Introduction

This textbook is designed to give you an overview of what an operating system is, and how a modern operating system works. There are many different examples of operating systems available for computers. The most popular operating systems belong to the Microsoft Windows family (such as Windows 98, XP and Vista). Other examples are Unix and Linux, Mac OS X, and specialized operating systems for handheld devices like mobile phones.

An operating system is the software that controls (or operates) all of the parts of your computer. It manages all of your resources, and lets you interface with the computer. This textbook takes a look at the main problems that an operating system must be able to overcome, and the main functions that it must be able to perform. We begin by reviewing the architecture (or structure) of the physical parts of a computer, and how they communicate with each other. Then we take a look at fundamental operating system concepts, processes and process management, memory management, controlling input and output devices, and file system management.

Unit 1: Computer Architecture Review

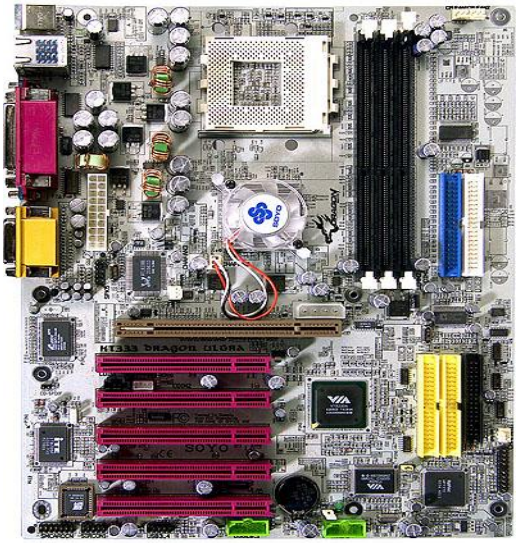
Why Review Computer Architecture?

This textbook focuses on how operating systems work. *Operating systems* (or *system software*) are one of the two main types of software (the other is *application software*). However, we need to know some important things about the hardware inside of a computer in order to understand some of the critical functions of any operating system.

Computer architecture refers to the overall design of the physical parts of the computer. That is, it refers to:

- what the main parts are;
- how they are physically connected to each other; and
- how they work together;

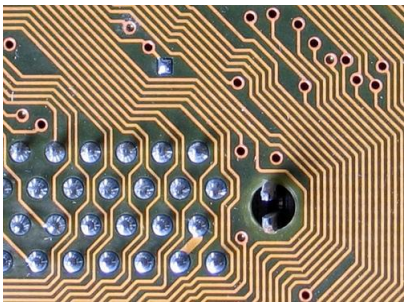
Although all of the parts of a computer are connected to each other by the *motherboard*, the operating system is essential in order to control how those parts talk to each other. Without the operating system, the parts of the computer would not be able to do anything that the user needs them to do! We need to know some basics about how the main parts of a computer are physically connected to each other before we can truly understand what an operating system does.



A motherboard

In this chapter we will look at a general map of a computer's architecture, showing the physical structure that allows all of the parts to exchange information and instructions. We will also take a brief look at some of the major components of a computer that an operating system must be concerned with, including the CPU, memory and how devices actually talk to each other.

A Map of Your Computer's Architecture



Lines (traces) on a motherboard are like roads in a city

You can think of the *motherboard* as a big city, and all of the parts of the computer as buildings throughout the city. Of course, there are roads to get between all of the buildings. These roads are those little metal lines (called *traces*) running all over the motherboard. These traces are part of what is called the *system bus*. There are several different busses on the motherboard, depending upon which devices they are connecting.

Some of the major components of your computer's architecture (the main buildings) that are controlled by the operating system include:

CPU – Central Processing Unit

- This is the brain of your computer. It performs all of the calculations.

RAM – Random Access Memory

- This is your system memory.
- This is like a desk, or a workspace, where your computer temporarily stores all of the information (data) and instructions (software or program code) that it is currently using.
- Most computers today have between 1 to 4 Gigabytes (GB) of RAM.

Graphics

- Many computers have a dedicated system bus and expansion card slot just for a video card.
- Many video cards include their own memory so that you do not need to use up all of the RAM to run your monitor.

I/O Busses

- Special busses (roads) connecting all of your input/output devices to your motherboard.
- The three main types of I/O busses are ISA, PCI and USB.

- ISA – Industry Standard Architecture
  - This was the industry standard in the 1980s and early 1990s.
  - It is now used to provide support for older and slower devices.
  - Common devices connected to the ISA bus might include an older modem, a joystick, a mouse, or a printer (using the older, wide-style printer port).

- PCI – Peripheral Component Interconnect
  - This is for newer and faster devices than ISA.
  - You can think of this like a wider road, with a faster speed limit!
  - Some common devices connected to the PCI bus include your network card, EIDE devices (hard disk, CD/DVD drive, etc).

- USB – Universal Serial Bus
  - Many new devices can connect to your computer using a USB port.
  - Examples include webcams, MP3 players, printers, PDAs, etc.

Figure 1.1 (below) is a diagram of the architecture of these main components (how they are all connected)

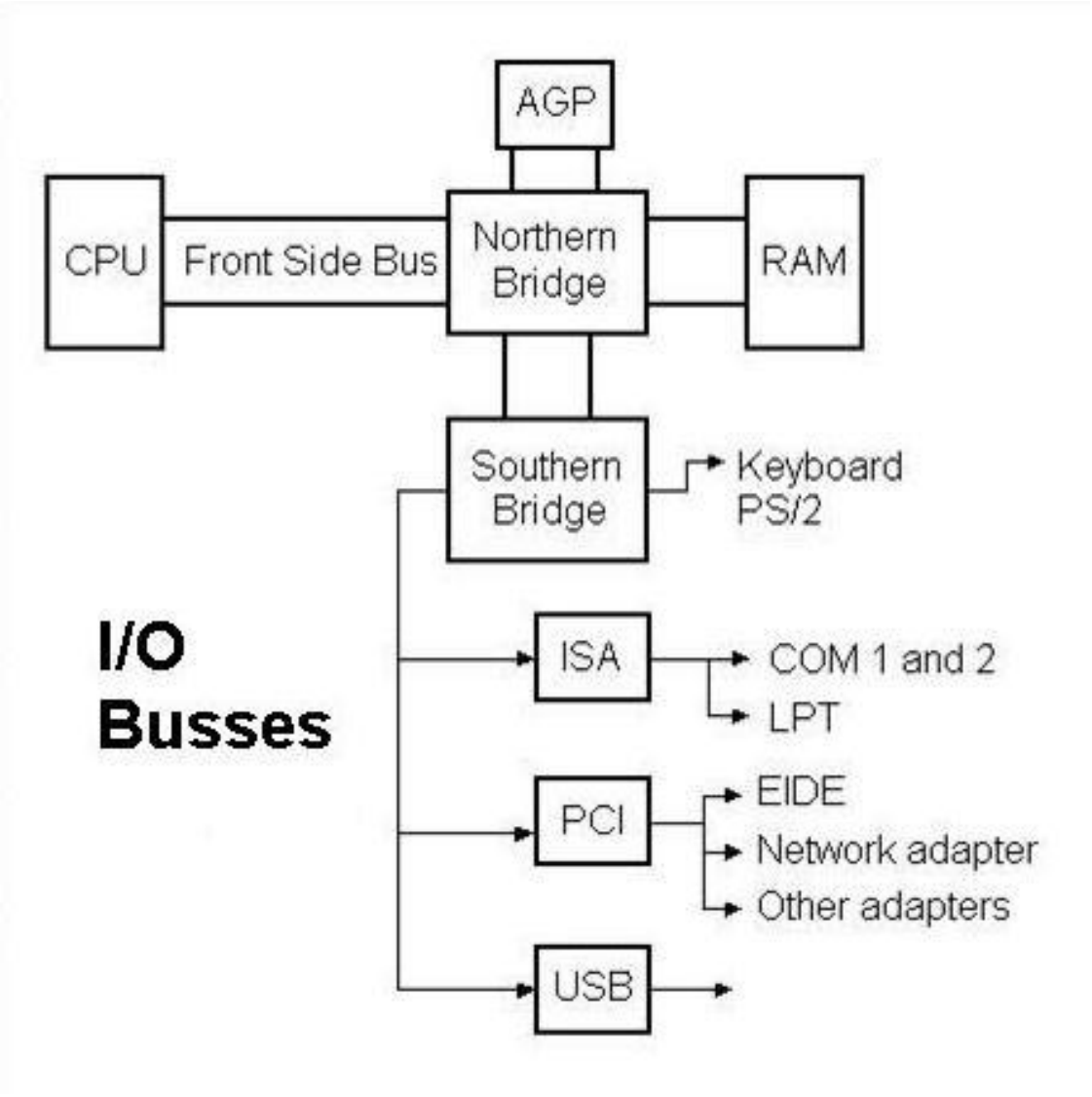


Figure 1.1
Diagram of system bus architecture

Busses and Bridges

In Figure 1.1 we see that the major components are all connected by different *busses*. The *Front Side Bus* provides the main connection between the *CPU*, *RAM*, the *graphics* card (*AGP – Accelerated Graphics Port*), and the *Northern* and *Southern Bridges*. The Front Side Bus looks wider than the *I/O busses* because it is wider and faster. It contains more wires (traces) for the transmission of data between the devices. In the comparison to a city, the Front Side Bus is like a major freeway with a fast speed limit. The smaller I/O busses are like smaller side streets. Some of the I/O busses are narrower and slower than others.

The two bridges in Figure 1.1 perform the same function inside your computer that would be performed by bridges or roundabouts in a city. They are major intersections where data from

different devices cross paths. Of course, like any bridge or roundabout, there needs to be traffic laws to govern who goes first. If there were no rules (and no police to enforce the rules) then everyone would crash together. In computer terms, your data would become corrupted, and no information would ever reach its destination.

Figure 1.2 (below) compares the system bus architecture to a series of city streets with roundabouts:

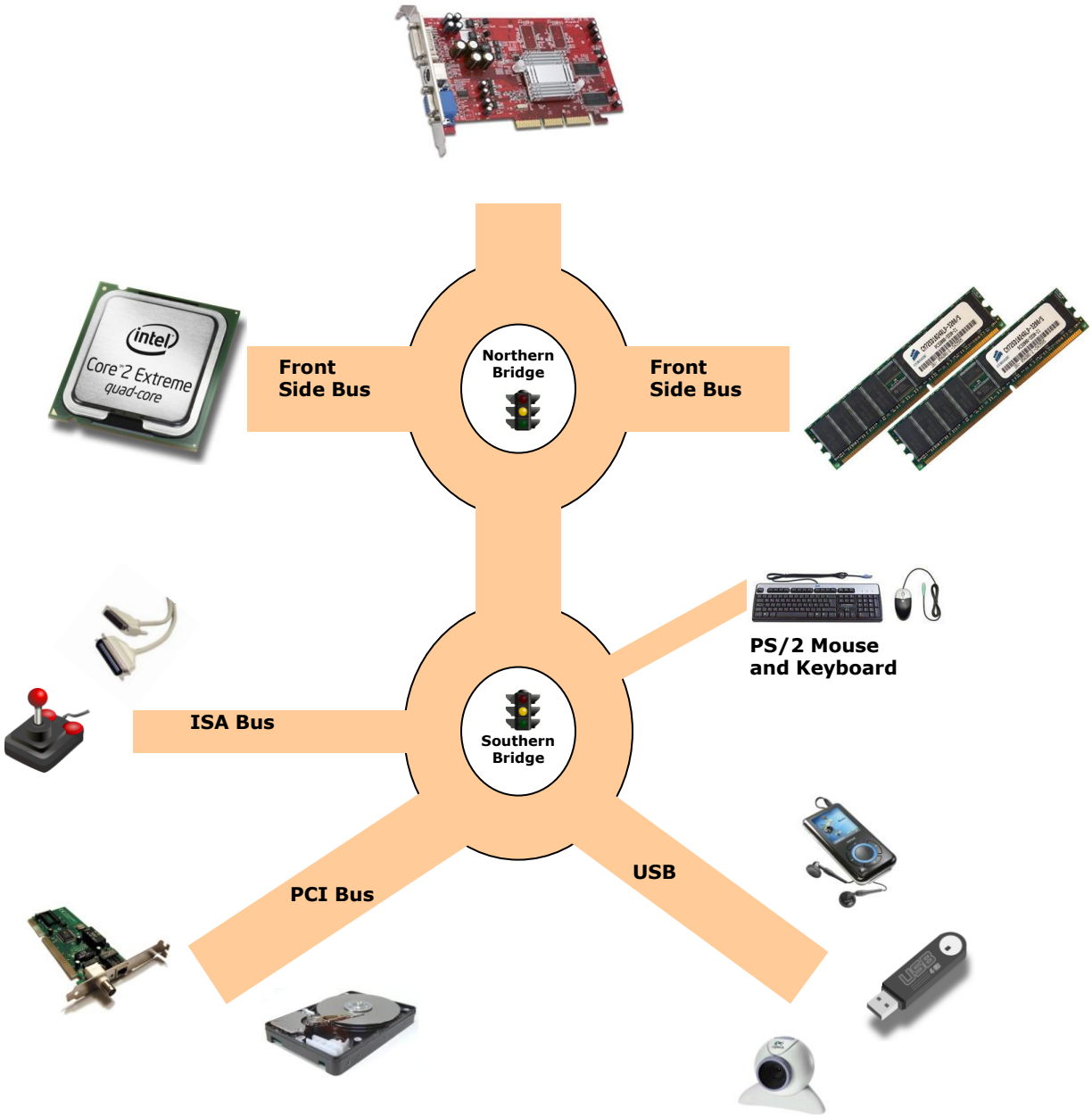


Figure 1.2
A different view of system bus architecture

The Central Processing Unit (CPU)

The *CPU* is the brain of your computer. It performs all of the calculations. Of course, in order to do its job, the CPU needs commands to perform, and data to work with. These instructions and data travel to and from the CPU on the system bus. The operating system provides rules for how that information gets back and forth, and how it will be used by the CPU.

Inside the CPU

Inside the CPU there are many important parts:

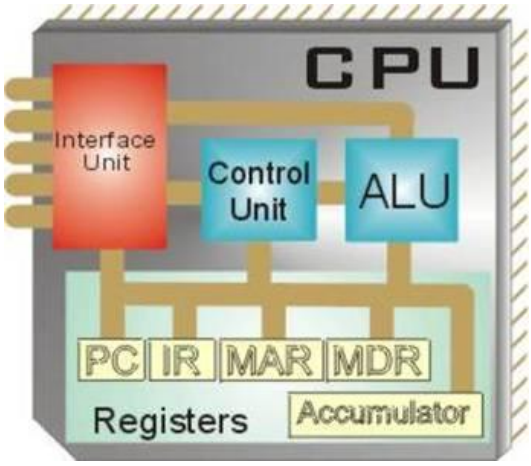


Figure 1.3
The parts inside a typical CPU

- The *Arithmetic Logic Unit* (*ALU*), which performs the calculations
- The *Control Unit*, which controls the flow of data inside the CPU
- The *Interface Unit*, or the *I/O Unit*, which acts like a gate for information entering and leaving the CPU
- *Registers*, which temporarily hold data and instructions waiting to be used
- The *Program Counter* (*PC Register*), which is a special register holding the address of the next instruction the CPU needs from the RAM

The Fetch—Decode—Execute Cycle

The CPU finds, interprets, and executes program code using a specific cycle, as follows:

1. The CPU looks in the PC register for the location of the next program instruction.
2. The CPU retrieves the next instruction from RAM, and places it in a register.
3. The CPU changes the PC register with the address in RAM for the next instruction.
4. The CPU performs the first instruction, and repeats the cycle until the power is lost.

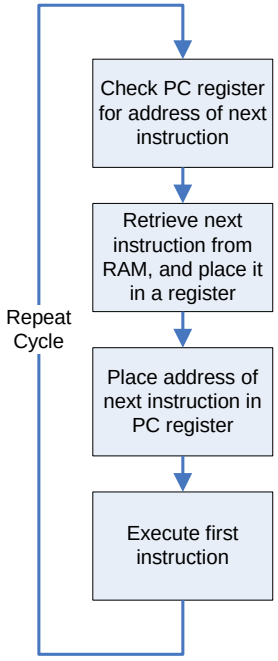


Figure 1.4
The Fetch—Decode—Execute Cycle

Memory

Memory is stored on Radom Access Memory (RAM) chips. A typical computer today has between 1 Gigabyte (GB) and 4 GB of RAM.

Memory is used to:

- Store data
- Store commands (instructions)
- Store system settings

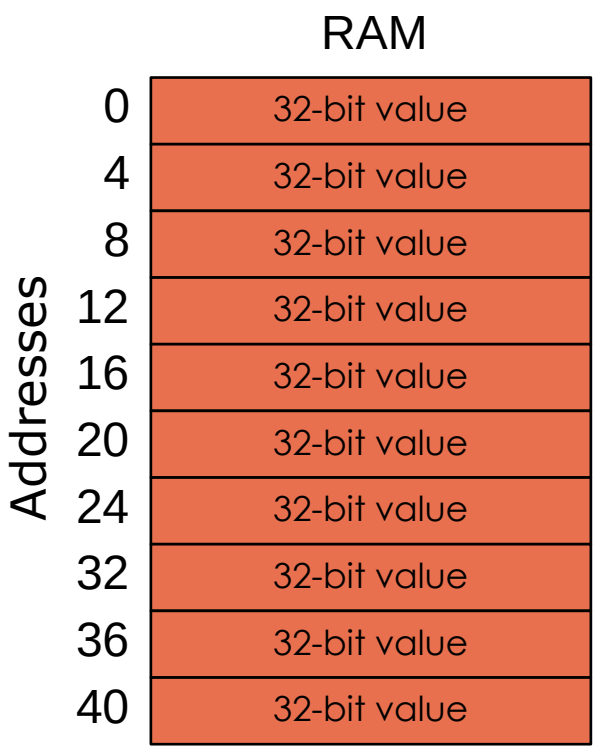


Figure 1.5
Address structure of RAM

Each RAM chip contains millions of address spaces. Each address space is the same size, and has its own unique identifying number (address). The operating system provides the rules for using these memory spaces, and controls storage and retrieval of information from RAM.

Fast fact – RAM is a device!

Without an operating system, a computer would not be able to use RAM chips. This is because your computer treats the RAM chips like a device that has been installed (just like a webcam, or a printer). When your computer first starts up, it can only use a small amount of RAM memory (1 Megabyte (MB)) that is built into the motherboard. Device drivers for RAM chips are included with the operating system, and must be loaded as part of the boot process in order for the RAM to work!

Problem: *If RAM needs an operating system to work, and an operating system needs RAM in order to work, how does your computer activate its RAM to load the operating system?*

Talking to Devices

Devices talk to each other and to the CPU. They need to communicate in order to share information, and in order to be told what to do! There are two types of devices that are controlled by information from the CPU:

- Programmed devices, and
- Interrupt-driven devices

Programmed Input/Output Devices

Programmed I/O devices need to be completely controlled by the CPU. That means the CPU must stop whatever task it is doing, and focus on the device until it has finished whatever it has been told to do. This wastes a lot of processing time!

Interrupt-Driven Devices

A more efficient way to control devices is by using an *interrupt controller*. The interrupt controlled keeps track of whichever devices need to talk to the CPU, and gives different priority to different devices. For example, the keyboard gets higher priority than a modem. When a device needs new instructions, or when it has finished a task, the interrupt controller issues an interrupt to the CPU (like raising your hand in class). The CPU stops whatever it is doing long enough to talk to the device. Although this is more difficult to program, it results in better computer performance.

Of course, the operating system provides all of the rules for communicating with both programmed and *interrupt-driven devices.*

Direct Memory Access

Sometimes devices may want to talk to each other without 'going through' the CPU. The *DMA Controller* controls access to the system bus, and RAM, and bypasses the CPU. The CPU does not need to get involved in the process, other than to set up the transfer. The CPU will get an interrupt when the transfer is complete.

Direct Memory Access is like adding police officers to a roundabout who will let traffic go through to other streets when the road is clear.

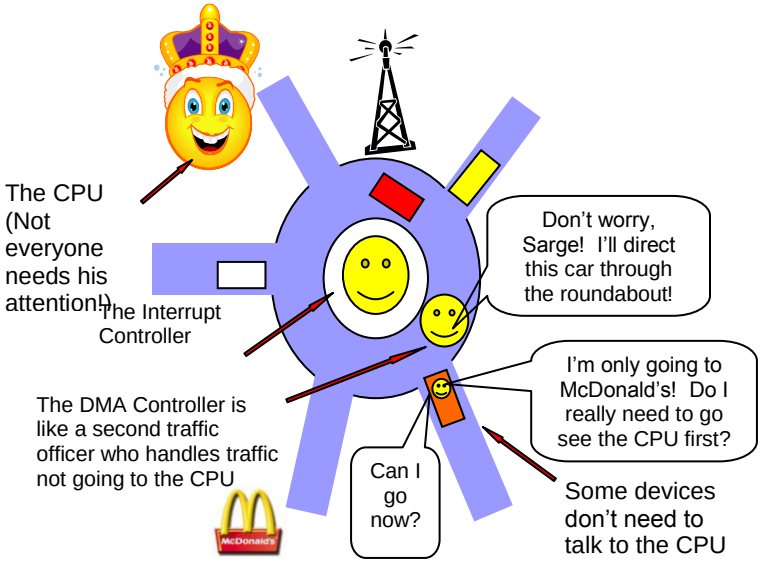


Figure 1.6
DMA is like an extra police office who guides cars through a busy intersection without bothering anyone back at the police station first

Unit 2: Operating System Fundamentals

What is an Operating System?

You need two types of software in order to use your computer (or any other computerized device). These are *applications* and *system software*. Applications are the programs you use to do tasks, such as write a document, surf the web, or play games. System software runs the computer system for you. Another name for system software is an operating system. There are many different operating systems, but they all have a similar architecture (or structure). That is because they must all overcome the same problems and perform the same basic functions. An operating system must be able to:

- Manage system resources
  - CPU scheduling
  - Process management
  - Memory management
  - Input/Output device management
  - Storage device management (hard disks, CD/DVD drives, etc)
  - File System Management

- Simplify the development and use of applications

Examples of Operating Systems

A number of operating systems are available for personal computers. The most popular is Microsoft Windows, which is the operating system used on over ninety percent of the world's personal computer systems. Another popular operating system is *Mac OS X*, which is the operating system used for Apple Macintosh computers (like the Mac Book Pro laptop series). While IMB PCs (mostly Windows) and Mac computers are not directly compatible, it is possible to use virtualization to run one operating system on an incompatible computer.

Another group of widely used operating systems is based on *UNIX*. UNIX was a command line interface operating system developed for large scale computers and networks in the 1960s. The latest generation of operating systems derived from UNIX is called *Linux*. It is a free, open-source operating system that is supported by most computer platforms.

Special Purpose Operating Systems

Operating systems are not limited to just personal computers. Most electronic devices today use an operating system to manage their physical components and to make it easier to develop applications for use on the devices. Examples include the *Symbian, Blackberry, Palm* and *Windows Mobile* operating systems used for personal digital assistants (*PDA*s) and mobile phones. *Specialized operating systems* have even been developed to control computerized aircraft systems (*VxWorks, pSOS* and *QNX* are examples).

The Structure of Operating Systems

Layers

Accessing computer resources is divided into *layers*. The user represents one layer at one end of the system. Your computer's hardware represents the layer at the opposite end of the system. In order to use your hardware to do anything with the computer, you need software. Software forms the layers in between the user and the hardware and is divided up into application software and the operating system. The operating system must be able to manage resources from both the applications and hardware layers.

