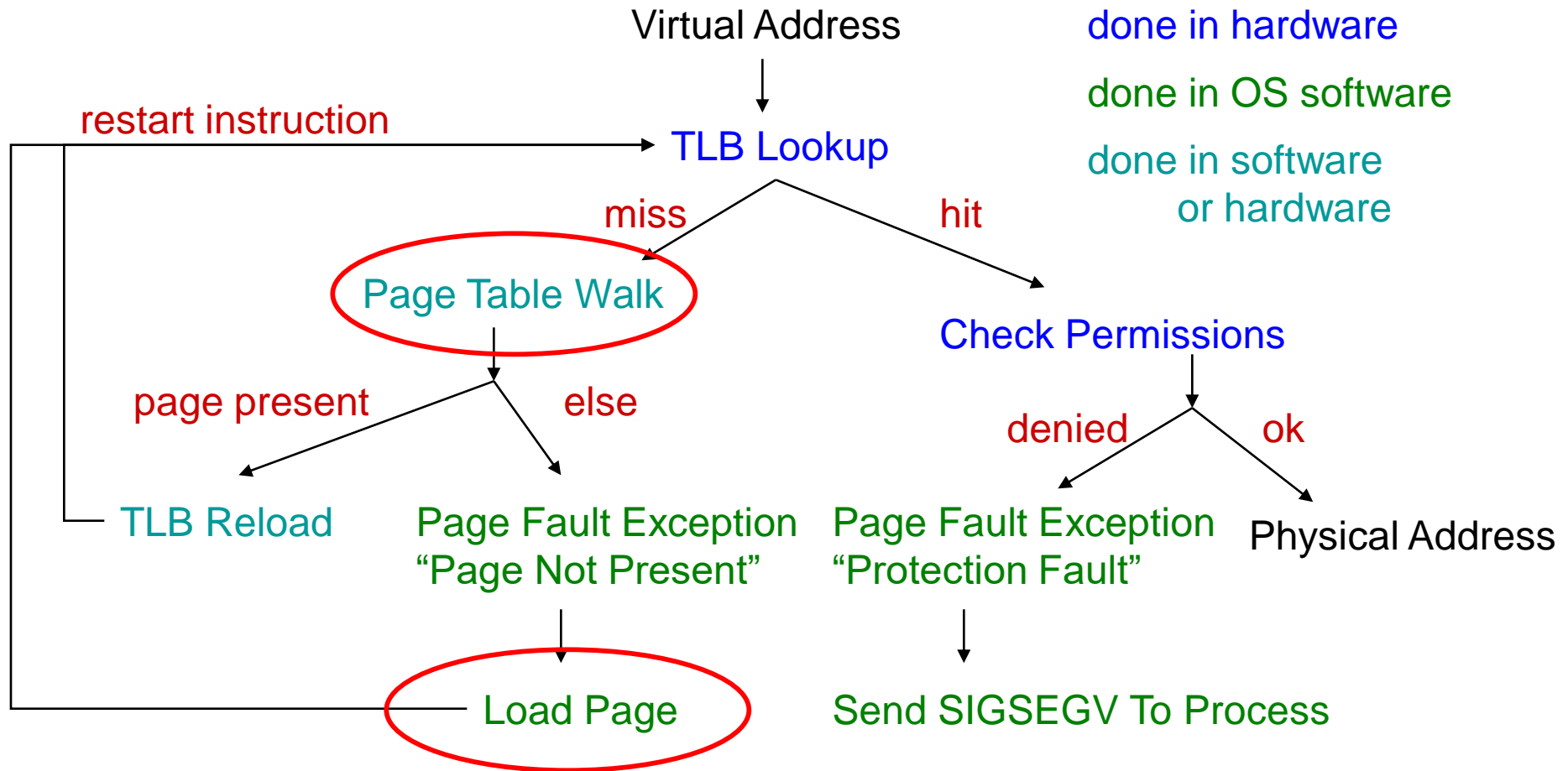
Key goals for Virtual Memory

- Virtualization
 1. Maintain illusion that each process has entire memory to itself
 - Per-process address spaces
 2. Allow processes access to more memory than is really in the machine (or: sum of all memory used by all processes > physical memory)
 - Makes DRAM a cache for disk
- Protection
 1. make sure there's no way for any process to access another process's data unintentionally
 2. protect system-internal data/kernel data

Address Translation

- Provides a way for OS to interpose on memory accesses
- OS maintains *for each process* a mapping { virtual addresses } → { physical addresses } in a per-process page table
 - Which virtual addresses are valid (depends on process memory layout)
 - Where they map to (depends on availability of physical memory)
 - What kind of accesses are allowed (read/write/execute)
- OS manages page tables
 - Based on input/commands from user processes
 - Based on resource management decisions

