工程

基本功 | 一文讲清多线程和多线程同步

多线程编程是现代软件开发中的一项关键技术，在多线程编程中，开发者可以将复杂的任务分解为多个独立的线程，使其并行执行，从而充分利用多核处理器的优势。然而，多线程编程也带来了挑战，例如线程同步、死锁和竞态条件等问题。本篇文章将深入探讨多线程编程的基本概念（原子操作、CAS、Lock-free、内存屏障、伪共享、乱序执行等）、常见模式和最佳实践。通过具体的代码示例，希望能够帮助大家掌握多线程编程的核心技术，并在实际开发中应用这些知识，提升软件的性能和稳定性。

1 多线程

1.1 线程的概念

十多年前，主流观点主张在可能的情况下优先选择多进程而非多线程。如今，多线程编程已经成为编程领域的事实标准。多线程技术在很大程度上改善了程序的性能和响应能力，使其能够更加高效地利用系统资源，这不仅归功于多核处理器的普及和软硬件技术的进步，还归功于开发者对多线程编程的深入理解和技术创新。

那么什么是线程呢？线程是一个执行上下文，它包含诸多状态数据：每个线程有自己的执行流、调用栈、错误码、信号掩码、私有数据。Linux 内核用任务（Task）表示一个执行流。

1.1.1 执行流

一个任务里被依次执行的指令会形成一个指令序列（IP 寄存器值的历史记录），这个

指令序列就是一个指令流，每个线程会有自己的执行流。考虑下面的代码（本文代码块为 C++）：

```
int calc(int a, int b, char op) {
  int c = 0;
  if (op == '+')
    c = a + b;
  else if (op == '-')
    c = a - b;
  else if (op == '*')
    c = a * b;
  else if (op == '/')
    c = a / b;
  else
    printf("invalid operation\n");
  return c;
}
```

calc 函数被编译成汇编指令，一行 C 代码对应一个或多个汇编指令，在一个线程里执行 calc，那么这些机器指令会被依次执行。但是，被执行的指令序列跟代码顺序可能不完全一致，代码中的分支、跳转等语句，以及编译器对指令重排、处理器乱序执行会影响指令的真正执行顺序。

1.1.2 逻辑线程 vs 硬件线程

线程可以进一步区分为逻辑线程和硬件线程。

逻辑线程

程序上的线程是一个逻辑上的概念，也叫任务、软线程、逻辑线程。线程的执行逻辑由代码描述，比如编写一个函数实现对一个整型数组的元素求和：

```
int sum(int a[], int n) {
    int x = 0;
    for (int i = 0; i < n; ++i)
        x += a[i];
    return x;
}
```

这个函数的逻辑很简单，它没有再调用其他函数（更复杂的功能逻辑可以在函数里调

用其他函数）。我们可以在一个线程里调用这个函数对某数组求和；也可以把 sum 设置为某线程的入口函数，每个线程都会有一个入口函数，线程从入口函数开始执行。sum 函数描述了逻辑，即要做什么以及怎么做，偏设计；但它没有描述物质，即没有描述这个事情由谁做，事情最终需要派发到实体去完成。

硬件线程

与逻辑线程对应的是硬件线程，这是逻辑线程被执行的物质基础。

芯片设计领域，一个硬件线程通常指为执行指令序列而配套的硬件单元，一个 CPU 可能有多个核心，然后核心还可能支持超线程，1 个核心的 2 个超线程复用一些硬件。从软件的视角来看，无须区分是真正的 Core 和超出来的 VCore，基本上可以认为是 2 个独立的执行单元，每个执行单元是一个逻辑 CPU，从软件的视角看 CPU 只需关注逻辑 CPU。一个软件线程由哪个 CPU/ 核心去执行，以及何时执行，不归应用程序员管，它由操作系统决定，操作系统中的调度系统负责此项工作。

1.2 线程、核心、函数的关系

线程入口函数是线程执行的起点，线程从入口函数开始、一个指令接着一个指令执行，中间它可能会调用其他函数，那么它的控制流就转到了被调用的函数继续执行，被调用函数里还可以继续调用其他函数，这样便形成一个函数调用链。

前面的数组求和例子，如果数组特别大，则哪怕是一个简单的循环累加也可能耗费很长的时间，可以把这个整型数组分成多个小数组，或者表示成二维数组（数组的数组），每个线程负责一个小数组的求和，多个线程并发执行，最后再累加结果。

所以，为了提升处理速度，可以让多个线程在不同数据区段上执行相同（或相似）的计算逻辑，同样的处理逻辑可以有多个执行实例（线程），这对应对数据拆分线程。当然，也可以为两个线程指定不同的入口函数，让各线程执行不同的计算逻辑，这对应对逻辑拆分线程。

我们用一个例子来阐述线程、核心和函数之间的关系，假设有遛狗、扫地两类工作要做：

- 遛狗就是为狗系上绳子然后牵着它在小区里溜达一圈，这句话就描述了遛狗的逻辑，即对应到函数定义，它是一个对应到设计的静态的概念。
- 每项工作，最终需要人去做，人就对应到硬件：CPU/Core/VCore，是任务被完成的物质基础。

那什么对应软件线程？ 任务拆分。

一个例子

假设现在有 2 条狗需要遛、3 个房间需要打扫。可以把遛狗拆成 2 个任务，一个任务是遛小狗，另一个任务是遛大狗；打扫房间拆分为 3 个任务，3 个房间对应 3 个任务，执行这样的拆分策略后，将会产生 2+3=5 个任务。但如果只有 2 个人，2 个人无法同时做 5 件事，让某人在某时干某事由调度系统负责。

如果张三在遛小狗，那就对应一个线程被执行，李四在扫房间 A，则表示另一个线程在执行中，可见线程是一个动态的概念。

软件线程不会一直处于执行中，原因是多方面的。上述例子是因为人手不够，所以遛大狗的任务还处于等待被执行的状态，其他的原因包括中断、抢占、条件依赖等。比如李四扫地过程中接到一个电话，他需要去处理更紧急的事情（接电话），则扫地这个事情被挂起，李四打完电话后继续扫地，则这个线程会被继续执行。

如果只有 1 个人，则上述 5 个任务依然可以被依次或交错完成，所以多线程是一个编程模型，多线程并不一定需要多 CPU 多 Core，单 CPU 单 Core 系统依然可以运行多线程程序（虽然最大化利用多 CPU 多 Core 的处理能力是多线程程序设计的一个重要目标）。1 个人无法同时做多件事，单 CPU/ 单 Core 也不可以，操作系统通过时间分片技术应对远多于 CPU/Core 数的多任务执行的挑战。也可以把有些任务只分配给某些人去完成，这对应到 CPU 亲和性和绑核。

1.3 程序、进程、线程、协程

进程和线程是操作系统领域的两个重要概念，两者既有区别又有联系。

1.3.1 可执行程序

C/C++ 源文件经过编译器（编译 + 链接）处理后，会产生可执行程序文件，不同系统有不同格式，比如 Linux 系统的 ELF 格式、Windows 系统的 EXE 格式，可执行程序文件是一个静态的概念。

1.3.2 进程是什么

可执行程序在操作系统上的一次执行对应一个进程，进程是一个动态的概念：进程是执行中的程序。同一份可执行文件执行多次，会产生多个进程，这跟一个类可以创建多个实例一样。进程是资源分配的基本单位。

1.3.3 线程是什么

一个进程内的多个线程代表着多个执行流，这些线程以并发模式独立执行。操作系统中，被调度执行的最小单位是线程而非进程。进程是通过共享存储空间对用户呈现的逻辑概念，同一进程内的多个线程共享地址空间和文件描述符，共享地址空间意味着进程的代码（函数）区域、全局变量、堆、栈都被进程内的多线程共享。

1.3.4 进程和线程的关系

先看看 linus 的论述，在 1996 年的一封邮件里，Linus 详细阐述了他对进程和线程关系的深刻洞见，他在邮件里写道：

- 把进程和线程区分为不同的实体是背着历史包袱的传统做法，没有必要做这样的区分，甚至这样的思考方式是一个主要错误。
- 进程和线程都是一回事：一个执行上下文（context of execution），简称为 COE，其状态包括：
  - CPU 状态（寄存器等）
  - MMU 状态（页映射）
  - 权限状态（uid、gid 等）
  - 各种通信状态（打开的文件、信号处理器等）
- 传统观念认为：进程和线程的主要区别是线程有 CPU 状态（可能还包括其他

最小必要状态），而其他上下文来自进程；然而，这种区分法并不正确，这是一种愚蠢的自我设限。

- Linux 内核认为根本没有所谓的进程和线程的概念，只有 COE（Linux 称之为任务），不同的 COE 可以相互共享一些状态，通过此类共享向上构建起进程和线程的概念。
- 从实现来看，Linux 下的线程目前是 LWP 实现，线程就是轻量级进程，所有的线程都当作进程来实现，因此线程和进程都是用 task_struct 来描述的。这一点通过 /proc 文件系统也能看出端倪，线程和进程拥有比较平等的地位。对于多线程来说，原本的进程称为主线程，它们在一起组成一个线程组。
- 简言之，内核不要基于进程 / 线程的概念做设计，而应该围绕 COE 的思考方式去做设计，然后，通过暴露有限的接口给用户去满足 pthreads 库的要求。

1.3.5 协程

用户态的多执行流，上下文切换成本比线程更低，微信用协程改造后台系统后，获得了更大吞吐能力和更高稳定性。如今，协程库也进了 C++ 20 新标准。

1.4 为什么需要多线程

1.4.1 什么是多线程

一个进程内多个线程并发执行的情况就叫多线程，每个线程是一个独立的执行流，多线程是一种编程模型，它与处理器无关、跟设计有关。

需要多线程的原因包括：

- **并行计算**：充分利用多核，提升整体吞吐，加快执行速度。
- **后台任务处理**：将后台线程和主线程分离，在特定场景它是不可或缺的，如：响应式用户界面、实时系统等。

我们用 2 个例子作说明。

1.4.2 通过多线程并发提升处理能力

假设你要编写一个程序，用于统计一批文本文件的单词出现次数，程序的输入是文件名列表，输出一个单词到次数的映射。

```
// 类型别名：单词到次数的映射
using word2count = std::map<std::string, unsigned int>;

// 合并“单词到次数映射列表”
word2count merge(const std::vector<word2count>& w2c_list) {/*todo*/}

// 统计一个文件里单词出现次数（单词到次数的映射）
word2count word_count_a_file(const std::string& file) {/*todo*/}

// 统计一批文本文件的单词出现次数
word2count word_count_files(const std::vector<std::string>& files) {
    std::vector<word2count> w2c_list;
    for (auto &file : files) {
        w2c_list.push_back(word_count_a_file(file));
    }
    return merge(w2c_list);
}

int main(int argc, char* argv[]) {
    std::vector<std::string> files;
    for (int i = 1; i < argc; ++i) {
        files.push_back(argv[i]);
    }
    auto w2c = word_count_files(files);
    return 0;
}
```

