

SIXUENET

To Learn Without Thinking Is To Be Useless, To Think Without Learning Is To Lose
(<http://www.mysixue.com/>)

MEMORY MANAGEMENT (2): PHYSICAL MEMORY MANAGEMENT

📅 April 20, 2019 ([Http://Www.Mysixue.Com/?P=112](http://Www.Mysixue.Com/?P=112)) 👤 JL
([Http://Www.Mysixue.Com/?Author=1](http://Www.Mysixue.Com/?Author=1)) 💬

2 Physical memory management

2.1 Overview

The entire memory management system can be divided into two parts : the first part is the management of physical memory, and the second part is the management of virtual memory .

Physical objects are onboard memory management of physical memory (DDRAM) , which divide the physical memory in pages , and these pages into a pool inside . The purpose is to provide physical memory management API to memory to the consumer , the consumer When memory is needed , the API is called to apply to the physical memory management system , and the physical memory management system takes the physical page frame from the pool to the consumer .

The object of virtual memory management is virtual address space . Its main function is to allocate virtual pages , maintain page tables , and associate virtual pages with disks or physical page frames . The details are detailed in the next chapter .

This chapter mainly focuses on the physical memory management system . We can understand the system in depth from the following perspectives :

- The internal details of the pool , that is , how to describe the physical page frame and how to manage the physical page frame within the pool . We call this part "memory organization"
- 📦💧 to fill the pool with water , that is, how to divide an onboard DRAM into page frames , and add these page frames to the pool . We call this part "initialized memory management"

➤📁📁 to provide consumers with an interface to obtain memory , that is, how to allocate physical memory .
This part is called "physical memory allocation"

2.2 Pool - Memory Organization

2.2.1 Concept introduction

Let's start with the simplest case . Let's say there is a single-core CPU with a piece of DDRAM onboard . The size of DDRAM is 2GB, and the addresses are continuous .

How is the physical memory organized in this case ?

- we know the physical memory can be divided into one page, if page_size is 4KB, then there are a total of 512KB two pages.
- these pages has been classified into different zones (Zone) , typical regional types . 3 Species : DMA memory field (the ZONE_DMA) , low memory field (the ZONE_NORMAL) , high memory field (the ZONE_HIGHMEM) .
- 🔗🔗🔗🔗🔗🔗 areas belong to a certain node (node) .

Therefore , the memory organization of the above situation is :

```

N ode
  | - ZONE_DMA
    | - Pages
  | - ZONE_NORMAL
    | - Pages
  | - ZONE_HIGHMEM
    | - Pages

```

The concept of Page and why we introduced the concept of Page has been introduced in the first chapter , the main purpose is to facilitate the management of memory and address translation .

Zone reason for introducing this concept is not difficult to understand , the first chapter we mentioned the concept of high-end memory , the kernel virtual address space is divided into logical and virtual addresses , logical addresses one mapping to the low-end area of physical memory (ZONE_NORMAL) , the virtual address is dynamically mapped to the high-end area of the physical memory (ZONE_HIGHMEM) . In some CPU architectures , DMA has limited addressing capabilities and can only access the address space of 0-16M , so the physical address is in the low-end area. A DMA area (ZONE_DMA) is divided .

Then why introduce the concept of Node ?

Imagine a case , assuming single-core CPU onboard 2 blocks DDRAM, the size of each block is 128MB, 2 blocks DDRAM address discontinuity , one from 0x20000000 beginning , another from 0x40000000 start .

In this case , we must use 2 Ge node to describe , each Node has its own under Zones and Pages.

```

N ode1
  | - ZONE_DMA
    | - Pages
  | - ZONE_NORMAL
    | - Pages
  | - ZONE_HIGHMEM
    | - Pages

N ode2

```

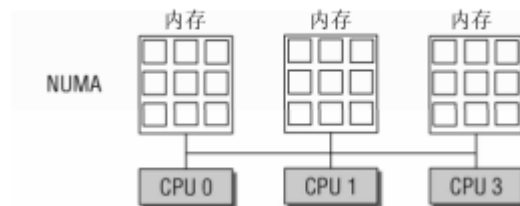
```

|- ZONE_DMA
    |- Pages
|- ZONE_NORMAL
    |- Pages
|- ZONE_HIGHMEM
    |- Pages

```

In addition to the above-mentioned case where there are large holes in the physical address space , multiple nodes are required , and there is also a case where multiple nodes are used .

One type of computer is called NUMA computer (non-uniform memory access) . NUMA is always a multi-processor computer . Each CPU of the system has local memory, which can support particularly fast access. Each processor is connected by a bus to support access to the local memory of other CPUs , which is of course slower than accessing the local memory.



In this case, multiple nodes are also needed to describe memory . When allocating memory for a process, the kernel always tries to perform on the NUMA node associated with the currently running CPU , which can improve performance .

Although it is the most ideal situation to allocate memory on the current node, it is not always feasible. For example, the memory of the node may have been exhausted. For such cases, each node provides a spare list (with the help of struct zonelist). The list contains the memory domains of other nodes , which can be used to allocate memory instead of the current node. The lower the position of the list item, the less suitable it is for distribution .

So far , we have introduced the following concepts:

```

Node
Zone & Zone_type
Page
Zonelist

```

Next , we will introduce these data structures in detail from the code level

2.2.2 Main data structure

Node (pg_data_t)

pg_data_t is the basic element used to represent the node and is defined as follows .

Header file : include/linux/ mmzone.h

<pre> typedef struct pglist_data { } pg_data_t; </pre>	Comment
-----------------------------------------------------------------	---------

struct zone node_zones[MAX_NR_ZONES]	Represents all memory domains under the node . The struct zone structure represents a memory domain , and the data structure will be described in detail below MAX_NR_ZONES is a macro definition , which represents the maximum number of memory domains possible under a node
struct zonelist node_zonelists[MAX_ZONELISTS]	Specify the list of memory domains of the standby node , so that when the current node has no free space, memory is allocated on the standby node
int nr_zones	Represents the number of memory domains under the node
struct page *node_mem_map	node_mem_map is a pointer to an array of page instances, used to describe all physical memory pages of the node . It contains the pages of all memory domains in the node
struct bootmem_data *bdata	During system startup, the kernel also needs to use memory before the memory management subsystem is initialized (in addition, some memory must be reserved for initializing the memory management subsystem) . To solve this problem, the kernel used later to explain the bootstrap memory allocator (the Boot Memory allocator) . bdata points to an instance of the bootstrap memory allocator data structure
unsigned long node_start_pfn	node_start_pfn is the logical number of the first page frame of the node. All system nodes sequentially numbered page frame , the number of each page frame is globally unique (not just the unique nodal point) node_start_pfn is always 0 in the UMA system , because there is only one node, so its first page frame number is always 0
unsigned long node_present_pages	node_present_pages specifies the number of page frames in the node
unsigned long node_spanned_pages	And node_spanned_pages gives the length of the node calculated in the unit of page frame. The values of the two are not necessarily the same, because there may be some holes in the node, which do not correspond to the real Page frame
int node_id	The global node ID . NUMA nodes in the system are numbered starting from 0
struct pglist_data *pgdat_next	Connect to the next memory node, all nodes in the system are connected by a singly linked list, and the end is marked by a null pointer
wait_queue_head_t kswapd_wait	It is the waiting queue of the swap daemon , which will be used when swapping out the page frame from the node (this article will not discuss the page exchange mechanism in detail for the time being)
struct task_struct *kswapd	Point to the task_struct of the exchange daemon responsible for the node
int kswapd_max_order	Used for the implementation of the page swap subsystem, used to define the length of the area that needs to be released (we are not currently interested)

node_states (node state management)

If there is more than one node in the system, the kernel will maintain a bitmap to provide status information of each node. The state is specified with a bit mask, and the following values can be used:

Header file : include /linux/ nodemask.h

enum node_states	Comment
N_POSSIBLE	The node may become online at some point

N_ONLINE	Node is online
N_NORMAL_MEMORY	Common memory domain
<pre> #ifdef CONFIG_HIGHMEM N_HIGH_MEMORY, #else N_HIGH_MEMORY = N_NORMAL_MEMORY, #endif </pre>	Nodes have ordinary or high-end memory domains
N_CPU	The node has one or more CPUs
NR_NODE_STATES	

The states N_POSSIBLE , N_ONLINE and N_CPU are used for hot swapping of CPU and memory, and these characteristics are not considered in this book. The signs necessary for memory management are N_HIGH_MEMORY and N_NORMAL_MEMORY . If the node has normal or high memory, use N_HIGH_MEMORY . Only when the node does not have high memory, set N_NORMAL_MEMORY.

Two auxiliary functions are used to set or clear a bit in a bit field or a specific node :

Header file : include /linux/ nodemask.h

```
void node_set_state(int node, enum node_states state)
```

```
void node_clear_state(int node, enum node_states state)
```

In addition, the macro **for_each_node_state** (__node, __state) is used to iterate all nodes in a specific state, while **for_each_online_node** (node) iterates all active nodes.

If the kernel is compiled to only support a single node (that is, using a flat memory model), there is no node bitmap, and the above function to manipulate the bitmap becomes a no-op

Zone (struct zone)

The kernel uses the zone structure to describe the memory domain. It is defined as follows:

Header file : include/linux/ mmzone.h

struct zone	Comment
-------------	---------

unsigned long watermark[NR_WMARK]	<p>The "watermark" used when the page is swapped out . This is an array with 3 elements :</p> <p>watermark [WMARK_MIN],watermark [WMARK_LOW],watermark [WMARK_HIGH]</p> <p>If the memory is insufficient, the kernel can write the page to the hard disk. These 3 members affect the behavior of the exchange daemon :</p> <ul style="list-style-type: none"> ➤ If the free page than Watermark [WMARK_HIGH] , the state of the memory field is desirable ➤ If the number of free pages below Watermark [WMARK_LOW] , the kernel starts pages swapped out to the hard disk ➤ If the number of free pages below the Watermark [WMARK_MIN] , then the page reclamation work pressure is relatively large, as in urgent need of free memory domain page. "In-depth Linux kernel architecture of the 18 chapters ," will discuss the kernel used in various ways to ease this situation <p>There will be a short article at the end of this section that specifically introduces how the initial values of these 3 watermarks are obtained.</p>
unsigned long lowmem_reserve[MAX_NR_ZONES]	The lowmem_reserve array specifies several pages for various memory domains, which are used for critical memory allocations that cannot fail anyway. The share of each memory domain is determined according to the importance
struct pglist_data *zone_pgdat	The association between the memory domain and the parent node is established by zone_pgdat , and zone_pgdat points to the corresponding pglist_data instance
<pre>#ifndef CONFIG_SPARSEMEM unsigned long *pageblock_flags; #endif /* CONFIG_SPARSEMEM */</pre>	<pre>/* * Flags for a pageblock_nr_pages block. See pageblock-flags.h. * In SPARSEMEM, this map is stored in struct mem_section */</pre> <p>The significance of this parameter , see 2.4.3 section "Auxiliary functions and variables"</p>
struct per_cpu_pageset __percpu *pageset	<p>pageset is an array used to implement a list of hot / cold page frames for each CPU . The kernel uses these lists to store "fresh" pages that can be used to satisfy the implementation. However, the cache states corresponding to hot and cold page frames are different: some page frames are also likely to be still in the cache, so they can be accessed quickly, so they are called hot; uncached page frames are the opposite, so they are called cold. of.</p> <p>This article does not intend to discuss the hot / cold page mechanism in detail , about structper_cpu_pageset For the details of the data structure , if you are interested, you can read "Deep into the Linux Kernel Architecture 3.2.2 Data Structure - Hot and Cold Page "</p>
unsigned long zone_start_pfn	<pre>/* zone_start_pfn == zone_start_paddr >> PAGE_SHIFT */</pre> <p>zone_start_pfn is the index of the first page frame of the memory domain</p>
unsigned long spanned_pages	<p>spanned_pages is the total pages spanned by the zone, including holes, which is calculated as:</p> <p>spanned_pages = zone_end_pfn – zone_start_pfn;</p>

unsigned long present_pages	present_pages is physical pages existing within the zone, which is calculated as: present_pages = spanned_pages – absent_pages(pages in holes);
unsigned long managed_pages	managed_pages is present pages managed by the buddy system, which is calculated as (reserved_pages includes pages allocated by the bootmem allocator): managed_pages = present_pages – reserved_pages;
const char *name;	name is a string that holds the customary name of the memory area. There are currently 3 options available: Normal , DMA and HighMem
seqlock_t span_seqlock	See the comments in the source code for details * Locking rules: (The details of the lock use are introduced here)
wait_queue_head_t *wait_table	These three variables implement a waiting queue that can be used by processes waiting for a page to become available. The details of the mechanism will not be discussed in this article . The intuitive concept is well understood: the process is queued and waits for certain conditions. When the condition becomes true, the kernel will notify the process to resume work
unsigned long wait_table_hash_nr_entries	
unsigned long wait_table_bits	
ZONE_PADDING(pad1_)	<p>The special aspect of this structure is that it is divided into several parts by ZONE_PADDING . This is because the access to the zone structure is very frequent. On a multi-processor system, there are usually different CPUs trying to access structure members at the same time. Therefore, a lock mechanism is used to prevent them from interfering with each other and avoid errors and inconsistencies. Since the kernel accesses this structure very frequently, it will frequently acquire the two spin locks zone->lock and zone->lru_lock of the structure.</p> <p>If the data is stored in the CPU cache, it will be processed faster. The cache is divided into rows, and each row is responsible for a different memory area. The kernel uses the ZONE_PADDING macro to generate a "padding" field and add it to the structure to ensure that each spinlock is in its own cache line . The compiler keyword __cacheline_maxaligned_in_smp is also used to achieve optimal cache alignment.</p> <p>The structure of the first part and the last part also by filling in fields separated from each other. Both do not contain a lock, the main purpose is to keep the data in a cache line, for fast access, thereby eliminating the need to load data from memory (with the CPU compared cache memory is relatively slow). The increase in the length of the structure due to padding is negligible, especially the relatively few instances of the zone structure in the kernel memory .</p>
struct free_area free_area[MAX_ORDER]	<p>free_area is an array of data structures with the same name, used to implement the buddy system. Each array element represents some contiguous memory area of a certain fixed length. For the management of free memory pages contained in each area, free_area is a starting point.</p> <p>The data structure used here is worthy of discussion. The following article will specifically introduce the implementation details of the partner system , and we will study this structure carefully.</p>

unsigned long flags	<p>The flags describe the current state of the memory domain. The following flags are allowed (see the source code comments for the meaning of the flags)</p> <pre><mmzone.h> enum zone_flags { ZONE_RECLAIM_LOCKED, ZONE_OOM_LOCKED, ZONE_CONGESTED, ZONE_DIRTY, ZONE_WRITEBACK, ZONE_FAIR_DEPLETED, };</pre> <p>It is also possible that none of these flags are set. This is the normal state of the memory domain.</p> <p>The kernel provides 3 auxiliary functions for testing and setting the flags of the memory domain:</p> <pre><mmzone.h> void zone_set_flag(struct zone *zone, zone_flags_t flag) int zone_test_and_set_flag(struct zone *zone, zone_flags_t flag) void zone_clear_flag(struct zone *zone, zone_flags_t flag)</pre>
spinlock_t lock	<p>See the comments in the source code for details</p> <p>* Locking rules:</p> <p>..... .. (The details of the lock use are introduced here)</p>
ZONE_PADDING(_pad2_)	Same as above ZONE_PADDING(_pad 1 _)
/* Write-intensive fields used by page reclaim */	<p>These signs are related to page recycling and page exchange mechanisms , and will not be discussed in this article</p> <p>For details, see "In-depth Linux Kernel Architecture Chapter 18 "</p>
spinlock_t lru_lock	
struct lruvec lruvec	
atomic_long_t inactive_age	
unsigned long percpu_drift_mark	
compact_xxx	
...	
ZONE_PADDING(_pad3_)	Same as above ZONE_PADDING(_pad 1 _)
atomic_long_t vm_stat[NR_VM_ZONE_STAT_ITEMS]	<p>vm_stat maintains a large number of statistical information about the memory domain. Since most of the information maintained therein is of little significance at present, please refer to "In-depth Linux Kernel Architecture Section 17.7.1 " for a detailed discussion of the structure . Now, as long as you know that many places in the kernel will update the information. The auxiliary function zone_page_state is used to read the information in vm_stat :</p> <pre><vmstat.h> static inline unsigned long zone_page_state(struct zone *zone, enum zone_stat_item item)</pre> <p>For example, you can set the item parameter to NR_FREE_PAGES to get the number of free pages in the memory domain ; set it to NR_xxx to view other information</p>

Zone type (zone_type)

According to the background knowledge introduced in Chapter 1 , the memory domain can be divided into many different types , which are defined as follows :

Header file : include/linux/ mmzone.h

enum zone_type	Comment
<pre>#ifdef CONFIG_ZONE_DMA ZONE_DMA, #endif</pre>	<p>ZONE_DMA is used when there are devices that are not able to do DMA to all of addressable memory (ZONE_NORMAL). Then we carve out the portion of memory that is needed for these devices.</p> <p>The range is arch specific.</p> <p>Some examples :</p> <p>Architecture Limit</p> <hr/> <p>parisc, ia64, sparc <4G</p> <p>s390 <2G</p> <p>arm Various</p> <p>i386, x86_64 and multiple other arches</p> <p><16M</p>
<pre>#ifdef CONFIG_ZONE_DMA32 ZONE_DMA32, #endif</pre>	<p>x86_64 needs two ZONE_DMAS because it supports devices that are only able to do DMA to the lower 16M but also 32 bit devices that can only do DMA areas below 4G.</p>
<pre>ZONE_NORMAL,</pre>	<p>Normal addressable memory is in ZONE_NORMAL. DMA operations can be performed on pages in ZONE_NORMAL if the DMA devices support transfers to all addressable memory.</p>
<pre>#ifdef CONFIG_HIGHMEM ZONE_HIGHMEM, #endif</pre>	<p>ZONE_HIGHMEM marks the physical memory beyond the kernel that can be directly mapped</p> <p>A memory area that is only addressable by the kernel through mapping portions into its own address space. This is for example used by i386 to allow the kernel to address the memory beyond 900MB. The kernel will set up special mappings (page table entries on i386) for each page that the kernel needs to access .</p>
<pre>ZONE_MOVABLE,</pre>	<p>The kernel defines a pseudo memory domain ZONE_MOVABLE , which is needed in the mechanism to prevent physical memory fragmentation.</p> <p>This mechanism will be explained in more detail later .</p>
<pre>__MAX_NR_ZONES</pre>	<p>Serves as an end marker, this constant is used when the kernel wants to iterate over all memory areas in the system.</p>

Note that depending on the configuration at compile time, some memory domains may not need to be considered. For example, in a 64 -bit system, the high-end memory domain is not required. If 32 -bit peripherals that can only access memory below 4 GiB are supported , the DMA32 memory domain is needed .

Page

The page frame represents the smallest unit of system memory, and an instance of struct page is created for each page in the memory . Kernel programmers need to take care to keep this structure as small as possible, because even under a moderate memory configuration, the system's memory will also be broken down into a large number of pages. For example, the standard page length of the IA-32 system is 4 KiB . When the main memory size is 384 MiB , there are approximately 100,000 pages in total . According to today's standards, this capacity is not very large, but the number of pages is already very impressive.

This is why the kernel tries its best to keep the struct page as small as possible. In a typical system, due to the huge number of pages, small changes to the page structure may also cause the physical memory required to store all page instances to skyrocket.

The widespread use of pages increases the difficulty of maintaining the length of the structure: Many parts of memory management use pages for various purposes. One part of the kernel may completely depend on the specific information provided by the struct page, and this information may be completely useless for another part of the kernel. This part depends on other information provided by the struct page, and this part of the information may also be for other parts of the kernel. Completely useless, etc.

The combination of the C language is very suitable for this problem, although it fails to increase the clarity of the struct page. Consider an example: a physical memory page can be mapped to the virtual address space through different page tables in multiple places, and the kernel wants to keep track of how many places the page is mapped. To this end, there is a counter in the struct page to count the number of mappings. If a page is used for the slab allocator (a method of subdividing the whole page into smaller parts, see the introduction to slab below), then it can be ensured that only the kernel will use the page and no other places. Therefore, the mapping count information is redundant. Therefore, the kernel can reinterpret this field to indicate how many small memory objects the page is subdivided into. In the data structure definition, this double interpretation is as follows:

```
struct page {
    ...
    union {
        atomic_t _mapcount; /* The page table entry count mapped in the memory management subsystem,
                             * Used to indicate whether the page has been mapped, and also used to limit the reverse
mapping search.
                             */
        unsigned int inuse; /* For SLUB allocator: the number of objects */
    };
    ...
}
```

Definition of page

Header file : include/linux/mm_types.h

The format of this structure is independent of the architecture and does not depend on the type of CPU used. Each page frame is described by this structure.

struct page	Comment
/* First double word block */	

<p>unsigned long flags;</p>	<p>Flags stores architecture-independent flags, used to describe the attributes of the page .</p> <p>The different attributes of the page are described by a series of page flags, which are stored as bits in the flags member of the struct page . These flags are independent of the architecture used and therefore cannot provide information specific to the CPU or computer (this information is stored in the page table, see below) .</p> <p>Each flag is defined by the macros in include/linux/page-flags.h . In addition, some macros are generated for flag setting, deletion, and query. In doing so, the kernel follows a general naming scheme (using the C language connector " ## " to automatically build these macros , please read the code for details , for example, < TESTPAGEFLAG > is used to build query macros).</p> <p>The implementation of these macros is atomic. Although some of them consist of several statements, special processor commands are used to ensure that they behave like a single statement. That is, these statements cannot be interrupted, otherwise it will cause race conditions.</p> <p>What page flags are available? page-flags.h in enum pageflags</p> <p>All the signs are listed .</p> <p>(Here temporarily as to the meaning of each sign detailed description , followed by a need to supplement , are interested can read the "in-depth Linux kernel architecture " on page 122 Ye , 3.2.2 section)</p>
<pre>union { struct address_space *mapping; void *s_mem; }</pre>	<p>Mapping is very important and more complicated , according to the difference in its lowest bit :</p> <p>➤👉➤ ⬢🌀🌀 lowest bit is 0 , it points to the inode address_space (representing the address space of a file , a simple understanding is how to read file data from this file), or null.</p> <p>➤👉➤ ⬢🌀🌀 lowest bit is 1, it points to the anon_vma object , which means it is an anonymous physical page</p> <p>The specific role of mapping is still not understood , so let's improve it when we understand it later .</p> <p>s_mem: slab first object (I don't understand it now)</p>
<p>/* Second double word */</p>	
<pre>struct { union { pgoff_t index; void *freelist; }; };</pre>	<p>index: I do n't understand it temporarily</p> <p>freelist: If this page is used for the slab allocator , then freelist points to the header management data of the slab cache</p>

<pre> union { #if defined(CONFIG_HAVE_CMPXCHG_ DOUBLE) && \ defined(CONFIG_HAVE_ALIGNED_ _STRUCT_PAGE) /* Used for cmpxchg_doub le in slub */ unsigned long counters; #else unsigned counters; #endif </pre>	I don't understand the details yet
<pre> struct { union { atomic_t _map count; struct { /* SLUB */ unsi gned inuse:16; unsig ned objects:15; unsi gned frozen:1; }; int units; /* SLOB */ }; atomic_t _coun t; }; </pre>	I don't understand the details yet
<pre> unsigned int active; /* S LAB */ }; </pre>	I don't understand the details yet
};	This super long struct structure is finally over
Third double word block	
<pre> union { ... } </pre>	It is a long union, anyway, not now understand , temporarily not stick out the details , b ack to refine the understanding of the
/* Remainder is not double word aligned */	

<pre>union { unsigned long private; #ifdef USE_SPLIT_PTE_PTLOCKS #ifdef ALLOC_SPLIT_PTLOCKS spinlock_t *ptl; #else spinlock_t ptl; #endif #endif struct kmem_cache *slab_cache; };</pre>	<p>private : is a pointer to "private" data, which will be ignored by virtual memory management. Depending on the purpose of the page, the pointer can be used in different ways . In most cases, it is used to associate the page with the data buffer. This article does not go into details</p> <p>ptl : don't understand</p> <p>slab_cache : for SLUB dispenser: pointing slab pointer slab_cache and the aforementioned freelist and inuse members are used in the slub allocator. We do not need to pay attention to the specific layout of these members, if the kernel compilation does not enable slub allocator support, these members will not be used</p>
<pre>#ifdef CONFIG_MEMCG struct mem_cgroup *mem_cgrou up; #endif</pre>	<p>Don't understand</p>
<pre>#if defined(WANT_PAGE_VIRTUAL) void *virtual; #endif</pre>	<p>Virtual is used for pages in the upper memory area, in other words, it cannot be directly mapped to pages in kernel memory. virtual is used to store the virtual address of the page. If there is no mapping, it is NULL .</p> <p>According to the preprocessor statement <code>#if defined(WANT_PAGE_VIRTUAL)</code> , virtual can become part of the struct page only if the corresponding macro is defined . Currently, only a few architectures are like this, namely Motorola m68k , FRV and Extensa .</p> <p>All other architectures use a different scheme to address virtual memory pages. Its core is to find a hash table of all high-end memory page frames. Section 2.4 "Kernel Mapping" will study this technology in more detail. Processing the hash table requires some mathematical operations, which is relatively slow on the aforementioned computers, so this direct method can only be chosen.</p>
<pre>#ifdef CONFIG_KMEMCHECK void * shadow ; #endif</pre>	<p>kmemcheck wants to track the status of each byte in a page; this is a pointer to such a status block. NULL if not tracked.</p>
<pre>#ifdef LAST_CPUID_NOT_IN_PAGE_FLAGS int _last_cpupid; #endif</pre>	<p>Don't understand</p>

Zonelist

In the `pg_data_t` structure , there is such an element:

```
typedef struct pglist_data {
    ...
    struct zonelist node_zonelists [ MAX_ZONELISTS ];
    ...
} pg_data_t ;
```

`node_zonelists` specifies the list of memory domains of the standby node , so that when the current node has no free space, memory is allocated on the standby node .

