#ant
The dreams in which I'm dying are the best I've ever had...

导航
C++博客
首页
联系
聚合 XML
管理

统计信息
随笔 - 11
文章 - 0
评论 - 235
Trackbacks - 0

常用链接
我的随笔
我的评论
我参与的随笔

留言簿(9)
给我留言
查看公开留言
查看私人留言

随笔分类
C++(6) (RSS)
Design Pattern(2) (RSS)
Encrypt(2) (RSS)
The Annotated CRT Sources(1) (RSS)

随笔档案

strlen源码剖析

学习高效编程的有效途径之一就是阅读高手写的源代码，*CRT(C/C++ Runtime Library)*作为底层的函数库，实现必然高效。恰好手中就有glibc和VC的CRT源代码，于是挑了一个相对简单的函数strlen研究了一下，并对各种实现作了简单的效率测试。

strlen的函数原形如下：

```
size_t strlen(const char *str);
```

strlen返回str中字符的个数，其中str为一个以'\0'结尾的字符串(a null-terminated string)。

1. 简单实现

如果不管效率，最简单的实现只需要4行代码：

```
1 size_t strlen_a(const char * str) {
2     size_t length = 0 ;
3     while (*str++ )
4         ++ length;
5     return  length;
6 }
```

也许可以稍加改进如下：

```
1 size_t strlen_b(const char * str) {
2     const char *cp =  str;
3     while (*cp++ )
```

2007年10月 (2)
2007年9月 (4)
2007年8月 (5)

最新随笔

1. strlen源码剖析
2. TEA加密算法的C/C++实现
3. Win32 Service的简单封装
4. MD5算法的C++实现
5. 非完美C++ Singleton实现[2]
6. 非完美C++ Singleton实现[1]
7. 从Win32 API封装Thread类[2]
8. C++&Win32写的空当接龙
9. 为什么不要特化函数模版
10. 从Win32 API封装Thread类[1]
11. 考虑用Macro替换Uncopyable

搜索

搜索

积分与排名

积分 - 111353
排名 - 140

最新评论 XML

1. re: MD5算法的C++实现
评论内容较长,点击标题查看
--KomeijiKuroko
2. re: MD5算法的C++实现
你好，我想请问一下，对于字节串如何按512位分块，需要先转化为对应的二进制编码吗
--snoopy
3. re: MD5算法的C++实现

```
4          ;
5      return (cp - str - 1 );
6  }
```

2. 高效实现

很显然，标准库的实现肯定不会如此简单，上面的strlen_a以及strlen_b都是一次判断一个字符直到发现'\0'为止，这是非常低效的。比较高效的实现如下(在这里WORD表示计算机中的一个字，不是WORD类型)：

(1) 一次判断一个字符直到内存对齐，如果在内存对齐之前就遇到'\0'则直接return，否则到(2)；
(2) 一次读入并判断一个WORD，如果此WORD中没有为0的字节，则继续下一个WORD，否则到(3)；
(3) 到这里则说明WORD中至少有一个字节为0，剩下的就是找出第一个为0的字节的位置然后return。

NOTE：

数据对齐(*data alignment*)，是指数据所在的内存地址必须是该数据长度的整数倍，这样CPU的存取速度最快。比如在32位的计算机中，一个WORD为4 byte，则WORD数据的起始地址能被4整除的时候CPU的存取效率比较高。CPU的优化规则大概如下：对于n字节(n = 2,4,8...)的元素，它的首地址能被n整除才能获得最好的性能。

为了便于下面的讨论，这里假设所用的计算机为32位，即一个WORD为4个字节。下面给出在32位计算机上的C语言实现(假设unsigned long为4个字节)：

```
1  typedef unsigned long  ulong;
2
3  size_t strlen_c(const char * str) {
4
5      const char * char_ptr;
6      const ulong * longword_ptr;
7      register ulong longword, magic_bits;
8
9      for (char_ptr =  str; ((ulong)char_ptr
10         & (sizeof(ulong) - 1)) != 0 ;
11         ++ char_ptr) {
12         if (*char_ptr == '\0' )
13             return char_ptr -  str;
14     }
```

想请问一下，那个我该怎么在主函数中调用MD5算法

--Dan

4. re: TEA加密算法的C/C++实现[未登录]

为什么不能run的？

是有什么问题吗？

这应该是 c++ 吧

--jk

5. re: 从Win32 API封装Thread类[1][未登录]

这里有个问题, 是 如果你没有join的话, 线程对象就会被马上析构, 程序出现崩溃.

