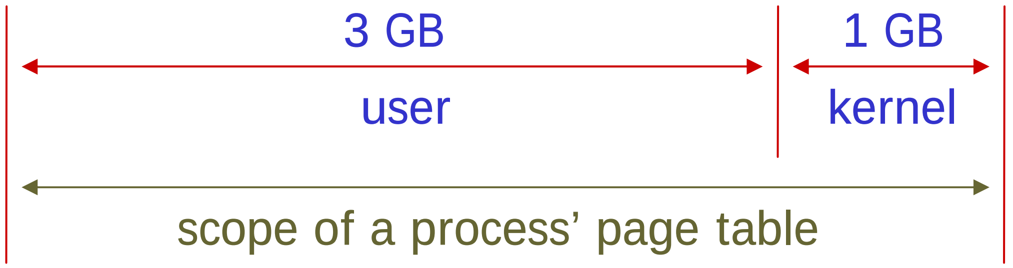
Physical Memory Management in Linux

Hao-Ran Liu

Table of Contents

Virtual Address Space and Memory Allocators in Linux

Linux Virtual Address Layout



3 GB

user

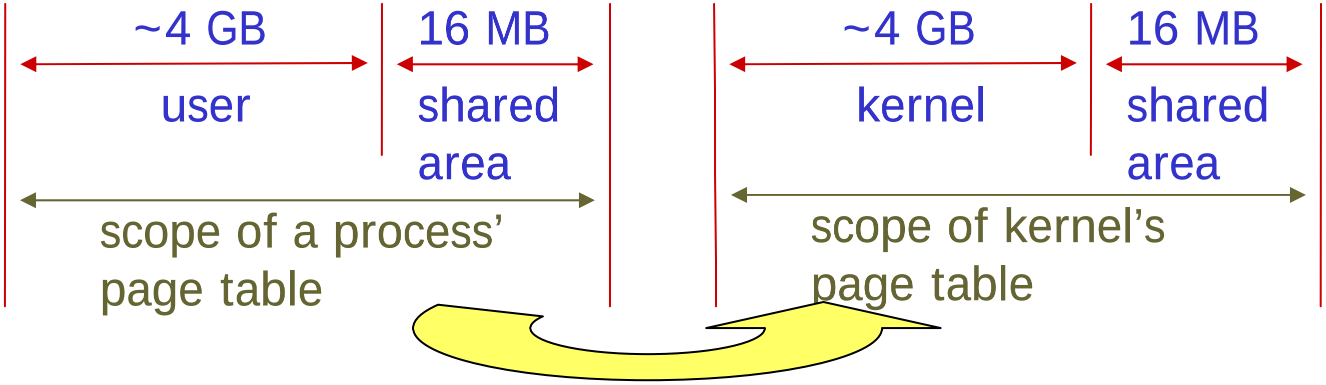
1 GB

kernel

scope of a process' page table

- 3G/1G partition
  - The way Linux partition a 32-bit address space
  - Cover user and kernel address space at the same time
  - Advantage
    - Incurs no extra overhead (no TLB flushing) for system calls
  - Disadvantage
    - With 64 GB RAM, `mem_map` alone takes up 512 MB memory from lowmem (ZONE_NORMAL).

Linux Virtual Address Layout



~4 GB

user

16 MB

shared area

scope of a process' page table

~4 GB

kernel

16 MB

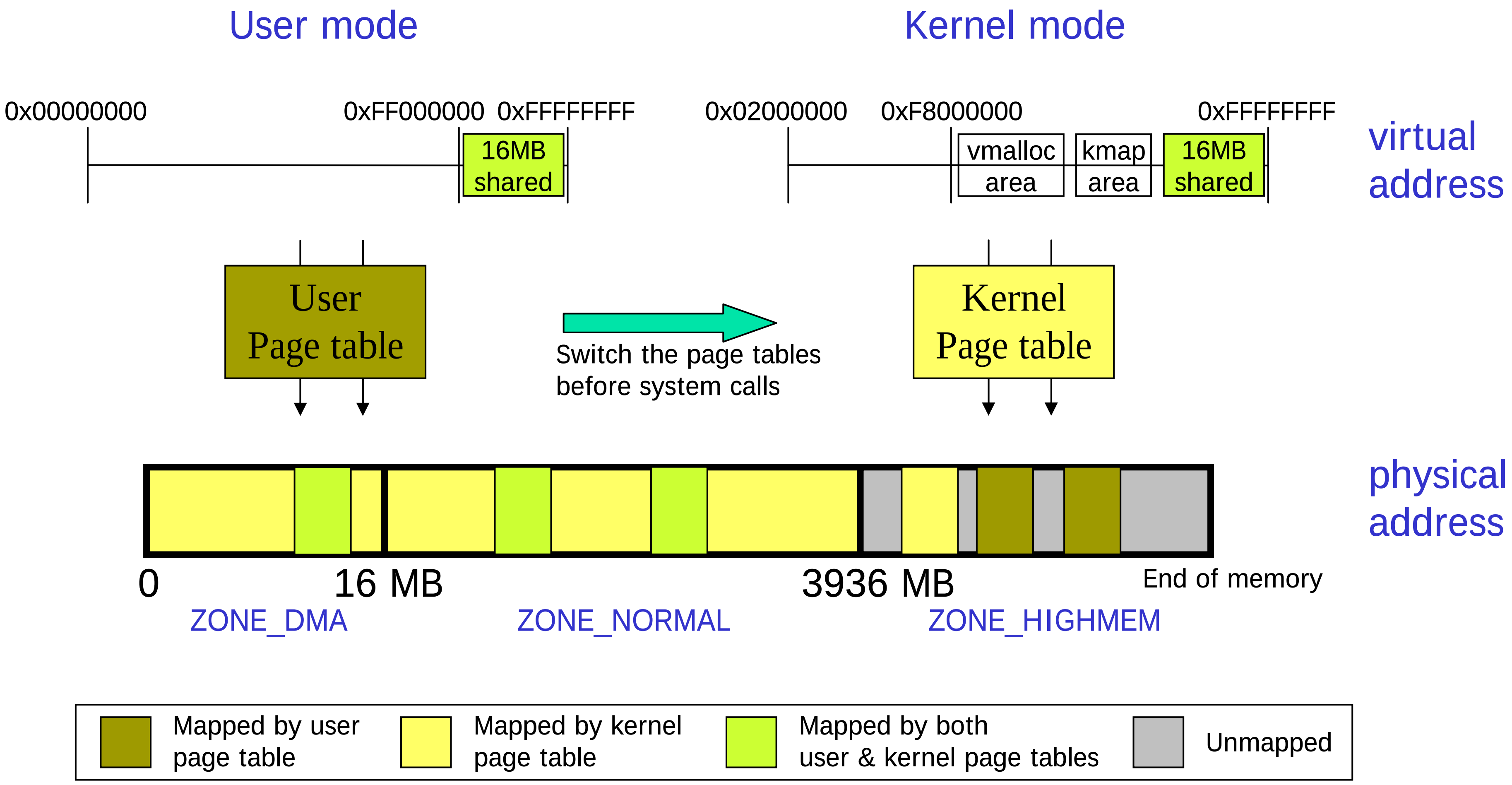
shared area

scope of kernel's page table

switch the page table before system calls

- 4G/4G partition
  - Proposed by Red Hat to solve `mem_map` problem
  - Disadvantage (Performance drop!)
    - Switch page table and flush TLB for every system call!
    - Data is copied "indirectly" (with the help of `kmap`) between user and kernel space
  - Advantage
    - Only on machine with large RAM

Page Table Switch in a 4G/4G Configuration



User mode

Kernel mode

0x00000000 0xFF000000 0xFFFFFFFF

16MB shared

0x02000000 0xF8000000 0xFFFFFFFF

vmalloc area

kmap area

16MB shared

virtual address

User Page table

Switch the page tables before system calls

Kernel Page table

physical address

0

16 MB

3936 MB

End of memory

ZONE_DMA

ZONE_NORMAL

ZONE_HIGHMEM

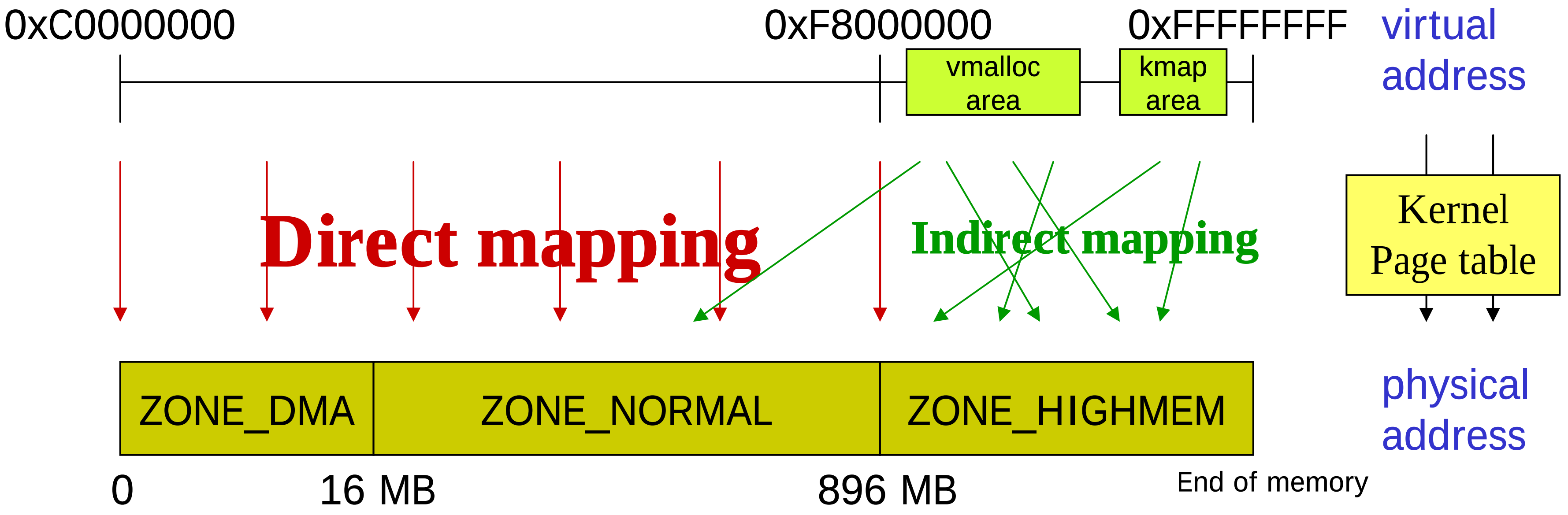
🟫 Mapped by user page table

🟨 Mapped by kernel page table

🟩 Mapped by both user & kernel page tables

⬜ Unmapped

Partition of Physical Memory (Zone)

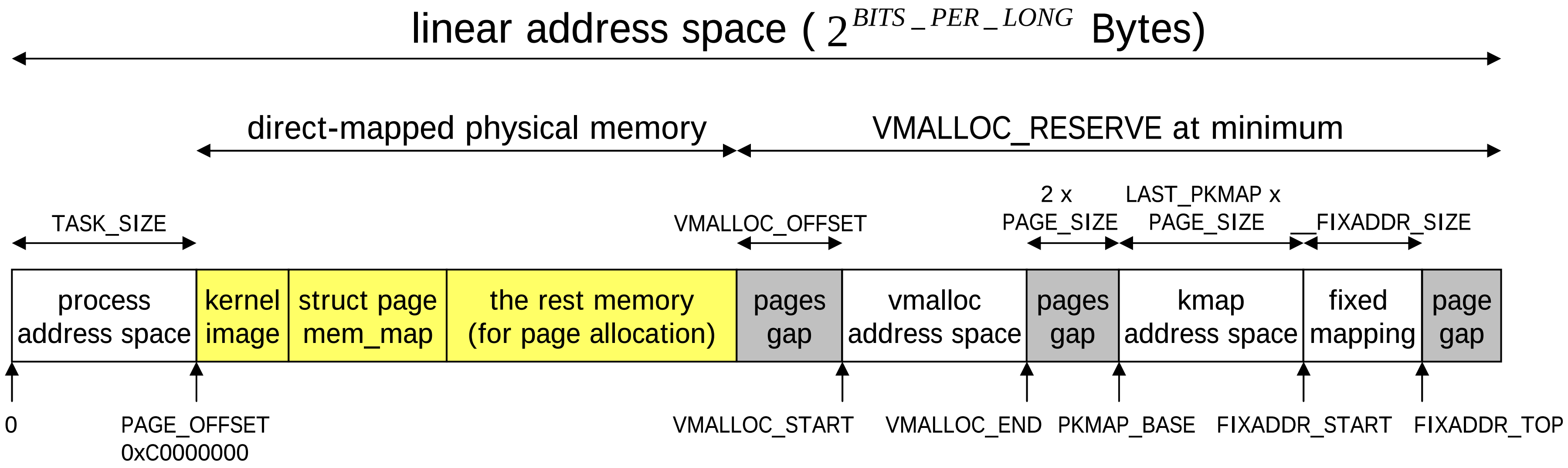


This figure shows the partition of physical memory its mapping to virtual address in 3G/1G layout

Why not map kernel memory indirectly?