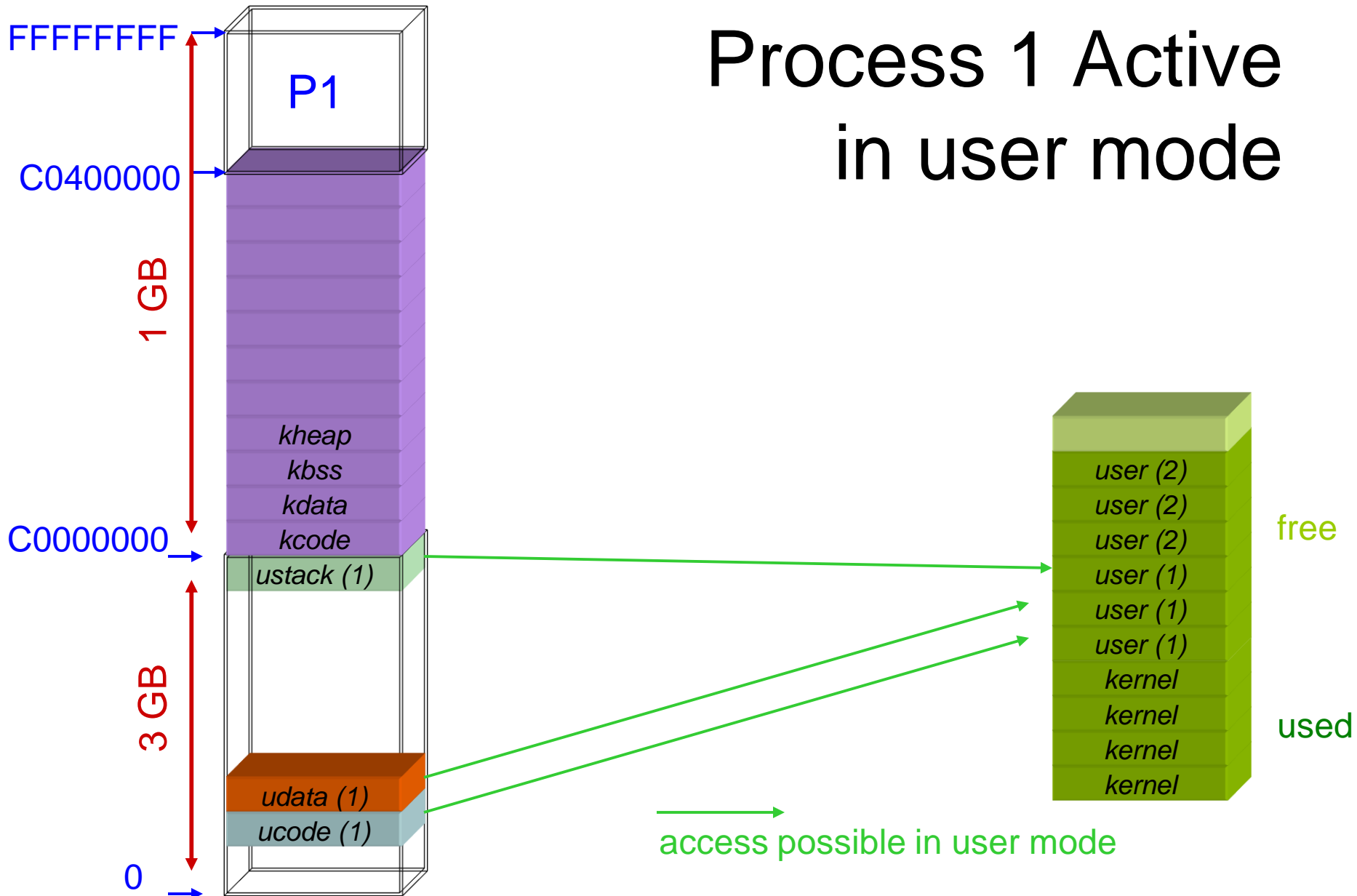
Address Translation & TLB



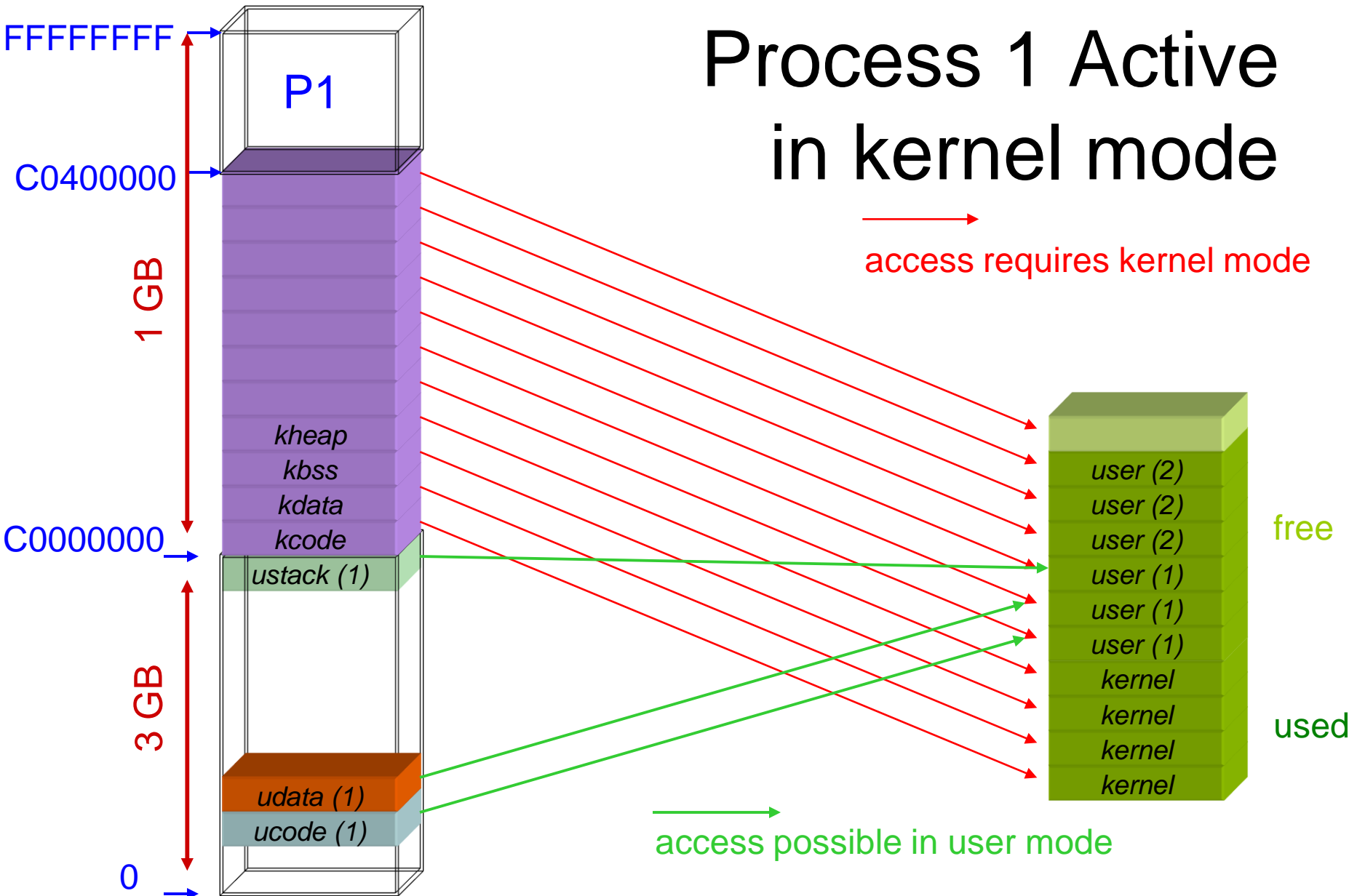
Switching Address Spaces

- Following slides show how virtual-to-physical mappings change on mode switch/context switch/mode switch sequence
 - Show a bit of kernel-level implementation detail
- In multi-threaded case, context switch *may or may not* involve a change in current address space
- Costs of switching address spaces adds to context switch cost
 - Mainly opportunity cost: need to flush TLB & then take the misses to repopulate it