这是一个单线程程序，word_count_files 函数在主线程里被 main 函数调用。如果文件不多、又或者文件不大，那么运行这个程序，很快就会得到统计结果，否则，可能要等一段长的时间才能返回结果。

重新审视这个程序会发现：函数 word_count_a_file 接受一个文件名，吐出从该文件计算出的局部结果，它不依赖于其他外部数据和逻辑，可以并发执行，所以，可以为每个文件启动一个单独的线程去运行 word_count_a_file，等到所有线程都执行完，再合并得到最终结果。

实际上，为每个文件启动一个线程未必合适，因为如果有数万个小文件，那么启动数

万个线程，每个线程运行很短暂的时间，大量时间将耗费在线程创建和销毁上，一个改进的设计：

- 开启一个线程池，线程数等于 Core 数或二倍 Core 数（策略）。
- 每个工作线程尝试去文件列表（文件列表需要用锁保护起来）里取一个文件。
  - 成功，统计这个文件的单词出现次数。
  - 失败，该工作线程就退出。
- 待所有工作线程退出后，在主线程里合并结果。

这样的多线程程序能加快处理速度，前面数组求和可以采用相似的处理，如果程序运行在多 CPU 多 Core 的机器上，就能充分利用多 CPU 多 Core 硬件优势，多线程加速执行是多线程的一个显而易见的主要目的，此其一。

1.4.3 通过多线程改变程序编写方式

其二，有些场景会有阻塞的调用，如果不用多线程，那么代码不好编写。

比如某程序在执行密集计算的同时，需要监控标准输入（键盘），如果键盘有输入，那么读取输入并解析执行，但如果获取键盘输入的调用是阻塞的，而此时键盘没有输入到来，那么其他逻辑将得不到机会执行。

代码看起来会像下面这样子：

```
// 从键盘接收输入，经解释后，会构建一个 Command 对象返回
Command command = getCommandFromStdInput();
// 执行命令
command.run();
```

针对这种情况，我们通常会开启一个单独的线程去接收输入，而用另外的线程去处理其他计算逻辑，避免处理输入阻塞其他逻辑处理，这也是多线程的典型应用，它改变了程序的编写方式，此其二。

1.5 线程相关概念

1.5.1 时间分片

CPU 先执行线程 A 一段时间，然后再执行线程 B 一段时间，然后再执行线程 A 一段时间，CPU 时间被切分成短的时间片、分给不同线程执行的策略就是 CPU 时间分片。时间分片是对调度策略的一个极度简化，实际上操作系统的调度策略非常精细，要比简单的时间分片复杂的多。如果一秒钟被分成大量的非常短的时间片，比如 100 个 10 毫秒的时间片，10 毫秒对人的感官而言太短了，以致于用户觉察不到延迟，仿佛计算机被该用户的任务所独占（实际上并不是），操作系统通过进程的抽象获得了这种任务独占 CPU 的效果（另一个抽象是进程通过虚拟内存独占存储）。

1.5.2 上下文切换

把当前正在 CPU 上运行的任务迁走，并挑选一个新任务到 CPU 上执行的过程叫调度，任务调度的过程会发生上下文切换（context swap），即保存当前 CPU 上正在运行的线程状态，并恢复将要被执行的线程的状态，这项工作由操作系统完成，需要占用 CPU 时间（sys time）。

1.5.3 线程安全函数与可重入

一个进程可以有多个线程在同时运行，这些线程可能同时执行一个函数，如果多线程并发执行的结果和单线程依次执行的结果是一样的，那么就是线程安全的，反之就不是线程安全的。

不访问共享数据，共享数据包括全局变量、static local 变量、类成员变量，只操作参数、无副作用的函数是线程安全函数，线程安全函数可多线程重入。每个线程有独立的栈，而函数参数保存在寄存器或栈上，局部变量在栈上，所以只操作参数和局部变量的函数被多线程并发调用不存在数据竞争。

C 标准库有很多编程接口都是非线程安全的，比如时间操作 / 转换相关的接口：ctime()/gmtime()/localtime()，c 标准通过提供带 _r 后缀的线程安全版本，比如：

```
char* ctime_r(const time* clock, char* buf);
```

这些接口的线程安全版本，一般都需要传递一个额外的 char * buf 参数，这样的话，函数会操作这块 buf，而不是基于 static 共享数据，从而做到符合线程安全的要求。

1.5.4 线程私有数据

因为全局变量（包括模块内的 static 变量）是进程内的所有线程共享的，但有时应用程序设计中需要提供线程私有的全局变量，这个变量仅在函数被执行的线程中有效，但却可以跨多个函数被访问。

比如在程序里可能需要每个线程维护一个链表，而会使用相同的函数来操作这个链表，最简单的方法就是使用同名而不同变量地址的线程相关数据结构。这样的数据结构可以由 Posix 线程库维护，成为线程私有数据 (Thread-specific Data，或称为 TSD)。

Posix 有线程私有数据相关接口，而 C/C++ 等语言提供 thread_local 关键字，在语言层面直接提供支持。

1.5.5 阻塞和非阻塞

一个线程对应一个执行流，正常情况下，指令序列会被依次执行，计算逻辑会往前推进。但如果因为某种原因，一个线程的执行逻辑不能继续往前走，那么我们就说线程被阻塞住了。就像下班回家，但走到家门口发现没带钥匙，只能在门口徘徊，任由时间流逝，而不能进入房间。

线程阻塞的原因有很多种，比如：

- 线程因为 acquire 某个锁而被操作系统挂起，如果 acquire 睡眠锁失败，线程会让出 CPU，操作系统会调度另一个可运行线程到该 CPU 上执行，被调度走的线程会被加入等待队列，进入睡眠状态。
- 线程调用了某个阻塞系统调用而等待，比如从没有数据到来的套接字上读数据，从空的消息队列里读消息。

- 线程在循环里紧凑的执行测试 & 设置指令并一直没有成功，虽然线程还在 CPU 上执行，但它只是忙等（相当于白白浪费 CPU），后面的指令没法执行，逻辑同样无法推进。

如果某个系统调用或者编程接口有可能导致线程阻塞，那么便被称之为阻塞系统调用；与之对应的是非阻塞调用，调用非阻塞的函数不会陷入阻塞，如果请求的资源不能得到满足，它会立即返回并通过返回值或错误码报告原因，调用的地方可以选择重试或者返回。

2 多线程同步

前面讲了多线程相关的基础知识，现在进入第二个话题，多线程同步。

2.1 什么是多线程同步

同一进程内的多个线程会共享数据，对共享数据的并发访问会出现 Race Condition，这个词的官方翻译是竞争条件，但 condition 翻译成条件令人困惑，特别是对初学者而言，它不够清晰明了，翻译软件显示 condition 有状况、状态的含义，可能翻译成竞争状况更直白。

多线程同步是指：

- 协调多个线程对共享数据的访问，避免出现数据不一致的情况。
- 协调各个事件的发生顺序，使多线程在某个点交汇并按预期步骤往前推进，比如某线程需要等另一个线程完成某项工作才能开展该线程的下一步工作。

要掌握多线程同步，需先理解为什么需要多线程同步、哪些情况需要同步。

2.2 为什么需要同步

理解为什么要同步（Why）是多线程编程的关键，它甚至比掌握多线程同步机制（How）本身更加重要。识别什么地方需要同步是编写多线程程序的难点，只有准确识别需要保护的数据、需要同步的点，再配合系统或语言提供的合适的同步机制，才

能编写安全高效的多线程程序。

下面通过几个例子解释为什么需要同步。

示例 1

有 1 个长度为 256 的字符数组 msg 用于保存消息，函数 read_msg() 和 write_msg() 分别用于 msg 的读和写：

```
// example 1
char msg[256] = "this is old msg";

char* read_msg() {
    return msg;
}

void write_msg(char new_msg[], size_t len) {
    memcpy(msg, new_msg, std::min(len, sizeof(msg)));
}

void thread1() {
    char new_msg[256] = "this is new msg, it's too looooooong";
    write_msg(new_msg, sizeof(new_msg));
}

void thread2() {
    printf("msg=%s\n", read_msg());
}
```

如果线程 1 调用 write_msg()，线程 2 调用 read_msg()，并发操作，不加保护。因为 msg 的长度是 256 字节，完成长达 256 字节的写入需要多个内存周期，在线程 1 写入新消息期间，线程 2 可能读到不一致的数据。即可能读到“this is new msg”，而后半段内容“it’s very…”线程 1 还没来得及写入，它不是完整的新消息。

在这个例子中，不一致表现为数据不完整。

示例 2

比如对于二叉搜索树（BST）的节点，一个结构体有 3 个成分：

- 一个指向父节点的指针

- 一个指向左子树的指针
- 一个指向右子树的指针

```
// example 2
struct Node {
  struct Node *parent;
      struct Node *left_child, *right_child;
};
```

这 3 个成分是有关联的，将节点加入 BST，要设置这 3 个指针域，从 BST 删除该节点，要修改该节点的父、左孩子节点、右孩子节点的指针域。对多个指针域的修改，不能在一个指令周期完成，如果完成了一个成分的写入，还没来得修改其他成分，就有可能被其他线程读到了，但此时节点的有些指针域还没有设置好，通过指针域去取数可能会出错。

示例 3

考虑两个线程对同一个整型变量做自增，变量的初始值是 0，我们预期 2 个线程完成自增后变量的值为 2。

```
// example 3
int x = 0; // 初始值为 0
void thread1() { ++x; }
void thread2() { ++x; }
```

简单的自增操作，包括三步：

- **加载**：从内存中读取变量 x 的值存放到寄存器
- **更新**：在寄存器里完成自增
- **保存**：把位于寄存器中的 x 的新值写入内存

两个线程并发执行 ++x，让我们看看真实情况是什么样的：

1. 如果 2 个线程，先后执行自增，在时间上完成错开。无论是 1 先 2 后，或是 2 先 1 后，那么 x 的最终值是 2，符合预期。但多线程并发并不能确保对一个变量的访问在时间上完全错开。

2. 如果时间上没有完全错开，假设线程 1 在 core1 上执行，线程 2 在 core2 上执行，那么，一个可能的执行过程如下：

- 首先，线程 1 把 x 读到 core1 的寄存器，线程 2 也把 x 的值加载到 core2 的寄存器，此时，存放在两个 core 的寄存器中 x 的副本都是 0。
- 然后，线程 1 完成自增，更新寄存器里 x 的值的副本（0 变 1），线程 2 也完成自增，更新寄存器里 x 的值的副本（0 变 1）。
- 再然后，线程 1 将更新后的新值 1 写入变量 x 的内存位置。
- 最后，线程 2 将更新后的新值 1 写入同一内存位置，变量 x 的最终值是 1，不符合预期。

