System Memory Heaps

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

Assuming you already grasp the basics, let's review some of the essential base techniques. I'd like to underscore the word "some" because drivers can do a lot of things. Since it is impossible to go further without the knowledge of memory management, let's start with it.

Memory Manager provides to the user processes a rich set of API for memory management, which can be divided into three groups of functions: virtual memory functions, memory-mapped file functions, and heap functions.

Kernel components, including drivers, have more advanced facilities. For example, the driver is able to allocate a contiguous memory range in the physical address space. The function names, Memory Manager provides, begin with the prefix "Mm". In addition, the executive support routines that begin with "Ex" are used to allocate and free memory from the system pools (paged and nonpaged) as well as to manipulate look-aside lists (look-aside lists provide more faster allocation but only predefined fixed block size).

6.1 System Memory Heaps

System memory heaps (have nothing common with user heaps, btw) are represented with two so called memory pools located in the system address space.

- Nonpaged Pool. Nonpaged Pool pages cannot be paged out to the swap file and, of course, never paged in back. This pool always presents in the physical memory and thus can be accessed at any time (at any IRQL level) without incurring a page fault. One of the reasons the nonpaged pool is required is that any access to its pages never causes the Page Fault. Such errors leads to system crash at IRQL >= DISPATCH_LEVEL;

- Paged Pool. These pages can be paged in and out. You can use this memory only at IRQL less than DISPATCH_LEVEL.

Both pools are located in the system address space and available from within any process context. There is ExAllocatePoolXXX function set to allocate memory from the system pools and ExFreePool to release.

Before we start using one of them, I'd like to note some fundamental points:

Access to paged out memory at IRQL >= DISPATCH_LEVEL, as I said, leads to system crash.

Keep in mind that if at the moment you are accessing the pageable memory it physically presents no crash happens, even at IRQL >= DISPATCH_LEVEL. But sooner or later the system pages this memory out and your access to it results in system crash.

It is not worth to use nonpaged memory anywhere you like. This resource is more expensive than paged one.

After you have allocated memory from any system pool, no matter what happens with your driver, this memory will not return to the system back until ExFreePool is called. I.e. until the driver explicitly releases allocated memory it will stay resident inside the system even after unloading the driver. As I already told many times, the drivers must explicitly release all allocated resources.

Memory allocated from system pools doesn't zeroed out and could contain garbage left by the last owner. So it is better to zero it before usage.

It is easy to define the memory type (paged or nonpaged) you need. If some code should access the memory at IRQL >= DISPATCH_LEVEL it is definitely should be nonpaged type only. Both the code itself and the data it touches should reside in nonpaged memory. By default the driver is loaded in the nonpaged memory except "INIT" section and sections, whose names starts with "PAGE". If you do not make any actions to change driver's memory attributes (do not call MmPageEntireDriver that makes the driver's image pageable, for example) you should not care about driver itself - it always presents in the memory.

In the previous articles we have discussed the IRQL levels common driver routines (DriverEntry, DriverUnload, DispatchXxx) called at.

DDK gives us information about IRQL each function called at. For example, we will use IoInitializeTimer function in one of the further articles. The description says that it executes by the system when timer event occurs at IRQL = DISPATCH_LEVEL. This means that this routine along any memory it accesses must be nonpaged.

At worst, if you not sure about current IRQL, you determine it calling KeGetCurrentIrql like this:

```
invoke KeGetCurrentIrql
.if eax < DISPATCH_LEVEL
    ; use any memory
.else
    ; use nonpaged memory only
.endif
```

