**内存管理get\_free\_page()函数分析**

2014年11月11日 20:57:14 [linpeng1577](https://me.csdn.net/linpeng12358) 阅读数：2563

 版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/linpeng12358/article/details/41017961

Linux-0.11内存管理模块是源代码中比较难以理解的部分，现在把笔者个人的理解发表  
先发Linux-0.11内核内存管理get\_free\_page()函数分析

有时间再写其他函数或者文件的:)

/\*

 \*Author  : DavidLin

 \*Date    : 2014-11-11pm

 \*Email   : linpeng1577@163.com or linpeng1577@gmail.com

 \*world   : the city of SZ, in China

 \*Ver     : 000.000.001

 \*history :     editor      time            do

            1)LinPeng       2014-11-11      created this file!

            2)

 \*/

以下是Linus的源码：

*/\**

*\* Get physical address of first (actually last :-) free page, and mark it*

*\* used. If no free pages left, return 0.*

*\*/*

unsigned long get\_free\_page(void)

{

register unsigned long \_\_res asm("ax");

\_\_asm\_\_("std ; repne ; scasb\n\t"

"jne 1f\n\t"

"movb $1,1(%%edi)\n\t"

"sall $12,%%ecx\n\t"

"addl %2,%%ecx\n\t"

"movl %%ecx,%%edx\n\t"

"movl $1024,%%ecx\n\t"

"leal 4092(%%edx),%%edi\n\t"

"rep ; stosl\n\t"

"movl %%edx,%%eax\n"

"1:"

:"=a" (\_\_res)

:"0" (0),"i" (LOW\_MEM),"c" (PAGING\_PAGES),

"D" (mem\_map+PAGING\_PAGES-1)

:"di","cx","dx");

return \_\_res;

}

1.函数目的：  
    寻找mem\_map[0..(PAGING\_PAGES-1)]中的空闲项，即mem\_map[i]==0的项，如果找到，  
    就返回物理地址，找不到返回0  
2.技巧：  
    这段代码为何用C嵌套汇编实现？  
    笔者个人认为C函数会开辟栈帧，可能会污染任务堆栈，  
    同时该函数需要经常频繁的调用，汇编中，寄存器级别的汇编指令操作的效率比C更高:)  
3.代码分析：  
(1)register unsigned long \_\_res asm("ax");  
    \_\_res是寄存器级变量，值保存在ax寄存器中,就是说对\_\_res的操作等于ax寄存器的操作，为效率考虑  
(2)\_\_asm\_\_("std ; repne ; scasb\n\t"  
    循环比较，找出mem\_map[i]==0的页;  
    std设置DF=1，所以scasb执行递减操作，涉及寄存器al, ecx, es:(e)di三个寄存器，在函数尾部的定义中  
    :"0" (0),"i" (LOW\_MEM),"c" (PAGING\_PAGES),  
    "D" (mem\_map+PAGING\_PAGES-1)  
    :"di","cx","dx");  
    即有

    al       = 0;    //如果mem\_map[i] == 0,表示为空闲页，否则为已分配占用,al保存0值，用于比较

    ecx    = PAGING\_PAGES;                                 //主内存叶表个数

    es:di =  (mem\_map+PAGING\_PAGES-1);   //内存管理数组最后一项  
    这句指令的意思是从数组mem\_map[0..(PAGING\_PAGES-1)]的最后一项  
    mem\_map[PAGING\_PAGES-1]开始，比较mem\_map[i]是否等于0(0值保存在al寄存器中);  
    每比较一次,es:di值减1,如果不相等,es:di值减1,即mem\_map[i--],继续比较,直到ecx == 0;  
    如果相等，则跳出循环

    C语言实现如下:

1. ......
2. index\_ = 0;
4. for (i = PAGING\_PAGES-1; i != 0; i--)
5. {
6. if(0 != mem\_map[i]) {
7. continue; *//继续循环*
8. }
9. else {
10. index\_ = i; *//跳出循环*
11. break;
12. }
13. }
15. if(0 == index\_) {
16. goto Label\_1;
17. }
19. Label\_1:
20. return index\_;
21. ......

