# 2. Hardware, Software, File System, and Programming Languages

## 2.1 Hardware and Software

The resources provided by a computing system may be grouped into two broad categories (Lane and Mooney 2001):
- **Physical resources** (also called hardware resources)
- **Logical resources** (also known as software resources)

As the name implies, physical resources are the permanent physical components of the computer system. The principal physical resources are:

- A processor (sometimes more than one)
- Main memory
- Input and output (I/O) devices such as screens, keyboards, printers, or network devices
- Secondary storage devices such as disks and tapes
- Internal devices such as clocks and timers.

Logical resources are collections of information, such as data or programs. Logical resources must be stored within physical resources, for instance, within main or secondary memory. Resources of interest in this category may include:

- Units of work, such as jobs or interactive sessions
- Processes (programs in execution)
- Files (named sets of information)
- Shared programs and data
- Procedures that perform a variety of useful services.

The resources that must be managed by a typical general-purpose operating system are summarized in Table 1.

Table 1. Computer resources managed by an OS.

| **PHYSICAL RESOURCES** | **LOGICAL RESOURCES** |
|---|---|
| processor | jobs and sessions |
| main memory | processes |
| I/O devices and controllers | files |
| secondary storage | system services |
| timers and clocks | |

The resource management provided by an OS has two principal objectives. The first objective is to support **convenient use**. In the early days of computing, each user had to provide his or her own programs to manage each detail in the control and use of I/O devices and other common resources. Today's users of computers, by and large, are not programmers. They view the computer as a tool for saving their own valuable time, and they rightly expect such tools to be easy to use. They should not have to worry about complex details involved in resource management. The services provided to the users of today's computing systems must be easy to use, yet provide sufficient power and flexibility to meet each user's needs.

The second objective of resource management by an operating system is to support and enforce **controlled sharing**. When computer systems are shared by more than one user, programs must be prevented from interfering with each other in any way. Especially in a general-purpose computing environment, programs of any type may be run, including many which contain errors or try to do improper things. Allowing a program to have direct control of common resources without the supervision of an OS would place other programs at risk of interference or damage to their code or resources. Controlled sharing of resources is a complex problem. Besides being properly controlled, the sharing must be *efficient* and *fair*. Resources should seldom have to wait unused when there are programs that need them. When a resource is needed by several programs, each one should get a fair turn.

In general, we restrict our view of the role of the OS to direct management of resources which must be controlled by a central authority for the benefit of all users. We deviate from this narrow view, however, in including treatment of two subjects which could arguably be excluded: file systems and the user interface.

Although files are not pure physical resources, they comprise an important category of shareable logical resources formed from physical storage devices, such as magnetic disks. The design of a file system imposes on all users the same high-level view of these storage devices. In some current operating systems, especially those designed to control multiple computers connected by a network, a file system is treated as a separate and independent component. However, files are an important system resource expected by most computer users, and we follow widely accepted practice in studying them together with operating systems.

## 2.2 Responsibilities of an OS

To fulfill the role outlined in the previous section, an operating system must meet many responsibilities. Some of these, such as file management, are concerned with managing particular categories of resources or interfaces. Others are concerned with services that span a range of resources, such as error handling or security control. The major responsibilities that most operating systems must meet are as follows:

(1) The User Interface

The user interface provides a mechanism for direct interaction between users and the OS, or between users and other programs. This interaction may be based on typing lines of text, selecting menu items or graphic icons, speaking phrases, or some other appropriate paradigm. A closely related mechanism is the program interface, which provides services upon request to programs during execution. These services, requested by a mechanism similar to procedure calls, include such things as input and output, file access, and program termination requests.

The **user interface** establishes the "user friendliness" of the system. Commands or graphic elements that are consistent and easy to use will result in a user's more rapid acceptance of a computer system. Structures that are difficult to learn or understand quickly frustrate users and often cause them to move to other computer systems that are easier to use, whenever such systems are available. Although in some environments the user interface may be viewed as a separate, replaceable program, it is a critical element of a complete computing system.

The **program interface** defines the procedures and conventions programs must follow to request operating system services. Such things as the parameters required and the means of invoking the services of the operating system are defined by this interface. The program interface is often known as the **application program interface (API)**.