所以我把线程对象创建在堆上面了, 然后在Run调用完之后在threadProc里面delete 完成

--Jackie

6. re: MD5算法的C++实现

评论内容较长,点击标题查看

--WWE

7. re: MD5算法的C++实现

评论内容较长,点击标题查看

--xwj

8. re: MD5算法的C++实现

学习了！很多地方看不太懂，收藏起来了。实在看不懂就直接用啦！

--aking

9. re: C++&Win32写的空当接龙

ai.dll这个文件 是windos自带的嘛

--rgb123

10. re: 非完美C++ Singleton实现[2]

评论内容较长,点击标题查看

```
15
16      longword_ptr = (ulong* )char_ptr;
17
18      magic_bits = 0x7efefeffL ;
19
20      while (1 ) {
21
22          longword = *longword_ptr++ ;
23
24          if ((((longword + magic_bits) ^ ~longword) & ~magic_bits) != 0 ) {
25
26              const char *cp = (const char*)(longword_ptr - 1 );
27
28              if (cp[0] == 0 )
29                  return cp -  str;
30              if (cp[1] == 0 )
31                  return cp - str + 1 ;
32              if (cp[2] == 0 )
33                  return cp - str + 2 ;
34              if (cp[3] == 0 )
35                  return cp - str + 3 ;
36          }
37      }
38  }
```

3. 源码剖析

上面给出的C语言实现虽然不算特别复杂，但也值得花点时间来弄清楚，先看9-14行：

```
for (char_ptr = str; ((ulong)char_ptr & (sizeof(ulong) - 1)) != 0; ++char_ptr) {
    if (*char_ptr == '\0')
        return char_ptr - str;
}
```

--SYBILRobbins

阅读排行榜
1. MD5算法的C++实现(44284)
2. strlen源码剖析(23403)
3. TEA加密算法的C/C++实现(11725)
4. 非完美C++ Singleton实现[1](4055)
5. 非完美C++ Singleton实现[2](4014)
6. 从Win32 API封装Thread类[1](3520)
7. Win32 Service的简单封装(2529)
8. 从Win32 API封装Thread类[2](2474)
9. C++&Win32写的空当接龙(2468)
10. 为什么不要特化函数模版(2348)

上面的代码实现了数据对齐，如果在对齐之前就遇到'\0'则可以直接return char_ptr - str;

第16行将longword_ptr指向数据对齐后的首地址

```
longword_ptr = (ulong*)char_ptr;
```

第18行给magic_bits赋值(在后面会解释这个值的意义)

```
magic_bits = 0x7efefeffL;
```

第22行读入一个WORD到longword并将longword_ptr指向下一个WORD

```
longword = *longword_ptr++;
```

第24行的if语句是整个算法的核心，该语句判断22行读入的WORD中是否有为0的字节

```
if ((((longword + magic_bits) ^ ~longword) & ~magic_bits) != 0)
```

if语句中的计算可以分为如下3步：

(1) longword + magic_bits

其中magic_bits的二进制表示如下：

```
                  b3       b2        b1        b0
              31---------------------------------->0
  magic_bits: 01111110 11111110 11111110 11111111
```

magic_bits中的31,24,16,8这些bits都为0，我们把这几个bits称为holes，注意在每个byte的左边都有一个hole。

检测0字节：

如果longword 中有一个字节的所有bit都为0，则进行加法后，从这个字节的右边的字节传递来的进位都会落到这个字节的最低位所在的hole上，而从这个字节的最高位则永远不会产生向左边字节的hole的进位。则这个字节左边的hole在进行加法后不会改变，由此可以检测出0字节；相反，如果longword中所有字节都不为0，则每个字节中至少有1位为1，进行加法后所有的hole都会被改变。

