Atria Institute of Technology

Department of Information Science and Engineering

Bengaluru-560024

ACADEMIC YEAR: 2021-2022

EVEN SEMESTER NOTES

Semester : 4th Semester

Subject Name : Operating Systems

Subject Code : 18CS43

Faculty Name : Mr. Omprakash B

OS-MODULE I

What is an Operating System?

A program that acts as an intermediary between a user of a computer and the computer hardware

Operating system goals:

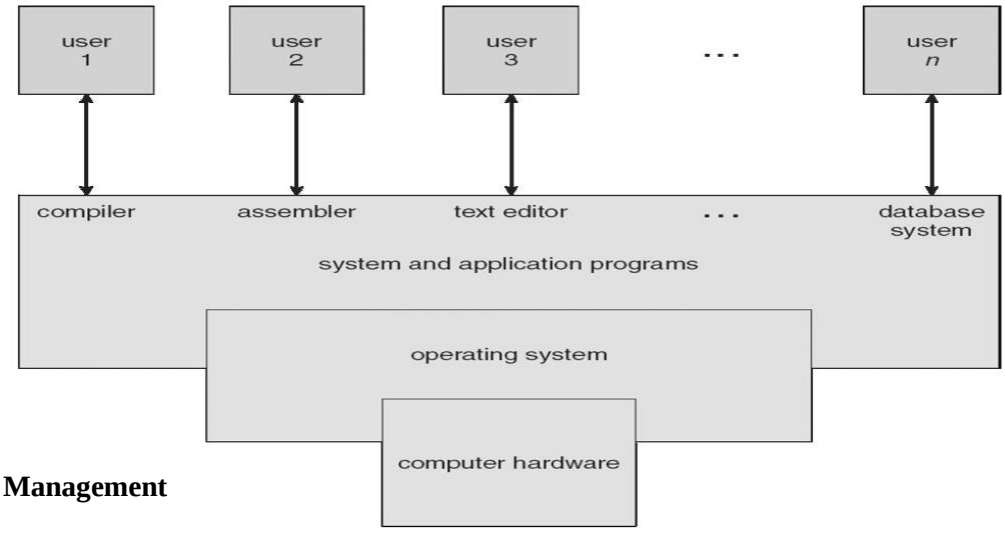
- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

Computer System Structure

Computer system can be divided into four components

- Hardware – provides basic computing resources CPU, memory, I/O devices
- Operating system-Controls and coordinates use of hardware among various applications and users
- Application programs – define the ways in which the system resources are used to solve the computing problems of the users
  -Word processors, compilers, web browsers, database systems, video games
- Users
  - People, machines, other computers

Four Components of a Computer System



Process Management

- A process is a program in execution. It is a unit of work within the system. Program is a *passive entity*, process is an *active entity*.
- Process needs resources to accomplish its task
- CPU, memory, I/O, files
- Initialization data
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one **program counter** specifying location of next instruction to execute
- Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
- Concurrency by multiplexing the CPUs among the processes / threads

Process Management Activities

- The operating system is responsible for the following activities in connection with process management:
- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

Memory Management

- All data in memory before and after processing
- All instructions in memory in order to execute
- Memory management determines what is in memory when
- Optimizing CPU utilization and computer response to users
- **Memory management activities**
- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes (or parts thereof) and data to move into and out of memory
- Allocating and deal locating memory space as needed

Storage Management

- OS provides uniform, logical view of information storage
- Abstracts physical properties to logical storage unit - **file**
- Each medium is controlled by device (i.e., disk drive, tape drive)
- Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
- Files usually organized into directories
- Access control on most systems to determine who can access what

OS activities include

- Creating and deleting files and directories
- Primitives to manipulate files and dirs

-
- Mapping files onto secondary storage
- Backup files onto stable (non-volatile) storage media

Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a "long" period of time
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms
-

MASS STORAGE activities

- Free-space management
- Storage allocation
- Disk scheduling
- Some storage need not be fast
- Tertiary storage includes optical storage, magnetic tape
- Still must be managed
- Varies between WORM (write-once, read-many-times) and RW (read-write)

1.1.2 Operating-System Structure

Simple Structure

Many commercial systems do not have well-defined structures. Frequently, such operating systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system.

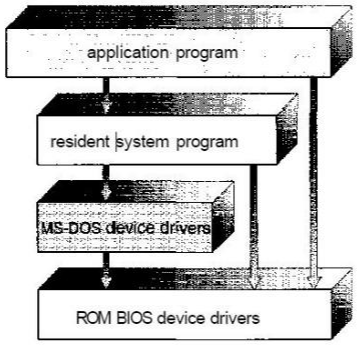


Figure 2.10 MS-DOS layer structure.

It was written to provide the most functionality in the least space, so it was not divided into modules carefully. In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail. Of course, MS-DOS was also limited by the hardware of its era. Another example of limited structuring is the original UNIX operating system. UNIX is another system that initially was limited by hardware functionality.

It consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved.

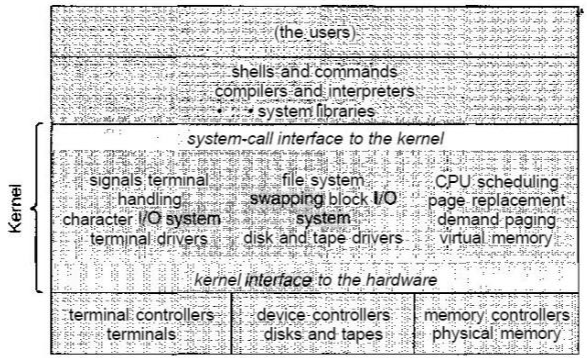


Figure 2.11 UNIX system structure.

Layered Approach

The operating system can then retain much greater control over the computer and over the applications that make use of that computer. Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems. Under the top down approach, the overall functionality and features are determined and are separated into components. Information hiding is also important, because it leaves programmers free to implement the low-level routines as they see fit, provided that the external interface of the routine stays unchanged and that the routine itself performs the advertised task.

A system can be made modular in many ways. One method is the **layered approach,** in which the operating system is broken up into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.

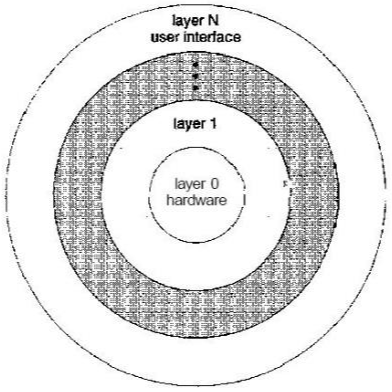


Figure 2.12 A layered operating system.

An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data. A typical operating-system layer—say, layer M—consists of data structures and a set of routines that can be invoked by higher-level layers. Layer *M,* in turn, can invoke operations on lower-level layers.

The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers. This

Approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system is simplified.

Each layer is implemented with only those operations provided by lower level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

The major difficulty with the layered approach involves appropriately defining the various layers. The backing-store driver would normally be above the CPU scheduler, because the driver may need to wait for I/O and the CPU can be rescheduled during this time. A final problem with layered implementations is that they tend to be less efficient than other types. For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware.

Micro kernels

The kernel became large and difficult to manage. In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the **microkernel** approach. This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel. Micro kernels provide minimal process and memory management, in addition to a communication facility.

The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space. One benefit of the microkernel approach is ease of extending the operating system. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel.

The resulting operating system is easier to port from one hardware design to another. The microkernel also provides more security and reliability, since most services are running as user rather than kernel processes. If a service fails, the rest of the operating system remains untouched.

Modules

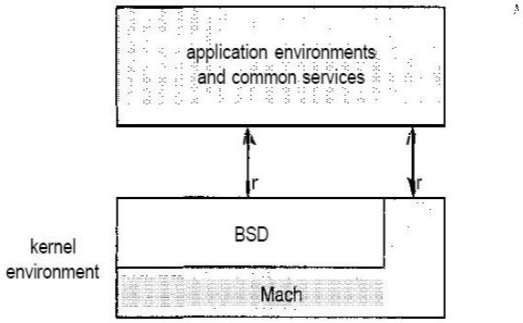
The best current methodology for operating-system design involves using object-oriented programming techniques to create a modular kernel. Here, the kernel has a set of core components and dynamically links in additional services either during boot time or during run time. Such a strategy uses dynamically loadable modules and is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X.

A core kernel with seven types of loadable kernel modules:

1. Scheduling classes
2. File systems
3. Loadable system calls
4. Executable formats
5. STREAMS modules

6. Miscellaneous
7. Device and bus drivers

Such a design allows the kernel to provide core services yet also allows certain features to be implemented dynamically. The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system in that any module can call any other module. The approach is like the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.
The Apple Macintosh Mac OS X operating system uses a hybrid structure. Mac OS X (also known as *Danvin)* structures the operating system using a layered technique where one layer consists of the Mach microkernel. The top layers include application environments and a set of services providing a graphical interface to applications. Below these layers is the kernel environment, which consists primarily of the Mach microkernel and the BSD kernel. Mach provides memory management; support for remote procedure calls (RPCs) and inter process communication (IPC) facilities, including message passing; and thread scheduling. The BSD component provides a BSD command line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads.



1.1.3 Operating-System Operations

1. modern operating systems are **interrupt driven.** If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt or a trap
2. A **trap (or** an **exception)** is a software-generated interrupt caused
   either by an error or by a specific request from a user program
   that an operating-system service is performed.
3. The interrupt-driven nature of an operating system defines that system's general structure. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided that is responsible for dealing with the interrupt.
4. The operating system and the users share the hardware and software resources of the computer system, we need to make sure that an error in a user program could cause problems only for the one program that was running. With sharing, many processes could be adversely affected by a bug in one program. For example, if a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes.
5. Without protection against these sorts of errors, either the computer must execute only one process at a time or all output must be suspect.

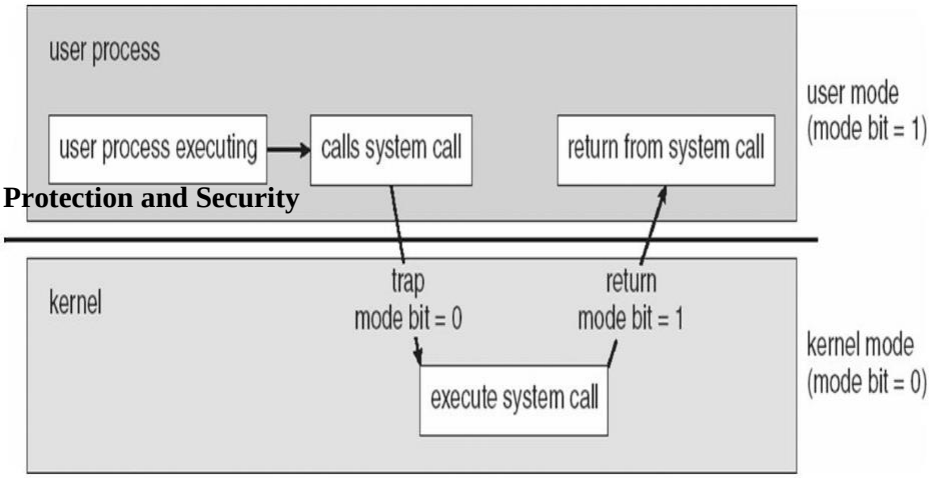
Dual-Mode Operation

Dual-mode operation allows OS to protect itself and other system components **User mode** and **kernel mode**

Mode bit provided by hardware Provides ability to distinguish when system is running user code or kernel code Some instructions designated as **privileged**, only executable in kernel mode System call changes mode to kernel, return from call resets it to user

Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources Set interrupt after specific period
- Operating system decrements counter
- When counter zero generate an interrupt
- Set up before scheduling process to regain control or terminate program that exceeds allotted time



Protection and Security

If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.

1.1.4 Protection and security

Protection is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means for specification of the controls to be imposed and means for enforcement.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. An unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides a means to distinguish between authorized and unauthorized usage, A system can have adequate protection but still be prone to failure and allow inappropriate access.

It is the job of **security** to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of service attacks Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain a list of user names and associated **user identifiers (user IDs).**

- User ID then associated with all files, processes of that user to determine access control

- Group identifier (**group ID**) allows set of users to be defined and controls managed, then also associated with each process, file **Privilege escalation** allows user to change to effective ID with more rights

1.1.5 Kernel Data Structures

The operating system must keep a lot of information about the current state of the system. As things happen within the system these data structures must be changed to reflect the current reality. For example, a new process might be created when a user logs onto the system. The kernel must create a data structure representing the new process and link it with the data structures representing all of the other processes in the system.

Mostly these data structures exist in physical memory and are accessible only by the kernel and its subsystems. Data structures contain data and pointers, addresses of other data structures, or the addresses of routines. Taken all together, the data structures used by the Linux kernel can look very confusing. Every data structure has a purpose and although some are used by several kernel subsystems, they are more simple than they appear at first sight.

Understanding the Linux kernel hinges on understanding its data structures and the use that the various functions within the Linux kernel makes of them. This section bases its description of the Linux kernel on its data structures. It talks about each kernel subsystem in terms of its algorithms, which are its methods of getting things done, and their usage of the kernel's data structures.

1.1.6 Computing Environments

Traditional Computing

As computing matures, the lines separating many of the traditional computing environments are blurring. this environment consisted of PCs connected to a network, with servers providing file and print services. Terminals attached to mainframes were prevalent at many companies as well, with even fewer remote access and portability options.

The current trend is toward providing more ways to access these computing environments. Web technologies are stretching the boundaries of traditional computing. Companies establish **portals,** which provide web accessibility to their internal servers. **Network computers** are essentially terminals that understand web-based computing. Handheld computers can synchronize with PCs to allow very portable use of company information. Handheld PDAs can also connect to **wireless networks** to use the company's web portal.

Batch system processed jobs in bulk, with predetermined input. Interactive systems waited for input from users. To optimize the use of the computing resources, multiple users shared time on these systems. Time-sharing systems used a timer and scheduling algorithms to rapidly cycle processes through the CPU, giving each user a share of the resources.

Client-Server Computing

Designers have shifted away from centralized system architecture. Terminals connected to centralized systems are now being supplanted by PCs. Correspondingly, user interface functionality once handled directly by the centralized systems is increasingly being handled by the PCs. As a result, many of today's systems acts as **server systems to** satisfy requests generated by **client systems** Server systems can be broadly categorized as compute servers and file servers:

• The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data); in response, the server executes the action and sends back results to the

client. A server running a database that responds to client requests for data is an example of such a system.

The **file-server system** provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a web server that delivers files to clients running web browsers.

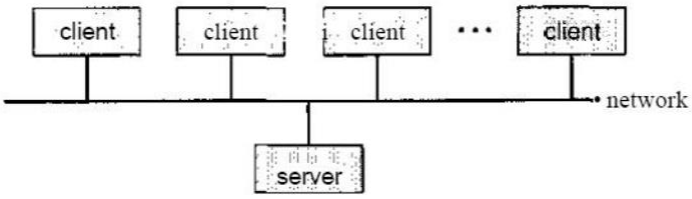


Figure 1.11 General structure of a client-server system.

Peer-to-Peer Computing

In this model, clients and servers are not distinguished from one another; instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service. Peer-to-peer systems offer an advantage over traditional client-server systems. In a client-server system, the server is a bottleneck; but in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.

To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network.

Determining what services are available is accomplished in one of two general ways:

• When a node joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.

• A peer acting as a client must first discover what node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request. To support this approach, a *discovery protocol* must be provided that allows peers to discover services provided by other peers in the network.

Web-Based Computing

The Web has become ubiquitous, leading to more access by a wider variety of devices than was dreamt of a few years ago. Web computing has increased the emphasis on networking. Devices that were not previously networked now include wired or wireless access. Devices that were networked now have faster network connectivity, provided by either improved networking technology, optimized network implementation code, or both.

The implementation of web-based computing has given rise to new categories of devices, such as **load balancers,** which distribute network connections among a pool of similar servers. Operating systems like Windows 95, which acted as web clients, have evolved into Linux and Windows XP, which can act as web servers as well as clients. Generally, the Web has increased the complexity of devices, because their users require them to be web-enabled.

1.1.7 Open-Source Operating Systems

- Operating systems made available in source-code format rather than just binary closed-source
- Counter to the copy protection and Digital Rights Management (DRM) movement

- Started by Free Software Foundation (FSF), which has "copy left" GNU Public License (GPL)
- Examples include GNU/Linux, BSD UNIX (including core of Mac OS X), and Sun Solaris

1.2 OPERATING SYSTEM STRUCTURE

1.2.1 Operating System Services

- One set of operating-system services provides functions that are helpful to the user
- Communications – Processes may exchange information, on the same computer or between computers over a network.
- Communications may be via shared memory or through message passing (packets moved by the OS)
- Error detection – OS needs to be constantly aware of possible errors may occur in the CPU and memory hardware, in I/O devices, in user program
- For each type of error, OS should take the appropriate action to ensure correct and consistent computing.
- Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system.
- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
- **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
- Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
- **Accounting -** To keep track of which users use how much and what kinds of computer resources
- **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other.
- **Protection** involves ensuring that all access to system resources is controlled.
- **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts.
- If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

1.2.2 User and Operating System Interface - CLI

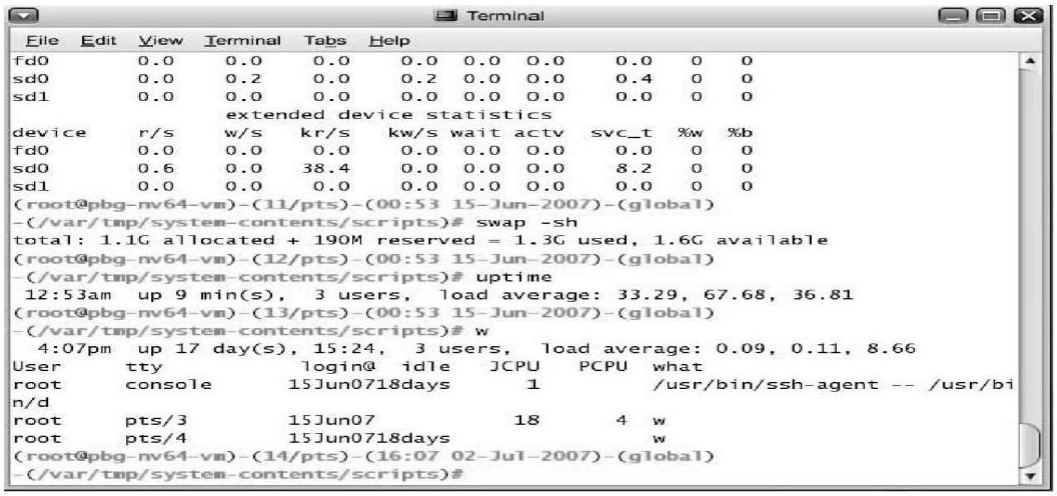
- Command Line Interface (CLI) or command interpreter allows direct command entry Sometimes implemented in kernel, sometimes by systems program
    - Sometimes multiple flavors implemented – shells
    - Primarily fetches a command from user and executes it
- Sometimes commands built-in, sometimes just names of programs If the latter, adding new features doesn't require shell modification

User Operating System Interface - GUI

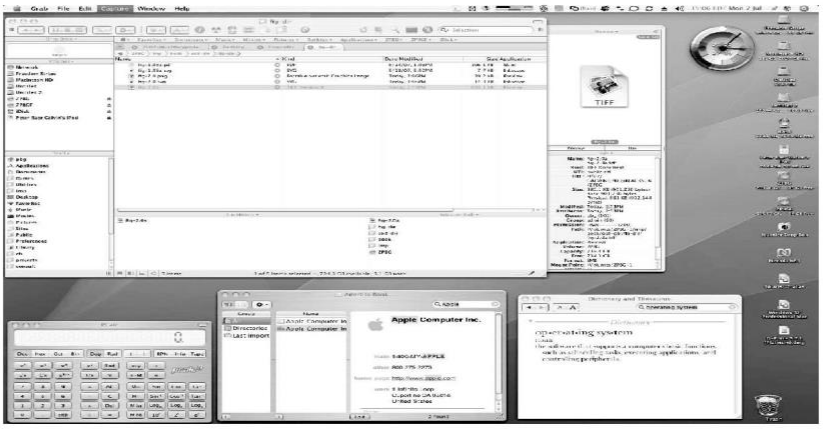
- User-friendly desktop metaphor interface
- Usually mouse, keyboard, and monitor
- Icons represent files, programs, actions, etc
- Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)

- Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
- Microsoft Windows is GUI with CLI "command" shell
- Apple Mac OS X as "Aqua" GUI interface with UNIX kernel underneath and shells available
- Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)

Bourne Shell Command Interpreter



The Mac OS X GUI

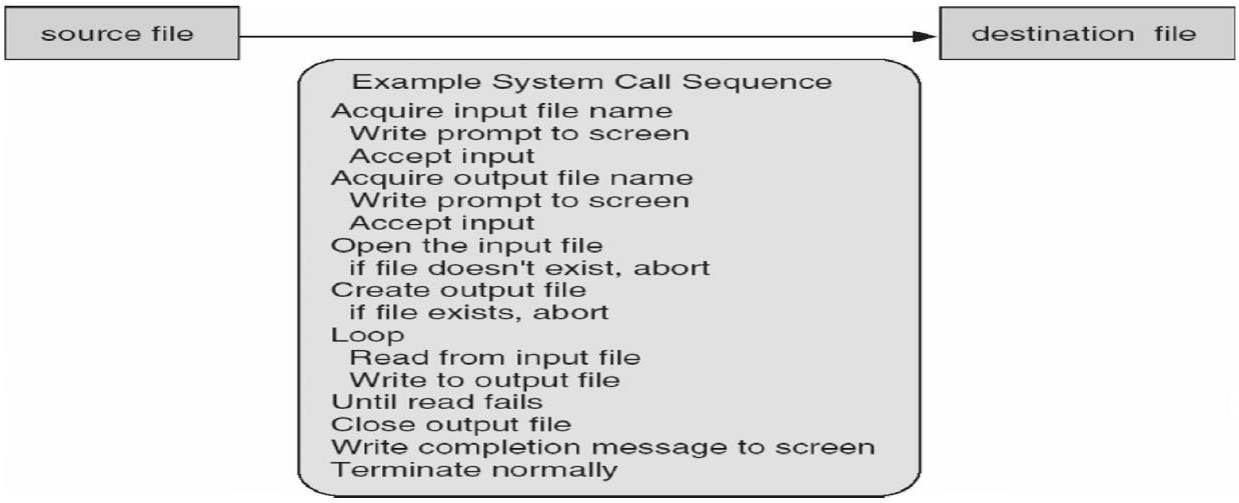


1.2.3 System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call usenThree most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

- Why use APIs rather than system calls?(Note that the system-call names used throughout this text are generic)

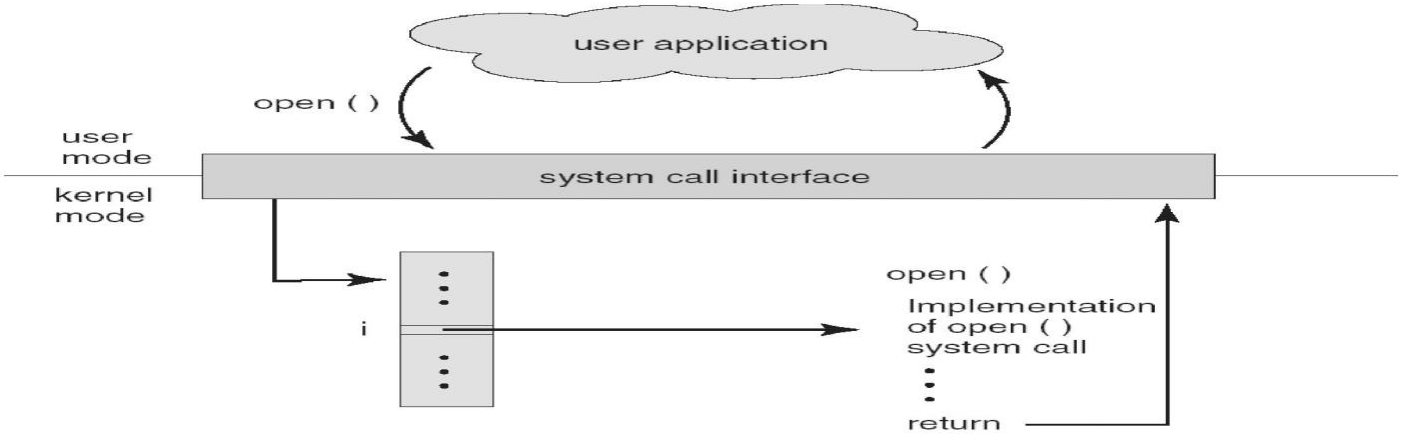
Example of System Calls

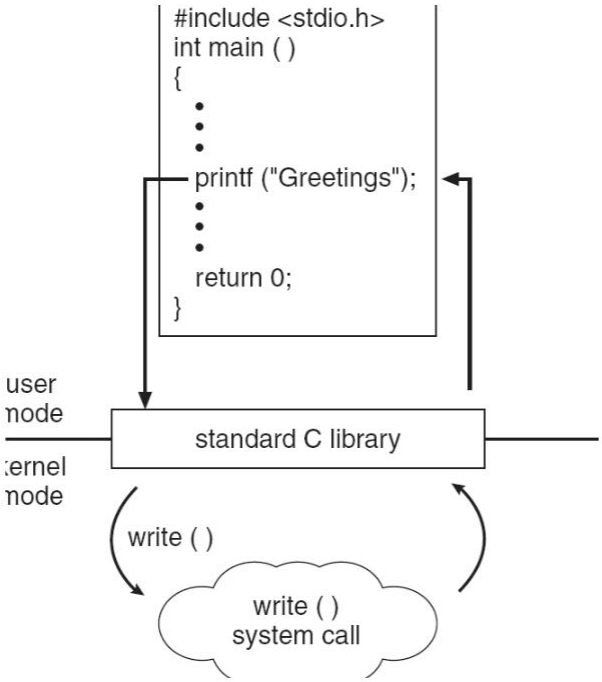


System Call Implementation

- Typically, a number associated with each system call
- System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
- Just needs to obey API and understand what OS will do as a result call
- Most details of OS interface hidden from programmer by API Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship





System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
- Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
- Simplest: pass the parameters in *registers*
  - In some cases, may be more parameters than registers
- Parameters stored in a *block*, or table, in memory, and address of block passed as a parameter in a register

This approach taken by Linux and Solaris

- Parameters placed, or *pushed*, onto the *stack* by the program and *popped* off the stack by the operating system
- Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table

1.2.4 Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications
- Protection

Process Control

A running program needs to be able to halt its execution either normally (end) or abnormally (abort). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated. The dump is written to disk and may be examined by a **debugger**—a system program designed to aid the programmer in finding and correcting bugs-—to determine the cause of the problem. Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command. In an interactive system, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to any error.

File Management

We first need to be able to create and delete files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to open it and to use it. We may also read, write, or reposition (rewinding or skipping to the end of the file, for example). Finally, we need to close the file, indicating that we are no longer using it. We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them if necessary. File attributes include the file name, a file type, protection codes, accounting information, and so on.

At least two system calls, get file attribute and set file attribute, are required for this function. Some operating systems provide many more calls, such as calls for file move and copy.

Device Management

A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available. The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, tapes), while others can be thought of as abstract or virtual devices (for example, files). If there are multiple users of the system, the system may require us to first request the device, to ensure exclusive use of it. After we are finished with the device, we release it. These functions are similar to the open and

close system calls for files.

Information Maintenance

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current t I m e and date . Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.

In addition, the operating system keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to reset the process information (get process attributes and set process attributes) .

Communication

There are two common models of inter process communication: the message passing model and the shared-memory model. In the message-passing model, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network. Each computer in a network has a *host name* by which it is commonly known. A host also has a network identifier, such as an IP address. Similarly, each process has a *process name,* and this name is translated into an identifier by which the operating system can refer to the process. The get host id and get processid system calls do this translation. The identifiers are then passed to the general purpose open and close calls provided by the file system or to specific open connection and close connection system calls, depending on the system's model of communication.

In the shared-memory model, processes use shared memory creates and shared memory attaches system calls to create and gain access to regions of memory owned by other processes. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction.

They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by the processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

1.2.5 System Programs

At the lowest level is hardware. Next are the operating system, then the system programs, and finally the application programs. System programs provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls; others are considerably more complex. They can be divided into these categories:

- **File management.** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
- **Status information.** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed

performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a registry, which is used to store and retrieve configuration information.

• **File modification.** Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.

• **Programming-language support.** Compilers, assemblers, debuggers and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system.

• **Program loading and execution.** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.

• **Communications.** These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send electronic-mail messages, to log in remotely, or to transfer files from one machine to another.

In addition to systems programs, most In addition to systems programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such programs include web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games. These programs are known as system utilities or application programs.

1.2.6 Operating-System Structure

Refer above pages

1.2.7 Operating-System Debugging

- Debugging is finding and fixing errors, or bugs
- OS generate log files containing error information
- Failure of an application can generate core dump file capturing memory of the process
- Operating system failure can generate crash dump file containing kernel memory
- Beyond crashes, performance tuning can optimize system performance
- Kernighan's Law: "Debugging is twice as hard as writing the code in the rst place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."
- DTrace tool in Solaris, FreeBSD, Mac OS X allows live instrumentation on production systems
- Probes fire when code is executed, capturing state data and sending it to consumers of those probes

1.2.8 System Boot

The procedure of starting a computer by loading the kernel is known as *booting* the system. On most computer systems, a small piece of code known as the **bootstrap program** or **bootstrap loader** locates the kernel, loads it into main memory, and starts its execution. Some computer systems, such as PCs, use a two-step process in which a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel.

When a CPU receives a reset event—for instance, when it is powered up or rebooted—the instruction register is loaded with a predefined memory location, and execution starts there. At that location is the initial bootstrap program. This program is in the form of **read-only memory (ROM),** because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot be infected by a computer virus.

The bootstrap program can perform a variety of tasks. Usually, one task is to run diagnostics to determine the state of the machine. If the diagnostics pass, the program can continue with the booting steps. It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system.

Some systems—such as cellular phones, PDAs, and game consoles—store the entire operating system in ROM. Storing the operating system in ROM is suitable for small operating systems, simple supporting hardware, and rugged operation. A problem with this approach is that changing the bootstrap code requires changing the ROM hardware chips. Some systems resolve this problem by using **erasable programmable read-only memory** (EPROM), which is read only except when explicitly given a command to become writable. All forms of ROM are also known as **firmware,** since their characteristics fall somewhere between those of hardware and those of software. A problem with firmware in general is that executing code there is slower than executing code in RAM.

Some systems store the operating system in firmware and copy it to RAM for fast execution. A final issue with firmware is that it is relatively expensive, so usually only small amounts are available.

For large operating systems (including most general-purpose operating systems like Windows, Mac OS X, and UNIX) or for systems that change frequently, the bootstrap loader are stored in firmware, and the operating system is on disk. In this case, the bootstrap runs diagnostics and has a bit of code that can read a single block at a fixed location (say block zero) from disk into memory and execute the code from that **boot block.** The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution. More typically, it is simple code (as it fits in a single disk block) and only knows the address on disk and length of the remainder of the bootstrap program. All of the disk-bound bootstrap, and the operating system itself, can be easily changed by writing new versions to disk.

1.3 PROCESSES

1.3.1 Process concepts

Process : A process is a program in execution. A process is more than the program code, which is sometimes known as the **text section.** It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers. A process generally also includes the process **stack,** which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section,** which contains global variables. A process may also include a **heap,** which is memory that is dynamically allocated during process run time.

Structure of a process

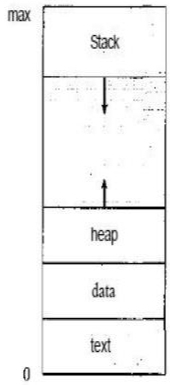


Figure 3.1 Process in memory.

We emphasize that a program by itself is not a process; a program is a *passive* entity, such as a file containing a list of instructions stored on disk (often called an **executable file),** whereas a process is an *active* entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory.

Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog. exe or a. out.)

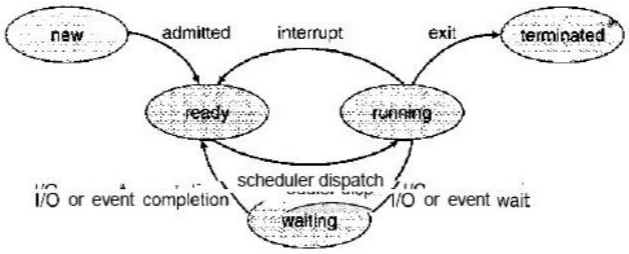


Figure 3.2 Diagram of process state.

Process State

As a process executes, it changes **state.** The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

- **New.** The process is being created.
- **Running.** Instructions are being executed.
- **Waiting.** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready.** The process is waiting to be assigned to a processor.
- **Terminated.** The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are fotind on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be *running* on any processor at any instant.

Process Control Block

Each process is represented in the operating system by a **process control block (PCB)**—also called a *task control block.*

Process state. The state may be new, ready, running, and waiting, halted, and so on.

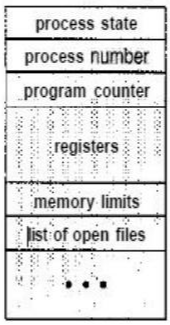


Figure 3.3 Process control block (PCB).

Program counter-The counter indicates the address of the next instruction to be executed for this process.

• CPU **registers-** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.

CPU-scheduling information- This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

Memory-management information- This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system

Accounting information-This information includes the amount of CPU and real time used, time limits, account members, job or process numbers, and so on.

I/O status information-This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

1.3.2 Process Scheduling

The **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU.

As processes enter the system, they are put into a **job queue,** which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue.**

This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

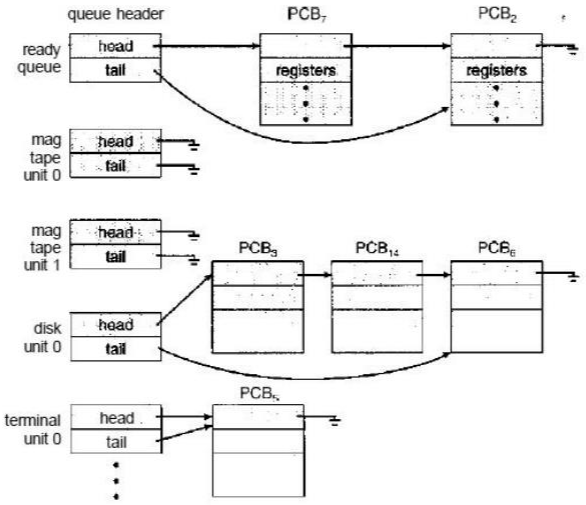


Figure 3.6 The ready queue and various I/O device queues.

Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits there till it is selected for execution, or is **dispatched.** Once the process is allocated the CPU and is executing, one of several events could occur:

• The process could issue an I/O request and then be placed in an I/O queue.

• The process could create a new sub process and wait for the sub process's termination.

• The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Schedulers

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion.

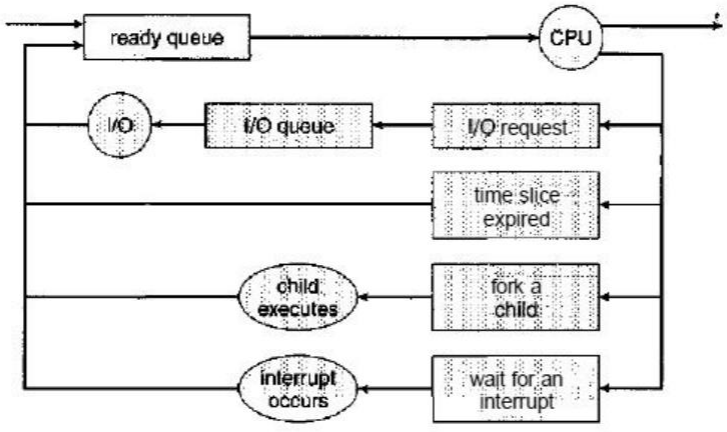


Figure 3.7 Queueing-diagram representation of process scheduling.

The selection process is carried out by the appropriate **scheduler.** The **long-term scheduler,** or **job scheduler,** selects processes from this pool and loads them into memory for execution. The **short-term scheduler, or** CPU **scheduler,** selects from among the processes that are ready to execute and allocates the CPU to one of them.

1.3.3 Operations on Processes

Process Creation

A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a **parent** process, and the new processes are called the **children** of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes.

Most operating systems identify processes according to a unique **process identifier** (or **pid),** which is typically an integer number. These processes are responsible for managing memory and file systems. The sched process also creates the init process, which serves as the root parent process for all user processes.

When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

```
#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 */
execlpf"/bin/Is","Is",NULL);
}
else {/* parent process */
/* parent will wait for the child to complete */
wait(NULL);
printf("Child Complete");
exit (0) ;

}

}
```

In UNIX, as we've seen, each process is identified by its process identifier, which is a unique integer. A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the f ork(), with one difference: The return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent. the exec() system call is used after a fork() system call by one of the two processes to replace the process's memory space with a new program. The exec () system call loads a binary file into memory (destroying the memory image of the program containing the execO system call) and starts its execution.

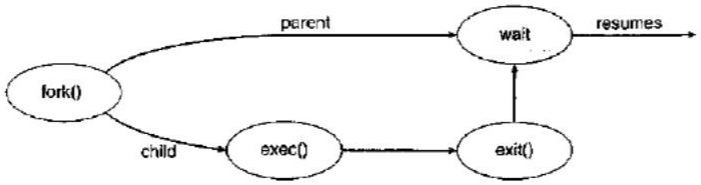


Figure 3.11 Process creation.

Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit () system call. At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deal located by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, TerminateProcessO in Win32). Usually, such a system call can be invoked only by the parent of the process that is to be terminated.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated.
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Consider that, in UNIX, we can terminate a process by using the exit() system call; its parent process may wait for the termination of a child process by using the wait() system call. The wait () system call returns the process identifier of a terminated child so that the parent can tell which of its possibly many children has terminated.

If the parent terminates, however, all its children have assigned as their new parent the init process.

1.3.4 Interprocess Communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is **independent** if it cannot affect or be affected by the other processes executing in the system.

Any process that does not share data with any other process is independent. A process is **cooperating** if **it** can affect or be affected by the other processes executing in the system.

There are several reasons for providing an environment that allows process cooperation:

- **Information sharing.** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

• **Computation speedup.** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
• **Modularity.** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads. • **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication:

(1) shared memory and (2) **message passing.** In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.

Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement than is shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication, as it can be done at memory speeds when within a computer.

Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time consuming task of kernel intervention.

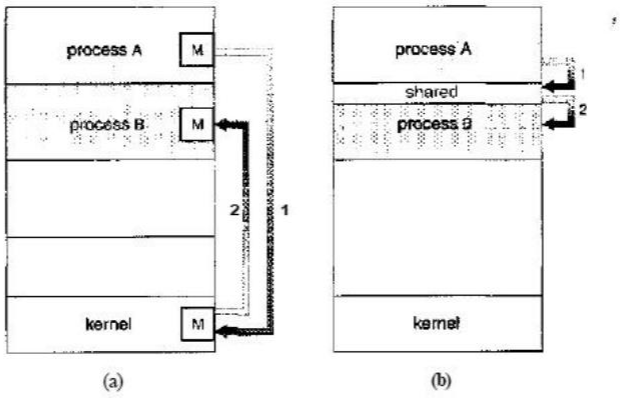


Figure 3.13 Communications models. (a) Message passing. (b) Shared memory.