- Reasons for direct mapping
  - No changes of kernel page table for contiguous allocation in physical memory
  - Faster translation between virtual and physical addresses
- Implications of direct mapping
  - kernel memory is not swappable

Kernel Virtual Address Space



linear address space ($2^{BITS\_PER\_LONG}$ Bytes)

direct-mapped physical memory

VMALLOC_RESERVE at minimum

TASK_SIZE

VMALLOC_OFFSET

2 x PAGE_SIZE

LAST_PKMAP x PAGE_SIZE

__FIXADDR_SIZE

process address space

kernel image

struct page mem_map

the rest memory (for page allocation)

pages gap

vmalloc address space

pages gap

kmap address space

fixed mapping

page gap

0

PAGE_OFFSET 0xC0000000

VMALLOC_START

VMALLOC_END

PKMAP_BASE

FIXADDR_START

FIXADDR_TOP

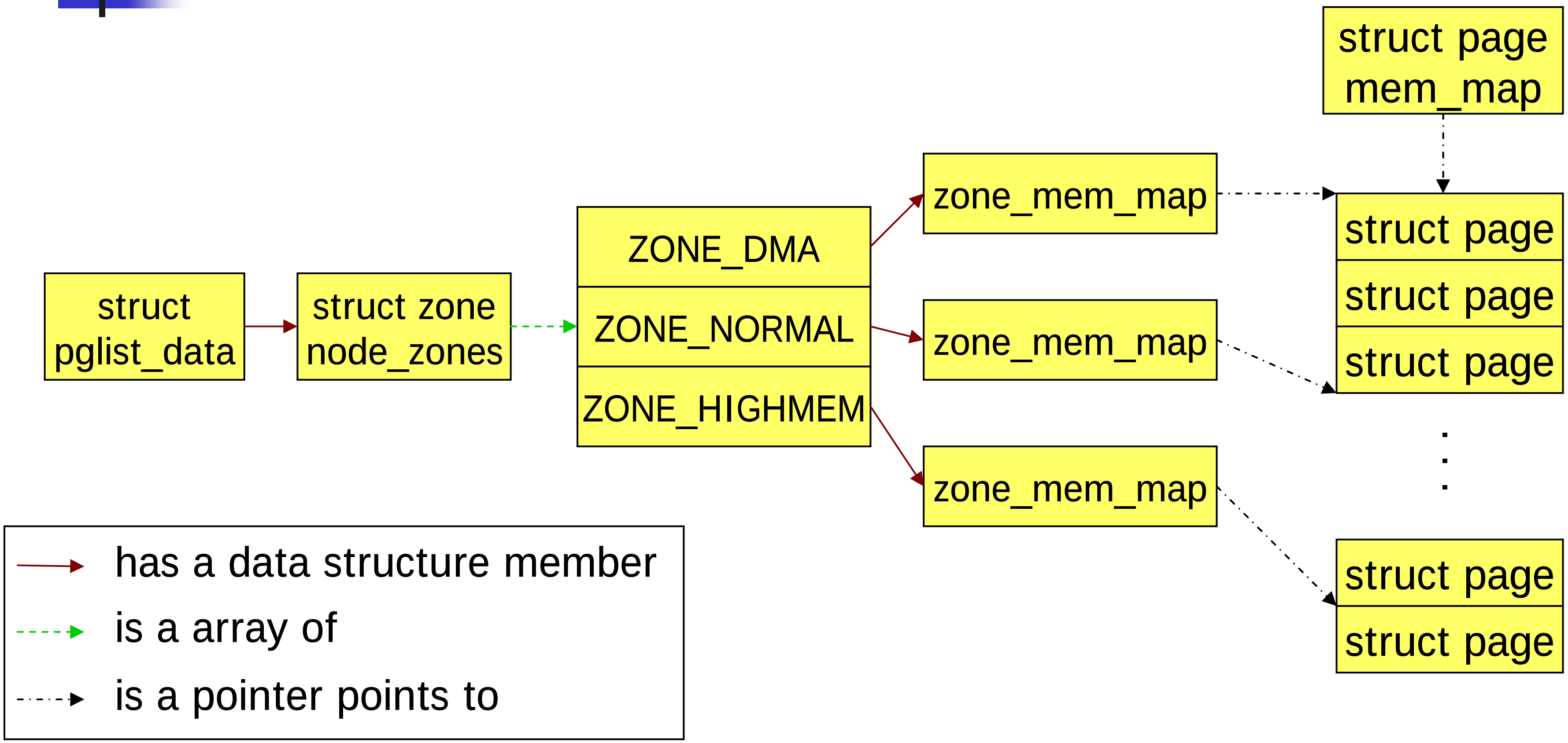
- vmalloc address space
  - Noncontiguous physical memory allocation
- kmap address space
  - Allocation of memory from ZONE_HIGHMEM
- Fixed mapping
  - Compile-time virtual memory allocation

Memory Allocators in Linux

	Description	Used at	functions
Boot Memory Allocator	1. A first-fit allocator, to allocate and **free** memory during kernel boots 2. Can handle allocations of sizes smaller than a page	System boot time	`alloc_bootmem()` `free_bootmem()`
Physical Page Allocator (buddy system)	1. **Page-size** physical frame management 2. Good at dealing with **external fragmentation**	After `mem_init()`, at which boot memory allocator retires	`alloc_pages()` `__get_free_pages()`
Slab Allocator	1. Deal with **Internal fragmentation** (**for allocations < page-size**) 2. Caching of commonly used objects 3. Better use of the hardware cache	After `mem_init()`, at which boot memory allocator retires	`kmalloc()` `kfree()`
Virtual Memory Allocator	1. Built on top of page allocator and **map noncontiguous physical pages to logically contiguous vmalloc space** 2. Required altering the kernel page table 3. Size of all allocations <= vmalloc address space	1. Large allocation size 2. contiguous physical memory is not available	`vmalloc()` `vfree()`

Describing Physical Memory

Data Structures to Describe Physical Memory



struct pglist_data

struct zone node_zones

ZONE_DMA

ZONE_NORMAL

ZONE_HIGHMEM

zone_mem_map

zone_mem_map

zone_mem_map

struct page mem_map

struct page

struct page

struct page

.
.
.

struct page

struct page

→ has a data structure member

⇢ is a array of

⇢ is a pointer points to

All these data structures are initialized by `free_area_init()` at `start_kernel()`

Page Tables vs. `struct page`s

- Page tables
  - Used by CPU memory management unit to map virtual address to physical address
- `struct page`s
  - Used by Linux to keep track of the status of all physical pages
  - Some status (eg. dirty, accessed) is read from the page tables.

Nodes

- Designed for NUMA (Non-Uniform Memory Access) machine
- Each bank (The memory assigned to a CPU) is called a node and is represented by `struct pglist_data`
- On Normal x86 PCs (which use UMA model), Linux uses a single node (`contig_page_data`) to represent all physical memory.

struct pglist_data

Type	Name	Description
`struct zone []`	`node_zones`	Array of zone descriptors of the node
`struct zonelist []`	`node_zonelists`	The order of zones that allocations are preferred from
`int`	`nr_zones`	Number of zones in the node
`struct page *`	`node_mem_map`	This is the first page of the `struct page` array that represents each physical frame in the node
`struct bootmem_data *`	`bdata`	Used by boot memory allocator during kernel initialization
`unsigned long`	`node_start_pfn`	The starting physical page frame number of the node
`unsigned long`	`node_present_pages`	Total number of physical pages in the node
`unsigned long`	`node_spanned_pages`	Total size of physical page range, including holes
`int`	`node_id`	Node ID (NID) of the node
`struct pglist_data *`	`pgdat_next`	Pointer to next node in a NULL terminated list

Zones

- Because of hardware limitations, the kernel cannot treat all pages as identical
  - Some hardware devices can perform DMA only to certain memory address
  - Some architectures cannot map all physical memory into the kernel address space.
- Three zones in Linux, described by `struct zone`
  - ZONE_DMA
    - Contains pages capable of undergoing DMA
  - ZONE_NORMAL
    - Contains regularly mapped pages
  - ZONE_HIGHMEM
    - Contains pages not permanently mapped into the kernel address space

struct zone (1)

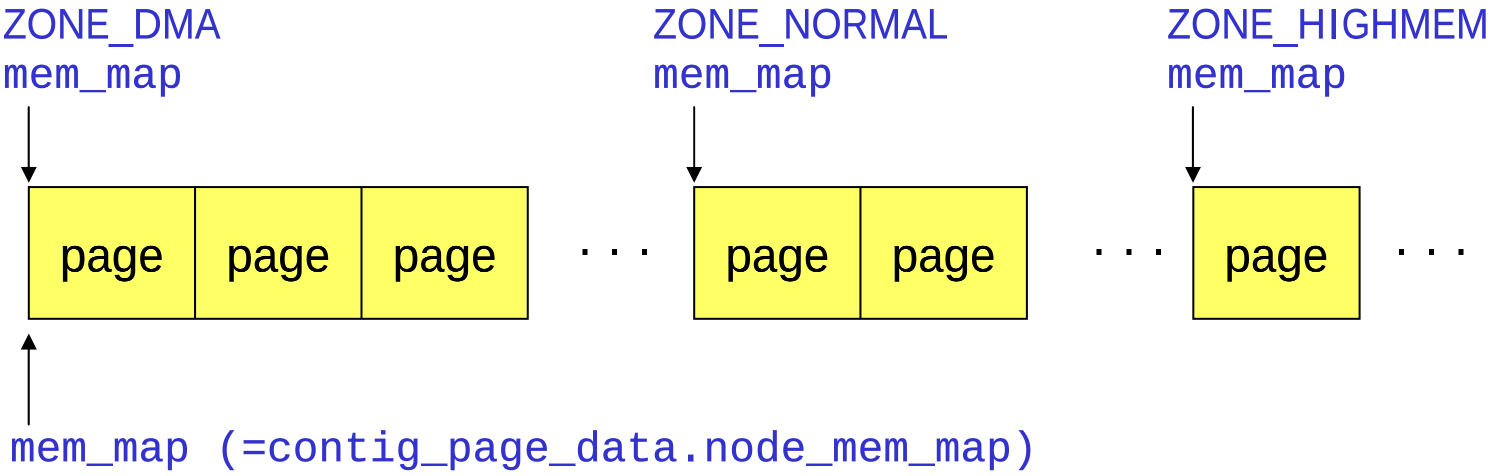
Type	Name	Description	Notes
`spinlock_t`	`lock`	Spin lock protecting the descriptor	
`unsigned long`	`free_pages`	Number of free pages in the zone	
`unsigned long`	`pages_min`	Minimum number of pages of the zone that should remain free	Kswapd
`unsigned long`	`pages_low,` `pages_high`	Lower and upper threshold value for the zone's page balancing algorithm	Kswapd
`spinlock_t`	`lru_lock`	Spin lock protecting the following two linked lists	Page cache
`struct list_head`	`active_list,` `inactive_list`	Active and inactive lists (LRU lists) of pages in the zone	Page cache
`unsigned long`	`nr_active,` `nr_inactive`	The number of pages on the `active_list` and `inactive_list`	Page cache

struct zone (2)