(2) Process Management

The term **process** is used to describe a program in execution under the control of an operating system. Another widely used term for process is **task**. We will use the term process consistently throughout this text. Perhaps the most fundamental responsibility of any operating system is **process management**. When there is only one CPU, processes must either run one at a time to completion, or take turns using the CPU for a short period. Allowing processes to take turns in this manner is called **interleaved execution**; processes that share the CPU in this way are called **concurrent processes**. An OS which allows application programs to concurrently share the CPU is called a **multiprogramming system**.

Controlling this interleaved execution so that work proceeds in a fair and effective way is the problem to be solved by **process scheduling**. This scheduling must be considered at several levels, from overall admission of new jobs into a system down to moment-by-moment allocation of the use of the CPU.
For various reasons, concurrent processes may interact with one another. One form of interaction occurs because resources are shared, and some resources required by one process may be in use by another. A second form of interaction occurs when processes deliberately seek to exchange information. Controlling the interaction that may occur among concurrent processes is an important responsibility for a multiprogramming OS.

(3) Device Management

All computers have a variety of input, output and storage devices that must be controlled by the operating system. Printers, terminals, plotters, hard disks, floppy disks, magnetic tapes, and communication devices are among the most common devices found in such systems today. In the past, additional devices such as punched card readers and paper tape readers were also prevalent.

In addition, many computers include internal devices that may be accessed to perform special functions. An important example is clocks or timers, which may be used to measure the timing relationships among various events. **Device management** encompasses all aspects of controlling these devices: starting operation, requesting and waiting for data transfers, responding to errors that may occur, and so on.

As might be suspected, the varieties of devices that can be attached to a computer use many different commands and controls. The user cannot be expected to write the software to directly control such a wide range of devices. Furthermore, multiprogramming operating systems that support the sharing of a computer system by two or more processes cannot allow an application program to access these devices without going through the operating system program interface. Otherwise, the integrity of other users' and programs' data might be sacrificed (e.g., by a direct write request to a disk that intentionally or erroneously writes over another user's data). Device management, therefore, is a very important function of the operating system. Device support can be quite complex, and implementing such support consumes much of the time and effort in the implementation of any operating system.

(4) Time Management

Another important OS responsibility is **time management**, that is, controlling the time and sequence for various events. A special category of I/O device is the **timer**, whose role is to measure time and cause events to occur at specific times.

(5) Memory Management

Memory management is the responsibility of controlling the use of main memory, a critical resource in any computer system. Since secondary storage devices, such as disks and tapes, are used in various ways to extend the storage available in main memory, the subject of memory management must deal with these devices as well. The performance of operating systems can be greatly affected by the way the memory is managed.

Early operating systems allowed only one program to be in progress at a time, so memory management was relatively simple. The operating system used what it needed of main memory to store its own information, and the program could have the rest. Responsibility for managing the available memory rested solely with the application program. Often the OS had no means of protecting even its own memory. If a program interfered with OS memory, it could do harm only to itself, since a failure of the operating system would harm only that single program.

The introduction of multiprogramming systems changed all that, because main memory had to be shared among several processes. In such systems the OS must be responsible for allocating memory and for protecting processes from one another.
Early forms of memory management were concerned primarily with allocating portions of main memory to each process when the process started. Newer strategies allow additional areas of memory to be allocated and removed as desired, and provide for temporary **swapping** of programs and their data from main memory to a disk when not immediately needed. This strategy makes the memory space available for more immediate needs.

As an evolution of swapping techniques, **virtual memory** systems have now become common. These systems can automate the swapping process by a combination of hardware and software techniques, providing powerful solutions to a full range of memory management problems.

(6) File Management

Information stored by a computer system is usually organized into files. File management is concerned with all aspects of reading, writing, organizing, and controlling the access to information stored in files. Another major function of such management is to provide security, that is, protection of information from improper access. Security has been one of the weakest aspects of file management in many operating systems.

