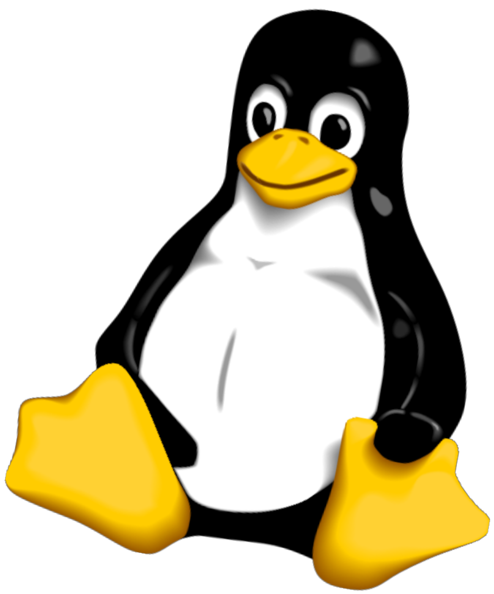
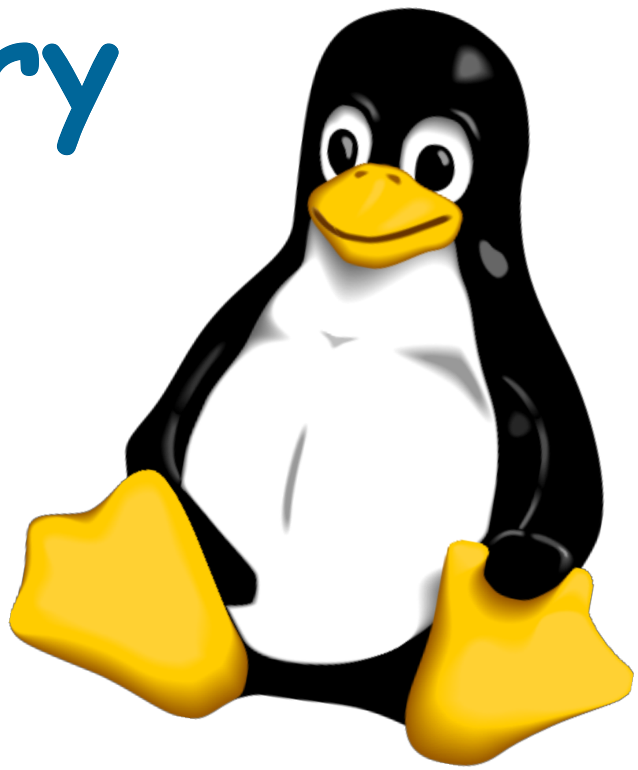
# Virtual Memory and Linux

Based on Alan Ott's slides on
Embedded Linux Conference
April 4-6, 2016



# Physical Memory



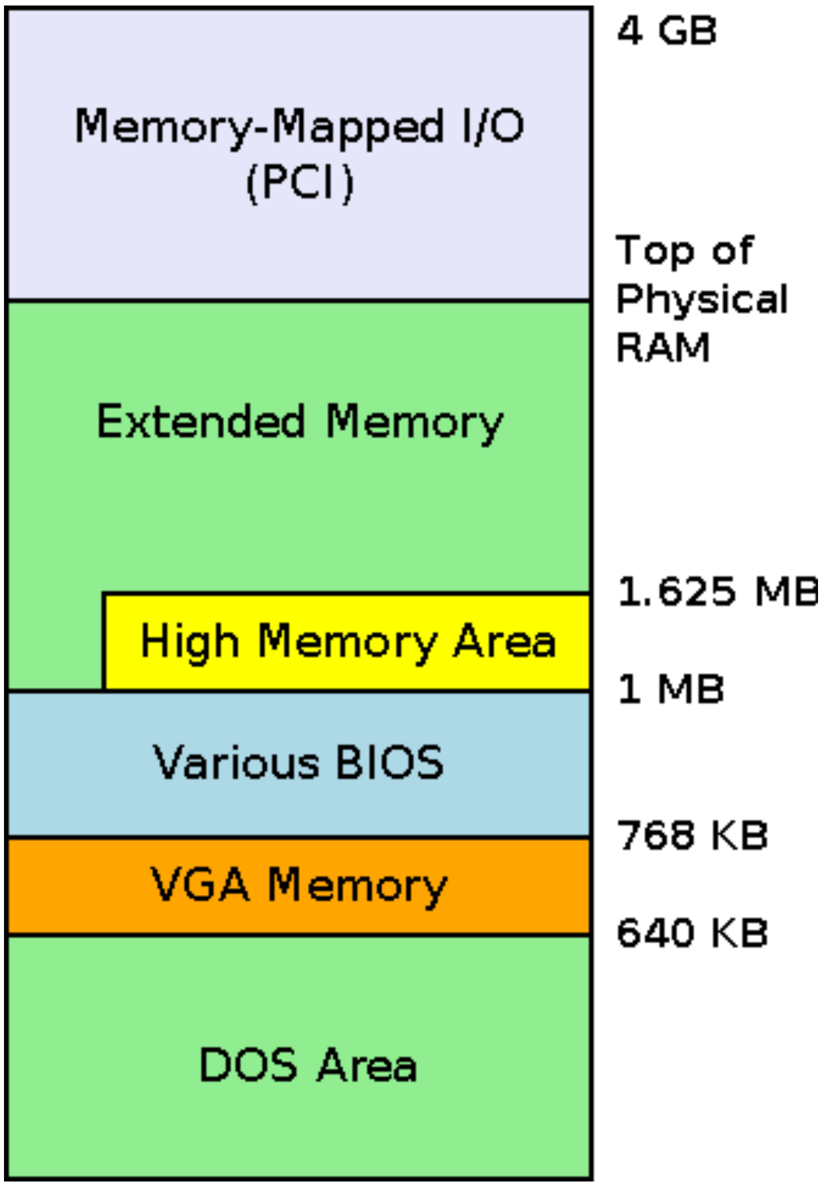
# Flat Memory

- Older and modern, but simple systems have a single address space
  - Memory and peripherals share
    - Memory will be mapped to one part
    - Peripherals will be mapped to another
  - All processes and OS share the same memory space
    - No memory protection!
    - User space can stomp kernel mem!

# Flat Memory

- CPUs with flat memory
  - 8086-80206
  - ARM Cortex-M
  - IoT chips - most 8- and 16-bit systems

# x86 Memory Map

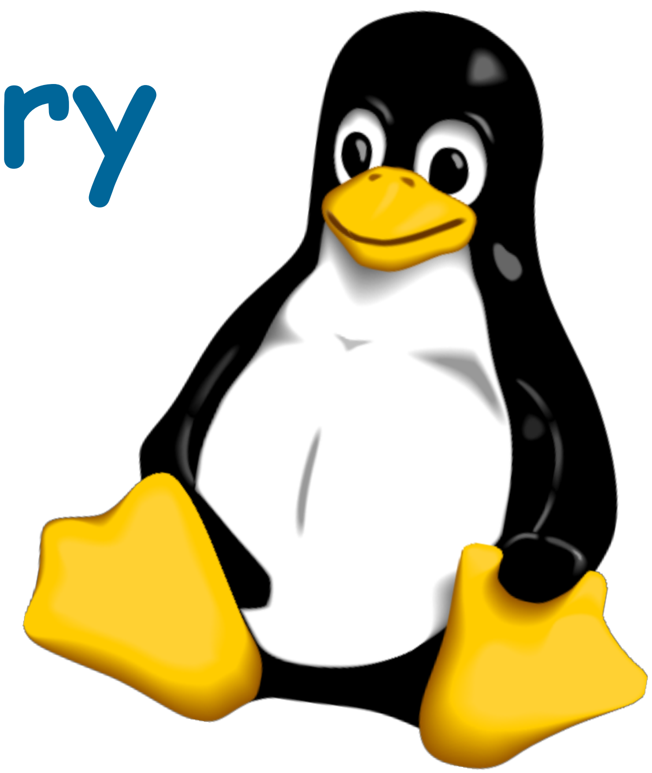



- Lots of Legacy
- RAM is split (DOS Area and Extended)
- Hardware mapped between RAM areas.
- High and Extended accessed differently

# Limitations

- Portable C programs expect flat memory
  - Accessing memory by segments limits portability
- Management is tricky
  - Need to keep processes separated
- No protection
  - Rogue programs can corrupt the  entire system

# Virtual Memory



# What is Virtual Memory?

- Virtual Memory is an address mapping
  - Maps virtual address space to physical address space
    - Maps virtual addresses to physical RAM
    - Maps virtual addresses to hardware devices
      - PCI devices
      - GPU RAM
      - On-SoC IP blocks

# What is Virtual Memory?

- Advantages
  - Each processes can have a different memory mapping
    - One process's RAM is inaccessible (and invisible) to other processes.
      - Built-in memory protection
    - Kernel RAM is invisible to userspace processes
  - Memory can be moved
  - Memory can be swapped to disk

# What is Virtual Memory?

- Advantages (cont)
  - Hardware device memory can be mapped into a process's address space
    - Requires kernel perform the mapping
  - Physical RAM can be mapped into multiple processes at once
    - Shared memory, shared libraries
  - Memory regions can have access permissions
    - Read, write, execute

# Virtual Memory Details

- Two address spaces
  - Physical addresses
    - Addresses as used by the hardware (except CPU)
      - DMA, peripherals
  - Virtual addresses
    - Addresses as used by software (CPU generated)
      - Load/Store instructions (RISC)
      - Any instruction accessing RAM (CISC)

# Virtual Memory Details

- Mapping is performed in hardware
  - No performance penalty for accessing already-mapped RAM regions
  - Permissions are handled without penalty
  - The same CPU instructions are used for accessing RAM and mapped hardware
  - Software, during its normal operation, will only use virtual addresses.
    - Includes kernel and user space

# Memory-Management Unit

- The memory-management unit (MMU) is the hardware responsible for implementing virtual memory.
  - Sits between the CPU core and memory
  - Most often part of the physical CPU itself.
    - On ARM, it's part of the licensed core.
  - Separate from the RAM controller
    - DDR controller is a separate IP block

# Memory-Management Unit

- MMU (cont)
  - Transparently handles all memory accesses from Load/Store instructions
    - Maps accesses using virtual addresses to system RAM
    - Maps accesses using virtual addresses to memory-mapped peripheral hardware
    - Handles permissions
    - Generates an exception (page fault) on an invalid access
      - Unmapped address or insufficient permissions

# Translation Lookaside Buffer

- The TLB stores the mappings from virtual to physical address space in hardware
  - Also holds permission bits
- TLB is a part of the MMU
- TLB is consulted by the MMU when the CPU accesses a virtual address

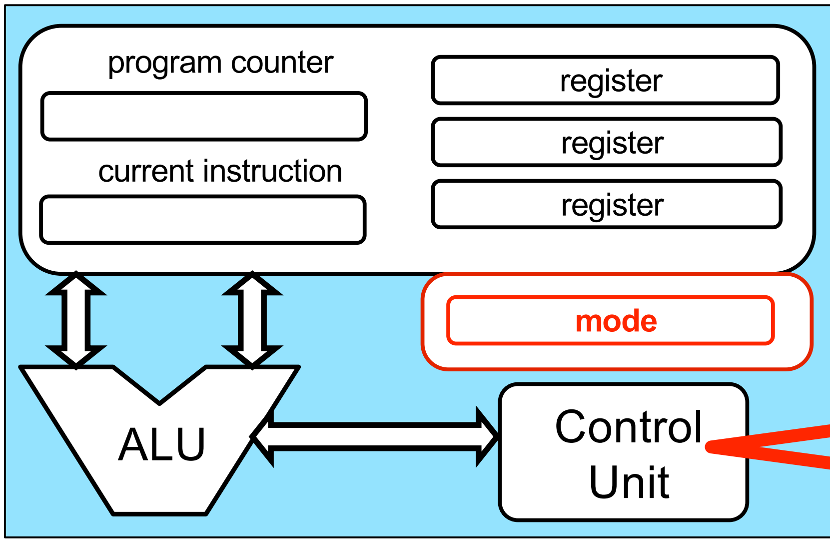
# Page Faults

- A page fault is a CPU exception, generated when software attempts to use an invalid virtual address. There are three cases:
  - The virtual address is not mapped for the process requesting it.
  - The processes has insufficient permissions for the address requested.
  - The virtual address is valid, but swapped out
    - This is a software condition

# Lazy Allocation

- The kernel uses lazy allocation of physical memory.
  - When memory is requested by userspace, physical memory is not allocated until it's touched.
  - This is an optimization, knowing that many userspace programs allocate more RAM than they ever touch.
    - Buffers, etc.

