# 高性能计算导论第四次作业

李晨昊 2017011466

2019-5-5

## 目录

# 1 Exercise 4.7

1. 一个生产者 + 一个消费者

生产者和消费者的逻辑完全无关，也不用放在循环里，所以没必要写成一个函数，就写成生产者 + 消费者两个函数即可。

```
void* cons(void*) {
  while (true) { // 循环直到成功消费一次为止
    // 本质上来讲这个锁的意义不是特别大
    // 只要能保证 ok 和 msg 的内存序的话，其实是不用上锁的
    Locker<Mutex> _lk(&mu);
    if (ok) {
      printf("get msg %s\n", msg);
      break;
    }
  }
  return nullptr;
}

void* prod(void*) { // 生产一次就退出
  // 上锁可以保证消费者不会看到 ok 为 true 而 msg 未设置的状态
  // 但是其实只要正确地设置内存序，也是可以的
  Locker<Mutex> _lk(&mu);
  strcpy(msg, "hello world!");
  ok = true;
  return nullptr;
}

  // caller
  pthread_t cons, prod;
  pthread_create(&cons, nullptr, ::cons, nullptr);
  pthread_create(&prod, nullptr, ::prod, nullptr);

  pthread_join(cons, nullptr);
  pthread_join(prod, nullptr);
```

关于这里提到的内存序的问题，看起来是 `volatile` 试图解决的问题，实际上不是的。具体可以参见附。

2. 偶数生产者，奇数消费者

与 1 的逻辑完全一致，只是需要循环创建线程，而偶数号线程承担生产者工作，奇数号线程承担消费者工作。

```
void *thread_fn(void *_tid) {
  usize tid = (usize)_tid;
  if (tid % 2 == 0) {  // prod
    while (true) {
      Locker<Mutex> _lk(&mu);
      if (!ok) {
        sprintf(msg, "hello world from tid = %lu", tid);
        ok = true;
        break;
      }
    }
  } else {  // cons
    while (true) {
      Locker<Mutex> _lk(&mu);
      if (ok) {
        printf("%lu: get msg: %s\n", tid, msg);
        ok = false;
        break;
      }
    }
  }
  return nullptr;
}
```

3. 消费前一个，生产给后一个

```
u32 recv, n;

void *thread_fn(void *_tid) {
  usize tid = (usize)_tid;
  bool sended = false, recved = false;
  while (!sended || !recved) { // 收到一次且发出一次时退出
    Locker<Mutex> _lk(&mu);
    if (ok) {
```

```
      if (!recved && recv == tid) { // 没收到过，且允许的接收方是自身
        printf("%lu: get msg: %s\n", tid, msg);
        ok = false;
        recved = true;
      }
    } else if (!sended) {
      sprintf(msg, "hello world from tid = %lu", tid);
      ok = true;
      sended = true;
      recv = (tid + 1) % n; // 设置允许的接收方为下一个线程
    }
  }
  return nullptr;
}
```

## 2 Exercise 4.11

1. 两个 Delete 操作同时执行

线程 $T_0$ 删除头结点，线程 $T_1$ 删除头结点下一个节点，执行顺序如下：

```
0: if (pred_p == NULL) // true
0: *head_p = curr_p->next;
0: free(curr_p);
1: if (pred_p == NULL) // false
1: pred_p->next = curr_p->next; // access freed memory
```

2. 一个 Insert 和一个 Delete 操作同时执行。

线程 $T_0$ 删除头结点，线程 $T_1$ 插入在原先链表中能排次小的节点，执行顺序如下：

```
0: if (pred_p == NULL) // true
0: *head_p = curr_p->next;
0: free(curr_p);
1: if (pred_p == NULL) // false
1: pred_p->next = temp_p; // access freed memory
```

3. 一个 Member 和一个 Delete 操作同时执行。

线程 $T_0$ 删除头结点，线程 $T_1$ 查找头结点，执行顺序如下：

```
0: if (pred_p == NULL) // true
0: *head_p = curr_p->next;
0: free(curr_p);
1: while (curr_p != NULL && curr_p->date < value) // access freed memory
```

