实验报告-LAB1

甘晨 181240014

2020 年 1 月 8 日

1 实验进度:

1.1 任务一

完成了任务一所有内容。

1.2 任务二

完成了任务二所有内容。

1.3 任务三

完成了任务三所有内容。

2 实验过程

2.1 任务一

任务一的实现主要思路是按照竖式计算来实现的，但也有些区别。首先利用公式：

$$a * b\%m = (a\%m) * (b\%m)\%m \tag{1}$$

这样以来，若 m<=2147483647，则计算结果不会溢出，这样解决了问题的一部分。对于 m>2147483647 的部分，若 a%m 和 b%m 都小于 2147483647，则也不会溢出。而其他的部分，我们实际上时需要计算 (a%m)*(b%m)，这里采取了将数字存储位字符串，按照小学竖式的计算思路来实现对字符串表示的数字的求积。求得积之后，我们本来应该通过做除法来求余，但实际上我们并不需要知道商是多少，只要知道余数，因此可以利用减法来求余。最初的思路是字符串表示的数字的竖式减法，然后不断减去 m，直到被减数小于 m，但经过测试发现，似乎算到考完期末考试也不一定能算完一个结果。因此，采用了与除法相似的思想，先将减数与被减数最高位对齐，做减法，一直减到被减

数与减数对齐的部分小于减数，然后减数与被减数次高位对齐，以此类推，最终被减数剩下的部分就是余数。

由于 python 没有溢出的问题，利用 python 编写了类似于 PA 中的表达式生成的程序, 生成了测试文件 test.txt，并利用 1.sh 脚本做测试，测试结果均正确。

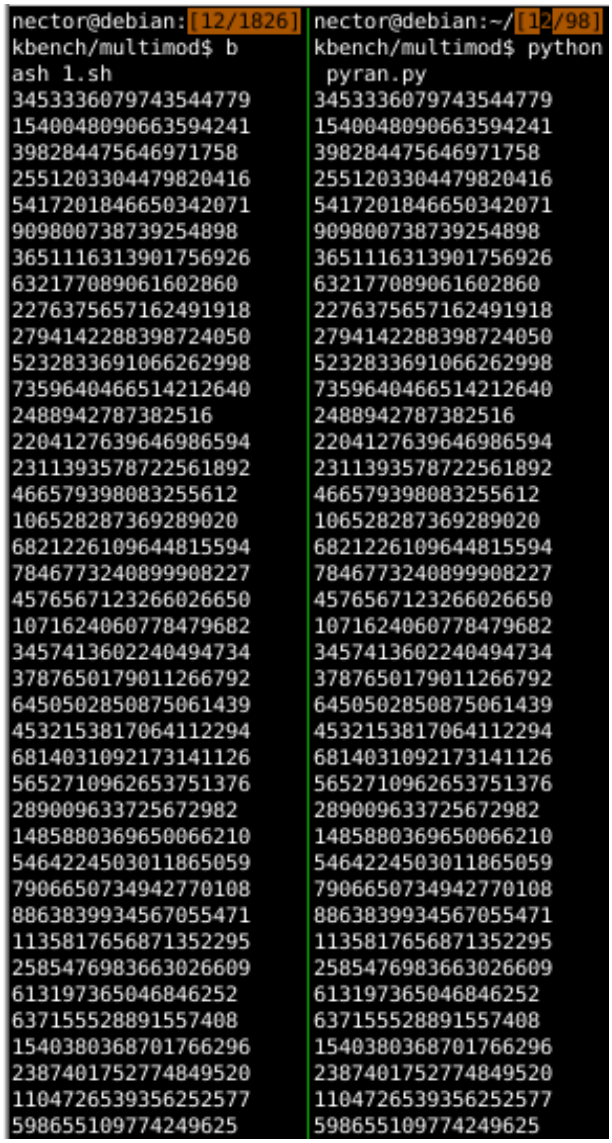

图 1: 这是任务一的某一次测试样例，左边时利用调用 p1.c 计算的结果，右边时 python 计算的结果，比较发现两边一样。

2.2 任务二

我选择的时间度量方法是在进入调用函数和结束调用函数返回之前时各创建一个 struct timeval，以两者的差值来反应 mltimod 函数的执行时间。我这里采用的时最后输出差值的方法来显示运行时间，但由于会调用 printf 函数，会陷入内核执行，因此差值所反应的不仅仅时 multimod 函数的运行时间，但是对于基准实现和优化实现都采用这一度量方法，那么两者都会将陷入内核执行的时间计入进去，相当于都加上了一段近似相等的时间，从定性比较的角度分析而言，我认为仍是相当合理的。

对于任务二，基于给出的提示，我通过 STFW 了解到有关快速幂，而基于这一思路，将相应的 base 的变化由平方改为乘 2 实际上就可实现快速乘，依靠快速乘算法和如下的某运算公式实现的 multimod 函数仍然会有溢出问题。于是尝试将参数强制类型装换为 uint64_t，进行运算，最后的结果再强制类型转换为 int64_t 输出，便可解决溢出的问题。

$$(a+b)\%m = ((a\%m) + (b\%m))\%m \tag{2}$$

对于基准实现，50 个随机产生的测试样例在-O0，-O1，-O2 优化下，每个样例所用平均时间分别如下表所示。

表 1: 时间对比

类别	-O0 (ms)	-O1 (ms)	-O2 (ms)
基准实现	25.94	25.12	24.86
优化实现	1.06	1.06	1.06

通过查阅资料发现-O0 是不做任何优化的，因此就是对应的我的原函数；-O1 将会降低代码块大小和加快程序运行速度，同时保持编译时间没有较大增加；-O2 将会牺牲编译时间来提高代码运行速度。通过表格可以发现，对于基准实现，编译器经过优化执行时间明显减少，说明我所写的代码中有很多地方被优化了，而对优化实现而言则没有明显的时间变化。据此，似乎可以猜想位运算所占的时钟周期较少，因此，有极大可能编译器在进行优化时也是趋向于多使用位运算的这一方向，而优化实现中主要时基于位运算的，所以说编译器无法再做出效果明显的优化。

