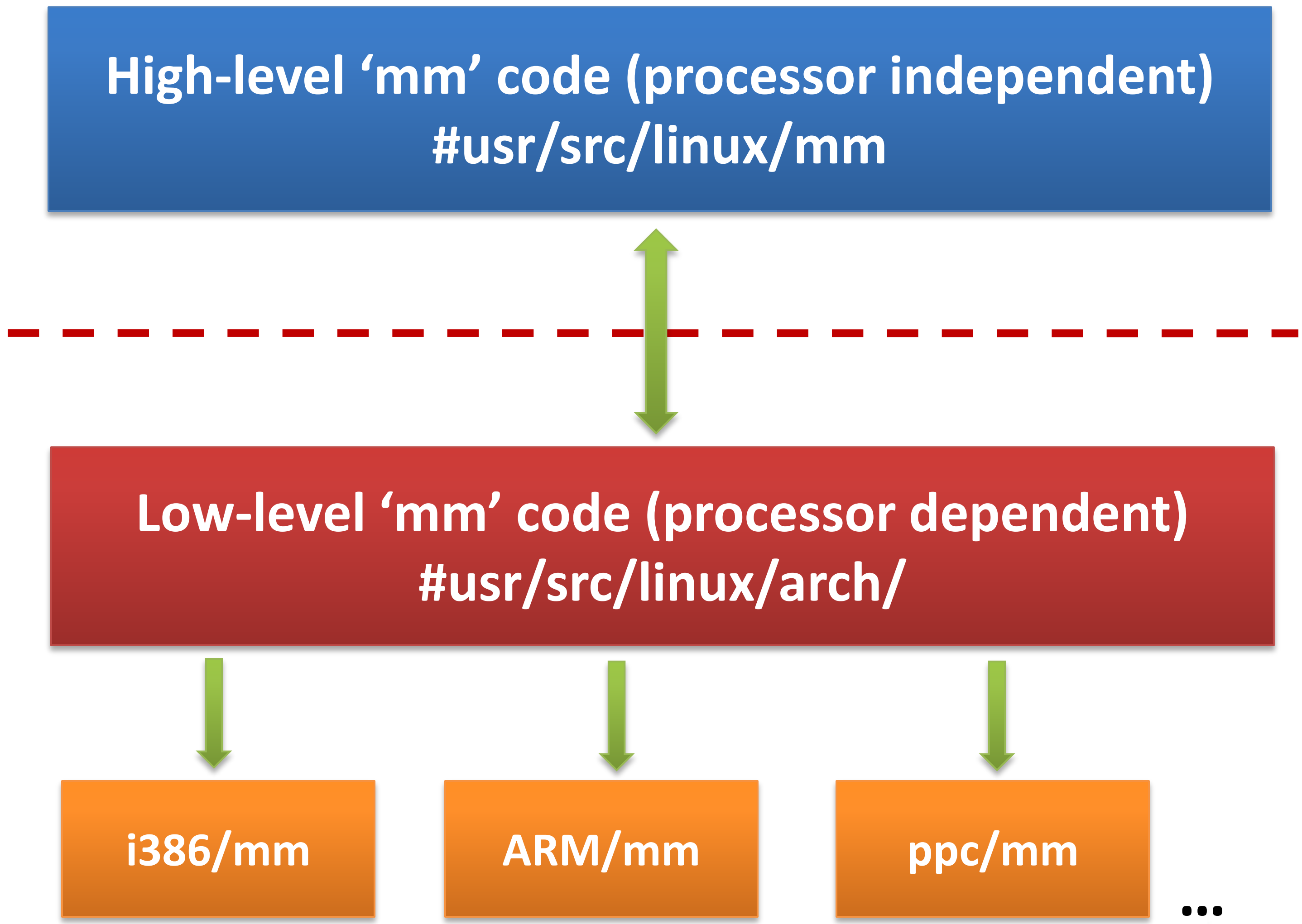
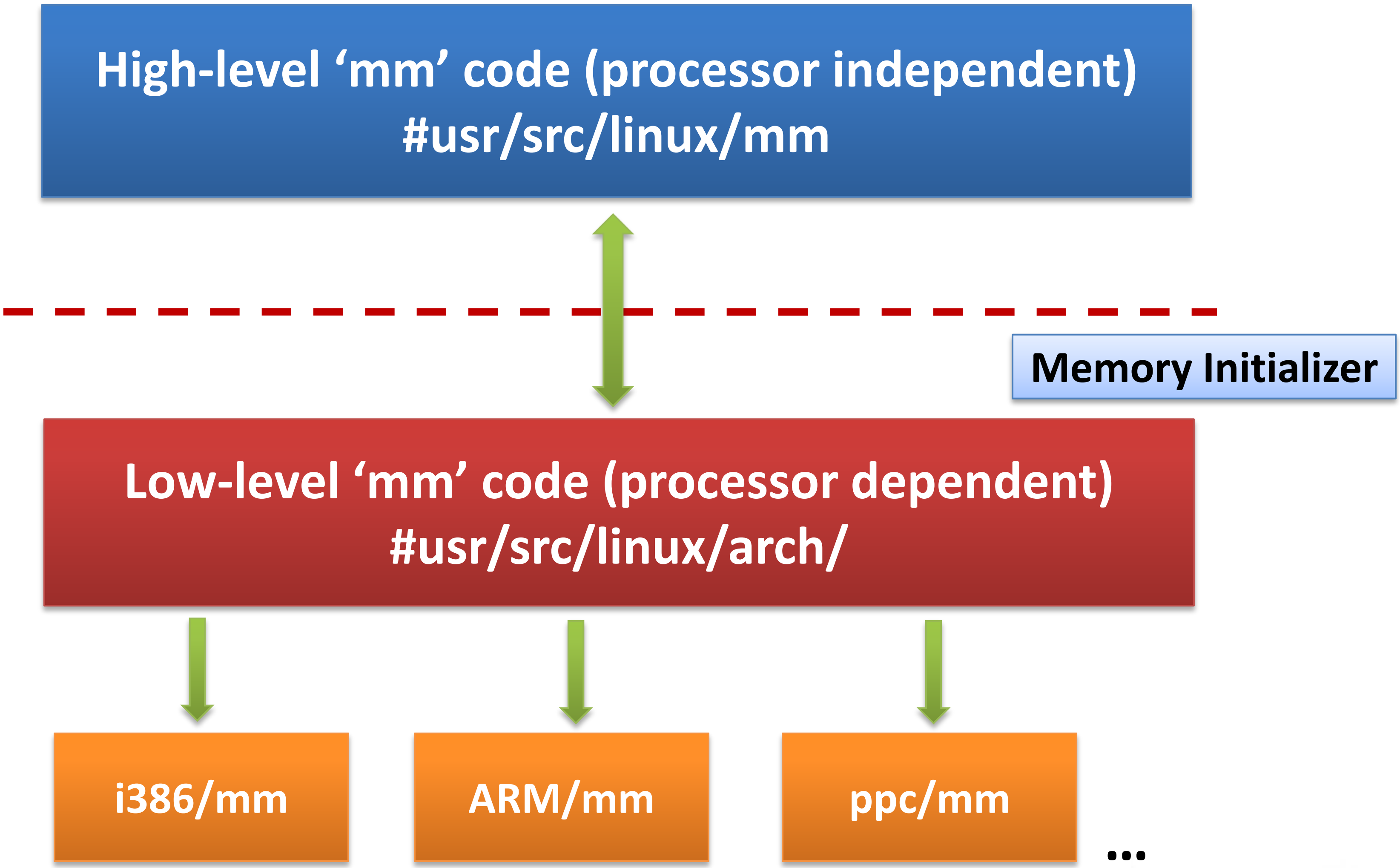
# Memory Management

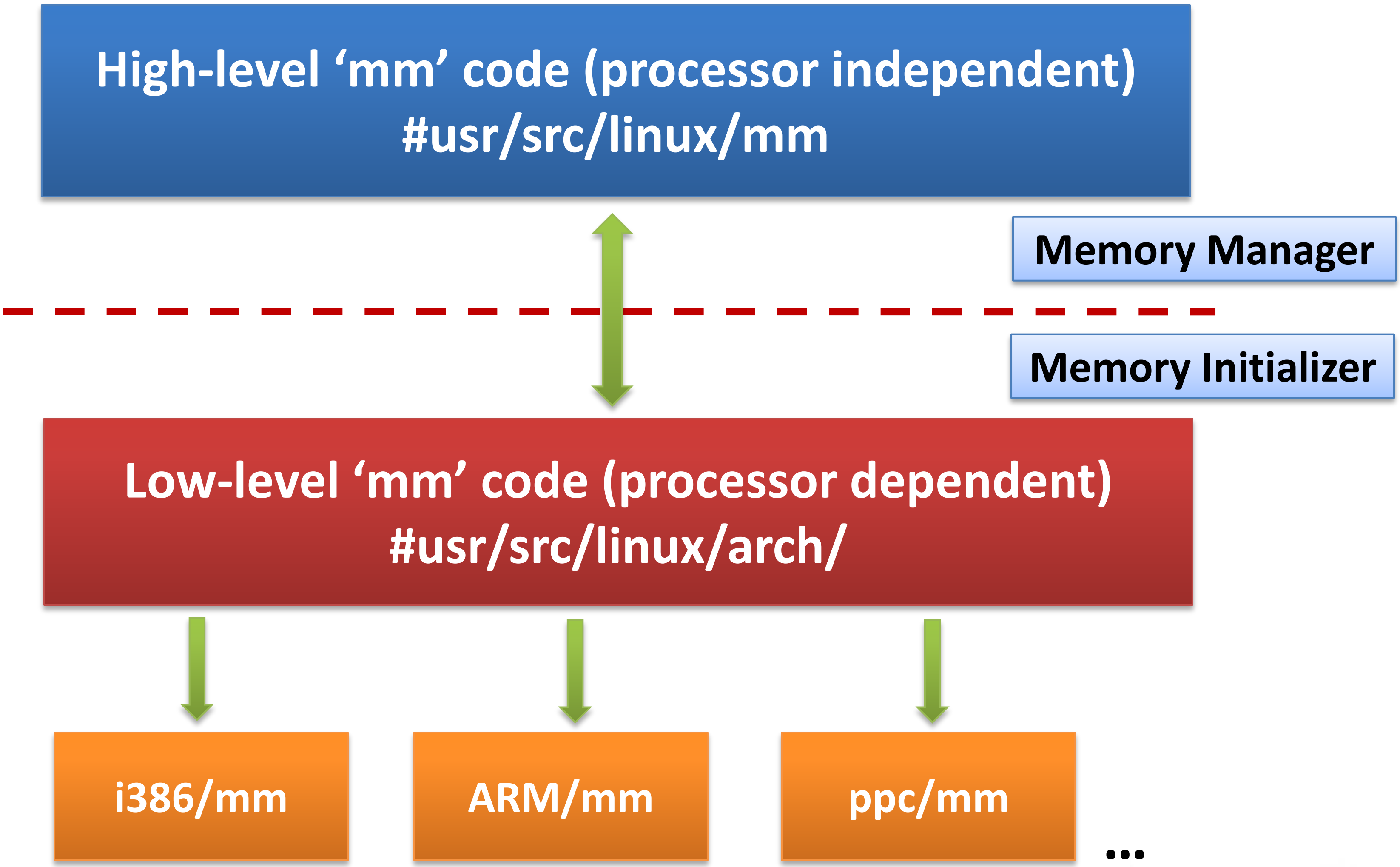
# Memory Subsystem Overview



# Memory Subsystem Overview



# Memory Subsystem Overview



# Memory View

**Processor's view of memory in Real-mode**

As a single Array

# Memory View

Processor's view of memory in Real-mode

As a single Array

Physical address = offset

# Memory View

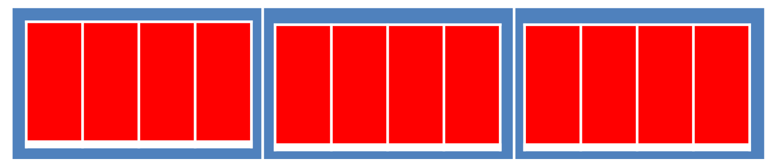
**Processor's view of memory in Real-mode**

