

Device and Driver Management and Configuration

Milwaukee School of Engineering
Milwaukee, Wisconsin

Author: Christopher Weedall
Data: February 2, 2003
Submitted to: Dr. Christopher Taylor
Course: Operating Systems (CS-384)

Abstract

Device management is a basis, if not the fundamental building block, of a computer system. It unifies hardware, a basic input / output system, device drivers, and an operating system in a way that makes the operation of a device transparent to the user. Through powerful yet simple communications structures, all of the components of the system can interact to allocate system resources and prevent conflicts that may have otherwise crippled a system. The operating system has powerful new components to coordinate all of the associated hardware and software needed to support device management and additionally Plug and Play. Plug and Play represents an important step toward cooperating hardware and software, which is based entirely off of the specifics of device management. In the future, device management will become smoother and more refined, resulting in easier usage for users, faster communications between the operating system, processor, and devices, and more robust handling of functionality and error handling.

TITLE PAGE	i
ABSTRACT	ii
TABLE OF CONTENTS	iii
INTRODUCTION.....	1
Background	1
Scope.....	1
Purpose.....	2
Thesis.....	2
DEVICE MANAGEMENT OVERVIEW.....	2
Architecture.....	2
Specifications.....	3
Drivers.....	4
Interfaces.....	5
KERNEL and DEVICE STRUCTURE.....	5
Structure / Interface.....	5
Drivers and Kernel.....	5
Input and Output.....	7
System Organization.....	7
DEVICE and DRIVER STRUCTURE.....	9
Abstracting Device Differences.....	9
Driver Types.....	10
Block Drivers.....	10
Character Drivers.....	11
Terminal Drivers.....	12
STREAMS Drivers.....	12
Buffering Strategies.....	13
Recovery from Failures.....	15
Interrupt Service Routines.....	16
Interrupt Request.....	16
Interrupts versus Polling.....	17
Interrupt-Driven I/O.....	17
DEVICE HARDWARE.....	18
Serial Devices.....	18
Parallel Devices.....	19
Bus	19
Controllers.....	19
Direct Memory Access.....	20
Memory-Mapped I/O.....	20
Direct I/O with Polling.....	21
CONCLUSION.....	22
APPENDIX.....	23
Works Cited.....	24

INTRODUCTION

Soon after the first successes of digital computer experiments, computers moved out of the lab and into practical use. The first practical application of experimental digital computers was the generation of artillery tables for the British and American armies. Much of the early research in computers was paid for by the British and American militaries. Business and scientific applications followed. As computer use increased, programmers noticed that they were duplicating the same efforts.

Every programmer was writing his or her own routines for I/O functionality, such as reading input from a magnetic tape or writing output to a printer. It made sense to write a common device driver for each input or output device and then have every programmer share the same device drivers rather than each programmer writing his or her own. Some programmers resisted the use of common device drivers in the belief that they could write “more efficient” or faster or “better” device drivers of their own.

Additionally each programmer was writing his or her own routines for fairly common and repeated functionality, such as mathematics or string functions. Again, it made sense to share the work instead of everyone repeatedly “reinventing the wheel”. These shared functions could be organized into libraries and also inserted into programs as needed. In the spirit of cooperation among early researchers, these library functions were published and distributed for free, an early example of the power of the open source approach to software development.

This report discusses device management and is meant to inform the reader about the different hardware and software components that are needed to allow a user to interact with processes within an operating system by means of external devices. Device management encompasses hardware devices, device drivers, and the operating system, as a complex combination of electrical signals and high level software working together. These topics are addressed by describing the responsibilities and interactions with components within the system. In the report, it is assumed that behavior and components of device management is generic and applies to any operating system, unless otherwise stated.

DEVICE MANAGEMENT OVERVIEW

Architecture

This is where the various different kernel architectures become apparent. There are two main categories:

Microkernel

Device drivers are isolated from the kernel, and are usually in their own user-mode address spaces. They communicate with the main kernel and with each other by means of messaging. Passing a message from, say, the kernel to a driver usually involves a process switch (switching address spaces) and a context switch (switching threads) which incurs a relatively high speed penalty. Microkernel's

main advantage is stability: if one buggy device driver crashes, all that is affected is access to that device (until the driver is restarted).

Monolithic Kernel

Device drivers run as part of the kernel, either compiled in or as run-time loadable modules. Monolithic kernels have the advantage of speed and efficiency: calls to driver functions are simple local calls instead of whole address space switches.