为了便于理解，请看下面的例子：

```
                  b3       b2        b1        b0
              31---------------------------------->0
```

```
  longword:   XXXXXXXX XXXXXXXX 00000000 XXXXXXXX
+ magic_bits: 01111110 11111110 11111110 11111111
```

上面longword中的b1为0，X可能为0也可能为1。因为b1的所有bit都为0，而从b0传递过来的进位只可能是0或1，很显然b1永远也不会产生进位，所以加法后longword的第16 bit这个hole不会变。

(2) ^ ~longword

这一步取出加法后longword中所有未改变的bit。

(3) & ~magic_bits

最后取出longword中未改变的hole，如果有任何hole未改变则说明longword中有为0的字节。

根据上面的描述，如果longword中有为0的字节，则if中的表达式结果为非0，否则为0。

NOTE:

如果b3为10000000，则进行加法后第31 bit这个hole不会变，这说明我们无法检测出b3为10000000的所有WORD。值得庆幸的是用于strlen的字符串都是ASCII标准字符，其值在0-127之间，这意味着每一个字节的第一个bit都为0。因此上面的算法是安全的。

一旦检测出longword中有为0的字节，后面的代码只需要找到第一个为0的字节并返回相应的长度就OK:

```
const char *cp = (const char*)(longword_ptr - 1);

if (cp[0] == 0)
    return cp - str;
if (cp[1] == 0)
    return cp - str + 1;
if (cp[2] == 0)
    return cp - str + 2;
if (cp[3] == 0)
    return cp - str + 3;
```

4. 另一种实现

```
1 size_t strlen_d(const char *str) {
2
```

```
 3     const char *char_ptr;
 4     const ulong *longword_ptr;
 5     register ulong longword, himagic, lomagic;
 6
 7     for (char_ptr = str; ((ulong)char_ptr
 8         & (sizeof(ulong) - 1)) != 0;
 9         ++char_ptr) {
10         if (*char_ptr == '\0')
11             return char_ptr - str;
12     }
13
14     longword_ptr = (ulong*)char_ptr;
15
16     himagic = 0x80808080L;
17     lomagic = 0x01010101L;
18
19     while (1) {
20
21         longword = *longword_ptr++;
22
23         if (((longword - lomagic) & himagic) != 0) {
24
25             const char *cp = (const char*)(longword_ptr - 1);
26
27             if (cp[0] == 0)
28                 return cp - str;
29             if (cp[1] == 0)
30                 return cp - str + 1;
31             if (cp[2] == 0)
32                 return cp - str + 2;
33             if (cp[3] == 0)
34                 return cp - str + 3;
35         }
```

```
36     }
37 }
```

上面的代码与strlen_c基本一样，不同的是：
magic_bits换成了himagic和lomagic

```
himagic = 0x80808080L;
lomagic = 0x01010101L;
```

以及 if语句变得比较简单了

```
if (((longword - lomagic) & himagic) != 0)
```

if语句中的计算可以分为如下2步：

(1) longword - lomagic

himagic和lomagic的二进制表示如下：

```
              b3      b2       b1       b0
           31--------------------------------->0
  himagic:  10000000 10000000 10000000 10000000
  lomagic:  00000001 00000001 00000001 00000001
```

在这种方法中假设所有字符都是ASCII标准字符，其值在0-127之间，因此longword总是如下形式：

```
              b3      b2       b1       b0
           31--------------------------------->0
  longword: 0XXXXXXX 0XXXXXXX 0XXXXXXX 0XXXXXXX
```

检测0字节：

如果longword 中有一个字节的所有bit都为0，则进行减法后，这个字节的最高位一定会从0变为1；相反，如果longword中所有字节都不为0，则每个字节中至少有1位为1，进行减法后这个字节的最高位依然为0。

