# 1、逻辑地址、虚拟地址、线性地址、物理地址的区别是什么

（1）逻辑地址：机器语言指令中的内存地址，都是逻辑地址，需要转换成线性地址，再转为物理地址才能被访问。逻辑地址由段标识符（段选择符，16位，由6种段寄存器存放）和指定段内相对地址的偏移量（32位）组成。

分段单元将逻辑地址转换为线性地址流程如下：

图片包含 游戏机

描述已自动生成

段描述符8字节（段基址线性地址，长度，权限DPL等），常用有代码段描述符，数据段描述符，任务状态段描述符TSSD（表示任务状态段TSS，保存处理器寄存器值），局部描述符表描述符LDTD。

段选择符中13位是指定在GDT/LDT中相应段描述符入口，TI为1指明段描述符在GDT中，为0表示在LDT中，剩余2位是请求者特权级RPL，0为内核态，3为用户态。

注意：Linux中逻辑地址等于线性地址。Linux所有的段（用户代码段、用户数据段、内核代码段、内核数据段）的线性地址都是从 0x00000000开始，长度4G。GDT的第12和13项段描述符是 \_\_KERNEL\_CS 和\_\_KERNEL\_DS，第14和15项段描述符是 \_\_USER\_CS 和\_\_USER\_DS。内核任务使用\_\_KERNEL\_CS 和\_\_KERNEL\_DS，所有的用户任务共用\_\_USER\_CS 和\_\_USER\_DS，也就是说不需要给每个任务再单独分配段描述符。内核段描述符和用户段描述符虽然起始线性地址和长度都一样，但DPL(描述符特权级)是不一样的。\_\_KERNEL\_CS 和\_\_KERNEL\_DS 的DPL值为0（最高特权），\_\_USER\_CS 和\_\_USER\_DS的DPL值为3。

（2）线性地址（也叫虚拟地址）

32位无符号整数，表示4GB的地址。线性地址被以固定长度为单位划分为页page（一个数据库，可以存放在任何页框或磁盘中），可指定一页的存取权限。分页单元把RAM分成固定长度页框（物理页），是主存的一部分，也是一个存储区域。

分页单元通过页表将线性地址转为物理地址，注意比较权限，内存访问无效时产生缺页异常：（控制寄存器CR0的最高位PG位控制着分页管理机制是否生效，如果PG=1，分页机制生效）

（3）物理地址

用于内存芯片级的单元寻址，与处理器和CPU连接的地址总线相对应。由32位或36位无符号整数表示。

# 2、为什么每个进程都需要自己的页表，每个线程是否有自己的页表，每个内核线程呢？

进程如果没有自己的页表，共用物理内存空间，会导致互相污染内存数据，无法做到进程之间的隔离。为了解决进程之间内存隔离问题，提供了虚拟地址，不同进程的虚拟空间布局可能相同，但是通过不同进程自己的页表转换得到的物理内存空间则不同。

线程与进程共享地址空间，没有自己的页表。进程有独立的地址空间，线程之间共享大部分数据（除了栈（用户栈通过mmap传给clone）之外的：堆、代码段、数据段（已初始化且不为0的全局变量和静态变量）、bss段（未初始化或初始化为0的全局变量和静态变量）；打开的文件描述符；进程当前目录；进程用户ID和进程组ID；信号处理器。但有自己的PID、栈、寄存器集合、信号掩码、优先级），使用与进程相同的地址空间。线程创建的时候，加上了 CLONE\_VM 标记，这样线程的内存描述符将直接指向父进程的内存描述符；否则就要通过dup\_mm->allocate\_mm新建mm\_struct，并复制主内核页表的内核地址空间页目录项（768项之后的部分），然后拷贝父进程的用户空间页表项并设置物理页写保护，最后写时复制时才真的复制物理页。

内核线程运行在内核态，不访问低于PAGE\_OFFSET的地址，而大于PAGE\_OFFSET线性地址的相应页表项都相同，因此，内核线程使用一组最近运行的普通进程的页表，即mm为空，但是active\_mm初始化为前一个运行进程的active\_mm（普通线程的这两个字段存放相同指针）。

内核本身维持主内核页全局目录，其最高目录项部分作为参考模型，为系统中每个普通进程对应的页全局目录项提供参考模型。处于内核态的进程修改了高端线性地址时（vmalloc和vfree），linux采用延迟更新，vmalloc时先修改内核页表，当真正使用时，触发缺页处理程序，会判断这个地址所在空间，如果是在内核地址空间的VMALLOC区，那么内核会去内核页表中查看，该地址是不是在内核中有记录。如果有记录则把内核的addr对应的pmd项复制给进程的pmd项，意味着，进程和内核公用一个pte页表；进程要vfree释放这个区域，其实修改的还是内核页表，会把addr对应的pte页表项设置为0。其它的都不做改变。那么当进程试图访问一个已经被释放区间的地址addr时候，由于它和内核对于addr的pmd项是一样的，所以，会继续去访问内核页表关于addr的pte页表，最后发现，pte页表项为0，又触发了缺页异常。这次的缺页异常和上面分配的流程一样，只是最后对内核页表pte项做检查时候，发现内核页表关于addr的pte页表项是0，就会报错。这样就避免了进程的非法访问。