Shared-Memory Systems Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. The operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Message-Passing Systems The scheme requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility. Message passing provides a mechanism to allow processes to communicate and to synchronize their

actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

A message-passing facility provides at least two operations: send(message) and receive(message). Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating system design.

Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.
Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send.0 and receive() primitives are defined as:
• send(P, message)—Send a message to process P.
• receive (Q, message)—Receive a message from process Q.

• A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
• A link is associated with exactly two processes.
• Between each pair of processes, there exists exactly one link.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions.

Synchronization

Communication between processes takes place through calls to send() and receive () primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous.**
• **Blocking send-** The sending process is blocked until the message is received by the receiving process or by the mailbox.
• **Nonblocking send-** The sending process sends the message and resumes operation.
• **Blocking receive-** The receiver blocks until a message is available.
• **Nonblocking receive-** The receiver retrieves either a valid message or a null.

Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:
• **Zero capacity-** The queue has a maximum length of zero; thus, the link
cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
• **Bounded capacity-** The queue has finite length *n;* thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The links capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

• **Unbounded capacity-** The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

1.3.5 Examples of IPC Systems

An Example: POSIX Shared Memory

Several IPC mechanisms are available for POSIX systems, including shared memory and message passing. A process must first create a shared memory segment using the shmget () system call (shmget () is derived from SHared Memory GET).

The following example illustrates the use of shmget ():
segment_id = shmget(IPCJPRIVATE, size, SJRUSR | SJVVUSR) ;

This first parameter specifies the key (or identifier) of the shared-memory segment. If this is set to IPC-PRIVATE, a new shared-memory segment is created. The second parameter specifies the size (in bytes) of the shared memory segment. Finally, the third parameter identifies the mode, which indicates how the shared-memory segment is to be used—that is, for reading, writing, or both. By setting the mode to SJRUSR | SJVVUSR, we are indicating that the owner may read or write to the shared memory segment.

Processes that wish to access a shared-memory segment must attach it to their address space using the shmat () (SHared Memory ATtach) system call.

The call to shmat () expects three parameters as well. The first is the integer identifier of the shared-memory segment being attached, and the second is a pointer location in memory indicating where the shared memory will be attached. If we pass a value of NULL, the operating system selects the location on the user's behalf. The third parameter identifies a flag that allows the shared memory region to be attached in read-only or read-write mode; by passing a parameter of 0, we allow both reads and writes to the shared region.

The third parameter identifies a mode flag. If set, the mode flag allows the shared-memory region to be attached in read-only mode; if set to 0, the flag allows both reads and writes to the shared region. We attach a region of shared memory using shmat () as follows:

shared_memory = (char *) shmat(id, NULL, 0);

If successful, shmat () returns a pointer to the beginning location in memory where the shared-memory region has been attached.

An Example: Windows XP

The Windows XP operating system is an example of modern design that employs modularity to increase functionality and decrease the time needed to implement new features. Windows XP provides support for multiple operating environments, or *subsystems,* with which application programs communicate via a message-passing mechanism. The application programs can be considered clients of the Windows XP subsystem server.

The message-passing facility in Windows XP is called the **local procedure call (LPC)** facility. The LPC in Windows XP communicates between two processes on the same machine. It is similar to the standard RPC mechanism that is widely used, but it is optimized for and specific to Windows XP. Windows XP uses a port object to establish and maintain a connection between two processes. Every client that calls a subsystem needs a communication channel, which is provided by a port object and is never inherited. Windows XP uses two types of ports: connection ports and communication ports. They are really the same but are given different names according to how they are used. Connection ports are named *objects* and are visible to all processes

The communication works as follows:

• The client opens a handle to the subsystem's connection port object.

• The client sends a connection request.
• The server creates two private communication ports and returns the handle to one of them to the client.
• The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Windows XP uses two types of message-passing techniques over a port that the client specifies when it establishes the channel. The simplest, which is used for small messages, uses the port's message queue as intermediate storage and copies the message from one process to the other. Under this method, messages of up to 256 bytes can be sent. If a client needs to send a larger message, it passes the message through a section object, which sets up a region of shared memory. The client has to
decide when it sets up the channel whether or not it will need to send a large message. If the client determines that it does want to send large messages, it asks for a section object to be created. Similarly, if the server decides that replies will be large, it creates a section object. So that the section object can be used, a small message is sent that contains a pointer and size information about the section object. This method is more complicated than the first method, but it avoids data copying. In both cases, a callback mechanism can be used when either the client or the server cannot respond immediately to a request.

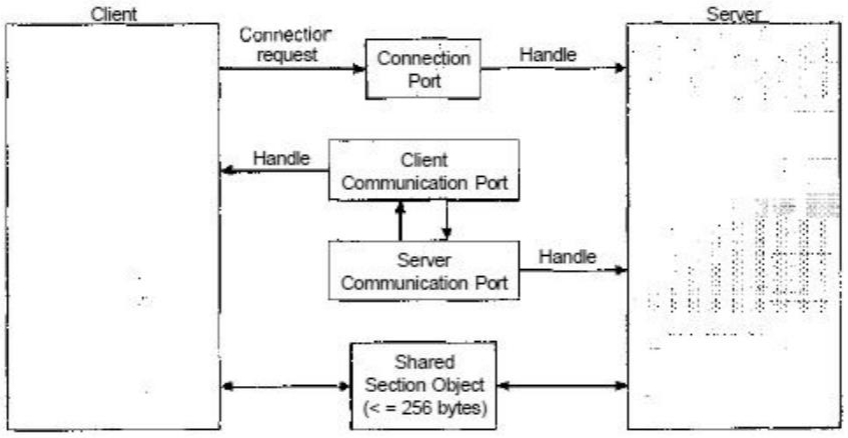


Figure 3.17 Local procedure calls in Windows XP.

MODULE-II

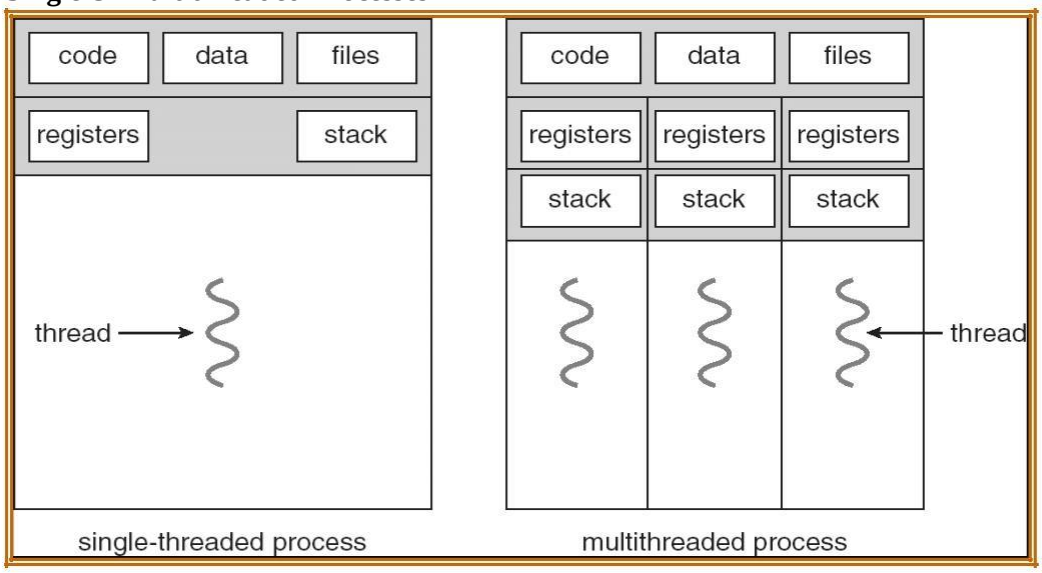
2.1 THREADS

2.1.1 Overview

2.1.2 Multicore Programming

2.1.3 Multithreading Models

Single & Multithreaded Processes



Benefits

- Responsiveness
- Resource Sharing
- Economy
- Utilization of MP Architectures

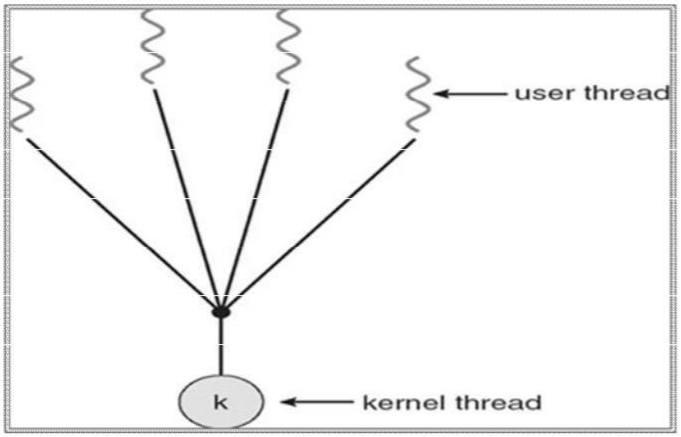
User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
    - POSIX Pthreads
    - Win32 threads
    - Java threads

Multithreading Models

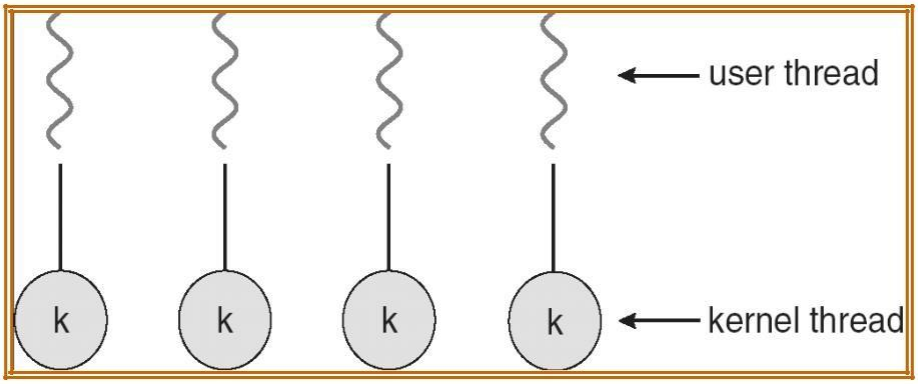
- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One



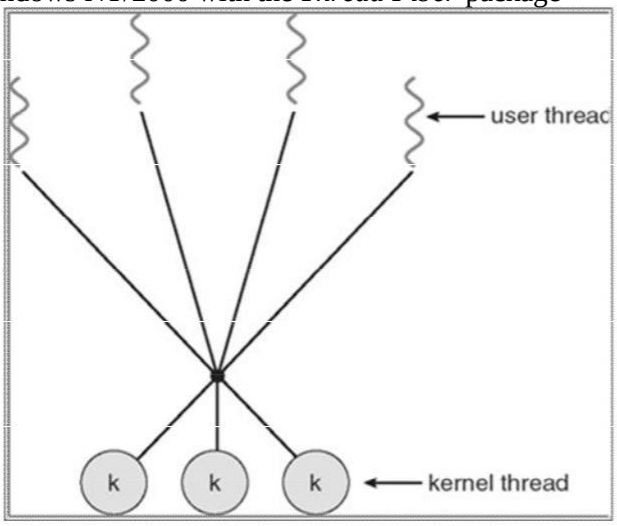
Many-to-One Model

One-to-One



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *Thread Fiber* package



Many-to-Many Model

2.1.4 Thread Libraries

2.1.5 Implicit Threading 2.1.6 Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations

Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Windows XP Threads

- Implements the one-to-one mapping
  - Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
  - The register set, stacks, and private storage area are known as the **context** of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through clone() system call
- clone() allows a child task to share the address space of the parent task (process)

Java Threads

- Java threads are managed by the JVM
  - Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

2.2 Process Synchronization

- Concurrent access to shared data may result in data inconsistency (change in behavior)
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the "producer-consumer" problem that fills all the buffers.
- We can do so by having an integer variable "count" that keeps track of the number of full buffers.
- Initially, count is set to 0.
- It is incremented by the producer after it produces a new buffer.
- It is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {

        /* produce an item and put in next Produced */
                while (count == BUFFER_SIZE)
                        ; // do nothing
                buffer [in] = next Produced;
                in = (in + 1) % BUFFER_SIZE;
                count++;
}
```

Consumer

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

2.2.1 Critical section problem:- A section of code which reads or writes shared data.

Race Condition

- The situation where two or more processes try to access and manipulate the same data and output of the process depends on the orderly execution of those processes is called as Race Condition.
- count++ could be implemented as
  register1 = count
  register1 = register1 + 1
  count = register1
- count-- could be implemented
  as register2 = count
  register2 = register2 -
  1 count = register2
- Consider this execution interleaving with "count = 5" initially:
  - S0: producer execute register1 = count {register1 = 5}
  - S1: producer execute register1 = register1 + 1 {register1 = 6}
  - S2: consumer execute register2 = count {register2 = 5}

- S3: consumer execute register2 = register2 - 1 {register2 = 4}
- S4: producer execute count = register1 {count = 6 }
- S5: consumer execute count = register2 {count = 4}

Requirements for the Solution to Critical-Section Problem

1. **Mutual Exclusion**: - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress:** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting:** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

- To general approaches are used to handle critical sections in operating systems: (1) Preemptive Kernel (2) Non Preemptive Kernel
  - Preemptive Kernel allows a process to be preempted while it is running in kernel mode.
  - Non Preemptive Kernel does not allow a process running in kernel mode to be preempted. (these are free from race conditions)

2.2.2 Peterson's Solution

- It is restricted to two processes that alternates the execution between their critical and remainder sections.
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready!

Note:- Peterson's Solution is a software based solution.

Algorithm for Process P_i

```
while (true) {
  flag[i] = TRUE;
  turn = j;
  while ( flag[j] && turn == j);
      CRITICAL SECTION
  flag[i] = FALSE;
       REMAINDER SECTION
}
```

Fig: structure of process P_i in Peterson's solution

Solution to Critical Section Problem using Locks.

```
do{
        acquire lock

            Critical Section
        release lock
```

Remainder Section

```
}while (True);
```

Note:- Race Conditions are prevented by protecting the critical region by the locks.

2.2.3 Synchronization Hardware

- In general we can provide any solution to critical section problem by using a simple tool called as LOCK where we can prevent the race condition.
- Many systems provide hardware support (hardware instructions available on several systems) for critical section code.
- In UniProcessor hardware environment by disabling interrupts we can solve the critical section problem. So that Currently running code would execute without any preemption .
- But by disabling interrupts on multiprocessor systems is time taking so that it is inefficient compared to UniProcessor system.
- Now a days Modern machines provide special atomic hardware instructions that allow us to either *test memory word and set value* Or *swap contents of two memory words automatically* i.e. done through an uninterruptible unit.

Special Atomic hardware Instructions

- TestAndSet()
- Swap()
- The TestAndSet() Instruction is one kind of special atomic hardware instruction that allow us to test memory and set the value. We can provide Mutual Exclusion by using TestAndSet() instruction.
- Definition:

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

- Mutual Exclusion Implementation with TestAndSet()
- To implement Mutual Exclusion using TestAndSet() we need to declare Shared Boolean variable called as ‘lock’ (initialized to false) .
- Solution:

```
while (true) {
        while ( TestAndSet (&lock ))
                ;   /* do nothing
              //    critical section
        lock = FALSE;
              //      remainder section
}
```

- The Swap() Instruction is another kind of special atomic hardware instruction that allow us to swap the contents of two memory words.
- By using Swap() Instruction we can provide Mutual Exclusion.
- Definition:-

```
void Swap (Boolean *a, Boolean *b)
{
      Boolean temp = *a;
```

```
        *a = *b;
        *b = temp:
    }
```

- Shared Boolean variable called as 'lock' is to be declared to implement Mutual Exclusion in Swap() also, which is initialized to FALSE.

Solution:

```
while (true) {
        key = TRUE;
        while ( key == TRUE)
              Swap (&lock, &key );

                // critical section
         lock = FALSE;
                //      remainder section
    }
```

Note:- Each process has a local Boolean variable called as 'key'.

2.2.4 Mutex Locks

2.2.5 Semaphores

- As it is difficult for the application programmer to use these hardware instructions, to overcome this difficulty we use the synchronization tool called as Semaphore (that does not require busy waiting)
- Semaphore *S* _ integer variable, apart from this initialization we can access this only through two standard atomic operations called as wait() and signal().
- Originally the wait() and signal() operations are termed as P() and V() respectively. Which are termed from the Dutch words "proberen" and "verhogen".
- *The definition for wait() is as follows*:

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

- *The definition for signal() is as follows*:

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

- All the modifications to the integer value of the semaphore in the wait() and signal() atomic operations must be executed indivisibly. i.e. when one process changes the semaphore value, no other process will change the same semaphore value simultaneously.
- *Usage of semaphore*:- we have two types of semaphores
  - _ Counting semaphore
  - _ Binary Semaphore.
- The value of the Counting Semaphore can ranges over an unrestricted domain.
- The value of the Binary Semaphore can ranges between 0 and 1 only.
- In some systems the Binary Semaphore is called as Mutex locks, because, as they are locks to provide the mutual exclusion.
- We can use the Binary Semaphore to deal with critical section problem for multiple processes.
- Counting Semaphores are used to control the access of given resource each of which consists of some finite no. of instances. This counting semaphore is initialized to number of resources available.

- The process that wish to use a resource must performs the wait() operation (count is decremented)
- The process that releases a resource must performs the signal() operation (count is incremented)
- When the count for the semaphore is 0 means that all the resources are being used by some processes. Otherwise resources are available for the processes to allocate .
- When a process is currently using a resource means that it blocks the resource until the count becomes > 0.
- For example:
  - Let us assume that there are two processes p_0 and p_1 which consists of two statements s_0 & s_1 respectively.
  - Also assume that these two processes are running concurrently such that process p_1 executes the statement s_1 only after process p_0 executes the statement s_0.
  - Let us assume the process p_0 & p_1 share the same semaphore called as "synch" which is initialized to 0 by inserting the statements

```
S0;
Signal (synch);
```

in process p_0 and the statements wait

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

in process p_1 .

Implementation:

- The main disadvantage of the semaphore definition is, it requires the busy waiting.
- Because when one process is in critical section and if another process needs to enter in to the critical section must have to loop in the entry code continuously.

- To overcome the need of the busy waiting we have to modify the definition of wait() and signal() operations. i.e. when a process executes wait() operation and finds that it is not positive then it must wait.
- Instead of engaging the busy wait, the process block itself so that there will be a chance to the CPU to select another process for execution. It is done by block() operation.
  - Blocked processes are placed in waiting queue.
- Later the process that has already been blocked by itself is restarted by using wakeup() operation, so that the process will move from waiting state to ready state.
  - Blocked processes that are placed in waiting queue are now placed into ready queue.
- To implement the semaphore with no busy waiting we need to define the semaphore of the wait() and signal() operation by using the 'C' Struct. Which is as follows: typedef

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

  - i.e. each semaphore has an integer value stored in the variable "value" and the list of processes list.
  - When a process perform the wait() operation on the semaphore then it will adds list of processes to the list .
  - When a process perform the signal() operation on the semaphore then it removes the processes from the list.

Semaphore Implementation with no Busy waiting

Implementation of wait: (definition of wait with no busy waiting) wait

```
(S){
        value--;
        if (value < 0) {
```

add this process to waiting queue
block(); }

}

Implementation of signal: (definition of signal with no busy waiting)

```
Signal (S){
        value++;
        if (value <= 0) {
            remove a process P from the waiting queue
            wakeup(P); }
}
```

Deadlock and Starvation

- The implementation of semaphore with waiting queue may result in the situation where two or more processes are waiting for an event is called as Deadlocked.
- To illustrate this, let us assume two processes P_0 and P_1 each accessing two semaphores S and Q which are initialized to 1 :-

```
P0                  P1
  wait (S);                  wait (Q);
   wait (Q);                 wait (S);
.                   .
.                   .
.                   .
      signal (S); signal (Q); signal (Q); signal (S);
```

- Now process P_0 executes *wait(S)* and P_1 executes *wait(Q)*, assume that P_0 wants to execute *wait(Q)* and P_1 executes *wait(S)*. But it is possible only after process P_1 executes the *signal(Q)* and P_0 executes *signal(S)*.
- Starvation or indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

2.2.6 Classical Problems of Synchronization

- **Bounded-Buffer Problem**
- **Readers and Writers Problem**
- **Dining-Philosophers Problem**

Bounded-Buffer Problem

- Let us assume *N* buffers, each can hold only one item.
- Semaphore mutex initialized to the value 1 which is used to provide mutual exclusion.
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.
- Semaphore full and empty are used to count the number of buffers.
- The structure of the producer process

```
while (true) {
        //  produce an
    item wait (empty);
    wait (mutex);
        //  add the item to the buffer
     signal (mutex);
     signal (full);
}
```

The structure of the consumer process

```
while (true) {
     wait (full);
     wait (mutex);
```

```
            // remove an item from buffer
        signal (mutex);
        signal (empty);

            // consume the removed item
    }
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set, do not perform any updates
  - Writers – can both read and write the data set (perform the updates).
- If two readers read the shared data simultaneously, there will be no problem. If both a reader(s) and writer share the same data simultaneously then there will be a problem.
- In the solution of reader-writer problem, the reader process share the following data structures: Semaphore Mutex, wrt;
int readcount;
- Where → Semaphore mutex is initialized to 1.

