Lookaside Lists

7.1 Lookaside Lists
7.2 LookasideList driver source code
7.3 Working with Lookaside List
7.4 AddEntry routine
7.5 RemoveEntry routine

Source code: KmdKit\examples\basic\MemoryWorks\LookasideList

7.1 Lookaside Lists

Heap manager manages system and user heaps splitting heap space into blocks of equal size. When a heap allocation query arrives heap manager is trying to choose a free block with appropriate size. This can take some time, of course. If you need a memory blocks with fixed size but you don't know its amount and usage frequency beforehand you should use, for performance reasons, so called lookaside lists, which exists in the kernel mode only. The main difference of lookaside lists from the system pools is that you can allocate memory blocks with fixed and predefined size only. Allocation from the lookaside lists is faster because no need to search free region with suitable size.

When you first come to the lookaside lists the first problem you have to solve apart from the creation lookaside list itself will be management of the memory blocks you wish to allocate from it. Specifically where and how to store the block addresses you will refer and free to. This could be a serious problem since you don't know the quantity of these blocks. There are three structures for resolving such a problems:

- Singly linked list;
- S-list, sequenced singly-linked list (a singly linked list modification);
- Doubly linked list.

We'll examine doubly linked list only as the most universal solution.

The following code may look complicated if you deal with lookaside and doubly linked lists concept for the first time, but anyway it is rather simple.

Both are called lists, but they are completely different things though. The lookaside list is a group of preallocated memory blocks of equal size. Some of the blocks may be in use and some of them are not. System will walk through the list searching for the nearest free block when allocation request arrives. If free block found, the allocation can be satisfied very quickly. Otherwise the system must allocate from paged or nonpaged pool. The system automatically tune the number of freed blocks that lookaside lists store according to how often the allocations from the list occur - the more frequent the allocations, the more blocks are stored on a list. Lookaside lists are automatically reduced in size if they aren't being allocated from.

Double linked list is just a form of data organization. It is convenient to link homogeneous structures in a list and traverse through it. Double linked lists are used by the system intensively for internal structure handling.

7.2 LookasideList driver source code

I was thinking really hard but I was failed to depict sensible and simple example for this article. That's why this driver will act probably senselessly. However this should not prevent you from understanding the concepts or associative and double linked lists.