# Kernel address space vs user address space




- Decode instruction
- Determine if instruction is privileged or not
  - Based on the instruction code (e.g., the binary code for all privileged instructions could start with '00')
- If instruction is privileged and mode == user, then abort!
  - Raise a "trap"

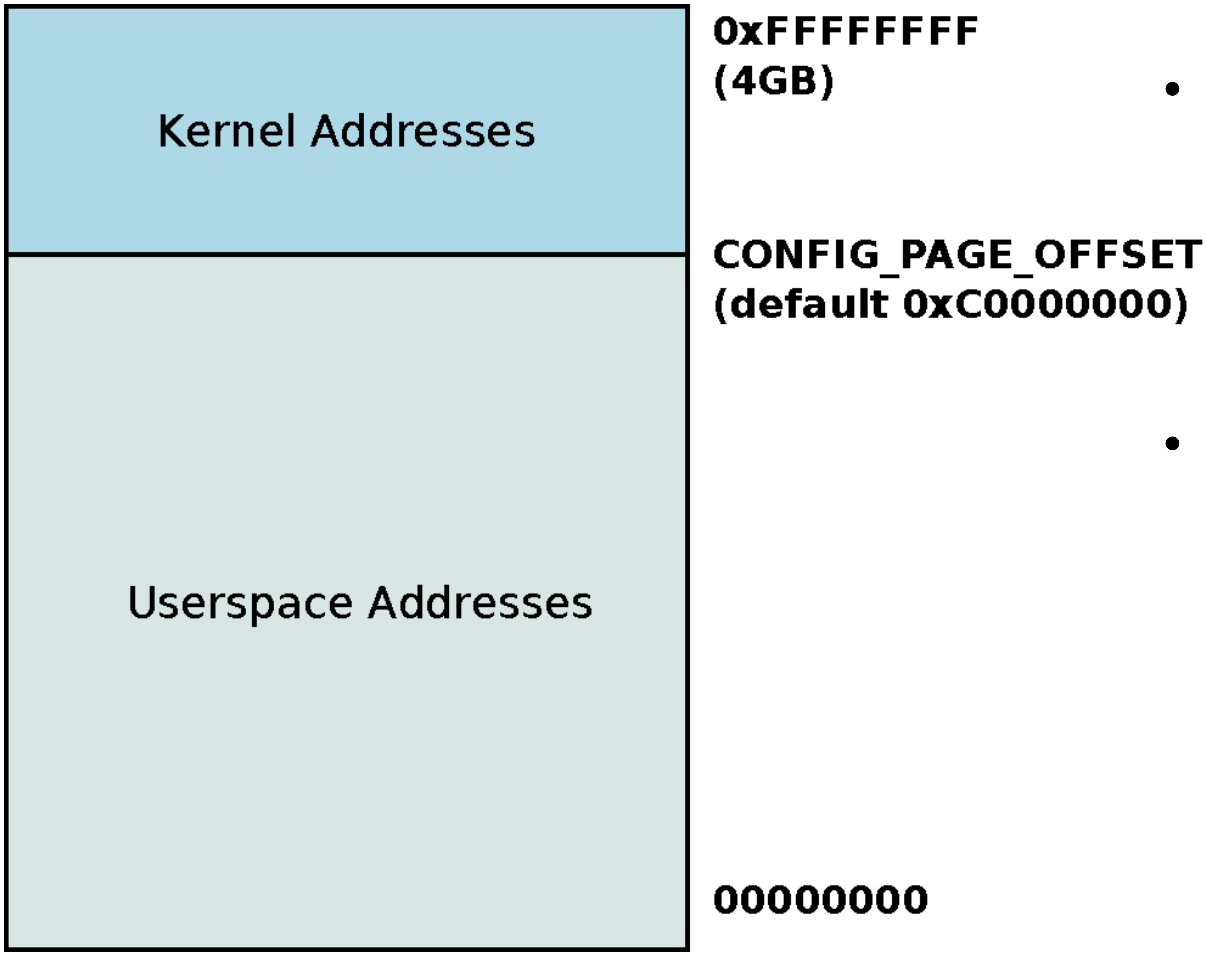
# Kernel address space vs user address space

- Address space can be privileged or non-privileged too
- For address space (AS) isolation, dividing AS into two parts
  - Kernel AS (privileged) and user AS (non-privileged)
  - Kernel code can access both AS
  - User code can only access user AS, cannot access kernel AS

# Virtual Addresses

- In Linux, the kernel uses virtual addresses, as userspace processes do.
- Virtual address space is split.
  - The upper part is used for the kernel
  - The lower part is used for userspace
- On 32-bit, the split is at
  0xC0000000

# Virtual Addresses - Linux




- By default, the kernel uses the top 1GB of virtual address space.
- Each userspace processes get the lower 3GB of virtual address space.

# Virtual Addresses – Linux

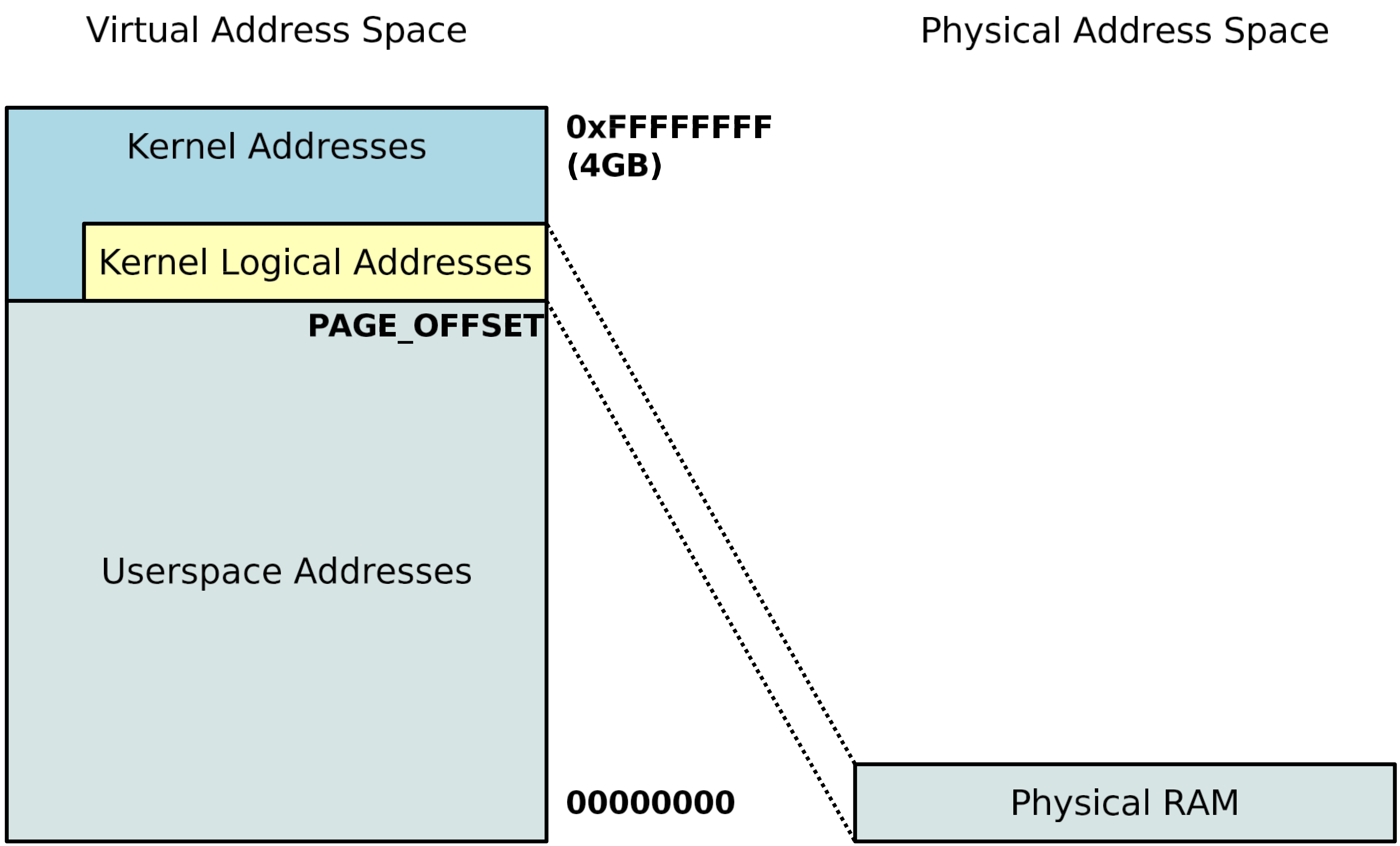
- Kernel address space is the area above CONFIG_PAGE_OFFSET.
  - For 32-bit, this is configurable at kernel build time.
    - The kernel can be given a different amount of address space as desired
      - See CONFIG_VMSPLIT_1G, CONFIG_VMSPLIT_2G, etc.
  - For 64-bit, the split varies by architecture, but it's high enough
    - 0x8000000000000000– ARM
    - 0xffff880000000000– x86_64

# Virtual address and physical RAM mapping

- How kernel manage physical RAM?
  - Such as allocate a physical frame to user process

- Must map physical RAM to kernel AS
  - Must have an address in kernel AS
  - For 32-bit, kernel address space (AS) is 1GB
    - No problem to handle small RAM (<=896MB)

# Kernel Logical Addresses

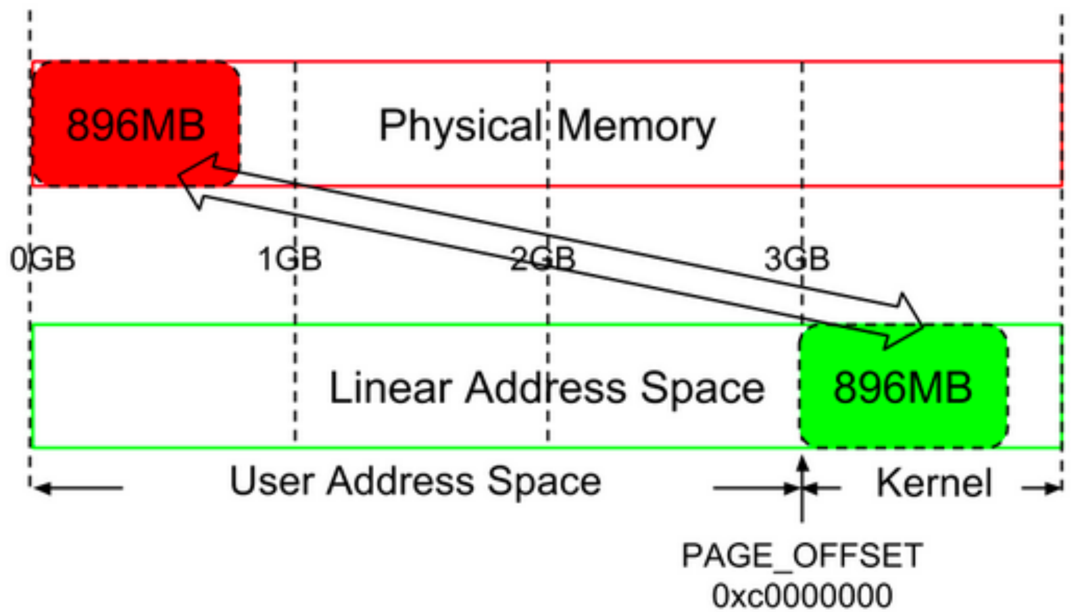


# Kernel Logical Addresses

- Kernel Logical Addresses
  - Addresses above PAGE_OFFSET
  - First 896MB of the kernel AS
  - Virtual addresses are a linear offset from their physical addresses.
    - Eg: Virt: 0xc0000000 → Phys: 0x00000000
  - This makes converting between physical and virtual addresses easy

# How to do VA to PA translation?

- Given a virtual address *v*, what is the physical address?
- Given a physical address *p*, what is the virtual address?



# Kernel Logical Addresses

- Kernel logical address space includes:
  - Memory allocated with kmalloc() and most other allocation methods
  - Kernel stacks (per process)
- Kernel logical memory can never be swapped out!

