Introduction

References:

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, **Ninth** Edition ", Chapter 1

Just as in **_The Blind Men and the Elephant_**, this chapter looks at Operating Systems from a number of different viewpoints. No one view really shows the complete picture, but by looking from a number of different views, we can get a pretty good overall picture of what operating systems are all about.

## 1.1 What Operating Systems Do - For Users, For Applications, etc.
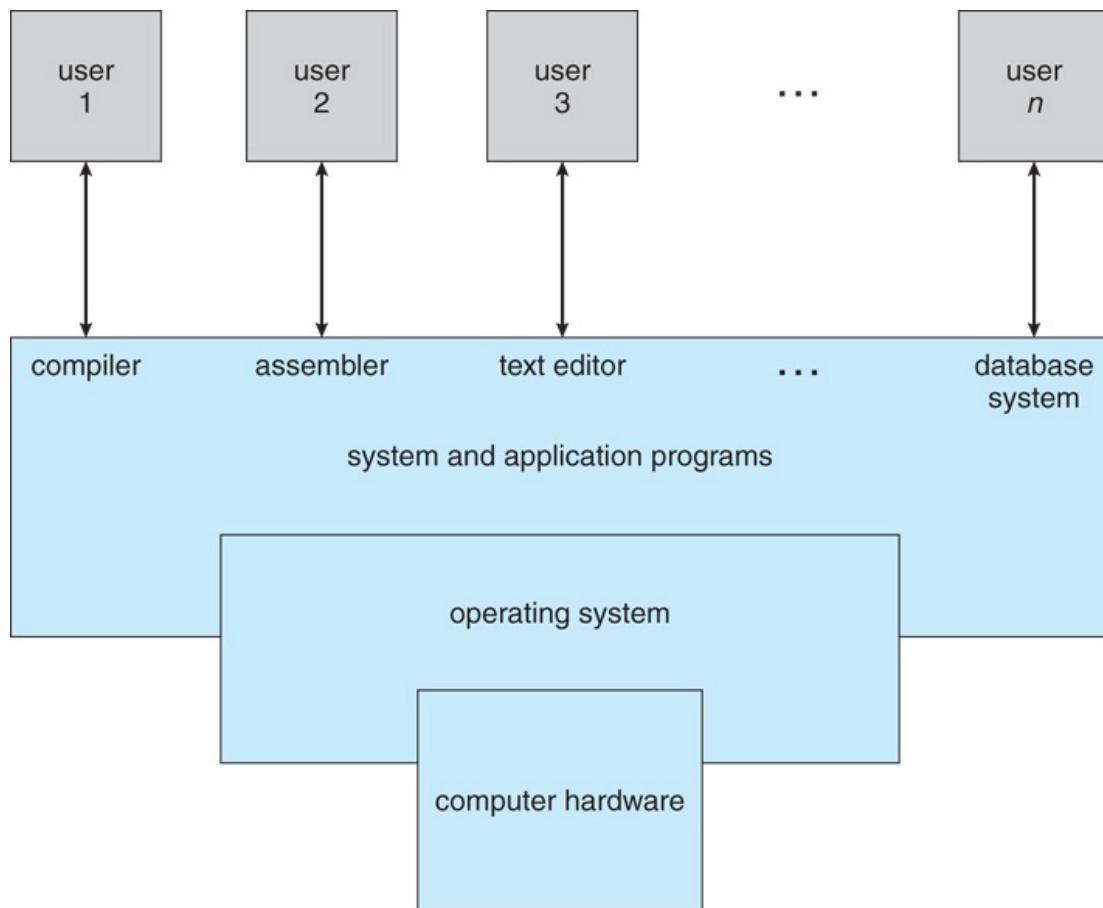
Figure 1.1 - Abstract view of the components of a computer system

- Computer = HW + OS + Apps + Users
- OS serves as interface between HW and (Apps & Users)
- OS provides services for Apps & Users
- OS manages resources (Government model, it doesn't produce anything.)
- Debates about what is included in the OS - Just the kernel, or everything the vendor ships?
  (Consider the distinction between system applications and 3rd party or user apps.)

1.2 Computer-System Organization - What are all the parts, and how do they fit together?
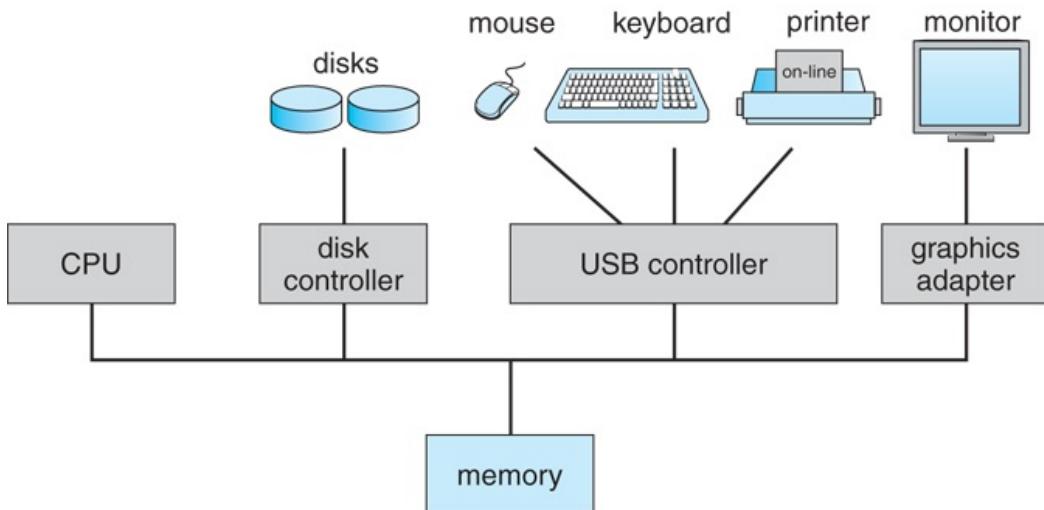
Figure 1.2 - A modern computer system

1.2.1 Computer-System Operation

- Bootstrap program
- Shared memory between CPU and I/O cards
- Time slicing for multi-process operation
- Interrupt handling - clock, HW, SW
- Implementation of system calls
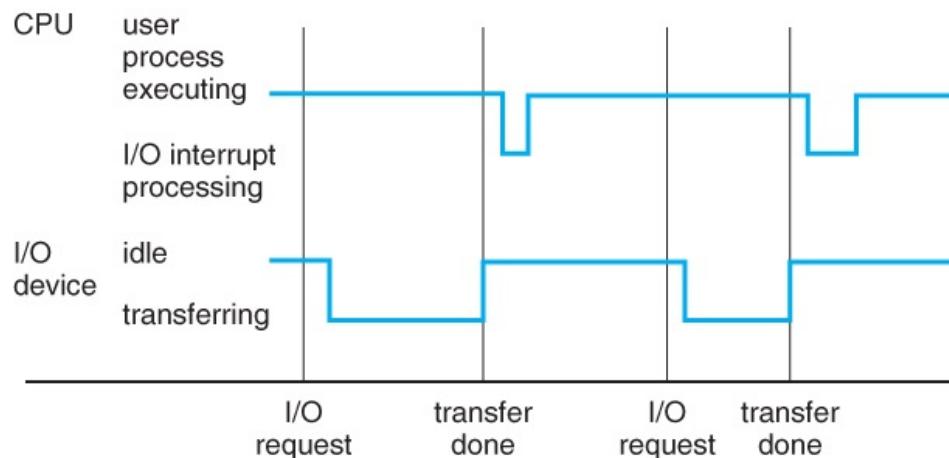
Figure 1.3 - Interrupt timeline for a single process doing output

1.2.2 Storage Structure

- Main memory (RAM)
    - Programs must be loaded into RAM to run.
    - Instructions and data fetched from RAM into registers.
    - RAM is volatile
    - "Medium" size and speed
- Other electronic (volatile) memory is faster, smaller, and more expensive per bit:
    - Registers
    - CPU Cache
- Non-volatile memory ("permanent" storage) is slower, larger, and less expensive per bit:
    - Electronic disks
    - Magnetic disks
   - Optical disks
    - Magnetic Tapes
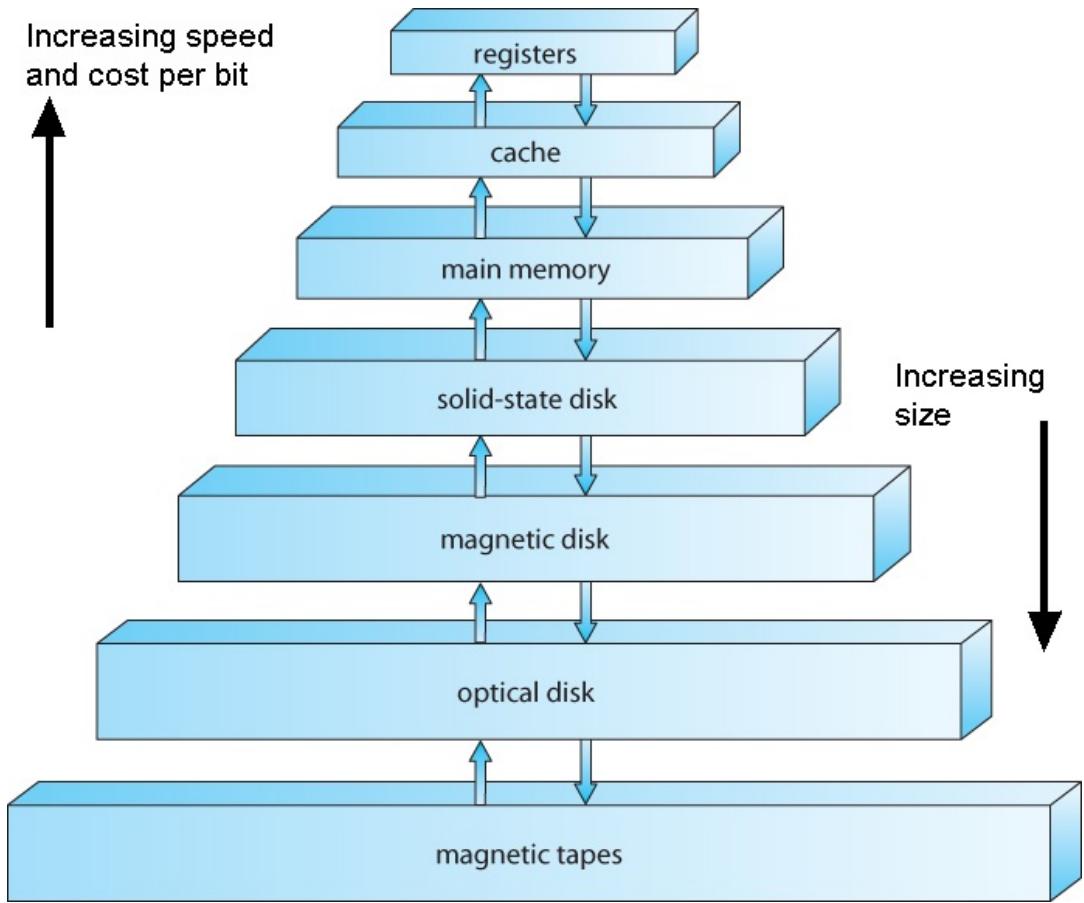
Figure 1.4 - Storage-device hierarchy

1.2.3 I/O Structure

- Typical operation involves I/O requests, direct memory access (DMA), and interrupt handling.
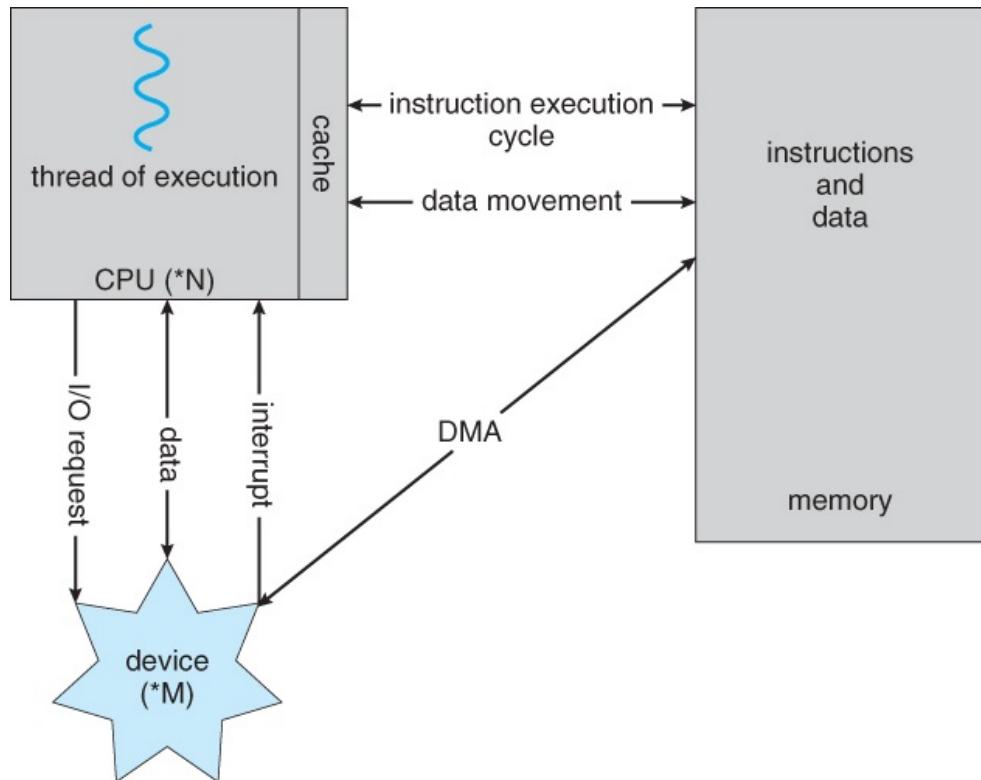

Figure 1.5 - How a modern computer system works

1.3 Computer-System Architecture - Different Operating Systems for Different Kinds of Computer Environments

1.3.1 Single-Processor Systems

- One main CPU which manages the computer and runs user apps.
- Other specialized processors (disk controllers, GPUs, etc.) do not run user apps.

1.3.2 Multiprocessor Systems

1. Increased throughput - Faster execution, but not 100% linear speedup.
2. Economy of scale - Peripherals, disks, memory, shared among processors.
3. Increased reliability
   - Failure of a CPU slows system, doesn't crash it.
   - Redundant processing provides system of checks and balances. (e.g. NASA)
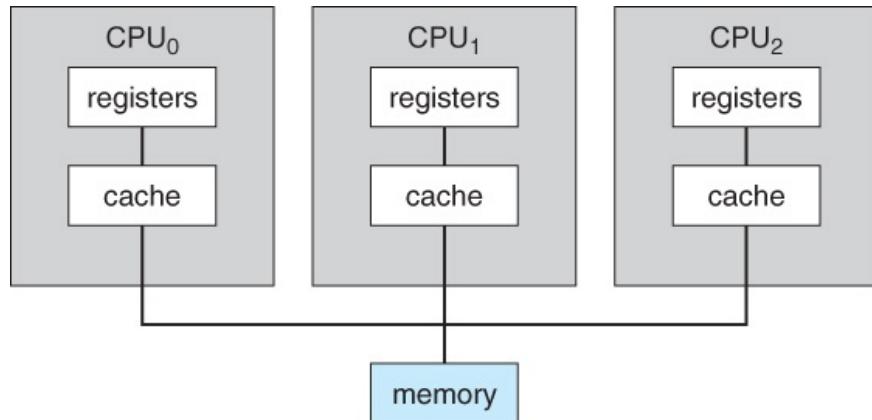
Figure 1.6 - Symmetric multiprocessing architecture
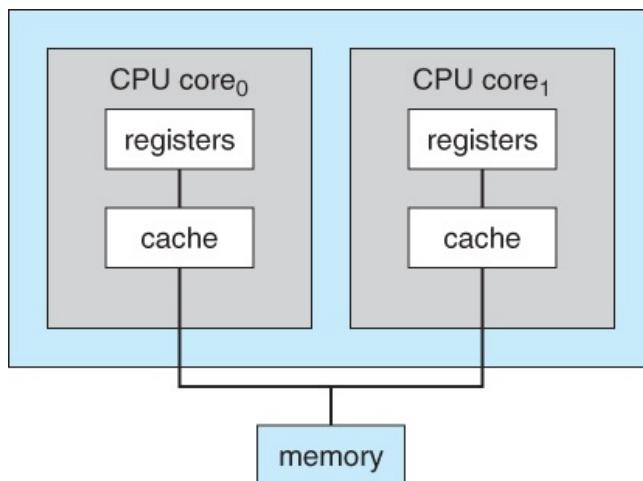
**Figure 1.7 - A dual-core design with two cores placed on the same chip**

1.3.3 Clustered Systems

- Independent systems, with shared common storage and connected by a high-speed LAN, working together.
- Special considerations for access to shared storage are required, (Distributed lock management), as are collaboration protocols.
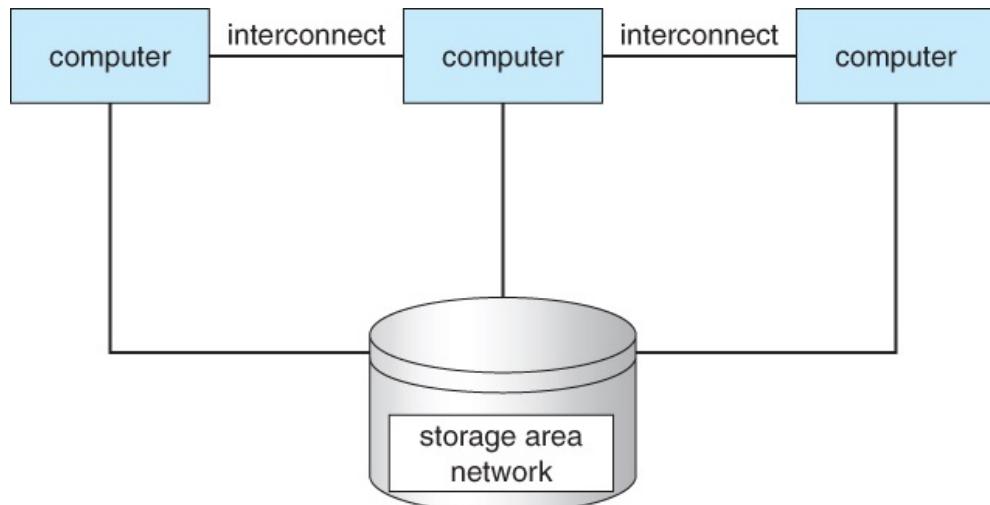

Figure 1.8 - General structure of a clustered system

1.4 Operating-System Structure

A time-sharing (multi-user multi-tasking) OS requires:

- Memory management
- Process management
- Job scheduling
- Resource allocation strategies
- Swap space / virtual memory in physical memory
- Interrupt handling
- File system management
- Protection and security
- Inter-process communications
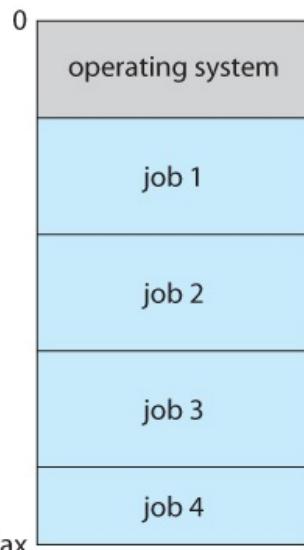
Figure 1.9 - Memory layout for a multiprogramming system

1.5 Operating-System Operations

Interrupt-driven nature of modern OSes requires that erroneous processes not be able to disturb anything else.

1.5.1 Dual-Mode and Multimode Operation

- User mode when executing harmless code in user applications
- Kernel mode (a.k.a. system mode, supervisor mode, privileged mode) when executing potentially dangerous code in the system kernel.
- Certain machine instructions (privileged instructions) can only be executed in kernel mode.
- Kernel mode can only be entered by making system calls. User code cannot flip the mode switch.
- Modern computers support dual-mode operation in hardware, and therefore most modern OSes support dual-mode operation.
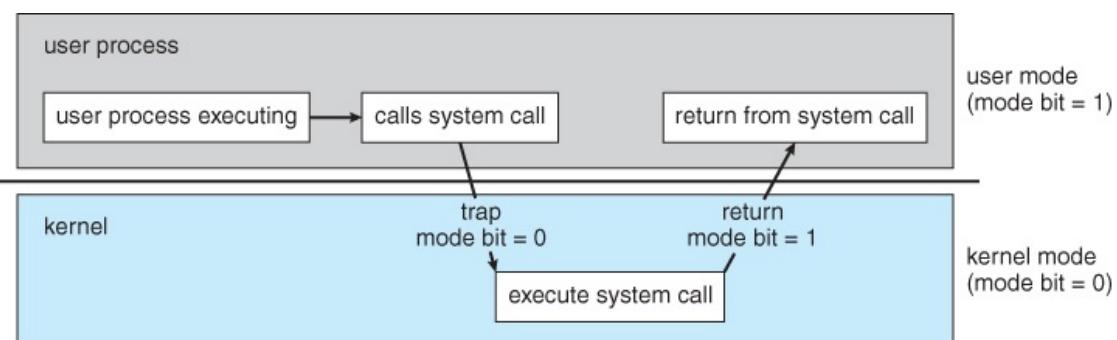
Figure 1.10 - Transition from user to kernel mode

- The concept of modes can be extended beyond two, requiring more than a single mode bit
- CPUs that support virtualization use one of these extra bits to indicate when the virtual machine manager, VMM, is in control of the system. The VMM has more privileges than ordinary user programs, but not so many as the full kernel.
- System calls are typically implemented in the form of software interrupts, which causes the hardware's interrupt

handler to transfer control over to an appropriate interrupt handler, which is part of the operating system, switching the mode bit to kernel mode in the process. The interrupt handler checks exactly which interrupt was generated, checks additional parameters (generally passed through registers) if appropriate, and then calls the appropriate kernel service routine to handle the service requested by the system call.

- User programs' attempts to execute illegal instructions (privileged or non-existent instructions), or to access forbidden memory areas, also generate software interrupts, which are trapped by the interrupt handler and control is transferred to the OS, which issues an appropriate error message, possibly dumps data to a log (core) file for later analysis, and then terminates the offending program.

1.5.2 Timer

- Before the kernel begins executing user code, a timer is set to generate an interrupt.
- The timer interrupt handler reverts control back to the kernel.
- This assures that no user process can take over the system.
- Timer control is a privileged instruction, (requiring kernel mode.)

1.6 Process Management

An OS is responsible for the following tasks with regards to process management:

- Creating and deleting both user and system processes
- Ensuring that each process receives its necessary resources, without interfering with other processes.
- Suspending and resuming processes
- Process synchronization and communication
- Deadlock handling

1.7 Memory Management

An OS is responsible for the following tasks with regards to memory management:

- Keeping track of which blocks of memory are currently in use, and by which processes.
- Determining which blocks of code and data to move into and out of memory, and when.
- Allocating and deallocating memory as needed. (E.g. new, malloc)

1.8 Storage Management

1.8.1 File-System Management

An OS is responsible for the following tasks with regards to filesystem management:

- Creating and deleting files and directories
- Supporting primitives for manipulating files and directories. (open, flush, etc.)
- Mapping files onto secondary storage.
- Backing up files onto stable permanent storage media.

1.8.2 Mass-Storage Management

An OS is responsible for the following tasks with regards to mass-storage management:

- Free disk space management
- Storage allocation
- Disk scheduling

Note the trade-offs regarding size, speed, longevity, security, and re-writability between different mass storage devices, including floppy disks, hard disks, tape drives, CDs, DVDs, etc.

1.8.3 Caching

- There are many cases in which a smaller higher-speed storage space serves as a cache, or temporary storage, for some of the most frequently needed portions of larger slower storage areas.
- The hierarchy of memory storage ranges from CPU registers to hard drives and external storage. (See table below.)
- The OS is responsible for determining what information to store in what level of cache, and when to transfer data from one level to another.
- The proper choice of cache management can have a profound impact on system performance.

- Data read in from disk follows a migration path from the hard drive to main memory, then to the CPU cache, and finally to the registers before it can be used, while data being written follows the reverse path. Each step (other than the registers) will typically fetch more data than is immediately needed, and cache the excess in order to satisfy future requests faster. For writing, small amounts of data are frequently buffered until there is enough to fill an entire "block" on the next output device in the chain.
- The issues get more complicated when multiple processes (or worse multiple computers) access common data, as it is important to ensure that every access reaches the most up-to-date copy of the cached data (amongst several copies in different cache levels.)

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

**Figure 1.11 - Performance of various levels of storage**
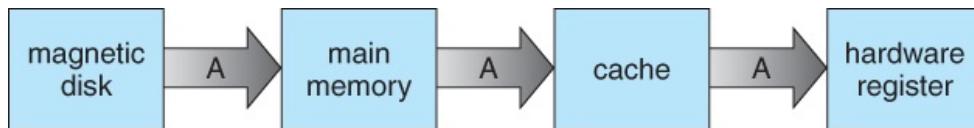
Figure 1.12 - Migration of integer A from disk to register

1.8.4 I/O Systems

The I/O subsystem consists of several components:
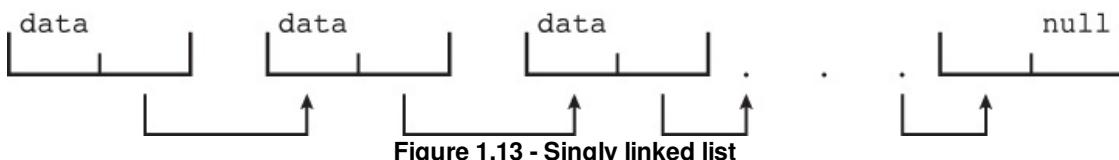
- A memory-management component that includes buffering, caching, and spooling.
- A general device-driver interface.
- Drivers for specific hardware devices.
- (UNIX implements multiple device interfaces for many types of devices, one for accessing the device character by character and one for accessing the device block by block. These can be seen by doing a long listing of /dev, and looking for a "c" or "b" in the first position. You will also note that the "size" field contains two numbers, known as the major and minor device numbers, instead of the normal one. The major number signifies which device driver handles I/O for this device, and the minor number is a parameter passed to the driver to let it know which specific device is being accessed. Where a device can be accessed as either a block or character device, the minor numbers for the two options usually differ by a single bit.)

1.9 Protection and Security

- **Protection** involves ensuring that no process access or interfere with resources to which they are not entitled, either by design or by accident. (E.g. "protection faults" when pointer variables are misused.)
- **Security** involves protecting the system from deliberate attacks, either from legitimate users of the system attempting to gain unauthorized access and privileges, or external attackers attempting to access or damage the system.

1.10 Kernel Data Structures

### 1.10.1 Lists, Stacks, and Queues

Figure 1.13 - Singly linked list
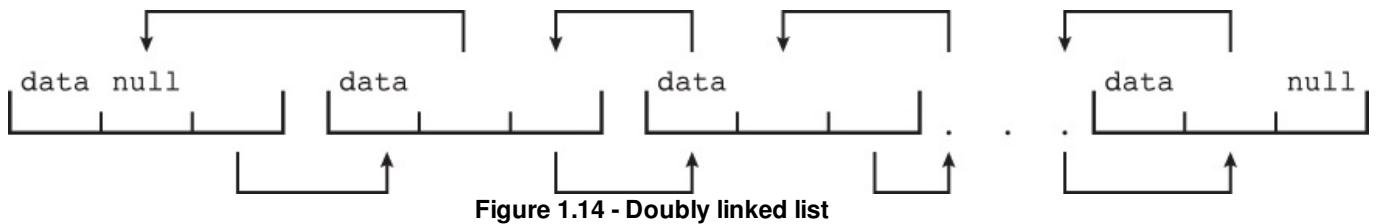
Figure 1.14 - Doubly linked list
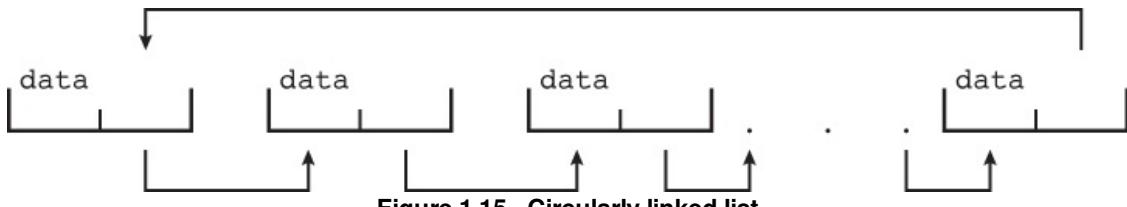
Figure 1.15 - Circularly linked list

1.10.2 Trees
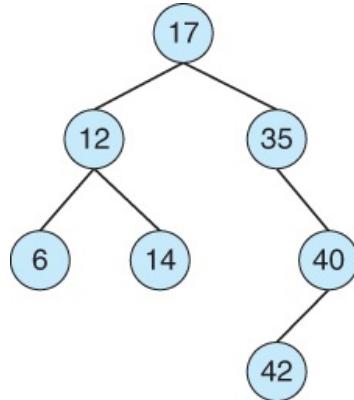
Figure 1.16 - Binary search trees

### 1.10.3 Hash Functions and Maps
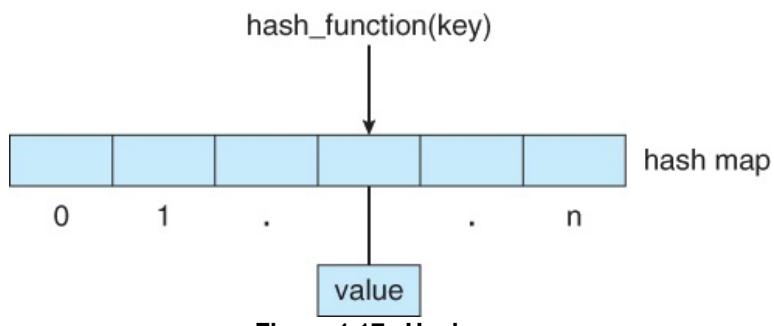
Figure 1.17 - Hash map

1.10.4 Bitmaps

- A string of 1s and 0s used to keep track of the boolean state of a collection of objects, such as the free state of blocks on a disk or pages in memory..

1.11 Computing Environments

1.11.1 Traditional Computing

1.11.2 Mobile Computing

- Computing on small handheld devices such as smart phones or tablets. (As opposed to laptops, which still fall under traditional computing.)
- May take advantage of additional built-in sensors, such as GPS, tilt, compass, and inertial movement.
- Typically connect to the Internet using wireless networking (IEEE 802.11) or cellular telephone technology.
- Limited in storage capacity, memory capacity, and computing power relative to a PC.
- Generally uses slower processors, that consume less battery power and produce less heat.
- The two dominant OSes today are Google Android and Apple iOS.

1.11.3 Distributed Systems

- Distributed Systems consist of multiple, possibly heterogeneous, computers connected together via a network and cooperating in some way, form, or fashion.
- Networks may range from small tight LANs to broad reaching WANs.
  - WAN = Wide Area Network, such as an international corporation
  - MAN =Metropolitan Area Network, covering a region the size of a city for example.
  - LAN =Local Area Network, typical of a home, business, single-site corporation, or university campus.
  - PAN = Personal Area Network, such as the bluetooth connection between your PC, phone, headset, car, etc.
- Network access speeds, throughputs, reliabilities, are all important issues.
- OS view of the network may range from just a special form of file access to complex well-coordinated network operating systems.
- Shared resources may include files, CPU cycles, RAM, printers, and other resources.

1.11.4 Client-Server Computing

- A defined server provides services (HW or SW) to other systems which serve as clients. (Technically clients and servers are processes, not HW, and may co-exist on the same physical computer.)
- A process may act as both client and server of either the same or different resources.
- Served resources may include disk space, CPU cycles, time of day, IP name information, graphical displays (X Servers), or other resources.
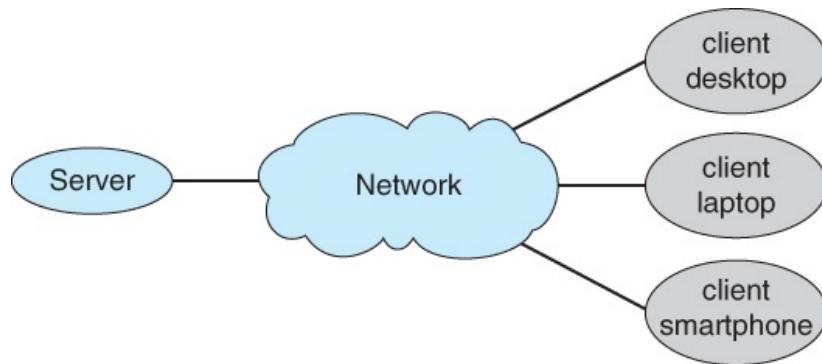
Figure 1.18 - General structure of a client-server system

1.11.5 Peer-to-Peer Computing

- Any computer or process on the network may provide services to any other which requests it. There is no clear "leader" or overall organization.
- May employ a central "directory" server for looking up the location of resources, or may use peer-to-peer searching to find resources.
- E.g. Skype uses a central server to locate a desired peer, and then further communication is peer to peer.
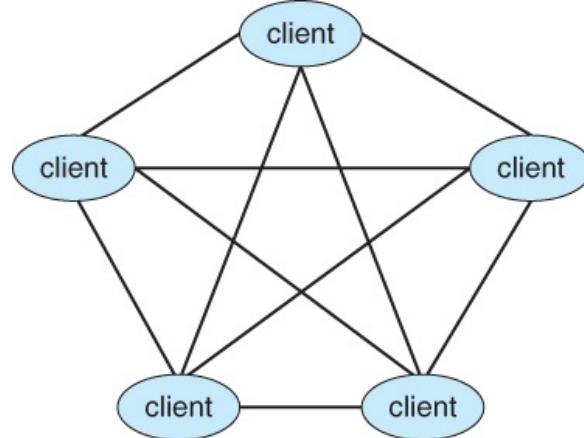
Figure 1.19 - Peer-to-peer system with no centralized service

1.11.6 Virtualization

- Allows one or more "guest" operating systems to run on virtual machines hosted by a single physical machine and the virtual machine manager.
- Useful for cross-platform development and support.

- For example, a student could run UNIX on a virtual machine, hosted by a virtual machine manager on a Windows based personal computer. The student would have full root access to the virtual machine, and if it crashed, the underlying Windows machine should be unaffected.
- System calls have to be caught by the VMM and translated into (different) system calls made to the real underlying OS.
- Virtualization can slow down program that have to run through the VMM, but can also speed up some things if virtual hardware can be accessed through a cache instead of a physical device.
- Depending on the implementation, programs can also run simultaneously on the native OS, bypassing the virtual machines.
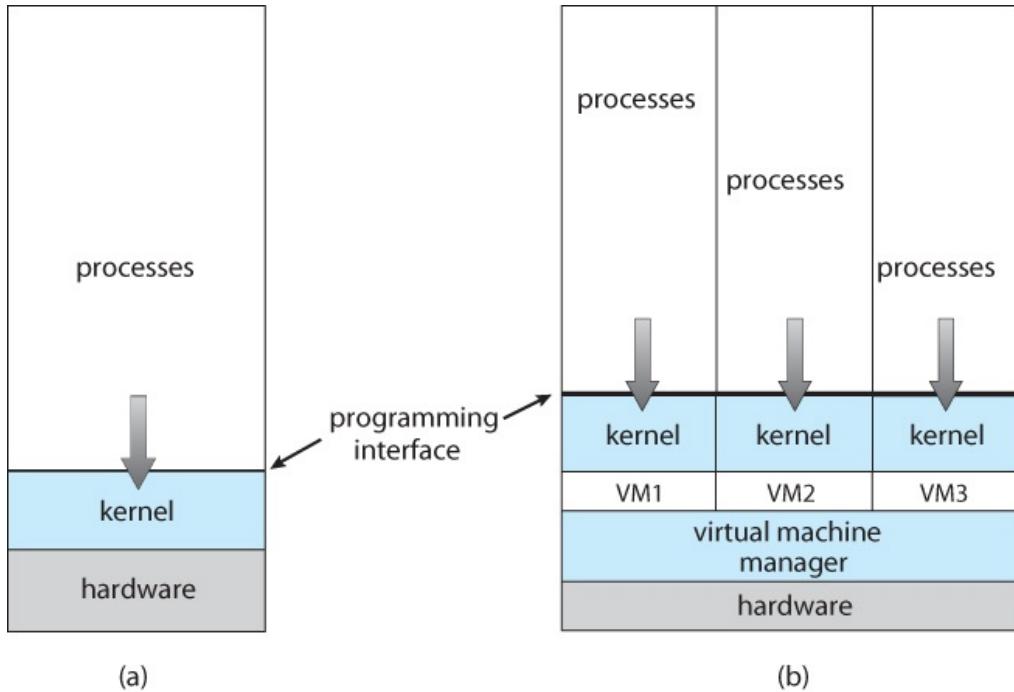

Figure 1.20 - VMWare

- To run Linux on a Windows system using VMware, follow these steps:
    1. Download the free "VMware Player" tool from http://www.vmware.com/download/player and install it on your system
    2. Choose a Linux version from among hundreds of virtual machine images at http://www.vmware.com/appliances
    3. Boot the virtual machine within VMware Player.

1.11.7 Cloud Computing

- Delivers computing, storage, and applications as a service over a network.
- Types of cloud computing:
    - Public cloud - Available to anyone willing to pay for the service.
    - Private cloud - Run by a company for internal use only.
    - Hybrid cloud - A cloud with both public and private components.
    - Software as a Service - SaaS - Applications such as word processors available via the Internet
    - Platform as a Service - PaaS - A software stack available for application use, such as a database server
    - Infrastructure as a Service - IaaS - Servers or storage available on the Internet, such as backup servers, photo storage, or file storage.
    - Service providers may provide more than one type of service
- Clouds may contain thousands of physical computers, millions of virtual ones, and petabytes of total storage.
- Web hosting services may offer (one or more) virtual machine(s) to each of their clients.
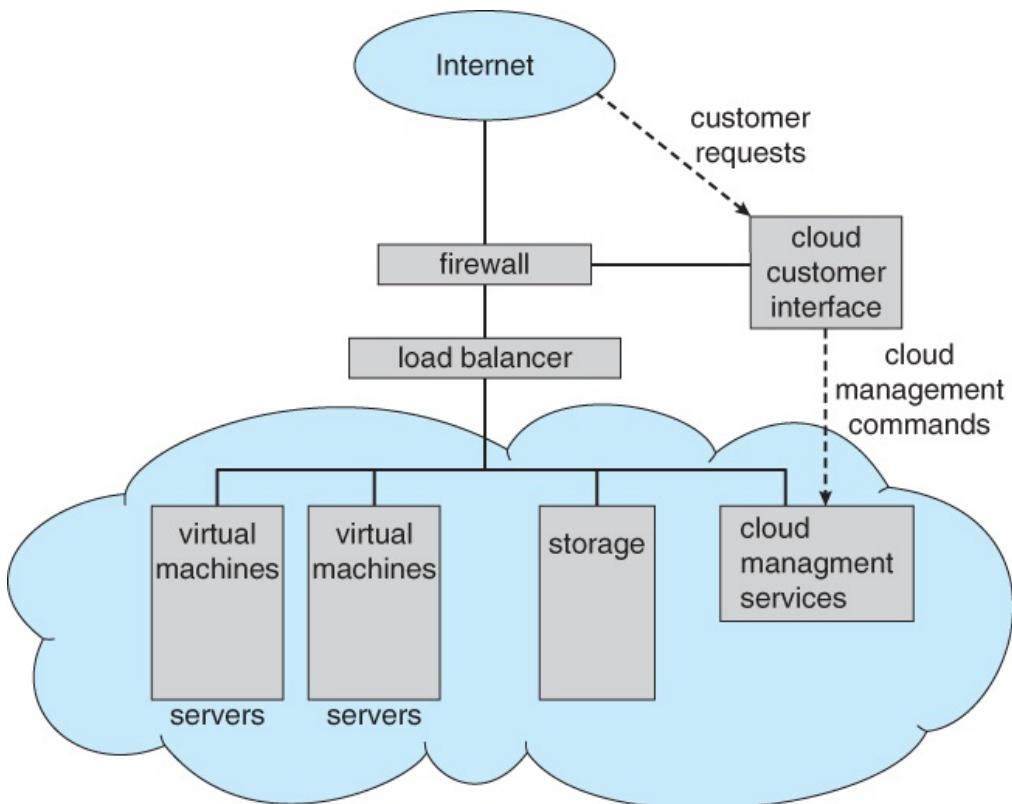
Figure 1.21 - Cloud computing

1.11.8 Real-Time Embedded Systems

- Embedded into devices such as automobiles, climate control systems, process control, and even toasters and refrigerators.
- May involve specialized chips, or generic CPUs applied to a particular task. (Consider the current price of 80286 or even 8086 or 8088 chips, which are still plenty powerful enough for simple electronic devices such as kids toys.)
- Process control devices require real-time (interrupt driven) OSes. Response time can be critical for many such devices.

1.12 Open-Source Operating Systems

- (For more information on the Flourish conference held at UIC on the subject of Free Libre and Open Source Software , visit http://www.flourishconf.com)
- Open-Source software is published (sometimes sold) with the source code, so that anyone can see and optionally modify the code.
- Open-source SW is often developed and maintained by a small army of loosely connected often unpaid programmers, each working towards the common good.
- Critics argue that open-source SW can be buggy, but proponents counter that bugs are found and fixed quickly, since there are so many pairs of eyes inspecting all the code.
- Open-source operating systems are a good resource for studying OS development, since students can examine the source code and even change it and re-compile the changes.

1.12.1 History

- At one time (1950s) a lot of code was open-source.
- Later, companies tried to protect the privacy of their code, particularly sensitive issues such as copyright protection algorithms.
- In 1983 Richard Stallman started the GNU project to produce an open-source UNIX.
- He later published the GNU Manifesto, arguing that ALL software should be open-source, and founded the Free Software Foundation to promote open-source development.
- FSF and GNU use the GNU General Public License which essentially states that all users of the software have full rights to copy and change the SW however they wish, so long as anything they distribute further contain the same license agreement. (Copylefting)

1.12.2 Linux

- Developed by Linus Torvalds in Finland in 1991 as the first full operating system developed by GNU.
- Many different distributions of Linux have evolved from Linus's original, including RedHat, SUSE, Fedora, Debian, Slackware, and Ubuntu, each geared toward a different group of end-users and operating environments.
- To run Linux on a Windows system using VMware, follow these steps:
    1. Download the free "VMware Player" tool from http://www.vmware.com/download/player and install it on your system
    2. Choose a Linux version from among hundreds of virtual machine images at http://www.vmware.com/appliances
    3. Boot the virtual machine within VMware Player.

1.12.3 BSD UNIX

- UNIX was originally developed at ATT Bell labs, and the source code made available to computer science students at many universities, including the University of California at Berkeley, UCB.
- UCB students developed UNIX further, and released their product as BSD UNIX in both binary and source-code format.
- BSD UNIX is not open-source, however, because a license is still needed from ATT.
- In spite of various lawsuits, there are now several versions of BSD UNIX, including Free BSD, NetBSD, OpenBSD, and DragonflyBSD
- The source code is located in /usr/src.
- The core of the Mac operating system is Darwin, derived from BSD UNIX, and is available at http://developer.apple.com/opensource/index.html

1.13.4 Solaris

- Solaris is the UNIX operating system for computers from Sun Microsystems.
- Solaris was originally based on BSD UNIX, and has since migrated to ATT SystemV as its basis.
- Parts of Solaris are now open-source, and some are not because they are still covered by ATT copyrights.
- It is possible to change the open-source components of Solaris, re-compile them, and then link them in with binary libraries of the copyrighted portions of Solaris.
- Open Solaris is available from http://www.opensolaris.org/os/
- Solaris also allows viewing of the source code online, without having to download and unpack the entire package.

1.13.5 Utility

- The free software movement is gaining rapidly in popularity, leading to thousands of ongoing projects involving untold numbers of programmers.
- Sites such as http://freshmeat.net/ and http://distrowatch.com/ provide portals to many of these projects.

1.14 Summary

Operating-System Structures

References:

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 2

This chapter deals with how operating systems are structured and organized. Different design issues and choices are examined and compared, and the basic structure of several popular OSes are presented.

2.1 Operating-System Services
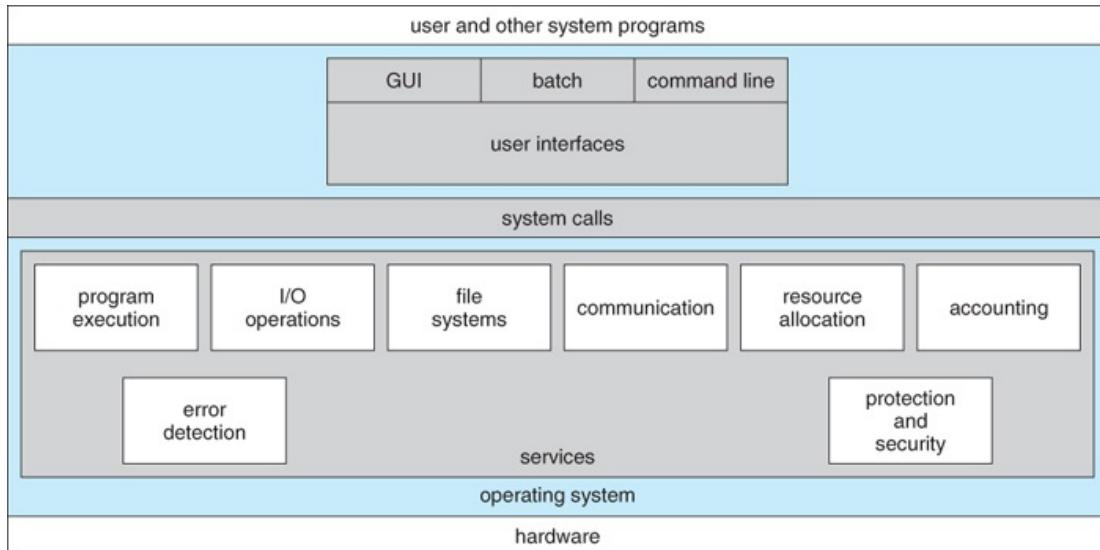
Figure 2.1 - A view of operating system services

OSes provide environments in which programs run, and services for the users of the system, including:

- **User Interfaces** - Means by which users can issue commands to the system. Depending on the system these may be a command-line interface (e.g. sh, csh, ksh, tcsh, etc.), a GUI interface (e.g. Windows, X-Windows, KDE, Gnome, etc.), or a batch command systems. The latter are generally older systems using punch cards of job-control language, JCL, but may still be used today for specialty systems designed for a single purpose.
- **Program Execution** - The OS must be able to load a program into RAM, run the program, and terminate the program, either normally or abnormally.
- **I/O Operations** - The OS is responsible for transferring data to and from I/O devices, including keyboards, terminals, printers, and storage devices.
- **File-System Manipulation** - In addition to raw data storage, the OS is also responsible for maintaining directory and subdirectory structures, mapping file names to specific blocks of data storage, and providing tools for navigating and utilizing the file system.
- **Communications** - Inter-process communications, IPC, either between processes running on the same processor, or between processes running on separate processors or separate machines. May be implemented as either shared memory or message passing, (or some systems may offer both.)
- **Error Detection** - Both hardware and software errors must be detected and handled appropriately, with a minimum of harmful repercussions. Some systems may include complex error avoidance or recovery systems, including backups, RAID drives, and other redundant systems. Debugging and diagnostic tools aid users and administrators in tracing down the cause of problems.
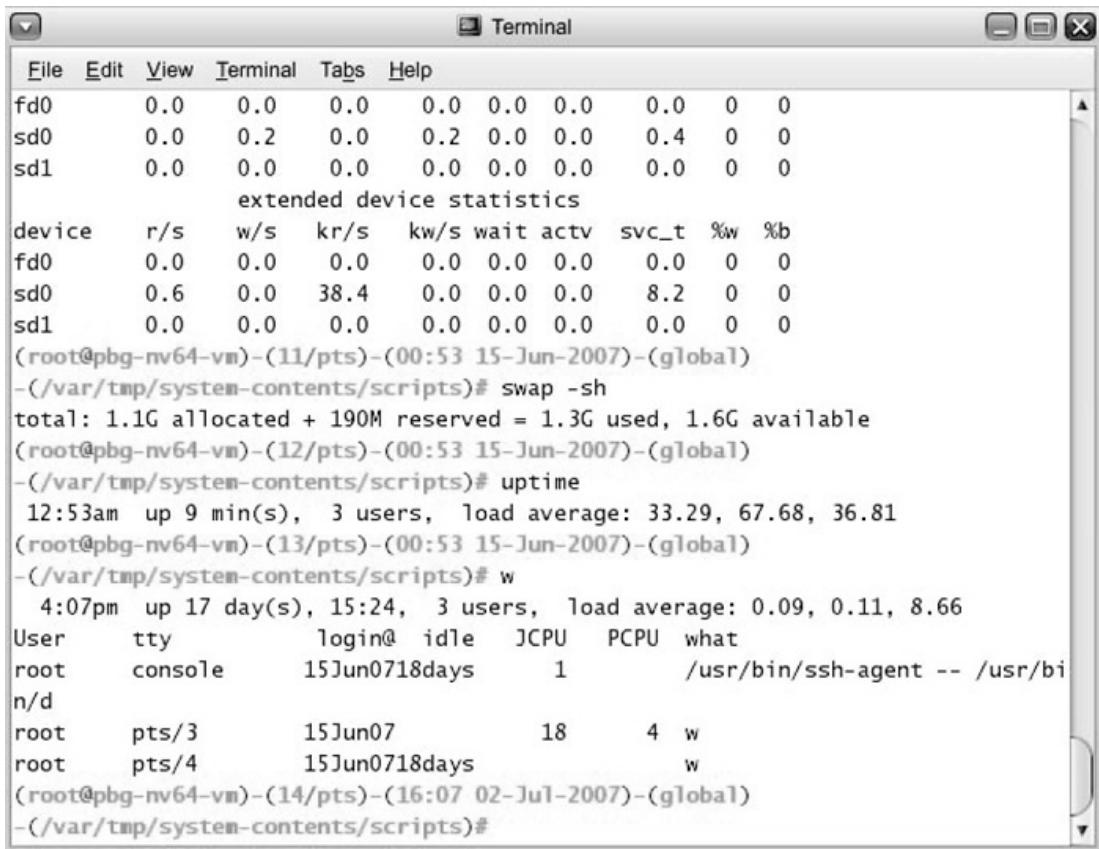
Other systems aid in the efficient operation of the OS itself:

- **Resource Allocation** - E.g. CPU cycles, main memory, storage space, and peripheral devices. Some resources are managed with generic systems and others with very carefully designed and specially tuned systems, customized for a particular resource and operating environment.
- **Accounting** - Keeping track of system activity and resource usage, either for billing purposes or for statistical record keeping that can be used to optimize future performance.
- **Protection and Security** - Preventing harm to the system and to resources, either through wayward internal processes or malicious outsiders. Authentication, ownership, and restricted access are obvious parts of this system. Highly secure systems may log all process activity down to excruciating detail, and security regulation dictate the storage of those records on permanent non-erasable medium for extended times in secure (off-site) facilities.

2.2 User Operating-System Interface

2.2.1 Command Interpreter

- Gets and processes the next user request, and launches the requested programs.
- In some systems the CI may be incorporated directly into the kernel.
- More commonly the CI is a separate program that launches once the user logs in or otherwise accesses the system.
- UNIX, for example, provides the user with a choice of different shells, which may either be configured to launch automatically at login, or which may be changed on the fly. (Each of these shells uses a different configuration file of initial settings and commands that are executed upon startup.)
- Different shells provide different functionality, in terms of certain commands that are implemented directly by the shell without launching any external programs. Most provide at least a rudimentary command interpretation structure for use in shell script programming (loops, decision constructs, variables, etc.)
- An interesting distinction is the processing of wild card file naming and I/O re-direction. On UNIX systems those details are handled by the shell, and the program which is launched sees only a list of filenames generated by the shell from the wild cards. On a DOS system, the wild cards are passed along to the programs, which can interpret the wild cards as the program sees fit.

Figure 2.2 - The Bourne shell command interpreter in Solaris 10

2.2.2 Graphical User Interface, GUI

- Generally implemented as a desktop metaphor, with file folders, trash cans, and resource icons.
- Icons represent some item on the system, and respond accordingly when the icon is activated.
- First developed in the early 1970's at Xerox PARC research facility.
- In some systems the GUI is just a front end for activating a traditional command line interpreter running in the background. In others the GUI is a true graphical shell in its own right.
- Mac has traditionally provided ONLY the GUI interface. With the advent of OSX (based partially on UNIX), a command line interface has also become available.
- Because mice and keyboards are impractical for small mobile devices, these normally use a touch-screen interface today, that responds to various patterns of swipes or "gestures". When these first came out they often had a physical keyboard and/or a trackball of some kind built in, but today a virtual keyboard is more commonly implemented on the touch screen.
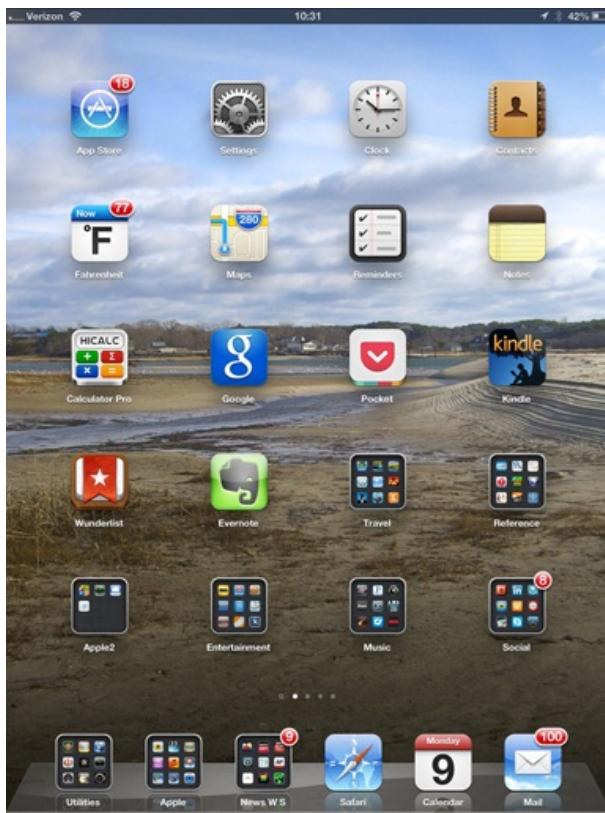
Figure 2.3 - The iPad touchscreen

2.2.3 Choice of interface

- Most modern systems allow individual users to select their desired interface, and to customize its operation, as well as the ability to switch between different interfaces as needed. System administrators generally determine which interface a user starts with when they first log in.
- GUI interfaces usually provide an option for a terminal emulator window for entering command-line commands.
- Command-line commands can also be entered into **shell scripts**, which can then be run like any other programs.

Figure 2.4 - The Mac OS X GUI

2.3 System Calls

- System calls provide a means for user or application programs to call upon the services of the operating system.
- Generally written in C or C++, although some are written in assembly for optimal performance.
- Figure 2.4 illustrates the sequence of system calls required to copy a file:


```
       Example System Call Sequence
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

Figure 2.5 - Example of how system calls are used.

- You can use "strace" to see more examples of the large number of system calls invoked by a single simple command. Read the man page for strace, and try some simple examples. (strace mkdir temp, strace cd temp, strace date > t.t, strace

cp t.t t.2, etc.)

- Most programmers do not use the low-level system calls directly, but instead use an "Application Programming Interface", API. The following sidebar shows the read( ) call available in the API on UNIX based systems::

EXAMPLE OF STANDARD API

As an example of a standard API, consider the read( ) function that is available in UNIX and Linux systems. The API for this function is obtained from the man page by invoking the command

    man read

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t      read(int fd, void *buf, size_t count)
  |_____|    |____|  |_____|
   return      function           parameters
   value         name
```

A program that uses the read( ) function must include the unistd.h header file, as this file defines the ssize_t and size_t data types (among other things.) The parameters passed to read( ) are as follows:

- int fd - The file descriptor to be read
- void *buf - A buffer where the data will be read into
- size_t count - The maximum number of bytes to be read into the buffer.

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, read( ) returns -1.

The use of APIs instead of direct system calls provides for greater program portability between different systems. The API then makes the appropriate system calls through the **system call interface**, using a table lookup to access specific numbered system calls, as shown in Figure 2.6:
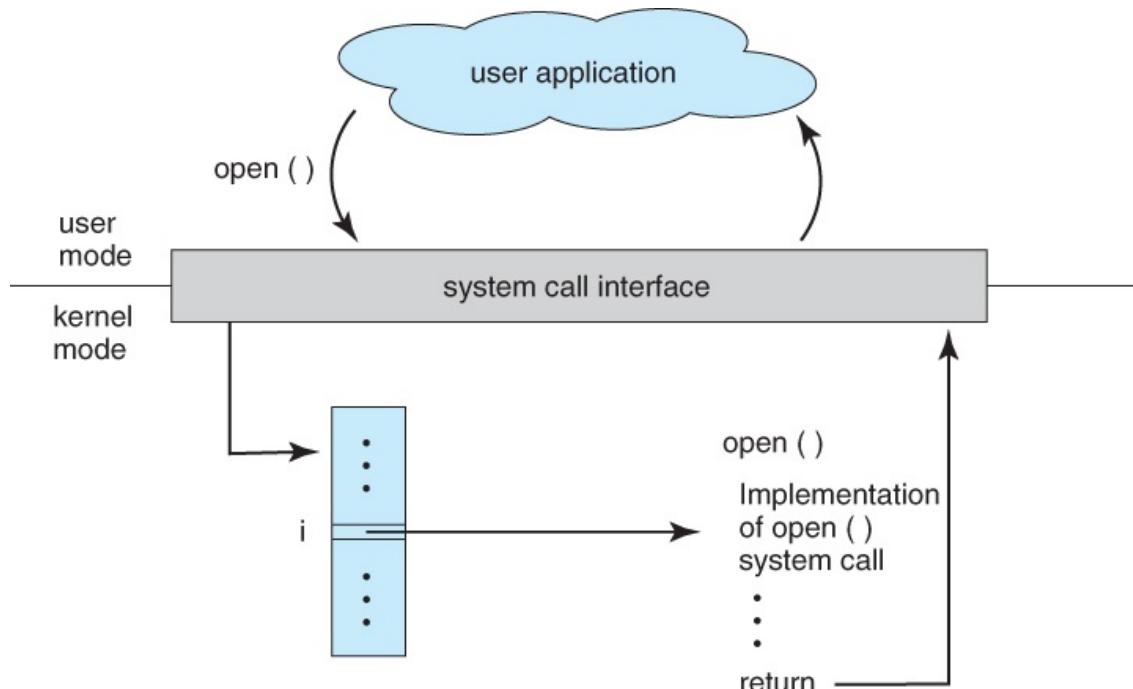
Figure 2.6 - The handling of a user application invoking the open( ) system call

- Parameters are generally passed to system calls via registers, or less commonly, by values pushed onto the stack. Large blocks of data are generally accessed indirectly, through a memory address passed in a register or on the stack, as shown in Figure 2.7:
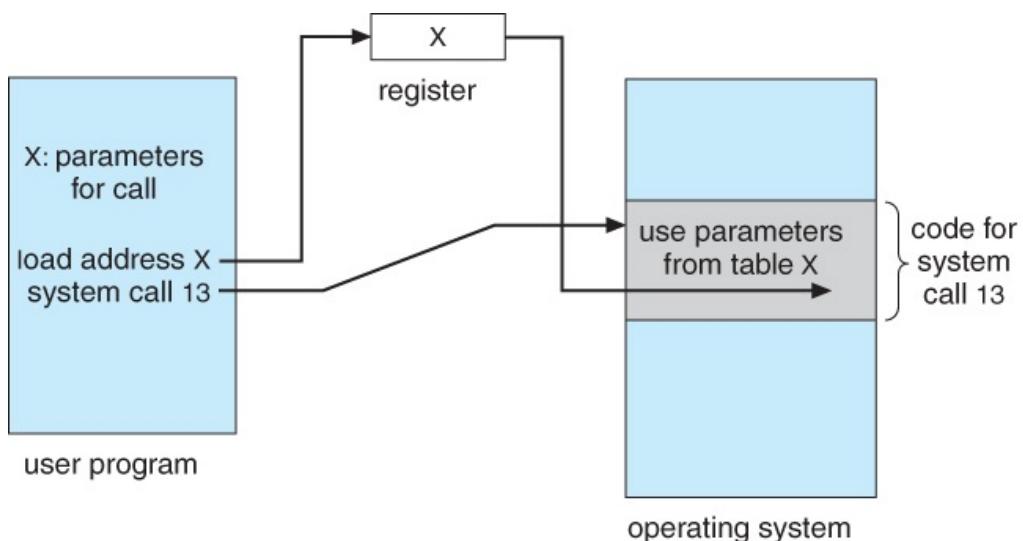
Figure 2.7 - Passing of parameters as a table

2.4 Types of System Calls

Six major categories, as outlined in Figure 2.8 and the following six subsections:

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices

Figure 2.8 Types of system calls.
(Sixth type, protection, not shown here but described below.)
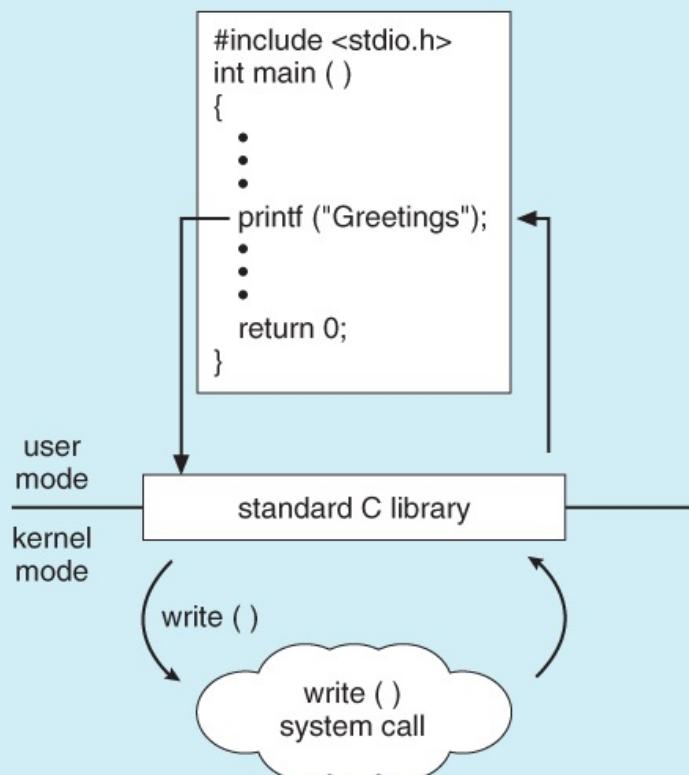
EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

- Standard library calls may also generate system calls, as shown here:

EXAMPLE OF STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. For example, let's assume a C program invokes the printf( ) statement. The C library intercepts this call and invokes the necessary ssystem call(s) in the operating system - in this instance, the write( ) system call. The C library takes the value returned by write( ) and passes it back to the user program. This is shown below:

```
#include <stdio.h>
int main ( )
{
    •
    •
    •
    printf ("Greetings");
    •
    •
    •
    return 0;
}
```

user
mode

_____ standard C library _____

kernel
mode

write ( )

write ( )
system call

2.4.1 Process Control

- Process control system calls include end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, and allocate and free memory.
- Processes must be created, launched, monitored, paused, resumed,and eventually stopped.
- When one process pauses or stops, then another must be launched or resumed
- When processes stop abnormally it may be necessary to provide core dumps and/or other diagnostic or recovery tools.
- Compare DOS (a single-tasking system) with UNIX (a multi-tasking system).
    - When a process is launched in DOS, the command interpreter first unloads as much of itself as it can to free up memory, then loads the process and transfers control to it. The interpreter does not resume until the process has completed, as shown in Figure 2.9:
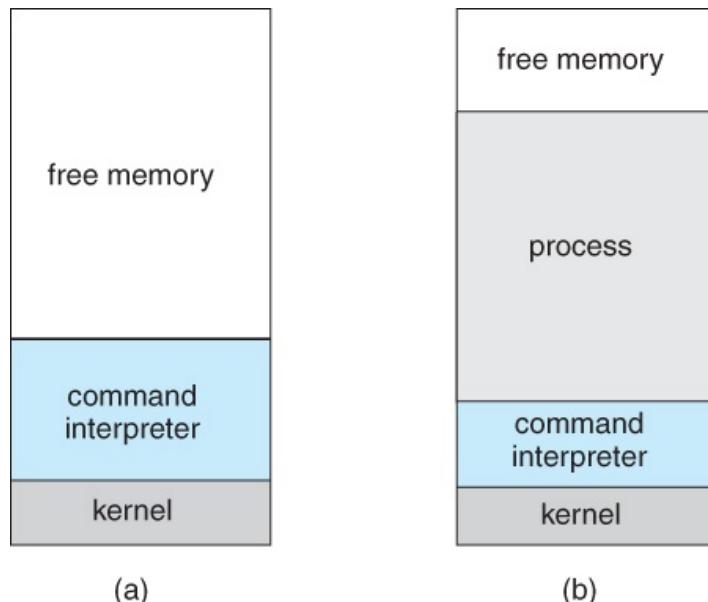
Figure 2.9 - MS-DOS execution. (a) At system startup. (b) Running a program.

- Because UNIX is a multi-tasking system, the command interpreter remains completely resident when executing a process, as shown in Figure 2.11 below.
    - The user can switch back to the command interpreter at any time, and can place the running process in the background even if it was not originally launched as a background process.
    - In order to do this, the command interpreter first executes a "fork" system call, which creates a second process which is an exact duplicate (clone) of the original command interpreter. The original process is known as the parent, and the cloned process is known as the child, with its own unique process ID and parent ID.
    - The child process then executes an "exec" system call, which replaces its code with that of the desired process.
    - The parent (command interpreter) normally waits for the child to complete before issuing a new command prompt, but in some cases it can also issue a new prompt right away, without waiting for the child process to complete. (The child is then said to be running "in the background", or "as a background process".)
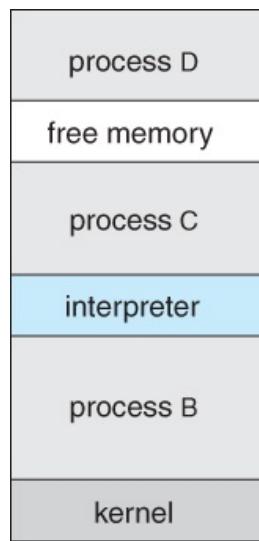
Figure 2.10 - FreeBSD running multiple programs

2.4.2 File Management

- File management system calls include create file, delete file, open, close, read, write, reposition, get file attributes, and set file attributes.
- These operations may also be supported for directories as well as ordinary files.
- (The actual directory structure may be implemented using ordinary files on the file system, or through other means. Further details will be covered in chapters 11 and 12.)

2.4.3 Device Management

- Device management system calls include request device, release device, read, write, reposition, get/set device attributes, and logically attach or detach devices.
- Devices may be physical (e.g. disk drives), or virtual / abstract (e.g. files, partitions, and RAM disks).
- Some systems represent devices as special files in the file system, so that accessing the "file" calls upon the appropriate device drivers in the OS. See for example the /dev directory on any UNIX system.

2.4.4 Information Maintenance

- Information maintenance system calls include calls to get/set the time, date, system data, and process, file, or device attributes.
- Systems may also provide the ability to dump memory at any time, single step programs pausing execution after each instruction, and tracing the operation of programs, all of which can help to debug programs.

2.4.5 Communication

- Communication system calls create/delete communication connection, send/receive messages, transfer status information, and attach/detach remote devices.
- The **message passing** model must support calls to:
    - Identify a remote process and/or host with which to communicate.
    - Establish a connection between the two processes.
    - Open and close the connection as needed.
    - Transmit messages along the connection.
    - Wait for incoming messages, in either a blocking or non-blocking state.
    - Delete the connection when no longer needed.
- The **shared memory** model must support calls to:
    - Create and access memory that is shared amongst processes (and threads.)
    - Provide locking mechanisms restricting simultaneous access.
    - Free up shared memory and/or dynamically allocate it as needed.
- Message passing is simpler and easier, (particularly for inter-computer communications), and is generally appropriate for small amounts of data.
- Shared memory is faster, and is generally the better approach where large amounts of data are to be shared, (particularly when most processes are reading the data rather than writing it, or at least when only one or a small number of processes need to change any given data item.)

2.4.6 Protection

- Protection provides mechanisms for controlling which users / processes have access to which system resources.
- System calls allow the access mechanisms to be adjusted as needed, and for non-priveleged users to be granted elevated access permissions under carefully controlled temporary circumstances.
- Once only of concern on multi-user systems, protection is now important on all systems, in the age of ubiquitous network connectivity.

2.5 System Programs

- System programs provide OS functionality through separate applications, which are not part of the kernel or command interpreters. They are also known as system utilities or system applications.
- Most systems also ship with useful applications such as calculators and simple editors, (e.g. Notepad). Some debate arises as to the border between system and non-system applications.
- System programs may be divided into these categories:
    - **File management** - programs to create, delete, copy, rename, print, list, and generally manipulate files and directories.
    - **Status information** - Utilities to check on the date, time, number of users, processes running, data logging, etc. System **registries** are used to store and recall configuration information for particular applications.
    - **File modification** - e.g. text editors and other tools which can change file contents.
    - **Programming-language support** - E.g. Compilers, linkers, debuggers, profilers, assemblers, library archive management, interpreters for common languages, and support for make.
    - **Program loading and execution** - loaders, dynamic loaders, overlay loaders, etc., as well as interactive debuggers.
    - **Communications** - Programs for providing connectivity between processes and users, including mail, web browsers, remote logins, file transfers, and remote command execution.
    - **Background services** - System daemons are commonly started when the system is booted, and run for as long as the system is running, handling necessary services. Examples include network daemons, print servers, process

schedulers, and system error monitoring services.

- Most operating systems today also come complete with a set of **application programs** to provide additional services, such as copying files or checking the time and date.
- Most users' views of the system is determined by their command interpreter and the application programs. Most never make system calls, even through the API, (with the exception of simple (file) I/O in user-written programs.)

2.6 Operating-System Design and Implementation

2.6.1 Design Goals

- **Requirements** define properties which the finished system must have, and are a necessary first step in designing any large complex system.
  - **User requirements** are features that users care about and understand, and are written in commonly understood vernacular. They generally do not include any implementation details, and are written similar to the product description one might find on a sales brochure or the outside of a shrink-wrapped box.
  - **System requirements** are written for the developers, and include more details about implementation specifics, performance requirements, compatibility constraints, standards compliance, etc. These requirements serve as a "contract" between the customer and the developers, (and between developers and subcontractors), and can get quite detailed.
- Requirements for operating systems can vary greatly depending on the planned scope and usage of the system. (Single user / multi-user, specialized system / general purpose, high/low security, performance needs, operating environment, etc.)

2.6.2 Mechanisms and Policies

- Policies determine *what* is to be done. Mechanisms determine *how* it is to be implemented.
- If properly separated and implemented, policy changes can be easily adjusted without re-writing the code, just by adjusting parameters or possibly loading new data / configuration files. For example the relative priority of background versus foreground tasks.

2.6.3 Implementation

- Traditionally OSes were written in assembly language. This provided direct control over hardware-related issues, but inextricably tied a particular OS to a particular HW platform.
- Recent advances in compiler efficiencies mean that most modern OSes are written in C, or more recently, C++. Critical sections of code are still written in assembly language, (or written in C, compiled to assembly, and then fine-tuned and optimized by hand from there.)
- Operating systems may be developed using **emulators** of the target hardware, particularly if the real hardware is unavailable (e.g. not built yet), or not a suitable platform for development, (e.g. smart phones, game consoles, or other similar devices.)

2.7 Operating-System Structure

For efficient performance and implementation an OS should be partitioned into separate subsystems, each with carefully defined tasks, inputs, outputs, and performance characteristics. These subsystems can then be arranged in various architectural configurations:

2.7.1 Simple Structure

When DOS was originally written its developers had no idea how big and important it would eventually become. It was written by a few programmers in a relatively short amount of time, without the benefit of modern software engineering techniques, and then gradually grew over time to exceed its original expectations. It does not break the system into subsystems, and has no distinction between user and kernel modes, allowing all programs direct access to the underlying hardware. (Note that user versus kernel mode was not supported by the 8088 chip set anyway, so that really wasn't an option back then.)
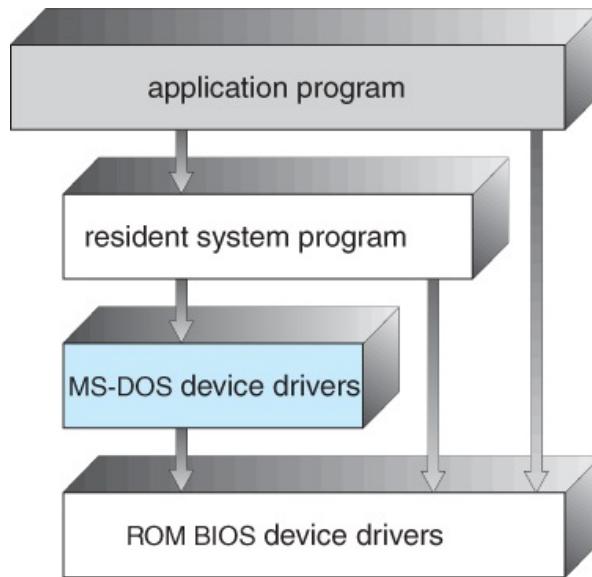
Figure 2.11 - MS-DOS layer structure

The original UNIX OS used a simple layered approach, but almost all the OS was in one big layer, not really breaking the OS down into layered subsystems:
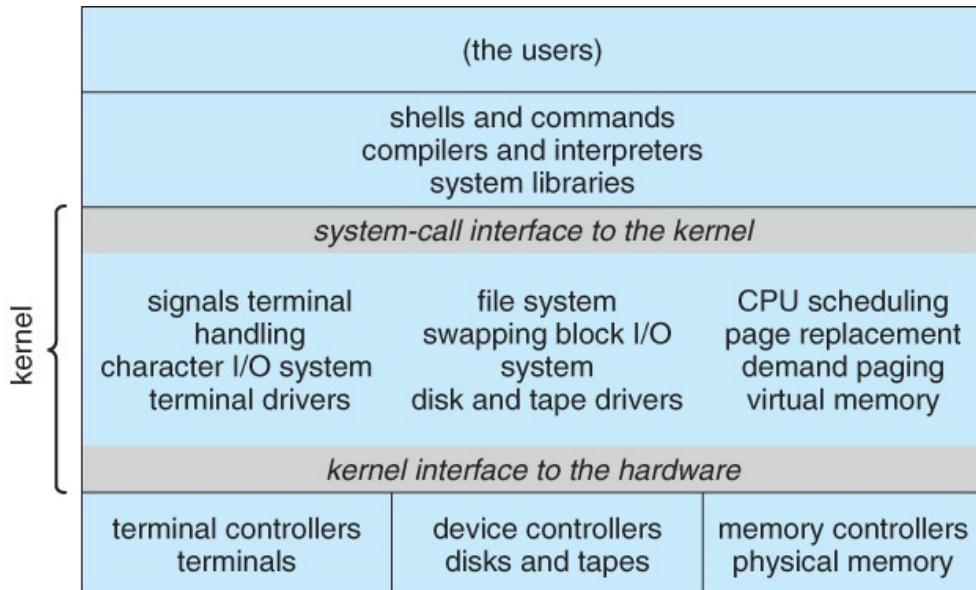
Figure 2.12 - Traditional UNIX system structure

2.7.2 Layered Approach

- Another approach is to break the OS into a number of smaller layers, each of which rests on the layer below it, and relies solely on the services provided by the next lower layer.
- This approach allows each layer to be developed and debugged independently, with the assumption that all lower layers have already been debugged and are trusted to deliver proper services.
- The problem is deciding what order in which to place the layers, as no layer can call upon the services of any higher layer, and so many chicken-and-egg situations may arise.
- Layered approaches can also be less efficient, as a request for service from a higher layer has to filter through all lower layers before it reaches the HW, possibly with significant processing at each step.
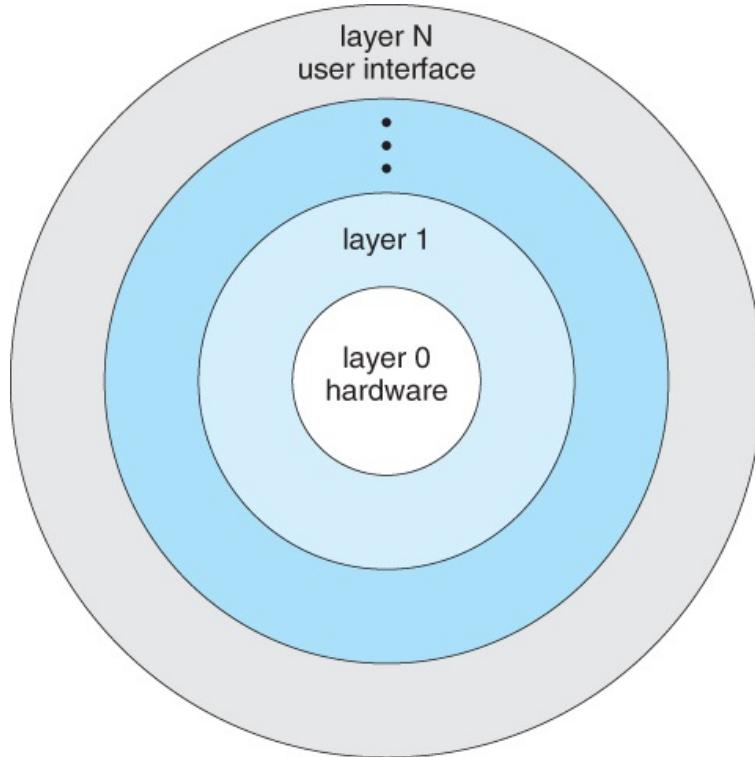
Figure 2.13 - A layered operating system

2.7.3 Microkernels

- The basic idea behind micro kernels is to remove all non-essential services from the kernel, and implement them as system applications instead, thereby making the kernel as small and efficient as possible.
- Most microkernels provide basic process and memory management, and message passing between other services, and not much more.
- Security and protection can be enhanced, as most services are performed in user mode, not kernel mode.
- System expansion can also be easier, because it only involves adding more system applications, not rebuilding a new kernel.
- Mach was the first and most widely known microkernel, and now forms a major component of Mac OSX.
- Windows NT was originally microkernel, but suffered from performance problems relative to Windows 95. NT 4.0 improved performance by moving more services into the kernel, and now XP is back to being more monolithic.
- Another microkernel example is QNX, a real-time OS for embedded systems.
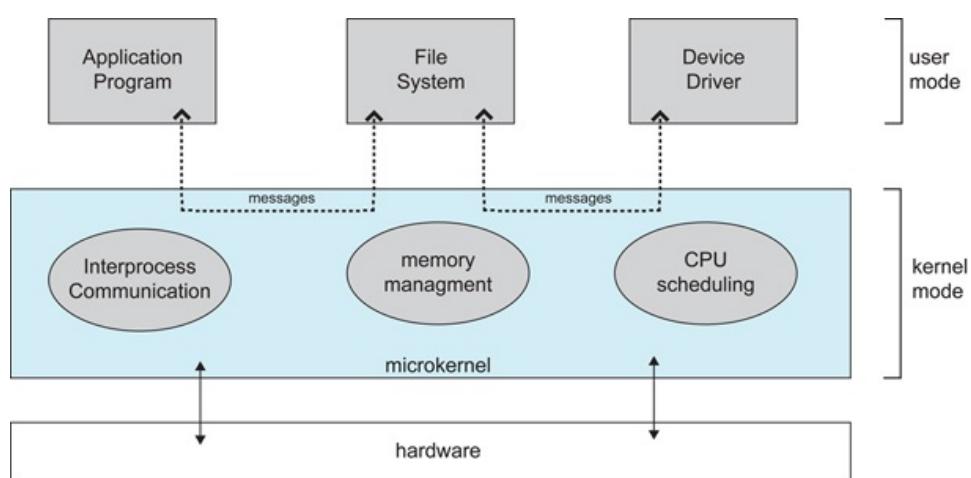
Figure 2.14 - Architecture of a typical microkernel

2.7.4 Modules

- Modern OS development is object-oriented, with a relatively small core kernel and a set of *modules* which can be linked in dynamically. See for example the Solaris structure, as shown in Figure 2.13 below.
- Modules are similar to layers in that each subsystem has clearly defined tasks and interfaces, but any module is free to contact any other module, eliminating the problems of going through multiple intermediary layers, as well as the chicken-and-egg problems.

- The kernel is relatively small in this architecture, similar to microkernels, but the kernel does not have to implement message passing since modules are free to contact each other directly.
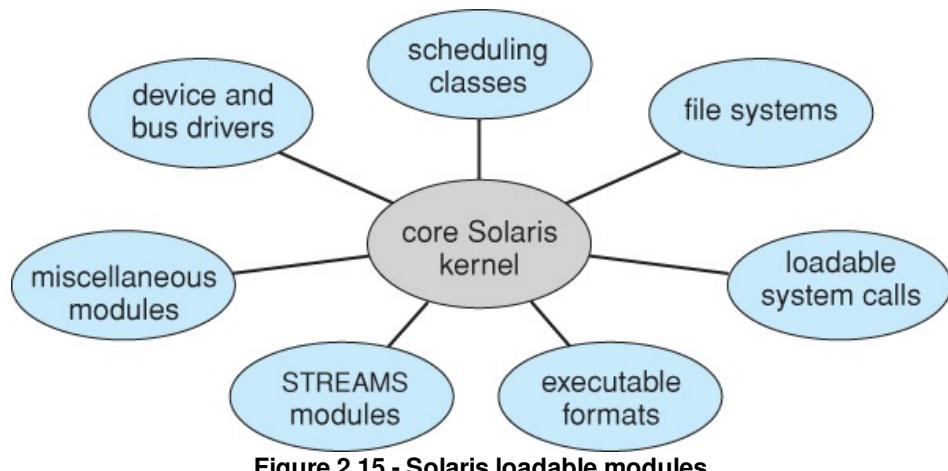
Figure 2.15 - Solaris loadable modules

2.7.5 Hybrid Systems

- Most OSes today do not strictly adhere to one architecture, but are hybrids of several.

2.7.5.1 Mac OS X

- The Max OSX architecture relies on the Mach microkernel for basic system management services, and the BSD kernel for additional services. Application services and dynamically loadable modules (kernel extensions) provide the rest of the OS functionality:
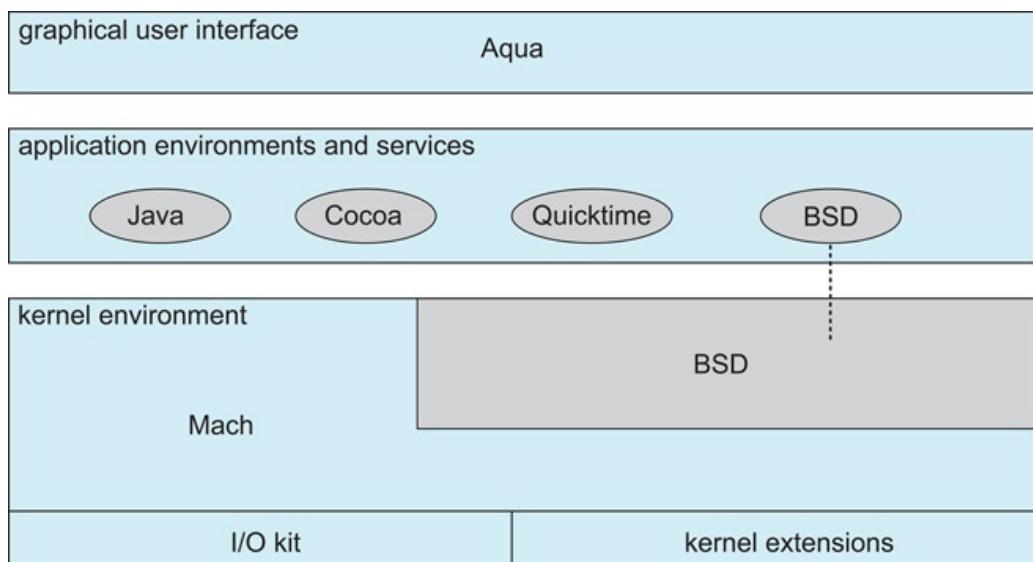
Figure 2.16 - The Mac OS X structure

2.7.5.2 iOS

- The **iOS** operating system was developed by Apple for iPhones and iPads. It runs with less memory and computing power needs than Max OS X, and supports touchscreen interface and graphics for small screens:

Figure 2.17 - Architecture of Apple's iOS.

2.7.5.3 Android

- The Android OS was developed for Android smartphones and tablets by the Open Handset Alliance, primarily Google.
- Android is an open-source OS, as opposed to iOS, which has lead to its popularity.
- Android includes versions of Linux and a Java virtual machine both optimized for small platforms.
- Android apps are developed using a special Java-for-Android development environment.
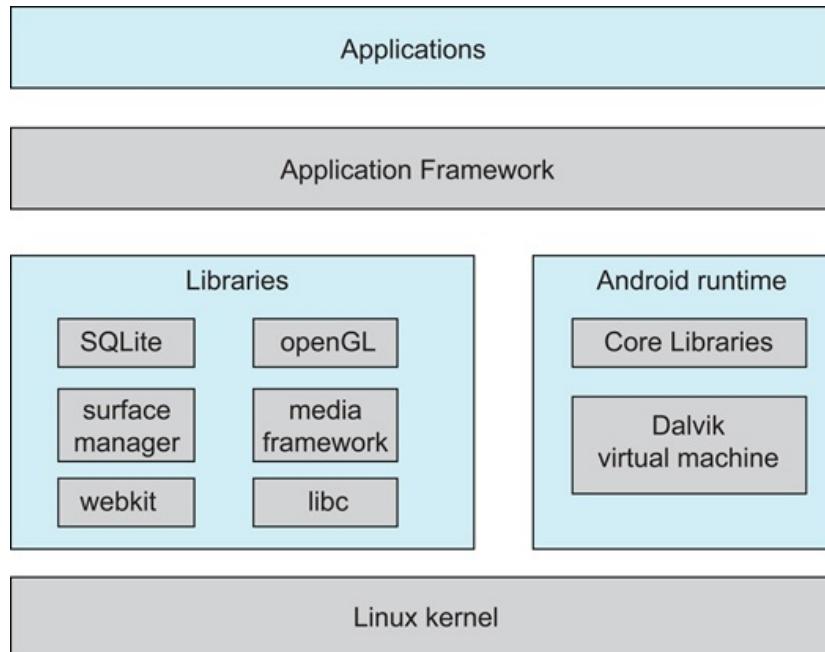
**Figure 2.18 - Architecture of Google's Android**

2.8 Operating-System Debugging

Kernighan's Law

"Debugging is twice as hard as writing the code in the first place. Therefore,
if you write the code as cleverly as possible, you are, by definition, not smart
enough to debug it."

- Debugging here includes both error discovery and elimination and performance tuning.

2.8.1 Failure Analysis

- Debuggers allow processes to be executed stepwise, and provide for the examination of variables and expressions as the execution progresses.
- Profilers can document program execution, to produce statistics on how much time was spent on different sections or even lines of code.
- If an ordinary process crashes, a memory dump of the state of that process's memory at the time of the crash can be saved to a disk file for later analysis.
  - The program must be specially compiled to include debugging information, which may slow down its performance.

- These approaches don't really work well for OS code, for several reasons:
  - The performance hit caused by adding the debugging (tracing) code would be unacceptable. (Particularly if one tried to "single-step" the OS while people were trying to use it to get work done!)
  - Many parts of the OS run in kernel mode, and make direct access to the hardware.
  - If an error occurred during one of the kernel's file-access or direct disk-access routines, for example, then it would not be practical to try to write a crash dump into an ordinary file on the filesystem.
    - Instead the kernel crash dump might be saved to a special unallocated portion of the disk reserved for that purpose.

2.8.2 Performance Tuning

- Performance tuning (debottlenecking) requires monitoring system performance.
- One approach is for the system to record important events into log files, which can then be analyzed by other tools. These traces can also be used to evaluate how a proposed new system would perform under the same workload.
- Another approach is to provide utilities that will report system status upon demand, such as the unix "top" command. (w, uptime, ps, etc.)
- System utilities may provide monitoring support.
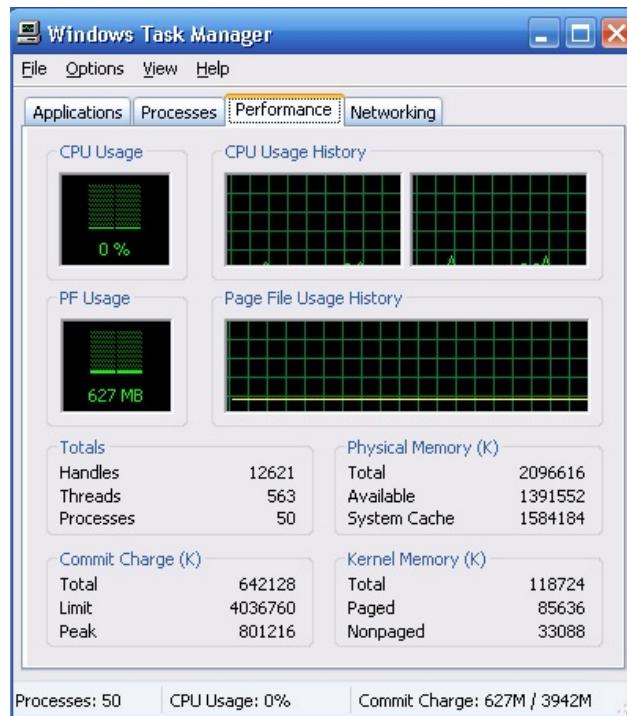
Figure 2.19 - The Windows task manager.

2.8.3 DTrace

- DTrace is a special facility for tracing a running OS, developed for Solaris 10.
- DTrace adds "probes" directly into the OS code, which can be queried by "probe consumers".
- Probes are removed when not in use, so the DTrace facility has zero impact on the system when not being used, and a proportional impact in use.
- Consider, for example, the trace of an ioctl system call as shown in Figure 2.22 below.

```
# ./all.d `pgrep xclock` XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0  -> XEventsQueued                            U
  0    -> _XEventsQueued                         U
  0      -> _X11TransBytesReadable               U
  0      <- _X11TransBytesReadable               U
  0      -> _X11TransSocketBytesReadable U
  0      <- _X11TransSocketBytesreadable U
  0      -> ioctl                                U
  0        -> ioctl                              K
  0          -> getf                             K
  0            -> set_active_fd                  K
  0            <- set_active_fd                  K
  0          <- getf                             K
  0          -> get_udatamodel                  K
  0          <- get_udatamodel                  K
...
  0          -> releasef                         K
  0            -> clear_active_fd                K
  0            <- clear_active_fd                K
  0            -> cv_broadcast                   K
  0            <- cv_broadcast                   K
  0          <- releasef                         K
  0        <- ioctl                              K
  0      <- ioctl                                U
  0    <- _XEventsQueued                         U
  0  <- XEventsQueued                            U
```

Figure 2.20 - Solaris 10 dtrace follows a system call within the kernel

- Probe code is restricted to be "safe", (e.g. no loops allowed), and to use a minimum of system resources.
- When a probe fires, **enabling control blocks, ECBs,** are performed, each having the structure of an if-then block
- When a consumer terminates, the ECBs associated with that consumer are removed. When no more ECBs remain interested in a particular probe, then that probe is also removed.
- For example, the following D code monitors the CPU time of each process running with user ID of 101. The output is shown in Figure 2.23 below.

```
sched:::on-cpu
uid == 101
{
     self->ts = timestamp;

}

sched:::off-cpu
self->ts
{
     @time[execname] = sum( timestamp - self->ts );
     self->ts = 0;

}
```

```
# dtrace -s sched.d
dtrace: script 'sched.d' matched 6 probes
^C
        gnome-settings-d                142354
        gnome-vfs-daemon                158243
        dsdm                            189804
        wnck-applet                     200030
        gnome-panel                     277864
        clock-applet                    374916
        mapping-daemon                  385475
        xscreensaver                    514177
        metacity                        539281
        Xorg                           2579646
        gnome-terminal                 5007269
        mixer_applet2                  7388447
        java                          10769137
```

Figure 2.23 Output of the D code.

Figure 2.21

- Use of DTrace is restricted, due to the direct access to (and ability to change) critical kernel data structures.
- Because DTrace is open-source, it is being adopted by several UNIX distributions. Others are busy producing similar utilities.

2.9 Operating-System Generation

- OSes may be designed and built for a specific HW configuration at a specific site, but more commonly they are designed with a number of variable parameters and components, which are then configured for a particular operating environment.
- Systems sometimes need to be re-configured after the initial installation, to add additional resources, capabilities, or to tune performance, logging, or security.
- Information that is needed to configure an OS include:
    - What CPU(s) are installed on the system, and what optional characteristics does each have?
    - How much RAM is installed? (This may be determined automatically, either at install or boot time.)
    - What devices are present? The OS needs to determine which device drivers to include, as well as some device-specific characteristics and parameters.
    - What OS options are desired, and what values to set for particular OS parameters. The latter may include the size of the open file table, the number of buffers to use, process scheduling (priority) parameters, disk scheduling algorithms, number of slots in the process table, etc.

- At one extreme the OS source code can be edited, re-compiled, and linked into a new kernel.
- More commonly configuration tables determine which modules to link into the new kernel, and what values to set for some key important parameters. This approach may require the configuration of complicated makefiles, which can be done either automatically or through interactive configuration programs; Then make is used to actually generate the new kernel specified by the new parameters.
- At the other extreme a system configuration may be entirely defined by table data, in which case the "rebuilding" of the system merely requires editing data tables.
- Once a system has been regenerated, it is usually required to reboot the system to activate the new kernel. Because there are possibilities for errors, most systems provide some mechanism for booting to older or alternate kernels.

2.10 System Boot

The general approach when most computers boot up goes something like this:

- When the system powers up, an interrupt is generated which loads a memory address into the program counter, and the system begins executing instructions found at that address. This address points to the "bootstrap" program located in ROM chips (or EPROM chips) on the motherboard.
- The ROM bootstrap program first runs hardware checks, determining what physical resources are present and doing power-on self tests (POST) of all HW for which this is applicable. Some devices, such as controller

cards may have their own on-board diagnostics, which are called by the ROM bootstrap program.

- The user generally has the option of pressing a special key during the POST process, which will launch the ROM BIOS configuration utility if pressed. This utility allows the user to specify and configure certain hardware parameters as where to look for an OS and whether or not to restrict access to the utility with a password.
    - Some hardware may also provide access to additional configuration setup programs, such as for a RAID disk controller or some special graphics or networking cards.
- Assuming the utility has not been invoked, the bootstrap program then looks for a non-volatile storage device containing an OS. Depending on configuration, it may look for a floppy drive, CD ROM drive, or primary or secondary hard drives, in the order specified by the HW configuration utility.
- Assuming it goes to a hard drive, it will find the first sector on the hard drive and load up the fdisk table, which contains information about how the physical hard drive is divided up into logical partitions, where each partition starts and ends, and which partition is the "active" partition used for booting the system.
- There is also a very small amount of system code in the portion of the first disk block not occupied by the fdisk table. This bootstrap code is the first step that is not built into the hardware, i.e. the first part which might be in any way OS-specific. Generally this code knows just enough to access the hard drive, and to load and execute a (slightly) larger boot program.
- For a single-boot system, the boot program loaded off of the hard disk will then proceed to locate the kernel on the hard drive, load the kernel into memory, and then transfer control over to the kernel. There may be some opportunity to specify a particular kernel to be loaded at this stage, which may be useful if a new kernel has just been generated and doesn't work, or if the system has multiple kernels available with different configurations for different purposes. (Some systems may boot different configurations automatically, depending on what hardware has been found in earlier steps.)
- For dual-boot or multiple-boot systems, the boot program will give the user an opportunity to specify a particular OS to load, with a default choice if the user does not pick a particular OS within a given time frame. The boot program then finds the boot loader for the chosen single-boot OS, and runs that program as described in the previous bullet point.
- Once the kernel is running, it may give the user the opportunity to enter into single-user mode, also known as maintenance mode. This mode launches very few if any system services, and does not enable any logins other than the primary log in on the console. This mode is used primarily for system maintenance and diagnostics.
- When the system enters full multi-user multi-tasking mode, it examines configuration files to determine which system services are to be started, and launches each of them in turn. It then spawns login programs (gettys) on each of the login devices which have been configured to enable user logins.
    - (The getty program initializes terminal I/O, issues the login prompt, accepts login names and passwords, and authenticates the user. If the user's password is authenticated, then the getty looks in system files to determine what shell is assigned to the user, and then "execs" (becomes) the user's shell. The shell program will look in system and user configuration files to initialize itself, and then issue prompts for user commands. Whenever the shell dies, either through logout or other means, then the system will issue a new getty for that terminal device.)

2.11 Summary

Old 2.8 Virtual Machines (Moved elsewhere in the 9th edition.)

- The concept of a virtual machine is to provide an interface that looks like independent hardware, to multiple different OSes running simultaneously on the same physical hardware. Each OS believes that it has access to and control over its own CPU, RAM, I/O devices, hard drives, etc.
- One obvious use for this system is for the development and testing of software that must run on multiple platforms and/or OSes.
- One obvious difficulty involves the sharing of hard drives, which are generally partitioned into separate smaller virtual disks for each operating OS.
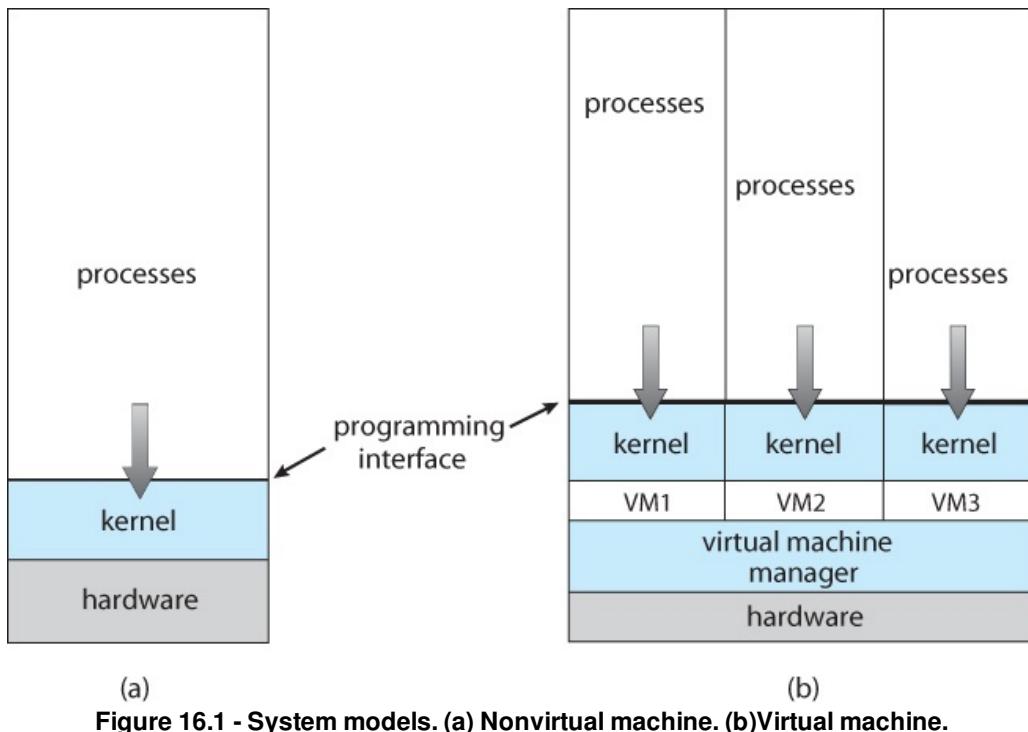
Figure 16.1 - System models. (a) Nonvirtual machine. (b)Virtual machine.

2.8.1 History

- Virtual machines first appeared as the VM Operating System for IBM mainframes in 1972.

2.8.2 Benefits

- Each OS runs independently of all the others, offering protection and security benefits.
- (Sharing of physical resources is not commonly implemented, but may be done as if the virtual machines were networked together.)
- Virtual machines are a very useful tool for OS development, as they allow a user full access to and control over a virtual machine, without affecting other users operating the real machine.
- As mentioned before, this approach can also be useful for product development and testing of SW that must run on multiple OSes / HW platforms.

2.8.3 Simulation

- An alternative to creating an entire virtual machine is to simply run an *emulator*, which allows a program written for one OS to run on a different OS.
- For example, a UNIX machine may run a DOS emulator in order to run DOS programs, or vice-versa.
- Emulators tend to run considerably slower than the native OS, and are also generally less than perfect.

2.8.4 Para-virtualization

- Para-virtualization is another variation on the theme, in which an environment is provided for the guest program that is *similar to* its native OS, without trying to completely mimic it.
- Guest programs must also be modified to run on the para-virtual OS.
- Solaris 10 uses a *zone* system, in which the low-level hardware is not virtualized, but the OS and its devices (device drivers) are.
    - Within a zone, processes have the view of an isolated system, in which only the processes and resources within that zone are seen to exist.
    - Figure 2.18 shows a Solaris system with the normal "global" operating space as well as two additional zones running on a small virtualization layer.
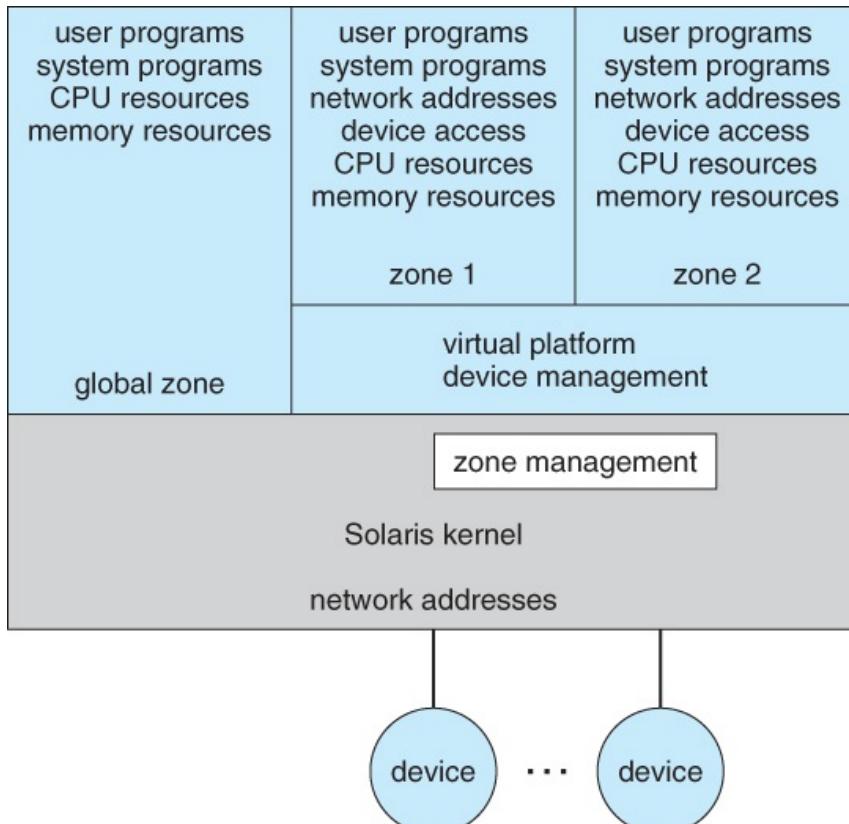
Figure 16.7 - Solaris 10 with two zones.

2.8.5 Implementation

- Implementation may be challenging, partially due to the consequences of user versus kernel mode.
  - Each of the simultaneously running kernels needs to operate in kernel mode at some point, but the virtual machine actually runs in user mode.
  - So the kernel mode has to be simulated for each of the loaded OSes, and kernel system calls passed through the virtual machine into a true kernel mode for eventual HW access.
- The virtual machines may run slower, due to the increased levels of code between applications and the HW, or they may run faster, due to the benefits of caching. (And virtual devices may also be faster than real devices, such as RAM disks which are faster than physical disks.)

