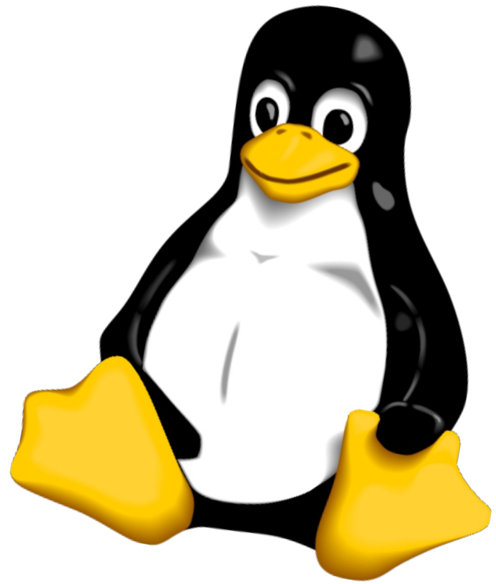
Virtual Memory and Linux

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



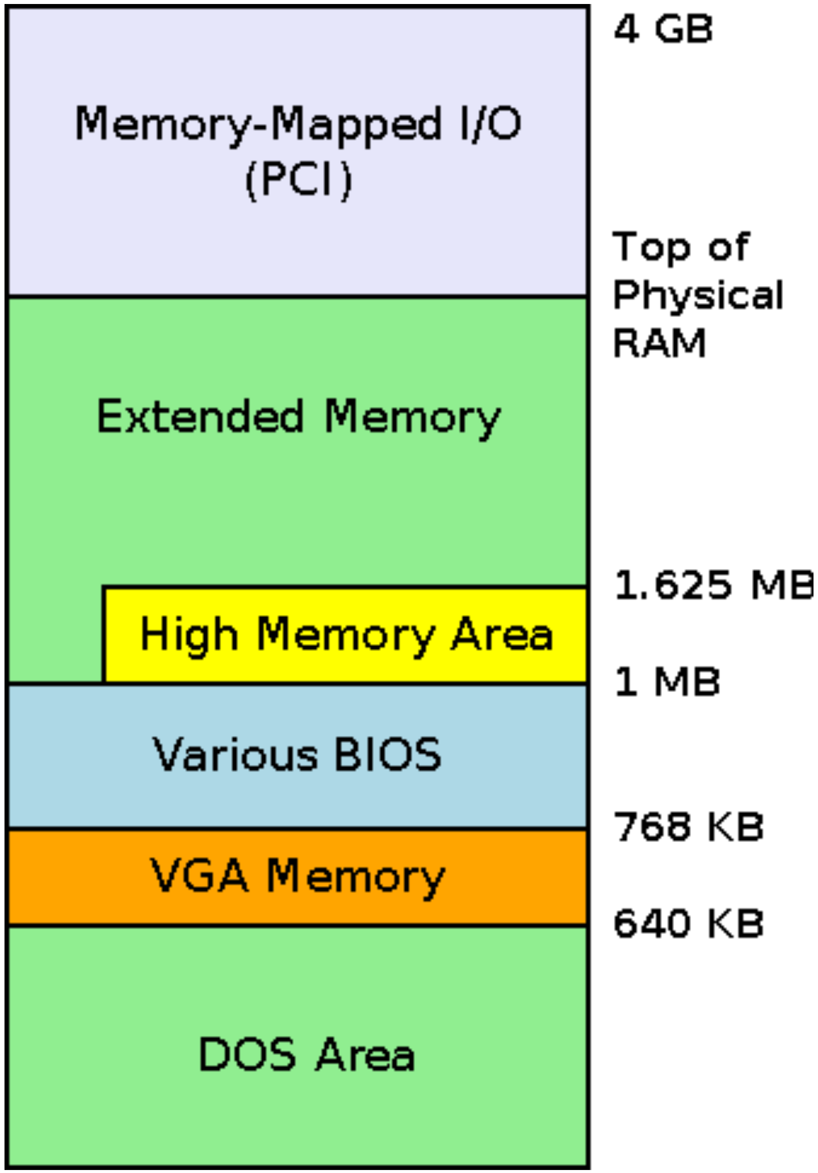
Flat Memory

- 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

Old x86 Memory Map

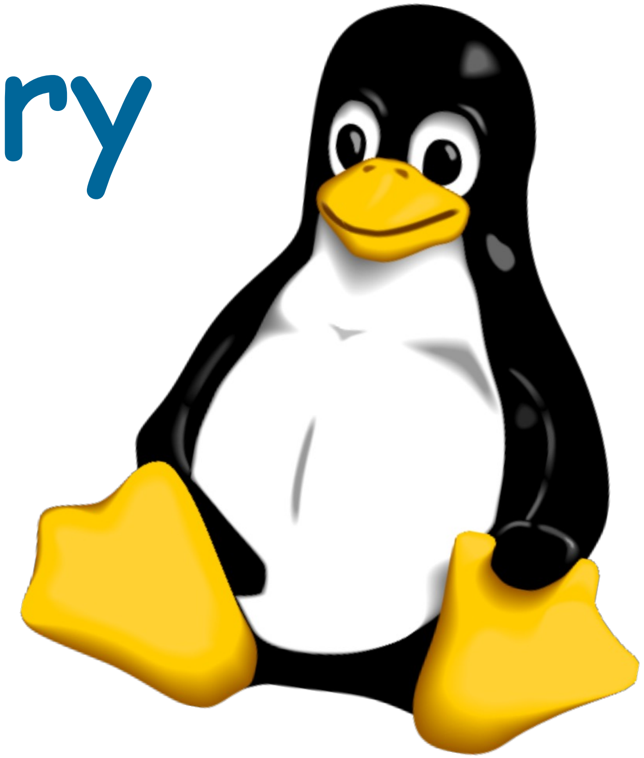


- Lots of Legacy
- RAM is split (DOS Area and Extended)
- Hardware mapped between RAM areas
- High and Extended area accessed differently

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 (logical 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 user-space 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

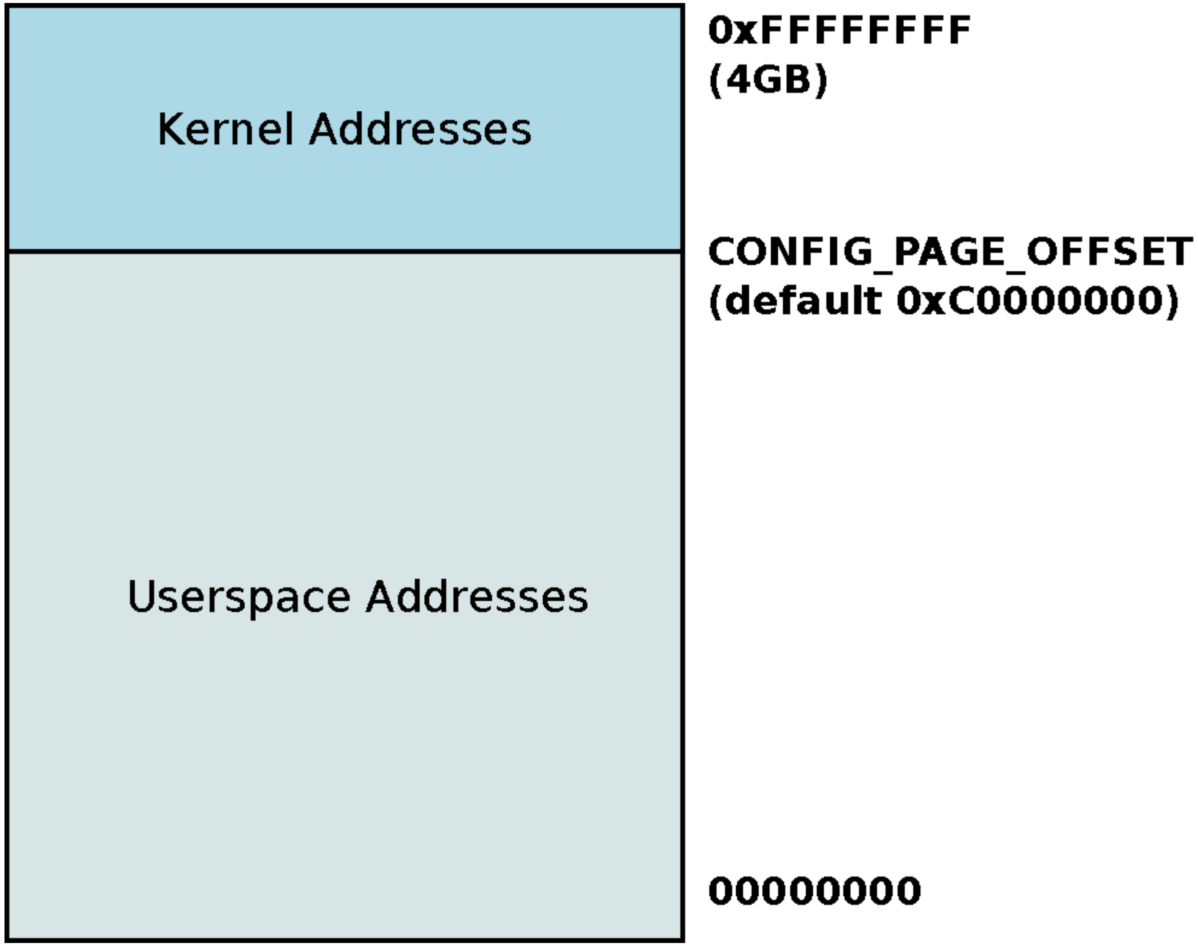
- The kernel uses lazy allocation of physical memory.
  - When memory is requested by userspace, physical memory is not allocated until it's touched.
  - This is an optimization, knowing that many userspace programs allocate more RAM than they ever touch.
    - Buffers, etc.

Virtual Addresses

- In Linux, the kernel uses virtual addresses, as userspace processes do.