# 3、进程切换（换入、换出）时是如何保留自己的页表的

进程切换涉及到schedule通过context\_switch实现上下文切换，首先通过switch\_mm把虚拟内存从一个进程切换到另一个进程，更换通过task\_struct->mm描述的内存管理上下文（load\_cr3(next->pgd);//主要包括将下一个进程的页目录基地址（页目录在内核地址空间，pgd与其物理地址相差PAGE\_OFFSET）写入cr3寄存器（存放物理地址）以加载页表，刷出TLB、向MMU提供新信息。

# 4、x86\_32的虚拟地址空间如何分布，x86\_64呢

（1）x86\_32

0-3g是用户地址空间，3-4g是内核地址空间。各个进程的内核地址空间一样，但用户空间各自不同确保不相互干扰。



用户虚拟地址空间从下到上分布：

代码段（可执行代码、字符串字面值、只读变量）

数据段（初始化且不为0的全局和静态局部变量。占用可执行文件空间和内存，会缺页再分配物理内存）

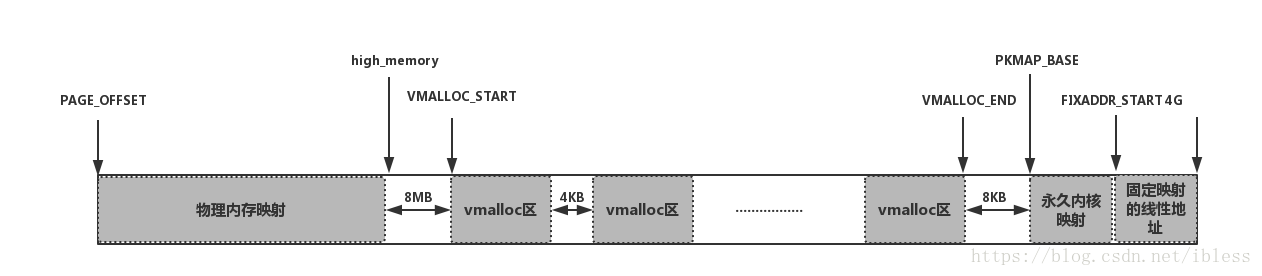
bss（未初始化或初值为0的全局变量和静态局部变量。不占可执行文件空间，仅为变量总和预留位置，加载可执行文件时会为bss中的变量分配足够的空间并初始化为0，占用内存空间，不会缺页，因为会转到全0页面，属于静态内存分配）

堆（动态分配的内存。Malloc和free。当小于128k时brk增大连续内存，释放时不归还虚拟和物理内存，因为仅一个brk指针，而后可重用碎片。分配失败返回NULL）

mmap（可共享内存，大于128k的malloc用此在mmap区新建vm\_area\_struct，munmap释放虚拟和物理内存）

栈（局部变量、函数参数、返回地址等。为函数内部声明的非静态局部变量(C语言中称“自动变量”)提供存储空间；记录函数调用过程相关的维护性信息，称为栈帧；临时存储区，用于暂存长算术表达式部分计算结果或alloca()函数分配的栈内内存。函数结束则释放此栈帧。可动态增长）

内核虚拟地址空间划分如下：



上图中PAGE\_OFFSET通常为0xC000 0000（3G），而high\_memory指的是0xF7FF FFFF（3G+896M），在物理内存映射区和和第一个vmalloc区之间插入的8MB的内存区是一个安全区，其目的是为了“捕获”对内存的越界访问。处于同样的理由，插入其他4KB大小的安全区来隔离非连续的内存区。Linux内核可以采用三种不同的机制将页框（物理）映射到高端内存区（虚拟），分别叫做持久内核映射、临时内核映射以及非连续内存分配。相关区域如下：

|  |  |
| --- | --- |
| 虚拟内存区域 | 说明 |
| 物理内存映射 | 映射了物理内存空间0~896M部分，这一部分是内核直接映射的 |
| 持久内核映射 kmap | 允许内核建立高端页框到内核虚拟地址空间的长期映射。永久内核映射不能用于中断处理程序和可延迟函数，因为建立永久内核映射可能阻塞当前进程 |
| 固定映射的线性地址空间 | 其中的一部分用于建立临时内核映射 |
| vmalloc区 | 该区用于建立非连续内存分配，虚拟固定映射地址与物理内存位置之间的关联可以自行定义 |

（2）X86\_64

64位下0~0x7fff ffff ffff为用户虚拟地址空间，0xffff 8000 0000 0000 ~ 0xffff ffff ffff ffff为内核虚拟地址空间，由于内核空间虚拟地址可表示的内存范围很大（247，即128TB,而一般机器的物理内存都不会这么大，因此内核虚拟地址可以全部一致映射到物理内存，即64位下对于每个pa,都可以通过加上page\_offset的方式得到对应的va,而32位下则需要高端内存）

64位下线性地址布局：

手机屏幕截图

描述已自动生成

0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm

hole caused by [48:63] sign extension

ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for hypervisor

ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys. memory

ffffc80000000000 - ffffc8ffffffffff (=40 bits) hole

ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap space

ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole

ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)

... unused hole ...

ffffec0000000000 - fffffc0000000000 (=44 bits) kasan shadow memory (16TB)

... unused hole ...

ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks

... unused hole ...

ffffffff80000000 - ffffffffa0000000 (=512 MB) kernel text mapping, from phys 0

ffffffffa0000000 - ffffffffff5fffff (=1525 MB) module mapping space

ffffffffff600000 - ffffffffffdfffff (=8 MB) vsyscalls

ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole

上述布局中0000000000000000 - 00007fffffffffff (=47 bits)是进程的用户空间，而剩下的ffff80000000以上的空间是系统空间，所以线性地址空间是对半分的，应用程序可使用最大达128TB的地址空间。

ffff880000000000 - ffffc7ffffffffff (=64 TB)是对所有物理内存的线性映射，这个映射可以用个函数来表示： 假设线性地址X在这个地址空间中，那么 X的物理地址 = X - 0xffff8800 00000000内核初始化阶段中会在这片空间中建立起对所有物理内存(<=64TB)的永久映射，这个映射建立好之后，内核就可以用物理地址+0xffff8800 00000000来访问0～64TB之间的任意内存。

ffffffff80000000 - ffffffffa0000000 (=512 MB) 这片空间存放着内核指令、数据。所以内核要小于512MB。可以看到内核映像、指令是放在物理内存最开始的地方。且对于内核指令数据这512m内存中的某个物理地址pa，使用pa+ ffffffff80000000或者pa+ ffff880000000000都可以得到对应的虚拟地址，内核中这两块虚拟地址都映射到这块物理内存。

# 5、不同页表的两个页表项可以指向同一个页么

可以。描述页框状态信息的page页描述中，字段\_count表示页的引用计数，为-1时表示空闲，大于或等于0时，表示页框被分配给一个或多个进程。

1、 父子进程，写时复制

新建子进程，不共享地址空间，通过dup\_mm->allocate\_mm新建mm\_struct，并mm\_init->

（1）mm\_alloc\_pgd为子进程页表分配页目录，并复制主内核页表的内核地址空间的页目录项，即页目录项的768项之后的部分（前768项是3G用户地址空间）；

（2）dup\_mmap->copy\_page\_range会通过vm\_flags设置物理页的写保护，再拷贝用户空间页表项。copy\_pud\_range->copy\_pmd\_range-> copy\_pte\_range->copy\_one\_pte，分别拷贝各级页表条目，最后的copy\_one\_pte进行pte的复制，父子进程页表的写时复制也在这里体现：

if (is\_cow\_mapping(vm\_flags)) {

ptep\_set\_wrprotect(src\_mm, addr, src\_pte);

pte = pte\_wrprotect(pte);

}

判断父进程的页是否支持写时复制，如果支持则将父子进程的页都置为写保护。等到父子进程某一方需要对页写的时候，就会复制一个新的页，没有写的时候两个页表指向相同的只读页。

2、共享内存

System v和mmap共享内存。进程间需要共享的数据被放在该共享内存区域中，所有需要访问该共享区域的进程都要把该共享区域映射到本进程的地址空间中去。

以system v为例，当进程第一次通过虚拟地址访问该共享内存区域时，会因为还没有建立对应的页表项而产生page fault异常。Linux内核处理这个page fault，其先找到发生异常的虚拟地址所在的vm\_area\_struct数据结构。由于之前映射了shm文件，并将该文件对应的操作函数赋值给vma->vm\_ops，因此这里直接使用vma->vm\_ops->fault处理该异常，对应到shmem\_fault，该函数从交换缓存中查找对应的页，或者从交换区中换入对应的页。对于第一次访问，该页还没有分配，所以这里是直接分配对应的物理页并建立进程到该物理页面的页表项。

当下一个进程试图访问这块内存并得到一个page fault的时候，经过同样的路径，也会走到函数shmem\_fault。此时，该内存页已经分配且在交换缓存中，因此可以直接建立对应的页表项，而不需要重新分配物理页。所以，是第一个访问共享内存页的进程使得这一页被创建，而随后访问它的其它进程仅把此页映射到它们的虚拟地址空间。

# 6、X86：32位，PAE，64位，页表组织是怎样

地址总线宽度决定了最大寻址空间，如i386地址总线为32位，寻址4GB；32位PAE模式下地址总线36位（可能更多），可寻址64GB；64位下地址总线一般为48位，寻址256TB。

1、32位地址总线32位，寻址4G物理内存。常规分页 页表项4字节 （1）4KB页 2级分页 线性地址划分：10-10-12；如下表。

|  |  |
| --- | --- |
| 单元 | 描述 |
| 页目录表Directory | 最高10位 |
| 页中间表Table | 中间10位 |
| 页内偏移 | 最低12位 |

（2）扩展分页 4MB页 1级页表 10-22。

2、32位PAE扩展分页，地址总线36位，寻址64G，但一个进程碍于虚拟地址宽度，只能访问4G物理内存。页表项8字节（因此每一页能存放的pte数目从1024变成512，相应地，页表层级发生了变化，只需要使用9位查找一个页表项）。（1）4KB页 3级分页 2-9-9-12；如下表。

|  |  |  |
| --- | --- | --- |
| 字段 | 描述 | 位数 |
| cr3 | 指向一个PDPT | crs寄存器存储 |
| PGD | 指向PDPT中4个项中的一个 | 位31~30。 2 |
| PMD | 指向页目录中512项中的一个 | 位29~21。 9 |
| PTE | 指向页表中512项中的一个 | 位20~12。 9 |
| page offset | 4KB页中的偏移 | 位11~0。 12 |

