> 作 者：道哥，10+年嵌入式开发老兵，专注于：C/C++、嵌入式、Linux。
>
> 关注下方公众号，回复【书籍】，获取 Linux、嵌入式领域经典书籍；回复【PDF】，获取所有原创文章(PDF 格式)。

目录

别人的经验，我们的阶梯！

在开发中经常遇到多个并发执行的线程，需要对同一个资源进行访问，也就是发生资源竞争。

在这种场景中，一般的做法就是加锁，通过锁机制对临界区进行保护，以达到资源独占的目的。

这篇文章主要描述的就是使用分段锁来解决这个问题，说起来很简单：就是把锁的粒度降低，以达到资源独占、最大程度避免竞争的目的。

问题描述

周末和朋友聊天说到最近的工作，他们有个项目，需要把之前的一个单片机程序，移植到x86平台。

由于历史的原因，代码中到处都充斥着全局变量，你懂得：在以前的单片机中充斥着大量的全局变量，方便、好用啊！

> 在代码中，尽量避免使用全局变量。坏处有：不方便模块化，函数不可重入，耦合性大。。。

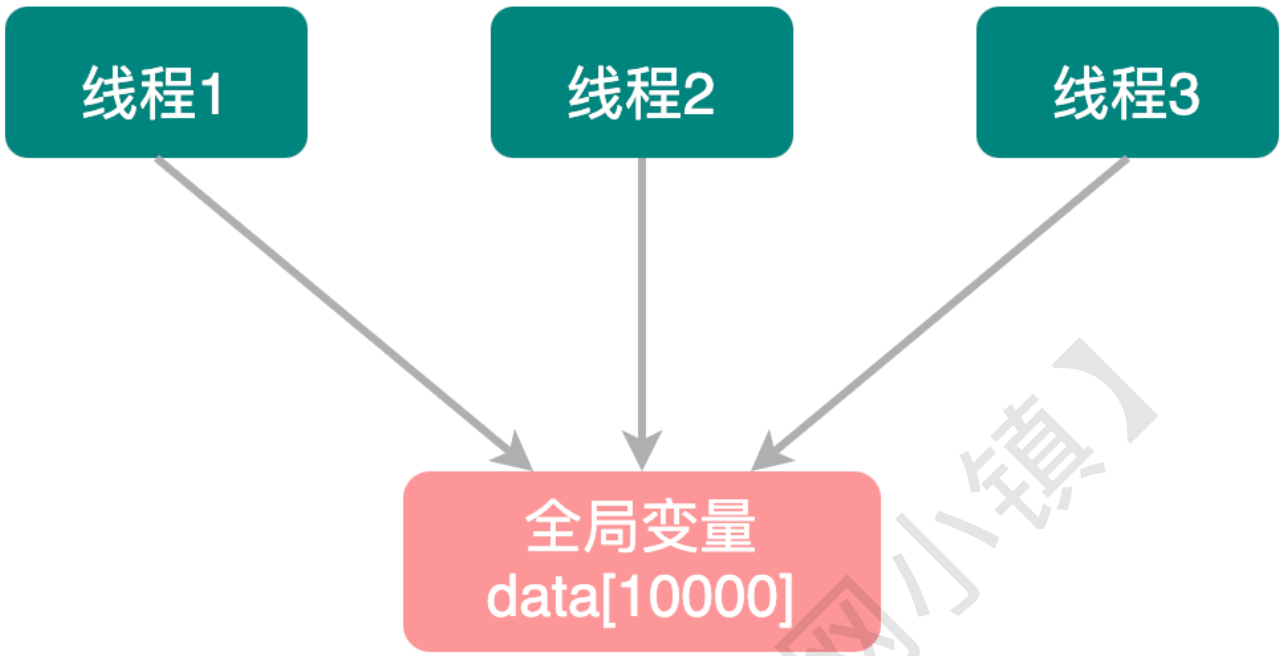
由于大部分的单片机都只有一个CPU，是真正的串行操作。

也许你会说：会发生中断啊，这也是一种异步操作。

没错，但是可以在访问全局变量的地方把中断关掉，这样就不会避免了资源竞争的情况了。

但是，移植到x86平台之后，在多核的情况下，多个线程(任务)是真正的并发执行序列。

如果多个线程同时操作某一个全局变量，就一定存在竞争的情况。



针对这个问题，首先想到的方案就是：分配一般互斥锁，无论哪个线程想访问全局变量，首先获取到锁，然后才能操作全局变量，操作完成之后再释放锁。

但是，这个方案有一个很大的问题，就是：当并发线程很多的情况下，程序的执行效率太低。

他们最后的解决方案是分段加锁，也就是对全局变量按照数据索引进行分割，每一段数据分配一把锁。