4. 两个 Insert 同时执行。

线程 $T_0$ 和 $T_1$ 都插入在原先链表中能排最小的节点 (这两个值的大小关系随意)，执行顺序如下：

```
0: if (pred_p == NULL) // true
0: *head_p = temp_p;
1: if (pred_p == NULL) // true
1: *head_p = temp_p; // override T0's change
```

5. 一个 Insert 和一个 Member 同时执行。

线程 $T_0$ 插入的节点正是线程 $T_1$ 正在寻找的。严格来说，这种情况下的 “正确” 与否并不是很好界定，如果定义为：输出必须与操作序列按顺序到来时的输出完全一致，则如果插入先开始，但是查找结束时仍未成功插入，则这次 “不存在” 的返回结果是错误的；如果定义为：输出应该符合操作到来时刻的链表状态，则如果插入先开始，但是查找结束时已经成功插入，则这次 “存在” 的返回结果是错误的。

但是，还是有一个确定的标准可以判定，这两个操作同时执行是会出错的，即内存安全的标准。表面上来看，插入和查找同时执行是不会产生内存不安全的，但实际情况并非如此。

首先编译器可以进行代码重排，对于插入的这个代码片段：

```
  ...
1: temp_p->next = curr_p;
2: if (pred_p == NULL)
3:   *head_p = temp_p;
4: else
5:   pred_p->next = temp_p;
```

2345 行可能被调整到 1 行前执行，此时查找过程可能读取到一个 `next` 尚未赋值的节点，从而发生内存错误。

即使在语法层面指示了编译器不要这样重排，处理器执行过程中仍然不能保证内存读写顺序就是汇编代码的顺序。如果发生 store-store 重排，上面的情况仍然可能发生。幸运 (?) 的是，平时常用的 `x86_64` 架构下这种重排是不会发生的。但是这对别的架构就不一定成立了。具体可以参见附。

## 3 Exercise 4.12

不安全。

同时删除：如果在删除的查找阶段只使用读锁，可能会发生两个线程同时试图删除同一个节点，它们也都能顺利地找到这个节点，虽然后续的删除过程会分为一先一后，但是后执行的线程会对一个已经删除过的节点再删除一次，可能发生包括 double free 之类的各种错误。

同时插入：如果两个试图插入的节点都位于原链表的同两个节点中间，则它们会找到同一个插入位置，后续插入时不能保证这两个节点中，大的被放在小的后面。

一插入一删除：如果插入节点的位置在待删除节点之后，且删除操作先执行，则删除完后插入时，会访问被 free 了的内存。

## 4 Exercise 5.4

初值是对应运算的单位元。

| 操作 | 初值 |
|---|---|
| && | true(1) |
| \|\| | false(0) |
| & | $\underbrace{11111...11111_2}_{\text{sizeof(T) * CHAR\_BIT}}$ |
| \| | 0 |
| ^ | 0 |

## 5 Exercise 5.5

1. 计算完四次加法后得到 1008，存入内存后四舍五入到 $101 * 10^1$，所以输出为 `1010.0`
2. 线程 0 计算出 4.000，线程 1 计算出 1003，存入内存后，参与下一次加法的是 4.00 和 $100 * 10^1$。加法结果为 1004，保存后结果为 $100 * 10^1$，输出结果为 `1000.0`。

## 6 Exercise 5.8

只需利用等差数列求和公式即可：

```
#pragma omp parallel for
  for (int i = 0; i < n; ++i) {
    a[i] = i * (i + 1) / 2;
```

```
}
```

# 7 Exercise 5.14

1. 为了储存向量 y，最少需要多少个缓存行?

$y \in R^8$，最少需要 $\frac{8*8}{64} = 1$ 个缓存行。

2. 为了储存向量 y，最多需要多少个缓存行?

$y$ 至多可以存在于两个连续的缓存行，故最多需要 2 个缓存行。

3. 如果缓存行的边界和双精度数的边界始终一致，那么一共有多少种方式将 y 中的元素分配给缓存行?

