

CHAPTER 3.

Contents

1. Background

1.1 Operating System

1.1.1	Introduction	...19
1.1.2	Basic Operating System Functions	...21
1.1.3	Operating System Architectures	...23
1.1.4	Operating System Design Issues	...26

1.2 Web Browsers

1.2.1	World Wide Web	...35
1.2.2	Hypertext Protocol-HTTP/1.0	...35
1.2.3	HTML	...36
1.2.4	Uniform Resource Locator	...36
1.2.5	Web browser Software	...37
1.2.6	Web Browser Concepts	
1.2.6.1	Protocol Stack	...38
1.2.6.2	Domain Name Server	...39
1.2.6.3	Caching	...40
1.2.7	Components of a Web Browser	
1.2.7.1	HTTP Client	...44
1.2.7.2	Layout Engine	...50
1.2.7.3	Rendering Engine	...52
1.2.8	Working of a Web Browser	...55

3. BACKGROUND

This section discusses the functions and design issues of contemporary operating systems and web-browsers. This gives a better insight into what this project must have and must not incorporate.

3.1 OPERATING SYSTEMS

3.1.1. INTRODUCTION

An operating system is a set of programs that work together as a layer between the user and the computer hardware. Many years ago, it became abundantly clear that some way had to be found to shield programmers from the complexity of the underlying hardware. The way that has evolved gradually is to put a layer of software on top of the bare hardware, to manage all parts of the system, and present the user with an interface that is easier to program, and work with. This layer of software is referred to as the Operating System.

Operating systems, today, are becoming more and more powerful, stable and reliable than ever before. Today, every variant and type of operating system is available to the user community according to their use in homes, research, offices, defense, education, and various other fields. With it emerged a new generation of operating systems, with GUIs (Graphical User-Interfaces) and better system utilities and application packages.

With passage of time, operating systems got classified into:

General-purpose operating systems

A general-purpose operating system is a comprehensive user-interface between a human user and the hardware. They provide users with a myriad of features, like well-structured file-systems, a well-defined input/output interface etc. Consistency and completeness are their signature. Common examples include Microsoft Windows, and Linux.

Embedded-Operating Systems

Embedded operating systems are small programs that run in a hardware-constrained environment. Such operating systems are often embedded in small programmable memory modules. Such modules are programmed for very specific applications. Common examples include Robot-controlling programs.

Specific-purpose operating systems

They are small operating systems meant for specific purpose. They lack in rich features but provide ample facilities to achieve what they are designed for.

A conventional operating system appears as a layer between hardware and the user/programs using the hardware.

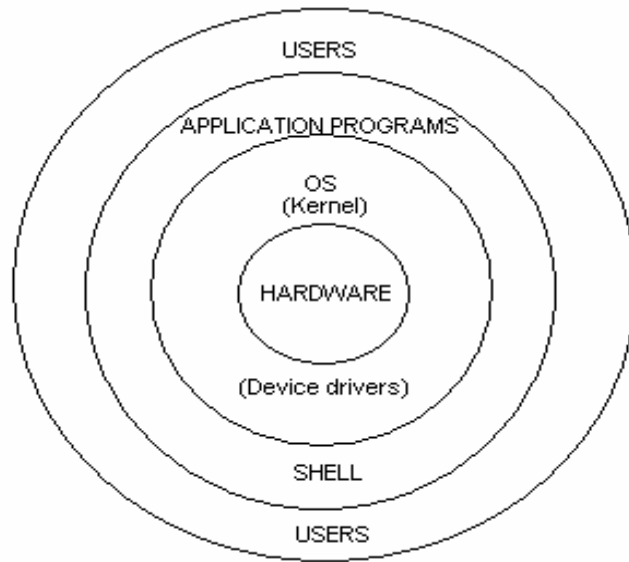


Figure 1 Where an OS fits in a computer system...

3.1.2. Basic Operating System Functions

An operating system comprises of the following components:

- Process Management module
- Main-memory management module
- File- and Bulk-storage management
- I/O Sub-system
- Secondary Storage Management

Process management

A process is a program in execution. A process needs certain resources including CPU-time, memory, files, and an I/O device to accomplish its tasks. These resources are either given to the process when it is created or allocated to it while it is running. In connection to process management, an operating system is responsible for:

- Creating and deleting both system- and user-processes
- Suspending and resuming processes
- Providing mechanisms for process-synchronization
- Providing mechanisms for process-communication
- Providing mechanisms for deadlock-handling

Main-memory management

The main memory is a large array of words or bytes, ranging in number from hundreds of thousands to billions. The main memory is a repository for quickly accessible data shared by the CPU and I/O devices. For a program to be executed, it must be mapped to absolute address space and located into the memory. As the program executes, the CPU accesses program instructions and data from the memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available and the next program can be loaded and executed. Many different memory-management schemes are available and effectiveness of the algorithms depend on the particular situation and application. The operating system is responsible for

- Allocating and de-allocating memory to processes as needed
- Deciding which processes are to be loaded into the memory when the space is available
- Keeping track of which parts of the memory are currently being used and by which process

File Management

File management is one of the most visible components of an OS. Computers can store information, in several different types of physical storage media. For convenient use of computer systems, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of the devices to define a logical storage unit, the file. The operating system maps files onto physical media and accesses these files via the storage devices.

The operating system implements the abstract concept of a file by managing mass storage media, such as, hard disks and tapes, and the devices that control them. Also, files are

normally organized into directories to ease their use. The operating system is responsible for: -

- Creating and deleting files
- Creating and deleting directories
- Supporting primitives for manipulating files and directories
- Mapping files onto secondary storage
- Backing up files on stable storage media

I/O Subsystem

One of the purposes of the operating system is to hide the peculiarities of specific hardware devices from the users. The I/O sub-system consists of: -

- A memory management component for buffering, caching, spooling, etc.
- A general device driver interface
- Drivers for specific hardware devices

Secondary Storage Management

The main purpose of an operating system is to execute programs. These programs with the data they access must be in the main memory during execution. Because the main memory has to accommodate all the data and programs, and because the data it holds is lost when power is lost, the computer system must provide secondary storage. Most programs including compilers, assemblers, sort routines, editors and formatters are stored on the disk until loaded into memory and then use the disk as both the source and the destination of their processing. The operating system is responsible for the following activities in connection to disk management: -

- Free-space management
- Storage management
- Disk scheduling

Command Interpreter system

One of the most important system programs for an operating system is the command interpreter, which is the interface between the user and the operating system. Some operating systems include the command interpreter in the kernel. Others treat the command interpreter as a special program that is running when the job is initiated. Many commands are given to the operating system by control statements. A program that reads and executes the control instruction is executed automatically. This program is called the command-line interpreter or the *shell*.

3.1.3 Operating System Architectures

Operating systems can be designed using several different models. Four common design models are: -

- Monolithic systems
- Layered systems
- Virtual machines
- Client-server systems

Monolithic systems

In a monolithic system, the operating system is written as a collection of procedures, each of which can call any of the other ones whenever it needs to. Each procedure has a well-defined interface in terms of parameters and results, and each one is free to call any other, if the latter provides some useful computation that the former needs. To construct the object program of the operating system, each of the individual procedures are compiled and then bound together using the system linker.

To achieve structuring in monolithic systems, abstractions such as system calls are used. Most monolithic systems define two modes of processing – the kernel mode and the user mode. In the kernel mode, a central program called the operating system kernel, is in control of the CPU. It is a privileged mode in which all instructions to the CPU are allowed. The user-mode does not allow certain instructions, like those that access the system hardware directly, to execute. This provides a certain degree of protection to the system from user-programs.