How is the zonelist structure defined ?

The header file is in : include/linux/ mmzone.h

We post the code directly

MAX_ZONELISTS definition :

```
/*
 * The NUMA zonelists are doubled because we need zonelists that restrict the
 * allocations to a single node for __GFP_THISNODE.
 *
 * [0]: Zonelist with fallback
 * [1]: No fallback (__GFP_THISNODE)
 */
#define MAX_ZONELISTS 2
#else
#define MAX_ZONELISTS 1
#endif
```

zonelist defined

```
/* Maximum number of zones on a zonelist */
#define MAX_ZONES_PER_ZONELIST (MAX_NUMNODES * MAX_NR_ZONES)

#ifdef CONFIG_NUMA

/*
 * This struct contains information about a zone in a zonelist. It is stored
 * here to avoid dereferences into large structures and lookups of tables
 */
struct zoneref {
    struct zone * zone ; /* Pointer to actual zone */
    int zone_idx ; /* zone_idx(zoneref->zone) */
};

/*
 * One allocation request operates on a zonelist. A zonelist
 * is a list of zones, the first one is the 'goal' of the
 * allocation, the other zones are fallback zones, in decreasing
 * priority.
 *
 * To speed the reading of the zonelist, the zonerefs contain the zone index
 * of the entry being read. Helper functions to access information given
 * a struct zoneref are
 *
 * zonelist_zone() - Return the struct zone * for an entry in _zonerefs
 * zonelist_zone_idx() - Return the index of the zone for an entry
 * zonelist_node_idx() - Return the index of the node for an entry
 */
```

```
*/  
struct zonelist {  
    struct zoneref _zonerefs [ MAX_ZONES_PER_ZONELIST + 1 ];  
};
```

A zonelist structure represents a list of spare memory domains . Since the spare list must include all memory domains of all nodes, it is composed of `MAX_NUMNODES * MAX_NZ_ZONES` items, plus a null pointer to mark the end of the list.

The creation of the standby list must be carefully considered . The most "cheap" node must be at the top of the list , followed by the "cheap" meaning that the access cost is the least . Because when the current node has no free space , the system will from the start of the first element of the standby list , "cheap" at the top , we can guarantee the best system performance .

The creation of the alternate list is delegated to the `build_zonelists` function , which is automatically called when the memory management system is initialized .

Since the backup list is not very related to driver development , it is more like an operating system strategy level thing , so this article will not elaborate on it for the time being .

Calculation of memory domain watermark (`min_free_kbytes`)

Before calculating various watermarks, the kernel first determines the minimum amount of memory space that needs to be reserved for critical allocation. This value increases non-linearly with the size of the available memory and is stored in the global variable `min_free_kbytes` . The following figure summarizes this non-linear proportional relationship. The horizontal axis of the main diagram uses logarithmic coordinates, and the horizontal axis of the illustration uses ordinary coordinates. The illustration enlarges the change curve of the total memory capacity between 0 and 4 GiB . The following table gives some typical values, which are mainly suitable for desktop systems equipped with an appropriate amount of memory to provide readers with a little perceptual knowledge. A constant constraint is that it cannot be less than 128 KiB and cannot be more than 64 MiB . But it should be noted that the upper bound can only be reached when the amount of memory is really large.

The user layer can read and modify this setting through the file `/proc/sys/vm/min_free_kbytes` .

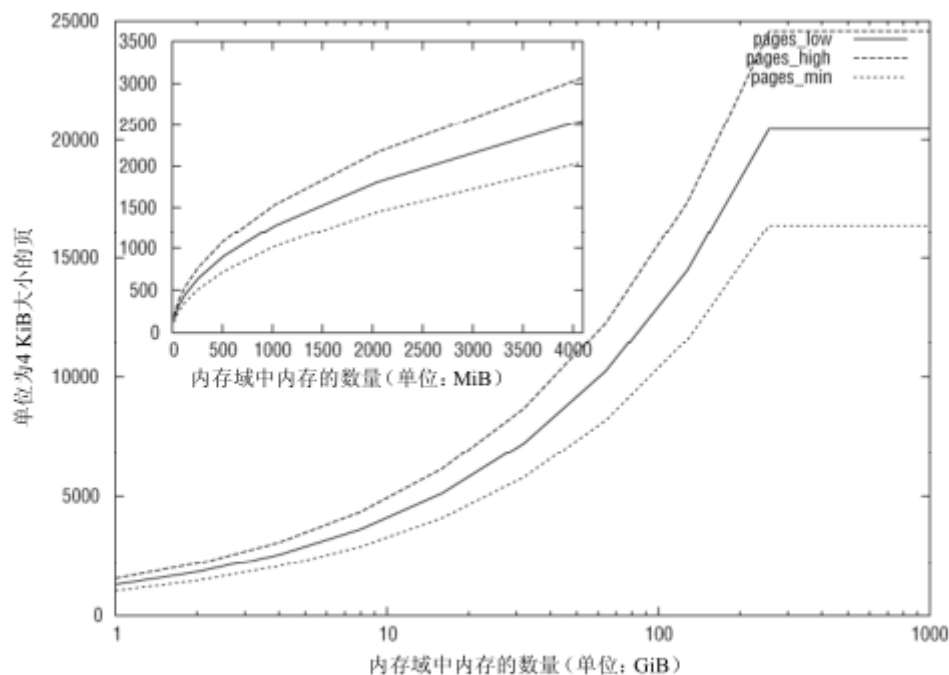


图3-4 内存域水印和为关键性分配保留的内存空间的最小值，与计算机主内存大小之间的关系（pages_min即按页计算的min_free_kbytes）

表3-1 主内存大小与可用于关键性分配的内存空间最小值之间的关系

主内存大小	保留内存大小
16 MiB	512 KiB
32 MiB	724 KiB
64 MiB	1024 KiB
128 MiB	1448 KiB
256 MiB	2048 KiB
512 MiB	2896 KiB
1024 MiB	4096 KiB
2048 MiB	5792 KiB
4096 MiB	8192 KiB
8192 MiB	11584 KiB
16384 MiB	16384 KiB

Watermark calculation - code flow (init_per_zone_wmark_min)

The filling of the various fields of the struct zone structure is handled by init_per_zone_wmark_min (mm/page_alloc.c). This function is called by the kernel during startup and does not need to be explicitly called .

init_per_zone_wmark_min will further call setup_per_zone_wmarks and setup_per_zone_lowmem_reserve .

- setup_per_zone_wmarks is mainly responsible for filling the zone->watermark field
- setup_per_zone_lowmem_reserve is mainly responsible for filling zone-> lowmem_reserve . All nodes of the kernel iteration system, calculate the minimum reserved memory for each memory domain of each node. The specific algorithm is to divide the total number of page frames in the memory domain by sysctl_lowmem_reserve_ratio [zone] . The default setting of the divisor is 256 for the low -end memory domain and 32 for the high-end memory domain

2.3 Initialize memory management

The key to initializing the memory management system is the initialization of the pg_data_t data structure (and its subordinate structures) .

2.3.1 Background knowledge

NODE_DATA

All platforms have implemented the architecture-specific `NODE_DATA` macro, which is used to query the `pgdata_t` instance related to a NUMA node by the node number .

The definition of the `NODE_DATA` macro is as follows :

Header file : `include/linux/mmzone.h`

```
#ifndef CONFIG_NEED_MULTIPLE_NODES

extern struct pglist_data contig_page_data ;
#define NODE_DATA(nid) (&contig_page_data)
#define NODE_MEM_MAP(nid) mem_map

#else /* CONFIG_NEED_MULTIPLE_NODES */

#include <asm/mmzone.h>

#endif /* !CONFIG_NEED_MULTIPLE_NODES */
```

The implementation of `NODE_DATA` is divided into two cases: single node and multiple nodes . If `CONFIG_NEED_MULTIPLE_NODES` is defined , it means that there are multiple nodes in the system , otherwise there is only one node in the system .

If the system has multiple nodes , the `NODE_DATA` implementation depends on the specific architecture , in `<asm / mmzone.h>` definition . Because most systems have only one node , therefore we will not focus on the details of multiple nodes .

If only one node , the `NODE_DATA` defined as `#define NODE_DATA (nid) (& contig_page_data)` , this time `nid` typically 0. directly acquired `contig_page_data` address .

`contig_page_data` is in `mm / bootmem.c` defined a `pg_data_t` example , manage all system memory .

```
#ifndef CONFIG_NEED_MULTIPLE_NODES
struct pglist_data __refdata contig_page_data = {
    . bdata = & bootmem_node_data [ 0 ]
};
EXPORT_SYMBOL ( contig_page_data );
#endif
```

The layout of the kernel code in memory

Before discussing each specific memory initialization operation, we need to figure out the specific layout of the memory at this time after the boot loader copies the kernel to the memory, and the assembler part of the initialization routine has been executed. I focus on the default situation where the kernel is loaded into a fixed location in physical memory, which is determined at compile time.

If the crash dump mechanism is enabled, the initial location of the kernel binary code in physical memory can also be configured. In addition, some embedded systems also require this capability. The configuration option `PHYSICAL_START` is used to determine the location of the kernel in the memory, which will be affected by the physical alignment set by the configuration option `PHYSICAL_ALIGN`.

In addition, the kernel can be built into a relocatable binary program, in which case the physical start address given at compile time is completely ignored. The boot loader can determine where to put the kernel. We have encountered more of this situation, and a detailed explanation will be introduced in the article "Linux Kernel Boot Process".

The following figure shows the layout of the minimum few megabytes of physical memory, and the resident status of each part of the kernel image.

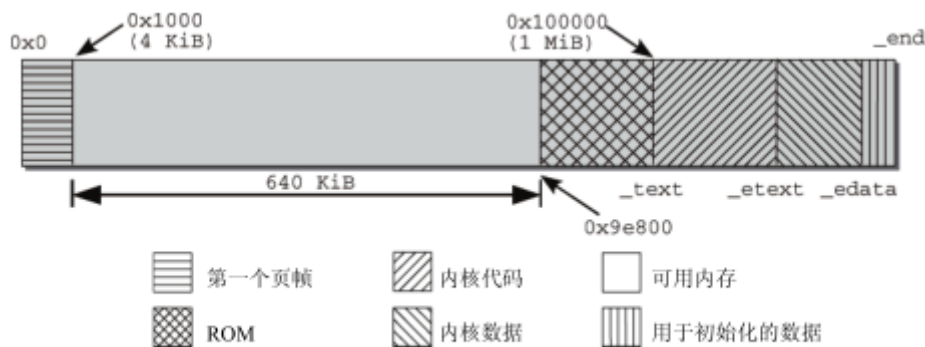


图3-11 Linux内核在内存中的布局

The figure shows the first few megabytes of physical memory, the specific length depends on the length of the kernel binary file. The first 4 KiB is the first page frame, which is generally ignored because it is usually reserved for BIOS use. The next 640 KiB is usable in principle, but it is not used for kernel loading either. The reason is that the area immediately after this area is reserved by the system for mapping various ROMs (usually system BIOS and graphics card ROM). It is impossible to write data to the mapped ROM area. But the kernel will always be loaded into a contiguous memory area. If you want to load the kernel image from 4 KiB as the starting position, the kernel must be less than 640 KiB.

To solve these problems, the kernel generally uses 0x100000 as the starting address. This corresponds to the beginning of the second megabyte in memory. From here, there is enough contiguous memory area to hold the entire kernel. (In the ARM architecture, the kernel generally uses 0x8000<32KB> as the starting address, 0 – 32KB is reserved, of which 16KB-32KB is used to store the initial kernel page table swapper_pg_dir)

The memory occupied by the kernel is divided into several segments, the boundaries of which are stored in variables.

- `_text` and `_etext` are the start and end addresses of the code segment, including the compiled kernel code.
- `_etext` and `_edata` are the start and end addresses of the data segment, which saves most of the kernel variables.
- `_end` is the end address of the initialization data segment (for example, the BSS segment containing all static global variables initialized to 0) is saved in the last segment, from `_edata` to `_end`. After the kernel is initialized, most of the data can be deleted from the memory, leaving more space for the application. This memory area is divided into smaller sub-intervals to control which can be deleted and which cannot be deleted, but this is of little significance to our current discussion.

Although the variables used to delimit the segment boundaries are defined in the kernel source code (`arch/$ (SRCARCH)/kernel/ setup.c`), they have not been assigned at this time. This is because it is unlikely. How can the compiler know how big the kernel is in the end at compile time? Only after the target file is linked, can the exact value be known, and then packaged into a binary file. This operation is performed by `Arch / $ (SRCARCH) / Kernel / vmlinux.ld.S` control (for ARM is, the file is `Arch / ARM / Kernel / the vmlinux .ld.S`) , which also defined the kernel memory layout.

The exact value varies depending on the kernel configuration, because the code segment and data segment length of each configuration is different, depending on which parts of the kernel are enabled and disabled. Only the starting address (`_text`) is always the same.

Every time the kernel is compiled, a file `System.map` is generated and saved in the source code directory. In addition to the addresses of all other (global) variables, kernel-defined functions and routines, the file also includes the values of the constants given below.

```
wolfgang@meitner> cat System.map
```

```
...
c0100000 A _text
...
c0381ecd A _etext
...
c04704e0 A _edata
...
c04c3f44 A _end
...
```

All the above address values are offset by `0xC0000000` , which is the starting address of the kernel segment when the standard 3:1 division between the user and the kernel address space is adopted . This address is a virtual address, because when the physical memory is mapped to the virtual address space, a linear mapping method starting from this address is used. Subtract `0xC0000000` to obtain the corresponding physical address.

`/proc/iomem` also provides some information about the segments divided by physical memory.

```
wolfgang@meitner> cat /proc/iomem
```

```
00000000-0009e7ff: System RAM
0009e800-0009ffff: reserved
000a0000-000bffff: Video RAM area
000c0000-000c7fff: Video ROM
000f0000-000fffff: System ROM
00100000-17ceffff: System RAM
    00100000-00381ecc: Kernel code
    00381ecd-004704df: Kernel data
...
```

The kernel image starts after the first megabyte (`0x00100000`) . The length of the code segment is approximately 2.5 MiB , and the data segment is approximately 0.9 MiB .

Similar information can also be obtained on AMD64 systems. Here the kernel starts 2 MiB after the first page frame , and the physical memory is mapped into the virtual address space starting from `0xffffffff80000000` . The related items in `System.map` are as follows:

```
wolfgang@meitner> cat System.map
```

```
ffffffff80200000 A _text
```

```
...
ffffff8041fc6f A _etext
```

```
...
ffffff8056c060 A _edata
```

```
...
ffffff8077548c A _end
```

At runtime, you can also get information about the kernel from `/proc/iomem` :

```
root@meitner # cat/proc/iomem
```

```
...
00100000-cff7ffff: System RAM
    00200000-0041fc6e: Kernel code
    0041fc6f-0056c05f: Kernel data
    006b6000-0077548b: Kernel bss
...
```

memblock

As we mentioned at the beginning of Section 2.3 , the key to the initialization of the memory management subsystem is the initialization of the `pg_data_t` data structure .

In 2.6.19 must come before the kernel version depending on the architecture itself establish the required data structures . With the evolution of the kernel , the latest version of the kernel , most of the initial work will be independent of , and architecture It doesn't matter anymore .

Architecture-related code only needs to add all active memory areas to a pool, and the general code of the kernel generates the main data structure based on this .

This pool is the `memblock`, and the basic element of the pool is an active memory area , called a memory block .

A Memory Block represents a physical memory , the "block" is a concept which can physically , or may be a logical concept . Concept called physical we like on the board 2 blocks of physical memory , the starting address is not the same , The size is 1GB , this is 2 memory blocks ; the so-called logical concept is that we logically divide a piece of physical memory into multiple blocks, and each block has a starting address and a length .

Or logical physical regardless , there is no difference for us . In essence , as long as a given starting physical address and length , for a given a memory block.

Note that , a memory block inside can not have empty .

Main data structure

struct `memblock_region` : We use this structure to abstract a memory block.

Header file : `include/linux/memblock.h`

```
struct memblock_region {
    phys_addr_t base ;
    phys_addr_t size ;
    unsigned long flags ;
#ifdef CONFIG_HAVE_MEMBLOCK_NODE_MAP
    int nid ;
#endif
};
```

struct memblock_type : When multiple memory blocks are combined , they are called memblock_type.

Header file : include/linux/memblock.h

```
struct memblock_type {
    unsigned long cnt ; /* number of regions */
    unsigned long max ; /* size of the allocated array */
    phys_addr_t total_size ; /* size of all regions */
    struct memblock_region * regions ;
};
```

struct memblock : There may be multiple memblock_types in the system. Together , they are called memblock.

The kernel code specifies 3 types of memblock_types, namely : memory , reserved , physmem

Header file : include/linux/memblock.h

```
struct memblock {
    bool bottom_up ; /* is bottom up direction? */
    phys_addr_t current_limit ;
    struct memblock_type memory ;
    struct memblock_type reserved ;
#ifdef CONFIG_HAVE_MEMBLOCK_PHYS_MAP
    struct memblock_type physmem ;
#endif
};
```

- The memblock_type of memory is used to add memory blocks . Whenever there is a new memory block , it will be added to it.
- Reserved This memblock_type for memory allocation , before the memory management subsystem initialization is complete , if the need to apply the kernel memory , it will be this way . We will discuss this in detail in the "bootstrap distributor" section Kind of situation .

memblock (mm/memblock.c) : It is an instance of statically defined struct memblock . This example is equivalent to setting up the shelf of the pool , but the pool is empty . During the kernel startup process , the setup_arch function will detect physical memory available to the system , and there is added to the pool memory block.

Main API

Header file : include/linux/memblock.h

Implementation file : mm/memblock.c

```
int __init memblock memblock_add_range ( struct memblock_type * type ,
    phys_addr_t base , phys_addr_t size ,
    int nid , unsigned long flags )
```

- Add a block to the corresponding memblock_type .
- It is the basis for adding class APIs below , and the following APIs will eventually call it to complete the actual work .
- nid refers to the NUMA ID of the node to which the block belongs . The UMA system is set to 0.
- Note , block and block may be between Coverlap ; Also when registering two adjacent memory area, the API will merge them into one .

```
int __init_memblock memblock_add ( phys_addr_t base , phys_addr_t size )
```

- to **memory** this `memblock_type` which added a block

```
int __init_memblock memblock_remove ( phys_addr_t base , phys_addr_t size )
```

- Remove a block from the `memblock_type` of **memory**

```
int __init_memblock memblock_reserve ( phys_addr_t base , phys_addr_t size )
```

- to **reserved** this `memblock_type` which added a block

```
phys_addr_t __init memblock_alloc ( phys_addr_t size , phys_addr_t align )
```

- allocated block of memory , the size of size, and to add memory allocated to this **reserved** this `memblock_type` inside

```
int __init_memblock memblock_free ( phys_addr_t base , phys_addr_t size )
```

- release memory , and from the **reserved** this `memblock_type` removed inside a block

2.3.2 The macro process of initialization

The initialization of the `pg_data_t` data structure starts from the global startup routine `start_kernel` , which is executed after the kernel is loaded and the various subsystems are activated. Since memory management is a very important part of the kernel , the initialization of memory management will be performed immediately after the memory is detected in the architecture-specific code and the allocation of memory in the system is determined. At this point, an instance of `pgdata_t` already exists , which is used to save information such as the amount of memory in the node and the allocation of memory among various memory domains . The initialization process is mainly to assign values to the corresponding fields of the instance.

The general process is as follows :

start_kernel (`init/main.c`)

- **setup_arch :**

`setup_arch` is an architecture-specific setup function, and its main tasks are

- ✓ Check the available physical memory of the system.
- ✓ `low_end_mem` the cutoff value of low-end/high-end memory
- ✓ Responsible for initializing the bootstrap allocator.
- ✓ initialization data structure `pg_data_t`

- **setup_per_cpu_areas**

On the SMP system, `setup_per_cpu_areas` initializes the static per-cpu variables defined in the source code (using the `per_cpu` macro) . This variable has an independent copy for each CPU in the system. Such variables are stored in a separate section of the kernel binary image. The purpose of `setup_per_cpu_areas` is to create a copy of these data for each CPU of the system.

This function is a no-op on non-SMP systems.

This article does not intend to discuss the details of this function

- **build_all_zonelists**

Create a list of spare memory domains for the current node, so that when the current node has no remaining memory, memory is allocated from the spare list

- **mm_init**

- **mem_init**

mem_init is another architecture-specific function used to deactivate the bootstrap allocator and migrate to the actual memory management function, which will be discussed later .

- **kmem_cache_init**

kmem_cache_init is used to initialize the slab allocator. The smallest unit allocated by the buddy allocator is a page. The slab works on the buddy and is used to allocate memory space less than one page.

We will discuss this function in section 2.5.

➤ **setup_per_cpu_pageset**

Used to initialize the pageset member of the zone structure , this member is related to the hot and cold page mechanism, this article will not discuss it temporarily

➤ **rest_init**

If you look at the kernel code, you will find that rest_init is the last function called by start_kernel. rest_init will eventually execute the init process in the user space, and the entire kernel will be initialized at this point.

This article does not intend to discuss the boot process of the kernel, but focuses on a function called by rest_init:

rest_init -> kernel_init -> free_initmem .