按照 $y$ 的第一个元素在其所在的缓存行中的 8 个位置分类，得一共有 8 种方式将 y 中的元素分配给缓存行。

4. 如果我们只考虑两个线程共享一个处理器，那么在我们的计算机上一共有多少种方式将四个线程分配给处理器？我们假定在同一个处理器上的内核共享缓存。

$\frac{C_4^2}{2} = 3$ 种。其中 $C_4^2$ 是让一个处理器选择两个线程，除以二是因为选择某两个线程和选择除他们外的两个线程被视作相同的选择。

5. 在我们的例子中，有没有哪一种对向量的分配和对线程的分配方式不会产生伪共享？换句话说，对于处于不同处理器上的线程，他们各自要处理的向量的元素不会出现在同一个缓存行内?

有。例如：$y$ 的前四个元素在一个缓存行，后四个元素在另一个；线程 01 位于核 0 上，它们负责计算 $y$ 的前四个元素；线程 23 位于核 1 上，它们负责计算 $y$ 的后四个元素。

6. 有多少种方法将向量元素分配给缓存行和将线程分配给处理器?

$3 * 8 = 24$

7. 在这些分配中，有多少会使得程序没有伪共享?

    1. 若不是 $y$ 的前四个元素在一个缓存行，后四个元素在另一个，则必有一个缓存行拥有超过 4 个 $y$ 中元素，不能被一个核上的两个线程完全处理，从而一定会需要两个核的缓存。
    2. (假定线程 0，1，2，3 分别负责 $y$ 的 01，23,45,67 元素) 若不是线程 01 在一个核，线程 23 在另一个核，则必定导致一个缓存行被两个核处理。

    由 12 知，有且仅有一种分配方法，就是 5 中提到的那种。

# 8 Programming Assignments 5.1

我编写了 pthread 和 openmp 两个版本的 ~~(其实是因为没看清要求，先写了个 pthread 版的)~~。

openmp 版非常简单，开启一个 omp parallel 块后，执行两次 omp for，一次是统计每个线程内的桶的计数，一次是把每个桶的计数合并起来，两次 omp for 间用 omp barrier 作为路障，保证前一阶段执行完成。关键代码如下

```
#pragma omp parallel num_threads(th)
  {
#pragma omp for
    for (u32 tid = 0; tid < th; ++tid) {
      // 分配任务，这样的分配方法并不是最均匀的，但是比较容易写
      // 而且 n 一般远大于 p，所以影响不大
      u32 beg = n / th * tid, end = (tid == th - 1) ? n : n / th * (tid + 1);
      u32 *x = loc_cnt.get() + bin * tid;
      memset(x, 0, bin * sizeof(u32));
      // 直接根据除法来计算位于哪个桶
      for (u32 i = beg; i < end; ++i) {
        ++x[u32((data[i] - min) / ((max - min) / bin))];
      }
    }
#pragma omp barrier
#pragma omp for
    for (u32 i = 0; i < bin; ++i) {
      // 使用临时变量保存一个桶在所有线程上的和
      // 确保 false sharing 不会发生 (虽然我也相信编译器可以自动做出这个转换)
      u32 sum = 0;
      for (u32 j = 0; j < th; ++j) {
        sum += loc_cnt[j * bin + i];
      }
      loc_cnt[i] = sum;
    }
  }
```

pthread 版本我写的稍复杂一些，我只开启了一次线程，在这一个函数中执行统计和树形规约的操作。在主线程中可以计算出树形规约的每一层有多少线程需要参与规约，然后每一层都使用 pthread_barrier 来同步这些线程。

```
// 主线程中初始化路障，这个计算方法是我自己推导出来的
// 每一层，剩下的线程数是上一层的线程数/2 上取整
// 参与计算的线程数是本层线程数/2 下取整
for (u32 i = th, idx = 0; i != 1; ++idx, i = (i + 1) >> 1) {
  pthread_barrier_init(&reduce[idx], nullptr, i >> 1);
}

...

u32 beg = n / th * tid, end = (tid == th - 1) ? n : n / th * (tid + 1);
u32 *x = loc_cnt.get() + bin * tid;
memset(x, 0, bin * sizeof(u32));
for (u32 i = beg; i < end; ++i) {
  ++x[u32((data[i] - min) / ((max - min) / bin))];
}
pthread_barrier_wait(&calc); // 所有线程等待统计完成

// 树形求和
// 退出条件为本层的加法的右手项超出了线程总数，或者本层的编号不能整除 2^(高度)
for (u32 i = 0; tid + (1 << i) < th && !(tid >> i & 1); ++i) {
  u32 *y = x + (bin << i);
  for (u32 j = 0; j < bin; ++j) {
    x[j] += y[j];
  }
  pthread_barrier_wait(&reduce[i]);
}
```