(7) Job and Session Management

A complete unit of work performed or submitted by a user may include a series of programs, carried out by a set of processes that may proceed one at a time, or concurrently in some cases. Such a unit is called a job when submitted to the OS all at once, as was common in early "batch" systems. When a set of work is specified by a series of commands typed by a user at a terminal, it may more aptly be called a session. For various reasons, it is sometimes desirable to manage jobs and sessions as complete units. For example, the OS may need to decide when a new job or session can be started, and it may need to associate the complete job or session with a particular user for authorization or accounting purposes.

(8) Error Handling

Operating systems must deal with a variety of errors that can occur in a computer system. Such errors include hardware errors in devices, memory, and channels, and software errors in application and system programs. Software errors can be a result of such things as arithmetic overflow, invalid instructions, or references to invalid memory. Error handling is the process of detecting and recovering from such errors. Effective error handling must try to detect as many types of errors as possible. When errors are detected, reasonable attempts should be made to recover and continue, without such drastic actions as terminating the offending process or stopping the entire system. Whenever possible, failures in hardware components should result in the isolation of the faulty component without affecting the rest of the system operation. Software errors should be recognized, corrected if possible, and prevented from causing damage to other programs and resources. In many cases, error detection and recovery should be as invisible as possible to the user. However, some types of errors should be reported to the application program so that appropriate action can be taken.

(9) Reliability and Security

An operating system may have partial responsibility for ensuring that computer systems continue to operate correctly and data and programs are protected from damage or improper access. These goals may need to be met even when errors or failures exist in hardware or software, or unauthorized persons deliberately seek to corrupt or gain access to the system. These system responsibilities are known as reliability and security. Their importance can vary greatly from one system to another. Providing extremely good reliability and security can be very costly.

The importance of reliability of keeping a computer system running correctly is evident. The importance of ensuring the security of information stored in files has already been mentioned. In addition to this critical aspect of security, the operating system must protect one process from another, the operating system from a process, a process from the operating system, and the operating system from itself. The most obvious way an OS does this is to isolate memory areas for processes and for itself; another is to protect against the use of privileged instructions and system resources by unauthorized processes. Access to OS services must also be controlled by the OS. Other areas of security include those of session and job management, which require the appropriate identifications, passwords and other mechanisms before use of the system is granted.

(10) Monitoring and Accounting

Most sophisticated operating systems, especially in environments where computing is considered to have significant costs, include some facilities for monitoring the behavior of the system as it proceeds and for keeping records of that behavior for various purposes. One important purpose of this monitoring is accounting, keeping track of resource use by each user so that users can be billed for the services of the computing system. Another important purpose of monitoring is to measure the performance of the system, that is, the effectiveness with which it completes its work within the limitations of available resources. Such monitoring may identify adjustments that should be made to keep that performance as good as possible.

(11) System Management

Some key responsibilities that fall under the heading of system management include methods for initial installation of operating systems; adjusting their characteristics to each particular environment; resuming normal operation after a shutdown or a serious error; maintaining records of system usage by each job or user; maintaining user information and other sensitive records; and providing appropriate responses to any unusual situations that may occur.

An important goal in real operating systems is to maintain the best possible performance. This leads to the need to measure or predict performance and to use this information to tune the system for the best performance that is practically attainable.

## 2.3 File System

### 2.3.1 File management

For most users, the **file system** is the most visible aspect of an OS. A collection of information maintained on long-term storage for a set of users is called a file system. The file system consists of two distinct parts: a collection of **files**, each storing related data, and a **directory** structure, which organizes and provides information about all the files in the system.

The portion of an operating system responsible for managing a file system, the **file manager,** must provide users and programs with a suitable set of operations to maintain and use files. Many of the basic operations appear as services at the program interface. Programs must be able to read information from files, and modify or add to file contents. In most cases files may be created when needed and destroyed when no longer useful. Various attributes and characteristics may be associated with files; we require methods to examine and change these as needed.

Some operating systems provide much more elaborate services for managing special types of files. These may include support for indexed files, large text documents, database systems, and other categories. In other OSs only basic file support is available; other structures must be managed by application programs.

(1) File concept

Computers can store information on several different storage media, such as magnetic disks, magnetic tapes, and optical disks. The operating system provides a uniform logical view of information storage and abstracts from the physical properties of its storage devices to define a logical storage unit (the file). Files are mapped, by the operating system, onto physical devices.

