Module 3
Kernel Memory Allocation and Mapping

Linux Memory Management

- User programs access the virtual addresses that are mapped to physical memory address when accessed (page fault). Kernel and drivers also use virtual addresses but mapping to physical addresses are done in advance (no page fault)
- Hardware device called MMU (Memory Management Unit) and operating system translation tables (page tables) work together to translates these virtual addresses into physical memory addresses
  - TLB (Translation Lookaside Buffer) caches virtual to physical address mapping.
  - Page tables are consulted when TLB miss occurs. This can be achieved by combination of software or hardware.
- User address space is made up of linear range of virtual memory addresses. Where virtual address space is made up multiple memory segments, examples: stack, heap, text, data segments. file */proc/<pid>maps*
- Kernel has its own address space. kernel requires memory too for its kernel data structures, instructions and caches. Kernel memory is locked to avoid potential deadlock situations that could occur within the kernel if a kernel memory management function caused a page fault while holding a lock for another critical resource

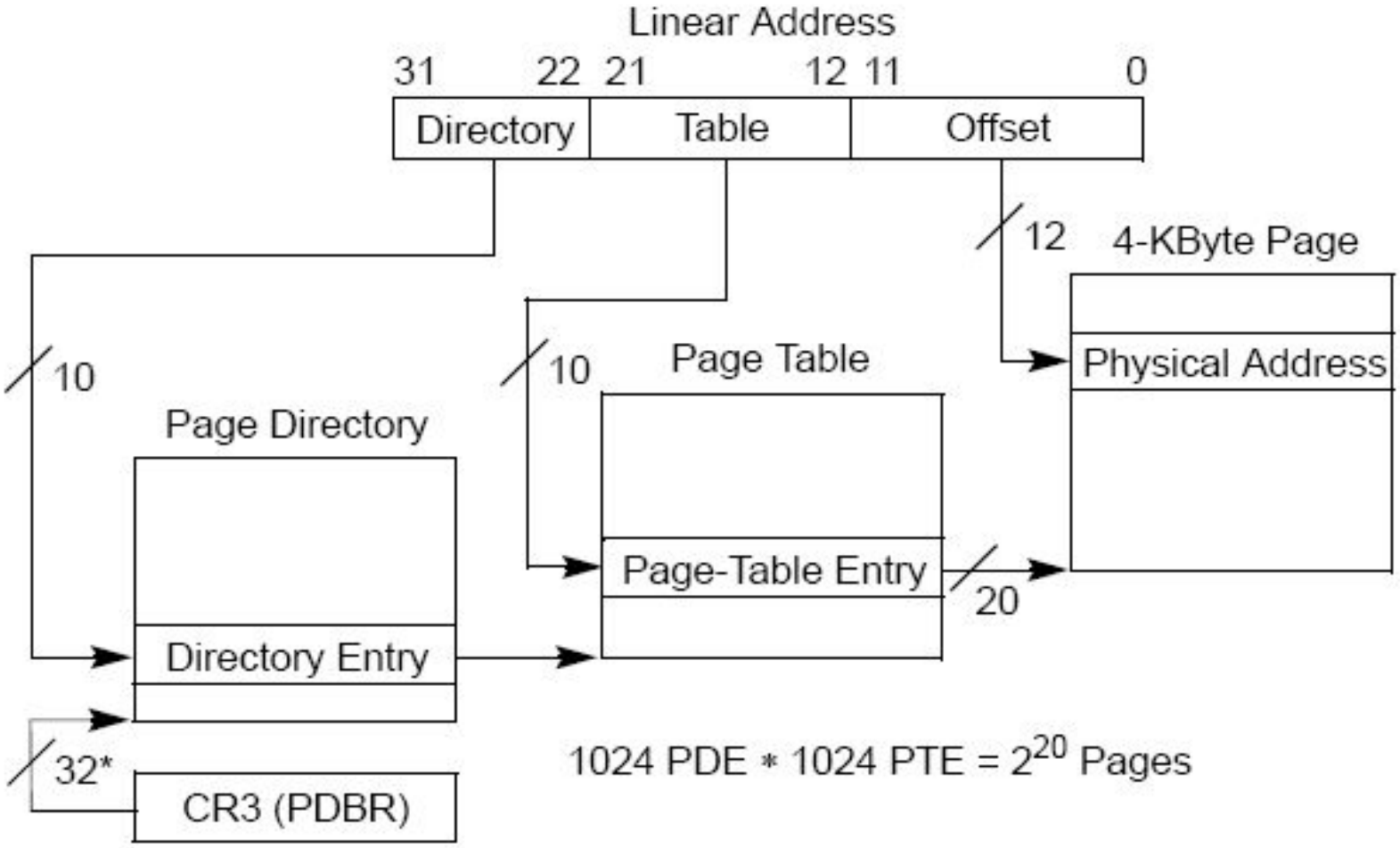
Linux Memory Management

- **Page frame:** A page is a group of contiguous linear addresses in physical memory (page frame). The Linux kernel handles memory with this page unit. A page is usually 4K bytes in size.
- **Page Descriptor:** State information of a page frame is kept in a page descriptor (*struct page*) and stored in *mem_map* array. Type of information is kept in page descriptor are: reference count, state of the page(dirty, reserved,locked etc..)
  - virt_to_page(addr): provides the address of the page descriptor associated with the virtual address, *addr*
  - pfn_to_page(pfn): provides the address of the page descriptor associated with the page frame, *pfn*.
- **PAGE_SHIFT:** Tells the offset field length in a linear address. On x86 architecture, it yields the value of 12 (2^12 or 4k boundary). Considering all addresses in a page must fit in the offset field, this macro is used by *PAGE_SIZE* to return size of the page.

```
#define PAGE_SHIFT 12
#define PAGE_SIZE (1UL << PAGE_SHIFT)
```

