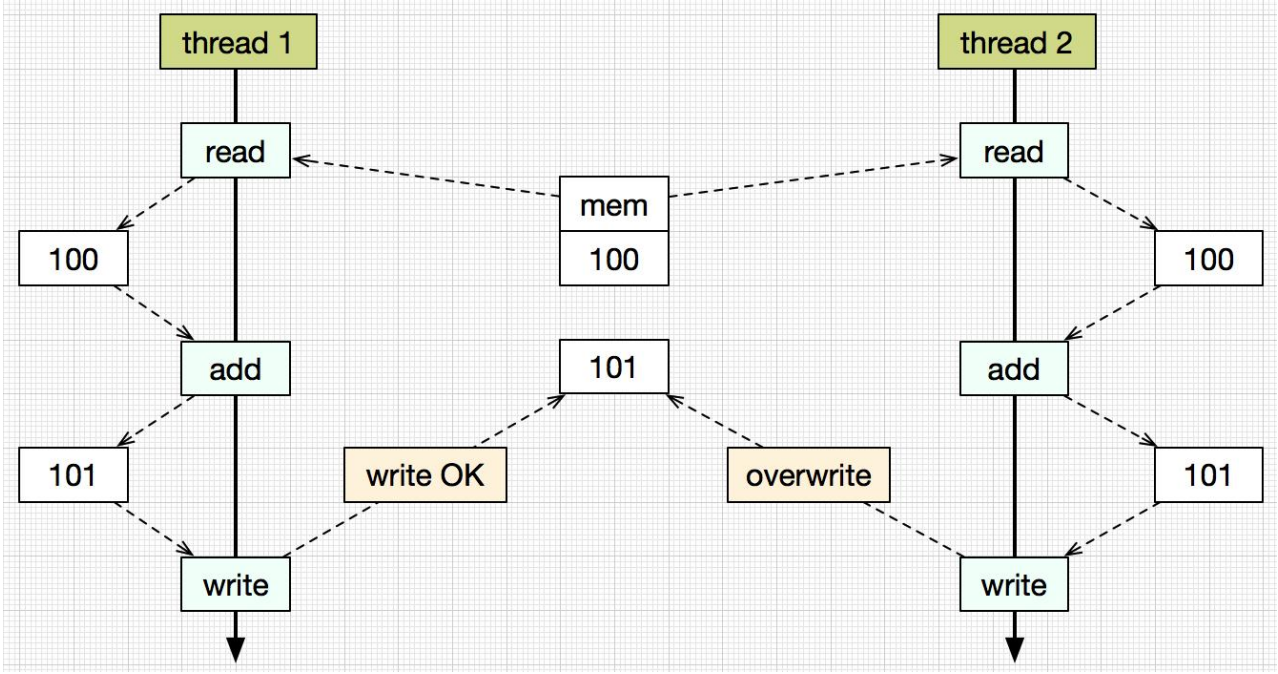
作者：代兴　邮箱：503268771@qq.com　Vx 公众号：东方架构师
https://github.com/drink-cat/Book_Program-Principles

并发写需要 CAS 指令保证原子性

并发写可能导致写覆盖、更新丢失，并发写需要 CAS 指令保证原子性。
lock 指令和 cmpxchg 指令配合使用，lock 指令表示锁住高速缓存行，cmpxchg 指令表示比较并交换。

写覆盖的示意图：



用 C 分析多线程累加 1 个整数

编写代码： multi_cas.c

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

// 线程安全的 CAS
bool func_cas(int *param_addr, int old_value, int new_value)
{
    int ret_eax = 0;                    // 记录返回的 eax
    asm volatile(                       // 内联汇编
        "lock \n"                       // 加锁。锁定缓存行
        "cmpxchgl %%ebx, (%%rcx) \n" // 执行 cas
        : "=a"(ret_eax)                 // 返回值，eax，写到 ret_eax
```

```
        : "a"(old_value),           // 旧值，写到 rax
          "b"(new_value),           // 新值，写到 rcx
          "c"(param_addr));         // 地址，写到 rdx

    if (ret_eax == old_value) // cas 成功
    {
        return true;
    }
    else // cas 失败
    {
        return false;
    }
}

// 线程安全的 INC
void func_inc_int(int *addr)
{
    while (true) // 循环，直到操作成功
    {
        int old_v = *addr;                          // 旧值
        int new_v = old_v + 1;                      // 新值
        bool cas_ok = func_cas(addr, old_v, new_v); // 执行 cas

        if (cas_ok) // cas 成功，就退出
        {
            break;
        }
        else // cas 失败，就继续
        {
            continue;
        }
    }
}

// 变量。累加。
int count = 0;

// 是否使用 cas
int use_cas = 0;

// 线程的函数
void *thread_func(void *param)
{
    // 执行很多次
    for (int k = 1; k <= 10000; ++k)
    {
        if (use_cas) // 使用 cas
        {
            func_inc_int(&count);
```