(3)"jne 1f\n\t"  
    如果mem\_map[0..(PAGING\_PAGES-1)]均不等于0,  
    跳转到标签1f处执行,Nf表示向前标签,Nb表示向后标签,N是取值1-10的十进制数字  
(4)"movb $1,1(%%edi)\n\t"  
    mem\_map[i]==0是mem\_map[0..(PAGING\_PAGES-1)]中逆序第一个找到的等于0的目标，  
    将edi的最低位置1，即mem\_map[i]=1,标志为该页已被占用，不是空闲位  
(5)"sall $12,%%ecx\n\t"  
    此时ecx保存的是mem\_map[i]的下标i,即相对页面数,  
    举例:  
        假设mem\_map[0..(PAGING\_PAGES-1)]最后一个参数  
    mem\_map[PAGING\_PAGES-1] == 0，即i == (PAGING\_PAGES-1),  
    所以此时\*ecx == PAGING\_PAGES-1;  
    此时相对页面地址是4k\*(PAGING\_PAGES-1),  
    每一页1024个4字节物理页,左移12位等于4096(2的12次方),  
(6)    "addl %2,%%ecx\n\t"  
    加上低端内存地址，得到实际物理地址  
    %2等于LOW\_MEM，在如下语句中定义  
    "0" (0),"i" (LOW\_MEM),"c" (PAGING\_PAGES),  
    提问：  
        为什么4k\*(PAGING\_PAGES-1)不是实际物理地址呢？  
        答案是初始化的时候如下:  
        mem\_map[0..(PAGING\_PAGES)]是主内存管理数组  
        管理的只是1-16M的空间，即PAGING\_MEMORY = ((16-1)\*1024\*1024)

        不包括0-1M(0-1M,其实是0-640K已经被内核占用)

#define LOW\_MEM 0x100000

#define PAGING\_MEMORY (15\*1024\*1024)

#define PAGING\_PAGES (PAGING\_MEMORY>>12)

#define MAP\_NR(addr) (((addr)-LOW\_MEM)>>12)

void mem\_init(long start\_mem, long end\_mem)

{

int i;

HIGH\_MEMORY = end\_mem;

for (i=0 ; i<PAGING\_PAGES ; i++){

mem\_map[i] = USED;

}*//所有主内存区初始化为被占用*

i = MAP\_NR(start\_mem);

end\_mem -= start\_mem;

end\_mem >>= 12;

while (end\_mem-->0)

mem\_map[i++]=0;

}

(7)"movl %%ecx,%%edx\n\t"  
    将ecx寄存器的值保存到edx寄存器中，即将实际物理地址保存到edx寄存器中。  
(8)"movl $1024,%%ecx\n\t"  
    将1024保存到ecx寄存器中，因为每一页占用4096字节(4K),  
    实际物理内存,每项占用4字节,有1024项。  
(9)"leal 4092(%%edx),%%edi\n\t"  
    因为按照4字节对齐，所以每项占用4字节,  
    取当前物理页最后一项4096 = 4096-4 = 1023\*4 = (1024-1)\*4 。  
    将该物理页面的末端保存在edi寄存器中,  
    即ecx+4092处的地址保存在edi寄存器中。  
(10)"rep ; stosl\n\t"  
    从ecx+4092处开始，反方向，步进4，重复1024次，  
    将该物理页1024项全部填入eax寄存器的值，  
    在如下代码定义中，eax初始化为0(al=0,eax =0,ax =0)  
    :"0" (0),"i" (LOW\_MEM),"c" (PAGING\_PAGES),  
    所以该物理页1024项全部清零。  
(11)"movl %%edx,%%eax\n"  
    将该物理页面起始地址放入eax寄存器中，  
    Intel的EABI规则中，  
    eax寄存器用于保存函数返回值  
(12)"1:"  
    标签1，用于"jne 1f\n\t"语句跳转返回0值，  
    注意：  
        eax寄存器只在"movl %%edx,%%eax\n"中被赋值，  
        eax寄存器初始值是'0'，如果跳转到标签"1:"处，  
        返回值是0，表示没有空闲物理页。  
(13):"=a" (\_\_res)  
    输出寄存器列表，这里只有一个，其中a表示eax寄存器  
(14):"0" (0),"i" (LOW\_MEM),"c" (PAGING\_PAGES),  
    "0"表示与上面同个位置的输出相同的寄存器，即"0"等于输出寄存器eax，  
    即eax既是输出寄存器，同时也是输入寄存器，  
    当然，在时间颗粒度最小的情况小，eax不能同时作为输入或者输出寄存器，  
    只能作为输入或者输出寄存器;  
  
    "i" (LOW\_MEM)是%2，从输出寄存器到输入寄存器依次编号%0，%1，%2.....%N,  
    其中"i"表示立即数，不是edi的代号，edi的代号是"D";  
  
    "c" (PAGING\_PAGES)表示将ecx寄存器存入PAGING\_PAGES，  
    ecx寄存器代号"c"。  
  
(15)"D" (mem\_map+PAGING\_PAGES-1)  
    "D"使用edi寄存器，即edi寄存器保存的值是(mem\_map+PAGING\_PAGES-1)  
    即%%edi = &mem\_map[PAGING\_PAGES-1]。  
(16):"di","cx","dx");  
    保留寄存器，告诉编译器"di","cx","dx"三个寄存器已经被分配，  
    在编译器编译中，不会将这三个寄存器分配为输入或者输出寄存器。  
(17)return \_\_res;  
    返回\_\_res保存的值，  
    相当于汇编的ret，隐含将eax寄存器返回，  
    C语言中是显式返回。  
  
