CHAPTER 7

Windows 2000 Object Management

There is hardly anything more fascinating in the internals of Windows 2000 than the world of its objects. If the memory space of an operating system is viewed as the surface of a planet, the objects are the creatures living on it. Several types of objects exist—small and large ones, simple and complex ones—and they interact in various ways. Windows 2000 features a clever, well-structured object management mechanism that is almost completely undocumented. This chapter attempts to give you a small insight into this huge, complex universe. Unfortunately, this part of Windows 2000 is one of the best-kept secrets of Microsoft, and many questions must be left unanswered here. However, I hope that this chapter will serve as a starting point for others, helping them to go "where no man has gone before."

WINDOWS 2000 OBJECT STRUCTURES

The companion CD of this book contains a large header file named `w2k_def.h` in the `\src\common\include` directory that makes the heart of a Windows 2000 system programmer throb with joy. It is a large collection of constant and type definitions, resulting from years of Windows NT/2000 spelunking. The `w2k_def.h` file is designed to be included in Win32 applications as well as kernel-mode drivers, using conditional compilation to account for their different build environments. For example, Win32 applications can't make use of the `ntdef.h` and `ntddk.h` files that contain most of the kernel data type definitions. Therefore, `w2k_def.h` includes all `#define's` and `typedef's` found in the Device Documentation Kit (DDK) header files that are required in the definitions of the undocumented items. To avoid redefinition errors in a kernel-mode driver build, these definitions are put into an `#ifdef _USER_MODE_` clause, so they are ignored by the compiler if the `_USER_MODE_` symbol is not defined. This means that you must put a `#define _USER_MODE_` line into your source code before including `w2k_def.h` to enable the processing of the DDK definitions in a Win32 application or DLL build. The `#else` clause of the `#ifdef _USER_MODE_` construct contains a small number of definitions that are missing from the Windows 2000 DDK header files, such as the `SECURITY_DESCRIPTOR` and `SECURITY_DESCRIPTOR_CONTROL` types.

BASIC OBJECT CATEGORIES

Although objects are clearly the gist of the Windows 2000 operating system, you will find remarkably little information about their inner structure in the DDK. Out of the 21 `Ob*()` object manager API functions exported by `ntoskrnl.exe`, only 6 are listed in the DDK documentation. API functions that receive pointers to objects as arguments usually define these pointers as simple `PVOID` types. If you search the main DDK header files `ntdef.h` and `ntddk.h` for occurrences of type definitions that somehow are related to objects, you won't find much useful information. Some important object data types are defined as placeholders only. For example, the `OBJECT_TYPE` structure appears as `typedef struct _OBJECT_TYPE *POBJECT_TYPE;` just to keep the compiler happy, without revealing anything useful about its internals.

Whenever you come across an object pointer, you should view it as a linear address that divides a memory-resident structure into two parts: an `object header` and an `object body`. The object pointer doesn't point to the base address of the object itself, but to its body section that immediately follows the header. Therefore, the header parts of an object must be accessed by applying negative offsets to the object pointer. The internals of the object body are completely dependent on the type of object and may vary considerably. The most simple object is the event object with its 16-byte body. Among the most complex ones are thread and process objects, which are several hundred bytes. Basically, the object body types can be sorted into the following three main categories:

1. `Dispatcher objects` reside on the lowest system level and share a common data structure called `DISPATCHER_HEADER` (Listing 7-1) at the beginning of their object bodies. This header contains an object type ID and the length of the object body in 32-bit `DWORD` units. The names of all dispatcher object structures start with a `K` for "kernel." The presence of a `DISPATCHER_HEADER` makes an object "waitable." This means that the object can be passed to the synchronization functions `KeWaitForSingleObject()` and `KeWaitForMultipleObjects()`, which are the ones the Win32 API functions `WaitForSingleObject()` and `WaitForMultipleObjects()` are built upon.

```
typedef struct _DISPATCHER_HEADER
        {
/*000*/ BYTE       Type;         // DISP_TYPE_*
/*001*/ BYTE       Absolute;
/*002*/ BYTE       Size;         // number of DWORDs
/*003*/ BYTE       Inserted;
/*004*/ LONG       SignalState;
/*008*/ LIST_ENTRY WaitListHead;
/*010*/ }
        DISPATCHER_HEADER,
      * PDISPATCHER_HEADER,
     **PPDISPATCHER_HEADER;
```

LISTING 7-1. *Definition of the* `DISPATCHER_HEADER`

2. `I/O system data structures` are higher-level objects whose body starts with a `SHORT` member specifying an object type ID. Usually, this ID is followed by another `SHORT` or `WORD` member indicating the object body size in 8-bit `BYTE` units. However, not all objects of this category follow this guideline.
3. `Other objects`—some objects fit into neither of the above categories.

Note that the type IDs of dispatcher objects and I/O system data structures—named I/O objects from now on—are assigned independently and hence overlap. Table 7-1 lists the dispatcher object types of which I'm currently aware. Some of the structures in the "C Structure" column are defined in the DDK header file `ntddk.h`. Unfortunately, the most interesting ones, such as `KPROCESS` and `KTHREAD`, are missing. Don't worry, however—these special object types will be discussed in detail later in this chapter. All undocumented structures whose internals are at least partially known to me are included in the header file `w2k_def.h` on the companion CD, as well as in Appendix C of this book.

TABLE 7-1. *Summary of Dispatcher Objects*

ID	TYPE	C STRUCTURE	DEFINITION
0	DISP_TYPE_NOTIFICATION_EVENT	KEVENT	ntddk.h
1	DISP_TYPE_SYNCHRONIZATION_EVENT	KEVENT	ntddk.h
2	DISP_TYPE_MUTANT	KMUTANT, KMUTEX	ntddk.h
3	DISP_TYPE_PROCESS	KPROCESS	w2k_def.h
4	DISP_TYPE_QUEUE	KQUEUE	w2k_def.h

(continued)

TABLE 7-1. *(continued)*

ID	TYPE	C STRUCTURE	DEFINITION
5	DISP_TYPE_SEMAPHORE	KSEMAPHORE	ntddk.h
6	DISP_TYPE_THREAD	KTHREAD	w2k_def.h
8	DISP_TYPE_NOTIFICATION_TIMER	KTIMER	ntddk.h
9	DISP_TYPE_SYNCHRONIZATION_TIMER	KTIMER	ntddk.h

Table 7-2 summarizes the I/O objects I have identified so far. Only the first 13 IDs are defined in `ntddk.h.` Again, some of the structures in the "C Structure" column can be looked up in the DDK. Some of the remaining ones are included in `w2k_def.h` and in Appendix C of this book.

TABLE 7-2. *Summary of I/O Objects*

ID	TYPE	C STRUCTURE	DEFINITION
1	IO_TYPE_ADAPTER	ADAPTER_OBJECT	
2	IO_TYPE_CONTROLLER	CONTROLLER_OBJECT	ntddk.h
3	IO_TYPE_DEVICE	DEVICE_OBJECT	ntddk.h
4	IO_TYPE_DRIVER	DRIVER_OBJECT	ntddk.h
5	IO_TYPE_FILE	FILE_OBJECT	ntddk.h
6	IO_TYPE_IRP	IRP	ntddk.h
7	IO_TYPE_MASTER_ADAPTER		
8	IO_TYPE_OPEN_PACKET		
9	IO_TYPE_TIMER	IO_TIMER	w2k_def.h
10	IO_TYPE_VPB	VPB	ntddk.h
11	IO_TYPE_ERROR_LOG	IO_ERROR_LOG_ENTRY	w2k_def.h
12	IO_TYPE_ERROR_MESSAGE	IO_ERROR_LOG_MESSAGE	ntddk.h
13	IO_TYPE_DEVICE_OBJECT_ EXTENSION	DEVOBJ_EXTENSION	ntddk.h
18	IO_TYPE_APC	KAPC	ntddk.h
19	IO_TYPE_DPC	KDPC	ntddk.h
20	IO_TYPE_DEVICE_QUEUE	KDEVICE_QUEUE	ntddk.h
21	IO_TYPE_EVENT_PAIR	KEVENT_PAIR	w2k_def.h
22	IO_TYPE_INTERRUPT	KINTERRUPT	
23	IO_TYPE_PROFILE	KPROFILE	

THE OBJECT HEADER

The body of an object can assume any form suitable for the creator of the object. The Windows 2000 object manager doesn't impose any restrictions on the size and structure of the object body. Contrary to this, there is much less freedom with the header portion of an object. Figure 7-1 shows the memory layout of a full-featured object, with the maximum number of header fields. Every object features at least a basic `OBJECT_HEADER` structure, immediately preceding the object body, plus up to four optional structures that supply additional information about the object. As already noted, an object pointer always refers to the object body, not to the header, so the header fields are accessed via negative offsets relative to the object pointer. The basic header contains information about the availability and location of additional header fields, which are stacked up on the `OBJECT_HEADER` structure in the order shown in Figure 7-1, if present. However, this sequence isn't mandatory, and your programs should never rely on it. The information in the `OBJECT_HEADER` is sufficient to locate all header fields regardless of their order, as will be shown in a moment. The only exception is the `OBJECT_CREATOR_INFO` structure that always precedes the `OBJECT_HEADER` immediately if it is included.

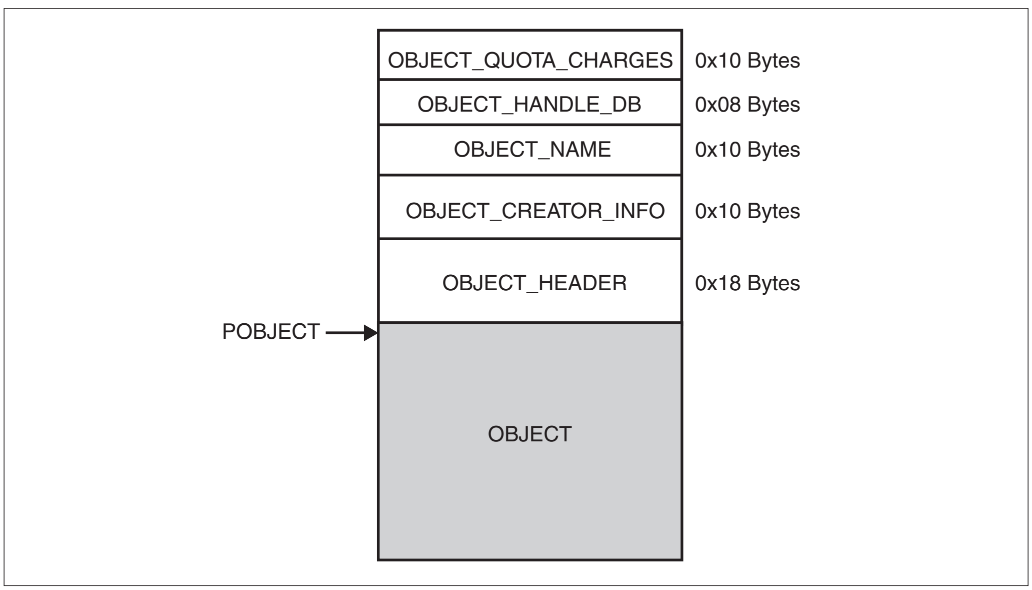

FIGURE 7-1. *Memory Layout of an Object*

Listing 7-2 shows the definition of the `OBJECT_HEADER` structure. Its members serve the following purposes:

- The `PointerCount` member indicates how many active pointer references to this object currently exist. This value is similar to the reference count maintained by Component Object Model (COM) objects. The `ntoskrnl.exe` API functions `ObfReferenceObject()`, `ObReferenceObjectByHandle()`, `ObReferenceObjectByName()`, and `ObReferenceObjectByPointer()` increment the `PointerCount`, and `ObfDereferenceObject()` and `ObDereferenceObject()` decrement it.
- The `HandleCount` member indicates how many open handles currently refer to this object.

```
#define OB_FLAG_CREATE_INFO    0x01 // has OBJECT_CREATE_INFO
#define OB_FLAG_KERNEL_MODE    0x02 // created by kernel
#define OB_FLAG_CREATOR_INFO   0x04 // has OBJECT_CREATOR_INFO
#define OB_FLAG_EXCLUSIVE      0x08 // OBJ_EXCLUSIVE
#define OB_FLAG_PERMANENT      0x10 // OBJ_PERMANENT
#define OB_FLAG_SECURITY       0x20 // has security descriptor
#define OB_FLAG_SINGLE_PROCESS 0x40 // no HandleDBList

typedef struct _OBJECT_HEADER
        {
/*000*/ DWORD        PointerCount;       // number of references
/*004*/ DWORD        HandleCount;        // number of open handles
/*008*/ POBJECT_TYPE ObjectType;
/*00C*/ BYTE         NameOffset;         // -> OBJECT_NAME
/*00D*/ BYTE         HandleDBOffset;     // -> OBJECT_HANDLE_DB
/*00E*/ BYTE         QuotaChargesOffset; // -> OBJECT_QUOTA_CHARGES
/*00F*/ BYTE         ObjectFlags;        // OB_FLAG_*
/*010*/ union
            { // OB_FLAG_CREATE_INFO ? ObjectCreateInfo : QuotaBlock
/*010*/     PQUOTA_BLOCK        QuotaBlock;
/*010*/     POBJECT_CREATE_INFO ObjectCreateInfo;
/*014*/     };
/*014*/ PSECURITY_DESCRIPTOR SecurityDescriptor;
/*018*/ }
        OBJECT_HEADER,
     * POBJECT_HEADER,
    **PPOBJECT_HEADER;
```

LISTING 7-2. *The* `OBJECT_HEADER` *Structure*

- The ObjectType member points to an OBJECT_TYPE structure (described later) representing the type object that has been used in the creation of this object.
- The NameOffset specifies the number of bytes to be subtracted from the OBJECT_HEADER address to locate the object header's OBJECT_NAME portion. If zero, this structure is not available.
- The HandleDBOffset specifies the number of bytes to be subtracted from the OBJECT_HEADER address to locate the object header's OBJECT_HANDLE_DB portion. If zero, this structure is not available.
- The QuotaChargesOffset specifies the number of bytes to be subtracted from the OBJECT_HEADER address to locate the object header's OBJECT_QUOTA_CHARGES portion. If zero, this structure is not available.
- The ObjectFlags specify various binary properties of an object, as listed in the top section of Listing 7-2. If the OB_FLAG_CREATOR_INFO bit is set, the object header includes an OBJECT_CREATOR_INFO structure that immediately precedes the OBJECT_HEADER. In Windows NT/2000 Native API Reference, Gary Nebbett mentions these flags with slightly different names in his description of the SystemObjectInformation class of the ZwQuerySystemInformation() function (Nebbett 2000, p. 24), as shown in Table 7-3.
- The QuotaBlock and ObjectCreateInfo members are mutually exclusive. If the ObjectFlags member has the OB_FLAG_CREATE_INFO flag set, this member contains a pointer to the OBJECT_CREATE_INFO structure (described later) used in the creation of this object. Otherwise, it points to a QUOTA_BLOCK that provides information about the usage of the paged and nonpaged memory pools. Many objects have their QuotaBlock pointer set to the internal PspDefaultQuotaBlock structure. The value of this union can be NULL.
- The SecurityDescriptor member points to a SECURITY_DESCRIPTOR structure if the OB_FLAG_SECURITY bit of the ObjectFlags is set. Otherwise, its value is NULL.

