

SIXUENET

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

MEMORY MANAGEMENT (1): CONCEPT INTRODUCTION

📅 April 20, 2019 ([Http://Www.Mysixue.Com/?P=110](http://Www.Mysixue.Com/?P=110)) 👤 JL
([Http://Www.Mysixue.Com/?Author=1](http://Www.Mysixue.Com/?Author=1)) 💬

1 Concept introduction

1.1 Address & Address Space

Suppose we have a board , CPU is ARM 's 32 -bit processor , onboard DDRAM bit 2GB. Against this background , we will discuss some basic concepts .

1.1.1 Physical address & physical addressing & physical address space

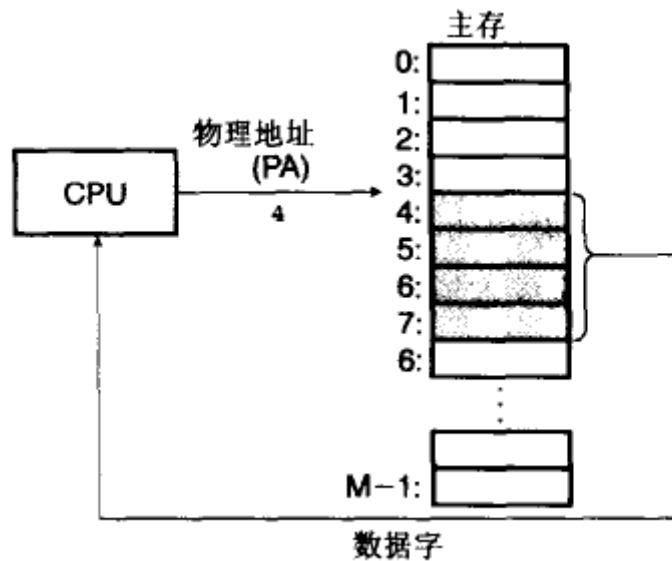
Physical address

In the above context , DDRAM can be appreciated by the $M (2 * 1024 * 1024 * 1024)$ an array of consecutive bytes of the size of the units . Each byte has a unique physical address (PHYSICAL address, PA) . Suppose The address of the first byte is 0, the address of the second byte is 1, and so on .

Physical addressing

CPU most natural way is to use the physical address to access a memory , for example, when we write code bare metal , is used in this way , we call this approach a physical address (PHYSICAL Addressing) .

The following figure shows an example of physical addressing . The context of this example is a load instruction that reads a word starting from physical address 4 .



When the CPU execution of this load instruction , it will generate a valid physical address , via a memory bus , pass it to the main memory . Main memory read from the physical address 4 a word beginning , and returns it to the CPU, CPU Will store it in a register .

Physical address space

Address space (address Space ordered set) non-negative integer address: $\{0, 1, 2, \dots\}$

A size of the address space is represented by the maximum number of bits required to address described . For example , contains $N = 2^n$ th address is called the address space of n -bit address space , its size is n.

Physical address space (PHYSICAL Space address is set) All physical address . It is the system physical memory with M number corresponding bytes: $\{0, 1, 2, \dots, M\}$

M is not required to be 2 of the power of power , but in order to simplify the discussion , we assume that $M = 2^m$. And so most of the real world .

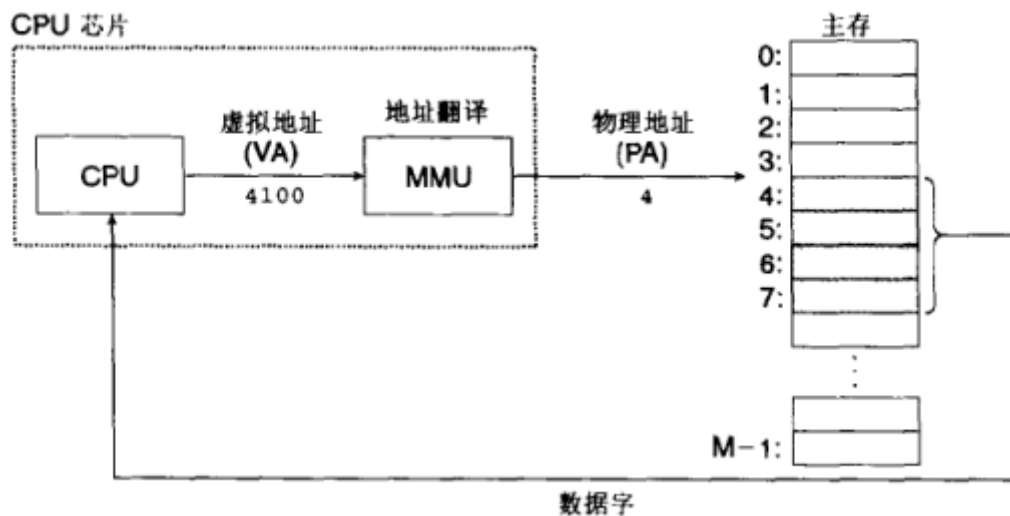
The size of the physical address space depends on the actual physical memory available in the system . In the above example , $2G = 2^{31}$, so the size of the physical address space is 31 bits .

1.1.2 Virtual Address & Virtual Addressing & Virtual Address Space

Virtual address / addressing

Bare-metal programs generally use physical addressing , but most operating systems use virtual addressing . All programs use virtual address (VA) when compiling and linking . When the CPU accesses virtual addresses, it uses the help of MMU , Convert the virtual address to a physical address , and then transfer it to the physical memory .

The following figure shows an example of virtual addressing :



CPU reads and parses the load instruction , to give a virtual address . CPU passes the virtual address to MMU, MMU to convert virtual addresses into physical addresses , and then transferred to the main memory . Conversion process is called address translation (address Translation) .

