Chapter 4

Memory Management

IN THIS CHAPTER

- Examining memory models in Microsoft operating systems
- Examining how Windows NT uses features of the 80386 processor's architecture
- Exploring the function of virtual memory

MEMORY MANAGEMENT HAS ALWAYS been one of the most important and interesting aspects of any operating system for serious developers. It is an aspect that kernel developers ignore. Memory management, in essence, provides a thumbnail impression of any operating system.

Microsoft has introduced major changes in the memory management of each new operating system they have produced. Microsoft had to make these changes because they developed all of their operating systems for Intel microprocessors, and Intel introduced major changes in memory management support with each new microprocessor they introduced. This chapter is a journey through the various Intel microprocessors and the memory management changes each one brought along with it in the operating system that used it.

Memory Models in Microsoft Operating Systems

Early PCs based on Intel 8086/8088 microprocessors could access only 640K of RAM and used the segmented memory model. Consequently, good old DOS allows only 640K of RAM and restricts the programmer to the segmented memory model.

In the segmented model, the address space is divided into segments. Proponents of the segmented model claim that it matches the programmer's view of memory. They claim that a programmer views memory as different segments containing code, data, stack, and heap. Intel 8086 supports very primitive segmentation. A segment, in the 8086 memory model, has a predefined base address. The length of each segment is also fixed and is equal to 64K. Some programs find a single segment insufficient. Hence, there are a number of memory models under DOS. For example, the tiny model that supports a single segment for code, data, and stack

together, or the small model that allows two segments - one for code and the other for data plus stack, and so on. This example shows how the memory management provided by an operating system directly affects the programming environment.

The Intel 80286 (which followed the Intel 8086) could support more than 640K of RAM. Hence, programmers got new interface standards for accessing extended and expanded memory from DOS. Microsoft's second-generation operating system, Windows 3.1, could run on 80286 in standard mode and used the segmented model of 80286. The 80286 provided better segmentation than the 8086. In 80286's model, segments can have a programmable base address and size limit. Windows 3.1 had another mode of operation, the *enhanced mode*, which required the Intel 80386 processor. In the enhanced mode, Windows 3.1 used the paging mechanisms of 80386 to provide additional performance. The virtual 8086 mode was also used to implement multiple DOS boxes on which DOS programs could run.

Windows 3.1 does not make full use of the 80386's capabilities. Windows 3.1 is a 16-bit operating system, meaning that 16-bit addresses are used to access the memory and the default data size is also 16 bits. To make full use of 80386's capabilities, a 32-bit operating system is necessary. Microsoft came up with a 32-bit operating system, Windows NT. The rest of this chapter examines the details of Windows NT memory management. Microsoft also developed Windows 95 after Windows NT. Since both these operating systems run on 80386 and compatibles, their memory management schemes have a lot in common. However, you can best appreciate the differences between Windows NT and Windows 95/98 after we review Windows NT memory management. Therefore, we defer this discussion until a later section of this chapter.

Windows NT Memory Management Overview

We'll first cover the view Windows NT memory management presents to the outside world. In the next section, we explain the special features provided by Intel microprocessors to implement memory management. Finally, we discuss how Windows NT uses these features to implement the interface provided to the outside world.

Memory Management Interface — Programmer's View

Windows NT offers programmers a 32-bit flat address space. The memory is not segmented; rather, it is 4GB of continuous address space. (Windows NT marked the end of segmented architecture — programmers clearly preferred flat models to segmented ones.) Possibly, with languages such as COBOL where you need to declare data and code separately, programmers view memory as segments. However, with new languages such as C and C++, data variables and code can be freely mixed and

the segmented memory model is no longer attractive. Whatever the reason, Microsoft decided to do away with the segmented memory model with Windows NT. The programmer need not worry whether the code/data fits in 64K segments. With the segmented memory model becoming extinct, the programmer can breathe freely. At last, there is a single memory model, the 32-bit flat address space.

Windows NT is a protected operating system; that is, the behavior (or misbehavior) of one process should not affect another process. This requires that no two processes are able to see each other's address space. Thus, Windows NT should provide each process with a separate address space. Out of this 4GB address space available to each process, Windows NT reserves the upper 2GB as kernel address space and the lower 2GB as user address space, which holds the user-mode code and data. The entire address space is not separate for each process. The kernel code and kernel data space (the upper 2GB) is common for all processes; that is, the kernel-mode address space is shared by all processes. The kernel-mode address space is protected from being accessed by user-mode code. The system DLLs (for example, KERNEL32.DLL, USER32.DLL, and so on) and other DLLs are mapped in user-mode space. It is inefficient to have a separate copy of a DLL for each process. Hence, all processes using the DLL or executable module share the DLL code and incidentally the executable module code. Such a shared code region is protected from being modified because a process modifying shared code can adversely affect other processes using the code.

Sharing of the kernel address space and the DLL code can be called implicit sharing. Sometimes two processes need to share data explicitly. Windows NT enables explicit sharing of address space through memory-mapped files. A developer can map a named file onto some address space, and further accesses to this memory area are transparently directed to the underlying file. If two or more processes want to share some data, they can map the same file in their respective address spaces. To simply share memory between processes, no file needs to be created on the hard disk.

Below the Operating System

In her book *Inside Windows NT,* Helen Custer discusses memory management in the context of the MIPS processor. Considering that a large number of the readers would be interested in a similar discussion that focuses on Intel processors, we discuss the topic in the context of the Intel 80386 processor (whose memory management architecture is mimicked by the later 80486 and Pentium series). If you are already conversant with the memory management features of the 80386 processor, you may skip this section entirely.

We now examine the 80386's addressing capabilities and the fit that Windows NT memory management provides for it. Intel 80386 is a 32-bit processor; this implies that the address bus is 32-bit wide, and the default data size is as well. Hence, 4GB (2^{32} bytes) of physical RAM can be addressed by the microprocessor. The microprocessor supports segmentation as well as paging. To access a memory

location, you need to specify a 16-bit segment selector and a 32-bit offset within the segment. The segmentation scheme is more advanced than that in 8086. The 8086 segments start at a fixed location and are always 64K in size. With 80386, you can specify the starting location and the segment size separately for each segment.

Segments may overlap - that is, two segments can share address space. The necessary information (the starting offset, size, and so forth) is conveyed to the processor via segment tables. A segment selector is an index into the segment table. At any time, only two segment tables can be active: a Global Descriptor Table (GOT) and a Local Descriptor Table (GOT). A bit in the selector indicates whether the processor should refer to the LOT or the GDT. Two special registers, GDTR and LDTR, point to the GDT and the LOT, respectively. The instructions to load these registers are privileged, which means that only the operating system code can execute them.

A segment table is an array of segment descriptors. A *segment descriptor* specifies the starting address and the size of the segment. You can also specify some access permission bits with a segment descriptor. These bits specify whether a particular segment is read-only, read-write, executable, and so on. Each segment descriptor has 2 bits specifying its privilege level, called as the descriptor privilege level (DPL).

The processor compares the DPL with the Requested Privilege Level (RPL) before granting access to a segment. The RPL is dictated by 2 bits in the segment selector while specifying the address. The Current Privilege Level (CPL) also plays an important role here. The CPL is the DPL of the code selector being executed. The processor grants access to a particular segment only if the DPL of the segment is less than or equal to the RPL as well as the CPL. This serves as a protection mechanism for the operating system. The CPL of the processor can vary between 0 and 3 (because 2 bits are assigned for CPL). The operating system code generally runs at CPL=0, also called as ring 0, while the user processes run at ring 3. In addition, all the segments belonging to the operating system are allotted DPL=0. This arrangement ensures that the user mode cannot access the operating system memory segments.

It is very damaging to performance to consult the segment tables, which are stored in main memory, for every memory access. Caching the segment descriptor in special CPU registers, namely, CS (Code Selector), DS (Data Selector), SS (Stack Selector), and two general-purpose selectors called ES and FS, solves this problem. The first three selector registers in this list- that is, CS, DS, and SS - act as default registers for code access, data access, and stack access, respectively.

To access a memory location, you specify the segment and offset within that segment. The first step in address translation is to add the base address of the segment to the offset. This 32-bit address is the physical memory address if paging is not enabled. Otherwise this address is called as the logical or linear address and is converted to a physical RAM address using the page address translation mechanism (refer to Figure 4-1).

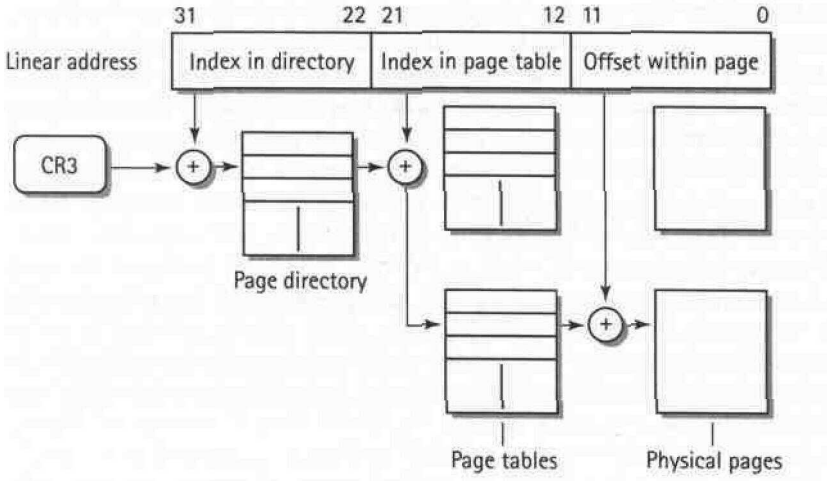

Figure 4-1: Linear to physical address translation

The memory management scheme is popularly known as *paging* because the memory is divided into fixed-size regions called *pages.* On Intel processors (80386 and higher), the size of one page is 4 kilobytes. The 32-bit address bus can access up to 4GB of RAM. Hence, there are one million (4GB/4K) pages.

Page address translation is a logical to physical address mapping. Some bits in the logical/linear address are used as an index in the page table, which provides a logical to physical mapping for pages. The page translation mechanism on Intel platforms has two levels, with a structure called page table directory at the second level. As the name suggests, a *page table directory* is an array of pointers to page tables. Some bits in the linear address are used as an index in the page table directory to get the appropriate page table to be used for address translation.

The page address translation mechanism in the 80386 requires two important data structures to be maintained by the operating system, namely, the page table directory and the page tables. A special register, CR3, points to the current page table directory. This register is also called Page Directory Base Register (PDBR). A page table directory is a 4096-byte page with 1024 entries of 4 bytes each. Each entry in the page table directory points to a page table. A page table is a 4096-byte page with 1024 entries of 4 bytes *(32* bits) each. Each Page Table Entry (PTE) points to a physical page. Since there are 1 million pages to be addressed, out of the 32 bits in a PTE, 20 bits act as upper 20 bits of physical address. The remaining 12 bits are used to maintain attributes of the page.

Some of these attributes are access permissions. For example, you can denote a page as read-write or read-only. A page also has an associated security bit called as the supervisor bit, which specifies whether a page can be accessed from the user-mode code or only from the kernel-mode code. A page can be accessed only at ring 0 if this bit is set. Two other bits, namely, the accessed bit and the dirty bit, indicate

the status of the page. The processor sets the accessed bit whenever the page is accessed. The processor sets the dirty bit whenever the page is written to. Some bits are available for operating system use. For example, Windows NT uses one such bit for implementing the copy-on-write protection. You can also mark a page as invalid and need not specify the physical page address. Accessing such a page generates a page fault exception. An exception is similar to a software interrupt. The operating system can install an exception handler and service the page faults. You'll read more about this in the following sections.

32-bit memory addresses break down as follows. The upper 10 bits of the linear address are used as the page directory index, and a pointer to the corresponding page table is obtained. The next 10 bits from the linear address are used as an index in this page table to get the base address of the required physical page. The remaining 12 bits are used as offset within the page and are added to the page base address to get the physical address.

The Inside Look

In this section, we examine how Windows NT has selectively utilized existing features of the 80386 processor's architecture to achieve its goals.

