提到自旋锁那就必须要说链表，在上一篇《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。

在了解自旋锁之前需简单介绍一下内核中如何分配内存，一般而言分配内存有两个函数来实现 `ExAllocatePool` 可实现分配不带有任何标签的内存空间，而 `ExAllocatePoolWithTag` 则可分配带标签的，两者在使用上没有任何区别与之对应的就是释放 `ExFreePool` 用于释放非标签内存，而 `ExFreePoolWithTag` 则用于通过传入的标签释放对应的内存。

此处的分页属性常用的有三种，`NonPagedPool` 用于分配非分页内存，`PagedPool` 是分页内存，`NonPagedPoolExecute` 是带有执行权限的非分页内存。

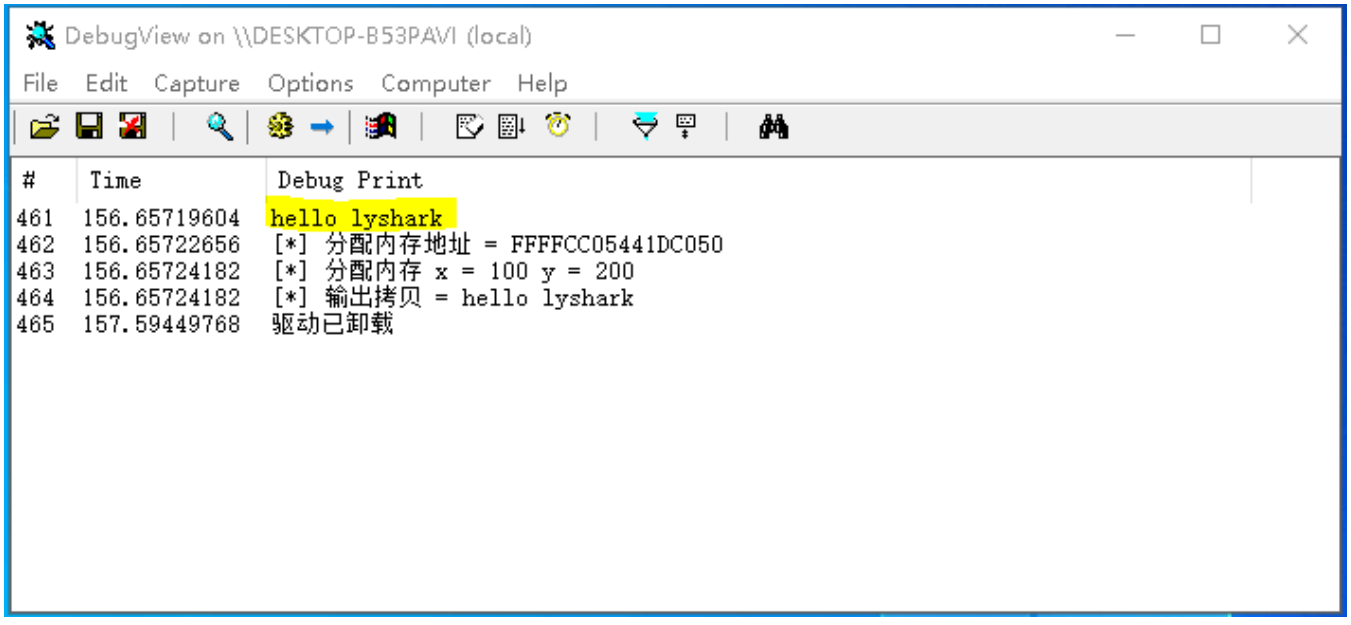
- NonPagedPool: 用于分配非分页内存，该内存不会被交换到磁盘上，并且可以直接被内核访问。适用于需要快速访问的内存，例如驱动程序的代码、中断处理程序、系统调用等。
- PagedPool: 用于分配分页内存，该内存可能会被交换到磁盘上，需要通过分页机制进行访问。适用于占用空间较大、访问频率较低的内存，例如缓存、数据结构等。
- NonPagedPoolExecute: 是带有执行权限的非分页内存，适用于需要执行代码的情况，例如一些特殊的驱动程序。

