華中科技大学

研究生课程报告

并发队列的实现实验报告

学　　号　M202470678

姓　　名　梅硕

专　　业　机械工程

课程指导老师　周健

院（系、所）　机械科学与工程学院

2024 年 12 月 28 日

目　　录

1 并发队列的实现方法

本次实践中，基于数组的无锁队列使用老师上课所介绍的方法实现，使用预分配的指针来控制对数组的读写不发生在同一位置（数组的循环复用问题后面讨论）。课后的资料查询中我推测该方法是 Michael and Scott 无锁队列算法的基于数组情况下的变体，原文[1] 给出的实现为基于链表的实现，细节上存在一定的差别，基于链表的实现方案中，使用 CAS 操作，首先将尾指针指向的后续为空的元素链接到新元素，然后调整尾指针指向新元素，因为这两步是分开进行的，因此需要保调整尾指针指向新元素时，尾指针原本指向的元素没有改变，为一开始指向的后续为空，当前指向新元素的节点。这一点是通过此两步操作前的初始化赋值确保的，并在 CAS 操作中进行比较判定而确认生效。基于数组的实现则是首先将预分配指针增加，然后在预分配位置写入数据，最后判断当前尾指针是否达到当前分配节点的前一步，如果成立则将尾指针指向当前节点。

上述操作是不难于理解的，但是具体实现上存在有一定的问题。使用循环数组的情况下，当入队元素超过数组大小，此时如何控制入队元素不覆写之前写入的元素。实际上，在无锁队列的实现中，向一个线程提供队列是否为空/满是没有意义的，即使提供的信息是正确的，在线程收到信息与调用出队或者入队操作间，已经可能有其他线程对队列进行了操作，队列的空/满信息变得无效了。

为解决上述问题，在队列操作内部实现对不合法操作的处理。基于需要与 TBB 中 concurrentqueue 队列对比的考虑，对于出队操作，参考 TBB::concurrent-queue 的实现，当尝试出队时如果队列为空，跳过操作，返回 bool 值为假，否则将出队元素写入传入引用变量。对于入队操作，由于 TBB::concurrentqueue 动态分配内存，不会遇到队列为满的问题，因此合理解决即可。受 TBB 中队列空/满判断的启发，可以设置一个 size 变量，其在进行入队和出队操作前先变化，每个入队操作前，都需要保证 size 变量增加后不超过队列容量，如果不满足的该条件，则在始终尝试入队操作直到队列产生空位。

算法 1.1. NonBlockingQueue-enqueue

Algorithm 1.1 Enqueue

Input: nonblockqueue class member $arr[N], size, head, prehead, tail$
enqueue value v

1: **while** true **do**
2: $t\_size = size$(atomically)
3: **while** $t\_size >= N - 1$ **do**
4: $t\_size = size$(atomically)
5: **end while**
6: **if** ***CAS***$(size, t\_size, t\_size + 1)$ **then**
7: $allo\_pos =$ ***fetchadd***$(prehead, 1)$
8: $arr[allo\_pos\%N] = v$(atomically)
9: **while** !***CAS***$(head, allo\_pos, allo\_pos + 1)$ **do**
10: pass
11: **end while**
12: **break**
13: **end if**
14: **end while**

算法 1.1 代码中 ***fetchadd*** 为 C++ std::atomic 变量的一个原子级操作方式，将原子变量加上一个值，并返回原子变量增加前的值。

算法 1.2. NonBlockingQueue-dequeue

Algorithm 1.2 Dequeue

Input: nonblockqueue class member $arr[N], size, head, prehead, tail$
dequeue write address a
Output: flag

1: Initial $flag = false$
2: $t\_size = size$(atomically)
3: **while** $t\_size > 0$ **do**
4: **if** ***CAS***$(size, t\_size, t\_size - 1)$ **then**
5: $allo\_pos =$ ***fetchadd***$(prehead, 1)$
6: $a = arr[allo\_pos\%N]$(atomically)
7: $flag = true$
8: **end if**
9: $t\_size = size$(atomically)
10: **end while**
11: **return** flag