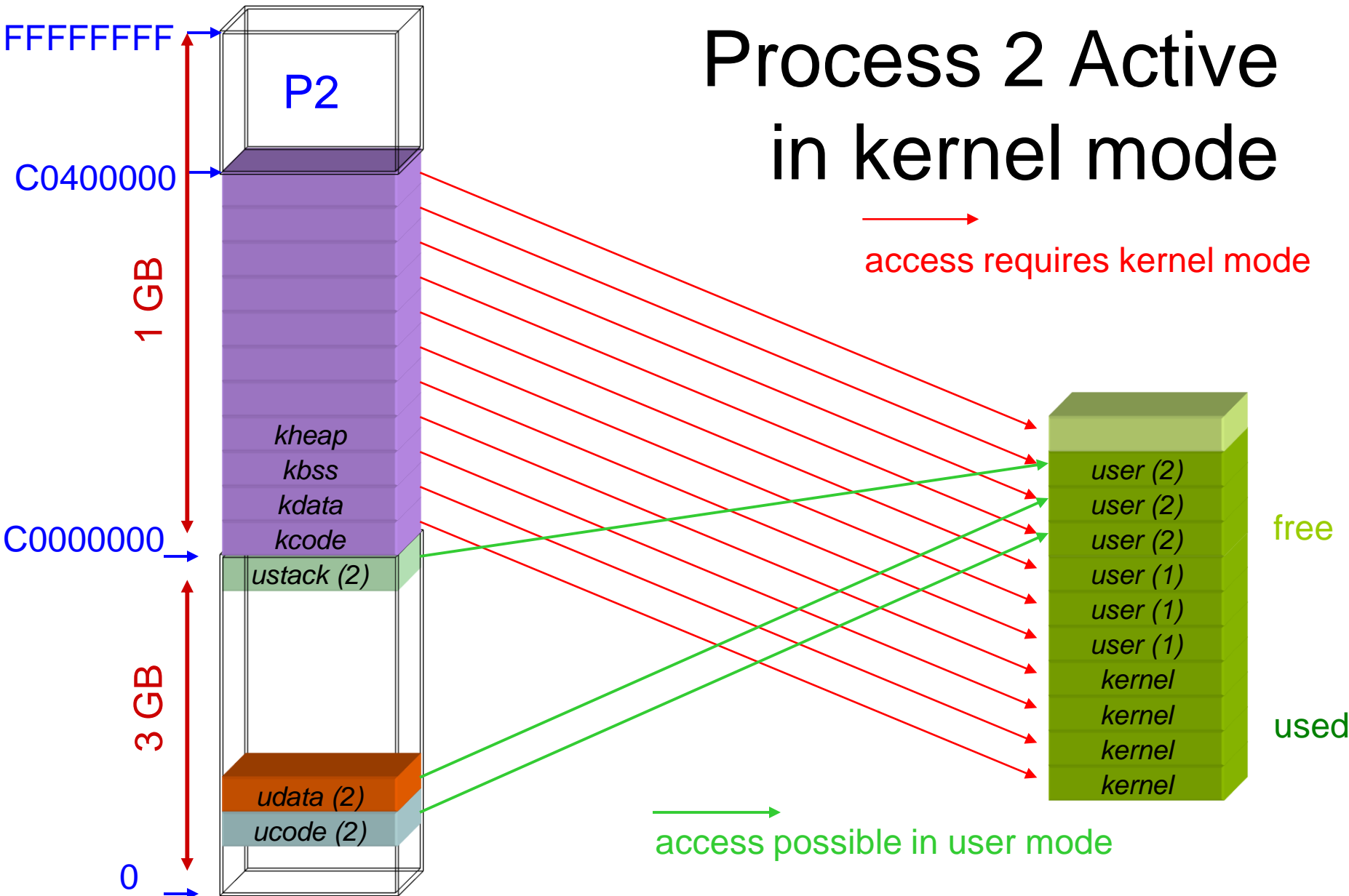
Process 1 Active in user mode



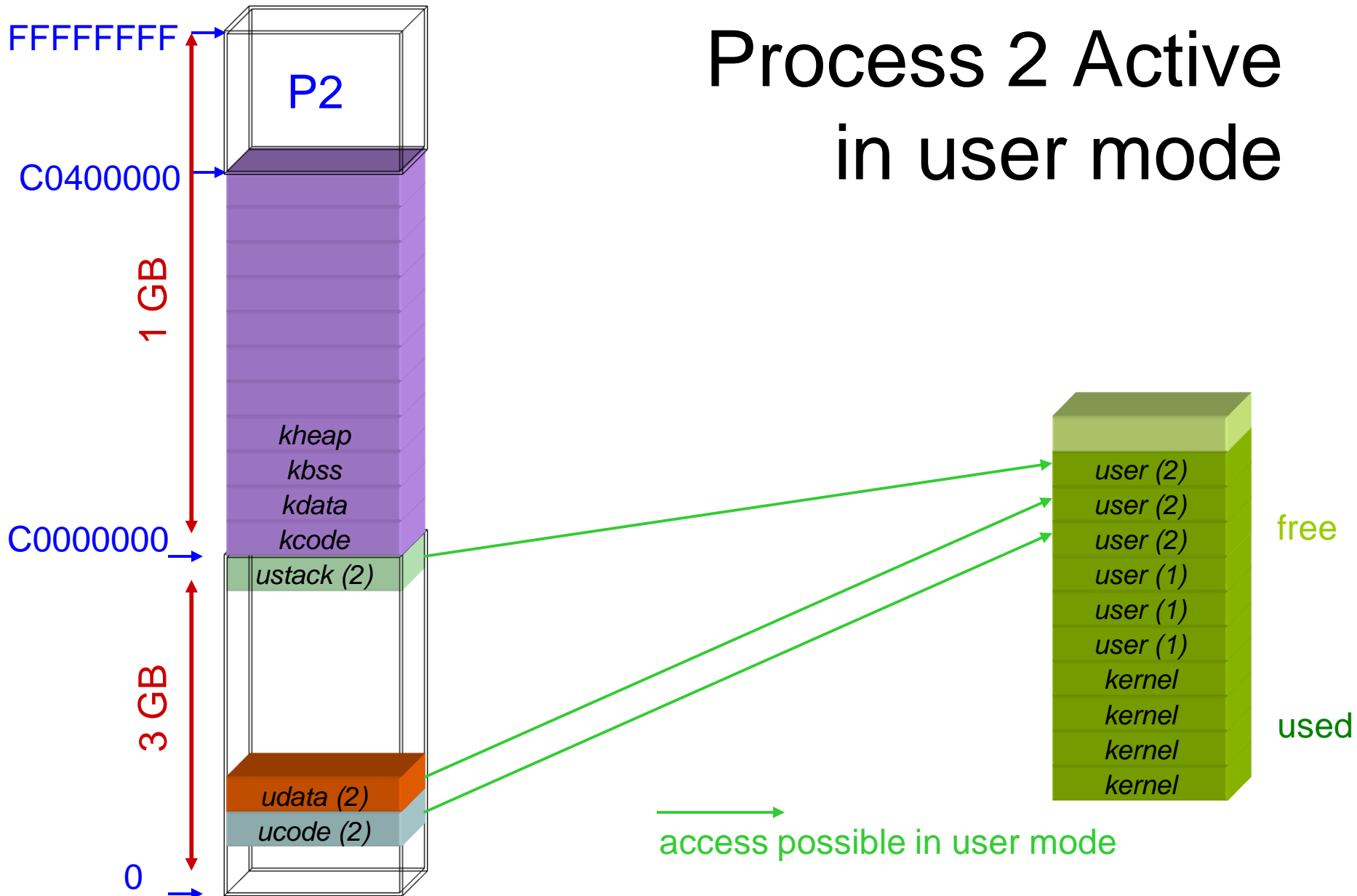
Process 1 Active in kernel mode



Process 2 Active in kernel mode



Process 2 Active in user mode





Meltdown Mitigation

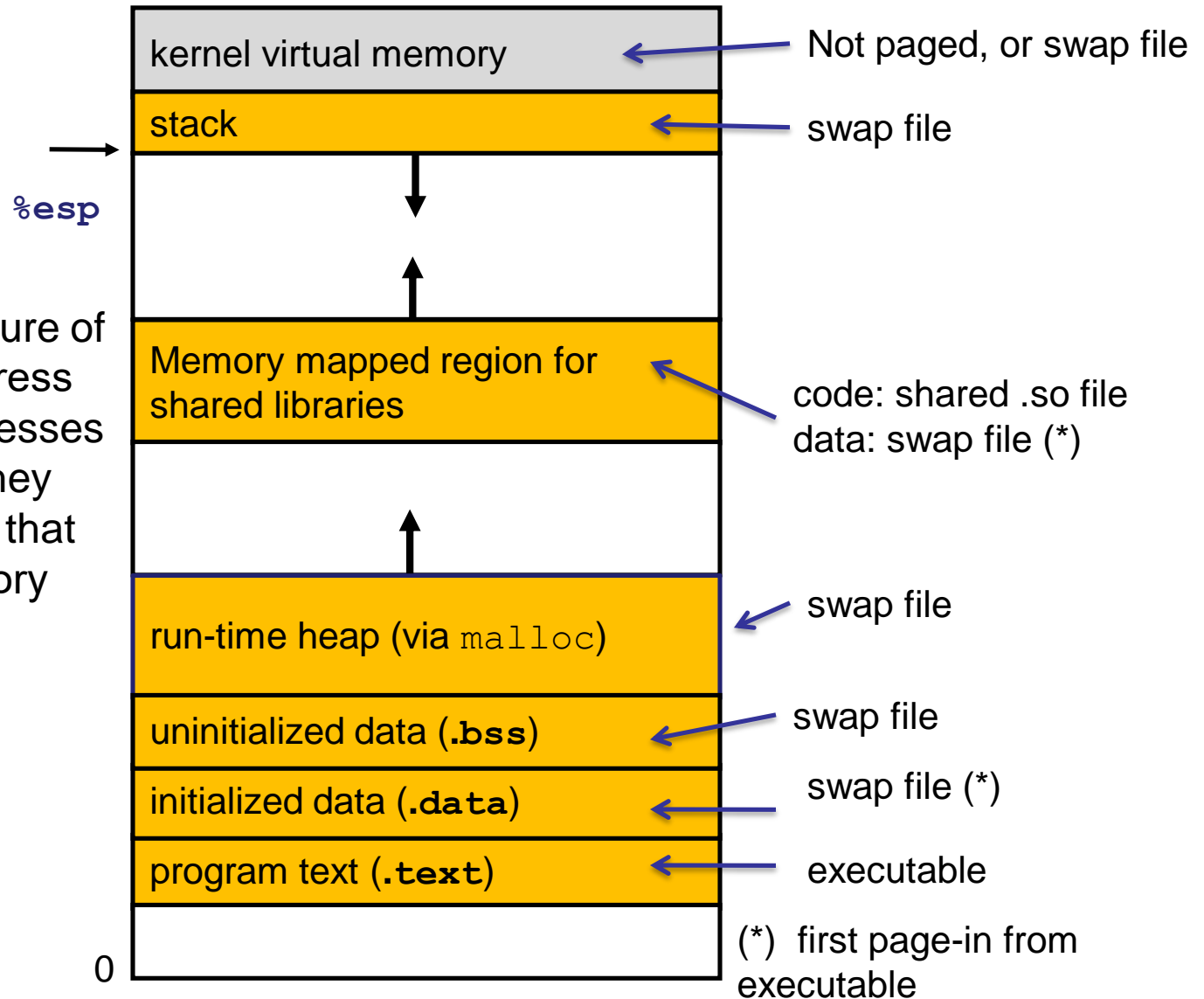
- Post Meltdown, kernel and user mode no longer use the same page table.
- Therefore, the (red) kernel mappings are no longer immediately accessible once the processor switches into kernel mode.
- Requires additional page table switch once the kernel is entered (expensive), otherwise, it's the same setup.

