A Comparison of the Linux and Windows Device Driver Architectures

Melekam Tsegaye
Rhodes University, South Africa
g98t4414@campus.ru.ac.za

Richard Foss
Rhodes University, South Africa
r.foss@ru.ac.za

Abstract: In this paper the device driver architectures currently used by two of the most popular operating systems, Linux and Microsoft's Windows, are examined. Driver components required when implementing device drivers for each operating system are presented and compared. The process of implementing a driver, for each operating system, that performs I/O to a kernel buffer is also presented. The paper concludes by examining the device driver development environments and facilities provided to developers by each operating system.

1. Introduction

Modern operating system kernels consist of a number of components such as a memory manager, process scheduler, hardware abstraction layer (HAL) and security manager. For a detailed look at the Windows kernel refer to [Russinovich, 98] and for the Linux kernel [Rusling, 99], [Beck *et al*, 98]. The kernel can be viewed as a black box that should know how to interact with the many different types of hardware devices that exist and the many more devices that do not yet exist. Creating a kernel that has inbuilt functionality for interacting with all known hardware devices may be possible but is not practical. It would consume too many system resources, needlessly.

1.1. Kernel Modularity

A kernel is not expected to know how to interact with new types of devices that do not yet exist at the time of its creation. Instead modern operating system kernels allow their functionality to be extended by the addition of device driver modules at runtime. A module implements functionality that will allow the kernel to interact with a particular new device. Each module implements a routine that the kernel will call at module load time and a routine that gets called at module removal time. Modules also implement various routines that will implement I/O functionality for transferring data to and from a device as well as a routine for issuing device I/O control instructions to a device. The above applies to both the Linux and Windows driver architectures.

1.2. Organisation of this paper

The material in this paper is divided into the following sections:

- General driver architecture of the two operating systems (section 2)
- Driver architecture components of each operating system (sections 3)
- Implementation of a driver that performs I/O to a kernel buffer (section 4)
- Driver development environments and facilities offered by the two operating systems to developers (section 5)

1.3. Related Work

The Windows device driver architecture is documented by documentation that accompanies the Windows Device Driver Development kit [Microsoft DDK, 02]. Further, the works produced by Walter Oney [Oney, 99] and Chris Cant [Cant, 99] present a detailed account of the Windows Driver Architecture. The Linux device driver architecture is documented well by the freely available publication authored by Rubini *et al* [Rubini *et al*, 01].

2. Device Driver Architectures

A device driver enables the operation of a piece of hardware by exposing a programming interface that allows a device to be controlled externally by applications and parts of an operating system. This section presents the driver architectures currently in use by two of the most commonly used operating systems, Microsoft's Windows and Linux, and the origin of their architecture.

2.1. Origin of the Linux Driver Architecture

Linux is a clone of the UNIX operating system first created by Linux Travolds [Linus FAQ, 02], [LinuxHQ, 02]. It follows that the Linux operating system utilises a similar architecture to UNIX systems. UNIX operating systems view devices as file system nodes. Devices appear as special file nodes in a directory designated by convention to contain device file system node entries [Deitel, 90]. The aim of representing devices as file system nodes is so that applications can access devices in a *device independent* manner [Massie, 86],[Flynn *et al*, 97]. Applications can still perform device dependent operations with a device I/O control operation. Devices are identified by major and minor numbers. A major number serves as an index to an array of drivers and a minor number is used to group similar physical devices [Deitel, 90]. Two types of UNIX devices exist, *char* and *block*. *Char* device drivers manage devices that are accessed sequentially with no buffering, and *Block* device drivers manage devices where random access is possible, and data is accessed in blocks. Buffering is also utilised in block device drivers. A block device must be mounted as a file system node for it to be accessible [Beck *et al*, 98].

Linux retains much of the UNIX architecture, the difference being that *char* device nodes corresponding to block devices have to be created in UNIX systems, whereas in Linux, the Virtual File System (VFS) interface blurs the distinction between *char* and *block* devices [Beck *et al*, 98]. Linux also introduces a third type of device called a *network* device. Network device drivers are accessed in a different way to *char* and *block* drivers. A set of APIs different from the file system I/O APIs are used e.g. the socket API, which is used for accessing network devices.

2.2. Origin of the Windows Driver Architecture

In 1980, Microsoft licensed the UNIX operating system from Bell labs, later releasing it as the XENIX operating system. With the first IBM PC, MS DOS version 1 was released in 1981. MS DOS version 1 had a similar driver architecture to UNIX systems based on XENIX [Deitel, 90]. The difference to UNIX systems was that the operating system came with built in drivers for common devices. Device entries did not appear as file system nodes. Instead reserved names were assigned to devices. E.g. CON was the keyboard or screen, PRN the printer and AUX the serial ports. Applications could open these devices and obtain a handle to associated drivers as they would with file system nodes, and perform I/O to them. The operating system, transparent to applications, translated reserved device names to devices that its drivers managed. MS DOS version 2 introduced the concept of loadable drivers. Since Microsoft had made the interface to its driver architecture open, this encouraged third party device manufacturers to produce new devices [Davis, 83]. Drivers for these new devices could then be supplied by hardware manufacturers and be loaded/unloaded at runtime into the kernel, manually.

Later on, Windows 3.1 was released by Microsoft. It had support for many more devices and utilised an architecture based on MS DOS. With its later operating systems, Windows 95, 98 and NT, Microsoft introduced the Windows Driver Mode (WDM). The WDM came about because Microsoft wanted to make device drivers *source code* compatible with all of its new operating systems [Microsoft WDM, 02]. Thus, the advantage of making drivers WDM compliant is that once created, a driver need only be recompiled before it is usable on any of Microsoft's later operating systems.

2.3. The Windows Driver Architecture

There are two types of Windows drivers, legacy and Plug and Play (PnP) drivers. The focus here is only on PnP drivers, as all drivers should be PnP drivers where the possible. PnP drivers are user friendly since very little effort is required from users to install them. Another benefit of making drivers PnP is that they get loaded by the operating system only when needed, thus they do not use up system resources needlessly. Legacy drivers were implemented for Microsoft's earlier operating systems and their architecture is outdated. The Windows Driver Model (WDM) is a standard model specified by Microsoft [Microsoft DDK, 02]. WDM drivers are usable on all of Microsoft's recent operating systems (Windows 95 and later).

2.3.1. The WDM driver architecture

There are three classes of WDM drivers: filter, functional and bus drivers [Oney, 01]. They form the stack illustrated in figure 2.3. In addition, WDM drivers must be PnP aware, support power management and Windows Management Instrumentation. Figure 2.3 shows how data and messages are exchanged between the various driver layers. A standard structure called an I/O Request Packet (IRP) is used for communication. Whenever a request is made from an application to a driver, the I/O manager builds an IRP and passes it down to the driver, which processes it, and when done, 'completes' the IRP [Cant, 99]. Not every IRP filters down to a bus driver. Some IRPs get handled by the layers above and are returned to the I/O manager from there. Hardware access to a device is done through a hardware abstraction layer (HAL).

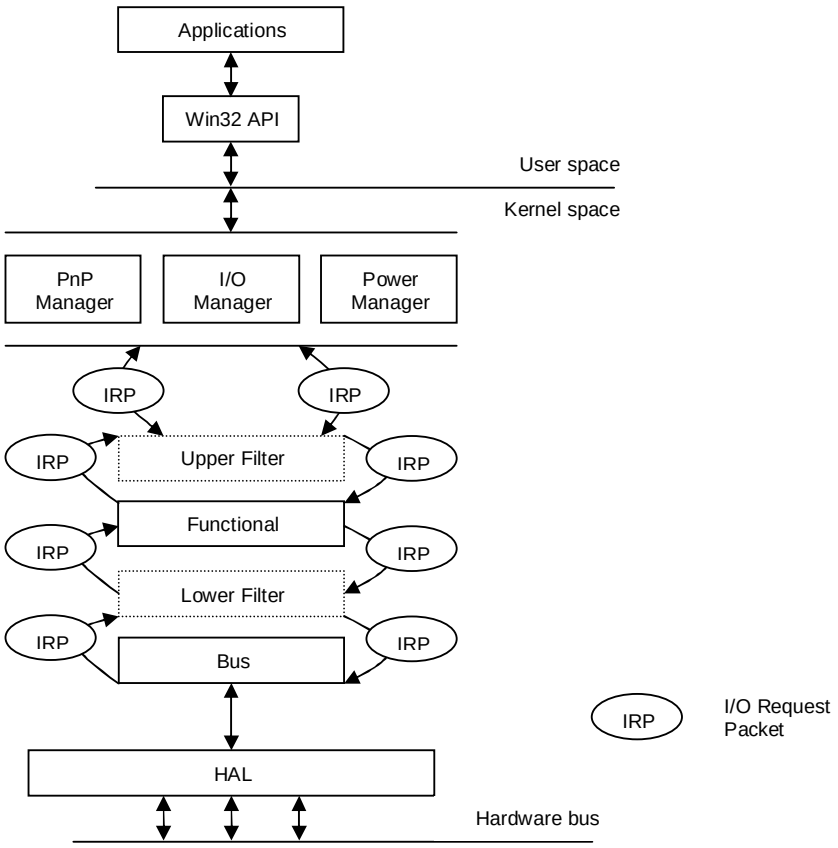

Figure 2.3 The WDM Driver Architecture

2.4. The Linux Driver Architecture

Drivers in Linux are represented as modules, which are pieces of code that extend the functionality of the Linux kernel [Rubini *et al*, 01]. Modules can be layered as shown in figure 2.4. Communication between modules is achieved using function calls. At load time a module exports all functions it wants to make public to a symbol table that the Linux kernel maintains. These functions are then visible to all modules. Access to devices is done through a hardware abstraction layer (HAL) whose implementation depends on the hardware platform that the kernel is compiled for, e.g. x86 or SPARC.