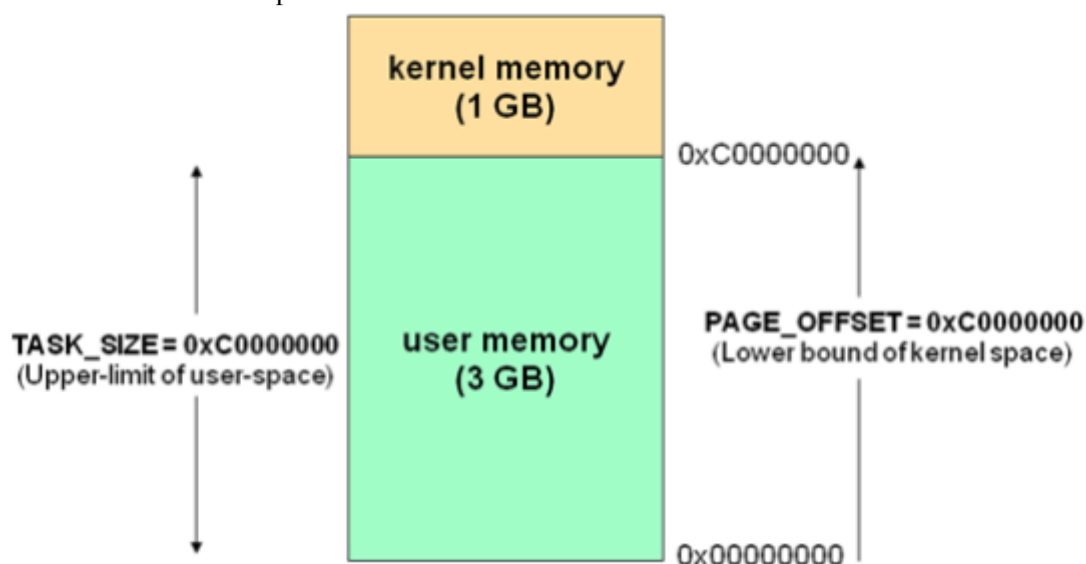
Address Translation CPU close cooperation between the hardware and the operating system . CPU on a chip the MMU (Memory Management Unit, memory management unit) , utilizing stored in the main memory lookup table to dynamically translate the virtual address , the scientific name of the table is called " Page table " , its content is managed by the operating system .

Virtual address space

Virtual address space (virtual address space) is the collection of all virtual addresses .

The size of the virtual address space has nothing to do with the actual physical memory available in the system , but is related to the architecture of the CPU . Generally we say 32 -bit CPU and 64 -bit CPU , the virtual address space size of 32 -bit CPU is 2^{32} , 64 bits The virtual address space of the CPU is 2^{64}

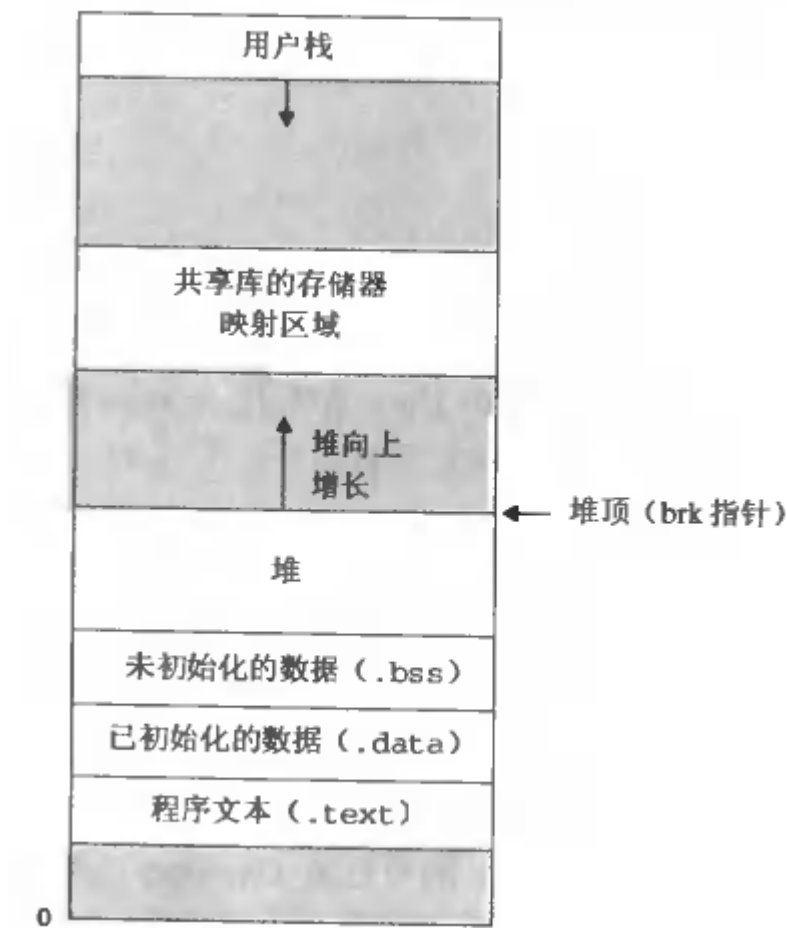
For 32 -bit CPUs, the virtual address space of the Linux operating system is divided into user process virtual address space and kernel virtual address space .



User process virtual address space

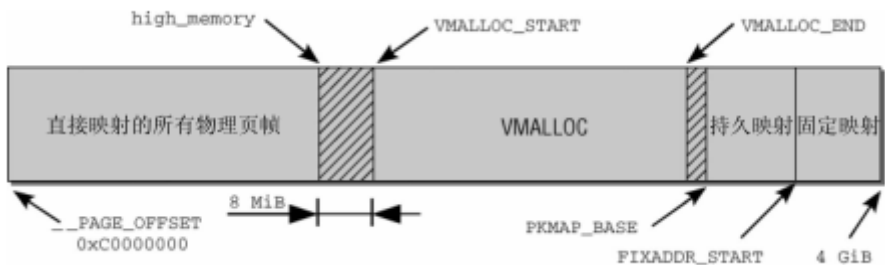
Each user process in the system has its own virtual address range , from 0 to TASK_SIZE . The area above the user space (from PAGE_OFFSET to 2³²) is reserved for the kernel and cannot be accessed by user processes . This division is the same as that available in the system The amount of physical memory is irrelevant .

All user processes when compiling links , both to the user process virtual address space-based . Snippet user process, read-only data segment, read / write data segment, stack, stack are located within the virtual space of the range . General As follows :



Kernel virtual address space (logical / virtual address)

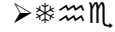
Kernel code compiled and linked stage , is the kernel virtual address space basis . Distribution kernel virtual address space is as follows :



Core logical address

The kernel virtual address space segment represented by " all physical page frames directly mapped " in the above figure is the kernel logical address .

The main characteristics of the kernel logical address are :

- There is a one-to-one mapping relationship between it and the physical address , and there is a fixed offset between the two . Therefore, the logical address is also called a linear address .
-  physical address corresponding to the area with continuous logical addresses is also continuous , which is different from the virtual address space of the user process . For the continuous area of the virtual address space of the user process , the corresponding physical address is not necessarily continuous, and may be scattered and discontinuous The physical page frame .

Note that the logical / linear address itself is a virtual address , which is part of the kernel virtual address space , starting from `__PAGE_OFFSET` (`0xc0000000`) .

In the 32 -bit architecture architectures , the kernel virtual address space maximum is only 1GB (`0xc0000000` - `0xffffffff`) . If this 1GB of virtual address space for all linear mapping , the kernel can only lead to physical access to the system memory of a fixed In the 1GB range , other physical memory cannot be accessed .

E.g. 32 bit CPU onboard 2GB of physical memory , assuming physical memory 0 address is mapped to `0xc0000000`, then only access kernel physical memory 0-1GB range , can not access 1GB-2GB interval .