Type	Name	Description
`struct free_area []`	`free_area`	Free area bitmaps used by the buddy allocator
`wait_queue_head_t *`	`wait_table`	A hash table of wait queues of processes waiting on a page to be freed
`unsigned long`	`wait_table_size`	The number of queues in the hash table
`unsigned long`	`wait_table_bits`	The number of bits in a page address from left to right being used as an index within the `wait_table`
`struct per_cpu_pageset []`	`pageset`	Per CPU pageset for order-0 page allocation (to avoid interrupt-safe spinlock on SMP system)
`struct pglist_data *`	`zone_pgdat`	Points to the descriptor of the parent node
`struct page *`	`zone_mem_map`	The first page in the global `mem_map` that this zone refers to
`unsigned long`	`zone_start_pfn`	The starting physical page frame number of the zone
`char *`	`name`	The string name of the zone: “DMA”, “Normal” or “HighMem”
`unsigned long`	`spanned_pages`	Total size of physical page range, including holes
`unsigned long`	`present_pages`	Total number of physical pages in the zone

Pages

- To keep track of all physical pages, all physical pages are described by an array of `struct page` called `mem_map`



ZONE_DMA
`mem_map`

ZONE_NORMAL
`mem_map`

ZONE_HIGHMEM
`mem_map`

page page page . . . page page . . . page . . .

`mem_map (=contig_page_data.node_mem_map)`

`struct page`

Type	Name	Description
`page_flag_t`	`flags`	The status of the page and mapping of the page to a zone
`atomic_t`	`_count`	The reference count to the page. If it drops to zero, it may be freed
`unsigned long`	`private`	Mapping private opaque data: usually used for `buffer_heads` if PagePrivate set
`struct address_space *`	`mapping`	Points to the address space of a inode when files or devices are memory mapped.
`pgoff_t`	`index`	Our offset within mapping
`struct list_head`	`lru`	Linked to LRU lists of pages if the page is in page cache Linked to `free_area` lists if the page is free and is managed by buddy allocator

Flags describing page status

Flag name	Meaning
`PG_locked`	The page is involved in a disk I/O operation
`PG_error`	An I/O error occurred while transferring the page
`PG_referenced`	The page has been recently accessed for a disk I/O operation. This bit is used during page replacement for moving the page around the LRU lists.
`PG_uptodate`	When a page is read from disk without error, this bit will be set
`PG_dirty`	This indicates if a page needs to be flushed to disk.
`PG_lru`	The page is in the active or inactive page list
`PG_active`	The page is in the active page list
`PG_highmem`	The page frame belongs to the `ZONE_HIGHMEM` zone
`PG_reserved`	The page frame is reserved to kernel code or is unusable

Translating kernel virtual address

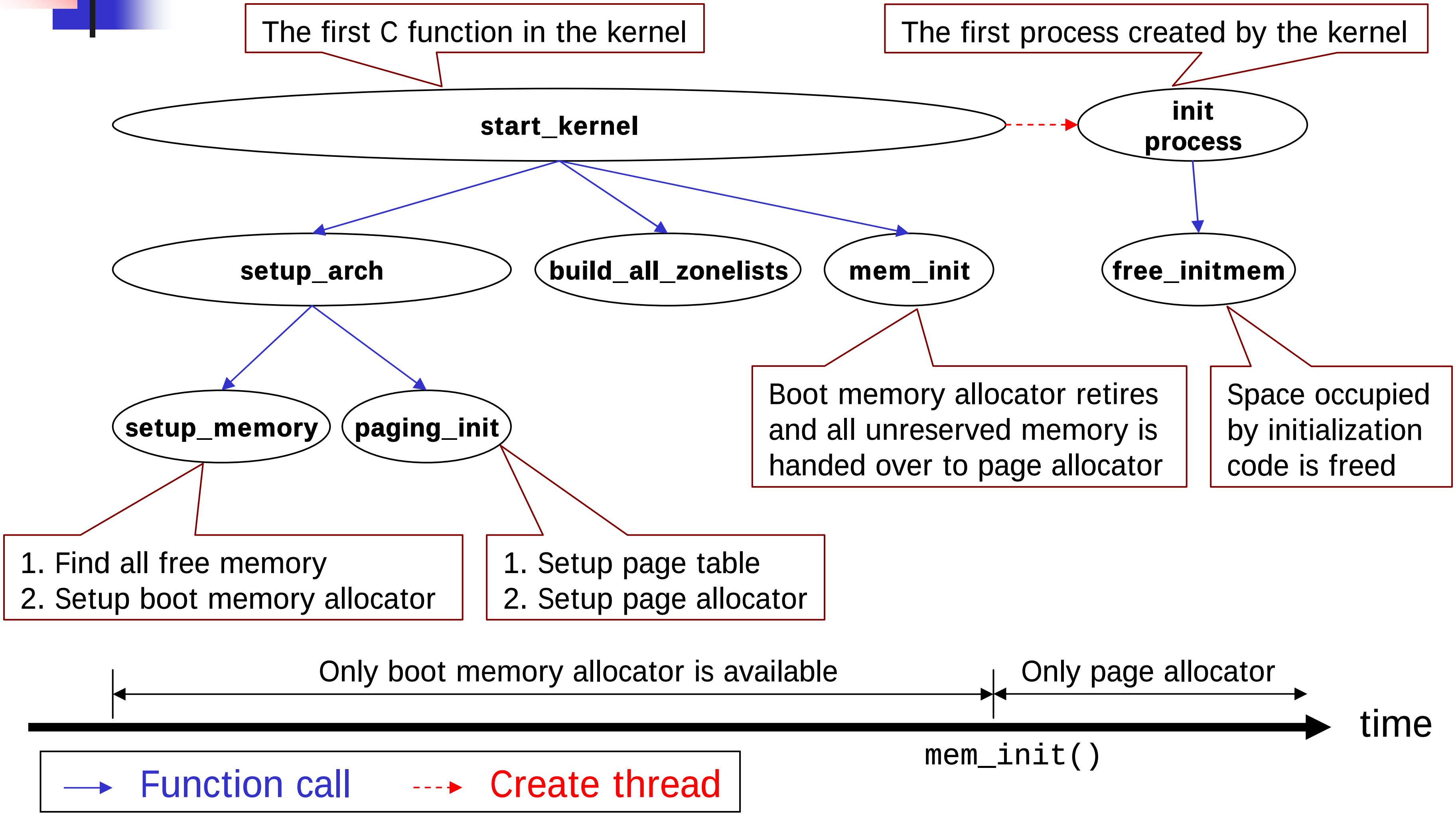
- Recall: memory in `ZONE_DMA` and `ZONE_NORMAL` is direct-mapped and all page frames are described by `mem_map` array
- Kernel virtual address -> physical address
- Physical address -> `struct page`
  - Use physical address as an index into the `mem_map` array

```
#define __pa(x)                    ((unsigned long)(x)-PAGE_OFFSET)
#define pfn_to_page(pfn)           (mem_map + (pfn))
#define virt_to_page(kaddr)        pfn_to_page(__pa(kaddr) >> PAGE_SHIFT)

static inline unsigned long virt_to_phys(volatile void * address)
{
        return __pa(address);
}
```

Boot Memory Allocator

The Flowchart of Initializing All Memory Allocators



The first C function in the kernel

The first process created by the kernel

start_kernel

init process

setup_arch

build_all_zonelists

mem_init

free_initmem

setup_memory

paging_init

Boot memory allocator retires and all unreserved memory is handed over to page allocator

Space occupied by initialization code is freed

1. Find all free memory
2. Setup boot memory allocator

1. Setup page table
2. Setup page allocator

Only boot memory allocator is available

Only page allocator

time

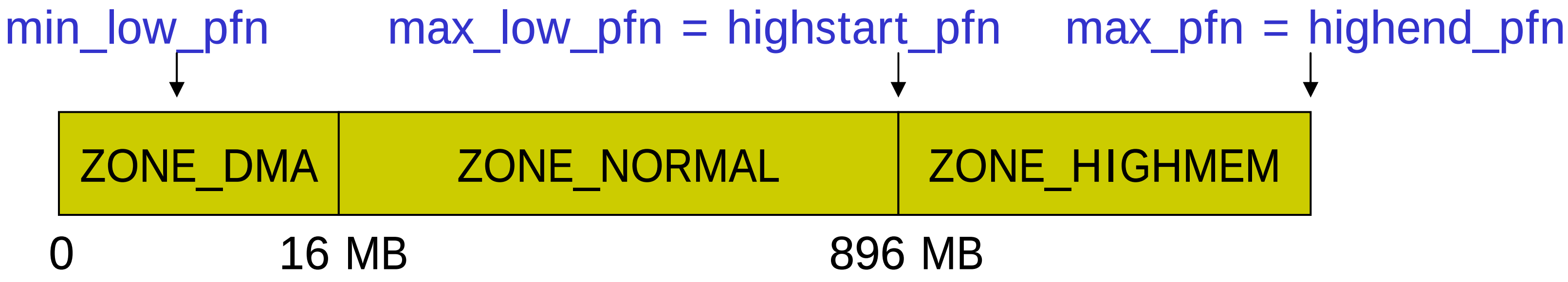
`mem_init()`

Function call

Create thread

Determining the size of each zone

Global variables	Description
`max_pfn`	The last page frame in the system. `find_max_pfn()` determine the value by reading through the e820 map from the BIOS
`min_low_pfn`	the lowest PFN available (the end of kernel image)
`max_low_pfn`	the end PFN of `ZONE_NORMAL`, determined by `find_max_low_pfn()`
`highstart_pfn`, `highend_pfn`	the start and end PFN of `ZONE_HIGHMEM`