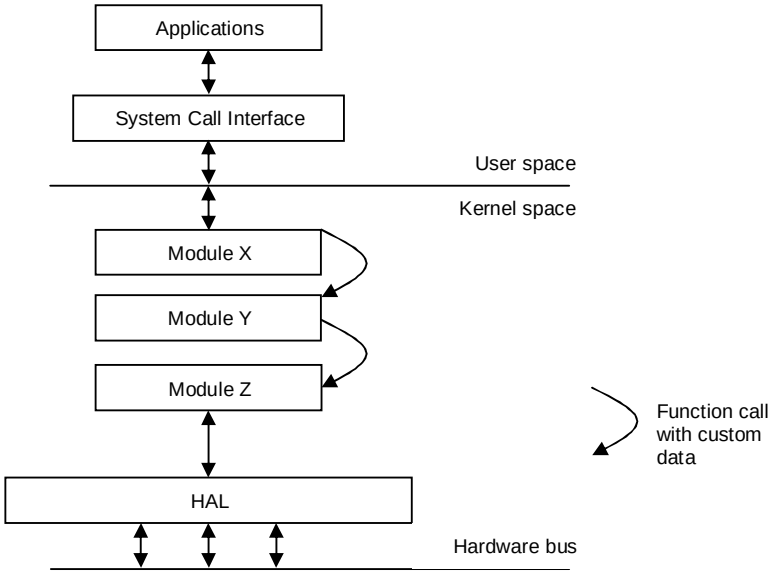

Figure 2.4 The Linux Driver Architecture

2.5. The Linux and Windows Driver Architectures Compared

As can be seen in figures 2.3 and 2.4, a number of similarities exist between the two operating systems. On both systems, drivers are modular components that extend the functionality of the kernel. Communication between driver layers in Windows is through the use of I/O Request Packets (IRPs) supplied as arguments to standard system and driver defined functions, whereas in Linux function calls with parameters customized to a particular driver are used. Windows has separate kernel components that manage PnP, I/O and Power. These components send messages to drivers using IRPs at appropriate times.

In Linux, there is no clear distinction between layered modules, i.e. modules are not categorised as bus, functional or filter drivers. There is no clearly defined PnP or Power manager in the kernel that sends standardised messages to modules at appropriate times. The kernel may have modules loaded that implement Power Management or PnP functionality, but the interface of these modules to drivers is not clearly specified. This functionality is likely to be incorporated in later Linux kernels as the Linux kernel is always in development. Once data is passed to a driver that is part of a stack of modules by the kernel, the data may be shared with other drivers in the stack through an interface specific to that set of drivers.

In both environments, hardware access through a HAL interface is implemented for the specific platform the kernel is compiled for, i.e. x86, SPARC etc. A common feature of both architectures is that drivers are modules that can be loaded into a kernel at runtime. Each module contains an entry point that the kernel knows to start code execution from. A module will also contain routines that the kernel knows to call when an I/O operation is requested to a device managed by that module. This enables the kernel to provide a device independent interface to the application layer. A more in-depth comparison of driver components from the two architectures is presented later in Section 3.3.

3. Drivers Components

The process of creating a device driver requires knowledge of how the associated hardware device is expected to operate. For example, just about every device will allow its clients to read data from and write data to it. In this section driver components that must be implemented by all drivers are presented, as well as a comparison of the two operating systems' driver components. The implementation of a driver that performs I/O to a kernel buffer is also presented. The section concludes with a look at the driver development environments and facilities offered by each operating system.

3.1. Windows Driver Components

Drivers in Windows consist of various routines. Some are required, others optional. This section presents the routines that every driver must implement. A device driver in Windows is represented by a structure called a *DriverObject*. It is necessary to represent a driver with a structure such as a driver object because the kernel implements various routines that can be performed for every driver. These routines, discussed in the following sections, operate on a driver object.

3.1.1. Driver Initialisation

Every device driver in Windows contains a routine called *DriverEntry*. As its name suggests, this routine is the first driver routine executed when a driver is loaded and is where initialisation of the device driver's device object is performed. Microsoft's DDK [Microsoft DDK, 02] states that a driver object represents a currently loaded kernel driver whereas a device object represents a physical, logical or virtual device. A single loaded kernel driver (represented by a driver object) can manage multiple devices (represented by device objects). During initialisation, fields in the device object that specify the driver's *unload* routine, *add device* routine and *dispatch* routines are set. The *unload* routine is a routine that is called when the driver is about to be unloaded so that it can perform cleanup operations e.g. freeing up memory allocated off the kernel heap. *addDevice* is a routine that is called after the *DriverEntry* routine if the driver being loaded is a PnP driver, while the *dispatch* routines are routines that implement driver I/O operations.

3.1.2. The AddDevice Routine

PnP drivers implement a routine called *AddDevice*. In this routine a device object is created at, which time space for storing global data for a device is allocated. Device resource allocation and initialisation is also performed. Device objects are referred to by different names depending on where they where created. If a device object is created in a currently loaded driver to manage that driver, it is called a Function Device Object (FDO). If it is a device object from a lower driver in a stack of drivers, it is called a Physical Device Object (PDO). If it is a device object from an upper driver in a stack of drivers, it is called a Filter Driver Object (FIDO).

3.1.2.1.Creating a device object

A device object corresponding to a device is created using the I/O Manager routine called *IoCreateDevice* inside the *add device* routine. The most important requirements for *IoCreateDevice* are a name for the device object and device type. The name allows applications and other kernel drivers to gain a handle to the driver, in order to perform I/O operations. The device type specifies the type of device the driver is used for, for example a storage device.

3.1.2.2. Global Driver Data

When a device object is created it is possible to associate with it a block of memory, called *DeviceExtension* in Windows, where custom driver data can be stored. This is an important facility, as it eliminates the need to use global data structures in driver code, which can be difficult to manage. For example, in the case where a local variable with the same name as a global variable is declared in a routine mistakenly, the driver writer may find it difficult to track a bug in the driver. It also makes it easier to manage device object specific data, when more than one device object exists in a single driver, as is the case when a bus driver manages child physical device objects for devices present on its bus.

3.1.2.3.Device naming

A device can be named at device object creation time. This name can be used for accessing a handle to a driver. The handle is then used for performing I/O. Microsoft recommends not naming functional device objects created in filter and functional drivers. As pointed out by Oney [Oney, 99], if a device object is named, any client can open the device object and perform I/O for non-disk device drivers. This is because the default access control status Windows gives to non-disk device objects is an unrestricted one. Another problem is that the name specified does not have to follow any naming protocol, so the name specified may not be a well chosen one. For example two driver writers may come up with the same name for their device objects, which would cause a clash.

Windows supports a second device object naming scheme using device interfaces. Device interfaces are constructed with 128 bit globally unique identifiers (GUIDs) [Open Group, 97]. A GUID can be generated using a utility provided by the Microsoft DDK. Once generated, a GUID can be publicised. A driver registers the

GUID for a device interface in its *add device* routine through a call to the I/O manager routine *IoRegisterDeviceInterface*. Once registered, the driver must enable the device interface through a call to the I/O manager routine *IoSetDeviceInterfaceState*. The registration process will add an interface data entry to the Windows registry file, which can be accessed by applications.

3.1.2.4. Driver Access from an Application

An application that wants to perform I/O operations with a device driver must obtain a handle to a device driver through the *CreateFile* Win32 API call. It requires a path to a device such as \\device\devicex. Named devices will have their names appear in the name space called \\device, thus the previous path is for a device named *devicex*. *CreateFile* also requires access mode flags such as read, write and file sharing flags for the device.

Accesses to unnamed devices that have registered a device interface are performed differently as shown in figure 3.1.2.4. This requires obtaining a handle to a device information structure using the driver's GUID, and calling the *SetupDiGetClassDevs* Win32 API routine. This is only possible if the driver registered a device interface, through which applications can access the device (called a device interface class).

Each time a driver calls the I/O manager routine *IoRegisterDeviceInterface*, a new instance of the device interface class is created. Once a device information handle is obtained by an application, multiple calls to the Win32 API routine *SetupDiEnumDeviceInterfaces* will return device interface data for each instance of the device interface class. Lastly, a device path for each of the driver instances can be retrieved from the interface data obtained from the previous call with another Win32 API routine, *SetupGetDeviceInterfaceDetail*. *CreateFile* can then be called with the device path for the device instance of interest, to obtain a handle for performing I/O.

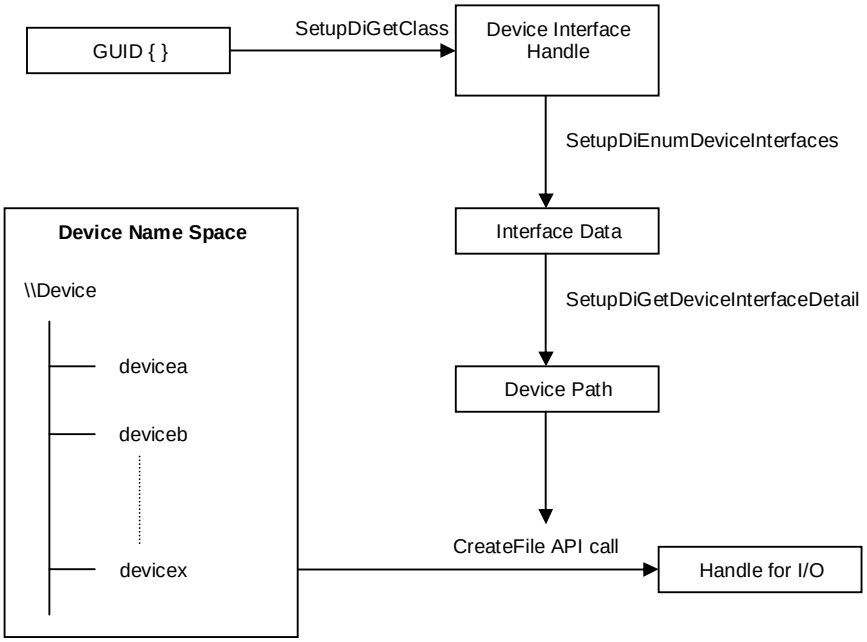

Figure 3.1.2.4 Obtaining a handle an application can use for I/O from a device GUID.