算法 1.2 表示出队逻辑，实现上述的队列不空执行出队操作的行为。

同时需要解释的是，为何入队操作算法 1.1 第 3 行比较的为 N-1，由于 size 的限制只能保证队列满后不能再添加，而产生的一个情况是，队列满时，出队操

作使 size-1，此时一个入队进程触发，并在读出位置写入新数据，导致读出出错，因此需要间隔一位空位。

2 并发队列的正确性验证

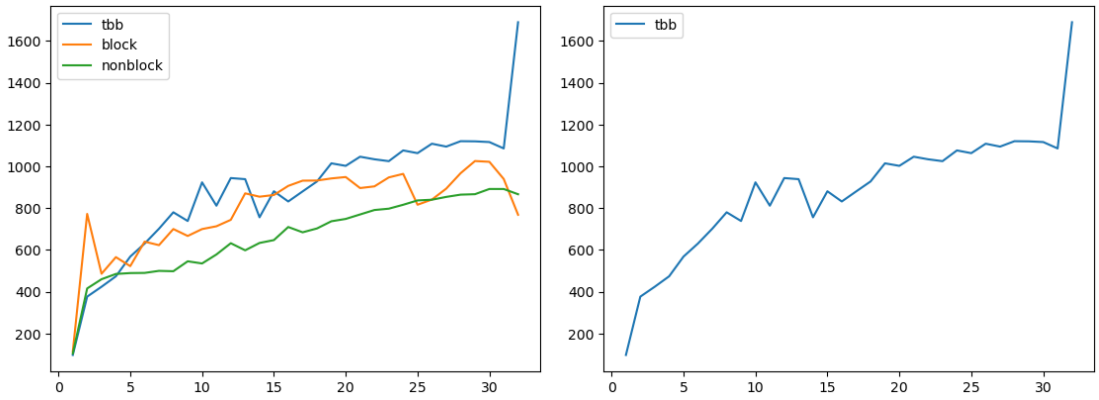
对于并发队列正确性的验证，参考了 TBB 文档中的叙述：”In a single-threaded program, a queue is a first-in first-out structure. But if multiple threads are pushing and popping concurrently, the definition of ”first” is uncertain. Use of concurrent_queue guarantees that if a thread pushes two values, and another thread pops those two values, they will be popped in the same order that they were pushed.”（在单线程程序中，队列是一种先进先出的结构。但是，如果多个线程同时进行推入和弹出操作，那么“先进”的定义就不确定了。使用 concurrent_queue 可以保证，如果一个线程推入两个值，而另一个线程弹出这两个值，那么它们将以与推入顺序相同的顺序被弹出。）

那么实际上，验证实验可以设计为，使用 N 个生产者线程，每个线程将对 N 求余等于线程 id（0~N-1) 的一定范围内的数按序加入队列，使用一个消费线程，进行出队操作，并在出队时，判断该数对 N 求余的组别，验证该组中的数是否单调上升即可。实验中的有锁队列和无锁队列均能通过该测试。

3 性能对比

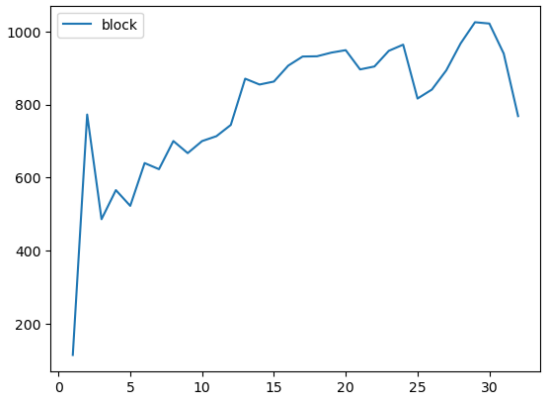
下载并编译 tbb 源码，编译环境 Windows11 wsl2 Ubuntu20.04，编译工具 GNU gcc，cmake release 默认优化选项。

测试中使用 1~32 个线程操作并发队列，需要完成 10^7 次操作，操作使用随机数随机选择，提前测试过随机数的随机效果能够达到出现 0，1 概率接近 1/2。由于基于数组的并发队列实现队列容量有限，当检测到队列为满时，修改输入操作为输出。但由于数组分配空间较大 10^6，因此该情况基本不会发生（概率上相当小）。

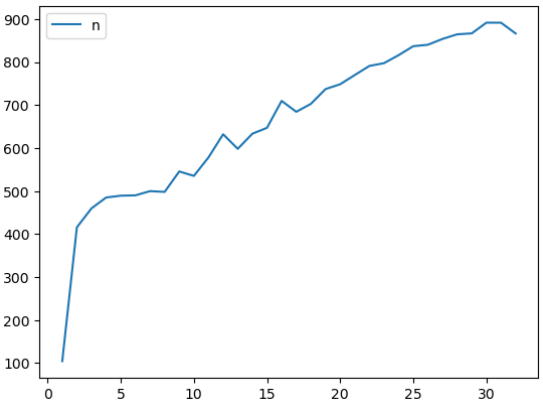


(a) 并发队列对比

(b) tbb 并发队列实现



(c) 有锁队列实现



(d) 无锁队列实现

图 3-1　并发队列实验结果. 图片纵坐标单位 (ms)

由图 3-1 可以看出，三种实现方式在线程数增加时，均出现了时间的增长，即由于线程间的竞争出现了性能的下降。其中 tbb 并发队列在 0~10 线程过程中经历一段接近线性的较为剧烈的上升，在 10~30 线程的阶段则为较为缓慢的性能下降，直到 32 线程时，再次出现了性能的剧烈下降。而实验中基于数组实现的有所队列和无锁队列，主要的特点是在线程数增加到 2 个时，性能大幅下降，5 线程以上进入一段接近线性的性能减小阶段。

