used to work around it. Filters can also implement completely new functionality, such as turning disks into partitions or multiple disks into RAID volumes.

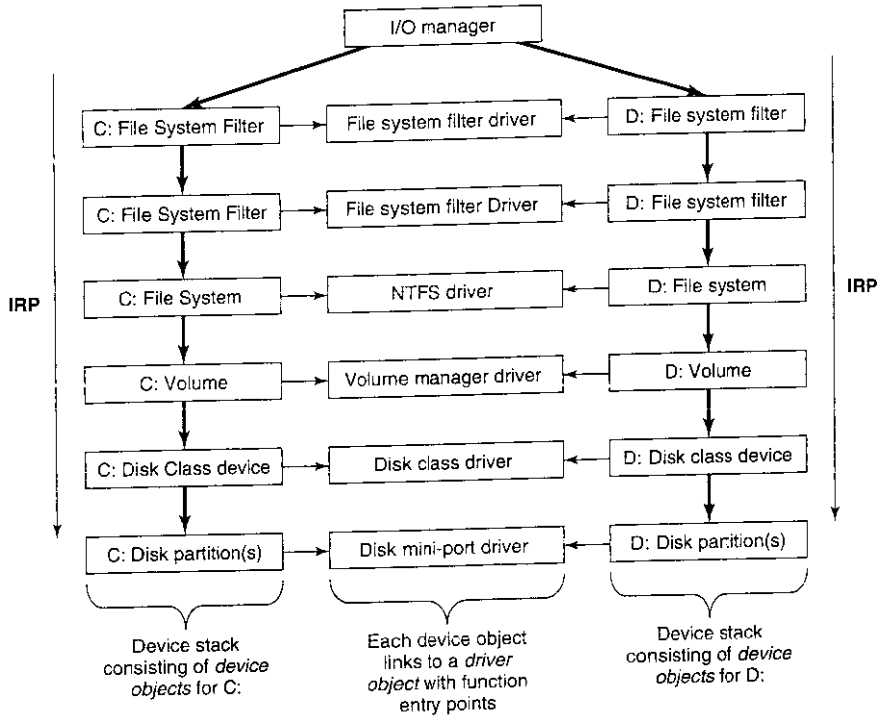

Figure 11-16. Simplified depiction of device stacks for two NTFS file volumes. The I/O request packet is passed from down the stack. The appropriate routines from the associated drivers are called at each level in the stack. The device stacks themselves consist of device objects allocated specifically to each stack.

The file systems are loaded as drivers. Each instance of a volume for a file system has a device object created as part of the device stack for that volume. This device object will be linked to the driver object for the file system appropriate to the volume's formatting. Special filter drivers, called **file system filter drivers**, can insert device objects before the file system device object to apply functionality to the I/O requests being sent to each volume, such as inspecting data read or written for viruses.

The network protocols, such as Windows Vista's integrated IPv4/IPv6 TCP/IP implementation, are also loaded as drivers using the I/O model. For compatibility with the older MS-DOS-based Windows, the TCP/IP driver implements a special protocol for talking to network interfaces on top of the Windows I/O model.

There are other drivers that also implement such arrangements, which Windows calls **mini-ports**. The shared functionality is in a **class driver**. For example, common functionality for SCSI or IDE disks or USB devices is supplied by a class driver, which mini-port drivers for each particular type of such devices link to as a library.

We will not discuss any particular device driver in this chapter, but will provide more detail about how the I/O manager interacts with device drivers in Sec. 11.7.

11.3.2 Booting Windows Vista

Getting an operating system to run requires several steps. When a computer is turned on, the CPU is initialized by the hardware, and then set to start executing a program in memory. But the only available code is in some form of nonvolatile CMOS memory that is initialized by the computer manufacturer (and sometimes updated by the user, in a process called **flashing**). On most PC's this initial program is the BIOS (Basic Input/Output System) which knows how to talk to the standard types of devices found on a PC. The BIOS brings up Windows Vista by first loading small bootstrap programs found at the beginning of the disk drive partitions.

The bootstrap programs know how to read enough information off a file system volume to find the standalone Windows *BootMgr* program in the root directory. *BootMgr* determines if the system had previously been hibernated or was in stand-by mode (special power-saving modes that allow the system to turn back on without booting). If so, *BootMgr* loads and executes *WinResume.exe*. Otherwise it loads and executes *WinLoad.exe* to perform a fresh boot. *WinLoad* loads the boot components of the system into memory: the kernel/executive (normally *ntoskrnl.exe*), the HAL (*hal.dll*), the file containing the SYSTEM hive, the *Win32k.sys* driver containing the kernel-mode parts of the Win32 subsystem, as well as images of any other drivers that are listed in the SYSTEM hive as **boot drivers**—meaning they are needed when the system first boots.

Once the Windows boot components are loaded into memory, control is given to low-level code in NTOS which proceeds to initialize the HAL, kernel and executive layers, link in the driver images, and access/update configuration data in the SYSTEM hive. After all the kernel-mode components are initialized, the first user-mode process is created using for running the *smss.exe* program (which is like */etc/init* in UNIX systems).