# Kernel Logical Addresses

- Kernel Logical Addresses use a linear mapping between physical and virtual address space.
- This means virtually-contiguous regions are by nature also physically contiguous
  - This, combined with the inability to be swapped out, makes them suitable for DMA transfers.

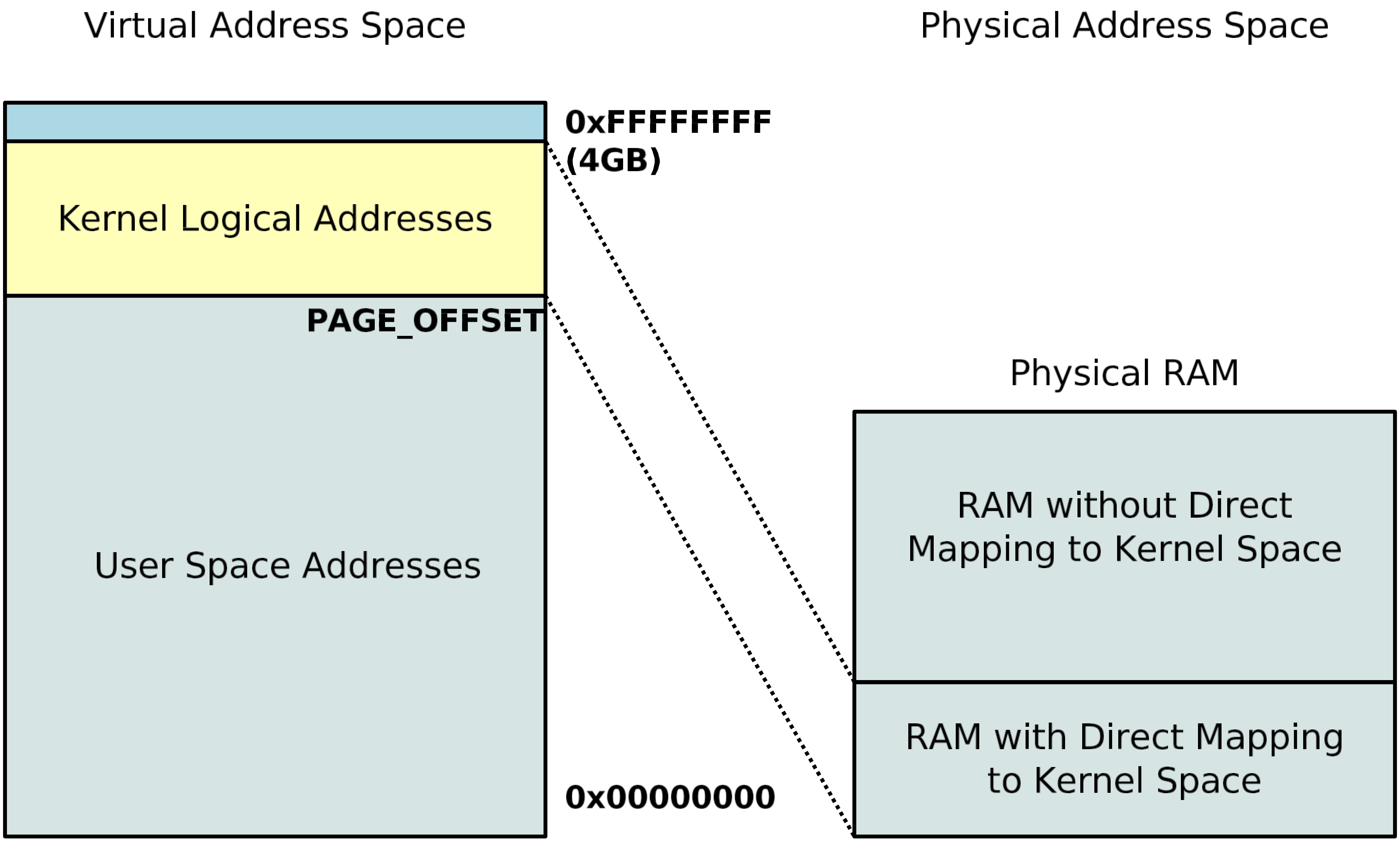
# Large physical RAM

- How kernel manage physical RAM?
  - Such as allocate a physical frame to user process
- Must map physical RAM to kernel AS
  - For 32-bit, kernel address space (AS) is 1GB
    - No problem to handle small RAM (<=896MB)
    - How about large physical RAM >= 1GB
      - Even larger than the kernel AS

# Large physical RAM

- For large memory systems (more than ~1GB RAM), not all of the physical RAM can be linearly mapped into the kernel's address space.
  - Kernel address space is the top 1GB of virtual address space, by default.
  - Further, 128 MB is reserved at the top of the kernel's memory space for non-contiguous allocations
    - See vmalloc() described later

# Kernel Logical Addresses (Large Mem)



# Large physical RAM

- As a result, only the bottom part of physical RAM has a kernel logical address
  - Note that on 64-bit systems, kernel AS is much larger than physical RAM, this case never happens
    - 39-bit kernel AS is 512GB
    - 48-bit kernel AS is 256TB
- Thus, in a large memory situation, only the bottom part of physical RAM is mapped linearly into kernel logical address space
  - Top 128MB guarantees that there is always enough kernel address space to accommodate all the RAM

# Kernel Virtual Addresses

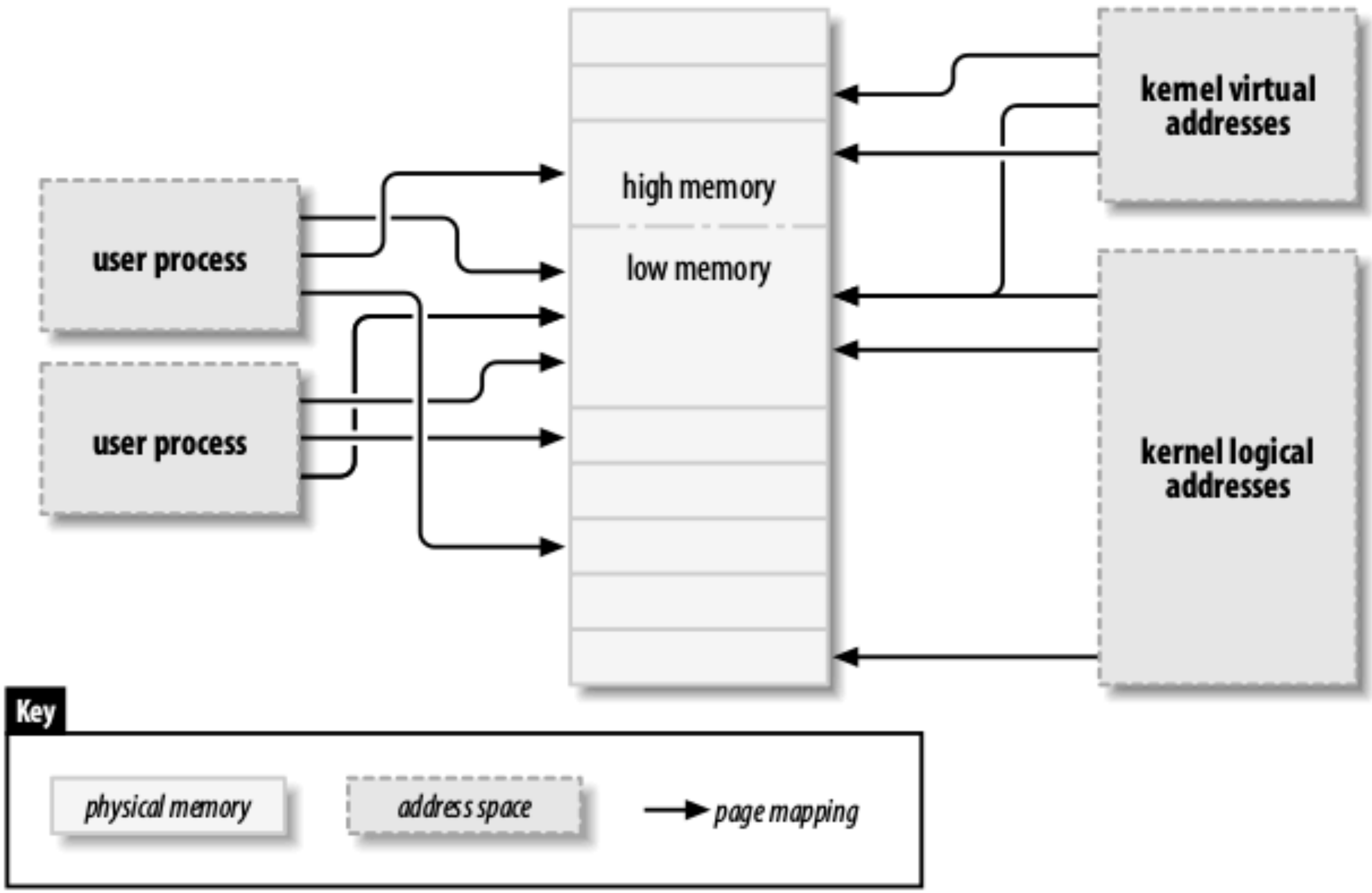
- Kernel Virtual Addresses are addresses in the region above the kernel logical address mapping.
- Kernel Virtual Addresses are used for non-contiguous memory mappings
  - Often for large buffers which could potentially be unable to get physically contiguous regions allocated.
- Also referred to as the vmalloc() area

# Virtual Addresses - Summary

- There are three kinds of virtual addresses in Linux.
  - The terminology varies, even in the kernel source, but the definitions in *Linux Device Drivers, 3rd Edition,* chapter 15, are somewhat standard.
  - LDD 3 can be downloaded for free at:
    https://lwn.net/Kernel/LDD3/

# Virtual Addresses - Linux

- User virtual address
- Kernel virtual address
- Kernel logical address



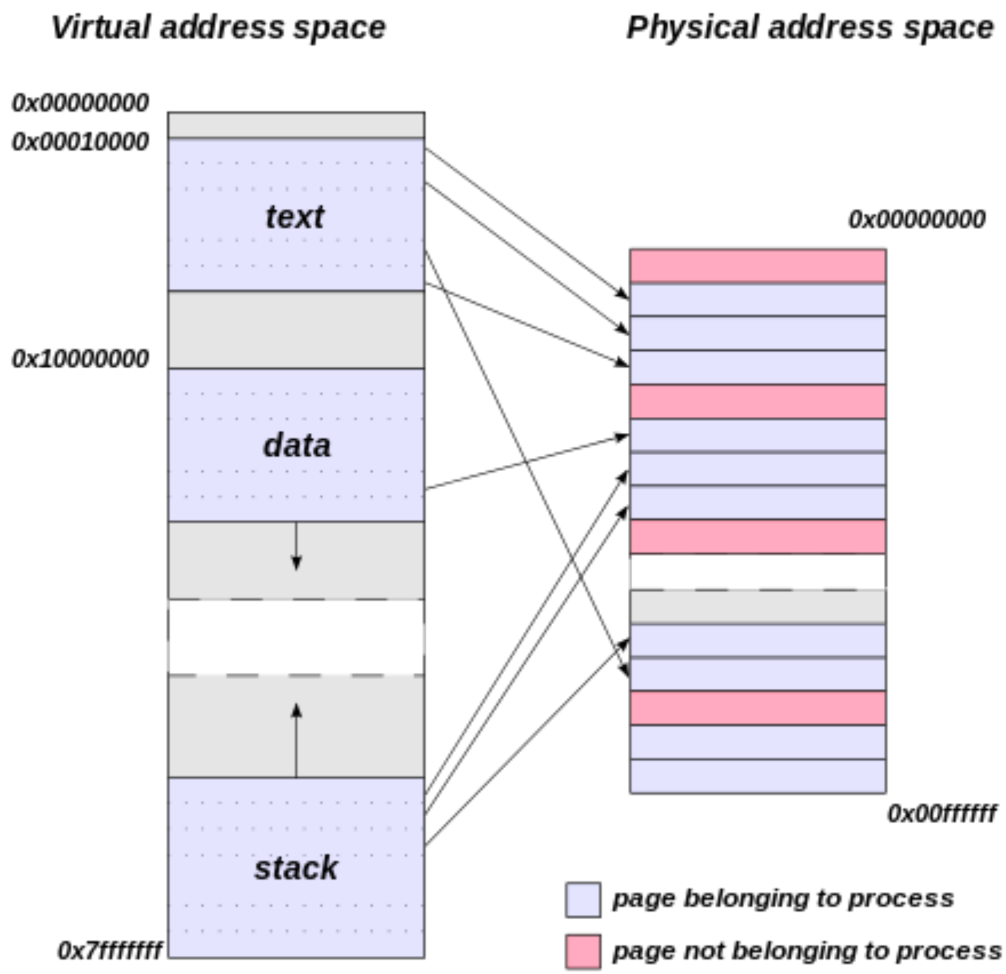
# Virtual Addresses - Linux

- User virtual address, Kernel virtual address, Kernel logical address
  - All belong to virtual address, but has different mappings to physical memory
  - Who has contiguous mapping?
  - Who has non-contiguous mapping?

