|  |  |  |  |
| --- | --- | --- | --- |
| **宏名称** | **宏位置** | **宏定义** | **宏作用** |
| **ROUND(freemem,align)** | types.h | #define ROUND(a, n) (((((u\_long)(a))+(n)-1)) & ~((n)-1)) | 以align对齐freemem  Example:freemem=0x80103111  align=0x1000  对齐后freemem=0x80104000  (这里align必须为2的幂次) |
| **bzero(void\*, size\_t)** | mmu.h | void \*max;  max = b + len;  while (b+3 < max)  {  \*(int \*)b = 0;  b+=4;  }  while (b < max)  {  \*(char\*)b++ = 0;  } | bzero函数的主要功能就是用来清零，我们可以发现，实际上在空间大于4的时候使用强制转换为int类型的指针来进行清零；当空间小于4时，则使用char型指针清零。因为int是4个字节，而char是1个字节。 |
| **Pde \*,Pte \*** | mmu.h | typedef u\_long Pde;  typedef u\_long Pte; | Pde的原形其实就是u\_long,u\_long即unsigned long,无符号整数,占4个字节。其实选择unsigned long,4个字节,而指针是要寻找到4G虚拟空间的,所以指针也是4个字节,所以在这一点上来说,指针和u\_long的互相转换是完全没有问题的。 |
| **PDX(va),PTX(va)** | mmu.h | #define PDX(va) ((((u\_long)(va))>>22) & 0x03FF)  #define PTX(va) ((((u\_long)(va))>>12) & 0x03FF) | PDX宏实际上就是提取va的页目录项索引,所以要右移22位。再和低12位为1的0x03FF&一下,即可得到页目录索引。PTX宏实际上是用来提取页表项的索引，向右移动12位后低10位则是页表项的索引。 |
| **PTE\_ADDR(pte)** | mmu.h | #define PTE\_ADDR(pte) ((u\_long)(pte)&~0xFFF) | PTE\_ADDR宏是用来将pte的值后12位清零。因为每一页的基址都应该是后12位为零的，不论是页表，页目录，或者是普通的页。 |
| **KADDR(pa),PADDR(kva)** | mmu.h | #define PADDR(kva)  ({ u\_longa=(u\_long)(kva)  if(a<ULIM)  panic("PADDRcalled with invalidkva %08lx", a);  a-ULIM;  })  #define KADDR(pa)  ({  u\_long ppn = PPN(pa); if(ppn>=npage) panic("KADDRcalled with invalidpa%08lx",(u\_long)pa);  (pa) + ULIM;  })  #define PPN(va) (((u\_long)(va))>>12) | KADDR和PADDR的功能相反，KADDR是给定一个物理地址，获得其虚拟地址，如果给定的物理地址超过最大物理内存的话则报错；PADDR是给定一个内核虚拟地址，获得其物理地址，如果给定的虚拟地址超过ULIM的话则报错。实际上这里的ULIM的值就是0x80000000，即内核与用户区的分界线。 |
| **PTE\_V,PTE\_R** | mmu.h | #define PTE\_V 0x0200  #define PTE\_R 0x0400 | PTE\_V的功能实际上如注释所说，是Valid bit,而PTE\_R是dirty bit,这两者实际上对应的十六进制数均为在低12位中某一位为1的一个二进制数。产生的原因则是因为我们在页目录项和页表项中存入的物理地址只有前20位是有效的地址，因为我们操作系统从0开始按页分配，1页是4KB，所以一页的物理地址的基址后12位始终为0。而我们在页目录项和页表项中存入的最终却是32位的二进制数，一方面是考虑到了字节对齐，一方面在后12位中加入了一些flag位来表明其目前的状态。 |
| **LIST\_INIT (&page\_free\_list);** | queue.h | #define LIST\_INIT(head) do {  LIST\_FIRST((head))=NULL;  } while (0)  LIST\_FIRST((head)): #define LIST\_FIRST(head) ((head)->lh\_first)  head: #define LIST\_HEAD(name, type)  struct name {  struct type \*lh\_first; /\* first element \*/  }  LIST\_HEAD(Page\_list, Page); | 空闲物理页面链表是双向链表。head为指向srtuct Page\_list结构体的首指针，这个宏定义的意思是初始化首指针中的lh\_first的值为0。 |
| **LIST\_INSERT\_HEAD(&page\_free\_list,&pages[i],pp\_link)** | queue.h | #defineLIST\_INSERT\_HEAD(head, elm, field) do {  if ((LIST\_NEXT((elm), field) = LIST\_FIRST((head))) != NULL) LIST\_FIRST((head))->field.le\_prev = &LIST\_NEXT((elm),field);  LIST\_FIRST((head))=(elm);  (elm)->field.le\_prev = &LIST\_FIRST((head));  } while (0) | head是头指针，elm由两部分组成：一个是引用计数，一个是指针域。field是指elm里的指针域。实际上这个宏就是在头结点前面插入结点的一个宏。其中尤其值得注意的就是如果一开始head为空的话，那么就会自动帮你建立一个头指针，之后再进行结点的插入，每次插入的结点都会成为新的头结点。 |
| **LIST\_REMOVE (LIST\_FIRST(&page\_free\_list),pp\_link);** | queue.h | #define LIST\_REMOVE(elm, field) do {  if (LIST\_NEXT((elm), field) != NULL) LIST\_NEXT((elm), field)->field.le\_prev =  (elm)->field.le\_prev;  \*(elm)->field.le\_prev = LIST\_NEXT((elm),field);  } while (0) | struct {  struct type \*le\_next; /\* next element \*/  struct type \*\*le\_prev; /\* address of previous next element \*/  }  LIST\_NEXT((elm),field)是((elm)->field.le\_next)，(elm)->field.le\_next就是下一个元素，具体过程如图所示。（可以拿出来看） |
| **page2pa(pageTablePage)** | pmap.h | static inline u\_long  page2pa(struct Page \*pp)  {  return  page2ppn(pp)<<PGSHIFT;  }  page2ppn:  static inline u\_long  page2ppn(struct Page \*pp)  {  return pp - pages;  }  PGSHIFT:  #define PGSHIFT 12 | 首先用pp-pages得到的是物理页框号，因为一页对应一个4B的物理地址，对应一个页框号。那么页框号左移12位其实得到的就是物理地址。 |