The kernel will call the **free_initmem** function immediately before executing the init process to release the initialization data.

2.3.3 setup_arch

After the kernel has been loaded into the memory and the initialization part of the assembler has been executed, what architecture-specific steps must the kernel perform to complete the initialization of the memory management system?

The macro process is as follows:

- **setup_arch**
 - Check the physical memory available to the system (setup_machine_fdt/setup_machine_tags, and parse_early_param)
 - Determine the cutoff value of low-end / high-end memory (sanity_check_meminfo)
 - Initialize the bootstrap allocator (arm_memblock_init)
 - paging_init
- **bootmem_init**
 - **zone_sizes_init**
 - **free_area_init_node** (initialize pg_data_t data structure)

paging_init has an important function is to establish a kernel logical address space mapped page tables, this part will be in the third chapter discussed in detail the kernel virtual address space, this chapter being only concerned free_area_init_node function.

Check the physical memory available to the system

There are two ways to check the physical memory available to the system:

The first is to find the memory node in dtb to determine the physical memory available to the system. (On a machine that does not use dtb, the physical memory is detected in uboot, and then uboot will be passed to the kernel through the atags mechanism).

The second is that the user can specify the physical memory available to the system through mem=size@start in bootargs.

If both of the above two methods exist, the kernel will overwrite the first method with the value of the second method, which means that the second method is preferred.

The code call flow of the first method is roughly as follows:

- setup_arch
 - setup_machine_fdt
 - early_init_dt_scan_nodes
 - early_init_dt_scan_memory
 - Parse the memory node (refer to the article " device tree " for the grammatical rules of the memory node) to obtain the starting physical address and length of the memory
 - early_init_dt_add_memory_arch
 - memblock_add

The code call flow of the second method is roughly as follows:

- setup_arch
 - parse_early_param
 - By parse_cmdline mechanism (see "kernel code base element" article) , the final call early_mem (Arch / ARM / Kernel / setup.c) function , the function parses uboot transmission over bootargs in mem = size @ start parameters , acquired memory of Starting physical address and length
 - arm_add_memory
 - Do some alignment checks
 - memblock_add

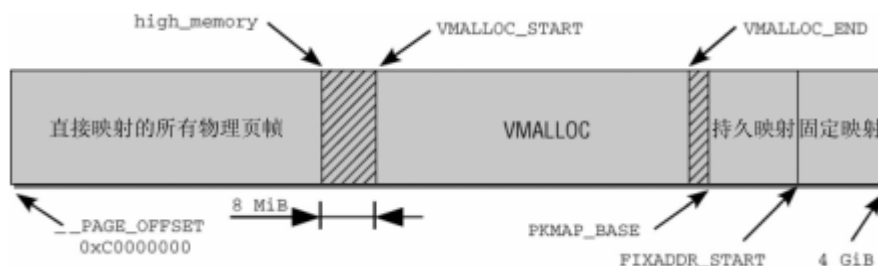
The above two methods will eventually call the memblock_add API. We introduced memblock and its related APIs in the background knowledge of section 2.3.1 .

Determine the cutoff value of low-end / high-end memory

Remember what low-end / high-end memory is ? We introduced them in section 1.1.2 "Kernel Virtual Address Space" . In simple terms , we divide physical memory into 2 parts : one part is used for kernel logical mapping (one one Mapping) , which is called low-end memory ; the other part is used for dynamic mapping by the kernel through page tables , which is called high-end memory .

So how to determine the cutoff value of low-end / high-end ?

Let's take a look at a map , this map represents the division in the kernel virtual address space , already in 1.1.2 "kernel virtual address space" section introduced before , here once again repeat posted :



From PAGE_OFFSET - high_memory: represents the virtual address space that the kernel can map one by one .

From high_memory - VMALLOC_START: is a segment of 8M partition .

From VMALLOC_START - VMMLLOC_END : Represents the virtual address space that the kernel can dynamically map .

From VMALLOC_END - PKMAP_BASE: was a partition , typically 2 th page.

In the ARM architecture , the persistent mapping area is placed before the PAGE_OFFSET , and the VMALLOC area is followed by the fixed mapping area . Chapter 3 discusses the details .

From the above picture , high_memory is the boundary value of low-end / high-end memory , so what is high_memory ? Who determines the value of high_memory ?

The answer is the sanity_check_meminfo function , and its calling process is start_kernel -> setup_arch -> sanity_check_meminfo .

Let's take a look at the code of sanity_check_meminfo . For more clarity , we will introduce this function in sections .

Implementation file : arch/arm/mm/mmu.c

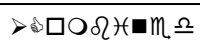
```
void __init sanity_check_meminfo ( void )
{
    phys_addr_t memblock_limit = 0 ;
    int highmem = 0 ;
    phys_addr_t vmalloc_limit = __pa ( vmalloc_min - 1 ) + 1 ;
    struct memblock_region * reg ;
    bool should_use_highmem = false ;
```

- define some initial variables . Particular attention vmalloc_limit this value , the text of the main logic related to the value related to . It depends on vmalloc_min, we first introduce vmalloc_min, understand it helps to understand the logic behind .

vmalloc_min

```
//arch/arm/mm/mmu.c
static void * __initdata vmalloc_min =
    ( void * ) ( VMALLOC_END - ( 240 << 20 ) - VMALLOC_OFFSET );

//arch/arm/include/asm/pgtable.h
/*
 * Just any arbitrary offset to the start of the vmalloc VM area: the
 * current 8MB value just means that there will be a 8MB "hole" after the
 * physical memory until the kernel virtual memory starts. That means that
 * any out-of-bounds memory accesses will hopefully be caught.
 * The vmalloc() routines leaves a hole of 4kB between each vmallocated
 * area for the same reason. 🤔
 */
#define VMALLOC_OFFSET (8*1024*1024)
#define VMALLOC_START (((unsigned long)high_memory + VMALLOC_OFFSET) & ~
    (VMALLOC_OFFSET-1))
#define VMALLOC_END 0xff800000UL
```

-  with the picture of the kernel virtual address space division , VMALLOC_END is defined as a fixed value . From VMALLOC_END - 4G, there is still 8M of space left , which 8M is used for the fixed mapping area .
- VMALLOC_START value depends on high_memory, high_memory low-end / divide the value of high-end memory , the following will explain how to determine when its value . From here we can see , the kernel vmalloc size of the area with high_memory related .

- `vmalloc_min` value from `VMALLOC_END` beginning , subtracting 240M space , and then subtracting `VMALLOC_OFFSET` (8M of divided sections). It can be seen here , the default kernel to `vmalloc` region of virtual address space reserved is 240M.

`vmalloc_min` value what is it ? Imagine , if we are from `0xc0000000` start at one mapping of physical memory , if the physical memory is relatively large , the upper bound of a map that will exceed `vmalloc_min`, this time will be diverted `vmalloc` virtual area Address space . Therefore , if $(0xc0000000 + \text{physical memory capacity}) > \text{vmalloc_min}$, then `high_memory` is equal to `vmalloc_min`; if $(0xc0000000 + \text{physical memory capacity}) < \text{vmalloc_min}$, then `high_memory` is equal to $(0xc0000000 + \text{physical memory capacity})$.

By default reserved 240M to `vmalloc` region is calculated , the maximum one mapping area value (`VMALLOC_END` - (`<< 20` is 240) - `VMALLOC_OFFSET`) . = 768M that is, if more than the physical memory 768M, the kernel can be one mapping The maximum area is only 768M, and the extra part must be enabled with `CONFIG_HIGHMEM` before it can be used . If the system does not enable `HIGHMEM` support in this case , the excess memory core will not be available . In this case, the content will remind the user to enable `CONFIG_HIGHMEM` through a print message .

Although the default address space size of `vmalloc` is 240M, we can modify it through `bootargs` . The parameter rule is `vmalloc = size`.The unit of size is bytes .

Responsible for parsing function definition in `arch / arm / mm / mmu.c` in .

```
/*
 * vmalloc=size forces the vmalloc area to be exactly'size'
 * bytes. This can be used to increase (or decrease) the vmalloc
 * area - the default is 240m.
 */
static int __init early_vmalloc ( char * arg )
{
    unsigned long vmalloc_reserve = memparse ( arg , NULL );

    if ( vmalloc_reserve < SZ_16M ) {
        vmalloc_reserve = SZ_16M ;
        pr_warn ( "vmalloc area too small, limiting to %luMB\n" ,
            vmalloc_reserve >> 20 );
    }

    if ( vmalloc_reserve > VMALLOC_END - ( PAGE_OFFSET + SZ_32M ) ) {
        vmalloc_reserve = VMALLOC_END - ( PAGE_OFFSET + SZ_32M );
        pr_warn ( "vmalloc area is too big, limiting to %luMB\n" ,
            vmalloc_reserve >> 20 );
    }

    vmalloc_min = ( void *) ( VMALLOC_END - vmalloc_reserve );
    return 0 ;
}
early_param ( "vmalloc" , early_vmalloc );
```

Okay , let's look at `sanity_check_meminfo`

```
for_each_memblock ( memory , reg ) {
    phys_addr_t block_start = reg -> base ;
```

```

phys_addr_t block_end = reg -> base + reg -> size ;
phys_addr_t size_limit = reg -> size ;

if ( reg -> base >= vmalloc_limit )
    highmem = 1 ;
else
    size_limit = vmalloc_limit - reg -> base ;

if (! IS_ENABLED ( CONFIG_HIGHMEM ) || cache_is_vipt_aliasing ()) {

    if ( highmem ) {
        pr_notice ( "Ignoring RAM at %pa-%pa (!CONFIG_HIGHMEM)\n" ,
                    & block_start , & block_end );
        memblock_remove ( reg -> base , reg -> size );
        should_use_highmem = true ;
        continue ;
    }

    if ( reg -> size > size_limit ) {
        phys_addr_t overlap_size = reg -> size - size_limit ;

        pr_notice ( "Truncating RAM at %pa-%pa to -%pa" ,
                    & block_start , & block_end , & vmalloc_limit );
        memblock_remove ( vmalloc_limit , overlap_size );
        block_end = vmalloc_limit ;
        should_use_highmem = true ;
    }
}

if (! highmem ) {
    if ( block_end > arm_lowmem_limit ) {
        if ( reg -> size > size_limit )
            arm_lowmem_limit = vmalloc_limit ;
        else
            arm_lowmem_limit = block_end ;
    }
}

/*
 * Find the first non-pmd-aligned page, and point
 * memblock_limit at it. This relies on rounding the
 * limit down to be pmd-aligned, which happens at the
 * end of this function.
 *
 * With this algorithm, the start or end of almost any

```

```

    * bank can be non-pmd-aligned. The only exception is
    * that the start of the bank 0 must be section-
    * aligned, since otherwise memory would need to be
    * allocated when mapping the start of bank 0, which
    * occurs before any free memory is mapped.
    */
    if ( ! memblock_limit ) {
        if ( ! IS_ALIGNED ( block_start , PMD_SIZE ))
            memblock_limit = block_start ;
        else if ( ! IS_ALIGNED ( block_end , PMD_SIZE ))
            memblock_limit = arm_lowmem_limit ;
    }

}

if ( should_use_highmem )
    pr_notice ( "Consider using a HIGHMEM enabled kernel.\n" );

high_memory = __va ( arm_lowmem_limit - 1 ) + 1 ;

```

- This code is the core logic , the main purpose is the last word , determined high_memory value in the end is how much . Once determined , the core low-end / high-end memory division to determine .
- high_memory specific value depends on arm_lowmem_limit, which is calculated by the previous code , calculating principle <vmalloc_min> has been introduced over , here not analyzed the details , read the code itself .

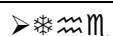
Continue to look at sanity_check_meminfo

```

/*
 * Round the memblock limit down to a pmd size. This
 * helps to ensure that we will allocate memory from the
 * last full pmd, which should be mapped.
 */
if ( memblock_limit )
    memblock_limit = round_down ( memblock_limit , PMD_SIZE );
if ( ! memblock_limit )
    memblock_limit = arm_lowmem_limit ;

memblock_set_current_limit ( memblock_limit );
}

```

-  main purpose is to call memblock_set_current_limit . What is its function ? high_memory is the boundary value of the low-end / high-end address in the kernel virtual address space , which is a virtual address ; and the memblock_limit here is the low-end of the physical address space / end address boundary value , a physical address , memblock module stores the value .

After the code runs here , the memblock module already stores all the available physical memory in the system and the boundary value of the low-end / high-end address of the physical memory . With these two information , the kernel general code can initialize the representative in the `pg_data_t` structure The `struct page *node_mem_map` of all memory page frames and the `struct zone node_zones[MAX_NR_ZONES]` representing each memory `zone` .

Initialize the bootstrap allocator

During the startup process, although the memory management has not yet been initialized, the kernel still needs to allocate memory to create various data structures. The bootstrap allocator is used to allocate memory early in the startup phase.

Obviously, the demand for this dispenser is focused on simplicity rather than performance and versatility. After the memory management subsystem is initialized, the bootstrap allocator is no longer used, and instead uses the API interfaces provided by the memory management subsystem because these APIs are more efficient.

There are two ways to implement the bootstrap allocator . The first method is called bootmem, which is described in detail in "In-depth Linux Kernel Architecture Section 3.4.3 " . I will not go into details here .

If the kernel at compile time can make a `CONFIG_NO_BOOTMEM` , is not used bootmem, but will help " 2.3.1 Background" which introduced memblock.

The memblock does not need to be initialized , and the kernel code statically defines an instance of struct memblock with the same name .

memblock there are two memblock_type, one Memory , and the other is Reserved .

In the stage of ``Detecting the available physical memory of the system" , all available physical memory blocks will be added to the memory .

If you want to allocate kernel memory initialization process earlier , it will be from memblock acquired inside a memory block , and add to the memory block reserved inside , indicating that the memory block has been dispensed , can not be released before the person using it .

In addition to the memory block has been allocated to the need to add reserved inside , and some memory space is not available for allocation , for example, kernel code memory space occupied by itself , the kernel page table initial `swapper_pg_dir` space occupied by , and the like .

Kernel initial page table `swapper_pg_dir` main purpose is before the actual kernel page table is created , the kernel startup code (virtual address) is mapped to physical memory . In the ARM architecture in , `swapper_pg_dir` itself is located 16KB-32KB position , 32KB (0x8000) Start to store the kernel code . `swapper_pg_dir` only maps 1M address space , which is enough , the kernel initialization code will not exceed 1M.

For those who can not be used for memory allocation , `arm_memblock_init` function reserved for these memory blocks , add them to the **reserved** inside .

arm_memblock_init

Its implementation file is `arch/arm/mm/init.c`, and the calling process is `setup_arch-> arm_memblock_init` . This function mainly reserves the following memory blocks :

`arm_memblock_init`

- `memblock_reserve(__pa(_stext), _end - _stext)` : Reserve the physical memory occupied by the kernel itself (" 2.3.1 Background Knowledge - Kernel Code Layout in Memory" section introduces the meaning of `_text/_stext, _end`)
- `memblock_reserve(phys_initrd_start, phys_initrd_size)` : Reserve the memory space occupied by Ramdisk (initrd) (<http://blog.csdn.net/ruixj/article/details/3772752>) . The initrd is loaded into the memory by the bootloader . At this time, the bootloader will pass the start address and end address to the kernel , the global `initrd_start` in the kernel And `initrd_end` point to the start address and end address of `initrd` , respectively .
(<http://blog.csdn.net/ruixj/article/details/3772752>)
- `arm_mm_memblock_reserve(void)` : Reserve the memory space occupied by `swapper_pg_dir`
- `if (mdesc->reserve) mdesc->reserve()`: If `reserve` is defined in `machine_desc` , reserve this part of the space
- `early_init_fdt_reserve_self()` : Reserve the memory space occupied by `dtb` itself
- `early_init_fdt_scan_reserved_mem()` : If the space that needs to be reserved is defined in `dtb` , then this part of the space is reserved
- `dma_contiguous_reserve(arm_dma_limit)` : /* reserve memory for DMA contiguous allocations */
- `memblock_dump_all` : If `memblock_debug=1` is enabled , the contents of memory and reserved in `memblock` will be printed . From here, we can see the total amount of available physical memory and the reserved physical memory status .

By default, `memblock_debug=0`, we can add `memblock=debug` to `bootargs` to make `memblock_debug=1`, so that during the kernel startup process , `memblock_dump_all` will print the following :

```

[ 0.000000] MEMBLOCK configuration:
[ 0.000000] memory size = 0x1fe00000 reserved size = 0x253c844
[ 0.000000] memory.cnt = 0x1
[ 0.000000] memory[0x0] = [0x0000000800000000-0x00000009fdffff], 0x1fe00000 bytes flags: 0x0
[ 0.000000] reserved.cnt = 0x4
[ 0.000000] reserved[0x0] = [0x0000000800040000-0x000000080007ffff], 0x4000 bytes flags: 0x0
[ 0.000000] reserved[0x1] = [0x0000000800082400-0x000000080d31a83], 0xd29844 bytes flags: 0x0
[ 0.000000] reserved[0x2] = [0x00000008fffe0000-0x00000008ffffcfff], 0xf000 bytes flags: 0x0
[ 0.000000] reserved[0x3] = [0x00000009e0000000-0x00000009f7ffff], 0x1800000 bytes flags: 0x0

```

Screenshot of the printed information on the beaglebone black board

Memory allocation API

To start the process of the kernel by `memblock` to allocate memory , you can use the following APIs:

Header file : `include/linux/memblock.h`

```

phys_addr_t __init memblock_alloc ( phys_addr_t size , phys_addr_t align )
...

```

```

int memblock_free ( phys_addr_t base , phys_addr_t size );

```

API implementation details read the code by yourself , so I won't talk about it here .

Initialize the `pg_data_t` data structure

First recall the composition of the `pg_data_t` data structure . We only consider the simple case , that is, there is only one node in the system . Then there is only one `pg_data_t` instance in the system. There are two important data structures under this instance that we currently need to pay attention to :

`pg_data_t`

- `struct page *node_mem_map`
- `struct zone node_zones[MAX_NR_ZONES]`

`node_mem_map` is a pointer to an array of `page` instances, used to describe all physical memory pages of the node. It contains the pages of all memory domains in the node . In the stage of "Checking the available physical memory of the system", we have added the available physical memory of the system to `memblock->memory` , so

the general code of the kernel can get the available physical memory from the memblock , and then Create a struct page data structure for each page frame , and then initialize the node_mem_map pointer.

node_zones represent all domains under the node memory , the memory is divided into domains are generally the ZONE_DMA, the ZONE_NORMAL, the ZONE_HIGHMEM these . 3 species. In the ARM architecture, there are generally no special requirements for the DMA area, and the memory in the NORMAL area can also be used for DMA operations . Many ARM architectures do not enable CONFIG_ZONE_DMA , so the most important thing here is to determine the boundary value of NORMAL/HIGHMEM . Fortunately, in "Determining the boundary value of low-end / high-end memory", we have determined the boundary value (memblock_limit) and stored it in memblock . At this time, the system code only needs to be obtained from memblock .

Let's take a look at the specific code below :

setup_arch-> paging_init -> bootmem_init

The implementation file of **bootmem_init** is arch/arm/mm/init.c

- find_limits(&min, &max_low, &max_high);
- zone_sizes_init(min, max_low, max_high);

The main purpose of find_limits is to determine 3 values , min = PFN_UP(memblock_start_of_DRAM()) ; max_high = PFN_DOWN(memblock_end_of_DRAM()) ; and max_low is the boundary value of NORMAL/HIGHMEM , max_low = PFN_DOWN(memblock_get_current_limit()) .

After obtaining these 3 values , **zone_sizes_init** starts to use these 3 values to initialize the relevant data structure of pg_data_t .

```
//arch/arm/mm/init.c
static void __init zone_sizes_init ( unsigned long min , unsigned long max_low
,
    unsigned long max_high )
{
    unsigned long zone_size [ MAX_NR_ZONES ], zhole_size [ MAX_NR_ZONES ];
    struct memblock_region * reg ;

    /*
     * initialise the zones.
     */
    memset ( zone_size , 0 , sizeof ( zone_size ) );

    /*
     * The memory size has already been determined. If we need
     * to do anything fancy with the allocation of this memory
     * to the zones, now is the time to do it.
     */
    zone_size [ 0 ] = max_low - min ;
#ifdef CONFIG_HIGHMEM
    zone_size [ ZONE_HIGHMEM ] = max_high - max_low ;
#endif
}
```

```

/*
 * Calculate the size of the holes.
 * holes = node_size - sum(bank_sizes)
 */
memcpy ( zhole_size , zone_size , sizeof ( zhole_size ) );
for_each_memblock ( memory , reg ) {
    unsigned long start = memblock_region_memory_base_pfn ( reg );
    unsigned long end = memblock_region_memory_end_pfn ( reg );

    if ( start < max_low ) {
        unsigned long low_end = min ( end , max_low );
        zhole_size [ 0 ] -= low_end - start ;
    }
#ifdef CONFIG_HIGHMEM
    if ( end > max_low ) {
        unsigned long high_start = max ( start , max_low );
        zhole_size [ ZONE_HIGHMEM ] -= end - high_start ;
    }
#endif
}

#ifdef CONFIG_ZONE_DMA
/*
 * Adjust the sizes according to any special requirements for
 * this machine type.
 */
if ( arm_dma_zone_size )
    arm_adjust_dma_zone ( zone_size , zhole_size ,
        arm_dma_zone_size >> PAGE_SHIFT );
#endif

free_area_init_node ( 0 , zone_size , min , zhole_size );
}

```

All the previous actions are to determine the number of page frames in each memory domain , and the last sentence calls `free_area_init_node`

free_area_init_node

```

//mm/page_alloc.c
void __paginginit free_area_init_node ( int nid , unsigned long * zones_size ,
    unsigned long node_start_pfn , unsigned long * zholes_size )
{
    pg_data_t * pgdat = NODE_DATA ( nid );
    unsigned long start_pfn = 0 ;
    unsigned long end_pfn = 0 ;

```



```

/* pg_data_t should be reset to zero when it's allocated */
WARN_ON ( pgdat -> nr_zones || pgdat -> classzone_idx );

reset_deferred_meminit ( pgdat );
pgdat -> node_id = nid ;
pgdat -> node_start_pfn = node_start_pfn ;
#ifdef CONFIG_HAVE_MEMBLOCK_NODE_MAP
get_pfn_range_for_nid ( nid , & start_pfn , & end_pfn );
pr_info ( "Initmem setup node %d [mem %#018Lx-%#018Lx]\n" , nid ,
        ( u64 ) start_pfn << PAGE_SHIFT ,
        end_pfn ? (( u64 ) end_pfn << PAGE_SHIFT ) - 1 : 0 );
#endif
calculate_node_totalpages ( pgdat , start_pfn , end_pfn ,
                           zones_size , zholes_size );

alloc_node_mem_map ( pgdat );
#ifdef CONFIG_FLAT_NODE_MEM_MAP
printk ( KERN_DEBUG "free_area_init_node: node %d, pgdat %08lx,
node_mem_map %08lx\n" ,
        nid , ( unsigned long ) pgdat ,
        ( unsigned long ) pgdat -> node_mem_map );
#endif

free_area_init_core ( pgdat );
}

```

- initialization pgdat-> node_id , pgdat-> node_start_pfn
- calculate_node_totalpages
 - initialize pgdat-> node_spanned_pages , pgdat-> node_present_pages
- alloc_node_mem_map
 - each page frame is allocated struct page data structure , and then initializes pgdat-> node_mem_map
- free_area_init_core : Traverse all memory domains of the node and initialize each memory domain

free_area_init_core

```

//mm/page_alloc.c
static void __paginginit free_area_init_core ( struct pglist_data * pgdat )
{
    enum zone_type j ;
    int nid = pgdat -> node_id ;
    unsigned long zone_start_pfn = pgdat -> node_start_pfn ;
    int ret ;

    ...
    for ( j = 0 ; j < MAX_NR_ZONES ; j ++ ) {

```

```

struct zone * zone = pgdat -> node_zones + j ;
unsigned long size , realsize , freesize , memmap_pages ;

size = zone -> spanned_pages ;
realsize = freesize = zone -> present_pages ;


```

- `zone->spanned_pages` represents all pages in the memory domain , including holes , see the explanation of the data structure for details
- `zone->present_pages` represents all pages after removing the holes

```

if ( ! is_highmem_idx ( j ) )
    nr_kernel_pages += freesize ;
/* Charge for highmem memmap if there are enough kernel pages */
else if ( nr_kernel_pages > memmap_pages * 2 )
    nr_kernel_pages -= memmap_pages ;
nr_all_pages += freesize ;

```

-  kernel uses two global variables to track the number of pages in the system. `nr_kernel_pages` counts all consistent mapped pages, and `nr_all_pages` also includes high-end memory pages

The remaining part of `free_area_init_core` 's task is to initialize each header in the zone structure and initialize each structure member to 0 . We are more interested in several auxiliary functions called .