6.2 Allocation from the system pool

Let's take a look at very simple driver SystemModules as an example. The main action concentrates inside DriverEntry routine. We'll allocate paged memory (I'm sure, you remember the DriverEntry runs at IRQL = PASSIVE_LEVEL, so it's OK to use paged memory), write something useful to it, release and force the system to unload the driver.

```
;@echo off
;goto make

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;
;  SystemModules - Allocate memory from the system pool and use it
;
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

.386
.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\native.inc
include \masm32\include\w2k\ntoskrnl.inc


includelib \masm32\lib\w2k\ntoskrnl.lib

include \masm32\Macros\Strings.mac

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                              D I S C A R D A B L E   C O D E
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

.code INIT

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

DriverEntry proc uses esi edi ebx pDriverObject:PDRIVER_OBJECT, pusRegistryPath:PUNICODE_STRING

local cb:DWORD
local p:PVOID
local dwNumModules:DWORD
local pMessage:LPSTR
local buffer[256+40]:CHAR

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

    and cb, 0
    invoke ZwQuerySystemInformation, SystemModuleInformation, addr p, 0, addr cb
    .if cb != 0

        invoke ExAllocatePool, PagedPool, cb
        .if eax != NULL
            mov p, eax

            invoke DbgPrint, \
                   $CTA0("SystemModules: %u bytes of paged memory allocted at address %08X\n"), cb, p

            invoke ZwQuerySystemInformation, SystemModuleInformation, p, cb, addr cb
            .if eax == STATUS_SUCCESS
                mov esi, p

                push dword ptr [esi]
                pop dwNumModules

                mov cb, (sizeof SYSTEM_MODULE_INFORMATION.ImageName + 100)*2

                invoke ExAllocatePool, PagedPool, cb
                .if eax != NULL
                    mov pMessage, eax

                    invoke DbgPrint, \
                           $CTA0("SystemModules: %u bytes of paged memory allocted at address %08X\n"), \
                            cb, pMessage

                    invoke memset, pMessage, 0, cb

                    add esi, sizeof DWORD
                    assume esi:ptr SYSTEM_MODULE_INFORMATION

                    xor ebx, ebx
                    .while ebx < dwNumModules

                        lea edi, [esi].ImageName
                        movzx ecx, [esi].ModuleNameOffset
                        add edi, ecx

                        invoke _strnicmp, edi, $CTA0("ntoskrnl.exe", szNtoskrnl, 4), sizeof szNtoskrnl - 1
                        push eax
                        invoke _strnicmp, edi, $CTA0("ntice.sys", szNtIce, 4), sizeof szNtIce - 1
                        pop ecx

                        and eax, ecx
                        .if ZERO?
                            invoke _snprintf, addr buffer, sizeof buffer, \
                                    $CTA0("SystemModules: Found %s base: %08X size: %08X\n", 4), \
                                    edi, [esi].Base, [esi]._Size
                            invoke strcat, pMessage, addr buffer
                        .endif

                        add esi, sizeof SYSTEM_MODULE_INFORMATION
                        inc ebx

                    .endw
                    assume esi:nothing

                    mov eax, pMessage
                    .if byte ptr [eax] != 0
                        invoke DbgPrint, pMessage
                    .else
                        invoke DbgPrint, \
                               $CTA0("SystemModules: Found neither ntoskrnl nor ntice.\n")
                    .endif

                    invoke ExFreePool, pMessage
                    invoke DbgPrint, $CTA0("SystemModules: Memory at address %08X released\n"), pMessage

                .endif
            .endif

            invoke ExFreePool, p
            invoke DbgPrint, $CTA0("SystemModules: Memory at address %08X released\n"), p

        .endif
    .endif

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

    mov eax, STATUS_DEVICE_CONFIGURATION_ERROR
    ret

DriverEntry endp

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

end DriverEntry

:make

set drv=SystemModules

\masm32\bin\ml /nologo /c /coff %drv%.bat
\masm32\bin\link /nologo /driver /base:0x10000 /align:32 /out:%drv%.sys /subsystem:native /ignore:4078 %drv%.
obj

del %drv%.obj

echo.
pause
```

As a "something useful" let's take a modules list loaded into the system address space (this list includes system modules - ntoskrnl.exe, hal.dll, etc. and device drivers) and try to find ntoskrnl.exe and ntice.sys. We get system modules list calling ZwQuerySystemInformation with SystemModuleInformation information class. You can find description of this function in the Garry Nebbett's book "Windows NT-2000 Native API Reference". Btw, ZwQuerySystemInformation is a unique function. It returns a huge amount of system information of any kind.

I not provide driver control program here. Use KmdManager (included in KmdKit package) or similar utility and check driver debug messages using DebugView (http://www.sysinternals.com) or SoftICE console.

```
    and cb, 0
    invoke ZwQuerySystemInformation, SystemModuleInformation, addr p, 0, addr cb
```

First of all we should determine how much space information of our interest will take. Calling ZwQuerySystemInformation above showed the way we get STATUS_INFO_LENGTH_MISMATCH (this is quite normal since size of the buffer is zero), but cb variable receives desired buffer size. This way we get buffer size needed. Address of p is needed here for normal functioning of ZwQuerySystemInformation only.

```
    .if cb != 0
        invoke ExAllocatePool, PagedPool, cb
```

ExAllocatePool allocates required amount of memory from paged pool (first argument, for nonpaged memory specify NonPagedPool accordingly). ExAllocatePool is even simpler than its user-mode counterpart HeapAlloc. It takes only two arguments: the pool type (paged or nonpaged) and memory size required. Piece of cake!

```
        .if eax != NULL
```

If ExAllocatePool returns nonzero value, then it is a pointer to allocated buffer.

Examining debug information notice that buffer address ExAllocatePool returned is multiple of page size. If the size of requested memory is more or equal to page size (in our case this size is noticeably larger than the size of one page) then allocated memory starts from the page boundary.

```
            mov p, eax
            invoke ZwQuerySystemInformation, SystemModuleInformation, p, cb, addr cb
```

We call ZwQuerySystemInformation again using pointer to buffer and its size as an arguments.

```
            .if eax == STATUS_SUCCESS
                mov esi, p
```

If STATUS_SUCCESS returned our buffer contains the system modules list in the form of SYSTEM_MODULE_INFORMATION (defined in include\w2k\native.inc) structures array.

```
SYSTEM_MODULE_INFORMATION STRUCT        ;Information Class 11
    Reserved            DWORD   2 dup(?)
    Base                PVOID   ?
    _Size               DWORD   ?
    Flags               DWORD   ?
    Index               WORD    ?
    Unknown             WORD    ?
    LoadCount           WORD    ?
    ModuleNameOffset    WORD    ?
    ImageName           CHAR 256 dup(?)
SYSTEM_MODULE_INFORMATION ENDS
```

cb variable receives the number of bytes actually returned , but it is useless for us now.