有锁的并发队列实现在大部分情况下性能与 tbb 基本一致，无锁队列则能够在性能上略微由于 tbb 库中的并发队列，原因推测为 tbb 中并发队队列需要动态分配内存，同时，作为一款面向使用的程序，其中存在大量的错误处理逻辑，从而性能上较裸的并发队列实现有所下降理所应当。

4 调试记录

为了找到第一章中所提到的出队操作与入队操作相互干扰的问题，花了不少时间，直到测试正确性时将多线程生产单线程消费改为单线程生产单线程消费才终于在调试中发现了问题，同时使用命令行调试在多线程程序调试中发挥了关键作用，学到了很多。

参考文献

[1] MM. NBCAQ[EB/OL]. .
https://lrita.github.io/images/posts/datastructure/Simple_Fast_and_Practical_Non-Blocking_and_Blocking_Concurrent_Queue_Algorithms.pdf.

[2] INTEL. TBBCQC[EB/OL]. .
https://www.intel.com/content/www/us/en/docs/onetbb/developer-guide-api-reference/2022-0/concurrent-queue-classes.html.

華中科技大學

研究生课程报告

带宽测试实验报告

学　　号　M202470678

姓　　名　梅硕

专　　业　机械工程

课程指导老师　姚杰

院（系、所）　机械科学与工程学院

2024 年 12 月 28 日

目　　录

1 （绪论）

（姚老师给的模板太完善导致有点不知道该怎么写这个报告了 QAQ，本质上主要工作就是把姚老师的模板函数复制一份稍加修改就写完了函数，把姚老师的代码复制一份稍加修改就完成了带宽测试）

2 纯 C 语言带宽测试

本章节主要关注的是通过利用 CPU 的位宽、循环展开、使用预取指令来提高拷贝效率。同时为了降低本次实验的难度，拷贝操作均发生在对齐后的内存地址上，但需要了解内存地址对齐对于内存拷贝效率的影响。

内存对齐核心公式是 $align\_pos = \lfloor pos + (block - 1)/block \rfloor * block$，其中 $alignpos$ 表示对齐到的位置，pos 表示原本的位置，$block$ 为对齐的块大小，该公式将 pos 对齐到 $block$ 的下一个整数倍。在 $block$ 为 2 的整数倍时，可用 C 语言运算符表示为 $align\_pos = pos + (block - 1)\&\sim(block - 1)$。需要完成的函数 *alloc_four_aligned_buffers* 只需要按照需要分配的内存累加指针并对齐到指定块大小即可。

CPU 位宽的充分使用在 C 语言中的体现时每次内存复制时，使用 int64_t* 类型的指针，从而使得 CPU 的 64 位充分使用。同时，我们在每次循环中，进行数次内存的拷贝，进行人为的循环展开，以尽可能地提高程序地效率。由于在地址递增取值的情况下，可能出现无效的硬件地址预取，所以我们还是用反向拷贝的方法，优化程序的效果。

本章节中需要进行一定探索的是通过 *__builting_prefetch* 预取指令优化，预取太晚，在硬件预取之后，那么只是徒增了预取指令本身的开销，而预取过早，则可能在预取数据使用前，又被从 cache 中换出，因此需要一定的尝试探索预取指令的发生时机。同时，预取指令取出的 cache 行大小也是需要确定的，分别写出假设预取 32 字节和预取 64 字节的代码进行对比。首先，通过实验发现预取当前存取的位置后（由于是反向，实际上按指针大小应该是前）448~512 字节的数据，较为理想。同时，对于假设预取 32 字节时，使用两条语句预取该片段 64 字

节的数据，而假设预取长度位 64 字节时，代码使用一条语句，如果测试结果中，64 字节性能更好，则表示预取长度为 64 字节。

表 2-1　CPU 预取优化实验数据 单位 MB/s

次数	假设长度预取长度 32			假设长度预取长度 64		
	优化	不优化	提升	优化	不优化	提升
1	11 864.7	11 635.7	0.0197	12 098.8	11 888.2	0.0177
2	11 820.1	11 593.1	0.0196	12 502.7	12 302.8	0.0162
3	11 728.8	11 581.2	0.0127	12 337.8	12 116.2	0.0183
4	11 862.8	11 612.9	0.0215	11 895.7	11 699.4	0.0168
5	12 509.2	12 218.2	0.0238	12 449.8	12 192.9	0.0211
6	11 842.2	11 643.2	0.0171	12 405.9	12 215.2	0.0156
7	12 415.6	12 135.3	0.0231	12 406.4	12 227.9	0.0146
8	11 868.3	11 637.8	0.0198	12 469.6	12 294.3	0.0143
9	12 311.4	12 096	0.0178	12 439.3	12 202.5	0.0194
10	12 487.4	12 213.1	0.0225	12 052.6	11 911.2	0.0119
平均			0.0198			0.0166

表 2-1 为进行 10 次实验记录的到的数据，可以推测预取指令预取 cache 行大小为 32 字节。

实验过程中还发现，正向拷贝与逆向拷贝差异不大，一般情况下正向内存拷贝稍快于逆向，推测是加法运算的速度快于减法，同时，由于在本实验中简化了场景，仅对大块已对齐内存进行拷贝，因此正向拷贝的无效预取效能损失极少，可以忽略。

