



# Memory management



# Table of contents

- Physical memory, creating page directories and page tables
  - Process page tables
  - Provisional and final kernel page tables
- Page frame management
  - Page descriptors
  - Zones
- Mapping frames from highmem
  - Persistent Kernel Mapping (PKMAP)
  - Temporary Kernel Mapping (FIXMAP)
  - Managing noncontiguous memory areas (VMALLOC)
- An end to high memory?



# Physical memory

The layout of **physical memory** depends on **architecture**.

At the stage of memory initialization, the kernel builds **a map of physical addresses**, which indicates the **range of physical address used by the kernel** and **inaccessible addresses** (containing BIOS data or occupied by I/O memory mapped to RAM).

Frames treated by the kernel as **reserved**:

- those from the range of **inaccessible** addresses,
- those which contain **code** and **initialized kernel data structures**.

Such page **will never be allocated dynamically or swapped out**.

Usually, the Linux kernel is installed in RAM starting from the physical address 0x00100000 (1 MB), that is **from the second megabyte** (usually this address is set when compiling the kernel). The number of frames required depends on the configuration.

Linux prefers to **skip the first megabyte** to avoid loading into groups of noncontiguous frames.



# Process and kernel page tables

In 32-bit architectures the **linear address space** of the **process** consists of two parts: 0 – 3 GB and 3 GB – 4 GB .

The content of the first **768** entries of the **Page Global Directory** that map addresses lower than PAGE\_OFFSET **depends on the specific process**.

The remaining entries should be the same for all processes and equal to the corresponding entries of the **master kernel Page Global Directory** (but **KPTI!**)

The kernel initializes its own page tables in two phases:

- In the first phase (kernel image loaded into memory, CPU running in **real mode**, **paging not enabled**), kernel creates a **limited address space** including:
  - the kernel's code and data segments,
  - the initial page tables,
  - some amount of memory for dynamic data structures.

This minimal address space is large enough to install the kernel in RAM and to initialize its core data structures.

- In the second phase, kernel takes advantage of **all** existing RAM and sets up the page tables properly.



# Provisional kernel page tables

Let's assume that the kernel's segments, the **provisional page tables**, and the amount needed for dynamic data structures fit in the first **24 MB** of RAM.

In order to map **24 MB** of RAM, **six page tables** are required (one table contains  $2^{10}$  entries, so it can address  $2^{10} * 2^2 * 2^{10} = 4 * 2^{20} = 4 \text{ MB}$ ).

Kernel maps the **linear addresses**:

- from 0x00000000 (address 0) to 0x017fffff ( $2^{23} + 2^{24} - 1 = 8 \text{ MB} + 16 \text{ MB} - 1 = 24 \text{ MB} - 1$ )
- from 0xc0000000 (address  $3 * 2^{30} = 3 \text{ GB}$ ) to 0xc17fffff ( $3 \text{ GB} + 24 \text{ MB} - 1$ )

into the **physical addresses**:

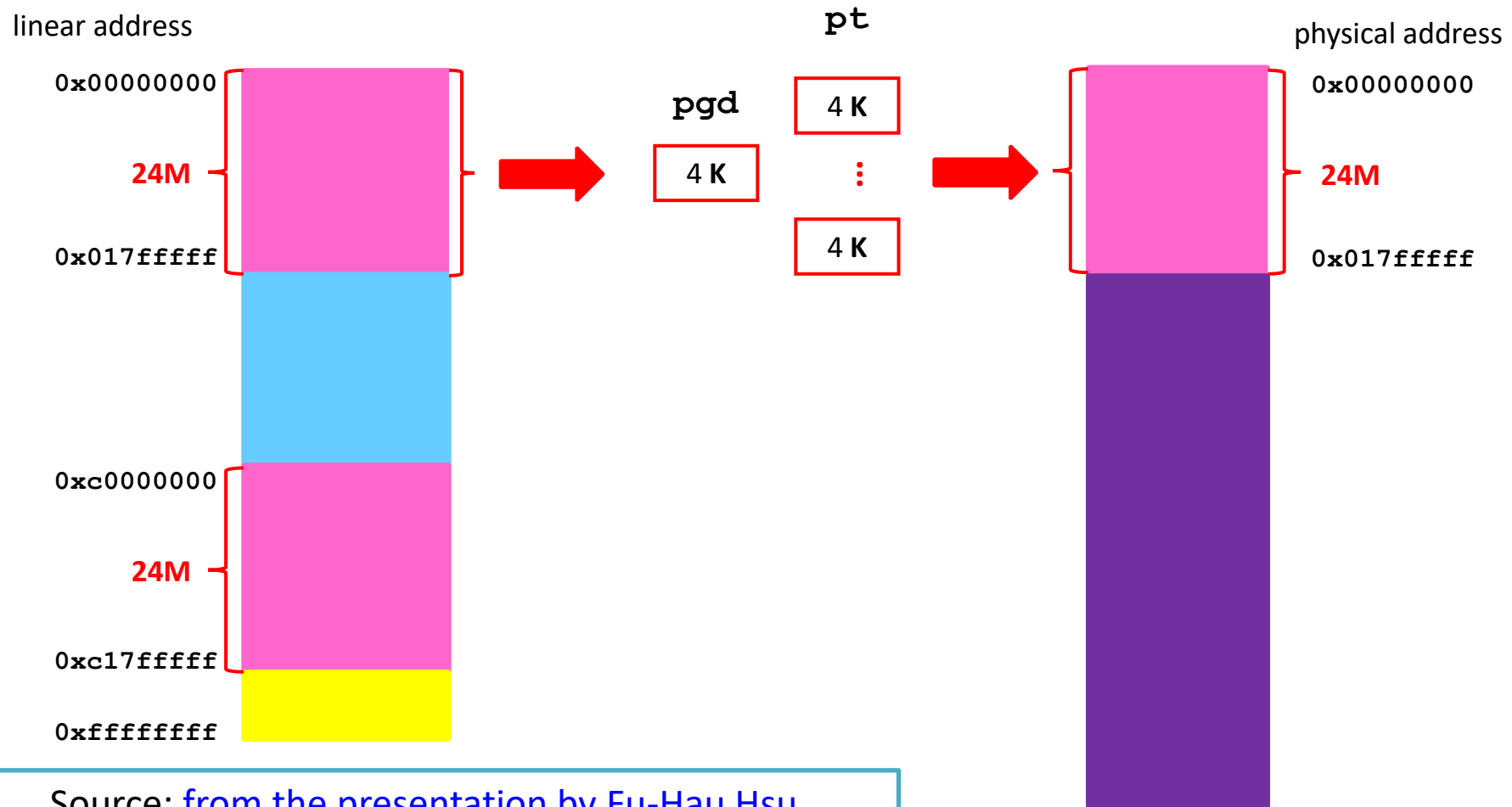
- from 0x00000000 (address 0) to 0x017fffff (24 MB - 1)

The objective of this first phase of paging is to allow these 24 MB of RAM to be **easily addressed both in real mode and protected mode** (using either **physical addresses** from 0 to 24 MB, or **virtual addresses** from 0 to 24 MB or from 3 GB to (3 GB + 24 MB). Physical addresses are obtained from virtual addresses by subtracting PAGE\_OFFSET.

At the end, the kernel **enables paging** by setting the **PG flag** of the **cr0 control registry**.