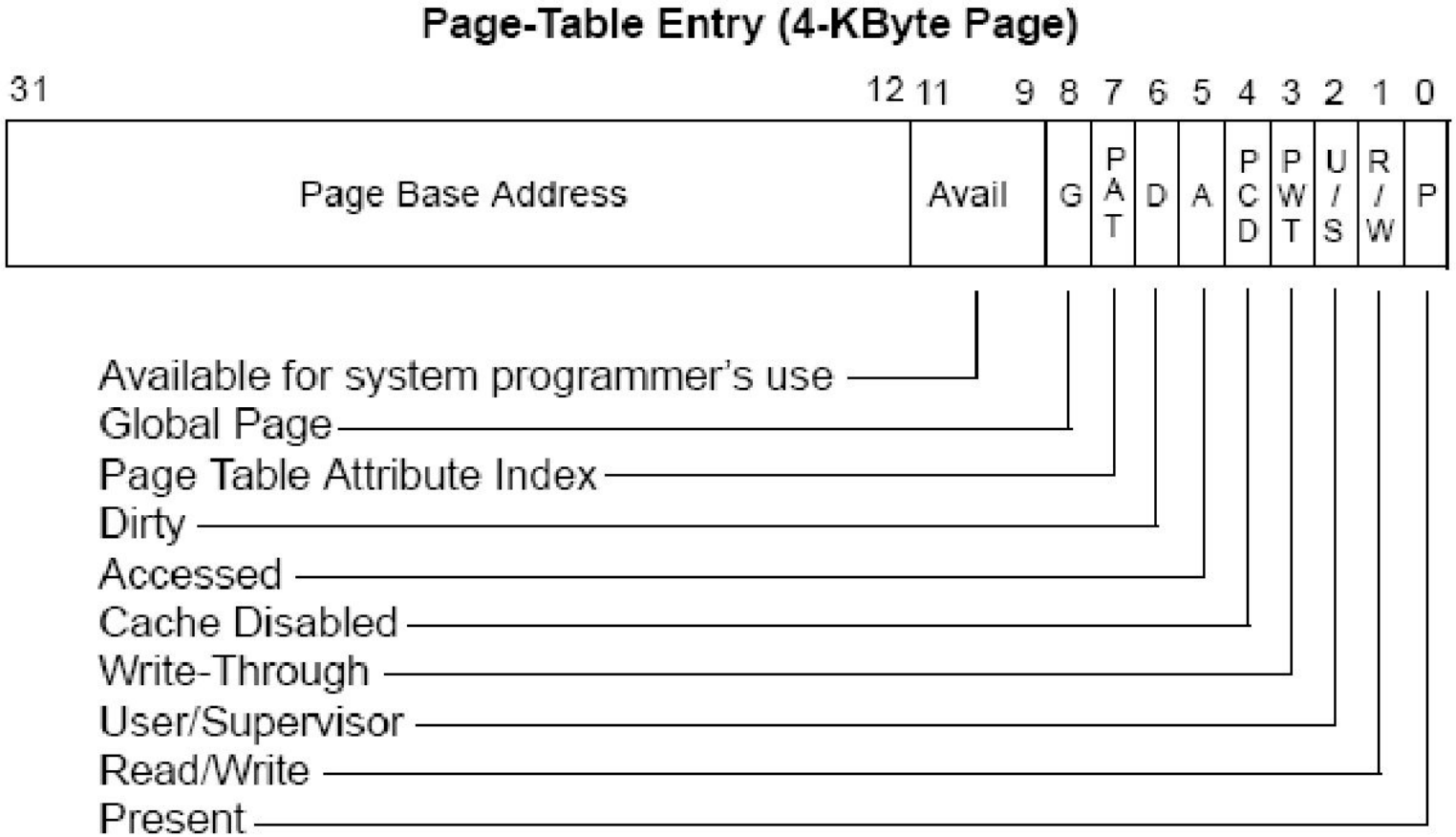
- **PAGE_MASK**: It is used to mask all the bits of the offset field in a linear address, mainly used for aligning the address at a page boundary.

x86 Address Translation
2-level Page Table



*32 bits aligned onto a 4-KByte boundary.

X86 PTE (Page Table Entry)



Kernel Memory Allocation Zones

All allocations take place from one out of three zones:

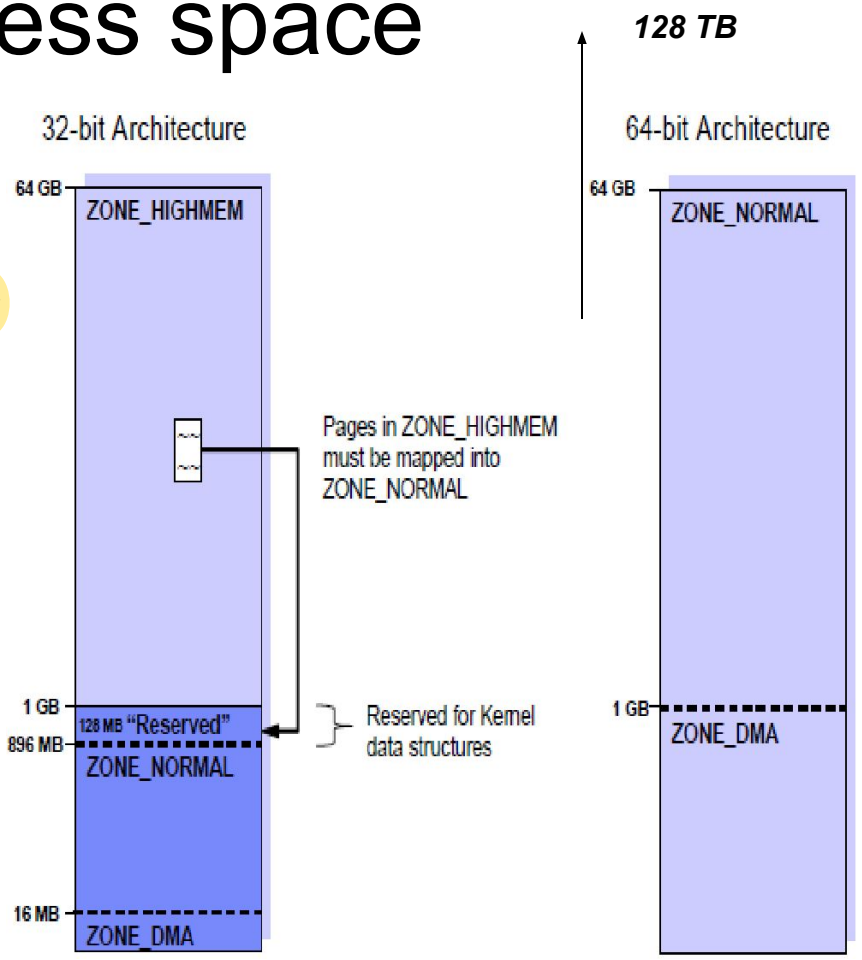
1. **ZONE_DMA**: Used for devices that can only perform DMA to a limited addresses of physical memory. Used by ISA devices. PCI devices has no such limitation
2. **ZONE_NORMAL**: For kernel memory allocation below 1G (896M) of physical address. This is where most of the kernel memory is allocated. This memory is directly mapped by simply adding/subtracting the PAGE_OFFSET (0xc000000 in x86), thus can be immediately accessed by the kernel