In the above list, several structures have been mentioned that weren't discussed in detail so far. Each of them will be introduced now, starting with the four optional header parts shown in Figure 7-1.

TABLE 7-3. *Comparison of* ObjectFlags *Interpretations*

SCHREIBER	VALUE	NEBBETT
OB_FLAG_CREATE_INFO	0x01	N/A
OB_FLAG_KERNEL_MODE	0x02	KERNEL_MODE
OB_FLAG_CREATOR_INFO	0x04	CREATOR_INFO
OB_FLAG_EXCLUSIVE	0x08	EXCLUSIVE
OB_FLAG_PERMANENT	0x10	PERMANENT
OB_FLAG_SECURITY	0x20	DEFAULT_SECURITY_QUOTA
OB_FLAG_SINGLE_PROCESS	0x40	SINGLE_HANDLE_ENTRY

THE OBJECT CREATOR INFORMATION

The `OBJECT_HEADER` of an object is immediately preceded by an `OBJECT_CREATOR_INFO` structure if the `OB_FLAG_CREATOR_INFO` bit of its `ObjectFlags` member is set. The definition of this optional header part is shown in Listing 7-3. The `ObjectList` member is a node within a doubly linked list (cf. Listing 2-7 in Chapter 2) that connects objects of the same type to each other. As usual, this list is circular. The list head where the object list originates and ends is located within the `OBJECT_TYPE` structure that represents the common type object of the list members. By default, only `Port` and `WaitablePort` objects include `OBJECT_CREATOR_INFO` data in their headers. The `SystemObjectInformation` class of the `ZwQuerySystemInformation()` API function uses the `ObjectList` to return complete lists of currently allocated objects, grouped by object type. Gary Nebbett points out in `Windows NT/2000 Native API Reference` that "[...] this information class is only available if `FLG_MAINTAIN_OBJECT_TYPELIST` was set in the `NtGlobalFlag`s at boot time" (Nebbett 2000, p. 25).

```
typedef struct _OBJECT_CREATOR_INFO
        {
/*000*/ LIST_ENTRY ObjectList;      // OBJECT_CREATOR_INFO
/*008*/ HANDLE     UniqueProcessId;
/*00C*/ WORD       Reserved1;
/*00E*/ WORD       Reserved2;
/*010*/ }
        OBJECT_CREATOR_INFO,
     * POBJECT_CREATOR_INFO,
    **PPOBJECT_CREATOR_INFO;
```

LISTING 7-3. *The* `OBJECT_CREATOR_INFO` *Structure*

The UniqueProcessId is the zero-based numeric ID of the process that created the object. Although defined as a HANDLE, this member is not a handle in the usual sense. It might be described more accurately as an opaque 32-bit unsigned integer. Actually, the Win32 GetCurrentProcessId() API function returns these HANDLE values as DWORD types.

THE OBJECT NAME

If the NameOffset member of the OBJECT_HEADER is nonzero, it specifies the inverse offset of an OBJECT_NAME structure with respect to the base address of the OBJECT_HEADER. Typical values are 0x10 or 0x20, depending on the presence of an OBJECT_CREATOR_INFO header part. Listing 7-4 shows the definition of the OBJECT_NAME structure. The Name member is a UNICODE_STRING whose Buffer member points to the name string, which is usually not part of the memory block containing the object. Not all named objects use an OBJECT_NAME structure in the header to store the name. For example, some objects rely on a QueryNameProcedure() provided by their associated OBJECT_TYPE.

If the Directory member is not NULL, it points to the directory object representing the layer in the system's object hierarchy where this object is located. Like files in a file system, Windows 2000 objects are kept in a hierarchically structured tree consisting of directory and leaf objects. More details about the OBJECT_DIRECTORY structure follow in a moment.

```
typedef struct _OBJECT_NAME
        {
/*000*/ POBJECT_DIRECTORY Directory;
/*004*/ UNICODE_STRING    Name;
/*00C*/ DWORD             Reserved;
/*010*/ }
        OBJECT_NAME,
      * POBJECT_NAME,
     **PPOBJECT_NAME;
```

LISTING 7-4. *The* OBJECT_NAME *Structure*

THE OBJECT HANDLE DATABASE

Some objects maintain process-specific handle counts stored in a so-called "handle database." If this is the case, the HandleDBOffset member of the OBJECT_HEADER contains a nonzero value. Just like the NameOffset described above, this is an offset to be subtracted from the base address of the OBJECT_HEADER to locate this header part. The OBJECT_HANDLE_DB structure is defined in Listing 7-5. If the OB_FLAG_SINGLE_PROCESS flag is set in the ObjectFlags, the Process member of the union at the beginning of this structure is valid and points to a process object. If more that one process holds handles to the object, the OB_FLAG_SINGLE_PROCESS flag is cleared, and the HandleDBList member becomes valid, pointing to an OBJECT_HANDLE_DB_LIST that constitutes an array of OBJECT_HANDLE_DB structures, preceded by a count value.

```
typedef struct _OBJECT_HANDLE_DB
        {
/*000*/ union
            {
/*000*/     struct _EPROCESS              *Process;
/*000*/     struct _OBJECT_HANDLE_DB_LIST *HandleDBList;
/*004*/     };
/*004*/ DWORD HandleCount;
/*008*/ }
        OBJECT_HANDLE_DB,
     * POBJECT_HANDLE_DB,
    **PPOBJECT_HANDLE_DB;

#define OBJECT_HANDLE_DB_ \
        sizeof (OBJECT_HANDLE_DB)

// -----------------------------------------------------------------

typedef struct _OBJECT_HANDLE_DB_LIST
        {
/*000*/ DWORD            Count;
/*004*/ OBJECT_HANDLE_DB Entries [];
/*???*/ }
        OBJECT_HANDLE_DB_LIST,
     * POBJECT_HANDLE_DB_LIST,
    **PPOBJECT_HANDLE_DB_LIST;

#define OBJECT_HANDLE_DB_LIST_ \
        sizeof (OBJECT_HANDLE_DB_LIST)
```

LISTING 7-5. *The* OBJECT_HANDLE_DB *Structure*

RESOURCE CHARGES AND QUOTAS

If a process opens a handle to an object, the process must "pay" for usage of system resources caused by this operation. The paid dues are referred to as charges, and the upper limit a process may spend for resources is termed the `quota`. In the glossary of the DDK documentation (Microsoft, 2000F), Microsoft defines the "quota" term in the following way:

QUOTA

> *A per-process limit on the use of system resources.*
> *For each process, Windows NT®/Windows® 2000 sets limits on certain system resources the process's threads can use, including quotas for paging-file, paged-pool, and nonpaged-pool usage, etc. For example, the Memory Manager "charges quota" against the process as its threads use page-file, paged-pool, or nonpaged-pool memory; it also updates these values when threads release memory.* (Windows 2000 DDK \ Kernel-Mode Drivers \ Design Guide \ Kernel-Mode Glossary \ Q \ quota)

By default, an object's `OBJECT_TYPE` determines the charges to be applied for paged/nonpaged pool usage and security. However, this default can be overridden by adding an `OBJECT_QUOTA_CHARGES` structure to the object header. The location of this data relative to the `OBJECT_HEADER` base address is specified by the `QuotaChargesOffset` member of the `OBJECT_HEADER` as an inverse offset, as usual. Listing 7-6 shows the structure definition. The usages of the paged and nonpaged pools are charged separately. If the object requires security, an additional `SecurityCharge` is added to the paged-pool usage. The default security charge is `0x800`.

If the `OB_FLAG_CREATE_INFO` bit of the `ObjectFlags` in the `OBJECT_HEADER` is zero, the `QuotaBlock` member points to a `QUOTA_BLOCK` structure (Listing 7-7) that contains statistical information about the current resource usage of the object.

```
#define OB_SECURITY_CHARGE 0x00000800

typedef struct _OBJECT_QUOTA_CHARGES
        {
/*000*/ DWORD PagedPoolCharge;
/*004*/ DWORD NonPagedPoolCharge;
/*008*/ DWORD SecurityCharge;
/*00C*/ DWORD Reserved;
/*010*/ }
        OBJECT_QUOTA_CHARGES,
      * POBJECT_QUOTA_CHARGES,
     **PPOBJECT_QUOTA_CHARGES;
```

LISTING 7-6. *The* `OBJECT_QUOTA_CHARGES` *Structure*

```
typedef struct _QUOTA_BLOCK
        {
/*000*/ DWORD Flags;
/*004*/ DWORD ChargeCount;
/*008*/ DWORD PeakPoolUsage [2]; // NonPagedPool, PagedPool
/*010*/ DWORD PoolUsage     [2]; // NonPagedPool, PagedPool
/*018*/ DWORD PoolQuota     [2]; // NonPagedPool, PagedPool
/*020*/ }
        QUOTA_BLOCK,
     * PQUOTA_BLOCK,
    **PPQUOTA_BLOCK;
```

LISTING 7-7. *The* `QUOTA_BLOCK` *Structure*

OBJECT DIRECTORIES

As already noted in the discussion of the `OBJECT_NAME` header part, the Windows 2000 object manager keeps individual objects in a tree of `OBJECT_DIRECTORY` structures, also known as "directory objects." An `OBJECT_DIRECTORY` is just another fancy type of object, with an ordinary `OBJECT_HEADER` and everything a real object needs. The Windows 2000 object directory management is quite tricky. As Listing 7-8 shows, the `OBJECT_DIRECTORY` is basically a hash table with 37 entries. This unusual size has probably been chosen because it is a prime number. Each table entry can hold a pointer to an `OBJECT_DIRECTORY_ENTRY` whose `Object` member refers to an object. When a new object is created, the object manager computes a hash value in the range 0 to 36 from the object name and creates an `OBJECT_DIRECTORY_ENTRY`. If the target slot of the hash table is empty, this slot is set up to point to the new directory entry. If the slot is already in use, the new entry is inserted into a singly-linked list of entries originating from the target slot, using the `NextEntry` members of the involved `OBJECT_DIRECTORY_ENTRY` structures. To represent hierarchical object relationships, object directories can be nested in a straightforward way by simply adding an `OBJECT_DIRECTORY_ENTRY` with an `Object` member that points to a subordinate directory object.

To optimize the access to frequently used objects, the object manager applies a simple most recently used (MRU) algorithm. Whenever an object has successfully been retrieved, it is put in front of the linked list of entries that are assigned to the same hash table slot. Moreover, a pointer to the updated list is kept in the `CurrentEntry` member of the `OBJECT_DIRECTORY`. The `CurrentEntryValid` flag indicates whether the `CurrentEntry` pointer is valid. Access to the system's global object directory is synchronized by means of an `ERESOURCE` lock called `ObpRootDirectoryMutex`. This lock is neither documented nor exported.

```
typedef struct _OBJECT_DIRECTORY_ENTRY
        {
/*000*/ struct _OBJECT_DIRECTORY_ENTRY *NextEntry;
/*004*/ POBJECT                         Object;
/*008*/ }
        OBJECT_DIRECTORY_ENTRY,
     * POBJECT_DIRECTORY_ENTRY,
    **PPOBJECT_DIRECTORY_ENTRY;

// -----------------------------------------------------------------

#define OBJECT_HASH_TABLE_SIZE 37

typedef struct _OBJECT_DIRECTORY
        {
/*000*/ POBJECT_DIRECTORY_ENTRY HashTable [OBJECT_HASH_TABLE_SIZE];
/*094*/ POBJECT_DIRECTORY_ENTRY CurrentEntry;
/*098*/ BOOLEAN                 CurrentEntryValid;
/*099*/ BYTE                    Reserved1;
/*09A*/ WORD                    Reserved2;
/*09C*/ DWORD                   Reserved3;
/*0A0*/ }
        OBJECT_DIRECTORY,
     * POBJECT_DIRECTORY,
    **PPOBJECT_DIRECTORY;
```

LISTING 7-8. *The* `OBJECT_DIRECTORY` *and* `OBJECT_DIRECTORY_ENTRY` *Structures*

OBJECT TYPES

The above object header part descriptions have frequently referred to "type objects" or `OBJECT_TYPE` structures, so it is now time to introduce these. Formally, a type object is nothing but a special kind of object, such as an event, device, or process, and as such has an `OBJECT_HEADER` and potentially some of the optional header substructures. The only difference is that type objects are related in a special way to other objects. A type object is sort of a "master object" that defines common properties of objects of the same kind, and optionally keeps all of its subordinate objects in a doubly-linked list, as explained earlier in the description of the `OBJECT_CREATOR_INFO` structure. Therefore, type objects are frequently referred to as "object types" to emphasize that they are more than just ordinary objects.

