Virtual Memory and Linux

Yajin Zhou (http://yajin.org)

Zhejiang University

Acknowledgement: https://events.static.linuxfound.org/sites/events/files/slides/elc_2016_mem.pdf

Virtual Memory Details

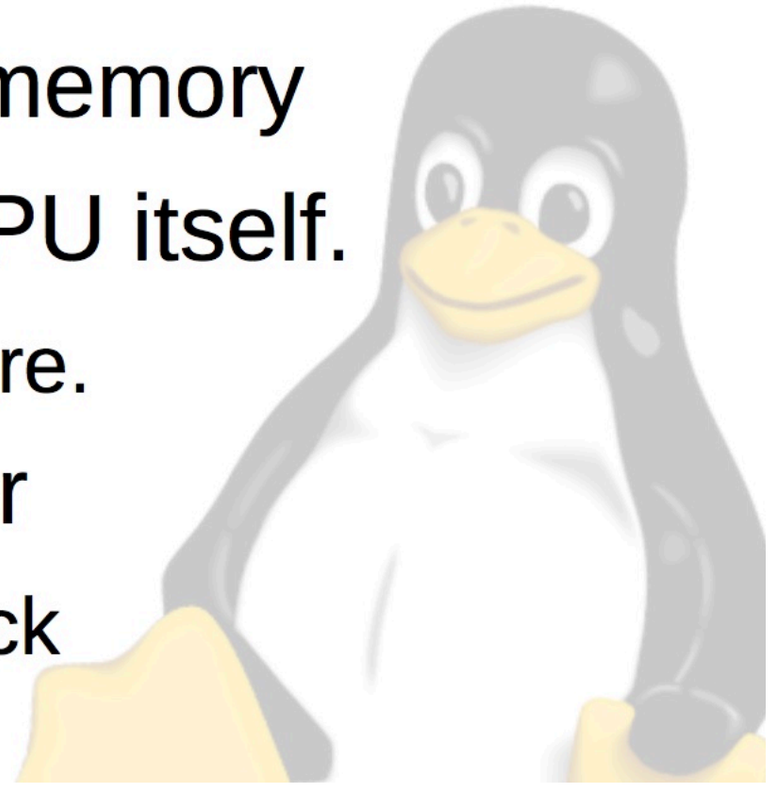
- Two address spaces
  - Physical addresses - used by hardware
    - DMA, Peripherals
- Virtual addresses
  - Addresses used by software instructions
    - load/store
    - Any instruction accesses memory

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 userspace

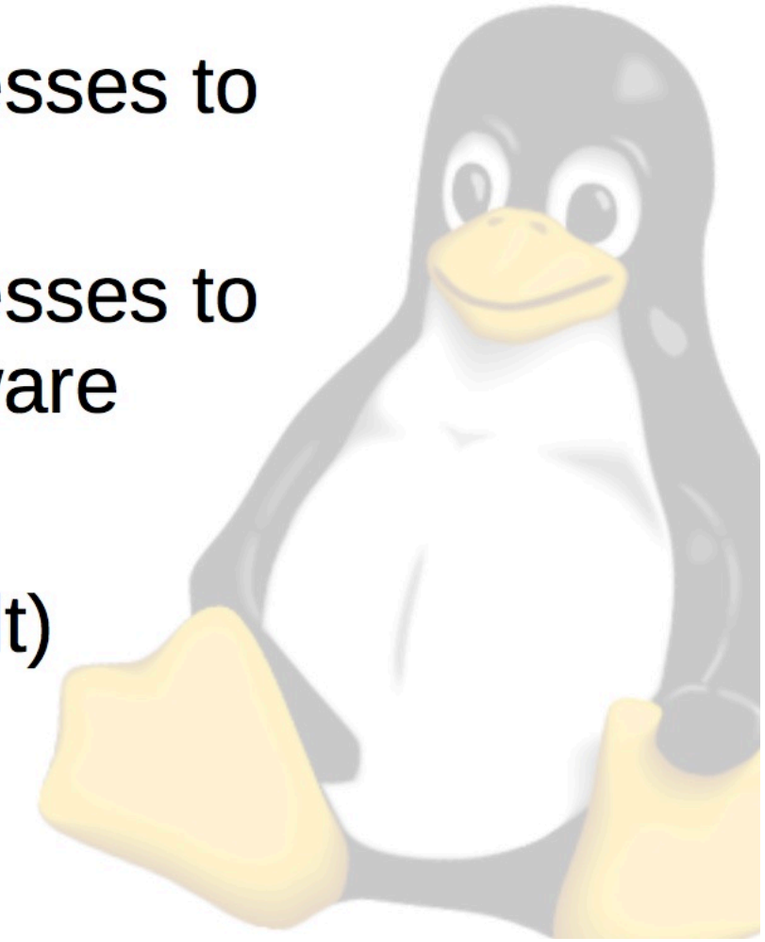
MMU

- 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



MMU

- 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

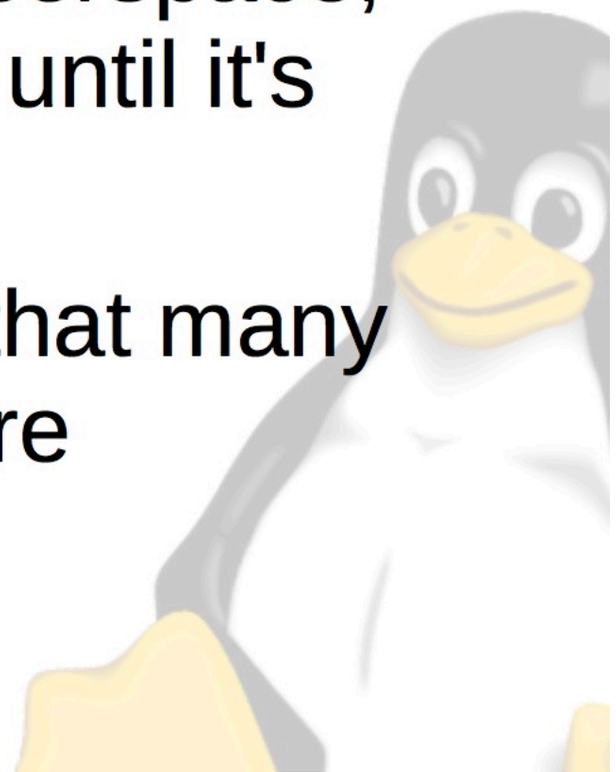


Page fault

- 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

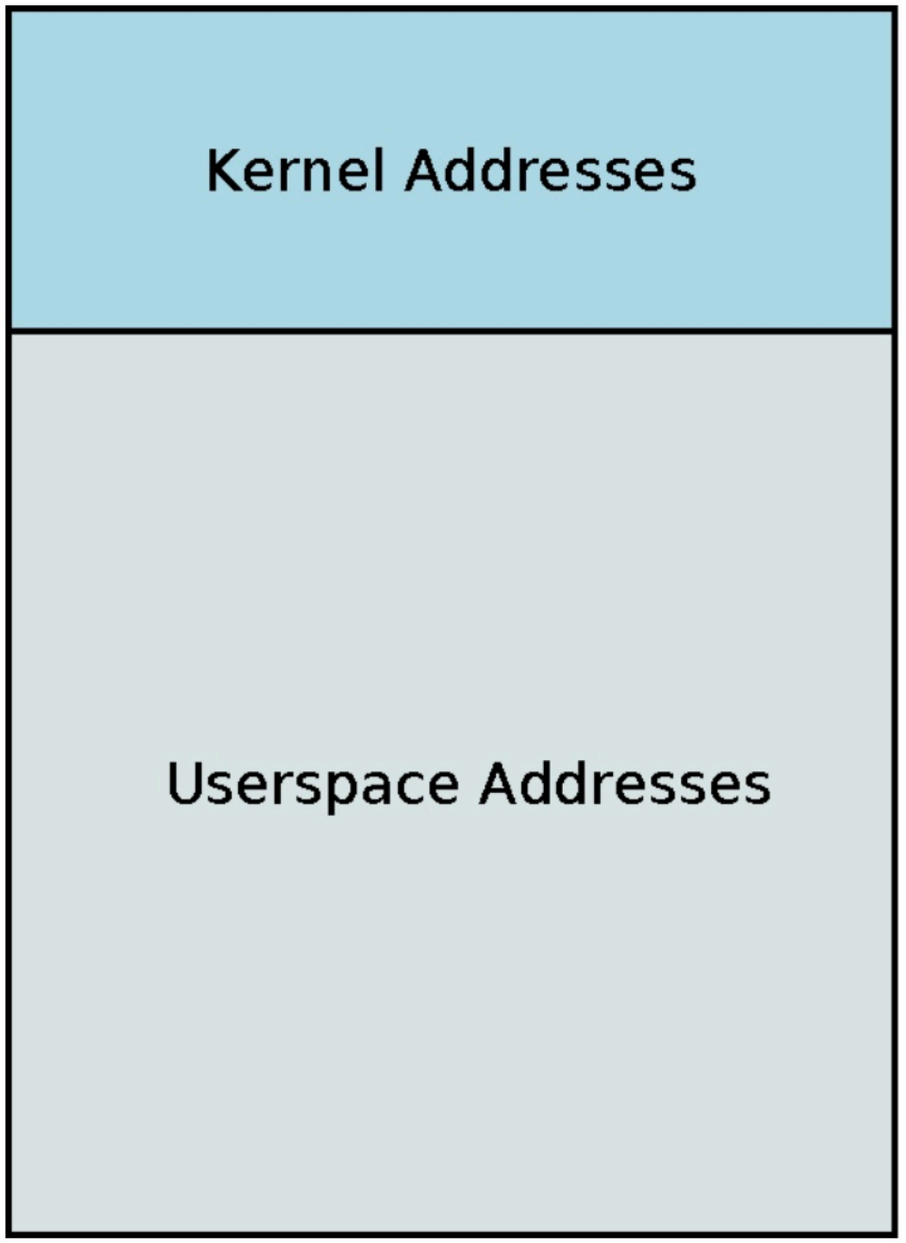
- The kernel uses lazy allocation of physical memory.
  - When memory is requested by userspace, physical memory is not allocated until it's touched.
- This is an optimization, knowing that many userspace programs allocate more RAM than they ever touch.
  - Buffers, etc.