As a single Array

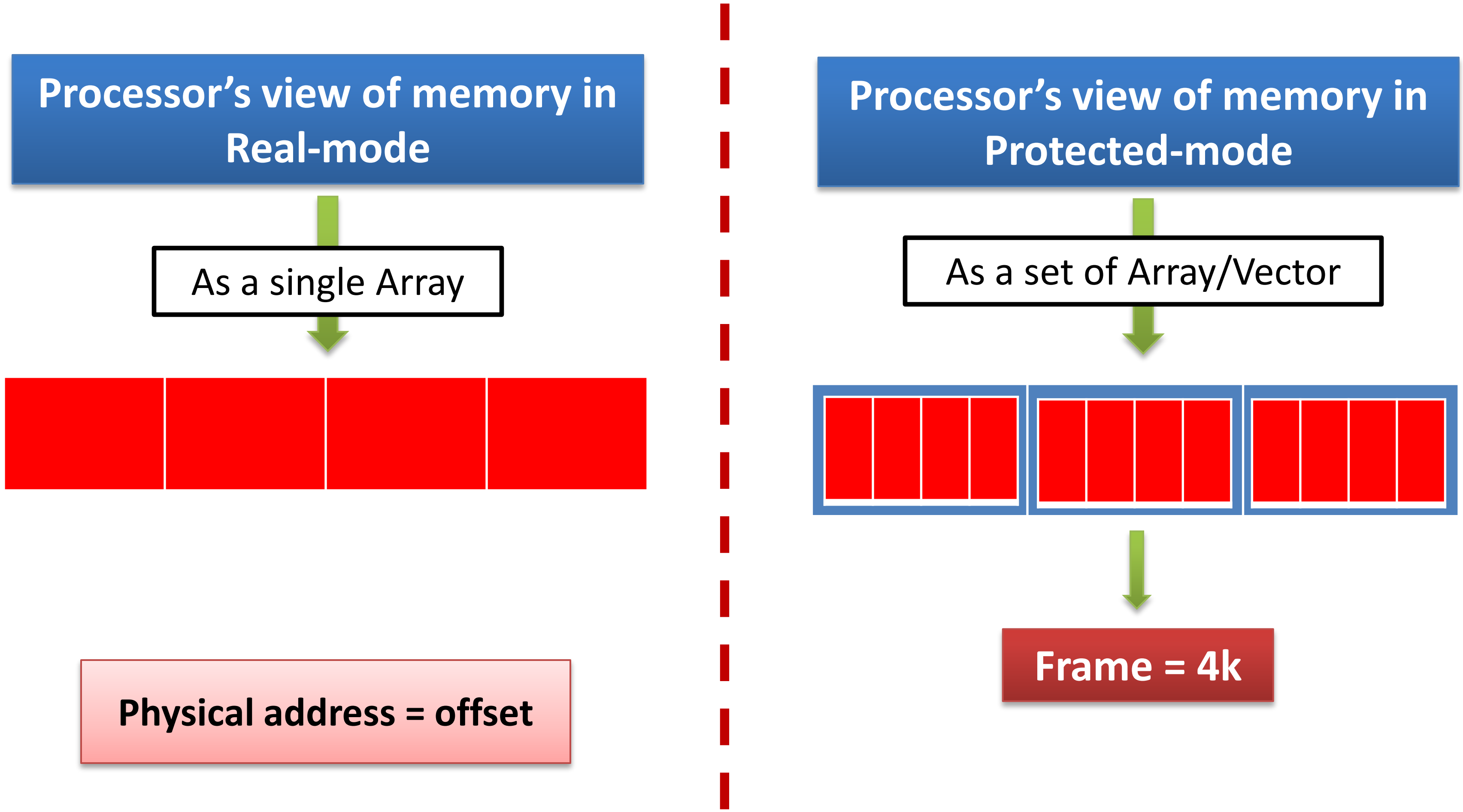
**Physical address = offset**

**Processor's view of memory in Protected-mode**

As a set of Array/Vector



# Memory View


Processor's view of memory in Real-mode
As a single Array
Physical address = offset
Processor's view of memory in Protected-mode
As a set of Array/Vector
Frame = 4k

# Memory View

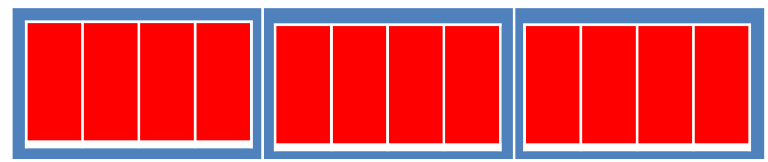
**Processor's view of memory in Real-mode**