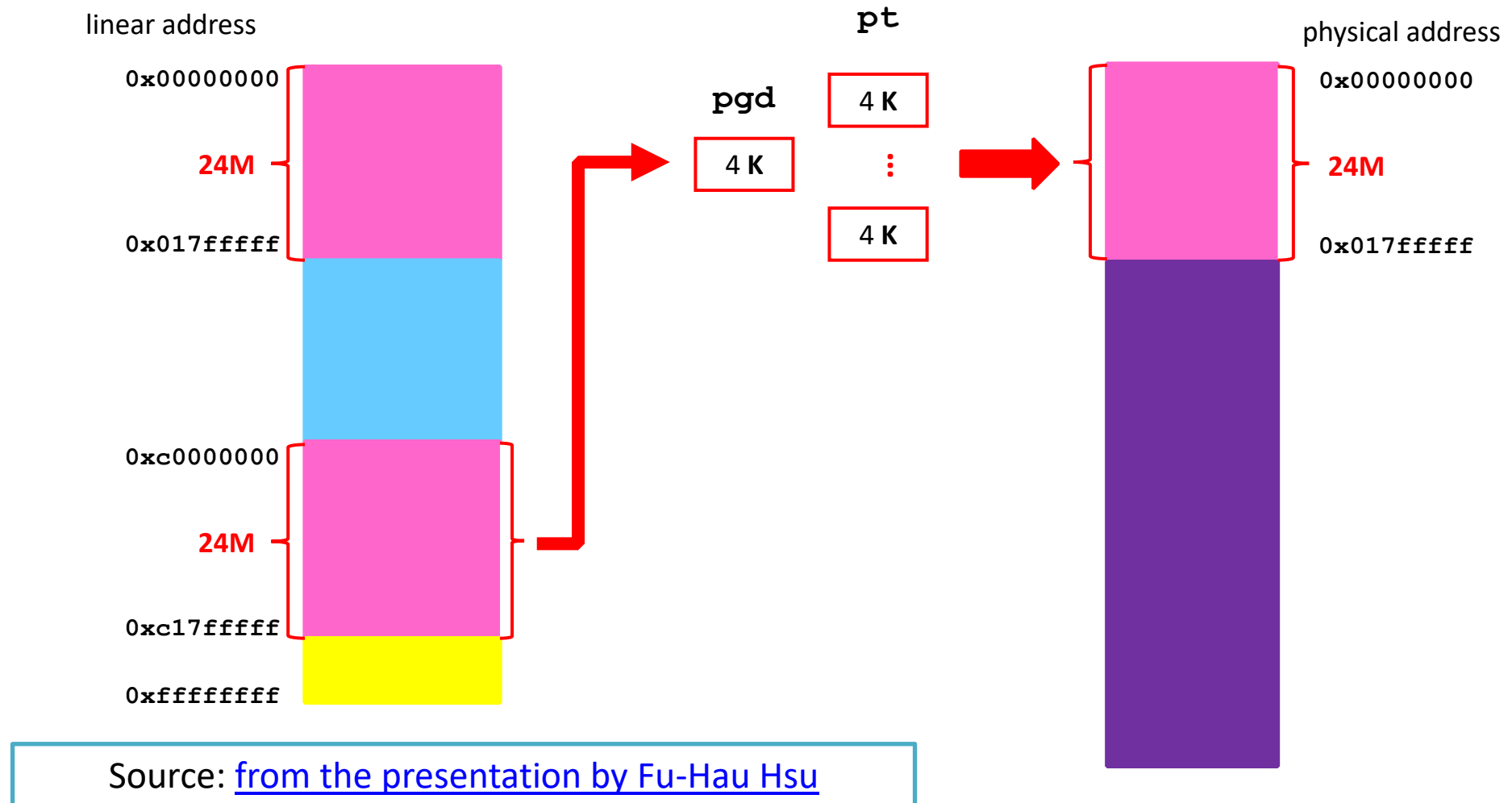
# Provisional kernel page tables

## Mapping Linear Addresses to Physical Addresses in Phase One (1)



# Provisional kernel page tables

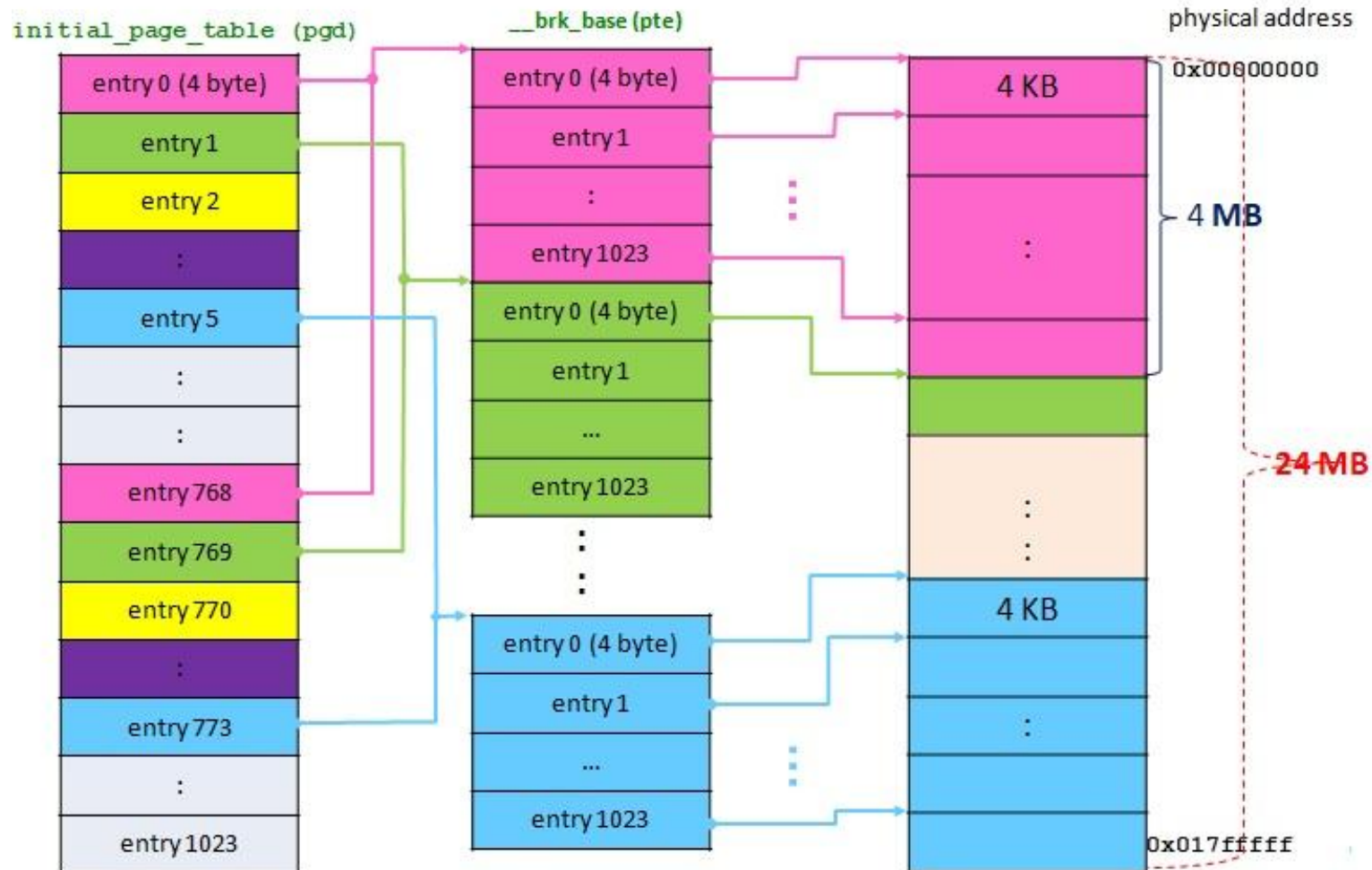
## Mapping Linear Addresses to Physical Addresses in Phase One (2)





# Provisional kernel page tables

## Phase 1: Page Table Layout



Source: [from the presentation by Fu-Hau Hsu](#)





# Final kernel page tables

The purpose of the *second phase* is to fill the kernel page tables so that **all linear addresses** starting with PAGE\_OFFSET are mapped to physical addresses starting from 0.

The mapping depends on the amount of RAM present. Let's assume that the CONFIG\_HIGHMEM flag is enabled. There are three possible cases:

1. RAM size is below 900 MB,
2. RAM size is between 900 MB and 4096 MB,
3. RAM size is more than 4096 MB.

## Case 1

If a computer has less than 900 MB of RAM, 32-bit physical addresses are sufficient to address all the available RAM.

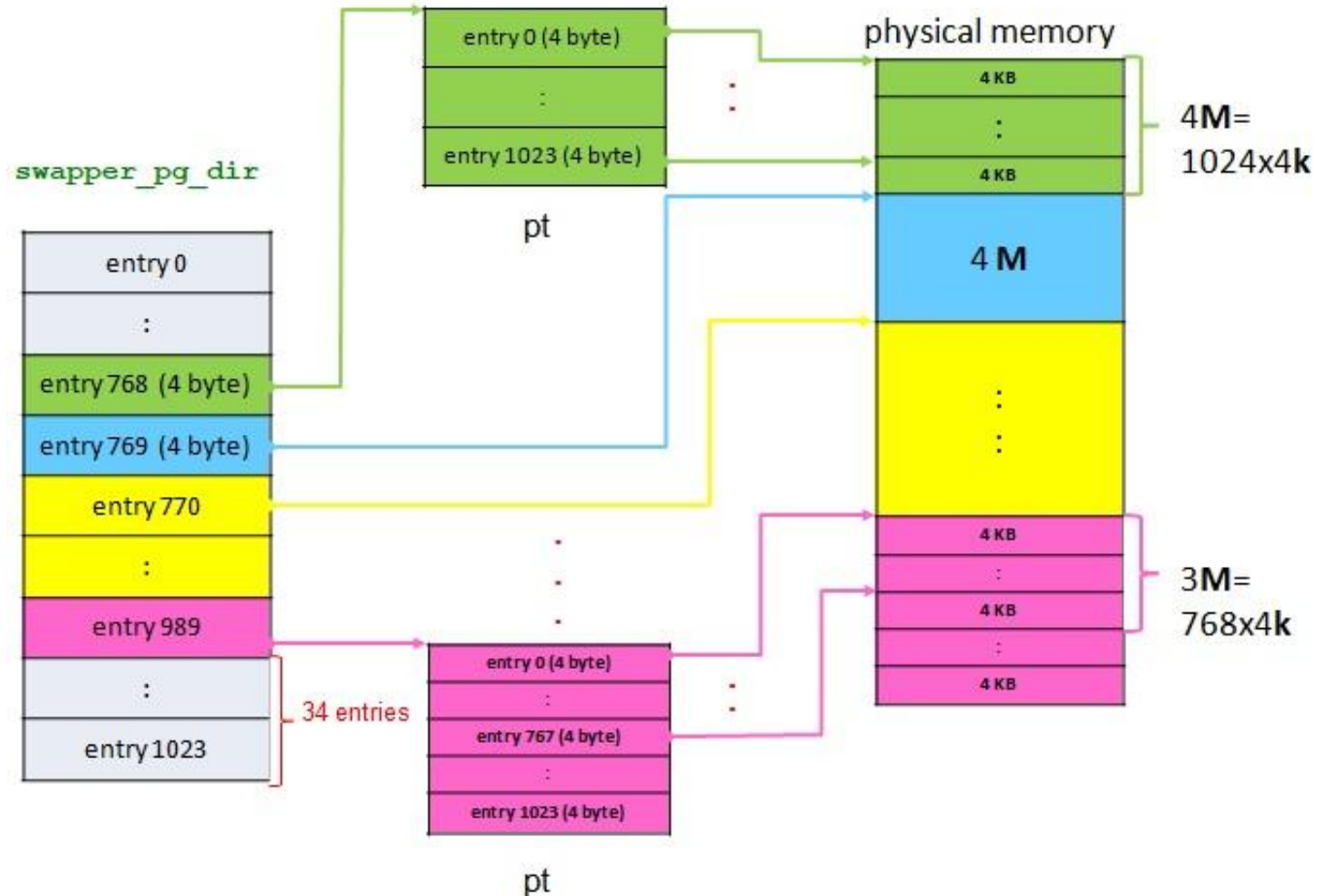
Identity mapping of the first 24 MB RAM is needed only to complete the kernel initialization phase, **after the end of the kernel initialization phase, the corresponding page table entries are reset.**