3.1.2.5. Device Object Stacking

When the *add device* routine is called by the PnP manager, one of the parameters passed to it is a device object (PDO) for a driver below the current one. Device object stacking is performed in the *add device* routine so that IRPs sent by drivers in the layer below the driver being loaded can be received by it. Device object

stacking is achieved by a call to the I/O Manager routine *IoAttachDeviceToDeviceStack* as shown in figure 3.1.2.5. A physical device object (PDO) is required, which is lower in the stack than the new device object when calling *IoAttachDeviceToDeviceStack*. The routine attaches the specified device object to the top of the driver stack and returns a device object that is one below the new one e.g. in the example shown on figure 3.1.2.5 this would be lower device object X. The lower physical device object (PDO) can be any number of layers below the new device object but *IoAttachDeviceToStack* returns the device object one below the current one.

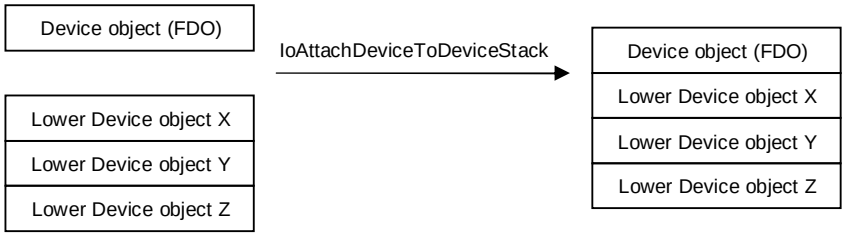

Figure 3.1.2.5 Attaching a device object to the top of a device object stack.

3.1.2.6. User to Kernel and Kernel to User Data Transfer Modes in Windows

The mode used to transfer data from kernel space to user space and vice versa is specified in the flags field of a device object. There are three modes: buffered I/O, direct I/O and I/O that does not use any of the latter methods termed "method neither I/O". Figure 3.1.2.6 illustrates the three modes. In buffered I/O mode the operating system allocates a kernel buffer that can handle a request. In the case of a write operation, the operating system validates the supplied user space buffer and copies data from the user space buffer to the newly allocated kernel buffer and passes the kernel buffer to the driver. In the case of reads, the operating system validates the user space buffer and copies data from the newly allocated kernel buffer to the user space buffer. The kernel buffer is accessible to drivers as the *AssociatedIrp.SystemBuffer* field of an IRP. Drivers read from or write to this buffer to communicate with applications when buffered I/O is in use.

Direct I/O is the second I/O method that can be used for data exchanges between applications and a driver. An application-supplied buffer is locked into memory by the operating system, so that it will not be swapped out, and a memory descriptor list (MDL) for the locked memory is passed to a driver. An MDL is an opaque structure. Its implementation details are not visible to drivers. The driver then performs DMA to the user space buffer through the MDL. The MDL is accessible to drivers through the *MdlAddress* field of an IRP. The advantage of using direct I/O is that it is faster than buffered I/O since no copying of data to and from user and kernel space is necessary and I/O is performed directly into a user space buffer.

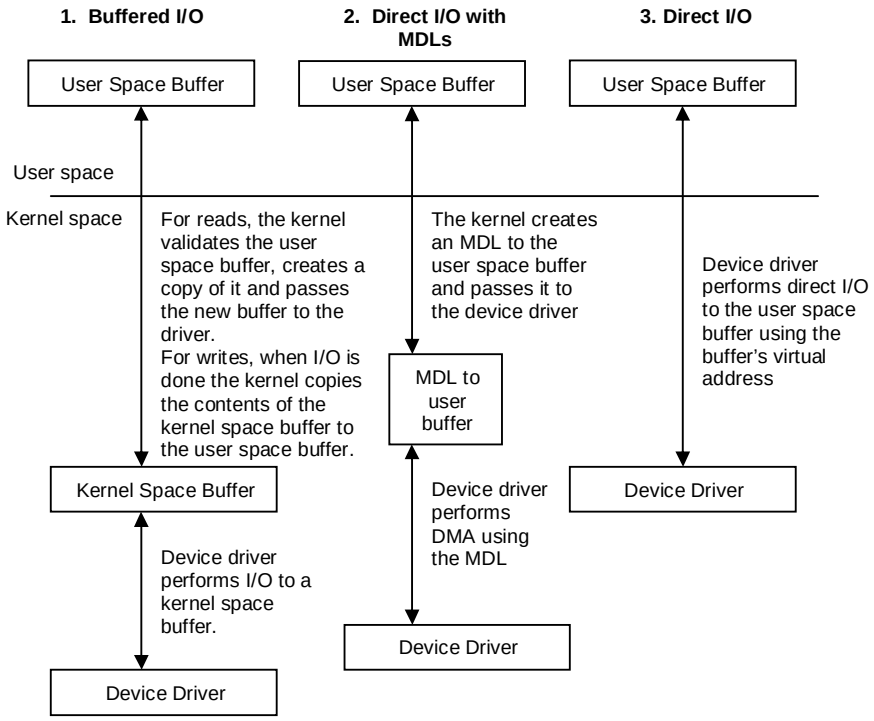

Figure 3.1.2.6 The three ways in which data from kernel to user and user to kernel space is exchanged.

The third method for I/O is neither buffered nor uses MDLs. Instead the operating system passes the virtual address for a user space buffer to the driver. The driver is then responsible for checking the validity of the buffer before it makes use of it. In addition, the user space buffer is only accessible if the current thread context is the same as the application's, otherwise a page fault will occur since the virtual address is valid only while that application's process is active.

3.1.3. Dispatch Routines

Dispatch routines are routines that handle incoming I/O requests packaged as IRPs (I/O request packets). When an IRP arrives (e.g. when an application initiates I/O), an appropriate dispatch routine is selected from the array of routines specified in the *MajorFunction* field of a driver object as shown in figure 3.1.3. These dispatch routines are initialised in the driver's entry routine. Every IRP is associated with an I/O stack location structure (used for storing an IRP's parameters) when created. This structure contains a field, which specifies the dispatch routine the IRP is meant for and the relevant parameters for that dispatch routine. The I/O manager determines from an IRP which dispatch routine it will send IRPs to.

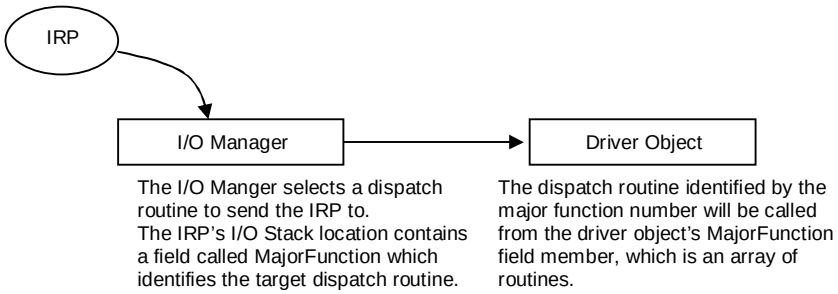

Figure 3.1.3 dispatching IRP's to dispatch routines.

Thus IRPs are routed to an appropriate driver supplied routine so that they can be handled there. Required dispatch routine IDs are shown in table 3.1.3. They are indexes for the array of routines specified by the MajorFunction field of a device object. The dispatch routines have custom driver supplied names that are implemented by the driver. They all accept an IRP and a device object to which the IRP is being sent.

IRP_MJ_PNP	Handles PnP messages
IRP_MJ_CREATE	Handles the opening of device to gain a handle
IRP_MJ_CLEANUP	Handles the closing of the device handle gained above
IRP_MJ_CLOSE	Same as clean up, called after cleanup
IRP_MJ_READ	Handles a read request to a device
IRP_MJ_WRITE	Handles a write request to a device
IRP_MJ_DEVICE_CONTROL	Handles a I/O control request to a device
IRP_MJ_INTERNAL_DEVICE_CONTROL	Handles driver specific I/O control requests
IRP_MJ_SYSTEM_CONTROL	Handles WMI requests
IRP_MJ_POWER	Handles power management messages

Table 3.1.3 Required Windows driver dispatch routines

3.1.4. Windows Driver Installation

Windows uses installation information contained in a text file called an INF file to install drivers. The creator of a driver is responsible for providing an INF file for the driver. A GUI application that is provided with the Windows DDK called *GenInf* allows the generation of an INF file for a driver. It requires a company name and a Windows Device class under which the driver will be installed. Windows has various pre-defined device classes for installed drivers. The Windows device manager applet, accessible through the system control panel applet, shows all installed drivers categorised using these device classes. Examples of existing classes are the 1394 and PCMCIA device classes. A custom device class can be added by adding a *ClassInstall32* section in the INF file.

The hardware ID for a PnP-aware device must also be specified in the INF file since it will be used by the system to identify the device when the device is inserted into the system. A hardware ID is an identification string used by the PnP manager to identify devices that are inserted into the system. Microsoft publishes PnP hardware IDs for the various devices that are usable with the Windows operating system. This hardware ID is stored on the hardware device and read off the device by the system when that device is inserted into the system. Once an INF file for a new device is successfully installed into the system, the driver for that device (which has a specific hardware ID) will be loaded each time the device is inserted into the system and unloaded when the device is removed from the system.

3.1.5. Obtaining Driver Usage Information in Windows

The device manager found in the control panel system applet provides driver information for users. It lists all currently loaded drivers and information on the providers of each driver and their resource usage. It also displays drivers that failed to load and their error codes.

3.2. Linux Driver Architecture Components

Device drivers in Linux are similar to those in Windows in that they too are made up of various routines that perform I/O and device control operations. There is no driver object visible to a driver, instead drivers are internally managed by the kernel.