CR3存储的是页目录基地址，页目录表中每个表项是8个字节，共有四个表项，每个表项对应1GB的内存空间（212+9+9）。对应于这四个表项，处理器为每个表项维护一个寄存器称为PDPTE0，PDPTE1，PDPTE2，PDPTE3。处理器在适当的时候从页目录表中加载值到这些寄存器，在访问的时候使用寄存器来访问，可以减少了一次内存访问。

（2）2MB页 2级分页 2-9-21。如下图。

手机屏幕的截图

描述已自动生成

其中M表示实际的物理地址位宽，假如说是64GB即36位，那么M=36。其他位说明如下，对应到各级可能存在差异，但大致都一样，如bit0为存在位，1为读写位，保留位为0，PDE的bit7置位说明指向一个2M的大页，否则指向下一级页表。Bit 64置位说明指向的页面不可执行（注意：此处pgd的读写位及不可执行保护位均为ignored）【32位开启PAE模式，及x86-64架构下提供了不可执行位。在分页结构的任何级别置位不可执行位都可以激活不可执行保护，而不用考虑分页的其他级】

3、64位分页 x86\_64平台 页表项8字节 寻址使用48位 4级分页 9-9-9-9-12。（通过取消PUD和PMD来适用于32位3和2级分页，对应页目录项数设置为1，并将其映射到PGD的适当目录项）。

64位处理器的硬件分页系统都使用了额外的分页级别. 使用的级别取决于处理器的类型：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 平台名称 | 页大小 | 寻址所使用的位数 | 分页级别数 | 线性地址分级 |
| alpha | 8KB | 43 | 3 | 10 + 10 + 10 + 13 |
| ia64 | 4KB | 39 | 3 | 9 + 9 + 9 + 12 |
| ppc64 | 4KB | 41 | 3 | 10 + 10 + 9 + 12 |
| sh64 | 4KB | 41 | 3 | 10 + 10 + 9 + 12 |
| x86\_64 | 4KB | 48 | 4 | 9 + 9 + 9 + 9 + 12 |

X86-64下各级页表如下：

|  |  |  |
| --- | --- | --- |
| 字段 | 描述 | 位数 |
| PGD | 指向页全局目录512项中的一个 | 位47~39。 9 |
| PUD | 指向页上级目录512个项中的一个 | 位38~30。 9 |
| PMD | 指向页中间目录中512项中的一个 | 位29~21。 9 |
| PTE | 指向页表中512项中的一个 | 位20~12。 9 |
| page offset | 4KB页中的偏移 | 位11~0。 12 |

页表寻址如下：

手机屏幕截图

描述已自动生成

64位下各级页表项标志如下：

图片包含 游戏机

描述已自动生成

各个位的标志与32位PAE下类似，但64位提供了4K,2M,1G三种页面类型。

# 7、x86架构下的内存为什么分为ZONE\_DMA、ZONE\_DMA32、ZONE\_NORMAL、ZONE\_HIGH？ X86\_64下包含哪些Zone类型？

因为80x86架构有两种硬件约束：

（1）ISA总线的直接内存存取（DMA）处理器有一个严格的限制：它们只能对RAM的前16MB寻址；

（2）在具有大容量RAM的现代32位计算机中，由于线性地址空间大小，CPU不能直接访问所有物理内存。（只有3G+896M的内核虚拟地址空间直接映射到物理内存的前896M）。

为了应对这两种限制，把每个内存节点的物理内存划分为3个管理区。

|  |  |
| --- | --- |
| 内存区域 | 说明 |
| ZONE\_DMA | 范围是0~16M，该区域的物理页面专门供I/O设备的DMA使用，某些工业标准体系结构中的(ISA)设备需要用到 |
| ZONE\_NORMAL | 范围是16~896M，该区域的物理页面可直接映射到内核（相差一个PAGE\_OFFSET即3G的偏移量，pa得到虚拟地址） |
| ZONE\_HIGHMEM | 范围是896~结束，系统中预留的可用内存空间, 不能被内核直接映射 |

高端内存页框分配只能用alloc\_pages和alloc\_page，不返回第一个被分配页框的线性地址，因为不存在，而是返回第一个被分配页框的页描述符的线性地址，总是存在的，因为所有页描述符一旦被分配在低端内存中，在内核初始化阶段就不会改变。内核线性地址空间的最后128M的一部分专门用于映射高端内存页框,通过永久内核映射、临时内核映射、非连续内存分配来将页框暂时映射到高端内存。

X86\_64，64位系统没有寻址问题，也不需要ZONE\_HIGHMEM；最开始的16M内存是ZONE\_DMA内存，用slab分配器的kmalloc分配获取；ZONE\_DMA32为16M~4G；高于4G的内存为ZONE\_NORMAL。

# 8、伙伴系统算法原理

将所有空闲页框分组成11个块链表，每个块链表分别包含大小为1,2,4,8,16,32,64,128,256,512,1024个连续的页框。每个块的第一个页框的物理地址是该块大小的整数倍。

假设请求一个256个页框的块。先在256个页框的链表中检查是否有一个空闲块，若没有，则查找下一个更大的页块，也就是在512个页框的链表中找一个空闲块；若存在，则把512的页框分为两份，一半满足要求，另一半插入256个页框的链表。若512个页框的块链表也没有找到空闲块，则继续寻找更大的块，在1024个页框块找；若存在，将256个页框满足请求，512个插入512个页框的链表，最后256插入256个页框的链表；若不存在，则放弃并发出错信号。