The body of a type object consists of an `OBJECT_TYPE` structure with an embedded `OBJECT_TYPE_INITIALIZER`, both of which are shown in Listing 7-9. The latter is used during object creation via `ObCreateObject()` to build a proper object header. For example, the `MaintainHandleCount` and `MaintainTypeList` members are used

```
typedef struct _OBJECT_TYPE_INITIALIZER
        {
/*000*/ WORD            Length;          //0x004C
/*002*/ BOOLEAN         UseDefaultObject;//OBJECT_TYPE.DefaultObject
/*003*/ BOOLEAN         Reserved1;
/*004*/ DWORD           InvalidAttributes;
/*008*/ GENERIC_MAPPING GenericMapping;
/*018*/ ACCESS_MASK     ValidAccessMask;
/*01C*/ BOOLEAN         SecurityRequired;
/*01D*/ BOOLEAN         MaintainHandleCount; // OBJECT_HANDLE_DB
/*01E*/ BOOLEAN         MaintainTypeList;    // OBJECT_CREATOR_INFO
/*01F*/ BYTE            Reserved2;
/*020*/ BOOL            PagedPool;
/*024*/ DWORD           DefaultPagedPoolCharge;
/*028*/ DWORD           DefaultNonPagedPoolCharge;
/*02C*/ NTPROC          DumpProcedure;
/*030*/ NTPROC          OpenProcedure;
/*034*/ NTPROC          CloseProcedure;
/*038*/ NTPROC          DeleteProcedure;
/*03C*/ NTPROC_VOID     ParseProcedure;
/*040*/ NTPROC_VOID     SecurityProcedure; // SeDefaultObjectMethod
/*044*/ NTPROC_VOID     QueryNameProcedure;
/*048*/ NTPROC_BOOLEAN  OkayToCloseProcedure;
/*04C*/ }
        OBJECT_TYPE_INITIALIZER,
     * POBJECT_TYPE_INITIALIZER,
    **PPOBJECT_TYPE_INITIALIZER;

// ---------------------------------------------------------------

typedef struct _OBJECT_TYPE
        {
/*000*/ ERESOURCE      Lock;
/*038*/ LIST_ENTRY     ObjectListHead; // OBJECT_CREATOR_INFO
/*040*/ UNICODE_STRING ObjectTypeName; // see above
/*048*/ union
            {
/*048*/     PVOID DefaultObject; // ObpDefaultObject
/*048*/     DWORD Code;          // File: 5C, WaitablePort: A0
            };
/*04C*/ DWORD                   ObjectTypeIndex; // OB_TYPE_INDEX_*
/*050*/ DWORD                   ObjectCount;
/*054*/ DWORD                   HandleCount;
/*058*/ DWORD                   PeakObjectCount;
/*05C*/ DWORD                   PeakHandleCount;
/*060*/ OBJECT_TYPE_INITIALIZER ObjectTypeInitializer;
/*0AC*/ DWORD                   ObjectTypeTag;   // OB_TYPE_TAG_*
/*0B0*/ }
```

```
        OBJECT_TYPE,
      * POBJECT_TYPE,
     **PPOBJECT_TYPE;
```

LISTING 7-9. *The* `OBJECT_TYPE` *and* `OBJECT_TYPE_INITIALIZER` *Structures*

by the internal `ntoskrnl.exe` function `ObpAllocateObject()` to decide whether all newly created objects will comprise `OBJECT_HANDLE_DB` and `OBJECT_CREATOR_INFO` header parts, respectively. Setting the `MaintainTypeList` flag has the nice side effect that the objects of this type will be tied to each other in a doubly linked list, originating from and ending at the `ObjectListHead` member of the `OBJECT_TYPE`. The `OBJECT_TYPE_INITIALIZER` also provides the default quota charges (mentioned earlier in the discussion of the `OBJECT_QUOTA_CHARGES` header component) via its `Default-PagedPoolCharge` and `DefaultNonPagedPoolCharge` members.

Because type objects/object types are essential building blocks of the Windows 2000 object universe, `ntoskrnl.exe` stores them in named variables, making it easy to verify the type of an object by simply comparing the `ObjectType` member of its `OBJECT_HEADER` to the stored type object in question. Type objects are unique—the system never creates more than one type object for each kind of object. Table 7-4 summarizes the type objects maintained by Windows 2000. The information in the various columns has the following meaning:

TABLE 7-4. *Available Object Types*

INDEX	TAG	NAME	C STRUCTURE	PUBLIC	SYMBOL
1	"ObjT"	"Type"	OBJECT_TYPE	No	ObpTypeObjectType
2	"Dire"	"Directory"	OBJECT_DIRECTORY	No	ObpDirectoryObjectType
3	"Symb"	"SymbolicLink"		No	ObpSymbolicLinkObjectType
4	"Toke"	"Token"	TOKEN	No	SepTokenObjectType
5	"Proc"	"Process"	EPROCESS	Yes	PsProcessType
6	"Thre"	"Thread"	ETHREAD	Yes	PsThreadType
7	"Job "	"Job"		Yes	PsJobType
8	"Even"	"Event"	KEVENT	Yes	ExEventObjectType
9	"Even"	"EventPair"	KEVENT_PAIR	No	ExEventPairObjectType
10	"Muta"	"Mutant"	KMUTANT	No	ExMutantObjectType
11	"Call"	"Callback"	CALLBACK_OBJECT	No	ExCallbackObjectType

(continued)

TABLE 7-4. *(continued)*

INDEX	TAG	NAME	C STRUCTURE	PUBLIC	SYMBOL
12	"Sema"	"Semaphore"	KSEMAPHORE	Yes	ExSemaphoreObjectType
13	"Time"	"Timer"	ETIMER	No	ExTimerObjectType
14	"Prof"	"Profile"	KPROFILE	No	ExProfileObjectType
15	"Wind"	"WindowStation"		Yes	ExWindowStationObjectType
16	"Desk"	"Desktop"		Yes	ExDesktopObjectType
17	"Sect"	"Section"		Yes	MmSectionObjectType
18	"Key"	"Key"		No	CmpKeyObjectType
19	"Port"	"Port"		Yes	LpcPortObjectType
20	"Wait"	"WaitablePort"		No	LpcWaitablePortObjectType
21	"Adap"	"Adapter"	ADAPTER_OBJECT	Yes	IoAdapterObjectType
22	"Cont"	"Controller"	CONTROLLER_OBJECT	No	IoControllerObjectType
23	"Devi"	"Device"	DEVICE_OBJECT	Yes	IoDeviceObjectType
24	"Driv"	"Driver"	DRIVER_OBJECT	Yes	IoDriverObjectType
25	"IoCo"	"IoCompletion"	IO_COMPLETION	No	IoCompletionObjectType
26	"File"	"File"	FILE_OBJECT	Yes	IoFileObjectType
27	"WmiG"	"WmiGuid"	GUID	No	WmipGuidObjectType

- The "Index" column specifies the value of the `ObjectTypeIndex` member of the `OBJECT_TYPE` structure.
- The "Tag" is the 32-bit identifier stored in the `ObjectTypeTag` member of the `OBJECT_TYPE` structure. Windows 2000 tags are typically binary values generated by concatenation of four ANSI characters. During debugging, these characters can easily be identified in a hex dump listing. Testing the `ObjectTypeTag` value is the easiest way to verify that a given type object is of the expected kind. When allocating memory for an object, Windows 2000 also uses this value—logically OR'ed with `0x80000000`—to tag the new memory block.
- The "Name" column states the object name, as it is specified by the type object's `OBJECT_NAME` header component. It is obvious that the type tag is generated from the object name by truncating it to four characters, appending spaces if the name is shorter.
- "C Structure" is the name of the object body structure associated with the object type. Some of them are documented in the DDK and some in the

w2k_def.h header file on the CD provided with this book. If no name is present, the structure is currently unknown or unidentified.

- The "Symbol" column indicates the name of the pointer variable that refers to the type object. If the "Public" column contains "yes," the variable is exported and can be accessed by kernel-mode drivers or applications that link to the kernel via the w2k_call.dll library presented in Chapter 6.

The "Index" column requires further explanation. The value shown here is taken from the ObjectTypeIndex member of the corresponding OBJECT_TYPE structure. This value is not a predefined type ID as are the DISP_TYPE_* and IO_TYPE_* constants used by dispatcher and I/O objects (see Tables 7-1 and 7-2). It merely reflects the order in which the system created these type objects. Therefore, you should never use the ObjectTypeIndex to identify the type of an object. It is safer to use the ObjectTypeTag instead, which is certainly more stable across future operating system versions.

OBJECT HANDLES

Whereas a kernel-mode driver can directly contact an object by querying a pointer to its object body, a user-mode application cannot. When it calls one of the API functions that open an object, it receives back a handle that must be used in subsequent operations on the object. Although Windows 2000 applies the "handle" metaphor to a variety of things that are not necessarily related, there is a construct that can be called the handle in the strictest sense. This pure form of a handle is a process-specific 16-bit number that is usually a multiple of four and constitutes an index into a handle table maintained by the kernel for each process. The main HANDLE_TABLE structure is shown at the end Listing 7-10. This table points to a HANDLE_LAYER1 structure that consists of pointers to HANDLE_LAYER2 structures, which in turn are composed of HANDLE_LAYER3 pointers. Finally, the third indirection layer contains pointers to the actual handle table entries, represented by HANDLE_ENTRY structures.

```
// HANDLE BIT-FIELDS
// -----------------
//  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
//  F E D C B A 9 8 7 6 5 4 3 2 1 0 F E D C B A 9 8 7 6 5 4 3 2 1 0
// _________________________________________________________________
// |x|x|x|x|x|x|a|a|a|a|a|a|a|a|b|b|b|b|b|b|b|b|c|c|c|c|c|c|c|c|y|y|
```

(continued)

```
// | not used  | HANDLE_LAYER1 | HANDLE_LAYER2 | HANDLE_LAYER3 |tag|

#define HANDLE_LAYER_SIZE 0x00000100

// -----------------------------------------------------------------

#define HANDLE_ATTRIBUTE_INHERIT 0x00000002
#define HANDLE_ATTRIBUTE_MASK    0x00000007
#define HANDLE_OBJECT_MASK       0xFFFFFFF8

typedef struct _HANDLE_ENTRY // cf. OBJECT_HANDLE_INFORMATION
        {
/*000*/ union
            {
/*000*/     DWORD          HandleAttributes;// HANDLE_ATTRIBUTE_MASK
/*000*/     POBJECT_HEADER ObjectHeader;    // HANDLE_OBJECT_MASK
/*004*/     };
/*004*/ union
            {
/*004*/     ACCESS_MASK    GrantedAccess;   // if used entry
/*004*/     DWORD          NextEntry;       // if free entry
/*008*/     };
/*008*/ }
        HANDLE_ENTRY,
     * PHANDLE_ENTRY,
    **PPHANDLE_ENTRY;

// -----------------------------------------------------------------

typedef struct _HANDLE_LAYER3
        {
/*000*/ HANDLE_ENTRY Entries [HANDLE_LAYER_SIZE]; // bits 2 to 9
/*800*/ }
        HANDLE_LAYER3,
     * PHANDLE_LAYER3,
    **PPHANDLE_LAYER3;

// -----------------------------------------------------------------

typedef struct _HANDLE_LAYER2
        {
/*000*/ PHANDLE_LAYER3 Layer3 [HANDLE_LAYER_SIZE]; // bits 10 to 17
/*400*/ }
        HANDLE_LAYER2,
     * PHANDLE_LAYER2,
    **PPHANDLE_LAYER2;

// -----------------------------------------------------------------
```

```
typedef struct _HANDLE_LAYER1
        {
/*000*/ PHANDLE_LAYER2 Layer2 [HANDLE_LAYER_SIZE]; // bits 18 to 25
/*400*/ }
        HANDLE_LAYER1,
      * PHANDLE_LAYER1,
     **PPHANDLE_LAYER1;

// -----------------------------------------------------------------

typedef struct _HANDLE_TABLE
        {
/*000*/ DWORD             Reserved;
/*004*/ DWORD             HandleCount;
/*008*/ PHANDLE_LAYER1    Layer1;
/*00C*/ struct _EPROCESS *Process; // passed to PsChargePoolQuota ()
/*010*/ HANDLE            UniqueProcessId;
/*014*/ DWORD             NextEntry;
/*018*/ DWORD             TotalEntries;
/*01C*/ ERESOURCE         HandleTableLock;
/*054*/ LIST_ENTRY        HandleTableList;
/*05C*/ KEVENT            Event;
/*06C*/ }
        HANDLE_TABLE,
      * PHANDLE_TABLE,
     **PPHANDLE_TABLE;
```

LISTING 7-10. *Handle Tables, Layers, and Entries*

This three-layered addressing mechanism is a clever trick to be able to dynamically increase or decrease the storage needed for handle entries with minimum effort while also minimizing waste of memory. Because each handle table layer takes up to 256 pointers, a process can theoretically open `256 * 256 * 256, or 16,777,216` handles. With each handle entry consuming 8 bytes, the required maximum storage amounts to 128 MB. However, because a process rarely needs that many handles, it would be an immense waste of space to allocate the complete handle table from the start. The three-layered approach used by Windows 2000 starts out with the minimum set of a single subtable per layer. Not counting the `HANDLE_TABLE` itself, the required storage is `256 * 4 + 256 * 4 + 256 * 8, or 4,096` bytes. The initial handle table material fits exactly into a single physical memory page.

To look up the `HANDLE_ENTRY` of a `HANDLE,` the system divides the 32-bit value of the handle into three 8-bit fragments, discarding bits #0 and #1, as well as the topmost six bits. Given these three fragments, the handle resolution mechanism proceeds as follows:

1. Bits #18 to #25 of the HANDLE are used as an index into the Layer2 array of the HANDLE_LAYER1 block referred to by the Layer1 member of the HANDLE_TABLE.
2. Bits #10 to #17 of the HANDLE are used as an index into the Layer3 array of the HANDLE_LAYER2 block retrieved in the previous step.
3. Bits #2 to #9 of the HANDLE are used as an index into the Entries array of the HANDLE_LAYER3 block retrieved in the previous step.
4. The HANDLE_ENTRY retrieved in the previous step provides a pointer to the OBJECT_HEADER (see Listing 7-2) of the object associated to the HANDLE.