Virtual Addresses

- In Linux, the kernel uses virtual addresses, as userspace processes do.
  - This is not true in all OS's
- 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`

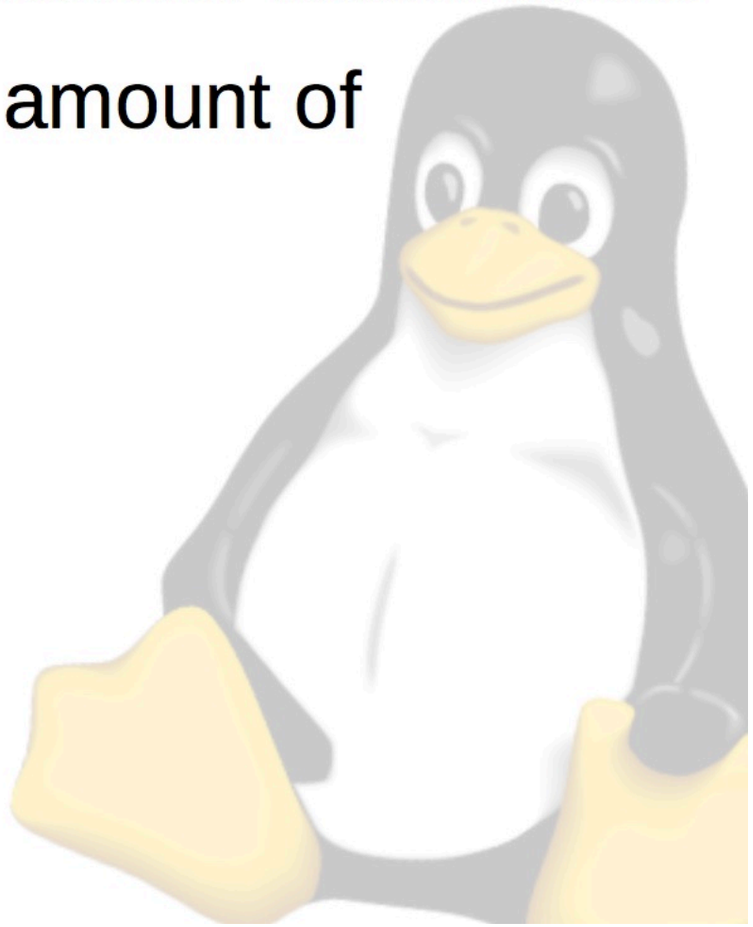
Virtual Addresses - Linux



- 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



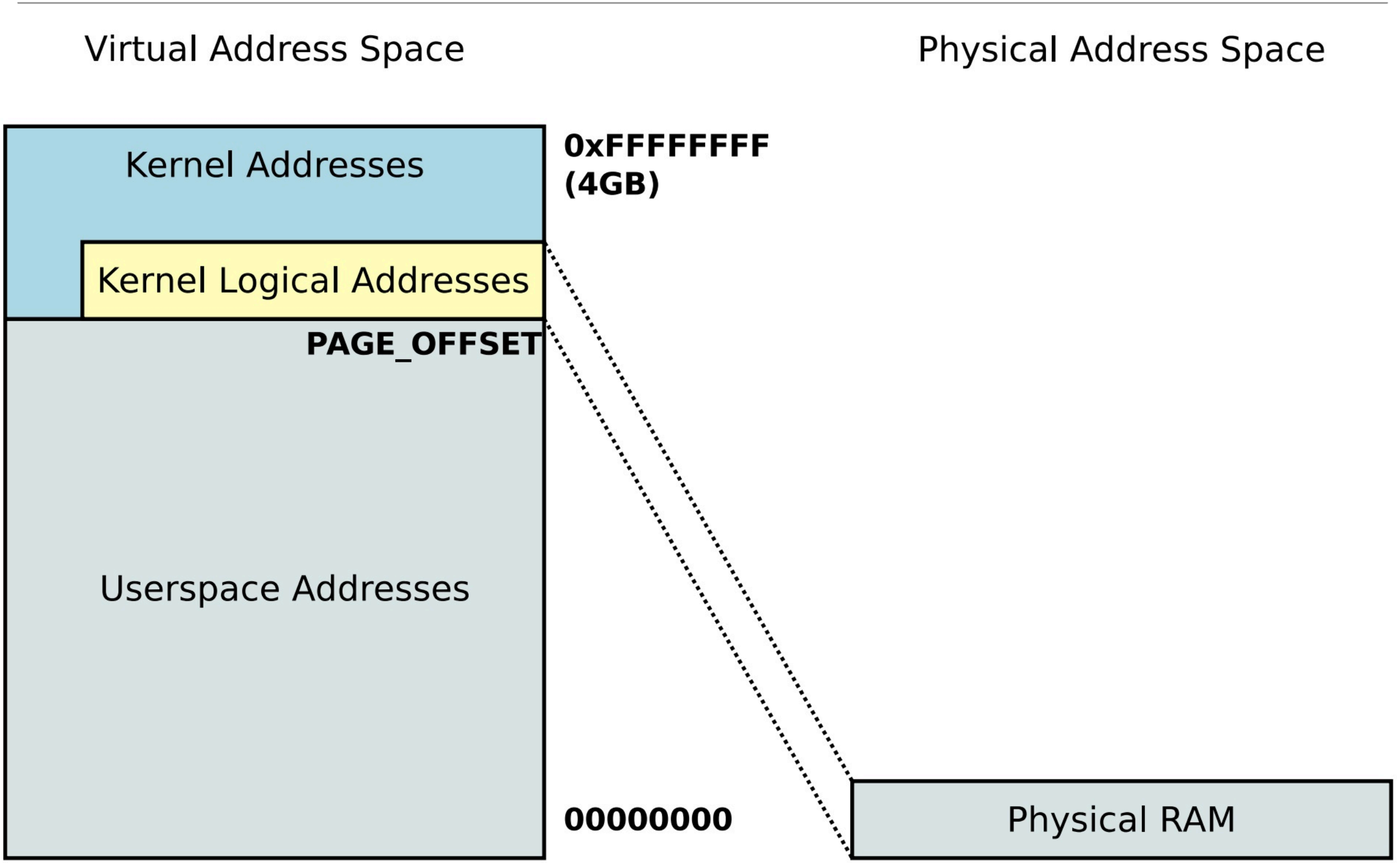
Virtual Addresses - Linux

- 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/

Kernel Logical Address

- Kernel Logical Addresses
  - Normal address space of the kernel
  - Addresses above PAGE_OFFSET
  - Virtual addresses are a fixed offset from their physical addresses.
    - Eg: Virt: 0xc0000000 → Phys: 0x00000000
  - This makes converting between physical and virtual addresses easy

