作者：代兴　邮箱：503268771@qq.com　Vx 公众号：东方架构师
https://github.com/drink-cat/Book_Program-Principles

内存屏障指令

内存屏障，使用 lfence、sfence、mfence、lock 指令。
lfence 指令，表示读屏障。
sfence 指令，表示写屏障。
mfence 指令，表示读写屏障，包含 lfence、sfence 的功能。
lock 指令，表示锁高速缓存行，有内存屏障的作用。

在代码的合适位置插入内存屏障指令：
```
asm volatile("mfence" ::: "memory");
asm volatile("lock ; addq $0, (%%rsp) ; " ::: "memory");
```
比如，如下的函数使用了 lock 指令。
```
void thread_func2()
{
    bb = 3;
    asm volatile("lock ; addq $0, (%%rsp) ; " ::: "memory"); // 内存屏障指令
    read_aa = aa;
}
```

用 C 和汇编分析运行期指令乱序

编写代码：　cpu_reorder.c
```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <pthread.h>

// 公共变量。让 2 个线程同时操作，形成相互依赖。
int aa = 0;
int bb = 0;
int read_aa = 55;
int read_bb = 55;

// thread_func1 thread_func2 对变量的操作形成死锁结构。
// 如果没有乱序，就不可能出现 read_aa==0 && read_bb==0

// 操作公共变量
void thread_func1()
{
    aa = 3;
```

```
    asm volatile("" ::: "memory"); // 防止编译乱序
    read_bb = bb;
}

// 操作公共变量
void thread_func2()
{
    bb = 3;
    asm volatile("" ::: "memory"); // 防止编译乱序
    read_aa = aa;
}

// 初始化。
void init()
{
    aa = 0;
    bb = 0;
    read_aa = 55;
    read_bb = 55;
}

// 控制 2 个子线程。
volatile bool thread_stop = false; // 子线程，停止
volatile bool thread_wait1 = true; // 第一个子线程，等待
volatile bool thread_wait2 = true; // 第二个子线程，等待

void *thread_loop1(void *arg)
{
    while (!thread_stop)
    {
        // 为 1，等待。为 0，跳过。
        while (thread_wait1)
        {
        }

        // 操作公共变量
        thread_func1();

        // 等待下一次同步操作
        thread_wait1 = true;
    }
    printf("线程 1，退出。\n");
}

void *thread_loop2(void *arg)
{
    while (!thread_stop)
    {
        // 为 1，等待。为 0，跳过。
```

```
        while (thread_wait2)
        {
        }

        // 操作公共变量
        thread_func2();

        // 等待下一次同步操作
        thread_wait2 = true;
    }
    printf("线程 2，退出。\n");
}

int main()
{
    printf("主线程，开始。\n");
    // 创建 2 个线程，操作公共变量。
    pthread_t t1;
    pthread_t t2;
    pthread_create(&t1, NULL, thread_loop1, NULL);
    pthread_create(&t2, NULL, thread_loop2, NULL);

    // 循环尝试。
    int count = 0;
    while (true)
    {
        ++count;

        // 初始化公共变量。
        init();
        printf("主线程，让子线程执行。  count=%d \n", count);

        // 让 2 个子线程，执行操作公共变量。
        thread_wait1 = false;
        thread_wait2 = false;

        // 等待 2 个子线程，都操作一次。
        while (!thread_wait1)
        {
        }
        while (!thread_wait2)
        {
        }

        // 判断是否发生指令乱序。
        if (read_aa == 0 && read_bb == 0)
        {
            printf("遇到情况：(read_aa == 0 && read_bb == 0)  count=%d \n", count);
            break;
```

```
        }
        if (count % 1000 == 0)
        {
            printf("循环次数=%d \n", count);
        }
    }

    // 让子线程退出。
    thread_stop = true;
    thread_wait1 = false;
    thread_wait2 = false;

    sleep(1);
    printf("主线程，退出。\n");
    return 0;
}
```