As a single Array

**Physical address = offset**

**Processor's view of memory in Protected-mode**

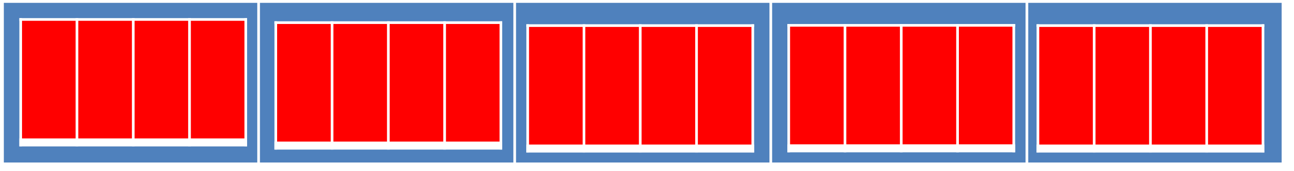
As a set of Array/Vector



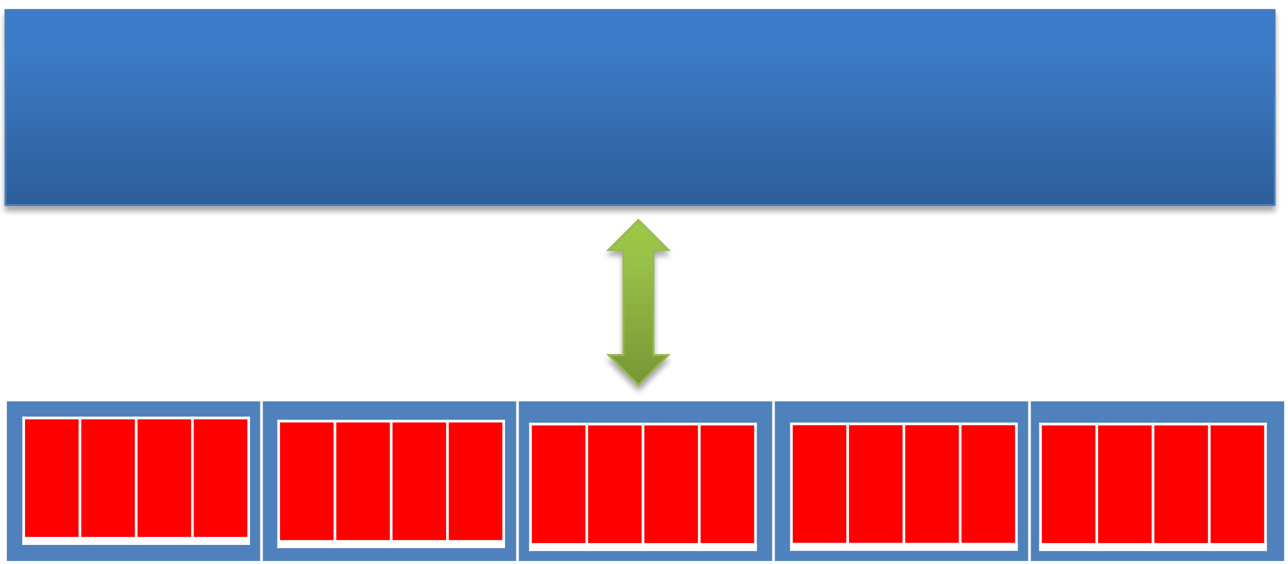
**Frame = 4k**

**Physical address =  Frame no. + offset**

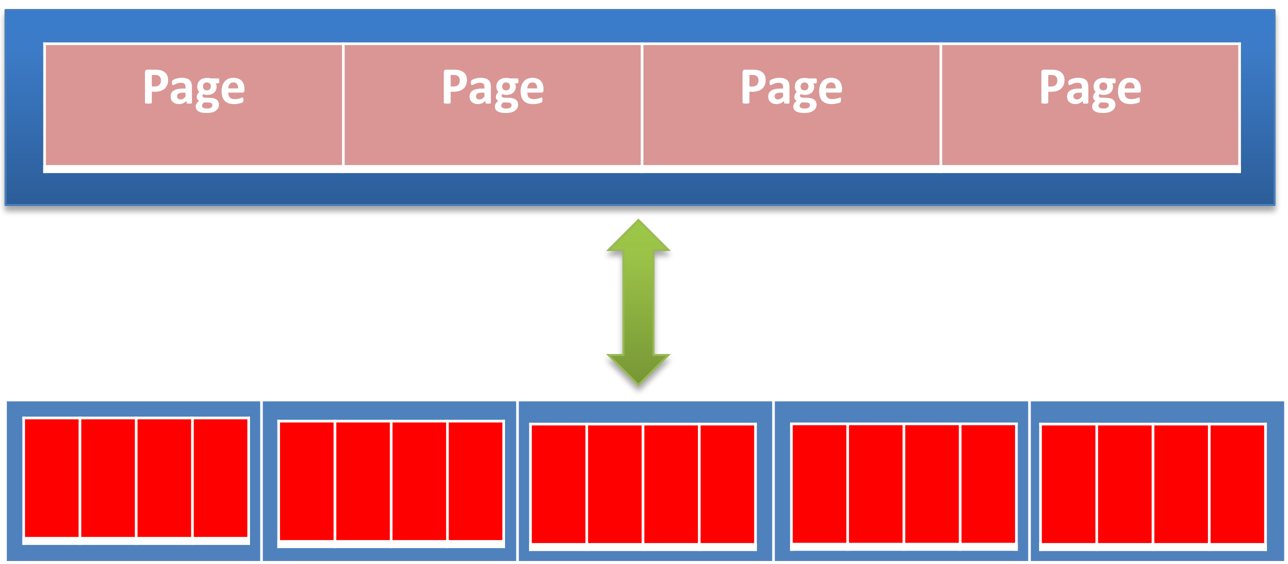
# Memory View



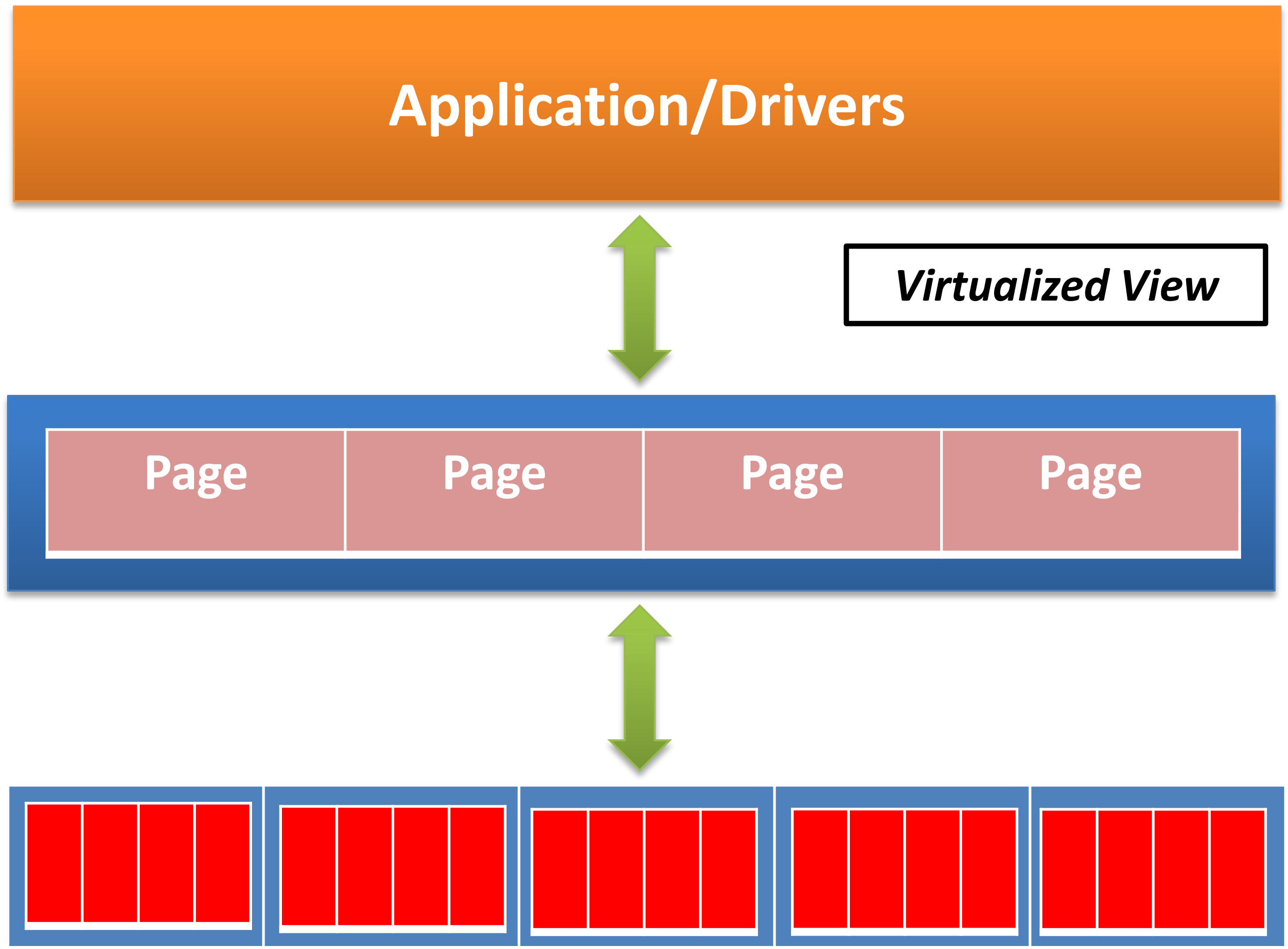
# Memory View



# Memory View



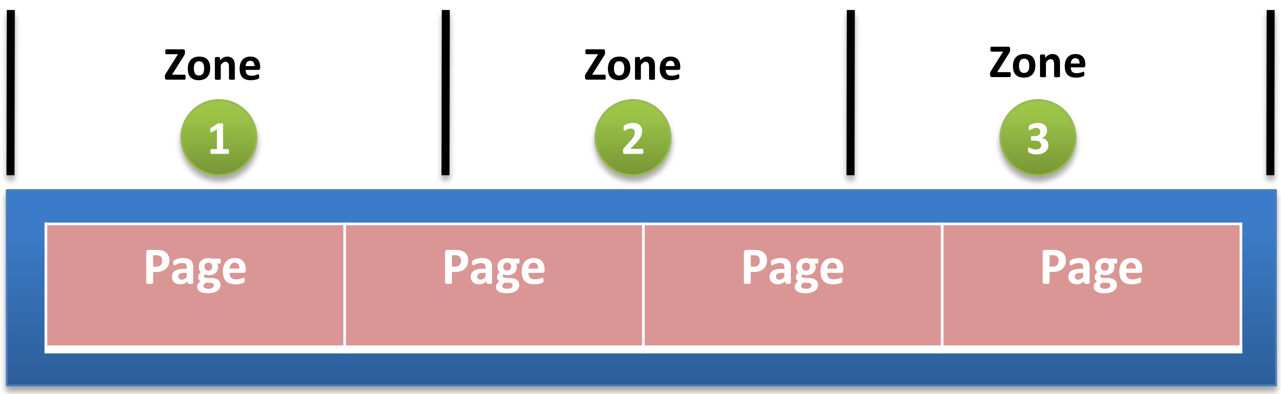