If this sounds confusing, Figure 7-2 may clarify what occurs in this situation. Actually, Figure 7-2 is remarkably similar in structure to Figure 4-3 in Chapter 4, where the i386 CPU's linear-to-physical address translation is depicted. Both algorithms break an input value into three fragments, with two of them used as offsets into two hierarchically arranged indirection layers and the third one selecting an entry from the target layer. Note that the layered handle table model is new to Windows 2000. Windows NT 4.0 provided a single-layered table that had to be expanded if the currently opened handles didn't fit into the memory block currently allocated for the handle table (cf. Custer 1993, Solomon 1998).

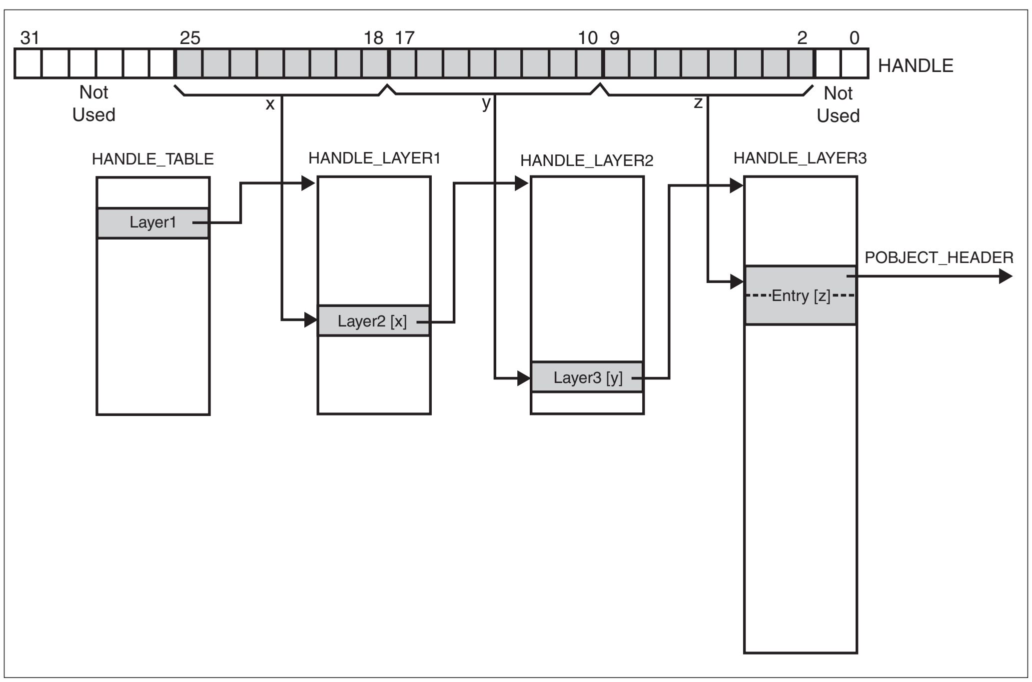

FIGURE 7-2. HANDLE *to* OBJECT_HEADER *Resolution*

Because each process has its own handle table, the kernel must somehow keep track of the currently allocated tables. Therefore, ntoskrnl.exe maintains a LIST_ENTRY variable named HandleTableListHead that is the root of a doubly linked list of HANDLE_TABLE structures, chained together by means of their HandleTableList members. When following their Flink or Blink pointers, you must always subtract the HandleTableList member offset 0x54 to get to the base address of the surrounding HANDLE_TABLE structure. The owning process of each table can easily be determined by consulting its UniqueProcessId member. The first HANDLE_TABLE in the list is usually owned by the System process (ID=8), followed by the table of the System Idle Process (ID=0). The latter HANDLE_TABLE is also reachable by an internal variable referred to as ObpKernelHandleTable.

When accessing handle tables, the system uses a couple of synchronization objects to preserve data integrity in multithreaded handle access scenarios. The entire handle table list is locked by means of the global HandleTableListLock inside ntoskrnl.exe, which is an ERESOURCE structure. This type of synchronization object allows exclusive or shared locks, acquired with the help of the ExAcquireResourceExclusiveLite() and ExAcquireResourceSharedLite() API functions, respectively. The lock is released by calling ExReleaseResourceLite(). After locking the handle table list for exclusive access, you are guaranteed that the system will not change any list entries until the lock is released. Each HANDLE_TABLE in the list entry has its own ERESOURCE lock, termed HandleTableLock in Listing 7-10. ntoskrnl.exe provides the internal functions ExLockHandleTableExclusive() and ExLockHandleTableShared() to acquire this ERESOURCE, and ExUnlockHandleTableShared() to release it (no matter whether the lock is exclusive or shared, even though the name suggests that it is good for shared locks only). These functions are simply wrappers around ExAcquireResourceExclusiveLite(), ExAcquireResourceSharedLite(), and ExReleaseResourceLite(), taking a pointer to a HANDLE_TABLE and passing over its HandleTableLock.

Unfortunately, all essential functions and global variables used by the kernel's handle manager are not only undocumented, but also inaccessible because they are not exported by the ntoskrnl.exe module. Although it is certainly possible to look up objects by their handles using the kernel call interface proposed in Chapter 6 and the scheme outlined in Figure 7-2, I don't recommend doing so. One reason is that this code would deliberately give up compatibility with Windows NT 4.0 because of the radical handle table design change. Another reason is that the kernel provides a luxurious function that returns the contents of all handle tables owned by the currently active processes. This function is NtQuerySystemInformation(), and the information class required to obtain the handle information is SystemHandleInformation (16). Please refer to Schreiber (1999) or Nebbett (2000) for extensive details on how to issue this API call. The SystemHandleInformation data are obtained from the internal function ExpGetHandleInformation() that relies on ObGetHandleInformation(). The latter in turn calls ExSnapShotHandleTables(), where the handle table list enumeration is ultimately performed. ExSnapShotHandleTables() expects a pointer to a callback function that is called for each HANDLE_ENTRY referring to an object. ObGetHandleInformation() uses the internal ObpCaptureHandleInformation() callback function to fill the caller's buffer with an array of structures containing information about each handle currently maintained by the system.

PROCESS AND THREAD OBJECTS

Probably the most interesting and complex inhabitants of the Windows 2000 object world are the process and thread objects. These are usually the top-level entities a software developer must deal with. A kernel-mode component always runs in the context of a thread, and this thread is often part of a user process. Therefore, it is quite natural that process and thread objects are object types that frequently are explored in debugging situations. The Windows 2000 Kernel Debugger accounts for this requirement by providing the "bang" commands !processfields and !threadfields, exported by the debugger extension kdextx86.dll. Both commands output a simple list of name/offset pairs describing the members of the EPROCESS and ETHREAD structures, respectively (cf. Examples 1-1 and 1-2 in Chapter 1). These object structures are undocumented, so these debugger commands are currently the only official source of information about them.

Unfortunately, the !processfields output (cf. Example 1-1) starts with a member named Pcb that refers to a substructure comprising 0x6C bytes, because the next member ExitStatus is located at this offset. Pcb is a KPROCESS structure that is completely undocumented. This arrangement is interesting: Obviously, a process is represented by a smaller kernel object embedded in a larger executive object. This nesting scheme reappears with the thread object. The debugger's !threadfields command (cf. Example 1-2) reveals a Tcb member of no less than 0x1B0 bytes at the beginning of the ETHREAD structure. This is a KTHREAD structure, representing another kernel object inside an executive object.

Although it is helpful that the Kernel Debugger provides symbolic information about the executive's process and thread objects, the plain member names do not necessarily provide enough cues to identify the members' data types. Moreover, the opacity of the Pcb and Tcb members makes it quite difficult to understand the nature of these objects. In a disassembly listing generated by the Kernel Debugger, you will frequently see instructions referencing data within the confines of these opaque members. The used offsets are completely useless without information about the name and type of the referenced data. Therefore, I have collected information from various sources plus results of my investigation, to figure out what these objects look like. Part one of the results is shown in Listings 7-11 and 7-12, defining the `KPROCESS` and `KTHREAD` structures, respectively. The `DISPATCHER_HEADER` at the beginning of both objects qualifies processes and threads as dispatcher objects, which in turn means they can be waited for using `KeWaitForSingleObject()` and `KeWaitForMultipleObjects()`. A thread object becomes signaled after execution of the thread has ceased, and a process object enters the signaled state after all of its threads have terminated. This is nothing new for Win32 programmers—it is quite common to wait for termination of a process spawned by another process by means of the Win32 API function `WaitForSingleObject()`. However, now you finally know why waiting for processes and threads is possible in the first place.

```
typedef struct _KPROCESS
        {
/*000*/ DISPATCHER_HEADER Header; // DO_TYPE_PROCESS (0x1B)
/*010*/ LIST_ENTRY        ProfileListHead;
/*018*/ DWORD             DirectoryTableBase;
/*01C*/ DWORD             PageTableBase;
/*020*/ KGDTENTRY         LdtDescriptor;
/*028*/ KIDTENTRY         Int21Descriptor;
/*030*/ WORD              IopmOffset;
/*032*/ BYTE              Iopl;
/*033*/ BOOLEAN           VdmFlag;
/*034*/ DWORD             ActiveProcessors;
/*038*/ DWORD             KernelTime; // ticks
/*03C*/ DWORD             UserTime;   // ticks
/*040*/ LIST_ENTRY        ReadyListHead;
/*048*/ LIST_ENTRY        SwapListEntry;
/*050*/ LIST_ENTRY        ThreadListHead; // KTHREAD.ThreadListEntry
/*058*/ PVOID             ProcessLock;
/*05C*/ KAFFINITY         Affinity;
/*060*/ WORD              StackCount;
/*062*/ BYTE              BasePriority;
/*063*/ BYTE              ThreadQuantum;
/*064*/ BOOLEAN           AutoAlignment;
/*065*/ BYTE              State;
/*066*/ BYTE              ThreadSeed;
/*067*/ BOOLEAN           DisableBoost;
/*068*/ DWORD             d68;
/*06C*/ }
        KPROCESS,
     * PKPROCESS,
    **PPKPROCESS;
```

LISTING 7-11. *The* `KPROCESS` *Object Structure*

```
typedef struct _KTHREAD
        {
/*000*/ DISPATCHER_HEADER         Header; // DO_TYPE_THREAD (0x6C)
/*010*/ LIST_ENTRY                MutantListHead;
/*018*/ PVOID                     InitialStack;
/*01C*/ PVOID                     StackLimit;
/*020*/ struct _TEB              *Teb;
/*024*/ PVOID                     TlsArray;
/*028*/ PVOID                     KernelStack;
/*02C*/ BOOLEAN                   DebugActive;
/*02D*/ BYTE                      State; // THREAD_STATE_*
/*02E*/ BOOLEAN                   Alerted;
/*02F*/ BYTE                      bReserved01;
/*030*/ BYTE                      Iopl;
/*031*/ BYTE                      NpxState;
/*032*/ BYTE                      Saturation;
/*033*/ BYTE                      Priority;
/*034*/ KAPC_STATE                ApcState;
/*04C*/ DWORD                     ContextSwitches;
/*050*/ DWORD                     WaitStatus;
/*054*/ BYTE                      WaitIrql;
/*055*/ BYTE                      WaitMode;
/*056*/ BYTE                      WaitNext;
/*057*/ BYTE                      WaitReason;
/*058*/ PLIST_ENTRY               WaitBlockList;
/*05C*/ LIST_ENTRY                WaitListEntry;
/*064*/ DWORD                     WaitTime;
/*068*/ BYTE                      BasePriority;
/*069*/ BYTE                      DecrementCount;
/*06A*/ BYTE                      PriorityDecrement;
/*06B*/ BYTE                      Quantum;
/*06C*/ KWAIT_BLOCK               WaitBlock [4];
/*0CC*/ DWORD                     LegoData;
/*0D0*/ DWORD                     KernelApcDisable;
/*0D4*/ KAFFINITY                 UserAffinity;
/*0D8*/ BOOLEAN                   SystemAffinityActive;
/*0D9*/ BYTE                      Pad [3];
/*0DC*/ PSERVICE_DESCRIPTOR_TABLE pServiceDescriptorTable;
/*0E0*/ PVOID                     Queue;
/*0E4*/ PVOID                     ApcQueueLock;
/*0E8*/ KTIMER                    Timer;
/*110*/ LIST_ENTRY                QueueListEntry;
/*118*/ KAFFINITY                 Affinity;
/*11C*/ BOOLEAN                   Preempted;
/*11D*/ BOOLEAN                   ProcessReadyQueue;
/*11E*/ BOOLEAN                   KernelStackResident;
/*11F*/ BYTE                      NextProcessor;
/*120*/ PVOID                     CallbackStack;
```

```
/*124*/ struct _WIN32_THREAD     *Win32Thread;
/*128*/ PVOID                     TrapFrame;
/*12C*/ PKAPC_STATE               ApcStatePointer;
/*130*/ PVOID                     p130;
/*134*/ BOOLEAN                   EnableStackSwap;
/*135*/ BOOLEAN                   LargeStack;
/*136*/ BYTE                      ResourceIndex;
/*137*/ KPROCESSOR_MODE           PreviousMode;
/*138*/ DWORD                     KernelTime; // ticks
/*13C*/ DWORD                     UserTime;   // ticks
/*140*/ KAPC_STATE                SavedApcState;
/*157*/ BYTE                      bReserved02;
/*158*/ BOOLEAN                   Alertable;
/*159*/ BYTE                      ApcStateIndex;
/*15A*/ BOOLEAN                   ApcQueueable;
/*15B*/ BOOLEAN                   AutoAlignment;
/*15C*/ PVOID                     StackBase;
/*160*/ KAPC                      SuspendApc;
/*190*/ KSEMAPHORE                SuspendSemaphore;
/*1A4*/ LIST_ENTRY                ThreadListEntry;  // see KPROCESS
/*1AC*/ BYTE                      FreezeCount;
/*1AD*/ BYTE                      SuspendCount;
/*1AE*/ BYTE                      IdealProcessor;
/*1AF*/ BOOLEAN                   DisableBoost;
/*1B0*/ }
        KTHREAD,
     * PKTHREAD,
    **PPKTHREAD;
```

LISTING 7-12. *The* KTHREAD *Object Structure*

A KPROCESS links to its threads via its ThreadListHead member, which is the starting and ending point of a doubly linked list of KTHREAD objects. The list nodes of the threads are represented by their ThreadListEntry members. As usual with LIST_ENTRY nodes, the base address of the surrounding object is computed by subtracting the offset of the LIST_ENTRY member from its address, because the Flink and Blink members always point to the next LIST_ENTRY inside the list, not to the owner of the list node. This makes it possible to interlink objects in multiple lists without any interference.