# Final kernel page tables

## MKPGD Mapping



Source: [from the presentation by Fu-Hau Hsu](#)



# Final kernel page tables

## Case 2

If a computer has more than 900 MB of RAM, but less than 4 GB, RAM cannot be mapped entirely into the kernel linear address space.

- The first 900 MB are mapped as for smaller memories.
- In order for the program to reach more than 900 MB of physical memory, the kernel must **dynamically map certain ranges of linear addresses to the appropriate physical addresses**.

This requires **dynamic changes to the page tables**.

## Case 3

If a computer has more than 4 GB, then the PAE mechanism must be available, the kernel must be compiled with **PAE support** and use **three-level paging** (assuming the addresses are 32-bit).

*How to use the **upper addresses of the four-gigabyte linear space** of the kernel for dynamic mapping will be discussed later.*





# Page descriptors

The RAM physical memory is divided into **page frames**, the frame size is defined by the constant **PAGE\_SIZE**.

In 32-bit protected mode x86 supports two kinds of pages:

- **Normal-sized pages**, which are 4 KB (some architectures allow other values, e.g. 16 KB on ARM64 or 8 KB, 16 KB or 64 KB on IA64).
- **Huge pages**, which are 4 MB (in some architectures that may be 2 MB).

Default page can be checked by executing from shell:

```
getconf PAGE_SIZE
```

State information of a page frame is kept in a **page descriptor** of type **struct page**. All page descriptors are stored in the **mem\_map** array.

```
struct page *mem_map;
```



# Page descriptors

The **struct page** must be as small as possible (this is the reason for many unions in the definition; they are omitted here).

Size – **64 bytes** for every **4KB** memory page.

See file  
**include/linux/mm\_types.h**

```
struct page {
    unsigned long flags;
    union {
        struct { /* Page cache and anonymous pages */
            struct list_head lru;
            struct address_space *mapping;
            pgoff_t index;
            unsigned long private;
            ...
        };
        struct {
            ...
        };
        ...
    };
    atomic_t _mapcount;
    atomic_t _refcount;
    ...
    #if defined(WANT_PAGE_VIRTUAL)
        void *virtual; /* Kernel virtual address (NULL if not kmapped, ie. highmem) */
    ...
};
```



# Page descriptors

Page descriptors are used to describe the **state of physical memory frames**, not their **contents**.

## **struct page** – field **\_refcount**

Page frame's **reference\_counter**. The **page\_count()** function returns the value of this field. The frame can be used by the **page\_cache** (in this case the field **mapping** points to the object **address\_space** associated with this frame), as **private\_data** (then the field **private** points to it) or be mapped from the **process\_page table**. See **include/linux/page\_ref.h**.

## **struct page** – field **\_mapcount**

Number of page table entries that refer to the page frame (-1 if none).

## **struct page** – field **private**

Available to the kernel component that is using the page (for instance, it is a **buffer head pointer** in case of a **buffer\_page**). If the page is **free**, this field is used by the **buddy allocator**.





# Page descriptors

## struct page – field **mapping**

Used when the page is inserted into the **page cache** (points to the `address_space`), or when it belongs to an **anonymous region** (points to `anona_vma`), or used by the **slab allocator** when the page is free.

## struct page – field **index**

Used by several kernel components with different meanings. For instance, it identifies the position of the data stored in the page frame within the page's **disk image** or within an **anonymous region**, or it stores a **swapped-out** page identifier.

## struct page – field **lru**

Contains pointers to the **least recently used doubly linked list of pages**. When the frame is **free**, it is used in the **buddy allocator** to link free areas.

## struct page – field **virtual**

The **kernel virtual address** of the frame or NULL. On machines where all RAM is mapped into kernel address space, we can simply **calculate** the virtual address.

On machines with **highmem** some memory is mapped into kernel virtual memory **dynamically**, so we need a place to store that address.



# Page descriptors – flags

## struct page – field **flags**

Flags in the flags field are **independent** of architecture (unlike bits in **page tables**). Changing and testing the value of this field is usually an atomic operation.

```
enum pageflags {  
    PG_locked,          /* Page is locked. Don't touch. */  
    PG_referenced,  
    PG_uptodate,  
    PG_dirty,  
    PG_lru,  
    PG_active,  
    PG_error,  
    PG_slab,  
    PG_reserved,  
    ...  
};
```



# Struct page flags

## PG\_locked bit

All process pages can participate in **input-output operations**:

- the pages of **files mapped to memory** can be read from the disk,
- the pages of the **memory mapped files that have been modified and are shared** (MAP\_SHARED) may have to be saved to disk,
- **private pages** that have been modified may need to be **swapped** to the swap area and later **reloaded** into memory.

This bit is used to lock a page during the **I/O operation**. It is set before the operation is initiated and cleared after completion.

## PG\_referenced bit

This bit together with the **accessed bit in the page table** is used to manipulate the **page age** and move the page in the **active and inactive lists**.

It is checked and modified in the function responsible for selecting the frame to be released. If the frame has the PG\_referenced bit set, it is reset and the page is not sent back to the swap device.



# Struct page flags

## **PG\_uptodate** bit

This flag is set after completing a read operation, unless a disk I/O error happened.

## **PG\_highmem** bit

The page frame belongs to the **ZONE\_HIGHMEM** zone. The pages with the PG\_highmem bit set are not permanently mapped to the virtual address space of the kernel, they must be mapped dynamically. The **struct page** (these bits with information) are always mapped to the kernel address space (have a linear address).

## **PG\_error** bit

An I/O error occurred while transferring the page.

## **PG\_slab** bit

The page frame is part of slab (under supervision of the **slab allocator**).

## **PG\_reserved** bit

The page frame is reserved for kernel code or is unusable.



# Page descriptors

Extra reading:

- [How many page flags do we really have?](#), Jonathan Corbet, 2009.
- [Transparent huge pages in 2.6.38](#), Jonathan Corbet, January 2011.
- [Cramming more into struct page](#), Jonathan Corbet, August 2013.
- [ZONE\\_DEVICE and the future of struct page](#), Jonathan Corbet, March 2017.
- [Willy's memory-management to-do list](#) – section **Cleaning up struct page** (Jonathan Corbet, April 2018). [A set of diagrams](#) showing how the various fields of **struct page** are used.
- [Repurposing page->mapping](#), Jonathan Corbet, April 2018.
- [Memory: the flat, the discontinuous, and the sparse](#), Mike Rapoport, May 2019.
- [Reducing page structures for huge pages](#), Jonathan Corbet, December 2020.
- [Sidestepping kernel memory management with DMEMFS](#), Jonathan Corbet, December 2020.



# Folios

- **Compound page** is a group of pages, represented by a **head page**. Other pages are called **tail pages**.
- A **folio** is a way of representing a set of **physically contiguous base pages**.
- It is a container for a **struct page** that is guaranteed not to be a tail page.
- Any function accepting a folio will operate on the full compound page (if, indeed, it is a compound page) with no ambiguity.
- The result is greater clarity in the kernel's memory-management subsystem; as functions are converted to take folios as arguments, it will become clear that they are not meant to operate on tail pages.
- The first set of folio patches was merged for the **5.16 kernel**.
- [Clarifying memory management with page folios](#), Jonathan Corbet, March 2021
- [A folio update](#), Jonathan Corbet, January 2022
- [A memory-folio update](#), Jonathan Corbet, May2022
- [Memory folios](#), Matthew Wilcox, Open Source Summit, 2022.



Matthew Wilcox



# Zones

The physical memory of RAM is divided into **zones**. This division results from the **hardware memory constraints**. The zones do not have any physical relevance but are simply **logical groupings** used by the kernel to keep track of pages.

Zone types (include/linux/mmzone.h)

- **ZONE\_DMA** – contains pages that can undergo DMA.
- **ZONE\_DMA32** – like ZONE\_DMA, contains pages that can undergo DMA. Unlike ZONE\_DMA, these pages are accessible only by 32-bit devices.
- **ZONE\_NORMAL** – contains normal, regularly mapped, pages.
- **ZONE\_HIGHMEM** – contains **high memory**, which are pages not permanently mapped into the kernel's address space.
- **ZONE\_MOVABLE** – similar to ZONE\_NORMAL, except that it contains movable pages with few exceptional cases.
- **ZONE\_DEVICE** – created to satisfy the need to perform DMA operations on persistent memory.