页框块的释放过程是上述的逆过程。满足一下条件的两个块为伙伴：

1. 两个块具有相同大小b；
2. 它们的物理地址连续；
3. 第一个块的第一个页框的物理地址是2\*b\*2^12的倍数。（因为要合并为一个块，而每个块的第一个页框的物理地址是该块大小的整数倍）。

该算法是迭代的，成功合并所释放的块，就会视图合并2b的块，以再次形成更大的块。当释放一个块时，先在其对应的链表中考查是否有伙伴存在，如果没有伙伴块，就直接把要释放的块挂入链表头；如果有，则从链表中摘下伙伴，合并成一个大块，然后继续考察合并后的块在更大一级链表中是否有伙伴存在，直到不能合并或者已经合并到了最大的块

# 9、释放一个page到伙伴系统后，这个page需要跟旁边的空闲page合并成更大的块，那么是跟左边的合并还是跟右边的合并呢？

释放函数核心是\_\_free\_one\_page，其中\_\_find\_buddy\_index是分析是否能够对此段页框附近的页框进行合并的，往前检查order次方个连续页框是否为空闲页框，再往后检查order次方个连续页框是否为空闲页框，如果其中一者成立，则合并，并order++，继续检查，但是注意，这些页框都必须为同一个管理区，因为伙伴系统是以管理区为单位的。

/\*

\* Locate the struct page for both the matching buddy in our

\* pair (buddy1) and the combined O(n+1) page they form (page).

\*

\* 1) Any buddy B1 will have an order O twin B2 which satisfies

\* the following equation:

\* B2 = B1 ^ (1 << O)

\* For example, if the starting buddy (buddy2) is #8 its order

\* 1 buddy is #10:

\* B2 = 8 ^ (1 << 1) = 8 ^ 2 = 10

\*

\* 2) Any buddy B will have an order O+1 parent P which

\* satisfies the following equation:

\* P = B & ~(1 << O)

\*

\* Assumption: \*\_mem\_map is contiguous at least up to MAX\_ORDER

\*/

static inline unsigned long

\_\_find\_buddy\_index(unsigned long page\_idx, unsigned int order)

{

return page\_idx ^ (1 << order);

}

伙伴页框号是page\_idx的第（1<<order）位的相反数, 比如(1<<order)是4，page\_idx/buddy\_idx是01110，则page\_idx/buddy\_idx是01010，由此可见伙伴并不一定是之后的区间。

# 10、Pthread\_create()和fork()有什么区别

都是用户态API，Pthread\_create是创建线程，调用clone系统调用，fork创建进程，调用fork系统调用，但最后都调用do\_fork，参数不同。

# 11、IDT表是如何被初始化的

当计算机还运行在实模式时，IDT被初始化并被BIOS例程使用。然而，一旦Linux接管，IDT就被移到RAM的另一个区域,进行第二次初始化。

1、在内核初始化过程中，setup\_idt()汇编语言函数用同一个中断门（指向ignore\_int()中断处理程序，可看做空处理程序，栈中保存寄存器内容，打印系统消息，从栈中恢复寄存器内容，执行iret指令恢复被中断程序）填充idt\_table表项（共256个）；

2、在系统初始化时start\_kernel()，中断向量表的初始化分为两个部分：

（1）对0~19号和0x80号系统保留中断向量的初始化，在trap\_init中完成

（2）对其它中断向量的初始化，在init\_IRQ中完成。init\_IRQ()函数通过替换由setup\_idt()所建立的中断门来更新IDT，主要在interrupt数组中找到用于建立中断门的中断处理程序地址。Interrupt数组中第n项存放IRQn的中断处理程序地址。数组包括NR\_IRQS宏产生的数为224（新晋I/O APIC芯片）或16（旧的8259A可编程控制器芯片）。

中断向量表的初始化：

手机屏幕的截图

描述已自动生成

一些文字和图片的手机截图

描述已自动生成

# 12、内核处理中断的完整过程

整体流程：

地图上有字

描述已自动生成

手机屏幕截图

描述已自动生成

中断响应流程

一些文字和图片的手机截图

描述已自动生成

第一阶段－－获取中断号

每个CPU都有响应中断的能力, 每个CPU响应中断时都走相同的流程. 这个流程就是内核提供的中断服务程序.

在进入中断服务程序时, CPU已经自动禁止了本CPU上的中断响应, 因为CPU不能假定中断服务程序是可重入的.  
中断处理程序的第一步要做两件事情:  
1. 将中断号压入栈中; (不同中断号的中断对应不同的中断服务程序入口)  
2. 将当前寄存器信息压入栈中; (以便中断退出时恢复)

显然, 这两步都是不可重入的(如果在保存寄存器值时被中断了, 那么另外的操作很可能就把寄存器给改写了, 现场将无法恢复), 所以前面说到的CPU进入中断服务程序时要自动禁止中断.栈上的信息被作为函数参数, 调用do\_IRQ函数.  
  
第二阶段－－中断串行化

进入do\_IRQ函数, 第一步进行中断的串行化处理, 将多个CPU同时产生的某一中断进行串行化. 其方法是如果当前中断处于"执行"状态(表明另一个CPU正在处理相同的中断), 则重新设置它的"触发"标记, 然后立即返回. 正在处理同一中断的那个CPU完成一次处理后, 会再次检查"触发"标记, 如果设置, 则再次触发处理过程.于是, 中断的处理是一个循环过程, 每次循环调用handle\_IRQ\_event来处理中断.  
  