Kernel Logical Address



Kernel Logical Address

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

```
__pa(x)
__va(x)
```

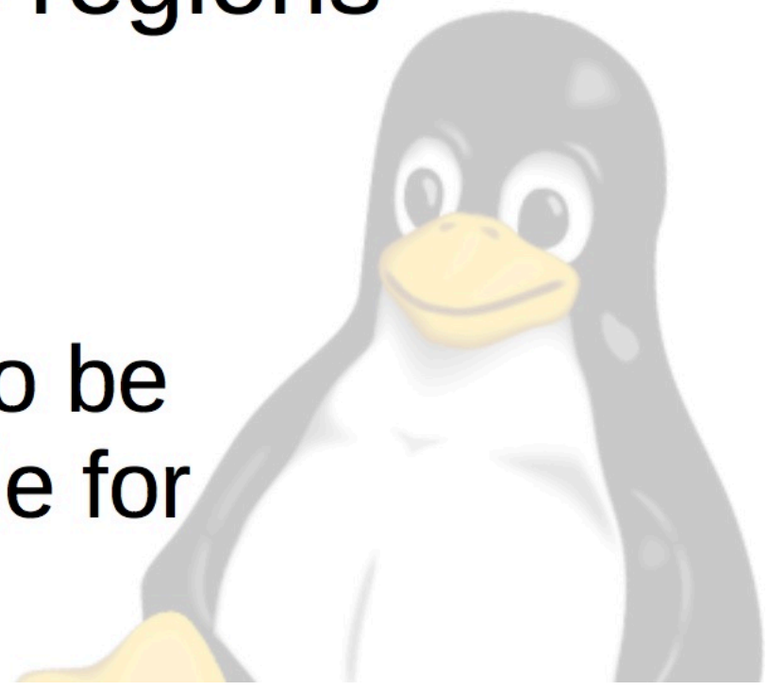
- For low-memory systems (below ~1G of RAM) Kernel Logical address space starts at PAGE_OFFSET and goes through the end of *physical* memory.

Kernel Logical Address

- 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!

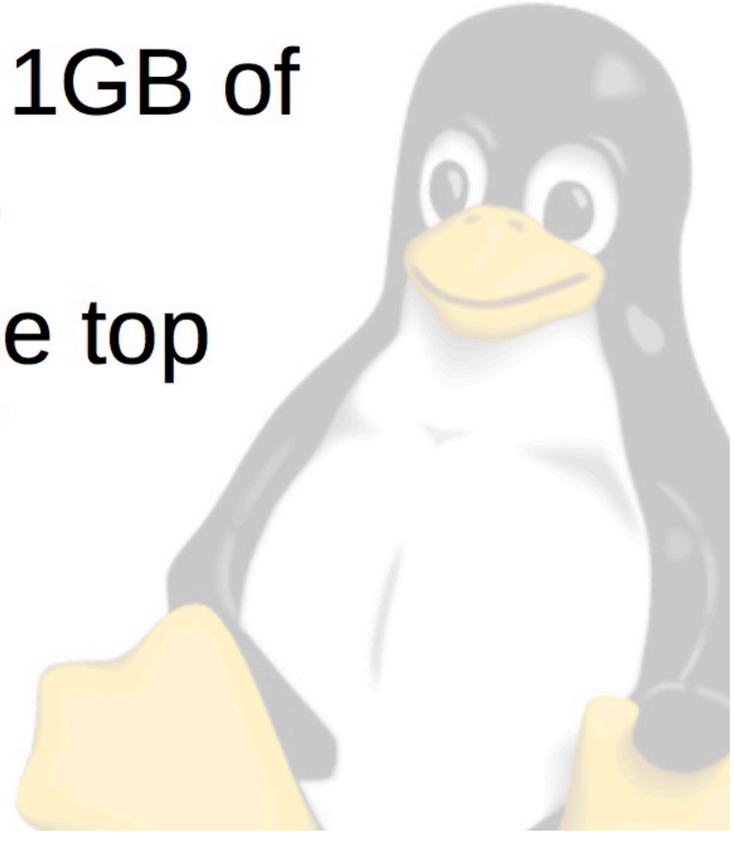
Kernel Logical Address

- Kernel Logical Addresses use a fixed 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.



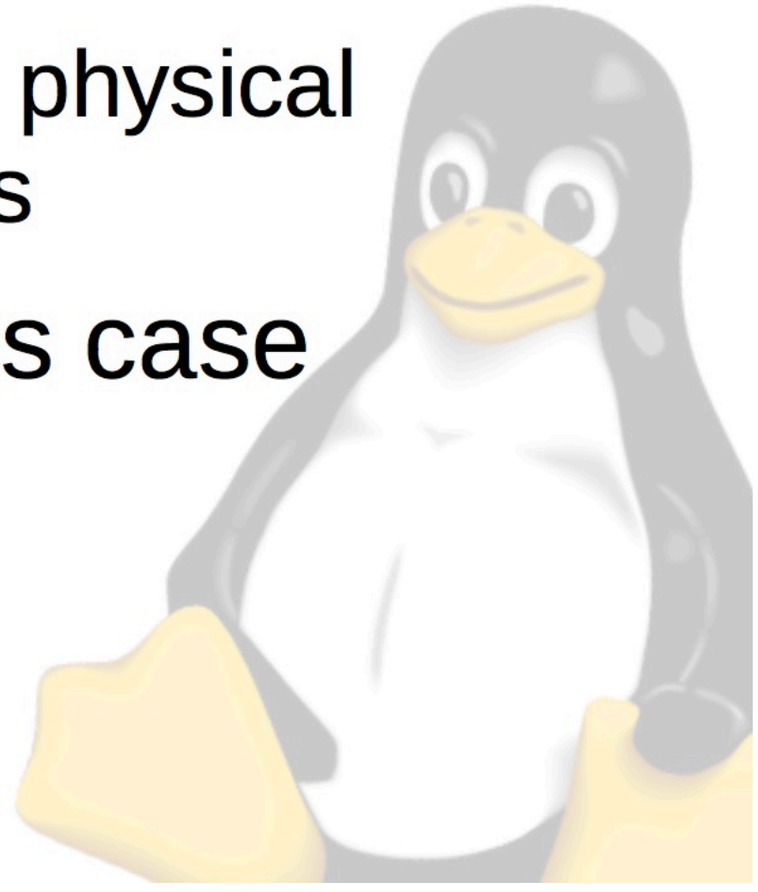
Kernel Logical Address

- For large memory systems (more than ~1GB RAM), not all of the physical RAM can be mapped into the kernel's address space.
  - Kerrnel 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
-