Flat Address Space

First, let's see how Windows NT provides 32-bit flat address space to the processes. As we know from the previous section, Intel 80386 offers segmentation as well as paging. So how does Windows NT provide a flat memory instead of a segmented one? Turn off segmentation? You cannot turn off segmentation on 80386. However, the 80386 processor enables the operating system to load the segment register once and then specify only 32-bit offsets for subsequent instructions. This is exactly what Windows NT does. Windows NT initializes all the segment registers to point to memory locations from 0 to 4GB, that is, the base is set as 0 and the limit is set as 4GB. The CS, SS, DS, and ES are initialized with separate segment descriptors all pointing to locations from 0 to 4GB. So now the applications can use only 32-bit offset, and hence see a 32-bit flat address space. A 32-bit application running under Windows NT is not supposed to change any of its segment registers.

Process Isolation

The next question that comes to mind is, "How does Windows NT keep processes from seeing each other's address space?" Again, the mechanism for achieving this design goal is simple. Windows NT maintains a separate page table directory for each process and based on the process in execution, it switches to the corresponding page table directory. As the page table directories for different processes point

to different page tables and these page tables point to different physical pages and only one directory is active at a time, no process can see any other process's memory. When Windows NT switches the execution context, it also sets the CR3 register to point to the appropriate page table directory. The kernel-mode address space is mapped for all processes, and all page table directories have entries for kernel address space. However, another feature of 80386 is used to disallow user-mode code from accessing kernel address space. All the kernel pages are marked as supervisor pages; therefore, user-mode code cannot access them.

Code Page Sharing in DLLs

For sharing code pages of **a** DLL, Windows NT maps corresponding page table entries for all processes sharing the DLL onto the same set of physical pages. For example, if process A loads X.DLL at address **xxxx** and process B loads the same X.DLL at address yyyy, then the PTE for **xxxx** in process As page table and the PTE **f**[or] yyyy [m] process B's page table point to the same physical page. Figure 4-2 shows two processes sharing a page via same page table entries. The DLL pages are marked as read-only so that a process inadvertently attempting to write to this area will not cause other processes to crash.

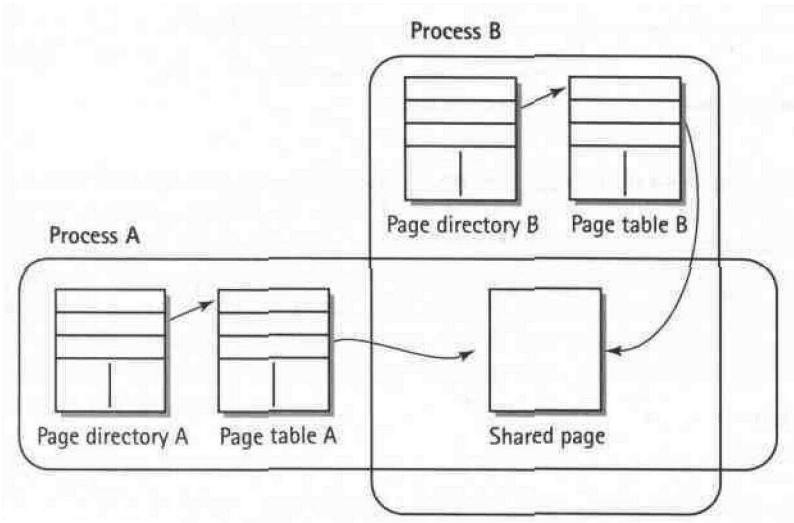

Figure 4-2: Sharing pages via same page table entries

This is guaranteed to be the case when xxxx==yyyy. However, if xxxx!=yyyy, the physical page might not be same. We will discuss the reason behind this later in the chapter.

Kernel address space is shared using a similar technique. Because the entire kernel space is common for all processes, Windows NT can share page tables directly. Figure 4-3 shows how processes share physical pages by using same page tables. Consequently, the upper half of the page table directory entries are the same for all processes.

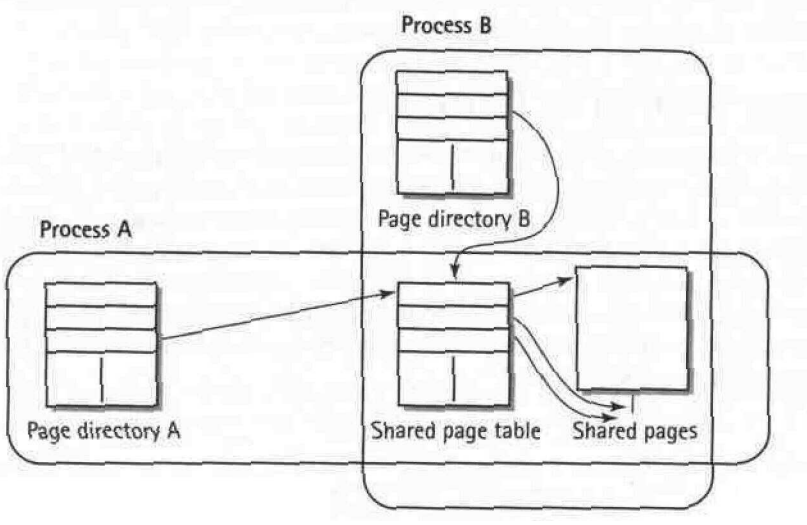

Figure 4-3: Sharing pages via same page directory entries

Listing 4-1 shows the sample program that demonstrates this.

Listing 4-1: SHOWDIR.C

```
/* Should be compiled in release mode to run properly */
#include <windows.h>
#include <string.h>

#include <stdio.h>
#include "gate.h"

/* Global array to hold the page directory */
DWORD PageDirectory[1024];
```

This initial portion of the SHOWDIR.C file contains, apart from the header inclusion, the global definition for the array to hold the page directory. The inclusion of the header file GATE.H is of interest. This header file prototypes the functions for using the callgate mechanism. Using the callgate mechanism, you can execute your code in the kernel mode without writing a new device driver.

We discuss the callgate mechanism **in** Chapter **10.**

For this sample program, we need this mechanism because the page directory is not accessible to the user-mode code. For now, it's sufficient to know that the mechanism allows a function inside a normal executable to be executed in kernel mode. Turning on to the definition of the page directory, we have already described that the size of each directory entry is 4 bytes and a page directory contains 1024 entries. Hence, the PageDirectory is an array of 1024 DWORDs. Each DWORD in the array represents the corresponding directory entry.

```
/* C function called from the assembly stub */
void _stdcall CFuncGetPageDirectory()
{
DWORD *PageDir=(DWORD *)0xC0300000;
int i=0;

for (i=0; i<1024; i++) {
PageDirectory[i] = PageDir[i];
}
```

CfuncGetPageDirectoryO is the function that is executed in the kernel mode using the callgate mechanism. This function simply makes a copy of the page directory in the user-mode memory area so that the other user-mode code parts in the program can access it. The page directory is mapped at virtual address OxC0300000 in every process's address space. This address is not accessible from the user mode. The CFuncGetPageDirectoryO function copies 1024 DWORDs from the OxC0300000 address to the global PageDirectory variable that is accessible to the user-mode code in the program.

```
/* Displays the contents of page directory. Starting
 * virtual address represented by the page directory
 * entry is shown followed by the physical page
 * address of the page table
 */
void DisplayPageDirectory()
{
int i;
int ctr=0;
```

```
printf("Page directory for the process, pid=%x\n'
GetCurrentProcessId());
for (i=0; i<1024; i++) {
if (PageDirectory[i]&0x01) {
if ((ctr%3)==0) {
printf("\n");
}
printf("%08x:%08x ", i << 22,
PageDirectory[i] & 0xFFFFF000);
ctr++;
}
}
printf("\n");
}
```

The DisplayPageDirectory() function operates in user mode and prints the PageDirectory array that is initialized by the CfuncGetPageDirectoryO function. The function checks the Least Significant Bit (LSB) of each of the entries. A page directory entry is valid only if the last bit or the LSB is set. The function skips printing invalid entries. The function prints three entries on every line or, in other words, prints a newline character for every third entry. Each directory entry is printed as the logical address and the address of the corresponding page table as obtained from the page directory. As described earlier, the first 10 bits (or the 10 Most Significant Bits [MSB]) of the logical address are used as an index in the page directory. In other words, a directory entry at index i represents the logical addresses that have i as the first 10 bits. The function prints the base of the logical address range for each directory entry. The base address (that is, the least address in the range) has the last 22 bits (or 22 LSBs) as zeros. The function obtains this base address by shifting i to the first 10 bits. The address of the page table corresponding to the logical address is stored in the first 20 bits (or 20 MSBs) of the page directory entry. The 12 LSBs are the flags for the entry. The function calculates the page table address by masking off the flag bits.

```
main()
{
WORD CallGateSelector;
int rc;
static short farcall[3];

/* Assembly stub that is called through callgate */
extern void GetPageDirectory(void);

/* Creates a callgate to read the page directory
 * from Ring 3 */
```

```
rc = CreateCallGate(GetPageDirectory, 0,
&CallGateSelector);

if (rc == SUCCESS) {
farcall[2] = CallGateSelector;

_asm {
call fword ptr [farcall]
}

DisplayPageDirectory();
getchar();

/* Releases the callgate */
rc=FreeCallGate(CallGateSelector);
if (rc!=SUCCESS) {
printf("FreeCallGate failed, "
"CallGateSelector=%x, rc=%x\n",
                                    CallGateSelector, rc);
}

} else {
printf("CreateCallGate failed, rc=%x\n", rc);
}
return 0;
}
```

The main() function starts by creating a callgate that sets up the GetPageDirectoryO function to be executed in the kernel mode. The GetPageDirectoryO function is written in Assembly language and is a part of the RINGO.ASM file. The CreateCallGate() function, used by the program to create the callgate, is provided by CALLGATE.DLL. The function returns with a callgate selector.

The mechanism of calling the desired function through callgate is explained in Chapter 10.

We'll quickly mention a few important points here. The callgate selector returned by CreateCallGate() is a segment selector for the given function: in this case, GetPageDirectoryO. To invoke the function pointed by the callgate selector, you need to issue a far call instruction. The far call instruction expects a 16-bit segment selector and a 32-bit offset within the segment. When you are calling through a

callgate, the offset does not matter; the processor always jumps at the start of the function pointed to by the callgate. Hence, the program only initializes the third member of the farcall array that corresponds to the segment selector. Issuing a call through the callgate transfers the execution control to the GetPageDirectoryO function. This function calls the CfuncGetPageDirectoryO function that copies the page directory in the PageDirectory array. After the callgate call returns, the program prints the page directory copied in the PageDirectory by calling the DisplayPageDirectoryO function. The program frees the callgate before exiting.

Listing 4-2: RINGO.ASM

```
.386
.model small

.code

include ..\include\undocnt.inc

public _GetPageDirectory
extrn _CFuncGetPageDirectory@0:near

;Assembly stub called from callgate
_GetPageDirectory proc
Ring0Prolog
call _CFuncGetPageDirectory@0
Ring0Epilog
retf
_GetPageDirectory endp

END
```

The function to be called from the callgate needs to be written in assembly language for a couple of reasons. First, the function needs to execute a prolog and an epilog, both of which are assembly macros, to allow paging in kernel mode. Second, the function needs to issue a far return at the end. The function leaves the rest of the job to the CFuncGetPageDirectoryO function written in C.

If you compare the output of the showdir program for two different processes, you find that the upper half of the page table directories for the two processes is exactly the same except for two entries. In other words, the corresponding kernel address space for these two entries is not shared by the two processes.