2.8.6 Examples

2.8.6.1 VMware

- Abstracts the 80x86 hardware platform, allowing simultaneous operation of multiple Windows and Linux OSes, as shown by example in Figure 2.19:
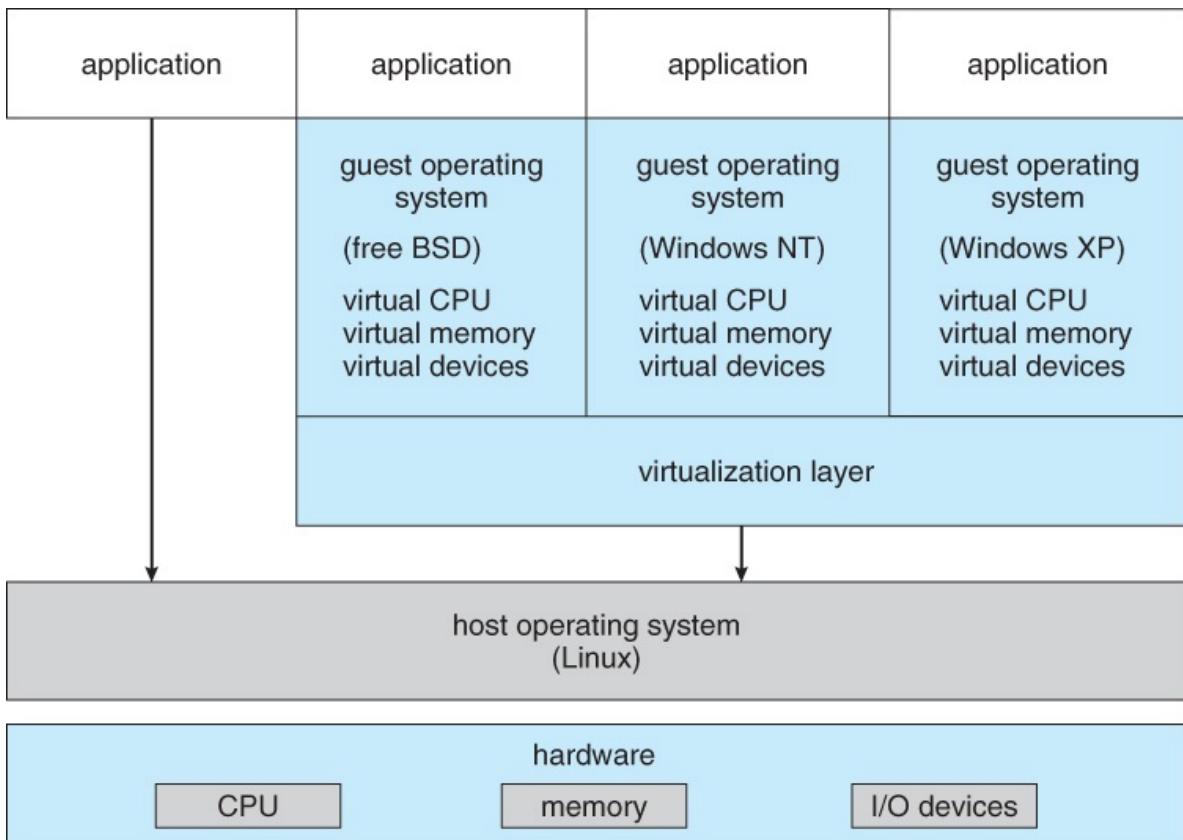
Figure 16.9 - VMWare Workstation architecture

2.8.6.2 The Java Virtual Machine

- Java was designed from the beginning to be platform independent, by running Java only on a Java Virtual Machine, JVM, of which different implementations have been developed for numerous different underlying HW platforms.
- Java source code is compiled into Java byte code in .class files. Java byte code is binary instructions that will run on the JVM.
- The JVM implements memory management and garbage collection.
- Java byte code may be interpreted as it runs, or compiled to native system binary code using just-in-time (JIT) compilation. Under this scheme, the first time that a piece of Java byte code is encountered, it is compiled to the appropriate native machine binary code by the Java interpreter. This native binary code is then cached, so that the next time that piece of code is encountered it can be used directly.
- Some hardware chips have been developed to run Java byte code directly, which is an interesting application of a real machine being developed to emulate the services of a virtual one!
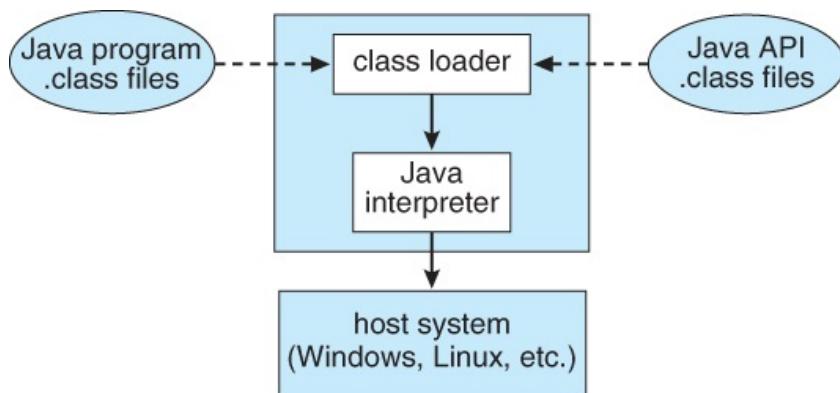

Figure 16.10 - The Java virtual machine

- The .NET framework also relies on the concept of compiling code for an intermediary virtual machine, (Common Language Runtime, CLR), and then using JIT compilation and caching to run the programs on specific hardware, as shown in Figure 2.21:

THE .NET FRAMEWORK

The .NET Framework is a collection of technologies, including a set of class libraries, and an execution environment that come together to provide a platform for developing software. This platform allows programs to be written to target the .NET Framework instead of a specific architecture. A program written for the .NET Framework need not worry about the specifics of the hardware or the operating system on which it will run. Thus, any architecture implementing .NET will be able to successfully execute the program. This is because the execution environment abstracts these details and provides a virtual machine as an intermediary between the executing program and the underlying architecture.

At the core of the .NET Framework is the Common Language Runtime (CLR). The CLR is the implementation of the .NET virtual machine. It provides an environment for execution of programs written in any of the languages targeted at the .NET Framework. Programs written in languages such as C# (pronounced *C-sharp*) and VB.NET are compiled into an intermediate, architecture-independent language called Microsoft Intermediate Language (MS-IL). These compiled files, called assemblies, include MS-IL instructions and metadata. They have a file extension of either .EXE or .DLL. Upon execution of a program, the CLR loads assemblies into what is known as the **Application Domain**. As instructions are requested by the executing program, the CLR converts the MS-IL instructions inside the assemblies into native code that is specific to the underlying architecture using just-in-time compilation. Once instructions have been converted to native code, they are kept and will continue to run as native code for the CPU. The architecture of the CLR for the .NET framework is shown in Figure 2.18.
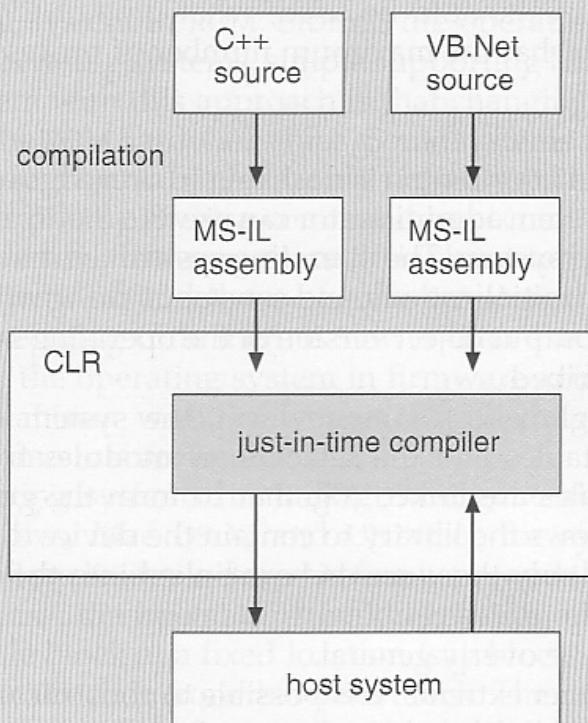

Figure 2.18 Architecture of the CLR for the .NET Framework.

Figure 2.21

Processes

References:

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 3

3.1 Process Concept

- A process is an instance of a program in execution.
- Batch systems work in terms of "jobs". Many modern process concepts are still expressed in terms of jobs, (e.g. job scheduling), and the two terms are often used interchangeably.

3.1.1 The Process

- Process memory is divided into four sections as shown in Figure 3.1 below:
    - The text section comprises the compiled program code, read in from non-volatile storage when the program is launched.
    - The data section stores global and static variables, allocated and initialized prior to executing main.
    - The heap is used for dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
    - The stack is used for local variables. Space on the stack is reserved for local variables when they are declared (at function entrance or elsewhere, depending on the language), and the space is freed up when the variables go out of scope. Note that the stack is also used for function return values, and the exact mechanisms of stack management may be language specific.
    - Note that the stack and the heap start at opposite ends of the process's free space and grow towards each other. If they should ever meet, then either a stack overflow error will occur, or else a call to new or malloc will fail due to insufficient memory available.
- When processes are swapped out of memory and later restored, additional information must also be stored and restored. Key among them are the program counter and the value of all program registers.
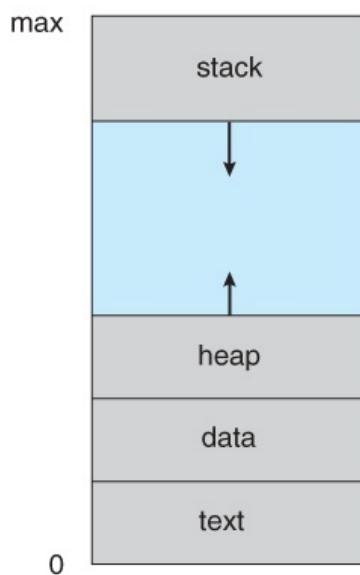

Figure 3.1 - A process in memory

3.1.2 Process State

- Processes may be in one of 5 states, as shown in Figure 3.2 below.
    - **New** - The process is in the stage of being created.
    - **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
    - **Running** - The CPU is working on this process's instructions.
    - **Waiting** - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur. For example the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.
    - **Terminated -** The process has completed.
- The load average reported by the "w" command indicate the average number of processes in the "Ready" state over the last 1, 5, and 15 minutes, i.e. processes who have everything they need to run but cannot because the CPU is busy doing something else.
- Some systems may have other states besides the ones listed here.
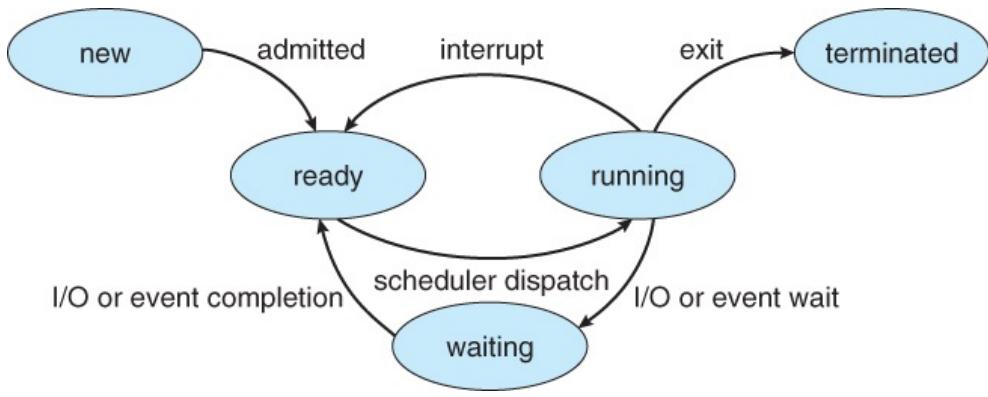
Figure 3.2 - Diagram of process state

3.1.3 Process Control Block

For each process there is a Process Control Block, PCB, which stores the following (types of) process-specific information, as illustrated in Figure 3.1. (Specific details may vary from system to system.)

- **Process State** - Running, waiting, etc., as discussed above.
- **Process ID**, and parent process ID.
- **CPU registers and Program Counter** - These need to be saved and restored when swapping processes in and out of the CPU.
- **CPU-Scheduling information** - Such as priority information and pointers to scheduling queues.
- **Memory-Management information** - E.g. page tables or segment tables.
- **Accounting information** - user and kernel CPU time consumed, account numbers, limits, etc.
- **I/O Status information** - Devices allocated, open file tables, etc.
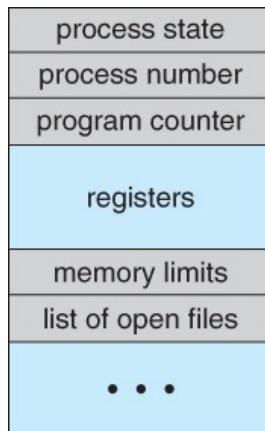

Figure 3.3 - Process control block (PCB)
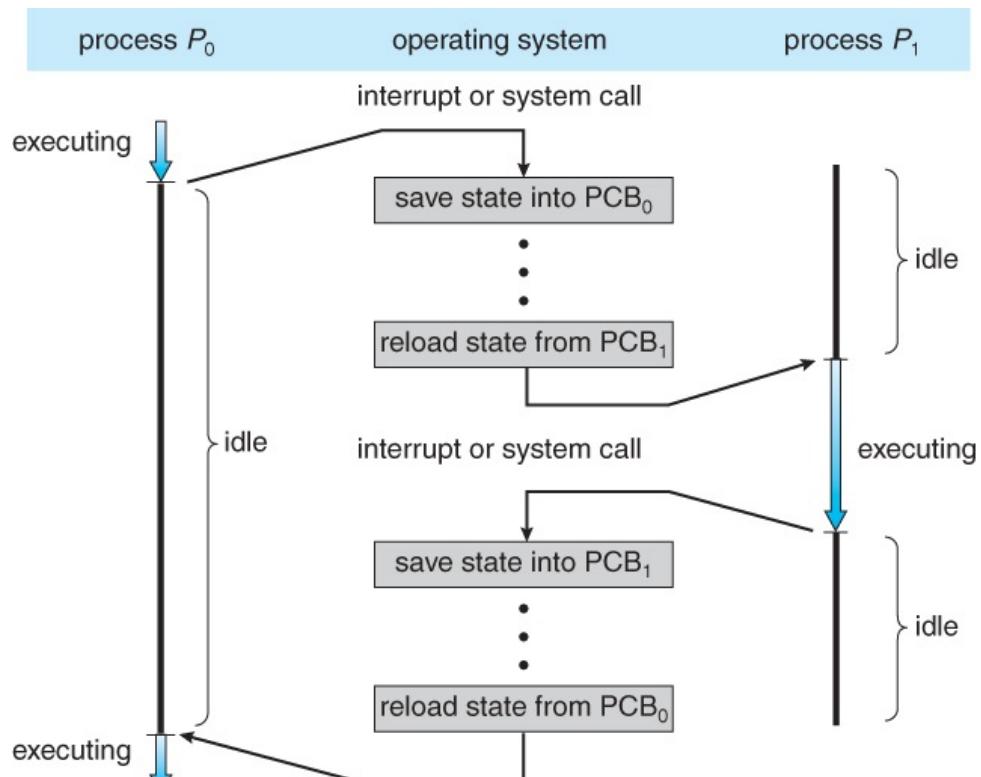
Figure 3.4 - Diagram showing CPU switch from process to process

PROCESS REPRESENTATION IN LINUX

The process control block in the Linux operating system is represented by the C structure `task_struct`. This structure contains all the necessary information for representing a process, including the state of the process, scheduling and memory-management information, list of open files, and pointers to the process's parent and any of its children. (A process's *parent* is the process that created it; its *children* are any processes that it creates.) Some of these fields include:

```
pid_t pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

For example, the state of a process is represented by the field `long state` in this structure. Within the Linux kernel, all active processes are represented using a doubly linked list of `task_struct`, and the kernel maintains a pointer — current — to the process currently executing on the system. This is shown in Figure 3.5.
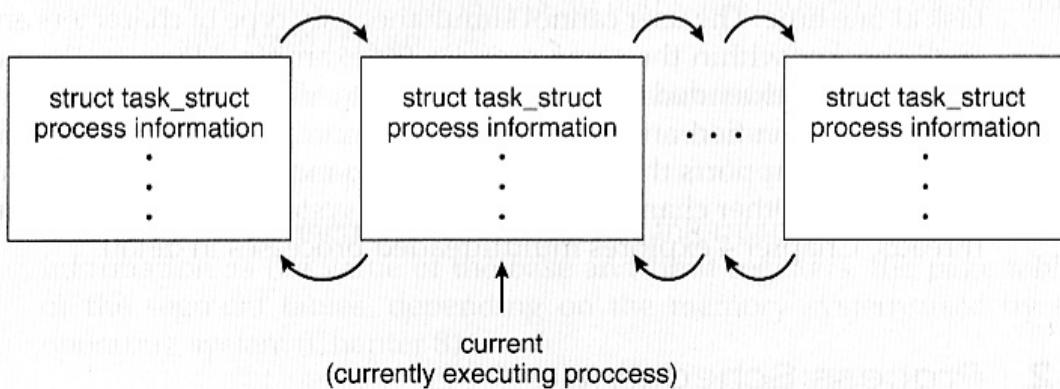

Figure 3.5 Active processes in Linux.

As an illustration of how the kernel might manipulate one of the fields in the `task_struct` for a specified process, let's assume the system would like to change the state of the process currently running to the value `new_state`. If `current` is a pointer to the process currently executing, its state is changed with the following:

```
current->state = new_state;
```

Unnumbered side bar

Digging Deeper: The Linux task_struct definition in sched.h (See also the top of that file.)

3.1.4 Threads

- Modern systems allow a single process to have multiple threads of execution, which execute concurrently. Threads are covered extensively in the next chapter.

3.2 Process Scheduling

- The two main objectives of the process scheduling system are to keep the CPU busy at all times and to deliver "acceptable" response times for all programs, particularly for interactive ones.
- The process scheduler must meet these objectives by implementing suitable policies for swapping processes in and out of the CPU.
- (Note that these objectives can be conflicting. In particular, every time the system steps in to swap processes it takes up time on the CPU to do so, which is thereby "lost" from doing any useful productive work.)

3.2.1 Scheduling Queues

- All processes are stored in the **job queue.**
- Processes in the Ready state are placed in the **ready queue.**
- Processes waiting for a device to become available or to deliver data are placed in **device queues**. There is generally a separate device queue for each device.
- Other queues may also be created and used as needed.
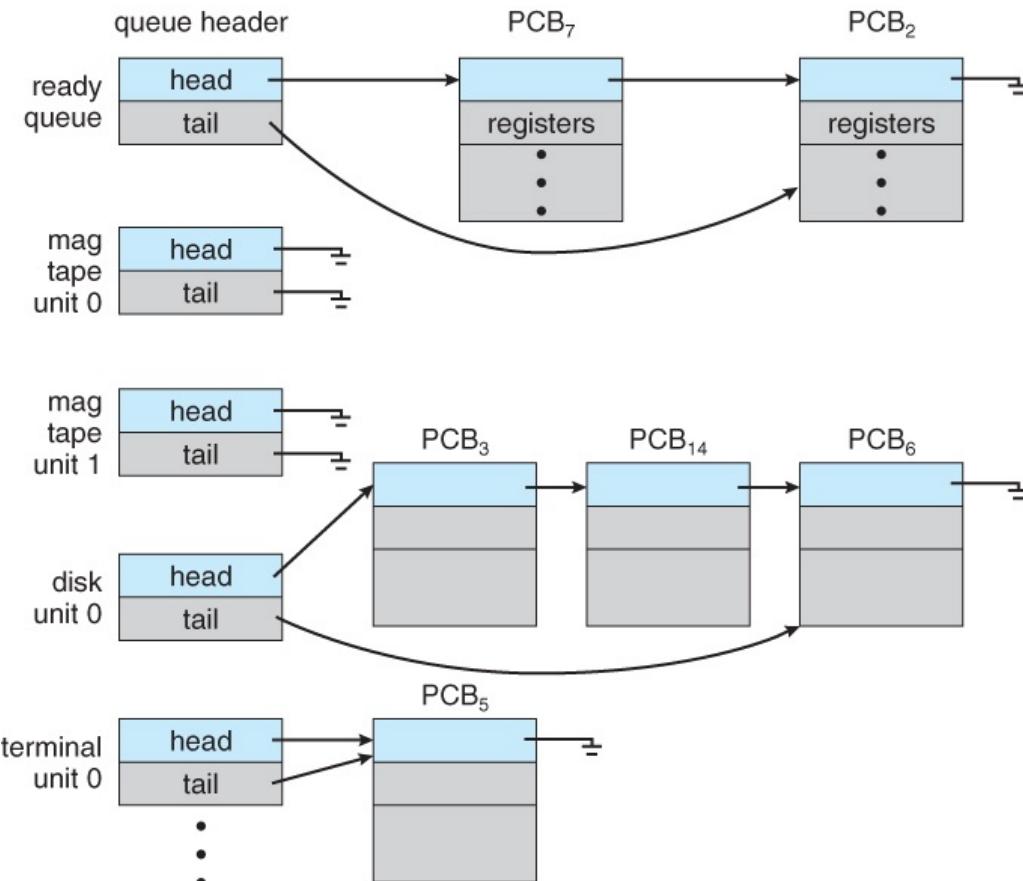
Figure 3.5 - The ready queue and various I/O device queues

3.2.2 Schedulers

- A **long-term scheduler** is typical of a batch system or a very heavily loaded system. It runs infrequently, (such as when one process ends selecting one more to be loaded in from disk in its place), and can afford to take the time to implement intelligent and advanced scheduling algorithms.
- The **short-term scheduler**, or CPU Scheduler, runs very frequently, on the order of 100 milliseconds, and must very quickly swap one process out of the CPU and swap in another one.
- Some systems also employ a **medium-term scheduler**. When system loads get high, this scheduler will swap one or more processes out of the ready queue system for a few seconds, in order to allow smaller faster jobs to finish up quickly and clear the system. See the differences in Figures 3.7 and 3.8 below.
- An efficient scheduling system will select a good **process mix** of **CPU-bound** processes and **I/O bound** processes.
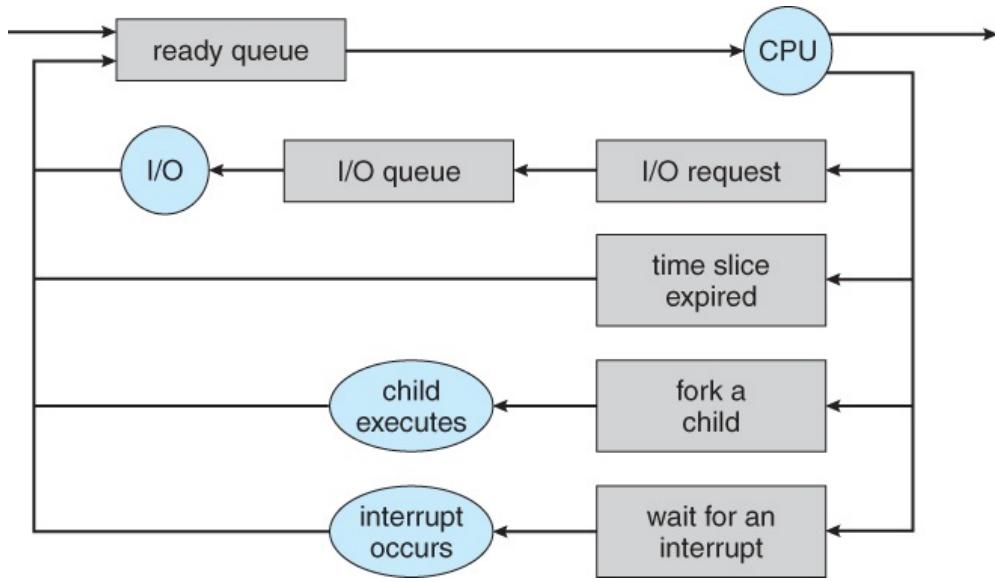
**Figure 3.6 - Queueing-diagram representation of process scheduling**
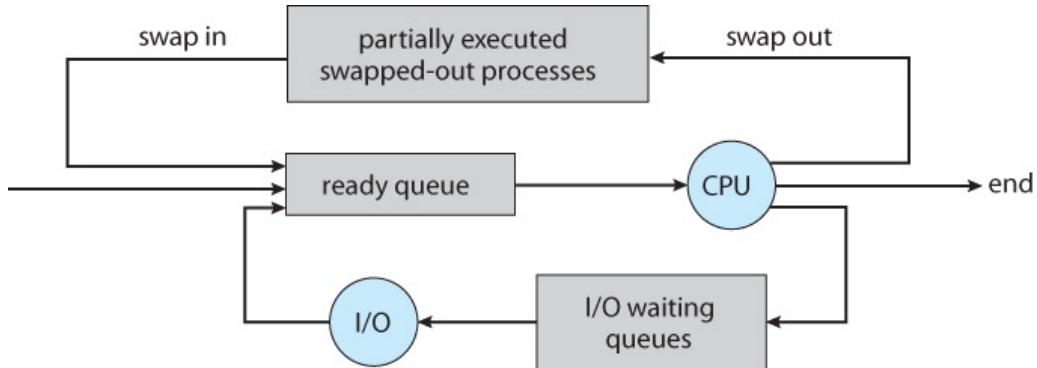
Figure 3.7 - Addition of a medium-term scheduling to the queueing diagram

3.2.3 Context Switch

- Whenever an interrupt arrives, the CPU must do a **state-save** of the currently running process, then switch into kernel mode to handle the interrupt, and then do a **state-restore** of the interrupted process.
- Similarly, a **context switch** occurs when the time slice for one process has expired and a new process is to be loaded from the ready queue. This will be instigated by a timer interrupt, which will then cause the current process's state to be saved and the new process's state to be restored.
- Saving and restoring states involves saving and restoring all of the registers and program counter(s), as well as the process control blocks described above.
- Context switching happens VERY VERY frequently, and the overhead of doing the switching is just lost CPU time, so context switches (state saves & restores) need to be as fast as possible. Some hardware has special provisions for speeding this up, such as a single machine instruction for saving or restoring all registers at once.
- Some Sun hardware actually has multiple sets of registers, so the context switching can be speeded up by merely switching which set of registers are currently in use. Obviously there is a limit as to how many processes can be switched between in this manner, making it attractive to implement the medium-term scheduler to swap some processes out as shown in Figure 3.8 above.

Because of the constraints imposed on mobile devices, early versions of iOS did not provide user-application multitasking; only one application runs in the foreground and all other user applications are suspended. Operating-system tasks were multitasked because they were written by Apple and well behaved. However, beginning with iOS 4, Apple now provides a limited form of multitasking for user applications, thus allowing a single foreground application to run concurrently with multiple background applications. (On a mobile device, the foreground application is the application currently open and appearing on the display. The background application remains in memory, but does not occupy the display screen.) The iOS 4 programming API provides support for multitasking, thus allowing a process to run in the background without being suspended. However, it is limited and only available for a limited number of application types, including applications

- running a single, finite-length task (such as completing a download of content from a network);

- receiving notifications of an event occurring (such as a new email message);

- with long-running background tasks (such as an audio player.)

Apple probably limits multitasking due to battery life and memory use concerns. The CPU certainly has the features to support multitasking, but Apple chooses to not take advantage of some of them in order to better manage resource use.

Android does not place such constraints on the types of applications that can run in the background. If an application requires processing while in the background, the application must use a service, a separate application component that runs on behalf of the background process. Consider a streaming audio application: if the application moves to the background, the service continues to send audio files to the audio device driver on behalf of the background application. In fact, the service will continue to run even if the background application is suspended. Services do not have a user interface and have a small memory footprint, thus providing an efficient technique for multitasking in a mobile environment.

3.3 Operations on Processes

3.3.1 Process Creation

- Processes may create other processes through appropriate system calls, such as **fork** or **spawn**. The process which does the creating is termed the **parent** of the other process, which is termed its **child**.
- Each process is given an integer identifier, termed its **process identifier**, or PID. The parent PID (PPID) is also stored for each process.
- On typical UNIX systems the process scheduler is termed **sched**, and is given PID 0. The first thing it does at system startup time is to launch **init**, which gives that process PID 1. Init then launches all system daemons and user logins, and becomes the ultimate parent of all other processes. Figure 3.9 shows a typical process tree for a Linux system, and other systems will have similar though not identical trees:
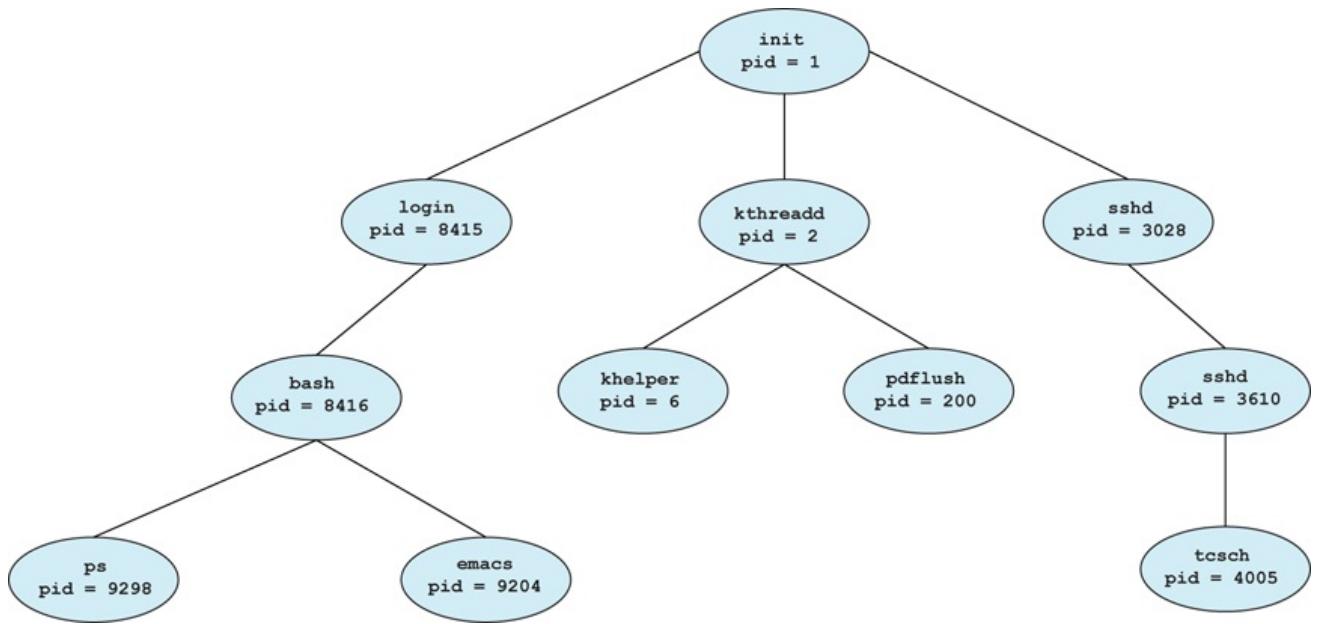
Figure 3.8 - A tree of processes on a typical Linux system

- Depending on system implementation, a child process may receive some amount of shared resources with its parent. Child processes may or may not be limited to a subset of the resources originally allocated to the parent, preventing runaway children from consuming all of a certain system resource.
- There are two options for the parent process after creating the child:
    1. Wait for the child process to terminate before proceeding. The parent makes a wait( ) system call, for either a specific child or for any child, which causes the parent process to block until the wait( ) returns. UNIX shells normally wait for their children to complete before issuing a new prompt.
    2. Run concurrently with the child, continuing to process without waiting. This is the operation seen when a UNIX shell runs a process as a background task. It is also possible for the parent to run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation. (E.g. the parent may fork off a number of children without waiting for any of them, then do a little work of its own, and then wait for the children.)
- Two possibilities for the address space of the child relative to the parent:
    1. The child may be an exact duplicate of the parent, sharing the same program and data segments in memory. Each will have their own PCB, including program counter, registers, and PID. This is the behavior of the **fork** system call in UNIX.
    2. The child process may have a new program loaded into its address space, with all new code and data segments. This is the behavior of the **spawn** system calls in Windows. UNIX systems implement this as a second step, using the **exec** system call.
- Figures 3.10 and 3.11 below shows the fork and exec process on a UNIX system. Note that the **fork** system call returns the PID of the processes child to each process - It returns a zero to the child process and a non-zero child PID to the parent, so the return value indicates which process is which. Process IDs can be looked up any time for the current process or its direct parent using the getpid( ) and getppid( ) system calls respectively.

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) {/* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) {/* child process */
        execlp("/bin/ls","ls",NULL);
    }
    else {/* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}
```

Figure 3.10 C program forking a separate process.

Figure 3.9 Creating a separate process using the UNIX fork( ) system call.
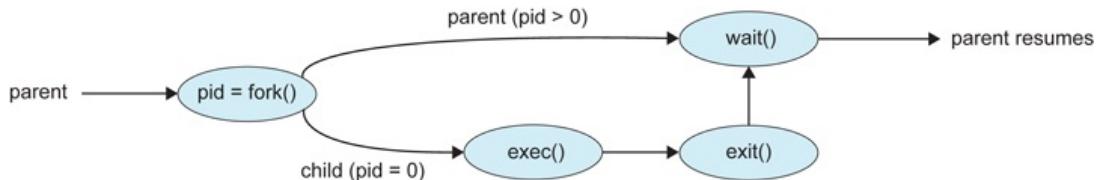
**Figure 3.10 - Process creation using the fork( ) system call**

- Related man pages:
    - fork(2)
    - exec(3)
    - wait(2)

- Figure 3.12 shows the more complicated process for Windows, which must provide all of the parameter information for the new process as part of the forking process.

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    // allocate memory
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // create child process
    if (!CreateProcess(NULL, // use command line
    "C:\\WINDOWS\\system32\\mspaint.exe", // command line
    NULL, // don't inherit process handle
    NULL, // don't inherit thread handle
    FALSE, // disable handle inheritance
    0, // no creation flags
    NULL, // use parent's environment block
    NULL, // use parent's existing directory
    &si,
    &pi))
    {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }
    // parent will wait for the child to complete
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    // close handles
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Figure 3.12 Creating a separate process using the Win32 API.

Figure 3.11

3.3.2 Process Termination

- Processes may request their own termination by making the **exit( )** system call, typically returning an int. This int is passed along to the parent if it is doing a **wait( )**, and is typically zero on successful completion and some non-zero code in the event of problems.
    - child code:

        ```
        int exitCode;
        exit( exitCode );  // return exitCode; has the same effect when executed from main( )
        ```

    - parent code:

        ```
        pid_t pid;
        int status
        pid = wait( &status );
        // pid indicates which child exited. exitCode in low-order bits of status
        // macros can test the high-order bits of status for why it stopped
        ```

- Processes may also be terminated by the system for a variety of reasons, including:
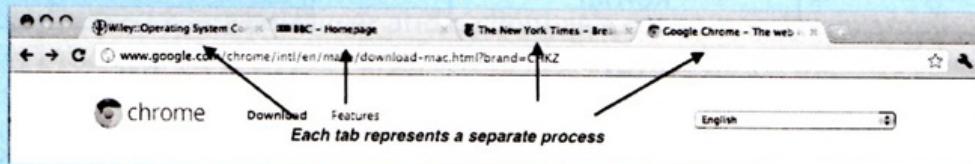
- The inability of the system to deliver necessary system resources.
- In response to a KILL command, or other un handled process interrupt.
- A parent may kill its children if the task assigned to them is no longer needed.
- If the parent exits, the system may or may not allow the child to continue without a parent. (On UNIX systems, orphaned processes are generally inherited by init, which then proceeds to kill them. The UNIX *nohup* command allows a child to continue executing after its parent has exited.)
- When a process terminates, all of its system resources are freed up, open files flushed and closed, etc. The process termination status and execution times are returned to the parent if the parent is waiting for the child to terminate, or eventually returned to init if the process becomes an orphan. (Processes which are trying to terminate but which cannot because their parent is not waiting for them are termed **zombies**. These are eventually inherited by init as orphans and killed off. Note that modern UNIX shells do not produce as many orphans and zombies as older systems used to.)

3.4 Interprocess Communication

- **Independent Processes** operating concurrently on a systems are those that can neither affect other processes or be affected by other processes.
- **Cooperating Processes** are those that can affect or be affected by other processes. There are several reasons why cooperating processes are allowed:
    - Information Sharing - There may be several processes which need access to the same file for example. (e.g. pipelines.)
    - Computation speedup - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks to be solved simultaneously (particularly when multiple processors are involved.)
    - Modularity - The most efficient architecture may be to break a system down into cooperating modules. (E.g. databases with a client-server architecture.)
    - Convenience - Even a single user may be multi-tasking, such as editing, compiling, printing, and running the same code in different windows.

MULTIPROCESS ARCHITECTURE—CHROME BROWSER

Many websites contain active content such as JavaScript, Flash, and HTML5 to provide a rich and dynamic web-browsing experience. Unfortunately, these web applications may also contain software bugs, which can result in sluggish response times and can even cause the web browser to crash. This isn't a big problem in a web browser that displays content from only one website. But most contemporary web browsers provide tabbed browsing, which allows a single instance of a web browser application to open several websites at the same time, with each site in a separate tab. To switch between the different sites , a user need only click on the appropriate tab. This arrangement is illustrated below:

Each tab represents a separate process

A problem with this approach is that if a web application in any tab crashes, the entire process—including all other tabs displaying additional websites —crashes as well.

Google's Chrome web browser was designed to address this issue by using a multiprocess architecture. Chrome identifies three different types of processes: browser, renderers, and plug-ins.

- The browser process is responsible for managing the user interface as well as disk and network I/O. A new browser process is created when Chrome is started. Only one browser process is created.

- Renderer processes contain logic for rendering web pages. Thus, they contain the logic for handling HTML, Javascript, images, and so forth. As a general rule, a new renderer process is created for each website opened in a new tab, and so several renderer processes may be active at the same time.

- A plug-in process is created for each type of plug-in (such as Flash or QuickTime) in use. Plug-in processes contain the code for the plug-in as well as additional code that enables the plug-in to communicate with associated renderer processes and the browser process.

The advantage of the multiprocess approach is that websites run in isolation from one another. If one website crashes, only its renderer process is affected; all other processes remain unharmed. Furthermore, renderer processes run in a sandbox, which means that access to disk and network I/O is restricted, minimizing the effects of any security exploits.

- Cooperating processes require some type of inter-process communication, which is most commonly one of two types: Shared Memory systems or Message Passing systems. Figure 3.13 illustrates the difference between the two systems:
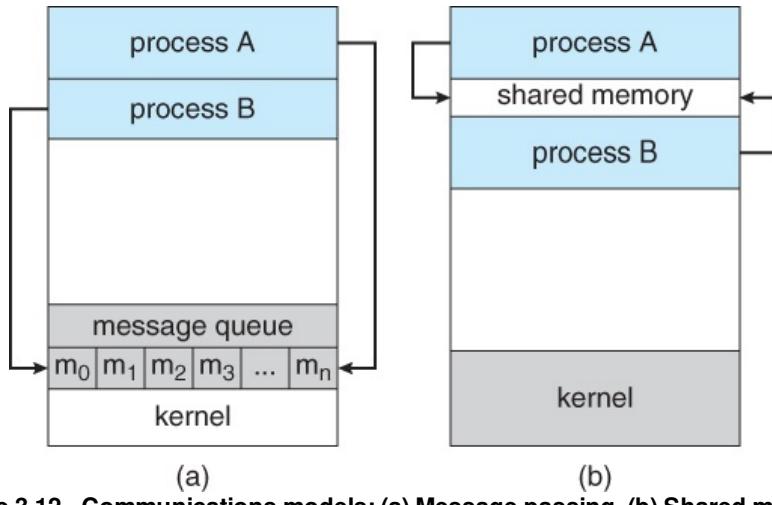
Figure 3.12 - Communications models: (a) Message passing. (b) Shared memory.

- Shared Memory is faster once it is set up, because no system calls are required and access occurs at normal memory speeds. However it is more complicated to set up, and doesn't work as well across multiple computers. Shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.
- Message Passing requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers. Message passing is generally preferable when the amount and/or frequency of data transfers is small, or when multiple computers are involved.

3.4.1 Shared-Memory Systems

- In general the memory to be shared in a shared-memory system is initially within the address space of a particular process, which needs to make system calls in order to make that memory publicly available to one or more other processes.
- Other processes which wish to use the shared memory must then make their own system calls to attach the shared memory area onto their address space.
- Generally a few messages must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.

Producer-Consumer Example Using Shared Memory

- This is a classic example, in which one process is producing data and another process is consuming the data. (In this example in the order in which it is produced, although that could vary.)
- The data is passed via an intermediary buffer, which may be either unbounded or bounded. With a bounded buffer the producer may have to wait until there is space available in the buffer, but with an unbounded buffer the producer will never need to wait. The consumer may need to wait in either case until there is data available.
- This example uses shared memory and a circular queue. Note in the code below that only the producer changes "in", and only the consumer changes "out", and that they can never be accessing the same array location at the same time.
- First the following data is set up in the shared memory area:

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} item;

item buffer[ BUFFER_SIZE ];
int in = 0;
int out = 0;
```

- Then the producer process. Note that the buffer is full when "in" is one less than "out" in a circular sense:

```
// Code from Figure 3.13

item nextProduced;

while( true ) {

    /* Produce an item and store it in nextProduced */
    nextProduced = makeNewItem( . . . );

    /* Wait for space to become available */
    while( ( ( in + 1 ) % BUFFER_SIZE ) == out )
```

```
        ; /* Do nothing */

    /* And then store the item and repeat the loop. */
    buffer[ in ] = nextProduced;
    in = ( in + 1 ) % BUFFER_SIZE;

}
```

- Then the consumer process. Note that the buffer is empty when "in" is equal to "out":

```
// Code from Figure 3.14

item nextConsumed;

while( true ) {

    /* Wait for an item to become available */
    while( in == out )
        ; /* Do nothing */

    /* Get the next available item */
    nextConsumed = buffer[ out ];
    out = ( out + 1 ) % BUFFER_SIZE;

    /* Consume the item in nextConsumed
        ( Do something with it ) */

}
```

3.4.2 Message-Passing Systems

- Message passing systems must support at a minimum system calls for "send message" and "receive message".
- A communication link must be established between the cooperating processes before messages can be sent.
- There are three key issues to be resolved in message passing systems as further explored in the next three subsections:
  - Direct or indirect communication (naming)
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering.

3.4.2.1 Naming

- With **direct communication** the sender must know the name of the receiver to which it wishes to send a message.
  - There is a one-to-one link between every sender-receiver pair.
  - For **symmetric** communication, the receiver must also know the specific name of the sender from which it wishes to receive messages.
    For **asymmetric** communications, this is not necessary.
- **Indirect communication** uses shared mailboxes, or ports.
  - Multiple processes can share the same mailbox or boxes.
  - Only one process can read any given message in a mailbox. Initially the process that creates the mailbox is the owner, and is the only one allowed to read mail in the mailbox, although this privilege may be transferred.
    - (Of course the process that reads the message can immediately turn around and place an identical message back in the box for someone else to read, but that may put it at the back end of a queue of messages.)
  - The OS must provide system calls to create and delete mailboxes, and to send and receive messages to/from mailboxes.

3.4.2.2 Synchronization

- Either the sending or receiving of messages (or neither or both) may be either **blocking** or **non-blocking**.

3.4.2.3 Buffering

- Messages are passed via queues, which may have one of three capacity configurations:
  1. **Zero capacity** - Messages cannot be stored in the queue, so senders must block until receivers accept the messages.
  2. **Bounded capacity**- There is a certain pre-determined finite capacity in the queue. Senders must block if the queue is full, until space becomes available in the queue, but may be either blocking or

non-blocking otherwise.

3. **Unbounded capacity** - The queue has a theoretical infinite capacity, so senders are never forced to block.

```
message next_produced;

while (true) {
      /* produce an item in next_produced */

      send(next_produced);
}
```

Figure 3.15 The producer process using message passing.

```
message next_consumed;

while (true) {
      receive(next_consumed);

      /* consume the item in next_consumed */
}
```

Figure 3.16 The consumer process using message passing.

3.5 Examples of IPC Systems

3.5.1 An Example: POSIX Shared Memory (Eighth Edition Version)

1. The first step in using shared memory is for one of the processes involved to allocate some shared memory, using shmget:

```
int segment_id = shmget( IPC_PRIVATE, size, S_IRUSR | S_IWUSR );
```

   - The first parameter specifies the key (identifier) of the segment. IPC_PRIVATE creates a new shared memory segment.
   - The second parameter indicates how big the shared memory segment is to be, in bytes.
   - The third parameter is a set of bitwise ORed flags. In this case the segment is being created for reading and writing.
   - The return value of shmget is an integer identifier

2. Any process which wishes to use the shared memory must *attach* the shared memory to their address space, using shmat:

```
char * shared_memory = ( char * ) shmat( segment_id, NULL, 0 );
```

   - The first parameter specifies the key (identifier) of the segment that the process wishes to attach to its address space
   - The second parameter indicates where the process wishes to have the segment attached. NULL indicates that the system should decide.
   - The third parameter is a flag for read-only operation. Zero indicates read-write; One indicates readonly.
   - The return value of shmat is a void *, which the process can use (type cast) as appropriate. In this example it is being used as a character pointer.

3. Then processes may access the memory using the pointer returned by shmat, for example using sprintf:

```
sprintf( shared_memory, "Writing to shared memory\n" );
```

4. When a process no longer needs a piece of shared memory, it can be detached using shmdt:

```
shmdt( shared_memory );
```

5. And finally the process that originally allocated the shared memory can remove it from the system suing shmctl.

```
shmctl( segment_id, IPC_RMID );
```

6. Figure 3.16 from the eighth edition illustrates a complete program implementing shared memory on a POSIX system:

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the identifier for the shared memory segment */
int segment_id;=
/* a pointer to the shared memory segment */
char* shared_memory;
/* the size (in bytes) of the shared memory segment */
const int size = 4096;

    /* allocate a shared memory segment */
    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

    /* attach the shared memory segment */
    shared_memory = (char *) shmat(segment_id, NULL, 0);

    /* write a message to the shared memory segment */
    sprintf(shared_memory, "Hi there!");

    /* now print out the string from shared memory */
    printf("*%s\n", shared_memory);

    /* now detach the shared memory segment */
    shmdt(shared_memory);

    /* now remove the shared memory segment */
    shmctl(segment_id, IPC_RMID, NULL);

    return 0;
}
```

Figure 3.16   C program illustrating POSIX shared-memory API.

3.5.1 An Example: POSIX Shared Memory (Ninth Edition Version)

1. The ninth edition shows an alternate approach to shared memory in POSIX systems. Under this approach, the first step in using shared memory is to create a shared-memory object using shm_open( ),in a fashion similar to other file opening commands. The name provided will be the name of the memory-mapped file.

```
shm_fd = shm_open( name,O_CREAT | O_RDRW,0666 );
```

2. The next step is to set the size of the file using ftruncate:

```
ftruncate( shm_fd, 4096 );
```

3. Finally the mmap system call maps the file to a memory address in the user program space.and makes it shared. In this example the process that created the shared memory will be writing to it:

```
ptr = mmap( 0, SIZE,PROT_WRITE, MAP_SHARED, shm_fd, 0 );
```

4. The "borrower" of the shared memory, (not the one who created it), calls `shm_open( )` and `mmap( )` with different arguments, skips the `ftruncate( )` step and unlinks (removes) the file name when it is

done with it. Note that the "borrower" must use the same file name as the "lender" who created it. (This information could have been passed using messages.)

```
shm_unlink( name );
```

5. Note that writing to and reading from the shared memory is done with pointers and memory addresses (sprintf) in both the 9th and 8th edition versions, even though the 9th edition is illustrating memory mapping of a file.
6. Figures 3.17 and 3.18 from the ninth edition illustrate a complete program implementing shared memory on a POSIX system:

```
#include <stdio.h>
#include <stlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

Figure 3.17 Producer process illustrating POSIX shared-memory API.

```c
#include <stdio.h>
#include <stlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Figure 3.18 Consumer process illustrating POSIX shared-memory API.

3.5.2 An Example: Mach

- Recall that the Mach kernel is a micro kernel, which performs few services besides delivering messages between other tasks (both system tasks and user tasks.)
- Most communication in Mach, including all system calls and inter-process communication is done via messages sent to mailboxes, also known as ports.
- Whenever a task (process) is created, it automatically gets two special mailboxes: a Kernel mailbox, and a Notify mailbox.
    - The kernel communicates with the task using the Kernel mailbox.
    - The kernel sends notification of events to the Notify mailbox.
- Three system calls are used for message transfer:
    - msg_send( ) sends a message to a mailbox
    - msg_receive( ) receives a message.
    - msg_rpc( ) sends a message and waits for exactly one message in response from the sender.
- Port_allocate( ) creates a new mailbox and the associated queue for holding messages (size 8 by default.)
- Only one task at a time can own or receive messages from any given mailbox, but these are transferable.
- Messages from the same sender to the same receiver are guaranteed to arrive in FIFO order, but no guarantees are made regarding messages from multiple senders.
- Messages consist of a fixed-length header followed by variable length data.
    - The header contains the mailbox number (address) of the receiver and the sender.
    - The data section consists of a list of typed data items, each containing a type, size, and value.
- If the receiver's mailbox is full, the sender has four choices:
    1. Wait indefinitely until there is room in the mailbox.
    2. Wait at most N milliseconds.
    3. Do not wait at all.

4. Temporarily cache the message with the kernel, for delivery when the mailbox becomes available.
   - Only one such message can be pending at any given time from any given sender to any given receiver.
   - Normally only used by certain system tasks, such as the print spooler, which must notify the "client" of the completion of their job, but cannot wait around for the mailbox to become available.

- Receive calls must specify the mailbox or mailbox set from which they wish to receive messages.
- Port_status( ) reports the number of messages waiting in any given mailbox.
- If there are no messages available in a mailbox (set), the receiver can either block for N milliseconds, or not block at all.
- In order to avoid delays caused by copying messages (multiple times), Mach re-maps the memory space for the message from the sender's address space to the receiver's address space (using virtual memory techniques to be covered later), and does not actually move the message anywhere at all. (When the sending and receiving task are both on the same computer.)

3.5.3 An Example: Windows XP
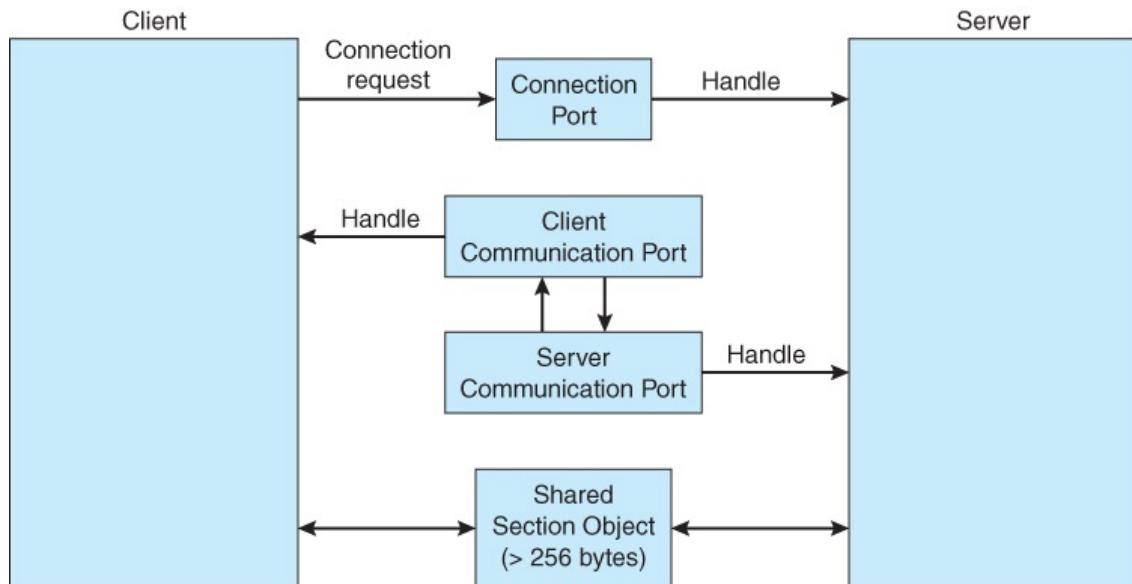
Figure 3.19 - Advanced local procedure calls in Windows

3.6 Communication in Client-Server Systems

3.6.1 Sockets

- A **socket** is an endpoint for communication.
- Two processes communicating over a network often use a pair of connected sockets as a communication channel. Software that is designed for client-server operation may also use sockets for communication between two processes running on the same computer - For example the UI for a database program may communicate with the back-end database manager using sockets. (If the program were developed this way from the beginning, it makes it very easy to port it from a single-computer system to a networked application.)
- A socket is identified by an IP address concatenated with a port number, e.g. 200.100.50.5:80.
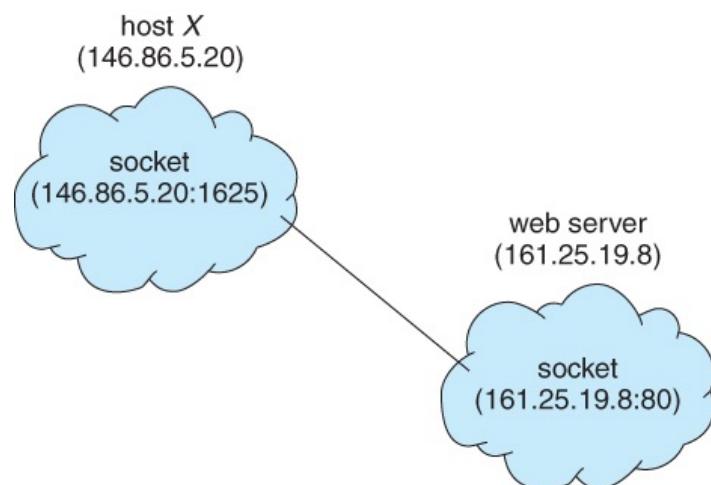
Figure 3.20 - Communication using sockets

- Port numbers below 1024 are considered to be *well-known*, and are generally reserved for common Internet services. For example, telnet servers listen to port 23, ftp servers to port 21, and web servers to port 80.

- General purpose user sockets are assigned unused ports over 1024 by the operating system in response to system calls such as socket( ) or soctkepair( ).
- Communication channels via sockets may be of one of two major forms:
  - **Connection-oriented (TCP, Transmission Control Protocol)** connections emulate a telephone connection. All packets sent down the connection are guaranteed to arrive in good condition at the other end, and to be delivered to the receiving process in the order in which they were sent. The TCP layer of the network protocol takes steps to verify all packets sent, re-send packets if necessary, and arrange the received packets in the proper order before delivering them to the receiving process. There is a certain amount of overhead involved in this procedure, and if one packet is missing or delayed, then any packets which follow will have to wait until the errant packet is delivered before they can continue their journey.
  - **Connectionless (UDP, User Datagram Protocol)** emulate individual telegrams. There is no guarantee that any particular packet will get through undamaged (or at all), and no guarantee that the packets will get delivered in any particular order. There may even be duplicate packets delivered, depending on how the intermediary connections are configured. UDP transmissions are much faster than TCP, but applications must implement their own error checking and recovery procedures.
- Sockets are considered a low-level communications channel, and processes may often choose to use something at a higher level, such as those covered in the next two sections.
- Figure 3.19 and 3.20 illustrate a client-server system for determining the current date using sockets in Java.

```java
import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);

            // now listen for connections
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                  PrintWriter(client.getOutputStream(), true);

                // write the Date to the socket
                pout.println(new java.util.Date().toString());

                // close the socket and resume
                // listening for connections
                client.close();
            }
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

Figure 3.19 Date server.

```java
import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) {
        try {
```

```java
try {
    //make connection to server socket
    Socket sock = new Socket("127.0.0.1",6013);

    InputStream in = sock.getInputStream();
    BufferedReader bin = new
        BufferedReader(new InputStreamReader(in));

    // read the date from the socket
    String line;
    while ( (line = bin.readLine()) != null)
        System.out.println(line);

    // close the socket connection
    sock.close();
}
catch (IOException ioe) {
    System.err.println(ioe);
}
}
}
```

Figure 3.20 Date client.

Figure 3.21 and Figure 3.22

3.6.2 Remote Procedure Calls, RPC

- The general concept of RPC is to make procedure calls similarly to calling on ordinary local procedures, except the procedure being called lies on a remote machine.
- Implementation involves **stubs** on either end of the connection.
  - The local process calls on the stub, much as it would call upon a local procedure.
  - The RPC system packages up (marshals) the parameters to the procedure call, and transmits them to the remote system.
  - On the remote side, the RPC daemon accepts the parameters and calls upon the appropriate remote procedure to perform the requested work.
  - Any results to be returned are then packaged up and sent back by the RPC system to the local system, which then unpackages them and returns the results to the local calling procedure.
- One potential difficulty is the formatting of data on local versus remote systems. (e.g. big-endian versus little-endian.) The resolution of this problem generally involves an agreed-upon intermediary format, such as XDR (external data representation.)
- Another issue is identifying which procedure on the remote system a particular RPC is destined for.
  - Remote procedures are identified by *ports*, though not the same ports as the socket ports described earlier.
  - One solution is for the calling procedure to know the port number they wish to communicate with on the remote system. This is problematic, as the port number would be compiled into the code, and it makes it break down if the remote system changes their port numbers.
  - More commonly a *matchmaker* process is employed, which acts like a telephone directory service. The local process must first contact the matchmaker on the remote system (at a well-known port number), which looks up the desired port number and returns it. The local process can then use that information to contact the desired remote procedure. This operation involves an extra step, but is much more flexible. An example of the matchmaker process is illustrated in Figure 3.21 below.
- One common example of a system based on RPC calls is a networked file system. Messages are passed to read, write, delete, rename, or check status, as might be made for ordinary local disk access requests.
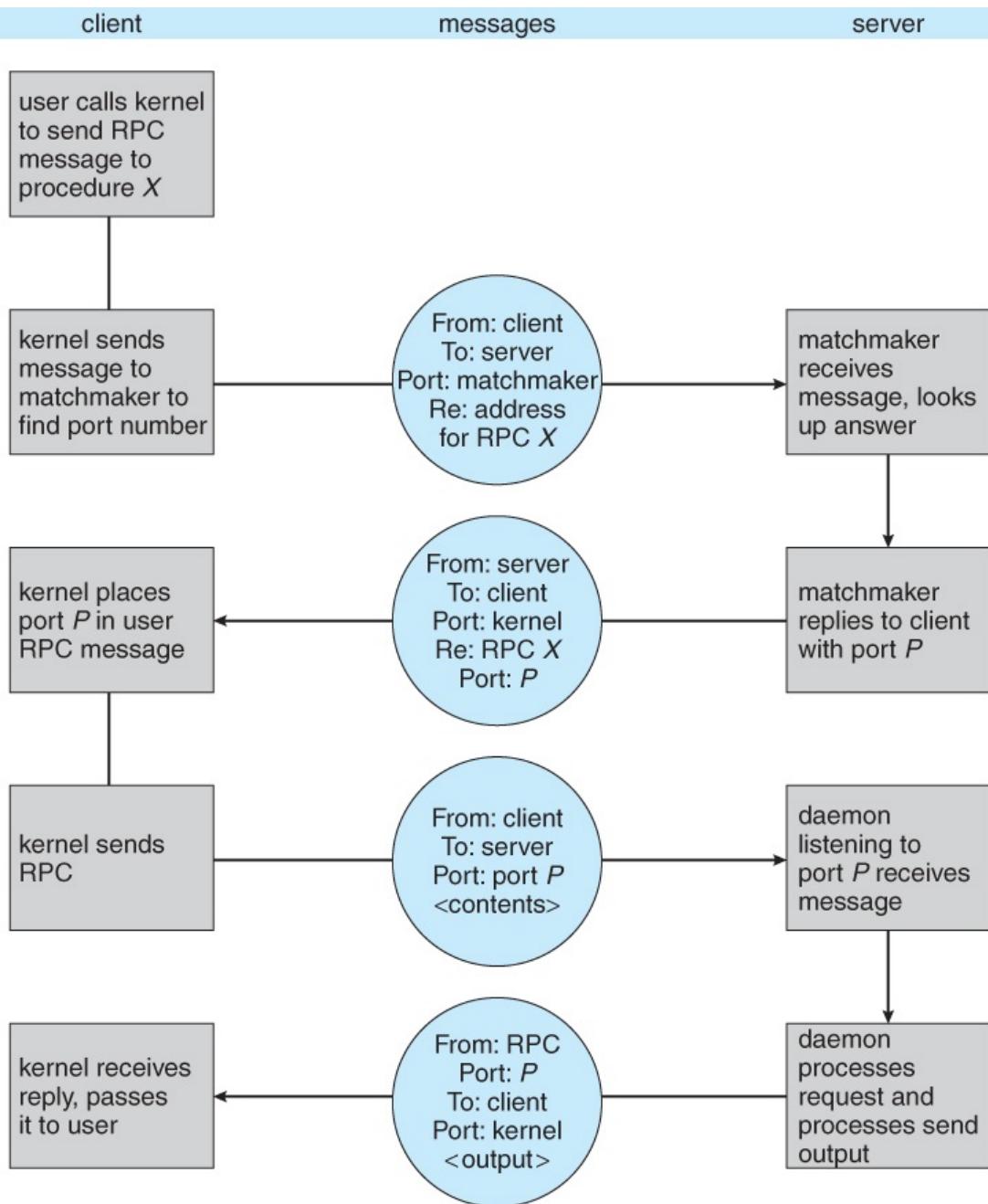
Figure 3.23 - Execution of a remote procedure call (RPC).
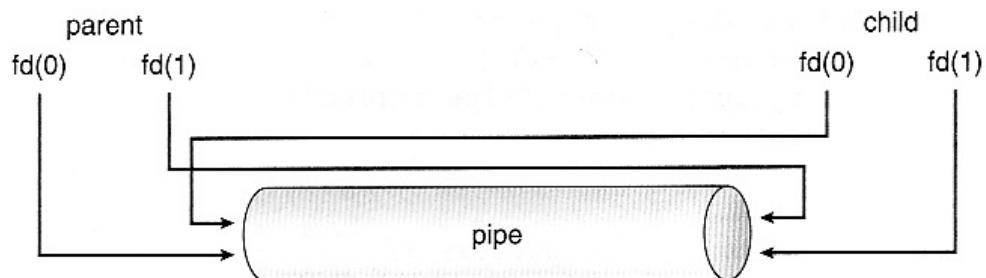
3.6.3 Pipes

- **Pipes** are one of the earliest and simplest channels of communications between (UNIX) processes.
- There are four key considerations in implementing pipes:
  1. Unidirectional or Bidirectional communication?
  2. Is bidirectional communication half-duplex or full-duplex?
  3. Must a relationship such as parent-child exist between the processes?
  4. Can pipes communicate over a network, or only on the same machine?
- The following sections examine these issues on UNIX and Windows

3.6.3.1 Ordinary Pipes

- Ordinary pipes are uni-directional, with a reading end and a writing end. (If bidirectional communications are needed, then a second pipe is required.)
- In UNIX ordinary pipes are created with the system call "`int pipe( int fd [ ] )`".
  - The return value is 0 on success, -1 if an error occurs.
  - The int array must be allocated before the call, and the values are filled in by the pipe system call:
    - fd[0] is filled in with a file descriptor for the reading end of the pipe
    - fd[1] is filled in with a file descriptor for the writing end of the pipe
  - UNIX pipes are accessible as files, using standard read( ) and write( ) system calls.
  - Ordinary pipes are only accessible within the process that created them.
    - Typically a parent creates the pipe before forking off a child.

- When the child inherits open files from its parent, including the pipe file(s), a channel of communication is established.
- Each process (parent and child) should first close the ends of the pipe that they are not using. For example, if the parent is writing to the pipe and the child is reading, then the parent should close the reading end of its pipe after the fork and the child should close the writing end.

• Figure 3.22 shows an ordinary pipe in UNIX, and Figure 3.23 shows code in which it is used.

Figure 3.22 File descriptors for an ordinary pipe.

Figure 3.24

```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;

/* create the pipe */
if (pipe(fd) == -1) {
   fprintf(stderr,"Pipe failed");
   return 1;
}

/* fork a child process */
pid = fork();

if (pid < 0) { /* error occurred */
   fprintf(stderr, "Fork Failed");
   return 1;
}

if (pid > 0) { /* parent process */
    /* close the unused end of the pipe */
    close(fd[READ_END]);

    /* write to the pipe */
    write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
```

```
            /* close the write end of the pipe */
            close(fd[WRITE_END]);
        }
        else { /* child process */
            /* close the unused end of the pipe */
            close(fd[WRITE_END]);

            /* read from the pipe */
            read(fd[READ_END], read_msg, BUFFER_SIZE);
            printf("read %s",read_msg);

            /* close the write end of the pipe */
            close(fd[READ_END]);
        }

        return 0;
    }
```

Figure 3.23 Ordinary pipes in UNIX.

Figure 3.24 Continuation of Figure 3.23 program.

Figure 3.25 and Figure 3.26

- Ordinary pipes in Windows are very similar
  - Windows terms them *anonymous* pipes
  - They are still limited to parent-child relationships.
  - They are read from and written to as files.
  - They are created with `CreatePipe( )` function, which takes additional arguments.
  - In Windows it is necessary to specify what resources a child inherits, such as pipes.

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE ReadHandle, WriteHandle;
STARTUPINFO si;
PROCESS_INFORMATION pi;
char message[BUFFER_SIZE] = "Greetings";
DWORD written;

/* set up security attributes allowing pipes to be inherited */
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};
/* allocate memory */
ZeroMemory(&pi, sizeof(pi));

/* create the pipe */
if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
  fprintf(stderr, "Create Pipe Failed");
  return 1;
}

/* establish the START INFO structure for the child process */
```

```
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* redirect standard input to the read end of the pipe */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;

/* don't allow the child to inherit the write end of pipe */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* create the child process */
CreateProcess(NULL, "child.exe", NULL, NULL,
    TRUE, /* inherit handles */
    0, NULL, NULL, &si, &pi);

/* close the unused end of the pipe */
CloseHandle(ReadHandle);

/* the parent writes to the pipe */
if (!WriteFile(WriteHandle, message, BUFFER_SIZE, &written, NULL))
    fprintf(stderr, "Error writing to pipe.");

/* close the write end of the pipe */
CloseHandle(WriteHandle);

/* wait for the child to exit */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}
```

Figure 3.25 Windows anonymous pipes — parent process.

Figure 3.26 Continuation of Figure 3.25 program.

Figure 3.27 and Figure 3.28

```
#include <stdio.h>
#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE Readhandle;
CHAR buffer[BUFFER_SIZE];
DWORD read;

    /* get the read handle of the pipe */
    ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

    /* the child reads from the pipe */
    if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
       printf("child read %s",buffer);
    else
       fprintf(stderr, "Error reading from pipe");

    return 0;
}
```

Figure 3.27 Windows anonymous pipes — child process.

Figure 3.29

3.6.3.2 Named Pipes

- Named pipes support bidirectional communication, communication between non parent-child related processes, and persistence after the process which created them exits. Multiple processes can also share a named pipe, typically one reader and multiple writers.
- In UNIX, named pipes are termed fifos, and appear as ordinary files in the file system.
    - (Recognizable by a "p" as the first character of a long listing, e.g. /dev/initctl)
    - Created with `mkfifo( )` and manipulated with `read( )`, `write( )`, `open( )`, `close( )`, etc.
    - UNIX named pipes are bidirectional, but half-duplex, so two pipes are still typically used for bidirectional communications.
    - UNIX named pipes still require that all processes be running on the same machine. Otherwise sockets are used.
- Windows named pipes provide richer communications.
- Full-duplex is supported.
- Processes may reside on the same or different machines
- Created and manipulated using `CreateNamedPipe( )`, `ConnectNamedPipe( )`, `ReadFile( )`, and `WriteFile( )`.

Race Conditions (Not from the book)

Any time there are two or more processes or threads operating concurrently, there is potential for a particularly difficult class of problems known as *race conditions.* The identifying characteristic of race conditions is that the performance varies depending on which process or thread executes their instructions before the other one, and this becomes a problem when the program runs correctly in some instances and incorrectly in others. Race conditions are notoriously difficult to debug, because they are unpredictable, unrepeatable, and may not exhibit themselves for years.

Here is an example involving a server and a client communicating via sockets:

1. First the server writes a greeting message to the client via the socket:

```
const int BUFFLENGTH = 100;
char buffer[ BUFFLENGTH ];
sprintf( buffer, "Hello Client %d!", i );
write( clientSockets[ i ], buffer, strlen( buffer ) + 1 );
```

2. The client then reads the greeting into its own buffer. The client does not know for sure how long the message is, so it allocates a buffer bigger than it needs to be. The following will read all available characters in the socket, up to a maximum of BUFFLENGTH characters:

```
const int BUFFLENGTH = 100;
char buffer[ BUFFLENGTH ];
read( mysocket, buffer, BUFFLENGTH );
cout << "Client received: " << buffer << "\n";
```

3. Now the server prepares a packet of work and writes that to the socket:

```
write( clientSockets[ i ], & wPacket, sizeof( wPacket ) );
```

4. And finally the client reads in the work packet and processes it:

```
read( mysocket, & wPacket, sizeof( wPacket ) );
```

The Problem: The problem arises if the server executes step 3 before the client has had a chance to execute step 2, which can easily happen depending on process scheduling. In this case, when the client finally gets around to executing step 2, it will read in not only the original greeting, but also the first part of the work packet. And just to make things harder to figure out, the cout << statement in step 2 will only print out the greeting message, since there is a null byte at the end of the greeting. This actually isn't even a problem at this point, but then later when the client executes step 4, it does not accurately read in the work packet because part of it has already been read into the buffer in step 2.

Solution I: The easiest solution is to have the server write the entire buffer in step 1, rather than just the part filled with the greeting, as:

```
write( clientSockets[ i ], buffer, BUFFLENGTH );
```

Unfortunately this solution has two problems: (1) It wastes bandwidth and time by writing more than is needed, and more importantly, (2) It leaves the code open to future problems if the BUFFLENGTH is not the same in the client and in the server.

Solution II: A better approach for handling variable-length strings is to first write the length of the string, followed by the string itself as a separate write. Under this solution the server code changes to:

```
sprintf( buffer, "Hello Client %d!", i );
int length = strlen( buffer ) + 1;
write( clientSockets[ i ], &length, sizeof( int ) );
write( clientSockets[ i ], buffer, length );
```

and the client code changes to:

```
int length;
if( read( mysocket, &length, sizeof( int ) ) != sizeof( int ) ) {
    perror( "client read error: " );
    exit( -1 );
}

if( length < 1 || length > BUFFLENGTH ) {
    cerr << "Client read invalid length = " << length << endl;
    exit( -1 );
}

if( read( mysocket, buffer, length ) != length ) {
perror( "client read error: " );
exit( -1 );
}

cout << "Client received: " << buffer << "\n";
```
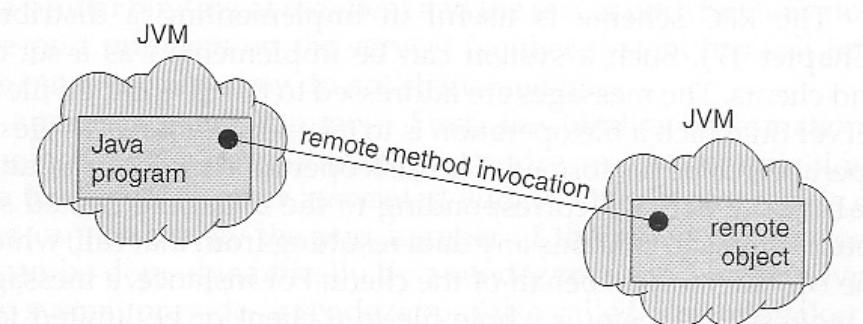
Note that the above solution also checks the return value from the read system call, to verify that the number of characters read is equal to the number expected. (Some of those checks were actually in the original code, but were omitted from the notes for clarity. The real code also uses select( ) before reading, to verify that there are characters present to read and to delay if not.)

Note also that this problem could not be (easily) solved using the synchronization tools covered in chapter 6, because the problem is not really one of two processes accessing the same data at the same time.

3.7 Summary

OLD 3.6.3 Remote Method Invocation, RMI (Optional, Removed from 8th edition)
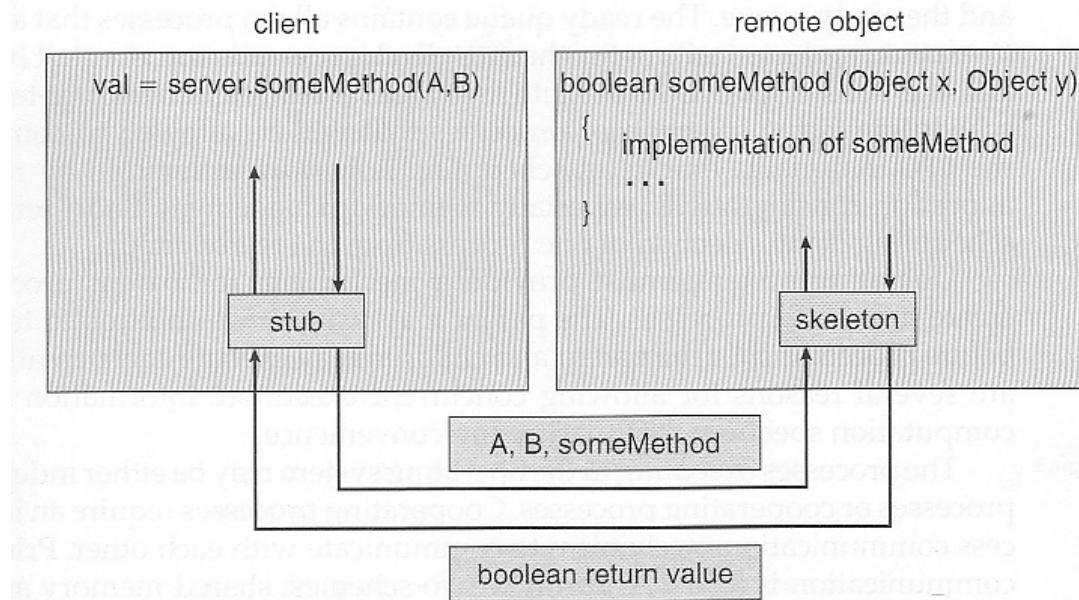
- RMI is the Java implementation of RPC for contacting processes operating on a different Java Virtual Machine, JVM, which may or may not be running on a different physical machine.
- There are two key differences between RPC and RMI, both based on the object-oriented nature of Java:
    - RPC accesses remote procedures or functions, in a procedural-programming paradigm. RMI accesses methods within remote Objects.
    - The data passed by RPC as function parameters are ordinary data only, i.e. ints, floats, doubles, etc. RMI also supports the passing of Objects.
- RMI is implemented using stubs (on the client side) and skeletons (on the servers side), whose responsibility is to package (marshall) and unpack the parameters and return values being passed back and forth, as illustrated in Figures 3.22 and 3.23:
- See the Java online documentation for more details on RMI, including issues with value vs. reference parameter passing, and the requirements that certain passed data items be *Serializable.*

Figure 3.22 Remote method invocation.

Figure omitted in 8th edition

Figure 3.23 Marshalling parameters.

Figure omitted in 8th edition
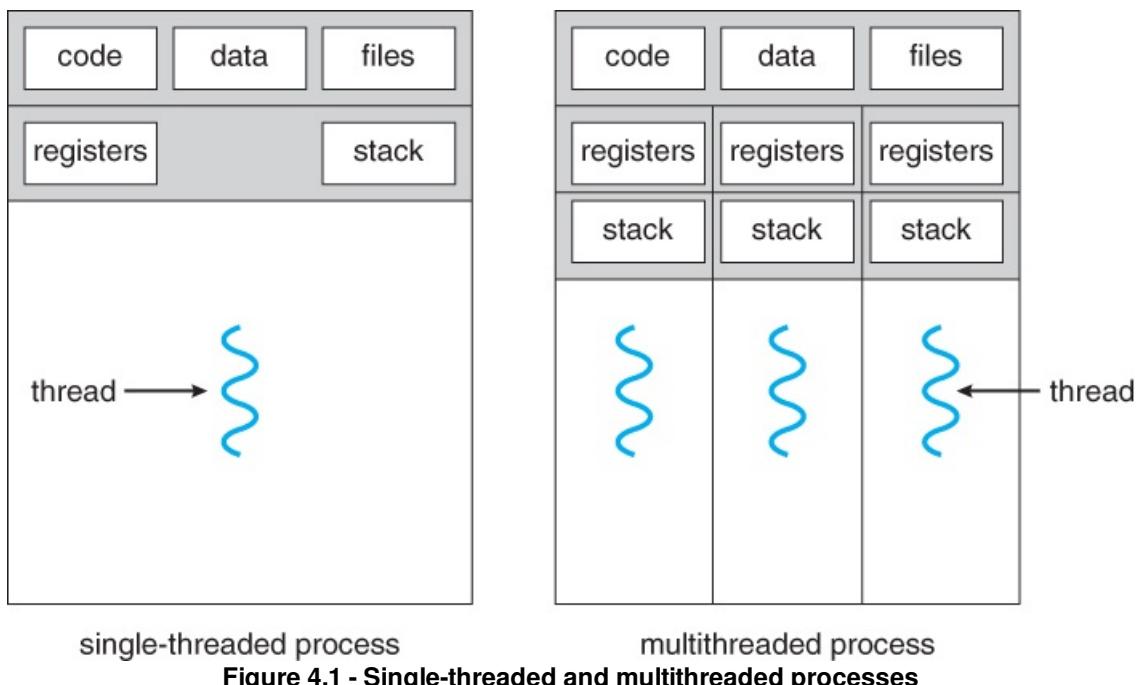
Threads

References:

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 4

4.1 Overview
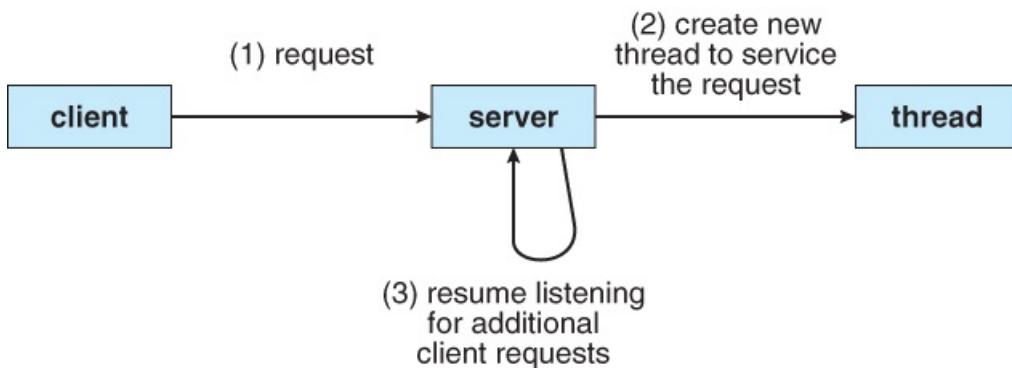
- A ***thread*** is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID.)
- Traditional (heavyweight) processes have a single thread of control - There is one program counter, and one sequence of instructions that can be carried out at any given time.
- As shown in Figure 4.1, multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.

Figure 4.1 - Single-threaded and multithreaded processes

4.1.1 Motivation

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others.
- This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.
- For example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input (keystrokes), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.
- Another example is a web server - Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork off separate processes for every incoming request. (The latter is how this sort of thing was done before the concept of threads was developed. A daemon would listen at a port, fork off a child for every incoming request to be processed, and then go back to listening to the port.)
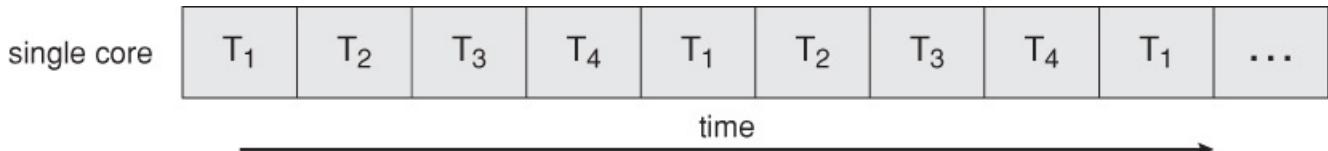
Figure 4.2 - Multithreaded server architecture

4.1.2 Benefits
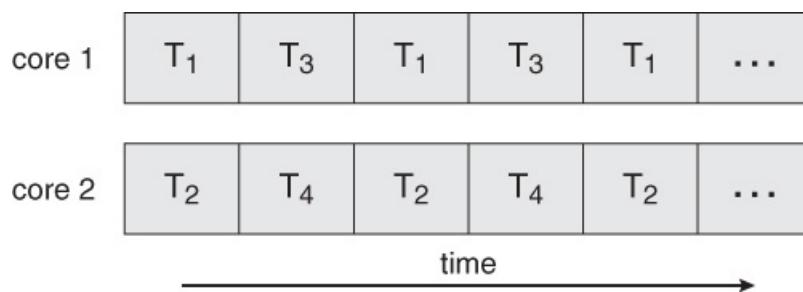
- There are four major categories of benefits to multi-threading:
    1. Responsiveness - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
    2. Resource sharing - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
    3. Economy - Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.
    4. Scalability, i.e. Utilization of multiprocessor architectures - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors. (Note that single threaded processes can still benefit from multi-processor architectures when there are multiple processes contending for the CPU, i.e. when the load average is above some certain threshold.)

4.2 Multicore Programming

- A recent trend in computer architecture is to produce chips with multiple **cores**, or CPUs on a single chip.
- A multi-threaded application running on a traditional single-core chip would have to interleave the threads, as shown in Figure 4.3. On a multi-core chip, however, the threads could be spread across the available cores, allowing true parallel processing, as shown in Figure 4.4.


**Figure 4.3 - Concurrent execution on a single-core system.**

Figure 4.4 - Parallel execution on a multicore system

- For operating systems, multi-core chips require new scheduling algorithms to make better use of the multiple cores available.
- As multi-threading becomes more pervasive and more important (thousands instead of tens of threads), CPUs have been developed to support more simultaneous threads per core in hardware.

4.2.1 Programming Challenges (New section, same content ?)

- For application programmers, there are five areas where multi-core chips present new challenges:
    1. **Identifying tasks** - Examining applications to find activities that can be performed concurrently.
    2. **Balance** - Finding tasks to run concurrently that provide equal value. I.e. don't waste a thread on trivial tasks.

3. **Data splitting** - To prevent the threads from interfering with one another.
4. **Data dependency** - If one task is dependent upon the results of another, then the tasks need to be synchronized to assure access in the proper order.
5. **Testing and debugging** - Inherently more difficult in parallel processing situations, as the race conditions become much more complex and difficult to identify.

4.2.2 Types of Parallelism (new)

In theory there are two different ways to parallelize the workload:

1. **Data parallelism** divides the data up amongst multiple cores (threads), and performs the same task on each subset of the data. For example dividing a large image up into pieces and performing the same digital image processing on each piece on different cores.
2. **Task parallelism** divides the different tasks to be performed among the different cores and performs them simultaneously.

In practice no program is ever divided up solely by one or the other of these, but instead by some sort of hybrid combination.

4.3 Multithreading Models

- There are two types of threads to be managed in a modern system: User threads and kernel threads.
- User threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs.
- Kernel threads are supported within the kernel of the OS itself. All modern OSes support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.
- In a specific implementation, the user threads must be mapped to kernel threads, using one of the following strategies.

4.3.1 Many-To-One Model

- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is very efficient.
- However, if a blocking system call is made, then the entire process blocks, even if the other user threads would otherwise be able to continue.
- Because a single kernel thread can operate only on a single CPU, the many-to-one model does not allow individual processes to be split across multiple CPUs.
- Green threads for Solaris and GNU Portable Threads implement the many-to-one model in the past, but few systems continue to do so today.
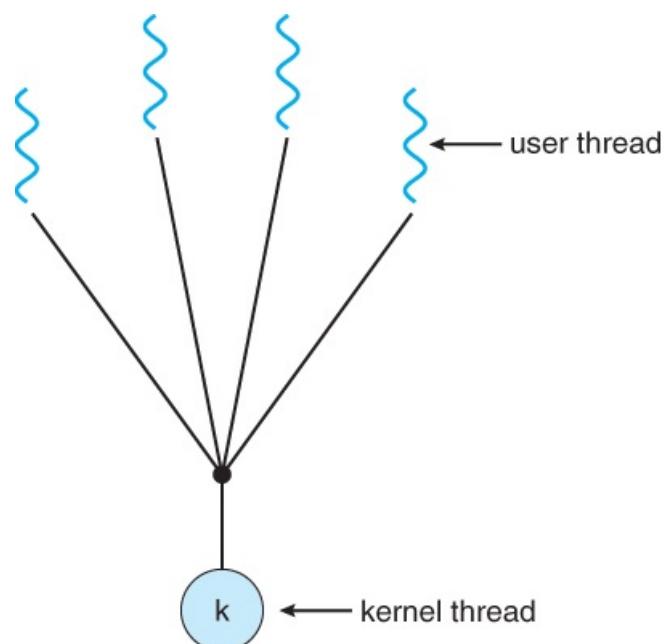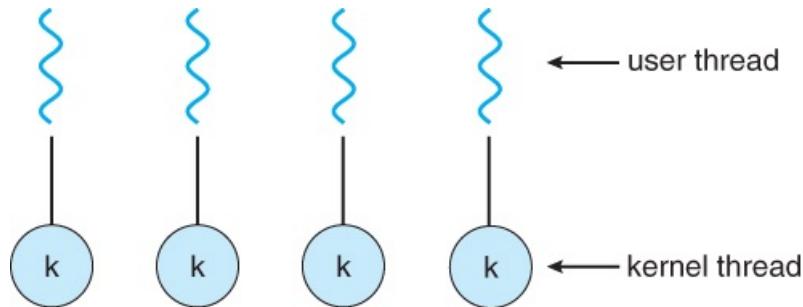

Figure 4.5 - Many-to-one model

4.3.2 One-To-One Model

- The one-to-one model creates a separate kernel thread to handle each user thread.
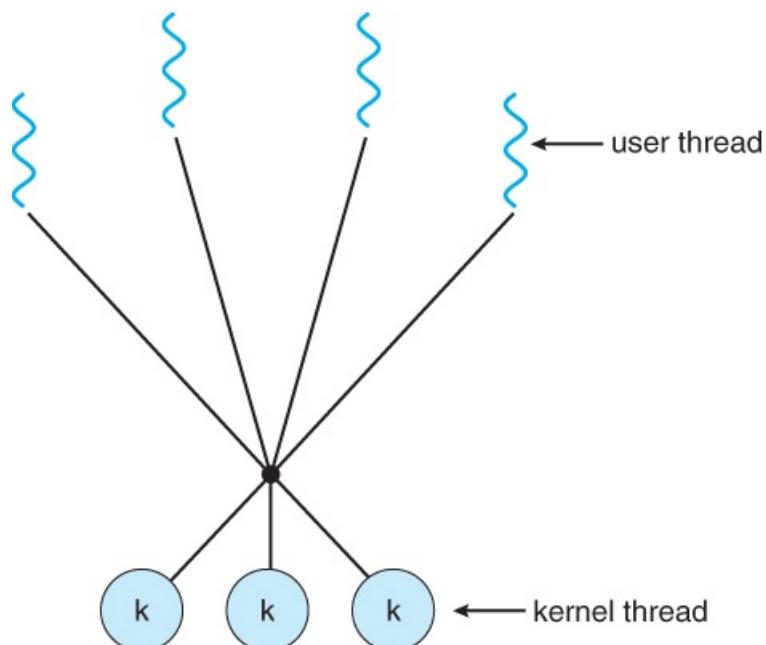
- One-to-one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.
- However the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.

Figure 4.6 - One-to-one model
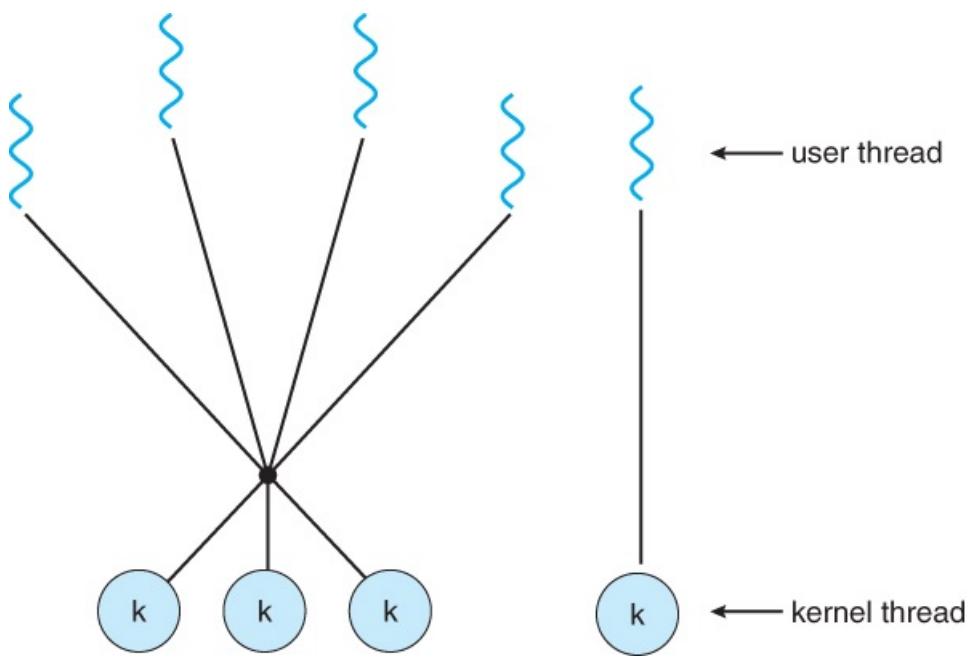
4.3.3 Many-To-Many Model

- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users have no restrictions on the number of threads created.
- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.
- Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.

Figure 4.7 - Many-to-many model

- One popular variation of the many-to-many model is the two-tier model, which allows either many-to-many or one-to-one operation.
- IRIX, HP-UX, and Tru64 UNIX use the two-tier model, as did Solaris prior to Solaris 9.

Figure 4.8 - Two-level model

4.4 Thread Libraries

- Thread libraries provide programmers with an API for creating and managing threads.
- Thread libraries may be implemented either in user space or in kernel space. The former involves API functions implemented solely within user space, with no kernel support. The latter involves system calls, and requires a kernel with thread library support.
- There are three main thread libraries in use today:
    1. POSIX Pthreads - may be provided as either a user or kernel library, as an extension to the POSIX standard.
    2. Win32 threads - provided as a kernel-level library on Windows systems.
    3. Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.
- The following sections will demonstrate the use of threads in all three systems for calculating the sum of integers from 0 to N in a separate thread, and storing the result in a variable "sum".

4.4.1 Pthreads

- The POSIX standard (IEEE 1003.1c) defines the *specification* for pThreads, not the *implementation*.
- pThreads are available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows.
- Global variables are shared amongst all threads.
- One thread can wait for the others to rejoin before continuing.
- pThreads begin execution in a specified function, in this example the runner( ) function:

```c
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
        return -1;
    }

    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Figure 4.6 Multithreaded C program using the Pthreads API.

Figure 4.9

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Figure 4.10 Pthread code for joining ten threads.

New

4.4.2 Windows Threads

- Similar to pThreads. Examine the code example to see the differences, which are mostly syntactic & nomenclature:

```c
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */

DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
     Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;
   /* perform some basic error checking */
   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }

   // create the thread
   ThreadHandle = CreateThread(
      NULL, // default security attributes
      0, // default stack size
      Summation, // thread function
      &Param, // parameter to thread function
      0, // default creation flags
      &ThreadId); // returns the thread identifier

   if (ThreadHandle != NULL) {
      // now wait for the thread to finish
      WaitForSingleObject(ThreadHandle,INFINITE);

      // close the thread handle
      CloseHandle(ThreadHandle);

      printf("sum = %d\n",Sum);
   }
}
```

Figure 4.7 Multithreaded C program using the Win32 API.

Figure 4.11

4.4.3 Java Threads

- ALL Java programs use Threads - even "common" single-threaded ones.
- The creation of new Threads requires Objects that implement the Runnable Interface, which means they contain a method "public void run( )" . Any descendant of the Thread class will naturally contain such a method. (In practice the run( ) method must be overridden / provided for the thread to have any practical functionality.)
- Creating a Thread Object does not start the thread running - To do that the program must call the Thread's "start( )" method. Start( ) allocates and initializes memory for the Thread, and then calls the run( ) method. (Programmers do not call run( ) directly.)
- Because Java does not support global variables, Threads must be passed a reference to a shared Object in order to share data, in this example the "Sum" Object.
- Note that the JVM runs on top of a native OS, and that the JVM specification does not specify what model to use for mapping Java threads to kernel threads. This decision is JVM implementation dependant, and may be one-to-one, many-to-many, or many to one.. (On a UNIX system the JVM normally uses PThreads and on a Windows system it normally uses windows threads.)

```
class Sum
{
  private int sum;

  public int getSum() {
    return sum;
  }

  public void setSum(int sum) {
    this.sum = sum;
  }
}

class Summation implements Runnable
{
  private int upper;
  private Sum sumValue;

  public Summation(int upper, Sum sumValue) {
    this.upper = upper;
    this.sumValue = sumValue;
  }

  public void run() {
    int sum = 0;
    for (int i = 0; i <= upper; i++)
      sum += i;
    sumValue.setSum(sum);
  }
}

public class Driver
{
  public static void main(String[] args) {
    if (args.length > 0) {
      if (Integer.parseInt(args[0]) < 0)
        System.err.println(args[0] + " must be >= 0.");
      else {
        // create the object to be shared
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);
        Thread thrd = new Thread(new Summation(upper, sumObject));
        thrd.start();
        try {
          thrd.join();
          System.out.println
            ("The sum of "+upper+" is "+sumObject.getSum());
        } catch (InterruptedException ie) { }
      }
    }
    else
      System.err.println("Usage: Summation <integer value>"); }
}
```

Figure 4.8   Java program for the summation of a non-negative integer.

Figure 4.12

4.5 Implicit Threading (Optional)

Shifts the burden of addressing the programming challenges outlined in section 4.2.1 above from the application programmer to the compiler and run-time libraries.

4.5.1 Thread Pools

- Creating new threads every time one is needed and then deleting it when it is done can be inefficient, and can also lead to a very large (unlimited) number of threads being created.
- An alternative solution is to create a number of threads when the process first starts, and put those threads into a **thread pool**.
  - Threads are allocated from the pool as needed, and returned to the pool when no longer needed.
  - When no threads are available in the pool, the process may have to wait until one becomes available.
- The (maximum) number of threads available in a thread pool may be determined by adjustable parameters, possibly dynamically in response to changing system loads.
- Win32 provides thread pools through the "PoolFunction" function. Java also provides support for thread pools through the java.util.concurrent package, and Apple supports thread pools under the Grand Central Dispatch architecture..

4.5.2 OpenMP

- OpenMP is a set of compiler directives available for C, C++, or FORTRAN programs that instruct the compiler to automatically generate parallel code where appropriate.
- For example, the directive:

```
#pragma omp parallel
{
      /* some parallel code here */
}
```

would cause the compiler to create as many threads as the machine has cores available, (e.g. 4 on a quad-core machine), and to run the parallel block of code, (known as a **parallel region**) on each of the threads.

- Another sample directive is "`#pragma omp parallel for`", which causes the for loop immediately following it to be parallelized, dividing the iterations up amongst the available cores.

4.5.3 Grand Central Dispatch, GCD

- GCD is an extension to C and C++ available on Apple's OSX and iOS operating systems to support parallelism.
- Similar to OpenMP, users of GCD define **blocks** of code to be executed either serially or in parallel by placing a carat just before an opening curly brace, i.e. `^{ printf( "I am a block.\n" ); }`
- GCD schedules blocks by placing them on one of several **dispatch queues**.
  - Blocks placed on a serial queue are removed one by one. The next block cannot be removed for scheduling until the previous block has completed.
  - There are three concurrent queues, corresponding roughly to low, medium, or high priority. Blocks are also removed from these queues one by one, but several may be removed and dispatched without waiting for others to finish first, depending on the availability of threads.
- Internally GCD manages a pool of POSIX threads which may fluctuate in size depending on load conditions.

4.5.4 Other Approaches

There are several other approaches available, including Microsoft's Threading Building Blocks (TBB) and other products, and Java's util.concurrent package.

4.6 Threading Issues

4.6.1 The fork( ) and exec( ) System Calls

- Q: If one thread forks, is the entire process copied, or is the new process single-threaded?

- A: System dependant.
- A: If the new process execs right away, there is no need to copy all the other threads. If it doesn't, then the entire process should be copied.
- A: Many versions of UNIX provide multiple versions of the fork call for this purpose.

4.6.2 Signal Handling

- Q: When a multi-threaded process receives a signal, to what thread should that signal be delivered?
- A: There are four major options:
  1. Deliver the signal to the thread to which the signal applies.
  2. Deliver the signal to every thread in the process.
  3. Deliver the signal to certain threads in the process.
  4. Assign a specific thread to receive all signals in a process.
- The best choice may depend on which specific signal is involved.
- UNIX allows individual threads to indicate which signals they are accepting and which they are ignoring. However the signal can only be delivered to one thread, which is generally the first thread that is accepting that particular signal.
- UNIX provides two separate system calls, `kill( pid, signal )` and `pthread_kill( tid, signal )`, for delivering signals to processes or specific threads respectively.
- Windows does not support signals, but they can be emulated using Asynchronous Procedure Calls (APCs). APCs are delivered to specific threads, not processes.

4.6.3 Thread Cancellation

- Threads that are no longer needed may be cancelled by another thread in one of two ways:
  1. **Asynchronous Cancellation** cancels the thread immediately.
  2. **Deferred Cancellation** sets a flag indicating the thread should cancel itself when it is convenient. It is then up to the cancelled thread to check this flag periodically and exit nicely when it sees the flag set.
- (Shared) resource allocation and inter-thread data transfers can be problematic with asynchronous cancellation.

4.6.4 Thread-Local Storage (was 4.4.5 Thread-Specific Data)

- Most data is shared among threads, and this is one of the major benefits of using threads in the first place.
- However sometimes threads need thread-specific data also.
- Most major thread libraries (pThreads, Win32, Java) provide support for thread-specific data, known as **thread-local storage** or **TLS.** Note that this is more like static data than local variables,because it does not cease to exist when the function ends.

4.6.5 Scheduler Activations

- Many implementations of threads provide a virtual processor as an interface between the user thread and the kernel thread, particularly for the many-to-many or two-tier models.
- This virtual processor is known as a "Lightweight Process", LWP.
  - There is a one-to-one correspondence between LWPs and kernel threads.
  - The number of kernel threads available, (and hence the number of LWPs) may change dynamically.
  - The application (user level thread library) maps user threads onto available LWPs.
  - kernel threads are scheduled onto the real processor(s) by the OS.
  - The kernel communicates to the user-level thread library when certain events occur (such as a thread about to block) via an *upcall*, which is handled in the thread library by an *upcall handler*. The upcall also provides a new LWP for the upcall handler to run on, which it can then use to reschedule the user thread that is about to become blocked. The OS will also issue upcalls when a thread becomes unblocked, so the thread library can make appropriate adjustments.
- If the kernel thread blocks, then the LWP blocks, which blocks the user thread.
- Ideally there should be at least as many LWPs available as there could be concurrently blocked kernel threads. Otherwise if all LWPs are blocked, then user threads will have to wait for one to become available.
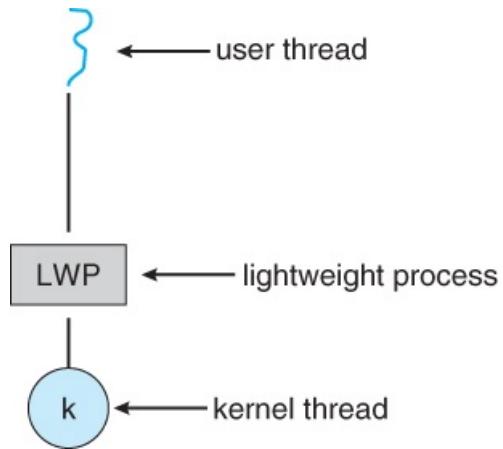
Figure 4.13 - Lightweight process (LWP)

4.7 Operating-System Examples (Optional)

4.7.1 Windows XP Threads

- The Win32 API thread library supports the one-to-one thread model
- Win32 also provides the **fiber** library, which supports the many-to-many model.
- Win32 thread components include:
  - Thread ID
  - Registers
  - A user stack used in user mode, and a kernel stack used in kernel mode.
  - A private storage area used by various run-time libraries and dynamic link libraries (DLLs).
- The key data structures for Windows threads are the ETHREAD (executive thread block), KTHREAD (kernel thread block), and the TEB (thread environment block). The ETHREAD and KTHREAD structures exist entirely within kernel space, and hence are only accessible by the kernel, whereas the TEB lies within user space, as illustrated in Figure 4.10:
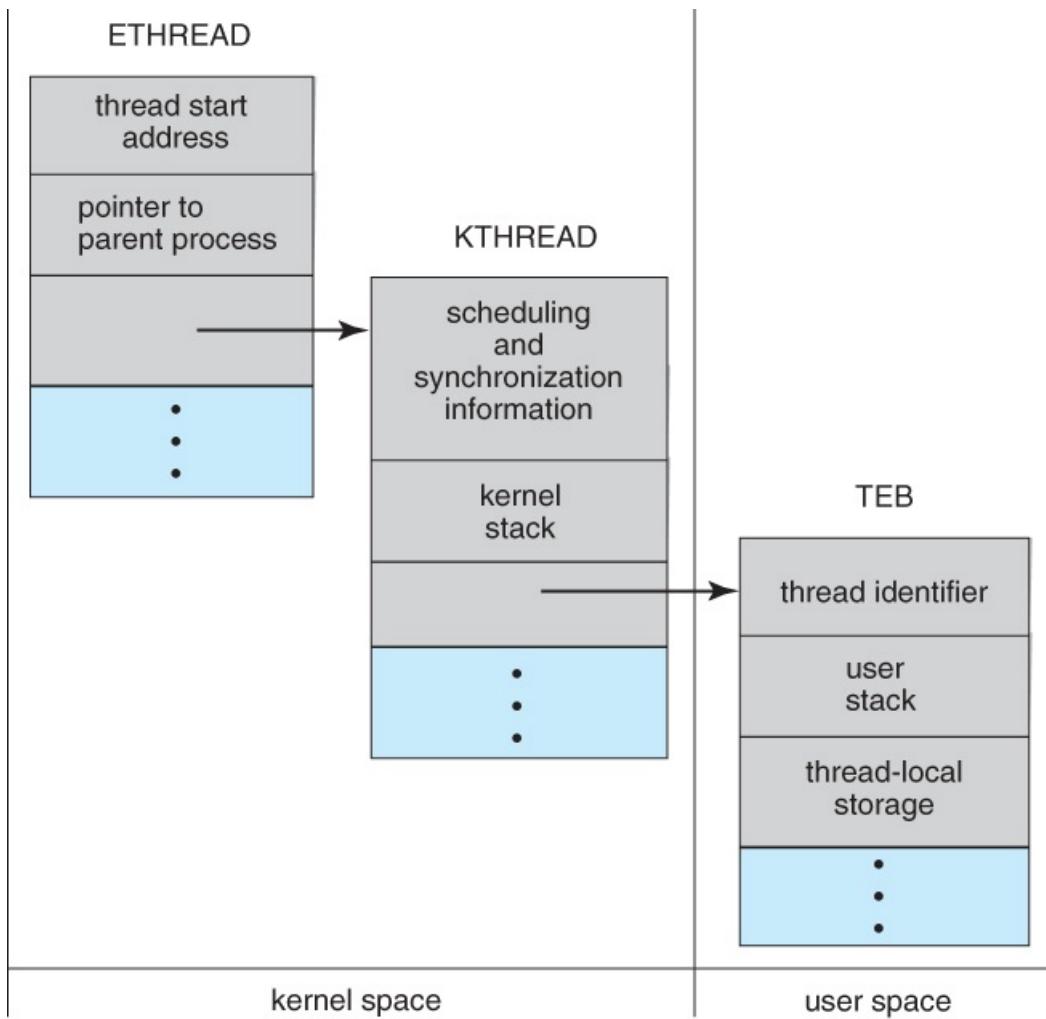
Figure 4.14 - Data structures of a Windows thread

4.7.2 Linux Threads

- Linux does not distinguish between processes and threads - It uses the more generic term "tasks".
- The traditional fork( ) system call completely duplicates a process (task), as described earlier.
- An alternative system call, clone( ) allows for varying degrees of sharing between the parent and child tasks, controlled by flags such as those shown in the following table:

flag	Meaning
CLONE_FS	File-system information is shared
CLONE_VM	The same memory space is shared
CLONE_SIGHAND	Signal handlers are shared
CLONE_FILES	The set of open files is shared

- Calling clone( )with no flags set is equivalent to fork( ). Calling clone( ) with CLONE_FS, CLONE_VM, CLONE_SIGHAND, and CLONE_FILES is equivalent to creating a thread, as all of these data structures will be shared.
- Linux implements this using a structure `task_struct`, which essentially provides a level of indirection to task resources. When the flags are not set, then the resources pointed to by the structure are copied, but if the flags are set, then only the pointers to the resources are copied, and hence the resources are shared. (Think of a deep copy versus a shallow copy in OO programming.)
- **(Removed from 9th edition)** Several distributions of Linux now support the NPTL (Native POXIS Thread Library)
    - POSIX compliant.
    - Support for SMP (symmetric multiprocessing), NUMA (non-uniform memory access), and multicore processors.
    - Support for hundreds to thousands of threads.

4.8 Summary

Process Synchronization

References:

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 5

Warning: This chapter requires some heavy thought. As you read each of the algorithms below, you need to satisfy yourself that they do indeed work under all conditions. Think about it, and don't just accept them at face value.

5.1 Background

- Recall that back in Chapter 3 we looked at cooperating processes (those that can effect or be effected by other simultaneously running processes), and as an example, we used the producer-consumer cooperating processes:

  Producer code from chapter 3:

  ```
  item nextProduced;

  while( true ) {

      /* Produce an item and store it in nextProduced */
      nextProduced = makeNewItem( . . . );

      /* Wait for space to become available */
      while( ( ( in + 1 ) % BUFFER_SIZE ) == out )
          ; /* Do nothing */

      /* And then store the item and repeat the loop. */
      buffer[ in ] = nextProduced;
      in = ( in + 1 ) % BUFFER_SIZE;

  }
  ```

  Consumer code from chapter 3:

  ```
  item nextConsumed;

  while( true ) {

      /* Wait for an item to become available */
      while( in == out )
          ; /* Do nothing */

      /* Get the next available item */
      nextConsumed = buffer[ out ];
      out = ( out + 1 ) % BUFFER_SIZE;

      /* Consume the item in nextConsumed
          ( Do something with it ) */

  }
  ```

  - The only problem with the above code is that the maximum number of items which can be placed into the buffer is BUFFER_SIZE - 1. One slot is unavailable because there always has to be a gap between the producer and the consumer.
  - We could try to overcome this deficiency by introducing a counter variable, as shown in the following code segments:

Producer Process:

```
while (true)
{
    /* produce an item in nextProduced */
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer Process:

```
while (true)
{
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in nextConsumed */
}
```

- Unfortunately we have now introduced a new problem, because both the producer and the consumer are adjusting the value of the variable counter, which can lead to a condition known as a **race condition**. In this condition a piece of code may or may not work correctly, depending on which of two simultaneous processes executes first, and more importantly if one of the processes gets interrupted such that the other process runs between important steps of the first process. (Bank balance example discussed in class.)
- The particular problem above comes from the producer executing "counter++" at the same time the consumer is executing "counter--". If one process gets part way through making the update and then the other process butts in, the value of counter can get left in an incorrect state.
- But, you might say, "Each of those are single instructions - How can they get interrupted halfway through?" The answer is that although they are single instructions in C++, they are actually three steps each at the hardware level: (1) Fetch counter from memory into a register, (2) increment or decrement the register, and (3) Store the new value of counter back to memory. If the instructions from the two processes get interleaved, there could be serious problems, such as illustrated by the following:

Producer:

$$register_1 = \text{counter}$$
$$register_1 = register_1 + 1$$
$$\text{counter} = register_1$$

Consumer:

$$register_2 = \text{counter}$$
$$register_2 = register_2 - 1$$
$$\text{counter} = register_2$$

Interleaving:

T_0:	producer	execute	$register_1 = \text{counter}$	$\{register_1 = 5\}$
T_1:	producer	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	consumer	execute	$register_2 = \text{counter}$	$\{register_2 = 5\}$
T_3:	consumer	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	producer	execute	$\text{counter} = register_1$	$\{counter = 6\}$
T_5:	consumer	execute	$\text{counter} = register_2$	$\{counter = 4\}$

- **Exercise:** What would be the resulting value of counter if the order of statements T4 and T5 were reversed? (What **should** the value of counter be after one producer and one consumer, assuming the original value was 5?)
- Note that race conditions are **notoriously difficult** to identify and debug, because by their very nature they only occur on rare occasions, and only when the timing is just exactly right. (or wrong! :-)) Race conditions are also very difficult to reproduce. :-(
- Obviously the solution is to only allow one process at a time to manipulate the value "counter". This is a very common occurrence among cooperating processes, so lets look at some ways in which this is done, as well as some classic problems in this area.