第三阶段－－关中断条件下的中断处理

进入handle\_IRQ\_event函数, 调用对应的内核或内核模块通过request\_irq函数注册的中断处理函数.

注册的中断处理函数有个中断开关属性, 一般情况下, 中断处理函数总是在关中断的情况下进行的. 而调用request\_irq注册中断处理函数时也可以设置该中断处理函数在开中断的情况下进行, 这种情况比较少见, 因为这要求中断处理代码必须是可重入的. (另外, 这里如果开中断, 正在处理的这个中断一般也是会被阻塞的. 因为正在处理某个中断的时候, 硬件中断控制器上的这个中断并未被ack, 硬件不会发起下一次相同的中断.)

中断处理函数的过程可能会很长, 如果整个过程都在关中断的情况下进行, 那么后续的中断将被阻塞很长的时间.于是, 有了soft\_irq. 把不可重入的一部分在中断处理程序中(关中断)去完成, 然后调用raise\_softirq设置一个软中断, 中断处理程序结束. 后面的工作将放在soft\_irq里面去做.  
  
第四阶段－－开中断条件下的软中断

上一阶段循环调用完当前所有被触发的中断处理函数后, do\_softirq函数被调用, 开始处理软件中断.

在软中断机制中, 为每个CPU维护了一个若干位的掩码集, 每位掩码代表一个中断号. 在上一阶段的中断处理函数中, 调用raise\_softirq设置了对应的软中断, 到了这里, 软中断对应的处理函数就会被调用(处理函数由open\_softirq函数来注册).

软中断与中断的模型很类似, 每个CPU有一组中断号, 中断有其对应的优先级, 每个CPU处理属于自己的中断. 最大的不同是开中断与关中断.于是, 一个中断处理过程被分成了两部分, 第一部分在中断处理函数里面关中断的进行, 第二部分在软中断处理函数里面开中断的进行.

由于这一步是在开中断条件下进行的，这里还可能发生新的中断（中断嵌套），然后新中断对应的中断处理又将开始一个新的第一阶段~第三阶段。在新的这个第三阶段中，可能又会触发新的软中断。但是这个新的中断处理过程并不会进入第四阶段，而是当它发现自己是嵌套的中断时，完成第三阶段之后就会退出了。也就是说，只有第一层中断处理过程会进入第四阶段，嵌套发生的中断处理过程只执行到第三阶段。

然而嵌套发生的中断处理过程也可能会触发软中断，所以第一层中断处理过程在第四阶段需要是一个循环的过程，需要循环处理嵌套发生的所有软中断。因为这样可以按软中断触发的顺序来执行这些软中断，否则后来的软中断可能就会先执行完成了。

极端情况下，嵌套发生的软中断可能非常多，全部处理完可能需要很长的时间，于是内核会在处理完一定数量的软中断后，将剩下未处理的软中断推给一个叫ksoftirqd的内核线程来处理，然后结束本次中断处理过程。

第五阶段－－开中断条件下的tasklet

实际上, 软中断很少直接被使用. 而第三部分开中断情况下的进行的处理过程一般是由tasklet机制来完成的.

tasklet是由软中断引出的, 内核定义了两个软中断掩码HI\_SOFTIRQ和TASKLET\_SOFTIRQ(两者优先级不同), 这两个掩码对应的软中断处理函数作为入口, 进入tasklet处理过程.

于是, 在第三阶段的中断处理函数中, 完成关中断的部分后, 然后调用tasklet\_schedule/tasklet\_hi\_schedule标记一个tasklet, 然后中断处理程序结束.

后面的工作由HI\_SOFTIRQ/TASKLET\_SOFTIRQ对应的软中断处理程序去处理被标记的tasklet(每个tasklet在其初始化时都设置了处理函数).

看上去, tasklet只不过是在softirq的基础上多了一层调用, 其作用是什么呢? 前面说过, softirq是与CPU相对应的, 每个CPU处理自己的softirq. 这些softirq的处理函数需要设计为可重入的, 因为它们可能在多个CPU上同时运行. 而tasklet则是在多个CPU间被串行化执行的, 其处理函数不必考虑可重入的事情.

然而, softirq毕竟还是要比tasklet少绕点弯路, 所以少数实时性要求相对较高的处理过程还是在精心设计之后, 直接使用softirq了. 比如: 时钟中断处理过程, 网络发送/接收处理过程.

结尾阶段

CPU接收到中断以后, 以历以上五个阶段, 中断处理完成. 最后需要恢复第一阶段中被保存在栈上的寄存器信息. 中断处理结束.