Listing 4-3: First instance of SHOWDIR

```
Page directory for the process, pid=6f
```

```
00000000:01026000 00400000:00f65000 10000000:0152f000
5f800000:00e46000 77c00000:0076b000 7f400000:012cb000
7fc00000:0007e000 80000000:00000000 80400000:00400000
80800000:00800000 80c00000:00c00000 81000000:01000000
81400000:01400000 81800000:01800000 81c00000:01c00000
82000000:02000000 82400000:02400000 82800000:02800000
82c00000:02c00000 83000000:03000000 83400000:03400000
83800000:03800000 83c00000:03c00000 84000000:04000000
84400000:04400000 84800000:04800000 84c00000:04c00000
85000000:05000000 85400000:05400000 85800000:05800000
85c00000:05c00000 86000000:06000000 86400000:06400000
86800000:06800000 86c00000:06c00000 87000000:07000000
87400000:07400000 87800000:07800000 87c00000:07c00000
a0000000:0153d000 c0000000:00e5d000 c0400000:00c9e000
c0c00000:00041000 c1000000:00042000 c1400000:00043000
c1800000:00044000 c1c00000:00045000 c2000000:00046000
c2400000:00047000 c2800000:00048000 c2c00000:00049000
c3000000:0004a000 c3400000:0004b000 c3800000:0004c000
c3c00000:0004d000 c4000000:0004e000 c4400000:0000f000
c4800000:00050000 c4c00000:00051000 c5000000:00052000
c5400000:00053000 c5800000:00054000 c5c00000:00055000
c6000000:00056000 c6400000:00057000 c6800000:00058000
c6c00000:00059000 c7000000:0005a000 c7400000:0005b000
c7800000:0005c000 c7c00000:0005d000 c8000000:0005e000
c8400000:0005f000 c8800000:00020000 c8c00000:00021000
c9000000:00022000 c9400000:00023000 c9800000:00024000
c9c00000:00025000 ca000000:00026000 ca400000:00027000
ca800000:00028000 cac00000:00029000 cb000000:0002a000
cb400000:0002b000 cb800000:0002c000 cbc00000:0002d000
cc000000:0002e000 cc400000:0002f000 cc800000:002f0000
ccc00000:002f1000 cd000000:002f2000 cd400000:002f3000
cd800000:002f4000 cdc00000:002f5000 ce000000:002f6000
ce400000:00037000 ce800000:00038000 cec00000:00039000
cf000000:0003a000 cf400000:0003b000 cf800000:0003c000
cfc00000:0003d000 d0000000:0003e000 d0400000:0003f000
d0800000:00380000 d0c00000:00301000 d1000000:00302000
d1400000:00303000 d1800000:00304000 d1c00000:00305000
d2000000:00306000 d2400000:00307000 d2800000:00308000
d2c00000:00309000 d3000000:0030a000 d3400000:0030b000
d3800000:0030c000 d3c00000:0030d000 d4000000:0030e000
d4400000:0004f000 d4800000:00310000 d4c00000:00311000
e1000000:00315000 e1400000:010fe000 fc400000:0038d000
fc800000:0038e000 fcc00000:0038f000 fd000000:00390000
fd400000:00391000 fd800000:00392000 fdc00000:00393000
```

```
fe000000:00394000 fe400000:00395000 fe800000:00396000
fec00000:00397000 ff000000:00398000 ff400000:00399000
ff800000:0039a000 ffc00000:00031000
```

Listing 4-4: Second instance of SHOWDIR

```
Page directory for the process, pid=7d

00000000:00fa1000 00400000:00fa0000 10000000:0110a000
5f800000:015ac000 77c00000:01a73000 7f400000:013ac000
7fc00000:0145e000 80000000:00000000 80400000:00400000
80800000:00800000 80c00000:00c00000 81000000:01000000
81400000:01400000 81800000:01800000 81c00000:01c00000
82000000:02000000 82400000:02400000 82800000:02800000
82c00000:02c00000 83000000:03000000 83400000:03400000
83800000:03800000 83c00000:03c00000 84000000:04000000
84400000:04400000 84800000:04800000 84c00000:04c00000
85000000:05000000 85400000:05400000 85800000:05800000
85c00000:05c00000 86000000:06000000 86400000:06400000
86800000:06800000 86c00000:06c00000 87000000:07000000
87400000:07400000 87800000:07800000 87c00000:07c00000
a0000000:0153d000 c0000000:00d94000 c0400000:01615000
c0c00000:00041000 c1000000:00042000 c1400000:00043000
c1800000:00044000 c1c00000:00045000 c2000000:00046000
c2400000:00047000 c2800000:00048000 c2c00000:00049000
c3000000:0004a000 c3400000:0004b000 c3800000:0004c000
c3c00000:0004d000 c4000000:0004e000 c4400000:0000f000
c4800000:00050000 c4c00000:00051000 c5000000:00052000
c5400000:00053000 c5800000:00054000 c5c00000:00055000
c6000000:00056000 c6400000:00057000 c6800000:00058000
c6c00000:00059000 c7000000:0005a000 c7400000:0005b000
c7800000:0005c000 c7c00000:0005d000 c8000000:0005e000
c8400000:0005f000 c8800000:00020000 c8c00000:00021000
c9000000:00022000 c9400000:00023000 c9800000:00024000
c9c00000:00025000 ca000000:00026000 ca400000:00027000
ca800000:00028000 cac00000:00029000 cb000000:0002a000
cb400000:0002b000 cb800000:0002c000 cbc00000:0002d000
cc000000:0002e000 cc400000:0002f000 cc800000:002f0000
ccc00000:002f1000 cd000000:002f2000 cd400000:002f3000
cd800000:002f4000 cdc00000:002f5000 ce000000:002f6000
ce400000:00037000 ce800000:00038000 cec00000:00039000
cf000000:0003a000 cf400000:0003b000 cf800000:0003c000
cfc00000:0003d000 d0000000:0003e000 d0400000:0003f000
d0800000:00380000 d0c00000:00301000 d1000000:00302000
```

```
d1400000:00303000 d1800000:00304000 d1c00000:00305000
d2000000:00306000 d2400000:00307000 d2800000:00308000
d2c00000:00309000 d3000000:0030a000 d3400000:0030b000
d3800000:0030c000 d3c00000:0030d000 d4000000:0030e000
d4400000:0004f000 d4800000:00310000 d4c00000:00311000
e1000000:00315000 e1400000:010fe000 fc400000:0038d000
fc800000:0038e000 fcc00000:0038f000 fd000000:00390000
fd400000:00391000 fd800000:00392000 fdc00000:00393000
fe000000:00394000 fe400000:00395000 fe800000:00396000
fec00000:00397000 ff000000:00398000 ff400000:00399000
ff800000:0039a000 ffc00000:00031000
```

Let's analyze, one step at a time, why the two entries are different. The page tables themselves need to be mapped onto some linear address. When Windows NT needs to access the page tables, it uses this linear address range. To represent 4GB of memory divided into 1MB pages of 4K each, we need IK page tables each having IK entries. To map these IK page tables, Windows NT reserves 4MB of linear address space in each process. As we saw earlier, each process has a different set of page tables. Whatever the process, Windows NT maps the page tables on the linear address range from 0xC0000000 to OxC03FFFFF. Let's call this linear address range as the page table address range. In other words, the page table address range maps to different page tables-that is, to different physical pages-for different processes. As you may have noticed, the page table addresses range falls in the kernel address space. Windows NT cannot map this crucial system data structure in the user address space and allow user-mode processes to play with the memory. Ultimately, the result is that two processes cannot share pages in the page table address range although the addresses lie in the kernel-mode address range.

Exactly one page table is required to map 4MB address space because each page table has IK entries and each entry corresponds to a 4K page. Consequently, Windows NT cannot share the page table corresponding to the page table address range. This accounts for one of the two mysterious entries in the page table directory. However, the entry's mystery does not end here-there is one more subtle twist to this story. The physical address specified in this entry matches the physical address of the page table directory. The obvious conclusion is that the page table directory acts also as the page table for the page table address range. This is possible because the formats of the page table directory entry and PTE are the same on 80386.

The processor carries out an interesting sequence of actions when the linear address within the page table address range is translated to a physical address. Let's say that the CR3 register points to page X. As the first step in the address translation process, the processor treats the page X as the page table directory and finds out the page table for the given linear address. The page table happens to be page X again. The processor now treats page X as the required page table and finds out

the physical address from it. A more interesting case occurs when the operating system is accessing the page table directory itself. In this case, the physical address also falls in page X!

Let's now turn to the second mysterious entry. The 4MB area covered by this page directory entry is internally referred to as *hyperspace.* This area is used for mapping the physical pages belonging to other processes into virtual address space. For example, a function such as MmMapPageInHyperspace() uses the virtual addresses in this range. This area is also used during the early stages of process creation. For example, when a parent process such as PROGMAN.EXE spawns a child process such as NOTEPAD.EXE, PROGMAN.EXE has to create the address space for NOTEPAD.EXE. This is done as a part of the MmCreateProcessAddressSpace() function. For starting any process, an address space must be created for the process. Address space is nothing but page directory. Also, the upper-half entries of page directory are common for all processes except for the two entries that we have already discussed. These entries need to be created for the process being spawned. The MmCreateProcessAddressSpace() function allocates three pages of memory: the first page for the page directory, the second page for holding the hyperspace page table entries, and the third page for holding the working set information for the process being spawned.

Once these pages are allocated, the function maps the first physical page in the address space using the MmMapPagelnHyperSpacefJ function. Note that the MmMapPagelnHyperSpaceQ function runs in the context of PROGMAN.EXE. Now the function copies the page directory entries in the upper half of the page directory to the mapped hyperspace virtual address. In short, PROGMAN.EXE creates the page directory for the NOTEPAD.EXE.

Windows NT supports memory-mapped files. When two processes map the same file, they share the same set of physical pages. Hence, memory-mapped files can be used for sharing memory. In fact, Windows NT itself uses memory-mapped files to load DLLs and executables. If two processes map the same DLL, they automatically share the DLL pages. The memory-mapped files are implemented using the section object under Windows NT. A data structure called PROTOPTE is associated with each section object. This data structure is a variable-length structure based on the size of the section. This data structure contains a 4-byte entry for each page in the virtual address space mapped by the section object. Each 4-byte entry has the same structure as that of the PTE. When the page is not being used by any of the processes, the protopte entry is invalid and contains enough information to get the page back. In this case, the CPU PTE contains a fixed value that is OxFFFFF480, which indicates that accessing this page will be considered a protopte fault.

Now comes the toughest of all questions: "How can Windows NT give away 4GB of memory to each process when there is far less physical RAM available on the board?" Windows NT, as well as all other operating systems that allow more address space than actual physical memory, uses a technique called *virtual memory* to achieve this. In the next section, we discuss virtual memory management in Windows NT.

Virtual Memory Management

The basic idea behind virtual memory is very simple. For each process, the operating system maps few addresses to real physical memory because RAM is expensive and relatively rare. Remaining memory for each process is really maintained on secondary storage (usually a hard disk). That's why it is called virtual memory. The addresses that are not mapped on physical RAM are marked as such. Whenever a process accesses such an address, the operating system brings the data into memory from secondary storage. If the operating system runs out of physical RAM, some data is thrown out to make space. We can always get back this data because a copy is maintained on secondary storage. The data to be thrown out is decided by the replacement policy. Windows NT uses First-In-First-Out (FIFO) replacement policy. According to this policy, the oldest data (that is, the data that was brought in the RAM first) is thrown out whenever there is a space crunch.

To implement virtual memory management, Windows NT needs to maintain a lot of data. First, it needs to maintain whether each address is mapped to physical RAM or the data is to be brought in from secondary storage when a request with the address comes. Maintaining this information for each byte itself takes a lot of space (actually, more space than the address space for which the information is to be maintained). So Windows NT breaks the address space into 4KB pages and maintains this information in page tables. As we saw earlier, a page table entry (PTE) consists of the address of the physical page (if the page is mapped to physical RAM) and attributes of the page. Since the processor heavily depends on PTEs for address translation, the structure of PTE is processor dependent.

If a page is not mapped onto physical RAM, Windows NT marks the page as invalid. Any access to this page causes a page fault, and the page fault handler can bring in the page from the secondary storage. To be more specific, when the page contains DLL code or executable module code, the page is brought in from the DLL or executable file. When the page contains data, it is brought in from the swap file. When the page represents a memory-mapped file area, it is brought in from the corresponding file. Windows NT needs to keep track of free physical RAM so that it can allocate space for a page brought in from secondary storage in case of a page fault. This information is maintained in a kernel data structure called the Page Frame Database (PFD). The PFD also maintains a FIFO list of in-memory pages so that it can decide on pages to throw out in case of a space crunch.

Before throwing out a page, Windows NT must ensure that the page is not dirty. Otherwise, it needs to write that page to secondary storage before throwing it out. If the page is not shared, the PFD contains the pointer to PTE so that if the operating system decides to throw out a particular page, it can then go back and mark the PTE as invalid. If the page is shared, the PFD contains a pointer to the corresponding PROTOPTE entry. In this case, the PFD also contains a reference count for the page. A page can be thrown out only if its reference count is 0. In general, the PFD maintains the status of every physical page.

The PFD is an array of 24-byte entries, one for each physical page. Hence, the size of this array is equal to the number of physical pages that are stored in a kernel variable, namely, MmNumberOfPhysicalPages. The pointer to this array is stored in a kernel variable, namely, MmpfnDatabase. A physical page can be in several states - for example, it can be in-use, free, free but dirty, and so on. A PFD entry is linked in a doubly linked list, depending on the state of the physical page represented by it. For example, the PFD entry representing a free page is linked in the free pages list. Figure 4-4 shows these lists linked through the PFD. The forward links are shown on the left side of the PFD, and the backward links are shown on the right side.

