并行与分布式作业

“数据级并行-线程级并行”
第一次作业

姓名：谷正阳
班级：行政一班
学号：18308045

一、 问题描述

利用 LLVM （C、C++）或者 Soot （Java）等工具检测多线程程序中潜在的数据竞争以及是否存在不可重入函数，给出案例程序并提交分析报告。

参考：http://clang.llvm.org/docs/ThreadSanitizer.html。

二、 解决方案

1. 数据竞争：编写多线程程序，使用 ThreadSanitizer 工具检测
2. 不可重入函数：编写多线程程序，根据 LLVM 语法编写程序找出不可重入函数

三、 实验结果

1. 数据竞争

步骤：

1. 找出实验一编写过的多线程程序：列向量内元素求和和列向量求和
2. 找出实验二编写给过的分治算法求矩阵乘积，并改为多线程
3. 分别对三个程序进行分析，如果出现问题予以修改

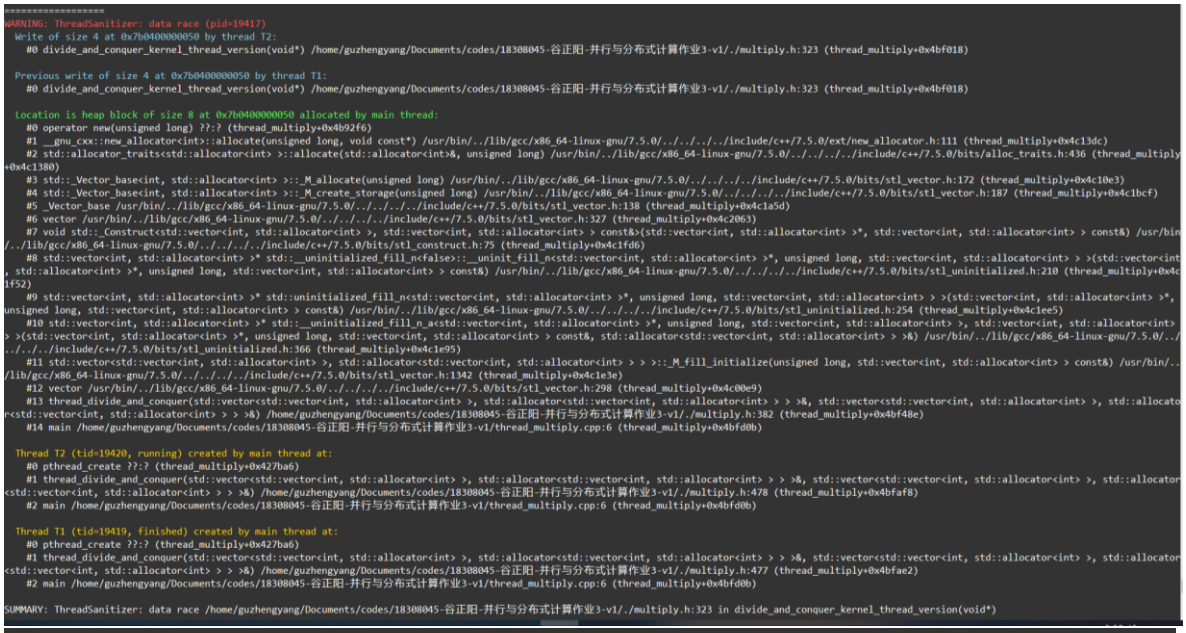
结果分析：

1. 列向量内元素求和和列向量求和不存在数据竞争

```
业3-v1guzhengyang@guzhengyang-Vostro-14-5480:~/Documents/codes/18308045-谷正阳-并行与分布式计算作业3-v1$ ./thread_sum_all
Thread0:
res:    49995000
```

```
业3-v1guzhengyang@guzhengyang-Vostro-14-5480:~/Documents/codes/18308045-谷正阳-并行与分布式计算作业3-v1$ ./thread_sum_vec
5threads:
verify_res:     1
```