- All these mapping can be done by MMU
  - By walking page table
  - Review: what is page table?

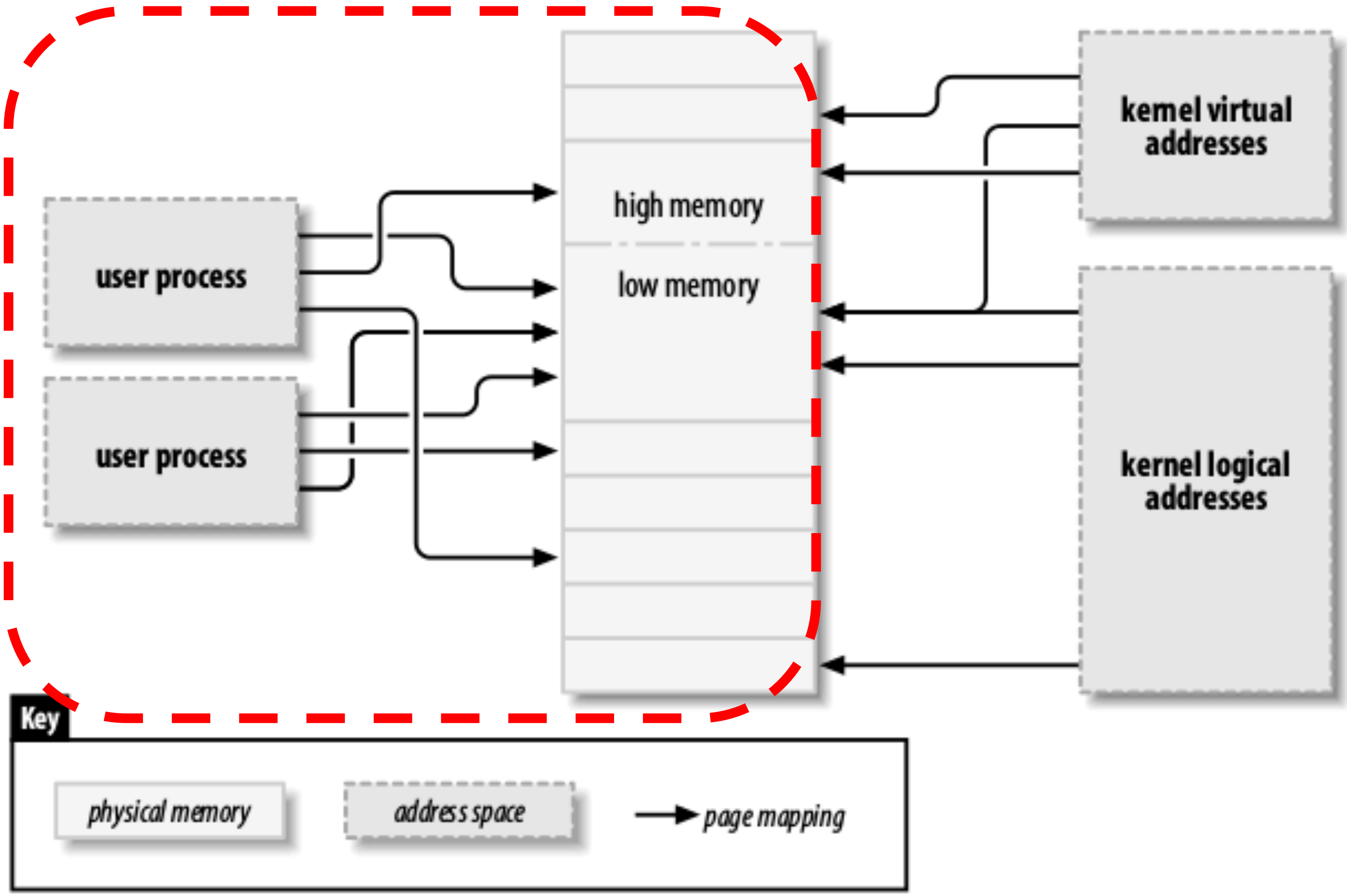
# User space page table mapping

- Non-contiguous



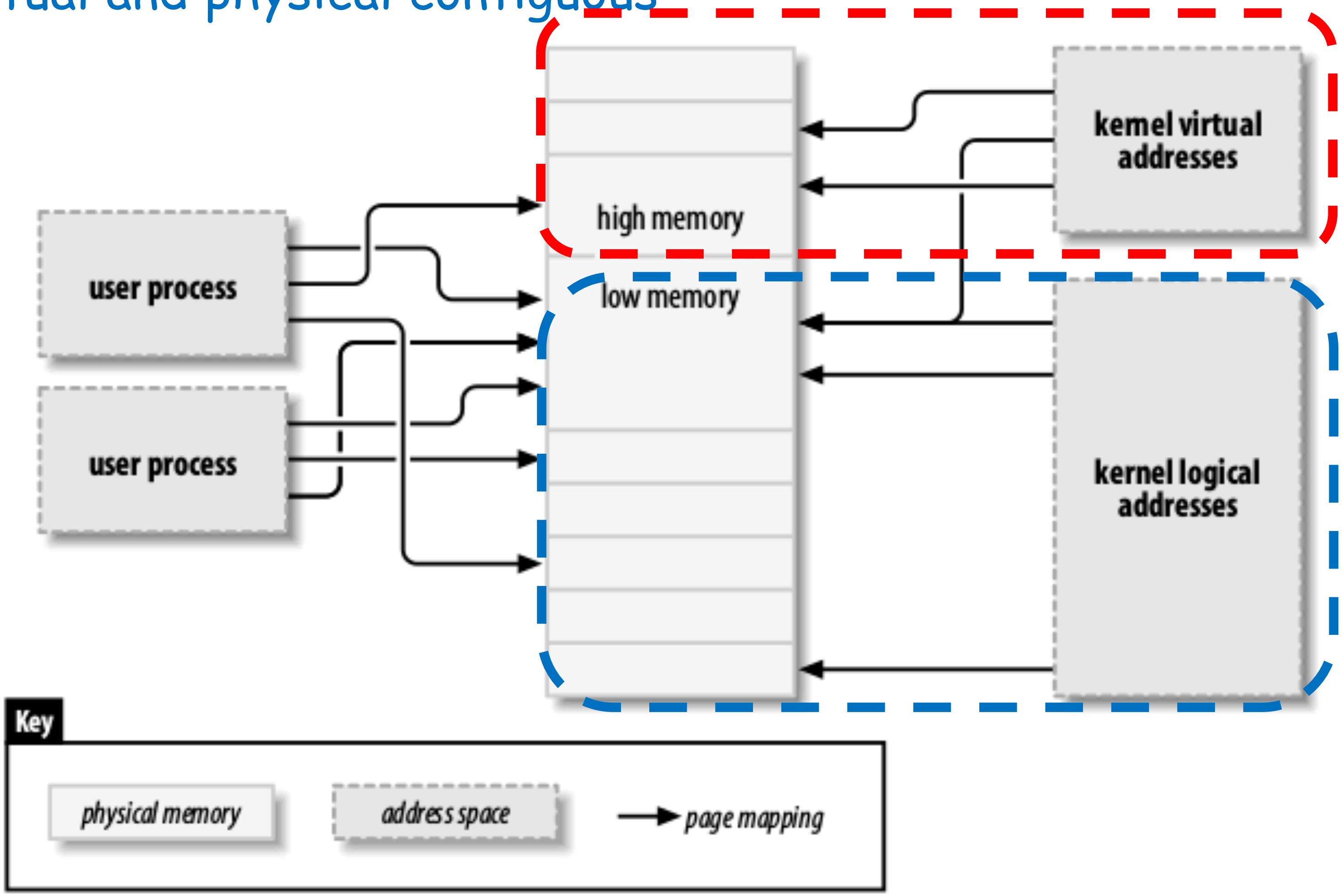
# User space page table mapping

- Virtual contiguous, physical non-contiguous

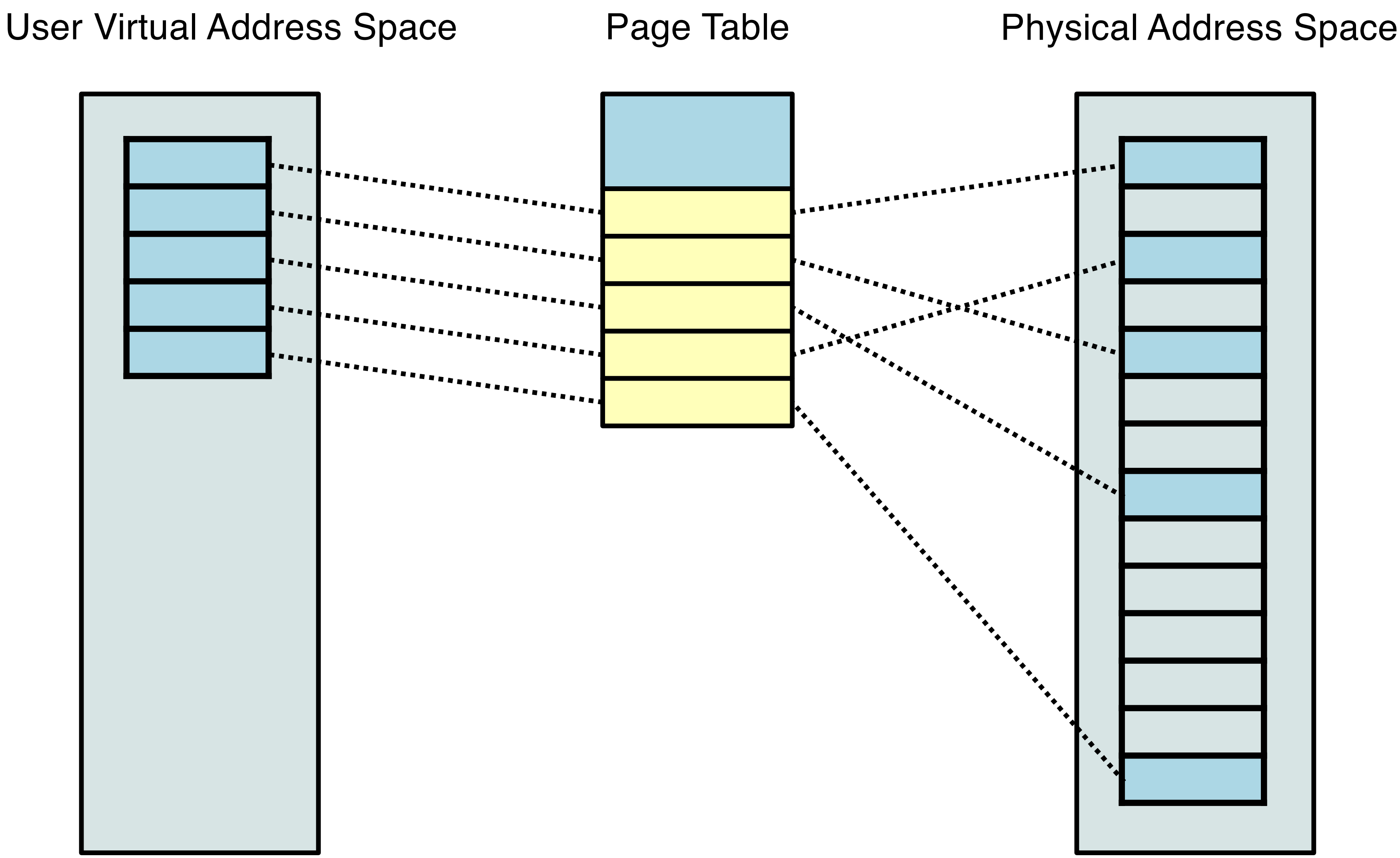


# Kernel space page table mapping

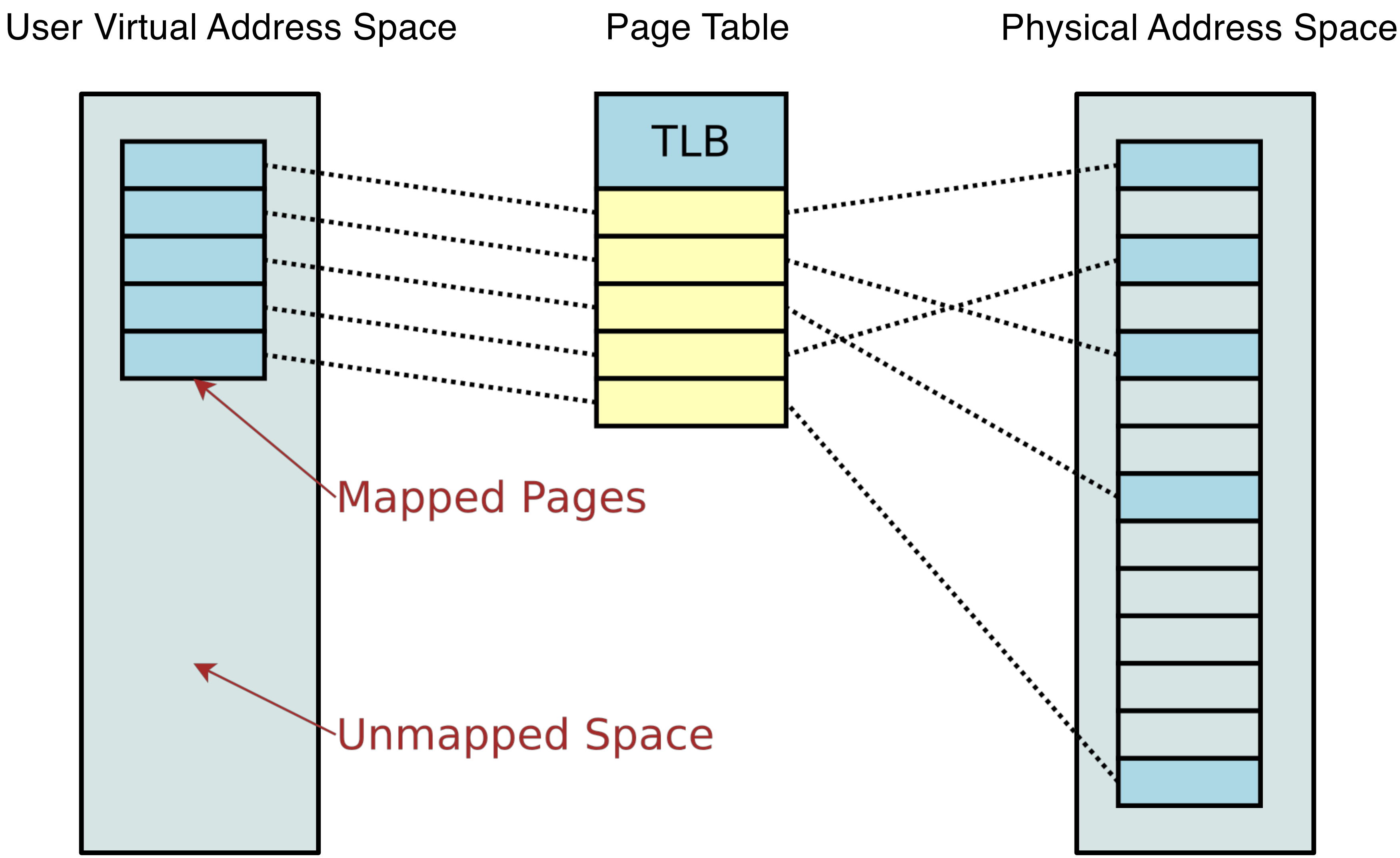
- Virtual contiguous, physical non-contiguous
- Virtual and physical contiguous