线程 1 和线程 2 在同一个 core 上交错执行，也有可能出现同样的问题，这个问题跟硬件结构无关。之所以会出现不符合预期的情况，主要是因为“加载 + 更新 + 保存”这 3 个步骤不能在一个内存周期内完成。多个线程对同一变量并发读写，不加同步的话会出现数据不一致。

在这个例子中，不一致表现为 x 的终值既可能为 1 也可能为 2。

示例 4

用 C++ 类模板实现一个队列：

```
// example 4
template <typename T>
class Queue {
    static const unsigned int CAPACITY = 100;
    T elements[CAPACITY];
    int num = 0, head = 0, tail = -1;
public:
    // 入队
    bool push(const T& element) {
        if (num == CAPACITY) return false;
        tail = (++tail) % CAPACITY;
        elements[tail] = element;
        ++num;
        return true;
    }
    // 出队
```

```
    void pop() {
        assert(!empty());
        head = (++head) % CAPACITY;
        --num;
    }
    // 判空
    bool empty() const {
        return num == 0;
    }
    // 访队首
    const T& front() const {
        assert(!empty());
        return elements[head];
    }
};
```

代码解释：

- T elements[] 保存数据；2 个游标，分别用于记录队首 head 和队尾 tail 的位置（下标）。
- push() 接口，先移动 tail 游标，再把元素添加到队尾。
- pop() 接口，移动 head 游标，弹出队首元素（逻辑上弹出）。
- front() 接口，返回队首元素的引用。
- front()、pop() 先做断言，调用 pop()/front() 的客户代码需确保队列非空。

假设现在有一个 Queue<int> 实例 q，因为直接调用 pop 可能 assert 失败，我们封装一个 try_pop()，代码如下：

```
Queue<int> q;
void try_pop() {
    if (!q.empty()) {
        q.pop();
    }
}
```

如果多个线程调用 try_pop()，会有问题，为什么?

原因：判空 + 出队这 2 个操作，不能在一个指令周期内完成。如果线程 1 在判断队列非空后，线程 2 穿插进来，判空也为伪，这样就有可能 2 个线程竞争弹出唯一的元素。

多线程环境下，读变量然后基于值做进一步操作，这样的逻辑如果不加保护就会出错，这是由数据使用方式引入的问题。

示例 5

再看一个简单的，简单的对 int32_t 多线程读写。

```
// example 5
int32_t data[8] = {1,2,3,4,5,6,7,8};

struct Foo {
    int32_t get() const { return x; }
    void set(int32_t x) { this->x = x; }
    int32_t x;
} foo;

void thread_write1() {
    for (;;) { for (auto v : data) { foo.set(v); } }
}

void thread_write2() {
    for (;;) { for (auto v : data) { foo.set(v); } }
}

void thread_read() {
    for (;;) { printf("%d", foo.get()); }
}
```

2 个写线程 1 个读线程，写线程在无限循环里用 data 里的元素值设置 foo 对象的 x 成分，读线程简单的打印 foo 对象的 x 值。程序一直跑下去，最后打印出来的数据，会出现除 data 初始化值外的数据吗?

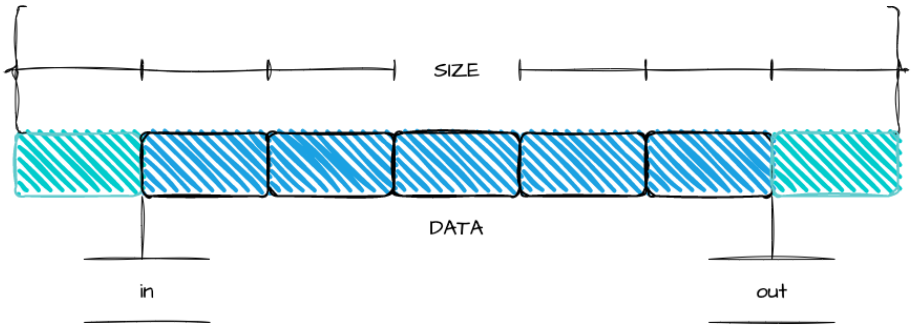
Foo::get 的实现有问题吗？如果有问题？是什么问题?

示例 6

看一个用数组实现 FIFO 队列的程序，一个线程写 put()，一个线程读 get()。

```
// example 6
#include <iostream>
#include <algorithm>
```

```
// 用数组实现的环型队列
class FIFO {
    static const unsigned int CAPACITY = 1024;  // 容量：需要满足是 2^N

    unsigned char buffer[CAPACITY];             // 保存数据的缓冲区
    unsigned int in = 0;                        // 写入位置
    unsigned int out = 0;                       // 读取位置

    unsigned int free_space() const { return CAPACITY - in + out; }
public:
    // 返回实际写入的数据长度（<= len），返回小于 len 时对应空闲空间不足
    unsigned int put(unsigned char* src, unsigned int len) {
        // 计算实际可写入数据长度（<=len）
        len = std::min(len, free_space());

        // 计算从 in 位置到 buffer 结尾有多少空闲空间
        unsigned int l = std::min(len, CAPACITY - (in & (CAPACITY - 1)));
        // 1. 把数据放入 buffer 的 in 开始的缓冲区，最多到 buffer 结尾
        memcpy(buffer + (in & (CAPACITY - 1)), src, l);
        // 2. 把数据放入 buffer 开头（如果上一步还没有放完），len - l 为 0 代表上一步
完成数据写入
        memcpy(buffer, src + l, len - l);

        in += len; // 修改 in 位置，累加，到达 uint32_max 后溢出回绕
        return len;
    }

    // 返回实际读取的数据长度（<= len），返回小于 len 时对应 buffer 数据不够
    unsigned int get(unsigned char *dst, unsigned int len) {
        // 计算实际可读取的数据长度
        len = std::min(len, in - out);

        unsigned int l = std::min(len, CAPACITY - (out & (CAPACITY - 1)));
        // 1. 从 out 位置开始拷贝数据到 dst，最多拷贝到 buffer 结尾
        memcpy(dst, buffer + (out & (CAPACITY - 1)), l);
        // 2. 从 buffer 开头继续拷贝数据（如果上一步还没拷贝完），len - l 为 0 代表上
一步完成数据获取
        memcpy(dst + l, buffer, len - l);

        out += len; // 修改 out，累加，到达 uint32_max 后溢出回绕
        return len;
    }
};
```



kfifo

环型队列只是逻辑上的概念，因为采用了数组作为数据结构，所以实际物理存储上并非环型。

- put() 用于往队列里放数据，参数 src+len 描述了待放入的数据信息。
- get() 用于从队列取数据，参数 dst+len 描述了要把数据读到哪里、以及读多少字节。
- capacity 精心选择为 2 的 n 次方，可以得到 3 个好处：
  - 非常技巧性的利用了无符号整型溢出回绕，便于处理对 in 和 out 移动
  - 便于计算长度，通过按位与操作 & 而不必除余
  - 搜索 kfifo 获得更详细的解释
- in 和 out 是 2 个游标：
  - in 用来指向新写入数据的存放位置，写入的时候，只需要简单增加 in。
  - out 用来指示从 buffer 的什么位置读取数据的，读取的时候，也只需简单增加 out。
  - in 和 out 在操作上之所以能单调增加，得益于上述 capacity 的巧妙选择。
- 为了简化，队列容量被限制为 1024 字节，不支持扩容，这不影响多线程的讨论。

写的时候，先写入数据再移动 in 游标；读的时候，先拷贝数据，再移动 out 游标；in 游标移动后，消费者才获得 get 到新放入数据的机会。

直觉告诉我们 2 个线程不加同步的并发读写，会有问题，但真有问题吗？如果有，到底有什么问题？怎么解决？

2.3 保护什么

多线程程序里，我们要保护的是数据而非代码，虽然 Java 等语言里有临界代码、sync 方法，但最终要保护的还是代码访问的数据。

2.4 串行化

如果有一个线程正在访问某共享（临界）资源，那么在它结束访问之前，其他线程不能执行访问同一资源的代码（访问临界资源的代码叫临界代码），其他线程想要访问同一资源，则它必须等待，直到那个线程访问完成，它才能获得访问的机会，现实中有很多这样的例子。比如高速公路上的汽车过检查站，假设检查站只有一个车道，则无论高速路上有多少车道，过检查站的时候只能一辆车接着一辆车，从单一车道鱼贯而入。

对多线程访问共享资源施加此种约束就叫串行化。

2.5 原子操作和原子变量

针对前面的两个线程对同一整型变量自增的问题，如果“load、update、store”这 3 个步骤是不可分割的整体，即自增操作 ++x 满足原子性，上面的程序便不会有问题。

因为这样的话，2 个线程并发执行 ++x，只会有 2 个结果：

- 线程 a ++x，然后线程 b ++x，结果是 2。
- 线程 b ++x，然后线程 a ++x，结果是 2。

除此之外，不会出现第三种情况，线程 a、b 孰先孰后，取决于线程调度，但不影响最终结果。

Linux 操作系统和 C/C++ 编程语言都提供了整型原子变量，原子变量的自增、自减等操作都是原子的，操作是原子性的，意味着它是一个不可细分的操作整体，原子变量的用户观察它，只能看到未完成和已完成 2 种状态，看不到半完成状态。

如何保证原子性是实现层面的问题，应用程序员只需要从逻辑上理解原子性，并能恰当的使用它就行了。原子变量非常适用于计数、产生序列号这样的应用场景。

2.6 锁

前面举了很多例子，阐述多线程不加同步并发访问数据会引起什么问题，下面讲解用锁如何做同步。

2.6.1 互斥锁

针对线程 1 write_msg() + 线程 2 read_msg() 的问题，如果能让线程 1 write_msg() 的过程中，线程 2 不能 read_msg()，那就不会有问题。这个要求，其实就是要让多个线程互斥访问共享资源。

互斥锁就是能满足上述要求的同步机制，互斥是排他的意思，它可以确保在同一时间，只能有一个线程对那个共享资源进行访问。

互斥锁有且只有 2 种状态：

- 已加锁（locked）状态
- 未加锁（unlocked）状态

互斥锁提供加锁和解锁两个接口：

- **加锁（acquire）**：当互斥锁处于未加锁状态时，则加锁成功（把锁设置为已加锁状态），并返回；当互斥锁处于已加锁状态时，那么试图对它加锁的线程会被阻塞，直到该互斥量被解锁。
- **解锁（release）**：通过把锁设置为未加锁状态释放锁，其他因为申请加锁而陷入等待的线程，将获得执行机会。如果有多个等待线程，只有一个会获得锁而继续执行。