Monolithic operating systems can be structured as shown in the figure below :-

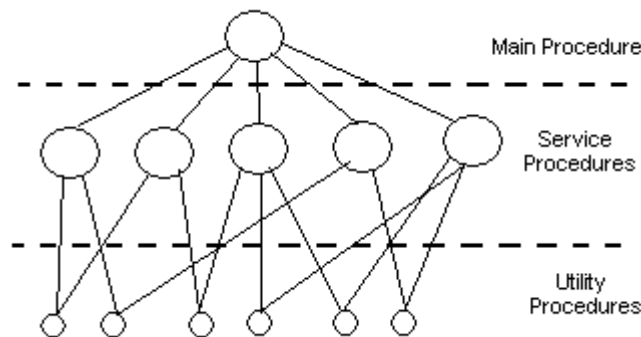


Figure 2 Structuring in monolithic kernels

Layered Systems

In this approach, the operating system is organized into a hierarchy of layers, each one constructed upon the one below it. Each layer presents a well-defined interface to the next higher layer, thereby hiding the lower layer details. Such functionality makes it easier to develop applications and cleaner (less bugs) code. However, it does cause a certain performance problem because each message (a function call, with parameters) must pass through many different layers before actually getting executed.

An example of a layered system is the THE (Technische Hogeschool Eindhoven) system developed by E.W. Dijkstra. The system had six layers as shown in Fig.4.

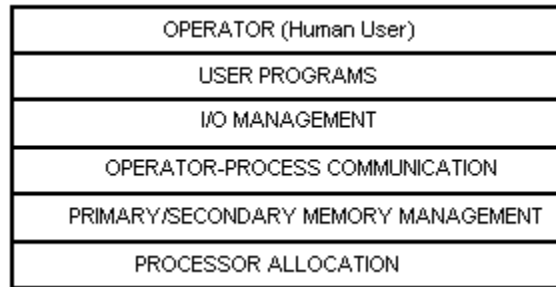


Figure 3 T.H.E. system layers

Virtual Machines

Unlike layered and monolithic operating systems, virtual machines are not extensions of the hardware. Rather, they emulate different hardware schemes on different machines. For example, a Pentium system wanting to run an 8086 MS-DOS program needs a virtual machine to emulate the working of the 8086 processor. Virtual machines are a way to hide the underlying architecture while maintaining a known interface for the higher layer programs. These days, however, virtual machines are used in a rather different context. For example in Java, a Java Virtual Machine is the interpreter that translates compiled Java programs, hiding the actual architecture of the machine it is running on. This greatly increases the portability of code written in Java, because the JVM can be implemented for most of the current hardware configurations.

A recent development in the area of virtual machines is the use of an *exokernel*, a privileged program that allocates different virtual machines to different programs running on a system, as is required by them. The *Aegis* exokernel is an example of such an operating system.

Client-server model

A trend in modern operating systems is to take the idea of moving code up into higher layers even further and remove as much as possible from the operating system, leaving a minimal *microkernel*. The usual approach is to implement most of the operating system functions in user-processes. To request a service, such as reading a block of a file, a user process (*client*) sends the request to a *server process*, which then does the work and sends back the answer.

In this model, all the kernel does is handle the communication between clients and servers. By splitting the operating system up into parts, each of which handles one facet of the system, such as file-service, process service, terminal service, or memory service, each part becomes small and manageable. This has the advantage of isolation of sub-systems (independent failures), adaptability to be used in distributed systems, and the abstraction for programs hiding details of the location of the server.

A related architecture is that of an object-oriented operating system. All system components are represented as objects with well-defined methods and attributes. Object-oriented systems differ from client-server systems in that, that the latter do not have a peer-relationship between the two communicating parties.

Being in the user-mode, the servers cannot directly access the hardware. This minimizes the chances of a total system crash, even if some servers fail. A simple scheme for a client-server model is depicted in the following figure.

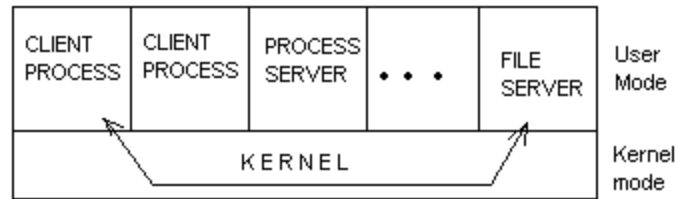


Figure 4 Client/Server model of an operating system

3.1.4 Operating System Design Issues

Several design aspects must be considered before the development of any operating system. The following issues are of primary importance: -

- System call interface
- Process-scheduling
- Inter-process – communication
- Process deadlock handling
- Memory management
- Virtual memory management
- File-system
- I/O management

System Calls

System calls are a set of routines that form an interface between the operating system components and the different application programs running on top of it. System calls are an abstraction of operating system functions. They provide a means for the applications to use the different operating system services. System calls may be classified by different operating systems in different ways. One way of classification is identification by means of the identity of the operating system component for which the system call is meant. This classification results in the following categories of system calls: -

- Process management system calls
 - Deal with process creation, suspending, resuming, status recalling etc.
- Signals
 - Sending signals to processes, such as process abortion
- File I/O (Stream and block input-output) system calls
 - Creating, reading from and writing to disk files
 - Reading from and writing to directories, renaming and checking accessibility of directories
 - Random file pointer seeking etc.
- Directory and File-system management system calls
 - Creating and removing directories
 - Linking and unlinking file-system entries
 - Mounting and un-mounting file systems
- Protection system calls
 - Changing access rights to a disk file or directory
 - Checking access and ownership to a file, etc.
- Time management system calls
 - Getting and setting the system time

Following is a list of the system calls provided in the MINIX¹ operating system: -

Process management	Fork
	Waitpid
	Wait
	Execve
	Exit
	Brk
	Getpid
	getpgrp
	setsid
	ptrace
Signals	sigaction
	sigreturn
	sigprocmask
	sigpending
	sigsuspend
	kill
	alarm
	pause
File I/O management	creat
	mknod
	open
	close
	read
	write
	lseek
	stat
	fstat
	dup
	pipe
	ioctl
	access
	rename
	fcntl
Directory and file-system management	mkdir
	rmdir
	link
	unlink
	mount
	unmount
	sync
	chdir
	chroot
Protection	chmod
	getuid
	setuid
	setgid
	getgid
	chown
	unmask
Time management	time
	stime
	utime
	times

¹ Minix operating system is a UNIX-clone written by A.S. Tanenbaum of Vrije Universiteit, Netherlands

Process-Scheduling

In a multiprogramming environment, when more than one process is runnable, the operating system must decide which one to run first. The part of the operating system that makes this decision is called the **scheduler**. The scheduler is concerned with deciding on the policy when it comes to selecting a process, rather than providing a mechanism for it.

Process scheduling algorithms may be evaluated on the basis of the fairness (making sure that each process gets its fair share of the CPU), efficiency (keeping the CPU busy 100 percent of the time), response time (minimizing response time for interactive users), turnaround time (minimize output time for batch users), and throughput (maximize the number of jobs processed per unit time).

Scheduling algorithms are either *preemptive* or *run-to-completion (non-preemptive)*. The strategy of allowing processes that are logically runnable to be temporarily suspended is called preemptive scheduling. Several scheduling algorithms exist for use in different types of operating systems. Common non-preemptive algorithms include *Shortest-Job-First*, *First-come-first-served*, and *priority-scheduling*. Common preemptive algorithms include *Round-Robin algorithm*, *weighted-fair-queue algorithms*, *multilevel-feedback queue scheduling etc.*

There are cases when a process might have spawned several child processes. In such cases, many times the parent process knows which of the child processes is the most critical (and hence must be dispatched the CPU soonest). However, scheduling algorithms do not provide any means for the parent process to inform the scheduler about such priorities. The solution to this problem is the separation of scheduling algorithm from the scheduling policy. What this means is that the scheduling algorithm is parameterized in some way, but the parameters can be filled in by user-processes. This allows parent processes to control the scheduling of its child processes.

Inter-Process Communication

Processes frequently need to communicate with other processes. There are three issues with regards to inter-process communication: -

- how one process can pass information to another
- making sure that two or more processes do not get into each other's way when engaging in critical activities
- proper sequencing when dependencies are present

Situations in which two or more processes are reading and writing some shared data and the final result depends on what process runs precisely when, are called Race conditions. Race conditions arise when more than one process attempt access the same memory (or buffer, or any other storage) location at nearly the same instant. *Mutual exclusion* is the mechanism used to prevent such race conditions. Mutual exclusion mechanisms are based on the recognition of *critical-sections* in programs. These are typically regions where a process is reading from or writing to a resource which can be accessed for writing by another process. Mutual exclusion is the notion of not allowing different processes from

entering their critical sections with respect to a particular share resource. *Semaphores* are a solution that can be used to enforce mutual exclusion. *Deadlocks* sometimes occur owing to the enforcement of mutual exclusion. *Monitors* are a higher-level synchronization mechanism. They encapsulate procedures and data structures. They have the special property that make them useful in achieving mutual exclusion – only one process can be active in a monitor at any instant.