There will be no driver control program, use KmdManager (included in KmdKit package) or something similar. Use DebugView (http://www.sysinternals.com) or SoftICE console to watch driver's debug messages.

```
;@echo off
;goto make

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;
;  LookasideList - Merely allocates and releases some fixed-size blocks of memory.
;
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

.486
.model flat, stdcall
option casemap:none
```

```
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                  I N C L U D E   F I L E S
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

include \masm32\include\w2k\ntstatus.inc
include \masm32\include\w2k\ntddk.inc
include \masm32\include\w2k\ntoskrnl.inc


includelib \masm32\lib\w2k\ntoskrnl.lib

include \masm32\Macros\Strings.mac

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                   S T R U C T U R E S
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

SOME_STRUCTURE STRUCT
    SomeField1     DWORD        ?
    SomeField2     DWORD        ?
    ; . . .                         ; Any other fields come here

    ListEntry    LIST_ENTRY    <>   ; For tracking memory blocks.
                                    ; It can be the first member but
                                    ; to place it into is more common solution.

    ; . . .                         ; Any other fields come here
    SomeFieldX     DWORD        ?
SOME_STRUCTURE ENDS

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                           U N I N I T I A L I Z E D  D A T
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

.data?

g_pPagedLookasideList    PPAGED_LOOKASIDE_LIST    ?
g_ListHead               LIST_ENTRY               <>
g_dwIndex                DWORD                    ?

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                          C O D E
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

.code

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                          AddEntry
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

AddEntry proc uses esi

    invoke ExAllocateFromPagedLookasideList, g_pPagedLookasideList
    .if eax != NULL
        mov esi, eax

        invoke DbgPrint, \
               $CTA0("LookasideList: + Memory block allocated from lookaside list at address %08X\n"), esi

        invoke memset, esi, 0, sizeof SOME_STRUCTURE

        assume esi:ptr SOME_STRUCTURE

        lea eax, g_ListHead
        lea ecx, [esi].ListEntry
        InsertHeadList eax, ecx

        inc g_dwIndex
        mov eax, g_dwIndex
        mov [esi].SomeField1, eax

        invoke DbgPrint, $CTA0("LookasideList: + Entry #%d added\n"), [esi].SomeField1

        assume esi:nothing

    .else
        invoke DbgPrint, $CTA0("LookasideList: Very bad. Couldn't allocate from lookaside list\n")
    .endif

    ret

AddEntry endp

;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                        RemoveEntry
;:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

RemoveEntry proc uses esi

    IsListEmpty addr g_ListHead
    .if eax != TRUE

        lea eax, g_ListHead
        RemoveHeadList eax
        sub eax, SOME_STRUCTURE.ListEntry
        mov esi, eax

        invoke DbgPrint, $CTA0("LookasideList: - Entry #%d removed\n"), \
                            (SOME_STRUCTURE PTR [esi]).SomeField1

        invoke ExFreeToPagedLookasideList, g_pPagedLookasideList, esi

        invoke DbgPrint, \
            $CTA0("LookasideList: - Memory block at address %08X returned to lookaside list\n"), esi
    .else
        invoke DbgPrint, \
            $CTA0("LookasideList: An attempt was made to remove entry from empty lookaside list\n")
    .endif

    ret

RemoveEntry endp

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                                       DriverEntry
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

DriverEntry proc uses ebx pDriverObject:PDRIVER_OBJECT, pusRegistryPath:PUNICODE_STRING

    invoke DbgPrint, $CTA0("\nLookasideList: Entering DriverEntry\n")

    invoke ExAllocatePool, NonPagedPool, sizeof PAGED_LOOKASIDE_LIST
    .if eax != NULL

        mov g_pPagedLookasideList, eax

        invoke DbgPrint, \
        $CTA0("LookasideList: Nonpaged memory for lookaside list allocated at address %08X\n"), \
        g_pPagedLookasideList

        invoke ExInitializePagedLookasideList, g_pPagedLookasideList, NULL, NULL, \
                                               0, sizeof SOME_STRUCTURE, 'msaW', 0

        invoke DbgPrint, $CTA0("LookasideList: Lookaside list initialized\n")

        lea eax, g_ListHead
        InitializeListHead eax

        invoke DbgPrint, $CTA0("LookasideList: Doubly linked list head initialized\n")

        invoke DbgPrint, $CTA0("\nLookasideList: Start to allocate/free from/to lookaside list\n")

        and g_dwIndex, 0

        xor ebx, ebx
        .while ebx < 5

            invoke AddEntry
            invoke AddEntry

            invoke RemoveEntry

            inc ebx
        .endw

        invoke DbgPrint, $CTA0("\nLookasideList: Free the rest to lookaside list\n")

        .while TRUE

            invoke RemoveEntry

            lea eax, g_ListHead
            IsListEmpty eax
            .if eax == TRUE
                invoke DbgPrint, $CTA0("LookasideList: Doubly linked list is empty\n\n")
                .break
            .endif

        .endw

        invoke ExDeletePagedLookasideList, g_pPagedLookasideList

        invoke DbgPrint, $CTA0("LookasideList: Lookaside list deleted\n")

        invoke ExFreePool, g_pPagedLookasideList

        invoke DbgPrint, \
        $CTA0("LookasideList: Nonpaged memory for lookaside list at address %08X released\n"), \
        g_pPagedLookasideList

    .else
```

```
        invoke DbgPrint, \
                $CTA0("LookasideList: Couldn't allocate nonpaged memory for lookaside list control structure")
    .endif

    invoke DbgPrint, $CTA0("LookasideList: Leaving DriverEntry\n")

    mov eax, STATUS_DEVICE_CONFIGURATION_ERROR
    ret

DriverEntry endp

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

end DriverEntry

:make

set drv=LookasideList

\masm32\bin\ml /nologo /c /coff %drv%.bat
\masm32\bin\link /nologo /driver /base:0x10000 /align:32 /out:%drv%.sys /subsystem:native %drv%.obj

del %drv%.obj

echo.
pause
```