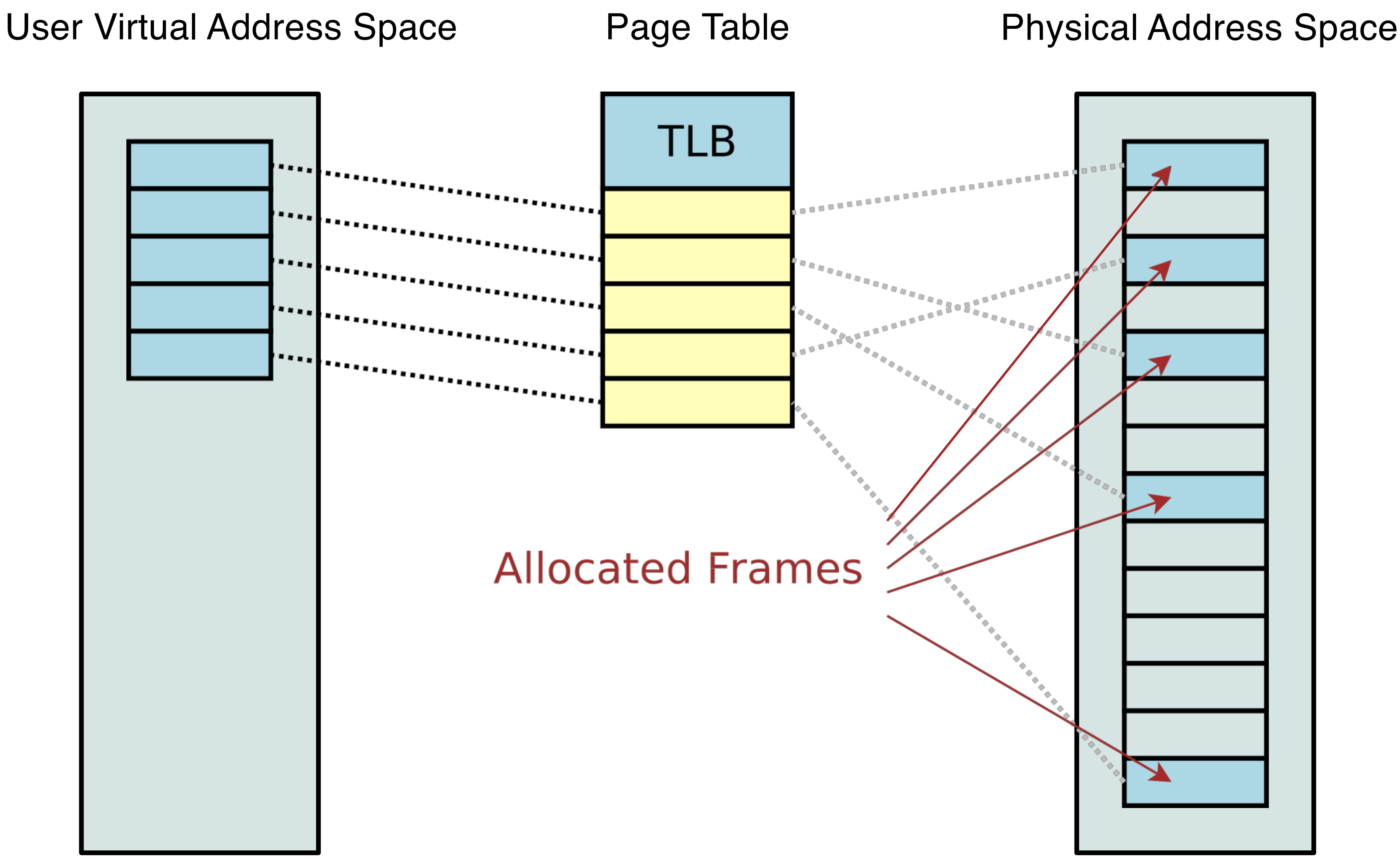
# Basic Page Table Mappings



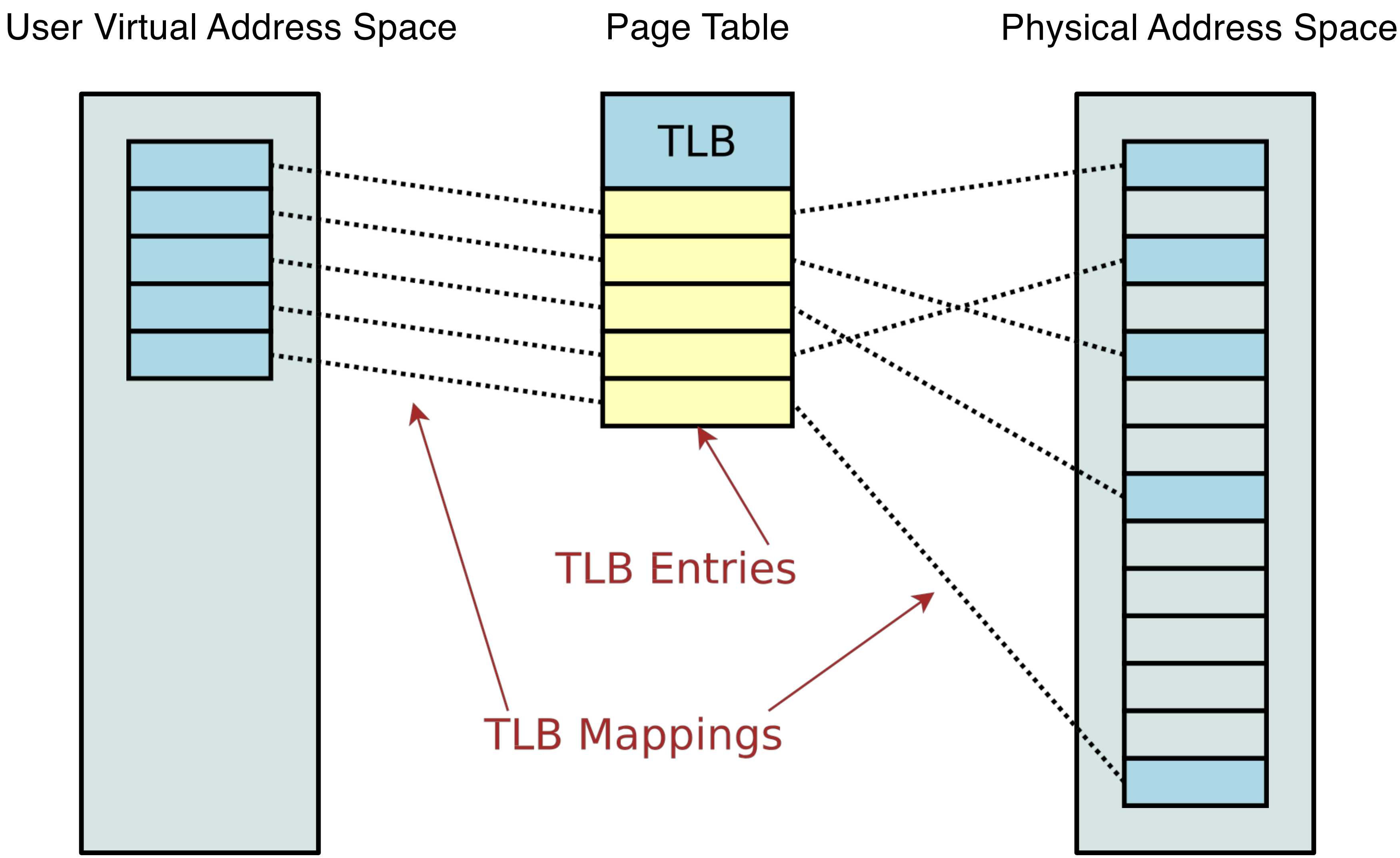
# Basic Page Table Mappings



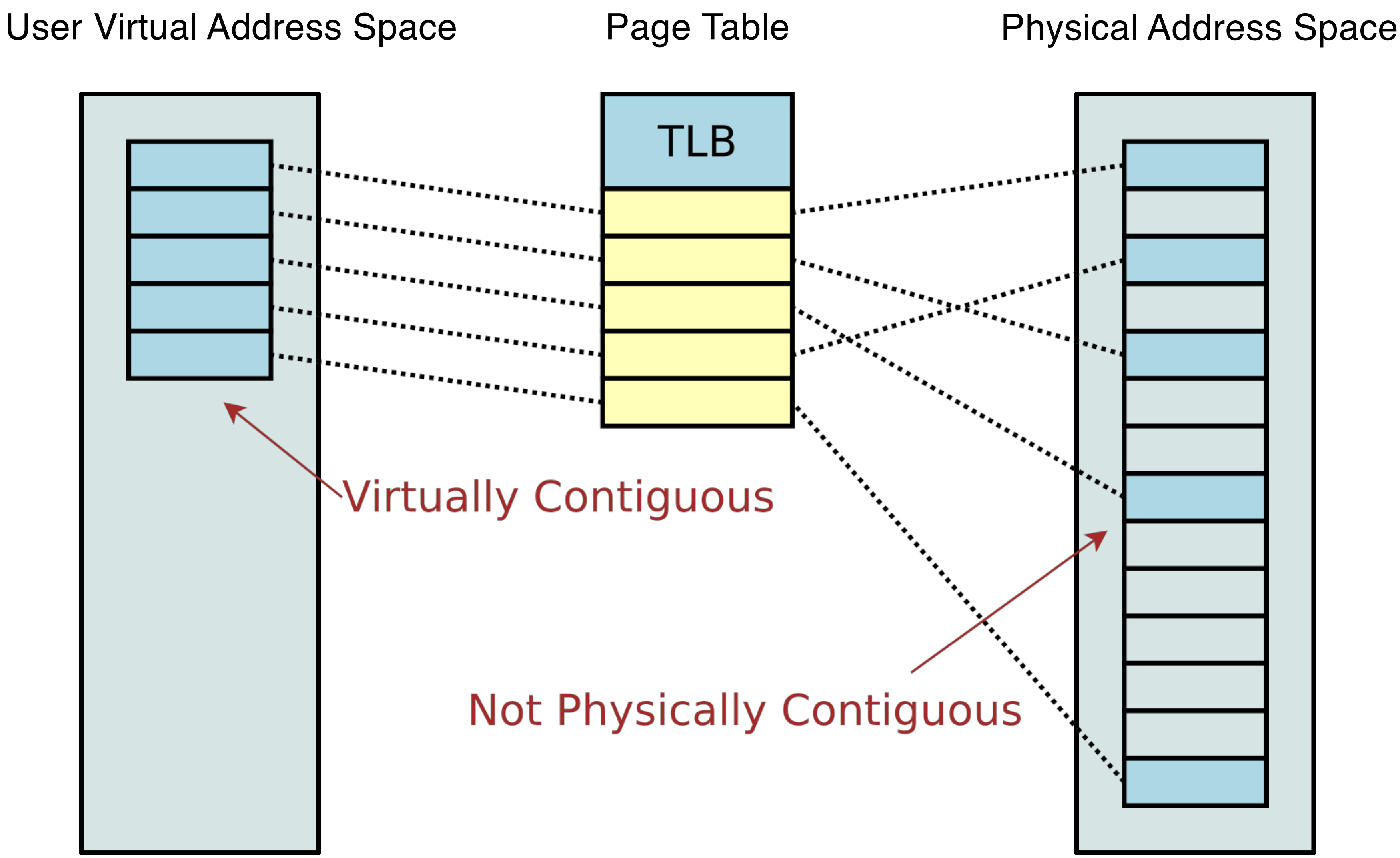
# Basic Page Table Mappings



# Basic Page Table Mappings



# Basic Page Table Mappings



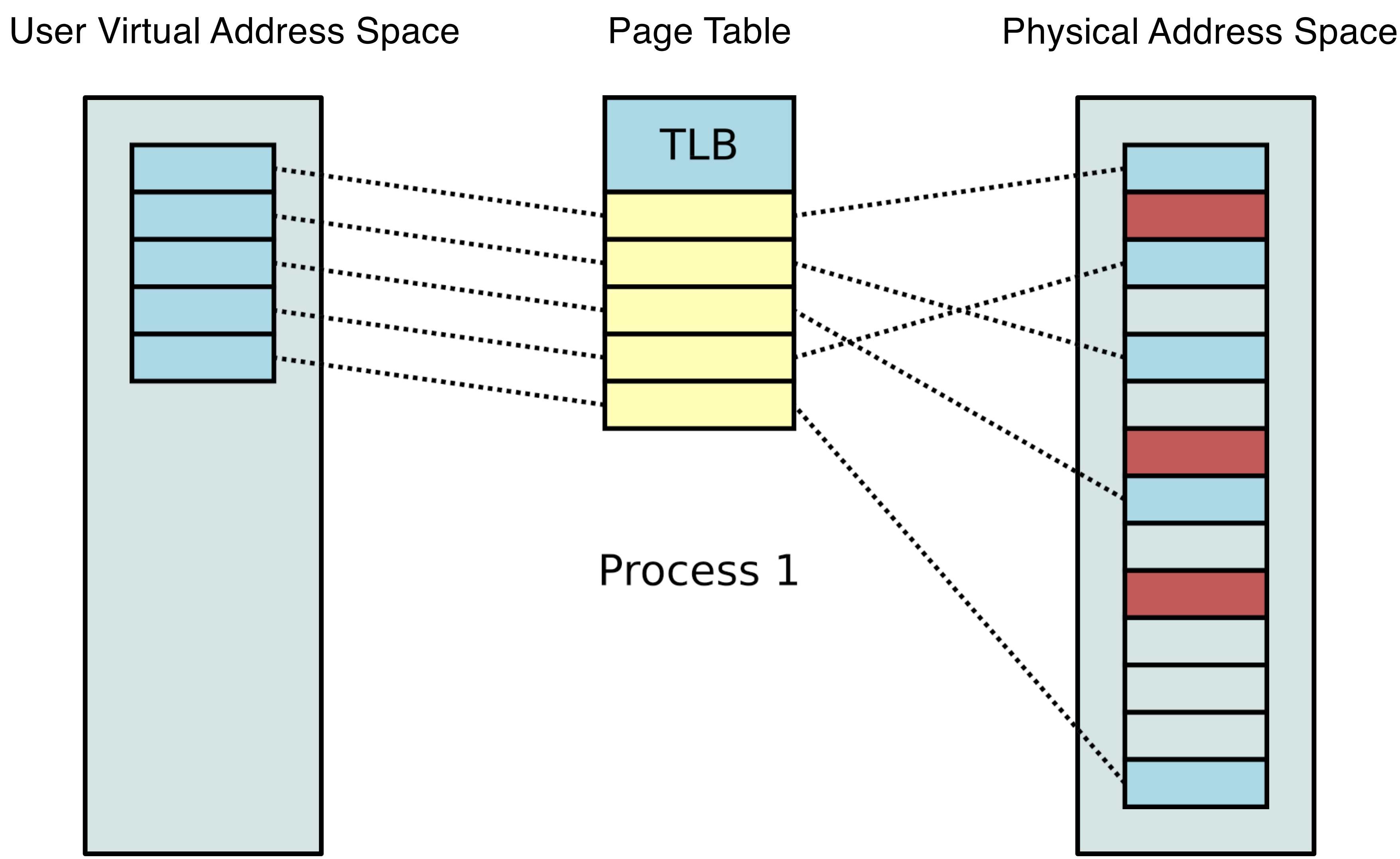
# Page Table Mappings

- Mappings to virtually contiguous regions do not have to be physically contiguous.
