什么时候虚拟页处于未缓存（未缓存和缓存状态的具体区别）？

Brk

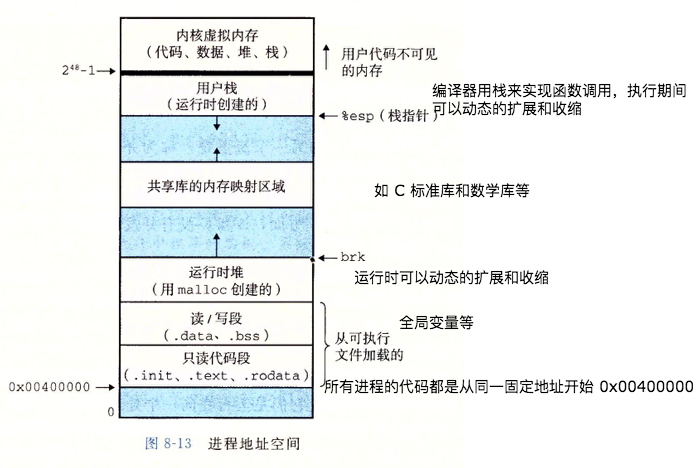
Mmap



https://www.cnblogs.com/huxiao-tee/p/4660352.html

未缓存的虚拟页是怎么和磁盘对应起来的？

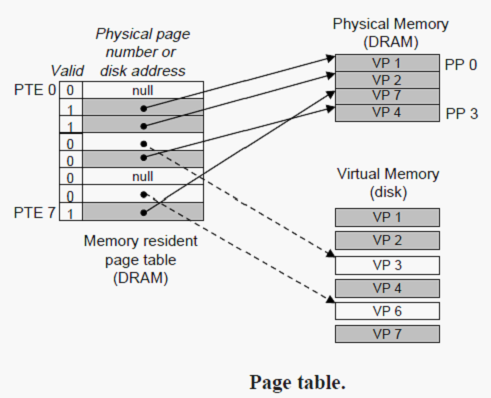
1.磁盘存储的文件包括可执行等文件。