min_low_pfn

max_low_pfn = highstart_pfn

max_pfn = highend_pfn

ZONE_DMA

ZONE_NORMAL

ZONE_HIGHMEM

0

16 MB

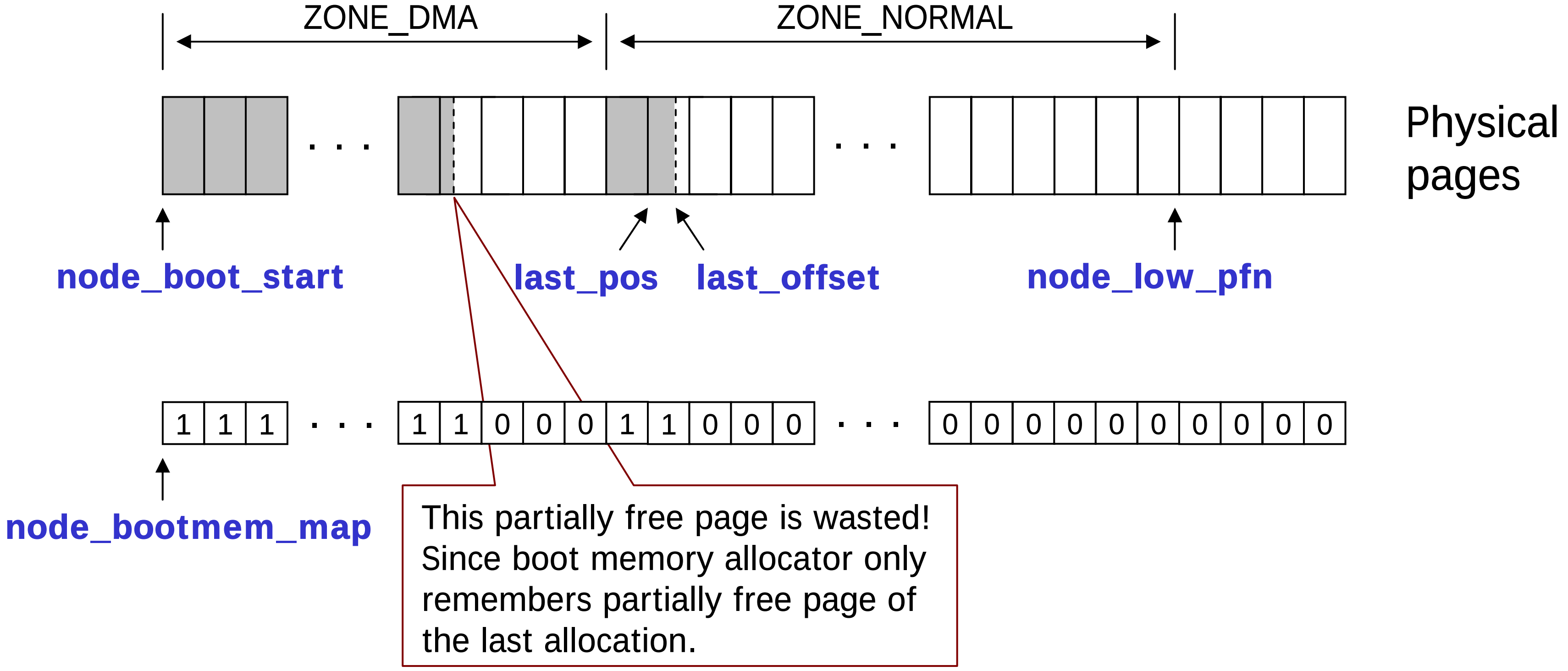
896 MB

Data Structures for Boot Memory Allocator

- A `struct bootmem_data` for each node of memory

Type	Name	Description
`unsigned long`	`node_boot_start`	The starting physical address of the represented block
`unsigned long`	`node_low_pfn`	The end physical address in PFN (end of `ZONE_NORMAL`)
`void *`	`node_bootmem_map`	The location of the bitmap representing allocated or free pages with each bit
`unsigned long`	`last_offset`	The offset within the end page of the last allocation. If 0, the page used is full.
`unsigned long`	`last_pos`	The PFN of the end page of the last allocation. By using this with the `last_offset` field, a test can be made to see if allocations can be merged with the page used for the last allocation rather than using up a full new page.
`unsigned long`	`last_success`	The PFN of the start page of the last allocation. It is used to speed up the search of a block of free memory.

Example of boot memory allocation



ZONE_DMA

ZONE_NORMAL

Physical pages

node_boot_start

last_pos

last_offset

node_low_pfn

1 1 1 . . . 1 1 0 0 0 1 1 0 0 0 . . . 0 0 0 0 0 0 0 0 0 0

node_bootmem_map

This partially free page is wasted! Since boot memory allocator only remembers partially free page of the last allocation.

Pages allocated are gray-colored and marked “1” in the bitmap

`init_bootmem() & free_all_bootmem()`

`unsigned long init_bootmem(unsigned long start, unsigned long page)`

Initialized `contig_page_data.bdata` for page PFN between 0 and `page`. The beginning of usable memory is at the PFN `start` (for bootmem bitmap). The entire bitmap is initialized to 1

`unsigned long free_all_bootmem()`

Used at the boot allocator end of life. It cycles through all pages in the bitmap. For each unallocated page, the `PG_reserved` flag in its `struct page` is cleared, and the page is freed to the physical page allocator (`__free_pages()`) so that it can build its free lists. The pages for boot allocator bitmap are freed too

- Since there is no architecture independent way to detect holes in memory, `init_bootmem()` initializes the entire bitmap to 1. The bitmap will be updated by architecture dependent code later.

`reserve_bootmem()` & `free_bootmem()`

```
void reserve_bootmem(unsigned long addr, unsigned long size)
```

Marks the pages between the address `addr` and `addr+size` reserved (allocated). Requests to partially reserve a page will result in the full page being reserved

```
void free_bootmem(unsigned long addr, unsigned long size)
```

Marks the pages between the address `addr` and `addr+size` as free. An important restriction is that only full pages may be freed. It is never recorded when a page is partially allocated, so, if only partially freed, the full page remains reserved

- Pages used by kernel code, bootmem bitmap are reserved by calling `reserve_bootmem()`
- `free_bootmem()` is used together with `alloc_bootmem()`

`alloc_bootmem()`

```
void * alloc_bootmem(unsigned long size)
```

Allocates `size` number of bytes from `ZONE_NORMAL`. The allocation will be aligned to the L1 hardware cache to get the maximum benefit from the hardware cache.

```
void * alloc_bootmem_low(unsigned long size)
```

Allocates `size` number of bytes from `ZONE_DMA`. The allocation will be aligned to the L1 hardware cache.

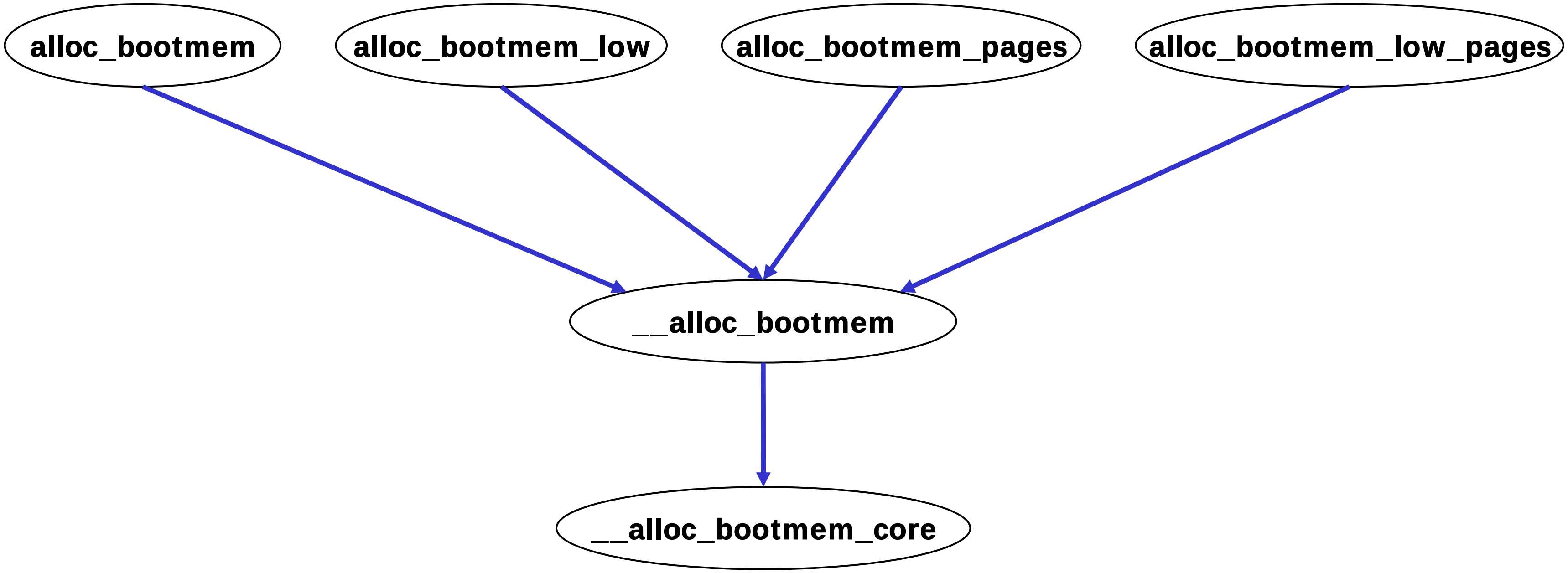
```
void * alloc_bootmem_pages(unsigned long size)
```

Allocates `size` number of bytes from `ZONE_NORMAL` aligned on a page size so that full pages will be returned to the caller.

```
void * alloc_bootmem_low_pages(unsigned long size)
```

Allocates `size` number of bytes from `ZONE_DMA` aligned on a page size so that full pages will be returned to the caller.

Call Graph of `alloc_bootmem()`



alloc_bootmem

alloc_bootmem_low

alloc_bootmem_pages

alloc_bootmem_low_pages

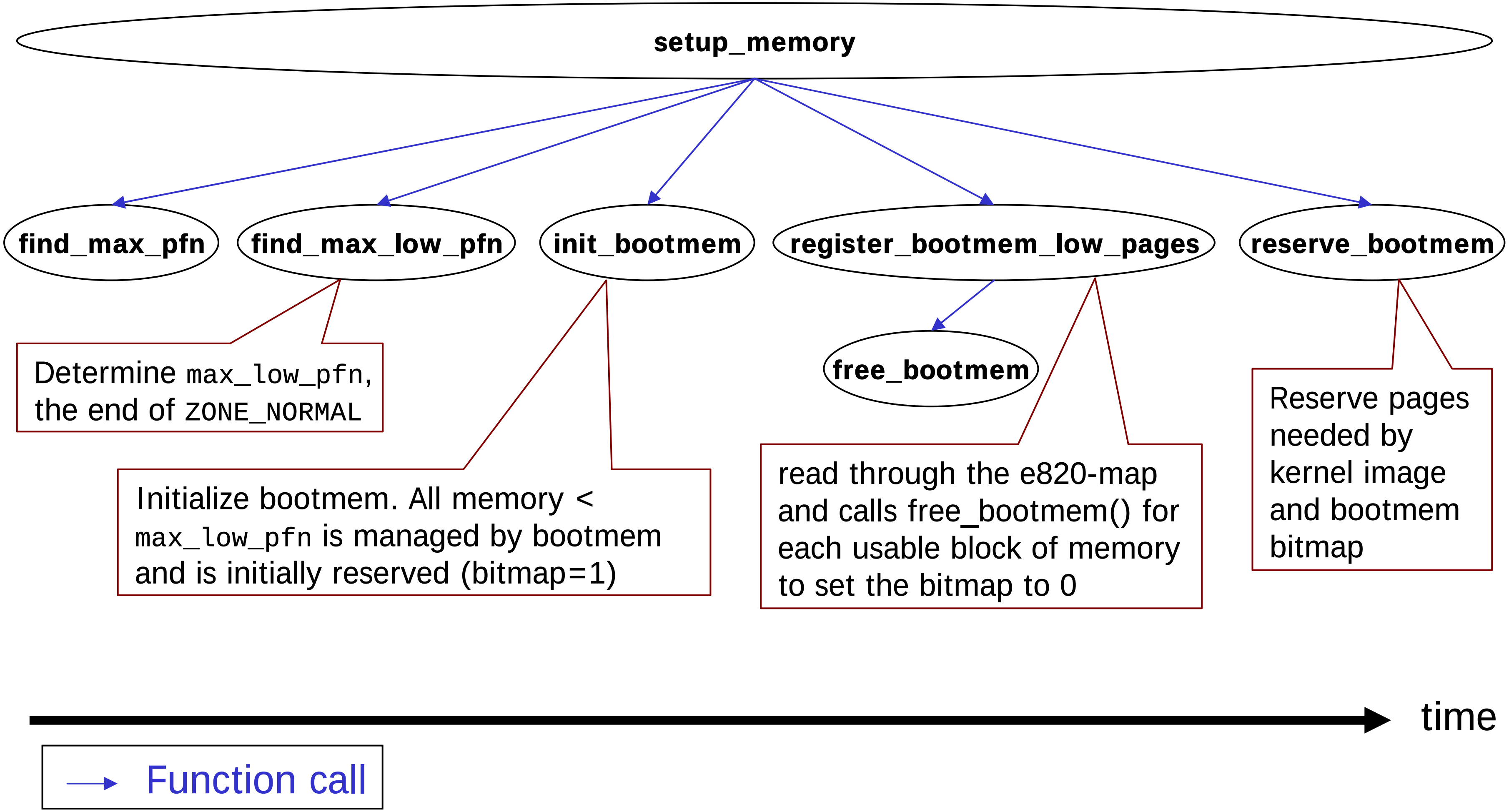
__alloc_bootmem

__alloc_bootmem_core

The core function: `__alloc_bootmem_core()`

- It linearly scans memory starting from preferred address for a block of memory large enough to satisfy the allocation
    - Preferred address may be:
      1. the starting address of a zone or
      2. the address of last successful allocation
- When a satisfied memory block is found, this new allocation can be merged with the previous one if all of the following conditions hold:
    - The page used for the previous allocation (`bootmem_data.pos`) is adjacent to the page found for this allocation
    - The previous page has some free space in it (`bootmem_data.offset != 0`)
    - The alignment is less than `PAGE_SIZE`

