

提到自旋锁那就必须要说链表，在上一篇《驱动开发：内核中的链表与结构体》文章中简单实用链表结构来存储进程信息列表，相信读者应该已经理解了内核链表的基本使用，本篇文章将讲解自旋锁的简单应用，自旋锁是为了解决内核链表读写时存在线程同步问题，解决多线程同步问题必须要用锁，通常使用自旋锁，自旋锁是内核中提供了一种高IRQL锁，用同步以及独占的方式访问某个资源。

首先以简单的链表为案例，链表主要分为单向链表与双向链表，单向链表的链表节点中只有一个链表指针，其指向后一个链表元素，而双向链表节点中有两个链表节点指针，其中Blink指向前一个链表节点Flink指向后一个节点，以双向链表为例。

```
#include <ntifs.h>
#include <ntstrsafe.h>

/*
// 链表节点指针
typedef struct _LIST_ENTRY
{
    struct _LIST_ENTRY *Flink;    // 当前节点的后一个节点
    struct _LIST_ENTRY *Blink;    // 当前节点的前一个节点
}LIST_ENTRY, *PLIST_ENTRY;
*/

typedef struct _MyStruct
{
    ULONG x;
    ULONG y;
    LIST_ENTRY lpListEntry;
}MyStruct, *pMyStruct;

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("驱动卸载成功 \n");
}

// By: LyShark
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("By:LyShark \n");
    DbgPrint("Email:me@lyshark.com \n");
    // 初始化头节点
    LIST_ENTRY ListHeader = { 0 };
    InitializeListHead(&ListHeader);

    // 定义链表元素
    MyStruct testA = { 0 };
    MyStruct testB = { 0 };
    MyStruct testC = { 0 };

    testA.x = 100;
    testA.y = 200;

    testB.x = 1000;
    testB.y = 2000;
```

```

testC.x = 10000;
testC.y = 20000;

// 分别插入节点到头部和尾部
InsertHeadList(&ListHeader, &testA.lplListEntry);
InsertTailList(&ListHeader, &testB.lplListEntry);
InsertTailList(&ListHeader, &testC.lplListEntry);

// 节点不为空 则 移除一个节点
if (IsListEmpty(&ListHeader) == FALSE)
{
    RemoveEntryList(&testA.lplListEntry);
}

// 输出链表数据
PLIST_ENTRY pListEntry = NULL;
pListEntry = ListHeader.Flink;

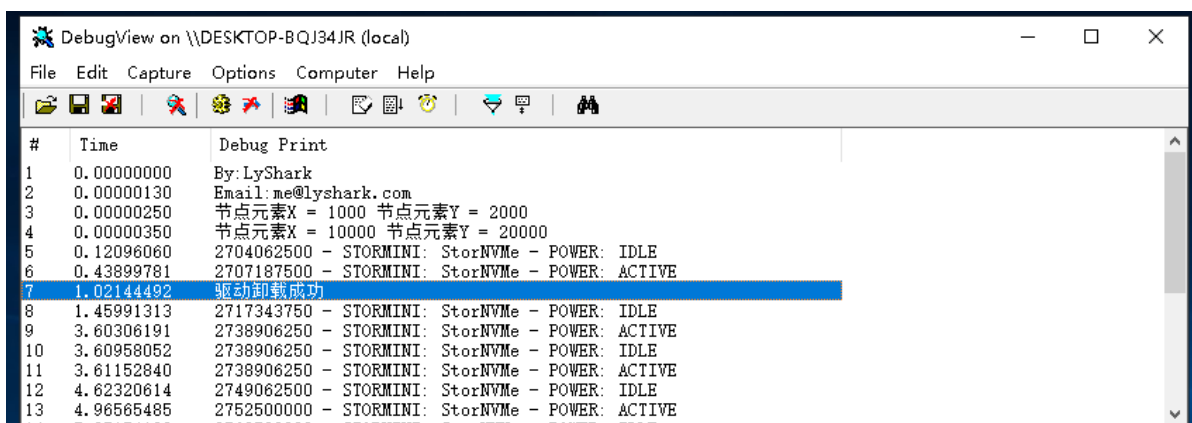
while (pListEntry != &ListHeader)
{
    // 计算出成员距离结构体顶部内存距离
    pMyStruct ptr = CONTAINING_RECORD(pListEntry, MyStruct, lplListEntry);
    DbgPrint("节点元素X = %d 节点元素Y = %d \n", ptr->x, ptr->y);

    // 得到下一个元素地址
    pListEntry = pListEntry->Flink;
}

Driver->DriverUnload = UnDriver;
return STATUS_SUCCESS;
}

```

链表输出效果如下：



#	Time	Debug Print
1	0.00000000	By: LyShark
2	0.00000130	Email: me@lyshark.com
3	0.00000250	节点元素X = 1000 节点元素Y = 2000
4	0.00000350	节点元素X = 10000 节点元素Y = 20000
5	0.12096060	2704062500 - STORMINI: StorNVMe - POWER: IDLE
6	0.43899781	2707187500 - STORMINI: StorNVMe - POWER: ACTIVE
7	1.02144492	驱动卸载成功
8	1.45991313	2717343750 - STORMINI: StorNVMe - POWER: IDLE
9	3.60306191	2738906250 - STORMINI: StorNVMe - POWER: ACTIVE
10	3.60958052	2738906250 - STORMINI: StorNVMe - POWER: IDLE
11	3.61152840	2738906250 - STORMINI: StorNVMe - POWER: ACTIVE
12	4.62320614	2749062500 - STORMINI: StorNVMe - POWER: IDLE
13	4.96565485	2752500000 - STORMINI: StorNVMe - POWER: ACTIVE