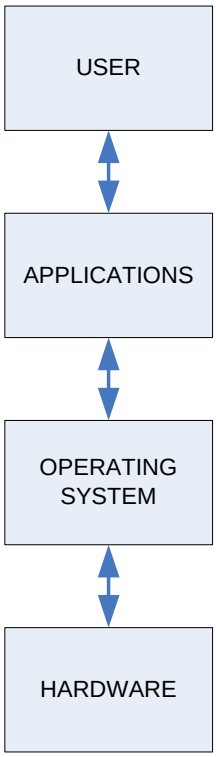


Figure 2.1
Layers in a computer system

In the computer layer system the user interacts directly with software applications. The applications interact with both the user and the operating system. The operating system interacts with the applications and controls the hardware.

Each layer is isolated and only interacts directly with the layer below or above it. If you make changes to any one layer, they only directly affect the layer next to it. For example, if you install a new hardware device you do not need to change anything about the user or applications. However, you do need to make changes to the operating system. You need to install the device drivers that the operating system will use to control the new device. If you install a new software application you do not need to make any changes to your hardware. But you do need to make sure the application is supported by the operating system and the user will need to learn how to use the new application. If you change the operating system you need to make sure that both your applications and your hardware will work with the new operating system.

Running Multiple Operating Systems

It is possible to install more than one operating system on a computer. You can do this by partitioning your hard disk(s) and installing different operating systems on different *partitions*. This can be very useful, because you may want to use different operating systems to perform different tasks. For example, you may have specialized applications that will only work with one operating system, making them incompatible with the rest of your software. When you turn your computer on, you are given a choice of which operating system to use. You can only run one operating system at a time. Figure 2.2 (right) shows the system of layers when multiple operating systems are installed on the same computer.

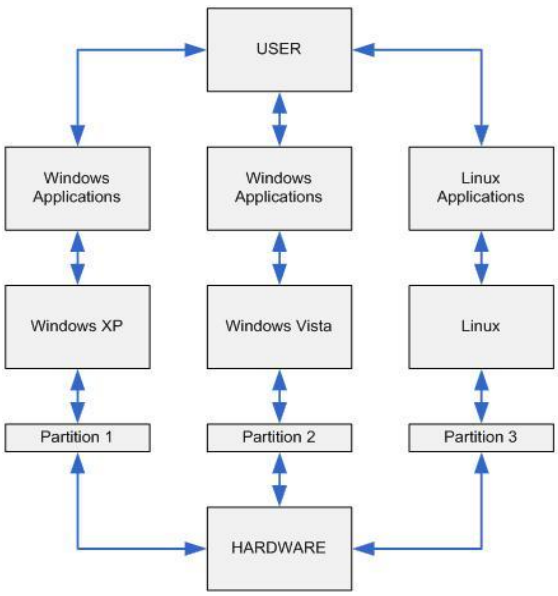


Figure 2.2
Layers in a computer with multiple partitions and operating systems

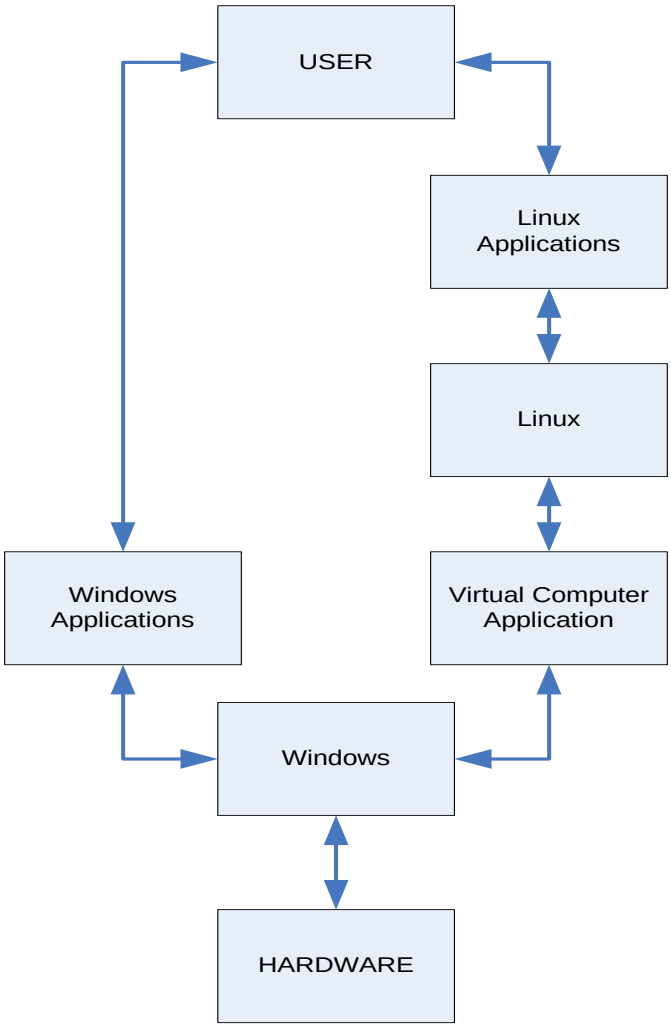


Figure 2.3
Layers with a virtual operating system

Running a Virtual Operating System

What happens if you want to work on applications in two operating systems at the same time? What about if you want to run an operating system that is not compatible with your computer's hardware? (For example, you cannot install the Mac OS X operating system on an IBM compatible PC). You can get around these problems by running a virtual computer.

A *virtual computer* is really an application within one operating system that lets you pretend you have a different operating system installed. Virtual computer applications like VMWare and Virtual PC act as translators. They convert instructions from the virtual operating system into instructions from the real operating system, which then controls your computer's hardware.

Figure 2.3 (left) shows the structure of layers when you run a virtual operating system within a Windows operating system. As far as Windows is concerned, it is simply running another application. Notice that the layers between the virtual computer application and the user are just like the layers for a single operating system (Figure 2.1).

Operating System Modes

A typical operating system has two *modes* of operation. These are like layers of operation within the operating system layer (Figure 2.1). The *User Mode* is concerned with the actual interface between the user and the system. It controls things like running applications and accessing files. The *Kernel Mode* is concerned with everything running in the background. It controls things like accessing system resources, controlling hardware functions and processing program instructions. The *Kernel* forms the core of the operating system, and it acts like a supervisor for everything that is happening in the computer. In the *client-server model* of an operating system, the User Mode is considered a client. That is, the User Mode accesses resources provided by the Kernel (the server). Figure 2.4 (below) shows what operating system functions are controlled by the User Mode and Kernel Mode.

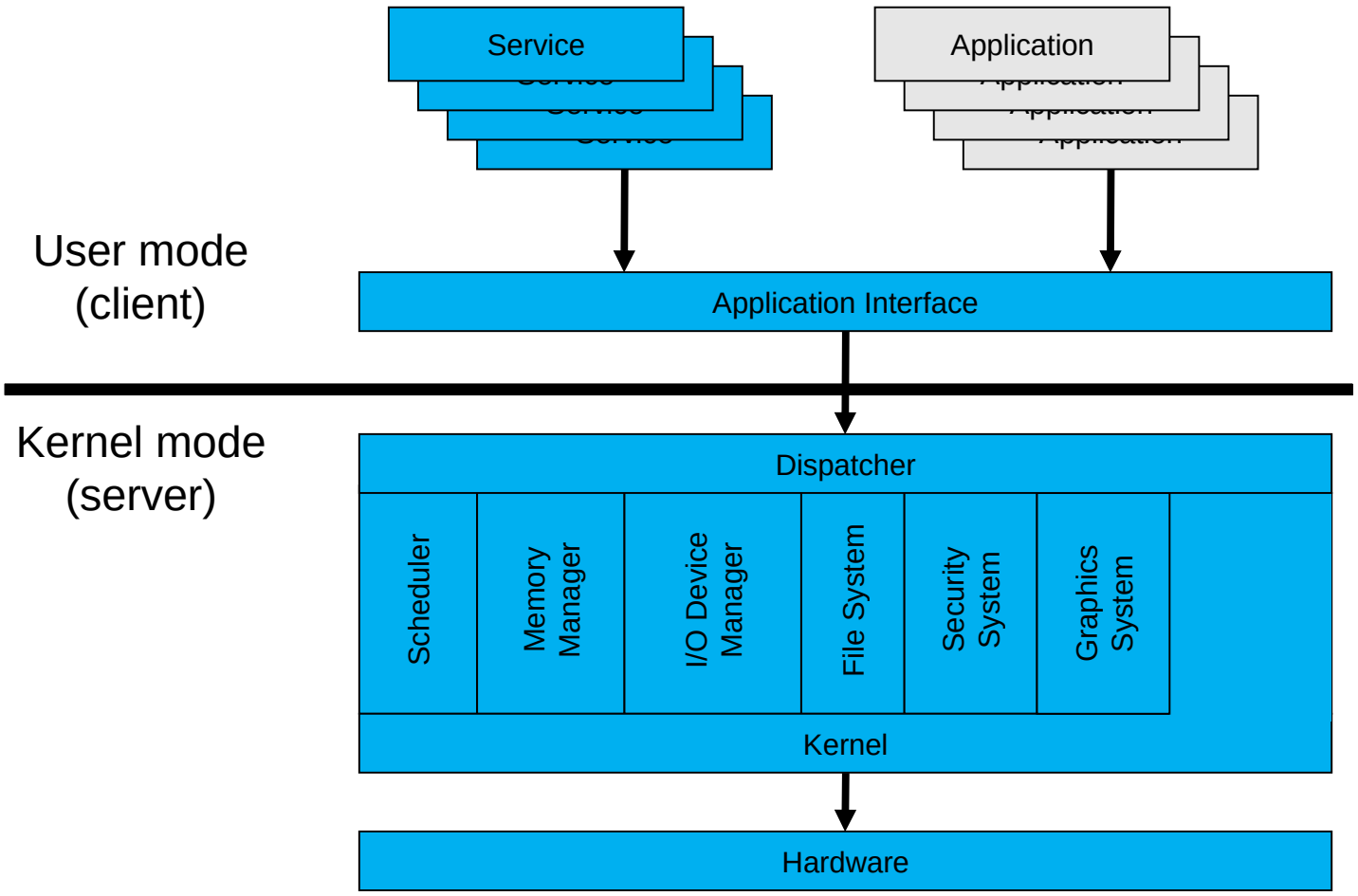


Figure 2.4
Typical structure in the Client (User Mode) – Server (Kernel Mode) model of an operating system

Starting an Operating System

Most personal computers have similar architecture and can use a variety of different operating systems. When a computer is first made, there is no operating system installed. Even after you have an operating system installed, you can remove it and install a different one. As we discussed earlier, you can even have multiple operating systems installed on the same personal computer. This raises the question—how does your computer start the operating system? If you have more than one operating system installed, how does your computer choose which operating system to use?

Your computer is designed to start in stages. In the first stage, you turn on the power supply to your computer. This sends electricity to the motherboard on a wire called the '*Voltage Good*' line. If the power supply is good, then the *BIOS* (*Basic Input/Output System*) chip takes over. At this stage the computer's CPU is operating in *Real Mode* (or real address mode), which means that it is only capable of using approximately 1 MB of memory built into the motherboard. RAM will be initialized later using device drivers from the operating system.

The BIOS chip contains basic instructions for starting up the rest of the computer system. The first thing that it will do is a *Power-On Self Test* (*POST*), which will check to make sure all your

hardware is working properly. If the hardware is all working, BIOS will then look for a small sector at the very beginning of your primary hard disk called the *Master Boot Record* (MBR). The MBR contains a list, or map, of all of the partitions on your computer's hard disk (or disks). After the MBR is found the *Bootstrap Loader* follows basic instructions for starting up the rest of the computer, including the operating system. If multiple operating systems are installed, the user will be given a choice of which operating system to use.

Remember – RAM is a device!

In the first unit we said that without an operating system a computer would not be able to use RAM chips. Your computer treats RAM chips like a device that has been installed. When your computer first starts up, it can only use a small amount of RAM memory (1 MB) that is built into the motherboard. Device drivers for RAM chips are included with the operating system, and must be loaded as part of the boot process in order for the RAM to work!

Problem: *If RAM needs an operating system to work, and an operating system needs RAM in order to work, how does your computer activate its RAM to load the operating system?*

Solution: *Device drivers for RAM are loaded during the Early Kernel Initialization stage.*

The next stage is called *Early Kernel Initialization*. Remember that the Kernel is the core of the operating system, and it regulates all of the background functions of your computer. In the Early Kernel Initialization stage, a smaller core of the Kernel is activated. This core includes the device drivers needed to use your computer's RAM chips. Without the extra memory provided by RAM, it is not possible to run the more complicated code for the remainder of the operating system.

Once the Early Kernel Initialization is complete, the CPU switches to *Protected Mode*. The computer can now take advantage of the extended memory address system provided by RAM, and the operating system's Kernel is fully initialized. Only at this stage are the first User Mode processes initialized, and the user can begin interacting with the operating system, applications and hardware. Figure 2.5 (below) shows the stages in starting an operating system.

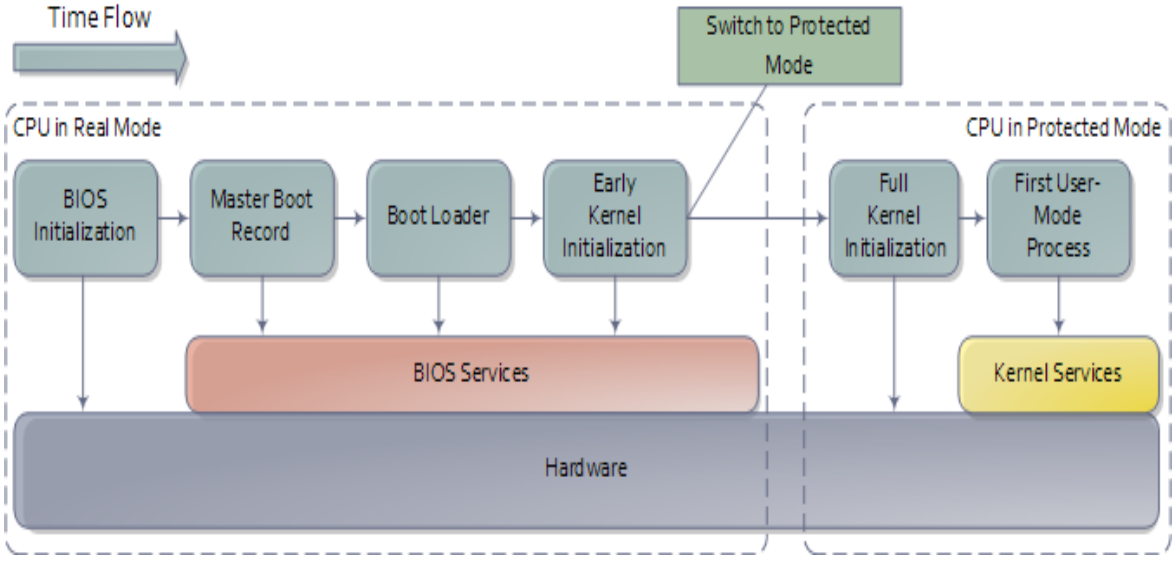


Figure 2.5
Stages in the startup of an operating system

Interfacing with an Operating System

Types of User Interfaces

An operating system operates the functions of a computer. It also provides a way for users to interface with, or access, a computer's applications, resources and hardware. There are two main types of user interfaces for an operating system:

- Command Line Interface
- Graphical User Interface (GUI)

A *command line interface* uses typed commands to issue instructions to the computer. It can be more difficult to use because the user must type the precise commands and locations of files. *DOS* (*Disk Operating System*) and UNIX are examples of command line interface operating systems.

A *GUI* uses graphics (or pictures) and menus to help the user access resources and issue commands. Windows XP, Linux and Mac OS X are examples of GUI operating systems.

Figure 2.6
Examples of a command line and GUI interface

The Command Line Interpreter

Applications are accessed at the User Mode level. This means that they do not have the authority to directly access system resources that are controlled at the Kernel Mode level. When a user types a command (in a command line interface) or performs a task within an application (using a GUI), *processes* are initiated. Since those processes usually require access to system resources, the command line interpreter converts them into system actions (called *system calls*). Most interpreters execute applications to perform the system calls.

Managing System Resources

An operating system needs to manage a wide range of system resources. Some of the main resources controlled by the operating system include CPU scheduling and process management, memory (RAM), access to peripheral devices and file system management.

CPU Scheduling

Memory is like a workspace for the information and program instructions that are being used by the computer. The Central Processing Unit (CPU) is the component that actually does the work. The CPU performs all of the program instructions, sends commands to devices, and receives information back from those devices. Just like memory, something needs to regulate which devices and applications get to use the CPU, and for how long. This task is handled by the operating system.

Most modern CPUs and operating systems can handle *multitasking* and *multithreading*. That is, they can run more than one application at a time and they can process threads from more than one device and application at a time. However, the CPU has limited resources. It needs a schedule of processes to carry out, or nothing will run properly.

In older operating systems, it was up to each application to determine how long it needed to use the CPU and what priority it should be given over other applications or interrupts from devices. This was called *cooperative multitasking*. Unfortunately, this system was rather like having roads with no traffic laws or police officers. If someone wanted to take complete control and cut off all other traffic, it was possible. Newer operating systems use *preemptive multitasking*. That is, the operating system sets out the rules for the use of the CPU and enforces those rules. Preemptive multitasking means that the operating system shares the CPU between everything that needs its attention. It also gives priority to certain devices and applications based on how critical they are to keeping the whole system functioning.

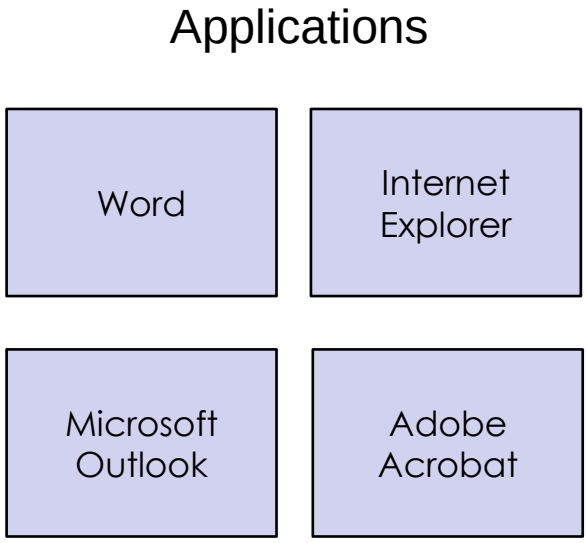


Figure 2.7
An operating system shares the CPU between applications

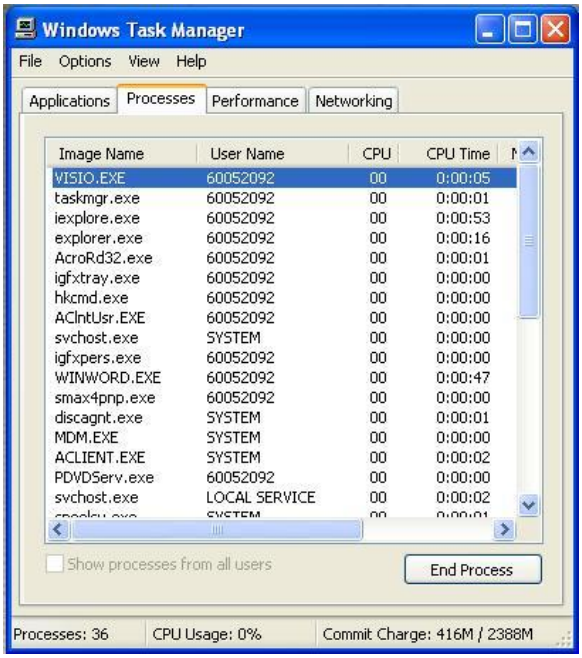


Figure 2.8
The Windows XP Task Manager showing processes from the process table

The Process Table

As previously discussed, processes need to share the CPU. Sometimes the CPU does not complete an entire process before the operating system tells it to start working on another one. This system of sharing is what makes multitasking possible. Keeping track of all of the processes is done with the *Process Table*. The Process Table lists all of the processes that are currently being run, those that are waiting to be executed and those that have been temporarily suspended. It also keeps track of the current status, or *state*, of each process. This allows the CPU to restart those processes again when they are needed. Figure 2.8 (right) shows processes from the Windows XP process table, as displayed in the Task Manager. We will take a closer look at processes and process management in Unit 3.

Memory Management

Memory is used by a computer to temporarily hold data and instructions that are being used by applications, the operating system and hardware devices. Since a typical computer has between 1 and 4 GB of memory (RAM), and since modern operating systems can run many devices and applications at the same time, there is a lot of memory to keep track of. As we noted in the previous chapter, RAM is divided up into small spaces (usually 32 bits). Each space has its own address. An operating system must be able to keep track of all of those memory addresses and how they are currently being used. The operating system typically performs three major functions with respect to memory management:

1. Gives memory to each application and device as needed;
2. Protects applications (and their data) from each other;
3. Protects the system from 'bad' applications (that might try to use too much memory, or corrupt data from other applications);

We will take a more detailed look at how operating systems manage memory in Unit 4.

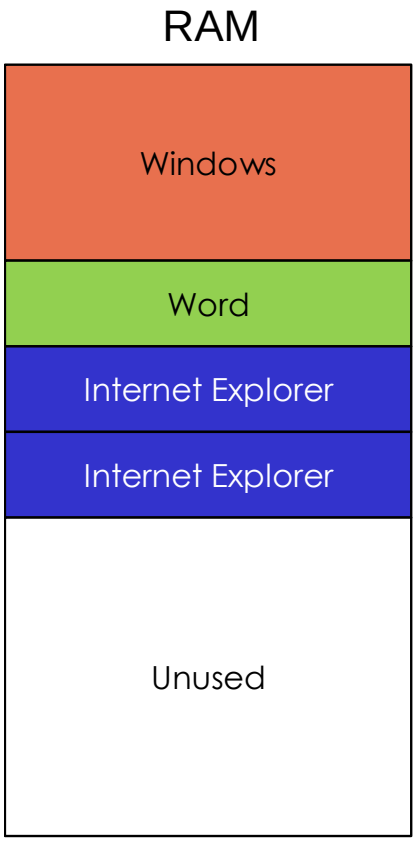


Figure 2.9
An operating system shares memory between applications

Peripherals

Peripheral devices are hardware devices that are connected to the computer by connection ports on the motherboard. Examples include the monitor, keyboard, mouse, webcam and printer. Peripheral devices are difficult to program and manage. Although many different applications need to use peripheral devices, the task of accessing them is simplified by the operating system. Applications do not directly access peripheral devices. These devices are programmed and controlled using *device drivers* provided by the operating system. When an application needs to use a device it talks to the device drivers. The device drivers then tell the device what to do.

When a new device is installed, the operating system looks for built in device drivers or adds new drivers to control the device. Most newer operating systems and devices are Plug and Play compatible, which means that the operating system will handle everything related to installing the new device and its drivers without any action from the user (other than confirming installation options).

We'll take a closer look at Input/Output management in Unit 5

File System

Your computer contains more than just your hardware resources. It also contains all of the information that you use and manipulate. This information is stored on your hard disk, CD/DVD discs, and removable storage devices. Your operating system controls the actual physical operation of these storage devices. It also helps you to manage the files stored on these devices.