```
        }
        else // 不使用 cas
        {
            ++count; // 直接累加
        }
    }
}

int main()
{
    printf("Please choose use CAS or not : 0 = NoCAS  1 = YesCAS \n");
    scanf("%d", &use_cas); // 开关。是否使用 cas

    int before = count; // 之前的值

    // 创建多个线程，并发执行
    pthread_t p1;
    pthread_create(&p1, NULL, thread_func, NULL);
    pthread_t p2;
    pthread_create(&p2, NULL, thread_func, NULL);
    pthread_t p3;
    pthread_create(&p3, NULL, thread_func, NULL);
    pthread_t p4;
    pthread_create(&p4, NULL, thread_func, NULL);
    pthread_t p5;
    pthread_create(&p5, NULL, thread_func, NULL);

    sleep(2); // 等待线程完成。

    int after = count; // 之后的值
    printf("count  before = %d  after = %d \n\n", before, after);

    return 0;
}
```

编译代码：

```
gcc multi_cas.c -lpthread -std=gnu99 -o multi_cas
```

运行代码：

```
[root@local cas]# ./multi_cas
Please choose use CAS or not : 0 = NoCAS  1 = YesCAS
0
count  before = 0  after = 48224

[root@local cas]# ./multi_cas
Please choose use CAS or not : 0 = NoCAS  1 = YesCAS
0
count  before = 0  after = 47135
```

```
[root@local cas]# ./multi_cas
Please choose use CAS or not : 0 = NoCAS  1 = YesCAS
1
count  before = 0  after = 50000
```

分析结果：
第一次运行。输入 0，输出 48224。没有使用 CAS 指令。
第二次运行。输入 0，输出 47135。没有使用 CAS 指令。
第三次运行。输入 1，输出 50000。使用 CAS 指令。
没有使用 CAS 指令，多个线程写同一个变量，变量的值可能被覆盖，丢失部分更新。
使用 CAS 指令，写变量具有原子性，多个线程的写操作互斥，累加全部的整数。

linux 源码中的 CAS 指令

文件：
linux-5.6.3/arch/x86/include/asm/cmpxchg.h
linux-5.6.3/arch/x86/include/asm/cmpxchg_64.h
linux-5.6.3/tools/arch/x86/include/asm/atomic.h

代码：

```
/*
 * Atomic compare and exchange.  Compare OLD with MEM, if identical,
 * store NEW in MEM.  Return the initial value in MEM.  Success is
 * indicated by comparing RETURN with OLD.
 */
#define __raw_cmpxchg(ptr, old, new, size, lock)           \
({                                  \
    __typeof__(*(ptr)) __ret;                   \
    __typeof__(*(ptr)) __old = (old);               \
    __typeof__(*(ptr)) __new = (new);               \
    switch (size) {                         \
    case __X86_CASE_B:                      \
    {                                   \
        volatile u8 *__ptr = (volatile u8 *)(ptr);      \
        asm volatile(lock "cmpxchgb %2,%1"          \
                : "=a" (__ret), "+m" (*__ptr)       \
                : "q" (__new), "0" (__old)          \
                : "memory");                \
        break;                              \
    }                                   \
    case __X86_CASE_W:                      \
    {                                   \
        volatile u16 *__ptr = (volatile u16 *)(ptr);        \
        asm volatile(lock "cmpxchgw %2,%1"          \
                : "=a" (__ret), "+m" (*__ptr)       \
                : "r" (__new), "0" (__old)          \
```

```
                : "memory");                \
        break;                              \
    }                                       \
    case __X86_CASE_L:                      \
    {                                       \
        volatile u32 *__ptr = (volatile u32 *)(ptr);        \
        asm volatile(lock "cmpxchgl %2,%1"          \
                : "=a" (__ret), "+m" (*__ptr)       \
                : "r" (__new), "0" (__old)          \
                : "memory");                \
        break;                              \
    }                                       \
    case __X86_CASE_Q:                      \
    {                                       \
        volatile u64 *__ptr = (volatile u64 *)(ptr);        \
        asm volatile(lock "cmpxchgq %2,%1"          \
                : "=a" (__ret), "+m" (*__ptr)       \
                : "r" (__new), "0" (__old)          \
                : "memory");                \
        break;                              \
    }                                       \
    default:                                \
        __cmpxchg_wrong_size();                 \
    }                                       \
    __ret;                                  \
})

#define __cmpxchg(ptr, old, new, size)                  \
    __raw_cmpxchg((ptr), (old), (new), (size), LOCK_PREFIX)

#define __sync_cmpxchg(ptr, old, new, size)             \
    __raw_cmpxchg((ptr), (old), (new), (size), "lock; ")

#define __cmpxchg_local(ptr, old, new, size)                \
    __raw_cmpxchg((ptr), (old), (new), (size), "")

#define LOCK_PREFIX "\n\tlock; "

static __always_inline int atomic_cmpxchg(atomic_t *v, int old, int new)
{
    return cmpxchg(&v->counter, old, new);
}
```

cmpxchg 指令区分 8 位、16 位、32 位、64 位，每种情况的代码结构相似。
并发场景，使用 lock 指令，`__raw_cmpxchg((ptr), (old), (new), (size), LOCK_PREFIX)`。
非并发场景，不使用 lock 指令，`__raw_cmpxchg((ptr), (old), (new), (size), "")`。