  - This makes memory easier to allocate.
  - Almost all user space code does not need physically contiguous memory.

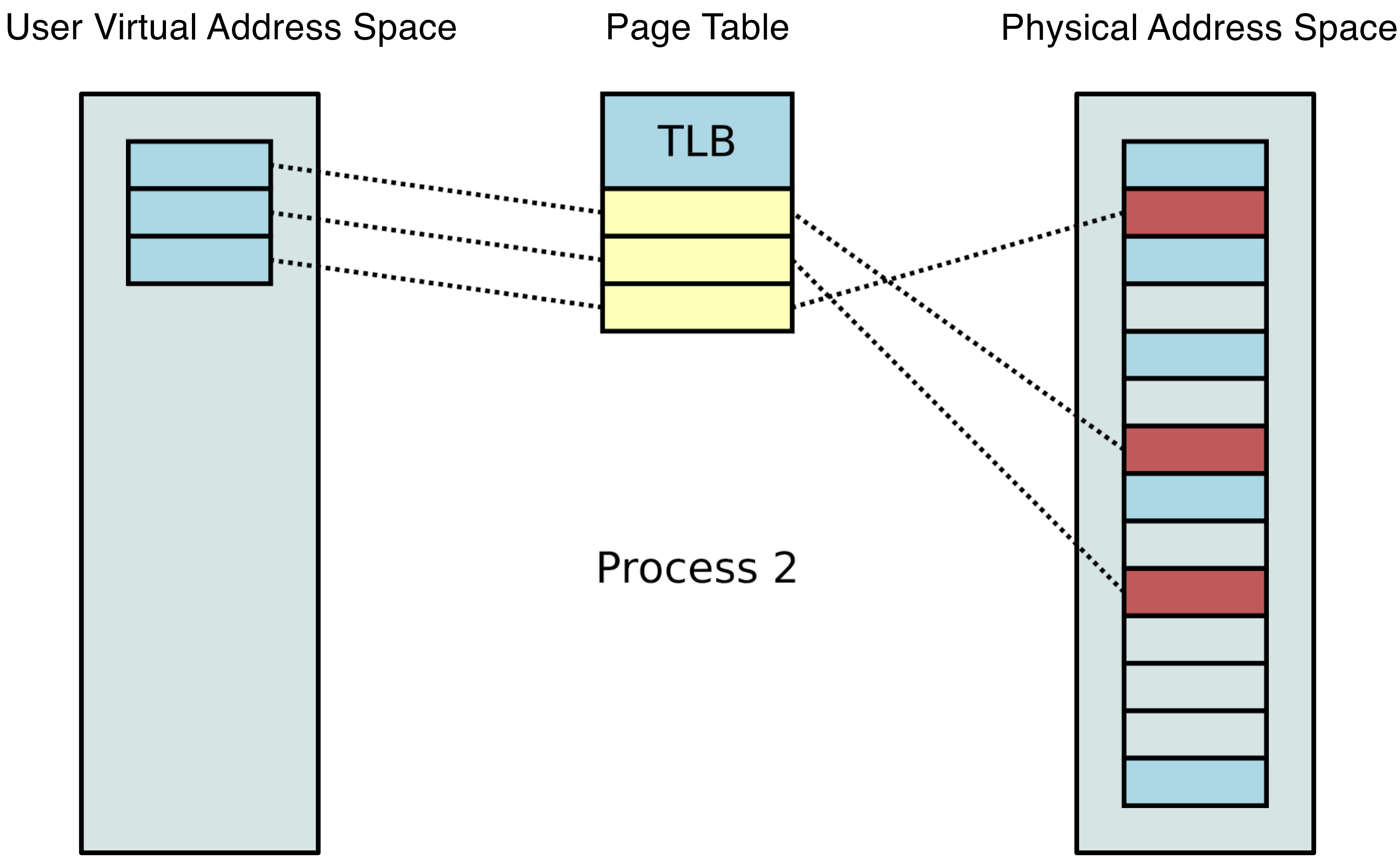
# Multiple Processes

- Each process has its own mapping.
  - The same virtual addresses in different processes point to different physical addresses in other processes