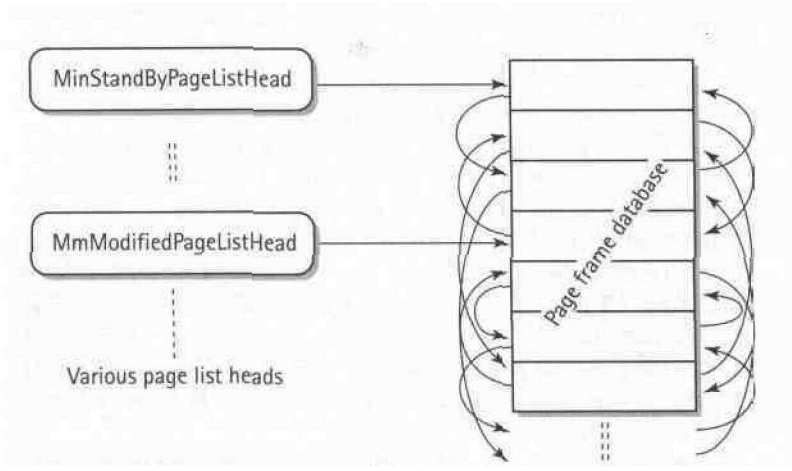

Figure 4-4: Various lists linked through PFD

There are in all six kinds of lists. The heads of these lists are stored in following kernel variables:

if

MmStandbyPageListHead

MmModifiedNoWritePageListHead

MmModifiedPageListHead

MmFreePageListHead

MmBadPageListHead

MmZeroedPageListHead

All these list heads are actually structures of 16 bytes each. Here is the structure definition:

```
typedef struct PageListHead {
        DWORD NumberOfPagesInList,
```

```
        DWORD TypeOfList,
        DWORD FirstPage,
        DWORD LastPage
} PageListHead_t;
```

The FirstPage field can be used as an index into the PFD. The PFD entry contains a pointer to the next page. Using this, you can traverse any of the lists. Here is the structure definition for the PFD entry:

```
typedef struct PfdEntry {
        DWORD NextPage,
        void *PteEntry/*PpteEntry,
        DWORD PrevPage,
        DWORD PteReferenceCount,
        void *OriginalPte,
        DWORD Flags;
} PfdEntry_t;
```

Using this, you can easily write a program to dump the PFD. However, there is one problem: kernel variables, such as list heads, MmPfnDatabase, and MmNumberOfPhysicalPages, are not exported. Therefore, you have to deal with absolute addresses, which makes the program dependent on the Windows NT version and build type.

Virtual Address Descriptors

Along with the free physical pages, Windows NT also needs to keep track of the virtual address space allocation for each process. Whenever a process allocates a memory block - for example, to load a DLL - Windows NT checks for a free block in the virtual address space, allocates virtual address space, and updates the virtual address map accordingly. The most obvious place to maintain this information is page tables. For each process, Windows NT maintains separate page tables. There are 1 million pages, and each page table entry is 4 bytes. Hence, full page tables for a single process would take 4MB of RAM! There is a solution to this: Page tables themselves can be swapped out. It is inefficient to swap in entire page tables when a process wants to allocate memory. Hence, Windows NT maintains a separate binary search tree containing the information about current virtual space allocation for each process. A node in this binary search tree is called a Virtual Address Descriptor (VAD). For each block of memory allocated to a process, Windows NT adds a VAD entry to the binary search tree. Each VAD entry contains the allocated address range - that is, the start address and the end address of the allocated block, pointers to left and right children VADs, and a pointer to the parent VAD. The process environment block (PEB) contains a pointer, namely, VadRoot, to the root of this tree.

Listing 4-5: VADDUMP.C

```
/* Should be compiled in release mode */
#define _X86_

#include <ntddk.h>
#include <string.h>
#include <stdio.h>

#include "undocnt.h"
#include "gate.h"

/*Define the WIN32 calls we are using, since we can not include both
NTDDK.H and
WINDOWS.H in the same 'C' file.*/
typedef struct _OSVERSIONINFO{
    ULONG dwOSVersionInfoSize;
    ULONG dwMajorVersion;
    ULONG dwMinorVersion;
    ULONG dwBuildNumber;
    ULONG dwPlatformId;
    CCHAR szCSDVersion[ 128 ];
} OSVERSIONINFO, *LPOSVERSIONINFO;
BOOLEAN _stdcall GetVersionExA(LPOSVERSIONINFO);
PVOID _stdcall VirtualAlloc(PVOID, ULONG, ULONG, ULONG);

/* Max vad entries */
#define MAX_VAD_ENTRIES             0x200

/* Following variables are accessed in RINGO.ASM */
ULONG NtVersion;
ULONG PebOffset;
ULONG VadRootOffset;

#pragma pack(1)
typedef struct VadInfo {
       void *VadLocation;
       VAD Vad;
} VADINFO, *PVADINFO;
#pragma pack()

VADINFO VadInfoArray[MAX_VAD_ENTRIES];
int VadInfoArrayIndex;
PVAD VadTreeRoot;
```

The initial portion of the VADDUMP.C file has a few definitions apart from the header inclusion. In this program, we use the callgate mechanism as we did in the showdir program - hence the inclusion of the GATE.H header file. After the header inclusion, the file defines the maximum number of VAD entries that we'll process. There is no limit on the nodes in a VAD tree. We use the callgate mechanism for kernel-mode execution of a function that dumps the VAD tree in an array accessible from the user mode. This array can hold up to MAX_VAD_ENTRIES entries. Each entry in the array is of type VADINFO. The VADINFO structure has two members: the address of the VAD tree node and the actual VAD tree node. The VAD tree node structure is defined in the UNDOCNT.H file as follows:

```
typedef struct vad {
void *StartingAddress;
void *EndingAddress;
struct vad *ParentLink;
struct vad *LeftLink;
struct vad *RightLink;
DWORD Flags;
}VAD, *PVAD;
```

The first two members dictate the address range represented by the VAD node. Each VAD tree node maintains a pointer to the parent node and a pointer to the left child and the right child. The VAD tree is a binary tree. For every node in the tree, the left subtree consists of nodes representing lower address ranges, and the right subtree consists of nodes representing the higher address ranges. The last member in the VAD node is the flags for the address range.

The VADDUMP.C file has a few other global variables apart from the VadlnfoArray. A couple of global variables are used while locating the root of the VAD tree. The PEB of a process points to the VAD tree root for that process. The offset of this pointer inside the PEB varies with the Windows NT version. We set the VadRootOffset to the appropriate offset value of the VAD root pointer depending on the Windows NT version. There is a similar problem of Windows NT version dependency while accessing the PEB for the process. We use the Thread Environment Block (TEB) to get to the PEB. One field in TEB points to the PEB, but the offset of this field inside the TEB structure varies with the Windows NT version. We set the PebOffset variable to the appropriate offset value of the PEB pointer inside the TEB structure depending on the Windows NT version. Another global variable, NtVersion, stores the version of Windows NT running on the machine.

That leaves us with two more global variables, namely, VadlnfoArraylndex and VadTreeRoot. The VadlnfoArraylndex is the number of initialized entries in the VadlnfoArray. The VadlnfoArray entries after VadlnfoArraylndex are free. The VadTreeRoot variable stores the root of the VAD tree.

The sample has been tested on Windows NT 3.51, 4.0 and Windows 2000 beta2. The sample will run on other versions of Windows 2000, provided the offsets of VadRoot and PEB remain same.

```
/* Recursive function which walks the vad tree and
 * fills up the global VadInfoArray with the Vad
 * entries. Function is limited by the
 * MAX_VAD_ENTRIES. Other VADs after this are not
 * stored
 */
void _stdcall VadTreeWalk(PVAD VadNode)
{
if (VadNode == NULL) {
return;
}
if (VadInfoArrayIndex >= MAX_VAD_ENTRIES) {
return;
}

VadTreeWalk(VadNode->LeftLink);

        VadInfoArray[VadInfoArrayIndex].VadLocation = VadNode;
VadInfoArray[VadInfoArrayIndex].Vad.StartingAddress =
VadNode->StartingAddress;
VadInfoArray[VadInfoArrayIndex].Vad.EndingAddress =
VadNode->EndingAddress;

if (NtVersion == 5) {
(DWORD)VadInfoArray[VadInfoArrayIndex].
Vad.StartingAddress <<= 12;
(DWORD)VadInfoArray[VadInfoArrayIndex].
Vad.EndingAddress += 1;
(DWORD)VadInfoArray[VadInfoArrayIndex].
Vad.EndingAddress <<= 12;
(DWORD)VadInfoArray[VadInfoArrayIndex].
Vad.EndingAddress -= 1;
}

VadInfoArray[VadInfoArrayIndex].Vad.ParentLink =
VadNode->ParentLink;
VadInfoArray[VadInfoArrayIndex].Vad.LeftLink =
VadNode->LeftLink;
VadInfoArray[VadInfoArrayIndex].Vad.RightLink =
VadNode->RightLink;
```

```
VadInfoArray[VadInfoArrayIndex].Vad.Flags =
VadNode->Flags;
VadInfoArrayIndex++;

        VadTreeWalk(VadNode->RightLink);
}
```

The VadTreeWalk() function is executed in the kernel mode using the callgate mechanism. The function traverses the VAD tree in the in-order fashion and fills up the VadlnfoArray. The function simply returns if the node pointer parameter is NULL or the VadlnfoArray is full. Otherwise, the function recursively calls itself for the left subtree. The recursion is terminated when the left child pointer is NULL. The function then fills up the next free entry in the VadlnfoArray and increments the VadlnfoArraylndex to point to the next free entry. Windows **2000** stores the page numbers instead of the actual addresses in VAD. Hence, for Windows **2000,** we need to calculate the starting address and the ending address from the page numbers stored in these fields. As the last step in the in-order traversal, the function issues **a** self-recursive to process the right subtree.

```
/* C function called through assembly stub */
void _stdcall CFuncDumpVad(PVAD VadRoot)
{
VadTreeRoot = VadRoot;
VadlnfoArrayIndex = 0;
VadTreeWalk(VadRoot);
```

The CfuncDumpVad is the caller of the VadTreeWalkO function. It just initializes the global variables used by the VadTreeWalkO function and calls the VadTreeWalkO function for the root of the VAD tree.

```
/* Displays the Vad tree */
void VadTreeDisplay()
(
int i;

printf("VadRoot is located @%08x\n\n",
VadTreeRoot);
printf("Vad@\t StartingU EndingU ParentU "
"LeftLinkU RightLink\n");
for (i=0; i < VadlnfoArrayIndex; i++) {
printf("%08x %08x %08x %8x %08x %08x\n",
VadInfoArray[i].VadLocation,
VadlnfoArray[i].Vad.StartingAddress,
```

```
VadInfoArray[i].Vad.EndingAddress,
VadInfoArray[i].Vad.ParentLink,
VadInfoArray[i].Vad.LeftLink,
VadInfoArray[i].Vad.RightLink);
}
printf("\n\n");
```

The VadTreeDisplayO function is a very simple function that is executed in user mode. The function iterates through all the entries initialized by the VadTreeWalk() function and prints the entries. Essentially, the function prints the VAD tree in the infix order because the VadTreeWalk() function dumps the VAD tree in the infix order.

```
void SetDataStructureOffsets()
(
switch (NtVersion) {
case 3:
PebOffset = 0x40;
VadRootOffset = 0x170;
break;

case 4:
PebOffset = 0x44;
VadRootOffset = 0x170;
break;

case 5:
PebOffset = 0x44;
VadRootOffset = 0x194;
break;
)
}
```

As we described earlier, the offset of the PEB pointer within TEB and the offset of the VAD root pointer within the PEB are dependent on the Windows NT version. The SetDataStructureOffsetsO function sets the global variables indicating these offsets depending on the Windows NT version.

```
main()

WORD CallGateSelector;
int re;
short farcall[3];
void DumpVad(void);
```

```
void *ptr;
OSVERSIONINFO VersionInfo;

VersionInfo.dwOSVersionInfoSize = sizeof(VersionInfo);
if (GetVersionEx(&VersionInfo) == TRUE) {
NtVersion = VersionInfo.dwMajorVersion;
}

if ((NtVersion < 3)||(NtVersion > 5)) {
printf("Unsupported NT version, exiting...");
return 0;
}
SetDataStructureOffsets();

/* Creates call gate to read vad tree from Ring 3
 */
rc = CreateCallGate(DumpVad, 0, &CallGateSelector);
if (rc != SUCCESS) {
printf("CreateCallGate failed, rc=%x\n", rc);
return 1;
}

farcall[2] = CallGateSelector;

_asm {
call fword ptr [farcall]
}

printf("Dumping the Vad tree ...\n\n");
VadTreeDisplay();

printf("Allocating memory using VirtualAlloc");

ptr = VirtualAlloc(NULL, 4096, MEM_COMMIT,
PAGE_READONLY);
if (ptr == NULL) {
printf("Unable to allocate memory\n");
goto Quit;
}
printf("\nMemory allocated @%x\n", ptr);

_asm {
call fword ptr [farcall]
```

```
printf("\n\nDumping the Vad tree again...\n\n");
VadTreeDisplay();

Quit:
re = FreeCallGate(Cal1GateSelector);
if (re != SUCCESS) {
printf("FreeCal1Gate failed, Selector=%x, rc=%x\n"
                              CallGateSelector, re);
}

       return 0;
}
```

The main() function starts by getting the Windows NT version and calling SetDataStructureOffsetsO to set the global variables storing the offsets for the PEB and the VAD tree root. It then creates a callgate in the same manner as in the SHOWDIR sample program. Issuing a call through this callgate ultimately results in the execution of the VadTreeWalk() function that fills up the VadlnfoArray. The main() function then calls the VadTreeDisplay() function to print the VadlnfoArray entries.