3　汇编内联带宽测试

本章节主要考虑的问题也是充分利用 CPU 的位宽以及预取。

实验中发现，nontemporal copy 效率高于普通的 copy，查找资料得到的解释是，*moventdq* 所执行的是以非缓存形式写入，写入数据绕过缓存，直接写入内存，因此，能够提高效率，提高性能大约 40%。

而对于预取，则遇到了失败。*prefetchnta* 尝试过无数种添加该指令的时间点，均以无效告终，因此也无法测试其预取大小，但是从参考文档[1] 中可以了解到，其至少取 32 字节数据。

综合纯 C 代码和汇编内联代码，可以在测试框架下得到结果如表 3-1 ，从中可以发现，C 语言实现中，使用预取优化可以得到性能的微小提升，接近于汇编

使用 *movedqa* 语句的内存拷贝实现。而 C 库函数 memcpy 的性能，则接近于汇编中使用 *moventdq* 语句的内存拷贝，性能最好的是 nontemporal copy，可以达到 17263.8MB/s 的带宽。

表 3-1　内存带宽测试结果

测试项目	内存带宽 (MB/s)	样本标准偏差 (%)
C copy	11737.3	1.9
C copy backwards	11735.3	0.4
C copy prefetched	12064.6	0.3
C fill	11896.8	0.2
C *memcpy*()	17055.4	2.8
SSE2 copy	12081.1	0.4
SSE2 nontemporal copy	17263.8	1.7
SSE2 nontemporal copy prefetched	17153.1	0.7
SSE2 nontemporal fill	17253.2	2.9

4　多线程带宽测试

本章使用多线程测试带宽，并使用 CPU 亲和性 API 探索达到峰值性能的最小线程数。细节上对于 *pthread* 及 *sched_setaffinity* 的使用不做具体叙述。主要说明测试方案的选择及论述其合理性。

代码框架基本上未作太大修改，多线程版本相当于多个单线程版本同时运行。但是，单线程版本中，以统计到的最大带宽作为实现方法的带宽结果表示，在该情况下是合理的，但是在多线程需要聚合时可能并非合理，因为很可能存在线程调度中，线程分别占用较多资源而不同时达到最大带宽。因此此处稍做的修改是，使多个线程在几乎相同的时间开始进行内存拷贝，并统计最终结束时间，如果结束时间接近，则认为是合理的，同时带宽聚合使用平均带宽而非最大带宽。在线程同步方面，使用了 Barrier 的策略，在每个线程绑定到对应 CPU 之后设置该同步点。

由于记录了每个线程的情况，输出数据较多，因此以 9 个线程时为例分析。表 4-1 中数据表明，线程基本同步开始进行，符合预期，同时可以由记过可以发现，最大速度之和远大于平均速度之和，高出 33% 显然该结果是不合适的。同时平均速度的方差较大，说明了线程调度中存在一定的负载不平衡问题，而 id 为 8 的线程平均速度远远快于其他线程，可能与 CPU 底层调度逻辑相关。

表 4-1 线程性能数据

线程 id	平均速度 (MB/s)	开始时间 (ms)	结束时间 (ms)	最大速度 (MB/s)
0	2347.2	347900.4	347905.6	5265.4
1	2635.8	347900.4	347904.5	5305.7
2	2340.9	347900.4	347905.6	5258.8
3	2635.8	347900.4	347904.5	5306.3
4	4198.0	347900.4	347905.6	5314.7
5	5267.6	347900.4	347904.5	5429.4
6	4198.5	347900.4	347905.6	5295.0
7	5267.4	347900.4	347904.5	5437.5
8	7540.7	347900.4	347903.3	7573.1
均值	4048.0	347900.4	347904.8	
方差	1768.3	1.0	1.3	
总和	36431.9			50185.9

为探索使带宽达到最大的最小线程数，基于 CPU 编号的规律，可以猜测将线程分配到不同 CPU 能够使用最少线程达到最大的带宽占用，主观猜测将线程分别分配到 0~13，14~27，28~41，42~55 CPU 节点各一个即可达到最大带宽，但是测试中数据表现上并非如此，分配一个线程 0~13，一个 14~27，两个线程后，即可达到 33781.3 MB/s 带宽，然而分配 0~13，14~27，28~41，分别一个后，带宽下降到 22256.1 MB/s 并出现了较高的负载不平衡问题，直到分配 9 个线程，按均匀分配的方法（多出线程仍然按照该顺序分配），达到了 36431.9MB/s 带宽，最高带宽出现在 14 线程时，为 42013.0 MB/s。当然，上述结果只有一定的参考性，因为实验时间过长，服务器可能有其他程序的占用变化。

5 调试记录

不知道为什么不能使用 vscode 的 GUI 进行代码调试，因此接触了 gdb 命令行调试方法，感受到了其在多线程调试中的巨大作用和在各种情况均能使用的泛用性。