# Memory View



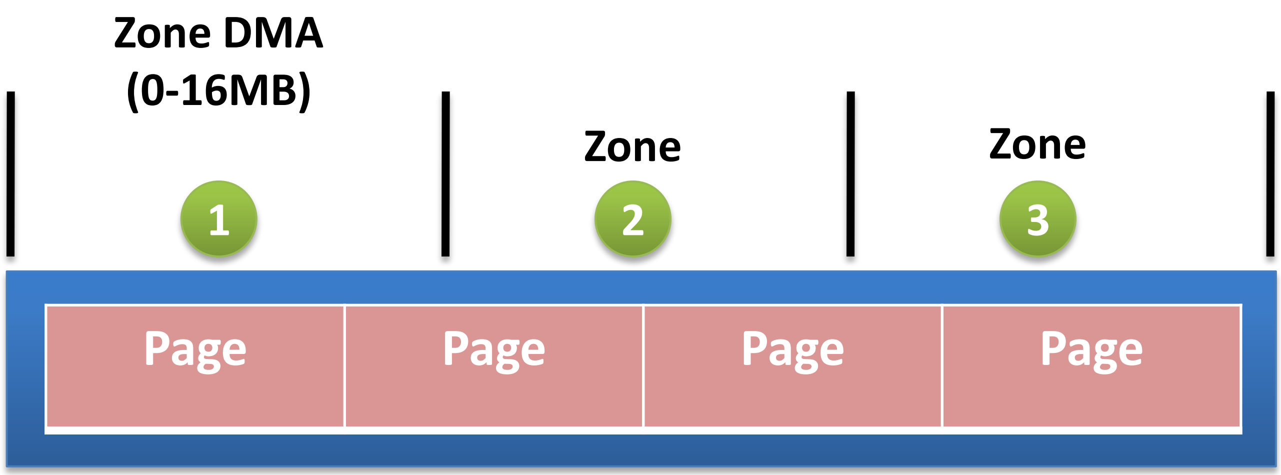
# Memory Manager Overview

| | | | |
|---|---|---|---|
| Page | Page | Page | Page |

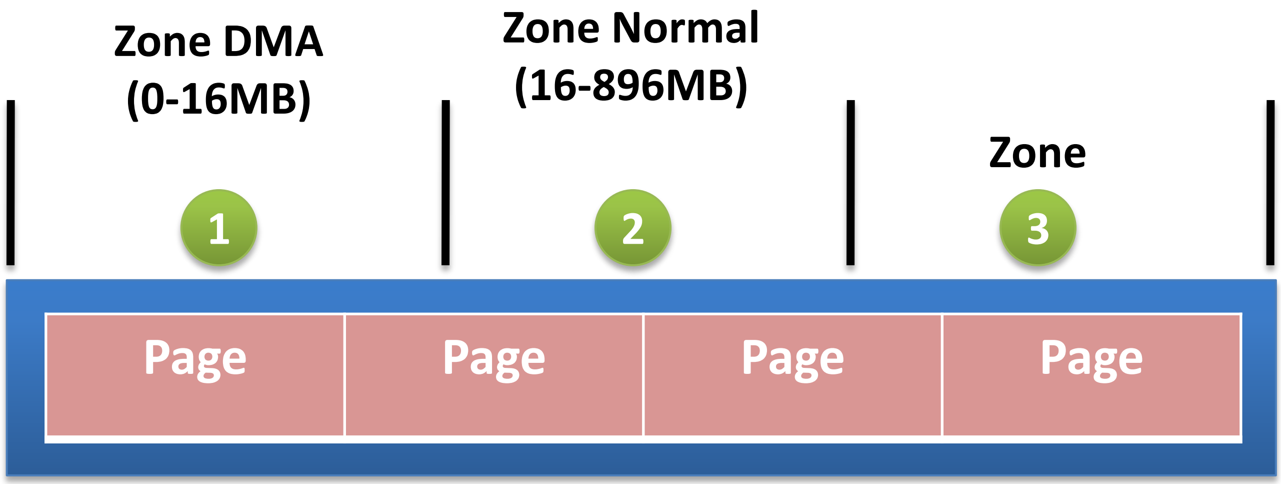
# Memory Manager Overview



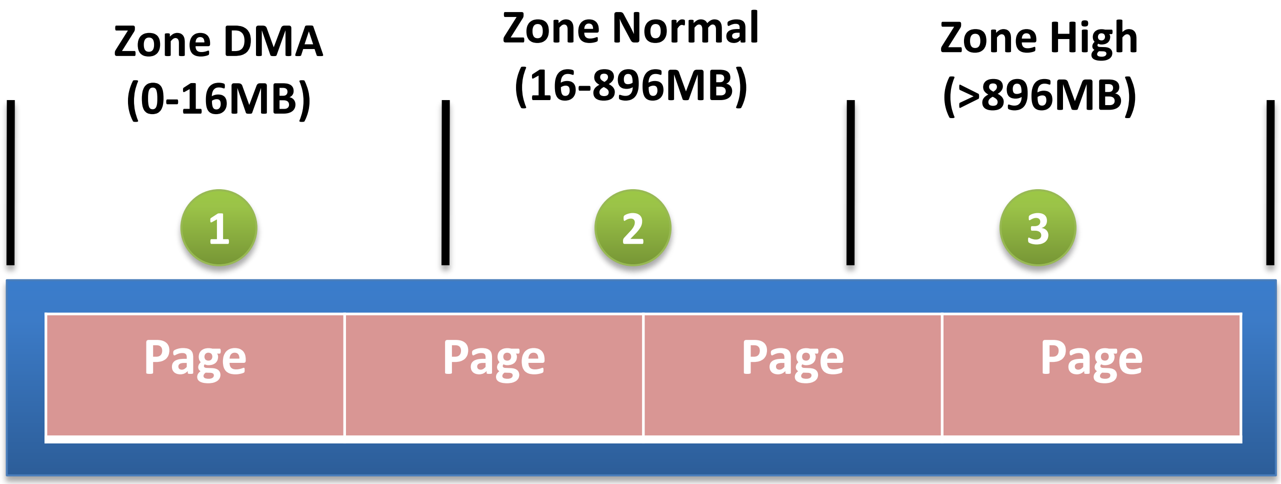
# Memory Manager Overview



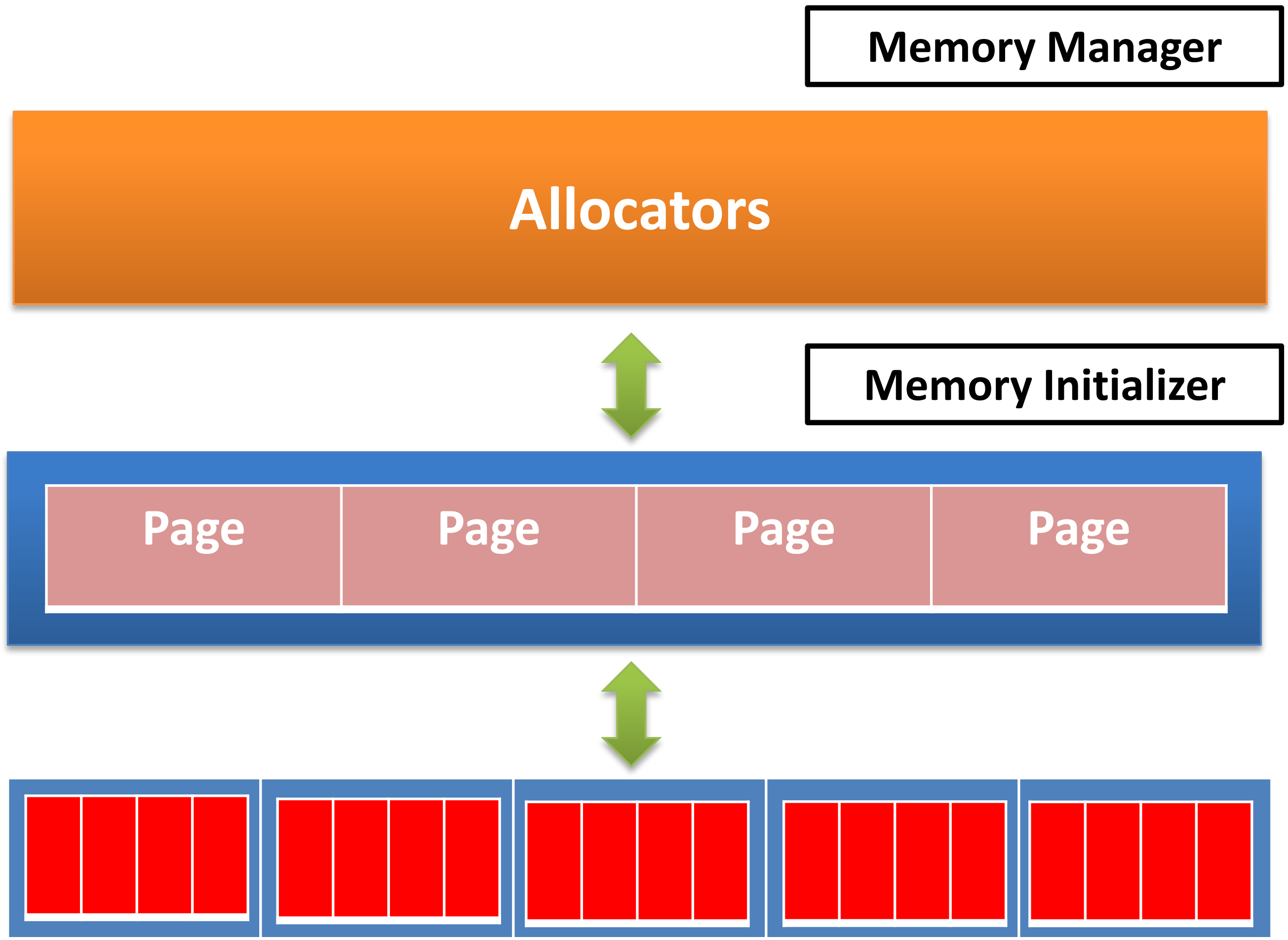
# Memory Manager Overview



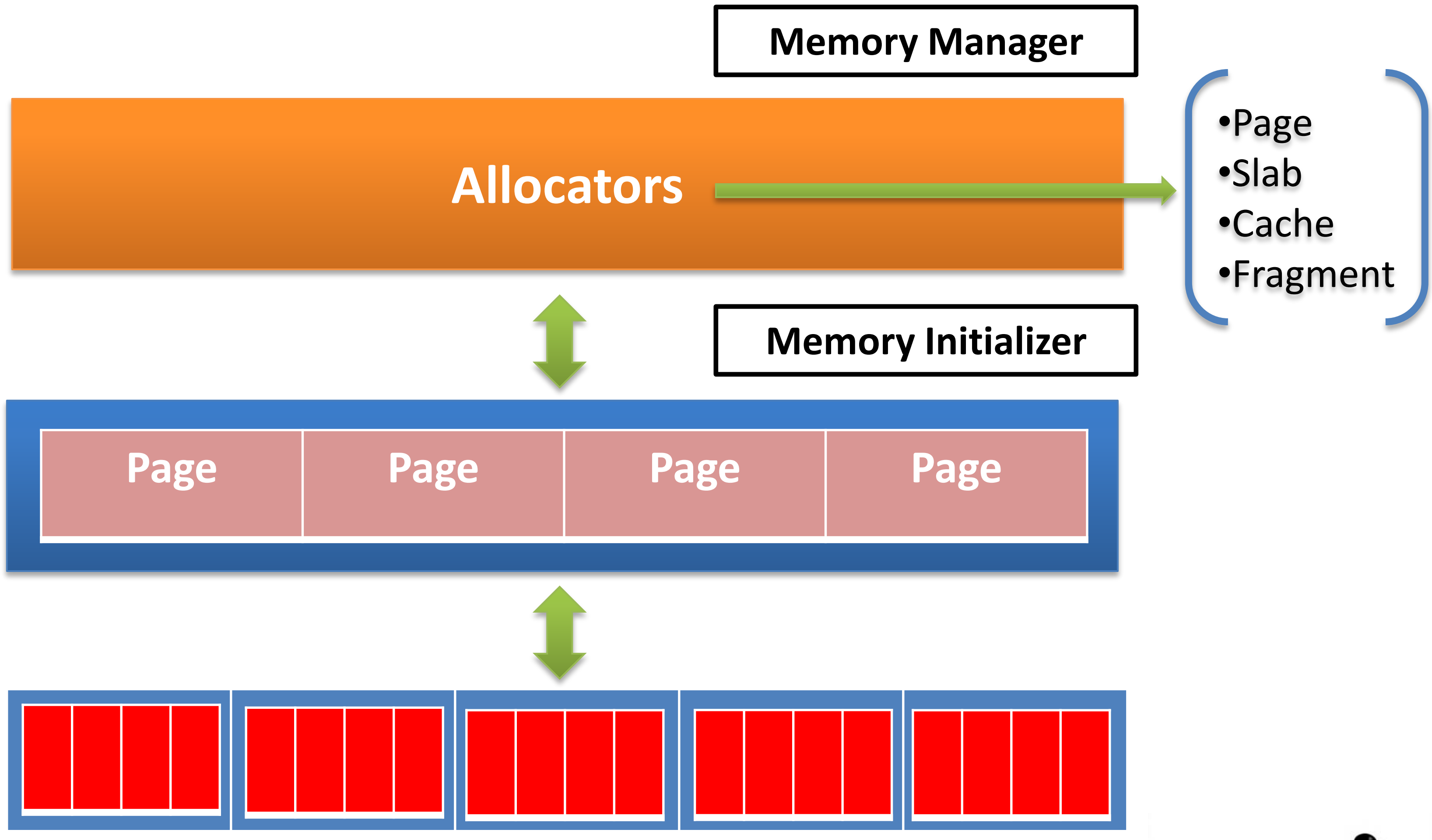
# Memory Manager Overview



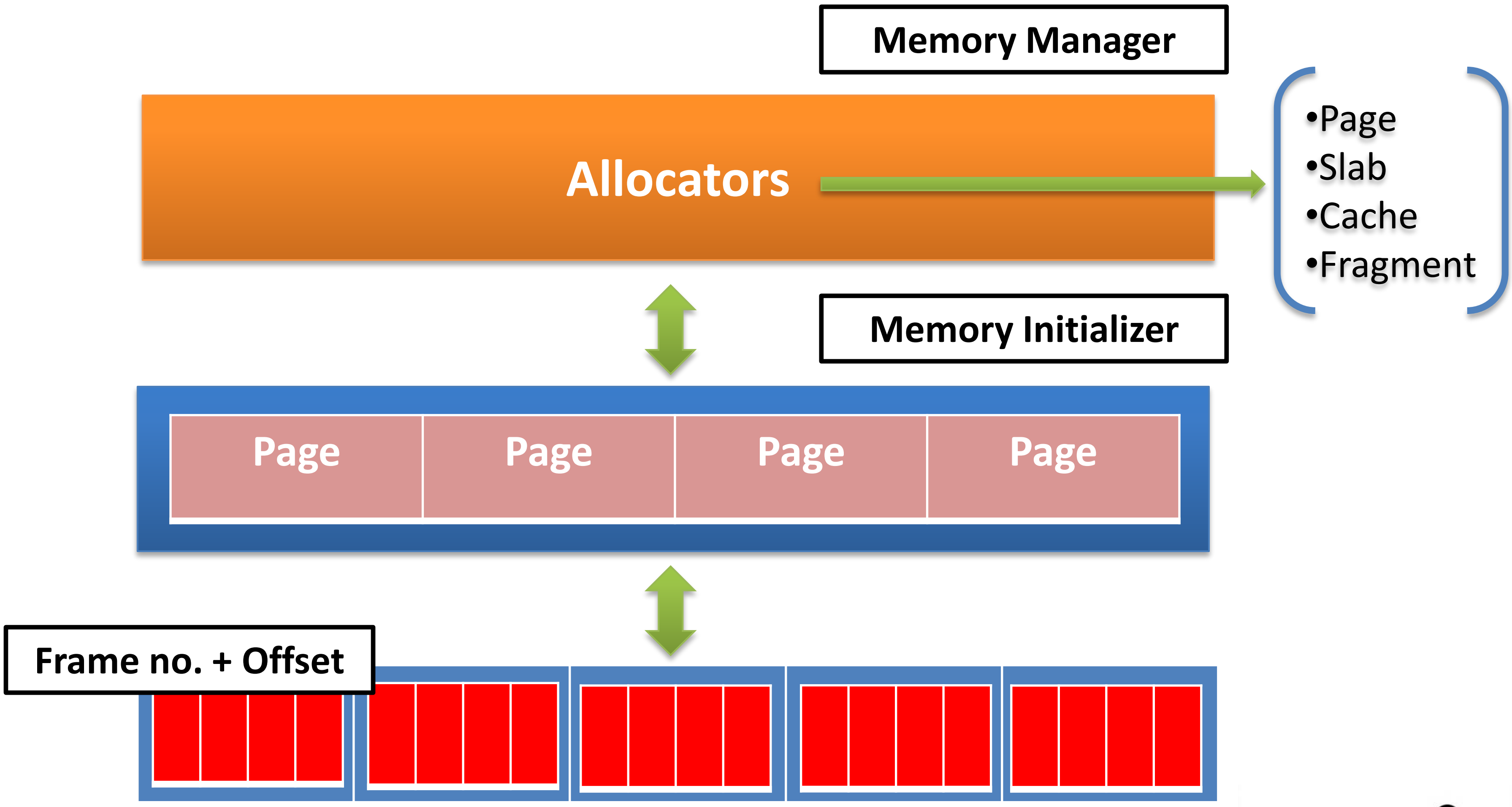