IPC sometimes involves the need for message-passing – the notion of one process notifying other processes. Message passing incorporates several aspects such as authentication of the communicating processes.

Process Deadlock Handling

In multiprogramming environments, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

Deadlocks occur only when all four of the following conditions hold simultaneously in a system – *Mutual exclusion, Hold and Wait, No preemption, and Circular wait*.

Deadlocks can be handled in a number of ways – preventing them by constraining how requests for resources can be made, avoiding them on the basis of additional knowledge about the process, detecting and recovering from them using preemptive techniques.

Deadlock avoidance is achieved using certain algorithms that decide whether or not the system is in a *safe* state, and whether a process may be allocated the resources it is demanding or not. Commonly employed algorithms to this end are the Resource-allocation graph algorithm and the Banker's algorithm. Systems can recover from deadlocks by means of preemptive techniques like process-termination, or resource-preemption.

Memory Management

RAM or the main memory is one of the most constrained resource in a personal-computer environment. The optimal utilization of this resource is thus a duty of the operating system, especially in a multiprogramming environment. Managing the main memory involves the management of the following activities: -

- Managing the allocation and deallocation of memory to processes
- Managing the view of the memory to the programs
- Protecting memory occupied by different programs from affecting each other

Several schemes for memory management exist. For an environment where only one program is loaded into the memory, memory management is trivial. The main use of the schemes comes into play when the system uses multiprogramming.

Processes can be allocated memory in a contiguous manner, meaning that each process occupies a single contiguous portion of memory. The most common contiguous memory allocation scheme is the *multiple-partition method*; when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system

keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered as one large hole. When a process arrives and needs memory, a hole large enough for this process. If such a hole is found then the process is allocated only as much memory as is needed, keeping the rest available for future requests. For allocation of the holes, a *first, best or worst fit* may be sought. These algorithms suffer from *external fragmentation*. As processes are loaded and removed from memory, the free memory space is broken into little pieces. This sometimes results in a situation when the total available memory is enough to suffice a new allocation request, but is fragmented into a large number of small holes (non-contiguous), and so cannot be allocated to the requesting process. Compaction is a solution to this problem, but it may not always be possible, and may cause significant delays. Fragmentation can also be internal as in the case of fixed-sized partitions. In this case, a partition may not be completely filled by a process, and yet this space cannot be allocated to any other process.

The way around such problems is to use non-contiguous memory allocation schemes. These include *paging* and *segmentation*.

Paging is a memory-management scheme that permits the physical-address space of a process to be non-contiguous. In this scheme, the physical memory is broken into fixed-sized blocks called *frames*. Logical memory is also broken into blocks of the same size called *pages*. When a process is to be executed, its pages are loaded into any available memory frames from the backing store (secondary storage). Every address generated by the CPU is divided into two parts – a page number, and a page offset. The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. Paging completely solves the problem of external fragmentation, because any free frames can be allocated to a process that needs it. However, the problem of internal fragmentation persists. frame will be wasted. The following figure shows the whole paging model.

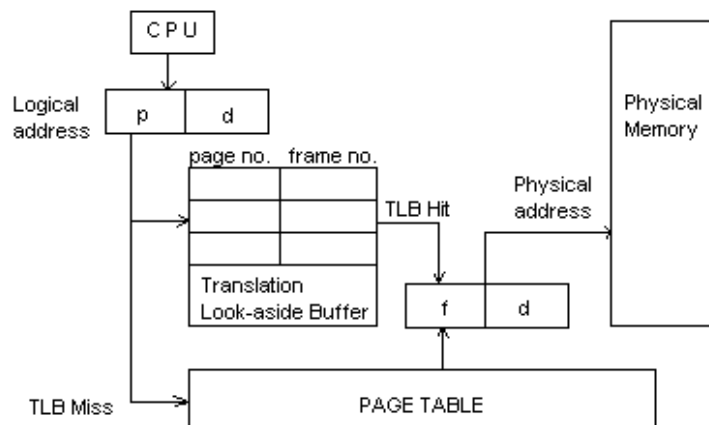


Figure 5 Paging with hardware support

Another popular approach of memory management is the use of segmentation. *Segmentation* is a memory-management scheme that supports the user view of memory – the idea that memory is arranged in segments, each containing some recognizable unit of

data or instructions (such as a function). A logical-address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within that segment.

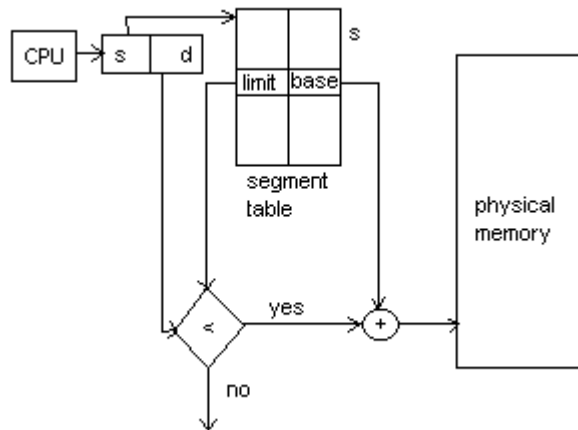


Figure 6 Segmentation hardware support

The mapping from the two-dimensional ($\langle \text{segment}, \text{offset} \rangle$ pairs) to the single-dimensional physical addresses is effected by a segment table. Each entry of the segment table has a segment *base* and a segment *limit*. The segment base contains the starting physical address where the segment resides in the memory, whereas the segment limit specifies the length of the segment. A logical address consists of two parts – a segment address (s), and an offset (d) into that segment. The offset d of the logical address must be between 0 and the segment limit. The segment base address is added to the offset to obtain the actual physical address.

Segmentation maybe coupled with paging to improve on each.

Virtual Memory Management

Virtual memory is a technique that allows the execution of processes that may not be completely in memory. This allows the execution of programs that are larger than the available physical memory. Virtual memory is based on the fact that in many cases the entire program is not needed at the same time. For instance, programs have error-handling routines that are not executed frequently. This means, that most of the time, they are not needed, and yet they must be in the memory if no virtual memory scheme is employed.

Virtual memory is commonly implemented by *demand-paging*. A demand-paging system is similar to a paging system with swapping. Processes reside on secondary memory. Rather than swapping the entire process into memory, a *lazy swapper* is used to swap in only those pages that are immediately required for the process. A lazy swapper never puts a page into memory unless that page will be needed. When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus it avoids reading into memory those pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed. However, the

guesswork of the paging program is not always accurate. Therefore, sometimes a condition known as *page fault* occurs, indicating that either the page was not brought into memory or it was invalid address. In this case, first the validity of the page requested is tested for, using a table stored with the process control block. If the page number requested is found invalid, the process is terminated. If it was valid, but the page was not brought in, a free-frame is found and a disk operation is scheduled to read the desired page into the newly allocated frame. When the disk read is complete, the internal table keeping the process information is modified to indicate that the page is now in the memory. The instruction that was interrupted is restarted. The instruction can then access the page as if it were always there. *Page replacement* is required when the memory is fully occupied and a demanded page is not in the memory, that is, a page-fault occurs. In this case the effective access time is increased because of the two disk operations (page-out and page-in). Several algorithms exist for reducing the overheads associated with page replacement. They are called *page-replacement algorithms*, common of which are the First-in-first-out algorithm, the optimal-page replacement algorithm, the least-recently used page replacement, and the second-chance algorithm.

In single-user systems, all free-frames are initially queued up. When a user process starts, it generates a sequence of page-faults. All the free-frames are allocated to pages of the process. If there are still more pages to be loaded, then page-replacement algorithms are used to replace pages that are not required immediately. In multiprogramming systems, the issue is more important. If a process has all its pages active and yet it needs some more pages to be in the memory, then in the situation when the memory is fully occupied, pages must be replaced. In this case, all pages of the process in the memory are active and may be needed frequently. This means that the page just paged-out may be needed to be brought into the memory soon. This leads to a condition of high paging-activity called *Thrashing*. An operating system needs to control the amount of thrashing.