We also show you the change in the VAD tree due to memory allocation in this sample program. After printing the VAD tree once, the program allocates a chunk of memory. Then, the program issues the callgate call again and prints the VAD tree after returning from the call. You can observe the updates that happened to the VAD tree because of the memory allocation. The program frees up the callgate before exiting.

Listing **4-6:** RINGO.ASM

```
.386
.model small

.code

public _DumpVad
extrn _CFuncDumpVad@4:near
extrn _PebOffset:near
extrn _VadRootOffset:near

include ..\include\undocnt.inc

_DumpVad proc
RingOProlog

;Gets the current thread
```

```
MOV     EAX,FS:[00000124h]

;Gets the current process
ADD     EAX, DWORD PTR [_PebOffset]
MOV     EAX,[EAX]

;Push Vad Tree root
ADD     EAX, DWORD PTR [_VadRootOffset]
MOV     EAX, [EAX]
PUSH    EAX
CALL    _CFuncDumpVad@4

RingOEpilog
RETF
_DumpVad endp

END
```

The function to be called from the callgate needs to be written in the Assembly language for reasons already described. The DumpVadO function gets hold of the VAD root pointer and calls the CFuncDumpVadO function that dumps the VAD tree in the VadlnfoArray. The function gets hold of the VAD root from the PEB after getting hold of the PEB from the TEB. The TEB of the currently executing thread is always pointed to by FS:128h. As described earlier, the offset of the VAD root pointer inside PEB and the offset of the PEB pointer inside the TEB vary with the Windows NT version. The DumpVad() function uses the offset values stored in the global variable by the SetDataStructureOffsets() function.

Listing 4-7 presents the output from an invocation of the VADDUMP program. Note that the VAD tree printed after allocating memory at address 0x300000 shows an additional entry for that address range.

Listing **4-7:** Program output

```
Dumping the Vad tree...

VadRoot is located @fe21a9c8

Vad@      Starting  Ending    Parent    LeftLink  RightLink
fe216b08  00010000  00010fff  fe21a9c8  00000000  fe25a0e8
fe25a0e8  00020000  00020fff  fe216b08  00000000  fe275da8
fe275da8  00030000  0012ffff  fe25a0e8  00000000  fe22a428
fe22a428  00130000  00130fff  fe275da8  00000000  fe26b328
fe26b328  00140000  0023ffff  fe22a428  00000000  fe210fc8
fe210fc8  00240000  0024ffff  fe26b328  00000000  fe21a8c8
fe21a8c8  00250000  00258fff  fe210fc8  00000000  fe21be68
```

fe21be68	00260000	0026dfff	fe21a8c8	00000000	fe215dc8
fe215dc8	00270000	002b0fff	fe21be68	00000000	fe231e88
fe231e88	002c0000	002c0fff	fe215dc8	00000000	fe2449e8
fe2449e8	002d0000	002dffff	fe231e88	00000000	fe21cb48
fe21cb48	002e0000	002e0fff	fe2449e8	00000000	fe23b7a8
fe23b7a8	002f0000	002fffff	fe21cb48	00000000	00000000
fe21a9c8	00400000	0040cfff	0	fe216b08	fe23c488
fe21b3e8	10000000	1000dfff	fe2333e8	00000000	fe226348
fe2176c8	77e20000	77e4bfff	fe226348	00000000	fe2326e8
fe2152c8	77e50000	77e54fff	fe2326e8	00000000	00000000
fe2326e8	77e60000	77e9bfff	fe2176c8	fe2152c8	00000000
fe226348	77ea0000	77ed7fff	fe21b3e8	fe2176c8	fe2197c8
fe2197c8	77ee0000	77f12fff	fe226348	00000000	00000000
fe2333e8	77f20000	77f73fff	fe23c488	fe21b3e8	00000000
fe23c488	77f80000	77fcdfff	fe21a9c8	fe2333e8	fe25aa88
fe22b408	7f2d0000	7f5cffff	fe25aa88	00000000	fe22c4a8
fe22c4a8	7f5f0000	7f7effff	fe22b408	00000000	fe23f5e8
fe23f5e8	7ff70000	7ffaffff	fe22c4a8	00000000	00000000
fe25aa88	7ffb0000	7ffd3fff	fe23c488	fe22b408	fe218288
fe21da88	7ffde000	7ffdefff	fe218288	00000000	00000000
fe218288	7ffdf000	7ffdffff	fe25aa88	fe21da88	00000000

```
Allocating memory using VirtualAlloc
Memory allocated @300000

Dumping the Vad tree again...

VadRoot is located @fe21a9c8
```

Vad@	Starting	Ending	Parent	LeftLink	RightLink
fe216b08	00010000	00010fff	fe21a9c8	00000000	fe25a0e8
fe25a0e8	00020000	00020fff	fe216b08	00000000	fe275da8
fe275da8	00030000	0012ffff	fe25a0e8	00000000	fe22a428
fe22a428	00130000	00130fff	fe275da8	00000000	fe26b328
fe26b328	00140000	0023ffff	fe22a428	00000000	fe210fc8
fe210fc8	00240000	0024ffff	fe26b328	00000000	fe21a8c8
fe21a8c8	00250000	00258fff	fe210fc8	00000000	fe21be68
fe21be68	00260000	0026dfff	fe21a8c8	00000000	fe215dc8
fe215dc8	00270000	002b0fff	fe21be68	00000000	fe231e88
fe231e88	002c0000	002c0fff	fe215dc8	00000000	fe2449e8
fe2449e8	002d0000	002dffff	fe231e88	00000000	fe21cb48
fe21cb48	002e0000	002e0fff	fe2449e8	00000000	fe23b7a8
fe23b7a8	002f0000	002fffff	fe21cb48	00000000	fe27b628
fe27b628	*00300000*	*00300fff*	*fe23b7a8*	*00000000*	*00000000*

fe21a9c8	00400000	0040cfff	0	fe216b08	fe23c488
fe21b3e8	10000000	l000dfff	fe2333e8	00000000	fe226348
fe2176c8	77e20000	77e4bfff	fe226348	00000000	fe2326e8
fe2152c8	77e50000	77e54fff	fe2326e8	00000000	00000000
fe2326e8	77e60000	77e9bfff	fe2176c8	fe2152c8	00000000
fe226348	77ea0000	77ed7fff	fe21b3e8	fe2176c8	fe2197c8
fe2197c8	77ee0000	77fl2fff	fe226348	00000000	00000000
fe2333e8	77f20000	77f73fff	fe23c488	fe21b3e8	00000000
fe23c488	77f80000	77fcdfff	fe21a9c8	fe2333e8	fe25aa88
fe22b408	7f2d0000	7f5cffff	fe25aa88	00000000	fe22c4a8
fe22c4a8	7f5f0000	7f7effff	fe22b408	00000000	fe23f5e8
fe23f5e8	7ff70000	7ffaffff	fe22c4a8	00000000	00000000
fe25aa88	7ffb0000	7ffd3fff	fe23c488	fe22b408	fe218288
fe21da88	7ffde000	7ffdefff	fe218288	00000000	00000000
fe218288	7ffdf000	7ffdffff	fe25aa88	fe21da88	00000000

The output of the VADDUMP program does not really look like a tree. You have to trace through the output to get the tree structure. The entry with a null parent link is the root of the tree. Once you find the root, you can follow the child pointers. To follow a child pointer, search the pointer in the first column, named Vad@, in the output. The Vad entry with the same Vad@ is the entry for the child that you are looking for. An all-zero entry for a left/right child pointer indicates that there is no left/right subtree for the node. Figure 4-5 shows a partial tree constructed from the output shown previously.

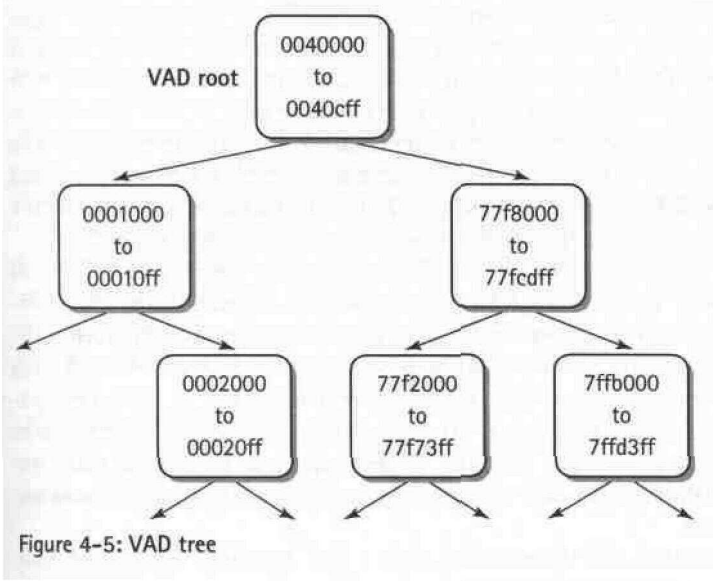

Figure 4-5: VAD tree

Impact on Hooking

Now we'll look at the impact of the memory management scheme explained in the last section in the area of hooking DLL API calls. To hook a function from a DLL, you need to change the first few bytes from the function code. As you saw earlier, the DLL code is shared by all processes and is write protected so that a misbehaving process cannot affect other processes. Does this mean that you cannot hook a function in Windows NT? The answer is, "Hooking is possible under Windows NT, but you need to do a bit more work to comply with stability requirements." Windows NT provides a system call, VirtualProtect, that you can use to change page attributes. Hence, hooking is now a two-step process: Change the attributes of the page containing DLL code to read-write, and then change the code bytes.

Copy-on-Write

"Eureka!" you might say, "I violated Windows NT security. I wrote to a shared page used by other processes also." No! You did not do that. You changed only your copy of the DLL code. The DLL code page was being shared while you did not write to the page. The moment you wrote on that page, a separate copy of it was made, and the writes went to this copy. All other processes are safely using the original copy of the page. This is how Windows NT protects processes from each other while consuming as few resources as possible.

The VirtualProtectO function does not mark the page as read-write - it keeps the page as read-only Nevertheless, to distinguish this page from normal read-only pages, it is marked for copy-on-write. Windows NT uses one of the available PTE bits for doing this. When this page is written onto, because it is a read-only page, the processor raises a page fault exception. The page fault handler makes a copy of the page and modifies the page table of the faulting process accordingly. The new copy is marked as read-write so that the process can write to it.

Windows NT itself uses the copy-on-write mechanism for various purposes. The DLL data pages are shared with the copy-on-write mark. Hence, whenever a process writes to a data page, it gets a personal copy of it. Other processes keep sharing the original copy, thus maximizing the sharing and improving memory usage.