# Multiple Processes – Process 1



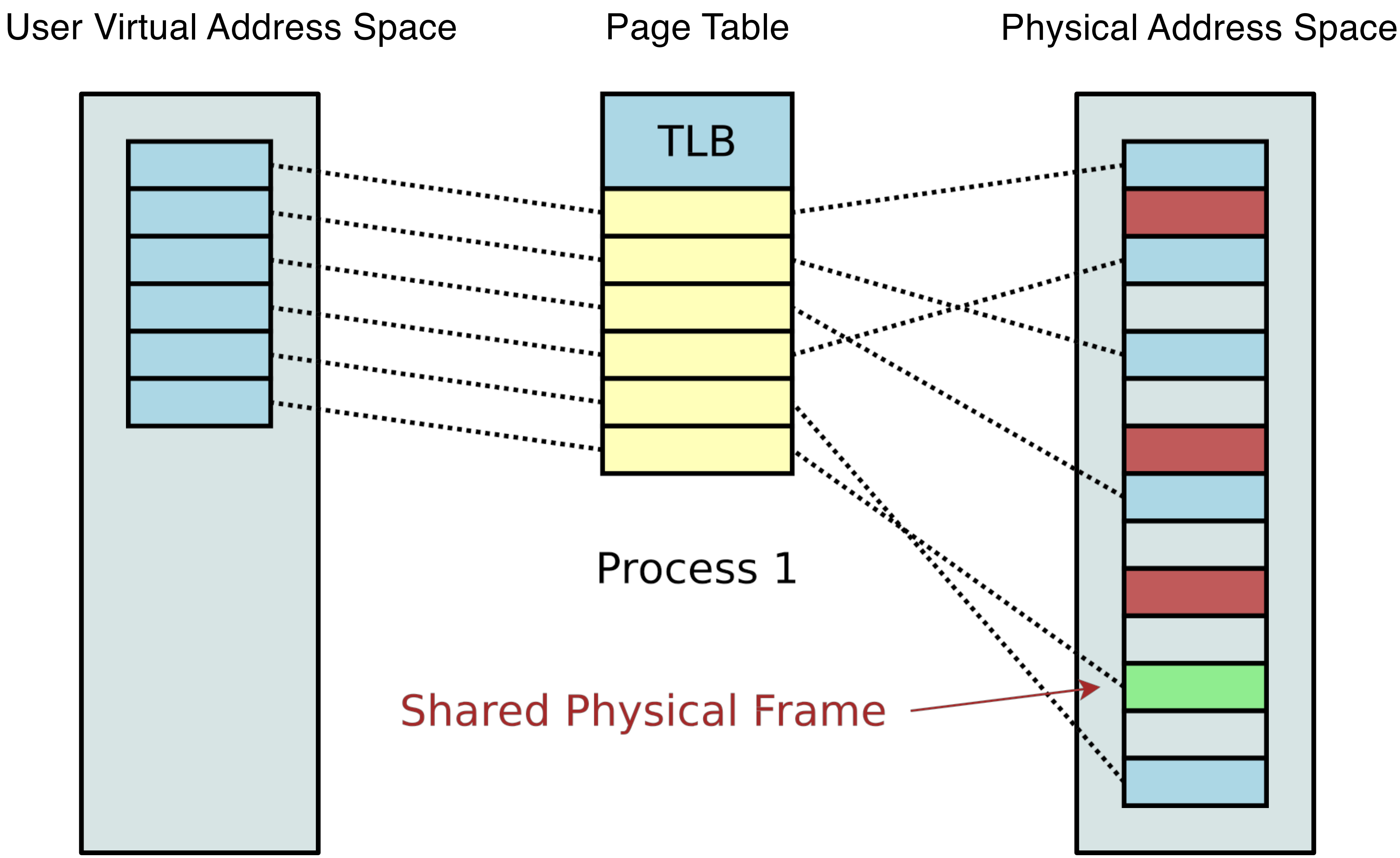
# Multiple Processes – Process 2



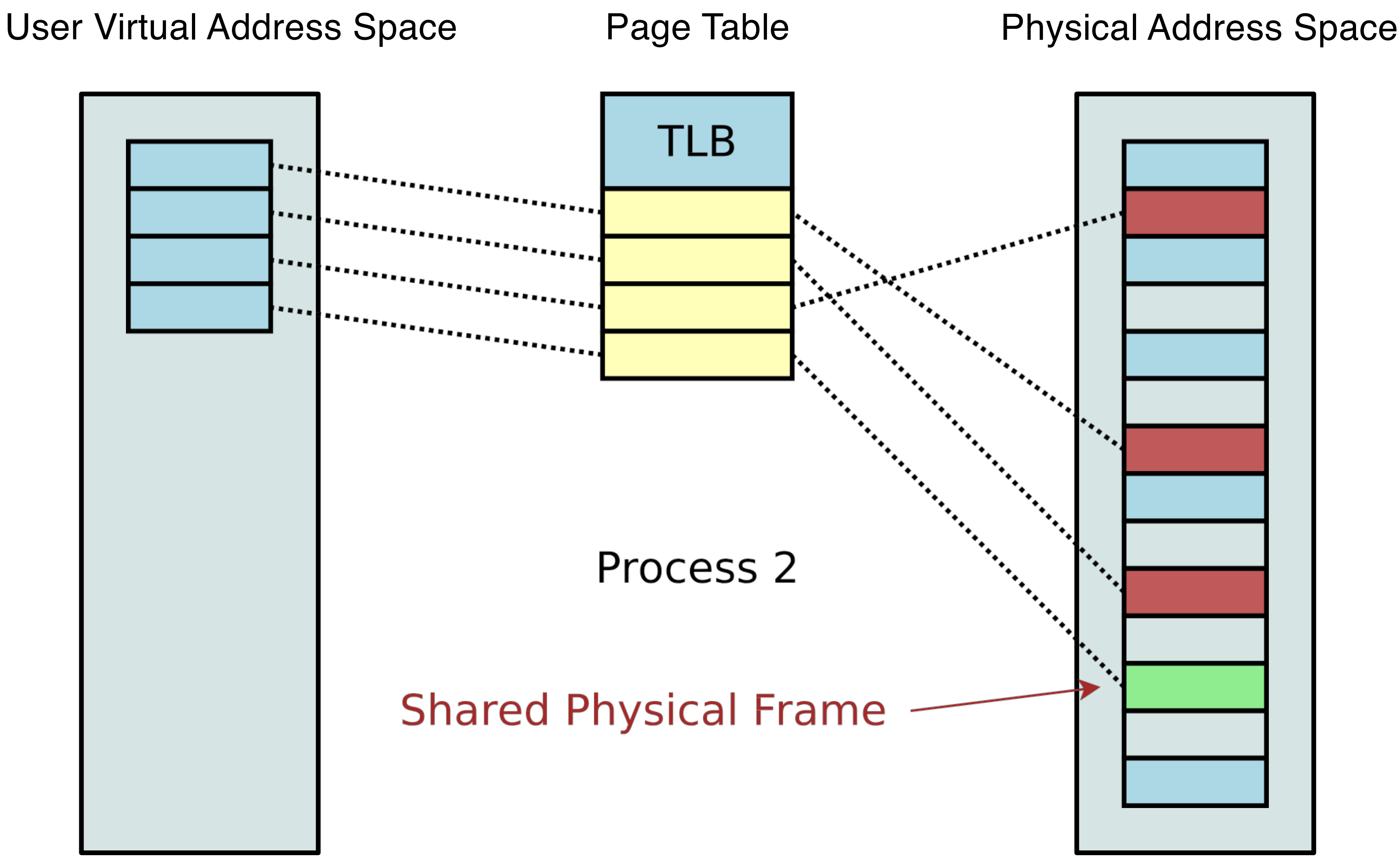
# Shared Memory

- Shared memory is easily implemented with an MMU.
  - Simply map the same physical frame into two different processes.
  - The virtual addresses need not be the same.
    - If pointers to values inside a shared memory region are used, it might be important for them to have the same virtual addresses, though.

# Shared Memory - Process 1



# Shared Memory - Process 2



# Shared Memory

- Note in the previous example, the shared memory region was mapped to different virtual addresses in each process.
- The `mmap()` system call allows the user space process to request a virtual address to map the shared memory region.
  - The kernel may not be able to grant a mapping at this address, causing `mmap()`to return failure.

# Kernel Logical Addresses - Repeat

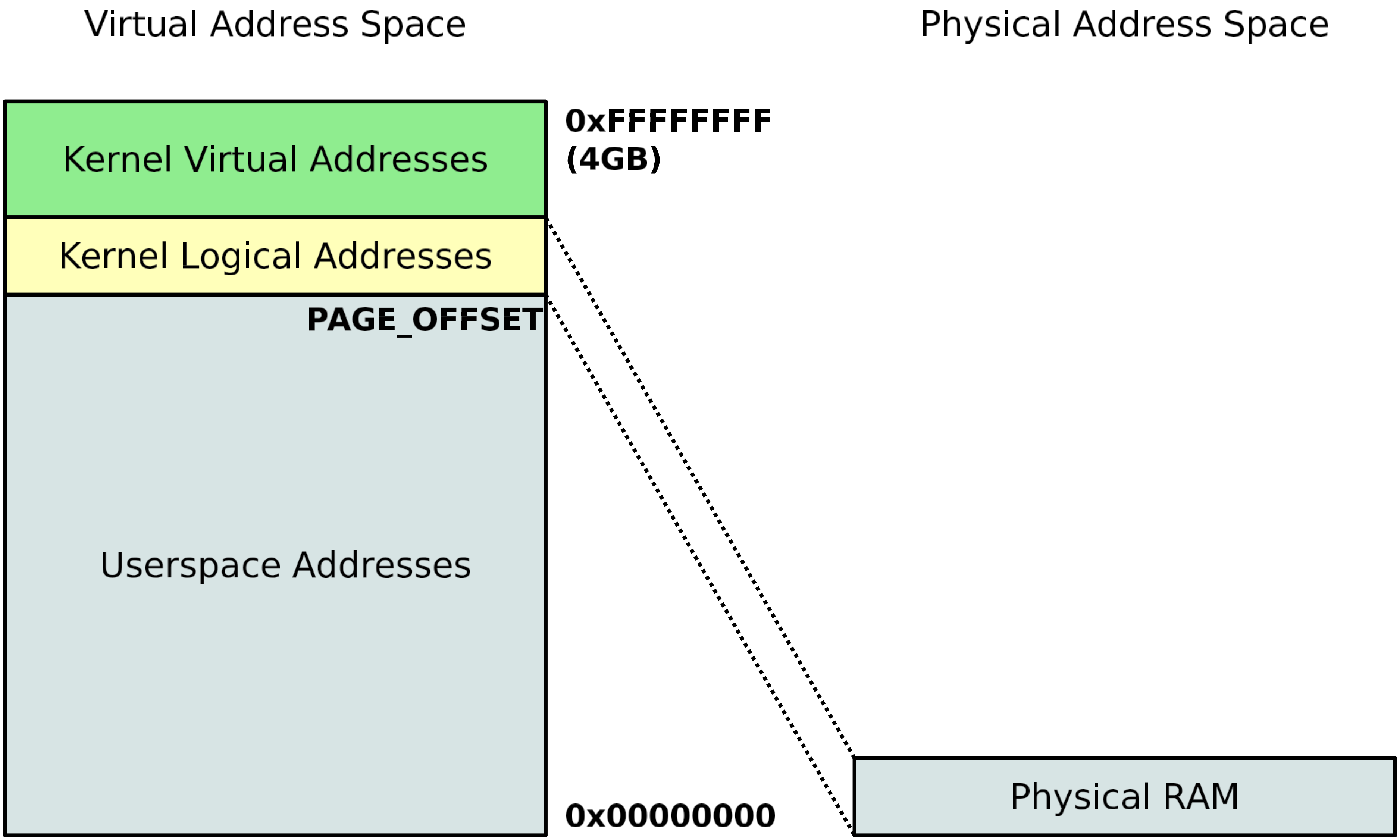
- Kernel Logical addresses can be converted to and from physical addresses using the macros:

  __pa(x)

  __va(x)

- For low-memory systems (below ~1G of RAM) Kernel Logical address space starts at PAGE_OFFSET and goes through the end of *physical* memory.