However, because it is running in kernel

The monolithic architecture is more common in modern operating systems because it is usually more efficient. Monolithic kernels can have access to all the kernel's code and data, making it possible to share internal functions and data with device drivers. By contrast, microkernel drivers are fully self-contained applications, which make it a lot harder for them to share common functions. The microkernel separation between the kernel and the rest of the OS components isn't restricted to device drivers.

Specifications

The device manager is the interface between the device drivers and the both the rest of the kernel and user applications. It must isolate device drivers from the kernel so that driver writers can worry about interfacing to the hardware and not about interfacing to the kernel. It must also isolate user applications from the hardware so that applications can work on the majority of devices the user might connect to their system.

In most operating systems, the device manager is the only part of the kernel that programmers really see. It consists of the following:

Asynchronous I/O

Applications will be able to start an I/O operation and continue to run until it terminates. This is instead of blocking I/O, whereby applications are stalled while I/O operations execute. Blocking I/O can be easily implemented as a special case of asynchronous I/O.

Plug and Play

Drivers will be able to be loaded and unloaded as devices are added to and removed from the system. Devices will be detected automatically on system startup, if possible.

Drivers

Drivers are files that tell hardware (modem, scanner, mouse, etc.) how to operate. A driver is at the lowest level for input and output control. In conjunction with interrupt handlers, they transfer information between the main processor and hardware. The driver translates a high-level command, from operating system or process, into a low-level command which is a hardware-specific instruction used by hardware controllers. The controllers interface an I/O device with the entire system. The driver commonly will send a specific bit pattern to an I/O controller which tells the controller which location a certain device will act from and what actions will be taken.

Interfaces

Once the devices installed in the system have been detected, a record of it must be kept somewhere. Depending on the operating system, these records may be hidden, or may be accessible, by a user of the system.

Typically a user application will issue commands to a specific device by means of the device records previously explained. The application will trap the kernel into a system call. The kernel will then request data from the device needed. It is then ready to pass the request onto the driver responsible for that device. The interface between the kernel and the driver is generally a table of function pointers. Different routines can be used for each function, but usually similar tasks (i.e. reading and writing) go to the same routine. The driver can tell the requests apart by the information in the I/O Request Packet (IRP) passed to a function.

KERNEL and DEVICE STRUCTURE

Drivers and Kernel

A device driver is a program that controls a particular type of hardware device that is attached to the computer. There are device drivers for printers, displays, CD-ROM readers, diskette drives, etc. For any operating system, many device drivers are built into the product. However, if a newer device is installed within the system after the operating system was designed and made, there is a chance that no sufficient driver will be available within the operating system. A device driver essentially converts the more

general input/output instructions of the operating system to messages that the device type can understand.

Some operating systems allow for virtual device drivers. In those operating systems, there is a virtual device driver for each main hardware device in the system, including the hard disk drive controller, keyboard, and serial and parallel ports. They are used to maintain the status of a hardware device that has changeable settings. Virtual device drivers handle software interrupts from the system rather than hardware interrupts.

The kernel is the essential center of an operating system, the core that provides basic services for all other parts of the operating system. A synonym is nucleus. A kernel can be contrasted with a shell, the outermost part of an operating system that interacts with user commands. Kernel and shell are terms used more frequently in Unix and some other operating systems than in IBM mainframe systems (i.e. Windows).

Typically, a kernel (or comparable for an operating system) includes an interrupt handler that handles all requests or completed I/O operations that compete for the kernel's services, a scheduler that determines which programs share the kernel's processing time in what order, and a supervisor that actually gives use of the computer to each process when it is scheduled. A kernel may also include a manager of the operating system's address spaces in memory or storage, sharing these among all components and other users of the kernel's services. A kernel's services are requested by other parts of the operating system or by a process, through a specified set of system calls.

Because the code that makes up the kernel is needed continuously, it is usually loaded into computer storage in an area that is protected so that it will not be overlaid with other less frequently used parts of the operating system.

Input and Output

Input and Output (I/O) describes any operation, program, or device that transfers data to or from a computer. Typical I/O devices are printers, hard disks, keyboards, and mice. Some devices are basically input-only devices (keyboards and mice), while others are primarily output-only devices (printers). Then there are devices which provide both input and output of data (hard disks, diskettes, CD burners).