可执行文件固定的虚拟起始地址是0X00400000，这样虚拟地址0X00400000就映射到磁盘上可执行文件。但是不一定缓存到物理内存中。

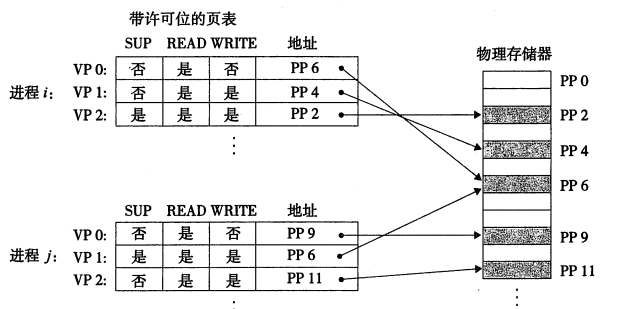
虚拟页、虚拟页表、物理页。

OS、MMU、DISK、DRAM



虚拟存储对物理内存的保护：

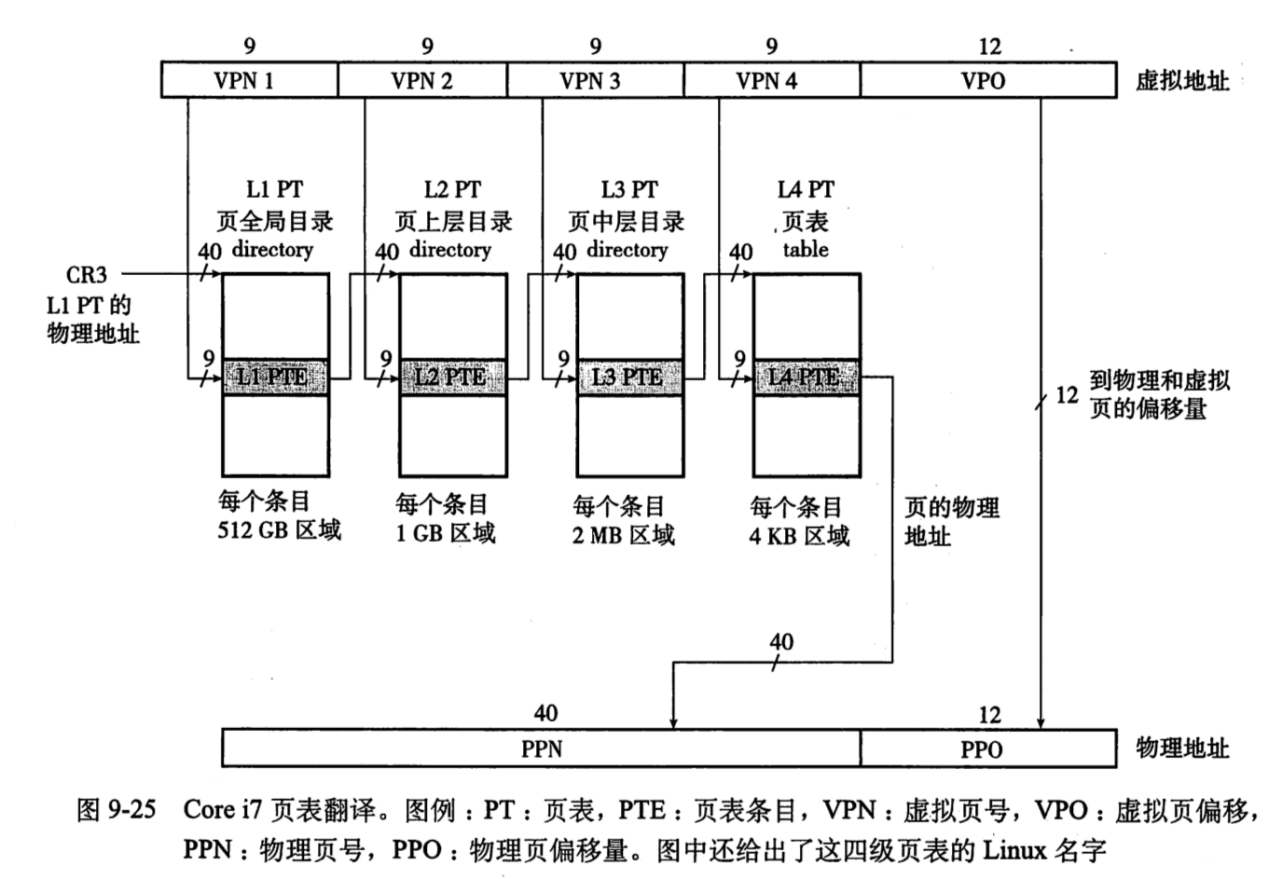
PTE添加了三个选项.