我们为某个共享资源配置一个互斥锁，使用互斥锁做线程同步，那么所有线程对该资源的访问，都需要遵从“加锁、访问、解锁”的三步：

```
DataType shared_resource;
Mutex shared_resource_mutex;

void shared_resource_visitor1() {
    // step1: 加锁
    shared_resource_mutex.lock();
    // step2: operate shared_resouce
    // operation1
    // step3: 解锁
    shared_resource_mutex.unlock();
}

void shared_resource_visitor2() {
    // step1: 加锁
    shared_resource_mutex.lock();
    // step2: operate shared_resouce
    // operation2
    // step3: 解锁
    shared_resource_mutex.unlock();
}
```

shared_resource_visitor1() 和 shared_resource_visitor2() 代表对共享资源的不同操作，多个线程可能调用同一个操作函数，也可能调用不同的操作函数。

假设线程 1 执行 shared_resource_visitor1()，该函数在访问数据之前，申请加锁，如果互斥锁已经被其他线程加锁，则调用该函数的线程会阻塞在加锁操作上，直到其他线程访问完数据，释放（解）锁，阻塞在加锁操作的线程 1 才会被唤醒，并尝试加锁：

- 如果没有其他线程申请该锁，那么线程 1 加锁成功，获得了对资源的访问权，完成操作后，释放锁。
- 如果其他线程也在申请该锁，那么：
  - 如果其他线程抢到了锁，那么线程 1 继续阻塞。
  - 如果线程 1 抢到了该锁，那么线程 1 将访问资源，再释放锁，其他竞争该锁的线程得以有机会继续执行。

如果不能承受加锁失败而陷入阻塞的代价，可以调用互斥量的 try_lock() 接口，它在加锁失败后会立即返回。

注意：在访问资源前申请锁访问后释放锁，是一个编程契约，通过遵守契约而获得数据一致性的保障，它并非一种硬性的限制，即如果别的线程遵从三步曲，而另一个线程不遵从这种约定，代码能通过编译且程序能运行，但结果可能是错的。

2.6.2 读写锁

读写锁跟互斥锁类似，也是申请锁的时候，如果不能得到满足则阻塞，但读写锁跟互斥锁也有不同，读写锁有 3 个状态：

- 已加读锁状态
- 已加写锁状态
- 未加锁状态

对应 3 个状态，读写锁有 3 个接口：加读锁，加写锁，解锁：

- 加读锁：如果读写锁处于已加写锁状态，则申请锁的线程阻塞；否则把锁设置为已加读锁状态并成功返回。
- 加写锁：如果读写锁处于未加锁状态，则把锁设置为已加写锁状态并成功返回；否则阻塞。
- 解锁：把锁设置为未加锁状态后返回。

读写锁提升了线程的并行度，可以提升吞吐。它可以让多个读线程同时读共享资源，而写线程访问共享资源的时候，其他线程不能执行，所以，读写锁适合对共享资源访问“读大于写”的场合。读写锁也叫“共享互斥锁”，多个读线程可以并发访问同一资源，这对应共享的概念，而写线程是互斥的，写线程访问资源的时候，其他线程无论读写，都不可以进入临界代码区。

考虑一个场景：如果有线程 1、2、3 共享资源 x，读写锁 rwlock 保护资源，线程 1 读访问某资源，然后线程 2 以写的形式访问同一资源 x，因为 rwlock 已经被加了读锁，所以线程 2 被阻塞，然后过了一段时间，线程 3 也读访问资源 x，这时候线程 3 可以继续执行，因为读是共享的，然后线程 1 读访问完成，线程 3 继续访问，过了一段时间，在线程 3 访问完成前，线程 1 又申请读资源，那么它还是会获得访问权，但

是写资源的线程 2 会一直被阻塞。

为了避免共享的读线程饿死写线程，通常读写锁的实现，会给写线程优先权，当然这处决于读写锁的实现，作为读写锁的使用方，理解它的语义和使用场景就够了。

2.6.3 自旋锁

自旋锁（Spinlock）的接口跟互斥量差不多，但实现原理不同。线程在 acquire 自旋锁失败的时候，它不会主动让出 CPU 从而进入睡眠状态，而是会忙等，它会紧凑的执行测试和设置 (Test-And-Set) 指令，直到 TAS 成功，否则就一直占着 CPU 做 TAS。

自旋锁对使用场景有一些期待，它期待 acquire 自旋锁成功后很快会 release 锁，线程运行临界区代码的时间很短，访问共享资源的逻辑简单，这样的话，别的 acquire 自旋锁的线程只需要忙等很短的时间就能获得自旋锁，从而避免被调度走陷入睡眠，它假设自旋的成本比调度的低，它不愿耗费时间在线程调度上（线程调度需要保存和恢复上下文需要耗费 CPU）。

内核态线程很容易满足这些条件，因为运行在内核态的中断处理函数里可以通过关闭调度，从而避免 CPU 被抢占，而且有些内核态线程调用的处理函数不能睡眠，只能使用自旋锁。

而运行在用户态的应用程序，则推荐使用互斥锁等睡眠锁。因为运行在用户态应用程序，虽然很容易满足临界区代码简短，但持有锁时间依然可能很长。在分时共享的多任务系统上、当用户态线程的时间配额耗尽，或者在支持抢占式的系统上、有更高优先级的任务就绪，那么持有自旋锁的线程就会被系统调度走，这样持有锁的过程就有可能很长，而忙等自旋锁的其他线程就会白白消耗 CPU 资源，这样的话，就跟自旋锁的理念相背。

Linux 系统优化过后的 mutex 实现，在加锁的时候会先做有限次数的自旋，只有有限次自旋失败后，才会进入睡眠让出 CPU，所以，实际使用中，它的性能也足够好。此外，自旋锁必须在多 CPU 或者多 Core 架构下，试想如果只有一个核，那么它执

行自旋逻辑的时候，别的线程没有办法运行，也就没有机会释放锁。

2.6.4 锁的粒度

合理设置锁的粒度，粒度太大会降低性能，太小会增加代码编写复杂度。

2.6.5 锁的范围

锁的范围要尽量小，最小化持有锁的时间。

2.6.6 死锁

程序出现死锁有两种典型原因：

ABBA 锁

假设程序中有 2 个资源 X 和 Y，分别被锁 A 和 B 保护，线程 1 持有锁 A 后，想要访问资源 Y，而访问资源 Y 之前需要申请锁 B，而如果线程 2 正持有锁 B，并想要访问资源 X，为了访问资源 X，所以线程 2 需要申请锁 A。线程 1 和线程 2 分别持有锁 A 和 B，并都希望申请对方持有的锁，因为线程申请对方持有的锁，得不到满足，所以便会陷入等待，也就没有机会释放自己持有的锁，对方执行流也就没有办法继续前进，导致相持不下，无限互等，进而死锁。

上述的情况似乎很明显，但如果代码量很大，有时候，这种死锁的逻辑不会这么浅显，它被复杂的调用逻辑所掩盖，但抽茧剥丝，最根本的逻辑就是上面描述的那样。这种情况叫 ABBA 锁，既某个线程持有 A 锁申请 B 锁，而另一个线程持有 B 锁申请 A 锁。这种情况可以通过 try lock 实现，尝试获取锁，如果不成功，则释放自己持有的锁，而不一根筋下去。另一种解法就是锁排序，对 A/B 两把锁的加锁操作，都遵从同样的顺序（比如先 A 后 B），也能避免死锁。

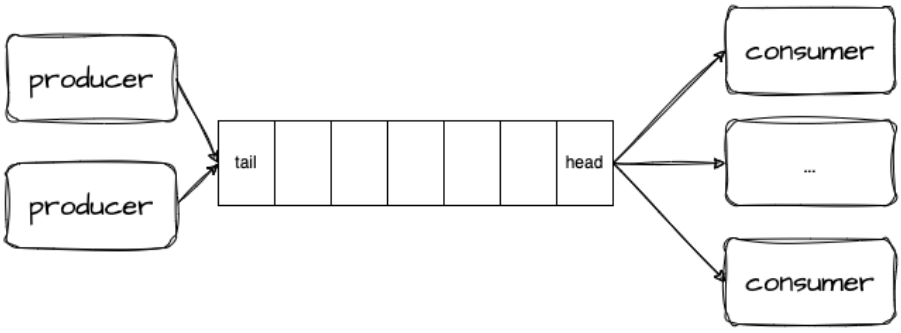
自死锁

对于不支持重复加锁的锁，如果线程持有某个锁，而后又再次申请锁，因为该锁已经被自己持有，再次申请锁必然得不到满足，从而导致死锁。

2.7 条件变量

条件变量常用于生产者消费者模式，需配合互斥量使用。

假设你要编写一个网络处理程序，I/O 线程从套接字接收字节流，反序列化后产生一个个消息（自定义协议），然后投递到一个消息队列，一组工作线程负责从消息队列取出并处理消息。这是典型的生产者 – 消费者模式，I/O 线程生产消息（往队列 put），Work 线程消费消息（从队列 get），I/O 线程和 Work 线程并发访问消息队列，显然，消息队列是竞争资源，需要同步。



proceduer–consumer

可以给队列配置互斥锁，put 和 get 操作前都先加锁，操作完成再解锁。代码差不多是这样的：

```
void io_thread() {
    while (1) {
        Msg* msg = read_msg_from_socket();
        msg_queue_mutex.lock();
        msg_queue.put(msg);
        msg_queue_mutex.unlock();
    }
}

void work_thread() {
    while (1) {
        msg_queue_mutex.lock();
        Msg* msg = msg_queue.get();
        msg_queue_mutex.unlock();
```

```
        if (msg != nullptr) {
            process(msg);
        }
    }
}
```

work 线程组的每个线程都忙于检查消息队列是否有消息，如果有消息就取一个出来，然后处理消息，如果没有消息就在循环里不停检查，这样的话，即使负载很轻，但 work_thread 还是会消耗大量的 CPU 时间。

我们当然可以在两次查询之间加入短暂的 sleep，从而让出 cpu，但是这个睡眠的时间设置为多少合适呢？设置长了的话，会出现消息到来得不到及时处理（延迟上升）；设置太短了，还是无辜消耗了 CPU 资源，这种不断问询的方式在编程上叫轮询。

轮询行为逻辑上，相当于你在等一个投递到楼下小邮局的包裹，你下楼查验没有之后就上楼回房间，然后又下楼查验，你不停的上下楼查验，其实大可不必如此，何不等包裹到达以后，让门卫打电话通知你去取呢？

条件变量提供了一种类似通知 notify 的机制，它让两类线程能够在一个点交汇。条件变量能够让线程等待某个条件发生，条件本身受互斥锁保护，因此条件变量必须搭配互斥锁使用，锁保护条件，线程在改变条件前先获得锁，然后改变条件状态，再解锁，最后发出通知，等待条件的睡眠中的线程在被唤醒前，必须先获得锁，再判断条件状态，如果条件不成立，则继续转入睡眠并释放锁。