5.2 The Critical-Section Problem

- The producer-consumer problem described above is a specific example of a more general situation known as the **critical section** problem. The general idea is that in a number of cooperating processes, each has a critical section of code, with the following conditions and terminologies:
  - Only one process in the group can be allowed to execute in their critical section at any one time. If one process is already executing their critical section and another process wishes to do so, then the second process must be made to wait until the first process has completed their critical section work.
  - The code preceding the critical section, and which controls access to the critical section, is termed the entry section. It acts like a carefully controlled locking door.
  - The code following the critical section is termed the exit section. It generally releases the lock on someone else's door, or at least lets the world know that they are no longer in their critical section.
  - The rest of the code not included in either the critical section or the entry or exit sections is termed the remainder section.

```
do {
```

entry section

critical section

exit section

remainder section

```
} while (TRUE);
```

Figure 5.1 - General structure of a typical process Pi

- A solution to the critical section problem must satisfy the following three conditions:
  1. **Mutual Exclusion** - Only one process at a time can be executing in their critical section.
  2. **Progress** - If no process is currently executing in their critical section, and one or more processes want to execute their critical section, then only the processes not in their remainder sections can participate in the decision, and the decision cannot be postponed indefinitely. (I.e. processes cannot be blocked forever waiting to get into their critical sections.)
  3. **Bounded Waiting** - There exists a limit as to how many other processes can get into their critical sections after a process requests entry into their critical section and before that request is granted. (I.e. a process requesting entry into their critical section will get a turn eventually, and there is a limit as to how many other processes get to go first.)
- We assume that all processes proceed at a non-zero speed, but no assumptions can be made regarding the *relative* speed of one process versus another.
- Kernel processes can also be subject to race conditions, which can be especially problematic when updating commonly shared kernel data structures such as open file tables or virtual memory management. Accordingly kernels can take on one of two forms:
  - Non-preemptive kernels do not allow processes to be interrupted while in kernel mode. This eliminates the possibility of kernel-mode race conditions, but requires kernel mode operations to complete very quickly, and can be problematic for real-time systems, because timing cannot be guaranteed.
  - Preemptive kernels allow for real-time operations, but must be carefully written to avoid race conditions. This can be especially tricky on SMP systems, in which multiple kernel processes may be running simultaneously on different processors.
- Non-preemptive kernels include Windows XP, 2000, traditional UNIX, and Linux prior to 2.6; Preemptive kernels include Linux 2.6 and later, and some commercial UNIXes such as Solaris and IRIX.

5.3 Peterson's Solution

- Peterson's Solution is a classic software-based solution to the critical section problem. It is unfortunately not guaranteed to work on modern hardware, due to vagaries of load and store operations, but it illustrates a number of important concepts.
- Peterson's solution is based on two processes, P0 and P1, which alternate between their critical sections and remainder sections. For convenience of discussion, "this" process is Pi, and the "other" process is Pj. (I.e. j = 1 - i)
- Peterson's solution requires two shared data items:
  - **int turn** - Indicates whose turn it is to enter into the critical section. If turn = = i, then process i is allowed into their critical section.
  - **boolean flag[2]** - Indicates when a process *wants to* enter into their critical section. When process i wants to enter their critical section, it sets flag[i] to true.
- In the following diagram, the entry and exit sections are enclosed in boxes.
  - In the entry section, process i first raises a flag indicating a desire to enter the critical section.
  - Then turn is set to *j* to allow the *other* process to enter their critical section *if process j so desires.*
  - The while loop is a busy loop (notice the semicolon at the end), which makes process i wait as long as process j has the turn and wants to enter the critical section.
  - Process i lowers the flag[i] in the exit section, allowing process j to continue if it has been waiting.

```
do {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Figure 5.2 - The structure of process Pi in Peterson's solution.

- To prove that the solution is correct, we must examine the three conditions listed above:
  1. **Mutual exclusion** - If one process is executing their critical section when the other wishes to do so, the second process will become blocked by the flag of the first process. If both processes attempt to enter at the same time, the last process to execute "turn = j" will be blocked.
  2. **Progress** - Each process can only be blocked at the while if the other process wants to use the critical section (flag[j] = = true), AND it is the other process's turn to use the critical section (turn = = j). If both of those conditions are true, then the other process (j) will be allowed to enter the critical section, and upon exiting the critical section, will set flag[j] to false, releasing process i. The shared variable turn assures that only one process at a time can be blocked, and the flag variable allows one process to release the other when exiting their critical section.
  3. **Bounded Waiting** - As each process enters their entry section, they set the turn variable to be the other processes turn. Since no process ever sets it back to their own turn, this ensures that each process will have to let the other process go first at most one time before it becomes their turn again.
- Note that the instruction "turn = j" is ***atomic,*** that is it is a single machine instruction which cannot be interrupted.

5.4 Synchronization Hardware

- To generalize the solution(s) expressed above, each process when entering their critical section must set some sort of ***lock***, to prevent other processes from entering their critical sections simultaneously, and must release the lock when exiting their critical section, to allow other processes to proceed. Obviously it must be possible to attain the lock only when no other process has already set a lock. Specific implementations of this general procedure can get quite complicated, and may include hardware solutions as outlined in this section.
- One simple solution to the critical section problem is to simply prevent a process from being interrupted while in their critical section, which is the approach taken by non preemptive kernels. Unfortunately this does not work well in multiprocessor environments, due to the difficulties in disabling and the re-enabling interrupts on all processors. There is also a question as to how this approach affects timing if the clock interrupt is disabled.
- Another approach is for hardware to provide certain ***atomic*** operations. These operations are guaranteed to operate as a single instruction, without interruption. One such operation is the "Test and Set", which simultaneously sets a boolean lock variable and returns its previous value, as shown in Figures 5.3 and 5.4:

```
boolean TestAndSet(boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

Figure 5.3 The definition of the `TestAndSet()` instruction.

```
do {
    while (TestAndSetLock(&lock))
        ; // do nothing

        // critical section

    lock = FALSE;

        // remainder section
}while (TRUE);
```

Figure 5.4 Mutual-exclusion implementation with `TestAndSet()`.

Figures 5.3 and 5.4 illustrate "test_and_set( )" function

- Another variation on the test-and-set is an atomic swap of two booleans, as shown in Figures 5.5 and 5.6:

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;

    return temp;
}
```

Figure 5.5 The definition of the `compare_and_swap()` instruction.

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

        /* critical section */

    lock = 0;

        /* remainder section */
} while (true);
```

Figure 5.6 Mutual-exclusion implementation with the `compare_and_swap()` instruction.

- The above examples satisfy the mutual exclusion requirement, but unfortunately do not guarantee bounded waiting. If

there are multiple processes trying to get into their critical sections, there is no guarantee of what order they will enter, and any one process could have the bad luck to wait forever until they got their turn in the critical section. (Since there is no guarantee as to the relative **rates** of the processes, a very fast process could theoretically release the lock, whip through their remainder section, and re-lock the lock before a slower process got a chance. As more and more processes are involved vying for the same resource, the odds of a slow process getting locked out completely increase.)

- Figure 5.7 illustrates a solution using test-and-set that does satisfy this requirement, using two shared data structures, `boolean lock` and `boolean waiting[ N ]`, where N is the number of processes in contention for critical sections:

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

        // critical section

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;

        // remainder section
}while (TRUE);
```

Figure 5.7 Bounded-waiting mutual exclusion with TestAndSet( ).

- The key feature of the above algorithm is that a process blocks on the AND of the critical section being locked and that this process is in the waiting state. When exiting a critical section, the exiting process does not just unlock the critical section and let the other processes have a free-for-all trying to get in. Rather it first looks in an orderly progression (starting with the next process on the list) for a process that has been waiting, and if it finds one, then it releases that particular process from its waiting state, without unlocking the critical section, thereby allowing a specific process into the critical section while continuing to block all the others. Only if there are no other processes currently waiting is the general lock removed, allowing the next process to come along access to the critical section.
- Unfortunately, hardware level locks are especially difficult to implement in multi-processor architectures. Discussion of such issues is left to books on advanced computer architecture.

5.5 Mutex Locks

- The hardware solutions presented above are often difficult for ordinary programmers to access, particularly on multi-processor machines, and particularly because they are often platform-dependent.
- Therefore most systems offer a software API equivalent called **mutex locks** or simply **mutexes.** (For mutual exclusion)
- The terminology when using mutexes is to **acquire** a lock prior to entering a critical section, and to **release it when exiting, as shown in Figure 5.8:**

```
do {
```

```
acquire lock
```

critical section

```
release lock
```

remainder section

```
} while (TRUE);
```

Figure 5.8 - Solution to the critical-section problem using mutex locks

- Just as with hardware locks, the acquire step will block the process if the lock is in use by another process, and both the acquire and release operations are atomic.
- Acquire and release can be implemented as shown here, based on a boolean variable "available":

Acquire:

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}
```

Release:

```
release() {
    available = true;
}
```

- One problem with the implementation shown here, (and in the hardware solutions presented earlier), is the busy loop used to block processes in the acquire phase. These types of locks are referred to as **spinlocks**, because the CPU just sits and spins while blocking the process.
- Spinlocks are wasteful of cpu cycles, and are a really bad idea on single-cpu single-threaded machines, because the spinlock blocks the entire computer, and doesn't allow any other process to release the lock. (Until the scheduler kicks the spinning process off of the cpu.)
- On the other hand, spinlocks do not incur the overhead of a context switch, so they are effectively used on multi-threaded machines when it is expected that the lock will be released after a short time.

5.6 Semaphores

- A more robust alternative to simple mutexes is to use **semaphores**, which are integer variables for which only two (atomic) operations are defined, the wait and signal operations, as shown in the following figure.
- Note that not only must the variable-changing steps (S-- and S++) be indivisible, it is also necessary that for the wait operation when the test proves false that there be no interruptions before S gets decremented. It IS okay, however, for the busy loop to be interrupted when the test is true, which prevents the system from hanging forever.

Wait:

```
wait(S) {
    while S <= 0
        ; // no-op
    S--;
}
```

Signal:

```
signal(S) {
    S++;
}
```

5.6.1 Semaphore Usage

- In practice, semaphores can take on one of two forms:
  - **Binary semaphores** can take on one of two values, 0 or 1. They can be used to solve the critical section problem as described above, and can be used as mutexes on systems that do not provide a separate mutex mechanism.. The use of mutexes for this purpose is shown in Figure 6.9 (from the 8th edition) below.

```
do {
    waiting(mutex);

        // critical section

    signal(mutex);

        // remainder section
}while (TRUE);
```

Mutual-exclusion implementation with semaphores. (From 8th edition.)

  - **Counting semaphores** can take on any integer value, and are usually used to count the number remaining of some limited resource. The counter is initialized to the number of such resources available in the system, and whenever the counting semaphore is greater than zero, then a process can enter a critical section and use one of the resources. When the counter gets to zero (or negative in some implementations), then the process blocks until another process frees up a resource and increments the counting semaphore with a signal call. (The binary semaphore can be seen as just a special case where the number of resources initially available is just one.)
  - Semaphores can also be used to synchronize certain operations between processes. For example, suppose it is important that process P1 execute statement S1 before process P2 executes statement S2.
    - First we create a semaphore named synch that is shared by the two processes, and initialize it to zero.
    - Then in process P1 we insert the code:

```
S1;
signal( synch );
```

    - and in process P2 we insert the code:

```
wait( synch );
S2;
```

    - Because synch was initialized to 0, process P2 will block on the wait until after P1 executes the call to signal.

5.6.2 Semaphore Implementation

- The big problem with semaphores as described above is the busy loop in the wait call, which consumes CPU cycles without doing any useful work. This type of lock is known as a **spinlock**, because the lock just sits there and spins while it waits. While this is generally a bad thing, it does have the advantage of not invoking context switches, and so it is sometimes used in multi-processing systems when the wait time is expected to be short - One thread spins on one processor while another completes their critical section on another processor.
- An alternative approach is to block a process when it is forced to wait for an available semaphore, and swap it out of the CPU. In this implementation each semaphore needs to maintain a list of processes that are blocked waiting for it, so that one of the processes can be woken up and swapped back in when the semaphore becomes available. (Whether it gets swapped back into the CPU immediately or whether it needs to hang out in the ready queue for a while is a scheduling problem.)
- The new definition of a semaphore and the corresponding wait and signal operations are shown as follows:

Semaphore Structure:

```
typedef struct {
        int value;
        struct process *list;
} semaphore;
```

Wait Operation:

```
wait(semaphore *S) {
            S->value--;
            if (S->value < 0) {
                        add this process to S->list;
                        block();
            }
}
```

Signal Operation:

```
signal(semaphore *S) {
            S->value++;
            if (S->value <= 0) {
                        remove a process P from S->list;
                        wakeup(P);
            }
}
```

- Note that in this implementation the value of the semaphore can actually become negative, in which case its magnitude is the number of processes waiting for that semaphore. This is a result of decrementing the counter before checking its value.
- Key to the success of semaphores is that the wait and signal operations be atomic, that is no other process can execute a wait or signal on the same semaphore at the same time. (Other processes could be allowed to do other things, including working with other semaphores, they just can't have access to **this** semaphore.) On single processors this can be implemented by disabling interrupts during the execution of wait and signal; Multiprocessor systems have to use more complex methods, including the use of spinlocking.

5.6.3 Deadlocks and Starvation

- One important problem that can arise when using semaphores to block processes waiting for a limited resource is the problem of **deadlocks**, which occur when multiple processes are blocked, each waiting for a resource that can only be freed by one of the other (blocked) processes, as illustrated in the following example. (Deadlocks are covered more completely in chapter 7.)

```
        P₀                    P₁

    wait(S);              wait(Q);
    wait(Q);              wait(S);
        .                     .
        .                     .
        .                     .
    signal(S);           signal(Q);
    signal(Q);           signal(S);
```

- Another problem to consider is that of ***starvation***, in which one or more processes gets blocked forever, and never get a chance to take their turn in the critical section. For example, in the semaphores above, we did not specify the algorithms for adding processes to the waiting queue in the semaphore in the wait( ) call, or selecting one to be removed from the queue in the signal( ) call. If the method chosen is a FIFO queue, then every process will eventually get their turn, but if a LIFO queue is implemented instead, then the first process to start waiting could starve.

5.6.4 Priority Inversion

- A challenging scheduling problem arises when a high-priority process gets blocked waiting for a resource that is currently held by a low-priority process.
- If the low-priority process gets pre-empted by one or more medium-priority processes, then the high-priority process is essentially made to wait for the medium priority processes to finish before the low-priority process can release the needed resource, causing a ***priority inversion.*** If there are enough medium-priority processes, then the high-priority process may be forced to wait for a very long time.
- One solution is a ***priority-inheritance protocol,*** in which a low-priority process holding a resource for which a high-priority process is waiting will temporarily inherit the high priority from the waiting process. This prevents the medium-priority processes from preempting the low-priority process until it releases the resource, blocking the priority inversion problem.
- The book has an interesting discussion of how a priority inversion almost doomed the Mars Pathfinder mission, and how the problem was solved when the priority inversion was stopped. Full details are available online at http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.htm

5.7 Classic Problems of Synchronization

The following classic problems are used to test virtually every new proposed synchronization algorithm.

5.7.1 The Bounded-Buffer Problem

- This is a generalization of the producer-consumer problem wherein access is controlled to a shared group of buffers of a limited size.
- In this solution, the two counting semaphores "full" and "empty" keep track of the current number of full and empty buffers respectively (and initialized to 0 and N respectively.) The binary semaphore mutex controls access to the critical section. The producer and consumer processes are nearly identical - One can think of the producer as producing full buffers, and the consumer producing empty buffers.

```
do {
    . . .
    // produce an item in nextp
    . . .
    wait(empty);
    wait(mutex);
    . . .
    // add nextp to buffer
    . . .
    signal(mutex);
    signal(full);
}while (TRUE);
```

Figure 5.9 The structure of the producer process.

```
do {
    wait(full);
    wait(mutex);
    . . .
    // remove an item from buffer to nextc
    . . .
    signal(mutex);
    signal(empty);
    . . .
    // consume the item in nextc
    . . .
}while (TRUE);
```

Figure 5.10 The structure of the consumer process.

Figures 5.9 and 5.10 use variables next_produced and next_consumed

5.7.2 The Readers-Writers Problem

- In the readers-writers problem there are some processes (termed readers) who only read the shared data, and never change it, and there are other processes (termed writers) who may change the data in addition to or instead of reading it. There is no limit to how many readers can access the data simultaneously, but when a writer accesses the data, it needs exclusive access.
- There are several variations to the readers-writers problem, most centered around relative priorities of readers versus writers.
  - The *first* readers-writers problem gives priority to readers. In this problem, if a reader wants access to the data, and there is not already a writer accessing it, then access is granted to the reader. A solution to this problem can lead to starvation of the writers, as there could always be more readers coming along to access the data. (A steady stream of readers will jump ahead of waiting writers as long as there is currently already another reader accessing the data, because the writer is forced to wait until the data is idle, which may never happen if there are enough readers.)
  - The *second* readers-writers problem gives priority to the writers. In this problem, when a writer wants access to the data it jumps to the head of the queue - All waiting readers are blocked, and the writer gets access to the data as soon as it becomes available. In this solution the readers may be starved by a steady stream of writers.
- The following code is an example of the first readers-writers problem, and involves an important counter and two binary semaphores:
  - readcount is used by the reader processes, to count the number of readers currently accessing the data.
  - mutex is a semaphore used only by the readers for controlled access to readcount.
  - rw_mutex is a semaphore used to block and release the writers. The first reader to access the data

will set this lock and the last reader to exit will release it; The remaining readers do not touch `rw_mutex`. (Eighth edition called this variable `wrt`.)
- Note that the first reader to come along will block on `rw_mutex` if there is currently a writer accessing the data, and that all following readers will only block on `mutex` for their turn to increment `readcount`.

```
do {
    wait(rw_mutex);
        .  .  .
    /* writing is performed */
        .  .  .
    signal(rw_mutex);
} while (true);
```

Figure 5.11   The structure of a writer process.
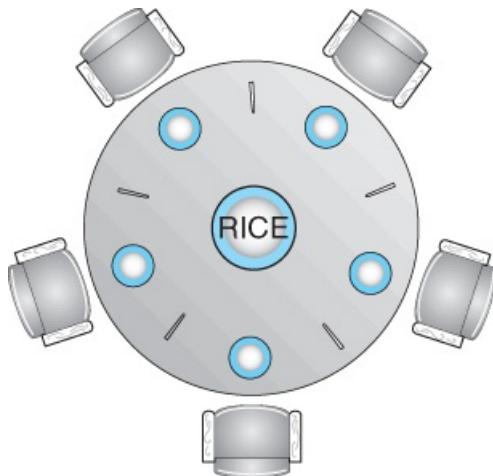
```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
        .  .  .
    /* reading is performed */
        .  .  .
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);
```

Figure 5.12   The structure of a reader process.

- Some hardware implementations provide specific reader-writer locks, which are accessed using an argument specifying whether access is requested for reading or writing. The use of reader-writer locks is beneficial for situation in which: (1) processes can be easily identified as either readers or writers, and (2) there are significantly more readers than writers, making the additional overhead of the reader-writer lock pay off in terms of increased concurrency of the readers.

5.7.3 The Dining-Philosophers Problem

- The dining philosophers problem is a classic synchronization problem involving the allocation of limited resources amongst a group of processes in a deadlock-free and starvation-free manner:
  - Consider five philosophers sitting around a table, in which there are five chopsticks evenly distributed and an endless bowl of rice in the center, as shown in the diagram below. (There is exactly one chopstick between each pair of dining philosophers.)
  - These philosophers spend their lives alternating between two activities: eating and thinking.
  - When it is time for a philosopher to eat, it must first acquire two chopsticks - one from their left and one from their right.
  - When a philosopher thinks, it puts down both chopsticks in their original locations.

Figure 5.13 - The situation of the dining philosophers

- One possible solution, as shown in the following code section, is to use a set of five semaphores (chopsticks[5]), and to have each hungry philosopher first wait on their left chopstick (chopsticks[i]), and then wait on their right chopstick (chopsticks[(i + 1) % 5])
- But suppose that all five philosophers get hungry at the same time, and each starts by picking up their left chopstick. They then look for their right chopstick, but because it is unavailable, they wait for it, forever, and eventually all the philosophers starve due to the resulting deadlock.

```
do {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
        . . .
    // eat
        . . .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
        . . .
    // think
        . . .
}while (TRUE);
```

Figure 5.14 - The structure of philosopher i.

- Some potential solutions to the problem include:
  - Only allow four philosophers to dine at the same time. (Limited simultaneous processes.)
  - Allow philosophers to pick up chopsticks only when both are available, in a critical section. (All or nothing allocation of critical resources.)
  - Use an asymmetric solution, in which odd philosophers pick up their left chopstick first and even philosophers pick up their right chopstick first. (Will this solution always work? What if there are an even number of philosophers?)
- Note carefully that a deadlock-free solution to the dining philosophers problem does not necessarily guarantee a starvation-free one. (While some or even most of the philosophers may be able to get on with their normal lives of eating and thinking, there may be one unlucky soul who never seems to be able to get both chopsticks at the same time. :-(

5.8 Monitors

- Semaphores can be very useful for solving concurrency problems, **but only if programmers use them properly.** If even one process fails to abide by the proper use of semaphores, either accidentally or deliberately, then the whole system breaks down. (And since concurrency problems are by definition rare events, the problem code may easily go unnoticed and/or be heinous to debug.)
- For this reason a higher-level language construct has been developed, called **monitors**.

5.8.1 Monitor Usage

- A monitor is essentially a class, in which all data is private, and with the special restriction that only one method within any given monitor object may be active at the same time. An additional restriction is that monitor methods may only access the shared data within the monitor and any data passed to them as parameters. I.e. they cannot access any data external to the monitor.

```
monitor monitor name
{
    // shared variable declarations

    procedure P1 ( . . . ) {
        . . .
    }

    procedure P2 ( . . . ) {
        . . .
    }


        .

        .

        .
    procedure Pn ( . . . ) {
        . . .
    }

    initialization code ( . . . ) {
        . . .
    }
}
```
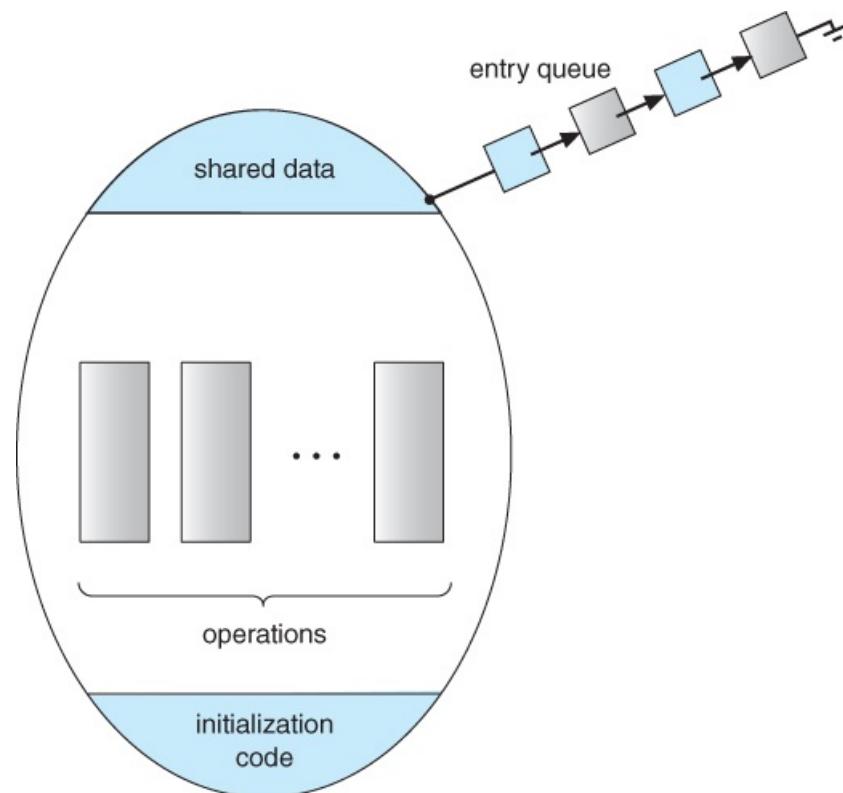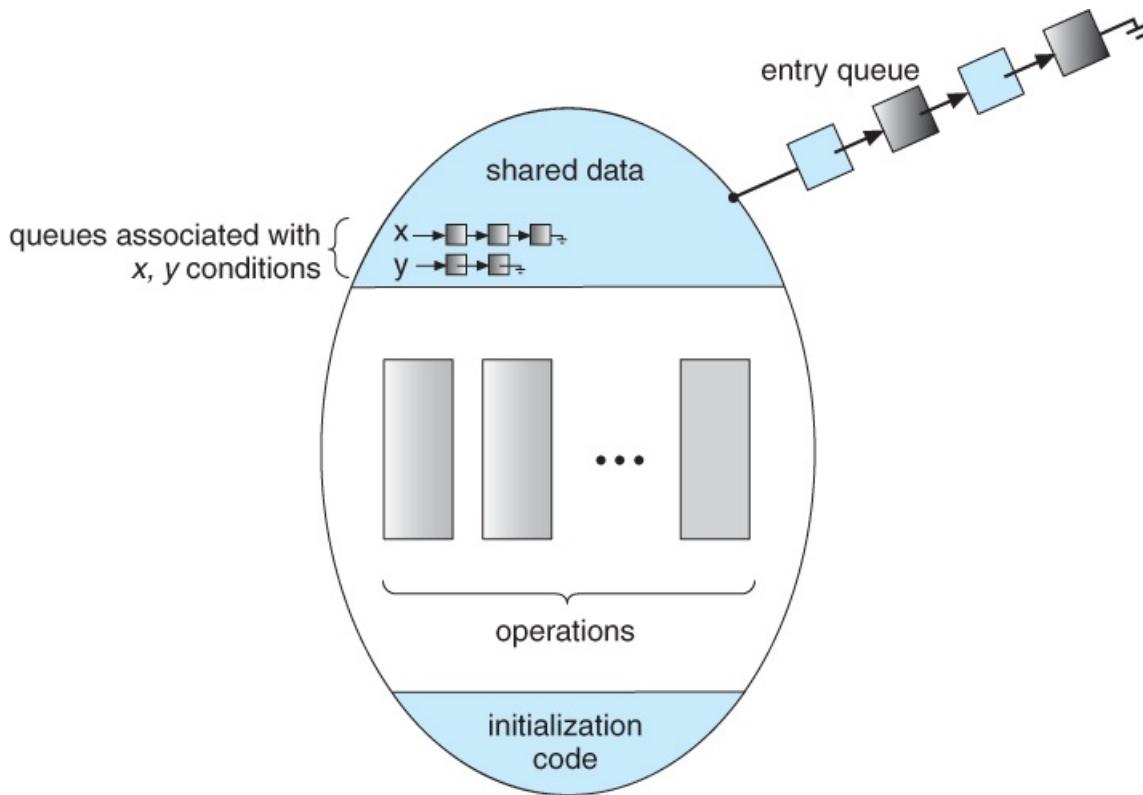
Figure 5.15 - Syntax of a monitor.

- Figure 5.16 shows a schematic of a monitor, with an entry queue of processes waiting their turn to execute monitor operations (methods.)

Figure 5.16 - Schematic view of a monitor

- In order to fully realize the potential of monitors, we need to introduce one additional new data type, known as a **condition**.
  - A variable of type condition has only two legal operations, **wait** and **signal**. I.e. if X was defined as type condition, then legal operations would be X.wait( ) and X.signal( )
  - The wait operation blocks a process until some other process calls signal, and adds the blocked process onto a list associated with that condition.
  - The signal process does nothing if there are no processes waiting on that condition. Otherwise it wakes up exactly one process from the condition's list of waiting processes. (Contrast this with counting semaphores, which always affect the semaphore on a signal call.)
- Figure 6.18 below illustrates a monitor that includes condition variables within its data space. Note that the condition variables, along with the list of processes currently waiting for the conditions, are in the data space of the monitor - The processes on these lists are not "in" the monitor, in the sense that they are not executing any code in the monitor.

Figure 5.17 - Monitor with condition variables

- But now there is a potential problem - If process P within the monitor issues a signal that would wake up process Q also within the monitor, then there would be two processes running simultaneously within the monitor, violating the exclusion requirement. Accordingly there are two possible solutions to this dilemma:

  Signal and wait - When process P issues the signal to wake up process Q, P then waits, either for Q to leave the monitor or on some other condition.

  Signal and continue - When P issues the signal, Q waits, either for P to exit the monitor or for some other condition.

  There are arguments for and against either choice. Concurrent Pascal offers a third alternative - The signal call causes the signaling process to immediately exit the monitor, so that the waiting process can then wake up and proceed.

- Java and C# (C sharp) offer monitors bulit-in to the language. Erlang offers similar but different constructs.

5.8.2 Dining-Philosophers Solution Using Monitors

- This solution to the dining philosophers uses monitors, and the restriction that a philosopher may only pick up chopsticks when both are available. There are also two key data structures in use in this solution:
  1. `enum { THINKING, HUNGRY,EATING } state[ 5 ];` A philosopher may only set their state to eating when neither of their adjacent neighbors is eating. (state[(i + 1) % 5] != EATING &&

state[(i + 4) % 5] != EATING).
   2. `condition self[ 5 ];` This condition is used to delay a hungry philosopher who is unable to acquire chopsticks.
- In the following solution philosophers share a monitor, DiningPhilosophers, and eat using the following sequence of operations:
   1. DiningPhilosophers.pickup( ) - Acquires chopsticks, which may block the process.
   2. eat
   3. DiningPhilosophers.putdown( ) - Releases the chopsticks.

```
monitor DiningPhilosophers
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING)) {
           state[i] = EATING;
           self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
   }
}
```

Figure 5.18   A monitor solution to the dining-philosopher problem.

5.8.3 Implementing a Monitor Using Semaphores

- One possible implementation of a monitor uses a semaphore "mutex" to control mutual exclusionary access to the monitor, and a counting semaphore "next" on which processes can suspend themselves after they are already "inside" the monitor (in conjunction with condition variables, see below.) The integer next_count keeps track of how many processes are waiting in the next queue. Externally accessible monitor processes are then implemented as:

```
wait(mutex);

    ...
    body of F
    ...
if (next_count > 0)
    signal(next);
else
    signal(mutex);
```

- Condition variables can be implemented using semaphores as well. For a condition x, a semaphore "x_sem" and an integer "x_count" are introduced, both initialized to zero. The wait and signal methods are then implemented as follows. (This approach to the condition implements the signal-and-wait option described above for ensuring that only one process at a time is active inside the monitor.)

Wait:
```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

Signal:
```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

5.8.4 Resuming Processes Within a Monitor

- When there are multiple processes waiting on the same condition within a monitor, how does one decide which one to wake up in response to a signal on that condition? One obvious approach is FCFS, and this may be suitable in many cases.
- Another alternative is to assign (integer) priorities, and to wake up the process with the smallest (best) priority.
- Figure 5.19 illustrates the use of such a condition within a monitor used for resource allocation. Processes wishing to access this resource must specify the time they expect to use it using the acquire(time) method, and must call the release( ) method when they are done with the resource.

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;

    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }

    void release() {
        busy = FALSE;
        x.signal();
    }

    initialization_code() {
        busy = FALSE;
    }
}
```

Figure 5.19 - A monitor to allocate a single resource.

- Unfortunately the use of monitors to restrict access to resources still only works if programmers make the requisite acquire and release calls properly. One option would be to place the resource allocation code into the monitor, thereby eliminating the option for programmers to bypass or ignore the monitor, but then that would substitute the monitor's scheduling algorithms for whatever other scheduling algorithms may have been chosen for that particular resource. Chapter 14 on Protection presents more advanced methods for enforcing "nice" cooperation among processes contending for shared resources.
- Concurrent Pascal, Mesa, C#, and Java all implement monitors as described here. Erlang provides concurrency support using a similar mechanism.

5.9 Synchronization Examples (Optional)

This section looks at how synchronization is handled in a number of different systems.

5.9.1 Synchronization in Windows

Figure 5.20 - Mutex dispatcher object

5.9.2 Synchronization in Linux

single processor	multiple processors
Disable kernel preemption.	Acquire spin lock.
Enable kernel preemption.	Release spin lock.

5.9.3 Synchronization in Solaris

- Solaris controls access to critical sections using five tools: semaphores, condition variables, adaptive mutexes, reader-writer locks, and turnstiles. The first two are as described above, and the other three are described here:

Adaptive Mutexes

- Adaptive mutexes are basically binary semaphores that are implemented differently depending upon the conditions:
  - On a single processor system, the semaphore sleeps when it is blocked, until the block is released.
  - On a multi-processor system, if the thread that is blocking the semaphore is running on the same processor as the thread that is blocked, or if the blocking thread is not running at all, then the blocked thread sleeps just like a single processor system.
  - However if the blocking thread is currently running on a different processor than the blocked thread, then the blocked thread does a spinlock, under the assumption that the block will soon be released.
  - Adaptive mutexes are only used for protecting short critical sections, where the benefit of not doing context switching is worth a short bit of spinlocking. Otherwise traditional semaphores and condition variables are used.

Reader-Writer Locks

- Reader-writer locks are used only for protecting longer sections of code which are accessed frequently but which are changed infrequently.

Turnstiles

- A *turnstile* is a queue of threads waiting on a lock.
- Each synchronized object which has threads blocked waiting for access to it needs a separate turnstile. For efficiency, however, the turnstile is associated with the thread currently holding the object, rather than the object itself.
- In order to prevent *priority inversion,* the thread holding a lock for an object will temporarily acquire the highest priority of any process in the turnstile waiting for the blocked object. This is called a *priority-inheritance protocol.*
- User threads are controlled the same as for kernel threads, except that the priority-inheritance protocol does not apply.

5.9 4 Pthreads Synchronization(was 6.8.4)

5.10 Alternate Approaches (Optional)

5.10.1 Transactional Memory

5.10.2 OpenMP

5.10.3 Functional Programming Languages

5.11 Summary

Old 6.9 Atomic Transactions (Optional, Not in Ninth Edition)

- Database operations frequently need to carry out *atomic transactions,* in which the entire transaction must either complete or not occur at all. The classic example is a transfer of funds, which involves withdrawing funds from one account and depositing them into another - Either both halves of the transaction must complete, or neither must complete.
- Operating Systems can be viewed as having many of the same needs and problems as databases, in that an OS can be said to manage a small database of process-related information. As such, OSes can benefit from emulating some of the techniques originally developed for databases. Here we first look at some of those techniques, and then how they can be used to benefit OS development.

6.9.1 System Model

- A transaction is a series of actions that must either complete in its entirety or must be *rolled-back* as if it had

never commenced.
- The system is considered to have three types of storage:
  - Volatile storage usually gets lost in a system crash.
  - Non-volatile storage usually survives system crashes, but may still get lost.
  - Stable storage "never" gets lost or damaged. In practice this is implemented using multiple copies stored on different media with different failure modes.

6.9.2 Log-Based Recovery

- **Before** each step of a transaction is conducted, a entry is written to a log on stable storage:
  - Each transaction has a unique serial number.
  - The first entry is a "start"
  - Every data changing entry specifies the transaction number, the old value, and the new value.
  - The final entry is a "commit".
  - All transactions are idempotent - The can be repeated any number of times and the effect is the same as doing them once. Likewise they can be undone any number of times and the effect is the same as undoing them once. (I.e. "change x from 5 to 6", rather than "add 1 to x").
- After a crash, any transaction which has "commit" recorded in the log can be redone from the log information. Any which has "start" but not "commit" can be undone.

6.9.3 Checkpoints

- In the event of a crash, all data can be recovered using the system described above, by going through the entire log and performing either redo or undo operations on all the transactions listed there.
- Unfortunately this approach can be slow and wasteful, because many transactions are repeated that were not lost in the crash.
- Alternatively, one can periodically establish a **checkpoint,** as follows:
  - Write all data that has been affected by recent transactions (since the last checkpoint) to stable storage.
  - Write a <checkpoint> entry to the log.
- Now for crash recovery one only needs to find transactions that did not commit prior to the most recent checkpoint. Specifically one looks backwards from the end of the log for the last <checkpoint> record, and then looks backward from there for the most recent transaction that started before the checkpoint. Only that transaction and the ones more recent need to be redone or undone.

6.9.4 Concurrent Atomic Transactions

- All of the previous discussion on log-based recovery assumed that only one transaction could be conducted at a time. We now relax that restriction, and allow multiple transactions to occur concurrently, while still keeping each individual transaction atomic.

6.9.4.1 Serializability

- Figure 6.22 below shows a **schedule** in which transaction 0 reads and writes data items A and B, followed by transaction 1 which also reads and writes A and B.
- This is termed a **serial schedule**, because the two transactions are conducted serially. For any N transactions, there are N! possible serial schedules.
- A **nonserial schedule** is one in which the steps of the transactions are not completely serial, i.e. they interleave in some manner.
- Nonserial schedules are not necessarily bad or wrong, so long as they can be shown to be equivalent to some serial schedule. A nonserial schedule that can be converted to a serial one is said to be **conflict serializable,** such as that shown in Figure 6.23 below. Legal steps in the conversion are as follows:
  - Two operations from different transactions are said to be **conflicting** if they involve the same data item and at least one of the operations is a write operations. Operations from two transactions are **non-conflicting** if they either involve different data items or do not involve any write operations.
  - Two operations in a schedule can be swapped if they are from two different transactions and if they are non-conflicting.
  - A schedules is conflict serializable if there exists a series of valid swap that converts the schedule into a serial schedule.

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Figure 6.22 Schedule 1: A serial schedule in which T_0 is followed by T_1.

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Figure 6.23 Schedule 2: A concurrent serializable schedule.

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

Figure 6.24 Schedule 3: A schedule possible under the timestamp protocol.

6.9.4.2 Locking Protocol

- One way to ensure serializability is to use locks on data items during atomic transactions.
- Shared and Exclusive locks correspond to the Readers and Writers problem discussed above.
- The **two-phase locking protocol** operates in two phases:
    - A **growing phase,** in which the transaction continues to gain additional locks on data items as it needs them, and has not yet relinquished any of its locks.
    - A **shrinking phase,** in which it relinquishes locks. Once a transaction releases any lock, then it is in the shrinking phase and cannot acquire any more locks.
- The two-phase locking protocol can be proven to yield serializable schedules, but it does not guarantee avoidance of deadlock. There may also be perfectly valid serializable schedules that

are unobtainable with this protocol.

6.9.4.3 Timestamp-Based Protocols

- Under the timestamp protocol, each **transaction** is issued a unique timestamp entry before it begins execution. This can be the system time on systems where all process access the same clock, or some non-decreasing serial number. The timestamp for transaction Ti is denoted TS(Ti)
- The schedules generated are all equivalent to a serial schedule occurring in timestamp order.
- Each **data item** also has two timestamp values associated with it - The W-timestamp is the timestamp of the last transaction to successfully write the data item, and the R-timestamp is the stamp of the last transaction to successfully read from it. (Note that these are the timestamps of the respective transactions, not the time at which the read or write occurred.)
- The timestamps are used in the following manner:
    - Suppose transaction Ti issues a read on data item Q:
        - If TS(Ti) < the W-timestamp for Q, then it is attempting to read data that has been changed. Transaction Ti is rolled back, and restarted with a new timestamp.
        - If TS(Ti) > the W-timestamp for Q, then the R-timestamp for Q is updated to the later of its current value and TS(Ti).
    - Suppose Ti issues a write on Q:
        - If TS(Ti) < the R-timestamp for Q, then it is attempting to change data that has already been read in its unaltered state. Ti is rolled back and restarted with a new timestamp.
        - If TS(Ti) < the W-timestamp for Q it is also rolled back and restarted, for similar reasons.
        - Otherwise, the operation proceeds, and the W-timestamp for Q is updated to TS(Ti).

CPU Scheduling

References:

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 6

6.1 Basic Concepts

- Almost all programs have some alternating cycle of CPU number crunching and waiting for I/O of some kind. (Even a simple fetch from memory takes a long time relative to CPU speeds.)
- In a simple system running a single process, the time spent waiting for I/O is wasted, and those CPU cycles are lost forever.
- A scheduling system allows one process to use the CPU while another is waiting for I/O, thereby making full use of otherwise lost CPU cycles.
- The challenge is to make the overall system as "efficient" and "fair" as possible, subject to varying and often dynamic conditions, and where "efficient" and "fair" are somewhat subjective terms, often subject to shifting priority policies.

6.1.1 CPU-I/O Burst Cycle

- Almost all processes alternate between two states in a continuing **cycle**, as shown in Figure 6.1 below **:**
  - A CPU burst of performing calculations, and
  - An I/O burst, waiting for data transfer in or out of the system.
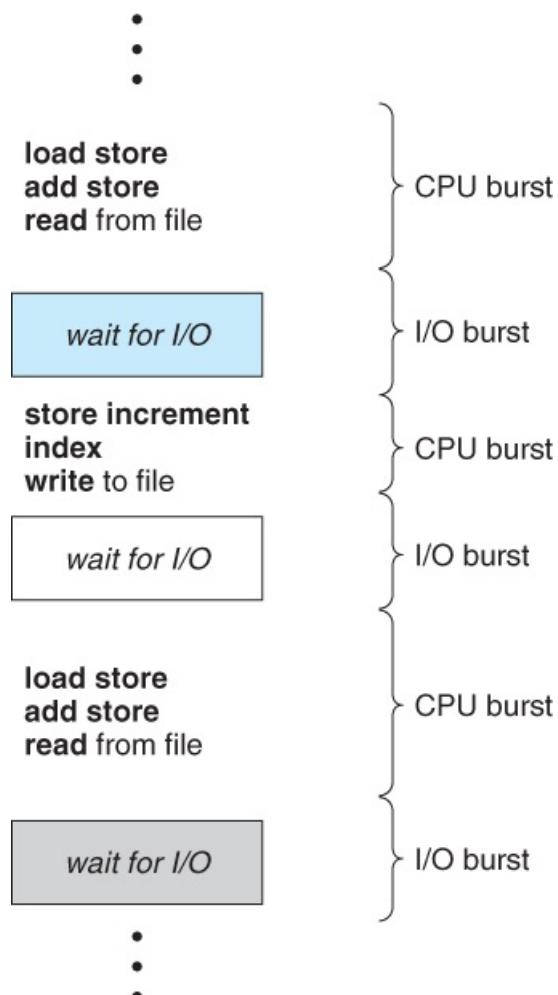
Figure 6.1 - Alternating sequence of CPU and I/O bursts.

- CPU bursts vary from process to process, and from program to program, but an extensive study shows frequency patterns similar to that shown in Figure 6.2:
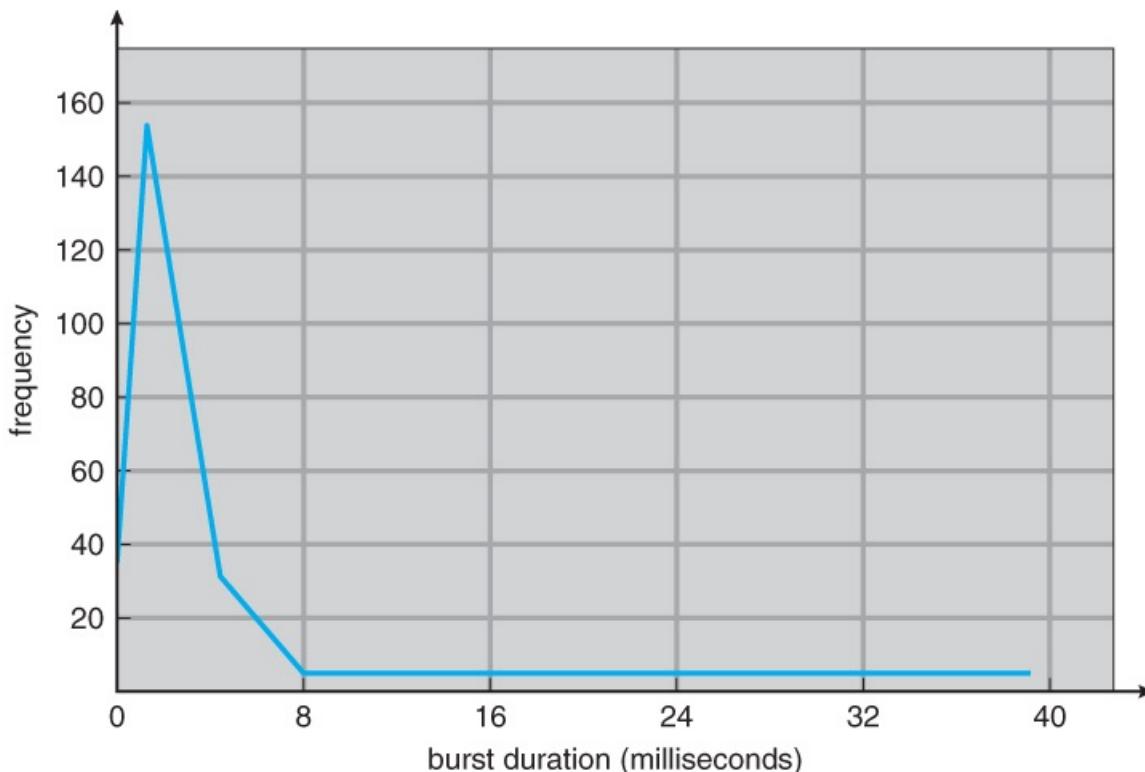
Figure 6.2 - Histogram of CPU-burst durations.

6.1.2 CPU Scheduler

- Whenever the CPU becomes idle, it is the job of the CPU Scheduler (a.k.a. the short-term scheduler) to select another process from the ready queue to run next.
- The storage structure for the ready queue and the algorithm used to select the next process are not necessarily a FIFO queue. There are several alternatives to choose from, as well as numerous adjustable parameters for each algorithm, which is the basic subject of this entire chapter.

6.1.3. Preemptive Scheduling

- CPU scheduling decisions take place under one of four conditions:
    1. When a process switches from the running state to the waiting state, such as for an I/O request or invocation of the wait( ) system call.
    2. When a process switches from the running state to the ready state, for example in response to an interrupt.
    3. When a process switches from the waiting state to the ready state, say at completion of I/O or a return from wait( ).
    4. When a process terminates.
- For conditions 1 and 4 there is no choice - A new process must be selected.
- For conditions 2 and 3 there is a choice - To either continue running the current process, or select a different one.
- If scheduling takes place only under conditions 1 and 4, the system is said to be **non-preemptive**, or **cooperative**. Under these conditions, once a process starts running it keeps running, until it either voluntarily blocks or until it finishes. Otherwise the system is said to be **preemptive.**
- Windows used non-preemptive scheduling up to Windows 3.x, and started using pre-emptive scheduling with Win95. Macs used non-preemptive prior to OSX, and pre-emptive since then. Note that pre-emptive scheduling is only possible on hardware that supports a timer interrupt.
- Note that pre-emptive scheduling can cause problems when two processes share data, because one process may get interrupted in the middle of updating shared data structures. Chapter 5 examined this issue in greater detail.
- Preemption can also be a problem if the kernel is busy implementing a system call (e.g. updating critical kernel data structures) when the preemption occurs. Most modern UNIXes deal with this problem by making the process wait until the system call has either completed or blocked before allowing the preemption Unfortunately this solution is problematic for real-time systems, as real-time response can no longer be guaranteed.
- Some critical sections of code protect themselves from con currency problems by disabling interrupts before entering the critical section and re-enabling interrupts on exiting the section. Needless to say, this should

only be done in rare situations, and only on very short pieces of code that will finish quickly, (usually just a few machine instructions.)

6.1.4 Dispatcher

- The **dispatcher** is the module that gives control of the CPU to the process selected by the scheduler. This function involves:
    - Switching context.
    - Switching to user mode.
    - Jumping to the proper location in the newly loaded program.
- The dispatcher needs to be as fast as possible, as it is run on every context switch. The time consumed by the dispatcher is known as **dispatch latency.**

6.2 Scheduling Criteria

- There are several different criteria to consider when trying to select the "best" scheduling algorithm for a particular situation and environment, including:
    - **CPU utilization** - Ideally the CPU would be busy 100% of the time, so as to waste 0 CPU cycles. On a real system CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)
    - **Throughput** - Number of processes completed per unit time. May range from 10 / second to 1 / hour depending on the specific processes.
    - **Turnaround time** - Time required for a particular process to complete, from submission time to completion. (Wall clock time.)
    - **Waiting time** - How much time processes spend in the ready queue waiting their turn to get on the CPU.
        - (**Load average** - The average number of processes sitting in the ready queue waiting their turn to get into the CPU. Reported in 1-minute, 5-minute, and 15-minute averages by "uptime" and "who".)
    - **Response time** - The time taken in an interactive program from the issuance of a command to the **commence** of a response to that command.
- In general one wants to optimize the average value of a criteria (Maximize CPU utilization and throughput, and minimize all the others.) However some times one wants to do something different, such as to minimize the maximum response time.
- Sometimes it is most desirable to minimize the *variance* of a criteria than the actual value. I.e. users are more accepting of a consistent predictable system than an inconsistent one, even if it is a little bit slower.

6.3 Scheduling Algorithms
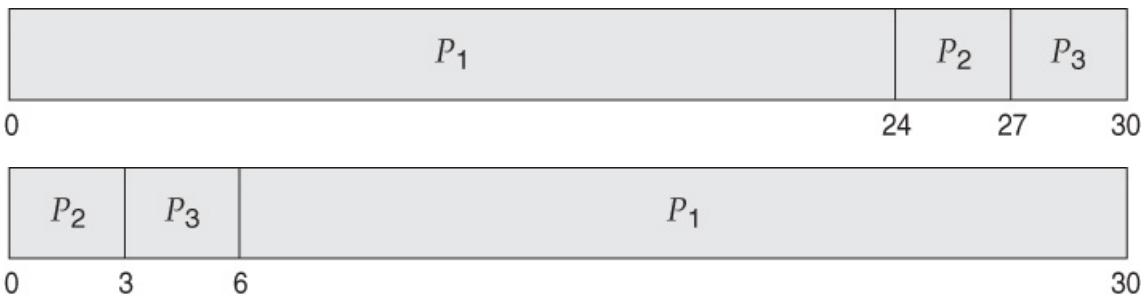
The following subsections will explain several common scheduling strategies, looking at only a single CPU burst each for a small number of processes. Obviously real systems have to deal with a lot more simultaneous processes executing their CPU-I/O burst cycles.

6.3.1 First-Come First-Serve Scheduling, FCFS

- FCFS is very simple - Just a FIFO queue, like customers waiting in line at the bank or the post office or at a copying machine.
- Unfortunately, however, FCFS can yield some very long average wait times, particularly if the first process to get there takes a long time. For example, consider the following three processes:

Process	Burst Time
P1	24
P2	3
P3	3

- In the first Gantt chart below, process P1 arrives first. The average waiting time for the three processes is (0 + 24 + 27) / 3 = 17.0 ms.
- In the second Gantt chart below, the same three processes have an average wait time of (0 + 3 + 6) / 3 = 3.0 ms. The total run time for the three bursts is the same, but in the second case two of the three finish much quicker, and the other process is only delayed by a short amount.
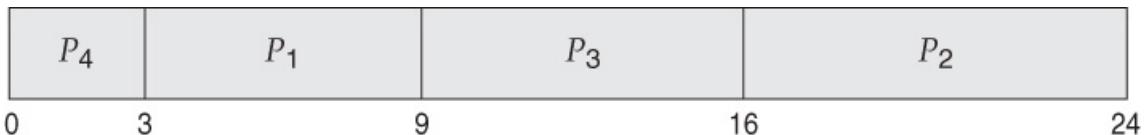
- FCFS can also block the system in a busy dynamic system in another way, known as the ***convoy effect***. When one CPU intensive process blocks the CPU, a number of I/O intensive processes can get backed up behind it, leaving the I/O devices idle. When the CPU hog finally relinquishes the CPU, then the I/O processes pass through the CPU quickly, leaving the CPU idle while everyone queues up for I/O, and then the cycle repeats itself when the CPU intensive process gets back to the ready queue.

6.3.2 Shortest-Job-First Scheduling, SJF

- The idea behind the SJF algorithm is to pick the quickest fastest little job that needs to be done, get it out of the way first, and then pick the next smallest fastest job to do next.
- (Technically this algorithm picks a process based on the next shortest **CPU burst**, not the overall process time.)
- For example, the Gantt chart below is based upon the following CPU burst times, (and the assumption that all jobs arrive at the same time.)
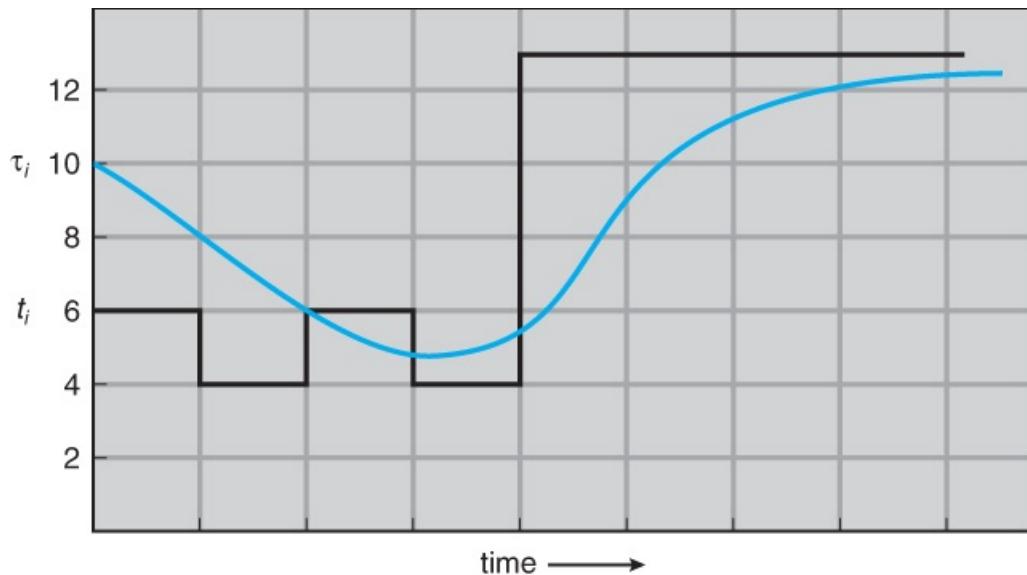
Process	Burst Time
P1	6
P2	8
P3	7
P4	3

- In the case above the average wait time is (0 + 3 + 9 + 16) / 4 = 7.0 ms, (as opposed to 10.25 ms for FCFS for the same processes.)
- SJF can be proven to be the fastest scheduling algorithm, but it suffers from one important problem: How do you know how long the next CPU burst is going to be?
  - For long-term batch jobs this can be done based upon the limits that users set for their jobs when they submit them, which encourages them to set low limits, but risks their having to re-submit the job if they set the limit too low. However that does not work for short-term CPU scheduling on an interactive system.
  - Another option would be to statistically measure the run time characteristics of jobs, particularly if the same tasks are run repeatedly and predictably. But once again that really isn't a viable option for short term CPU scheduling in the real world.
  - A more practical approach is to ***predict*** the length of the next burst, based on some historical measurement of recent burst times for this process. One simple, fast, and relatively accurate method is the ***exponential average***, which can be defined as follows. (The book uses tau and t for their variables, but those are hard to distinguish from one another and don't work well in HTML.)

    estimate[i + 1] = alpha * burst[i] + (1.0 - alpha) * estimate[i]
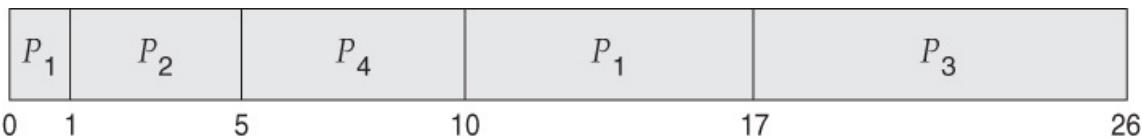
  - In this scheme the previous estimate contains the history of all previous times, and alpha serves as a weighting factor for the relative importance of recent data versus past history. If alpha is 1.0, then past history is ignored, and we assume the next burst will be the same length as the last burst. If alpha is 0.0, then all measured burst times are ignored, and we just assume a constant burst time. Most commonly alpha is set at 0.5, as illustrated in Figure 5.3:

CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Figure 6.3 - Prediction of the length of the next CPU burst.

- SJF can be either preemptive or non-preemptive. Preemption occurs when a new process arrives in the ready queue that has a predicted burst time shorter than the time remaining in the process whose burst is currently on the CPU. Preemptive SJF is sometimes referred to as ***shortest remaining time first scheduling.***
- For example, the following Gantt chart is based upon the following data:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
p4	3	5
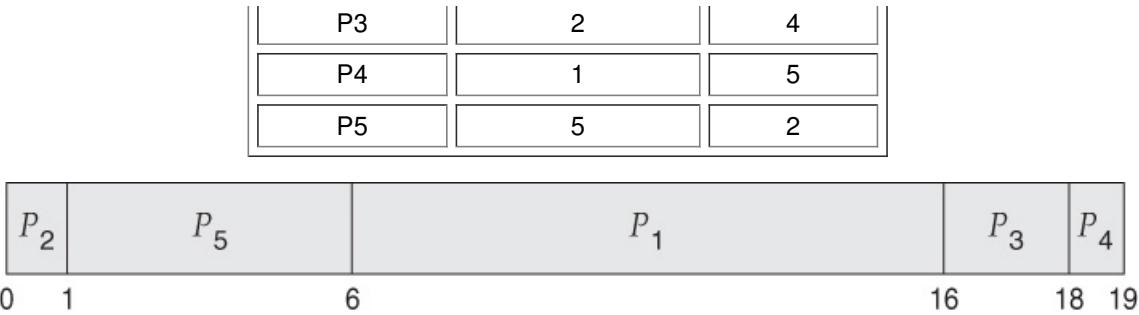
- The average wait time in this case is ((5 - 3) + (10 - 1) + (17 - 2)) / 4 = 26 / 4 = 6.5 ms. (As opposed to 7.75 ms for non-preemptive SJF or 8.75 for FCFS.)

6.3.3 Priority Scheduling

- Priority scheduling is a more general case of SJF, in which each job is assigned a priority and the job with the highest priority gets scheduled first. (SJF uses the inverse of the next expected burst time as its priority - The smaller the expected burst, the higher the priority.)
- Note that in practice, priorities are implemented using integers within a fixed range, but there is no agreed-upon convention as to whether "high" priorities use large numbers or small numbers. This book uses low number for high priorities, with 0 being the highest possible priority.
- For example, the following Gantt chart is based upon these process burst times and priorities, and yields an average waiting time of 8.2 ms:
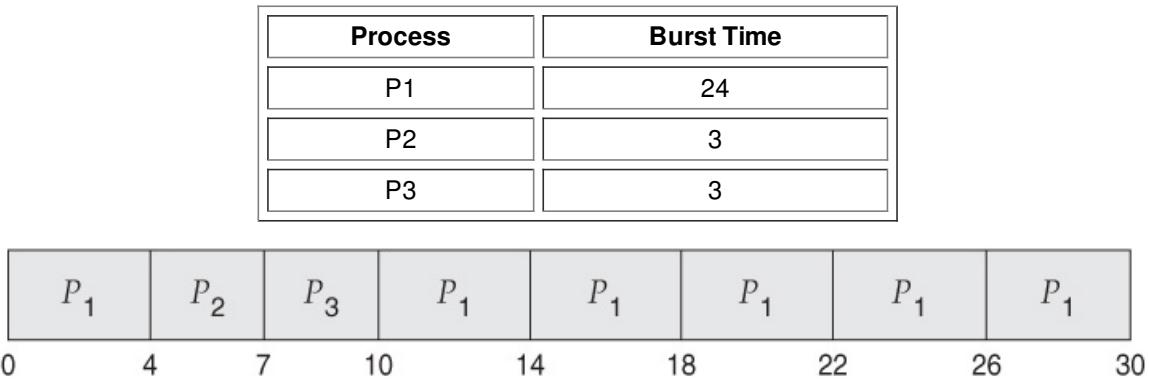
Process	Burst Time	Priority
P1	10	3
P2	1	1

P3	2	4
P4	1	5
P5	5	2

P_2	P_5	P_1	P_3	P_4

```
0   1          6                    16      18  19
```
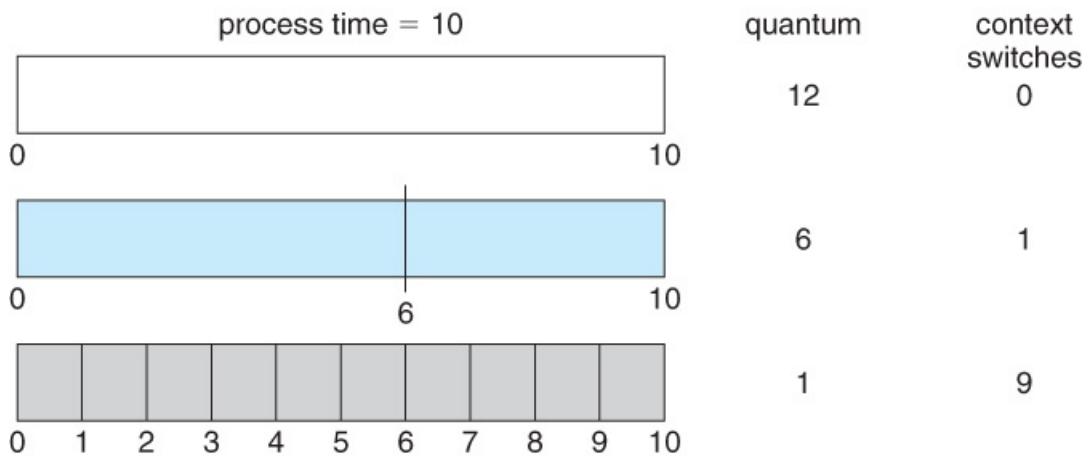
- Priorities can be assigned either internally or externally. Internal priorities are assigned by the OS using criteria such as average burst time, ratio of CPU to I/O activity, system resource use, and other factors available to the kernel. External priorities are assigned by users, based on the importance of the job, fees paid, politics, etc.
- Priority scheduling can be either preemptive or non-preemptive.
- Priority scheduling can suffer from a major problem known as **indefinite blocking**, or **starvation**, in which a low-priority task can wait forever because there are always some other jobs around that have higher priority.
  - If this problem is allowed to occur, then processes will either run eventually when the system load lightens (at say 2:00 a.m.), or will eventually get lost when the system is shut down or crashes. (There are rumors of jobs that have been stuck for years.)
  - One common solution to this problem is **aging**, in which priorities of jobs increase the longer they wait. Under this scheme a low-priority job will eventually get its priority raised high enough that it gets run.

6.3.4 Round Robin Scheduling

- Round robin scheduling is similar to FCFS scheduling, except that CPU bursts are assigned with limits called **time quantum**.
- When a process is given the CPU, a timer is set for whatever value has been set for a time quantum.
  - If the process finishes its burst before the time quantum timer expires, then it is swapped out of the CPU just like the normal FCFS algorithm.
  - If the timer goes off first, then the process is swapped out of the CPU and moved to the back end of the ready queue.
- The ready queue is maintained as a circular queue, so when all processes have had a turn, then the scheduler gives the first process another turn, and so on.
- RR scheduling can give the effect of all processors sharing the CPU equally, although the average wait time can be longer than with other scheduling algorithms. In the following example the average wait time is 5.66 ms.

Process	Burst Time
P1	24
P2	3
P3	3

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

```
0     4     7     10     14     18     22     26     30
```
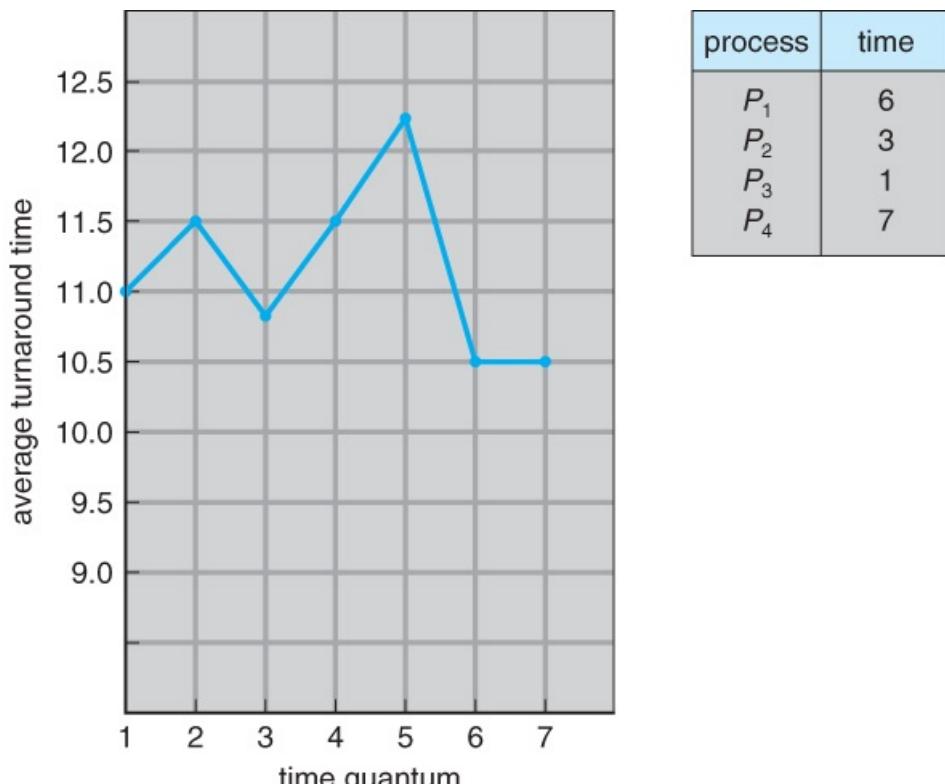
- The performance of RR is sensitive to the time quantum selected. If the quantum is large enough, then RR reduces to the FCFS algorithm; If it is very small, then each process gets 1/nth of the processor time and share the CPU equally.
- **BUT**, a real system invokes overhead for every context switch, and the smaller the time quantum the more context switches there are. (See Figure 6.4 below.) Most modern systems use time quantum between 10 and 100 milliseconds, and context switch times on the order of 10 microseconds, so the overhead is small relative to the time quantum.

Figure 6.4 - The way in which a smaller time quantum increases context switches.

- Turn around time also varies with quantum time, in a non-apparent manner. Consider, for example the processes shown in Figure 6.5:

process	time
P_1	6
P_2	3
P_3	1
P_4	7
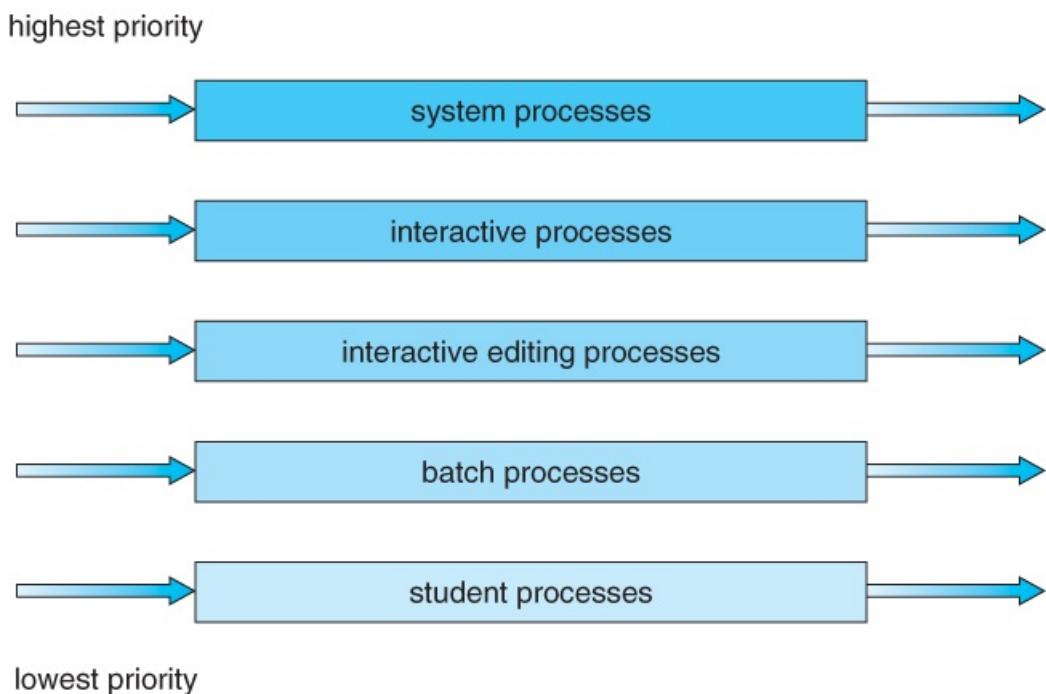
Figure 6.5 - The way in which turnaround time varies with the time quantum.

- In general, turnaround time is minimized if most processes finish their next cpu burst within one time quantum. For example, with three processes of 10 ms bursts each, the average turnaround time for 1 ms quantum is 29, and for 10 ms quantum it reduces to 20. However, if it is made too large, then RR just degenerates to FCFS. A rule of thumb is that 80% of CPU bursts should be smaller than the time quantum.

6.3.5 Multilevel Queue Scheduling
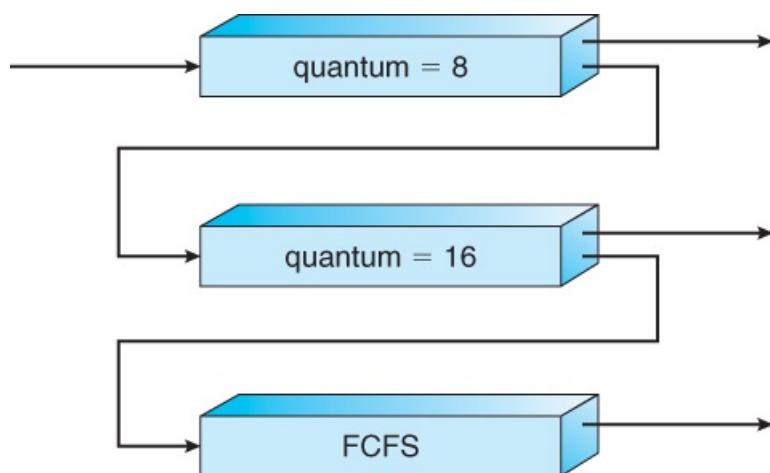
- When processes can be readily categorized, then multiple separate queues can be established, each implementing whatever scheduling algorithm is most appropriate for that type of job, and/or with different parametric adjustments.
- Scheduling must also be done between queues, that is scheduling one queue to get time relative to other queues. Two common options are strict priority (no job in a lower priority queue runs until all higher priority queues are empty) and round-robin (each queue gets a time slice in turn, possibly of different sizes.)
- Note that under this algorithm jobs cannot switch from queue to queue - Once they are assigned a queue, that is their queue until they finish.

Figure 6.6 - Multilevel queue scheduling

6.3.6 Multilevel Feedback-Queue Scheduling

- Multilevel feedback queue scheduling is similar to the ordinary multilevel queue scheduling described above, except jobs may be moved from one queue to another for a variety of reasons:
  - If the characteristics of a job change between CPU-intensive and I/O intensive, then it may be appropriate to switch a job from one queue to another.
  - Aging can also be incorporated, so that a job that has waited for a long time can get bumped up into a higher priority queue for a while.
- Multilevel feedback queue scheduling is the most flexible, because it can be tuned for any situation. But it is also the most complex to implement because of all the adjustable parameters. Some of the parameters which define one of these systems include:
  - The number of queues.
  - The scheduling algorithm for each queue.
  - The methods used to upgrade or demote processes from one queue to another. (Which may be different.)
  - The method used to determine which queue a process enters initially.

Figure 6.7 - Multilevel feedback queues.

6.4 Thread Scheduling

- The process scheduler schedules only the kernel threads.
- User threads are mapped to kernel threads by the thread library - The OS (and in particular the scheduler) is unaware of them.

6.4.1 Contention Scope

- **Contention scope** refers to the scope in which threads compete for the use of physical CPUs.
- On systems implementing many-to-one and many-to-many threads, **Process Contention Scope, PCS,** occurs, because competition occurs between threads that are part of the same process. (This is the management / scheduling of multiple user threads on a single kernel thread, and is managed by the thread library.)
- **System Contention Scope, SCS,** involves the system scheduler scheduling kernel threads to run on one or more CPUs. Systems implementing one-to-one threads (XP, Solaris 9, Linux), use only SCS.
- PCS scheduling is typically done with priority, where the programmer can set and/or change the priority of threads created by his or her programs. Even time slicing is not guaranteed among threads of equal priority.

6.4.2 Pthread Scheduling

- The Pthread library provides for specifying scope contention:
    - PTHREAD_SCOPE_PROCESS schedules threads using PCS, by scheduling user threads onto available LWPs using the many-to-many model.
    - PTHREAD_SCOPE_SYSTEM schedules threads using SCS, by binding user threads to particular LWPs, effectively implementing a one-to-one model.
- getscope and setscope methods provide for determining and setting the scope contention respectively:

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* get the default attributes */
    pthread_attr_init(&attr);

    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }

    /* set the scheduling algorithm to PCS or SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i],&attr,runner,NULL);

    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */

    pthread_exit(0);
}
```

Figure 6.8 - Pthread scheduling API.

6.5 Multiple-Processor Scheduling

- When multiple processors are available, then the scheduling gets more complicated, because now there is more than one CPU which must be kept busy and in effective use at all times.
- *Load sharing* revolves around balancing the load between multiple processors.
- Multi-processor systems may be **heterogeneous,** (different kinds of CPUs), or **homogenous,** (all the same kind of

CPU). Even in the latter case there may be special scheduling constraints, such as devices which are connected via a private bus to only one of the CPUs. This book will restrict its discussion to homogenous systems.

6.5.1 Approaches to Multiple-Processor Scheduling

- One approach to multi-processor scheduling is **asymmetric multiprocessing,** in which one processor is the master, controlling all activities and running all kernel code, while the other runs only user code. This approach is relatively simple, as there is no need to share critical system data.
- Another approach is **symmetric multiprocessing, SMP,** where each processor schedules its own jobs, either from a common ready queue or from separate ready queues for each processor.
- Virtually all modern OSes support SMP, including XP, Win 2000, Solaris, Linux, and Mac OSX.

6.5.2 Processor Affinity

- Processors contain cache memory, which speeds up repeated accesses to the same memory locations.
- If a process were to switch from one processor to another each time it got a time slice, the data in the cache (for that process) would have to be invalidated and re-loaded from main memory, thereby obviating the benefit of the cache.
- Therefore SMP systems attempt to keep processes on the same processor, via **processor affinity. Soft affinity** occurs when the system attempts to keep processes on the same processor but makes no guarantees. Linux and some other OSes support **hard affinity,** in which a process specifies that it is not to be moved between processors.
- Main memory architecture can also affect process affinity, if particular CPUs have faster access to memory on the same chip or board than to other memory loaded elsewhere. (Non-Uniform Memory Access, NUMA.) As shown below, if a process has an affinity for a particular CPU, then it should preferentially be assigned memory storage in "local" fast access areas.
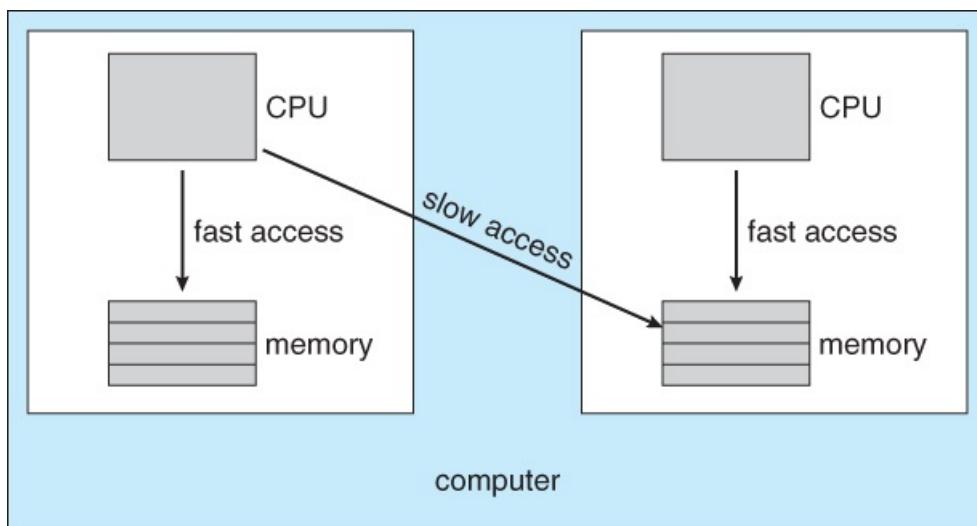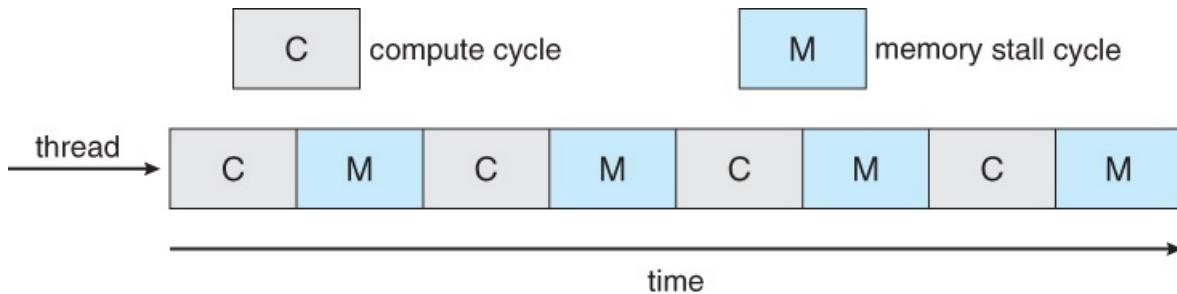
Figure 6.9 - NUMA and CPU scheduling.

6.5.3 Load Balancing

- Obviously an important goal in a multiprocessor system is to balance the load between processors, so that one processor won't be sitting idle while another is overloaded.
- Systems using a common ready queue are naturally self-balancing, and do not need any special handling. Most systems, however, maintain separate ready queues for each processor.
- Balancing can be achieved through either **push migration** or **pull migration:**
  - **Push migration** involves a separate process that runs periodically, (e.g. every 200 milliseconds), and moves processes from heavily loaded processors onto less loaded ones.
  - **Pull migration** involves idle processors taking processes from the ready queues of other processors.
  - Push and pull migration are not mutually exclusive.
- Note that moving processes from processor to processor to achieve load balancing works against the principle of processor affinity, and if not carefully managed, the savings gained by balancing the system can be lost in rebuilding caches. One option is to only allow migration when imbalance surpasses a given threshold.
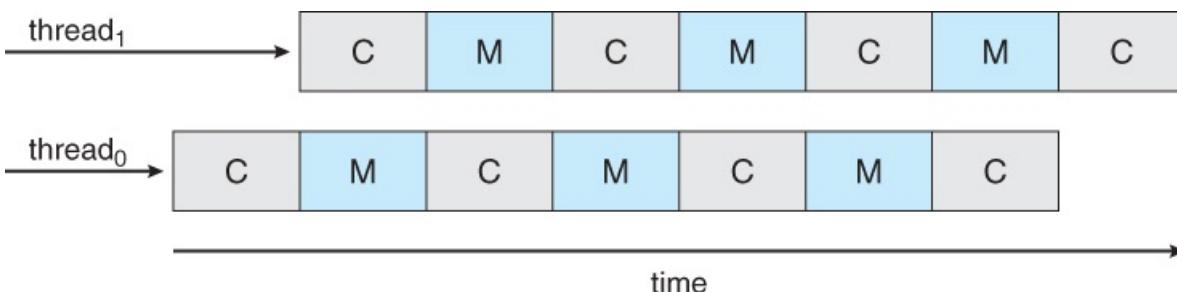
6.5.4 Multicore Processors

- Traditional SMP required multiple CPU chips to run multiple kernel threads concurrently.
- Recent trends are to put multiple CPUs (cores) onto a single chip, which appear to the system as multiple processors.
- Compute cycles can be blocked by the time needed to access memory, whenever the needed data is not already present in the cache. (Cache misses.) In Figure 5.10, as much as half of the CPU cycles are lost to memory stall.

Figure 6.10 - Memory stall.

- By assigning multiple kernel threads to a single processor, memory stall can be avoided (or reduced) by running one thread on the processor while the other thread waits for memory.

Figure 6.11 - Multithreaded multicore system.

- A dual-threaded dual-core system has four logical processors available to the operating system. The UltraSPARC T1 CPU has 8 cores per chip and 4 hardware threads per core, for a total of 32 logical processors per chip.
- There are two ways to multi-thread a processor:
    1. **Coarse-grained** multithreading switches between threads only when one thread blocks, say on a memory read. Context switching is similar to process switching, with considerable overhead.
    2. **Fine-grained** multithreading occurs on smaller regular intervals, say on the boundary of instruction cycles. However the architecture is designed to support thread switching, so the overhead is relatively minor.
- Note that for a multi-threaded multi-core system, there are **two** levels of scheduling, **at the kernel level:**
    - The OS schedules which kernel thread(s) to assign to which logical processors, and when to make context switches using algorithms as described above.
    - On a lower level, the hardware schedules logical processors on each physical core using some other algorithm.
        - The UltraSPARC T1 uses a simple round-robin method to schedule the 4 logical processors (kernel threads) on each physical core.
        - The Intel Itanium is a dual-core chip which uses a 7-level priority scheme (urgency) to determine which thread to schedule when one of 5 different events occurs.

/* **Old 5.5.5 Virtualization and Scheduling (Optional, Omitted from 9th edition)**

- Virtualization adds another layer of complexity and scheduling.
- Typically there is one host operating system operating on "real" processor(s) and a number of guest operating systems operating on virtual processors.
- The Host OS creates some number of virtual processors and presents them to the guest OSes as if they were real processors.
- The guest OSes don't realize their processors are virtual, and make scheduling decisions on the assumption of real processors.
- As a result, interactive and especially real-time performance can be severely compromised on guest systems. The time-of-day clock will also frequently be off.

*/

6.6 Real-Time CPU Scheduling

Real-time systems are those in which the time at which tasks complete is crucial to their performance

- **Soft real-time systems** have degraded performance if their timing needs cannot be met. Example: streaming video.
- **Hard real-time systems** have total failure if their timing needs cannot be met. Examples: Assembly line robotics, automobile air-bag deployment.

6.6.1 Minimizing Latency

- **Event Latency** is the time between the occurrence of a triggering event and the (completion of) the system's response to the event:

event E first occurs

event latency

t_0 t_1

real-time system responds to E
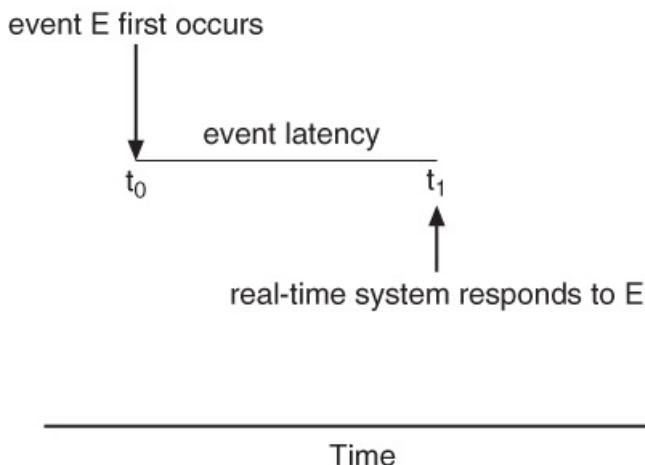
Time

Figure 6.12 - Event latency.

- In addition to the time it takes to actually process the event, there are two additional steps that must occur before the event handler (Interrupt Service Routine, ISR), can even start:
  - Interrupt processing determines which interrupt(s) have occurred, and which interrupt handler routine to run. In the event of simultaneous interrupts, a priority scheme is needed to determine which ISR has the highest priority and gets to go next.
  - Context switching involves removing a running process from the CPU, saving its state, and loading up the ISR so that it can run.

interrupt

task T running

determine interrupt type

context switch

ISR

interrupt latency

time
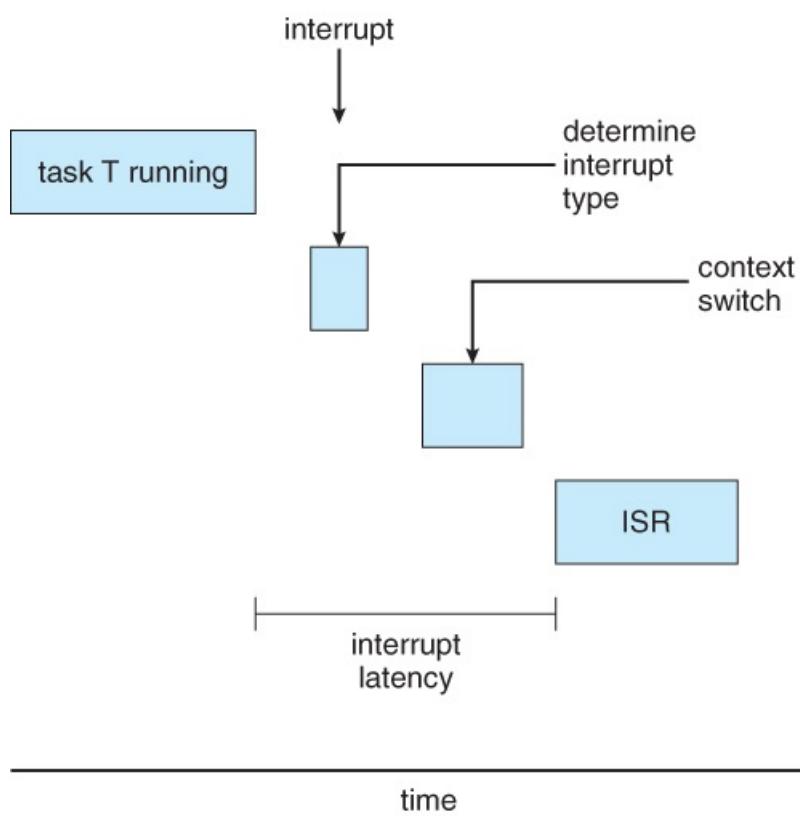
Figure 6.13 - Interrupt latency.

- The total dispatch latency (context switching) consists of two parts:
    - Removing the currently running process from the CPU, and freeing up any resources that are needed by the ISR. This step can be speeded up a lot by the use of pre-emptive kernels.
    - Loading the ISR up onto the CPU, (dispatching)
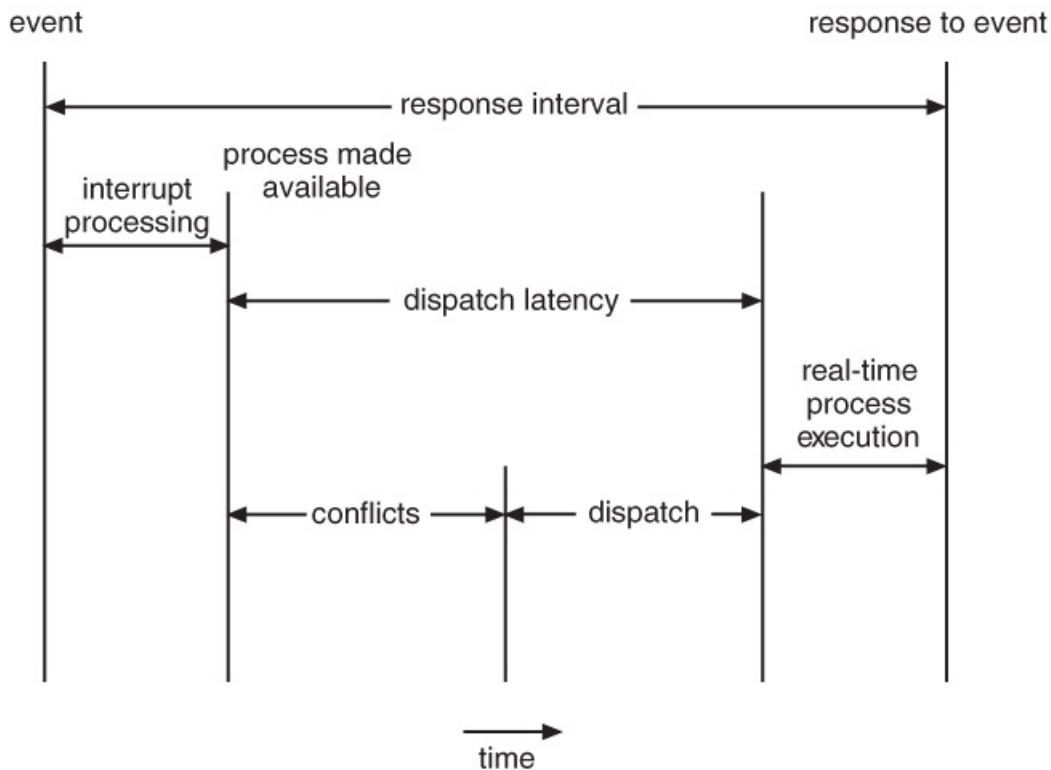
Figure 6.14 - Dispatch latency.

6.6.2 Priority-Based Scheduling

- Real-time systems require pre-emptive priority-based scheduling systems.
- Soft real-time systems do the best they can, but make no guarantees. They are generally discussed elsewhere.
- Hard real-time systems, described here, must be able to provide firm guarantees that a tasks scheduling needs can be met.
    - One technique is to use a ***admission-control*** algorithm, in which each task must specify its needs at the time it attempts to launch, and the system will only launch the task if it can guarantee that its needs can be met.
- Hard real-time systems are often characterized by tasks that must run at regular periodic intervals, each having a period p, a constant time required to execute, (CPU burst), t, and a deadline after the beginning of each period by which the task must be completed, d.
- In all cases, $t <= d <= p$
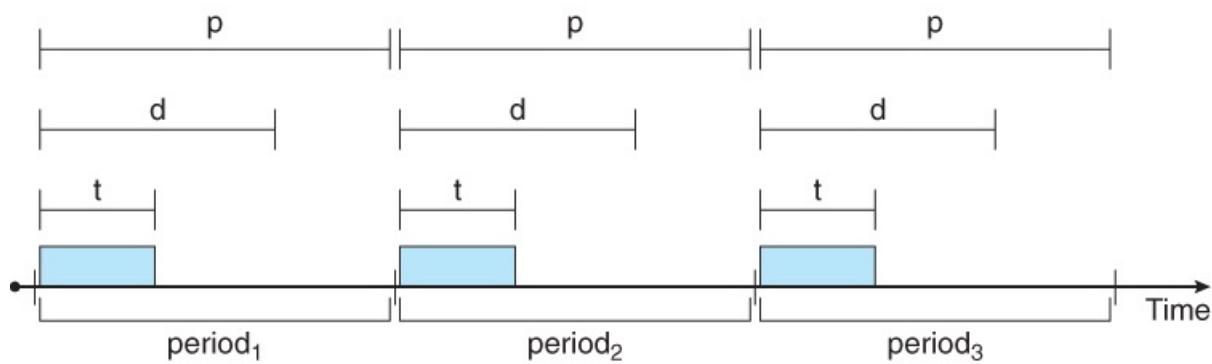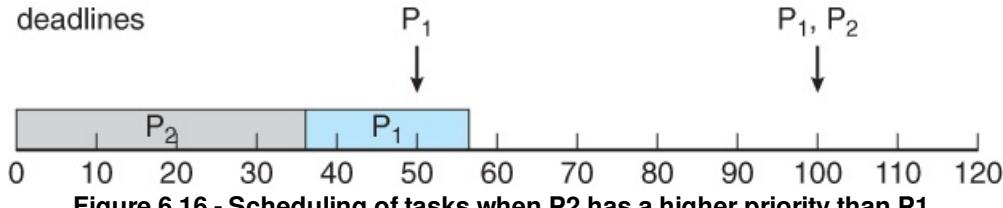
Figure 6.15 - Periodic task.

6.6.3 Rate-Monotonic Scheduling
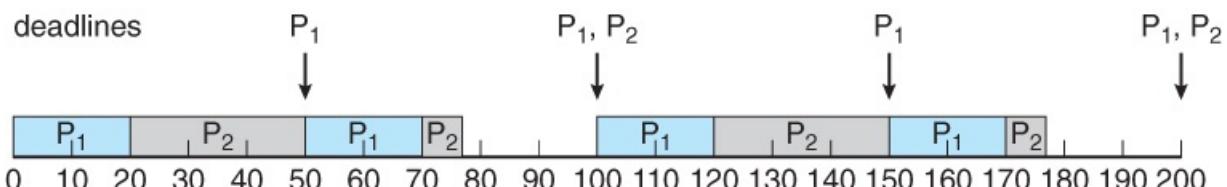
- The rate-monotonic scheduling algorithm uses pre-emptive scheduling with static priorities.

- Priorities are assigned inversely to the period of each task, giving higher (better) priority to tasks with shorter periods.
- Let's consider an example, first showing what can happen if the task with the longer period is given higher priority:
  - Suppose that process P1 has a period of 50, an execution time of 20, and a deadline that matches its period (50).
  - Similarly suppose that process P2 has period 100, execution time of 35, and deadline of 100.
  - The total CPU utilization time is 20 / 50 = 0.4 for P1, and 25 / 100 = 0.35 for P2, or 0.75 (75%) overall.
  - However if P2 is allowed to go first, then P1 cannot complete before its deadline:
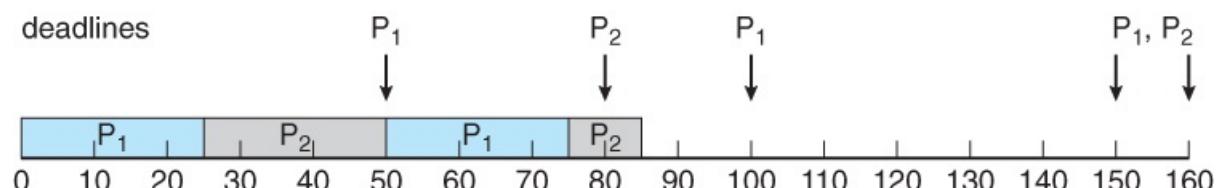
Figure 6.16 - Scheduling of tasks when P2 has a higher priority than P1.

  - Now on the other hand, if P1 is given higher priority, it gets to go first, and P2 starts after P1 completes its burst.
  - At time 50 when the next period for P1 starts, P2 has only completed 30 of its 35 needed time units, but it gets pre-empted by P1.
  - At time 70, P1 completes its task for its second period, and the P2 is allowed to complete its last 5 time units.
  - Overall both processes complete at time 75, and the cpu is then idle for 25 time units, before the process repeats.

Figure 6.17 - Rate-monotonic scheduling.

  - Rate-monotonic scheduling is considered optimal among algorithms that use static priorities, because any set of processes that cannot be scheduled with this algorithm cannot be scheduled with any other static-priority scheduling algorithm either.
  - There are, however, some sets of processes that cannot be scheduled with static priorities.
    - For example, supposing that P1 =50, T1 = 25, P2 = 80, T2 = 35, and the deadlines match the periods.
    - Overall CPU usage is 25/50 = 0.5 for P1, 35 / 80 =0.44 for P2, or 0.94 (94%) overall, indicating it should be possible to schedule the processes.
    - With rate-monotonic scheduling, P1 goes first, and completes its first burst at time 25.
    - P2 goes next, and completes 25 out of its 35 time units before it gets pre-empted by P1 at time 50.
    - P1 completes its second burst at 75, and then P2 completes its last 10 time units at time 85, missing its deadline of 80 by 5 time units. :-(

Figure 6.18 - Missing deadlines with rate-monotonic scheduling.

  - The worst-case CPU utilization for scheduling N processes under this algorithm is $N * (2^{(1/N)} - 1)$, which is 100% for a single process, but drops to 75% for two processes and to 69% as N approaches infinity. Note that in our example above 94% is higher than 75%.

6.6.4 Earliest-Deadline-First Scheduling

- Earliest Deadline First (EDF) scheduling varies priorities dynamically, giving the highest priority to the process with the earliest deadline.

- Figure 6.19 shows our previous example repeated under EDF scheduling:
  - At time 0 P1 has the earliest deadline, highest priority, and goes first., followed by P2 at time 25 when P1 completes its first burst.
  - At time 50 process P1 begins its second period, but since P2 has a deadline of 80 and the deadline for P1 is not until 100, P2 is allowed to stay on the CPU and complete its burst, which it does at time 60.
  - P1 then starts its second burst, which it completes at time 85. P2 started its second period at time 80, but since P1 had an earlier deadline, P2 did not pre-empt P1.
  - P2 starts its second burst at time 85, and continues until time 100, at which time P1 starts its third period.
  - At this point P1 has a deadline of 150 and P2 has a deadline of 160, so P1 preempts P2.
  - P1 completes its third burst at time 125, at which time P2 starts, completing its third burst at time 145.
  - The CPU sits idle for 5 time units, until P1 starts its next period at 150 and P2 at 160.
  - **Question:** Which process will get to run at time 160, and why?
    - **Answer:**
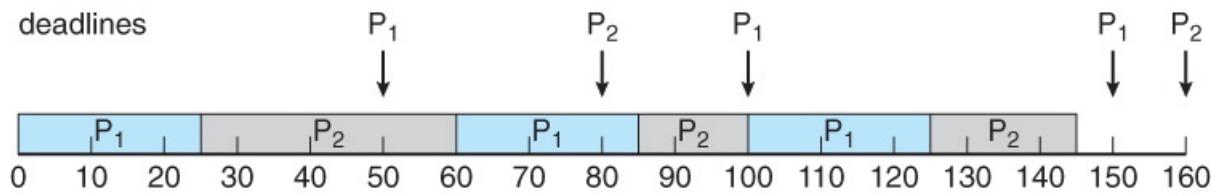
Figure 6.19 - Earliest-deadline-first scheduling.

6.6.5 Proportional Share Scheduling

- Proportional share scheduling works by dividing the total amount of time available up into an equal number of shares, and then each process must request a certain share of the total when it tries to start.
- Say for example that the total shares, T, = 100, and then a particular process asks for N = 30 shares when it launches. It will then be guaranteed 30 / 100 or 30% of the available time.
- Proportional share scheduling works with an admission-control policy, not starting any task if it cannot guarantee the shares that the task says that it needs.