File System

A file-system provides a mechanism for online storage of and access to both data and programs of the operating system and the users of the operating system. The file system has two parts – a collection of *files*, and a *directory structure*. A file stores a set of related data. The directory structure organizes and provides information about all the files in the system. Some file systems have a third part, *partitions*, which are used to separate physically or logically large collections of directories.

File systems are usually implemented internally as hierarchical structures. They provide an accessible structure to data stored on permanent data storage media.

File systems provide common operations for manipulating entries in the file system. Creating, deleting, truncating, writing to, and reading from disk files are the commonest operations required from a file system. In addition, each disk file has certain attributes associated with it, viz. the filename, the file size, location on the physical media, etc. Operations for modifying these attributes are also provided by the file system. Access methods – sequential and random (direct) need also to be managed.

Commonly, directory structures are arranged as hierarchies – trees, general and acyclic graphs. The allocation of space on the data storage media is another problem that the operating system must handle. It can be handled in many different ways – contiguous,

linked, and indexed allocation. Different problems that arise in space management and allocation are parallel to those that occur in memory management.

I/O Sub-system

The role of operating system in computer I/O is to manage and control I/O operations and I/O devices. The I/O hardware comprises of an I/O device, a bus or wire, a port and a controller. The controller is responsible for carrying out the I/O independent of the processor. The controller has a set of registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers. One way this communication can occur is the use of special I/O instructions that specify the transfer of a byte or word to an I/O port address. The I/O instruction triggers bus lines to select the proper device and to move bits in or out of the device registers. These I/O instructions can be invoked directly by the processor, or a *memory-mapped* system for writing to I/O ports. Basically, the I/O controller registers are mapped onto the host's main memory region. An example of this is the VDU memory region. This technique has the disadvantage that the contents of the I/O memory is exposed to change from external programs.

An I/O port typically consists of four registers, called the status, control, data-in and data-out registers. Several techniques for specifying I/O instructions exist. One such technique is *polling*, where in, the processor repeatedly monitors the status register, until it gives a *free* (not busy) pattern. It then sets the write bit in the command register and writes a byte into the data-out register. The host then sets the command-ready bit. When the controller notices that the command-ready bit is set, it sets the busy bit. The controller then reads the command register and sees the write command. It reads the data-out register to get the byte and does the I/O to the device. The controller clears the command-ready bit, clears the error bit in the status register to indicate that the device I/O succeeded and clears the busy bit to indicate that it is finished. The process which initiates this kind of I/O is called handshaking.

Another technique used for managing I/O is the use of *I/O interrupts*. The CPU hardware has a wire called the IRQ (Interrupt Request) line, that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt request line, the CPU saves a small amount of state, such as the current value of the instruction pointer (IP), and jumps to the *interrupt-handler* routine at a fixed address in the memory. The interrupt handler determines the cause of the interrupt, performs the necessary processing and executes an IRET (return from Interrupt) instruction to return the CPU to the execution state prior to the interrupt. This mechanism allows the CPU to handle asynchronous events. It is required in modern operating systems to defer interrupts when the CPU is handling certain critical executions. Also, an efficient way to dispatch to the proper handler for a device without resorting to polling. Aside from this, multilevel interrupts are required to distinguish between high- and low-priority interrupts. Most CPUs have two IRQ line. One is the non-maskable interrupt, which is reserved for events such as unrecoverable memory errors. The second interrupt line is maskable – i.e. it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted.

Interrupt handler routines are located at fixed locations in the main memory. The addresses of these locations are stored in what is known as an *interrupt vector*. The

interrupt vectors sometimes use interrupt-chaining, which involves storing a the address of the head of a list of addresses of the interrupt handlers related to each interrupt. All the handlers are invoked in a chain, until the right one is obtained. The events from 0 to 31 are non-maskable interrupts and are used to specify error conditions and exceptions. The events from 32 to 255, which are maskable are used for purposes such as device-generated interrupts.

A modern operating system interacts with the interrupt mechanism in several ways. At boot time, the operating system probes the hardware buses to determine what devices are present, and installs the corresponding interrupt handlers into the interrupt vector. During I/O, the various device controllers raise interrupts when they are ready for service. These interrupts signify that output has completed, or that input data are available, or that a failure has been detected. The interrupt mechanism is also used to handle a wide variety of exceptions. Software interrupts are generated by the operating system and application programs that require the operating system kernel services. When this occurs, the interrupt hardware saves the current context and then switches to the kernel mode in order to execute kernel code.

The kernel must deal with several I/O issues, including I/O scheduling, buffering, caching, spooling and device reservation, error handling etc.

3.2 WEB BROWSERS

3.2.1 The World-wide Web

The World Wide Web is a global collection of linked hypermedia documents. The World Wide Web is based on the client-server model. The software at the user-side acts as the client and the remote system hosting the information is referred to as a *web-server*. The software at the client-side, which is called a *web-browser*, requests data or resources from the server (a data-house catering to the needs of thousands of users). There are several kinds of servers differentiated on the basis of their functionality. Servers can: -

- Deliver web pages (web server).
- Handle e-mail traffic (mail server).
- Provide access to particular files (file server).

Like in any other client-server model, data is transferred on the web using the request-response sequence. The client requests a resource or service, and the server responds by sending the relevant information.

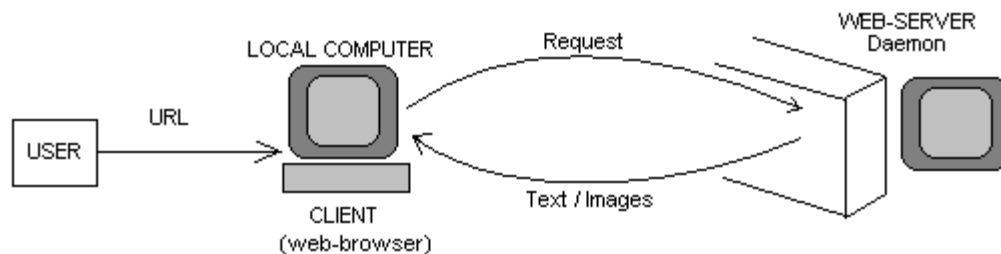


Figure 7 Basic Client Server Architecture

The world-wide web is based on three main elements – HTTP (Hypertext Transfer Protocol), HTML (Hypertext Markup Language), and URLs (Uniform Resource Locators). HTTP is the application-layer protocol that manages data transfers between the client and the server, rendering services as specifying the type of data being transferred, or providing a template for the request of a resource. HTML is a formatting language for text displayed on web-pages. HTML is an interpreted markup language. The software at the client-side is responsible for interpreting HTML content. A URL serves as a universal name for the location of a particular web-resource, such as an HTML document, or an image.

3.2.2 Hyper-Text Protocol -- HTTP/1.0

The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for *distributed, collaborative, hypermedia information systems*². It is a generic, stateless, object-oriented protocol which can be used for many

² The W3C defines the world-wide web as comprising of such systems.

tasks, such as name servers and distributed object management systems, through extension of its request methods (commands). A feature of HTTP is the typing of data representation, allowing systems to be built independently of the data being transferred.

Practical information systems require more functionality than simple retrieval, including search, front-end update, and annotation. HTTP allows an open-ended set of methods to be used to indicate the purpose of a request. It builds on the discipline of reference provided by the Uniform Resource Identifier (URI), as a location (URL) or name (URN), for indicating the resource on which a method is to be applied. Messages are passed in a format similar to that used by Internet Mail and the Multipurpose Internet Mail Extensions (MIME).

HTTP is also used as a generic protocol for communication between user agents and proxies/gateways to other Internet protocols, such as SMTP, NNTP, FTP, Gopher, and WAIS, allowing basic hypermedia access to resources available from diverse applications and simplifying the implementation of user.