4.汇编指令及语法规则解析，参照Intel官方文档《Volume 2A Instruction Set Reference (A-M)》  
《Volume 2B Instruction Set Reference (N-Z)》,GNU汇编规则  
(1)std:  
    主要将ESI and/or EDI方向设置为递减，对应cld(用于方向设置为递增)  
    1)Operation  
    Sets the DF flag in the EFLAGS register. When the DF flag is set to 1, string operations  
    decrement the index registers (ESI and/or EDI).  
    This instruction’s operation is the same in non-64-bit modes and 64-bit mode.  
    2)Operation  
    DF -> 1;  
(2)repne:  
    1)Description  
    Repeats a string instruction the number of times specified in the count register or  
    until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE  
    (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and  
    REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of  
    the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS,  
    and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be  
    added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous  
    forms of the REPE and REPNE prefixes, respectively.) The behavior of the REP  
    prefix is undefined when used with non-string instructions.  
    The REP prefixes apply only to one string instruction at a time. To repeat a block of  
    instructions, use the LOOP instruction or another looping construct. All of these  
    repeat prefixes cause the associated instruction to be repeated until the count in  
    register is decremented to 0. See Table 4-13.

    2)Operation

IF AddressSize = 16

THEN

Use CX for CountReg;

ELSE IF AddressSize = 64 and REX.W used

THEN Use RCX for CountReg; FI;

ELSE

Use ECX for CountReg;

FI;

WHILE CountReg = 0

DO

Service pending interrupts (if any);

Execute associated string instruction;

CountReg <- (CountReg – 1);

IF CountReg = 0

THEN exit WHILE loop; FI;

IF (Repeat prefix is REPZ or REPE) and (ZF = 0)

or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)

THEN exit WHILE loop; FI;

OD;

(3)scasb:  
    GNU汇编  
    在汇编语言中SCASB是一条字符串操作指令，源自“SCAN String Byte”的缩写。该指令的具体操作是 ：

*---------------------------------------------------------------------------------------*

Code | Mnemonic | Description

*---------------------------------------------------------------------------------------*

AE | SCAS m8 | Compare AL with byte at ES:(E)DI and set status flags

*---------------------------------------------------------------------------------------*

AF | SCAS m16 | Compare AX with word at ES:(E)DI and set status flags

*---------------------------------------------------------------------------------------*

AF | SCAS m32 | Compare EAX with doubleword at ES(E)DI and set status flags

*---------------------------------------------------------------------------------------*

AE | SCASB | Compare AL with byte at ES:(E)DI and set status flags

*---------------------------------------------------------------------------------------*

AF | SCASW | Compare AX with word at ES:(E)DI and set status flags

*---------------------------------------------------------------------------------------*

AF | SCASD | Compare EAX with doubleword at ES:(E)DI and set status flags

*---------------------------------------------------------------------------------------*

    计算 AL - byte of [ES:EDI] , 设置相应的标志寄存器的值；  
    修改寄存器EDI的值：如果标志DF为0，则 inc EDI；如果DF为1，则 dec EDI。  
    SCASB指令常与循环指令REPZ/REPNZ合用。例如，REPNZ SCASB 语句表示当 寄存器ECX>0 且 标志寄存器ZF=0，则再执行一次SCASB指令。  
    比较寄存器AL的值不相等则重复查找的字  
(4)sall  
    如sall $12, %ecx.  
    这个指令是算法左移,相当于c语言中的左移操作符<<.  
    intel汇编指令中的SAL,(Shit Arithmetic left).  
    根据AT&T的语法规则，  
    因为是一个长型的操作(ecx),  
    所以在intel汇编指令sal上加一个"l",  
    即转换成sall。  
(5)stosl  
    STOSL指令相当于将EAX中的值保存到ES:EDI指向的地址中，  
    若设置了EFLAGS中的方向位置位(即在STOSL指令前使用STD指令)  
    则EDI自减4，否则(使用CLD指令)EDI自增4。

(6)eax,ax,ah,al

00000000 00000000 00000000 00000000

|*===============EAX===============|--32个0,4个字节，2个字，1个双字*

|*======AX=======|--16个0,2个字节，1个字*

|==AH*===|-----------8个0,1个字节*

|*===AL==|---8个0,1个字节*

        EAX是32位的寄器，只是在原有的8086CPU的寄存器AX上增加了一倍的数据位数。

        故而EAX与AX根本不可能独立，二者是整体与部分的关系。  
        对EAX直接赋值，若更改了低16位自然会改变了AX值，  
        而AX又可以影响EAX整体。而AH,AL寄存器和AX之间的关系也是如此。  
  
转载请注明出处，谢谢:-)  
林鹏！加油，向李云和陈皓看齐！  
MyBlog   : http://blog.csdn.net/linpeng12358  
MyMail   : linpeng1577@163.com or linpeng1577@gmail.com  
MyGithub : DavilLin1577  
    welcome everybody!  
    :-)