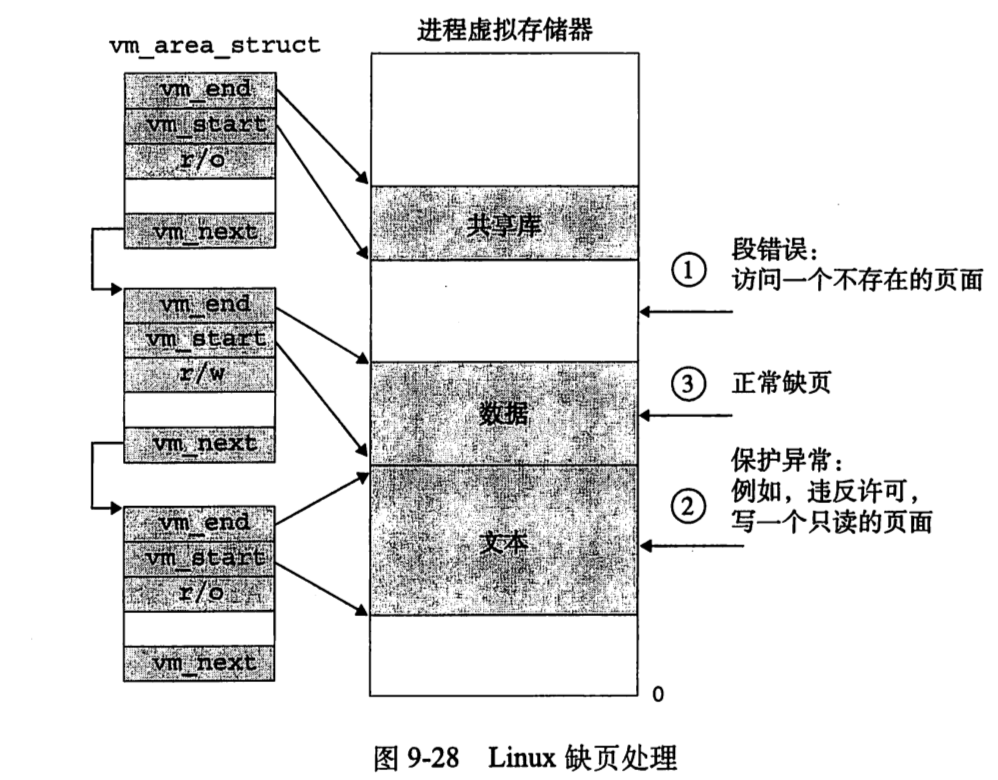


64位的linux core i7上的虚拟地址48位，物理地址是52位。

(gdb) p &buf

$5 = (struct rusage \*) 0x7fffffffd8e0





通过mmap映射相应的虚拟地址，保存在vm\_area\_struct链表中，当访问的虚拟地址不在vm\_area\_struct链表范围内就会触发段错误。

Bite 位：一位二进制。

Byte 字节：8位二进制 标示一个ascii码。

Word 字：根据计算机位数决定，32位的是32为一个字。

1. 当malloc()一块很小的内存是, glibc调用brk(), 只需要在heap中移动一下指针, 即可获得可用虚存, 这样分配得到的地址较小.

2. 当malloc()一块较大内存时, glibc调用mmap(), 需要在内核中重新分配vma结构等, 他会在靠近栈的地方分配虚存, 这样返回的地址大.

**内核代码分析linux 内存管理：**

**1.虚拟内存的管理：**

1)创建虚拟地址

unsigned long do\_mmap(struct file \*file, unsigned long addr,

unsigned long len, unsigned long prot,

unsigned long flag, unsigned long offset)

do\_mmap\_pgoff

|

|-------get\_unmapped\_area-------mm->get\_unmapped\_area(arch\_get\_unmapped\_area) 查找可用的地址空间

|

|-------mmap\_region 创建新vma线性区

|

|-------find\_vma\_prepare-------do\_munmap vma线性区已经存在进行unmap 成功返回找到的vma

|

|-------vma\_merge 合并vma线性区

|

|-------vma = kmem\_cache\_zalloc-------file->f\_op->mmap(file, vma)

|

|-------vma\_link mm->map\_count++

2)查找虚拟地址空间

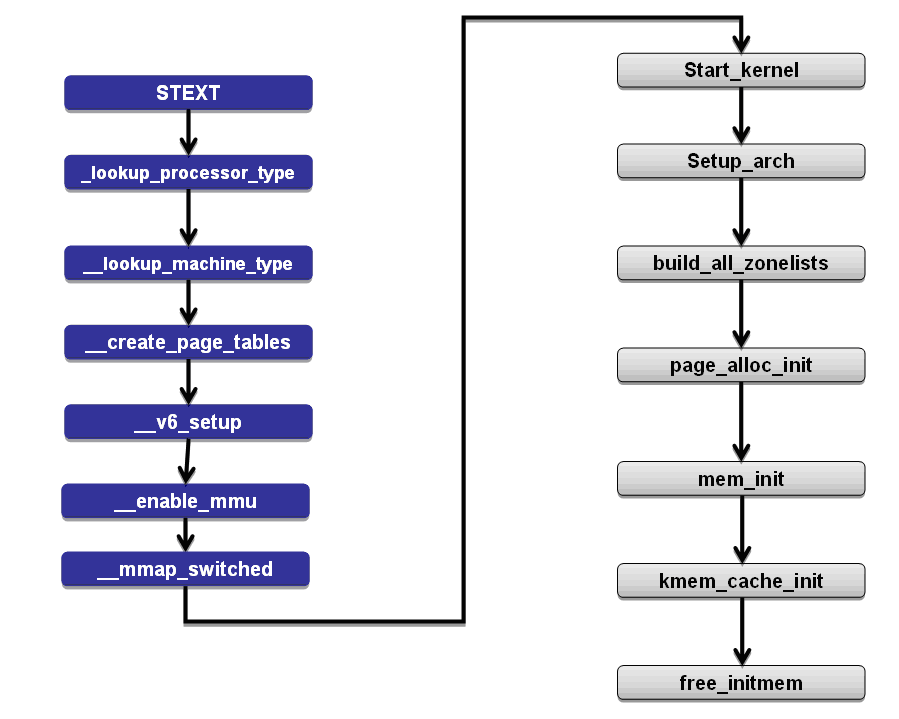
struct vm\_area\_struct \*find\_vma(struct mm\_struct \*mm, unsigned long addr)