To solve this problem , the introduction of **high-end memory** (`high_memory` concept) is , the kernel virtual address space and a physical address space is divided :

- First , kernel virtual address space into `__PAGE_OFFSET` - `high_memory` and `high_memory` - `0xffffffff`: the former for-one mapping , called a core logic address ; the latter for non-linear mapping , called the kernel virtual address . Logical address only A certain fixed interval of physical memory can be accessed , and the corresponding physical memory interval is continuous ; the virtual address can be dynamically mapped to any physical page , so that the kernel can access all physical pages .
- Secondly , the physical address space is divided into `0` - `high_memory` and `high_memory` - `MAX_MEMORY`: the former one mapped to a logical address into the kernel ; the latter for non-linear mapping , can be mapped to kernel virtual address , you may be mapped to the user process virtual Address space .

Kernel virtual address

Combining the aforementioned concepts , the kernel virtual address is easy to understand .

It is similar to the virtual address of the user process , both of which are dynamically mapped to the physical address through the page table , and the corresponding physical memory is not necessarily continuous .

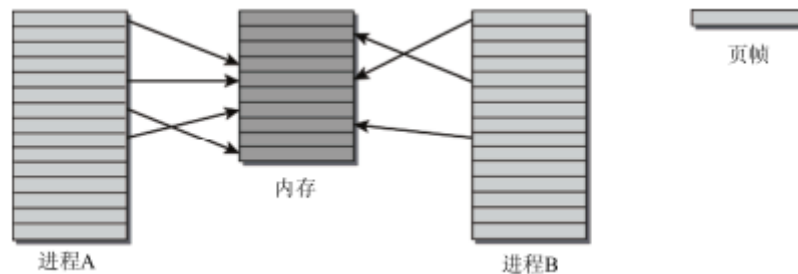
1.2 Address translation

In section 1.1 , we introduced the related concepts of virtual addresses and physical addresses . Virtual addresses must eventually be converted to physical addresses to obtain memory data correctly . This section mainly introduces the concepts related to the address translation process .

1.3.1 Page (Page)

In order to facilitate management , we divide the virtual address space into many blocks of equal length, and each block is called a page . The physical memory is also divided into pages of the same size . The size of each page can be determined by itself , in the ARM architecture In Linux systems , the page size is generally 4KB.

The so-called address mapping is to a virtual address space mapped to a physical address space , a simple diagram is as follows :



Virtual address space page generally called page , the physical address space of the page , in order to distinguish , commonly called page frames .

The concept is very important page , the entire memory management is basically around the page spreads of : How to get a physical page frame ; how to map a virtual page to a physical page frame ; how missing page exception handling ; how to deal with the page swap when memory is tight , the temporary physical page frame not need to use a swap to disk , time and other needs in return for ; and so on . these details we will be gradually introduced below .

1.3.2 Page Table (Page Table)

Assuming a page size is 4KB, a 32-bit virtual address can be divided into two parts :

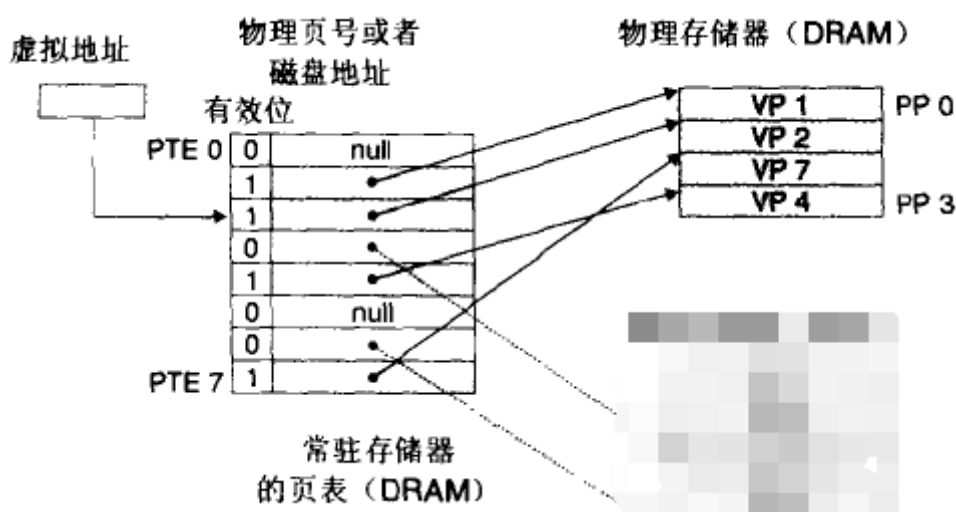
- [0 - . 11] : Total 12 is th bit, the page offset representatives
- [12 - 31] : a total of 20 Ge bit, on behalf of the index page

Address translation process is , first of all find the corresponding physical page by page index , and then add the offset within the page , you can get from a byte of physical memory to the data .

So how to find the corresponding physical page through the page index ? At this time, the page table is needed . We can think of the page table as an array PTE[N_Pages], the page index is the array id, and PTE[id] can be found by id , And the address of the physical page is stored in PTE[id] .

Refinement about the translation process , it has become such : When the CPU After getting any virtual address , known Page Size under the circumstances , it is easy to know the index page , find the page by page table index , from the page table The corresponding location obtains the address of the physical page , and then sends the physical page address + page offset to the memory bus , and the memory data can be obtained .

The general process is as follows :



The above process is actually completed by the MMU hardware , so it is very fast .

Operating system for each user-space process maintains a separate page table , stored in `task_struct-> mm_struct-> pgd` in . When the process of switching , will the process corresponding to the page table into a DRAM, then the page table The position in the DRAM tells the MMU.

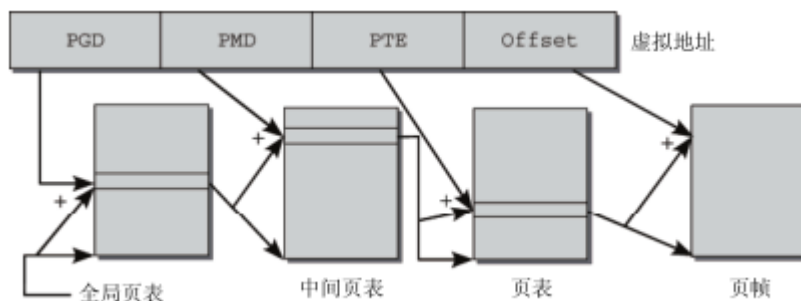
If you open the MMU, then the CPU when accessing any address , will be this address to the MMU, MMU through the page table lookup a physical address , to complete the address translation , the results of conversion (physical address) is transmitted to the memory bus , and memory Return the data of the corresponding position to the CPU.