# Memory View


Memory Manager
Allocators
Memory Initializer
Page
Page
Page
Page

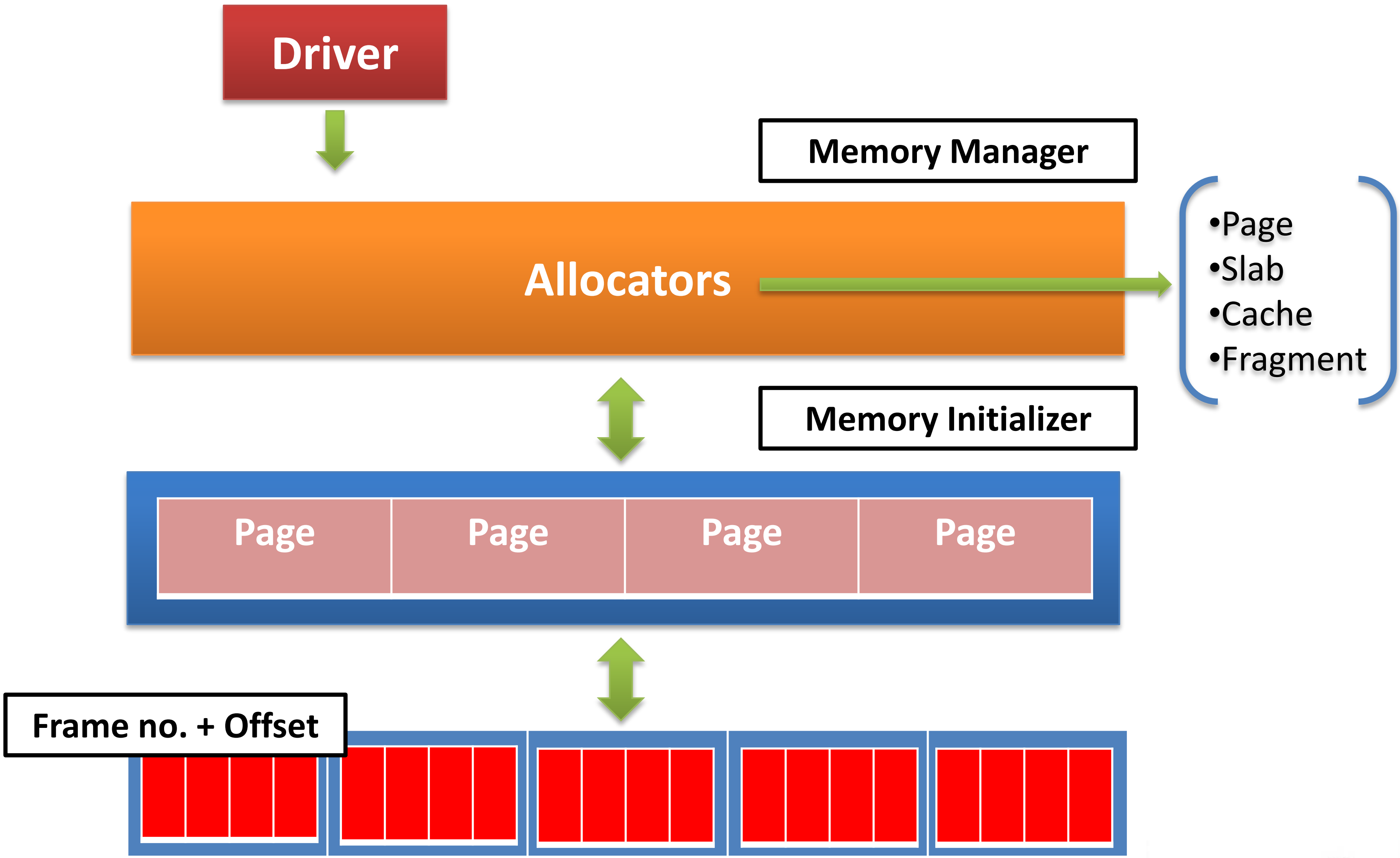
# Memory View


Memory Manager
Allocators
•Page
•Slab
•Cache
•Fragment
Memory Initializer
Page
Page
Page
Page

# Memory View

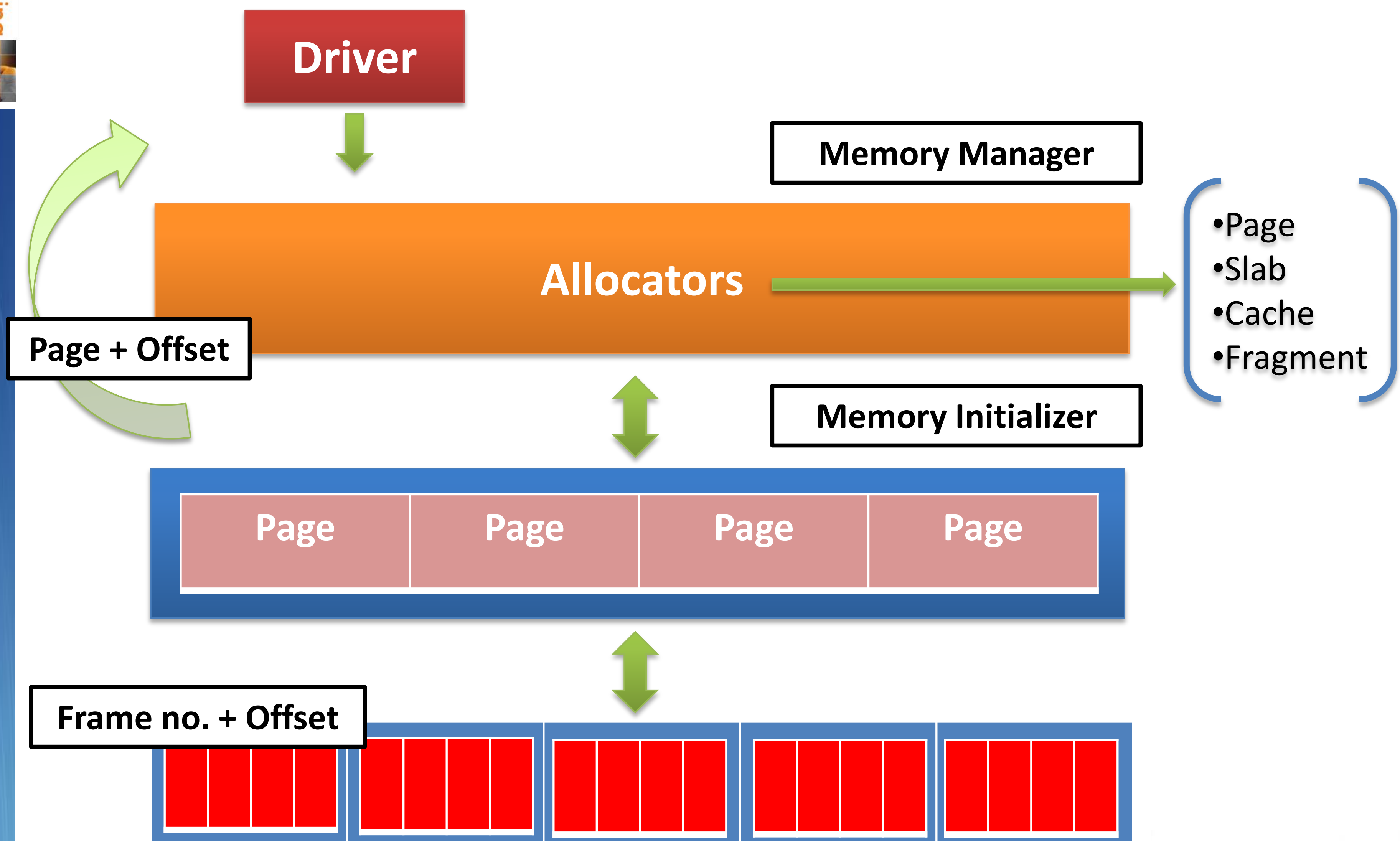

Memory Manager
Allocators
•Page
•Slab
•Cache
•Fragment
Memory Initializer
Page
Page
Page
Page
Frame no. + Offset