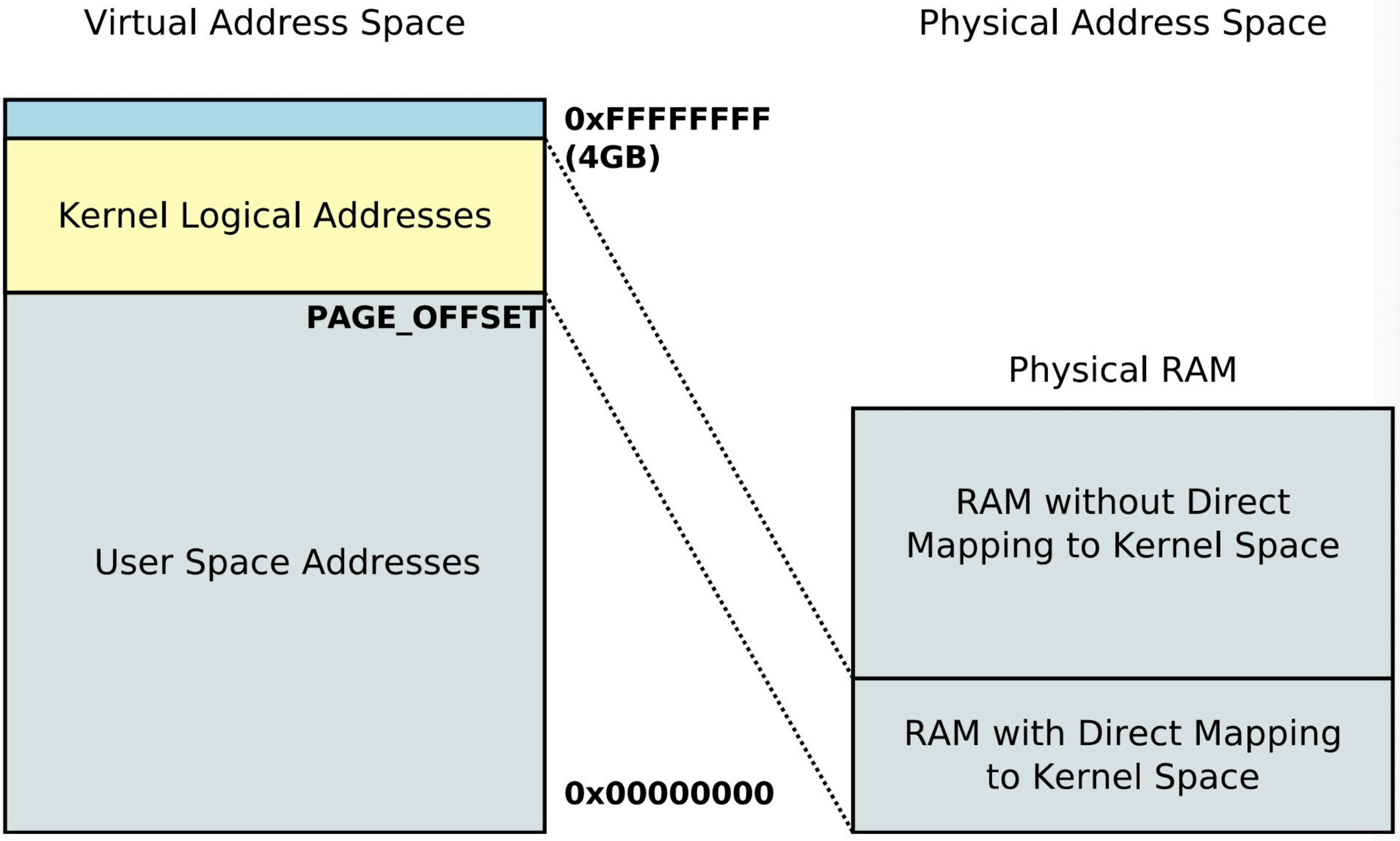


Kernel Logical Address

- Thus, in a large memory situation, only the bottom part of phyical RAM is mapped directly into kernel logical address space
  - Or rather, only the bottom part of physical RAM has a kernel logical address
- Note that on 64-bit systems, this case never happens.
  - There is always enough kernel address space to accommodate all the RAM.

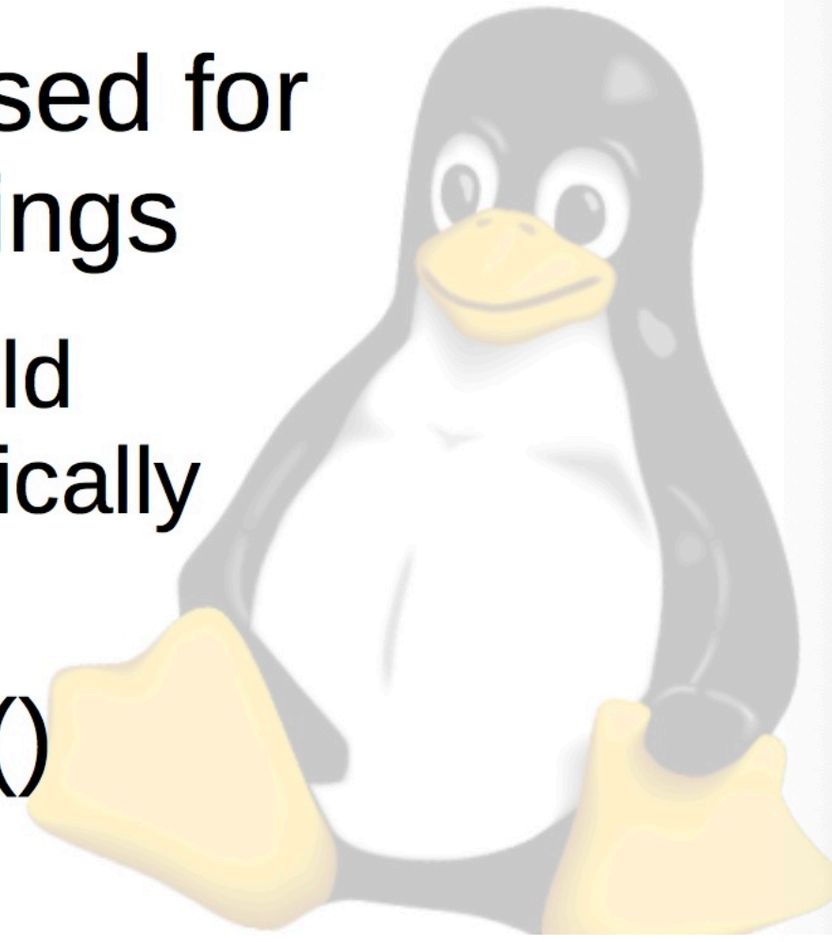


Kernel Logical Addresses (Large Mem)