1.3.3 Multi-level page tables : reduce page table storage space

According 1.2.2 case described section , assuming the page size is 4KB, for 32 -bit systems , the virtual address space will be divided into $4\text{GB} / 4\text{KB} = 1\text{MB}$ a page , if a page table corresponding to each page of , each The item requires 4 bytes of space , so 4MB of memory space is required when the entire page table is loaded into memory .

However , the vast majority of applications only use a small portion of the virtual address space , such as a helloworld program , you may only need 2 pages can be buttoned . In this case , the page table occupied by a large physical memory on It seems wasteful and unreasonable .

The typical solution is to use a multi-level page table : divide the virtual address into multiple parts , as shown in the following figure :



- The first part of the virtual address is called the Global Page Directory (Page Global Directory , PGD) . PGD is used to index an array in the process (each process has one and only one) , the array is the so-called global page directory or PGD . The PGD array item points to the starting address of some other arrays, these arrays are called the middle page directory (Page Middle Directory , PMD)
- virtual address in the second portion is called PMD , by PGD array locate the entry corresponding to the PMD after use PMD to index PMD . PMD array items are also pointers, pointing to the next level of array, called page table or page directory
- The third part of the virtual address is called PTE (Page Table Entry , page table array) , which is used as the index of the page table. The mapping between the virtual memory page and the page frame is completed, because the array item of the page table points to the page frame
- 最后部分 of the virtual address is called the offset. It specifies a byte position within the page . In the final analysis, each address points to a uniquely defined byte in the address space

Why multi-level page table memory space can save it ? Multistage a page table that featured , virtual address space in unwanted areas , without having to create an intermediate page directory or page table . Such physical memory only (global page table + Used intermediate page table + used page table) , those unused intermediate page tables and page tables will not exist in memory , so a lot of memory can be saved

1.3.4 TLB: Speed up address translation

Address Translation is a very, very frequent action , and therefore its processing speed is very important , will affect the overall system performance . If the page table is stored in DRAM in , each translation , MMU must proceed from DRAM to read the page table . Although compared For disks , the access speed of DRAM is much faster , but compared to the computing speed of the CPU , the access speed of DRAM has become a bottleneck .

In order to solve such problems , the usual approach is to increase the cache (Cache), the usual CPU is data cache (D-Cache) and instruction cache (I-Cache) . In addition , there are other multi-level caches , L1 L2 Cache. the closer the CPU storage device , access faster , the cost is higher .

MMU also has its own Cache, called the TLB. MMU can put frequently accessed page table is loaded into the TLB, when each such translation , you can directly from the TLB read the page table , do not always have access to DRAM, thus speeding up Processing speed .

1.3 Virtual page status

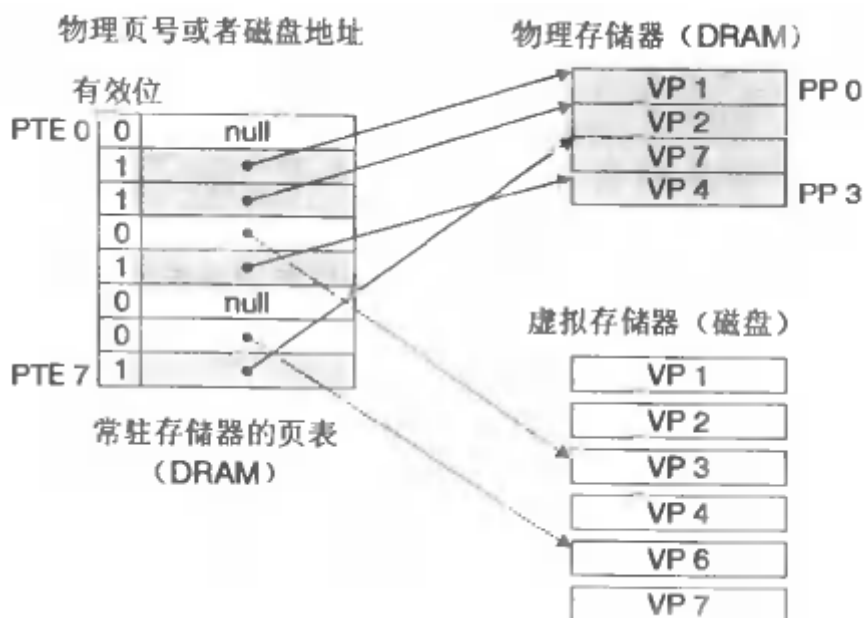
At any moment , the set of virtual pages is divided into three disjoint subsets :

- Unassigned : The virtual page is not used , corresponding to the data stored in the page table entry is null
- has not allocated cache : the virtual page has been used , but the content of the page is not cached in the DRAM in , but exists on disk . Data corresponding page table entry address stored on disk
- cached : The virtual page is already being used , and the content has been cached in the DRAM in , corresponding to the data stored in the page table entry for the DRAM physical address

For uncached and cached pages , their corresponding page table entries have addresses stored . How do we know whether this address is an address on the disk or a physical address of DRAM ?

A page table entry typically occupies 4 bytes of storage space , which is 32bit. Since the memory system is divided into one page , so page table entry in the stored data is page aligned (. $1 \ll \text{PAGE_SIZE}$) , so 32bit lower bits Both are 0. We can borrow these bits to indicate the status of the page .

For example, we can use the lower 1bit as the valid bit to indicate whether the page is cached : 0 means not cached , 1 means cached .



For any given virtual address , MMU find the corresponding page table entry through the index , by looking at the significant bits of the page table entry , to determine in the end is a disk or a page table entry address stored in DRAM physical address .


Lower page table entry is not used , in addition to be used as an effective bit (valid bit only takes 1bit) , but also many other purposes as , for example, marking the page is readable / writable, executable , and the like . These The logo can play a role in storage protection and prevent incorrect access or malicious tampering of the page .

1.4.1 page hit

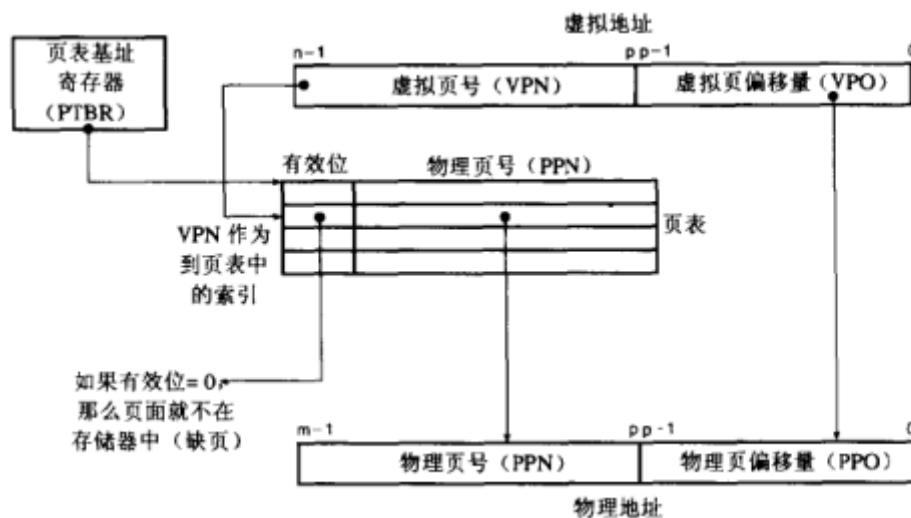
The concept of page hits is very simple . When a page that has been cached is accessed , it is called a page hit . This is the most ideal situation .