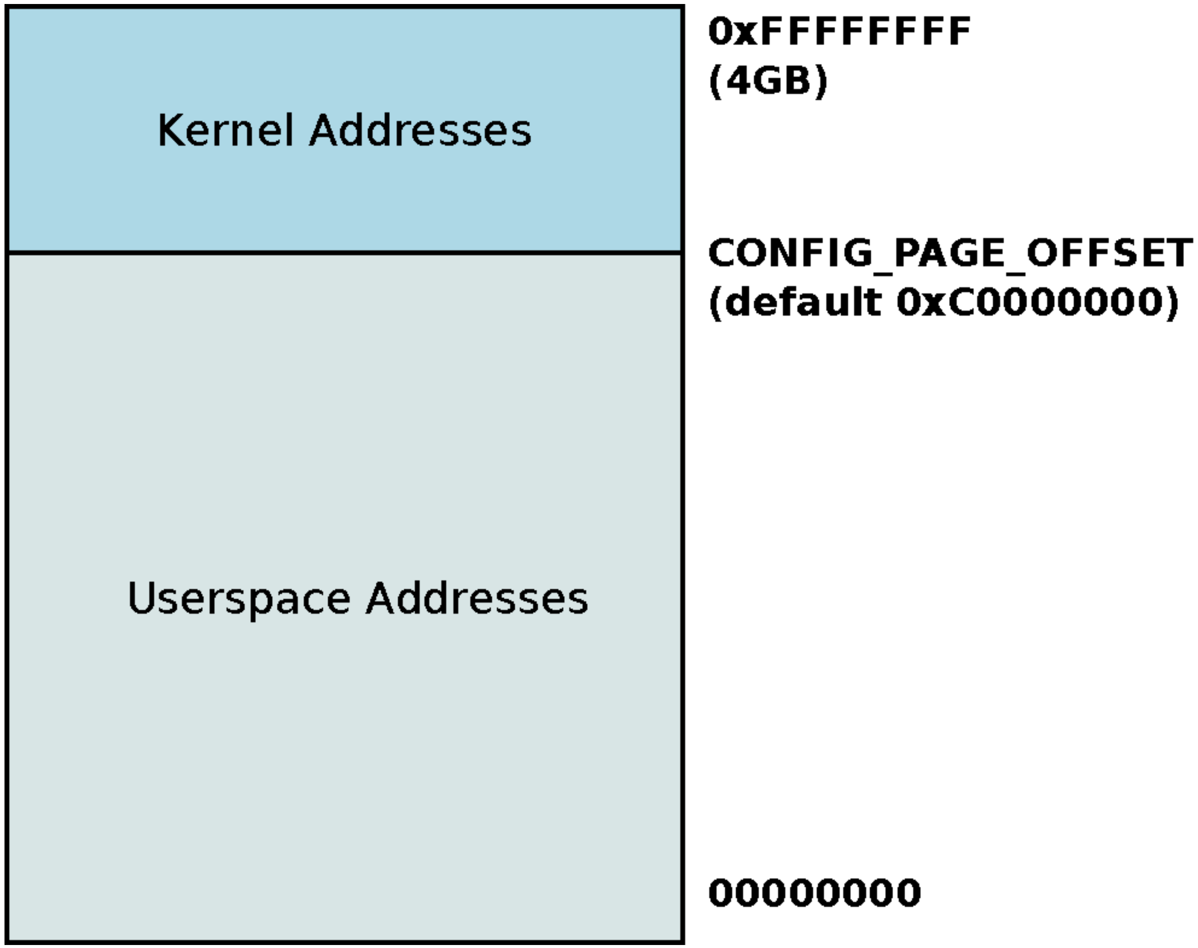
- Virtual address space is split.
  - The upper part is used for the kernel
  - The lower part is used for userspace
- On 32-bit, the split is at
  0xC0000000

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.

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

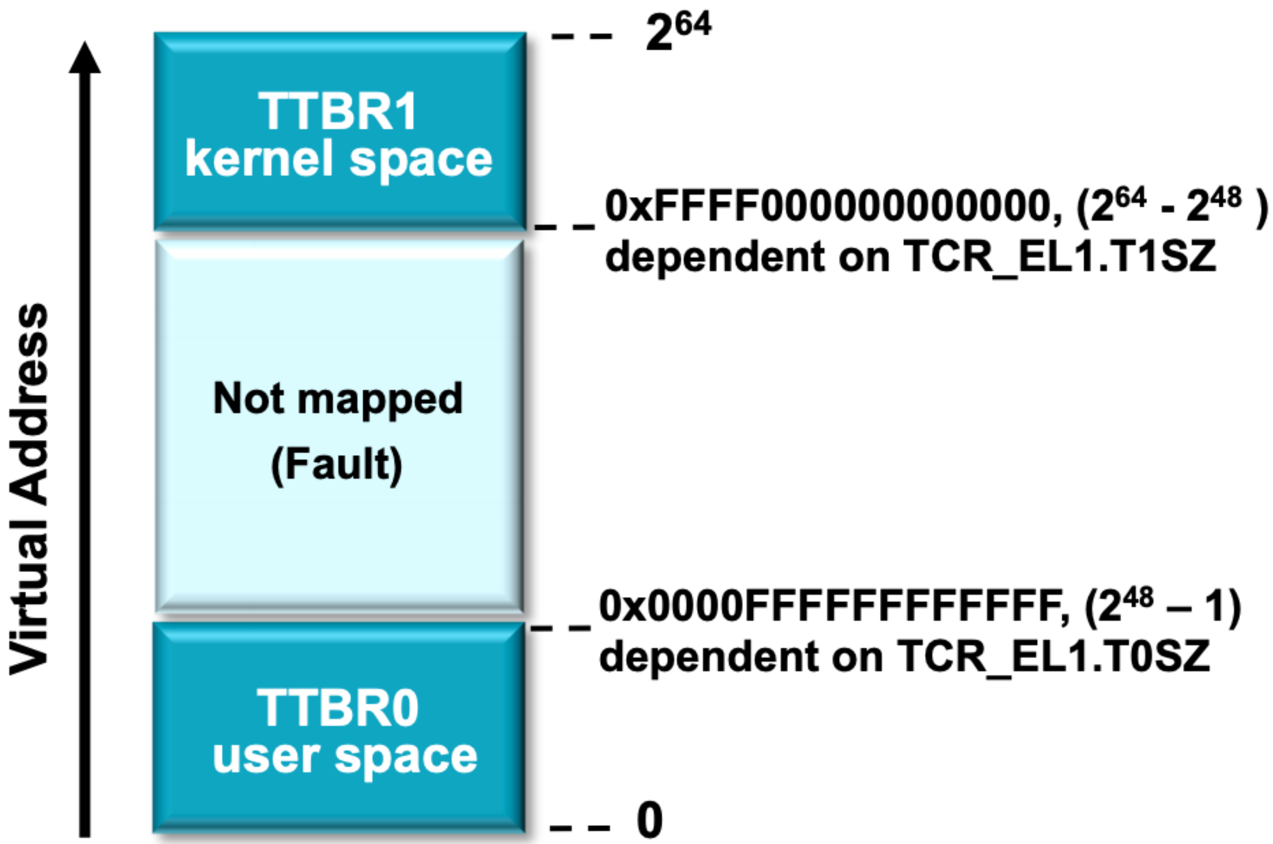
- Talk is cheap, show me the code
  - switch_mm()

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
    - 0xffff000000000000– ARM
    - 0xffff000000000000– x86_64

Virtual Addresses 64-bit

- ARM 64
  - 48-bit page table
  - 4-level: 9+9+9+9+12



Virtual Addresses 64-bit

- ARM 64
  - 39-bit page table
  - 3-level: 9+9+9+12

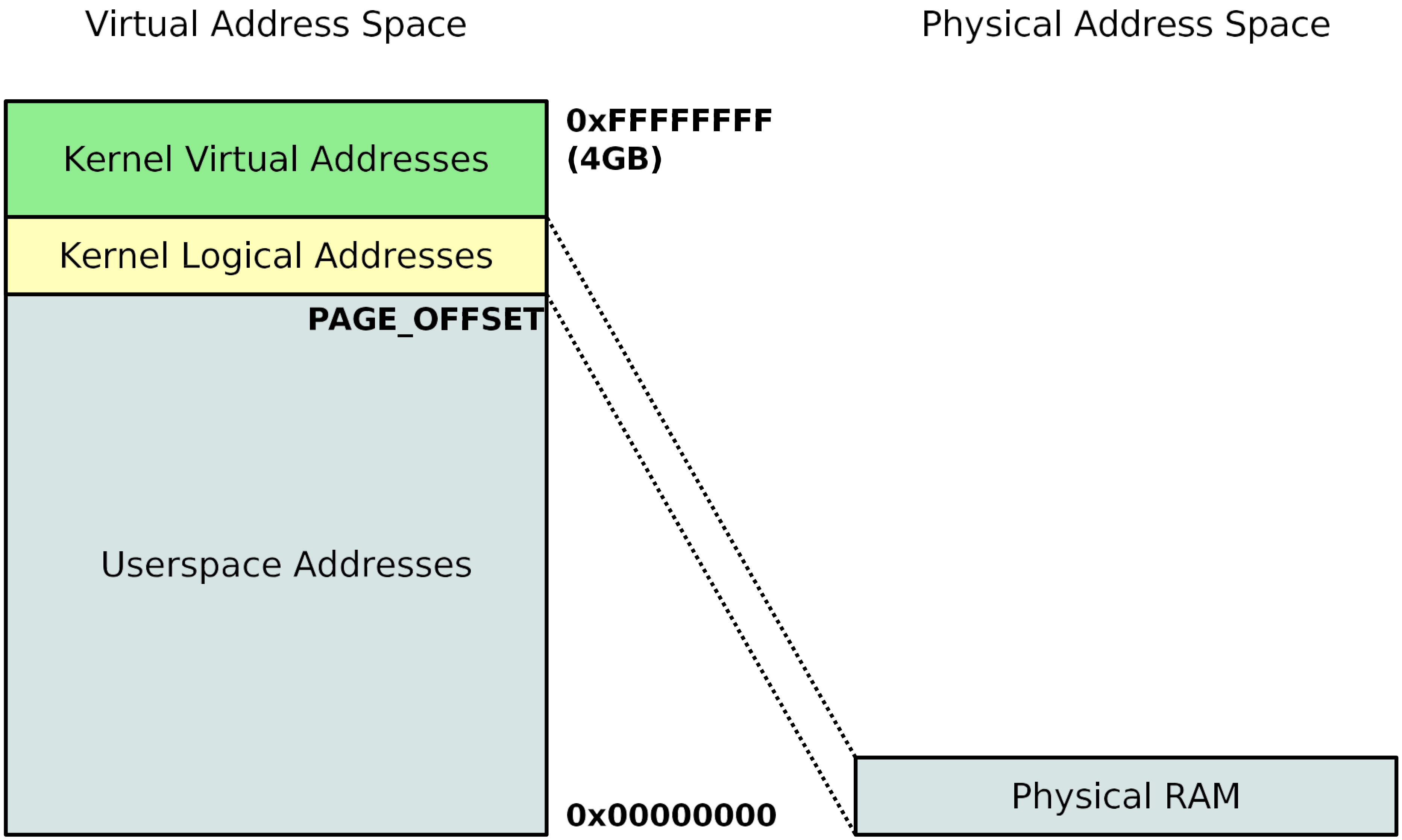
- 39-bit virtual address for both user and kernel
  - `0000000000000000-0000007fffffffff` (512GB): user
  - [architectural gap]
  - `ffffff8000000000-ffffffbbfffeffff` (~240MB): vmalloc
  - `ffffffbbffff0000-ffffffbcffffffff` (64KB): [guard]
  - `ffffffbc00000000-ffffffbdffffffff` (8GB): vmemmap
  - `ffffffbe00000000-ffffffbffbffffff` (~8GB): [guard]
  - `ffffffbffc000000-ffffffbfffffffff` (64MB): modules
  - `ffffffc000000000-ffffffffffffffff` (256GB): mapped RAM
- 4KB page configuration
  - 3 levels of page tables (pgtable-nopud.h)
  - Linear mapping using 4KB, 2MB or 1GB blocks
  - AArch32 (compat) supported