至于每一段的数据长度是多少，这需要根据实际的业务场景进行调整，以达到最优的性能。

回来之后，我觉得这个想法非常巧妙。

这个机制看起来很简单，但是真的能解决大问题。

于是我就写了一段代码来测试一下：这种方案对程序的性能有多大的影响。

> 代码已经上传到网盘了，文末有具体的下载地址。

测试代码

在测试代码中，定义了一个全局变量：

```
volatile int test_data[DATA_TOTAL_NUM];
```

数组的长度是10000(宏定义：DATA_TOTAL_NUM)，然后创建100个线程来并发访问这个全局变量，每个线程访问100000次。

然后执行3个测试用例：

测试1：不使用锁

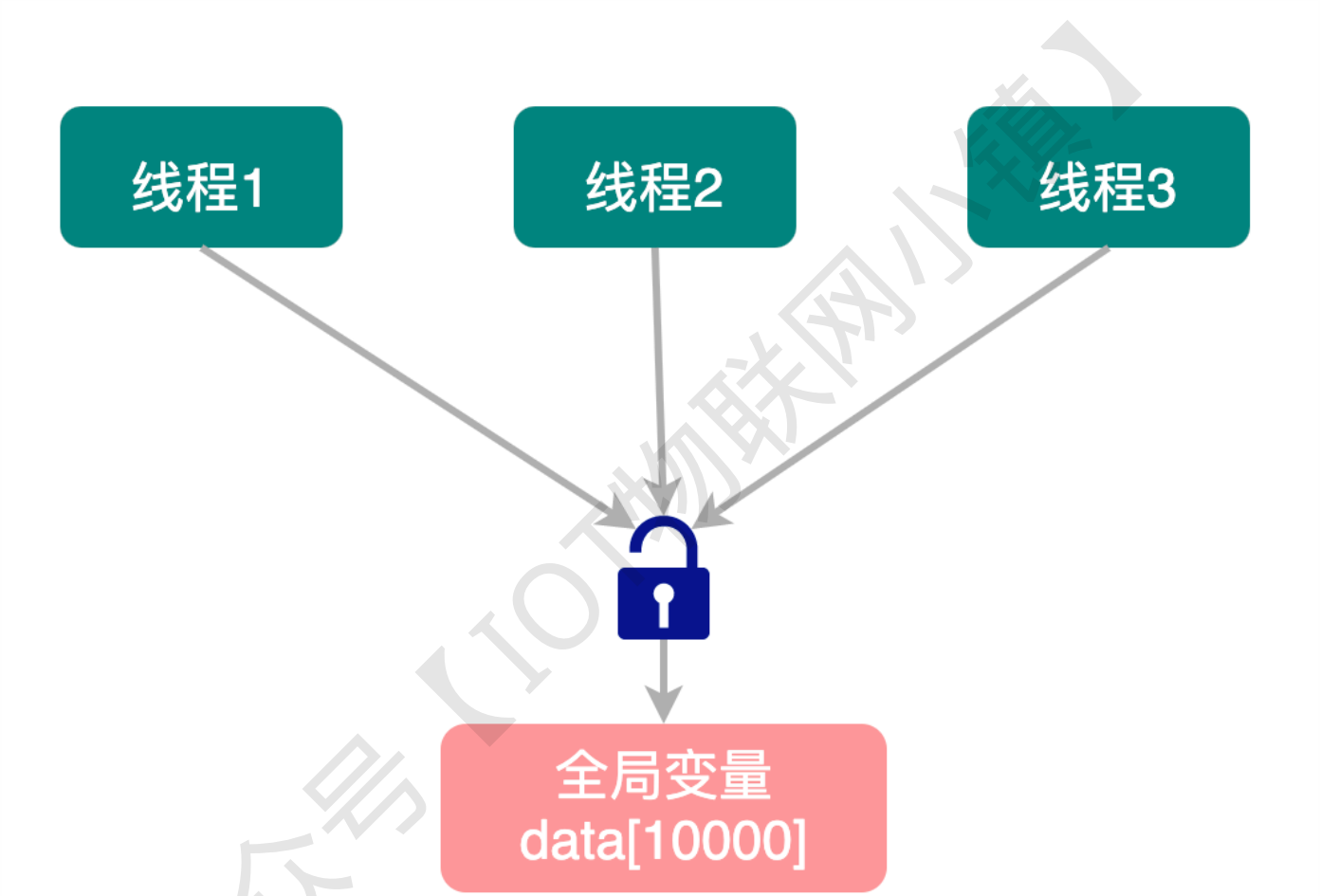
00个线程同时操作全局变量，访问的数据索引随机产生，最后统计每个线程的平均执行时间。