# Zones

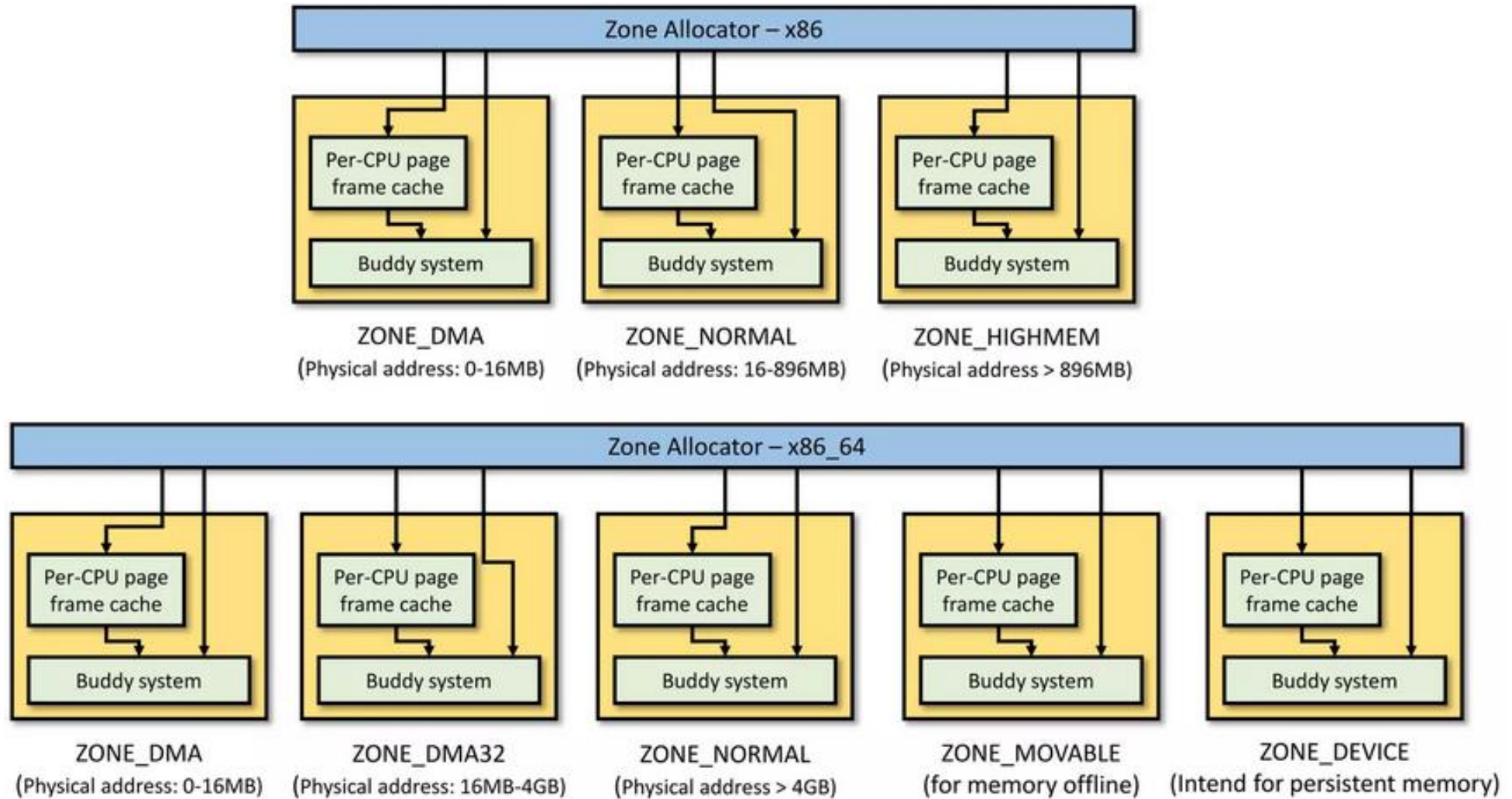
The layout of the memory zones is **architecture-dependent**.

- Some architectures can perform DMA into any memory address. In these architectures, **ZONE\_DMA** is empty and **ZONE\_NORMAL** is used for allocations regardless of their use.
- On the x86 architecture, ISA devices cannot perform DMA into the full 32-bit address space, because **ISA devices can access only the first 16 MB** of physical memory. Consequently, **ZONE\_DMA** on x86 consists of all memory in the range 0MB-16MB.
- On 32-bit x86 systems **ZONE\_HIGHMEM** is all memory above the physical ~900MB mark. A **64-bit architecture** such as **Intel's x86-64 can fully map and handle 64-bits of memory** — has no **ZONE\_HIGHMEM**, because **all memory is directly mapped**.
- On x86 **ZONE\_NORMAL** is all physical memory from 16 MB to ~900 MB. On other architectures, **ZONE\_NORMAL** is all available memory.





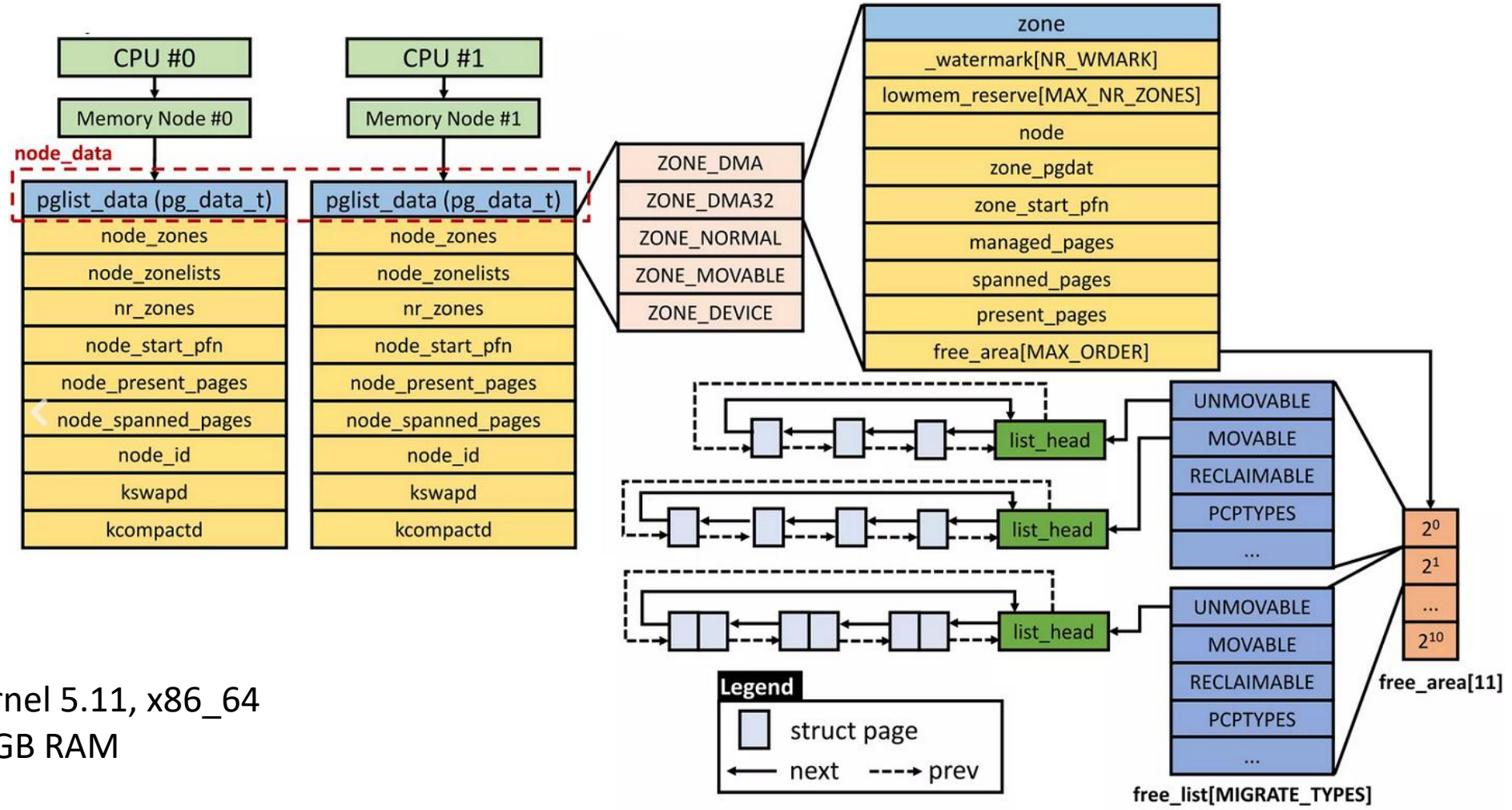
# Zones



(source: Adrian Huang, [Physical Memory Management](#), 2022)