调试过程中遇到的最大的问题就是因为疏忽将多线程内存拷贝中，源地址和目的地址写为一个地址，从而导致出现了性能上急剧下降的情况。由于缺乏基本常识，我以为单独开一个线程的运行本来就会有所下降，因此前期的调试并未走在正确的道路上。直到联系了姚老师，他指出了这个问题，还耐心帮我看了代码，并建议首先使用 *memcpy* 测试，使用 *memcpy* 的测试中立马发现了问题，带宽变成了接近 inf，因此我是用调试功能查看了进入 *memcpy* 函数时的状态，然后发现传入的两个参数相同，定位到了问题。这次 debug 的经历让我明白了，debug 过程中需要尽可能使用稳定的函数一步一步测试，同时需要有一定的经验常识，否则容易误入歧途。

致谢

感谢姚老师在我调试出问题的时候的帮助！没有姚老师我将无法完成本次任务，再次感谢。

参考文献

[1] UNKOWN. Prefetch Data Into Caches[EB/OL]. .
https://www.felixcloutier.com/x86/prefetchh.

華中科技大學

研究生课程报告

哲学家就餐问题实验报告

学　　号　M202470678

姓　　名　梅硕

专　　业　机械工程

课程指导老师　周健

院（系、所）　机械科学与工程学院

2024 年 12 月 26 日

目　　录

1 问题描述

哲学家就餐问题（The Dining Philosophers Problem）是一个经典的计算机科学问题，最初由计算机科学家 Edsger Dijkstra 提出，用于研究并发编程中的同步问题。它的意义在于，通过一个简单但抽象的场景，展示了多线程或多进程环境中资源竞争与死锁问题的核心挑战，并为解决这些问题提供启发。在问题中，五位哲学家围坐在圆桌周围，他们在思考和进餐之间交替进行。每位哲学家需要两只叉子才能进餐，而每只叉子都位于两位哲学家之间。如果哲学家同时尝试拿取左右两只叉子，就可能引发以下问题：

- **死锁**：所有哲学家都拿起了右手的叉子，等待左边的叉子，导致所有人无法继续操作。
- **资源饥饿**：某些哲学家可能永远无法获得两只叉子，因为其他哲学家始终占用。
- **并发性低**：哲学家可能采取不必要的等待，降低资源的利用效率。

本实验中，通过哲学家就餐这一经典问题进行研究，了解并发计算中多线程竞争时产生的上述问题，并探索解决他们的方案。

2 “不太聪明”的哲学家

本章节的任务是模拟线程并发中产生的产生的死锁问题，因此本章节中哲学家可能表现得不那么“聪明”。

他们中的所有人均在饥饿时首先尝试拿起左手的叉子，然后拿起右手的叉子，如果拿起叉子的过程失败，那么他会一直等待，直到成功。同时，他们表现出对旁人的漠不关心，只考虑自己的事情，因此，当他们在就餐时，如果身旁的哲学家准备拿起在他们手中的叉子，他们不予理会。上述哲学家行为可以用伪代码表示如算法 2.1 ：

算法 2.1. “不太聪明”的哲学家

此版本的“筷子”使用互斥锁表示即可，当某个哲学家（线程）占用筷子时，

Algorithm 2.1 stupid philosopher

Output: State of philosopher

```
1:  Initialize philosopher's state as thinking, random choose thingkingtime
2:  while true do
3:      if state is hungry then
4:          repeat   try get left chopstick
5:          until    sccess
6:          repeat   try get right chopstick
7:          until    success
8:          change state to eating
9:      end if
10:     if state is eating then occupy chopsticks for eatingtime random choose eating-
        time
11:         change state to thinking
12:     end if
13:     if state is thinking then do nothing for thinkingtime
14:         change state to hungry
15:     end if
16: end while
17: return
```

该线程获得代表筷子的互斥锁。哲学家尝试拿起筷子并一直等待的过程即是上锁阻塞，成功后即是占用互斥锁。

显然，这些不太聪明并对他人漠不关心的哲学家很快进入了每个人都拿起左手筷子的状态，并进入了永无止境的饥饿中（即死锁）。

3 “比较聪明”的哲学家

本章节的任务是解决模拟线程并发中产生的产生的死锁问题，解决的方法是经典的资源层次解决方案。

编号非 0 的哲学家行为与第 2 章所述相同。然而其中编号 0 的哲学家，他首先尝试拿起自己右手的筷子。该哲学家行为可以用伪代码表示如算法 3.1 ：

算法 3.1. “比较聪明”的哲学家

Algorithm 3.1 philosopher

Output: State of philosopher

```
1:  Initialize philosopher's state as thinking, random choose thinkingtime
2:  while true do
3:      if state is hungry then
4:          repeat   try get right chopstick
5:          until    success
6:          repeat   try get left chopstick
7:          until    sccess
8:      end if
9:      if state is eating then
10:         occupy chopsticks for eatingtime
11:         random choose eatingtime
12:     end if
13:     if state is thinking then do nothing for thinkingtime
14:     end if
15: end while
16: return
```