3.2.3 HTML³

The Hypertext Markup Language (HTML) is a simple markup language used to create hypertext documents that are platform independent. HTML documents are SGML documents with generic semantics that are appropriate for representing information from a wide range of domains. HTML markup can represent hypertext news, mail, documentation, and hypermedia; menus of options; database query results; simple structured documents with in-lined graphics; and hypertext views of existing bodies of information. HTML has been in use by the World Wide Web (WWW) global information initiative.

HTML comprises of certain pre-defined tags each of which is used for formatting text and other hypermedia documents. HTML is an interpreted language, with a loose syntax, meaning that errors in HTML are generally ignored and treated as plaintext. The complete grammar of the HTML is attached in the Appendix-A of this document.

3.2.4 URL (Uniform Resource Locator)⁴

A URL is a compact string representation of a resource available via the Internet. A URL is a special case of the generic Uniform Resource Identifier standard. URLs are used to locate resources by providing an abstract identification of the resource location. A URL contains the name of the scheme being used (<scheme>) followed by a colon and then a string (the <scheme-specific-part>) whose interpretation depends on the scheme. In BNF the URL can be represented as:

<scheme>:<scheme-specific-part>

For example: In <http://www.google.com/>, the <http> scheme is used, followed by the colon and the scheme specific location. The HTTP URL can be further broken down into the following parts:

<HTTP>://<Host address><location of resource as a directory path and/or filename>.

³ Adapted from RFC1866, also refer to Appendix-A for HTML syntax

⁴ The complete URL specification can be obtained from RFC 1738

3.2.5 Web-browser Software

A web browser is software that resides on the user's system enabling the user to view World Wide Web (WWW) documents and access the Internet. It is the user's window to the Internet.

3.2.5.1 Type of web-browsers

There are three basic types of web-browsers in use today:

- Line-mode browsers
- Full-screen browsers (like Lynx)
- Graphical browsers (like Netscape).

3.2.5.1.1 Line Mode Browsers:

They work much like FTP. The user types a command, the relevant information appears on the user's screen. Then the user types another command for more information and so on. They are very unfriendly and are hardly used in the present world.

3.2.5.1.2 Full-Screen Browsers:

Full screen browsers places a text based menu on the user's screen which looks much like Gopher menus. Different full screen browser work differently but most of them allows a user to move cursor up or down the screen, select a highlighted word or phrase (link), press the ENTER or RETURN keys (or the right arrow key), and the user is automatically taken to a new document or file. They are generally easy to work with, however they too have become obsolete as they are incapable of displaying pictures and graphical images.

3.2.5.1.3 Graphical Browsers:

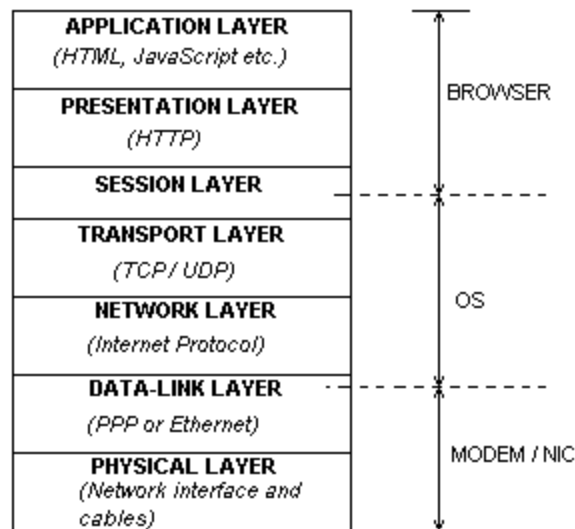
These browsers are capable of displaying pictures and other graphical images such as hypermedia. They are extremely user friendly and used extensively in today's world. There are a host of graphical browsers available today. They all work the same way only notable difference is in the convenience features they offer for navigation and the manner in which they manage the web and keeping track of URLs.

3.2.6 Web-browser concepts

3.2.6.1 *Protocol stacks*

Browsers always open files and process them; however some of the files are local to the user's machine, while others require a network connection. Most browsers require that network software be available when they start up, even though the user may not need the network software. Thus even if the user does not have a network or modem to the system, still he/she can use the web browser locally, as long as the network software is available. This network software is often referred as the protocol stack as it implements various layers of the OSI model.

Like any other networked application, browsers too make use of a series of protocols to connect them with server computers. The following model represents the OSI layers with respect to web-browsers.



7

Figure 8 The OSI model and the role of the browser

3.2.6.2 *Domain Name Server (DNS)*

The DNS is the system, which resolves a host name to an IP address, since the computer can recognize only IP address (say, 192.32.9.0) and not mnemonic names (ex: www.yahoo.com).

A DNS server is just a system running DNS software consisting of two parts:

- The actual name server
- A Resolver

3.6.2.2.1 Working of DNS:

The client types a mnemonic representation of the destination machine address into the 'Browser Address Bar', which is sent as a DNS query message to the DNS system, specifying the hostname that needs to be translated to an IP address.

Now the DNS system first tries to 'Map' (i.e. name-to-address conversion) the supplied address itself, and if it is unable to do so then a **Resolver** queries its nearest name server and which in turns queries its nearest name server till the address is resolved.

After a delay, ranging from milliseconds to tens of seconds, the client receives a DNS reply message that provides the desired mapping. Thus, from the client' s perspective, DNS is a simple, straightforward translation service. But in fact, the DNS that implements the service is complex, consisting of a large number of distributed name servers distributed around the globe, as well as an application-layer protocol that specifies how the name servers and querying hosts communicate.

DNS Hierarchy:

There are three types of name servers which interact with each other to translate the host machine's address. These servers are the *local name servers*, *root name servers*, and *authoritative name servers*.

Local Name Server:

Each ISP (Internet Service Provider) has a local name server (also called a default name server). When a host issues a DNS query message, the message is first sent to the host' s local name server. The local name server is typically "close to" the client; in the case of an institutional ISP, it may be on the same LAN as the client host; for a residential ISP, the name server is typically separated from the client host by no more than a few routers. If a host requests a translation for another host that is part of the same Local ISP, then the local name server will be able to immediately provide the requested IP address.

Root Name Servers:

In the Internet there are a dozen or so root name servers, when a local name server cannot immediately satisfy a query from a host (because it does not have a record for the hostname being requested), the local name server behaves as a DNS client and queries one of the root name servers. If the root name server has a record for the hostname, it sends a DNS reply message to the local name server, and the local name server then sends a DNS reply to the querying host. But the root name server may not have a record for the hostname. Instead, the rootname server knows the IP address of an "Authoritative name server" that has the mapping for that particular hostname.

Authoritative Name Server:

Every host is registered with an authoritative name server. Typically, the authoritative name server for a host is a name server in the host' s local ISPBy definition, a name server is authoritative for a host if it always has a DNS record that translates the host' s hostname to that host' s IP address. When a root server queries an authoritative name server, the authoritative name server responds with a DNS reply that contains the requested mapping. The root server then forwards the mapping to the local name server,

which in turn forwards the mapping to the requesting host. Many name servers act as both local and authoritative name servers.

3.2.6.3 Caching

Caching is the technique of storing last visited web pages (on some heuristics say, last-visited, most frequently viewed page, or pages viewed within some time-span) in the local hard drive or the main memory space of the client and are made available to the client, if he/she requests for the same pages next time.

3.2.6.3.1 Working of Cache-Servers

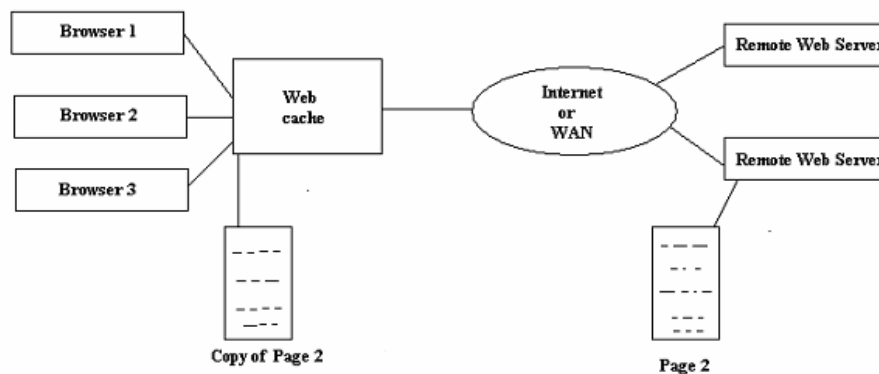


Figure 9 The OSI model and the role of the browser

Whenever a client requests for a web-page (say, Page 2) from a remote web server, the request first goes to the cache server, which searches the requested page in its' repository

and if found returns the same. However if not found, the request is then forwarded to the remote web server for which the original request was meant.