The Flowchart of Initializing Boot Memory Allocator



setup_memory

find_max_pfn

find_max_low_pfn

init_bootmem

register_bootmem_low_pages

free_bootmem

reserve_bootmem

Determine `max_low_pfn`, the end of `ZONE_NORMAL`

Initialize bootmem. All memory < `max_low_pfn` is managed by bootmem and is initially reserved (bitmap=1)

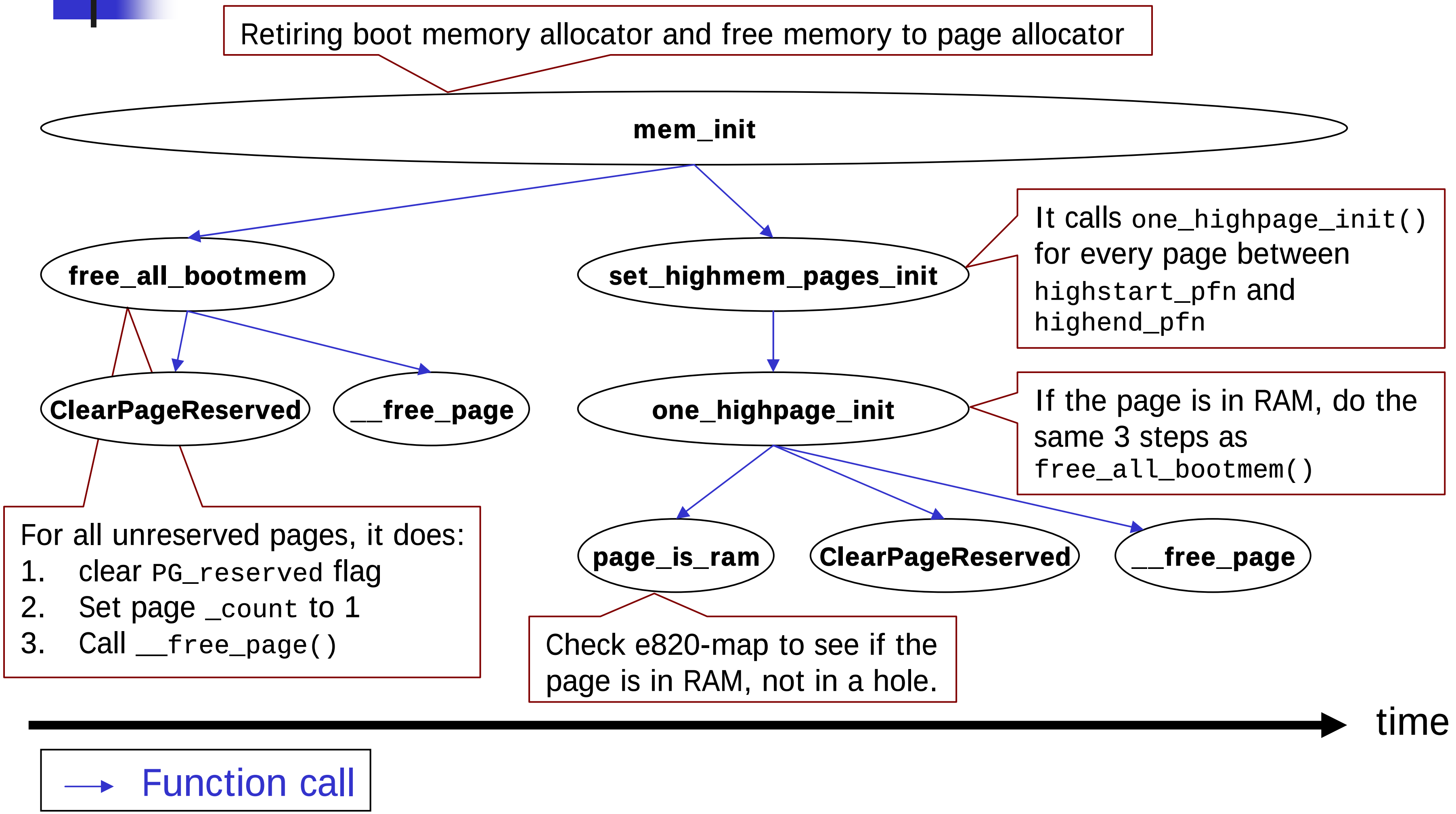
read through the e820-map and calls free_bootmem() for each usable block of memory to set the bitmap to 0

Reserve pages needed by kernel image and bootmem bitmap

time

→ Function call

`mem_init()` - Retiring the Boot Memory Allocator



Retiring boot memory allocator and free memory to page allocator

mem_init

free_all_bootmem

set_highmem_pages_init

It calls `one_highpage_init()` for every page between `highstart_pfn` and `highend_pfn`

ClearPageReserved

__free_page

one_highpage_init

If the page is in RAM, do the same 3 steps as `free_all_bootmem()`

For all unreserved pages, it does:

1. clear `PG_reserved` flag
2. Set page `_count` to 1
3. Call `__free_page()`

page_is_ram

ClearPageReserved

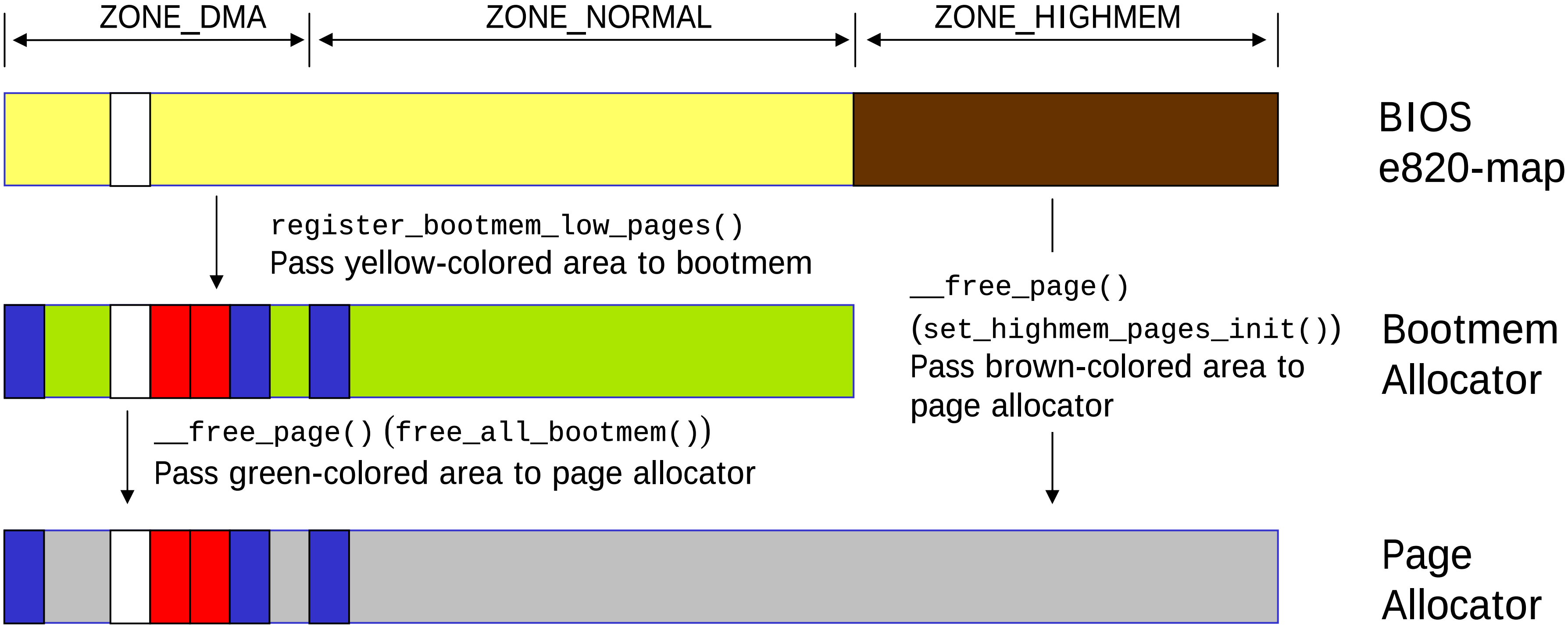
__free_page

Check e820-map to see if the page is in RAM, not in a hole.

time

→ Function call

From Boot Memory Allocator to Page Allocator



ZONE_DMA

ZONE_NORMAL

ZONE_HIGHMEM

BIOS e820-map

`register_bootmem_low_pages()`
Pass yellow-colored area to bootmem

`__free_page()`
`(set_highmem_pages_init())`
Pass brown-colored area to page allocator

Bootmem Allocator

`__free_page()` `(free_all_bootmem())`
Pass green-colored area to page allocator

Page Allocator

Gray-colored area is free and allocable from page allocator

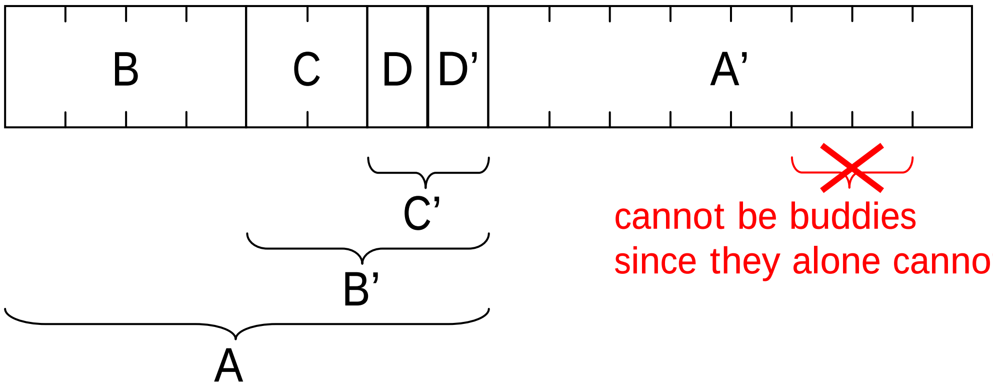
☐ hole 🟥 kernel image 🟦 other

Physical Page Allocator

The Buddy System:
the Algorithm of the Page Allocator

- An allocation scheme that combines free buffer coalescing with a power-of-two allocator
- Memory is split into blocks of pages where each block is a power of two number of pages.
- It create small blocks by repeatedly halving a large block and coalescing adjacent free blocks whenever possible.
- When a block is split, each half is called the buddy of the other.