A DLL may be loaded in memory at different linear address for different processes. The memory references - for example, address for call instruction, address for a memory to register move instruction, and so on — in the DLL need to be adjusted (patched) depending on the linear address where the DLL gets loaded. This process is called as relocating the DLL. Obviously, relocation has to be done separately for each process. While relocating, Windows NT marks the DLL code pages as copy-on-write temporarily. Thus, only the pages requiring page relocation are copied per process. Other pages that do not have memory references in them are shared by all processes.

This is the reason Microsoft recommends that a DLL be given a preferred base address and be loaded at that address. The binding of the DLL to a specific base ad-

dress ensures that the DLL need not be relocated if it is loaded at the specified base address. Hence, if all processes load the DLL at the preferred base address, all can share the same copy of DLL code.

The POSIX subsystem of Windows NT uses the copy-on-write mechanism to implement the fork system call. The fork system call creates a new process as a child of a calling process. The child process is a replica of the parent process, and it has the same state of code and data pages as the parent. Since these are two different processes, the data pages should not be shared by them. However, generally it is wasteful to make a copy of the parent's data pages because in most cases the child immediately invokes the exec system call. The exec system call discards the current memory image of the process, loads a new executable module, and starts executing the new executable module. To avoid copying the data pages, the fork system call marks the data pages as copy-on-write. Hence, a data page is copied only if the parent or the child writes to it.

Copy-on-write is an extremely important concept contributing to the efficiency of NT memory management.

The following sample program demonstrates how copy-on-write works. By running two instances of the program, you can see how the concepts described in this section work. The application loads a DLL, which contains two functions and two data variables. One function does not refer to the outside world, so no relocations are required for it. The other function accesses one global variable, so it contains relocatable instructions or instructions that need relocation. One data variable is put in a shared data section so it will be shared across multiple instances of DLL. One variable is put in a default data section. The two functions are put in separate code sections just to make them page aligned.

When you run the first instance of the application, the application loads and prints the physical addresses of two functions and two data variables. After this, you run the second instance of the same application. In the second instance, the application arranges to load the DLL at a different base address than that of the first instance. Then it prints the physical addresses of two functions and two data variables. Next, the application arranges to load the DLL at the same base address as that of the first instance. In this case, all physical pages are seen to be shared. Next, the application modifies the shared and nonshared variable and modifies the first few bytes of one function, and it prints the physical addresses for two functions and two variables again. We first discuss the code for this sample program and then describe how the output from the sample program demonstrates memory sharing and the effects of the copy-on-write mechanism.

Listing 4-8: SHOWPHYS.C

```
#include<windows.h>
#include <stdio.h>

//"include "gate.h"
//include "getphys.h"
```

```
HANDLE hFileMapping;

/* Imported function/variable addresses */
static void *NonRelocatableFunction = NULL;
static void *RelocatableFunction = NULL;
static void *SharedVariable = NULL;
static void *NonSharedVariable = NULL;

HINSTANCE hDll Instance;
```

The initial portion of the file contains the header inclusion and global variable definitions. The program demonstrates the use of various page attributes, especially to implement the copy-on-write mechanism. As described earlier, the program uses four different types of memory sections. The pointers to the four different types of memory sections are defined as global variables. The hDHInstance stores the instance of the instance handle of the DLL that contains the different kind of memory sections used in this demonstration.

```
/* Loads MYDLL.DLL and initializes addresses of
 * imported functions/variables from MYDLL.DLL and
 * locks the imported areas
 */
int LoadDllAndInitializeVirtualAddresses()
{
hDllInstance = LoadLibrary("MYDLL.DLL");
if (hDllInstance == NULL) {
printf("Unable to load MYDLL.DLL\n");
return -1;
}
printf("MYDLL.DLL loaded at base address = %x\n",
hDllInstance);

NonRelocatableFunction =
GetProcAddress(GetModuleHandle("MYDLL"),
"_NonRelocatableFunction@0");
RelocatableFunction =
GetProcAddress(GetModuleHandle("MYDLL"),
"_RelocatableFunction@0");
       SharedVariable =
GetProcAddress(GetModuleHandle("MYDLL"),
"SharedVariable");
       NonSharedVariable =
GetProcAddress(GetModuleHandle("MYDLL"),
"NonSharedVariable");
```

```
if((!NonRelocatableFunction) ||
(!RelocatableFunction) ||
(!SharedVariable) ||
(!NonSharedVariable)) {
printf("Unable to get the virtual addresses for"
"imports from MYDLL.DLL\n");
FreeLibrary(hDllInstance);
HDllInstance = 0;
return -1;
        }

VirtualLock(NonRelocatableFunction, 1);
VirtualLock(RelocatableFunction, 1);
VirtualLock(SharedVariable, 1);
VirtualLock(NonSharedVariable, 1);

return 0;
```

The four different types of memory sections that we use for the demonstration reside in MYDLL.DLL. The LoadDllAndlnitializeVirtualAddressesO function loads MYDLL.DLL in the calling process's address space and initializes the global variables to point to different types of memory sections in the DLL. The function uses the GetProcAddress() function to get hold of pointers to the exported functions and variables in MYDLL.DLL. The function stores the instance handle for MYDLL.DLL in a global variable so that the FreeDllfJ function can later use it to unload the DLL. The function also locks the different memory sections so that the pages are loaded in memory and the page table entries are valid. Generally, Windows NT does not load the page table entries unless the virtual address is actually accessed. In other words, the memory won't be paged in unless accessed. Also, the system can page out the memory that is not used for some time, again marking the page table entries as invalid. We use the VirtualLock() function to ensure that the pages of interest are always loaded and the corresponding page table entries remain valid.

```
/* Unlocks the imported areas and frees the MYDLL.DLL
 */
void FreeDl1()
{
VirtualUnlock(NonRelocatab!eFunction, 1);
VirtualUnlock(RelocatableFunction, 1);
VirtualUnlock(SharedVariable, 1);
VirtualUnlock(NonSharedVariable, 1);

FreeLibrary(hDllInstance);
```

```
HD11Instance = 0;
NonRelocatableFunction = NULL;
RelocatableFunction = NULL;
SharedVanable = NULL;
NonSharedVariable = NULL;
}
```

The FreeDll() function uses the VirtualUnlock() function to unlock the memory locations locked by the LoadDllAndInitializeVirtualAddressesO function. The function unloads MYDLL.DLL after unlocking the memory locations from the DLL. As the DLL is unloaded, the global pointers to the memory sections in the DLL become invalid. The function sets all these pointers to NULL according to good programming practice.

```
/* Converts the page attributes in readable form
 */
char *GetPageAttributesString(unsigned int PageAttr)
{
static char buffer[100];

strcpy(buffer, "");

strcat(buffer, (PageAttr&0x01)? "P  ": "NP ");
strcat(buffer, (PageAttr&0x02)? "RW ": "R  ");
strcat(buffer, (PageAttr&0x04)? "U ": "S ");
strcat(buffer, (PageAttr&0x40)? "D ": "  ");

return buffer;
}
```

The GetPageAttributesStringO function returns a string with characters showing the page attributes given the page attribute flags. The LSB in the page attributes indicates whether the page is present in memory or the page table entry is invalid. This information is printed as P or NP, which stands for *present* or *not present.* Similarly, R or RW means a *read-only* or *read-write* page; S or U means a *supervisor-mode* or a *user-mode* page; and D means a *dirty* page. The various page attributes are represented by different bits in the PageAttr parameter to this function. The function checks the bits and determines whether the page possesses the particular attributes.

```
/* Displays virtual to physical address mapping
 */
int DisplayVirtualAndPhysicalAddresses()
{
```

```
DWORD pNonRelocatableFunction = 0;
DWORD pRelocatableFunction = 0;
DWORD pSharedVariable = 0;
DWORD pNonSharedVariable = 0;

DWORD aNonRelocatableFunction = 0;
DWORD aRelocatableFunction = 0;
DWORD aSharedVariable = 0;
DWORD aNonSharedVariable = 0;

printf("\nVirtual to Physical address mapping\n");
printf("\n-----------------------------------\n");
printf("Variable/function   Virtual Physical Page\n");
printf("                    Address Address Attributes\n")
      printf("-----------------------------------\n");

GetPhysicalAddressAndPageAttributes(
NonRelocatableFunction,
&pNonRelocatableFunction, &aNonRelocatableFunction);

GetPhysicalAddressAndPageAttributes(
RelocatableFunction,
&pRelocatableFunction, &aRelocatableFunction);
GetPhysicalAddressAndPageAttributes(
SharedVariable,
                    &pSharedVariable,
&aSharedVariable);

GetPhysicalAddressAndPageAttributes(
NonSharedVariable,
&pNonSharedVariable,
&aNonSharedVariable);

printf("NonRelocatableFunction\t %8x\t %8x\t %s\n",
NonRelocatableFunction,
pNonRelocatableFunction,
GetPageAttributesString(
aNonRelocatableFunction));

printf("RelocatableFunction\t %8x\t %8x\t %s\n",
                    RelocatableFunction,
pRelocatableFunction,
GetPageAttributesString(
aRelocatableFunction));
```

```
printf("SharedVariable\t %8x\t %8x\t %s\n",
SharedVariable,
pSharedVariable,
GetPageAttributesString(
aSharedVariable));

printf("NonSharedVariable\t %8x\t %8x\t %s\n",
NonSharedVariable,
pNonSharedVariable,
GetPageAttributesString(
aNonSharedVariable));

printf("-----------------------------------\n\n");

return 0;
}
```

The DisplayVirtualAndPhysicalAddressesO function is a utility function that displays the virtual address, the physical address, and the page attributes for different memory sections. It uses the global pointers to the different sections in MYDLL.DLL initialized by the LoadDllAndInitializeVirtualAddresses() function. It uses the GetPhysicalAddressAndPageAttributesO function to get hold of the physical page address and the page attributes for the given virtual address. The first parameter to the GetPhysicalAddressAndPageAttributesO function is the input virtual address. The function fills in the physical address for the input virtual address in the memory location pointed to by the second parameter and the page attributes in the location pointed to by the third parameter.

```
int  FirstInstance()
(
printf("***This is the first instance of the"
" showphys program***\n\n");
printfC'Loading DLL MYDLL.DLL\n");

if (LoadDllAndInitializeVirtualAddresses()!=0) {
return -1;
}

DisplayVirtualAndPhysicalAddresses();
printf("Now Run another copy of showphys ...\n");
getchar();
FreeDll();
```

We want to demonstrate the sharing of memory sections by the DLL loaded by two different processes. You need to run two instances of the demonstration program. The FirstInstance() function is executed when you run the first instance of the program. The first instance loads the DLL and prints the physical addresses and page attributes for the various memory sections in the DLL. Then, the function asks you to run another instance of the program. Now there are two processes that loaded MYDLL.DLL. You can compare the outputs from these two instances to check how the memory sections are shared. More on this when we explain the output from this sample program.

```
int NonFirstInstance()
{
       DWORD OldAttr;
       HINSTANCE hJunk;

       printf("***This is another instance of the showphys
program***\n\n");
       printf("Loading DLL MYDLL.DLL at diffrent base address than
that of the first instance\n");

       CopyFile("MYDLL.DLL", "JUNK.DLL", FALSE);

       hJunk=LoadLibrary("JUNK.DLL");

       if (hJunk==NULL) {
              printf("Could not find JUNK.DLL\n");
              return -1;
       }

       if (LoadDllAndInitializeVirtualAddresses()!=0) {
              FreeLibrary(hJunk);
              return -1;
       }
       FreeLibrary(hJunk);

       DisplayVirtualAndPhysicalAddresses();

       FreeDll();

       printf("Loading DLL MYDLL.DLL at same base address as that of
the first instance\n");

       if (LoadDllAndInitializeVirtualAddresses()!=0) {
              return -1;
```

```
        }

        DisplayVirtualAndPhysicalAddresses();

        printfC1....Modifying the code bytes at the start of
NonRelocatableFunction\n");
        VirtualProtect(NonRelocatableFunction, 1, PAGE_READWRITE,
&01dAttr);
        *(unsigned char *)NonRelocatableFunction=0xE9;


        printf("... .Modifying the value of SharedVan able\n");
        *(char *)SharedVariable=0x10;

        pnntf("... .Modi fying the NonSharedVan able1s value\n\n";
        *(char *)NonSharedVanable=0xl0;

        DisplayVirtualAndPhysicalAddresses();

        FreeDll();

        return 0;
}
```

The second instance of the program does a lot more work than the first instance. The sharing of the DLL memory sections depends on the way the instance loads the DLL and accesses the memory locations in the DLL. In more concrete terms, the sharing depends on whether the second instance loads the DLL at the same base address as the first instance. It also depends on whether the instances only read the memory sections or any of the instances write to the memory sections. To demonstrate this, the NonFirstlnstanceO function first loads the DLL at a different base address than the first instance. The function ensures that the DLL is loaded at a different base address by loading JUNK.DLL before loading MYDLL.DLL. JUNK.DLL has the same preferred base address as that of MYDLL.DLL. The first instance loads MYDLL.DLL at its preferred base address by default. In the second instance, MYDLL.DLL cannot be loaded at its preferred base address because the address range is already occupied by JUNK.DLL. After MYDLL.DLL is loaded at a different base address, there is no reason for the program to keep JUNK.DLL loaded, and so it frees the JUNK.DLL instance. Next, the function prints the physical addresses and page attributes of the memory sections in MYDLL.DLL using the DisplayVirtual AndPhysicalAddresses() function. The information printed here can be compared with the output of the first instance of the program to get an idea of how the DLLs loaded at different base addresses share the memory sections.