   phys_to_virt() adds 0xc0000000
   virt_to_phys() subtracts 0xc0000000
3. **ZONE_HIGHMEM**: For kernel memory allocation that reside above 1G of physical memory address on **32-bit** System. Unlike ZONE_NORMAL, this memory zone is not directly mapped. Temporarily mapped into kernel virtual memory is created when kernel needs access. Use for kernel data that needs occasionally access, such as: page cache, process memory and page tables.

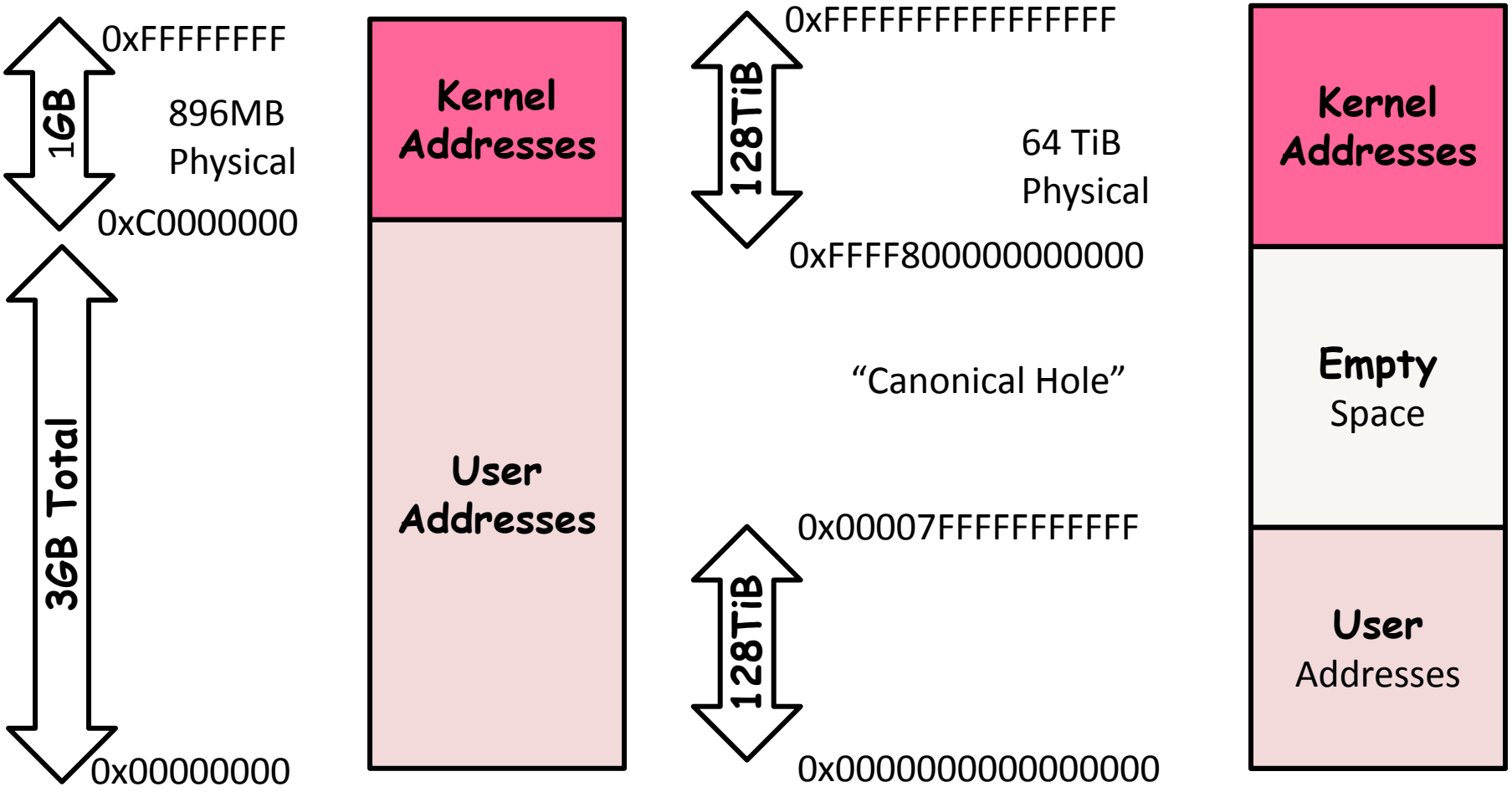
- Memory allocation functions specify the zone when requesting memory.
  - For instance: if a page frame must be directly mapped into the 4th GB of linear addresses but it is not ISA DMA transfers, the page frame will be allocated from the ZONE_NORMAL

32/64-bit Address space

- Application on 32-bit kernel can use less than 4G of memory.
- With PAE (Physical Address Extension), 32-bit system can able to utilize 64 GB of memory. However each process is still restricted to maximum of less than 4G
- Kernel can directly address first gigabyte of physical memory, also called ZONE_NORMAL, directly on 32-bit architecture.
- Kernel access to memory that resides in ZONE_HIGHMEM in 32-bit systems is slow due to global lock required in SMP machines and frequent mapping/unmapping needed to address this memory
- 64-bit architecture does not have these restriction and kernel can address large amount of memory directly with little overhead.
- In 64-bit platform, parameters are passed in cpu registers (faster) than through the stack (slower)