In Listings 7-11 and 7-12, as well as in the following listings, you see occasional members with names consisting of a lower-case letter and a three-digit hexadecimal number. These are members whose identity and purpose is currently unknown to me. The leading character reflects the supposed member type (e.g., d for DWORD or p for PVOID), and the numeric trailer specifies the member's offset from the beginning of the structure.

The EPROCESS and ETHREAD executive objects surrounding the KPROCESS and KTHREAD dispatcher objects are shown in Listings 7-13 and 7-14. These structures contain several unidentified members that hopefully will be analyzed soon by others, maybe encouraged by the material in this book. However, the most important and most frequently referenced members are included, and at least it is known what information is missing.

```
typedef struct _EPROCESS
        {
/*000*/ KPROCESS              Pcb;
/*06C*/ NTSTATUS              ExitStatus;
/*070*/ KEVENT                LockEvent;
/*080*/ DWORD                 LockCount;
/*084*/ DWORD                 d084;
/*088*/ LARGE_INTEGER         CreateTime;
/*090*/ LARGE_INTEGER         ExitTime;
/*098*/ PVOID                 LockOwner;
/*09C*/ DWORD                 UniqueProcessId;
/*0A0*/ LIST_ENTRY            ActiveProcessLinks;
/*0A8*/ DWORD                 QuotaPeakPoolUsage [2]; // NP, P
/*0B0*/ DWORD                 QuotaPoolUsage     [2]; // NP, P
/*0B8*/ DWORD                 PagefileUsage;
/*0BC*/ DWORD                 CommitCharge;
/*0C0*/ DWORD                 PeakPagefileUsage;
/*0C4*/ DWORD                 PeakVirtualSize;
/*0C8*/ LARGE_INTEGER         VirtualSize;
/*0D0*/ MMSUPPORT             Vm;
/*100*/ DWORD                 d100;
/*104*/ DWORD                 d104;
/*108*/ DWORD                 d108;
/*10C*/ DWORD                 d10C;
/*110*/ DWORD                 d110;
/*114*/ DWORD                 d114;
/*118*/ DWORD                 d118;
/*11C*/ DWORD                 d11C;
/*120*/ PVOID                 DebugPort;
/*124*/ PVOID                 ExceptionPort;
/*128*/ PHANDLE_TABLE         ObjectTable;
/*12C*/ PVOID                 Token;
/*130*/ FAST_MUTEX            WorkingSetLock;
/*150*/ DWORD                 WorkingSetPage;
/*154*/ BOOLEAN               ProcessOutswapEnabled;
/*155*/ BOOLEAN               ProcessOutswapped;
/*156*/ BOOLEAN               AddressSpaceInitialized;
/*157*/ BOOLEAN               AddressSpaceDeleted;
/*158*/ FAST_MUTEX            AddressCreationLock;
/*178*/ KSPIN_LOCK            HyperSpaceLock;
/*17C*/ DWORD                 ForkInProgress;
```

```
/*180*/ WORD                   VmOperation;
/*182*/ BOOLEAN                ForkWasSuccessful;
/*183*/ BYTE                   MmAgressiveWsTrimMask;
/*184*/ DWORD                  VmOperationEvent;
/*188*/ HARDWARE_PTE           PageDirectoryPte;
/*18C*/ DWORD                  LastFaultCount;
/*190*/ DWORD                  ModifiedPageCount;
/*194*/ PVOID                  VadRoot;
/*198*/ PVOID                  VadHint;
/*19C*/ PVOID                  CloneRoot;
/*1A0*/ DWORD                  NumberOfPrivatePages;
/*1A4*/ DWORD                  NumberOfLockedPages;
/*1A8*/ WORD                   NextPageColor;
/*1AA*/ BOOLEAN                ExitProcessCalled;
/*1AB*/ BOOLEAN                CreateProcessReported;
/*1AC*/ HANDLE                 SectionHandle;
/*1B0*/ struct _PEB           *Peb;
/*1B4*/ PVOID                  SectionBaseAddress;
/*1B8*/ PQUOTA_BLOCK           QuotaBlock;
/*1BC*/ NTSTATUS               LastThreadExitStatus;
/*1C0*/ DWORD                  WorkingSetWatch;
/*1C4*/ HANDLE                 Win32WindowStation;
/*1C8*/ DWORD                  InheritedFromUniqueProcessId;
/*1CC*/ ACCESS_MASK            GrantedAccess;
/*1D0*/ DWORD                  DefaultHardErrorProcessing; // HEM_*
/*1D4*/ DWORD                  LdtInformation;
/*1D8*/ PVOID                  VadFreeHint;
/*1DC*/ DWORD                  VdmObjects;
/*1E0*/ PVOID                  DeviceMap; // 0x24 bytes
/*1E4*/ DWORD                  SessionId;
/*1E8*/ DWORD                  d1E8;
/*1EC*/ DWORD                  d1EC;
/*1F0*/ DWORD                  d1F0;
/*1F4*/ DWORD                  d1F4;
/*1F8*/ DWORD                  d1F8;
/*1FC*/ BYTE                   ImageFileName [16];
/*20C*/ DWORD                  VmTrimFaultValue;
/*210*/ BYTE                   SetTimerResolution;
/*211*/ BYTE                   PriorityClass;
/*212*/ union
            {
            struct
                {
/*212*/         BYTE                   SubSystemMinorVersion;
/*213*/         BYTE                   SubSystemMajorVersion;
                };
            struct
                {
/*212*/         WORD                   SubSystemVersion;
                };
```

(continued)

```
            };
/*214*/ struct _WIN32_PROCESS *Win32Process;
/*218*/ DWORD                  d218;
/*21C*/ DWORD                  d21C;
/*220*/ DWORD                  d220;
/*224*/ DWORD                  d224;
/*228*/ DWORD                  d228;
/*22C*/ DWORD                  d22C;
/*230*/ PVOID                  Wow64;
/*234*/ DWORD                  d234;
/*238*/ IO_COUNTERS            IoCounters;
/*268*/ DWORD                  d268;
/*26C*/ DWORD                  d26C;
/*270*/ DWORD                  d270;
/*274*/ DWORD                  d274;
/*278*/ DWORD                  d278;
/*27C*/ DWORD                  d27C;
/*280*/ DWORD                  d280;
/*284*/ DWORD                  d284;
/*288*/ }
        EPROCESS,
     * PEPROCESS,
    **PPEPROCESS;
```

LISTING 7-13. *The* EPROCESS *Object Structure*

```
typedef struct _ETHREAD
        {
/*000*/ KTHREAD       Tcb;
/*1B0*/ LARGE_INTEGER CreateTime;
/*1B8*/ union
            {
/*1B8*/     LARGE_INTEGER ExitTime;
/*1B8*/     LIST_ENTRY    LpcReplyChain;
            };
/*1C0*/ union
            {
/*1C0*/     NTSTATUS      ExitStatus;
/*1C0*/     DWORD         OfsChain;
            };
```

```
/*1C4*/ LIST_ENTRY    PostBlockList;
/*1CC*/ LIST_ENTRY    TerminationPortList;
/*1D4*/ PVOID         ActiveTimerListLock;
/*1D8*/ LIST_ENTRY    ActiveTimerListHead;
/*1E0*/ CLIENT_ID     Cid;
/*1E8*/ KSEMAPHORE    LpcReplySemaphore;
/*1FC*/ DWORD         LpcReplyMessage;
/*200*/ DWORD         LpcReplyMessageId;
/*204*/ DWORD         PerformanceCountLow;
/*208*/ DWORD         ImpersonationInfo;
/*20C*/ LIST_ENTRY    IrpList;
/*214*/ PVOID         TopLevelIrp;
/*218*/ PVOID         DeviceToVerify;
/*21C*/ DWORD         ReadClusterSize;
/*220*/ BOOLEAN       ForwardClusterOnly;
/*221*/ BOOLEAN       DisablePageFaultClustering;
/*222*/ BOOLEAN       DeadThread;
/*223*/ BOOLEAN       Reserved;
/*224*/ BOOL          HasTerminated;
/*228*/ ACCESS_MASK   GrantedAccess;
/*22C*/ PEPROCESS     ThreadsProcess;
/*230*/ PVOID         StartAddress;
/*234*/ union
            {
/*234*/     PVOID         Win32StartAddress;
/*234*/     DWORD         LpcReceivedMessageId;
            };
/*238*/ BOOLEAN       LpcExitThreadCalled;
/*239*/ BOOLEAN       HardErrorsAreDisabled;
/*23A*/ BOOLEAN       LpcReceivedMsgIdValid;
/*23B*/ BOOLEAN       ActiveImpersonationInfo;
/*23C*/ DWORD         PerformanceCountHigh;
/*240*/ DWORD         d240;
/*244*/ DWORD         d244;
/*248*/ }
        ETHREAD,
      * PETHREAD,
     **PPETHREAD;
```

LISTING 7-14. *The* ETHREAD *Object Structure*

It is apparent that both the EPROCESS and ETHREAD object structures contain additional members after the ones listed by the !processfields and !threadfields debugger commands. You may wonder how I dare to claim that. Well, there are two principal ways to find out details about undocumented object structure members. One is to observe how system functions operating on objects access their members; the other one is to examine how objects are created and initialized. The latter approach yields the size of an object. The basic object creation function inside ntoskrnl.exe is ObCreateObject(). It allocates the memory for the object header and body and initializes common object parameters. However, ObCreateObject() is absolutely ignorant about the type of object it creates, so the caller must specify the number of bytes required for the object body. Hence, the problem of finding out the size of an object boils down to finding an ObCreateObject() call for this object type. Process objects are created by the Native API function NtCreateProcess(), which lets PspCreateProcess() do the dirty work. Inside this function, an ObCreateObject() call can be found that requests an object body size of 0x288 bytes. That's why Listing 7-13 contains a couple of unidentified trailing members until a final offset of 0x288 is reached. The situation is similar for the ETHREAD structure. The NtCreateThread() API function calls PspCreateThread(), which in turn calls ObCreateObject(), requesting 0x248 bytes.

The list of currently running processes is formed by interlinking the ActiveProcessLinks member of the EPROCESS structure. The head of this list is stored in the internal global variable PsActiveProcessHead, and the associated FAST_MUTEX synchronization object is named PspActiveProcessMutex. Unfortunately, the PsActiveProcessHead variable is not exported by ntoskrnl.exe, but PsInitialSystemProcess is, pointing to the EPROCESS structure of the System process with the process ID 8. Following the Blink of its ActiveProcessLinks list entry leads us directly to the PsActiveProcessHead. Basically, the linkage of processes and threads is structured as shown in Figure 7-3. Figure 7-3 is overly simplified because the illustrated process list contains only two items. In a real-world scenario, the list will be much longer. (While I am writing this paragraph, my task manager reports 36 processes!) To keep the picture as simple as possible, only the thread list of one process is shown, assuming that this process has two active threads.

Listings 7-12 and 7-13 suggest that there must be a third process and thread object layer above the kernel and executive layers, indicated by pointers to WIN32_PROCESS and WIN32_THREAD structures inside EPROCESS and KTHREAD. These undocumented structures constitute the process and thread representations of the Win32 subsystem. Although the purposes of some of their members are quite obvious, they still contain too many unidentified holes to be included here. This is another area of future research.

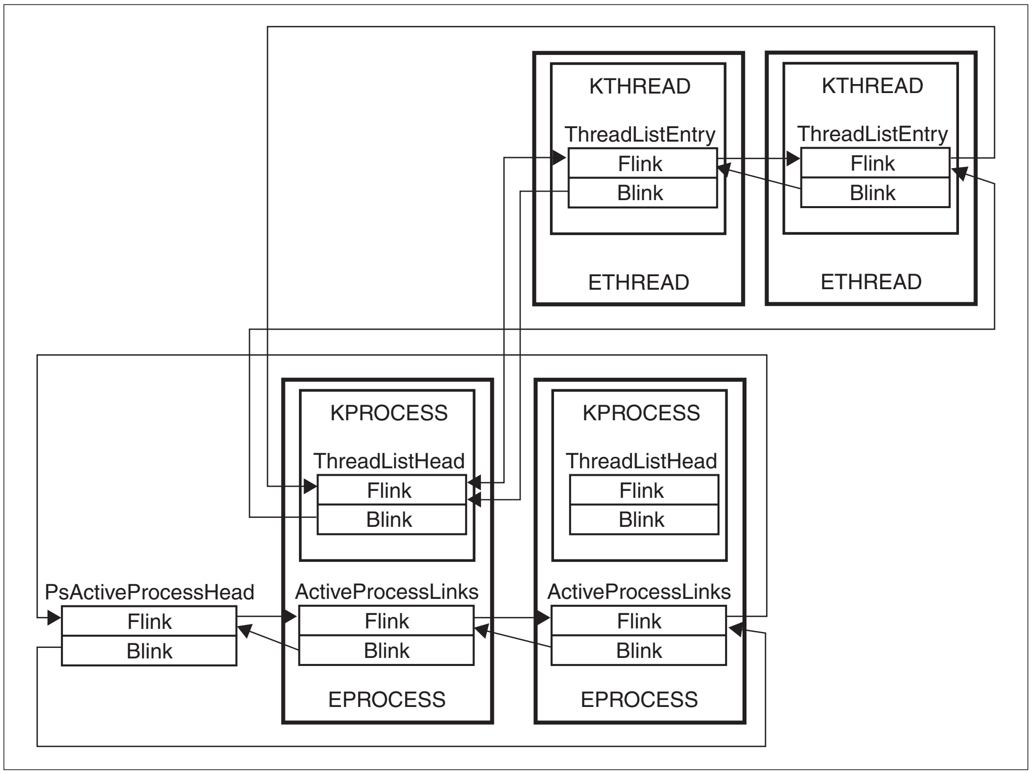

FIGURE 7-3. *Process and Thread Object Lists*

THREAD AND PROCESS CONTEXTS

While the system executes code, the execution always takes place in the context of a thread that is part of some process. In several situations, the system has to look up thread- or process-specific information from the current context. Therefore, the system always keeps a pointer to the current thread in the Kernel's Processor Control Block (KPRCB). This structure, defined in ntddk.h, is shown in Listing 7-15.

```
typedef struct _KPRCB // base address 0xFFDFF120
        {
/*000*/ WORD                    MinorVersion;
/*002*/ WORD                    MajorVersion;
/*004*/ struct _KTHREAD        *CurrentThread;
/*008*/ struct _KTHREAD        *NextThread;
/*00C*/ struct _KTHREAD        *IdleThread;
/*010*/ CHAR                    Number;
/*011*/ CHAR                    Reserved;
```

