WRITE_ONCE READ_ONCE 函数的介绍与使用

原创 Wakke Wang 2020-06-23 14:49:04 208 收藏 版权

分类专栏： Linux linux driver

今天看 内核中链表中的代码 include/linux/list.h， 发现其中有很多代码用到了 WRITE_ONCE ,就引发了我的思考

```
#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
    struct list_head name = LIST_HEAD_INIT(name)

static inline void INIT_LIST_HEAD(struct list_head *list)
{
    WRITE_ONCE(list->next, list);
    list->prev = list;
}
```


上面的代码是初始化一个双向循环链表， 将list中的两个指针 next 和 prev 都指向 自己，也就是 list，那为什么不直接赋值呢？笔者就查了查以前版本的内核代码，发现 linux4.5 以下的版本都是直接赋值的，linux4.5以上的版本都进行了优化。

```
static inline void INIT_LIST_HEAD(struct list_head *list)
{
    list->next = list;
    list->prev = list;
}
```

那我们进行思考以下两个问题：

1、内核出于什么原因进行优化呢？ 它和直接赋值有什么区别？

2、我们什么时候要使用 WRITE_ONCE?

来，先看看它的定义

```
static __always_inline void __write_once_size(volatile void *p, void *res, int size)
{
    switch (size) {
    case 1: *(volatile __u8 *)p = *(__u8 *)res; break;
    case 2: *(volatile __u16 *)p = *(__u16 *)res; break;
    case 4: *(volatile __u32 *)p = *(__u32 *)res; break;
    case 8: *(volatile __u64 *)p = *(__u64 *)res; break;
    default:
        barrier();
        __builtin_memcpy((void *)p, (const void *)res, size);
        barrier();
    }
}

#define WRITE_ONCE(x, val) \
({                          \
    union { typeof(x) __val; char __c[1]; } __u =   \
        { .__val = (__force typeof(x)) (val) }; \
    __write_once_size(&(x), __u.__c, sizeof(x));    \
    __u.__val;                  \
})
```

定义一个union 结构体,这里以 int *为例

```
union{
    int *           int * 类型转换
  tpyeof(x) __val = ( tpyeof(x) ) x;
  char      __c[1]; // 临时变量 __val 地址
}__u
```

最后一行 __u.__val; 没有任何意义

为什么要用READ_ONCE()和WRITE_ONCE()这两个宏呢？ 这里起到关键作用的就是 **volatile** ， 它主要告诉编译器：

1、声明这个变量很重要，不要把它当成一个普通的变量，做出错误的优化。

2、保证 CPU 每次都从内存重新读取变量的值，而不是用寄存器中暂存的值。

因为在 **多线程/多核** 环境中，不会被当前线程修改的变量，可能会被其他的线程修改，从内存读才可靠。

还有一部分原因是，这两个宏可以作为标记，提醒编程人员这里面是一个多核/多线程共享的变量，必要的时候应该加互斥锁来保护。

搞明白了之后，开头提到的两个问题是不是就有了答案呢？

总结一下：

在多核多线程编程时，要注意共享变量的使用，要保证是 volatile的