需要注意的是，使用 `NonPagedPoolExecute` 分配内存存在一定的安全风险，因为恶意软件可能会利用该内存进行攻击。因此，建议仅在必要时使用该分页属性。

```
#include <ntifs.h>

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

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("驱动已卸载 \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark \n");

    // 用于分配与释放非标签内存
    PVOID buffer = ExAllocatePool(NonPagedPool, 1024);

    DbgPrint("[*] 分配内存地址 = %p \n", buffer);

    ExFreePool(buffer);

    // 用于分配带有标签的内存
    pMyStruct ptr_buffer = (pMyStruct)ExAllocatePoolWithTag(NonPagedPoolExecute,
sizeof(pMyStruct),"lyshark");
```

```
    ptr_buffer->x = 100;
    ptr_buffer->y = 200;

    DbgPrint("[*] 分配内存 x = %d y = %d \n", ptr_buffer->x, ptr_buffer->y);

    ExFreePoolWithTag(ptr_buffer, "lyshark");

    UNICODE_STRING dst = { 0 };
    UNICODE_STRING src = RTL_CONSTANT_STRING(L"hello lyshark");

    dst.Buffer = (PWCHAR)ExAllocatePool(NonPagedPool, src.Length);
    if (dst.Buffer == NULL)
    {
        DbgPrint("[-] 分配空间错误 \n");
    }

    dst.Length = dst.MaximumLength = src.Length;

    RtlCopyUnicodeString(&dst, &src);

    DbgPrint("[*] 输出拷贝 = %wZ \n", dst);

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

代码输出效果如下图所示；



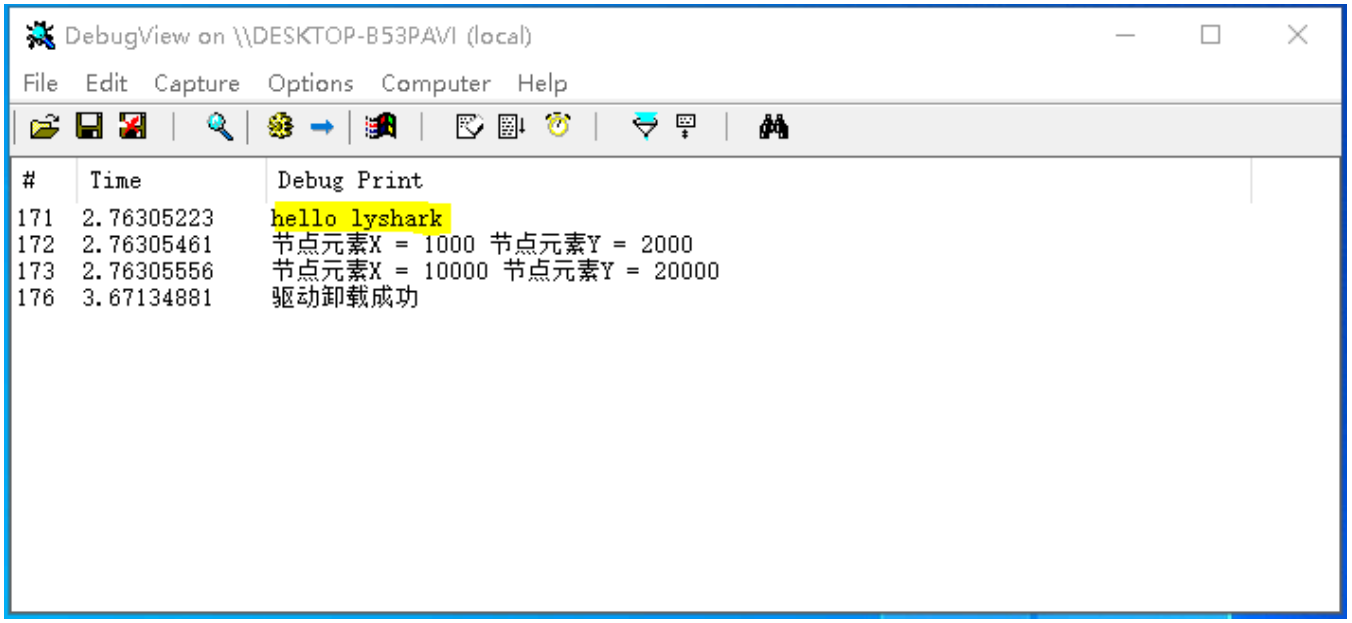
接着步入正题，以简单的链表为案例，链表主要分为单向链表与双向链表，单向链表的链表节点中只有一个链表指针，其指向后一个链表元素，而双向链表节点中有两个链表节点指针，其中`Blink`指向前一个链表节点`Flink`指向后一个节点，以双向链表为例。