→ Semaphore wrt is initialized to 1.
→ Integer readcount is initialized to 0.

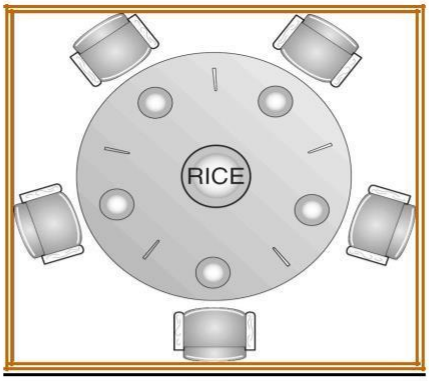
The structure of a writer process

```
while (true) {
        wait (wrt) ;

            //    writing is
        performed signal (wrt) ;
}
```

The structure of a reader process

```
while (true) {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1) wait (wrt) ;
        signal (mutex)

            // reading is performed
        wait (mutex) ; readcount -
        - ;
        if (readcount == 0) signal (wrt) ;
        signal (mutex) ;
}
```

Dining-Philosophers Problem



- **Shared data**
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem

- **The structure of Philosopher *i*:**

```
While (true) {
    wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

            // eat
        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

       // think
}
```

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex) → Case 1
  - wait (mutex) … wait (mutex) → Case 2
  - Omitting of wait (mutex) or signal (mutex) (or both) → Case 3
- As the semaphores used incorrectly as above may results the timing errors.
- Case 1 → Several processes may execute in critical section by violating the mutual exclusion requirement.
- Case 2 → Dead lock will occur.
- Case 3 → either mutual exclusion is violated or dead lock will occur
- To deal with such type of errors, researchers have developed high-level language constructs.
- One type of high-level language constructs that is to be used to deal with the above type of errors is → the *Monitor* type.

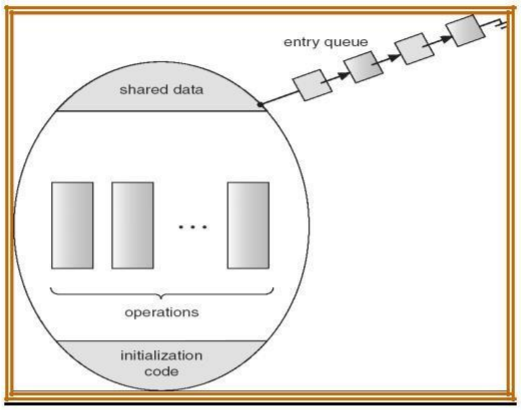
2.2.7 Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization.
- A procedure can access only those variables that are declared in a monitor and formal parameters .
- Only one process may be active within the monitor at a time

Syntax of the monitor :-

```
monitor monitor-name
{
        // shared variable declarations
        procedure P1 (…) { …. }
                …
        procedure Pn (…) {……}
    Initialization code ( ….) { … }
                …
        }
```

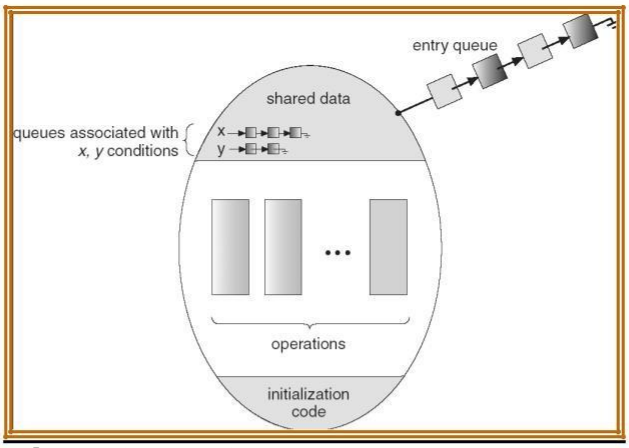
Schematic view of a Monitor



Condition Variables

- Synchronization scheme is not effective within the monitors.
- A programmer who needs to write the synchronization scheme can define one or more variables of type *Condition*
- condition x, y;
- The only operations that can be invoked on a condition variable are wait() and signal().
- The operations are
  - x.wait () – a process that request an operation is suspended until another process invokes x.signal ()
  - x.signal () – resumes only one suspended processes (if any) that invoked x.wait ()
- Now suppose that when x.signal() operation is invoked by a process P, there i s a suspended process Q associated with condition x. if Q is allowed to resume its execution, the signaling process P must wait. Else both P and Q would be active simultaneously with in a monitor.
- There are two possibilities
  - 1) signal and wait:- P either waits until Q leaves the monitor or waits for another condition.
  - 2) signal and continue:- Q either waits Until P leaves the monitor or waits for another condition.

Monitor with Condition Variables



Solution to Dining Philosophers using Monitors

```
monitor DP
  {
        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 left and right
               neighbors 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;
        }
}
```

Each philosopher *i* invokes the operations pickup()
and putdown() in the following sequence:

```
dp.pickup (i)
    EAT
dp.putdown (i)
```

Monitor Implementation Using Semaphores

Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;     // (initially  = 0)
```

```
int next-count = 0;
```

Each procedure *F* will be replaced by

```
wait(mutex);
    …
        body of F;
    …
if (next-count > 0)
        signal(next)
else
        signal(mutex);
```

Mutual exclusion within a monitor is ensured.

Monitor Implementation

For each condition variable *x*, we have:

```
semaphore x-sem; // (initially = 0)
int x-count = 0;
```

The operation x.wait can be implemented as:

```
x-count++;
if (next-count > 0)
        signal(next);
else
        signal(mutex);
wait(x-sem);
x-count--;
```

The operation x.signal can be implemented as:

```
if (x-count > 0) {
        next-count++;
        signal(x-sem);
        wait(next);
        next-count--;
}
```

2.2.8 Synchronization Examples

- **Windows XP**
- **Linux**

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

Linux Synchronization

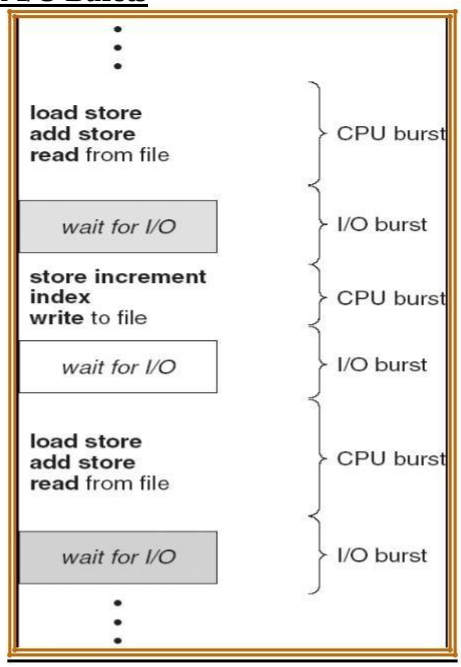
- Linux:
  - disables interrupts to implement short critical sections
- Linux provides:
  - semaphores
  - spin locks

2.2.9 Alternative approaches

2.3 CPU Scheduling

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- CPU burst distribution

Alternating Sequence of CPU & I/O Bursts



CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready
4. Terminates

- Scheduling under 1 and 4 is *nonpreemptive*
- All other scheduling is *preemptive*

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running

2.3.1 Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – No. of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Optimization Criteria

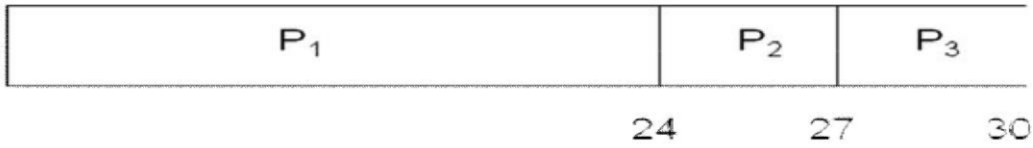
- Max CPU utilization
- Max throughput

- Min turnaround time
- Min waiting time
- Min response time

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

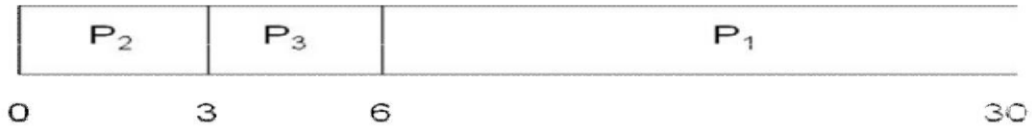
Suppose that the processes arrive in the order: P_1 , P_2 , P_3
The Gantt Chart for the schedule is:



- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

- Suppose that the processes arrive in the order

- The Gantt chart for the schedule is:



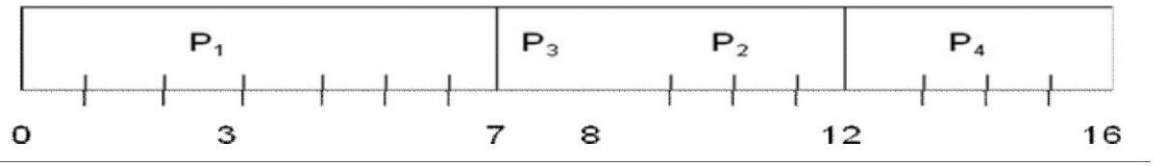
- Waiting time for P_1 = 6; P_2 = 0; P_3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- *Convoy effect* short process behind long process

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF)
- SJF is optimal – gives minimum average waiting time for a given set of processes

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

SJF (non-preemptive)

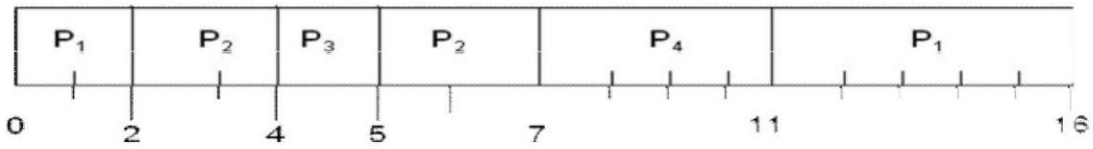


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- **SJF (preemptive)**



- Average waiting time = (9 + 1 + 0 +2)/4 = 3

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Non preemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging - as time progresses increase the priority of the process (means Aging increases the priority of the processes so that to terminate in finite amount of time).

Round Robin (RR)

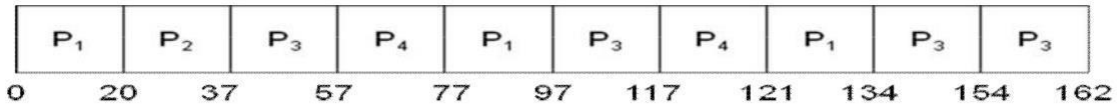
- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Performance
  - q large ⇒ FIFO
  - q small ⇒ q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68

P_4 24

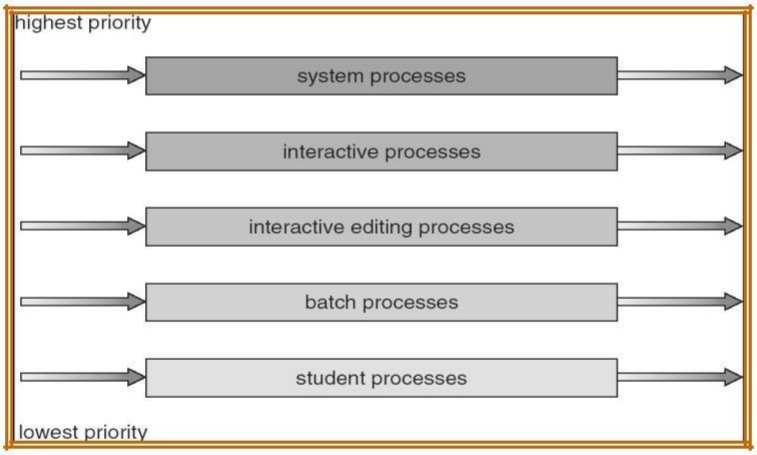
The Gantt chart is:



- Typically, higher average turnaround than SJF, but better *response*

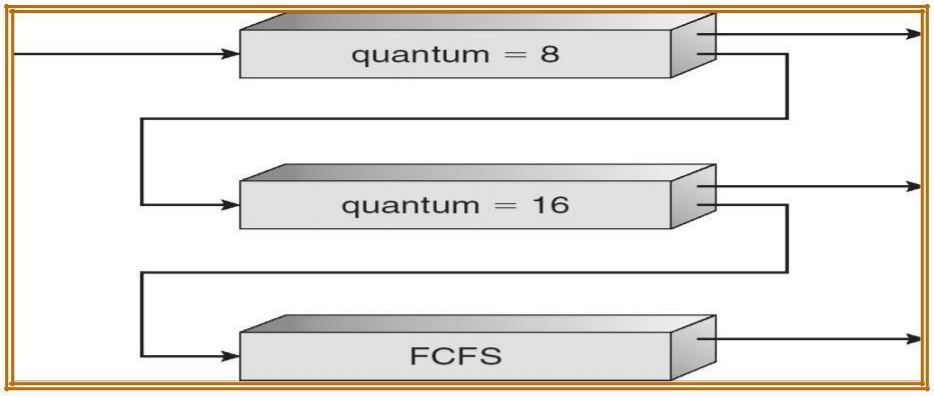
Multilevel Queue Scheduling

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS



Multilevel Feedback Queue Scheduling

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service



2.3.2 Scheduling Algorithms

2.3.3 Thread Scheduling

- Local Scheduling – How the threads library decides which thread to put onto an available LWP
- Global Scheduling – How the kernel decides which kernel thread to run next

2.3.4 Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- *Homogeneous processors* within a multiprocessor
- *Load sharing*
- *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing

Operating System Examples

Windows XP Priorities

	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

Linux Scheduling

- Two algorithms: time-sharing and real-time
- Time-sharing
    - Prioritized credit-based – process with most credits is scheduled next
    - Credit subtracted when timer interrupt occurs
    - When credit = 0, another process chosen
    - When all processes have credit = 0, recrediting occurs
        - Based on factors including priority and history
- Real-time
    - Soft real-time
    - Posix.1b compliant – two classes
        - FCFS and RR

- Highest priority process always runs first

Java Thread Scheduling

- JVM Uses a Preemptive, Priority-Based Scheduling Algorithm
- FIFO Queue is Used if There Are Multiple Threads With the Same

Priority JVM Schedules a Thread to Run When:

1. The Currently Running Thread Exits the Runnable State
2. A Higher Priority Thread Enters the Runnable State

* Note _ the JVM Does Not Specify Whether Threads are Time-Sliced or Not

3.3 DEAD LOCKS

3.3.1. 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. Example: system calls open( ), malloc( ), new( ), and request( ). 2. Use - The process uses the resource.

Example: prints to the printer or reads from the file.

3. Release - The process relinquishes the resource. so that it becomes available for other processes.

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.) 3.3.2. Deadlock Characterization Necessary Conditions: There are four conditions that are necessary to achieve deadlock:

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. 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. 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. 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.)

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:

*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 resourceallocation graph indicates the possibility of a deadlock, but does not guarantee one. Consider, for example, Figures 7.3 and 7.4 below:

3.3.3. Methods for Handling Deadlocks

Generally speaking there are three ways of handling deadlocks: Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state. Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected. 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.

3.3.4. Deadlock Prevention Deadlocks can be prevented by preventing at least one of the four required conditions: 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. 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:

i. 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. ii. 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. iii. Either of the methods described above can lead to starvation if a process requires one or more popular resources. No Preemption Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible. i. 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 reacquire the old resources along with the new resources in a single request, similar to the previous discussion. ii. 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. iii. 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. Circular Wait i. 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. ii. In other words, in order to request resource Rj, a process must first release all Ri such that i >= j. iii. One big challenge in this scheme is determining the relative ordering of the different resources

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. Safe State i. 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. ii. 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.) iii. 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.)

Fig: 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

i: What happens to the above table if process P2 requests and is granted one more tape drive? ii. 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.

Resource-Allocation Graph Algorithm i. If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs. ii. 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. iii. 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.) iv. 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. v. 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:

Fig: Resource allocation graph for dead lock avoidance The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.

Fig: An Unsafe State in a Resource Allocation Graph

Banker's Algorithm i. 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. ii. 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.) iii. 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. iv. 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. Data Structures for the Banker's Algorithm Let n = number of processes, and m = number of resources types. N Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available n Max: n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rjn Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rjn Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task Need [i,j] = Max[i,j] – Allocation [i,j] Safety Algorithm 1. Let Work and Finish be vectors of length m and n, respectively. Initialize: Work = Available Finish [i] = false for i = 0, 1, …, n- 1

2. Find and i such that both:

(a) Finish [i] = false

(b) Needi <= Work

If no such i exists, go to step 4

3. 3. Work = Work + Allocationi

Finish[i] = true go to step 2

4. If Finish [i] == true for all i, then the system is in a safe state

Resource-Request Algorithm for Process Pi

Request = request vector for process Pi . If Requesti [j] = k then process Pi wants k instances of resource type Rj

1. If Requesti <= Needi

go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If Requesti £ Available, go to step 3. Otherwise Pi must wait, since resources are not available

3. Pretend to allocate requested resources to Pi by modifying the state as follows:

Available = Available – Request;

Allocationi = Allocationi + Requesti;

Needi = Needi – Requesti;

lIf safe Þ the resources are allocated to Pi lIf unsafe Þ Pi must wait, and the old resource-allocation state is restored An Illustrative Example Consider the following situation:

The system is in a safe state since the sequence < P1, P3, P4, P2, P0> satisfies safety criteria. Example: P1 Request (1,0,2) Check that Request £ Available (that is, (1,0,2) <= (3,3,2) Þ true Executing safety algorithm shows that sequence < P1, P3, P4, P0, P2> satisfies safety requirement. 3.3.6. Deadlock Detection i. If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow. ii. 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. 6.1 Single Instance of Each Resource Type i. 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. ii. 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. iii. 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.

As before, cycles in the wait-for graph indicate deadlocks. This algorithm must maintain the wait-for graph, and periodically search it for cycles. Several Instances of a Resource Type Available:

A vector of length m indicates the number of available resources of each type. Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process. Request: An n x m matrix indicates the current request of each process. If Request [ij] = k, then process Pi is requesting k more instances of resource type. Rj . Detection Algorithm

1. Let Work and Finish be vectors of length m and n, respectively Initialize:

(a) Work = Available(b) For i = 1,2, …, n,

if Allocationi != 0, then

Finish[i] = false;

otherwise,

Finish[i] = true

2. Find an index i such that both:

(a) Finish[i] == false

(b) Requesti <=Work

If no such i exists, go to step 4

3. 3. Work = Work + Allocationi

Finish[i] = true go to step 2

4. If Finish[i] == false, for some i, 1 <= i <=n, then the system is in deadlock state.

Moreover, if Finish[i] == false, then Pi is deadlocked Algorithm requires an order of O(m x n2) operations to detect whether the system is in deadlocked state. Example of Detection Algorithm

Five processes P0 through P4; three resource types A (7 instances), B (2 instances), and C (6 instances)

Snapshot at time T0:

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?

Detection-Algorithm Usage i. 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.) ii. 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. Recovery From Deadlock There are three basic approaches to recovery from deadlock: i. Inform the system operator, and allow him/her to take manual intervention. ii. Terminate one or more processes involved in the deadlock iii. Preempt resources. Process Termination 1. Two basic approaches, both of which recover resources allocated to terminated processes:

i. 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. ii. Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step. 2. In the latter case there are many factors that can go into deciding which processes to terminate next: i. Process priorities. ii. How long the process has been running, and how close it is to finishing. iii. How many and what type of resources is the process holding. (Are they easy to preempt and restore?) iv. How many more resources does the process need to complete. v. How many processes will need to be terminated vi. Whether the process is interactive or batch.

Resource Preemption When preempting resources to relieve deadlock, there are three important issues to be addressed: Selecting a victim - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above. 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.) 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.

MEMORY MANAGEMENT

Memory management is concerned with managing the primary memory. Memory consists of array of bytes or words each with their own address. The instructions are fetched from the memory by the CPU based on the value program counter. Functions of memory management:

- Keeping track of status of each memory location.
- Determining the allocation policy.
- Memory allocation technique.
- De-allocation technique.

Address Binding: Programs are stored on the secondary storage disks as binary executable files. When the programs are to be executed they are brought in to the main memory and placed within a process. The collection of processes on the disk waiting to enter the main memory forms the input queue. One of the processes which are to be executed is fetched from the queue and placed in the main memory. During the execution it fetches instruction and data from main memory. After the process terminates it returns back the memory space. During execution the process will go through different steps and in each step the address is represented in different ways. In source program the address is symbolic. The compiler converts the symbolic address to re-locatable address. The loader will convert this re-locatable address to absolute address. Binding of instructions and data can be done at any step along the way: Compile time:-If we know whether the process resides in memory then absolute code can be generated. If the static address changes then it is necessary to re-compile the code from the beginning. Load time:-If the compiler doesn't know whether the process resides in memory then it generates the re-locatable code. In this the binding is delayed until the load time. Execution time:-If the process is moved during its execution from one memory segment to another then the binding is delayed until run time. Special hardware is used for this. Most of the general purpose operating system uses this method.

Logical versus physical address:

The address generated by the CPU is called logical address or virtual address. The address seen by the memory unit i.e., the one loaded in to the memory register is called the physical address. Compile time and load time address binding methods generate some logical and physical address. The execution time addressing binding generate different logical and physical address. Set of logical address space generated by the programs is the logical address space. Set of physical address corresponding to these logical addresses is the physical address space. The mapping of virtual address to physical address during run time is done by the hardware device called memory management unit

(MMU). The base register is also called re-location register. Value of the re-location register is added to every address generated by the user process at the time it is sent to memory.

Dynamic re-location using a re-location registers

The above figure shows that dynamic re-location which implies mapping from virtual addresses space to physical address space and is performed by the hardware at run time. Re-location is performed by the hardware and is invisible to the user dynamic relocation makes it possible to move a partially executed process from one area of memory to another without affecting.

Dynamic Loading:

For a process to be executed it should be loaded in to the physical memory. The size of the process is limited to the size of the physical memory. Dynamic loading is used to obtain better memory utilization. In dynamic loading the routine or procedure will not be loaded until it is called. Whenever a routine is called, the calling routine first checks whether the called routine is already loaded or not. If it is not loaded it cause the loader to load the desired program in to the memory and updates the programs address table to indicate the change and control is passed to newly called routine.

Advantage: Gives better memory utilization. Unused routine is never loaded. Do not need special operating system support. This method is useful when large amount of codes are needed to handle in frequently occurring cases.

Dynamic linking and Shared libraries:

Some operating system supports only the static linking. In dynamic linking only the main program is loaded in to the memory. If the main program requests a procedure, the procedure is loaded and the link is established at the time of references. This linking is postponed until the execution time. With dynamic linking a “stub” is used in the image of each library referenced routine. A “stub” is a piece of code which is used to indicate how to locate the appropriate memory resident library routine or how to load library if the routine is not already present. When “stub” is executed it checks whether the routine is present is memory or not. If not it loads the routine in to the memory. This feature can be used to update libraries i.e., library is replaced by a new version and all the programs can make use of this library. More than one version of the library can be loaded in memory at a time and each program uses its version of the library. Only the programs that are compiled with the new version are

affected by the changes incorporated in it. Other programs linked before new version is installed will continue using older libraries this type of system is called “shared library”.

3.1.1 Swapping

Swapping is a technique of temporarily removing inactive programs from the memory of the system. A process can be swapped temporarily out of the memory to a backing store and then brought back in to the memory for continuing the execution. This process is called swapping. Eg:-In a multi-programming environment with a round robin CPU scheduling whenever the time quantum expires then the process that has just finished is swapped out and a new process swaps in to the memory for execution.