简单测试一下，两边输入数据都为：

```
Enter the number of bins
10
Enter the minimum measurement
0
Enter the maximum measurement
100
Enter the number of data
1000000000
Enter the number of threads
```

```
10
```

统计实际计算耗时，`openmp` 版本为 1.574s，`pthread` 版本为 1.382s。可见后者虽然稍复杂些，但是性能也好一些。

其实分析一下，二者的前半部分是一样的。对于后半部分，`openmp` 版本理论耗时为 $O(\frac{bin*th}{th} = bin)$，`pthread` 版本理论耗时为 $O(bin \log th)$，后者应该慢一些才对。实际结果并非如此，我猜测可能与访存的局部性优劣有关。`openmp` 版本每个线程内部以步长 $bin$ 来访问内存，虽然所有线程总体上访问的内存还是连续的，但是这也只能在 L3 cache 的层面体现出来，而 L3 cache 的速度已经不算很快了。而 `pthread` 版本每个线程内部都是以步长 1 访问内存，那么 L1 cache 和 L2 cache 的作用则能体现出来。

# 9 Pthread Programming

我尝试了一下直接将文字说明翻译成代码，发现这样的效率实在太低了，完全不能接受。因此我直接实现了一个比较优化的版本，主要改动了两点：

1. $n$ 较小 ($n \leq 20$) 时调用串行版本
2. 可以开启线程时，不开启两个，而是只开启一个，本线程充当另一个线程继续计算

关键代码如下：

```
Mutex mu;
u32 remain;

usize seq_fib(usize n) {
  if (n <= 2) {
    return 1;
  }
  return seq_fib(n - 1) + seq_fib(n - 2);
}

void *par_fib(void *_n) {
  usize n = (usize)_n;
  if (n <= 20) {
    return (void *)seq_fib(n);
  }
  bool spawn = false;
  {
    Locker<Mutex> _lk(&mu);
```

```
    if (remain) {
      --remain;
      spawn = true;
    }
  }

  if (spawn) {
    pthread_t r;
    usize rres;
    pthread_create(&r, nullptr, par_fib, (void *)(n - 2));
    usize lres = (usize)par_fib((void *)(n - 1));
    pthread_join(r, (void **)&rres);
    {
      Locker<Mutex> _lk(&mu);
      ++remain;
    }
    return (void *)(lres + rres);
  } else {
    return (void *)((usize)par_fib((void *)(n - 1)) + (usize)par_fib((void *)(n - 2)));
  }
}
```

测试结果如下，注意表格中的 `p` 实际上是 `th + 1`，其中 `th` 是输入参数，表示允许**新生成**的线程数：

| | | n | | | | |
|---|---|---|---|---|---|---|
| | | 30 | 34 | 38 | 42 | 46 |
| seq_fib | | 0.002 | 0.011 | 0.049 | 0.34 | 2.321 |
| par_fib | p = 1 | 0.002 | 0.011 | 0.047 | 0.321 | 2.199 |
| | p = 2 | 0.002 | 0.008 | 0.043 | 0.201 | 1.366 |
| | p = 4 | 0.001 | 0.003 | 0.019 | 0.15 | 0.689 |
| | p = 8 | 0.001 | 0.003 | 0.015 | 0.096 | 0.443 |
| | p = 16 | 0.005 | 0.007 | 0.012 | 0.066 | 0.26 |
| | p = 24 | 0.009 | 0.009 | 0.012 | 0.055 | 0.196 |

数据基本符合预期，有一点比较奇怪的地方是 `p=1` 时的 `par_fib` 略快于 `seq_fib`。我猜测这可能是因为 `par_fib` 中调用了 `seq_fib`，在这个地方编译器可以多展开几层函数调用，从而减少了递归的开销。