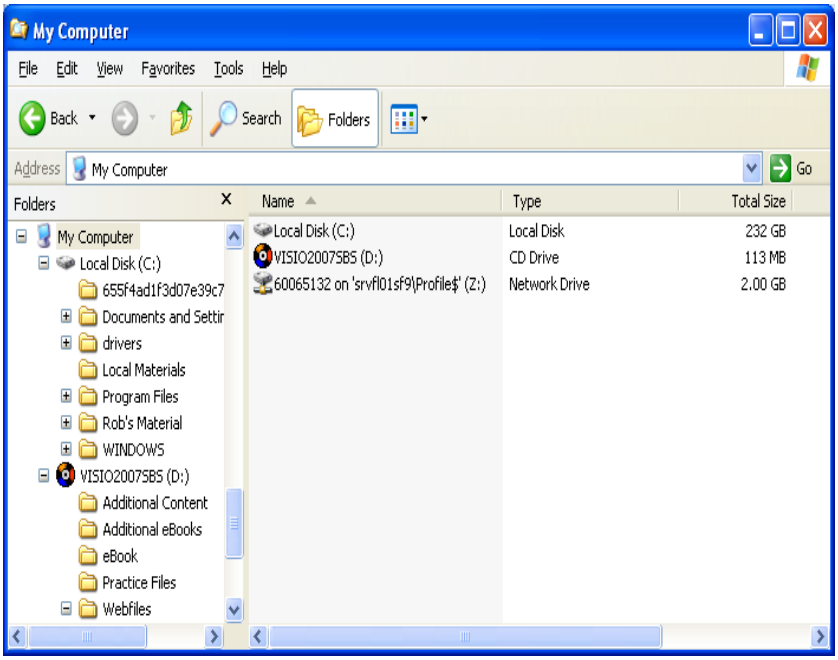


Figure 2.10
Windows Explorer is a tool for viewing and navigating your computer's file system

Different operating systems use different file systems to encode and organize your information. For example, older versions of Windows used either *FAT16* or *FAT32* (FAT stands for File Allocation Table). These older file systems limited the amount of information you could store on a hard disk, so newer versions of Windows (XP, Vista, and Windows 7) use *NTFS* (*New Technology File System*). NTFS lets you store up to 2 Terabytes (TB) of information on a single volume and provides greater file security than the older FAT file systems. Other operating systems use different file systems such as *EXT3* for Linux, or *HFS+* for Mac OS X.

Regardless of which file system an operating system uses, the operating system must perform certain key file management tasks for the user:

- Manage the storage and retrieval of information; and
- Provide a common, easy to navigate system for viewing and accessing storage devices and the information stored on them.

We'll take a closer look at file system management in Unit 6

Unit Summary

Computerized devices need an operating system to control the actual functioning of the device. Whether the device is a personal computer, mobile phone, or computerized aircraft controls, an operating system must provide some way for the user to interface with the device. Modern operating systems use a graphical user interface approach to simplify access to applications and hardware resources. Operating systems act as one layer in the functioning of a device. Other layers include the hardware, applications and the user. It is possible to install more than one operating system on a computer, which creates multiple sets of layers (however, only one of these sets of identical layers can operate at any given time). It is also possible to use virtualization to simulate the use of two different operating systems at the same time. Regardless of which operating system is being used, there are similar tasks that the operating system must perform. The primary tasks performed by the operating system include the management of CPU scheduling and tracking processes, memory management, management of Input/Output systems and file system management.

Key Terms

Applications
BIOS
Blackberry OS
Bootstrap loader
Client-server model
Command Line Interface
Command Line Interpreter
Cooperative multitasking
Device drivers
DOS
Early Kernel Initialization
EXT3
FAT 16
FAT 32
File system
Full Kernel Initialization
Graphical User Interface
HFS+
Kernel mode
Layers
Linux
Master Boot Record (MBR)
Max OS X
Mode
Multitasking
Multithreading
NTFS
Operating system
Palm OS
Partition
PDA
Preemptive multitasking
Process
Process state
Process Table
Protected mode
pSOS
QNX
Real mode
Special purpose operating system
Symbian
System call
System software
UNIX
User mode
Virtual computer
Virtualization
Voltage good line
VxWorks
Windows Mobile

Unit 3: Processes

Processes and Multitasking

Many people like to try to speed up several tasks by performing them at the same time, such as using a mobile phone while driving. While this seems like you are accomplishing two things at the same time, the truth is that your brain specifically focuses on just a single task at any specific time. The act of talking only occurs while you are not actively making decisions about the task of driving. To put this into perspective, if you know you are about to have an accident, you will stop talking.

To further illustrate the idea of performing simultaneous actions consider the problem of reading a text message while watching TV. Your eyes can only look at one device at a time. You must switch back and forth between the two devices, or look at the mobile phone only while unimportant things are happening on the TV.

A CPU inside a computer is simply a high speed calculator that can perform relatively simple operations on a set of data. If we ignore for the moment the idea of *dual* and *quad core CPU*s, the CPU can only process a single instruction at any given time from a program. If we would like to have more than one program executing on the processor at the same time, the programs will need to take turns using the CPU. Since the computer switches back and forth between the two programs often enough, then it will look like both applications are running at the same time.

This unit takes a detailed look at the definitions of *processes* and *threads*, and how the operating system manages processes and threads in *multitasking* environments. The first section deals with the definition of a process, and the concept of *process states*. We then take a look at *state changing*, process creation and stopping processes. From there, we compare processes to threads, and take a look at why threads are important. This is followed with a detailed look at *inter process* (and *inter thread*) *communication*, including process synchronization, memory sharing, the use of *signals* (or *semaphores*), *critical sections*, and the use of *message queues*. We conclude by looking at how an operating system actually handles *process scheduling*. This will include topics like completion scheduling, *round robin scheduling*, *priority-based scheduling*, and scheduling in multi-core/multi-processor environments.

Process

Definition

In order to manage individual applications (or what we often refer to as programs) most operating systems use the term *process*. An application or program is a set of instructions. A process is the actual execution of those instructions, along with the memory and I/O devices assigned to execute the given instructions.

State Machines

Without the need to switch from one process to the next, the creation of an operating system is significantly reduced. In fact, many embedded systems that have no multi-tasking often do not include an operating system for cost reasons.

If we consider the act of attempting to read a text message while we are driving, we know that the driver will only take their eyes off the road when the traffic appears to be under control. The driver will then quickly look at some of the text on the mobile phone and then return to looking at the road. This process will continue as long as there are text messages to be read or places to drive to.

If we draw a small diagram showing these two actions we have something like this:

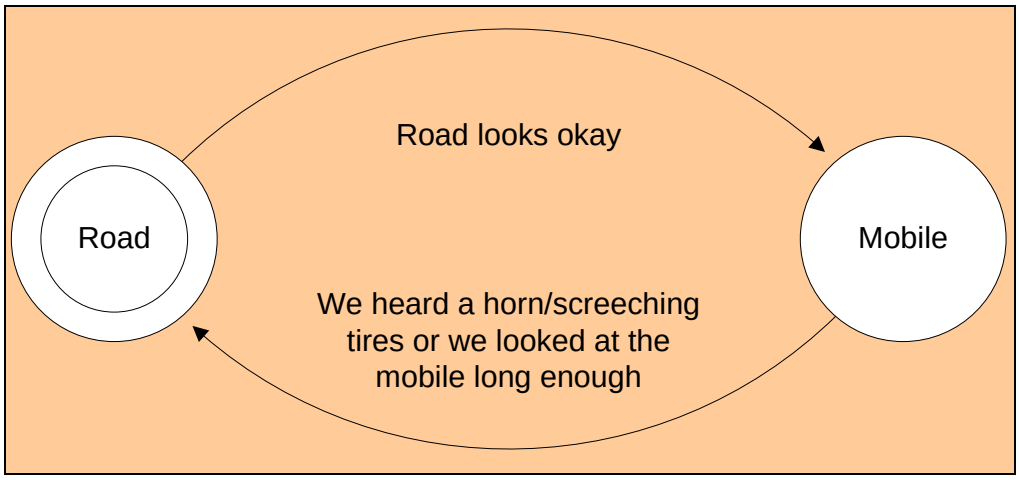


Figure 3.1
Example of a simple state machine

This diagram shows that we start off by watching the road and when we feel that the road conditions look okay (no cars or pedestrians) we then look at the mobile phone. We then read the message on the mobile phone until one of two things occurs:

1. We hear something that needs our attention such as some screeching tire or we notice something in our peripheral vision.

2. We have been looking at the mobile phone for some time and realize that we should probably see if we are about to hit something.

Figure 3.1 (above) is used in many computer design documents and is called a *state machine*. The circles represent the state of the machine and the arrows represent actions that cause us to change from one state to another. In the case of our texting driver, there are two states: looking at the road, or looking at the mobile. Although it may be possible to hold the phone in such a way that the peripheral vision encounters most of the road, it remains a fact that you cannot actually look at both the phone and the road at the same time. Regardless of the opinion of texting while driving, a single core CPU can only perform one instruction at a time which is the whole reason for describing this analogy. The important thing to take away from this analogy is that some things in life are modeled really well by state machines and that events can cause some resources to change from one state to the next.

Computer Process

We now return to the world of computers and processes again. The fact that a single-core CPU can only process one instruction stream at a time means that two applications must take turns running on the CPU. Although a rather obvious statement, if we consider just one single process we realize that it must be either running or is must not be running. This sounds suspiciously like a set of states. In addition to two states, the next question that should be asked is how does a process go from 'not-running' to the 'running' state?

We will call the 'not-running' state the '*Ready*' state because it indicates that the process would like to run but currently cannot (suggests that some other process is actually running). We will also introduce two new terms:

- *Dispatch* – means that the operating system has decided that the process should start running now.

- *Interrupt* – means that the operating system has decided that the process should now stop running so that another process can have a turn.

Putting all of this information into a single state diagram produces this version.

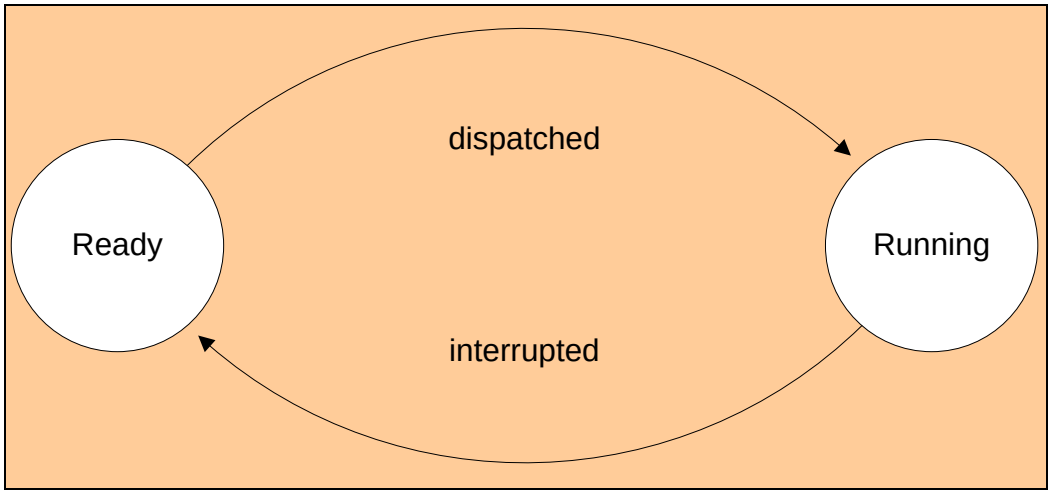


Figure 3.2
A computer-based simple state machine

Many operating system students see diagrams such as this and easily understand the concept, but many textbooks fail to remind the students that there are in fact probably several processes (each with their own state machine) going at the same time.

The operating system would normally keep a list of all processes current loaded in a table known simply as the *process table*. The process table would need to keep the current state value. So if an operating system were running four programs (Word, Internet Explorer, Excel, and Visio) the table might look like this:

Process ID	Name	State
4	Word.exe	Running
6	IExplorer.exe	Ready
7	Excel.exe	Ready
8	Visio.exe	Ready

Figure 3.3
A sample process table

At no time would it be possible to have two processes both with the state '*Running*' as we are assuming for now that the CPU has only a single core.

Processor Preservation

An important concept in operating system design is the ability to hide multi-tasking concepts from the applications so that they do not know that other applications are running at the same time and sharing the CPU.

The registers in the CPU become a resource that is shared by every process on the operating system, including the operating system itself. Keeping these values correct is critical to making an operating system that functions correctly. As an example if one application performs the instruction "mov $50, %eax" it means that the program expects the value 50 to be placed into the EAX register. If this instruction were executed then another process ran that put the value 0 into the EAX register, the old value of 50 needs to be replaced before the first process runs again.

Each process on the system requires a place where the registers can be stored while the process is not running. The operating system can either put this information directly into the process table or it can store the information somewhere else but keep a pointer to the information in the process table.

State Changing

Changing the state of a process from one state to another is usually the responsibility of the operating system. The actual switch could occur because the process has informed the operating system that it is finished, or the operating system could determine that the process has simply used too much time and it is another process' turn to run.

Of course, the operating system itself is software that must run on the CPU along with the applications. So how does the application actually become active so that it can stop Word in the example above? Many hardware platforms include a clock that periodically fires interrupts and the operating system has likely attached a function called a *scheduler* to the *interrupt* so that every so many milliseconds (a typical number is 10 ms) the operating system scheduler gets executed.

When the scheduler is called (by the interrupt) the scheduler code can manipulate the process table and set the state of the processes involved and then allow the new process to take over.

As an example suppose that the process table looks like the example above with the four processes and Word.exe executing. Here is the order in which things happen.

1. The word process is executing normally using the CPU registers as it requires.
2. The clock triggers an interrupt.
3. The interrupt service routine attached to the clock runs the OS scheduler.
4. The scheduler searches the process table for the process in the Running state.
5. The scheduler changes the state of the running process to "ready".
6. The scheduler copies all of the current value of the registers into the process table (or information block)
7. The scheduler looks in the table to find the next process to run. The topic of selecting a process to run is covered in the next chapter but for here let's assume that iexplorer is about to run.
8. The scheduler marks iexplorer as "Running"
9. The scheduler copies all of the registers for iexplorer into the CPU from the process table.
10. The scheduler then jumps to the next instruction that iexplorer should perform.

Preemptive and Non-preemptive Switching

The steps above describe a type of *process switching* known as *preemptive switching*. The operating system always has control of the computer by way of the interrupt service routine and will always be able to stop the currently executing process.

In a *non-preemptive* operating system, the operating system never interrupts the currently executing process but waits for the process to release control voluntarily. In such a system, the operating system is generally less complex. However, if a process does not wish to cooperate with the other processes on the computer then this could lead to other processes, including some operating system parts, never having a chance to execute.

The most recent popular non-preemptive operating system was Windows 3.1. In this operating system, a programmer who accidentally created certain types of infinite loops would often have to reboot their computer in order to stop their program. As a result, most operating systems today utilize preemptive process switching so that no single process can monopolize the computer.

Blocking

Generally the CPU executes instructions very quickly and transferring data to and from RAM also takes place with very little delay. Unfortunately, if a process wishes to interact with some external device the CPU cannot do very much useful work.

As an example, suppose we have written a very simple program that does nothing until the user presses a key on the keyboard. The computer instructions may look something like this:

```
Main() {
    System.in.read(value);
}
```

When this process is started, there will be some instructions required to get the application ready, but eventually the instruction "wait for key" is reached. When the process reaches this instruction what does the operating system do with the process state? In our two state process model introduced earlier (see Figure 3.2), leaving the process as 'Running' is probably not a good idea because it really is not doing anything. Moving the process to the 'Ready' state does not help either because the process is not actually ready to run. It needs to wait for the user to do something.

We solve this problem by the introduction of a third state that we will call '*Blocked*'. This state tells the operating system that the process is currently waiting for something, and therefore it should never be considered by the scheduler when a new process is being selected for running. The term blocked exists because the execution is being "blocked" by some external event. Some operating systems might use the term *wait*.

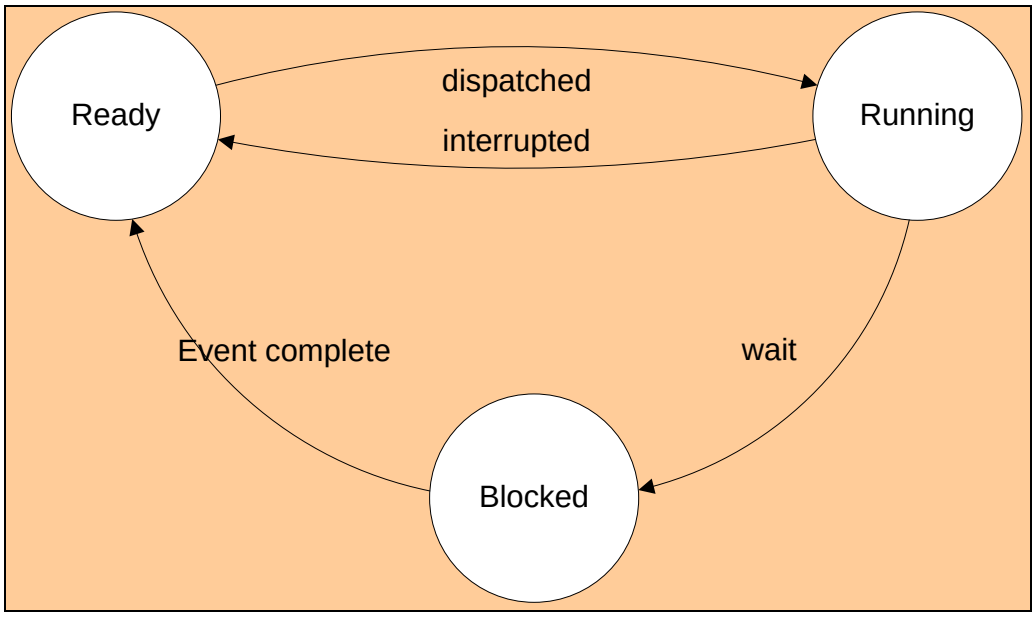


Figure 3.4
A state machine with a blocked state

In this model, a process can never become blocked unless it is actually running. Once a process has been put into the blocked state, it must eventually be *unblocked* or *released* and put back in to the 'Ready' state before it can run again.

How does the operating system manage these extra state changes? As mentioned in the earlier sections of the book, the operating system is responsible for managing all system resources including I/O devices. In the case of waiting for the user to press a key, the program actually issues a request to the operating system (through a function call) asking to wait for a key. In response, the operating system changes the state of the process to 'blocked' and runs the scheduler *algorithm* to find a new process to run.

How does the process get out of the blocked state? Each I/O device has its own mechanism that it uses to interact with the operating system. We will consider just the keyboard example here. During boot up, the operating system has probably attached a service routine to the keyboard interrupt. Each time that the user presses a key, it generates an interrupt which causes the operating system routine to generate a key event and the scheduler routine will be called. The scheduler will then look through all of the blocked processes waiting for key events and the state will be changed from blocked to ready. Whether this keyboard interrupt causes the currently running process to be moved from Running to Ready depends on the operating system itself.
If you were to examine the states of processes on most computers, you will probably find that just about every process on the system is blocked most of the time.

Having a lot of blocked processes is a good thing. This means that the CPU is not actually being used for any useful work. Normally when a CPU does works it consumes power, while if it is not doing work it can be put into a low power state. Tricks such as these are used in mobile devices such as phones in order to extend the life of the battery. A process that is actively using the CPU will consume more power, which is why batteries do not last as long when you are watching videos on your mobile phone than when you are simply on stand-by waiting for calls.

A Typical State Model

There are two additional states that most operating systems include in addition to the Ready, Running, and Blocked states. These two extra states are used for house keeping, and only exist for a short amount of time during the creation and the removal of processes.

When a new process is created there needs to be a new entry placed into the process table. Setting the process table entry actually takes a bit of time. If the new process was marked as 'Ready' and the operating system was in the middle of initializing the other columns in the process table, and suddenly a reschedule operating occurred, how would the operating system know that it should not select this new process? We control this by the introduction of a '*New*' state. This is simply a place holder that is used by the operating system until the process is actually ready to start going.

Similarly there is an '*Exit*' state that is used when a process is being cleaned up. This state is also temporary, and the operating system will mark the process first then remove it from the process table.

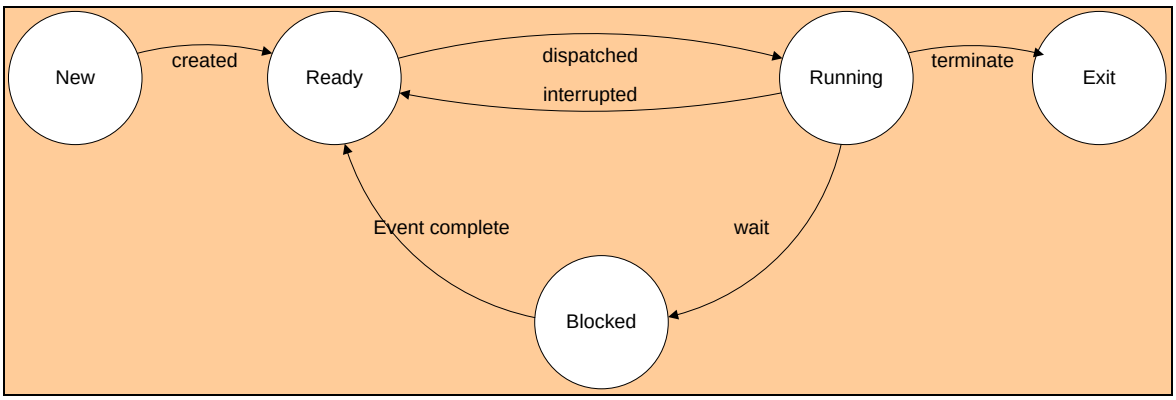


Figure 3.5
A typical state machine

Switch Prevention

There may be situations where a process is in the middle of doing something really important and it would like to ensure that no other process is allowed to run in the meantime. There are two mechanisms that are typically available: disabling the scheduler and disabling interrupts. By disabling the scheduler the process is requesting to the operating system that even if the timer fires, the process would like the operating system to skip the selection of a new process. In the case of disabling interrupts, the program is actually asking the CPU to ignore any interrupts that occur.

Both of these techniques have the ability to seriously hinder the functionality of the computer. As such, most operating systems will refuse to comply with such requests unless the programs doing the request have enough privileges. However, some parts of the operating system itself are very vulnerable to interruption and the operating system itself is allowed to prevent switching and even interrupts if required. How privileges are enforced is a topic for later discussion.

Process Creation

The actual creation of a new process on a computer is very specific to the operating system itself and is often tied very closely to the hardware. We will attempt to describe the creation of processes in a high level view.

Most operating systems utilize a process table to track all of the processes currently residing on the system. The creation of a new process will usually require a new entry to be created within the process table. During the addition of the new entry, the table needs to be locked or switching needs to be prevented so that the full details can be added without interruption. Marking the state as 'New' is only part of the problem. The table itself might have issues if the scheduler sees it partially updated.

In addition to the process table entry, most operating systems keep information about the process in a separate space called the "process information block" (although this can be viewed as part of the process table). The space for this process information block needs to be set aside as part of the process creation step.

Each process will be executing some code. Part of the process creation step will be to set aside enough memory for the code, and then possibly load the code from the executable (some operating systems work a little differently as explained next).

Process Creation in Unix/Linux

In the Unix (and Linux) operating system, new processes are created by a simple system call named '*fork()*'. The term *fork* comes from the description of hitting a *fork-in-the-road*, in other words a place where a decision has to be made. It is easily shown by this diagram.

Suppose a program has the following statements:

```
Statement_A();
Statement_B();
Statement_C();
Fork();
Statement_D();
Statement_E();
Statement_F();
```

If the *fork()* call was not included it is easy to see that the program simply executes the statements one at a time. However with the inclusion of the *fork()* command, things get a bit more interesting.