2.3 任务三

首先分析这一神秘代码：不妨先记：

$$a * b = k * m + q \tag{3}$$

基于此来研究这一代码，首先对于其中最精髓的一段，(int64_t)((double)a * b / m)，这里将 a 强制类型转换为了浮点数因此 a*b/m 的运算时，b、m 也都被转换为了浮点数进

行浮点数的运算，最后再将结果强制类型转换为 int64_t，从感觉看，这段代码实际上是求除了我们上式中的 k，之后我们再看 (a * b - (int64_t)((double)a * b / m) * m)，这里感觉上是求出了 q, 之后又领 q 对 m 取模，且若 t<0 则加上 m，再返回。这段代码让人看起来一头雾水，首先在转化位 double 时，double 可表示的数的范围时大于 int64_t 的，但由于 double 的尾数总共有 53 位，除非 a、b 及其乘积后面有足够多的 0，否则就已经造成精度损失了，但转化为 double 确实解决的溢出问题。在强制转换回 int64_t 时，实际上也还是只有 k*m 的低 64 位得到了保留。根据余数与除数的大小关系：

$$q < m \tag{4}$$

那么 q 一定是可用 int64_t 表示。我们把原等式做一个变形：

$$a * b - k * m = q \tag{5}$$

想一想可以想明白，a*b-k*m 实际上利用低 64 位做差就可表示，根据 m 的范围：

$$0 < m \leq 2^{63} - 1 \tag{6}$$

所以，可以得出：

$$0 < q < m \leq 2^{63} - 1 \tag{7}$$

如果 k*m+q 没有产生向高 64 位，即第 65 位的进位，那么 q 的值可以直接用 a*b 与 k*m 的低 64 位做差得到，也就是代码中的 t>0 的情况，如果产生了这一进位，则要另行讨论。为了方便表示，我们将 a*b 的高 64 位记为 $\overline{X}$，低 64 位记为 $\overline{Y}$, 将 k*m 的高 64 位记为 $\overline{M}$，将低 64 位记作 $\overline{N}$。若 q+k*m 有向 $\overline{M}$ 的进位，则 $\overline{N}$ 的最高位必为 1，且由于 q 的范围，q 的第 64 位也必为 0，所以说，进位之后，$\overline{Y}$ 的最高位必位 0。这样一来，我们可以加以计算。首先按照假设 128 位数来计算，则:

$$a * b - k * m = 2^{64} + \overline{Y} - \overline{N} = 2^{64} + \sum_{k=0}^{62} i * 2^k - 2^{63} - \sum_{k=0}^{62} j * 2^k \tag{8}$$

对于高位，借位之后剩余的高位相同，抵消了，因此在等式中没有写出。再按照 64 位带符号数来计算

$$\overline{Y} - \overline{N} = \sum_{k=0}^{62} i * 2^k - (-2^{63} + \sum_{k=0}^{62} j * 2^k) \tag{9}$$

不难看出，上面两式的结果时相等的。

$$-2^{63} + 1 \leq \sum_{k=0}^{62} i * 2^k - \sum_{k=0}^{62} j * 2^k) \leq 2^{63} - 1 \tag{10}$$

因此，可以得出：

$$\overline{Y} - \overline{N} \geq 0 \tag{11}$$

所以，我认为在精髓代码没有损失精度的情况下，是可以准确求出余数 q 的，这里令我困惑的就是为什么会在 return 时有一个三元运算符，经过我刚才的分析，$\overline{Y} - \overline{N} \geq 0$ 是

始终成立的。那么下面需要解决的问题就是，在什么范围呢，精髓代码不损失精度，在没有确定范围时，我用小于等于 $2^{63}-1$ 这范围做了尝试，结果发现 1000 条测试样例中，错误率高达 96.8%，显然这一神秘代码适用范围不大。基于前面对于类型转换的分析，明显的可以感受到，必须要求 a*b 的值小于 2^{53}，以防止精度损失，同样由于计算时同一会转换为 double 类型，m 的值也要小于 2^{53}，以此为范围生成测试代码。结果显示 1000 条测试样例，正确率 100%，由此可以确定神秘代码适用范围。

$$0 \leq a * b < 2^{53}$$
$$0 \leq m < 2^{53}$$

表 2: 时间对比

类别	-O0 (ms)	-O1 (ms)	-O2 (ms)
基准实现	25.94	25.12	24.86
优化实现	1.06	1.06	1.06
神秘代码	0.15	0.0	0.05

如上可见，通过 20 次得出的平均时间发现，该神秘代码所花费时间远远小于我的基准实现和优化代码。

3 后记

文件夹中多的 pyrand.py 文件是用于生成测试样例的，只需要修改代码中的文件名和选择的函数即可。其中 test.txt 是 p1.c 的测试用例，test2.txt 和 test3.txt 分别是 p2.c 和 p3.c 的测试用例。对于 p1.c 和 p2.c 的测试，只要先执行 pyrand.py，改代码会将正确计算结果输出到屏幕上，并在对应的文件中生成测试样例。之后，再执行对应的 *.sh 测试文件即可在屏幕上输出调用函数计算所得结果。对于 p3.c 的执行，在执行完上述流程之后，需再执行 compare.py 即可输出错误率信息。