# Zones

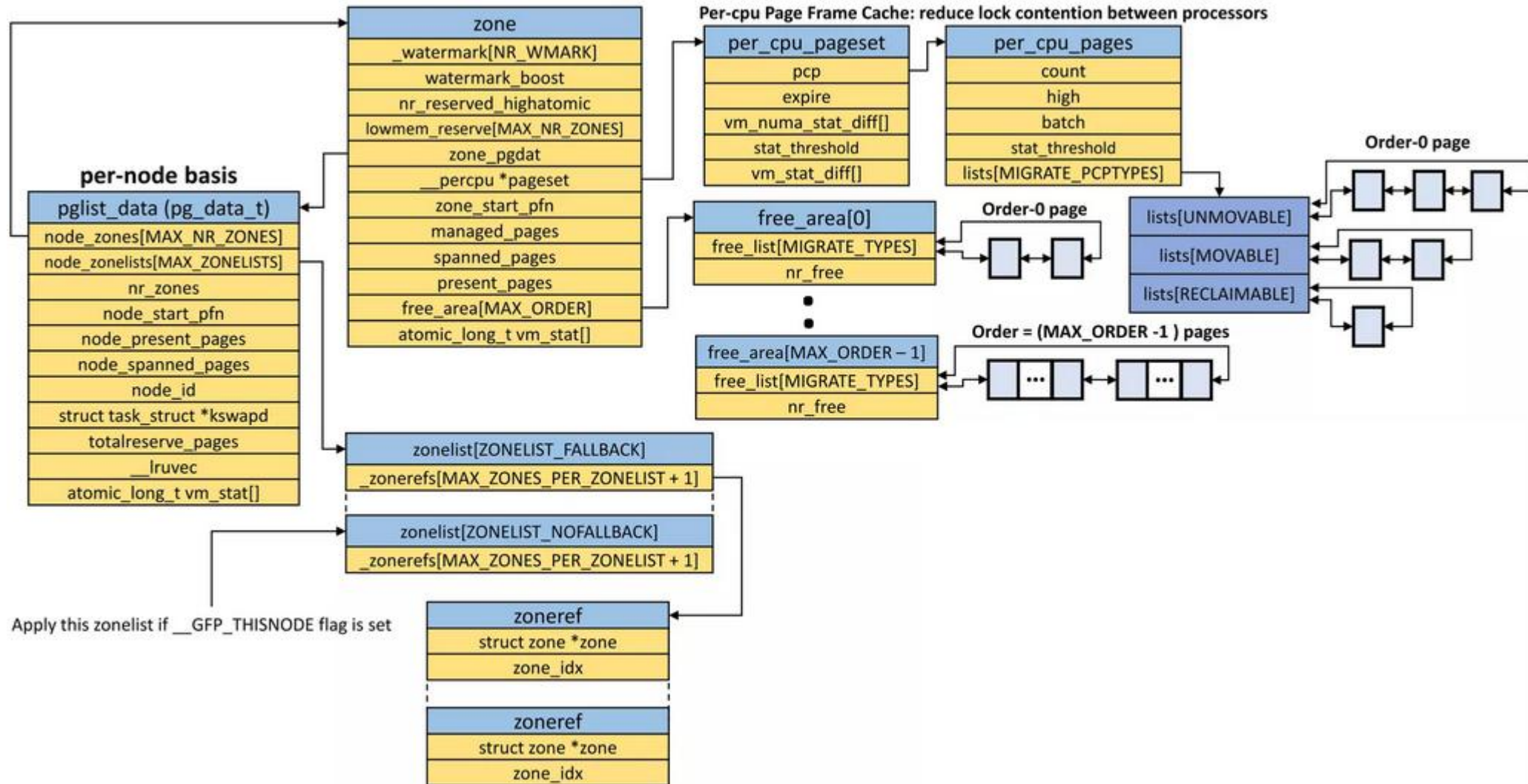


Kernel 5.11, x86\_64  
16GB RAM

(source: Adrian Huang, [Physical Memory Management](#), 2022)



# Zones





# Zones

Some fields of the zone descriptor:

- **pglist\_data** – describes the whole memory on a per-zone basis, in particular LRU lists of pages accessed by the page reclaim scanner (struct list\_head lists[NR\_LRU\_LISTS]),
- **pageset** – data structure implementing special caches for single frames,
- **zone\_start\_pfn** – page frame number of the first page frame in the zone,
- **free\_area** – identifies the blocks of free page frames in the zone (handled by the **buddy allocator**),
- **present\_pages** – the total size of the zone in the number of pages, excluding holes.

The **watermarks** are per-zone fields, used to determine when a zone needs to be **balanced**.

- **watermark[WMARK\_MIN]** – the number of reserved zone frames,
- **watermark[WMARK\_LOW]** – the lower limit of the number of frames for the page reclaiming mechanism,
- **watermark[WMARK\_HIGH]** – the upper limit of the number of frames for the page reclaiming mechanism,





# Zone watermarks

Some kernel control paths **cannot be blocked** while requesting memory, e.g.

- when handling an interrupt,
- when executing code inside a critical region.

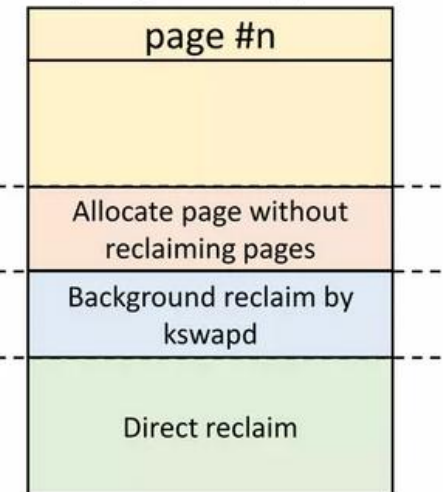
In these cases, a kernel control path should issue **atomic memory allocation requests** (using the **GFP\_ATOMIC** flag).  
An **atomic request never blocks**: if there are not enough free pages, the allocation fails.

The kernel reserves a **pool of page frames for atomic memory allocation requests** to be used only on low-on-memory conditions.

Note 1: This is the old formula. The new formula includes kswapd watermarks distance according to the scale factor.

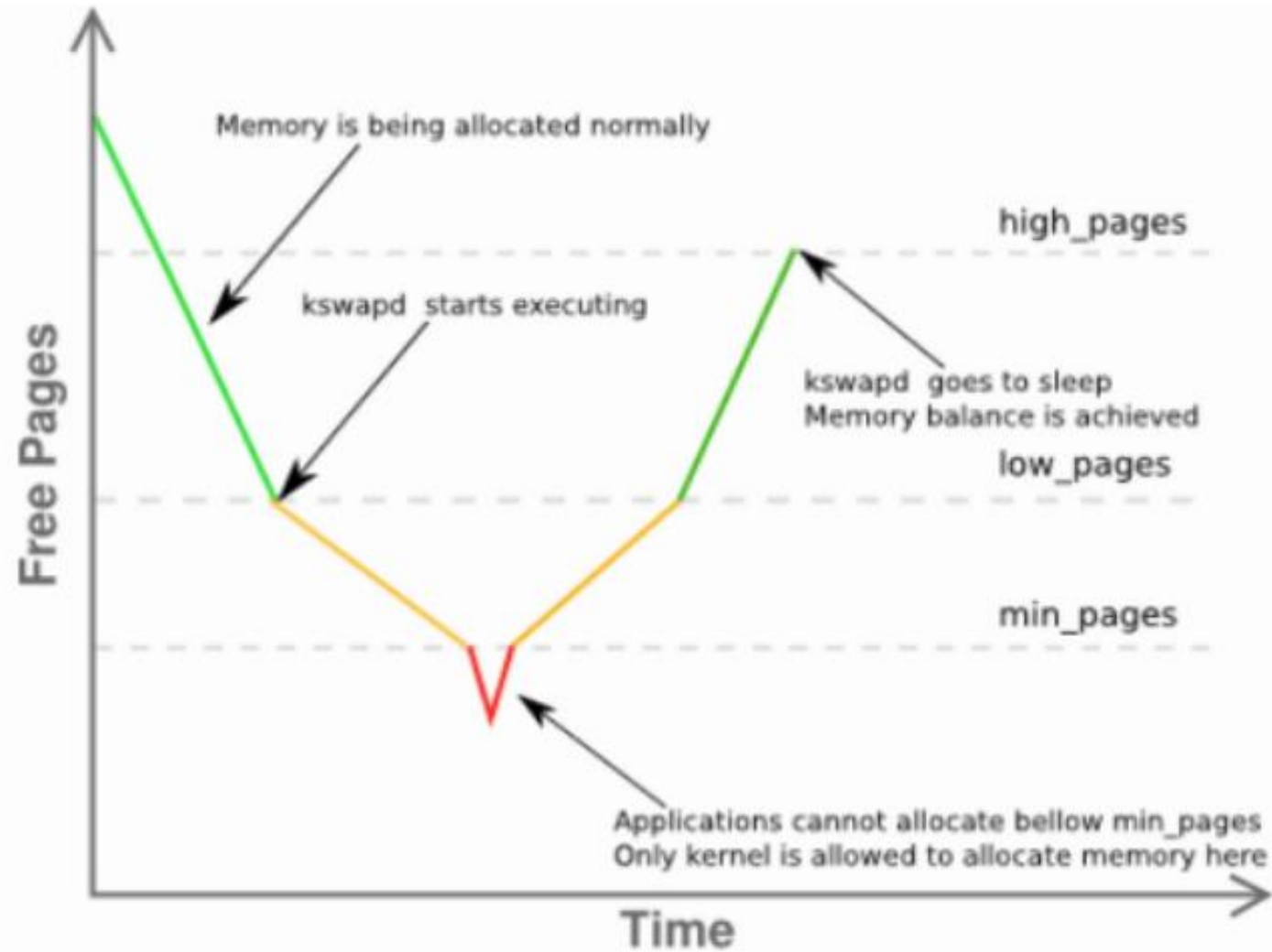
$$\begin{aligned} \_watermark[WMARK\_HIGH] &= \_watermark[WMARK\_MIN] * (3/2) \text{ Note 1} \\ \_watermark[WMARK\_LOW] &= \_watermark[WMARK\_MIN] * (5/4) \text{ Note 1} \\ \_watermark[WMARK\_MIN] &= \text{min\_free\_pages} * \frac{\text{Zone's managed pages}}{\text{All Zones' managed pages}} \end{aligned}$$