# Memory View



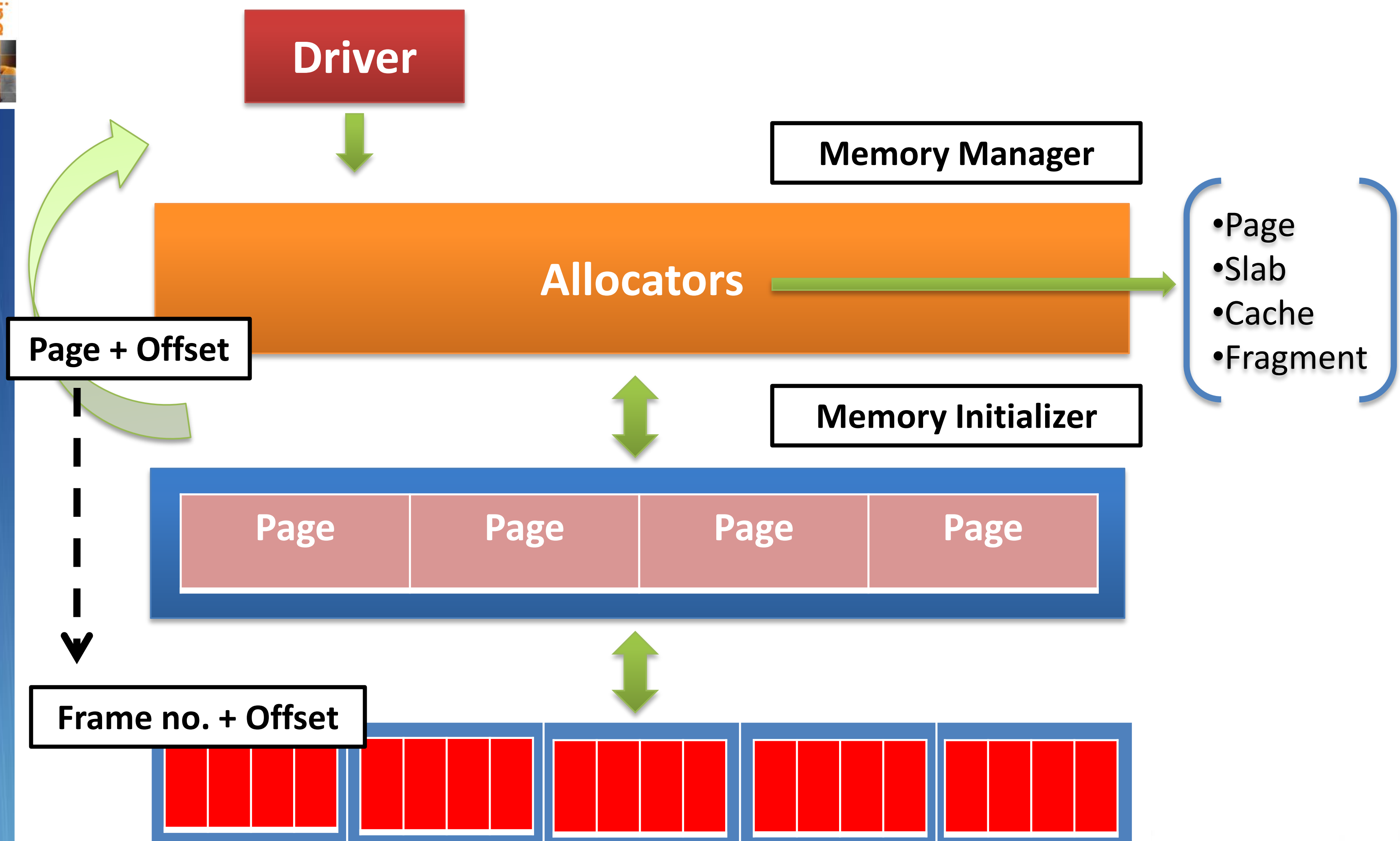
Driver
Memory Manager
Allocators
•Page
•Slab
•Cache
•Fragment
Memory Initializer
Page
Page
Page
Page
Frame no. + Offset

# Memory View

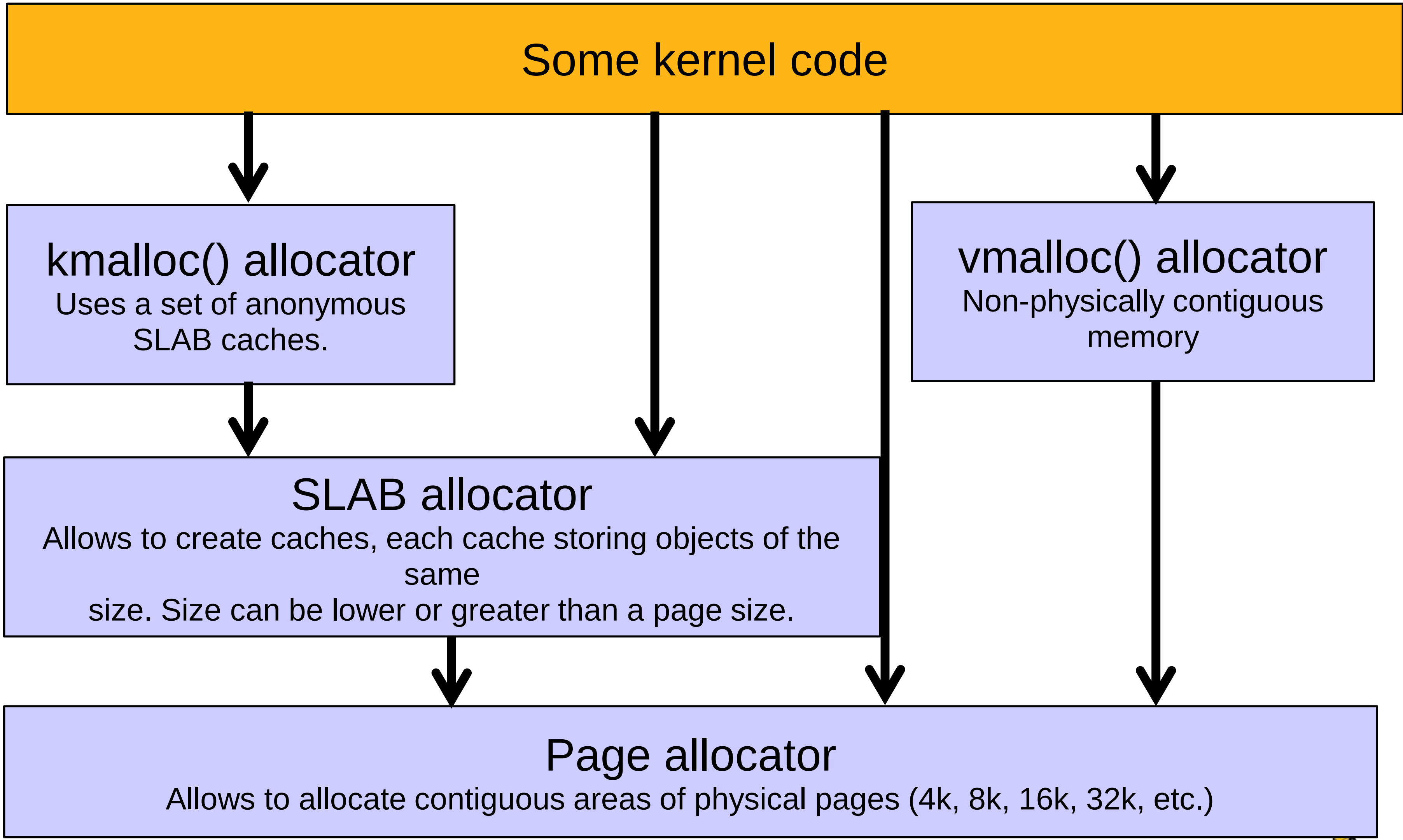


Driver
Memory Manager
Allocators
•Page
•Slab
•Cache
•Fragment
Page + Offset
Memory Initializer
Page
Page
Page
Page
Frame no. + Offset

# Memory View


Driver
Memory Manager
Allocators
•Page
•Slab
•Cache
•Fragment
Page + Offset
Memory Initializer
Page
Page
Page
Page
Frame no. + Offset

# Allocators in the kernel



# kmalloc allocator

- The kmalloc allocator is the general purpose memory allocator in the Linux kernel, for objects from 8 bytes to 128 KB
- 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 next power of two size (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.
- Should be used as the primary allocator unless there is a strong reason to use another one.

# kmalloc API

- #include <linux/slab.h>
  - void *kmalloc(size_t size, int flags);
    Allocate size bytes, and return a pointer to the area (virtual address)
        size: number of bytes to allocate
        flags: same flags as the page allocator
  - void kfree (const void *objp);
    Free an allocated area
- Example: (drivers/infiniband/core/cache.c)

```
struct ib_update_work *work;
work = kmalloc(sizeof *work, GFP_ATOMIC);
...
kfree(work);
```

# priority flags

The most common ones are:

- GFP_KERNEL

Standard kernel memory allocation. The allocation may block in order to find enough available memory. Fine for most needs, except in interrupt handler context.

- GFP_ATOMIC

RAM allocated from code which is not allowed to block (interrupt handlers or critical sections). Never blocks, allows to access emegency pools, but can fail if no free memory is readily available.

- GFP_DMA

Allocates memory in an area of the physical memory usable for DMA transfers.

- Others are defined in include/linux/gfp.h (GFP: __get_free_pages).

# kmalloc API (2)

- void *kzalloc(size_t size, gfp_t flags);
  Allocates a zero-initialized buffer

- void *kcalloc(size_t n, size_t size, gfp_t flags);
Allocates memory for an array of n elements of size size, and zeroes its contents.

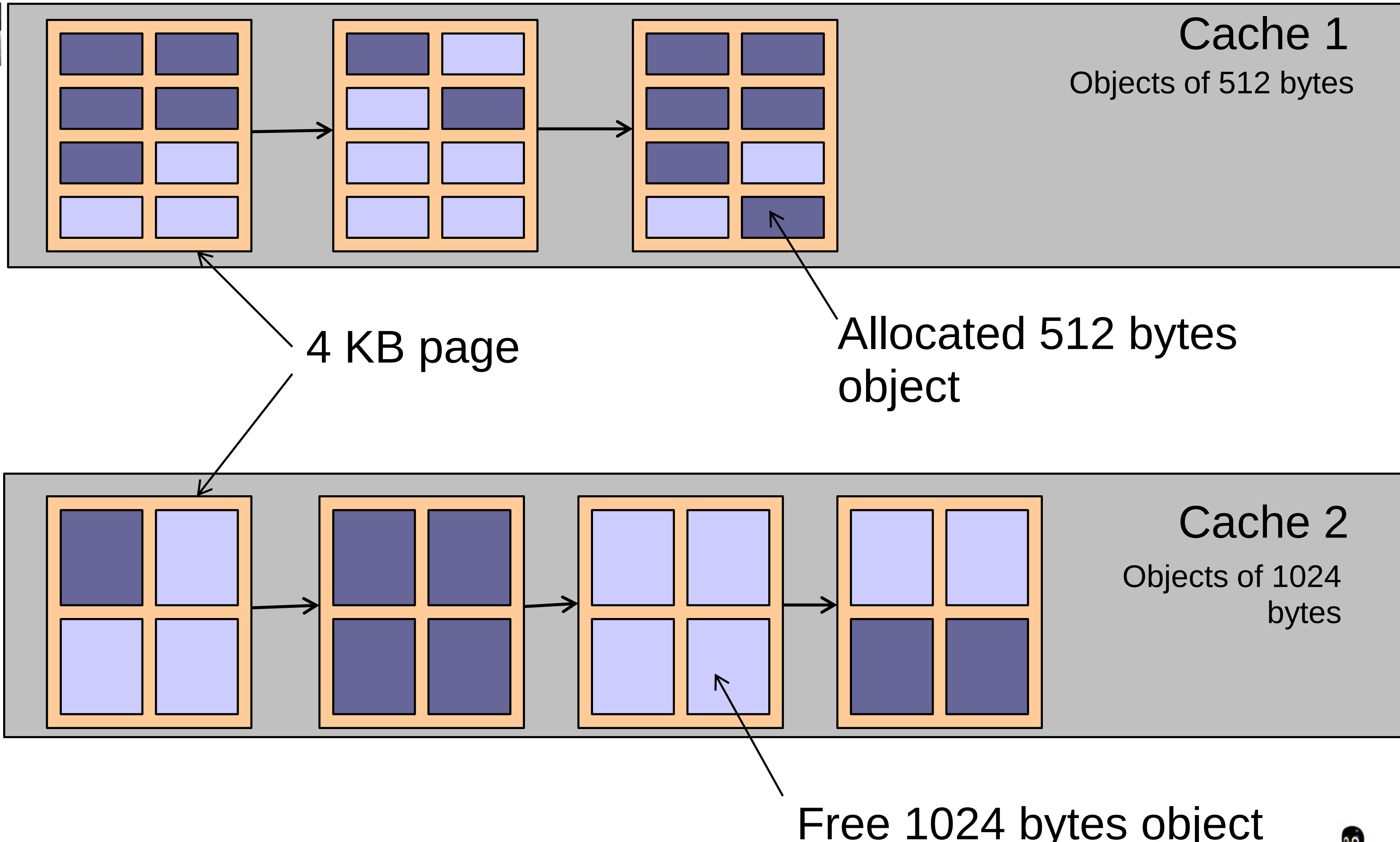
- void *krealloc(const void *p, size_t new_size,gfp_t flags);