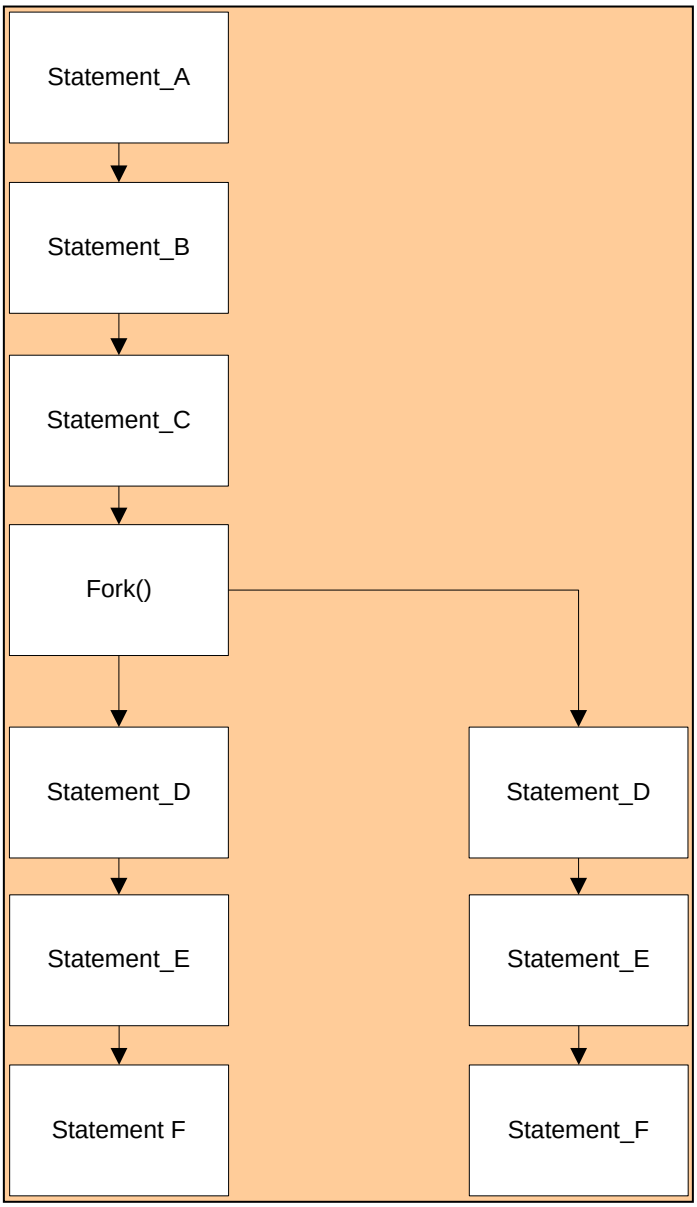


Figure 3.6
A program issuing the *fork()* command in Unix

When the system called is made at *fork()*, the operating system will actually duplicate the current process so that there are now two. This means that the original process will keep running statements D through F, but there will also be another process also running statement D through F.

When a Unix program issues the *fork()* system call, the operating system duplicates the current process exactly and once completed there are now two processes executing that look identical. However, they will be two distinct processes sharing the CPU. Any variables (memory) associated with one process will be different than the memory associated with the other process.

Who issues the fork command? Most applications that developers write would never use the fork system call unless they were trying to create a program that starts another program. In fact the *fork()* system call remains a mystery to most software developers, so if you never see it again outside of this text, that would be quite normal. When you are working in the GNOME desktop environment, you select applications from a launch menu. When you finally select OpenOffice Write, guess what happens? Yes, GNOME will issue a *fork()* system call.

If you are comfortable with the idea of the *fork()* system call, you should now ask the question: "If GNOME issues the *fork()* call, why then do we see OpenOffice Write instead of another copy of GNOME?". As soon as the new process is created the second process will replace its current executable code with new code from the drive.

The source code for launching a new application then typically looks like this:

```
Main()
{
      While (true) {
            Display prompt on screen
            Wait for user to enter command (i.e. block!)
            If command is a shell command process the command (i.e. set, echo)
            Assume the command is a program, check if the program exists.
            Fork() a new process.
            If (we are original process) {
                  Wait for new program to finish
            }
            Else { // we must be the new process
                  Load the program and run it
            }
      }
}
```

Stopping a Process

There are three typical ways that a process will terminate and release the memory back to the operating system:

1. The process can request the operating system to stop itself. This is actually accomplished when the "main" member function of most programming languages finishes. The programs that you write are actually linked with some library functions to look after making the system call for you. Generally an application will always have permission to terminate itself.

2. The process could generate an *unmanaged exception*. An exception is simply an error that has occurred that cannot be dealt with by the program. Examples of exceptions include division by zero and the process trying to perform an instruction that is not permitted by the operating system. You have likely seen the failure on Windows which states something like "This program has performed and illegal operation".

3. One process can request that another process be terminated. Most operating systems provide some form of security to prevent processes from terminating other processes unless the originating process has some sort of privilege. In Linux, the *kill* system is used to send a variety of messages including a message to terminate a target program.

Threads

Although on a CPU with a single core it is not possible to execute more than one instruction at the same time, there are often reasons why a programmer would prefer to structure a program in such a way that it is really programmed more like two or more separate programs. Take for example the program Outlook. We are all familiar that when using Outlook, you can be in the middle of creating a mail message and even while you are writing the message, a new mail messages can arrive causing Outlook to play a sound and show the arrival flag in the tool tray.

One way that the program could be organized is in the following pseudocode:

```
Forever(;;) {
      Wait for a single key
      Add the character to the message
      If there is a new message raise flag and play sound
}
```

The pseudocode above is valid but not really efficient. Notice that the algorithm only checks for new mail every time that a key is pressed. On the positive side, it is expected that the program will not be very intensive on the CPU because it will spend a great deal of time waiting for the keyboard.

In order to solve the problem of only checking after each key is pressed we could change the algorithm as follows:

```
Forever(;;) {
      Wait up to 1 second for a new key
      If a key was pressed add it to the message
      If there is new mail, raise the flag
}
```

This approach solves the problem of only checking for mail every time that the user is present, but it comes at an expense of efficiency. If a user starts to type a message and then leaves for several hours, the program will wake up every 1 second and see if there was a key pressed and check to see if there is mail. It would be more efficient to do neither if nothing is happening.

A second point about this solution is that as we add more features to our e-mail program, this main loop becomes more and more complex.

If we are able to organize the program into two tasks, each solving just one problem, then we could have the following:

```
While (!messageNotSent) {
      Wait for key
      Add character to message
}
```

And

```
Forever() {
      Sleep 10 seconds
      Check for mail
Raise flag if new mail
}
```

If both of these small algorithms can run at the same time within the same application then we have organized our program into a much easier to understand system, and in a way that adding more features can be easier.

Now that we have a model to help make the program easier to create, we need a mechanism by which a program can be organized. These two independent mini-programs have their own set of instructions that need to be executed on a single processor. This concept sounds very similar to multi-processing that was covered in the previous chapter, only in this case these mini-programs are actually part of a single large application.

In an operating system a single sequence of instructions being executed is called a *thread*. Every process will always contain at least one thread of execution unless it has been written in a special way such as what we have described for our e-mail program. In our application we have one thread whose task is to read the keyboard and compose the message, and another thread responsible for occasionally checking to see if there is new mail.

Because there are now two separate threads of execution, there will need to be the concept of *switching* and *scheduling* within our application. Luckily, a lot of operating systems look after this as part of their services, and the way they function is nearly identical to *process switching*.

In addition to making certain types of programs easier to create, a second advantage of using multiple threads arises when a processor with more than one core is used (*dual-core*, *quad-core*, etc.). If a program is created that equally divides a lot of work into four separate threads, then running the application on a quad core processor will make that application run four times faster. The art of developing programs this way is the topic of a full course.

Why threads?

Performing a quick search on the Internet or looking at most textbooks will often provide the description that a thread is a "light-weight" process. It is starting to appear that processes and threads are almost interchangeable terms. If you recall one from the section on the overview of an operating system, most operating systems have a responsibility of protecting one application from another application (if Internet Explorer crashes it should not cause Word to crash or lose data). As a result, having two processes communicate or share information is a bit complex due to the security. However, within a single application there is usually no such protection between threads. As a result, it is often very easy for two threads to cooperate on a common set of data.

If the lack of protection between threads sounds like it is a bit risky, then you are completely correct! Improperly designed applications that rely on multiple threads can result in very difficult to solve problems that only sometimes occur. With multiple threads, the scheduler is still allowed to select any thread to run that is ready and this means that if you run a multi-threaded program ten times, it might actually run ten different ways (sometimes one thread might get more time at the start). This is often why a program might work fine today, but might crash unexpectedly tomorrow. Multi-threaded programs written correctly have a great advantage but it takes the right type of programming design and testing to remove bugs.

In many life critical systems such as medical equipment or avionics systems, and even high demand systems such as telephone equipment, the design team will often avoid using more than one thread to reduce unpredictability at the cost of increasing the complexity of the code.

Inter Process Communication

The term *inter process communication* refers to information being exchanged between two processes. Although the title of the section seems to suggest that the techniques described here are only for processes, they actually work equally for multiple threads within a single process.

When two or more threads (regardless if they are in the same process or not) are trying to cooperatively work on a solution to some problem, they need some sort of mechanism to exchange information so that they can decide which part of the problem will be solved by which thread, and to ensure that they do not both try to work on a single part of the problem at the same time. There may be other situations where one thread would like to wait for the other thread to finish first before continuing on. All of these cases require some sort of communication so that the two threads can coordinate.

Synchronization

We will start with a very simple task. Suppose that we have two threads (again, if they are in the same or different processes the same problems exist) as follows:

```
threadA()
{
system.out.print("Thread A");
sysmtem.out.println();
}
threadB()
{
      system.out.print("Thread B");
      system.out.println();
}
```

Suppose we were to start the threads at exactly the same time (this is not actually possibly by the way…why?). The result should be that both messages will appear in the output window. The question that has to arise however is which message appears first? Does it look like this?

```
Thread A
Thread B
```

Does the result look like this?

```
Thread B
Thread A
```

Does the result look like this?

```
Thread A Thread B
```

Unfortunately, with the information provided the output is actually unknown! This type of solution is called *non-deterministic* because we cannot predict (or determine) what will happen every time. In fact every time that we run the program we might see a different result.

Why do we end up with so many possible results? It looks like in the first example the first thread got to run completed, then the scheduler switched to the second thread. In the last

example, the first thread got to run but was not completed (it did not make it to the println() function), then threadB ran and then threadA finished. Non-deterministic solutions are extremely dangerous in critical systems where people's lives are at stake.

Unfortunately we did not actually say what the desired output should actually look like. Let us suppose that we want the output to be in the order of "Thread A" followed by "Thread B" on the next line, and we want to be guaranteed of this order, regardless of what else possibly happens. This suggests that we need some mechanism of making sure threadB waits for threadA to finish. As always, we would like to try to minimize the amount of CPU time actually used. This first thing we will talk about is called *synchronization*.

Shared Memory

Our first technique that we will try to introduce is the concept of *shared memory*. This is a very simple to understand concept but, unfortunately, not a very good method in most cases.

The term shared memory means that both threads have access to a piece of memory that they can both read and write to at the same time. If one thread puts a value into the memory location, then when the second thread looks at the memory it will see exactly the same thing. Of course, unless the two threads are executing on a multi-core processor, there will only be a single thread executing at a given instant. Configuring a variable to be shared between two processes on an operating system is usually possible. However, it is not particularly easy due to the security. The creation of shared variables between two threads of the same process is generally easy (in fact, too easy because a lot of mistakes are generally made by assuming).

For the purpose of discussion we will assume that there is a variable called "turn" which is a simple integer that is available to both threadA and threadB. We will assume that the sharing is already set up. Now consider the following pseudocode:

```
Configure shared variable turn
Turn = 1
Start threadA
Start threadB
```

You might jump to the conclusion that threadA will be ahead of threadB because we asked it to start first. Keep in mind that the scheduler is responsible for picking the order. Just because you asked to start A first does not mean it actually had a chance to run! Now let's try to modify the two threads to make use of the 'turn' variable to make sure that threadA runs first then threadB.

```
threadA()
{
      System.out.print("Thread A");
      System.out.println();
      Turn = 2
}
threadB()
{
      While (turn == 1)
            ;
      System.out.print("Thread B");
      System.out.println();
}
```

Let us take a quick look to see if this actually does what we want. Notice that we started by setting the shared variable called turn (both A and B can see this) to the value 1. When threadA is finished it sets the value to 2. If we look at threadB() we see that it looks at the value of turn and if it is equal to 1 it does nothing, but we expect that eventually threadA will finish and the value will be set to 2, allowing threadB to execute.

This is good; this provides us a way that allows the two threads to synchronize. It does not matter if threadB gets to run for ten minutes before threadA even starts, because it will do nothing until threadA is finished. The variable called turn is an example of a *synchronization variable*.

Although this solution works, it does pose a bit of a problem. Look again at threadB, we see that it starts with a loop that continuously checks the value called turn. Now if threadB were to run for five seconds, would the value of turn ever change? Of course not, it is threadA that changes it. This type of checking is called *polling* and burns a lot of CPU time, which may consume more power from our battery (or may just cause the processor to heat up for no reason).

Self-Yield

A better option would be to have threadB look, and if the turn variable is not set then it should voluntarily go into the blocked state. Moving into the blocked state would allow the scheduler to run the other thread. We will introduce a new command called "*sleep*" which causes the current thread (and possibly process) to go into the blocked state until a certain amount of time has passed.

```
threadB()
{
      While (turn == 1)
            Thread.sleep(5000); // sleep for 5000ms or 5 seconds
      System.out.print("Thread B");
      System.out.println();
}
```

This code is much better for the CPU because now even if threadB runs first, it will immediately go into a blocked state and will wake up only every five seconds to see if the turn variable has been set. Therefore, even if threadA takes hours to complete, at least threadB is not using too much CPU time. Of course there is still a bit of a problem with this solution. Suppose that threadB executed and saw that the turn variable was still 1 and went to sleep for five seconds.

Then threadA ran and set the value turn value one second later… threadB would only wake up in another four seconds to see if the variable was set. This means that our problem has been slowed down because threadB was asleep. You may be tempted to reduce the amount of sleeping time but this means that threadB will wake up more often and use more CPU time. A better option would be to have threadA actually wake up threadB when it is finished.

Signals or Semaphores

A *signal* (or *semaphore*) is a special type of variable supporting two operations called *wait* and *raise* (when using the term semaphore the operation names are usually *take* and *give*). Because the term signal is used in Unix to mean something critical has occurred (such as a crash), we will avoid using the term signal here. However, keep in mind that many books use the terms interchangeably. Semaphores can be used in a number of ways, but the first thing that we will consider is using them for synchronization.

The two operations are:

1. *Take*: When a thread asks to take a semaphore, either the thread is given immediate control because the semaphore is available or else it goes into a blocked state waiting for the semaphore to become available. As soon as the semaphore is made available (usually by some other thread) the requesting thread will be immediately woken up.

2. *Give*: When a thread gives a semaphore it is potentially waking up another thread that is waiting.

We turn back to our example of the two threads again and consider the pseudocode to get things started:

Create a semaphore variable called "turn"
Make sure the semaphore turn is not available
Start threadA
Start threadB

Now we change the thread code slightly:

```
threadA()
{
      System.out.print("Thread A");
      System.out.println();
      Turn.give();
}
threadB()
{
      Turn.take();
      System.out.print("Thread B");
      System.out.println();
}
```

We had changed our startup code to indicate that we need to actually create the variable called turn. We have also included an instruction to make sure that the semaphore is not actually available to start. This is very important. We want to make sure that the "give" instruction in threadA is the one that makes the semaphore available.

When threadB starts it will try to take the semaphore. If the semaphore is not available, the operating system scheduler will move the thread (and possibly the process) into the blocked state. While in this blocked state, the thread (and process) will consume no CPU. When threadA finishes its tasks, it will give the semaphore and the give operation will cause the operating system scheduler to wake up the blocked thread and make it as ready.

The creation of these semaphores is very specific to each operating system, and there are often a lot of options available for dealing with very complex problems. As an example, suppose there are two threads waiting on a single semaphore. Which thread gets the semaphore when it is given? You will probably have an opinion of this immediately by saying the first thread to ask should receive it, but perhaps the second thread was actually much more important. The options provided for the semaphores can be used to control specific behavior depending on the needs of the application. Another issue that we are deferring to the section on scheduling is how long does threadB wait? In our particular example we probably want to wait until threadA is finished, regardless of how long it actually takes. In some programs waiting a really long time might not be the right answer, and often the semaphore take operations allow for an alarm clock to wake them if the semaphore is not actually given. Of course what you do when you wake up from an alarm, rather than actually receiving the semaphore, is very specific to the problem at hand. It is impossible to suggest in this text how to properly handle the situation.

Critical Sections

A *critical section* is a part of code (or more often a set of variables) that must be accessed in a controlled way when dealing with a multi-threaded system. Again, as previously mentioned, it does not really matter if the multiple threads are within the same process or spread across multiple processes. Critical code sections are actually quite difficult to visualize, so we will start with a very simple example and build up a solution technique. We will then introduce some more typical programming requirements.

Suppose again that we have two threads which we will call threadA and threadB. This time both threads are responsible for doing some long calculation (the actual calculation is not relevant), and once the calculation is finished then each thread prints some information on the screen. We will add some additional complexity to cover a new topic.

```
threadA()
{
      Long calculation
      System.out.print("Thread A is finished: ");
      For (I = 0; I < 10; i++) {
            System.out.print(i);
            System.out.print(" ");
      }
      System.out.println();
}
```

Let us assume that threadB looks identical except for the message "Thread B is finished: ". Although these examples are nonsense, they are easy to describe and introduce the concept of a critical section of code.

First we consider what happens when we run threadA without adding threadB to the mix. This thread will perform some long calculation and then print out the numbers 0 through 9 on a

single line with a space between each. The behavior of the program is really quite simple. However, if we now launch both threads at the same time we might end up with the scheduler switching back and forth several times. The result could look something like this:

Thread A is finished: 0*Thread B is finished: 0 1 2* 1 2 3 4 5 6 7 *3 4 5 6 7 8* 8 9

9

We have marked the output from threadB using italics so that you can see what is happening. The problem is that the scheduler keeps switching back and forth between the two threads while they are printing to the screen. This is quite common. Most operating systems will take an I/O request as a chance to reschedule the threads or processes. Unfortunately, there are no actual guarantees of when the rescheduling will occur. Our goal for this small program is to make sure that the output does not get mixed up. The problem is that all of the output statements in our threads are part of a critical section, which means that we do not want the other thread interfering while we are trying to write to the screen.

Preventing Rescheduling

The first technique that we consider is having the thread/process make a request to the operating system to disallow rescheduling while they are performing the output. The thread code then looks something like this:

```
threadA {
      Long calculation
      Lock Rescheduler
      System.out.print("Thread A is finished: ");
      For (I = 0; I < 10; i++) {
            System.out.print(i);
            System.out.print(" ");
      }
      System.out.println();
      Unlock Rescheduler
}
```

By putting a lock and an unlock around the print statements we are asking the operating system to no do any type of scheduling operation while these statements are being executed.

This solution is completely valid, and if the other thread does the same operation the outcome will function perfectly fine. Again as expected, this simple approach does have some downfalls. Suppose that we execute the program containing these two threads but there is another application (such as Outlook) running on the same computer. If the first thread requests that the scheduler be disabled then the other applications will not have a chance to run at all, even though they may have no interest in the screen. The problem here is that the thread has asked that no other threads, regardless of who they are or what they want, are allowed to run.

An alternative solution that might be suggested is to use a synchronization semaphore, and make sure that threadA runs to completion before threadB. This is also a valid solution. However, one should ask if that is actually a valid requirement. We have not provided any details about the length of the "long operation". Suppose that we were to implement a synchronization semaphore and always forced taskA to finish before taskB. Now suppose that the "long" calculation in taskB takes one second, but the long calculation in taskA takes 100

seconds. Using the synchronization variable means that taskB has to wait 100 seconds before it can even run. If taskB could have run after one second maybe we should have let it.

To solve the problem in the best way it seems that we should have a race between threadA and threadB to see who gets to the end of the long calculation first, and then once the first thread passes a gate we lock out the other thread. We will start by trying to use a simple variable...

```
threadA {
      Long calculation
      While (busy == 1)
      Do nothing;
      Busy = 1;
      System.out.print("Thread A is finished: ");
      For (I = 0; I < 10; i++) {
            System.out.print(i);
            System.out.print(" ");
      }
      System.out.println();
Busy = 0
}
```

Here we assume that a simple integer variable called busy has been created and initialized to 0 before starting. Both threadA and threadB look nearly identical with the exception of the printed message. At first glance this code looks like it is going to solve our problem. The first thread to finish the long calculation will see that the busy variable is zero and will then set it to 1. If the second thread comes along it will see that the busy flag is 1 and will wait for the other thread to clear it. There are two problems with this solution. The first problem is the while loop which is a form of busy waiting, this burns CPU time. The second problem which is much more important is that the solution does not actually work.

Suppose that threadA checks the busy flag and sees that it is zero. This means that the next instruction is the "busy = 1". But lets suppose that just as threadA is about to change the value to 1, that the scheduler causes threadB to execute and threadB checks the value of busy. The value of busy is still 0 because nobody has set it. The result now is that both threads are executing within their critical section and you will end up with a messed up output. It looks like there is actually another critical section between the value of the flag checking and the setting.

Let us look again at the use of a semaphore, because it allowed one task to block (without consuming CPU) until another thread gave the semaphore. However, this time we are going to initialize the semaphore just a little bit differently. Here is the pseudocode that starts everything:

```
Main()
{
      Create semaphore called lock
      Make sure that the lock is available
      Start ThreadA
      Start threadB
}
```

Now for the thread code, again threadB looks identical except for the message displayed.

```
threadA {
      Long calculation
      Lock.take();
      System.out.print("Thread A is finished: ");
      For (I = 0; I < 10; i++) {
            System.out.print(i);
            System.out.print(" ");
      }
      System.out.println();
Lock.give();
}
```

In this case the first thread through the long calculation will take the semaphore, and the second thread to complete will have to wait because the first thread already took it. The operating system code would have ensured that if there were critical sections in the take and give code, they would be protected against switches (likely by disabling interrupts). As soon as the first thread gets through the critical section, the semaphore lock is given and the other thread would be woken up and would execute.

This example of a program that does some calculations and prints some numbers to the screen is just meant to be an easy to understand example. We now consider a real problem involved with our e-mail program.

To set this up, let us suppose that an individual e-mail message is stored in a Java object of type EmailMessage. Our e-mail program keeps the e-mail messages in a simple array of EmailMessage, and there is an integer variable called inboxCount that keeps track of how many messages are actually in the inbox.

```
EmailMessage inBox[1000];
Int inboxCount;
```

Suppose that the software developer responsible for the e-mail program has decided that if there are 10 messages in the inbox, they will be stored in the array in locations 0 through 9. New messages that arrive will always be placed in the very last position and the inboxCount will be increased by 1 and that the developer has created the following member function to handle new messages:

```
Void newMessage(EmailMessage msg)
{
      inbox[inboxCount] = msg;
      inboxCount++;
}
```

So we put the new message into the array at the current count position and increment the counter by one for the next message that comes in.

Now suppose that the software developer has decided that there needs to be a function that removes a single message from the inbox given the "index" of the message. This code would likely look something like this:

```
Void deleteMessage(int index)
{
      Delete inbox[index];
      For (int I = index; I < inboxCount; i++)
            Inbox[i] = inbox[i+1];
      inboxCount--;
}
```

The function operates by releasing all of the information for the message at the given location then shifts all the messages to the left to fill in the gap. It then finishes by reducing the number of messages by one.

Now suppose that the software developer decides that new messages coming in should be filled in by a separate thread than the thread interacting with the user in order to make programming easier.

Unfortunately the above code may lead to some difficulties with multiple threads. Suppose that a user is trying to delete a message and is part way through the deleteMessage function. Just before the inboxCount is about to be reduced a new message arrives, and the newMessage() function is called. The fact that both the variables inbox and inboxCount are being updated by two separate threads creates what is called a *race condition*. A race condition is simply a sequence of code executed by two or more threads in which the end result depends on which thread finishes first. In this case if either completely finishes, then it is not a race condition. However, if one of the threads is interrupted during the critical section, the results may not be as expected.

This example should show that although simple code solutions work, when more than a single thread is introduced into the solution there is a risk when it comes to shared variables.

Message Queues

A popular mechanism provided by many operating systems for the purpose of inter process communication is the concept of a *message queue*. In many multi-threaded programs there is often a lot of information that one thread will want to exchange with another thread rather than just doing synchronization.

In order to send actual data from one thread to another, most operating systems provide a mechanism known as a message queue or a *pipe*. In a message queue there is always at least one sender and one receiver. The sender creates a message and puts it into a queue for delivery by the operating system. The receiver listens to the queue for messages and will often block waiting for messages to arrive. Once the message arrives, the receiver is woken up and the message is processed.