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 Virtual Addresses

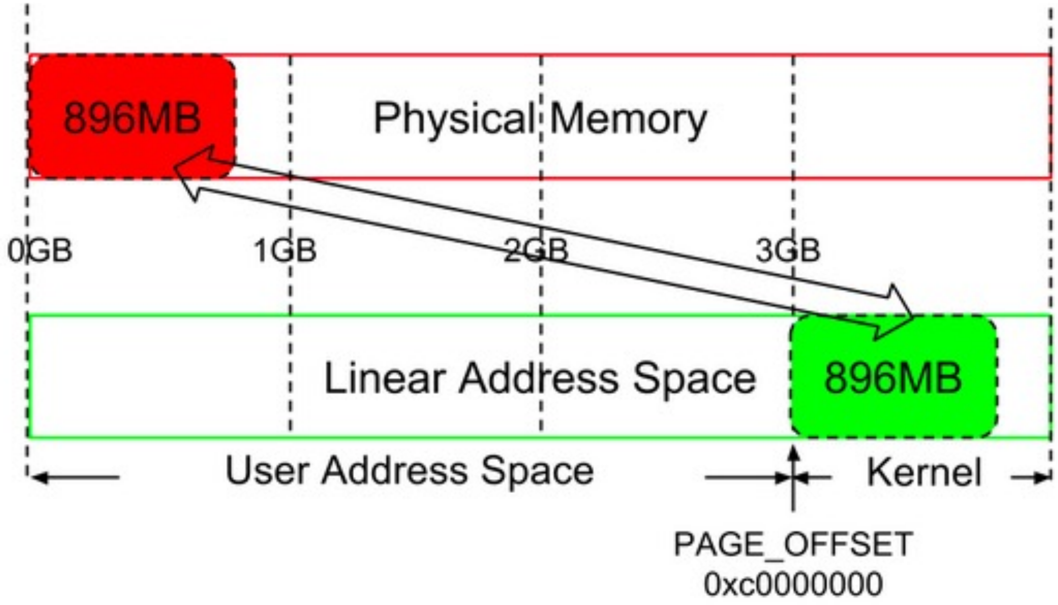


Kernel Logical Addresses

- 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?




```
#define __pa_to_va_nodebug(x)   ((void *)((unsigned long) (x) + va_pa_offset))
#define __va_to_pa_nodebug(x)   ((unsigned long)(x) - va_pa_offset)
```

Kernel Logical Addresses

- 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.

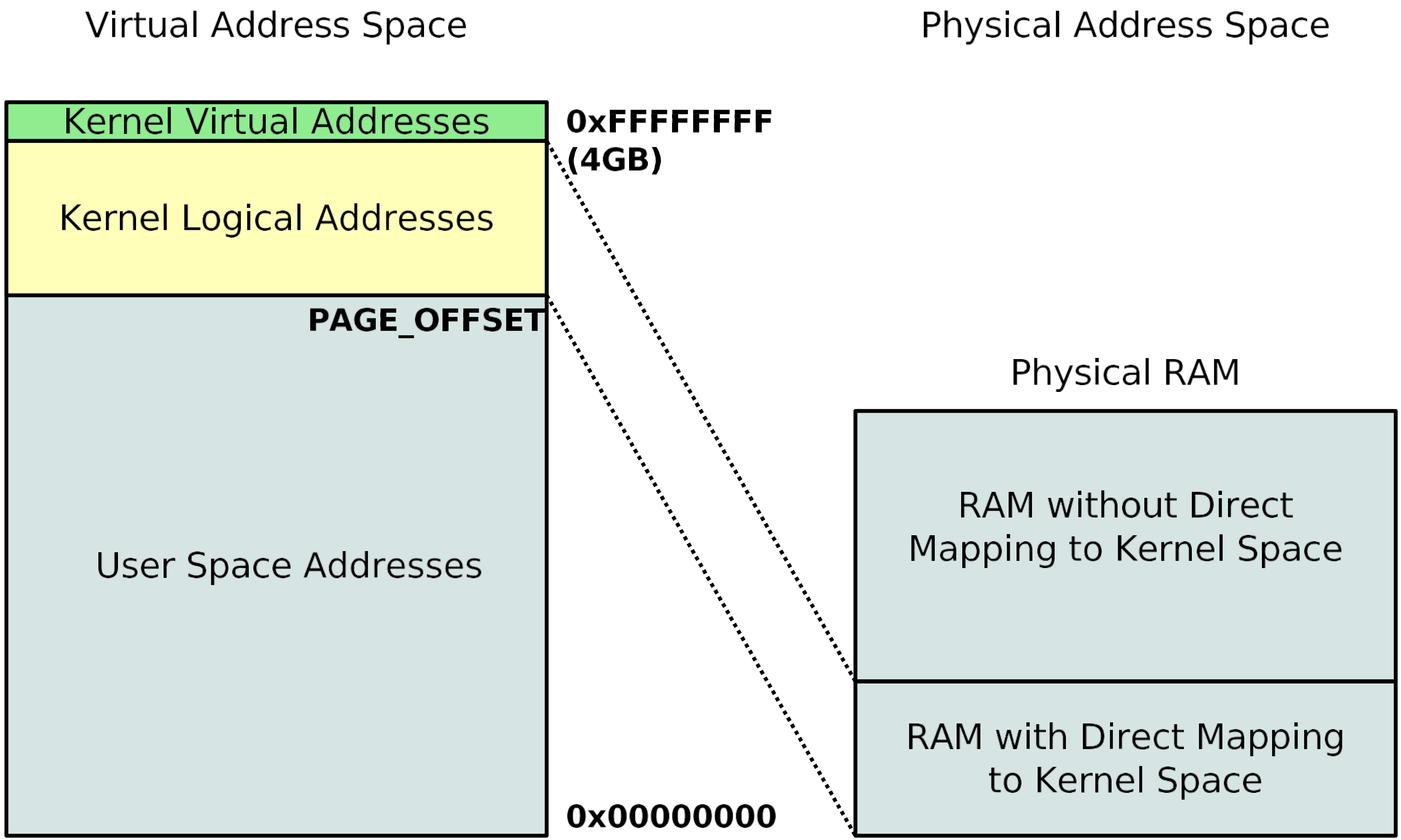
Large physical RAM

- 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 Virtual 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

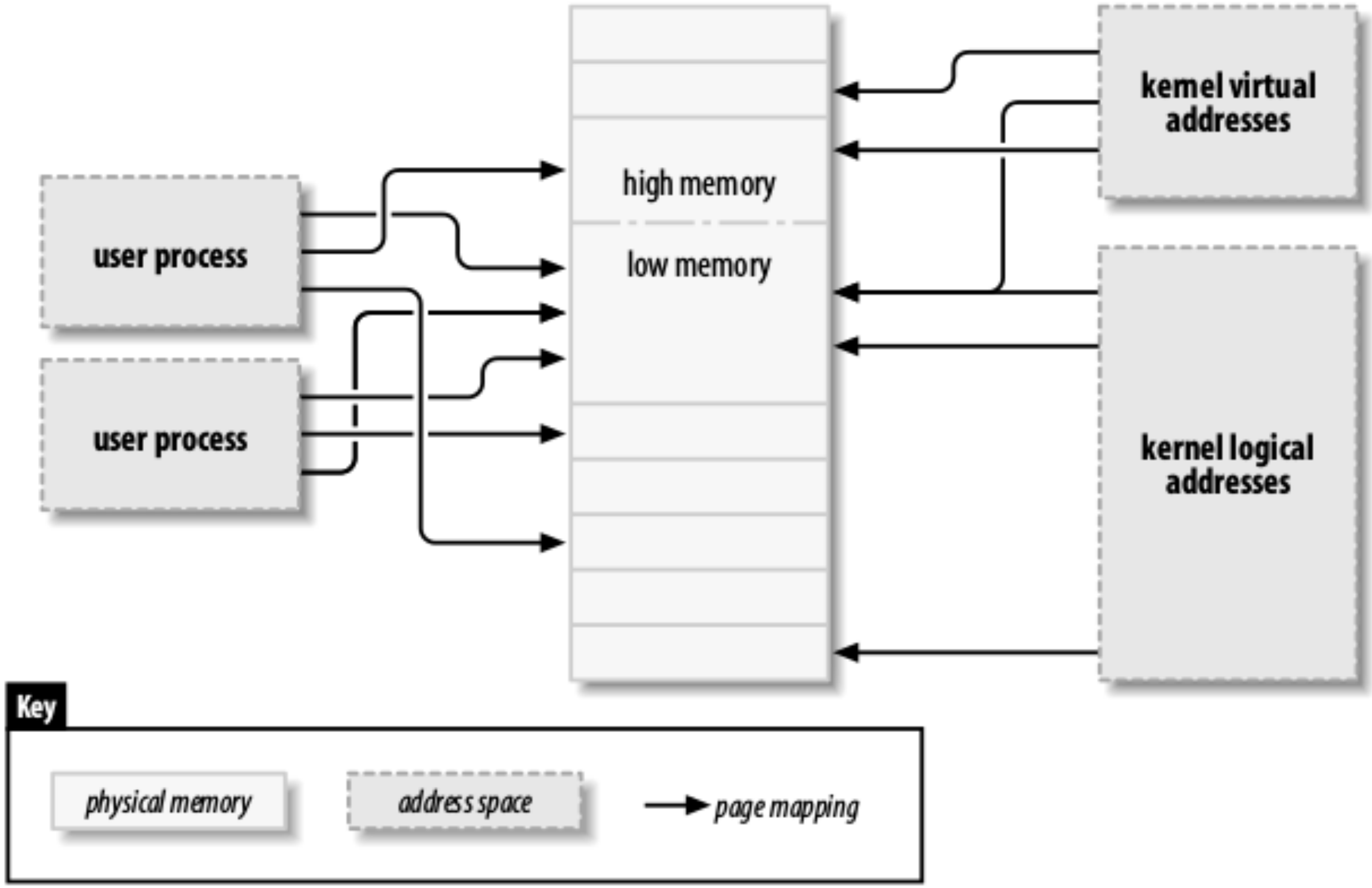
- Kernel Virtual Addresses are addresses in the region above the kernel logical address mapping.
- Kernel Virtual Addresses are used for non-contiguous memory mappings
  - Often for large buffers which could potentially be unable to get physically contiguous regions allocated.
- Also referred to as the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