(continued)

```
/*012*/ WORD                  BuildType;/*014*/ KAFFINITY              SetMember;
/*018*/ struct _RESTART_BLOCK *RestartBlock;
/*01C*/ }
        KPRCB,
     * PKPRCB,
    **PPKPRCB;
```

LISTING 7-15. *The Kernel's Processor Control Block* (KPRCB)

The `KPRCB` structure is found at linear address `0xFFDFF120`, and a pointer to it is stored in the `Prcb` member of the Kernel's Processor Control Region (`KPCR`), also defined in `ntddk.h` (Listing 7-16) and located at address `0xFFDFF000`. As explained in Chapter 4, this essential data area is readily accessible in kernel-mode via the `FS` segment; that is, reading from address `FS:0` is equivalent to reading from linear address `DS:0xFFDFF000`. At address `0xFFDFF13C`, immediately following the `KPRCB`, the system keeps low-level CPU information in a `CONTEXT` structure (Listing 7-17).

```
typedef struct _KPCR // base address 0xFFDFF000
        {
/*000*/ NT_TIB              NtTib;
/*01C*/ struct _KPCR       *SelfPcr;
/*020*/ PKPRCB              Prcb;
/*024*/ KIRQL               Irql;
/*028*/ DWORD               IRR;
/*02C*/ DWORD               IrrActive;
/*030*/ DWORD               IDR;
/*034*/ DWORD               Reserved2;
/*038*/ struct _KIDTENTRY *IDT;
/*03C*/ struct _KGDTENTRY *GDT;
/*040*/ struct _KTSS       *TSS;
/*044*/ WORD                MajorVersion;
/*046*/ WORD                MinorVersion;
/*048*/ KAFFINITY           SetMember;
/*04C*/ DWORD               StallScaleFactor;
/*050*/ BYTE                DebugActive;
/*051*/ BYTE                Number;
/*054*/ }
        KPCR,
     * PKPCR,
    **PPKPCR;
```

LISTING 7-16. *The Kernel's Processor Control Region* (KPCR)

```
#define SIZE_OF_80387_REGISTERS 80

typedef struct _FLOATING_SAVE_AREA // base address 0xFFDFF158
        {
/*000*/ DWORD ControlWord;
/*004*/ DWORD StatusWord;
/*008*/ DWORD TagWord;
/*00C*/ DWORD ErrorOffset;
/*010*/ DWORD ErrorSelector;
/*014*/ DWORD DataOffset;
/*018*/ DWORD DataSelector;
/*01C*/ BYTE  RegisterArea [SIZE_OF_80387_REGISTERS];
/*06C*/ DWORD Cr0NpxState;
/*070*/ }
        FLOATING_SAVE_AREA,
     * PFLOATING_SAVE_AREA,
    **PPFLOATING_SAVE_AREA;

// -----------------------------------------------------------------

#define MAXIMUM_SUPPORTED_EXTENSION 512

typedef struct _CONTEXT // base address 0xFFDFF13C
        {
/*000*/ DWORD       ContextFlags;
/*004*/ DWORD       Dr0;
/*008*/ DWORD       Dr1;
/*00C*/ DWORD       Dr2;
/*010*/ DWORD       Dr3;
/*014*/ DWORD       Dr6;
/*018*/ DWORD       Dr7;
/*01C*/ FLOATING_SAVE_AREA FloatSave;
/*08C*/ DWORD       SegGs;
/*090*/ DWORD       SegFs;
/*094*/ DWORD       SegEs;
/*098*/ DWORD       SegDs;
/*09C*/ DWORD       Edi;
/*0A0*/ DWORD       Esi;
/*0A4*/ DWORD       Ebx;
/*0A8*/ DWORD       Edx;
/*0AC*/ DWORD       Ecx;
/*0B0*/ DWORD       Eax;
/*0B4*/ DWORD       Ebp;
/*0B8*/ DWORD       Eip;
/*0BC*/ DWORD       SegCs;
/*0C0*/ DWORD       EFlags;
/*0C4*/ DWORD       Esp;
/*0C8*/ DWORD       SegSs;
/*0CC*/ BYTE        ExtendedRegisters [MAXIMUM_SUPPORTED_EXTENSION];
/*2CC*/ }
        CONTEXT,
     * PCONTEXT,
    **PPCONTEXT;
```

LISTING 7-17. *The CPU's* CONTEXT *and* FLOATING_SAVE_AREA

According to Listing 7-15, the KPRCB contains three KTHREAD pointers at the offsets 0x004, 0x008, and 0x00C:

1. CurrentThread points to the KTHREAD object of the thread that is currently executing. This member is accessed very frequently by the kernel code.
2. NextThread points to the KTHREAD object of the thread scheduled to run after the next context switch.
3. IdleThread points to the KTHREAD object of an idle thread that performs background tasks while no other threads are ready to run. The system provides a dedicated idle thread for each installed CPU. On a single-processor machine, the idle thread object is named P0BootThread and is the only thread in the thread list of the PsIdleProcess object.

Because the first member of an ETHREAD is a KTHREAD, a KTHREAD pointer always points to an ETHREAD as well, and vice versa. This means that KTHREAD and ETHREAD can be typecast interchangeably. The same is true for KPROCESS and EPROCESS pointers.

Because the Windows 2000 kernel maps the linear address 0xFFDFF000 to address 0x00000000 of the CPU's FS segment in kernel-mode, the system always finds the current KPCR, KPRCB, and CONTEXT data at the addresses FS:0x0, FS:0x120, and FS:13C. When you are disassembling kernel code in a debugger, you will frequently see the system retrieve a pointer from FS:0x124, which is obviously the current thread object. Example 7-1 lists the output of the Kernel Debugger if the command u PsGetCurrentProcessId is issued, instructing the debugger to unassemble 10 lines of code, starting at the address of the symbol PsGetCurrentProcessId. The implementation of the PsGetCurrentProcessId() function simply retrieves the KTHREAD/ETHREAD of the current thread and returns the value of the member at offset 0x1E0, which happens to be the UniqueProcess ID of the CLIENT_ID Cid member of the ETHREAD, according to Listing 7-14. PsGetCurrentThreadId() is almost identical, except that it retrieves the UniqueThread ID at offset 0x1E4. By the way, the CLIENT_ID structure has been introduced in Chapter 2, Listing 2-8.

```
kd> u PsGetCurrentProcessId
u PsGetCurrentProcessId
ntoskrnl!PsGetCurrentProcessId:
8045252a 64a124010000     mov     eax,fs:[00000124]
80452530 8b80e0010000     mov     eax,[eax+0x1e0]
```

```
80452536 c3             ret
80452537 cc             int     3
ntoskrnl!PsGetCurrentThreadId:
80452538 64a124010000    mov     eax,fs:[00000124]
8045253e 8b80e4010000    mov     eax,[eax+0x1e4]
80452544 c3             ret
80452545 cc             int     3
```

EXAMPLE 7-1. *Retrieving Process and Thread IDs*

Sometimes, the system needs a pointer to the process object that owns the current thread. This address can be looked up quite easily by reading the `Process` member of the `ApcState` substructure inside the current `KTHREAD`.

THREAD AND PROCESS ENVIRONMENT BLOCKS

You may wonder about the purpose of the `Teb` and `Peb` members inside the `KTHREAD` and `EPROCESS` structures. The `Teb`, points to a `Thread Environment Block (TEB)`, outlined in Listing 7-18. The first part of the `TEB` the `Thread Information Block (NT_TIB)`, is defined in the Platform Software Development Kit (SDK) and DDK header files `winnt.h` and `ntddk.h`, respectively. The remaining members are undocumented. Windows 2000 maintains a `TEB` structure for each thread object in the system. In the address space of the current process, the `TEB`s of its threads are mapped to the linear addresses `0x7FFDE000`, `0x7FFDD000`, `0x7FFDC000`, and so on, always stepping down one 4-KB page per thread. As noted in Chapter 4, the `TEB` of the current thread is also accessible via the `FS` segment in user-mode. Many `ntdll.dll` functions access the current `TEB` by reading the value at address `FS:0x18`, which is the `Self` member of the embedded `NT_TIB`. This member always provides the linear address of the surrounding `TEB` within the 4-GB address space of the current process.

```
// typedef struct _NT_TIB // see winnt.h / ntddk.h
//         {
// /*000*/ struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList;
// /*004*/ PVOID                                  StackBase;
// /*008*/ PVOID                                  StackLimit;
// /*00C*/ PVOID                                  SubSystemTib;
// /*010*/ union
```

(continued)

```
//              {
// /*010*/      PVOID FiberData;
// /*010*/      ULONG Version;
//              };
// /*014*/ PVOID           ArbitraryUserPointer;
// /*018*/ struct _NT_TIB *Self;
// /*01C*/ }
//          NT_TIB,
//       * PNT_TIB,
//      **PPNT_TIB;

// ----------------------------------------------------------------

typedef struct _TEB // base addresses 0x7FFDE000, 0x7FFDD000, ...
        {
/*000*/ NT_TIB    Tib;
/*01C*/ PVOID     EnvironmentPointer;
/*020*/ CLIENT_ID Cid;
/*028*/ HANDLE    RpcHandle;
/*02C*/ PPVOID    ThreadLocalStorage;
/*030*/ PPEB      Peb;
/*034*/ DWORD     LastErrorValue;
/*038*/ }
        TEB,
     * PTEB,
    **PPTEB;
```

LISTING 7-18. *The Thread Environment Block* (TEB)

Just as each thread has its own `TEB`, each process has an associated `PEB` or `Process Environment Block`. The `PEB` is much more complex than the `TEB`, as Listing 7-19 demonstrates. It contains various pointers to subordinate structures that refer to more subordinate structures, and most of them are undocumented. Listing 7-19 includes raw sketches of some of them, using tentative names and leaving much to be desired. The `PEB` is located at linear address `0x7FFDF000`, that is, in the first 4-KB page following the `TEB` stack of the process. The system can easily access the `PEB` by simply referencing the `Peb` member of the current thread's `TEB`.

```
typedef struct _MODULE_HEADER
        {
/*000*/ DWORD      d000;
/*004*/ DWORD      d004;
/*008*/ LIST_ENTRY List1;
/*010*/ LIST_ENTRY List2;
```

```
/*018*/ LIST_ENTRY List3;
/*020*/ }
        MODULE_HEADER,
     * PMODULE_HEADER,
    **PPMODULE_HEADER;

// -----------------------------------------------------------------

typedef struct _PROCESS_MODULE_INFO
        {
/*000*/ DWORD         Size; // 0x24
/*004*/ MODULE_HEADER ModuleHeader;
/*024*/ }
        PROCESS_MODULE_INFO,
     * PPROCESS_MODULE_INFO,
    **PPPROCESS_MODULE_INFO;

// -----------------------------------------------------------------
// see RtlCreateProcessParameters()

typedef struct _PROCESS_PARAMETERS
        {
/*000*/ DWORD          Allocated;
/*004*/ DWORD          Size;
/*008*/ DWORD          Flags; // bit 0: all pointers normalized
/*00C*/ DWORD          Reserved1;
/*010*/ LONG           Console;
/*014*/ DWORD          ProcessGroup;
/*018*/ HANDLE         StdInput;
/*01C*/ HANDLE         StdOutput;
/*020*/ HANDLE         StdError;
/*024*/ UNICODE_STRING WorkingDirectoryName;
/*02C*/ HANDLE         WorkingDirectoryHandle;
/*030*/ UNICODE_STRING SearchPath;
/*038*/ UNICODE_STRING ImagePath;
/*040*/ UNICODE_STRING CommandLine;
/*048*/ PWORD          Environment;
/*04C*/ DWORD          X;
/*050*/ DWORD          Y;
/*054*/ DWORD          XSize;
/*058*/ DWORD          YSize;
/*05C*/ DWORD          XCountChars;
/*060*/ DWORD          YCountChars;
/*064*/ DWORD          FillAttribute;
/*068*/ DWORD          Flags2;
/*06C*/ WORD           ShowWindow;
/*06E*/ WORD           Reserved2;
/*070*/ UNICODE_STRING Title;
/*078*/ UNICODE_STRING Desktop;
/*080*/ UNICODE_STRING Reserved3;
```