A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage. The information in a file is defined by its creator. Many different types of information may be stored in a file – source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on.

(2) File attributes

A file is named, for the convenience of its users, and is referred to by its name. A name is usually a string of characters, such as example.doc, example.c. In some systems, name is case sensitive. A file has certain other attributes, which vary from one operating system to another, but typically consist of these:

- Name: The symbolic file name is the only information kept in human readable form.
- Identifier: This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.

- Type: This information is needed for those systems that support different types.
- Location: This information is a pointer to a device and to the location of the file on that device.
- Size: The current size of the file (in bytes, words, or blocks), and possibly the maximum allowed size are included in this attribute.
- Protection: Access-control information determines who can do reading, writing, and executing.
- Time, date, and user identification: This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

(3) File operations

A file is an abstract data type. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.

- Creating a file: Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory. The directory entry records the name of the file and the location in the file system, possibly other information.
- Writing s file: To write a file, we make a system call specifying both the name of the file and the information to be written to the file.
- Reading a file: To read a file, we use a system call that specifies the name of the file and where (in memory) the text block of the file should be put.
- Repositioning within a file: The directory is searched for the appropriate entry, and the current-file-position is set to a given value. This file operation is also known as a file seeks.
- Deleting a file: To delete a file, we search the directory for the named file. Having found the associated directory entry, we delete all file space.
- Truncating a file: The user may want to erase the contents of a file but keep its attributes.

(4) File types

A common technique for implementing file types is to include the types as part of the file name. The name is split into two parts – a name and an extension, usually separated by a period character. Typical file types in MS-DOS and Windows are summarized in Table 2.

Table 2. Common file types (Source: Silberschatz et al. 2002).

| File type | Usual extension | Function |
|---|---|---|
| executable | exe, com, bin, or none | read to run machine-language program |
| object | obj, o | compiled, machine language, not linked |
| source code | c, cpp, java, bas, frm | source code in various languages |

| | | |
|---|---|---|
| batch | bat, sh | commands to the command interpreter |
| text | txt, doc | ASCII text data, documents |
| word processor | wp, txt, rtf, doc | various word processor formats |
| library | lib, dll | libraries of routines for programmers |
| print or view | zip, tar | ASCII or binary file in a format for printing or viewing |
| archive | zip, tar | related files grouped together into one file, sometimes compressed for archiving storage |
| multimedia | mpeg, mov, avi | binary file containing audio/video information |

(5) File Naming

To allow external users and programs to identify files, every file is assigned a name. File names play a critical role in a file system. Given a name, the file manager must be able to rapidly locate the file, if it exists. File names are usually stored in each file descriptor, and they serve as a key by which the complete descriptor can be retrieved. File names are usually defined as sequences of characters represented by a character code, such as ASCII. While users wish to define names as a convenient indicator of the contents of the file, each file system imposes different rules about how names may be constructed. The choice of rules can have an impact on both the usefulness of names and the efficiency with which they can be stored and manipulated. Some issues to be considered in selecting a structure for file names include types of components, length of names, and permissible characters.

(6) File name components

Although some file systems employ names that are simple character strings with no meaningful structure, in most cases a name is made up of several distinct parts, separated by appropriate punctuation. Some examples of complete structured file names are:

MS-DOS: A:\BOOK\CHAPTERS\FILEMGT.TXT
Windows: C:\Book\Chapters\File Management
UNIX: /usr/jwang/book/chapters/filemgt.t

In most of these examples, the main file name is the same: filemgt. The Windows example uses File Management, a typically longer and more descriptive name. Some components, usually written to the left of the name, identify groups of files to which the specified file belongs. Usually most or all of these components may be omitted if there is an understanding about what the "current" group is. The meaning of some of these components may include:

- *node name.* In a file system spanning a number of computers connected by a network, the node name identifies the network node where the file is located.
- *device name* or *volume name.* This component identifies the volume or storage device on which the file is found. Either a physical name, such as "disk1" or a logical volume name, such as "book" might be used.  Other examples are A in the MS-DOS example.
- *directory name.* If a file system supports multiple directories, this component identifies the directory in which the file resides. In some cases there are subdirectories within directories. UNIX has multiple directories but does not distinguish subdirectories. In the above UNIX example, all of the names usr, jwang, book, and chapters are considered directory names. Moreover, the first slash represents a special home directory with an empty name.