(2) & himagic

这一步取出每个字节最高位的1，如果有任意字节最高位为1则说明longword中有为0的字节。

根据上面的描述，如果longword中有为0的字节，则if中的表达式结果为非0，否则为0。

5. 汇编实现

VC CRT的汇编实现与前面strlen_c算法类似

```
        page    ,132
        title   strlen - return the length of a null-terminated string
;***
;strlen.asm - contains strlen() routine
;
;       Copyright (c) Microsoft Corporation. All rights reserved.
;
;Purpose:
;       strlen returns the length of a null-terminated string,
;       not including the null byte itself.
;
;*******************************************************************************

        .xlist
        include cruntime.inc
        .list

page
;***
;strlen - return the length of a null-terminated string
;
;Purpose:
;       Finds the length in bytes of the given string, not including
;       the final null character.
;
;       Algorithm:
;       int strlen (const char * str)
;       {
;           int length = 0;
;
```

```
31 ;           while( *str++ )
32 ;                   ++length;
33 ;
34 ;           return( length );
35 ;       }
36 ;
37 ;Entry:
38 ;       const char * str - string whose length is to be computed
39 ;
40 ;Exit:
41 ;       EAX = length of the string "str", exclusive of the final null byte
42 ;
43 ;Uses:
44 ;       EAX, ECX, EDX
45 ;
46 ;Exceptions:
47 ;
48 ;*******************************************************************************
49
50         CODESEG
51
52         public  strlen
53
54 strlen  proc \
55         buf:ptr byte
56
57         OPTION PROLOGUE:NONE, EPILOGUE:NONE
58
59         .FPO    ( 0, 1, 0, 0, 0, 0 )
60
61 string  equ     [esp + 4]
62
63         mov     ecx,string              ; ecx -> string
```

```
64          test    ecx,3                   ; test if string is aligned on 32 bits
65          je      short main_loop
66
67  str_misaligned:
68          ; simple byte loop until string is aligned
69          mov     al,byte ptr [ecx]
70          add     ecx,1
71          test    al,al
72          je      short byte_3
73          test    ecx,3
74          jne     short str_misaligned
75
76          add     eax,dword ptr 0         ; 5 byte nop to align label below
77
78          align   16                      ; should be redundant
79
80  main_loop:
81          mov     eax,dword ptr [ecx]     ; read 4 bytes
82          mov     edx,7efefeffh
83          add     edx,eax
84          xor     eax,-1
85          xor     eax,edx
86          add     ecx,4
87          test    eax,81010100h
88          je      short main_loop
89          ; found zero byte in the loop
90          mov     eax,[ecx - 4]
91          test    al,al                   ; is it byte 0
92          je      short byte_0
93          test    ah,ah                   ; is it byte 1
94          je      short byte_1
95          test    eax,00ff0000h           ; is it byte 2
96          je      short byte_2
```

```
 97          test    eax,0ff000000h          ; is it byte 3
 98          je      short byte_3
 99          jmp     short main_loop         ; taken if bits 24-30 are clear and bit
100                                          ; 31 is set
101
102  byte_3:
103          lea     eax,[ecx - 1]
104          mov     ecx,string
105          sub     eax,ecx
106          ret
107  byte_2:
108          lea     eax,[ecx - 2]
109          mov     ecx,string
110          sub     eax,ecx
111          ret
112  byte_1:
113          lea     eax,[ecx - 3]
114          mov     ecx,string
115          sub     eax,ecx
116          ret
117  byte_0:
118          lea     eax,[ecx - 4]
119          mov     ecx,string
120          sub     eax,ecx
121          ret
122
123  strlen  endp
124
125          end
```