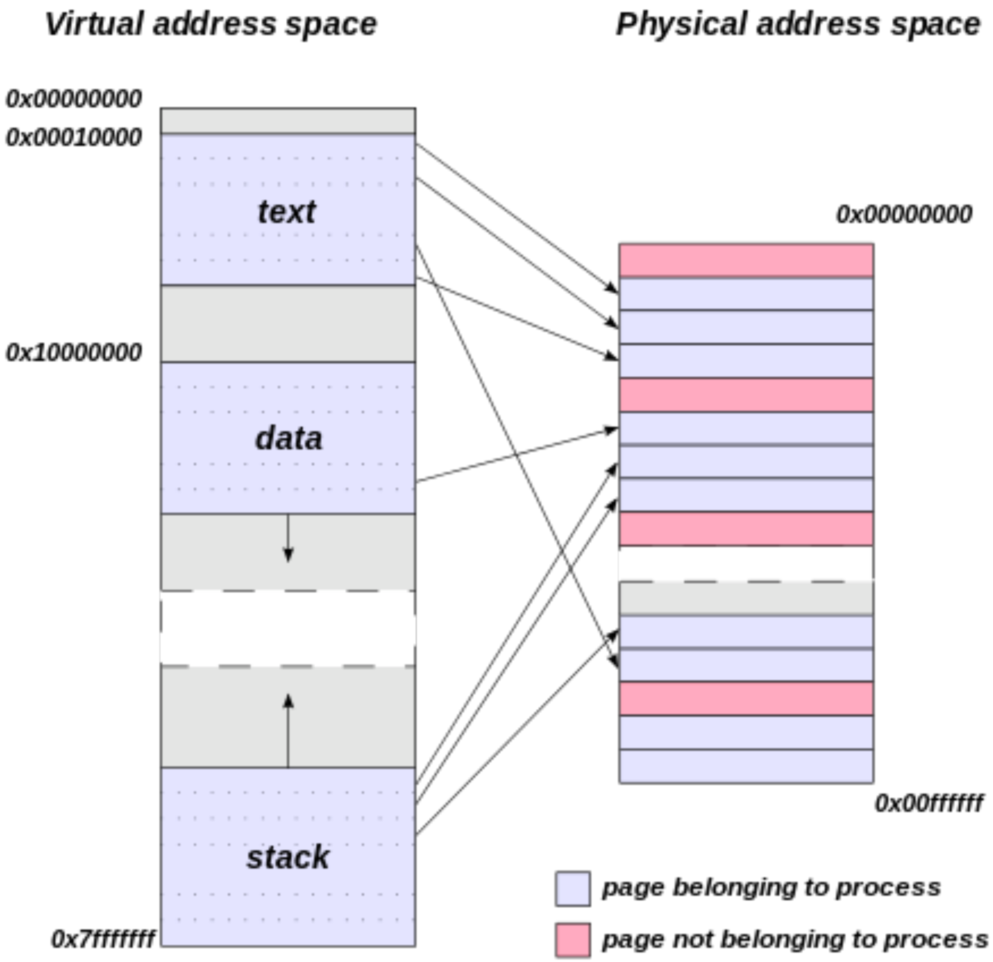
Virtual Addresses - Linux

- 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?
    - Linear mapping
  - Who has non-contiguous mapping?
    - Non-linear mapping
  - How to convert a VA to PA?