As an example we will slightly reconsider our e-mail program. Suppose that we have a single thread that is responsible for adding and removing e-mail from the inbox. We have seen that by having only one thread we remove critical sections. Is it possible in this type of solution to still have a separate thread that checks for new messages? Yes! A solution is to configure a message queue between the checking thread and the inbox manager thread as follows:

Figure 3.7
Configuring a message queue

The main message processor will accept messages either a receiving thread or from a user interface thread. The processing thread can only process one single message at a time and its pseudocode would look like this:

```
Forever(;;) {
      Wait for message from queue
      If (message is delete)
            Call delete function
      Else if (message is new)
            Call new message function
}
```

The message queues in many operating systems will allow for the listener to time-out if that is important for the program being developed. Most operating systems send data across their message queues in a first-in-first-out (FIFO) arrangement, and some allow for queue jumping so that important messages can be inserted at the very front of the list.

Most operating system message queues provide for a "backlog" of messages. If a sender quickly sends ten messages, the receiver must request each of the messages to remove them from the queue. This leads to a significant problem in that if the receiver is not able to process the messages fast enough, you can end up with some strange results. Most people have probably used a computer at one time that was not responding even though you were typing. After a few seconds, the program suddenly responded and all of the characters typed appeared instantly. This is an example where the typed characters were put into a message queue but the receiver (the program) was busy doing other tasks.

Process Scheduling Algorithms

In this section we will describe the common algorithms that are used for scheduling processes and threads. Here the word *scheduling* will refer to the task of deciding which process or thread should be selected from the process table and put into the Run state.

Most current operating systems actually implement thread scheduling so that it does not matter if the two threads are in the same process or different processes. In this section we will try to use only the term "*process scheduling*." However, if the term thread is mentioned it really does mean the same thing.

Scheduler and the Process Table

The *scheduler* is a piece of software code (it is generally a function) that is included as part of the operating system. The scheduler is often tied to the periodic clock interrupt and to most of the operating system calls. By connecting to both a clock and the I/O, it means that processes can be scheduled each time there is a clock tick and they can be scheduled each time that the process talks to the OS (such as when it asks to take a semaphore, asks to sleep, or performs an I/O operation).

Each time that the scheduler code is executed, the code will examine the process table and make a decision (based on what it reads from the process table) about which process should be run next. Over the next few sections we may realize that some additional items are required to be added to the process table to help the scheduler.

Round-Robin Scheduling

If a computer has four processes currently running, it would seem logical that one of the fairest ways to divide the CPU time is to give each process a turn in a certain order and continue to use that order. For example if the processes are called A, B, C, and D then the order could be: A, B, C, D, A, B, C, D, A, B, etc. This type of scheduling is called *round-robin scheduling*. Most modern operating systems provide this type of scheduling algorithms and it is one of the easiest algorithms to understand.

A common way of showing the algorithm is to create a small table that looks like this:

Process	T1	T2	T3	T4	T5	T6	T7	T8	T9
A	■				■				■
B		■				■			
C			■				■		
D				■				■	

The labels T1, T2, etc. refer to small units of time called *time slices* or *quantums*. The actual number of seconds (or more likely milliseconds) is dependent on the operating system but can sometimes be configured. Typical values for the time slices are in the order of 10ms.

Each process is given a number of milliseconds to execute and then interrupted, and the scheduling algorithm will pick the next process in the list. However, if the process blocks (waits for a semaphore or sleeps) the scheduler would wake up early and schedule the next process. While a process is in the blocked state it will be not considered until it gets back into the ready state.

The size of the time slice is quite important. If you pick a really large time slice (say ten seconds), then it means that each process gets to execute for up to ten seconds. If Windows used a ten second time slice, then it means when you click on the start menu (part of the Explorer process) you might have to wait up to ten seconds for Word to finish what it is doing before seeing the menu appear. On the other extreme, if you set the time slice to a really small value (such as 1 ms) then it means that the scheduler is executing 1,000 times per second. When the scheduler code is busy selecting a process it means that no other process can be running. The result is an inefficient system that spends more time trying to decide what to do than actually doing something.

Priority-Based Scheduling

The round-robin type scheduling appears to be a very fair algorithm, and in most cases the use of this algorithm works well. However, there are some situations where another form of scheduling leads to a better system from the user's point of view.

Most people have probably used a computer at one time when the computer did not seem to be responding. While using MS Word you may find yourself typing a few keys but nothing appears on the screen for a few seconds, then suddenly all of the characters you typed appear at the same time. Although it might be okay if this happens just once a month, it should not be happening every few minutes. While the user is typing they generally like to have some sort of instant feedback so they know their keys are being accepted. If it took five seconds for your phone to show the digit you pressed when dialing a number, you would find this quite unacceptable.

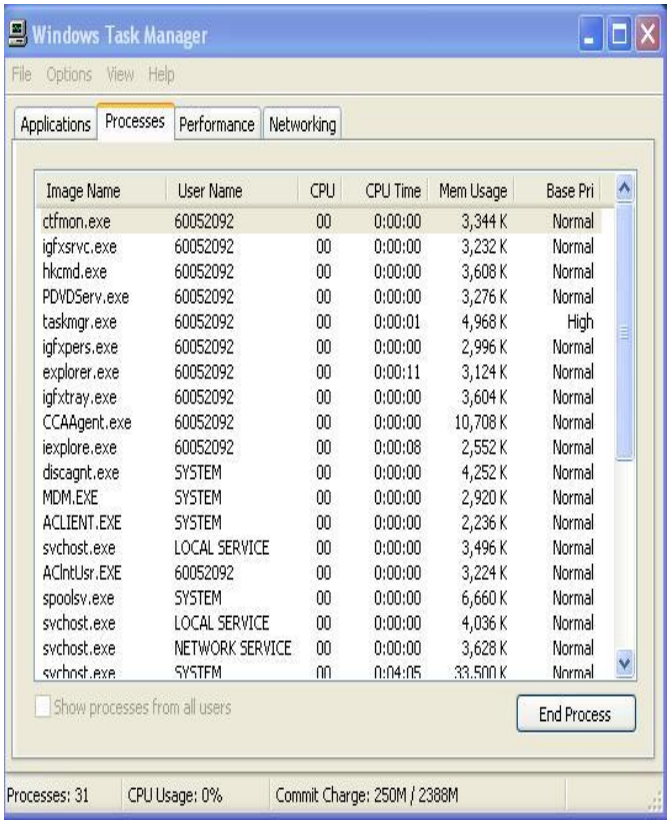


Figure 3.8
Windows Task Manager Showing Process Priorities

Unfortunately, with round-robin scheduling instant feedback is not possible unless we choose a very small time slice and we do not have a lot of processes. While your computer (or mobile) is running another process, we need some way of having the user application interrupt other applications. In order to keep track of which processes are more important, the process table generally contains an entry called the *priority*. Each time the scheduling algorithm is executed, the scheduler will look at the process table for the process that is in the Ready state and pick the process with the highest priority.

An operating system could allow the process itself to pick its own priority, or the priority could be picked by the operating system. In some operating systems, priorities are simply low, medium or high. In other operating systems, the priorities may be arranged between 0 and 255 with 0 being the highest and 255 being the lowest (the meanings could be reversed with a larger number being higher in priority). What do we do if there are two processes with the same priority? The usual implementation is to utilize round-robin scheduling for all processes that are of equal priority.

When a process is put into the Run state, it will get to run until one of the following things happens:

1. it finishes,
2. it blocks,
3. a higher priority process becomes ready.

What if the process never blocks or finishes? Then the process gets to monopolize the CPU and never allows any other process to run. When a process never gets to run because a high priority process is using all of the time, it is called *starvation*. Starvation is a danger of any operating system relying on priorities for scheduling and it requires careful programming to ensure that it does not happen. You may be wondering how a higher priority process would ever become ready if another process is running all of the time. As long as the currently running process eventually waits or gives a semaphore that the higher priority process is waiting for, then the high priority process gets to run.

In most operating systems, processes started by the user (application) are assigned the same priority, and generally the processes created by the operating system are provided a higher priority so that the developers do no need to worry about this aspect. In the Windows Task manager, it is possible to change the priority of the processes. Linux also allows for the user to lower the priority of a process.

Case Study: Priority Does Not Mean Importance

A nuclear power plant produces power by having a nuclear reaction heat up water which generates steam and the steam is used to turn a turbine. To slow down the reaction the plant can insert control rods into the reactor. If the reactor gets too hot, you might cause a nuclear meltdown (you do not want this to happen!). Putting in the control rods is usually done by a robot and takes a certain amount of time.

Suppose there is a sensor on the reactor that notifies a controlling computer when the temperature gets too hot. Also let us pretend that the computer program to move the control rods into place takes 30 seconds to complete, but if you do not get the task done in 60 seconds there is a meltdown.

Meanwhile, inside the break room at the power plant is a coffee maker. The coffee maker has a sensor which can detect when the pot is about to overflow, and it sends a signal to a computer program which is responsible for switching off the pot. It takes one second for the computer to switch off the pot, but if you do not switch it off within five seconds the pot will overflow.

Due to cost cutting, the owner of the nuclear power plant has decided to buy only one computer to control both the reactor rods and the coffee pot! You of course have two processes running,

one for the coffee and the other for the reactor. Which process should have the higher priority? We will assume that both alarms happen at the same time.

Scenario One: The Control Rods get Higher Priority

This makes sense because preventing a meltdown is a lot more important!

1. Time = 0s. Both alarms trigger. Control rod robot activated, ignoring coffee pot.
2. Time = 5s. Coffee pot starts overflowing.
3. Time = 30s. Control rods in place. Lots of water on the floor. Coffee pot is instructed to switch off.
4. Time = 31s. Coffee pot is off.

Is this a good solution? The plant did not meltdown, therefore this is good. The coffeepot overflowed which means there is a mess in the break room, but we only need a mop.

Scenario Two: The Coffee Pot gets Higher Priority

This sounds like a silly idea but consider the sequence of events.

1. Time = 0s. Both alarms sound. Coffee pot starts to be switched off.
2. Time = 1s. Coffee pot is switched off. Reactor rod robot started.
3. Time = 31s. Reactor rods inserted and meltdown prevented.

In this solution, the control rods were inserted well within the required time and we also avoided the flooded break room.

Conclusion

Processes with important responsibilities do not always need the highest priority. The selection of priorities is based a combination of how quickly a process must react and how long it will take to complete.

Other Scheduling Algorithms

There are a number of other scheduling algorithms that are described in various texts, but they are often based on older operating systems where programs ran without a user sitting in front of the computer. We include a brief description of each of these for completeness.

First Come First Serve

In this algorithm, there is no pre-emptive switching. When a process is scheduled it runs until it is completely finished (or perhaps until it blocks). The idea is quite simple once a process is started the most efficient thing to do is to let it run until is completely finished before allowing another process to run.

This scheduling algorithm does not work for situations where two processes are working together to solve some problem however from a purely efficiency point of view it works well and there is no fear of having one process being interrupted by any other process.

In a system running this type of scheduling the process table is created with a list of all the processes and then run without modification until all tasks are done (or new tasks added only between processes).

Shortest Task Remaining

The idea for this algorithm is to figure out which process will finish in the shorted amount of time and schedule the processes in an order that finishes as quickly as possible. It tends to favor short processes over longer processes. The danger is that a lot of short processes could easily starve a longer process. It is also not practical because you cannot actually know for certain how much more time is remaining for a given process.

Scheduling and Blocking

The scheduling algorithms presented earlier in this section have completely ignored tasks which are currently blocked. We finish the section on scheduling by looking at one particular problem of blocking from a scheduling point of view.

Suppose we have the following processes:

Process A (priority high)
Wait for 10 seconds
Take semaphore S (block until it is available)
Do calculation

Process B (priority medium)
Sleep 5 seconds
Do calculation that takes 3000 seconds

Process C (low priority)
Take semaphore S
Do calculation that takes 10 seconds
Give semaphore S

Suppose that all three processes are started at exactly the same time and that the semaphore S is available to the first process that requests it.

As soon as all processes are started the process table will look like this:

Name	Priority	State
Process A	High	Ready
Process B	Medium	Ready
Process C	Low	Ready

This means that the scheduler will select process A because it is the highest process that is 'Ready'. Process A will execute, but the first thing that it does is sleep. This means that its state will become 'Blocked'.

Now the scheduler will wake up and select Process B because it is the highest process that is ready to run, but as soon as Process B runs its state turns to block because of the Sleep. Next, process C gets a chance to run, and it takes the semaphore S and starts doing some calculations. Process C does not block because it is running some long calculation.

After five seconds, Process B wakes up and runs its 3000 second calculation. Remember that process C is still 'Ready' because it has another five seconds to go and it is also holding semaphore S.

At ten seconds, process A wakes up from its sleep but immediately goes into a blocked state because it requests semaphore S (but C has it). The scheduler then runs process B because it is the process with the highest priority that is available.

The problem here is that Process A has the highest priority and wants to run, but it cannot because Process C has a semaphore it needs. But Process C is not able to run because Process B is monopolizing the CPU. This problem is called *priority inversion* because the medium priority process is preventing the high priority process from running.

Solutions to this problem are not covered in this text, but two observations are noted:

1. There is a shared semaphore between a high and low priority task; this is generally a bad idea.

2. The task in the middle has a high priority but it monopolizes the CPU for a relatively long period of time. Such a lengthy task should probably have had a lower priority.

Unit Summary

Processes are instances of programs that are currently being run in a computer system. In order to improve the efficiency of process execution and CPU scheduling, processes are often broken down into smaller units called threads. Processes and threads are managed in by the operating system using a Process Table, which lists all processes and threads and their current state. A process or thread can either be running, ready to run, or blocked (which means that they will be ignored by the process scheduler until their state has been changed back to "Ready"). Processes can also be marked in the Process Table as either "new" or "exit," which is a form of blocking preventing them from being executed because they are not actually ready.

A number of different strategies are used to allow processes and threads to exchange information so that they can synchronize themselves, or arrange themselves to be scheduled only in the correct sequence (or when specific required information is available). The most common methods of inter process communication include the use of signals (often called semaphores) and message queues. When exchanging information between processes or threads, certain critical sections of code most be handled carefully to make sure that they are not corrupted, that they are accessible when needed by other processes or threads, and that the desired outcome is achieved by the instructions being executed.

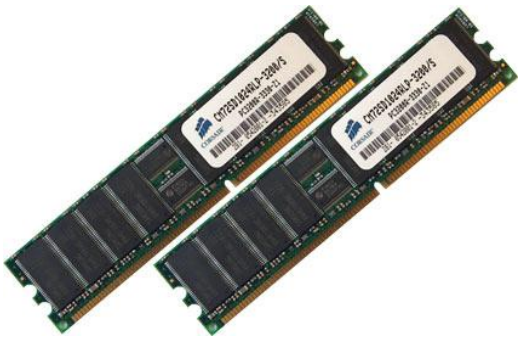
The scheduling of processes can be handled using a variety of algorithms, but the most common methods are to handle all processes or threads in sequence (round-robin scheduling), or to schedule based on process priority (priority-based scheduling).

Key Terms

Algorithm
Blocked
Blocking
Critical section
Dispatch
Dual-core
Exit
First come first serve
fork()
Give
Inter process communication
Interrupt
Message queues
Multicore
Multiprocessor
Multitasking
Multithreading
New
Non-deterministic solution
Non-preemptive switching
Pipe
Polling
Preemptive switching
Priority
Priority inversion
Priority-based scheduling
Process
Process scheduling
Process table
Processor preservation
Quad-core
Quantum
Race condition
Raise
Ready
Round-robin scheduling
Running
Scheduler
Self-yield
Semaphore
Shared memory
Shortest task remaining
Signal
Sleep
Starvation
State changing
State machine
Synchronization
Synchronization variable
Take
Thread
Time slice
Unmanaged exception
Variable
Wait

Unit 4: Memory Management

What is Memory Management?



Memory is vital to the functioning of any computer or computerized device. As discussed in Units 1 and 2, memory consists primarily of RAM chips that are installed on the motherboard of a computer. A typical computer now has between 1 GB and 4 GB of RAM. RAM is divided into segments (typically 32 bits) that are used to temporarily store data and instructions that are being used by the computer. This is different from your hard drive, which permanently stores files and programs. RAM is much faster than your hard drive, so your data and instructions are loaded from your hard drive into RAM when the computer is using them. When you turn off the power, everything stored in RAM is gone. In addition to the physical RAM installed in your computer, most modern operating systems allow your computer to use a *virtual memory* system. Virtual memory allows your computer to use part of a permanent storage device (such as a hard disk) as extra memory.

Memory resources are managed by the operating system. The operating system is responsible for allocating memory address ranges as needed to run applications and processes. In this unit, we will look at the function of the memory manager in an operating system, and the types of problems that the memory manager must resolve. We will look at some of the techniques that operating systems can use to actually allocate memory, and some of the problems that can occur when using specific memory allocation strategies. We will also look at the purpose of virtual memory and how it works. This will include a look at page files, the page table, and the page replacement policies used by operating systems to manage virtual memory.

The Memory Manager

The *memory manager* is part of the kernel (core) of the operating system. It is responsible for efficiently managing all memory resources including RAM and virtual memory, and allocating memory space to applications and processes as needed. The memory manager must keep track of all memory addresses and what they are currently being used for. It must also protect those address ranges, and the data and instructions stored in them, from data and instructions being used by other processes. The memory manager is also responsible for freeing up memory when it is no longer being used so that other processes can use it.

The memory manager is responsible for four critical tasks:

1. Allocating *main memory* to processes
2. Retrieving and storing the contents to and from main memory when requested
3. Effective sharing of main memory
4. Minimizing memory *access time*

Efficient Memory Management

A well designed operating system should make access and the management of resources as efficient as possible for both the user and the system. Efficient memory management includes keeping track of all memory resources that have been allocated to different processes. It also includes using different strategies for allocating free memory as it is needed by new processes.

When it comes to the allocation of memory spaces and the retrieval of data from memory, you can compare the memory manager to the local telephone company. There are many ways that the phone company can keep track of telephone numbers that have been assigned to customers. One way is to keep a sequential list of all phone numbers, along with the names of the subscribers. Another way is to keep track of the names of subscribers, along with their assigned phone numbers (like a phone book). An efficient memory manager would use both types of strategies to keep track of data stored in RAM. The first strategy is useful for examining address ranges and finding free memory that is available for use. The second strategy is useful for examining individual processes, and keeping track of what memory each process is using.

An operating system's memory manager is also responsible for using different strategies to allocate free memory to new processes. Each strategy has its advantages and disadvantages. The aim of the operating system is to use the most effective strategy to both minimize the time needed to save and access data, and to maximize the amount of usable space left in memory. You can compare this task to parking vehicles in a parking lot. For example, you could park a motorcycle in the first available space. If it is a very big space, then you will end up reducing the maximum number of vehicles you can park in the lot, because the rest of the space may not be quite big enough to fit another vehicle. Conversely, you could try looking for the smallest possible parking space where the motorcycle will fit. The drawback here is that it might take you longer to find such a spot.

There are three main strategies that the memory manager can use when allocating free memory to processes:

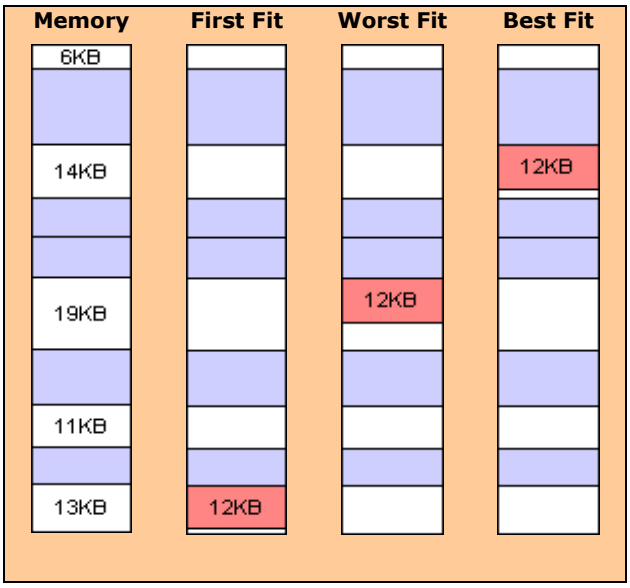


Figure 4.1
Using memory allocation strategies to assign 12KB of memory

- Best Fit
  - Find the smallest free memory block that will fit the process needs
  - Idea is minimize wastage of free memory space
- Worst Fit
  - Find the largest free memory block that will fit the process needs
  - Idea is to increase the possibility that another process can use the left-over space
- First Fit
  - Find the first space to fit the memory needs
  - Minimize the time to analyze the memory space available

Fragmentation

The use of particular memory allocation strategies may result in wasting valuable free memory. That is because using either the best fit or first fit strategies may leave small chunks of free memory that are not large enough to be useful for any other processes (like parking a motorcycle in a parking space, and not having enough space left for another vehicle). The allocation and de-allocation of memory creates a condition called *fragmentation*. This means that there are lots of small fragments (or holes) of free memory that cannot be used by any other process. This results in a reduction in the amount of total available memory.

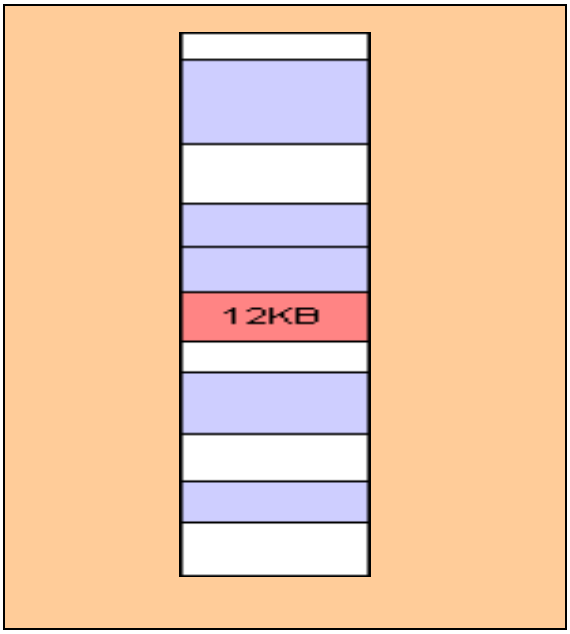


Figure 4.2
The worst fit strategy assigns the largest free memory chunk so that it leaves behind the largest possible fragment

The *worst fit* memory allocation strategy attempts to minimize fragmentation by finding the largest chunk of free memory available to allocate to a process. By doing this, the memory manager tries to ensure that the unused memory "holes" it leaves are as large as possible. The goal is to leave as many big chunks of memory as possible so that as many fragments as possible are large enough to be used by other processes.

Relocation

Another task that must be handled by the Memory Manager is the *relocation* of applications in memory. Applications have certain memory requirements when they are loaded. However, it is not possible for the application to know in advance which memory range it will be allocated, because the application itself does not know what other processes will be running, how much physical memory the system will have available to it, and whether or not some of its required memory will be comprised of *virtual memory* pages (see next topic). For this reason, the application will only make reference to "*relative*" or "*logical*" address ranges. When the application is compiled to load a process into memory, the operating system will allocate the actual address range based upon the relative ranges provided by the application, and the actual memory resources available. As demonstrated in Figure 4.3 (right), the process that is loaded will provide a *logical address*, which is combined with a *base address* provided by the operating system, to determine its actual location in memory. The application itself will be unaware of its location in memory. However, once a process has been started, it is not possible for the Memory Manager to relocate that process in memory unless it is aware of the relocation, and the new base address.