<letter> and <letter>' are buddies



cannot be buddies
since they alone cannot form a block

struct free_area

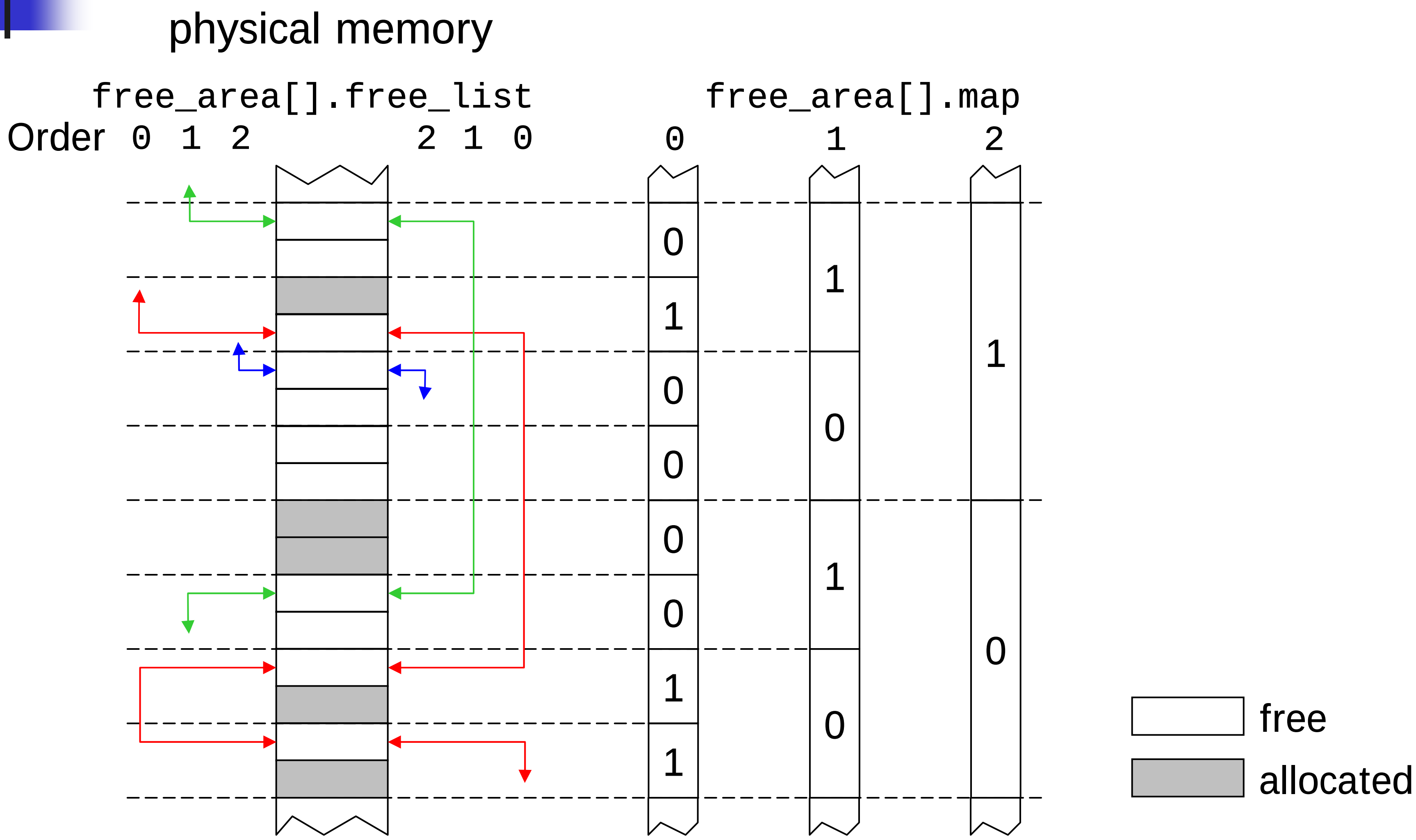
Type	Name	Description
`struct list_head`	`free_list`	A linked list of free page blocks
`unsigned long *`	`map`	A bitmap representing the state of a pair of buddies

- The exponent for the power of two-sized block is referred to as the *order*. An array of `free_area` of size MAX_ORDER is maintained for *orders* from `0` to `MAX_ORDER-1`
- `free_area[i].free_list` is a linked list of free blocks of 2^i page size
- `free_area[i].map` represents the allocation status of all pairs of buddies of 2^i page size. Each time a buddy is allocated or freed, the bit representing the pair of buddies is toggled so that the bit is 0 if the pair of pages are both free or both full and 1 if only one buddy is in use

Think in another way about the meaning of maps in `free_area`

- Each bit in the `free_area[i].map` tells if a pair of buddies is in `free_area[i].free_list`
  - If a bit of the map is 0, the represented buddies are not in the free list. It may be both allocated, or both free and in the free list of higher order
  - If it is 1, exactly one of the buddies is in the free list. It may be reunified with its buddy when it is freed.

Example of the contents of maps in `free_area`



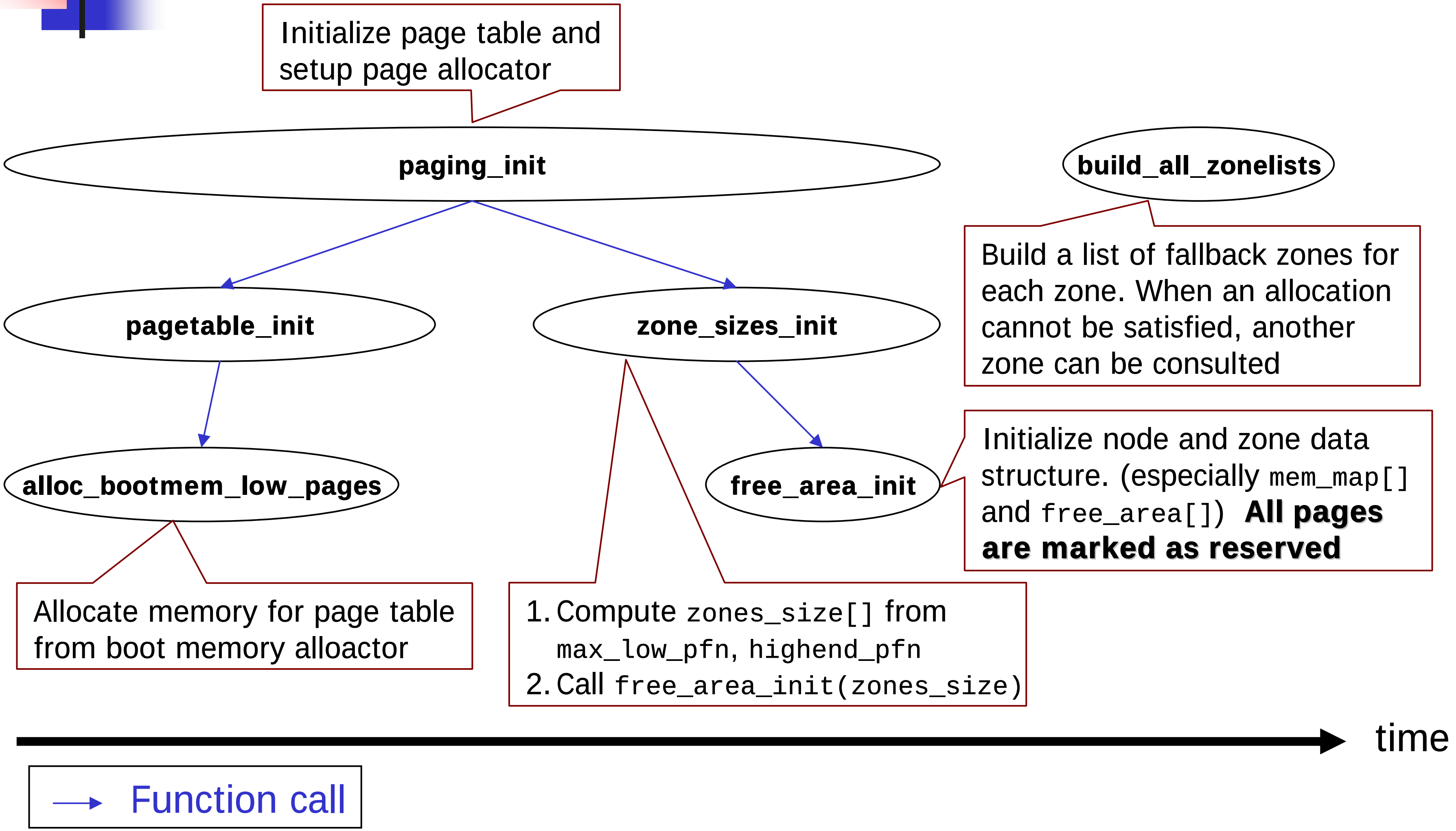
Pseudo Code: Allocating Pages in `free_area`

1. Get a block out from the free list of the desired-`order` free area. If the area is empty, get it from `order+1` free area. Repeat this step until we get a block
2. Toggle the associated bit in the bitmap
3. If the block gotten is from a higher `order` free area, halve it, keep the first half, add the second half to `order-1` free list and toggle the associated bit in the bitmap. Repeat this step until we have a desired-size block.

Pseudo Code:
Freeing Pages in `free_area`

1. For the block being freed, toggle the associated bit in the free area's bitmap. If the value of the bit before the toggle is 0 (i.e. the buddy is still allocated), go to step 3
2. Remove the buddy from the free list and merge it with the block. Then carry the resulting block to `order+1` free area and repeat step 1 and 2.
3. Put the block into the free list.

The Flowchart of Initializing Physical Page Allocator



Initialize page table and setup page allocator

paging_init

pagetable_init

zone_sizes_init

alloc_bootmem_low_pages

free_area_init

build_all_zonelists

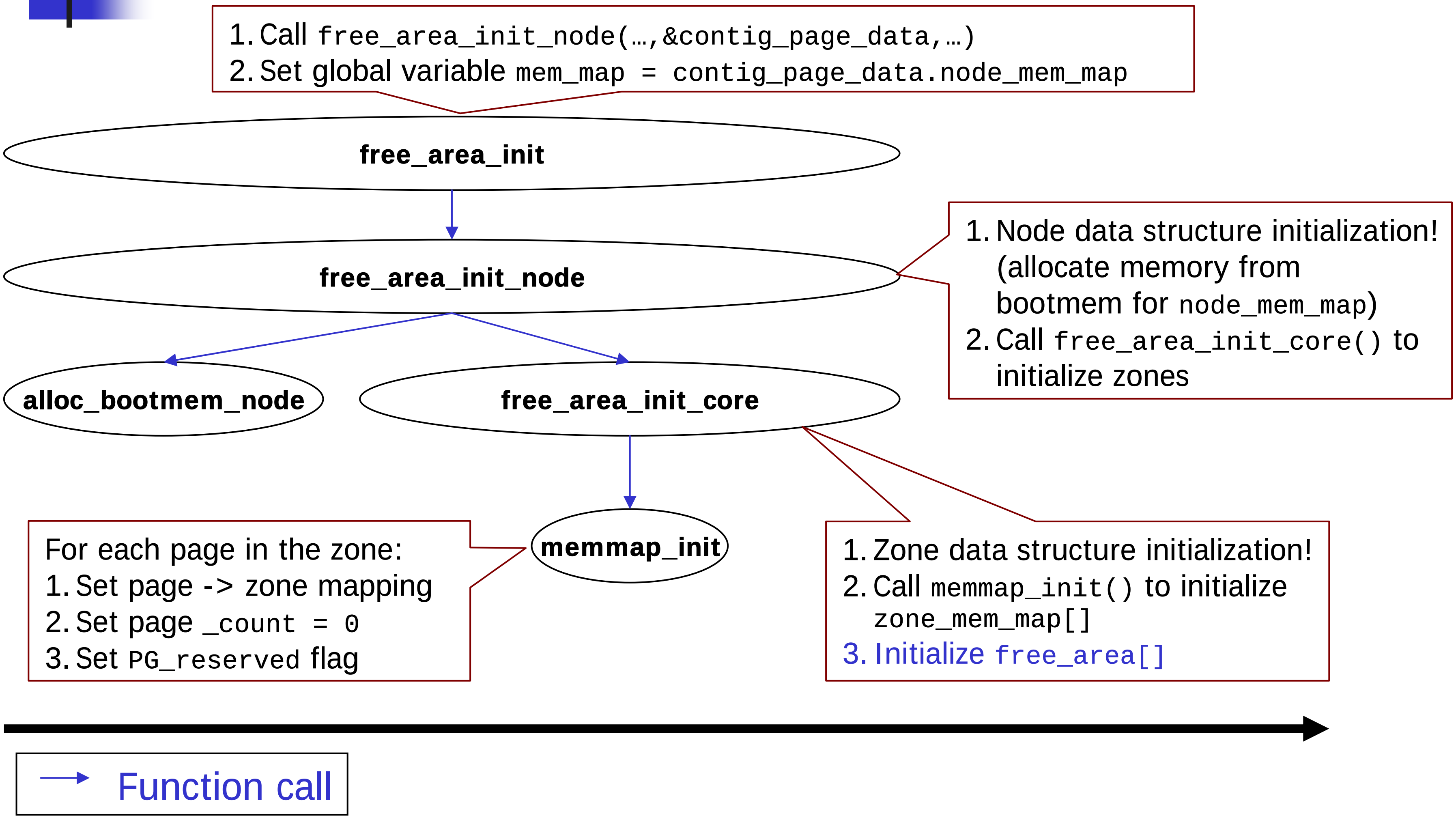
Build a list of fallback zones for each zone. When an allocation cannot be satisfied, another zone can be consulted