对应到上面的例子，工作线程等待的条件是消息队列有消息（非空），用条件变量改写上面的代码：

```
void io_thread() {
    while (1) {
        Msg* msg = read_msg_from_socket();
        {
            std::lock_guard<std::mutex> lock(msg_queue_mutex);
            msg_queue.push_back(msg);
        }
        msg_queue_not_empty.notify_all();
    }
}
```

```
void work_thread() {
    while (1) {
        Msg* msg = nullptr;
        {
            std::unique_lock<std::mutex> lock(msg_queue_mutex);
            msg_queue_not_empty.wait(lock, []{ return !msg_queue.empty(); });
            msg = msg_queue.get();
        }
        process(msg);
    }
}
```

std::lock_guard 是互斥量的一个 RAII 包装类，std::unique_lock 除了会在析构函数自动解锁外，还支持主动 unlock()。

生产者在往 msg_queue 投递消息的时候，需要对 msg_queue 加锁，通知 work 线程的代码可以放在解锁之后，等待 msg_queue_not_empty 条件必须受 msg_queue_mutex 保护，wait 的第二个参数是一个 lambda 表达式，因为会有多个 work 线程被唤醒，线程被唤醒后，会重新获得锁，检查条件，如果不成立，则再次睡眠。条件变量的使用需要非常谨慎，否则容易出现不能唤醒的情况。

C 语言的条件变量、Posix 条件变量的编程接口跟 C++ 的类似，概念上是一致的，故在此不展开介绍。

2.8 lock-free 和无锁数据结构

2.8.1 锁同步的问题

线程同步分为阻塞型同步和非阻塞型同步。

- 互斥量、信号、条件变量这些系统提供的机制都属于阻塞型同步，在争用资源的时候，会导致调用线程阻塞。
- 非阻塞型同步是指在无锁的情况下，通过某种算法和技术手段实现不用阻塞而同步。

锁是阻塞同步机制，阻塞同步机制的缺陷是可能挂起你的程序，如果持有锁的线程崩

溃或者 hang 住，则锁永远得不到释放，而其他线程则将陷入无限等待；另外，它也可能导致优先级倒转等问题。所以，我们需要 lock-free 这类非阻塞的同步机制。

2.8.2 什么是 lock-free

lock-free 没有锁同步的问题，所有线程无阻碍的执行原子指令，而不是等待。比如一个线程读 atomic 类型变量，一个线程写 atomic 变量，它们没有任何等待，硬件原子指令确保不会出现数据不一致，写入数据不会出现半完成，读取数据也不会读一半。

那到底什么是 lock-free？有人说 lock-free 就是不使用 mutex / semaphores 之类的无锁（lock-Less）编程，这句话严格来说并不对。

我们先看一下 wiki 对 Lock-free 的描述：

> Lock-freedom allows individual threads to starve but guarantees system-wide throughput. An algorithm is lock-free if, when the program threads are run for a sufficiently long time, at least one of the threads makes progress (for some sensible definition of progress). All wait-free algorithms are lock-free. In particular, if one thread is suspended, then a lock-free algorithm guarantees that the remaining threads can still make progress. Hence, if two threads can contend for the same mutex lock or spinlock, then the algorithm is not lock-free. (If we suspend one thread that holds the lock, then the second thread will block.)

翻译一下：

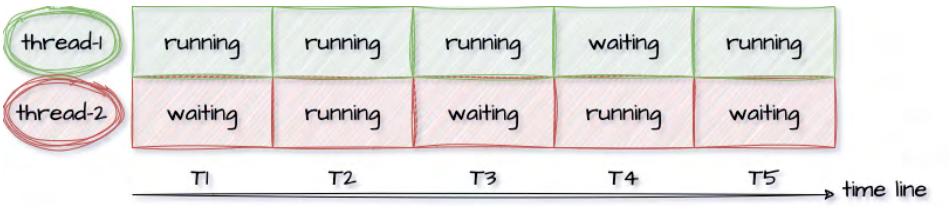
- 第 1 段：lock-free 允许单个线程饥饿但保证系统级吞吐。如果一个程序线程执行足够长的时间，那么至少一个线程会往前推进，那么这个算法就是 lock-free 的。
- 第 2 段：尤其是，如果一个线程被暂停，lock-free 算法保证其他线程依然能够往前推进。

第 1 段给 lock-free 下定义，第 2 段则是对 lock-free 作解释：如果 2 个线程竞争同一个互斥锁或者自旋锁，那它就不是 lock-free 的；因为如果暂停（Hang）持有锁的线程，那么另一个线程会被阻塞。

wiki 的这段描述很抽象，它不够直观，稍微再解释一下：lock-free 描述的是代码逻辑的属性，不使用锁的代码，大部分具有这种属性。大家经常会混淆这 lock-free 和无锁这 2 个概念。其实，lock-free 是对代码（算法）性质的描述，是属性；而无锁是说代码如何实现，是手段。

lock-free 的关键描述是：如果一个线程被暂停，那么其他线程应能继续前进，它需要有系统级（system-wide）的吞吐。

如图，两个线程在时间线上，至少有一个线程处于 running 状态。



Lock-free

我们从反面举例来看，假设我们要借助锁实现一个无锁队列，我们可以直接使用线程不安全的 std::queue + std::mutex 来做：

```
template <typename T>
class Queue {
public:
    void push(const T& t) {
        q_mutex.lock();
        q.push(t);
        q_mutex.unlock();
    }
private:
    std::queue<T> q;
    std::mutex q_mutex;
};
```

如果有线程 A/B/C 同时执行 push 方法，最先进入的线程 A 获得互斥锁。线程 B 和 C 因为获取不到互斥锁而陷入等待。这个时候，线程 A 如果因为某个原因（如出现异常，或者等待某个资源）而被永久挂起，那么同样执行 push 的线程 B/C 将被永久挂起，系统整体（system-wide）没法推进，而这显然不符合 lock-free 的要求。因此：所有基于锁（包括 spinlock）的并发实现，都不是 lock-free 的。

因为它们都会遇到同样的问题：即如果永久暂停当前占有锁的线程 / 进程的执行，将会阻塞其他线程 / 进程的执行。而对照 lock-free 的描述，它允许部分 process（理解为执行流）饿死但必须保证整体逻辑的持续前进，基于锁的并发显然是违背 lock-free 要求的。

2.8.3 CAS loop 实现 Lock-free

Lock-Free 同步主要依靠 CPU 提供的 read-modify-write 原语，著名的“比较和交换”CAS（Compare And Swap）在 X86 机器上是通过 cmpxchg 系列指令实现的原子操作，CAS 逻辑上用代码表达是这样的：

```
bool CAS(T* ptr, T expect_value, T new_value) {
   if (*ptr != expect_value) {
      return false;
   }
   *ptr = new_value;
   return true;
}
```

CAS 接受 3 个参数：

- 内存地址
- 期望值，通常传第一个参数所指内存地址中的旧值
- 新值

逻辑描述：CAS 比较内存地址中的值和期望值，如果不相同就返回失败，如果相同就将新值写入内存并返回成功。

当然这个 C 函数描述的只是 CAS 的逻辑，这个函数操作不是原子的，因为它可以划

分成几个步骤：读取内存值、判断、写入新值，各步骤之间是可以插入其他操作的。不过前面讲了，原子指令相当于把这些步骤打包，它可能是通过 lock; cmpxchg 指令实现的，但那是实现细节，程序员更应该注重在逻辑上理解它的行为。

通过 CAS 实现 Lock-free 的代码通常借助循环，代码如下：

```
do {
    T expect_value = *ptr;
} while (!CAS(ptr, expect_value, new_value));
```

1. 创建共享数据的本地副本：expect_value。
2. 根据需要修改本地副本，从 ptr 指向的共享数据里 load 后赋值给 expect_value。
3. 检查共享的数据跟本地副本是否相等，如果相等，则把新值复制到共享数据。

第三步是关键，虽然 CAS 是原子的，但加载 expect_value 跟 CAS 这 2 个步骤，并不是原子的。所以，我们需要借助循环，如果 ptr 内存位置的值没有变（*ptr \== expect_value），那就存入新值返回成功；否则说明加载 expect_value 后，ptr 指向的内存位置被其他线程修改了，这时候就返回失败，重新加载 expect_value，重试，直到成功为止。

CAS loop 支持多线程并发写，这个特点太有用了，因为多线程同步，很多时候都面临多写的问题，我们可以基于 CAS 实现 Fetch-and-add(FAA) 算法，它看起来像这样：

```
T faa(T& t) {
    T temp = t;
    while (!compare_and_swap(x, temp, temp + 1));
}
```

第一步加载共享数据的值到 temp，第二步比较 + 存入新值，直到成功。

2.8.4 无锁数据结构：lock-free stack

无锁数据结构是通过非阻塞算法而非锁保护共享数据，非阻塞算法保证竞争共享资源

的线程，不会因为互斥而让它们的执行无限期暂停；无阻塞算法是 lock-free 的，因为无论如何调度都能确保有系统级的进度。wiki 定义如下：

> A non-blocking algorithm ensures that threads competing for a shared resource do not have their execution indefinitely postponed by mutual exclusion. A non-blocking algorithm is lock-free if there is guaranteed system-wide progress regardless of scheduling.

下面是 C++ atomic compare_exchange_weak() 实现的一个 lock-free 堆栈（from CppReference）：

```
template <typename T>
struct node {
    T data;
    node* next;
    node(const T& data) : data(data), next(nullptr) {}
};

template <typename T>
class stack {
    std::atomic<node<T>*> head;
public:
    void push(const T& data) {
      node<T>* new_node = new node<T>(data);
      new_node->next = head.load(std::memory_order_relaxed);
      while (!head.compare_exchange_weak(new_node->next, new_node,
                                         std::memory_order_release,
                                         std::memory_order_relaxed));
    }
};
```

代码解析：

- 节点（node）保存 T 类型的数据 data，并且持有指向下一个节点的指针。
- std::atomic<node*> 类型表明 atomic 里放置的是 Node 的指针，而非 Node 本身，因为指针在 64 位系统上是 8 字节，等于机器字长，再长没法保证原子性。
- stack 类包含 head 成员，head 是一个指向头结点的指针，头结点指针相当于堆顶指针，刚开始没有节点，head 为 NULL。

- push 函数里，先根据 data 值创建新节点，然后要把它放到堆顶。
- 因为是用链表实现的栈，所以，如果新节点要成为新的堆顶（相当于新节点作为新的头结点插入），那么新节点的 next 域要指向原来的头结点，并让 head 指向新节点。
- new_node->next = head.load 把新节点的 next 域指向原头结点，然后 head.compare_exchange_weak(new_node->next, new_node)， 让 head 指向新节点。
- C++ atomic 的 compare_exchange_weak() 跟上述的 CAS 稍有不同，head.load() 不等于 new_node->next 的时候，它会把 head.load() 的值重新加载到 new_node->next。
- 所以，在加载 head 值和 CAS 之间，如果其他线程调用 push 操作，改变了 head 的值，那没有关系，该线程的本次 cas 失败，下次重试便可以了。
- 多个线程同时 push 时，任一线程在任意步骤阻塞 / 挂起，其他线程都会继续执行并最终返回，无非就是多执行几次 while 循环。