不使用锁的话，最后的结果(全局变量中的数据内容)肯定是错误的，这里仅仅是为了看一下时间消耗。

测试2：使用一把全局锁(大锁)

100个线程使用一把锁。

每个线程在操作全局变量之前，首先要获取到这把锁，然后才能操作全局变量，否则的话只能阻塞着等其它线程释放锁。



测试3：使用分段锁

根据全局变量的长度，分配多把锁。

每个线程在访问的时候，根据访问的数据索引，获取不同的锁，这样就降低了竞争的几率。

在这个测试场景中，全局变量test_data的长度是10000，每100个数据分配一把锁，那么一共需要100把锁。

比如，在某个时刻：

线程1想访问test_data[110]，

线程2想访问test_data[120]，

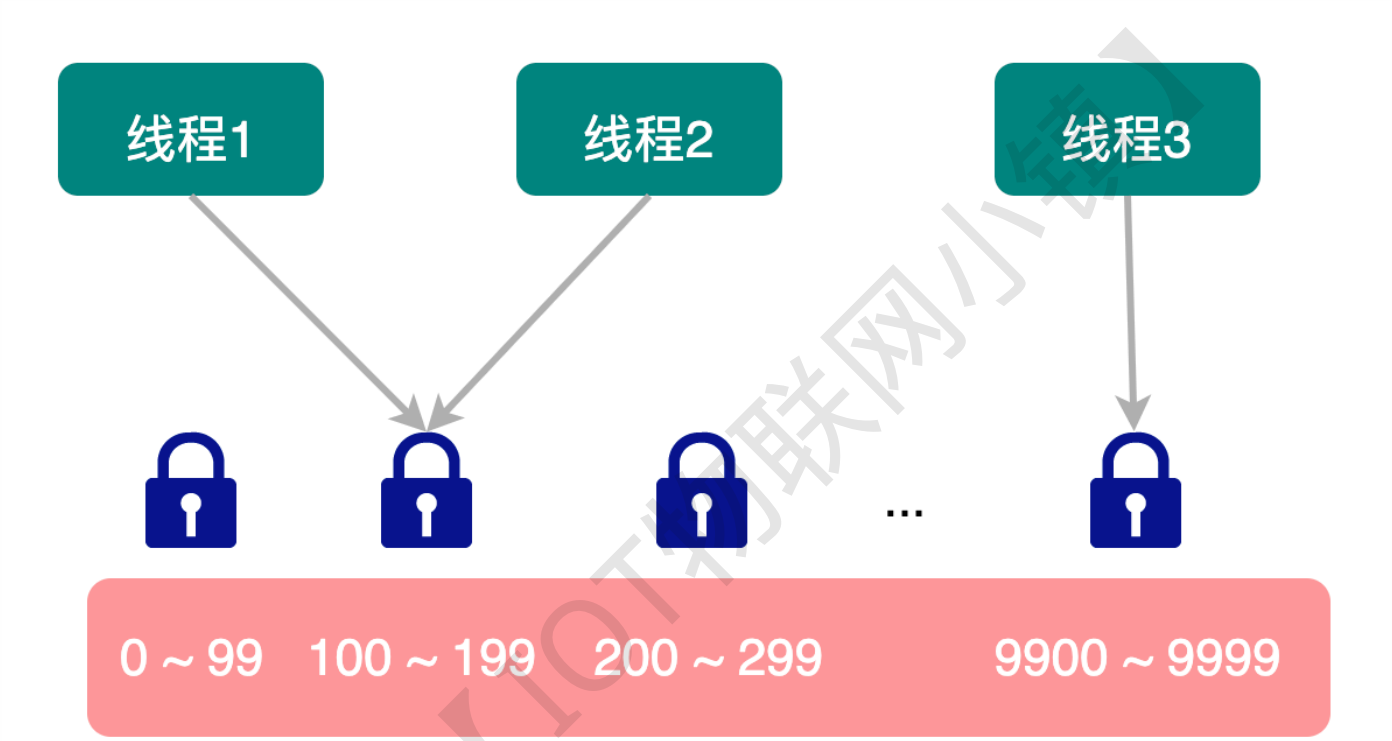
线程3想访问test_data[9900]。

首先根据每个线程要访问的数据索引进行计算：这个索引对应的哪一把锁？

> 计算方式：访问索引 % 每把锁对应的数据长度

经过计算得知：线程1、线程2就会对第二把锁进行竞争；

而线程3就可以独自获取最后一把锁，这样的话线程3就避开了与线程1、线程2的竞争。



测试结果

```
$ ./a.out
test1_naked:        average = 2876 ms
test2_one_big_lock: average = 11233 ms
test3_segment_lock: average = 3216 ms
```

从测试结果上看，分段加锁比使用一把全局锁，对于程序性能的提高确实很明显。

当然了，测试结果与不同的系统环境、业务场景有关，特别是线程的竞争程度、在临界区中的执行时间等。

测试代码简介

这里贴一下代码的结构，文末有完整的代码下载链接。

测试代码没有考虑跨边界的情况。

比如：某个线程需要访问190 ~ 210这些索引上的数据，这个区间正好跨越了200这个分界点。

> 第0把锁：0 ~ 99;
>
> 第1把锁：100 ~ 199;
>
> 第2把锁：200 ~ 299;