使用 `openmp` 来实现也是可行的，基本的逻辑都没变，调用的东西稍有改变而已：

```
OmpMutex mu;
u32 remain;

usize par_fib(usize n) {
  if (n <= 20) {
    return seq_fib(n);
  }
  bool spawn = false;
  {
    Locker<OmpMutex> _lk(&mu);
    if (remain) {
      --remain;
      spawn = true;
    }
  }
  if (spawn) {
    usize lres, rres;
    // 开启两个并行的 section，正常的实现应该只会启动一个新线程
#pragma omp parallel sections
    {
#pragma omp section
      lres = par_fib(n - 1);
```

```
#pragma omp section
      rres = par_fib(n - 2);
    }
    {
      Locker<OmpMutex> _lk(&mu);
      ++remain;
    }
    return lres + rres;
  } else {
    return par_fib(n - 1) + par_fib(n - 2);
  }
}

  // caller
  // 设置调度策略，经测试这样能显著变快，可能是因为计算两个 fib 的耗时差距很大
  // 如果动态分配线程的话等待时间可能更少
  omp_set_dynamic(1);
  // 设置允许并行区域嵌套（默认不允许），否则最多只能在两个线程上运行
  omp_set_nested(1);
  usize res = par_fib(n);
```

测试结果如下：

| | | n | | | | |
|---|---|---|---|---|---|---|
| | | 30 | 34 | 38 | 42 | 46 |
| par_fib | p = 1 | 0.002 | 0.012 | 0.048 | 0.327 | 2.239 |
| | p = 2 | 0.001 | 0.005 | 0.031 | 0.203 | 1.389 |
| | p = 4 | 0.001 | 0.004 | 0.021 | 0.137 | 0.766 |
| | p = 8 | 0.001 | 0.005 | 0.014 | 0.097 | 0.465 |
| | p = 16 | 0.006 | 0.007 | 0.012 | 0.066 | 0.272 |
| | p = 24 | 0.008 | 0.013 | 0.025 | 0.068 | 0.228 |

可见使用 `omp section` 的性能与直接使用 `pthread` 差不多。

除此之外，`openmp` 还有一套更高层的 api：`omp task`，它可以动态的分发任务，只需要在一开始设定好最大线程数，则 `openmp` 的运行时库会帮我们处理好所需的逻辑。通过使用它，基本可以直接完成要求：

```
usize par_fib(usize n) {
  if (n <= 20) {
    return seq_fib(n);
  }
  usize lres, rres;
#pragma omp task shared(lres)
  lres = par_fib(n - 1);
#pragma omp task shared(rres)
  rres = par_fib(n - 2);
#pragma omp taskwait
  return lres + rres;
}

  // caller
  usize res;
  // +1 是因为包括了主线程在内
#pragma omp parallel num_threads(atoi(argv[2]) + 1)
#pragma omp single
  res = par_fib(n);
```

测试结果如下：

| | | n | | | | |
|---|---|---|---|---|---|---|
| | | 30 | 34 | 38 | 42 | 46 |
| par_fib | p = 1 | 0.002 | 0.012 | 0.049 | 0.336 | 2.302 |
| | p = 2 | 0.002 | 0.007 | 0.041 | 0.232 | 1.558 |
| | p = 4 | 0.001 | 0.006 | 0.041 | 0.241 | 1.077 |
| | p = 8 | 0.001 | 0.005 | 0.03 | 0.171 | 0.918 |
| | p = 16 | 0.002 | 0.005 | 0.031 | 0.154 | 1.215 |
| | p = 24 | 0.003 | 0.008 | 0.044 | 0.305 | 1.926 |

性能测试结果不是很理想，并且观察到进程的 CPU 占用率并未达到 $(th+1)*100\%$，这可能是 `openmp` 的调度策略不够理想导致的。

# 10 附：对课本/PPT 上关于 volatile 的说法的质疑

## 10.1 volatile 简介

选自cppreference: cv type qualifiers

> … Every access (read or write operation, member function call, etc.) made through a glvalue expression of volatile-qualified type is treated as a **visible side-effect** for the purposes of optimization (that is, within a single thread of execution, volatile accesses cannot be optimized out or reordered with another visible side effect that is sequenced-before or sequenced-after the volatile access. This makes volatile objects **suitable for communication with a signal handler**, but **not with another thread of execution**, see std::memory_order). …