6.6.6 POSIX Real-Time Scheduling

- POSIX defines two scheduling classes for real time threads, SCHED_FIFO and SCHED_RR, depending on how threads *of equal priority* share time.
- SCHED_FIFO schedules tasks in a first-in-first-out order, with no time slicing among threads of equal priority.
- SCHED_RR performs round-robin time slicing among threads of equal priority.
- POSIX provides methods for getting and setting the thread scheduling policy, as shown below:

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
    int i, policy;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* get the default attributes */
    pthread_attr_init(&attr);

    /* get the current scheduling policy */
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Unable to get policy.\n");
    else {
        if (policy == SCHED_OTHER)
            printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR)
            printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO)
            printf("SCHED_FIFO\n");
    }

    /* set the scheduling policy - FIFO, RR, or OTHER */
    if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
        fprintf(stderr, "Unable to set policy.\n");

    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i],&attr,runner,NULL);

    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */

    pthread_exit(0);
}
```

Figure 6.20 - POSIX real-time scheduling API.
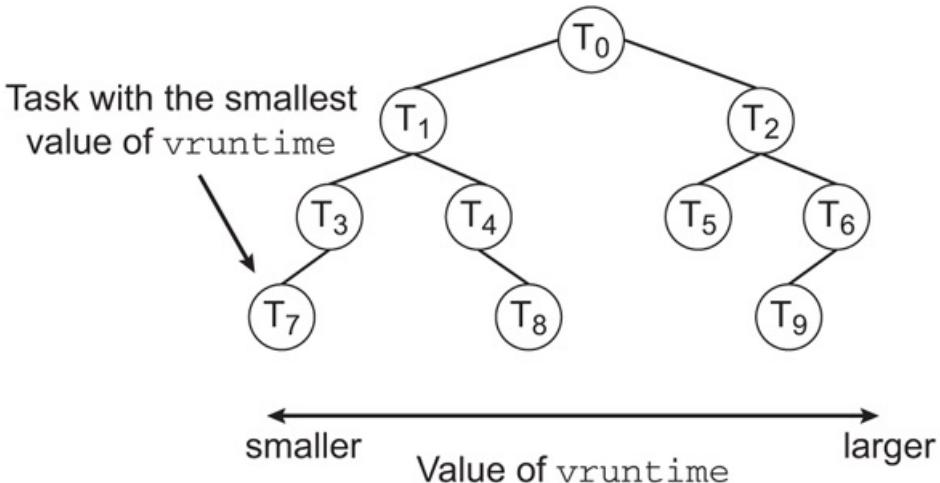
6.7 Operating System Examples (Optional)

6.7.1 Example: Linux Scheduling (was 5.6.3)

- Prior to version 2.5, Linux used a traditional UNIX scheduling algorithm.
- Version 2.6 used an algorithm known as O(1), that ran in constant time regardless of the number of tasks, and provided better support for SMP systems. However it yielded poor interactive performance.
- Starting in 2.6.23, the **Completely Fair Scheduler, CFS,** became the standard Linux scheduling system. See sidebar below

CFS (Completely Fair Scheduler) Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:

When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority . Because the red-black tree is balanced, navigating it to discover the leftmost node will require **O(lgN)** operations (where **N** is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

New sidebar in ninth edition

- The Linux scheduler is a preemptive priority-based algorithm with two priority ranges - **Real time** from 0 to 99 and a **nice** range from 100 to 140.
- Unlike Solaris or XP, Linux assigns longer time quantums to higher priority tasks.
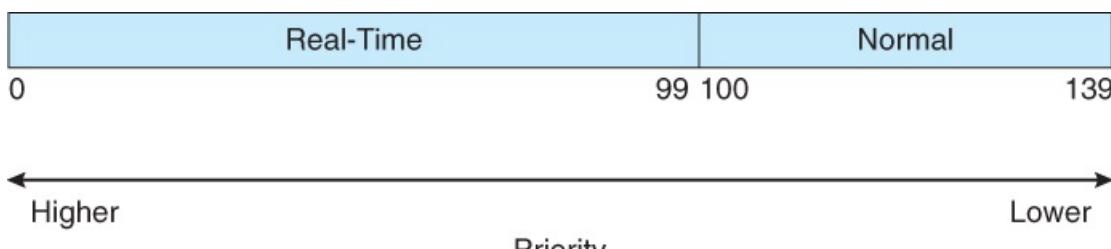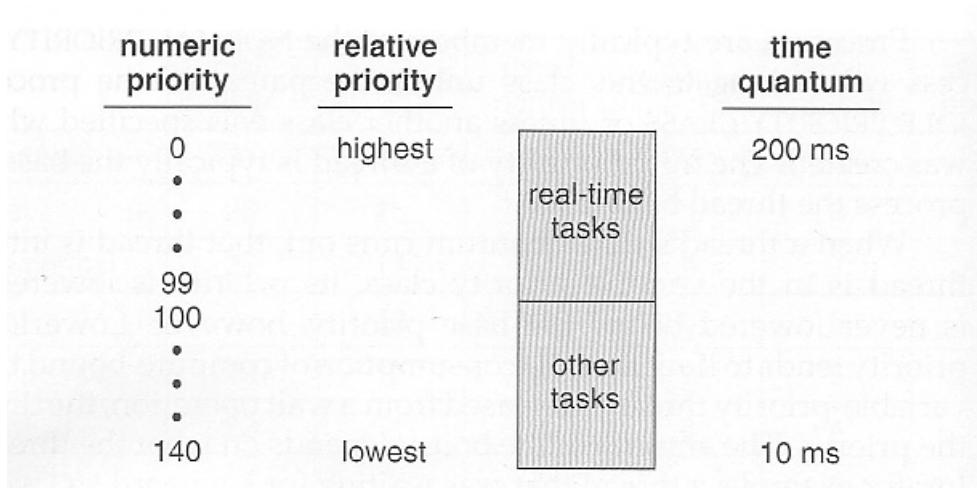
Figure 6.21 - Scheduling priorities on a Linux system.
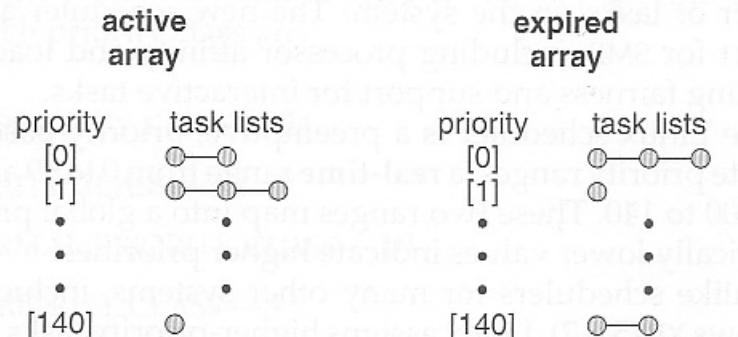
Figure 5.13 The relationship between priorities and time-slice length.

Replaced by Figure 6.21 in the ninth edition. Was Figure 5.15 in eighth edition

/* The Following material was omitted from the ninth edition:

- ○ A runnable task is considered eligible for execution as long as it has not consumed all the time available in it's time slice. Those tasks are stored in an ***active array***, indexed according to priority.
- ○ When a process consumes its time slice, it is moved to an ***expired array.*** The tasks priority may be re-assigned as part of the transferal.
- ○ When the active array becomes empty, the two arrays are swapped.
- ○ These arrays are stored in ***runqueue*** structures. On multiprocessor machines, each processor has its own scheduler with its own runqueue.

Figure 5.14 List of tasks indexed according to priority.

Omitted from ninth edition. Figure 5.16 in eighth edition.

*/

6.7.2 Example: Windows XP Scheduling (was 5.6.2)

- Windows XP uses a priority-based preemptive scheduling algorithm.
- The dispatcher uses a 32-level priority scheme to determine the order of thread execution, divided into two classes - ***variable class*** from 1 to 15 and ***real-time class*** from 16 to 31, (plus a thread at priority 0 managing memory.)
- There is also a special ***idle*** thread that is scheduled when no other threads are ready.
- Win XP identifies 7 priority classes (rows on the table below), and 6 relative priorities within each class (columns.)
- Processes are also each given a **base priority** within their priority class. When variable class processes consume their entire time quanta, then their priority gets lowered, but not below their base priority.
- Processes in the foreground (active window) have their scheduling quanta multiplied by 3, to give better response to interactive processes in the foreground.

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Figure 6.22 - Windows thread priorities.

6.7.1 Example: Solaris Scheduling

- Priority-based kernel thread scheduling.
- Four classes (real-time, system, interactive, and time-sharing), and multiple queues / algorithms within each class.
- Default is time-sharing.
  - Process priorities and time slices are adjusted dynamically in a multilevel-feedback priority queue system.
  - Time slices are inversely proportional to priority - Higher priority jobs get smaller time slices.
  - Interactive jobs have higher priority than CPU-Bound ones.
  - See the table below for some of the 60 priority levels and how they shift. "Time quantum expired" and "return from sleep" indicate the new priority when those events occur. (Larger numbers are a higher, i.e. better priority.)

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Figure 6.23 - Solaris dispatch table for time-sharing and interactive threads.

- Solaris 9 introduced two new scheduling classes: Fixed priority and fair share.
  - Fixed priority is similar to time sharing, but not adjusted dynamically.
  - Fair share uses shares of CPU time rather than priorities to schedule jobs. A certain share of the available CPU time is allocated to a project, which is a set of processes.
- System class is reserved for kernel use. (User programs running in kernel mode are NOT considered in the
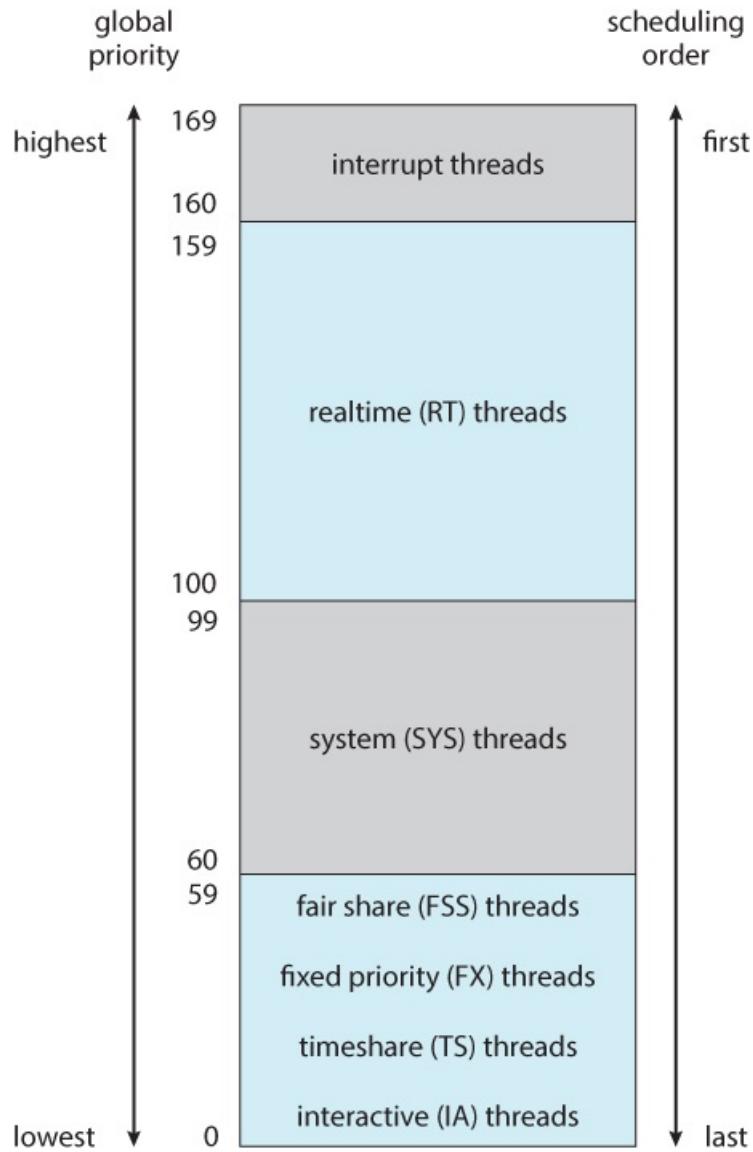
Figure 6.24 - Solaris scheduling.
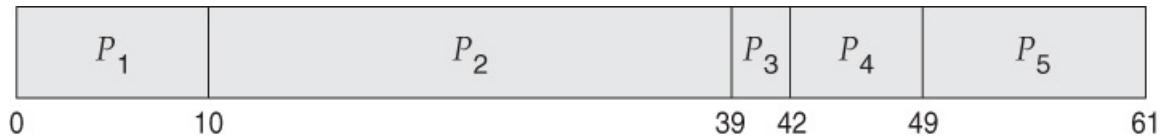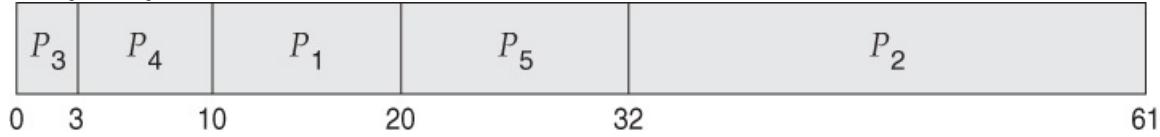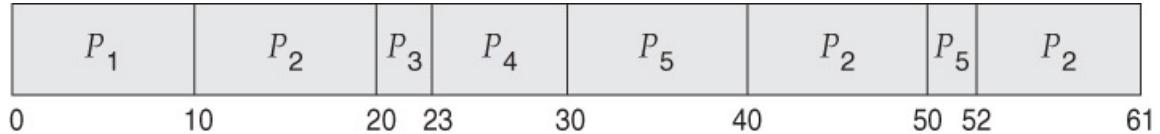
6.8 Algorithm Evaluation

- The first step in determining which algorithm (and what parameter settings within that algorithm) is optimal for a particular operating environment is to determine what criteria are to be used, what goals are to be targeted, and what constraints if any must be applied. For example, one might want to "maximize CPU utilization, subject to a maximum response time of 1 second".
- Once criteria have been established, then different algorithms can be analyzed and a "best choice" determined. The following sections outline some different methods for determining the "best choice".

6.8.1 Deterministic Modeling

- If a specific workload is known, then the exact values for major criteria can be fairly easily calculated, and the "best" determined. For example, consider the following workload (with all processes arriving at time 0), and the resulting schedules determined by three different algorithms:

Process	Burst Time
P1	10
P2	29
P3	3
P4	7
P5	12

FCFS:

P_1	P_2	P_3	P_4	P_5

0   10   39 42   49   61

Non-preemptive SJF:

P_3	P_4	P_1	P_5	P_2

0   3   10   20   32   61

Round Robin:

P_1	P_2	P_3	P_4	P_5	P_2	P_5	P_2

0   10   20   23   30   40   50 52   61

- The average waiting times for FCFS, SJF, and RR are 28ms, 13ms, and 23ms respectively.
- Deterministic modeling is fast and easy, but it requires specific known input, and the results only apply for that particular set of input. However by examining multiple similar cases, certain trends can be observed. (Like the fact that for processes arriving at the same time, SJF will always yield the shortest average wait time.)

6.8.2 Queuing Models

- Specific process data is often not available, particularly for future times. However a study of historical performance can often produce statistical descriptions of certain important parameters, such as the rate at which new processes arrive, the ratio of CPU bursts to I/O times, the distribution of CPU burst times and I/O burst times, etc.
- Armed with those probability distributions and some mathematical formulas, it is possible to calculate certain performance characteristics of individual waiting queues. For example, **Little's Formula** says that for an average queue length of N, with an average waiting time in the queue of W, and an average arrival of new jobs in the queue of Lambda, then these three terms can be related by:

$$N = Lambda * W$$

- Queuing models treat the computer as a network of interconnected queues, each of which is described by its probability distribution statistics and formulas such as Little's formula. Unfortunately real systems and modern scheduling algorithms are so complex as to make the mathematics intractable in many cases with real systems.

6.8.3 Simulations

- Another approach is to run computer simulations of the different proposed algorithms (and adjustment parameters) under different load conditions, and to analyze the results to determine the "best" choice of operation for a particular load pattern.
- Operating conditions for simulations are often randomly generated using distribution functions similar to those described above.
- A better alternative when possible is to generate *trace tapes*, by monitoring and logging the performance of a real system under typical expected work loads. These are better because they provide a more accurate picture of system loads, and also because they allow multiple simulations to be run with the identical process load, and not just statistically equivalent loads. A compromise is to randomly determine system loads and then save the results into a file, so that all simulations can be run against identical randomly determined system loads.
- Although trace tapes provide more accurate input information, they can be difficult and expensive to collect and store, and their use increases the complexity of the simulations significantly. There is also some question as to whether the future performance of the new system will really match the past performance of the old system. (If the system runs faster, users may take fewer coffee breaks, and submit more processes per hour than under the old system. Conversely if the turnaround time for jobs is longer, intelligent users

may think more carefully about the jobs they submit rather than randomly submitting jobs and hoping that one of them works out.)
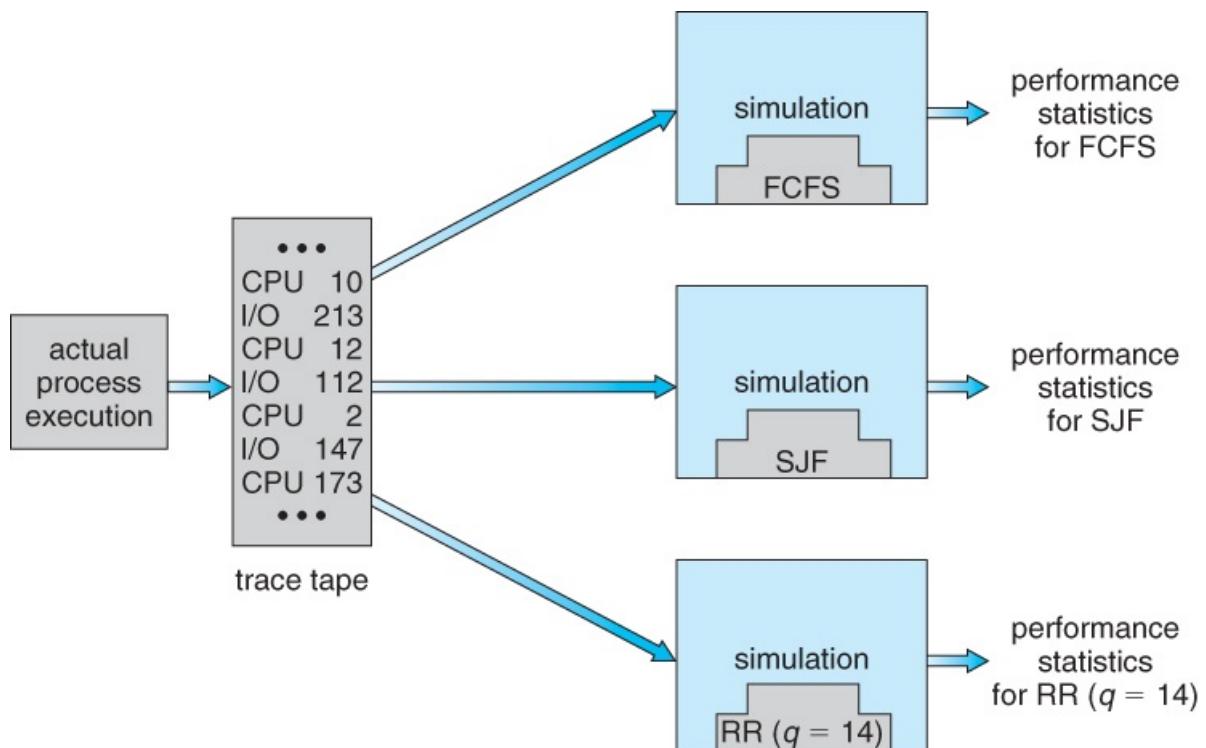
Figure 6.25 - Evaluation of CPU schedulers by simulation.

6.8.4 Implementation

- The only real way to determine how a proposed scheduling algorithm is going to operate is to implement it on a real system.
- For experimental algorithms and those under development, this can cause difficulties and resistance among users who don't care about developing OSes and are only trying to get their daily work done.
- Even in this case, the measured results may not be definitive, for at least two major reasons: (1) System work loads are not static, but change over time as new programs are installed, new users are added to the system, new hardware becomes available, new work projects get started, and even societal changes. (For example the explosion of the Internet has drastically changed the amount of network traffic that a system sees and the importance of handling it with rapid response times.) (2) As mentioned above, changing the scheduling system may have an impact on the work load and the ways in which users use the system. (The book gives an example of a programmer who modified his code to write an arbitrary character to the screen at regular intervals, just so his job would be classified as interactive and placed into a higher priority queue.)
- Most modern systems provide some capability for the system administrator to adjust scheduling parameters, either on the fly or as the result of a reboot or a kernel rebuild.
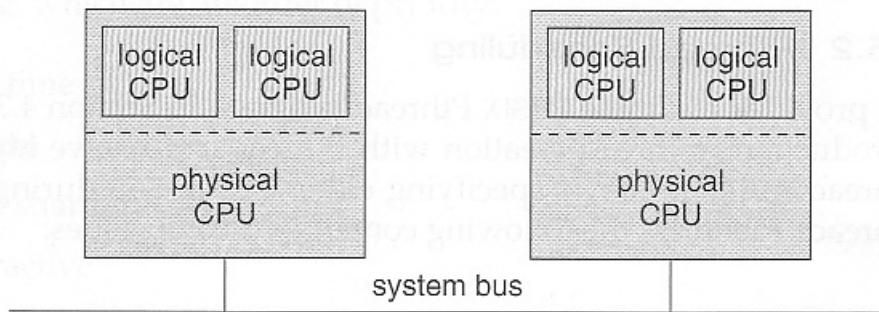
6.9 Summary

Material Omitted from the Eighth Edition:

Was 5.4.4 Symmetric Multithreading (Omitted from 8th edition)

- An alternative strategy to SMP is **SMT, Symmetric Multi-Threading,** in which multiple virtual (logical) CPUs are used instead of (or in combination with) multiple physical CPUs.
- SMT must be supported in hardware, as each logical CPU has its own registers and handles its own interrupts. (Intel refers to SMT as **hyperthreading technology.)**
- To some extent the OS does not need to know if the processors it is managing are real or virtual. On the other hand, some scheduling decisions can be optimized if the scheduler knows the mapping of virtual processors to real CPUs. (Consider the scheduling of two CPU-intensive processes on the architecture shown below.)

Figure 5.8 A typical SMT architecture

omitted

Deadlocks

References:

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 7

7.1 System Model

- For the purposes of deadlock discussion, a system can be modeled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes, each having different needs.
- Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc.
- By definition, all the resources within a category are equivalent, and a request of this category can be equally satisfied by any one of the resources in that category. If this is not the case (i.e. if there is some difference between the resources within a category), then that category needs to be further divided into separate categories. For example, "printers" may need to be separated into "laser printers" and "color inkjet printers".
- Some categories may have a single resource.
- In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence:
    1. Request - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. For example the system calls open( ), malloc( ), new( ), and request( ).
    2. Use - The process uses the resource, e.g. prints to the printer or reads from the file.
    3. Release - The process relinquishes the resource. so that it becomes available for other processes. For example, close( ), free( ), delete( ), and release( ).
- For all kernel-managed resources, the kernel keeps track of what resources are free and which are allocated, to which process they are allocated, and a queue of processes waiting for this resource to become available. Application-managed resources can be controlled using mutexes or wait( ) and signal( ) calls, (i.e. binary or counting semaphores.)
- A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set (and which can only be released when that other waiting process makes progress.)

7.2 Deadlock Characterization

DEADLOCK WITH MUTEX LOCKS

Let's see how deadlock can occur in a multithreaded Pthread program using mutex locks. The `pthread_mutex_init()` function initializes an unlocked mutex. Mutex locks are acquired and released using `pthread_mutex_lock()` and `pthread_mutex_unlock()`, respectively. If a thread attempts to acquire a locked mutex, the call to `pthread_mutex_lock()` blocks the thread until the owner of the mutex lock invokes `pthread_mutex_unlock()`.

Two mutex locks are created in the following code example:

```
/* Create and initialize the mutex locks */
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

Next, two threads—`thread_one` and `thread_two`—are created, and both these threads have access to both mutex locks. `thread_one` and `thread_two` run in the functions `do_work_one()` and `do_work_two()`, respectively, as shown below:

```
/* thread_one runs in this function */
void *do_work_one(void *param)
```

```
{
        pthread_mutex_lock(&first_mutex);
        pthread_mutex_lock(&second_mutex);
        /**
         * Do some work
         */
        pthread_mutex_unlock(&second_mutex);
        pthread_mutex_unlock(&first_mutex);

        pthread_exit(0);
}

        /* thread_two runs in this function */
        void *do_work_two(void *param)
        {
        pthread_mutex_lock(&second_mutex);
        pthread_mutex_lock(&first_mutex);
        /**
         * Do some work
         */
        pthread_mutex_unlock(&first_mutex);
        pthread_mutex_unlock(&second_mutex);

        pthread_exit(0);
        }
```

In this example, thread_one attempts to acquire the mutex locks in the order
(1) first_mutex, (2) second_mutex, while thread_two attempts to acquire
the mutex locks in the order (1) second_mutex, (2) first_mutex. Deadlock
is possible if thread_one acquires first_mutex while thread_two acquires
second_mutex.

Note that, even though deadlock is possible, it will not occur if thread_one
can acquire and release the mutex locks for first_mutex and second_mutex
before thread_two attempts to acquire the locks. And, of course, the order
in which the threads run depends on how they are scheduled by the CPU
scheduler. This example illustrates a problem with handling deadlocks: it is
difficult to identify and test for deadlocks that may occur only under certain
scheduling circumstances.

New Sidebar in Ninth Edition

7.2.1 Necessary Conditions

- There are four conditions that are necessary to achieve deadlock:
  1. **Mutual Exclusion** - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.
  2. **Hold and Wait** - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.
  3. **No preemption** - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.
  4. **Circular Wait** - A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[i] is waiting for P[(i + 1) % (N + 1)]. (Note that this condition implies the hold-and-wait condition, but it is easier to deal with the conditions if the four are considered separately.)

7.2.2 Resource-Allocation Graph

- In some cases deadlocks can be understood more clearly through the use of **Resource-Allocation Graphs**, having the following properties:
  - A set of resource categories, { R1, R2, R3, . . ., RN }, which appear as square nodes on the graph. Dots inside the resource nodes indicate specific instances of the resource. (E.g. two dots might represent two laser printers.)
  - A set of processes, { P1, P2, P3, . . ., PN }
  - **Request Edges -** A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available.
  - **Assignment Edges -** A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj.
  - Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted. (However note also that request edges point to the category box, whereas assignment edges emanate from a particular instance dot within the box.)
  - For example:
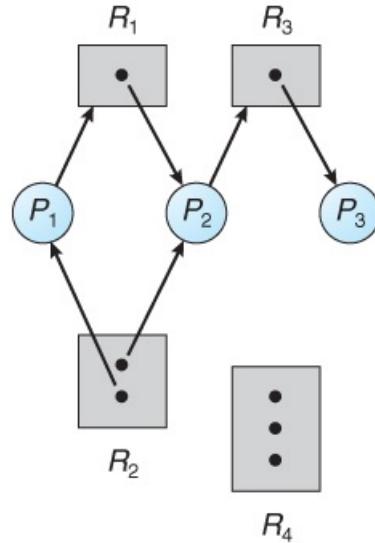
Figure 7.1 - Resource allocation graph

- If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are **directed** graphs.) See the example in Figure 7.2 above.
- If a resource-allocation graph does contain cycles **AND** each resource category contains only a single instance, then a deadlock exists.
- If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the *possibility* of a deadlock, but does not guarantee one. Consider, for example, Figures 7.3 and 7.4 below:
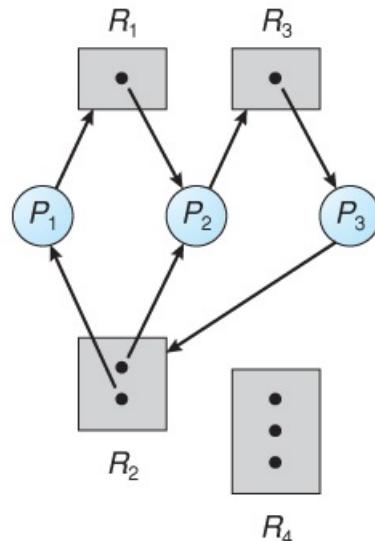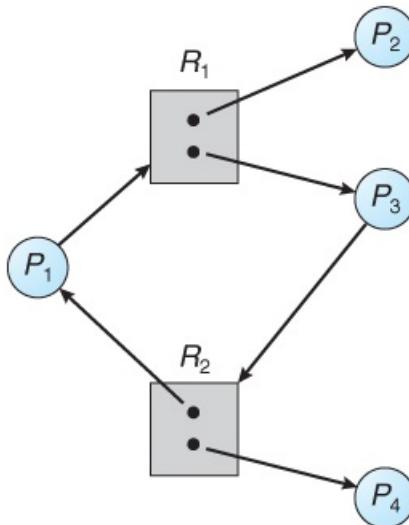
Figure 7.2 - Resource allocation graph with a deadlock

Figure 7.3 - Resource allocation graph with a cycle but no deadlock

7.3 Methods for Handling Deadlocks

- Generally speaking there are three ways of handling deadlocks:
  1. Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.
  2. Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected.
  3. Ignore the problem all together - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to incur the constant overhead and system performance penalties associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take.
- In order to avoid deadlocks, the system must have additional information about all processes. In particular, the system must know what resources a process will or may request in the future. (Ranging from a simple worst-case maximum to a complete resource request and release plan for each process, depending on the particular algorithm.)
- Deadlock detection is fairly straightforward, but deadlock recovery requires either aborting processes or preempting resources, neither of which is an attractive alternative.
- If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually slow down, as more and more processes become stuck waiting for resources currently held by the deadlock and by other waiting processes. Unfortunately this slowdown can be indistinguishable from a general system slowdown when a real-time process has heavy computing needs.

7.4 Deadlock Prevention

- Deadlocks can be prevented by preventing at least one of the four required conditions:

7.4.1 Mutual Exclusion

- Shared resources such as read-only files do not lead to deadlocks.
- Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

7.4.2 Hold and Wait

- To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:
  - Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
  - Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.
  - Either of the methods described above can lead to starvation if a process requires one or more popular resources.

7.4.3 No Preemption

- Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.
  - One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.
  - Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources *and* are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.
  - Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

7.4.4 Circular Wait

- One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
- In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.
- One big challenge in this scheme is determining the relative ordering of the different resources

7.5 Deadlock Avoidance

- The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the aforementioned conditions.
- This requires more information about each process, AND tends to lead to low device utilization. (I.e. it is a conservative approach.)
- In some algorithms the scheduler only needs to know the *maximum* number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the *schedule* of exactly what resources may be needed in what order.
- When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.
- A resource allocation *state* is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.

7.5.1 Safe State

- A state is *safe* if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.
- More formally, a state is safe if there exists a *safe sequence* of processes { P0, P1, P2, ..., PN } such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i. (I.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)
- If a safe sequence does not exist, then the system is in an unsafe state, which *MAY* lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)
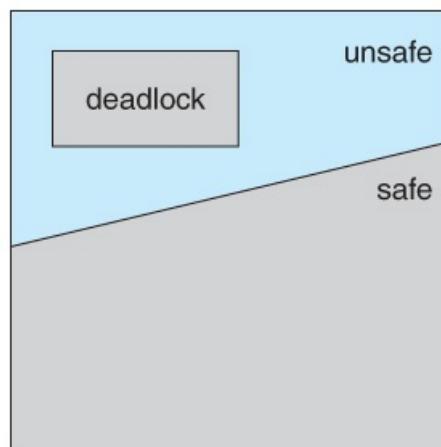
Figure 7.6 - Safe, unsafe, and deadlocked state spaces.

- For example, consider a system with 12 tape drives, allocated as follows. Is this a safe state? What is the

safe sequence?

	Maximum Needs	Current Allocation
P0	10	5
P1	4	2
P2	9	2

- What happens to the above table if process P2 requests and is granted one more tape drive?
- Key to the safe state approach is that when a request is made for resources, the request is granted only if the resulting allocation state is a safe one.

7.5.2 Resource-Allocation Graph Algorithm

- If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.
- In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with *claim edges*, noted by dashed lines, which point from a process to a resource that it may request in the future.
- In order for this technique to work, all claim edges must be added to the graph for any particular process before that process is allowed to request any resources. (Alternatively, processes may only make requests for resources for which they have already established claim edges, and claim edges cannot be added to any process that is currently holding resources.)
- When a process makes a request, the claim edge Pi->Rj is converted to a request edge. Similarly when a resource is released, the assignment reverts back to a claim edge.
- This approach works by denying requests that would produce cycles in the resource-allocation graph, taking claim edges into effect.
- Consider for example what happens when process P2 requests resource R2:
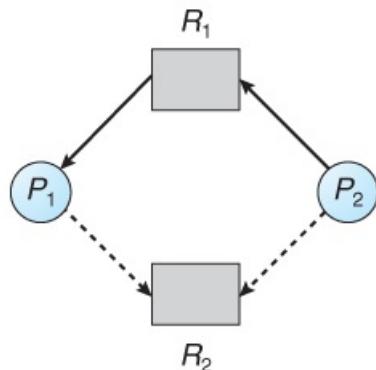
Figure 7.7 - Resource allocation graph for deadlock avoidance

- The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.
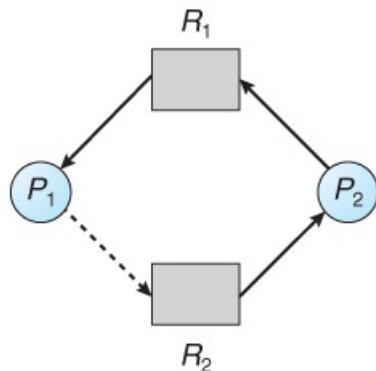
Figure 7.8 - An unsafe state in a resource allocation graph

7.5.3 Banker's Algorithm

- For resource categories that contain more than one instance the resource-allocation graph method does not work, and more complex (and less efficient) methods must be chosen.
- The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients. (A banker won't loan out a little money to start building a house unless they are assured that they will later be able to loan out the rest of the money to finish the house.)
- When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system.
- When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely.
- The banker's algorithm relies on several key data structures: (where n is the number of processes and m is the number of resource categories.)
  - Available[m] indicates how many resources are currently available of each type.
  - Max[n][m] indicates the maximum demand of each process of each resource.
  - Allocation[n][m] indicates the number of each resource category allocated to each process.
  - Need[n][m] indicates the remaining resources needed of each type for each process. (Note that Need[i][j] = Max[i][j] - Allocation[i][j] for all i, j.)
- For simplification of discussions, we make the following notations / observations:
  - One row of the Need vector, Need[i], can be treated as a vector corresponding to the needs of process i, and similarly for Allocation and Max.
  - A vector X is considered to be <= a vector Y if X[i] <= Y[i] for all i.

7.5.3.1 Safety Algorithm

- In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe.
- This algorithm determines if the current state of a system is safe, according to the following steps:
  1. Let Work and Finish be vectors of length m and n respectively.
     - Work is a working copy of the available resources, which will be modified during the analysis.
     - Finish is a vector of booleans indicating whether a particular process can finish. (or has finished so far in the analysis.)
     - Initialize Work to Available, and Finish to false for all elements.
  2. Find an i such that both (A) Finish[i] == false, and (B) Need[i] < Work. This process has not finished, but could with the given available working set. If no such i exists, go to step 4.
  3. Set Work = Work + Allocation[i], and set Finish[i] to true. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.
  4. If finish[i] == true for all i, then the state is a safe state, because a safe sequence has been found.
- (JTB's Modification:
  1. In step 1. instead of making Finish an array of booleans initialized to false, make it an array of ints initialized to 0. Also initialize an int s = 0 as a step counter.
  2. In step 2, look for Finish[i] == 0.
  3. In step 3, set Finish[i] to ++s. S is counting the number of finished processes.
  4. For step 4, the test can be either Finish[i] > 0 for all i, or s >= n. The benefit of this method is that if a safe state exists, then Finish[ ] indicates one safe sequence (of possibly many.)
)

7.5.3.2 Resource-Request Algorithm (The Bankers Algorithm)

- Now that we have a tool for determining if a particular state is safe or not, we are now ready to look at the Banker's algorithm itself.
- This algorithm determines if a new request is safe, and grants it only if it is safe to do so.
- When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request, as follows:
  1. Let Request[n][m] indicate the number of resources of each type currently requested by processes. If Request[i] > Need[i] for any process i, raise an error condition.
  2. If Request[i] > Available for any process i, then that process must wait for resources to become available. Otherwise the process can continue to step 3.
  3. Check to see if the request can be granted safely, by pretending it has been granted and

then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must wait until its request can be granted safely.The procedure for granting a request (or pretending to for testing purposes) is:

- Available = Available - Request
- Allocation = Allocation + Request
- Need = Need - Request

7.5.3.3 An Illustrative Example

- Consider the following situation:

	Allocation A B C	Max A B C	Available A B C	Need A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

- And now consider what happens if process P1 requests 1 instance of A and 2 instances of C. (Request[1] = (1, 0, 2))

	Allocation A B C	Need A B C	Available A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	
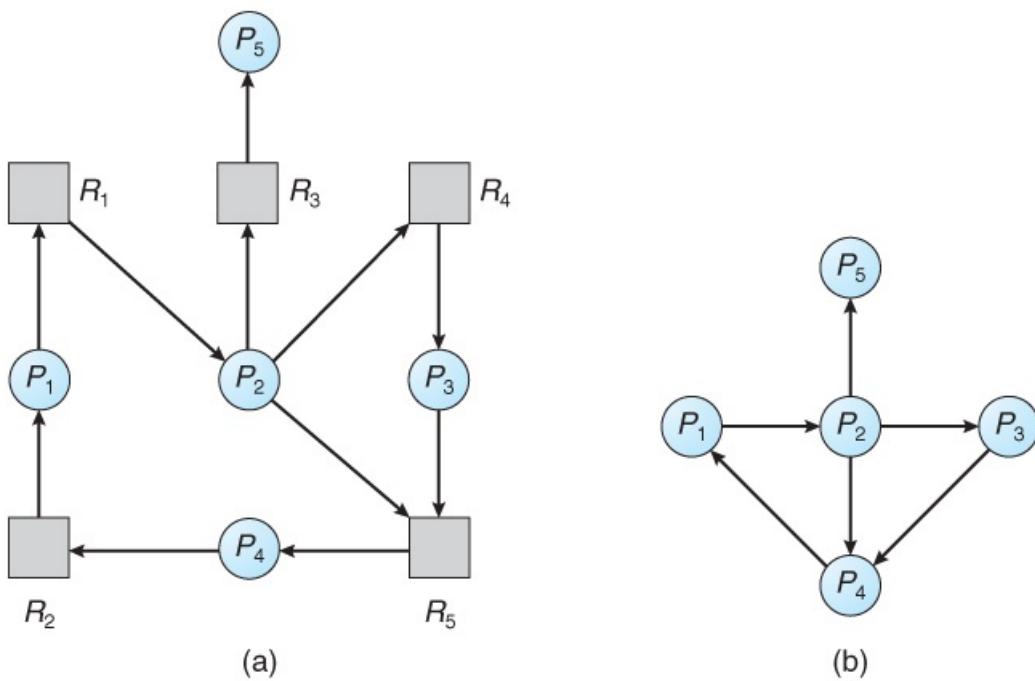
- What about requests of (3, 3,0) by P4? or (0, 2, 0) by P0? Can these be safely granted? Why or why not?

7.6 Deadlock Detection

- If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow.
- In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources preempted.

7.6.1 Single Instance of Each Resource Type

- If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a *wait-for graph*.
- A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.
- An arc from Pi to Pj in a wait-for graph indicates that process Pi is waiting for a resource that process Pj is currently holding.

Figure 7.9 - (a) Resource allocation graph. (b) Corresponding wait-for graph

- As before, cycles in the wait-for graph indicate deadlocks.
- This algorithm must maintain the wait-for graph, and periodically search it for cycles.

7.6.2 Several Instances of a Resource Type

- The detection algorithm outlined here is essentially the same as the Banker's algorithm, with two subtle differences:
  - In step 1, the Banker's Algorithm sets Finish[i] to false for all i. The algorithm presented here sets Finish[i] to false only if Allocation[i] is not zero. If the currently allocated resources for this process are zero, the algorithm sets Finish[i] to true. This is essentially assuming that IF all of the other processes can finish, then this process can finish also. Furthermore, this algorithm is specifically looking for which processes are involved in a deadlock situation, and a process that does not have any resources allocated cannot be involved in a deadlock, and so can be removed from any further consideration.
  - Steps 2 and 3 are unchanged
  - In step 4, the basic Banker's Algorithm says that if Finish[i] == true for all i, that there is no deadlock. This algorithm is more specific, by stating that if Finish[i] == false for any process Pi, then that process is specifically involved in the deadlock which has been detected.
- (Note: An alternative method was presented above, in which Finish held integers instead of booleans. This vector would be initialized to all zeros, and then filled with increasing integers as processes are detected which can finish. If any processes are left at zero when the algorithm completes, then there is a deadlock, and if not, then the integers in finish describe a safe sequence. To modify this algorithm to match this section of the text, processes with allocation = zero could be filled in with N, N - 1, N - 2, etc. in step 1, and any processes left with Finish = 0 in step 4 are the deadlocked processes.)
- Consider, for example, the following state, and determine if it is currently deadlocked:

	Allocation	*Request*	*Available*
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Now suppose that process P2 makes a request for an additional instance of type C, yielding the state shown below. Is the system now deadlocked?

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

7.6.3 Detection-Algorithm Usage

- When should the deadlock detection be done? Frequently, or infrequently?
- The answer may depend on how frequently deadlocks are expected to occur, as well as the possible consequences of not catching them immediately. (If deadlocks are not removed immediately when they occur, then more and more processes can "back up" behind the deadlock, making the eventual task of unblocking the system more difficult and possibly damaging to more processes.)
- There are two obvious approaches, each with trade-offs:
  1. Do deadlock detection after every resource allocation which cannot be immediately granted. This has the advantage of detecting the deadlock right away, while the minimum number of processes are involved in the deadlock. (One might consider that the process whose request triggered the deadlock condition is the "cause" of the deadlock, but realistically all of the processes in the cycle are equally responsible for the resulting deadlock.) The down side of this approach is the extensive overhead and performance hit caused by checking for deadlocks so frequently.
  2. Do deadlock detection only when there is some clue that a deadlock may have occurred, such as when CPU utilization reduces to 40% or some other magic number. The advantage is that deadlock detection is done much less frequently, but the down side is that it becomes impossible to detect the processes involved in the original deadlock, and so deadlock recovery can be more complicated and damaging to more processes.
  3. (As I write this, a third alternative comes to mind: Keep a historical log of resource allocations, since that last known time of no deadlocks. Do deadlock checks periodically (once an hour or when CPU usage is low?), and then use the historical log to trace through and determine when the deadlock occurred and what processes caused the initial deadlock. Unfortunately I'm not certain that breaking the original deadlock would then free up the resulting log jam.)

7.7 Recovery From Deadlock

- There are three basic approaches to recovery from deadlock:
  1. Inform the system operator, and allow him/her to take manual intervention.
  2. Terminate one or more processes involved in the deadlock
  3. Preempt resources.

7.7.1 Process Termination

- Two basic approaches, both of which recover resources allocated to terminated processes:
  - Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.
  - Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.
- In the latter case there are many factors that can go into deciding which processes to terminate next:
  1. Process priorities.
  2. How long the process has been running, and how close it is to finishing.
  3. How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
  4. How many more resources does the process need to complete.
  5. How many processes will need to be terminated
  6. Whether the process is interactive or batch.
  7. (Whether or not the process has made non-restorable changes to any resource.)

7.7.2 Resource Preemption

- When preempting resources to relieve deadlock, there are three important issues to be addressed:
  1. **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
  2. **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (I.e. abort the process and make it start over.)
  3. **Starvation** - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.

7.8 Summary

Main Memory

References:
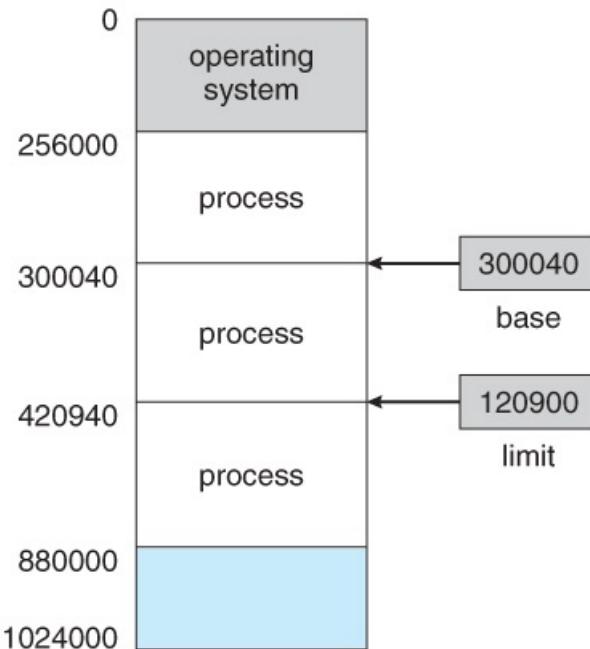
1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 8

8.1 Background
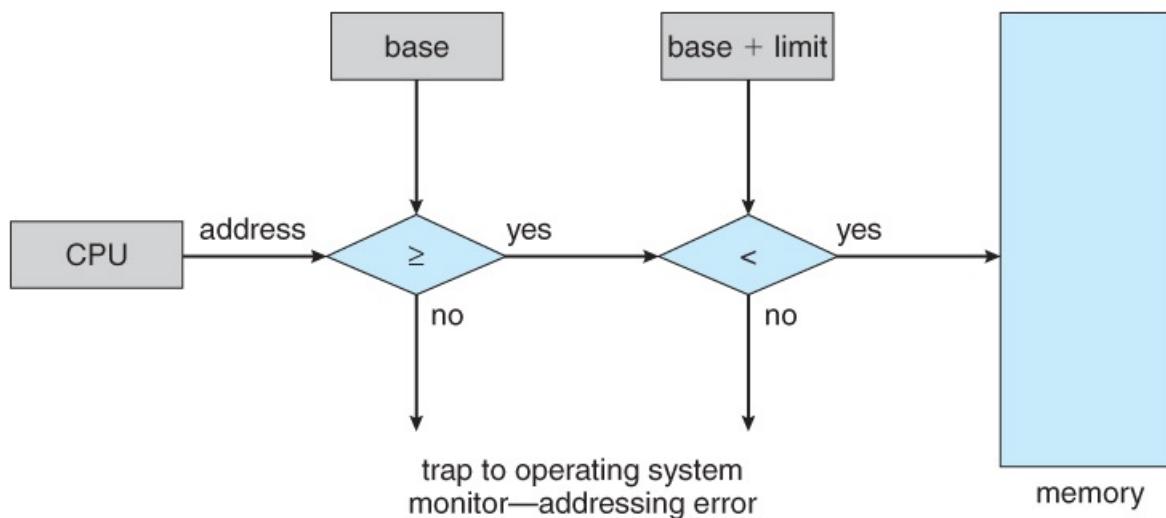
- Obviously memory accesses and memory management are a very important part of modern computer operation. Every instruction has to be fetched from memory before it can be executed, and most instructions involve retrieving data from memory or storing data in memory or both.
- The advent of multi-tasking OSes compounds the complexity of memory management, because because as processes are swapped in and out of the CPU, so must their code and data be swapped in and out of memory, all at high speeds and without interfering with any other processes.
- Shared memory, virtual memory, the classification of memory as read-only versus read-write, and concepts like copy-on-write forking all further complicate the issue.

8.1.1 Basic Hardware

- It should be noted that from the memory chips point of view, all memory accesses are equivalent. The memory hardware doesn't know what a particular part of memory is being used for, nor does it care. This is almost true of the OS as well, although not entirely.
- The CPU can only access its registers and main memory. It cannot, for example, make direct access to the hard drive, so any data stored there must first be transferred into the main memory chips before the CPU can work with it. (Device drivers communicate with their hardware via interrupts and "memory" accesses, sending short instructions for example to transfer data from the hard drive to a specified location in main memory. The disk controller monitors the bus for such instructions, transfers the data, and then notifies the CPU that the data is there with another interrupt, but the CPU never gets direct access to the disk.)
- Memory accesses to registers are very fast, generally one clock tick, and a CPU may be able to execute more than one machine instruction per clock tick.
- Memory accesses to main memory are comparatively slow, and may take a number of clock ticks to complete. This would require intolerable waiting by the CPU if it were not for an intermediary fast memory *cache* built into most modern CPUs. The basic idea of the cache is to transfer chunks of memory at a time from the main memory to the cache, and then to access individual memory locations one at a time from the cache.
- User processes must be restricted so that they only access memory locations that "belong" to that particular process. This is usually implemented using a base register and a limit register for each process, as shown in Figures 8.1 and 8.2 below. *Every* memory access made by a user process is checked against these two registers, and if a memory access is attempted outside the valid range, then a fatal error is generated. The OS obviously has access to all existing memory locations, as this is necessary to swap users' code and data in and out of memory. It should also be obvious that changing the contents of the base and limit registers is a privileged activity, allowed only to the OS kernel.
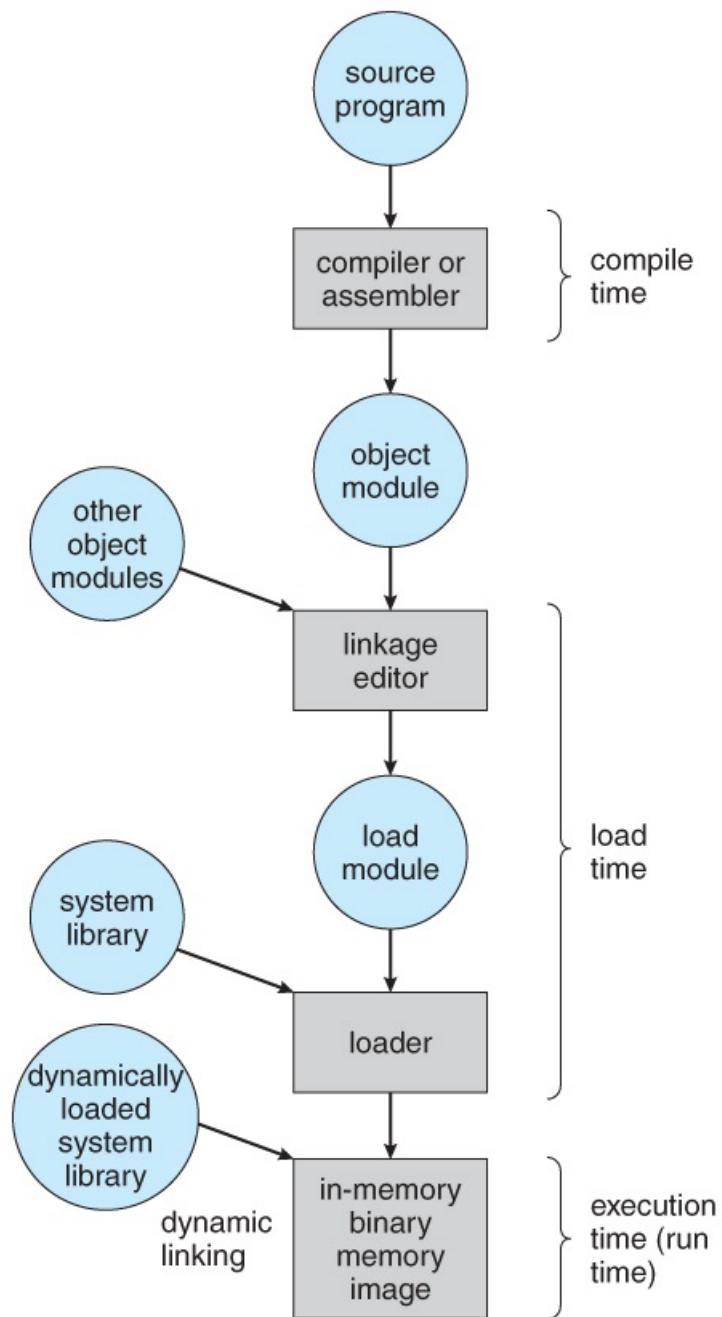
Figure 8.1 - A base and a limit register define a logical addresss space

Figure 8.2 - Hardware address protection with base and limit registers

8.1.2 Address Binding
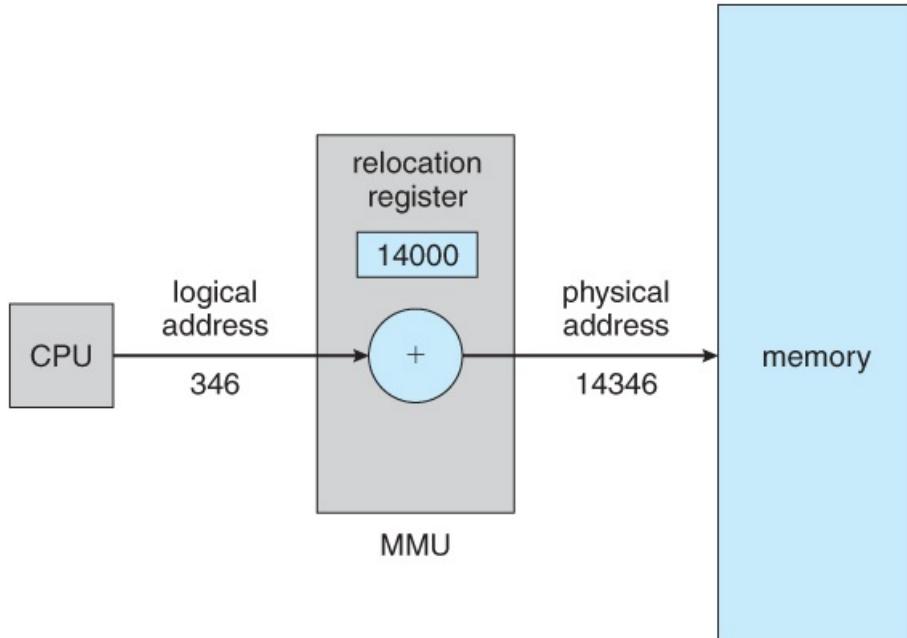
- User programs typically refer to memory addresses with symbolic names such as "i", "count", and "averageTemperature". These symbolic names must be mapped or **bound** to physical memory addresses, which typically occurs in several stages:
    - **Compile Time** - If it is known at compile time where a program will reside in physical memory, then **absolute code** can be generated by the compiler, containing actual physical addresses. However if the load address changes at some later time, then the program will have to be recompiled. DOS .COM programs use compile time binding.
    - **Load Time** - If the location at which a program will be loaded is not known at compile time, then the compiler must generate **relocatable code**, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.
    - **Execution Time** - If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time. This requires special hardware, and is the method implemented by most modern OSes.
- Figure 8.3 shows the various stages of the binding processes and the units involved in each stage:

Figure 8.3 - Multistep processing of a user program

8.1.3 Logical Versus Physical Address Space

- The address generated by the CPU is a *logical address*, whereas the address actually seen by the memory hardware is a *physical address*.
- Addresses bound at compile time or load time have identical logical and physical addresses.
- Addresses created at execution time, however, have different logical and physical addresses.
  - In this case the logical address is also known as a *virtual address*, and the two terms are used interchangeably by our text.
  - The set of all logical addresses used by a program composes the *logical address space*, and the set of all corresponding physical addresses composes the *physical address space.*
- The run time mapping of logical to physical addresses is handled by the *memory-management unit, MMU*.
  - The MMU can take on many forms. One of the simplest is a modification of the base-register scheme described earlier.
  - The base register is now termed a *relocation register*, whose value is added to every memory request at the hardware level.
- Note that user programs never see physical addresses. User programs work entirely in logical address space, and any memory references or manipulations are done using purely logical addresses. Only when the address gets sent to the physical memory chips is the physical memory address generated.

Figure 8.4 - Dynamic relocation using a relocation register

8.1.4 Dynamic Loading

- Rather than loading an entire program into memory at once, dynamic loading loads up each routine as it is called. The advantage is that unused routines need never be loaded, reducing total memory usage and generating faster program startup times. The downside is the added complexity and overhead of checking to see if a routine is loaded every time it is called and then then loading it up if it is not already loaded.

8.1.5 Dynamic Linking and Shared Libraries

- With **static linking** library modules get fully included in executable modules, wasting both disk space and main memory usage, because every program that included a certain routine from the library would have to have their own copy of that routine linked into their executable code.
- With **dynamic linking**, however, only a stub is linked into the executable module, containing references to the actual library module linked in at run time.
    - This method saves disk space, because the library routines do not need to be fully included in the executable modules, only the stubs.
    - We will also learn that if the code section of the library routines is **reentrant**, (meaning it does not modify the code while it runs, making it safe to re-enter it), then main memory can be saved by loading only one copy of dynamically linked routines into memory and sharing the code amongst all processes that are concurrently using it. (Each process would have their own copy of the **data** section of the routines, but that may be small relative to the code segments.) Obviously the OS must manage shared routines in memory.
    - An added benefit of **dynamically linked libraries** (**DLLs**, also known as **shared libraries** or **shared objects** on UNIX systems) involves easy upgrades and updates. When a program uses a routine from a standard library and the routine changes, then the program must be re-built (re-linked) in order to incorporate the changes. However if DLLs are used, then as long as the stub doesn't change, the program can be updated merely by loading new versions of the DLLs onto the system. Version information is maintained in both the program and the DLLs, so that a program can specify a particular version of the DLL if necessary.
    - In practice, the first time a program calls a DLL routine, the stub will recognize the fact and will replace itself with the actual routine from the DLL library. Further calls to the same routine will access the routine directly and not incur the overhead of the stub access. (Following the UML **Proxy Pattern**.)
    - (Additional information regarding dynamic linking is available at http://www.iecc.com/linker/linker10.html)

8.2 Swapping

- A process must be loaded into memory in order to execute.
- If there is not enough memory available to keep all running processes in memory at the same time, then some processes who are not currently using the CPU may have their memory swapped out to a fast local disk called the

backing store.

8.2.1 Standard Swapping

- If compile-time or load-time address binding is used, then processes must be swapped back into the same memory location from which they were swapped out. If execution time binding is used, then the processes can be swapped back into any available location.
- Swapping is a very slow process compared to other operations. For example, if a user process occupied 10 MB and the transfer rate for the backing store were 40 MB per second, then it would take 1/4 second (250 milliseconds) just to do the data transfer. Adding in a latency lag of 8 milliseconds and ignoring head seek time for the moment, and further recognizing that swapping involves moving old data out as well as new data in, the overall transfer time required for this swap is 512 milliseconds, or over half a second. For efficient processor scheduling the CPU time slice should be significantly longer than this lost transfer time.
- To reduce swapping transfer overhead, it is desired to transfer as little information as possible, which requires that the system know how much memory a process ***is*** using, as opposed to how much it ***might*** use. Programmers can help with this by freeing up dynamic memory that they are no longer using.
- It is important to swap processes out of memory only when they are idle, or more to the point, only when there are no pending I/O operations. (Otherwise the pending I/O operation could write into the wrong process's memory space.) The solution is to either swap only totally idle processes, or do I/O operations only into and out of OS buffers, which are then transferred to or from process's main memory as a second step.
- Most modern OSes no longer use swapping, because it is too slow and there are faster alternatives available. (e.g. Paging.) However some UNIX systems will still invoke swapping if the system gets extremely full, and then discontinue swapping when the load reduces again. Windows 3.1 would use a modified version of swapping that was somewhat controlled by the user, swapping process's out if necessary and then only swapping them back in when the user focused on that particular window.
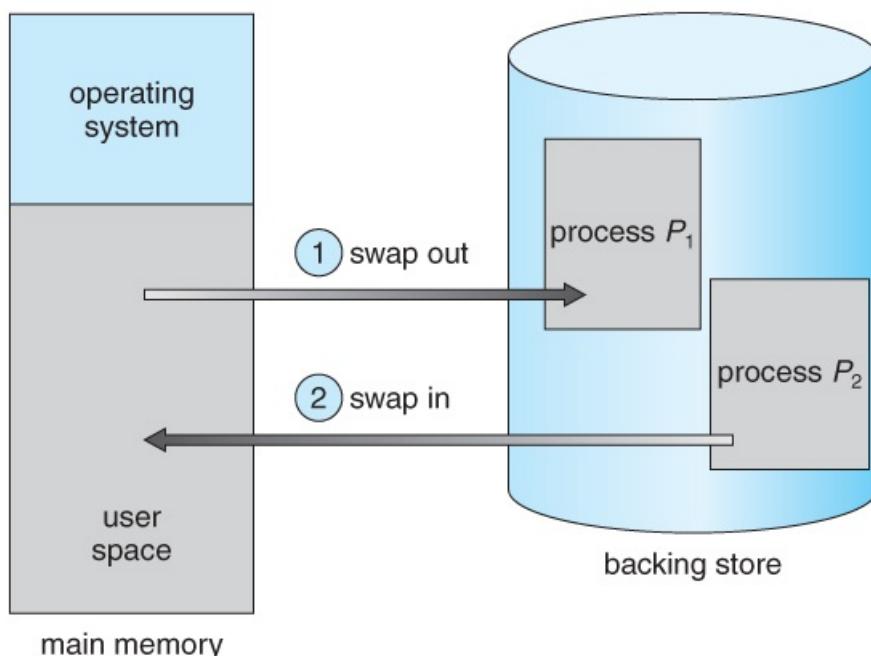
Figure 8.5 - Swapping of two processes using a disk as a backing store

8.2.2 Swapping on Mobile Systems (New Section in 9th Edition)

- Swapping is typically not supported on mobile platforms, for several reasons:
  - Mobile devices typically use flash memory in place of more spacious hard drives for persistent storage, so there is not as much space available.
  - Flash memory can only be written to a limited number of times before it becomes unreliable.
  - The bandwidth to flash memory is also lower.
- Apple's IOS asks applications to voluntarily free up memory
  - Read-only data, e.g. code, is simply removed, and reloaded later if needed.
  - Modified data, e.g. the stack, is never removed, but . . .
  - Apps that fail to free up sufficient memory can be removed by the OS
- Android follows a similar strategy.
  - Prior to terminating a process, Android writes its ***application state*** to flash memory for quick restarting.

8.3 Contiguous Memory Allocation

- One approach to memory management is to load each process into a contiguous space. The operating system is allocated space first, usually at either low or high memory locations, and then the remaining available memory is allocated to processes as needed. (The OS is usually loaded low, because that is where the interrupt vectors are located, but on older systems part of the OS was loaded high to make more room in low memory (within the 640K barrier) for user processes.)

8.3.1 Memory Protection (was Memory Mapping and Protection)

- The system shown in Figure 8.6 below allows protection against user programs accessing areas that they should not, allows programs to be relocated to different memory starting addresses as needed, and allows the memory space devoted to the OS to grow or shrink dynamically as needs change.
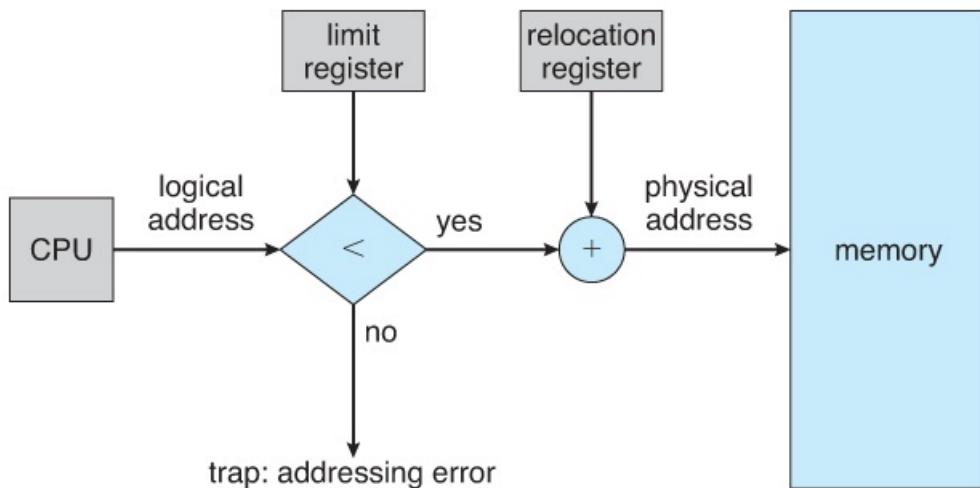
Figure 8.6 - Hardware support for relocation and limit registers

8.3.2 Memory Allocation

- One method of allocating contiguous memory is to divide all available memory into equal sized partitions, and to assign each process to their own partition. This restricts both the number of simultaneous processes and the maximum size of each process, and is no longer used.
- An alternate approach is to keep a list of unused (free) memory blocks (holes), and to find a hole of a suitable size whenever a process needs to be loaded into memory. There are many different strategies for finding the "best" allocation of memory to processes, including the three most commonly discussed:
    1. **First fit** - Search the list of holes until one is found that is big enough to satisfy the request, and assign a portion of that hole to that process. Whatever fraction of the hole not needed by the request is left on the free list as a smaller hole. Subsequent requests may start looking either from the beginning of the list or from the point at which this search ended.
    2. **Best fit** - Allocate the *smallest* hole that is big enough to satisfy the request. This saves large holes for other process requests that may need them later, but the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted. Keeping the free list sorted can speed up the process of finding the right hole.
    3. **Worst fit** - Allocate the largest hole available, thereby increasing the likelihood that the remaining portion will be usable for satisfying future requests.
- Simulations show that either first or best fit are better than worst fit in terms of both time and storage utilization. First and best fits are about equal in terms of storage utilization, but first fit is faster.

8.3.3. Fragmentation

- All the memory allocation strategies suffer from **external fragmentation**, though first and best fits experience the problems more so than worst fit. External fragmentation means that the available memory is broken up into lots of little pieces, none of which is big enough to satisfy the next memory requirement, although the sum total could.
- The amount of memory lost to fragmentation may vary with algorithm, usage patterns, and some design decisions such as which end of a hole to allocate and which end to save on the free list.
- Statistical analysis of first fit, for example, shows that for N blocks of allocated memory, another 0.5 N will be lost to fragmentation.

- ***Internal fragmentation*** also occurs, with all memory allocation strategies. This is caused by the fact that memory is allocated in blocks of a fixed size, whereas the actual memory needed will rarely be that exact size. For a random distribution of memory requests, on the average 1/2 block will be wasted per memory request, because on the average the last allocated block will be only half full.
    - Note that the same effect happens with hard drives, and that modern hardware gives us increasingly larger drives and memory at the expense of ever larger block sizes, which translates to more memory lost to internal fragmentation.
    - Some systems use variable size blocks to minimize losses due to internal fragmentation.
- If the programs in memory are relocatable, (using execution-time address binding), then the external fragmentation problem can be reduced via ***compaction***, i.e. moving all processes down to one end of physical memory. This only involves updating the relocation register for each process, as all internal work is done using logical addresses.
- Another solution as we will see in upcoming sections is to allow processes to use non-contiguous blocks of physical memory, with a separate relocation register for each block.

8.4 Segmentation

8.4.1 Basic Method

- Most users (programmers) do not think of their programs as existing in one continuous linear address space.
- Rather they tend to think of their memory in multiple ***segments***, each dedicated to a particular use, such as code, data, the stack, the heap, etc.
- Memory ***segmentation*** supports this view by providing addresses with a segment number (mapped to a segment base address) and an offset from the beginning of that segment.
- For example, a C compiler might generate 5 segments for the user code, library code, global (static) variables, the stack, and the heap, as shown in Figure 8.7:
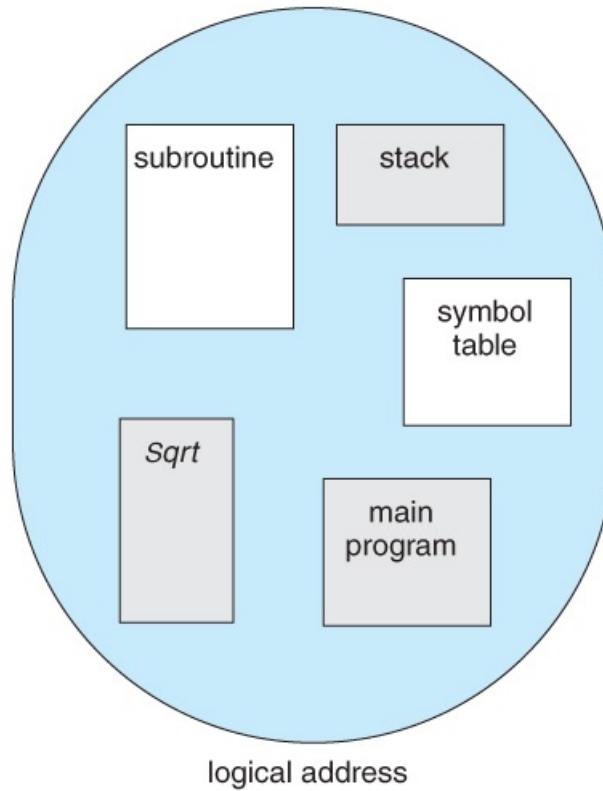
Figure 8.7 Programmer's view of a program.

8.4.2 Segmentation Hardware

- A ***segment table*** maps segment-offset addresses to physical addresses, and simultaneously checks for invalid addresses, using a system similar to the page tables and relocation base registers discussed previously. (Note that at this point in the discussion of segmentation, each segment is kept in contiguous memory and may be of different sizes, but that segmentation can also be combined with paging as we shall

see shortly.)
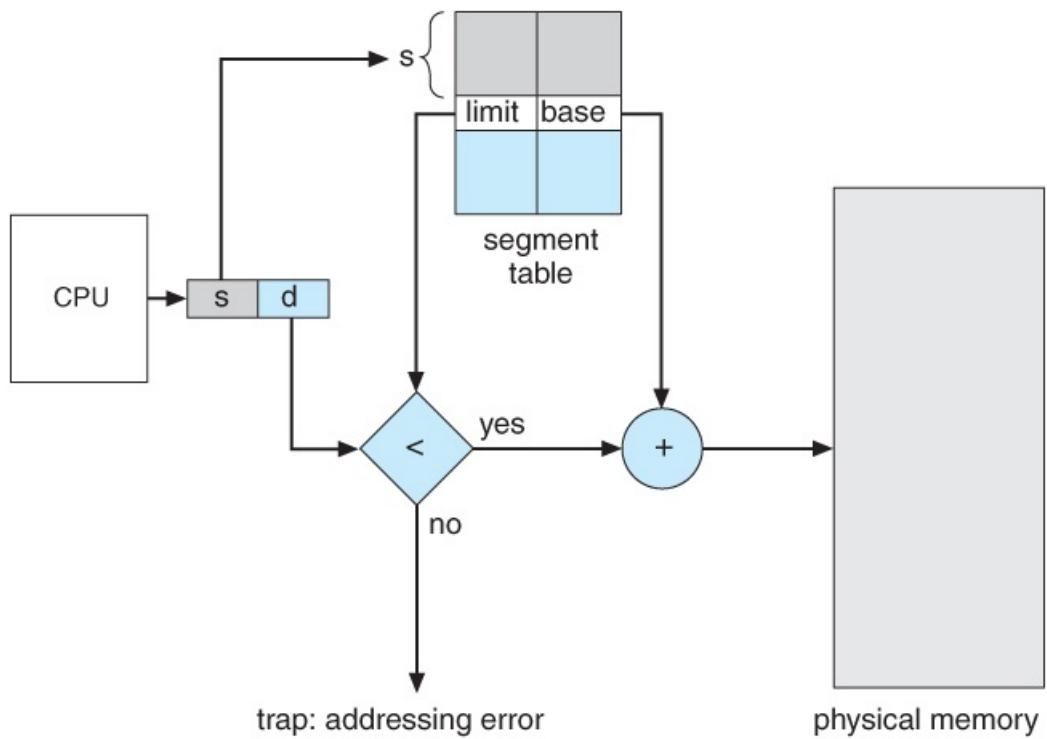
Figure 8.8 - Segmentation hardware
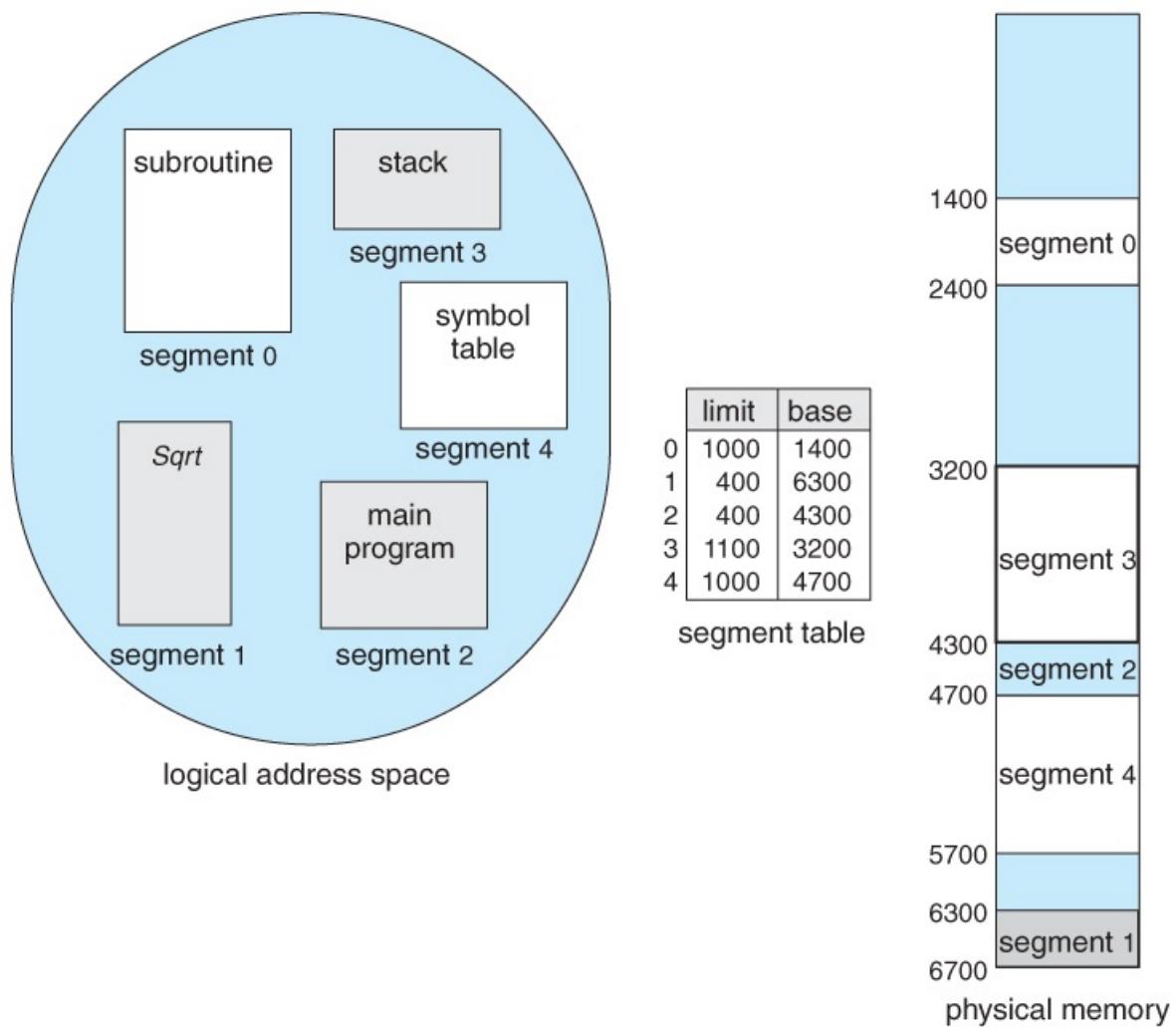
Figure 8.9 - Example of segmentation

8.5 Paging

- Paging is a memory management scheme that allows processes physical memory to be discontinuous, and which

eliminates problems with fragmentation by allocating memory in equal sized blocks known as **pages**.
- Paging eliminates most of the problems of the other methods discussed previously, and is the predominant memory management technique used today.

8.5.1 Basic Method

- The basic idea behind paging is to divide physical memory into a number of equal sized blocks called **frames**, and to divide a programs logical memory space into blocks of the same size called **pages.**
- Any page (from any process) can be placed into any available frame.
- The **page table** is used to look up what frame a particular page is stored in at the moment. In the following example, for instance, page 2 of the program's logical memory is currently stored in frame 3 of physical memory:
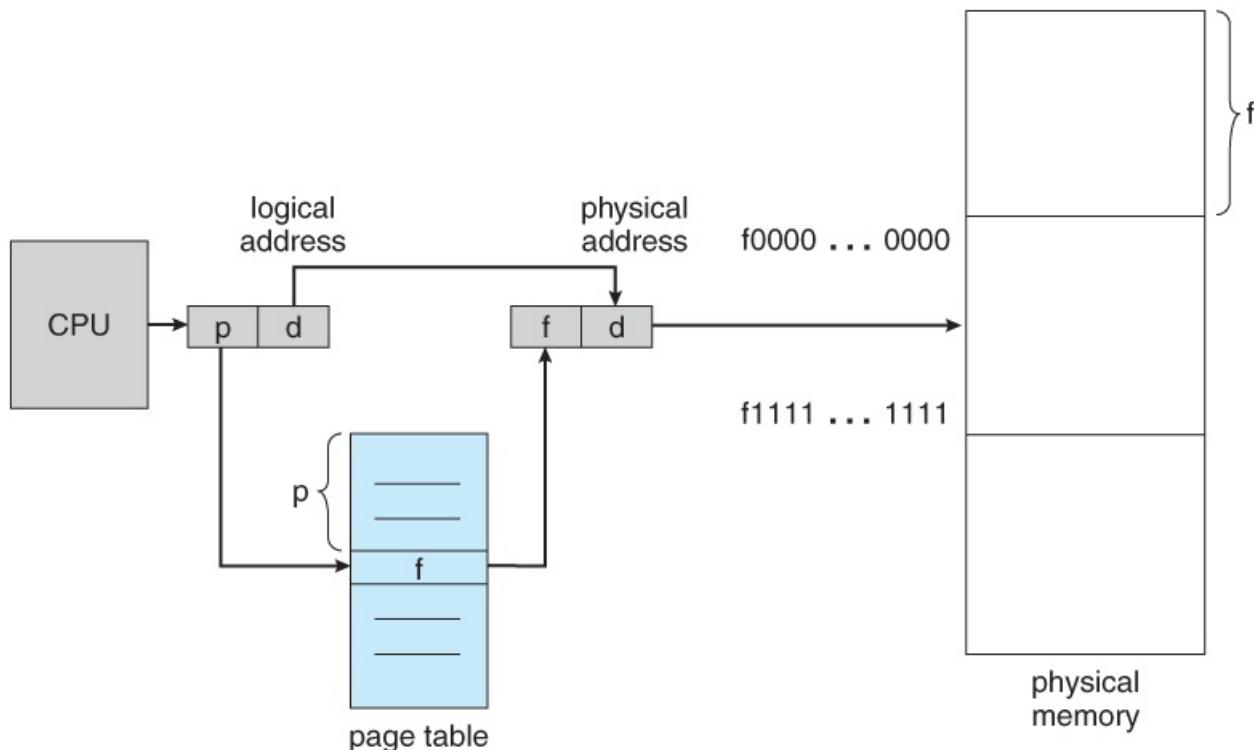
Figure 8.10 - Paging hardware
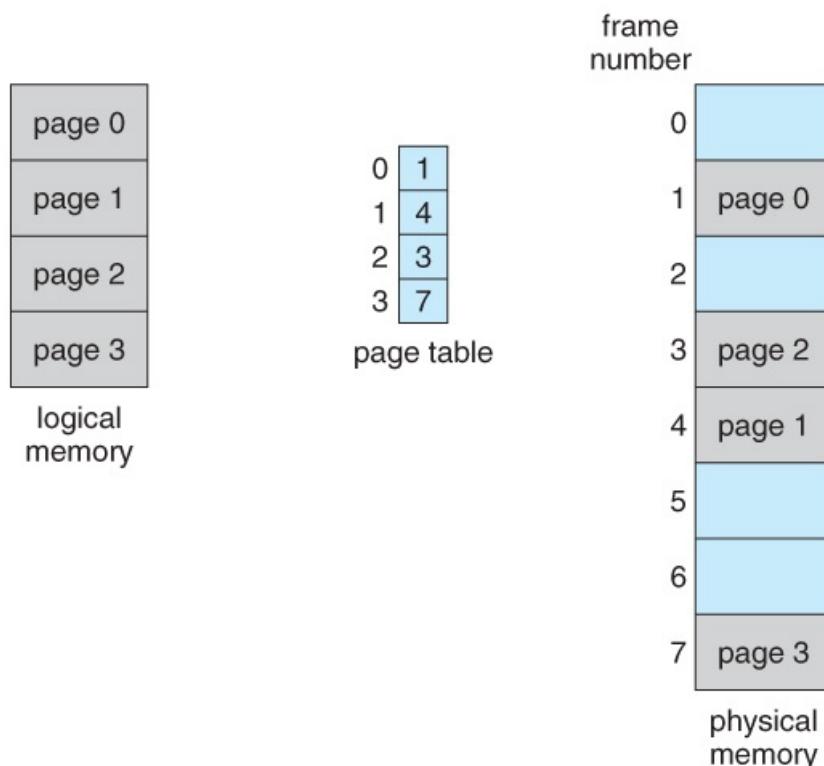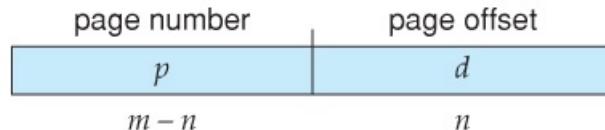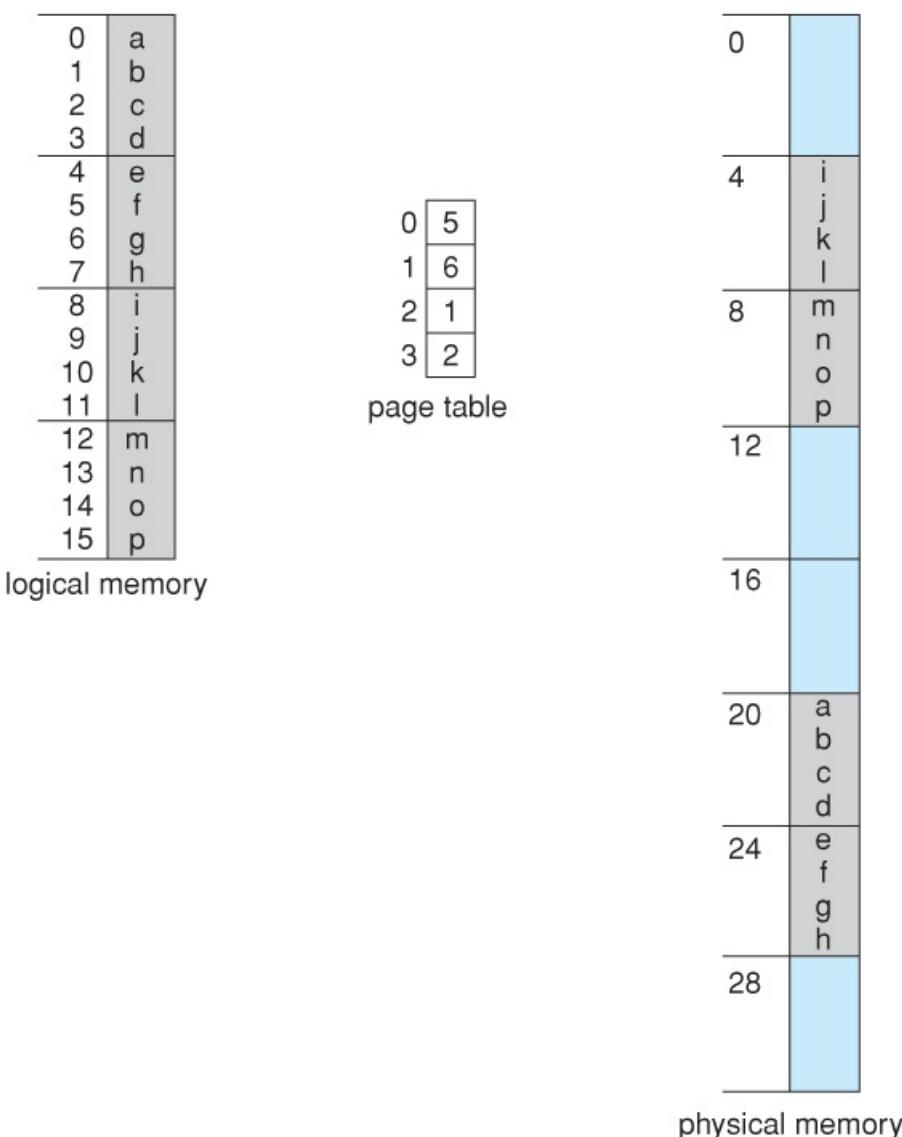
Figure 8.11 - Paging model of logical and physical memory

- A logical address consists of two parts: A page number in which the address resides, and an offset from the beginning of that page. (The number of bits in the page number limits how many pages a single process can address. The number of bits in the offset determines the maximum size of each page, and should correspond to the system frame size.)
- The page table maps the page number to a frame number, to yield a physical address which also has two parts: The frame number and the offset within that frame. The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame.
- Page numbers, frame numbers, and frame sizes are determined by the architecture, but are typically powers of two, allowing addresses to be split at a certain number of bits. For example, if the logical address size is 2^m and the page size is 2^n, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.
- Note also that the number of bits in the page number and the number of bits in the frame number do not have to be identical. The former determines the address range of the logical address space, and the latter relates to the physical address space.

page number	page offset
p	d
$m - n$	n
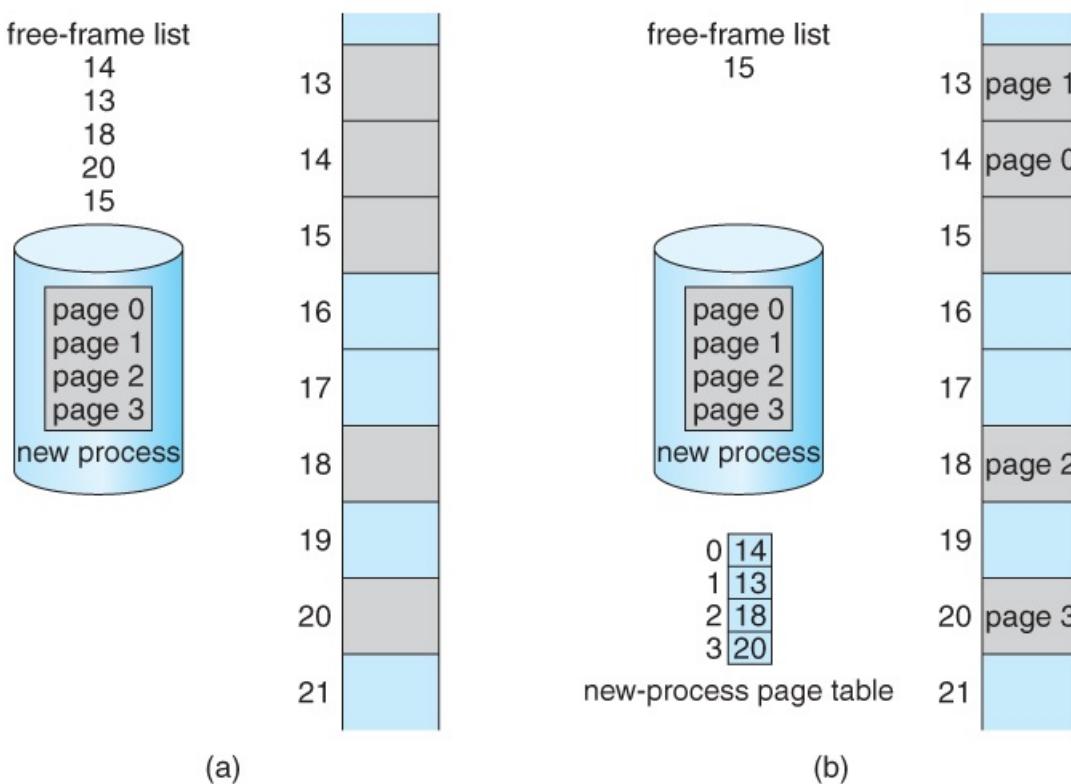
- (DOS used to use an addressing scheme with 16 bit frame numbers and 16-bit offsets, on hardware that only supported 24-bit hardware addresses. The result was a resolution of starting frame addresses finer than the size of a single frame, and multiple frame-offset combinations that mapped to the same physical hardware address.)
- Consider the following micro example, in which a process has 16 bytes of logical memory, mapped in 4 byte pages into 32 bytes of physical memory. (Presumably some other processes would be consuming the remaining 16 bytes of physical memory.)

Figure 8.12 - Paging example for a 32-byte memory with 4-byte pages

- Note that paging is like having a table of relocation registers, one for each page of the logical memory.
- There is no external fragmentation with paging. All blocks of physical memory are used, and there are no gaps in between and no problems with finding the right sized hole for a particular chunk of memory.
- There is, however, internal fragmentation. Memory is allocated in chunks the size of a page, and on the average, the last page will only be half full, wasting on the average half a page of memory per process. (Possibly more, if processes keep their code and data in separate pages.)
- Larger page sizes waste more memory, but are more efficient in terms of overhead. Modern trends have been to increase page sizes, and some systems even have multiple size pages to try and make the best of both worlds.
- Page table entries (frame numbers) are typically 32 bit numbers, allowing access to 2^32 physical page frames. If those frames are 4 KB in size each, that translates to 16 TB of addressable physical memory. (32 + 12 = 44 bits of physical address space.)
- When a process requests memory (e.g. when its code is loaded in from disk), free frames are allocated from a free-frame list, and inserted into that process's page table.
- Processes are blocked from accessing anyone else's memory because all of their memory requests are mapped through their page table. There is no way for them to generate an address that maps into any other process's memory space.
- The operating system must keep track of each individual process's page table, updating it whenever the process's pages get moved in and out of memory, and applying the correct page table when processing system calls for a particular process. This all increases the overhead involved when swapping processes in and out of the CPU. (The currently active page table must be updated to reflect the process that is currently running.)

Figure 8.13 - Free frames (a) before allocation and (b) after allocation

8.5.2 Hardware Support

- Page lookups must be done for every memory reference, and whenever a process gets swapped in or out of the CPU, its page table must be swapped in and out too, along with the instruction registers, etc. It is therefore appropriate to provide hardware support for this operation, in order to make it as fast as possible and to make process switches as fast as possible also.
- One option is to use a set of registers for the page table. For example, the DEC PDP-11 uses 16-bit addressing and 8 KB pages, resulting in only 8 pages per process. (It takes 13 bits to address 8 KB of offset, leaving only 3 bits to define a page number.)
- An alternate option is to store the page table in main memory, and to use a single register (called the **page-table base register, PTBR**) to record where in memory the page table is located.
  - Process switching is fast, because only the single register needs to be changed.
  - However memory access just got half as fast, because every memory access now requires **two**

memory accesses - One to fetch the frame number from memory and then another one to access the desired memory location.

- The solution to this problem is to use a very special high-speed memory device called the **translation look-aside buffer, TLB.**
  - The benefit of the TLB is that it can search an entire table for a key value in parallel, and if it is found anywhere in the table, then the corresponding lookup value is returned.
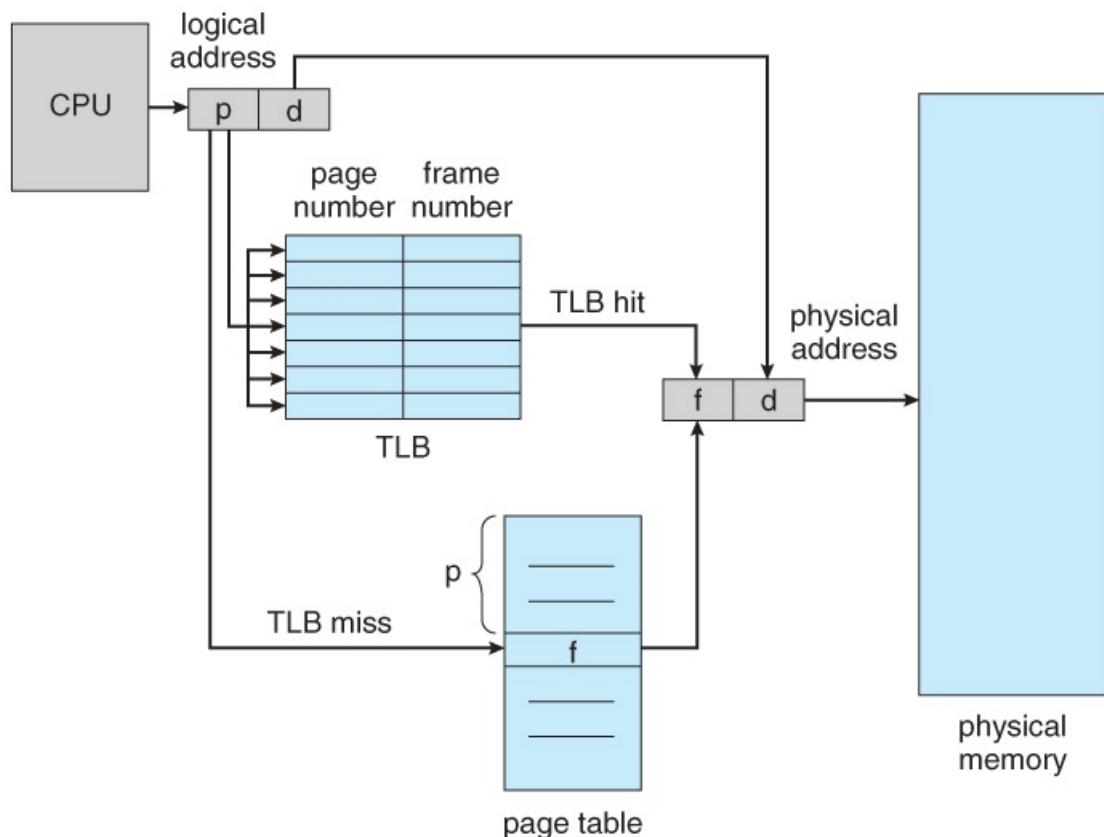
Figure 8.14 - Paging hardware with TLB

- The TLB is very expensive, however, and therefore very small. (Not large enough to hold the entire page table.) It is therefore used as a cache device.
  - Addresses are first checked against the TLB, and if the info is not there (a TLB miss), then the frame is looked up from main memory and the TLB is updated.
  - If the TLB is full, then replacement strategies range from **least-recently used, LRU** to random.
  - Some TLBs allow some entries to be **wired down**, which means that they cannot be removed from the TLB. Typically these would be kernel frames.
  - Some TLBs store **address-space identifiers, ASIDs**, to keep track of which process "owns" a particular entry in the TLB. This allows entries from multiple processes to be stored simultaneously in the TLB without granting one process access to some other process's memory location. Without this feature the TLB has to be flushed clean with every process switch.
- The percentage of time that the desired information is found in the TLB is termed the **hit ratio**.
- **(Eighth Edition Version:)** For example, suppose that it takes 100 nanoseconds to access main memory, and only 20 nanoseconds to search the TLB. So a TLB hit takes 120 nanoseconds total (20 to find the frame number and then another 100 to go get the data), and a TLB miss takes 220 (20 to search the TLB, 100 to go get the frame number, and then another 100 to go get the data.) So with an 80% TLB hit ratio, the average memory access time would be:

$$0.80 * 120 + 0.20 * 220 = 140 \text{ nanoseconds}$$

for a 40% slowdown to get the frame number. A 98% hit rate would yield 122 nanoseconds average access time (you should verify this), for a 22% slowdown.

- **(Ninth Edition Version:)** The ninth edition ignores the 20 nanoseconds required to search the TLB, yielding

$$0.80 * 100 + 0.20 * 200 = 120 \text{ nanoseconds}$$

for a 20% slowdown to get the frame number. A 99% hit rate would yield 101 nanoseconds average access time (you should verify this), for a 1% slowdown.

8.5.3 Protection

- The page table can also help to protect processes from accessing memory that they shouldn't, or their own memory in ways that they shouldn't.
- A bit or bits can be added to the page table to classify a page as read-write, read-only, read-write-execute, or some combination of these sorts of things. Then each memory reference can be checked to ensure it is accessing the memory in the appropriate mode.
- Valid / invalid bits can be added to "mask off" entries in the page table that are not in use by the current process, as shown by example in Figure 8.12 below.
- Note that the valid / invalid bits described above cannot block all illegal memory accesses, due to the internal fragmentation. (Areas of memory in the last page that are not entirely filled by the process, and may contain data left over by whoever used that frame last.)
- Many processes do not use all of the page table available to them, particularly in modern systems with very large potential page tables. Rather than waste memory by creating a full-size page table for every process, some systems use a **page-table length register, PTLR**, to specify the length of the page table.
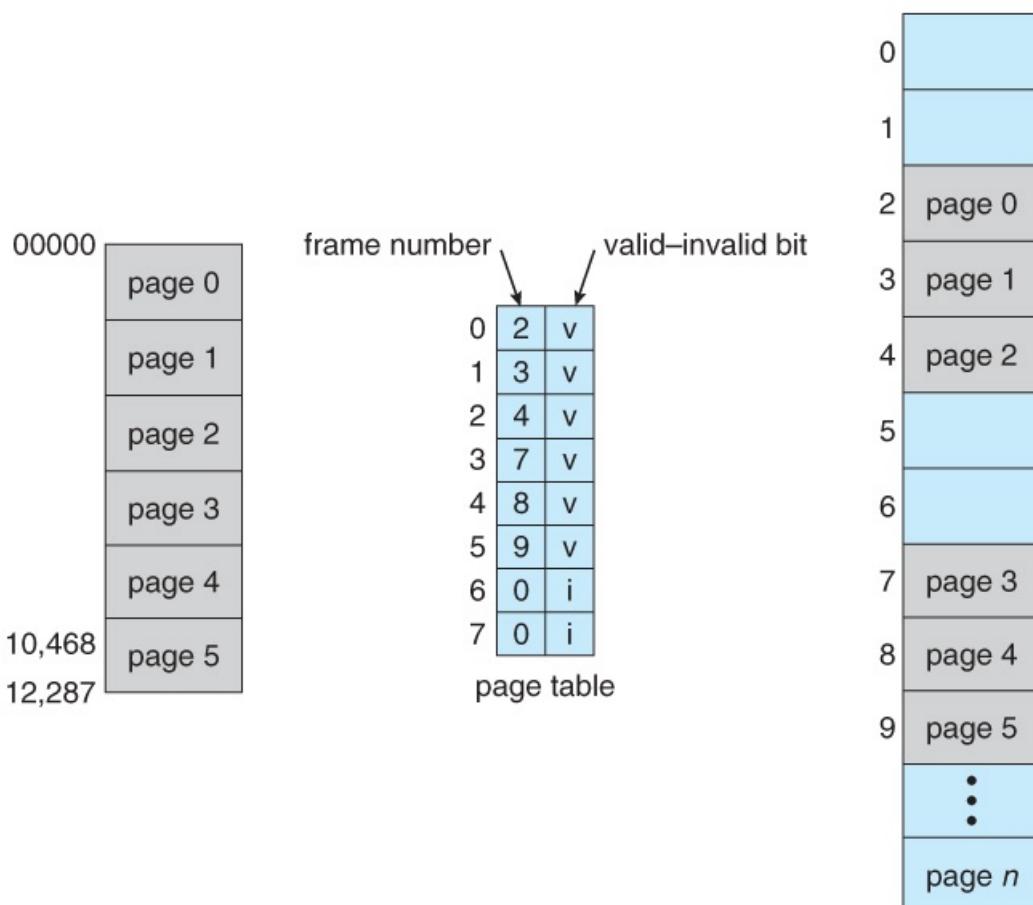
Figure 8.15 - Valid (v) or invalid (i) bit in page table

8.5.4 Shared Pages

- Paging systems can make it very easy to share blocks of memory, by simply duplicating page numbers in multiple page frames. This may be done with either code or data.
- If code is **reentrant**, that means that it does not write to or change the code in any way (it is non self-modifying), and it is therefore safe to re-enter it. More importantly, it means the code can be shared by multiple processes, so long as each has their own copy of the data and registers, including the instruction register.
- In the example given below, three different users are running the editor simultaneously, but the code is only loaded into memory (in the page frames) one time.
- Some systems also implement shared memory in this fashion.
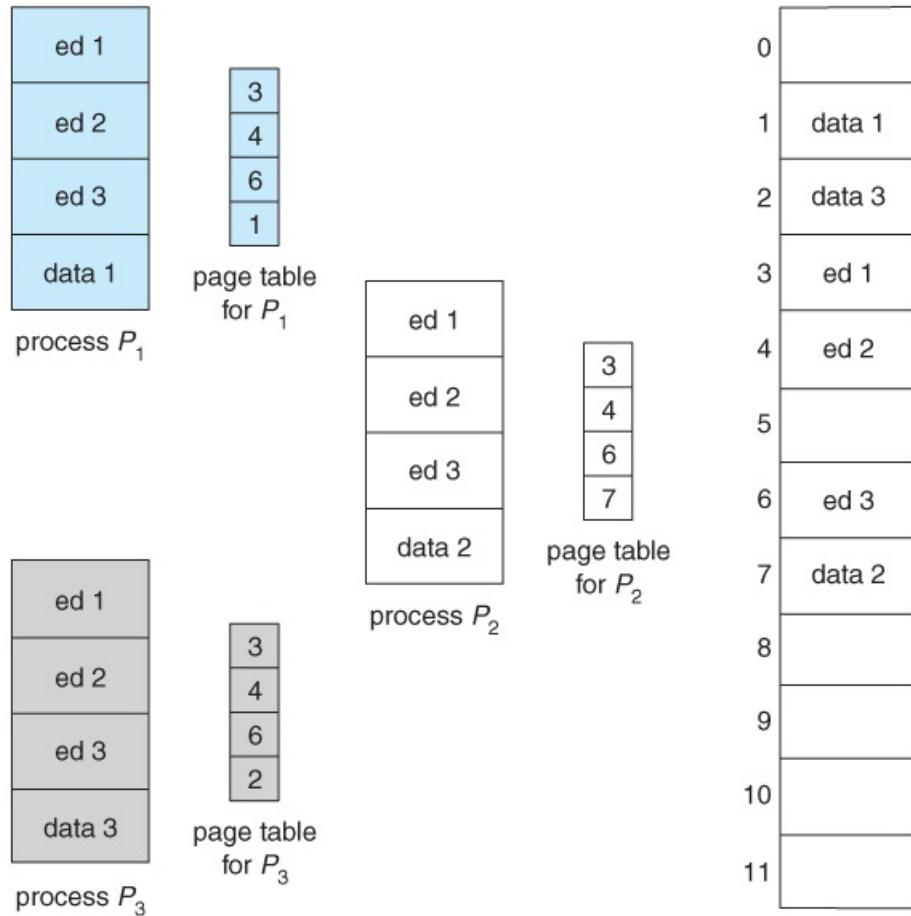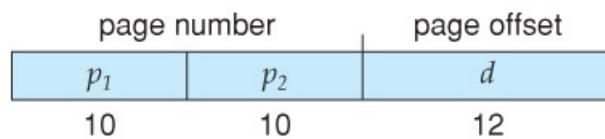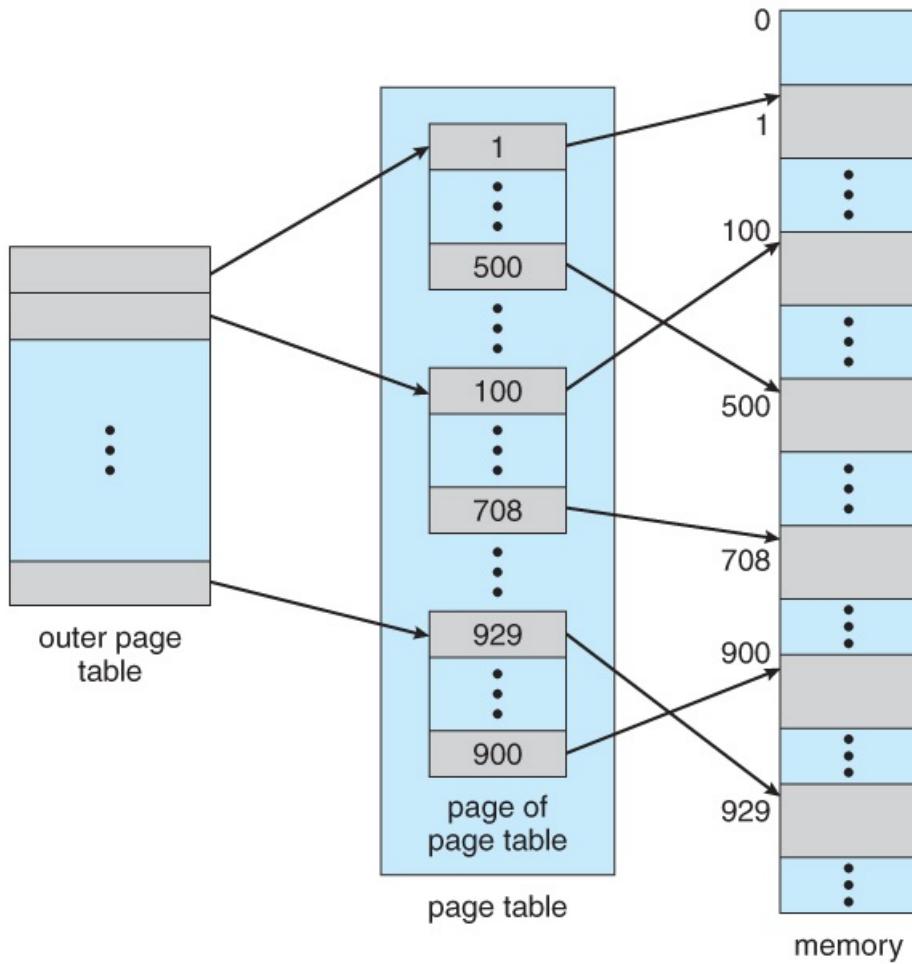
Figure 8.16 - Sharing of code in a paging environment

8.6 Structure of the Page Table
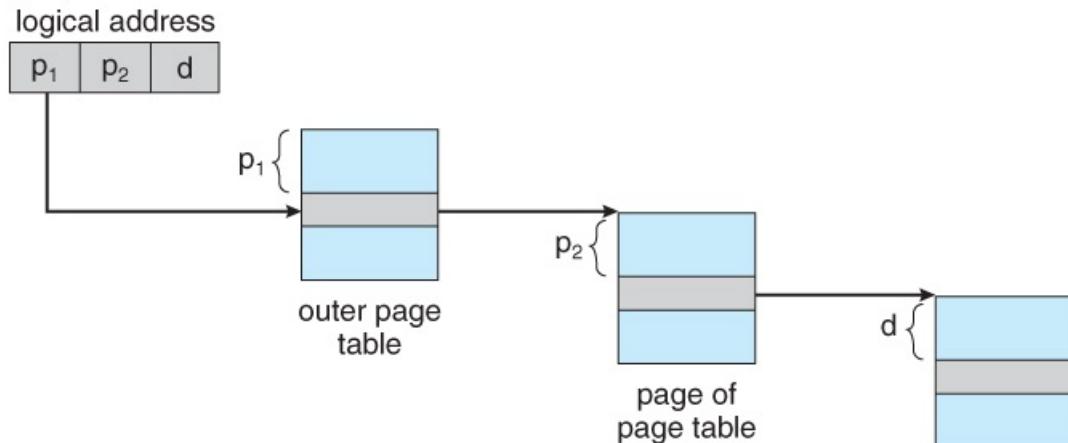
8.6.1 Hierarchical Paging

- Most modern computer systems support logical address spaces of 2^32 to 2^64.
- With a 2^32 address space and 4K (2^12) page sizes, this leave 2^20 entries in the page table. At 4 bytes per entry, this amounts to a 4 MB page table, which is too large to reasonably keep in contiguous memory. (And to swap in and out of memory with each process switch.) Note that with 4K pages, this would take 1024 pages just to hold the page table!
- One option is to use a two-tier paging system, i.e. to page the page table.
- For example, the 20 bits described above could be broken down into two 10-bit page numbers. The first identifies an entry in the outer page table, which identifies where in memory to find one page of an inner page table. The second 10 bits finds a specific entry in that inner page table, which in turn identifies a particular frame in physical memory. (The remaining 12 bits of the 32 bit logical address are the offset within the 4K frame.)
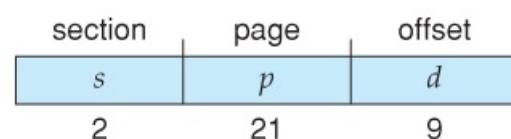
Figure 8.17 A two-level page-table scheme

Figure 8.18 - Address translation for a two-level 32-bit paging architecture

- VAX Architecture divides 32-bit addresses into 4 equal sized sections, and each page is 512 bytes, yielding an address form of:

section	page	offset
s	p	d
2	21	9

- With a 64-bit logical address space and 4K pages, there are 52 bits worth of page numbers, which is still too many even for two-level paging. One could increase the paging level, but with 10-bit page tables it would take 7 levels of indirection, which would be prohibitively slow memory access. So some other approach must be used.

outer page	inner page	offset
p_1	p_2	d
42	10	12

64-bits Two-tiered leaves 42 bits in outer table

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Going to a fourth level still leaves 32 bits in the outer table.

