**精通C语言？短短20行经典C语言代码很多人看不明白，你来试一下吧**

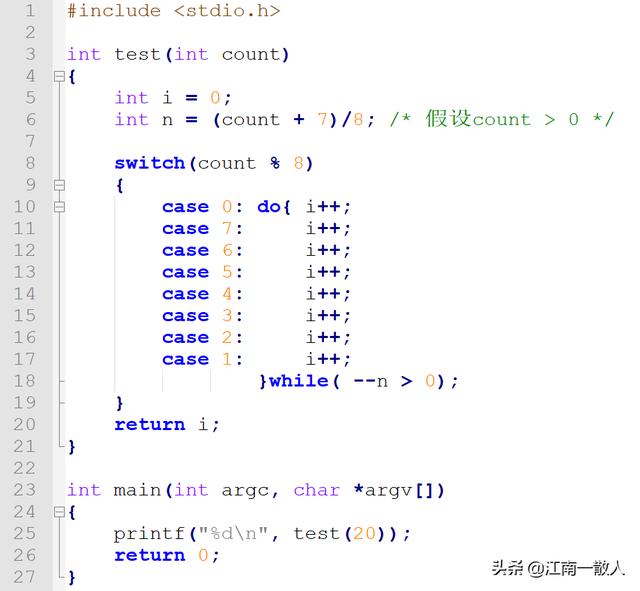
江南一散人 2020-05-02 08:56:02

对编译、链接、OS内核、性能优化等技术感兴趣的童鞋，不妨右上角关注一下吧，近期会持续更新相关方面的专题文章！

**引言**

昨天发了一个文章[《简历上写精通C语言？有道C语言的题来做一下吧》](https://www.toutiao.com/a1665405188317195)，引来很多童鞋围观。很多童鞋表示不太明白，于是就有了这篇文章，详细解释下这个题目的来龙去脉。

题目如下图所示（对原题目做了少许改动）：



test.c

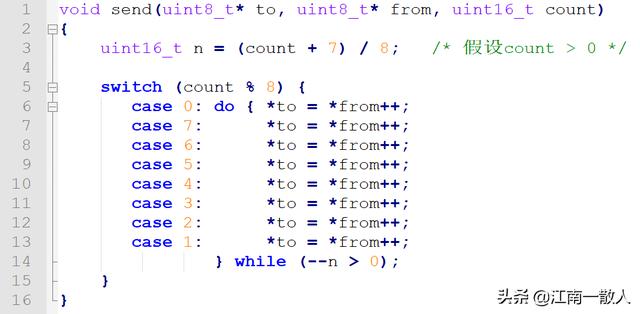
如果你是第一次看到的话，不妨试一下，看你能得出正确答案吗？

其实，这个题目还是源自大师之手，我只是做了少许修改。先来聊一下这段历史渊源吧。

注：为了尽量解释清楚，篇幅有点长，请耐心读完，相信你会有收获的！

**历史渊源**

1983年11月，一位叫Tom Duff的大牛在编写串口通信程序时，发现使用一般的写法时，性能总是不能让人满意。后来，这位老兄凭借深厚的编程功底和精湛的C语言技巧，利用C语言中switch语句的一个鲜为人知的特性，发明如了下图所示的经典代码：



Duff's Device

结果，引来无数吃瓜群众膜拜。在此之前，还没有人发现并利用过C语言的这个特性，于是他便以自己的名字命名这段代码，叫做Duff's Device，一般译为“达夫机器”。