因此，访问190 ~ 210就需要同时获取到第1、2把锁。

在实际项目中需要考虑到这种跨边界的情况，通过计算开始和结束索引，把这些锁都获取到才可以。

当然了，为了防止发生死锁，需要按照顺序来获取。

```
#define THREAD_NUMBER           100         // 线程个数
#define LOOP_TIMES_EACH_THREAD  100000      // 每个线程中 for 循环的执行次数
#define DATA_TOTAL_NUM          10000       // 全局变量的长度
#define SEGMENT_LEN             100         // 多少个数据分配一把锁

volatile int test_data[DATA_TOTAL_NUM];     // 被竞争的全局变量

void main(void)
{
    test1_naked();
    test2_one_big_lock();
    test3_segment_lock();

    while (1)
        sleep(3);  // 主线程保持运行，也可以使用getchar();
}

// 测试1：子线程执行的函数
void *test1_naked_function(void *arg)
{
    struct timeval tm1, tm2;
    gettimeofday(&tm1, NULL);
    for (unsigned int i = 0; i < LOOP_TIMES_EACH_THREAD; i++)
    {
        do_some_work();             // 模拟业务操作
        unsigned int pos = rand() % DATA_TOTAL_NUM;
        test_data[pos] = i * i;     // 随机访问全局变量中的某个数据
    }
    gettimeofday(&tm2, NULL);

    return (tm2 - tm1);
}

// 测试2：子线程执行的函数
void *test2_one_big_lock_function(void *arg)
{
```

```
    test2_one_big_lock_arg *data = (test2_one_big_lock_arg *)arg;
    struct timeval tm1, tm2;
    gettimeofday(&tm1, NULL);
    for (unsigned int i = 0; i < LOOP_TIMES_EACH_THREAD; i++)
    {
        pthread_mutex_lock(&data->lock);  // 上锁

        do_some_work();             // 模拟业务操作
        unsigned int pos = rand() % DATA_TOTAL_NUM;
        test_data[pos] = i * i;     // 随机访问全局变量中的某个数据

        pthread_mutex_unlock(&data->lock); // 解锁
    }
    gettimeofday(&tm2, NULL);

    return (tm2 - tm1);
}

// 测试3：子线程执行的函数
void *test3_segment_lock_function(void *arg)
{
    test3_segment_lock_arg *data = (test3_segment_lock_arg *)arg;
    struct timeval tm1, tm2;
    gettimeofday(&tm1, NULL);
    for (unsigned int i = 0; i < LOOP_TIMES_EACH_THREAD; i++)
    {
        unsigned int pos = rand() % DATA_TOTAL_NUM;      // 产生随机访问的索引
        unsigned int lock_index = pos / SEGMENT_LEN;     // 根据索引计算需要获取哪一把锁

        pthread_mutex_lock(data->lock + lock_index);     // 上锁

        do_some_work();             // 模拟业务操作
        test_data[pos] = i * i;     // 随机访问全局变量中的某个数据

        pthread_mutex_unlock(data->lock + lock_index);   // 解锁
    }
    gettimeofday(&tm2, NULL);

    return (tm2 - tm1);
}

void test1_naked()
{
    创建 100 个线程，线程执行函数是 test1_naked_function()
    printf("test1_naked:        average = %ld ms \n", ms_total / THREAD_NUMBER);
}

void test2_one_big_lock()
{
    创建 100 个线程，线程执行函数是 test2_one_big_lock_function()，需要把锁作为参数传递给子线
程。
    printf("test2_one_big_lock: average = %ld ms \n", ms_total / THREAD_NUMBER);
```

```
}

void test3_segment_lock()
{
    根据全局变量的长度，初始化很多把锁。
    创建 100 个线程，线程执行函数是 test2_one_big_lock_function()，需要把锁作为参数传递给子线
程。

    printf("test3_segment_lock: average = %ld ms \n", ms_total / THREAD_NUMBER);
}
```

------ End ------

如果文中有什么问题，欢迎留言、讨论，谢谢！

在公众号后台恢复：220417，可以收到示例代码。

在Linux系统中可以直接编译、执行，拿来即用。

祝您好运！

推荐阅读

【1】《Linux 从头学》系列文章

【2】C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻

【3】原来gdb的底层调试原理这么简单

【4】Linux中对【库函数】的调用进行跟踪的3种【插桩】技巧

【5】内联汇编很可怕吗？看完这篇文章，终结它！

【6】gcc编译时，链接器安排的【虚拟地址】是如何计算出来的？

【7】GCC 链接过程中的【重定位】过程分析

【8】Linux 动态链接过程中的【重定位】底层原理

其他系列专辑：精选文章、应用程序设计、物联网、 C语言。