The need for overhead involved in caching, as against simply sending the request to the 'Remote Web Server' each time a page is needed, can be understood by the points stated below:

- Lower bandwidth costs and faster response times. Faster response times are achieved since 'cache hits' are served to the clients at LAN speeds, since the data resides in the local machine or the domain and 'misses' at WAN speeds as the data has to be fetched from the remote server.
- Reduce burden on origin servers (host servers).
- Enhanced security (especially in case of LANs and Intranets).
- More reliable and better service quality.

3.2.6.3.2 Classification of Cache-Servers

Cache-Servers can be classified on the basis of:

- The physical storage location (i.e. where the data repository is present)
- Approach to 'caching' (i.e. through software or by embedding hardware in appliances).

Accelerators

These first generation web caches are also called 'browser caches' since they reside on the local machines and are used by the browser. A browser's cache-size typically ranges between 5-50 MB.

Link-based accelerators

When a web-page is being loaded by a web-browser, the accelerator hits all the links present on the web page, and saves the linked web-pages into the 'cache repository'. Thus when the user clicks a link, all the browser has to do, is to load the page from the cache and display.

However such a technique proves to be a bottleneck, as the browser would be busy downloading all the links into the cache, although they may not be required.

History based accelerators

Such accelerators cache only those pages that have been most frequently used or browsed by the client. Thus they are capable of providing a performance boost when same pages are viewed again.

However such accelerators suffer if the site is browsed deeper than usually done.

Proxy Servers

A proxy server, also called a proxy cache, is an application-layer network service for caching Web objects. Unlike browser caches, proxy caches can be simultaneously accessed and shared by many users. Proxy caches often operate on dedicated hardware which is generally dedicated systems with fast processors with 5 - 50 GB of disk space, and 64 - 512 MB of RAM.

Proxy servers are usually operated much like any other network service (e-mail, Web servers, DNS). These servers lie between the boundary of the client and the remote web server, operating at the edges of the ISP or the enterprise network. The requests for connection to the remote server passes through it, which makes it to act as a 'Server' itself and it even acts as a Client, when the requested resource is not found within its' repository and a request is sent to the remote web server for the same resource.

Software and Hardware approaches

Even today most used caching mechanisms are Software based, such as Microsoft's and Netscape's proxy servers running on UNIX and Windows NT.

However such softwares require high-end computing capabilities or the burden has to be borne by the OS since 'software caching' consumes a lot of machines' resources.

On the contrary 'hardware caching' is all together different since these cache servers are embedded into the hardware and run as '*sealed packages*'.

Network Appliance' s NetCache, for example, is based on a 64-bit AlphaCPU and a proprietary RTOS, which provides a four-fold performance improvement over software-only solutions running on standard 32-bit NT-based general-purpose servers.

Cobalt Micro server introduced its CacheQube recently, built around a 64-bit MIPS RISC and customized version of the public-domain Linux operating system, which has been optimized for real-time caching transactions.

3.2.6.3.3 Various Caching techniques

There are five caching techniques in use as mentioned below:

- Cache update algorithms
- Hierarchical caching
- Transparent caching
- Load balancing
- Replication and/or Pre-fetching

Cache update algorithms

There is always a trade-off between "correct values" to be served to the client and the 'efficiency' to be attained. To ensure that correct values have been served, the cached values need to be matched with the 'Expiry tags' date' and if values are returned without checking for expiry, the chances are high that 'old content' may be served although overall performance will increase.

Some of the possible ways have been enlisted, which help to ensure that the content has not expired.

Method 1

Every time the cache sever receives some web objects, it firsts calculates the expiry time of the object from the response header.

- It looks for an "Expires:" line in the HTTP response header
- If configured, it looks for a "Last-Modified" header field and calculates an expiry time as a fraction of the time since last modified. For example, if a document was modified ten days ago, and the last modified factor is set to 0.1, then the document would expire in one day.

Method 2

If available, the caching server should be configured with a "cache refresh setting" (a specific time interval before doing the up-to-date check) to reduce latency. In this approach, whenever an up-to-date check takes place, the cache can submit a GET If-Modified-Since request to the remote HTTP server. The cache sends the content of the Last-Modified header that was stored as an "If-Modified-Since:"

header and the cache can use the time a file was last modified to estimate how long a document is likely to remain unchanged.

Hierarchical caching

A single Web cache will reduce the amount of traffic generated by the clients behind it. Similarly, a group of Web caches can benefit by sharing another cache in much the same way (hierarchically).

In this technique caching server checks if the next caching server in line, or the requested object's origin server, is reachable and able to complete the request. If a caching server is part of a mesh, it must be able to redirect requests for documents, which normally would be directed to a caching sever higher in the cache hierarchy (a so called parent).

Transparent Caching

Distributed caching solution automatically and transparently directs Web traffic to high-performance cache servers. Using this kind of solution, ISPs can dramatically improve user response times and reduce wide area network bandwidth costs or more with minimum configuration and administration requirements.

Transparent caching allows network administrators to easily deploy caches anywhere in the network without the need to modify end-user browsers or other software. This improves Internet response time and service availability, and reduces WAN operating costs by redirecting web traffic destined to remote Internet hosts to a group of local cache servers.

Load Balancing

A caching service should be able to spread the load over a number of caching servers. A single caching server is a bottleneck in a limit to the number of clients that can use the same cache. A round-robin Domain Name Service (DNS) can be used to distribute the access to several similar configured caching servers.

Web clients using HTTP rely heavily on the DNS. If web-clients cache the result of a host name lookup, in order to achieve a performance improvement, it must observe the TTL information reported by the DNS.

Replication and/or pre-fetching

Replication and/or pre-fetching in combination with caching may give a major performance improvement.

Adding a mirror (replication) of popular sites will increase the cache hit rate significantly. Pre-fetching (read ahead but less duplication than in case of mirroring) of popular pages gives a similar effect. A possible strategy for pre-fetching is the following: identify pages in your cache that are "warm" (have had hits since fetching), but have since expired. Re-fetch them and cache them.

3.2.7 Components of a Web Browser

A web browser must play several different roles in bringing web-pages to the user and displaying the appropriate information. The web-browser communicates with the web-servers and DNS servers, in order to fetch the information. Then, it must translate the raw information and organize it into web-pages. Later, it must render the web-pages in the form of display screens.

A web browser can thus be divided into three basic components:

- HTTP Client
- Layout engine
- Rendering engine

3.2.7.1 HTTP Client

The browser at the client end is called HTTP-Client. Interaction between HTTP-Client and Server (i.e. Request-Response session) proceeds in following steps:

- First a TCP connection is established with the host (server).
- Next, browser (HTTP Client) sends a '*HTTP-Request*' to the server.
- Finally the web server sends a '*HTTP-Response*'.

3.2.7.1.1 HTTP-Requests and Response

The following section, describes HTTP-Request and HTTP-Response in detail.

HTTP Request

A request message from a client to a server includes within the first line of that Message – the method to be applied to the resource, the identifier of the resource, and the protocol version in use. For backwards compatibility with the more limited HTTP/0.9 protocol, there are two valid formats for an HTTP request:

Request = Simple-Request | Full-Request

Simple-Request = "GET" SP Request-URI CRLF

**Full-Request = Request-Line
(General-Header
| Request-Header
| Entity-Header)
CRLF
[Entity-Body]**

If an HTTP/1.0 server receives a Simple-Request, it must respond with an HTTP/0.9 Simple-Response. An HTTP/1.0 client capable of receiving a Full-Response should never generate a Simple-Request.