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

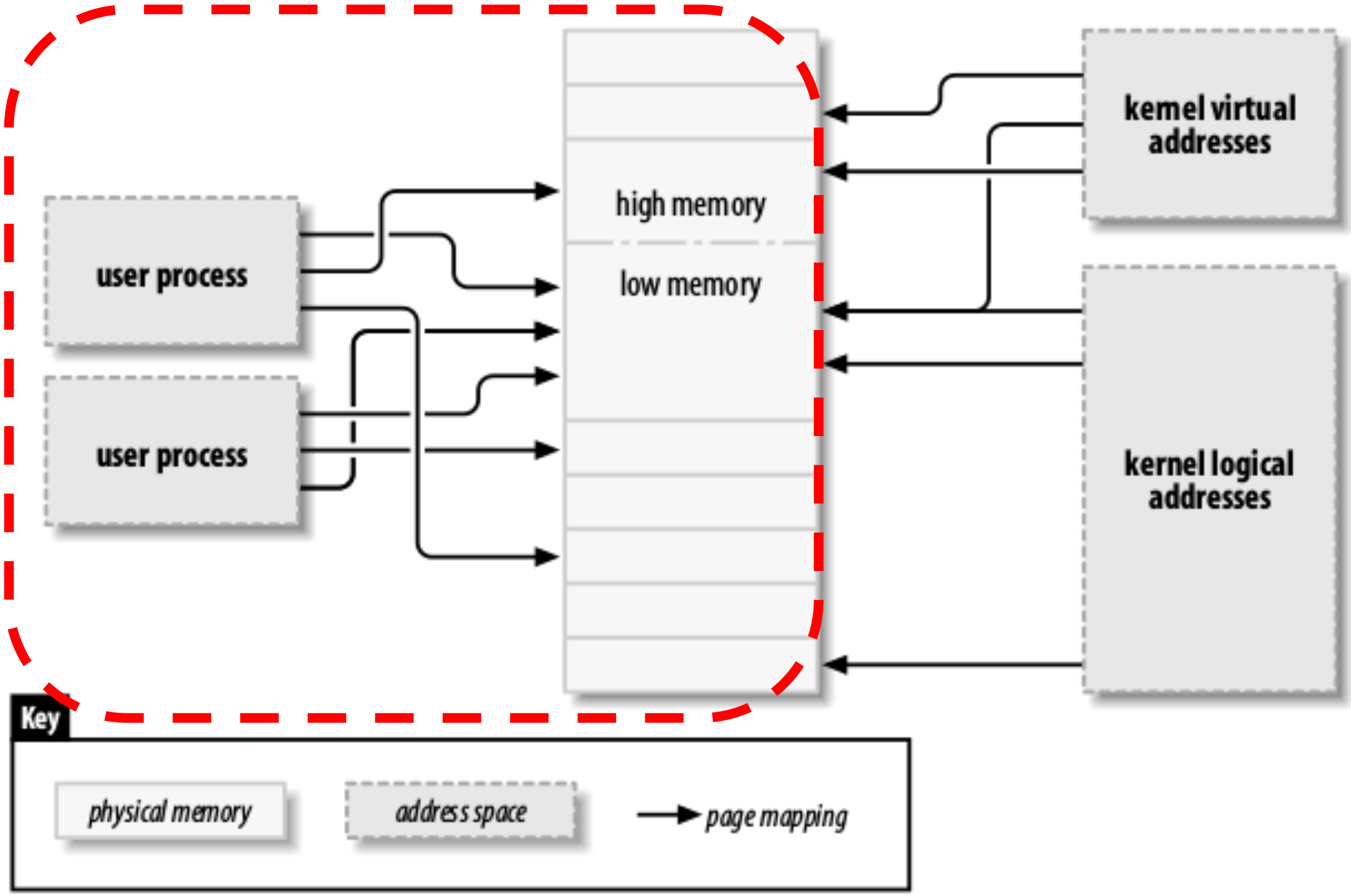
User space page table mapping

- Non-contiguous



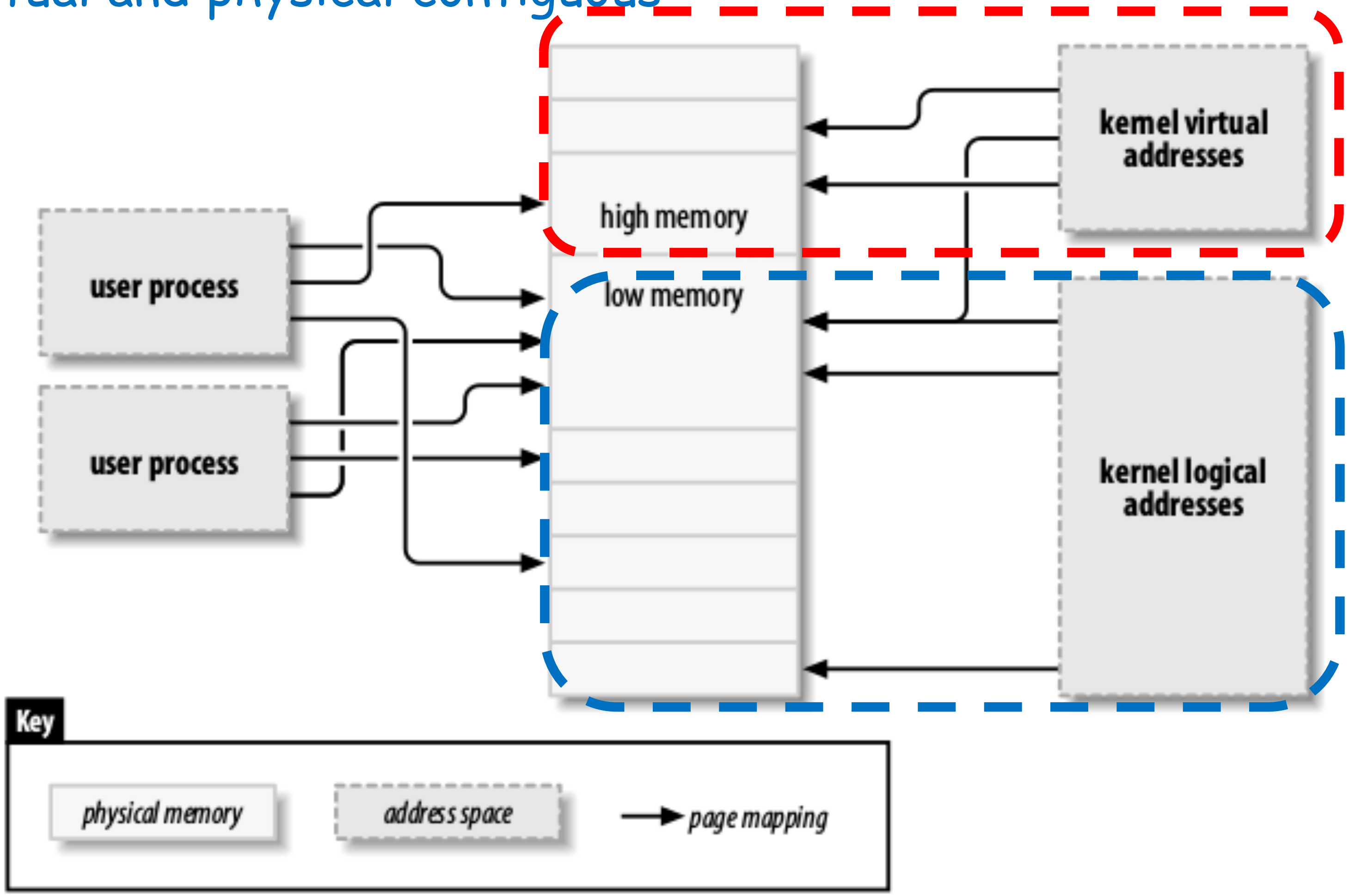
User space page table mapping

- Virtual contiguous, physical non-contiguous

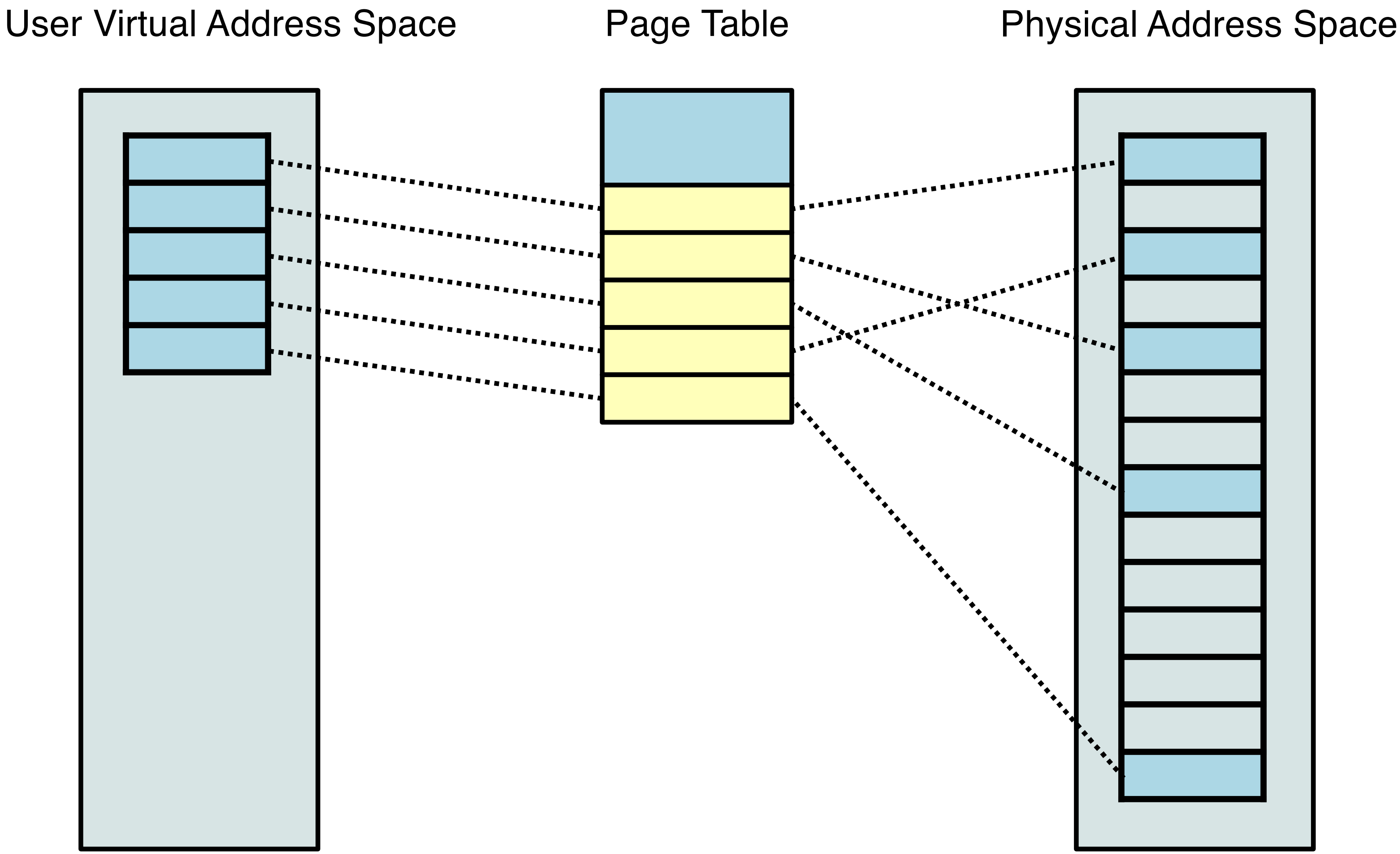


Kernel space page table mapping

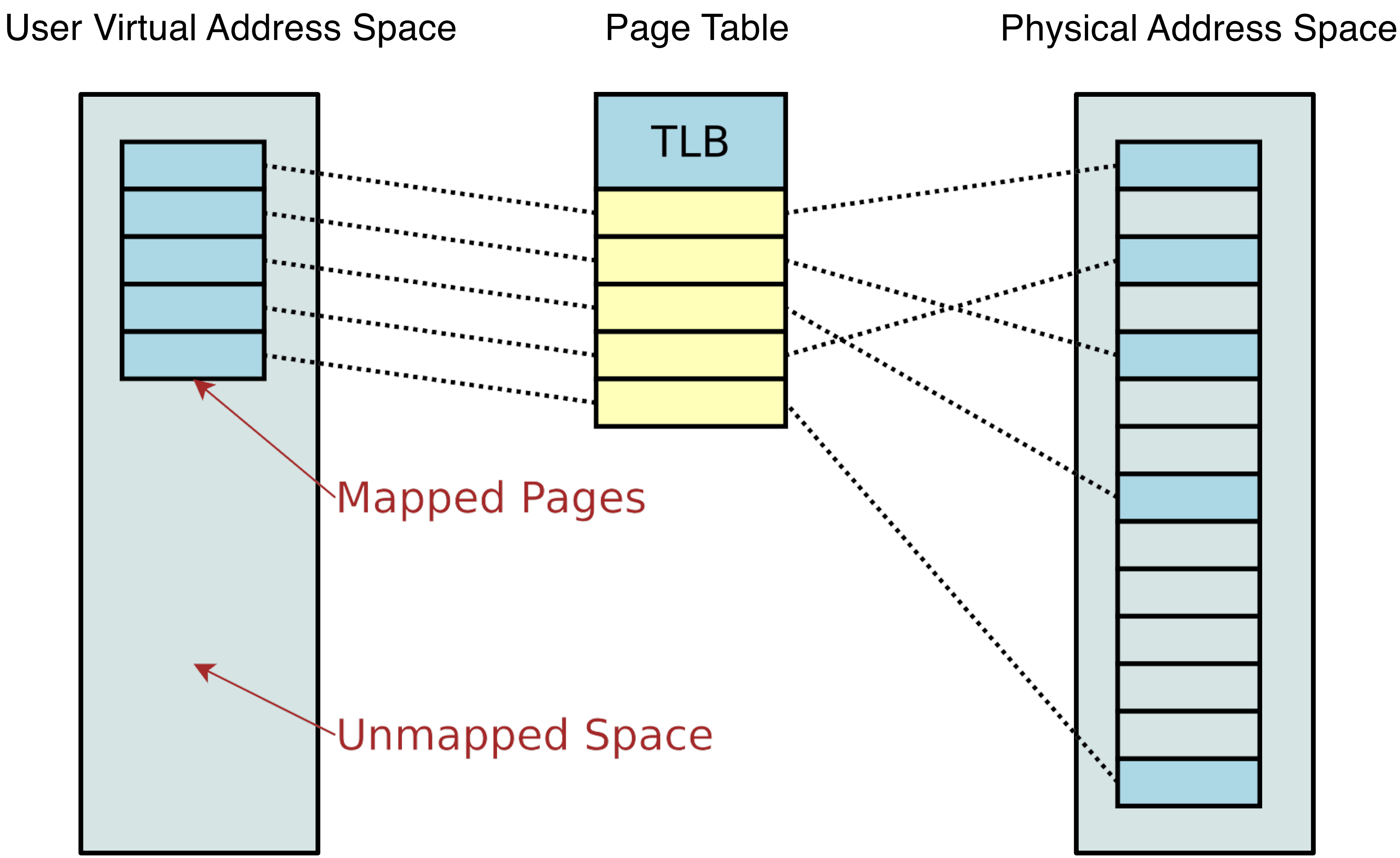
- Virtual contiguous, physical non-contiguous
- Virtual and physical contiguous