算法 3.1 几乎是将算法 2.1 重复了一遍，仅在 4~7 行有顺序上的调整。然而这样微小的调整，却确实保证了程序不会陷入死锁，因为哲学家 0 不能够与哲学家 4 同时拿起第一根需要拿起的筷子，而陷入无限的等待。

然而，这些“比较聪明”的哲学家虽然不再陷入死锁状态，但出现了新的问题：其中几位哲学家陷入了长久的饥饿。例如，在 0 号哲学家进食时，1~3 号哲学家拿起了自己左手的筷子，4 号哲学家未拿起筷子。在这样的情况产生后，如果 0 号哲学家的动作远远快于 4 号和 1 号，那么 4 号哲学家将始终无法拿起筷子，而 1~3 号哲学家也不会产生任何进展。

具体的程序实现上，除了 0 号线程占用锁的顺序外，两个版本没有什么不同的。

4 “聪明得多”的哲学家

本章节的任务是解决模拟线程并发中产生的产生的死锁和饥饿问题，解决的方法是 Chandy/Misra solution。通过恰当的初始情况设定，该方法避免了死锁，对刚刚吃过的线程造成不利，从而避免了饥饿情况，使得任务能够一直推进。在

进一步之前，我们需要对问题进行一定的限制说明。在本次实验中，为了为了使得饥饿现象进一步解决，在原方法中加入抢占式的调度。

由于本问题实际上在研究多线程中的线程调度，那么处理过程中需要引入一些与实际更为接近的限制条件。

- 哲学家每段就餐时间对应于线程的一个任务需要完成的时间，在该任务完成前，线程不能进行下一个任务。如果任务被中断，下一次获得资源后将继续进行。也就是说，**哲学家如果就餐时被打断而退出，他将进入饥饿状态而不是思考状态**。
- **哲学家的每次进餐有最小时间限制**，这表示了进程执行任务的资源准备，也是线程调度时线程切换的开销和负载均衡的权衡结果。

这个版本下，筷子将总是被哲学家持有并在两个特定的哲学家之间传递。一个持有筷子且使用过筷子（此时筷子是脏的）的哲学家在他的邻桌哲学家需要使用筷子时，将他的筷子擦干净交给请求者，否则始终保留筷子即使是在思考的时候。上述哲学家行为可以用伪代码表示如算法 4.1 。

算法 4.1 该方法的具体操作上比算法 2.1 和算法 3.1 稍微复杂一些。首先，我们需要对表示筷子的方法进行一定的修改，单使用互斥量的方法，不足以使得哲学家线程能够相互通信了解对方是否发出请求。因此，我们的“筷子类”，需要包含有一个互斥锁表示筷子的持有，一个共享锁（使用互斥锁亦可）控制对筷子请求信息的修改，同时需要一个布尔型变量表示筷子是否已经使用，一个整形变量表示该筷子的请求者。

由于这里引入了一个新的锁，控制对筷子的请求，首先应当说明这个锁是不会锁死的，因为该锁仅需要在修改单个筷子的请求时修改，修改完成即可释放，并不存在需要在占用时等待其他资源的情况，因此没有死锁条件，不会死锁。此外在具体的实现细节上，我们需要注意的是，当发起对某个筷子的请求时，如果此时请求变量被邻居设置过，那么尝试标记另外的筷子，直到两个筷子均被标记。标记完成后再进行拾起筷子亦即占有资源的动作，当拾起筷子时，将自己的请求标记清除。

算法 4.1. “比较聪明”的哲学家（加入抢占机制的 Chandy/Misra solution）

Algorithm 4.1 smart philosopher

Output: State of philosopher

```
1: Initialize philosopher's state as thinking, random choose thingkingtime
2: Initialize all chopsticks are used, give chopstick to philosopher with low ID
3: while true do
4:     if state is hungry then
5:         request left chopstick
6:         request right chopstick
7:         pick left chopstick
8:         pick right chopstick
9:     end if
10:    if state is eating then
11:        while eatingtime ≠ 0 do
12:            if neighbor make a request then
13:                mark chopsticks as clean
14:                release corresponding chopstick
15:                break
16:            end if
17:            occupy chopsticks mini-eatingtime
18:            mark chopsticks as used
19:            eatingtime = eatingtime − mini − eatingtime
20:        end while
21:    end if
22:    if state is thinking then
23:        while looptime ≤ thinkingtime do
24:            if neighbor make a request then
25:                mark chopsticks as clean
26:                release corresponding chopstick
27:                break
28:            end if
29:            update looptime
30:        end while
31:    end if
32: end while
33: return
```

5 调试记录

5.1 C++ 的死锁预防机制

依据 cppreference 中的描述，鼓励互斥量上锁底层实现中加入死锁的检测并抛出系统异常 resource_deadlock_would_occur[1]，这一特性导致 version1 在不进行特殊处理的情况下，会出现报错并且不会正常停止，因此程序中使用 try 语句捕捉错误并使用 exit 语句终止程序。

5.2 Windows11 Clang19 多线程 bug

Windows11 使用 Clang19-msvc 作为 C++ 标准库，一定条件下对锁进行释放时会出现两次释放，并报错无法释放未持有的锁。相同代码换用换用 GNU gcc 编译没有问题。（只能说在 Windows 下不用 VS 是个错误，查一个并不存在于自己代码中的 bug 实在是难受）

参考文献

[1] CPPREFERENCE. cpp mutex lock[EB/OL]. .

https://en.cppreference.com/w/cpp/thread/mutex/lock.