如上所述，内核链表读写时存在线程同步问题，解决多线程同步问题必须要用锁，通常使用自旋锁，自旋锁是内核中提供了一种高IRQL锁，用同步以及独占的方式访问某个资源。

```

#include <ntifs.h>
#include <ntstrsafe.h>

/*
// 链表节点指针

```

```

typedef struct _LIST_ENTRY
{
    struct _LIST_ENTRY *Flink;    // 当前节点的后一个节点
    struct _LIST_ENTRY *Blink;    // 当前节点的前一个结点
}LIST_ENTRY, *PLIST_ENTRY;
*/

typedef struct _MyStruct
{
    ULONG x;
    ULONG y;
    LIST_ENTRY lpListEntry;
}MyStruct, *pMyStruct;

// 定义全局链表和全局锁
LIST_ENTRY my_list_header;
KSPIN_LOCK my_list_lock;

// 初始化
void Init()
{
    InitializeListHead(&my_list_header);
    KeInitializeSpinLock(&my_list_lock);
}

// 函数内使用锁
void function_ins()
{
    KIRQL Irql;

    // 加锁
    KeAcquiresSpinLock(&my_list_lock, &Irql);

    DbgPrint("锁内部执行 \n");

    // 释放锁
    KeReleaseSpinLock(&my_list_lock, Irql);
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("驱动卸载成功 \n");
}

// By: LyShark
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("By:LyShark \n");
    DbgPrint("Email:me@lyshark.com \n");

    // 初始化链表
    Init();

    // 分配链表空间

```

```

    pMyStruct testA = (pMyStruct)ExAllocatePool(NonPagedPoolExecute,
sizeof(pMyStruct));
    pMyStruct testB = (pMyStruct)ExAllocatePool(NonPagedPoolExecute,
sizeof(pMyStruct));

    // 赋值
    testA->x = 100;
    testA->y = 200;

    testB->x = 1000;
    testB->y = 2000;

    // 向全局链表中插入数据
    if (NULL != testA && NULL != testB)
    {
        ExInterlockedInsertHeadList(&my_list_header, (PLIST_ENTRY)&testA-
>lpListEntry, &my_list_lock);
        ExInterlockedInsertTailList(&my_list_header, (PLIST_ENTRY)&testB-
>lpListEntry, &my_list_lock);
    }

    function_ins();

    // 移除节点A并放入到remove_entry中
    PLIST_ENTRY remove_entry = ExInterlockedRemoveHeadList(&testA->lpListEntry,
&my_list_lock);

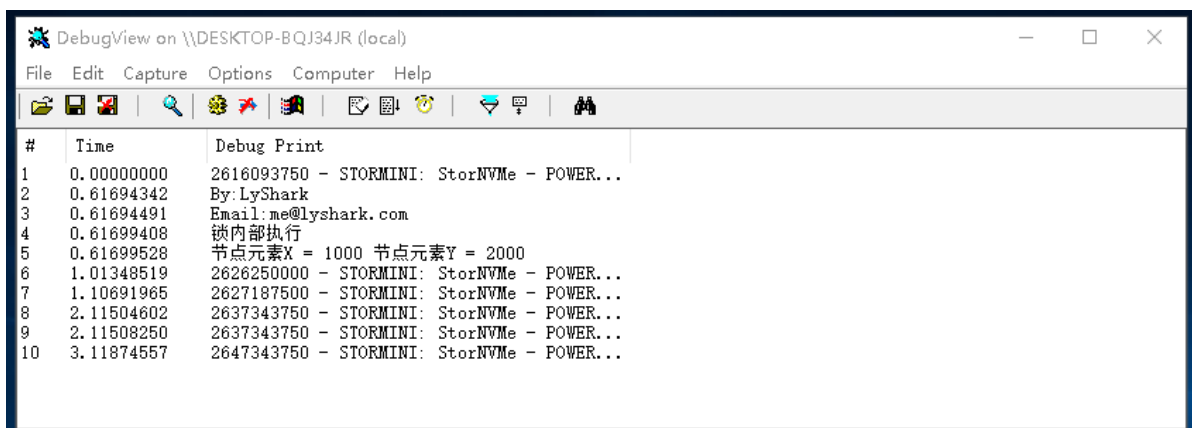
    // 输出链表数据
    while (remove_entry != &my_list_header)
    {
        // 计算出成员距离结构体顶部内存距离
        pMyStruct ptr = CONTAINING_RECORD(remove_entry, MyStruct, lpListEntry);
        DbgPrint("节点元素X = %d 节点元素Y = %d \n", ptr->x, ptr->y);

        // 得到下一个元素地址
        remove_entry = remove_entry->Flink;
    }

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}

```

加锁后执行效果如下：



#	Time	Debug Print
1	0.00000000	2616093750 - STORMINI: StorNVMe - POWER...
2	0.61694342	By: LyShark
3	0.61694491	Email: me@lyshark.com
4	0.61699408	锁内部执行
5	0.61699528	节点元素X = 1000 节点元素Y = 2000
6	1.01348519	2626250000 - STORMINI: StorNVMe - POWER...
7	1.10691965	2627187500 - STORMINI: StorNVMe - POWER...
8	2.11504602	2637343750 - STORMINI: StorNVMe - POWER...
9	2.11508250	2637343750 - STORMINI: StorNVMe - POWER...
10	3.11874557	2647343750 - STORMINI: StorNVMe - POWER...