Paging to/from disk

- Idea: hold only those data in physical memory that are actually accessed by a process
- Maintain map **for each process**
 $\{ \text{virtual addresses} \} \rightarrow \{ \text{physical addresses} \} \cup \{ \text{disk addresses} \}$
- OS manages mapping, decides which virtual addresses map to physical (if allocated) and which to disk
- Disk addresses include:
 - Executable .text, initialized data
 - Swap space (typically lazily allocated)
 - Memory-mapped (mmap'd) files (see example)
- **Demand paging**: bring data in from disk lazily, on first access
 - Unbeknownst to application

Process Memory Image

Backed by



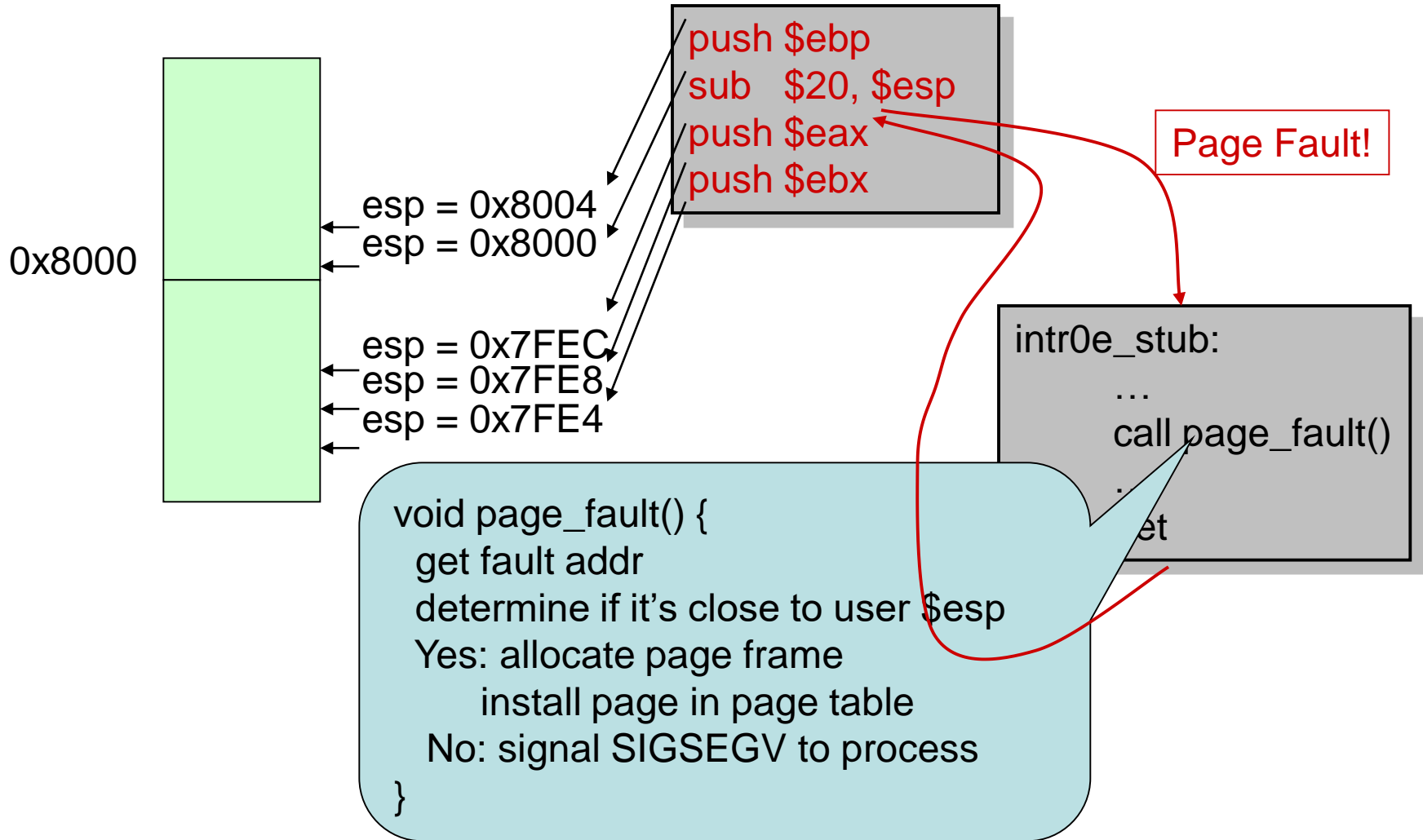
OS maintains structure of each process's address space – which addresses are valid, what do they refer to, even those that aren't in main memory currently

Try:
`cat /proc/self/maps`

Servicing Page Faults

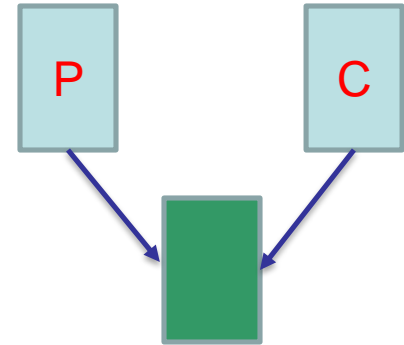
- When process accesses address that is not currently mapped, the hardware will signal a fault
 - If address is in kernel space, or refers to unmapped region
 - Send SIGSEGV to process
 - Else determine which region address is in
 - If heap, allocate new page (“minor fault”), or swap page from disk
 - If code segment, read code from executable
 - If first access to global variable, read data from disk; else swap from disk
 - If access to mmaped file, read data from file
 - Establish new v-p mapping in page table, and retry
- Note: there are no page faults for pages that are present in memory
 - There may be TLB misses, however – on x86, these are handled in hardware – can introduce hidden performance cost