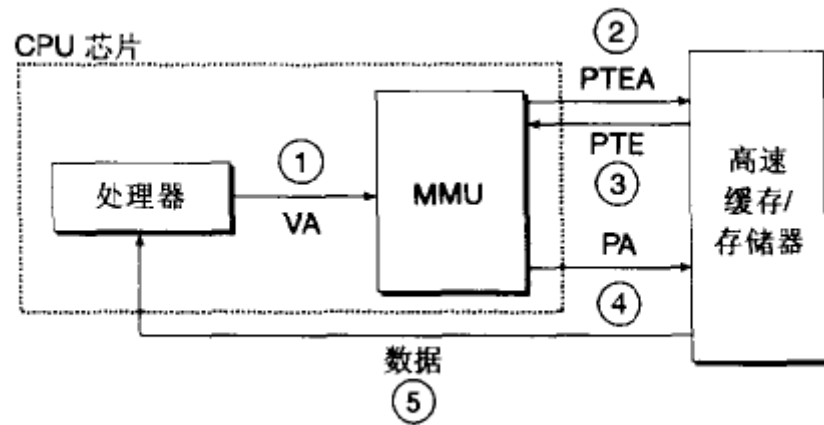
However , reality is always cruel , physical memory is always insufficient, and it is impossible for all virtual pages to be cached in physical memory , so page faults often occur .

The basic steps for page hits are :

- The first step : CPU generates a virtual address , and transmits it to the MMU
- Step two : MMU generate PTE address , and TLB / DRAM query page table
-  third step : TLB/DRAM returns the value of the page table entry to the MMU
- Step Four : MMU configured physical address , and transmits it to the memory bus
- Step 5 : The memory returns the requested data word to the CPU

The icons are as follows :





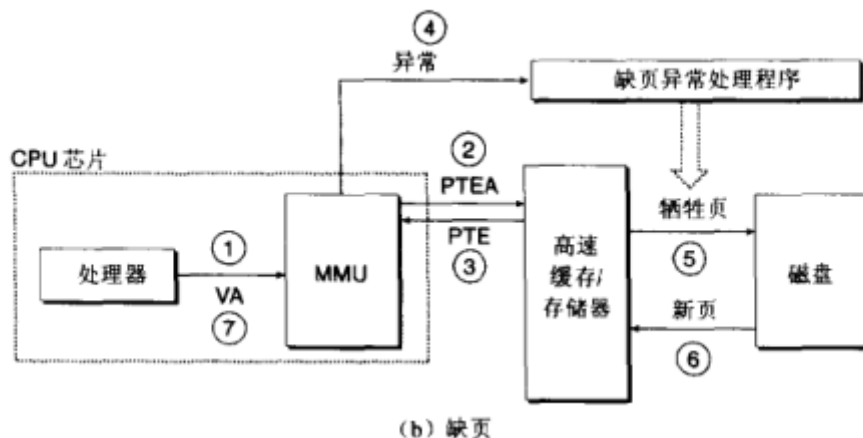
(a) 页面命中

The above steps are all completed by hardware , no operating system intervention is required .

However, if it is a page missing , the operating system needs to intervene . Let's take a look at the details of the page missing .

1.4.2 Missing pages

The handling of missing pages is as follows :



(b) 缺页

- first step -the third step : the same as the page hit
- Step 4 : The effective bit in the PTE is 0, so the MMU triggers a page fault exception and transfers the control in the CPU to the page fault exception handler in the operating system kernel .
- Step 5 : The page fault exception handler determines the victim page in the physical memory, and if the page has been modified , write its content back to the disk .
- Step Six : page fault handler swapped in from disk new page , and update the page table PTE.
- 7 : The page fault processing program returns to the original process and drives the instruction that caused the page fault to restart . The CPU re-sends the instruction that caused the page fault to the MMU. Then it enters the process described in the page hit .

1.4.3 working set (Working the SET)

When page faults occur , the cost is very high . The CPU needs to swap out & swap pages , and the speed of accessing the disk is usually very slow . If page faults occur frequently , program performance is bound to be affected .

But do not worry , there is an old friend can help us , locality (locality) .

Although the entire run , the total number of different procedures referenced page may exceed the total size of physical memory , but the locality principle ensures that at any given time , these pages will tend to work on a smaller event page collection , this collection called the working set (working the sET) . in the initial costs , then that it is the working set paging to memory , followed by a reference to the working set will result in a hit , but no additional disk traffic .

Note , not all programs can show a good locality , if the size of the working set exceeds the size of physical memory , then the program will result in an unfortunate state , called a bump (thrashing) , this is the page will continue to change into Swap out . If your program is slow like a crawl , then you may want to consider whether there will be bumps

1.4.4 Page recycling and page swapping

To meet the needs of users, or always meet the needs of memory-intensive applications, no matter how much physical memory is available on the computer, it is not enough. Therefore, the kernel swaps out part of the rarely used memory to disk, which is equivalent to providing more main memory. This mechanism is called paging (swapping, or in) or paging (Paging) .

This article does not intend to introduce the page recycling and exchange mechanism in detail , because the details will not be involved in the driver development process . From the perspective of driver development , we are more concerned about how the physical memory is managed , how the kernel and the application program are application memory , the impact of different ways of application memory performance .

If the reader is interested in the exchange mechanism page , you can read the "in-depth Linux kernel architecture," a book of 18 chapters .

1.4 Fragment

Since we are by page management of physical memory , so when allocating memory , but also an application page , inevitably introduce debris .

The macro definition of fragmentation is when the actual available physical memory is greater than the requested memory , and the application cannot be successful , the reason for the failure is fragmentation .

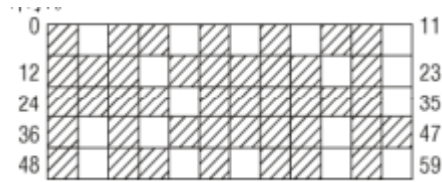
Fragments are divided into internal fragments and external fragments .

1.4.1 Internal Fragmentation

Assume that the page size is 4KB, when we only need 1KB when physical memory , the system will allocate a page frame to us . This is also the remaining frames of 3KB it can not be the people to use it , which creates internal fragmentation .