- **zone_pcp_init** initializes the hot and cold caches of the memory domain . Regarding the hot and cold caches , I don't understand how to understand them , so I won't go into details here . If you are interested, you can read "In-depth Linux Kernel Architecture Section 3.4.2 - Initialization of Hot and Cold Caches "
- **init_currently_empty_zone** initializes the `free_area` list, and sets all page instances belonging to the memory domain to the initial default value. Initialized by `zone_init_free_lists` to complete

```

//mm/page_alloc.c
static void __meminit zone_init_free_lists ( struct zone * zone )
{
    unsigned int order , t ;
    for_each_migratetype_order ( order , t ) {
        INIT_LIST_HEAD ( & zone -> free_area [ order ] . free_list [ t ] ) ;
        zone -> free_area [ order ] . nr_free = 0 ;
    }
}

```

The number of free pages (`nr_free`) is currently still specified as 0 , which obviously does not reflect the real situation. Because all physical memory pages are managed by `memblock` at this time , the correct value will not be set until the `memblock` allocator is deactivated and the normal partner allocator takes effect.

- **memmap_init** , this function will eventually call `memmap_init_zone` to initialize the pages of the memory domain, and set the attributes of all pages to the `MIGRATE_MOVABLE` type . `MIGRATE_MOVABLE` we did not mention before , its main purpose is to avoid fragmentation . Because fragmentation is in the process of memory allocation produced , so we will be in 2.4 Jie discussed in detail introducing a buddy system `memmap_init_zone` function and `MIGRATE_MOVABLE` type .

2.3.4 build_all_zonelists

When we introduced the zonelist data structure in section 2.2.2 , we mentioned the role of build_all_zonelists . The purpose of this function is Creating a memory domain list of alternate current node , so that the node is not remaining in the current memory , allocated from the spare list.

Since the implementation of this function has little to do with the CPU architecture (in fact , the implementation of UMA and NUMA architecture is slightly different) , it is something at the operating system strategy level , so I will not go into details here , only give a rough code flow , in the future There is a need for analysis .

Implementation file : mm/page_alloc.c

build_all_zonelists

```
➤ build_all_zonelists_init
    ■ __build_all_zonelists
        ◆ #ifdef CONFIG_NUMA
            build_zonelists ...
        #else
            build_zonelists ...
        #endif
```

2.3.5 mem_init

After the system is initialized and the partner system allocator can assume the responsibility of memory management, the bootstrap allocator must be disabled. After all, two allocators cannot be used to manage memory at the same time. mem_init purpose it is deactivated bootstrap distributor .

The core actions are completed by the free_unused_memmap and free_all_bootmem functions . The processing details of these two functions have not been carefully studied . They will eventually call __free_pages (mm/page_alloc.c) , __free_pages will eventually make zone->free_area[order].nr_free++

After this , only the partner system can be used for memory allocation.

2.3.6 free_initmem

Many kernel code blocks and data tables are only needed during the system initialization phase. For example, for drivers linked to the kernel, it is not necessary to maintain initialization routines for their data structures in kernel memory. After the structure is established, these routines are no longer needed. Similarly, the driver is used to detect the hardware database of its device. After the related device has been identified, it is no longer needed.

The kernel provides two attributes (__init and __initcall) for marking initialization functions and data. These must be placed before the declaration of the function or data. For example, the detection routine of the network card HyperHopper2000 (hypothetical) will no longer be used after the system has been initialized.

```
int __init hyper_hopper_probe(struct net_device *dev)
```

The __init attribute is inserted between the return type and the function name in the function declaration.

Data segments can also be marked as initialization data. For example, a hypothetical network card driver requires some strings that are only used during the initialization phase of the system, after which these strings can be discarded.

```
static char search_msg[] __initdata = "%s: Desperately looking for HyperHopper at address %x...";
```

```
static char stilllooking_msg[] __initdata = "still searching...";
static char found_msg[] __initdata = "found.\n";
static char notfound_msg[] __initdata = "not found (reason = %d)\n";
static char couldnot_msg[] __initdata = "%s: HyperHopper not found\n";
```

__init and __initdata cannot be implemented in ordinary C language, so the kernel must resort to special GNU C compiler statements.

Behind the implementation of the initialization function, the general idea is to keep the data in a specific part of the kernel image, which can be completely deleted from the memory at the end of the startup. The following macro definitions serve this purpose:

```
< include/linux/ init . h >
#define __init __attribute__ ((__section__ (".init.text"))) __cold
#define __initdata __attribute__ ((__section__ (".init.data")))
```

__attribute__ is a special GNU C keyword through which attributes are used. __section__ attribute tells the compiler to write the subsequent data or function of each binary .init.data and .init.text segments. The prefix __cold also informs the compiler that the code path leading to the function is less likely, that is, the function will not be called frequently, which is usually the case for the initialization function.

The readelf tool can be used to display the various segments of the kernel image.

```
liuxin@ubuntu:~/bbb/linux$ readelf --sections vmlinux
There are 46 section headers, starting at offset 0x3d829a0:

Section Headers:
[Nr] Name                Type           Addr           Off           Size     ES Flg Lk Inf Al
[ 0]                     NULL           00000000       000000       000000  00  00 0  0  0
[ 1] .head.text             PROGBITS       c0008000       008000       00022c  00  AX  0  0  4
[ 2] .text                  PROGBITS       c0008240       008240       779358  00  AX  0  0 64
[ 3] .fixup                 PROGBITS       c0781598       781598       000024  00  AX  0  0  4
[ 4] .rodata                PROGBITS       c0782000       782000       3785b0  00  A   0  0 256
[ 5] __bug_table            PROGBITS       c0afa5b0       afa5b0       009e4c  00  A   0  0  4
[ 6] .builtin_fw            PROGBITS       c0b043fc       b043fc       000030  00  A   0  0  4
[ 7] __ksymtab              PROGBITS       c0b0442c       b0442c       009910  00  A   0  0  4
[ 8] __ksymtab_gpl          PROGBITS       c0b0dd3c       b0dd3c       007ec0  00  A   0  0  4
[ 9] __kcrctab              PROGBITS       c0b15bfc       b15bfc       004c88  00  A   0  0  4
[10] __kcrctab_gpl          PROGBITS       c0b1a884       b1a884       003f60  00  A   0  0  4
[11] __ksymtab_strings      PROGBITS       c0b1e7e4       b1e7e4       02a90d  00  A   0  0  1
[12] __param                PROGBITS       c0b490f4       b490f4       0012fc  00  A   0  0  4
[13] __modver               PROGBITS       c0b4a3f0       b4a3f0       000c10  00  A   0  0  4
[14] __ex_table             PROGBITS       c0b4b000       b4b000       000f50  00  A   0  0  8
[15] .ARM.unwind_idx        ARM_EXIDX      c0b4bf50       b4bf50       043998  00  AL 20  0  4
[16] .ARM.unwind_tab        PROGBITS       c0b8f8e8       b8f8e8       004338  00  A   0  0  4
[17] .notes                  NOTE           c0b93c20       b93c20       000024  00  AX  0  0  4
[18] .vectors               PROGBITS       00000000       ba0000       000020  00  AX  0  0  2
[19] .stubs                 PROGBITS       00001000       ba1000       0002c0  00  AX  0  0 32
[20] .init.text             PROGBITS       c0b942e0       ba42e0       04ab1c  00  AX  0  0 32
[21] .exit.text             PROGBITS       c0bdeffc       beedfc       001b98  00  AX  0  0  4
[22] .init.arch.info        PROGBITS       c0be0994       bf0994       000270  00  A   0  0  4
[23] .init.tagtable         PROGBITS       c0be0c04       bf0c04       000040  00  A   0  0  4
[24] .init.smpalt           PROGBITS       c0be0c44       bf0c44       009ae0  00  A   0  0  4
[25] .init.pv_table         PROGBITS       c0bea724       bf724       000258  00  A   0  0  1
[26] .init.data             PROGBITS       c0beb000       bfb000       038aec  00  WA  0  0 4096
[27] .data..percpu          PROGBITS       c0c24000       c34000       0064c0  00  WA  0  0 64
```

In order to release the initialization data from the memory, the kernel does not need to know the nature of the data, that is, which data and functions are stored in the memory and their purpose are completely irrelevant. The only relevant information is the start and end addresses of these data and functions in memory. Since this information is not available at compile time, it is inserted by the kernel at link time. This information is provided by vmlinux.ld.S, which defines a pair of variables __init_begin and __init_end, which have obvious meanings.

free_initmem is responsible for releasing the memory area used for initialization and returning the relevant pages to the partner system. This function will be called at the end of the startup process, followed by init as the first process in the system. The boot log contains a message indicating how much memory was freed.

```
[7.800069] Freeing unused kernel memory: 608K (c0b94000 – c0c2c000)
```

Compared with the size of main memory that is usually equipped today, the amount of memory released about 600 KiB is not large, but it also has a relatively important role. Especially on handheld or embedded systems, it is very important to clear the initialization data. The nature of such devices determines that they can only run with a small amount of memory.

2.4 Physical memory allocation (buddy)

2.4.1 Principle Introduction

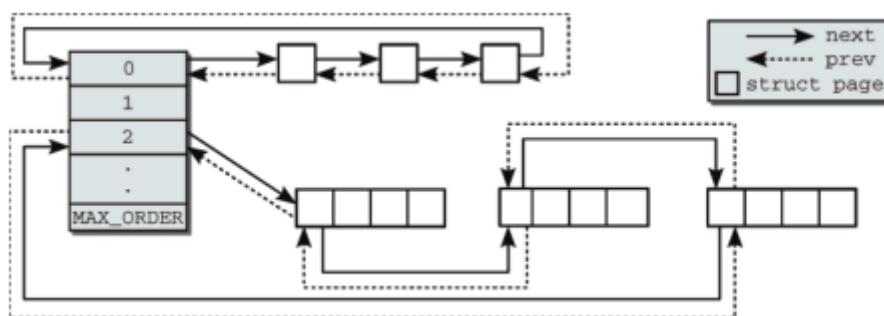
An excellent storage allocation system needs to consider two factors , speed and efficiency .

The so-called speed refers to the fast distribution speed , and the so-called efficiency refers to avoiding fragmentation as much as possible . The two are relative, and a balance point needs to be carefully sought . Here we do not intend to discuss how the partner system finds this balance point , just know The buddy system is based on a relatively simple but surprisingly powerful algorithm that has been with us for almost 40 years. It combines two key characteristics of a good memory allocator: speed and efficiency.

It has been introduced , the physical memory is divided into one page frame , we can simply put these pages together with a one-way linked list , when memory needs to be allocated , from the head of the list start searching for a free block to meet the requirements . Assumed that need to be assigned . 4 months For consecutive pages , we search for 4 consecutive free pages from the head of the linked list , mark them as allocated after finding them , and then return the page address to the applicant .

One problem of this is that the distribution rate will increase and increase physical memory . The more physical memory , the more the number of pages , the longer the list , the search time will be longer .

To solve the problem of excessively long linked lists , we can combine the advantages of linked lists and arrays , as shown in the following figure :



The vertical is an array , and the horizontal is a linked list .

The array item 0 means that there are 1 free pages ($2^0 = 1$) .

Array item 1 means that there are 2 consecutive free pages ($2^1 = 2$).

The array item 2 represents 4 consecutive free pages ($2^2 = 4$).

The array item MAX_ORDER represents N consecutive free pages ($2^{\text{MAX_ORDER}} = N$).

Array above item called " order " , such as 0 -order , an order .

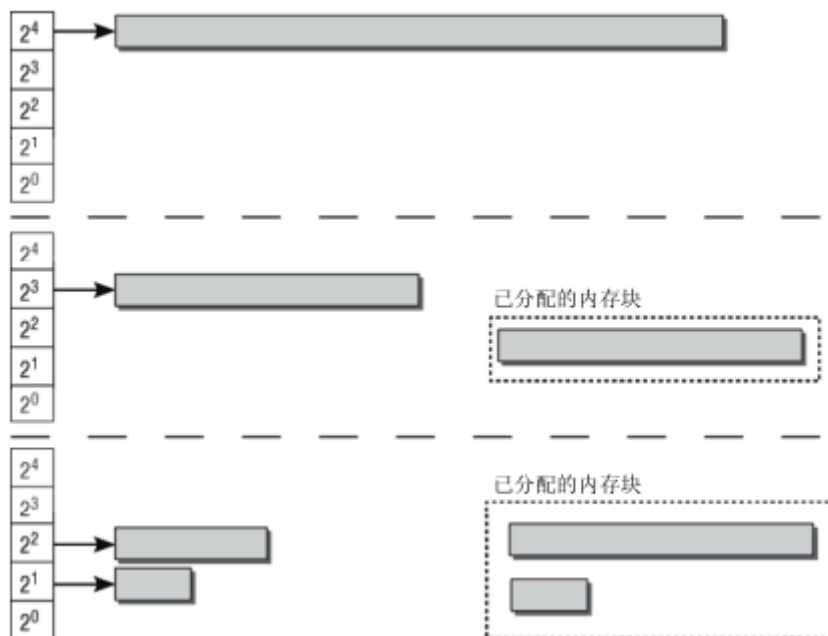
In this way , when we need to apply for 4 consecutive physical pages , we can directly search in the linked list behind the second- order array , and there is no need to search the linked list from the beginning .

The above is the basic idea of the buddy system . The buddy system can only allocate 2^{order} pages , which means that it can allocate 1, 2, 4, 8 ... continuous physical pages , but cannot allocate 3 continuous physical pages .

Come to think about a problem , the kernel initialization process , we have the physical memory is divided into a one-page frame of , the initial stage of these page frames are all continuous , then we should give 0 -order how many page frames array , to a number of bands a page frame block (multiple consecutive page frames to frame blocks called pages) it , ... ?

The real approach of the kernel is to initially treat all physical pages as a page block and mount it on the array of the corresponding order . Then, when allocating memory, this large page block is split into partners .

Assuming that the system has a total of 16 physical pages , the following figure demonstrates the working situation of the partner system :



If the system now needs 8 page frames, the block composed of 16 page frames is split into two partners. One of the blocks is used to satisfy the request of the application, and the remaining 8 page frames are placed in the list of corresponding 8- page memory blocks.

If the next request only requires 2 consecutive page frames, the block consisting of 8 pages will be split into 2 partners, each containing 4 page frames. One piece is placed back into the buddy list, and the other is split into 2 buddies again , each containing 2 pages. One of them goes back to the buddy system, and the other is passed to the application.

When the application releases the memory, the kernel can directly check the address to determine whether a group of partners can be created and merged into a larger memory block and put back into the partner list. This is just the reverse process of the memory block split. This increases the likelihood that larger memory blocks will be available.

Why can directly check the address to determine whether to create a group of partners ? For example, we put 8-16 this 8 consecutive physical pages split into 2 pages block : the start address of the first page of the block is 0x8 (binary 1000); the start address of the second page block is 0xC (binary 1100); a block and its partner have only 1 bit difference in address . Therefore, we can check the address and when a pair of partners is found , the two Merge into a larger block . The purpose of merging is to avoid fragmentation , otherwise when the system runs for a long time , we will not be able to apply for a large free block .

However, the buddy system cannot completely eliminate fragmentation . During the development of kernel version 2.6.24 , some effective measures have been added to prevent memory fragmentation . We will introduce its details later .

2.4.2 Related data structure

free_area

Each physical memory page (page frame) in the system memory corresponds to a struct page instance. Each memory domain is associated with an instance of struct zone , which holds the main array used to manage partner data.

```
//include/linux/ mmzone.h
struct zone {
    ...
    /*
     * Free areas of different lengths
     */
    struct free_area free_area [ MAX_ORDER ];
    ...
};
```

free_area is an auxiliary data structure, we haven't introduced it in detail before . It is defined as follows:

```
//include/linux/mmzone.h
struct free_area {
    struct list_head free_list [ MIGRATE_TYPES ];
    unsigned long      nr_free ;
};
```

nr_free of the idle memory area of the current **page block** number (for 0 -order memory area by page calculation, for a page number of the calculation order of the memory section, 2 -order memory area calculation 4 number of sets of pages, and so on) . free_list It is a linked list used to connect free pages. According to the discussion in "Introduction to Principles", the page linked list contains contiguous memory blocks of the same size. Although the definition provides multiple page linked lists, I will ignore this fact for now, and discuss its reasons in "Avoiding Fragmentation" below.

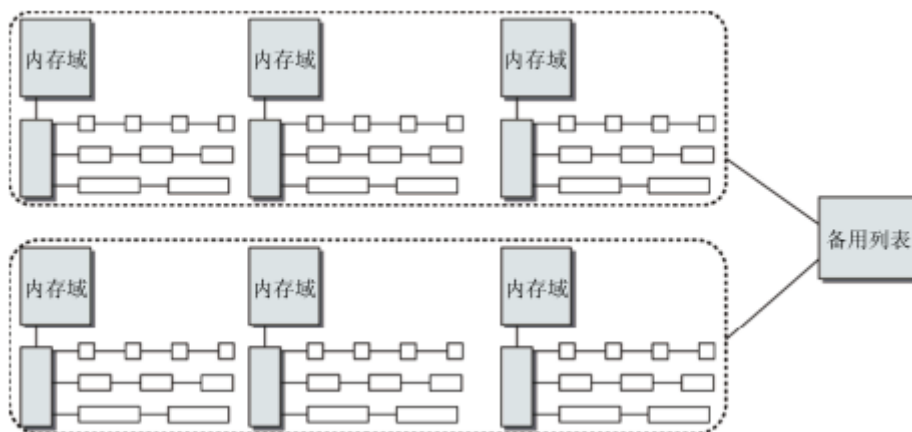
MAX_ORDER represents the maximum order of the system , and it is defined as follows :

```
//include/linux/mmzone.h
/* Free memory management - zoned buddy allocator. */
#ifndef CONFIG_FORCE_MAX_ZONEORDER
#define MAX_ORDER 11
#else
#define MAX_ORDER CONFIG_FORCE_MAX_ZONEORDER
#endif
#define MAX_ORDER_NR_PAGES (1 << (MAX_ORDER - 1))
```

This constant is usually set to 11 , which means that the maximum number of pages that can be requested for an allocation is $2^{11} = 2048$. But if the architecture-specific code sets the `FORCE_MAX_ZONEORDER` configuration option, this value can also be changed manually.

`free_area [MAX_ORDER]` is the array we discussed in "Introduction to Principles" .

The memory management based on the buddy system focuses on a certain memory domain of a certain node, for example, DMA or high-end memory domain. However, all memory domains and node partner systems are connected through alternate allocation lists. As shown below :



When the preferred memory domain or node cannot satisfy the memory allocation request, first try another memory domain of the same node, and then try another node until the request is satisfied.

/proc/buddyinfo

Finally, note that information about the current state of the buddy system can be obtained in `/proc/buddyinfo` :

```
wolfgang@meitner> cat /proc/buddyinfo
Node 0, zone DMA 3 5 7 4 6 3 3 3 1 1 1
Node 0, zone DMA32 130 546 695 271 107 38 2 2 1 4 479
Node 0, zone Normal 23 6 6 8 1 4 3 0 0 0 0
```

The above output gives the number of free items in each allocation order in each memory domain, from left to right, the order increases. The information given above is taken from an AMD64 system with 4 GiB of physical memory .

We look at BeagleBone Black circumstances of this board , it uses the TI 3358 CPU, 512M onboard memory .

```
root@bebest:~# cat /proc/buddyinfo
Node 0, zone Normal 33 50 39 19 7 3 3 2 2 3 3 52
```

The system only . 1 nodes , nodes below only . 1 two memory domain . No DMA memory field , because the ARM architecture of DMA no special requirements accessed ; there is no HIGHMEM domain , since on-board memory is too small .

2.4.3 Avoid Fragmentation - Mobility Grouping

According to "introduce the principle of" an idea , each step of the memory block , just a list head mount to the next can , why `struct free_area` defined inside the head of the list is actually an array ?

In kernel version 2.6.23 before , indeed only one head of the list . But in kernel 2.6.24 During development , the kernel developers debate on the buddy system lasted quite a long time . This is because the buddy system is a core part of the most respected , it changes everybody will not be easily accepted . the end result is that each stage of

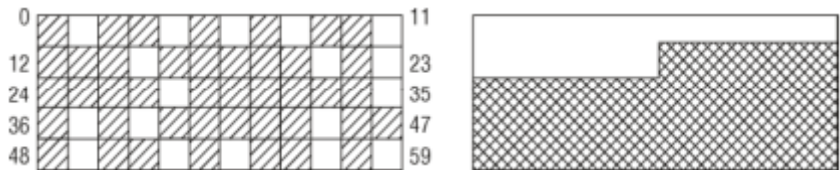
the debate memory block classified under multiple mount head of the list , the purpose of doing so is to prevent fragmentation .

Let's take a look at why this can prevent fragmentation .

Background knowledge : organize pages according to mobility

The basic principles of the buddy system have been discussed above, and its solution has indeed worked very well in recent years. But in Linux

In terms of memory management, there is a long-standing problem: After the system starts and runs for a long time, the physical memory will produce a lot of fragments. This situation is shown in the figure below.



Assuming that the memory consists of 60 pages, this is obviously not a supercomputer, but it is sufficient for the example. Free pages are scattered in the address space on the left. Although approximately 25% of the physical memory is still unallocated, the largest contiguous free area is only one page. This is no problem for user-space applications: its memory is mapped through the page table, and regardless of the distribution of free pages in physical memory, the memory that the application sees always seems to be continuous. In the situation given in the figure on the right, the number of free pages and used pages is the same as the figure on the left, but all free pages are located in a contiguous area.

But for the kernel, fragmentation is a problem. Since (most) physical memory is consistently mapped to the kernel part of the address space, in the scenario on the left, it is impossible to map a memory area larger than a page. Although many times the kernel allocates relatively small memory, but sometimes it is necessary to allocate more than one page of memory. Obviously, in the case of allocating larger memory, it is more preferable to have all the allocated pages and free pages in the contiguous memory area in the picture on the right.




It is interesting to note that fragmentation problems may also occur when most of the memory is still unallocated. Consider the situation in the figure below. Only 4 pages are allocated , but the largest contiguous area that can be allocated is only 8 pages, because the allocation range that the buddy system can work can only be a power of 2 .