8.6.2 Hashed Page Tables

- One common data structure for accessing data that is sparsely distributed over a broad range of possible values is with **hash tables**. Figure 8.16 below illustrates a **hashed page table** using chain-and-bucket hashing:
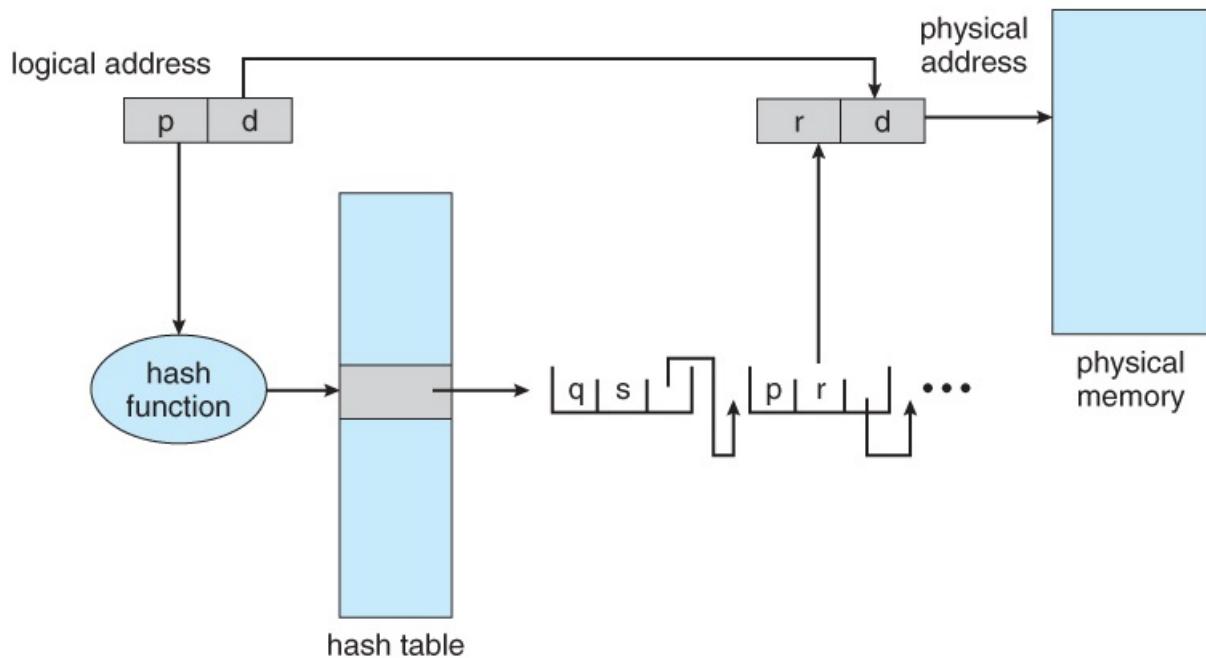
Figure 8.19 - Hashed page table

8.6.3 Inverted Page Tables

- Another approach is to use an **inverted page table**. Instead of a table listing all of the pages for a particular process, an inverted page table lists all of the pages currently loaded in memory, for all processes. (I.e. there is one entry per **frame** instead of one entry per **page**.)
- Access to an inverted page table can be slow, as it may be necessary to search the entire table in order to find the desired page (or to discover that it is not there.) Hashing the table can help speedup the search process.
- Inverted page tables prohibit the normal method of implementing shared memory, which is to map multiple logical pages to a common physical frame. (Because each frame is now mapped to one and only one process.)
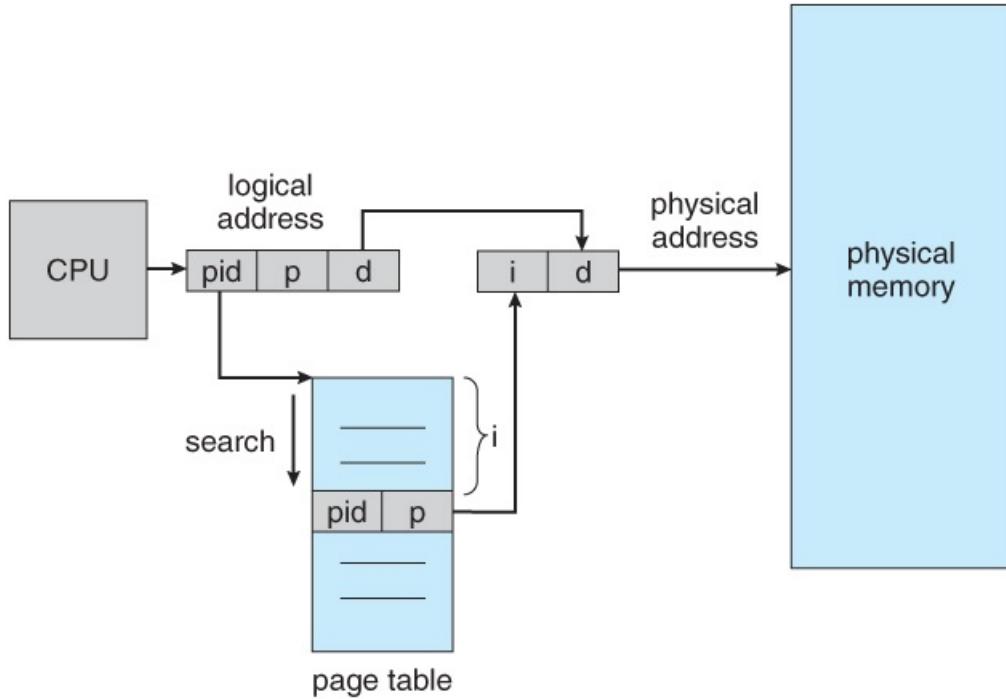
Figure 8.20 - Inverted page table

8.6.4 Oracle SPARC Solaris (Optional, New Section in 9th Edition)

8.7 Example: Intel 32 and 64-bit Architectures (Optional)

8.7.1.1 IA-32 Segmentation

- The Pentium CPU provides both pure segmentation and segmentation with paging. In the latter case, the CPU generates a logical address (segment-offset pair), which the segmentation unit converts into a logical linear address, which in turn is mapped to a physical frame by the paging unit, as shown in Figure 8.21:
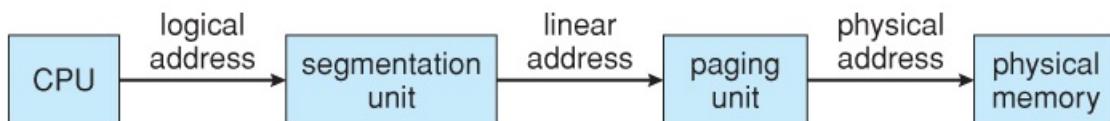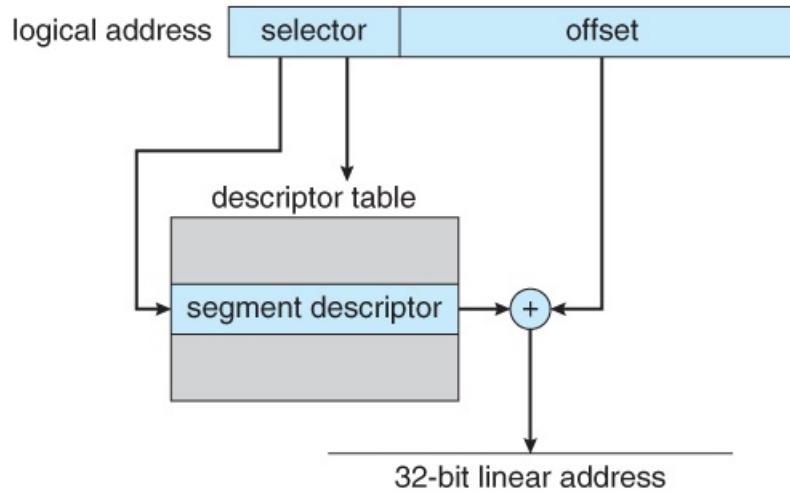
Figure 8.21 - Logical to physical address translation in IA-32

8.7.1.1 IA-32 Segmentation

- The Pentium architecture allows segments to be as large as 4 GB, (24 bits of offset).
- Processes can have as many as 16K segments, divided into two 8K groups:
  - 8K private to that particular process, stored in the **Local Descriptor Table, LDT.**
  - 8K shared among all processes, stored in the **Global Descriptor Table, GDT.**
- Logical addresses are (selector, offset) pairs, where the selector is made up of 16 bits:
  - A 13 bit segment number (up to 8K)
  - A 1 bit flag for LDT vs. GDT.
  - 2 bits for protection codes.
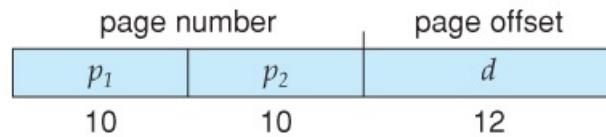
  - The descriptor tables contain 8-byte descriptions of each segment, including base and limit registers.
  - Logical linear addresses are generated by looking the selector up in the descriptor table and adding the appropriate base address to the offset, as shown in Figure 8.22:

Figure 8.22 - IA-32 segmentation

8.7.1.2 IA-32 Paging

- Pentium paging normally uses a two-tier paging scheme, with the first 10 bits being a page number for an outer page table (a.k.a. page directory), and the next 10 bits being a page number within one of the 1024 inner page tables, leaving the remaining 12 bits as an offset into a 4K page.

- A special bit in the page directory can indicate that this page is a 4MB page, in which case the remaining 22 bits are all used as offset and the inner tier of page tables is not used.
- The CR3 register points to the page directory for the current process, as shown in Figure 8.23 below.
- If the inner page table is currently swapped out to disk, then the page directory will have an "invalid bit" set, and the remaining 31 bits provide information on where to find the swapped out page table on the disk.
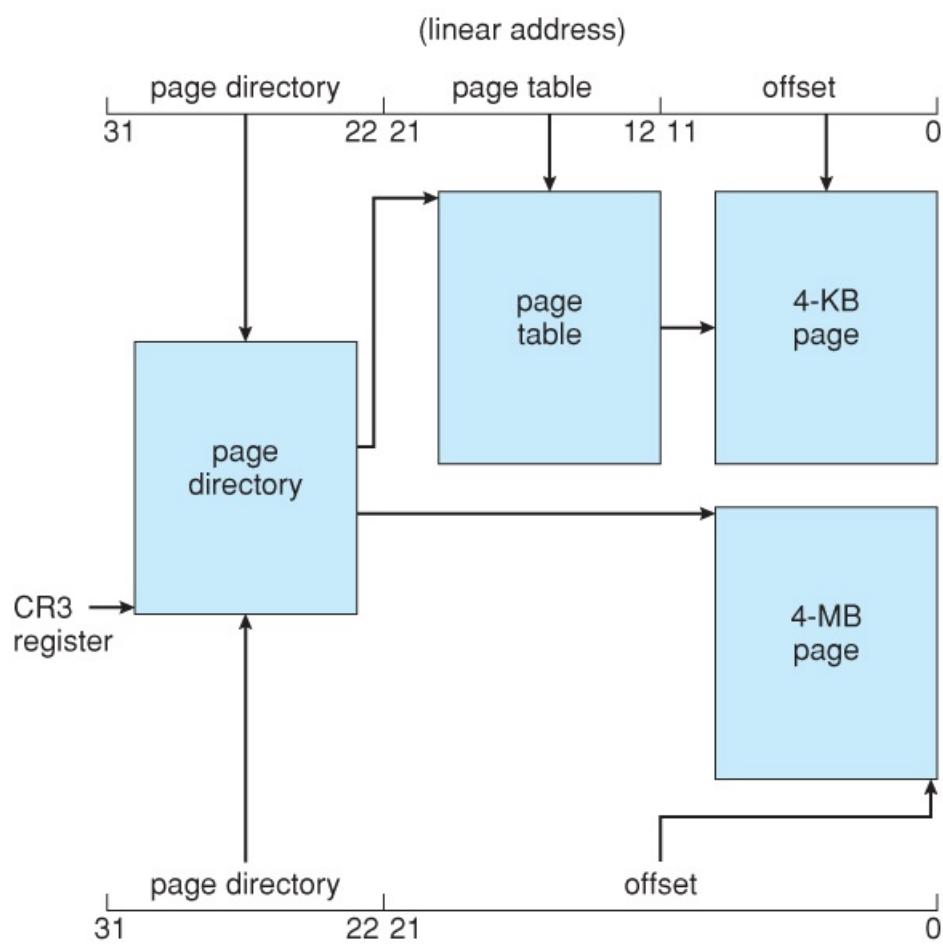
Figure 8.23 - Paging in the IA-32 architecture.
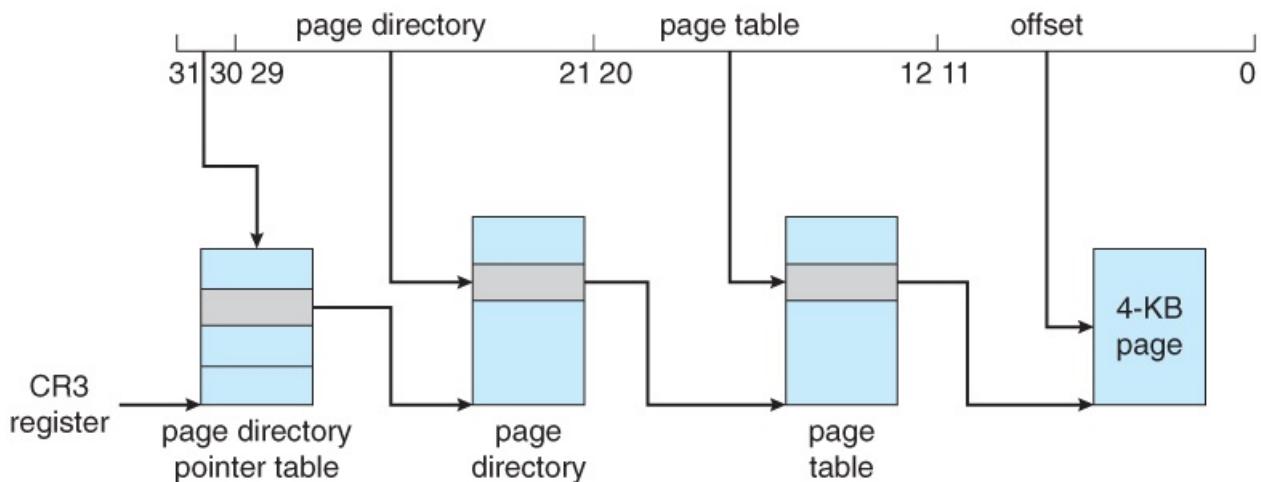
Figure 8.24 - Page address extensions.

8.7.2 x86-64
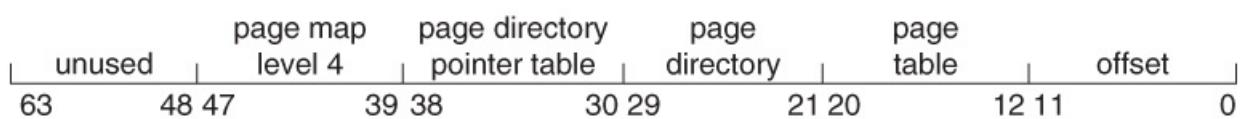
Figure 8.25 - x86-64 linear address.

8.8 Example: ARM Architecture (Optional)
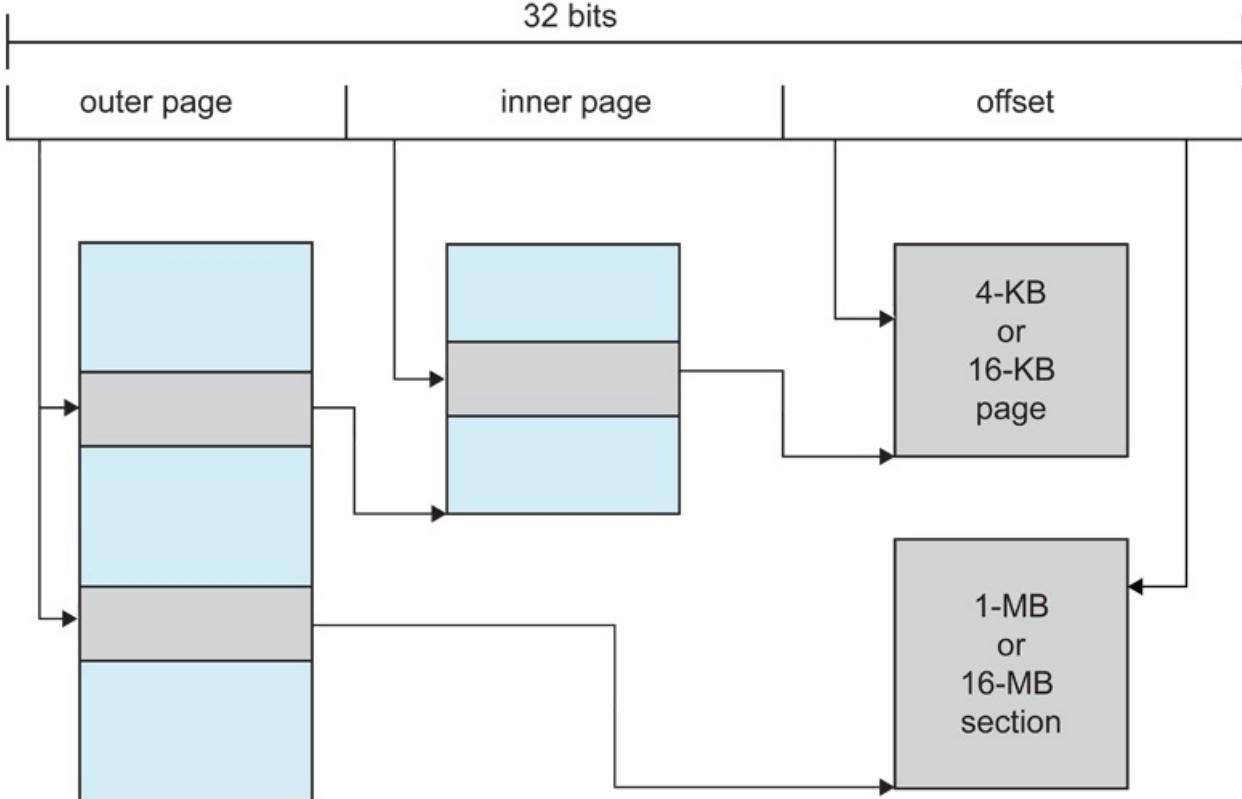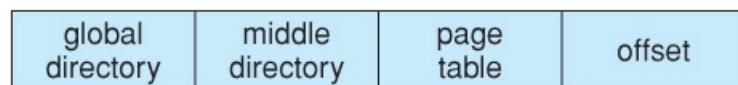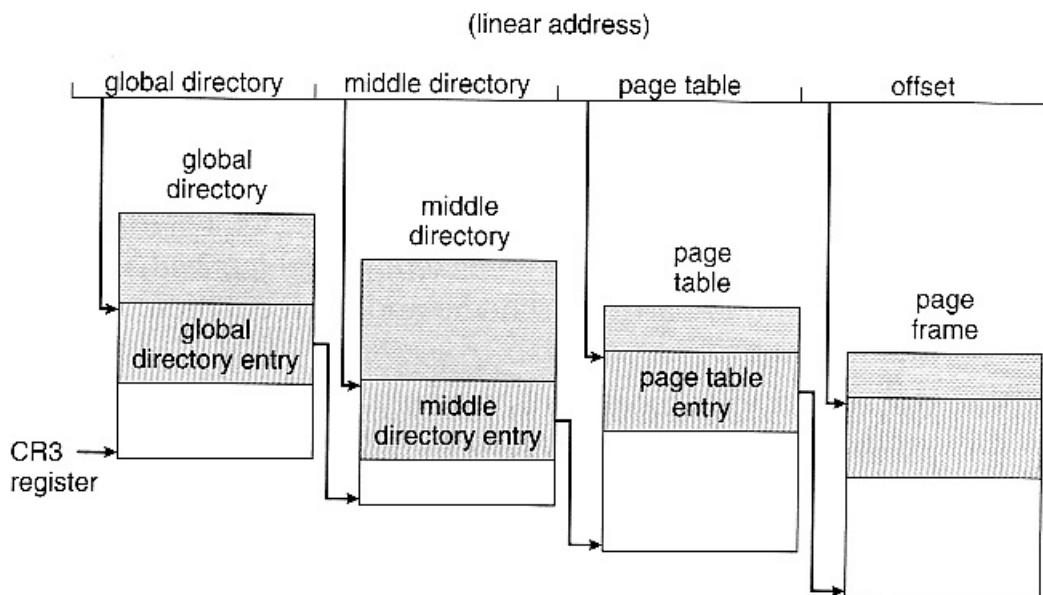
Figure 8.26 - Logical address translation in ARM.

Old 8.7.3 Linux on Pentium Systems - Omitted from the Ninth Edition

- Because Linux is designed for a wide variety of platforms, some of which offer only limited support for segmentation, Linux supports minimal segmentation. Specifically Linux uses only 6 segments:
    1. Kernel code.
    2. Kerned data.
    3. User code.
    4. User data.
    5. A task-state segment, TSS
    6. A default LDT segment
- All processes share the same user code and data segments, because all process share the same logical address space and all segment descriptors are stored in the Global Descriptor Table. (The LDT is generally not used.)
- Each process has its own TSS, whose descriptor is stored in the GDT. The TSS stores the hardware state of a process during context switches.
- The default LDT is shared by all processes and generally not used, but if a process needs to create its own LDT, it may do so, and use that instead of the default.
- The Pentium architecture provides 2 bits (4 values) for protection in a segment selector, but Linux only uses two values: user mode and kernel mode.
- Because Linux is designed to run on 64-bit as well as 32-bit architectures, it employs a three-level paging strategy as shown in Figure 8.24, where the number of bits in each portion of the address varies by architecture. In the case of the Pentium architecture, the size of the middle directory portion is set to 0 bits, effectively bypassing the middle directory.

Figure 8.24 Three-level paging in Linux.

8.8 Summary

- (For a fun and easy explanation of paging, you may want to read about [The Paging Game]().)

Virtual Memory

References:

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 9

9.1 Background

- Preceding sections talked about how to avoid memory fragmentation by breaking process memory requirements down into smaller bites (pages), and storing the pages non-contiguously in memory. However the entire process still had to be stored in memory somewhere.
- In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:
    1. Error handling code is not needed unless that specific error occurs, some of which are quite rare.
    2. Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
    3. Certain features of certain programs are rarely used, such as the routine to balance the federal budget. :-)
- The ability to load only the portions of processes that were actually needed (and only *when* they were needed) has several benefits:
    - Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer.
    - Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
    - Less I/O is needed for swapping processes in and out of RAM, speeding things up.
- Figure 9.1 shows the general layout of **virtual memory**, which can be much larger than physical memory:
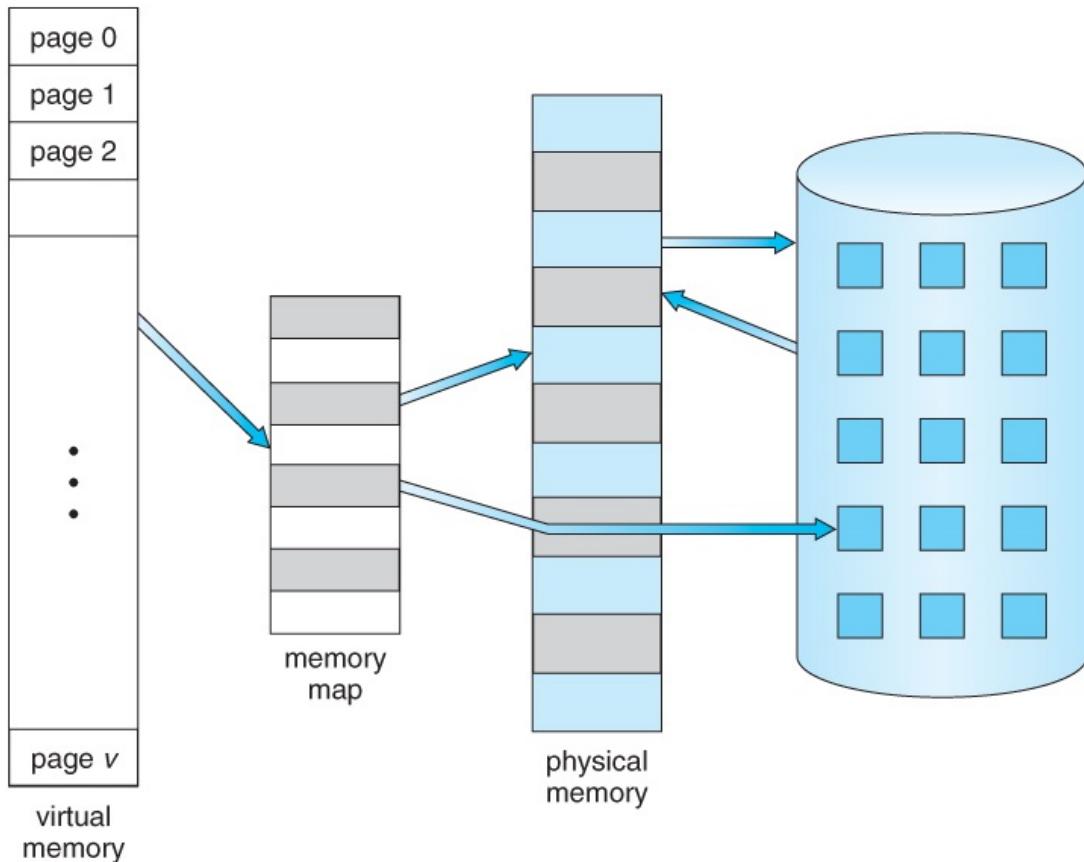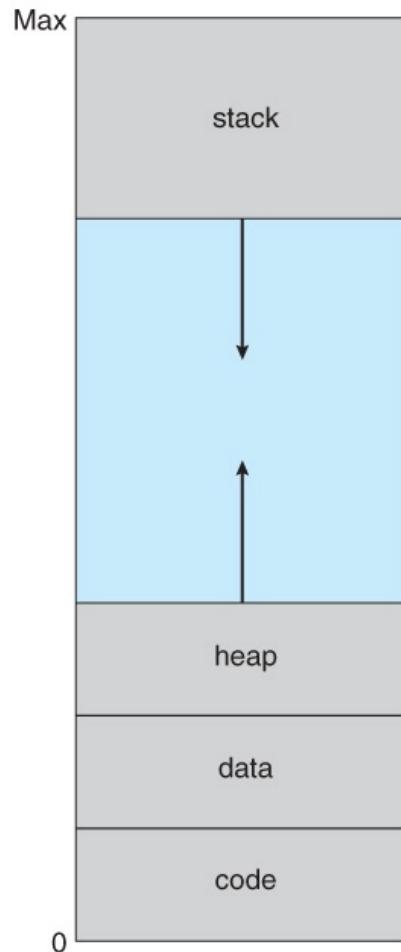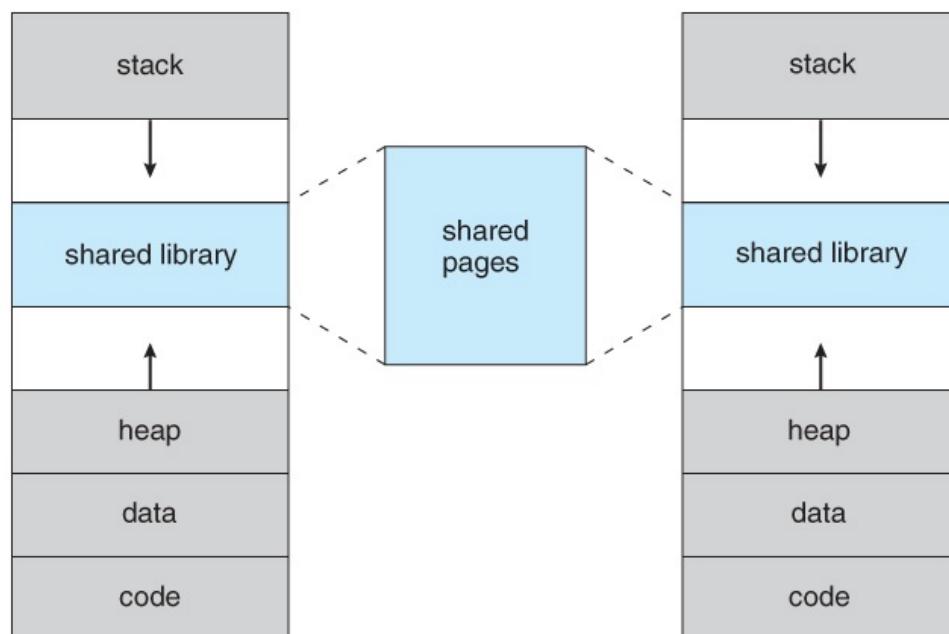
Figure 9.1 - Diagram showing virtual memory that is larger than physical memory

- Figure 9.2 shows **virtual address space**, which is the programmers logical view of process memory storage. The actual physical layout is controlled by the process's page table.
- Note that the address space shown in Figure 9.2 is **sparse** - A great hole in the middle of the address space is never used, unless the stack and/or the heap grow to fill the hole.

Figure 9.2 - Virtual address space

- Virtual memory also allows the sharing of files and memory by multiple processes, with several benefits:
  - System libraries can be shared by mapping them into the virtual address space of more than one process.
  - Processes can also share virtual memory by mapping the same block of memory to more than one process.
 - Process pages can be shared during a fork( ) system call, eliminating the need to copy all of the pages of the original (parent) process.

Figure 9.3 - Shared library using virtual memory

9.2 Demand Paging

- The basic idea behind ***demand paging*** is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them. (on demand.) This is termed a ***lazy swapper***, although a ***pager*** is a

more accurate term.
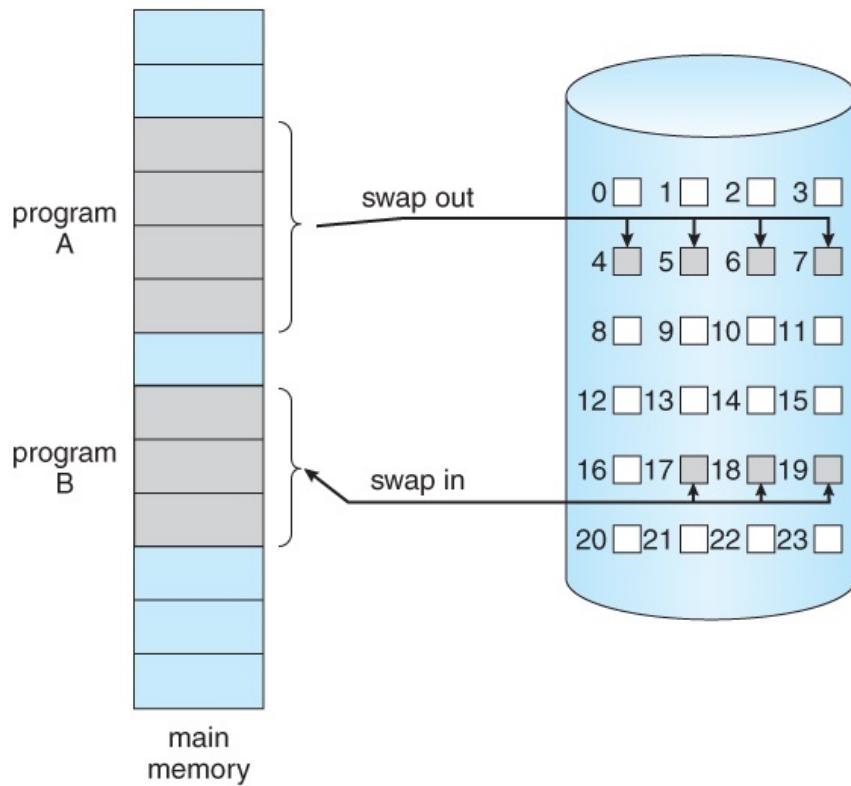
Figure 9.4 - Transfer of a paged memory to contiguous disk space

9.2.1 Basic Concepts

- The basic idea behind paging is that when a process is swapped in, the pager only loads into memory those pages that it expects the process to need (right away.)
- Pages that are not loaded into memory are marked as invalid in the page table, using the invalid bit. (The rest of the page table entry may either be blank or contain information about where to find the swapped-out page on the hard drive.)
- If the process only ever accesses pages that are loaded in memory (**memory resident** pages), then the process runs exactly as if all the pages were loaded in to memory.
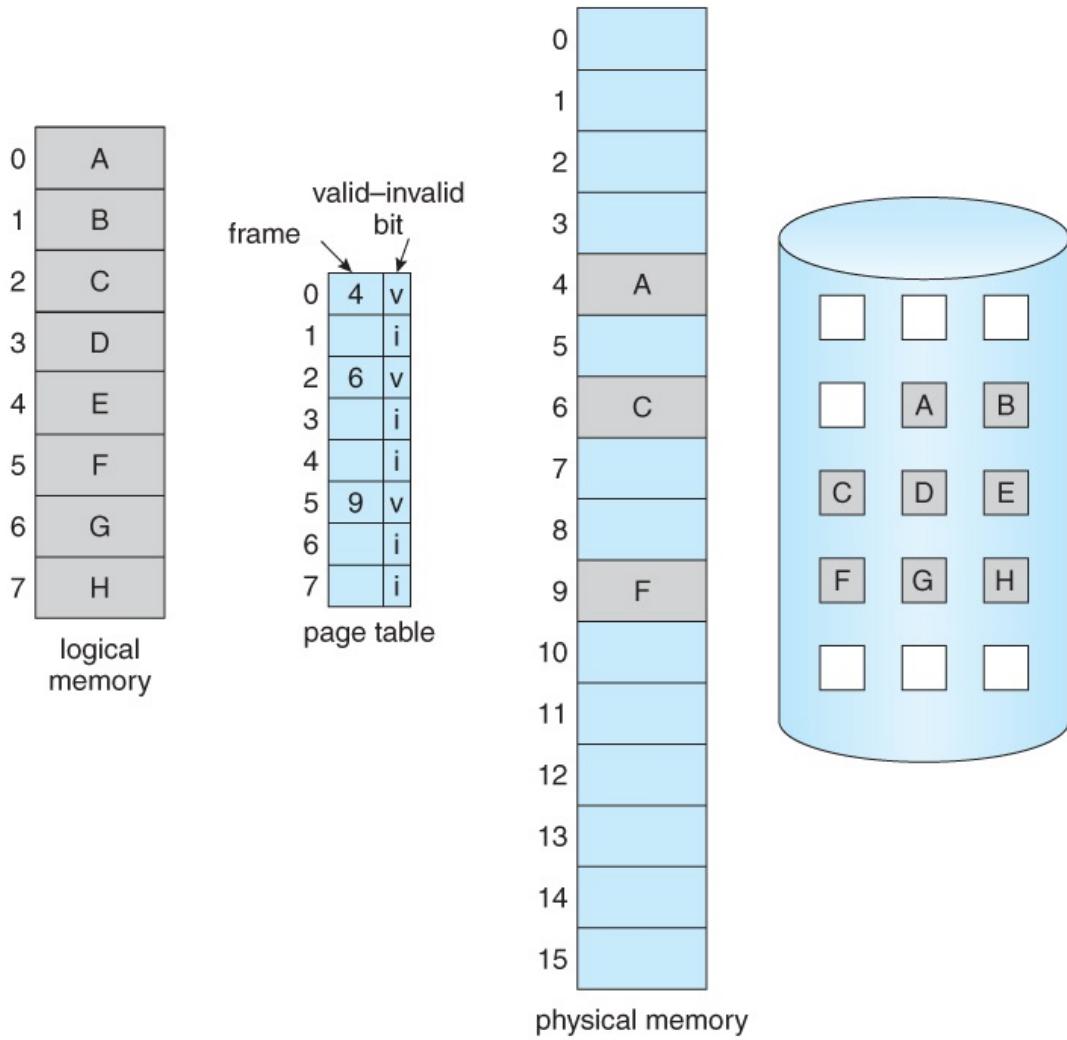
Figure 9.5 - Page table when some pages are not in main memory.

- On the other hand, if a page is needed that was not originally loaded up, then a ***page fault trap*** is generated, which must be handled in a series of steps:
  1. The memory address requested is first checked, to make sure it was a valid memory request.
  2. If the reference was invalid, the process is terminated. Otherwise, the page must be paged in.
  3. A free frame is located, possibly from a free-frame list.
  4. A disk operation is scheduled to bring in the necessary page from disk. (This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime.)
  5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
  6. The instruction that caused the page fault must now be restarted from the beginning, (as soon as this process gets another turn on the CPU.)
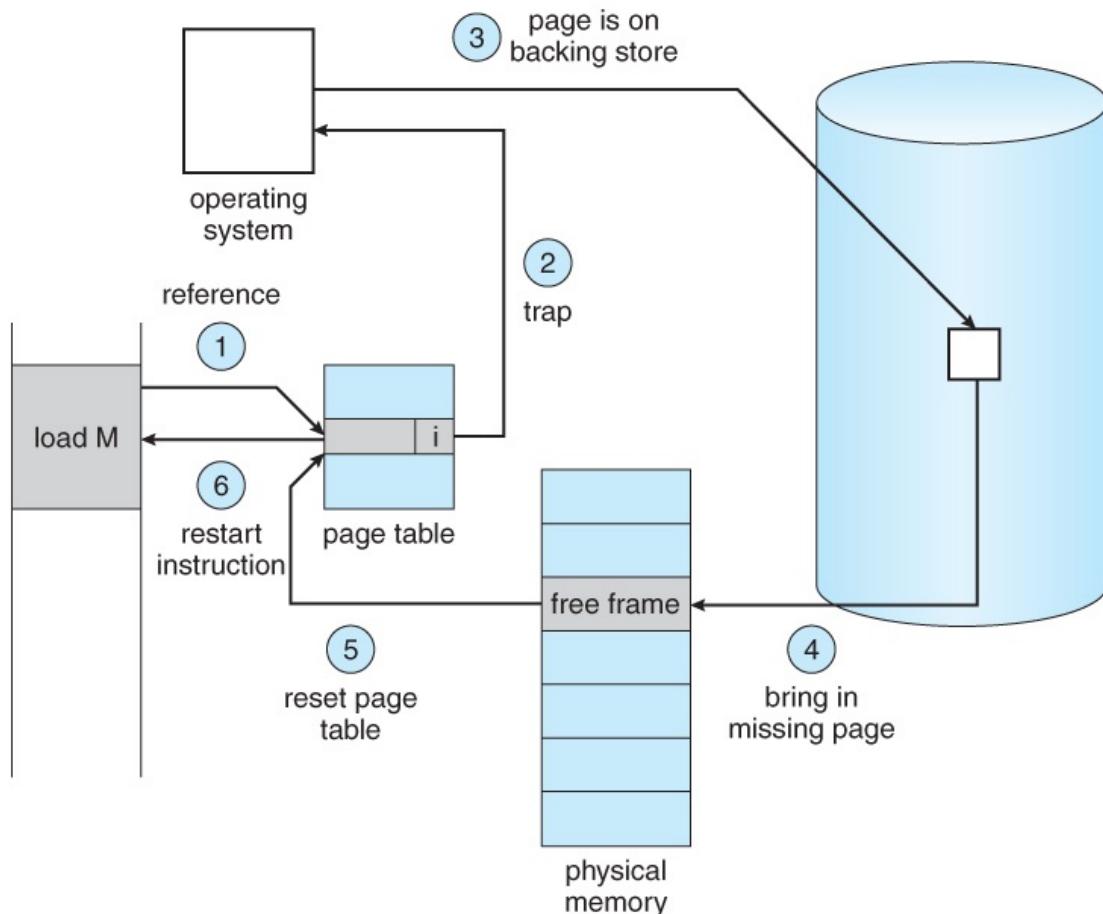
Figure 9.6 - Steps in handling a page fault

- In an extreme case, NO pages are swapped in for a process until they are requested by page faults. This is known as **pure demand paging.**
- In theory each instruction could generate multiple page faults. In practice this is very rare, due to **locality of reference**, covered in section 9.6.1.
- The hardware necessary to support virtual memory is the same as for paging and swapping: A page table and secondary memory. (**Swap space,** whose allocation is discussed in chapter 12.)
- A crucial part of the process is that the instruction must be restarted from scratch once the desired page has been made available in memory. For most simple instructions this is not a major difficulty. However there are some architectures that allow a single instruction to modify a fairly large block of data, (which may span a page boundary), and if some of the data gets modified before the page fault occurs, this could cause problems. One solution is to access both ends of the block before executing the instruction, guaranteeing that the necessary pages get paged in before the instruction begins.

9.2.2 Performance of Demand Paging

- Obviously there is some slowdown and performance hit whenever a page fault occurs and the system has to go get it from memory, but just how big a hit is it exactly?
- There are many steps that occur when servicing a page fault (see book for full details), and some of the steps are optional or variable. But just for the sake of discussion, suppose that a normal memory access requires 200 nanoseconds, and that servicing a page fault takes 8 milliseconds. (8,000,000 nanoseconds, or 40,000 times a normal memory access.) With a **page fault rate** of p, (on a scale from 0 to 1), the effective access time is now:

$$(1 - p) * (200) + p * 8000000$$

$$= 200 + 7,999,800 * p$$
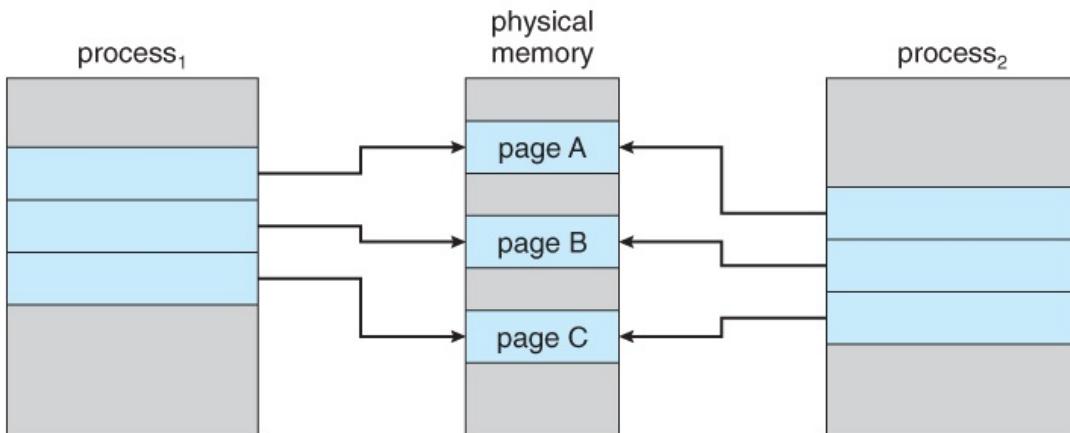
which **clearly** depends heavily on p! Even if only one access in 1000 causes a page fault, the effective access time drops from 200 nanoseconds to 8.2 microseconds, a slowdown of a factor of 40 times. In order to keep the slowdown less than 10%, the page fault rate must be less than 0.0000025, or one in 399,990 accesses.

- A subtlety is that swap space is faster to access than the regular file system, because it does not have to go through the whole directory structure. For this reason some systems will transfer an entire process from the file system to swap space before starting up the process, so that future paging all occurs from the (relatively) faster swap space.
- Some systems use demand paging directly from the file system for binary code (which never changes and hence
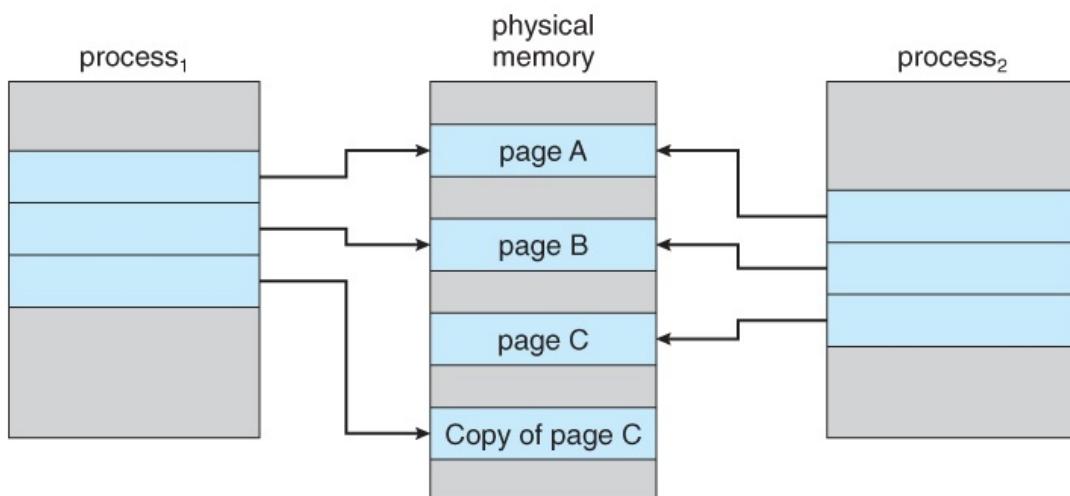
does not have to be stored on a page operation), and to reserve the swap space for data segments that must be stored. This approach is used by both Solaris and BSD Unix.

9.3 Copy-on-Write

- The idea behind a copy-on-write fork is that the pages for a parent process do not have to be actually copied for the child until one or the other of the processes changes the page. They can be simply shared between the two processes in the meantime, with a bit set that the page needs to be copied if it ever gets written to. This is a reasonable approach, since the child process usually issues an exec( ) system call immediately after the fork.

Figure 9.7 - Before process 1 modifies page C.

Figure 9.8 - After process 1 modifies page C.

- Obviously only pages that can be modified even need to be labeled as copy-on-write. Code segments can simply be shared.
- Pages used to satisfy copy-on-write duplications are typically allocated using ***zero-fill-on-demand***, meaning that their previous contents are zeroed out before the copy proceeds.
- Some systems provide an alternative to the fork( ) system call called a ***virtual memory fork, vfork( )***. In this case the parent is suspended, and the child uses the parent's memory pages. This is very fast for process creation, but requires that the child not modify any of the shared memory pages before performing the exec( ) system call. (In essence this addresses the question of which process executes first after a call to fork, the parent or the child. With vfork, the parent is suspended, allowing the child to execute first until it calls exec( ), sharing pages with the parent in the meantime.)

9.4 Page Replacement

- In order to make the most use of virtual memory, we load several processes into memory at the same time. Since we only load the pages that are actually needed by each process at any given time, there is room to load many more processes than if we had to load in the entire process.
- However memory is also needed for other purposes (such as I/O buffering), and what happens if some process suddenly decides it needs more pages and there aren't any free frames available? There are several possible solutions to consider:
  1. Adjust the memory used by I/O buffering, etc., to free up some frames for user processes. The decision of how to allocate memory for I/O versus user processes is a complex one, yielding different policies on different systems. (Some allocate a fixed amount for I/O, and others let the I/O system contend for memory along with everything else.)
  2. Put the process requesting more pages into a wait queue until some free frames become available.
  3. Swap some process out of memory completely, freeing up its page frames.

4. Find some page in memory that isn't being used right now, and swap that page only out to disk, freeing up a frame that can be allocated to the process requesting it. This is known as **page replacement**, and is the most common solution. There are many different algorithms for page replacement, which is the subject of the remainder of this section.
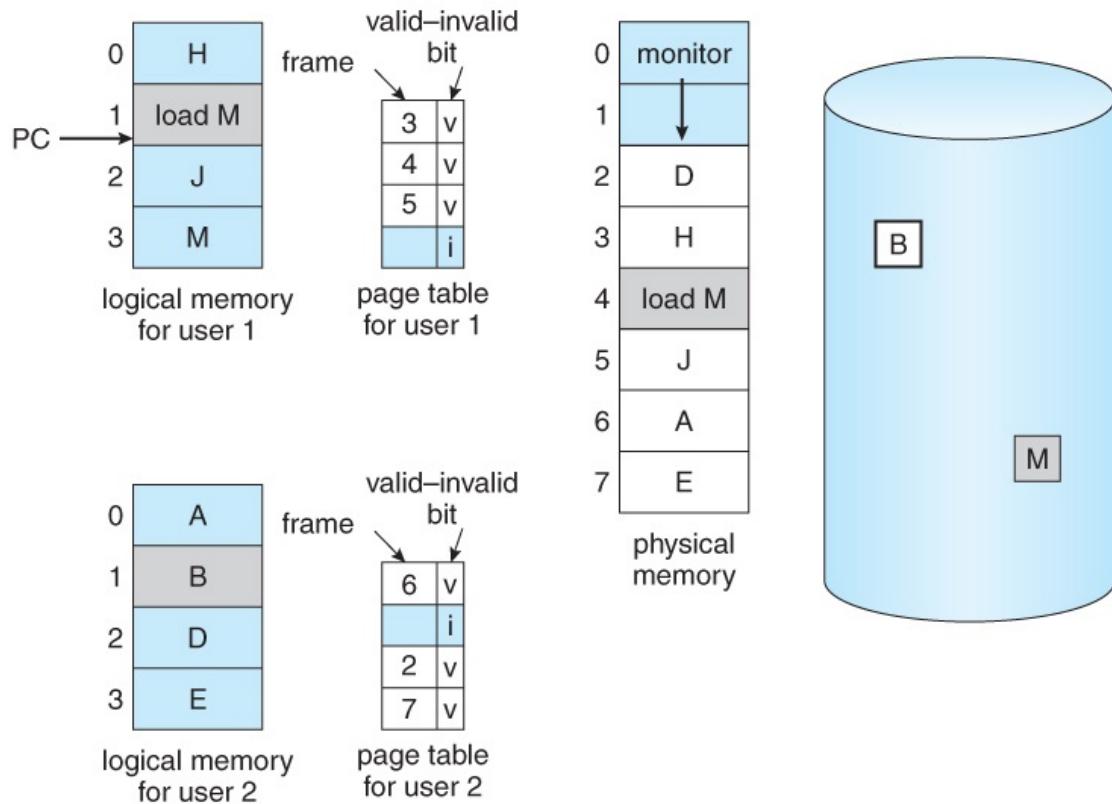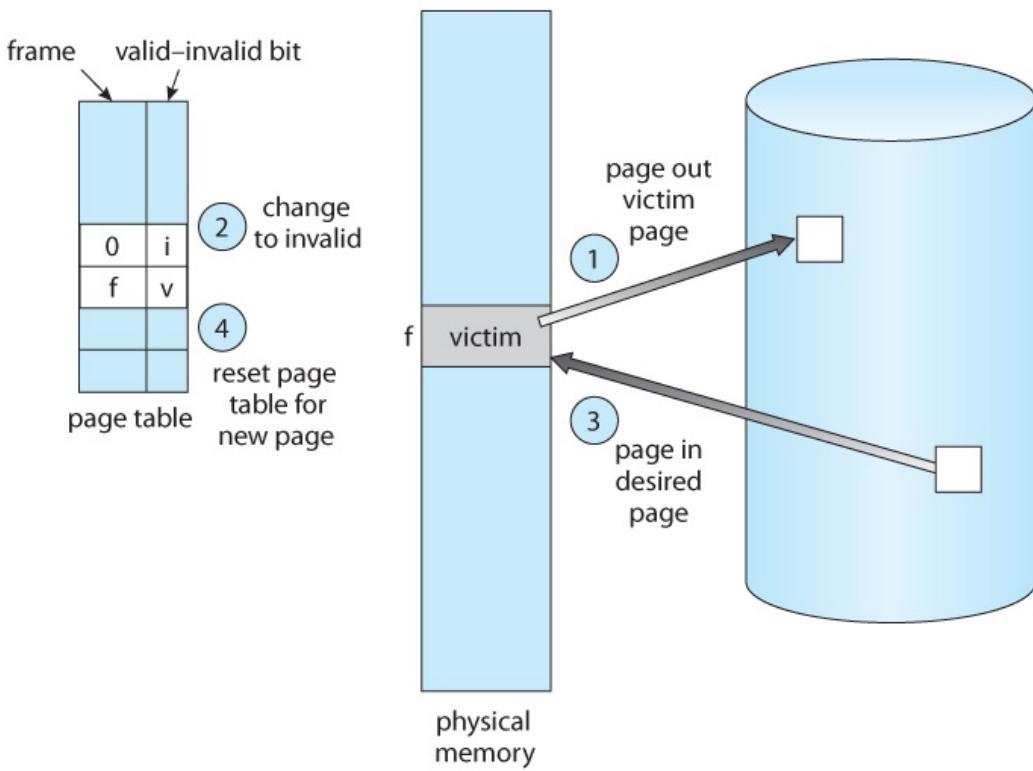

Figure 9.9 - Ned for page replacement.

9.4.1 Basic Page Replacement

- The previously discussed page-fault processing assumed that there would be free frames available on the free-frame list. Now the page-fault handling must be modified to free up a frame if necessary, as follows:
    1. Find the location of the desired page on the disk, either in swap space or in the file system.
    2. Find a free frame:
        a. If there is a free frame, use it.
        b. If there is no free frame, use a page-replacement algorithm to select an existing frame to be replaced, known as the **victim frame**.
        c. Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
    3. Read in the desired page and store it in the frame. Adjust all related page and frame tables to indicate the change.
   4. Restart the process that was waiting for this page.

Figure 9.10 - Page replacement.

- Note that step 3c adds an extra disk write to the page-fault handling, effectively doubling the time required to process a page fault. This can be alleviated somewhat by assigning a *modify bit,* or *dirty bit* to each page, indicating whether or not it has been changed since it was last loaded in from disk. If the dirty bit has not been set, then the page is unchanged, and does not need to be written out to disk. Otherwise the page write is required. It should come as no surprise that many page replacement strategies specifically look for pages that do not have their dirty bit set, and preferentially select clean pages as victim pages. It should also be obvious that unmodifiable code pages never get their dirty bits set.
- There are two major requirements to implement a successful demand paging system. We must develop a *frame-allocation algorithm* and a *page-replacement algorithm.* The former centers around how many frames are allocated to each process (and to other needs), and the latter deals with how to select a page for replacement when there are no free frames available.
- The overall goal in selecting and tuning these algorithms is to generate the fewest number of overall page faults. Because disk access is so slow relative to memory access, even slight improvements to these algorithms can yield large improvements in overall system performance.
- Algorithms are evaluated using a given string of memory accesses known as a *reference string,* which can be generated in one of (at least) three common ways:
  1. Randomly generated, either evenly distributed or with some distribution curve based on observed system behavior. This is the fastest and easiest approach, but may not reflect real performance well, as it ignores locality of reference.
  2. Specifically designed sequences. These are useful for illustrating the properties of comparative algorithms in published papers and textbooks, (and also for homework and exam problems. :-))
  3. Recorded memory references from a live system. This may be the best approach, but the amount of data collected can be enormous, on the order of a million addresses per second. The volume of collected data can be reduced by making two important observations:
     1. Only the page number that was accessed is relevant. The offset within that page does not affect paging operations.
     2. Successive accesses within the same page can be treated as a single page request, because all requests after the first are guaranteed to be page hits. (Since there are no intervening requests for other pages that could remove this page from the page table.)
        - So for example, if pages were of size 100 bytes, then the sequence of address requests (0100, 0432, 0101, 0612, 0634, 0688, 0132, 0038, 0420) would reduce to page requests (1, 4, 1, 6, 1, 0, 4)
- As the number of available frames increases, the number of page faults should decrease, as shown in Figure 9.11:
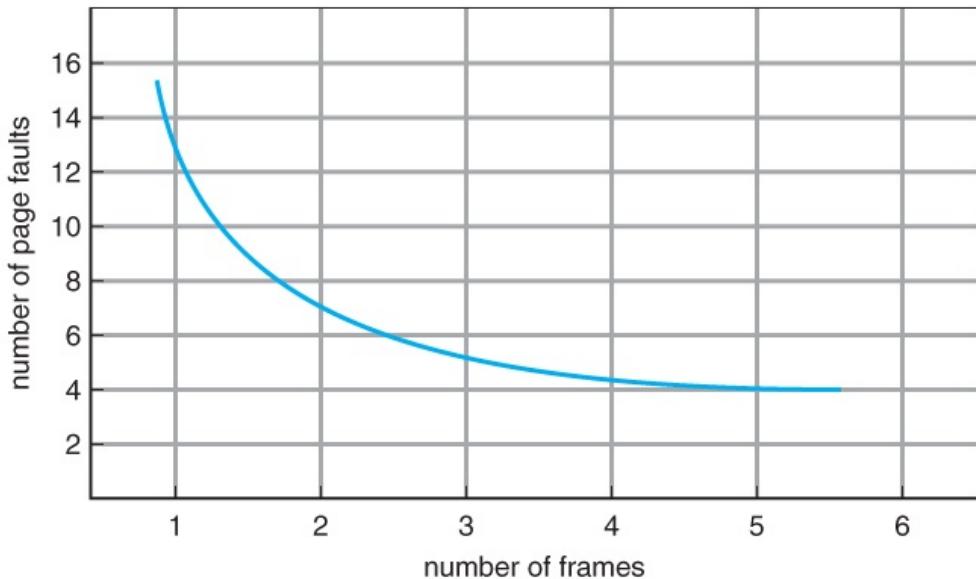
Figure 9.11 - Graph of page faults versus number of frames.

9.4.2 FIFO Page Replacement

- A simple and obvious page replacement strategy is **FIFO**, i.e. first-in-first-out.
- As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim. In the following example, 20 page requests result in 15 page faults:
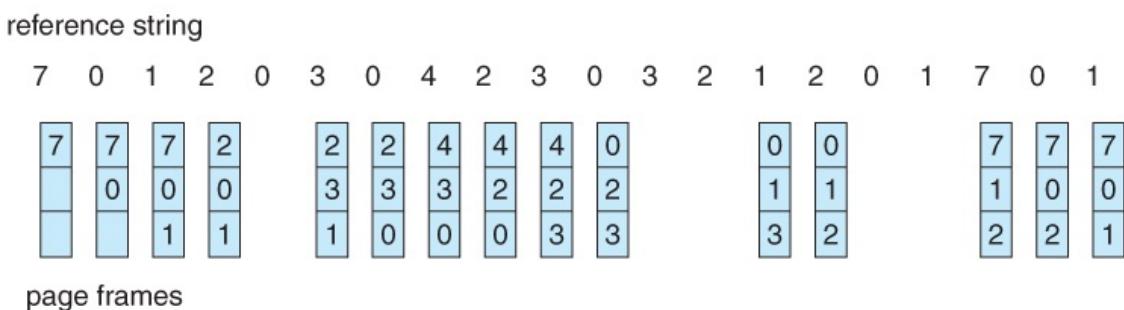
Figure 9.12 - FIFO page-replacement algorithm.

- Although FIFO is simple and easy, it is not always optimal, or even efficient.
- An interesting effect that can occur with FIFO is **Belady's anomaly**, in which increasing the number of frames available can actually **increase** the number of page faults that occur! Consider, for example, the following chart based on the page sequence (1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5) and a varying number of available frames. Obviously the maximum number of faults is 12 (every request generates a fault), and the minimum number is 5 (each page loaded only once), but in between there are some interesting results:
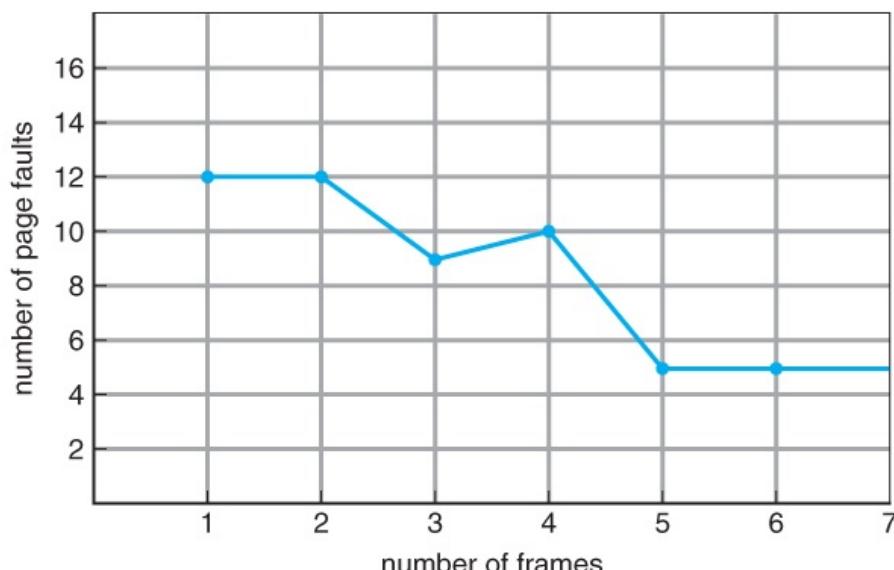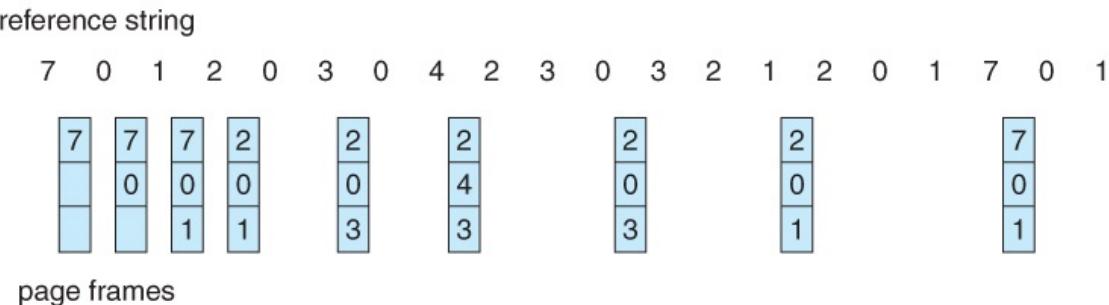

Figure 9.13 - Page-fault curve for FIFO replacement on a reference string.

9.4.3 Optimal Page Replacement
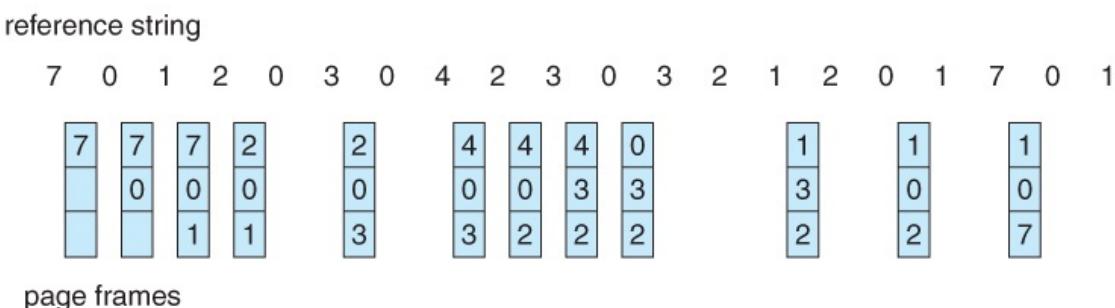
- The discovery of Belady's anomaly lead to the search for an ***optimal page-replacement algorithm***, which is simply that which yields the lowest of all possible page-faults, and which does not suffer from Belady's anomaly.
- Such an algorithm does exist, and is called ***OPT or MIN.*** This algorithm is simply "Replace the page that will not be used for the longest time in the future."
- For example, Figure 9.14 shows that by applying OPT to the same reference string used for the FIFO example, the minimum number of possible page faults is 9. Since 6 of the page-faults are unavoidable (the first reference to each new page), FIFO can be shown to require 3 times as many (extra) page faults as the optimal algorithm. (Note: The book claims that only the first three page faults are required by all algorithms, indicating that FIFO is only twice as bad as OPT.)
- Unfortunately OPT cannot be implemented in practice, because it requires foretelling the future, but it makes a nice benchmark for the comparison and evaluation of real proposed new algorithms.
- In practice most page-replacement algorithms try to approximate OPT by predicting (estimating) in one fashion or another what page will not be used for the longest period of time. The basis of FIFO is the prediction that the page that was brought in the longest time ago is the one that will not be needed again for the longest future time, but as we shall see, there are many other prediction methods, all striving to match the performance of OPT.


Figure 9.14 - Optimal page-replacement algorithm

9.4.4 LRU Page Replacement

- The prediction behind ***LRU,*** the ***Least Recently Used,*** algorithm is that the page that has not been used in the longest time is the one that will not be used again in the near future. (Note the distinction between FIFO and LRU: The former looks at the oldest ***load*** time, and the latter looks at the oldest ***use*** time.)
- Some view LRU as analogous to OPT, except looking backwards in time instead of forwards. (OPT has the interesting property that for any reference string S and its reverse R, OPT will generate the same number of page faults for S and for R. It turns out that LRU has this same property.)
- Figure 9.15 illustrates LRU for our sample string, yielding 12 page faults, (as compared to 15 for FIFO and 9 for OPT.)


Figure 9.15 - LRU page-replacement algorithm.

- LRU is considered a good replacement policy, and is often used. The problem is how exactly to implement it. There are two simple approaches commonly used:
  1. **Counters.** Every memory access increments a counter, and the current value of this counter is stored in the page table entry for that page. Then finding the LRU page involves simple searching the table for the page with the smallest counter value. Note that overflowing of the counter must be considered.
  2. **Stack.** Another approach is to use a stack, and whenever a page is accessed, pull that page from the middle of the stack and place it on the top. The LRU page will always be at the bottom of the stack. Because this requires removing objects from the middle of the stack, a doubly linked list is the recommended data structure.
- Note that both implementations of LRU require hardware support, either for incrementing the counter or for managing the stack, as these operations must be performed for ***every*** memory access.
- Neither LRU or OPT exhibit Belady's anomaly. Both belong to a class of page-replacement algorithms called ***stack algorithms,*** which can never exhibit Belady's anomaly. A stack algorithm is one in which the pages kept in memory

for a frame set of size N will always be a subset of the pages kept for a frame size of N + 1. In the case of LRU, (and particularly the stack implementation thereof), the top N pages of the stack will be the same for all frame set sizes of N or anything larger.
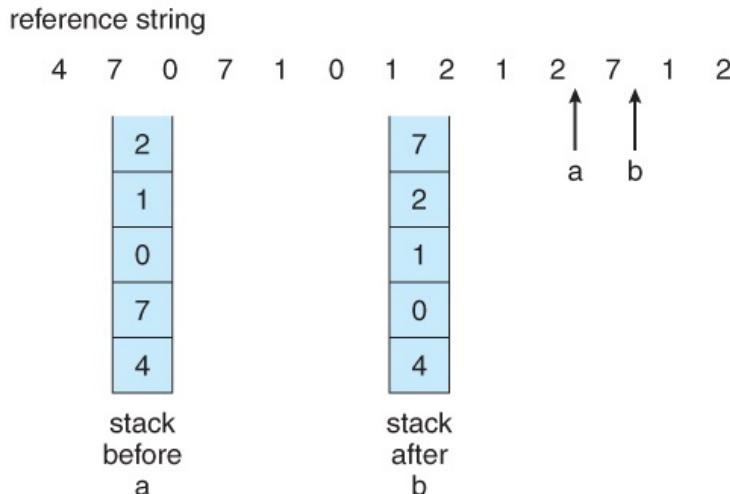

Figure 9.16 - Use of a stack to record the most recent page references.

9.4.5 LRU-Approximation Page Replacement

- Unfortunately full implementation of LRU requires hardware support, and few systems provide the full hardware support necessary.
- However many systems offer some degree of HW support, enough to approximate LRU fairly well. (In the absence of ANY hardware support, FIFO might be the best available choice.)
- In particular, many systems provide a **reference bit** for every entry in a page table, which is set anytime that page is accessed. Initially all bits are set to zero, and they can also all be cleared at any time. One bit of precision is enough to distinguish pages that have been accessed since the last clear from those that have not, but does not provide any finer grain of detail.

9.4.5.1 Additional-Reference-Bits Algorithm

- Finer grain is possible by storing the most recent 8 reference bits for each page in an 8-bit byte in the page table entry, which is interpreted as an unsigned int.
    - At periodic intervals (clock interrupts), the OS takes over, and right-shifts each of the reference bytes by one bit.
    - The high-order (leftmost) bit is then filled in with the current value of the reference bit, and the reference bits are cleared.
    - At any given time, the page with the smallest value for the reference byte is the LRU page.
- Obviously the specific number of bits used and the frequency with which the reference byte is updated are adjustable, and are tuned to give the fastest performance on a given hardware platform.

9.4.5.2 Second-Chance Algorithm

- The **second chance algorithm** is essentially a FIFO, except the reference bit is used to give pages a second chance at staying in the page table.
    - When a page must be replaced, the page table is scanned in a FIFO (circular queue) manner.
    - If a page is found with its reference bit not set, then that page is selected as the next victim.
    - If, however, the next page in the FIFO **does** have its reference bit set, then it is given a second chance:
        - The reference bit is cleared, and the FIFO search continues.
        - If some other page is found that did not have its reference bit set, then that page will be selected as the victim, and this page (the one being given the second chance) will be allowed to stay in the page table.
        - If , however, there are no other pages that do not have their reference bit set, then this page will be selected as the victim when the FIFO search circles back around to this page on the second pass.
- If all reference bits in the table are set, then second chance degrades to FIFO, but also requires a complete search of the table for every page-replacement.
- As long as there are some pages whose reference bits are not set, then any page referenced frequently enough gets to stay in the page table indefinitely.
- This algorithm is also known as the **clock** algorithm, from the hands of the clock moving around the

circular queue.
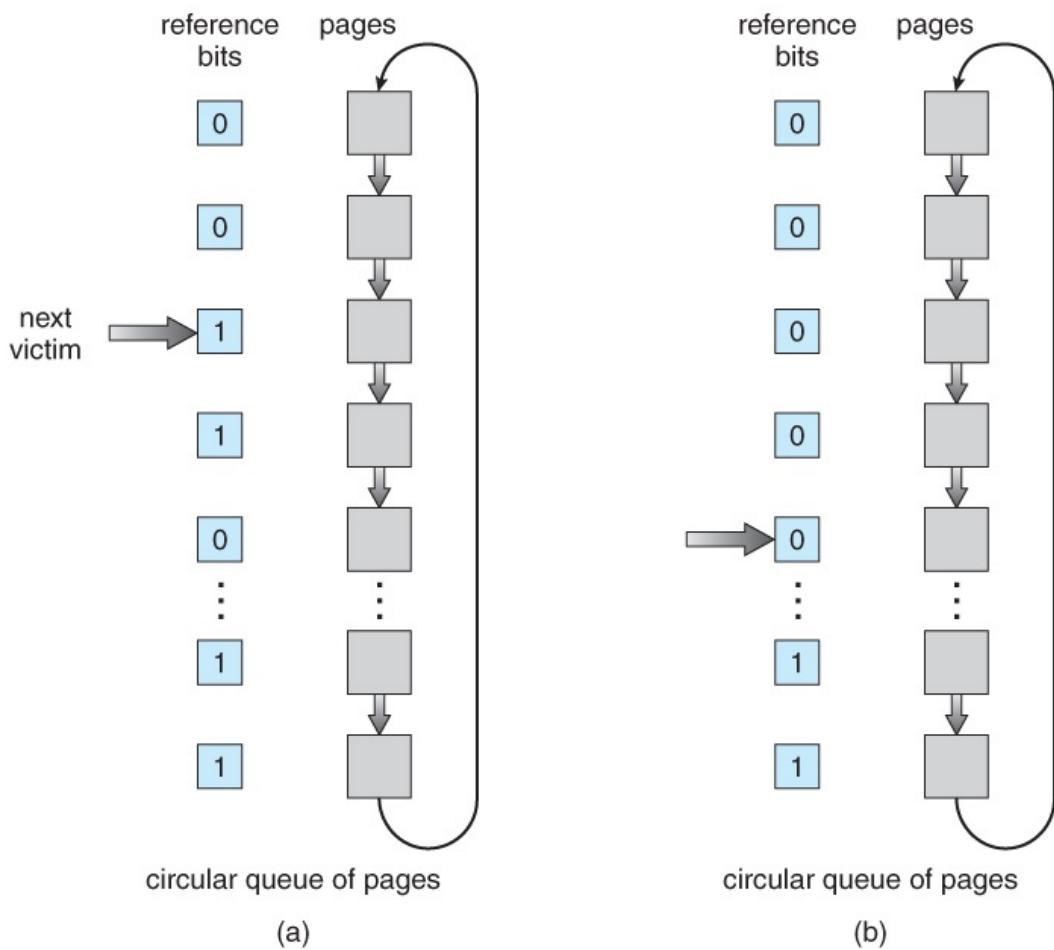
Figure 9.17 - Second-chance (clock) page-replacement algorithm.

9.4.5.3 Enhanced Second-Chance Algorithm

- The **enhanced second chance algorithm** looks at the reference bit and the modify bit (dirty bit) as an ordered page, and classifies pages into one of four classes:
    1. (0, 0) - Neither recently used nor modified.
    2. (0, 1) - Not recently used, but modified.
    3. (1, 0) - Recently used, but clean.
    4. (1, 1) - Recently used and modified.
- This algorithm searches the page table in a circular fashion (in as many as four passes), looking for the first page it can find in the lowest numbered category. I.e. it first makes a pass looking for a (0, 0), and then if it can't find one, it makes another pass looking for a (0, 1), etc.
- The main difference between this algorithm and the previous one is the preference for replacing clean pages if possible.

9.4.6 Counting-Based Page Replacement

- There are several algorithms based on counting the number of references that have been made to a given page, such as:
    - **Least Frequently Used, LFU:** Replace the page with the lowest reference count. A problem can occur if a page is used frequently initially and then not used any more, as the reference count remains high. A solution to this problem is to right-shift the counters periodically, yielding a time-decaying average reference count.
    - **Most Frequently Used, MFU:** Replace the page with the highest reference count. The logic behind this idea is that pages that have already been referenced a lot have been in the system a long time, and we are probably done with them, whereas pages referenced only a few times have only recently been loaded, and we still need them.
- In general counting-based algorithms are not commonly used, as their implementation is expensive and they do not approximate OPT well.

9.4.7 Page-Buffering Algorithms

There are a number of page-buffering algorithms that can be used in conjunction with the afore-mentioned

algorithms, to improve overall performance and sometimes make up for inherent weaknesses in the hardware and/or the underlying page-replacement algorithms:

- Maintain a certain minimum number of free frames at all times. When a page-fault occurs, go ahead and allocate one of the free frames from the free list first, to get the requesting process up and running again as quickly as possible, and then select a victim page to write to disk and free up a frame as a second step.
- Keep a list of modified pages, and when the I/O system is otherwise idle, have it write these pages out to disk, and then clear the modify bits, thereby increasing the chance of finding a "clean" page for the next potential victim.
- Keep a pool of free frames, but remember what page was in it before it was made free. Since the data in the page is not actually cleared out when the page is freed, it can be made an active page again without having to load in any new data from disk. This is useful when an algorithm mistakenly replaces a page that in fact is needed again soon.

9.4.8 Applications and Page Replacement

- Some applications (most notably database programs) understand their data accessing and caching needs better than the general-purpose OS, and should therefore be given reign to do their own memory management.
- Sometimes such programs are given a *raw disk partition* to work with, containing raw data blocks and no file system structure. It is then up to the application to use this disk partition as extended memory or for whatever other reasons it sees fit.

9.5 Allocation of Frames

We said earlier that there were two important tasks in virtual memory management: a page-replacement strategy and a frame-allocation strategy. This section covers the second part of that pair.

9.5.1 Minimum Number of Frames

- The absolute minimum number of frames that a process must be allocated is dependent on system architecture, and corresponds to the worst-case scenario of the number of pages that could be touched by a single (machine) instruction.
- If an instruction (and its operands) spans a page boundary, then multiple pages could be needed just for the instruction fetch.
- Memory references in an instruction touch more pages, and if those memory locations can span page boundaries, then multiple pages could be needed for operand access also.
- The worst case involves indirect addressing, particularly where multiple levels of indirect addressing are allowed. Left unchecked, a pointer to a pointer to a pointer to a pointer to a . . . could theoretically touch every page in the virtual address space in a single machine instruction, requiring every virtual page be loaded into physical memory simultaneously. For this reason architectures place a limit (say 16) on the number of levels of indirection allowed in an instruction, which is enforced with a counter initialized to the limit and decremented with every level of indirection in an instruction - If the counter reaches zero, then an "excessive indirection" trap occurs. This example would still require a minimum frame allocation of 17 per process.

9.5.2 Allocation Algorithms

- **Equal Allocation -** If there are m frames available and n processes to share them, each process gets m / n frames, and the leftovers are kept in a free-frame buffer pool.
- **Proportional Allocation -** Allocate the frames proportionally to the size of the process, relative to the total size of all processes. So if the size of process i is S_i, and S is the sum of all S_i, then the allocation for process P_i is a_i = m * S_i / S.
- Variations on proportional allocation could consider priority of process rather than just their size.
- Obviously all allocations fluctuate over time as the number of available free frames, m, fluctuates, and all are also subject to the constraints of minimum allocation. (If the minimum allocations cannot be met, then processes must either be swapped out or not allowed to start until more free frames become available.)

9.5.3 Global versus Local Allocation

- One big question is whether frame allocation (page replacement) occurs on a local or global level.
- With local replacement, the number of pages allocated to a process is fixed, and page replacement occurs only amongst the pages allocated to this process.
- With global replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not.
- Local page replacement allows processes to better control their own page fault rates, and leads to more consistent performance of a given process over different system load levels.
- Global page replacement is overall more efficient, and is the more commonly used approach.

9.5.4 Non-Uniform Memory Access
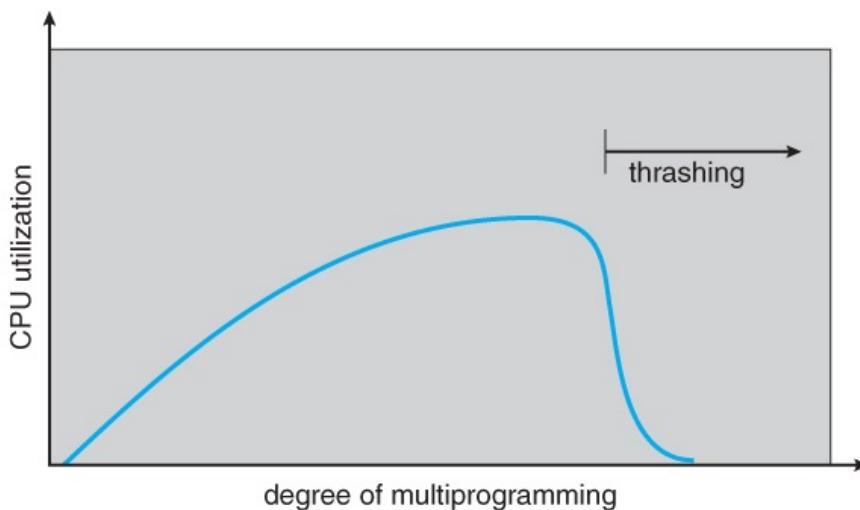
- The above arguments all assume that all memory is equivalent, or at least has equivalent access times.
- This may not be the case in multiple-processor systems, especially where each CPU is physically located on a separate circuit board which also holds some portion of the overall system memory.
- In these latter systems, CPUs can access memory that is physically located on the same board much faster than the memory on the other boards.
- The basic solution is akin to processor affinity - At the same time that we try to schedule processes on the same CPU to minimize cache misses, we also try to allocate memory for those processes on the same boards, to minimize access times.
- The presence of threads complicates the picture, especially when the threads get loaded onto different processors.
- Solaris uses an *lgroup* as a solution, in a hierarchical fashion based on relative latency. For example, all processors and RAM on a single board would probably be in the same lgroup. Memory assignments are made within the same lgroup if possible, or to the next nearest lgroup otherwise. (Where "nearest" is defined as having the lowest access time.)

9.6 Thrashing

- If a process cannot maintain its minimum required number of frames, then it must be swapped out, freeing up frames for other processes. This is an intermediate level of CPU scheduling.
- But what about a process that can keep its minimum, but cannot keep all of the frames that it is currently using on a regular basis? In this case it is forced to page out pages that it will need again in the very near future, leading to large numbers of page faults.
- A process that is spending more time paging than executing is said to be *thrashing.*
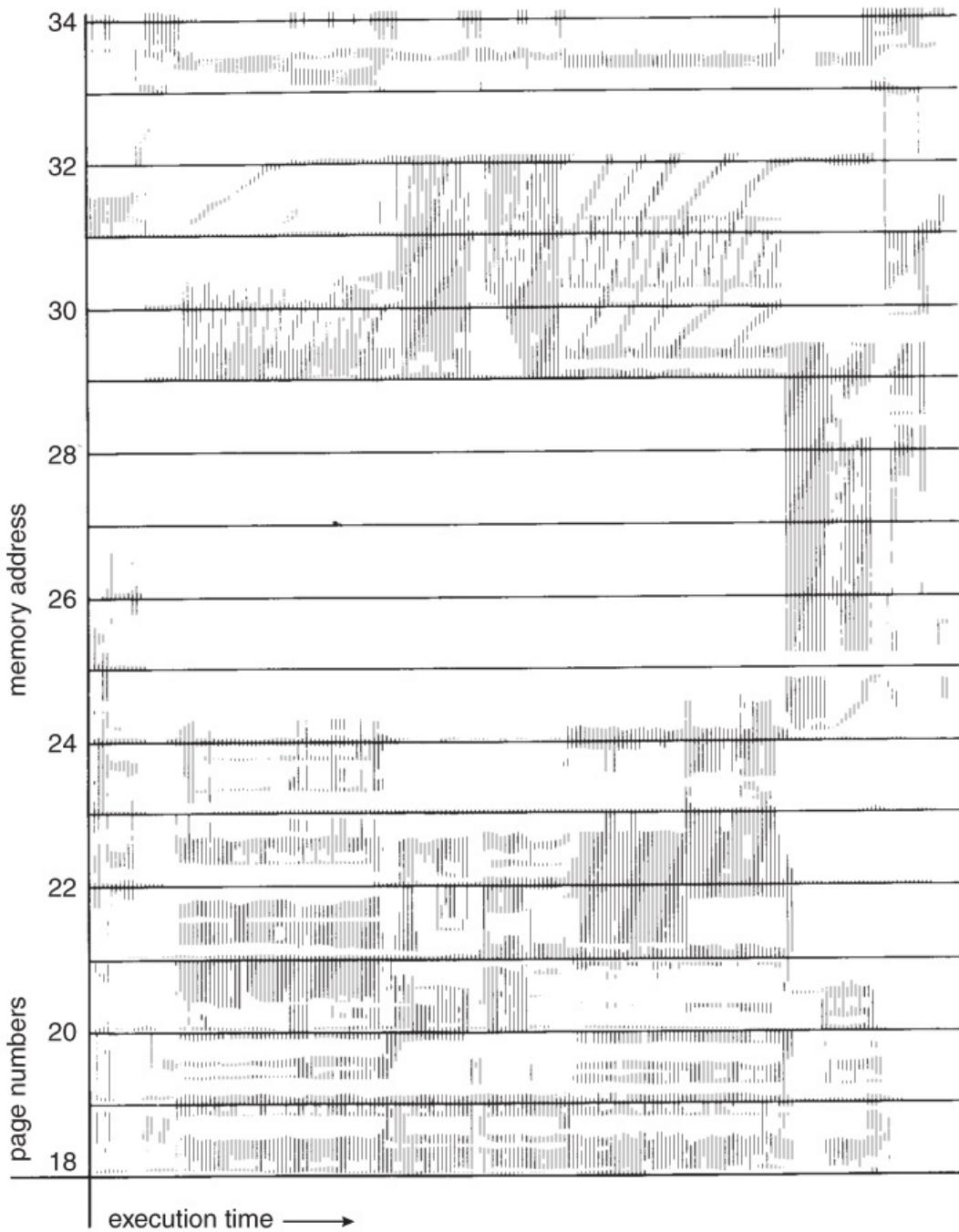
9.6.1 Cause of Thrashing

- Early process scheduling schemes would control the level of multiprogramming allowed based on CPU utilization, adding in more processes when CPU utilization was low.
- The problem is that when memory filled up and processes started spending lots of time waiting for their pages to page in, then CPU utilization would lower, causing the schedule to add in even more processes and exacerbating the problem! Eventually the system would essentially grind to a halt.
- Local page replacement policies can prevent one thrashing process from taking pages away from other processes, but it still tends to clog up the I/O queue, thereby slowing down any other process that needs to do even a little bit of paging (or any other I/O for that matter.)


Figure 9.18 - Thrashing
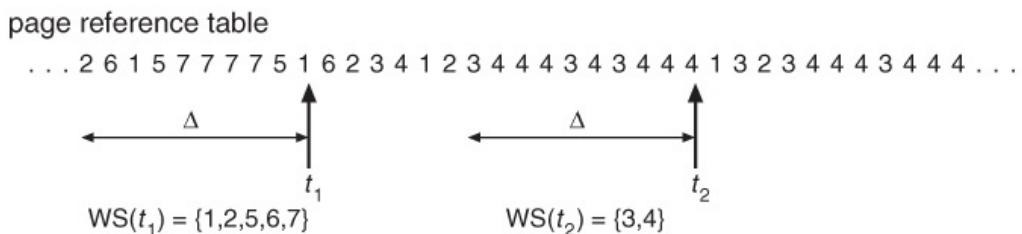
- To prevent thrashing we must provide processes with as many frames as they really need "right now", but how do we know what that is?
- The *locality model* notes that processes typically access memory references in a given *locality,* making lots of references to the same general area of memory before moving periodically to a new locality, as shown in Figure 9.19 below. If we could just keep as many frames as are involved in the current locality, then page faulting would occur primarily on switches from one locality to another. (E.g. when one function exits and another is called.)

Figure 9.19 - Locality in a memory-reference pattern.

9.6.2 Working-Set Model

- The **working set model** is based on the concept of locality, and defines a **working set window**, of length **delta.** Whatever pages are included in the most recent delta page references are said to be in the processes working set window, and comprise its current working set, as illustrated in Figure 9.20:

page reference table

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .

Δ Δ

t_1 t_2

$WS(t_1) = \{1,2,5,6,7\}$ $WS(t_2) = \{3,4\}$

Figure 9.20 - Working-set model.
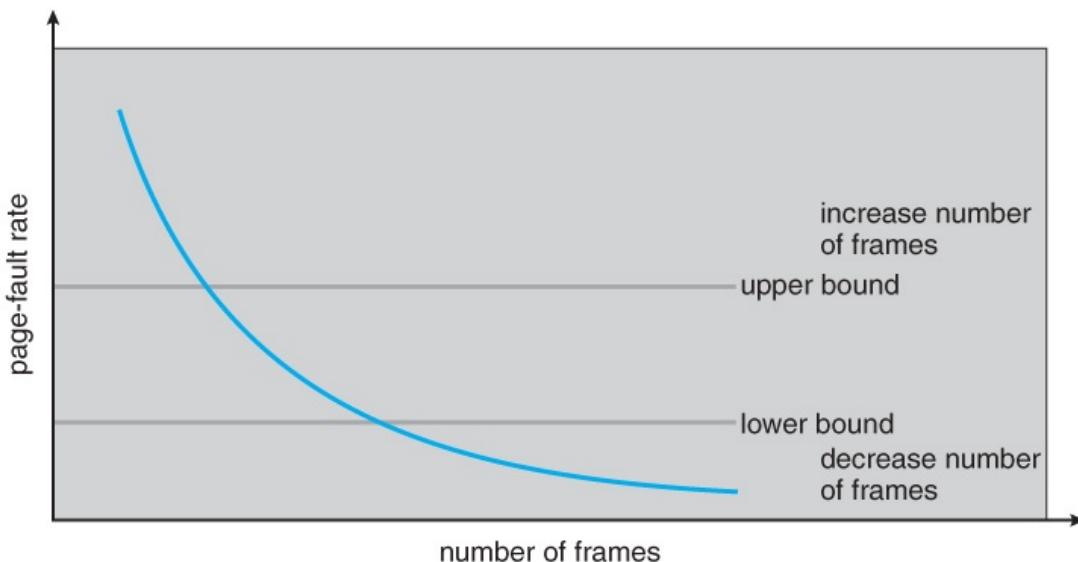
- The selection of delta is critical to the success of the working set model - If it is too small then it does not encompass all of the pages of the current locality, and if it is too large, then it encompasses pages that are no longer being frequently accessed.
- The total demand, D, is the sum of the sizes of the working sets for all processes. If D exceeds the total number of

available frames, then at least one process is thrashing, because there are not enough frames available to satisfy its minimum working set. If D is significantly less than the currently available frames, then additional processes can be launched.

- The hard part of the working-set model is keeping track of what pages are in the current working set, since every reference adds one to the set and removes one older page. An approximation can be made using reference bits and a timer that goes off after a set interval of memory references:
  - For example, suppose that we set the timer to go off after every 5000 references (by any process), and we can store two additional historical reference bits in addition to the current reference bit.
  - Every time the timer goes off, the current reference bit is copied to one of the two historical bits, and then cleared.
  - If any of the three bits is set, then that page was referenced within the last 15,000 references, and is considered to be in that processes reference set.
  - Finer resolution can be achieved with more historical bits and a more frequent timer, at the expense of greater overhead.
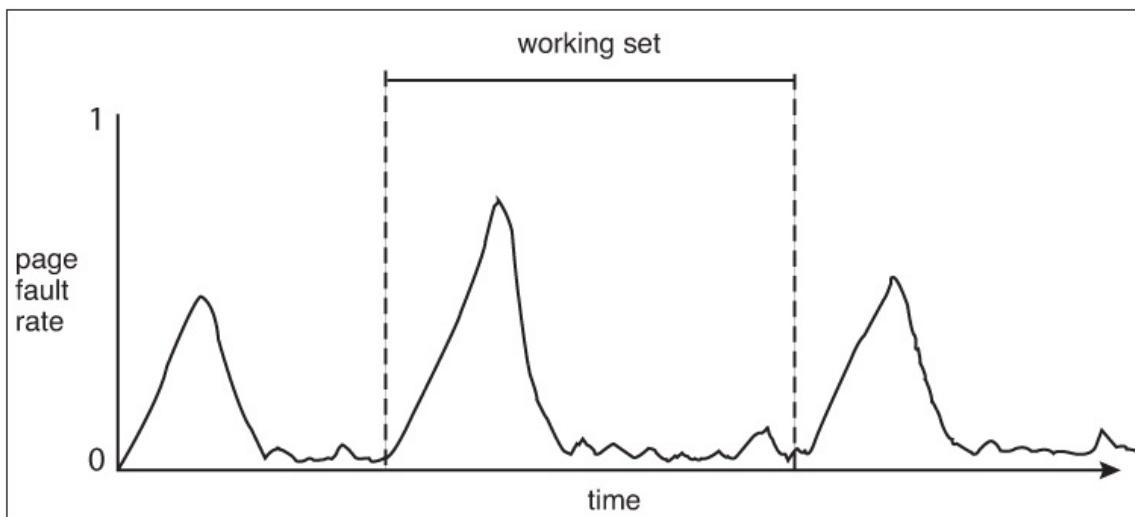
9.6.3 Page-Fault Frequency

- A more direct approach is to recognize that what we really want to control is the page-fault rate, and to allocate frames based on this directly measurable value. If the page-fault rate exceeds a certain upper bound then that process needs more frames, and if it is below a given lower bound, then it can afford to give up some of its frames to other processes.
- (I suppose a page-replacement strategy could be devised that would select victim frames based on the process with the lowest current page-fault frequency.)


Figure 9.21 - Page-fault frequency.

- Note that there is a direct relationship between the page-fault rate and the working-set, as a process moves from one locality to another:

Unnumbered side bar in Ninth Edition

9.7 Memory-Mapped Files

- Rather than accessing data files directly via the file system with every file access, data files can be paged into memory the same as process files, resulting in much faster accesses (except of course when page-faults occur.) This is known as **memory-mapping** a file.

9.7.1 Basic Mechanism

- Basically a file is mapped to an address range within a process's virtual address space, and then paged in as needed using the ordinary demand paging system.
- Note that file writes are made to the memory page frames, and are not immediately written out to disk. (This is the purpose of the "flush( )" system call, which may also be needed for stdout in some cases. See the timekiller program for an example of this.)
- This is also why it is important to "close( )" a file when one is done writing to it - So that the data can be safely flushed out to disk and so that the memory frames can be freed up for other purposes.
- Some systems provide special system calls to memory map files and use direct disk access otherwise. Other systems map the file to process address space if the special system calls are used and map the file to kernel address space otherwise, but do memory mapping in either case.
- File sharing is made possible by mapping the same file to the address space of more than one process, as shown in Figure 9.23 below. Copy-on-write is supported, and mutual exclusion techniques (chapter 6) may be needed to avoid synchronization problems.
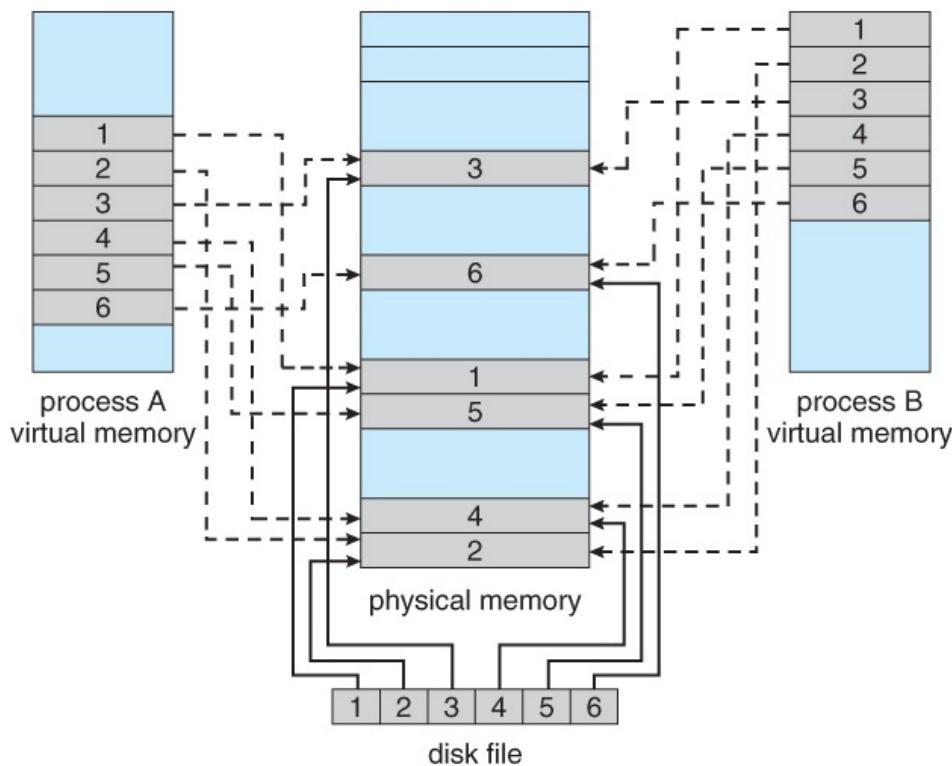
Figure 9.22 Memory-mapped files.

- Shared memory can be implemented via shared memory-mapped files (Windows), or it can be implemented through a separate process (Linux, UNIX.)

9.7.2 Shared Memory in the Win32 API

- Windows implements shared memory using shared memory-mapped files, involving three basic steps:
    1. Create a file, producing a HANDLE to the new file.
    2. Name the file as a shared object, producing a HANDLE to the shared object.
    3. Map the shared object to virtual memory address space, returning its base address as a void pointer (LPVOID).
- This is illustrated in Figures 9.24 to 9.26 (annotated.)
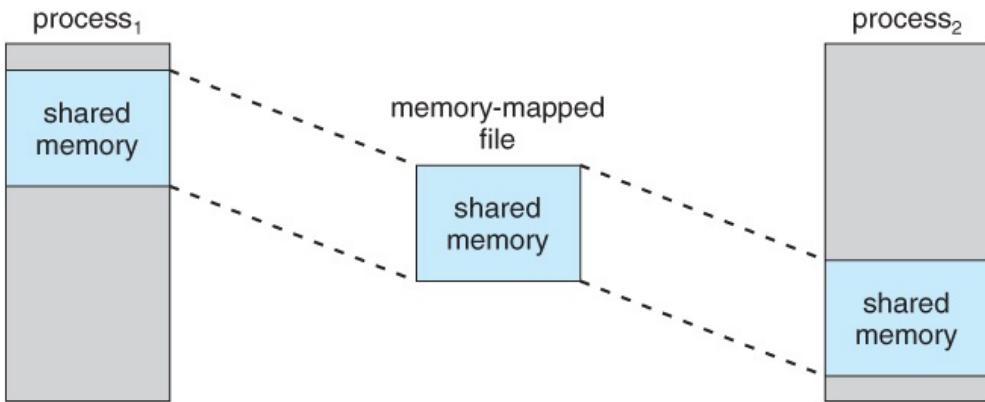
Figure 9.23 - Shared memory in Windows using memory-mapped I/O.

Producer creates file, makes it a shared object, maps it to memory, writes to it, and closes

```c
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   HANDLE hFile, hMapFile;
   LPVOID lpMapAddress;

   hFile = CreateFile("temp.txt", // file name
      GENERIC_READ | GENERIC_WRITE, // read/write access
      0, // no sharing of the file
      NULL, // default security                  Create the file
      OPEN_ALWAYS, // open new or existing file
      FILE_ATTRIBUTE_NORMAL, // routine file attributes
      NULL); // no file template

   hMapFile = CreateFileMapping(hFile, // file handle
      NULL, // default security
      PAGE_READWRITE, // read/write access to mapped pages
      0, // map entire file
      0,                             Make it a named shared object
      TEXT("SharedObject")); // named shared memory object

   lpMapAddress = MapViewOfFile(hMapFile, // mapped object handle
      FILE_MAP_ALL_ACCESS, // read/write access
      0, // mapped view of entire file
      0,                   Map it into virtual memory address space
      0);

   // write to shared memory
   sprintf(lpMapAddress,"Shared memory message");

   UnmapViewOfFile(lpMapAddress);          Write and close
   CloseHandle(hFile);
   CloseHandle(hMapFile);
}
```

Figure 9.25 Producer writing to shared memory using the Win32 API.

Figure 9.24

Consumer accesses named shared object, maps to memory, reads, and closes

```c
#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    HANDLE hMapFile;
    LPVOID lpMapAddress;
                                        // Access shared memory
    hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, // R/W access
        FALSE, // no inheritance
        TEXT("SharedObject")); // name of mapped file object

    lpMapAddress = MapViewOfFile(hMapFile, // mapped object handle
        FILE_MAP_ALL_ACCESS, // read/write access
        0, // mapped view of entire file
        0,                      // Map to virtual memory space
        0);

    // read from shared memory
    printf("Read message %s", lpMapAddress);

    UnmapViewOfFile(lpMapAddress);       // Read and close
    CloseHandle(hMapFile);
}
```

Figure 9.26 Consumer reading from shared memory using the Win32 API.

Figure 9.25

9.7.3 Memory-Mapped I/O

- All access to devices is done by writing into (or reading from) the device's registers. Normally this is done via special I/O instructions.
- For certain devices it makes sense to simply map the device's registers to addresses in the process's virtual address space, making device I/O as fast and simple as any other memory access. Video controller cards are a classic example of this.
- Serial and parallel devices can also use memory mapped I/O, mapping the device registers to specific memory addresses known as **I/O Ports**, e.g. 0xF8. Transferring a series of bytes must be done one at a time, moving only as fast as the I/O device is prepared to process the data, through one of two mechanisms:
    - **Programmed I/O (PIO)**, also known as **polling.** The CPU periodically checks the control bit on the device, to see if it is ready to handle another byte of data.
    - **Interrupt Driven.** The device generates an interrupt when it either has another byte of data to deliver or is ready to receive another byte.

9.8 Allocating Kernel Memory

- Previous discussions have centered on process memory, which can be conveniently broken up into page-sized chunks, and the only fragmentation that occurs is the average half-page lost to internal fragmentation for each process (segment.)
- There is also additional memory allocated to the kernel, however, which cannot be so easily paged. Some of it is used for I/O buffering and direct access by devices, example, and must therefore be contiguous and not affected by paging. Other memory is used for internal kernel data structures of various sizes, and since kernel memory is often locked (restricted from being ever swapped out), management of this resource must be done carefully to avoid internal fragmentation or other waste. (I.e. you would like the kernel to consume as little memory as possible, leaving as much as possible for user processes.) Accordingly there are several classic algorithms in place for allocating kernel memory structures.

9.8.1 Buddy System

- The Buddy **System** allocates memory using a **power of two allocator**.
- Under this scheme, memory is always allocated as a power of 2 (4K, 8K, 16K, etc), rounding up to the next nearest

power of two if necessary.
- If a block of the correct size is not currently available, then one is formed by splitting the next larger block in two, forming two matched buddies. (And if that larger size is not available, then the next largest available size is split, and so on.)
- One nice feature of the buddy system is that if the address of a block is exclusively ORed with the size of the block, the resulting address is the address of the buddy of the same size, which allows for fast and easy *coalescing* of free blocks back into larger blocks.
  - Free lists are maintained for every size block.
  - If the necessary block size is not available upon request, a free block from the next largest size is split into two buddies of the desired size. (Recursively splitting larger size blocks if necessary.)
  - When a block is freed, its buddy's address is calculated, and the free list for that size block is checked to see if the buddy is also free. If it is, then the two buddies are coalesced into one larger free block, and the process is repeated with successively larger free lists.
  - See the (annotated) Figure 9.27 below for an example.
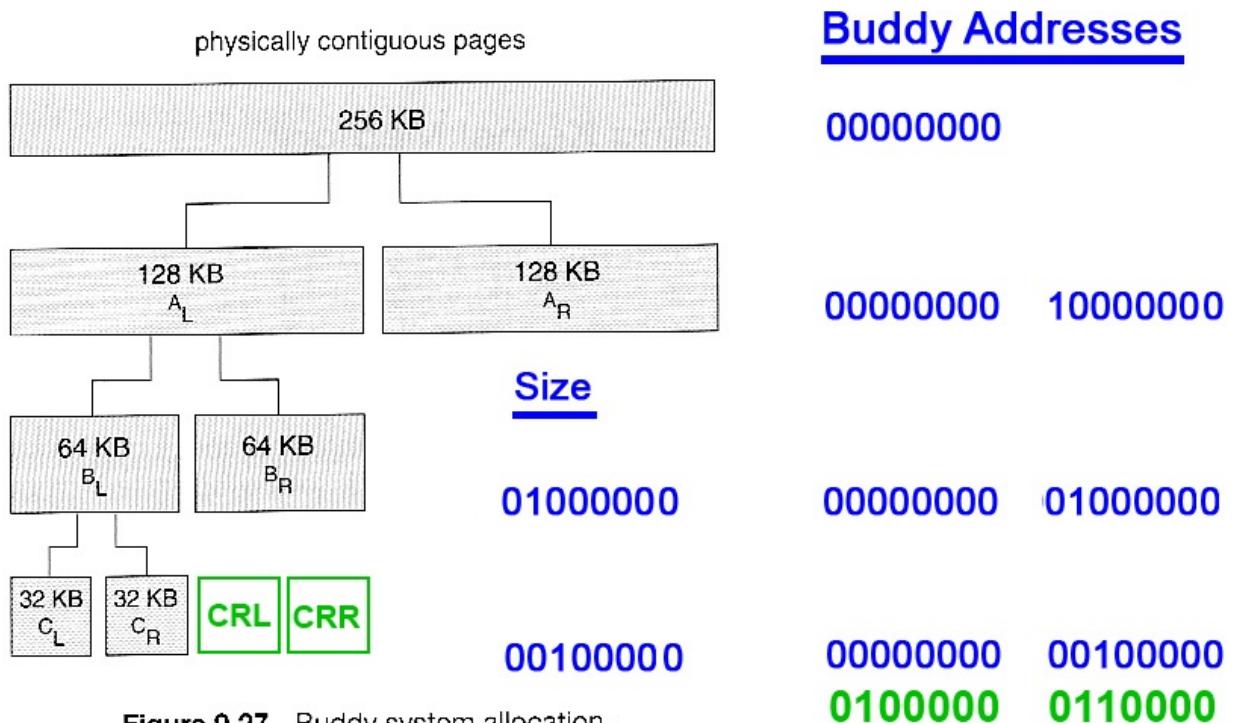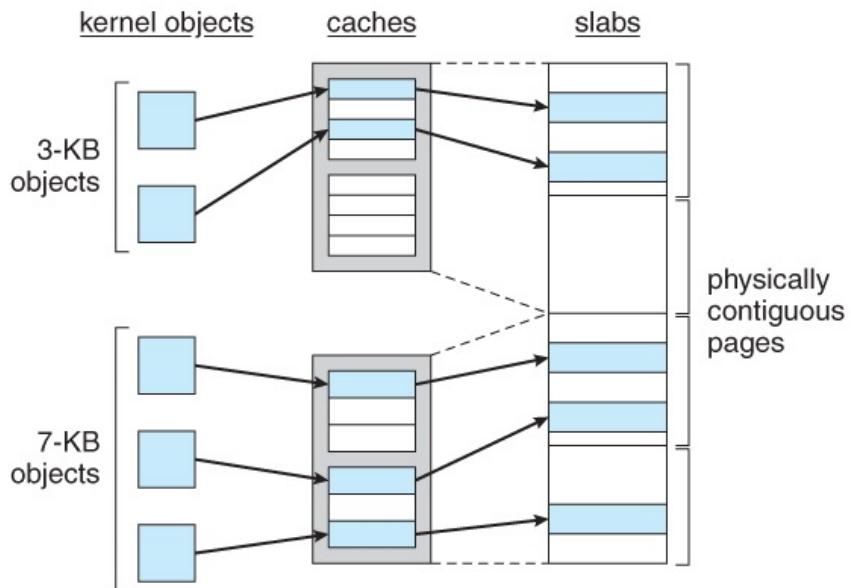
Figure 9.27 Buddy system allocation.

Figure 9.26

9.8.2 Slab Allocation

- *Slab Allocation* allocates memory to the kernel in chunks called *slabs*, consisting of one or more contiguous pages. The kernel then creates separate caches for each type of data structure it might need from one or more slabs. Initially the caches are marked empty, and are marked full as they are used.
- New requests for space in the cache is first granted from empty or partially empty slabs, and if all slabs are full, then additional slabs are allocated.
- (This essentially amounts to allocating space for arrays of structures, in large chunks suitable to the size of the structure being stored. For example if a particular structure were 512 bytes long, space for them would be allocated in groups of 8 using 4K pages. If the structure were 3K, then space for 4 of them could be allocated at one time in a slab of 12K using three 4K pages.
- Benefits of slab allocation include lack of internal fragmentation and fast allocation of space for individual structures
- Solaris uses slab allocation for the kernel and also for certain user-mode memory allocations. Linux used the buddy system prior to 2.2 and switched to slab allocation since then.
- **New in 9th Edition: Linux SLOB and SLUB allocators replace SLAB**
  - SLOB, Simple List of Blocks, maintains 3 linked lists of free blocks - small, medium, and large - designed for (imbedded) systems with limited amounts of memory.
  - SLUB modifies some implementation issues for better performance on systems with large numbers of processors.

Figure 9.27 - Slab allocation.

9.9 Other Considerations

9.9.1 Prepaging

- The basic idea behind *prepaging* is to predict the pages that will be needed in the near future, and page them in before they are actually requested.
- If a process was swapped out and we know what its working set was at the time, then when we swap it back in we can go ahead and page back in the entire working set, before the page faults actually occur.
- With small (data) files we can go ahead and prepage all of the pages at one time.
- Prepaging can be of benefit if the prediction is good and the pages are needed eventually, but slows the system down if the prediction is wrong.

9.9.2 Page Size

- There are quite a few trade-offs of small versus large page sizes:
- Small pages waste less memory due to internal fragmentation.
- Large pages require smaller page tables.
- For disk access, the latency and seek times greatly outweigh the actual data transfer times. This makes it much faster to transfer one large page of data than two or more smaller pages containing the same amount of data.
- Smaller pages match locality better, because we are not bringing in data that is not really needed.
- Small pages generate more page faults, with attending overhead.
- The physical hardware may also play a part in determining page size.
- It is hard to determine an "optimal" page size for any given system. Current norms range from 4K to 4M, and tend towards larger page sizes as time passes.

9.9.3 TLB Reach

- *TLB Reach* is defined as the amount of memory that can be reached by the pages listed in the TLB.
- Ideally the working set would fit within the reach of the TLB.
- Increasing the size of the TLB is an obvious way of increasing TLB reach, but TLB memory is very expensive and also draws lots of power.
- Increasing page sizes increases TLB reach, but also leads to increased fragmentation loss.
- Some systems provide multiple size pages to increase TLB reach while keeping fragmentation low.
- Multiple page sizes requires that the TLB be managed by software, not hardware.