華中科技大學

研究生课程报告

多线程求素数

学　　号　M202470678

姓　　名　梅硕

专　　业　机械工程

课程指导老师　周健

院（系、所）　机械科学与工程学院

2024 年 12 月 24 日

目　　录

1 问题描述

我们公司面临一个任务：需要查找 1 到 10^9 之间的所有素数。计划使用 8 个并发线程的来加速这一过程。通过设计一个高效的程序，既能确保计算准确，又能尽量缩短计算时间，降低机器使用的成本。（由于 10^8 范围内，使用筛法求素数效率极高，使用线程带来的常数提升无法弥补使用线程的开销，因此考察 10^9）

强调一下所求范围为 10^9，比原题高一个数量级

2 解决方案

高效求取素数一个有效的算法是埃拉托斯特尼筛法（Sieve of Eratosthenes）。该算法通过遍历数字并标记每个素数的倍数来高效地筛选出非素数。埃拉托斯特尼筛法的时间复杂度为$\mathcal{O}(nloglog(n))$，尽管该方法的时间复杂度并不是求素数的方法中最优的，通过筛至平方根和使用 C++ 类 bitset 的优化方式，在给定的求素数范围内可以得到比欧拉筛法更好的效果[1]，同时该方法更易于并行计算。此外，还可以使用只筛奇数的方法对程序进一步优化，下面对使用的方法做进一步的描述。

2.1 只筛奇数优化

给出埃式筛伪代码之前，首先介绍一下本程序中进行只筛奇数的优化的具体方法。我们使用如下的映射，即 i 映射到 $2i+1$，程序中使用下标表示原像，下面给出在该映射下乘法和平方数的计算规则。记 $\circ$ 为该体系下的乘号。

$$i \Rightarrow 2i+1 \tag{2.1}$$

$$i(\Rightarrow 2i+1) \circ j(\Rightarrow 2j+1) = 2ij+i+j(\Rightarrow 4ij+2i+2j+1) \tag{2.2}$$

$$i(\Rightarrow 2i+1) \circ i(\Rightarrow 2i+1) = 2i^2+2i(\Rightarrow 4i^2+4i+1) \tag{2.3}$$

然而如果仅仅做这样的映射，实际上计算性能会有所降低，因为尽管避免了偶数项的计算，奇数项的乘法计算却从一次乘法变为了两次乘法和两次加法，我们使用 C++ 位移运算符代替 2 的乘法，可以将计算效率提高。即最终的乘法计算公

式为：

$$i \circ j = (ij << 1) + i + j \tag{2.4}$$

$$i \circ i = ((i+1)i << 1) \tag{2.5}$$

这样可以将奇数项的计算变为一次乘法两次加法和一次位移，对运算效率有一定的帮助。程序中均使用该方法来进行下标的计算，但后续的算法描述中为了简洁，均不对该映射做任何说明。

2.2 常数优化

在 10^9 数据范围内使用只筛奇数筛至平方根的埃式筛相当于时间复杂度为$\mathcal{O}(n)$，并存在一个常数倍数约为 1.134，多线程对于线性时间夫再度算法的优化也只能是常数项的提升，为了达到更为严苛的测试环境，依据[1] 中提出的，对埃式筛使用 bitset 类进行常数优化，以验证多线程方法的有效性，即线程调度的开销不会导致负优化的产生。

2.3 单线程埃氏筛

埃式筛的算法基于素数的定义，将为素数倍数的数标记为合数，剩下未标记的即为素数。算法相当简单，为方便后续说明，给出伪代码如下。

算法 2.1. 埃式筛伪代码

Algorithm 2.1 Eratosthenes Sieve

Input: UpBound N
Output: Array PrimeFlag[N]

```
1:  Initialize PrimeFlag[N] as all true
2:  PrimeFlag[0] = false, PrimeFlag[1] = false
3:  for i = 2, ..., √N do
4:      if PrimeFlag[i] then
5:          for j = i, ..., N do
6:              if i · j > N then break
7:              end if
8:              PrimeFlag[ i · j ] = true
9:          end for
10:     end if
11: end for
12: return result
```

2.4 多线程埃氏筛

接下来考虑如何对该算法进行并行计算。该算法主要的操作集中在第二个循环内部，需要对 i 的倍数在待求区间内进行标记。一个显然的并行化策略是将待标记区间按线程数 t 均匀分为 t 段，每段由不同线程进行计算，从而加快运算效率。该方法在对于 i 的倍数进行标记时的局部范围内各个线程互不干扰高度并行。然而对于不同的 i，线程间需要进行同步，以避免多个线程对同一个标记数组的相同区域进行操作。每个线程将自己的区域标记完成后，等待其他线程的的操作完成，才能继续外部对于 i 的访问循环。使用基本的同步屏障 Barrier 方式可以完成这一任务。