1.4.2 External Fragments

After the system starts and runs for a long time, the physical memory will generate a lot of external fragments , as shown in the figure :



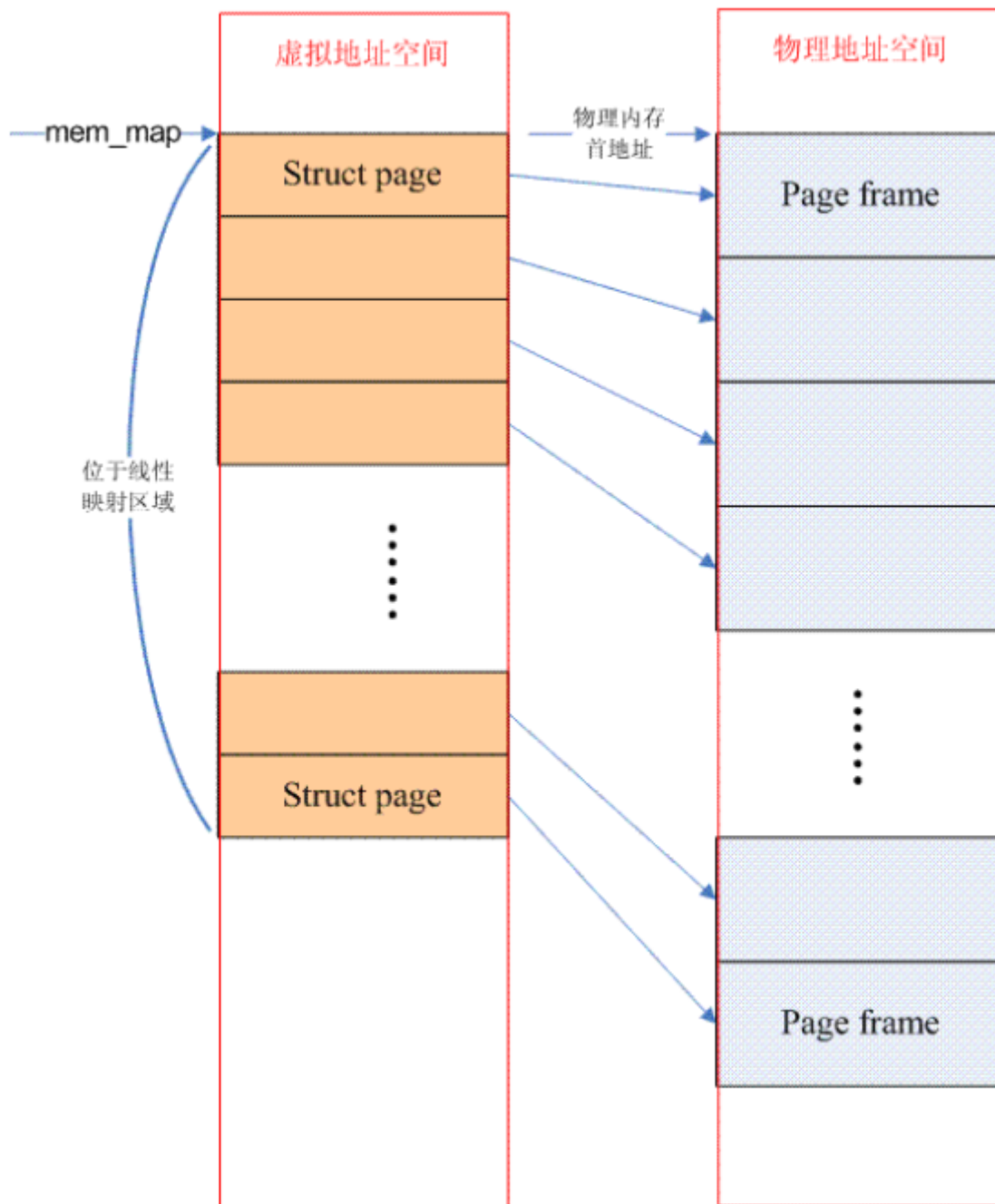
Although there are many page frames available in physical memory , we cannot apply for more than 2 consecutive pages . This situation is external fragmentation .

1.5 Three memory models in the Linux kernel

There are 3 memory models supported in the linux kernel , namely flat memory model , Discontiguous memory model and sparse memory model . The so-called memory model is actually the distribution of its physical memory from the perspective of the cpu , and in the Linux kernel , what method is used to manage these physical memory.

1.5.1 FLAT memory model

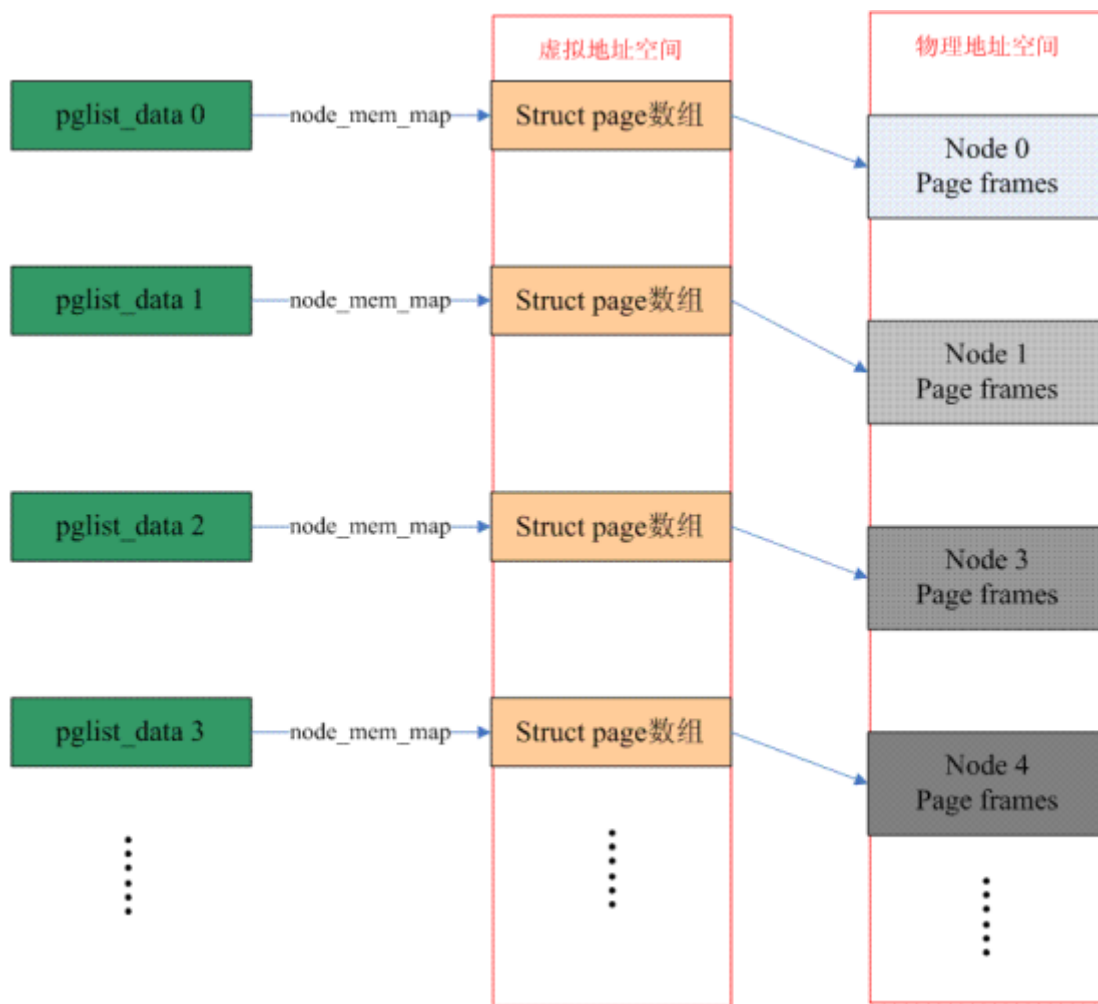
From the point of view of any processor in the system , when it accesses physical memory, the physical address space is a continuous address space without holes, then the memory model of this computer system is Flat memory . Under this memory model, the management of physical memory is relatively simple. Each physical page frame will have a page data structure to abstract. Therefore, there is an array of struct page (mem_map) in the system, and each array entry points to an actual physical page. Frame (page frame). In the case of flat memory , the relationship between PFN (page frame number) and mem_map array index is linear (there is a fixed offset, if the physical address corresponding to the memory is equal to 0 , then PFN is the array index). Therefore, it is very easy to get from PFN to the corresponding page data structure, and vice versa. For details, please refer to the definition of page_to_pfn and pfn_to_page . In addition, for flat memory model , there is only one node (struct pglist_data) (in order to use the same mechanism as the Discontiguous Memory Model). The following picture describes the situation of flat memory :