System Organization

A computer system is comprised of four main areas. Processing is the first and encompasses the processor, the operating system, drivers, and other very low level software/hardware. Permanent storage is the second area and contains any kind of media which has information stored on it that the operating system can use for files and processes. Input is the third area and contains hardware such as a mouse or keyboard which are used mostly for inputting data of some form into the operating system. Lastly, there is the output area. This contains hardware devices like a computer monitor which mostly output data in some form to the user or other hardware not contained within the system.

Some devices encompass more than one area, such as a modem, which inputs data into the system and also outputs data from the system to an external source. A graphical representation of a generic system can be seen in Figure (1).

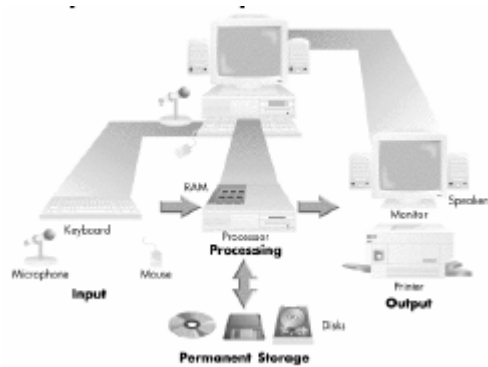


Figure (1)

A more detailed chart describing how the operating system, device drivers, device controllers, and devices is shown in Figure (2).

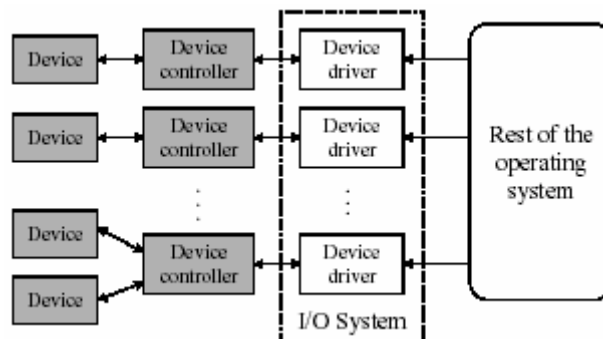


Figure (2)

DEVICE and DRIVER STRUCTURE

Abstracting Device Differences

The path between the operating system and almost all hardware not on the computer's motherboard goes through a special program called a driver. Much of a driver's use is to be the translator between the electrical signals of the hardware subsystems and the high-level programming languages of the operating system and application programs. Drivers take data that the operating system has defined as a file and translate them into streams of bits placed in specific locations on storage devices.

Because there are such wide differences in the hardware controlled through drivers, there are differences in the way that the driver programs function, but most are run when the device is required, and function similar to any other process. The operating system will frequently assign high-priority blocks to drivers so that the hardware resource can be released and readied for use again as quickly as possible.

One reason that drivers are separate from the operating system is so that new functions can be added to the driver, and thus to the hardware subsystems, without requiring the operating system itself to be modified, recompiled and redistributed. Through the development of new hardware device drivers, development often performed or paid for by the manufacturer of the subsystems instead of the publisher of the operating system, input/output capabilities of the overall system can be greatly enhanced.

Managing input and output is mostly a matter of managing queues and buffers, special storage facilities that take a stream of bits from a device, such as a keyboard or a serial port, hold those bits, and release them to the processor at a rate slow enough for the processor to deal with. This function is especially important when a number of processes are running and taking up processor time. The operating system can instruct a buffer to continue taking input from the device, but stop sending data to the processor while the process using the input is suspended. Then, when the process needing input is made active once again, the operating system will command the buffer to send data. This process allows a keyboard or a modem to deal with external users or computers at a high speed even though there are times when the CPU can not use input from those sources.

Driver Types

The types of drivers can be divided into many specific categories. The categories contain things such as monitors, hard drives, printers, etc. However, the specific drivers can be organized into four main types:

Block drivers

Block drivers are for hardware devices, organized as fixed size blocks of data, where blocks can be accessed at random, but it is nearly impossible to read or write to just part of a block at a time. This includes hard and floppy disks, where data is stored as magnetization of the disk surface in fixed size blocks.

Because this type of device is so common and fundamental to the operation of the processor, there is a lot of specialized support for it within most operating systems.

In particular the operating system maintains a cache of recently used blocks and uses this to directly take care of application read and write requests. Then the device driver does not have to deal with these directly. The main thing which the block driver must provide is a "strategy" routine. The operating system calls the strategy routine when it wants to get a new block to put in its cache or when it wants to write a block in its cache back to the physical device (i.e. after it has been written to). So the device driver just has to worry about moving whole blocks of data back and forth at the request of the operating system, while the operating system handles the differences between blocks and the single characters which may be read and written, and also maintains a cache in memory which can greatly improve system performance without complicating the device driver.

