Physical Memory Management in Linux

Hao-Ran Liu
Table of Contents

- Virtual Address Space and Memory Allocators in Linux
- Describing Physical Memory
- Boot Memory Allocator
- Physical Page Allocator
- Reference
Virtual Address Space and Memory Allocators in Linux
Linux Virtual Address Layout

- **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, \texttt{mem\_map} alone takes up 512 MB memory from lowmem (ZONE\_NORMAL).
4G/4G partition

- Proposed by Red Hat to solve `mmap` 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

0x00000000 0xFFFFFFFF
16MB shared

User Page table

Kernel mode

0x00000000 0xFFFFFFFF
0x02000000 0xF8000000
16MB shared

Kernel Page table

Switch the page tables before system calls

virtual address

0 16 MB 3936 MB 0xFFFFFFFF
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

<table>
<thead>
<tr>
<th>process address space</th>
<th>kernel image</th>
<th>struct page mem_map</th>
<th>the rest memory (for page allocation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages gap</td>
<td>vmalloc address space</td>
<td>pages gap</td>
<td>kmap address space</td>
</tr>
<tr>
<td>fixed mapping</td>
<td>fixed mapping</td>
<td>page gap</td>
<td></td>
</tr>
</tbody>
</table>

- **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

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

All these data structures are initialized by `free_area_init()` at `start_kernel()`
Page Tables vs. `struct pages`

- **Page tables**
  - Used by CPU memory management unit to map virtual address to physical address

- **struct pages**
  - Used by Linux to keep track of the status of all physical pages
  - Some status (e.g. 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 pgl list_data**

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>spinlock_t</td>
<td>lock</td>
<td>Spin lock protecting the descriptor</td>
<td></td>
</tr>
<tr>
<td>unsigned long</td>
<td>free_pages</td>
<td>Number of free pages in the zone</td>
<td></td>
</tr>
<tr>
<td>unsigned long</td>
<td>pages_min</td>
<td>Minimum number of pages of the zone that should remain free</td>
<td>Kswapd</td>
</tr>
<tr>
<td>unsigned long</td>
<td>pages_low, pages_high</td>
<td>Lower and upper threshold value for the zone's page balancing algorithm</td>
<td>Kswapd</td>
</tr>
<tr>
<td>spinlock_t</td>
<td>lru_lock</td>
<td>Spin lock protecting the following two linked lists</td>
<td>Page cache</td>
</tr>
<tr>
<td>struct list_head</td>
<td>active_list, inactive_list</td>
<td>Active and inactive lists (LRU lists) of pages in the zone</td>
<td>Page cache</td>
</tr>
<tr>
<td>unsigned long</td>
<td>nr_active, nr_inactive</td>
<td>The number of pages on the active_list and inactive_list</td>
<td>Page cache</td>
</tr>
</tbody>
</table>
## struct zone (2)

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

```plaintext
ZONE_DMA
mem_map

page  page  page

ZONE_NORMAL
mem_map

page  page

ZONE_HIGHMEM
mem_map

page

mem_map (contig_page_data.node_mem_map)
```
<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>page_flag_t</td>
<td>flags</td>
<td>The status of the page and mapping of the page to a zone</td>
</tr>
<tr>
<td>atomic_t</td>
<td>_count</td>
<td>The reference count to the page.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If it drops to zero, it may be freed</td>
</tr>
<tr>
<td>unsigned long</td>
<td>private</td>
<td>Mapping private opaque data; usually used for buffer_heads if Page_Private set</td>
</tr>
<tr>
<td>struct address_space *</td>
<td>mapping</td>
<td>Points to the address space of a inode when files or devices are memory mapped.</td>
</tr>
<tr>
<td>pgoff_t</td>
<td>index</td>
<td>Our offset within mapping</td>
</tr>
<tr>
<td>struct list_head</td>
<td>lru</td>
<td>Linked to LRU lists of pages if the page is in page cache</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linked to free_area lists if the page is free and is managed by buddy allocator</td>
</tr>
</tbody>
</table>
## Flags describing page status

<table>
<thead>
<tr>
<th>Flag name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PG_locked</td>
<td>The page is involved in a disk I/O operation</td>
</tr>
<tr>
<td>PG_error</td>
<td>An I/O error occurred while transferring the page</td>
</tr>
<tr>
<td>PG_referenced</td>
<td>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.</td>
</tr>
<tr>
<td>PG_uptodate</td>
<td>When a page is read from disk without error, this bit will be set</td>
</tr>
<tr>
<td>PG_dirty</td>
<td>This indicates if a page needs to be flushed to disk</td>
</tr>
<tr>
<td>PG_lru</td>
<td>The page is in the active or inactive page list</td>
</tr>
<tr>
<td>PG_active</td>
<td>The page is in the active page list</td>
</tr>
<tr>
<td>PG_highmem</td>
<td>The page frame belongs to the ZONE_HIGHMEM zone</td>
</tr>
<tr>
<td>PG_reserved</td>
<td>The page frame is reserved for kernel code or is unusable</td>
</tr>
</tbody>
</table>
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

```c
#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);
}
```
The Flowchart of Initializing All Memory Allocators