It should be emphasized that the memory occupied by struct page is in the directly mapped area, so the operating system does not need to create a page table for it .

1.5.2 Discontiguous Memory Model

If the cpu has some holes in its address space when accessing physical memory and is discontinuous, then the memory model of this computer system is Discontiguous memory . Generally speaking, the memory model of NUMA -based computer systems chooses Discontiguous Memory , but these two concepts are actually different. NUMA emphasizes the positional relationship between memory and processor , which has nothing to do with the memory model. However, because the memory and processor on the same node are more closely coupled (access faster), multiple nodes are required to manage . Discontiguous memory is essentially an extension of the flat memory memory model. The address space of the entire physical memory is mostly slices of large memory, with some holes in the middle. Each slice of memory address space belongs to a node (if limited to one nodeInternally, its memory model is flat memory). The following picture describes the situation of Discontiguous memory :



Therefore, under this memory model, there are multiple node data (struct pglist_data), and the macro definition NODE_DATA can get the struct pglist_data of the specified node . However, the physical memory managed by each node is stored in the node_mem_map member of the struct pglist_data data structure (the concept is similar to mem_map in flat memory). At this time, from PFN converted to a specific struct page will be a little more complicated, we first of all from PFN to get the Node ID , and then based on this ID to find for the pglist_data data structure, will find a corresponding page array, after a similar method to flat memory .

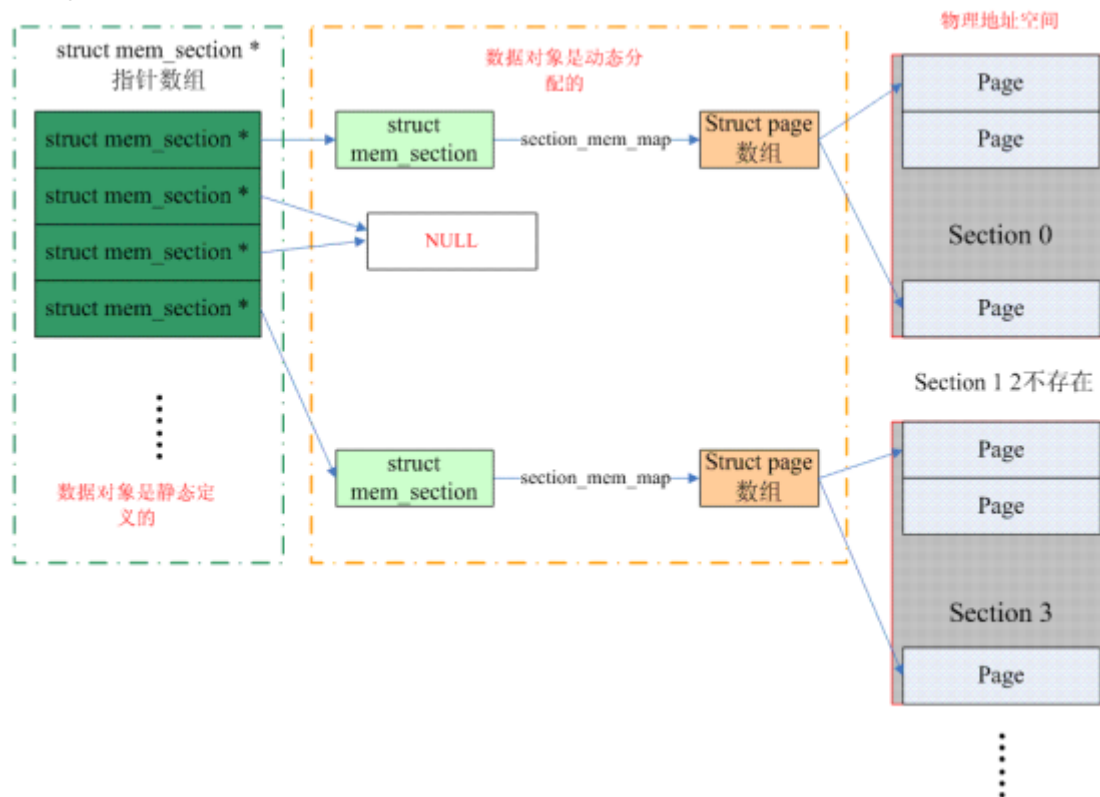
1.5.3 Sparse Memory Model

Memory model is an evolutionary process, the beginning of time, the use of flat memory to abstract a contiguous memory address space (mem_maps []), appears NUMA Thereafter, the entire discontinuous memory is divided into several node , each node on a The continuous memory address space, that is, the original single mem_maps[] has become several mem_maps[] . Everything looks perfect, but the appearance of memory hotplug makes the original perfect design imperfect, because even the mem_maps[] in a node may be discontinuous. In fact, after the emergence of sparse memory , the Discontiguous memory memory model is no longer so important. It stands to reason that sparse memory can eventually replace Discontiguous memory . This replacement process is in progress. The 4.4 kernel still has 3 memory models to choose from.