C/C++ 中 `volatile` 关键字的唯一语义是：每次对改变量的读写都会产生副作用，并且以此为基础指示 (限制) 编译器优化。`volatile` 的直接推论如下：

1. `volatile` 变量的每次内存读写都需经过存储设备 (包括内存和高速缓存)。
2. `volatile` 变量不能编译器被完全优化掉。
3. `volatile` 变量间读写顺序不能**在汇编层面**被改变。

## 10.2 课本/PPT 上的用法的错误之处

看起来一切都很正常，对于课本上计算 $\pi$ 的程序，似乎只需要把 `sum` 和 `flag` 均标记成 `volatile`，即可保证生成的汇编代码不会随意调整临界区附近的代码顺序。

问题在于，这种顺序仅限于汇编代码层面，并不直接对应于 CPU 实际执行时的访存顺序。现代 CPU 都具备乱序执行的能力，即使汇编代码顺序看起来是合乎逻辑要求的，实际的执行结果也不完全保证符合汇编代码的顺序。

这其中，关于内存读写的顺序，不同平台的处理器的标准不尽相同。一些平台的内存顺序可参考wikipedia: memory ordering。这里简单总结几点：

1. 对于桌面电脑最常用的 `x86_64` 和 `amd64`，唯一的内存重排序是 load 前的 store 可能被重排到 load 后
   - 所以对于计算 $\pi$ 的程序，临界区的语义是可以保证的
2. 但是很多其它平台则没有这么严格的内存序，例如手机最常用的 `arm64` 平台。
   - 由于 store-store 乱序的存在，`flag` 可能在 `sum` 前被更新，这样一来对于 `sum` 的更新就位于临界区外了

我编写了一个简单的程序来证明这个理论结果，简单起见，也为了避免浮点误差对结果的干扰，这里就测试一系列整数的自增：

```c
#include <pthread.h>
#include <stdint.h>
#include <stdio.h>

#define ALL (10000000)
#define THREAD_NUM (4)
#define EACH (ALL / THREAD_NUM)

volatile intptr_t sum;
volatile intptr_t flag;

void *inc(void *_tid) {
  intptr_t tid = (intptr_t)_tid;
  for (int i = 0; i < EACH; ++i) {
    while (flag != tid)
      ;
    ++sum;
    flag = (flag + 1) % THREAD_NUM;
  }
  return NULL;
}

int main() {
  pthread_t th[THREAD_NUM];
  for (intptr_t i = 0; i < THREAD_NUM; ++i) {
    pthread_create(&th[i], NULL, inc, (void *)i);
  }
  for (intptr_t i = 0; i < THREAD_NUM; ++i) {
    pthread_join(th[i], NULL);
  }
  printf("%ld %d\n", sum, ALL);
}
```

其中主线程生成四个线程，每个线程对全局的 `sum` 自增 `EACH` 次。`sum` 和 `flag` 均用 `volatile` 标记。

在桌面平台 (`AMD R7-2700, gcc 8.3.0, -O3`) 上测试 10 次，结果都是正确的。在手机平台 (`OnePlus3 MSM8996, aarch64-linux-gnu-gcc 8.3.0, -O3`) 上测试 10 次，结果如下：

9999952 10000000
9998286 10000000
9999807 10000000
9999920 10000000
9997961 10000000
9997514 10000000
9997219 10000000
9999809 10000000
9999899 10000000
9999654 10000000