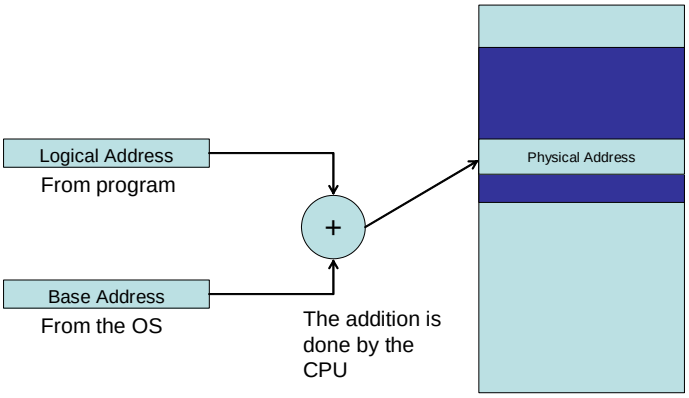


Figure 4.3
Allocation of Physical Memory

Virtual Memory

Despite the fact that most modern computers have between 1 GB and 4 GB of RAM, it is possible to run out of memory. This can happen when you are running too many processes at the same time, or when a process requires more memory than is currently available (unused). One solution to this problem is to use *virtual memory*.

Virtual memory means that the operating system's memory manager will use a portion of another storage device to act as if it is extra RAM. In most cases, this virtual memory space will be on a permanent storage device such as a hard disk. However, some newer operating systems (such as Windows Vista and Windows 7) also allow you to use a removable media device like a USB flash drive as virtual memory (called *Windows Ready Boost*). The memory manager will manage this resource so that it gives the illusion that your computer has more RAM than is actually installed.

How Virtual Memory Works

Virtual memory works by breaking down large programs into smaller units called *pages*. The memory manager will store these pages on an area of the storage device (hard disk) that has been allocated for use as virtual memory. This area is called a *Page File* (or Paging File).

All of the pages that are currently loaded in *main memory* (RAM) will be listed in a *Page Table*, which is maintained by the memory manager. If a process requests a page that is not currently loaded into RAM (in the Page Table), then a *page fault* will be generated. The memory manager will attempt to resolve the page fault by locating the requested page in virtual memory and then loading it into RAM (and listing it in the Page File). The memory manager will then try to re-execute the instruction that caused the fault.

Pages, Virtual Addresses and Physical Memory

Figure 4.3 (right) shows how pages, virtual addresses and physical memory are combined to create a virtual memory system. The memory manager maintains both a virtual memory space and a Page Table. All processes and data that are loaded into either main memory or the page file (hard disk) are given addresses in the *virtual memory space*. The Page Table keeps track of the locations of all the pages. Although pages may be located on different physical devices, the memory manager treats them as if they are all contained in one large memory system. When a page is actually needed for processing, it is loaded into main memory (RAM). Older pages that are not currently being processed may be *swapped* from RAM back to the paging device (usually a hard disk).

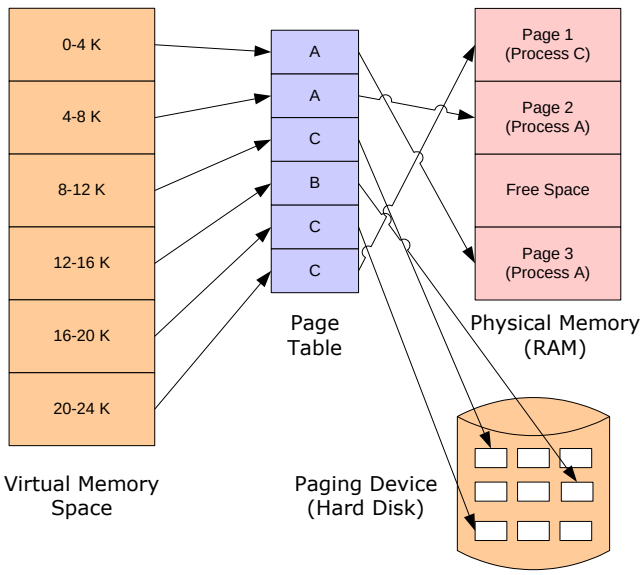


Figure 4.4
Combining Elements to Create a Virtual Memory System

Swapping Pages

Whenever main memory (RAM) is full, or when a page is requested that is located in virtual memory, the memory manager needs to *swap* old or unused pages from RAM into virtual memory in order to make space for the new pages being loaded. There are a number of strategies used by the memory manager to handle this task. These strategies are often called *replacement policies*. There are five main replacement policies that are used by operating systems:

- Random Replacement
- First In First Out (FIFO)
- Second Chance
- Least Recently used
- Least Frequently Used

Each of these replacement policies uses different algorithms for selecting pages to be swapped from RAM into the page file.

Random Replacement
- Replace pages in main memory randomly.
- On the average, does not work well.

FIFO
- Uses a queue data structure to keep track of the pages in main memory.
- Oldest page at the front (head) and newest page at the back (tail).
- Always replace (get rid of) the oldest page.
- Does not always work, because the oldest page may still be used by the process.

Second Chance
- Another version of FIFO to address the problem of FIFO.
- All pages in the page table are tracked to see if they have been referenced recently by a process.
- A Reference (R) bit for each page is used for this purpose:
    - R = 1 when the page is being referenced.
    - R = 0 when the page has not been used after a time period.
    - The OS will periodically check the (R) bit for each page and move the pages those (R) = 1 to the tail of the queue thus given a 2^{nd} chance.

Least recently Used (LRU)
- Replace the page in main memory that has not been used the longest.

Least Frequently Used (LFU)
- Counter are used to record the number of times each page have been used.
- The pages that have been used the least (lowest count) would be replaced.

Hit Ratios: Determining Which Replacement Policy to Use

Using each of the five main replacement policies to swap pages between main memory and the page file will result in different hit ratios. A *hit ratio* is the number of times that a page is actually found in main memory, as opposed to the number of page faults generated (requiring the memory manager to retrieve the page from virtual memory). To calculate the hit ratio, we divide the number of non page faults by the total number of page requests (the number of times that data has been sent between the CPU and main memory). Remember—page faults occur when a page is requested by the CPU that is not currently in the page table (in other words, not currently in RAM). The memory manager resolves the page fault be swapping an old page from RAM to the page file (on the hard disk), then retrieving the requested page from virtual memory to RAM, and then re-executing the instruction. Since there are extra steps involved in order to execute the instruction, and since retrieving a page from a hard disk is slower than RAM, page faults result in longer processing time. We use hit ratios to determine which replacement policy will result in the fewest number of page faults, and the fastest overall processing time. The policy that produces the lowest number of page faults is usually the policy we want to use.

Calculating Hit Ratios

To understand how to calculate hit ratios, we will examine an example that uses a RAM space of three (3) frames, and the following processing sequence:

1 2 1 3 1 4 1 5 2 3 2 4 1 5

In these examples, Y = a hit (the page is found in RAM), and N = a page fault (the page must be retrieved from virtual memory. A frame is a segment of RAM that can be allocated to hold a page, and is typically the same size as a page. Frames 1-3 are located in RAM. Frames 4-5 represent pages located in virtual memory.

Using FIFO:

Sequence	1	2	1	3	1	4	1	5	2	3	2	4	1	5
Hit?	N	N	Y	N	Y	N	N	N	N	N	Y	N	N	N
Frame 1	1		1		1	~~1~~ 4			~~4~~ 2		2		~~2~~ 1	
Frame 2		2					~~2~~ 1			~~1~~ 3				~~3~~ 5
Frame 3				3				~~3~~ 5				~~5~~ 4		
Hit Ratio	3/14													

Using the *First In First Out* (*FIFO*) policy, we end up with a total of only three hits out of fourteen attempts to send information between the CPU and RAM.

Using LRU:

Sequence	1	2	1	3	1	4	1	5	2	3	2	4	1	5
Hit?	N	N	Y	N	Y	N	Y	N	N	N	Y	N	N	N
Frame 1	1		1		1		1			~~1~~ 3				~~3~~ 5
Frame 2		2				~~2~~ 4			4 2		2		~~2~~ 1	
Frame 3				3				~~3~~ 5				~~5~~ 4		
Hit Ratio	4/14													

Using the *Least Recently Used* (*LRU*) replacement policy, we end up with a total of four hits out of fourteen attempts to send information between the CPU and RAM.

Using LFU:

Sequence	1	2	1	3	1	4	1	5	2	3	2	4	1	5
Hit?	N	N	Y	N	Y	N	Y	N	N	N	Y	N	Y	N
Frame 1	1		1		1		1						1	
Frame 2		2				~~2~~ 4			4 2		2			
Frame 3				3				~~3~~ 5		~~5~~ 3		~~5~~ 4		4 5
Count	1	1	2	1	3	1	4	1	1	1	2	1	5	1
Hit Ratio	5/14													

Using the *Least Frequently Used* (*LFU*) replacement policy, we end up with a total of five hits out of fourteen attempts to exchange information between the CPU and RAM. In this example, the LFU replacement policy has resulted in the highest hit ratio, and the least number of page faults. That means that using the LFU replacement policy will result in the least amount of times swapping pages between RAM and the page file, and the lowest overall processing time.

Thrashing in Virtual Memory

The use of virtual memory has both benefits and drawbacks. The main benefit is that it artificially increases the overall amount of memory available for the execution of processes. However, as we have seen, swapping pages between RAM and virtual memory slows down the overall processing time. It takes much more time to locate and retrieve data from a hard disk than it does from RAM. When your computer spends too much time swapping pages between RAM and the page file, we call this condition *thrashing.* Thrashing not only results in longer processing time, it also leads to increased wear and tear on your hard disk. Normally, your computer only reads data from your hard disk when you are loading it into memory to be used. Your computer normally only writes data to your hard disk when you are finished with it, and you want to save it.

What Causes Thrashing?

Thrashing is typically caused when you have too many processes running at the same time. All of these processes will compete for the limited amount of physical memory installed in your computer. If your computer is thrashing, your applications may stop responding (or run very slowly). At the same time, you may notice your hard drive light blinking (and you may hear the hard drive spinning). There are really only two ways to correct thrashing:

1. Kill (stop) some of the processes (temporary solution)

2. Install more RAM

Quick Tip:

You can change the size of your operating system's page file (virtual memory). How you do this depends on the specific operating system. However, your page file should never be bigger than 1.5 times the size of the amount of RAM installed in your computer. If it is bigger than that, your memory manager will rely too much on your virtual memory, and you will experience thrashing.

Unit Summary

The memory manager has the responsibility of allocating memory resources to processes and threads as needed. This task includes assigning memory as needed and retrieving information from memory when it is needed by the CPU. The memory manager also has the responsibility for managing available memory as efficiently as possible to make sure that as much memory as possible is actually usable by processes, and to make sure that processes can be executed as quickly as possible. The three main strategies used to assign memory spaces are called Best Fit, Worst Fit and First Fit. Most modern operating systems also use virtual memory, which uses part of another storage device to act as extra RAM. The memory manager must handle swapping pages (chucks of data and process code) between main memory (RAM) and virtual memory (storage device). The page table is used to keep track of the location of pages in RAM, and if a page is requested that is not located in RAM a page fault will be generated. The memory manager must find the requested page in virtual memory, load it into RAM, and re-execute the instruction. There are five main strategies (called policies) that are used the manage page swapping between RAM and virtual memory. The memory manager must use the most appropriate strategy to minimize the number of page faults that are generated, thus minimizing the overall time needed to execute a process. When too many processes are competing for main memory, and too many pages are being swapped between RAM and virtual memory, a condition called thrashing is created. Thrashing can cause undue wear and tear on a hard disk, and increases the amount of time needed to execute an instruction set. The memory manager tries to minimize thrashing, but the only real solutions are to either reduce the number of running processes or add more RAM to the system.

Unit 5: Input/Output

What are Input/Output Resources?

Input/Output devices were briefly discussed in Unit 1: Architecture Review. Input devices are any devices that allow data to be input into a computer system. The most common examples are the keyboard and mouse, although there are many others. Output devices are any devices to which the computer can send output data, such as the monitor or printer. These I/O devices are connected to the CPU by a series of system busses on the motherboard. The operating system is responsible for issuing commands to I/O devices, as well as handling all interrupts and errors generated by the devices. The operating system needs some way to efficiently manage all of these devices and the flow of data coming in from them, or going out to them. These responsibilities are complicated by the fact that many processes being executed by the operating system may need to share the same *I/O resources*. In this unit, we will examine I/O resource management from two perspectives.

First, we will look at I/O resource management from a hardware management perspective. We will examine the role of *device controllers*, as well as the differences between *preemptable* and *non-preemptable I/O resources*, and *block* and *character I/O devices*. Later in the unit, we will take a detailed look at the actual management of one of the most common I/O hardware devices in any computer system: magnetic storage devices (hard disks and floppy disks).

Common Input Devices

Common Output Device

We will also be examining I/O resource management from a software perspective. We will take a look at the role of I/O management software, and I/O software system layers. This will include an examination of the software operating at each layer. In particular, we will focus on the role and structure of device drivers.

We will conclude this unit with a brief look at the role of the system *clock* in the management of I/O resources.

I/O Resources

Input/Output resources are any I/O devices (and their supporting hardware and software components) that are available for use by processes being executed by the operating system. These resources are frequently shared between processes, so the operating system must have some way to regulate access to the resources to prevent conflicts and *deadlocks*. The operating system uses device controllers to handle communications with I/O devices. I/O resources can be categorized as either preemptable or non-preemptable. In addition to regulating access to I/O resources, the operating system must also control how data is transmitted to and from I/O devices. Data transmission can be handled as either character or block transmission, depended upon the type of I/O device.

Device Controllers

Device controllers are components on the motherboard (or on expansion cards) that act as an interface between the CPU and the actual device. The operating system actually controls the device by using device controllers, which interpret the commands being issued to the device. Instructions sent to the device are compared to a list of device commands stored on the controller, which then forwards the appropriate command directly to the device. When a device needs to access the CPU, the device controller issues an Interrupt Request (IRQ), which is then forwarded to the interrupt controller. The interrupt controller then forwards the request to the CPU.

Some of the more common device controllers found in a typical computer include:

- Keyboard Controller – controls the keyboard and PS/2 mouse (not always needed in newer systems)
- DMA Controller – controls Direct Memory Access
- Network Adaptor Controller – controls the Network Adaptor/ Network Interface Card (NIC)
- IDE Controller – controls EIDE devices, including the hard disk and CD/DVD drive
- Graphics Adaptor – controls video output devices, such as a monitor or LCD projector
- USB Controller – controls devices connected by USB

The Interrupt Controller

The *interrupt controller* is a special component on the motherboard that manages all interrupts, prioritizes them based on a predetermined priority sequence, and then forwards the interrupts to the CPU. You can think of the interrupt controller like a police officer at a busy intersection, or a security guard at the main gate to a government office. When a device wants the CPU's attention, its device driver initiates an interrupt request (IRQ). Each type of device has a different IRQ number assigned to it, so when multiple devices signal for the CPU's attention at the same time, the interrupt controller checks their IRQ number, and places them in a queue. The device with the lowest IRQ number gets the highest priority, just like a police officer letting more important vehicles (perhaps a government convoy or an ambulance) proceed through an intersection first.

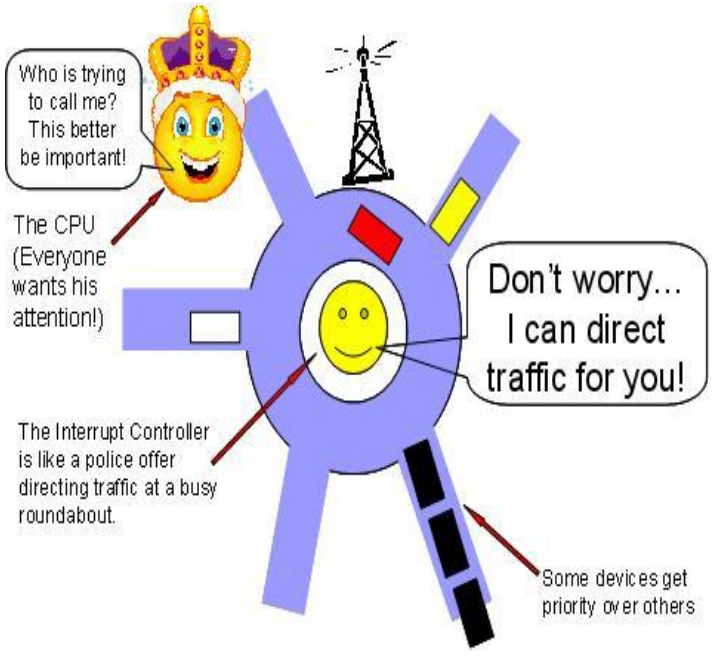


Figure 5.1
The Interrupt Controller is like a traffic officer giving priority to more important vehicles

Managing Interrupts

A typical modern computer system has two interrupt controllers that function as one unit. Each controller has eight lines. (This is because in the first personal computers developed in the late 1970s and early 1980s, the system bus was only eight bits wide.) Since every function of the computer needs the system clock in order to manage its timing, the highest priority is given to the system clock (which is given IRQ 0). The keyboard gets the next highest priority, since it is needed in order to manually override any other processes being carried by the computer. Thus, the keyboard is given IRQ 1.

Although we say that the higher the IRQ number is, the lower priority the device is given, devices using IRQ numbers 8-15 actually get higher priority than devices using IRQ numbers 3-7. This is because of the use of two controllers acting as one unit. All interrupt requests coming from the second controller are actually sent to IRQ number 2 on the first controller, which then forwards them on to the CPU. Figure 5.2 (below) demonstrates the structure of the interrupt controller system.

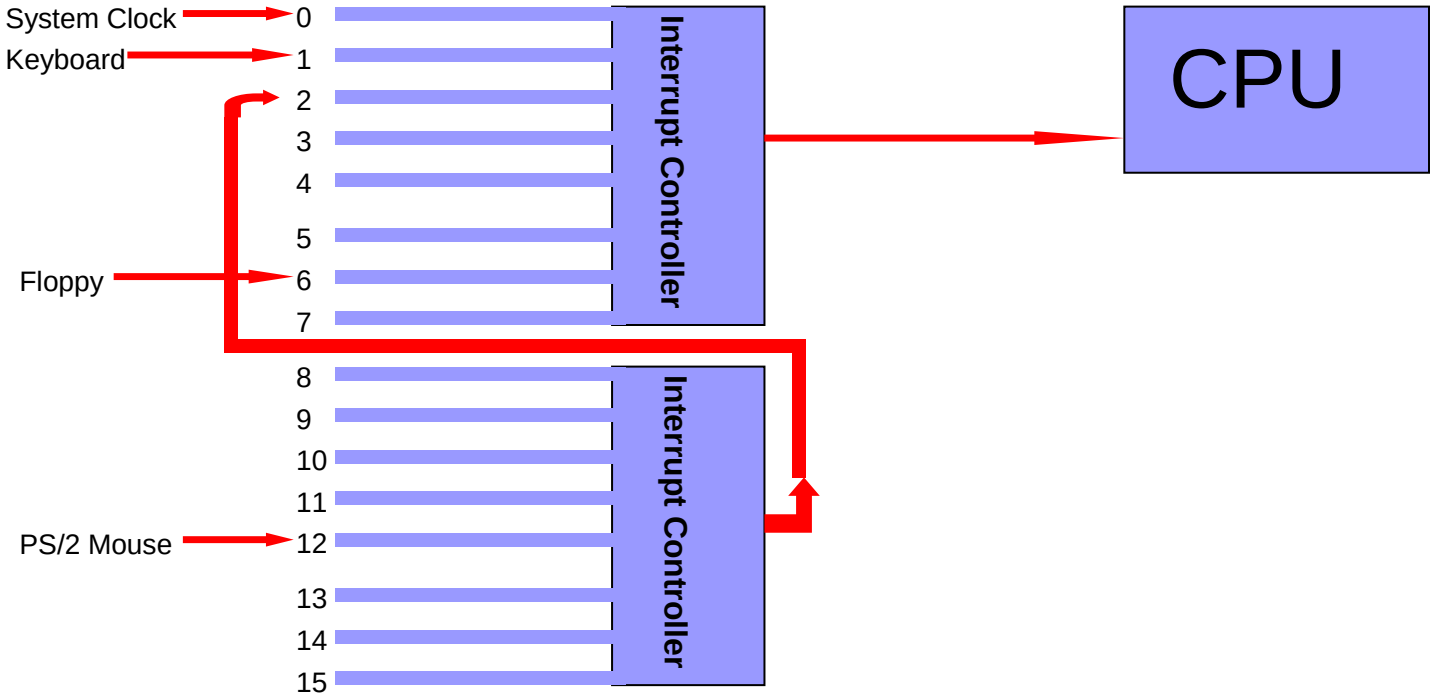


Figure 5.2
Structure of the Interrupt Controller System

Preemptable I/O Resources

Memory is a preemptable resource

Preemptable I/O resources are resources that can be taken away from a process that is currently using them. In order to be considered a preemptable resource, the reallocation of the resource must have no negative effect on the processes involved (other than the overall time needed to complete the process execution). A common example would be memory. When a resource is preempted, or taken away from a process, the process is often placed into a blocked state, and must wait until the resources is made available again before it can return to a ready state and continue being executed.

Non-Preemptable I/O Resources

Optical storage devices are non-preemtable

Non-preemptable I/O resources are resources that cannot be taken away from a process that is currently using them without having some negative effect. An example of a non-preemptable resource would be a CD/DVD drive. If a process is reading from or writing to an optical storage device, it is difficult to take that resource away from the process without corrupting whatever is being read or written to the drive.

Deadlocks

Deadlocks occur when multiple processes are holding I/O resources, and they each require resources that currently in use by another process that is unwilling to release the resource. When this occurs, neither process is able to make any progress. For example, one process may currently hold control of the optical storage (CD/DVD) drive, and may need to send something to the printer. A second process may hold control of the printer, and may need to access data from the CD/DVD drive. If neither process is willing to give up control of the resources it currently holds, then neither process will be able to proceed.

Block I/O Devices

Some I/O devices send and receive data from the computer system in blocks of characters. Such devices are referred to as *block I/O devices*. The operating system manages reading and writing data to block devices by using a data buffer system. *Data buffers* are allocated to hold a single block of characters. When the buffer is full, the data in the buffer is then sent to or from the I/O device in one chunk. Common examples of block I/O devices are hard disks, optical storage drives, and memory regions.

Character I/O Devices

Some I/O devices send and receive data from the computer one character at a time. Such devices are called *character I/O devices*. Common examples include virtual terminals and serial modems. Character transmission to character devices is *unbuffered*.

I/O Management Software

From a software perspective, there are really two main elements to consider. The first is *device drivers*, which are the operating system software components that interact with the devices controllers. The second element is the *interrupt handler*. This software is really part of the device driver, but it is responsible for issuing interrupt signals to the interrupt controller when a device requests access to the CPU.

The I/O software system functions in layers, as illustrated in Figure 5.3 (below). One layer is the device drivers. Each type of device shown below has its own controller, which is responsible for relaying instructions to all devices connected to it. Device drivers issue instructions from the operating system to the device controller, which in turn handles the actual functioning of the device. Another layer is the interrupts. When a device controller issues a request from a device to gain access to the CPU, the interrupt handler (part of the device driver) sends the interrupt signal to the interrupt controller. The interrupt is then relayed to the CPU, as outlined in the previous section.