编译代码：

```
gcc cpu_reorder.c -lpthread -o cpu_reorder
objdump -D cpu_reorder > cpu_reorder.dump.txt
```

运行代码：

```
[root@local barrier]# ./cpu_reorder
主线程，开始。
主线程，让子线程执行。  count=1
主线程，让子线程执行。  count=2
主线程，让子线程执行。  count=3
遇到情况：(read_aa == 0 && read_bb == 0)  count=3
线程 2，退出。
线程 1，退出。
主线程，退出。

[root@local barrier]# ./cpu_reorder
主线程，开始。
主线程，让子线程执行。  count=1
主线程，让子线程执行。  count=2
主线程，让子线程执行。  count=3
主线程，让子线程执行。  count=4
主线程，让子线程执行。  count=5
遇到情况：(read_aa == 0 && read_bb == 0)  count=5
线程 1，退出。
线程 2，退出。
主线程，退出。
```

代码逻辑：

使用 2 个线程，相互依赖对方的变量操作，形成死锁结构。第 1 个线程，写 aa，读 bb。第 2 个线程，写 bb，读 aa。

使用 `asm volatile("" ::: "memory");`，禁止编译期指令乱序。

使用标志位 thread_wait1、thread_wait2，让 2 个线程同时操作变量。第 1 个线程，调用 thread_func1()。第 2 个线程，调用 thread_func2()。

主线程，使用大循环控制整体流程，首先初始化变量，然后修改标志位，进而多次触发 2 个线程并发操作。
判断是否发生指令乱序 `if (read_aa == 0 && read_bb == 0)` 。如果发生指令乱序，就退出大循环，否则继续大循环。

分析结果：
第 1 次运行，在第 3 次循环时，发生运行期指令乱序。 `遇到情况：(read_aa == 0 && read_bb == 0)  count=3` 。
第 2 次运行，在第 5 次循环时，发生运行期指令乱序。 `遇到情况：(read_aa == 0 && read_bb == 0)  count=5` 。
死锁结构可以保证，发生运行期指令乱序，才会出现 `(read_aa == 0 && read_bb == 0)` 。
不能保证每次并发都出现运行期指令乱序，所以使用循环法多次触发线程并发操作。

查看 cpu_reorder.dump.txt，找到函数 thread_func1()的汇编代码，没有内存屏障指令。

```
000000000040063d <thread_func1>:
  40063d:   55                      push   %rbp
  40063e:   48 89 e5                mov    %rsp,%rbp
  400641:   c7 05 11 0a 20 00 03    movl   $0x3,0x200a11(%rip)        # 60105c <aa>
  400648:   00 00 00
  40064b:   8b 05 0f 0a 20 00       mov    0x200a0f(%rip),%eax        # 601060 <bb>
  400651:   89 05 f9 09 20 00       mov    %eax,0x2009f9(%rip)        # 601050 <read_bb>
  400657:   5d                      pop    %rbp
  400658:   c3                      retq
```

用内存屏障指令解决运行期指令乱序

编写代码：fix_cpu_reorder.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <pthread.h>

// 公共变量。让 2 个线程同时操作，形成相互依赖。
int aa = 0;
int bb = 0;
int read_aa = 55;
int read_bb = 55;

// 操作公共变量
void thread_func1()
{
    aa = 3;
    asm volatile("mfence" ::: "memory"); // 内存屏障指令
    read_bb = bb;
}

// 操作公共变量
void thread_func2()
```

```
{
    bb = 3;
    asm volatile("mfence" ::: "memory"); // 内存屏障指令
    read_aa = aa;
}

// 初始化。
void init()
{
    aa = 0;
    bb = 0;
    read_aa = 55;
    read_bb = 55;
}

