实现 C 语言高效编程的四大秘技

2006-12-01 10:23:00 来源: VCKBASE 网友评论 5 条 进入论坛

引言：

编写高效简洁的 C 语言代码，是许多软件工程师追求的目标。本文就工作中的一些体会和经验做相关的阐述，不对的地方请各位指教。

第 1 招：以空间换时间

计算机程序中最大的矛盾是空间和时间的矛盾，那么，从这个角度出发逆向思维来考虑程序的效率问题，我们就有了解决问题的第 1 招——以空间换时间。

例如：字符串的赋值。

方法 A，通常的办法：

```
#define LEN 32

char string1 [LEN];

memset (string1,0,LEN);

strcpy (string1,“This is a example!!”）；
```

方法 B：

```
const char string2[LEN] =“This is a example!”;

char * cp;

cp = string2 ;

(使用的时候可以直接用指针来操作。)
```

从上面的例子可以看出，A 和 B 的效率是不能比的。在同样的存储空间下，B 直接使用指针就可以操作了，而 A 需要调用两个字符函数才能完成。B 的缺点在于灵活性没有 A

好。在需要频繁更改一个字符串内容的时候，A 具有更好的灵活性；如果采用方法 B，则需要预存许多字符串，虽然占用了大量的内存，但是获得了程序执行的高效率。

如果系统的实时性要求很高，内存还有一些，那我推荐你使用该招数。

该招数的变招——使用宏函数而不是函数。举例如下：

方法 C：

```
#define bwMCDR2_ADDRESS 4

#define bsMCDR2_ADDRESS 17

int BIT_MASK(int __bf)

{

return ((1U << (bw ## __bf)) - 1) << (bs ## __bf);

}

void SET_BITS(int __dst, int __bf, int __val)

{

__dst = ((__dst) & ~(BIT_MASK(__bf))) | \(((__val) << (bs ## __bf)) & (BIT_MASK(__bf))))

}

SET_BITS(MCDR2, MCDR2_ADDRESS, RegisterNumber);
```

方法 D：

```
#define bwMCDR2_ADDRESS 4

#define bsMCDR2_ADDRESS 17

#define bmMCDR2_ADDRESS BIT_MASK(MCDR2_ADDRESS)

#define BIT_MASK(__bf) (((1U << (bw ## __bf)) - 1) << (bs ## __bf))
```

```
#define SET_BITS(__dst, __bf, __val) \
((__dst) = ((__dst) & ~(BIT_MASK(__bf))) | \
(((__val) << (bs ## __bf)) & (BIT_MASK(__bf))))
SET_BITS(MCDR2, MCDR2_ADDRESS, RegisterNumber);
```

函数和宏函数的区别就在于，宏函数占用了大量的空间，而函数占用了时间。大家要知道的是，函数调用是要使用系统的栈来保存数据的，如果编译器里有栈检查选项，一般在函数的头会嵌入一些汇编语句对当前栈进行检查；同时，CPU 也要在函数调用时保存和恢复当前的现场，进行压栈和弹栈操作，所以，函数调用需要一些 CPU 时间。而宏函数不存在这个问题。宏函数仅仅作为预先写好的代码嵌入到当前程序，不会产生函数调用，所以仅仅是占用了空间，在频繁调用同一个宏函数的时候，该现象尤其突出。 D 方法是我看到的最好的置位操作函数，是 ARM 公司源码的一部分，在短短的三行内实现了很多功能，几乎涵盖了所有的位操作功能。C 方法是其变体，其中滋味还需大家仔细体会。