Character drivers

Character drivers are a "lowest common denominator" device driver and get relatively little help from the operating system. The character device driver must implement a number of standard functions which the operating system uses to control the device and to which application requests (open, read, write, etc.) are passed on. These standard functions are known as entry points. The device driver's role is to attempt to satisfy calls to those entry points by interacting with the device. The operating system will typically provide some support in terms of, error handling and reporting, and buffering data.

Terminal drivers

Terminal drivers are a specialized kind of character driver. They extend the normal behavior of character drivers with user-terminal oriented facilities such as line editing, tab expansion, high-level flow control. The kernel provides a standard framework for performing these additional functions, to make writing such devices easier. In particular it provides a number of "line discipline" routines which the device driver can use to automate these standard services. To coordinate with these routines the character driver has to provide one additional entry point.

STREAMS drivers

STREAMS drivers are used for high-speed communication of variable-sized data blocks and include the chance of directly realizing layered protocols, such as in computer networking. With STREAMS devices the operating system provides a STREAM head which interfaces to a standard Application Program Interface (API) and has its own standard STREAM interface. These STREAM interfaces can be used to connect together a chain or stack of STREAM modules, each of which can do some operations on the data as it passes by, and then pass it on to the next module in the stack or chain. These modules realize the different layers of a layered protocol such as TCP over IP. Finally, a STREAMS driver converts from the STREAMS interface of the last module (or the head, if there are no modules) to the actual hardware device, such as an Ethernet card.

Buffering Strategies

I/O is the process of transferring data between a program and an external device. The process of optimizing I/O consists primarily of making the best possible use of the slowest part of the path between the program and the device.

The slowest part is usually the physical channel, which is often slower than the CPU or a memory-to-memory data transfer. The time spent in I/O processing overhead can reduce the amount of time that a channel can be used, and therefore reducing the effective transfer rate. The biggest factor in maximizing this channel speed is often the reduction of I/O processing overhead.

A buffer is a temporary storage location for data while the data is being transferred. Small I/O requests can be collected into a buffer, and the overhead of making many relatively expensive system calls can be greatly reduced. A collection buffer of this type can be sized and handled so that the physical I/O requests made to the operating system match the physical characteristics of the device being used.

During the write process, a buffer can be used as a work area where control words can be inserted into the data stream (a process called blocking). The blocked data is then written to the device. During the read process, the same buffer work area can be used to examine and remove these control words before passing the data on to the user (a process called deblocking).

When data access is random, the same data may be requested many times. A cache is a buffer that keeps old requests in the buffer in case these requests are needed again. A cache that is sufficiently large and/or efficient can avoid a large part of the physical I/O by having the data ready in a buffer. When the data is often found in the cache buffer, it is referred to as having a high hit rate. For example, if the entire file fits in the cache and the file is present in the cache, no more physical requests are required to perform the I/O. In this case, the hit rate will be 100%.

Running disks and the processor in parallel often improves performance. Therefore, it is useful to keep the processor busy while data is being moved. To do this when writing, data can be transferred to the buffer at memory-to-memory copy speed and an asynchronous I/O request can be made. The control is then immediately returned to the program, which continues to execute as though the I/O were complete (a process called write-behind). A similar process can be used while reading. In this process, data is read into a buffer before the actual request is issued for it. When it is needed, it is already in the buffer and can be transferred to the user at very high speed. This is another form or use of a cache.

Unbuffered I/O

The simplest form of buffering is none at all. This unbuffered I/O is known as raw I/O. For large, well-formed requests, buffering is not necessary. It can add unnecessary overhead and delay.

Library buffering

The term library buffering refers to a buffer that the I/O library associates with a file. When a file is opened, the I/O library checks the access, form, and any attributes declared on the assign command to determine the type of processing that should be used on the file. Buffers are usually an vital part of the processing.

System cache

The operating system or kernel uses a set of buffers in kernel memory for I/O operations. These are collectively called the system cache. The I/O library uses system calls to move data between the user memory space and the system buffer. The system cache ensures that the actual I/O to the logical device is well formed, and it tries to remember recent data in order to reduce physical I/O requests. In many cases, though, it is desirable to bypass the system cache and to perform I/O directly between the user's memory and the logical device.