I assume here no new modules appear between two ZwQuerySystemInformation calls. It is rarely happen so highly unlikely. This is acceptable for a learning purpose but you'd better use more safe approach - repeatedly call ZwQuerySystemInformation in a loop increasing the buffer size at every iteration until its size will satisfy the request.

```
                push dword ptr [esi]
                pop dwNumModules
```

The very first double word of the buffer contains the number of modules followed immediately by an array of SYSTEM_MODULE_INFORMATION. Keep modules number in the dwNumModules.

```
                mov cb, (sizeof SYSTEM_MODULE_INFORMATION.ImageName + 100)*2
                invoke ExAllocatePool, PagedPool, cb
                .if eax != NULL
                    mov pMessage, eax
```

We need one buffer more to keep the names of two modules we are looking for and some additional information. We assume (sizeof SYSTEM_MODULE_INFORMATION.ImageName + 100)*2 is enough.

Notice that this time buffer address is not multiple of page size because it size is less than a page size.

```
                    invoke memset, pMessage, 0, cb
```

For safety precautions fill the buffer with zero to assure the strings we are going to copy is zero terminated.

```
                    add esi, sizeof DWORD
                    assume esi:ptr SYSTEM_MODULE_INFORMATION
```

Skip DWORD containing modules number so esi now points to the first SYSTEM_MODULE_INFORMATION structure.

```
                    xor ebx, ebx
                    .while ebx < dwNumModules
```

We walk dwNumModules times through the structures array searching for ntoskrnl.exe and ntice.sys.

On the multiprocessor system ntoskrnl.exe module has the name ntkrnlmp.exe, and on the system with PAE support ntkrnlpa.exe and ntkrpamp.exe respectively. I assume here you are not the lucky one who owns such a system.

```
                        lea edi, [esi].ImageName
                        movzx ecx, [esi].ModuleNameOffset
                        add edi, ecx
```

ImageName and ModuleNameOffset fields contain full path to the module and relative offset to the module's name inside the path respectively.

```
                        invoke _strnicmp, edi, $CTA0("ntoskrnl.exe", szNtoskrnl, 4), sizeof szNtoskrnl - 1
                        push eax
                        invoke _strnicmp, edi, $CTA0("ntice.sys", szNtIce, 4), sizeof szNtIce - 1
                        pop ecx
```

strnicmp do case-insensitive comparison of two ANSI strings. Third argument is the number of characters to compare. It is not necessary here since names of the modules in the SYSTEM_MODULE_INFORMATION are zero terminated and we can use _stricmp. But I use _strnicmp for a safety reasons.

Btw, ntoskrnl.exe exports a lot of basic string functions - strcmp, strcpy, strlen etc.

```
                        and eax, ecx
                        .if ZERO?
                            invoke _snprintf, addr buffer, sizeof buffer, \
                                    $CTA0("SystemModules: Found %s base: %08X size: %08X\n", 4), \
                                    edi, [esi].Base, [esi]._Size
                            invoke strcat, pMessage, addr buffer
                        .endif

                        add esi, sizeof SYSTEM_MODULE_INFORMATION
                        inc ebx

                    .endw
                    assume esi:nothing
```

If any of the above mentioned modules founded, we format the string containing modules names, base address and size using _snprintf (exported by the kernel too) and add it to the buffer. Labels szNtoskrnl and szNtIce used in the sizeof operator. You can switch label and alignment in my string macros - they are detected automatically. You can also use either label or alignment only (see macros \Strings.mac for details).

```
                    mov eax, pMessage
                    .if byte ptr [eax] != 0
                        invoke DbgPrint, pMessage
                    .else
                        invoke DbgPrint, \
                               $CTA0("SystemModules: Found neither ntoskrnl nor ntice.\n")
                    .endif
```

Since we zeroed buffer before use the nonzero value in first byte means we have found something.

```
                    invoke ExFreePool, pMessage
                .endif
            .endif
            invoke ExFreePool, p
        .endif
    .endif
```

Release the memory allocated from the system pools.

```
    mov eax, STATUS_DEVICE_CONFIGURATION_ERROR
    ret
```

Force the system to unload the driver.

As you can see, using the system pools even simpler than using heaps in the user-mode. The only question is to correctly define the pool type.

A lot of ZwXxx functions exported by the user-mode ntdll.dll are just the gates to the kernel-mode equivalents. Notice that number of arguments and their meaning entirely the same. You can make profit of it. Since errors inside the kernel lead to the system crash you can debug your code in the user-mode and then copy it in your driver with very little changes if any. For example, ZwQuerySystemInformation called from the ntdll.dll returns the same information. Using this trick you can avoid a lot of reboots.