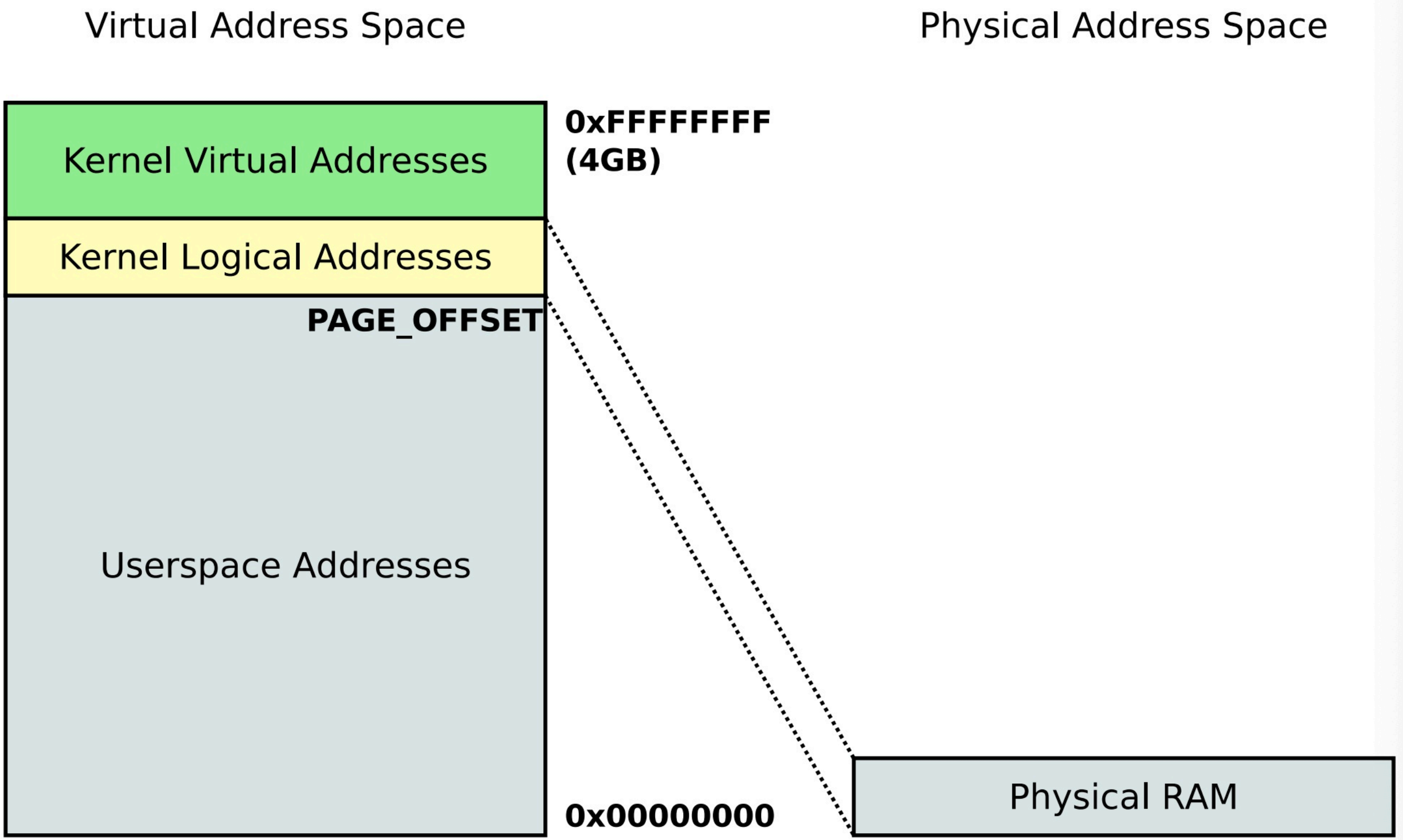


Kernel Virtual Addresses

- 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



Kernel Virtual Addresses



Kernel Virtual Addresses

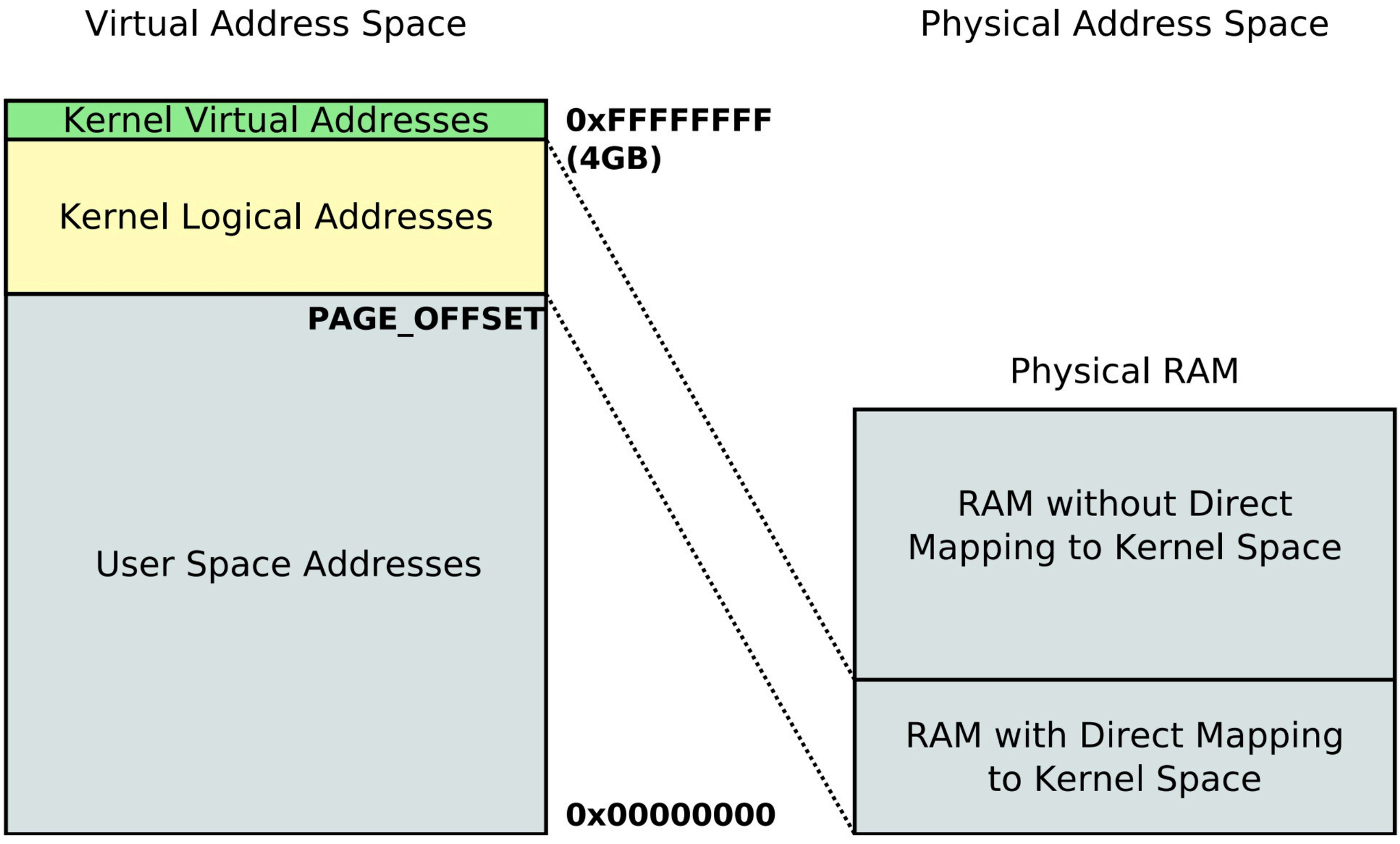
- In the small memory model, as shown, since all of RAM can be represented by logical addresses, all virtual addresses will also have logical addresses.
  - One mapping in virtual address area
  - One mapping in logical address area

double mapping

Kernel Virtual Addresses

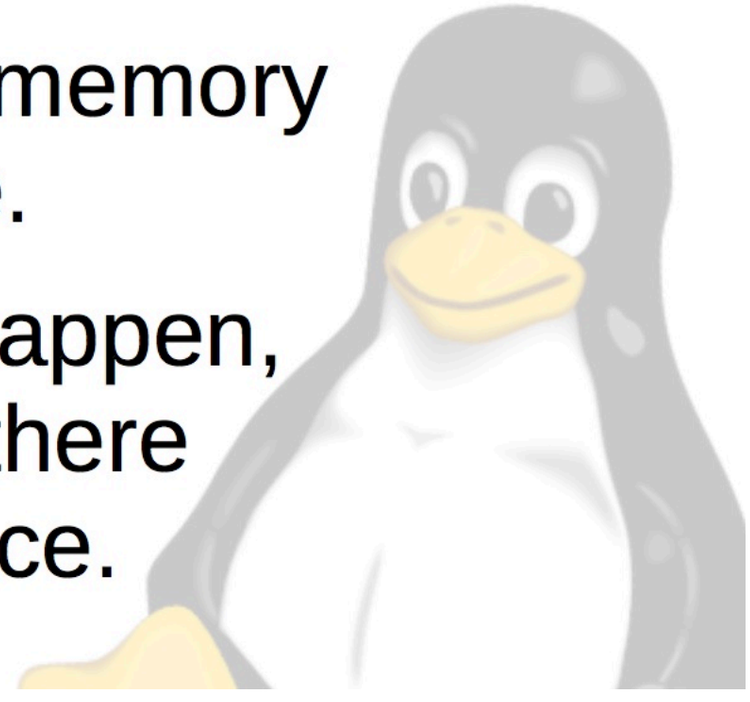
- The important difference is that memory in the kernel virtual address area (or vmalloc() area) is non-contiguous physically.
- This makes it easier to allocate, especially for large buffers
- This makes it unsuitable for DMA

Kernel Virtual Addresses (Large Mem)



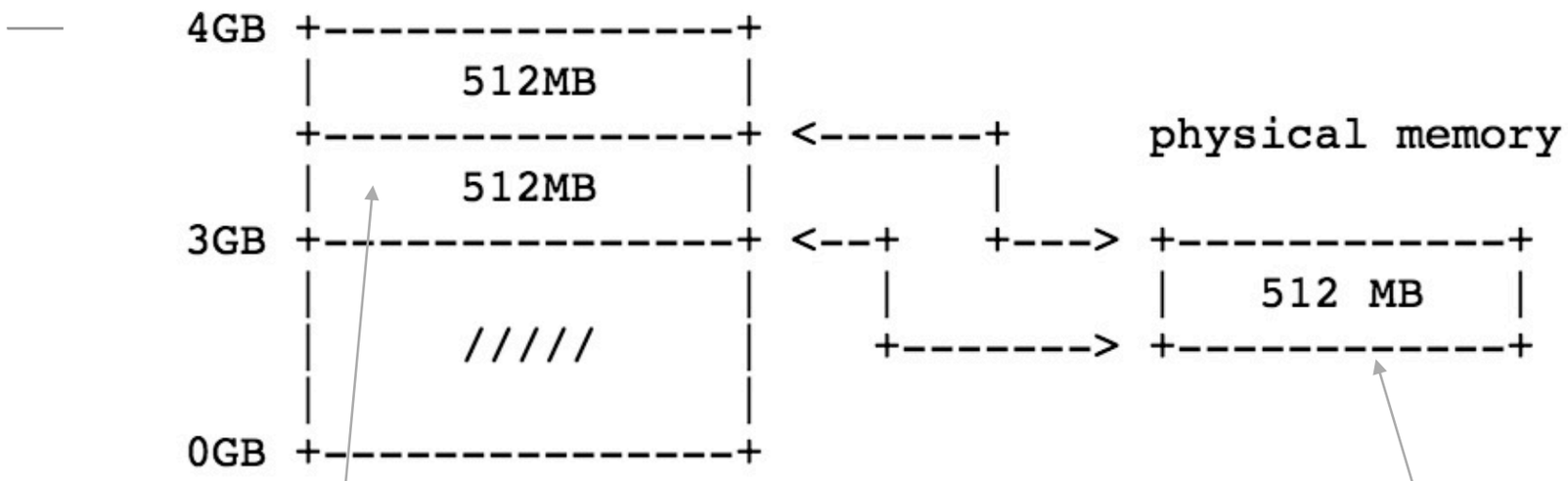
Kernel Virtual Addresses

- 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.



A little confusing?

In the early days, where a machine's physical space is much less than 1GB, it is OK, the whole physical memory is mapped to this 1GB virtual address.



Example: Physical address {x} is mapped in kernel address space, virtual address is {PAGE_OFFSET + x}, where PAGE_OFFSET is defined as 3GB in Linux kernel for the 3/1 split.