The NonFirstInstance also demonstrates the sharing of memory sections by MYDLL.DLL loaded at the same base address by two processes. It unloads MYDLL.DLL and loads it again. This time MYDLL.DLL is loaded at its preferred base address because now that JUNK.DLL is no more loaded, the virtual address space is not occupied by anything. Thus, MYDLL.DLL is loaded at the same base address in both the first and the second instance of the program. The physical addresses and the page attributes printed here demonstrate the memory sharing by MYDLL.DLL when loaded at the same base address in two processes. Next, the NonFirstInstance() function writes to some of the memory locations in MYDLL.DLL. As we explain soon, this action affects the memory sharing between the instances. As described earlier, the code sections are marked read-only by Windows NT. The function uses the VirtualProtectO API function to change the attributes of the NonRelocatable Function() so that it can modify a few bytes at the start of this function. You can modify the data variables from MYDLL.DLL without any such hassle because the data variables have the read-write attribute.

```
int DecideTheInstanceAndAct()
{

hFileMapping = CreateFileMapping(
(HANDLE)0xFFFFFFFF,
NULL,
PAGE_READWRITE,
                                0,
0x1000,
"MyFileMapping");
if (hFileMapping == NULL) {
printf("Unable to create file mapping\n");
FreeDll();
return -1;
      }
if (GetLastError() == ERROR_ALREADY_EXISTS) {
NonFirstInstance();
} else {
FirstInstance();
}
}
```

The sample program does not accept any parameter to indicate whether it's the first instance. It uses a simple trick to decide it: It creates a named file mapping. The call to the CreateFileMappingO API function sets the last error to ERROR_ALREADY_EXISTS if a mapping with the same name already exists. This indicates that an instance that created the file mapping is already running. In other words, if the program can successfully create the named file mapping, it's the first instance of

the program. Otherwise, another instance (that As, the fixst instance) of the program is already running and the current instance is the second instance. Depending on whether it's the first instance, the DecideThelnstanceAndActO function calls the NonFirstlnstanceO function or the FirstInstance() function. A file mapping is automatically destroyed by the operating system when the reference count drops to zero. The sample program does not explicitly close the handle to the mapping. The handle is closed and the reference count for the memory mapping is decremented when the program exits. The mapping is freed up when the last instance of the program exits.

```
main()
{
int rc;

/* Creates callgate to get PTE entries from ring 3
 * application
 */
if ((rc = CreateRingOCallGate()) != SUCCESS) {
printf("Unable to create callgate, rc=%x\n",rc);
return -1;
       }
DecideTheInstanceAndDoTheThings();

/* Releases the callgate */
FreeRingOCallGate();
}
```

The main() function starts by a call to the CreateRingOCallGate() function that is located in the GETPHYS.C file. The sample program uses the callgate mechanism to access the page tables. As described earlier, the page tables reside in the kernel memory and are not accessible to the user-mode code. The CreateRingOCallGatefJ function sets up a function that reads in the page tables to be executed in kernel mode. The DisplayVirtualAndPhysicalAddressesO function later uses this function to get hold of the physical address and the page attributes for a given virtual address. After creating the callgate, the main function passes control to the DecideThelnstanceAndDoTheThingsO function. The callgate is freed up by the program before exiting.

Listing **4-9: GETPHYS.C**

```
#include <windows.h>
#include <stdio.h>
#include "..\cgate\dll\gate.h"

static short Cal1GateSelector;
```

The GETPHYS.C file implements the function to access the page table using the callgate mechanism. The GATE.H file is included because it contains the prototypes for functions that deal with the callgate manipulation. The segment selector of the callgate used by the program is stored in the global variable, CallGateSelector.

```
/* C function called from assembly langauage stub */
BOOL _stdcall
CFuncGetPhysicalAddressAndPageAttributes(
unsigned int VirtualAddress,
unsigned int *PhysicalAddress,
unsigned int *PageAttnbutes)
{
unsigned int *PageTableEntry;

*PhysicalAddress = 0;
*PageAttributes = 0;

PageTableEntry = (unsigned int *)0xC0000000U +
(VirtualAddress > 0x0CU);

if ((*PageTableEntry)&0x01) {
*PhysicalAddress =
((*PageTableEntry)&0xFFFFF000U) +
(VirtualAddress&0x00000FFFU);

*PageAttributes = (*PageTableEntry)&0x00000FFFU;
return TRUE;
} else {
return FALSE;
)
}
```

The CfuncGetPhysicalAddressAndPageAttributesO function executes in kernel mode using the callgate mechanism. The function depends on the fact that page tables for a process are always mapped at the virtual address 0xC0000000. It's an array of 1024 page tables where each page table is an array of 1024 page table entries. You can access the memory area as if it were a single contiguous array of page table entries. The first entry in this big array corresponds to a virtual address in the range 0 - 4096, the second entry corresponds to virtual address range 4096 - 8192, and so on. The function calculates the index in the big PTE array by dividing the given virtual address by 4096 - that is, by shifting the virtual address by 12 bits. Adding the index in the base address of the PTE array gives us the required PTE. Each PTE is 4 bytes (32 bits) long. Out of these 32 bits, the upper 20 bits in the PTE denote the address of the physical page, and the lower 12 bits denote the page

attributes. The physical address and the page attributes are valid only if the LSB is set. The function checks the LSB and if the bit is set, it separates out the physical page address and the page attributes by masking off appropriate bits from the PTE. The function adds the offset within the page to the physical page address to get the physical address for the given virtual address.

```
BOOL GetPhysicalAddressAndPageAttributes(
void *VirtualAddress,
unsigned int *PhysicalAddress,
unsigned int *PageAttributes)
{
BOOL rc;
static short farcall[3];

if (!CallGateSelector) {
return FALSE;
}

farcall[2] = CallGateSelector;

_asm {
mov eax, PageAttributes
mov ecx, PhysicalAddress
mov edx, VirtualAddress
call fword ptr [farcall]
mov rc, eax
}
return rc;
}
```

The GetPhysicalAddressAndPageAttributesO function runs in user mode and invokes the CfuncGetPhysicalAddressAndPageAttributesO function in kernel mode using the callgate mechanism. It uses the callgate initialized by the call to the CreateRing0CallGate() function. The parameters to the kernel-mode function are passed through the processor registers. An intermediate Assembly language function, namely, GetPhysicalAddressAndPageAttributes(), converts the register parameters to stack parameters.

```
int CreateRing0CallGate()
{
DWORD rc;

rc = CreateCallGate(
_GetPhysicalAddressAndPageAttributes,
```

```
0,
&CallGateSelector);
return  re;
)
```

The CreateRing0CallGate() function is a utility function that uses the CreateCallGate() function provided by GATE.DLL to create a callgate to execute the GetPhysicalAddressAndPageAttributesO function in kernel mode. It stores the segment selector of the created callgate in the CallGateSelector global variable, which is used later by the GetPhysicalAddressAndPageAttributesQ function while invoking the kernel-mode function.

```
int FreeRing0CallGate()
{
DWORD rc;

rc = FreeCallGate(CallGateSelector);
if (rc == SUCCESS) {
CallGateSelector = 0;
      }
return rc;
}
```

The FreeRing0CallGate() function is another utility function that destroys the callgate created by the CreateCallGate() function. It uses the FreeCallGate() interface function provided by GATE.DLL.

Listing 4-10: RINGO.ASM

```
.386
.model small

.code

public __GetPhysicalAddressAndPageAttributes
extrn _CFuncGetPhysicalAddressAndPageAttributes@12:near

include ..\include\undocnt.inc

__GetPhysicalAddressAndPageAttributes proc
Ring0Prolog
push eax
push ecx
push edx
call _CFuncGetPhysicalAddressAndPageAttributes@12
```

```
RingOEpilog
retf
_GetPhysicalAddressAndPageAttributes  endp

END
```

The GetPhysicalAddressAndPageAttributes() function gets control through the callgate. The function executes the RingOProlog macro just after entering the function to enable paging in kernel mode. It converts the register parameters to stack parameters because CfuncGetPhysicalAddressAndPageAttributes() is a C function that expects the parameters on stack.

Listing 4-11 presents the output from the previous sample program. Note the differences between the physical addresses and page attributes printed by the first instance and the second instance. See if you can explain the output and match your findings with our description that comes after this output.

Here are two instances of the showphys program.

Listing **4-11: showphys** program

```
Loading DLL MYDLL.DLL
MYDLL.DLL loaded at base address    20000000

Virtual address to Physical address mapping


Variable/function           VirtualPhysical       Page
AddressAddressAttributes
-------------------------------------------------------
NonRelocatableFunction      20001000      d8b000 P  R  U
RelocatableFunction         20002000      d8a000 P  R  U
SharedVariable              2000c000      e44000 P  RW U
NonSharedVariable           2000b000      6b7000 P  R  U
-------------------------------------------------------
Now Run another copy of showphys ...
```

This is another instance of the showphys program:

```
Loading DLL MYDLL.DLL at diffrent base address than that of the
first instance
MYDLL.DLL loaded at base address = 7eOOOO

Virtual address to Physical address mapping
```

```
Variable/function          VirtualPhysical      Page
                                 AddressAddressAttributes
-----------------------------------------------------------
NonRelocatableFunction     7e1000 d8b000 P  R  U
RelocatableFunction        7e2000 1d6c000P  R  U
SharedVariable             7ec000 e44000 P  RW U
NonSharedVariable          7eb000 6b7000 P  R  U
-----------------------------------------------------------

Loading DLL MYDLL.DLL at same base address as that of the first
instance
MYDLL.DLL loaded at base address = 20000000

Virtual address to Physical address mapping

-----------------------------------------------------------
Variable/function          VirtualPhysical      Page
                           AddressAddressAttributes
-----------------------------------------------------------
NonRelocatableFunction     20001000      d8b000 P  R  U
RelocatableFunction        20002000      d8a000 P  R  U
SharedVariable             2000c000      e44000 P  RW U
NonSharedVariable          2000b000      6b7000 P  R  U
-----------------------------------------------------------

....Modifying the code bytes at the start of NonRelocatableFunction
....Modifying the value of SharedVariable
....Modifying the NonSharedVariable's value

Virtual address to Physical address mapping

-----------------------------------------------------------
Variable/function    VirtualPhysical      Page
AddressAddressAttributes
-----------------------------------------------------------
NonRelocatableFunction     20001000      87e000 P  RW U D
RelocatableFunction        20002000      d8a000 P  R  U
SharedVariable             2000c000      e44000 P  RW U D
NonSharedVariable          2000b000      1ceb000P  RW U D
-----------------------------------------------------------
```

Note the page attributes from the output of the first instance. The functions are marked read-only, as expected. The unshared variable is also marked read-only. This is because Windows NT tries to share the data space also. As described earlier, such pages are marked for copy-on-write, and as soon as the process modifies any location in the page, the process gets a private copy of the page to write to. The other page attributes- show that the *PTE is* valid, the page is a user-mode page, and nobody has modified the page so far.