Figure 3-25

For a long time, the fragmentation of physical memory is indeed one of the weaknesses of Linux . Although many methods have been proposed, no method can meet the demanding requirements of various types of workloads that Linux needs to handle, while at the same time having little impact on other matters. During the development of kernel 2.6.24 , the method of preventing fragmentation was finally added to the kernel. Before I discuss specific strategies, there is one point that needs to be clarified. The file system is also fragmented, and the fragmentation problem in this area is mainly solved by fragment merging tools. They analyze the file system and reorder the allocated storage blocks to create larger contiguous storage areas. In theory, this method is also possible for physical memory, but because many physical memory pages cannot be moved to any position, the implementation of this method is hindered. Therefore, the kernel method is anti-fragmentation (Anti-fragmentation) , that is trying to prevent debris as possible from the very beginning.

How does anti-fragmentation work? To understand this method, we must know that the kernel divides the allocated pages into the following three different types.



➤    page: There is a fixed position in the memory and cannot be moved to other places.

Most of the memory allocated by the core kernel falls into this category.

➤    page: It cannot be moved directly, but it can be deleted, and its content can be regenerated from some sources. For example, data mapped from a file belongs to this category.

The kswapd daemon will periodically release this type of memory according to the frequency of reclaimable page accesses. This is a complicated process, which itself needs to be discussed in detail. Currently, it is enough to know that the kernel will reclaim when reclaimable pages occupy too much memory.

In addition, page reclamation can also be initiated when there is a memory shortage (that is, an allocation failure). For more specific information about the timing of page recycling initiated by the kernel, please refer to "In-depth Linux Kernel Architecture Chapter 18 "

➤   movable page can be moved at will. Pages belonging to user space applications belong to this category. They are mapped through the page table. If they are copied to the new location, the page table entries can be updated accordingly, and the application will not notice anything.

The mobility of a page depends on which of the three categories the page belongs to . The anti-fragmentation technology used by the kernel is based on the idea of grouping pages with the same mobility. Why does this method help reduce fragmentation? Recall that in Figure 3-25 , because the page cannot be moved, continuous allocation cannot be performed in the memory area that was almost completely empty. According to the mobility of the page, assign it to different lists to prevent this situation. For example, an immovable page cannot be located in the middle of a removable memory area, otherwise a large contiguous memory block cannot be allocated from this memory area.

Suppose that most of the free pages in Figure 3-25 belong to the recyclable category, and the allocated pages are not removable. If these pages are gathered into two different lists, as shown in Figure 3-26 . It is still difficult to find large continuous free space in non-removable pages, but it is much easier for recyclable pages.

可回收页 

不可移动页 

图3-26 根据页的可移动性将其分组，减少了内存碎片

But it should be noted that from the beginning, the memory was not divided into areas with different mobility. These are formed at runtime. Another method of the kernel does partition the memory for the allocation of movable and non-movable pages. I will discuss its working principle below. But this division is not necessary for the method described here.

Related data structure (MIGRATE_XXX)

Although the anti-fragmentation technology used by the kernel is very effective, it has little effect on the code and data structure of the partner allocator. The kernel defines some macros to indicate different migration types:

```
//include/linux/mmzone.h
enum {
    MIGRATE_UNMOVABLE ,
    MIGRATE_MOVABLE ,
```

```

MIGRATE_RECLAIMABLE ,
MIGRATE_PCPTYPES , /* the number of types on the pcp lists */
MIGRATE_HIGHATOMIC = MIGRATE_PCPTYPES ,
#ifdef CONFIG_CMA
/*
 * MIGRATE_CMA migration type is designed to mimic the way
 * ZONE_MOVABLE works. Only movable pages can be allocated
 * from MIGRATE_CMA pageblocks and page allocator never
 * implicitly change migration type of MIGRATE_CMA pageblock.
 *
 * The way to use it is to change migratetype of a range of
 * pageblocks to MIGRATE_CMA which can be done by
 * __free_pageblock_cma() function. What is important though
 * is that a range of pageblocks must be aligned to
 * MAX_ORDER_NR_PAGES should biggest page be bigger than
 * a single pageblock.
 */
MIGRATE_CMA ,
#endif
#ifdef CONFIG_MEMORY_ISOLATION
MIGRATE_ISOLATE , /* can't allocate from here */
#endif
MIGRATE_TYPES
};

```

The types `MIGRATE_UNMOVABLE` , `MIGRATE_RECLAIMABLE` and `MIGRATE_MOVABLE` have already been introduced.

`MIGRATE_ISOLATE` is a special virtual area used to move physical memory pages across NUMA nodes. On large systems, it is beneficial to move the physical memory page close to the CPU that uses the page most frequently .

`MIGRATE_TYPES` only represents the number of migration types, and does not represent a specific area.

The main adjustment to the data structure of the buddy system is to decompose the free list into `MIGRATE_TYPE` lists:

```

//include/linux/mmzone.h
struct free_area {
    struct list_head free_list[ MIGRATE_TYPES ];
    unsigned long nr_free;
};

```

`nr_free` counts the number of free pages on all lists, and each migration type corresponds to a free list. The macro `for_each_migratetype_order(order, type)` can be used to iterate all allocation orders of a specified migration type.

What if the kernel cannot satisfy the allocation request for a given migration type ? A similar problem has occurred before, that is, when a specific NUMA memory domain cannot satisfy the allocation request. The kernel's approach in this case is similar, providing an alternate list that specifies which migration type should be used next when the allocation request cannot be met in the specified list:

```

mm/page_alloc.c
/*

```

* This array describes the order of other free lists in the standby list when the free list of the specified migration type is exhausted.

```
*/
/*
 * This array describes the order lists are fallen back to when
 * the free lists for the desirable migrate type are depleted
 */
static int fallbacks[MIGRATE_TYPES][4] = {
    [MIGRATE_UNMOVABLE] = { MIGRATE_RECLAIMABLE, MIGRATE_MOVABLE, MIGRATE_TYPES },
    [MIGRATE_RECLAIMABLE] = { MIGRATE_UNMOVABLE, MIGRATE_MOVABLE, MIGRATE_TYPES },
    [MIGRATE_MOVABLE] = { MIGRATE_RECLAIMABLE, MIGRATE_UNMOVABLE, MIGRATE_TYPES },
#ifdef CONFIG_CMA
    [MIGRATE_CMA] = { MIGRATE_TYPES }, /* Never used */
#endif
#ifdef CONFIG_MEMORY_ISOLATION
    [MIGRATE_ISOLATE] = { MIGRATE_TYPES }, /* Never used */
#endif
};
```

This data structure is generally self-explanatory: for example, when the kernel wants to allocate an immovable page, if the corresponding linked list is empty, it will fall back to the recyclable page linked list, and then to the movable page linked list.

Enable mobility group (`page_group_by_mobility_disabled`)

Although the page mobility grouping feature is always compiled into the kernel, it only makes sense when there is enough memory in the system that can be allocated to the linked lists corresponding to multiple migration types. If there is not a large continuous memory in the linked list of each migration type, then page migration will not provide any benefits, so the kernel will turn off this feature when the available memory is too small.

A global variable is declared in `include/linux/mmzone.h` : `extern int page_group_by_mobility_disabled;`

`page_group_by_mobility_disabled = 1`: It means that the mobility group feature is prohibited

`page_group_by_mobility_disabled = 0`: means that the mobility group feature can be used

This global variable is initialized in the `build_all_zonelists` function :

```
//mm/page_alloc.c
void __ref build_all_zonelists ( pg_data_t * pgdat , struct zone * zone )
{
    ...

    vm_total_pages = nr_free_pagecache_pages ();
    /*
     * Disable grouping by mobility if the number of pages in the
     * system is too low to allow the mechanism to work. It would be
     * more accurate, but expensive to check per-zone. This check is
     * made on memory-hotadd so a system can start with mobility
     * disabled and enable it later
     */
    if ( vm_total_pages < ( pageblock_nr_pages * MIGRATE_TYPES ) )
        page_group_by_mobility_disabled = 1 ;
    else
        page_group_by_mobility_disabled = 0 ;

    ...
}
```

```
}
```

The judgment condition is if $(vm_total_pages < (pageblock_nr_pages * MIGRATE_TYPES))$: vm_total_pages represents the total number of memory pages available in the system ; $MIGRATE_TYPES$ is a constant , $pageblock_nr_pages$ is a macro , and its definition is as follows :

```
// include/linux/pageblock-flags.h

#ifdef CONFIG_HUGETLB_PAGE

#ifdef CONFIG_HUGETLB_PAGE_SIZE_VARIABLE

/* Huge page sizes are variable */
extern unsigned int pageblock_order ;

#else /* CONFIG_HUGETLB_PAGE_SIZE_VARIABLE */

/* Huge pages are a constant size */
#define pageblock_order HUGETLB_PAGE_ORDER

#endif /* CONFIG_HUGETLB_PAGE_SIZE_VARIABLE */

#else /* CONFIG_HUGETLB_PAGE */

/* If huge pages are not used, group by MAX_ORDER_NR_PAGES */
#define pageblock_order (MAX_ORDER-1)

#endif /* CONFIG_HUGETLB_PAGE */

#define pageblock_nr_pages (1UL << pageblock_order)
```

$pageblock_nr_pages$ depends on $pageblock_order$, and the latter depends on whether the system enables $CONFIG_HUGETLB_PAGE$:

- If not enabled $CONFIG_HUGETLB_PAGE$, the $pageblock_order = (MAX_ORDER-1)$. ARM architecture most of this situation .
- If enabled $CONFIG_HUGETLB_PAGE$, for example, in the IA-32 on the architecture, the page length is huge . 4 MiB , so each page consists of a giant 1024 ordinary pages, and $HUGETLB_PAGE_ORDER$ was defined as 10 .

In contrast, the IA-64 architecture allows variable normal and jumbo page lengths to be set, so the value of $HUGETLB_PAGE_ORDER$ depends on the kernel configuration.

Auxiliary functions and variables

If the characteristics of mobility packets available, then , when we call the buddy system API an application for memory , the kernel How do I know in the end should get memory which packet inside from ? The answer is that the applicant specified . The applicant in the application memory , you need to specify from Which node of which memory domain and which migration type applies for memory .

gfpflags_to_migratetype

The kernel provides two flags to indicate that the allocated memory is removable (`__GFP_MOVABLE`) or recyclable (`__GFP_RECLAIMABLE`) . If none of these flags are set, the allocated memory is assumed to be immovable. The following auxiliary functions can be used to convert the allocation flag and the corresponding migration type:

```
//include/linux/gfp.h
static inline int gfpflags_to_migratetype ( const gfp_t gfp_flags )
{
    VM_WARN_ON (( gfp_flags & GFP_MOVABLE_MASK ) == GFP_MOVABLE_MASK );
    BUILD_BUG_ON (( 1UL << GFP_MOVABLE_SHIFT ) !=! __GFP_MOVABLE );
    BUILD_BUG_ON (( __GFP_MOVABLE >> GFP_MOVABLE_SHIFT ) != MIGRATE_MOVABLE );

    if ( unlikely ( page_group_by_mobility_disabled ) )
        return MIGRATE_UNMOVABLE ;

    /* Group based on mobility */
    return ( gfp_flags & GFP_MOVABLE_MASK ) >> GFP_MOVABLE_SHIFT ;
}
```

If the page migration feature is disabled, all pages are immovable. Otherwise, the return value of this function can be directly used as the array index of `free_area.free_list` .

pageblock_flags

Assuming that in the initial stage , the mobility properties of all pages are MOVABLE (in fact, this is also the case , we will introduce when the page properties are marked as removable during the initialization process) , when we apply for a non-movable When the memory is used , we must record the mobility properties of this memory , so that when this memory is released , we can mount it on the correct migration list (`free_list[MIGRATE_TYPES]`) .

Where should the movability attribute of the memory block be recorded ? Should it be stored in the struct page structure ? This is inappropriate . The movability attribute is for the memory block , not for the page. A memory block may contain multiple a page, if the attributes are stored in page structure inside , and that each of the memory block page are to be marked as corresponding property , this will lead to complicated operation , and will increase the page structure size.

So the kernel saves this field in the struct zone structure . Since this field is currently only used by code related to page mobility, I did not introduce this field in detail before :

```
//include/linux/mmzone.h
struct zone {
    ...
    unsigned long * pageblock_flags ;
    ...
}
```

The migration attribute of each memory block only needs to occupy a few bits, so one unsigned long data can represent multiple memory blocks , and unsigned long * is equivalent to defining multiple unsigned longs , so it can represent many, many memory blocks !

During the memory management system initialization , the kernel is * pageblock_flags pointer allocate enough memory space , to ensure that it is stored in the system migration properties of all memory blocks . Function responsible for allocating storage space is setup_usemap (mm / page_alloc.c) .

The current , the migration properties of the needs of a memory block . 4 th bit to mark :

```
//include/linux/pageblock-flags.h

/* Bit indices that affect a whole block of pages */
enum pageblock_bits {
    PB_migrate ,
    PB_migrate_end = PB_migrate + 3 - 1 ,
        /* 3 bits required for migrate types */
    PB_migrate_skip , /* If set the block is skipped by compaction */

    /*
     * Assume the bits will always align on a word. If this assumption
     * changes then get/set pageblock needs updating.
     */
    NR_PAGEBLOCK_BITS
};
```

set /get _pageblock_migratetype

set_pageblock_migratetype is responsible for setting the migration type of a memory block headed by page :

```
//mm/page_alloc.c

void set_pageblock_migratetype ( struct page * page , int migratetype )
{
    if ( unlikely ( page_group_by_mobility_disabled &&
        migratetype < MIGRATE_PCPTYPES ))
        migratetype = MIGRATE_UNMOVABLE ;

    set_pageblock_flags_group ( page , ( unsigned long ) migratetype ,
        PB_migrate , PB_migrate_end );
}
```

The migratetype parameter can be constructed by the gfpflags_to_migratetype helper function described above .

When the memory is released, the page must be returned to the correct migration list. This is possible because it can be downloaded from get_pageblock_migratetype to obtain the migration type of the memory block.

```
//include/linux/mmzone.h

#define get_pageblock_migratetype(page) \
    get_pfnblock_flags_mask(page, page_to_pfn(page), \
        PB_migrate_end, MIGRATETYPE_MASK)
```

Initial default value (memmap_init_zone)

During the initialization of the memory subsystem, memmap_init_zone (setup_arch -> paging_init -> bootmem_init -> free_area_init_node -> free_area_init_core -> memmap_init -> memmap_init_zone) is responsible for handling page instances of the memory domain . This function completes some not-so-interesting standard initialization work, but one of them is substantial, that is, all pages are initially marked as removable!

```
//mm/page_alloc.c

void __meminit memmap_init_zone ( unsigned long size , int nid , unsigned long
zone ,
    unsigned long start_pfn , enum memmap_context context )
{
    ...

    IF ((! PFN & ( pageblock_nr_pages - . 1 ))) {
        struct Page * Page = pfn_to_page ( PFN );

        __init_single_page ( Page , PFN , Zone , NID );
        set_pageblock_migratetype ( page , MIGRATE_MOVABLE );
    } else {
        __init_single_pfn ( PFN , Zone , NID );
    }

    ...
}
```

According to the discussion in "Related Data Structure (MIGRATE_XXX)", when allocating memory, if the required migration type does not have available memory, apply to the " larger " migration type. Since all the original pages are mobile, so the kernel is not allocated when the removable memory area, you must "steal."

In fact, there are fewer cases of allocating removable memory areas during startup, so the allocator has a high probability of acquiring memory from the removable area and converting it from the removable list to the non-movable list. At this time, no fragmentation is introduced into the removable memory.

All in all, this approach avoids the memory allocated by the kernel during startup (often not released during the entire running time of the system) scattered throughout the physical memory, so that other types of memory allocation are free from fragmentation, which is also page movable One of the most important goals of the sexual grouping framework.

/proc/pagetypeinfo

/proc/pagetypeinfo reflects " Free pages count per migrate type at order " , which is equivalent to the refinement of /proc/buddyinfo .

Look at the output on the BBB board :


```

root@embest:~# cat /proc/pagetypeinfo
Page block order: 11
Pages per block: 2048
Free pages count per migrate type at order
Node 0, zone Normal, type Unmovable 30 1 2 3 4 5 6 7 8 9 10 11
Node 0, zone Normal, type Movable 1 50 15 6 3 1 0 1 0 1 1 0
Node 0, zone Normal, type Reclaimable 1 0 1 1 1 1 1 1 1 0 1 0
Node 0, zone Normal, type HighAtomic 0 0 0 0 0 0 0 0 0 0 0 0
Node 0, zone Normal, type CMA 0 1 1 1 1 1 0 1 1 1 1 2
Node 0, zone Normal, type Isolate 0 0 0 0 0 0 0 0 0 0 0 0
Number of blocks type unmovable Movable Reclaimable HighAtomic CMA Isolate
Node 0, zone Normal 2 57 2 0 3 0
root@embest:~# cat /proc/buddyinfo
Node 0, zone Normal 32 83 35 12 6 3 2 3 2 3 3 52

```

2.4.4 Avoid fragmentation - virtual removable memory domain

background knowledge

Organizing pages according to mobility is a possible way to prevent physical memory fragmentation, and the kernel also provides another means to prevent this problem: the virtual memory domain `ZONE_MOVABLE`. This mechanism has been incorporated into the kernel during the development of kernel 2.6.23, which is one version earlier than the mobility grouping framework added to the kernel.

Contrary to the mobility grouping, the `ZONE_MOVABLE` feature **will not be compiled into the kernel by default**. This feature is available only when the `CONFIG_HAVE_MEMBLOCK_NODE_MAP` switch is enabled.

The basic idea is simple: the available physical memory is divided into two memory domains, one for removable allocation and one for non-movable allocation. This will automatically prevent non-movable pages from introducing fragmentation into the removable memory domain.

This immediately leads to another question: how does the kernel allocate available memory between two competing memory domains? This obviously requires too much of the kernel, so the system administrator must make a decision. After all, people can better predict the scenarios that the computer needs to handle and the expected distribution of various types of memory allocation.

Related data structure

required_kernelcore & required_movablecore

The kernel code defines two `early_param`, which can be modified through `bootargs`. The code is as follows:

```

//mm/page_alloc.c

#ifdef CONFIG_HAVE_MEMBLOCK_NODE_MAP
...
static unsigned long __initdata required_kernelcore ;
static unsigned long __initdata required_movablecore ;
...
#endif /* CONFIG_HAVE_MEMBLOCK_NODE_MAP */

#ifdef CONFIG_HAVE_MEMBLOCK_NODE_MAP
.....
/*
* kernelcore=size sets the amount of memory for use for allocations that

```

```

* cannot be reclaimed or migrated.
*/
static int __init cmdline_parse_kernelcore ( char * p )
{
    return cmdline_parse_core ( p , & required_kernelcore );
}

/*
* movablecore=size sets the amount of memory for use for allocations that
* can be reclaimed or migrated.
*/
static int __init cmdline_parse_movablecore ( char * p )
{
    return cmdline_parse_core ( p , & required_movablecore );
}

early_param ( "kernelcore" , cmdline_parse_kernelcore );
early_param ( "movablecore" , cmdline_parse_movablecore );
#endif /* CONFIG_HAVE_MEMBLOCK_NODE_MAP */

```

- kernelcore parameter is used to specify the amount of memory used for non-removable allocation, that is, the amount of memory that can neither be reclaimed nor migrated. The remaining memory is used for removable allocation. After analyzing this parameter, the result is saved in the global variable required_kernelcore .
- You can also use the parameter movablecore control for the amount of memory allocated removable memory. The result is stored in the global variable required_movablecore . The size of required_kernelcore will be calculated accordingly.
- some smart people specify two parameters at the same time, the kernel will calculate the required_kernelcore value according to the aforementioned method , and then take the larger of the calculated value and the specified value.

ZONE_MOVABLE

Depending on the architecture and kernel configuration, the ZONE_MOVABLE memory domain may be located in the high-end or normal memory domain:

```

//include/linux/mmzone.h

enum zone_type {
#ifdef CONFIG_ZONE_DMA
    ZONE_DMA ,
#endif
#ifdef CONFIG_ZONE_DMA32
    ZONE_DMA32 ,
#endif
/*
* Normal addressable memory is in ZONE_NORMAL. DMA operations can be
* performed on pages in ZONE_NORMAL if the DMA devices support
* transfers to all addressable memory.

```

```

    */
    ZONE_NORMAL ,
#ifdef CONFIG_HIGHMEM
    ZONE_HIGHMEM ,
#endif
    ZONE_MOVABLE ,
#ifdef CONFIG_ZONE_DEVICE
    ZONE_DEVICE ,
#endif
    __MAX_NR_ZONES
};

```

Contrary to all other memory domains in the system, ZONE_MOVABLE is not associated with any meaningful memory range on the hardware. In fact, the memory in the memory domain is accessed from the high-end memory domain or the ordinary memory domain, so we will call ZONE_MOVABLE a virtual memory domain in the following.

movable_zone & zone_movable_pfn

There are also two global variables briefly introduced :

```

//mm/page_alloc.c

#ifdef CONFIG_HAVE_MEMBLOCK_NODE_MAP
...

static unsigned long __meminitdata zone_movable_pfn [ MAX_NUMNODES ];

/* movable_zone is the "real" zone pages in ZONE_MOVABLE are taken from */
int movable_zone ;
EXPORT_SYMBOL ( movable_zone );
#endif /* CONFIG_HAVE_MEMBLOCK_NODE_MAP */

```

- Movable_zone represents the real physical memory zone from which the memory of the virtual memory zone ZONE_MOVABLE comes from
- for each node is, zone_movable_pfn [node_id] represents ZONE_MOVABLE in movable_zone starting memory address of the memory the acquired domain.

accomplish

How is the data structure described so far applied? Similar to the page migration method, the allocation flag plays a key role here. The specific implementation will be discussed in more detail in section 2.4.5 . At present, as long as you know that all movable allocations must specify __GFP_HIGHMEM and __GFP_MOVABLE .

Since the kernel determines the memory domain for memory allocation according to the allocation flag, when the above flag is set, the ZONE_MOVABLE memory domain can be selected . This is the only change required to integrate ZONE_MOVABLE into the partner system! The rest can be handled by common routines applicable to all memory domains, which we will discuss below.

2.4.5 Allocator API

As far as the API of the partner system is concerned, there is no difference between the NUMA and UMA architectures, and the calling syntax of the two is the same. All functions are one thing in common: only allocate 2 integer power of two pages. Therefore, the interface does not specify the required memory size as a parameter like the malloc function of the C standard library or the bootstrap allocator. Instead, the allocation order must be specified, and the partner system will allocate 2^{order} pages in memory. The fine-grained allocation in the kernel can only be done with the help of the slab allocator (or slub, slob allocator), which is based on the buddy system (more details are given in section 2.5).

Allocation page API

Header file : include/linux/gfp.h

Note: gfp is short for get free page.

Allocation page API	Comment
alloc_pages(gfp_mask, order)	Allocate 2^{order} pages and return an instance of struct page, which represents the starting page of the allocated memory block
alloc_page(gfp_mask)	It is a simplified form of the former in the case of order = 0, only one page is allocated
get_zeroed_page(gfp_mask)	Allocate a page and return a page instance, the memory corresponding to the page is filled with 0 (all other functions, the content of the page after the allocation is undefined)
__get_free_pages(gfp_mask, order) __get_free_page(gfp_mask)	Works in the same way as the above function, but returns the virtual address of the allocated memory block instead of the page instance
__get_dma_pages(gfp_mask, order)	Used to obtain pages suitable for DMA. The implementation of this API is very simple, with the help of the previous API: __get_free_pages((gfp_mask GFP_DMA), (order))

In the case where the free memory cannot satisfy the request and the allocation fails, all the above functions return null pointers (alloc_pages and alloc_page) or 0 (get_zeroed_page, __get_free_pages and __get_free_page). Therefore, the kernel must check the returned result after each allocation. This convention is no different from a well-designed user-level application, but ignoring checks in the kernel can cause much more serious failures.