# Kernel Virtual Addresses



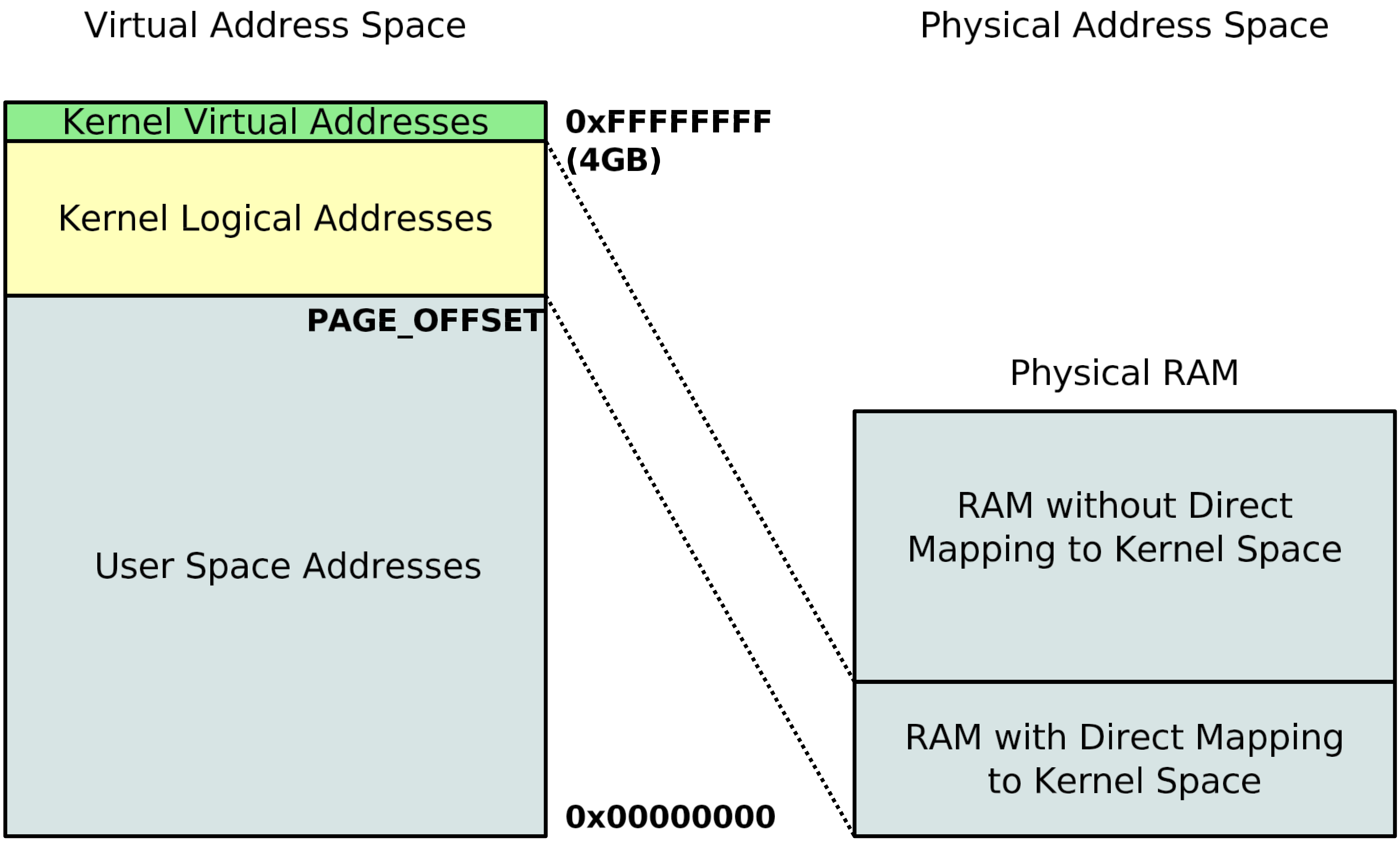
# Kernel Vitrual Addresses

- In the small memory model, as shown, since all of RAM can be represented by logical addresses, all virtual addresses will also have logical addresses.
  - One mapping in virtual address area
  - One mapping in logical address area

# Kernel Virtual Addresses

- The important difference is that memory in the kernel virtual address area (or vmalloc() area) is non-contiguous physically.
  - This makes it easier to allocate, especially for large buffers
  - This makes it unsuitable for DMA

# Kernel Virtual Addresses (Large Mem )



# Kernel Virtual Addresses

- In a large memory situation, the kernel virtual address space is smaller, because there is more physical memory.
  - An interesting case, where more memory means less virtual address space.
  - In 64-bit, of course, this doesn't happen, as PAGE_OFFSET is large, and there is much more virtual address space.

# User Virtual Addresses

- User Virtual Addresses represent memory used by user space programs.
  - This is most of the memory on most systems
  - This is where most of the complication is
- User virtual addresses are all addresses below PAGE_OFFSET.
- Each process has its own mapping

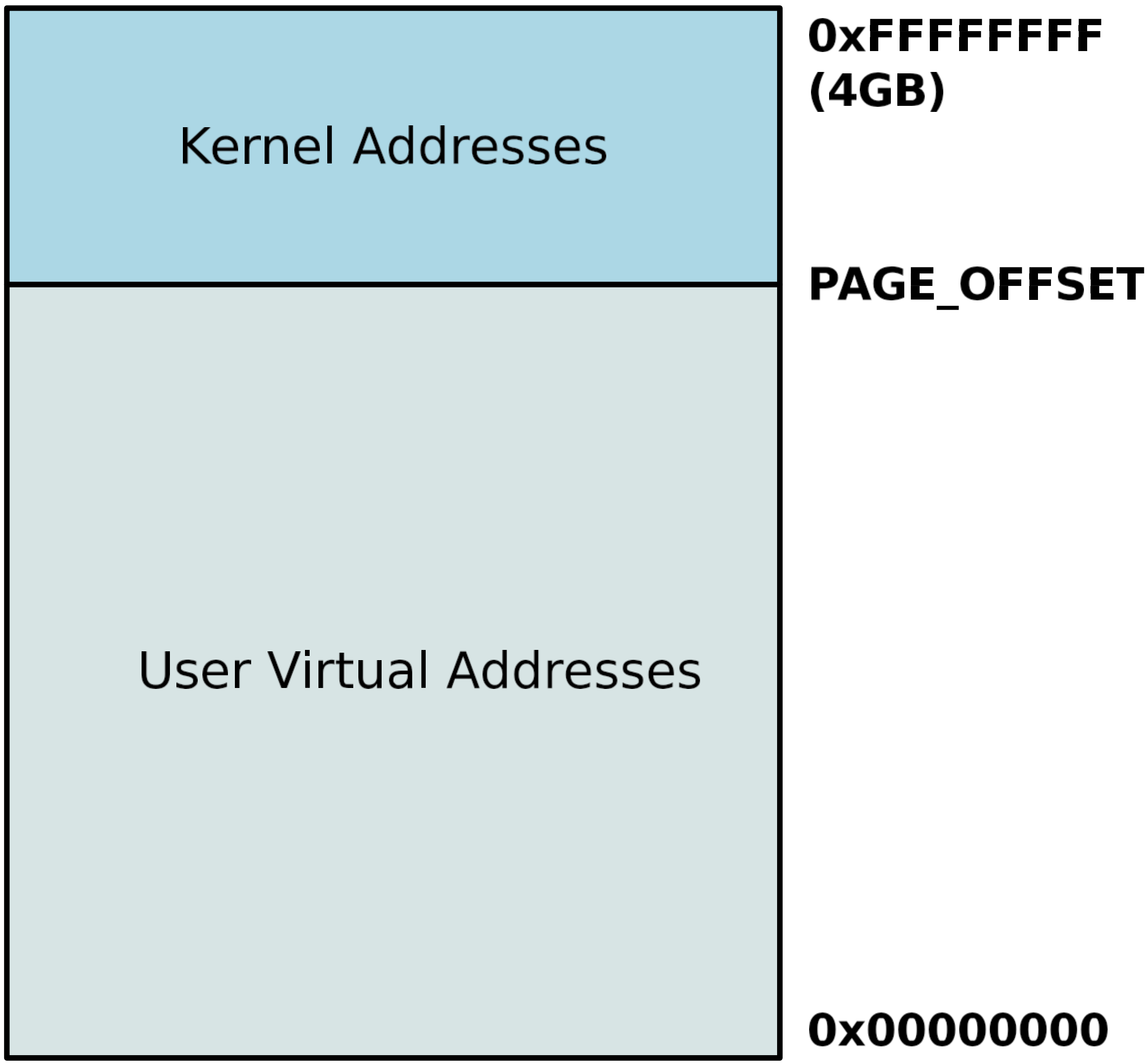
# User Virtual Addresses

- Unlike kernel logical addresses, which use a linear mapping between virtual and physical addresses, user space processes make full use of the MMU.
  - Only the used portions of RAM are mapped
  - Memory is not contiguous
  - Memory may be swapped out
  - Memory can be moved

# User Virtual Addresses

- Since user virtual addresses are not guaranteed to be swapped in, or even allocated at all, user pointers are not suitable for use with kernel buffers or DMA, by default.
- Each process has its own memory map
  - mm_struct
- At context switch time, the memory map of the new process will be switched in
  - This is part of the context switch overhead

# User Virtual Addresses




- Each process will have its own mapping for user virtual addresses
- The mapping is changed during context switch

# Show me the code

- Talk is cheap, show me the code

# Memory Management Unit

# The MMU

- The Memory Management Unit (MMU) is a hardware component which manages virtual address mappings
  - Maps virtual addresses to physical addresses
- The MMU operates on basic units of memory called pages
  - Page size varies by architecture
  - Some architectures have configurable page sizes

# The MMU

- Common page sizes:
  - ARM – 4k
  - ARM64 – 4k or 64k
  - MIPS – Widely Configurable
  - x86 – 4k
    - ➢ *Architectures which are configurable are configured at kernel build time.*

# The MMU

- Terminology
  - A **page** is a unit of memory sized and aligned at the page size.
  - A **page frame**, or frame, refers to a page-sized and page-aligned physical memory block.
    - ➢ *A page is somewhat abstract, where a frame is concrete*
    - ➢ *In the kernel, the abbreviation **pfn**, for **page frame number**, is often used to refer to refer to physical page frames*

# The MMU

- The MMU operates in pages
  - The MMU maps physical frames to virtual addresses.
  - The TLB holds the entries of the mapping
    - Virtual address
    - Physical address
    - Permissions
  - A memory map for a process will contain many mappings

# Page Faults

- When a process accesses a region of memory that is not mapped, the MMU will generate a page fault exception.
  - The kernel handles page fault exceptions regularly as part of its memory management design.

# Lazy Allocation

- The kernel will not allocate pages requested by a process immediately.
  - The kernel will wait until those pages are actually used.
  - This is called lazy allocation and is a performance optimization.
    - For memory that doesn't get used, allocation never has to happen!

# Lazy Allocation

- Process
  - When memory is requested, the kernel simply creates a record of the request, and then returns (quickly) to the process, without updating the TLB.
  - When that newly-allocated memory is touched, the CPU will generate a page fault, because the CPU doesn't know about the mapping

# Lazy Allocation

- Process (cont)
  - In the page fault handler, the kernel determines that the mapping is valid (from the kernel's point of view).
  - The kernel updates the TLB with the new mapping
  - The kernel returns from the exception handler and the user space program resumes.

# Lazy Allocation

- In a lazy allocation case, the user space program never is aware that the page fault happened.
  - The page fault can only be detected in the time that was lost to handle it.
- For processes that are time-sensitive, data can be pre-faulted, or simply touched, at the start of execution.
  - Also see mlock() and mlockall() for pre-faulting.

# Page Tables

- The entries in the TLB are a limited resource.
- Far more mappings can be made than can exist in the TLB at one time.
- The kernel must keep track of all of the mappings all of the time.
- The kernel stores all this information in the page tables.

# Page Tables

- Since the TLB can only hold a limited subset of the total mappings for a process, some mappings will not have TLB entries.
  - When these addresses are touched, the CPU will generate a page fault, because the CPU has knowledge of the mapping.

# Page Tables

- When the page fault handler executes in this case, it will:
  - Find the appropriate mapping for the offending address in the kernel's page tables
  - Select and remove an existing TLB entry
  - Create a TLB entry for the page containing the address
  - Return to the user space process
    - ➢ *Observe the similarities to lazy allocation handling*

# Swapping

- When memory allocation is high, the kernel may swap some frames to disk to free up RAM.
  - Having an MMU makes this possible.
- The kernel can copy a frame to disk and remove its TLB entry.
- The frame can be re-used by another process

# Swapping

- When the frame is needed again, the CPU will generate a page fault (because the address is not in the TLB)
- The kernel can then, at page fault time:
  - Put the process to sleep
  - Copy the frame from the disk into an  unused frame in RAM
  - Fix the page table entry  Wake the
  - process

# Swapping

- Note that when the page is restored to RAM, it's not necessarily restored to the same physical frame where it originally was located.
- The MMU will use the same virtual address though, so the user space program will not know the difference
  - *This is why user space memory cannot typically be used for DMA.*

# Takeaway

- Virtual memory in Linux
  - Kernel logical address
  - Kernel virtual address
  - User virtual address
- Contiguous vs non-contiguous
  - Virtually, physically
- Linux data struct
  - mm_struct, switch_mm, mm_struct.pgd
- Lazy allocation
  - Page fault

# Page table quiz

1) In 32-bit architecture, for 1-level page table, how large is the whole page table?
2) In 32-bit architecture, for 2-level page table, how large is the whole page table?
    1) How large for the 1st level PGT?
    2) How large for the 2nd level PGT?
3) Why can 2-level PGT save memory?
4) 2-level page table walk example
    1) Page table base register holds 0x0061 9000
    2) Virtual address is 0xf201 5202
    3) Page table base register holds 0x1051 4000
    4) Virtual address is 0x2190 7010