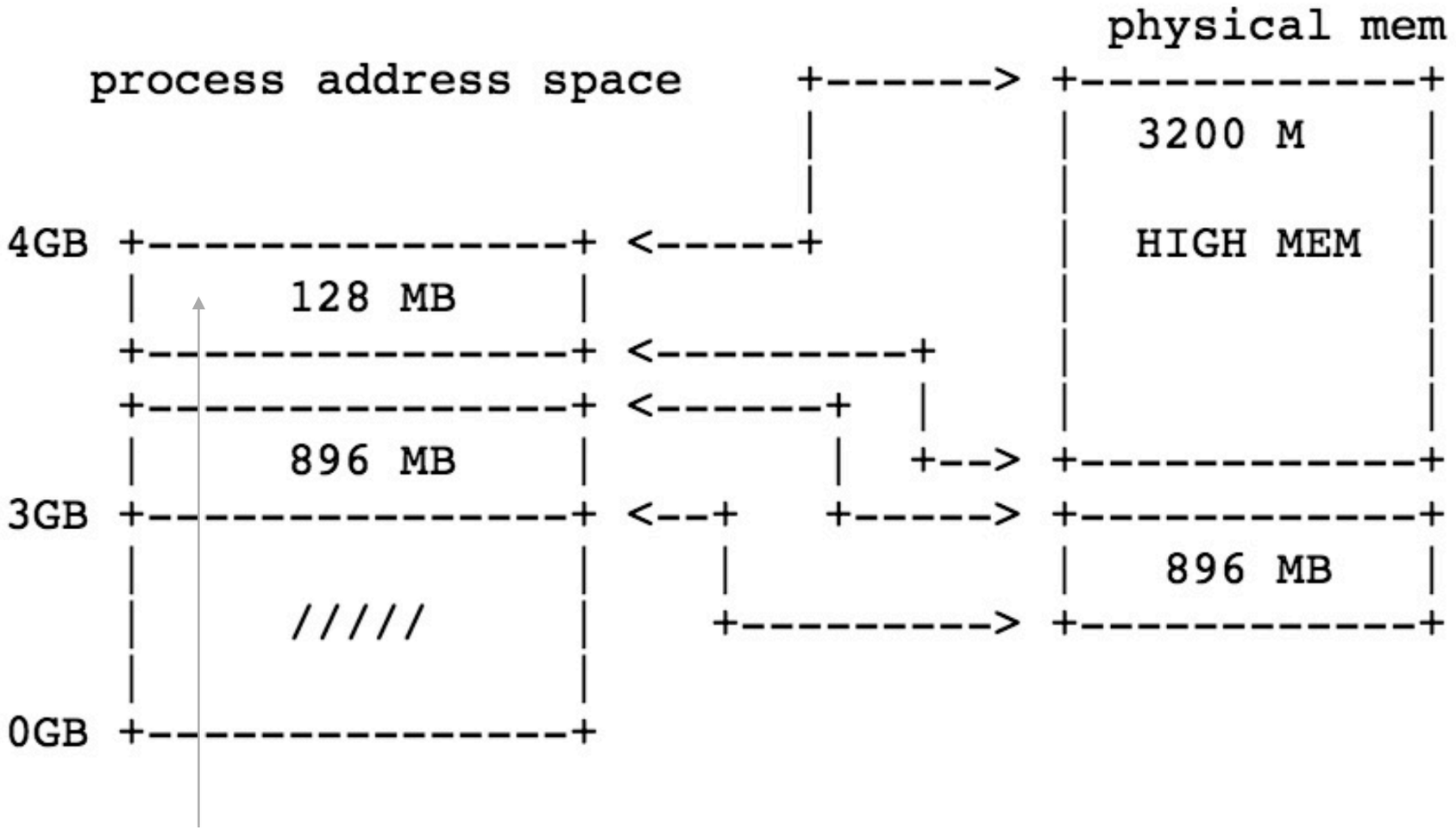
Kernel logical address

Low memory

Two general observations related to this:

- When all physical memory can be directly mapped into virtual address space, those corresponding virtual addresses are also called "kernel logical address". These logical address can often be mapped to physical address through constant offset, 3GB for example.

- The part of physical memory can be mapped into virtual space is also known as "Low Memory", conversely, those can't be mapped (say the portion over 1GB) is known as High Memory. In above case, all 512MB is Low Memory.



Kernel virtual address

Putting this in Linux context: kmalloc() will return you a chunk of virtual memory: yes, pointed by a virtual address, but more importantly, that is also a kernel logical address, meaning it has direct mapping to *continuous" physical pages.

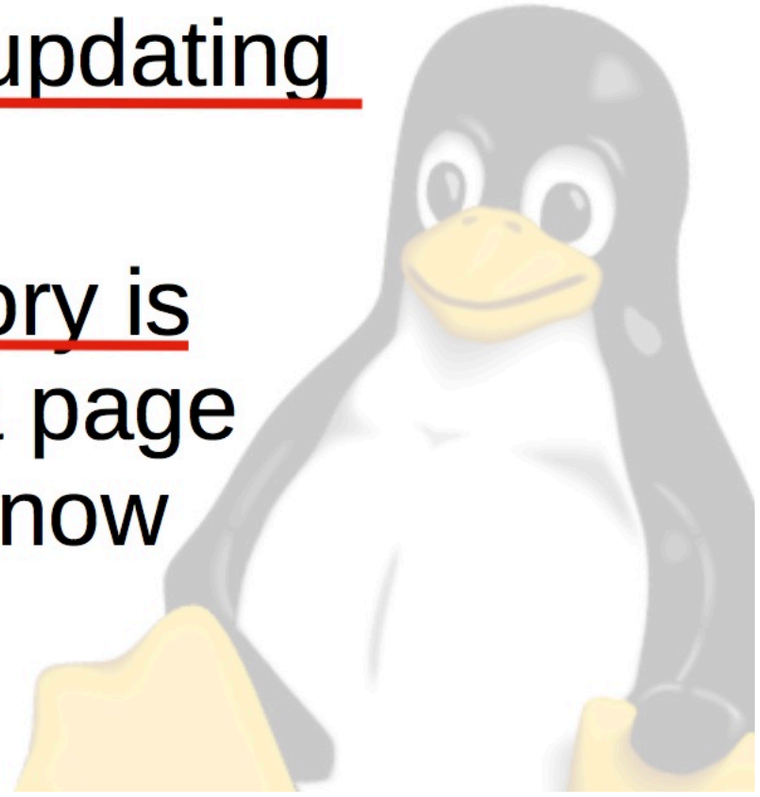
vmalloc() is another kernel call that will return you a chunk of virtual memory. However, this virtual memory is only continous on virtual space, it may not be continous on physical space. Also, the actual mapped physical pages can not only come from Low Memory, but also can come from High Memory, especially when you are asking for a large chunk of it.

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

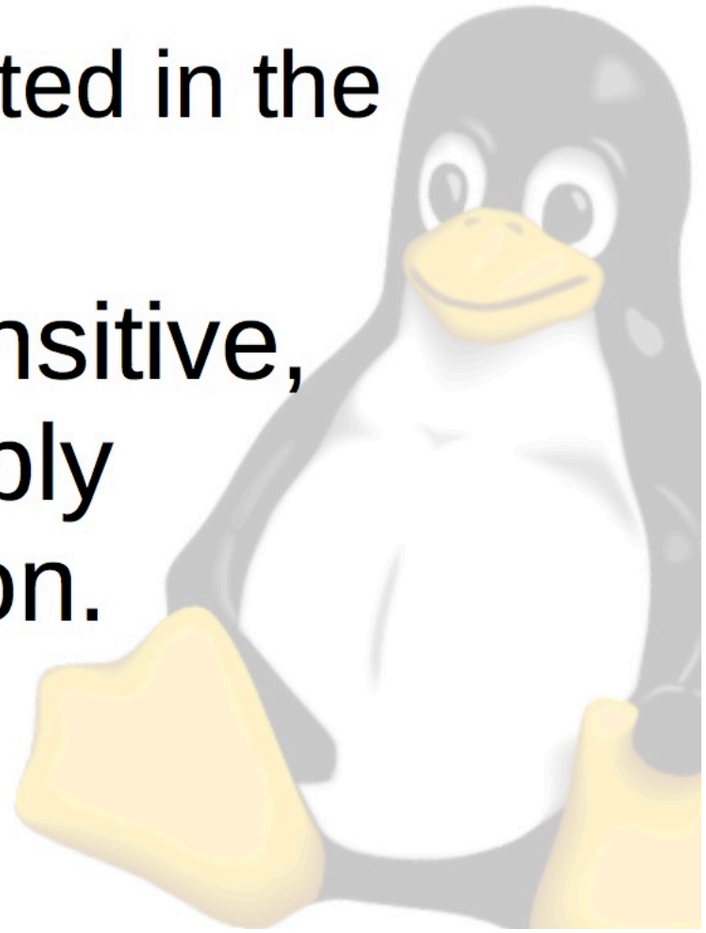


Lazy Allocation

- 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.