Linux Virtual memory map



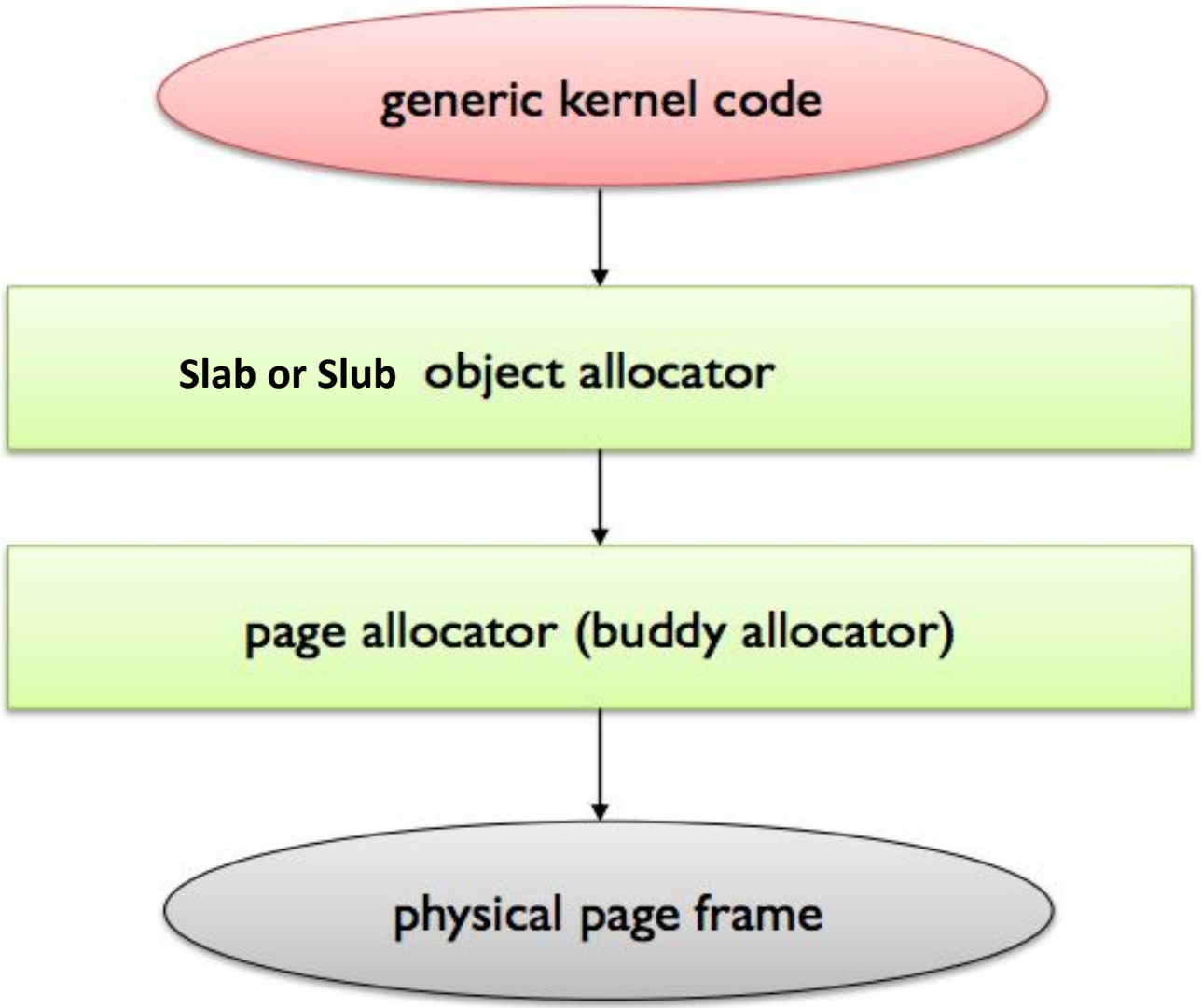
32-Bit Virtual Address Space

64-Bit Virtual Address Space

Type of Addresses

- **User virtual addresses:** User application memory and address space. It can be 32-64 bit
- **Physical addresses :** The addresses used between the processor and the system's memory. It also can be 32-64-bit quantities; Even 32-bit systems can use larger physical address using High Memory
- **Bus addresses**: Addresses used between peripheral buses and memory. On old x86 platform, it is the same as physical address. Platform that supports IOMMU can remap addresses between a bus and main memory (RAM)
- **Kernel Logical Addresses**: Kernel memory or address space. Treated as physical addresses. Memory returned from kmalloc has a kernel logical address.
- **Kernel Virtual Addresses**: Similar to logical addresses. Unlike logical address, kernel virtual addresses may not have a linear, one-to-one mapping to physical addresses. All logical addresses are kernel virtual addresses, but not all kernel virtual addresses are logical addresses (high memory addresses do not have logical addresses).