先来看一下大牛的风采吧：



Tom Duff

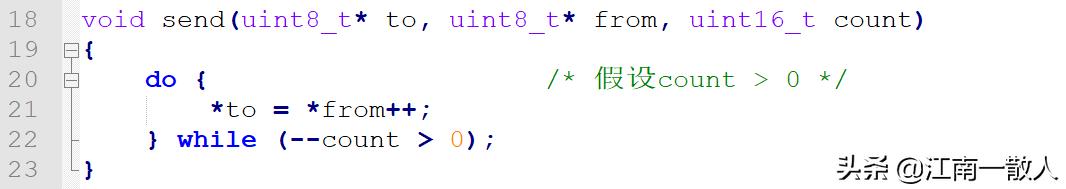
下面来讲解一下这段代码吧。

**Duff's Device - 达夫机器**

当时Duff的需求是把一段起始地址为from，长度为count的数据，写入到一个内存映射的I/O（Memory Mapped I/O ）寄存器to中。

**最简单的实现**

需求很简单，对吧？很容易想到直接用for或者while循环就可以解决了，如下图所示：



最简单的实现

代码清晰简洁，很直观，对吧？

Duff却对此很不满意，因为他觉得这种写法虽然简单，但太过低效，无法接受。

那么，为什么如此简单的代码，却说它性能低下呢？其实主要有两个问题：

* 无用指令太多
* 热点路径上分支指令太多，无法充分发挥CPU的ILP（Instruction-Level Parallelism）技术

我们来分析一下。

1. **无用指令太多**

所谓无用指令，是指不直接对所期望的结果产生影响的指令。

对于这段代码，我们期望的结果就是把数据都拷贝到I/O寄存器to中。那么对于这个期望的结果来说，真正有用的代码，其实只有中间那一行赋值操作：

\*to = \*from++;

而每次迭代过程中的while (--count > 0)产生的指令，以及每次迭代结束后的跳转指令，对结果来说都是无用指令。

上面最简单的实现中，每次循环迭代只拷贝一个字节数据。这就意味着，有多少个字节的数据，就需要执行多少次跳转和条件判断，以及--count的操作。

我们看一下汇编代码：



send() 汇编代码

有些童鞋对汇编不太熟悉，我简单讲解一下：

* x64上优先使用寄存器传递，对于send()函数，第一个参数to存放在寄存器rdi中，第二个参数from存放在rsi中，第三个参数count存放在寄存器edx中。
* 第2~7行，把三个参数分别压入栈中；
* 第9~14行，对应C语言的\*to = \*from++；
* 第15~19行，对应C语言的while (--count > 0)；
* 最后几句，恢复栈帧并返回

所以，第9~19行属于热点路径，也就是主循环体。第9~14行属于有效指令，第15~19行对于期望的数据结果来说就是无用指令。

我们看到，热点路径中，无用指令数占了整个热点路径指令数的一半，其开销也占到整个函数的50%！

1. **热点路径上分支指令太多，无法充分发挥CPU的ILP技术优势**

现代CPU为了提高指令执行的速度和吞吐率，提升系统性能，不仅一直致力于提升CPU的主频，还实现了多种ILP(Instruction-Level Parallelism 指令级并行)技术，如超流水线、超标量、乱序执行、推测执行、分支预测等。

一个设计合理的程序，往往能够充分利用CPU的这些ILP机制，以使性能达到最优。

但是，在代码热点路径上，分支指令太多，导致程序运行中产生大量指令流水线停顿，无法充分发挥ILP的技术优势，从而导致巨大的性能优势。

**注：由于ILP涉及到很底层的CPU硬件知识，很多童鞋可能不熟悉，要讲清楚的话，需要花费大量篇幅。而且，很多童鞋可能也不会有耐心去看，所以这里就暂时先不展开了。**

**但是，了解一些ILP的知识，对于进行系统性的性能优化大有裨益。而且，要想真正理解并掌握如perf这样的性能测量工具，ILP更是必须要掌握的知识。**

**因此，后续我会更新专题文章进行讲解这方面的知识，有兴趣的童鞋可以关注一下。**

现在，我们知道上面那个简单实现性能低下的原因了，那么如何去优化它呢？这就需要用到循环展开的优化手段了。

**循环展开**

所谓循环展开，是通过增加每次迭代内数据操作的次数，来减小迭代次数，甚至彻底消除循环迭代的一种优化手段。

循环展开，有以下优点：

1. 有效减少因循环引起的分支指令。我们前面提过了，这种指令，实际上是对结果不产生影响的无用指令。减少这些指令，就可以减少这些指令本身执行所需的开销，从而提升整体性能。
2. 由于减少了分支指令，可以减少由此引起的CPU指令流水线的停顿，更加有效的利用指令级并行（ILP）技术。

循环展开是一个很常用的性能优化手段。几乎所有的现代编译器，在编译代码时，都会尝试进行循环展开优化。

有童鞋可能会好奇，循环展开到底能提升多少性能呢？我们还是用数据说话，看一个实例吧。

**实例 - 循环展开对性能的影响**

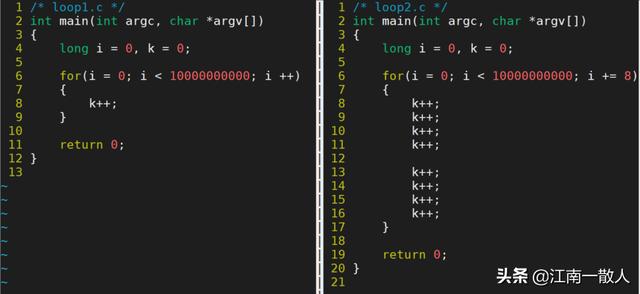
测试环境：

* OS：Ubuntu 19.04(Linux Kernel 5.0.0)
* CPU：Intel(R) Xeon(R) Gold 6130
* 主频：2.10GHz
* Cache 大小：22MB
* Cache line 大小：64 Bytes