```
#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");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark \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;
}
```

链表输出效果如下图所示;



如上所述，在内核开发中，多线程访问同一数据结构时会存在线程同步问题，为了解决这种问题，可以使用锁机制进行同步。自旋锁是一种常用的锁机制，它是一种高IRQL锁，用于同步和独占地访问某个资源。

- 自旋锁的基本思想是：当一个线程尝试获取锁时，如果锁已经被占用，则该线程不断循环（即自旋），直到锁被释放。自旋锁适用于锁的持有时间较短，且竞争者较少的情况下，可以避免进程上下文的切换和调度开销。

Windows内核提供了多种类型的自旋锁，例如`KSPIN_LOCK、KIRQL`等。其中，`KSPIN_LOCK`是最常用的自旋锁类型，可以通过`KeInitializeSpinLock`函数初始化一个自旋锁，并通过`KeAcquireSpinLock`和`KeReleaseSpinLock`函数进行加锁和解锁操作。

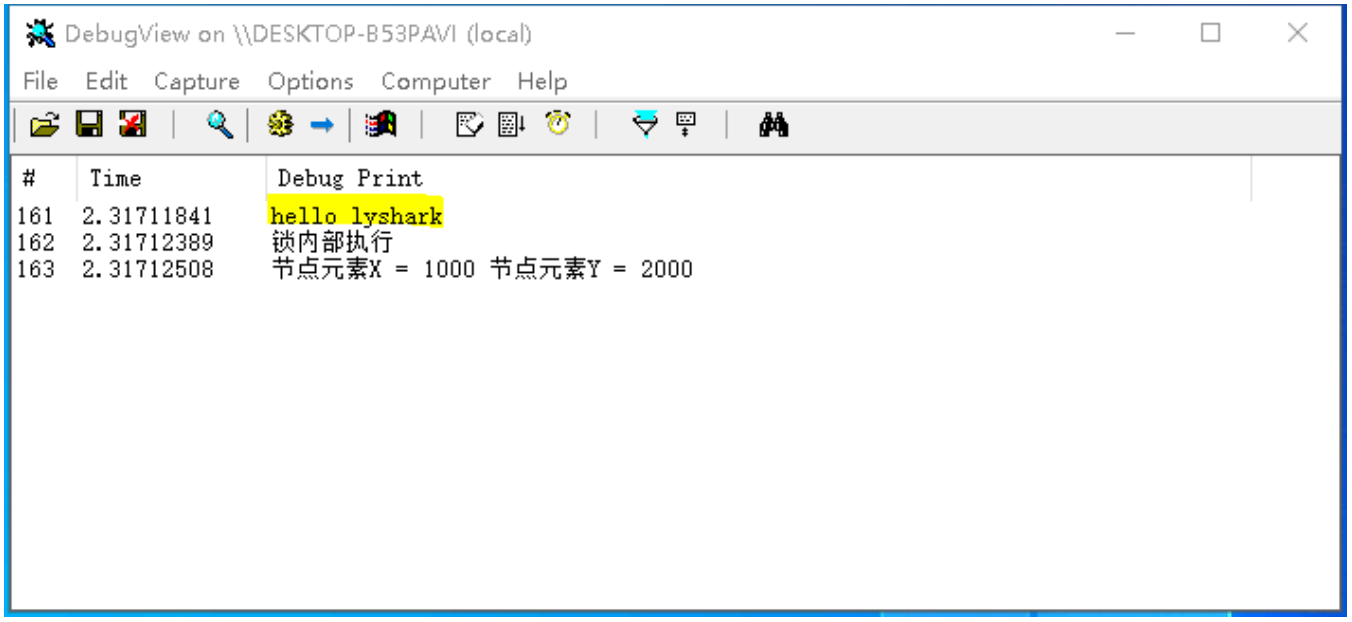
需要注意的是，使用自旋锁要注意死锁和优先级反转等问题，因此在实际应用中需要谨慎使用。

```
#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");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark \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图所示;


DebugView on \\DESKTOP-B53PAVI (local)
File Edit Capture Options Computer Help
Time Debug Print
161 2.31711841 hello lyshark
162 2.31712389 锁内部执行
163 2.31712508 节点元素X = 1000 节点元素Y = 2000