2. 矩阵乘法存在数据竞争，报错信息太长，因而缩小问题规模为 2*2 矩阵乘法
3. 观察报错信息得知多个线程使用同一块堆内存，导致数据竞争



3. 按照特性：只有一步乘法，一步加法和一步数据转移，选择合适的锁：spin lock

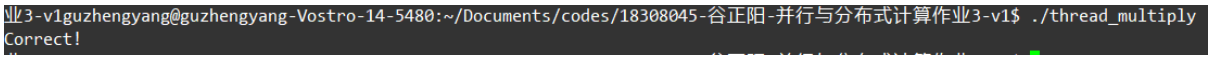
   参考如下：

   内核当发生访问资源冲突的时候，可以有两种锁的解决方案选择：

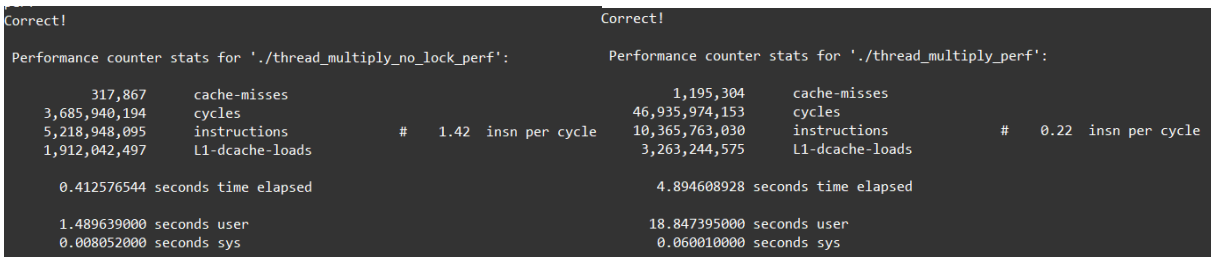
   - 一个是原地等待
   - 一个是挂起当前进程，调度其他进程执行（睡眠）

   Spinlock 是内核中提供的一种比较常见的锁机制，自旋锁是“原地等待”的方式解决资源冲突的，即，一个线程获取了一个自旋锁后，另外一个线程期望获取该自旋锁，获取不到，只能够原地“打转”（忙等待）。由于自旋锁的这个忙等待的特性，注定了它使用场景上的限制 —— 自旋锁不应该被长时间的持有（消耗 CPU 资源）。

4. 重新使用 tsan 分析，发现无错误

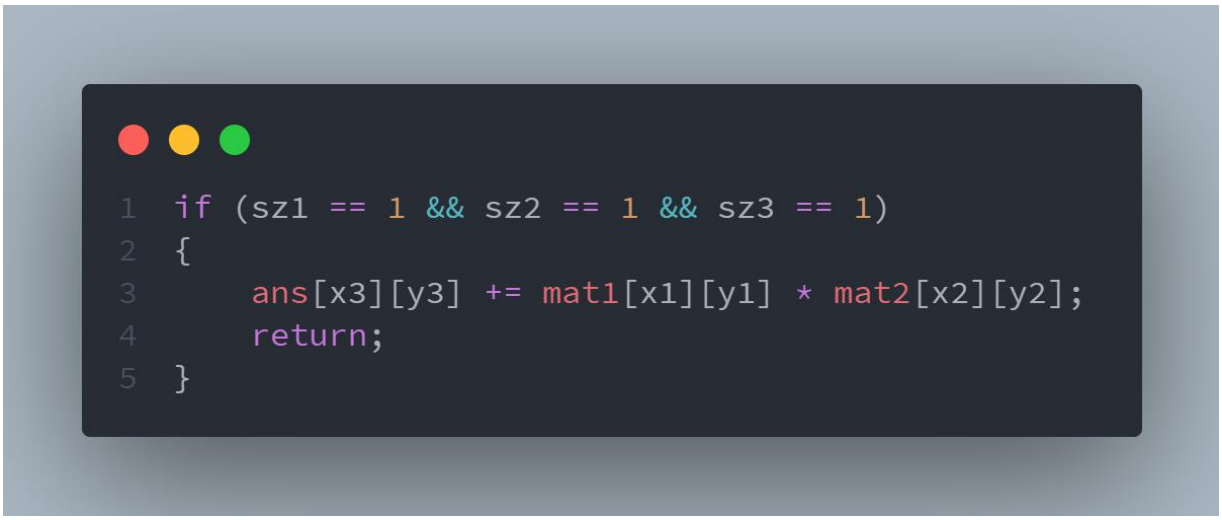


5. 使用 perf 分析运行效率，发现有锁的效率低很多，和不使用多线程的效率近似



```
Correct!

 Performance counter stats for './divide_and_conquer':

            80,806      cache-misses
     2,852,178,453      cycles
     7,110,703,063      instructions              #    2.49  insn per cycle
                 0      mem-loads

       1.066457271 seconds time elapsed

       1.066293000 seconds user
       0.000000000 seconds sys
```