Recovery from Failures

If a device becomes unusable, either because the hardware failed or has been taken down for some reason, the driver should reject all new and outstanding I/O requests, and return an appropriate error code indicating that the device is not usable. At each major entry point, the driver shall check the device state, so that resources are not mistakenly committed to a device that cannot perform its function.

If the driver discovers that hardware has failed in some way, the driver should log information about the failure. This allows system management software and/or a human user to trace events to determine the time and root cause of the failure.

Interrupt Service Routines

A programmer may allow Interrupt Service Routines (ISR's) to handle hardware interrupts. These routines are not independent threads, but more like signals. Any currently executing thread is suspended by an interrupt, and the ISR called. The ISR runs on a separate interrupt stack only if the hardware supports it. Otherwise, the ISR stack frames are pushed onto the stack of the interrupted thread.

Most of the thread manipulation routines can be called within interrupt service routines. If the ISR switches threads, the thread switch will take effect when the ISR returns. This might be useful, for example, in implementing a round-robin scheduler. However, the ISR, although more flexible than in typical OS's, still must not block in any way. To do so would cause deadlock. This limits the synchronization tools that ISR's can use.

Interrupt Request

An interrupt request (IRQ) value is an assigned location where the computer can expect a particular device to interrupt it when the device sends the processor a signal about its operation. For example, when a printer has finished printing, it sends an interrupt signal to the processor. The signal momentarily interrupts the processor so that it can decide what processing to do next. Since multiple signals

to the processor on the same interrupt line might not be understood by the processor, a unique value must be specified for each device and its path to the processor. Prior to Plug-and Play (PnP) devices, users often had to set IRQ values manually when adding a new device to a processor.

Interrupts versus Polling

The simplest way for the processor to know if there is an I/O event is to check using a loop. This is called polling. In polling, the device puts information into a status register or in memory-mapped I/O. The processor can then read it during the loop.

The disadvantage to polling is that it can waste the processor's time. By multiplying the amount of polling per second required for a device by the number of cycles it takes, the number of cycles per second can be acquired. Taking that number and dividing it by the number of cycles the processor has per second (i.e. processor clock speed), the amount of overhead can be acquired. The number is not very large for some devices, such as a mouse, but can become much larger when working with the transfer from or to a disk.

Interrupt-Driven I/O

The overhead in a polling interface lead to the creation of interrupts to notify the processor when an I/O device needs attention from the processor. When the interrupt is detected, control is transferred to the OS which only then polls the

device. There is a 'CauseRegister' which is used to tell the processor which device sent the interrupt. These interrupts relieve the processor of having to wait for the devices. However, the processor still must transfer data from memory to the device.

DEVICE HARDWARE

Serial Devices

Serial means one event at a time. It is opposite of parallel, which means more than one event happening at a time. In data transmission, the techniques of time division and space division are used, where time separates the transmission of individual bits of information sent serially and space (on multiple lines or paths) can be used to have multiple bits sent in parallel.

For computer hardware and data transmission, serial connection, operation, and media usually indicate a simpler, slower operation and parallel indicates a faster operation. This does not always hold true, however, since a serial medium (i.e. fiber optic cable) can be much faster than a slower medium that carries multiple signals in parallel.

Most processors and their programs operate in a serial manner, with the processor reading a process and performing its instructions one after the other. However, there are computers with multiple processors and can perform instructions in parallel.

Parallel Devices

Parallel means more than one event happening at a time. In data transmission, the techniques of time division and space division are used, where time separates the transmission of individual bits of information sent serially and space (in multiple lines or paths) can be used to have multiple bits sent in parallel.

Parallel connection and operation generally indicates faster operation. Again, this does not always hold true, because a serial medium may be faster than a slower medium that carries multiple signals in parallel.

Bus

The mechanism by which most systems connect a processor and its hardware devices is called a bus. A bus is a communication medium that can carry one piece of communication at a time, so the processor must grab the bus to talk to some device, such as memory, an IDE controller, or a sound card, then talk and let go of the bus. Every device connected to the bus has a unique identity value and will recognize signals sent from the processor intended for it.

Controllers

Most hardware devices have what are known as controllers. Controllers are dedicated pieces of hardware inside a system that take care of the details of accessing a specific piece of hardware. Device access involves talking to the controller and telling it what to do with the hardware device.

All controllers have some kind of interface that they present to the processor. The interface often involves registers, or special values that can be read from or written to, so that a device can be used.