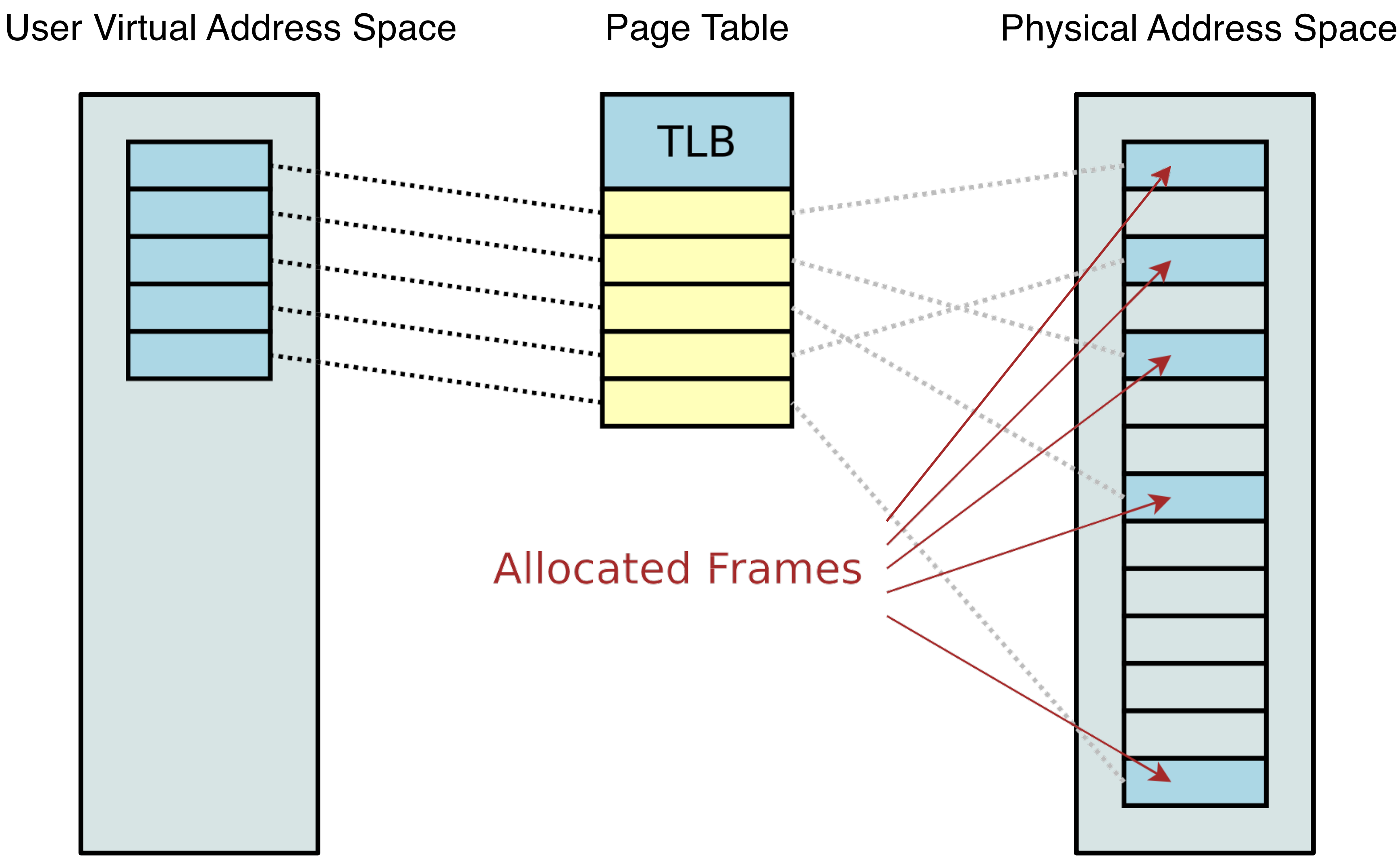
Basic Page Table Mappings



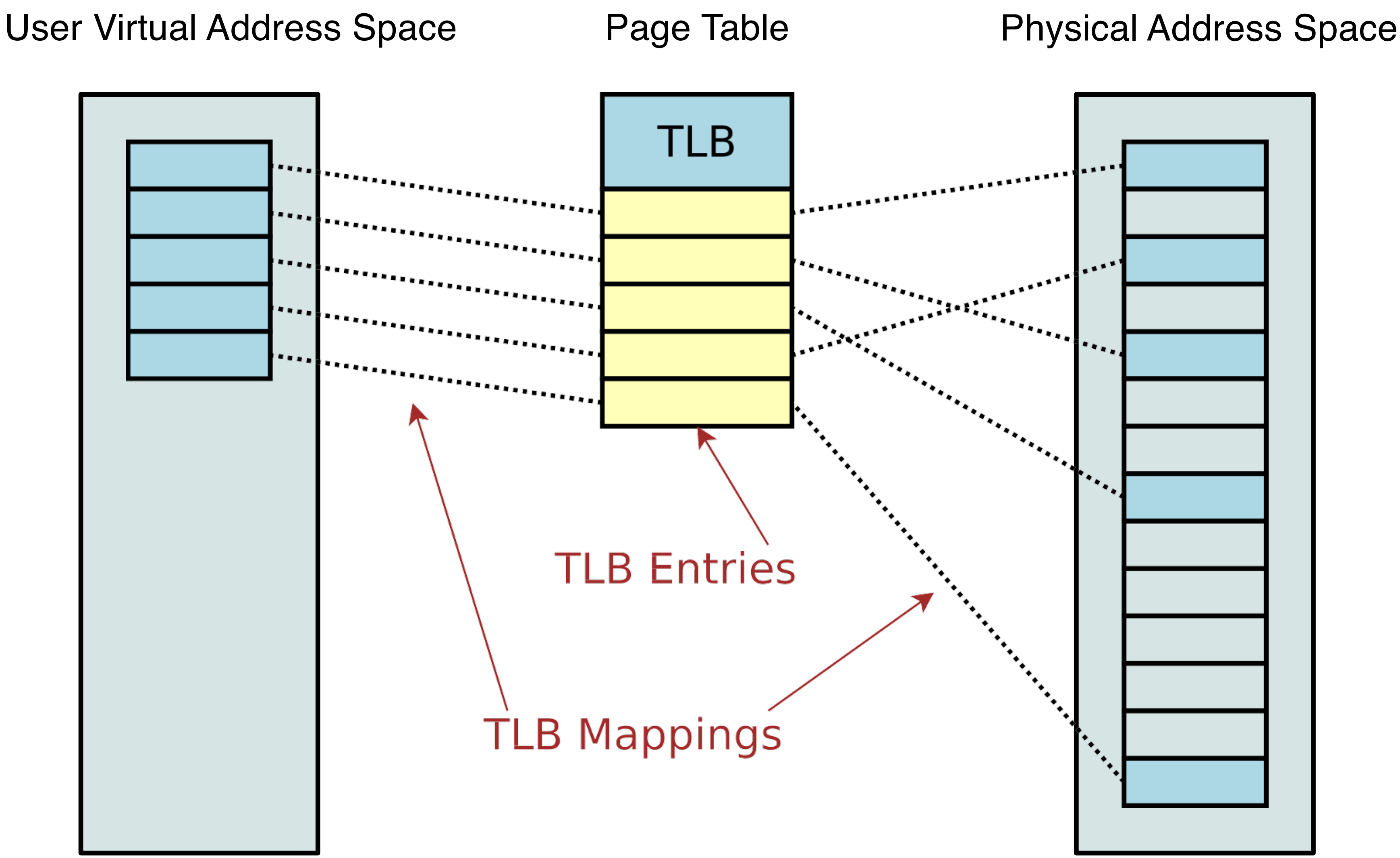
Basic Page Table Mappings



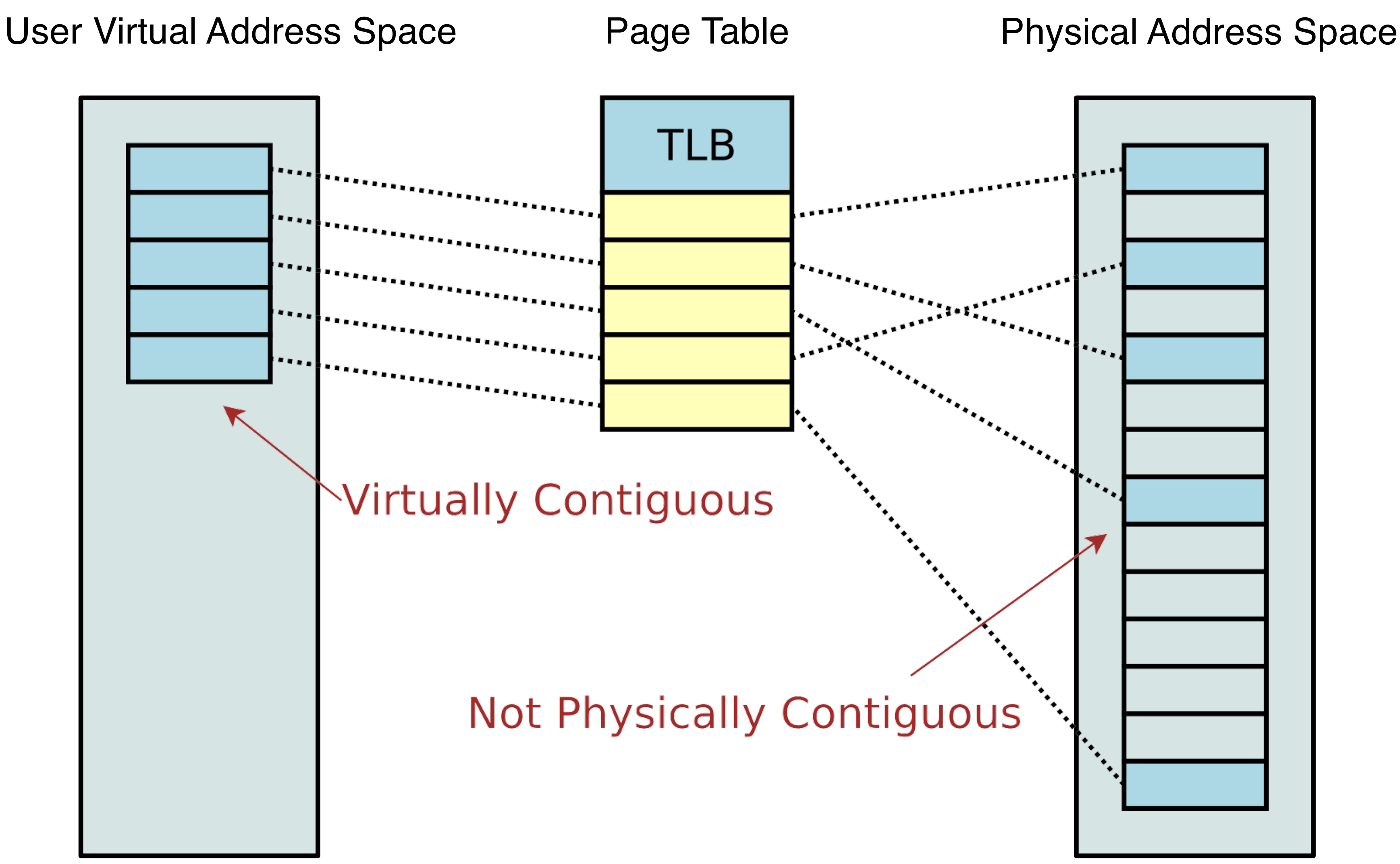
Basic Page Table Mappings



Basic Page Table Mappings



Basic Page Table Mappings



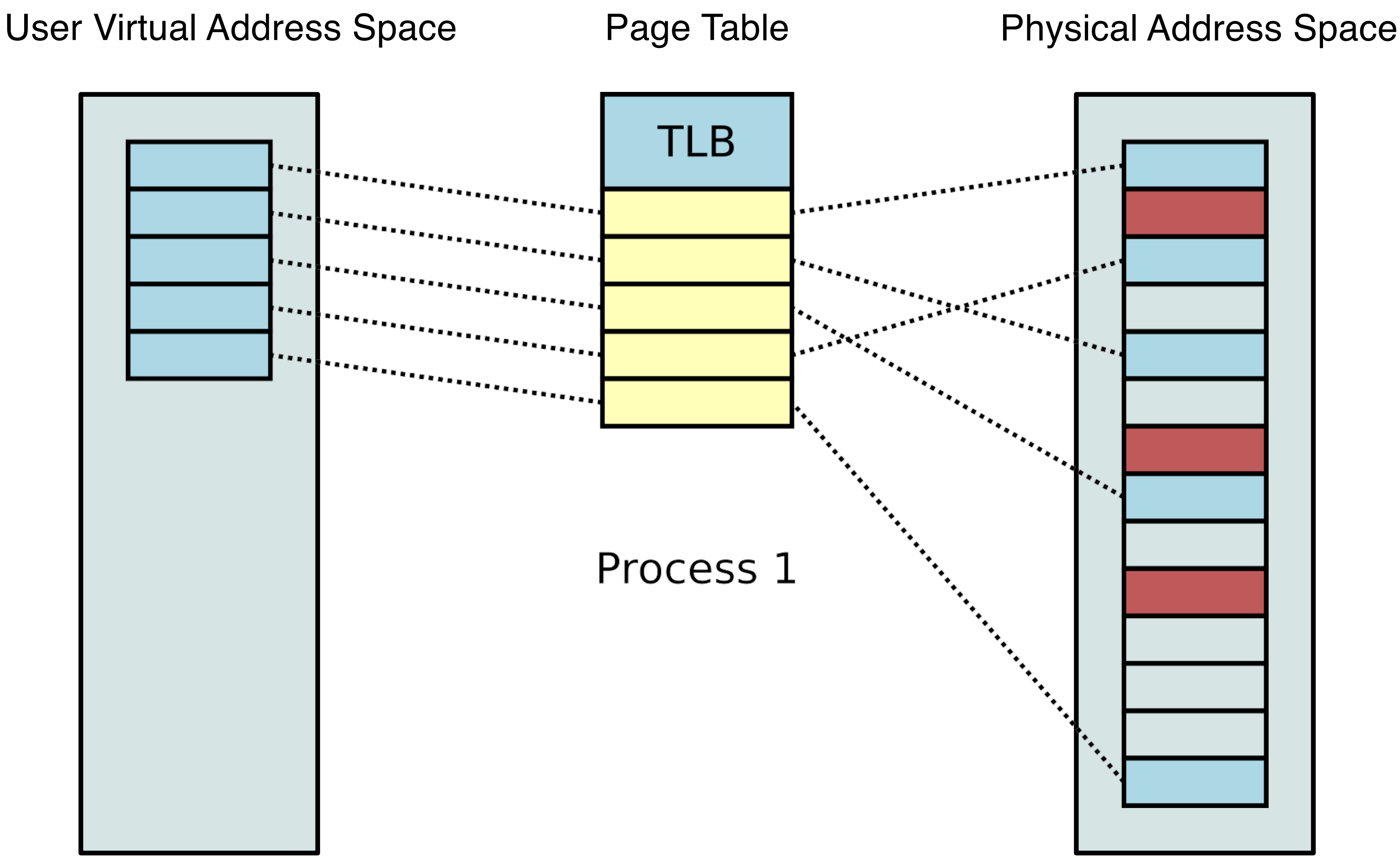
Page Table Mappings

- 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.