Initialize node and zone data structure. (especially `mem_map[]` and `free_area[]`) **All pages are marked as reserved**

Allocate memory for page table from boot memory alloactor

1. Compute `zones_size[]` from `max_low_pfn`, `highend_pfn`
2. Call `free_area_init(zones_size)`

The Flowchart of `free_area_init()`



1. Call `free_area_init_node(…,&contig_page_data,…)`
2. Set global variable `mem_map = contig_page_data.node_mem_map`

free_area_init

free_area_init_node

1. Node data structure initialization! (allocate memory from bootmem for `node_mem_map`)
2. Call `free_area_init_core()` to initialize zones

alloc_bootmem_node

free_area_init_core

memmap_init

For each page in the zone:
1. Set page -> zone mapping
2. Set page `_count = 0`
3. Set `PG_reserved` flag

1. Zone data structure initialization!
2. Call `memmap_init()` to initialize `zone_mem_map[]`
3. Initialize `free_area[]`

→ Function call

Initializing `free_area[]` for each zone

```
for (i = 0; ; i++) {
        unsigned long bitmap_size;

        INIT_LIST_HEAD(&zone->free_area[i].free_list);
        if (i == MAX_ORDER-1) {
                zone->free_area[i].map = NULL;
                break;
        }

        bitmap_size = (size-1) >> (i+4);
        bitmap_size = LONG_ALIGN(bitmap_size+1);
        zone->free_area[i].map =
          (unsigned long *) alloc_bootmem_node(pgdat, bitmap_size);
}
```

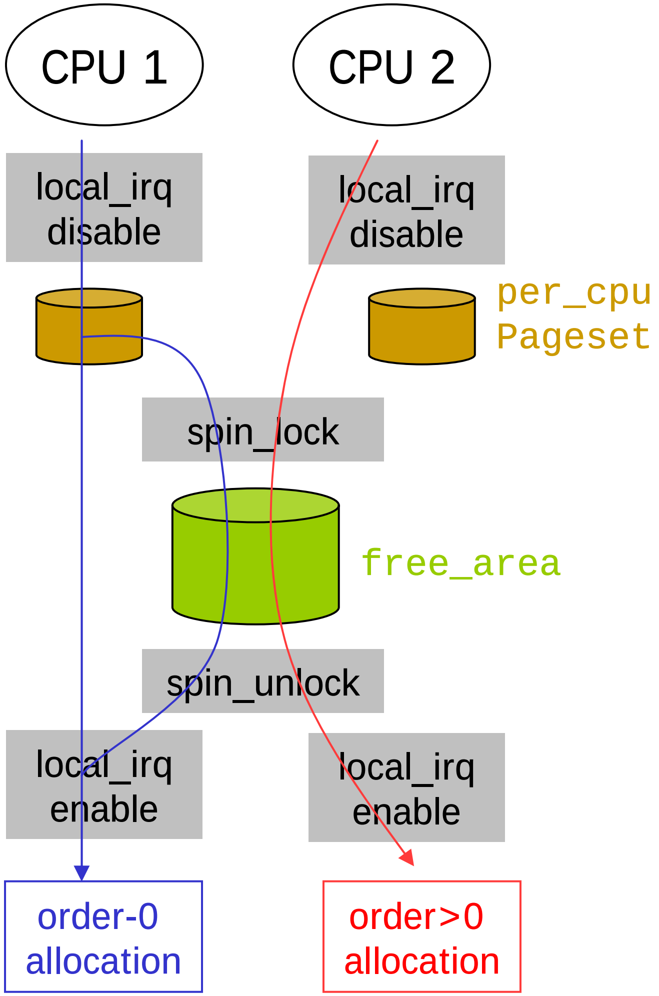
Since MAX_ORDER-1 is the highest order, blocks at this order are not merged. So free area map is not needed.

`size` = number of pages in a zone

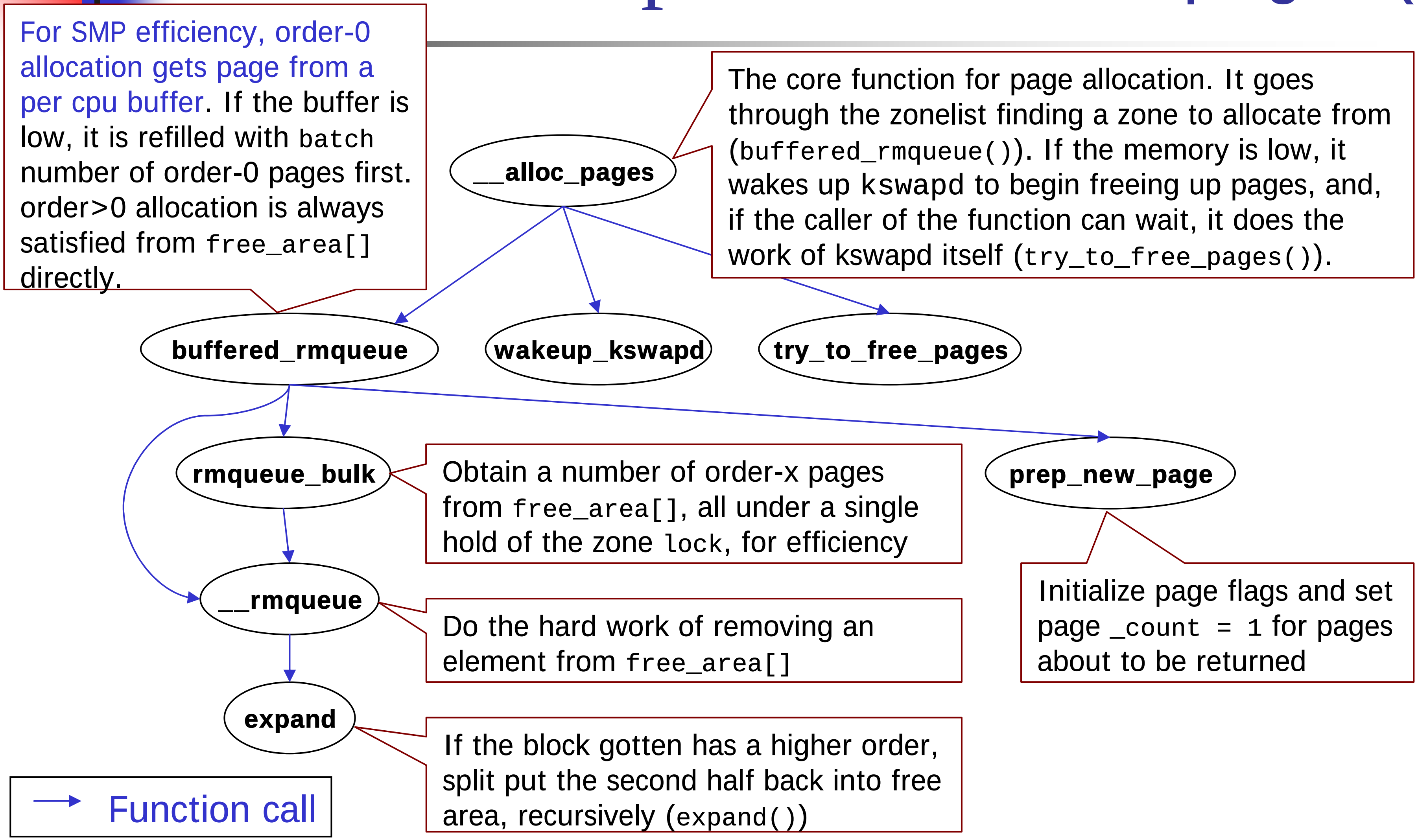
The calculation here (since Linux 2.4) is correct but hard to understand. It may be a little larger than the actual bytes needed. It should be ***`bitmap_size = LONG_ALIGN(((size >> (i+1)) + 7) >> 3)`***. The ***`i`*** is the order of the free area. The ***`+1`*** is because the buddy system uses a single bit to represent two blocks. ***`(size >> i+1)`*** is the number of bits in the bitmap. This value is shifted down by ***`3`*** to get the number of bytes, but we need to have a ***`+7`*** first to round up to byte size.

Per-CPU Page Sets in Linux 2.6

- Recall: `zone[].lock` spinlock protects the `free_area` from concurrent access
  - Lock contention between multiple CPUs may degrade the performance
- Linux 2.6 reduces the number of times acquiring this spinlock by introducing a per CPU page set (`per_cpu_pageset`)
  - It stores only order-0 pages since higher order allocations are rare
  - Order-0 block allocation requires no spinlock being held. But if the page set is low, a number of pages will be allocated in bulk with the spinlock held
  - Side effect: splits and coalescing of blocks for order-0 allocation are delayed



The Call Graph of `__alloc_pages()`



For SMP efficiency, order-0 allocation gets page from a per cpu buffer. If the buffer is low, it is refilled with `batch` number of order-0 pages first. order>0 allocation is always satisfied from `free_area[]` directly.

The core function for page allocation. It goes through the zonelist finding a zone to allocate from (`buffered_rmqueue()`). If the memory is low, it wakes up `kswapd` to begin freeing up pages, and, if the caller of the function can wait, it does the work of kswapd itself (`try_to_free_pages()`).

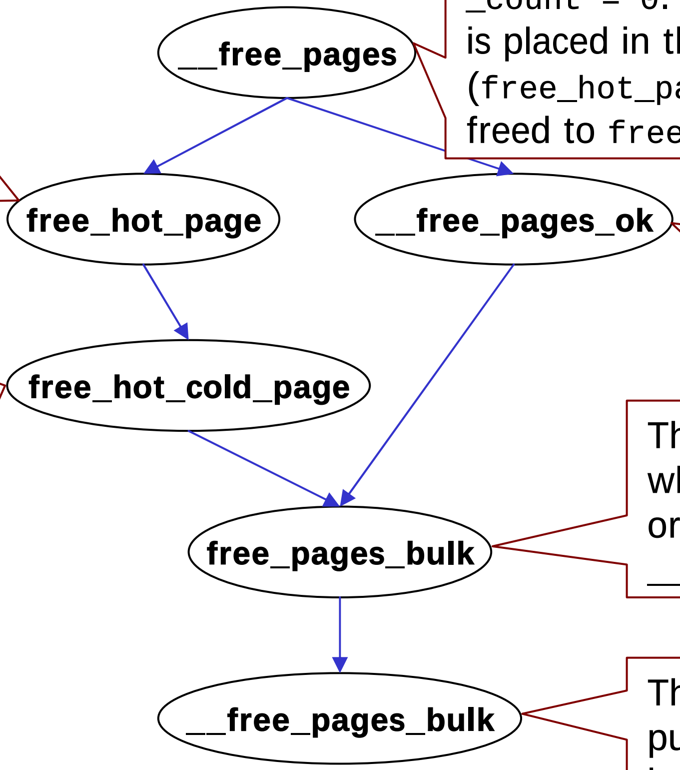
Obtain a number of order-x pages from `free_area[]`, all under a single hold of the zone `lock`, for efficiency

Initialize page flags and set page `_count = 1` for pages about to be returned

Do the hard work of removing an element from `free_area[]`

If the block gotten has a higher order, split put the second half back into free area, recursively (`expand()`)

The Call Graph of `__free_pages()`



There are 2 page sets per CPU. One is for hot pages and the other is for cold pages. `__free_pages()` always free order-0 block into the hot page set.