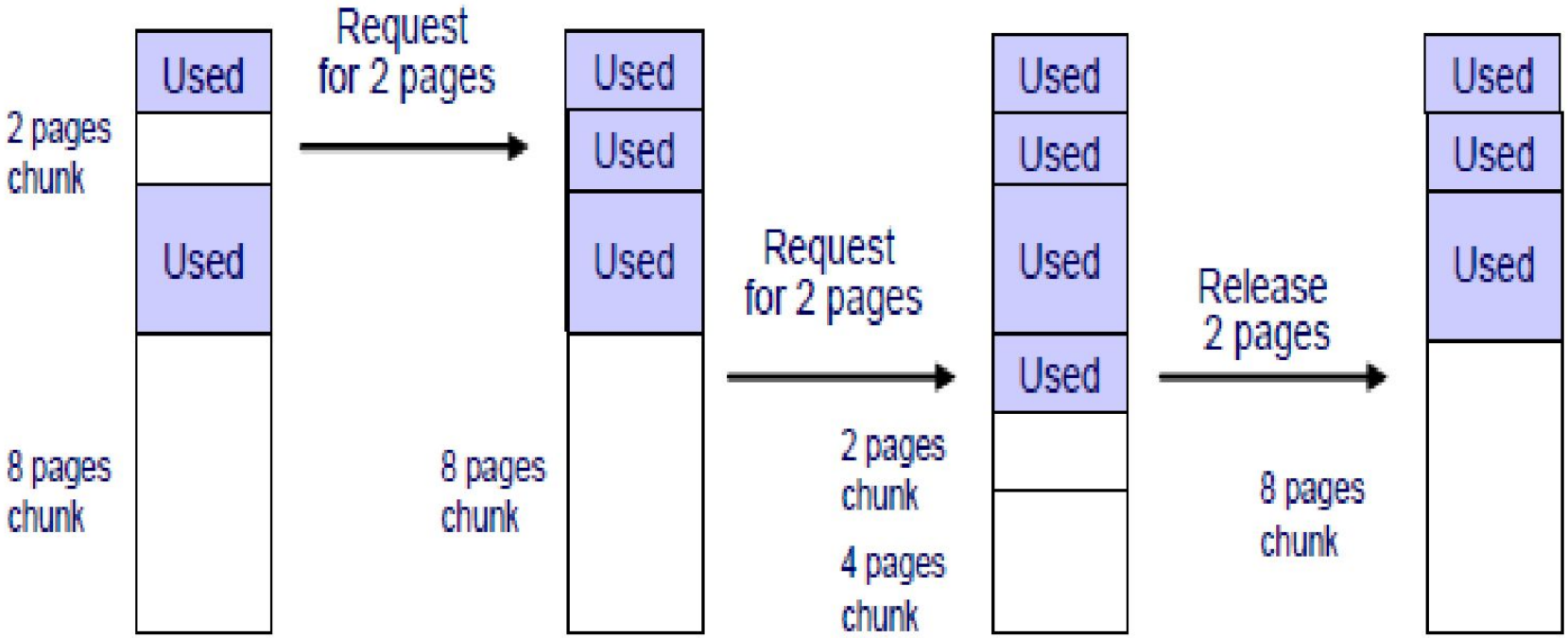
Linux Kernel Memory Allocation



Buddy Allocator

- Linux kernel maintains its free pages by using a mechanism called a buddy system.
- The buddy system maintains free pages and tries to allocate pages for page allocation requests. It strives to keep the memory area contiguous.
  - All free page frames are grouped into 11 lists of blocks that contain groups of *1,2,4,8,16,32,64,128,256,512 and 1024* contiguous 4k page frames respectively. Largest request for contiguous chunk is: 4M (1024 contiguous page frames)
- Memory allocation request of contiguous page frames are checked in the list that has equal or more contiguous page frames requested.
  - Example: Request for 512 contiguous page frames is granted from the block of 512 page frames, if available. Otherwise, next larger (1024) block is checked. It allocates 512 contiguous page frames and insert the remaining 512 pages into the list of 512 blocks. If 1024-page-frame blocks is empty, the error is returned.
- When the memory is released, kernel attempts to merge block of released page frames to free adjacent (buddy) block of the same size to create a single larger contiguous block. This helps reduce fragmentation

Buddy System



Slab Allocator

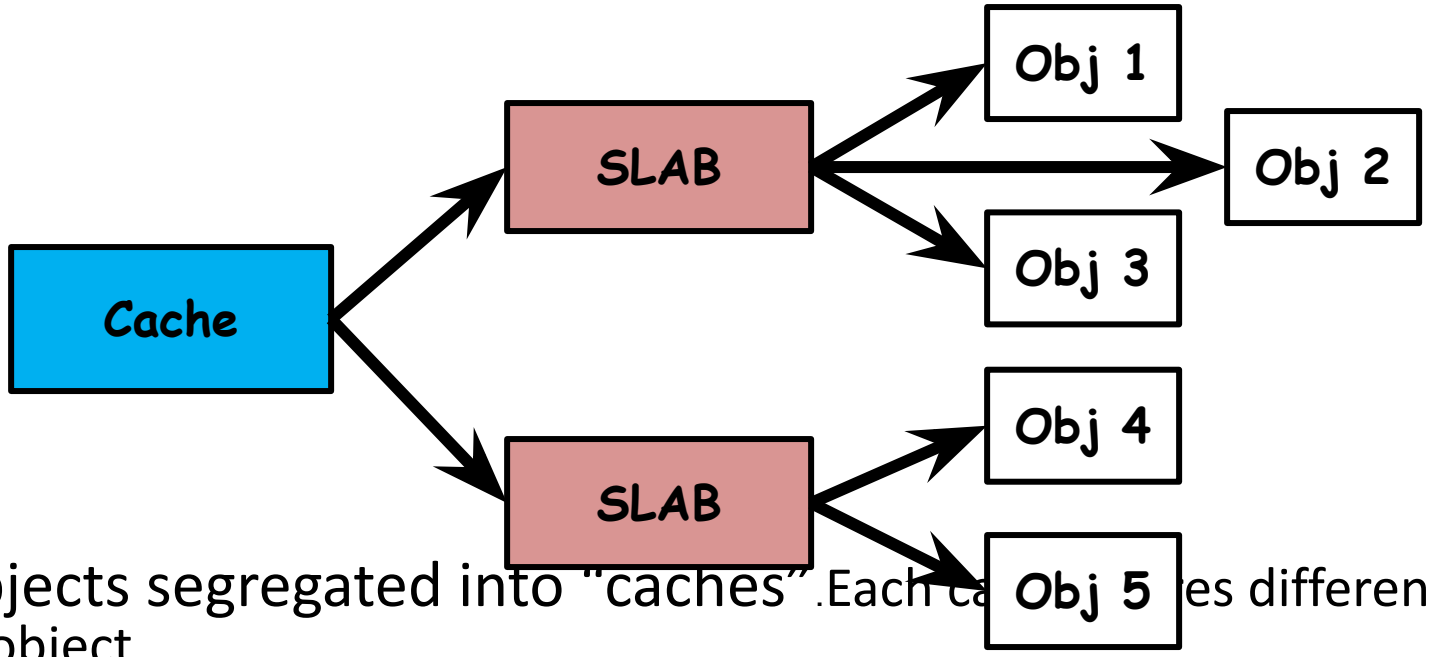
Slab allocator is kernel allocator and a client of buddy system. It subdivides contiguous physical memory allocated from the buddy system into small chunks of frequently used data structures (objects). Slab allocator has number of key features:

- Many drivers requests memory that is less than a full page. Since kernel works in unit of 4k pages, it is wasteful to give the whole page of memory. Slab allocator can manage various sizes of allocation
- Reduce memory fragmentation. Objects can be allocated and freed numerous times over the lifetime by recycling through the same cache.
  - The slab cache allocator provides this through the caching of similarly sized objects, The allocation of small blocks of memory eliminates internal fragmentation that would be otherwise caused by the buddy system
- Speed up the memory allocation by keeping caches of small objects of varying sizes. The slab allocator also supports the initialization of common objects, thus avoiding the need to repeatedly initialize an object for the same purpose.
  - objects are partially pre allocated, that speeds up the memory allocation for latency sensitive tasks. e.g: sk_buff, scsi structures
- Allocation requests are handled by going to a pool that holds sufficiently large objects and handing an entire memory chunk back to the requester.

slabinfo and slabtop Linux utilities provide statistics of slab usage

SLAB Allocator

- Efficient when objects allocated and freed frequently



- Objects segregated into "caches". Each ca es different type of object.
- Data inside cache divided into "slabs", which are continuous groups of pages (often only 1 page). This avoid memory fragmentation

SLAB Allocator: Caches

- How new caches are created:

```
struct kmem_cache * kem_cache_create(const char *name,
                    size_t size,
                    size_t align,
                    unsigned long flags,
                    void (*ctor)(void *));
```

  - name: name of cache
  - size: size of each element in the cache
  - align: alignment for each object (often 0)
  - flags: possible flags about allocation

    - SLAB_HWCACHE_ALIGN: Align objects to cache lines
    - SLAB_POISON: Fill slabs to known value (*0xa5a5a5a5*) in order to catch use of uninitialized memory
    - SLAB_RED_ZONE: Insert empty zones around objects to help detect buffer overruns

SLAB Allocator: Named Caches

- Create Cache Example:

```
task_struct_cachep =
 kmem_cache_create(“task_struct”,
                   sizeof(struct task_struct),
                  ARCH_MIN_TASKALIGN,
                  SLAB_PANIC | SLAB_NOTRACK,
                  NULL);
```

- Use Cache example:

```
struct task_struct *tsk;

tsk = kmem_cache_alloc(task_struct_cachep, GFP_KERNEL);
if (!tsk)
 return NULL;

kmem_free(task_struct_cachep,tsk);
```

Slab, Slub and Slob – What's the difference

- A number of options in the kernel for object allocation:
  - SLAB: original allocator based on Bonwick's paper from SunOS
  - SLUB: Newer allocator with same interface but better use of metadata
    - Keeps SLAB metadata in the page data structure (for pages that happen to be in kernel caches)
    - Debugging options compiled in by default, just need to be enabled
  - SLOB: low-memory footprint allocator for embedded systems

Slub Allocator

- Not much different than Slab allocator
- Recommended for large NUMA servers due to numa awareness when allocating memory
- Enhanced debugging features

https://www.kernel.org/doc/Documentation/vm/slub.txt

```
slub_debug=<Debug-Options>       Enable options for all slabs
slub_debug=<Debug-Options>,<slab name>
                                 Enable options only for select slabs

Possible debug options are
        F               Sanity checks on (enables SLAB_DEBUG_FREE. Sorry
                        SLAB legacy issues)
        Z               Red zoning
        P               Poisoning (object and padding)
        U               User tracking (free and alloc)
        T               Trace (please only use on single slabs)
        A               Toggle failslab filter mark for the cache
        O               Switch debugging off for caches that would have
                        caused higher minimum slab orders
        -               Switch all debugging off (useful if the kernel is
                        configured with CONFIG_SLUB_DEBUG_ON)

F.e. in order to boot just with sanity checks and red zoning one would specify:

        slub_debug=FZ
```

Type of Kernel Allocation

- **alloc_bootmem():** used at boot time. Code is deleted after initialization!
- **get_free_pages():** get a power-of-two multiple (buddy system blocks) of PAGE_SIZE contiguous physical pages. Use *get_order()* to determine number of pages from a linear size. Sizes up to about 8MB are OK.
- **kmalloc():** get any size (but actually a power-of-two is allocated from the geometrically distributed size slab caches). Maximum size is 128KB
- **kmem_cache_alloc():** get predefined size from a named cache (slab kmem_caches). Various parts of the kernel creates their own cache via slab interface. These caches contain similar type of objects (data structures) that are initialized by their own constructor E.g: cache for process descriptor, file objects, skbuff..etc.. When a program sets up the name cache, it allocates number of objects from the slabs.

Type of Kernel Allocation

- **vmalloc():** allocate contiguous virtual memory, which corresponds to non-contiguous physical memory. Use instead of kmalloc for large chunks of data
- **request_mem_region():** Drivers calls this function to reserves specific *physical* addresses ranges where device memory is mapped.
- **remap_pfn_range():** reserves a virtual address range and maps it to a given range of physical pages. Pages must have been allocated already. Typically used for implementing user-space *mmap*() for a device. Allows direct access to device memory or kernel memory from user space
- **ioremap():** This function does not allocate any memory, instead it creates additional pages tables to provide access to device memory

get_free_pages – page size allocation

- It is a page size allocation. Page size is 4k on x86 platform
- To allocate large chunk of memory, page size allocation mechanism is used.
- memory obtained by *get_free_pages* uses logical addresses, You can get a ***struct page (page descriptor)*** pointer by calling ***virt_to_page***()

Note: Virt_to_page() does not work with kernel virtual address return by vmalloc() considering pages are not contiguous. For kernel virtual address, use vmalloc_to_page() to get a struct page.

kmalloc

- Kernel equivalent malloc()/free is kmalloc() and kfree().
- Unlike malloc() that only gives virtual addresses, kmalloc() allocates physical memory
- Any size of memory can be requested (but return memory is actually a power-of-two and allocated from the default slab). Maximum size is usually 128KB

Features:

- Fast, unless block due to memory shortage and waiting for memory freed
- Memory area is contiguous in physical memory
- Adds few management bytes, allocates as 2^n sizes.
  - Example: If you ask for an arbitrary amount of memory, you are likely to get slightly more due to limited number of fixed size slabs available.
- Smallest allocation that kmalloc() can handle is 32 byte on x86

get_free_pages

- Get a power-of-two multiple of PAGE_SIZE contiguous physical pages.
- Use get_order() to determine number of pages from a linear size.
- Sizes up to about 8MB are OK.
- Order is the base-two algorithm of the number of pages you are requesting or freeing (i.e., log2*N*). For example, order is 0 if you want one page and 3 if you request eight pages. If order is too big (no contiguous area available) allocation fails.

```
unsigned long get_zeroed_pages (int flags);
unsigned logn __get_free_pages(int flags);
unsigned long __get_free_pages (int flags, unsigned int order);

void free_page (unsigned long addr);
void free_pages (unsigned long addr, unsigned int order)
```

vmalloc

•To allocate contiguous virtual memory, which corresponds to non-contiguous physical memory.