3.2.1. Driver Initialisation

Every driver in Linux contains a register driver routine and a deregister driver routine. The register driver routine is the counterpart to the Windows driver entry routine. Driver writers use the *module_init* and *module_exit* kernel defined macros to specify custom routines that will be designated as the register and deregister routines.

3.2.1.1. Driver Registration and Deregistration

The routine designated by the *module_init* macro as the registration routine is the first routine executed when a driver is loaded. The driver is registered here by using a kernel character device registration routine called *register_chrdev*. The important requirements for this routine are a name for the driver, a driver major number (discussed later in section 3.2.2) and a set of routines for performing file operations. Other driver-specific initialisation should take place in this routine. The deregistration routine gets executed when the driver is being unloaded. Its function is to perform cleanup operations before a driver is unloaded. A call to the kernel routine *unregister_chrdev* with a device name and major number is necessary when deregistering a driver that was previously registered with a *register_chrdev* call.

3.2.2. Device Naming

In Linux, devices are named using numbers in the range 0 to 255, called device major numbers. This implies that there can be a maximum of 256 usable devices i.e. devices that an application can gain a handle to, but each driver for such a major device can manage as many as 256 additional devices. These driver-managed devices are numbered using numbers in the range 0 to 255, called device minor numbers. It is therefore possible for applications to gain access up to 65535 (256x256) devices. Major numbers are assigned to well known devices for example major number 171 is assigned to IEEE1394 devices. The file *Documentation/devices.txt* in the Linux kernel source tree contains all major number assignments and a contact address for the device number registration authority. Currently, major numbers 240-254 are available for experimental use. A driver can specify a major number of 0 to request automatic assignment of a major number for a device, if one is available. The use of major number 0 for this purpose does not cause problems, as it is reserved for the null device and no new driver should register itself as a the null device driver.

3.2.2.1. Driver Access from an Application

Drivers are accessed by applications through file system entries (nodes). By convention, the drivers directory is */dev* on a particular system. Applications that want to perform I/O with a driver use the *open* system call to obtain a handle to a particular driver. The *open* system call requires a device node name such as /dev/tty and access flags. After obtaining a handle, the application can use the handle in calls to other system I/O calls such as read, write and IOCTL.

3.2.3. File operations

In Windows, dispatch routines were set up in the driver entry routine of a driver. In Linux, these dispatch routines are known as file operations and are represented by a structure called *file_operations*. A typical driver would implement the file operations listed in table 3.2.3.

Open	Handles the opening of device to gain a handle
Release	Handles the closing of device handled gained above

Read	Handles a read request to a device
Write	Handles a write request to a device
Llseek	Handles a seek operation on a device
IoCtl	Handles a device control request for a device

Table 3.2.3 Most commonly defined driver file operations in Linux

These file operations are specified during driver registration. A structure called *file* is created by the kernel whenever an application requests a handle to a device and is supplied to the driver whenever one of the file operation routines is called. The file operation routines serve many clients, each represented by the *file* structure. The structure has a field named *f_op*. This field is a pointer to the original set of file operations that were specified at registration time of a major driver. It is therefore possible to change the original file operations during a call to any of the file operation routines by changing the value of the field named *f_op* to point to a new set of file operations.

3.2.3.1. Global Driver Data

Whenever an application issues the system *open* call on a device file node in */dev*, the application, gets back a handle to a device from the operating system. At this time the driver's *open* function is called with a file structure created for that open call. This file structure is passed by the kernel to the driver whenever any of the file operations routines are executed. The *private_data* field of the *file* structure can be any driver-supplied custom data structure. Driver private data is usually set up in the *open* file operations function by allocating memory, and freed in the release file operations function. The private data field of the *file* structure can be used to point to data that is global to a driver instead of using global variables.

3.2.4. How Driver Major and Minor Numbers Work

3.2.4.1.The Problem

In Linux only one driver can register itself to manage a device with a particular major number i.e. driver registration uses only a major number. If for example, two device nodes /dev/device1 (major no 4 minor 1) and /dev/device2 (major number 4 minor number 2) exit, only one driver can handle requests, from applications, to these two device nodes. This restriction exists because no driver registration facility exists whereby a driver can register itself to manage a device with major number x and minor number y.

3.2.4.2.The Workaround

- A driver is loaded to manage devices with major number 4. This driver registers itself with the kernel (see section 3.2.1.1 for how this is done).
- Two separate drivers are loaded. One manages a device with major number 4 minor 1 and the other a device with major number 4 minor 2. These drivers do not register themselves with the kernel, but with the driver that manages devices with major number 4. This driver responsible for implementing the registration facility and keeping track of drivers that register with it.
- When an application opens any one of the two device nodes (/dev/device1 or /dev/device2), the open routine of the driver registered to manage devices with major number 4 is called by the kernel. A *file* structure that represents the opened device is passed to this open routine.
- At this point, the driver that manages devices with major number 4 alters the file operation function pointers (*f_op* member of the *file* structure) to point to I/O routines implemented by the driver that manages the opened device. Opens of minor devices by applications are identified by the driver that manages devices with major number 4 in the following manner:
    - A structure called *inode* is also passed to the *open* routine. This structure contains a field named *i_rdev*, which specifies the major and minor numbers for the device the open operation was targeted at. The kernel macros MINOR and MAJOR can be used to extract these values from the *i_rdev* field. In this example the MAJOR number would be 4 and the minor number either 1 or 2. The driver that manages devices with major number 4 can then locate a minor device driver from its registration database using this information.

3.2.5. User to Kernel and Kernel to User Data Transfer Modes in Linux

In Linux, three ways exist for drivers and applications to exchange data to and from kernel and user space. These are buffered I/O, direct I/O and mmap. In buffered I/O mode, data is copied by the kernel from user space to a kernel space buffer before it is used inside a driver. Unlike Windows the Linux kernel does not perform buffering for I/O automatically. Instead, kernel user space access routines are made available that allow copying of data to and from user space by drivers. In direct I/O mode, drivers can read and write data to and from user space buffers directly. This is achieved through the *kiobuf* interface, which involves mapping a user space buffer to a *kiobuf* defined structure through a *kiobuf* kernel call. The operation locks a user space buffer so that it does not get swapped out and is always available for device I/O. The third method, called mmap, involves the driver mapping a chunk of kernel memory to user space using mmap kernel calls, so that applications can perform I/O to the mapped kernel memory. [Rubini *et al*, 01].

3.2.6. Linux Driver Installation

Drivers are installed in Linux by transferring the driver files into a system specific directory. In the RedHat distribution [Redhat, 02], modules are located in the directory */lib/modules/kernel_version* where *kernel_version* specifies the version of the kernel currently loaded e.g. 2.4.19. A configuration file called modules.conf located in the system's configuration file directory e.g. */etc*, is used by the kernel while loading modules. It can be used to override the location for a particular driver. It is also used for defining other module loading options, such as defining parameters to be passed to a driver when loading it.

Module loading and unloading is performed using programs that come with the kernel module utilities package called insmod, modprobe and rmmod. *Insmod* and *modprobe* load the binary image of a driver into the kernel and rmmod removes it. Another program called *lsmod* lists all currently loaded modules. Insmod will attempt to load a module and return an error if the module being loaded depends on other modules. *modprobe* will try to satisfy module dependencies by attempting to load any modules the current module may depend on before loading it. The module dependency information is obtained from a file called modules.dep located in the system's modules directory.

Before a driver can be accessible to applications, a device node (see sections 2.1 and 3.2.2.1 for how Linux represents devices in the system) for that driver, with the devices major and minor numbers, must be created in the devices directory */dev*. A system program called *mknod* is used for this purpose. When creating a device node, it is necessary to specify whether the node is for a character device or a block device.

3.2.7. Obtaining Driver Usage Information in Linux

It is often necessary to check the status of loaded drivers in a system. In Linux, the proc file system is used to publish kernel information for application usage. The proc file system is just like any other file system. It contains directories and file nodes that applications can access and perform I/O operations with. The difference between files on the proc file system and ordinary files is that data from I/O operations on a proc file entry gets passed to and from kernel memory instead of disk storage. The proc file system is a communication medium for applications and kernel components. For example, reading data from the file */proc/modules* will return currently loaded modules and their dependencies. The proc file system is very useful for obtaining driver status information and publishing driver specific data for application use.

3.3. The Windows and Linux Driver Architecture Components Compared

Drivers in both Windows and Linux are dynamically loadable modules that consist of various routines that perform I/O. When loading a module, the kernel will locate a the routine designated by the particular operating system as the driver entry routine and It will start driver code execution from there.

3.3.1. Driver Routines

Drivers in both systems have initialisaton and de-initialisation routines. In Linux, both these routines can have custom names. In Windows, the initialisation routine name is fixed (called *DriverEntry*) but the de-initialisation routine can be a custom one. Windows manages a driver object structure for each loaded driver. Multiple instances of a driver are each represented by a device object. In Linux, the Kernel maintains information for each driver registered to manage a device major number. i.e. for each driver that acts as a major device driver.

Both operating systems require drivers to implement standard I/O routines, which are called dispatch routines in Windows and file operations in Linux. In Linux, a different set of file operations can be provided for each device handle returned to an application. In Windows, dispatch routines are defined once in the *DriverEntry* routine inside a driver object. Since there is one driver object for each loaded driver, it is not advisable to modify the dispatch routines assigned to it when an application requests a handle through an open call. Windows drivers have an *add device* routine that gets called by the PnP manager for PnP aware devices. There is no PnP manager in Linux and such a routine does not exist in Linux.

Dispatch routines in Windows operate on device objects and IRPs. In Linux, file operations operate on a *file* structure. Custom global driver data is stored in device objects in Windows and in the *file* structure in Linux. A device object is created at load time in Windows whereas in Linux a *file* structure is created when an application requests a handle to a driver with a system *open* call. An important implication of this is that in Linux global data per application can be kept in the *file operations* structure. In Windows, global data can only be present in the FDO that the driver manages. Global data per application in Windows has to be kept in a list structure contained within the FDO's custom data structure.