6. 测试结果

为了对上述各种实现的效率有一个大概的认识，我在VC8和GCC下分别进行了测试，测试时均采用默认优化方式。下面是在GCC下运行几百万次后的结果(在VC8下的运行结果与此相似)：

```
strlen_a
-------------------------------------------------
       1:        515 ticks         0.515 seconds
       2:        375 ticks         0.375 seconds
       3:        375 ticks         0.375 seconds
       4:        375 ticks         0.375 seconds
       5:        375 ticks         0.375 seconds
   total:       2015 ticks         2.015 seconds
 average:        403 ticks         0.403 seconds
-------------------------------------------------

strlen_b
-------------------------------------------------
       1:        360 ticks          0.36 seconds
       2:        390 ticks          0.39 seconds
       3:        375 ticks         0.375 seconds
       4:        360 ticks          0.36 seconds
       5:        375 ticks         0.375 seconds
   total:       1860 ticks          1.86 seconds
 average:        372 ticks         0.372 seconds
-------------------------------------------------

strlen_c
-------------------------------------------------
       1:        187 ticks         0.187 seconds
       2:        172 ticks         0.172 seconds
       3:        187 ticks         0.187 seconds
       4:        187 ticks         0.187 seconds
       5:        188 ticks         0.188 seconds
   total:        921 ticks         0.921 seconds
 average:        184 ticks        0.1842 seconds
-------------------------------------------------
```

```
strlen_d
-------------------------------------------------
      1:        172 ticks         0.172 seconds
      2:        187 ticks         0.187 seconds
      3:        172 ticks         0.172 seconds
      4:        187 ticks         0.187 seconds
      5:        188 ticks         0.188 seconds
   total:       906 ticks         0.906 seconds
 average:       181 ticks        0.1812 seconds
-------------------------------------------------

strlen
-------------------------------------------------
      1:        187 ticks         0.187 seconds
      2:        172 ticks         0.172 seconds
      3:        188 ticks         0.188 seconds
      4:        172 ticks         0.172 seconds
      5:        187 ticks         0.187 seconds
   total:       906 ticks         0.906 seconds
 average:       181 ticks        0.1812 seconds
-------------------------------------------------
```

源代码：点击下载

posted on 2007-10-12 13:19 蚂蚁终结者 阅读(23403) 评论(34) 编辑 收藏 引用 所属分类: The Annotated CRT Sources

.

Feedback

 # re: strlen源码剖析 2007-09-26 19:29 msdn47

果然很快，效率高啊 回复 更多评论

 # re: strlen源码剖析 2007-09-26 21:29 hi

(2) ^ ~longword
这一步取出加法后longword中所有未改变的bit。

==========

这个什么意思？ 回复 更多评论

 # re: strlen源码剖析 2007-09-27 08:19 蚂蚁终结者

可能我说的不够清楚，看下面的例子：

```
                  b3       b2       b1       b0
               31--------------------------------->0
     longword: 00001001 00011000 00000000 00001100
+  magic_bits: 01111110 11111110 11111110 11111111
          sum: 10001000 00010110 11111111 00001011
^   ~longword: 11110110 11100111 11111111 11110011
            a: 01111110 11110001 00000000 11111000
& ~magic_bits: 10000001 00000001 00000001 00000000
       result: 00000000 00000001 00000000 00000000
```

sum = longword + magic_bits;
a = sum ^ ~longword;
即用sum与longword逐位比较，如果有某个位相同，就说这个位在加法后未改变，这样在a中为1的位就是未改变的。

result = a & ~magic_bits;
得到未改变的hole位，从上例可以看出第16 bit这个hole加法后未改变，这样就检测出了0字节。

回复 更多评论

 # re: strlen源码剖析 2007-09-27 08:28 wangmuy

赞啊！以前不懂的今天终于看懂了！ 回复 更多评论

 # re: strlen源码剖析[未登录] 2007-09-27 08:42 **漂舟**

楼主剖析得全面，顶！ 回复 更多评论

 # re: strlen源码剖析 2007-09-27 08:52 **Yong Sun**

只能检测0~127，是个问题，最好能兼容0~255，有部分制表符和扩展字符位于这个区域。 回复 更多评论

 # re: strlen源码剖析 2007-09-27 09:06 **蚂蚁终结者**

实际上0~255都能检测出来的：

```
if (cp[0] == 0)
    return cp - str;
if (cp[1] == 0)
    return cp - str + 1;
if (cp[2] == 0)
    return cp - str + 2;
if (cp[3] == 0)
    return cp - str + 3;
```

如果上面的语句执行完还没有return，则会继续下一次循环，这样还是能检测到在if语句中漏掉的128~255，只不过效率上会有所损失。如果要检测0~255之间的字符，strlen_c比strlen_d要好。因为strlen_c只会漏掉这样的WORD：

```
    10000000 XXXXXXXX XXXXXXXX XXXXXXXX
```