Why is it that sparse memory can finally replace Discontiguous memory ? In fact sparse memory the memory model, contiguous address space in accordance with the SECTION (eg . 1G) is divided into some section , wherein each section is hotplug , so sparse memory , the memory address space may be divided finer cut , Support more discrete Discontiguous memory . In addition, before sparse memory appeared, NUMA and Discontiguous memory always cut continuously, and the relationship was still messy: NUMA did not stipulate the continuity of its memory, and the Discontiguous memory system was not necessarily a NUMA system, but these two configurations They are

all multi-node . With sparse memory , we can finally compare the continuity of memory with NUMAThe concept is separated: a NUMA system can be flat memory or sparse memory , and a sparse memory system can be NUMA or UMA .

The following picture illustrates how sparse memory manages the page frame (SPARSEMEM_EXTREME is configured):

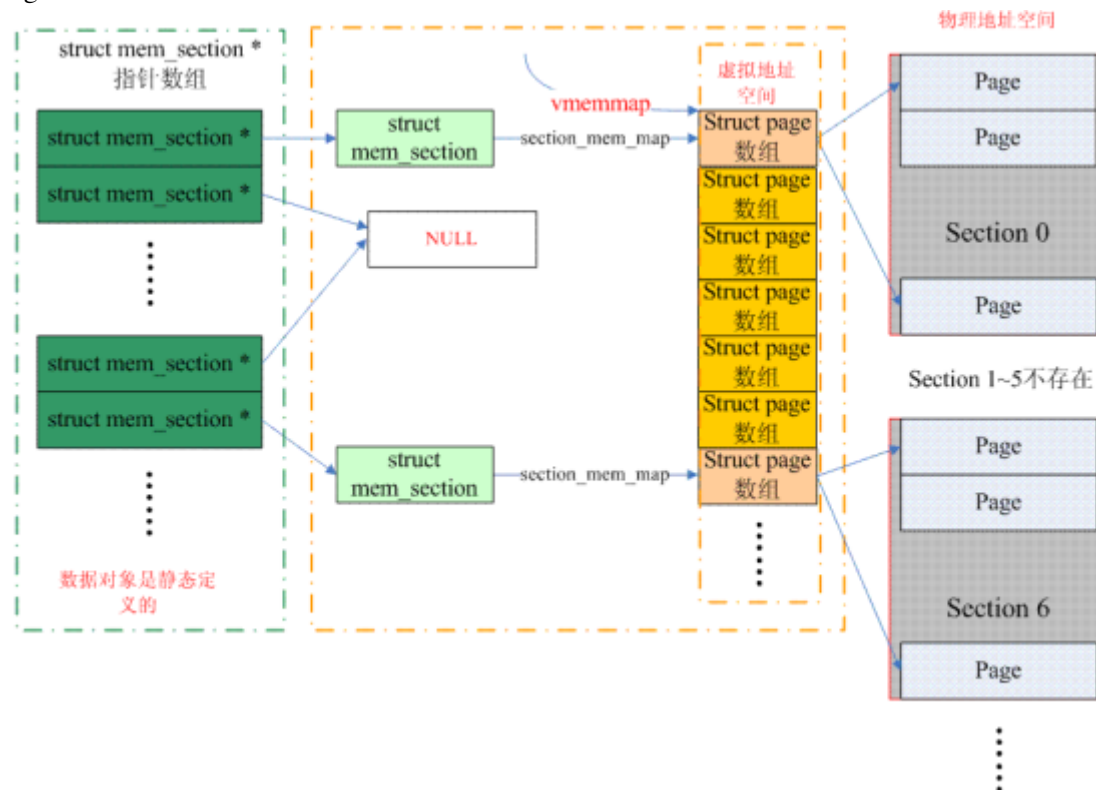


(Note: a mem_section pointer in the above figure should point to a page , and there are several struct mem_section data units in a page)

The entire continuous physical address space is cut off according to one section and one section . Inside each section , its memory is continuous (that is, it conforms to the characteristics of flat memory). Therefore, the page array of mem_map is attached to the section structure (struct mem_section). It is not a node structure (struct pglist_data). Of course, no matter what kind of memory model , you need to deal with the correspondence between PFN and page , but sparse memory has one more section concept, so that the conversion becomes PFN<—>Section<—>page .

Let's first look at how to convert from PFN to page structure: A pointer array of mem_section is statically defined in the kernel . A section often includes multiple pages . Therefore, PFN needs to be converted to a section number by right shifting , using section number as Index can find the section data structure corresponding to the PFN in the mem_section pointer array . After finding the section , you can find the corresponding page data structure along its section_mem_map . By the way, at the beginning, sparse memory uses a one-dimensional memory_section array (not an array of pointers). Such an implementation is a waste of memory for a particularly sparse (CONFIG_SPARSEMEM_EXTREME) system. In addition, save the pointer to hotplugThe support of is more convenient, the pointer is equal to NULL means that the section does not exist. The above picture describes the one-dimensional mem_section pointer array (configured with SPARSEMEM_EXTREME). For non-SPARSEMEM_EXTREME configuration, the concept is similar. For specific operations, you can read the code yourself .

There is a little trouble from page to PFN . In fact, PFN is divided into two parts: one part is the section index , and the other part is the offset of the page in the section . We need first of all from the page to get section index , will get the corresponding memory_section , know memory_section will know the page in section_mem_map , will know the page in the section offset, and finally can be synthesized PFN . For the conversion from page to section index , sparse memory has two schemes. Let's take a look at the classic scheme first, which is saved in page->flags (SECTION_IN_PAGE_FLAGS is configured)). The biggest problem with this method is that the number of bits in page->flags is not necessarily enough, because this flag carries too much information, various page flags , node id , zone id now add a section id , in A consistent algorithm cannot be achieved in different architectures . Is there a general algorithm? This is CONFIG_SPARSEMEM_VMEMMAP . The specific algorithm can refer to the following figure:



(There is a problem with the above picture. `vmemmap` points to the first `struct page` array only when `PHYS_OFFSET` is equal to 0. Generally speaking, there should be an offset , but I don't want to change it, haha)

For the classic sparse memory model, the memory occupied by the `struct page` array of a section comes from the directly mapped area. The page table is created when it is initialized, and the page frame is allocated, that is, the virtual address is allocated. However, for `SPARSEMEM_VMEMMAP` , the virtual address is allocated at the beginning. It is a continuous virtual address space starting from `vmemmap` . Each page has a corresponding `struct page` . Of course, there are only virtual addresses and no physical addresses. Therefore, when a section is found, the virtual address of the corresponding `struct page` can be found immediately . Of course, a physical page frame needs to be allocated , and then a page table or something needs to be created. Therefore, for this kind of sparse memory , the overhead will be slightly larger. Some (more on the process of establishing a mapping).