3.3.2. Device Naming

Drivers in Windows are named using driver-defined strings and are found in the \\device namespace. In Linux, drivers are given textual names but applications are not required to know these. Driver identification is performed through the use of a major-minor number pair. Major and minor numbers are in the range 0-255, since a 16 bit number is used to represent the major-minor pair thus allowing a maximum of 65535 devices to be installed in a system.

Devices in Linux are accessible to applications through file system nodes. In most Linux distributions the directory */dev* contains device file system nodes. Each node is created with a driver's major and minor number. Applications obtain a handle to a driver, for performing I/O, through the *open* system call targeted at a device file system node. In Windows, another driver naming method exists, whereby a 128 bit GUID is registered by each driver. Applications access the Windows registry to obtain a textual name in the \\device namespace using the GUID. This textual name is used to obtain a handle for performing I/O with a driver through the *CreateFile* Win32 API call.

3.3.3. User-Kernel Space Data Exchanges

Data exchanges to and from user space are performed similarly by both operating systems, enabling buffered data transfer, performed by the I/O Manager in Windows and by the driver in Linux. Direct I/O to a user space buffer is achieved in both operating systems by locking the user space buffer so that it stays in physical memory. This arises from the fact that drivers cannot always access user space buffers directly, since they will not always be executing in the same process context as the application that owns the user space buffers. The application has its own virtual address space, which is only valid in its own process context. Thus when the driver accesses a virtual address from some application outside of that application's process context, it will be accessing an invalid address.

3.3.4. Driver Installation and Management

Driver installation is through a text file called an INF file in Windows. Once installed, the driver for a device will be automatically loaded by the PnP manager when the device is present in the system. In a Linux system, programs are used to load driver binary images into the kernel. Entries need to be inserted manually into system start up files, so that driver loading programs like *modprobe* are executed with a driver image path or an alias to the driver as a parameter. Driver aliases are defined in the file */etc/modules.conf*, which programs like *modprobe* inspect before loading a driver. An example of an alias entry in *modules.conf* would be "alias sounddriver testdriver", which aliases the name *sounddriver* to the driver binary image testdriver. Executing *modprobe* with *sounddriver* as a parameter would make *modprobe* load *testdriver*. In this way, users can use a standard and simpler name such as *sounddriver* to load a sound driver without having to know what the name of a specific driver for a sound card is. Driver status information is available in Windows through the device manager applet, or directly reading the data from the system registry. In Linux, driver information is available through entries in the proc file system. The file */proc/module* for example contains a list of loaded modules.

4. A Kernel Buffer Driver

This section presents the implementation of a simple driver that performs I/O to blocks of kernel memory, a hypothetical virtual device. Discussion of the various components needed to make the driver work on both Windows and Linux is presented, so that the similarities and differences in the driver components of each operating system can be highlighted. The virtual device, managed by the driver, is shown in figure 4.0. It consists of a number of blocks of kernel memory. Applications can perform I/O operations on the virtual device. The driver will be able to select the memory bank it wants to access and the offset within a memory bank's block.

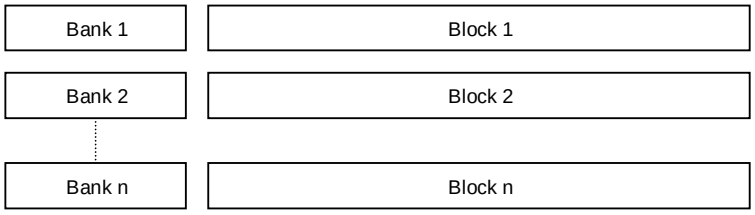

Figure 4.0 A simple virtual device

4.1. Required Driver Components.

Both the Windows and Linux drivers will implement the read, write and IOCTL driver routines. Required driver routines for each operating system are shown in figure 4.1. Naming of the driver routines is flexible. In figure 4.1 the routines for the different operating systems could have been given the same names, instead the conventional platform names have been utilised.

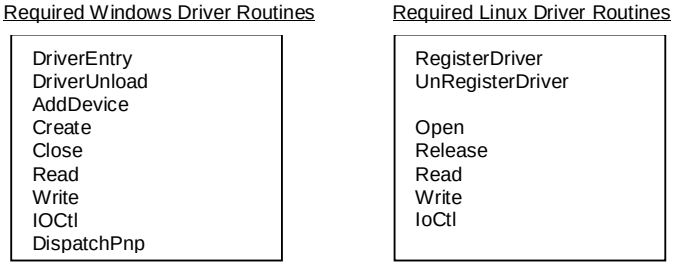

Figure 4.1 The Windows and Linux basic driver routines

4.1.1. Driver Load and Unload Routines

In Windows, the step performed in the driver load routine, *DriverEntry*, is the setting up of I/O dispatch routines as shown in figure 4.1.1a.

```
driverObject->DriverUnload = DriverUnload;
driverObject->DriverExtension->AddDevice = AddDevice;
driverObject->MajorFunction[IRP_MJ_READ] = Read;
driverObject->MajorFunction[IRP_MJ_WRITE] = Write;
driverObject->MajorFunction[IRP_MJ_CREATE] = Create;
driverObject->MajorFunction[IRP_MJ_CLOSE] = Close;
```

```
driverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnp;
driverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = Ioctl;
```

Figure 4.1.1a Initialisation of a driver Object in the driver entry routine

In Linux, the step performed in the driver load routine, *RegisterDriver*, is the registration of a driver major number as shown in figure 4.1.1b. The tagged file operation initialisation, specific only to the GCC compiler, is shown in figure 4.1.1b in the declaration of the structure *fops*, which is not valid ANSI C syntax. The compiler will initialise the various fields of the *file_operations* structure (fops) with the supplied driver implemented routine names i.e. *open* is a field name in the structure and *Open* is a routine implemented by the driver and the compiler will assign a function pointer for *Open* to *open.*

```
struct file_operations fops
{       open: Open,
        release: Release,
        read: Read,
        write: Write,
        ioctl: IoCtl,
}
result = register_chrdev(major_number,"testdriver",&fops);
if(result < 0) PRINT_ERROR("driver didn't load successfully");
```

Figure 4.1.1b Registration of a driver major number in Linux

In the driver unload routine for the Linux driver the registered driver must be unregistered as shown in figure 4.1.1c.

```
unregister_chrdev(major_number,"testdriver");
```

Figure 4.1.1c Driver major number deregistration in Linux

4.1.2. Global Driver Structure

A structure must be defined for storing global driver data that will be operated on by the driver in its routines. For the memory device the same structure will be used for both Windows and Linux versions of the driver. It is defined as shown in figure 4.1.2.

```
#define MAX_BANK_SIZE 4
typedef char byte_t;
#define BLOCK_SIZE 1024;

typedef struct _DEVICE_EXTENSION
{       byte_t * memoryBank[MAX_BANK_SIZE];
        int currentBank;
        int offsets[MAX_BANK_SIZE];
```

```
}DEVICE_EXTENSTION, *PDEVICE_EXTENSION;
```

Figure 4.1.2 Structure used to store global data for generic driver

memoryBank is an array of 4 blocks of memory where the size of a block is 1K. *currentBank* indicates the currently selected block of memory and *offsets* records offsets within each of the blocks of memory.

4.1.3. Add Device Routine

The *add device* routine is only specific to Windows. Linux does not have an add device routine. All initialisation must be done in the driver load routine instead. The operations performed in the Windows *addDevice* routine are shown in figure 4.1.3. A device object is created with a call to the I/O manager routine *IoCreateDevice*.

An interface that applications will use to gain access to a handle for the driver is then created with the I/O manager routine call *IoRegisterDeviceInterface*. One of the arguments to this routine is a GUID manually generated with the system *guidgen* application. The different ways data could be exchanged by drivers and applications to and from kernel space in Windows were presented in section 3.1.2.6. A driver indicates what form of data exchange method it wants to use by setting the *flags* field of its device object (see sections 3.1 and 3.1.2 for a discussion of device objects). In this example the *flags* field is set so that the driver performs buffered I/O. Space for each of the blocks of memory to be used by the memory device is then allocated with one of the kernel memory allocation routines called *ExAllocatePool*. The memory is allocated from the kernel's non-paged pool of memory, thus the device's memory will always be in physical memory.

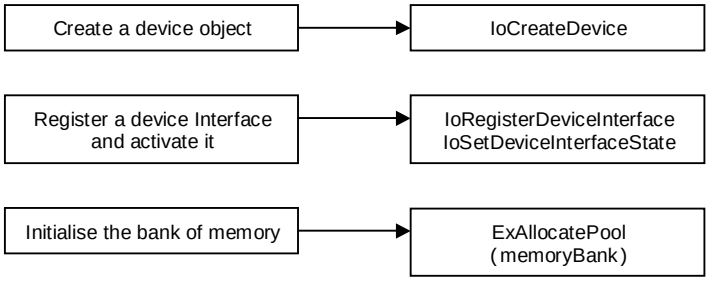

Figure 4.1.3 Operations performed in the Windows driver's add device routine

4.1.4. Open and Close Routines

Most of the initialisation required for the Windows driver has already been done in the *add device* routine so there is nothing to be done in the *open* routine. The *Open* routine in Linux performs the operations shown in figure 4.1.4a. Firstly, memory for storing global driver data is allocated and the file structure's *private_data* field set to point to it. Blocks of memory for the memory device are then allocated in exactly the same way as for Windows. Only the name of the memory allocation routine differs, *ExAllocatePool* in Windows and *kmalloc* in Linux.