7.3 Working with Lookaside List

```
    invoke ExAllocatePool, NonPagedPool, sizeof PAGED_LOOKASIDE_LIST
    .if eax != NULL

        mov g_pPagedLookasideList, eax
```

We allocate nonpaged memory for PAGED_LOOKASIDE_LIST structure, which is used to manage lookaside list and save the pointer into the g_pPagedLookasideList variable. Note that lookaside list itself we use here is pageable, i.e. the memory we get from it can be paged out. The documentation is pretty clear about it.

```
        invoke ExInitializePagedLookasideList, g_pPagedLookasideList, NULL, NULL, \
                                        0, sizeof SOME_STRUCTURE, 'msaW', 0
```

ExInitializePagedLookasideList fills the PAGED_LOOKASIDE_LIST structure we allocated on the previous step. Now lookaside list is ready to use.

Note that during initialization we did not specify how many blocks we are require. But how the system knows exactly how much memory it should allocate? Indeed, if memory allocation hasn't been done beforehand the lookaside list will not work faster than common system pool allocation. The point is that initially the system allocates just a few (the quantity is defined by the system itself) blocks. So if we start to allocate from the lookaside list we'll get the pointers to those preallocated memory blocks. Once per second the system adjusts all system lookaside lists calling ExAdjustLookasideDepth. The system will allocate new blocks if it founds free block spare diminished during the adjustment. The number of the extra blocks depends on lookaside list's load, i.e. its allocation frequency. The system tries to adjust lookaside lists more effective way. If we had exhausted all preallocated blocks within adjustment time, the system just uses system pool allocation until the next adjustment. The important thing to understand is that if the allocation speed is too high we have no performance gain comparing the allocation form the system pool. You can estimate the efficiency of the lookaside list using MS Kernel Debugger's command "!lookaside".

```
kd> !lookaside ed374840

Lookaside "" @ ed374840 "Regm"
    Type     =     0001 PagedPool
    Current Depth  =        2   Max Depth  =        4
    Size           =     1024   Max Alloc  =     4096
    AllocateMisses =        4   FreeMisses =        0
    TotalAllocates =  1319722   TotalFrees =  1319720
    Hit Rate       =       99%  Hit Rate   =      100%
```

Let's see the lookaside list use efficiency of the RegMon (http://www.sysinternals.com) utility. As you can see the efficiency approaches to 100% considering huge amount (above one million) of alloc/free operations. The reason is that RegMon does not keep allocated block for a long time.

```
        lea eax, g_ListHead
        InitializeListHead eax
```

Calling InitializeListHead macro we initialize doubly linked list head. Now both LIST_ENTRY's fields contain the pointers to this structure itself. This means doubly linked list is empty (figure 7.1, imgage 1)