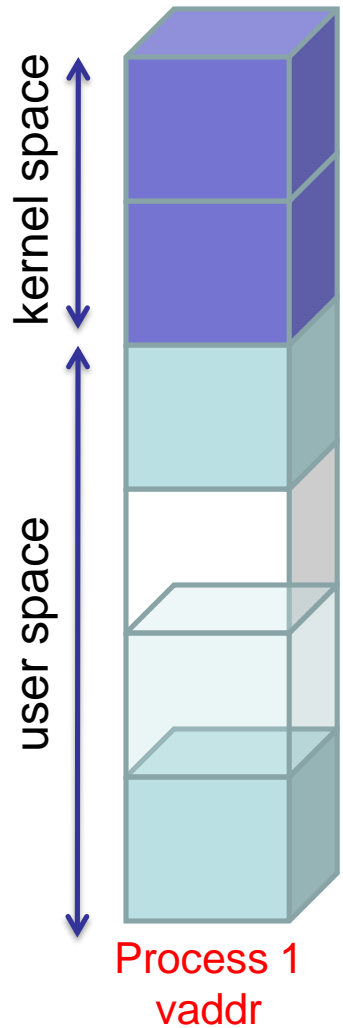
Microscopic View of Stack Growth



fork()/exec() revisited

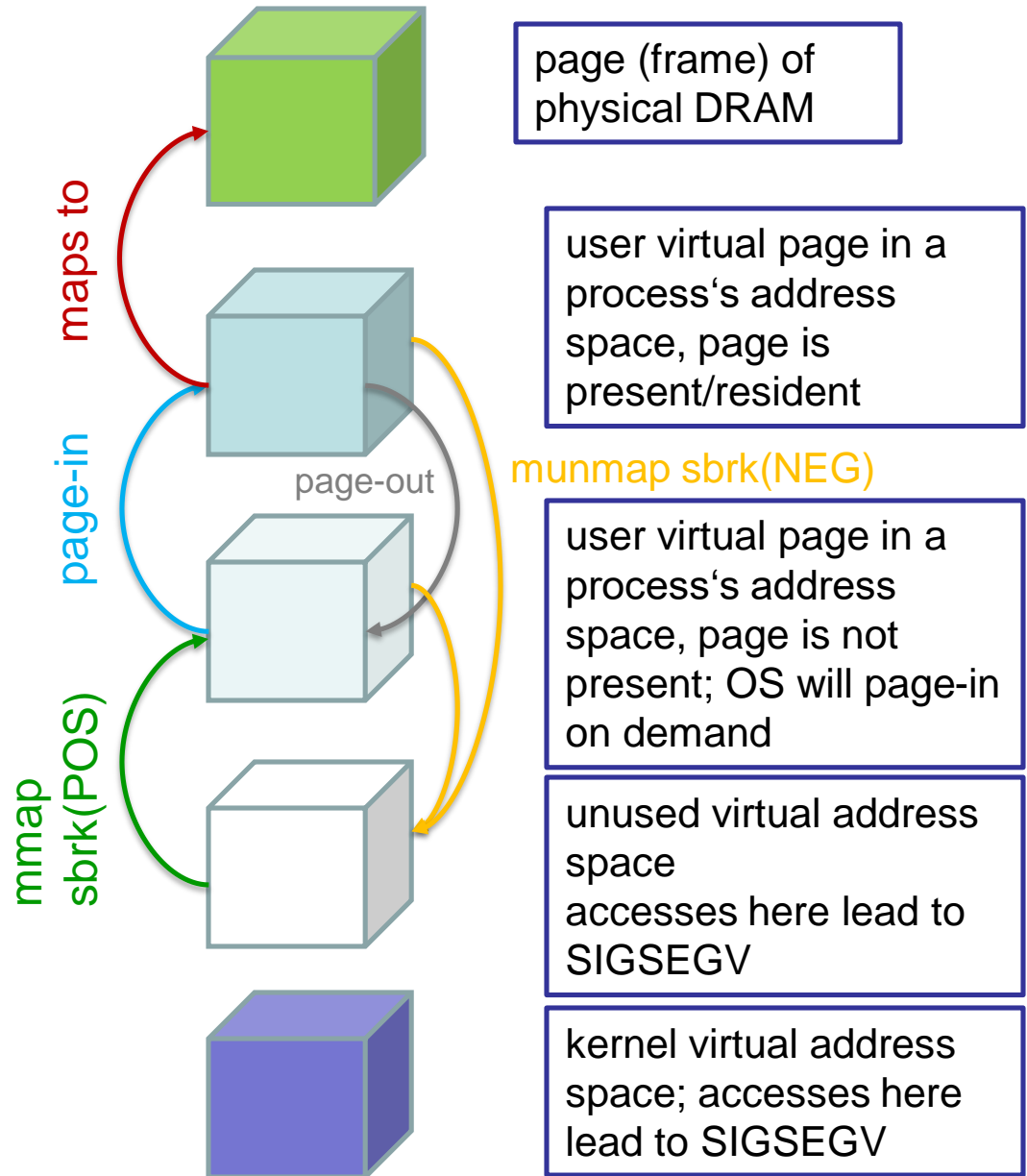
- fork():
 - Clone page table of parent
 - Set all entries read-only
 - Perform copy on write (if it happens while shared)
- exec():
 - Remove all existing page table entries
 - Unshares parent's entries
 - Start over as per instructions in executable
- Optimizes common case: child does an exec() shortly after fork()

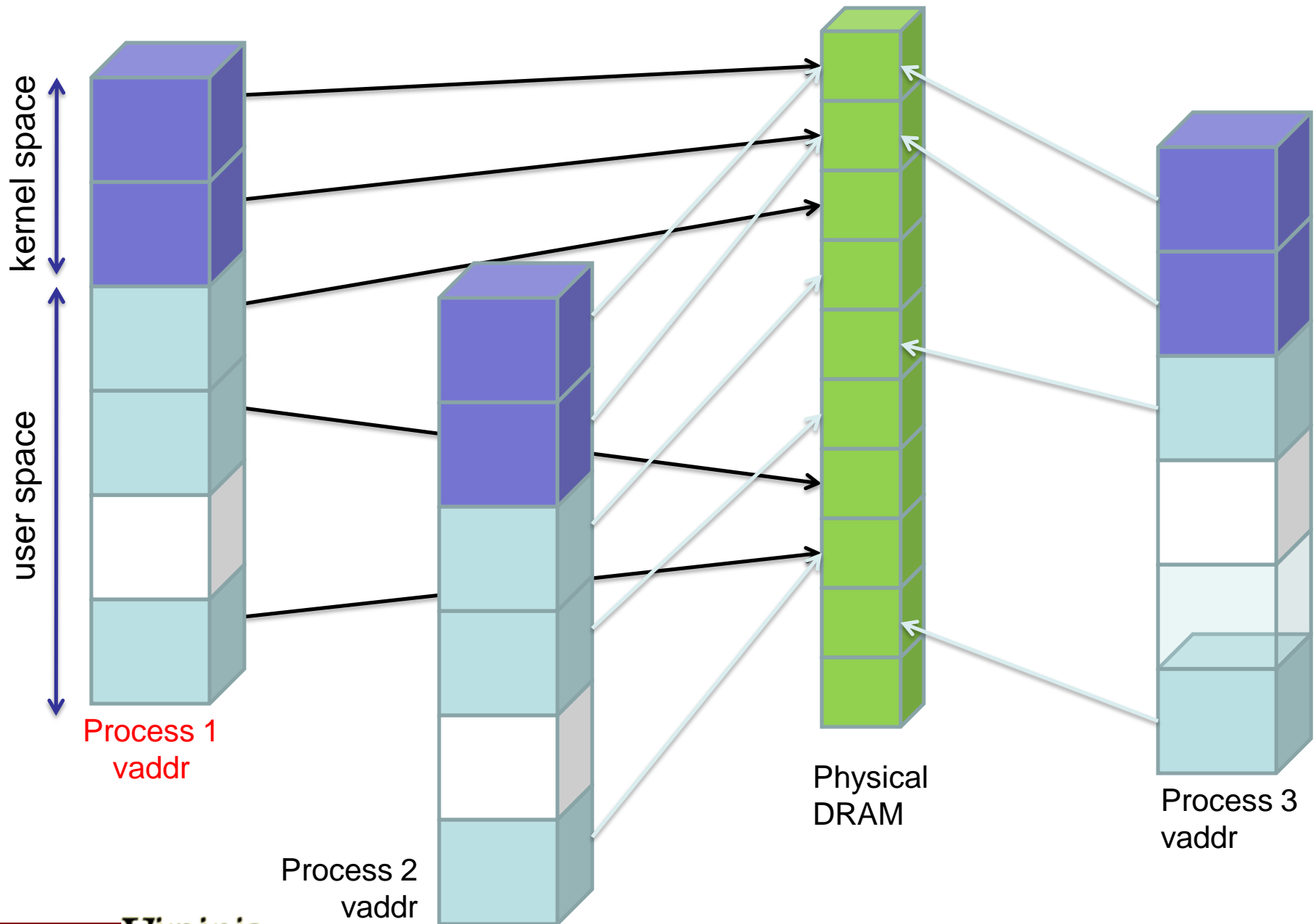




mapping in currently active page table (1 set per CPU for current process)