(7) File name length

The choice of length restrictions on file names must balance several factors. On one hand, many users would prefer file names that are long and descriptive. Especially when one directory contains many files, names must be long enough to clearly distinguish each file and avoid conflicts.

On the other hand, searching of directories will be most efficient if file descriptors are of fixed length. To keep the size of each descriptor (which includes the name) reasonable, name length must be limited. Apart from this concern, it is difficult for programs to allocate space for file names if the names (perhaps including all components) may be arbitrarily long. This has been a difficulty, for example, with some versions of UNIX and Macintosh.

Most OSs impose a maximum length on file names. Typical values for this maximum include 6 (RT-11), 8 (CP/M and MS-DOS), 9 (VMS), 14 (some versions of UNIX), 31(Macintosh), 32 (DOS-99), 44(OS/MVT). A few, such as EXEC on UNIVAC, have required an exact number of characters in all names.

These limits apply to the main file name component only. In many cases, the file type component is limited to three characters. Directory names usually must adhere to the same rules as file names. Other components tend to have a very restricted structure.

## 2.3.2 File access methods

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files while other systems support many access methods.

### (1) Sequential access

It is the simplest access method. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion. Sequential access is based on a tape model of a file.

### (2) Direct access

Direct access is also called relative access. A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order. The direct access method is based on a disk model of a file, since disks allow random access to any file block. Direct access files are of great use for immediate access to large amounts of information. Databases are often of this type.

### (3) Other access methods

Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file. The index, like an index in the back of a book, contains pointers to the blocks. To find a record in the file, we first search the index, and then use the pointer to access the file directly and to find the desired record (Figure 1).

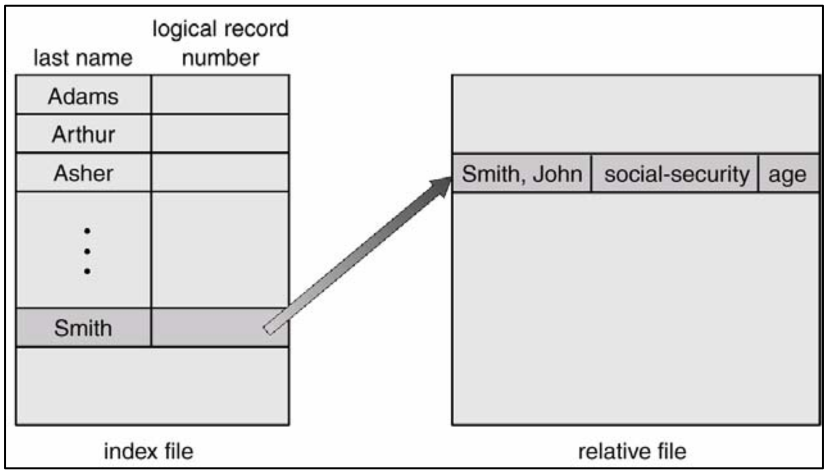

Figure 1. Example of index and relative files.
(Source: Silberschatz et al. 2002)

### 2.3.3 Directory structure

The file systems of computers can be extensive. Some systems store millions of files on terabytes of disk. To manage all these data, we need to organize them. This organization is usually done in two parts. First, disks are split into one or more **partitions**, also known as **volumes** in the PC. Second, each partition contains information about files within it. This information is kept in entries in a **device directory** or **volume table of contents**. The device directory (more commonly known as a directory) records the following information for all files on that partition (Figure 2).