算法 2.2. 多线程埃式筛伪代码

Algorithm 2.2 Eratosthenes Sieve

Input: UpBound N Threadid id Threadnum n
Output: Array PrimeFlag[N]

Initialize PrimeFlag[N] as all true
PrimeFlag[0] = false, PrimeFlag[1] = false
for $i = 2, ..., \sqrt{N}$ **do**
    if PrimeFlag[i] **then**
        $jmax = \lfloor N/i \rfloor + 1$
        $jmin = i$
        $piece = (jmax - jmin)/n$
        $startj = id \cdot piece$
        $endj = (id + 1) \cdot piece$
        for $j = startj, ..., endj$ **do**
            if $i \cdot j > N$ **then break**
            end if
            PrimeFlag[$i \cdot j$] = true
        end for
    end if
    Barrier wait for all threads finish
end for
return result

需要说明的是其中一些细节方面的处理，如整除问题和保证数组访问不越界的限制被略去了，如需了解可以参考源代码。

3 实验结果及分析

3.1 实验结果

实验环境：**Windows11 wsl2 Ubuntu20.04**

对于单线程的优化已经达到相当高的水平，在这样的条件下，多线程仍能表现出比单线程更优秀的性能表现。

表 3-1 单线程与多线程耗时

	单线程	多线程
用时	$2215ms$	$1000ms$

表 3-1 数据表明，使用 8 个线程计算能够进一步提高筛法计算的效率，优化后的算法运行时间从 2215ms 降至 1000ms，优化后的算法运行时间从 2215ms 降至 1000ms，实现了约 55% 的时间节省和约 2.2 的加速比，能够更为有效地利用 cpu 计算能力和内存 IO 性能。当然，需要说明的是，实验中测试结果是有一定浮动的，但基本保持在该值附近，因此选取上述取值。

此外，给出每条线程标记素数部分的用时作为线程负载均衡情况的参考。

表 3-2 线程素数标记用时

线程 id	0	1	2	3
用时	$882110us$	$882110us$	$882100us$	$882075us$
线程 id	4	5	6	7
用时	$882012us$	$881931us$	$882104us$	$882146us$

依据表 3-2 中的数据，可知线程负载基本是均衡的，但是实际上这个结果是不可信的。因为线程间存在同步操作，而计时又在同步操作之外，时间的近似相等是程序默认保证的结果。

3.2 实验结果分析

尽管上述的提高是较为令人满意的，但是不容忽视的问题：是使用 8 个线程并发的情况下，加速比仅仅为 2.2。下面提出产生该问题的几点可能原因。

3.2.1 线程同步的影响

在本解决方案中，有一个较大的性能瓶颈是对于每个新的素数，进行筛法之前，线程间必须进行同步，该同步发生的次数等于待求区间中的素数个数，由素数分布定理 $\pi(x) \approx \frac{x}{ln(x)}$，其中 $\pi(x)$ 为素数计数函数。因此在相当于存在一个接近$C \cdot \mathcal{O}(n)$ 的时间复杂度附加项，C 为一个小于 1 的常数。此外，更为糟糕的是，这样的同步等待导致线程的进度被最慢的线程所限制，因此相当依赖于操作系统本身的线程调度策略，当操作系统的线程调度越均衡时，程序的并行性越高。

3.3 素数标记外的时间开销

表 3-2 尽管不能够反应线程的负载是否均衡，但结合该表与表 3-1 的结果，却向我们指出了另一个信息：素数标记之外的时间开销是存在的且不应该被忽略。二者之差，即素数标记之外的时间开销占到了整个程序时间开销的约 12%，其中包含的部分有：bitset 类的初始化（非并行），素数的求和统计（并行），线程的派生（非并行），文件的读写（非并行）。可以预测的是，bitset 类的初始化在进一步增大时应当将之设置为并行的，否则其可能会称为程序效率的瓶颈所在。

4 调试记录

4.1 虚假唤醒问题

该问题发生在 Windows11 下使用 clang19 msvc 版本的调试中，由于在筛法最后进行的部分，对应于一个数值需要标记的部分不足线程数，此时选择的方法是只用线程 0 标记，然而在调试中出现了部分线程执行完成，但程序无法退出的问题。反复阅读相关文档后推测是线程到达屏障的间隔太小，发生了条件变量虚假唤醒的情况。在不进行标记操作的线程中插入微量的延迟，问题解决。

参考文献

[1] OIWIKI. bitset: 与埃氏筛结合 [EB/OL]. .
https://oi-wiki.org/lang/csl/bitset/#%E4%B8%8E%E5%9F%83%E6%B0%8F%E7%AD%9B%E7%BB%93%E5%90%88.