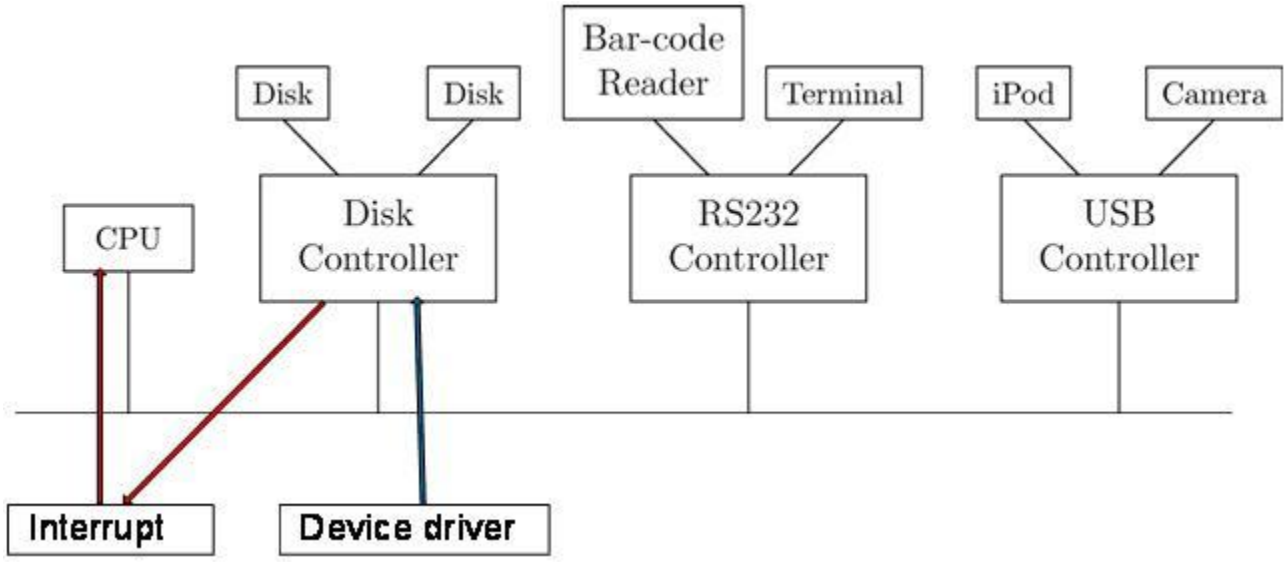


Figure 5.3
I/O Software System Layers

Device Drivers

Device drivers are essential components of an operating system. They translate instructions from the operating system and other processes into instructions that are used to drive the device controllers. In order to handle controlling the input/output devices that are part of the computer system, device drivers are structured into two layers. The *Upper Half* of the device driver handles taking requests in from the operating system, and places them in a "*Shared Requests List*." The *Lower Half* of the device driver handles taking requests from the shared requests list, and programs them for the device control to carry out the instructions. The *interrupt handler* is part of the Lower Half of the device driver. It is used when the device controller calls for an interrupt request, and it handles the issuing of the interrupt request to the interrupt controller. Figure 5.4 (below) illustrates the structure of a typical device driver.

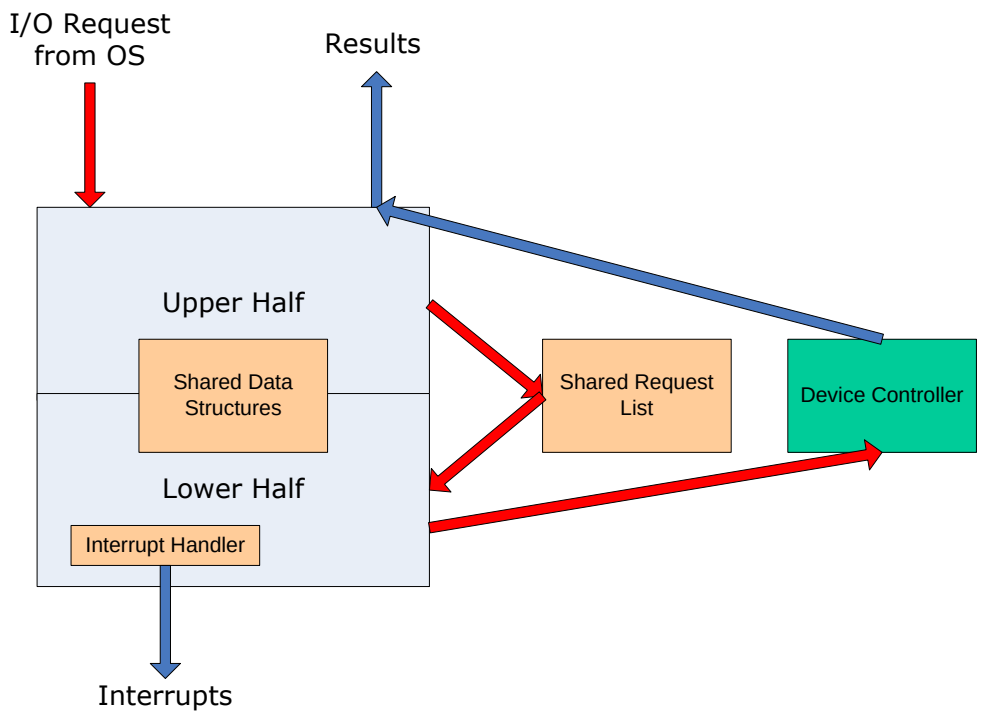
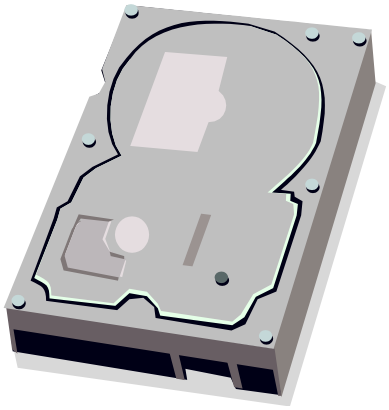


Figure 5.4
Device Driver Structure

In this diagram, the blue box represents the device driver, which is divided into the Upper Half and Lower Half. The red arrows represent the flow of instructions from the operating system, which first pass into the Upper Half, then into the Shared Request List. These instructions are then programmed by the Lower Half, and forwarded to the device controller for execution. The blue arrows represent the flow of requests and data out of the device driver. In this diagram, the device controller is returning results back to the device driver, which then sends them back out to the operating system. The interrupt handler is also demonstrated issuing an interrupt request to the interrupt controller.

Managing Magnetic Disks

Magnetic disk storage represents one of the most popular categories of I/O resources used by a computer. Processes and threads must be able to read data from, and write data to magnetic storage devices such as hard disks and floppy disks. This is most frequently done when accessing pages from virtual memory, but it also occurs when processes try to open or close files stored on disks. In this section, we will look at how magnetic storage devices are formatted to hold data, and how that data is structured on a disk so that the operating system can read and manage it. We will also take a look at the elevator algorithm, which is a pattern used by operating systems to manage data retrieval as efficiently as possible.



Hard disks are common I/O resources

Disk Formatting

Disk formatting is the process of preparing a disk to hold data. When you reformat a disk, it destroys all of the data previously stored on it. There are three key stages to the formatting process for a disk:

Low-level formatting:
- Takes place at the factory.
- Involves physically drawing tracks and sectors on the disk surface.

Partitioning:
- Done by the user.
- Involves dividing the disk into logical sections.
- Only primary hard disks can be partitioned (cannot partition external hard disks or floppy disks)

High-level formatting:
- Done by the user.
- Involves the selection of a file system (such as FAT or NTFS for Windows), and the installation of an operating system.
- Different partitions can use different file systems, and can even be used to install different operating systems in the same computer.

What are Partitions?

Partitions are logical divisions of a hard disk. That is, you divide your hard disk into sections that will be used for different purposes. You could create multiple partitions if you want to install multiple operating systems. (When you boot your computer and select the operating system to use, it will use the appropriate partition, and ignore the others.) You could also partition a hard disk so that you can store files using two different file systems, or so that you can provide additional security for your files (for example, if you store your personal files on a different partition than your operating system and software, then your files will be less likely to get infected by a virus). Partitioning a hard disk has other file system benefits, such as reducing the size of file *clusters* (which results in less wasted space as more files are stored on the disk).

A hard disk can only have four primary partitions, each of which could be used for a different file system and/or operating system. A partition that contains an operating system is called an *active partition*. However, once an operating system is installed, you can select a partition and further divide it into *logical drives* (each of which would be assigned a drive letter). Although the data on logical drives is not physically isolated (as it is in partitions), each logical drive appears as if it were a separate disk for file management purposes. This could be useful for organizing file storage, or for allocating logical drive space for the exclusive use of different computer users.

How is Data Structured on a Magnetic Disk?

Data on magnetic disks is physically organized using *tracks*, *sectors* and *clusters*. On hard disks, data is also organized using *platters* (a floppy disk has only one platter, while a hard disk can have several physical platters, which are each organized into tracks, sectors and clusters). A *track* is a concentric circle on a disk or platter. A *sector* is a division in a concentric circle. A *cluster* is the smallest unit of storage space available on a disk. The larger a hard disk is, the bigger the clusters become. On a floppy disk, the cluster size is the same as the size of a sector (512 bytes). Figure 5.5 (below) shows the organization of tracks and sectors on a floppy disk.

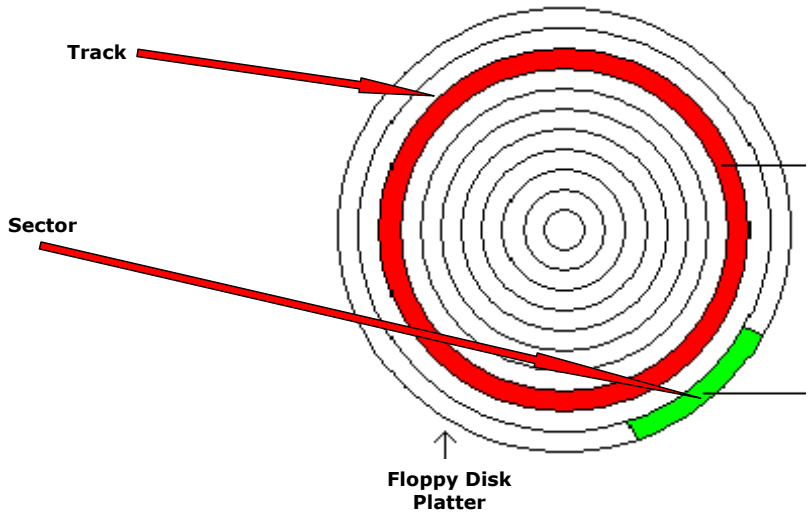


Figure 5.5
Tracks and Sectors on a Floppy Disk

In addition to creating tracks and sectors, when you format a floppy disk, you also create a *boot record*, a *File Allocation Table* (*FAT*) and a *Root Directory*. The *boot record* is always the first sector on a disk. It contains a *bootstrap loader*, which can be used to boot a computer from the disk, as well as information about how the disk is organized. The *File Allocation Table* lists the location of all clusters on the disk, and how they are currently being used. The *Root Directory* lists all of the files and subdirectories currently stored on the disk.

Hard disks are somewhat different from floppy disks. A hard disk contains several physical *platters* which are stacked on top of each other. Each platter is double-sided, with tracks and sectors drawn on the surface. All of the concentric tracks that line up with each other on each platter are referred to as a *cylinder*. Figure 5.6 (below) shows how the platters stack up inside of a hard disk. There is a *read/write arm* (with a *read/write head*) for each platter surface.

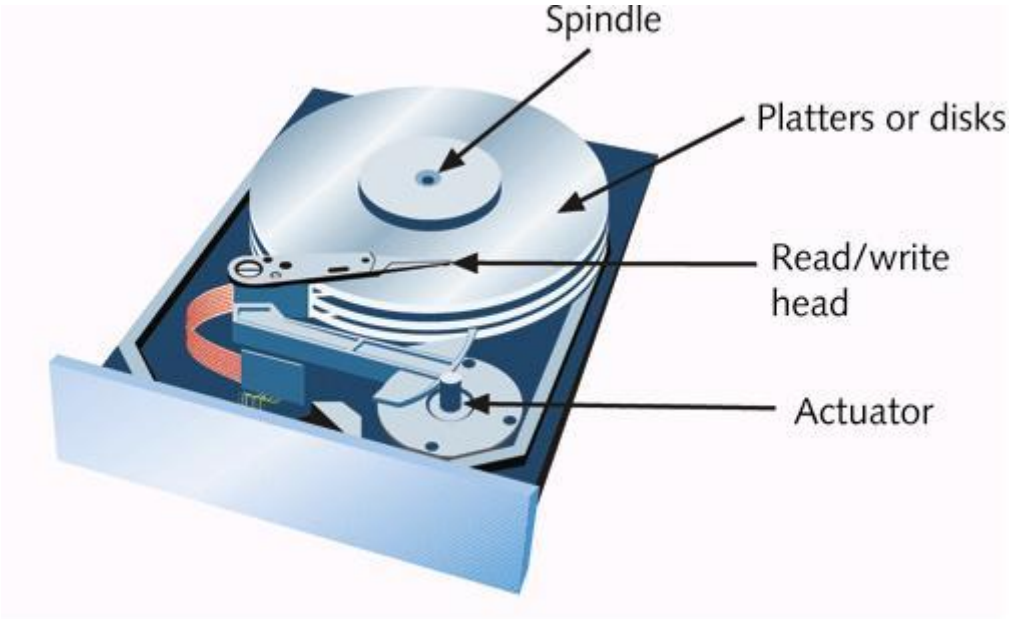


Figure 5.6
Inside a Hard Disk

Another difference between the physical structure of a floppy disk and a hard disk is the size of the sectors and clusters on the disk. Cluster size varies on hard disks, depending upon the actual size of the disk, the file system used, and the number of partitions. On a floppy disk, the sector size is fixed at 512 bytes. That means, the closer you get to the center of the physical disk, the fewer the number of sectors per track. On newer hard disks, the sectors get smaller as you get closer to the center of the disk. This means that you can have the same number of sectors for each track. The differences in how tracks are divided into sectors on floppy disks and hard disks are shown in Figure 5.7 (below).

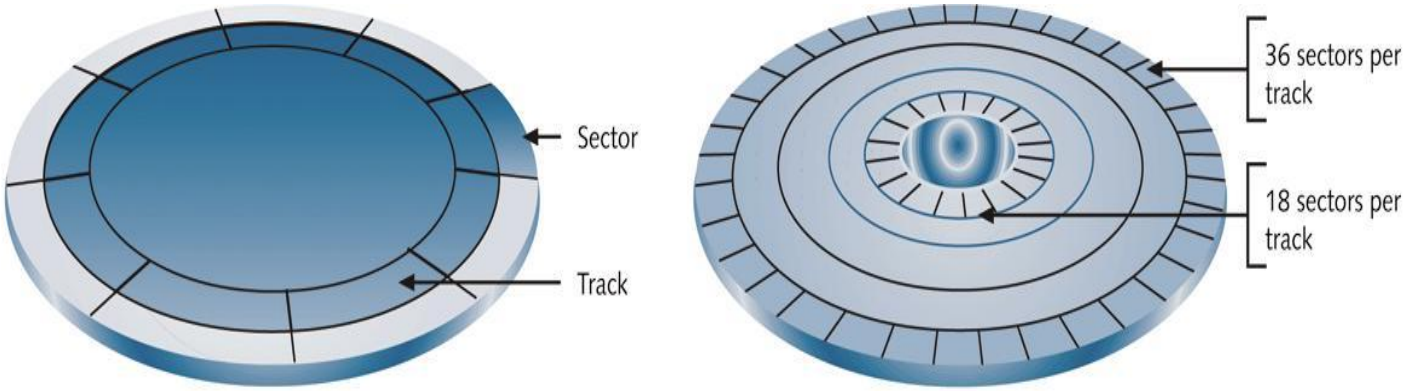


Figure 5.7
Tracks and Sectors on Floppy Disks and Hard Disks

The Elevator Algorithm

Now that we know how magnetic disks are physically structured, we can turn our attention to how the interrupt handler actually handles requests to retrieve information from disk resources. This is typically done using the *elevator algorithm*. For this discussion, we will concentrate on requests for data from a hard disk. However, it should be noted that the elevator algorithm can be used by the interrupt handler to handle requests for data from other storage devices, as well.

The elevator algorithm is a strategy used by the interrupt handler to find data on a storage device when it is requested by a process. Using a first come first serve strategy to fill requests for data that have accumulated in the shared request folder is not an effective strategy, because it would actually take too long to constantly change the direction of travel of a disk's read/write head. This strategy would also increase wear and tear on a disk. Instead, the elevator algorithm organizes data requests based on the data's cylinder location on a disk. Using the elevator algorithm, the read/write head will only move in one direction at a time, and will "pick up" requested data as it moves from cylinder to cylinder.

The actual decision making process used by the interrupt handler is:

1. If the request queue is empty, exit interrupt handler
2. If read/write arm's direction is UP, move from the current cylinder (just completed request) to the next higher cylinder to perform that request.
3. If direction is DOWN, move the arm from the current cylinder (just completed request) to next lower cylinder to perform that request.
4. If no request lies in current direction, reverse direction.

Using this algorithm, we can examine the handling of requests for data from cylinders on a hard disk. Assume that the original request order in the shared request folder is for data from the following cylinders:

1, 16, 9, 31, 11, 12, 29

Figure 5.8 (below) shows the current position of the disk's read/write head, the current direction of movement and the locations of the requested data.

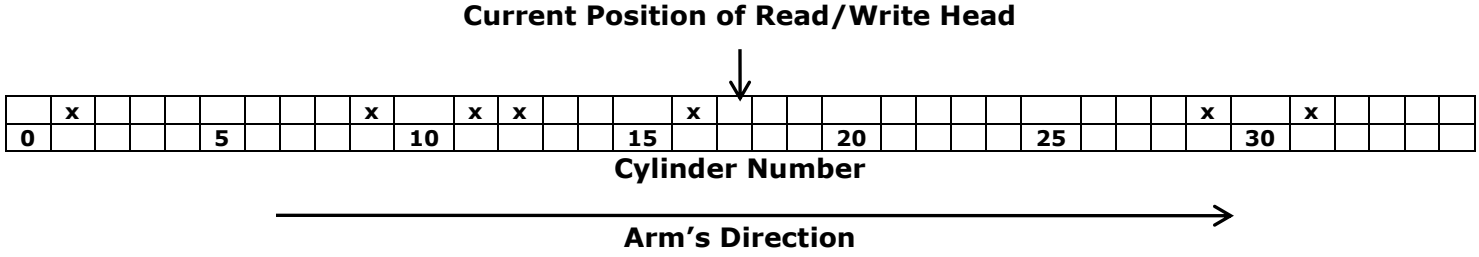


Figure 5.8
Retrieving Data using the Elevator Algorithm

In this example, the read/write head is currently located at cylinder 17, and the arm is moving in the Up direction. The next piece of data requested in the current direction is on cylinder 29, so the actual order in which the data requests are filled would be:

29, 31, 16, 12, 11, 9, 1

System Clocks

Every process carried out by a computer is timed by a clock. While we generally refer to the *system clock* as one single timing device, there are actually several clocks operating in a modern computer.

The main system clock is part of the motherboard, and is typically timed in *Gigahertz* (*GHz*). One *hertz* means one cycle per second, so a system clock operating a 2.4 GHz (a typical speed for a modern PC) operates at 2.4 billion ticks (or cycles) per second. A single *cycle* of a system clock is the shortest possible amount of time that any process running in a computer can be completed. The main system bus operates in time with the main system clock. Other clocks in a modern computer include the processor clock, the cache clock and clocks for the various I/O busses. The operating speeds for each of these clocks are different, but they are all tied to the main system clock. Their clock speeds are calculated by either multiplying or dividing the system clock speed by a set number.

Some clocks operate more slowly than others because the types of devices they regulate (such as I/O devices) cannot function at speeds as high as the system clock. Typically, the CPU operates at a faster speed than the system clock. It is obvious that, since some devices operate at slower intervals, the processor may actually spend extra time waiting for those devices to complete tasks before it can continue with a particular process from the Process Table. However, this does not necessarily mean that the entire computer system will be slowed down, as the process waiting on a device can go into a blocked or wait state, and the process scheduler can cycle to another process from the Process Table.

Unit Summary

Every computer has numerous input/output devices that must be managed by the operating system. The operating system needs to control the functioning of these devices. It must also manage sending and receiving information from the devices, and sharing these resources amongst the many processes that are running (or waiting to be run) at any given time. In order to help the operating system with these tasks, each type of I/O device usually has its own device controller. The device controller talks to the operating system's device drivers, and translates instructions from the operating system so that they can be carried out by the device. A special device called the Interrupt Controller handles the task of receiving interrupt requests (IRQs – requests to either send or receive information from the CPU), and prioritizes them to be forwarded to the processor.

Some I/O resources (like memory) are considered preemtable. That means that if they are being shared by multiple processes, they can be taken away from whatever process is currently using them and reallocated to another process. Other I/O resources (like CD/DVD drives) are considered non-preemptable. That means that once they have been allocated to a process, they cannot be taken away from that process until it has been completed (without causing some sort of problem or corruption to the task). Deadlocks can occur when two (or more) processes have control of different I/O resources that are needed by the other processes, and they are unwilling to give up control of the device. When this happens, neither process is able to be completed.

From a software perspective, there are two main elements that come into play in managing I/O resources. The first is the device driver, which is supplied by the operating system to control an

I/O device. Device drivers are structured into an Upper Half and a Lower Half. The Upper Half handles taking requests from the operating system and places them into a shared request folder for the device. The Lower Half takes requests from the shared request folder, and programs them to be carried out by the device. The second major element is the interrupt handler, which is really a part of the Lower Half of the device driver. It is responsible for issuing interrupt signals to the computers interrupt controller when a device needs the attention of the CPU.

One of the most common types of I/O device managed by an operating system is magnetic storage devices. Devices such as floppy disks and hard disk drives need to be physically formatted into tracks, sectors and cylinders (for hard disks), and they need to be formatted with a file system that the operating system knows how to read. When retrieving data from a hard disk, the operating system often relies on the Elevator Algorithm to determine in which direction to move the disk's read/write head to find the next nearest piece of data, and in which order the data should be most efficiently retrieved.

All of the I/O resources in a computer system operate in time with the main system clock. There are actually several clocks inside of a modern computer, each operating at different speeds (depending upon what can be handled by the type of device they govern). Each of these clocks is actually tied to the main system clock, and their speeds can be determined by either multiplying or dividing the system clock speed by a set number. One cycle of a system clock is the shortest amount of time that any process running in a computer can actually be completed.

Key Terms

Active partition
Block I/O devices
Boot record
Bootstrap loader
Character I/O devices
Cluster
Cycle
Cylinder
Data buffer
Deadlock
Device controller
Elevator algorithm
File Allocation Table (FAT)
Formatting
Gigahertz (GHz)
Graphics adaptor
Hertz (Hz)
High-level formatting
IDE controller
Input/Output resources
Interrupt controller
Interrupt handler
Keyboard controller
Logical drive
Lower Half
Low-level formatting
Magnetic disk storage
Network adaptor controller
Non-preemptable I/O resources
Partition
Partitioning
Platter
Preemptable I/O resources
Read/write arm
Read/write head
Root directory
Sector
Shared requests folder
System clock
Track
Unbuffered
Upper Half
USB controller

Unit 6: File Systems

Managing File Systems

We have already taken a detailed look at the structure of a typical operating system, as well as how an operating system manages all of a computer's hardware resources, RAM and the scheduling of all of the processes and threads that need to be executed. In this unit, we will examine the last key element of a user's experience with a computer system. File system management encompasses the provision of a way to store your data in a computer, as well as a way for you to find and access that data when you need it.

We will begin this unit by looking at the requirements in a computer system in order to provide long-term storage of information. We will also look at an operating system's File Manager and all of its responsibilities. This will be followed by an examination of common strategies used by the File Manager to store and retrieve information from storage media. From there, we will turn our attention to the file systems that are created when an operating system formats a storage device (such as a hard disk). This will include a comparison of the characteristics of the FAT32 and NTFS file systems, as well the purposes of the Master Boot Record and Partition Tables created on a hard disk during formatting. We will conclude this unit with a look at how users actually interact with a file system. This will include an examination of the purposes of directories or folders, and the navigation of hierarchical file structures.

Long-Term Storage of Information

When we examined computer memory, we saw that RAM is only useful for the temporary storage of data and instructions that are currently being used by a computer. Although RAM is very fast, it is *volatile memory*. That is, when you turn off the power to the computer, all data stored in RAM is lost. In addition, a typical computer has a relatively small amount of RAM compared to a user's long-term data storage needs. In order to permanently store information, an operating system has several requirements:

Hard disks: the most common long-term storage media

- A storage device (typically a magnetic hard disk).
- A device controller and device driver for the storage disk.
- Strategies for reading and writing data to a disk.
- A file system that provides a structure and rules for file encoding, management and security.