Request-Line

The Request-Line begins with a method token, followed by the Request-URI and the protocol version, and ending with CRLF. The elements are separated by SP characters (i.e. space). No CR (Carriage Return) or LF (Line feed) are allowed except in the final CRLF sequence (this marks the end of request line).

Request-Line = Method SP Request-URI SP HTTP-Version CRLF

Method:

The Method token indicates the method to be performed on the resource identified by the Request-URI. The method is case-sensitive.

Method = "GET"
| "HEAD"
| "POST"
| extension-method

extension-method = token

The list of methods acceptable by a specific resource can change dynamically; the client is notified through the return code of the response if a method is not allowed on a resource. Servers should return the status code 501 (not implemented) if the method is unrecognized or not implemented.

Request-URI:

The Request-URI is a Uniform Resource Identifier and identifies the resource upon which to apply the request.

Request-URI = absoluteURI | absolute_path

The two options for Request-URI are dependent on the nature of the request.

The absoluteURI form is only allowed when the request is being made to a proxy. The proxy is requested to forward the request and return the response. If the request is GET or HEAD and a prior response is cached, the proxy may use the cached message if it passes any restrictions in the Expires header field. Note that the proxy may forward the request on to another proxy or directly to the server specified by the absoluteURI. In order to avoid request loops, a proxy must be able to recognize all of its server names, including any aliases, local variations, and the numeric IP address.

An example for a Request-Line would be:

GET http://www.w3.org/pub/WWW/TheProject.html HTTP/1.0

The most common form of Request-URI is that used to identify a resource on an origin server or gateway. In this case, only the absolute path of the URI is transmitted. For example, a client wishing to retrieve the resource above

directly from the origin server would create a TCP connection to port 80 of the host "www.w3.org" and send the line:

GET /pub/WWW/TheProject.html HTTP/1.0

followed by the remainder of the Full-Request. Note that the absolute path cannot be empty; if none is present in the original URI, it must be given as "/" (the server root).

General-Header Fields

There are a few header fields which have general applicability for both request and response messages, but which do not apply to the entity being transferred. These headers apply only to the message being transmitted.

**General-Header = Date
| Pragma**

General header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields may be given the semantics of general header fields if all parties in the communication recognize them to be general header fields. Unrecognized header fields are treated as Entity-Header fields.

Request Header Fields

The request header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers, with semantics equivalent to the parameters on a programming language method procedure invocation.

**Request-Header = Authorization
| From
| If-Modified-Since
| Referer
| User-Agent**

Request-Header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields may be given the semantics of request header fields if all parties in the communication recognize them to be request header fields. Unrecognized header fields are treated as Entity-Header fields

Entity Header Fields

Entity-Header fields define optional meta-information about the Entity-Body or, if no body is present, about the resource Identified by the request.

**Entity-Header = Allow
| Content-Encoding
| Content-Length
| Content-Type
| Expires
| Last-Modified
| extension-header**

extension-header = HTTP-header

The extension-header mechanism allows additional Entity-Header Fields to be defined without changing the protocol, but these fields cannot be assumed to be recognizable by the recipient. Unrecognized header fields should be ignored by the recipient and forwarded by proxies.

Entity Body

The entity body (if any) sent with an HTTP request or response is in a format and encoding defined by the Entity-Header fields.

Entity-Body = OCTET <any 8-bit sequence>

An entity body is included with a request message only when the request method calls for one. The presence of an entity body in a request is signaled by the inclusion of a Content-Length header field in the request message headers. HTTP/1.0 requests containing an entity body must include a valid Content-Length header field.

For response messages, whether or not an entity body is included with a message is dependent on both the request method and the response code. All responses to the HEAD request method must not include a body, even though the presence of entity header fields may lead one to believe they do. All 1xx (informational), 204 (no content), and 304 (not modified) responses must not include a body. All other responses must include an entity body or a Content-Length header field defined with a value of zero (0).

HTTP Response

After receiving and interpreting a request message, a server responds in the form of an HTTP response message.

Response = Simple-Response | Full-Response

Simple-Response = [Entity-Body]

**Full-Response = Status-Line
*(General-Header
| Response-Header
| Entity-Header)
CRLF
[Entity-Body]**

A Simple-Response should only be sent in response to an HTTP/0.9 Simple- Request or if the server only supports the more limited HTTP/0.9 protocol. If a client sends an HTTP/1.0 Full-Request and receives a response that does not begin with a Status-Line, it should assume that the response is a Simple-Response and parse it accordingly. Note that the Simple-Response consists only of the entity body and is terminated by the server closing the connection.

Status-Line

The first line of a Full-Response message is the Status-Line, consisting of the protocol version followed by a numeric status code and its associated textual phrase, with each element separated by SP characters. No CR or LF is allowed except in the final CRLF sequence.

Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF

Since a status line always begins with the protocol version and status code

"HTTP/" 1*DIGIT "." 1*DIGIT SP 3DIGIT SP

(e.g., "HTTP/1.0 200 "), the presence of that expression is sufficient to differentiate a Full-Response from a Simple-Response. Although the Simple-Response format may allow such an expression to occur at the beginning of an entity body, and thus cause a misinterpretation of the message if it was given in response to a Full-Request, most HTTP/0.9 servers are limited to responses of type "text/html" and therefore would never generate such a response.

Status Code and Reason Phrase

The Status-Code element is a 3-digit integer result code of the attempt to understand and satisfy the request. The Reason-Phrase is intended to give a short textual description of the Status-Code. The Status-Code is intended for use by automata and the Reason-Phrase is intended for the human user. The client is not required to examine or display the Reason-Phrase.

The first digit of the Status-Code defines the class of response. The last two digits do not have any categorization role. There are 5 values for the first digit:

- 1xx: Informational - Not used, but reserved for future use
- 2xx: Success - The action was successfully received, understood, and accepted.
- 3xx: Redirection - Further action must be taken in order to complete the request.
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled.
- 5xx: Server Error - The server failed to fulfill an apparently valid request

The individual values of the numeric status codes defined for HTTP/1.0, and an example set of corresponding Reason-Phrase's, are presented below. The reason phrases listed here are only recommended -- they may be replaced by local equivalents without affecting the protocol.

Status-Code = "200" ; OK
 | "201" ; Created
 | "202" ; Accepted

```
| "204" ; No Content
| "301" ; Moved Permanently
| "302" ; Moved Temporarily
| "304" ; Not Modified
| "400" ; Bad Request
| "401" ; Unauthorized
| "403" ; Forbidden
| "404" ; Not Found
| "500" ; Internal Server Error
| "501" ; Not Implemented
| "502" ; Bad Gateway
| "503" ; Service Unavailable
| extension-code
extension-code = 3DIGIT
```

HTTP status codes are extensible, but the above codes are the only ones generally recognized in current practice. HTTP applications are not required to understand the meaning of all registered status codes, though such understanding is obviously desirable. However, applications must understand the class of any status code, as indicated by the first digit, and treat any unrecognized response as being equivalent to the x00 status code of that class, with the exception that an unrecognized response must not be cached.

For example, if an unrecognized status code of 431 is received by the client, it can safely assume that there was something wrong with its request and treat the response as if it had received a 400 status code.

Response Header Fields:

The response header fields allow the server to pass additional information about the response which cannot be placed in the Status-Line.

These header fields give information about the server and about further access to the resource identified by the Request-URI.

```
Response-Header = Location
                  | Server
                  | WWW-Authenticate
```

Response-Header field names can be extended reliably only in combination with a change in the protocol version. However, new or experimental header fields may be given the semantics of response header fields if all parties in the communication recognize them to be response header fields. Unrecognized header fields are treated as Entity-Header fields.

3.2.7.2 Layout Engine

The layout engine provides a set of components which serve in the transformation process as a document moves from source to target form. We refer to these objects as components because they are combined dynamically at runtime to achieve the transformation. By substituting a different set of components, you can perform alternate transformations.

The Major components of a Layout Engine are:

- Scanner
- Parser
- DTD
- Sink

3.2.7.2.1 Scanner

The first major component in the layout engine is the Scanner. The Scanner provides an incremental "push-based" API that offers methods for accessing characters in the input stream (usually a URL), finding particular sequences, collating input data and skipping over unwanted data.