In addition to the API provided by the partner system, the kernel also provides other memory management functions. They are based on the buddy system, but do not belong to the buddy distributor itself. These functions include:

- vmalloc and vmalloc_32, use the page table to map discontinuous memory to the kernel address space to make it look continuous. Its implementation will be discussed in detail when introducing virtual memory management in Chapter 3.
- kmalloc , kfree , kzalloc , kfreez , kmemdup , kmalloc_node , kfree_node , kmalloc_node_slab , kfree_node_slab , kmalloc_cmm , kfree_cmm , kmalloc_cmm_node , kfree_cmm_node , $\text{kmalloc_cmm_node_slab}$, $\text{kfree_cmm_node_slab}$, $\text{kmalloc_cmm_node_slab}$, $\text{kfree_cmm_node_slab}$, $\text{kmalloc_cmm_node_slab}$, $\text{kfree_cmm_node_slab}$ also a set of kmalloc type functions, which are used to allocate memory areas less than a full page. Its realization will be discussed in the slab allocator.

Free page API

Header file : include/linux/gfp.h

Free page API	Comment
---------------	---------

<code>__free_pages (struct page *, order)</code>	Used to return one or two ^{order} pages to the memory management subsystem. The starting address of the memory area is represented by a pointer to the first page instance of the memory area .
<code>__free_page (struct page *)</code>	It is a simplified form of the former in the case of order = 0 , only one page is released
<code>free_pages (addr, order)</code> <code>free_page (addr)</code>	The semantics are similar to the first two functions, but when representing the memory area that needs to be released, the virtual memory address is used instead of the page instance.

Assign mask GFP_XXX

What is the semantics of the mandatory `gfp_mask` parameter used in all the aforementioned functions ?

1. Select memory domain

From the foregoing discussion of us know, Linux divides memory into the memory domain. The kernel provides so-called memory domain modifiers to specify from which memory domain the required pages are allocated.

```
//include/linux/gfp.h

/* Plain integer GFP bitmasks. Do not use this directly. */
#define __GFP_DMA 0x01u
#define __GFP_HIGHMEM 0x02u
#define __GFP_DMA32 0x04u
#define __GFP_MOVABLE 0x08u
#define __GFP_RECLAIMABLE 0x10u
#define __GFP_HIGH 0x20u
...

/*
 * Physical address zone modifiers (see linux/mmzone.h - low four bits)
 *
 * Do not put any conditional on these. If necessary modify the definitions
 * without the underscores and use them consistently. The definitions here may
 * be used in bit comparisons.
 */
#define __GFP_DMA      ((__force gfp_t) __GFP_DMA)
#define __GFP_HIGHMEM  ((__force gfp_t) __GFP_HIGHMEM)
#define __GFP_DMA32   ((__force gfp_t) __GFP_DMA32)
#define __GFP_MOVABLE  ((__force gfp_t) __GFP_MOVABLE) /* Page is movable */
#define __GFP_MOVABLE  ((__force gfp_t) __GFP_MOVABLE) /* ZONE_MOVABLE
allowed */
#define GFP_ZONEMASK ( __GFP_DMA | __GFP_HIGHMEM | __GFP_DMA32 | __GFP_MOVABLE)
...
```

`__GFP_MOVABLE` does not indicate a physical memory domain, it informs the kernel that it should be in a special virtual memory domain `ZONE_MOVABLE`

Make the appropriate allocation .

It is very interesting that there is no `__GFP_NORMAL` constant, but the main burden of memory allocation falls on the `ZONE_NORMAL` memory domain!

The kernel provides a function to calculate the highest memory area compatible with a given allocation flag. Memory allocation can be performed from this memory domain or a lower memory domain.

```
//include/linux/gfp.h

static inline enum zone_type gfp_zone ( gfp_t flags )
{
    enum zone_type z ;
    int bit = ( __force int ) ( flags & GFP_ZONEMASK );

    z = ( GFP_ZONE_TABLE >> ( bit * ZONES_SHIFT ) ) &
        (( 1 << ZONES_SHIFT ) - 1 );
    VM_BUG_ON (( GFP_ZONE_BAD >> bit ) & 1 );
    return z ;
}
```

Implementation of this function also depends on `GFP_ZONE_TABLE`, `ZONES_SHIFT`, `GFP_ZONE_BAD` these macros , they are in `gfp.h` definition . Implement logic functions not specifically analyzed , looked a little burnt head .

However, the running result of this function is clear and clear , let me share with you :

Parameter `flags` is `__GFP_DMA`, `__GFP_DMA32`, `__GFP_MOVABLE` and `__GFP_HIGHMEM` combination , these . 4 values can be combined `0x0 - 0xf` this 16 Scenario , operation results in each case are as follows :

```
* bit result
* =====
* 0x0 => NORMAL
* 0x1 => DMA or NORMAL
* 0x2 => HIGHMEM or NORMAL
* 0x3 => BAD (DMA+HIGHMEM)
* 0x4 => DMA32 or DMA or NORMAL
* 0x5 => BAD (DMA+DMA32)
* 0x6 => BAD (HIGHMEM+DMA32)
* 0x7 => BAD (HIGHMEM+DMA32+DMA)
* 0x8 => NORMAL (MOVABLE+0)
* 0x9 => DMA or NORMAL (MOVABLE+DMA)
* 0xa => MOVABLE (Movable is valid only if HIGHMEM is set too)
* 0xb => BAD (MOVABLE+HIGHMEM+DMA)
* 0xc => DMA32 (MOVABLE+DMA32)
* 0xd => BAD (MOVABLE+DMA32+DMA)
* 0xe => BAD (MOVABLE+DMA32+HIGHMEM)
* 0xf => BAD (MOVABLE+DMA32+HIGHMEM+DMA)
*
```

It is noteworthy that , individually set `__GFP_MOVABLE` and not from the time `MOVABLE` domain allocating memory , unless you also set `__GFP_MOVABLE | __GFP_HIGHMEM` will try from `MOVABLE` memory allocation domain .

2. Other masks

In addition to memory domain modifiers, some flags can also be set in the mask. Contrary to memory domain modifiers, these additional flags do not limit the physical memory segment from which to allocate memory, but they can indeed change the behavior of the allocator.







```
//include/linux/gfp.h

...
#define __GFP_HIGH      0x20u
#define __GFP_IO 0x40u
#define __GFP_FS 0x80u
#define __GFP_COLD 0x100u
#define __GFP_NOWARN 0x200u
#define __GFP_REPEAT 0x400u
#define __GFP_NOFAIL 0x800u
#define __GFP_NORETRY 0x1000u
#define __GFP_MEMALLOC 0x2000u
#define __GFP_COMP 0x4000u
#define __GFP_ZERO 0x8000u
#define __GFP_NOMEMALLOC 0x10000u
#define __GFP_HARDWALL 0x20000u
#define __GFP_THISNODE 0x40000u
#define __GFP_ATOMIC 0x80000u
#define __GFP_NOACCOUNT 0x100000u
#define __GFP_NOTRACK 0x200000u
#define __GFP_DIRECT_RECLAIM 0x400000u
#define __GFP_OTHER_NODE 0x800000u
#define __GFP_WRITE 0x1000000u
#define __GFP_KSWAPD_RECLAIM 0x2000000u

#define __GFP_IO ((__force gfp_t) __GFP_IO)
#define __GFP_... ((__force gfp_t) __GFP_...) // Not listed one by one
```

Some of the constants given above are rarely used, so I will not discuss them. The semantics of some of the most important constants are shown below.

- If the request is very important, set `__GFP_HIGH` , that is urgently needed when the kernel memory. When the failure to allocate memory may bring serious consequences to the kernel (such as threatening system stability or system crash), this flag is always used. Note that it has nothing to do with `HIGHMEM`
- `__GFP_IO` indicates that the kernel can perform I/O during the search for free memory operations . In practice, this means that if the kernel swaps out pages during memory allocation, only when this flag is set, can the selected page be written to the hard disk.
- `__GFP_FS` allows the kernel to perform VFS operations. Must be disabled in the kernel subsystem connected with the VFS layer, because this may cause cyclic recursive calls

-   you need to allocate "cold" pages that are not in the CPU cache, set `__GFP_COLD`
- `__GFP_NOWARN` disables the kernel failure warning when the allocation fails. This sign is useful in rare occasions
- `__GFP_REPEAT` will automatically retry after the allocation fails, but it will stop after several attempts
- `__GFP_NOFAIL` keeps retrying after the allocation fails until it succeeds
- `__GFP_ZERO` will return the page with padding byte 0 when the allocation is successful
- `__GFP_HARDWALL` is only meaningful on NUMA systems. It restricts the allocation of memory only to the nodes associated with each CPU allocated to the current process. If the process is allowed to run on all CPUs (the default), this flag is meaningless. CPU that only processes can run. This flag has an effect only when on which the limited
- `__GFP_THISNODE` is also meaningful only on NUMA systems. If this bit is set, it is not allowed to use other nodes as a backup in the case of memory allocation failure, and it is necessary to ensure that the memory is successfully allocated on the current node or explicitly designated nodes
- `__GFP_ATOMIC` for dispensing atoms, can not be interrupted, in any case, can not SLEEP. Interrupt handler allocating memory commonly use this flag. The flag is often associated with `__GFP_HIGH` used together, the representative may use the emergency list of allocated memory
-     are some signs that are not introduced. Interested readers can read `gfp.h`, and the meaning of the comments in the source code is also very clear

Since these flags are almost always used in combination, the kernel has made some groupings, including appropriate flags for various standard situations. If possible, when requesting memory from the memory management subsystem, one of the following groups should be used as much as possible. In the source code kernel, double underlined lines are typically used for internal data and definitions. These predefined group names do not have a double underscore prefix.

```
//include/linux/gfp.h

#define GFP_ATOMIC (__GFP_HIGH|__GFP_ATOMIC|__GFP_KSWAPD_RECLAIM)
#define GFP_KERNEL (__GFP_RECLAIM | __GFP_IO | __GFP_FS)
#define GFP_NOWAIT (__GFP_KSWAPD_RECLAIM)
#define GFP_NOIO (__GFP_RECLAIM)
#define GFP_NOFS (__GFP_RECLAIM | __GFP_IO)
#define GFP_TEMPORARY (__GFP_RECLAIM | __GFP_IO | __GFP_FS | \
    __GFP_RECLAIMABLE)
#define GFP_USER (__GFP_RECLAIM | __GFP_IO | __GFP_FS | __GFP_HARDWALL)
#define GFP_DMA __GFP_DMA
#define GFP_DMA32 __GFP_DMA32
#define GFP_HIGHUSER (GFP_USER | __GFP_HIGHMEM)
#define GFP_HIGHUSER_MOVABLE (GFP_HIGHUSER | __GFP_MOVABLE)
#define GFP_TRANSHUGE ((GFP_HIGHUSER_MOVABLE | __GFP_COMP | \
    __GFP_NOMEMALLOC | __GFP_NORETRY | __GFP_NOWARN) & \
    ~__GFP_KSWAPD_RECLAIM)
```

- `GFP_ATOMIC` is used for atomic allocation and cannot be interrupted under any circumstances
- `GFP_KERNEL` and `GFP_USER` are the default settings of the kernel and user request memory respectively. The failure of both will not immediately threaten the stability of the system. `GFP_KERNEL` is definitely the most commonly used flag in the kernel source code.

- GFP_HIGHUSER is GFP_USER an extension, but also for user space. It allows allocation of high-end memory that cannot be directly mapped. There is no harm in using high memory pages, because the address space of user processes is always organized by non-linear page tables. The purpose of GFP_HIGHUSER_MOVABLE is similar to that of GFP_HIGHUSER , but the allocation will be performed from the virtual memory domain ZONE_MOVABLE .
- GFP_DMA is used to allocate memory suitable for DMA . It is currently a synonym for __GFP_DMA . GFP_DMA32 is also a synonym for __GFP_GMA32 .

2.4.6 Implementation details of page allocation (__alloc_pages)

If you look up the source code , you will find that the various APIs of the allocation page will eventually be called to the __alloc_pages function , and __alloc_pages will be called directly __alloc_pages_nodemask, The kernel source code __alloc_pages_nodemask called the "buddy system of the heart" , because it deals with substantial memory allocation .

The implementation of __alloc_pages_nodemask is in mm/page_alloc.c , and its declaration is as follows :

```
struct page *
__alloc_pages_nodemask ( gfp_t gfp_mask , unsigned int order ,
                        struct zonelist * zonelist , nodemask_t * nodemask )
```

- gfp_mask is the allocation mask , and the specific meaning has been explained in "Allocation Mask GFP_XXX"
- order is the order to be allocated
- zonelist is a list of memory domains , 2.2.2 " Zonelist " introduced its data structure . This memory domain list contains all the memory domains that can be used to allocate memory (including the memory domain of the current node and the memory domain of the standby node) . According to the priority The levels are arranged in order from high to low . The kernel will first try to allocate memory from the first zone in the list .

Give an example to illustrate the order of the list :

Assuming that the allocation mask is set to __GFP_HIGHMEM , which represents the allocation of memory from the HIGHMEM domain , the order of the list is [this node -> HIGHMEM domain ; this node -> NORMAL domain ; this node -> DMA domain ; standby node -> XXX domain]

Assuming that the allocation mask is set to __GFP_DMA, which means that memory is allocated from the DMA domain , the order of the list is [this node -> DMA domain ; standby node -> XXX domain]

2.3.4 `build_all_zonelists" is responsible for initializing the memory zone list .

- nodemask does not understand the specific role temporarily

The implementation details of this function are very complicated , especially when the system memory is insufficient , the function will try repeatedly through various means (such as waking up the page recovery / swap process , swapping out infrequently used pages to disk ; or enabling OMM killer, Kill infrequent processes ; etc.) . These details are not going to be discussed here for the time being , interested readers can read "In-depth Linux Kernel Architecture 3.5.5 Allocation Page" and study the kernel code .

Here we only focus on the simplest situation , that is, the system has sufficient memory and the code is successfully obtained memory . Through this simple situation , we can get a macro feel of the general flow of page allocation .

Streamline the situation

__alloc_pages_nodemask -> get_page_from_freelist .

get_page_from_freelist

```
// mm/page_alloc.c

static struct page *
get_page_from_freelist ( gfp_t gfp_mask , unsigned int order , int alloc_flags
,
                        const struct alloc_context * ac )
{
    ...
    for_each_zone_zonelist_nodemask ( zone , z , zonelist , ac -> high_zoneidx
, ac -> nodemask ) {
        ...
        mark = zone -> watermark [ alloc_flags & ALLOC_WMARK_MASK ];
        if ( ! zone_watermark_ok ( zone , order , mark ,
                                ac -> classzone_idx , alloc_flags ) ) {
            int ret ;

            /* Checked here to keep the fast path fast */
            BUILD_BUG_ON ( ALLOC_NO_WATERMARKS < NR_WMARK );
            if ( alloc_flags & ALLOC_NO_WATERMARKS )
                goto try_this_zone ;

            if ( zone_reclaim_mode == 0 ||
                ! zone_allows_reclaim ( ac -> preferred_zone , zone ) )
                continue ;

            ret = zone_reclaim ( zone , gfp_mask , order );
            switch ( ret ) {
            case ZONE_RECLAIM_NOSCAN :
                /* did not scan */
                continue ;
            case ZONE_RECLAIM_FULL :
                /* scanned but unreclaimable */
                continue ;
            default :
                /* did we reclaim enough */
                if ( zone_watermark_ok ( zone , order , mark ,
                                        ac -> classzone_idx , alloc_flags ) )
                    goto try_this_zone ;

                continue ;
            }
        }
    }
}
```

```

try_this_zone :
    page = buffered_rmqueue ( ac -> preferred_zone , zone , order ,
                              gfp_mask , alloc_flags , ac -> migratetype );

    ...

    return page ;
}
}
...
}

```

The for loop is for each memory domain (zone) :

- ✓ Call `zone_watermark_ok` to determine whether the memory domain has enough free memory . At this time, the memory domain watermark is considered . Read this function for details .
- ✓ If there is enough free memory , call `buffered_rmqueue` to allocate memory . If the allocation is successful, you will get a `page` structure .

buffered_rmqueue

Buffered_rmqueue is a hugely complex function , again here we just take a brief look :

```

// mm/page_alloc.c

static inline
struct page * buffered_rmqueue ( struct zone * preferred_zone ,
                                struct zone * zone , unsigned int order ,
                                gfp_t gfp_flags , int alloc_flags , int migratetype )
{
    ...

    if ( likely ( order == 0 ) ) {
        struct per_cpu_pages * pcp ;
        struct list_head * list ;

        local_lock_irqsave ( pa_lock , flags );
        pcp = & this_cpu_ptr ( zone -> pageset ) -> pcp ;
        list = & pcp -> lists [ migratetype ];
        if ( list_empty ( list ) ) {
            pcp -> count += rmqueue_bulk ( zone , 0 ,
                                           pcp -> batch , list ,
                                           migratetype , cold );
            if ( unlikely ( list_empty ( list ) ) )
                goto failed ;
        }

        if ( cold )
    }
}

```

```

        page = list_entry ( list -> prev , struct page , lru );
    else
        page = list_entry ( list -> next , struct page , lru );

    list_del ( & page -> lru );
    pcp -> count -;
} else {
    ...

    if ( ! page )
        page = __rmqueue ( zone , order , migratetype , gfp_flags );

    ...
}

```

An if/else divides this function into 2 parts :

- If order != 0, represents the distribution is more continuous pages , directly call `__rmqueue` application page to partner system
- If order = 0, represents the assignment of a single page . This place is very interesting , it does not apply directly to a partner system , but attempts from `pcp->lists` get an inside this pool . If the pool is empty , It will call `rmqueue_bulk` to fill the pool first , and then allocate from this pool . `rmqueue_bulk` will eventually call `__rmqueue` to request a page from the partner system , and fill the pool with the obtained page .

`pcp` is a per-CPU mean , cold / related pages thermal mechanism . When only one request , the kernel tries by means of a per-CPU processing cache acceleration request , this article does not discuss in depth `pcp` mechanism

In the first 4 chapters " `vmalloc & buddyinfo` in." We did an experiment , when a `vmalloc` allocated 8 Shi pages , `buddyinfo` output without any change . The reason is that `vmalloc` application page is page after page of the application , this time from `pcp->lists` The memory obtained inside , `pcp->lists` has been filled before , so memory is not allocated from the partner system at this time , so there is no change in the output of `buddyinfo` .

`__rmqueue`

The `__rmqueue` function acts as a gatekeeper into the core of the partner system :

```

// mm/page_alloc.c

static struct page * __rmqueue ( struct zone * zone , unsigned int order ,
                                int migratetype , gfp_t gfp_flags )
{
    struct page * page ;

    page = __rmqueue_smallest ( zone , order , migratetype );
    if ( unlikely ( ! page ) ) {
        if ( migratetype == MIGRATE_MOVABLE )

```

```

        page = __rmqueue_cma_fallback ( zone , order );

    if (! page )
        page = __rmqueue_fallback ( zone , order , migratetype );
}

trace_mm_page_alloc_zone_locked ( page , order , migratetype );
return page ;
}

```

According to the allocation order passed in, the memory area used to obtain the page, and the migration type, `__rmqueue_smallest` scans the list of pages until it finds the appropriate contiguous memory block. In doing so, you can split higher-level memory blocks according to the method described in "Introduction to Principles 2.4.1". If the specified migration list cannot meet the allocation request, call `__rmqueue_fallback` to try other migration lists as an emergency measure.

`__rmqueue_smallest`

The implementation of `__rmqueue_smallest` is not very long. Essentially, it consists of a loop that traverses the list of free pages of each specific migration type in the memory domain in increasing order until a suitable one is found.

```

// mm/page_alloc.c

static inline

struct page * __rmqueue_smallest ( struct zone * zone , unsigned int order ,
                                   int migratetype )
{
    unsigned int current_order ;
    struct free_area * area ;
    struct page * page ;

    /* Find a page of the appropriate size in the preferred list */
    for ( current_order = order ; current_order < MAX_ORDER ; ++ current_order ) {
        area = &( zone -> free_area [ current_order ] );
        if ( list_empty (& area -> free_list [ migratetype ]))
            continue ;

        page = list_entry ( area -> free_list [ migratetype ]. next ,
                            struct page , lru );
        list_del (& page -> lru );
        rmv_page_order ( page );
        area -> nr_free --;
        expand ( zone , page , order , current_order , area , migratetype );
        set_pcppage_migratetype ( page , migratetype );
        return page ;
    }
}

```

```

    }

    return NULL ;
}

```

The search starts from the item corresponding to the specified allocation level. The small memory area is useless because the allocated pages must be contiguous.

After selecting the distribution level , select a list of migration types of the distribution level .

The operation of checking whether there is a suitable memory block is very simple. If there is an element in the list, then it is available because it contains the required number of consecutive pages. Otherwise, the kernel will choose the next higher allocation order and perform a similar search.

After removing a memory block from the linked list with `list_del` , note that the `nr_free` member of struct `free_area` must be reduced by 1 (the output in `buddyinfo` will change at this time) . `rmv_page_order` is an auxiliary function that removes the `PG_buddy` bit from the page flag , indicating that the page is no longer included in the buddy system, and sets the private member of the struct `page` to 0 .

If the length of the memory block that needs to be allocated is less than the selected continuous page range, that is, if a block of memory is allocated from a higher allocation order because there is no suitable smaller memory block available, then the memory block must be split according to the principle of the buddy system Into small pieces. This is done through the `expand` function.

expand

Basically, that is to achieve the " 2.4.1 Introduce the principle," which describes the split method . Will not elaborate here , are interested can read the "in-depth Linux kernel architecture 3.5.5 section on expand analytical functions."

2.4.7 Implementation details of page release (__free_pages)

All free page APIs will eventually call the `__free_pages` function .

```

// mm/page_alloc.c

void __free_pages ( struct page * page , unsigned int order )
{
    if ( put_page_testzero ( page ) ) {
        if ( order == 0 )
            free_hot_cold_page ( page , false );
        else
            __free_pages_ok ( page , order );
    }
}

```

If it is a single page , it will not be returned to the partner system , but returned to the cold / hot page pool (`pcp-lists`) .

If it is multiple pages , call `__free_pages_ok`.

`__free_pages_ok` After some roundabout operations , it will eventually call `__free_one_page`. And their names are different, this function not only deal with the release of a single page, the page also handle complex release. `__free_one_page` is the basis of the memory release function. The relevant memory area is added to the appropriate `free_area` list in the partner system . When releasing the partner pair, this function merges it into a contiguous memory area and places it in the higher-order `free_area` list. If a further partner pair can be merged, then merge as well and move to a higher-level list. The process repeats continue until all could have been a partner of the merger . The details of this process are interested can read the "in-depth Linux kernel architecture 3.5.6 release page."

2.5 slab distributor

2.5.1 Background introduction

Every C programmer is familiar with `malloc` and its related functions in the C standard library. When most programs allocate several bytes of memory, these functions are often called.





The kernel must also allocate memory frequently, but cannot rely on the functions of the standard library. The buddy system described above supports page-by-page allocation of memory, but this unit is too large. If you need to allocate space for a 10 -character string, allocating a complete page of 4 KiB or more space is not only wasteful but also totally unacceptable. The obvious solution is to split the page into smaller units that can accommodate a large number of small objects.

To this end, a new management mechanism must be introduced, which will bring greater overhead to the kernel. In order to minimize the impact of this additional burden on system performance, the implementation of the management layer should be as compact as possible so as not to have a significant impact on the processor's cache and TLB . At the same time, the kernel must ensure the speed and efficiency of memory utilization. Not only Linux , but also similar UNIX and all other operating systems need to face this problem. After a certain period of time, some good or bad solutions have been proposed, which are explained in the general operating system literature, such as [Tan07] .