mapping in currently inactive page table (1 set per process)



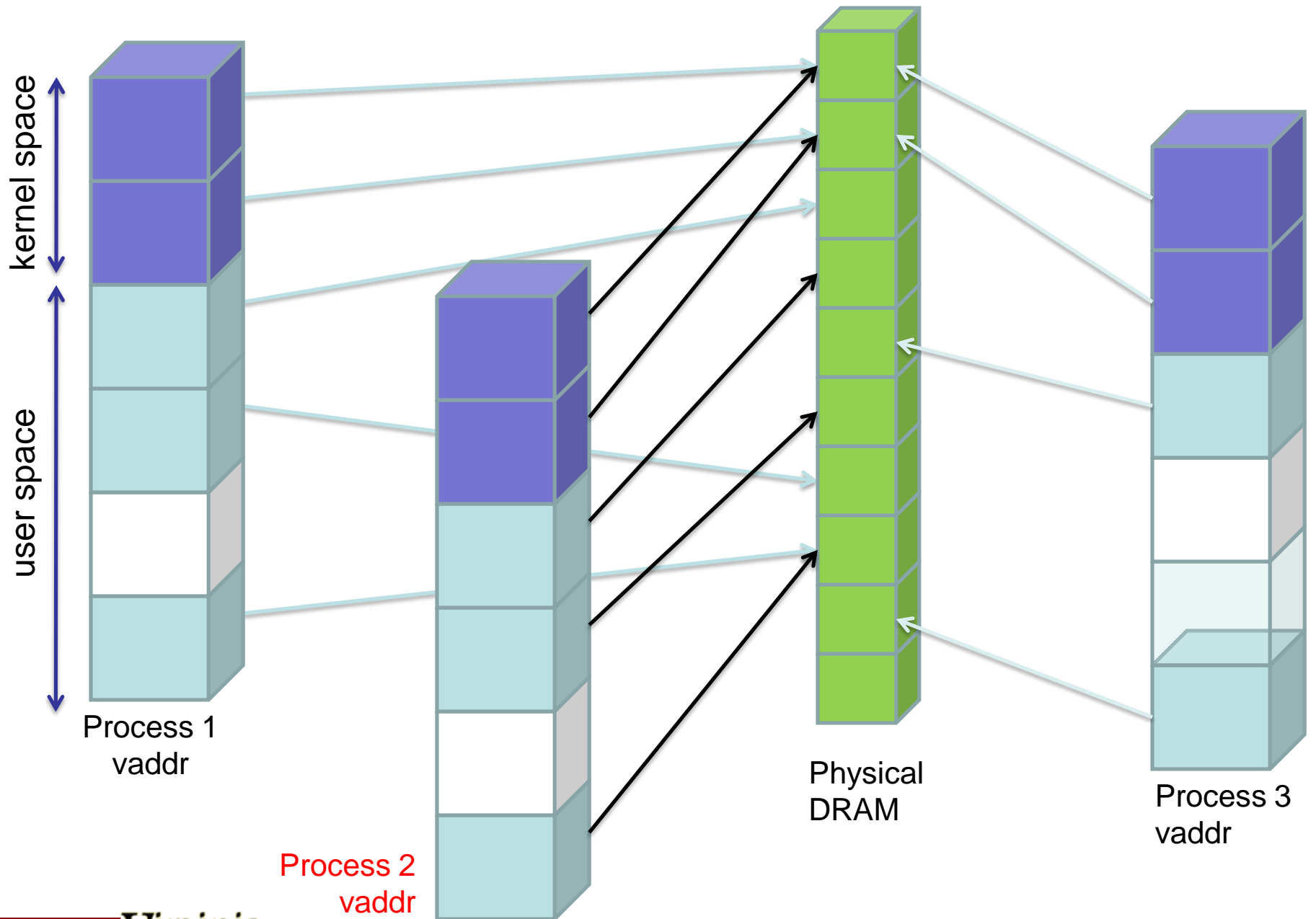


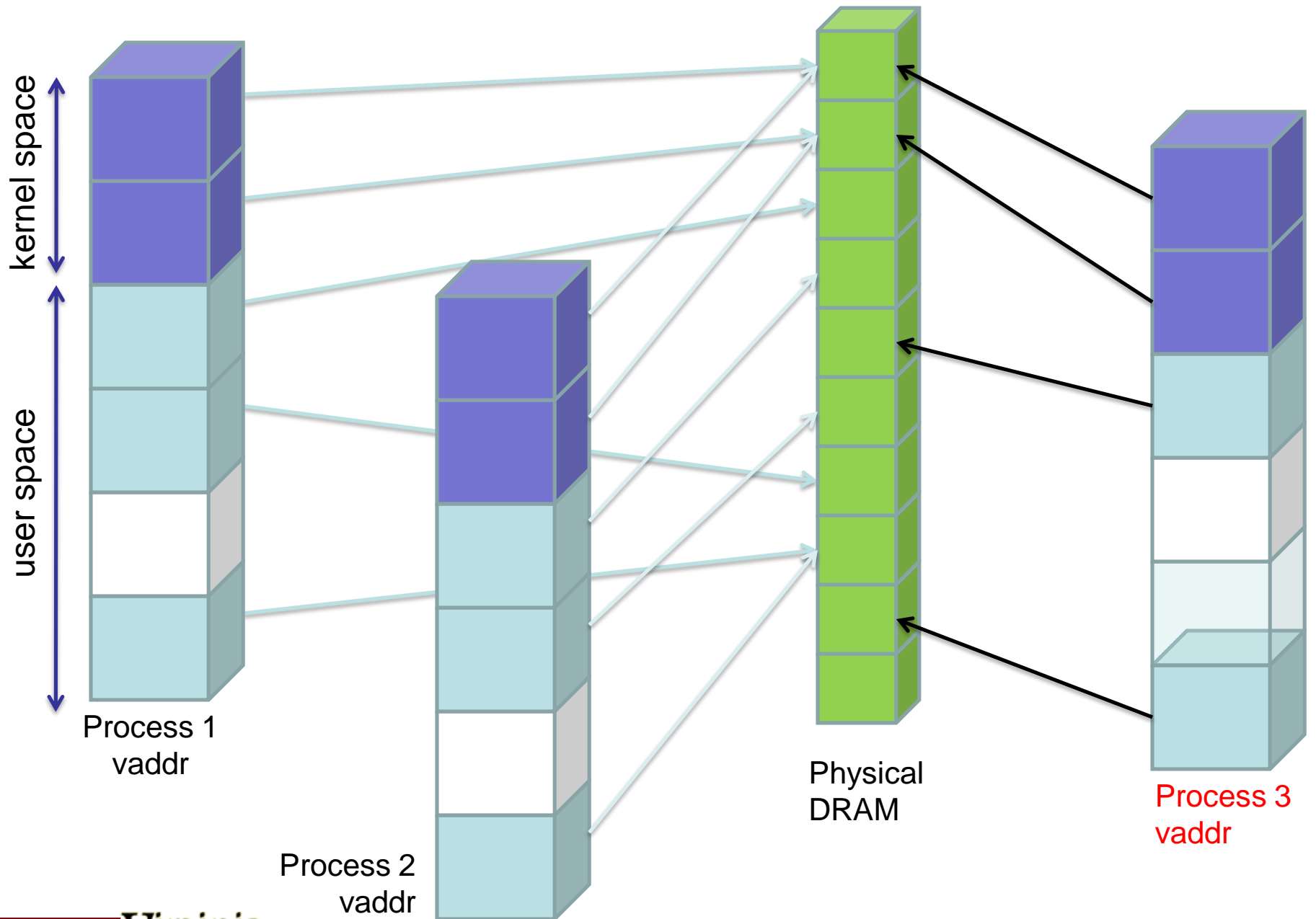
Process 1
vaddr

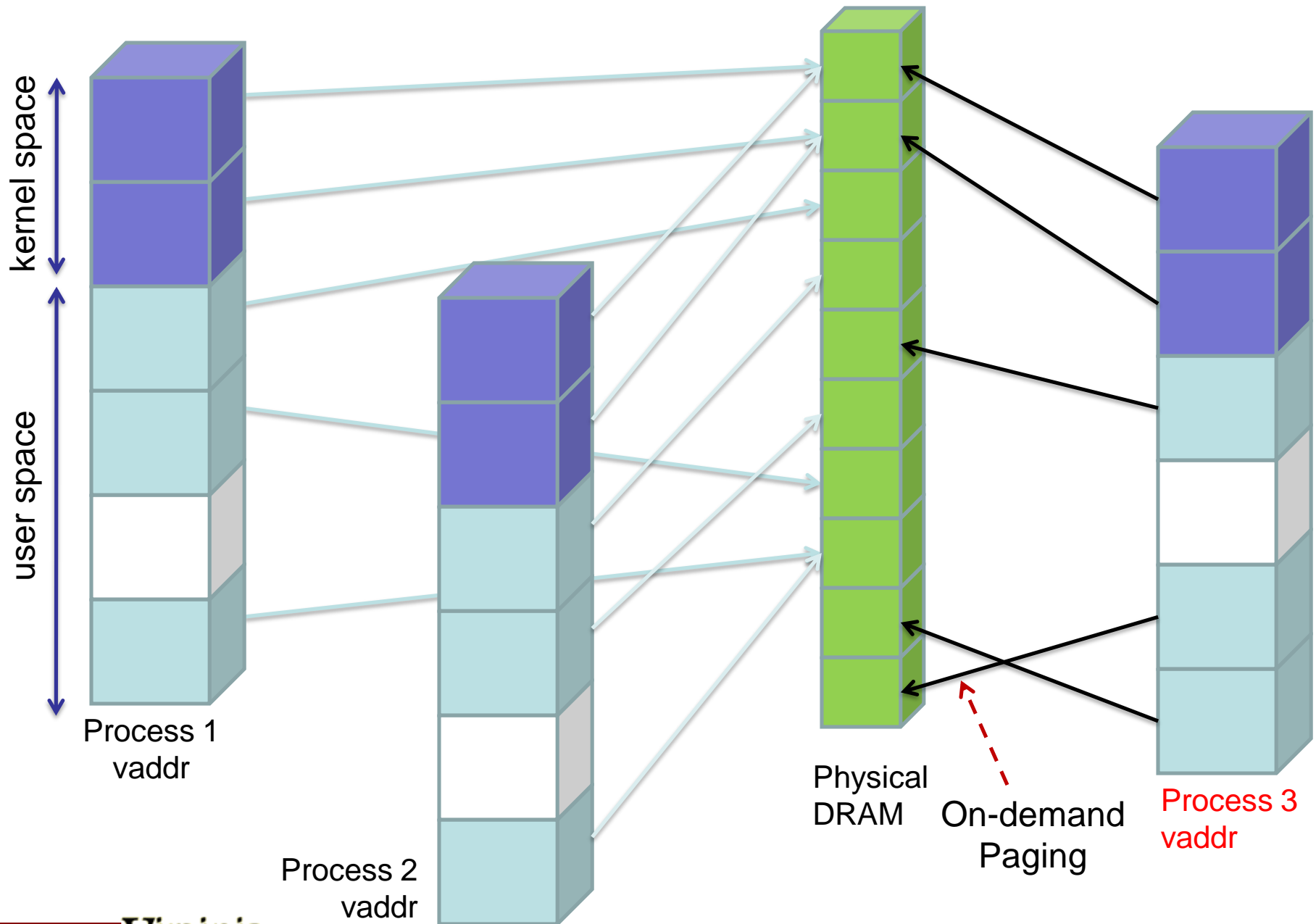
Process 2
vaddr

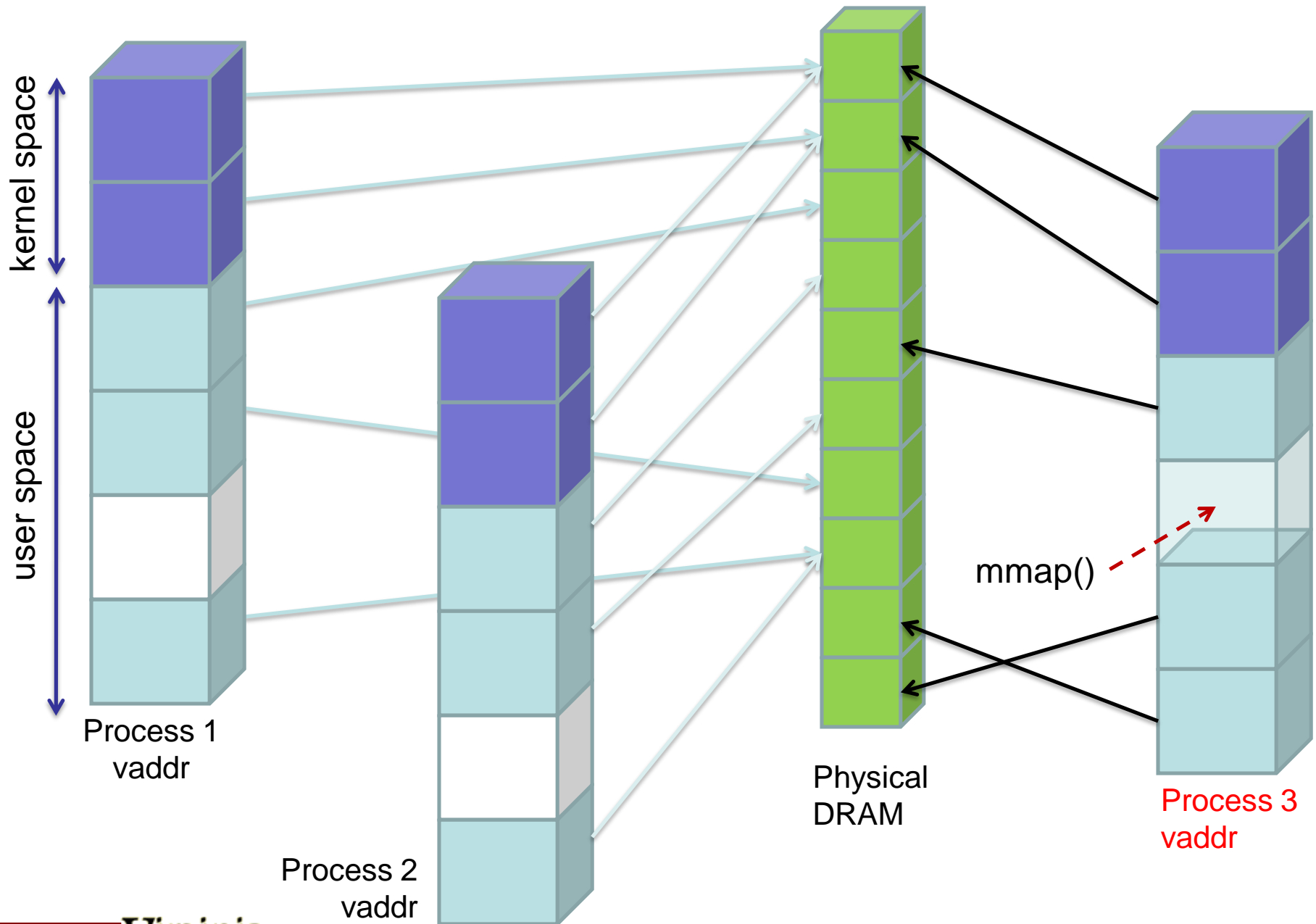
Physical
DRAM

Process 3
vaddr









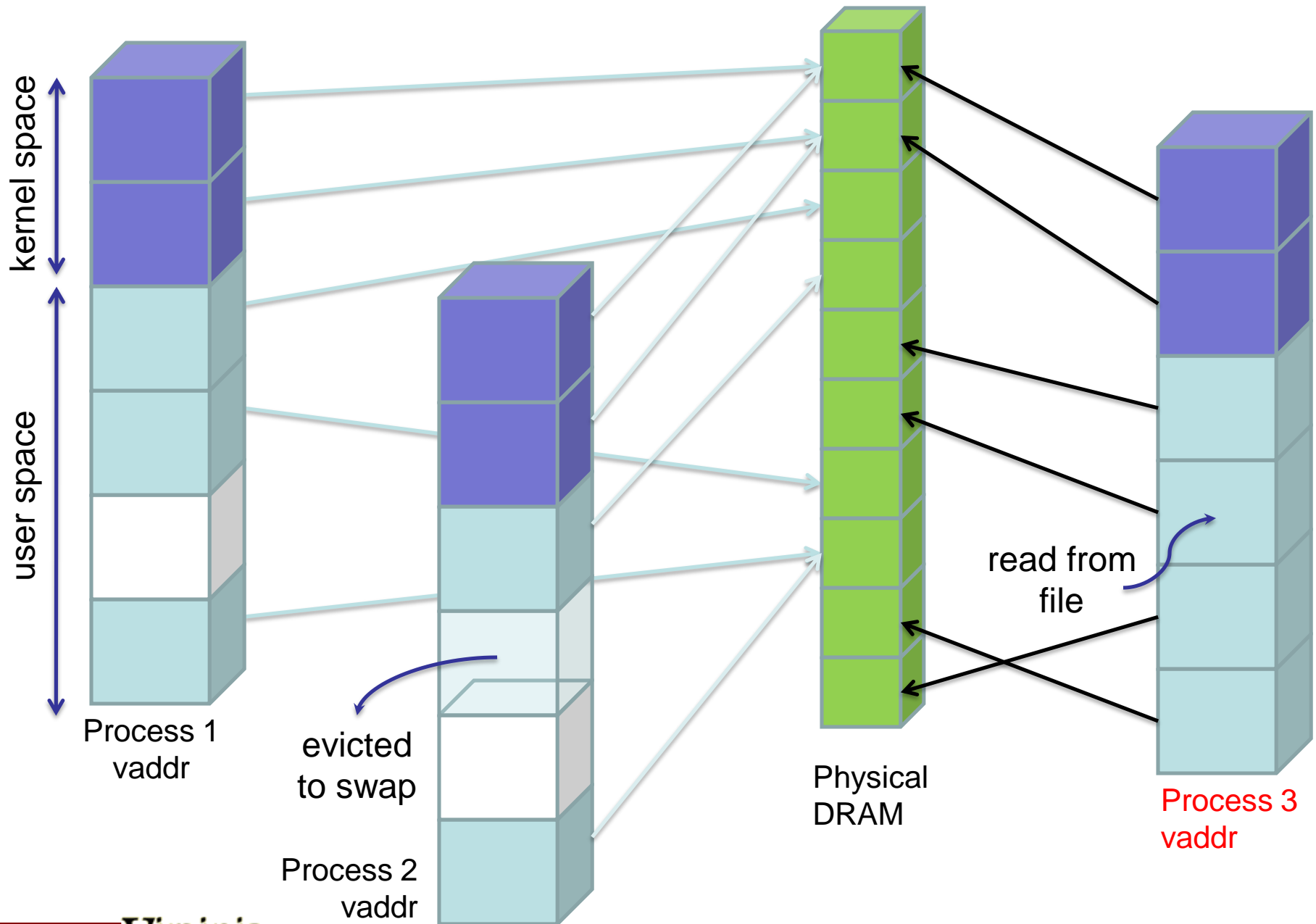
Process 1
vaddr

Process 2
vaddr

Physical
DRAM

Process 3
vaddr

mmap()



Managing Physical Memory

- OS must decide what to use physical memory for
 - Application Data
 - Mostly per process, except for shared memory areas
 - Heaps, stacks, BSS
 - File Data (Single copy per file)
 - Mmap'ed files, executables, shared libs
 - Chunks of files recently accessed via explicit I/O
- When demand is greater than supply, must rededicate physical memory by “evicting” pages to disk
 - Either done ahead of time with some hysteresis
 - Or last minute (“direct reclaim”)

Page Replacement Strategies

- Prediction game: optimal strategy is to replace (“evict”) the page whose data will be accessed **farthest in the future**
 - Of course, can’t know that → use heuristics