The first C function in the kernel

The first process created by the kernel

start_kernel

setup_arch

build_all_zonelists

mem_init

free_initmem

setup_memory

paging_init

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

1. Setup page table
   2. Setup page allocator

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

Space occupied by initialization code is freed

Only boot memory allocator is available

Only page allocator

Function call

Create thread

mem_init()
Determining the size of each zone

<table>
<thead>
<tr>
<th>Global variables</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{max_PFN}$</td>
<td>The last page frame in the system. $\text{find_max_PFN}()$ determine the value by reading through the e820 map from the BIOS.</td>
</tr>
<tr>
<td>$\text{min_low_PFN}$</td>
<td>the lowest PFN available (the end of kernel image)</td>
</tr>
<tr>
<td>$\text{max_low_PFN}$</td>
<td>the end PFN of ZONE_NORMAL, determined by $\text{find_max_low_PFN}()$</td>
</tr>
<tr>
<td>$\text{highstart_PFN}$, $\text{highend_PFN}$</td>
<td>the start and end PFN of ZONE_HIGHMEM</td>
</tr>
</tbody>
</table>

- $\text{min\_low\_PFN}$
- $\text{max\_low\_PFN} = \text{highstart\_PFN}$
- $\text{max\_PFN} = \text{highend\_PFN}$

<table>
<thead>
<tr>
<th>Zone</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZONE_DMA</td>
<td>0</td>
</tr>
<tr>
<td>ZONE_NORMAL</td>
<td>16 MB</td>
</tr>
<tr>
<td>ZONE_HIGHMEM</td>
<td>896 MB</td>
</tr>
</tbody>
</table>
### Data Structures for Boot Memory Allocator

- A `struct bootmem_data` for each node of memory

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned long</td>
<td>node_boot_start</td>
<td>The starting physical address of the represented block</td>
</tr>
<tr>
<td>unsigned long</td>
<td>node_low_pfn</td>
<td>The end physical address in PFN (end of <code>ZONE_NORMAL</code>)</td>
</tr>
<tr>
<td>void *</td>
<td>node_bootmem_map</td>
<td>The location of the bitmap representing allocated or free pages with each bit</td>
</tr>
<tr>
<td>unsigned long</td>
<td>last_offset</td>
<td>The offset within the end page of the last allocation. If 0, the page used is full.</td>
</tr>
<tr>
<td>unsigned long</td>
<td>last_pos</td>
<td>The PFN of the end page of the last allocation. By using this with the <code>last_offset</code> 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.</td>
</tr>
<tr>
<td>unsigned long</td>
<td>last_success</td>
<td>The PFN of the start page of the last allocation. It is used to speed up the search of a block of free memory.</td>
</tr>
</tbody>
</table>
Example of boot memory 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()

```c
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.

```c
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()

```c
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**
  - Determine \( \text{max}\_\text{low}\_\text{pf}\_n \), the end of \textit{ZONE}\_\textit{NORMAL}.

- **find_max_low_pfn**
  - Initialize bootmem. All memory < \( \text{max}\_\text{low}\_\text{pf}\_n \) is managed by bootmem and is initially reserved (bitmap=1).

- **init_bootmem**

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

- **reserve_bootmem**
  - Reserve pages needed by kernel image and bootmem bitmap.

**Function call**
mem_init() -
Retiring the Boot Memory Allocator

For all unreserved pages, it does:
1. clear PG_reserved flag
2. Set page_count to 1
3. Call __free_page()

It calls one_highpage_init() for every page between high_start_pfn and high_end_pfn
If the page is in RAM, do the same 3 steps as free_all_bootmem()

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

- __free_page() (free_all_bootmem())
  Pass green-colored area to 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 creates 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

\begin{table}[h]
\begin{tabular}{|c|c|c|c|c|}
\hline
B & C & D & D' & A' \\
\hline
\end{tabular}
\end{table}

\begin{itemize}
\item[C'] cannot be buddies since they alone cannot form a block
\item[B']
\item[A]
\end{itemize}
struct free_area

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct list_head</td>
<td>free_list</td>
<td>A linked list of free page blocks</td>
</tr>
<tr>
<td>unsigned long *</td>
<td>map</td>
<td>A bitmap representing the state of a pair of buddies</td>
</tr>
</tbody>
</table>

- The exponent for the power of two-sized block is referred to as the order. An array of \( \text{free}_\text{area} \) of size \( \text{MAX}_\text{ORDER} \) is maintained for orders from 0 to \( \text{MAX}_\text{ORDER} - 1 \).
- \( \text{free}_\text{area}[i].\text{free}_\text{list} \) is a linked list of free blocks of \( 2^i \) page size.
- \( \text{free}_\text{area}[i].\text{map} \) 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`

### Physical Memory

<table>
<thead>
<tr>
<th>Order</th>
<th><code>free_area[].free_list</code></th>
<th><code>free_area[].map</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Free**: Light grey
- **Allocated**: Dark grey
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

1. **paging_init**
   - Initialize page table and setup page allocator

2. **pagetable_init**
   - Allocate memory for page table from boot memory allocator

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

4. **free_area_init**

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

---

Function call → time
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**
    - **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. Node data structure initialization!
   (allocate memory from bootmem for `node_mem_map`)
2. Call `free_area_init_core()` to initialize zones

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

→ Function call
Initializing \texttt{free\_area[]} for each zone

\begin{verbatim}
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);
}
\end{verbatim}

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

\texttt{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 $\texttt{bitmap\_size} = \text{LONG\_ALIGN}((\text{size} >> (i+1)) + 7) >> 3$.

$\texttt{i}$ is the order of the free area. The $+1$ is because the buddy system uses a single bit to represent two blocks. $(\text{size} >> (i+1))$ 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.

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()`).

Initialize page flags and set `page_count = 1` for pages about to be returned.
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 `__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