9.9.4 Inverted Page Tables

- Inverted page tables store one entry for each frame instead of one entry for each virtual page. This reduces the memory requirement for the page table, but loses the information needed to implement virtual memory paging.
- A solution is to keep a separate page table for each process, for virtual memory management purposes. These are kept on disk, and only paged in when a page fault occurs. (I.e. they are not referenced with every memory access the way a traditional page table would be.)

9.9.5 Program Structure

- Consider a pair of nested loops to access every element in a 1024 x 1024 two-dimensional array of 32-bit ints.
- Arrays in C are stored in row-major order, which means that each row of the array would occupy a page of memory.
- If the loops are nested so that the outer loop increments the row and the inner loop increments the column, then an entire row can be processed before the next page fault, yielding 1024 page faults total.
- On the other hand, if the loops are nested the other way, so that the program worked down the columns instead of across the rows, then every access would be to a different page, yielding a new page fault for each access, or over a million page faults all together.
- Be aware that different languages store their arrays differently. FORTRAN for example stores arrays in column-major format instead of row-major. This means that blind translation of code from one language to another may turn a fast program into a very slow one, strictly because of the extra page faults.

9.9.6 I/O Interlock and Page Locking

There are several occasions when it may be desirable to **lock** pages in memory, and not let them get paged out:

- Certain kernel operations cannot tolerate having their pages swapped out.
- If an I/O controller is doing direct-memory access, it would be wrong to change pages in the middle of the I/O operation.
- In a priority based scheduling system, low priority jobs may need to wait quite a while before getting their turn on the CPU, and there is a danger of their pages being paged out before they get a chance to use them even once after paging them in. In this situation pages may be locked when they are paged in, until the process that requested them gets at least one turn in the CPU.
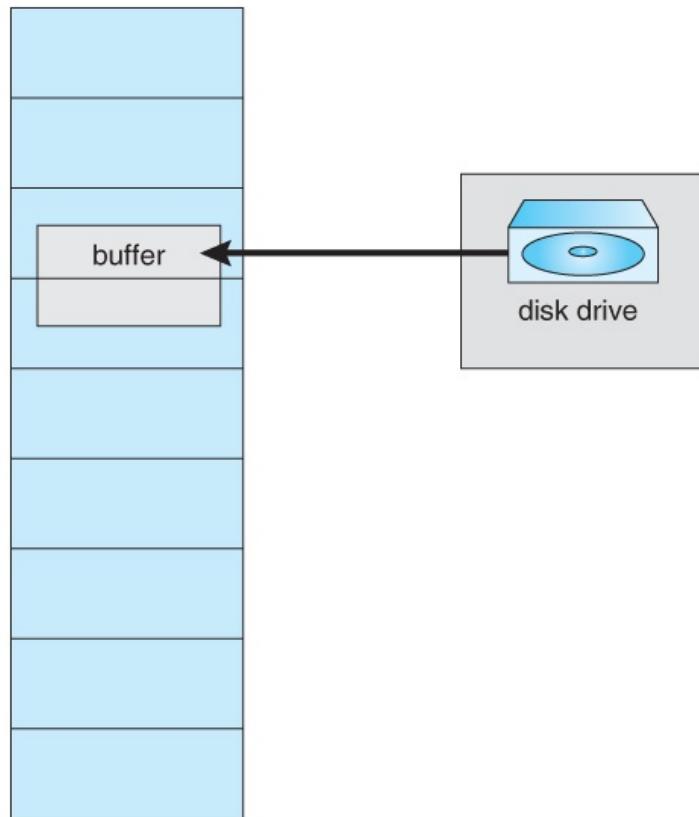

Figure 9.28 - The reason why frames used for I/O must be in memory.

9.10 Operating-System Examples (Optional)

9.10.1 Windows

- Windows uses demand paging with **clustering,** meaning they page in multiple pages whenever a page fault occurs.
- The working set minimum and maximum are normally set at 50 and 345 pages respectively. (Maximums can be exceeded in rare circumstances.)
- Free pages are maintained on a free list, with a minimum threshold indicating when there are enough free frames available.
- If a page fault occurs and the process is below their maximum, then additional pages are allocated. Otherwise some pages from this process must be replaced, using a local page replacement algorithm.
- If the amount of free frames falls below the allowable threshold, then **working set trimming** occurs, taking frames

away from any processes which are above their minimum, until all are at their minimums. Then additional frames can be allocated to processes that need them.

- The algorithm for selecting victim frames depends on the type of processor:
  - On single processor 80x86 systems, a variation of the clock (second chance) algorithm is used.
  - On Alpha and multiprocessor systems, clearing the reference bits may require invalidating entries in the TLB on other processors, which is an expensive operation. In this case Windows uses a variation of FIFO.

9.10.2 Solaris

- Solaris maintains a list of free pages, and allocates one to a faulting thread whenever a fault occurs. It is therefore imperative that a minimum amount of free memory be kept on hand at all times.
- Solaris has a parameter, *lotsfree,* usually set at 1/64 of total physical memory. Solaris checks 4 times per second to see if the free memory falls below this threshhold, and if it does, then the **pageout** process is started.
- Pageout uses a variation of the clock (second chance) algorithm, with two hands rotating around through the frame table. The first hand clears the reference bits, and the second hand comes by afterwards and checks them. Any frame whose reference bit has not been reset before the second hand gets there gets paged out.
- The Pageout method is adjustable by the distance between the two hands, (the **handspan**), and the speed at which the hands move. For example, if the hands each check 100 frames per second, and the handspan is 1000 frames, then there would be a 10 second interval between the time when the leading hand clears the reference bits and the time when the trailing hand checks them.
- The speed of the hands is usually adjusted according to the amount of free memory, as shown below. **Slowscan** is usually set at 100 pages per second, and **fastscan** is usually set at the smaller of 1/2 of the total physical pages per second and 8192 pages per second.
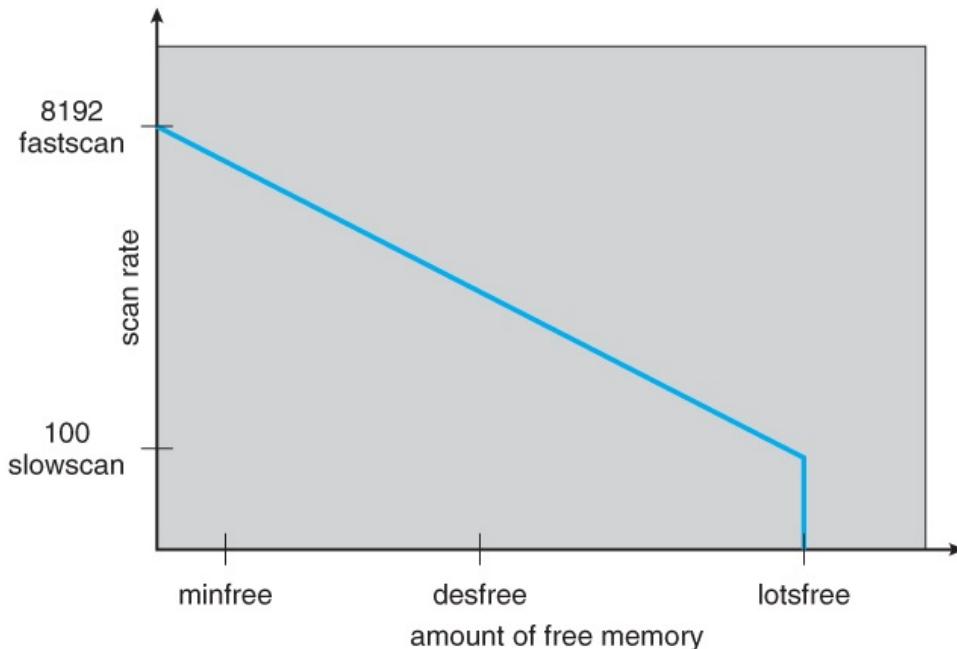

Figure 9.29 - Solaris page scanner.

- Solaris also maintains a cache of pages that have been reclaimed but which have not yet been overwritten, as opposed to the free list which only holds pages whose current contents are invalid. If one of the pages from the cache is needed before it gets moved to the free list, then it can be quickly recovered.
- Normally pageout runs 4 times per second to check if memory has fallen below *lotsfree.* However if it falls below *desfree,* then pageout will run at 100 times per second in an attempt to keep at least desfree pages free. If it is unable to do this for a 30-second average, then Solaris begins swapping processes, starting preferably with processes that have been idle for a long time.
- If free memory falls below *minfree,* then pageout runs with every page fault.
- Recent releases of Solaris have enhanced the virtual memory management system, including recognizing pages from shared libraries, and protecting them from being paged out.

9.11 Summary

Mass-Storage Structure

References:

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 10 (Was chapter 12)

10.1 Overview of Mass-Storage Structure

10.1.1 Magnetic Disks

- Traditional magnetic disks have the following basic structure:
    - One or more **platters** in the form of disks covered with magnetic media. **Hard disk** platters are made of rigid metal, while "**floppy**" disks are made of more flexible plastic.
    - Each platter has two working **surfaces.** Older hard disk drives would sometimes not use the very top or bottom surface of a stack of platters, as these surfaces were more susceptible to potential damage.
    - Each working surface is divided into a number of concentric rings called **tracks.** The collection of all tracks that are the same distance from the edge of the platter, (i.e. all tracks immediately above one another in the following diagram) is called a **cylinder**.
    - Each track is further divided into **sectors,** traditionally containing 512 bytes of data each, although some modern disks occasionally use larger sector sizes. (Sectors also include a header and a trailer, including checksum information among other things. Larger sector sizes reduce the fraction of the disk consumed by headers and trailers, but increase internal fragmentation and the amount of disk that must be marked bad in the case of errors.)
    - The data on a hard drive is read by read-write **heads.** The standard configuration (shown below) uses one head per surface, each on a separate **arm**, and controlled by a common **arm assembly** which moves all heads simultaneously from one cylinder to another. (Other configurations, including independent read-write heads, may speed up disk access, but involve serious technical difficulties.)
    - The storage capacity of a traditional disk drive is equal to the number of heads (i.e. the number of working surfaces), times the number of tracks per surface, times the number of sectors per track, times the number of bytes per sector. A particular physical block of data is specified by providing the head-sector-cylinder number at which it is located.
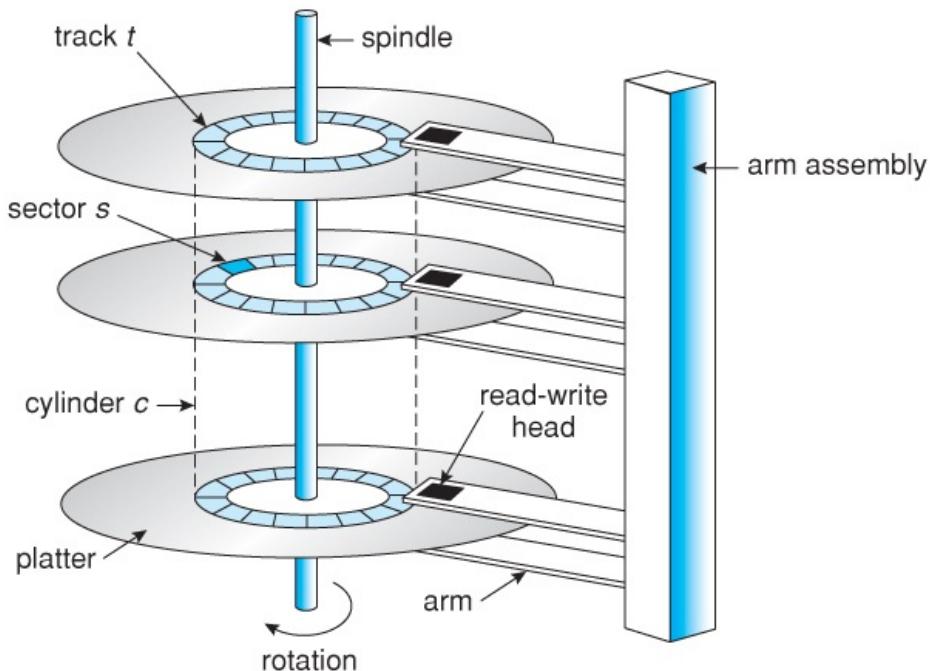

Figure 10.1 - Moving-head disk mechanism.

- In operation the disk rotates at high speed, such as 7200 rpm (120 revolutions per second.) The rate at which data can be transferred from the disk to the computer is composed of several steps:
    - The **positioning time**, a.k.a. the **seek time** or **random access time** is the time required to move the heads from one cylinder to another, and for the heads to settle down after the move. This is typically the slowest step in the process and the predominant bottleneck to overall transfer rates.
    - The **rotational latency** is the amount of time required for the desired sector to rotate around and come under the read-write head.This can range anywhere from zero to one full revolution, and on the average

will equal one-half revolution. This is another physical step and is usually the second slowest step behind seek time. (For a disk rotating at 7200 rpm, the average rotational latency would be 1/2 revolution / 120 revolutions per second, or just over 4 milliseconds, a long time by computer standards.

- ○ The *transfer rate*, which is the time required to move the data electronically from the disk to the computer. (Some authors may also use the term transfer rate to refer to the overall transfer rate, including seek time and rotational latency as well as the electronic data transfer rate.)

- Disk heads "fly" over the surface on a very thin cushion of air. If they should accidentally contact the disk, then a *head crash* occurs, which may or may not permanently damage the disk or even destroy it completely. For this reason it is normal to *park* the disk heads when turning a computer off, which means to move the heads off the disk or to an area of the disk where there is no data stored.
- Floppy disks are normally *removable*. Hard drives can also be removable, and some are even *hot-swappable*, meaning they can be removed while the computer is running, and a new hard drive inserted in their place.
- Disk drives are connected to the computer via a cable known as the *I/O Bus.* Some of the common interface formats include Enhanced Integrated Drive Electronics, EIDE; Advanced Technology Attachment, ATA; Serial ATA, SATA, Universal Serial Bus, USB; Fiber Channel, FC, and Small Computer Systems Interface, SCSI.
- The *host controller* is at the computer end of the I/O bus, and the *disk controller* is built into the disk itself. The CPU issues commands to the host controller via I/O ports. Data is transferred between the magnetic surface and onboard *cache* by the disk controller, and then the data is transferred from that cache to the host controller and the motherboard memory at electronic speeds.

10.1.2 Solid-State Disks - New

- As technologies improve and economics change, old technologies are often used in different ways. One example of this is the increasing used of *solid state disks, or SSDs.*
- SSDs use memory technology as a small fast hard disk. Specific implementations may use either flash memory or DRAM chips protected by a battery to sustain the information through power cycles.
- Because SSDs have no moving parts they are much faster than traditional hard drives, and certain problems such as the scheduling of disk accesses simply do not apply.
- However SSDs also have their weaknesses: They are more expensive than hard drives, generally not as large, and may have shorter life spans.
- SSDs are especially useful as a high-speed cache of hard-disk information that must be accessed quickly. One example is to store filesystem meta-data, e.g. directory and inode information, that must be accessed quickly and often. Another variation is a boot disk containing the OS and some application executables, but no vital user data. SSDs are also used in laptops to make them smaller, faster, and lighter.
- Because SSDs are so much faster than traditional hard disks, the throughput of the bus can become a limiting factor, causing some SSDs to be connected directly to the system PCI bus for example.

10.1.3 Magnetic Tapes - was 12.1.2

- Magnetic tapes were once used for common secondary storage before the days of hard disk drives, but today are used primarily for backups.
- Accessing a particular spot on a magnetic tape can be slow, but once reading or writing commences, access speeds are comparable to disk drives.
- Capacities of tape drives can range from 20 to 200 GB, and compression can double that capacity.

10.2 Disk Structure

- The traditional head-sector-cylinder, HSC numbers are mapped to linear block addresses by numbering the first sector on the first head on the outermost track as sector 0. Numbering proceeds with the rest of the sectors on that same track, and then the rest of the tracks on the same cylinder before proceeding through the rest of the cylinders to the center of the disk. In modern practice these linear block addresses are used in place of the HSC numbers for a variety of reasons:
  1. The linear length of tracks near the outer edge of the disk is much longer than for those tracks located near the center, and therefore it is possible to squeeze many more sectors onto outer tracks than onto inner ones.
  2. All disks have some bad sectors, and therefore disks maintain a few spare sectors that can be used in place of the bad ones. The mapping of spare sectors to bad sectors in managed internally to the disk controller.
  3. Modern hard drives can have thousands of cylinders, and hundreds of sectors per track on their outermost tracks. These numbers exceed the range of HSC numbers for many (older) operating systems, and therefore disks can be configured for any convenient combination of HSC values that falls within the total number of sectors physically on the drive.
- There is a limit to how closely packed individual bits can be placed on a physical media, but that limit is growing increasingly more packed as technological advances are made.
- Modern disks pack many more sectors into outer cylinders than inner ones, using one of two approaches:
  - ○ With *Constant Linear Velocity, CLV,* the density of bits is uniform from cylinder to cylinder. Because there are more sectors in outer cylinders, the disk spins slower when reading those cylinders, causing the rate of bits passing under

the read-write head to remain constant. This is the approach used by modern CDs and DVDs.
   - With **Constant Angular Velocity, CAV,** the disk rotates at a constant angular speed, with the bit density decreasing on outer cylinders. (These disks would have a constant number of sectors per track on all cylinders.)

10.3 Disk Attachment

Disk drives can be attached either directly to a particular host (a local disk) or to a network.

10.3.1 Host-Attached Storage

- Local disks are accessed through I/O Ports as described earlier.
- The most common interfaces are IDE or ATA, each of which allow up to two drives per host controller.
- SATA is similar with simpler cabling.
- High end workstations or other systems in need of larger number of disks typically use SCSI disks:
   - The SCSI standard supports up to 16 **targets** on each SCSI bus, one of which is generally the host adapter and the other 15 of which can be disk or tape drives.
   - A SCSI target is usually a single drive, but the standard also supports up to 8 **units** within each target. These would generally be used for accessing individual disks within a RAID array. (See below.)
   - The SCSI standard also supports multiple host adapters in a single computer, i.e. multiple SCSI busses.
   - Modern advancements in SCSI include "fast" and "wide" versions, as well as SCSI-2.
   - SCSI cables may be either 50 or 68 conductors. SCSI devices may be external as well as internal.
   - See wikipedia for more information on the SCSI interface.
- FC is a high-speed serial architecture that can operate over optical fiber or four-conductor copper wires, and has two variants:
   - A large switched fabric having a 24-bit address space. This variant allows for multiple devices and multiple hosts to interconnect, forming the basis for the **storage-area networks, SANs,** to be discussed in a future section.
   - The **arbitrated loop, FC-AL,** that can address up to 126 devices (drives and controllers.)

10.3.2 Network-Attached Storage

- Network attached storage connects storage devices to computers using a remote procedure call, RPC, interface, typically with something like NFS filesystem mounts. This is convenient for allowing several computers in a group common access and naming conventions for shared storage.
- NAS can be implemented using SCSI cabling, or **ISCSI** uses Internet protocols and standard network connections, allowing long-distance remote access to shared files.
- NAS allows computers to easily share data storage, but tends to be less efficient than standard host-attached storage.
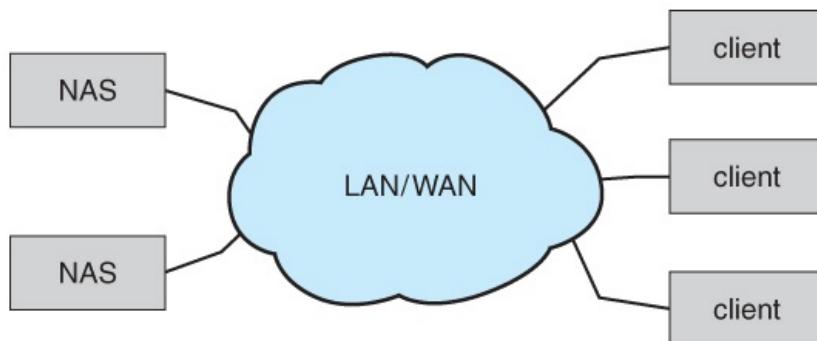

Figure 10.2 - Network-attached storage.

10.3.3 Storage-Area Network

- A **Storage-Area Network, SAN,** connects computers and storage devices in a network, using storage protocols instead of network protocols.
- One advantage of this is that storage access does not tie up regular networking bandwidth.
- SAN is very flexible and dynamic, allowing hosts and devices to attach and detach on the fly.
- SAN is also controllable, allowing restricted access to certain hosts and devices.
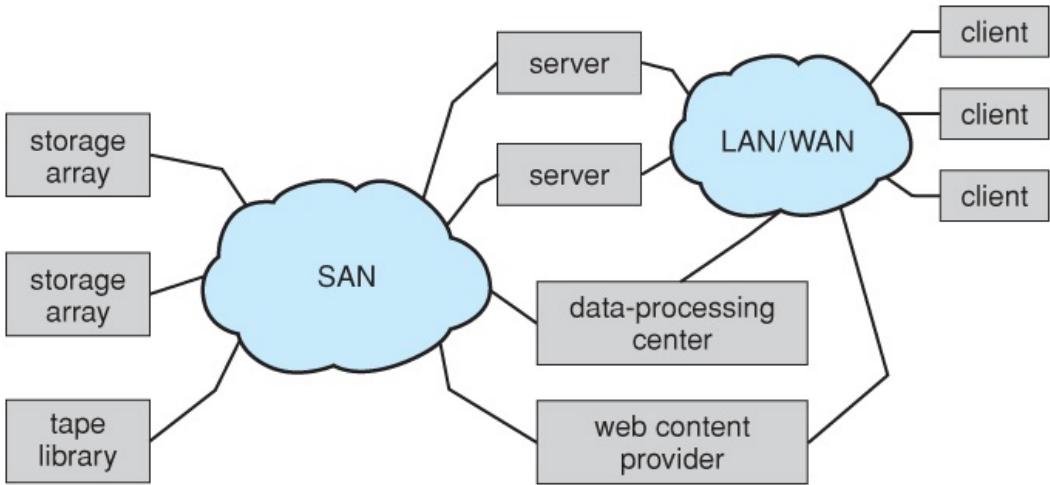
Figure 10.3 - Storage-area network.

10.4 Disk Scheduling

- As mentioned earlier, disk transfer speeds are limited primarily by **seek times** and **rotational latency.** When multiple requests are to be processed there is also some inherent delay in waiting for other requests to be processed.
- **Bandwidth** is measured by the amount of data transferred divided by the total amount of time from the first request being made to the last transfer being completed, (for a series of disk requests.)
- Both bandwidth and access time can be improved by processing requests in a good order.
- Disk requests include the disk address, memory address, number of sectors to transfer, and whether the request is for reading or writing.

10.4.1 FCFS Scheduling

- **First-Come First-Serve** is simple and intrinsically fair, but not very efficient. Consider in the following sequence the wild swing from cylinder 122 to 14 and then back to 124:
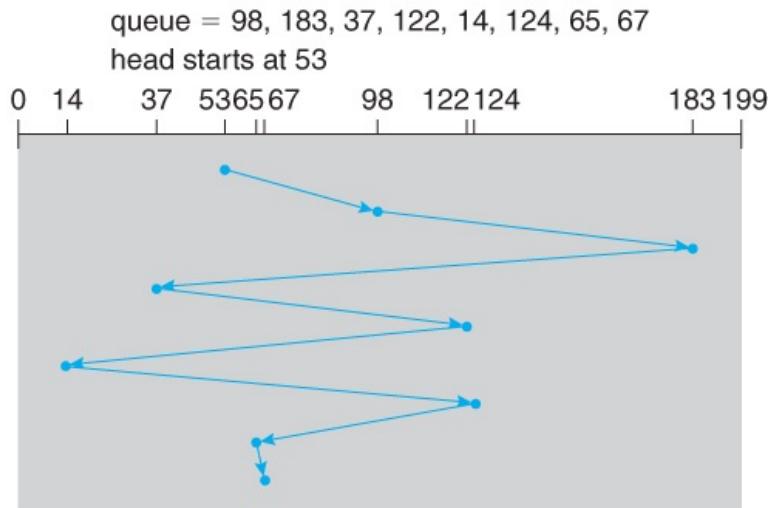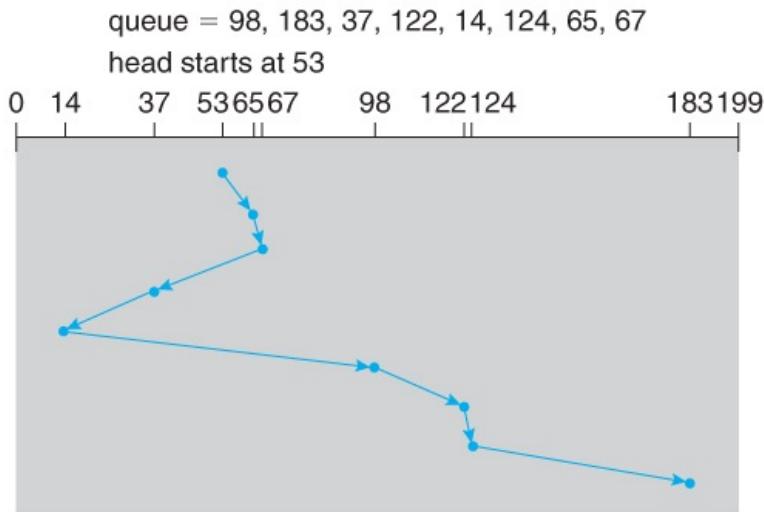
Figure 10.4 - FCFS disk scheduling.

10.4.2 SSTF Scheduling
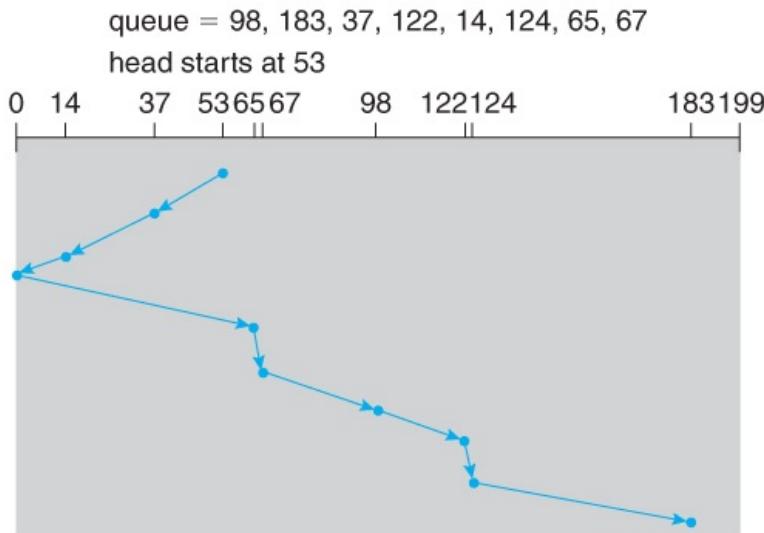
- **Shortest Seek Time First** scheduling is more efficient, but may lead to starvation if a constant stream of requests arrives for the same general area of the disk.
- SSTF reduces the total head movement to 236 cylinders, down from 640 required for the same set of requests under FCFS. Note, however that the distance could be reduced still further to 208 by starting with 37 and then 14 first before processing the rest of the requests.

Figure 10.5 - SSTF disk scheduling.

10.4.3 SCAN Scheduling

- The **SCAN** algorithm, a.k.a. the **elevator** algorithm moves back and forth from one end of the disk to the other, similarly to an elevator processing requests in a tall building.
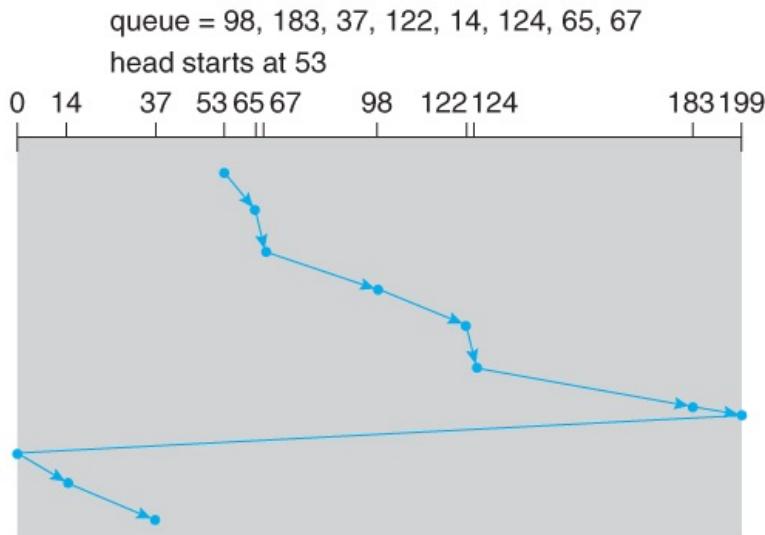

Figure 10.6 - SCAN disk scheduling.

- Under the SCAN algorithm, If a request arrives just ahead of the moving head then it will be processed right away, but if it arrives just after the head has passed, then it will have to wait for the head to pass going the other way on the return trip. This leads to a fairly wide variation in access times which can be improved upon.
- Consider, for example, when the head reaches the high end of the disk: Requests with high cylinder numbers just missed the passing head, which means they are all fairly recent requests, whereas requests with low numbers may have been waiting for a much longer time. Making the return scan from high to low then ends up accessing recent requests first and making older requests wait that much longer.

10.4.4 C-SCAN Scheduling
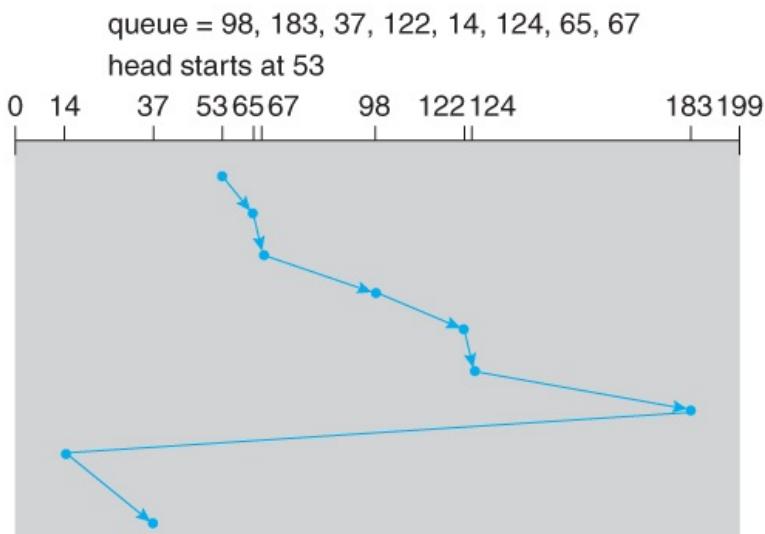
- The **Circular-SCAN** algorithm improves upon SCAN by treating all requests in a circular queue fashion - Once the head reaches the end of the disk, it returns to the other end without processing any requests, and then starts again from the beginning of the disk:

Figure 10.7 - C-SCAN disk scheduling.

12.4.5 LOOK Scheduling

- *LOOK* scheduling improves upon SCAN by looking ahead at the queue of pending requests, and not moving the heads any farther towards the end of the disk than is necessary. The following diagram illustrates the circular form of LOOK:


Figure 10.8 - C-LOOK disk scheduling.

10.4.6 Selection of a Disk-Scheduling Algorithm

- With very low loads all algorithms are equal, since there will normally only be one request to process at a time.
- For slightly larger loads, SSTF offers better performance than FCFS, but may lead to starvation when loads become heavy enough.
- For busier systems, SCAN and LOOK algorithms eliminate starvation problems.
- The actual optimal algorithm may be something even more complex than those discussed here, but the incremental improvements are generally not worth the additional overhead.
- Some improvement to overall filesystem access times can be made by intelligent placement of directory and/or inode information. If those structures are placed in the middle of the disk instead of at the beginning of the disk, then the maximum distance from those structures to data blocks is reduced to only one-half of the disk size. If those structures can be further distributed and furthermore have their data blocks stored as close as possible to the corresponding directory structures, then that reduces still further the overall time to find the disk block numbers and then access the corresponding data blocks.
- On modern disks the rotational latency can be almost as significant as the seek time, however it is not within the OSes control to account for that, because modern disks do not reveal their internal sector mapping schemes, (particularly when bad blocks have been remapped to spare sectors.)
  - Some disk manufacturers provide for disk scheduling algorithms directly on their disk controllers, (which do know the actual geometry of the disk as well as any remapping), so that if a series of requests are sent from the computer to the controller then those requests can be processed in an optimal order.

- Unfortunately there are some considerations that the OS must take into account that are beyond the abilities of the on-board disk-scheduling algorithms, such as priorities of some requests over others, or the need to process certain requests in a particular order. For this reason OSes may elect to spoon-feed requests to the disk controller one at a time in certain situations.
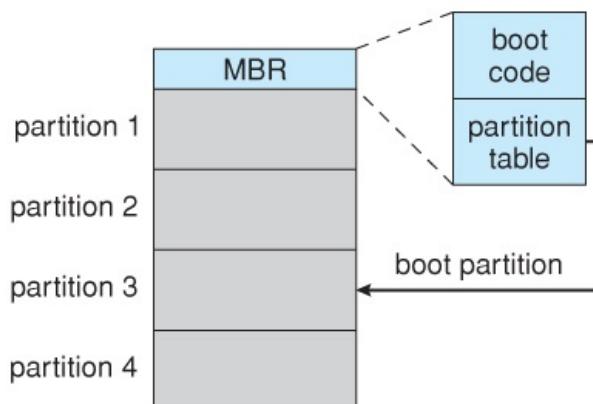
10.5 Disk Management

105.1 Disk Formatting

- Before a disk can be used, it has to be **low-level formatted**, which means laying down all of the headers and trailers marking the beginning and ends of each sector. Included in the header and trailer are the linear sector numbers, and **error-correcting codes, ECC,** which allow damaged sectors to not only be detected, but in many cases for the damaged data to be recovered (depending on the extent of the damage.) Sector sizes are traditionally 512 bytes, but may be larger, particularly in larger drives.
- ECC calculation is performed with every disk read or write, and if damage is detected but the data is recoverable, then a **soft error** has occurred. Soft errors are generally handled by the on-board disk controller, and never seen by the OS. (See below.)
- Once the disk is low-level formatted, the next step is to partition the drive into one or more separate partitions. This step must be completed even if the disk is to be used as a single large partition, so that the partition table can be written to the beginning of the disk.
- After partitioning, then the filesystems must be **logically formatted,** which involves laying down the master directory information (FAT table or inode structure), initializing free lists, and creating at least the root directory of the filesystem. (Disk partitions which are to be used as raw devices are not logically formatted. This saves the overhead and disk space of the filesystem structure, but requires that the application program manage its own disk storage requirements.)

10.5.2 Boot Block

- Computer ROM contains a **bootstrap** program (OS independent) with just enough code to find the first sector on the first hard drive on the first controller, load that sector into memory, and transfer control over to it. (The ROM bootstrap program may look in floppy and/or CD drives before accessing the hard drive, and is smart enough to recognize whether it has found valid boot code or not.)
- The first sector on the hard drive is known as the **Master Boot Record, MBR,** and contains a very small amount of code in addition to the **partition table.** The partition table documents how the disk is partitioned into logical disks, and indicates specifically which partition is the **active** or **boot** partition.
- The boot program then looks to the active partition to find an operating system, possibly loading up a slightly larger / more advanced boot program along the way.
- In a **dual-boot** (or larger multi-boot) system, the user may be given a choice of which operating system to boot, with a default action to be taken in the event of no response within some time frame.
- Once the kernel is found by the boot program, it is loaded into memory and then control is transferred over to the OS. The kernel will normally continue the boot process by initializing all important kernel data structures, launching important system services (e.g. network daemons, sched, init, etc.), and finally providing one or more login prompts. Boot options at this stage may include **single-user** a.k.a. **maintenance** or **safe** modes, in which very few system services are started - These modes are designed for system administrators to repair problems or otherwise maintain the system.


Figure 10.9 - Booting from disk in Windows 2000.

10.5.3 Bad Blocks

- No disk can be manufactured to 100% perfection, and all physical objects wear out over time. For these reasons

all disks are shipped with a few bad blocks, and additional blocks can be expected to go bad slowly over time. If a large number of blocks go bad then the entire disk will need to be replaced, but a few here and there can be handled through other means.

- In the old days, bad blocks had to be checked for manually. Formatting of the disk or running certain disk-analysis tools would identify bad blocks, and attempt to read the data off of them one last time through repeated tries. Then the bad blocks would be mapped out and taken out of future service. Sometimes the data could be recovered, and sometimes it was lost forever. (Disk analysis tools could be either destructive or non-destructive.)
- Modern disk controllers make much better use of the error-correcting codes, so that bad blocks can be detected earlier and the data usually recovered. (Recall that blocks are tested with every write as well as with every read, so often errors can be detected before the write operation is complete, and the data simply written to a different sector instead.)
- Note that re-mapping of sectors from their normal linear progression can throw off the disk scheduling optimization of the OS, especially if the replacement sector is physically far away from the sector it is replacing. For this reason most disks normally keep a few spare sectors on each cylinder, as well as at least one spare cylinder. Whenever possible a bad sector will be mapped to another sector on the same cylinder, or at least a cylinder as close as possible. **Sector slipping** may also be performed, in which all sectors between the bad sector and the replacement sector are moved down by one, so that the linear progression of sector numbers can be maintained.
- If the data on a bad block cannot be recovered, then a **hard error** has occurred., which requires replacing the file(s) from backups, or rebuilding them from scratch.

10.6 Swap-Space Management

- Modern systems typically swap out pages as needed, rather than swapping out entire processes. Hence the swapping system is part of the virtual memory management system.
- Managing swap space is obviously an important task for modern OSes.

10.6.1 Swap-Space Use

- The amount of swap space needed by an OS varies greatly according to how it is used. Some systems require an amount equal to physical RAM; some want a multiple of that; some want an amount equal to the amount by which virtual memory exceeds physical RAM, and some systems use little or none at all!
- Some systems support multiple swap spaces on separate disks in order to speed up the virtual memory system.

10.6.2 Swap-Space Location

Swap space can be physically located in one of two locations:

- As a large file which is part of the regular filesystem. This is easy to implement, but inefficient. Not only must the swap space be accessed through the directory system, the file is also subject to fragmentation issues. Caching the block location helps in finding the physical blocks, but that is not a complete fix.
- As a raw partition, possibly on a separate or little-used disk. This allows the OS more control over swap space management, which is usually faster and more efficient. Fragmentation of swap space is generally not a big issue, as the space is re-initialized every time the system is rebooted. The downside of keeping swap space on a raw partition is that it can only be grown by repartitioning the hard drive.

12.6.3 Swap-Space Management: An Example

- Historically OSes swapped out entire processes as needed. Modern systems swap out only individual pages, and only as needed. (For example process code blocks and other blocks that have not been changed since they were originally loaded are normally just freed from the virtual memory system rather than copying them to swap space, because it is faster to go find them again in the filesystem and read them back in from there than to write them out to swap space and then read them back.)
- In the mapping system shown below for Linux systems, a map of swap space is kept in memory, where each entry corresponds to a 4K block in the swap space. Zeros indicate free slots and non-zeros refer to how many processes have a mapping to that particular block (>1 for shared pages only.)
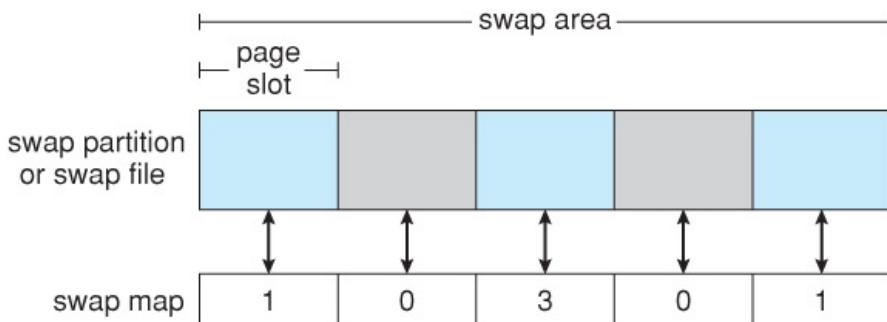
Figure 10.10 - The data structures for swapping on Linux systems.

10.7 RAID Structure

- The general idea behind RAID is to employ a group of hard drives together with some form of duplication, either to increase reliability or to speed up operations, (or sometimes both.)
- **RAID** originally stood for **Redundant Array of Inexpensive Disks,** and was designed to use a bunch of cheap small disks in place of one or two larger more expensive ones. Today RAID systems employ large possibly expensive disks as their components, switching the definition to **Independent** disks.

10.7.1 Improvement of Reliability via Redundancy

- The more disks a system has, the greater the likelihood that one of them will go bad at any given time. Hence increasing disks on a system actually **decreases** the **Mean Time To Failure, MTTF** of the system.
- If, however, the same data was copied onto multiple disks, then the data would not be lost unless **both** (or all) copies of the data were damaged simultaneously, which is a **MUCH** lower probability than for a single disk going bad. More specifically, the second disk would have to go bad before the first disk was repaired, which brings the **Mean Time To Repair** into play. For example if two disks were involved, each with a MTTF of 100,000 hours and a MTTR of 10 hours, then the **Mean Time to Data Loss** would be 500 * 10^6 hours, or 57,000 years!
- This is the basic idea behind disk **mirroring**, in which a system contains identical data on two or more disks.
  - Note that a power failure during a write operation could cause both disks to contain corrupt data, if both disks were writing simultaneously at the time of the power failure. One solution is to write to the two disks in series, so that they will not both become corrupted (at least not in the same way) by a power failure. And alternate solution involves non-volatile RAM as a write cache, which is not lost in the event of a power failure and which is protected by error-correcting codes.

10.7.2 Improvement in Performance via Parallelism

- There is also a performance benefit to mirroring, particularly with respect to reads. Since every block of data is duplicated on multiple disks, read operations can be satisfied from any available copy, and multiple disks can be reading different data blocks simultaneously in parallel. (Writes could possibly be sped up as well through careful scheduling algorithms, but it would be complicated in practice.)
- Another way of improving disk access time is with **striping**, which basically means spreading data out across multiple disks that can be accessed simultaneously.
  - With **bit-level striping** the bits of each byte are striped across multiple disks. For example if 8 disks were involved, then each 8-bit byte would be read in parallel by 8 heads on separate disks. A single disk read would access 8 * 512 bytes = 4K worth of data in the time normally required to read 512 bytes. Similarly if 4 disks were involved, then two bits of each byte could be stored on each disk, for 2K worth of disk access per read or write operation.
  - **Block-level striping** spreads a filesystem across multiple disks on a block-by-block basis, so if block N were located on disk 0, then block N + 1 would be on disk 1, and so on. This is particularly useful when filesystems are accessed in **clusters** of physical blocks. Other striping possibilities exist, with block-level striping being the most common.

10.7.3 RAID Levels

- Mirroring provides reliability but is expensive; Striping improves performance, but does not improve reliability. Accordingly there are a number of different schemes that combine the principals of mirroring and striping in different ways, in order to balance reliability versus performance versus cost. These are described by different **RAID levels**, as follows: (In the diagram that follows, "C" indicates a copy, and "P" indicates parity, i.e. checksum bits.)
  0. **Raid Level 0 -** This level includes striping only, with no mirroring.
  1. **Raid Level 1 -** This level includes mirroring only, no striping.

2. ***Raid Level 2 -*** This level stores error-correcting codes on additional disks, allowing for any damaged data to be reconstructed by subtraction from the remaining undamaged data. Note that this scheme requires only three extra disks to protect 4 disks worth of data, as opposed to full mirroring. (The number of disks required is a function of the error-correcting algorithms, and the means by which the particular bad bit(s) is(are) identified.)

3. ***Raid Level 3 -*** This level is similar to level 2, except that it takes advantage of the fact that each disk is still doing its own error-detection, so that when an error occurs, there is no question about which disk in the array has the bad data. As a result a single parity bit is all that is needed to recover the lost data from an array of disks. Level 3 also includes striping, which improves performance. The downside with the parity approach is that every disk must take part in every disk access, and the parity bits must be constantly calculated and checked, reducing performance. Hardware-level parity calculations and NVRAM cache can help with both of those issues. In practice level 3 is greatly preferred over level 2.

4. ***Raid Level 4 -*** This level is similar to level 3, employing block-level striping instead of bit-level striping. The benefits are that multiple blocks can be read independently, and changes to a block only require writing two blocks (data and parity) rather than involving all disks. Note that new disks can be added seamlessly to the system provided they are initialized to all zeros, as this does not affect the parity results.

5. ***Raid Level 5 -*** This level is similar to level 4, except the parity blocks are distributed over all disks, thereby more evenly balancing the load on the system. For any given block on the disk(s), one of the disks will hold the parity information for that block and the other N-1 disks will hold the data. Note that the same disk cannot hold both data and parity for the same block, as both would be lost in the event of a disk crash.

6. ***Raid Level 6 -*** This level extends raid level 5 by storing multiple bits of error-recovery codes, (such as the **Reed-Solomon codes**), for each bit position of data, rather than a single parity bit. In the example shown below 2 bits of ECC are stored for every 4 bits of data, allowing data recovery in the face of up to two simultaneous disk failures. Note that this still involves only 50% increase in storage needs, as opposed to 100% for simple mirroring which could only tolerate a single disk failure.
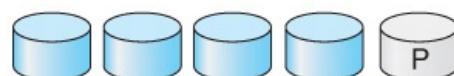
(a) RAID 0: non-redundant striping.
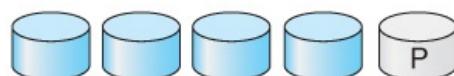
(b) RAID 1: mirrored disks.

(c) RAID 2: memory-style error-correcting codes.

(d) RAID 3: bit-interleaved parity.

(e) RAID 4: block-interleaved parity.
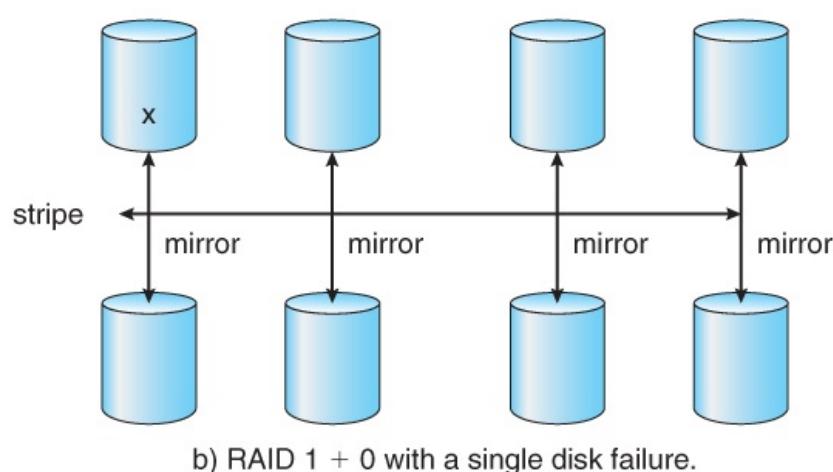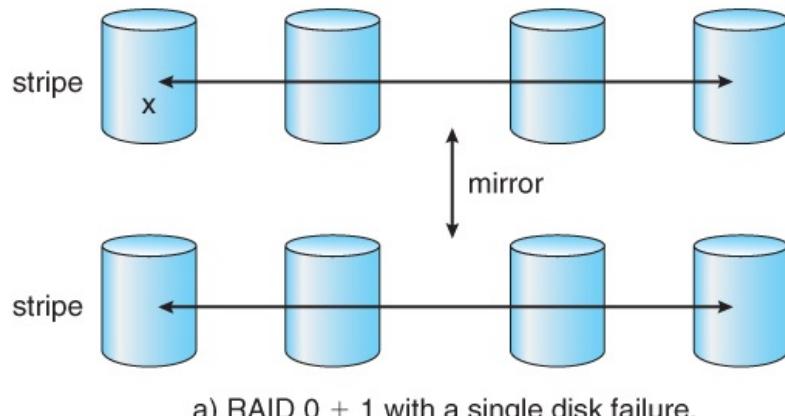
(f) RAID 5: block-interleaved distributed parity.

(g) RAID 6: P + Q redundancy.

Figure 10.11 - RAID levels.

- There are also two RAID levels which combine RAID levels 0 and 1 (striping and mirroring) in different combinations, designed to provide both performance and reliability at the expense of increased cost.
  - **RAID level 0 + 1** disks are first striped, and then the striped disks mirrored to another set. This level generally provides better performance than RAID level 5.
  - **RAID level 1 + 0** mirrors disks in pairs, and then stripes the mirrored pairs. The storage capacity, performance, etc. are all the same, but there is an advantage to this approach in the event of multiple disk failures, as illustrated below:.
    - In diagram (a) below, the 8 disks have been divided into two sets of four, each of which is striped, and then one stripe set is used to mirror the other set.
      - If a single disk fails, it wipes out the entire stripe set, but the system can keep on functioning using the remaining set.
      - However if a second disk from the other stripe set now fails, then the entire system is lost, as a result of two disk failures.
    - In diagram (b), the same 8 disks are divided into four sets of two, each of which is mirrored, and then the file system is striped across the four sets of mirrored disks.
      - If a single disk fails, then that mirror set is reduced to a single disk, but the system rolls on, and the other three mirror sets continue mirroring.
      - Now if a second disk fails, (that is not the mirror of the already failed disk), then another one of the mirror sets is reduced to a single disk, but the system can continue without data loss.
      - In fact the second arrangement could handle as many as four simultaneously failed disks, as long as no two of them were from the same mirror pair.
  - See the wikipedia article on nested raid levels for more information.
  - Here's a better explanation: http://www.storagereview.com/guide2000/ref/hdd/perf/raid/levels/multXY.html

a) RAID 0 + 1 with a single disk failure.

b) RAID 1 + 0 with a single disk failure.

Figure 10.12 - RAID 0 + 1 and 1 + 0

10.7.4 Selecting a RAID Level

- Trade-offs in selecting the optimal RAID level for a particular application include cost, volume of data, need for reliability, need for performance, and rebuild time, the latter of which can affect the likelihood that a second disk will fail while the first failed disk is being rebuilt.
- Other decisions include how many disks are involved in a RAID set and how many disks to protect with a single parity bit. More disks in the set increases performance but increases cost. Protecting more disks per parity bit

saves cost, but increases the likelihood that a second disk will fail before the first bad disk is repaired.

10.7.5 Extensions

- RAID concepts have been extended to tape drives (e.g. striping tapes for faster backups or parity checking tapes for reliability), and for broadcasting of data.

10.7.6 Problems with RAID

- RAID protects against physical errors, but not against any number of bugs or other errors that could write erroneous data.
- ZFS adds an extra level of protection by including data block checksums in all inodes along with the pointers to the data blocks. If data are mirrored and one copy has the correct checksum and the other does not, then the data with the bad checksum will be replaced with a copy of the data with the good checksum. This increases reliability greatly over RAID alone, at a cost of a performance hit that is acceptable because ZFS is so fast to begin with.
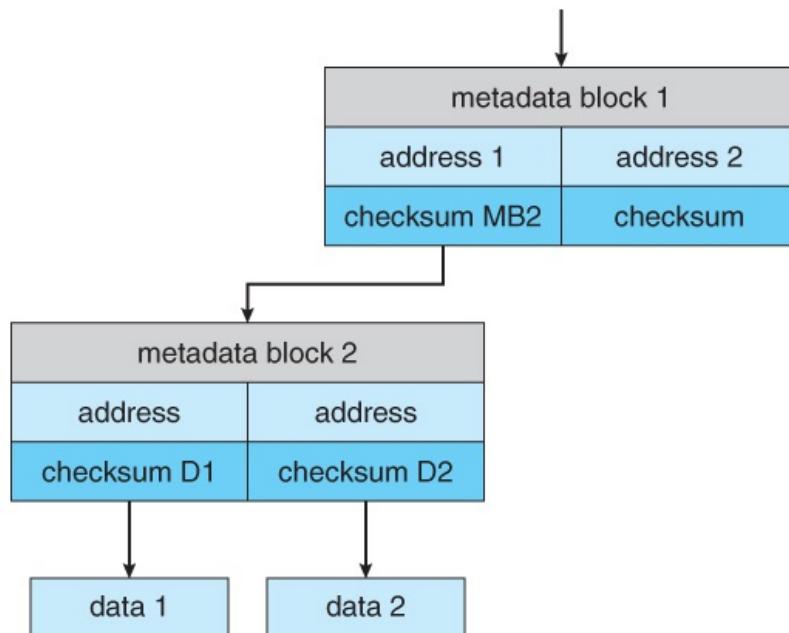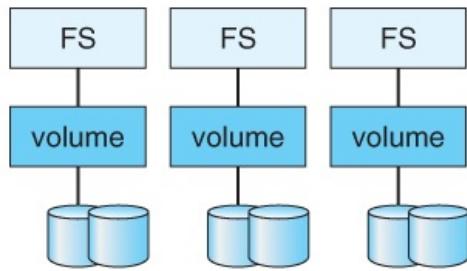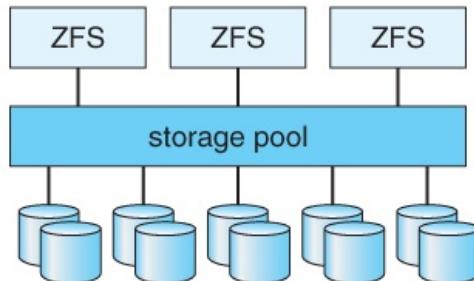
Figure 10.13 - ZFS checksums all metadata and data.

- Another problem with traditional filesystems is that the sizes are fixed, and relatively difficult to change. Where RAID sets are involved it becomes even harder to adjust filesystem sizes, because a filesystem cannot span across multiple filesystems.
- ZFS solves these problems by pooling RAID sets, and by dynamically allocating space to filesystems as needed. Filesystem sizes can be limited by quotas, and space can also be reserved to guarantee that a filesystem will be able to grow later, but these parameters can be changed at any time by the filesystem's owner. Otherwise filesystems grow and shrink dynamically as needed.

(a) Traditional volumes and file systems.

(b) ZFS and pooled storage.

Figure 10.14 - (a) Traditional volumes and file systems. (b) a ZFS pool and file systems.

10.8 Stable-Storage Implementation (Optional)

- The concept of stable storage (first presented in chapter 6) involves a storage medium in which data is **never** lost, even in the face of equipment failure in the middle of a write operation.
- To implement this requires two (or more) copies of the data, with separate failure modes.
- An attempted disk write results in one of three possible outcomes:
    1. The data is successfully and completely written.
    2. The data is partially written, but not completely. The last block written may be garbled.
    3. No writing takes place at all.
- Whenever an equipment failure occurs during a write, the system must detect it, and return the system back to a consistent state. To do this requires two physical blocks for every logical block, and the following procedure:
    1. Write the data to the first physical block.
    2. After step 1 had completed, then write the data to the second physical block.
    3. Declare the operation complete only after both physical writes have completed successfully.
- During recovery the pair of blocks is examined.
    - If both blocks are identical and there is no sign of damage, then no further action is necessary.
    - If one block contains a detectable error but the other does not, then the damaged block is replaced with the good copy. (This will either undo the operation or complete the operation, depending on which block is damaged and which is undamaged.)
    - If neither block shows damage but the data in the blocks differ, then replace the data in the first block with the data in the second block. (Undo the operation.)
- Because the sequence of operations described above is slow, stable storage usually includes NVRAM as a cache, and declares a write operation complete once it has been written to the NVRAM.

10.9 Summary

Was 12.9 Tertiary-Storage Structure - Optional, Omitted from Ninth Edition

- Primary storage refers to computer memory chips; Secondary storage refers to fixed-disk storage systems (hard drives); And **Tertiary Storage** refers to **removable media,** such as tape drives, CDs, DVDs, and to a lesser extend floppies, thumb drives, and other detachable devices.
- Tertiary storage is typically characterized by large capacity, low cost per MB, and slow access times, although there are exceptions in any of these categories.
- Tertiary storage is typically used for backups and for long-term archival storage of completed work. Another common use for tertiary storage is to swap large little-used files (or groups of files) off of the hard drive, and then swap them back in as needed in a fashion similar to secondary storage providing swap space for primary storage. (Review The Paging Game, note 5).

12.9.1 Tertiary-Storage Devices

12.9.1.1 Removable Disks

- Removable magnetic disks (e.g. floppies) can be nearly as fast as hard drives, but are at greater risk for damage due to scratches. Variations of removable magnetic disks up to a GB or more in capacity have been developed. (Hot-swappable hard drives?)
- A *magneto-optical* disk uses a magnetic disk covered in a clear plastic coating that protects the surface.
  - The heads sit a considerable distance away from the magnetic surface, and as a result do not have enough magnetic strength to switch bits *at normal room temperature.*
  - For writing, a laser is used to heat up a specific spot on the disk, to a temperature at which the weak magnetic field of the write head is able to flip the bits.
  - For reading, a laser is shined at the disk, and the *Kerr effect* causes the polarization of the light to become rotated either clockwise or counter-clockwise depending on the orientation of the magnetic field.
- *Optical disks* do not use magnetism at all, but instead use special materials that can be altered (by lasers) to have relatively light or dark spots.
  - For example the *phase-change disk* has a material that can be frozen into either a crystalline or an amorphous state, the latter of which is less transparent and reflects less light when a laser is bounced off a reflective surface under the material.
    - Three powers of lasers are used with phase-change disks: (1) a low power laser is used to read the disk, without effecting the materials. (2) A medium power erases the disk, by melting and re-freezing the medium into a crystalline state, and (3) a high power writes to the disk by melting the medium and re-freezing it into the amorphous state.
    - The most common examples of these disks are *re-writable* CD-RWs and DVD-RWs.
- An alternative to the disks described above are *Write-Once Read-Many, WORM* drives.
  - The original version of WORM drives involved a thin layer of aluminum sandwiched between two protective layers of glass or plastic.
    - Holes were burned in the aluminum to write bits.
    - Because the holes could not be filled back in, there was no way to re-write to the disk. (Although data could be erased by burning more holes.)
  - WORM drives have important legal ramifications for data that must be stored for a very long time and must be provable in court as unaltered since it was originally written. (Such as long-term storage of medical records.)
  - Modern CD-R and DVD-R disks are examples of WORM drives that use organic polymer inks instead of an aluminum layer.
- Read-only disks are similar to WORM disks, except the bits are pressed onto the disk at the factory, rather than being burned on one by one. (See http://en.wikipedia.org/wiki/CD_manufacturing#Premastering for more information on CD manufacturing techniques.)

12.9.1.2 Tapes

- Tape drives typically cost more than disk drives, but the cost per MB of the tapes themselves is lower.
- Tapes are typically used today for backups, and for enormous volumes of data stored by certain scientific establishments. (E.g. NASA's archive of space probe and satellite imagery, which is currently being downloaded from numerous sources faster than anyone can actually look at it.)
- Robotic tape changers move tapes from drives to archival tape libraries upon demand.
- (Never underestimate the bandwidth of a station wagon full of tapes rolling down the highway!)

12.9.1.3 Future Technology

- *Solid State Disks, SSDs,* are becoming more and more popular.
- *Holographic storage* uses laser light to store images in a 3-D structure, and the entire data structure can be transferred in a single flash of laser light.
- *Micro-Electronic Mechanical Systems, MEMS,* employs the technology used for computer chip fabrication to create VERY tiny little machines. One example packs 10,000 read-write heads within a square centimeter of space, and as media are passed over it, all 10,000 heads can read data in parallel.

12.9.2 Operating-System Support

- The OS must provide support for tertiary storage as removable media, including the support to transfer data

between different systems.

12.9.2.1 Application Interface

- File systems are typically not stored on tapes. (It might be technically possible, but it is impractical.)
- Tapes are also not low-level formatted, and do not use fixed-length blocks. Rather data is written to tapes in variable length blocks as needed.
- Tapes are normally accessed as raw devices, requiring each application to determine how the data is to be stored and read back. Issues such as header contents and ASCII versus binary encoding (and byte-ordering) are generally application specific.
- Basic operations supported for tapes include locate( ), read( ), write( ), and read_position( ).
- (Because of variable length writes), writing to a tape erases all data that follows that point on the tape.
  - Writing to a tape places the End of Tape (EOT) marker at the end of the data written.
  - It is not possible to locate( ) to any spot past the EOT marker.

12.9.2.2 File Naming

- File naming conventions for removable media are not entirely uniquely specific, nor are they necessarily consistent between different systems. (Two removable disks may contain files with the same name, and there is no clear way for the naming system to distinguish between them.)
- Fortunately music CDs have a common format, readable by all systems. Data CDs and DVDs have only a few format choices, making it easy for a system to support all known formats.

12.9.2.3 Hierarchical Storage Management

- Hierarchical storage involves extending file systems out onto tertiary storage, swapping files from hard drives to tapes in much the same manner as data blocks are swapped from memory to hard drives.
- A placeholder is generally left on the hard drive, storing information about the particular tape (or other removable media) on which the file has been swapped out to.
- A robotic system transfers data to and from tertiary storage as needed, generally automatically upon demand of the file(s) involved.

12.9.3 Performance Issues

12.9.3.1 Speed

- **Sustained Bandwidth** is the rate of data transfer during a large file transfer, once the proper tape is loaded and the file located.
- **Effective Bandwidth** is the effective overall rate of data transfer, including any overhead necessary to load the proper tape and find the file on the tape.
- **Access Latency** is all of the accumulated waiting time before a file can be actually read from tape. This includes the time it takes to find the file on the tape, the time to load the tape from the tape library, and the time spent waiting in the queue for the tape drive to become available.
- Clearly tertiary storage access is much slower than secondary access, although removable disks (e.g. a CD jukebox) have somewhat faster access than a tape library.
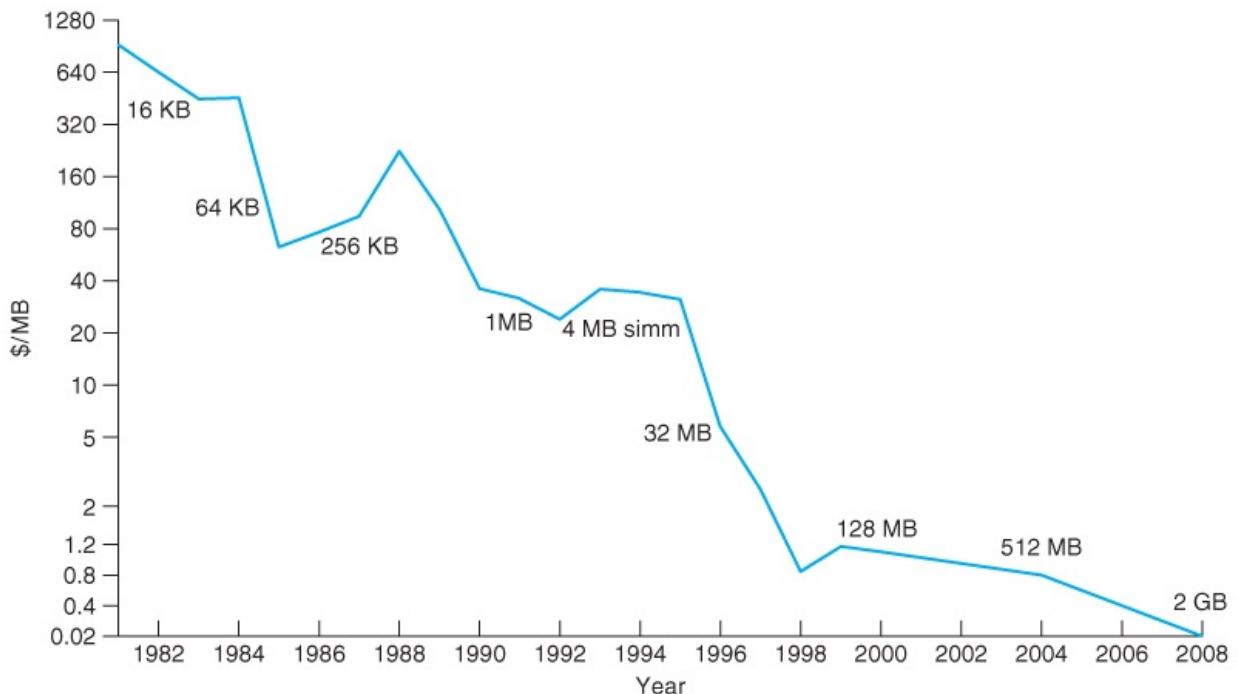
12.9.3.1 Reliability

- Fixed hard drives are generally more reliable than removable drives, because they are less susceptible to the environment.
- Optical disks are generally more reliable than magnetic media.
- A fixed hard drive crash can destroy all data, whereas an optical drive or tape drive failure will often not harm the data media, (and certainly can't damage any media not in the drive at the time of the failure.)
- Tape drives are mechanical devices, and can wear out tapes over time, (as the tape head is generally in much closer physical contact with the tape than disk heads are with platters.)
  - Some drives may only be able to read tapes a few times whereas other drives may be able to re-use the same tapes millions of times.
  - Backup tapes should be read after writing, to verify that the backup tape is readable. (Unfortunately that may have been the LAST time that particular tape was readable, and the only way to be sure is to read it again, . . .)
  - Long-term tape storage can cause degradation, as magnetic fields "drift" from one layer of tape
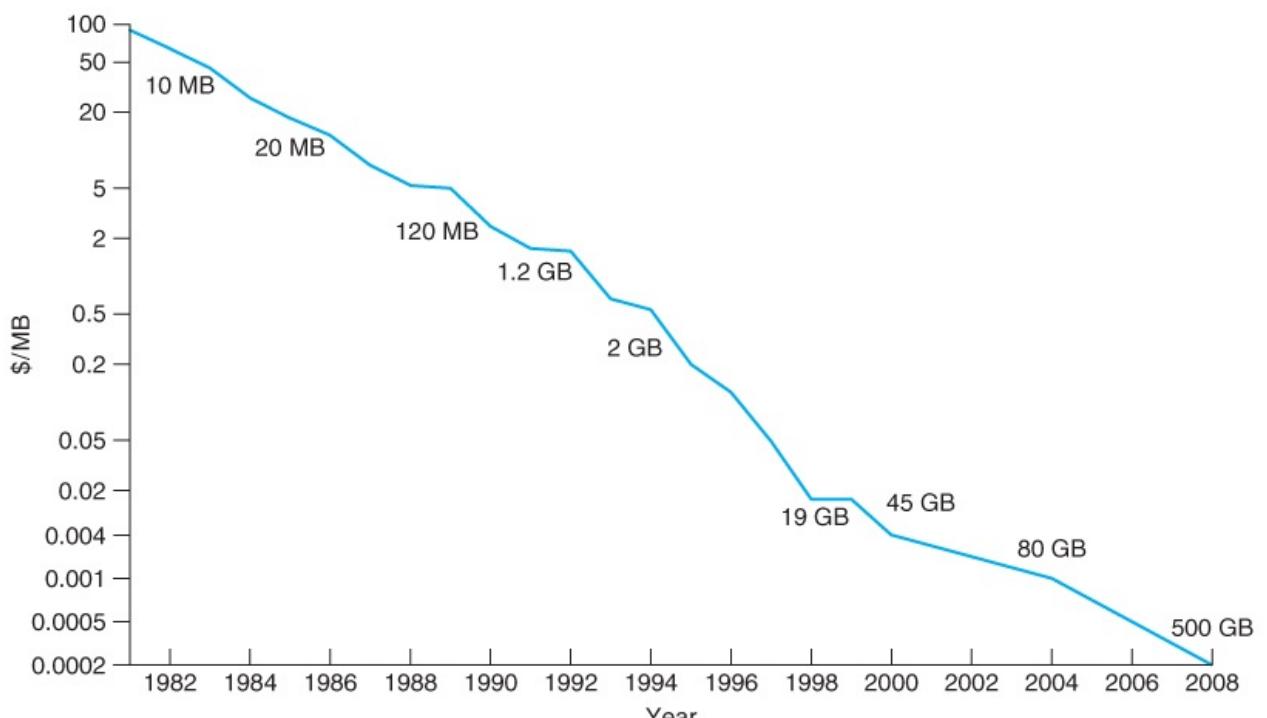
to the adjacent layers. Periodic fast-forwarding and rewinding of tapes can help, by changing which section of tape lays against which other layers.

12.9.3.3 Cost
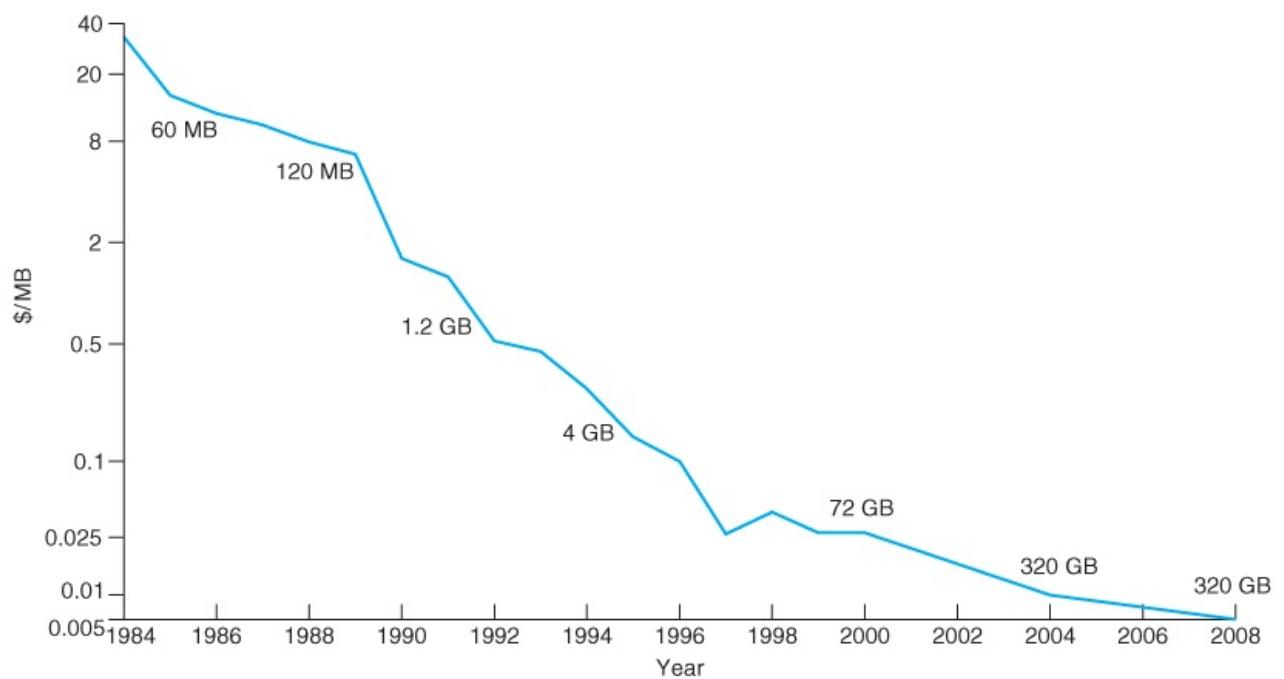
- The cost per megabyte for removable media is its strongest selling feature, particularly as the amount of storage involved (i.e. the number of tapes, CDs, etc) increases.
- However the cost per megabyte for hard drives has dropped more rapidly over the years than the cost of removable media, such that the currently most cost-effective backup solution for many systems is simply an additional (external) hard drive.
- (One good use for old unwanted PCs is to put them on a network as a backup server and/or print server. The downside to this backup solution is that the backups are stored on-site with the original data, and a fire, flood, or burglary could wipe out both the original data and the backups.)

Old Figure 12.15 - Price per megabyte of DRAM, from 1981 to 2008

Old Figure 12.16 - Price per megabyte of magnetic hard disk, from 1981 to 2008.

Old Figure 12.17 - Price per megabyte of a tape drive, from 1984 to 2008.

File-System Interface
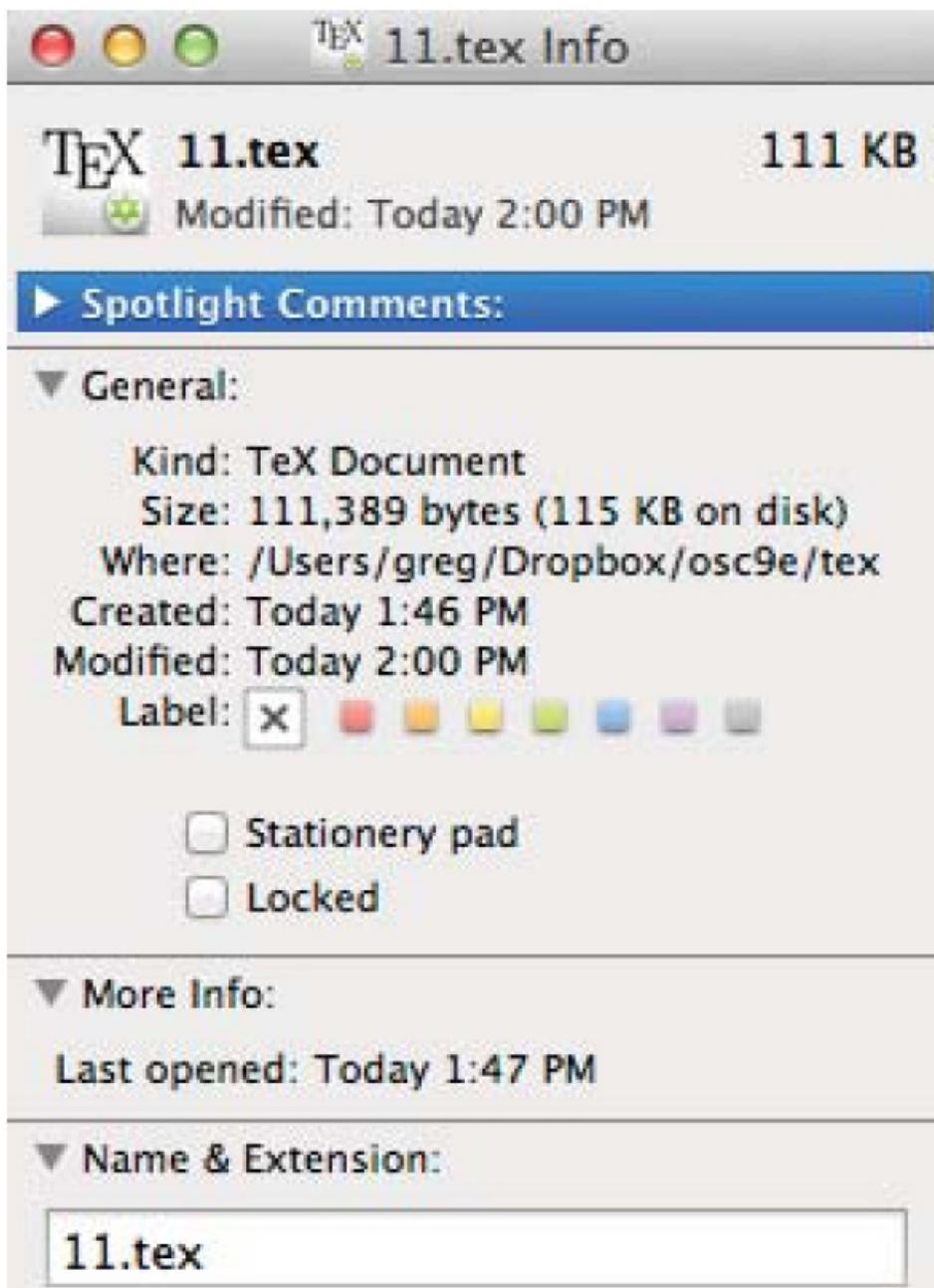
References:
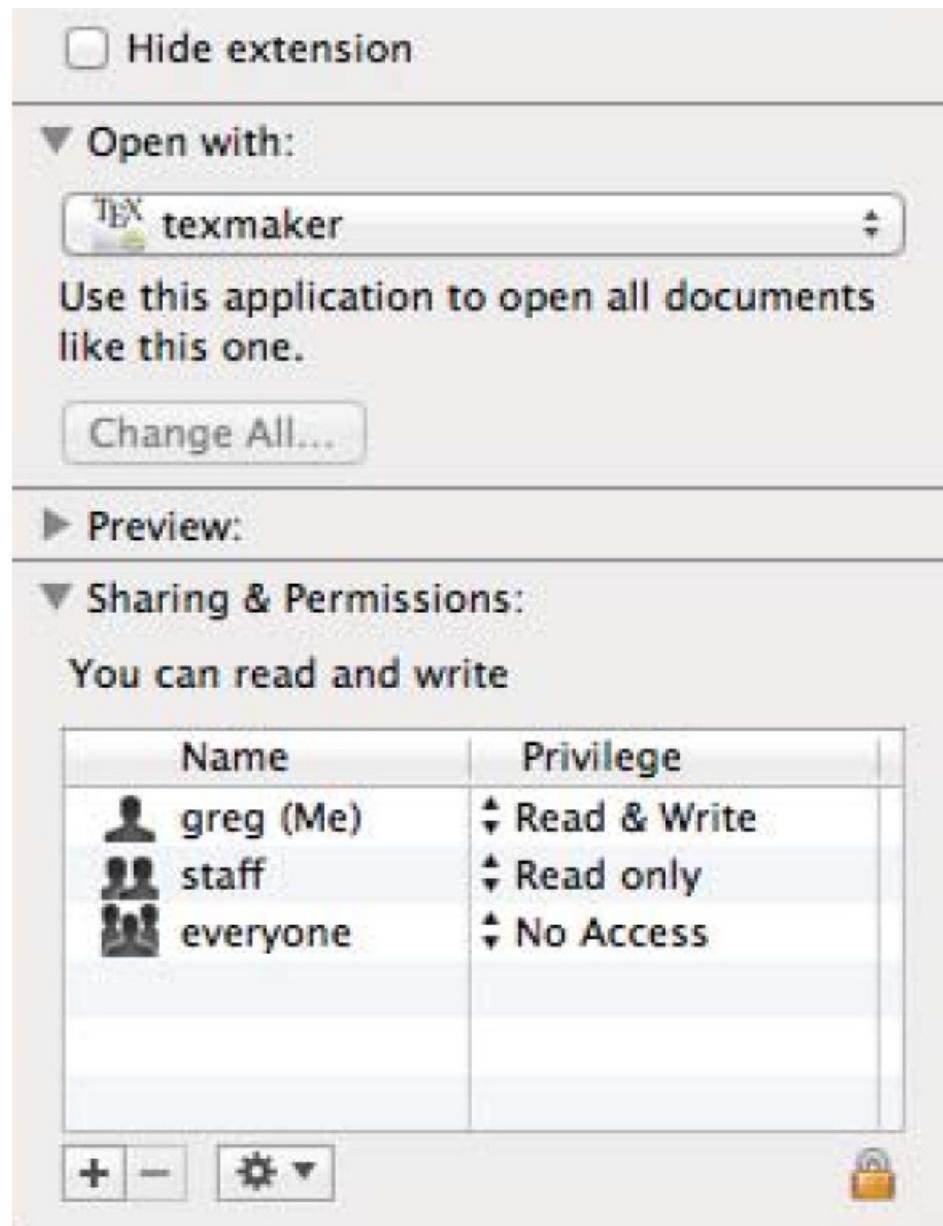
1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 11

11.1 File Concept

11.1.1 File Attributes

- Different OSes keep track of different file attributes, including:
  - **Name** - Some systems give special significance to names, and particularly extensions (.exe, .txt, etc.), and some do not. Some extensions may be of significance to the OS (.exe), and others only to certain applications (.jpg)
  - **Identifier** (e.g. inode number)
  - **Type** - Text, executable, other binary, etc.
  - **Location** - on the hard drive.
  - **Size**
  - **Protection**
  - **Time & Date**
  - **User ID**

11.1.2 File Operations

- The file ADT supports many common operations:
  - Creating a file
  - Writing a file
  - Reading a file
  - Repositioning within a file
  - Deleting a file
  - Truncating a file.
- Most OSes require that files be **opened** before access and **closed** after all access is complete. Normally the programmer must open and close files explicitly, but some rare systems open the file automatically at first access. Information about currently open files is stored in an **open file table**, containing for example:
  - **File pointer** - records the current position in the file, for the next read or write access.
  - **File-open count** - How many times has the current file been opened (simultaneously by different processes) and not yet closed? When this counter reaches zero the file can be removed from the table.
  - **Disk location of the file.**
  - **Access rights**
- Some systems provide support for **file locking.**
  - A **shared lock** is for reading only.
  - A **exclusive lock** is for writing as well as reading.
  - An **advisory lock** is informational only, and not enforced. (A "Keep Out" sign, which may be ignored.)
  - A **mandatory lock** is enforced. (A truly locked door.)

○ UNIX used advisory locks, and Windows uses mandatory locks.

FILE LOCKING IN JAVA (Cont.)

```java
import java.io.*;
import java.nio.channels.*;

public class LockingExample {
  public static final boolean EXCLUSIVE = false;
  public static final boolean SHARED = true;

  public static void main(String arsg[]) throws IOException {
    FileLock sharedLock = null;
    FileLock exclusiveLock = null;

    try {
      RandomAccessFile raf = new RandomAccessFile("file.txt","rw");

      // get the channel for the file
      FileChannel ch = raf.getChannel();

      // this locks the first half of the file - exclusive
      exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);

      /** Now modify the data . . . */

      // release the lock
      exclusiveLock.release();

      // this locks the second half of the file - shared
      sharedLock = ch.lock(raf.length()/2+1,raf.length(),SHARED);

      /** Now read the data . . . */

      // release the lock
      exclusiveLock.release();
    } catch (java.io.IOException ioe) {
      System.err.println(ioe);
    }
    finally {
      if (exclusiveLock != null)
          exclusiveLock.release();
      if (sharedLock != null)
          sharedLock.release();
    }
  }
}
```

Figure 11.2 - File-locking example in Java.

11.1.3 File Types

• Windows (and some other systems) use special file extensions to indicate the type of each file:

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Figure 11.3 - Common file types.

- Macintosh stores a creator attribute for each file, according to the program that first created it with the create( ) system call.
- UNIX stores magic numbers at the beginning of certain files. (Experiment with the "file" command, especially in directories such as /bin and /dev)

11.1.4 File Structure

- Some files contain an internal structure, which may or may not be known to the OS.
- For the OS to support particular file formats increases the size and complexity of the OS.
- UNIX treats all files as sequences of bytes, with no further consideration of the internal structure. (With the exception of executable binary programs, which it must know how to load and find the first executable statement, etc.)
- Macintosh files have two **forks** - a **resource fork**, and a **data fork**. The resource fork contains information relating to the UI, such as icons and button images, and can be modified independently of the data fork, which contains the code or data as appropriate.
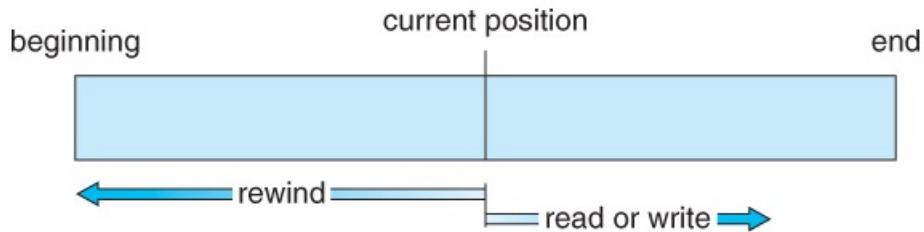
11.1.5 Internal File Structure

- Disk files are accessed in units of physical blocks, typically 512 bytes or some power-of-two multiple thereof. (Larger physical disks use larger block sizes, to keep the range of block numbers within the range of a 32-bit integer.)
- Internally files are organized in units of logical units, which may be as small as a single byte, or may be a larger size corresponding to some data record or structure size.
- The number of logical units which fit into one physical block determines its **packing**, and has an impact on the amount of internal fragmentation (wasted space) that occurs.
- As a general rule, half a physical block is wasted for each file, and the larger the block sizes the more space is lost to internal fragmentation.

11.2 Access Methods

11.2.1 Sequential Access

- A sequential access file emulates magnetic tape operation, and generally supports a few operations:
    - read next - read a record and advance the tape to the next position.
    - write next - write a record and advance the tape to the next position.
    - rewind
    - skip n records - May or may not be supported. N may be limited to positive numbers, or may be limited to +/- 1.

**Figure 11.4 - Sequential-access file.**

11.2.2 Direct Access
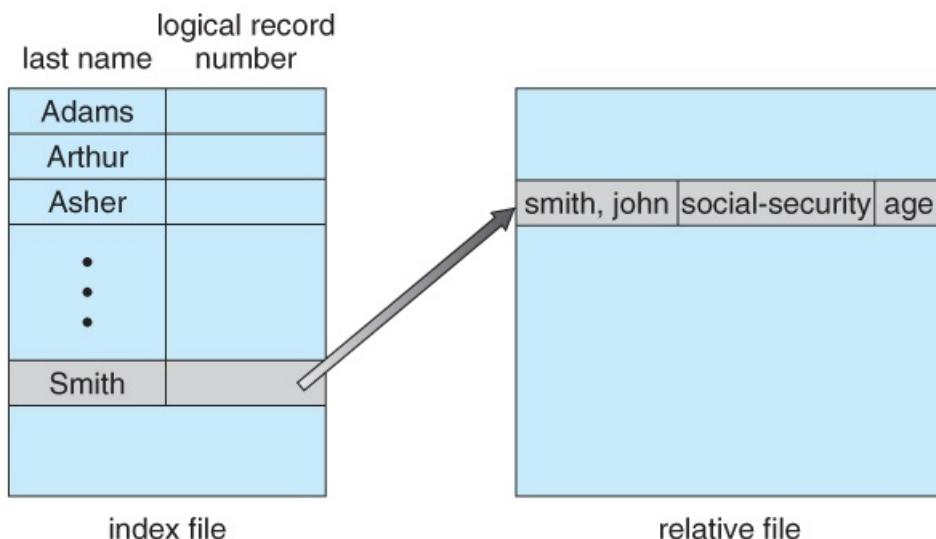
- Jump to any record and read that record. Operations supported include:
    - read n - read record number n. (Note an argument is now required.)
    - write n - write record number n. (Note an argument is now required.)
    - jump to record n - could be 0 or the end of file.
    - Query current record - used to return back to this record later.
    - Sequential access can be easily emulated using direct access. The inverse is complicated and inefficient.

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;
write_next	write cp; cp = cp + 1;

Figure 11.5 - Simulation of sequential access on a direct-access file.

11.2.3 Other Access Methods

- An indexed access scheme can be easily built on top of a direct access system. Very large files may require a multi-tiered indexing scheme, i.e. indexes of indexes.

Figure 11.6 - Example of index and relative files.

11.3 Directory Structure

11.3.1 Storage Structure

- A disk can be used in its entirety for a file system.
- Alternatively a physical disk can be broken up into multiple **partitions, slices, or mini-disks**, each of which becomes a virtual disk and can have its own filesystem. (or be used for raw storage, swap space, etc.)
- Or, multiple physical disks can be combined into one **volume**, i.e. a larger virtual disk, with its own filesystem spanning the physical disks.
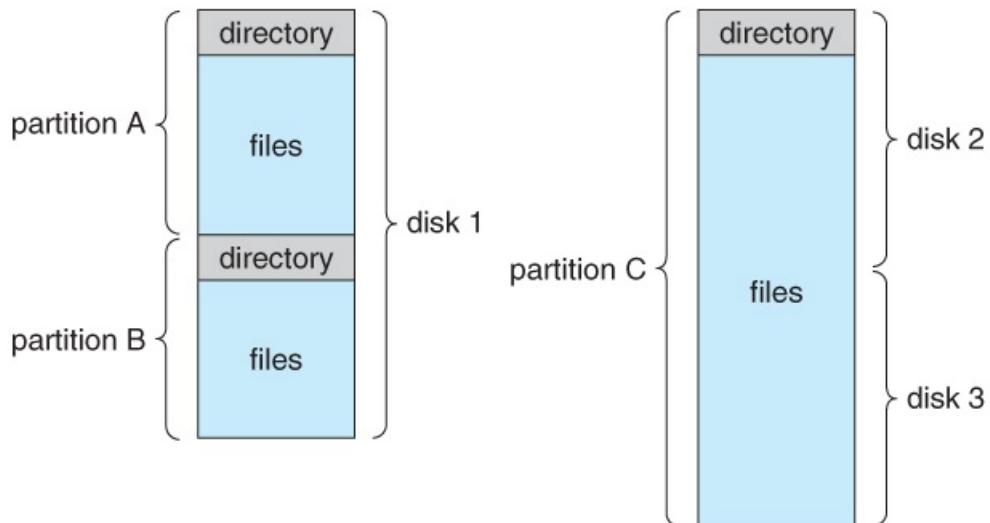

Figure 11.7 - A typical file-system organization.

/	ufs
/devices	devfs
/dev	dev
/system/contract	ctfs
/proc	proc
/etc/mnttab	mntfs
/etc/svc/volatile	tmpfs
/system/object	objfs
/lib/libc.so.1	lofs
/dev/fd	fd
/var	ufs
/tmp	tmpfs
/var/run	tmpfs
/opt	ufs
/zpbge	zfs
/zpbge/backup	zfs
/export/home	zfs
/var/mail	zfs
/var/spool/mqueue	zfs
/zpbg	zfs
/zpbg/zones	zfs

Figure 11.8 Solaris file systems.

11.3.2 Directory Overview

- Directory operations to be supported include:
    - Search for a file
    - Create a file - add to the directory
    - Delete a file - erase from the directory

- List a directory - possibly ordered in different ways.
- Rename a file - may change sorting order
- Traverse the file system.

11.3.3. Single-Level Directory

- Simple to implement, but each file must have a unique name.
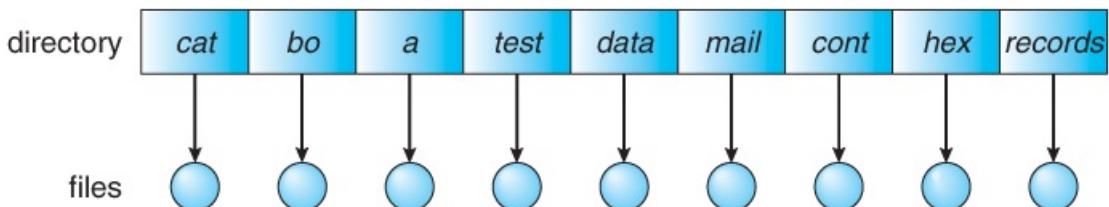
Figure 11.9 - Single-level directory.

11.3.4 Two-Level Directory

- Each user gets their own directory space.
- File names only need to be unique within a given user's directory.
- A master file directory is used to keep track of each users directory, and must be maintained when users are added to or removed from the system.
- A separate directory is generally needed for system (executable) files.
- Systems may or may not allow users to access other directories besides their own
  - If access to other directories is allowed, then provision must be made to specify the directory being accessed.
  - If access is denied, then special consideration must be made for users to run programs located in system directories. A **search path** is the list of directories in which to search for executable programs, and can be set uniquely for each user.
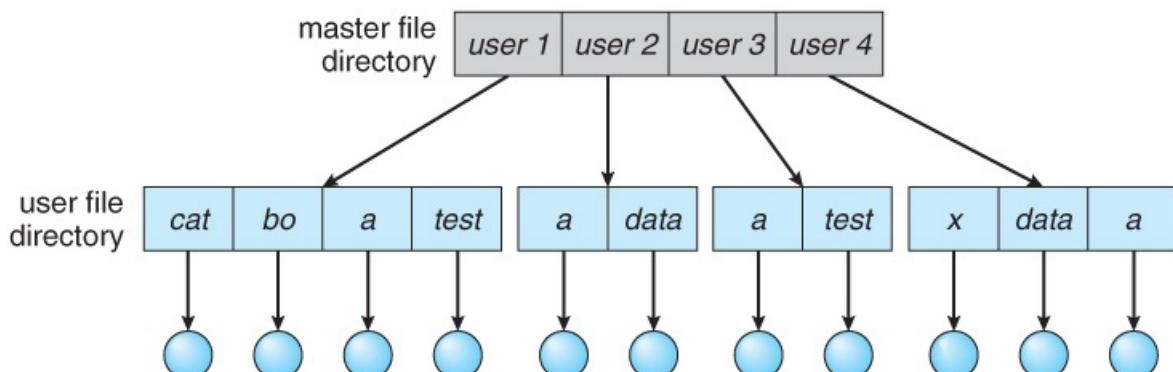
Figure 11.10 - Two-level directory structure.

11.3.5 Tree-Structured Directories

- An obvious extension to the two-tiered directory structure, and the one with which we are all most familiar.
- Each user / process has the concept of a **current directory** from which all (relative) searches take place.
- Files may be accessed using either absolute pathnames (relative to the root of the tree) or relative pathnames (relative to the current directory.)
- Directories are stored the same as any other file in the system, except there is a bit that identifies them as directories, and they have some special structure that the OS understands.
- One question for consideration is whether or not to allow the removal of directories that are not empty - Windows requires that directories be emptied first, and UNIX provides an option for deleting entire sub-trees.
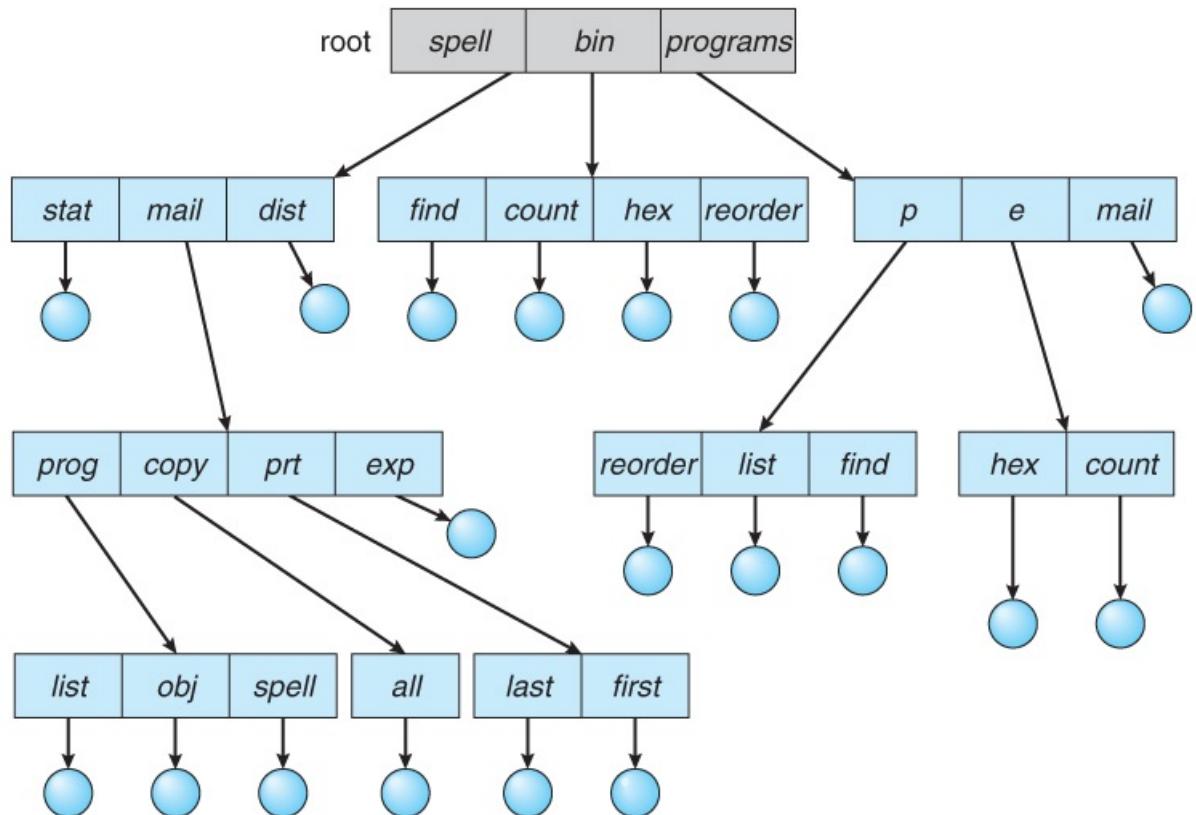
Figure 11.11 - Tree-structured directory structure.

### 11.3.6 Acyclic-Graph Directories

- When the same files need to be accessed in more than one place in the directory structure (e.g. because they are being shared by more than one user / process), it can be useful to provide an acyclic-graph structure. (Note the **directed** arcs from parent to child.)
- UNIX provides two types of **links** for implementing the acyclic-graph structure. (See "man ln" for more details.)
  - A **hard link** (usually just called a link) involves multiple directory entries that both refer to the same file. Hard links are only valid for ordinary files in the same filesystem.
  - A **symbolic link**, that involves a special file, containing information about where to find the linked file. Symbolic links may be used to link directories and/or files in other filesystems, as well as ordinary files in the current filesystem.
- Windows only supports symbolic links, termed **shortcuts.**
- Hard links require a **reference count**, or **link count** for each file, keeping track of how many directory entries are currently referring to this file. Whenever one of the references is removed the link count is reduced, and when it reaches zero, the disk space can be reclaimed.
- For symbolic links there is some question as to what to do with the symbolic links when the original file is moved or deleted:
  - One option is to find all the symbolic links and adjust them also.
  - Another is to leave the symbolic links dangling, and discover that they are no longer valid the next time they are used.
  - What if the original file is removed, and replaced with another file having the same name before the symbolic link is next used?
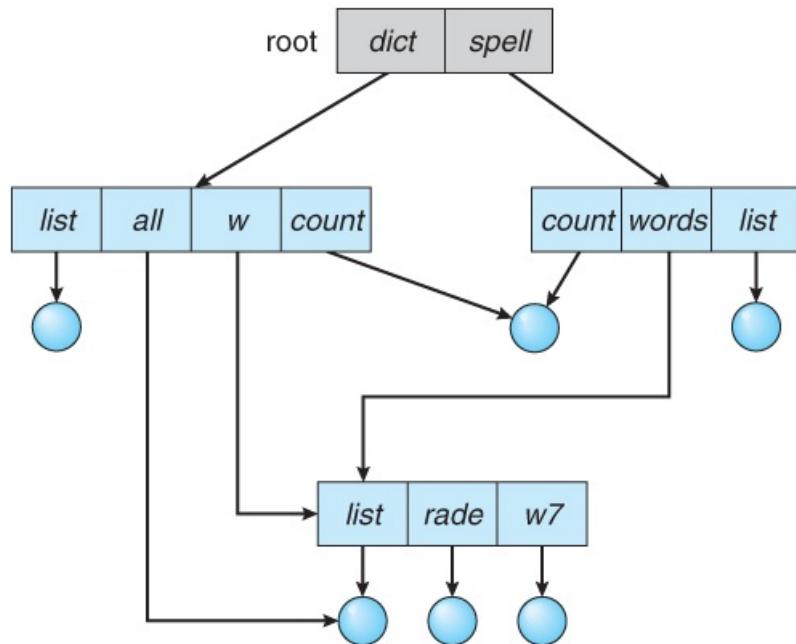
Figure 11.12 - Acyclic-graph directory structure.

11.3.7 General Graph Directory

- If cycles are allowed in the graphs, then several problems can arise:
  - Search algorithms can go into infinite loops. One solution is to not follow links in search algorithms. (Or not to follow symbolic links, and to only allow symbolic links to refer to directories.)
  - Sub-trees can become disconnected from the rest of the tree and still not have their reference counts reduced to zero. Periodic garbage collection is required to detect and resolve this problem. (chkdsk in DOS and fsck in UNIX search for these problems, among others, even though cycles are not supposed to be allowed in either system. Disconnected disk blocks that are not marked as free are added back to the file systems with made-up file names, and can usually be safely deleted.)
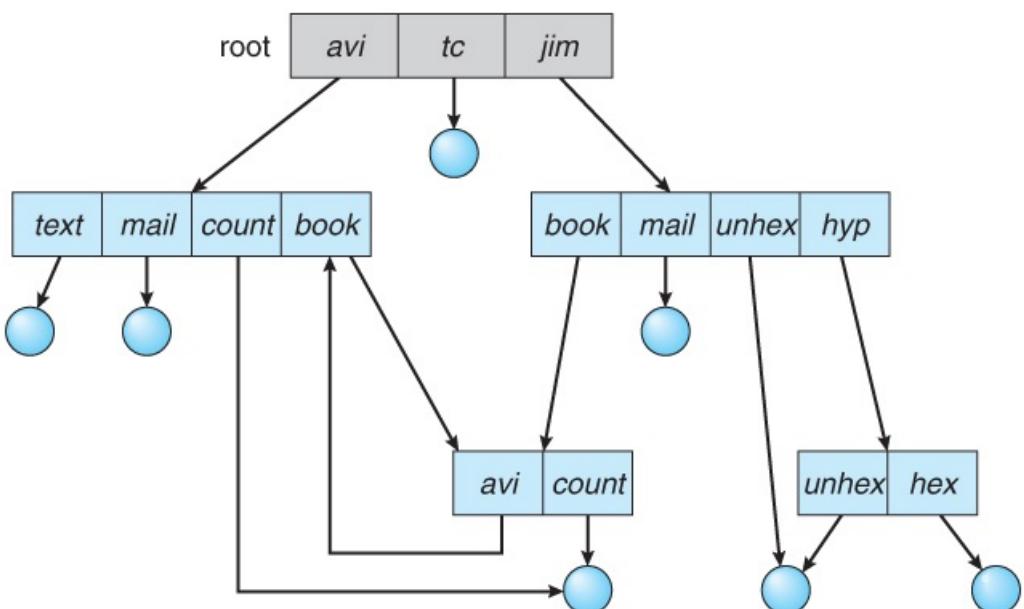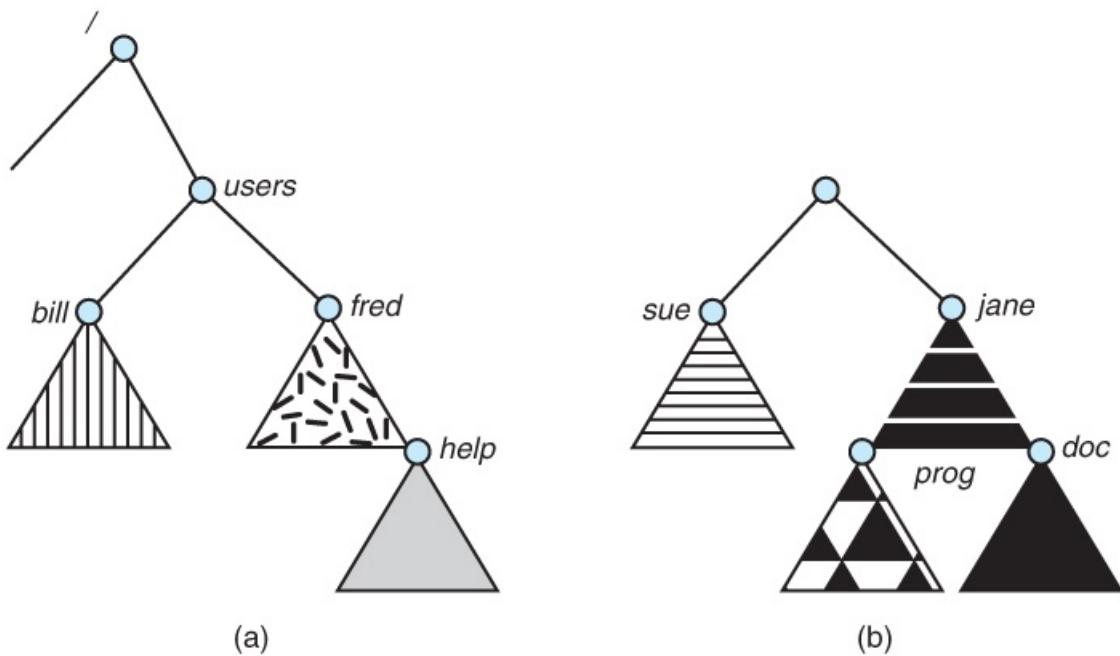
Figure 11.13 - General graph directory.

11.4 File-System Mounting
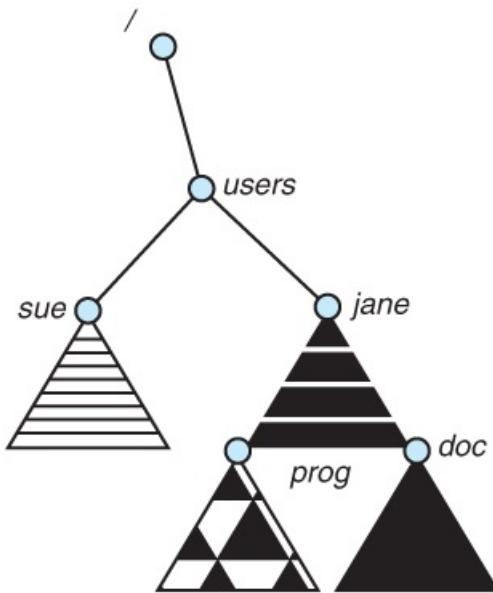
- The basic idea behind mounting file systems is to combine multiple file systems into one large tree structure.
- The mount command is given a filesystem to mount and a **mount point** (directory) on which to attach it.
- Once a file system is mounted onto a mount point, any further references to that directory actually refer to the root of the mounted file system.
- Any files (or sub-directories) that had been stored in the mount point directory prior to mounting the new filesystem are now hidden by the mounted filesystem, and are no longer available. For this reason some systems only allow mounting

onto empty directories.

- Filesystems can only be mounted by root, unless root has previously configured certain filesystems to be mountable onto certain pre-determined mount points. (E.g. root may allow users to mount floppy filesystems to /mnt or something like it.) Anyone can run the mount command to see what filesystems are currently mounted.
- Filesystems may be mounted read-only, or have other restrictions imposed.


Figure 11.14 - File system. (a) Existing system. (b) Unmounted volume.

Figure 11.15 - Mount point.

- The traditional Windows OS runs an extended two-tier directory structure, where the first tier of the structure separates volumes by drive letters, and a tree structure is implemented below that level.
- Macintosh runs a similar system, where each new volume that is found is automatically mounted and added to the desktop when it is found.
- More recent Windows systems allow filesystems to be mounted to any directory in the filesystem, much like UNIX.