• Use instead of kmalloc for large chunks of data.

getting many contiguous physical pages leads to fragmentation

• Can be used to obtain contiguous memory zones in virtual address space even if pages may not be contiguous in physical memory

•Kernel sees them as a contiguous range of addresses.

•Memory allocated via vmalloc is slightly less efficient to work.

•If possible, you should work directly with individual pages rather than trying to smooth things over with *vmalloc*

```
Void *vmalloc (unsigned long size);
Void vfree(void *addr);
```

Lookaside Caches

- If your driver frequently allocates objects of the same size and default set of caches that slab allocator maintains do n't fit your needs then you can create a special pool for these high-volume objects
- The facility that allows you to create this sort of pool is called a *lookaside cache*.

```
kmem_cache_t *kmem_cache_create(const char *name, size_t size, size_t offset,
unsigned long flags)
/* Once a cache of objects is created, you can allocate objects from it by calling */
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
```

Allocation flags

- Possible allocation type flags:
  - GFP_ATOMIC: Allocation high-priority and must never sleep. Use in interrupt handlers, top halves, while holding locks, or other times cannot sleep
  - GFP_NOWAIT: Like GFP_ATOMIC, except call will not fall back on emergency memory pools. Increases likelyhood of failure
  - GFP_NOIO: Allocation can block but must not initiate disk I/O.
  - GFP_NOFS: Can block, and can initiate disk I/O, but will not initiate filesystem ops.
  - GFP_KERNEL: Normal allocation, might block. Use in process context when safe to sleep. This should be default choice
  - GFP_USER: Normal allocation for processes
  - GFP_HIGHMEM: Allocation from ZONE_HIGHMEM
  - GFP_DMA Allocation from ZONE_DMA. Use in combination with a previous flag

memset and memcopy

- libc like memory utilities for device driver use
- Fills a region of memory with the given value
- Copies one area of memory to another
- Lots of equivalent functions to Standard C library such as sprintf().

```
/* Libc like Memory Utilities*/

Void *memset (void *s, int c, size_t count)
Void * memcpy (void * dest, const void *src, size_t count);
```

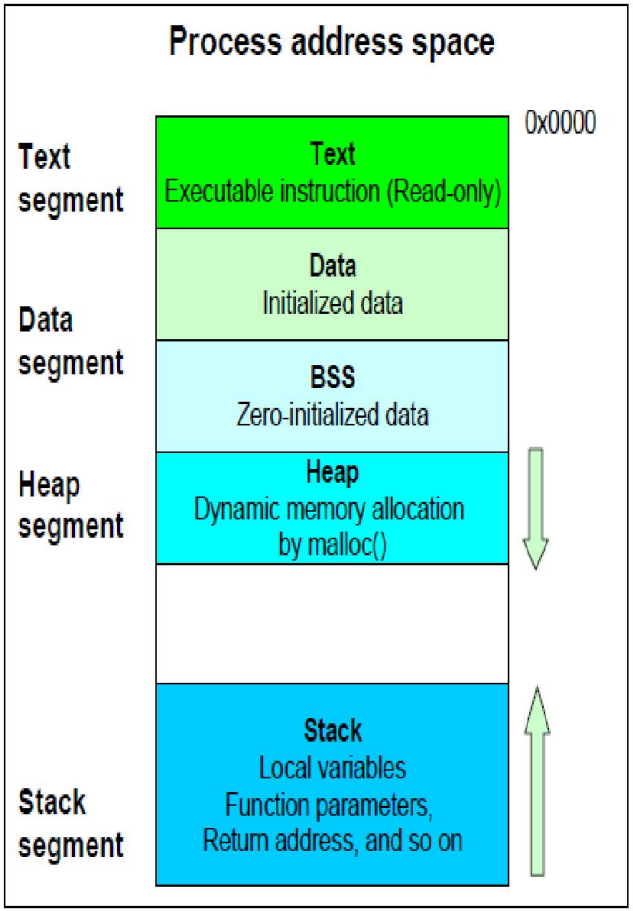
mmap

- This is a standard UNIX system-call that lets an application map a file into its virtual address-space, where it can then treat the file as if it were an ordinary memory array.
- This same system-call can also work on a device memory if the device driver provides ***mmap***() among its file-operations methods
- Take advantage of the paging mechanism by associating virtual addresses with the data. Drivers often implement mmap() to allow user application to have direct access to memory that was allocated/reserved within kernel space
- Mapping device memory can yield significant performance improvements
- Data changed through *mmap* does not go through read/write syscalls, that means no transitions to kernel mode.