# 13、网卡的中断服务程序正在执行过程中会被新的网卡中断打断么

中断可以被另一个优先级比自己高的中断打断，不能被同一级别（相同类型）或低级别的中断打断，更不能被软中断打断。

# 14、Linux支持哪些中断下半部，区别是什么

软中断、tasklet（软中断基础上实现）、工作队列。

1、软中断是在编译时静态分配；Tasklet的分配和初始化可以在运行时进行。

2、软中断（即使是同一种类型）可以并发运行在多个CPU，因此软中断是可重入函数并且必须明确使用自旋锁保护数据结构；相同类型tasklet总是被串行执行，不能再两个CPU上同时运行相同类型的tasklet，然而，类型不同的tasklet可以在多个CPU上并发执行，串行化使得tasklet函数不必是可重入的。

3、软中断和tasklet（可延迟函数）运行在中断上下文，不能阻塞，不能发生进程切换；工作队列运行在进程上下文，可以执行可阻塞函数，可以进程切换。

# 15、对/proc/interrupts和/proc/softirqs进行解读

读取interrupts会依次显示irq编号，每个cpu对该irq的处理次数，中断控制器的名字，设备名称（request\_irq的dev\_name字段）. /proc/interrupts 是一个 sequence file.，简单来说，就是通过代码生成的文件。对于/proc/interrupts,在读取这个文件时，系统会遍历 0 ～ nr\_irq (包含 nr\_irq)个中断号，对每个中断号都调用 show\_interrupts() 来获取该中断的信息。

图片包含 游戏机, 电脑

描述已自动生成

软中断类别，分别对应不同工作类型，同一种软中断在不同CPU上发生次数

电脑屏幕的照片上有文字

描述已自动生成

# 16、什么是内核抢占？系统中有哪些内核抢占时机？

如果进程正在执行内核函数时，即运行在内核态，此时允许内核切换（被切换的进程正执行内核函数的进程），这个内核就是抢占的。抢占内核的特点是一个运行在内核态的进程，可能在执行内核函数期间被另一个进程取代。如：执行异常处理的进程已经用完时间配额，如果内核是抢占的，进程可能会立即被取代，但如果内核是非抢占的，进程会继续运行直到它执行完异常处理程序或自动放弃CPU。

抢占时机：结束内核控制路径（通常是一个中断处理程序）;异常处理程序调用preempt\_enable()允许内核抢占;启用可延迟函数;一个任务在内核态中，显式的调用schedule()函数，任务主动放弃CPU使用权;一个任务在内核态中被阻塞，导致需要调用schedule()函数，任务主动放弃CPU使用权。

# 17、Linux内核有哪些同步机制?区别是?哪些同步机制可以用在中断处理函数?

1、每CPU变量

主要是数据结构的数组，系统中每个CPU对应数组的一个元素。一个CPU不应该访问与其他CPU对应的数组元素，可以随意读或修改自己的元素，而不用担心竞争条件。对来自不同CPU的并发访问提供保护，但是对异步函数（中断处理程序或可延迟函数）的访问不提供保护，此外，在单或多处理器中，内核抢占都可能使每CPU变量产生竞争条件。所以内核控制路径应该在禁止抢占的情况下访问每CPU变量.

2、原子操作

任何一个读-修改-写的操作都必须以单个指令执行，中间不能中断，且避免其他CPU访问同一存储单元。

3、优化和内存屏障

优化屏障保证编译程序不混淆放在原语操作前后的汇编语言指令，也就是barrier()宏，即asm volatile(“”:::”memory”)

内存屏障保证原语之后的操作开始执行之前，原语之前的操作已经完成，类似防火墙，让任何汇编语言指令都不能通过。汇编语言指令串行。Linux提供6个内存屏障原语：mb()/rmb()/wmb()/smp\_mb()/smp\_rmb()/smp\_wmb()。

4、自旋锁

发现锁由运行在另一个CPU上的内核控制路径锁着，就在周围旋转，反复执行一条紧凑的循环指令，直到锁被释放。忙等。基于原子操作。不会引起调用者睡眠。

5、读写自旋锁

把对共享资源的访问者划分成读者和写者，读者只对共享资源进行读访问，写者则需要对共享资源进行写操作。写者是排他性的，一个读写锁同时只能有一个写者或多个读者（与CPU数相关），但不能同时既有读者又有写者。

对比自旋锁：

* 这种锁相对于自旋锁而言，能提高并发性，因为在多处理器系统中，它允许同时有多个读者来访问共享资源，最大可能的读者数为实际的逻辑CPU数。
* 在读写锁保持期间也是抢占失效的。

读写锁访问规则:

1. 如果读写锁当前没有读者，也没有写者，那么写者可以立刻获得读写锁，否则它必须自旋在那里，直到没有任何写者或读者。
2. 如果读写锁没有写者，那么读者可以立即获得该读写锁，否则读者必须自旋在那里，直到写者释放该读写锁。

6、顺序锁

对读写锁的一种优化，对于顺序锁，读者绝不会被写者阻塞，也就说，读者可以在写者对被顺序锁保护的共享资源进行写操作时仍然可以继续读，而不必等待写者完成写操作，写者也不需要等待所有读者完成读操作才去进行写操作。但是，写者与写者之间仍然是互斥的，即如果有写者在进行写操作，其他写者必须自旋在那里，直到写者释放了顺序锁。

这种锁有一个限制，它必须要求被保护的共享资源不含有指针，因为写者可能使得指针失效，但读者如果正要访问该指针，将导致OOPs。   
如果读者在读操作期间，写者已经发生了写操作，那么，读者必须重新读取数据，以便确保得到的数据是完整的。

这种锁对于读写同时进行的概率比较小的情况，性能是非常好的，而且它允许读写同时进行，因而更大地提高了并发性。

7、RCU

被RCU保护的共享数据结构，读者不需要获得任何锁就可以访问它，但写者在访问它时首先拷贝一个副本，然后对副本进行修改，最后使用一个回调（callback）机制在适当的时机把指向原来数据的指针重新指向新的被修改的数据。内核每经过一个时钟滴答，就周期性检查本地CPU是否经过一个静止状态（CPU执行进程切换，CPU开始在用户态执行，CPU执行空循环），如果所有PCU都经过静止状态，本地tasklet就执行链表中的所有回调函数。被RCU保护的临界区，任何内核控制路径都不能睡眠。