11.5 File Sharing

11.5.1 Multiple Users

- On a multi-user system, more information needs to be stored for each file:
  - The owner (user) who owns the file, and who can control its access.
  - The group of other user IDs that may have some special access to the file.
  - What access rights are afforded to the owner (**U**ser), the **G**roup, and to the rest of the world (the

universe, a.k.a. **O**thers.)
- Some systems have more complicated access control, allowing or denying specific accesses to specifically named users or groups.

11.5.2 Remote File Systems

- The advent of the Internet introduces issues for accessing files stored on remote computers
  - The original method was ftp, allowing individual files to be transported across systems as needed. Ftp can be either account and password controlled, or **anonymous**, not requiring any user name or password.
  - Various forms of **distributed file systems** allow remote file systems to be mounted onto a local directory structure, and accessed using normal file access commands. (The actual files are still transported across the network as needed, possibly using ftp as the underlying transport mechanism.)
  - The WWW has made it easy once again to access files on remote systems without mounting their filesystems, generally using (anonymous) ftp as the underlying file transport mechanism.

11.5.2.1 The Client-Server Model

- When one computer system remotely mounts a filesystem that is physically located on another system, the system which physically owns the files acts as a **server**, and the system which mounts them is the **client.**
- User IDs and group IDs must be consistent across both systems for the system to work properly. (I.e. this is most applicable across multiple computers managed by the same organization, shared by a common group of users.)
- The same computer can be both a client and a server. (E.g. cross-linked file systems.)
- There are a number of security concerns involved in this model:
  - Servers commonly restrict mount permission to certain trusted systems only. Spoofing (a computer pretending to be a different computer) is a potential security risk.
  - Servers may restrict remote access to read-only.
  - Servers restrict which filesystems may be remotely mounted. Generally the information within those subsystems is limited, relatively public, and protected by frequent backups.
- The NFS (Network File System) is a classic example of such a system.

11.5.2.2 Distributed Information Systems

- The **Domain Name System, DNS,** provides for a unique naming system across all of the Internet.
- Domain names are maintained by the **Network Information System, NIS**, which unfortunately has several security issues. NIS+ is a more secure version, but has not yet gained the same widespread acceptance as NIS.
- Microsoft's **Common Internet File System, CIFS**, establishes a **network login** for each user on a networked system with shared file access. Older Windows systems used **domains**, and newer systems (XP, 2000), use **active directories.** User names must match across the network for this system to be valid.
- A newer approach is the **Lightweight Directory-Access Protocol, LDAP,** which provides a **secure single sign-on** for all users to access all resources on a network. This is a secure system which is gaining in popularity, and which has the maintenance advantage of combining authorization information in one central location.

11.5.2.3 Failure Modes

- When a local disk file is unavailable, the result is generally known immediately, and is generally non-recoverable. The only reasonable response is for the response to fail.
- However when a remote file is unavailable, there are many possible reasons, and whether or not it is unrecoverable is not readily apparent. Hence most remote access systems allow for blocking or delayed response, in the hopes that the remote system (or the network) will come back up eventually.

11.5.3 Consistency Semantics

- **Consistency Semantics** deals with the consistency between the views of shared files on a networked system. When one user changes the file, when do other users see the changes?

- At first glance this appears to have all of the synchronization issues discussed in Chapter 6. Unfortunately the long delays involved in network operations prohibit the use of atomic operations as discussed in that chapter.

11.5.3.1 UNIX Semantics

- The UNIX file system uses the following semantics:
  - Writes to an open file are immediately visible to any other user who has the file open.
  - One implementation uses a shared location pointer, which is adjusted for all sharing users.
- The file is associated with a single exclusive physical resource, which may delay some accesses.

11.5.3.2 Session Semantics

- The Andrew File System, AFS uses the following semantics:
  - Writes to an open file are not immediately visible to other users.
  - When a file is closed, any changes made become available only to users who open the file at a later time.
- According to these semantics, a file can be associated with multiple (possibly different) views. Almost no constraints are imposed on scheduling accesses. No user is delayed in reading or writing their personal copy of the file.
- AFS file systems may be accessible by systems around the world. Access control is maintained through (somewhat) complicated access control lists, which may grant access to the entire world (literally) or to specifically named users accessing the files from specifically named remote environments.

11.5.3.3 Immutable-Shared-Files Semantics

- Under this system, when a file is declared as *shared* by its creator, it becomes immutable and the name cannot be re-used for any other resource. Hence it becomes read-only, and shared access is simple.

11.6 Protection

- Files must be kept safe for reliability (against accidental damage), and protection (against deliberate malicious access.) The former is usually managed with backup copies. This section discusses the latter.
- One simple protection scheme is to remove all access to a file. However this makes the file unusable, so some sort of controlled access must be arranged.

11.6.1 Types of Access

- The following low-level operations are often controlled:
  - Read - View the contents of the file
  - Write - Change the contents of the file.
  - Execute - Load the file onto the CPU and follow the instructions contained therein.
  - Append - Add to the end of an existing file.
  - Delete - Remove a file from the system.
  - List -View the name and other attributes of files on the system.
- Higher-level operations, such as copy, can generally be performed through combinations of the above.

11.6.2 Access Control

- One approach is to have complicated *Access Control Lists, ACL,* which specify exactly what access is allowed or denied for specific users or groups.
  - The AFS uses this system for distributed access.
  - Control is very finely adjustable, but may be complicated, particularly when the specific users involved are unknown. (AFS allows some wild cards, so for example all users on a certain remote system may be trusted, or a given username may be trusted when accessing from any remote system.)
- UNIX uses a set of 9 access control bits, in three groups of three. These correspond to R, W, and X permissions for each of the Owner, Group, and Others. (See "man chmod" for full details.) The RWX bits control the following privileges for ordinary files and directories:

bit	Files	Directories
R	Read (view) file contents.	Read directory contents. Required to get a listing of the directory.
W	Write (change) file contents.	Change directory contents. Required to create or delete files.
X	Execute file contents as a program.	Access detailed directory information. Required to get a long listing, or to access any specific file in the directory. Note that if a user has X but not R permissions on a directory, they can still access specific files, but only if they already know the name of the file they are trying to access.

- In addition there are some special bits that can also be applied:
  - The set user ID (SUID) bit and/or the set group ID (SGID) bits applied to executable files temporarily change the identity of whoever runs the program to match that of the owner / group of the executable program. This allows users running specific programs to have access to files (**while running that program**) to which they would normally be unable to access. Setting of these two bits is usually restricted to root, and must be done with caution, as it introduces a potential security leak.
  - The sticky bit on a directory modifies write permission, allowing users to only delete files for which they are the owner. This allows everyone to create files in /tmp, for example, but to only delete files which they have created, and not anyone else's.
  - The SUID, SGID, and sticky bits are indicated with an S, S, and T in the positions for execute permission for the user, group, and others, respectively. If the letter is lower case, (s, s, t), then the corresponding execute permission is not also given. If it is upper case, (S, S, T), then the corresponding execute permission IS given.
  - The numeric form of chmod is needed to set these advanced bits.

```
-rw-rw-r--    1 pbg   staff     31200  Sep 3 08:30   intro.ps
drwx------    5 pbg   staff       512  Jul 8 09.33   private/
drwxrwxr-x    2 pbg   staff       512  Jul 8 09:35   doc/
drwxrwx---    2 jwg   student     512  Aug 3 14:13   student-proj/
-rw-r--r--    1 pbg   staff      9423  Feb 24 2012   program.c
-rwxr-xr-x    1 pbg   staff     20471  Feb 24 2012   program
drwx--x--x    4 tag   faculty     512  Jul 31 10:31  lib/
drwx------    3 pbg   staff      1024  Aug 29 06:52  mail/
drwxrwxrwx    3 pbg   staff       512  Jul 8 09:35   test/
```

Sample permissions in a UNIX system.

- Windows adjusts files access through a simple GUI:
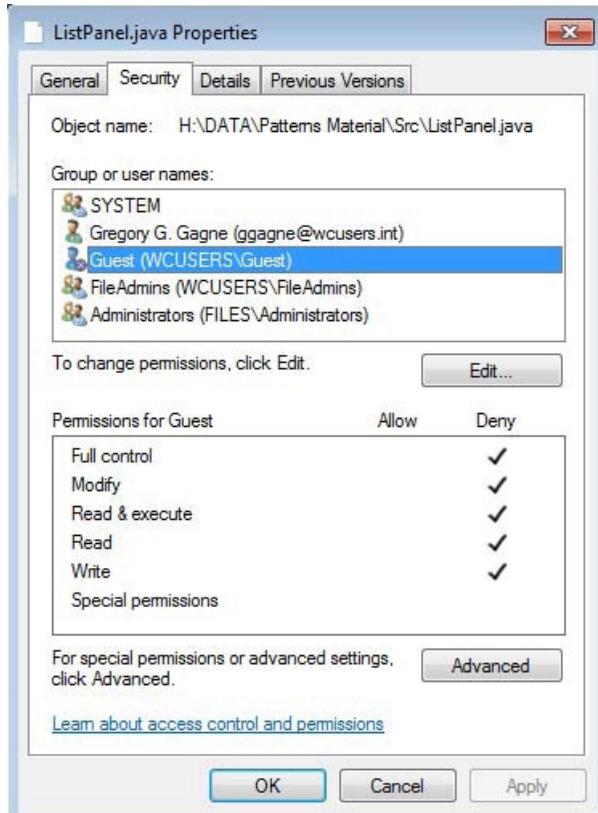
Figure 11.16 - Windows 7 access-control list management.

11.6.3 Other Protection Approaches and Issues

- Some systems can apply passwords, either to individual files, or to specific sub-directories, or to the entire system. There is a trade-off between the number of passwords that must be maintained (and remembered by the users) and the amount of information that is vulnerable to a lost or forgotten password.
- Older systems which did not originally have multi-user file access permissions (DOS and older versions of Mac) must now be *retrofitted* if they are to share files on a network.
- Access to a file requires access to all the files along its path as well. In a cyclic directory structure, users may have different access to the same file accessed through different paths.
- Sometimes just the knowledge of the existence of a file of a certain name is a security (or privacy) concern. Hence the distinction between the R and X bits on UNIX directories.

11.7 Summary

File-System Implementation

References:

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 12

12.1 File-System Structure

- Hard disks have two important properties that make them suitable for secondary storage of files in file systems: (1) Blocks of data can be rewritten in place, and (2) they are direct access, allowing any block of data to be accessed with only (relatively) minor movements of the disk heads and rotational latency. (See Chapter 12)
- Disks are usually accessed in physical blocks, rather than a byte at a time. Block sizes may range from 512 bytes to 4K or larger.
- File systems organize storage on disk drives, and can be viewed as a layered design:
  - At the lowest layer are the physical devices, consisting of the magnetic media, motors & controls, and the electronics connected to them and controlling them. Modern disk put more and more of the electronic controls directly on the disk drive itself, leaving relatively little work for the disk controller card to perform.
  - *I/O Control* consists of *device drivers*, special software programs (often written in assembly) which communicate with the devices by reading and writing special codes directly to and from memory addresses corresponding to the controller card's registers. Each controller card (device) on a system has a different set of addresses (registers, a.k.a. *ports*) that it listens to, and a unique set of command codes and results codes that it understands.
  - The *basic file system* level works directly with the device drivers in terms of retrieving and storing raw blocks of data, without any consideration for what is in each block. Depending on the system, blocks may be referred to with a single block number, (e.g. block # 234234), or with head-sector-cylinder combinations.
  - The *file organization module* knows about files and their logical blocks, and how they map to physical blocks on the disk. In addition to translating from logical to physical blocks, the file organization module also maintains the list of free blocks, and allocates free blocks to files as needed.
  - The *logical file system* deals with all of the meta data associated with a file (UID, GID, mode, dates, etc), i.e. everything about the file except the data itself. This level manages the directory structure and the mapping of file names to *file control blocks, FCBs*, which contain all of the meta data as well as block number information for finding the data on the disk.
- The layered approach to file systems means that much of the code can be used uniformly for a wide variety of different file systems, and only certain layers need to be filesystem specific. Common file systems in use include the UNIX file system, UFS, the Berkeley Fast File System, FFS, Windows systems FAT, FAT32, NTFS, CD-ROM systems ISO 9660, and for Linux the extended file systems ext2 and ext3 (among 40 others supported.)
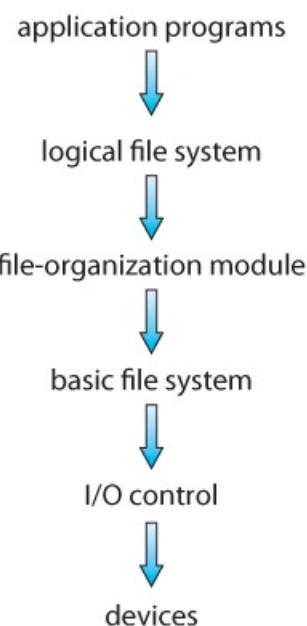
Figure 12.1 - Layered file system.

12.2 File-System Implementation

12.2.1 Overview

- File systems store several important data structures on the disk:
  - A **boot-control block**, (per volume) a.k.a. the **boot block** in UNIX or the **partition boot sector** in Windows contains information about how to boot the system off of this disk. This will generally be the first sector of the volume if there is a bootable system loaded on that volume, or the block will be left vacant otherwise.
  - A **volume control block,** (per volume) a.k.a. the **master file table** in UNIX or the **superblock** in Windows, which contains information such as the partition table, number of blocks on each filesystem, and pointers to free blocks and free FCB blocks.
  - A directory structure (per file system), containing file names and pointers to corresponding FCBs. UNIX uses inode numbers, and NTFS uses a **master file table.**
  - The **File Control Block, FCB,** (per file) containing details about ownership, size, permissions, dates, etc. UNIX stores this information in inodes, and NTFS in the master file table as a relational database structure.

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks
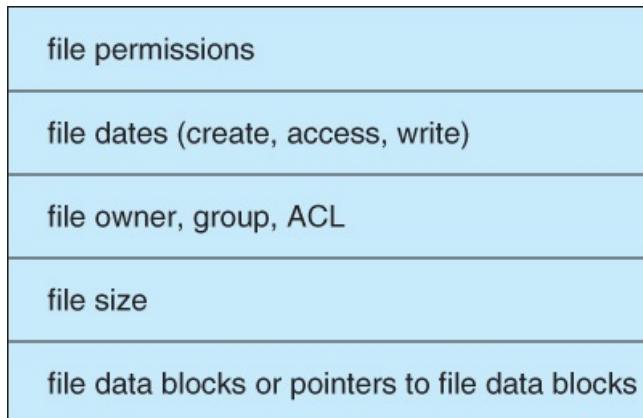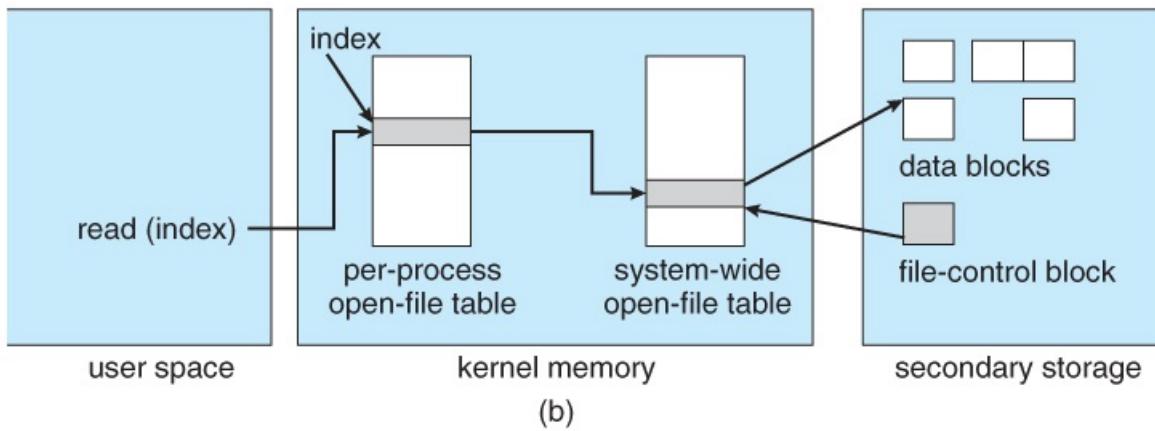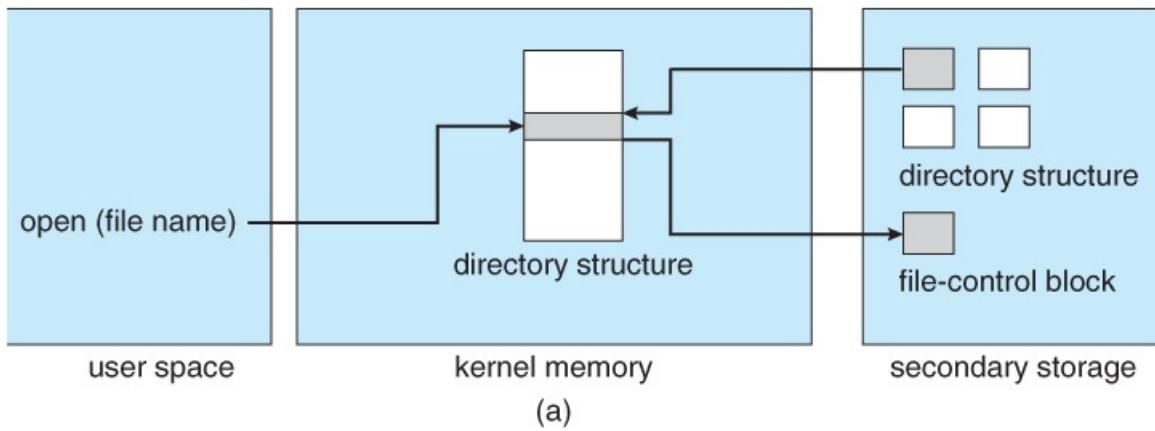
Figure 12.2 - A typical file-control block.

- There are also several key data structures stored in memory:
  - An in-memory mount table.
  - An in-memory directory cache of recently accessed directory information.
  - **A system-wide open file table**, containing a copy of the FCB for every currently open file in the system, as well as some other related information.
  - **A per-process open file table,** containing a pointer to the system open file table as well as some other information. (For example the current file position pointer may be either here or in the system file table, depending on the implementation and whether the file is being shared or not.)
- Figure 12.3 illustrates some of the interactions of file system components when files are created and/or used:
  - When a new file is created, a new FCB is allocated and filled out with important information regarding the new file. The appropriate directory is modified with the new file name and FCB information.
  - When a file is accessed during a program, the open( ) system call reads in the FCB information from disk, and stores it in the system-wide open file table. An entry is added to the per-process open file table referencing the system-wide table, and an index into the per-process table is returned by the open( ) system call. UNIX refers to this index as a **file descriptor**, and Windows refers to it as a **file handle**.
  - If another process already has a file open when a new request comes in for the same file, and it is sharable, then a counter in the system-wide table is incremented and the per-process table is adjusted to point to the existing entry in the system-wide table.
  - When a file is closed, the per-process table entry is freed, and the counter in the system-wide table is decremented. If that counter reaches zero, then the system wide table is also freed. Any data currently stored in memory cache for this file is written out to disk if necessary.

Figure 12.3 - In-memory file-system structures. (a) File open. (b) File read.
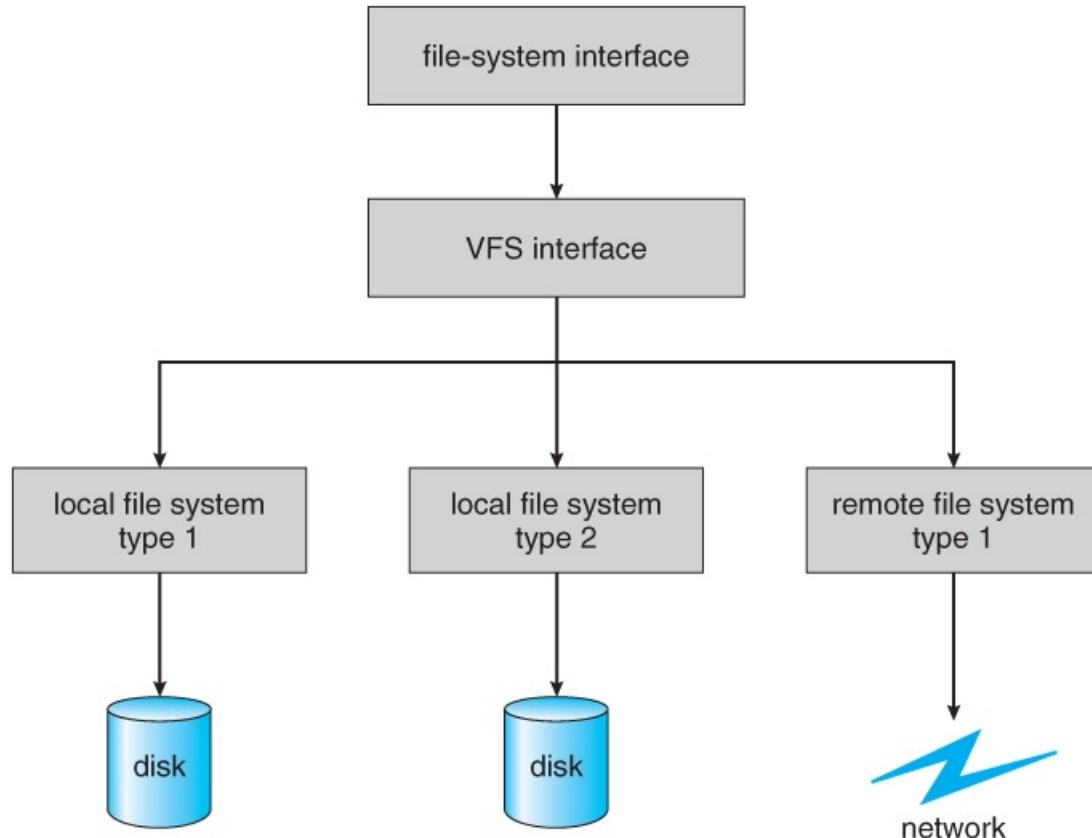
12.2.2 Partitions and Mounting

- Physical disks are commonly divided into smaller units called partitions. They can also be combined into larger units, but that is most commonly done for RAID installations and is left for later chapters.
- Partitions can either be used as raw devices (with no structure imposed upon them), or they can be formatted to hold a filesystem (i.e. populated with FCBs and initial directory structures as appropriate.) Raw partitions are generally used for swap space, and may also be used for certain programs such as databases that choose to manage their own disk storage system. Partitions containing filesystems can generally only be accessed using the file system structure by ordinary users, but can often be accessed as a raw device also by root.
- The boot block is accessed as part of a raw partition, by the boot program prior to any operating system being loaded. Modern boot programs understand multiple OSes and filesystem formats, and can give the user a choice of which of several available systems to boot.
- The *root partition* contains the OS kernel and at least the key portions of the OS needed to complete the boot process. At boot time the root partition is mounted, and control is transferred from the boot program to the kernel found there. (Older systems required that the root partition lie completely within the first 1024 cylinders of the disk, because that was as far as the boot program could reach. Once the kernel had control, then it could access partitions beyond the 1024 cylinder boundary.)
- Continuing with the boot process, additional filesystems get mounted, adding their information into the appropriate mount table structure. As a part of the mounting process the file systems may be checked for errors or inconsistencies, either because they are flagged as not having been closed properly the last time they were used, or just for general principals. Filesystems may be mounted either automatically or manually. In UNIX a mount point is indicated by setting a flag in the in-memory copy of the inode, so all future references to that inode get re-directed to the root directory of the mounted filesystem.

12.2.3 Virtual File Systems

- *Virtual File Systems, VFS*, provide a common interface to multiple different filesystem types. In addition, it provides for a unique identifier (vnode) for files across the entire space, including across all filesystems of different types. (UNIX inodes are unique only across a single filesystem, and certainly do not carry across networked file systems.)

- The VFS in Linux is based upon four key object types:
  - The ***inode*** object, representing an individual file
  - The ***file*** object, representing an open file.
  - The ***superblock*** object, representing a filesystem.
  - The ***dentry*** object, representing a directory entry.
- Linux VFS provides a set of common functionalities for each filesystem, using function pointers accessed through a table. The same functionality is accessed through the same table position for all filesystem types, though the actual functions pointed to by the pointers may be filesystem-specific. See /usr/include/linux/fs.h for full details. Common operations provided include open( ), read( ), write( ), and mmap( ).


Figure 12.4 - Schematic view of a virtual file system.

12.3 Directory Implementation

- Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.

12.3.1 Linear List

- A linear list is the simplest and easiest directory structure to set up, but it does have some drawbacks.
- Finding a file (or verifying one does not already exist upon creation) requires a linear search.
- Deletions can be done by moving all entries, flagging an entry as deleted, or by moving the last entry into the newly vacant position.
- Sorting the list makes searches faster, at the expense of more complex insertions and deletions.
- A linked list makes insertions and deletions into a sorted list easier, with overhead for the links.
- More complex data structures, such as B-trees, could also be considered.

12.3.2 Hash Table

- A hash table can also be used to speed up searches.
- Hash tables are generally implemented ***in addition to*** a linear or other structure

12.4 Allocation Methods

- There are three major methods of storing files on disks: contiguous, linked, and indexed.

12.4.1 Contiguous Allocation

- *Contiguous Allocation* requires that all blocks of a file be kept together contiguously.
- Performance is very fast, because reading successive blocks of the same file generally requires no movement of the disk heads, or at most one small step to the next adjacent cylinder.
- Storage allocation involves the same issues discussed earlier for the allocation of contiguous blocks of memory (first fit, best fit, fragmentation problems, etc.) The distinction is that the high time penalty required for moving the disk heads from spot to spot may now justify the benefits of keeping files contiguously when possible.
- (Even file systems that do not by default store files contiguously can benefit from certain utilities that compact the disk and make all files contiguous in the process.)
- Problems can arise when files grow, or if the exact size of a file is unknown at creation time:
    - Over-estimation of the file's final size increases external fragmentation and wastes disk space.
    - Under-estimation may require that a file be moved or a process aborted if the file grows beyond its originally allocated space.
    - If a file grows slowly over a long time period and the total final space must be allocated initially, then a lot of space becomes unusable before the file fills the space.
- A variation is to allocate file space in large contiguous chunks, called *extents.* When a file outgrows its original extent, then an additional one is allocated. (For example an extent may be the size of a complete track or even cylinder, aligned on an appropriate track or cylinder boundary.) The high-performance files system Veritas uses extents to optimize performance.


directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2
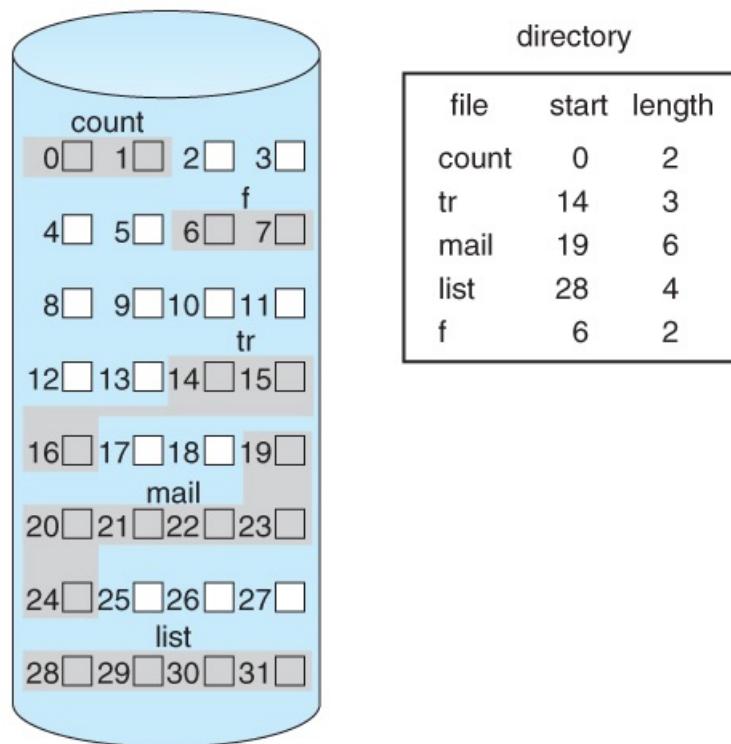
Figure 12.5 - Contiguous allocation of disk space.

12.4.2 Linked Allocation

- Disk files can be stored as linked lists, with the expense of the storage space consumed by each link. (E.g. a block may be 508 bytes instead of 512.)
- Linked allocation involves no external fragmentation, does not require pre-known file sizes, and allows files to grow dynamically at any time.
- Unfortunately linked allocation is only efficient for sequential access files, as random access requires starting at the beginning of the list for each new location access.
- Allocating *clusters* of blocks reduces the space wasted by pointers, at the cost of internal fragmentation.
- Another big problem with linked allocation is reliability if a pointer is lost or damaged. Doubly linked lists provide some protection, at the cost of additional overhead and wasted space.
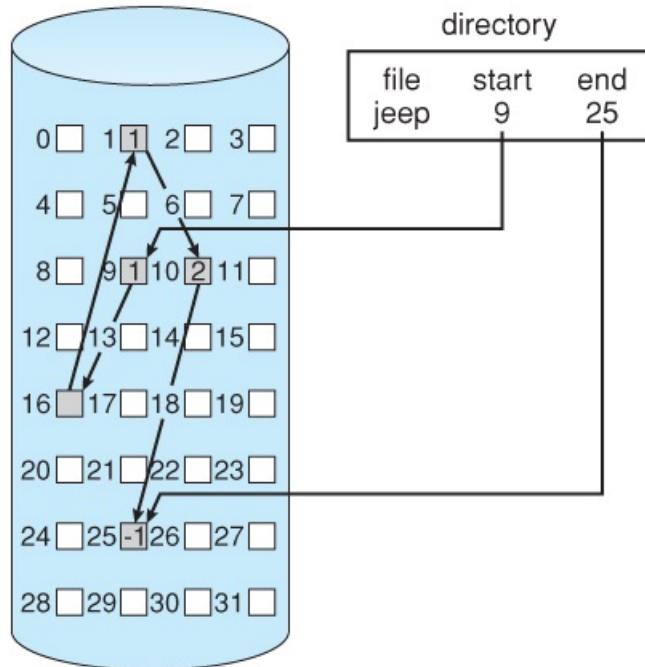
Figure 12.6 - Linked allocation of disk space.

- The *File Allocation Table, FAT,* used by DOS is a variation of linked allocation, where all the links are stored in a separate table at the beginning of the disk. The benefit of this approach is that the FAT table can be cached in memory, greatly improving random access speeds.
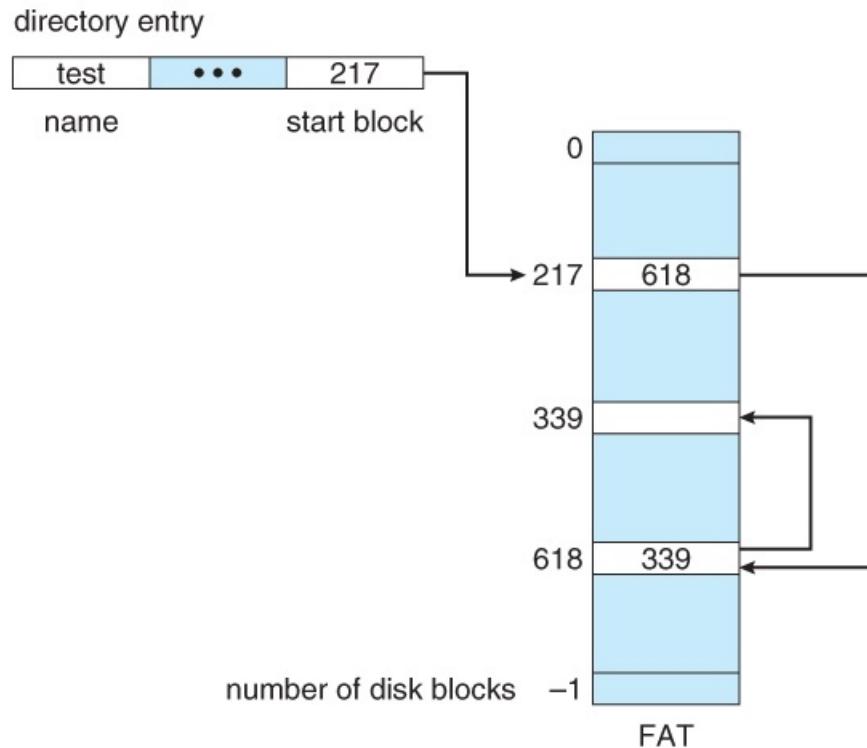

Figure 12.7 File-allocation table.

12.4.3 Indexed Allocation

- *Indexed Allocation* combines all of the indexes for accessing each file into a common block (for that file), as opposed to spreading them all over the disk or storing them in a FAT table.
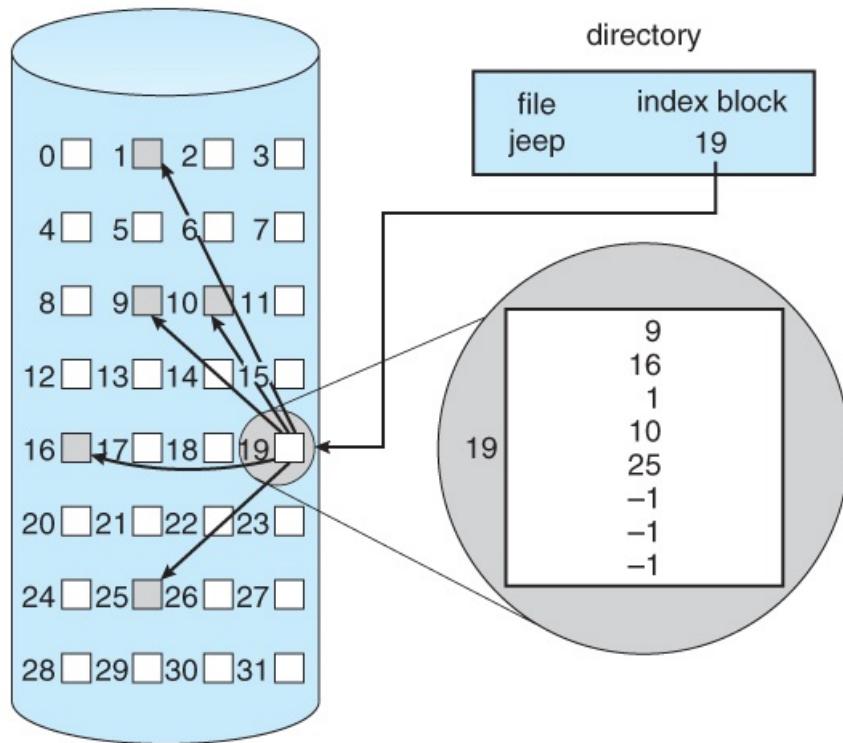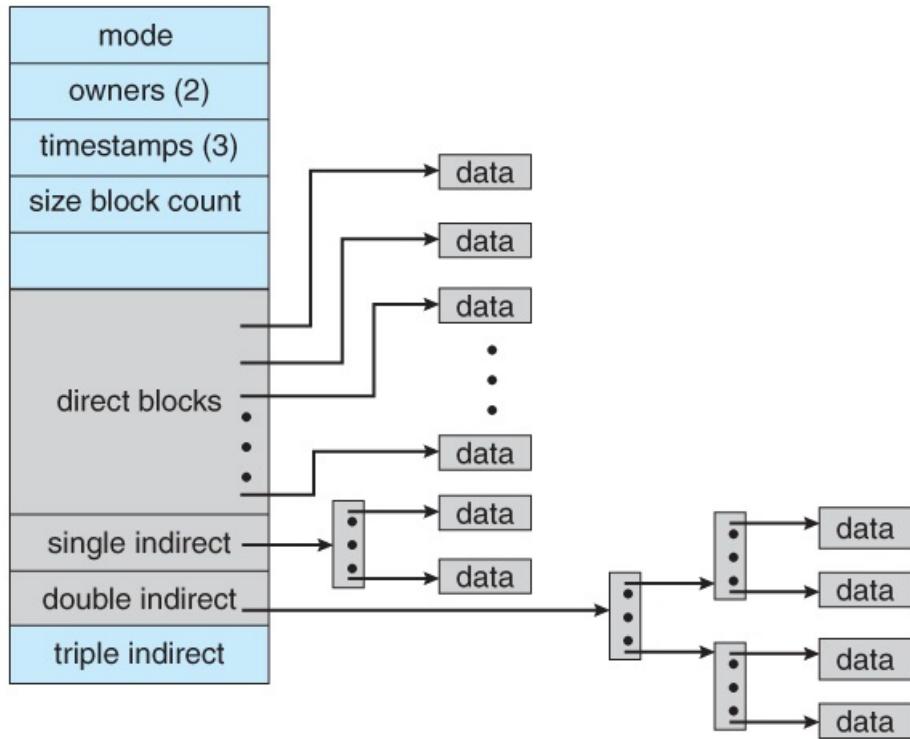
Figure 12.8 - Indexed allocation of disk space.

- Some disk space is wasted (relative to linked lists or FAT tables) because an entire index block must be allocated for each file, regardless of how many data blocks the file contains. This leads to questions of how big the index block should be, and how it should be implemented. There are several approaches:
    - **Linked Scheme -** An index block is one disk block, which can be read and written in a single disk operation. The first index block contains some header information, the first N block addresses, and if necessary a pointer to additional linked index blocks.
    - **Multi-Level Index -** The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.
    - **Combined Scheme -** This is the scheme used in UNIX inodes, in which the first 12 or so data block pointers are stored directly in the inode, and then singly, doubly, and triply indirect pointers provide access to more data blocks as needed. (See below.) The advantage of this scheme is that for small files (which many are), the data blocks are readily accessible (up to 48K with 4K block sizes); files up to about 4144K (using 4K blocks) are accessible with only a single indirect block (which can be cached), and huge files are still accessible using a relatively small number of disk accesses (larger in theory than can be addressed by a 32-bit address, which is why some systems have moved to 64-bit file pointers.)

Figure 12.9 - The UNIX inode.

12.4.4 Performance

- The optimal allocation method is different for sequential access files than for random access files, and is also different for small files than for large files.
- Some systems support more than one allocation method, which may require specifying how the file is to be used (sequential or random access) at the time it is allocated. Such systems also provide conversion utilities.
- Some systems have been known to use contiguous access for small files, and automatically switch to an indexed scheme when file sizes surpass a certain threshold.
- And of course some systems adjust their allocation schemes (e.g. block sizes) to best match the characteristics of the hardware for optimum performance.

12.5 Free-Space Management
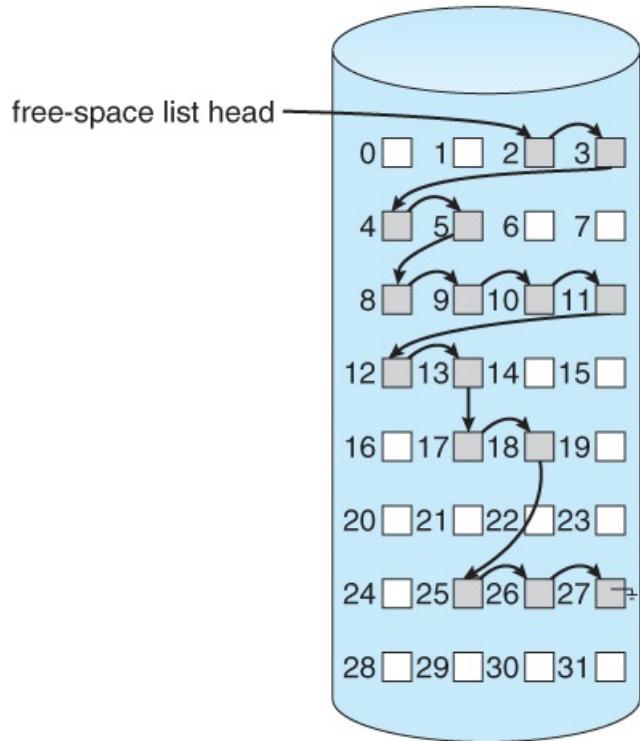
- Another important aspect of disk management is keeping track of and allocating free space.

12.5.1 Bit Vector

- One simple approach is to use a ***bit vector***, in which each bit represents a disk block, set to 1 if free or 0 if allocated.
- Fast algorithms exist for quickly finding contiguous blocks of a given size
- The down side is that a 40GB disk requires over 5MB just to store the bitmap. (For example.)

12.5.2 Linked List

- A linked list can also be used to keep track of all free blocks.
- Traversing the list and/or finding a contiguous block of a given size are not easy, but fortunately are not frequently needed operations. Generally the system just adds and removes single blocks from the beginning of the list.
- The FAT table keeps track of the free list as just one more linked list on the table.

Figure 12.10 - Linked free-space list on disk.

12.5.3 Grouping

- A variation on linked list free lists is to use links of blocks of indices of free blocks. If a block holds up to N addresses, then the first block in the linked-list contains up to N-1 addresses of free blocks and a pointer to the next block of free addresses.

12.5.4 Counting

- When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks. As long as the average length of a contiguous group of free blocks is greater than two this offers a savings in space needed for the free list. (Similar to compression techniques used for graphics images when a group of pixels all the same color is encountered.)

12.5.5 Space Maps

- Sun's ZFS file system was designed for HUGE numbers and sizes of files, directories, and even file systems.
- The resulting data structures could be VERY inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of free list bit maps if the file was spread across the disk.
- ZFS uses a combination of techniques, starting with dividing the disk up into (hundreds of) **metaslabs** of a manageable size, each having their own space map.
- Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.
- An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.
- The combination of the in-memory tree and the on-disk log provide for very fast and efficient management of these very large files and free blocks.

12.6 Efficiency and Performance

12.6.1 Efficiency

- UNIX pre-allocates inodes, which occupies space even before any files are created.
- UNIX also distributes inodes across the disk, and tries to store data files near their inode, to reduce the distance of disk seeks between the inodes and the data.

- Some systems use variable size clusters depending on the file size.
- The more data that is stored in a directory (e.g. last access time), the more often the directory blocks have to be re-written.
- As technology advances, addressing schemes have had to grow as well.
  - Sun's ZFS file system uses 128-bit pointers, which should theoretically never need to be expanded. (The mass required to store 2^128 bytes with atomic storage would be at least 272 trillion kilograms!)
- Kernel table sizes used to be fixed, and could only be changed by rebuilding the kernels. Modern tables are dynamically allocated, but that requires more complicated algorithms for accessing them.

12.6.2 Performance

- Disk controllers generally include on-board caching. When a seek is requested, the heads are moved into place, and then an entire track is read, starting from whatever sector is currently under the heads (reducing latency.) The requested sector is returned and the unrequested portion of the track is cached in the disk's electronics.
- Some OSes cache disk blocks they expect to need again in a ***buffer cache.***
- A ***page cache*** connected to the virtual memory system is actually more efficient as memory addresses do not need to be converted to disk block addresses and back again.
- Some systems (Solaris, Linux, Windows 2000, NT, XP) use page caching for both process pages and file data in a ***unified virtual memory.***
- Figures 11.11 and 11.12 show the advantages of the ***unified buffer cache*** found in some versions of UNIX and Linux - Data does not need to be stored twice, and problems of inconsistent buffer information are avoided.
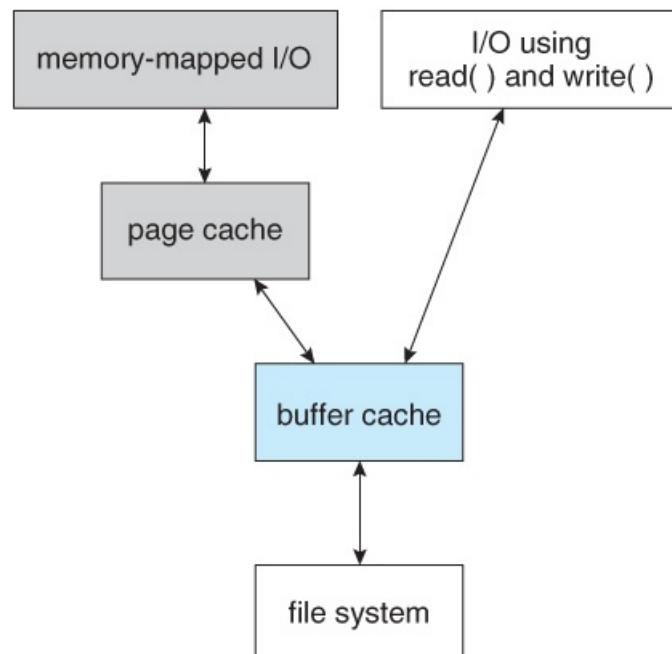

Figure 12.11 - I/O without a unified buffer cache.
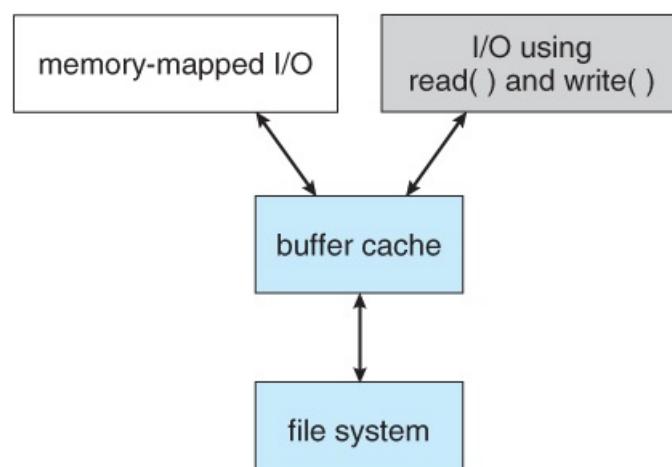

Figure 12.12 - I/O using a unified buffer cache.

- Page replacement strategies can be complicated with a unified cache, as one needs to decide whether to replace process or file pages, and how many pages to guarantee to each category of pages. Solaris, for example, has gone through many variations, resulting in **priority paging** giving process pages priority over file I/O pages, and setting limits so that neither can knock the other completely out of memory.
- Another issue affecting performance is the question of whether to implement **synchronous writes** or **asynchronous writes.** Synchronous writes occur in the order in which the disk subsystem receives them, without caching; Asynchronous writes are cached, allowing the disk subsystem to schedule writes in a more efficient order (See Chapter 12.) Metadata writes are often done synchronously. Some systems support flags to the open call requiring that writes be synchronous, for example for the benefit of database systems that require their writes be performed in a required order.
- The type of file access can also have an impact on optimal page replacement policies. For example, LRU is not necessarily a good policy for sequential access files. For these types of files progression normally goes in a forward direction only, and the most recently used page will not be needed again until after the file has been rewound and re-read from the beginning, (if it is ever needed at all.) On the other hand, we can expect to need the next page in the file fairly soon. For this reason sequential access files often take advantage of two special policies:
    - **Free-behind** frees up a page as soon as the next page in the file is requested, with the assumption that we are now done with the old page and won't need it again for a long time.
    - **Read-ahead** reads the requested page and several subsequent pages at the same time, with the assumption that those pages will be needed in the near future. This is similar to the track caching that is already performed by the disk controller, except it saves the future latency of transferring data from the disk controller memory into motherboard main memory.
- The caching system and asynchronous writes speed up disk writes considerably, because the disk subsystem can schedule physical writes to the disk to minimize head movement and disk seek times. (See Chapter 12.) Reads, on the other hand, must be done more synchronously in spite of the caching system, with the result that disk writes can counter-intuitively be much faster on average than disk reads.

12.7 Recovery

12.7.1 Consistency Checking

- The storing of certain data structures (e.g. directories and inodes) in memory and the caching of disk operations can speed up performance, but what happens in the result of a system crash? All volatile memory structures are lost, and the information stored on the hard drive may be left in an inconsistent state.
- A **Consistency Checker** (fsck in UNIX, chkdsk or scandisk in Windows) is often run at boot time or mount time, particularly if a filesystem was not closed down properly. Some of the problems that these tools look for include:
    - Disk blocks allocated to files and also listed on the free list.
    - Disk blocks neither allocated to files nor on the free list.
    - Disk blocks allocated to more than one file.
    - The number of disk blocks allocated to a file inconsistent with the file's stated size.
    - Properly allocated files / inodes which do not appear in any directory entry.
    - Link counts for an inode not matching the number of references to that inode in the directory structure.
    - Two or more identical file names in the same directory.
    - Illegally linked directories, e.g. cyclical relationships where those are not allowed, or files/directories that are not accessible from the root of the directory tree.
    - Consistency checkers will often collect questionable disk blocks into new files with names such as chk00001.dat. These files may contain valuable information that would otherwise be lost, but in most cases they can be safely deleted, (returning those disk blocks to the free list.)
- UNIX caches directory information for reads, but any changes that affect space allocation or metadata changes are written synchronously, before any of the corresponding data blocks are written to.

12.7.2 Log-Structured File Systems (was 11.8)

- **Log-based transaction-oriented** (a.k.a. **journaling**) filesystems borrow techniques developed for databases, guaranteeing that any given transaction either completes successfully or can be rolled back to a safe state before the transaction commenced:
    - All metadata changes are written sequentially to a log.
    - A set of changes for performing a specific task (e.g. moving a file) is a **transaction**.
    - As changes are written to the log they are said to be **committed,** allowing the system to return to its work.
    - In the meantime, the changes from the log are carried out on the actual filesystem, and a pointer keeps

track of which changes in the log have been completed and which have not yet been completed.

- When all changes corresponding to a particular transaction have been completed, that transaction can be safely removed from the log.
- At any given time, the log will contain information pertaining to uncompleted transactions only, e.g. actions that were committed but for which the entire transaction has not yet been completed.
  - From the log, the remaining transactions can be completed,
  - or if the transaction was aborted, then the partially completed changes can be undone.

12.7.3 Other Solutions (New)
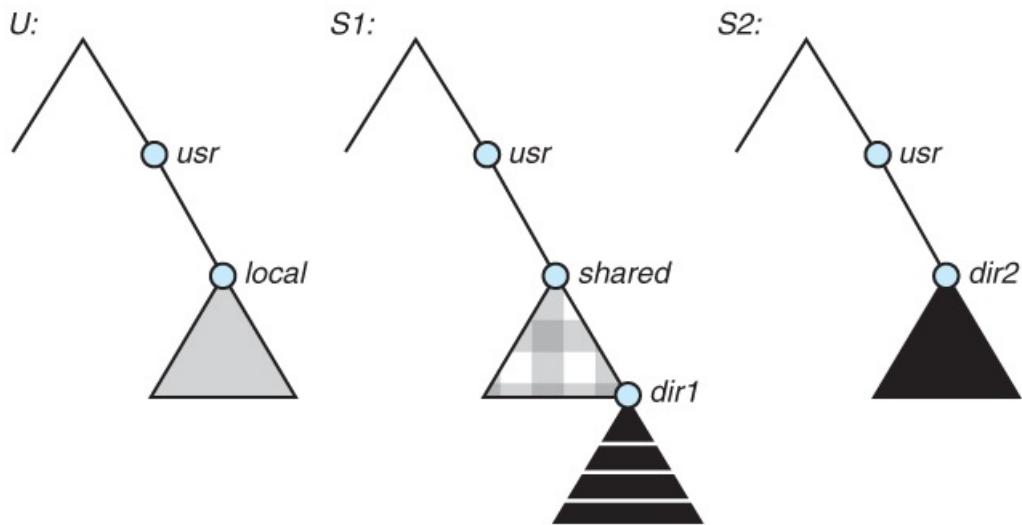
- Sun's ZFS and Network Appliance's WAFL file systems take a different approach to file system consistency.
- No blocks of data are ever over-written in place. Rather the new data is written into fresh new blocks, and after the transaction is complete, the metadata (data block pointers) is updated to point to the new blocks.
  - The old blocks can then be freed up for future use.
  - Alternatively, if the old blocks and old metadata are saved, then a *snapshot* of the system in its original state is preserved. This approach is taken by WAFL.
- ZFS combines this with check-summing of all metadata and data blocks, and RAID, to ensure that no inconsistencies are possible, and therefore ZFS does not incorporate a consistency checker.

12.7.4 Backup and Restore
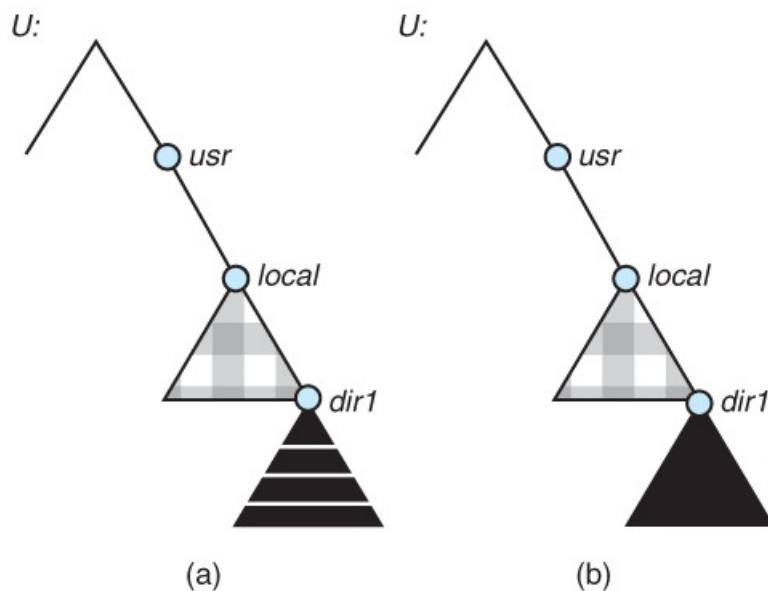
- In order to recover lost data in the event of a disk crash, it is important to conduct backups regularly.
- Files should be copied to some removable medium, such as magnetic tapes, CDs, DVDs, or external removable hard drives.
- A full backup copies every file on a filesystem.
- Incremental backups copy only files which have changed since some previous time.
- A combination of full and incremental backups can offer a compromise between full recoverability, the number and size of backup tapes needed, and the number of tapes that need to be used to do a full restore. For example, one strategy might be:
  - At the beginning of the month do a full backup.
  - At the end of the first and again at the end of the second week, backup all files which have changed since the beginning of the month.
  - At the end of the third week, backup all files that have changed since the end of the second week.
  - Every day of the month not listed above, do an incremental backup of all files that have changed since the most recent of the weekly backups described above.
- Backup tapes are often reused, particularly for daily backups, but there are limits to how many times the same tape can be used.
- Every so often a full backup should be made that is kept "forever" and not overwritten.
- ***Backup tapes should be tested, to ensure that they are readable!***
- For optimal security, backup tapes should be kept off-premises, so that a fire or burglary cannot destroy both the system and the backups. There are companies (e.g. Iron Mountain) that specialize in the secure off-site storage of critical backup information.
- ***Keep your backup tapes secure - The easiest way for a thief to steal all your data is to simply pocket your backup tapes!***
- Storing important files on more than one computer can be an alternate though less reliable form of backup.
- Note that incremental backups can also help users to get back a previous version of a file that they have since changed in some way.
- Beware that backups can help forensic investigators recover e-mails and other files that users had though they had deleted!

12.8 NFS (Optional)

12.8.1 Overview

Figure 12.13 - Three independent file systems.

Figure 12.14 - Mounting in NFS. (a) Mounts. (b) Cascading mounts.

12.8.2 The Mount Protocol

- The NFS mount protocol is similar to the local mount protocol, establishing a connection between a specific local directory (the mount point) and a specific device from a remote system.
- Each server maintains an **export list** of the local filesystems (directory sub-trees) which are exportable, who they are exportable to, and what restrictions apply (e.g. read-only access.)
- The server also maintains a list of currently connected clients, so that they can be notified in the event of the server going down and for other reasons.
- Automount and autounmount are supported.

12.8.3 The NFS Protocol

- Implemented as a set of remote procedure calls (RPCs):
    - Searching for a file in a directory
    - REading a set of directory entries
    - Manipulating links and directories
    - Accessing file attributes
   - Reading and writing files
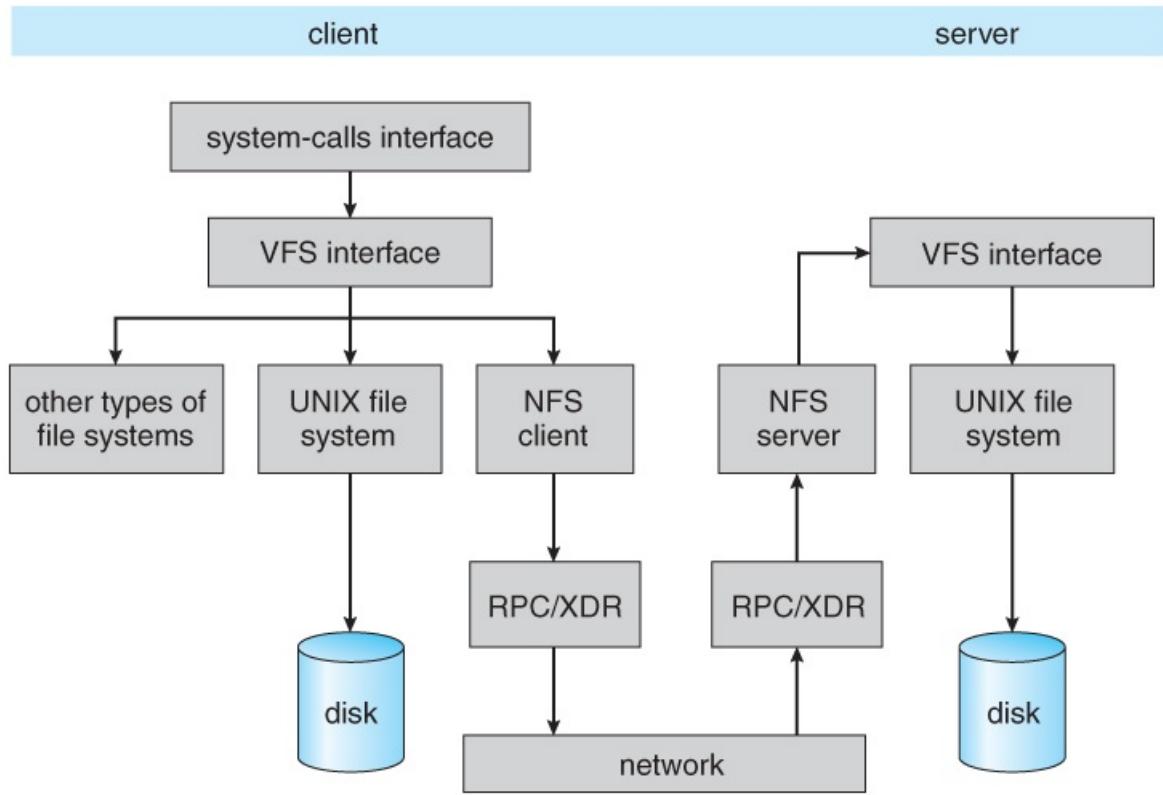
Figure 12.15 - Schematic view of the NFS architecture.

12.8.4 Path-Name Translation

11.8.5 Remote Operations

- Buffering and caching improve performance, but can cause a disparity in local versus remote views of the same file(s).

12.9 Example: The WAFL File System (Optional)

- Write Anywhere File Layout
- Designed for a specific hardware architecture.
- **Snapshots** record the state of the system at regular or irregular intervals.
  - The snapshot just copies the inode pointers, not the actual data.
  - Used pages are not overwritten, so updates are fast.
  - Blocks keep counters for how many snapshots are pointing to that block - When the counter reaches zero, then the block is considered free.
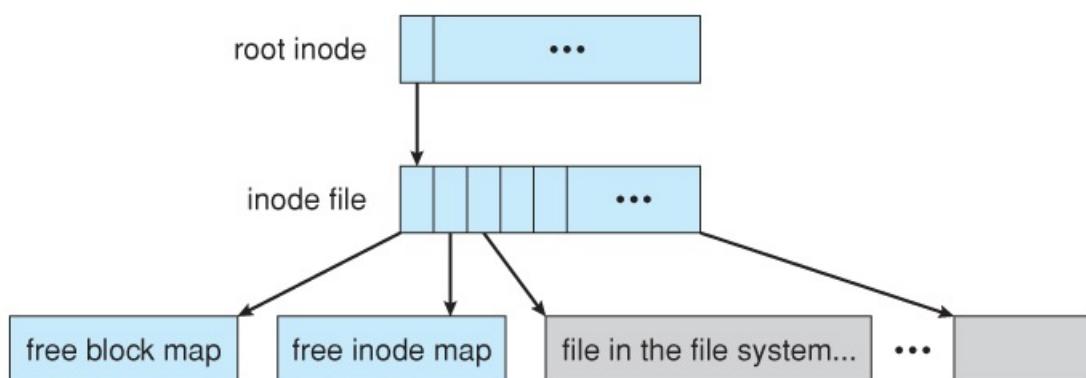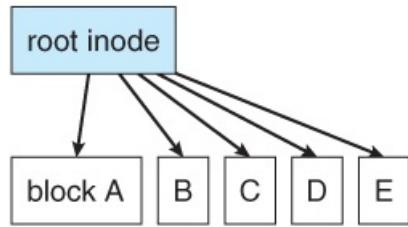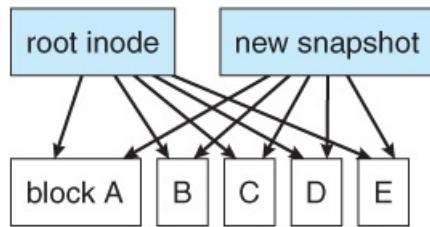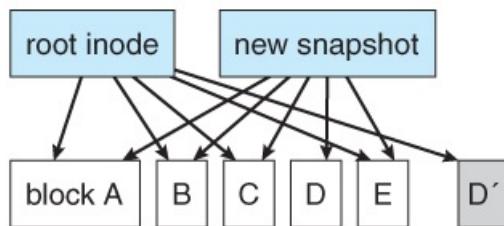

Figure 12.16 - The WAFL file layout.

(a) Before a snapshot.

(b) After a snapshot, before any blocks change.

(c) After block D has changed to D´.

Figure 12.17 - Snapshots in WAFL

12.10 Summary

I/O Systems

References:
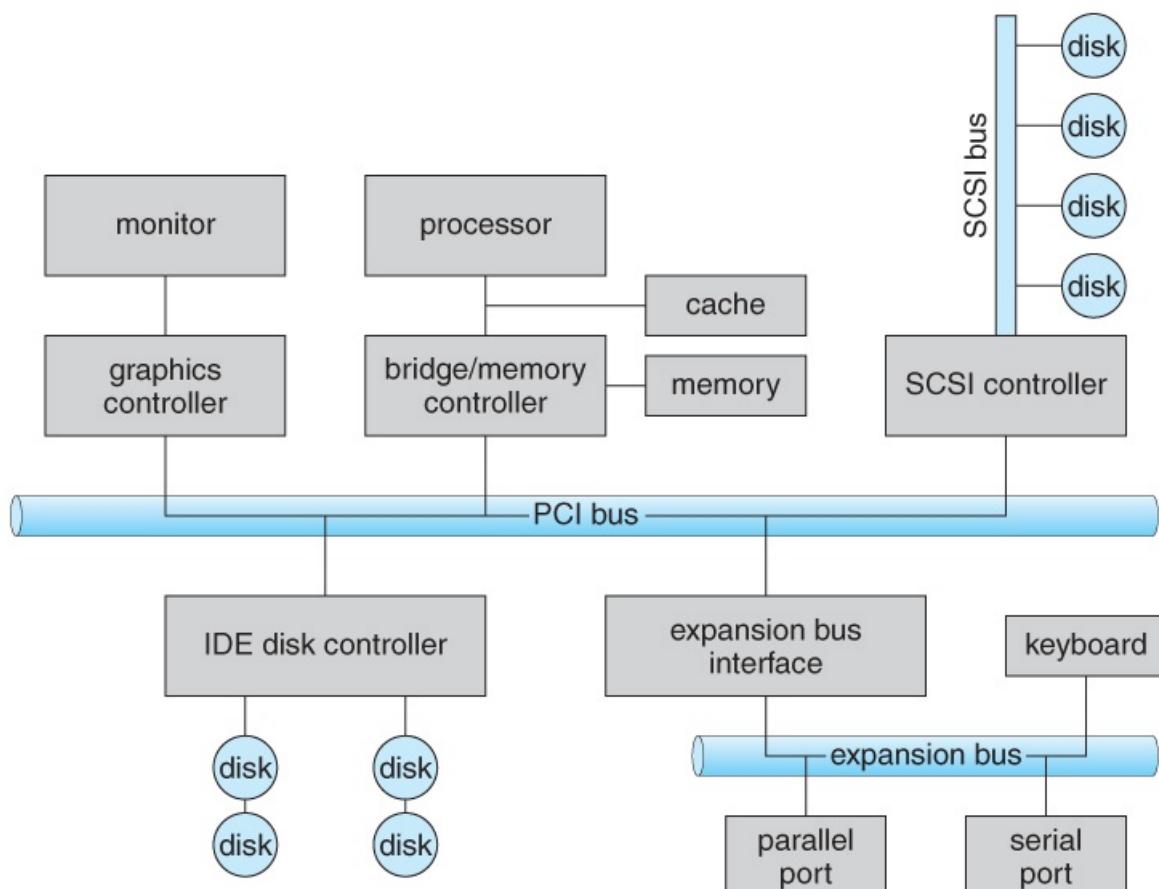
1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Eighth Edition ", Chapter 13

13.1 Overview

- Management of I/O devices is a very important part of the operating system - so important and so varied that entire I/O subsystems are devoted to its operation. (Consider the range of devices on a modern computer, from mice, keyboards, disk drives, display adapters, USB devices, network connections, audio I/O, printers, special devices for the handicapped, and many special-purpose peripherals.)
- I/O Subsystems must contend with two (conflicting?) trends: (1) The gravitation towards standard interfaces for a wide range of devices, making it easier to add newly developed devices to existing systems, and (2) the development of entirely new types of devices, for which the existing standard interfaces are not always easy to apply.
- **Device drivers** are modules that can be plugged into an OS to handle a particular device or category of similar devices.

13.2 I/O Hardware

- I/O devices can be roughly categorized as storage, communications, user-interface, and other
- Devices communicate with the computer via signals sent over wires or through the air.
- Devices connect with the computer via **ports**, e.g. a serial or parallel port.
- A common set of wires connecting multiple devices is termed a **bus.**
  - Buses include rigid protocols for the types of messages that can be sent across the bus and the procedures for resolving contention issues.
  - Figure 13.1 below illustrates three of the four bus types commonly found in a modern PC:
    1. The **PCI bus** connects high-speed high-bandwidth devices to the memory subsystem (and the CPU.)
    2. The **expansion bus** connects slower low-bandwidth devices, which typically deliver data one character at a time (with buffering.)
    3. The **SCSI bus** connects a number of SCSI devices to a common SCSI controller.
    4. A **daisy-chain bus,** (not shown) is when a string of devices is connected to each other like beads on a chain, and only one of the devices is directly connected to the host.

Figure 13.1 - A typical PC bus structure.

- One way of communicating with devices is through *registers* associated with each port. Registers may be one to four bytes in size, and may typically include (a subset of) the following four:
    1. The *data-in register* is read by the host to get input from the device.
    2. The *data-out register* is written by the host to send output.
    3. The *status register* has bits read by the host to ascertain the status of the device, such as idle, ready for input, busy, error, transaction complete, etc.
    4. The *control register* has bits written by the host to issue commands or to change settings of the device such as parity checking, word length, or full- versus half-duplex operation.
- Figure 13.2 shows some of the most common I/O port address ranges.

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

**Figure 13.2 - Device I/O port locations on PCs (partial).**

- Another technique for communicating with devices is *memory-mapped I/O.*
    - In this case a certain portion of the processor's address space is mapped to the device, and communications occur by reading and writing directly to/from those memory areas.
    - Memory-mapped I/O is suitable for devices which must move large quantities of data quickly, such as graphics cards.
    - Memory-mapped I/O can be used either instead of or more often in combination with traditional registers. For example, graphics cards still use registers for control information such as setting the video mode.
    - A potential problem exists with memory-mapped I/O, if a process is allowed to write directly to the address space used by a memory-mapped I/O device.
    - (Note: Memory-mapped I/O is not the same thing as direct memory access, DMA. See section 13.2.3 below.)

13.2.1 Polling

- One simple means of device *handshaking* involves polling:
    1. The host repeatedly checks the *busy bit* on the device until it becomes clear.
    2. The host writes a byte of data into the data-out register, and sets the *write bit* in the command register (in either order.)
    3. The host sets the *command ready bit* in the command register to notify the device of the pending command.
    4. When the device controller sees the command-ready bit set, it first sets the busy bit.
    5. Then the device controller reads the command register, sees the write bit set, reads the byte of data from the data-out register, and outputs the byte of data.
    6. The device controller then clears the *error bit* in the status register, the command-ready bit, and finally clears the busy bit, signaling the completion of the operation.
- Polling can be very fast and efficient, if both the device and the controller are fast and if there is significant data to transfer. It becomes inefficient, however, if the host must wait a long time in the busy loop waiting for the device, or if frequent checks need to be made for data that is infrequently there.

13.2.2 Interrupts

- Interrupts allow devices to notify the CPU when they have data to transfer or when an operation is complete, allowing the CPU to perform other duties when no I/O transfers need its immediate attention.
- The CPU has an **interrupt-request line** that is sensed after every instruction.
    - A device's controller **raises** an interrupt by asserting a signal on the interrupt request line.
    - The CPU then performs a state save, and transfers control to the **interrupt handler** routine at a fixed address in memory. (The CPU **catches** the interrupt and **dispatches** the interrupt handler.)
    - The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a **return from interrupt** instruction to return control to the CPU. (The interrupt handler **clears** the interrupt by servicing the device.)
        - (Note that the state restored does not need to be the same state as the one that was saved when the interrupt went off. See below for an example involving time-slicing.)
- Figure 13.3 illustrates the interrupt-driven I/O procedure:
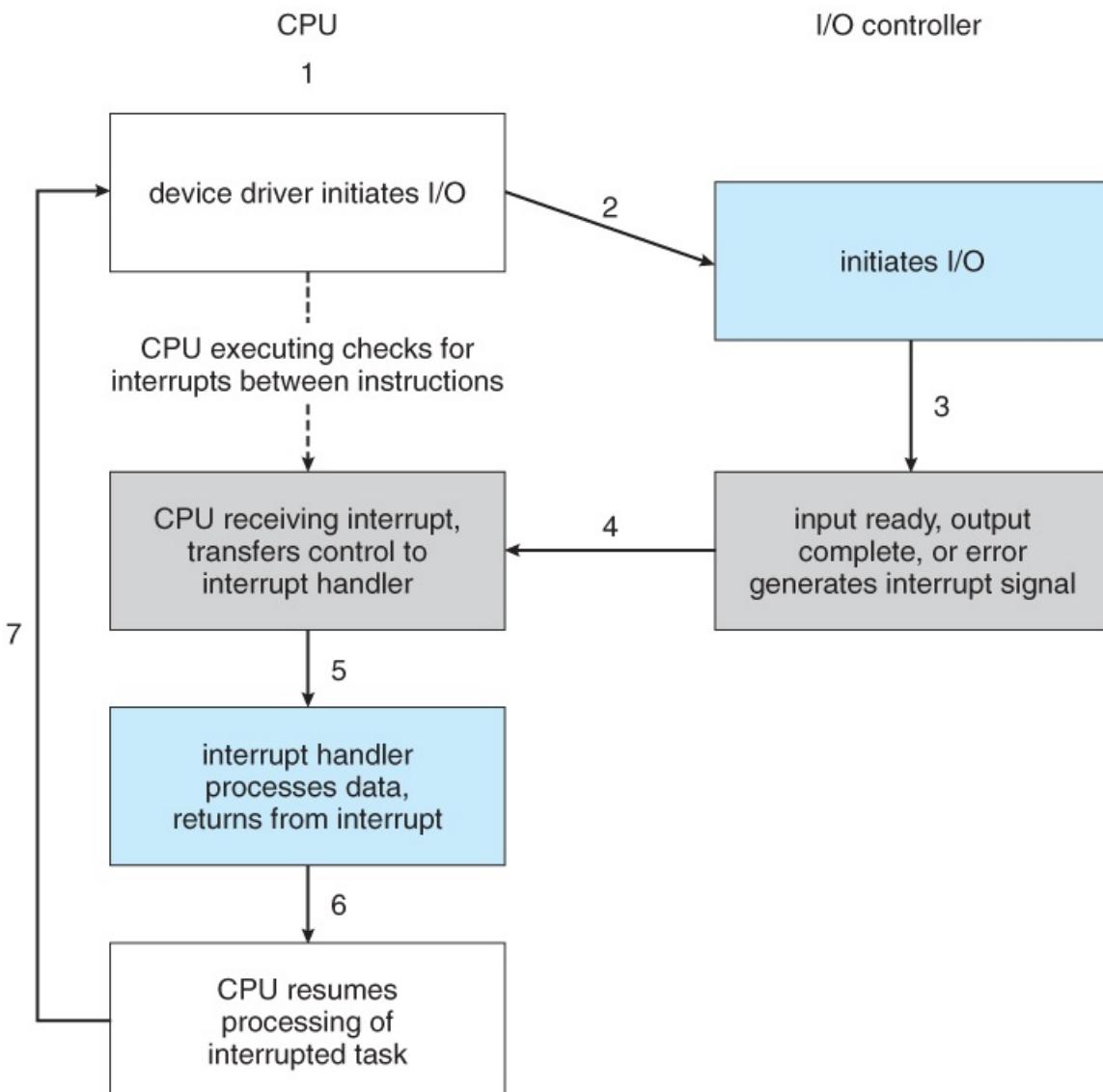
Figure 13.3 - Interrupt-driven I/O cycle.

- The above description is adequate for simple interrupt-driven I/O, but there are three needs in modern computing which complicate the picture:
    1. The need to defer interrupt handling during critical processing,
    2. The need to determine **which** interrupt handler to invoke, without having to poll all devices to see which one needs attention, and
    3. The need for multi-level interrupts, so the system can differentiate between high- and low-priority interrupts for proper response.
- These issues are handled in modern computer architectures with **interrupt-controller** hardware.
    - Most CPUs now have two interrupt-request lines: One that is **non-maskable** for critical error conditions and one that is **maskable,** that the CPU can temporarily ignore during critical processing.
    - The interrupt mechanism accepts an **address,** which is usually one of a small set of numbers for an offset into a table called the **interrupt vector.** This table (usually located at physical address zero ?) holds the addresses of routines prepared to process specific interrupts.

- The number of possible interrupt handlers still exceeds the range of defined interrupt numbers, so multiple handlers can be **interrupt chained**. Effectively the addresses held in the interrupt vectors are the head pointers for linked-lists of interrupt handlers.
- Figure 13.4 shows the Intel Pentium interrupt vector. Interrupts 0 to 31 are non-maskable and reserved for serious hardware and other errors. Maskable interrupts, including normal device I/O interrupts begin at interrupt 32.
- Modern interrupt hardware also supports **interrupt priority levels**, allowing systems to mask off only lower-priority interrupts while servicing a high-priority interrupt, or conversely to allow a high-priority signal to interrupt the processing of a low-priority one.

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Figure 13.4 - Intel Pentium processor event-vector table.

- At boot time the system determines which devices are present, and loads the appropriate handler addresses into the interrupt table.
- During operation, devices signal errors or the completion of commands via interrupts.
- Exceptions, such as dividing by zero, invalid memory accesses, or attempts to access kernel mode instructions can be signaled via interrupts.
- Time slicing and context switches can also be implemented using the interrupt mechanism.
    - The scheduler sets a hardware timer before transferring control over to a user process.
    - When the timer raises the interrupt request line, the CPU performs a state-save, and transfers control over to the proper interrupt handler, which in turn runs the scheduler.
    - The scheduler does a state-restore of a **different** process before resetting the timer and issuing the return-from-interrupt instruction.
- A similar example involves the paging system for virtual memory - A page fault causes an interrupt, which in turn issues an I/O request and a context switch as described above, moving the interrupted process into the wait queue and selecting a different process to run. When the I/O request has completed (i.e. when the requested page has been loaded up into physical memory), then the device interrupts, and the interrupt handler moves the process from the wait queue into the ready queue, (or depending on scheduling algorithms and policies, may go ahead and context switch it back onto the CPU.)
- System calls are implemented via **software interrupts,** a.k.a. **traps.** When a (library) program needs work performed in kernel mode, it sets command information and possibly data addresses in certain registers, and then raises a software interrupt. (E.g. 21 hex in DOS.) The system does a state save and then calls on the proper interrupt handler to process the request in kernel mode. Software interrupts generally have low

priority, as they are not as urgent as devices with limited buffering space.

- Interrupts are also used to control kernel operations, and to schedule activities for optimal performance. For example, the completion of a disk read operation involves **two** interrupts:
  - A high-priority interrupt acknowledges the device completion, and issues the next disk request so that the hardware does not sit idle.
  - A lower-priority interrupt transfers the data from the kernel memory space to the user space, and then transfers the process from the waiting queue to the ready queue.
- The Solaris OS uses a multi-threaded kernel and priority threads to assign different threads to different interrupt handlers. This allows for the "simultaneous" handling of multiple interrupts, and the assurance that high-priority interrupts will take precedence over low-priority ones and over user processes.

13.2.3 Direct Memory Access

- For devices that transfer large quantities of data (such as disk controllers), it is wasteful to tie up the CPU transferring data in and out of registers one byte at a time.
- Instead this work can be off-loaded to a special processor, known as the ***Direct Memory Access, DMA, Controller.***
- The host issues a command to the DMA controller, indicating the location where the data is located, the location where the data is to be transferred to, and the number of bytes of data to transfer. The DMA controller handles the data transfer, and then interrupts the CPU when the transfer is complete.
- A simple DMA controller is a standard component in modern PCs, and many ***bus-mastering*** I/O cards contain their own DMA hardware.
- Handshaking between DMA controllers and their devices is accomplished through two wires called the DMA-request and DMA-acknowledge wires.
- While the DMA transfer is going on the CPU does not have access to the PCI bus (including main memory), but it does have access to its internal registers and primary and secondary caches.
- DMA can be done in terms of either physical addresses or virtual addresses that are mapped to physical addresses. The latter approach is known as ***Direct Virtual Memory Access, DVMA,*** and allows direct data transfer from one memory-mapped device to another without using the main memory chips.
- Direct DMA access by user processes can speed up operations, but is generally forbidden by modern systems for security and protection reasons. (I.e. DMA is a kernel-mode operation.)
- Figure 13.5 below illustrates the DMA process.
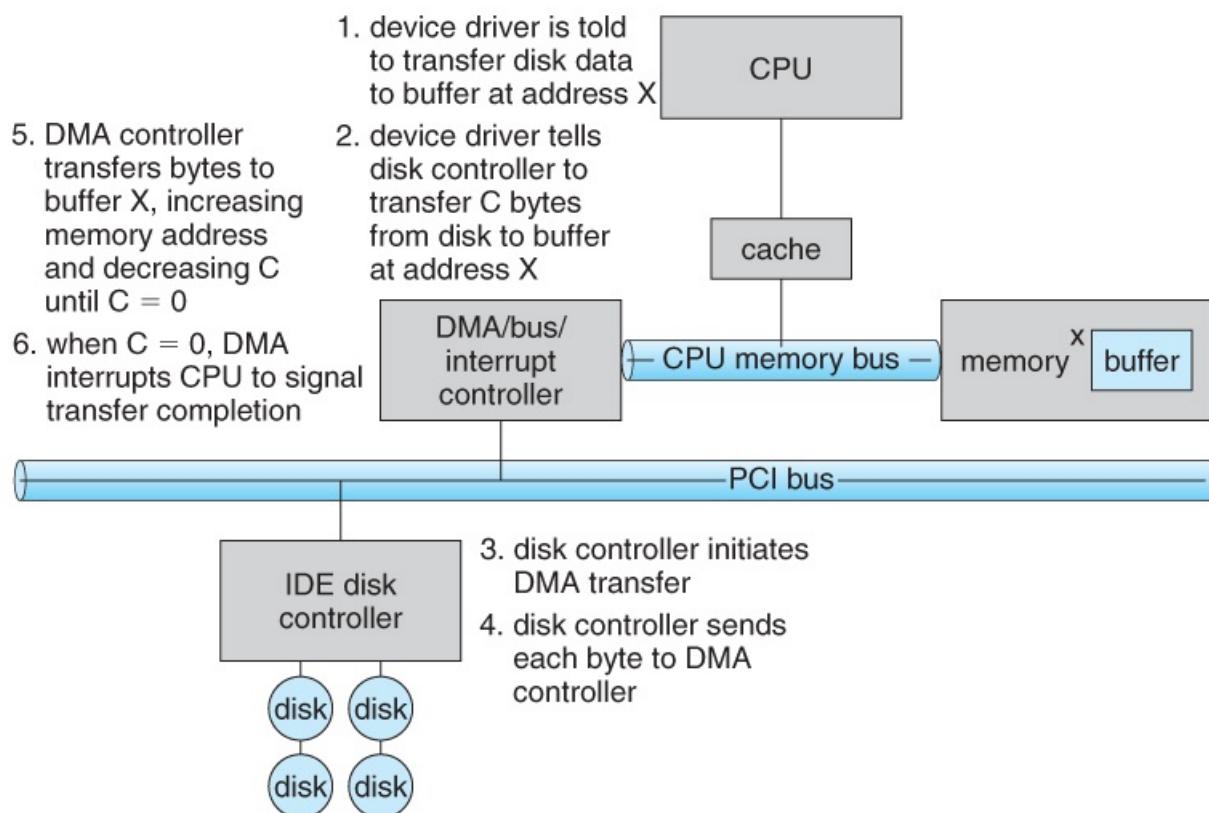
Figure 13.5 - Steps in a DMA transfer.

13.2.4 I/O Hardware Summary

13.3 Application I/O Interface

- User application access to a wide variety of different devices is accomplished through layering, and through encapsulating all of the device-specific code into **device drivers**, while application layers are presented with a common interface for all (or at least large general categories of) devices.
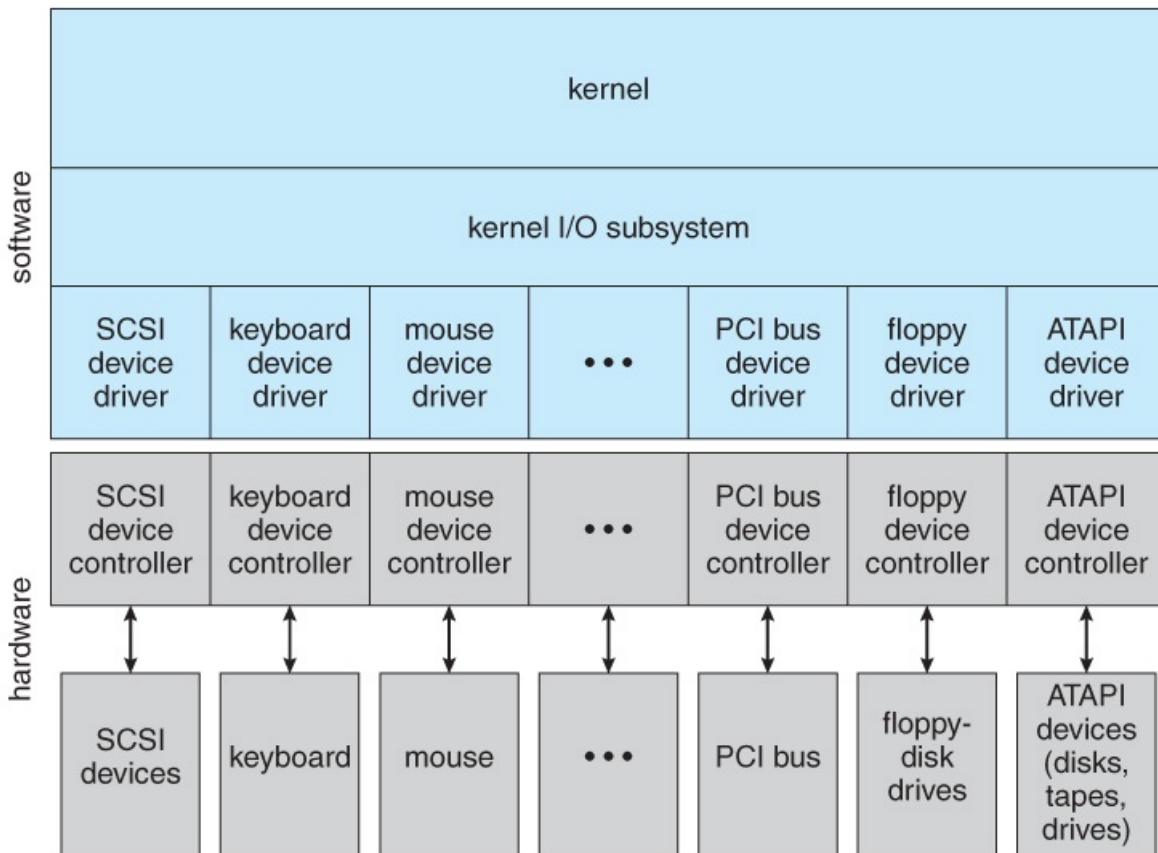
Figure 13.6 - A kernel I/O structure.

- Devices differ on many different dimensions, as outlined in Figure 13.7:

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Figure 13.7 - Characteristics of I/O devices.

- Most devices can be characterized as either block I/O, character I/O, memory mapped file access, or network sockets. A few devices are special, such as time-of-day clock and the system timer.
- Most OSes also have an **escape,** or **back door,** which allows applications to send commands directly to device drivers

if needed. In UNIX this is the *ioctl( )* system call (I/O Control). Ioctl( ) takes three arguments - The file descriptor for the device driver being accessed, an integer indicating the desired function to be performed, and an address used for communicating or transferring additional information.

13.3.1 Block and Character Devices

- **Block devices** are accessed a block at a time, and are indicated by a "b" as the first character in a long listing on UNIX systems. Operations supported include read( ), write( ), and seek( ).
  - Accessing blocks on a hard drive directly (without going through the filesystem structure) is called **raw I/O**, and can speed up certain operations by bypassing the buffering and locking normally conducted by the OS. (It then becomes the application's responsibility to manage those issues.)
  - A new alternative is **direct I/O,** which uses the normal filesystem access, but which disables buffering and locking operations.
- Memory-mapped file I/O can be layered on top of block-device drivers.
  - Rather than reading in the entire file, it is mapped to a range of memory addresses, and then paged into memory as needed using the virtual memory system.
  - Access to the file is then accomplished through normal memory accesses, rather than through read( ) and write( ) system calls. This approach is commonly used for executable program code.
- **Character devices** are accessed one byte at a time, and are indicated by a "c" in UNIX long listings. Supported operations include get( ) and put( ), with more advanced functionality such as reading an entire line supported by higher-level library routines.

13.3.2 Network Devices

- Because network access is inherently different from local disk access, most systems provide a separate interface for network devices.
- One common and popular interface is the **socket** interface, which acts like a cable or pipeline connecting two networked entities. Data can be put into the socket at one end, and read out sequentially at the other end. Sockets are normally full-duplex, allowing for bi-directional data transfer.
- The select( ) system call allows servers (or other applications) to identify sockets which have data waiting, without having to poll all available sockets.

13.3.3 Clocks and Timers

- Three types of time services are commonly needed in modern systems:
  - Get the current time of day.
  - Get the elapsed time (system or wall clock) since a previous event.
  - Set a timer to trigger event X at time T.
- Unfortunately time operations are not standard across all systems.
- A **programmable interrupt timer, PIT** can be used to trigger operations and to measure elapsed time. It can be set to trigger an interrupt at a specific future time, or to trigger interrupts periodically on a regular basis.
  - The scheduler uses a PIT to trigger interrupts for ending time slices.
  - The disk system may use a PIT to schedule periodic maintenance cleanup, such as flushing buffers to disk.
  - Networks use PIT to abort or repeat operations that are taking too long to complete. I.e. resending packets if an acknowledgement is not received before the timer goes off.
  - More timers than actually exist can be simulated by maintaining an ordered list of timer events, and setting the physical timer to go off when the next scheduled event should occur.
- On most systems the system clock is implemented by counting interrupts generated by the PIT. Unfortunately this is limited in its resolution to the interrupt frequency of the PIT, and may be subject to some drift over time. An alternate approach is to provide direct access to a high frequency hardware counter, which provides much higher resolution and accuracy, but which does not support interrupts.

13.3.4 Blocking and Non-blocking I/O

- With **blocking I/O** a process is moved to the wait queue when an I/O request is made, and moved back to the ready queue when the request completes, allowing other processes to run in the meantime.
- With **non-blocking I/O** the I/O request returns immediately, whether the requested I/O operation has (completely) occurred or not. This allows the process to check for available data without getting hung completely if it is not there.
- One approach for programmers to implement non-blocking I/O is to have a multi-threaded application, in which one thread makes blocking I/O calls (say to read a keyboard or mouse), while other threads continue to update the screen

or perform other tasks.

- A subtle variation of the non-blocking I/O is the **asynchronous I/O,** in which the I/O request returns immediately allowing the process to continue on with other tasks, and then the process is notified (via changing a process variable, or a software interrupt, or a callback function) when the I/O operation has completed and the data is available for use. (The regular non-blocking I/O returns immediately with whatever results are available, but does not complete the operation and notify the process later.)
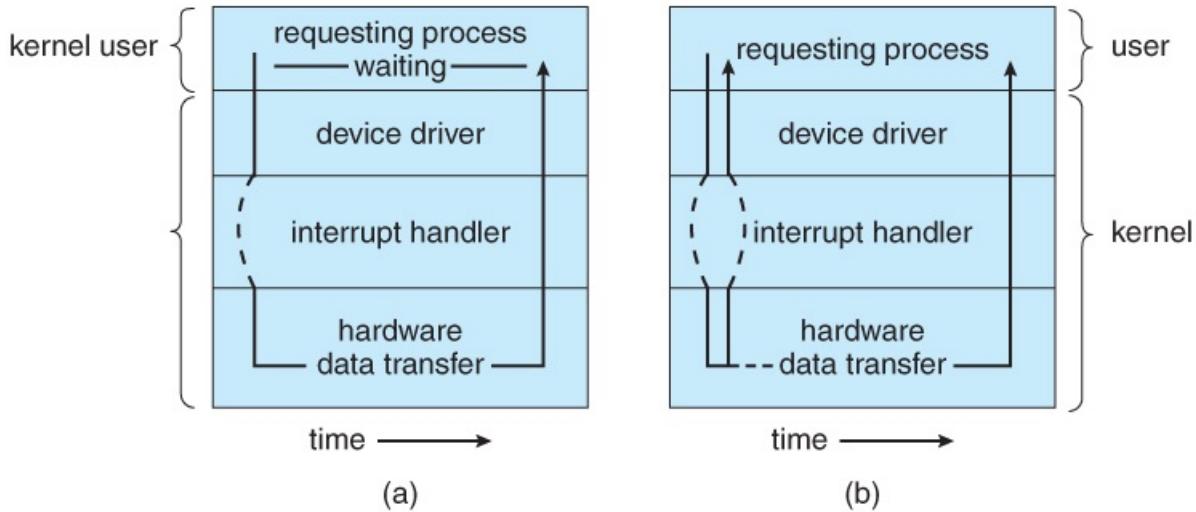
Figure 13.8 - Two I/O methods: (a) synchronous and (b) asynchronous.

13.3.5 Vectored I/O (NEW)

13.4 Kernel I/O Subsystem

13.4.1 I/O Scheduling

- Scheduling I/O requests can greatly improve overall efficiency. Priorities can also play a part in request scheduling.
- The classic example is the scheduling of disk accesses, as discussed in detail in chapter 12.
- Buffering and caching can also help, and can allow for more flexible scheduling options.
- On systems with many devices, separate request queues are often kept for each device:
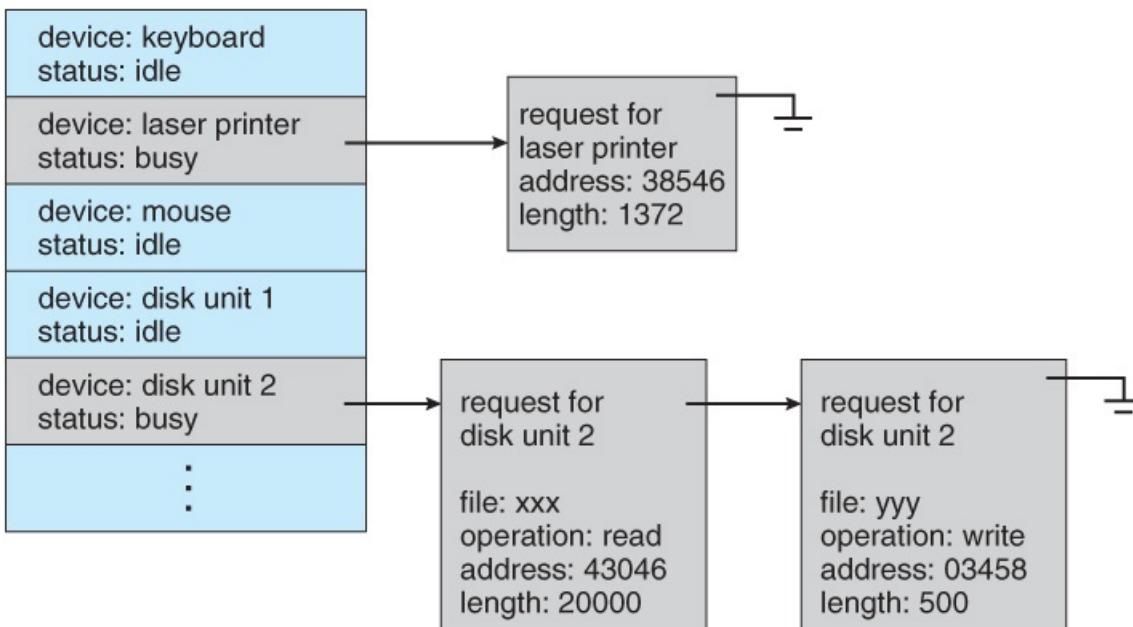
Figure 13.9 - Device-status table.

13.4.2 Buffering

- Buffering of I/O is performed for (at least) 3 major reasons:
    1. Speed differences between two devices. (See Figure 13.10 below.) A slow device may write data

into a buffer, and when the buffer is full, the entire buffer is sent to the fast device all at once. So that the slow device still has somewhere to write while this is going on, a second buffer is used, and the two buffers alternate as each becomes full. This is known as **double buffering.** (Double buffering is often used in (animated) graphics, so that one screen image can be generated in a buffer while the other (completed) buffer is displayed on the screen. This prevents the user from ever seeing any half-finished screen images.)

2. Data transfer size differences. Buffers are used in particular in networking systems to break messages up into smaller packets for transfer, and then for re-assembly at the receiving side.
3. To support **copy semantics.** For example, when an application makes a request for a disk write, the data is copied from the user's memory area into a kernel buffer. Now the application can change their copy of the data, but the data which eventually gets written out to disk is the version of the data at the time the write request was made.
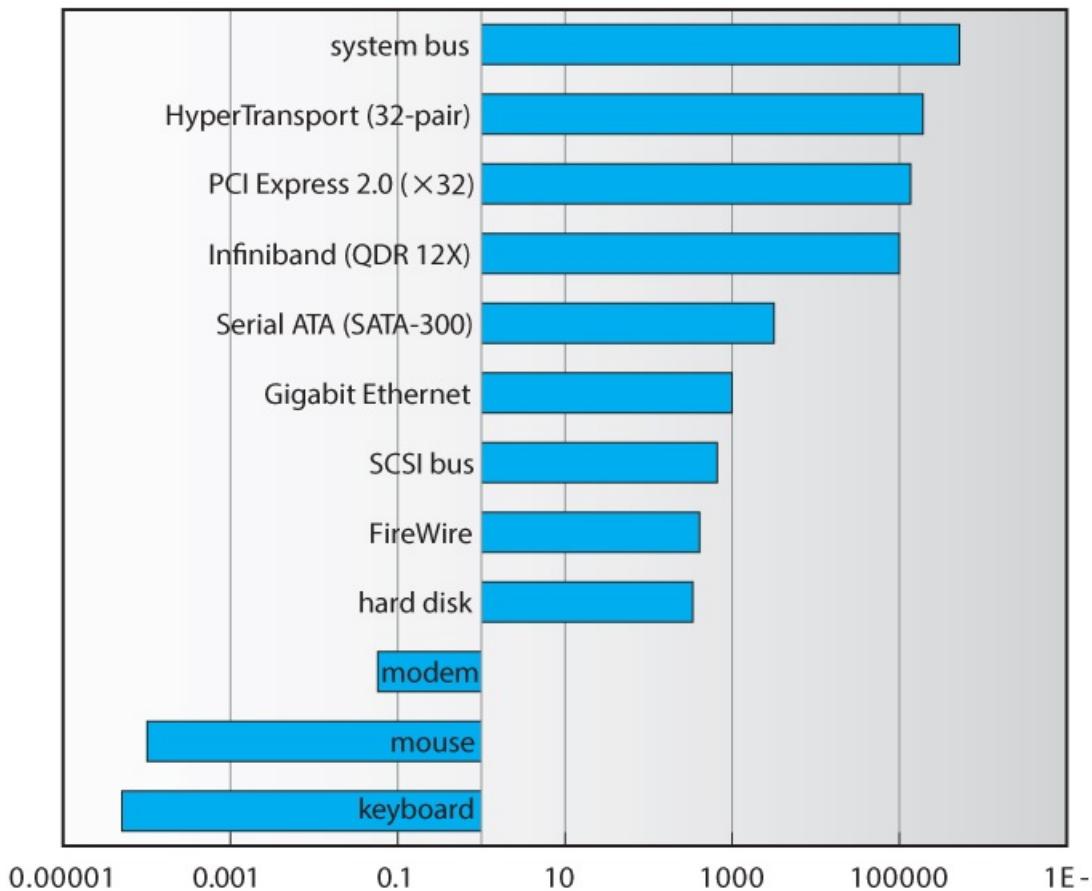

Figure 13.10 - Sun Enterprise 6000 device-transfer rates (logarithmic).

13.4.3 Caching

- Caching involves keeping a **copy** of data in a faster-access location than where the data is normally stored.
- Buffering and caching are very similar, except that a buffer may hold the only copy of a given data item, whereas a cache is just a duplicate copy of some other data stored elsewhere.
- Buffering and caching go hand-in-hand, and often the same storage space may be used for both purposes. For example, after a buffer is written to disk, then the copy in memory can be used as a cached copy, (until that buffer is needed for other purposes.)

13.4.4 Spooling and Device Reservation

- A **spool (Simultaneous Peripheral Operations On-Line)** buffers data for (peripheral) devices such as printers that cannot support interleaved data streams.
- If multiple processes want to print at the same time, they each send their print data to files stored in the spool directory. When each file is closed, then the application sees that print job as complete, and the print scheduler sends each file to the appropriate printer one at a time.
- Support is provided for viewing the spool queues, removing jobs from the queues, moving jobs from one queue to another queue, and in some cases changing the priorities of jobs in the queues.
- Spool queues can be general (any laser printer) or specific (printer number 42.)
- OSes can also provide support for processes to request / get exclusive access to a particular device, and/or

to wait until a device becomes available.

13.4.5 Error Handling

- I/O requests can fail for many reasons, either transient (buffers overflow) or permanent (disk crash).
- I/O requests usually return an error bit (or more) indicating the problem. UNIX systems also set the global variable *errno* to one of a hundred or so well-defined values to indicate the specific error that has occurred. (See errno.h for a complete listing, or man errno.)
- Some devices, such as SCSI devices, are capable of providing much more detailed information about errors, and even keep an on-board error log that can be requested by the host.

13.4.6 I/O Protection

- The I/O system must protect against either accidental or deliberate erroneous I/O.
- User applications are not allowed to perform I/O in user mode - All I/O requests are handled through system calls that must be performed in kernel mode.
- Memory mapped areas and I/O ports must be protected by the memory management system, **but** access to these areas cannot be totally denied to user programs. (Video games and some other applications need to be able to write directly to video memory for optimal performance for example.) Instead the memory protection system restricts access so that only one process at a time can access particular parts of memory, such as the portion of the screen memory corresponding to a particular window.
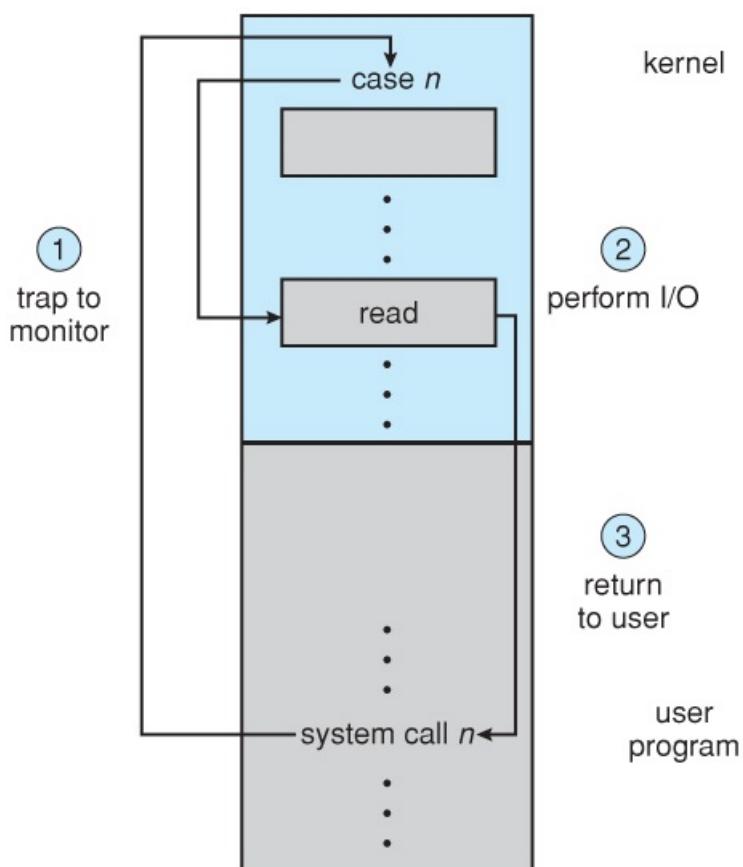
Figure 13.11 - Use of a system call to perform I/O.

13.4.7 Kernel Data Structures

- The kernel maintains a number of important data structures pertaining to the I/O system, such as the open file table.
- These structures are object-oriented, and flexible to allow access to a wide variety of I/O devices through a common interface. (See Figure 13.12 below.)
- Windows NT carries the object-orientation one step further, implementing I/O as a message-passing system from the source through various intermediaries to the device.
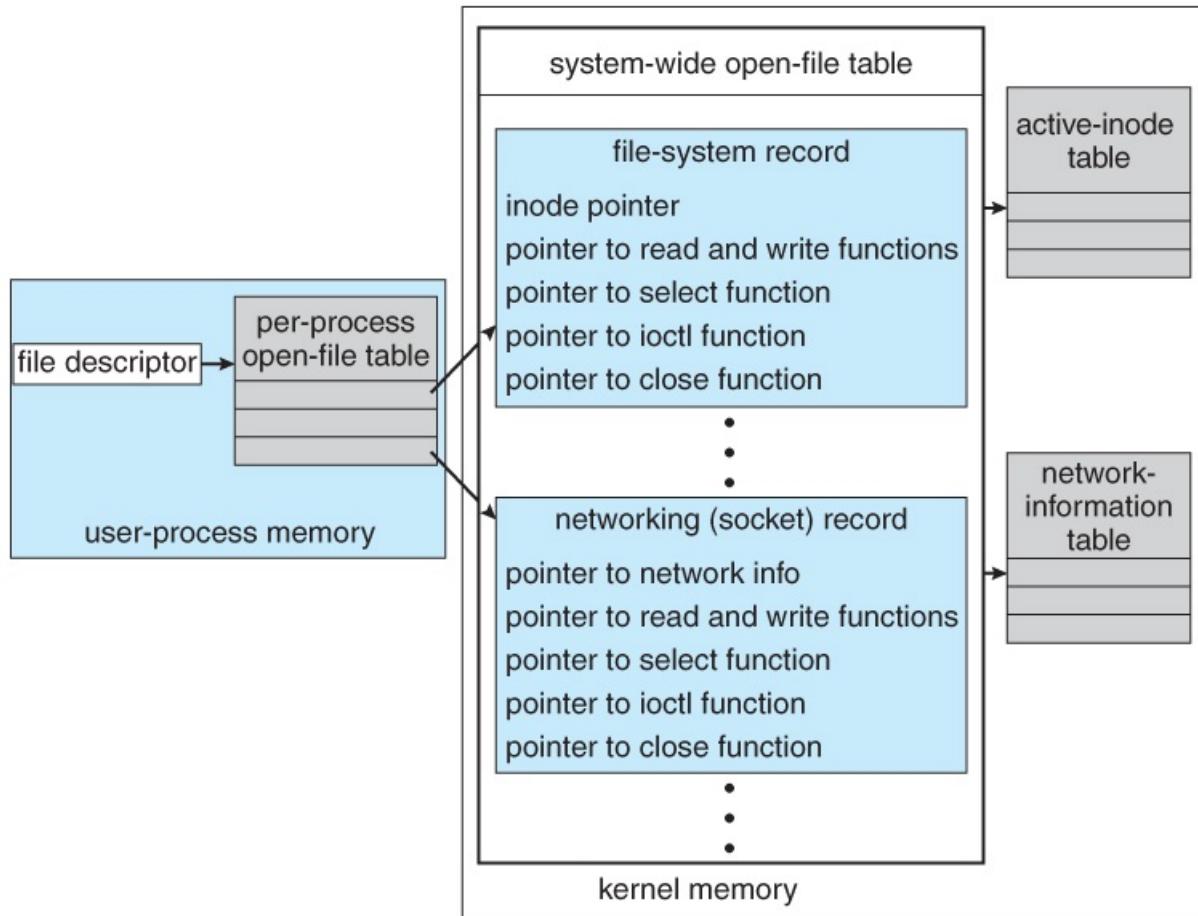
Figure 13.12 - UNIX I/O kernel structure.