测试环境

测试代码：

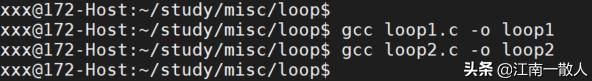


loop1.c 和 loop2.c

loop1.c和loop2.c做的事情一样，唯一的区别是：

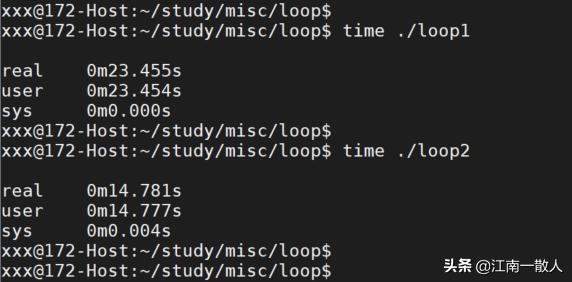
* loop1.c每次循环迭代执行一次k++
* loop2.c每次循环执行8次k++，但是循环的次数比loop1.c少了8倍

编译：



编译

测试结果：



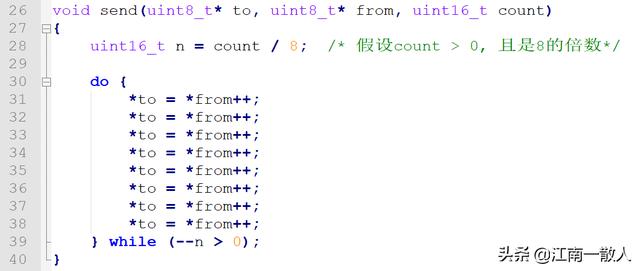
做同样的事情，通过循环展开优化，所消耗时间直接从25.4秒降到了14.7秒，提升了42.4%！

**第一次优化尝试**

了解了循环展开对性能提升的好处之后，我们就可以对上面的简单实现进行第一次优化尝试了。

我们先尝试把每次循环内拷贝字节的个数，由1个提高到到8个，这样就可以把迭代次数降低8倍。

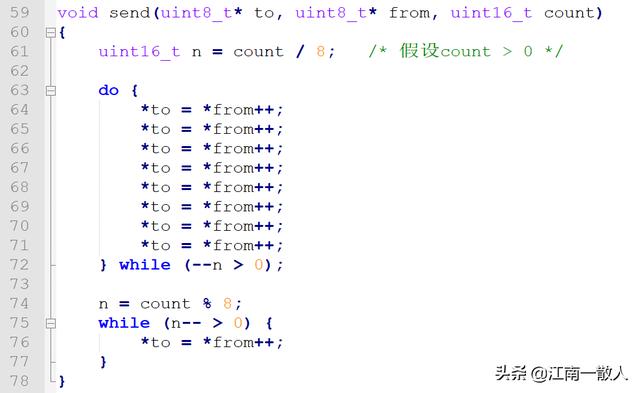
我们先假设，send()函数的参数count总是8的倍数，那么上面的代码就可以修改为：



第一次优化 - count是8的倍数

上面的代码很好理解，就是把原来迭代里的操作复制了8次，然后把迭代次数降低到了8倍。

但是，我们前面做了一个假设，就是count是8的倍数。那如果不是8的整数倍呢，比如20？那我们可能会想到这样的实现：



第一次优化，且 count > 0

其实，到了这里，相比原始的实现来说，性能已经能提升了不少了。但是，Duff仍然不满意，他看着第二个while循环非常不爽，尽管对整体性能已经没有太大影响了。

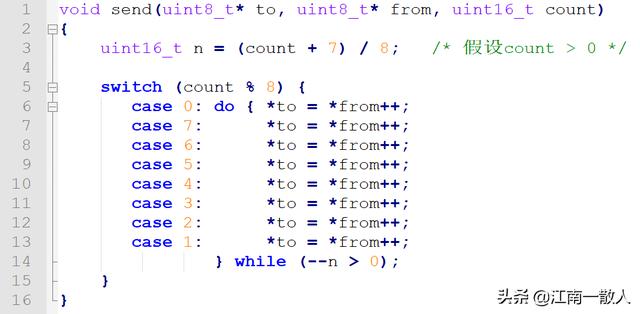
也许这就是大牛与我等普通码农的区别，大牛总是追求极致，总是可以在看似不可能的时候，再往前走一步。