(continued)

```
/*088*/ UNICODE_STRING Reserved4;
/*090*/ }
        PROCESS_PARAMETERS,
     * PPROCESS_PARAMETERS,
    **PPPROCESS_PARAMETERS;

// -----------------------------------------------------------------

typedef struct _SYSTEM_STRINGS
        {
/*000*/ UNICODE_STRING SystemRoot;       // d:\WINNT
/*008*/ UNICODE_STRING System32Root;     // d:\WINNT\System32
/*010*/ UNICODE_STRING BaseNamedObjects; // \BaseNamedObjects
/*018*/ }
        SYSTEM_STRINGS,
     * PSYSTEM_STRINGS,
    **PPSYSTEM_STRINGS;

// -----------------------------------------------------------------

typedef struct _TEXT_INFO
        {
/*000*/ PVOID           Reserved;
/*004*/ PSYSTEM_STRINGS SystemStrings;
/*008*/ }
        TEXT_INFO,
     * PTEXT_INFO,
    **PPTEXT_INFO;

// -----------------------------------------------------------------

typedef struct _PEB // base address 0x7FFDF000
        {
/*000*/ BOOLEAN              InheritedAddressSpace;
/*001*/ BOOLEAN              ReadImageFileExecOptions;
/*002*/ BOOLEAN              BeingDebugged;
/*003*/ BYTE                 b003;
/*004*/ DWORD                d004;
/*008*/ PVOID                SectionBaseAddress;
/*00C*/ PPROCESS_MODULE_INFO ProcessModuleInfo;
/*010*/ PPROCESS_PARAMETERS  ProcessParameters;
/*014*/ DWORD                SubSystemData;
/*018*/ HANDLE               ProcessHeap;
/*01C*/ PCRITICAL_SECTION    FastPebLock;
/*020*/ PVOID                AcquireFastPebLock; // function
/*024*/ PVOID                ReleaseFastPebLock; // function
/*028*/ DWORD                d028;
/*02C*/ PPVOID               User32Dispatch;     // function
/*030*/ DWORD                d030;
```

```
/*034*/ DWORD               d034;
/*038*/ DWORD               d038;
/*03C*/ DWORD               TlsBitMapSize;      // number of bits
/*040*/ PRTL_BITMAP         TlsBitMap;          // ntdll!TlsBitMap
/*044*/ DWORD               TlsBitMapData [2];  // 64 bits
/*04C*/ PVOID               p04C;
/*050*/ PVOID               p050;
/*054*/ PTEXT_INFO          TextInfo;
/*058*/ PVOID               InitAnsiCodePageData;
/*05C*/ PVOID               InitOemCodePageData;
/*060*/ PVOID               InitUnicodeCaseTableData;
/*064*/ DWORD               KeNumberProcessors;
/*068*/ DWORD               NtGlobalFlag;
/*06C*/ DWORD               d6C;
/*070*/ LARGE_INTEGER       MmCriticalSectionTimeout;
/*078*/ DWORD               MmHeapSegmentReserve;
/*07C*/ DWORD               MmHeapSegmentCommit;
/*080*/ DWORD               MmHeapDeCommitTotalFreeThreshold;
/*084*/ DWORD               MmHeapDeCommitFreeBlockThreshold;
/*088*/ DWORD               NumberOfHeaps;
/*08C*/ DWORD               AvailableHeaps; // 16, *2 if exhausted
/*090*/ PHANDLE             ProcessHeapsListBuffer;
/*094*/ DWORD               d094;
/*098*/ DWORD               d098;
/*09C*/ DWORD               d09C;
/*0A0*/ PCRITICAL_SECTION   LoaderLock;
/*0A4*/ DWORD               NtMajorVersion;
/*0A8*/ DWORD               NtMinorVersion;
/*0AC*/ WORD                NtBuildNumber;
/*0AE*/ WORD                CmNtCSDVersion;
/*0B0*/ DWORD               PlatformId;
/*0B4*/ DWORD               Subsystem;
/*0B8*/ DWORD               MajorSubsystemVersion;
/*0BC*/ DWORD               MinorSubsystemVersion;
/*0C0*/ KAFFINITY           AffinityMask;
/*0C4*/ DWORD               ad0C4 [35];
/*150*/ PVOID               p150;
/*154*/ DWORD               ad154 [32];
/*1D4*/ HANDLE              Win32WindowStation;
/*1D8*/ DWORD               d1D8;
/*1DC*/ DWORD               d1DC;
/*1E0*/ PWORD               CSDVersion;
/*1E4*/ DWORD               d1E4;
/*1E8*/ }
        PEB,
     * PPEB,
    **PPPEB;
```

LISTING 7-19. *The Process Environment Block* (PEB)

ACCESSING LIVE SYSTEM OBJECTS

The preceding sections have provided a lot of theoretical information. As a practical example to illustrate object management in the most useful form, I thought of writing a kernel object browser. This would show how objects are arranged hierarchically and how some of their properties can be retrieved. Unfortunately, `ntoskrnl.exe` fails to export several key structures and functions required in an object browser application. This means that not even a kernel-mode driver has access to them—they are reserved for internal system use. On the other hand, Chapter 6 introduced a mechanism that allows access to nonexported data and code by evaluating the Windows 2000 symbol files, so the object browser seemed to be an ideal test case to check out the practical suitability of this approach. The symbolic call interface from Chapter 6 passed this test, so I have included the sample application `w2k_obj.exe` with full source code on the companion CD in the directory tree `\src\w2k_obj`. However, the most interesting parts of the code are not buried inside `w2k_obj.c`. The hard work is really done by the `w2k_call.dll` library introduced in Chapter 6. Hence, many of the subsequent code snippets are pulled from `w2k_call.c`.

ENUMERATING OBJECT DIRECTORY ENTRIES

You probably know the small `objdir.exe` utility in the Windows 2000 DDK, in the `\ntddk\bin` directory. `objdir.exe` retrieves object directory information via the undocumented Native API function `NtQueryDirectoryObject()` exported by `ntdll.dll`. Contrary to this, my object browser `w2k_obj.exe` bangs directly at the object directory and its leaf objects. This sounds rather scary, but actually it isn't. The best proof is that `w2k_obj.exe` works on both Windows 2000 and Windows NT 4.0 without a single line of version-dependent code. Admittedly, there are a couple of subtle differences in the object structures of both operating system versions, but the basic model has remained the same. Providing a sample application that works directly on the raw object structures rather than using higher-level API functions is an illustrative means to verify whether the structures shown in the preceding sections are accurate.

The most important thing to do before accessing global system data structures is to lock them. Otherwise it might happen that the system alters the data in the context of a concurrent thread, so the application unexpectedly reads invalid data or reaches into the void. Windows 2000 provides a large set of locks for the numerous internal data items it maintains. The problem with these locks is that they are usually not exported. Although a kernel-mode driver can do all sorts of things forbidden in user-mode, it can't safely access nonexported data structures. However, the extended kernel call interface discussed in Chapter 6 and implemented by the `w2k_call.dll` sample library `can` make the impossible possible by looking up the addresses of internal symbols from the operating system's symbol files. This DLL exports the following three object manager data thunks that allow access to the kernel's object directory:

1. `__ObpRootDirectoryMutex()` returns the address of the `ERESOURCE` lock that synchronizes access to the object directory as a whole.
2. `__ObpRootDirectoryObject()` returns a pointer to the `OBJECT_DIRECTORY` structure representing the root node of the object directory.
3. `__ObpTypeDirectoryObject()` returns a pointer to the `OBJECT_DIRECTORY` structure representing the `\ObjectTypes` subdirectory node of the object directory.

An application must be extremely cautious when it works with pointers to kernel objects, especially after acquiring a global lock. If the lock isn't properly released, the system might be left in a handcuffed state, unable to perform even the simplest tasks.

Although the root directory lock is named `ObpRootDirectoryMutex`, it isn't really a mutex in the strict sense of the word. It is an `ERESOURCE` rather than a `KMUTEX`, and as such must be acquired with the help of the `ExAcquireResourceExclusiveLite()` or `ExAcquireResourceSharedLite()` API functions. The "Lite" suffix is important—never use the siblings `ExAcquireResourceExclusive()` or `ExAcquireResourcShared()` on Windows 2000 or NT4 `ERESOURCE` locks. This structure has been revised quite a bit since Windows NT 3.x, and the latter pair of functions works only with the old-style `ERESOURCE` type, included in `w2k_def.h` as `ERESOURCE_OLD` (see also Appendix C). The counterpart of the `ExAcquireResource*Lite()` functions is named `ExReleaseResourceLite()` and should be carefully distinguished from its old-style sibling `ExReleaseResource()`.

The basic approach of my object browser is to lock the object directory, take a snapshot of all nodes found in its hierarchic structure, and display the snapshot data after releasing the directory lock. This procedure guarantees the least interference with the system, and the application can take as much time as it needs to display the data without overusing the system. Taking a faithful snapshot of the directory requires very intimate knowledge of the system's object structures, so this application is a great test case for the reliability of the object information I have supplied above. This job can be subdivided into the following two basic tasks:

1. Copying the `structure` of the object directory tree. This involves copying and interlinking several `OBJECT_DIRECTORY` structures, each one representing an individual nonleaf node.
2. Copying the `contents` of the object directory tree. This means copying the `OBJECT_HEADER` and its related structures of each leaf node in the tree.

The `w2kDirectoryOpen()` function shown in Listing 7-20 performs the first task. It locks the directory and enumerates all children of the supplied `OBJECT_DIRECTORY`. To capture the entire object tree, this function must be called recursively for each directory entry that is itself an `OBJECT_DIRECTORY`. Please recall that each object directory node consists of a hash table that can accommodate a maximum of 37 entries. Each hash table slot can in turn refer to an arbitrary number of entries by putting them into a linked list. Therefore, enumeration of directory entries requires two nested loops: The outer one scans all 37 hash table slots for non-`NULL` entries, and the inner one walks down the linked lists. This is about all the `w2kDirectoryOpen()` function does. The resulting data is structurally equivalent to the original model, except that all pointers refer to memory blocks reachable in user-mode. The basic copying including automatic memory allocation is performed by the powerful `w2kSpyClone()` function, also exported by `w2k_call.dll` (see Listing 6-30). The `w2kDirectoryClose()` function in Listing 7-20 undoes the work done by `w2kDirectoryOpen()`, simply deallocating all cloned memory blocks.

```
POBJECT_DIRECTORY WINAPI
w2kDirectoryOpen (POBJECT_DIRECTORY pDir)
    {
    DWORD                    i;
    PERESOURCE               pLock;
    PPOBJECT_DIRECTORY_ENTRY ppEntry;
    POBJECT_DIRECTORY        pDir1 = NULL;

    if (((pLock = __ObpRootDirectoryMutex ()) != NULL) &&
        _ExAcquireResourceExclusiveLite (pLock, TRUE))
        {
        if ((pDir1 = w2kSpyClone (pDir, OBJECT_DIRECTORY_)) != NULL)
            {
            for (i = 0; i < OBJECT_HASH_TABLE_SIZE; i++)
                {
                ppEntry = pDir1->HashTable + i;

                while (*ppEntry != NULL)
                    {
                    if ((*ppEntry =
                            w2kSpyClone (*ppEntry,
                                         OBJECT_DIRECTORY_ENTRY_))
                        != NULL)
                        {
                        (*ppEntry)->Object =
                            w2kObjectOpen ((*ppEntry)->Object);

                        ppEntry = &(*ppEntry)->NextEntry;
                        }
                    }
                }
            }
```

```
        _ExReleaseResourceLite (pLock);
        }
    return pDir1;
    }

// -----------------------------------------------------------------

POBJECT_DIRECTORY WINAPI
w2kDirectoryClose (POBJECT_DIRECTORY pDir)
    {
    POBJECT_DIRECTORY_ENTRY pEntry, pEntry1;
    DWORD                   i;

    if (pDir != NULL)
        {
        for (i = 0; i < OBJECT_HASH_TABLE_SIZE; i++)
            {
            for (pEntry  = pDir->HashTable [i];
                 pEntry != NULL;
                 pEntry  = pEntry1)
                {
                pEntry1 = pEntry->NextEntry;

                w2kObjectClose   (pEntry->Object);
                w2kMemoryDestroy (pEntry);
                }
            }
        w2kMemoryDestroy (pDir);
        }
    return NULL;
    }
```

LISTING 7-20. *The* `w2kDirectoryOpen()` *and* `w2kDirectoryClose()` *API Functions*

A closer look at Listing 7-20 reveals that `w2kDirectoryOpen()` and `w2kDirectoryClose()` call the functions `w2kObjectOpen()` and `w2kObjectClose()`, respectively. `w2kObjectOpen()` takes care of part two of the directory copying procedure: It clones leaf objects. It doesn't produce complete object copies, because this would require identifying each object type and copying the appropriate number of bytes from the object body. `w2kObjectOpen()` copies the entire header portion of an object, including most of its subordinate structures, and builds a fake object body that contains pointers to the real object body and to various parts of the object header copy. Listing 7-21 shows the data structures built and initialized by `w2kObjectOpen()`. `W2K_OBJECT_FRAME` is a monolithic data block that comprises the object header copy and the fake object body. The latter is represented by the `W2K_OBJECT` structure, which is just a collection of pointers to members of

W2K_OBJECT_FRAME. w2kObjectOpen() allocates memory for the W2K_OBJECT_FRAME structure, initializes it with data from the original object, and returns a pointer to the object frame's Object member. If you recall the foregoing description of object bodies and headers, it becomes apparent that the W2K_OBJECT_FRAME mimics the structure of a real object. That is, it has all header fields the original object has, and an application can access them in the same way that the system accesses its objects in kernel-mode memory, using the offsets and flags in the OBJECT_HEADER.

```
typedef struct _W2K_OBJECT
    {
    POBJECT               pObject;
    POBJECT_HEADER        pHeader;
    POBJECT_CREATOR_INFO  pCreatorInfo;
    POBJECT_NAME          pName;
    POBJECT_HANDLE_DB     pHandleDB;
    POBJECT_QUOTA_CHARGES pQuotaCharges;
    POBJECT_TYPE          pType;
    PQUOTA_BLOCK          pQuotaBlock;
    POBJECT_CREATE_INFO   pCreateInfo;
    PWORD                 pwName;
    PWORD                 pwType;
    }
    W2K_OBJECT, *PW2K_OBJECT, **PPW2K_OBJECT;

#define W2K_OBJECT_ sizeof (W2K_OBJECT)

// -----------------------------------------------------------------

typedef struct _W2K_OBJECT_FRAME
    {
    OBJECT_QUOTA_CHARGES QuotaCharges;
    OBJECT_HANDLE_DB     HandleDB;
    OBJECT_NAME          Name;
    OBJECT_CREATOR_INFO  CreatorInfo;
    OBJECT_HEADER        Header;
    W2K_OBJECT           Object;
    OBJECT_TYPE          Type;
    QUOTA_BLOCK          QuotaBlock;
    OBJECT_CREATE_INFO   CreateInfo;
    WORD                 Buffer [];
    }
    W2K_OBJECT_FRAME, *PW2K_OBJECT_FRAME, **PPW2K_OBJECT_FRAME;

#define W2K_OBJECT_FRAME_ sizeof (W2K_OBJECT_FRAME)
#define W2K_OBJECT_FRAME__(_n) (W2K_OBJECT_FRAME_ + ((_n) * WORD_))
```