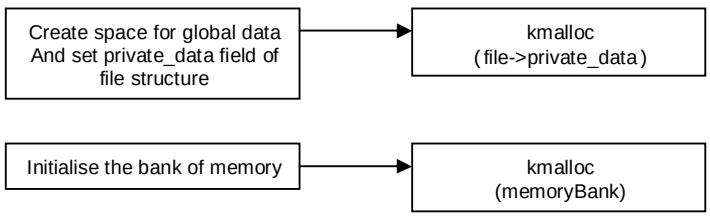

Figure 4.1.4a Operations performed in Linux's generic driver open routine

In Linux's close routine, the space allocated for global driver data as well as space allocated for the memory device are freed as shown in figure 4.1.4b. In Windows, the freeing up of allocated memory is done in response to the PnP *remove device* message, which is discussed later in this section.

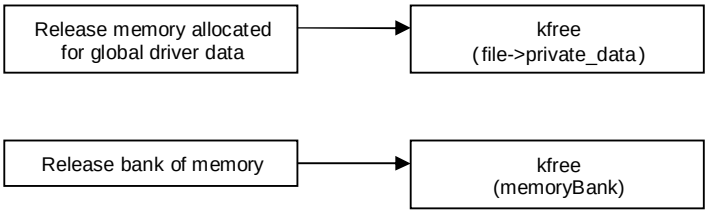

Figure 4.1.4b Operations performed in Linux's generic driver close routine

4.1.5. Read and Write Routines

The *read* and *write* routines will transfer data to and from the currently selected kernel memory bank. In Windows, the read routine is performed as shown in figure 4.1.5a. The length of the data to be read is obtained from an IRP's I/O stack location (see section 3.1.3 for what an I/O stack location is), in the field named *Parameters.Read.Length*. Data of the requested size is read from the currently selected bank of memory (applications perform memory bank selection through the driver's IOCTL routine discussed later) using the kernel runtime routine called *RtlMoveMemory*. *RtlMoveMemory* moves the data from the memory device's space to the buffer allocated for buffered I/O by the I/O manager i.e. the *AssociatedIrp.SystemBuffer* field of the IRP. The IRP is then completed, which informs the I/O manager that this driver has finished processing the IRP and that the I/O manager should return the IRP to the originator of the IRP.

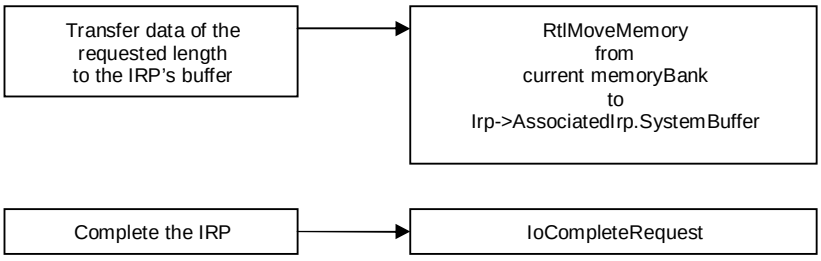

Figure 4.1.5a Performing a read operation in the Windows driver

The write routine performs the opposite of the above memory move operation as shown in figure 4.1.5b.

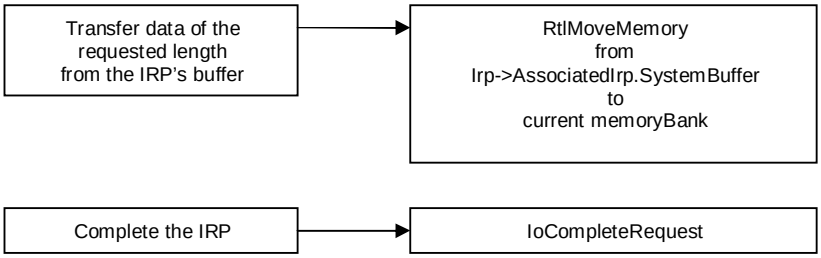

Figure 4.1.5b Performing a write operation in the Windows driver

In Linux, the read routine appears as shown in figure 4.1.5c. A reference to the global driver data is obtained from the *private_data* member of the file structure. From the global data, a reference to the *memoryBank* is obtained. Data is then transferred from this memory bank to user space using the user space access kernel routine called *copy_to_user*.

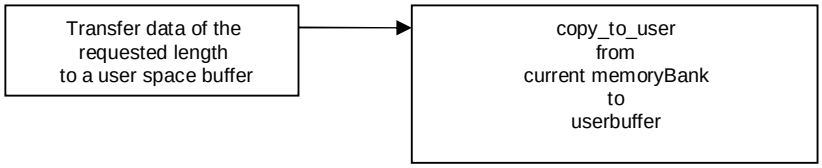

Figure 4.1.5c Performing a read operation in the Linux driver

The write routine performs the same operations as above, except this time data is transferred from user to kernel space as shown in figure 4.1.5d.

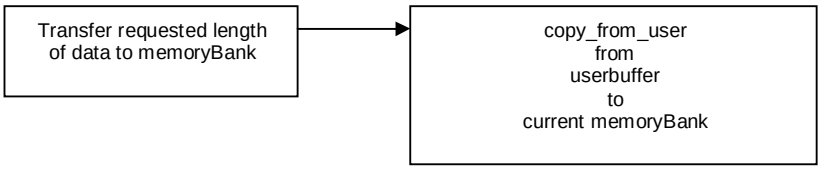

Figure 4.1.5d Performing a write operation in the Linux driver

4.1.6. Device Control Routines

Device control routines are used to set various states of a device. Applications make IOCTL calls to drivers by using the win32 routine *DeviceIoControl*. This routine requires an IOCTL code defined by a driver. An IOCTL code tell a driver what control operation an application is wanting to perform. In this example, the driver implemented IOCTL routine is used to select the current bank number. Driver-specific IOCTL codes must be defined prior to use. IOCTL codes in Windows are defined as shown in figure 4.1.6a. The macro CTL_CODE is used to define a particular device IOCTL code [Oney, 99]. The first argument to the CTL_CODE macro indicates the device ID. Device ID numbers are in the range 0-65535. Codes 0-32767 are reserved for the operating system. Codes 32768-65535 are available for custom use. The code chosen should be the same as the

device code specified during the *IoCreateDevice* call in the driver's *addDevice* routine (see section 4.1.3 for what happens in the *addDevice* routine).

The second argument indicates the function code and is 12 bits long. Codes 0 to 2047 are reserved by Microsoft so a function code greater than 2047 and less than 2^12 is used. It use used to define what control code is being defined. I.e. it distinguishes the two IOCTL codes shown in figure 4.1.6a. The third argument specifies the method used to pass parameters from user space to kernel space, and the fourth argument indicates the access rights that applications have to the device.

```
#define MEMORY_DEVICE 61000


#define IOCTL_SELECT_BANK \
    CTL_CODE(MEMORY_DEVICE, 3000, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_GET_VERSION_STRING \
    CTL_CODE(MEMORY_DEVICE, 3001, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

Figure 4.1.6a IOCTL code definition in Windows

In Linux, applications make IOCTL calls to drivers by using the system routine *ioctl*. IOCTL codes are specified in the file *Documentation/ioctl-numbers.txt*, which can be found in the Linux kernel source tree. Experimental drivers select an unused code, currently 0xFF and above. The IOCTL codes for this driver are defined as shown in figure 4.1.6b. Macro _IOWR indicates that data will be transferred to and from kernel space. Other macros available are _IO which indicates no parameters, _IOW which indicates that data will be passed from user space to kernel space only and lastly _IOR which indicates that data will be passed from kernel space to user space only. The above macros require the size of the IOCTL parameter that will be exchanged between kernel and user space. Rubini *et al* [Rubini *et al*, 01] suggest that for the driver to be portable, this size should be set to 255 (8bits) although current architecture dependent data ranges from 8-14 bits. The second argument is similar to the Windows function number. It is eight bits wide, i.e. ranges from 0-255.