回复 更多评论

 # re: strlen源码剖析 2007-09-27 09:23 **k120**

为了便于下面的讨论，这里假设所用的计算机为32位，即一个WORD为4个字节?呵呵,笔误吧?
回复 更多评论

 # re: strlen源码剖析 2007-09-27 09:35 **蚂蚁终结者**

Sorry，我不该用WORD这个单词。我说的WORD在这里表示计算机中的一个字长，不是一般为2个字节的WORD类型。
回复 更多评论

 # re: strlen源码剖析 2007-09-27 10:32 **Yong Sun**

另外，是否有endian的问题呢？ 回复 更多评论

 # re: strlen源码剖析 2007-09-27 10:39 **Yong Sun**

想了想，应该没有，呵呵 回复 更多评论

 # re: strlen源码剖析 2007-09-27 11:16 **蚂蚁终结者**

c语言的版本不会有endian的问题，如果用汇编就需要注意了。
假设有4个连续的字节abcd，现在要找出其中的第一个0字节：
1. 在PowerPC这种big-endian的计算机上，将这4个字节读到寄存器中依然是abcd，从左到右找到最左边的0字节就OK了。

2. 在X86这种little-endian的计算机上，将这4个字节读到寄存器中就会变成dcba，这就需要从右到左找到最右边的0字节。

可以看出，上面VC的汇编实现是针对X86计算机的。
回复 更多评论

 # re: strlen源码剖析 2007-09-27 12:51 **∑x**

如果能做到这样的分析，还有什么能学不会?! 回复 更多评论

 # re: strlen源码剖析 2007-10-12 16:36 **Minidx全文检索**

几天前看过的文章怎么又跑前面来了？！ 回复 更多评论

 # re: strlen源码剖析 2007-10-12 16:43 **蚂蚁终结者**

我也奇怪，就改了一下，再发布就变了。连日期都变了，郁闷。。。 回复 更多评论

 # re: strlen源码剖析 2007-10-12 17:04 **Minidx全文检索**

不过是好文章，多看几遍也不错~ 回复 更多评论

 # re: strlen源码剖析 2007-10-13 09:24 **erran**

强！希望这样的好文不要链接失效！o(∩_∩)o... 回复 更多评论

 # re: strlen源码剖析 2007-11-02 13:04 **really green**

```
for (char_ptr = str; ((ulong)char_ptr
& (sizeof(ulong) - 1)) != 0 ;
++ char_ptr) {
if (*char_ptr == '\0' )
return char_ptr - str;
}
```

我比较菜，这里就看不明白：

```
((ulong)char_ptr
& (sizeof(ulong) - 1)) != 0 ;
```

(ulong)char_ptr 这个转换把一个 char * 转成 ulong 会发生什么事情？ 回复 更多评论

 # re: strlen源码剖析 2007-11-02 19:28 **really green**

想了想这个问题问得挺幼稚，我理解 (ulong)char_ptr应该得到一个ulong的值，这个值就是指针char_ptr的值。 回复 更多评论

 # re: strlen源码剖析 2007-11-04 09:35 **蚂蚁终结者**

you gotta it! 回复 更多评论

 # re: strlen源码剖析 2007-12-04 05:39 **福福**

如果有中文字符，这个是不是有问题
if (((longword - lomagic) & himagic) != 0) 回复 更多评论

 # re: strlen源码剖析 2007-12-05 19:39 **蚂蚁终结者**

@福福
中文字符应该用wcslen才对，strlen是用来处理单字节字符串的。
详细描述请看MSDN：
However, strlen interprets the string as a single-byte character string, so its return value is always equal to the number of bytes, even if the string contains multibyte characters. wcslen is a wide-character version of strlen; the argument of wcslen is a wide-character string and the count of characters is in wide (two-byte) characters.
回复 更多评论