最高的出错率达到了 $\frac{10000000-9997219}{10000000} = 0.02781\%$。虽然这数字并不算大，但它毕竟不是 0。

这里分别截取一段汇编来分析一下：

amd64 平台：

```
    ...
    # 把全局变量flag读到rax
1   mov    0x2e39(%rip),%rax        # 4058 <flag>
    # 比较flag与tid
2   cmp    %rdi,%rax
    # 不等，则跳回1
3   jne    1218 <inc+0x8>
    # 全局变量sum自增1
4   mov    0x2e25(%rip),%rax        # 4050 <sum>
5   add    $0x1,%rax
6   mov    %rax,0x2e1a(%rip)        # 4050 <sum>
    # 更新flag，由于编译器的除/模常数优化，生成了一大堆代码
7   mov    0x2e1b(%rip),%rax        # 4058 <flag>
8   add    $0x1,%rax
9   cqto
10  shr    $0x3e,%rdx
11  add    %rdx,%rax
12  and    $0x3,%eax
13  sub    %rdx,%rax
    # 把更新好的flag写回全局变量
14  mov    %rax,0x2e01(%rip)        # 4058 <flag>
    # 外层循环更新和跳转
```

```
15  sub    $0x1,%ecx
16  jne    1218 <inc+0x8>
    ...
```

arm64 平台:

```
    ...
    # 辅助计算sum和flag的地址的一堆代码
    # 计算好后sum地址存在x5中，flag地址存在x2中
1   adrp    x2, 496000 <otimer.9463+0x18>
2   adrp    x5, 496000 <otimer.9463+0x18>
3   mov w4, #0x25a0                 // #9632
4   add x2, x2, #0x300
5   add x5, x5, #0x2f8
6   movk    w4, #0x26, lsl #16
7   ldr x1, [x2]
    # 比较flag与tid
8   cmp x1, x0
    # 不等，则跳回7
9   b.ne    400678 <inc+0x18>  // b.any
    # 全局变量sum自增1
    # 12可以重排到19之后，从而失去保护
10  ldr x1, [x5]
11  add x1, x1, #0x1
12  str x1, [x5]
    # 更新flag，由于编译器的除/模常数优化，生成了一大堆代码
13  ldr x1, [x2]
14  add x1, x1, #0x1
15  negs    x3, x1
16  and x1, x1, #0x3
17  and x3, x3, #0x3
18  csneg   x1, x1, x3, mi  // mi = first
    # 把更新好的flag写回全局变量
19  str x1, [x2]
    # 外层循环更新和跳转
20  subs    w4, w4, #0x1
21  b.ne    400678 <inc+0x18>  // b.any
    ...
```

基本上可以说是 C 语句和汇编语句间一一对应，看不出 volatile 的功能，**而这也恰好就是它的功能**。但是很可惜，对于 arm64 这种内存序较弱的架构，volatile 的功能用来实现临界区是不够的，就算已经是这么清晰的汇编顺序，最终结果仍然可能是错的。

## 10.3 可行的方案

cppreference 中已经写的比较明确了：volatile 并不适用于多线程间的通信。它当然不会让正确的程序变错，但是也不能让错误的程序变对；它只能让正确的程序变慢，或者让错误的程序变得看起来正确。然而我们一般并不认为“看起来正确”比“错误”要更优越一些。

回到最初的需求，本质上是希望能够保证“变量间读写顺序不能被改变”，而非仅仅“在汇编层面”不改变。为了告知处理器这一需求，需要使用特殊的机器指令，原有的指令无法表达这个语义。

C11 和 C++11 中引入了原子类型和内存序，它们可以表达这种语义。当然，一个可行的方案是直接把 sum 定义成原子类型，然后调用原子自增即可，但是这样并不能用来实现所有的临界区。这里还是使用 flag 来表示临界区的语义，虽然复杂一些，但也更通用一些：

```
#include <stdatomic.h>

intptr_t sum;
_Atomic intptr_t flag;

void *inc(void *_tid) {
  intptr_t tid = (intptr_t)_tid;
  for (int i = 0; i < EACH; ++i) {
1     while (atomic_load_explicit(&flag, memory_order_acquire) != tid)
        ;
2, 3  ++sum;
4     intptr_t new_flag = atomic_load_explicit(&flag, memory_order_relaxed);
      new_flag = (new_flag + 1) % THREAD_NUM;
5     atomic_store_explicit(&flag, new_flag, memory_order_release);
  }
  return NULL;
}
```

完整代码见 good_atomic.c。

总共有 5 个读写内存的地点，其中 1 处的 memory_order_acquire 和 5 处的 memory_order_release 保证了 1 一定最先执行，5 一定最后执行，中间的 234 还有一些自由调整的余地，但是这都不会影响到临界区的语义了。

