

# 实验报告-LAB3

甘晨 181240014

2020 年 1 月 12 日

## 1 实验进度:

### 1.1 实现命令行工具

完成了命令行工具的设计

### 1.2 获取精确时间

完成了函数运行精确时间的获取

### 1.3 比较 multimod 函数性能

对 lab1 中的 3 种 multimod 函数的性能做了比较

## 2 实验过程:

### 2.1 命令行工具

采用了终端直接输出的方式来反应运行时间数据的统计特性，包括循环次数，每一次运行的运行时间，平均运行时间，坏点个数，剔除坏点后的平均运行时间和方差。

```
nector@debian:~/ics-workbench/perf$ ./perf-64 multimod_p3 -r 30
-----THE FUNCTION CALLED IS multimod_p3-----
-----TOTAL LOOP TIME IS 30-----
-----THE FOLLOWING ARE STATISTICS-----
    No.0 Time:0.000011s
    No.1 Time:0.000002s
    No.2 Time:0.000002s
    No.3 Time:0.000002s
    No.4 Time:0.000001s
    No.5 Time:0.000001s
    No.6 Time:0.000001s
    No.7 Time:0.000001s
    No.8 Time:0.000002s
    No.9 Time:0.000002s
    No.10 Time:0.000002s
    No.11 Time:0.000001s
    No.12 Time:0.000001s
    No.13 Time:0.000001s
    No.14 Time:0.000002s
    No.15 Time:0.000002s
    No.16 Time:0.000002s
    No.17 Time:0.000002s
    No.18 Time:0.000001s
    No.19 Time:0.000001s
    No.20 Time:0.000001s
    No.21 Time:0.000001s
    No.22 Time:0.000002s
    No.23 Time:0.000002s
    No.24 Time:0.000002s
    No.25 Time:0.000001s
    No.26 Time:0.000001s
    No.27 Time:0.000001s
    No.28 Time:0.000001s
    No.29 Time:0.000002s
-----THE FOLLOWING ARE ANALYSES-----
    average time : 0.000002s
    update average time : 0.000001s
    update variance : 0.000025×10(-8)
    bad points : 1
```

图 1: 数据呈现

## 2.2 获取精确时间

在实现 `gettime` 函数时,调用了 `time.h` 中的 `clock()` 函数,用于返回该进程开始到调用处的 CPU 时钟计时单元数,将 `func` 函数前后的 `clock()` 取得的值做差,再除以 `time.h` 中定义的 `CLOCKS_PER_SEC`,就可以得到 `func` 运行的时间,以 `s` 为单位可精确到小数点后第六位。下面就考虑如何获得更加精确的函数调用时间了,由于进

程切换会导致运行时间的记录偏大, 所以我们这个运行时间的测试对于含有 `printf` 等需要陷入内核态的函数, 所测得的时间并不准确, 例如框架代码中的 `print_hello`。在多次循环测试程序运行时间时, 有时可以发现某一次或几次运行中, 程序运行所需时间远远大于其他运行, 甚至超过平均值的两倍。对于这些测试数据, 我认为有可能是进程被打断过, 于是采用了如下方式剔除“坏点”: 一、先对所有的时间取一次平均值, 二、剔除所有数据中大于平均值两倍的数据在求一次平均值和方差。

### 2.3 multimod 函数性能比较

由于原 `multimod` 函数需要从终端输入数据来计算, 在评估其性能时难以操作, 于是改为在 `multimod` 函数中生成随机值作为输入, 采取时间作为随机种子。这样会导致多次循环中, `a`、`b`、`m` 的值都是相同的。但这其实对于我们评估函数运行时间有利, 对于同样的数据重复运行, 这样得到的是对于这一组值的平均时间。可同样还有一个问题, 就是三个实现所采用的数据不同, 但考虑到这里只是定性比较三种实现的性能, 因而没有在这里做太多的文章。另外, 由于 C 库函数的随机数最大值为  $2^{31}-1$ , 为了进行大数运算, 可以对生成的随机数乘上了一个  $2^{32}$  再加上  $2^{32}-1$ , 以将数据范围扩大到  $(2^{32}-1, 2^{63}-1)$ 。

下面是三种实现的运行时间比较:

表 1: 时间对比

数据范围	p1.c 平均运行时间 (s)	p2.c 平均运行时间 (s)	p3.c 平均运行时间 (s)
$a, b, m \in (2^{32}-1, 2^{63}-1)$	1.374e-5	2.04e-6	1.21e-6
$a, b, m \in (0, 2^{31}-1)$	1.34e-6	2e-6	1.27e-6
$a, b \in (0, 2^{31}-1) \ m \in (2^{32}-1, 2^{63}-1)$	1.37e-6	2e-6	1.27e-6
$a, b \in (2^{32}-1, 2^{63}-1) \ m \in (0, 2^{31}-1)$	1.27e-6	2.02e-6	1.31e-6

上述数据每一个都是在终端执行了 100 次命令, 每一次命令是 50 次的函数运行得到的剔除“坏点”的平均运行时间, 观察数据可以发现, 实现 2 和实现 3 的平均运行时间在各个数据范围内大体上保持一致, 且都是在  $10^{-6}$  数量级上, 也可以看出实现 3, 即神秘代码实现所花费的时间较少, 性能较好。而实现 1, 即基准实现的性能在各个数据范围内表现的并不一致, 尤其是在  $a, b, m \in (2^{32}-1, 2^{63}-1)$  时, 所花费的时间相较于其他情况尤为巨大, 在  $10^{-5}$  数量级上, 但在其他情况下则在  $10^{-6}$  数量级上, 而在  $a, b \in (2^{32}-1, 2^{63}-1) \ m \in (0, 2^{31}-1)$ , 所花费时间甚至比神秘代码还要少。其实这是很好理解的, 因为我在设计基准现实时就是对数据范围进行分块处理的。根据公式:

$$(a * b) \% m = (a \% m) * (b \% m) \% m \quad (1)$$

那么分别先计算  $a\%m$  和  $b\%m$ ，如果  $m \leq 2^{31} - 1$ ，则前两式的结果一定小于  $2^{31} - 1$ ，而两者乘积一定可以用 64 位带符号整数表示，那么根据公式 (1) 直接计算就可得到结果。推广一下，若满足  $a\%m \leq 2^{31} - 1$  且  $b\%m \leq 2^{31} - 1$ ，一定可以通过直接计算求得余数。可以观察发现，表 1 的数据范围中第 2-4 行都满足这一条件，因此只有第一行的数据范围，基准实现需要转化为字符串进行竖式计算，因而花费时间较多，其余 3 行则可以通过运算直接获得余数，因而花费时间很少。

### 3 后记

这次实验给人的感受是，对一个程序运行时间的分析并不是简单的运行前运行后的时间相减，就像教材上讲到进程的上下文切换时提到的那个例子，从 shell 命令行输入命令，回车并不是程序运行的开始，还需要经历进程上下文切换，从用户态到内核态再到用户态。而在实验中，我对进程被打断这一情况的处理比较粗糙，只是简单地将运行时间远长于平均时间的数据剔除了。同时，我觉得也应该考虑 `gettime` 函数本身占用的时间，因为从创建 `clock()` 变量到返回，这之间应该也是有时间消耗的，可能也应该作为误差考虑。