- Changes the size of the buffer pointed by p to new_size, by reallocating a new buffer and copying the data, unless the new_size fits within the alignment of the existing buffer.

# SLAB allocator

- The SLAB allocator allows to create caches, which contains a set of objects of the same size
- The object size can be smaller or greater than the page size
- The SLAB allocator takes care of growing or reducing the size of the cache as needed, depending on the number of allocated objects.
- SLAB caches are used for data structures that are frequently needed in the kernel: directory entries, file objects, network packet descriptors, process descriptors, etc.

# SLAB allocator (2)



# Lookaside Caches (Slab Allocator)

## To create a cache for a tailored size

```
#include <linux/slab.h>

kmem_cache_t *
kmem_cache_create(const char *name, size_t size,
                  size_t offset, unsigned long flags,
                  void (*constructor) (void *, kmem_cache_t *,
                                       unsigned long flags),
                  void (*destructor) (void *, kmem_cache_t *,
                                      unsigned long flags));
```

# Lookaside Caches (Slab Allocator)

- ✓ **`name`**: memory cache identifier
  - Allocated string without blanks
- ✓ **`size`**: allocation unit
- ✓ **`offset`**: starting offset in a page to align memory
  - Most likely 0

# Lookaside Caches (Slab Allocator)

- `flags`: control how the allocation is done
  - `SLAB_NO_REAP`
    - Prevents the system from reducing this memory cache (normally a bad idea)
    - Obsolete
  - `SLAB_HWCACHE_ALIGN`
    - Requires each data object to be aligned to a cache line
    - Good option for frequently accessed objects on SMP machines
- Potential fragmentation problems

# Lookaside Caches (Slab Allocator)

- `SLAB_CACHE_DMA`
  - Requires each object to be allocated in the DMA zone
- See **`mm/slab.h`** for other flags

✓ **`constructor`**: initialize newly allocated objects

✓ **`destructor`**: clean up objects before an object is released

✓ Constructor/destructor may not sleep due to atomic context

# Lookaside Caches (Slab Allocator)

- ✓ To allocate an memory object from the memory cache, call
- ✓ `void *kmem_cache_alloc(kmem_cache_t *cache, int flags);`
  - ✓ **`cache`**: the cache created previously
  - ✓ **`flags`**: same flags for **`kmalloc`**
  - ✓ Failure rate is rather high
    - ✓ Must check the return value
- ✓ To free an memory object, call
- ✓ `void kmem_cache_free(kmem_cache_t *cache, const void *obj);`

# Lookaside Caches (Slab Allocator)

- To free a memory cache, call
- `int kmem_cache_destroy(kmem_cache_t *cache);`
  - Need to check the return value
  - Failure indicates memory leak
- Slab statistics are kept in `/proc/slabinfo`

# Different SLAB allocators

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

- SLAB: original, well proven allocator in Linux 2.6.
- SLOB: much simpler. More space efficient but doesn't scale well. Saves a few hundreds of KB in small systems (depends on CONFIG_EMBEDDED)
- SLUB: the new default allocator since 2.6.23, simpler than SLAB, scaling much better (in particular for huge systems) and creating less fragmentation.

| Choose SLAB allocator (NEW) | |
|---|---|
| SLAB | SLAB |
| SLUB (Unqueued Allocator) (NEW) | SLUB |
| SLOB (Simple Allocator) | SLOB |

# Page allocator

- Appropriate for large allocations
- A page is usually 4K, but can be made greater in some architectures (sh, mips: 4, 8, 16 or 64K, but not configurable in i386 or arm).
- Buddy allocator strategy, so only allocations of power of two number of pages are possible: 1 page, 2 pages, 4 pages, 8 pages, 16 pages, etc.
- Typical maximum size is 8192 KB, but it might depend on the kernel configuration.
- The allocated area is virtually contiguous (of course), but also physically contiguous. It is allocated in the identity-mapped part of the kernel memory space.
  - This means that large areas may not be available or hard to retrieve due to physical memory fragmentation.

# Page allocator API

- unsigned long get_zeroed_page(int flags);
  Returns the virtual address of a free page, initialized to zero
- unsigned long __get_free_page(int flags);
  Same, but doesn't initialize the contents
- unsigned long __get_free_pages(int flags,
  unsigned int order);
  Returns the starting virtual address of an area of several contiguous pages in physical RAM, with order being log2(<number_of_pages>).Can be computed from the size with the get_order() function.
- void free_page(unsigned long addr);
  Frees one page.
- void free_pages(unsigned long addr,
  unsigned int order);
  Frees multiple pages. Need to use the same order as in allocation.

# vmalloc allocator

- The vmalloc allocator can be used to obtain virtually contiguous memory zones, but not physically contiguous.
- Allocations of fairly large areas is possible, since physical memory fragmentation is not an issue, but areas cannot be used for DMA, as DMA usually requires physically contiguous buffers.
- API in <linux/vmalloc.h>
  - void *vmalloc(unsigned long size);
    Returns a virtual address
  - void vfree(void *addr);

# Acquiring a Dedicated Buffer at Boot Time

- To allocate, call one of these functions

```
#include <linux/bootmem.h>

void *alloc_bootmem(unsigned long size);

/* need low memory for DMA */
void *alloc_bootmem_low(unsigned long size);

/* allocated in whole pages */
void *alloc_bootmem_pages(unsigned long size);
void *alloc_bootmem_low_pages(unsigned long size);
```

# Acquiring a Dedicated Buffer at Boot Time

- ✓ To free, call
- ✓ `void free_bootmem(unsigned long addr, unsigned long size);`
- ✓ Need to link your driver into the kernel
  - ✓ See **`Documentation/kbuild`**

# Kernel memory debugging

Debugging features available since 2.6.31

- Kmemcheck
  Dynamic checker for access to uninitialized memory.
  Only available on x86 so far, but will help to improve architecture independent code anyway.
  See Documentation/kmemcheck.txt for details.
- Kmemleak
  Dynamic checker for memory leaks
  This feature is available for all architectures.
  See Documentation/kmemleak.txt for details.

Both have a significant overhead. Only use them in development!

# Memory/string utilities

- In <linux/string.h>
  - Memory-related: memset, memcpy, memmove, memscan, memcmp, memchr
  - String-related: strcpy, strcat, strcmp, strchr, strrchr, strlen and variants
  - Allocate and copy a string: kstrdup, kstrndup
  - Allocate and copy a memory area: kmemdup
- In <linux/kernel.h>
  - String to int conversion: simple_strtoul, simple_strtol, simple_strtoull, simple_strtoll
  - Other string functions: sprintf, sscanf

# Linked lists

- Convenient linked-list facility in <linux/list.h>
  - Used in thousands of places in the kernel
- Add a struct list_head member to the structure whose instances will be part of the linked list. It is usually named node when each instance needs to only be part of a single list.
- Define the list with the LIST_HEAD macro for a global list, or define a struct list_head element and initialize it with INIT_LIST_HEAD for lists embedded in a structure.
- Then use the list_*() API to manipulate the list
  - Add elements: list_add(), list_add_tail()
  - Remove, move or replace elements: list_del(), list_move(), list_move_tail(), list_replace()
  - Test the list: list_empty()
  - Iterate over the list: list_for_each_*() family of macros

# Driver development

## I/O memory and ports

# Port I/O vs. Memory-Mapped I/O

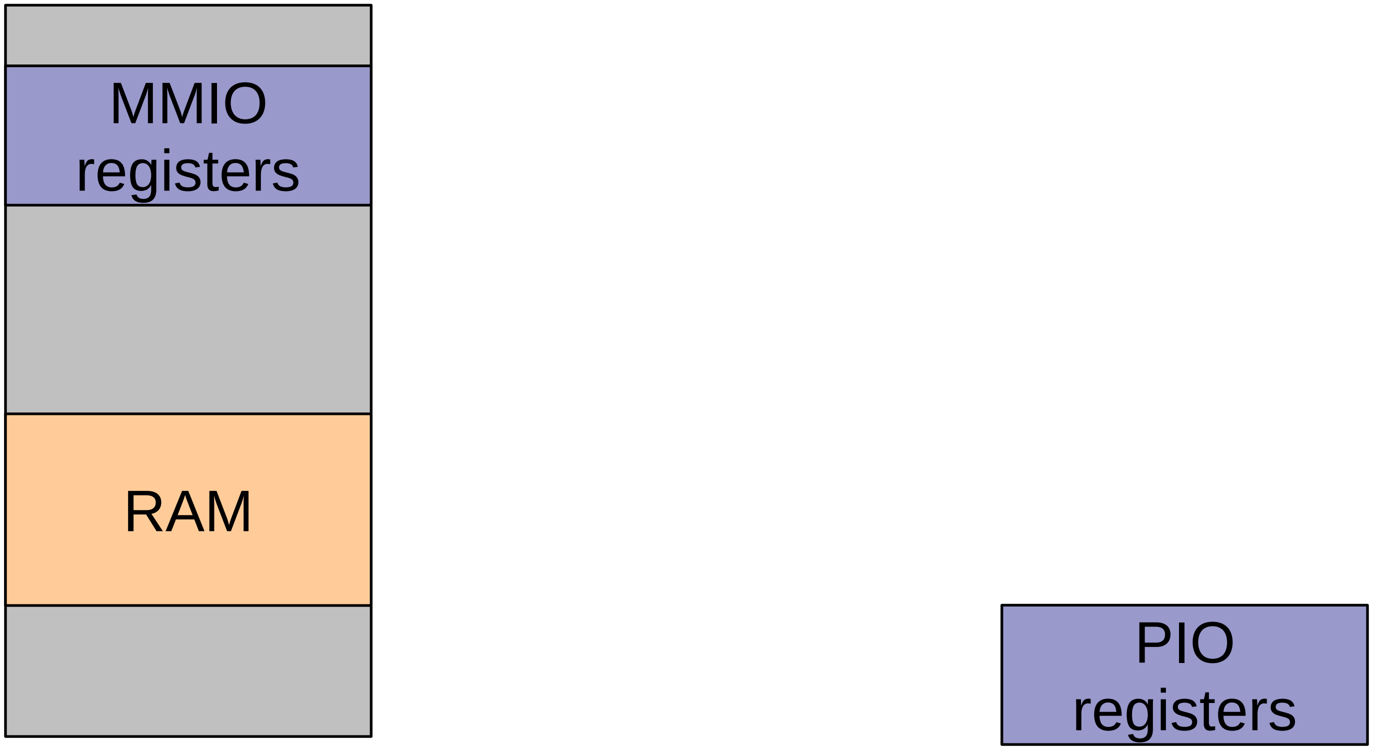
## MMIO

- Same address bus to address memory and I/O devices
- Access to the I/O devices using regular instructions
- Most widely used I/O method across the different architectures supported by Linux