这样的行为逻辑显然符合 lock-free 的定义，注意用 CAS+Loop 实现自旋锁不符合 lock-free 的定义，注意区分。

2.9 程序序：Program Order

对单线程程序而言，代码会一行行顺序执行，就像我们编写的程序的顺序那样。比如：

```
a = 1;
b = 2;
```

会先执行 a=1 再执行 b=2，从程序角度看到的代码行依次执行叫程序序，我们在此基础上构建软件，并以此作为讨论的基础。

2.10 内存序：Memory Order

与程序序相对应的内存序，是指从某个角度观察到的对于内存的读和写所真正发生

的顺序。内存操作顺序并不唯一，在一个包含 core0 和 core1 的 CPU 中，core0 和 core1 有着各自的内存操作顺序，这两个内存操作顺序不一定相同。从包含多个 Core 的 CPU 的视角看到的全局内存操作顺序跟单 core 视角看到的内存操作顺序亦不同，而这种不同，对于有些程序逻辑而言，是不可接受的，例如：

程序序要求 a = 1 在 b = 2 之前执行，但内存操作顺序可能并非如此，对 a 赋值 1 并不确保发生在对 b 赋值 2 之前，这是因为：

- 如果编译器认为对 b 赋值没有依赖对 a 赋值，那它完全可能在编译期调整编译后的汇编指令顺序。
- 即使编译器不做调整，到了执行期，也有可能对 b 的赋值先于对 a 赋值执行。

虽然对一个 Core 而言，如上所述，这个 Core 观察到的内存操作顺序不一定符合程序序，但内存操作序和程序序必定产生相同的结果，无论在单 Core 上对 a、b 的赋值哪个先发生，结果上都是 a 被赋值为 1、b 被赋值为 2，如果单核上乱序执行会影响结果，那编译器的指令重排和 CPU 乱序执行便不会发生，硬件会提供这项保证。

但多核系统，硬件不提供这样的保证，多线程程序中，每个线程所工作的 Core 观察到的不同内存操作序，以及这些顺序与全局内存序的差异，常常导致多线程同步失败，所以，需要有同步机制确保内存序与程序序的一致，内存屏障（Memory Barrier）的引入，就是为了解决这个问题，它让不同的 Core 之间，以及 Core 与全局内存序达成一致。

2.11 乱序执行：Out-of-order Execution

乱序执行会引起内存顺序跟程序顺序不同，乱序执行的原因是多方面的，比如编译器指令重排、超标量指令流水线、预测执行、Cache-Miss 等。内存操作顺序无法精确匹配程序顺序，这有可能带来混乱，既然有副作用，那为什么还需要乱序执行呢？答案是为了性能。

我们先看看没有乱序执行之前，早期的有序处理器（In-order Processors）是怎么

处理指令的?

- 指令获取，从代码节内存区域加载指令到 I-Cache
- 译码
- 如果指令操作数可用（例如操作数位于寄存器中），则分发指令到对应功能模块中；如果操作数不可用，通常是需要从内存加载，则处理器会 stall，一直等到它们就绪，直到数据被加载到 Cache 或拷贝进寄存器
- 指令被功能单元执行
- 功能单元将结果写回寄存器或内存位置

乱序处理器（Out-of-order Processors）又是怎么处理指令的呢?

- 指令获取，从代码节内存区域加载指令到 I-Cache
- 译码
- 分发指令到指令队列
- 指令在指令队列中等待，一旦操作数就绪，指令就离开指令队列，那怕它之前的指令未被执行（乱序）
- 指令被派往功能单元并被执行
- 执行结果放入队列（Store Buffer），而不是直接写入 Cache
- 只有更早请求执行的指令结果写入 cache 后，指令执行结果才写入 Cache，通过对指令结果排序写入 cache，使得执行看起来是有序的

指令乱序执行是结果，但原因并非只有 CPU 的乱序执行，而是由两种因素导致：

- **编译期**：指令重排（编译器），编译器会为了性能而对指令重排，源码上先后的两行，被编译器编译后，可能调换指令顺序，但编译器会基于一套规则做指令重排，有明显依赖的指令不会被随意重排，指令重排不能破坏程序逻辑。
- **运行期**：乱序执行（CPU），CPU 的超标量流水线、以及预测执行、Cache-Miss 等都有可能导致指令乱序执行，也就是说，后面的指令有可能先于前面的指令执行。

2.12 Store Buffer

为什么需要 Store Buffer？

考虑下面的代码：

```
void set_a() {
    a = 1;
}
```

- 假设运行在 core0 上的 set_a() 对整型变量 a 赋值 1，计算机通常不会直接写穿通到内存，而是会在 Cache 中修改对应 Cache Line
- 如果 Core0 的 Cache 里没有 a，赋值操作（store）会造成 Cache Miss
- Core0 会 stall 在等待 Cache 就绪（从内存加载变量 a 到对应的 Cache Line），但 Stall 会损害 CPU 性能，相当于 CPU 在这里停顿，白白浪费着宝贵的 CPU 时间
- 有了 Store Buffer，当变量在 Cache 中没有就位的时候，就先 Buffer 住这个 Store 操作，而 Store 操作一旦进入 Store Buffer，core 便认为自己 Store 完成，当随后 Cache 就位，store 会自动写入对应 Cache。

所以，我们需要 Store Buffer，每个 Core 都有独立的 Store Buffer，每个 Core 都访问私有的 Store Buffer，Store Buffer 帮助 CPU 遮掩了 Store 操作带来的延迟。

Store Buffer 会带来什么问题?

```
a = 1;
b = 2;
assert(a == 1);
```

上面的代码，断言 a==1 的时候，需要读（load）变量 a 的值，而如果 a 在被赋值前就在 Cache 中，就会从 Cache 中读到 a 的旧值（可能是 1 之外的其他值），所以断言就可能失败。但这样的结果显然是不能接受的，它违背了最直观的程序顺序性。

问题出在变量 a 除保存在内存外，还有 2 份拷贝：一份在 Store Buffer 里，一份在

Cache 里；如果不考虑这 2 份拷贝的关系，就会出现数据不一致。那怎么修复这个问题呢?

可以通过在 Core Load 数据的时候，先检查 Store Buffer 中是否有悬而未决的 a 的新值，如果有，则取新值；否则从 cache 取 a 的副本。这种技术在多级流水线 CPU 设计的时候就经常使用，叫 Store Forwarding。有了 Store Buffer Forwarding，就能确保单核程序的执行遵从程序顺序性，但多核还是有问题，让我们考查下面的程序：

多核内存序问题

```
int a = 0; // 被 CPU1 Cache
int b = 0; // 被 CPU0 Cache

// CPU0 执行
void x() {
    a = 1;
    b = 2;
}

// CPU1 执行
void y() {
    while (b == 0);
    assert(a == 1);
}
```

假设 a 和 b 都被初始化为 0；CPU0 执行 x() 函数，CPU1 执行 y() 函数；变量 a 在 CPU1 的 local Cache 里，变量 b 在 CPU0 的 local Cache 里。

- CPU0 执行 a = 1 的时候，因为 a 不在 CPU0 的 local cache，CPU0 会把 a 的新值 1 写入 Store Buffer 里，并发送 Read Invalidate 消息给其他 CPU。
- CPU1 执行 while (b == 0)，因为 b 不在 CPU1 的 local cache 里，CPU1 会发送 Read 消息去其他 CPU 获取 b 的值。
- CPU0 执行 b = 2，因为 b 在 CPU0 的 local Cache，所以直接更新 local cache 中 b 的副本。
- CPU0 收到 CPU1 发来的 read 消息，把 b 的新值 2 发送给 CPU1；同时存放 b 的 Cache Line 的状态被设置为 Shared，以反应 b 同时被 CPU0 和

CPU1 cache 住的事实。

- CPU1 收到 b 的新值 2 后结束循环，继续执行 assert(a == 1)，因为此时 local Cache 中的 a 值为 0，所以断言失败。
- CPU1 收到 CPU0 发来的 Read Invalidate 后，更新 a 的值为 1，但为时已晚，程序在上一步已经崩了（assert 失败）。

怎么办？答案留到内存屏障一节揭晓。

2.13 Invalidate Queue

为什么需要 Invalidate Queue？

当一个变量加载到多个 core 的 Cache，则这个 Cache Line 处于 Shared 状态，如果 Core1 要修改这个变量，则需要通过发送核间消息 Invalidate 来通知其他 Core 把对应的 Cache Line 置为 Invalid，当其他 Core 都 Invalid 这个 CacheLine 后，则本 Core 获得该变量的独占权，这个时候就可以修改它了。

收到 Invalidate 消息的 core 需要回 Invalidate ACK，一个个 core 都这样 ACK，等所有 core 都回复完，Core1 才能修改它，这样 CPU 就白白浪费。

事实上，其他核在收到 Invalidate 消息后，会把 Invalidate 消息缓存到 Invalidate Queue，并立即回复 ACK，真正 Invalidate 动作可以延后再做，这样一方面因为 Core 可以快速返回别的 Core 发出的 Invalidate 请求，不会导致发生 Invalidate 请求的 Core 不必要的 Stall，另一方面也提供了进一步优化可能，比如在一个 CacheLine 里的多个变量的 Invalidate 可以攒一次做了。

但写 Store Buffer 的方式其实是 Write Invalidate，它并非立即写入内存，如果其他核此时从内存读数，则有可能不一致。

2.14 内存屏障

那有没有方法确保对 a 的赋值一定先于对 b 的赋值呢？有，内存屏障被用来提供这个保障。

内存屏障（Memory Barrier），也称内存栅栏、屏障指令等，是一类同步屏障指令，是 CPU 或编译器在对内存随机访问的操作中的一个同步点，同步点之前的所有读写操作都执行后，才可以开始执行此点之后的操作。语义上，内存屏障之前的所有写操作都要写入内存；内存屏障之后的读操作都可以获得同步屏障之前的写操作的结果。

内存屏障，其实就是提供一种机制，确保代码里顺序写下的多行，会按照书写的顺序，被存入内存，主要是解决 Store Buffer 引入导致的写入内存间隙的问题。

```
void x() {
    a = 1;
    wmb();
    b = 2;
}
```

像上面那样在 a=1 和 b=2 之间插入一条内存屏障语句，就能确保 a=1 先于 b=2 生效，从而解决了内存乱序访问问题，那插入的这句 smp_mb()，到底会干什么呢?