在我的电脑和手机上又都分别进行了 10 次测试，结果都是正确的。这的确解决了问题，那么时间开销是否会比原来的版本高呢？依然可以分析汇编：

amd64 平台：

```
  mov    0x2e39(%rip),%rax        # 4058 <flag>
  cmp    %rdi,%rax
  jne    1218 <inc+0x8>
  # 用一条addq指令代替了原先的读，加，存
  addq   $0x1,0x2e24(%rip)        # 4050 <sum>
  mov    0x2e25(%rip),%rax        # 4058 <flag>
  add    $0x1,%rax
  cqto
  shr    $0x3e,%rdx
  add    %rdx,%rax
  and    $0x3,%eax
  sub    %rdx,%rax
  mov    %rax,0x2e0b(%rip)        # 4058 <flag>
  sub    $0x1,%ecx
  jne    1218 <inc+0x8>
```

因为在 amd64 平台上这样的内存序要求是天然成立的，所以并不需要任何特别的指令。不仅如此，由于去掉了 sum 变量的 volatile，编译器不再**错认为**对于 sum 读写有副作用，从而原先的读，加，存被用一条 addq 指令代替了，性能反而更好。值得注意的是，虽然有 addq 这种看起来很 “原子” 的指令，但实际上这条指令本身不是原子操作，多线程同时执行它仍然是存在竞争条件的，具体可参见StackOverFlow: Can num++ be atomic for 'int num'?。

arm64 平台：

```
  adrp  x2, 496000 <otimer.9463+0x18>
  adrp  x5, 496000 <otimer.9463+0x18>
  mov   w4, #0x25a0                  // #9632
  add   x2, x2, #0x300
  add   x5, x5, #0x2f8
  movk  w4, #0x26, lsl #16
  # 从ldr指令变成了ldar，表示acquire语义
  # LDAR Wt, [base{,#0}]
  # Load-Acquire Register: loads a word from memory addressed by base to Wt.
  ldar  x1, [x2]
  cmp   x1, x0
```

```
b.ne  400678 <inc+0x18>  // b.any
# 对sum的操作没有任何变化
ldr   x1, [x5]
add   x1, x1, #0x1
str   x1, [x5]
# 对flag的relaxed load，还是普通的ldr
ldr   x1, [x2]
add   x1, x1, #0x1
negs  x3, x1
and   x1, x1, #0x3
and   x3, x3, #0x3
csneg x1, x1, x3, mi  // mi = first
# 从str指令变成了stlr，表示release语义
# STLR Wt, [base{,#0}]
# Store-Release Register: stores a word from Wt to memory addressed by base.
stlr  x1, [x2]
subs  w4, w4, #0x1
b.ne  400678 <inc+0x18>  // b.any
```

在 arm64 上，情况反过来，没有因为去掉 volatile 而得到了额外的优化机会，但是为一次 acquire load 和一次 release store 专门生成了对应的指令，预计速度可能会有所下降。

为了验证我的预测，测试了运行 10 次的总时间如下：

| | amd64 | arm64 |
|---|---|---|
| volatile | 0m15.561s | 0m27.67s |
| atomic | 0m14.158s | 0m21.77s |

结果发现 arm64 + atomic 的组合性能并没有下降，反而显著提升了。我猜测这可能是由于语义的改变导致它在忙等待中浪费的时间更少。

## 10.4 结论

多线程编程时，可以根据自身的需求来选择合适的同步手段，如锁，信号量，条件变量等；如果临界区是简单的自增等操作，可以直接用原子变量实现；如果临界区没这么简单，但是也相当简单，可以考虑自己用原子变量 + 忙等待来实现临界区，开销可能会比使用线程库提供的锁小一些。

如果不在乎可移植性，或者对程序的运行环境非常确定，又由于某种原因用不了 C11 和 C++11 中的新 feature，可以考虑使用内嵌汇编来指示一个内存屏障：

```
asm volatile("" ::: "memory");
```

这并不会生成任何指令，只是会阻止汇编层面的重排。相比于 volatile，它不会让编译器错认为对于变量的读写有副作用 (就如同之前的 sum 的例子)，从而会有更多的优化机会。

无论如何，不要 (为了同步的目的，在 C/C++ 中) 用 volatile。