**C语言switch-case的一些特性**

Duff注意到C语言中switch-case语句的一些特性：

1. case语句后面的break语句不是必须的。
2. 在switch语句内，case标号可以出现在任意的子语句之前，甚至运行出现在if、for、while等语句内。

于是，Duff便利用switch-case的特性，用来处理第一个while循环之后仍然剩余的count % 8个字节的数据。于是便有了这样的代码：



Duff's Device

稍微解释下这段代码：

我们假设count = 20，那么：

n = (count + 7) / 8 = 27 / 8 = 3  
count % 8 = 4

所以：

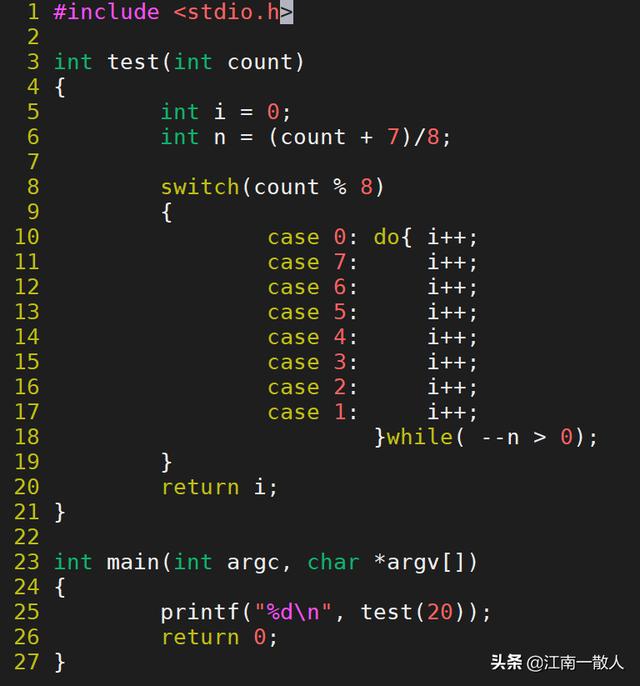
1. switch语句会落入case 4的标签内，然后依次执行了case 4、3、2、1四条语句。自此之后，其实就跟switch-case语句再也没有关系了。
2. while语句判断--n > 0，条件成立，于是跳转到case 0进入循环体执行，于是依次执行case 0、7、6、5、4、3、2、1一共8条语句。此时n = 2.
3. 再次进入while语句处判断--n >0，条件成立，再次跳转到case 0处进入循环体执行。此时n = 1。
4. 此时，while语句处判断--n >0，条件失败，退出循环，函数结束。

好了，到这里，大家应该理解Duff's Device了吧？还是不清楚的话，可以尝试单步跟踪一下，就会很清晰了。

**揭晓答案**

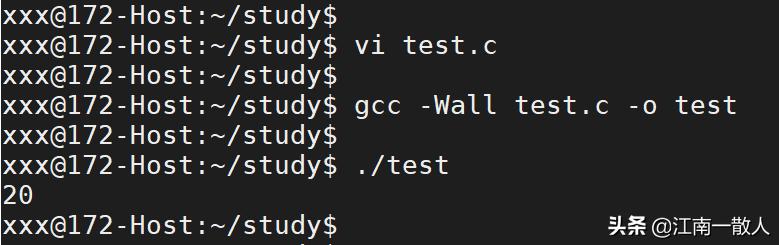
理解了Duff's Device之后，文章开头的那个题目就很好理解了，现在揭晓答案：

再看一下源码：



test.c

编译运行：



编译运行

所以，答案是：20

**结语**

随着硬件的性能越来越好，容量越来越大，导致很多童鞋觉得，现在去纠结诸如Cache、ILP、循环展开等优化手段没有太大的现实意义。

但是，当系统遇到性能瓶颈而又找不到解决思路时，往往在这些平时被绝大多数人忽略的地方，能收到意想不到的收获！而且，一个设计精良的系统，往往设计之初就要充分考虑到这些因素，只有这样才能把硬件的性能充分挖掘出来。

限于篇幅原因，对Cache、ILP技术无法展开介绍，后续我会更新一系列文章，详细介绍这些技术细节。感兴趣的童鞋，不妨右上角关注一下！谢谢！



最后，友情提示：Duff's Device虽然很精妙，但是对于绝大多数童鞋来说，理解起来还是相对比较困难的。因此，产品代码中，不建议使用。否则，轻则被其他童鞋群殴，重则直接被拿来祭天！