```
#define IOCTL_PARAM_SIZE 255
#define MEMORY_DEVICE 0xFF


#define IOCTL_SELECT_BANK \
    _IOWR(MEMORY_DEVICE, 1, IOCTL_PARAM_SIZE)
#define IOCTL_GET_VERSION_STRING \
    _IOWR(MEMORY_DEVICE, 2, IOCTL_PARAM_SIZE)
```

Figure 4.1.6b IOCTL code definition in Windows

Once the IOCTL codes have been selected, the IOCTL routines can be defined. In Windows, the IOCTL routine is defined as shown in figure 4.1.6c. Two IOCTL codes are handled. The first, IOCTL_SELECT_BANK, sets the current bank number. The second, IOCTL_GET_VERSION_STRING, returns a driver version string. Data returned to callers of the IOCTL routine is handled in the same way as for the read and write requests.

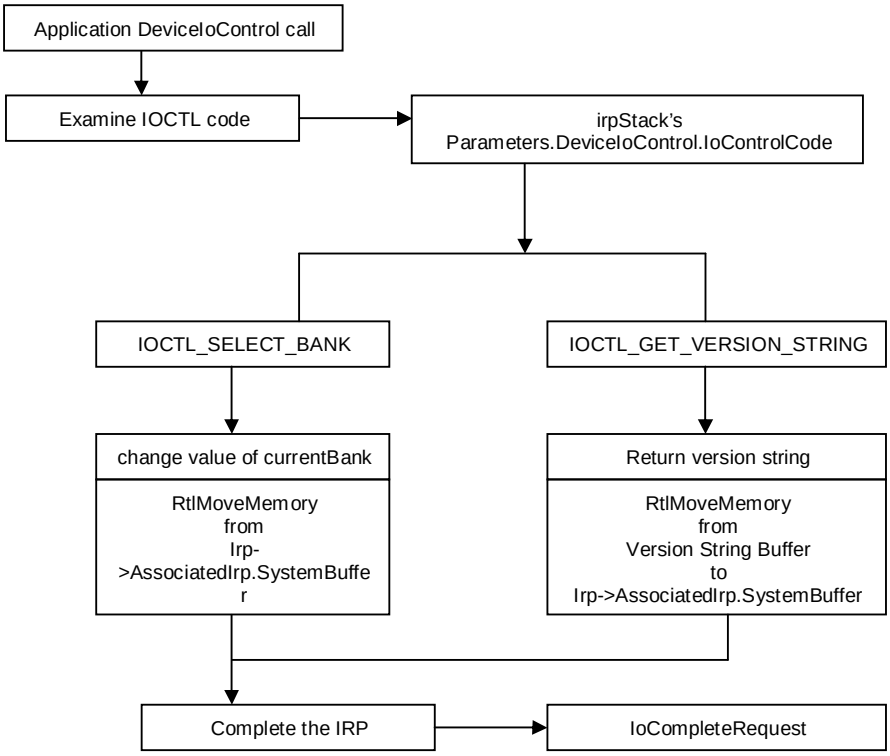

Figure 4.1.6c IOCTL routine definition in Windows

The Linux IOCTL routine definition is shown in figure 4.1.6d. The IOCTL codes handled are the same as for Windows. The only difference is that of syntax specific to each kernel. Data handling is performed in the same way as for reads and writes.

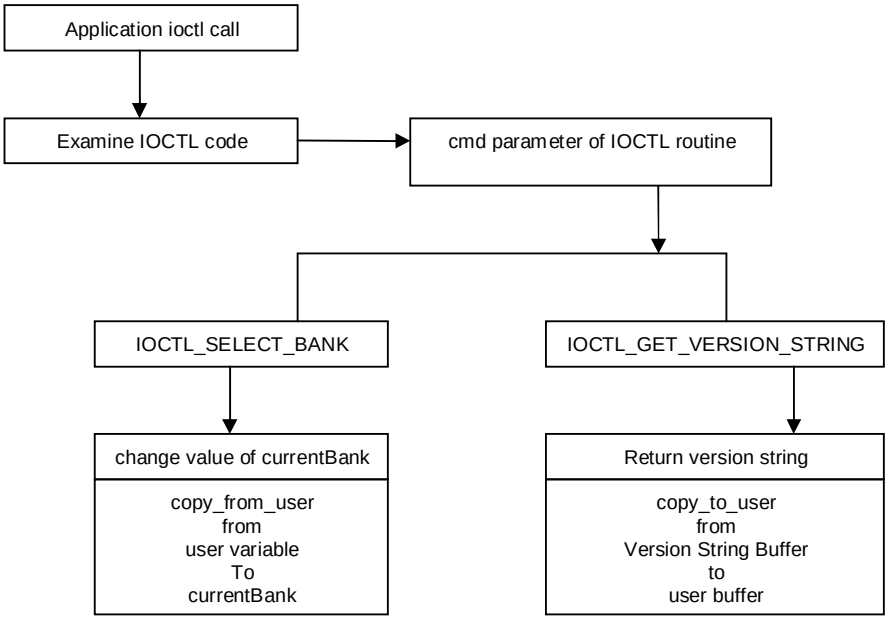

Figure 4.1.6d IOCTL routine definition in Linux

4.1.7. PnP Message Handling Routines

In Windows, PnP messages are dispatched to the driver at appropriate times. E.g. when a device in inserted into the system or removed from the system. These messages are handled by a driver that implements a PnP dispatch routine. In Linux, the kernel does not send PnP messages to the driver thus a PnP routine does not exist in the Linux driver. The Windows PnP message handler is shown in figure 4.1.7.

Only one of the PnP messages is handled for by the memory device driver in this example. The *remove device* message is sent when the driver is unloaded from the system. At this time, the driver's interface is disabled with a call to the I/O manager routine *IoSetDeviceInterface* and the driver's functional device object is deleted as well as the generic driver's blocks of memory.

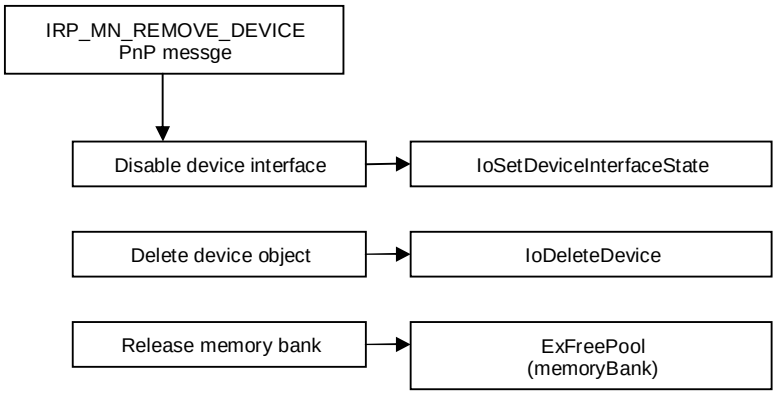

Figure 4.1.7 PnP Message handler routine

5. Driver Development Environments

Developing drivers for the two operating systems discussed so far, namely Microsoft's Windows and Linux, requires the use of software development tools specific to each platform. The operating system kernel on both Windows and Linux is constructed using the C Programming Language. It follows that drivers for both operating systems are created using the C programming language. Windows supports driver construction using the object oriented programming language C++, whereas Linux does not.

5.1. The Windows Driver Development Environment

Microsoft Windows is a proprietary, commercial operating system i.e. it must be purchased for a fee. A number of commercially available driver development environments aimed at Windows exist. One such example is the NuMega DriverStudio™ Suit [Compuware, 01] which comes with class libraries and driver construction wizards that aid in the development of drivers, as well as an integrated debugger that allows debugging of driver code.

5.1.1. The Windows Device Driver Development Kit

The standard route for driver development on Windows is to obtain a Device Driver Kit (DDK) from Microsoft and use its facilities to build drivers. The latest version of the DDK is available to Microsoft Software Development Network (MSDN) subscribers. The DDK contains programs required to build drivers. The DDK installation program will install batch files that set up a shell window to enable building of drivers for each of Microsoft's operating systems. The DDK release used in this investigation of the Windows driver architecture, Windows DDK 3590, has build environments for Windows ME, 2000, XP and .NET drivers. There are two build environments for each platform. The first is called a checked build environment i.e. debugging symbols are added to the driver code. The second is called a free build environment i.e. drivers built in this environment are not built with debugging symbols. The latter environment is where production drivers are built.

5.1.2. Windows Driver Makefiles

Once a DDK shell window is active the simple command *build* will build a driver. A *Makefile* is used to define driver source files from which a driver is to be built. The entries for the *Makefile* are specified in a file called *sources* which resides in the directory where the *build* command is issued. Figure 5.1.2 shows the format of a *Makefile* used to build a simple driver. The environment variable TARGETNAME specifies the driver output filename. For this example the final driver will be called mydriver.sys as drivers in Windows are assigned a .sys extension. TARGETPATH specifies where the object code for the driver will be produced. There is a file called *_objects.mac* in the directory called *obj* where additional paths for object file dump directories are defined. On Windows 2000, the default object file dump directories are *objchk_w2k* for checked builds and *objfre_w2k* for free builds. INCLUDES specifies the path for include files necessary to build drivers and finally SOURCES specifies the driver source file from which the driver will be built.

```
TARGETNAME=mydriver
TARGETPATH=obj
TARGETTYPE=DRIVER
DRIVERTYPE=WDM

INCLUDES=$(BASEDIR)\inc;
SOURCES= mydriver.c
```

Figure 5.1.2 A Makefile used for building a WDM driver with the Windows DDK

5.1.3. Windows DDK Documentation and Tools

The Windows DDK contains well organised API documentation as well as example driver source files from which newcomers to driver writing can learn to create drivers. The DDK also contains utility programs that aid in the driver development process. One of the utility programs is the device tree application that lists all

currently loaded drivers listed in the \\device name space (see section 3.1.2.4 which discusses device name spaces) in a hierarchical manner, showing any existing driver stacks as well as driver implemented routines and driver object memory addresses. Other programs that are provided with the Windows DDK are an INF file generation application called *geninf* used to generate INF files for driver installation and a PnP driver test application used to test if drivers are PnP compliant.

5.2. The Linux Driver Development Environment

The driver development environment on Linux is different from that on Windows. There is no counterpart to the Windows DDK on Linux i.e. there is no such thing as a Linux Device Driver Kit supplied by the kernel's creators. Instead the kernel's creators make all the kernel source code available to everyone. The kernel header files are all that are required for creating drivers. Drivers are built using the GNU C compiler, GCC, which is also used to build applications. Similarly to Windows, a *Makefile* is used to specify how a driver is to be built.

5.2.1. Linux Driver Makefiles

Once a *Makefile* is defined, the simple command *make* is used to build the driver. Figure 5.2.1 shows an example *Makefile* for building a driver, called *mydriver*, in Linux with a source file name *mydriver.c*. The first entry, KERNELDIR, defines an environment variable that specifies where the kernel header files are located. The next line includes the current kernel configuration. Before a kernel and its drivers are built, externally definable kernel variables are specified in a file called .config which is stored in the kernel source tree's root directory. These are included so that the kernel's header files can make use of them. CFLAGS is used to set additional flags to the compiler (GCC). '-O' turns on code optimisation, '-Wall' prints out all code warnings. The 'all' section is the default section examined when the 'make' command is executed. It provides a target called *mydriver*, which depends on the object file called *mydriver.o*, which is built by using GCC. The environment variable LD specifies the GNU linker to be used to build the final driver module. The option '-r' specifies that output should be relocatable i.e. memory locations within it will be offsets to some base address not known at compile time. '$^' is an alias for *mydriver.o* and '$@' is an alias for *mydriver* i.e. it requests the linker to produce relocatable code from the object file mydriver.o and produce an output called mydriver.