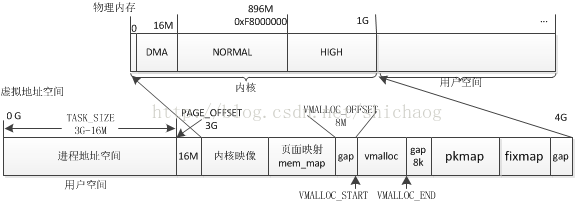
3)删除虚拟内存区域

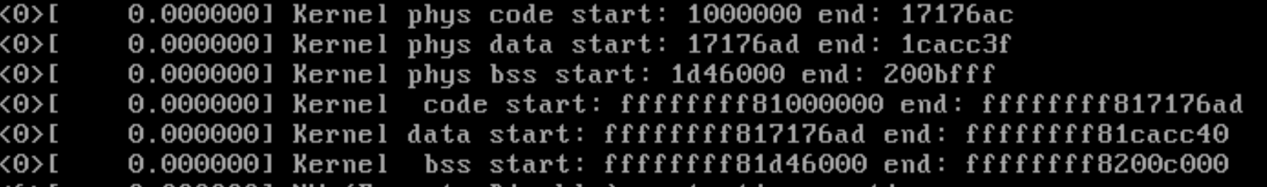
do\_munmap

2.物理内存的管理

物理内存初始化：







线性地址就是内核的虚拟地址，内核的线性地址对于进程来说也是对应的虚拟地址。

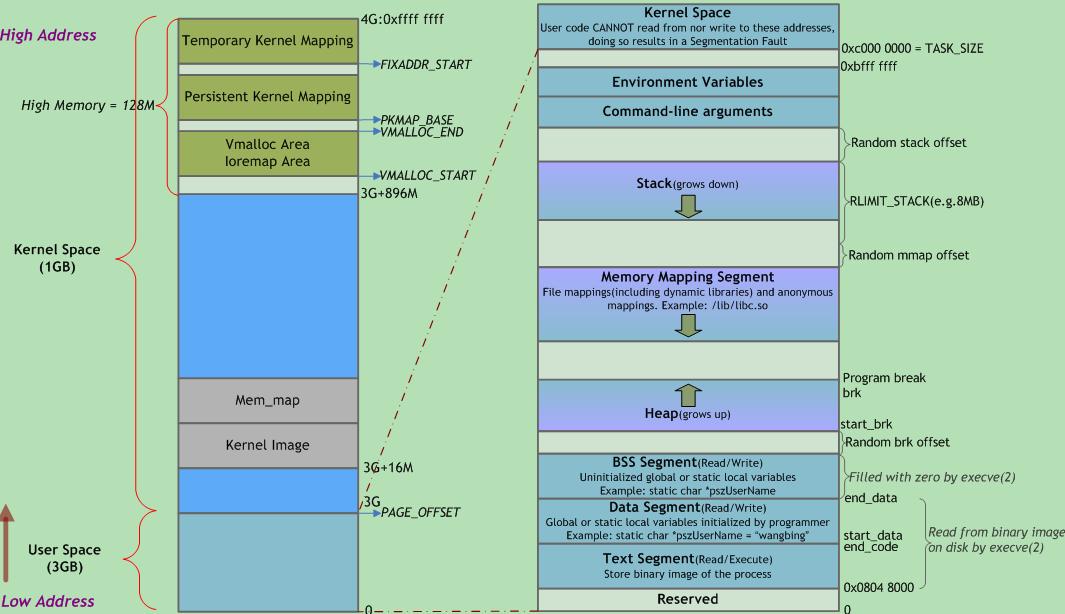
对于64位的线性地址和内存物理地址的关系：

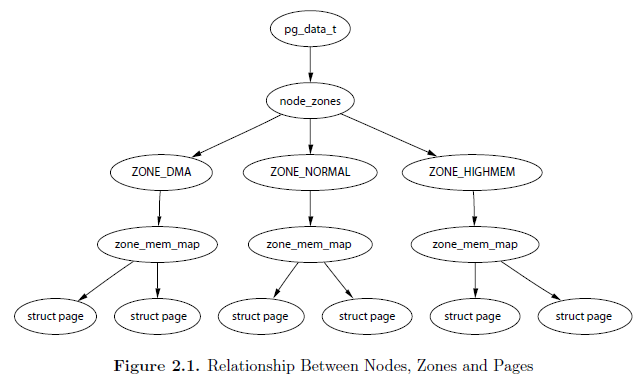
线性地址 = ffffffff80000000 + 物理地址

内核的代码段加载在物理内存16MB开始的地方。

内核的线性地址是所有task共享的，用户态的虚拟地址各个进程是不一样的。

所以用户态和内核态的区别就是用户态的时候只能访问当前进程的虚拟地址，而内核态的时候可以访问内核的线性地址。





Node:

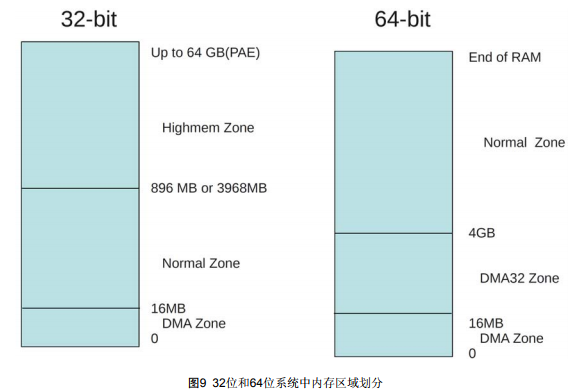
pglist\_data

Mmzone\_64.h (arch\x86\include\asm)

extern struct pglist\_data \*node\_data[]

Zone：

include\linux\Mmzone.h



ZONE\_DMA和早期的ISA设备是有关系的

ZONE\_NORMAL指示的是可直接映射到内核段的地址空间

ZONE\_HIGH是超出内核段的物理内存

Page：

3.虚拟内存和物理内存的映射

内核内存的映射：

1.计算max\_pfn

unsigned long \_\_init e820\_end\_of\_ram\_pfn(void)

{

return e820\_end\_pfn(MAX\_ARCH\_PFN, E820\_RAM);

}

//算出小于 limit\_pfn的物理内存的最大的页帧数。

static unsigned long \_\_init e820\_end\_pfn(unsigned long limit\_pfn, unsigned type)

{

循环遍历e820中的type为E820\_RAM的e820entry

通过e820entry中的addr和size算出start\_pfn和end\_pfn

取最大的end\_pfn作为物理内存在 limit\_pfn范围内的 last\_pfn

}