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

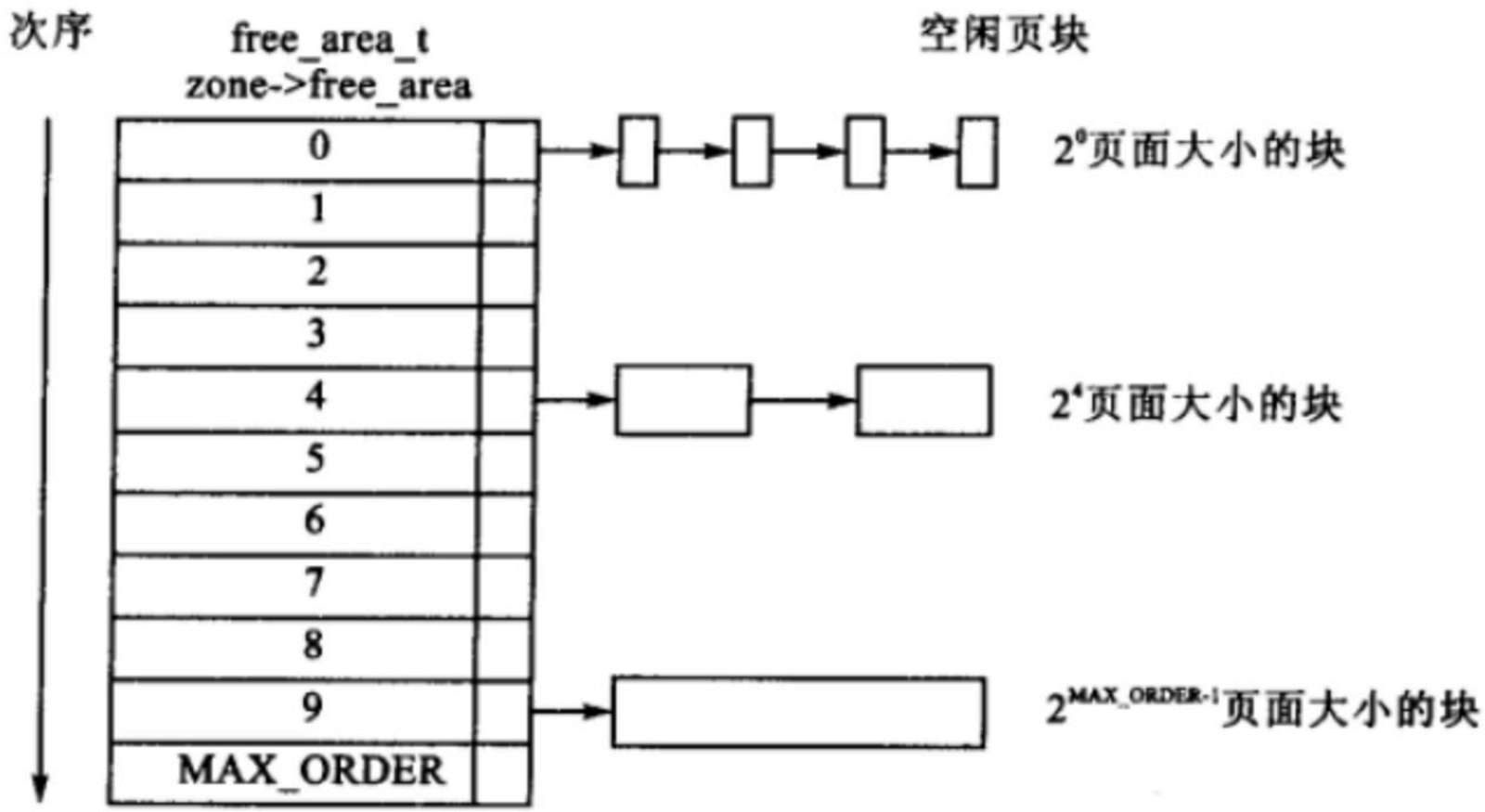
- When the frame is needed again, the CPU will generate a page fault (because the address is not in the TLB)
- The kernel can then, at page fault time:
  - Put the process to sleep
  - Copy the frame from the disk into an unused frame in RAM
  - Fix the page table entry
  - Wake the process

Swapping

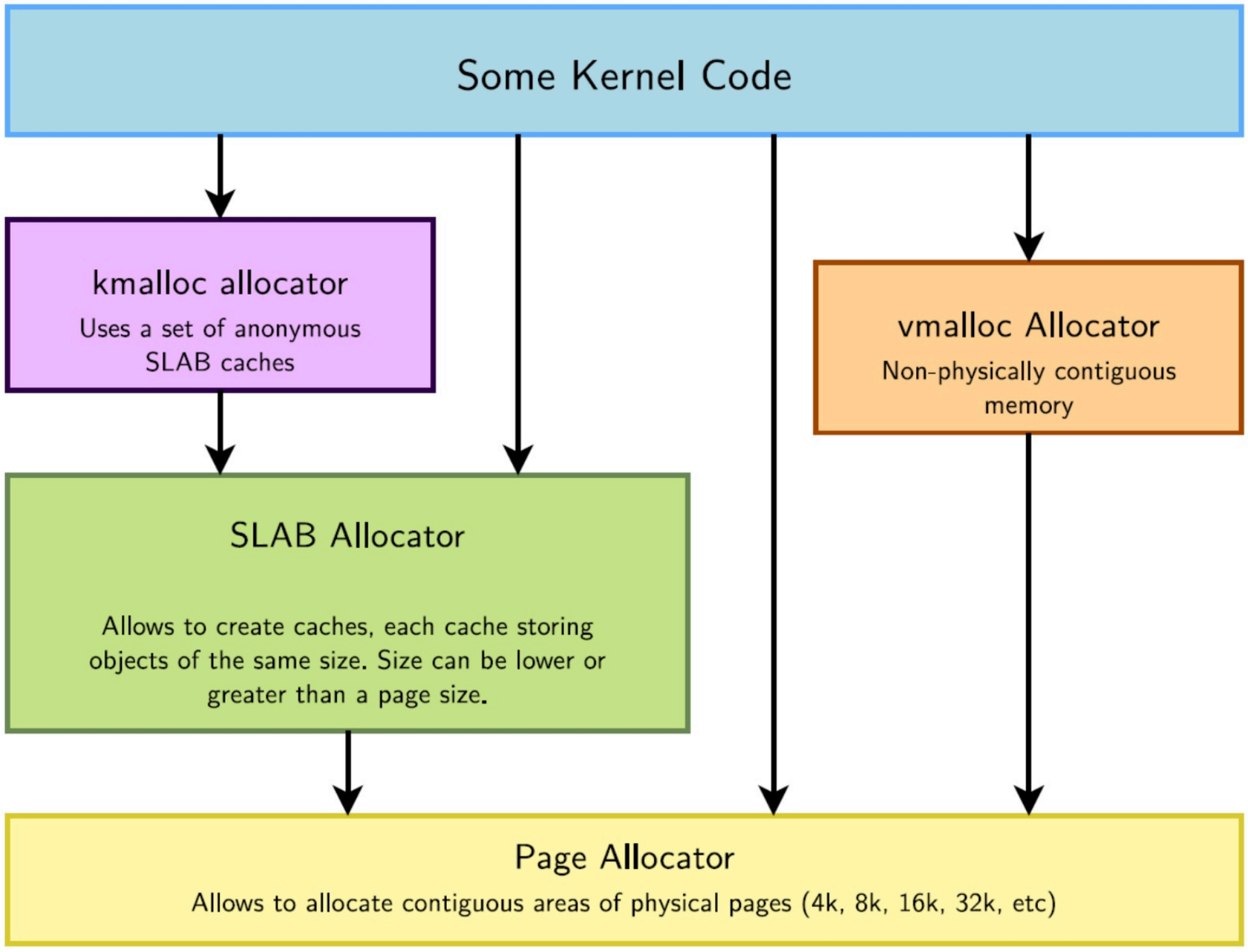
- Note that when the page is restored to RAM, it's not necessarily restored to the same physical frame where it originally was located.
- The MMU will use the same virtual address though, so the user space program will not know the difference
  - *This is why user space memory cannot typically be used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