有锁效率固然低，而且此处的数据竞争不会对结果产生坏的影响：



```
if (sz1 == 1 && sz2 == 1 && sz3 == 1)
{
    ans[x3][y3] += mat1[x1][y1] * mat2[x2][y2];
    return;
}
```

所有代码唯一的读写，而且读和写是分开的，不会改变结果。多线程比原本效率近似根据实验一的结果，应该是因为数据规模不大。

2. 不可重入函数

步骤：

1. 编写简单的程序 tiny_race.c，其中囊括不同种类的不可重入函数：

   'Thread1'：使用全局变量 Global

   'I'：调用标准 IO'getchar'

   'Malloc'：使用 malloc 分配的内存

'sub'：调用不可重入函数'I'

'Static'：使用静态变量‘a’

'circle0'：调用标准 IO‘puts’，调用不可重入函数'circle1'，且成环

'circle1'：调用不可重入函数'circle2'，且成环，调用可重入函数’normal’

'circle2'：调用不可重入函数'circle0'，且成环

2. 编写 analyzer.py 找出不可重入函数

结果分析：

```
布式计算作业3-v1/analyzer.py
{'Thread1', 'circle0', 'circle2', 'circle1', 'main', 'I', 'Malloc', 'sub', 'Static'}
```

不包括’normal’，正确

四、 遇到的问题及解决方法

1. 问题：修改分治算法求矩阵乘法时，修改内容较多，重复性工作繁重

   解决：编写 gen.py 根据规则打印要编写的代码

2. 问题：对 LLVM 语法不了解

   解决：主要是对比类似但是不同的代码，找不同。如全局变量和局部变量

3. 问题：malloc 申请的内存不好跟踪

   解决：暂时未解决，待定的方案是记录函数中全部寄存器，找出 malloc 返回值的寄存器，对寄存器的操作如 getelement（指针的偏移）则认为返回值的寄存器被‘感染’，store(取地址)则在返回值寄存器上加一个‘*’，load(解引用)则在返回值寄存器上减一个‘*’若无‘*’则认为是使用了 malloc 申请的内存，函数调用传入时则跳到相应函数重新执行如上方案。但是后来发现 getelement 也可以作为 load，store 的参数，再加上未考虑 delete 进而陷入迷茫。经过上网查询，少有对堆内存跟踪的别人实现的例

子，又发现有说法是调用 malloc 和 free 是有不可重入函数的风险，因而改为判断有无调用 malloc 和 free。另一方面调用了 malloc 和 free 却不使用的情况是不多的，因而如此做也有一定的可行性。

4. 问题：找出因调用不可重入函数而被称为不可重入函数的函数

   解决：最初的方案是简单的遍历一遍全部函数，找到当前函数所有调用，若出现调用不可重入函数，则认为该函数为不可重入函数。然而，有情况：

   > 先定义函数 A，再定义函数 B，再定义函数 C。函数 C 为不可重入函数，函数 AB 待定，函数 A 调用了函数 B，函数 B 调用了函数 C。在首次遍历的时候，函数 B 未被认定是不可重入函数，因而函数 A 未被认定是不可重入函数，若函数 A 是不可重入函数则出错。

   因而考虑构造一个有根树(根为 main)，来表示调用关系，然后通过后序遍历，先确定调用的函数是否是不可重入函数，再判定当前函数是不是不可重入函数。然而，有情况：

   > 若函数有递归，如 A 调用 B，B 调用 C，C 调用 A，则有环，不构成一棵树，而是一个有向图，如此后序遍历没有递归终点。

   因而转换思维，考虑若当前函数为不可重入函数则调用该函数的函数均为不可重入函数，然后按照拓扑排序来找出全部的不可重入函数，最终结果在 circle0，circle1，circle2 相互递归调用的情况下正确的找出了不可重入函数，符合预期。