limit\_pfn是理论上最大的页面帧数，在32位MAX\_ARCH\_PFN定义为1左移24（开PAE）或者20位的数目。在64位中 MAXMEM

#ifdef CONFIG\_X86\_32

# ifdef CONFIG\_X86\_PAE

# define MAX\_ARCH\_PFN (1ULL<<(36-PAGE\_SHIFT))

# else

# define MAX\_ARCH\_PFN (1ULL<<(32-PAGE\_SHIFT))

# endif

#else /\* CONFIG\_X86\_32 \*/

# define MAX\_ARCH\_PFN MAXMEM>>PAGE\_SHIFT

#endif

2.计算max\_low\_pfn

当max\_pfn大于4GB的时候需要去计算max\_low\_pfn

取出e820第4个e820entry中的end\_pfn作为max\_low\_pfn.

4.Memblock

struct memblock\_region {

phys\_addr\_t base;

phys\_addr\_t size;

#ifdef CONFIG\_HAVE\_MEMBLOCK\_NODE\_MAP

int nid;

#endif

};

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 {

phys\_addr\_t current\_limit;

struct memblock\_type memory;

struct memblock\_type reserved;

};

extern struct memblock memblock;

void \_\_init memblock\_x86\_fill(void)

{

int i;

u64 end;

memblock\_allow\_resize();

for (i = 0; i < e820.nr\_map; i++) {

struct e820entry \*ei = &e820.map[i];

end = ei->addr + ei->size;

if (end != (resource\_size\_t)end)

continue;

if (ei->type != E820\_RAM && ei->type != E820\_RESERVED\_KERN)

continue;

memblock\_add(ei->addr, ei->size);

}

memblock\_dump\_all();

}

5.bootmem