Zone (!highmem\_zone)





# Zone watermarks



(source: Edward Liu, [Out of memory events and decoding their logging](#))



# Zones

View zone statistics by executing:

**cat /proc/zoneinfo**

Servers on **students** and on **duch** have zones:  
DMA, DMA32, NORMAL, MOVABLE, DEVICE

Statistics are from **students**, cut off parts for  
other zones and cpus 1 .. 63 and some from the  
middle

Node 0, zone **Normal**

pages

free 16770626

min 18955

low 104499

high 190043

spanned 86975744

present 86975744

managed 85546808

protection: (0, 0, 0, 0, 0)

nr\_free\_pages 16770626    nr\_zone\_inactive\_anon 5559787

nr\_zone\_active\_anon 237185    nr\_zone\_inactive\_file 29311236

nr\_zone\_active\_file 4831654    nr\_zone\_unevictable 22199

nr\_zone\_write\_pending 599

nr\_mlock 22199

nr\_page\_table\_pages 92247

numa\_hit 39848507292

numa\_miss 0

numa\_foreign 0

numa\_interleave 41435

numa\_local 39848507292

numa\_other 0

pagesets

cpu: 0

count: 364

high: 378

batch: 63

vm stats threshold: 125

vm stats threshold: 125

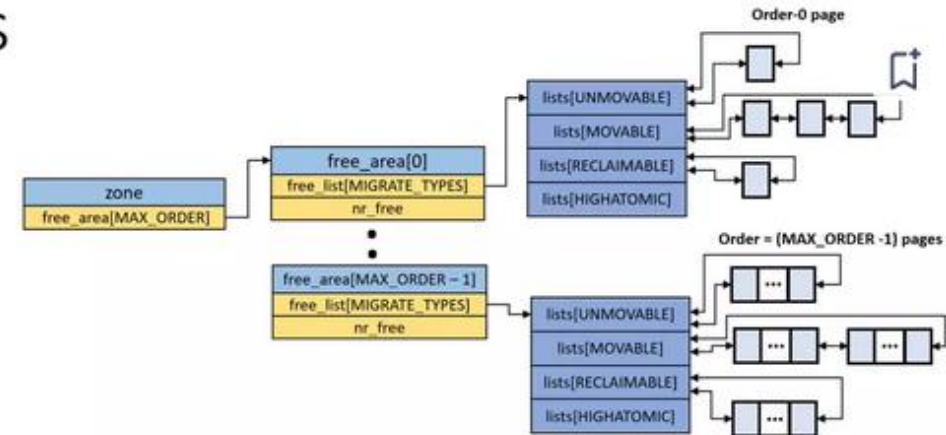
node\_unreclaimable: 0

start\_pfn: 1048576



# Zones

Placement of physical page frames  
right after system initialization



```
/ # cat /proc/pagetypeinfo
Page block order: 9
Pages per block: 512
```

Free pages count per migrate type at order				0	1	2	3	4	5	6	7	8	9	10
Node 0, zone DMA32, type Unmovable				0	1	0	1	0	0	1	0	1	0	0
Node 0, zone DMA32, type Movable				2	3	7	6	4	4	4	4	3	3	745
Node 0, zone DMA32, type Reclaimable				0	0	0	0	0	0	0	0	0	0	0
Node 0, zone DMA32, type HighAtomic				0	0	0	0	0	0	0	0	0	0	0
Node 0, zone Normal, type Unmovable				1	2	1	0	0	1	0	0	0	1	0
Node 0, zone Normal, type Movable				1	1	0	1	1	0	3	0	1	1	1235
Node 0, zone Normal, type Reclaimable				1	1	0	1	1	0	1	1	1	0	0
Node 0, zone Normal, type HighAtomic				0	0	0	0	0	0	0	0	0	0	0

Number of blocks type		Unmovable	Movable	Reclaimable	HighAtomic
Node 0, zone DMA32		1	1535	0	0
Node 0, zone Normal		16	2544	2	0

1 Most physical page frames are stored in MIGRATE\_MOVABLE

2 [Right after system boots] Most physical page frames are stored in MAX\_ORDER (order=10)





# Layout of the kernel address space

(source: W. Maurer, Professional Linux Kernel Architecture)  
(\_PAGE\_OFFSET constant – one zero is missing)

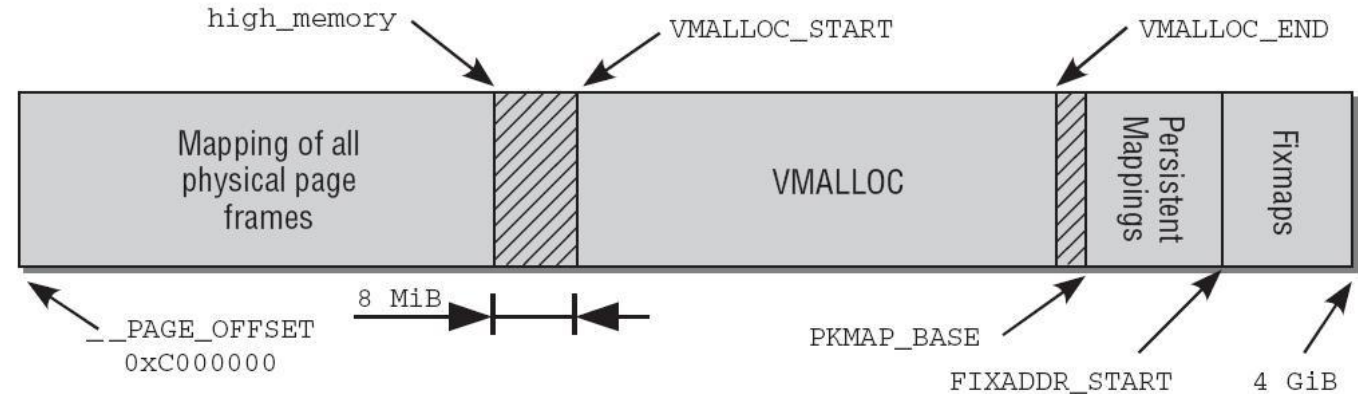


Figure 3-15: Division of the kernel address space on IA-32 systems.

IA-32 maps the page frames into the virtual address space starting from **PAGE\_OFFSET**.

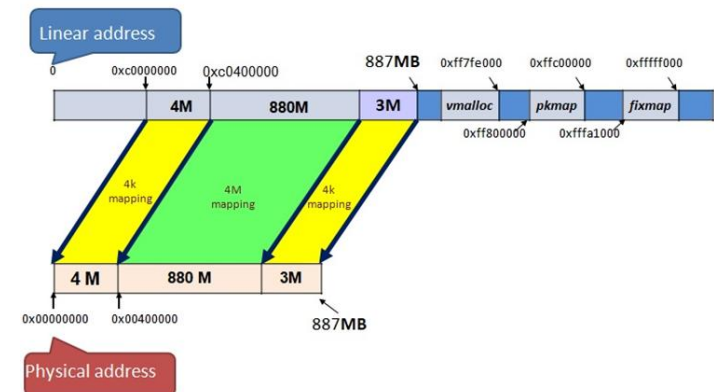
Kernel runs in a **virtual address space**, but uses **physical addresses** to do a lot of things (e.g. **preparing the page table entry, get dma address for device**).

So kernel needs functions like **\_\_va()** and **\_\_pa()**:

```
#define __va(x) ((void *)((unsigned long)(x) + PAGE_OFFSET))
#define __pa(x) ((unsigned long)(x) - PAGE_OFFSET)
```

This translation takes place at the **compilation time**. It does not use page tables.

This is called **directly-mapped** memory.





# Mapping frames from highmem

**High memory (highmem)** is used when the size of physical memory approaches or exceeds the maximum size of **virtual memory**. It becomes impossible for the kernel to keep all of the available physical memory mapped at all times. The kernel needs to start using **temporary mappings** of the pieces of physical memory that it wants to access.

Frames from **highmem** do not have **fixed linear addresses**. The last addresses of the kernel's linear space are **dedicated to mapping these frames**. Mapping is **temporary**, because otherwise access would be only to a part of the highmem.

Allocating frames from the **highmem**, the kernel uses **alloc\_pages()** and **alloc\_page()**\*

They do not pass the **linear address of the first allocated frame**, but the **linear address of the frame descriptor** (this address always exists because the **descriptors** are allocated in **directly addressed memory** – this happens during the initialization of the kernel data structures).

- **struct page \*alloc\_pages**(gfp\_mask, order) – allocates  $2^{\text{order}}$  of **physically contiguous frames** and forwards the link to the descriptor of the first of them,
- **struct page \*alloc\_page**(gfp\_mask) – allocates a single frame.