A variation of swap is priority based scheduling. When a low priority is executing and if a high priority process arrives then a low priority will be swapped out and high priority is allowed for execution. This process is also called as Roll out and Roll in.

Normally the process which is swapped out will be swapped back to the same memory space that is occupied previously. This depends upon address binding. If the binding is done at load time, then the process is moved to same memory location. If the binding is done at run time, then the process is moved to different memory location. This is because the physical address is computed during run time. Swapping requires backing store and it should be large enough to accommodate the copies of all memory images. The system maintains a ready queue consisting of all the processes whose memory images are on the backing store or in memory that are ready to run. Swapping is constant by other factors: To swap a process, it should be completely idle. A process may be waiting for an i/o operation. If the i/o is asynchronously accessing the user memory for i/o buffers, then the process cannot be swapped.

3.1.2 Contiguous memory allocation:

One of the simplest method for memory allocation is to divide memory in to several fixed partition. Each partition contains exactly one process. The degree of multi-programming depends on the number of partitions. In multiple partition method, when a partition is free, process is selected from the input queue and is loaded in to free partition of memory. When process terminates, the memory partition becomes available for another process. Batch OS uses the fixed size partition scheme.

The OS keeps a table indicating which part of the memory is free and is occupied. When the process enters the system it will be loaded in to the input queue. The OS keeps track of the memory requirement of each process and the amount of memory available and determines which process to allocate the memory. When a process requests, the OS searches for large hole for this process, hole is a large block of free memory available. If the hole is too large it is split in to two. One part is allocated to the requesting process and other is returned to the set of holes. The set of holes are searched to determine which hole is best to allocate. There are three strategies to select a free hole:

 First bit:-Allocates first hole that is big enough. This algorithm scans memory from the beginning and selects the first available block that is large enough to hold the process. Best bit:-It chooses the hole i.e., closest in size to the request. It allocates the smallest hole i.e., big enough to hold the process. Worst fit:-It allocates the largest hole to the process request. It searches for the largest hole in the entire list. Operating Systems

First fit and best fit are the most popular algorithms for dynamic memory allocation. First fit is generally faster. Best fit searches for the entire list to find the smallest hole i.e., large enough. Worst fit reduces the rate of production of smallest holes. All these algorithms suffer from fragmentation.

Memory Protection:

Memory protection means protecting the OS from user process and protecting process from one another. Memory protection is provided by using a re-location register, with a limit register. Re-location register contains the values of smallest physical address and limit register contains range of logical addresses.

(Re-location = 100040 and limit = 74600). The logical address must be less than the limit register; the MMU maps the logical address dynamically by adding the value in re-location register. When the CPU scheduler selects a process for execution, the dispatcher loads the re-location and limit register with correct values as a part of context switch. Since every address generated by the CPU is checked against these register we can protect the OS and other users programs and data from being modified.

Fragmentation: Memory fragmentation can be of two types: Internal Fragmentation External Fragmentation Internal Fragmentation there is wasted space internal to a portion due to the fact that block of data loaded is smaller than the partition. Eg:-If there is a block of 50kb and if the process requests 40kb and if the block is allocated to the process then there will be 10kb of memory left.

External Fragmentation exists when there is enough memory space exists to satisfy the request, but it not contiguous i.e., storage is fragmented in to large number of small holes. External Fragmentation may be either minor or a major problem. One solution for over-coming external fragmentation is compaction. The goal is to move all the free memory together to form a large block. Compaction is not possible always. If the relocation is static and is done at load time then compaction is not possible. Compaction is possible if the re-location is dynamic and done at execution time. Another possible solution to the external fragmentation problem is to permit the logical address space of a process to be non-contiguous, thus allowing the process to be allocated physical memory whenever the latter is available.

3.1.3 Segmentation Most users do not think memory as a linear array of bytes rather the users thinks memory as a collection of variable sized segments which are dedicated to a particular use such as code, data, stack, heap etc. A logical address is a collection of segments. Each segment has a name and length. The address specifies both the segment name and the offset within the segments. The users specify address by using two quantities: a segment name and an offset. For simplicity the segments are numbered and referred by a segment number. So the logical address consists of <segment number, offset>.

Hardware support: We must define an implementation to map 2D user defined address in to 1D physical address. This mapping is affected by a segment table. Each entry in the segment table has a segment base and segment limit. The segment base contains the starting physical address where the segment resides and limit specifies the length of the segment.

The use of segment table is shown in the above figure: Logical address consists of two parts: segment number‘s’ and an offset‘d’ to that segment. The segment number is used as an index to segment table. The offset’d’ must be in between 0 and limit, if not an error is reported to OS. If legal the offset is added to the base to generate the actual physical address. The segment table is an array of base limit register pairs.

Protection and Sharing: A particular advantage of segmentation is the association of protection with the segments. The memory mapping hardware will check the protection bits associated with each segment table entry to prevent illegal access to memory like attempts to write in to read-only segment. Another advantage of segmentation involves the sharing of code or data. Each process has a segment table associated with it. Segments are shared when the entries in the segment tables of two different processes points to same physical location. Sharing occurs at the segment table. Any information can

be shared at the segment level. Several segments can be shared so a program consisting of several segments can be shared. We can also share parts of a program. Advantages: Eliminates fragmentation. x Provides virtual growth. Allows dynamic segment growth. Assist dynamic linking. Segmentation is visible.

Differences between segmentation and paging:- Segmentation:

• Program is divided in to variable sized segments. x User is responsible for dividing the program in to segments. • Segmentation is slower than paging. • Visible to user. • Eliminates internal fragmentation. • Suffers from external fragmentation. • Process or user segment number, offset to calculate absolute address.

Paging:

• Programs are divided in to fixed size pages. • Division is performed by the OS. • Paging is faster than segmentation. • Invisible to user. • Suffers from internal fragmentation. • No external fragmentation. • Process or user page number, offset to calculate absolute address.

3.1.4 Paging

Paging is a memory management scheme that permits the physical address space of a process to be non-contiguous. Support for paging is handled by hardware. It is used to avoid external fragmentation. Paging avoids the considerable problem of fitting the varying sized memory chunks on to the backing store. When some code or date residing in main memory need to be swapped out, space must be found on backing store.

Basic Method: Physical memory is broken in to fixed sized blocks called frames (f). Logical memory is broken in to blocks of same size called pages (p). When a process is to be executed its pages are loaded in to available frames from backing store. The blocking store is also divided in to fixed-sized blocks of same size as memory frames. The following figure shows paging hardware:

Logical address generated by the CPU is divided in to two parts: page number (p) and page offset (d). The page number (p) is used as index to the page table. The page table contains base address of each page in physical memory. This base address is combined with the page offset to define the physical memory i.e., sent to the memory unit. The page size is defined by the hardware. The size of

a power of 2, varying between 512 bytes and 10Mb per page. If the size of logical address space is 2^m address unit and page size is 2^n, then high order m-n designates the page number and n low order bits represents page offset.

Eg:-To show how to map logical memory in to physical memory consider a page size of 4 bytes and physical memory of 32 bytes (8 pages). Logical address 0 is page 0 and offset 0. Page 0 is in frame 5. The logical address 0 maps to physical address 20. [(5*4) + 0]. logical address 3 is page 0 and offset 3 maps to physical address 23 [(5*4) + 3]. Logical address 4 is page 1 and offset 0 and page 1 is mapped to frame 6. So logical address 4 maps to physical address 24 [(6*4) + 0]. Logical address 13 is page 3 and offset 1 and page 3 is mapped to frame 2. So logical address 13 maps to physical address 9 [(2*4) + 1]. Hardware Support for Paging: The hardware implementation of the page table can be done in several ways: The simplest method is that the page table is implemented as a set of dedicated registers. These registers must be built with very high speed logic for making paging address translation. Every accessed memory must go through paging map. The use of registers for page table is satisfactory if the page table is small.

If the page table is large then the use of registers is not visible. So the page table is kept in the main memory and a page table base register [PTBR] points to the page table. Changing the page table requires only one register which reduces the context switching type. The problem with this approach is the time required to access memory location. To access a location [i] first we have to index the page table using PTBR offset. It gives the frame number which is combined with the page offset to produce the actual address. Thus we need two memory accesses for a byte.

The only solution is to use special, fast, lookup hardware cache called translation look aside buffer [TLB] or associative register. LB is built with associative register with high speed memory. Each register contains two paths a key and a value.

When an associative register is presented with an item, it is compared with all the key values, if found the corresponding value field is return and searching is fast. TLB is used with the page table as follows: TLB contains only few page table entries. When a logical address is generated by the CPU, its page number along with the frame number is added to TLB. If the page number is found its frame memory is used to access the actual memory. If the page number is not in the TLB (TLB miss) the memory reference to the page table is made. When the frame number is obtained use can use it to access the memory. If the TLB is full of entries the OS must select anyone for replacement. Each time a new page table is selected the TLB must be flushed [erased] to ensure that next executing

process do not use wrong information. The percentage of time that a page number is found in the TLB is called HIT ratio. Protection:

Memory protection in paged environment is done by protection bits that are associated with each frame these bits are kept in page table. x One bit can define a page to be read-write or read-only. To find the correct frame number every reference to the memory should go through page table. At the same time physical address is computed. The protection bits can be checked to verify that no writers are made to read-only page. Any attempt to write in to read-only page causes a hardware trap to the OS. This approach can be used to provide protection to read-only, read-write or execute-only pages. One more bit is generally added to each entry in the page table: a valid-invalid bit.

A valid bit indicates that associated page is in the processes logical address space and thus it is a legal or valid page. If the bit is invalid, it indicates the page is not in the processes logical addressed space and illegal. Illegal addresses are trapped by using the valid-invalid bit. The OS sets this bit for each page to allow or disallow accesses to that page.

3.1.5 Structure Of Page Table

a. Hierarchical paging:

Recent computer system support a large logical address apace from 2^32 to 2^64. In this system the page table becomes large. So it is very difficult to allocate contiguous main memory for page table. One simple solution to this problem is to divide page table in to smaller pieces. There are several ways to accomplish this division.

One way is to use two-level paging algorithm in which the page table itself is also paged. Eg:-In a 32 bit machine with page size of 4kb. A logical address is divided in to a page number consisting of 20 bits and a page offset of 12 bit. The page table is further divided since the page table is paged, the page number is further divided in to 10 bit page number and a 10 bit offset.

b. Hashed page table: Hashed page table handles the address space larger than 32 bit. The virtual page number is used as hashed value. Linked list is used in the hash table which contains a list of elements that hash to the same location. Each element in the hash table contains the following three fields: Virtual page number x Mapped page frame value x Pointer to the next element in the linked list Working: Virtual page number is taken from virtual address. Virtual page number is hashed in

to hash table. Virtual page number is compared with the first element of linked list. Both the values are matched, that value is (page frame) used for calculating the physical address. If not match then entire linked list is searched for matching virtual page number. Clustered pages are similar to hash table but one difference is that each entity in the hash table refer to several pages.

c. Inverted Page Tables: Since the address spaces have grown to 64 bits, the traditional page tables become a problem. Even with two level page tables. The table can be too large to handle. An inverted page table has only entry for each page in memory. Each entry consisted of virtual address of the page stored in that read-only location with information about the process that owns that page. Each virtual address in the Inverted page table consists of triple <process-id , page number , offset >. The inverted page table entry is a pair <process-id , page number>. When a memory reference is made, the part of virtual address i.e., <process-id , page number> is presented in to memory sub-system. The inverted page table is searched for a match. If a match is found at entry I then the physical address <i , offset> is generated. If no match is found then an illegal address access has been attempted. This scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. If the whole table is to be searched it takes too long.

Advantage:

• Eliminates fragmentation. • Support high degree of multiprogramming. • Increases memory and processor utilization. • Compaction overhead required for the re-locatable partition scheme is also eliminated.

Disadvantage:

• Page address mapping hardware increases the cost of the computer. • Memory must be used to store the various tables like page tables, memory map table etc. • Some memory will still be unused if the number of available block is not sufficient for the address space of the jobs to be run.

d. Shared Pages:

Another advantage of paging is the possibility of sharing common code. This is useful in timesharing environment. Eg:-Consider a system with 40 users, each executing a text editor. If the text editor is

of 150k and data space is 50k, we need 8000k for 40 users. If the code is reentrant it can be shared. Consider the following figure

If the code is reentrant then it never changes during execution. Thus two or more processes can execute same code at the same time. Each process has its own copy of registers and the data of two processes will vary. Only one copy of the editor is kept in physical memory. Each user's page table maps to same physical copy of editor but date pages are mapped to different frames. So to support 40 users we need only one copy of editor (150k) plus 40 copies of 50k of data space i.e., only 2150k instead of 8000k.

3.2 Virtual Memory Preceding sections talked about how to avoid memory fragmentation by breaking process memory requirements down into smaller bites (pages), and storing the pages noncontiguously 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: o Error handling code is not needed unless that specific error occurs, some of which are quite rare. o Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice. o 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: o Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer. o 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. o Less I/O is needed for swapping processes in and out of RAM, speeding things up. Figure below shows the general layout of virtual memory, which can be much larger than physical memory:

Fig: Diagram showing virtual memory that is larger than physical memory

 Figure below shows virtual address space, which is the programmer's 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 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.

Fig: Virtual address space Virtual memory also allows the sharing of files and memory by multiple processes, with several benefits: o System libraries can be shared by mapping them into the virtual address space of more than one process. o Processes can also share virtual memory by mapping the

same block of memory to more than one process. o Process pages can be shared during a fork( ) system call, eliminating the need to copy all of the pages of the original (parent) process.

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.

Fig: Transfer of a paged memory to contiguous disk space

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.

Fig: 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.)

Fig: 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. The hardware necessary to support virtual memory is the same as for paging and swapping: A page table and secondary memory. 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. 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. 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.

Fig: Before process 1 modifies page C

Fig: 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-fillon-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. 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.

\

Fig: Ned for page replacement

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.

Fig: 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: i. Only the page number that was accessed is relevant. The offset within that page does not affect paging operations. ii. 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.) Example: 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 below Figure:

Fig: Graph of page faults versus number of frames

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:

Fig: 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:

Fig: Page-fault curve for FIFO replacement on a reference string 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.

Fig: Optimal page-replacement algorithm 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.) Below figure illustrates LRU for our sample string, yielding 12 page faults, (as compared to 15 for FIFO and 9 for OPT.)

Fig: 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 pagereplacement 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.

Fig: Use of a stack to record the most recent page references

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. 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. o At periodic intervals (clock interrupts), the OS takes over, and right-shifts each of the reference bytes by one bit. o The high-order (leftmost) bit is then filled in with the current value of the reference bit, and the reference bits are cleared. o 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. 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. o When a page must be replaced, the page table is scanned in a FIFO (circular queue) manner. o If a page is found with its reference bit not set, then that page is selected as the next victim. o 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.

Fig: Second-chance (clock) page-replacement algorithm 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. 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: o 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 rightshift the counters periodically, yielding a time-decaying average reference count. o 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. Page-Buffering Algorithms There are a number of page-buffering algorithms that can be used in

conjunction with the aforementioned 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. 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. Allocation of Frames We said earlier that there were two important tasks in virtual memory management: a pagereplacement strategy and a frame-allocation strategy. This section covers the second part of that pair.

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. 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.) 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. 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.) 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. 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.)

Fig: 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.) 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 below Figure:

Fig: 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: o 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. o Every time the timer goes off, the current reference bit is copied to one of the two historical bits, and then cleared. o 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. o Finer resolution can be achieved with more historical bits and a more frequent timer, at the expense of greater overhead. Page-Fault

Frequency A more direct approach is to recognize that what we really want to control is the pagefault 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.)

Fig: Page-fault frequency

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. 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 time killer 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 below Figure. Copy-on-write is supported, and mutual exclusion techniques may be needed to avoid synchronization problems.

Fig: 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.) 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 below Figure

Fig: Shared memory in Windows using memory-mapped I/O 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: o 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. o Interrupt Driven. The device generates an interrupt when it either has another byte of data to deliver or is ready to receive another byte.

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. 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. o Free lists are maintained for every size block. o 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.) o 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. o See the (annotated) Figure below for an example.

Fig: Buddy System Allocation

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.

Fig: Slab Allocation

Other Considerations 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. 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. 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. 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.) Program Structure Consider a pair of nested loops to access every element in a 1024 x 1024 twodimensional 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. 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 :The reason why frames used for I/O must be in memory. Operating-System Examples (Optional) 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.

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 threshold, and if it does, then the page out 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.

Fig: 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 page out runs 4 times per second to check if memory has fallen below lotsfree. However if it falls below desfree, then page out 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 page out 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. 3

MODULE 4

Mass-Storage Structure

Overview of Mass-Storage Structure

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.

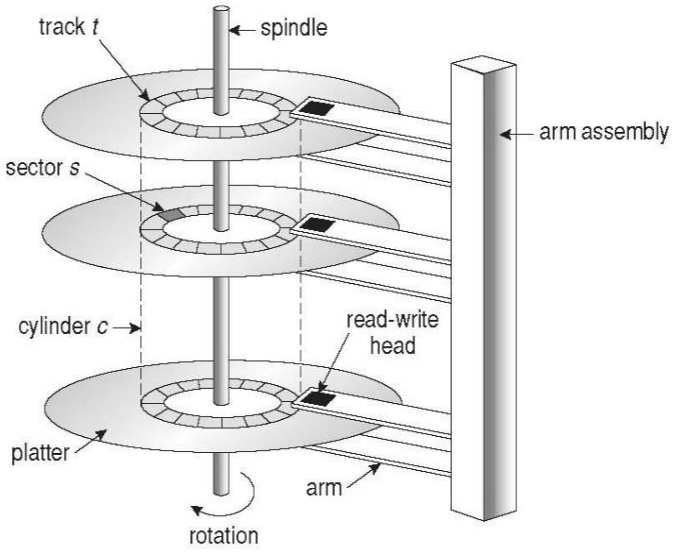


Fig: Moving-head disk mechanism

- 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 seeks 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.

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 use 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 file system meta-data, e.g. directory and anode 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.

Magnetic Tapes

- 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.

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.)

Disk Attachment

Disk drives can be attached either directly to a particular host (a local disk) or to a network.

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.
- 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.)

Network-Attached Storage

- Network attached storage connects storage devices to computers using a remote procedure call, RPC, interface, typically with something like NFS file system 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.

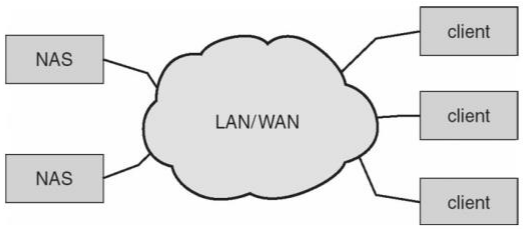


Fig: Network-attached storage.

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.

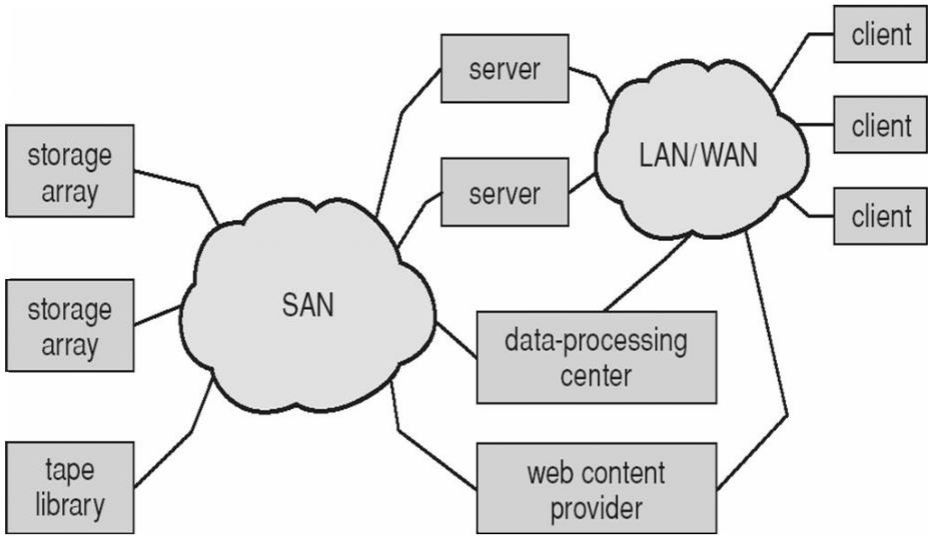


Fig: Storage-area network.

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.

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:

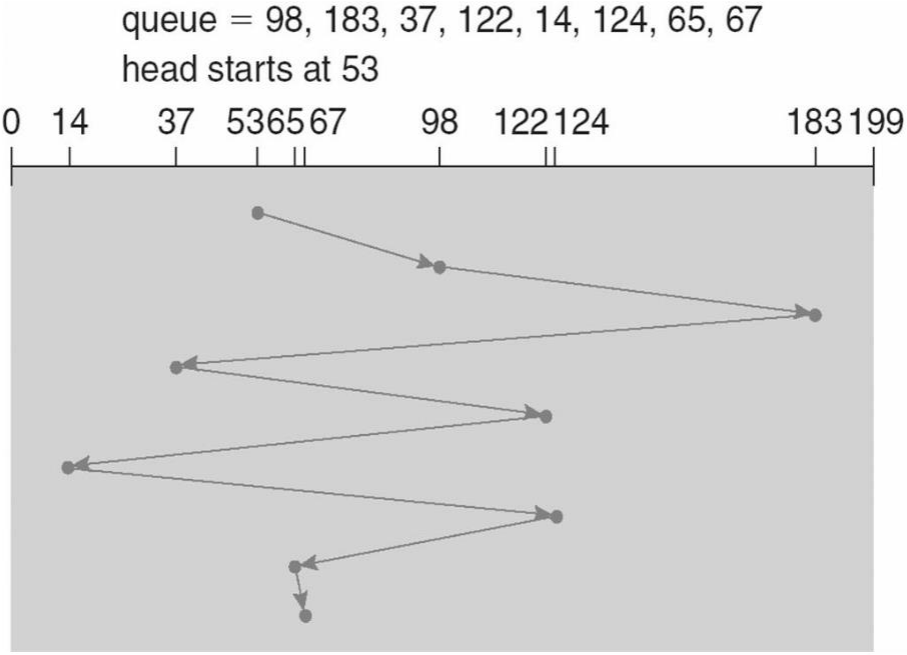


Fig: FCFS disk scheduling.

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.

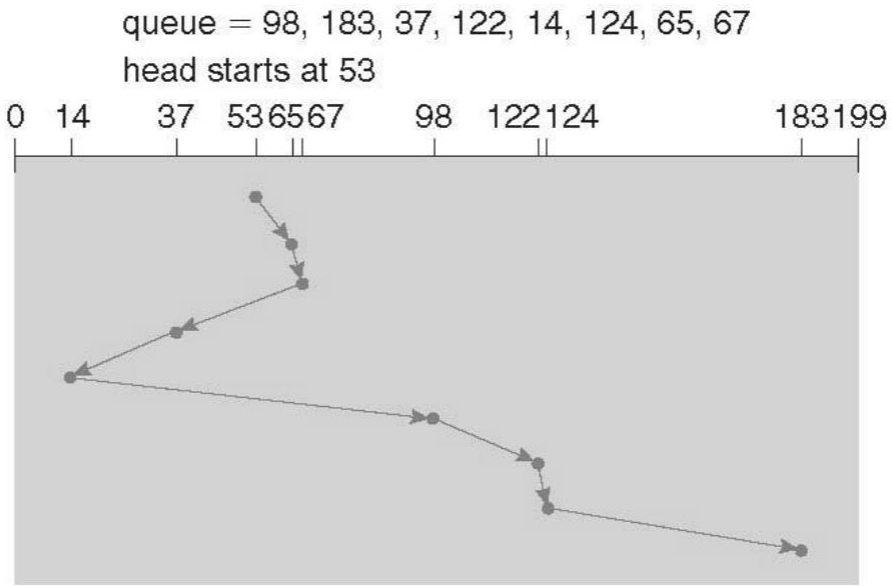


Fig: SSTF disk scheduling.

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.

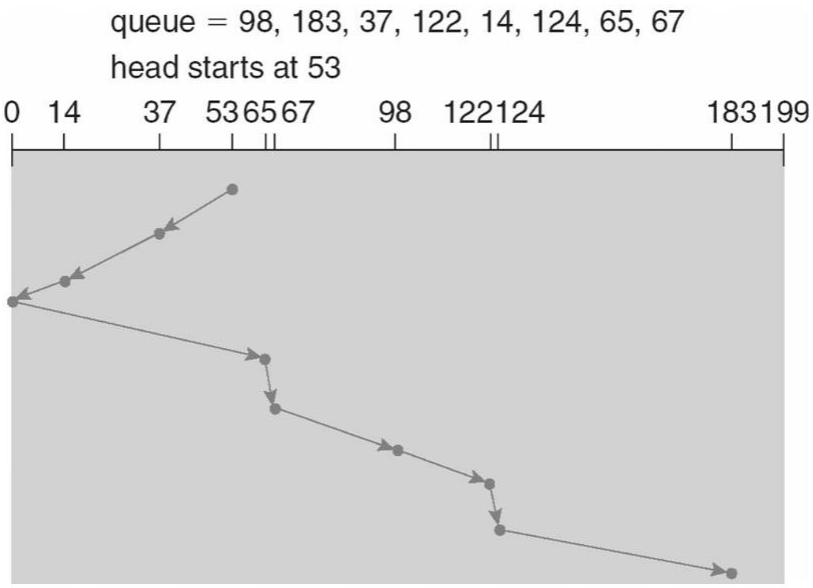


Fig: SCAN disk scheduling.

- 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.

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:

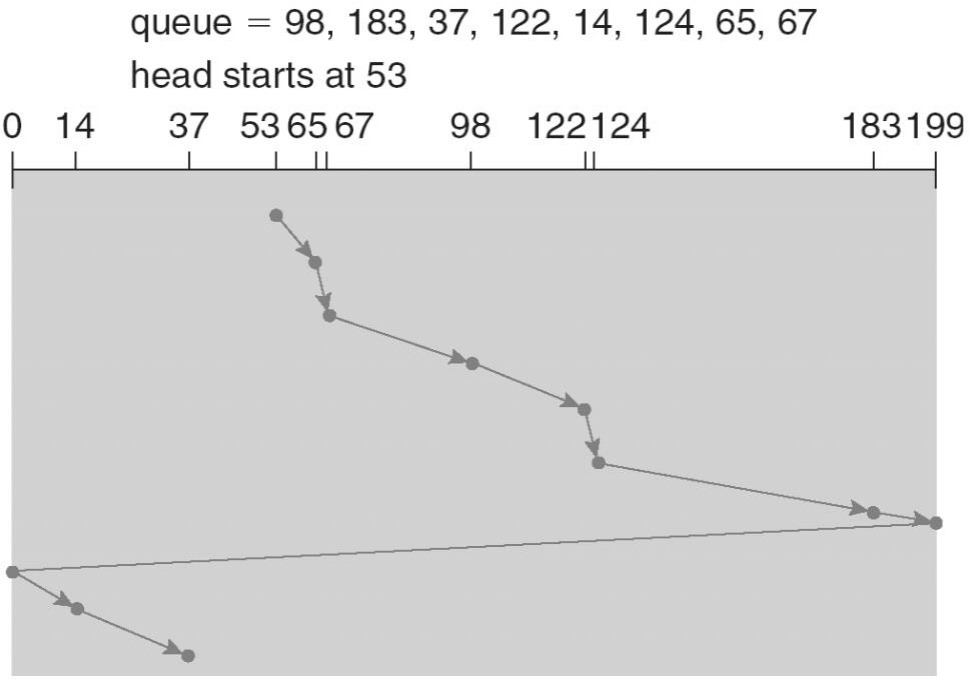


Fig: C-SCAN disk scheduling.

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:

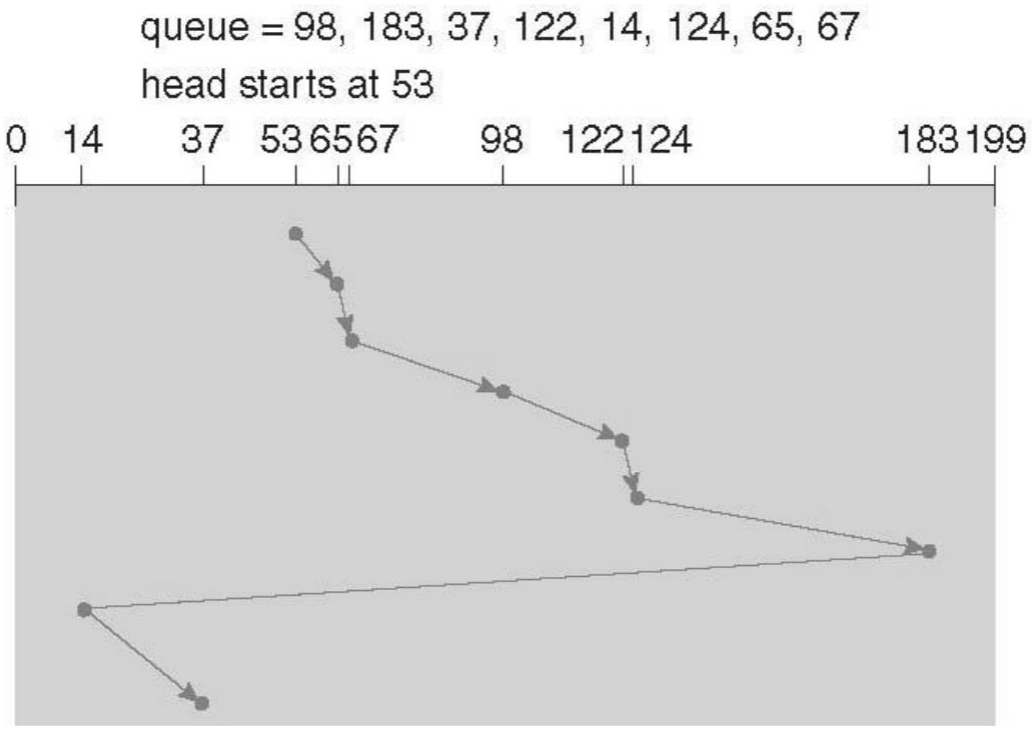


Fig: C-LOOK disk scheduling.

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 file system 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 they 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.

Disk Management

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 file systems 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 file system. (Disk partitions which are to be used as raw devices are not logically formatted. This saves the overhead and disk space of the file system structure, but requires that the application program manage its own disk storage requirements.)

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.

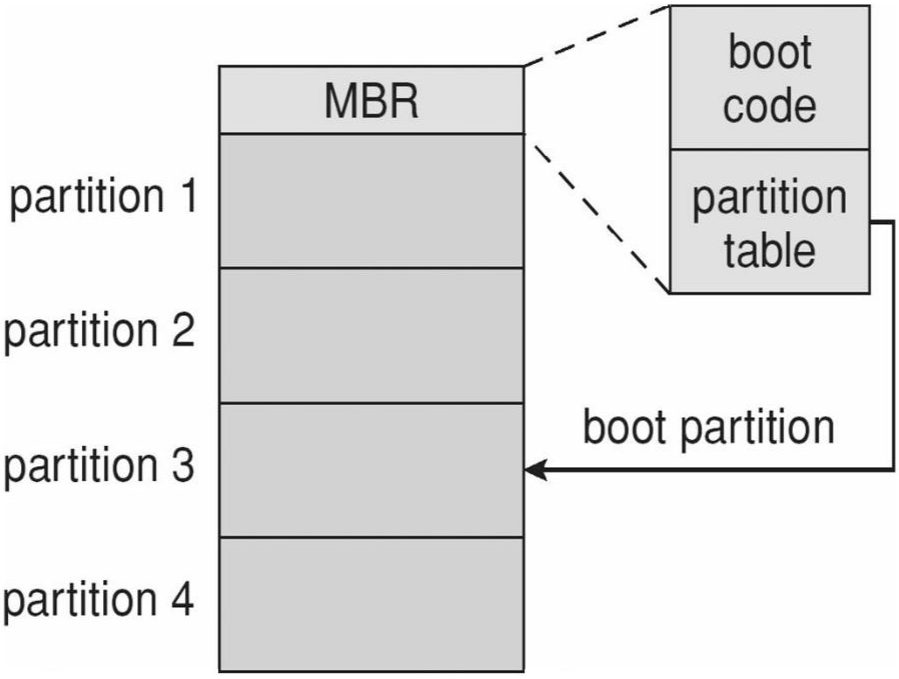


Fig: Booting from disk in Windows 2000.

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.

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.

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.

Swap-Space Location

Swap space can be physically located in one of two locations:

- As a large file which is part of the regular file system. 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.

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 file system 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.)

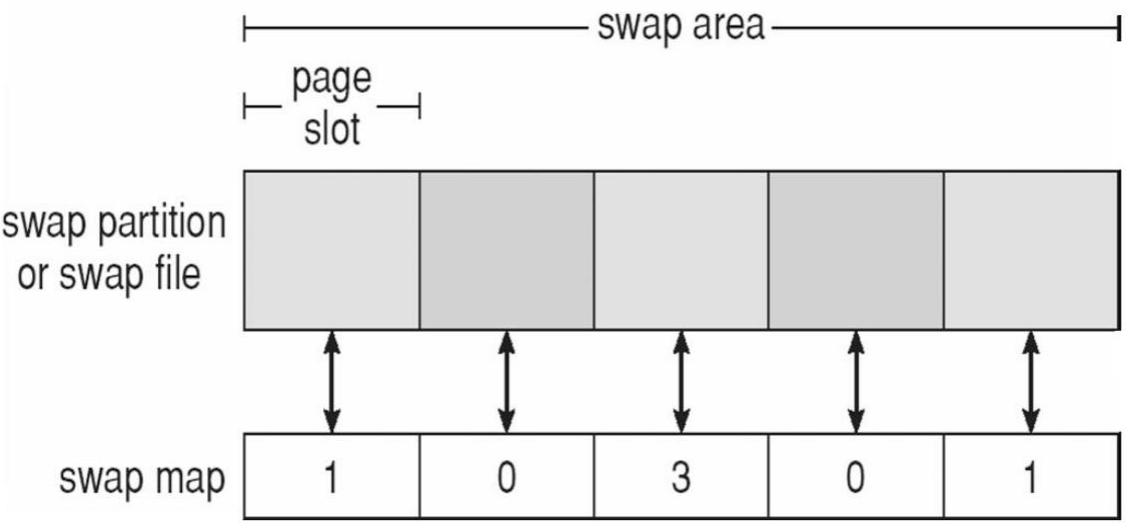


Fig: The data structures for swapping on Linux systems.

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.

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 a MUCH lower probability than for a single disk is 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.

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 file system 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 file systems are accessed in clusters of physical blocks. Other striping possibilities exist, with block-level striping being the most common.

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.)
  - **Raid Level 0** - This level includes striping only, with no mirroring.
  - **Raid Level 1** - This level includes mirroring only, no striping.
  - **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.)

- **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.
- **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.
- **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.
- **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.

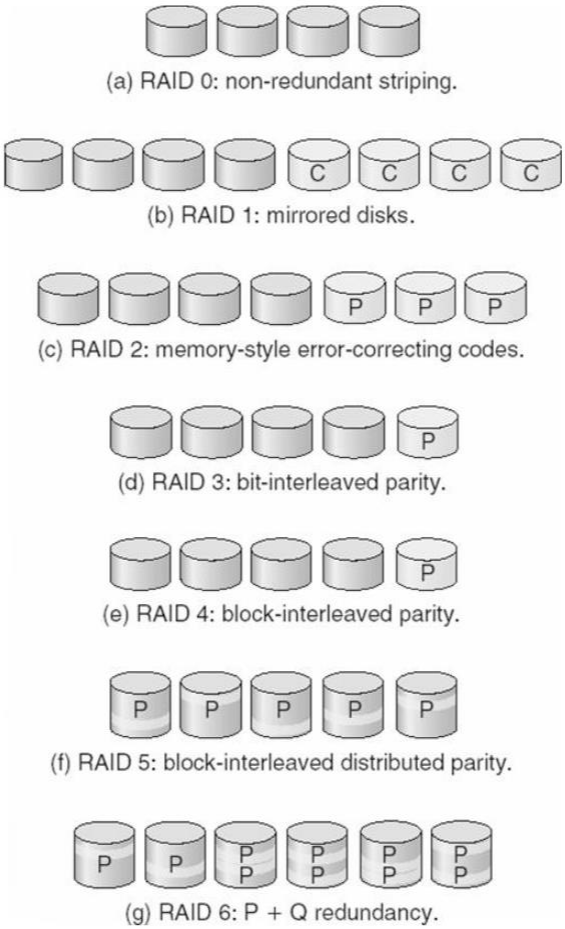


Fig: RAID levels.

- 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.

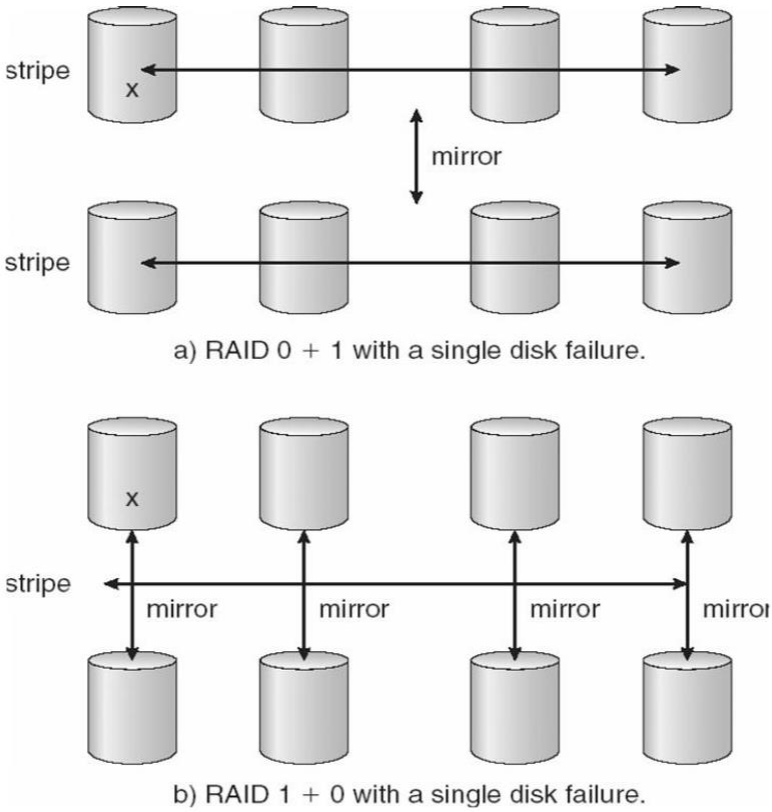


Fig: RAID 0 + 1 and 1 + 0

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.

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.

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.

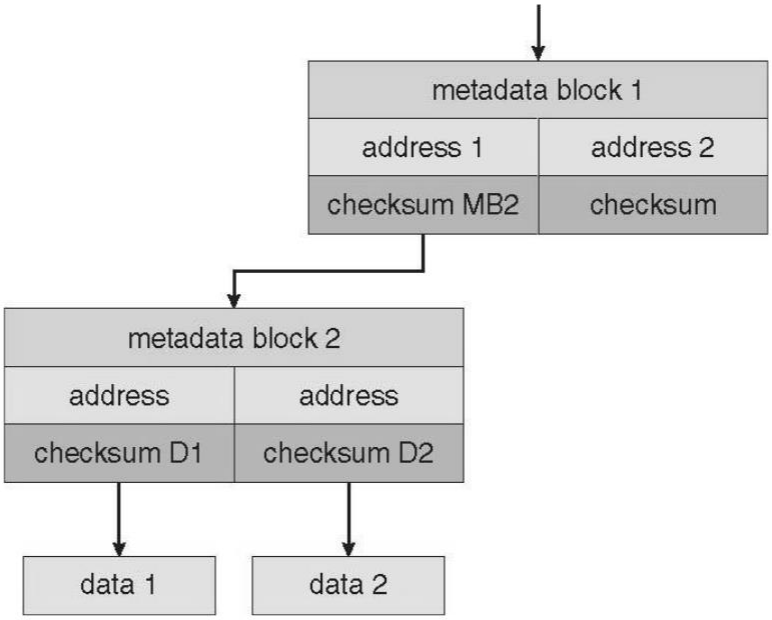


Fig: ZFS checksums all metadata and data.

- Another problem with traditional file systems is that the sizes are fixed, and relatively difficult to change. Where RAID sets are involved it becomes even harder to adjust file system sizes, because a file system cannot span across multiple file systems.
- ZFS solves these problems by pooling RAID sets, and by dynamically allocating space to file systems as needed. File system sizes can be limited by quotas, and space can also be reserved to guarantee that a file system will be able to grow later, but these parameters can be changed at any time by the file system's owner. Otherwise file systems grow and shrink dynamically as needed.

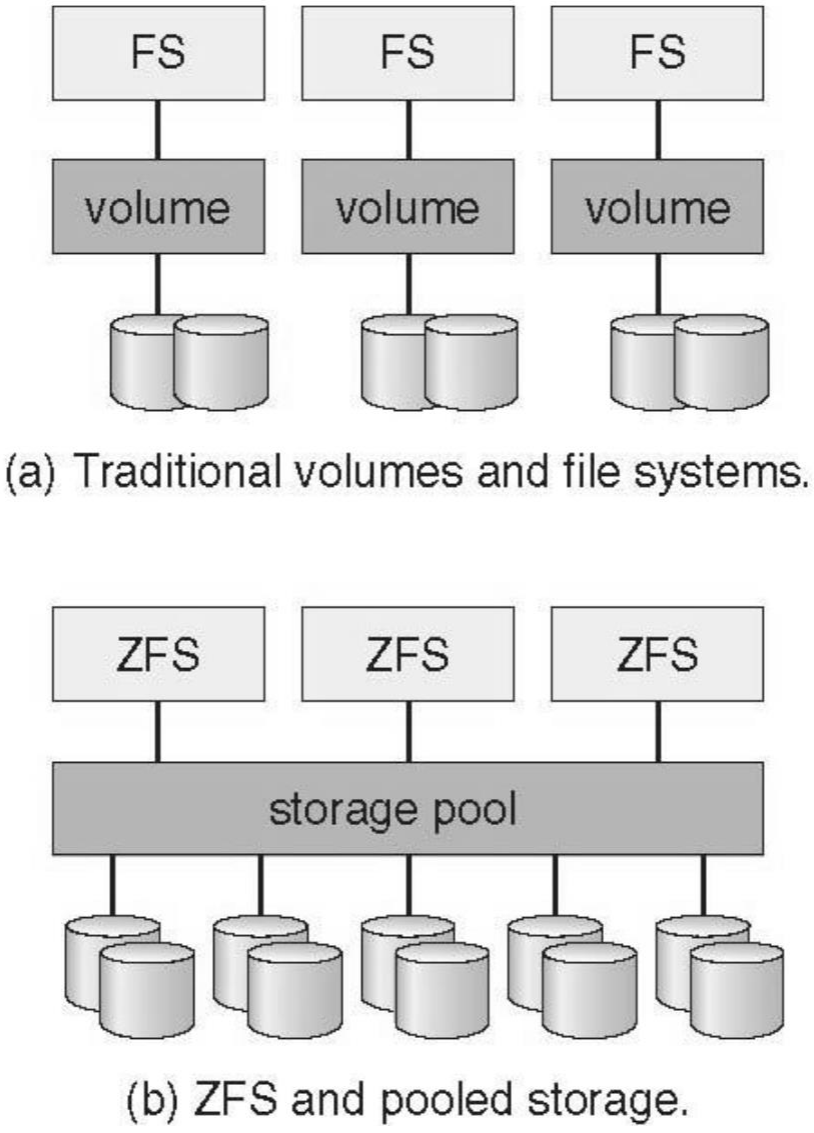


Fig: (a) Traditional volumes and file systems. (b) a ZFS pool and file systems.

Stable-Storage Implementation

- 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:
    - The data is successfully and completely written.
    - The data is partially written, but not completely. The last block written may be garbled.
    - 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:
    - Write the data to the first physical block.
    - After step 1 had completed, then write the data to the second physical block.
    - 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.

4.2 FILE SYSTEM INTERFACE

File Concept

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

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 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, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Fig: 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)

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.

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.

Access Methods

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. o **rewind**
  - **skip n records** - May or may not be supported. N may be limited to positive numbers, or may be limited to +/- 1.

Fig: Sequential File Access

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;

Fig: Simulation of sequential access on a direct-access file

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.

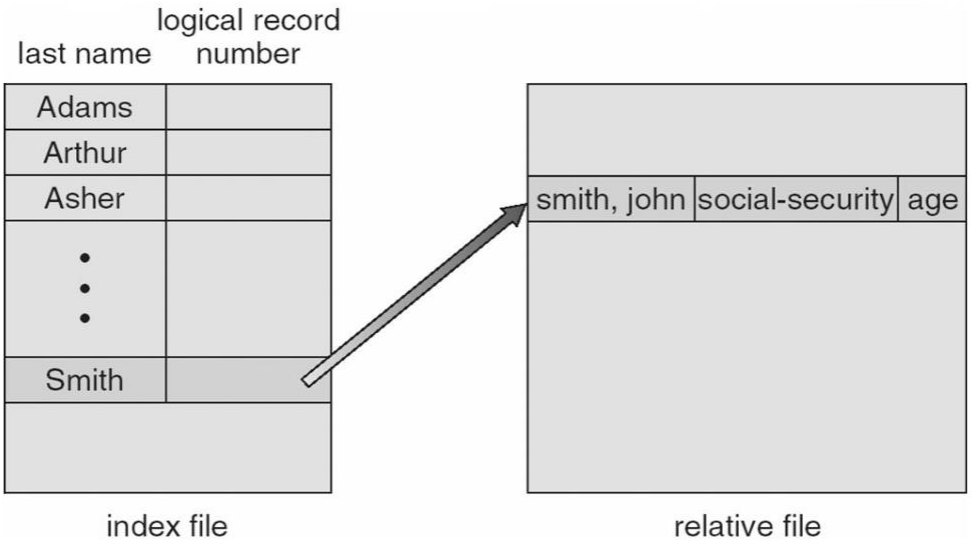


Fig: Example of index and relative files.

Directory Structure

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 file system. (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 file system spanning the physical disks.

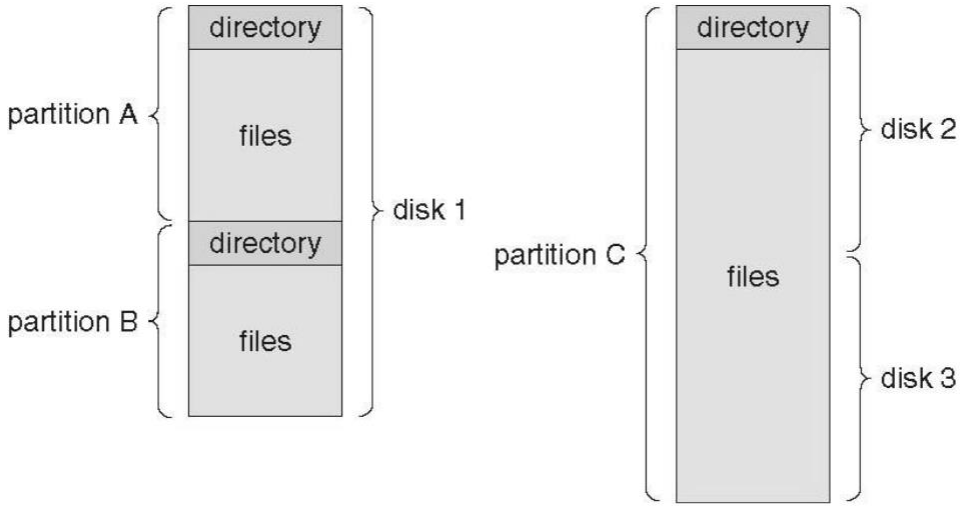


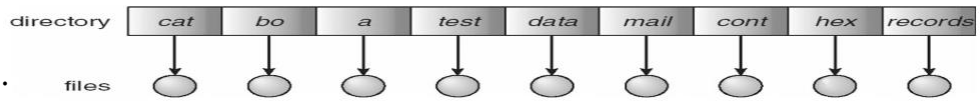
Fig: A typical file-system organization

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. Single-Level Directory

Simple to implement, but each file must have a unique name.



Draw Backs:

- Naming Problem
- Grouping Problem

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 user's 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.

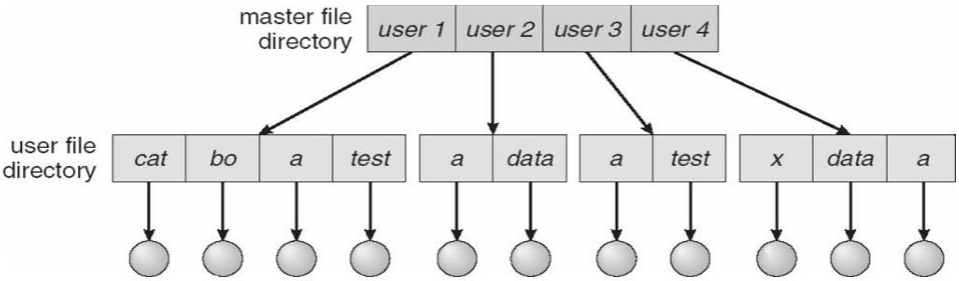


Fig: Two-level directory structure.

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.

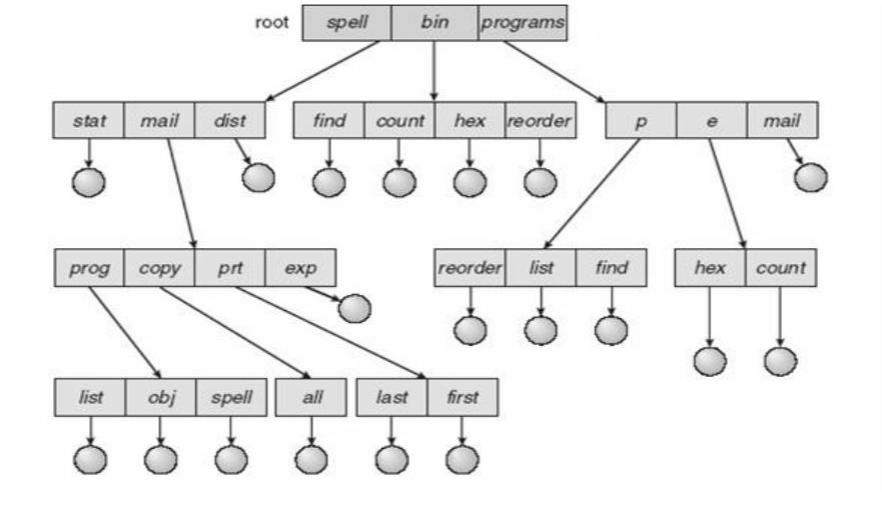


Fig: Tree Structured Directory Structure

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 file system.
    - 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 file systems, as well as ordinary files in the current file system.
- 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?

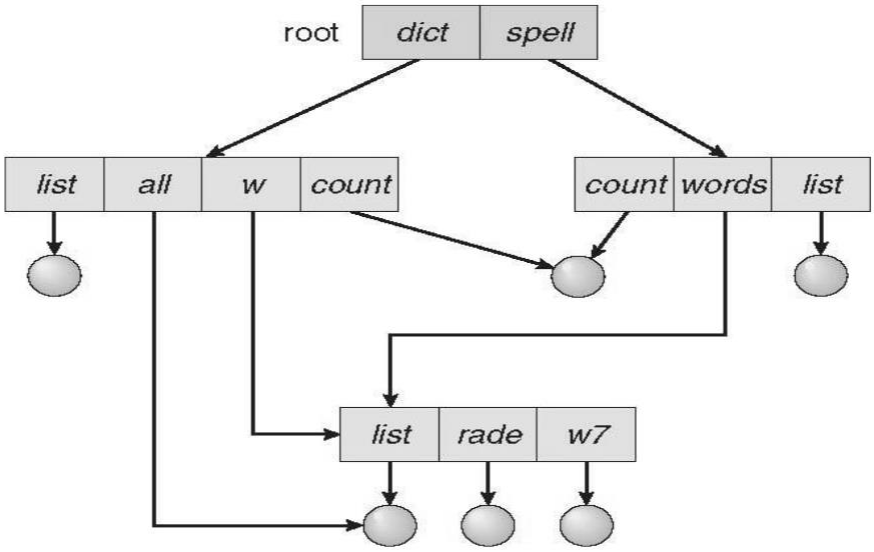


Fig: A cyclic-graph Directory Structure

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.)

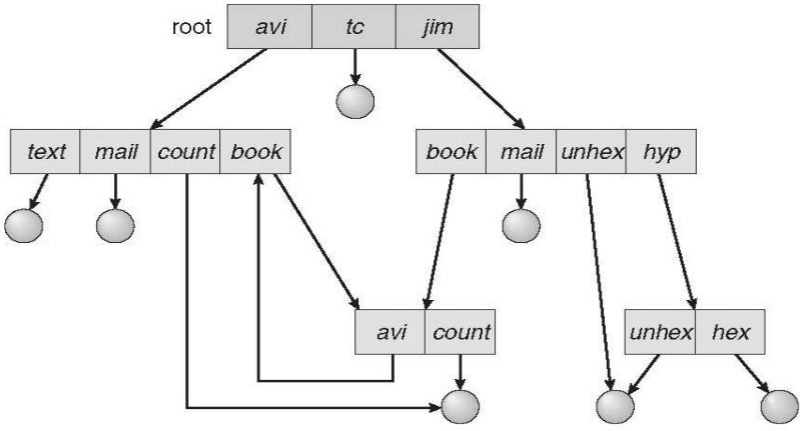


Fig: General Graph Directory

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 file system 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 file system are now hidden by the mounted file system, and are no longer available. For this reason some systems only allow mounting onto empty directories.
- File systems can only be mounted by root, unless root has previously configured certain file systems to be mountable onto certain pre-determined mount points. (E.g. root may allow users to mount floppy file systems to /mnt or something like it.) Anyone can run the mount command to see what file systems are currently mounted.
- File systems may be mounted read-only, or have other restrictions imposed.

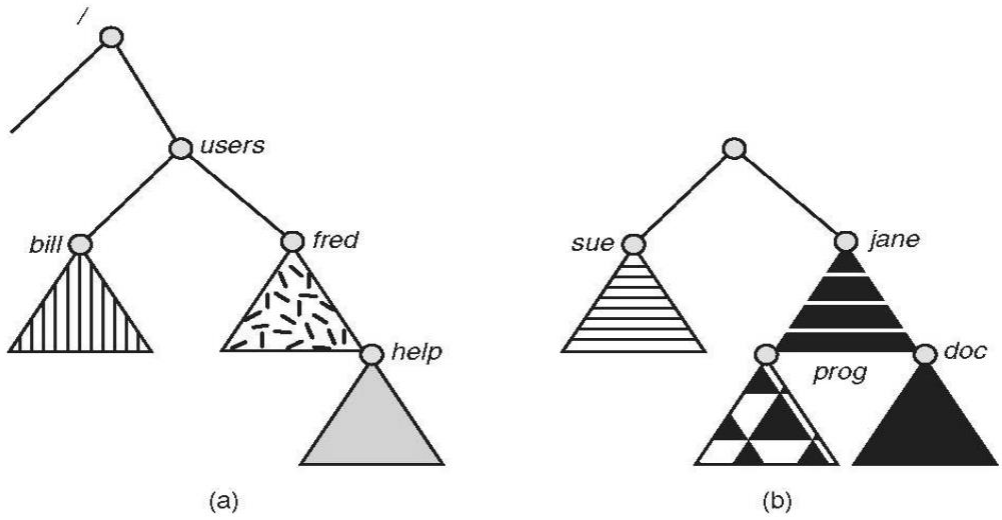


Figure :File system. (a) Existing system. (b) Unmounted volume.

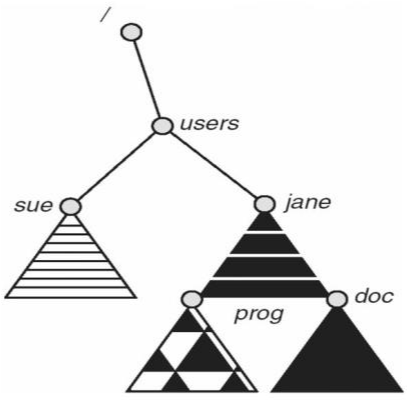


Figure :Mount point.

- 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 file systems to be mounted to any directory in the file system, much like UNIX.

File Sharing

Multiple Users

- On a multi-user system, more information needs to be stored for each file: o
  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 (User), the Group, and to the rest of the world (the universe, a.k.a. Others.)
  - Some systems have more complicated access control, allowing or denying specific accesses to specifically named users or groups.

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 file systems, generally using (anonymous) ftp as the underlying file transport mechanism.

The Client-Server Model

- When one computer system remotely mounts a file system 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 file systems 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.

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.

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.

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. Unfortunately the long delays involved in network operations prohibit the use of atomic operations.

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.

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.

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.

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.

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.

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:
- 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 pbg    student      512   Aug 3 14:13     student-proj/
-rw-r--r--    1 pbg    staff       9423   Feb 24 2003     program.c
-rwxr-xr-x    1 pbg    staff      20471   Feb 24 2003     program
drwx--x--x    4 pbg    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/
```