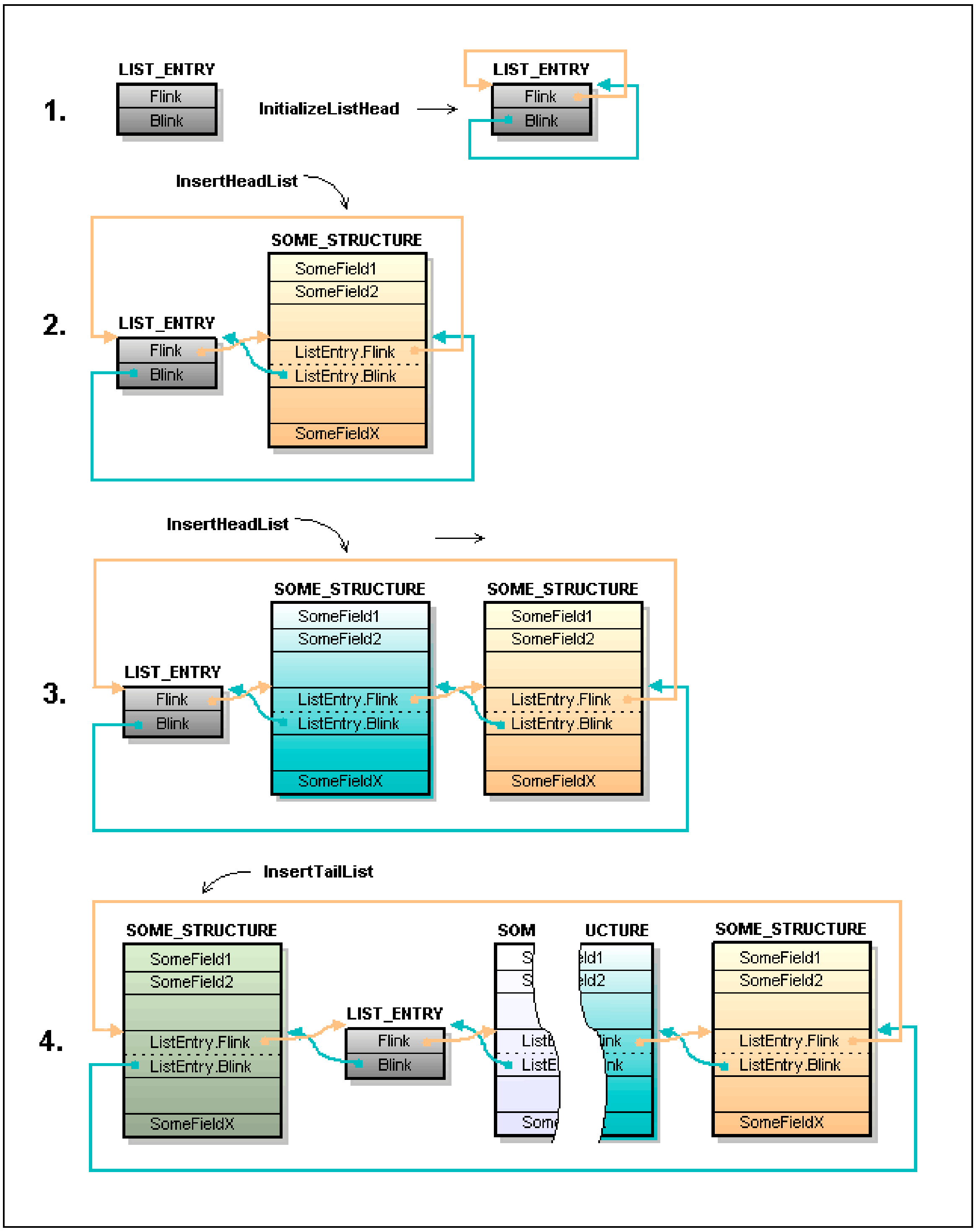

Figure. 7-1. This picture allows you visually realize how doubly linked list functions.

```
        and g_dwIndex, 0
```

This global variable is only used to place something into SOME_STRUCTURE allocated from lookaside list and to output its value through debug messages.

```
        xor ebx, ebx
        .while ebx < 5

            invoke AddEntry
            invoke AddEntry

            invoke RemoveEntry

            inc ebx
        .endw
```

Loop five times. Each pass adds two entries and removes one entry. Each entry represents SOME_STRUCTURE. All allocated structures linked to each other using doubly linked list.

This loop simulates the random allocation from the lookaside list. We assume here we work somehow with allocated entries. For example, we could write a driver intercepting calls of some system service, say ZwOpenKey, and saving information into the allocated memory.

```
        .while TRUE

            invoke RemoveEntry

            lea eax, g_ListHead
            IsListEmpty eax
            .if eax == TRUE
                .break
            .endif

        .endw
```

At this point we have some amount of allocated from lookaside list entries linked together in doubly linked lists. We assume these entries are not needed for us anymore.