The gfp\_mask tells the page allocator which pages can be allocated, whether the allocator can wait for more memory to be freed, etc.

The kernel uses **three mechanisms** to map frames from **highmem**.



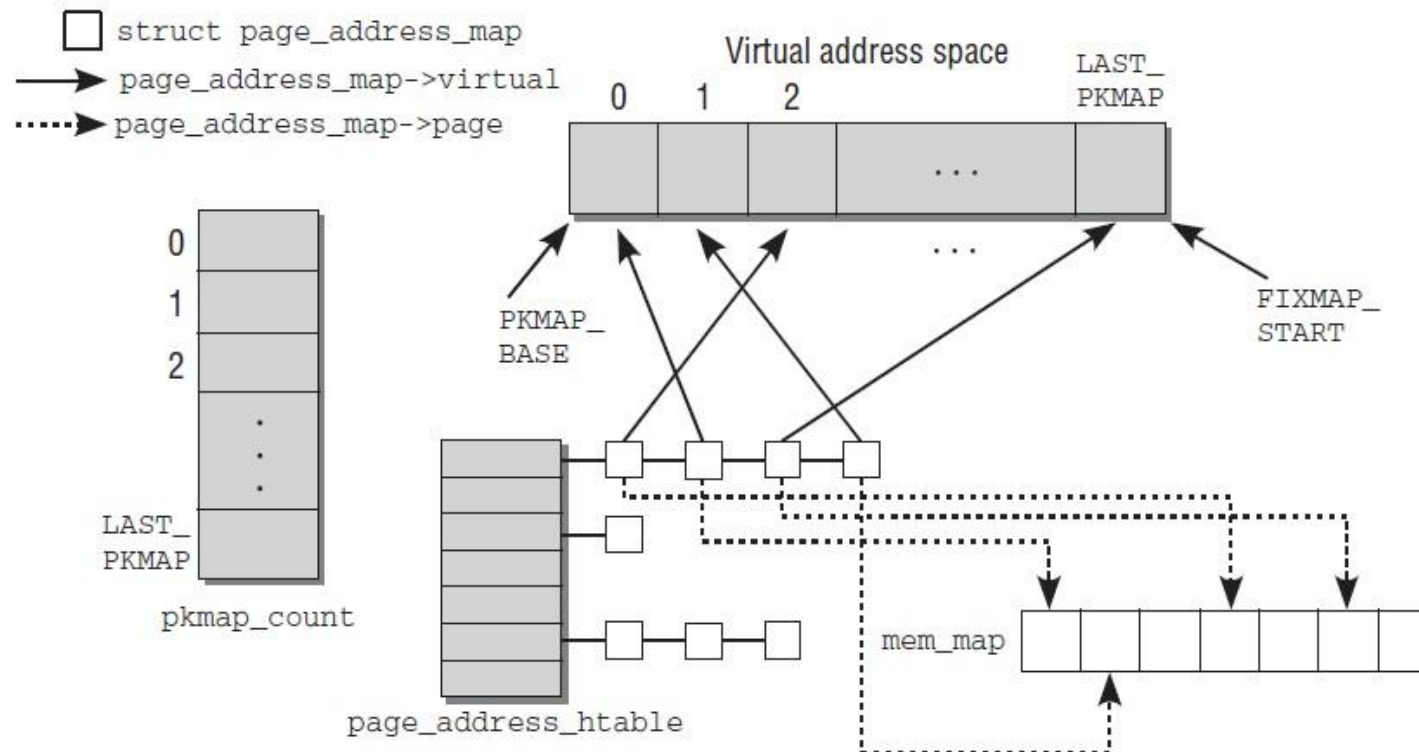


# Persistent Kernel Mapping (PKMAP)

**Permanent kernel mappings** allow the kernel to establish **long-lasting mappings** of high-memory page frames into the kernel address space.

They use a **dedicated Page Table in the Master Kernel Page Tables**. The **pkmap\_page\_table** variable stores the address of this page table. It maps linear addresses starting from **PKMAP\_BASE**, the **LAST\_PKMAP** macro yields the number of entries.

See [linux/latest/source/mm/highmem.c](#)



```
static int pkmap_count[LAST_PKMAP];  
  
pte_t *pkmap_page_table;  
  
struct page_address_map {  
    struct page *page;  
    void *virtual;  
    struct list_head list;  
};
```

Figure 3-41: Data structures for managing persistent mappings.

(source: W. Mauerer, Professional Linux Kernel Architecture)



# Persistent Kernel Mapping (**PKMAP**)

The array of counters **pkmap\_count** describes the entries in the dedicated page table.

**Descriptors** of mapped frames from **highmem** are available through a **hash table** (you can not use a linear address as an **index in a frame table**, because there are more frames than linear addresses). Each element stored in this table contains a **frame descriptor** and its assigned **linear address**.

The function **map\_new\_virtual()**:

- browses the **pkmap\_count** array,
- finds an unused entry,
- sets the corresponding linear address,
- fills the entry in the **pkmap\_count** array and entry in the **pkmap\_page\_table**,
- inserts a new element into the hash table **page\_address\_htable**,
- passes the linear address.

If there is no free entry, the function **blocks the current process** until another process does not free the entry – such mapping can not be used in the context of an interrupt.



# Persistent Kernel Mapping (**PKMAP**)

The use of **high memory** is a problem **only for the kernel**. The kernel must first use the **kmap()** and **kunmap()** functions to map frames from high memory to its virtual address space. It does not have to do this for frames from other zones.

For the **user process**, there is **no difference between frames from high memory and others**, because access to them always takes place through **page tables**, never **directly**.

The **kmap()** function must be used if highmem pages are to be mapped into kernel address space for a longer period (as a **persistent mapping**).

The page to be mapped is specified by means of a **pointer to page** as the function parameter.

The function creates a **mapping** when this is necessary (i.e. if the page really is a **highmem** page) and returns the **address of the data**.

```
void * kmap (struct page * page)
{
    if (!PageHighMem(page))
        return page_address(page);
    return kmap_high(page);
}

void * kmap_high (struct page * page)
{
    unsigned long vaddr;
    lock_kmap();
    vaddr = (unsigned long) page_address(page);
    if (!vaddr)
        vaddr = map_new_virtual(page);
    pkmap_count[PKMAP_NR(vaddr)]++;
    unlock_kmap();
    return (void *) vaddr;
}
```



# Temporary Kernel Mapping (**FIXMAP**)

Mappings done by **kmap()** are **costly**. To improve performance, the kernel developers introduced a special version:

```
/*
 * kmap_atomic/kunmap_atomic is significantly faster than kmap/kunmap because
 * no global lock is needed and because the kmap code must perform a global TLB
 * invalidation when the kmap pool wraps.
 *
 * However when holding an atomic kmap it is not legal to sleep, so
 * atomic kmaps are appropriate for short, tight code paths only.
 */

/* Find the page of interest. */
struct page *page = find_get_page(mapping, offset);

/* Gain access to the contents of that page. */
void *vaddr = kmap_atomic(page);

/* Do something to the contents of that page. */
memset(vaddr, 0, PAGE_SIZE);

/* Unmap that page. */
kunmap_atomic(vaddr);
```

**kmap\_atomic()** uses one of a small set of address slots for the mapping, and that mapping is only valid on the CPU where it is created.

Whenever code running in kernel space creates an atomic mapping, it can no longer be preempted or migrated, and it is not allowed to sleep, until all atomic mappings have been released.



# Temporary Kernel Mapping (**FIXMAP**)

The function **kmap\_atomic()** differs from **kmap()**. It creates a mapping on the **current CPU**, so there is no need to bother other processors with it.

It creates the mapping using one of a very **small set of kernel-space addresses**. These addresses are specified by a set of **slot constants**. There are about **twenty** of these slots defined in current kernels.

The use of **fixed slots** requires that the code using these mappings be **atomic**.

The per-CPU nature of atomic mappings means that any **cross-CPU migration would be disastrous**.

The **kmap\_atomic()** function **does not block the process**. It is ideal in short pieces of code that need a **temporary frame** quickly.

A **fixmap linear address** is a constant that corresponds to a **physical address**. Each such address maps one physical memory frame. Any physical address can be mapped in this way, without maintaining the linear order of addresses.

Such addressing is very **effective**.

See also: [Atomic kmaps become local](#), Jonathan Corbet. November 2020





# Managing noncontiguous memory areas (**VMALLOC**)

If references to memory areas are **not frequent**, instead of allocating areas of adjacent frames, the kernel can assigned frames that are not adjacent to each other (**noncontiguous memory area**), but are in the area of **contiguous linear addresses**.