One such proposal, the so-called slab allocation, proved to be very efficient for many kinds of workloads. It was designed and implemented in Solaris 2.4 by Jeff Bonwick , an employee of Sun. Since he disclosed his method [Bon94], it is also possible to implement a version for Linux .

Providing small memory blocks is not the only task of the slab allocator. Due to its structural characteristics, it is also used as a cache, mainly for objects that are frequently allocated and released. By establishing a slab cache, the kernel can reserve some objects for subsequent use, even in the initialization state. For example, in order to manage the file system data associated with the process, the kernel must frequently generate new instances of `struct fs_struct` . The memory block occupied by this type of instance also needs to be reclaimed frequently (at the end of the process). In other words, the kernel tends to allocate and release memory blocks of `sizeof{fs_struct}` very regularly . The slab allocator saves the released memory block in an internal list and does not immediately return it to the partner system. When a request is made to allocate a new instance of this type of object, the most recently released memory block will be used. This has two advantages. First, since the kernel does not have to use the buddy system algorithm, the processing time will be shorter. Secondly, because the memory block is still "new", there is a high probability that it will still reside in the CPU cache.

The slab distributor has two further benefits.

- The operation of calling the partner system has a considerable impact on the system's data and instruction cache. The more the kernel wastes these resources, the less available these resources are to user space processes. The lighter slab allocator reduces calls to the partner system when possible, helping to prevent unwelcome cache "pollution".
-     the CPU cache also caches physical page frames by page size, if the data is stored in the page directly provided by the partner system, we usually start from the first address of the obtained memory every time, and the result is that we will start from the page Start reading and writing a certain section of memory repeatedly at the first address . This has a negative impact on the utilization of the CPU cache. Due to this address distribution, some cache lines are overused, while other lines are hardly used . Multi-processor systems may exacerbate this disadvantage, because different memory addresses may be transmitted on different buses. The above situation will cause some bus congestion, while other buses are almost unused.







Through slab coloring (slab coloring) , the slab allocator can control the starting position of the object in the physical page frame to achieve uniform cache utilization. Frequently used kernel objects are stored in the CPU cache, which is the effect we want. The previous note mentioned that from the perspective of the slab allocator, the cache and TLB of the partner system occupy a large amount, which is a negative effect. Because this will cause unimportant data to reside in the CPU cache and important data to be replaced into memory, this should obviously be prevented.

The term coloring is metaphorical. It has nothing to do with color, it just represents the specific offset that the starting address of the object in the slab needs to be moved in order to place the object in a different cache line.

2.5.2 Introduction to slob & slub

Although the slab distributor works well for many possible workloads, there are some situations where it cannot provide optimal performance. Such as tiny embedded systems, massively parallel systems equipped with a large amount of physical memory. In the second case, the large amount of metadata required by the slab allocator may become a problem: the developers claim that the slab data structure alone requires a lot of gigabytes of memory on a large system . For embedded systems, the code size and complexity of the slab allocator are too high.

To deal with such situations, during the development of kernel version 2.6 , two alternatives to the slab allocator were added.

-    slob allocator has been specially optimized to reduce the amount of code. It revolves around a simple linked list of memory blocks (slob is the abbreviation of simple linked list of block). When allocating memory, the same simple first adaptation algorithm is used.
The slob allocator has only about 600 lines of code, and the total amount of code is very small. In fact, it is not the most efficient distributor in terms of speed, and it is certainly not designed for large systems.
-    slub allocator tries to minimize the required memory overhead by packing page frames into groups and managing these groups through unused fields in the struct page . Readers have seen before that this will not simplify the definition of the structure, but slub provides better performance than slab on large computers , indicating that this is correct.

Since slab allocation is the default option for most kernel configurations, I will not discuss alternative allocators in detail. But there is a very important point that needs to be emphasized, the rest of the kernel does not need to pay attention to which allocator is selected by the bottom layer. All of the dispensers provided by API is the same. In fact, no matter slab , SLOB or SLUB , they both refer to the header file slab.h and implemented slab.h defined API . Which allocator is enabled when the kernel is compiled, and which allocator is used by the caller.

In addition to the standard API , the kernel encapsulates some more convenient functions. For example, `kcalloc` allocates memory for an array, and `kzalloc` allocates a memory area filled with byte 0 .

2.5.3 Principle of slab distribution

Concept introduction

Object : slab system main purpose is to allocate a small memory , the " small memory " is called an object (obj, later in this section are used herein obj refers to an object generation) .

Object cache : The collection of all objects is called the object cache (`kmem_cache`, later in this section will use `kmem_cache` to refer to an object cache) , you can understand it as a pool , every time you apply for a storage space to the slab system , it Just get a " small block of memory " from this pool and return it to the applicant .

However, before applying for storage space , the applicant must first create an object cache . When creating an object cache , the applicant only needs to provide two parameters : name and size.

name refers to the object name of the cache , but size is " small memory " size . For example, we have a structure `struct this_obj`: if we want to slab apply for the storage structure of the system , then we have to create for The object cache of `struct this_obj` , when it is created, the size parameter is generally `sizeof(struct this_obj)`. As for the object cache, this pool initially has multiple " small blocks of memory " , don't care about it , the slab system will automatically calculate it . And when it's in the pool " When "small memory " is insufficient , the slab system will automatically expand ; when there is free " small memory " in the pool When there is too much , the slab system will automatically shrink .

The " small block of memory " in the pool comes from the partner system . When creating the pool , the slab system will apply for one or more pages from the partner system according to the situation , and then the resulting pages will be divided into " small blocks of memory " for management . when the pond expansion , will also apply to the partners page system ; pond pool shrink and destroy the time , we will return to the partner system memory .

From the above description , do you find the characteristics of the object cache pool ? Object cache is for a specific object , a specific object cache , the size of the " small memory " in it is fixed . As mentioned in the above example , when After we create an object cache for `struct this_obj` , this cache can only be used to allocate memory for `struct this_obj` .

However, there are thousands of different types of objects , and their sizes are all different (think about how many struct structures are in the kernel code) , what to do ? The answer is simple , just create your own object cache for each object . Therefore slab system may be a number of objects in the cache , the cache objects are mounted on a list , the list head is slab a global variable defined in the system .

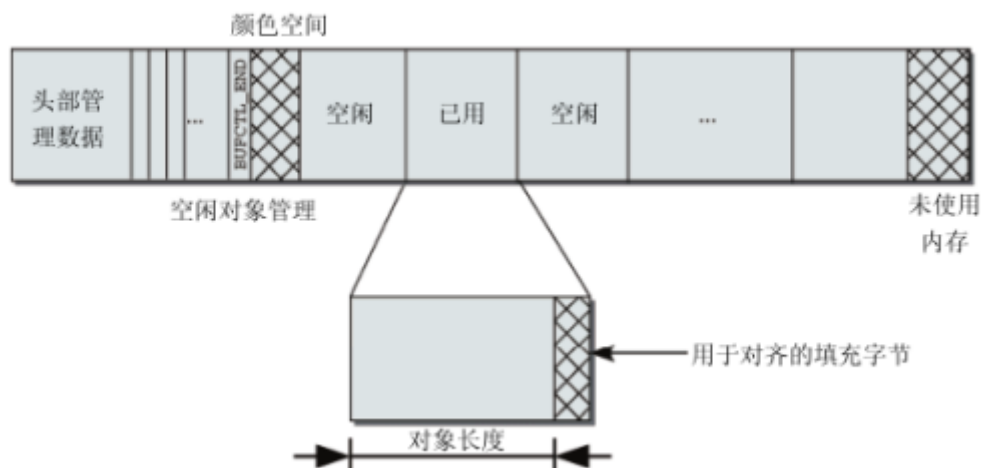
The object cache is a pool , but it is a large pool . This large pool has been further refined and divided into per-CPU cache and slab cache, These are two small ponds .

per-CPU cache : per-CPU cache is for the CPU, large pool inside How many per-CPU cache system depends on how many CPU. per-CPU cache stored inside those just been CPU visited obj (" small block memory ") . each time an application to this big pond inside obj time , slab system will favor from per-CPU get inside the cache , because

they might just be CPU visited , still CPU hardware cache inside , reallocate this memory to CPU, CPU can directly through cache access , quicker access . this is the slabOne of the advantages of the system .

slab cache : In the creation or expansion of a large pond , slab system will start with the partner system by the page frame , get pay-per-page frame is performed (for example, will get when you create once , each will get one expansion) , a few get page (possibly a , it may be a multi-page) is a slab system automatically determined . All pages each frame captured by a " slab cache " management , so big pond there are several slab cache , depending on the big pond to Buddy system memory apply several times . (Note that , due to the slab cache and slab on the system name has slab, in order to distinguish , this holiday paper slab cache on behalf of that small pond here described the system on behalf of that whole , slabslab distributor) .

Every time a page frame is requested from the partner system , a slab cache will be created . This slab cache will divide the obtained page frame into small objs for management . The details of a slab cache are as follows :



- * 头部管理数据 head management data can be divided into 2 arrays . The number of elements in each array depends on how many objs the slab cache divides the page into . The specific values are also determined by the slab system automatically .

The first array of purpose through it , we can quickly get obj addresses , in order to assign obj. Array of what is stored in each element of ? Best understanding is that obj start address . But it's not , actually is stored for each obj index , can be calculated by the kernel code index obj starting address , the details of the calculation will be described herein .

Why use an index ? Because if the number of small objects , such as the number of objects less than 256 Ge , then use a unsigned char representative index on the line ; that is, the number of objects a little more , unsigned short type is enough . If storage is For the starting address of the object , each address must occupy 4 bytes (sizeof(void *)).

The second purpose is to mark each array obj state , a state in two ways : OBJECT_ACTIVE, OBJECT_FREE. For indicating an idle state or in the end is the object has been assigned to the user .

These two arrays are next to each other , that is, consecutive in address .

The address of the header management data can be located on the page frame managed by the slab cache (the above figure is the case) , or a separate memory can be allocated to store the header management data (details will be described later) .

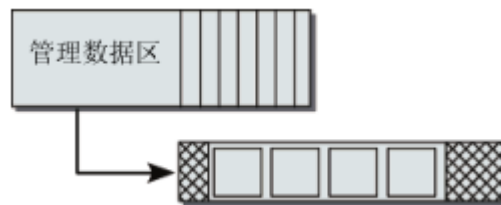
The head of data management on behalf of the slab cache , found the head of data management will find a corresponding slab cache . In the address stored in the header data management struct page -> freelist inside , this field before introducing the page data structure of time and No mention of it .

- **color space** is actually an offset . The purpose of the offset is to make the CPU hardware Cache fully utilized , instead of the CPU accessing the fixed line of the Cache every time . The section " 2.5.1 Background Introduction" describes this Reason .

Note that for different slab caches , the offsets here are different , and the purpose is to ensure that the Cache is fully utilized . If the offsets are the same , the CPU will still access certain fixed rows , but the number of rows has changed (for example, each time Will visit 1-5 rows , now 10-15 rows will be visited every time) .

- each obj size are aligned through the operation , so that each " small memory " size may be larger than the actual size of the object . For example sizeof (struct this_obj) is only 10 bytes , but when the memory each application , to give The size of the " small memory " is 12 bytes . The padding byte can speed up the access to the objects in the slab . If you use aligned addresses, memory accesses will be faster on almost all architectures. This makes up for the disadvantage that the use of padding bytes will inevitably lead to the need for more memory.

The storage space of the head management data can be allocated separately , the example diagram is as follows :



When assigning the individual , the address of the head of the management data is stored in the struct page -> freelist inside .

Finally , the kernel need for a method , can be identified by the object itself slab cache (and slab cache where the target cache) .

This is possible , by the physical address of the object can be positioned to the start of the physical page frame is located , by the page frame address to find the corresponding page (as described on page frames all page instance is stored in a linear array pg_data_t -> node_mem_map inside , the page frame address of the array can be used as the index , to find the corresponding page instance) . found page after instantiation , page-> freelist point slab cache , page-> slab_cache pointing object cache , page these two elements are in Initialized when creating object cache and slab cache .

Overview

Next , we use an icon to draw the internal structure of the slab system :

unsigned int batchcount	Specified in the per-CPU the buffer is empty case , every time the slab cache number aquired object
unsigned int limit	Specifies the per-CPU cache in Idle maximum number of objects , if the value is exceeded , the kernel will batchcount objects returned to the slab cache .
unsigned int shared	Unknown
unsigned int size	<p>5. The actual size of each object (" small block of memory ") in the object cache , which has been aligned .</p> <p>Assuming <code>object_size = sizeof(struct obj)</code>, then the size here is equal to <code>ALIGN (object_size, kmem_cache->align)</code>. <code>kmem_cache->align</code> will be introduced below . Generally speaking , after alignment , size will be greater than <code>object_size</code>.</p>
struct reciprocal_value reciprocal_buffer_size	<p>Assuming that an object is fetched into a slab cache , if the address of the object is known , how to determine the index of the object in the slab cache ?</p> <p>Since the objects are arranged in sequence in the slab cache , each object occupies <code>kmem_cache->size</code> bytes , the easiest way to calculate the partial index is to use the address of the object , minus the first object in the slab cache Start address, and then divide the obtained object offset by size to get the index value .</p> <p>For example, the address is an object 115, slab start address buffer is the first object 100, size size is 5, the value is the index of the object $(115 - 100) / 5 = 3$.</p> <p>But on some ancient computers , division is slower , and multiplication is much faster , so the kernel uses the so-called Newton-Raphson method , which only requires multiplication and shifting . Although for us , the mathematical details are not interesting (you can found in any standard textbooks) , but need to know , the kernel may not calculate $C = a / B$, instead of using $C = \text{reciprocal_divide}(a, \text{reciprocal_value}(B))$ in the embodiment , the two functions which are involved in the library Procedures .</p> <p>Because of the specific object in the cache , each object size is fixed , and the kernel will size the reciprocal value stored in <code>reciprocal_buffer_size</code> , in which the division value may be counted in subsequent use of the calculation .</p>
/* 2) touched by every alloc & free from the backend (buddy) */	

unsigned int flags	<p>The SLAB_Flags used in the slab system are defined as follows :</p> <p>Header file : include/linux/slab.h</p> <pre> #define SLAB_DEBUG_FREE 0x00000100UL /* DEBUG: Perform (expensive) checks on free */ #define SLAB_RED_ZONE 0x00000400UL /* DEBUG: Red zone objs in a cache */ #define SLAB_POISON 0x00000800UL /* DEBUG: Poison objects */ #define SLAB_HWCACHE_ALIGN 0x00002000UL /* Align objs on cache lines */ #define SLAB_CACHE_DMA 0x00004000UL /* Use GFP_DMA memory */ #define SLAB_STORE_USER 0x00010000UL /* DEBUG: Store the last owner for bug hunting */ #define SLAB_PANIC 0x00040000UL /* Panic if kmem_cache_create() fails */ #define SLAB_DESTROY_BY_RCU 0x00080000UL /* Defer freeing slabs to RCU */ #define SLAB_MEM_SPREAD 0x00100000UL /* Spread some memory over cpuset */ #define SLAB_TRACE 0x00200000UL /* Trace allocations and frees */ </pre> <p>There is also a FLAG defined in slab.c in :</p> <pre> #define CFLGS_OFF_SLAB (0x80000000UL) </pre> <p>This flag means that the slab cache management data in the end the head is stored in a slab external or internal cache (Concepts are discussed in the difference between them).</p>
unsigned int num	<p>When creating a new slab cache , num determines how many objs are divided into the cache .</p> <p>It also determines the number of elements in the array of head management data in the slab cache .</p>
/* 3) cache_grow/shrink */	
unsigned int gfporder	<p>When ^{creating} a new slab cache , apply for 2^{gfporder} pages from the partner system , and when reducing the object cache , return 2^{gfporder} pages to the partner system at a time</p>
gfp_t allocflags	Partner system-related distribution mask , see 2. 4 .5 Jie "distribution mask GFP_XXX "
size_t colour	<p>Each time a new slab when the cache , will be in the slab cache an object placed in the front section of " color space " , placing the object color space, see " 2.5.3 Concepts" .</p> <p>The size of the color space = (kmem_cache_node-> colour_next times the colour_off here).</p>
unsigned int colour_off	<p>colour_next represents a new next slab cache when " color value " , the color value from 0 Start increments by one, when (colour_next == herein colour time), colour_next and from 0 to start the cycle .</p> <p>In this way , the length of the " color space " in front of each slab cache is different , which can ensure that each cache line of the CPU hardware cache is used evenly</p>
struct kmem_cache *freelist_cache	<p>The head management data of the slab cache can be allocated storage space separately . If allocated separately , the storage space is obtained from the object cache pointed to by freelist_cache</p>

unsigned int freelist_size	The size of the slab cache header management data
void (*ctor)(void *obj)	Constructor , when creating and initializing a slab cache , the constructor is called once for each obj in the cache
/* 4) cache creation/removal */	
const char *name	The name of the object cache is unique in the entire slab system . The slab system is connected to multiple object caches , and different object caches are distinguished by name .
struct list_head list	Used to link the object cache to the global linked list of the slab system
int refcount	<p>Whenever a new object cache is created , refcount = 1, if an object cache with the same name is re-created next time , no actual creation action will be performed , only refcount + 1.</p> <p>When destroying an object cache , refcount--, the actual destruction action is executed only when refcount == 0 .</p>
int object_size	The actual size of each object in the cache object , refers to the size before the alignment operation , e.g. object_size = sizeof (struct this_obj)
int align	<p>Alignment value , used to align object_size to size. The calculation formula size=ALIGN(object_size, align).</p> <p>The value of align is divided into the following two situations :</p> <ul style="list-style-type: none"> ➤ If the flag SLAB_HWCACHE_ALIGN is used when creating the object cache , it will be aligned according to the return value of cache_line_size , which returns the processor-specific L1 cache size . If the object is less than half the length of the cache line , then a plurality of (e.g. nth) object into a cache line . That is to align cache_line_size / n ➤ not required if the cache line is aligned in hardware , the kernel objects by guaranteed BYTES_PER_WORD aligned , this value is a void required number of bytes the pointer 32 is the system bit 4 bytes .
/* 5) statistics */	
<pre>#ifdef CONFIG_DEBUG_SLAB ... #endif</pre>	
struct kmem_cache_node *node[MAX_NUMNODES]	<p>Each kmem_cache_node corresponds to a medium pool , which contains multiple small pools (slab cache) .</p> <p>System memory is the number of nodes , it corresponds to the number of middle-pond</p>

array_cache

Array_cache is used to describe a per-CPU cache , and each cpu corresponds to an array_cache. For simplicity , we assume that there is only one CPU in the system below .

Header file : mm/slab.c

struct array_cache	Comment
--------------------	---------

unsigned int avail	How many free objects are in the per-CPU cache . Every time an object is allocated , avail++ For each object added , avail--
unsigned int limit	Its value is equal to kmem_cache->limit, and its meaning is the same as the meaning of limit in kmem_cache . When the per-CPU number of idle cache object exceeds limit time , it returns batchcount object to the slab cache
unsigned int batchcount	Its value is equal to kmem_cache-> batchcount , and its meaning is the same as the meaning of batchcount in kmem_cache .
unsigned int touched	When the per-CPU removing an object cache , will be touched is set to 1 . And when the per-CPU cache when contracted , then touched is set to 0 . This allows the kernel to confirm whether the cache has been accessed since the last time it was shrunk , and is also a sign of the importance of the cache
void *entry[]	0- length array (zero- length array is a feature of GNU C , and its meaning can be found in the article ``Programming Basics"), and each array element points to the starting address of a free obj . When we want to get an object from per-CPU , the way to get it is void *obj = entry[-- avail]; when we release an object to the per-CPU cache , the way is entry[avail++] = obj. Noticed yet , when allocating and releasing objects , using last in first out of the original is (LIFO , Last in First OUT). Object Kernel assume just released is still in the CPU cache , will be as soon as next in response to an allocation request) reassigned it is , also aims to use cache properties faster access .

kmem_cache_node

kmem_cache_node is used to describe a collection of slab caches, and there may be multiple slab caches under a kmem_cache_node .

Each memory node (node) corresponds to a kmem_cache_node. For the sake of simplicity , we assume that there is only one node in the system .

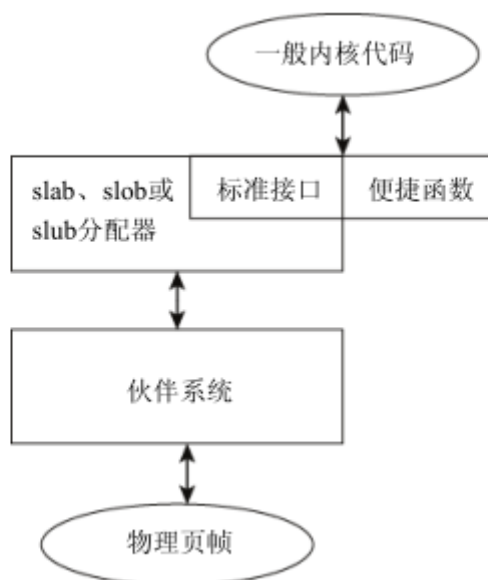
Header file : mm/slab.h

struct kmem_cache_node	Comment
spinlock_t list_lock	Atomic lock
struct list_head slabs_partial	Partially free linked list header , used to hook up all partially free slab caches . If part of a slab cache is used , and the other part is free , it is part of the free slab cache
struct list_head slabs_full	Used to mount all full slab caches . If all objects in a slab cache are used , it is called a full slab cache
struct list_head slabs_free	Used to mount all free slab caches . If all objects in a slab cache are not used , it is called a free slab cache
unsigned long free_objects	free_objects represents the total number of free objects in all slabs of slabs_partial and slabs_free
unsigned int free_limit	The maximum number of unused objects allowed on all slabs is specified , that is, if free_objects > free_limit, then the slab system will shrink and return a part of the memory to the partner system

unsigned int colour_next	The " color " of the next slab created by the kernel , see the description of colour and colour_off in struct kmem_cache for details
struct array_cache *shared	Per-CPU cache that can be shared within the node .
struct alien_cache **alien	Temporarily don't understand
unsigned long next_reap	Defines the time interval that the kernel must pass between two attempts to shrink the cache. The idea is to prevent system performance from degrading due to frequent cache shrinking and growth operations, which may occur under certain system loads. This technology is only used on NUMA systems and we will not pay further attention.
int free_touched	Indicates whether the slab cache is active. When fetching an object from the slab cache, the kernel sets the value of this variable to 1 . When the cache shrinks, the value is reset to 0 . This variable will be applied to the entire cache, so it is different from the per-CPU variable touched

2.5.4 APIs

Ordinary kernel code only needs to include slab.h to use all standard kernel functions for memory allocation. The following figure shows the approximate relationship between the slab- like distributor and the buddy system.