3.2.7.2.2 Parser

The second major component in the layout engine is the parser. The parser component controls and coordinates the activities with other components in the system. This approach relies upon the fact that regardless of the form of the source document, the transformation process remains the same (as we'll explain later). While other components of the system are meant to be dynamically substituted according to the source document type, it is rarely necessary to alter the parser component. The parser also drives tokenization. Tokenization refers to the process of collating atomic units (characters) in the input stream into higher level structures called *tokens*.

For example, the HTML tokenizer converts a raw input stream of characters into HTML tags. For maximum flexibility, the tokenizer makes no assumptions about the underlying grammar. Instead, the details of the actual grammar being parsed are up to the DTD object that understands the constructs that comprise the grammar. The importance of this design decision is that it allows the engine to dynamically vary the language it is tokenizing without changing the tokenizer itself

3.2.7.2.3 DTD

The final component in the parser engine is the DTD, which describes the rules for well-formed and/or valid documents in the target grammar. In HTML, the DTD declares and defines the tag set, the associated set of attributes and the hierarchical (nesting) rules of the HTML tags. Once again, by separating the DTD component from the other components in the parser engine, it becomes possible to use the same system to parse a much wide range of document types. Simply put, this means that the same parser can provide input to the browser, biased (via the DTD) to behave like Navigator, IE, or any other HTML browser. The same can be said for XML.

3.2.7.2.4 Sink

Once the tokenization process is complete, the parser engine needs to emit its content (tokens). Since the parser doesn't know anything about the document model, the containing application must provide a "content-sink". The sink is a simple API that accepts a container, leaf and text nodes, and constructs the underlying document model accordingly. The DTD interacts with the sink to cause the proper content-model to be constructed based on the set of input tokens.

3.2.7.3 *Rendering Engine*

This is the last module, and probably the most important one. This is responsible for displaying the contents on the screen, using *Graphic Primitives* from the system (System Calls). This includes standard graphical primitives such as lines, rectangles, bitmaps, text, etc.

There are basically two separate pieces to the graphics system:

- **Platform independent** set of structures, classes and interface definitions. These can be broken down as follows:
 - ♣ Geometrical primitive management (rectangle, point, margin, size, transformation, etc.).
 - ♣ Color definitions.
 - ♣ Font specification.
 - ♣ Image abstraction.
 - ♣ Interfaces to be implemented per-platform.
 - Rendering Context
 - Device Context
 - Font Metrics
 - Image
 - Region
 - Alpha blender
- **Platform dependent** implementation of the per-platform interfaces listed above.

3.2.7.3.1 *Implementation Details*

Geometrical primitives

This is a collection of classes to represent 2D objects such as points, margins, lines, rectangles and sizes. In addition there is a transformation class that knows how to deal with scale, translation and matrix concatenation operations. Most classes support operator overloading to allow mathematical operations to be expressed succinctly

Color definitions

All colors are defined as RGB values in the range 0-255 bit shifted to fit into a 32 bit unsigned word for efficiency. As a result, a certain amount of type safety is sacrificed. Macros are provided for packing and unpacking the various color components. There are also a set of lookup tables to go from named colors to RGB values.

Font specification

The font structure holds all of the information necessary a particular typeface. This includes:

- family name
- style
- variant

- weight
- decorations
- size

The font structure is used in conjunction with the device context to request a set of font metrics. The device context has a cache of the various font metrics that have been requested to date and creates new ones from fonts when a new font specification is encountered

Image abstraction

This is a set of interfaces and classes that interact with the image library to convert images from URLs to platform independent image data. To convert to actual bitmaps that can be rendered, the platform specific image class is necessary.

- An ImageGroup allows a user to manipulate and observe a collection of image loading requests.
- ImageRequests can be manipulated and observed individually.

Rendering Context

The rendering context is where all other pieces of the graphics library come together to actually display bits on an output device. As such, the rendering context implementation is unique to each type of output device. The rendering context, in addition to converting geometrical primitives to bits, also does state management including:

- font
- transformation
- clip rectangle
- color
- state stack
- internal state as necessary per-platform
- string length measurement

Since the rendering context is the focus point for manipulating bits, there are additional methods for creating off-screen drawing surfaces and managing double buffering

Device Context

In order to interface with the underlying system's graphics and UI (user interface) facilities, device context was created. The device context is essentially a set of methods for querying the system about such things as:

- output device resolution in terms of logical units
- conversion functions to go from application defined units to device units
- display zoom value
- font specification to metrics conversion
- creation of rendering contexts suitable for the device
- gamma correction

A device context is designed to be shared among all of the higher level objects that intend to output to the device described by the device context.

Font Metrics

Font metrics contain all of the properties of the glyphs of a typeface. Behind the font metrics interface is a platform dependent implementation that may represent vector based fonts such as Adobe Type 1, True Type, bitmap fonts or anything else that the output device is capable of rendering. The font metric methods are typically used by things such as a layout system to perform text measurement.

Image

Bitmaps usually have a very platform specific representation. The Image class is an encapsulation of the underlying native platform bitmap structures. Images can be either 8 or 24 bits per pixel on the XP side of the interface (whatever is closest to the native format) and can have any representation necessary on the implementation side of the interface. In addition to holding an array of bytes or triples representing pixel information, images also have an optional alpha channel and colormap for 8 bpp images. Gamma of the image color information can be queried. Since some platforms can render bitmaps faster if the bitmap is converted to an "opaque" format that only the underlying hardware understands, there are methods for optimizing a bitmap and querying for optimized status.

Regions

A region is a data structure that describes a masked area through which drawing is permitted in a rendering context. Regions can be created implicitly through various calls to the rendering context's `SetClipRect()` method, or explicitly by creating a region and adding or removing rectangles from the region.

Alpha blender

In order to allow for translucency effects, there is an object called a Blender that knows how to take two bitmaps, scale one by a value between 0 and 1, scale the other by the inverse of that value and add them together.

3.2.8 Working of web browser

When the user requests for a desired resource, the request sent is in a standard form accepted all over the network. The request generally includes: the **Protocol version**, **Path** of the resource asked for (any file, multimedia resource, or any other thing), **Method** used to ask for the requested resource. This request is referred as “script”.

The browser examines the script to figure out the location of requested resource. If the resource is on the internet, the browser uses the domain name to find the actual IP (numeric) address. This involves asking the Domain Name Server (DNS). If the DNS server does not respond, or the name isn't in its list, the user gets an error message and the process stops dead.

Before communicating the two, Client browser and the Server establish a connection (i.e. a Port is assigned to the client where it can send its request & get the response back from the browser, by the Server).

The browser contacts the remote Web server host (if the file requested is not on the local computer). A TCP packet is made up and sent through layers 3, 2, and 1 requesting to set up a session on the Web port (usually 80). If the host responds that it is able to create a session on the desired port (layer 5), the browser sends an HTTP request (layer 6) to ask for the file the user has requested for.

If the file is available with the server, the Web server (layer 7 on the remote computer) responds back to the client, along with the **Protocol version** in which it is replying and the **Status code** (which helps the client to know if the resource was found or not, or whether any other error occurred or not). After this response header the data is sent along with appropriate **Content-Type** (MIME type) which helps the client's browser to invoke the required routine or any other application which is capable of interpreting the sent information & then displaying the same using **System primitives**.

Once the browser receives the data it processes it in order to make the received data readable by the client. This happens for both local and remote files. The browser identifies HTML tags and interprets them, which itself invokes many routines dynamically according to the content to be displayed (say, some of the components will always be required to work with, as the HTML interpreter itself. Whereas the Gif interpreter could be loaded at run-time if any Gif data is sent along with HTML data). The HTML code received often contains a number of other files as inline code. The browser makes additional requests with the remote host, one per file.

The interpreted data is then passed to other modules of the browser itself, which could be responsible for checking the Semantics of the parsed data or sending it to the Display primitives to draw the bits on the output device. Some other routines could be invoked for changing the Font specification, Image Rendering, or say Double-Buffering routine could be called for reducing the flickering on the output screen.

When the browser has requested all the secondary images in the HTML text, it sends a packet advising the remote Web host that it is closing its Session Layer. This way, the host is free to process other requests while the user reads the web page displayed on his/her screen.