Although long-term data storage can be achieved using a variety of media (ranging from optical discs to floppy disks and USB Flash drives), this unit will focus on magnetic hard disks. That is because every personal computer system relies upon hard disk storage, and hard disks are by far the most common long-term data storage media.

The File Manager

The *File Manager* is the part of the operating system that is responsible for the management of long-term storage devices such as hard disks. The File Manager provides a framework (or a *hierarchical structure*) for the organization of data that is stored on a disk. The primary responsibilities of the File Manager include:

- The organization of similar data (text, images, music and sounds, video, etc) in some manner, and putting that data into files.
- The organization of files into directories and folders.
- The organization of different levels of folders.
- The management of different partitions in a disk.
- The management of multiple disks for data storage.

The File Manager has additional responsibilities that are based on the primary responsibilities listed above. These include:

- Partitioning and formatting disks.
- Establishing file-naming rules or conventions.
- Providing data integrity.
- Providing error recovery and prevention tools.
- Providing file security.
- Providing directory paths.
- Providing command sets to manipulate files.

Data Storage Strategies

Data storage strategies refer to how the operating system physically organizes and stores pieces of files on a disk. There are three common data storage strategies:

1. Contiguous Allocation Strategy
2. Linked Allocation Strategy
3. Indexed Allocation Strategy

Contiguous Allocation Strategy

The *contiguous allocation strategy* is perhaps the simplest strategy for an operating system to set up. In a contiguous allocation file system, all pieces of a file are stored in contiguous (or physically adjacent) storage spaces on a disk. To keep track of files stored on the disk, the operating system maintains a *file directory* that lists the names of the files, the start block for the file on the disk, and the actual size of the file. The contiguous strategy has both advantages and disadvantages.

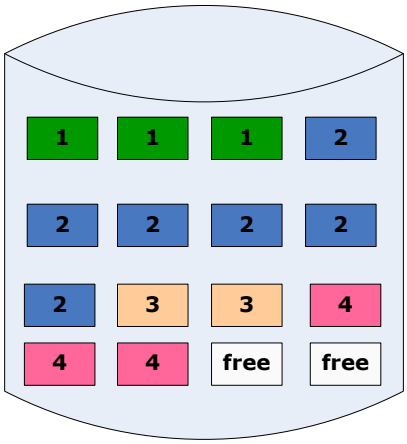
Advantages of the Contiguous Allocation Strategy

- Easy to set up.
- Easy approach for the sequential access of data.

Disadvantages of the Contiguous Allocation Strategy

- Not easy to expand the size of a file.
- Difficult to manage file fragmentation.
- OS must allocate and reserve contiguous space during the initial creation of the file.

Figure 6.1 (below) shows the allocation of disk space for files, and a file directory, for a sample hard disk.



Contiguous File Table

File Number	Start Block	File Size
1	1	3
2	4	6
3	10	2
4	13	3

Figure 6.1
Contiguous File Allocation

Linked Allocation Strategy

Using the linked allocation strategy, the contents of a file can be stored anywhere on a disk (they do not need to be *contiguous*). The operating system maintains a file directory which contains the file name and the start block number. Each block of data belonging to a file contains a pointer, which points to the next block of data belonging to that file. The last block of data contains a special *End of File (EOF) indicator*, so that the operating system will know that it has retrieved all of the data for the file. Figure 6.2 (right) shows the File Directory entry and the entries in the File Allocation Table for a sample file using the Linked allocation Strategy.

The linked allocation strategy also has advantages and disadvantages.

Advantages of the Linked Allocation Strategy

- No file fragmentation issues.
- Easy to expand the size of a file.

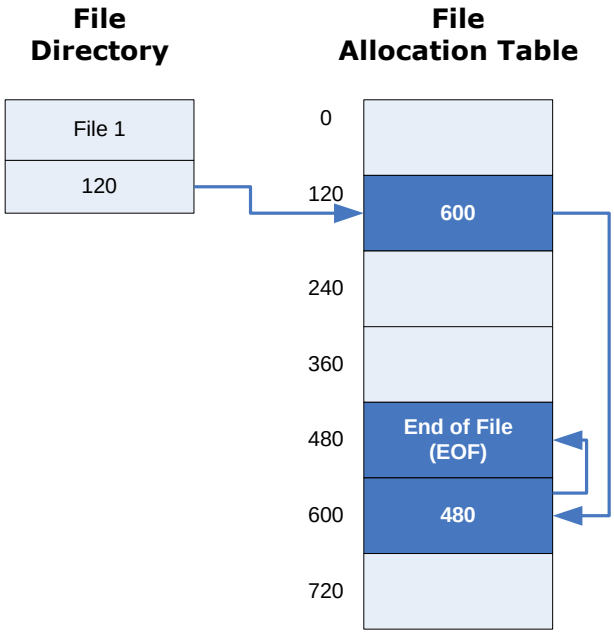


Figure 6.2
Linked Allocation Strategy

Disadvantages of the Linked Allocation Strategy

- The pointers field for each block of data uses up storage space.
- There are some inefficiencies in the direct access of data.

Indexed Allocation Strategy

When using an *indexed allocation strategy*, the operating system creates a file directory to track all of the files on a disk, and an index block for each file. The file directory entry for each file includes the file name, just like with the contiguous and linked allocation strategies. However, instead of listing the starting block on the disk for each file, the file directory entry will include its index block number. The *index block* contains pointers to all of the data blocks belonging to that file. Again, the indexed allocation strategy has both advantages and disadvantages.

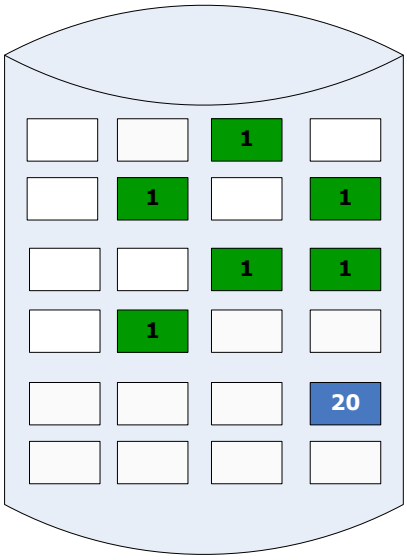
Advantages of the Indexed Allocation Strategy

- Grouping all pointers in one place increases reliability.
- Makes direct access of the contents of a file more efficient.

Disadvantages of the Indexed Allocation Strategy

- If a file is small, the index block containing all of the file pointers may contain a lot of wasted disk space.

Figure 6.3 (below) shows the allocation of disk space, the File Directory and the Index Block for a sample file on a hard disk using the Indexed Allocation Strategy.



File Directory

File	Index Block
File 1	20
File 2	…
…	…

The entry for File 1 in the File Directory points to Index Block 20.

Each entry in the Index Block points to the location on the hard disk of the next piece of File 1.

Index Block 20
3
12
5
11
14
8

Figure 6.3
Indexed File Allocation

File Systems

File systems provide the conventions for the encoding, storage and management of data on a storage device such as a hard disk. They also provide the tools that help users to interact with files. Different operating systems use different file systems, and files created/stored using one file system are not always compatible with an operating system that uses a different file system. As discussed in Unit 2: Operating System Fundamentals, some of the more common file systems include:

- FAT12 (floppy disks)
- FAT16 (DOS and older versions of Windows)
- FAT32 (older versions of Windows)
- NTFS (newer versions of Windows)
- EXT3 (Unix/Linux)
- HFS+ (Max OS X)

Regardless of which file system an operating system uses, the file system provides the following conventions and information for file management:

- Data and time the directory (folder) or file was created.
- Date and time the directory or file was last modified.
- Directory or file size.
- Directory or file attributes.

Each file system has its advantages and limitations. For example, the *FAT12* (12-bit *File Allocation Table*) file system used for floppy disks is ideal use with media with small storage capacities (a floppy disk can hold approximately 1.2 MB of data). However, it limits the disk to a maximum of 512 entries in the file table, which means a maximum of 512 files and folders (even if each file is only 1 byte!

Two of the most common file systems are the FAT32 (32-bit File Allocation Table) and NTFS (New Technology File System) systems used by later versions of Windows. FAT32 is an older file system with a disk size limitation of 32 GB. FAT32 also limits the size of any single file to a maximum of 4 GB. NTFS allows for disk (or volume) sizes of up to 2 terabytes (TB), with an unlimited number of files and folders. It also eliminates the 4 GB file size restriction. Table 6.1 (below) compares the characteristics of FAT32 and NTFS.

Table 6.1
Comparison of FAT32 and NTFS

FAT32	NTFS
• Used for older versions of Windows. • Still used for smaller capacity storage devices, such as USB flash drives. • Maximum disk (or volume) size of 32 GB. • Maximum file size of 4 GB. • File fragmentation issues.	• Default file system for Windows XP, Vista, and Windows 7. • Maximum disk (or volume) size of 2 TB. • No maximum file size. • No maximum number of files. • Greater security features, including individual *file compression*, *disk quotas*, and *file encryption*. • Easy to convert volumes from FAT32 to NTFS.

Is NTFS More Efficient Than FAT32?

In today's modern computing environment, where large hard disk sizes are standard on most personal computers, NTFS is a much more efficient file system for users of Windows operating systems. The reason for this lies in how NTFS handles space efficiency. Unlike FAT16 and FAT32, NTFS does not use a file allocation table. It operates as an actual database in terms of how it keeps track of files and their associated clusters. NTFS also allows for smaller cluster sizes on large disks, which results in less wasted space than FAT32. With FAT32, the larger the disk becomes, the larger the cluster size becomes. (Remember that a cluster is the smallest unit of storage space that can be allocated to a file. Even if a cluster only contains the last 2 KB of data from a file, if the cluster is 1024 KB, then the remaining 1022 KB cannot be used to store any other data!).

The database created by the operating system to manage file and cluster information for NTFS does use up a lot of space compared to the cluster maps used by FAT32. For instance, on a 50 MB disk drive, the NTFS cluster database could take up as much as 10 MB of space (which, obviously, cannot then be used to store actual files). For today's larger disk drives, the amount of space used by NTFS to manage files is relatively small and is therefore not a concern. NTFS also provides greater file compression capabilities than FAT32, which means that the same files may actually take up less space on a disk. However, for smaller capacity storage devices (such as USB flash drives or flash memory cards used by mobile phones and digital cameras), FAT32 is still a useful file system because it wastes less space to manage files and their associated clusters.

When working with larger hard disks, NTFS also provides increased file security over FAT32. Individual file compression is one technique used to achieve increased file security. File compression means that redundant data in files (ex: all of the information that is common between all MS Word files) is removed while the files are being stored, and added back in when the files are in use. NTFS allows operating system administrators to assign disk space quotas to individual user accounts, preventing users from using too much storage space. File encryption means that one user's files are encoded in such a way that they cannot be accessed by another user on the same computer (if the two users are using different accounts and if the files are not marked as *"shared" files*).

Master Boot Record and Partition Table

When a disk is partitioned and formatted for use with one or more file systems and operating systems, a *Master Boot Record* and a *Partition Table* are created. The Master Boot Record is located in the first 512 byte sector of a partitioned hard disk. It is actually located outside of any of the partitions that are created on the disk, and it serves the following purposes:

- Holds the partition table.
- Contains a bootstrap loader to continue the computer booting process after BIOS has completed its initial boot routines.
- Identifies each individual disk with a unique 32-bit disk signature.

The Partition Table is part of the Master Boot Record of a partitioned disk. It contains information about the size and type of partitions that have been created on a disk, including both the primary and extended partitions. Since Partition Table is always 64-bits in size, and each entry in the partition table is 16-bits, a disk can contain a maximum of four primary

partitions (although an extended partition can be further artificially subdivided into logical drives).

Interacting with File Systems

In this section, we will take a look at interacting with file systems from a user's point of view. Regardless of which file system your operating system uses, there are several concepts that are common to all operating systems. The first is the concept of a *file*. A file is simply a collection of data that is being used together. Pieces of a file may be scattered throughout a hard disk but, as we have seen, keeping track of all of those pieces is the responsibility of the File Manager. From a user's perspective, a file exists as a single item on a storage device. Files are organized based on *file type* (which is specified by a *file extension*) that associates the file with the application used to create (or view) it. *Directories* (or *folders*) are simply a means of organizing files so that they are easier for the user to find and manage. Sub-directories (or sub-folders) can exists inside of other folders, and act as a means of further organizing files for ease of management. For example, you can create a folder called "Operating System Fundamentals" to store all of your notes and assignments for a course on operating systems. Inside of that parent folder, you could create several child folders for each unit. A folder for this particular unit could be called "File Systems." You might also decide to keep your assignments in a separate folder called "Assignments."

In most operating systems, directories (or folders) are organized into a *hierarchical structure*. That is, they are organized from a root level (such as the C:\ drive in Windows), which then branches out like a tree into all of the directories and *subdirectories* created by the user. Some directories are created at the root level by the operating system itself, such as C:\Windows (which stores all of the files used by the operating system), or C:\Programs (which stores the files belonging to most of the applications installed on the computer). Figure 6.4 (below) shows the hierarchical structure of folders on a Windows XP system (as viewed using Windows Explorer).

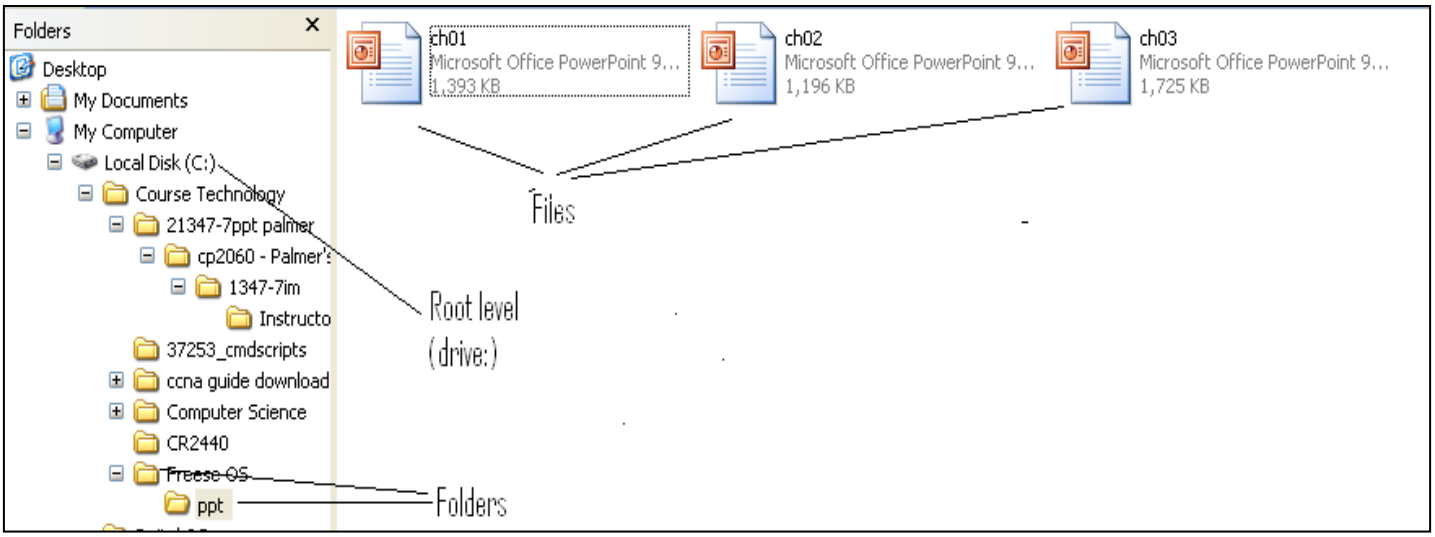


Figure 6.4
Hierarchical File Structure in Windows XP

The location of a file in a directory on a computer is referred to as its *path*. When we give the location of a file, we can provide it as either an *absolute path* or a *relative path*. An absolute

path provides the complete path to the file from the root level. For instance, in the example described above for folders you created for an operating systems course, the absolute path to that folder might be:

C:\Documents and Settings\Users\User 1\My Documents\Operating System Fundamentals\Assignments\Assignment_1.doc

This path provides the absolute location in the hierarchical file structure of a Microsoft Word document called Assignment_1.doc. We could also describe the location of this file using a relative path. A relative path is simply the path to a file from a predetermined relative level. For instance, we could provide the relative path for Assignment 1.doc as:

My Documents\Operating System Fundamentals\Assignments\Assignment 1.doc

Alternately, we could provide the relative path from the course directory:

Operating System Fundamentals\Assignments\Assignment 1.doc

Windows XP and Vista provide tools for navigating and managing files and folders. The most common tool is *Windows Explorer*, which is shown in Figure 6.4 (above). Windows Explorer shows the hierarchical directory structure of the folders in the left pane, and shows the contents of the currently selected folder in the right pane. The hierarchical file structure for in Windows Explorer usually shows shortcuts to the "Desktop" and the user's "My Documents" folders at the top of the left pane, followed by "Local Disk C:," which is the root level of the primary hard disk (or hard disk partition).

Windows Explorer also allows the user to specify how information about files will be displayed in the right pane. For instance, you can show just icons or tiles for each file (with a graphical depiction of the file type). You could also list all of the files, along with their associated sizes, attributes and their creation and modification dates. In addition, Windows Explorer provides the user with the ability to manipulate files and folders. Users can use Explorer to rename, delete, copy, paste, move or even create new files and folders.

File attributes allow the user control access to files, as well as the archiving and backup of files. The most common file *attributes* in Windows are:

- Read Only – the file can be read, but not modified.
- Hidden – the file is hidden from other users (usually used to hide critical operating system files so that they are not accidently modified by users).
- Archive – the file is marked for archiving, so that it will be included when the next file system backup operation is carried out.
- Index – the file is indexed in a database so that it will be easier for the operating system to find and retrieve the file from the disk when it is needed.

Of course, there are many other advanced attributes in Windows XP and Vista that can be set for particular users or groups of users.

Like Windows, Linux also uses a hierarchical file structure. Unlike Windows, the root level in Linux is actually called "*Root*." Even when logged into a Linux system as an administrator, the "Root" directory is usually hidden from the user when using the File Manager tool. As shown in Figure 6.5 (below), when a user is logged in with root level privileges, the *Linux File Manager* will display a warning to prevent the user from accidentally manipulating critical operating system files.

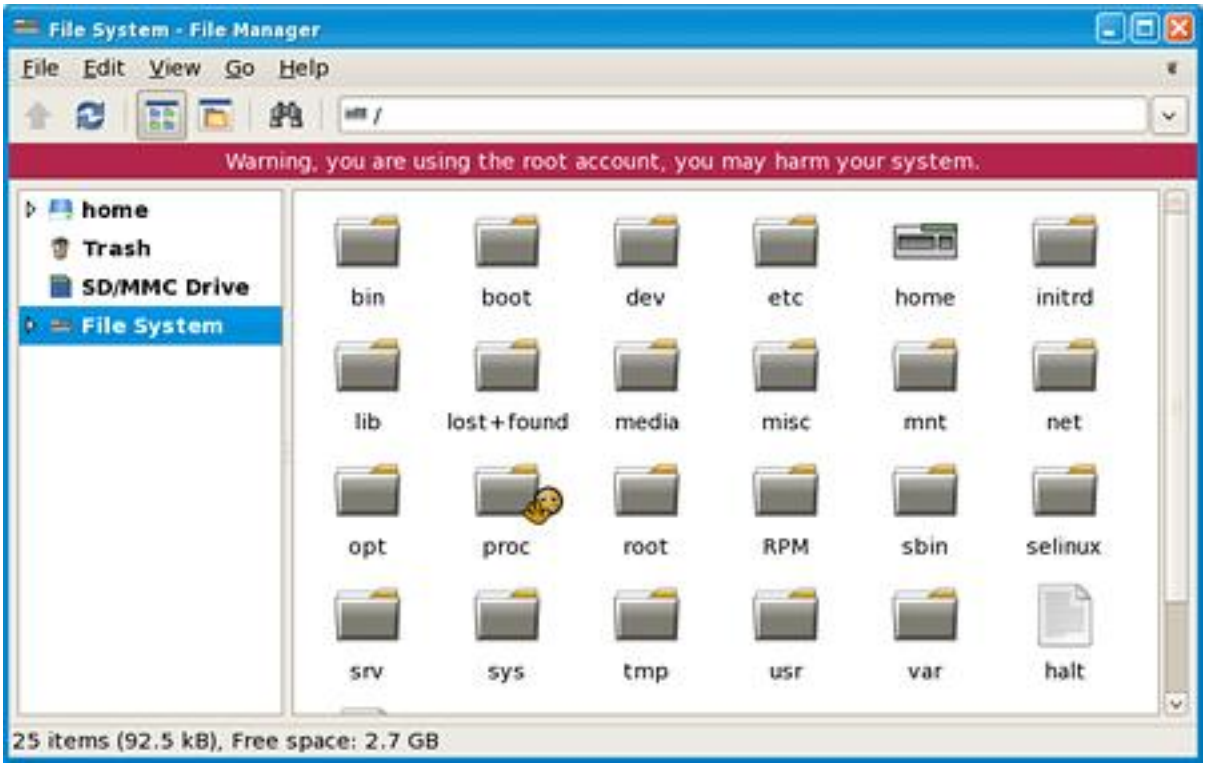


Figure 6.5
The Linux File Manager Tool

As with Windows Explorer, the Linux File Manager tool provides a graphical depiction of a disk's hierarchical file structure in the left pane. Again, icons depicting a selected directory's contents are shown in the right pane, and the user can select several options for what information is displayed about the files and folders listed.

Unit Summary

When using any computer, both the system and the user will need to store files when they are not in use. Because the RAM used to store data and instructions while they are in use is temporary, permanent data storage is achieved using large capacity storage devices such as magnetic hard disks. The operating system's File Manager is responsible for managing the data stored on a storage device, which includes providing a file system with rules for encoding, storing, organizing and retrieving data. This data is typically organized into files and folders in a hierarchical structure. Modern operating systems use a variety of strategies for allocating and managing the space used by pieces of files, which are often scattered (or fragmented) throughout a hard disk. The most common are the Contiguous Allocation, Linked Allocation and Indexed Allocation strategies. Each strategy uses a different method of allocating and tracking space for files, and each has advantages and disadvantages.

Different operating systems use different file systems for encoding and managing data. Two of the most common file systems are FAT32 and NTFS. While NTFS is used by newer versions of Windows (including XP, Vista and Windows 7), FAT32 is still useful because it is more efficient for use with smaller capacity storage devices (such as USB flash drives, memory cards, or flash memory in mobile devices).

When a hard disk is formatted and partitioned for use with a file system (and an operating system), a Master Boot Record is created before the first primary partition. The Master Boot Record contains instructions for booting a system once BIOS routines have been completed. It also contains a Partition Table, which describes the partitions that have been created on the disk.

From a user's perspective, files exist on a storage device as single items. Files are categorized by file type, which is indicated by a file extension. The extension associates a file with the application that will use it. Files also have associated attributes, which can be used to improve file security and management. Files are organized into directories (or folders) and subdirectories (or subfolders). Windows operating systems provide a tool called Windows Explorer which helps the user to graphically navigate and manipulate files and folders in the hierarchical file structure of a disk. A similar tool, called the File Manager, is provided by Linux.

Key Terms

Absolute path
Archive
Attributes
Contiguous
Contiguous allocation strategy
Directory
Disk quota
End of File (EOF) indicator
Extension
FAT12
File
File compression
File directory
File encryption
File Manager
File system
File type
Folder
Hidden
Hierarchical structure
Index
Index block
Indexed allocation strategy
Linked allocation strategy
Linux File Manager
Master Boot Record
NTFS
File Allocation Table
Partition Table
Path
Read Only
Relative path
Root
Shared files
Subdirectory
Subfolder
Volatile memory
Windows Explorer