Header file : include/linux/slab.h

Slab/Slob/Slub API	Comment
kmem_cache_create (const char *name, size_t size, size_t align, unsigned long flags, void (*ctor)(void *))	<p>* kmem_cache_create – Create a cache.</p> <p>* @name: A string which is used in /proc/slabinfo to identify this cache.</p> <p>* @size: The size of objects to be created in this cache.</p> <p>* @align: The required alignment for the objects.</p> <p>* @flags: SLAB flags</p> <p>* @ctor: A constructor for the objects.</p> <p>Regarding SLAB flags, I introduced it when introducing the kmem_cache data structure.</p>

<pre>#define KMEM_CACHE (__struct, __flag s) kmem_cache_create(#__struct,\ sizeof(struct __struct), __ali gnof__(struct __struct),\ (__flags), NULL)</pre>	<p>Please use this macro to create slab caches. Simply specify the name of the structure and maybe some flags that are listed above .</p> <p>The alignment of the struct determines object alignment. If you f.e. add <code>___cacheline_aligned_in_smp</code> to the struct declaration then the objects will be properly aligned in SMP configurations.</p> <p>This is a kernel-defined macros , used to create a slab caches</p>
<code>kmem_cache_alloc (struct kmem_cache *, gfp_t flags)</code>	Allocate an object contained in it
<code>kmem_cache_free (struct kmem_cache *, void *)</code>	Release an object contained in it
<code>kmem_cache_zalloc (struct kmem_cache *, gfp_t flags)</code>	Assign and fill with 0
<code>kmem_cache_destroy (struct kmem_cache *)</code>	Destroy the created object cache
<p>Note: The above API is used to create an object cache , and then allocate / release objects from the object cache . The following API s do not require you to display the creation of the object cache , they will use the general object cache created by the slab system .</p>	
<code>kmalloc (size_t size, gfp_t flags)</code>	<p>Allocate a memory area with a length of size bytes, and return a pointer to the beginning of the memory area</p> <p>A void pointer. If there is not enough memory (this situation is unlikely in the kernel, but it must always be considered) , the result is a NULL pointer.</p> <p>flags parameter 2. . 4 .5 discussed in Section GFP_ XXX constant, allocate memory to specify particular memory field, e.g. GFP_DMA specify the allocation is adapted to DMA memory area.</p> <p>There are thousands of kmalloc used in the kernel source code, but the patterns are the same. The memory area allocated by kmalloc is first converted to the correct type through type conversion, and then assigned to a pointer variable.</p> <p>info = (struct cd_info *) kmalloc (sizeof (struct cd_info), GFP_KERNEL);</p>
<code>kfree (const void *)</code>	Release the memory area pointed to by *ptr
<code>kzalloc (size_t size, gfp_t flags)</code>	<p>Allocate a memory area filled with byte 0</p> <p>Its implementation borrows kmalloc, which is very simple and clear :</p> <p>return kmalloc(size, flags __GFP_ZERO)</p>
<code>kzfree (const void *)</code>	Release the memory allocated by kzalloc
<code>kcalloc (size_t n, size_t size, gfp_t flags)</code>	<p>Allocate memory for the array , a total of n elements , and the memory size of each element is size</p> <p>Note that the API will fill the allocated memory area with 0</p>
<code>kmalloc_node (size_t size, gfp_t flags, int node)</code>	Allocate a memory area specific to a node
To be added	

/proc/slabinfo

The list of all object caches is stored in /proc/slabinfo (in order to save space, the output below omits unimportant parts)

```
root@ubuntu:~# cat /proc/slabinfo
slabinfo - version: 2.1
# name      <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tnumobjs <limit> <batchcount> <sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedtotal>
kzfs_delayed_data_ref      0      0    96    42    1 : tnumobjs      0      0      0 : slabdata      0      0      0
.....
task_struct      192    120    1920    8    4 : tnumobjs      0      0      0 : slabdata     15     15      0
anon_vma      432    960    48    95    1 : tnumobjs      0      0      0 : slabdata     10     10      0
radix_tree_node      324    324    312    13    1 : tnumobjs      0      0      0 : slabdata     25     25      0
trace_event_file      2040    2040    48    95    1 : tnumobjs      0      0      0 : slabdata     24     24      0
trace_event_field      5405    6016    32    120    1 : tnumobjs      0      0      0 : slabdata     47     47      0
idr_layer_cache      267    270    1072    15    4 : tnumobjs      0      0      0 : slabdata     18     18      0
kmallo-8192      28     28    8192    4    0 : tnumobjs      0      0      0 : slabdata      7      7      0
kmallo-4096      15     16    4096    8    0 : tnumobjs      0      0      0 : slabdata      2      2      0
kmallo-2048      216    216    2048    8    4 : tnumobjs      0      0      0 : slabdata     27     27      0
kmallo-1024      158    168    1024    8    2 : tnumobjs      0      0      0 : slabdata     21     21      0
kmallo-512      663    736    512    8    1 : tnumobjs      0      0      0 : slabdata     92     92      0
kmallo-256      526    672    256    16    1 : tnumobjs      0      0      0 : slabdata     42     42      0
kmallo-128      711    819    128    31    1 : tnumobjs      0      0      0 : slabdata     39     39      0
kmallo-64      3993    4112    64    64    1 : tnumobjs      0      0      0 : slabdata     66     66      0
kmem_cache_node      11840    11840    64    64    1 : tnumobjs      0      0      0 : slabdata    182    182      0
kmem_cache      120     120     64     64    1 : tnumobjs      0      0      0 : slabdata      2      2      0
kmem_cache      195     195     192     21    1 : tnumobjs      0      0      0 : slabdata      5      5      0
```

The first column is the name of each object cache, and the following columns are the detailed information of each object cache .

In the API section, we mentioned that when using kmalloc to allocate memory , the object cache automatically created by the slab system is used . These object caches are kmalloc-8192, kmalloc-4096, ..., kmalloc-64. The slab system will be based on kmalloc(The size parameter of size_t size, gfp_t flags) determines which object cache is used .

2.5.5 Key code analysis

slab system initialization (kmem_cache_init)

At first glance, the initialization of the slab system is not particularly troublesome, because the partner system has been fully enabled and the kernel is not subject to other special restrictions. Nevertheless, due to the structure of the slab distributor, there is a chicken and egg problem.

To initialize the slab data structure, the kernel needs several memory blocks much smaller than a full page, and these are most suitable for allocation by kmalloc . Here is the key point: at this time the slab system has not been used, kmalloc cannot be used .

The kmem_cache_init function is used to initialize the slab allocator. It is called after the kernel initialization phase (start_kernel -> mm_init -> kmem_cache_init) and the partner system is enabled. kmem_cache_init uses a multi-step process to gradually activate the slab allocator to solve the above chicken and egg problem.

➤☞☐♦◆ ☐☒ all , when creating any object cache , we need to allocate the struct kmem_cache structure .

Therefore , the slab system defines a static object cache , and the obj of the object cache is struct kmem_cache. The definition of the object cache is as follows :

```
//mm/slab.c

static struct kmem_cache kmem_cache_boot = {
    . batchcount = 1 ,
    . limit = BOOT_CPUCACHE_ENTRIES ,
    . shared = 1 ,
    . size = sizeof ( struct kmem_cache ) ,
    . name = "kmem_cache" ,
};
```

`kmem_cache_init` first is called `create_boot_cache` initialize the object cache's name, size, object_size, align and other parameters .

Then `create_boot_cache` will further call `__kmem_cache_create` (note that `slab_state = NONE` at this time) to initialize the per-CPU cache and slab cache of the object cache .

After this operation is completed , we will be able to apply to the object cache allocation obj up .

- Subsequently , `kmem_cache_init` will create multiple `kmalloc_caches` object caches . After creation, these object caches will be initialized . These object caches are used for `kmalloc` allocation .

After the above steps are completed , we can use `kmalloc` to allocate storage space .

`kmem_cache_init` implemented in `mm / slab.c` in , we simply described under its general idea , there are many tips and details are not set forth herein , are interested can read the source code itself .

Create object cache (`kmem_cache_create`)

To create a new slab cache, you must call `kmem_cache_create` . This function requires many parameters.

```
//mm/slab_common.c

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

- name : The name of this object cache , which will then appear in `/proc/slabinfo`
- size: the length of the managed object in bytes
- align: the offset used when aligning data (align , it is 0 in almost all cases)
- flags: `SLAB_Flags`, detailed in " 2.5.3 `kmem_cache` "
- ctor: Constructor

`kmem_cache_create` code is relatively simple , as shown below :

`kmem_cache_create`

- `__kmem_cache_alias` : Check if the slab system already has an object cache with the same name , then the `refcount++` of the object cache , and then the object cache is returned to the caller
- `create_cache` : otherwise call `create_cache` to create a new object cache

`create_cache` first memory buffer assigned to the object space , and then its original name, size and other parameters , and then calls `__kmem_cache_create` , then the object will be cached `refcount` assigned the value 1, and then add it to the global cache for all objects attached Under the head of the linked list .

The implementation of `__kmem_cache_create` is a complex and lengthy process . The code flow chart is as follows :

`__kmem_cache_create` (`mm/slab.c`)

- Calculate alignment value
- `calculate_slab_order`: Calculate the two important data of `num` and `gfporder` of the object cache
- `calculate_freelist_size`: determining slab cache header size of data management unit , i.e. `freelist_size` value
- Determine the storage location of the head management data : `ON_SLAB` or `OFF_SLAB`
- Calculate the offset (`colour_off`) and the maximum color value (`colour`)

- If the head of data management is OFF_SLAB of , you need to create a header object cache management data to allocate space for storage , and then freelist_cache point to the object cache
- setup_cpu_cache: Initialize the per-CPU cache and slab cache of the object cache . Note that at this time , storage space is only allocated to the relevant data structure , and no memory is requested from the partner system , which means that there is no obj available in the cache

Details of the code here is not posted , in conjunction with 2.5.3 of knowledge and data structures section of the description of the individual elements , reading is not difficult to understand the code .

Allocation object (kmem_cache_alloc)

kmem_cache_alloc is used to obtain objects from a specific cache. Similar to all malloc functions, the result may be a pointer to the allocated memory area, or the allocation may fail and return a NULL pointer. This function requires two parameters: the cache used to obtain the object, and a flag variable that accurately describes the characteristics of the allocation.

```
//mm/slab.c

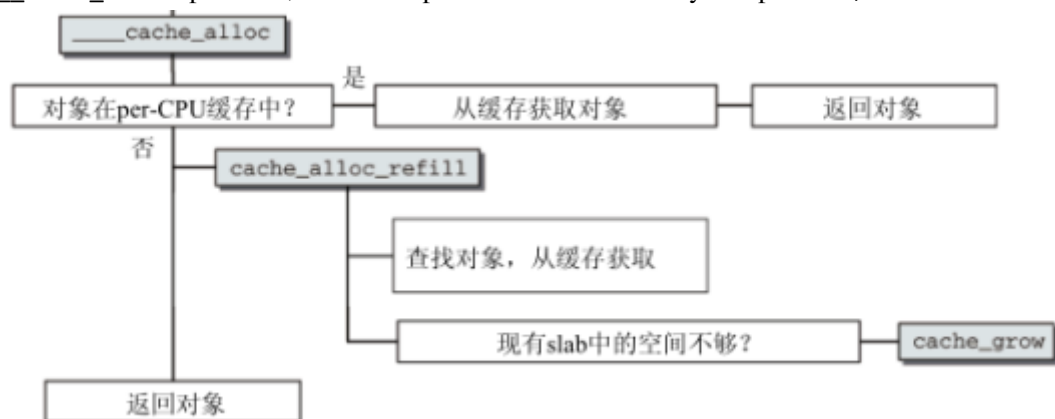
void * kmem_cache_alloc ( struct kmem_cache * cachep , gfp_t flags )
```

Have you noticed ? We did not specify GFP_XXX when creating the object cache flag , but only specified it here . This means that we only need to deal with the partner system at this time .

kmem_cache_alloc is based on the internal function slab_alloc with the same parameters , using this structure , the purpose of merging kmalloc and kmem_cache_alloc generic implementation section .

slab_alloc will call __do_cache_alloc , which further calls ____cache_alloc .

____cache_alloc is practical, and the implementation is also very complicated , as follows :



If there are free objects in the per-CPU cache, get them from them. But if all the objects in it have been allocated, the cache must be refilled. In the worst case, it may be necessary to create a new slab . Below we will elaborate on each situation .

There are free objects in the per-CPU cache

If in there are objects per-CPU cache, the `____cache_alloc` check is relatively easy, as shown in the following code snippet:

```
//mm/slab.c: ____cache_alloc

ac = cpu_cache_get ( cachep );
```

```

    if ( likely ( ac -> avail )) {
        ac -> touched = 1 ;
        objp = ac_get_obj ( cachep , ac , flags , false );
        ...
    }

```

`ac_get_obj` is used to get a obj and return its starting address :

```

//mm/slab.c

static inline void * ac_get_obj ( struct kmem_cache * cachep ,
                                struct array_cache * ac , gfp_t flags , bool force_refill )
{
    void * objp ;

    if ( unlikely ( sk_memalloc_socks ()))
        objp = __ac_get_obj ( cachep , ac , flags , force_refill );
    else
        objp = ac -> entry [- ac -> avail ] ;

    return objp ;
}

```

Red phrase , from entry to obtain a rearmost object address , and per-CPU number minus one idle object cache .
 From the dispensing end in order to achieve LIFO principle (the LIFO , Last in First OUT) , to improve access Speed .

per-CPU cache is not idle objects (cache_alloc_refill)

When there are no objects in the per-CPU cache, the workload will increase. The refill operation required in this case is implemented by `cache_alloc_refill` . This function is called when the per-CPU cache cannot directly satisfy the allocation request .

The kernel must now find `array_cache->batchcount` unused objects from the slab cache to refill the per-CPU cache.

First check the slab cache shared pointer release is empty , if not empty , then call `transfer_objects` function , try to share from the per-CPU transfer some of the cache obj to the current per-CPU cache .

If `transfer_objects` fails , then scan all parts of the free slab of the list (`slabs_partial`) , then `ac_put_obj` successively the discharge objects to the per-CPU cache , until the respective slab until there is no free objects.

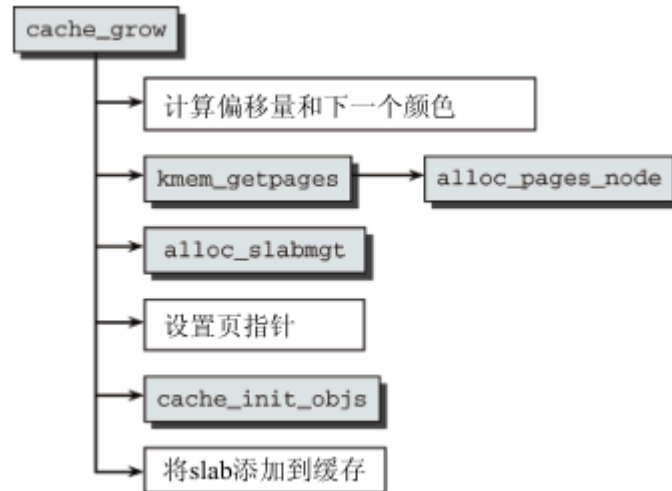
If you still can not find the required number of objects, the kernel will traverse `slabs_free` the list of all unused slab . When obtaining objects from the slab , the kernel must also place the slab in the correct slab linked list (`slabs_full` or `slabs_partial` , depending on whether the slab has been completely used up or still contains some free objects) .

If all slabs are scanned and no free objects are found, then `cache_grow` must be used to expand the cache. This is a costly operation and will be described in the next section.

The code details of `cache_alloc_refill` will not be posted , please read it yourself .

Cache expansion (`cache_grow`)

The following figure shows the code flow chart of `cache_grow` .



The parameters of `kmem_cache_alloc` are also passed to `cache_grow` . You can also specify a node explicitly for allocating new memory pages from it.

First calculate the color and offset:

```
//mm/slab.c: cache_grow

spin_lock (& n -> list_lock );

/* Get colour for the slab, and cal the next value. */
offset = n -> colour_next ;
n -> colour_next ++;
if ( n -> colour_next >= cachep -> colour )
    n -> colour_next = 0 ;
spin_unlock (& n -> list_lock );

offset *= cachep -> colour_off ;
```

If the maximum number of colors is reached, the kernel restarts counting from 0 , which automatically causes a zero offset.

The required memory space is allocated page by page from the partner system using the `kmem_getpages` helper function. The only purpose of this function is to call the `alloc_pages_node` function discussed in section 2.4.5 with appropriate parameters . The `PG_slab` flag bit is set on each page , indicating that the page belongs to the slab allocator. When a slab is used to satisfy short-term or reclaimable allocation, the flag `__GFP_RECLAIMABLE` is passed to the partner system. Recalling the content of section 2.4.5 , we know that the important thing is to allocate pages from the appropriate migration list.

The distribution of slab header management data is not interesting. If the slab header is stored outside the slab , the related alloc_slabmgmt function is called to allocate the required space. Otherwise, the corresponding space has been allocated in the slab . In both cases, the following two values will be set :

```
page -> active = 0 ;
page -> s_mem = addr + colour_off ;
```

page-> active variable to store the slab cache, how many obj has been assigned , this time apparently 0

page->s_mem stores the starting address of the first obj in the slab cache

Next, the kernel calls slab_map_pages create a page frame and object caching and slab associations between cache : First, initialize page-> slab_cache , to point to the current object cache ; then initialize page-> freelist, to point to the address of the head of data management , Which points to the slab cache .

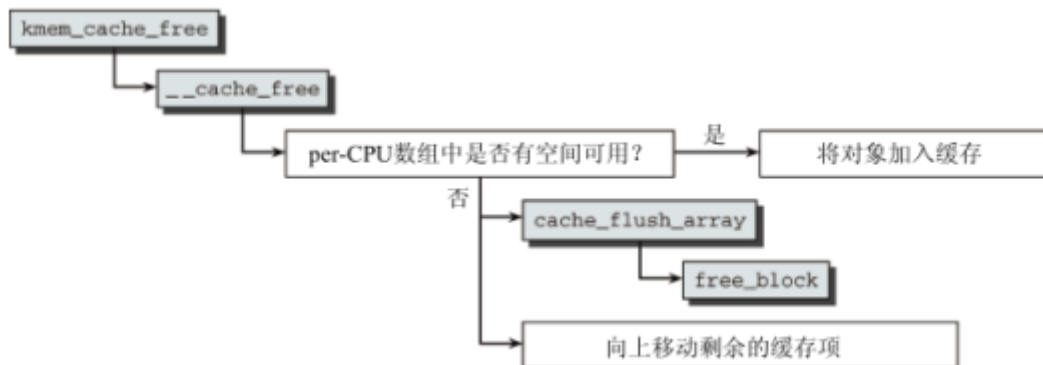
Cache_init_objs calls the constructor function (if any) of each object to initialize the objects in the new slab cache, but in reality, few people will define the constructor. In addition cache_init_objs also initializes the head of management data that 2 arrays.

Now that the slab has been fully initialized, it can be added to the cached slabs_free list. The number of newly generated objects is also added to the number of free objects in the cache (kmem_cache_node ->free_objects)

Free objects (kmem_cache_free)

If an allocated object is no longer needed, it must be returned to the slab allocator using kmem_cache_free . The following figure shows the code flow chart of this function.

kmem_cache_free immediately called __cache_free , and the parameters were passed directly. The reason is to prevent duplication of code in kfree .



Similar to allocation, there are two optional operating procedures according to the status of the per-CPU cache.

- If the number of free objects in the per-CPU cache is lower than the allowable limit, call ac_put_obj to store a pointer to the object in the cache.

```
//mm/slab.c

static inline void ac_put_obj ( struct kmem_cache * cachep , struct
array_cache * ac ,
                                void * objp )
{
    if ( unlikely ( sk_memalloc_socks ( )))
```



```

        objp = __ac_put_obj ( cachep , ac , objp );

        ac -> entry [ ac -> avail ++] = objp ;
    }

```

- Otherwise, you must be some objects (the exact number of array_cache-> batchcount given) move from the cache back slab , starting with the lowest number of array elements: cache implementation according to the FIFO principle, these objects have been long in the array Time, so it's unlikely to still reside CPU cache. The specific implementation is entrusted to cache_flusharray . This function calls free_block again , moves the object from the cache to the original slab , and moves the remaining objects to the beginning of the array. For example, if there is space for 30 objects in the cache and the batchcount is 15 , the objects at positions 0 to 14 will be moved back to the slab . The remaining objects numbered 15 to 29 move up in the cache and now occupy positions 0 to 14 .



Exception: The number of free objects in the slab cache exceeds the predefined limit cachep->free_limit . In this case, use slab_destroy to return the entire slab to the partner system.

If the slab contains both used and unused objects, they are inserted into slabs_partial linked list.

Destroy the object cache (kmem_cache_destroy)

If you want to destroy a cache containing only unused objects, you must call the kmem_cache_destroy function. This function is mainly called when the module is deleted. At this time, all allocated memory needs to be released. Of course this is not mandatory. If the module needs to obtain persistent memory and save it after unloading until the next time it is loaded (of course, it needs to be assumed that the system is not restarted during this period), it can retain a cache so that the data can be reused.

Since there is nothing new in the implementation of this function, below we just outline the main steps to delete the cache.

- kmem_cache -> refcount --; if refcount == 0, continue the following actions , otherwise exit and exit without destroying the current object cache .
- calls shutdown_cache destruction of real action
 - __kmem_cache_shutdown
 - ◆ release per-CPU cache
 - ◆   each memory node , release all slabs under the node caches

General cache

If you do not involve object caching, but allocate / release memory in the traditional sense , you must call kmalloc and kfree functions. These two functions are equivalent to the kernel equivalents of the C standard library malloc and free functions in user space .

kmalloc and kfree are implemented as the front end of the slab allocator, and their semantics imitate malloc/free as much as possible . Therefore, we will only briefly discuss its implementation.

Implementation of kmalloc

The basis of kmalloc's implementation is an array , and each element of the array represents an object cache :

```
//mm/slab_common.c
```

```
struct kmem_cache * kmalloc_caches [ KMALLOC_SHIFT_HIGH + 1 ]
```

This array is initialized in the `kmem_cache_init` phase .

The statement of `kmalloc` is as follows :

```
static __always_inline void * kmalloc ( size_t size , gfp_t flags )
```

Every time we call `kmalloc` to allocate a piece of memory , the slab system will select a suitable object cache from the above array according to the size , and then allocate an object from the object cache .

The function to select the object cache is as follows :

`kmalloc` -> `__kmalloc` -> `__do_kmalloc` -> `kmalloc_slab` .

```
//mm/slab_common.c
```

```
struct kmem_cache * kmalloc_slab ( size_t size , gfp_t flags )
```

```
{
    int index ;

    if ( unlikely ( size > KMALLOC_MAX_SIZE ) ) {
        WARN_ON_ONCE (!( flags & __GFP_NOWARN ));
        return NULL ;
    }

    if ( size <= 192 ) {
        if (! size )
            return ZERO_SIZE_PTR ;

        index = size_index [ size_index_elem ( size )];
    } else
        index = fls ( size - 1 );
```

```
#ifdef CONFIG_ZONE_DMA
```

```
    if ( unlikely (( flags & GFP_DMA )) )
        return kmalloc_dma_caches [ index ];
```

```
#endif
```

```
    return kmalloc_caches [ index ] ;
```

```
}
```

The code is not explained , when selecting the appropriate object cache , caches calling for the object `slab_alloc` function , previously "assignment (`kmem_cache_alloc`)" one introduced this function , there is not much to say .

Implementation of `kfree`

`kfree` is also easy to implement:

```
//mm/slab.c
```

```

void kfree ( const void * objp )
{
    struct kmem_cache * c ;
    unsigned long flags ;

    trace_kfree ( _RET_IP_ , objp );

    if ( unlikely ( ZERO_OR_NULL_PTR ( objp )))
        return ;
    local_irq_save ( flags );
    kfree_debugcheck ( objp );
    c = virt_to_cache ( objp );
    debug_check_no_locks_freed ( objp , c -> object_size );

    debug_check_no_obj_freed ( objp , c -> object_size );
    __cache_free ( c , ( void *) objp , _RET_IP_ );
    local_irq_restore ( flags );
}

```

Get through the object object cache address , and then call the cache for the object __cache_free function , earlier "release objects (kmem_cache_free)" one introduced the function , not much to say .