第 2 招：数学方法解决问题

现在我们演绎高效 C 语言编写的第二招——采用数学方法来解决问题。

数学是计算机之母，没有数学的依据和基础，就没有计算机的发展，所以在编写程序的时候，采用一些数学方法会对程序的执行效率有数量级的提高。

举例如下，求 1~100 的和。

方法 E

```
int I , j;
for (I = 1 ;I<=100; I ++）{
j += I;
}
```

方法 F

```
int I;

I = (100 * (1+100)) / 2
```

这个例子是我印象最深的一个数学用例，是我的计算机启蒙老师考我的。当时我只有小学三年级，可惜我当时不知道用公式 N×（N+1）/ 2 来解决这个问题。方法 E 循环了 100 次才解决问题，也就是说最少用了 100 个赋值，100 个判断，200 个加法（I 和 j）；而方法 F 仅仅用了 1 个加法，1 次乘法，1 次除法。效果自然不言而喻。所以，现在我在编程序的时候，更多的是动脑筋找规律，最大限度地发挥数学的威力来提高程序运行的效率

第 3 招：使用位操作，减少除法和取模的运算

在计算机程序中，数据的位是可以操作的最小数据单位，理论上可以用“位运算”来完成所有的运算和操作。一般的位操作是用来控制硬件的，或者做数据变换使用，但是，灵活的位操作可以有效地提高程序运行的效率。举例如下：

方法 G

```
int I,J;

I = 257 /8;

J = 456 % 32;
```

方法 H

```
int I,J;

I = 257 >>3;

J = 456 - (456 >> 4 << 4);
```

在字面上好像 H 比 G 麻烦了好多，但是，仔细查看产生的汇编代码就会明白，方法 G 调用了基本的取模函数和除法函数，既有函数调用，还有很多汇编代码和寄存器参与运算；

而方法 H 则仅仅是几句相关的汇编，代码更简洁，效率更高。当然，由于编译器的不同，可能效率的差距不大，但是，以我目前遇到的 MS C ,ARM C 来看，效率的差距还是不小。相关汇编代码就不在这里列举了。

运用这招需要注意的是，因为 CPU 的不同而产生的问题。比如说，在 PC 上用这招编写的程序，并在 PC 上调试通过，在移植到一个 16 位机平台上的时候，可能会产生代码隐患。所以只有在一定技术进阶的基础下才可以使用这招。

第 4 招：汇编嵌入

高效 C 语言编程的必杀技，第四招——嵌入汇编。

“在熟悉汇编语言的人眼里，C 语言编写的程序都是垃圾”。这种说法虽然偏激了一些，但是却有它的道理。汇编语言是效率最高的计算机语言，但是，不可能靠着它来写一个操作系统吧?所以，为了获得程序的高效率，我们只好采用变通的方法 ——嵌入汇编，混合编程。

举例如下，将数组一赋值给数组二,要求每一字节都相符。

```
char string1[1024],string2[1024];
```

方法 I

```
int I;

for (I =0 ;I<1024;I++)

*(string2 + I) = *(string1 + I)
```

方法 J

```
#ifdef _PC_

int I;

for (I =0 ;I<1024;I++)
```

```
*(string2 + I) = *(string1 + I);

#else

#ifdef _ARM_

__asm

{

MOV R0,string1

MOV R1,string2

MOV R2,#0

loop:

LDMIA R0!, [R3-R11]

STMIA R1!, [R3-R11]

ADD R2,R2,#8

CMP R2, #400

BNE loop

}

#endif
```

方法 I 是最常见的方法，使用了 1024 次循环；方法 J 则根据平台不同做了区分，在 ARM 平台下，用嵌入汇编仅用 128 次循环就完成了同样的操作。这里有朋友会说，为什么不用标准的内存拷贝函数呢?这是因为在源数据里可能含有数据为 0 的字节，这样的话，标准库函数会提前结束而不会完成我们要求的操作。这个例程典型应用于 LCD 数据的拷贝过程。根据不同的 CPU，熟练使用相应的嵌入汇编，可以大大提高程序执行的效率。

虽然是必杀技，但是如果轻易使用会付出惨重的代价。这是因为，使用了嵌入汇编，便限制了程序的可移植性，使程序在不同平台移植的过程中，卧虎藏龙，险象环生！同时该

招数也与现代软件工程的思想相违背，只有在迫不得已的情况下才可以采用。切记，切记。