LISTING 7-21. *Object Clone Structures*

I don't want to go into the details of `w2kObjectOpen()` and all of its subordinate functions. For illustrative purposes, the three-part set of functions shown in Listing 7-22 should suffice. `w2kObjectHeader()` creates a copy of an object's `OBJECT_HEADER`, and `w2kObjectCreatorInfo()` and `w2kObjectName()` copy the `OBJECT_CREATOR_INFO` and `OBJECT_NAME` header parts, if present. Again, `w2kSpyClone()` is the main workhorse. For more examples of this kind, please refer to the `w2k_call.c` source file on the accompanying CD.

```
#define BACK(_p,_d) ((PVOID) (((PBYTE) (_p)) - (_d)))

// -----------------------------------------------------------------

POBJECT_HEADER WINAPI
w2kObjectHeader (POBJECT pObject)
    {
    DWORD          dOffset = OBJECT_HEADER_;
    POBJECT_HEADER pHeader = NULL;

    if (pObject != NULL)
        {
        pHeader = w2kSpyClone (BACK (pObject, dOffset),
                               dOffset);
        }
    return pHeader;
    }

// -----------------------------------------------------------------

POBJECT_CREATOR_INFO WINAPI
w2kObjectCreatorInfo (POBJECT_HEADER pHeader,
                      POBJECT        pObject)
    {
    DWORD                dOffset;
    POBJECT_CREATOR_INFO pCreatorInfo = NULL;

    if ((pHeader != NULL) && (pObject != NULL) &&
        (pHeader->ObjectFlags & OB_FLAG_CREATOR_INFO))
        {
        dOffset = OBJECT_CREATOR_INFO_ + OBJECT_HEADER_;

        pCreatorInfo = w2kSpyClone (BACK (pObject, dOffset),
                                    OBJECT_CREATOR_INFO_);
        }
    return pCreatorInfo;
    }
```

(continued)

```
// ----------------------------------------------------------------

POBJECT_NAME WINAPI
w2kObjectName (POBJECT_HEADER pHeader,
               POBJECT        pObject)
    {
    DWORD        dOffset;
    POBJECT_NAME pName = NULL;

    if ((pHeader != NULL) && (pObject != NULL) &&
        (dOffset = pHeader->NameOffset))
        {
        dOffset += OBJECT_HEADER_;

        pName = w2kSpyClone (BACK (pObject, dOffset),
                             OBJECT_NAME_);
        }
    return pName;
    }
```

LISTING 7-22. *Object Cloning Helper Functions*

The bottom line of the story is that `w2kDirectoryOpen()` takes a pointer to a live `OBJECT_DIRECTORY` node and returns a copy that contains `W2K_OBJECT` pointers where the original directory stores its object body pointers. The object browser application calls this API function repeatedly, once for each directory layer it displays. Listing 7-23 is a heavily edited version of the browser code, stripped down to its bare essentials. The original code found in `w2k_obj.c` contains many distracting extras that would have obscured the basic functional layout. The top-level function is named `DisplayObjects()`. It requests the object root pointer from `w2k_call.dll` via `__ObpRootDirectoryObject()` and forwards it to `DisplayOject()`, which displays the type and name of the object and calls itself recursively if the object is an `OBJECT_DIRECTORY`. For each nesting level, `DisplayObject()` adds a line indentation of three spaces. I have added the functions in Listing 7-23 to `w2k_obj.c` on the companion CD under the section header "POOR MAN'S OBJECT BROWSER." However, this code is not called anywhere, although it does work.

```
VOID WINAPI _DisplayObject (PW2K_OBJECT pObject,
                            DWORD       dLevel)
    {
    POBJECT_DIRECTORY       pDir;
    POBJECT_DIRECTORY_ENTRY pEntry;
    DWORD                   i;
```

```
    for (i = 0; i < dLevel; i++) printf (L"   ");

    _printf (L"%+.-16s%s\r\n", pObject->pwType, pObject->pwName);

    if ((!lstrcmp (pObject->pwType, L"Directory")) &&
        ((pDir = w2kDirectoryOpen (pObject->pObject)) != NULL))
        {
        for (i = 0; i < OBJECT_HASH_TABLE_SIZE; i++)
            {
            for (pEntry  = pDir->HashTable [i];
                 pEntry != NULL;
                 pEntry  = pEntry->NextEntry)
                {
                _DisplayObject (pEntry->Object, dLevel+1);
                }
            }
        w2kDirectoryClose (pDir);
        }
    return;
    }

// -----------------------------------------------------------------

VOID WINAPI _DisplayObjects (VOID)
    {
    PW2K_OBJECT pObject;

    if ((pObject = w2kObjectOpen (__ObpRootDirectoryObject ()))
        != NULL)
        {
        _DisplayObject (pObject, 0);
        w2kObjectClose (pObject);
        }
    return;
    }
```

LISTING 7-23. *A Very Simple Object Browser*

In Example 7-2, I have compiled some characteristic parts of an object directory listing generated by the code in Listing 7-23. For example, the `\BaseNamedObjects` subdirectory comprises named objects that are typically shared between processes and can be opened by name. The `\ObjectTypes` subdirectory contains all 27 `OBJECT_TYPE` type objects (cf. Listing 7-9) supported by the system, as listed in Table 7-4.

```
Directory.......\
   Directory.......ArcName
      SymbolicLink....multi(0)disk(0)rdisk(0)
      SymbolicLink....multi(0)disk(0)rdisk(1)
      SymbolicLink....multi(0)disk(0)rdisk(1)partition(1)
      SymbolicLink....multi(0)disk(0)rdisk(0)partition(1)
      SymbolicLink....multi(0)disk(0)fdisk(0)
      SymbolicLink....multi(0)disk(0)rdisk(0)partition(2)
   Device..........Ntfs
   Port............SeLsaCommandPort
   Key.............REGISTRY
   Port............XactSrvLpcPort
   Port............DbgUiApiPort
   Directory.......NLS
      Section.........NlsSectionCP874
      Section.........NlsSectionCP950
      Section.........NlsSectionCP20290
      Section.........NlsSectionCP1255c_1255.nls
...
   Directory.......BaseNamedObjects
      Section.........DfSharedHeapE445BB
      Section.........DFMap0-14765686
      Mutant..........ZonesCacheCounterMutex
      Section.........DFMap0-14364447
      Event...........WINMGMT_COREDLL_UNLOADED
      Mutant..........MCICDA_DeviceCritSec_19
      Event...........AgentToWkssvcEvent
                Event...........userenv:  Machine  Group  Policy  has  been  applied
      SymbolicLink....Local
      Section.........DFMap0-15555297
      Section.........DfSharedHeapED2256
      Section.........DfSharedHeapE8F975
      Section.........DFMap0-15232696
      Section.........DFMap0-15170325
      Event...........Shell_NotificationCallbacksOutstanding
      Section.........DFMap0-14364985
      Event...........SETTermEvent
      Event...........winlogon:  User GPO Event 112121
...
   Directory.......ObjectTypes
      Type............Directory
      Type............Mutant
      Type............Thread
      Type............Controller
      Type............Profile
      Type............Event
      Type............Type
      Type............Section
```

```
      Type...........EventPair
      Type...........SymbolicLink
      Type...........Desktop
      Type...........Timer
      Type...........File
      Type...........WindowStation
      Type...........Driver
      Type...........WmiGuid
      Type...........Device
      Type...........Token
      Type...........IoCompletion
      Type...........Process
      Type...........Adapter
      Type...........Key
      Type...........Job
      Type...........WaitablePort
      Type...........Port
      Type...........Callback
      Type...........Semaphore
   Directory.......Security
      Event..........TRKWKS_EVENT
      WaitablePort....TRKWKS_PORT
      Event..........LSA_AUTHENTICATION_INITIALIZED
      Event..........NetworkProviderLoad
...
```

EXAMPLE 7-2. *Excerpts from an Object Directory*

The full-featured object browser code inside `w2k_obj.exe` not only displays the directory tree in a more pleasing visual form, but also allows display of additional object features and filtering of object types. Example 7-3 shows the various options offered by the `w2k_obj.exe` command line.

```
// w2k_obj.exe
// SBS Windows 2000 Object Browser V1.00
// 08-27-2000 Sven B. Schreiber
// sbs@orgon.com

Usage: w2k_obj [+-atf] [<type>] [<#>|-1] [/root] [/types]

       +a -a  : show/hide object addresses  (default: -a)
       +t -t  : show/hide object type names (default: -t)
       +f -f  : show/hide object flags      (default: -f)
```

(continued)

```
        <type> : show <type> objects only    (default:   *)
        <#>    : show <#> directory levels   (default:  -1)
        -1     : show all directory levels
        /root  : show ObpRootDirectoryObject tree
        /types : show ObpTypeDirectoryObject tree

Example: w2k_obj +atf *port 2 /root

This command displays all Port and WaitablePort objects,
starting in the root and scanning two directory levels.
Each line includes address, type, and flag information.
```

EXAMPLE 7-3. *The Command Help of* `w2k_obj.exe`

In Example 7-4, I have issued the sample command `w2k_obj +atf *port 2 /root` mentioned in the help screen. It restricts the output to `Port` and `WaitablePort` objects by applying the type filter expression `*port` and includes object body addresses, type names, and flags for each entry. The display is limited to two subordinate directory layers.

```
Root directory contents: (2 levels shown)
____________


 8149CDD0 Directory_____ <32> \
> |_ E26A0540 Port__________ <24> SeLsaCommandPort
> |_ E130CC20 Port__________ <24> XactSrvLpcPort
> |_ E13E2380 Port__________ <24> DbgUiApiPort
> |_ E13E4BA0 Port__________ <26> SeRmCommandPort
> |_ E26A9D20 Port__________ <24> LsaAuthenticationPort
> |_ E13E4CA0 Port__________ <24> DbgSsApiPort
> |_ E13E3260 Port__________ <24> SmApiPort
> |_ E2707680 Port__________ <24> ErrorLogPort
  |_ 81499B70 Directory_____ <32> \ArcName
  |_ 812FDB60 Directory_____ <10> \NLS
  |_ 814940B0 Directory_____ <32> \Driver
  |_ 81490B30 Directory_____ <32> \WmiGuid
  |_ 81499A90 Directory_____ <32> \Device
  |   |_ 814AEA90 Directory_____ <32> \Device\DmControl
  |   |_ 814AE4F0 Directory_____ <32> \Device\HarddiskDmVolumes
  |   |_ 8148BE50 Directory_____ <32> \Device\Ide
  |   |_ 814AB3D0 Directory_____ <32> \Device\Harddisk0
  |   |_ 814852F0 Directory_____ <32> \Device\Harddisk1
  |   |_ 814A9F50 Directory_____ <22> \Device\WinDfs
```

```
   |   \_ 814AB030 Directory_____ <32> \Device\Scsi
   |_ 81319030 Directory_____ <30> \Windows
 > |   |_ E2615520 Port__________ <24> SbApiPort
 > |   |_ E260E1A0 Port__________ <24> ApiPort
   |   \_ 812FC810 Directory_____ <32> \Windows\WindowStations
   |_ 81319150 Directory_____ <30> \RPC Control
 > |   |_ E26B6A20 Port__________ <24> tapsrvlpc
 > |   |_ E3228440 Port__________ <24> OLE3c
 > |   |_ E269F360 Port__________ <24> spoolss
 > |   |_ E269B6E0 Port__________ <24> OLE2
 > |   |_ E2C96C60 Port__________ <24> OLE3f
 > |   |_ E1306BC0 Port__________ <24> OLE3> |   |_ E269BD20 Port__________ <24>
LRPC0000021c.00000001
 > |   |_ E276D520 Port__________ <24> OLE5
 > |   |_ E2699D40 Port__________ <24> OLE6
 > |   |_ E2697C00 Port__________ <24> OLE7
 > |   |_ E26F0AE0 Port__________ <24> ntsvcs
 > |   |_ E26B6B20 Port__________ <24> policyagent
 > |   |_ E2814CA0 Port__________ <24> OLEa
 > |   |_ E29DC3C0 Port__________ <24> OLEb
 > |   |_ E304C8A0 Port__________ <24> OLE40
 > |   |_ E3165660 Port__________ <24> OLE41
 > |   |_ E26979A0 Port__________ <24> epmapper
 > |   |_ E13069A0 Port__________ <24> senssvc
 > |   \_ E2C8D040 Port__________ <24> OLE42
   |_ 812FD030 Directory_____ <30> \BaseNamedObjects
   |   \_ 812FDF50 Directory_____ <30> \BaseNamedObjects\Restricted
   |_ 8149CBD0 Directory_____ <32> \??
   |_ 814B5030 Directory_____ <32> \FileSystem
   |_ 8149CCB0 Directory_____ <32> \ObjectTypes
   |_ 81499C50 Directory_____ <32> \Security
 > |   \_ 8121EB20 WaitablePort__ <24> TRKWKS_PORT
   |_ 8149B2D0 Directory_____ <32> \Callback
   \_ 81446E90 Directory_____ <30> \KnownDlls

54 objects
```

EXAMPLE 7-4. *Output of the Command* `w2k_obj +atf *port 2 /root`

Note that `Directory` objects are always included in the list, even though the type name pattern doesn't match them. Otherwise, it would be unclear to which node in the directory hierarchy the matching objects are assigned. The > characters in the first display column act as visual cues that distinguish the objects with a matching object type from the additional `Directory` objects.

WHERE DO WE GO FROM HERE?

So much could still be said about Windows 2000 internals. But the number of words fitting into a reasonably sized book is limited, so it must end somewhere. The seven chapters of this book were tough reading, but maybe it was thrilling as well. If you are now seeing Windows 2000 with different eyes, I have reached my goal. If you are a programming or debugging tool developer, the programming and interfacing techniques in this book will help you add value to your products that none of the competitive tools can currently offer. If you are developing other kinds of software for Windows 2000, the understanding of the inner system dynamics imparted by this book will help you writing more efficient code that optimally exploits the features of your operating system. I also would like this book to spur the inquiring minds of developers everywhere, kicking off an avalanche of research that unveils the mysteries that still surround most parts of the Windows 2000 kernel. I never believed that treating the operating system as a black box was a good programming paradigm—and I still don't believe it.