8、信号量

类似自旋锁，当锁关闭时，不允许内核控制路径执行，然而，当内核控制路径试图获取内核信号量所保护的忙资源时，相应进程被挂起，只有资源被释放时，进程才再次变为可运行的。因此，只有可以睡眠的函数才能获取内核信号量，中断处理程序和可延迟函数都不能使用内核信号量。

9、读写信号量

类似读写自旋锁，有一点不同：在信号量再次变为打开之前，等待进程被挂起，而不是自旋。

10、补充原语

11、本地中断禁止

12、禁止和激活可延迟函数。

貌似除了不用down\_trylock的信号量（普通信号量的down，没有资源会睡眠，挂起），其他都可以用在中断处理程序？

# 18、获得自旋锁后可以发送内核抢占么？为什么？

由自旋锁保护的临界区域都是禁止内核抢占的。在单处理器上，此原语仅是禁止或启用内核抢占。在自旋锁忙等期间，内核抢占是有效的，因此，等待自旋锁释放的进程可能被更高优先级的进程替换。抢占式内核的spin\_lock宏会调用preempt\_disable()禁止内核抢占。

自旋锁有几个重要的特性：1、被自旋锁保护的临界区代码执行时不能进入休眠。2、被自旋锁保护的临界区代码执行时是不能被被其他中断中断。3、被自旋锁保护的临界区代码执行时，内核不能被抢占。从这几个特性可以归纳出一个共性：被自旋锁保护的临界区代码执行时，它不能因为任何原因放弃处理器。

如果获取自旋锁后还允许内核抢占，会让另外进程一直忙等，浪费资源，而且可能发生死锁（一个线程获取了一个锁，但是被中断处理程序打断，中断处理程序也获取了这个锁（但是之前已经被锁住了，无法获取到，只能自旋），中断无法退出，导致线程中后面释放锁的代码无法被执行，导致死锁）。

# 19、内核是如何维持墙钟和单调时钟的？系统睡眠唤醒对单调时钟是否有影响？设置系统时间对单调时钟是否有影响？

# 20、Linux中的高精度定时器和低精度定时器分别是如何组织的？

# 21、malloc()分配的物理内存是连续的吗，kmalloc()呢，为什么？

Malloc分配的物理内存不连续。malloc能够申请的空间大小与物理内存的大小没有直接关系，仅与程序的虚拟地址空间相关。程序运行时，堆空间只是程序向操作系统申请划出来的一大块虚拟地址空间。应用程序通过malloc申请空间，得到的是在虚拟地址空间中的地址，之后程序运行所提供的物理内存是由操作系统完成的。当程序的第一条指令执行时，发生缺页中断，操作系统负责分配物理内存，并建立虚拟内存和物理内存之间的映射关系。因此，不管是在32位的Windows下还是Linux下，如果真实内存只有1G，mallo(1.2G)也都会分配成功（将已经malloc出来的2G用户空间，依次填满内容呢？那么会在实际内存用完以后，在 do\_anonymous\_page() 中抛出oom，杀死调用该malloc的进程（oom\_kill\_process））。malloc之后尚未分配物理页，只有当真正写这段虚拟内存时（cpu检查到pte的present标志位，即相应的物理页不在内存），触发缺页中断（软中断），系统才会分配对应的实际物理页。一旦物理页分配完，会不会采用 swap out/in 机制，要看具体 linux config。

kmalloc分配的物理内存连续。如果不涉及对象缓存, 而是传统意义上的分配内存，则调用kmalloc。kmalloc函数可获取以字节为单位的一块内核内存. 所分配的内存区在物理上是连续的. kmalloc()分配的内存处于3GB~highmemory之间，这段内核空间与物理内存的映射一一对应，只相差一个PAGE\_OFFSET即3G；kmalloc分配的内存虚拟物理地址均连续。一般情况下，只有硬件设备才需要物理地址连续的内存，因为硬件设备往往存在于MMU之外，根本不了解虚拟地址；为了性能上的考虑内核中一般使用 kmalloc()；一般情况下在驱动程序中都是调用kmalloc()来给数据结构分配内存。

# 22、缺页异常处理过程是怎样的

在实际需要某个虚拟内存区域数据之前，虚拟和物理内存之间关联不会建立。如果进程访问的虚拟地址空间尚未与页帧关联，处理器自动引发一个缺页异常，内核必须处理该异常

图片包含 游戏机, 仪表, 钟表

描述已自动生成

手机屏幕截图

描述已自动生成

# 23、echo m > /proc/sysrq-trigger可以将内存信息比较详细的输出到dmesg中，请解读

# 24、ps可以看到内核线程的内存使用都是0，为什么

# 25、top命令中看到的CPU核在各种状态的百分比是如何计算的

# 26、gdb -p process\_A\_pid跟踪上进程A后可以修改进程A中变量的值，如何做到的？

# 27、系统调用流程是怎样的？x86上有哪些系统调用方式？当前x86\_64使用的是哪种系统调用方式？

# 28、主流的N路组相连的CPU cache是怎么回事？

# 29、x86的CPU有哪些主要cache？哪些是socket共享的，哪些是core共享的？