回忆前面的流程，CPU0 在执行完 a = 1 之后，执行 smp_mb() 操作，这时候，它会给 Store Buffer 里的所有数据项做一个标记（marked），然后继续执行 b = 2，但这时候虽然 b 在自己的 cache 里，但由于 store buffer 里有 marked 条目，所以，CPU0 不会修改 cache 中的 b，而是把它写入 Store Buffer；所以 CPU0 收到 Read 消息后，会把 b 的 0 值发给 CPU1，所以继续在 while (b) 自旋。

简而言之，Core 执行到 write memory barrier（wmb）的时候，如果 Store Buffer 还有悬而未决的 store 操作，则都会被 mark 上，直到被标注的 Store 操作进入内存后，后续的 Store 操作才能被执行，因此 wmb 保障了 barrier 前后操作的顺序，它不关心 barrier 前的多个操作的内存序，以及 barrier 后的多个操作的内存序，是否与 Global Memory Order 一致。

```
a = 1;
b = 2;
wmb();
c = 3;
d = 4;
```

wmb() 保证“a=1;b=2”发生在“c=3;d = 4”之前，不保证 a = 1 和 b = 2 的内存序，也不保证 c = 3 和 d = 4 的内部序。

Invalidate Queue 的引入的问题

就像引入 Store Buffer 会影响 Store 的内存一致性，Invalidate Queue 的引入会影响 Load 的内存一致性。因为 Invalidate queue 会缓存其他核发过来的消息，比如 Invalidate 某个数据的消息被 delay 处置，导致 core 在 Cache Line 中命中这个数据，而这个 Cache Line 本应该被 Invalidate 消息标记无效。如何解决这个问题呢?

一种思路是硬件确保每次 load 数据的时候，需要确保 Invalidate Queue 被清空，这样可以保证 load 操作的强顺序

软件的思路，就是仿照 wmb() 的定义，加入 rmb() 约束。rmb() 给我们的 invalidate queue 加上标记。当一个 load 操作发生的时候，之前的 rmb() 所有标记的 invalidate 命令必须全部执行完成，然后才可以让随后的 load 发生。这样，我们就在 rmb() 前后保证了 load 观察到的顺序等同于 global memory order

所以，我们可以像下面这样修改代码：

```
a = 1;
wmb();
b = 2;
while(b != 2) {};
rmb();
assert(a == 1);
```

系统对内存屏障的支持

gcc 编译器在遇到内嵌汇编语句 asm volatile(“”:::“memory”)，将以此作为一条内存屏障，重排序内存操作，即此语句之前的各种编译优化将不会持续到此语句之后。

Linux 内核提供函数 barrier() 用于让编译器保证其之前的内存访问先于其之后的完成。

```
#define barrier() __asm__ __volatile__("" ::: "memory")
```

CPU 内存屏障：

- 通用 barrier，保证读写操作有序，mb() 和 smp_mb()
- 写操作 barrier，仅保证写操作有序，wmb() 和 smp_wmb()
- 读操作 barrier，仅保证读操作有序，rmb() 和 smp_rmb()

小结

- 为了提高处理器的性能，SMP 中引入了 store buffer(以及对应实现 store buffer forwarding) 和 invalidate queue。
- store buffer 的引入导致 core 上的 store 顺序可能不匹配于 global memory 的顺序，对此，我们需要使用 wmb() 来解决。
- invalidate queue 的存在导致 core 上观察到的 load 顺序可能与 global memory order 不一致，对此，我们需要使用 rmb() 来解决。
- 由于 wmb() 和 rmb() 分别只单独作用于 store buffer 和 invalidate queue，因此这两个 memory barrier 共同保证了 store/load 的顺序。

3 伪共享

多个线程同时读写同一个 Cache Line 中的变量、导致 CPU Cache 频繁失效，从而使得程序性能下降的现象称为**伪共享**（False Sharing）。

```
const size_t shm_size = 16*1024*1024; //16M
static char shm[shm_size];
std::atomic<size_t> shm_offset{0};

void f() {
    for (;;) {
        auto off = shm_offset.fetch_add(sizeof(long));
        if (off >= shm_size) break;
        *(long*)(shm + off) = off; // 赋值
    }
}
```

考察上面的程序：shm 是一块 16M 字节的内存，我测试的机器的 L3 Cache 是 32M，16M 字节能确保 shm 在 Cache 里放得下。f() 函数的循环里，视 shm 为 long 类型的数组，依次给每个元素赋值，shm_offset 用于记录偏移位置，shm_offset.fetch_add(sizeof(long)) 原子性的增加 shm_offset 的值（因为 x86_64 系统上 long 的长度为 8，所以 shm_offset 每次增加 8），并返回增加前的值，对 shm 上 long 数组的每个元素赋值后，结束循环从函数返回。

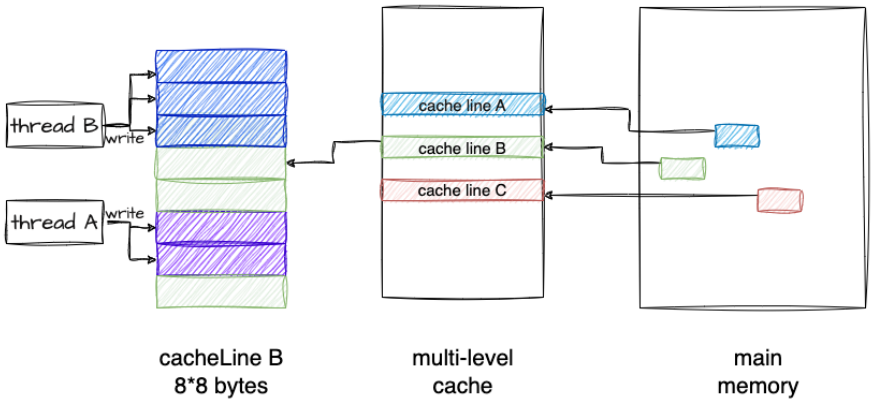
因为 shm_offset 是 atomic 类型变量，所以多线程调用 f() 依然能正常工作，虽然多个线程会竞争 shm_offset，但每个线程会排他性的对各 long 元素赋值，多线程并行会加快对 shm 的赋值操作。我们加上多线程调用代码：

```
std::atomic<size_t> step{0};

const int THREAD_NUM = 2;

void work_thread() {
    const int LOOP_N = 10;
    for (int n = 1; n <= LOOP_N; ++n) {
        f();
        ++step;
        while (step.load() < n * THREAD_NUM) {}
        shm_offset = 0;
    }
}

int main() {
    std::thread threads[THREAD_NUM];
    for (int i = 0; i < THREAD_NUM; ++i) {
        threads[i] = std::move(std::thread(work_thread));
    }
    for (int i = 0; i < THREAD_NUM; ++i) {
        threads[i].join();
    }
    return 0;
}
```

- main 函数里启动 2 个工作线程 work_thread。
- 工作线程对 shm 共计赋值 10 轮，后面的每一轮会访问 Cache 里的 shm 数据，step 用于 work_thread 之间每一轮的同步。

- 工作线程调用完 f() 后会增加 step，等 2 个工作线程都调用完之后，step 的值增加到 n * THREAD_NUM 后，while() 会结束循环，重置 shm_offset，重新开始新一轮对 shm 的赋值。

如图所示：



false-sharing-1

编译后执行上面的程序，产生如下的结果：

```
time ./a.out

real 0m3.406s
user 0m6.740s
sys 0m0.040s
```

time 命令用于时间测量，a.out 程序运行完成后会打印耗时，real 列显式耗时 3.4 秒。

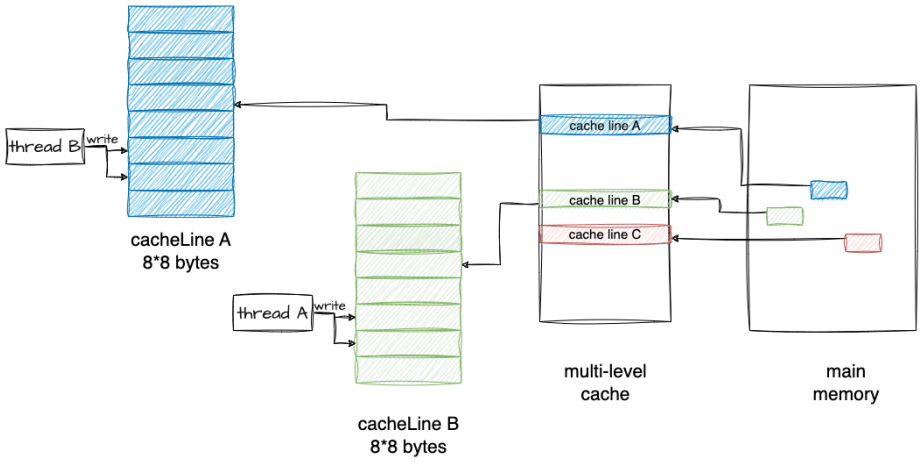
3.1 改进版 f_fast

我们稍微修改一下 f 函数，改进版 f 函数取名 f_fast：

```
void f_fast() {
    for (;;) {
        const long inner_loop = 16;
```

```
            auto off = shm_offset.fetch_add(sizeof(long) * inner_loop);
            for (long j = 0; j < inner_loop; ++j) {
                if (off >= shm_size) return;
                *(long*)(shm + off) = j;
                off += sizeof(long);
            }
        }
    }
```

for 循环里，shm_offset 不再是每次增加 8 字节（sizeof(long)），而是 8*16=128 字节，然后在内层的循环里，依次对 16 个 long 连续元素赋值，然后下一轮循环又再次增加 128 字节，直到完成对 shm 的赋值。如图所示：



no-false-sharing

编译后重新执行程序，结果显示耗时降低到 0.06 秒，对比前一种耗时 3.4 秒，f_fast 性能提升明显。

```
time ./a.out

real 0m0.062s
user 0m0.110s
sys 0m0.012s
```

f 和 f_fast 的行为差异

shm 数组总共有 2M 个 long 元素，因为 16M / sizeof(long) 得 2M：

1、f() 函数行为逻辑

- 线程 1 和线程 2 的 work_thread 里会交错地对 shm 元素赋值，shm 的 2M 个 long 元素，会顺序的一个接一个的派给 2 个线程去赋值。
- 可能的行为：元素 1 由线程 1 赋值，元素 2 由线程 2 赋值，然后元素 3 和元素 4 由线程 1 赋值，然后元素 5 又由线程 2 赋值…
- 每次分派元素的时候，shm_offset 都会 atomic 的增加 8 字节，所以不会出现 2 个线程给同 1 个元素赋值的情况。

2、f_fast() 函数行为逻辑

- 每次派元素的时候，shm_offset 原子性的增加 128 字节（16 个元素）。
- 这 16 个字节作为一个整体，派给线程 1 或者线程 2；虽然线程 1 和线程 2 还是会交错的操作 shm 元素，但是以 16 个元素（128 字节）为单元，这 16 个连续的元素不会被分开派发给不同线程。
- 一次派发的 16 个元素，会在一个线程里被一个接着一个的赋值（内部循环里）。

3.2 为什么 f_fast 更快

第一眼感觉是 f_fast() 里 shm_offset.fetch_add() 调用频次降低到了原来的 1/16，有理由怀疑是原子变量的竞争减少导致程序执行速度加快。为了验证，让我们在内层的循环里加一个原子变量 test 的 fetch_add，test 原子变量的竞争会像 f() 函数里 shm_offset.fetch_add() 一样激烈，修改后的 f_fast 代码变成下面这样：

```
void f_fast() {
    for (;;) {
        const long inner_loop = 16;
        auto off = shm_offset.fetch_add(sizeof(long) * inner_loop);
        for (long j = 0; j < inner_loop; ++j) {
            test.fetch_add(1);
            if (off >= shm_size) return;
```

```
                *(long*)(shm + off) = j;
                off += sizeof(long);
            }
        }
    }
```

为了避免 test.fetch_add(1) 的调用被编译器优化掉，我们在 main 函数的最后把 test 的值打印出来。编译后测试一下，结果显示：执行时间只是稍微增加到 real 0m0.326s，很显然，并不是 atomic 的调用频次减少导致性能飙升。

重新审视 f() 循环里的逻辑：f() 循环里的操作很简单：原子增加、判断、赋值。我们把 f() 的里赋值注释掉，再测试一下，发现它的速度得到了很大提升，看来是 *(long*)(shm + off) = off* 这一行代码执行慢，但这明明只是一行赋值。我们把它反汇编来看，它只是一个 mov 指令，源操作数是寄存器，目标操作数是内存地址，从寄存器拷贝数据到一个内存地址，为什么会这么慢呢?