Now, compare the output from the first instance with the output from the second instance when it loaded the MYDLL.DLL at a base address different from that in the first instance. As expected, the virtual addresses of all the memory sections are different than those for the first instance. The physical addresses are the same except for the physical address of the relocatable function. This demonstrates that the code pages are marked as copy-on-write, and when the loader modifies the code pages while performing relocation, the process gets a private writable copy. Our nonrelocatable function does not need any relocation; hence, the corresponding pages are not modified. The second instance can share these pages with the first instance and hence has the same physical page address.

To cancel out the effects of relocation, the second instance loads MYDLL.DLL at the same base address as that in the first instance. Yup! Now, the virtual address matches the ones from the first instance. Note that the physical address for the relocatable function also matches that in the output from the first instance. The loader need not relocate the function because the DLL is loaded at the preferred base address. This allows more memory sharing and provides optimal performance. It's reason enough to allocate proper, nonclashing preferred base addresses for your DLLs.

This ideal share-all situation ceases to exist as soon as a process modifies some memory location. Other processes cannot be allowed to view these modifications. Hence, the modifying process gets its own copy of the page The second instance of the sample program demonstrates this by modifying the data variables and a byte at the start of the nonrelocatable function. The output shows that the physical address of the nonrelocatable doesn't match with the first instance. The nonrelocatable function is not modified by the loader, but it had the same effect on sharing when we modified the function. The shared variable remains a shared variable. Its physical address matches that in the first instance because all the processes accessing a shared variable are allowed to see the modifications made by other processes. But the nonshared variable has a different physical address now. The second instance cannot share the variable with the first instance and gets its own copy. The copy was created by the system page fault handler when we tried to write to a read-only page and the page was also marked for copy-on-write. Note that the page is now marked read-write. Hence, further writes go through without the operating system getting any page faults. Also, note that the modified pages are marked as dirty by the processor.

Switching Context

As we saw earlier, Windows NT can switch the memory context to another process by setting the appropriate page table directory. The 80386 processor requires that the pointer to the current page table directory be maintained in the CR3 register. Therefore, when the Windows NT scheduler wants to perform a context switch to another process, it simply sets the CR3 register to the page table directory of the concerned process.

Windows NT needs to change only the memory context for some API calls such as VirtualAllocEx(). The VirtualAllocEx() API call allocates memory in the memory space of a process other than the calling process. Other system calls that require memory context switch are ReadProcessMemoryO and WriteProcessMemory(). The ReadProcessMemoryO and WriteProcessMemoryO system calls read and write, respectively, memory blocks from and to a process other than the calling process. These functions are used by debuggers to access the memory of the process being debugged. The subsystem server processes also use these functions to access the client process's memory. The undocumented KeAttchProcessO function from the NTOSKRNL module switches the memory context to specified process. The undocumented KeDetachProcessO function switches it back. In addition to switching memory context, it also serves as a notion of current process. For example, if you attach to a particular process and create a mutex, it will be created in the context of that process. The prototypes for KeAttachProcessO and KeDetachProcessO are as follows:

```
NTSTATUS  KeAttachProcess(PEB  *);
NTSTATUS  KeDetachProcess  ();
```

Another place where a call to the KeAttachProcessQ function appears is the NtCreateProcessO system call. This system call is executed in the context of the parent process. As a part of this system call, Windows NT needs to map the system DLL (NTDLL.DLL) in the child process's address space. Windows NT achieves this by calling KeAttachProcessO to switch the memory context to the child process. After mapping the DLL, Windows NT switches back to the parent process's memory context by calling the KeDetachProcessO function.

The following sample demonstrates how you can use the KeAttachProcessO and KeDetachProcessO functions. The sample prints the page directories for all the processes running in the system. The complete source code is not included. Only the relevant portion of the code is given. Because these functions can be called only from a device driver, we have written a device driver and provided an IOCTL that demonstrates the use of this function. We are giving the function that is called in response to DeviceloControl from the application. Also, the output of the program is shown in kernel mode debugger's window (such as SoftICE). Getting the information back to the application is left as an exercise for the reader.

```
void DisplayPageDirectory(void *Peb)
```

```
{
unsigned int *PageDirectory =
(unsigned int *)0xC0300000;
int i;
int ctr=0;

KeAttachProcess(Peb);

for (i = 0; i < 1024; i++) {
if (PageDirectory[i]&0x01) {
if ((ctr%8) == 0)
DbgPrint("  \n");
DbgPrint("%08x ", PageDirectory[i]&0xFFFFF000);
ctr++;
}
}
DbgPrint("\n\n");

KeDetachProcess();
}
```

The DisplayPageDirectoryO function accepts the PEB for the process whose page directory is to be printed. The function first calls the KeAttachProcess() function with the given PEB as the parameter. This switches the page directory to the desired one. Still, the function can access the local variables because the kernel address space is shared by all the processes. Now the address space is switched, and the OxC030000 address points to the page directory to be printed. The function prints the 1024 entries from the page directory and then switches back to the original address space using the KeDetachProcess() function.

```
void DisplayPageDirectoryForAllProcesses()
{
        PLIST_ENTRY ProcessListHead, ProcessListPtr;
        ULONG BuildNumber;
        ULONG ListEntryOffset;
        ULONG NameOffset;

        BuildNumber=NtBuildNumber & 0x0000FFFF;
        if ((BuildNumber==0x421) || (BuildNumber==0x565)) { // NT
3.51 or NT 4.0
                ListEntryOffset=0x98;
                NameOffset=0x1DC;
        } else if (BuildNumber==0x755) {// Windows 2000 beta2
                ListEntryOffset=0xA0;
```

```
            NameOffset=0x1FC;
        } else {
            DbgPrnt("Unsupported NT Version\n");
            return;
        }

        ProcessListHead=ProcessListPtr=(PLIST_ENTRY)(((char
*)PsImtialSystemProcess)+ListEntryOffset);
        while (ProcessListPtr >Flink!=ProcessListHead) {
            void *Peb;
            char ProcessName[16];

            Peb=(void *)(((char *)ProcessListPtr)-
ListEntryOffset);
            memset(ProcessName, 0, sizeof(ProcessName));
            memcpy(ProcessName, ((char *)Peb)+NameOffset, 16);

            DbgPrint("**%s Peb @%x**  ", ProcessName, Peb);
            DisplayPageDirectory(Peb);
            ProcessListPtr=ProcessListPtr->Flink;
        }
```

The DisplayPageDirectoryForAllProcesses() function calls the DisplayPage DirectoryO function for each process in the system. All the processes running in a system are linked in a list. The function gets hold of the list of the processes from the PEB of the initial system process. The PsInitialSystemProcess variable in NTOSKRNL holds the PEB for the initial system process. The process list node is located at an offset of 0x98 (OxAO for Windows NT 5.0) inside the PEB. The process list is a circular linked list. Once you get hold of any node in the list, you can traverse the entire list. The DisplayPageDirectoryForAllProcessesO function completes a traversal through the processes list by following the Flink member, printing the page directory for the next PEB in the list every time until it reaches back to the PEB it started with. For every process, the function first prints the process name that is stored at a version-dependent offset within the PEB and then calls the DisplayPageDirectoryO function to print the page directory.

Here, we list partial output from the sample program. Please note a couple of things in the following output. First, every page directory has 50-odd valid entries while the page directory size is 1024. The remaining entries are invalid, meaning that the corresponding page tables are either not used or are swapped out. In other words, the main memory overhead of storing page tables is negligible because the page tables themselves can be swapped out. Also, note that the page directories have the same entries in the later portion of the page directory. This is because this part represents the kernel portion shared across all processes by using the same set of page tables for the kernel address range.

Listing 4-12: Displaying page directories: output

```
**System Peb @fdf06b60**
00500000 008cf000 008ce000 00032000 00034000 00035000 ... ... ...
00040000 00041000 00042000 00043000 00044000 00045000 ... ... ...
00048000 00049000 0004a000 0004b000 0004c000 0004d000 ... ... ...
00050000 00051000 00052000 00053000 00054000 00055000 ... ... ...
00058000 00059000 0005a000 0005b000 0005c000 0005d000 ... ... ...
00020000 00021000 00023000 0040b000 0040c000 0040d000 ... ... ...
00410000 00411000 00412000 00413000 00414000 00415000 ... ... ...

**smss.exe Peb @fe2862e0**
00032000 00034000 00035000 00033000 00e90000 00691000 ... ... ...
00043000 00044000 00045000 00046000 00047000 00048000 ... ... ...
0004b000 0004c000 0004d000 0004e000 0004f000 00050000 ... ... ...
00053000 00054000 00055000 00056000 00057000 00058000 ... ... ...
0005b000 0005c000 0005d000 0005e000 0005f000 00020000 ... ... ...
0040b000 0040c000 0040d000 0040e000 0040f000 00410000 ... ... ...
00413000 00414000 00415000 00416000 00031000

... ... ...

**winlogon.exe Peb @fe27dde0**
00032000 00034000 00035000 00033000 00be1000 00953000 ... ... ...
00043000 00044000 00045000 00046000 00047000 00048000 ... ... ...
0004b000 0004c000 0004d000 0004e000 0004f000 00050000 ... ... ...
00053000 00054000 00055000 00056000 00057000 00058000 ... ... ...
0005b000 0005c000 0005d000 0005e000 0005f000 00020000 ... ... ...
0040b000 0040c000 0040d000 0040e000 0040f000 00410000 ... ... ...
00413000 00414000 00415000 00416000 00031000

... ... ...
```

Differences between Windows NT and Windows 95/98

Generally, the memory management features offered by Windows 95/98 are the same as those in Windows NT. Windows 95/98 also offers 32-bit flat separate address space for each process. Features such as shared memory are still available. However, there are some differences. These differences are due to the fact that Windows 95/98 is not as secure as Windows NT. Many times, Windows 95/98 trades off security for performance reasons. Windows 95/98 still has the concept of user-mode and kernel-mode code. The bottom 3GB is user-mode space, and the top

1GB is kernel-mode space. But the 3GB user-mode space can be further divided into shared space and private space for Windows 95/98. The 2GB to 3GB region is the shared address space for Windows 95/98 processes. For all processes, the page tables for this shared region point to the same set of physical pages.

-'• All the shared DLLs are loaded in the shared region. All the system DLLs — for example, KERNEL32.DLL and USER32.DLL-are shared DLLs. Also, a DLL's code/data segment can be declared shared while compiling the DLL, and the DLL will get loaded in the shared region. The shared memory blocks are also allocated space in the shared region. In Windows 95/98, once a process maps a shared section, the section is visible to all processes. Because this section is mapped in shared region, other processes need not map it separately.

There are advantages as well as disadvantages of having such a shared region. Windows 95/98 need not map the system DLLs separately for each process; the corresponding entries of page table directory can be simply copied for each process. Also, the system DLLs loaded in shared region can maintain global data about all the processes and separate subsystem processes are not required. Also, most system calls turn out to be simple function calls to the system DLLs, and as a result are very fast. In Windows NT, most system calls either cause a context switch to kernel mode or a context switch to the subsystem process, both of which are costly operations. For developers, loading system DLLs in a shared region means that they can now put global hooks for functions in system DLLs.

For all these advantages, Windows 95/98 pays with security features. In Windows 95/98, any process can access all the shared data even if it has not mapped it. It can also corrupt the system DLLs and affect all processes.

Summary

In this chapter, we discussed the memory management of Windows NT from three different perspectives. Memory management offers programmers a 32-bit flat address space for every process. A process cannot access another process's memory or tamper with it, but two processes can share memory if they need to. Windows NT builds its memory management on top of the memory management facilities provided by the microprocessor. The 386 (and above) family of Intel microprocessors provides support for segmentation plus paging. The address translation mechanism first calculates the virtual address from the segment descriptor and the specified offset within the segment. The virtual address is then converted to a physical address using the page tables. The operating system can restrict access to certain memory regions by using the security mechanisms that are provided both at the segment level and the page level.

Windows NT memory management provides the programmer with flat address space, data sharing, and so forth by selectively using the memory management features of the microprocessor. The virtual memory manager takes care of the paging and allows 4GB of virtual address space for each process, even when the entire

system has much less physical memory at its disposal. The virtual memory manager keeps track of all the physical pages in the system through the page frame database (PFD). The system also keeps track of the virtual address space for each process using the virtual address descriptor (VAD) tree. Windows NT uses the copy-on-write mechanism for various purposes, especially for sharing the DLL data pages. The memory manager has an important part in switching the processor context when a process is scheduled for execution. Windows 95/98 memory management is similar to Windows NT memory management with the differences being due to the fact that Windows 95/98 is not as security conscious as Windows NT.