This function frees a order-0 page into the hot or cold page set. If the page count of the page set for the running CPU has reached the high watermark, a number of pages are freed in bulk from the page set to `free_area[]`

The core function for freeing pages. It set page `_count = 0`. If the block to be freed is order-0, it is placed in the per-cpu pagesets (`free_hot_page()`). Higher-order block is always freed to `free_area[]` (`__free_pages_ok()`)

This is just a wrapper which, in turn, calls `free_pages_bulk()` to free a order-x block.

This function frees a list of blocks, which are in the same zone, of same order. It goes through the list and call `__free_pages_bulk()` for each block.

This function does the hard work of putting a block into free_area[]. If the buddy of the block is also free, merge them into larger block.

→ Function call

Physical Pages Allocation API

```
struct page * alloc_page(unsigned int gfp_mask)
```

Allocates a single page and return a pointer to its `page` structure.

```
struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)
```

Allocates 2^{order} pages and return a pointer to the first page’s `page` structure.

```
unsigned long __get_free_page(unsigned int gfp_mask)
```

Allocates a single page and return a pointer to its virtual address.

```
unsigned long __get_free_pages(unsigned int gfp_mask, unsigned int order)
```

Allocates 2^{order} pages and return a pointer to the first page’s virtual address.

```
unsigned long __get_dma_pages(unsigned int gfp_mask, unsigned int order)
```

Allocates 2^{order} pages from `ZONE_DMA` and return a pointer to the first page’s virtual address.

```
unsigned long get_zeroed_page(unsigned int gfp_mask)
```

Allocates a single page, zero its contents, and return a pointer to its virtual address.

Physical Pages Free API

`void __free_page(struct page *page)`

Frees a single page.

`void __free_pages(struct page *page, unsigned int order)`

Frees 2^{order} pages from the given page.

`void free_page(unsigned long addr)`

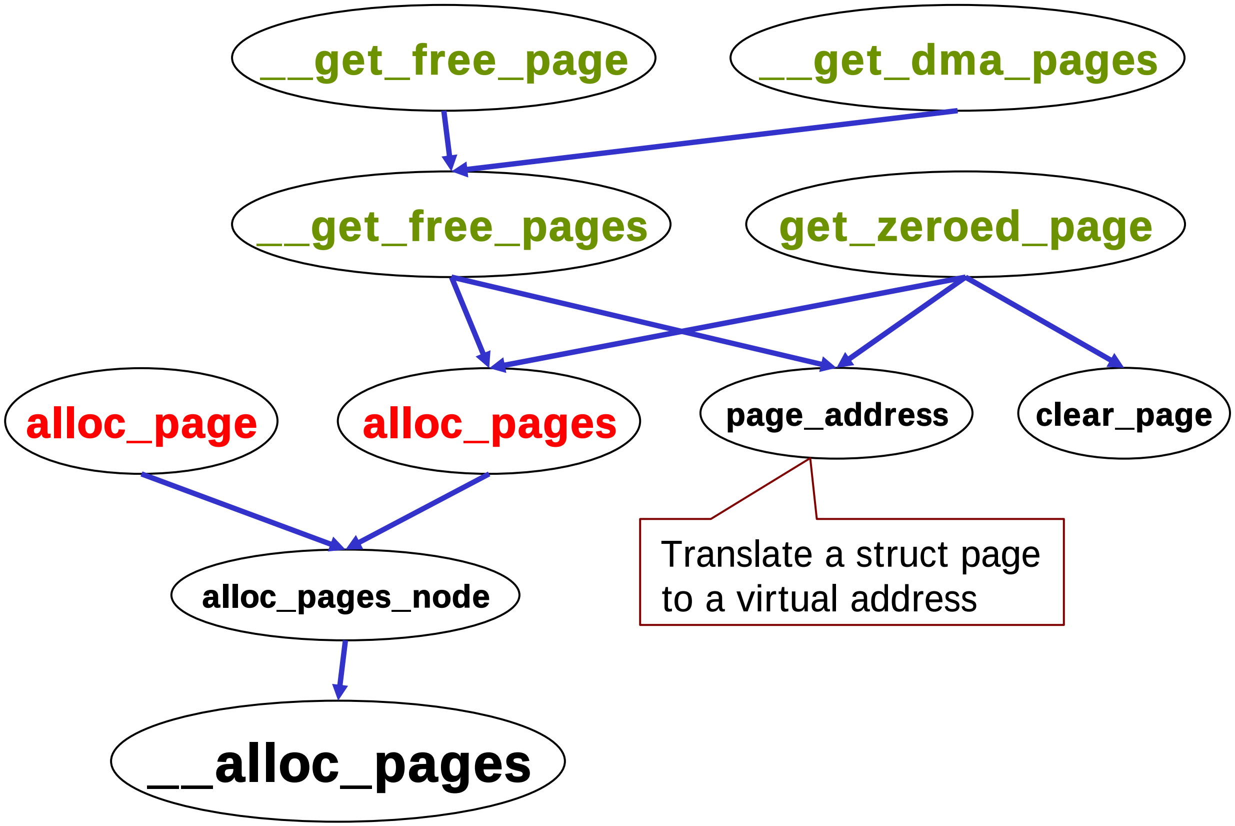
Frees a single page from the given virtual address.

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

Frees 2^{order} pages from the given virtual address.

- There are only two core function for page allocation and free, but two namespaces to them.
  - Pointer to `struct page`: `alloc_page*()` and `__free_page*()`
  - Virtual address: `*get*page*()` and `free_page*()`

The Call Graph of Physical Pages Allocation API



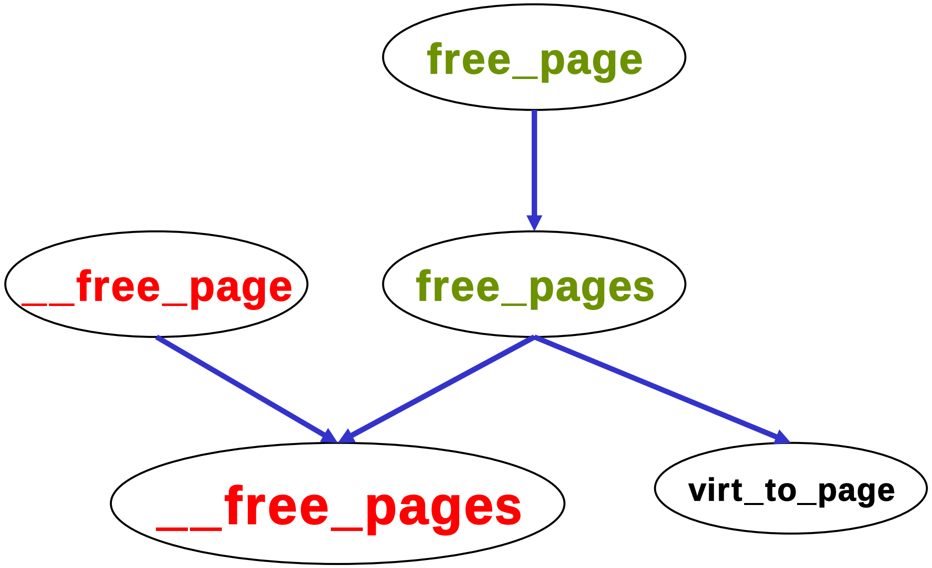
Translate a struct page to a virtual address

→ Function call

🟩 virtual address based

🔴 struct page based

The Call Graph of Physical Pages Free API



free_page

__free_page

free_pages

__free_pages

virt_to_page

→ Function call

● virtual address based

● struct page based

Get Free Page (`gfp_mask`) Flags

- 3 categories of flags
  - Zone modifiers
    - Specify from where to allocate memory
  - Action modifiers
    - Specify how the kernel is supposed to allocate the requested memory
  - Type flags
    - Specify a combination of action and zone modifiers as needed by a certain type of memory allocation
- Don’t use zone or action modifiers directly. Use type flags if there are suitable type flags.

`gfp_mask`: Zone Modifiers

- The kernel allocates memory from `ZONE_NORMAL` if none of the zone modifiers are specified
- If the memory is low, the allocations can fall back on another zone according to the fallback zonelists
- The fallback order
  - `ZONE_HIGHMEM->ZONE_NORMAL->ZONE_DMA`
- Don't use `__GFP_HIGHMEM` with `*get*page*()` or `kmalloc()`
  - They may return an invalid virtual address since the allocated pages are not mapped in the kernel's virtual address space

Flags	Description
`__GFP_DMA`	Allocate only from `ZONE_DMA`
`__GFP_HIGHMEM`	Allocate from `ZONE_HIGHMEM` or `ZONE_NORMAL`

`gfp_mask`: Action Modifiers

Flags	Description
`__GFP_WAIT`	The allocator can sleep
`__GFP_HIGH`	The allocator can access emergency pools of memory
`__GFP_IO`	The allocator can start disk I/O
`__GFP_FS`	The allocator can start filesystem I/O
`__GFP_COLD`	The allocator should use cache cold pages
`__GFP_NOWARN`	The allocator will not print failure warnings
`__GFP_REPEAT`	The allocator will repeat the allocation if it fails
`__GFP_NOFAIL`	The allocator will indefinitely repeat the allocation
`__GFP_NORETRY`	The allocator will never retry if the allocation fails
`__GFP_NOGROW`	Used internally by the slab layer

`gfp_mask`: Type Flags

Flags	Description (AC = Allocator)	Modifier flags
`GFP_ATOMIC`	AC is high priority and must not sleep. This flag is used in interrupt handlers, bottom halves, and other situations where you cannot sleep	`__GFP_HIGH`
`GFP_NOIO`	AC may block, but won't start disk I/O. This flag is used in block I/O code when you cannot cause more disk I/O	`__GFP_WAIT`
`GFP_NOFS`	AC may block and start disk I/O, but won't start filesystem I/O. This flag is used in filesystem code when you cannot start another filesystem operation	`(__GFP_WAIT | __GFP_IO)`
`GFP_KERNEL`	This is for normal allocation. AC may block. This flag is used in process context code when it is safe to sleep	`(__GFP_WAIT | __GFP_IO | __GFP_FS)`
`GFP_USER`	This is for normal allocation. AC may block. This flag is used to allocate memory for user-space processes.	`(__GFP_WAIT | __GFP_IO | __GFP_FS)`
`GFP_HIGHUSER`	AC may block. This flag is used to allocate memory from `ZONE_HIGHMEM` for user-space processes.	`(__GFP_WAIT | __GFP_IO | __GFP_FS |__GFP_HIGHMEM)`
`GFP_DMA`	Device drivers that need DMA-able memory use this flag, usually in combination with one of the above.	`__GFP_DMA`

Reference

- Understanding the Linux Virtual Memory Manager, Mel Gorman, Prentice Hall, 2004
- Understanding the Linux Kernel, Bovet & Cesati, O'REILLY, 2002
- Linux Kernel Development, Robert Love, Sams Publishing, 2003