 # re: strlen源码剖析 2007-12-27 14:01 **Fox**

几年没动过汇编了，看了代码，有种耳目一新的感觉~ ~ ~ ~ ~ ~ ~ ~ 回复 更多评论

 # re: strlen源码剖析 2007-12-31 17:44 **Sunky**

感谢博主的精彩剖析。
感觉到我不会的东西太多了
由此坚定了我看GNU C的决心
谢谢 回复 更多评论

 # re: strlen源码剖析 2008-09-07 16:11 **star**

博主的文章允许转载么？
昨天看glibc里的注释觉得云里雾里
今天看了博主的文章，突然觉得豁然开朗啊 ;-)
还有想问一个问题 glibc里是这样处理64位的
if (sizeof (longword) > 4)
{
/* 64-bit version of the magic. */
/* Do the shift in two steps to avoid a warning if long has 32 bits. */
magic_bits = ((0x7efefefeL << 16) << 16) | 0xfefefeffL;
himagic = ((himagic << 16) << 16) | himagic;
lomagic = ((lomagic << 16) << 16) | lomagic;
}
为什么不一次就移32位呢？ 回复 更多评论

 # re: strlen源码剖析 2008-09-07 19:54 **蚂蚁终结者**

@star
欢迎转载，转载当然是没有问题的，毕竟写文章就是能让更多的人看到！

为什么不一次就移32位呢？
我也不太清楚，可能就其中注释所说：
/* Do the shift in two steps to avoid a warning if long has 32 bits. */
只是为了避免warn吧呵呵！ 回复 更多评论

 # re: strlen源码剖析 2008-11-07 11:42 **test**

似乎CRT库的这种写法有点越界访问的意思，呵呵。 回复 更多评论

 # re: strlen源码剖析 2009-03-18 17:08 **123**

@福福
strlen算法在处理中文时不会出问题，关键点在后面的if(cp[?]==0)。
不过处理中文时效率会跟最简单的strlen一样。 回复 更多评论

 # re: strlen源码剖析 2009-04-29 14:32 **uestc**

分析得很清楚，写得也很详细，赞一个。 回复 更多评论

 # re: strlen源码剖析 2009-06-05 14:06 **宋兵乙**

```
for (char_ptr = str; ((ulong)char_ptr
& (sizeof(ulong) - 1)) != 0 ;
++ char_ptr) {
if (*char_ptr == '\0' )
return char_ptr - str;
}
```

如上的代码我看不懂了，我的分析如下，麻烦各位高手指出错误。
sizeof(ulong)-1等于3，也就是二进制的0000 0011，于是想要跳出for循环，只需char_ptr的最后两位是0即可。而每次for循环时，char_ptr执行了++操作，由于char_ptr是指向char的指针，因此每次++应该是增加一个char的长度就是8。表现在二进制上就是倒数第4位增加1，而后两位是不会变化的。故得出结论若char_ptr的初始值不满足最后两位是0，那for就永远是死循环了。。

求各位指出上述论证错误在哪。另外，我没明白此例中内存对齐到满足哪种条件才能提高代码效率。 回复 更多评论

 # re: strlen源码剖析 2009-06-08 11:51 **蚂蚁终结者**

@宋兵乙
“由于char_ptr是指向char的指针，因此每次++应该是增加一个char的长度就是8”
看来你对指针运算还不太了解，建议好好复习一下指针部分。
授人鱼不如授人渔呵呵 回复 更多评论

 # re: strlen源码剖析 2009-10-29 11:31 **似水之心**

学习，谢谢分享 回复 更多评论

 # re: strlen源码剖析 2010-07-31 20:38 **hoodlum1980**

不错。我今天反汇编看到strlen的汇编代码一直觉得很奇怪，
没搞懂这几句话在干什么。。。看了这篇文章很有帮助。
回复 更多评论

 # re: strlen源码剖析 2010-11-07 14:37 **郭龙**

学习，学习，时刻关注。 回复 更多评论

刷新评论列表

找优秀程序员，就在博客园

标题 re: strlen源码剖析

姓名

主页

验证码 * 8166

内容(提交失败后,可以通过“恢复上次提交”恢复刚刚提交的内容)

Remember Me?

提交 登录 使用高级评论 新用户注册 返回页首 恢复上次提交

[使用Ctrl+Enter键可以直接提交]

网站导航: 博客园 IT新闻 BlogJava 知识库 程序员招聘 管理

Powered by:
C++博客