```c
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.
```
There are only two core functions 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

- __get_free_page
- __get_dma_pages
- __get_free_pages
- get_zeroed_page
- alloc_page
- alloc_pages
- alloc_pages_node
- alloc_pages__node
- page_address
- clear_page

→ Function call

- green: virtual address based
- red: struct page based

Translate a struct page to a virtual address
The Call Graph of Physical Pages Free API

- __free_page
- free_page
- __free_pages
- 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 \textit{where} to allocate memory
  - Action modifiers
    - Specify \textit{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 \textit{type} of memory allocation

- Don’t use zone or action modifiers directly. Use type flags if there are suitable type flags.
The kernel allocates memory from \texttt{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:

\begin{itemize}
  \item \texttt{ZONE\_HIGHMEM} -> \texttt{ZONE\_NORMAL} -> \texttt{ZONE\_DMA}
\end{itemize}

Don’t use \texttt{\_\_GFP\_HI GHMEM} with \texttt{*get\_*page*()} or \texttt{kmalloc()}.

They may return an invalid virtual address since the allocated pages are not mapped in the kernel’s virtual address space.

<table>
<thead>
<tr>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{__GFP_DMA}</td>
<td>Allocate only from \texttt{ZONE_DMA}</td>
</tr>
<tr>
<td>\texttt{__GFP_HI GHMEM}</td>
<td>Allocate from \texttt{ZONE_HI GHMEM} or \texttt{ZONE_NORMAL}</td>
</tr>
</tbody>
</table>
### gfp_mask: Action Modifiers

<table>
<thead>
<tr>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__GFP_WAIT</td>
<td>The allocator can sleep</td>
</tr>
<tr>
<td>__GFP_HIGH</td>
<td>The allocator can access emergency pools of memory</td>
</tr>
<tr>
<td>__GFP_IO</td>
<td>The allocator can start disk I/O</td>
</tr>
<tr>
<td>__GFP_FS</td>
<td>The allocator can start filesystem I/O</td>
</tr>
<tr>
<td>__GFP_COLD</td>
<td>The allocator should use cache cold pages</td>
</tr>
<tr>
<td>__GFP_NOWARN</td>
<td>The allocator will not print failure warnings</td>
</tr>
<tr>
<td>__GFP_REPEAT</td>
<td>The allocator will repeat the allocation if it fails</td>
</tr>
<tr>
<td>__GFP_NOFAIL</td>
<td>The allocator will indefinitely repeat the allocation</td>
</tr>
<tr>
<td>__GFP_NORETRY</td>
<td>The allocator will never retry if the allocation fails</td>
</tr>
<tr>
<td>__GFP_NOGROW</td>
<td>Used internally by the slab layer</td>
</tr>
</tbody>
</table>
## gfp_mask: Type Flags

<table>
<thead>
<tr>
<th>Flags</th>
<th>Description (AC = Allocator)</th>
<th>Modifier flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFP_ATOMIC</td>
<td>AC is high priority and must not sleep. This flag is used in interrupt handlers, bottom halves, and other situations where you cannot sleep.</td>
<td>__GFP_HIGH</td>
</tr>
<tr>
<td>GFP_NOIO</td>
<td>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</td>
<td>__GFP_WAIT</td>
</tr>
<tr>
<td>GFP_NOFS</td>
<td>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.</td>
<td>(__GFP_WAIT</td>
</tr>
<tr>
<td>GFP_KERNEL</td>
<td>This is for normal allocation. AC may block. This flag is used in process context code when it is safe to sleep.</td>
<td>(__GFP_WAIT</td>
</tr>
<tr>
<td>GFP_USER</td>
<td>This is for normal allocation. AC may block. This flag is used to allocate memory for user-space processes.</td>
<td>(__GFP_WAIT</td>
</tr>
<tr>
<td>GFP_HIUSER</td>
<td>AC may block. This flag is used to allocate memory from ZONE_HIGHMEM for user-space processes.</td>
<td>(__GFP_WAIT</td>
</tr>
<tr>
<td>GFP_DMA</td>
<td>Device drivers that need DMA-able memory use this flag, usually in combination with one of the above.</td>
<td>__GFP_DMA</td>
</tr>
</tbody>
</table>
Reference

- Understanding the Linux Virtual Memory Manager, Mel Gorman, Prentice Hall, 2004
- Understanding the Linux Kernel, Bovet & Cesati, O’REILLY, 2002