Direct Memory Access

Direct Memory Access (DMA) is a capability provided by some computer bus architectures that allows data to be sent directly from an attached device to the memory on the computer's motherboard. The processor is freed from involvement with the data transfer, thus speeding up overall operation.

Usually a specified portion of memory is designated as an area to be used for direct memory access. In the ISA bus standard, up to 16 megabytes of memory can be addressed for DMA. Peripheral Component Interconnect (PCI) accomplishes DMA by using a bus master (with the processor assigning I/O control to the PCI controller).

Memory-Mapped I/O

Memory-mapped I/O is a setup where the on-board memory of a device is mapped into address space of the processor. Data to be written to the device is copied by the driver to the device memory, and data read in by the device is available in the shared memory for copying back into the system memory.

Memory-mapped I/O is frequently used by network and video devices. Many adapters offer a combination of programmed I/O and memory-mapped modes, where the data buffers are mapped into the processor's memory space and the internal device registers that provide status are accessed through the I/O space.

The adapter's memory is mapped into system address space through the PCI BIOS, a software setup program, or by setting jumpers on the device. Before the kernel can access the adapter's memory, it must map the adapter's entire physical address range into the kernel's virtual address space.

Direct I/O

Direct I/O is for a file system based files which bypasses the operating system's file system buffer cache. Direct I/O saves memory and boosts the performance of database applications that cache their own data independently. It also avoids performance problems associated with using database block sizes that do not match the file system buffer size exactly. Whether direct I/O is possible, and the way in which it can be enabled, are both dependent on the operating system and type of file system in use.

Direct I/O can sometimes be enabled for an entire file system, using a file system mount option. Application programs may also be able to request direct I/O for a particular file by setting a flag when opening the file.

Most operating systems support numerous types of file systems. Some aspects of file system operation are independent of file system type and are implemented in a common

part of the operating system kernel. However, many other aspects of file system operation, including direct I/O availability, are dependent on the type of file system in use. For example, some file systems cannot support direct I/O because their file system structure is such that file system blocks do not align with device block boundaries and so I/O requests cannot be mapped directly from file system blocks to device blocks.

Conclusion

Device Management involves a complex system of pieces comprised of the operating system, computer processor, hardware busses and controllers, and low-level software drivers. Although there are many components occupied with device management, the final result is an architecture which allows new or undiscovered methods to be used which can speed up the communication between a device and the user of a computer. Additionally, it permits other aspects of a system to be upgraded and improved or changed without needing to redesign the entire system.

The technology industry has made sharp standards, which are constantly updated and raised, that help advance a user's experience with computing easier and more powerful. Through communication and dedication, device management will become easier, more powerful, and faster, allowing for technology as a whole to have virtually no limits in its capabilities.

Appendix

Works Cited

- Device Management*. ADDRESS: <http://www.themoebius.org.uk/tutes/device.html>.
[Accessed 27 December 2002].
- Englander, Irv. 1996. *The Architecture of Computer Hardware Systems Software: An Information Technology Approach* (New York, John Wiley & Sons, 1996).
- Gagne, Peter Galvin, Avi Silberschatz. 2002. *Operating System Concepts Sixth Edition* (New York: John Wiley & Sons, Inc.).
- Larsen, Tore. "I/O Devices and Drivers." [Internet, PDF] Available:
<http://www.cs.uit.no/studier/kurs/d241/info/2002h/notes/tore/lectures/io-shortened.pdf>. [Accessed 16 January 2003].
- Schnier, Mitchell. 1996. Dictionary of PC Hardware and Data Communications Terms.
Cambridge: O.Reilly.
- Sodan, Dr. Angela C. Fall 2002 "Operating System Fundamentals."
Available:
<http://www.cs.uwindsor.ca/users/a/acsodan/courses/330f2002/lecture4.pdf>,
<http://www.cs.uwindsor.ca/users/a/acsodan/courses/330-f2002/lecture5.pdf>.
- Tanenbaum, Andrew S. 2001. *Modern Operating Systems* (New Jersey: Prentice Hall).
- Tanenbaum, Andrew S. 1997. *Operating Systems: Design and Implementation, Second Edition* (Prentice Hall).
- Wyeth, Peta. "Computer Basics." [Internet, PDF] Available:
<http://www.itee.uq.edu.au/~ienv1201/notes/Lecture10/Computer%20Basics.pdf>.
[Accessed 24 January 2003].