We call RemoveEntry routine in the endless loop. RemoveEntry removes an entry from the head of a doubly linked list and releases it back to the lookaside list. The loop runs until doubly linked list is empty. Calling IsListEmpty macro checks this condition. IsListEmpty checks to see whether both fields of the doubly linked list head (LIST_ENTRY structure) point to the head itself. At this point we came to the state we have right after InitializeListHead macro call (figure 7-1, imgage 1).

```
        invoke ExDeletePagedLookasideList, g_pPagedLookasideList
```

When doubly linked list is empty all entries allocated from lookaside list are returned back as well, since we used doubly linked list to manage allocations from the lookaside list. Now we can delete lookaside list by calling ExDeletePagedLookasideList. It frees any remaining entries in the lookaside list and then removes the list from the system-wide set of active lookaside lists.

```
        invoke ExFreePool, g_pPagedLookasideList
```

ExFreePool releases nonpaged memory allocated for PAGED_LOOKASIDE_LIST structure. In case you forget (I didn't) to call ExDeletePagedLookasideList beforehand you will see BSOD since in about a second the system will try to adjust missing lookaside list.

```
    mov eax, STATUS_DEVICE_CONFIGURATION_ERROR
    ret
```

Since we have released all allocated resources we can unload the driver.

7.4 AddEntry routine

We call AddEntry when we need a new memory block. It allocates new entry from the lookaside list and adds it into doubly linked list.

```
    invoke ExAllocateFromPagedLookasideList, g_pPagedLookasideList
    .if eax != NULL
        mov esi, eax
```

Calling ExAllocateFromPagedLookasideList we get the new memory block pointer and save it in esi. Note we don't tell the system about block size because the size was defined while calling ExInitializePagedLookasideList and is equal to the size of SOME_STRUCTURE.

```
        invoke memset, esi, 0, sizeof SOME_STRUCTURE
```

Zeroing allocated block. It doesn't need though.

```
        assume esi:ptr SOME_STRUCTURE
```

Now we have new instance of the SOME_STRUCTURE. We should link it to our doubly linked list, which is empty during the first AddEntry run.

```
        lea eax, g_ListHead
        lea ecx, [esi].ListEntry
        InsertHeadList eax, ecx
```

Figure 7-1, image 2 depicts a state after the first InsertHeadList macro call. Pay your attention to the second argument of the InsertHeadList macro, which is not an address of the SOME_STRUCTURE but an address of its ListEntry field. I.e. InsertHeadList macro accepts the pointers to the two LIST_ENTRY structures, the first one is the head of doubly linked list and the second one is a structure member we need to link to this doubly linked list. InsertHeadList links the new structure at the head of the doubly linked list (to the right in Figure 7-1). You can use InsertTailList macro to link at the tail.

Both macros produce the same result if you add the first entry to the doubly linked list and after that it will looks like it reflected on the Figure 7-1, image 2.

If doubly linked list is not empty InsertHeadList macro will split the doubly linked list between its head and the entry to the right and place the new entry in between (see Figure 7-1, image 3). InsertTailList macro does the same but at the tail.

Hope everything is pretty clear now.

```
        inc g_dwIndex
        mov eax, g_dwIndex
        mov [esi].SomeField1, eax
```

We save newly created entry number in the SomeField1. You can watch the order in which those structures are added/removed in DbgView.

7.5 RemoveEntry routine

RemoveEntry routine is the reciprocal of the AddEntry. It unlinks the entry from the head of the doubly linked list and returns it back to the lookaside list.

```
    IsListEmpty addr g_ListHead
    .if eax != TRUE
```

Make sure the doubly linked list is empty.

```
        lea eax, g_ListHead
        RemoveHeadList eax
```

RemoveHeadList unlinks the entry from the doubly linked list's head (as you already guess RemoveTailList macro does the same but from the tail. You can also remove any entry using RemoveEntryList macro). At this point extracted from the doubly linked list entry exist on its own and doubly linked list is enclosed to link remaining entries together.

```
        sub eax, SOME_STRUCTURE.ListEntry
        mov esi, eax
```

Pay your attention to this point. RemoveTailList/RemoveHeadList/RemoveEntryList return a pointer to the entry (nested LIST_ENTRY structure) that was at the tail/head/middle of the list but not the pointer to unlinked SOME_STRUCTURE structure itself. The macros don't know exact place of the LIST_ENTRY in the structure. And there is no way for them to know about. This is entirely up to you to calculate offset to the ListEntry field in the SOME_STRUCTUREto get the pointer to the structure itself (that's what DDK's CONTAINING_RECORD macro does).

```
        invoke ExFreeToPagedLookasideList, g_pPagedLookasideList, esi
```

ExFreeToPagedLookasideList returns the entry to the lookaside list or to paged pool.