// 控制 2 个子线程。
volatile bool thread_stop = false; // 子线程，停止
volatile bool thread_wait1 = true; // 第一个子线程，等待
volatile bool thread_wait2 = true; // 第二个子线程，等待

void *thread_loop1(void *arg)
{
    while (!thread_stop)
    {
        // 为 1，等待。为 0，跳过。
        while (thread_wait1)
        {
        }

        // 操作公共变量
        thread_func1();

        // 等待下一次同步操作
        thread_wait1 = true;
    }
    printf("线程 1，退出。\n");
}

void *thread_loop2(void *arg)
{
    while (!thread_stop)
    {
        // 为 1，等待。为 0，跳过。
        while (thread_wait2)
        {
        }

        // 操作公共变量
        thread_func2();
```

```
        // 等待下一次同步操作
        thread_wait2 = true;
    }
    printf("线程 2，退出。\n");
}

int main()
{
    printf("主线程，开始。\n");
    // 创建 2 个线程，操作公共变量。
    pthread_t t1;
    pthread_t t2;
    pthread_create(&t1, NULL, thread_loop1, NULL);
    pthread_create(&t2, NULL, thread_loop2, NULL);

    // 循环尝试。
    int count = 0;
    while (true)
    {
        ++count;

        // 初始化公共变量。
        init();
        // printf("主线程，让子线程执行。  count=%d \n", count);

        // 让 2 个子线程，执行操作公共变量。
        thread_wait1 = false;
        thread_wait2 = false;

        // 等待 2 个子线程，都操作一次。
        while (!thread_wait1)
        {
        }
        while (!thread_wait2)
        {
        }

        // 判断是否发生指令乱序。
        if (read_aa == 0 && read_bb == 0)
        {
            printf("遇到情况：(read_aa == 0 && read_bb == 0)  count=%d \n", count);
            break;
        }
        if (count % 1000000 == 0)
        {
            printf("循环次数=%d \n", count);
        }
    }
```

```
    // 让子线程退出。
    thread_stop = true;
    thread_wait1 = false;
    thread_wait2 = false;

    sleep(1);
    printf("主线程，退出。\n");
    return 0;
}
```

编译代码：

```
gcc fix_cpu_reorder.c -lpthread -o fix_cpu_reorder
objdump -D fix_cpu_reorder > fix_cpu_reorder.dump.txt
```

运行代码：

```
[root@local barrier]# ./fix_cpu_reorder
主线程，开始。
循环次数=1000000
循环次数=2000000
循环次数=3000000
循环次数=4000000
循环次数=5000000
循环次数=6000000
循环次数=7000000
循环次数=8000000
循环次数=9000000
循环次数=10000000
循环次数=11000000
循环次数=12000000
```

代码逻辑：
使用前一节的代码结构，增加内存屏障指令，对比运行结果。
使用 `asm volatile("mfence" ::: "memory")` ，禁止编译期指令乱序、运行期指令乱序。
使用 `if (count % 1000000 == 0)` ，查看循环次数。

分析结果：
循环一直运行，没有满足条件 `if (read_aa == 0 && read_bb == 0)` ，没有发生运行期指令乱序。
`asm volatile("mfence" ::: "memory")` 达到预期作用。

查看 fix_cpu_reorder.dump.txt，找到函数 thread_func1()的汇编代码，包含 mfence 指令，使得编译期内存屏障生效。

```
0000000000040063d <thread_func1>:
  40063d:   55                      push   %rbp
  40063e:   48 89 e5                mov    %rsp,%rbp
  400641:   c7 05 11 0a 20 00 03    movl   $0x3,0x200a11(%rip)        # 60105c <aa>
  400648:   00 00 00
  40064b:   0f ae f0                mfence
  40064e:   8b 05 0c 0a 20 00       mov    0x200a0c(%rip),%eax        # 601060 <bb>
```

```
400654:   89 05 f6 09 20 00       mov    %eax,0x2009f6(%rip)        # 601050 <read_bb>
40065a:   5d                      pop    %rbp
40065b:   c3                      retq
```