# 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的一种高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; // 当前节点的前一个结点

}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](mailto: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.lpListEntry); InsertTailList(&ListHeader, &testB.lpListEntry); InsertTailList(&ListHeader, &testC.lpListEntry);

// 节点不为空 则 移除一个节点

if (IsListEmpty(&ListHeader) == FALSE)

{

RemoveEntryList(&testA.lpListEntry);

}

// 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, 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

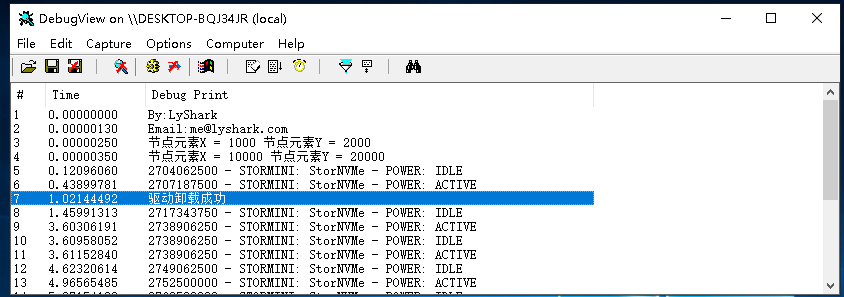
pListEntry = pListEntry->Flink;

}

Driver->DriverUnload = UnDriver; return STATUS\_SUCCESS;

}

# 链表输出效果如下：



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 旋锁是内核中提供的一种高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

KeAcquireSpinLock(&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](mailto: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：