```
KERNELDIR=/usr/src/linux
include $(KERNELDIR)/.config
CFLAGS=-D__KERNEL__ -DMODULE –I $(KERNELDIR)/include –O –Wall

all: mydriver

mydriver: mydriver.o
        $(LD) –r $^ -o $@
```

Figure 5.2.1 Makefile used to build a driver in Linux

Kernel module management programs such as *insmod* and *lsmod* can then be used to load the driver into the kernel and to observe currently loaded modules, respectively.

5.2.2. Linux Driver Development Documentation

There is some documentation on the various parts of the Linux kernel in the directory called "Documentation" found under the kernel source tree, but is not as complete and descriptive as the Windows DDK documentation. The Linux driver book written by Rubini *et al* [Rubini *et al*, 01] is a better source of information for device driver writers. There are no example drivers that come with the Linux kernel, but code for existing production drivers is available, which can be used as a basis for starting a new device driver. However, this does not provide a good introduction for novice device driver developers.

5.3. Debugging drivers

The creation of just about every piece of software requires it to be debugged sometime during the time of its development as there are always obscure bugs that are not discovered by examining source code manually. This is especially true for device drivers. At worst, bugs in applications might cause the application's process to become unstable. Serious bugs in drivers will cause the entire system to become unstable.

Debugging applications is a straightforward process. A break statement is set at a place of interest in source code using a debugger's debugging facilities. This is usually debugger specific. In Windows, using Microsoft's Visual Studio debugger, setting break points is as simple as clicking a line in the source code editor. The same applies to DDD (a GUI debugger found on Linux that uses the popular command line debugger GDB). When a program is executed in debug mode and a break point is reached, the execution of that program is paused and the program can be single stepped i.e. instructions from it executed one at a time and their effects observed. A debugger will usually allow the values of variables and variable memory addresses in a running program to be observed. The debugging facilities discussed thus far are also available for debugging device drivers, to a certain extent, on each operating system.

5.3.1. Debugging Drivers on Windows

Debugging drivers under Windows can be performed using a number of different methods. The simplest of these is to use the *DbgPrint* debugging routine which allows printing of messages to the Windows debugger buffer. If a Windows debugger such as WinDbg is running, then the messages can be observed from there, otherwise a special application that can retrieve messages from the debugger's buffer has to be used. One such application is *DebugView*, a free application provided by the SysInternals Corporation [Russinovich, 01]. The *DbgBreakPoint* routine sets a break point in a program. When executed, the system pauses and driver code execution is passed to the system debugger. The *Assert* macro transfers driver execution to the system debugger based on the value of a test condition.

The Microsoft kernel debugger, *WinDbg*, requires two PCs for operation. The first PC is where the driver code is developed and tested. The second PC is connected to the driver development PC via a serial port. A developer can interact with the debugger running on the first PC through a serial console from the second PC. The NuMega DriverStudio ™ [Compuware, 01] provides a debugger that allows drivers to be debugged from a single PC, which can be the driver development machine, and acts like an application debugger. It provides a console Window from which command line instructions can be issued to control it.

5.3.2. Debugging Drivers on Linux

In the same way as Windows, debugging of drivers in Linux can be performed by using debug routines provided by the kernel such as *printk*, which is the equivalent of the Windows *DbgPrint* routine. It behaves in the same way as the C standard I/O routine *printf* except that it takes an additional argument that specifies where the message will be printed to. A kernel debugger is also available as a patch that can be applied to the kernel sources. The patch for the built-in Linux kernel debugger (kdb) can be obtained from the KDB project page [KDB, 02]. It allows the same operations as a standard debugger i.e. setting break points, single stepping driver code, and examining driver memory.

6. Conclusion

Windows and Linux are two of the most popular operating systems in use today. Windows has the biggest market share, and Linux is gaining in popularity. Every new device that gets released to the public by a hardware manufacturer will almost certainly come equipped with a device driver that will make it operate on the Windows operating system. The two operating systems' driver architectures are different in many ways but have some similarities.

6.1. Device Driver Architectures

Comparison of the driver architectures used by the two operating systems has shown that the Windows operating system has a more developed architecture than Linux. This does not mean that the Windows architecture offers better functionality than that of Linux, rather it has a more formally defined driver model, which driver developers are encouraged to follow. Although driver writers can ignore the Windows driver model and construct their own monolithic drivers, it was found that most driver writers did not take this route. No formally defined driver model exists for the Linux operating system. Linux driver writers produce drivers

based on their own personal designs. Unless two groups of driver developers cooperate and produce drivers that work together, drivers from different developers cannot operate together under the Linux operating system.

Under Windows, drivers from two or more sets of developers can be made to work together, provided the developers have followed the Windows Driver Model (WDM) to construct their drivers. The Windows driver architecture supports PnP (Plug and Play) and Power management, by dispatching messages at appropriate times to device drivers which have been implemented to handle these messages. No such facility is offered by the current Linux driver architecture.

6.2. Designing device drivers

When designing device drivers the facilities offered by an operating system should be evaluated. The Windows and Linux operating systems are both modern operating systems. They make available implementations for data structures such as stacks, queues and spin locks, as well as HAL (Hardware Abstraction Layer) routines for performing hardware independent operations. This enables device drivers to operate on different architectures such as IA64 (Intel's 64 bit platform) and SPARC.

Driver functionality on both operating systems can be broken up into modules, which can be stacked together and that communicate using a standardised data structure. Under Windows this standardised data structure is the IRP (I/O Request Packet) and under Linux it can be any driver-defined structure, since no standardised structure exists on that operating system.

6.3. Implementing Device Drivers

Device drivers on both operating systems are made up of a set of routines that each operating system expects all drivers to implement. They include routines for standard I/O such as reading from and writing to a device, and for sending device I/O control commands to a device. Every driver for each operating system implements a routine that will be executed when the driver is loaded for the first time, and a routine that gets executed when a driver is unloaded. It is possible to construct a driver for each operating system that uses identical naming for the various driver routines, although the usual approach is to use conventional names for each operating system. The device driver naming scheme on Windows (using device interfaces) is a lot more flexible than the current device driver naming scheme used by Linux. Driver naming clashes are more likely to occur in Linux as compared to Windows, which uses a GUID (Globally Unique Identifier) for each device.

6.4. Driver Development Environments

The Windows operating system provides a DDK (Device Driver Developer's Kit), which contains relevant documentation and development tools that help decrease the time required for learning to create new drivers. The Linux operating system does not provide a DDK, therefore initially some time will have to be spent by device driver developers to gather other sources to aid in the driver development process. Once time has been spent in getting familiar with the two driver development environments, developers will find it easier to create Linux drivers than Windows drivers, because all of the Linux kernel source code is available to them. This enables driver developers to trace problems in their drivers by having a closer look at the kernel code that their drivers rely on. Under Windows, only binary debug builds of the operating system's components are available. These contain debug symbols such as function names and variable names and are not as useful as having the operating system's source code.

6.5. Concluding Remarks

Drivers should be designed so that use of them requires very little interaction from end users, and all of a driver's functionality is made available to applications. The former is one of the strong points of Windows, which fully supports PnP. Linux is an open source project, which is still actively being improved. It is expected that in the future Linux's driver architecture will become as formalised as Windows', which for example has a driver model such as the WDM. Growth of hardware vendor support for Linux is also expected as more and more individuals and organisations adopt it.

Acknowledgements

This research was made possible through the Andrew Mellon Foundation scholarship at Rhodes University, Grahamstown, South Africa.

References

[Beck *et al*, 98] Beck, Bohme, Dziadzka, Kunitz, Magnus, Verworner, Linux Kernel Internals, Addison Wesley, 1998.

[Cant C, 99] Cant C, Writing Windows WDM Device Drivers, CMP Books, 1999.

[Compuware, 01] Compuware, NuMega DriverStudio Version 2.5, http://www.compuware.com, 2001.

[Compuware, 01] Compuware, Using Driver Works, Version 2.5, Compuware, 2001.

[Deitel, 90] Deitel HM, Operating Systems, 2nd Edition, Addison Wesley, 1990.

[Davis, 83] Davis W, Operating Systems, 2nd Edition, Addison Wesley, 1983.

[Flynn *et al*, 91] Flynn IM, McHoes AM, Understanding Operating Systems, Brooks/Cole, 1991.

[Katzan, 73] Katzan H, Operating Systems: A Pragmatic Approach, Reinhold, 1973.

[KDB, 02] KDB, The Linux Built in Kernel Debugger, http://oss.sgi.com/projects/kdb, 2002.

[Laywer, 01] Lawyer D S, Plug and Play HOWTO/Plug-and-Play-HOWTO-1.html, http://www.tldp.org/HOWTO, 2001.

[Linus FAQ, 02] The Rampantly Unofficial Linus Torvalds FAQ, http://www.tuxedo.org/~esr/faqs/linus/index.html, 2002.

[Linux HQ, 02] The Linux Headquarters, http://www.linuxhq.com, 2002.

[Lorin *et al*, 81] Lorin H, Deitel HM, Operating systems, Addison Wesley, 1981.

[Microsoft DDK, 02] Microsoft ,DDK- Kernel Mode Driver Architecture, Microsoft, 2002.

[Microsoft WDM, 02] Microsoft, Introduction to the Windows Driver Model, http://www.microsoft.com/hwdev/driver/wdm, 2002.

[Oney, 99] Oney W, Programming the Microsoft Windows Driver Model, Microsoft, 1999.

[Open Group, 97] Open Group, Universal Unique Identifier, http://www.opengroup.org/onlinepubs/9629399/apdxa.htm, 1997.

[Redhat,02] Redhat, http://www.redhat.com,2002.

[Rubini *et al*, 01] Rubini A, Corbet J, Linux Device Drivers, 2nd Edition, Oreilly, 2001.

[Russinovich, 98] Russinovich M, Windows NT Architecture, http://www.winnetmag.com/Articles/Index.cfm?ArticleID=2984, 1998.

[Russinovich, 01] Russinovich M, SysInternals, http://www.sysinternals.com, 2001.

[Rusling, 99] Rusling D A, The Linux Kernel, http://www.tldp.org/LDP/tlk/tlk.html, 1999.