- Name
- Location
- Size
- Type

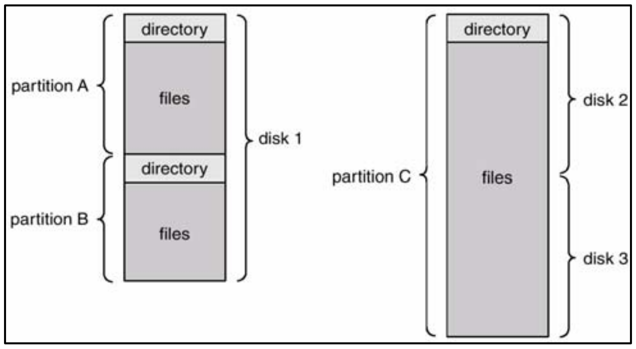

Figure 2. A typical file-system organization.
(Source: Silberschatz et al. 2002)

The following operation usually can be performed on a directory:

- Search a file: We need to be able to search a directory structure to find the entry for a particular file. We may want to be able to find all files whose names match a particular pattern.
- Create a file: New files need to be created and added into the directory.
- Delete a file: When a file is no longer needed, we want to remove it from the directory.
- List a directory: List all the files in this directory.
- Rename a file: Give a new name to the file.
- Traverse the file system: We may wish to access every directory, and every file within a directory structure.

In Windows system, we can easily perform the above operations on a directory. However, in MS-DOS, the related commands need to be used to perform the operations.

cd – change directory
dir – list files in a directory
del – delete a file
rename - rename the file

copy – copy files
format – format the disks

(1) Single-level directory

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand (Figure 3). However, the limitation to this structure is the difficulty of maintaining the files.

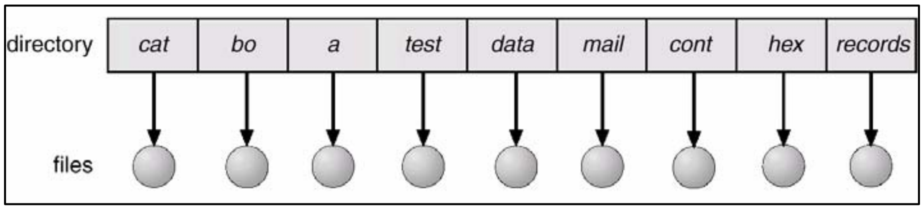

Figure 3. Single-level directory.
(Source: Silberschatz et al. 2002)

(2) Two-level directory

A single-level directory often leads to confusions of file names between users. The standard solution is to create a separate directory for each user (Figure 4). Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. This isolation is an advantage when the users are completely independent, but is a disadvantage when the users want to cooperate on some task and to access one another's files.

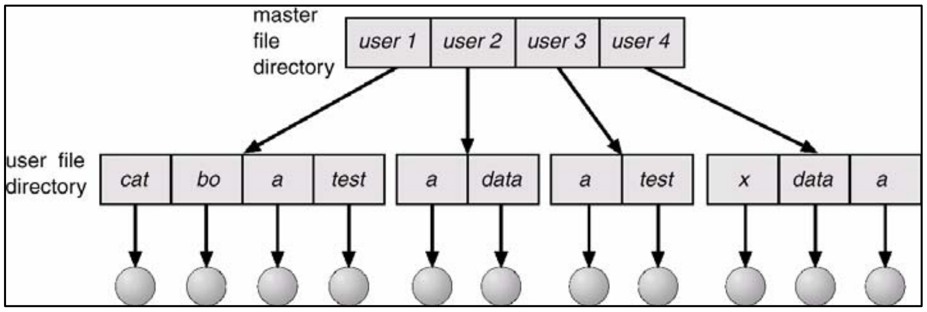

Figure 4. Two-level directory structure.
(Source: Silberschatz et al. 2002)

(3) Tree-structured directories

This structure allows users to create their own subdirectories and organize their files accordingly (Figure 5). The MS-DOS/Windows system, for example, is structured as a tree. In fact, a tree is the most common directory structure. The tree has a **root**

**directory.** Every file in the system has a unique **path name**. A path name is the path from the root, through all the subdirectories, to a specified file. A directory contains files and subdirectories.