Buddy System in Linux

- MAX_ORDER: 11 on x86-32, 2048 pages

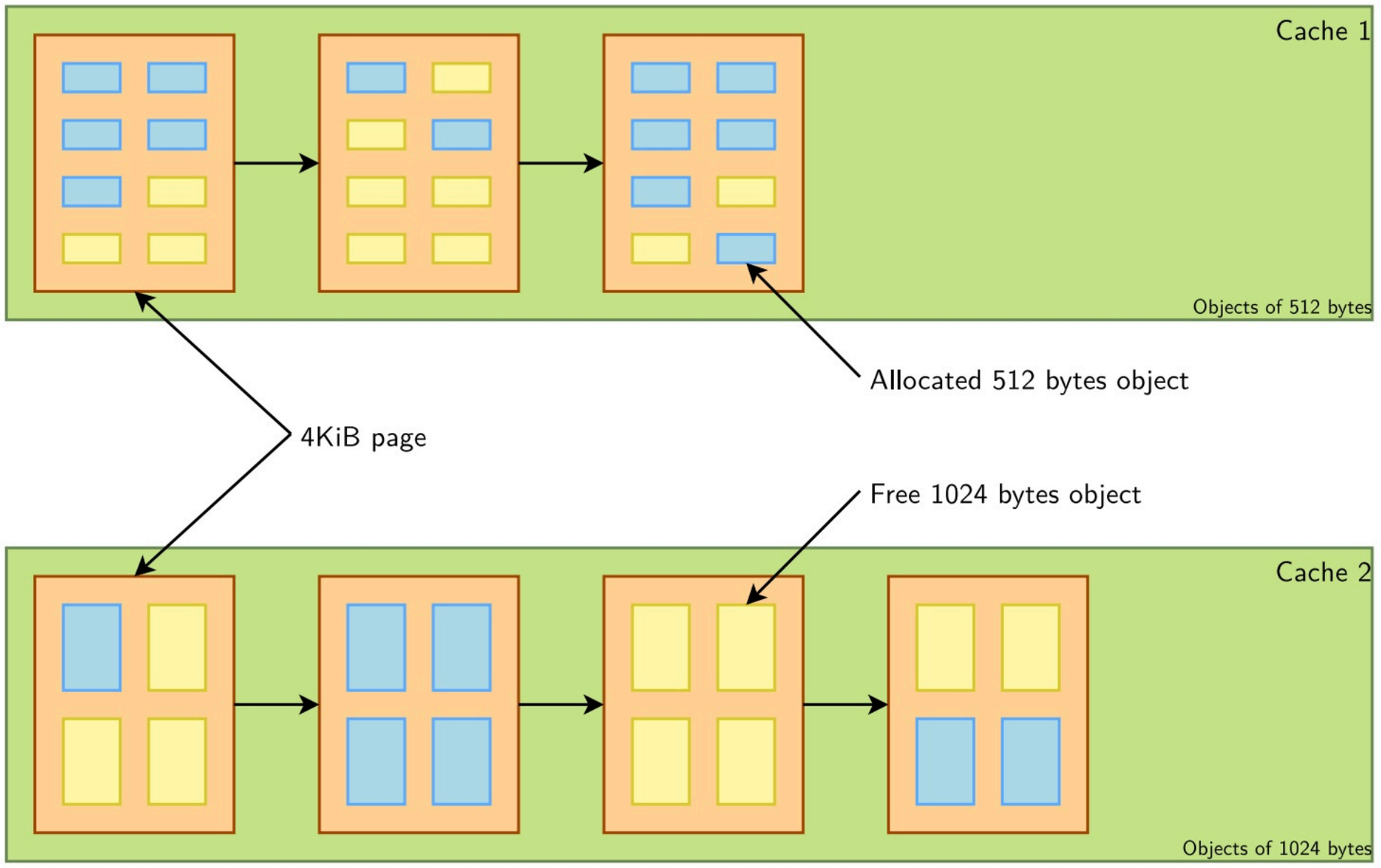


Slab, kmalloc, vmalloc and buddy



Acknowledgement:https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

Slab



Acknowledgement:https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

Different SLAB Allocators

There are three different, but API compatible, implementations of a SLAB allocator in the Linux kernel. A particular implementation is chosen at configuration time.

- SLAB: legacy, well proven allocator.
  Linux 4.20 on ARM: used in 48 `defconfig` files
- SLOB: much simpler. More space efficient but doesn't scale well. Saves a few hundreds of KB in small systems (depends on `CONFIG_EXPERT`).
  Linux 4.20 on ARM: used in 7 `defconfig` files
- SLUB: more recent and simpler than SLAB, scaling much better (in particular for huge systems) and creating less fragmentation.
  Linux 4.20 on ARM: used in 0 `defconfig` files

Acknowledgement:https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

kmalloc Allocator

- The kmalloc allocator is the general purpose memory allocator in the Linux kernel
- For small sizes, it relies on generic SLAB caches, named `kmalloc-XXX` in `/proc/slabinfo`
- For larger sizes, it relies on the page allocator
- The allocated area is guaranteed to be physically contiguous
- The allocated area size is rounded up to the size of the smallest SLAB cache in which it can fit (while using the SLAB allocator directly allows to have more flexibility)
- It uses the same flags as the page allocator (`GFP_KERNEL`, `GFP_ATOMIC`, `GFP_DMA`, etc.) with the same semantics.
- Maximum sizes, on `x86` and `arm` (see `http://j.mp/YIGq6W`):
  - Per allocation: 4 MB
  - Total allocations: 128 MB
- Should be used as the primary allocator unless there is a strong reason to use another one.

Acknowledgement:https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

vmalloc Allocator

- The `vmalloc()` allocator can be used to obtain virtually contiguous memory zones, but not physically contiguous. The requested memory size is rounded up to the next page.
- The allocated area is in the kernel space part of the address space, but outside of the identically-mapped area
- Allocations of fairly large areas is possible (almost as big as total available memory, see `http://j.mp/YIGq6W` again), since physical memory fragmentation is not an issue, but areas cannot be used for DMA, as DMA usually requires physically contiguous buffers.
- Example use: to allocate kernel buffers to load module code.
- API in `include/linux/vmalloc.h`
  - `void *vmalloc(unsigned long size);`
    - Returns a virtual address
  - `void vfree(void *addr);`

Acknowledgement:https://bootlin.com/doc/training/linux-kernel/linux-kernel-slides.pdf

Page Fault Handling

- handle_mm_fault()

```
1  //linux-2.6.34/arch/x86/mm/fault.c
2  dotraplinkage void __kprobes
3  do_page_fault(struct pt_regs *regs, unsigned long error_code)
4  {
5  …… ……
6  good_area:
7          write = error_code & PF_WRITE;
8
9          if (unlikely(access_error(error_code, write, vma))) {
10                 bad_area_access_error(regs, error_code, address);
11                 return;
12         }
13         fault = handle_mm_fault(mm, vma, address, write ? FAULT_FLAG
14 …… ……
15 }
```

Acknowledgement: http://edsionte.com/techblog/archives/4174

- handle_mm_fault(): allocate a physical frame
  - First determine whether the page directory entry exists

```
int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct *vma
                    unsigned long address, unsigned int flags)
{
        pgd_t *pgd;
        pud_t *pud;
        pmd_t *pmd;
        pte_t *pte;
        …… ……
        pgd = pgd_offset(mm, address);
        pud = pud_alloc(mm, pgd, address);
        if (!pud)
                return VM_FAULT_OOM;
        pmd = pmd_alloc(mm, pud, address);
        if (!pmd)
                return VM_FAULT_OOM;
        pte = pte_alloc_map(mm, pmd, address);
        if (!pte)
                return VM_FAULT_OOM;
          return handle_pte_fault(mm, vma, address, pte, pmd, flags)
}
```

Acknowledgement: http://edsionte.com/techblog/archives/4174

- If pte entry is none -> allocate physical frame
  - do_anonymous_page(), do_linear_fault() (for file)
- Physical frame exists, but does not in memory -> swap in

```
1  static inline int handle_pte_fault(struct mm_struct *mm,
2                  struct vm_area_struct *vma, unsigned long address,
3                  pte_t *pte, pmd_t *pmd, unsigned int flags)
4  {
5          …… ……
6          if (!pte_present(entry)) {
7                  if (pte_none(entry)) {
8                          if (vma->vm_ops) {
9                                  if (likely(vma->vm_ops->fault))
10                                         return do_linear_fault(mm, v
11                                                 pte, pmd, flags, ent
12                         }
13                         return do_anonymous_page(mm, vma, address,
14                                                  pte, pmd, flags);
15                 }
16                 if (pte_file(entry))
17                         return do_nonlinear_fault(mm, vma, address,
18                                 pte, pmd, flags, entry);
19                 return do_swap_page(mm, vma, address,
20                                 pte, pmd, flags, entry);
21         }
22 …… ……
23 }
```

Acknowledgement: http://edsionte.com/techblog/archives/4174

- do_anonymous_page->alloc_zeroed_user_highpage_movable->alloc_pages
- alloc_pages() uses the buddy allocator to allocate physical pages

```
1  static int do_anonymous_page(struct mm_struct *mm, struct vm_area_st
2                  unsigned long address, pte_t *page_table, pmd_t *pmd
3                  unsigned int flags)
4  {
5  …… ……
6          if (unlikely(anon_vma_prepare(vma)))
7                  goto oom;
8          page = alloc_zeroed_user_highpage_movable(vma, address);
9          if (!page)
10                 goto oom;
11 …… ……
12 }
```

Acknowledgement: http://edsionte.com/techblog/archives/4174