Fig: Sample permissions in a UNIX system.

- Windows adjusts files access through a simple GUI:

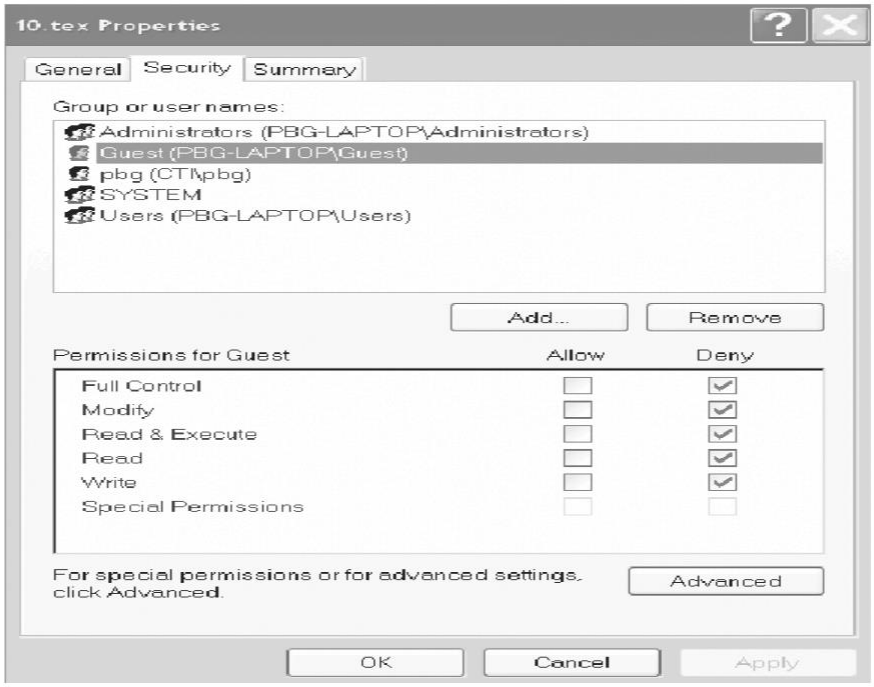


Figure - Windows 7 access-control list management.

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.

4.3 File-System Implementation

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.
- 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 file system 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)

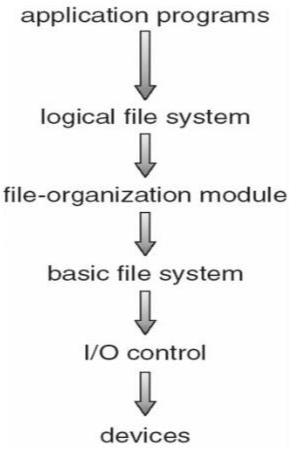


Fig: Layered file system

File-System Implementation

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 file system, 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.

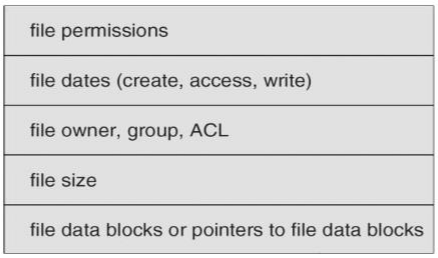


Fig: A typical file-control block.

- There are also several key data structures stored in memory: o An in-memory mount table.
  - o An in-memory directory cache of recently accessed directory information.
  - o 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.
  - o 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 below illustrates some of the interactions of file system components when files are created and/or used:
  - o 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.
  - o 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.
  - o 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.
  - o 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.

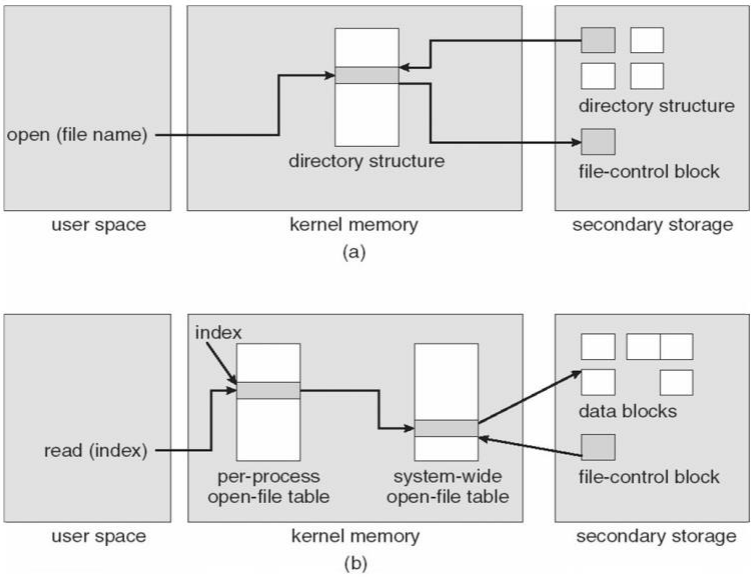


Fig: In-memory file-system structures. (a) File open. (b) File read.

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 file system (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 file systems 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 file system 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 file systems 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. File systems 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 file system.

Virtual File Systems

- Virtual File Systems, VFS, provide a common interface to multiple different file system types. In addition, it provides for a unique identifier (vnode) for files across the entire space, including across all file systems of different types. (UNIX inodes are unique only across a single file system, 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 file system.
    - The dentry object, representing a directory entry.
- Linux VFS provides a set of common functionalities for each file system, using function pointers accessed through a table. The same functionality is accessed through the same table position for all file system types, though the actual functions pointed to by the pointers may be file system-specific. See /usr/include/linux/fs.h for full details. Common operations provided include open( ), read( ), write( ), and mmap( ).

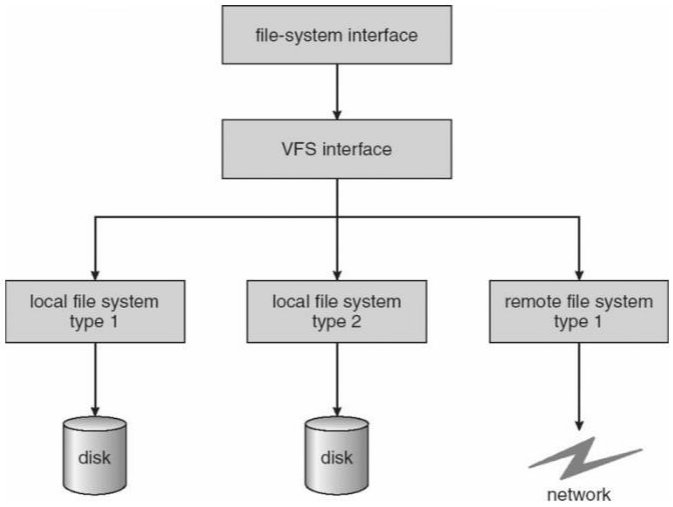


Fig: Schematic view of a virtual file system

Directory Implementation

Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.

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.

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

Allocation Methods

There are three major methods of storing files on disks:

- contiguous,
- linked, and
- indexed.

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.

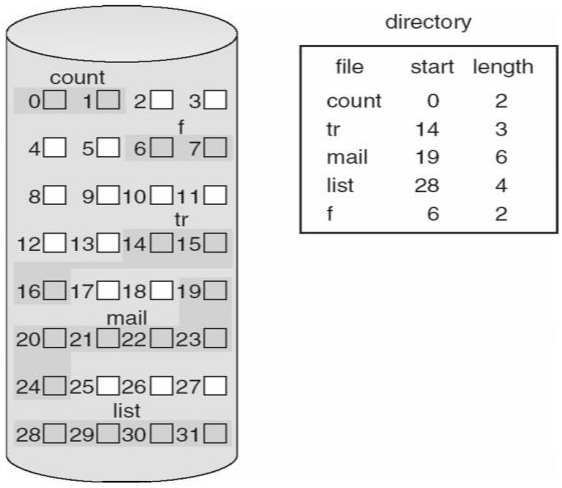


Fig: Contiguous allocation of disk space.

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.

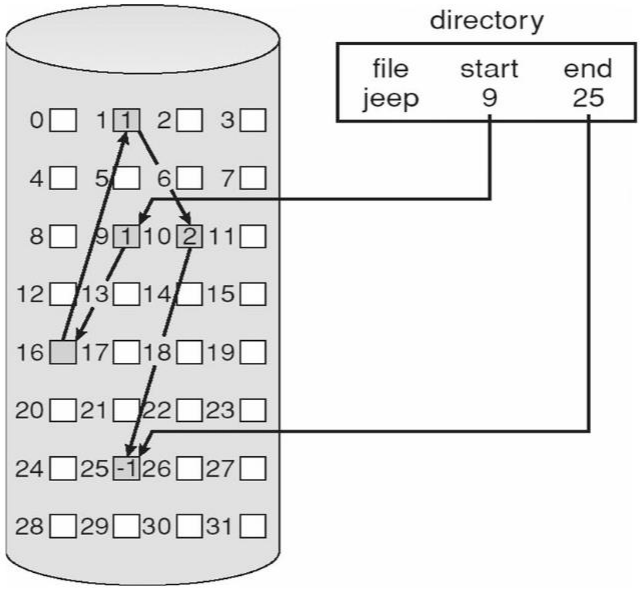


Fig: Linked allocation of disk space.

- 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.

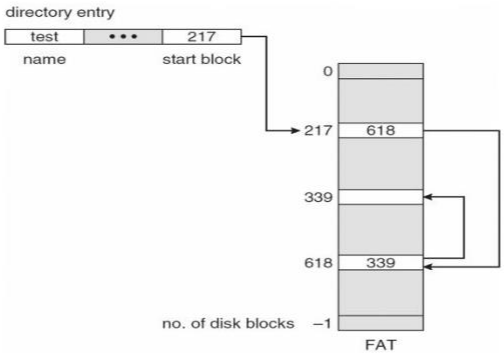


Fig: File-allocation table

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.

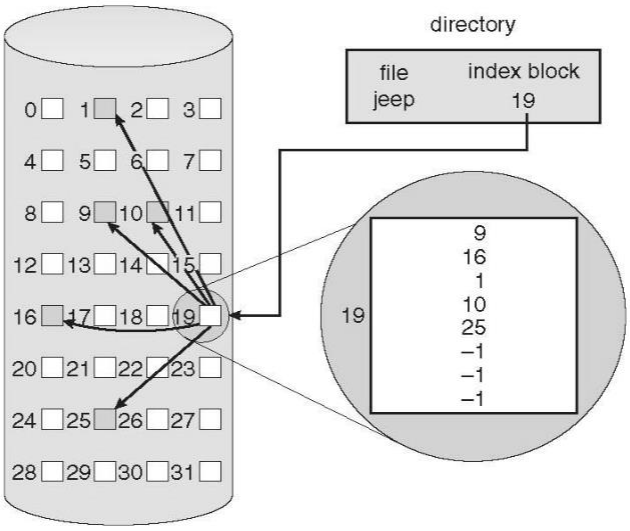


Fig: Indexed allocation of disk space

- 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.)

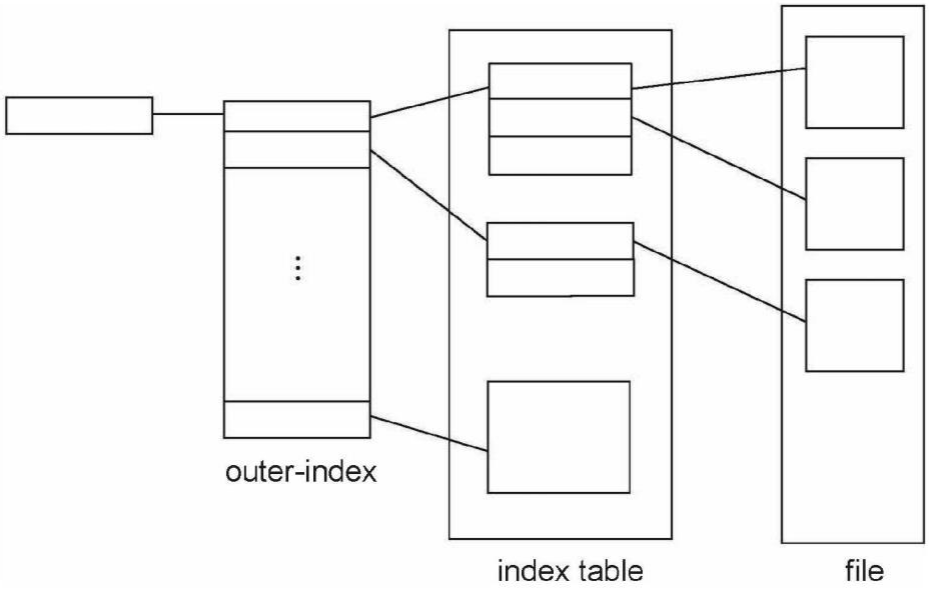


Fig: The UNIX inode

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.

Free-Space Management

Another important aspect of disk management is keeping track of and allocating free space.

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.)

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.

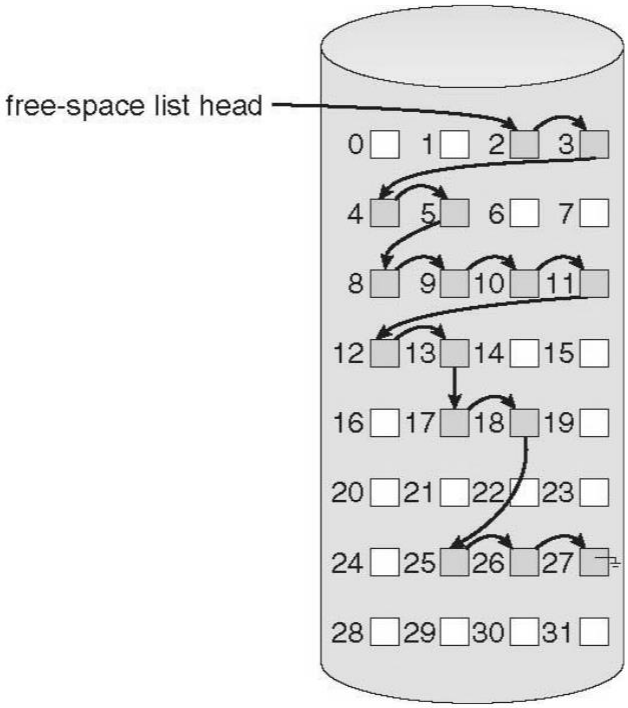


Fig: Linked free-space list on disk.

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.

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.)

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.

Efficiency and Performance

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.

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 below 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.

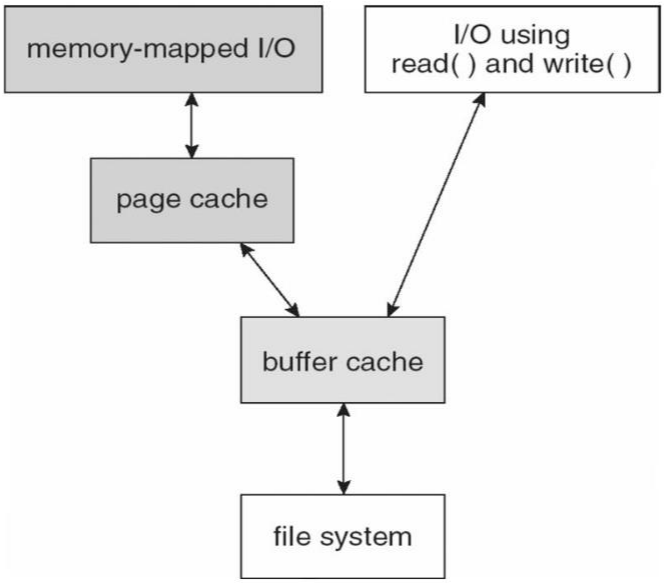


Figure : I/O without a unified buffer cache.

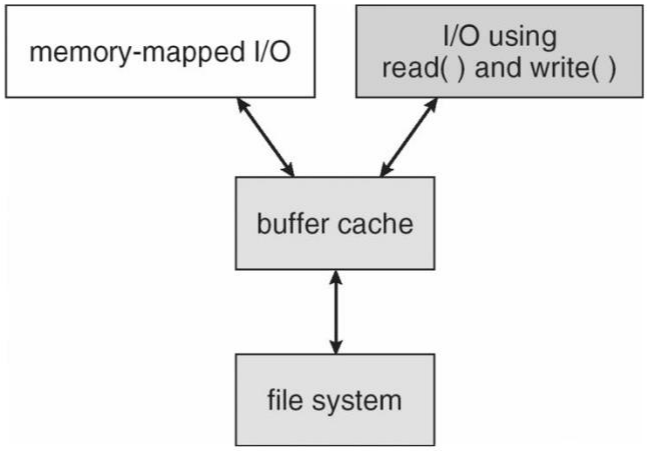


Figure 12.12 - I/O using a unified buffer cache.

- 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. 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.

Recovery

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 file system 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. o

    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.

Log-Structured File Systems

- Log-based transaction-oriented (a.k.a. journaling) file systems 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 file system, 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.

Other Solutions

- 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.

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 file system.
- 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!