## PIO

- Different address spaces for memory and I/O devices
- Uses a special class of CPU instructions to access I/O devices
- Example on x86: IN and OUT instructions

# MMIO vs PIO




Physical memory address space, accessed with normal load/store instructions

Separate I/O address space, accessed with specific CPU instructions

# Requesting I/O ports

**/proc/ioports example (x86)**

0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0060-006f : keyboard
0070-0077 : rtc
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
0100-013f : pcmcia_socket0
0170-0177 : ide1
01f0-01f7 : ide0
0376-0376 : ide1
0378-037a : parport0
03c0-03df : vga+
03f6-03f6 : ide0
03f8-03ff : serial
0800-087f : 0000:00:1f.0
0800-0803 : PM1a_EVT_BLK
0804-0805 : PM1a_CNT_BLK
0808-080b : PM_TMR
0820-0820 : PM2_CNT_BLK
0828-082f : GPE0_BLK
...

- Tells the kernel which driver is using which I/O ports
- Allows to prevent other drivers from using the same I/O ports, but is purely voluntary.
- struct resource *request_region(
  unsigned long start,
  unsigned long len,
  char *name);

Tries to reserve the given region and returns NULL if unsuccessful.
request_region(0x0170, 8, "ide1");

- void release_region(
  unsigned long start,
  unsigned long len);

# Accessing I/O ports

- Functions to read/write bytes (b), word (w) and longs (l) to I/O ports:

*unsigned* in[bwl](unsigned long *addr);
void out[bwl](*unsigned* port, unsigned long *addr);

- And the strings variants: often more efficient than the corresponding C loop, if the processor supports such operations!

void ins[bwl](*unsigned* port, void *addr,unsigned long count);
void outs[bwl](*unsigned* port, void *addr, unsigned long count);

- Examples
  - read 8 bits

    oldlcr = inb(baseio + UART_LCR);

  - write 8 bits

    outb(MOXA_MUST_ENTER_ENCHANCE, baseio + UART_LCR);

# Requesting I/O memory

**/proc/iomem example**

```
00000000-0009efff : System RAM
0009f000-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000cffff : Video ROM
000f0000-000fffff : System ROM
00100000-3ffadfff : System RAM
  00100000-0030afff : Kernel code
  0030b000-003b4bff : Kernel data
3ffae000-3fffffff : reserved
40000000-400003ff : 0000:00:1f.1
40001000-40001fff : 0000:02:01.0
  40001000-40001fff : yenta_socket
40002000-40002fff : 0000:02:01.1
  40002000-40002fff : yenta_socket
40400000-407fffff : PCI CardBus #03
40800000-40bfffff : PCI CardBus #03
40c00000-40ffffff : PCI CardBus #07
41000000-413fffff : PCI CardBus #07
a0000000-a0000fff : pcmcia_socket0
a0001000-a0001fff : pcmcia_socket1
e0000000-e7ffffff : 0000:00:00.0
e8000000-efffffff : PCI Bus #01
  e8000000-efffffff : 0000:01:00.0
...
```

- Functions equivalent to request_region() and release_region()
- struct resource * request_mem_region(
  unsigned long start,
  unsigned long len,
  char *name);
- void release_mem_region(
  unsigned long start,
  unsigned long len);

# Ioremap

- Load/store instructions work with virtual addresses
- To access I/O memory, drivers need to have a virtual address that the processor can handle, because I/O memory is not mapped by default in virtual memory.
- The ioremap functions satisfy this need:

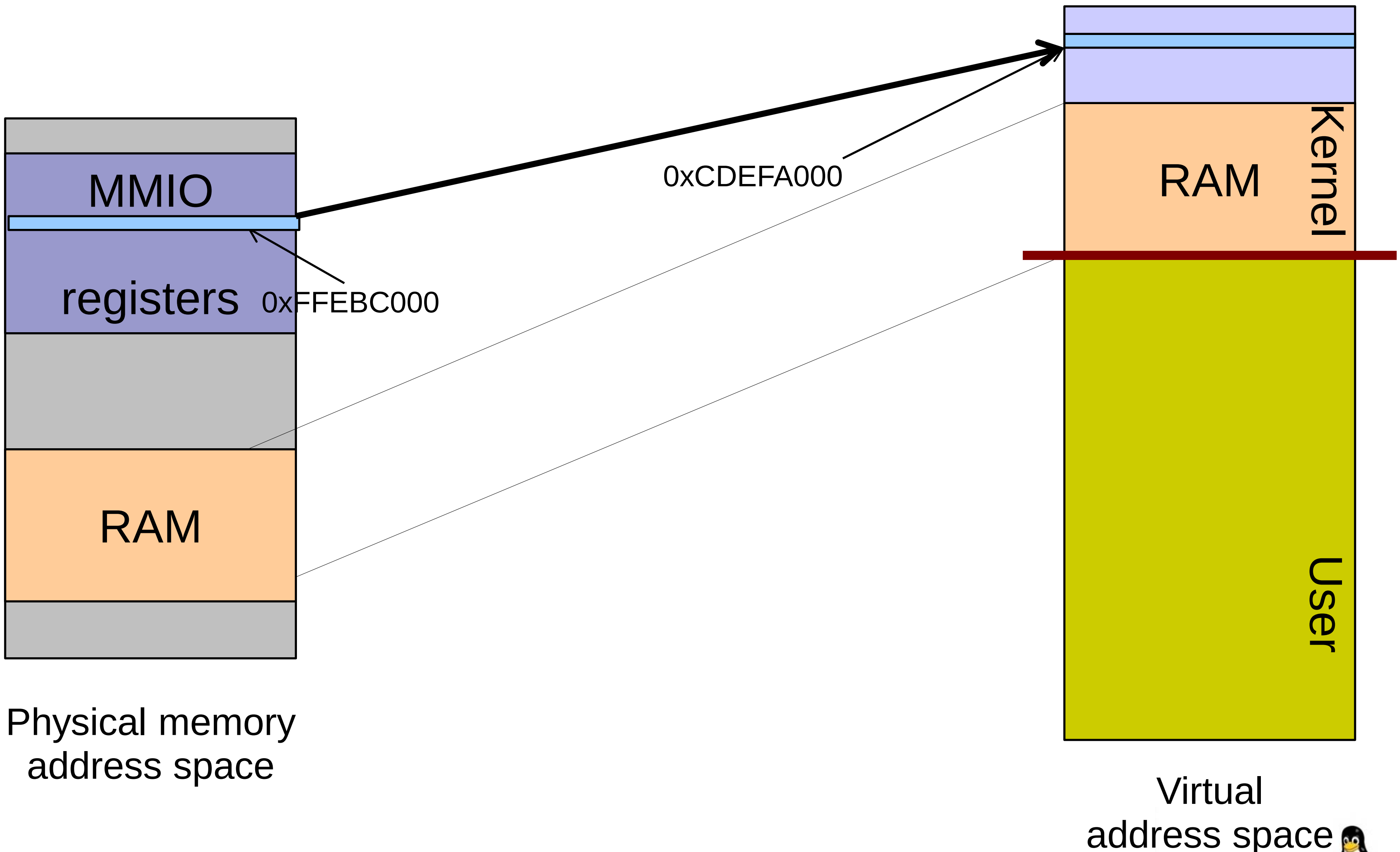
```
#include <asm/io.h>;

void *ioremap(unsigned long phys_addr,
              unsigned long size);
void iounmap(void *address);
```

- Caution: check that ioremap doesn't return a NULL address!

# ioremap()

ioremap(0xFFEBC00, 4096) = 0xCDEFA000



# Accessing MMIO devices

- Directly reading from or writing to addresses returned by ioremap ("pointer dereferencing") may not work on some architectures.
- To do PCI-style, little-endian accesses, conversion being done automatically
unsigned read[bwl](*void* *addr);
void write[bwl](*unsigned val*, *void* *addr);
- To do raw access, without endianess conversion
unsigned __raw_read[bwl](void *addr);
void __raw_write[bwl](unsigned val, void *addr);
  - Example
    - 32 bits write
```
__raw_writel(1 << KS8695_IRQ_UART_TX,
                        membase + KS8695_INTST);
```

# New API for mixed accesses

- A new API allows to write drivers that can work on either devices accessed over PIO or MMIO. A few drivers use it, but there doesn't seem to be a consensus in the kernel community around it.
- Mapping
  - For PIO: ioport_map() and ioport_unmap(). They don't really map, but they return a special cookie.
  - For MMIO: ioremap() and iounmap(). As usual.
- Access, works both on addresses returned by ioport_map() and ioremap()
  - ioread[8/16/32]() and iowrite[8/16/32] for single access
  - ioread_rep[8/16/32]() and iowrite_rep[8/16/32]() for repeated accesses

# Avoiding I/O access issues

- Caching on I/O ports or memory already disabled
- Use the macros, they do the right thing for your architecture
- The compiler and/or CPU can reorder memory accesses, which might cause troubles for your devices is they expect one register to be read/written before another one.
  - Memory barriers are available to prevent this reordering
  - rmb() is a read memory barrier, prevents reads to cross the barrier
  - wmb() is a write memory barrier
  - mb() is a read-write memory barrier
- Starts to be a problem with CPU that reorder instructions and SMP.
- See Documentation/memory-barriers.txt for details

# /dev/mem

- Used to provide user-space applications with direct access to physical addresses.
- Usage: open /dev/mem and read or write at given offset.
  What you read or write is the value
  at the corresponding physical address.
- Used by applications such as the X server to write directly to device memory.
- On x86, arm and tile: CONFIG_STRICT_DEVMEM option to restrict /dev/mem non-RAM addresses, for security reasons (2.6.37-rc2 status).

# Thank You