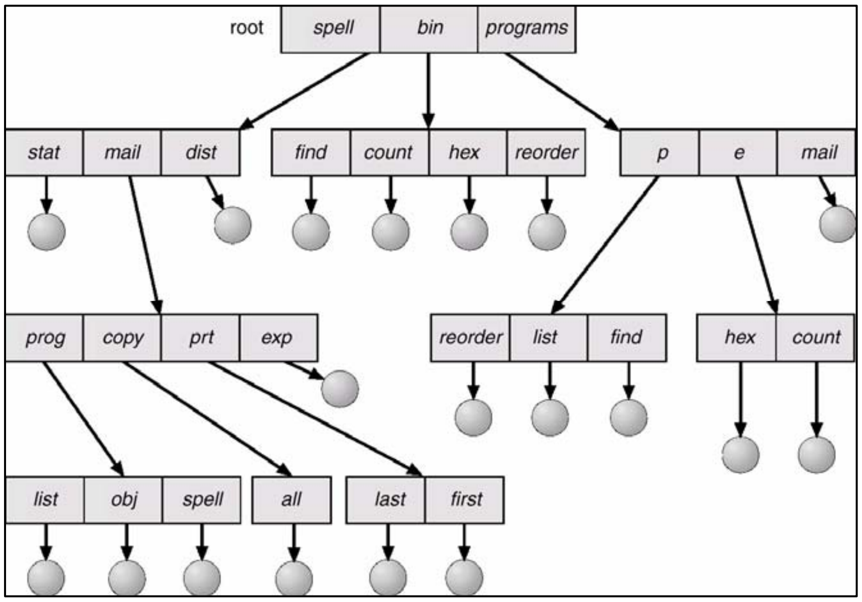

Figure 5. Hierarchical directory structure.
(Source: Silberschatz et al. 2002)

## 2.4 Programming Language Families

Existing programming languages can be classified into families based on their model of computation (Hayhurst 2002):

Imperative: are action-oriented languages: Pascal, C, and FORTRAN. The focus is on how the computer should perform its task. Computation is viewed as a sequence of actions. Instructions are viewed as performing actions on data stored in memory. A program is a series of steps each of which performs a calculation, retrieves input, or produces output. These languages encapsulate: procedural abstraction, assignments, loops, sequences, and conditional statements.

Functional Programming: computational model based on the recursive definition of functions (originated with LISP). A program is considered a function from inputs to outputs, defined in terms of simpler functions through a process of refinement. A program is a collection of mathematical functions each with an input (domain) and a result (range). Functions interact and combine with each other using functional composition, conditionals, and recursion: Lisp, Scheme.

Object oriented: relatively recent and can trace their roots to Simula 67. Closely related to imperative language. They have much more structure and a distributed model of both memory and computation. Rather that picture computation as the operation of a monolithic processor on a monolithic memory, OOL picture it as interactions among semi-independent objects each of which has both its own internal state and executable functions to manage that state. A program is viewed as a collection of objects that interact with one another by passing messages that transform an objects state. Object modeling, classification, inheritance, and information hiding are fundamental building blocks for OO languages: Ada 95, C++, and Java.

Logic programming: (constraint Based Programming) Inspiration from prepositional logic. The computation is performed as an attempt to find values that satisfy certain specified relationships using goal directed search through a list of logical rules. Attempts to use logical reasoning to answer queries. A program is a collection of logical declarations about what outcome a function should accomplish rather than how that outcome should be accomplished. Execution of the program applies these declarations to achieve a series of possible solutions to a problem, such as Prolog.

(1) Compilation vs. interpretation

There are 2 basic approaches to implementing a program in a higher-level language:

- The language is brought down or converted to the level of the machine using a translator called a compiler.
- The Machine is brought up to the level of the language, building a higher level machine (virtual machine) which can run the language directly: interpreter

At the highest level of abstraction, the compilation and execution of a program looks like (Figure 6):

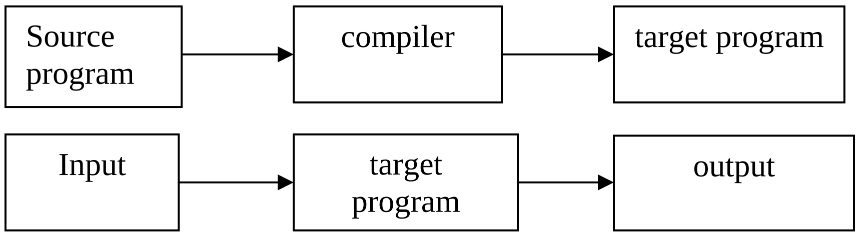

Figure 6. Compilation.