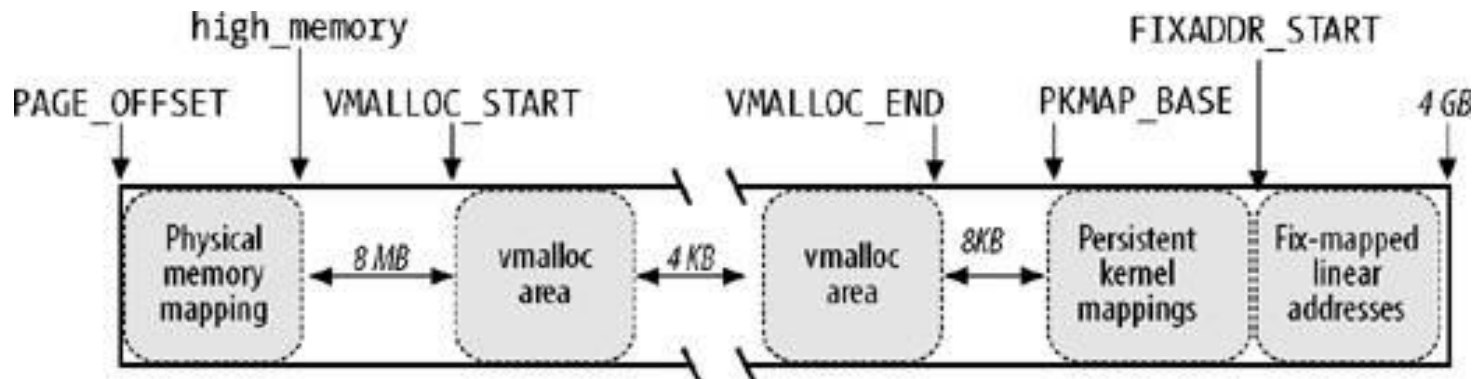
Thanks to this, you can **avoid external fragmentation**, but you must reach for the **kernel page tables**.

The size of the noncontiguous memory area must be a **multiple of the page size**.

Linux uses such **noncontiguous memory areas** by allocating there e.g.

- data structures for active swap areas,
- memory for modules,
- buffers for some input-output drivers.

Free ranges of linear addresses are available starting from **PAGE\_OFFSET**.



(source: Bovet, Cesati,  
Understanding the Linux Kernel)





# Managing noncontiguous memory areas (**VMALLOC**)

Linear addresses in the range from **VMALLOC\_START** to **VMALLOC\_END** are reserved for **noncontiguous memory areas**.

Each such area is described as **vm\_struct descriptor**.

**addr** – linear address of the first memory cell of the area,

**size** – size of the area increased by 4 KB (safety area),

**pages** – pointer to an array of **nr\_pages** pointers to page descriptors,

**nr\_pages** – number of entries in pages,

**next** – pointer to the next item in the list,

**flags** – flags specifying the area type

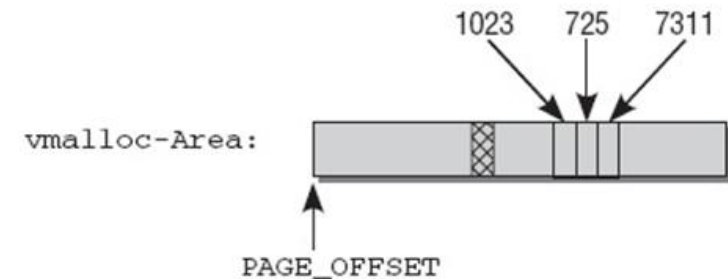
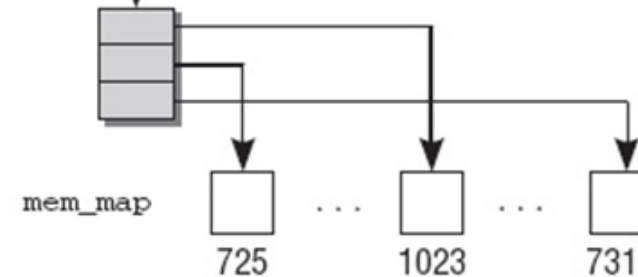
VM\_ALLOC – pages obtained from **vmalloc()**,

VM\_MAP – already assigned pages mapped with **vmap()**,

VM\_IOREMAP – memory on the hardware device board mapped using **ioremap()**.

```
struct vm_struct {  
    struct vm_struct  *next;  
    void              *addr;  
    unsigned long     size;  
    unsigned long     flags;  
    struct page       **pages;  
    unsigned int       nr_pages;  
    unsigned long     phys_addr;  
    void              *caller;  
};
```

```
addr=VMALLOC_START+100  
size=3*PAGE_SIZE  
nr_pages=3  
pages=
```



Mapping physical pages  
into the vmalloc area  
(source: Mauerer,  
Professional Linux  
Kernel Architecture)



# Managing noncontiguous memory areas (**VMALLOC**)

The global variable **vmist** points to the beginning of the list of areas **vm\_struct**.

The function **get\_vm\_area()** looks for a free range of linear addresses between VMALLOC\_START and VMALLOC\_END (by browsing the **vmist** list).

If the range of the right size can be found, it passes the completed descriptor.

The **vmalloc()** function allocates a **noncontiguous memory area** of a **fixed size**:

- rounds up this size to a **multiple of the page frame size** (4096 bytes).
- invokes **get\_vm\_area()** to get a correctly filled descriptor.
- invokes **kmalloc()** to get a group of contiguous page frames that will store the array with pointers to the page descriptors.
- the **memset()** function initializes all of them to 0.
- the function **alloc\_page()** is called in the loop, to allocate a page frame and store the address of the corresponding page descriptor in the pages array. Frames can come from highmem, so they may not be mapped to linear addresses yet.

The functions **vfree()** i **vunmap()** are used to release noncontiguous memory areas.



# Managing noncontiguous memory areas (**VMALLOC**)

The **mapping and unmapping of the physical memory in the virtual address space in the kernel** comes at price as you now need to **consult and maintain the page tables** and that incurs a performance hit. But:

1. **TLBs** alleviate this problem to a certain degree. They cache translations.
2. **Global pages.**

When you switch between applications and need to flush the current TLB, you can avoid invalidating global pages from the TLB by performing **Invalidate TLB entries by ASID** (*Address Space Identifier*) match with the application's ASID.

If you mark the **kernel's pages as global**, you don't invalidate their translations and the kernel itself doesn't suffer from unnecessary TLB invalidations.

4. The portion of the kernel address space which is identity mapped to RAM (kernel logical addresses) are **mapped using big pages** when possible, which may allow the **page table to be smaller** but more importantly **reduces the number of TLB misses**.

Are **kernel virtual addresses really translated by the TLB/MMU?** **YES.**



# To **kmalloc()** or to **vmalloc()**?

The answer is [kvmalloc\(\)](#) (Jonathan Corbet, January 2017)

Regardless of the fact that physically contiguous memory is needed only in special cases, the kernel mostly uses **kmalloc()** and **not vmalloc()** to allocate memory. The reason is **efficiency**.

Allocations with **vmalloc()** do not need physically contiguous pages and are more likely to succeed when memory is tight. But excessive use of `vmalloc()` is discouraged due to the extra overhead involved.

**vmalloc()** can only allocate **entire pages**, so it is not suitable for **small requests**.

The **address range** available for **vmalloc()** allocations is also **limited** on **32-bit** systems; that limitation is not present on 64-bit systems.

There are a **number** of **places** in the kernel where a **large allocation** must be **physically contiguous**, but there are probably **even more** where that **doesn't matter**.

In the latter case, the code doesn't have a reason to care which allocation method was used to obtain its memory, as long as the memory is available. In such case it makes sense to try an allocation **first** with **kmalloc()**, then fall back to **vmalloc()** should that attempt fail.

The kernel is full of code fragments that do exactly that.



# To **kmalloc()** or to **vmalloc()**?

The answer is [kvmalloc\(\)](#) (Jonathan Corbet, January 2017)

**kvmalloc()** attempts to allocate size bytes from the **slab allocator**; trying to minimize the cost (and avoid out-of-memory killer invocations) when the memory is not immediately available.

If the attempt fails, `kvmalloc()` will fall back to **vmalloc()** to perform the allocation.

The **kvzalloc()** variant will zero the memory before returning it.

```
void *kvmalloc(size_t size, gfp_t flags);  
void *kvzalloc(size_t size, gfp_t flags);
```



## An end to high memory?

(Jonathan Corbet, February 2020)

**64-bit systems** do not have the **4GB virtual address space limitation**, so they have **never needed** the high-memory concept.

But **high memory remains for 32-bit systems**, and traces of it can be seen throughout the kernel.

Calls to **kmap()** and **kmap\_atomic()** do **nothing on 64-bit systems**, but are needed to access high memory on smaller systems.

According to Linus high memory should be now considered to be **deprecated**: "***In this day and age, there is no excuse for running a 32-bit kernel with lots of physical memory***".

**Removing high-memory** would simplify the memory-management code significantly with no negative effects on the **64-bit systems** that everyone is using now.

Except, of course, not every system has a **64-bit CPU** in it.

The area of biggest concern is the **ARM architecture**, where **32-bit CPUs** are still being built, sold, and deployed.

See also: [The future of 32-bit Linux](#), Arnd Bergmann, December 2020



# Additional reading

- [High Memory Management](#) , May 2016.
- [Fixing kmap\\_atomic\(\)](#), Jonathan Corbet, 2009.
- [High memory handling](#), Peter Zijlstra.
- [Stack overflow – Kernel memory \(virtual address entries\) in TLB.](#)
- [How exactly do kernel virtual addresses get translated to physical RAM?](#)
- [Making kernel objects movable: A history and the way forward](#), Christoph Lameter, 2017.
- Highmem in 64-bit architectures
  - [ARM64 Linux kernel virtual address space](#)
  - [Is highmem relevant in 64 bit Linux? If not, why?](#)
  - [Why does highmemory not exist for 64-bit CPU?](#)
  - [Why do 64 bit systems have only a 48 bit address space?](#)
- [Linux kernel – memory in 64-bit architectures \(Marcin Walas\)](#)