The Windows boot programs have logic to deal with common problems users encounter when booting the system fails. Sometimes installation of a bad device driver, or running a program like *regedit* (which can corrupt the SYSTEM hive), will prevent the system from booting normally. There is support for ignoring recent changes and booting to the *last known good* configuration of the system. Other boot options include **safe-boot** which turns off many optional drivers and

the **recovery console**, which fires up a *cmd.exe* command-line window, providing an experience similar to single-user mode in UNIX.

Another common problem for users has been that occasionally some Windows systems appear to be very flaky, with frequent (seemingly random) crashes of both the system and applications. Data taken from Microsoft's On-line Crash Analysis program provided evidence that many of these crashes were due to bad physical memory, so the boot process in Windows Vista provides the option of running an extensive memory diagnostic. Perhaps future PC hardware will commonly support ECC (or maybe parity) for memory, but most of the desktop and notebook systems today are vulnerable to even single-bit errors in the billions of bits of memory they contain.

11.3.3 Implementation of the Object Manager

The object manager is probably the single most important component in the Windows executive, which is why we have already introduced many of its concepts. As described earlier, it provides a uniform and consistent interface for managing system resources and data structures, such as open files, processes, threads, memory sections, timers, devices, drivers, and semaphores. Even more specialized objects representing things like kernel transactions, profiles, security tokens, and Win32 desktops are managed by the object manager. Device objects link together the descriptions of the I/O system, including providing the link between the NT namespace and file system volumes. The configuration manager uses an object of type **Key** to link in the registry hives. The object manager itself has objects it uses to manage the NT namespace and implement objects using a common facility. These are directory, symbolic link, and object-type objects.

The uniformity provided by the object manager has various facets. All these objects use the same mechanism for how they are created, destroyed, and accounted for in the quota system. They can all be accessed from user-mode processes using handles. There is a unified convention for managing pointer references to objects from within the kernel. Objects can be given names in the NT namespace (which is managed by the object manager). Dispatcher objects (objects that begin with the common data structure for signaling events) can use common synchronization and notification interfaces, like WaitForMultipleObjects. There is the common security system with ACLs enforced on objects opened by name, and access checks on each use of a handle. There are even facilities to help kernel-mode developers debug problems by tracing the use of objects.

A key to understanding objects is to realize that an (executive) object is just a data structure in the virtual memory accessible to kernel mode. These data structures are commonly used to represent more abstract concepts. As examples, executive file objects are created for each instance of a file system file that has been opened. Process objects are created to represent each process.

A consequence of the fact that objects are just kernel data structures is that when the system is rebooted (or crashes) all objects are lost. When the system boots, there are no objects present at all, not even the object type descriptors. All object types, and the objects themselves, have to be created dynamically by other components of the executive layer by calling the interfaces provided by the object manager. When objects are created and a name is specified, they can later be referenced through the NT namespace. So building up the objects as the system boots also builds the NT namespace.

Objects have a structure, as shown in Fig. 11-17. Each object contains a header with certain information common to all objects of all types. The fields in this header include the object's name, the object directory in which it lives in the NT namespace, and a pointer to a security descriptor representing the ACL for the object.

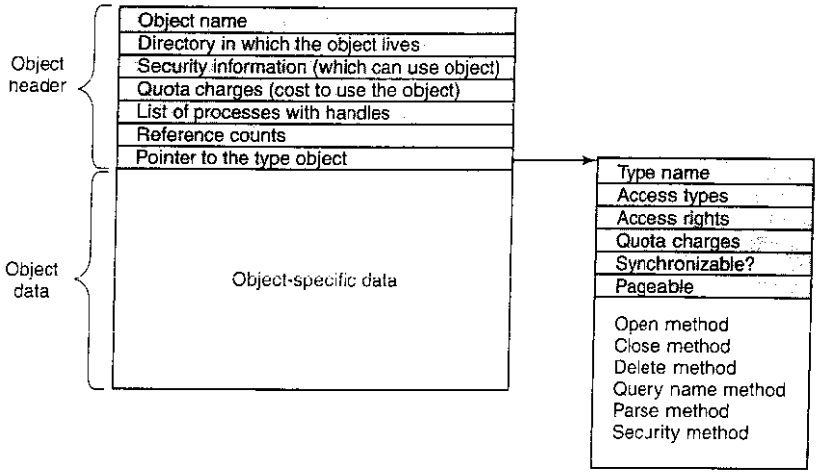

Figure 11-17. The structure of an executive object managed by the object manager

The memory allocated for objects comes from one of two heaps (or pools) of memory maintained by the executive layer. There are (malloc-like) utility functions in the executive that allow kernel-mode components to allocate either pageable kernel memory or nonpageable kernel memory. Nonpageable memory is required for any data structure or kernel-mode object that might need to be accessed from a CPU priority level of 2 or more. This includes ISRs and DPCs (but not APCs), and the thread scheduler itself. The pagefault handle also requires its data structures to be allocated from nonpageable kernel memory to avoid recursion.

Most allocations from the kernel heap manager are achieved using per-processor lookaside lists which contain LIFO lists of allocations the same size. These