Unlike a compiler, an interpreter stays around during execution, and is the focus of control during the execution (Figure 7). Interpreters implement a virtual machine, whose machine language is the high-level programming language, the interpreter reads statements one at a time, verifying and executing them as it goes along.

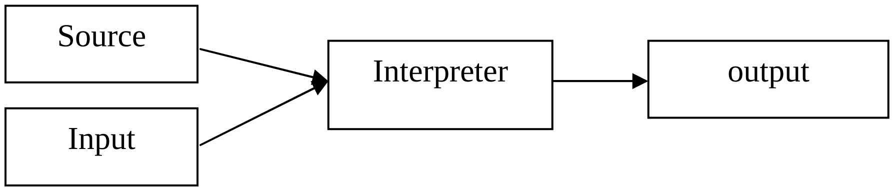

Figure 7. Interpretation.

(2) C/C++ compilers

Compilers for C and many other languages running on UNIX begin with a preprocessor that removes comments, and expands macros, such as #include. The preprocessor can also be asked to delete portions of the code providing conditional compilation: #if, #ifdef, #ifndef (Figure 8).

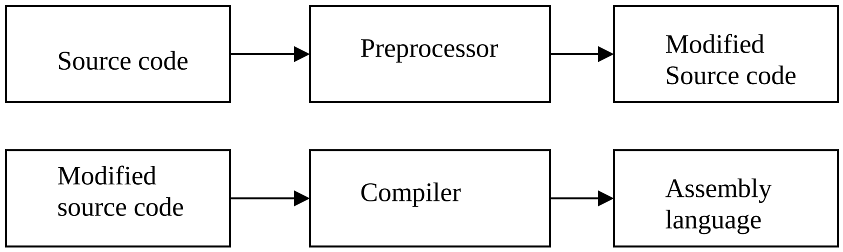

Figure 8. C compiler.

C++ compilers based on the early AT&T compiler actually generate an intermediate program in C instead of assembly language (Figure 9).

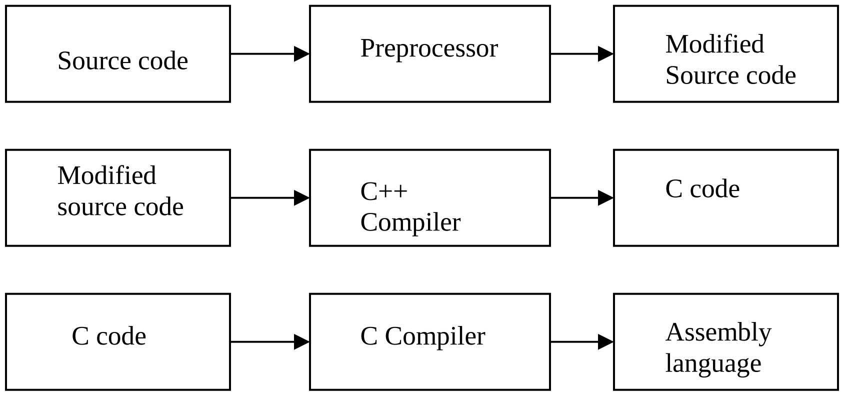

Figure 9. C++ compiler.

The C++ compiler is a true compiler and performs a complete analysis of the syntax and semantics of the C++ source program, and with very few exceptions generates all of the error messages that a programmer will see prior to running the program. Many programmers are generally unaware that the C compiler is being used behind the scenes.

The C++ compiler doesn't invoke the C compiler unless it can generate C code that should pass through the second round of compilation without producing any error messages.

## References

Lane, M. and J. Mooney. 2001. A Practical Approach to Operating Systems (Lecture Notes). Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV.

Hayhurst, C. 2002. Semantics of Programming Language (Lecture Notes). Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, West Virginia.

Silberschatz, A., P.B. Galvin, and G. Gagne. 2002. Operating System Concepts (6th Edition). John Wiley & Sons, Inc., New York.