13.4.6 Kernel I/O Subsystem Summary

13.5 Transforming I/O Requests to Hardware Operations

- Users request data using file names, which must ultimately be mapped to specific blocks of data from a specific device managed by a specific device driver.
- DOS uses the colon separator to specify a particular device (e.g. C:, LPT:, etc.)
- UNIX uses a ***mount table*** to map filename prefixes (e.g. /usr) to specific mounted devices. Where multiple entries in the mount table match different prefixes of the filename the one that matches the longest prefix is chosen. (e.g. /usr/home instead of /usr where both exist in the mount table and both match the desired file.)
- UNIX uses special ***device files,*** usually located in /dev, to represent and access physical devices directly.
  - Each device file has a major and minor number associated with it, stored and displayed where the file size would normally go.
  - The major number is an index into a table of device drivers, and indicates which device driver handles this device. (E.g. the disk drive handler.)
  - The minor number is a parameter passed to the device driver, and indicates which specific device is to be accessed, out of the many which may be handled by a particular device driver. (e.g. a particular disk drive or partition.)
- A series of lookup tables and mappings makes the access of different devices flexible, and somewhat transparent to users.
- Figure 13.13 illustrates the steps taken to process a (blocking) read request:
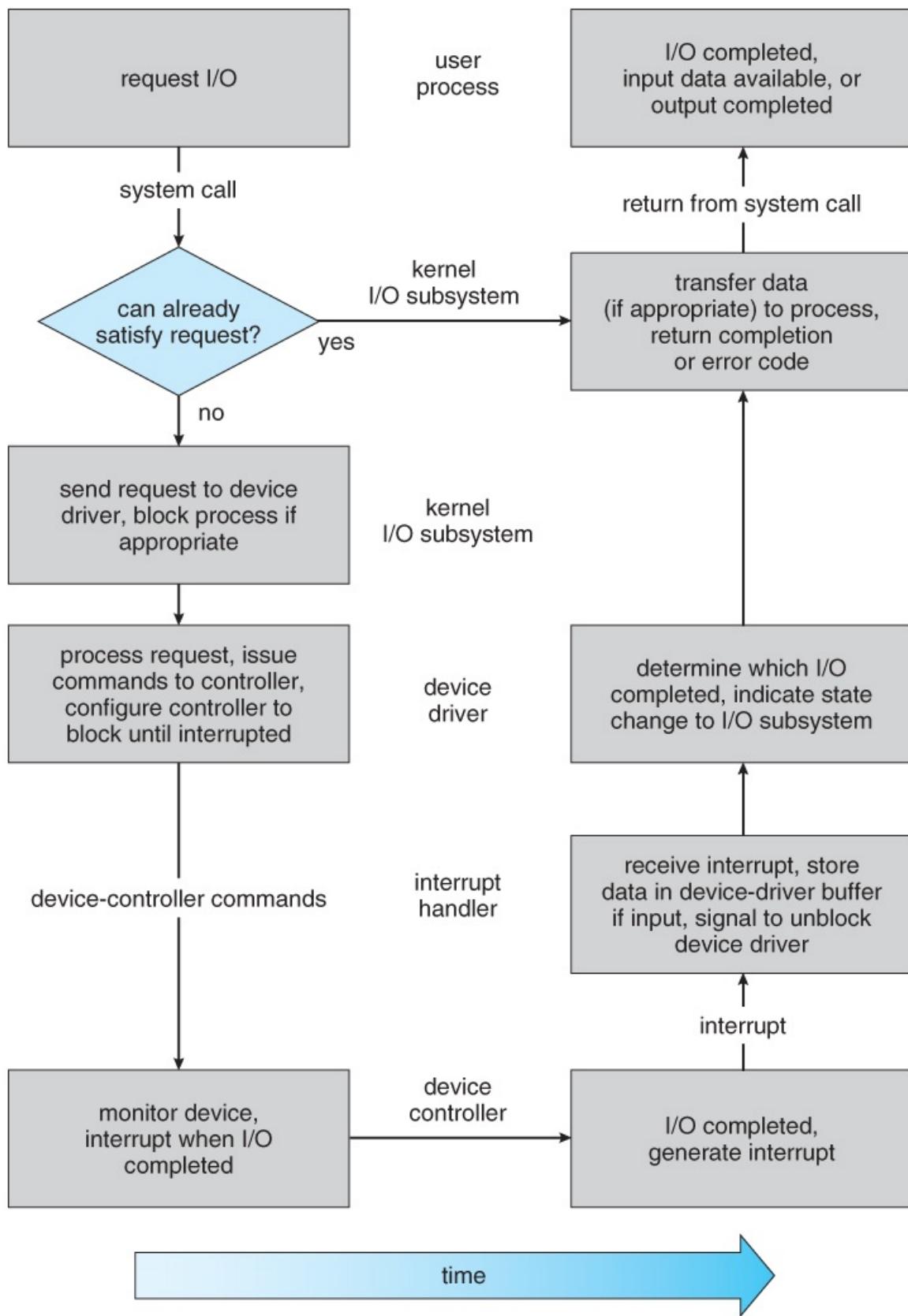
Figure 13.13 - The life cycle of an I/O request.

13.6 STREAMS (Optional)

- The **streams** mechanism in UNIX provides a bi-directional pipeline between a user process and a device driver, onto which additional modules can be added.
- The user process interacts with the **stream head.**
- The device driver interacts with the **device end.**
- Zero or more **stream modules** can be pushed onto the stream, using ioctl( ). These modules may filter and/or modify the data as it passes through the stream.
- Each module has a **read queue** and a **write queue.**
- **Flow control** can be optionally supported, in which case each module will buffer data until the adjacent module is

ready to receive it. Without flow control, data is passed along as soon as it is ready.

- User processes communicate with the stream head using either read( ) and write( ) (or putmsg( ) and getmsg( ) for message passing.)
- Streams I/O is asynchronous (non-blocking), except for the interface between the user process and the stream head.
- The device driver **must** respond to interrupts from its device - If the adjacent module is not prepared to accept data and the device driver's buffers are all full, then data is typically dropped.
- Streams are widely used in UNIX, and are the preferred approach for device drivers. For example, UNIX implements sockets using streams.
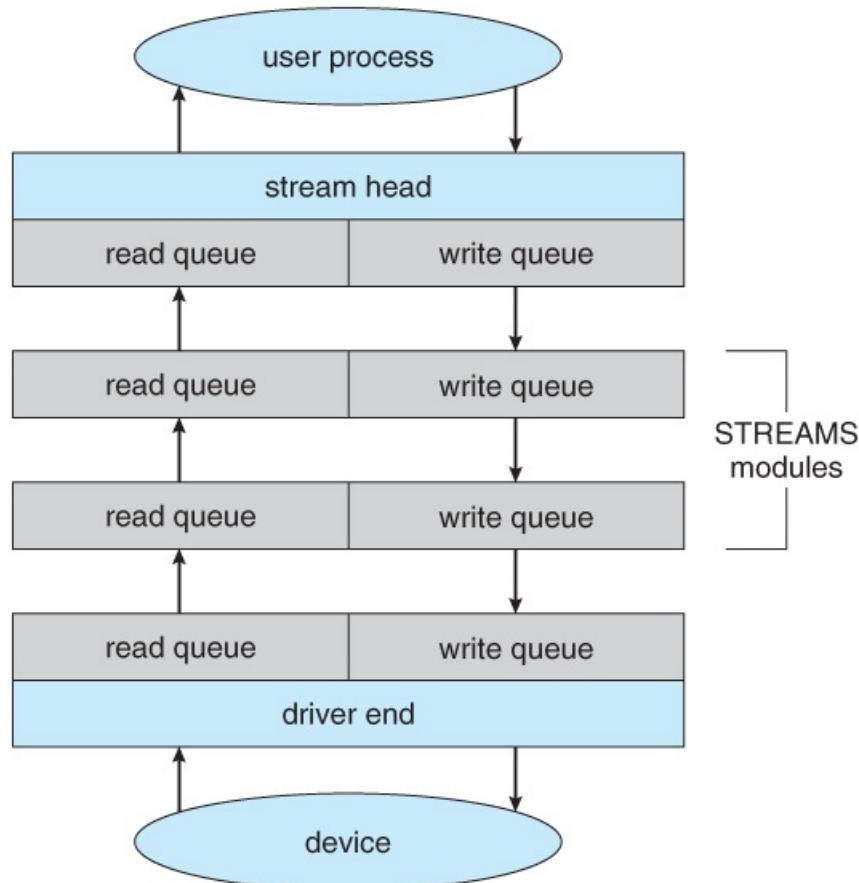
Figure 13.14 - The SREAMS structure.

13.7 Performance (Optional)

- The I/O system is a major factor in overall system performance, and can place heavy loads on other major components of the system (interrupt handling, process switching, memory access, bus contention, and CPU load for device drivers just to name a few.)
- Interrupt handling can be relatively expensive (slow), which causes programmed I/O to be faster than interrupt-driven I/O when the time spent busy waiting is not excessive.
- Network traffic can also put a heavy load on the system. Consider for example the sequence of events that occur when a single character is typed in a telnet session, as shown in figure 13.15. (And the fact that a similar set of events must happen in reverse to echo back the character that was typed.) Sun uses in-kernel threads for the telnet daemon, increasing the supportable number of simultaneous telnet sessions from the hundreds to the thousands.
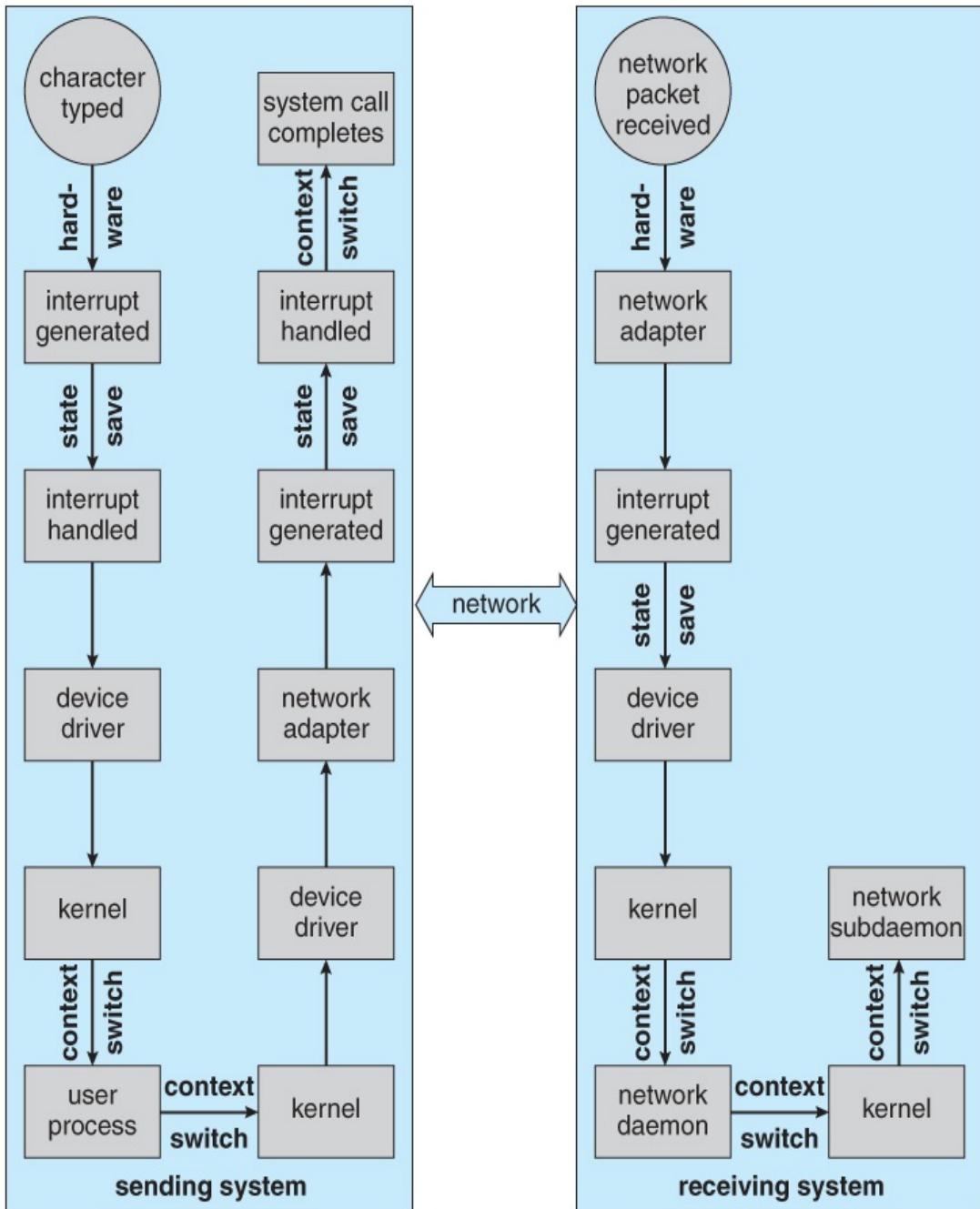
Figure 13.15 - Intercomputer communications.

- Other systems use **_front-end processors_** to off-load some of the work of I/O processing from the CPU. For example a **_terminal concentrator_** can multiplex with hundreds of terminals on a single port on a large computer.
- Several principles can be employed to increase the overall efficiency of I/O processing:
    1. Reduce the number of context switches.
    2. Reduce the number of times data must be copied.
    3. Reduce interrupt frequency, using large transfers, buffering, and polling where appropriate.
    4. Increase concurrency using DMA.
    5. Move processing primitives into hardware, allowing their operation to be concurrent with CPU and bus operations.
    6. Balance CPU, memory, bus, and I/O operations, so a bottleneck in one does not idle all the others.
- The development of new I/O algorithms often follows a progression from application level code to on-board hardware implementation, as shown in Figure 13.16. Lower-level implementations are faster and more efficient, but higher-level ones are more flexible and easier to modify. Hardware-level functionality may also be harder for higher-level authorities (e.g. the kernel) to control.
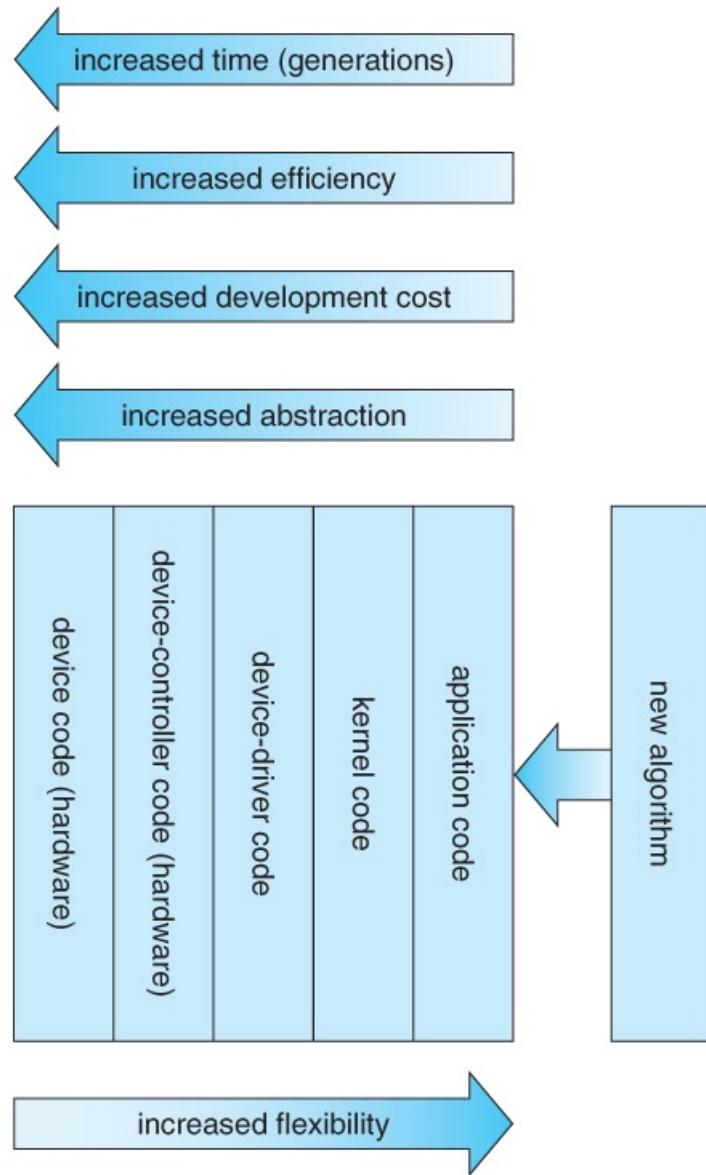
Figure 13.16 - Device functionality progression.

13.8 Summary

Protection

References:

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 14

14.1 Goals of Protection

- Obviously to prevent malicious misuse of the system by users or programs. See chapter 15 for a more thorough coverage of this goal.
- To ensure that each shared resource is used only in accordance with system *policies,* which may be set either by system designers or by system administrators.
- To ensure that errant programs cause the minimal amount of damage possible.
- Note that protection systems only provide the *mechanisms* for enforcing policies and ensuring reliable systems. It is up to administrators and users to implement those mechanisms effectively.

14.2 Principles of Protection

- The ***principle of least privilege*** dictates that programs, users, and systems be given just enough privileges to perform their tasks.
- This ensures that failures do the least amount of harm and allow the least of harm to be done.
- For example, if a program needs special privileges to perform a task, it is better to make it a SGID program with group ownership of "network" or "backup" or some other pseudo group, rather than SUID with root ownership. This limits the amount of damage that can occur if something goes wrong.
- Typically each user is given their own account, and has only enough privilege to modify their own files.
- The root account should not be used for normal day to day activities - The System Administrator should also have an ordinary account, and reserve use of the root account for only those tasks which need the root privileges

14.3 Domain of Protection

- A computer can be viewed as a collection of *processes* and *objects* (both HW & SW).
- The ***need to know principle*** states that a process should only have access to those objects it needs to accomplish its task, and furthermore only in the modes for which it needs access and only during the time frame when it needs access.
- The modes available for a particular object may depend upon its type.

14.3.1 Domain Structure

- A ***protection domain*** specifies the resources that a process may access.
- Each domain defines a set of objects and the types of operations that may be invoked on each object.
- An ***access right*** is the ability to execute an operation on an object.
- A domain is defined as a set of < object, { access right set } > pairs, as shown below. Note that some domains may be disjoint while others overlap.
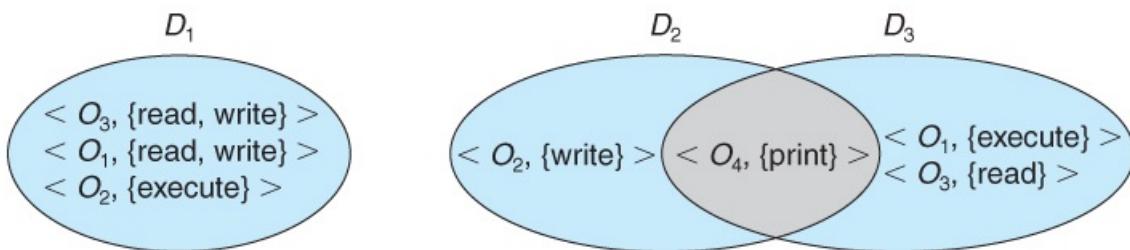
Figure 14.1 - System with three protection domains.

- The association between a process and a domain may be *static* or *dynamic.*
  - If the association is static, then the need-to-know principle requires a way of changing the contents of the domain dynamically.
  - If the association is dynamic, then there needs to be a mechanism for ***domain switching.***
- Domains may be realized in different fashions - as users, or as processes, or as procedures. E.g. if each user corresponds to a domain, then that domain defines the access of that user, and changing domains involves changing user ID.

14.3.2 An Example: UNIX

- UNIX associates domains with users.
- Certain programs operate with the SUID bit set, which effectively changes the user ID, and therefore the access domain, while the program is running. (and similarly for the SGID bit.) Unfortunately this has some potential for abuse.
- An alternative used on some systems is to place privileged programs in special directories, so that they attain the identity of the directory owner when they run. This prevents crackers from placing SUID programs in random directories around the system.
- Yet another alternative is to not allow the changing of ID at all. Instead, special privileged daemons are launched at boot time, and user processes send messages to these daemons when they need special tasks performed.

14.3.3 An Example: MULTICS

- The MULTICS system uses a complex system of rings, each corresponding to a different protection domain, as shown below:
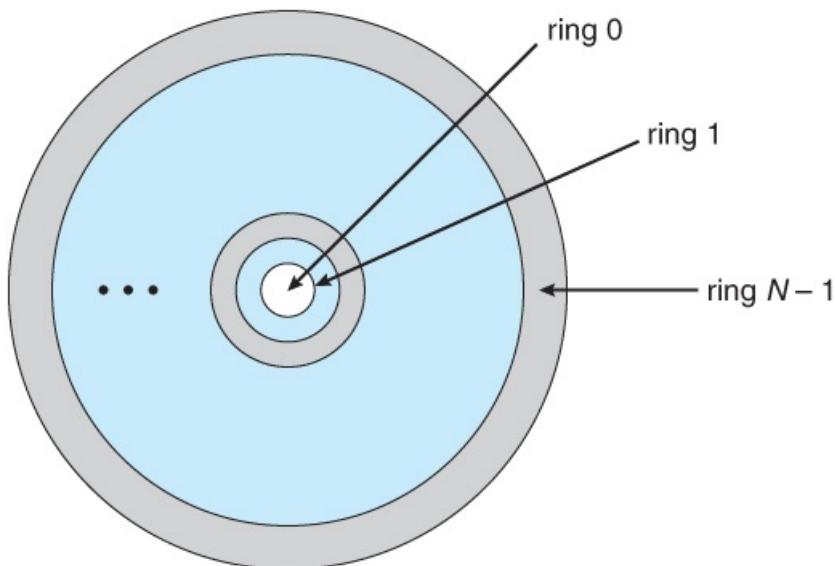

Figure 14.2 - MULTICS ring structure.

- Rings are numbered from 0 to 7, with outer rings having a subset of the privileges of the inner rings.
- Each file is a memory segment, and each segment description includes an entry that indicates the ring number associated with that segment, as well as read, write, and execute privileges.
- Each process runs in a ring, according to the *current-ring-number,* a counter associated with each process.
- A process operating in one ring can only access segments associated with higher (farther out) rings, and then only according to the access bits. Processes cannot access segments associated with lower rings.
- Domain switching is achieved by a process in one ring calling upon a process operating in a lower ring, which is controlled by several factors stored with each segment descriptor:
  - An **access bracket**, defined by integers $b1 <= b2$.
  - A *limit* $b3 > b2$
  - A **list of gates,** identifying the entry points at which the segments may be called.
- If a process operating in ring i calls a segment whose bracket is such that $b1 <= i <= b2$, then the call succeeds and the process remains in ring i.
- Otherwise a trap to the OS occurs, and is handled as follows:
  - If $i < b1$, then the call is allowed, because we are transferring to a procedure with fewer privileges. However if any of the parameters being passed are of segments below b1, then they must be copied to an area accessible by the called procedure.
  - If $i > b2$, then the call is allowed only if $i <= b3$ and the call is directed to one of the entries on the list of gates.
- Overall this approach is more complex and less efficient than other protection schemes.

14.4 Access Matrix

- The model of protection that we have been discussing can be viewed as an **access matrix,** in which columns represent different system resources and rows represent different protection domains. Entries within the matrix indicate

what access that domain has to that resource.

object \ domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figure 14.3 - Access matrix.

- Domain switching can be easily supported under this model, simply by providing "switch" access to other domains:

object \ domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Figure 14.4 - Access matrix of Figure 14.3 with domains as objects.

- The ability to **copy** rights is denoted by an asterisk, indicating that processes in that domain have the right to copy that access within the same column, i.e. for the same object. There are two important variations:
    - If the asterisk is removed from the original access right, then the right is **transferred,** rather than being copied. This may be termed a **transfer** right as opposed to a **copy** right.
    - If only the right and not the asterisk is copied, then the access right is added to the new domain, but it may not be propagated further. That is the new domain does not also receive the right to copy the access. This may be termed a **limited copy** right, as shown in Figure 14.5 below:

object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Figure 14.5 - Access matrix with *copy* rights.

- The ***owner*** right adds the privilege of adding new rights or removing existing ones:

object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Figure 14.6 - Access matrix with *owner* rights.

- Copy and owner rights only allow the modification of rights within a column. The addition of ***control rights***, which only apply to domain objects, allow a process operating in one domain to affect the rights available in other domains. For example in the table below, a process operating in domain D2 has the right to control any of the rights in domain D4.

object domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Figure 14.7 - Modified access matrix of Figure 14.4

14.5 Implementation of Access Matrix

14.5.1 Global Table

- The simplest approach is one big global table with < domain, object, rights > entries.
- Unfortunately this table is very large (even if sparse) and so cannot be kept in memory (without invoking virtual memory techniques.)
- There is also no good way to specify groupings - If everyone has access to some resource, then it still needs a separate entry for every domain.

14.5.2 Access Lists for Objects

- Each column of the table can be kept as a list of the access rights for that particular object, discarding blank entries.
- For efficiency a separate list of default access rights can also be kept, and checked first.

14.5.3 Capability Lists for Domains

- In a similar fashion, each row of the table can be kept as a list of the capabilities of that domain.
- Capability lists are associated with each domain, but not directly accessible by the domain or any user process.
- Capability lists are themselves protected resources, distinguished from other data in one of two ways:
  - A *tag,* possibly hardware implemented, distinguishing this special type of data. (other types may be floats, pointers, booleans, etc.)
  - The address space for a program may be split into multiple segments, at least one of which is inaccessible by the program itself, and used by the operating system for maintaining the process's access right capability list.

14.5.4 A Lock-Key Mechanism
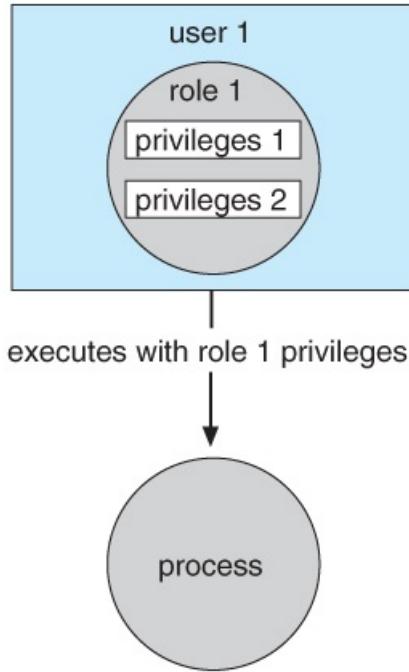
- Each resource has a list of unique bit patterns, termed locks.
- Each domain has its own list of unique bit patterns, termed keys.
- Access is granted if one of the domain's keys fits one of the resource's locks.
- Again, a process is not allowed to modify its own keys.

14.5.5 Comparison

- Each of the methods here has certain advantages or disadvantages, depending on the particular situation and task at hand.
- Many systems employ some combination of the listed methods.

14.6 Access Control

- *Role-Based Access Control, RBAC,* assigns privileges to users, programs, or roles as appropriate, where "privileges" refer to the right to call certain system calls, or to use certain parameters with those calls.
- RBAC supports the principle of least privilege, and reduces the susceptibility to abuse as opposed to SUID or SGID programs.

Figure 14.8 - Role-based access control in Solaris 10.

14.7 Revocation of Access Rights

- The need to revoke access rights dynamically raises several questions:
  - Immediate versus delayed - If delayed, can we determine when the revocation will take place?
  - Selective versus general - Does revocation of an access right to an object affect *all* users who have that right, or only some users?
  - Partial versus total - Can a subset of rights for an object be revoked, or are all rights revoked at once?
  - Temporary versus permanent - If rights are revoked, is there a mechanism for processes to re-acquire some or all of the revoked rights?
- With an access list scheme revocation is easy, immediate, and can be selective, general, partial, total, temporary, or permanent, as desired.
- With capabilities lists the problem is more complicated, because access rights are distributed throughout the system. A few schemes that have been developed include:
  - Reacquisition - Capabilities are periodically revoked from each domain, which must then re-acquire them.
  - Back-pointers - A list of pointers is maintained from each object to each capability which is held for that object.
  - Indirection - Capabilities point to an entry in a global table rather than to the object. Access rights can be revoked by changing or invalidating the table entry, which may affect multiple processes, which must then re-acquire access rights to continue.
  - Keys - A unique bit pattern is associated with each capability when created, which can be neither inspected nor modified by the process.
    - A master key is associated with each object.
    - When a capability is created, its key is set to the object's master key.
    - As long as the capability's key matches the object's key, then the capabilities remain valid.
    - The object master key can be changed with the set-key command, thereby invalidating all current capabilities.
    - More flexibility can be added to this scheme by implementing a *list* of keys for each object, possibly in a global table.

14.8 Capability-Based Systems (Optional)

14.8.1 An Example: Hydra

- Hydra is a capability-based system that includes both system-defined *rights* and user-defined rights. The interpretation of user-defined rights is up to the specific user programs, but the OS provides support for protecting access to those rights, whatever they may be
- Operations on objects are defined procedurally, and those procedures are themselves protected objects, accessed indirectly through capabilities.
- The names of user-defined procedures must be identified to the protection system if it is to deal with user-defined rights.

- When an object is created, the names of operations defined on that object become **auxiliary rights,** described in a capability for an **instance** of the type. For a process to act on an object, the capabilities it holds for that object must contain the name of the operation being invoked. This allows access to be controlled on an instance-by-instance and process-by-process basis.
- Hydra also allows **rights amplification,** in which a process is deemed to be **trustworthy,** and thereby allowed to act on any object corresponding to its parameters.
- Programmers can make direct use of the Hydra protection system, using suitable libraries which are documented in appropriate reference manuals.

14.8.2 An Example: Cambridge CAP System

- The CAP system has two kinds of capabilities:
  - **Data capability,** used to provide read, write, and execute access to objects. These capabilities are interpreted by microcode in the CAP machine.
  - **Software capability,** is protected but not interpreted by the CAP microcode.
    - Software capabilities are interpreted by protected (privileged) procedures, possibly written by application programmers.
    - When a process executes a protected procedure, it temporarily gains the ability to read or write the contents of a software capability.
    - This leaves the interpretation of the software capabilities up to the individual subsystems, and limits the potential damage that could be caused by a faulty privileged procedure.
    - Note, however, that protected procedures only get access to software capabilities for the subsystem of which they are a part. Checks are made when passing software capabilities to protected procedures that they are of the correct type.
    - Unfortunately the CAP system does not provide libraries, making it harder for an individual programmer to use than the Hydra system.

14.9 Language-Based Protection (Optional)

- As systems have developed, protection systems have become more powerful, and also more specific and specialized.
- To refine protection even further requires putting protection capabilities into the hands of individual programmers, so that protection policies can be implemented on the application level, i.e. to protect resources in ways that are known to the specific applications but not to the more general operating system.

14.9.1 Compiler-Based Enforcement

- In a compiler-based approach to protection enforcement, programmers directly specify the protection needed for different resources at the time the resources are declared.
- This approach has several advantages:
  1. Protection needs are simply declared, as opposed to a complex series of procedure calls.
  2. Protection requirements can be stated independently of the support provided by a particular OS.
  3. The means of enforcement need not be provided directly by the developer.
  4. Declarative notation is natural, because access privileges are closely related to the concept of data types.
- Regardless of the means of implementation, compiler-based protection relies upon the underlying protection mechanisms provided by the underlying OS, such as the Cambridge CAP or Hydra systems.
- Even if the underlying OS does not provide advanced protection mechanisms, the compiler can still offer some protection, such as treating memory accesses differently in code versus data segments. (E.g. code segments cant be modified, data segments can't be executed.)
- There are several areas in which compiler-based protection can be compared to kernel-enforced protection:
  - **Security.** Security provided by the kernel offers better protection than that provided by a compiler. The security of the compiler-based enforcement is dependent upon the integrity of the compiler itself, as well as requiring that files not be modified after they are compiled. The kernel is in a better position to protect itself from modification, as well as protecting access to specific files. Where hardware support of individual memory accesses is available, the protection is stronger still.
  - **Flexibility.** A kernel-based protection system is not as flexible to provide the specific protection needed by an individual programmer, though it may provide support which the programmer may make use of. Compilers are more easily changed and updated when necessary to change the protection services offered or their implementation.
  - **Efficiency.** The most efficient protection mechanism is one supported by hardware and microcode. Insofar as software based protection is concerned, compiler-based systems have the advantage that

many checks can be made off-line, at compile time, rather that during execution.

- The concept of incorporating protection mechanisms into programming languages is in its infancy, and still remains to be fully developed. However the general goal is to provide mechanisms for three functions:
    1. Distributing capabilities safely and efficiently among customer processes. In particular a user process should only be able to access resources for which it was issued capabilities.
    2. Specifying the *type* of operations a process may execute on a resource, such as reading or writing.
    3. Specifying the *order* in which operations are performed on the resource, such as opening before reading.

14.9.2 Protection in Java

- Java was designed from the very beginning to operate in a distributed environment, where code would be executed from a variety of trusted and untrusted sources. As a result the Java Virtual Machine, JVM incorporates many protection mechanisms
- When a Java program runs, it load up classes dynamically, in response to requests to instantiates objects of particular types. These classes may come from a variety of different sources, some trusted and some not, which requires that the protection mechanism be implemented at the resolution of individual classes, something not supported by the basic operating system.
- As each class is loaded, it is placed into a separate protection domain. The capabilities of each domain depend upon whether the source URL is trusted or not, the presence or absence of any digital signatures on the class (Chapter 15), and a configurable policy file indicating which servers a particular user trusts, etc.
- When a request is made to access a restricted resource in Java, (e.g. open a local file), some process on the current *call stack* must specifically assert a privilege to perform the operation. In essence this method *assumes responsibility* for the restricted access. Naturally the method must be part of a class which resides in a protection domain that includes the capability for the requested operation. This approach is termed *stack inspection,* and works like this:
    - When a caller may not be trusted, a method executes an access request within a `doPrivileged( )` block, which is noted on the calling stack.
    - When access to a protected resource is requested, `checkPermissions( )` inspects the call stack to see if a method has asserted the privilege to access the protected resource.
        - If a suitable doPriveleged block is encountered on the stack before a domain in which the privilege is disallowed, then the request is granted.
        - If a domain in which the request is disallowed is encountered first, then the access is denied and a AccessControlException is thrown.
        - If neither is encountered, then the response is implementation dependent.
- In the example below the untrusted applet's call to `get( )` succeeds, because the trusted URL loader asserts the privilege of opening the specific URL lucent.com. However when the applet tries to make a direct call to `open( )` it fails, because it does not have privilege to access any sockets.

protection domain:	untrusted applet	URL loader	networking
socket permission:	none	*.lucent.com:80, connect	any

class:	gui:
	    . . .
	    get(url);
	    open(addr);
	    . . .
		get(URL u):
		    . . .
		    doPrivileged {
		        open('proxy.lucent.com:80');
		    }
		    <request u from proxy>
		    . . .
			open(Addr a):
			    . . .
			    checkPermission
			    (a, connect);
			    connect (a);
			    . . .

Figure 14.9 - Stack inspection.

14.10 Summary

Security

References:

1. Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Seventh Edition ", Chapter 15

Note:

- This is a brief one-chapter introduction to a very large and important topic. Students interested in the topic of Computer Security may want to consider following this course up with CS 487- Building Trustworthy Computer Systems, CS 587 Computer Systems Security, or CS 588 Security and Privacy in Networked and Distributed Systems.

15.1 The Security Problem

- Chapter 14 (Protection) dealt with protecting files and other resources from accidental misuse by cooperating users sharing a system, generally using the computer for normal purposes.
- This chapter (Security) deals with protecting systems from deliberate attacks, either internal or external, from individuals intentionally attempting to steal information, damage information, or otherwise deliberately wreak havoc in some manner.
- Some of the most common types of *violations* include:
  - ○ *Breach of Confidentiality -* Theft of private or confidential information, such as credit-card numbers, trade secrets, patents, secret formulas, manufacturing procedures, medical information, financial information, etc.
  - ○ *Breach of Integrity -* Unauthorized *modification* of data, which may have serious indirect consequences. For example a popular game or other program's source code could be modified to open up security holes on users systems before being released to the public.
  - ○ *Breach of Availability -* Unauthorized *destruction* of data, often just for the "fun" of causing havoc and for bragging rites. Vandalism of web sites is a common form of this violation.
  - ○ *Theft of Service -* Unauthorized use of resources, such as theft of CPU cycles, installation of daemons running an unauthorized file server, or tapping into the target's telephone or networking services.
  - ○ *Denial of Service, DOS -* Preventing legitimate users from using the system, often by overloading and overwhelming the system with an excess of requests for service.
- One common attack is *masquerading,* in which the attacker pretends to be a trusted third party. A variation of this is the *man-in-the-middle,* in which the attacker masquerades as both ends of the conversation to two targets.
- A *replay attack* involves repeating a valid transmission. Sometimes this can be the entire attack, (such as repeating a request for a money transfer), or other times the content of the original message is replaced with malicious content.
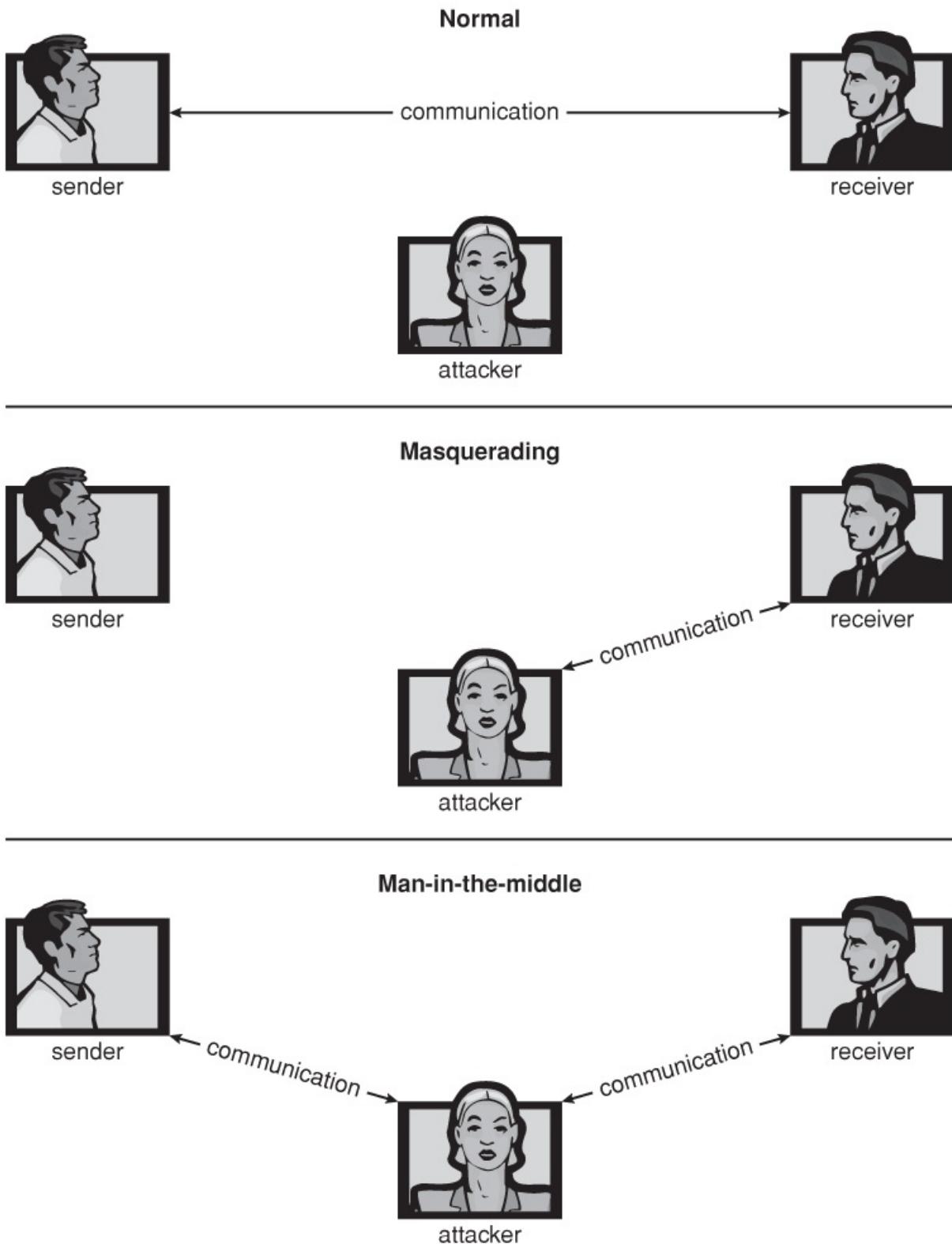
Figure 15.1 - Standard security attacks.

- There are four levels at which a system must be protected:
    1. **Physical -** The easiest way to steal data is to pocket the backup tapes. Also, access to the root console will often give the user special privileges, such as rebooting the system as root from removable media. Even general access to terminals in a computer room offers some opportunities for an attacker, although today's modern high-speed networking environment provides more and more opportunities for remote attacks.
    2. **Human -** There is some concern that the humans who are allowed access to a system be trustworthy, and that they cannot be coerced into breaching security. However more and more attacks today are made via *social engineering,* which basically means fooling trustworthy people into accidentally breaching security.
        - **Phishing** involves sending an innocent-looking e-mail or web site designed to fool people into revealing confidential information. E.g. spam e-mails pretending to be from e-Bay, PayPal, or any of a number of banks or credit-card companies.
        - **Dumpster Diving** involves searching the trash or other locations for passwords that are written down.

(Note: Passwords that are too hard to remember, or which must be changed frequently are more likely to be written down somewhere close to the user's station.)

- **Password Cracking** involves divining users passwords, either by watching them type in their passwords, knowing something about them like their pet's names, or simply trying all words in common dictionaries. (Note: "Good" passwords should involve a minimum number of characters, include non-alphabetical characters, and not appear in any dictionary (in any language), and should be changed frequently. Note also that it is proper etiquette to look away from the keyboard while someone else is entering their password.)

3. **Operating System -** The OS must protect itself from security breaches, such as runaway processes (denial of service), memory-access violations, stack overflow violations, the launching of programs with excessive privileges, and many others.

4. **Network -** As network communications become ever more important and pervasive in modern computing environments, it becomes ever more important to protect this area of the system. (Both protecting the network itself from attack, and protecting the local system from attacks coming in through the network.) This is a growing area of concern as wireless communications and portable devices become more and more prevalent.

15.2 Program Threats

- There are many common threats to modern systems. Only a few are discussed here.

15.2.1 Trojan Horse

- A *Trojan Horse* is a program that secretly performs some maliciousness in addition to its visible actions.
- Some Trojan horses are deliberately written as such, and others are the result of legitimate programs that have become infected with *viruses,* (see below.)
- One dangerous opening for Trojan horses is long search paths, and in particular paths which include the current directory (".") as part of the path. If a dangerous program having the same name as a legitimate program (or a common mis-spelling, such as "sl" instead of "ls") is placed anywhere on the path, then an unsuspecting user may be fooled into running the wrong program by mistake.
- Another classic Trojan Horse is a login emulator, which records a users account name and password, issues a "password incorrect" message, and then logs off the system. The user then tries again (with a proper login prompt), logs in successfully, and doesn't realize that their information has been stolen.
- **(Special Note to UIC students: Beware that someone has registered the domain name of uic.EU (without the "D"), and is running an ssh server which will accept requests to any machine in the domain, and gladly accept your login and password information, without, of course, actually logging you in. Access to this site is blocked from campus, but you are on your own off campus.)**
- Two solutions to Trojan Horses are to have the system print usage statistics on logouts, and to require the typing of non-trappable key sequences such as Control-Alt-Delete in order to log in. (This is why modern Windows systems require the Control-Alt-Delete sequence to commence logging in, which cannot be emulated or caught by ordinary programs. I.e. that key sequence always transfers control over to the operating system.)
- *Spyware* is a version of a Trojan Horse that is often included in "free" software downloaded off the Internet. Spyware programs generate pop-up browser windows, and may also accumulate information about the user and deliver it to some central site. (This is an example of *covert channels,* in which surreptitious communications occur.) Another common task of spyware is to send out spam e-mail messages, which then purportedly come from the infected user.

15.2.2 Trap Door

- A *Trap Door* is when a designer or a programmer (or hacker) deliberately inserts a security hole that they can use later to access the system.
- Because of the possibility of trap doors, once a system has been in an untrustworthy state, that system can never be trusted again. Even the backup tapes may contain a copy of some cleverly hidden back door.
- A clever trap door could be inserted into a compiler, so that any programs compiled with that compiler would contain a security hole. This is especially dangerous, because inspection of the code being compiled would not reveal any problems.

15.2.3 Logic Bomb

- A *Logic Bomb* is code that is not designed to cause havoc all the time, but only when a certain set of circumstances occurs, such as when a particular date or time is reached or some other noticeable event.
- A classic example is the *Dead-Man Switch*, which is designed to check whether a certain person (e.g. the

author) is logging in every day, and if they don't log in for a long time (presumably because they've been fired), then the logic bomb goes off and either opens up security holes or causes other problems.

15.2.4 Stack and Buffer Overflow

- This is a classic method of attack, which exploits bugs in system code that allows buffers to overflow. Consider what happens in the following code, for example, if argv[1] exceeds 256 characters:
    - The strcpy command will overflow the buffer, overwriting adjacent areas of memory.
    - (The problem could be avoided using str*n*cpy, with a limit of 255 characters copied plus room for the null byte.)
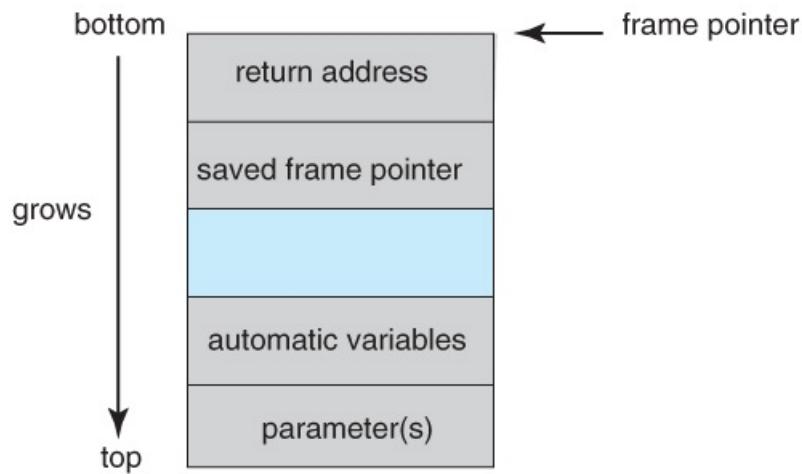
```
#include
#define BUFFER_SIZE 256

int main( int argc, char * argv[ ] )
{
    char buffer[ BUFFER_SIZE ];

    if( argc < 2 )
        return -1;
    else {
        strcpy( buffer, argv[ 1 ] );
        return 0;
    }
}
```

Figure 15.2 - C program with buffer-overflow condition.
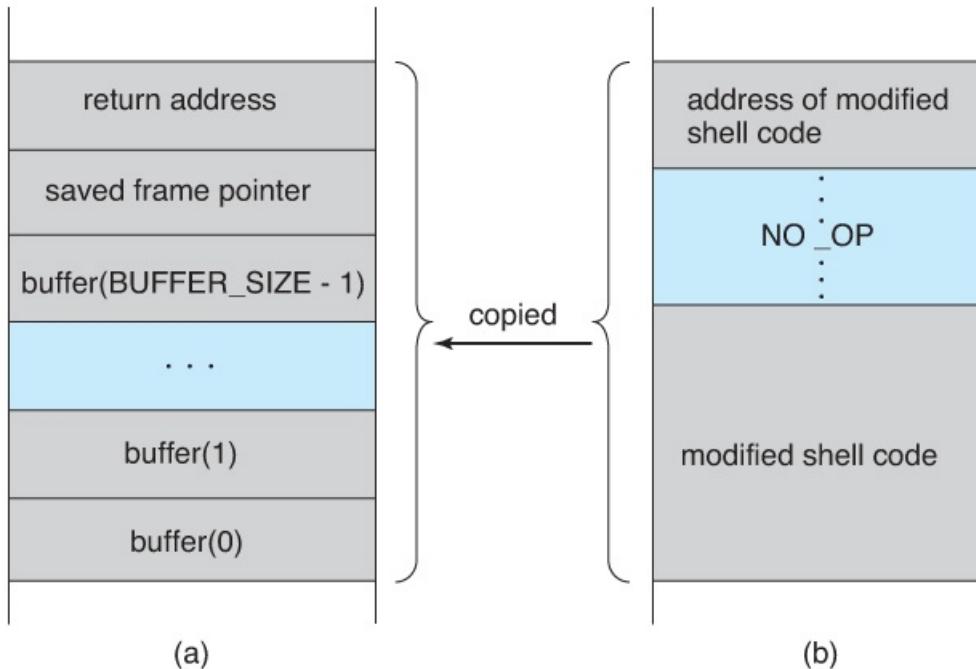
- So how does overflowing the buffer cause a security breach? Well the first step is to understand the structure of the stack in memory:
    - The "bottom" of the stack is actually at a high memory address, and the stack grows towards lower addresses.
    - However the address of an array is the lowest address of the array, and higher array elements extend to higher addresses. (I.e. an array "grows" towards the bottom of the stack.
    - In particular, writing past the top of an array, as occurs when a buffer overflows with too much input data, can eventually overwrite the return address, effectively changing where the program jumps to when it returns.

Figure 15.3 - The layout for a typical stack frame.

- Now that we know how to change where the program returns to by overflowing the buffer, the second step is to insert some nefarious code, and then get the program to jump to our inserted code.
- Our only opportunity to enter code is via the input into the buffer, which means there isn't room for very much. One of the simplest and most obvious approaches is to insert the code for "exec(/bin/sh)". To do this requires compiling a program that contains this instruction, and then using an assembler or debugging tool to extract the minimum extent that includes the necessary instructions.
- The bad code is then padded with as many extra bytes as are needed to overflow the buffer to the correct extent, and the address of the buffer inserted into the return address location. (Note, however, that neither the bad code or the padding can contain null bytes, which would terminate the strcpy.)

- The resulting block of information is provided as "input", copied into the buffer by the original program, and then the return statement causes control to jump to the location of the buffer and start executing the code to launch a shell.
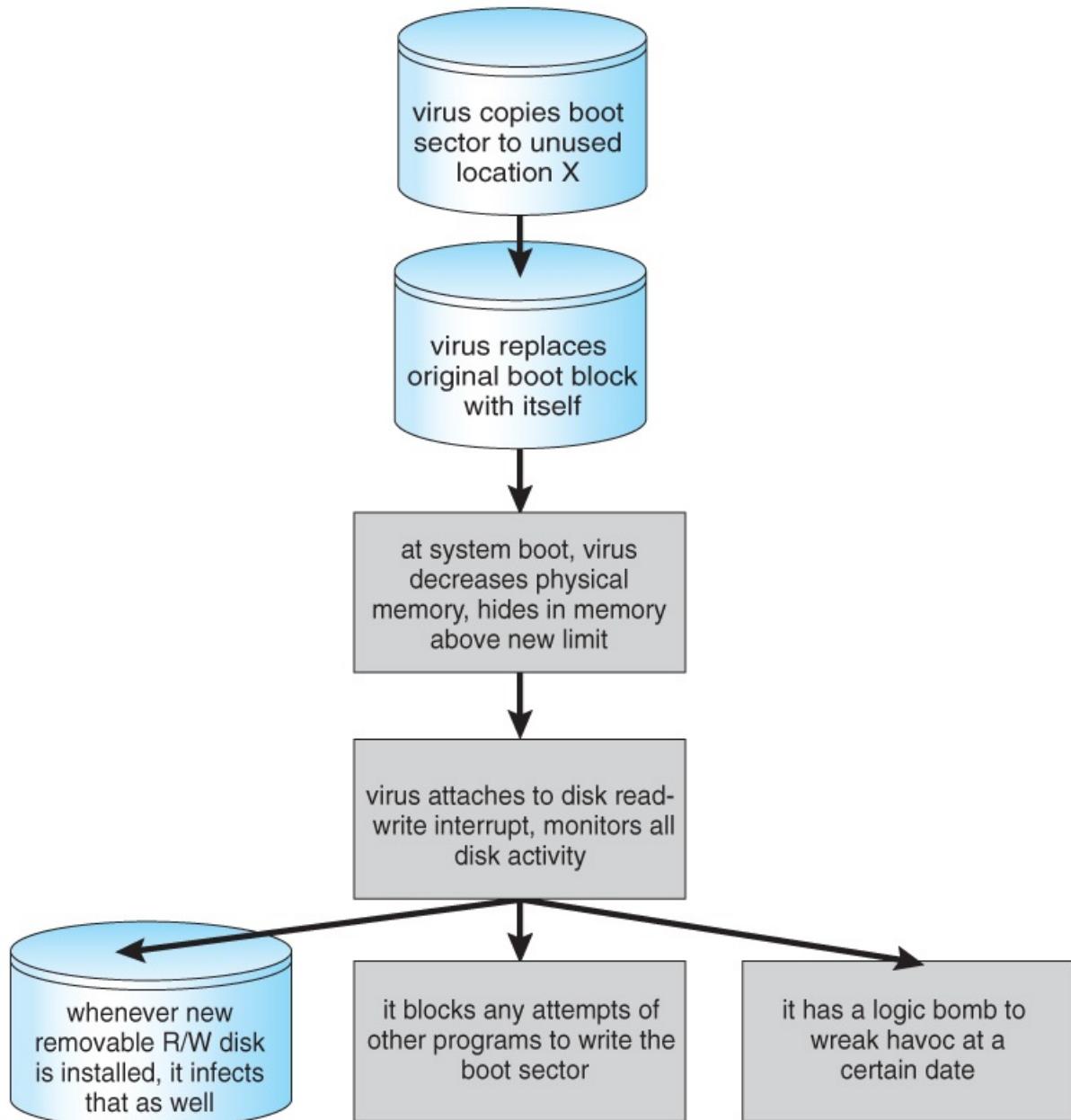
Figure 15.4 - Hypothetical stack frame for Figure 15.2, (a) before and (b) after.

- Unfortunately famous hacks such as the buffer overflow attack are well published and well known, and it doesn't take a lot of skill to follow the instructions and start attacking lots of systems until the law of averages eventually works out. (**Script Kiddies** are those hackers with only rudimentary skills of their own but the ability to copy the efforts of others.)
- Fortunately modern hardware now includes a bit in the page tables to mark certain pages as non-executable. In this case the buffer-overflow attack would work up to a point, but as soon as it "returns" to an address in the data space and tries executing statements there, an exception would be thrown crashing the program.
- (More details about stack-overflow attacks are available on-line from http://www.insecure.org/stf/smashstack.txt)

15.2.5 Viruses

- A virus is a fragment of code embedded in an otherwise legitimate program, designed to replicate itself (by infecting other programs), and (eventually) wreaking havoc.
- Viruses are more likely to infect PCs than UNIX or other multi-user systems, because programs in the latter systems have limited authority to modify other programs or to access critical system structures (such as the boot block.)
- Viruses are delivered to systems in a **virus dropper,** usually some form of a Trojan Horse, and usually via e-mail or unsafe downloads.
- Viruses take many forms (see below.) Figure 15.5 shows typical operation of a boot sector virus:

Figure 15.5 - A boot-sector computer virus.

- Some of the forms of viruses include:
    - **File -** A file virus attaches itself to an executable file, causing it to run the virus code first and then jump to the start of the original program. These viruses are termed ***parasitic,*** because they do not leave any new files on the system, and the original program is still fully functional.
    - **Boot -** A boot virus occupies the boot sector, and runs before the OS is loaded. These are also known as ***memory viruses***, because in operation they reside in memory, and do not appear in the file system.
    - **Macro -** These viruses exist as a macro (script) that are run automatically by certain macro-capable programs such as MS Word or Excel. These viruses can exist in word processing documents or spreadsheet files.
    - **Source code** viruses look for source code and infect it in order to spread.
    - **Polymorphic** viruses change every time they spread - Not their underlying functionality, but just their ***signature,*** by which virus checkers recognize them.
    - **Encrypted** viruses travel in encrypted form to escape detection. In practice they are self-decrypting, which then allows them to infect other files.
    - **Stealth** viruses try to avoid detection by modifying parts of the system that could be used to detect it. For example the read( ) system call could be modified so that if an infected file is read the infected part gets skipped and the reader would see the original unadulterated file.
    - **Tunneling** viruses attempt to avoid detection by inserting themselves into the interrupt handler chain, or into device drivers.
    - **Multipartite** viruses attack multiple parts of the system, such as files, boot sector, and memory.
    - **Armored** viruses are coded to make them hard for anti-virus researchers to decode and understand.

> In addition many files associated with viruses are hidden, protected, or given innocuous looking names such as "...".

- In 2004 a virus exploited three bugs in Microsoft products to infect hundreds of Windows servers (including many trusted sites) running Microsoft Internet Information Server, which in turn infected any Microsoft Internet Explorer web browser that visited any of the infected server sites. One of the back-door programs it installed was a ***keystroke logger,*** which records users keystrokes, including passwords and other sensitive information.
- There is some debate in the computing community as to whether a ***monoculture,*** in which nearly all systems run the same hardware, operating system, and applications, increases the threat of viruses and the potential for harm caused by them.

15.3 System and Network Threats

- Most of the threats described above are termed ***program threats***, because they attack specific programs or are carried and distributed in programs. The threats in this section attack the operating system or the network itself, or leverage those systems to launch their attacks.

15.3.1 Worms

- A ***worm*** is a process that uses the fork / spawn process to make copies of itself in order to wreak havoc on a system. Worms consume system resources, often blocking out other, legitimate processes. Worms that propagate over networks can be especially problematic, as they can tie up vast amounts of network resources and bring down large-scale systems.
- One of the most well-known worms was launched by Robert Morris, a graduate student at Cornell, in November 1988. Targeting Sun and VAX computers running BSD UNIX version 4, the worm spanned the Internet in a matter of a few hours, and consumed enough resources to bring down many systems.
- This worm consisted of two parts:
  1. A small program called a ***grappling hook,*** which was deposited on the target system through one of three vulnerabilities, and
  2. The main worm program, which was transferred onto the target system and launched by the grappling hook program.
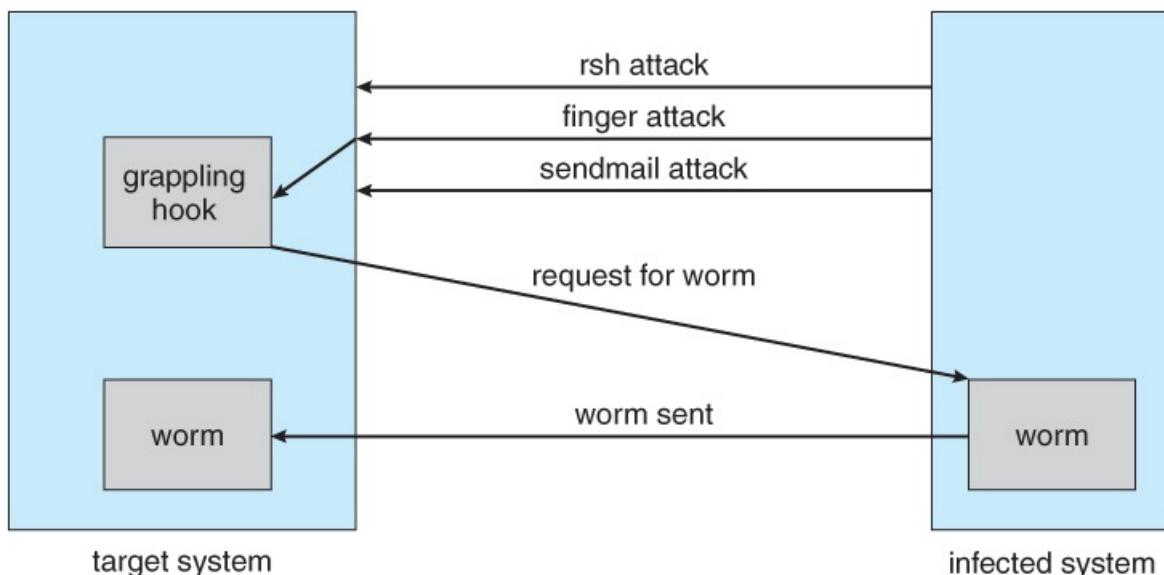
Figure 15.6 - The Morris Internet worm.

- The three vulnerabilities exploited by the Morris Internet worm were as follows:
  1. **rsh (remote shell)** is a utility that was in common use at that time for accessing remote systems without having to provide a password. If a user had an account on two different computers (with the same account name on both systems), then the system could be configured to allow that user to remotely connect from one system to the other without having to provide a password. Many systems were configured so that ***any*** user (except root) on system A could access the same account on system B without providing a password.
  2. **finger** is a utility that allows one to remotely query a user database, to find the true name and other information for a given account name on a given system. For example "finger joeUser@somemachine.edu" would access the finger daemon at somemachine.edu and return information regarding joeUser. Unfortunately the finger daemon (which ran with system privileges)

had the buffer overflow problem, so by sending a special 536-character user name the worm was able to fork a shell on the remote system running with root privileges.

3. **sendmail** is a routine for sending and forwarding mail that also included a debugging option for verifying and testing the system. The debug feature was convenient for administrators, and was often left turned on. The Morris worm exploited the debugger to mail and execute a copy of the grappling hook program on the remote system.

- Once in place, the worm undertook systematic attacks to discover user passwords:
    1. First it would check for accounts for which the account name and the password were the same, such as "guest", "guest".
    2. Then it would try an internal dictionary of 432 favorite password choices. (I'm sure "password", "pass", and blank passwords were all on the list.)
    3. Finally it would try every word in the standard UNIX on-line dictionary to try and break into user accounts.
- Once it had gotten access to one or more user accounts, then it would attempt to use those accounts to rsh to other systems, and continue the process.
- With each new access the worm would check for already running copies of itself, and 6 out of 7 times if it found one it would stop. (The seventh was to prevent the worm from being stopped by fake copies.)
- Fortunately the same rapid network connectivity that allowed the worm to propagate so quickly also quickly led to its demise - Within 24 hours remedies for stopping the worm propagated through the Internet from administrator to administrator, and the worm was quickly shut down.
- There is some debate about whether Mr. Morris's actions were a harmless prank or research project that got out of hand or a deliberate and malicious attack on the Internet. However the court system convicted him, and penalized him heavy fines and court costs.
- There have since been many other worm attacks, including the W32.Sobig.F@mm attack which infected hundreds of thousands of computers and an estimated 1 in 17 e-mails in August 2003. This worm made detection difficult by varying the subject line of the infection-carrying mail message, including "Thank You!", "Your details", and "Re: Approved".

15.3.2 Port Scanning

- **Port Scanning** is technically not an attack, but rather a search for vulnerabilities to attack. The basic idea is to systematically attempt to connect to every known (or common or possible) network port on some remote machine, and to attempt to make contact. Once it is determined that a particular computer is listening to a particular port, then the next step is to determine what daemon is listening, and whether or not it is a version containing a known security flaw that can be exploited.
- Because port scanning is easily detected and traced, it is usually launched from **zombie systems,** i.e. previously hacked systems that are being used without the knowledge or permission of their rightful owner. For this reason it is important to protect "innocuous" systems and accounts as well as those that contain sensitive information or special privileges.
- There are also port scanners available that administrators can use to check their own systems, which report any weaknesses found but which do not exploit the weaknesses or cause any problems. Two such systems are **nmap** (http://www.insecure.org/nmap) and **nessus** (http://www.nessus.org). The former identifies what OS is found, what firewalls are in place, and what services are listening to what ports. The latter also contains a database of known security holes, and identifies any that it finds.

15.3.3 Denial of Service

- **Denial of Service (DOS)** attacks do not attempt to actually access or damage systems, but merely to clog them up so badly that they cannot be used for any useful work. Tight loops that repeatedly request system services are an obvious form of this attack.
- DOS attacks can also involve social engineering, such as the Internet chain letters that say "send this immediately to 10 of your friends, and then go to a certain URL", which clogs up not only the Internet mail system but also the web server to which everyone is directed. (Note: Sending a "reply all" to such a message notifying everyone that it was just a hoax also clogs up the Internet mail service, just as effectively as if you had forwarded the thing.)
- Security systems that lock accounts after a certain number of failed login attempts are subject to DOS attacks which repeatedly attempt logins to all accounts with invalid passwords strictly in order to lock up all accounts.
- Sometimes DOS is not the result of deliberate maliciousness. Consider for example:
    - A web site that sees a huge volume of hits as a result of a successful advertising campaign.
    - CNN.com occasionally gets overwhelmed on big news days, such as Sept 11, 2001.
    - CS students given their first programming assignment involving fork( ) often quickly fill up process

tables or otherwise completely consume system resources. :-)
  - (Please use ipcs and ipcrm when working on the inter-process communications assignment !)

15.4 Cryptography as a Security Tool

- Within a given computer the transmittal of messages is safe, reliable and secure, because the OS knows exactly where each one is coming from and where it is going.
- On a network, however, things aren't so straightforward - A rogue computer (or e-mail sender) may spoof their identity, and outgoing packets are delivered to a lot of other computers besides their (intended) final destination, which brings up two big questions of security:
  - **Trust -** How can the system be sure that the messages received are really from the source that they say they are, and can that source be trusted?
  - **Confidentiality -** How can one ensure that the messages one is sending are received only by the intended recipient?
- Cryptography can help with both of these problems, through a system of **secrets** and **keys.** In the former case, the key is held by the sender, so that the recipient knows that only the authentic author could have sent the message; In the latter, the key is held by the recipient, so that only the intended recipient can receive the message accurately.
- Keys are designed so that they cannot be divined from any public information, and must be guarded carefully. (*Asymmetric encryption* involve both a public and a private key.)

15.4.1 Encryption

- The basic idea of encryption is to encode a message so that only the desired recipient can decode and read it. Encryption has been around since before the days of Caesar, and is an entire field of study in itself. Only some of the more significant computer encryption schemes will be covered here.
- The basic process of encryption is shown in Figure 15.7, and will form the basis of most of our discussion on encryption. The steps in the procedure and some of the key terminology are as follows:
  1. The **sender** first creates a **message, m** in plaintext.
  2. The message is then entered into an **encryption algorithm, E,** along with the **encryption key, Ke.**
  3. The encryption algorithm generates the **ciphertext, c, = E(Ke)(m).** For any key k, E(k) is an algorithm for generating ciphertext from a message, and both E and E(k) should be efficiently computable functions.
  4. The ciphertext can then be sent over an unsecure network, where it may be received by **attackers.**
  5. The **recipient** enters the ciphertext into a **decryption algorithm, D,** along with the **decryption key, Kd.**
  6. The decryption algorithm re-generates the plaintext message, m, = D(Kd)(c). For any key k, D(k) is an algorithm for generating a clear text message from a ciphertext, and both D and D(k) should be efficiently computable functions.
  7. The algorithms described here must have this important property: Given a ciphertext c, a computer can only compute a message m such that c = E(k)(m) if it possesses D(k). (In other words, the messages can't be decoded unless you have the decryption algorithm and the decryption key.)
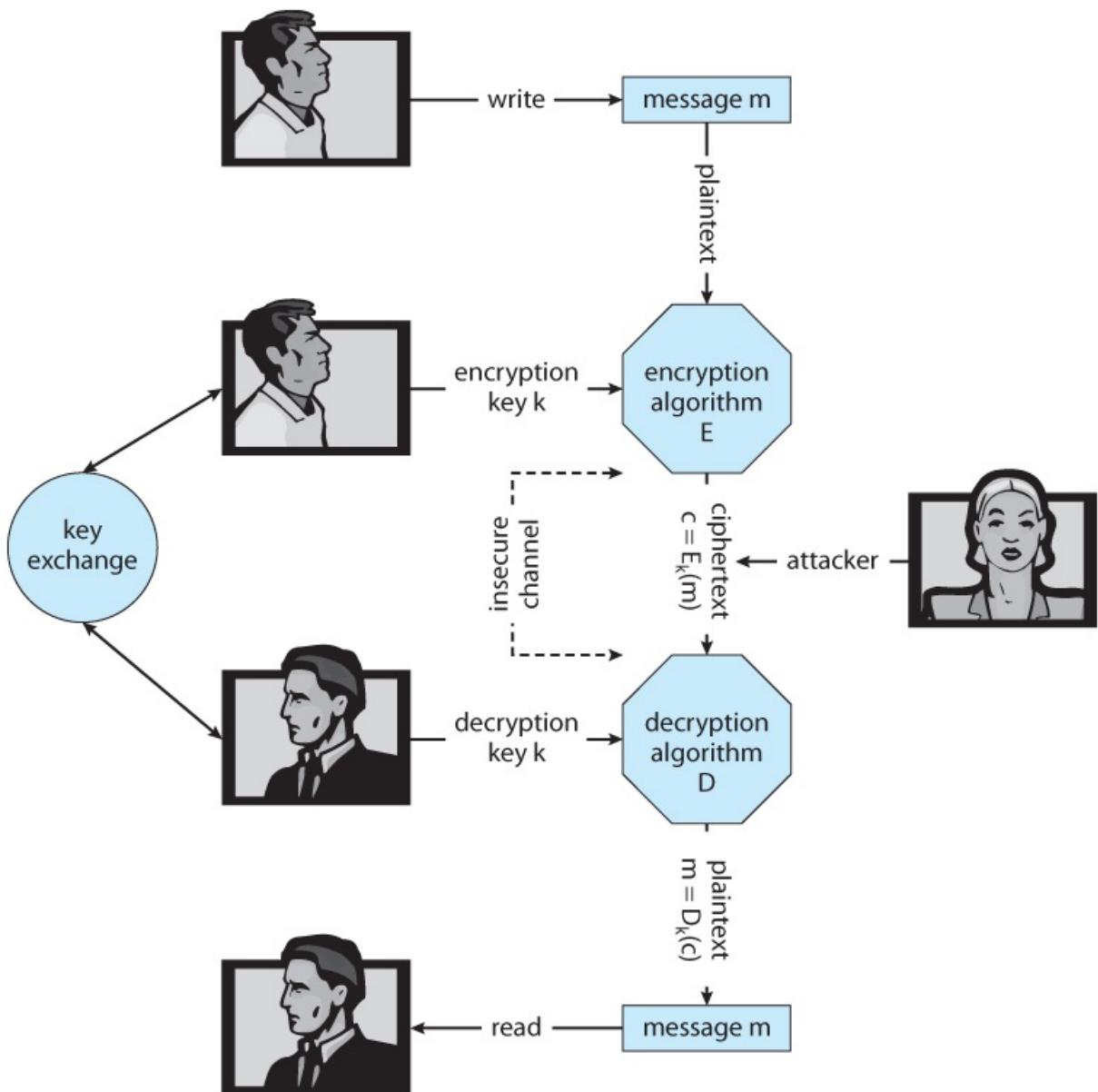
Figure 15.7 - A secure communication over an insecure medium.

15.4.1.1 Symmetric Encryption
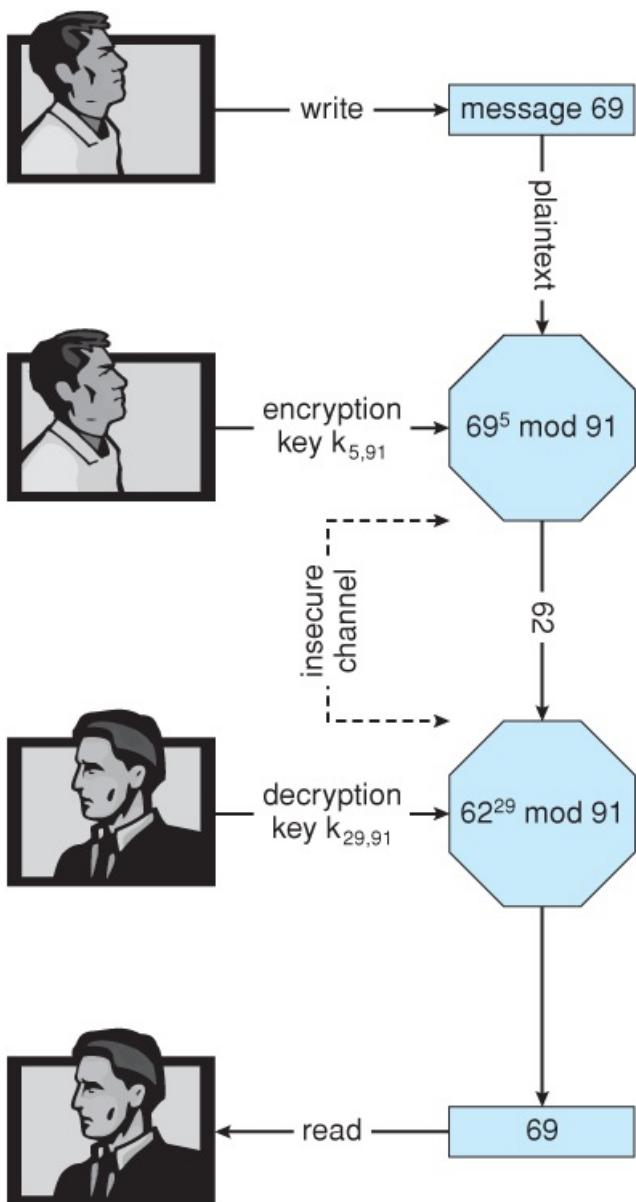
- With **symmetric encryption** the same key is used for both encryption and decryption, and must be safely guarded. There are a number of well-known symmetric encryption algorithms that have been used for computer security:
  - The **Data-Encryption Standard, DES,** developed by the National Institute of Standards, NIST, has been a standard civilian encryption standard for over 20 years. Messages are broken down into 64-bit chunks, each of which are encrypted using a 56-bit key through a series of substitutions and transformations. Some of the transformations are hidden (black boxes), and are classified by the U.S. government.
  - DES is known as a **block cipher,** because it works on blocks of data at a time. Unfortunately this is a vulnerability if the same key is used for an extended amount of data. Therefore an enhancement is to not only encrypt each block, but also to XOR it with the previous block, in a technique known as **cipher-block chaining.**
  - As modern computers become faster and faster, the security of DES has decreased, to where it is now considered insecure because its keys can be exhaustively searched within a reasonable amount of computer time. An enhancement called **triple DES** encrypts the data three times using three separate keys (actually two encryptions and one decryption) for an effective key length of 168 bits. Triple DES is in widespread use today.
  - The **Advanced Encryption Standard, AES,** developed by NIST in 2001 to replace DES uses key lengths of 128, 192, or 256 bits, and encrypts in blocks of 128 bits using 10 to 14 rounds of transformations on a matrix formed from the block.
  - The **twofish algorithm,** uses variable key lengths up to 256 bits and works on 128 bit

blocks.
- *RC5* can vary in key length, block size, and the number of transformations, and runs on a wide variety of CPUs using only basic computations.
- *RC4* is a *stream cipher,* meaning it acts on a stream of data rather than blocks. The key is used to seed a pseudo-random number generator, which generates a *keystream* of keys. RC4 is used in *WEP,* but has been found to be breakable in a reasonable amount of computer time.

15.4.1.2 Asymmetric Encryption

- With *asymmetric encryption,* the decryption key, Kd, is not the same as the encryption key, Ke, and more importantly cannot be derived from it, which means the encryption key can be made publicly available, and only the decryption key needs to be kept secret. (or vice-versa, depending on the application.)
- One of the most widely used asymmetric encryption algorithms is *RSA,* named after its developers - Rivest, Shamir, and Adleman.
- RSA is based on two large prime numbers, *p* and *q,* (on the order of 512 bits each), and their product *N.*
  - Ke and Kd must satisfy the relationship:
    (Ke * Kd) % [(p - 1) * (q - 1)] = = 1
  - The encryption algorithm is:
    $c = E(Ke)(m) = m^{Ke}$ % N
  - The decryption algorithm is:
    $m = D(Kd)(c) = c^{Kd}$ % N
- An example using small numbers:
  - p = 7
  - q = 13
  - N = 7 * 13 = 91
  - (p - 1) * (q - 1) = 6 * 12 = 72
  - Select Ke < 72 and relatively prime to 72, say 5
  - Now select Kd, such that (Ke * Kd) % 72 = = 1, say 29
  - The public key is now (5, 91) and the private key is (29, 91)
  - Let the message, m = 42
  - Encrypt: $c = 42^5$ % 91 = 35
  - Decrypt: $m = 35^{29}$ % 91 = 42

Figure 15.8 - Encryption and decryption using RSA asymmetric cryptography

- Note that asymmetric encryption is much more computationally expensive than symmetric encryption, and as such it is not normally used for large transmissions. Asymmetric encryption is suitable for small messages, authentication, and key distribution, as covered in the following sections.

15.4.1.3 Authentication

- Authentication involves verifying the identity of the entity who transmitted a message.
- For example, if D(Kd)(c) produces a valid message, then we know the sender was in possession of E(Ke).
- This form of authentication can also be used to verify that a message has not been modified
- Authentication revolves around two functions, used for **signatures** (or **signing**), and **verification:**
    - A signing function, **S(Ks)** that produces an **authenticator, A,** from any given message m.
    - A Verification function, **V(Kv,m,A)** that produces a value of "true" if A was created from m, and "false" otherwise.
    - Obviously S and V must both be computationally efficient.
    - More importantly, it must not be possible to generate a valid authenticator, A, without having possession of S(Ks).
    - Furthermore, it must not be possible to divine S(Ks) from the combination of (m and A), since both are sent visibly across networks.
- Understanding authenticators begins with an understanding of hash functions, which is the first step:
    - **Hash functions, H(m)** generate a small fixed-size block of data known as a **message**

digest, or ***hash value*** from any given input data.

- ○ For authentication purposes, the hash function must be ***collision resistant on m.*** That is it should not be reasonably possible to find an alternate message m' such that H(m') = H(m).
- ○ Popular hash functions are **MD5,** which generates a 128-bit message digest, and **SHA-1,** which generates a 160-bit digest.
- Message digests are useful for detecting (accidentally) changed messages, but are not useful as authenticators, because if the hash function is known, then someone could easily change the message and then generate a new hash value for the modified message. Therefore authenticators take things one step further by encrypting the message digest.
- A ***message-authentication code, MAC,*** uses symmetric encryption and decryption of the message digest, which means that anyone capable of verifying an incoming message could also generate a new message.
- An asymmetric approach is the ***digital-signature algorithm,*** which produces authenticators called ***digital signatures.*** In this case Ks and Kv are separate, Kv is the public key, and it is not practical to determine S(Ks) from public information. In practice the sender of a message signs it (produces a digital signature using S(Ks)), and the receiver uses V(Kv) to verify that it did indeed come from a trusted source, and that it has not been modified.
- There are three good reasons for having separate algorithms for encryption of messages and authentication of messages:
  1. Authentication algorithms typically require fewer calculations, making verification a faster operation than encryption.
  2. Authenticators are almost always smaller than the messages, improving space efficiency. (?)
  3. Sometimes we want authentication only, and not confidentiality, such as when a vendor issues a new software patch.
- Another use of authentication is ***non-repudiation,*** in which a person filling out an electronic form cannot deny that they were the ones who did so.

15.4.1.4 Key Distribution

- Key distribution with symmetric cryptography is a major problem, because all keys must be kept secret, and they obviously can't be transmitted over unsecure channels. One option is to send them ***out-of-band,*** say via paper or a confidential conversation.
- Another problem with symmetric keys, is that a separate key must be maintained and used for each correspondent with whom one wishes to exchange confidential information.
- Asymmetric encryption solves some of these problems, because the public key can be freely transmitted through any channel, and the private key doesn't need to be transmitted anywhere. Recipients only need to maintain one private key for all incoming messages, though senders must maintain a separate public key for each recipient to which they might wish to send a message. Fortunately the public keys are not confidential, so this ***key-ring*** can be easily stored and managed.
- Unfortunately there are still some security concerns regarding the public keys used in asymmetric encryption. Consider for example the following man-in-the-middle attack involving phony public keys:
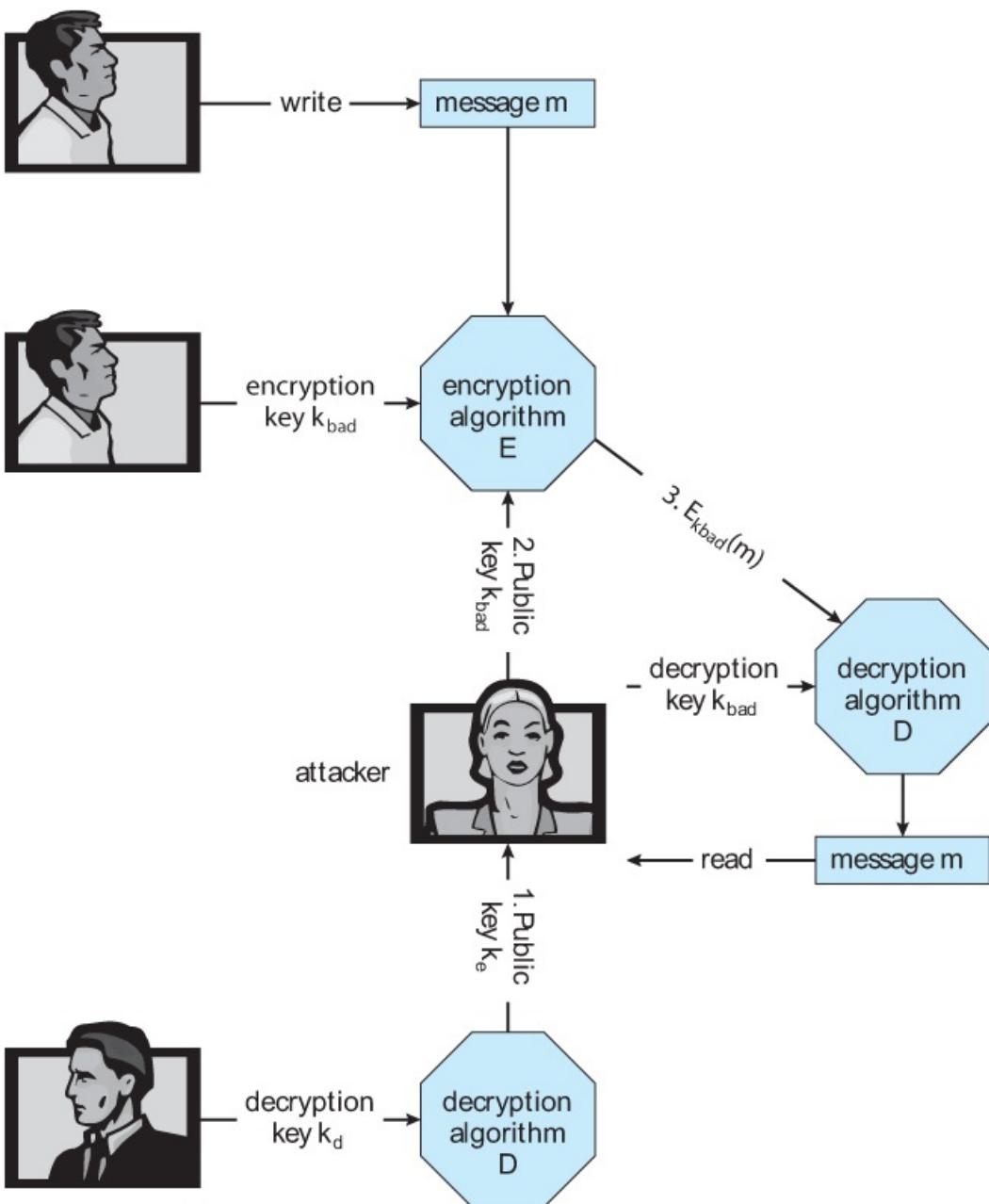
Figure 15.9 - A man-in-the-middle attack on asymmetric cryptography.

- One solution to the above problem involves *digital certificates,* which are public keys that have been digitally signed by a trusted third party. But wait a minute - How do we trust that third party, and how do we know *they* are really who they say they are? Certain *certificate authorities* have their public keys included within web browsers and other certificate consumers before they are distributed. These certificate authorities can then vouch for other trusted entities and so on in a web of trust, as explained more fully in section 15.4.3.

15.4.2 Implementation of Cryptography

- Network communications are implemented in multiple layers - Physical, Data Link, Network, Transport, and Application being the most common breakdown.
- Encryption and security can be implemented at any layer in the stack, with pros and cons to each choice:
    - Because packets at lower levels contain the contents of higher layers, encryption at lower layers automatically encrypts higher layer information at the same time.
    - However security and authorization may be important to higher levels independent of the underlying transport mechanism or route taken.
- At the network layer the most common standard is **IPSec,** a secure form of the IP layer, which is used to set up **Virtual Private Networks, VPNs.**
- At the transport layer the most common implementation is SSL, described below.

15.4.3 An Example: SSL

- SSL (Secure Sockets Layer) 3.0 was first developed by Netscape, and has now evolved into the industry-standard TLS protocol. It is used by web browsers to communicate securely with web servers, making it perhaps the most widely used security protocol on the Internet today.
- SSL is quite complex with many variations, only a simple case of which is shown here.
- The heart of SSL is *session keys*, which are used once for symmetric encryption and then discarded, requiring the generation of new keys for each new session. The big challenge is how to safely create such keys while avoiding man-in-the-middle and replay attacks.
- Prior to commencing the transaction, the server obtains a *certificate* from a *certification authority, CA,* containing:
    - Server attributes such as unique and common names.
    - Identity of the public encryption algorithm, E( ), for the server.
    - The public key, k_e for the server.
    - The validity interval within which the certificate is valid.
    - A digital signature on the above issued by the CA:
        - a = S(K_CA)((attrs, E(k_e), interval)
- In addition, the client will have obtained a public *verification algorithm,* V(K_CA), for the certifying authority. Today's modern browsers include these built-in by the browser vendor for a number of trusted certificate authorities.
- The procedure for establishing secure communications is as follows:
    1. The client, c, connects to the server, s, and sends a random 28-byte number, n_c.
    2. The server replies with its own random value, n_s, along with its certificate of authority.
    3. The client uses its verification algorithm to confirm the identity of the sender, and if all checks out, then the client generates a 46 byte random **premaster secret, pms,** and sends an encrypted version of it as cpms = E(k_s)(pms)
    4. The server recovers pms as D(k_s)(cpms).
    5. Now both the client and the server can compute a shared 48-byte *master secret, ms,* = f(pms, n_s, n_c)
    6. Next, both client and server generate the following from ms:
        - Symmetric encryption keys k_sc_crypt and k_cs_crypt for encrypting messages from the server to the client and vice-versa respectively.
        - MAC generation keys k_sc_mac and k_cs_mac for generating authenticators on messages from server to client and client to server respectively.
    7. To send a message to the server, the client sends:
        - c = E(k_cs_crypt)(m, S(k_cs_mac))(m)))
    8. Upon receiving c, the server recovers:
        - (m,a) = D(k_cs_crypt)(c)
        - and accepts it if V(k_sc_mac)(m,a) is true.
- This approach enables both the server and client to verify the authenticity of every incoming message, and to ensure that outgoing messages are only readable by the process that originally participated in the key generation.
- SSL is the basis of many secure protocols,including *Virtual Private Networks, VPNs,* in which private data is distributed over the insecure public internet structure in an encrypted fashion that emulates a privately owned network.

15.5 User Authentication

- A lot of chapter 14, Protection, dealt with making sure that only certain users were allowed to perform certain tasks, i.e. that a users privileges were dependent on his or her identity. But how does one verify that identity to begin with?

15.5.1 Passwords

- Passwords are the most common form of user authentication. If the user is in possession of the correct password, then they are considered to have identified themselves.
- In theory separate passwords could be implemented for separate activities, such as reading this file, writing that file, etc. In practice most systems use one password to confirm user identity, and then authorization is based upon that identification. This is a result of the classic trade-off between security and convenience.

15.5.2 Password Vulnerabilities

- Passwords can be guessed.
    - Intelligent guessing requires knowing something about the intended target in specific, or about people and commonly used passwords in general.

- Brute-force guessing involves trying every word in the dictionary, or every valid combination of characters. For this reason good passwords should not be in any dictionary (in any language), should be reasonably lengthy, and should use the full range of allowable characters by including upper and lower case characters, numbers, and special symbols.
- "Shoulder surfing" involves looking over people's shoulders while they are typing in their password.
  - Even if the lurker does not get the entire password, they may get enough clues to narrow it down, especially if they watch on repeated occasions.
  - Common courtesy dictates that you look away from the keyboard while someone is typing their password.
  - Passwords echoed as stars or dots still give clues, because an observer can determine how many characters are in the password. :-(
- "Packet sniffing" involves putting a monitor on a network connection and reading data contained in those packets.
  - SSH encrypts all packets, reducing the effectiveness of packet sniffing.
  - However you should still never e-mail a password, particularly not with the word "password" in the same message or worse yet the subject header.
  - Beware of any system that transmits passwords in clear text. ("Thank you for signing up for XYZ. Your new account and password information are shown below".) You probably want to have a spare throw-away password to give these entities, instead of using the same high-security password that you use for banking or other confidential uses.
- Long hard to remember passwords are often written down, particularly if they are used seldomly or must be changed frequently. Hence a security trade-off of passwords that are easily divined versus those that get written down. :-(
- Passwords can be given away to friends or co-workers, destroying the integrity of the entire user-identification system.
- Most systems have configurable parameters controlling password generation and what constitutes acceptable passwords.
  - They may be user chosen or machine generated.
  - They may have minimum and/or maximum length requirements.
  - They may need to be changed with a given frequency. (In extreme cases for every session.)
  - A variable length history can prevent repeating passwords.
  - More or less stringent checks can be made against password dictionaries.

15.5.3 Encrypted Passwords

- Modern systems do not store passwords in clear-text form, and hence there is no mechanism to look up an existing password.
- Rather they are encrypted and stored in that form. When a user enters their password, that too is encrypted, and if the encrypted version match, then user authentication passes.
- The encryption scheme was once considered safe enough that the encrypted versions were stored in the publicly readable file "/etc/passwd".
  - They always encrypted to a 13 character string, so an account could be disabled by putting a string of any other length into the password field.
  - Modern computers can try every possible password combination in a reasonably short time, so now the encrypted passwords are stored in files that are only readable by the super user. Any password-related programs run as setuid root to get access to these files. (/etc/shadow)
  - A random seed is included as part of the password generation process, and stored as part of the encrypted password. This ensures that if two accounts have the same plain-text password that they will not have the same encrypted password. However cutting and pasting encrypted passwords from one account to another will give them the same plain-text passwords.

15.5.4 One-Time Passwords

- One-time passwords resist shoulder surfing and other attacks where an observer is able to capture a password typed in by a user.
  - These are often based on a **challenge** and a **response.** Because the challenge is different each time, the old response will not be valid for future challenges.
    - For example, The user may be in possession of a secret function f(x). The system challenges with some given value for x, and the user responds with f(x), which the system can then verify. Since the challenger gives a different (random) x each time, the answer is constantly changing.
    - A variation uses a map (e.g. a road map) as the key. Today's question might be "On what corner is SEO located?", and tomorrow's question might be "How far is it from Navy Pier to

Wrigley Field?" Obviously "Taylor and Morgan" would not be accepted as a valid answer for the second question!
- Another option is to have some sort of electronic card with a series of constantly changing numbers, based on the current time. The user enters the current number on the card, which will only be valid for a few seconds. A *two-factor authorization* also requires a traditional password in addition to the number on the card, so others may not use it if it were ever lost or stolen.
- A third variation is a *code book,* or *one-time pad.* In this scheme a long list of passwords is generated, and each one is crossed off and cancelled as it is used. Obviously it is important to keep the pad secure.

15.5.5 Biometrics

- Biometrics involve a physical characteristic of the user that is not easily forged or duplicated and not likely to be identical between multiple users.
  - Fingerprint scanners are getting faster, more accurate, and more economical.
  - Palm readers can check thermal properties, finger length, etc.
  - Retinal scanners examine the back of the users' eyes.
  - Voiceprint analyzers distinguish particular voices.
  - Difficulties may arise in the event of colds, injuries, or other physiological changes.

15.6 Implementing Security Defenses

15.6.1 Security Policy

- A security policy should be well thought-out, agreed upon, and contained in a living document that everyone adheres to and is updated as needed.
- Examples of contents include how often port scans are run, password requirements, virus detectors, etc.

15.6.2 Vulnerability Assessment

- Periodically examine the system to detect vulnerabilities.
  - Port scanning.
  - Check for bad passwords.
  - Look for suid programs.
  - Unauthorized programs in system directories.
  - Incorrect permission bits set.
  - Program checksums / digital signatures which have changed.
  - Unexpected or hidden network daemons.
  - New entries in startup scripts, shutdown scripts, cron tables, or other system scripts or configuration files.
  - New unauthorized accounts.
- The government considers a system to be only as secure as its most far-reaching component. Any system connected to the Internet is inherently less secure than one that is in a sealed room with no external communications.
- Some administrators advocate "security through obscurity", aiming to keep as much information about their systems hidden as possible, and not announcing any security concerns they come across. Others announce security concerns from the rooftops, under the theory that the hackers are going to find out anyway, and the only one kept in the dark by obscurity are honest administrators who need to get the word.

15.6.3 Intrusion Detection

- Intrusion detection attempts to detect attacks, both successful and unsuccessful attempts. Different techniques vary along several axes:
  - The time that detection occurs, either during the attack or after the fact.
  - The types of information examined to detect the attack(s). Some attacks can only be detected by analyzing multiple sources of information.
  - The response to the attack, which may range from alerting an administrator to automatically stopping the attack (e.g. killing an offending process), to tracing back the attack in order to identify the attacker.
    - Another approach is to divert the attacker to a *honeypot*, on a *honeynet.* The idea behind a honeypot is a computer running normal services, but which no one uses to do any real work. Such a system should not see any network traffic under normal conditions, so any traffic going to or from such a system is by definition suspicious. Honeypots are normally kept on a honeynet

protected by a **reverse firewall,** which will let potential attackers in to the honeypot, but will not allow any outgoing traffic. (So that if the honeypot is compromised, the attacker cannot use it as a base of operations for attacking other systems.) Honeypots are closely watched, and any suspicious activity carefully logged and investigated.

- Intrusion Detection Systems, IDSs, raise the alarm when they detect an intrusion. Intrusion Detection and Prevention Systems, IDPs, act as filtering routers, shutting down suspicious traffic when it is detected.
- There are two major approaches to detecting problems:
  - **Signature-Based Detection** scans network packets, system files, etc. looking for recognizable characteristics of known attacks, such as text strings for messages or the binary code for "exec /bin/sh". The problem with this is that it can only detect previously encountered problems for which the signature is known, requiring the frequent update of signature lists.
  - **Anomaly Detection** looks for "unusual" patterns of traffic or operation, such as unusually heavy load or an unusual number of logins late at night.
    - The benefit of this approach is that it can detect previously unknown attacks, so called **zero-day attacks.**
    - One problem with this method is characterizing what is "normal" for a given system. One approach is to benchmark the system, but if the attacker is already present when the benchmarks are made, then the "unusual" activity is recorded as "the norm."
    - Another problem is that not all changes in system performance are the result of security attacks. If the system is bogged down and really slow late on a Thursday night, does that mean that a hacker has gotten in and is using the system to send out SPAM, or does it simply mean that a CS 385 assignment is due on Friday? :-)
    - To be effective, anomaly detectors must have a very low **false alarm (false positive)** rate, lest the warnings get ignored, as well as a low **false negative** rate in which attacks are missed.

15.6.4 Virus Protection

- Modern anti-virus programs are basically signature-based detection systems, which also have the ability (in some cases) of **disinfecting** the affected files and returning them back to their original condition.
- Both viruses and anti-virus programs are rapidly evolving. For example viruses now commonly mutate every time they propagate, and so anti-virus programs look for families of related signatures rather than specific ones.
- Some antivirus programs look for anomalies, such as an executable program being opened for writing (other than by a compiler.)
- Avoiding bootleg, free, and shared software can help reduce the chance of catching a virus, but even shrink-wrapped official software has on occasion been infected by disgruntled factory workers.
- Some virus detectors will run suspicious programs in a **sandbox,** an isolated and secure area of the system which mimics the real system.
- **Rich Text Format, RTF,** files cannot carry macros, and hence cannot carry Word macro viruses.
- Known safe programs (e.g. right after a fresh install or after a thorough examination) can be digitally signed, and periodically the files can be re-verified against the stored digital signatures. (Which should be kept secure, such as on off-line write-only medium.)

15.6.5 Auditing, Accounting, and Logging

- Auditing, accounting, and logging records can also be used to detect anomalous behavior.
- Some of the kinds of things that can be logged include authentication failures and successes, logins, running of suid or sgid programs, network accesses, system calls, etc. In extreme cases almost every keystroke and electron that moves can be logged for future analysis. (Note that on the flip side, all this detailed logging can also be used to analyze system performance. The down side is that the logging also *affects* system performance (negatively!), and so a Heisenberg effect applies.)
- "The Cuckoo's Egg" tells the story of how Cliff Stoll detected one of the early UNIX break ins when he noticed anomalies in the accounting records on a computer system being used by physics researchers.

Tripwire Filesystem (New Sidebar)

- The tripwire filesystem monitors files and directories for changes, on the assumption that most intrusions eventually result in some sort of undesired or unexpected file changes.
- The tw.config file indicates what directories are to be monitored, as well as what properties of each file are to be recorded. (E.g. one may choose to monitor permission and content changes, but not worry about read access times.)
- When first run, the selected properties for all monitored files are recorded in a database. Hash codes are

used to monitor file contents for changes.
- Subsequent runs report any changes to the recorded data, including hash code changes, and any newly created or missing files in the monitored directories.
- For full security it is necessary to also protect the tripwire system itself, most importantly the database of recorded file properties. This could be saved on some external or write-only location, but that makes it harder to change the database when legitimate changes are made.
- It is difficult to monitor files that are *supposed to* change, such as log files. The best tripwire can do in this case is to watch for anomalies, such as a log file that shrinks in size.
- Free and commercial versions are available at http://tripwire.org and http://tripwire.com.

15.7 Firewalling to Protect Systems and Networks

- Firewalls are devices (or sometimes software) that sit on the border between two security domains and monitor/log activity between them, sometimes restricting the traffic that can pass between them based on certain criteria.
- For example a firewall router may allow HTTP: requests to pass through to a web server inside a company domain while not allowing telnet, ssh, or other traffic to pass through.
- A common architecture is to establish a de-militarized zone, DMZ, which sort of sits "between" the company domain and the outside world, as shown below. Company computers can reach either the DMZ or the outside world, but outside computers can only reach the DMZ. Perhaps most importantly, the DMZ cannot reach any of the other company computers, so even if the DMZ is breached, the attacker cannot get to the rest of the company network. (In some cases the DMZ may have limited access to company computers, such as a web server on the DMZ that needs to query a database on one of the other company computers.)
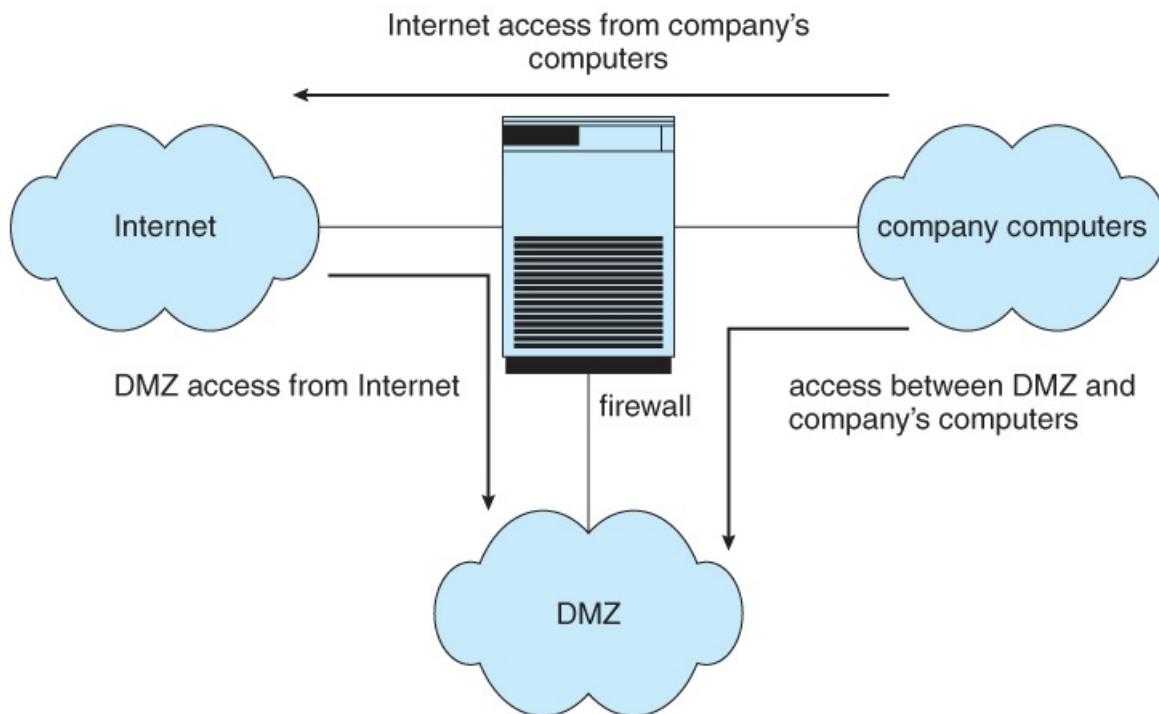

Figure 15.10 - Domain separation via firewall.

- Firewalls themselves need to be resistant to attacks, and unfortunately have several vulnerabilities:
  - *Tunneling,* which involves encapsulating forbidden traffic inside of packets that are allowed.
  - Denial of service attacks addressed at the firewall itself.
  - Spoofing, in which an unauthorized host sends packets to the firewall with the return address of an authorized host.
- In addition to the common firewalls protecting a company internal network from the outside world, there are also some specialized forms of firewalls that have been recently developed:
  - A *personal firewall* is a software layer that protects an individual computer. It may be a part of the operating system or a separate software package.
  - An *application proxy firewall* understands the protocols of a particular service and acts as a stand-in (and relay) for the particular service. For example, and SMTP proxy firewall would accept SMTP requests from the outside world, examine them for security concerns, and forward only the "safe" ones on to the real SMTP server behind the firewall.
  - *XML firewalls* examine XML packets only, and reject ill-formed packets. Similar firewalls exist for other specific protocols.

- **System call firewalls** guard the boundary between user mode and system mode, and reject any system calls that violate security policies.

15.8 Computer-Security Classifications (Optional)

- No computer system can be 100% secure, and attempts to make it so can quickly make it unusable.
- However one can establish a level of trust to which one feels "safe" using a given computer system for particular security needs.
- The U.S. Department of Defense's "Trusted Computer System Evaluation Criteria" defines four broad levels of trust, and sub-levels in some cases:
  - Level D is the least trustworthy, and encompasses all systems that do not meet any of the more stringent criteria. DOS and Windows 3.1 fall into level D, which has no user identification or authorization, and anyone who sits down has full access and control over the machine.
  - Level C1 includes user identification and authorization, and some means of controlling what users are allowed to access what files. It is designed for use by a group of mostly cooperating users, and describes most common UNIX systems.
  - Level C2 adds individual-level control and monitoring. For example file access control can be allowed or denied on a per-individual basis, and the system administrator can monitor and log the activities of specific individuals. Another restriction is that when one user uses a system resource and then returns it back to the system, another user who uses the same resource later cannot read any of the information that the first user stored there. (I.e. buffers, etc. are wiped out between users, and are not left full of old contents.) Some special secure versions of UNIX have been certified for C2 security levels, such as SCO.
  - Level B adds sensitivity labels on each object in the system, such as "secret", "top secret", and "confidential". Individual users have different clearance levels, which controls which objects they are able to access. All human-readable documents are labeled at both the top and bottom with the sensitivity level of the file.
  - Level B2 extends sensitivity labels to all system resources, including devices. B2 also supports covert channels and the auditing of events that could exploit covert channels.
  - B3 allows creation of access-control lists that denote users NOT given access to specific objects.
  - Class A is the highest level of security. Architecturally it is the same as B3, but it is developed using formal methods which can be used to *prove* that the system meets all requirements and cannot have any possible bugs or other vulnerabilities. Systems in class A and higher may be developed by trusted personnel in secure facilities.
  - These classifications determine what a system *can* implement, but it is up to security policy to determine *how* they are implemented in practice. These systems and policies can be reviewed and certified by trusted organizations, such as the National Computer Security Center. Other standards may dictate physical protections and other issues.

15.9 An Example: Windows XP (Optional)

- Windows XP is a general purpose OS designed to support a wide variety of security features and methods. It is based on user accounts which can be grouped in any manner.
- When a user logs on, a **security access token** is issued that includes the security ID for the user, security IDs for any groups of which the user is a member, and a list of any special privileges the user has, such as performing backups, shutting down the system, and changing the system clock.
- Every process running on behalf of a user gets a copy of the users security token, which determines the privileges of that process running on behalf of that user.
- Authentication is normally done via passwords, but the modular design of XP allows for alternative authentication such as retinal scans or fingerprint readers.
- Windows XP includes built-in auditing that allows many common security threats to be monitored, such as successful and unsuccessful logins, logouts, attempts to write to executable files, and access to certain sensitive files.
- Security attributes of objects are described by **security descriptors**, which include the ID of the owner, group ownership for POSIX subsystems only, a discretionary access-control list describing exactly what permissions each user or group on the system has for this particular object, and auditing control information.
- The access control lists include for each specified user or group either AccessAllowed or AccessDenied for the following types of actions: ReadData,WriteData, AppendData, Execute, ReadAttributes, WriteAttributes, ReadExtendedAttribute, and WriteExtendedAttribute.
- **Container objects** such as directories can logically contain other objects. When a new object is created in a container or copied into a container, by default it inherits the permissions of the new container. **Noncontainer objects** inherit no other permissions. If the permissions of the container are changed later, that does not affect the permissions of the contained objects.
- Although Windows XP is capable of supporting a secure system, many of the security features are not enabled by default, resulting in a fair number of security breaches on XP systems. There are also a large number of system daemons and other programs that start automatically at startup, whether the system administrator has thought about

them or not. (My system currently has 54 processes running, most of which I did not deliberately start and which have short cryptic names which makes it hard to divine exactly what they do or why. Faced with this situation, most users and administrators will simply leave alone anything they don't understand.)

15.10 Summary