Process Address Space

- A page is a group of contiguous linear addresses in physical memory (page frame) or virtual memory. The Linux kernel handles memory with this page unit. A page is usually 4K bytes in size
- To execute a process, kernel allocates memory areas in units of pages.
- Memory area used by the process to perform its work is called process address space.
  - Process address space is defined as the range of memory addresses that presented to process as its environment
- Typically process address space is consist of memory segments of type:
  - **Text:** Application instruction or code
  - **Data:** initialized and zero-initialized data (BSS)
  - **Heap:** dynamic memory allocation area based on demand, allocated using malloc(). Heap segment grows towards higher address
  - **Stack:** Local variables, function parameters and return address of the function are stored here. The stack grows towards lower address.
  - **mmap:** If the application has file data or device memory mapped in user space then it appears in user application as a separate segment

One can view process address space by running:
pmap –x <PID>



32-bit vs. 64-bit Application

- 64-bit processors can address up to 128 TB (in theory 16-exabyte), whereas 32-bit is limited to a maximum of 3GB (in theory 4-GB) of memory
- 64-bit addresses enables the application like databases to manipulate large sets of data in memory
- 64-bit instruction set extensions for x86 processor are referred to as AMD64, EMT64, x86-64 or just x64. Architecture has much improved instruction set and ABI (Application Binary Interface):
  - x64 has abandoned the stack-based calling convention. Function parameters are now passed in registers rather than through the stack. Thus no load and store to/from the stack is required when accessing function parameters. In x86 (32-bit code) when a function is called, all the parameters to that function needed to be stored on the stack.
  - x64 has more general-purpose registers (6 in 32-bit code, 14 in 64-bit code). That means reduce frequency of register spills and fills events.
- 64-bit executable may have some overhead when compared to 32-bit executables:
  - Longs and pointers become 8-bytes structure (64-bits) rather than 4-bytes (32-bits).
  - Memory footprint of a 64-bit application is larger than the same application compiled for 32-bits.
  - Performance can degrade for certain application after compiling as 64-bit. Application requires more CPU cache entries to fit the working set size. That may result in higher cache misses.

mmap

- Application address space is composed of number of memory segments of type: text, data, heap, stack
- When application invokes driver's mmap() file operation method, it requests to allocate a new memory segment of type ***mmap*** where kernel memory (or device IO memory space) should be mapped
- Application also specifies the offset in device IO memory to be mapped into this new memory segment. Normally the offset is 0 – means start of device IO memory region. However, for files on a file system, it can be any offset in the file.

mmap

There are four steps in setting up mmap() in driver:

- Compute memory segment starting point and length
- Check to make sure not to map past the end of device memory
- Mark mapped area as non-swappable
- Request kernel to set up the page-tables

mmap() - file operation method

int mmap(struct file *file, struct vm_area_struct *vma)

- When application makes *mmap*() syscall, kernel validates the argument passed and then invokes *mmap*() file operation method of the driver
- Second argument of driver *mmap*() method contains address of structure of type ***vma_area_struct***. vma structure contains all the information required for driver to map device or kernel memory to user process address space.
- Driver calculates length of the mapping requested by application: ***vma->vm_end – vma->vm_start*** . Return *–EINVAL*, if the requested size is bigger than the size of device memory

mmap() - file operation method

- Offset in the device memory space requested to be mapped is saved in ***vma->vm_pgoff***.
- Driver checks to see if the offset is valid. It then creates mapping for user virtual addresses to the device physical memory.
- Driver use helper function ***remap_pfn_range()*** to perform the task of mapping application virtual addresses to device physical memory
- ***remap_pfn_range***() function takes physical address as an argument. For example:
  - Virtual address returned by kmalloc() memory can be converted to physical memory by ***virt_to_phys***()
  - Physical address of the PCI device memory region can be extracted from pci configuration space by using ***pci_resource_start***() and ***pci_resource_len***()

mmap() - file operation method

- ***vma->vm_flags*** can be ORed by the driver to make all the pages in the mmap segment of the process to be flagged VM_RESERVED
- Pages with VM_RESERVED flag set is an indication to kernel not to swap out the pages during memory pressure
- Pages with VM_IO mean exclude these pages from the process core dump

```
vma->vm_flags |= VM_RESERVED;
vma->vm_flags |= VM_IO
```

NOTE: remap_pfn_range() routine in later Linux version sets VM_RESERVED flag even for kernel memory. Driver writers mapping kernel memory to user space are no longer need to set this flag.

mmap

System call:

mmap (caddr_t addr, size_t len, int prot, int flags, int fd, off_t offset)

- All memory segments including mmap() memory that are part of process address space can be displayed using: $cat /proc/<pid>/map or “pmap -x <pid>”

File operation method:

int (*mmap) (struct file *filp, struct vm_area_struct *vma);

- vma struct contains the information about the virtual address range and the device offset. To implement *mmap*, the driver builds page tables or mapping for the address range using **remap_pfn_range().**

remap_pfn_range()

remap_pfn_range(vma, vma->vm_start, pfn, vma->vm_end-vma->vm_start, vma->vm_page_prot);

- **vma** - the virtual memory area into which the page range is being mapped
- **vm_start:** It is a user virtual address where mapping should begin. It means build page tables for the virtual address range between **vm_start** and **vm_end**
- **pfn:** It is a page frame number - a physical address right-shifted by PAGE_SHIFT bits. On x86, this gives us a physical page frame that is aligned on a 4KB boundary.
- **size of mapping:** vm_end – vm_start, the area being mapped in bytes
- **prot** - The "protection" requested for the new VMA. The driver can (and should) use the value found in vma->vm_page_prot. It described whether pages mapped can be read (*PROT_READ*), written (*PROT_WRITE*) etc..

remap_pfn_range()

- Main objective of remap_pfn_range function is to create a page table entry that points to physical address of device memory or kernel memory.
- 3rd argument in remap_pfn_range is a page frame number (pfn). It is a physical address right-shifted by PAGE_SHIFT bits.
  - On x86, this gives us a physical page frame that is aligned on a 4KB boundary.
  - Processes uses memory in 4k chunks, thus it is required to align it in 4k boundary.
- When mmap-ing kernel memory allocated via kmalloc(), one has to verify if the memory buffer allocated is at least a multiple of page size and the address is aligned at the page boundary (kmalloc buffer can be smaller than 1 page size and may not be page aligned)
  - mmap system call creates a new memory segment into process address space. It is required that process memory segments start at the page aligned boundary and should have a size that is multiple of PAGE_SIZE.
  - you can always allocate kernel memory in page size chunks using *get_free_pages()* and thus may not be required to page aligned the buffer allocated, considering it is already page aligned
- Thus to perform mmap on kmalloc allocated memory, you should do the following:

km-page-aligned-buffer = (char *)(((unsigned long)km-buffer + PAGE_SIZE - 1) & PAGE_MASK)

km-page-aligned-buffer and **km-buffer** may have same value if **km-buffer** is already page aligned, otherwise, **kma-page-aligned-buffer** points to the beginning of the next page.