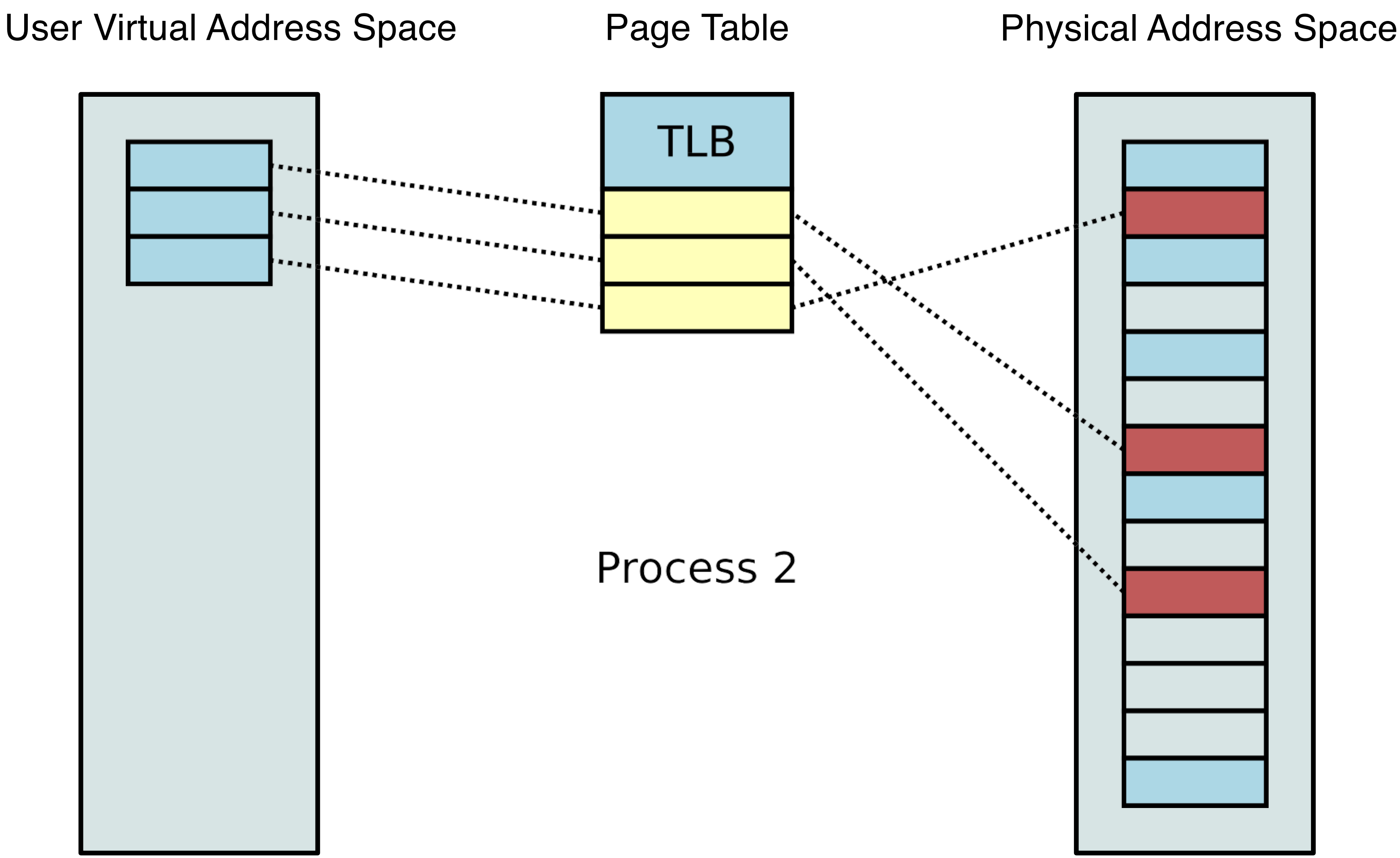
Multiple Processes

- 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



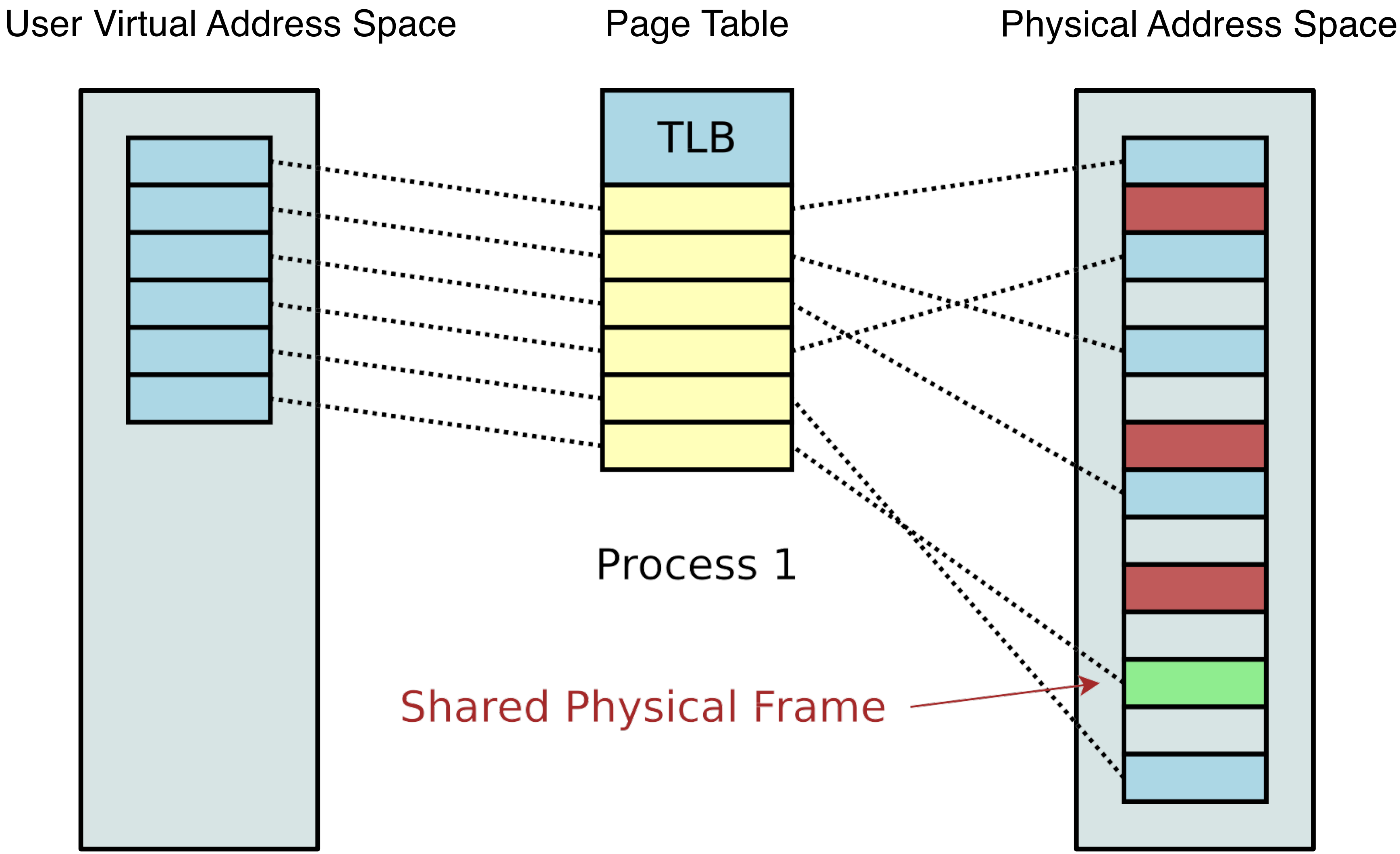
Multiple Processes - Process 2



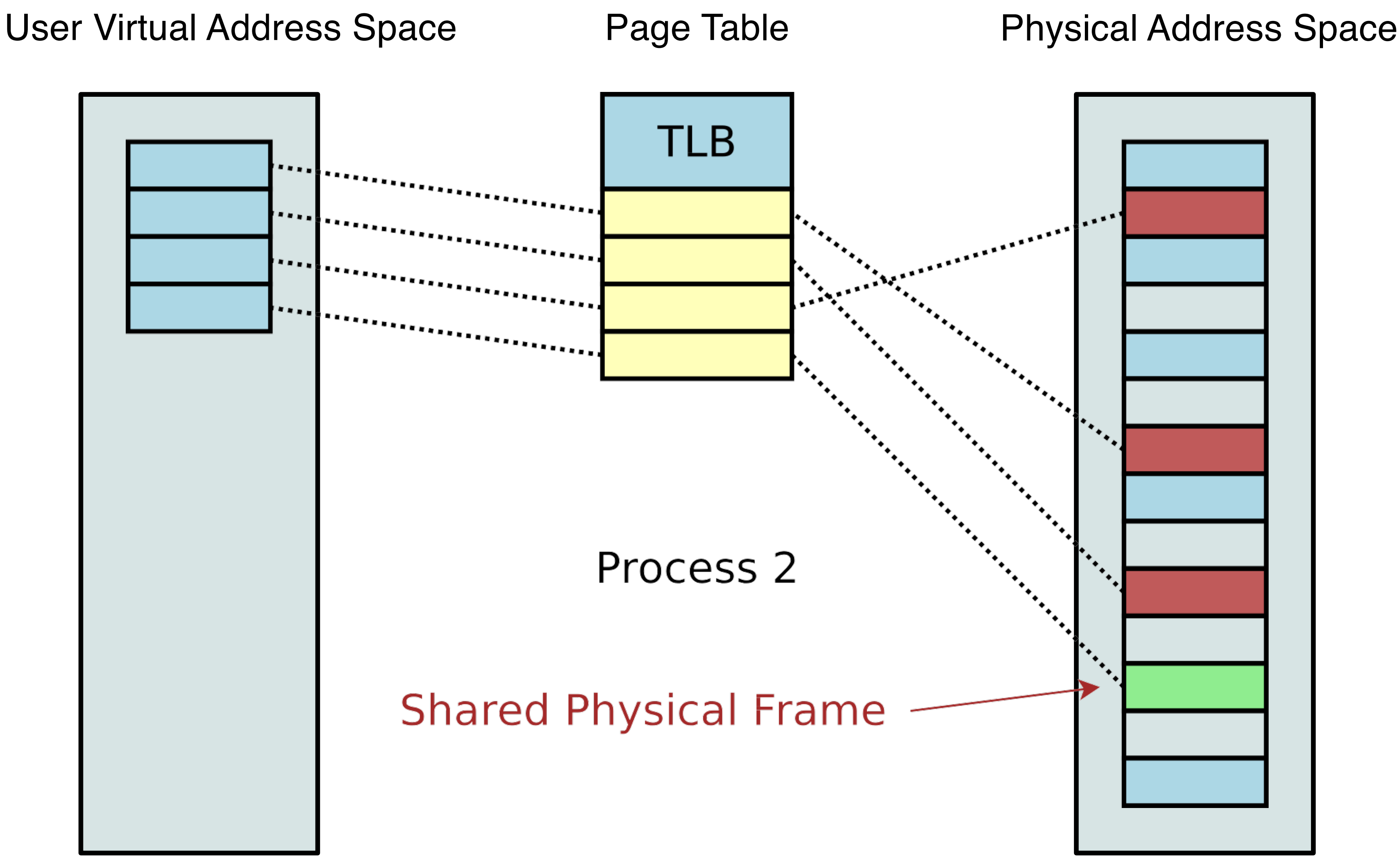
Shared Memory

- 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.

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 **frame**, or page 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, physical frame allocation never has to happen!

Lazy Allocation

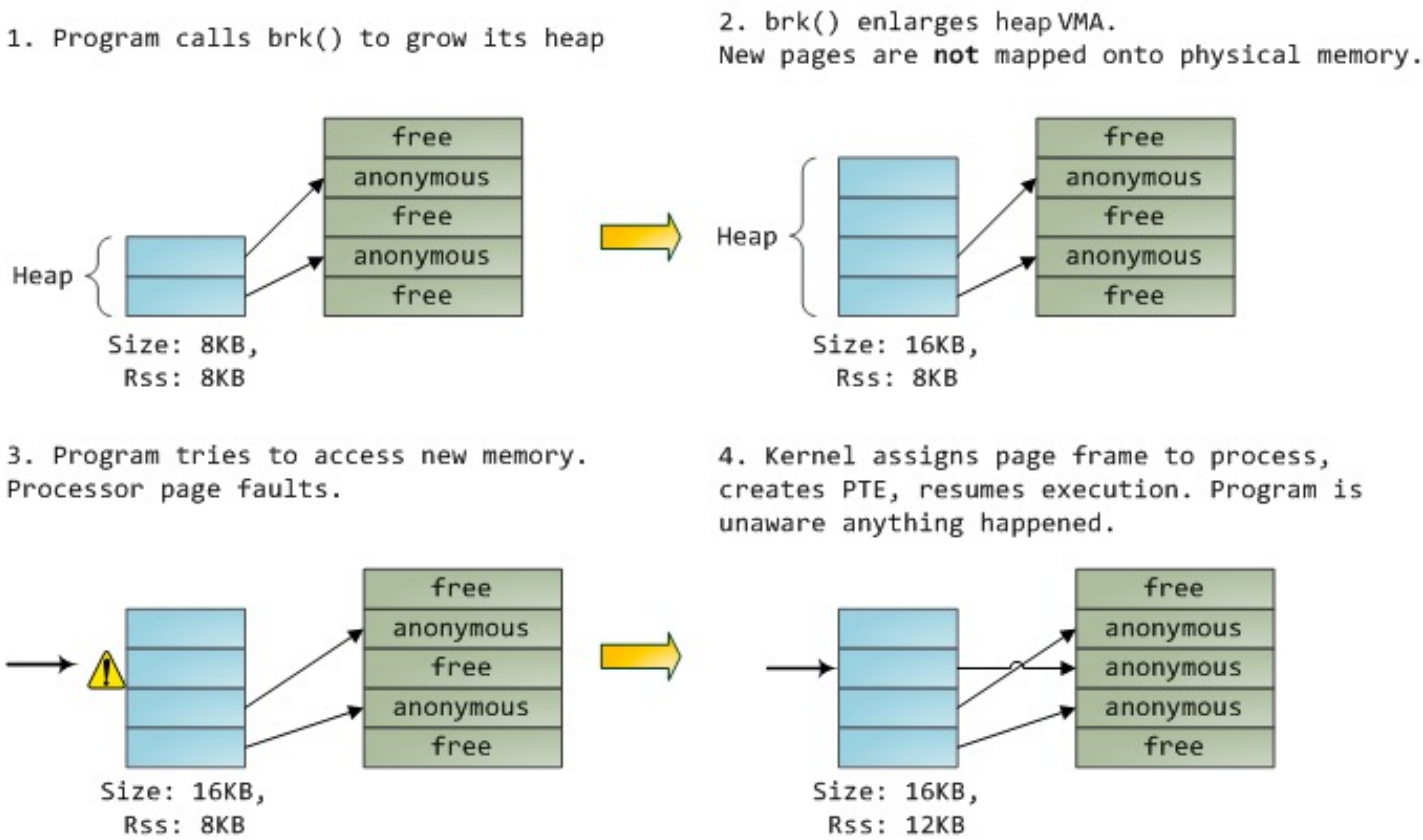
- Process
  - When memory is requested, the kernel simply creates a record of the request, and then returns (quickly) to the process, without updating the TLB.
  - When that newly-allocated memory is touched, the CPU will generate a page fault, because the CPU doesn't know about the mapping

Lazy Allocation

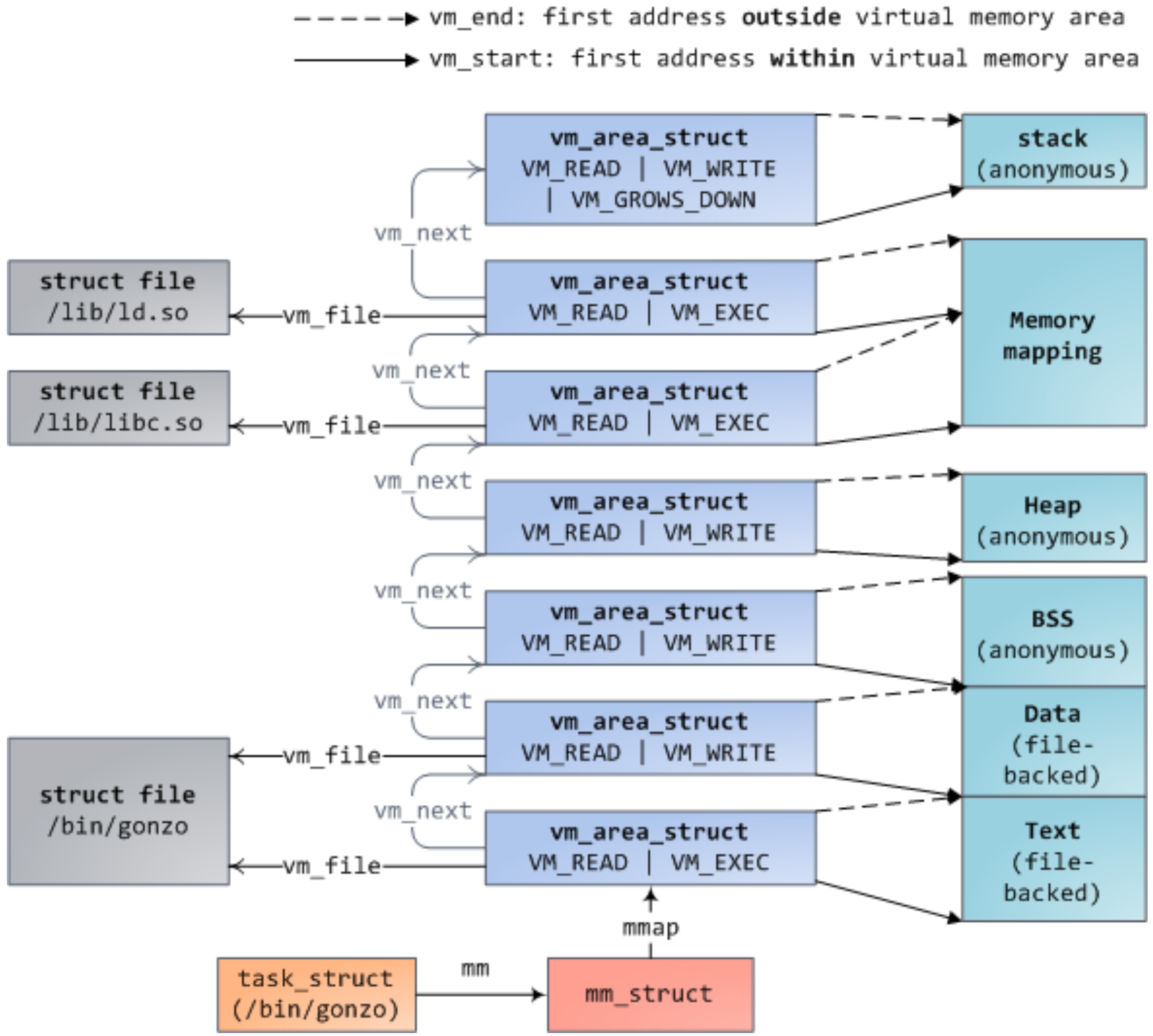
- Process (cont)
  - In the page fault handler, the kernel determines that the mapping is valid (from the kernel's point of view).
  - The kernel updates the page table with the new mapping
  - The kernel returns from the exception handler and the user space program resumes.

Page Faults

- How The Kernel Manages Your Memory
  - https://manybutfinite.com/post/how-the-kernel-manages-your-memory/



Page Faults




```
struct vm_area_struct {
        /* The first cache line has the info for VMA tree walking. */

        unsigned long vm_start;          /* Our start address within vm_mm. */
        unsigned long vm_end;            /* The first byte after our end address
                                            within vm_mm. */
```

Lazy Allocation

- In a lazy allocation case, the user space program is never aware that the page fault happened.
  - The page fault can only be detected in the time that needs to be handled.
- 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.

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