- Most heuristics are based on “past = future” idea and approximate LRU
 - While adding guards against scenarios in which LRU is known to fail, e.g. large looping accesses or single sequential reads
 - Must approximate because per-access maintenance of LRU lists is too expensive
- Must weigh file data vs. process data
- Must weigh other pages from same process vs. all processes
 - Local vs. global replacement policies

VM Access Time & Page Fault Rate

$$\text{access time} = p * \text{memory access time} + (1-p) * (\text{page fault service time} + \text{memory access time})$$

- Consider expected access time in terms of fraction p of page accesses that don't cause page faults.
- Then $1-p$ is page fault frequency
- Assume $p = 0.99$, assume memory is 100ns fast, and page fault servicing takes 10ms – how much slower is your VM system compared to physical memory?
- access time = 99ns + 0.01*(10000100) ns \approx 100,000ns or 0.1ms
 - Compare to 100ns or 0.0001ms speed \approx about 1000x slowdown
- Conclusion: even relatively low page fault rates lead to huge slowdown – must keep page fault rates *very low*

Thrashing

- VM works well if working set size (amount of memory accessed within an interesting time span) can be accommodated in physical memory
- If working set size grows too large, OS will continuously service page faults, and end up evicting pages accessed soon after
- Result: “**thrashing**”
 - Moving data to/from disk continually while not making progress on computation
 - Leads to *low* CPU utilization

Prefetching

- All modern VM systems use prefetching
 - Usual strategy: detect sequential accesses to file
 - even if done via virtual memory system & mmaped files
 - Sometimes application-guided
 - Linux readahead(2) system call
 - E.g. Windows Vista remembers which data an application touched (speeds up startup time)
- The performance of a VM system depends both on its page replacement and its prefetching strategies

Summary

- Virtual memory is a technique that combines
 - Address translation (Indirection)
 - Demand paging
 - Protection

to virtualize physical memory and protect applications and the kernel

- It is transparent to applications except for its possible performance impact