3.3 原因

现在揭晓答案：导致 f() 性能底下的元凶是伪共享（false sharing）。那什么是伪共享？要说清这个问题，还得联系 CPU 的架构以及 CPU 怎么访问数据，回顾一下关于多核 Cache 结构。

背景知识

现代 CPU 可以有多个核，每个核有自己的 L1-L2 缓存，L1 又区分数据缓存（L1-DCache）和指令缓存（L1-ICache），L2 不区分数据和指令 Cache，而 L3 是跨核共享的，L3 通过内存总线连接到内存，内存被所有 CPU 所有 Core 共享。

CPU 访问 L1 Cache 的速度大约是访问内存的 100 倍，Cache 作为 CPU 与内存之间的缓存，减少对内存的访问频率。

从内存加载数据到 Cache 的时候，是以 Cache Line 为长度单位的，Cache Line 的长度通常是 64 字节，所以，那怕只读一个字节，但是包含该字节的整个 Cache Line 都会被加载到缓存，同样，如果修改一个字节，那么最终也会导致整个 Cache

Line 被冲刷到内存。

如果一块内存数据被多个线程访问，假设多个线程在多个 Core 上并行执行，那么它便会被加载到多个 Core 的的 Local Cache 中；这些线程在哪个 Core 上运行，就会被加载到哪个 Core 的 Local Cache 中，所以，内存中的一个数据，在不同 Core 的 Cache 里会同时存在多份拷贝。

那么，便会存在缓存一致性问题。当一个 Core 修改其缓存中的值时，其他 Core 不能再使用旧值。该内存位置将在所有缓存中失效。此外，由于缓存以缓存行而不是单个字节的粒度运行，因此整个缓存行将在所有缓存中失效。如果我们修改了 Core1 缓存里的某个数据，则该数据所在的 Cache Line 的状态需要同步给其他 Core 的缓存，Core 之间可以通过核间消息同步状态，比如通过发送 Invalidate 消息给其他核，接收到该消息的核会把对应 Cache Line 置为无效，然后重新从内存里加载最新数据。

当然，被加载到多个 Core 缓存中的同一 Cache Line，会被标记为共享（Shared）状态，对共享状态的缓存行进行修改，需要先获取缓存行的修改权（独占），MESI 协议用来保证多核缓存的一致性，更多的细节可以参考 MESI 的文章。

示例分析

假设线程 1 运行在 Core1，线程 2 运行在 Core2。

- 因为 shm 被线程 1 和线程 2 这两个线程并发访问，所以 shm 的内存数据会以 Cache Line 粒度，被同时加载到 2 个 Core 的 Cache，因为被多核共享，所以该 Cache Line 被标注为 Shared 状态。
- 假设线程 1 在 offset 为 64 的位置写入了一个 8 字节的数据（sizeof(long)），要修改一个状态为 Shared 的 Cache Line，Core1 会发送核间通信消息到 Core2，去拿到该 Cache Line 的独占权，在这之后，Core1 才能修改 Local Cache
- 线程 1 执行完 shm_offset.fetch_add(sizeof(long)) 后，shm_offset 会增加到 72。

- 这时候 Core2 上运行的线程 2 也会执行 shm_offset.fetch_add(sizeof(long))，它返回 72 并将 shm_offset 增加到 80。
- 线程 2 接下来要修改 shm[72] 的内存位置，因为 shm[64] 和 shm[72] 在一个 Cache Line，而这个 Cache Line 又被置为 Invalidate，所以，它需要从内存里重新加载这一个 Cache Line，而在这之前，Core1 上的线程 1 需要把 Cache Line 冲刷到内存，这样线程 2 才能加载最新的数据。

这种交替执行模式，相当于 Core1 和 Core2 之间需要频繁的发送核间消息，收到消息的 Core 的 Cache Line 被置为无效，并重新从内存里加载数据到 Cache，每次修改后都需要把 Cache 中的数据刷入内存，这相当于废弃掉了 Cache，因为每次读写都直接跟内存打交道，Cache 的作用不复存在，这就是性能低下的原因。

这种多核多线程程序，因为并发读写同一个 Cache Line 的数据（临近位置的内存数据），导致 Cache Line 的频繁失效，内存的频繁 Load/Store，从而导致性能急剧下降的现象叫伪共享，伪共享是性能杀手。

3.4 另一个伪共享的例子

假设线程 x 和 y，分别修改 Data 的 a 和 b 变量，如果被频繁调用，也会出现性能低下的情况，怎么规避呢?

```
struct Data {
    int a;
    int b;
} data; // global

void thread1() {
    data.a = 1;
}

void thread2() {
    data.b = 2;
}
```

空间换时间

避免 Cache 伪共享导致性能下降的思路是用空间换时间，通过增加填充，让 a 和 b 两个变量分布到不同的 Cache Line，这样对 a 和 b 的修改就会作用于不同 Cache Line，就能避免 Cache 失效的问题。

```
struct Data {
    int a;
    int padding[60];
    int b;
};
```

在 Linux kernel 中存在 __cacheline_aligned_in_smp 宏定义用于解决 false sharing 问题。

```
#ifdef CONFIG_SMP
#define __cacheline_aligned_in_smp __cacheline_aligned
#else
#define __cacheline_aligned_in_smp
#endif

struct Data {
    int a;
    int b __cacheline_aligned_in_smp;
};
```

从上面的宏定义，可以看到：

- 在多核系统里，该宏定义是 __cacheline_aligned，也就是 Cache Line 的大小
- 在单核系统里，该宏定义是空的

4 小结

pthread 接口提供的几种同步原语如下：

同步原语	出现背景	常见应用场景	优势	劣势	备注
互斥锁（mutex）	每次只有一个线程可以向前执行	大部分场景	使用简单、锁竞争不激烈的时候性能非常高	竞争激烈时候性能较低	第一选择，pthread有多种锁定特性，建议只使用标准互斥类型
读写锁（read_write_lock）	允许更高的并行度，一次只有一个线程可以占有写模式的锁，但是多个读线程可以占用读模式的读写锁	适用于读的次数远大于写的情况	读多写少的情况下性能较好	可能导致写饥饿、读饥饿。（取决于实现）	可以实现为：读优先、写优先、公平。
条件变量（condition_variable）	允许线程以无竞争的方式等到特定的条件发生，同时避免忙等待：拥有通知机制。	生产者-消费者模型	可以用于通知线程条件满足	唤醒线程、重新获取锁和重新检查条件可能导致额外的性能开销。	需要配合互斥量使用、需要check虚假唤醒和唤醒多个（posix标准允许唤醒一个以上线程）
自旋锁（spin）	线程并不希望在重新调度上花时间。不通过休眠使线程阻塞，而是通过忙等近似阻塞，用来实现一些其他类型的锁	锁被持有的时间短，	非抢占式内核避免中断、一些系统函数和系统库使用	自旋的时间可能会比预期长（时间片作用下）	用户层不推荐使用，因为互斥锁足够高效
屏障（barrier）	协调多个线程并行工作，每个线程等待直到全部线程都到达一点然后从该点进行执行	适用于固定数量的线程的 并行算法、数据初始化等	简化同步的代码	某些实现中，当线程在屏障上等待时会消耗 CPU 资源（忙等） 只适用于固定数量的线程	和Memory Barrier是两种，使用pthread_barrier_* 接口。 类似Java中的CyclicBarrier或者C++20的std::barrier

由于 linux 下线程和进程本质都是 LWP，那么进程间通信使用的 IPC（管道、FIFO、消息队列、信号量）线程间也可以使用，也可以达到相同的作用。 但是由于 IPC 资源在进程退出时不会清理（因为它是系统资源），因此不建议使用。

以下是一些非锁但是也能实现线程安全或者部分线程安全的常见做法：

名称	简介	常见应用场景	优势	劣势	备注
原子赋值	简单类型的对齐读取和写入通常是原子的。比如 int32、int64	双buffer切换时候的指针赋值； 共享内存中简单类型之间修改	无锁且实现简单		本质上单条处理器指令是原子的 并非所有处理器架构都保证是原子的
简单原子变量(atomic)	通过编译器、语言等实现原子的CPU指令	原生类型的自增自减和立即数赋值等、不强调先后只追求最终结果的正确	性能高、实现简单		gcc、C、C++、Java等都有实现 所有原子类型都不支持拷贝 没有浮点类型的原子变量
CAS(简单原子变量就是一种weak的CAS)	内存屏障	对执行的先后顺序有严格要求	性能高、实现简单		在竞争严重的时候，自旋可能非常浪费CPU
双buffer	在内存中保存两份	更新不频繁的数据	性能高、无需加锁	浪费空间	只适用于一写多读的场景
延迟删除双buffer（Double Buffering with Deferred Deletion）	在更新后短期内双buffer，而后删除旧版本，通过指针赋值的原子性切换到新数据	更新不频繁的数据 读多写少的场景	性能高、无需加锁	更新频率有限制	只适用于一写多读的场景
thread_local	每个线程持有一份数据，彻底摆脱线程同步	线程间无需实时交互	性能高、无需加锁	每个线程都有一个实例	
per-cpu变量	每个处理器都分配了该变量的副本	绑核后无锁读写	性能高、无需加锁		参考：DEFINE_PER_CPU，get_cpu_var等
RCU（Read-Copy-Update）	需要修改时候创建副本，然后切换副本，	读多写少的场景			参见：rcu_read_lock， https://liburcu.org/

可以看到，上面很多做法都是采用了副本，尽量避免在 thread 中间共享数据。最快的同步就是没同步（The fastest synchronization of all is the kind that never takes place），Share nothing is best。